

-What is HLS

High-level Synthesis, what does synthesis mean here?

Synthesis is basically transforming our design from a higher level of abstraction to lower level. Usually, till now we were writing our code in the System Verilog code which is basically a hardware description language (higher level abstraction) and the output in Vivado on synthesis was the Gate-level implementation of the logic we wrote code for (lower level of abstraction or realization), but the problem in this is that to design complex logics like video or image processing it would be very tough as the emphasis in HDLs is on the functionality of the required specifications and not much on exact hardware model that results from it therefore for these types of complex tasks we would be left wanting for better power, area and time specifications, also as we ourselves experienced it a lot of time was required in verification and any small changes which we applied would also require a lot of time to be verified (debugging) also if we need to implement something for which our code is written in FPGA compatible library on a ASIC then this change is not accounted for in the traditional HDLs

So what exactly does High-Level Synthesis do; Clearly we see that many problems arise when our logic to be implemented is complex in nature, so to overcome these shortcomings of HDLs we move to HLS which is basically a methodology which takes high level languages (like c, c++) as the input / algorithmic description of our design and produces corresponding result as FSMs, data paths which is then further processed into HDL based Register Transfer level (RTL) netlist.

-Why HLS?

Say we want to implement a D-Flip Flop then in c it would simply be something like defining a D-latch
Void(int din, int& dout)

```
{  
  
    dout=din  
  
}
```

This can be used in all the cases like synchronous, asynchronous, reset D-flip flop by minimal changes to this code, this is what we do in HLS where we code in the High level language and by running standard testbenches verify the logic and clearly this is much more readable than our standard Verilog implementation of the same. Also we can't really directly modify our D-latch as effectively and efficiently as changing our C code, therefore we can focus more on our complex logic than the architectural constraints (basically tell the system what to do rather than how it does it, which can be controlled/influenced by us at any stage), therefore our main problem of complex logic implementation is 'almost' solved

-What are its advantages and limitations

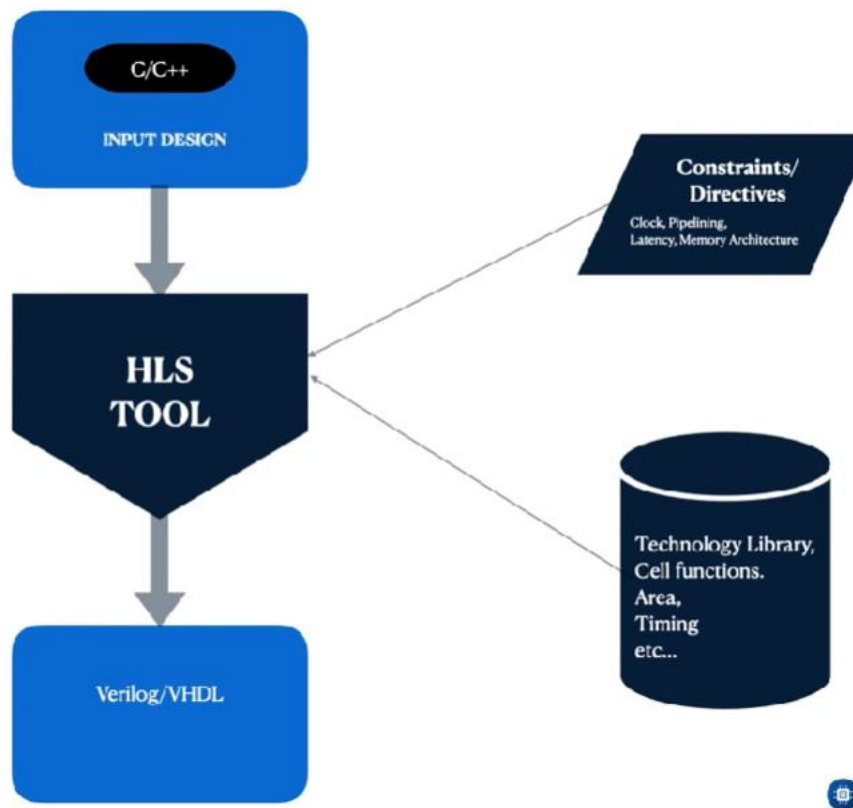
Much easier to write code in Higher level language than Verilog, technology independence (i.e same code can be used to implement both FPGA version as well as ASIC version), all small or large changes can be implemented with ease and time required to complete the synthesis is much lower, trade-off between timing and area constraints can be efficiently controlled by HLS, reusability is much better (i.e flexibility is more)

But the problem with higher abstraction is that the performance of the tasks can be very inefficient, that is HLS does produce the appropriate RTL but it is inefficient (this depends more on user skills than HLS) this is why we said 'almost solved and not completely solved, precise control of internal clocks, delays etc is not as easy as in HDLs, the generated HDL code can be very hard to read and understand and consequently debugging in the RTL level becomes tough, HLS may generate different hardware architectures depending on the input code (again based on users experience this can be eliminated)

-What are its use cases?

Used in, DSP applications (like fir, iir filters, image and video processing), ML algorithms, pipelining, encryption and decryption, basically all complex algorithms.

Sources:- [High Level Synthesis - Part 1 - Introduction | VLSI Concepts](#), [HLS Languages for FPGA Design: Pros, Cons, and Tips, Kinda Technical | A Guide to VHDL - Advantages and Disadvantages of HLS](#), [High Level Synthesis in VLSI. With the advancements in the digital... | by Harshada Belgi | logic-synthesis-in-vlsi | Medium](#)



3/06/25

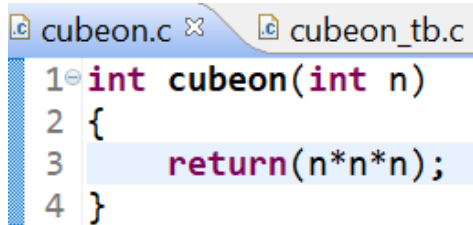
Basic Programs

A basic illustration of working of HLS with example of cube of a number

*Name your top function same as your function here cubeon

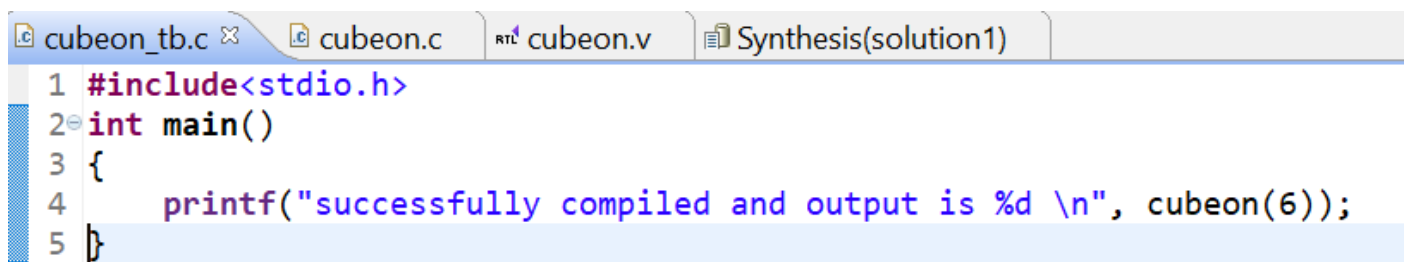
*In parts select family as Artix-7, sub part as cpg 236 and speed grade -1

Here is a basic program to calculate cube of a number in C



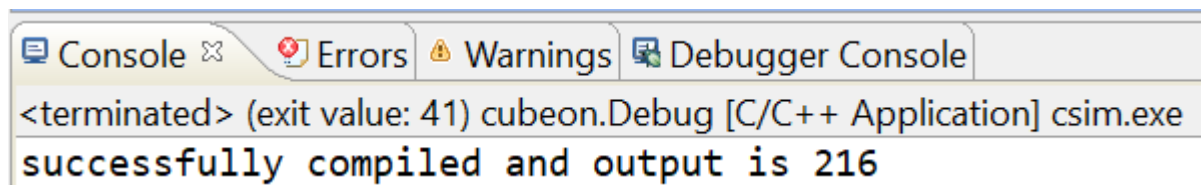
```
1 int cubeon(int n)
2 {
3     return(n*n*n);
4 }
```

Now the testbench for this program is as follows I've also used print statement to verify my output



```
1 #include<stdio.h>
2 int main()
3 {
4     printf("successfully compiled and output is %d \n", cubeon(6));
5 }
```

Output of the simulation is:



```
<terminated> (exit value: 41) cubeon.Debug [C/C++ Application] csim.exe
successfully compiled and output is 216
```

Here is the performance outputs on synthesis

Performance Estimates

- Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	1.27	1.25
- Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	none
0	0	0	0	

Detail

Instance

Loop

Utilization Estimates

Summary					
Name	BRAM_18K	DSP48E	FF	LUT	
DSP	-	-	-	-	
Expression	-	-	0	39	
FIFO	-	-	-	-	
Instance	-	-	-	-	
Memory	-	-	-	-	
Multiplexer	-	-	-	-	
Register	-	-	-	-	
Total	0	0	0	39	
Available	100	90	41600	20800	
Utilization (%)	0	0	0	~0	

Here is the output Verilog file after synthesis

```
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="cubeon,hls_ip_2017_4" *)

module cubeon (
    ap_start,
    ap_done,
    ap_idle,
    ap_ready,
    n,
    ap_return
);

input  ap_start;
output ap_done;
output ap_idle;
output ap_ready;
input  [31:0] n;
output [31:0] ap_return;

assign ap_done = ap_start;

assign ap_idle = 1'b1;

assign ap_ready = ap_start;

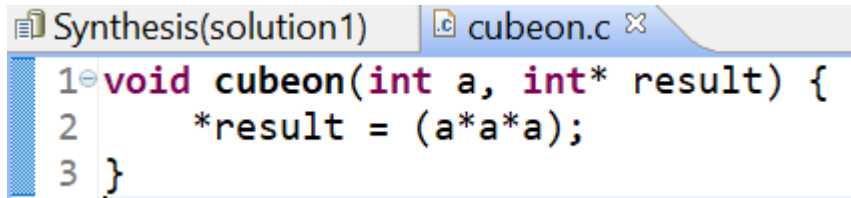
assign ap_return = (n ^ 32'd3);

endmodule //cubeon
```

Major drawback in HLS as we saw today was that we couldn't take input by scanf and typically take inputs during runtime itself, this is due to the fact that you can't determine the size of the input at the time of execution.

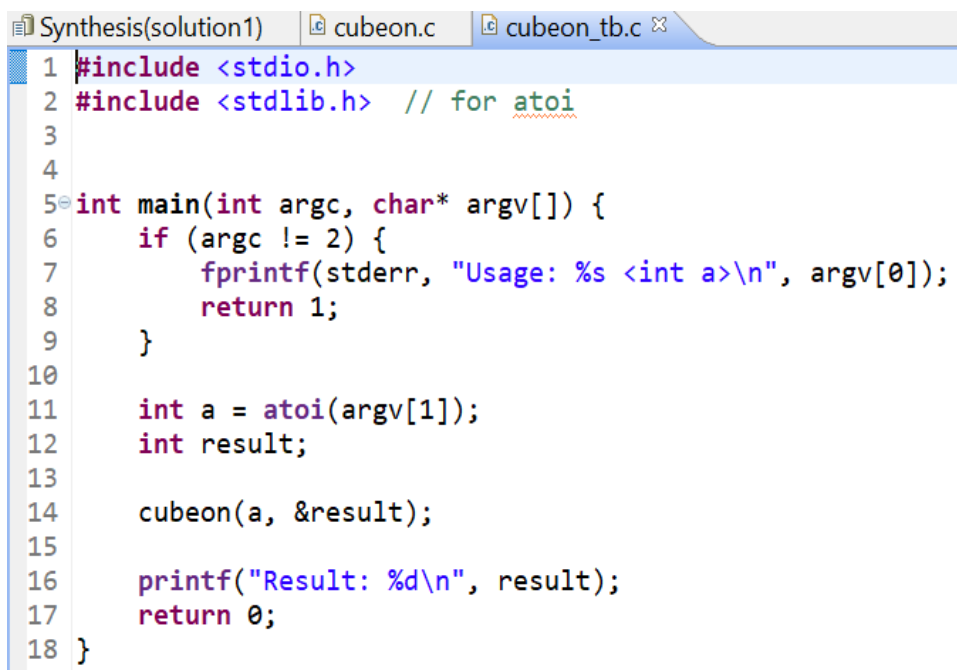
Hence we move to the next best option that is command line input which is still possible since size can be determined before execution during compilation hence in our example of cubing a number.

-This would be our main function: as you can see it's almost the same as the one we used during forced input



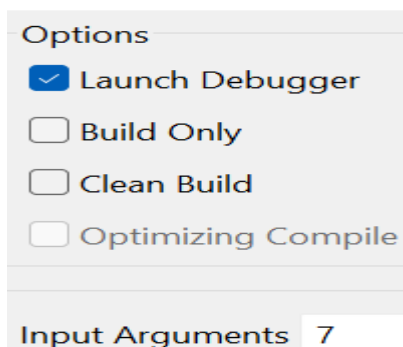
```
Synthesis(solution1) cubeon.c x
1 void cubeon(int a, int* result) {
2     *result = (a*a*a);
3 }
```

-This would be the test bench: completely different as we see we have an argc which is basically used to check number of input characters and argv array which is storing our input 'parameters' since the argv array is a character datatype array we need to convert it to integers as our input is in integer type therefore, we use 'atoi' and then usual computation is taking place.



```
Synthesis(solution1) cubeon.c cubeon_tb.c x
1 #include <stdio.h>
2 #include <stdlib.h> // for atoi
3
4
5 int main(int argc, char* argv[]) {
6     if (argc != 2) {
7         fprintf(stderr, "Usage: %s <int a>\n", argv[0]);
8         return 1;
9     }
10
11     int a = atoi(argv[1]);
12     int result;
13
14     cubeon(a, &result);
15
16     printf("Result: %d\n", result);
17     return 0;
18 }
```

-Input is given during compilation as follows



Options

☒ Launch Debugger

☐ Build Only

☐ Clean Build

☐ Optimizing Compile

Input Arguments 7

-This is the output of variables during compilation in the debug stage

(x)= Variables ✕ Breakpoints 1010 0101 Registers Expressions Modules		
Name	Type	Value
(x)= argc	int	2
> ➔ argv	char **	0xcf0e50
(x)= a	int	7
(x)= result	int	343

-And this is the console output

```

Console ✕ Tasks
<terminated> (exit value: 0)
Result: 343

```

-Verilog code output will remain same as that when forced inputs were given.

-Synthesis report also remains the same as no logical change has been made.

-The other way to take the input from users is via Files, ie we make a file which has a certain number of inputs and the corresponding expected outputs, we run our program to verify whether the actual outputs match the expected outputs for the corresponding inputs, this method is mostly used for large data's.

The testbench is :

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_TRANS 10 // a macros to specify number of inputs

int main()
{
    int a[NUM_TRANS], c_expected[NUM_TRANS];
    int c[NUM_TRANS];
    int i;

    FILE* fp = fopen("inpn.txt", "r");
    if (!fp)
    {
        perror("Error opening inpn.txt");
        return 1;
    }

    for (i = 0; i < NUM_TRANS; i++)
    {
        if (fscanf(fp, "%d %d", &a[i], &c_expected[i]) != 2)
        {
            fprintf(stderr, "Invalid data format on line %d\n", i + 1);
            fclose(fp);
            return 1;
        }
    }
    fclose(fp);
    int retval = 0; // a variable to help specify if the process failed or passed
    for (i = 0; i < NUM_TRANS; i++) {
        cubeoninfile(a[i], &c[i]);

        printf("Test %d: a=%d | Expected c=%d | Actual c=%d --> ",
            i + 1, a[i], c_expected[i], c[i]);

        if (c[i] == c_expected[i]) {
            printf("PASS\n");
        } else {
            printf("FAIL\n");
            retval = 1;
        }
    }

    FILE* fw = fopen("result.txt", "w");
    if (!fw) {
        perror("Error opening result.txt");
        return 1;
    }

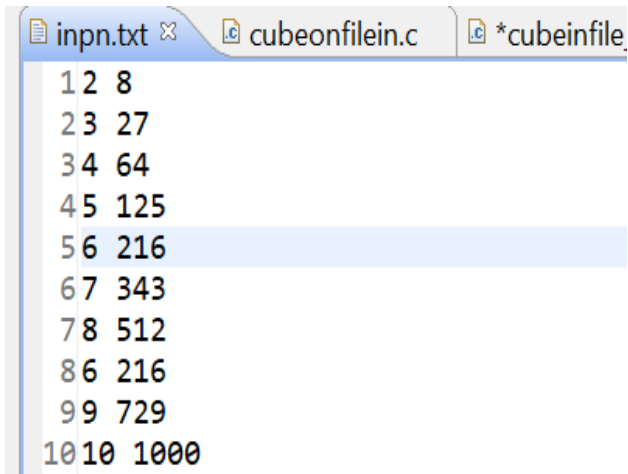
    for (i = 0; i < NUM_TRANS; i++) {
        fprintf(fw, "%d\n", c[i]);
    }
    fclose(fw);
    FILE* fa = fopen("inpn.txt", "a");
    if (!fa) {
        perror("Error opening result.txt");
        return 1;
    }
    fprintf(fw, "\n the calculated out is :\n");

    for (i = 0; i < NUM_TRANS; i++) {
        fprintf(fa, "%d\n", c[i]);
    }
    fclose(fa);

    return retval;
}
```

Clearly here we have used a pointer to a file for all three of its use cases of reading, writing and appending

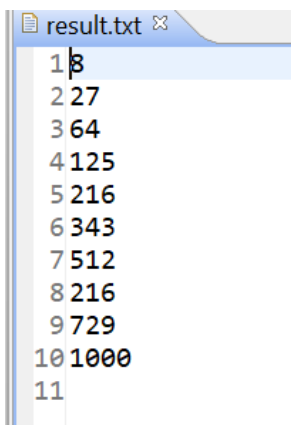
Our initial input read file is



The screenshot shows a text editor with three tabs: 'inpn.txt', 'cubeonfilein.c', and '*cubeinfile.'. The 'inpn.txt' tab is active and displays a list of numbers from 1 to 10, each followed by its cube. The text is as follows:

```
1 2 8
2 3 27
3 4 64
4 5 125
5 6 216
6 7 343
7 8 512
8 6 216
9 9 729
10 10 1000
```

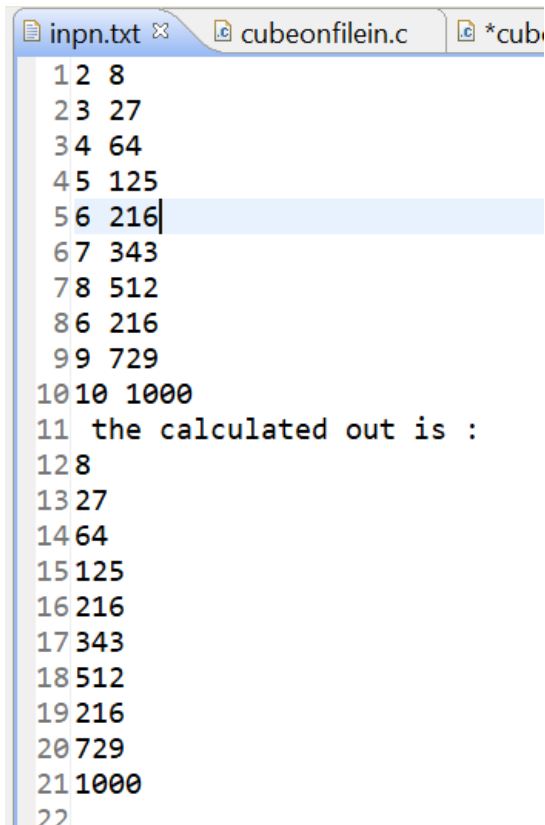
Written file output is



The screenshot shows a text editor with one tab: 'result.txt'. The tab is active and displays a list of numbers from 1 to 11, each followed by its cube. The text is as follows:

```
1 8
2 27
3 64
4 125
5 216
6 343
7 512
8 216
9 729
10 1000
11
```

Appended output is :



The screenshot shows a text editor with three tabs: 'inpn.txt', 'cubeonfilein.c', and '*cubeinfile.'. The 'inpn.txt' tab is active and displays the original input data followed by a new line of text. The text is as follows:

```
1 2 8
2 3 27
3 4 64
4 5 125
5 6 216
6 7 343
7 8 512
8 6 216
9 9 729
10 10 1000
11 the calculated out is :
12 8
13 27
14 64
15 125
16 216
17 343
18 512
19 216
20 729
21 1000
22
```


Then we tried for the first time to run the Verilog code that was generated in HLS in Vivado with a new testbench written in the Verilog format only(remember HLS only produces the Verilog code not the Testbench)

-This was the generated Verilog code

```

module cubeon (
    ap_clk,
    ap_rst,
    ap_start,
    ap_done,
    ap_idle,
    ap_ready,
    a,
    result,
    result_ap_vld
);

parameter    ap_ST_fsm_state1 = 2'd1;
parameter    ap_ST_fsm_state2 = 2'd2;

input  ap_clk;
input  ap_rst;
input  ap_start;
output ap_done;
output ap_idle;
output ap_ready;
input  [31:0] a;
output [31:0] result;
output  result_ap_vld;

reg ap_done;
reg ap_idle;
reg ap_ready;
reg result_ap_vld;

(* fsm_encoding = "none" *) reg [1:0] ap_CS_fsm;
wire ap_CS_fsm_state1;
wire signed [31:0] tmp1_fu_27_p2;
reg signed [31:0] tmp1_reg_43;
wire ap_CS_fsm_state2;
wire signed [31:0] tmp1_fu_27_p0;
wire signed [31:0] tmp1_fu_27_p1;
wire signed [31:0] tmp1_fu_33_p1;
reg [1:0] ap_NS_fsm;

// power-on initialization
initial begin
    #0 ap_CS_fsm = 2'd1;
end

always @ (posedge ap_clk) begin
    if (ap_rst == 1'b1) begin
        ap_CS_fsm <= ap_ST_fsm_state1;
    end else begin
        ap_CS_fsm <= ap_NS_fsm;
    end
end

always @ (posedge ap_clk) begin
    if (((ap_start == 1'b1) & (1'b1 == ap_CS_fsm_state1))) begin
        tmp1_reg_43 <= tmp1_fu_27_p2;
    end
end

always @ (*) begin
    if ((1'b1 == ap_CS_fsm_state2)) begin
        ap_done = 1'b1;
    end else begin
        ap_done = 1'b0;
    end
end

always @ (*) begin
    if (((ap_start == 1'b0) & (1'b1 == ap_CS_fsm_state1))) begin
        ap_idle = 1'b1;
    end else begin
        ap_idle = 1'b0;
    end
end

always @ (*) begin
    if ((1'b1 == ap_CS_fsm_state2)) begin
        ap_ready = 1'b1;
    end else begin
        ap_ready = 1'b0;
    end
end

always @ (*) begin
    if ((1'b1 == ap_CS_fsm_state2)) begin
        result_ap_vld = 1'b1;
    end else begin
        result_ap_vld = 1'b0;
    end
end

always @ (*) begin
    case (ap_CS_fsm)
        ap_ST_fsm_state1 : begin
            if (((ap_start == 1'b1) & (1'b1 == ap_CS_fsm_state1))) begin
                ap_NS_fsm = ap_ST_fsm_state2;
            end else begin
                ap_NS_fsm = ap_ST_fsm_state1;
            end
        end
        ap_ST_fsm_state2 : begin
            ap_NS_fsm = ap_ST_fsm_state1;
        end
    endcase
end

assign ap_CS_fsm_state1 = ap_CS_fsm[32'd0];
assign ap_CS_fsm_state2 = ap_CS_fsm[32'd1];

assign result = ($signed(tmp1_reg_43) * $signed(tmp1_fu_33_p1));

assign tmp1_fu_27_p0 = a;
assign tmp1_fu_27_p1 = a;
assign tmp1_fu_27_p2 = ($signed(tmp1_fu_27_p0) * $signed(tmp1_fu_27_p1));
assign tmp1_fu_33_p1 = a;

endmodule //cubeon

```

-The testbench was as shown

```
timescale 1ns/1ps

module cubeon_tb;

    // Inputs
    reg ap_clk;
    reg ap_rst;
    reg ap_start;
    reg [31:0] a;

    // Outputs
    wire ap_done;
    wire ap_idle;
    wire ap_ready;
    wire [31:0] result;
    wire result_ap_vld;

    // Instantiate the Unit Under Test (UUT)
    cubeon uut (
        .ap_clk(ap_clk),
        .ap_rst(ap_rst),
        .ap_start(ap_start),
        .a(a),
        .ap_done(ap_done),
        .ap_idle(ap_idle),
        .ap_ready(ap_ready),
        .result(result),
        .result_ap_vld(result_ap_vld)
    );

    // Clock generation: 10ns period
    initial ap_clk = 0;
    always #5 ap_clk = ~ap_clk;

    // Stimulus process
    initial begin
        $display("Starting cubeon testbench...");
        $dumpfile("cubeon_tb.vcd");
        $dumpvars(0, cubeon_tb);

        // Initialize signals
        ap_rst = 1;
        ap_start = 0;
        a = 0;

        // Apply reset
        #20;
        @(negedge ap_clk);
        ap_rst = 0;

        // Run tests
        run_test(32'd3); // 3^3 = 27
        run_test(32'd5); // 5^3 = 125
        run_test(32'd0); // 0^3 = 0
        run_test(32'd10); // 10^3 = 1000
        run_test(32'd1); // 1^3 = 1

        $display("All tests completed.");
        $stop;
    end

    // Task to run a single test
    task run_test(input [31:0] value);
        begin
            @(negedge ap_clk);
            a = value;
            ap_start = 1;
            @(negedge ap_clk);
            ap_start = 0;

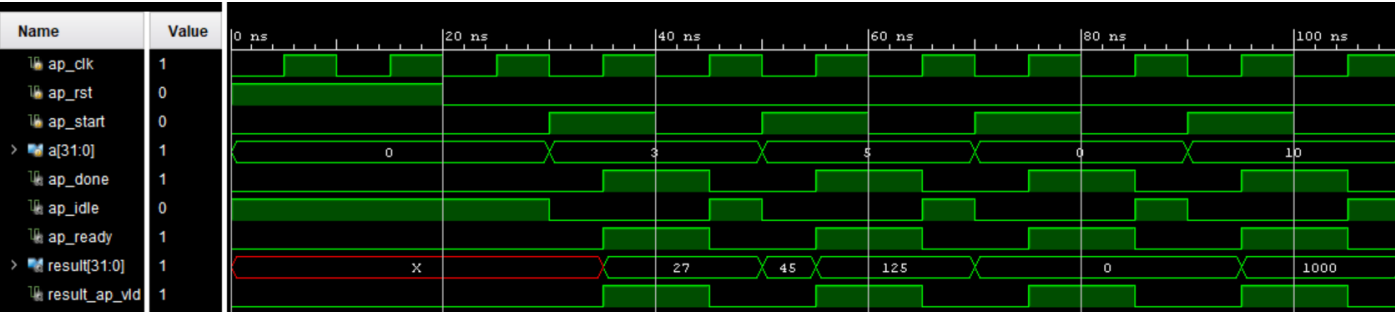
            // Wait for result to become valid
            wait (result_ap_vld == 1);
            @(posedge ap_clk); // One extra cycle for stable output

            $display("Input a = %0d, Result = %0d (Expected: %0d)", value, result, value * value * value);
        end
    endtask
endmodule
```

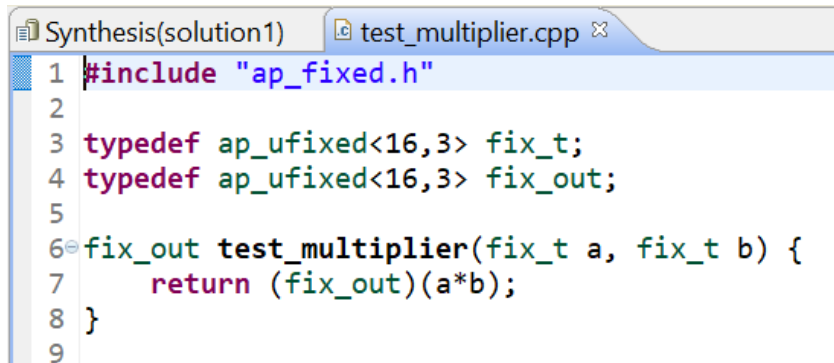
-This was the output on the console

```
Starting cubeon testbench...
Input a = 3, Result = 27 (Expected: 27)
Input a = 5, Result = 125 (Expected: 125)
Input a = 0, Result = 0 (Expected: 0)
Input a = 10, Result = 1000 (Expected: 1000)
Input a = 1, Result = 1 (Expected: 1)
All tests completed.
```

-The waveform was:



- 1) Signed integer : runs properly for our program
- 2) Unsigned integer : again no problem
- 3) Char: Worked fine for its application programs
- 4) Fixed-point : a special case (kind of) of float where we fix the length of the integer part and the decimal part this works only in c++ and although its hardware utilization is improved it is highly inaccurate in its structure (see limitations of Fixed point)



```
1 #include "ap_fixed.h"
2
3 typedef ap_ufixed<16,3> fix_t;
4 typedef ap_ufixed<16,3> fix_out;
5
6 fix_out test_multiplier(fix_t a, fix_t b) {
7     return (fix_out)(a*b);
8 }
9
```

-So we need to be wise in choosing our data type according to our computation requirements and try using Fixed point precision as a replacement to float if you do this then the only way to use fixed point is via the c++ language so be careful

-To run on fpga you would have to force the inputs to be of the length of 8 each max on fpga is 16 therefore each of 8 but output can be 16 as one output can take max 16 lights on fpga therefore I have modified the code to aptly run this way

-Limitations of using the fixed point form is that sometimes due to the fixed nature of the data type our input can have error like example 2.1 for input of ap_fixed<8,3> our actual interpreted input will be 2.09833 but for ap_fixed<8,2> it will be 2.10112 so then error approximation is required to get the required answer

6/6/25

FLOAT IEEE 754

Code:

```
Synthesis(solution1) floater_tcl.c floater_t
1 float floater_tcl(float a, float b)
2 {
3     return(a*b);
4 }
5
```

Synthesis report:

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	3	143	140
Memory	-	-	-	-
Multiplexer	-	-	-	27
Register	-	-	4	-
Total	0	3	147	167
Available	100	90	41600	20800
Utilization (%)	0	3	~0	~0
Detail				

Float : real challenge arises as it formed 2 verilog files and a tcl file was being generated which was a challenge to run and what was observed on running these files was that precision was only up to like 4 digits after decimal and the remaining part was very inaccurate and when I tried to increase the number of points after the decimal and it was noted that just to increase accuracy by 3 decimal points it would required almost double the number of the hardware resources that it took for initial case of 4 digits after decimal place hence it is very inefficient in its hardware usage.

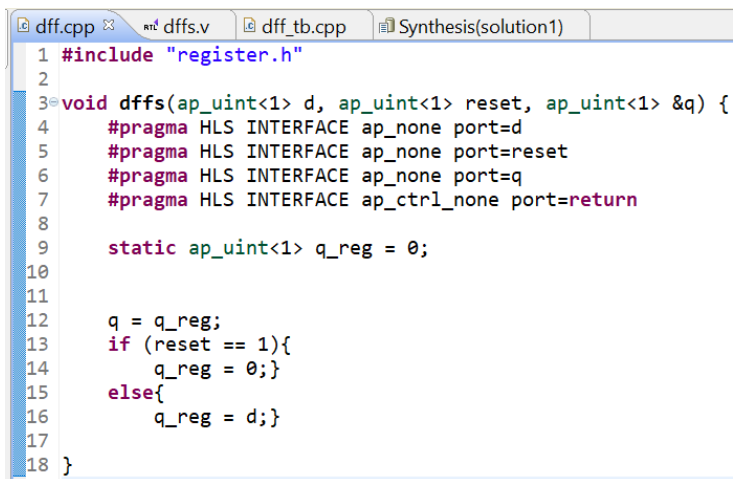
Another way is to take the inputs as float and internally convert it to integer type and do the operations on the converted values later convert this code back to the float type

9/6/25

D Flip Flop

First sequential Circuit that we implement

Code was



```
1 #include "register.h"
2
3 void dffs(ap_uint<1> d, ap_uint<1> reset, ap_uint<1> &q) {
4     #pragma HLS INTERFACE ap_none port=d
5     #pragma HLS INTERFACE ap_none port=reset
6     #pragma HLS INTERFACE ap_none port=q
7     #pragma HLS INTERFACE ap_ctrl_none port=return
8
9     static ap_uint<1> q_reg = 0;
10
11
12     q = q_reg;
13     if (reset == 1){
14         q_reg = 0;}
15     else{
16         q_reg = d;}
17
18 }
```

*see the way we have used `q=q_reg` it has been used before the if block, I tried to implement it after the if block (usually this was the way in SV) but the whole program was considered as a combinational logic and hence we were not even able to run it, the change of ordering of the assignment **DOES** matter in c.

The verilog code was

```

2 module dffs (
3     ap_clk,
4     ap_rst,
5     d_V,
5     reset_V,
7     q_V
3 );
9
10 parameter    ap_ST_fsm_state1 = 1'd1;
1
2 input    ap_clk;
3 input    ap_rst;
4 input    [0:0] d_V;
5 input    [0:0] reset_V;
5 output    [0:0] q_V;
7
3 reg    [0:0] q_reg_V;
3 wire    [0:0] p_d_V_fu_60_p2;
3 (* fsm_encoding = "none" *) reg    [0:0] ap_CS_fsm;
1 wire    ap_CS_fsm_state1;
2 wire    [0:0] not_reset_V_fu_54_p2;
3 reg    [0:0] ap_NS_fsm;
4
5 // power-on initialization
5 initial begin
7 #0 q_reg_V = 1'd0;
3 #0 ap_CS_fsm = 1'd1;
3 end
3
1 always @ (posedge ap_clk) begin
2     if (ap_rst == 1'b1) begin
3         ap_CS_fsm <= ap_ST_fsm_state1;
4     end else begin
5         ap_CS_fsm <= ap_NS_fsm;
5     end
3
3 always @ (*) begin
4     case (ap_CS_fsm)
5         ap_ST_fsm_state1 : begin
8             ap_NS_fsm = ap_ST_fsm_state1;
9         end
10        default : begin
11            ap_NS_fsm = 'bx;
12        end
13    endcase
14 end
15
16 assign ap_CS_fsm_state1 = ap_CS_fsm[32'd0];
17
18 assign not_reset_V_fu_54_p2 = (reset_V ^ 1'd1);
19
20 assign p_d_V_fu_60_p2 = (not_reset_V_fu_54_p2 & d_V);
21
22 assign q_V = q_reg_V;
23
24 endmodule //dffs

```

Here is the Synthesis Report: clearly two registers are being used which indicates the sequential nature of the d flip flop being implemented.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	0.94	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
0	0	0	0	none

Detail

Instance

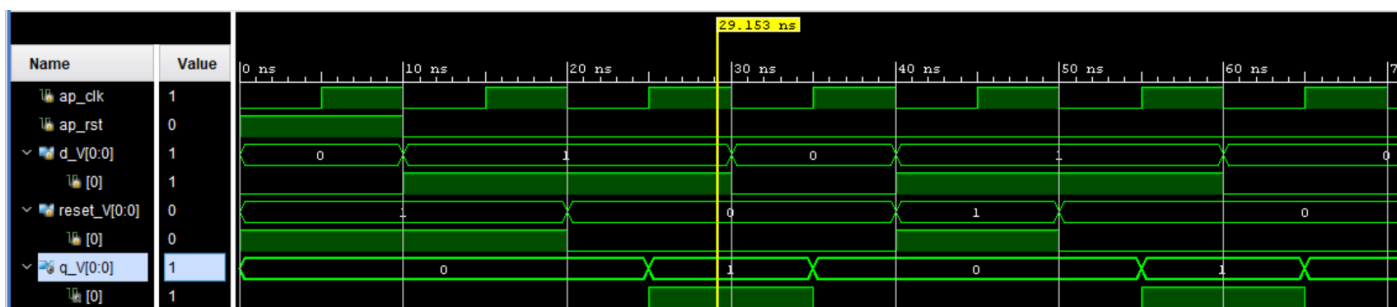
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	16
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
Total	0	0	2	16
Available	100	90	41600	20800
Utilization (%)	0	0	~0	~0

Simulated graph is :



C code:

```

1 #include <ap_int.h>
2 #include <hls_stream.h>
3 // Fixed-point type for I/Q values
4 typedef ap_int<16> symbol_type;
5 // Standard QPSK constellation values (normalized to 0.7071 )
6 // Scaled for 16-bit fixed point representation (Q15 format)
7 // 0.7071 * 32767 = 23170, -0.7071 * 32768 = -23170
8 const symbol_type QPSK_I_VALUES[4] = { 23170, -23170, -23170, 23170}; // I values
9 const symbol_type QPSK_Q_VALUES[4] = { 23170, 23170, -23170, -23170}; // Q values
10 /**
11  * @brief QPSK Constellation mapper function
12  *
13  * @param input_bytes Input byte array to map to QPSK constellation points
14  * @param output_symbols_I I-channel output symbols
15  * @param output_symbols_Q Q-channel output symbols
16  * @param num_bits Number of bits to process (should be even for QPSK) */
17 void constellation5(
18     ap_uint<8> *input_bytes,
19     symbol_type *output_symbols_I,
20     symbol_type *output_symbols_Q,
21     unsigned int num_bits
22 ) {
23     #pragma HLS INTERFACE m_axi port=input_bytes offset=slave depth=256
24     #pragma HLS INTERFACE m_axi port=output_symbols_I offset=slave depth=256
25     #pragma HLS INTERFACE m_axi port=output_symbols_Q offset=slave depth=256
26     #pragma HLS INTERFACE s_axilite port=num_bits bundle=AXILiteS
27     #pragma HLS INTERFACE s_axilite port=return bundle=AXILiteS
28     // QPSK: Process 2 bits per symbol
29     unsigned int num_symbols = (num_bits + 1) / 2; // Ceiling division by 2
30     for (unsigned int i = 0; i < (num_symbols + 3) / 4; i++) { // Process bytes
31         ap_uint<8> byte_val = input_bytes[i];
32         // Process each 2-bit group in the byte
33         for (int j = 0; j < 4 && (i*4 + j) < num_symbols; j++) {
34             // Extract 2 bits for QPSK
35             ap_uint<2> symbol_idx = (byte_val >> (6 - j*2)) & 0x3;
36             // Map to I/Q constellation point
37             output_symbols_I[i*4 + j] = QPSK_I_VALUES[symbol_idx];
38             output_symbols_Q[i*4 + j] = QPSK_Q_VALUES[symbol_idx];

```

- **Purpose:**
Maps input bits to QPSK constellation points (I/Q values) for digital communication.
- **Input:**
 - Bitstream packed as bytes.
 - Each QPSK symbol uses 2 bits.
- **Output:**
 - Two arrays: one for I (in-phase), one for Q (quadrature) components.

- **Constellation Mapping:**

```
const symbol_type QPSK_I_VALUES[4] = {23170, -23170, -23170, 23170}; // Gray coded
```

```
const symbol_type QPSK_Q_VALUES[4] = {23170, 23170, -23170, -23170};
```

The mapping here is done in whole numbers which is just a scaled version of the decimal values. i.e. $0.7071 \times 32767 = 23170$.

- **Process:**

- For each symbol, extract 2 bits from the input.
- Use these bits as an index into I/Q lookup tables.
- Store the corresponding I and Q values in the output arrays.

- **Looping:**

- Iterates over the number of symbols (half the number of input bits).
- Handles bit extraction even if symbols cross byte boundaries.

- **Efficiency:**

- Uses lookup tables for fast mapping.
- Suitable for both software simulation and hardware (FPGA) implementation.

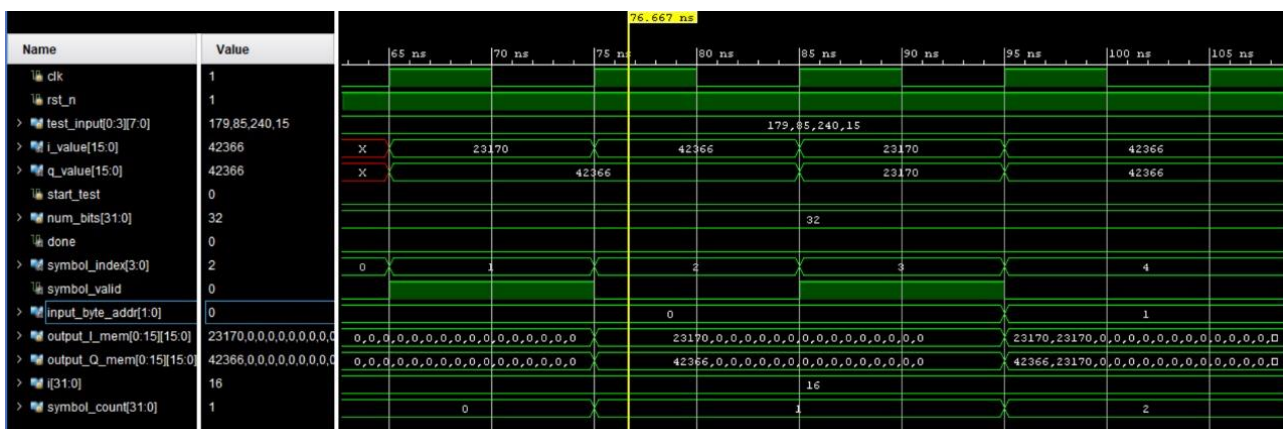
Verilog Code:

The C code generates 5 Verilog files as shown:

```
RTL constellation5_AXILiteS_s_axi.v
RTL constellation5_gmem_m_axi.v
RTL constellation5_gmem2_m_axi.v
RTL constellation5_mubkb.v
RTL constellation5.v
```

The main file is constellation5.v .

Simulation Graph:



Pulse Shaping Filter

C code:

```
Synthesis(solution1) pulse_shape.cpp pulse_shape_tb.cpp
1 #include "ap_fixed.h"
2 #include "hls_math.h"
3
4 #define DATA_LEN 512
5 #define NUM_WEIGHTS 41
6 #define SPS 25
7 #define ALPHA 0.5
8
9 typedef ap_fixed<16, 4> fixed_t;
10
11 void raised_cosine_filter(fixed_t rc[NUM_WEIGHTS]) {
12     const fixed_t PI = fixed_t(3.14159);
13     const fixed_t ALPHA_FIXED = fixed_t(ALPHA);
14     const fixed_t EPS = fixed_t(1e-5);
15     const fixed_t EPS_X = fixed_t(1e-3);
16     const fixed_t SCALE = fixed_t(0.9999);
17
18     int mid = NUM_WEIGHTS / 2;
19     fixed_t sum = 0;
20
21     for (int i = 0; i < NUM_WEIGHTS; i++) {
22         fixed_t idx = fixed_t(i - mid);
23         fixed_t x = SCALE * idx / fixed_t(SPS);
24         fixed_t pi_x = PI * x;
25
26         fixed_t sinc;
27         if (hls::abs(x) < EPS_X)
28             sinc = fixed_t(1.0);
29         else
30             sinc = hls::sin(pi_x) / pi_x;
31
32         fixed_t denom = fixed_t(1.0) - fixed_t(4.0) * ALPHA_FIXED * ALPHA_FIXED * x * x;
33         if (hls::abs(denom) < EPS)
34             denom = EPS;
35
36         // FIXED this line to avoid synthesis error
37         ap_fixed<32, 6> angle = PI * ALPHA_FIXED * x;
38         fixed_t cos_part = hls::cos(angle);
39
40         rc[i] = sinc * (cos_part / denom);
41         sum += rc[i];
42     }
43
44     for (int i = 0; i < NUM_WEIGHTS; i++) {
45         rc[i] = rc[i] / sum;
46     }
47 }
48
49 void convolve(const fixed_t data[DATA_LEN], const fixed_t filter[NUM_WEIGHTS], fixed_t result[DATA_LEN]) {
50     int mid = NUM_WEIGHTS / 2;
51
52     for (int i = 0; i < DATA_LEN; i++) {
53         fixed_t acc = 0;
54         for (int j = 0; j < NUM_WEIGHTS; j++) {
55             int k = i - mid + j;
56             if (k >= 0 && k < DATA_LEN)
57                 acc += data[k] * filter[j];
58         }
59         result[i] = acc;
60     }
61 }
62
63 void pulse_shape(fixed_t i_data[DATA_LEN], fixed_t q_data[DATA_LEN], fixed_t i_out[DATA_LEN], fixed_t q_out[DATA_LEN]) {
64     #pragma HLS INTERFACE s_axilite port=return bundle=CTRL
65     #pragma HLS INTERFACE bram port=i_data
66     #pragma HLS INTERFACE bram port=q_data
67     #pragma HLS INTERFACE bram port=i_out
68     #pragma HLS INTERFACE bram port=q_out
69
70     fixed_t rc_filter[NUM_WEIGHTS];
71     raised_cosine_filter(rc_filter);
72     convolve(i_data, rc_filter, i_out);
73     convolve(q_data, rc_filter, q_out);
74 }
75
```

Input:

- Two arrays:
 - `i_data[DATA_LEN]`: in-phase component
 - `q_data[DATA_LEN]`: quadrature component
- Each array contains baseband samples (fixed-point format)

Output:

- Two arrays:
 - `i_out[DATA_LEN]`: filtered in-phase output
 - `q_out[DATA_LEN]`: filtered quadrature output

Filter Design:

- **Raised Cosine Filter** with:
 - `NUM_WEIGHTS` = 41 taps
 - `ALPHA` = 0.5 roll-off factor
 - `SPS` = 25 samples per symbol
- Coefficients computed with:
 - Normalized `sinc()` function
 - Cosine roll-off term
 - Avoids divide-by-zero using small epsilon
- Normalization step ensures filter gain = 1

Process:

1. Compute `rc_filter[]` coefficients using `raised_cosine_filter()`
2. Convolve `i_data[]` and `q_data[]` with `rc_filter[]` using `convolve()`
3. Store results in `i_out[]`, `q_out[]`

Looping:

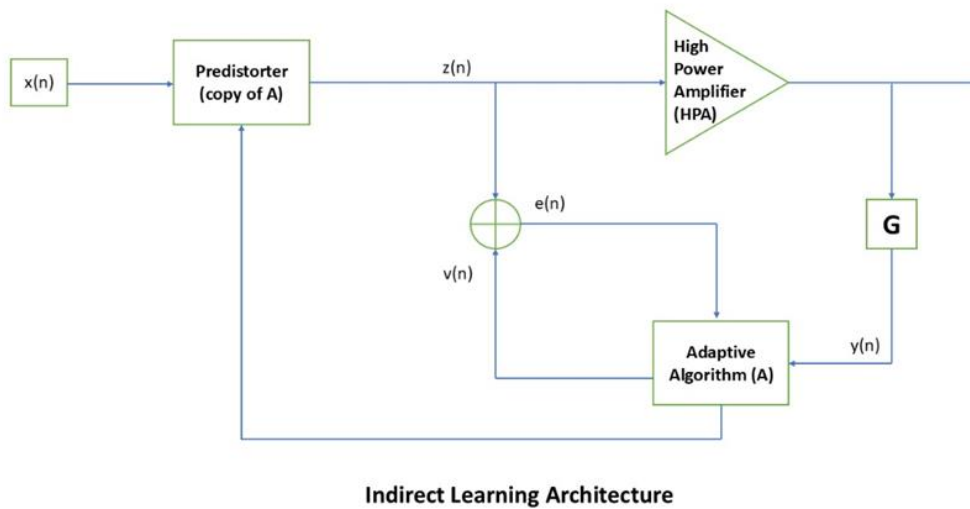
- Filter loop (`NUM_WEIGHTS` times) for each output sample
- Handles signal edges using boundary check
- Accumulates sum of products for FIR convolution

Efficiency:

- Uses fixed-point math for FPGA compatibility
- Coefficients and output fit within 16-bit precision
- Fully synthesizable via Vivado HLS
- Interface-ready via AXI & BRAM pragmas

Pre Distorter

The DPD circuit architecture followed is known as the Indirect Learning Architecture (ILA)



Purpose:

To apply the calculated inverse distortion to the input signal.

Input :

- Pulse shaped I and Q values from the pulse shape filter.
- Weight coefficients generated by the Pre Distortion Algorithm.

Output :

Complex Multiplication product of input signal (I&Q) values and the weights generated by the DPD algorithm.

Process :

- Receive I and Q shaped values from the pulse shape filter.
- Collect the weighted coefficients from the Pre Distortion algorithm block.
- Generate z values for pre distorted outputs.

```
sum_i += w_prev[k][m].real * real_phi[k][m] - w_prev[k][m].imag *  
imag_phi[k][m];  
sum_q += w_prev[k][m].real * imag_phi[k][m] + w_prev[k][m].imag *  
real_phi[k][m];
```

The above assignment is done by complex multiplication.

Efficiency :

Generation of coefficients is done recursively using the LMS method.

PRE-DISTORTION ALGORITHM

1. Orthogonal Polynomial Method:

Purpose :

To generate orthogonal polynomials and the weighted coefficients for digital pre distortion.

Inputs:

Delayed pulse shaped I and Q inputs (x), amplified output (y) from the feedback loop.

Outputs:

Weighted coefficients (w) for RLS applications.

Process:

The delayed signal is subtracted from the feedback output to calculate the error function (e(n)).

```
// Compute error: e = x_ref - y_model
data_t err_i = i_ref - *i_out;
data_t err_q = q_ref - *q_out;
```

```
// Compute |z|^2
phi_t abs2(data_t i, data_t q) {
    return i*i + q*q;
}

// Iterative shifted Legendre polynomial (or similar orthogonal poly)
phi_t iterative_P(int k, phi_t x) {
    phi_t P0 = 1, P1 = x - 1, Pk = 0;
    if (k == 0) return P0;
    if (k == 1) return P1;
    for (int n = 2; n <= k; n++) {
        Pk = ((2*n-1)*(x-1)*P1 - (n-1)*P0)/n;
        P0 = P1;
        P1 = Pk;
    }
    return Pk;
}

// Compute all orthogonal polynomial basis functions for all memory taps
void compute_phi_all(
    const data_t i_in[MEMORY_DEPTH], const data_t q_in[MEMORY_DEPTH],
    phi_t real_phi[K][MEMORY_DEPTH], phi_t imag_phi[K][MEMORY_DEPTH]
) {
    for (int m = 0; m < MEMORY_DEPTH; ++m) {
        phi_t x = abs2(i_in[m], q_in[m]);
        for (int k = 0; k < K; ++k) {
#pragma HLS UNROLL
            phi_t P = iterative_P(k, x); // P_k(|z|^2)
            real_phi[k][m] = i_in[m] * P;
            imag_phi[k][m] = q_in[m] * P;
        }
    }
}
```

The orthogonal polynomial is calculated recursively to generate the polynomial coefficients. The polynomial coefficients and the error function are used to compute the weighted coefficients.

Efficiency :

Use of the RLS method for polynomial coefficient enables faster iterations.
Calculation by LMS method is quick and efficient.
The Mean Squared Error is highly reduced and distortions are neatly fixed.

2. Modified Differential Evolution

Purpose:

To generate a new vector(signal) value for digital predistortion.

Inputs:

Delayed pulse shaped I and Q inputs (x), amplified output (y) from the feedback loop.

Outputs:

Most fit vector value with highest Digital Pre Distortion.

Process :

Generate a population of values from the feedback outputs

```
// 3. Compute DPD output z(n) using best individual
acc_t sum_i = 0, sum_q = 0;
for (int k = 0; k < K; k++) {
#pragma HLS PIPELINE II=1
    for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
#pragma HLS UNROLL
        sum_i += population[best_idx][k][tap].real * real_phi[k][tap] -
            population[best_idx][k][tap].imag * imag_phi[k][tap];
        sum_q += population[best_idx][k][tap].real * imag_phi[k][tap] +
            population[best_idx][k][tap].imag * real_phi[k][tap];
    }
    // --- Scale DPD output by g_dpd_scale (double) ---
    *z_i = data_t(double(sum_i) / g_dpd_scale);
    *z_q = data_t(double(sum_q) / g_dpd_scale);

    // 1. Initialize population
    if (!init_done) {
        for (int i = 0; i < POPULATION_SIZE; i++) {
#pragma HLS UNROLL
            for (int k = 0; k < K; k++) {
#pragma HLS UNROLL
                for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
#pragma HLS UNROLL
                    data_t perturbation_real = (generate_random_float(&
                        rand_seed) - ap_fixed<24,8>(0.5)) * ap_fixed<24,8>(2.0);
                    data_t perturbation_imag = (generate_random_float(&
                        rand_seed) - ap_fixed<24,8>(0.5)) * ap_fixed<24,8>(2.0);
                    if (k == 0 && tap == 0) {
                        population[i][k][tap].real = ap_fixed<24,8>(1.0) +
                            perturbation_real;
                        population[i][k][tap].imag = perturbation_imag;
                    } else {
                        population[i][k][tap].real = perturbation_real;
                        population[i][k][tap].imag = perturbation_imag;
                    }
                }
                fitness[i] = 1000.0f;
            }
        }
        init_done = true;
    }
}
```

The vectors are generated randomly.

Finding the best individual from the population based on fitness.


```

// 2. Find best individual
int best_idx = 0;
data_t best_fitness = fitness[0];
for (int i = 1; i < POPULATION_SIZE; i++) {
#pragma HLS UNROLL
    if (fitness[i] < best_fitness) {
        best_fitness = fitness[i];
        best_idx = i;
    }
}
// Print fitness progress every 10 generations
static int adapt_print_counter = 0;
adapt_print_counter++;
if (adapt_print_counter % 10 == 0) {
    std::cout << "[MDE] Generation " << generation_count << " best
    fitness: " << best_fitness << std::endl;
}

```

Compute z using the best individual based on fitness and cost factor

Output DPD using best individual

```

// 4. Output DPD for the batch using best individual
for (int b = 0; b < BATCH_SIZE; b++) {
    phi_t real_phi[K][MEMORY_DEPTH], imag_phi[K][MEMORY_DEPTH];
    compute_phi_all(i_in_batch[b], q_in_batch[b], real_phi, imag_phi);
    acc_t sum_i = 0, sum_q = 0;
    for (int k = 0; k < K; k++) {
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
            sum_i += population[best_idx][k][tap].real * real_phi[k][tap]
            - population[best_idx][k][tap].imag * imag_phi[k][tap];
            sum_q += population[best_idx][k][tap].real * imag_phi[k][tap]
            + population[best_idx][k][tap].imag * real_phi[k][tap];
        }
    }
    z_i_batch[b] = data_t(double(sum_i) / g_dpd_scale);
    z_q_batch[b] = data_t(double(sum_q) / g_dpd_scale);
}

// 6. Update weights after full DE cycle
generation_count++;
if (generation_count >= MAX_GENERATIONS) {
    for (int k = 0; k < K; k++) {
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
            w[k][tap] = population[best_idx][k][tap];
        }
    }
    generation_count = 0;
}
}

```

The vectors are mutated and evaluated for fitness

```
// 5. DE mutation/crossover/selection (using batch fitness)
for (int i = 0; i < POPULATION_SIZE; i++) {
    int r1 = generate_random_index(i, POPULATION_SIZE, &rand_seed);
    int r2 = generate_random_index(r1, POPULATION_SIZE, &rand_seed);
    int r3 = generate_random_index(r2, POPULATION_SIZE, &rand_seed);

    ccoef_t trial[K][MEMORY_DEPTH];
    for (int k = 0; k < K; k++) {
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
            trial[k][tap].real = population[r1][k][tap].real + F_SCALE *
                (population[r2][k][tap].real - population[r3][k][tap].real);
            trial[k][tap].imag = population[r1][k][tap].imag + F_SCALE *
                (population[r2][k][tap].imag - population[r3][k][tap].imag);

            int cross_rand = generate_random_index(0, 1000, &rand_seed);
            data_t rand_val = (data_t)cross_rand * (data_t)0.001;
            if (rand_val > CR_PROB) {
                trial[k][tap] = population[i][k][tap];
            }
        }
    }

    // Evaluate trial fitness over batch
    data_t trial_total_fitness = 0;
    for (int b = 0; b < BATCH_SIZE; b++) {
        phi_t real_phi[K][MEMORY_DEPTH], imag_phi[K][MEMORY_DEPTH];
        compute_phi_all(i_in_batch[b], q_in_batch[b], real_phi, imag_phi);
        acc_t sum_i = 0, sum_q = 0;
        for (int k = 0; k < K; k++) {
            for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
                sum_i += trial[k][tap].real * real_phi[k][tap] - trial[k][tap].imag * imag_phi[k][tap];
                sum_q += trial[k][tap].real * imag_phi[k][tap] + trial[k][tap].imag * real_phi[k][tap];
            }
        }
        data_t y_i = data_t(double(sum_i) / g_dpd_scale);
    }
}

// 5. DE mutation/crossover/selection (using batch fitness)
for (int i = 0; i < POPULATION_SIZE; i++) {
    int r1 = generate_random_index(i, POPULATION_SIZE, &rand_seed);
    int r2 = generate_random_index(r1, POPULATION_SIZE, &rand_seed);
    int r3 = generate_random_index(r2, POPULATION_SIZE, &rand_seed);

    ccoef_t trial[K][MEMORY_DEPTH];
    for (int k = 0; k < K; k++) {
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
            trial[k][tap].real = population[r1][k][tap].real + F_SCALE *
                (population[r2][k][tap].real - population[r3][k][tap].real);
            trial[k][tap].imag = population[r1][k][tap].imag + F_SCALE *
                (population[r2][k][tap].imag - population[r3][k][tap].imag);

            int cross_rand = generate_random_index(0, 1000, &rand_seed);
            data_t rand_val = (data_t)cross_rand * (data_t)0.001;
            if (rand_val > CR_PROB) {
                trial[k][tap] = population[i][k][tap];
            }
        }
    }

    // Evaluate trial fitness over batch
    data_t trial_total_fitness = 0;
    for (int b = 0; b < BATCH_SIZE; b++) {
        phi_t real_phi[K][MEMORY_DEPTH], imag_phi[K][MEMORY_DEPTH];
        compute_phi_all(i_in_batch[b], q_in_batch[b], real_phi, imag_phi);
        acc_t sum_i = 0, sum_q = 0;
        for (int k = 0; k < K; k++) {
            for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
                sum_i += trial[k][tap].real * real_phi[k][tap] - trial[k][tap].imag * imag_phi[k][tap];
                sum_q += trial[k][tap].real * imag_phi[k][tap] + trial[k][tap].imag * real_phi[k][tap];
            }
        }
        data_t y_i = data_t(double(sum_i) / g_dpd_scale);
    }
}
```

Update of weights after one full DE cycle.

```
    }
    data_t y_i = data_t(double(sum_i) / g_dpd_scale);
    data_t y_q = data_t(double(sum_q) / g_dpd_scale);
    trial_total_fitness += compute_fitness(i_ref_batch[b], q_ref_batch
[b], y_i, y_q);
}
data_t trial_fitness = trial_total_fitness / BATCH_SIZE;

if (trial_fitness < fitness[i]) {
    for (int k = 0; k < K; k++) {
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
            population[i][k][tap] = trial[k][tap];
        }
    }
    fitness[i] = trial_fitness;
}
}

// 6. Update weights after full DE cycle
generation_count++;
if (generation_count >= MAX_GENERATIONS) {
    for (int k = 0; k < K; k++) {
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
            w[k][tap] = population[best_idx][k][tap];
        }
    }
    generation_count = 0;
}
}
```

Inputs :

It takes discrete digital inputs from the pre-distorter.

Outputs :

It outputs discrete analog values.

Process:

The discrete input values are scaled for the expected 8-bit output.

While the inputs range lies in $[-1,1]$ the outputs are 8 bits in size.

Hence the values are scaled to adjust to the 8 bit values with the range in $[-128,127]$.

```
// Multi-bit DAC with channel select (8-bit output)
void dac_multibit_with_select(data_ty din, ap_fixed<24,8> &dout, bool
channel_select) {
#pragma HLS INLINE
    // Scale input from [-1, 1) to [-128, 127]
    data_ty quantized = din * data_ty(128);
    if (quantized > 127) quantized = 127;
    if (quantized < -128) quantized = -128;
    dout = ap_fixed<24,8>(quantized);
    printf( "dout value is %f",dout.to_float());
}
```

Values which fall outside the range are quantized either to -128 or 127.

This module is used so that the data can be easily quadrature modulated and upconverted.

Quadrature Modulation

- **Inputs:**
Analog discrete I and Q values generated by the DAC (Digital to Analog Converter).
- **Outputs:**
Real RF quadrature modulated signal with both magnitude and phase preserved .
- **Process:**
The NCO (Numerically Controlled Oscillator) is used to generate 64 cos and sin values (in the range of 0-pi/2).

```
// Extract quadrant and index
ap_uint<2> quadrant = phase >> 6; // Upper 2 bits for quadrant
ap_uint<6> index = phase & 0x3F; // Lower 6 bits for LUT index

data_t cos_val, sin_val;

// Get base values from LUT
ap_fixed<16,4> lut_val = cos_lut[index];

// Generate cos and sin based on quadrant
switch(quadrant) {
    case 0: // 0 to pi/2
        cos_val = (data_t)lut_val;
        sin_val = (data_t)cos_lut[63-index]; // sin = cos(pi/2 - teta)
        break;
    case 1: // pi/2 to pi
        cos_val = (data_t)(-cos_lut[63-index]);
        sin_val = (data_t)lut_val;
        break;
    case 2: // pi to 3pi/2
        cos_val = (data_t)(-lut_val);
        sin_val = (data_t)(-cos_lut[63-index]);
        break;
    case 3: // 3pi/2 to 2pi
        cos_val = (data_t)cos_lut[63-index];
        sin_val = (data_t)(-lut_val);
        break;
}

cos_lo = cos_val;
sin_lo = sin_val;
phase += phase_inc;
}
```

The quadrant and index are extracted from the inputs such that they may be conserved.

The extracted input is then used in the digital_qm mixer to return the quadrature modulated RF baseband real values with conserved phase and magnitude.

```
data_t digital_qm(data_t I, data_t Q, data_t cos_lo, data_t sin_lo) {
#pragma HLS INLINE
    acc_t mix = (acc_t)I * cos_lo - (acc_t)Q * sin_lo;
    return (data_t)mix;
}
```

Efficiency:

The quadrature modulated real values can easily be upconverted and amplified.

Digital Upconverter

Inputs:

Input to the block is taken from the output of the Quadrature Modulator.

Outputs:

Outputs of the block are upconverted and upsampled filtered RF realbaseband values.

Process :

The inputs are interpolated by a factor of 8 (upsampled). This is achieved by considering a FIR filter. The filter coefficients are then applied to the inputs for interpolation.

Cos and Sin values are generated by a numerically controlled oscillator and are saved in LUTs.

The obtained sin and cos values are then multiplied with the interpolated carrier signal for complete upconversion.

```
// Read one input sample
sample_type i_sample = 0, q_sample = 0;
if (!i_in.empty() && !q_in.empty()) {
    i_in >> i_sample;
    q_in >> q_sample;
} else {
    return;
}

// Perform interpolation and upconversion for each interpolated sample
for (int phase = 0; phase < INTERPOLATION_FACTOR; phase++) {
    #pragma HLS PIPELINE II=1

    // For simple interpolation, only use new input on phase 0, else use 0
    sample_type i_fir_in = (phase == 0) ? i_sample : sample_type(0);
    sample_type q_fir_in = (phase == 0) ? q_sample : sample_type(0);

    sample_type i_interp, q_interp;
    fir_filter(i_shift_reg, i_fir_in, i_interp);
    fir_filter(q_shift_reg, q_fir_in, q_interp);

    // Numerically controlled oscillator
    phase_acc += freq_control_word;
    phase_type lut_addr = phase_acc >> (32 - NCO_LUT_BITS);
    sample_type sin_val = sine_lut[lut_addr];
    sample_type cos_val = cosine_lut[lut_addr];

    // Complex multiplication (upconversion)
    sample_type upconverted = i_interp * cos_val - q_interp * sin_val;

    // Output the upconverted sample
    signal_out << upconverted;
```

Efficiency:

Since sin and cos values are precomputed and stored in LUTs. This module is not computationally intensive.

Power Amplifier

Inputs:

The inputs to the PA are RF realbaseband values from the Upconverter
Phase Distortions are simulated to substitute Q

Outputs:

The output are amplified I and Q magnitude values, Linear Gain and Gain in dB scale.

Process :

RF inputs are extracted from the Quadrature Modulator, and these values, along with constant parameters, are utilized to generate amplitude and phase amplification values. Amplitude compression is subsequently applied to the RF values. Non-linear distortion effects are introduced to create harmonics and intermodulation distortion (IMD). Additionally, AM and PM modulation effects are incorporated. The resulting I values are stored as out_i. The previously generated distortions serve as realistic RF q_out values. Finally, the ultimate outputs are scaled by a factor of 5 and set.

```
// Saleh amplitude model (creates compression)
float A_r = (alpha_a * rf_magnitude) / (1 + beta_a * rf_magnitude *
rf_magnitude);

// Saleh phase model (creates AM-PM distortion)
float P_r = (alpha_p * rf_magnitude * rf_magnitude) / (1 + beta_p *
rf_magnitude * rf_magnitude);

// Apply amplitude compression to the RF signal
float gain_factor = A_r / rf_magnitude;
float compressed_rf = in_i.to_float() * gain_factor;

// Add nonlinear distortion effects (creates harmonics and IMD)
float distorted_rf = compressed_rf;
if (std::abs(compressed_rf) > 0.1f) {
    // Add cubic and quintic nonlinearities
    float norm_signal = compressed_rf / 3.0f; // Normalize to prevent
overflow
    float cubic_term = norm_signal * norm_signal * norm_signal * 0.2f;
    float quintic_term = norm_signal * norm_signal * norm_signal *
norm_signal * norm_signal * 0.05f;

    distorted_rf = compressed_rf + cubic_term + quintic_term;

    // Add AM-PM phase modulation effect
    float phase_mod = P_r * 0.15f * std::sin(P_r * 8.0f);
    distorted_rf *= (1.0f + phase_mod);
}

// Put distorted RF in out_i, create some Q due to phase distortion
out_i = data_t(distorted_rf);

// Phase distortion creates small Q component (realistic for RF PA)
float q_distortion = distorted_rf * P_r * 0.1f * std::sin(P_r * 5.0f);
out_q = data_t(q_distortion);

// Apply the 5.0 scaling
out_i = data_t(out_i.to_float());
out_q = data_t(out_q.to_float());

// Set other outputs
magnitude = data_t(A_r);
gain_lin = data_t(gain_factor);

// FIXED: HLS-compatible dB calculation using pre-computed constant
const float INV_LN10 = 0.434294481903f; // 1/ln(10) to avoid division

if (gain_factor > 0.001f) {
    gain_db = data_t(20.0f * std::log(gain_factor * 5.0f) * INV_LN10);
} else {
    gain_db = data_t(-60.0f); // Very low gain
}
```

Analog to Digital Converter

Inputs:

Down converted and demodulated analog I and Q value arrays.

Outputs:

Discrete digital I and Q value arrays.

Process:

Based on Vref values the maximum and minimum values of the range are computed.

```
const ap_fixed<24,8> V_REF = 1.0;      // Update type
const ap_fixed<24,8> V_MIN = -V_REF;    // Update type
const ap_fixed<24,8> V_MAX = V_REF;     // Update type
const int scale = (1 << (W-1)) - 1;    // 32767 for 16-bit
```

Scale is computed to normalize voltage range to digital range.

All the input samples are iterated through, and compared with the maximum and minimum range values.

After the values are adjusted in the acceptable range ,these values are scaled accordingly.

```
for (int i = 0; i < N; i++) {
#pragma HLS PIPELINE II=1
    // I channel
    ap_fixed<24,8> clamped_I = (I_analog_in[i] > V_MAX) ? V_MAX :
                               ((I_analog_in[i] < V_MIN) ? V_MIN :
                                I_analog_in[i]);
    double scaled_I = clamped_I.to_double() * scale;
    I_digital_out[i] = (ap_int<W>)scaled_I;

    // Q channel
    ap_fixed<24,8> clamped_Q = (Q_analog_in[i] > V_MAX) ? V_MAX :
                               ((Q_analog_in[i] < V_MIN) ? V_MIN :
                                Q_analog_in[i]);
    double scaled_Q = clamped_Q.to_double() * scale;
    Q_digital_out[i] = (ap_fixed<24,8>)scaled_Q;

    // Debug output during simulation
    #ifndef __SYNTHESIS__
    if (i < 10 || i > N-10) {
        printf("Sample %d: I_analog=%f, I_digital=%d | Q_analog=%f, Q_digital=%d\n",
               i, clamped_I.to_double(), (int)I_digital_out[i],
               clamped_Q.to_double(), (int)Q_digital_out[i]);
    }
    #endif
}
```


Demodulation and Downconversion

Inputs:

Amplified (RF) values from the Power Amplifier.

Outputs:

Demodulated and Down converted I and Q complex baseband signals.

Process:

I and Q static buffers are initialized to zero. A Numerically Controlled Oscillator generates sine and cosine values. RF samples are multiplied by cosine and negative sine to produce I and Q values. These mixed I and Q values are then added to the buffer. For demodulation, the buffer values are multiplied by the low-pass filter coefficients. The resulting output values are scaled down by a factor of 8 and stored as I_out and Q_out.

```
// FIXED: Initialize buffers only once
if (!buffers_initialized) {
    for (int i = 0; i < FIR_TAPS; i++) {
        #pragma HLS UNROLL
        i_buffer[i] = 0;
        q_buffer[i] = 0;
    }
    buffers_initialized = true;
}

// FIXED: Static NCO phase accumulator (persists between calls)
static ap_uint<32> phase_acc = 0;

// Track output sample count
int out_sample_idx = 0;
int decim_count = 0;

// Calculate expected output samples
const int expected_outputs = num_samples / DECIM_FACTOR;

// FIXED: Reasonable scaling - your gain is way too high!
const filter_accum_t REASONABLE_GAIN = filter_accum_t(gain.to_float() /
1000.0f); // Scale down by 1000x

// Main processing loop
for (int n = 0; n < num_samples; n++) {
    #pragma HLS PIPELINE II=1
    #pragma HLS LOOP_TRIPCOUNT min=8192 max=65536 avg=32768

    // Bounds check
    if (n >= num_samples) break;

    // Get input sample
    rf_sample_t rf_sample = rf_in[n];
```