LUT optimization of Fast Fourier Transform

Mohankumar Ramachandran

January 21, 2019

Abstract

Fast Fourier Transform (FFT) remains of a great importance due to its substantial role in the field of signal processing and imagery. In this report, multiple designs of 8 point FFT algorithm is proposed. The developed architecture was implemented using an FPGA . Though, the material resources of the FPGA are limited, particularly the integrated DSP blocks, different approaches are used during the Verilog description with the aim to reduce the necessary number of LUTs. The experimental validation was done using ISIM simulation tool, where the numerical synthesis and the post and route described in Verilog was realized using ISE Design Suite 14.7. The FFT modules of all the implementations were tested using a python script with various corner input test vectors.

1 Cooley–Tukey algorithm

1.1 Overview

Cooley–Tukey algorithm re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size N = $N_1 N_2$ in terms of N_1 smaller DFTs of sizes N_2, recursively, to reduce the computation time to **O**(N log N) for highly composite N.

1.2 Calculation

This is a direct implementation of Cooley-Turkey Algorithm without any modifications , and for a 8 point FFT , it involves 2 multiplications for a single output calculation. The equations involving the calculation of FFT in shown below.

$t_1 = D(0) + D(4); m_3 = D(0) - D(4);$
$t_2 = D(6) + D(2); m_6 = j * (D(6) - D(2));$
$t_3 = D(1) + D(5); t_4 = D(1) - D(5);$
$t_5 = D(3) + D(7); t_6 = D(3) - D(7);$
$t_8 = t_5 + t_3; m_5 = j * (t_5 - t_3);$
$t_7 = t_1 + t_2; m_2 = t_1 - t_2;$

$m_0 = t_7 + t_8; m_1 = t_7 - t_8;$
$m_4 = sin(\pi/4) * (t_4 - t_6); m_7 = -j * sin(\pi/4) * (t_4 + t_6);$
$s_1 = m_3 + m_4; s2 = m_3 - m_4;$
$s_3 = m_6 + m_7; s4 = m_6 - m_7;$
$DO(0) = m_0; DO(4) = m_1;$
$DO(1) = s_1 + s_3; DO(7) = s_1 - s_3;$
$DO(2) = m_2 + m_5; DO(6) = m_2 - m_5;$
$DO(5) = s_2 + s_4; DO(3) = s_2 - s_4;$

where D and DO are input and output arrays of the complex data t_1,...,t_8, m_1,..., m_7, s_1,...,s_4 are the intermediate complex results. As we see the algorithm contains only 2 multiplications to the untrivial coefficient $\sin(\pi/4) = 0.7071$, and 22 real additions and subtractions. The multiplication to a coefficient j means the negation the imaginary part and swapping real and imaginary parts.

1.3 Implementation

The below code implements the Cooley–Tukey algorithm . inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated.

```
1  `timescale 1ns / 1ps
2
3  module fft8(
4          input signed [15:0] inp1,
5          input signed [15:0] inp2,
6          input signed [15:0] inp3,
7          input signed [15:0] inp4,
8          input signed [15:0] inp5,
9          input signed [15:0] inp6,
10         input signed [15:0] inp7,
11         input signed [15:0] inp8,
12         input clk,
13         input rst,
14         output signed [15:0] out1_real,
15         output signed [15:0] out1_imag,
16         output signed [15:0] out2_real,
17         output signed [15:0] out2_imag,
18         output signed [15:0] out3_real,
19         output signed [15:0] out3_imag,
20         output signed [15:0] out4_real,
21         output signed [15:0] out4_imag,
22         output signed [15:0] out5_real,
23         output signed [15:0] out5_imag,
```

```
24          output signed [15:0] out6_real,
25          output signed [15:0] out6_imag,
26          output signed [15:0] out7_real,
27          output signed [15:0] out7_imag,
28          output signed [15:0] out8_real,
29          output signed [15:0] out8_imag,
30          output out_stb
31  );
32
33          localparam signed sin_45 = 16'b00000000_10110101;
34          localparam signed sin_315 = 16'b11111111_01001011;
35
36          reg signed [31:0] t1_46,t2_46;
37          reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
38          reg signed [15:0] m7_imag,s1,s2,s3_imag,s4_imag,m5_imag,m6_imag;
39          reg output_stb;
40
41          initial
42                  begin
43                          output_stb = 1'b0;
44                  end
45
46          always @( posedge clk )
47                  begin
48                          if (rst == 1'b1)
49                                  begin
50                                          output_stb = 1'b0;
51                                  end
52                          else
53                                  begin
54                                          t1 = inp1 + inp5;
55                                          t2 = inp7 + inp3;
56                                          t3 = inp2 + inp6;
57                                          t5 = inp4 + inp8;
58                                          m3 = inp1 - inp5;
59                                          m6_imag = inp7 - inp3;
60                                          t4 = inp2 - inp6;
61                                          t6 = inp4 - inp8;
62                                          t8 = t5 + t3;
63                                          t7 = t1 + t2;
64                                          m0 = t7 + t8;
65                                          t1_46 = sin_45 * ( t4 - t6);
66                                          m4 = t1_46 [23:8];
67                                          m5_imag = t5 - t3;
68                                          m2 = t1 - t2;
69                                          m1 = t7 - t8;
70                                          t2_46 = sin_315 * ( t4 + t6);
71                                          m7_imag = t2_46 [23:8];
72                                          s1 = m3 + m4;
73                                          s2 = m3 - m4;
```

```
74                                          s3_imag = m6_imag + m7_imag;
75                                          s4_imag = m6_imag - m7_imag;
76                                          output_stb = 1'b1;
77                                  end
78                  end
79          assign out1_real = m0;
80          assign out1_imag = 16'b0000000000000000;
81          assign out2_real = s1;
82          assign out2_imag = s3_imag;
83          assign out3_real = m2;
84          assign out3_imag = m5_imag;
85          assign out4_real = s2;
86          assign out4_imag = ~s4_imag + 1'b1;
87          assign out5_real = m1;
88          assign out5_imag = 16'b0000000000000000;
89          assign out6_real = s2;
90          assign out6_imag = s4_imag;
91          assign out7_real = m2;
92          assign out7_imag = ~m5_imag + 1'b1;
93          assign out8_real = s1;
94          assign out8_imag = ~s3_imag + 1'b1;
95          assign out_stb = output_stb;
96  endmodule
```

The above code with Test Bench is available in FFT8_implementation_1 directory of the project GIT repository.

1.4 Analysis

This algorithm calculates the different stages of the Butterfly structure in parallel and hence total number of operations is as follows.

- Multiplications: 2
- Additions and Subtractions: 22

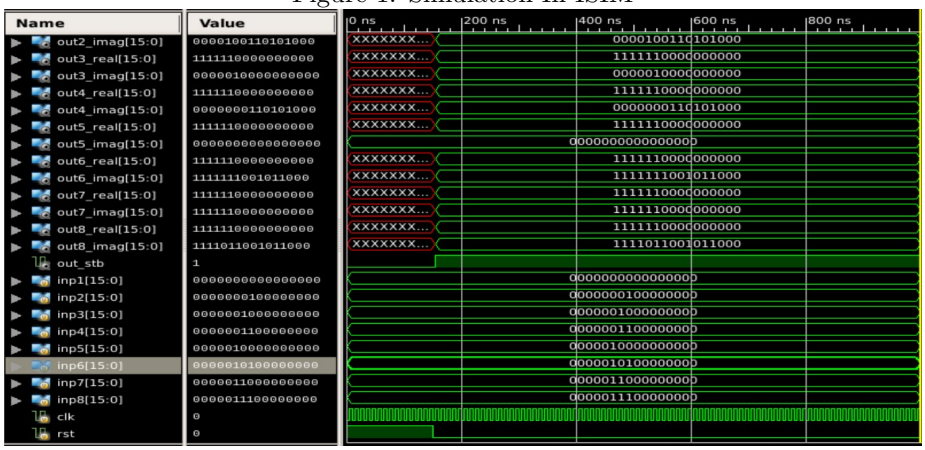
since both the multiplications happen in parallel , total 2 multipliers are used in this block to evaluate , hence improving speed.The simulation of this algorithm is shown in **Figure 1**.

2 Modified Cooley–Tukey algorithm

2.1 Overview

Cooley–Tukey algorithm for 8 point FFT involves 2 multiplication , hence using 2 multiplicative blocks as seen in the first implementation.In this modified algorithm , the multiplications are pipelined to run one by one , by reusing the same multiplier.

Figure 1: Simulation In ISIM



2.2 Calculation

The calculations of various intermediate complex numbers is divided into groups and executed group by group.The equation involving different groups are mentioned below.

2.2.1 Group 1

$t_1 = D(0) + D(4);$
$m_3 = D(0) - D(4);$
$t_2 = D(6) + D(2);$
$m_6 = j * (D(6) - D(2));$
$t_3 = D(1) + D(5);$
$t_4 = D(1) - D(5);$
$t_5 = D(3) + D(7);$
$t_6 = D(3) - D(7);$
$t_8 = t_5 + t_3;$
$m_5 = j * (t_5 - t_3);$
$t_7 = t_1 + t_2;$
$m_2 = t_1 - t_2;$
$m_0 = t_7 + t_8;$
$m_1 = t_7 - t_8;$
$m_4 = sin(\pi/4) * (t_4 - t_6);$

2.2.2 Group 2

$m_7 = -j * sin(\pi/4) * (t_4 + t_6);$
$s_1 = m_3 + m_4;$
$s2 = m_3 - m_4;$

$s_3 = m_6 + m_7;$
$s4 = m_6 - m_7;$
$DO(0) = m_0;$
$DO(4) = m_1;$
$DO(1) = s_1 + s_3;$
$DO(7) = s_1 - s_3;$
$DO(2) = m_2 + m_5;$
$DO(6) = m_2 - m_5;$
$DO(5) = s_2 + s_4;$
$DO(3) = s_2 - s_4;$

where D and DO are input and output arrays of the complex data t_1,...,t_8, m_1,..., m_7, s_1,...,s_4 are the intermediate complex results. As we see the algorithm contains only 2 multiplications to the untrivial coefficient $\sin(\pi/4) = 0.7071$, and 22 real additions and subtractions. The multiplication to a coefficient j means the negation the imaginary part and swapping real and imaginary parts.

2.3 Implementation

The below code implements the Modified Cooley–Tukey algorithm . inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated. multiplier module takes inp1, inp2 16 bit signed floating point inputs of Q format and outputs the product of the inputs to out.The module multiplier is initialized once and reused twice in the main FFT8 block .

```
1  `timescale 1ns / 1ps
2
3  module multiplier(
4      input signed [15:0] inp1,
5      input signed [15:0] inp2,
6      output signed[15:0] out
7      );
8          reg signed [31:0] inp12;
9          always @ ( * )
10                 begin
11                         inp12 = inp1*inp2;
12                 end
13         assign out = inp12 [23:8];
14 endmodule
15
16
17 module fft8(
18         input signed [15:0] inp1,
```

```
19          input signed [15:0] inp2,
20          input signed [15:0] inp3,
21          input signed [15:0] inp4,
22          input signed [15:0] inp5,
23          input signed [15:0] inp6,
24          input signed [15:0] inp7,
25          input signed [15:0] inp8,
26          input clk,
27          input rst,
28          output signed [15:0] out1_real,
29          output signed [15:0] out1_imag,
30          output signed [15:0] out2_real,
31          output signed [15:0] out2_imag,
32          output signed [15:0] out3_real,
33          output signed [15:0] out3_imag,
34          output signed [15:0] out4_real,
35          output signed [15:0] out4_imag,
36          output signed [15:0] out5_real,
37          output signed [15:0] out5_imag,
38          output signed [15:0] out6_real,
39          output signed [15:0] out6_imag,
40          output signed [15:0] out7_real,
41          output signed [15:0] out7_imag,
42          output signed [15:0] out8_real,
43          output signed [15:0] out8_imag,
44          output out_stb
45  );
46
47          localparam signed sin_45 = 16'b00000000_10110101;
48          localparam signed sin_315 = 16'b11111111_01001011;
49          localparam signed sf = 2.0**-8.0;
50
51          reg signed [31:0] t1_46,t2_46;
52          reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
53          reg signed [15:0] m7_imag,s1,s2,s3_imag,s4_imag,m5_imag,m6_imag;
54          reg [15:0] mult_inp1,mult_inp2;
55          wire [15:0] mult_out;
56          reg [1:0] stage;
57          reg output_stb;
58
59          multiplier mult (.inp1(mult_inp1), .inp2(mult_inp2), .out(mult_out));
60
61          initial
62                  begin
63                          stage = 2'b00;
64                          output_stb = 1'b0;
65                  end
66
67          always @( posedge clk)
68                  begin
```

```
69                          if (rst == 1'b1)
70                                  begin
71                                          output_stb = 1'b0;
72                                  end
73                          if (stage == 2'b00 && rst == 1'b0)
74                                  begin
75                                          $display("stage-1");
76                                          t1 = inp1 + inp5;
77                                          t2 = inp7 + inp3;
78                                          t3 = inp2 + inp6;
79                                          t5 = inp4 + inp8;
80                                          m3 = inp1 - inp5;
81                                          m6_imag = inp7 - inp3;
82                                          t4 = inp2 - inp6;
83                                          t6 = inp4 - inp8;
84                                          t8 = t5 + t3;
85                                          t7 = t1 + t2;
86                                          m0 = t7 + t8;
87                                          mult_inp1 = t4 - t6;
88                                          mult_inp2 = sin_45;
89                                          stage = 2'b01;
90                                  end
91                          else if (stage == 2'b01)
92                                  begin
93                                          $display("stage-2");
94                                          m4 = mult_out;
95                                          m5_imag = t5 - t3;
96                                          m2 = t1 - t2;
97                                          m1 = t7 - t8;
98                                          mult_inp1 = t4 + t6;
99                                          mult_inp2 = sin_315;
100                                         stage = 2'b10;
101                                 end
102                         else if (stage == 2'b10)
103                                 begin
104                                         m7_imag = mult_out;
105                                         s1 = m3 + m4;
106                                         s2 = m3 - m4;
107                                         s3_imag = m6_imag + m7_imag;
108                                         s4_imag = m6_imag - m7_imag;
109                                         stage = 2'b00;
110                                         output_stb = 1'b1;
111                                 end
112                 end
113         assign out_stb = output_stb;
114         assign out1_real = m0;
115         assign out1_imag = 16'b0000000000000000;
116         assign out2_real = s1;
117         assign out2_imag = s3_imag;
118         assign out3_real = m2;
```

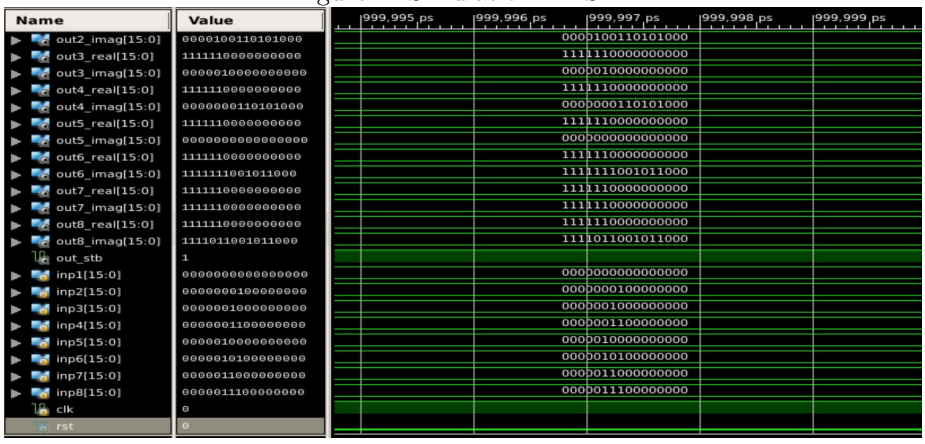
```
119         assign out3_imag = m5_imag;
120         assign out4_real = s2;
121         assign out4_imag = ~s4_imag + 1'b1;
122         assign out5_real = m1;
123         assign out5_imag = 16'b0000000000000000;
124         assign out6_real = s2;
125         assign out6_imag = s4_imag;
126         assign out7_real = m2;
127         assign out7_imag = ~m5_imag + 1'b1;
128         assign out8_real = s1;
129         assign out8_imag = ~s3_imag + 1'b1;
130 endmodule
```

The above code with Test Bench is available in FFT8_implementation_2 directory of the project GIT repository.

2.4 Analysis

This algorithm calculates the intermediate values in FFT block serially hence reusing the multipliers. Since multipliers are reused , the input bandwidth reduces by half from previous implementation , since 2 clock cycles are needed to perform 2 multiplications one by one. since both the multiplications happen in serial , total 1 multiplier is only used in this block to evaluate , hence reducing speed . Number of LUTs used is increased since multiplexers are implemented in LUT now to reuse the multiplier by changing inputs in alternate clock cycles . The simulation of this algorithm is shown in **Figure 2**.

Figure 2: Simulation In ISIM



3 Modified Cooley–Tukey algorithm using LUT based Multiplier

3.1 Overview

Cooley–Tukey algorithm for 8 point FFT involves 2 multiplication , hence using 2 multiplicative blocks as seen in the first implementation.In this modified algorithm , the multiplications are pipelined to run one by one , and using a LUT based 16 bit signed multiplier , reducing number of dedicated multiplier blocks , but increasing number of LUTs used.

3.2 Implementation

The below code implements the Modified Cooley–Tukey algorithm using LUT based multiplier. inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated. multiplier module takes inp1, inp2 16 bit signed floating point inputs of Q format and outputs the product of the inputs to out.The module multiplier is initialized once and reused twice in the main FFT8 block . The multiplier module takes input of 2 Q format number , and using Shift-Add algorithm , it multiplies the numbers and outputs every 16 clock cycles.

```
1
2  `timescale 1ns / 1ps
3
4  module multiplier(
5      input signed [15:0] inp1,
6      input signed [15:0] inp2,
7           input rst,
8           input clk,
9      output signed [15:0] out,
10          output out_stb
11 );
12    localparam sf = 2.0**-8.0;
13         reg [29:0] inp12;
14         reg [14:0] input_1;
15         reg [14:0] input_2;
16         reg output_stb;
17         reg out_sign;
18         integer counter;
19         assign out_stb = output_stb;
20         initial
21                 begin
22                         output_stb = 1'b0;
```

```
                        inp12 = 32'b0;
                        counter = 0;
                end

always @ ( posedge clk )
begin
if (rst == 1'b1)
        begin
        output_stb = 1'b0;
        inp12 = 30'b0;
        counter = 0;
        end
else if (output_stb == 1'b0 && counter < 15)
        begin
        if (counter == 0)
        begin
        out_sign = (inp1[15] && ~ inp2 [15]) + (inp2[15] && ~ inp1 [15]);
        input_1 = inp1[15]==1'b0 ? inp1[14:0] : ~inp1[14:0]+1'b1;
        input_2 = inp2[15]==1'b0 ? inp2[14:0] : ~inp2[14:0]+1'b1;
        end
        if(input_1[counter]==1'b1)
        begin
        inp12[29:14] = inp12[29:14] + input_2[14:0];
        end
        inp12 = inp12 >> 1;
        counter = counter + 1;
        end
else if (counter >= 15)
        begin
                output_stb = 1'b1;
        end
end
assign out = {out_sign,inp12[22:7]};
endmodule


module fft8(
        input signed [15:0] inp1,
        input signed [15:0] inp2,
        input signed [15:0] inp3,
        input signed [15:0] inp4,
        input signed [15:0] inp5,
        input signed [15:0] inp6,
        input signed [15:0] inp7,
        input signed [15:0] inp8,
        input clk,
        input rst,
        output signed [15:0] out1_real,
        output signed [15:0] out1_imag,
```

```
73          output signed [15:0] out2_real,
74          output signed [15:0] out2_imag,
75          output signed [15:0] out3_real,
76          output signed [15:0] out3_imag,
77          output signed [15:0] out4_real,
78          output signed [15:0] out4_imag,
79          output signed [15:0] out5_real,
80          output signed [15:0] out5_imag,
81          output signed [15:0] out6_real,
82          output signed [15:0] out6_imag,
83          output signed [15:0] out7_real,
84          output signed [15:0] out7_imag,
85          output signed [15:0] out8_real,
86          output signed [15:0] out8_imag,
87          output out_stb
88  );
89
90          localparam signed sin_45 = 16'b00000000_10110101;
91          localparam signed sin_315 = 16'b11111111_01001011;
92          localparam signed sf = 2.0**-8.0;
93
94          reg signed [31:0] t1_46,t2_46;
95          reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
96          ref signed [15:0] m5_imag,m6_imag,m7_imag,s1,s2,s3_imag,s4_imag;
97          reg [15:0] mult_inp1,mult_inp2;
98          wire [15:0] mult_out;
99          reg [1:0] stage;
100         reg output_stb,mult_rst;
101         wire mult_stb;
102
103         multiplier mult (
104                 .inp1(mult_inp1),
105                 .inp2(mult_inp2),
106                 .rst(mult_rst),
107                 .out(mult_out),
108                 .clk(clk),
109                 .out_stb(mult_stb)
110         );
111
112         initial
113                 begin
114                         stage = 2'b00;
115                         output_stb = 1'b0;
116                         mult_rst = 1'b1;
117                 end
118
119         always @( posedge clk)
120                 begin
121                         if (rst == 1'b1)
122                                 begin
```

```
123                                     output_stb = 1'b0;
124                                     mult_rst = 1'b1;
125                             end
126                     if (stage == 2'b00 && rst == 1'b0 && output_stb == 1'b0)
127                             begin
128                                     t1 = inp1 + inp5;
129                                     t2 = inp7 + inp3;
130                                     t3 = inp2 + inp6;
131                                     t5 = inp4 + inp8;
132                                     m3 = inp1 - inp5;
133                                     m6_imag = inp7 - inp3;
134                                     t4 = inp2 - inp6;
135                                     t6 = inp4 - inp8;
136                                     t8 = t5 + t3;
137                                     t7 = t1 + t2;
138                                     m0 = t7 + t8;
139                                     mult_inp1 = t4 - t6;
140                                     mult_inp2 = sin_45;
141                                     stage = 2'b01;
142                                     mult_rst = 1'b0;
143                             end
144                     else if (stage == 2'b01 && mult_stb == 1'b1)
145                             begin
146                                     m4 = mult_out;
147                                     mult_rst = 1'b1;
148                                     stage = 2'b10;
149                             end
150                     else if (stage == 2'b10)
151                             begin
152                                     m5_imag = t5 - t3;
153                                     m2 = t1 - t2;
154                                     m1 = t7 - t8;
155                                     mult_inp1 = t4 + t6;
156                                     mult_inp2 = sin_315;
157                                     mult_rst = 1'b0;
158                                     stage = 2'b11;
159                             end
160                     else if (stage == 2'b11 && mult_stb == 1'b1)
161                             begin
162                                     m7_imag = mult_out;
163                                     s1 = m3 + m4;
164                                     s2 = m3 - m4;
165                                     s3_imag = m6_imag + m7_imag;
166                                     s4_imag = m6_imag - m7_imag;
167                                     mult_rst = 1'b1;
168                                     stage = 2'b00;
169                                     output_stb = 1'b1;
170                             end
171             end
172     assign out_stb = output_stb;
```

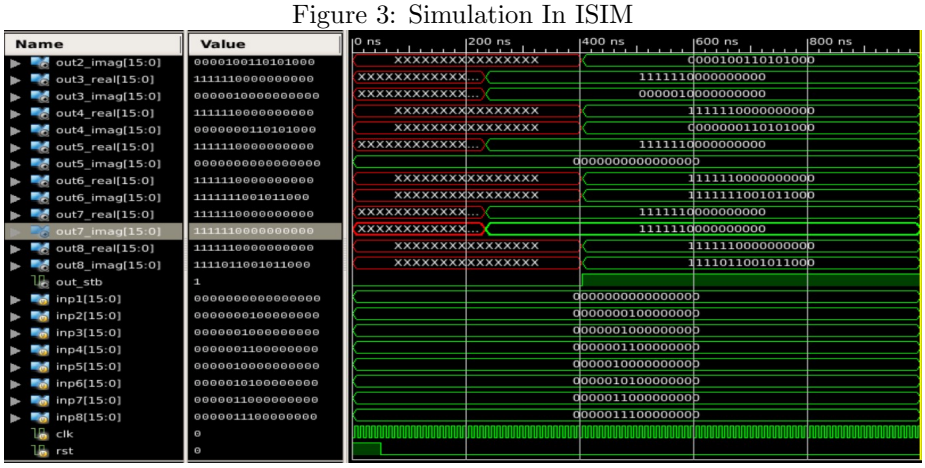
```
173         assign out1_real = m0;
174         assign out1_imag = 16'b0000000000000000;
175         assign out2_real = s1;
176         assign out2_imag = s3_imag;
177         assign out3_real = m2;
178         assign out3_imag = m5_imag;
179         assign out4_real = s2;
180         assign out4_imag = ~s4_imag + 1'b1;
181         assign out5_real = m1;
182         assign out5_imag = 16'b0000000000000000;
183         assign out6_real = s2;
184         assign out6_imag = s4_imag;
185         assign out7_real = m2;
186         assign out7_imag = ~m5_imag + 1'b1;
187         assign out8_real = s1;
188         assign out8_imag = ~s3_imag + 1'b1;
189 endmodule
```

The above code with Test Bench is available in FFT8_implementation_3 directory of the project GIT repository.

3.3 Analysis

This algorithm calculates the intermediate values in FFT block serially hence reusing the multipliers. Since multipliers are reused , the input bandwidth reduces by half from implementation where 2 multiplier blocks are used. Since the multiplier is LUT based , the usage of LUTs increase , but it reduces the amount number of DSP48A1s which are very scarce in FPGAs.

Figure 3: Simulation In ISIM



4 Validation Script

A python Validation script is written to validate the FFT Verilog Blocks. The python script below uses Numpy library to generate 100 random floating point test inputs and using ISIM command line options simulates all the inputs and validates them with the standard FFT function present in the numpy library of python.

```
1
2  import numpy as np
3  from bitstring import Bits
4  import os
5  implement=raw_input("design to test(allowed values 1,2,3)")
6  working_directory=r"/FFT8_implementation_"+str(implement)
7
8  print("Generating 100 Random Inputs ... ")
9  cmds = ""
10 for i in range(100):
11   input_array=np.random.uniform(low=-8.0, high=8.0, size=(8,))
12   binary_array=[]
13   for inp in input_array:
14     binary_array.append(Bits(int=int(inp*2**8), length=16).bin)
15   cmd="""
16    isim force add {/fft8_tb/inp1} %s -radix bin
17    isim force add {/fft8_tb/inp2} %s -radix bin
18    isim force add {/fft8_tb/inp3} %s -radix bin
19    isim force add {/fft8_tb/inp4} %s -radix bin
20    isim force add {/fft8_tb/inp5} %s -radix bin
21    isim force add {/fft8_tb/inp6} %s -radix bin
22    isim force add {/fft8_tb/inp7} %s -radix bin
23    isim force add {/fft8_tb/inp8} %s -radix bin
24    isim force add {/fft8_tb/clk} 1 -radix bin -value 0 -radix
25    bin -time 2500 ps -repeat 5 ns -cancel 1 us
26    isim force add {/fft8_tb/rst} 1 -radix bin -cancel 20 ns
27    isim force add {/fft8_tb/rst} 0 -radix bin -time 20 ps
28    -cancel 1 us
29    run
30    dump
31   """%tuple(binary_array)
32   cmds=cmds+cmd
33
34 os.chdir(working_directory)
35 f=open("inp.test","w")
36 f.write(cmds)
37 f.close()
38 print("Simulating all the inputs using ISIM...")
39 cmd='"'+working_directory+r'/fft8_tb_isim_beh.exe" < inp.test > out
       .test'
40 os.system(cmd)
41 f=open("out.test","r")
42 values=[]
43 inp={}
44 out={}
45 for line in f.readlines():
46  print line
```

```
 if "Signal:" in line:
  out[line.strip().split("{")[1].split("}")[0].split("[")[0].strip
    ()]=line.strip().split(":")[-1].strip()
 if "Variable:" in line:
  inp[line.strip().split("{")[1].split("}")[0].split("[")[0].strip
    ()]=line.strip().split(":")[-1].strip()
 if "{rst}" in line.strip():
  values.append([inp,out])
  inp={}
  out={}

f.close()

print("Verifying FFT output with the actual values...")
correct=0
count=0
for val in values:
  count=count+1
  inp = np.asarray([Bits(bin=val[0]['inp'+str(i)]).int/(2.0**8) for
     i in range(1,9)])
  out1 = np.array([Bits(bin=val[1]['out'+str(i)+"_real"]).int
    /(2.0**8) for i in range(1,9)])
  out2 = np.array([Bits(bin=val[1]['out'+str(i)+"_imag"]).int
    /(2.0**8) for i in range(1,9)])
  out=out1 + 1j*out2
  if np.allclose(out,np.fft.fft(inp),1e-2):
   correct=correct+1

print(str(correct)+r"/"+str(count)+" Correct ...")
```

4.1 Requirements

- ¿=Python 2.7
- bitstring library
- ISIM simulator

4.2 Sample Usage

Clone the GIT repository from the URL given in the project link section, and run validate.py script to validate the designs. The validate.py file is present in the root folder of the GIT repository.Sample usage of the script is shown in figure 4.

5 Project URL

Project GIT repository: https://github.com/Mohankumariitm/Dual-Degree-Project
Use Git or checkout with SVN using the web URL.

Figure 4: Python 2.7

6 Design Summary

Below is the resource utilization summary of the above implementations.

Logic Utilization	Implementation 1	Implementation 2	Implementation 3
Number of Slice Registers	47	317	357
Number of Slice LUTs	337	374	456
Number of fully used LUT-FF pairs	47	138	188
Number of BUFG/BUFGCTRLs	1	1	1
Number of DSP48A1s	2	1	0

7 To Do (in upcoming months)

: https://www.embedded.com/design/connectivity/4403178/Doing-Hartley-Smartly

- Implement FFT using Doing-Hartley-Smartly Method, and Analyze the resource usage
- Implement Variations in Doing-Hartley-Smartly Method by reusing resources to reduce the LUTs.