EE518 LAB
Experiment 8

Design a Verilog module to sort a M length array using quick sort

Submitted by,
Monish Nath
Roll No-224102409
April 9, 2023

Contents

List of Figures

List of Tables

1 Objective :

Design a Verilog module to sort a M length array with
1. N bit fixed point elements using quick sort
2. IEEE floating point elements using quick sort.

2 Theory :

QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
1. Always pick the first element as a pivot.
2. Always pick the last element as a pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

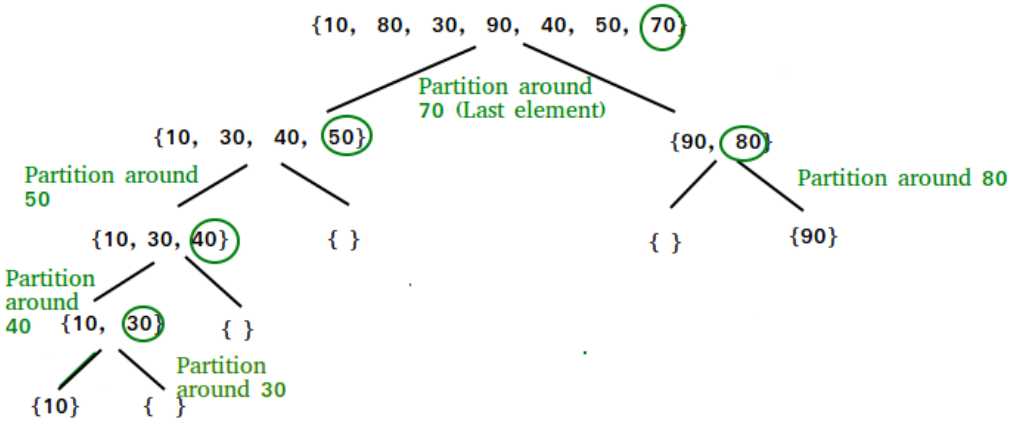

Figure 1: Quick Sort Working

3 Design Approach :

Here we have made a design using FSM and all the comparison are done is separate states. We have implemented a stack for the index storage and a sorter module which runs a for loop for partition index. As all the operations are done in different clock cycles it takes a lot of clock cycles to run all the iterations and give the final output.

4 Verilog Code :

```
module sort_flt
    #(parameter M=8, N=32)
        (a_in, clock, rst, outP, outvalid);

    input [N-1:0] a_in;
    input clock, rst;
    output reg [N-1:0] outP;
    output reg outvalid;

    parameter S0 = 4'd0, S1 = 4'd1, S2 = 4'd2, S3 = 4'd3,
    S4 = 4'd4, S5 = 4'd5, S6 = 4'd6, S7 = 4'd7, S8 = 4'd8,
    S9 = 4'd9;

    reg [3:0] next_state, curr_state;
    reg [5:0] i, j, i_srt, j_srt, i_stk;
    reg [N-1:0] arr [M-1:0];
    reg [5:0] stk1 [M-1:0];
    reg [5:0] stk2 [M-1:0];
    reg push, pop, done, start;
    reg empty;
    reg [5:0] indxp1, indxp2, ind1, ind2, pi;
    wire [N-1:0] pivot;
    wire comp_flt;

    always@(posedge clock) begin
        if(rst) curr_state <= S0;
        else curr_state <= next_state;
```

```
end

always@(curr_state, a_in) begin
    case(curr_state)
    S0:   begin next_state <= S1;
                i <= 'b0;
                j <= 'b0;
                i_stk <= 'b0;
                outP <= 'b0;
                outvalid <= 'b0;
          end
    S1:   begin push <= 'b0;
                arr[i] <= a_in;
                i <= i + 'b1;
                next_state <= (i==M-1) ? S2 : S1;
                outP <= 'b0;
             end
    S2:  begin
                indxp1 <= 'b0; //for starting sort
                indxp2 <= M-1;
                push <= 'b1;
                next_state <= S3;
         end
    S3:  begin
            push <= 'b0;
            pop <= 'b1;
            start <= 'b0;
            next_state <= S4;
         end
    S4:  begin
            pop <= 'b0;
            if(empty=='b1)
            next_state <= S8;
            else if($signed(ind1) >= $signed(ind2))
            next_state <= S3;
            else begin
            //start <= 'b1;
            next_state <= (done=='b1) ? S6 : S5;
```

```
            end
        end
    S5 : begin
            start <= 'b1;
            next_state <= (done=='b1) ? S6 : S4;
         end
    S6: begin
            indxp1 <= ind1;
            indxp2 <= pi-1;
            push <= 'b1;
            next_state <= S7;
        end
    S7: begin
            indxp1 <= pi+1;
            indxp2 <= ind2;
            push <= 'b1;
            next_state <= S3;
        end
    S8: begin
            outP <= arr[j];
            j <= j + 1;
            outvalid <= 'b1;
            next_state <= (j==M-1) ? S0 : S9;
        end
    S9: begin
            outP <= arr[j];
            j <= j + 1;
            outvalid <= 'b1;
            next_state <= (j==M-1) ? S0 : S8;
        end
   default: begin next_state <= S0;
               i <= 'b0;
               outP <= 'b0;
            end
   endcase
end

//SORTER
```

```
assign pivot = arr[ind2];
always@(posedge clock) begin
if(start == 'b0) begin
j_srt <= ind1;
i_srt <= ind1;
done <= 'b0; end
else if((start == 'b1) && (done == 'b0))  begin
if((comp_flt)&(j_srt<ind2)
    /*$signed(arr[j_srt]) < $signed(pivot)*/) begin
arr[i_srt] <= arr[j_srt];
arr[j_srt] <= arr[i_srt];
i_srt <= i_srt + 'b1; end
else if(j_srt == ind2) begin
arr[i_srt] <= pivot;
arr[ind2] <= arr[i_srt];
pi <= i_srt;
done <= 'b1; end
j_srt <= j_srt + 'b1;
end
else begin
j_srt <= 'b0;
i_srt <= 'b0;
done <= 'b0; end
end

//floating number compare
assign comp_flt = (pivot[N-1]>arr[j_srt%M][N-1]) ? 'b0 :
                    (pivot[N-1]<arr[j_srt%M][N-1]) ? 'b1 :
                      (pivot[N-2:0]<arr[j_srt%M][N-2:0]) ?
                      arr[j_srt%M][N-1] : ~arr[j_srt%M][N-1];

//STACK
always@(posedge clock) begin
if(push == 'b1) begin
stk1[i_stk] <= indxp1;
stk2[i_stk] <= indxp2;
i_stk <= i_stk + 'b1; end
else if((pop == 'b1) & (i_stk != 'b0)) begin
```

```
    ind1 <= stk1[i_stk-1];
    ind2 <= stk2[i_stk-1];
    i_stk <= i_stk - 'b1; end
    else empty = (pop == 'b1) & (i_stk == 'b0);
    end

endmodule
```

Note : The code will be almost similar for fixed point just the comparison logic will be changed. We can directly compare the fixed point numbers using signed function. Hence not writing a separate code for the same.

5 Test bench ;

```
module sort_flt_tb();

    reg clock, rst;
    reg [31:0] a;
    wire [31:0] outP;    wire outvalid;

    always #50 clock = ~clock;

    sort_fixed DUT (a, clock, rst, outP, outvalid);

    initial begin
    clock = 1; rst = 1;
    #100 rst = 0;
     a = 32'b0_10000001_00000000000000000000000;
    #100 a = 32'b0_10000001_01000000000000000000000;
    #100 a = 32'b0_10000000_10000000000000000000000;
    #100 a = 32'b1_10000010_01000000000000000000000;
    #100 a = 32'b0_10000000_01010000000000000000000;
    #100 a = 32'b0_10000001_11000000000000000000000;
    #100 a = 32'b1_10000001_00000000000000000000000;
    #100 a = 32'b0_10000000_00000000000000000000000;
    end

endmodule
```

6 RTL Schematic :

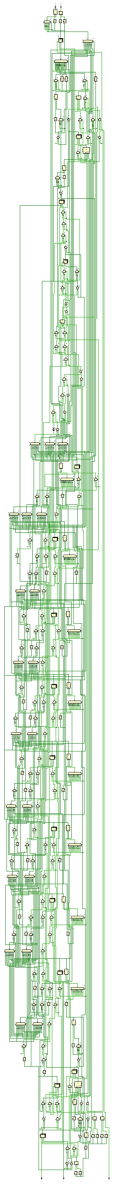

Figure 2: Schematic of quick sort circuit

7 Simulation :

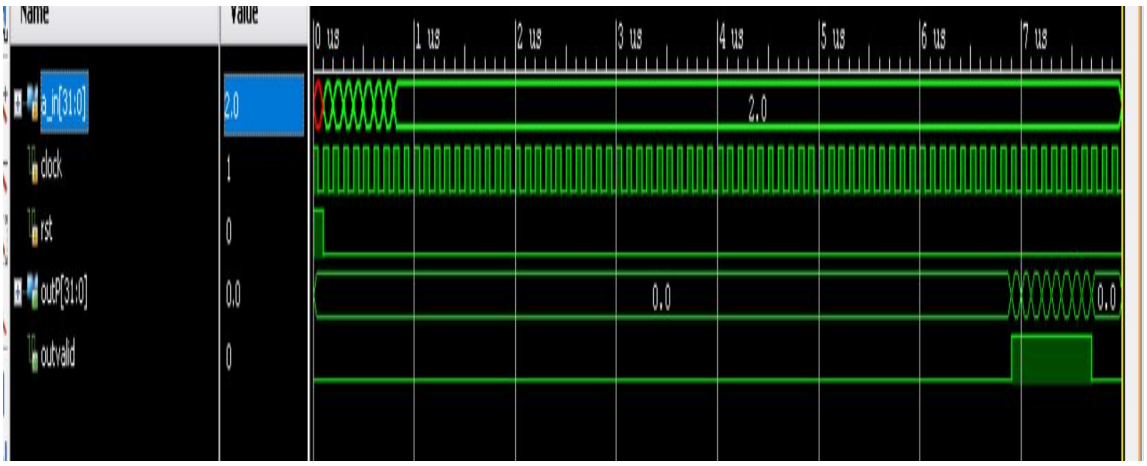

Figure 3: Behavioural simulation waveform

Figure 4: Behavioural simulation waveform for inputs

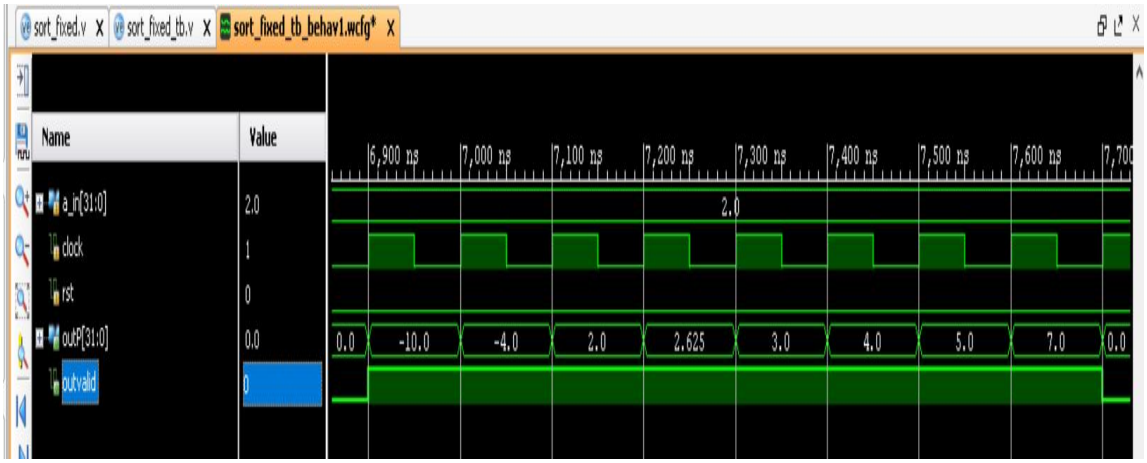

Figure 5: Behavioural simulation waveform for outputs

Note: The representation of the input and output are kept in single precision floating point representation.

8 Synthesis Reports :

The values have been found after synthesis of the corresponding designs. I have done the experiment on Vivado 2014.1. The FPGA board selected is Artix-7. The LUTs and Flops have been found from the utilization report. The delay has been found from the timing report and the power has been found from the power report.

We have added proper constraints and the synthesis results are shown below.

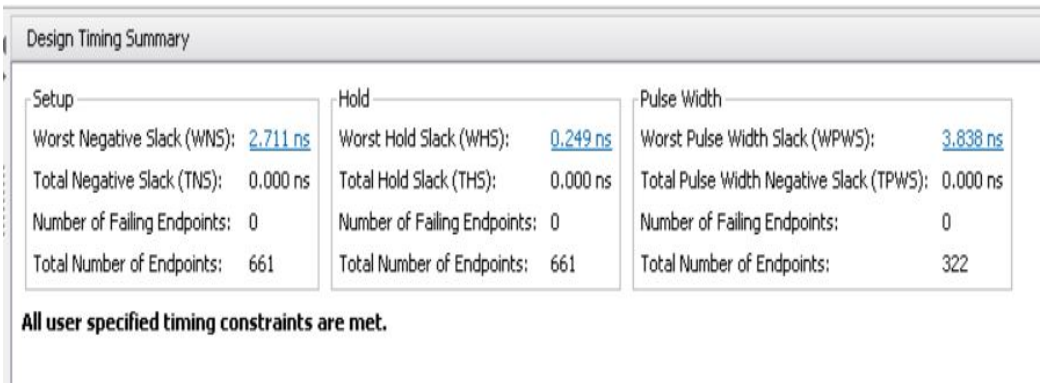

Figure 6: Timing summary

Site Type	Used	Loced	Available	Util%
Slice LUTs*	1054	0	134600	0.78
LUT as Logic	1046	0	134600	0.77
LUT as Memory	8	0	46200	0.01
LUT as Distributed RAM	8	0		
LUT as Shift Register	0	0		
Slice Registers	627	0	269200	0.23
Register as Flip Flop	305	0	269200	0.11
Register as Latch	322	0	269200	0.11
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

* Warning! The Final LUT count, after physical optimizations and full in

Figure 7: Utilization report

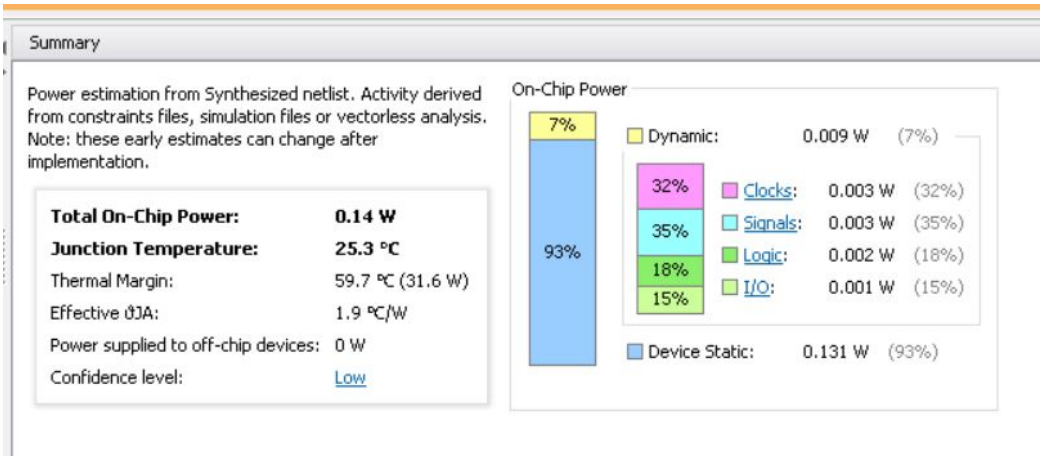

Figure 8: Power report

9 Conclusions :

- We have designed the circuit as per the requirement.
- The functionality of our design have been verified . The functionality are showing as expected.
- The different parameters of the design such as LUTs, delay and power have been calculated from the synthesis and tabulated.