Week 1

Week-1: Summary

Introduction to HLS Coding: Definition, need for HLS, comparison with Verilog coding, advantages, limitations, and applications (e.g., IIR/FIR filters, matrix multipliers).

Working with Vivado HLS: Steps for using Vivado HLS, illustrated with an "Addition of 2 Numbers" example. This section also addresses challenges encountered with user-defined inputs and file I/O, including handling input/output files and generating Verilog code.

Working with Datatypes in HLS: Observations and code examples (C and Verilog) for `unsigned int()`, `signed int()`, `char()`, and `float()`. This section highlights how HLS handles different data types, including the creation of FSMs for `float` operations and the use of `ap_fixed<W,I>` in C++ to manage fixed-point precision.

This document provides a summary of the work completed during Week 1 of a summer internship, spanning from June 2nd to June 6th, 2025. The primary focus of this period was on High-Level Synthesis (HLS) and its diverse applications.

The summary encompasses the following key areas:

Introduction to HLS Coding: This section defines HLS, articulates the necessity for its implementation, and offers a comparative analysis with Verilog coding. It further explores the advantages, limitations, and practical applications of HLS, including examples such as IIR/FIR filters and matrix multipliers.

Working with Vivado HLS: This part outlines the procedural steps for utilizing Vivado HLS, exemplified through an "Addition of 2 Numbers" scenario. It also addresses challenges encountered concerning user-defined inputs and file I/O, detailing methods for handling input/output files and the generation of Verilog code.

Working with Datatypes in HLS: This section presents observations and provides corresponding C and Verilog code examples for `unsigned int()`, `signed int()`, `char()`, and `float()`. It elucidates how HLS processes various data types, highlighting the creation of Finite State Machines (FSMs) for float operations and the application of `ap_fixed<W,I>` in C++ for managing fixed-point precision.

02/06/2025

HLS CODING

1. Introduction: HLS coding is the writing of code in a high level language such as C, C++ or SystemC and based on the code written, the system will automatically determine the microarchitecture and convert it into a register-transfer-level (RTL) design. Essentially, in HLS coding, what we do is describe what the hardware must do and the tool decides on its implementation.

2. Need for HLS: It is mainly used to bridge the gap between software and hardware design by allowing people to write code in high level languages. It allows for users to explore the aspects of power, performance and area while designing.

3. HLS v/s Verilog coding:
   i. HLS uses high level languages whereas Verilog is a hardware description language that directly maps the code to the hardware.
   ii. HLS works on a higher level of abstraction allowing the user to focus on the logic and its implementation. Verilog works on a lower level of abstraction describing the hardware structure and its logic in terms of components such as registers and pipelines.
   iii. HLS is used to build hardware accelerators or custom processors where the algorithm is given a higher priority. In Verilog coding, we make hardware designs such as simple logic blocks to complex systems.
   iv. HLS has fewer lines of code making it easier and faster to compile and run whereas Verilog has more lines of code taking more time to compile.

4. Advantages:
   1. It is easier to pivot around and to design and verify logic.
   2. Leads to faster development time as the system handles the translation of code.
   3. Different architectures can be explored by changing the format of the code written.

5. <u>Limitations</u>:

Performance can be affected based on the complexity of the algorithm.
Can sometimes result in less efficient hardware implementation compared to HDL coding.

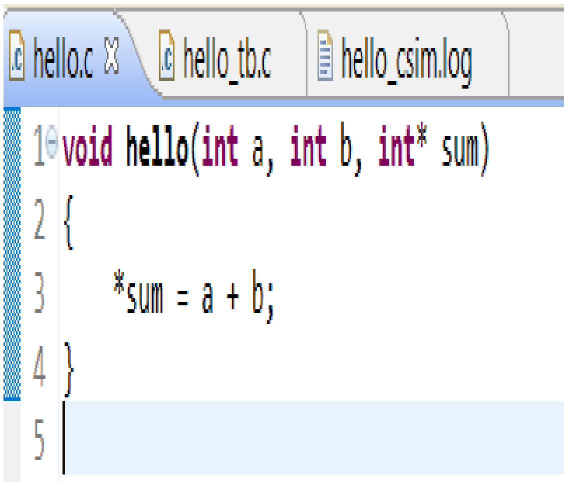
6. <u>Applications</u>:
    1. HLS enables pipelining and parallelism which is Important in the designing of IIR/FIR filters.
    2. HLS coded matrix multipliers for real-time physics simulations.
    3. Code reuse helps in reducing redundancy in code

03/06/2025

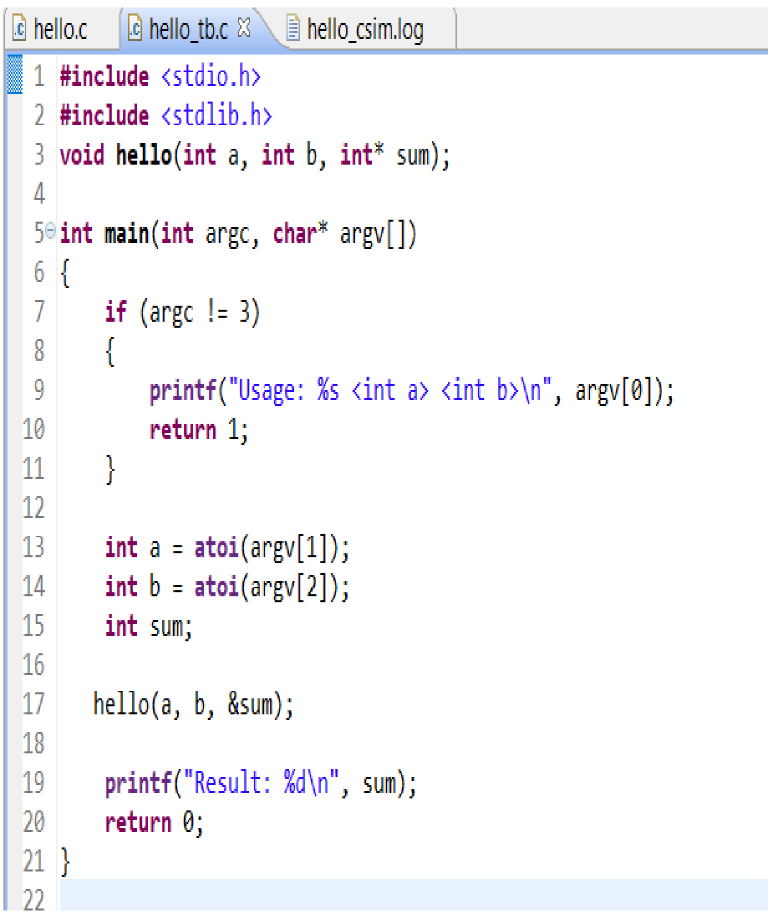
<u>Working with Vivado HLS</u>:

1. Install Xilinx and then open Vivado HLS.
2. Create New Project, select the required parts and name the main and test bench with the extension of *.c*, namely *hello.c* and *hello_tb.c* .
3. Write the respective codes in their folders and save them.
4. Click on **C Simulation** and confirm if the test bench runs as expected without errors.
5. Check the output if it is being displayed correctly.
6. Now, click on **Run C Synthesis** and after its done, go to **Solution Reports** where you will be able to see the Verilog and HDL equivalent of the written code.

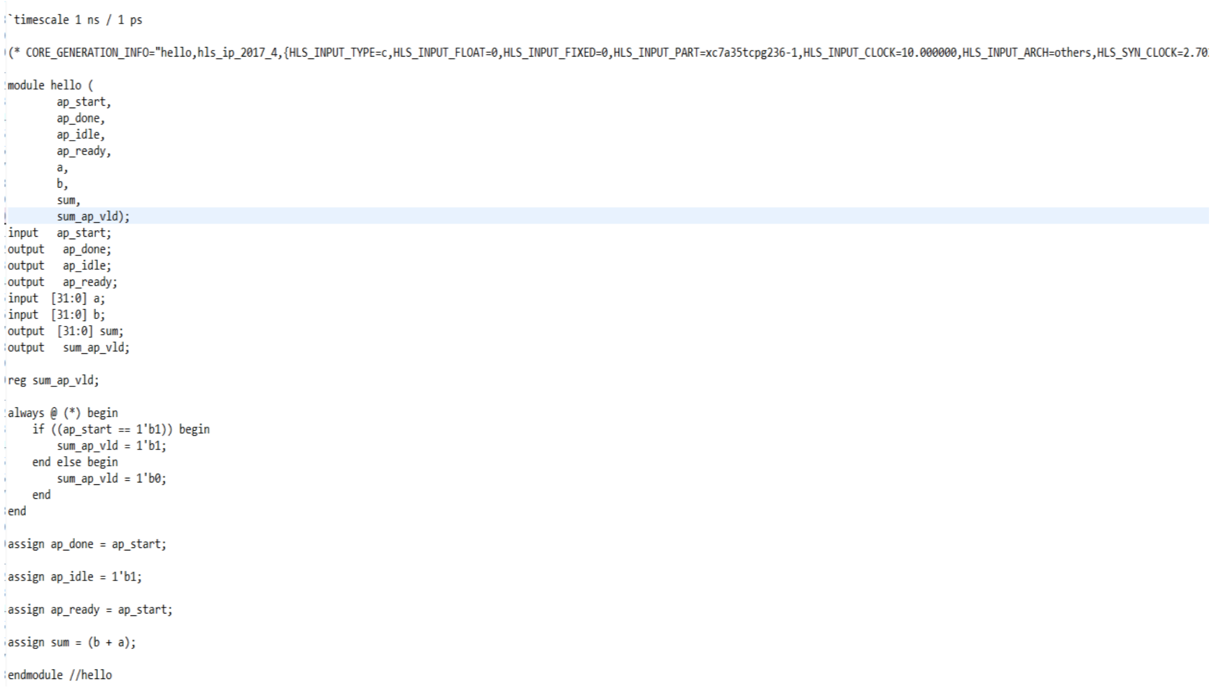
<u>Addition of 2 Numbers</u>:
First, addition of 2 numbers was performed by hard coding and then by user defined inputs.
By user- defined inputs,



```c
void hello(int a, int b, int* sum)
{
    *sum = a + b;
}

```

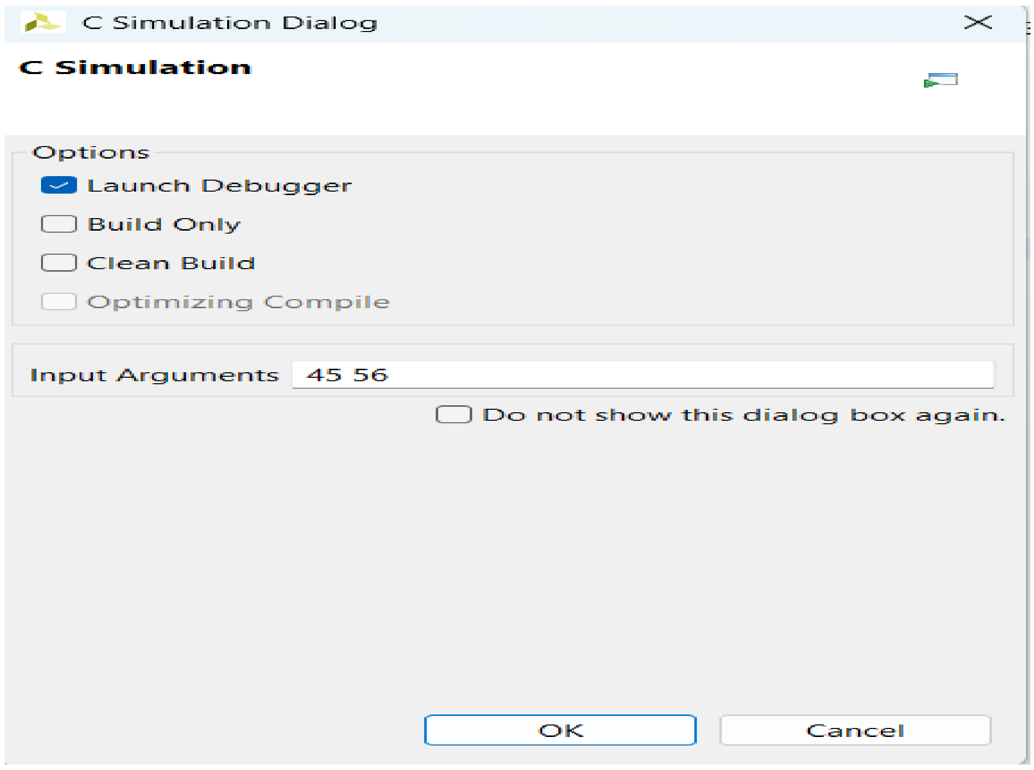


```c
#include <stdio.h>
#include <stdlib.h>
void hello(int a, int b, int* sum);

int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        printf("Usage: %s <int a> <int b>\n", argv[0]);
        return 1;
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int sum;

   hello(a, b, &sum);

    printf("Result: %d\n", sum);
    return 0;
}

```



```
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="hello,hls_ip_2017_4,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a35tcpg236-1,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=2.70

module hello (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a,
        b,
        sum,
        sum_ap_vld);
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [31:0] a;
input  [31:0] b;
output  [31:0] sum;
output   sum_ap_vld;

reg sum_ap_vld;

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

assign ap_done = ap_start;

assign ap_idle = 1'b1;

assign ap_ready = ap_start;

assign sum = (b + a);

endmodule //hello
```

We saw that we cannot use the regular scanf() or fscanf() or fopen() and fclose() commands to take in values during run-time of the program and also, HLS takes the given arguments as string type which had to be converted to integer type. So we gave the inputs before compile time.



- [✓] Launch Debugger
- [ ] Build Only
- [ ] Clean Build
- [ ] Optimizing Compile

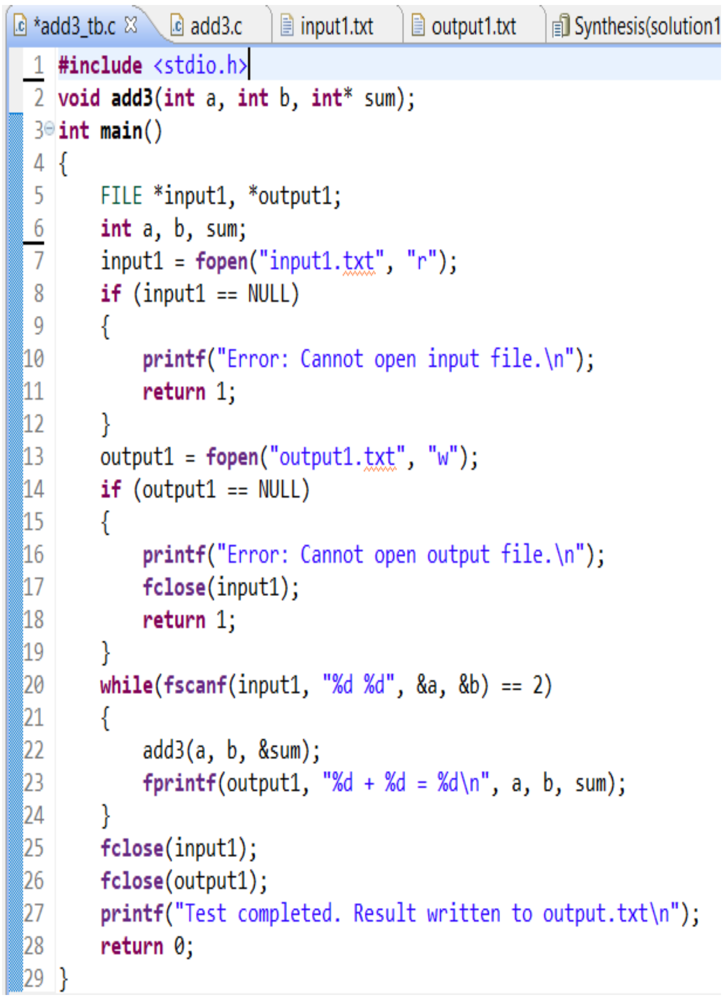
Input Arguments 45 56

- [ ] Do not show this dialog box again.

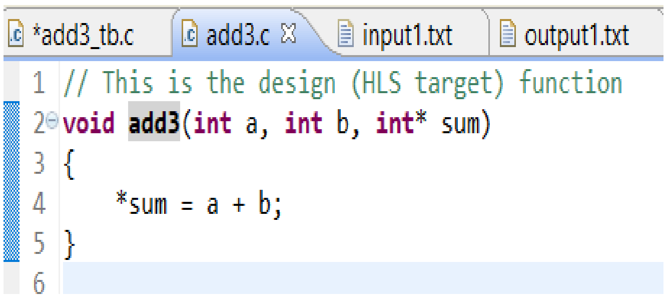
04/06/2025

Addition using File I/O:

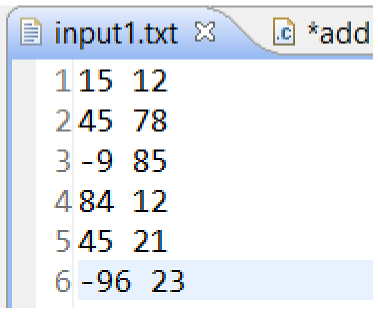
Today, we performed the addition of 2 numbers using File Inputs i.e. giving the inputs as values stored in a file "*input.txt*", extracting information from there and writing the outputs to a new file "*output.txt*".



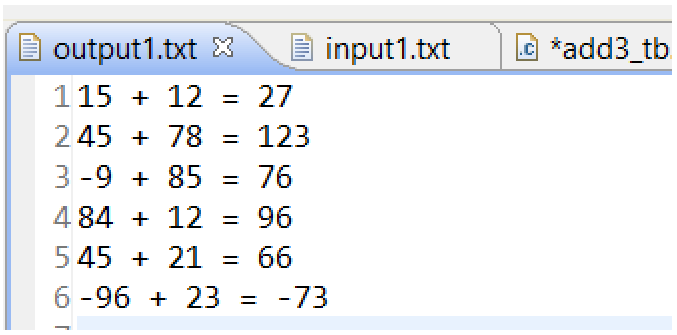
```c
#include <stdio.h>
void add3(int a, int b, int* sum);
int main()
{
    FILE *input1, *output1;
    int a, b, sum;
    input1 = fopen("input1.txt", "r");
    if (input1 == NULL)
    {
        printf("Error: Cannot open input file.\n");
        return 1;
    }
    output1 = fopen("output1.txt", "w");
    if (output1 == NULL)
    {
        printf("Error: Cannot open output file.\n");
        fclose(input1);
        return 1;
    }
    while(fscanf(input1, "%d %d", &a, &b) == 2)
    {
        add3(a, b, &sum);
        fprintf(output1, "%d + %d = %d\n", a, b, sum);
    }
    fclose(input1);
    fclose(output1);
    printf("Test completed. Result written to output.txt\n");
    return 0;
}
```



```c
// This is the design (HLS target) function
void add3(int a, int b, int* sum)
{
    *sum = a + b;
}
```

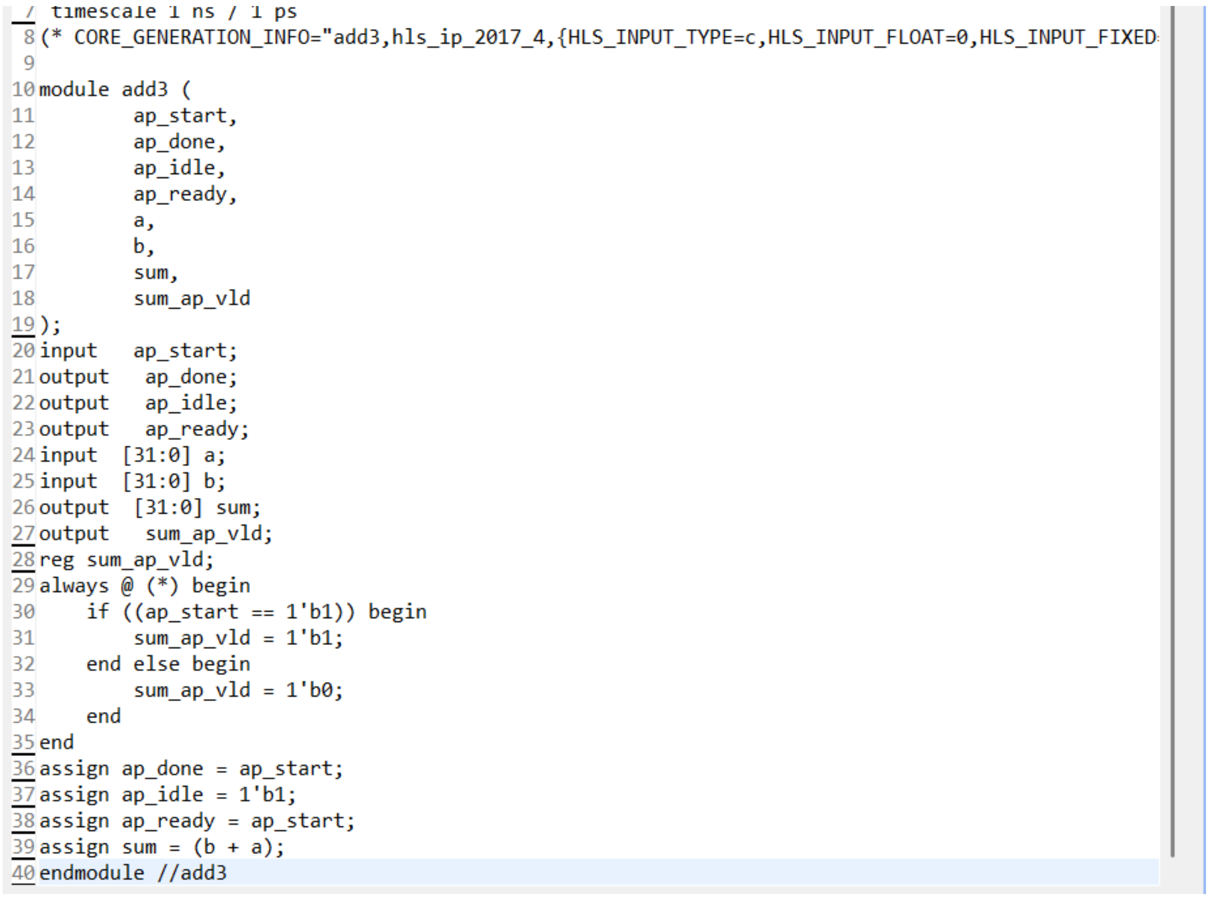


```
15 12
45 78
-9 85
84 12
45 21
-96 23
```



```
15 + 12 = 27
45 + 78 = 123
-9 + 85 = 76
84 + 12 = 96
45 + 21 = 66
-96 + 23 = -73
```

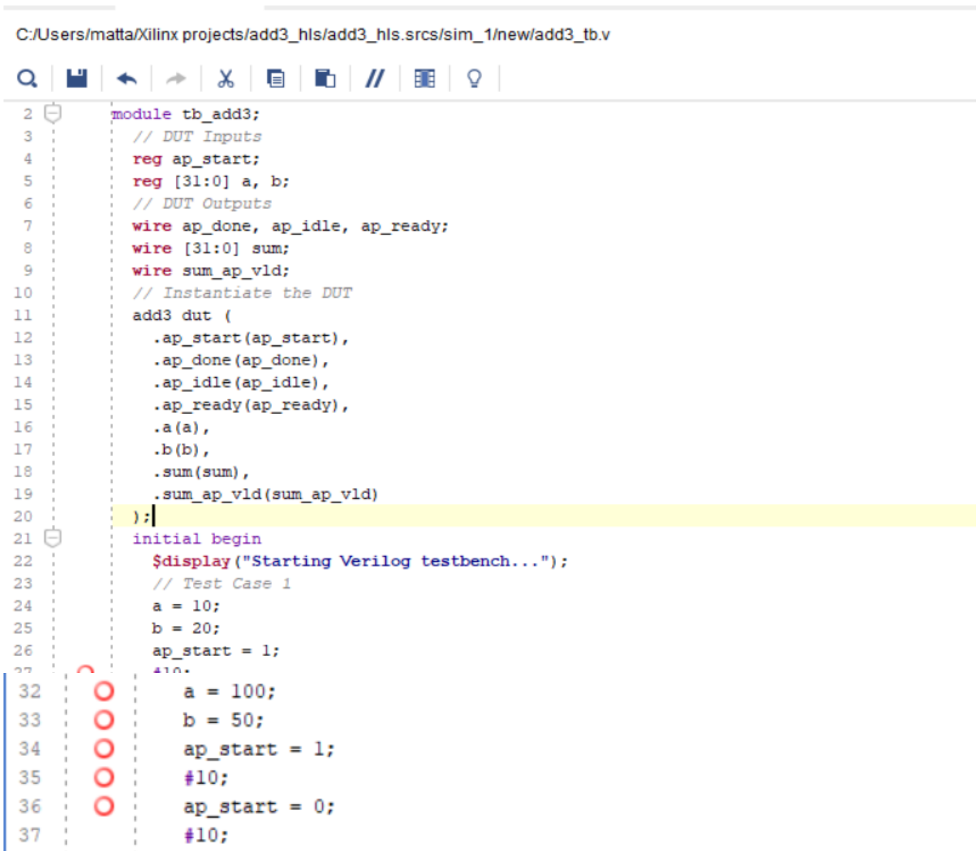
The generated Verilog code for the same code:



```
`timescale 1 ns / 1 ps
(* CORE_GENERATION_INFO="add3,hls_ip_2017_4,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED

module add3 (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a,
        b,
        sum,
        sum_ap_vld
);
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [31:0] a;
input  [31:0] b;
output  [31:0] sum;
output   sum_ap_vld;
reg sum_ap_vld;
always @ (*) begin
    if ((ap_start == 1'b1)) begin
        sum_ap_vld = 1'b1;
    end else begin
        sum_ap_vld = 1'b0;
    end
end
assign ap_done = ap_start;
assign ap_idle = 1'b1;
assign ap_ready = ap_start;
assign sum = (b + a);
endmodule //add3
```

The issue we saw today was the output file being in a different location than expected and leading to thinking that the code written was wrong and after a little tinkering and exploring the interface, we found the output file.

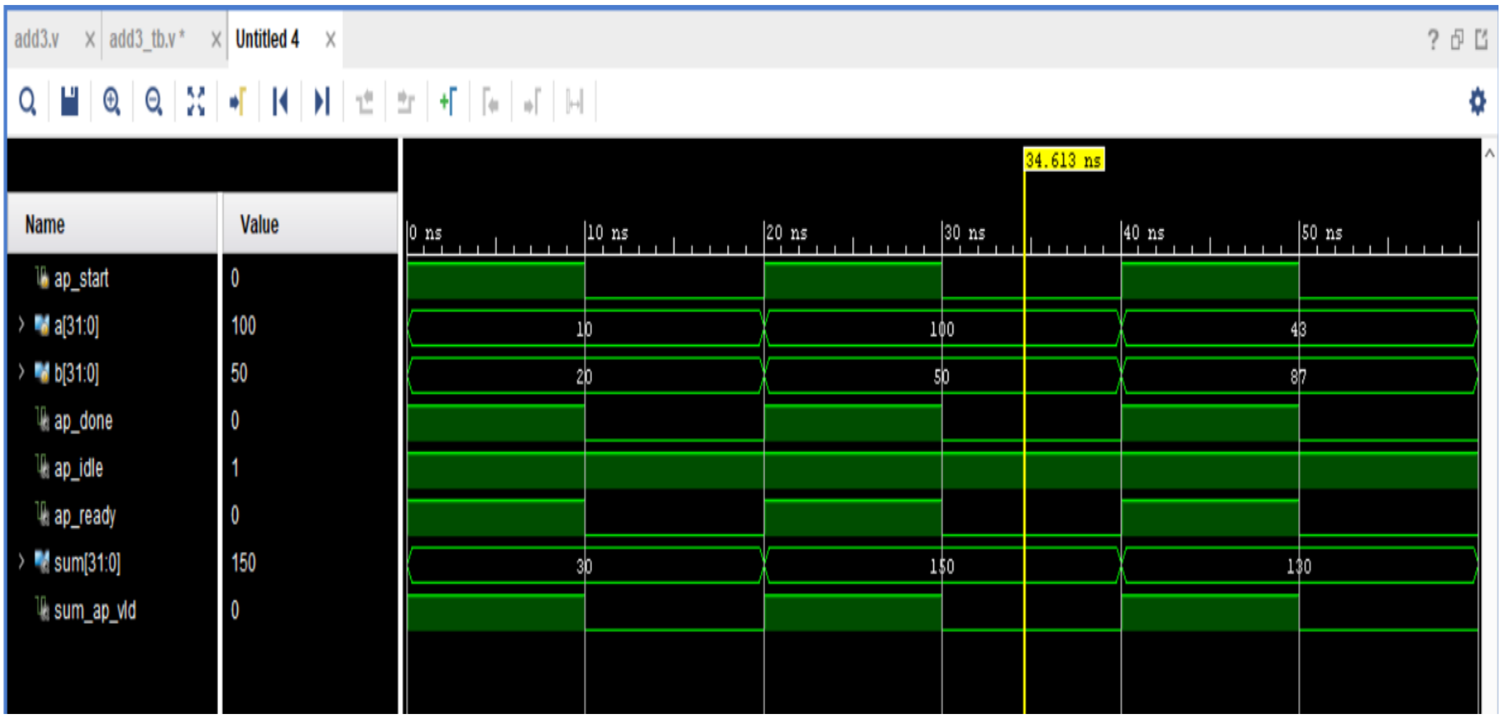
The next step was to take the generated Verilog code and implement it in Xilinx Vivado to check the correctness of the code by writing a testbench with hardcoded values for variables.



C:/Users/matta/Xilinx projects/add3_hls/add3_hls.srcs/sim_1/new/add3_tb.v

```
module tb_add3;
  // DUT Inputs
  reg ap_start;
  reg [31:0] a, b;
  // DUT Outputs
  wire ap_done, ap_idle, ap_ready;
  wire [31:0] sum;
  wire sum_ap_vld;
  // Instantiate the DUT
  add3 dut (
    .ap_start(ap_start),
    .ap_done(ap_done),
    .ap_idle(ap_idle),
    .ap_ready(ap_ready),
    .a(a),
    .b(b),
    .sum(sum),
    .sum_ap_vld(sum_ap_vld)
  );
  initial begin
    $display("Starting Verilog testbench...");
    // Test Case 1
    a = 10;
    b = 20;
    ap_start = 1;
    #10;
```

```
    a = 100;
    b = 50;
    ap_start = 1;
    #10;
    ap_start = 0;
    #10;
```

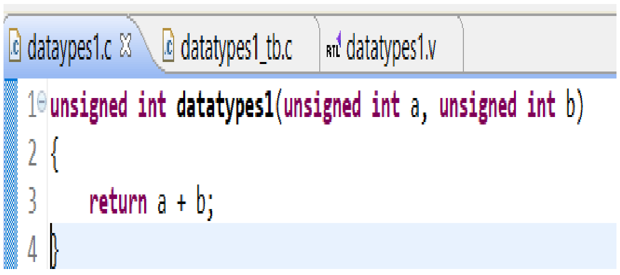


Name	Value
ap_start	0
a[31:0]	100
b[31:0]	50
ap_done	0
ap_idle	1
ap_ready	0
sum[31:0]	150
sum_ap_vld	0

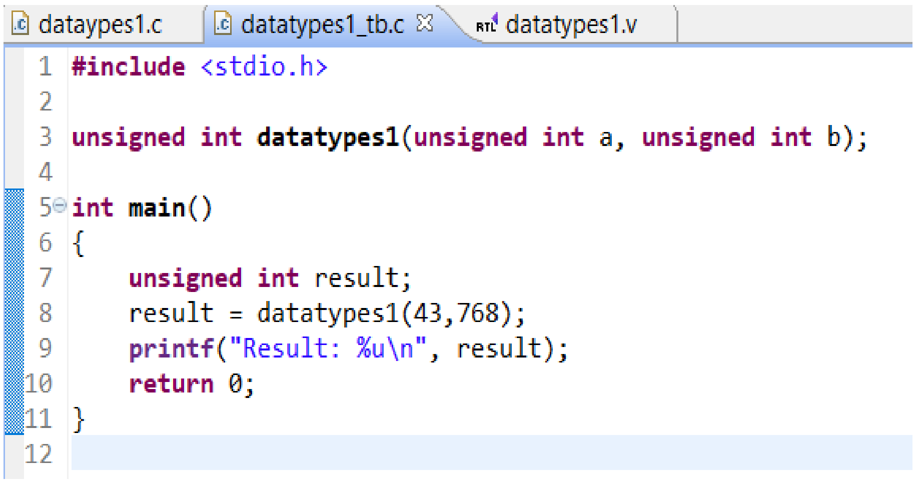
05/06/2025 - 06/06/2025

Working with Datatypes in HLS

1. Unsigned int():
   C code and testbench:

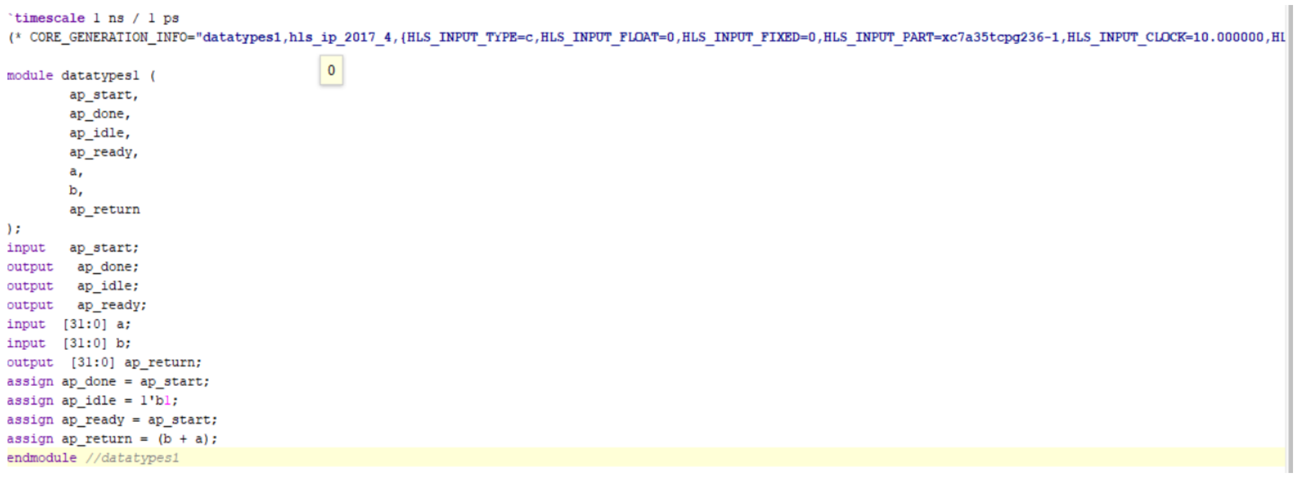


```c
unsigned int datatypes1(unsigned int a, unsigned int b)
{
    return a + b;
}
```

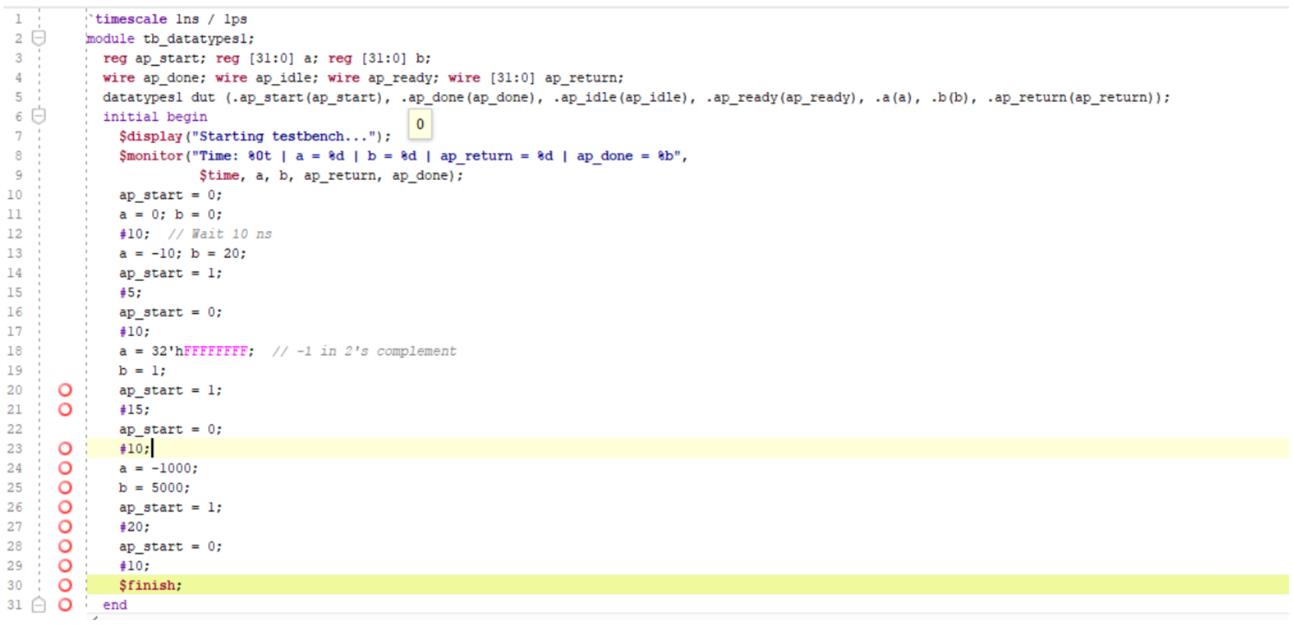


```c
#include <stdio.h>

unsigned int datatypes1(unsigned int a, unsigned int b);

int main()
{
    unsigned int result;
    result = datatypes1(43,768);
    printf("Result: %u\n", result);
    return 0;
}
```

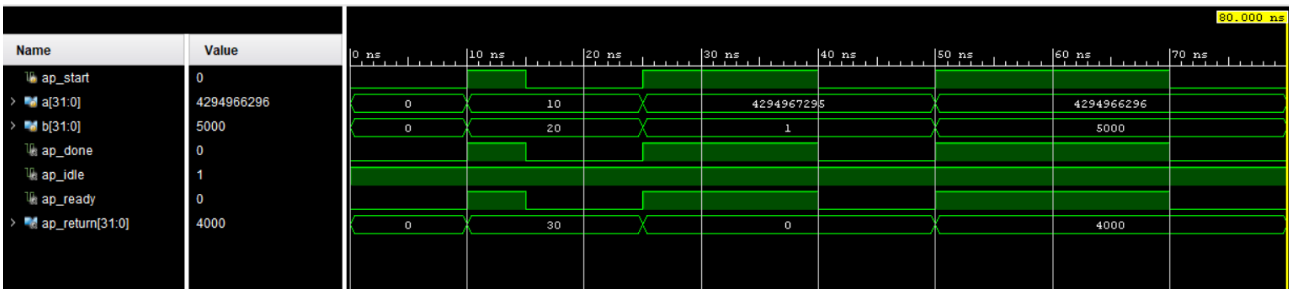
   Verilog code and testbench:



```verilog
`timescale 1 ns / 1 ps
(* CORE_GENERATION_INFO="datatypes1,hls_ip_2017_4,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a35tcpg236-1,HLS_INPUT_CLOCK=10.000000,HL

module datatypes1 (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a,
        b,
        ap_return
);
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [31:0] a;
input  [31:0] b;
output  [31:0] ap_return;
assign ap_done = ap_start;
assign ap_idle = 1'b1;
assign ap_ready = ap_start;
assign ap_return = (b + a);
endmodule //datatypes1
```



```verilog
`timescale 1ns / 1ps
module tb_datatypes1;
  reg ap_start; reg [31:0] a; reg [31:0] b;
  wire ap_done; wire ap_idle; wire ap_ready; wire [31:0] ap_return;
  datatypes1 dut (.ap_start(ap_start), .ap_done(ap_done), .ap_idle(ap_idle), .ap_ready(ap_ready), .a(a), .b(b), .ap_return(ap_return));
  initial begin
    $display("Starting testbench...");
    $monitor("Time: %0t | a = %d | b = %d | ap_return = %d | ap_done = %b",
             $time, a, b, ap_return, ap_done);
    ap_start = 0;
    a = 0; b = 0;
    #10;  // Wait 10 ns
    a = -10; b = 20;
    ap_start = 1;
    #5;
    ap_start = 0;
    #10;
    a = 32'hFFFFFFFF;  // -1 in 2's complement
    b = 1;
    ap_start = 1;
    #15;
    ap_start = 0;
    #10;
    a = -1000;
    b = 5000;
    ap_start = 1;
    #20;
    ap_start = 0;
    #10;
    $finish;
  end
```



Name	Value
ap_start	0
a[31:0]	4294966296
b[31:0]	5000
ap_done	0
ap_idle	1
ap_ready	0
ap_return[31:0]	4000

Observations:

I. Test Case 1: a=10, b=20

In this case, since both are positive, we see that the result is displayed properly i.e. 30

II. Test Case 2: a = 32'hFFFFFFFF, b = 1

In this case, since a is -1, it displays 4294967295 (2^{32}-1) and b as 1 itself and display the sum as 0 (-1+1=0).

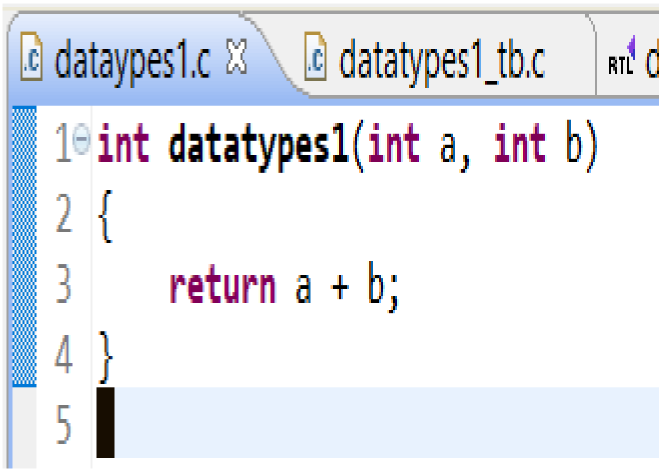
III. Test Case 3: a= -1000, b=5000

In this case, since a is negative, it is shown as 2^{32}-1 in terms of unsigned keeping b as

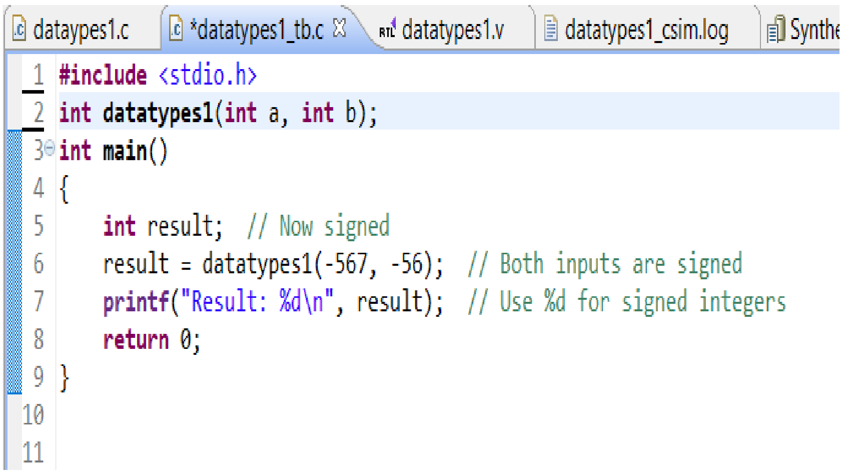
5000 itself giving the result as 4000.

2. Signed int():

C code and testbench:

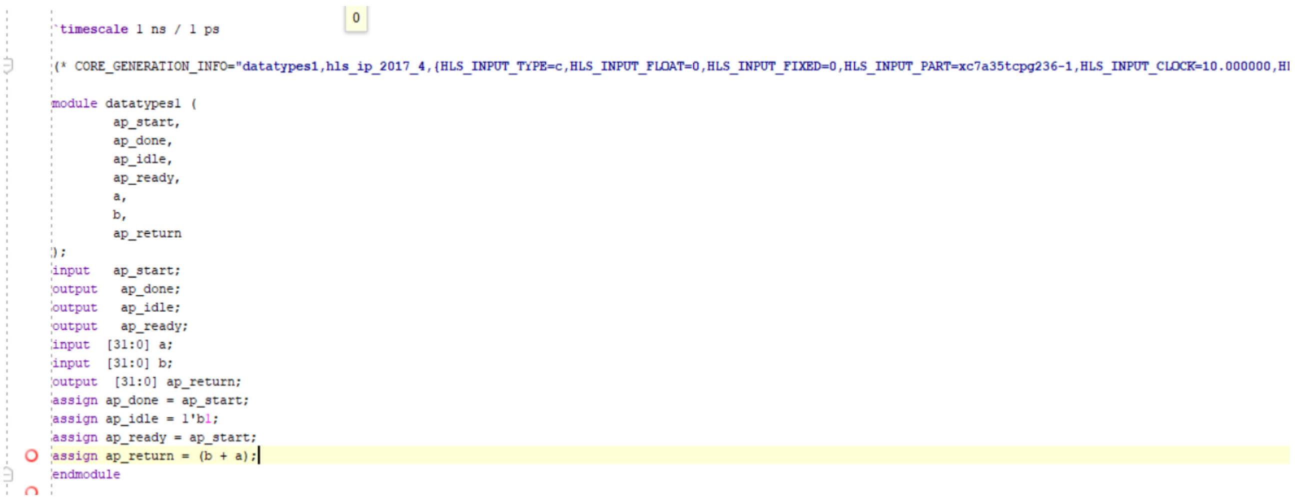


```c
int datatypes1(int a, int b)
{
    return a + b;
}
```

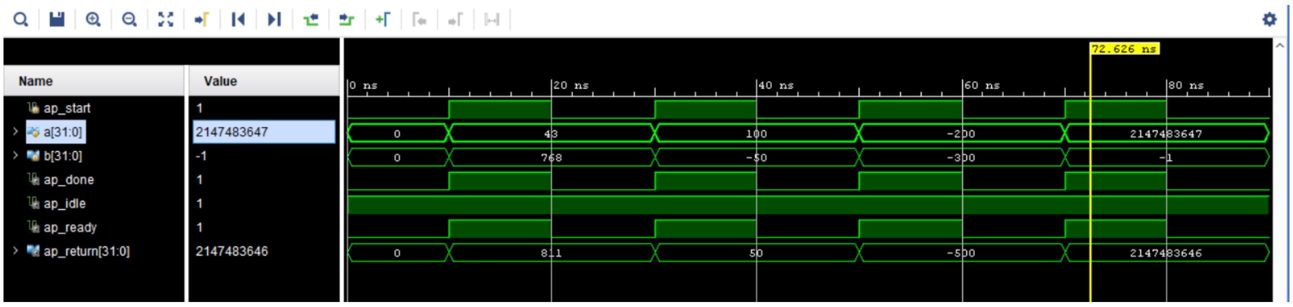


```c
#include <stdio.h>
int datatypes1(int a, int b);
int main()
{
    int result;  // Now signed
    result = datatypes1(-567, -56);  // Both inputs are signed
    printf("Result: %d\n", result);  // Use %d for signed integers
    return 0;
}
```

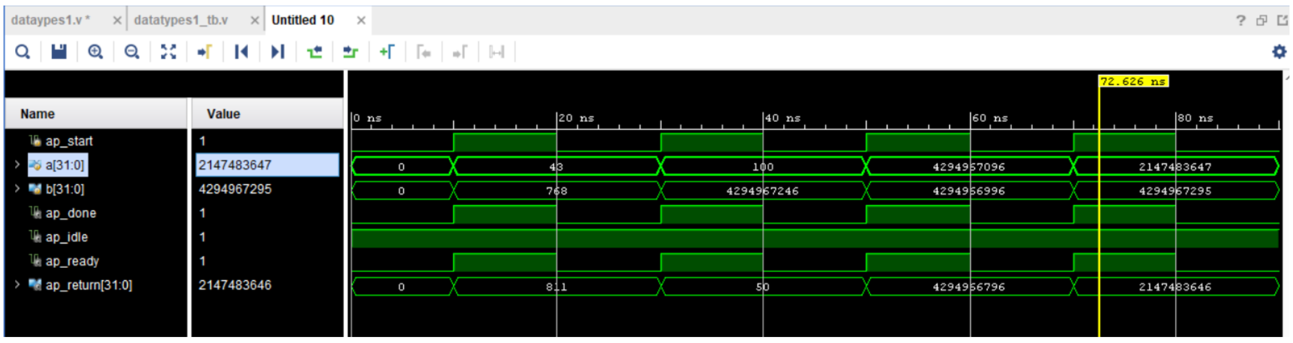
Verilog code and testbench:



```
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="datatypes1,hls_ip_2017_4,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a35tcpg236-1,HLS_INPUT_CLOCK=10.000000,HI

module datatypes1 (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a,
        b,
        ap_return
);
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [31:0] a;
input  [31:0] b;
output  [31:0] ap_return;
assign ap_done = ap_start;
assign ap_idle = 1'b1;
assign ap_ready = ap_start;
assign ap_return = (b + a);
endmodule
```



Name	Value					
ap_start	1					
a[31:0]	2147483647	0	43	100	-200	2147483647
b[31:0]	-1	0	768	-50	-300	-1
ap_done	1					
ap_idle	1					
ap_ready	1					
ap_return[31:0]	2147483646	0	811	50	-500	2147483646



Name	Value					
ap_start	1					
a[31:0]	2147483647	0	43	100	4294967096	2147483647
b[31:0]	4294967295	0	768	4294967246	4294966996	4294967295
ap_done	1					
ap_idle	1					
ap_ready	1					
ap_return[31:0]	2147483646	0	811	50	4294966796	2147483646

Observations:

I. Test case 1: a=0, b=0
   Result will be displayed as 0.

II. Test case 2: a=43, b=768
   Result will be displayed as 811.

III. Test Case 3: a= 100, b= -50

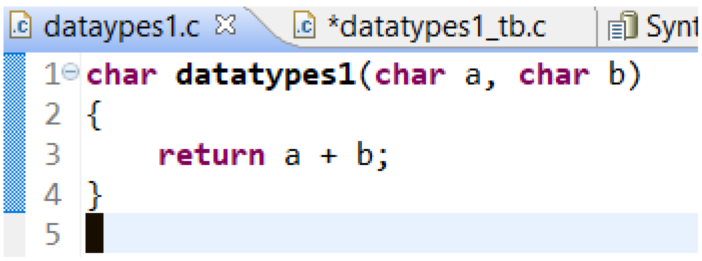
In this case, -50 will be written in its 2's complement form and then in unsigned decimal form is 4294967246, therefore addition is performed as 100+4294967246=50.

IV. Test Case 4: a= -200, b= -300
In this case both numbers are converted to unsigned decimal form and then added giving the result as 4294966796 which is -500.
(11111111111111111111111000001100 is the binary equivalent)

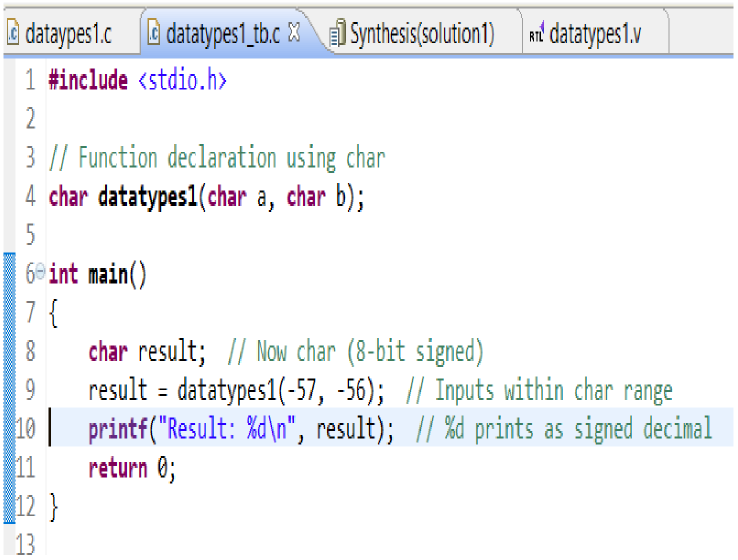
V. Test Case 5: a= 32'h7FFFFFFF, b= -1
This particular case represents a as the highest value stored in a possible for int i.e. 2^{32}+1 and b as -1, so simple addition of these 2 will result in 2147483646.

3. char():

C code and test bench:

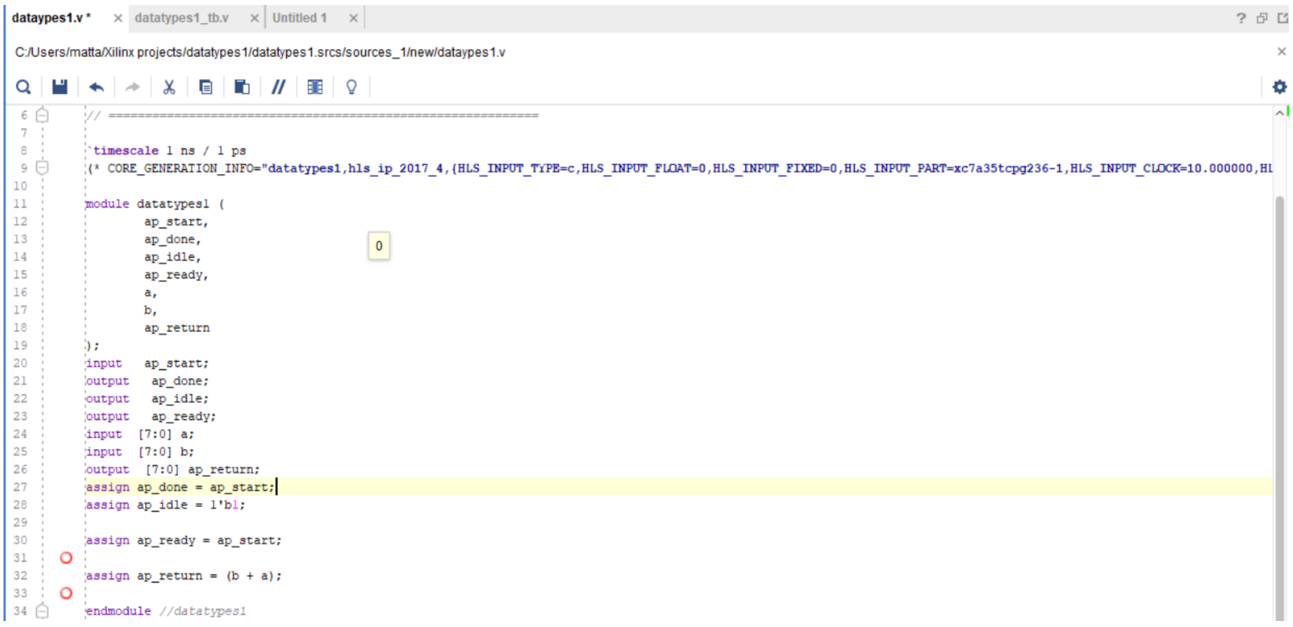


```
char datatypes1(char a, char b)
{
    return a + b;
}

```



```
#include <stdio.h>

// Function declaration using char
char datatypes1(char a, char b);

int main()
{
    char result;  // Now char (8-bit signed)
    result = datatypes1(-57, -56);  // Inputs within char range
    printf("Result: %d\n", result);  // %d prints as signed decimal
    return 0;
}

```

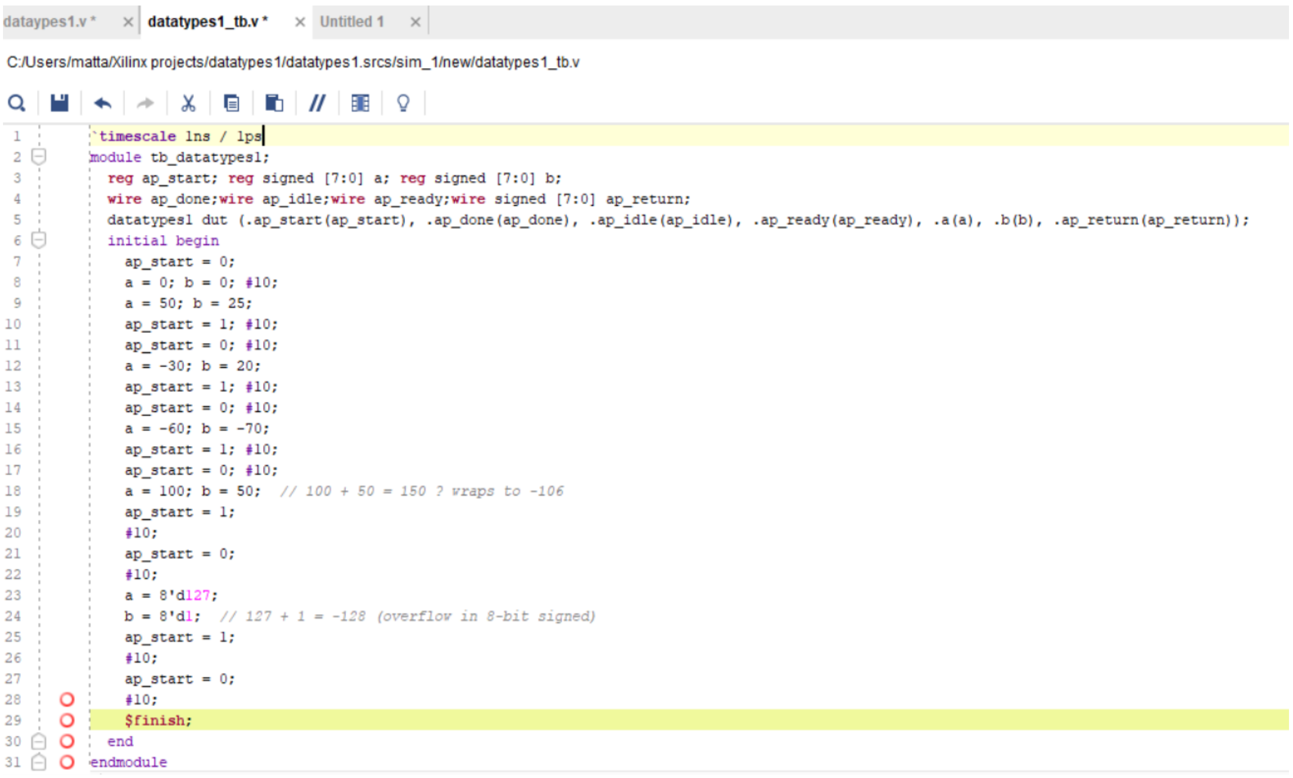
Verilog Code and Testbench:



datatypes1.v * | datatypes1_tb.v | Untitled 1

C:/Users/matta/Xilinx projects/datatypes1/datatypes1.srcs/sources_1/new/datatypes1.v

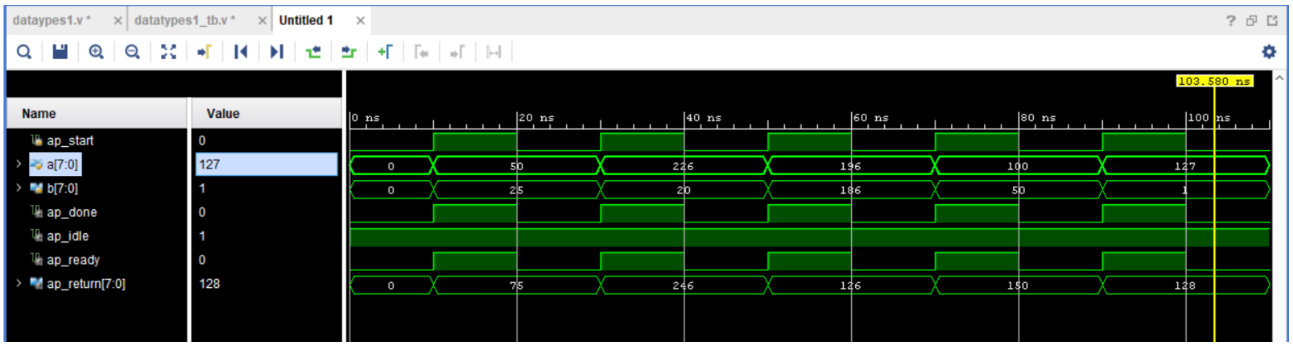
```verilog
// ============================================================

`timescale 1 ns / 1 ps
(* CORE_GENERATION_INFO="datatypes1,hls_ip_2017_4,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a35tcpg236-1,HLS_INPUT_CLOCK=10.000000,HL

module datatypes1 (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a,
        b,
        ap_return
);
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [7:0] a;
input  [7:0] b;
output  [7:0] ap_return;
assign ap_done = ap_start;
assign ap_idle = 1'b1;

assign ap_ready = ap_start;

assign ap_return = (b + a);

endmodule //datatypes1
```



datatypes1.v * | datatypes1_tb.v * | Untitled 1

C:/Users/matta/Xilinx projects/datatypes1/datatypes1.srcs/sim_1/new/datatypes1_tb.v

```verilog
`timescale 1ns / 1ps
module tb_datatypes1;
  reg ap_start; reg signed [7:0] a; reg signed [7:0] b;
  wire ap_done;wire ap_idle;wire ap_ready;wire signed [7:0] ap_return;
  datatypes1 dut (.ap_start(ap_start), .ap_done(ap_done), .ap_idle(ap_idle), .ap_ready(ap_ready), .a(a), .b(b), .ap_return(ap_return));
  initial begin
    ap_start = 0;
    a = 0; b = 0; #10;
    a = 50; b = 25;
    ap_start = 1; #10;
    ap_start = 0; #10;
    a = -30; b = 20;
    ap_start = 1; #10;
    ap_start = 0; #10;
    a = -60; b = -70;
    ap_start = 1; #10;
    ap_start = 0; #10;
    a = 100; b = 50;  // 100 + 50 = 150 ? wraps to -106
    ap_start = 1;
    #10;
    ap_start = 0;
    #10;
    a = 8'd127;
    b = 8'd1;  // 127 + 1 = -128 (overflow in 8-bit signed)
    ap_start = 1;
    #10;
    ap_start = 0;
    #10;
    $finish;
  end
endmodule
```



datatypes1.v * | datatypes1_tb.v * | Untitled 1

```
Name            Value
ap_start        0
a[7:0]          127
b[7:0]          1
ap_done         0
ap_idle         1
ap_ready        0
ap_return[7:0]  128
```

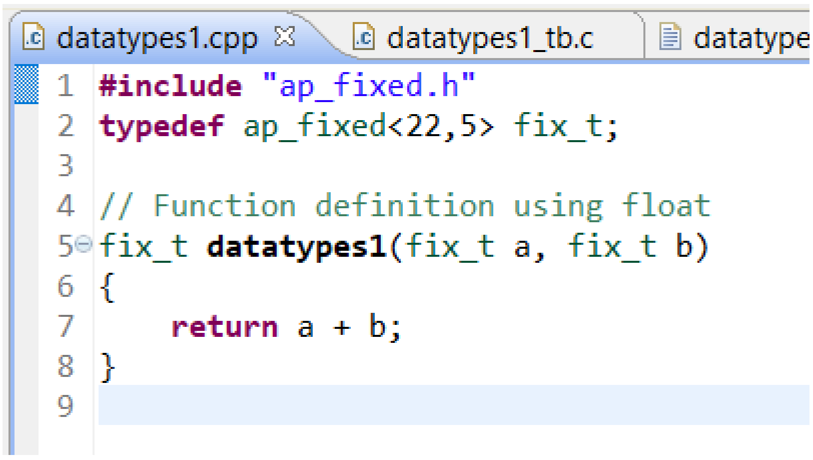
Observations:

a. Test Case 1: a=0, b=0
   Result is printed as 0.
b. Test case 2: a=50, b=25
   Result is printed as 75
c. Test Case 3: a=-30, b= 20
   The 2's complement is taken and the result is displayed as 246.
d. Test Case 4: a= -60, b=-70
   The 2's complement is again taken and the result displayed correspondingly as 126.
e. Test Case 5: a=100, b=50
   Addition is performed normally as 100+50=150 but since 150 crosses the upper limit of char (2^8-1), it wraps around to 106.
f. Test Case 6: a=127, b=1
   Addition is performed normally as 127+1=128 and displays accordingly.

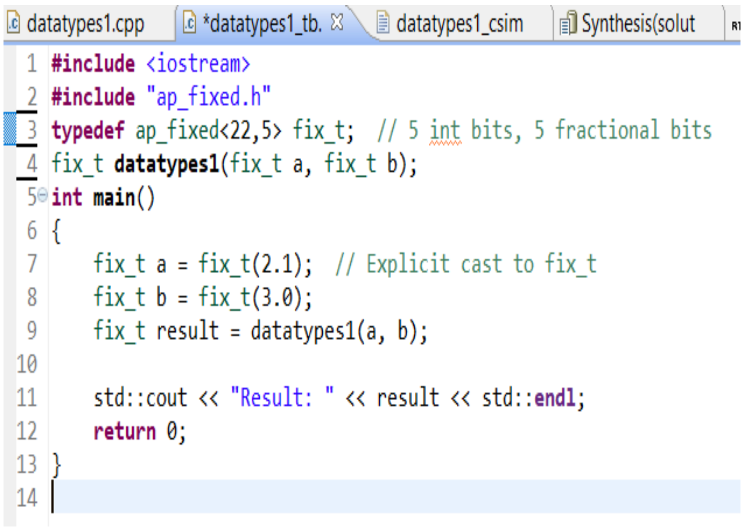
Point to note: When not specified by the compiler or user explicitly, the verilog code generated is unsigned char by default.

4. float():

C++ code and testbench:

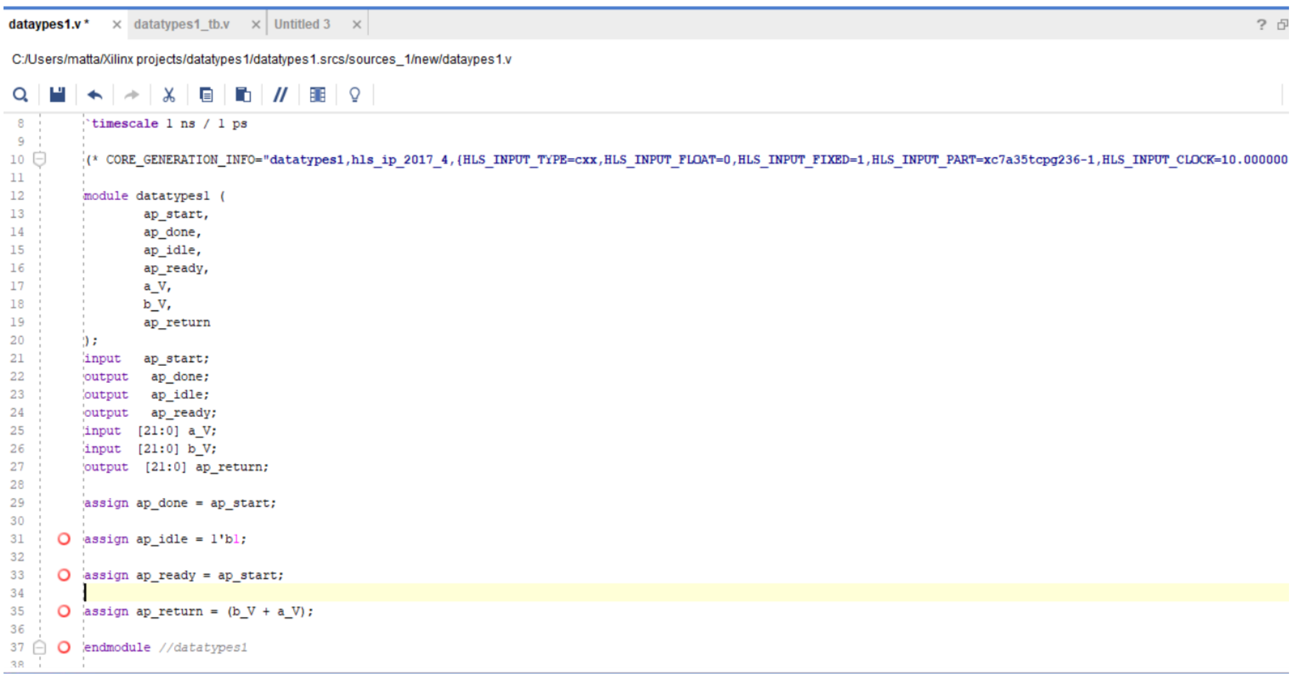


```
#include "ap_fixed.h"
typedef ap_fixed<22,5> fix_t;

// Function definition using float
fix_t datatypes1(fix_t a, fix_t b)
{
    return a + b;
}

```



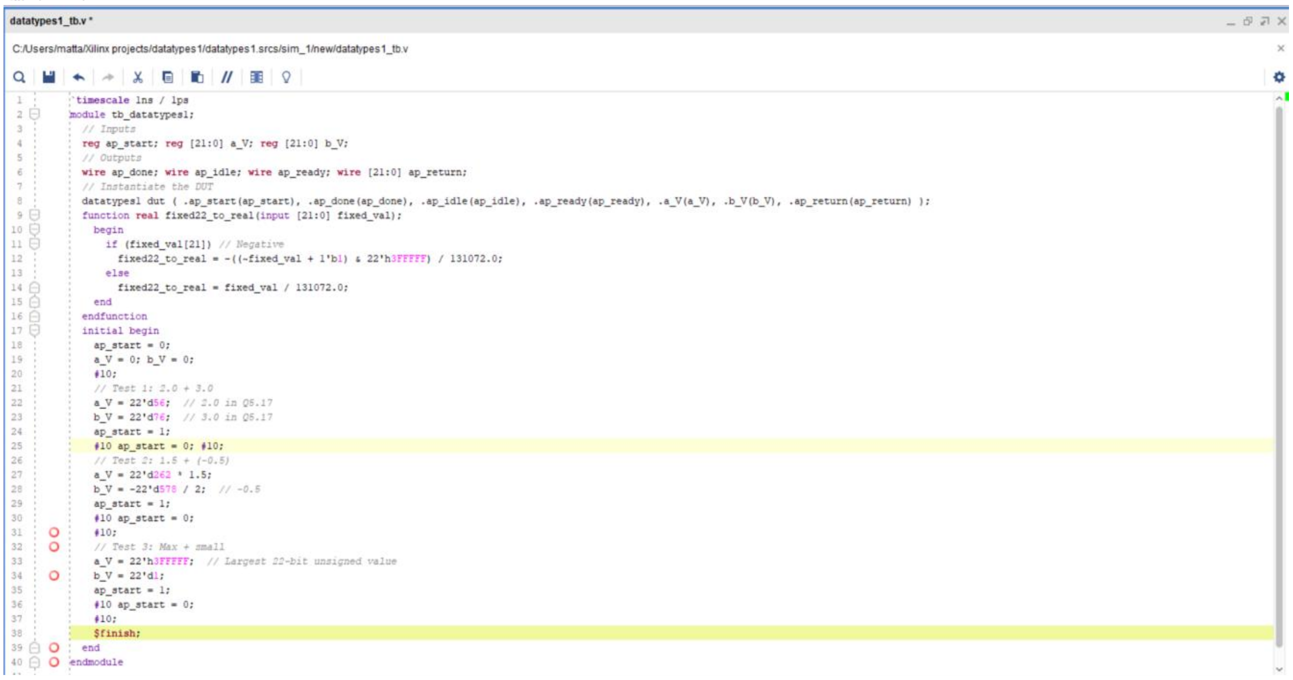
```
#include <iostream>
#include "ap_fixed.h"
typedef ap_fixed<22,5> fix_t;  // 5 int bits, 5 fractional bits
fix_t datatypes1(fix_t a, fix_t b);
int main()
{
    fix_t a = fix_t(2.1);  // Explicit cast to fix_t
    fix_t b = fix_t(3.0);
    fix_t result = datatypes1(a, b);

    std::cout << "Result: " << result << std::endl;
    return 0;
}

```

Verilog Code and testbench with simulation:



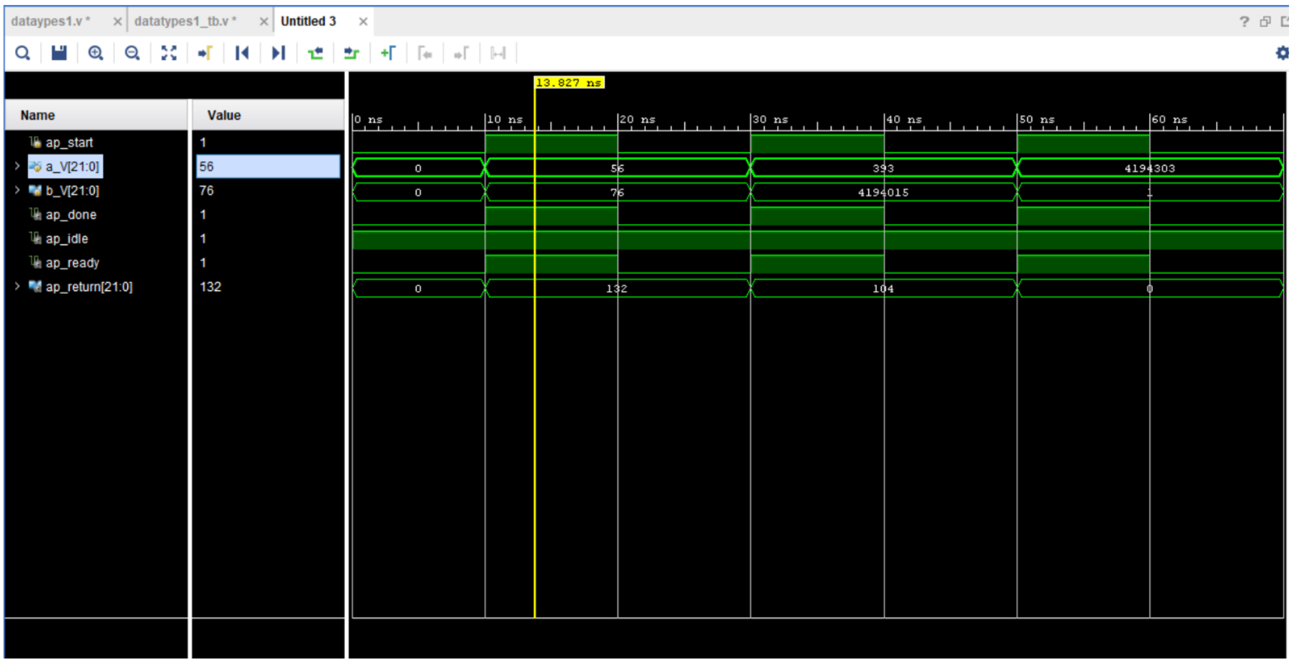
C:/Users/matta/Xilinx projects/datatypes1/datatypes1.srcs/sources_1/new/datatypes1.v

```
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="datatypes1,hls_ip_2017_4,{HLS_INPUT_TYPE=cxx,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=1,HLS_INPUT_PART=xc7a35tcpg236-1,HLS_INPUT_CLOCK=10.000000

module datatypes1 (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a_V,
        b_V,
        ap_return
);
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [21:0] a_V;
input  [21:0] b_V;
output  [21:0] ap_return;

assign ap_done = ap_start;

assign ap_idle = 1'b1;

assign ap_ready = ap_start;


assign ap_return = (b_V + a_V);

endmodule //datatypes1
```



C:/Users/matta/Xilinx projects/datatypes1/datatypes1.srcs/sim_1/new/datatypes1_tb.v

```
`timescale 1ns / 1ps
module tb_datatypes1;
  // Inputs
  reg ap_start; reg [21:0] a_V; reg [21:0] b_V;
  // Outputs
  wire ap_done; wire ap_idle; wire ap_ready; wire [21:0] ap_return;
  // Instantiate the DUT
  datatypes1 dut ( .ap_start(ap_start), .ap_done(ap_done), .ap_idle(ap_idle), .ap_ready(ap_ready), .a_V(a_V), .b_V(b_V), .ap_return(ap_return) );
  function real fixed22_to_real(input [21:0] fixed_val);
    begin
      if (fixed_val[21]) // Negative
        fixed22_to_real = -((~fixed_val + 1'b1) & 22'h3FFFFF) / 131072.0;
      else
        fixed22_to_real = fixed_val / 131072.0;
    end
  endfunction
  initial begin
    ap_start = 0;
    a_V = 0; b_V = 0;
    #10;
    // Test 1: 2.0 + 3.0
    a_V = 22'd56;  // 2.0 in Q5.17
    b_V = 22'd76;  // 3.0 in Q5.17
    ap_start = 1;
    #10 ap_start = 0; #10;
    // Test 2: 1.5 + (-0.5)
    a_V = 22'd262 * 1.5;
    b_V = -22'd578 / 2;  // -0.5
    ap_start = 1;
    #10 ap_start = 0;
    #10;
    // Test 3: Max + small
    a_V = 22'h3FFFFF;  // Largest 22-bit unsigned value
    b_V = 22'd1;
    ap_start = 1;
    #10 ap_start = 0;
    #10;
    $finish;
  end
endmodule
```



Observations:

1. We observed that when we tried coding for float type in C, FSMs were created to solve as float operation cannot be completed in a single clock cycle.'

2. **Floating-point arithmetic** (IEEE 754) is complex and involves multiple steps like alignment, normalization, rounding, and exception handling.

3. These steps cannot be completed in a **single clock cycle** due to their sequential dependencies and bit manipulations.

4. Vivado HLS maps `float` operations to **multi-stage IP cores**, typically pipelined over several cycles.

5. An **FSM (Finite State Machine)** is automatically generated to control the sequence and timing of these stages.

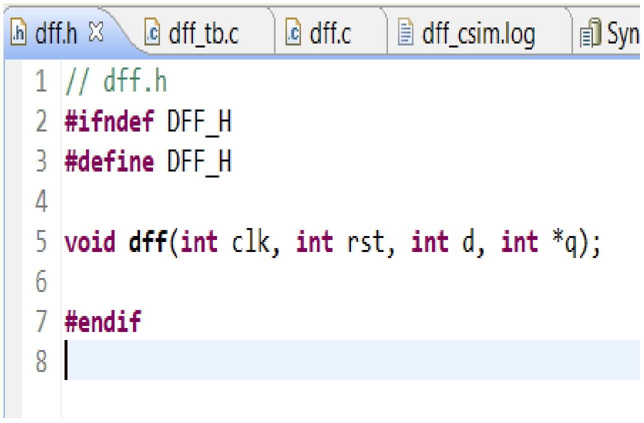
6. FSM ensures correct data flow, result readiness, and `ap_done/ap_ready` signaling during multi-cycle float operations.

To resolve this, we shift to using C++ code which makes use of *ap_fixed<W,I>* where W represents the total number of spaces used and I represents the number of integer spaces used which gives the number of fraction spaces as W-I and this allows us to limit the number of spaces it will be used. In our code, we have given 5 as the integer spaces which means 4 bits for calculation and 1 bit for sign. The larger value of W, more is its accuracy and precision. This plays an important role when conversion of decimal to binary numbers when the decimal does not have a n equivalent binary representation and needs more bits to represent it accurately.

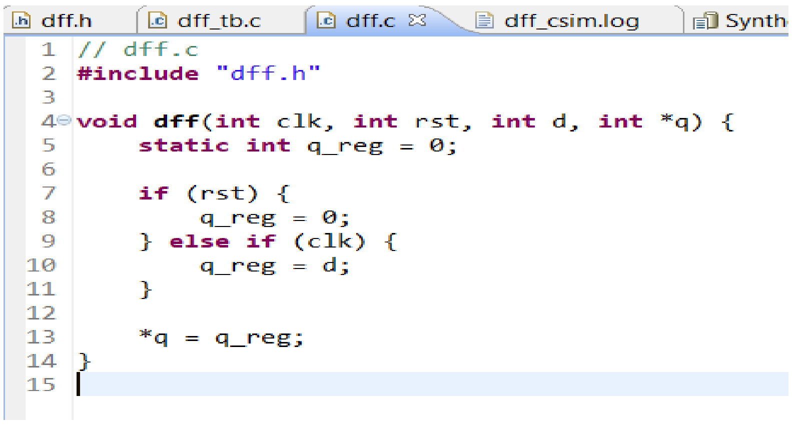
Week 2
09/06/2025

D FLIP-FLOP

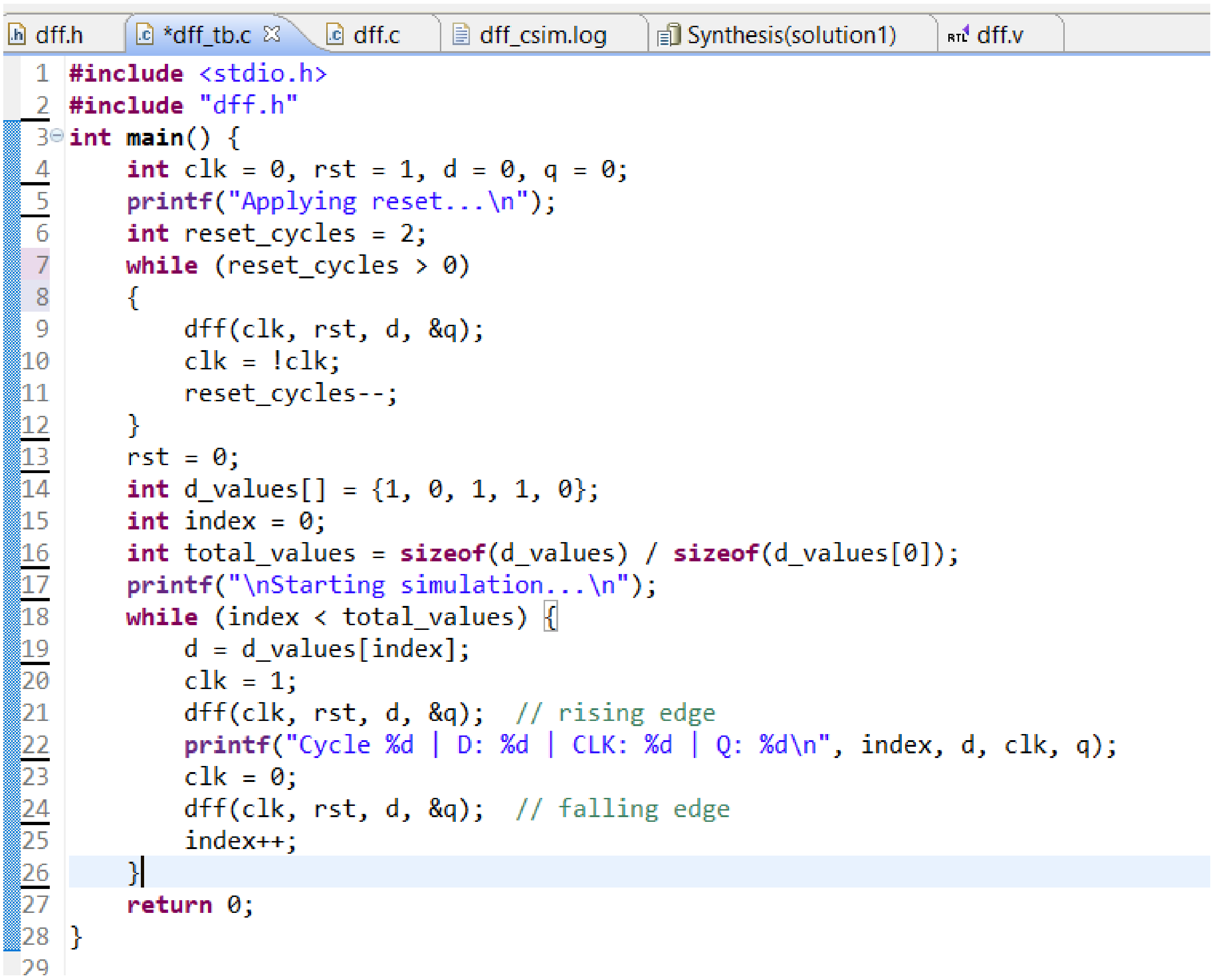
C Code and Testbench:



```c
// dff.h
#ifndef DFF_H
#define DFF_H

void dff(int clk, int rst, int d, int *q);

#endif
```



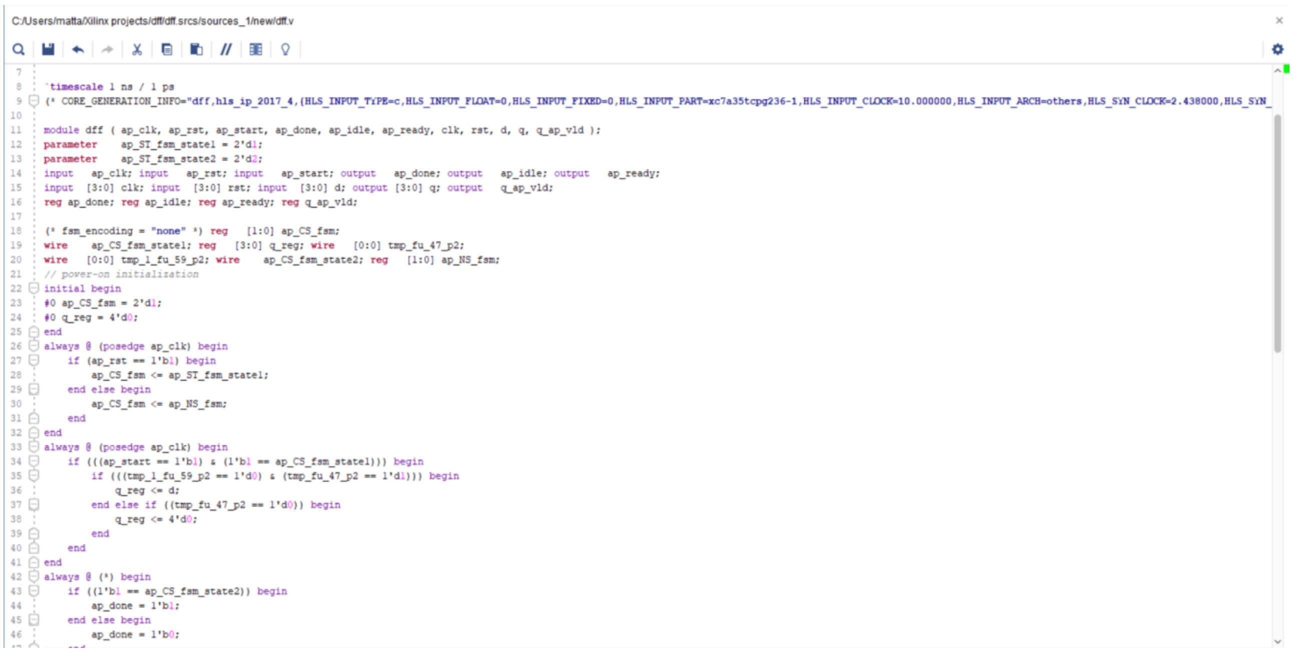
```c
// dff.c
#include "dff.h"

void dff(int clk, int rst, int d, int *q) {
    static int q_reg = 0;

    if (rst) {
        q_reg = 0;
    } else if (clk) {
        q_reg = d;
    }

    *q = q_reg;
}
```



```c
#include <stdio.h>
#include "dff.h"
int main() {
    int clk = 0, rst = 1, d = 0, q = 0;
    printf("Applying reset...\n");
    int reset_cycles = 2;
    while (reset_cycles > 0)
    {
        dff(clk, rst, d, &q);
        clk = !clk;
        reset_cycles--;
    }
    rst = 0;
    int d_values[] = {1, 0, 1, 1, 0};
    int index = 0;
    int total_values = sizeof(d_values) / sizeof(d_values[0]);
    printf("\nStarting simulation...\n");
    while (index < total_values) {
        d = d_values[index];
        clk = 1;
        dff(clk, rst, d, &q);  // rising edge
        printf("Cycle %d | D: %d | CLK: %d | Q: %d\n", index, d, clk, q);
        clk = 0;
        dff(clk, rst, d, &q);  // falling edge
        index++;
    }
    return 0;
}
```

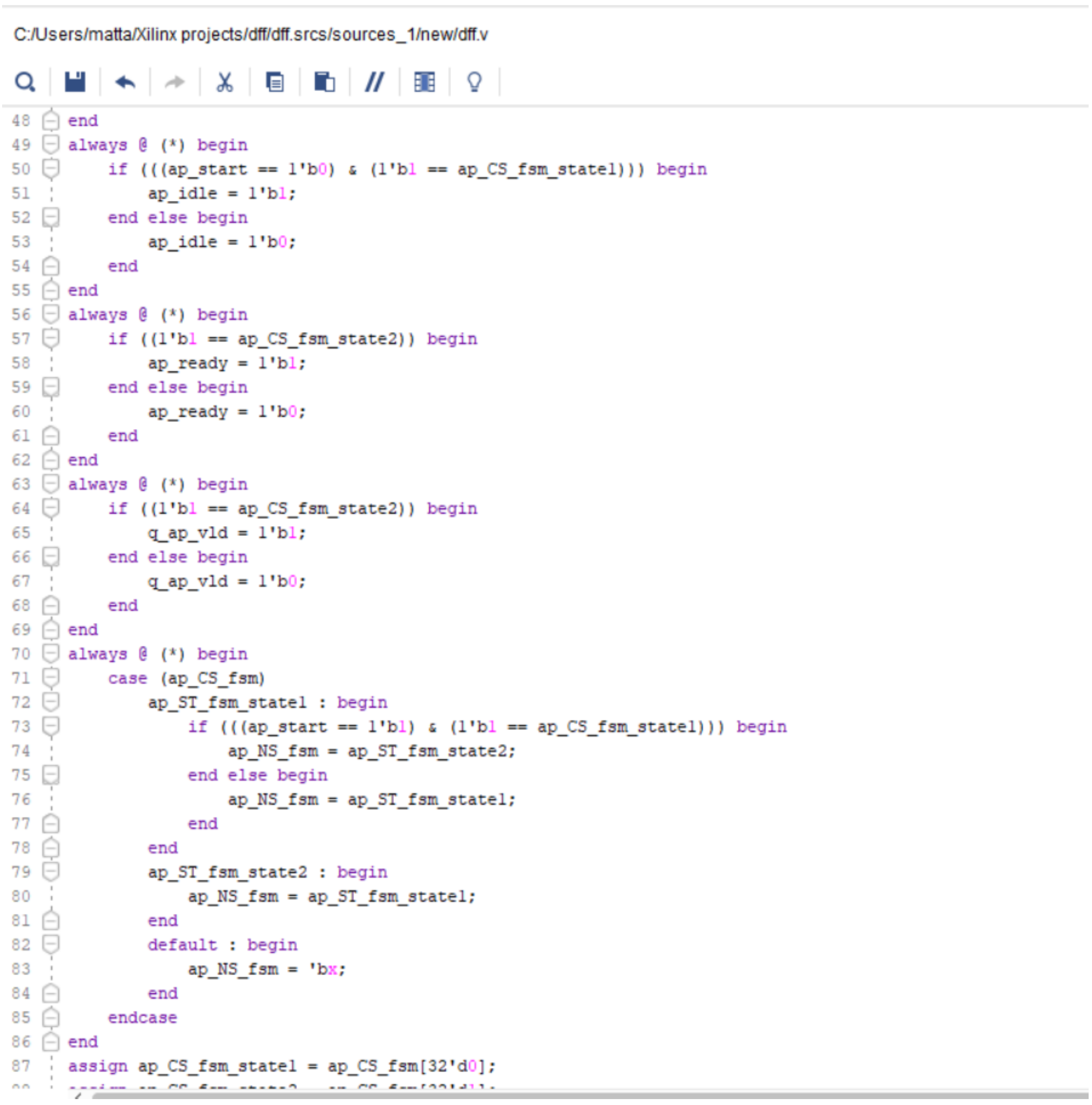
Here, we have used a header file “*dff.h*” where the function call is stored and is used in other modules”*dff.c*” and *“dff_tb.c”* where it is used for better code modularity and reusability.

Verilog code and testbench:

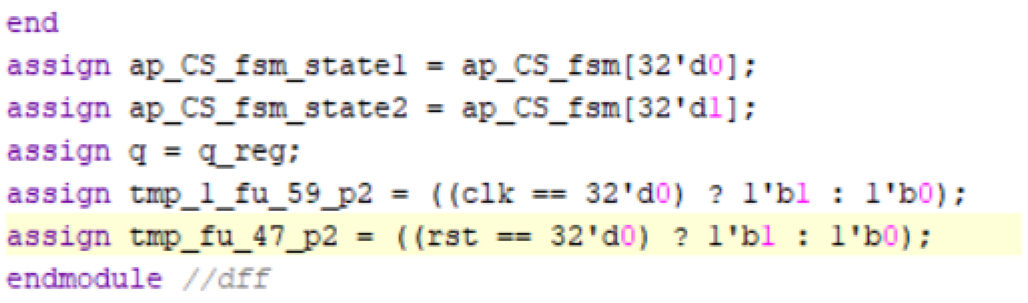


C:/Users/matta/Xilinx projects/dff/dff.srcs/sources_1/new/dff.v

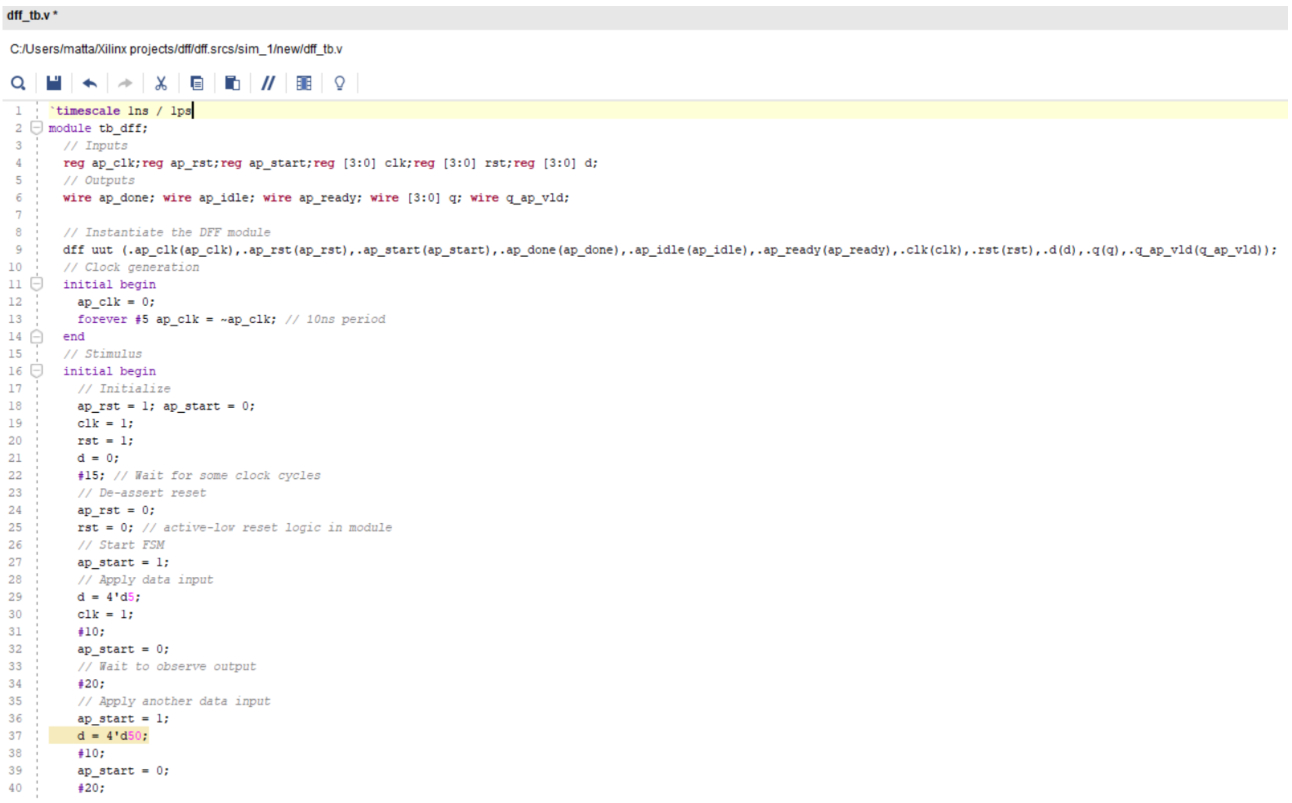
```
`timescale 1 ns / 1 ps
(* CORE_GENERATION_INFO="dff,hls_ip_2017_4,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a35tcpg236-1,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=2.438000,HLS_SYN_

module dff ( ap_clk, ap_rst, ap_start, ap_done, ap_idle, ap_ready, clk, rst, d, q, q_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  [3:0] clk; input  [3:0] rst; input  [3:0] d; output [3:0] q; output   q_ap_vld;
reg ap_done; reg ap_idle; reg ap_ready; reg q_ap_vld;

(* fsm_encoding = "none" *) reg   [1:0] ap_CS_fsm;
wire    ap_CS_fsm_state1; reg   [3:0] q_reg; wire   [0:0] tmp_fu_47_p2;
wire   [0:0] tmp_1_fu_59_p2; wire    ap_CS_fsm_state2; reg   [1:0] ap_NS_fsm;
// power-on initialization
initial begin
#0 ap_CS_fsm = 2'd1;
#0 q_reg = 4'd0;
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
        if (((tmp_1_fu_59_p2 == 1'd0) & (tmp_fu_47_p2 == 1'd1))) begin
            q_reg <= d;
        end else if ((tmp_fu_47_p2 == 1'd0)) begin
            q_reg <= 4'd0;
        end
    end
end
always @ (*) begin
    if ((1'b1 == ap_CS_fsm_state2)) begin
        ap_done = 1'b1;
    end else begin
        ap_done = 1'b0;
```


C:/Users/matta/Xilinx projects/dff/dff.srcs/sources_1/new/dff.v

```verilog
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
        q_ap_vld = 1'b1;
    end else begin
        q_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
        default : begin
            ap_NS_fsm = 'bx;
        end
    endcase
end
assign ap_CS_fsm_state1 = ap_CS_fsm[32'd0];
```

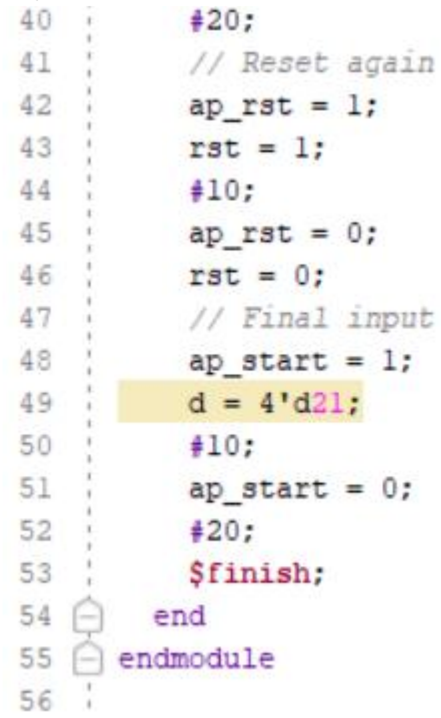


```verilog
end
assign ap_CS_fsm_state1 = ap_CS_fsm[32'd0];
assign ap_CS_fsm_state2 = ap_CS_fsm[32'd1];
assign q = q_reg;
assign tmp_1_fu_59_p2 = ((clk == 32'd0) ? 1'b1 : 1'b0);
assign tmp_fu_47_p2 = ((rst == 32'd0) ? 1'b1 : 1'b0);
endmodule //dff
```


dff_tb.v *

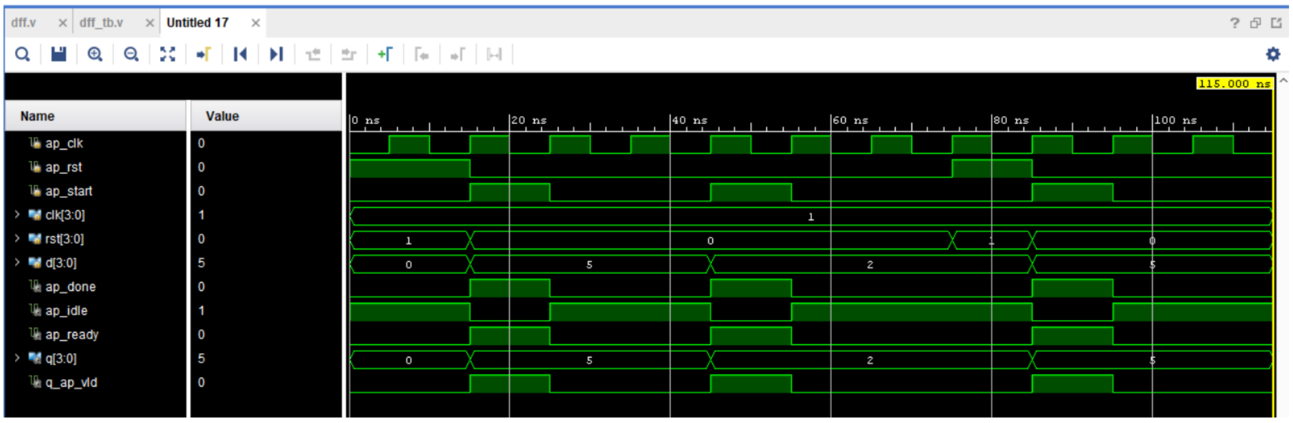
C:/Users/matta/Xilinx projects/dff/dff.srcs/sim_1/new/dff_tb.v

```verilog
`timescale 1ns / 1ps
module tb_dff;
  // Inputs
  reg ap_clk;reg ap_rst;reg ap_start;reg [3:0] clk;reg [3:0] rst;reg [3:0] d;
  // Outputs
  wire ap_done; wire ap_idle; wire ap_ready; wire [3:0] q; wire q_ap_vld;

  // Instantiate the DFF module
  dff uut (.ap_clk(ap_clk),.ap_rst(ap_rst),.ap_start(ap_start),.ap_done(ap_done),.ap_idle(ap_idle),.ap_ready(ap_ready),.clk(clk),.rst(rst),.d(d),.q(q),.q_ap_vld(q_ap_vld));
  // Clock generation
  initial begin
    ap_clk = 0;
    forever #5 ap_clk = ~ap_clk; // 10ns period
  end
  // Stimulus
  initial begin
    // Initialize
    ap_rst = 1; ap_start = 0;
    clk = 1;
    rst = 1;
    d = 0;
    #15; // Wait for some clock cycles
    // De-assert reset
    ap_rst = 0;
    rst = 0; // active-low reset logic in module
    // Start FSM
    ap_start = 1;
    // Apply data input
    d = 4'd5;
    clk = 1;
    #10;
    ap_start = 0;
    // Wait to observe output
    #20;
    // Apply another data input
    ap_start = 1;
    d = 4'd50;
    #10;
    ap_start = 0;
    #20;
```



```verilog
    #20;
    // Reset again
    ap_rst = 1;
    rst = 1;
    #10;
    ap_rst = 0;
    rst = 0;
    // Final input
    ap_start = 1;
    d = 4'd21;
    #10;
    ap_start = 0;
    #20;
    $finish;
  end
endmodule
```

Simulation:



Observations:

1. All inputs and outputs are 4 bits.
2. We observed that there are 2 clks in the code, ap_clk and an external clk. The program runs on ap_clk and the external clk is used as an enable flag.
3. It is a synchronous circuit and the first sequential circuit that we have designed.
4. Since all the inputs are 4 bits, there are 4 resets but even 1 button is enough and the same applies to clk as well.
5. Since it is a synchronous circuit, it works on a clock that is controlled by us which is given by a push button.
6. Initially the reset button is off and the data stored is latched onto q and when reset=1, data stored goes to 0.
7. We did face a few initial problems such as pin mapping or misreading the circuit as asynchronous when it was actually a synchronous circuit leading us to change the mapping of pins which we had given junk values prior.
8. When ap_start=0, the circuit doesn’t work and begins to work when ap_start goes high and ap_idle=0. After ap_start=1, we can give the input data and watch it getting stored.

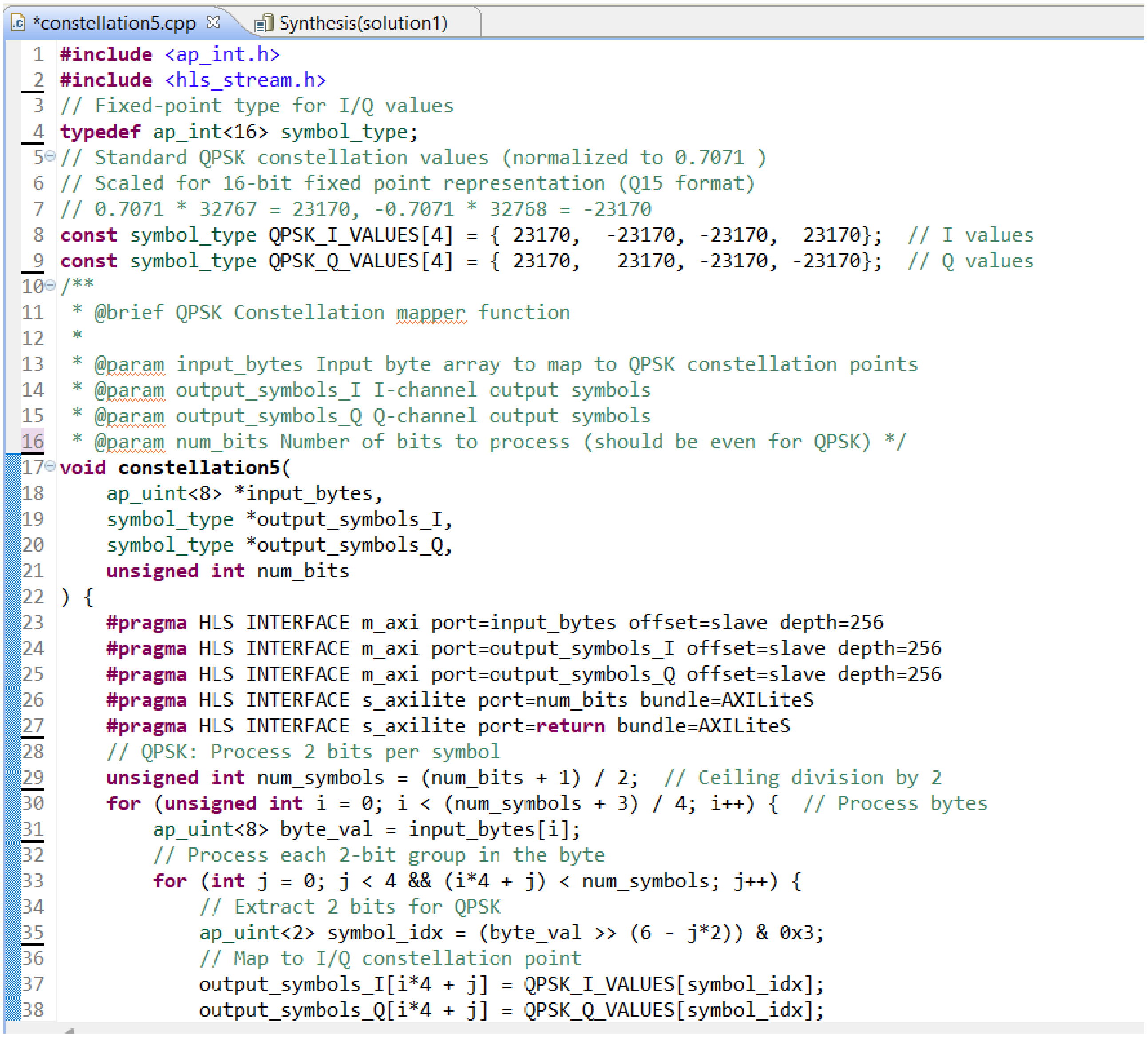
Pre-distortion Circuits

Pre-distortion Circuits

Constellation Mapper

Constellation Mapper

C code:



```
#include <ap_int.h>
#include <hls_stream.h>
// Fixed-point type for I/Q values
typedef ap_int<16> symbol_type;
// Standard QPSK constellation values (normalized to 0.7071 )
// Scaled for 16-bit fixed point representation (Q15 format)
// 0.7071 * 32767 = 23170, -0.7071 * 32768 = -23170
const symbol_type QPSK_I_VALUES[4] = { 23170,  -23170, -23170,  23170};  // I values
const symbol_type QPSK_Q_VALUES[4] = { 23170,   23170, -23170, -23170};  // Q values
/**
 * @brief QPSK Constellation mapper function
 *
 * @param input_bytes Input byte array to map to QPSK constellation points
 * @param output_symbols_I I-channel output symbols
 * @param output_symbols_Q Q-channel output symbols
 * @param num_bits Number of bits to process (should be even for QPSK) */
void constellation5(
    ap_uint<8> *input_bytes,
    symbol_type *output_symbols_I,
    symbol_type *output_symbols_Q,
    unsigned int num_bits
) {
    #pragma HLS INTERFACE m_axi port=input_bytes offset=slave depth=256
    #pragma HLS INTERFACE m_axi port=output_symbols_I offset=slave depth=256
    #pragma HLS INTERFACE m_axi port=output_symbols_Q offset=slave depth=256
    #pragma HLS INTERFACE s_axilite port=num_bits bundle=AXILiteS
    #pragma HLS INTERFACE s_axilite port=return bundle=AXILiteS
    // QPSK: Process 2 bits per symbol
    unsigned int num_symbols = (num_bits + 1) / 2;  // Ceiling division by 2
    for (unsigned int i = 0; i < (num_symbols + 3) / 4; i++) {  // Process bytes
        ap_uint<8> byte_val = input_bytes[i];
        // Process each 2-bit group in the byte
        for (int j = 0; j < 4 && (i*4 + j) < num_symbols; j++) {
            // Extract 2 bits for QPSK
            ap_uint<2> symbol_idx = (byte_val >> (6 - j*2)) & 0x3;
            // Map to I/Q constellation point
            output_symbols_I[i*4 + j] = QPSK_I_VALUES[symbol_idx];
            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*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

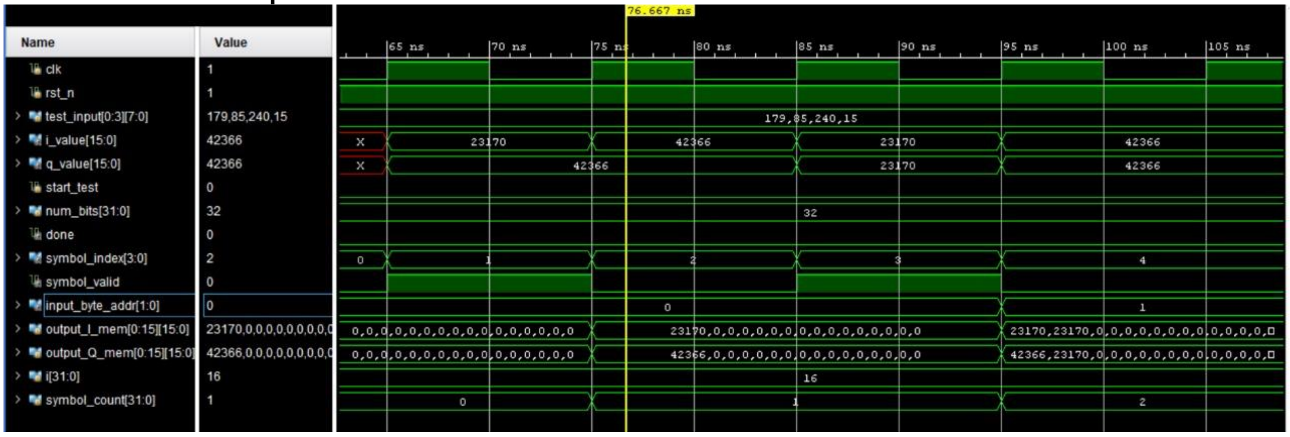
RTL constellation5_gmem2_m_axi.v

RTL constellation5_mubkb.v

RTL constellation5.v

The main file is constellation5.v .

Simulation Graph:

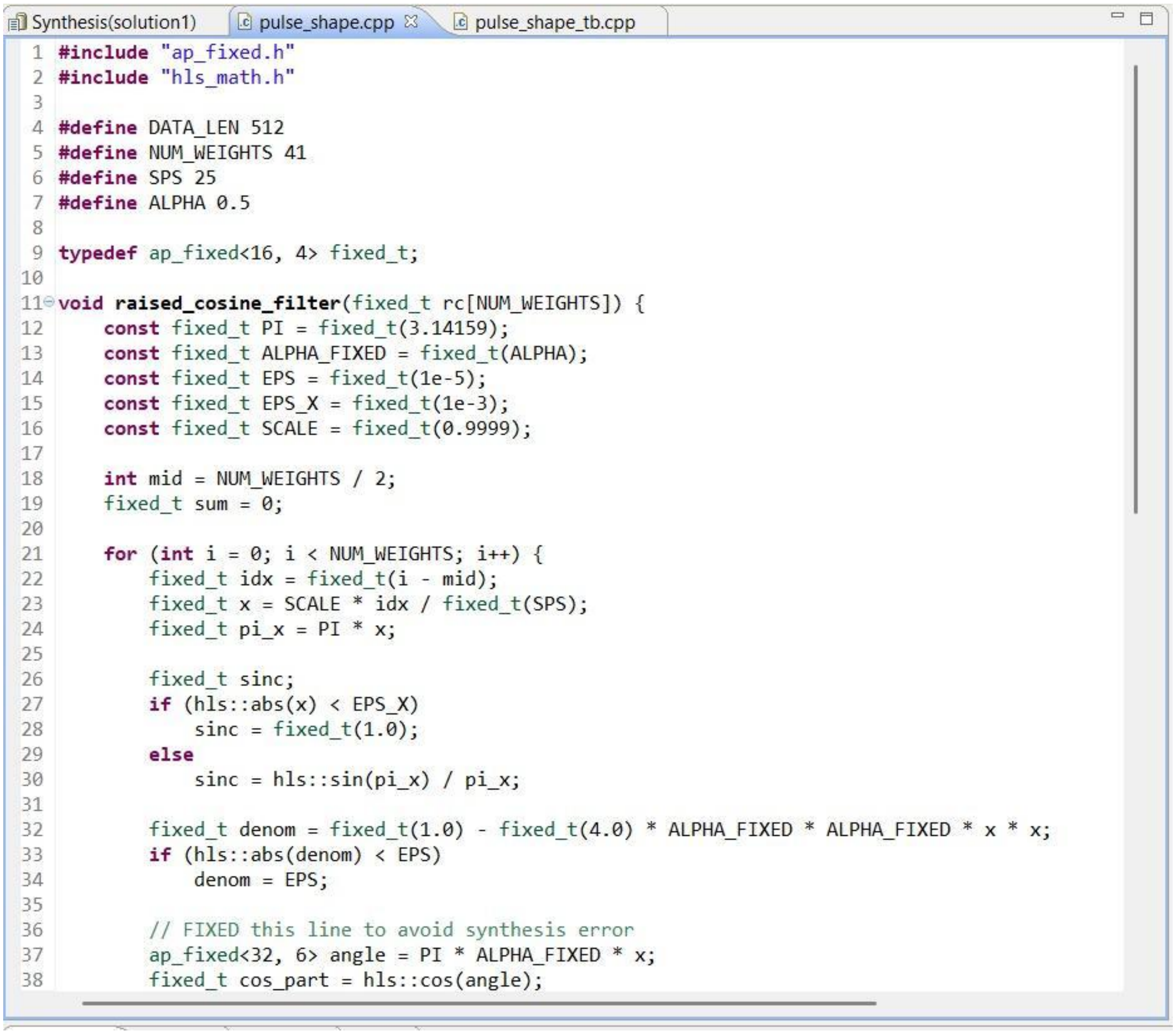


Name	Value
clk	1
rst_n	1
test_input[0:3][7:0]	179,85,240,15
i_value[15:0]	42366
q_value[15:0]	42366
start_test	0
num_bits[31:0]	32
done	0
symbol_index[3:0]	2
symbol_valid	0
input_byte_addr[1:0]	0
output_I_mem[0:15][15:0]	23170,0,0,0,0,0,0,0,0,0
output_Q_mem[0:15][15:0]	42366,0,0,0,0,0,0,0,0,0
i[31:0]	16
symbol_count[31:0]	1

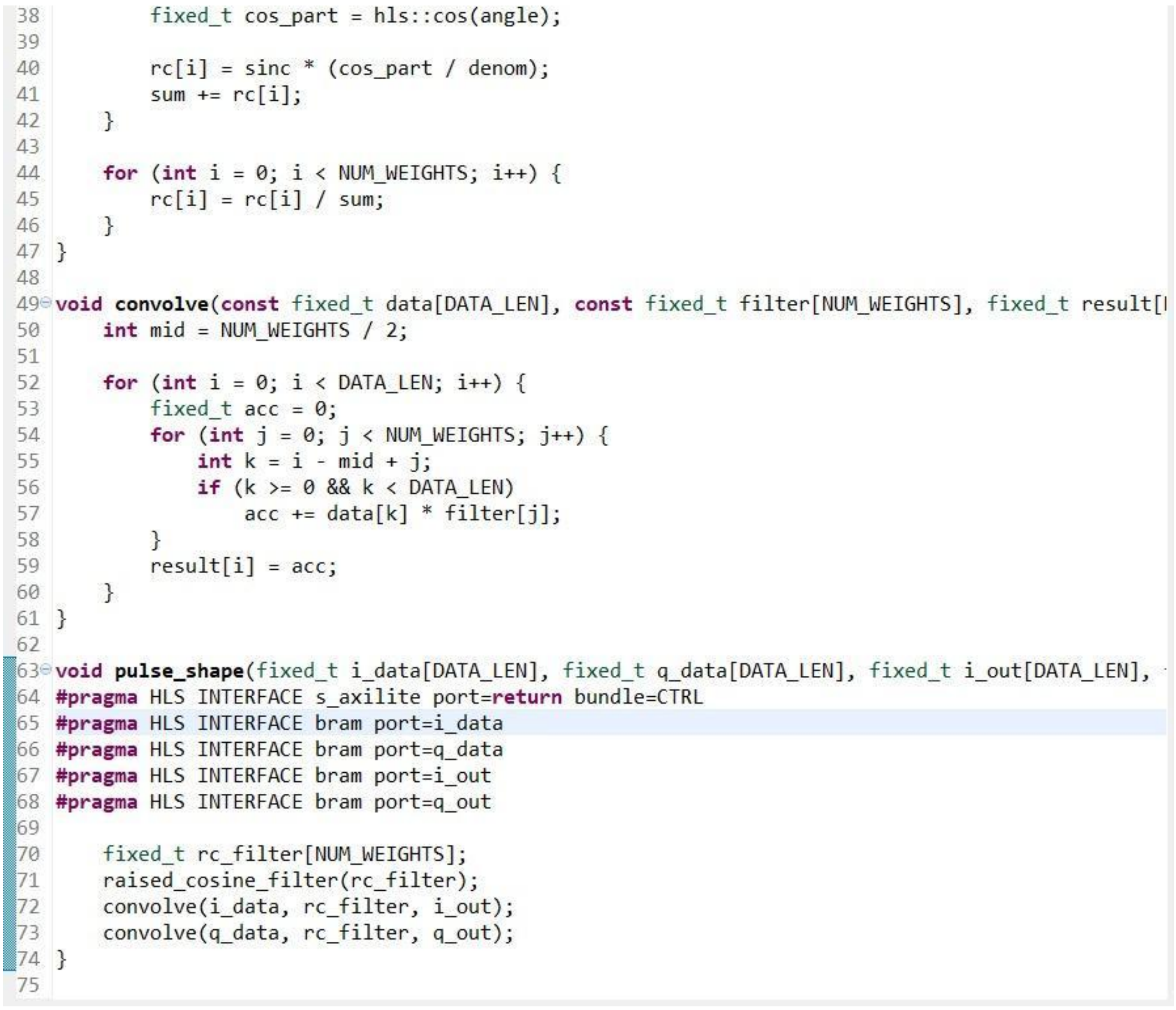
Pulse Shaping Filter

Pulse Shaping Filter

C code:



```
#include "ap_fixed.h"
#include "hls_math.h"

#define DATA_LEN 512
#define NUM_WEIGHTS 41
#define SPS 25
#define ALPHA 0.5

typedef ap_fixed<16, 4> fixed_t;

void raised_cosine_filter(fixed_t rc[NUM_WEIGHTS]) {
    const fixed_t PI = fixed_t(3.14159);
    const fixed_t ALPHA_FIXED = fixed_t(ALPHA);
    const fixed_t EPS = fixed_t(1e-5);
    const fixed_t EPS_X = fixed_t(1e-3);
    const fixed_t SCALE = fixed_t(0.9999);

    int mid = NUM_WEIGHTS / 2;
    fixed_t sum = 0;

    for (int i = 0; i < NUM_WEIGHTS; i++) {
        fixed_t idx = fixed_t(i - mid);
        fixed_t x = SCALE * idx / fixed_t(SPS);
        fixed_t pi_x = PI * x;

        fixed_t sinc;
        if (hls::abs(x) < EPS_X)
            sinc = fixed_t(1.0);
        else
            sinc = hls::sin(pi_x) / pi_x;

        fixed_t denom = fixed_t(1.0) - fixed_t(4.0) * ALPHA_FIXED * ALPHA_FIXED * x * x;
        if (hls::abs(denom) < EPS)
            denom = EPS;

        // FIXED this line to avoid synthesis error
        ap_fixed<32, 6> angle = PI * ALPHA_FIXED * x;
        fixed_t cos_part = hls::cos(angle);
```



```
        fixed_t cos_part = hls::cos(angle);

        rc[i] = sinc * (cos_part / denom);
        sum += rc[i];
    }

    for (int i = 0; i < NUM_WEIGHTS; i++) {
        rc[i] = rc[i] / sum;
    }
}

void convolve(const fixed_t data[DATA_LEN], const fixed_t filter[NUM_WEIGHTS], fixed_t result[
    int mid = NUM_WEIGHTS / 2;

    for (int i = 0; i < DATA_LEN; i++) {
        fixed_t acc = 0;
        for (int j = 0; j < NUM_WEIGHTS; j++) {
            int k = i - mid + j;
            if (k >= 0 && k < DATA_LEN)
                acc += data[k] * filter[j];
        }
        result[i] = acc;
    }
}

void pulse_shape(fixed_t i_data[DATA_LEN], fixed_t q_data[DATA_LEN], fixed_t i_out[DATA_LEN],
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL
#pragma HLS INTERFACE bram port=i_data
#pragma HLS INTERFACE bram port=q_data
#pragma HLS INTERFACE bram port=i_out
#pragma HLS INTERFACE bram port=q_out

    fixed_t rc_filter[NUM_WEIGHTS];
    raised_cosine_filter(rc_filter);
    convolve(i_data, rc_filter, i_out);
    convolve(q_data, rc_filter, q_out);
}

```

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 checks
- Accumulates sum of products for FIR convolution

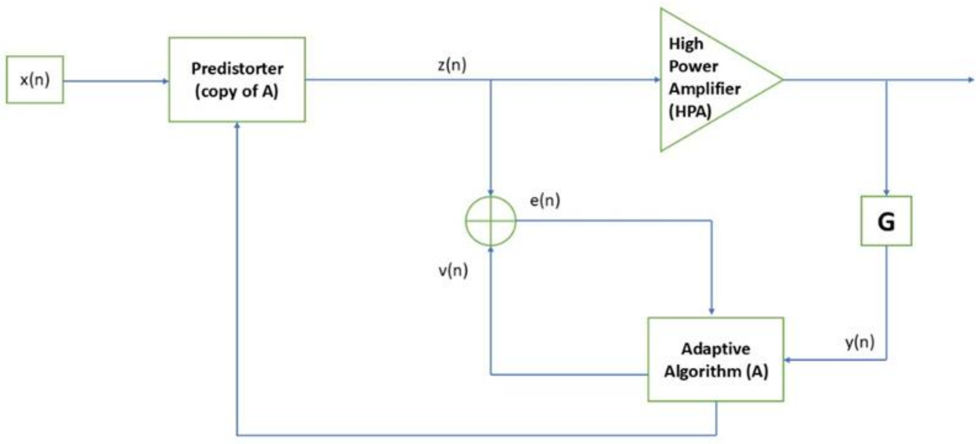
Efficiency:

- 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

DIGITAL PRE DISTORTION BLOCK

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



Indirect Learning Architecture

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.

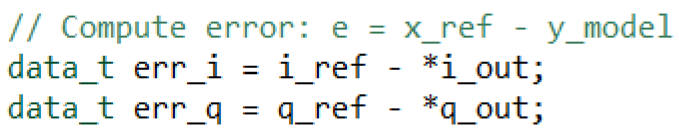
Efficiency :

Generation of coefficients is done recursively using the LMS method.

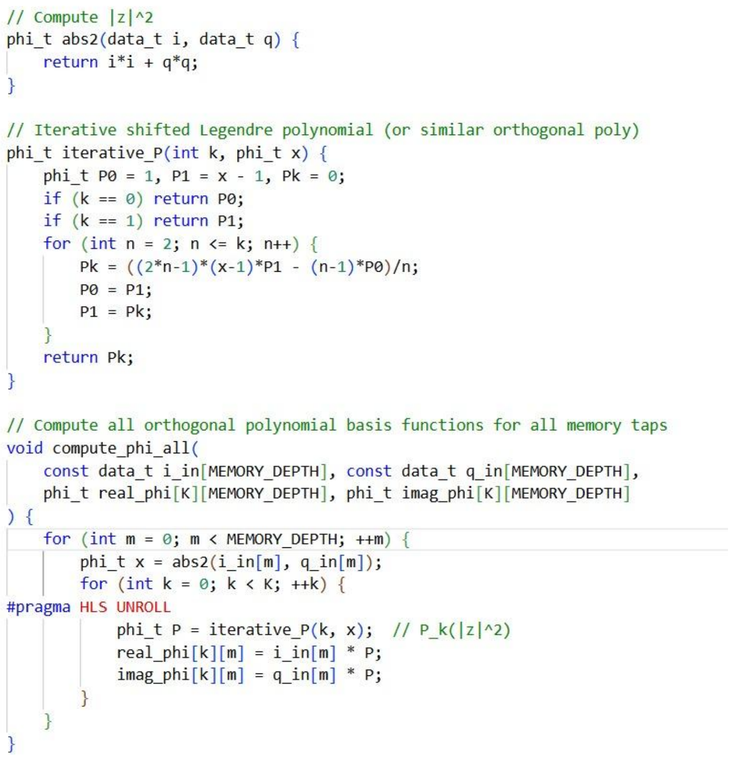
Pre-Distortion Algorithm

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:

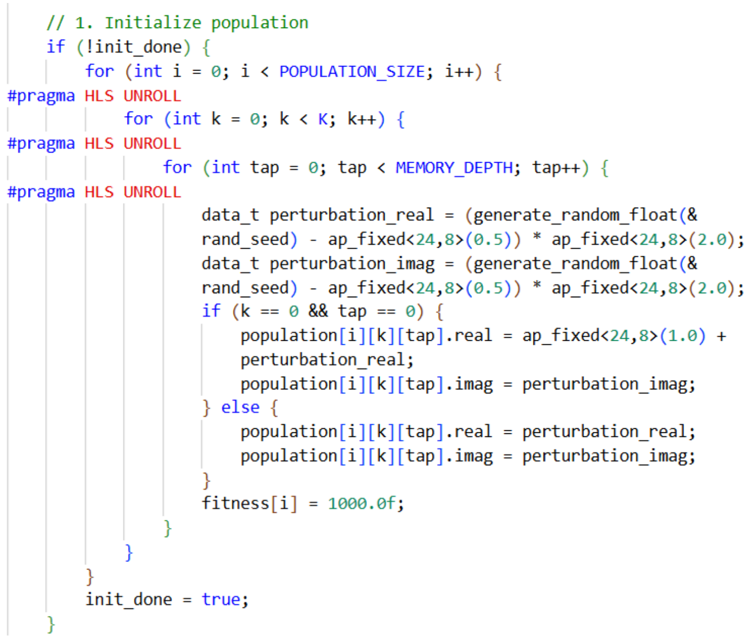
Delayed pulse shaped I and Q inputs (x), amplified output (y) from the feedback loop.

Outputs:

Most fit vector value with highest Digital Pre Distortion.

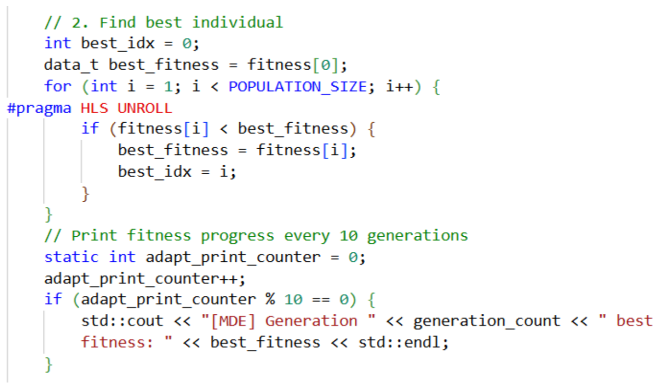
Process :

Generate a population of values from the feedback outputs.



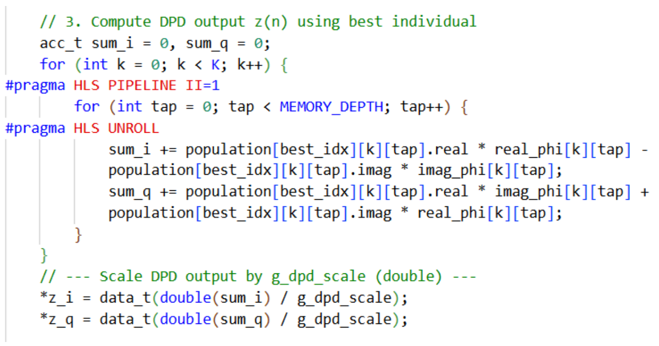
```
    // 1. Initialize population
    if (!init_done) {
        for (int i = 0; i < POPULATION_SIZE; i++) {
#pragma HLS UNROLL
            for (int k = 0; k < K; k++) {
#pragma HLS UNROLL
                for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
#pragma HLS UNROLL
                    data_t perturbation_real = (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.



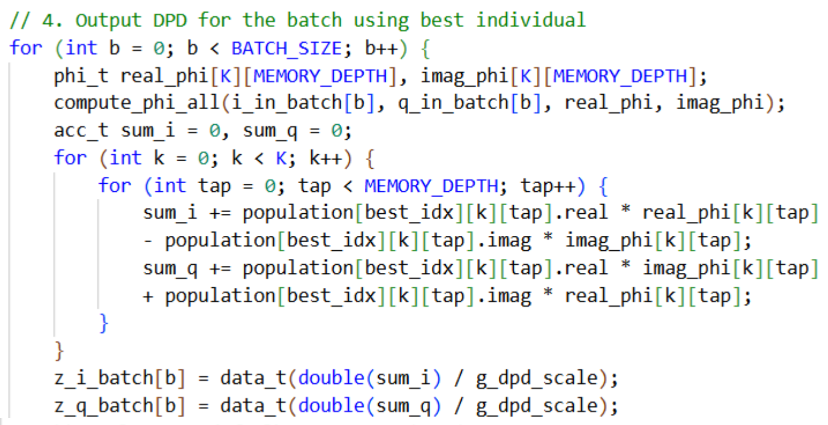
```
    // 2. Find best individual
    int best_idx = 0;
    data_t best_fitness = fitness[0];
    for (int i = 1; i < POPULATION_SIZE; i++) {
#pragma HLS UNROLL
        if (fitness[i] < best_fitness) {
            best_fitness = fitness[i];
            best_idx = i;
        }
    }
    // Print fitness 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

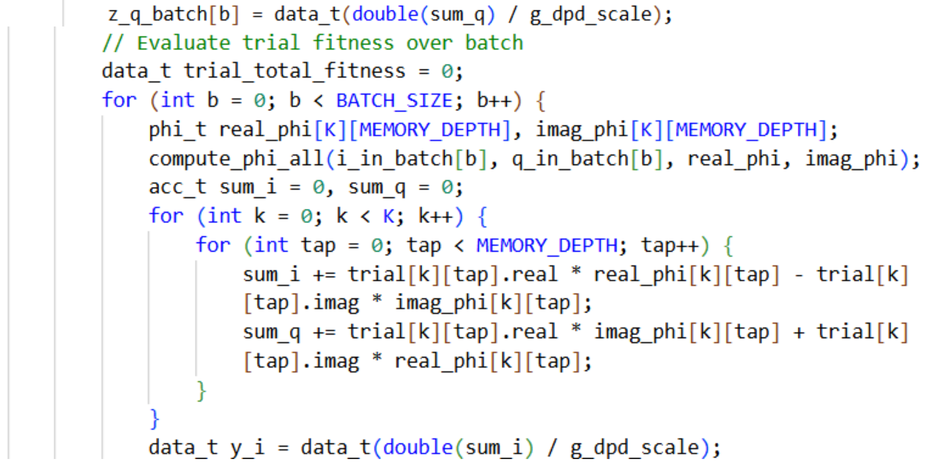


```
    // 3. Compute DPD output z(n) using best individual
    acc_t sum_i = 0, sum_q = 0;
    for (int k = 0; k < K; k++) {
#pragma HLS PIPELINE II=1
        for (int tap = 0; tap < MEMORY_DEPTH; tap++) {
#pragma HLS UNROLL
            sum_i += population[best_idx][k][tap].real * real_phi[k][tap] -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

The vectors are mutated and evaluated for fitness.

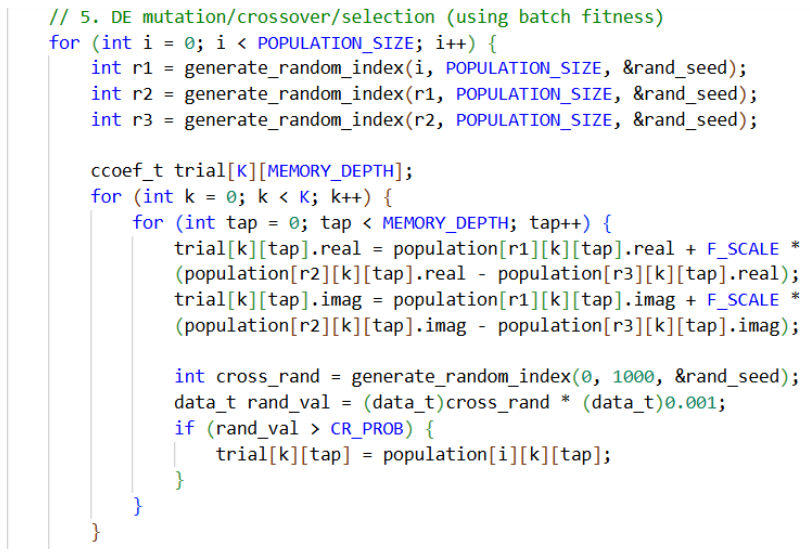


```
// 4. Output DPD for the batch using best individual
for (int b = 0; b < BATCH_SIZE; b++) {
    phi_t real_phi[K][MEMORY_DEPTH], imag_phi[K][MEMORY_DEPTH];
    compute_phi_all(i_in_batch[b], q_in_batch[b], real_phi, imag_phi);
    acc_t sum_i = 0, sum_q = 0;
    for (int k = 0; k < K; k++) {
        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);
```

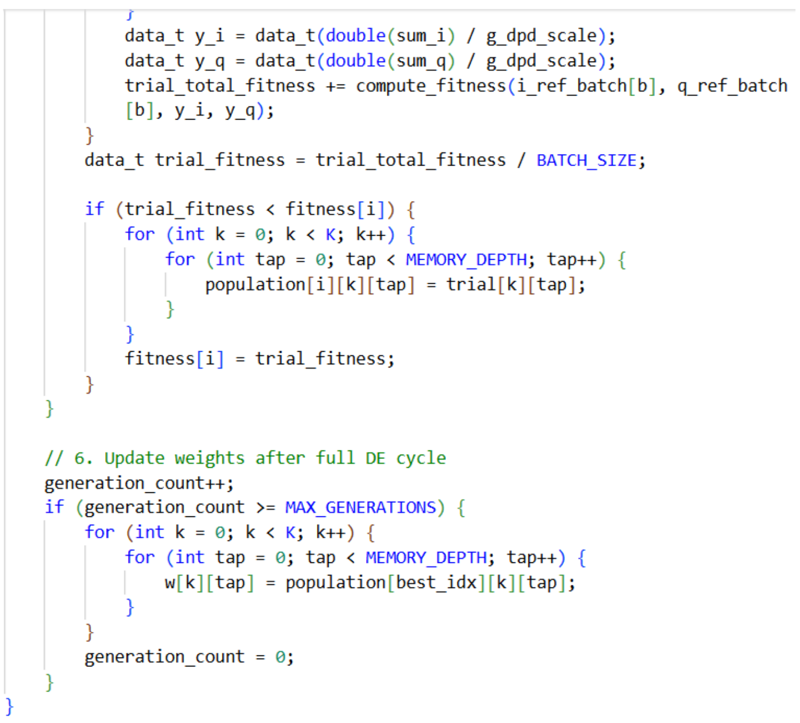


```
        // Evaluate trial fitness over batch
        data_t trial_total_fitness = 0;
        for (int b = 0; b < BATCH_SIZE; b++) {
            phi_t real_phi[K][MEMORY_DEPTH], imag_phi[K][MEMORY_DEPTH];
            compute_phi_all(i_in_batch[b], q_in_batch[b], real_phi, imag_phi);
            acc_t 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.



```
// 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];
            }
        }
    }
```



```
            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;
    }
}
```

DAC

Digital To Analog Converter

Inputs :

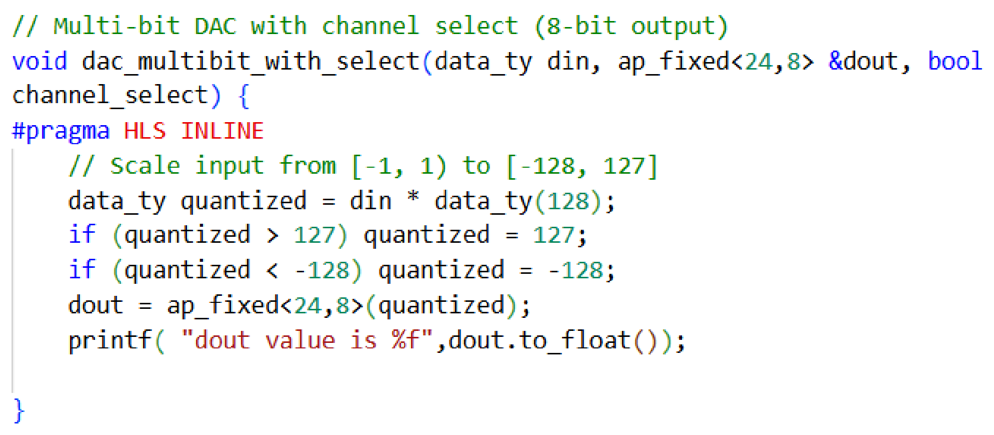
It takes discrete digital inputs from the pre-distorter.

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

Quadrature Modulation

Inputs:

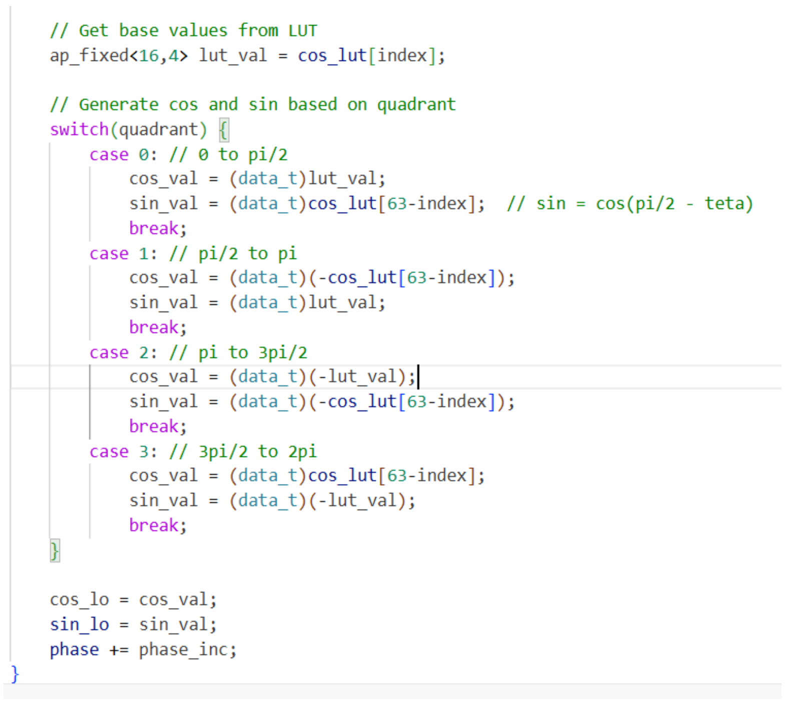
Analog discrete I and Q values generated by the DAC (Digital to Analog Converter).

Outputs:

Real RF quadrature modulated signal with both magnitude and phase preserved .

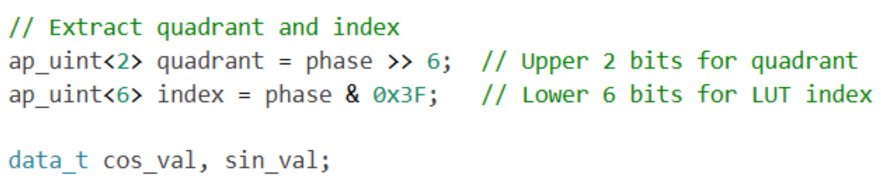
Process:

The NCO (Numerically Controlled Oscillator) is used to generate 64 cos and sin values (in the range of 0-pi/2).



```
    // 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.



```
// 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;
```

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

Digital Upconverter

Inputs:

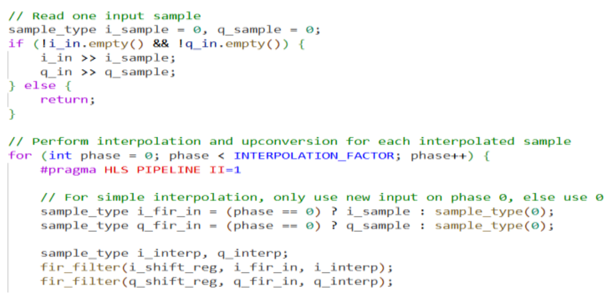
Input to the block is taken from the output of the Quadrature Modulator

Outputs:

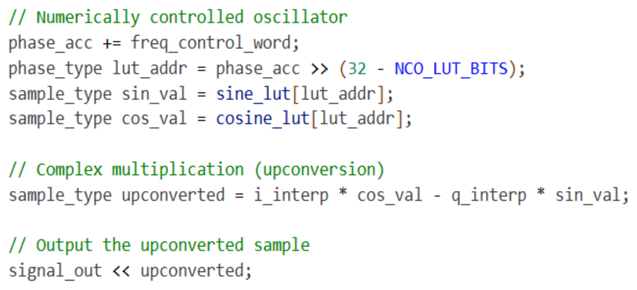
Outputs of the block are upconverted and upsampled filtered RF realbaseband values.

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

Power Amplifier

Inputs:

The inputs to the PA are RF realbaseband values from the Upconverter

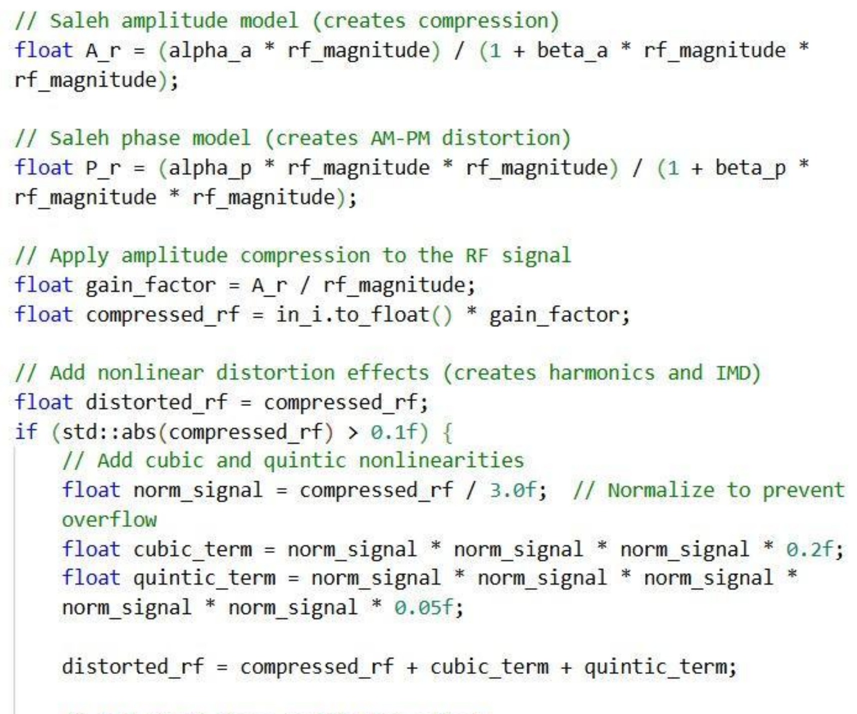
Phase Distortions are simulated to substitute Q

Outputs:

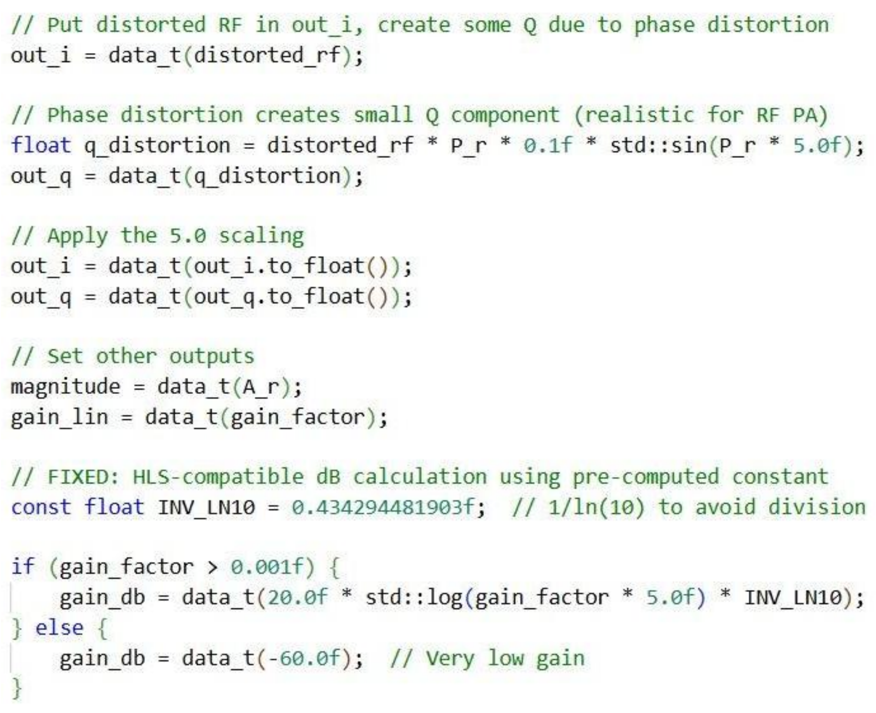
The output are amplified I and Q magnitude values, Linear Gain and Gain in dB scale.

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;
```



```
// 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
}
```

ADC

Analog to Digital Converter

Inputs:

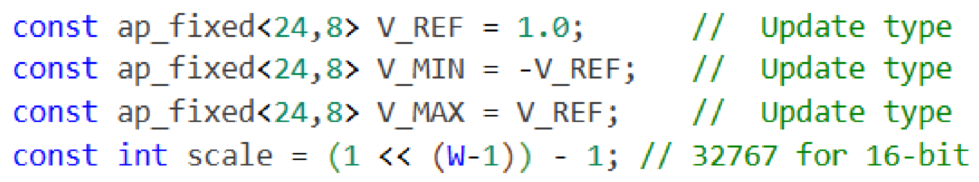
Down converted and demodulated analog I and Q value arrays.

Outputs:

Discrete digital I and Q value arrays.

Process:

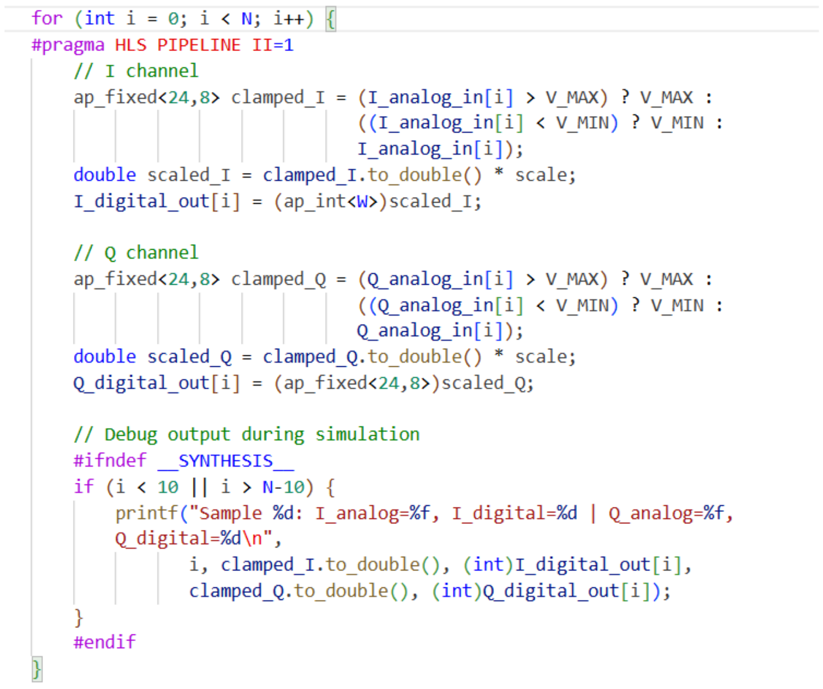
Based on Vref values the maximum and minimum values of the range are computed.



```
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
}
```

DDC

Demodulation and Downconversion

Inputs:

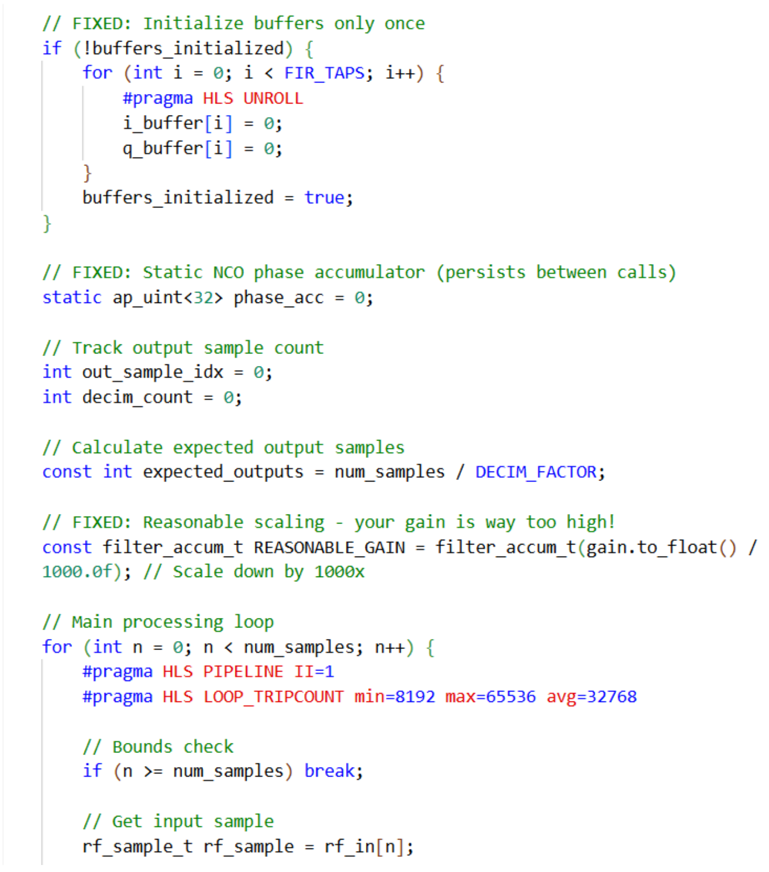
Amplified (RF) values from the Power Amplifier.

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];
```