CMOS VLSI Design

PROJECT REPORT - 2

Given 8 8-bit unsigned numbers, write Verilog code to sort the numbers in descending order, and output the sorted numbers. Report the timing and power when synthesized in 180 nm.

Satwik Kamath (1MS21EC098)
Shreesha T P (1MS21EC102)
Shreeya R (1MS21EC103)
Shreya P Manchala (1MS21EC105)

Submitted as part of CIE for EC53 (2023-2024)

Contents

1 Bubble Sort

1.1 Flowchart

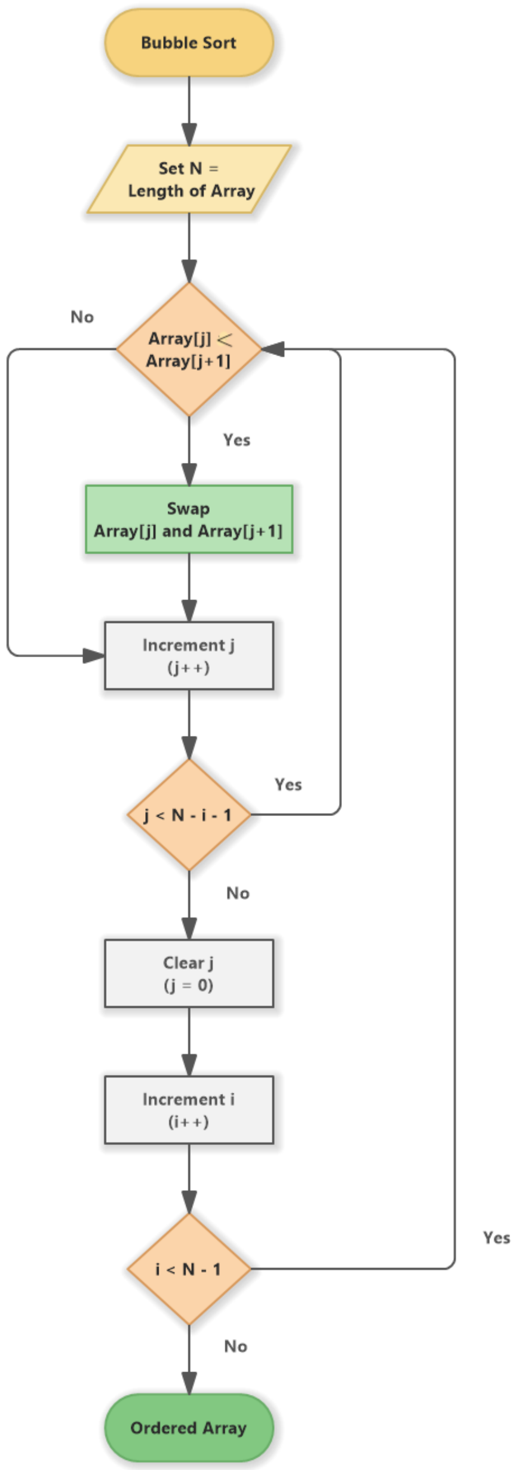


Figure 1: Bubble Sort

1.2 Theory

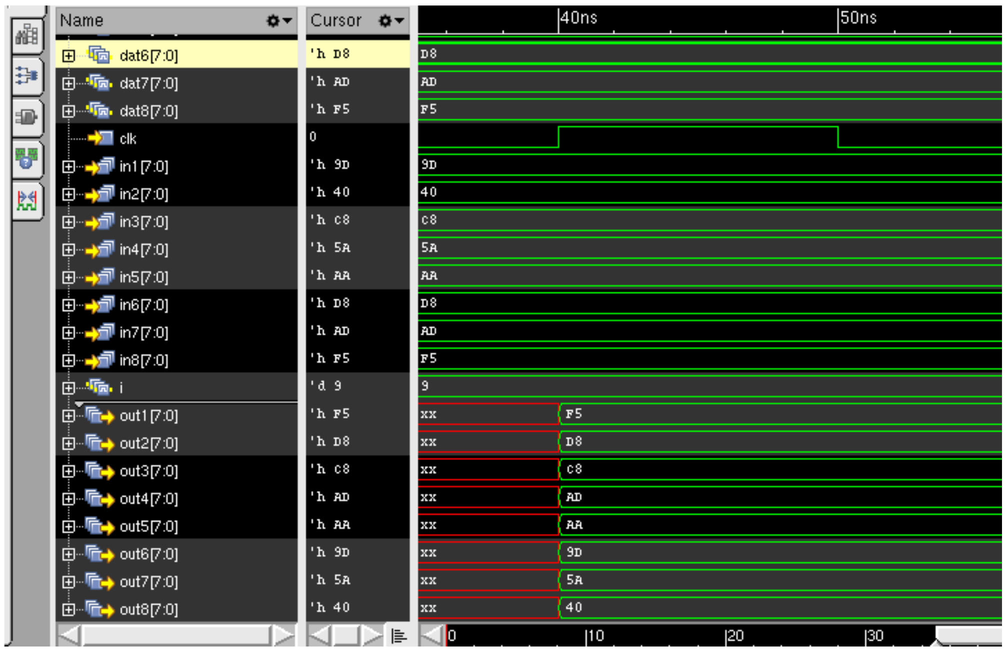
Bubble sort, a hardware-optimized algorithm, iteratively compares adjacent elements in a sequence, swapping if the latter is smaller. This process is repeated for n-1 iterations, systematically traversing the list. Despite its simplicity, bubble sort's practical performance diminishes for larger datasets, showcasing an O(n$\hat{2}$) complexity in the worst-case scenario. In hardware, its distinct characteristics are notable, yet limitations persist. The algorithm's flowchart visually outlines the systematic steps for its hardware implementation. While suitable for educational purposes or small datasets, its inefficiency in larger scenarios makes it less preferable compared to more advanced sorting algorithms in practical applications.

1.3 Code For Bubble Sort

```
module sort (
  input  wire clk,
  input  wire [7:0] in1, in2, in3, in4, in5, in6, in7,in8,
  output reg  [7:0] out1, out2, out3, out4, out5, out6,out7,out8
  );

  reg [7:0] dat1, dat2, dat3, dat4, dat5, dat6,dat7,dat8;
  always @(posedge clk)
  begin
      dat1 <= in1;
      dat2 <= in2;
      dat3 <= in3;
      dat4 <= in4;
      dat5 <= in5;
      dat6 <= in6;
      dat7 <= in7;
      dat8 <= in8;
  end
    integer i, j;
    reg [7:0] temp;
    reg [7:0] array [1:8];
 always @*
    begin
        array[1] = dat1;
        array[2] = dat2;
        array[3] = dat3;
        array[4] = dat4;
        array[5] = dat5;
        array[6] = dat6;
        array[7] = dat7;
        array[8] = dat8;
  for (i = 8; i > 0; i = i - 1)
    begin
    for (j = 1 ; j < i; j = j + 1)
        begin
        if (array[j] < array[j + 1])
          begin
            temp = array[j];
            array[j] = array[j + 1];
            array[j + 1] = temp;
          end
        end
    end
  end
    always @(posedge clk)
    begin
      out1 <= array[1];
      out2 <= array[2];
      out3 <= array[3];
      out4 <= array[4];
      out5 <= array[5];
      out6 <= array[6];
      out7 <= array[7];
      out8 <= array[8];
  end
endmodule
```

1.4 Simulation Using NCLaunch



Name	Cursor
dat6[7:0]	'h D8
dat7[7:0]	'h AD
dat8[7:0]	'h F5
clk	0
in1[7:0]	'h 9D
in2[7:0]	'h 40
in3[7:0]	'h C8
in4[7:0]	'h 5A
in5[7:0]	'h AA
in6[7:0]	'h D8
in7[7:0]	'h AD
in8[7:0]	'h F5
i	'd 9
out1[7:0]	'h F5
out2[7:0]	'h D8
out3[7:0]	'h C8
out4[7:0]	'h AD
out5[7:0]	'h AA
out6[7:0]	'h 9D
out7[7:0]	'h 5A
out8[7:0]	'h 40

Figure 2: Output

1.5 Schematic Generation Using Genus

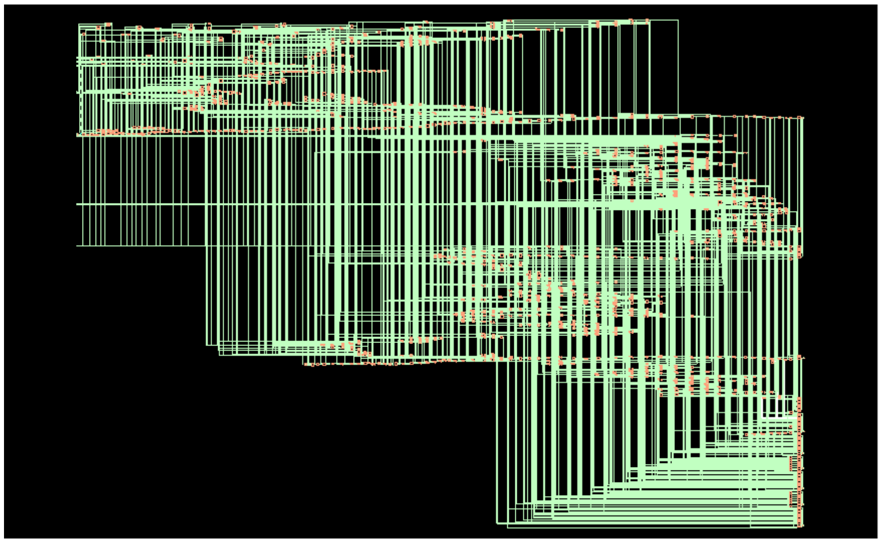


Figure 3: Schematic

1.6 Power Report

The following content is generated using 180nm technology in **Genus**

```
==========================================================================
  Generated by:            Genus(TM) Synthesis Solution 17.22-s017_1
  Generated on:            Dec 27 2023  05:42:55 am
  Module:                  sort
  Operating conditions:    typical (balanced_tree)
  Wireload mode:           enclosed
  Area mode:               timing library
==========================================================================

                    Leakage      Dynamic       Total
Instance  Cells  Power(nW)   Power(nW)     Power(nW)
------------------------------------------------------------
sort       1072     80.400  2241236.316  2241316.715
```

1.7 Timing Report

The following content is generated using 180nm technology in **Genus**

```
==========================================================================
  Generated by:            Genus(TM) Synthesis Solution 17.22-s017_1
  Generated on:            Dec 27 2023  05:42:31 am
  Module:                  sort
  Operating conditions:    typical (balanced_tree)
  Wireload mode:           enclosed
  Area mode:               timing library
==========================================================================


Path 1: UNCONSTRAINED Setup Check with Pin out2_reg[1]/CK->SE
     Startpoint: (R) dat1_reg[1]/CK
       Endpoint: (R) out2_reg[1]/SE


          Setup:-      392
      Data Path:-    24836
```

2 Selection Sort

2.1 Flowchart

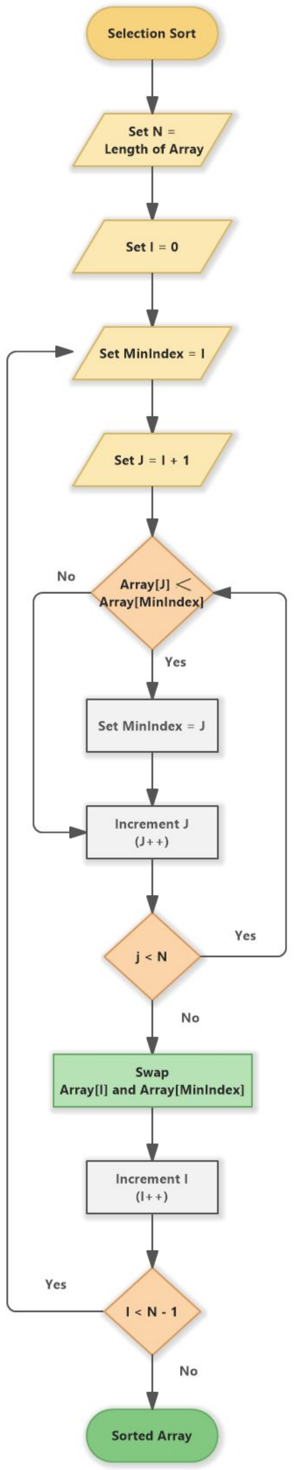


Figure 4: Selection Sort

2.2 Theory

The selection sort algorithm operates by sorting an array of numbers through a methodical process. It consistently identifies the smallest element from the unsorted portion of the array and places it in the first position. This method involves maintaining two distinct subarrays within the array: one that is already sorted and another that remains unsorted. The procedure is described as follows: The smallest element in the array is located - it is swapped with the element in the first position - the second smallest element is then located and it is interchanged with the element in the second position. This process continues until the entire array is sorted.

The algorithm's time complexity remains consistent across worst, average, and best-case scenarios, standing at O(n$\hat{2}$). This is akin to bubble sort, as the runtime is primarily contingent on the degree of order within the dataset. Despite sharing this time complexity, the selection sort algorithm proves to be an efficient sorting technique in situations where the order of elements is unpredictable.

2.3 Code for Selection Sort

```
module sort (
  input  wire clk,
  input  wire [7:0] in1, in2, in3, in4, in5,in6,in7,in8,
  output reg  [7:0] out1, out2, out3, out4, out5,out6,out7,out8
  );
  reg [7:0] dat1, dat2, dat3, dat4, dat5,dat6,dat7,dat8;
  always @(posedge clk)
  begin
      dat1 <= in1;
      dat2 <= in2;
      dat3 <= in3;
      dat4 <= in4;
      dat5 <= in5;
      dat6 <= in6;
      dat7 <= in7;
      dat8 <= in8;
  end
    integer i, j,max;
    reg [7:0] temp;
    reg [7:0] array [1:8];
    always @*
    begin
        array[1] = dat1;
        array[2] = dat2;
        array[3] = dat3;
        array[4] = dat4;
        array[5] = dat5;
        array[6] = dat6;
        array[7] = dat7;
        array[8]  =dat8;
    for (i = 1; i <=9; i=i+1)
    begin
          max=i;
        for (j = i+1 ; j <=8; j = j + 1)
        begin
          if (array[j] > array[max])
                begin
                 max=j;
                end
           end
            if(max!=i)
          begin
            temp = array[max];
            array[max] = array[i];
            array[i] = temp;
          end
        end
    end
    always @(posedge clk)
    begin
      out1 <= array[1];
      out2 <= array[2];
      out3 <= array[3];
      out4 <= array[4];
      out5 <= array[5];
      out6 <= array[6];
      out7 <= array[7];
      out8 <= array[8];
    end
endmodule
```

2.4 Simulation Using NCLaunch

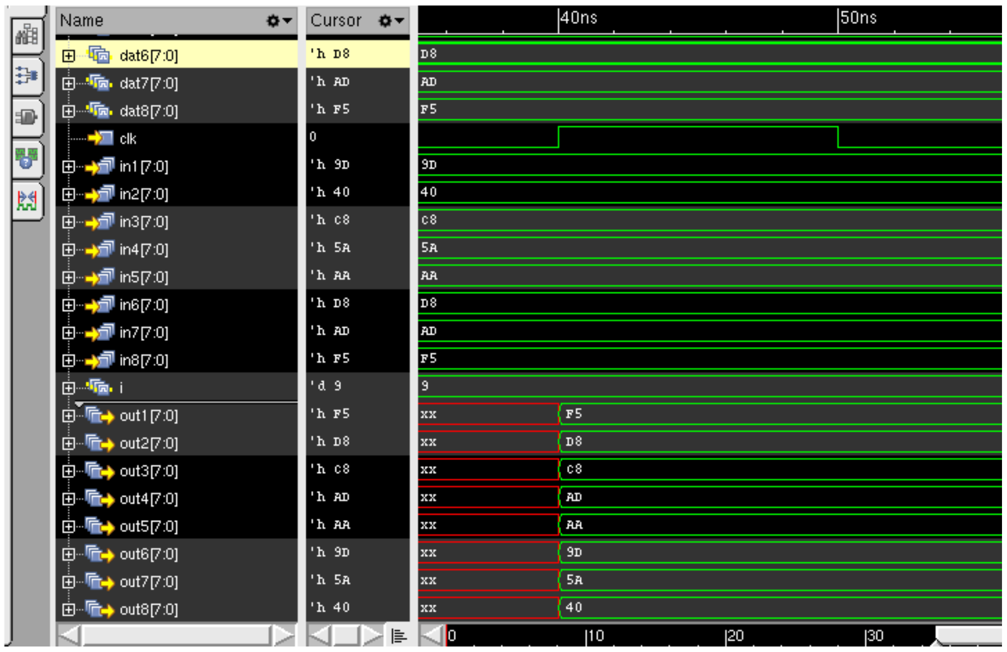


Figure 5: Output

2.5 Schematic Generation Using Genus

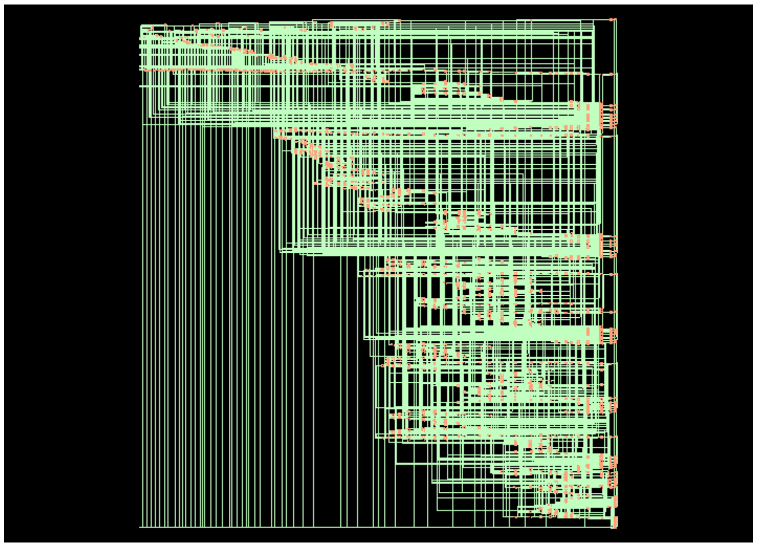


Figure 6: Schematic

2.6 Power Report

The following content is generated using 180nm technology in **Genus**

```
==========================================================================
  Generated by:            Genus(TM) Synthesis Solution 17.22-s017_1
  Generated on:            Jan 06 2024  11:35:16 am
  Module:                  sort
  Operating conditions:    typical (balanced_tree)
  Wireload mode:           enclosed
  Area mode:               timing library
==========================================================================

                   Leakage      Dynamic        Total
Instance  Cells  Power(nW)   Power(nW)     Power(nW)
-------------------------------------------------------------
sort       1397     91.529  3000982.030  3001073.560
```

2.7 Timing Report

The following content is generated using 180nm technology in **Genus**

```
==========================================================================
  Generated by:            Genus(TM) Synthesis Solution 17.22-s017_1
  Generated on:            Jan 06 2024  11:35:03 am
  Module:                  sort
  Operating conditions:    typical (balanced_tree)
  Wireload mode:           enclosed
  Area mode:               timing library
==========================================================================


Path 1: UNCONSTRAINED Setup Check with Pin out8_reg[1]/CK->SE
     Startpoint: (R) dat1_reg[1]/CK
       Endpoint: (R) out8_reg[1]/SE


          Setup:-      410
      Data Path:-    58511
```

3 Insertion Sort

3.1 Flowchart

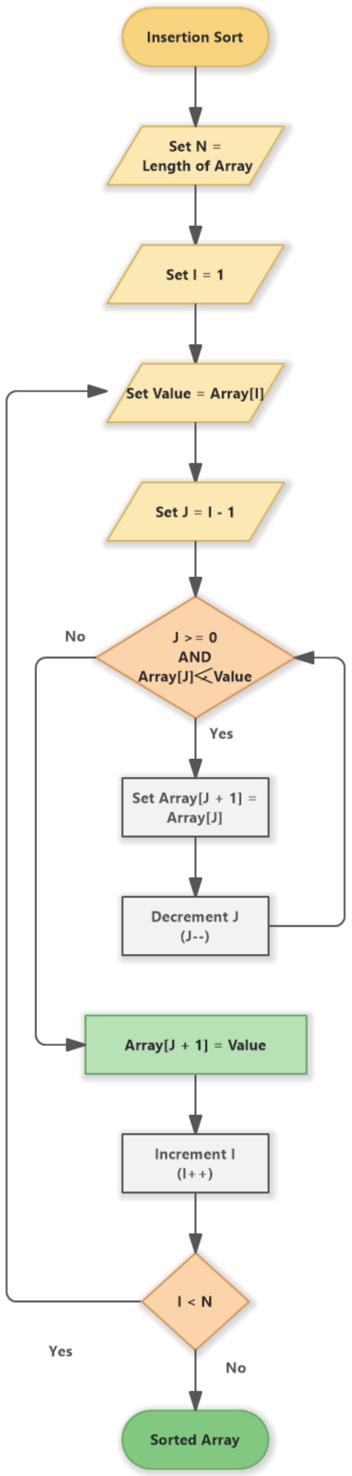


Figure 7: Insertion Sort

3.2 Theory

In this algorithm, each iteration involves choosing one element from the input data. The algorithm then recognizes the suitable location within the pre- sorted list and inserts the selected element into its fitting position. The insertion sort being a comparison-based, in-place sorting algorithm perpetuates a sub-list that is steadily sorted, typically protecting the lower part of the array. Systematic insertion of an element into the already sorted sub list is the quintessence of insertion sort

The O(n$\hat{2}$) worst-case performance of this algorithm is portrayed, particularly when the input data is presented in reverse order, entailing a great number of comparisons or swaps. Regardless of this, the insertion sort algorithm boasts an O(1) worst-case space complexity. The average case performance falls in line with the worst case, making the insertion sort algorithm less practical for performing the sorting of larger arrays. However, the efficiency of this algorithm in sorting the smaller arrays is high, surpassing the other sorting algorithms.

3.3 Code For Insertion Sort

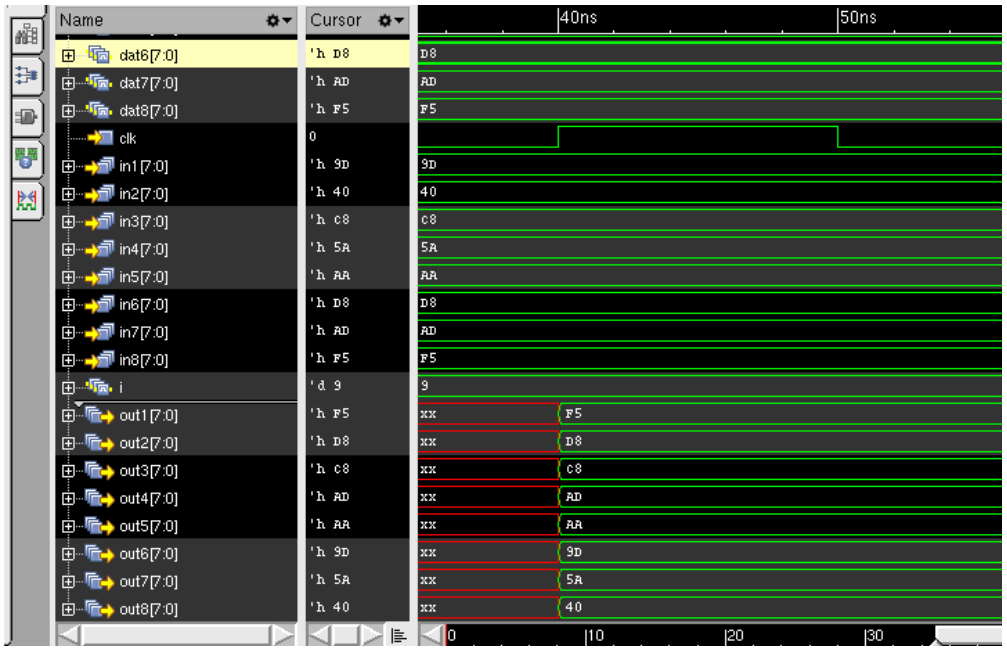
```
module ins (
  input  wire clk,
  input  wire [7:0] in1, in2, in3, in4, in5,in6,in7,in8,
  output reg  [7:0] out1, out2, out3, out4, out5,out6,out7,out8
  );
  reg [7:0] dat1, dat2, dat3, dat4, dat5,dat6,dat7,dat8;

  always @(posedge clk)
  begin
      dat1 <= in1;
      dat2 <= in2;
      dat3 <= in3;
      dat4 <= in4;
      dat5 <= in5;
      dat6 <= in6;
      dat7 <= in7;
      dat8 <= in8;
  end
    integer i, j=0;
    reg [7:0] cur;
    reg [7:0] array [1:8];
    always @*
    begin
        array[1] = dat1;
        array[2] = dat2;
        array[3] = dat3;
        array[4] = dat4;
        array[5] = dat5;
        array[6] = dat6;
        array[7] = dat7;
        array[8] = dat8;

        for (i=2; i<9; i=i+1)
        begin
            cur=array[i];
            j=i-1;

            while(j>=1 && cur>array[j])
            begin
                array[j+1]=array[j];
                j=j-1;
            end
            array[j+1]=cur;
        end
    end
    always @(posedge clk)
    begin
      out1 <= array[1];
      out2 <= array[2];
      out3 <= array[3];
      out4 <= array[4];
      out5 <= array[5];
      out6 <= array[6];
      out7 <= array[7];
      out8 <= array[8];
    end
endmodule
```

3.4 Simulation Using NCLaunch



Name	Cursor
dat6[7:0]	'h D8
dat7[7:0]	'h AD
dat8[7:0]	'h F5
clk	0
in1[7:0]	'h 9D
in2[7:0]	'h 40
in3[7:0]	'h C8
in4[7:0]	'h 5A
in5[7:0]	'h AA
in6[7:0]	'h D8
in7[7:0]	'h AD
in8[7:0]	'h F5
i	'd 9
out1[7:0]	'h F5
out2[7:0]	'h D8
out3[7:0]	'h C8
out4[7:0]	'h AD
out5[7:0]	'h AA
out6[7:0]	'h 9D
out7[7:0]	'h 5A
out8[7:0]	'h 40

Figure 8: Output

3.5 Schematic Generation Using Genus

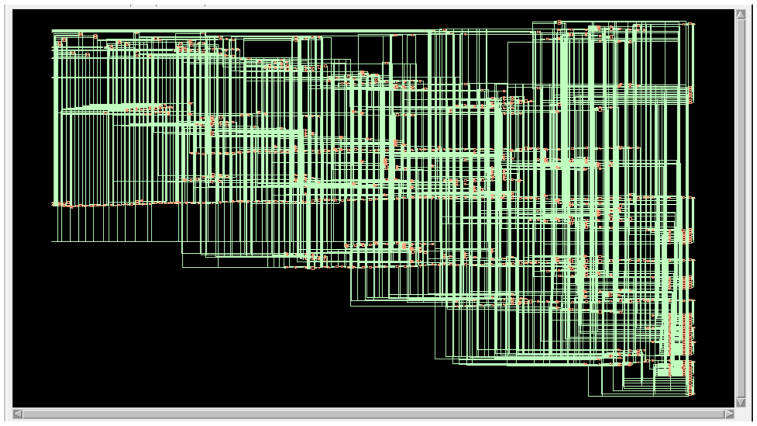


Figure 9: Schematic

3.6 Power Report

The following content is generated using 180nm technology in **Genus**

```
==========================================================================
  Generated by:             Genus(TM) Synthesis Solution 17.22-s017_1
  Generated on:             Jan 06 2024  11:23:27 am
  Module:                   ins
  Operating conditions:     typical (balanced_tree)
  Wireload mode:            enclosed
  Area mode:                timing library
==========================================================================

                   Leakage      Dynamic        Total
Instance  Cells  Power(nW)   Power(nW)     Power(nW)
-------------------------------------------------------
ins         911     70.751  1855662.325  1855733.077
```

3.7 Timing Report

The following content is generated using 180nm technology in **Genus**

```
==========================================================================
  Generated by:             Genus(TM) Synthesis Solution 17.22-s017_1
  Generated on:             Jan 06 2024  11:22:57 am
  Module:                   ins
  Operating conditions:     typical (balanced_tree)
  Wireload mode:            enclosed
  Area mode:                timing library
==========================================================================


Path 1: UNCONSTRAINED Setup Check with Pin out2_reg[6]/CK->D
     Startpoint: (R) dat2_reg[0]/CK
       Endpoint: (R) out2_reg[6]/D


          Setup:-        97
      Data Path:-     15678
```

4 Comparison

Sorting Algorithm	Time	Power
Bubble Sort	24.83 ns	2.2 mW
Selection Sort	58.51 ns	3 mW
Insertion Sort	15.67 ns	1.85 mW

Table 1: Comparison

5 Results

Based on the Comparison table, here are some insights:

- Insertion Sort
  - Execution Time (15.67 ns) Insertion Sort demonstrates the fastest execution time among the three algorithms. This means that, on average, it takes 15.67 nanoseconds to sort the given data. The efficiency of Insertion Sort is particularly notable when dealing with small to moderately sized datasets.
  - Power Consumption (1.85 mW) In addition to its quick execution time, Insertion Sort also exhibits the lowest power consumption at 1.85 watts. Lower power consumption is advantageous, especially in scenarios where energy efficiency is a critical consideration, such as in battery-powered devices or energy-conscious environments.
- Bubble Sort
  - Execution Time (24.83 ns) Bubble Sort takes a bit longer to execute compared to Insertion Sort. The execution time of 24.83 nanoseconds suggests that, on average, it requires more time to sort the given dataset. While it might be faster than Selection Sort, it is surpassed by Insertion Sort in terms of efficiency.
  - Power Consumption (2.2 mW) Bubble Sort consumes more power than Insertion Sort but is still more power-efficient than Selection Sort. The power consumption of 2.2 milliwatts indicates the amount of power the algorithm requires during its execution.
- Selection Sort
  - Execution Time (58.51 ns) Bubble Sort takes a bit longer to execute compared to Insertion Sort. The execution time of 24.83 nanoseconds suggests that, on average, it requires more time to sort the given dataset. While it might be faster than Selection Sort, it is surpassed by Insertion Sort in terms of efficiency.
  - Power Consumption (3 mW) Bubble Sort consumes more power than Insertion Sort but is still more power-efficient than Selection Sort. The power consumption of 2.2 milliwatts indicates the amount of power the algorithm requires during its execution.

6 Conclusion

In this project, three common sorting algorithms are implemented in verilog. The RTL diagram, timing report, power report and comparative analysis are discussed. From our point of view, insertion sort algorithm shows much faster operation when implemented in hardware.Despite the lesser efficiencies of these algorithms when large data is encountered, if the data is divided into several parts and proceeded into several blocks (designing several modules of sorter in a large module) for sorting then it can be a faster operation.