EE518 LAB
Experiment 6

Design and implement a matrix convolution circuit using Verilog

Submitted by,

Monish Nath

Roll No-224102409

February 26, 2023

Contents

List of Figures

List of Tables

1 Objective :

Design and implement Verilog modules to:
a. Perform matrix convolution between two matrices of size nxn and mxm (n>m) using 2's complement fixed point representation.
b. Perform matrix convolution between two matrices of size nxn and mxm (n>m) using IEEE754 format.

2 Theory :

Convolution is the treatment of a matrix by another one which is called "kernel".A convolution is a type of matrix operation, consisting of a kernel, a small matrix of weights, that slides over input data performing element-wise multiplication with the part of the input it is on, then summing the results into an output.

Intuitively, a convolution allows for weight sharing - reducing the number of effective parameters - and image translation (allowing for the same feature to be detected in different parts of the input space).

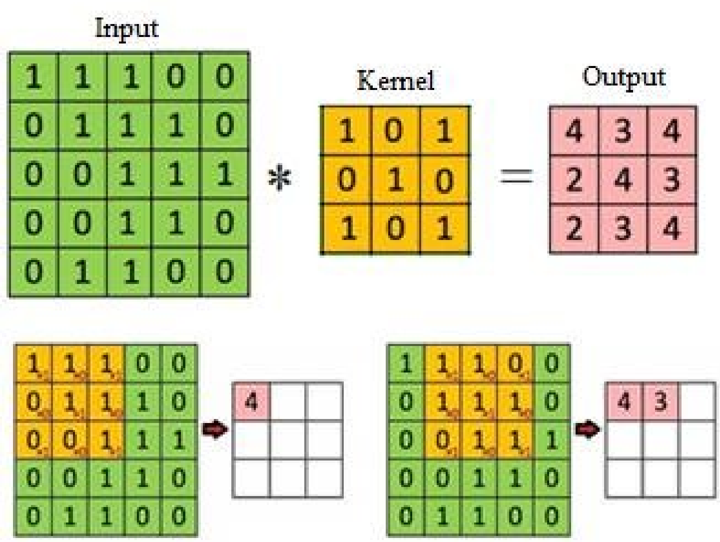

Figure 1: Basic Convolution Operation

3 Matrix Convolution with 2's complement representation :

3.1 Design Approach :

The design only has one input which accepts each input at the positive edge of clock. So, both the matrix elements are inserted to the design one by one. Initially the kernel inputs are send which is a mxm matrix (m=2) so first m^2 (4) elements are stored in a shift register which is our kernel. Now the main matrix input are fed one by one. We have such an arrangement of shift registers that we get a m*m window of the main n*n matrix at any instant of time and the convolution output is produced directly. The figure gives a rough representation of our algorithm. The output signals out_valid and end_conv indicate us which output values are correct and when the operation has ended.

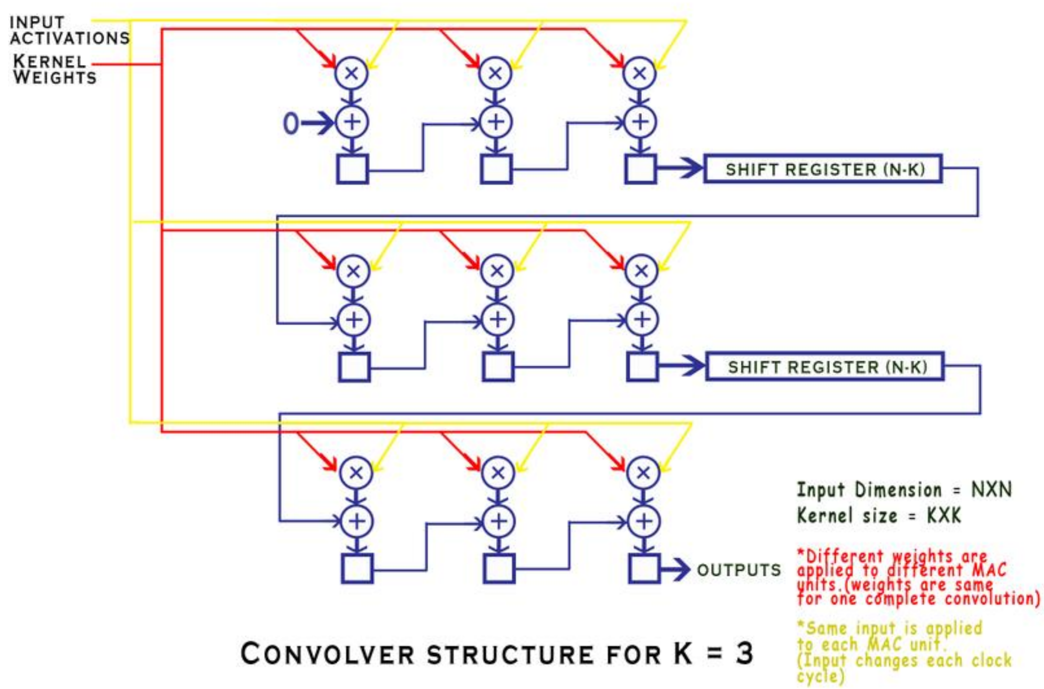

Figure 2: Convolution Algorithm Schematic

3.2 Verilog Code :

```
module conv_fix(
    input clock,
    input rst,
    input [31:0] a_in,
    output [31:0] result,
    output reg out_valid,
    output reg end_conv
    );
    parameter n=3, m=2;
    reg [31:0] k [m*m-1:0];
    reg [31:0] w [m*m-1:0];
    reg [31:0] u [m*m-1:0];
    reg [31:0] p [(m-1)*n-1:0];
    reg [5:0] count, count1, count2, count3;
    reg [5:0] y;

    always@(posedge clock) begin
    if(rst) count <= 'b0;
    else if(end_conv == 'b1) count <= 'b0;
    else count <= count + 1'b1;
    end

    always@(posedge clock)
    k[m*m-1] <= (count<m*m) ? a_in : k[m*m-1];

    genvar i;
    generate
    for(i=0;i<m*m-1;i=i+1)
    always@(posedge clock)
    k[m*m-1-i-1] <= (count<m*m) ? k[m*m-1-i] : k[m*m-1-i-1] ;
    endgenerate

    always@(posedge clock)
    w[m*m-1] <= a_in;

    genvar j,l;
```

```
generate
for(i=0;i<m-1;i=i+1) begin

for(j=0;j<m-1;j=j+1)
always@(posedge clock)
w[m*m-1-i*m-j-1] <= w[m*m-1-i*m-j];

always@(*)
p[0+i*m] <= w[m*m-1-i*m];
for(l=0;l<n-1;l=l+1)
always@(posedge clock) begin
p[l+i*m+1] <= p[l+i*m];
w[m*m-1-i*m-m] <= p[n-1];
end
end

for(j=0;j<m-1;j=j+1)
always@(posedge clock)
w[m-1-j-1] <= w[m-1-j];
endgenerate
//inputs at w and k

always@(*) u[m*m-1] <= w[m*m-1]*k[m*m-1];
generate
for(i=m*m-2;i>=0;i=i-1)
always@(*) u[i] <= u[i+1] + w[i]*k[i];
endgenerate

assign result = u[0];

// logic to generate valids
always@(posedge clock) begin
if(rst) begin out_valid <= 'b0; end_conv <= 'b0; end
else if(count < (m-1)*n + m - 'b1 + m*m );
else if(count == (m-1)*n + m - 'b1 + m*m) begin
out_valid <= 'b1; count1<='b1; count2<='b1; count3<='b1; end
else if(end_conv == 'b1) begin out_valid <= 'b0; end_conv <= 'b0; end
else if((count2 == n-m) && (count3 == n-m+'b1)) begin
```

```
    end_conv <= 'b1; count1<='b0; count2<='b0; count3<='b0; end
    else if(count2 == n-m+'b1) begin
    out_valid <= 'b0; count1<=count1+'b1; count2<='b0; end
    else if(count1%n == 'b0) begin
    out_valid <= 'b1; count1<=count1+'b1; count2<='b1; count3<=count3+'b1; end
    else begin count1<=count1+'b1; count2<=count2+'b1; end
    end

endmodule
```

Note: Decimal point is not defined in code. Just take proper care in testbench to have x bits after decimal in input and 2x bits in output. Here we take 5 bits after decimal in input and 10bits in output for simulation. Code works for all cases.

3.3 Test bench ;

```
module tbconv_fix(    );

reg clock, rst;
reg [31:0] a;
wire [31:0] result;
wire out_valid, end_conv;

always #50 clock = ~clock;

conv_fix DUT (clock, rst, a, result, out_valid, end_conv);

always@(negedge clock) a = a+6'b100000;

initial begin
clock = 0; rst = 1;
#100 rst = 0; a = 6'b000000;
end
endmodule
```

3.4 RTL Schematic :

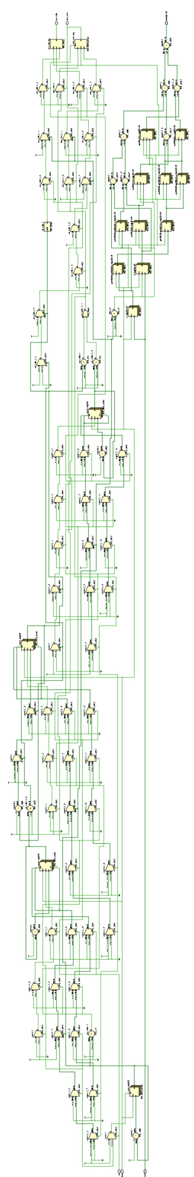

Figure 3: Schematic of 2's complement convolution circuit

3.5 Simulation :

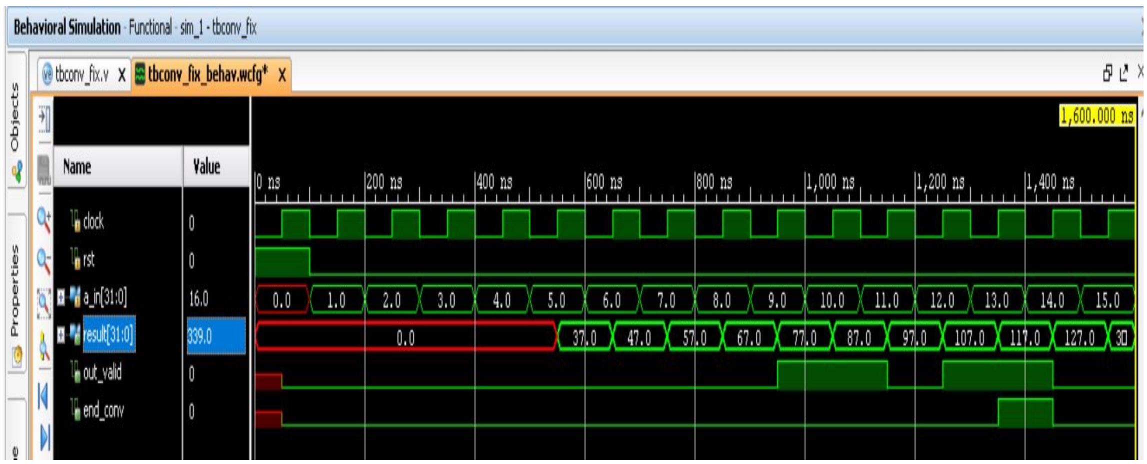

Figure 4: Behavioural simulation waveform

Figure 5: Post-synthesis functional simulation waveform

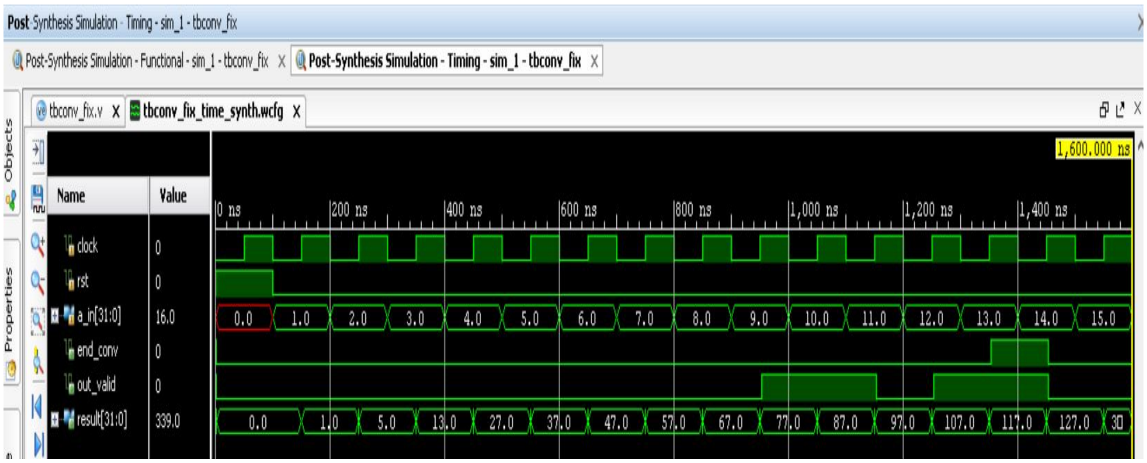

Figure 6: Post synthesis timing simulation waveform

Note: The representation of the input and output are kept in fixed point. The kernal matrix is 1,2,3,4 and main matrix is 5,6,7,8,9,10,11,12,13 so the outputs should be 77, 87, 107, 117. The results are as expected.

3.6 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.

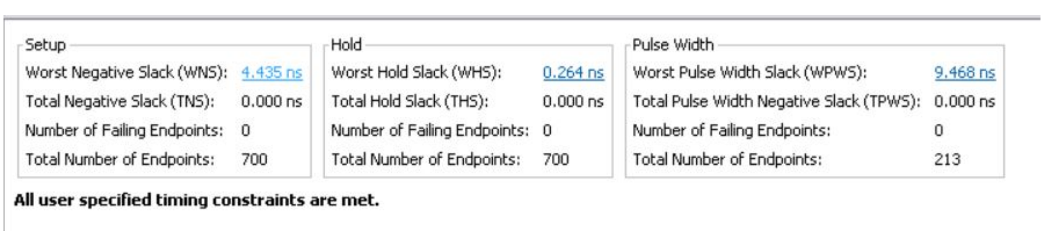

Figure 7: Timing summary

```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Loced | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  153 |     0 |    134600 |  0.11 |
|   LUT as Logic          |  153 |     0 |    134600 |  0.11 |
|   LUT as Memory         |    0 |     0 |     46200 |  0.00 |
| Slice Registers         |  203 |     0 |    269200 |  0.07 |
|   Register as Flip Flop |  203 |     0 |    269200 |  0.07 |
|   Register as Latch     |    0 |     0 |    269200 |  0.00 |
| F7 Muxes                |    0 |     0 |     67300 |  0.00 |
| F8 Muxes                |    0 |     0 |     33650 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and ful
```

Figure 8: Utilization report

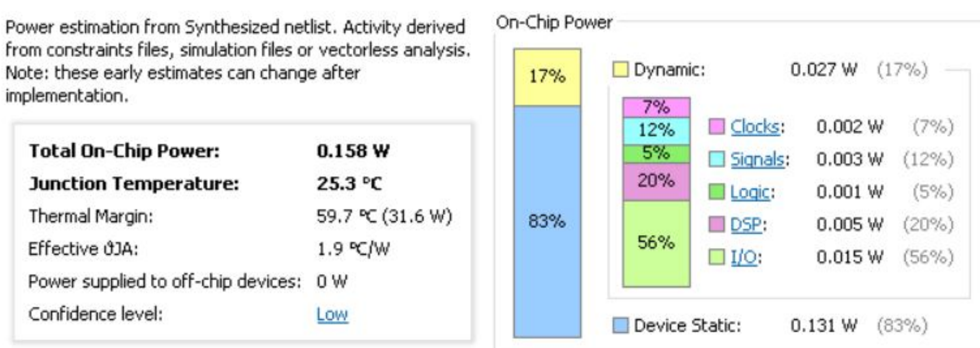

Figure 9: Power report

4 Matrix Convolution with IEEE754 representation :

4.1 Design Approach :

It is same as previous only the multipliers and adders are made by using the modules of previous experiments. Rest of the logic is all same.

4.2 Verilog Code :

```
module conv_flt(
    input clock,
    input rst,
    input [31:0] a_in,
    output [31:0] result,
    output reg out_valid,
    output reg end_conv
    );
    parameter n=3, m=2;
    reg [31:0] k [m*m-1:0];
    reg [31:0] w [m*m-1:0];
    wire [31:0] u [m*m-1:0];
    wire [31:0] v [m*m-2:0];
    reg [31:0] p [(m-1)*n-1:0];
    reg [5:0] count, count1, count2, count3;
    reg [5:0] y;

    always@(posedge clock) begin
    if(rst) count <= 'b0;
    else if(end_conv == 'b1) count <= 'b0;
    else count <= count + 1'b1;
    end

    always@(posedge clock)
    k[m*m-1] <= (count<m*m) ? a_in : k[m*m-1];

    genvar i;
    generate
```

```
for(i=0;i<m*m-1;i=i+1)
always@(posedge clock)
k[m*m-1-i-1] <= (count<m*m) ? k[m*m-1-i] : k[m*m-1-i-1] ;
endgenerate

always@(posedge clock)
w[m*m-1] <= a_in;

genvar j,l;
generate
for(i=0;i<m-1;i=i+1) begin

for(j=0;j<m-1;j=j+1)
always@(posedge clock)
w[m*m-1-i*m-j-1] <= w[m*m-1-i*m-j];

always@(*)
p[0+i*m] <= w[m*m-1-i*m];
for(l=0;l<n-1;l=l+1)
always@(posedge clock) begin
p[l+i*m+1] <= p[l+i*m];
w[m*m-1-i*m-m] <= p[n-1];
end
end

for(j=0;j<m-1;j=j+1)
always@(posedge clock)
w[m-1-j-1] <= w[m-1-j];
endgenerate
//inputs at w and k

fpmul_32b mul(w[m*m-1],k[m*m-1],u[m*m-1]);
generate
for(i=m*m-2;i>=0;i=i-1) begin
fpmul_32b mul(w[i],k[i],v[i]);
fpaddsub_32b adsb(u[i+1], v[i], 'b0, u[i]);
end
endgenerate
```

```
    assign result = u[0];

    // logic to generate valids
    always@(posedge clock) begin
    if(rst) begin out_valid <= 'b0; end_conv <= 'b0; end
    else if(count < (m-1)*n + m - 'b1 + m*m );
    else if(count == (m-1)*n + m - 'b1 + m*m) begin
    out_valid <= 'b1; count1<='b1; count2<='b1; count3<='b1; end
    else if(end_conv == 'b1) begin out_valid <= 'b0; end_conv <= 'b0; end
    else if((count2 == n-m) && (count3 == n-m+'b1)) begin
    end_conv <= 'b1; count1<='b0; count2<='b0; count3<='b0; end
    else if(count2 == n-m+'b1) begin
    out_valid <= 'b0; count1<=count1+'b1; count2<='b0; end
    else if(count1%n == 'b0) begin
    out_valid <= 'b1; count1<=count1+'b1; count2<='b1; count3<=count3+'b1; end
    else begin count1<=count1+'b1; count2<=count2+'b1; end
    end

endmodule

module fpmul_32b(a_in, b_in, c_out );
parameter m = 8, n = 23;
input [m+n:0] a_in, b_in;
output [m+n:0] c_out;

wire sign, is_zero;
wire [m-1:0] exp, exp_c, exp_d;
wire [2*n+1:0] mant;
wire [2*n+1:0] mul;

assign sign = a_in[m+n] ^ b_in[m+n];
assign exp_c = a_in[m+n-1:n] + b_in[m+n-1:n] - {(m-1){1'b1}};
assign mul = {1'b1,a_in[n-1:0]} * {1'b1,b_in[n-1:0]};
assign is_zero = (a_in[m+n-1:0] == 'b0) || (b_in[m+n-1:0] == 'b0);

assign exp = exp_c + mul[2*n+1];
assign mant = mul>>(mul[2*n+1]);
```

```
assign c_out = is_zero ? 'b0 : {sign,exp,mant[2*n-1:n]};

endmodule

module fpaddsub_32b(a_in, b_in, sub, c_out );
parameter m = 8, n = 23;
input [m+n:0] a_in, b_in;
input sub;
output [m+n:0] c_out;

wire sign, sub_b, a_is_big, b_is_big, both_equal, mant_abig;
wire [m-1:0] exp, exp_c, exp_d, shift_index;
wire [n+1:0] mant, mant_d, a_shifted, b_shifted, shifted;

assign a_is_big = a_in[m+n-1:n] > b_in[m+n-1:n];
assign b_is_big = a_in[m+n-1:n] < b_in[m+n-1:n];
assign both_equal = a_in[m+n-1:n] == b_in[m+n-1:n];
assign mant_abig = both_equal ?
                a_in[n-1:0] > b_in[n-1:0] : 'b0;

assign exp_c = a_is_big ? a_in[m+n-1:n] - b_in[m+n-1:n] :
                b_is_big ? b_in[m+n-1:n] - a_in[m+n-1:n] : 'b0;

assign a_shifted = b_is_big ? {1'b1,a_in[n-1:0]}>>(exp_c) :
                     {1'b1,a_in[n-1:0]};
assign b_shifted = a_is_big ? {1'b1,b_in[n-1:0]}>>(exp_c) :
                     {1'b1,b_in[n-1:0]};

assign sub_b = a_in[m+n] ^ (b_in[m+n] ^ sub);
assign mant_d = sub_b ? (a_is_big || mant_abig) ?
                a_shifted - b_shifted :
                b_shifted - a_shifted :
                a_shifted + b_shifted;

assign sign = (a_is_big || mant_abig) ? a_in[m+n] : (b_in[m+n] ^ sub);
assign exp_d = a_is_big ? a_in[m+n-1:n] : b_in[m+n-1:n];
```

```
//corner case logic- not needed for normal operation
mantshift mant_shifter(exp_d, mant_d, shifted, shift_index );
//--------------------------------------------------

assign mant = sub_b ? shifted : mant_d>>mant_d[n+1];
assign exp = sub_b ? exp_d - shift_index : exp_d + mant_d[n+1];

assign c_out = {sign,exp,mant[n-1:0]};

endmodule

module mantshift(exp, mant, shifted, shift_index );
parameter m = 8, n = 23;
    input [m-1:0] exp;
    input [n+1:0] mant;
    output reg [m-1:0] shift_index;
    output reg [n+1:0] shifted;
    reg [n+1:0] target;
    reg [$clog2(n)-1:0] cnt;
    always@(mant, exp) begin target = mant;
    shift_index = exp;
    for(cnt = 0; cnt < n+1; cnt = cnt + 1)begin
        if (target[cnt]) shift_index = n - cnt;
    end
    shifted = mant << shift_index;
    end
endmodule
```

4.3 Test bench ;

```
module tbconv_flt( );

    reg clock, rst;
    reg [31:0] a;
    wire [31:0] result;
    wire out_valid, end_conv;

    always #50 clock = ~clock;
```

```
    conv_flt DUT (clock, rst, a, result, out_valid, end_conv);

    initial begin
    clock = 1; rst = 1; #100 rst = 0;
    a = 32'b0_01111111_00000000000000000000000;
    #100 a = 32'b0_10000000_00000000000000000000000;
    #100 a = 32'b0_10000000_10000000000000000000000;
    #100 a = 32'b0_10000001_00000000000000000000000;
    #100 a = 32'b0_10000001_01000000000000000000000;
    #100 a = 32'b0_10000001_10000000000000000000000;
    #100 a = 32'b0_10000001_11000000000000000000000;
    #100 a = 32'b0_10000010_00000000000000000000000;
    #100 a = 32'b0_10000010_00100000000000000000000;
    #100 a = 32'b0_10000010_01000000000000000000000;
    #100 a = 32'b0_10000010_01100000000000000000000;
    #100 a = 32'b0_10000010_10000000000000000000000;
    #100 a = 32'b0_10000010_10100000000000000000000;
    #100 a = 32'b0_10000010_11000000000000000000000;
    end
endmodule
```

4.4 RTL Schematic :

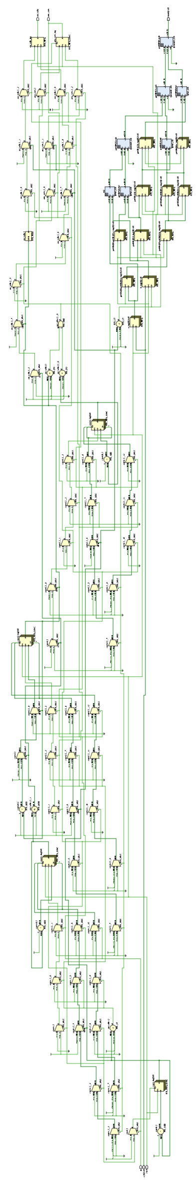

Figure 10: Schematic of IEEE754 convolution circuit

4.5 Simulation :

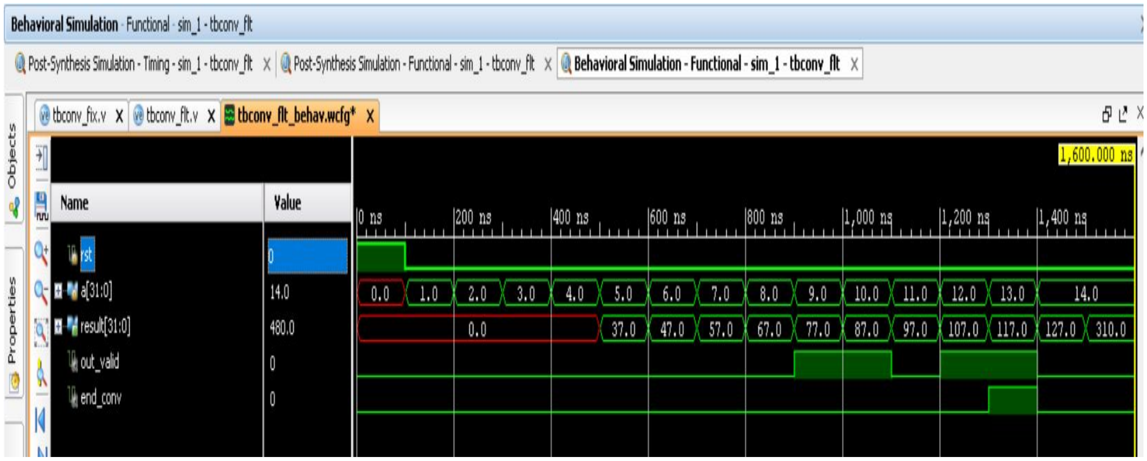

Figure 11: Behavioural simulation waveform

Figure 12: Post-synthesis functional simulation waveform

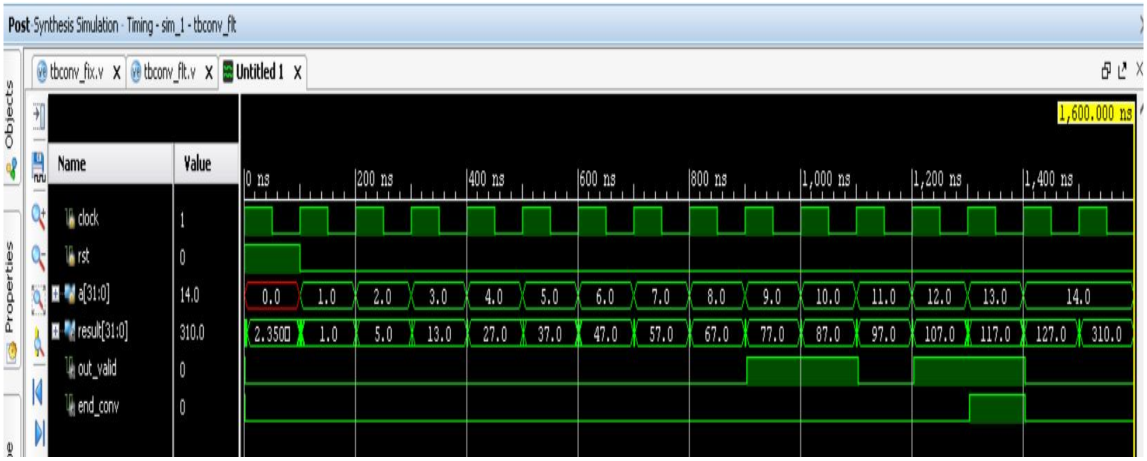

Figure 13: Post synthesis timing simulation waveform

Note: The representation of the input and output are kept in single precision floating point representation. The kernal matrix is 1,2,3,4 and main matrix is 5,6,7,8,9,10,11,12,13 so the outputs should be 77, 87, 107, 117. The results are as expected.

4.6 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.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.956 ns	Worst Hold Slack (WHS):	0.264 ns	Worst Pulse Width Slack (WPWS):	34.468 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	510	Total Number of Endpoints:	510	Total Number of Endpoints:	315

All user specified timing constraints are met.

Figure 14: Timing summary

```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Loced | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             | 1756 |     0 |    134600 |  1.30 |
|   LUT as Logic          | 1756 |     0 |    134600 |  1.30 |
|   LUT as Memory         |    0 |     0 |     46200 |  0.00 |
| Slice Registers         |  314 |     0 |    269200 |  0.11 |
|   Register as Flip Flop |  314 |     0 |    269200 |  0.11 |
|   Register as Latch     |    0 |     0 |    269200 |  0.00 |
| F7 Muxes                |    0 |     0 |     67300 |  0.00 |
| F8 Muxes                |    0 |     0 |     33650 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full
```

Figure 15: Utilization report

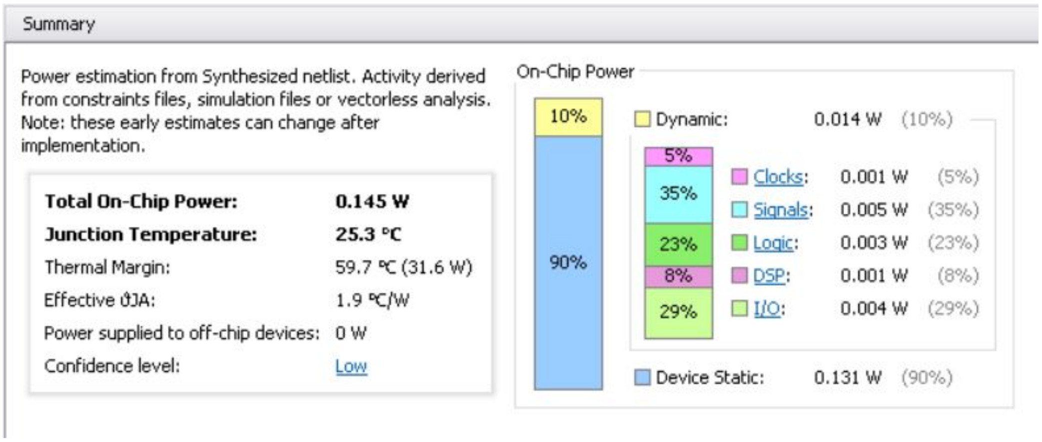

Figure 16: Power report

5 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.