Overview of Vivado HLS

Date: 2-6-2025

About Vivado HLS:

- Xilinx Vivado HLS (High-Level Synthesis) tool transforms 'C' specification into RTL implementation that we can synthesize into Xilinx FPGA.
- We can write C specification in C, C++ or systemC and FPGA provides massively parallel architecture with benefits in performance, cost and power over traditional processors.
- It's a part of the **Vivado Design Suite**, primarily used for designing and optimizing hardware accelerators, signal processing blocks, and custom logic for FPGAs.

Benefits of Xilinx Vivado HLS:

- Improved productivity for hardware designers and improved system performance for software designers.
- Using this tool will allow to:
  1. Develop algorithms and verify at C-level.
  2. Control the C synthesis process through optimization directives.
  3. Create multiple implementations from C using optimization directives.
  4. Create readable and portable C source code.

Basics of Xilinx Vivado HLS:

- Three phases in HLS:
  1. Scheduling
  2. Binding
  3. Control logic Extraction

1. **Scheduling:**
   - Determines which operations occur during each clock cycle based on:
     1) Length of the clock cycle
     2) Time it takes the operation to complete as defined by target device.
     3) User-specified optimization directives.
   - If the clock cycle is longer or faster FPGA is targeted then more operations are completed within single clock cycle.
   - If the clock cycle is shorter of slower FPGA is targeted then HLS automatically schedules the operation over more clock cycles.
2. **Binding:**
   - Determines which hardware resource implements each scheduled operation.
   - To improve the optimal solution, HLS uses information about target device.

3. **Control Logic Extraction:**
   - Extracts the control logic to create FSM that sequences the operation in RTL Design.

HLS Synthesizes the C code as follows:-

• Top-level function arguments synthesize into RTL I/O ports

• C functions synthesize into blocks in the RTL hierarchy

• Loops in the C functions are kept rolled by default

• Arrays in the C code synthesize into block RAM or UltraRAM in the final FPGA design

• Area: Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers etc.

• Latency: Number of clock cycles required for the function to compute all output values.

• Initiation interval (II): Number of clock cycles before the function can accept new input data.

• Loop iteration latency: Number of clock cycles it takes to complete one iteration of the loop.

• Loop initiation interval: Number of clock cycles before the next iteration of the loop starts to process data.

• Loop latency: Number of cycles to execute all iterations of the loop.

Vivado HLS design flow (Procedure):

1. Compile, execute (simulate), and debug the C algorithm.

2. Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives.

3. Generate comprehensive reports and analyze the design.

4. Verify the RTL implementation using a pushbutton flow.

5. Package the RTL implementation into a selection of IP formats.

Components of Vivado HLS Project:

1. Source Files: C/C++/SystemC

2. Testbench: For C-level simulation
3. Directives File: Contains optimization pragmas
4. Solution Folder: Synthesis and simulation results
5. IP Output: Verilog/VHDL + metadata
6. TCL Scripts: For automation and integration

Why Do We Use Vivado HLS?

Traditional digital design requires writing HDL code, which is low-level, complex, and time-consuming. Vivado HLS simplifies this process by:

1. **Reducing Development Time**: High-level programming is faster and more intuitive than RTL design.
2. **Allowing Algorithm Reuse**: You can reuse existing C/C++ algorithms from software and directly implement them in hardware.
3. **Improving Productivity**: Engineers can quickly explore different hardware architectures by adjusting synthesis directives.
4. **Automatic Optimizations**: Vivado HLS provides performance tuning options such as loop unrolling, pipelining, and dataflow optimizations.
5. **Verification Support**: Easier simulation and debugging using C/C++ test benches, and co-simulation with HDL.

Where is Vivado HLS Used?

Vivado HLS is used in **FPGA-based system design**, especially where **performance, power-efficiency, and flexibility** are critical. It is common in:

1. Signal and Image Processing

Used to implement filters, compression, edge detection, etc.

2. Wireless Communication Systems

Baseband processing, encoding/decoding, and digital modulation are implemented efficiently on FPGAs using HLS.

3. Computer Vision and AI

For real-time processing in embedded systems (e.g., drones, surveillance, automotive vision).

4. High-Performance Computing (HPC)

FPGA accelerators for compute-intensive tasks.

5. Automotive and Aerospace

Control systems and safety-critical applications benefit from the performance and reliability of FPGA-based designs.

Advantages and Disadvantages of Xilinx Vivado HLS

Aspect	**Advantage**	**Disadvantage**
Abstraction	High-level C/C++ design	Low-level tuning may still be needed
Speed	Fast development & prototyping	Less efficient than hand-tuned RTL in some cases
Optimization	Powerful with directives	Learning curve for efficient use
Tool Integration	Seamless with Vivado & IP Integrator	Xilinx-specific (not cross-vendor)
Debugging	Easy C/C++ simulation	RTL debugging is harder
Licensing	Available in Vivado HLx editions	Paid license for advanced devices

Comparison between Vivado HLS and Handwritten RTL

Aspect	**Vivado HLS**	**Handwritten RTL**
Development Speed	Fast	Slower
Performance Tuning	Directive-based	Manual, detailed
Code Reusability	High	Low
Debugging	C-level simulation	Waveform-based
Portability	High (source level)	Low
Hardware Control	Less granular	Full control

Components of Vivado HLS Project:-

1. Source Files: C/C++/SystemC

2. Testbench: For C-level simulation
3. Directives File: Contains optimization pragmas
4. Solution Folder: Synthesis and simulation results
5. IP Output: Verilog/VHDL + metadata
6. TCL Scripts: For automation and integration

Conclusion:-

Vivado HLS is a game-changer in hardware design:

- It brings **software-like simplicity** to **FPGA development**.
- While not a total replacement for HDL, it greatly speeds up the process.
- Perfect for high-performance, real-time applications where **custom acceleration** is needed (like image processing, AI, DSP).
- Requires a balance of **C programming** skills and **hardware knowledge** to fully unlock its potential.

References:

1. https://www.xilinx.com/support/documents/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf
2. https://ieeexplore.ieee.org/document/10143853

Date: 3-6-2025

Example code: - C code to check whether a number is positive, negative or equal to zero

Steps:-

1. Open Vivado HLS tool and click on create new project.
2. In **Project Configuration Window**, Name the project and click on next.
3. In **Add/Remove files Window**,

   Create the source file and choose the top function. The top function name must be the same as the function in source file and testbench file.

   Next create a testbench file.

4. In **Solution Configuration window**, click finish. Then project window appears.
5. Then modify the Parts Selection window.
6. Next Modify source and testbench file. Call the function in the test bench file and use print statement to verify output.
7. Click on Run C simulation icon, in order to build and debug C codes.

Source File:-

checknum.c | test_num.c | _tmainCRTStart | Synthesis(solut | check_number.v

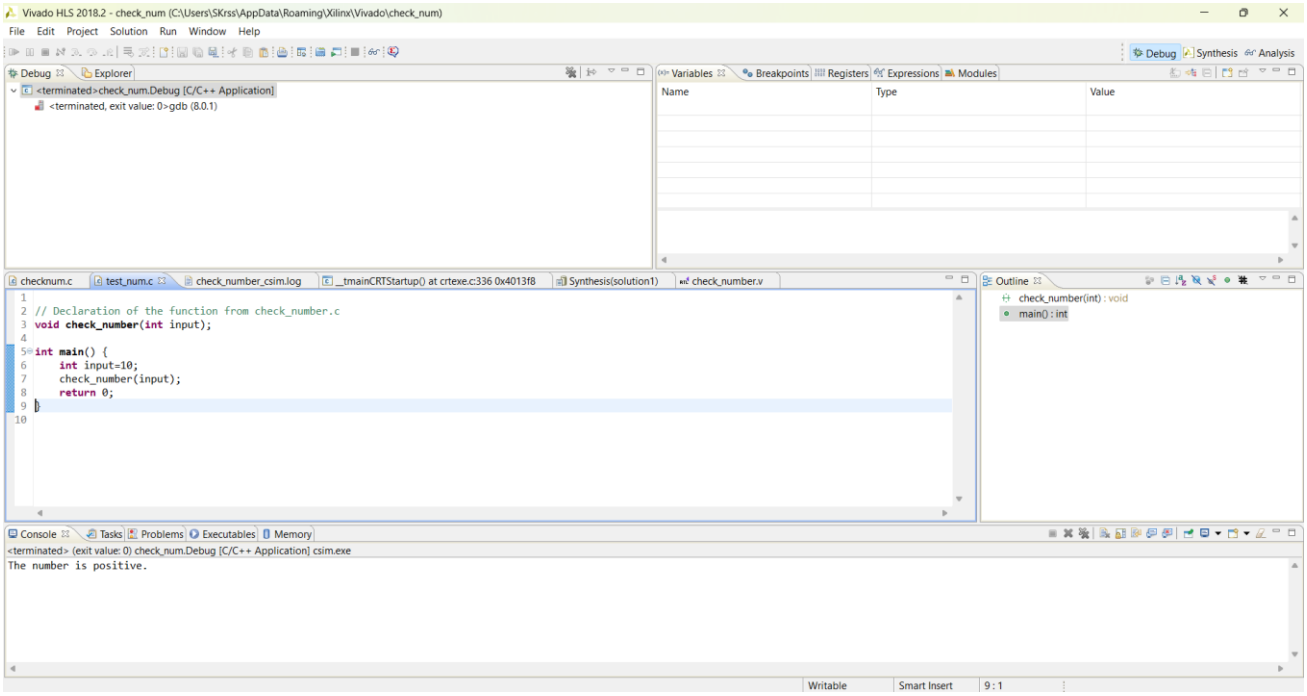
```c

#include <stdio.h>
void check_number(int input) {
    if (input > 0) {
        printf("The number is positive.\n");
    }
    else if (input < 0) {
        printf("The number is negative.\n");
    }
    else {
        printf("The number is zero.\n");
    }
}

```

Testbench File:-

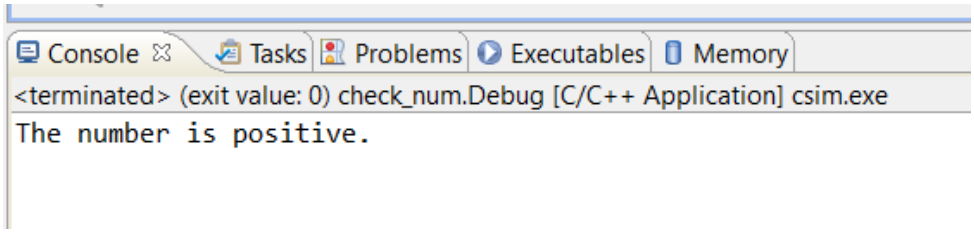
checknum.c | test_num.c | _tmainCRTStart | Synthesis(solut | check_number.v

```c

// Declaration of the function from check_number.c
void check_number(int input);

int main() {
    int input=10;
    check_number(input);
    return 0;
}

```

Debug Window:-



Output Console:-



Verilog Window:-

Synthesis Report:-

Synthesis Report for 'check_number'

General Information

Date:	Tue Jun 3 10:06:53 2025
Version:	2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project:	check_num
Solution:	solution1
Product family:	aartix7
Target device:	xa7a12tcsg325-1q

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	0.000	1.25

Latency (clock cycles)

Summary

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

Detail

Instance

N/A

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	0
Available	40	40	16000	8000
Utilization (%)	0	0	0	0

Detail

Instance

DSP48

Memory

FIFO

Expression

Multiplexer

N/A

Register

N/A

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_start	in	1	ap_ctrl_hs	check_number	return value
ap_done	out	1	ap_ctrl_hs	check_number	return value
ap_idle	out	1	ap_ctrl_hs	check_number	return value
ap_ready	out	1	ap_ctrl_hs	check_number	return value
input_r	in	32	ap_none	input_r	scalar

Export the report(.html) using the Export Wizard

Open Analysis Perspective Analysis Perspective

User Defined Inputs in HLS:-

Next instead of giving any value directly to testbench, use scanf function so that we can check for any integer whether is it positive or negative but when we use this in testbench, it throws an error because this features does not allow to take the inputs by using scanf.

Instead using command line arguments in testbench is the best choice, while debugging it takes the inputs and shows the output in the console.

Why scanf is Not Suitable in HLS (High-Level Synthesis)

Using scanf for user input is not supported or recommended in HLS environments due to the following reasons:

- **Undefined Data Size**: The scanf function does not have a fixed input size at compile time, making it unsuitable for hardware synthesis where deterministic behavior is required.
- **Lack of Predefined Inputs**: In HLS, all inputs to a function must be explicitly defined before synthesis. scanf relies on runtime interaction, which cannot be synthesized into hardware logic.

- **Ambiguous Input Handling**: The way scanf reads and interprets user input at runtime introduces ambiguity, which is not compatible with the strict, deterministic requirements of hardware description.

Alternative: Command-Line Arguments

Instead of scanf, **command-line arguments** are used in HLS-compatible test benches and software-driven testing because:

- **Defined at Compile Time**: Inputs are passed at the start of program execution, which satisfies the requirement for predefined input data in HLS.
- **Explicit Input Size and Count**: The use of argc (argument count) and argv (argument values) provides a clear structure to the number and size of inputs.
- **Deterministic Behavior**: Since the inputs are passed as arguments, the function operates in a controlled and predictable way — ideal for synthesis and verification.
- **Simulation-Friendly**: Command-line inputs make it easier to automate testing and integrate with simulation scripts in the HLS toolchain.

Source File:

checknum.c | test_num.c | check_number_cs | __tmainCRTStart | check_number.v

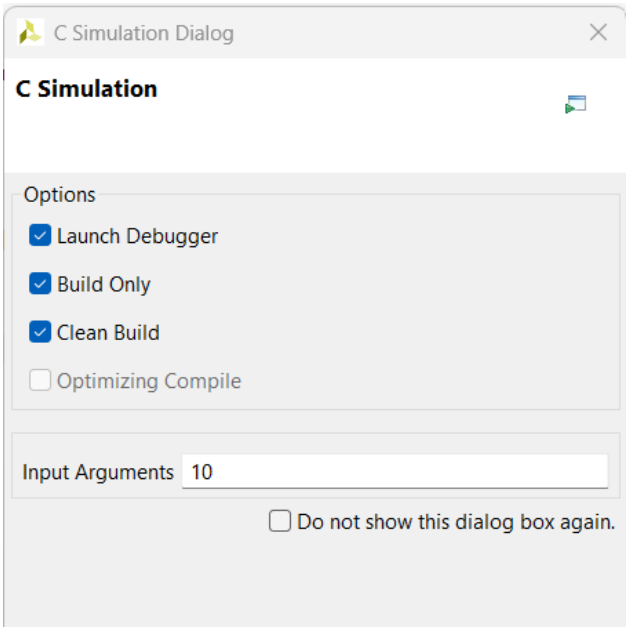
```c
#include <stdio.h>

void check_number(int* a) {
    if (a > 0)
        printf("Input: %d - The number is positive.\n", a);
    else if (a < 0)
        printf("Input: %d - The number is negative.\n", a);
    else
        printf("Input: %d - The number is zero.\n", a);
}
```

Testbench File:-

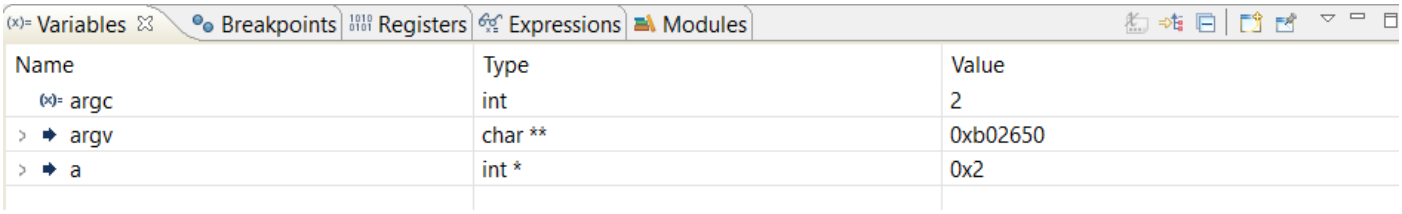
checknum.c | test_num.c | check_number_cs | __tmainCRTStart | check_number.v

```c
#include <stdio.h>
#include <stdlib.h>

// HLS function declaration
void check_number(int* a);

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s <int a>\n", argv[0]);
        return 1;
    }

    int* a = atoi(argv[1]);

    check_number(a);

    return 0;
}
```

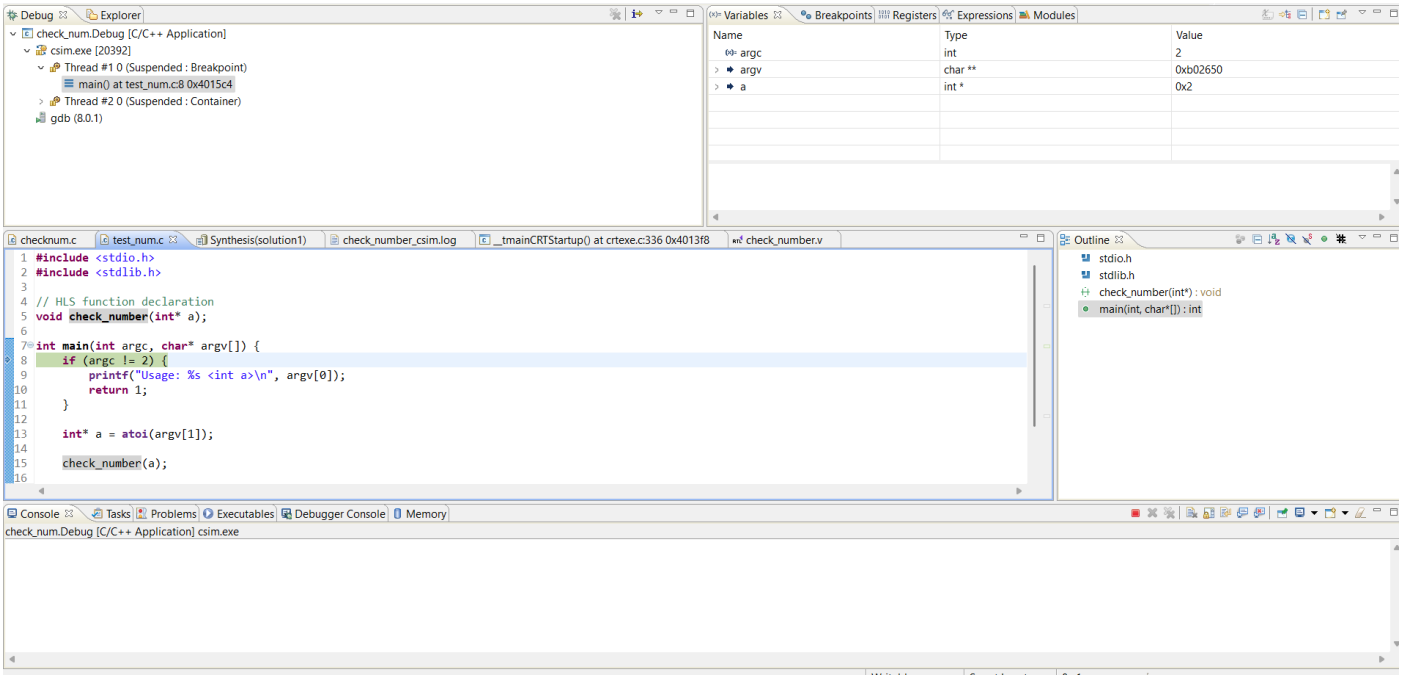
C Simulation Dialog:



Variables(Before):-



Before Debug:-

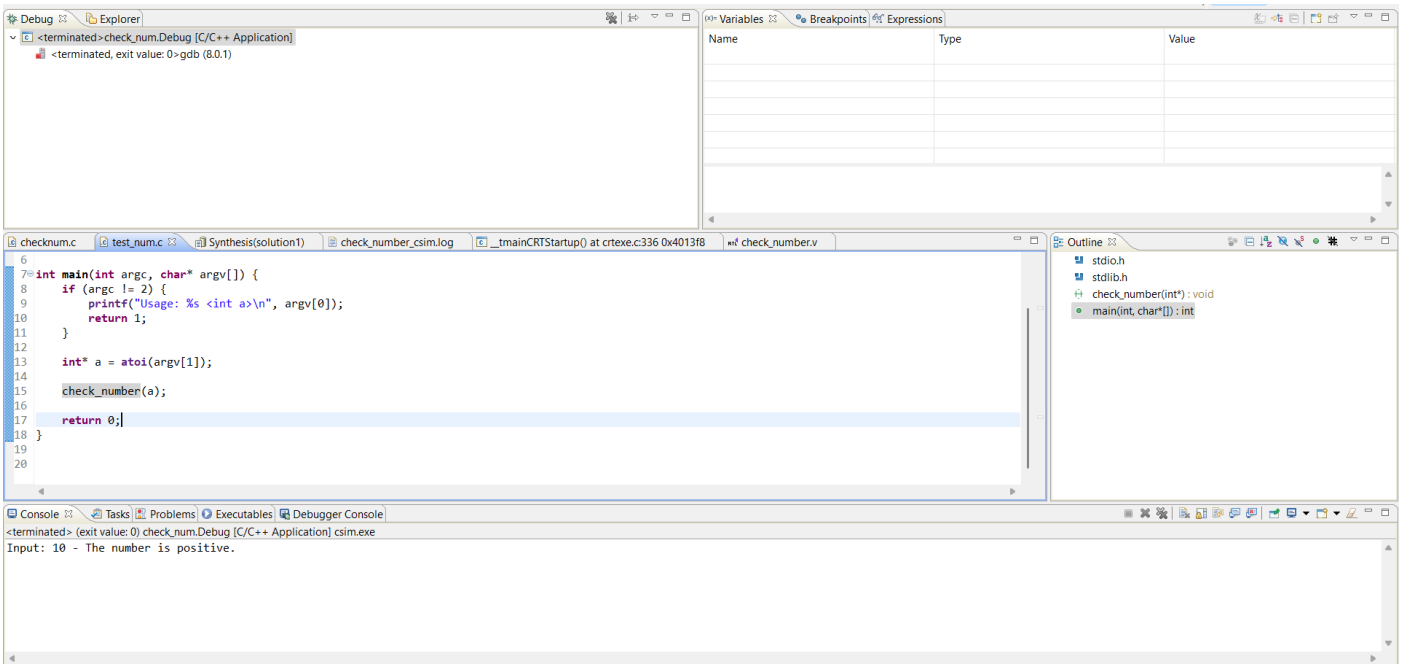


Variables while Debug:-

Variables | Breakpoints | Registers | Expressions | Modules

Name	Type	Value
argc	int	2
argv	char **	0x772650
a	int *	0xa

Outputs:-



Synthesization Report:

Synthesis Report for 'check_number'

General Information

Date:	Tue Jun 3 16:02:08 2025
Version:	2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project:	check_num
Solution:	solution1
Product family:	aartix7
Target device:	xa7a12tcsg325-1q

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	0.000	1.25

Latency (clock cycles)

Summary

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

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	0
Available	40	40	16000	8000
Utilization (%)	0	0	0	0

Detail

Instance

DSP48

Memory

FIFO

Expression

Multiplexer

Register

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_start	in	1	ap_ctrl_hs	check_number	return value
ap_done	out	1	ap_ctrl_hs	check_number	return value
ap_idle	out	1	ap_ctrl_hs	check_number	return value
ap_ready	out	1	ap_ctrl_hs	check_number	return value
a	in	32	ap_none	a	pointer

Export the report(.html) using the Export Wizard

Open Analysis Perspective Analysis Perspective

Summary on Possible user defined inputs allowed:-

Method	Synthesizable	Use Case
Function arguments	Yes	Main input method for HLS
Arrays as arguments	Yes	Multi-element data (e.g., vectors)
scanf() / cin	No	Testbench only
File handling	No	Testbench only
AXI/Streaming interfaces	Yes	Integration with IP cores
Command-line arguments	No	Testbench use only

Date:- 4-6-2025

To Design and implement a hardware-compatible C function using Vivado High-Level Synthesis (HLS) that classifies integers as **positive, negative, or zero** based on file handling concept:-

1. Read the inputs from the file named "input_file.txt" and display the output in the console.

Source file:-

checknumber.c | testbench_num.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8

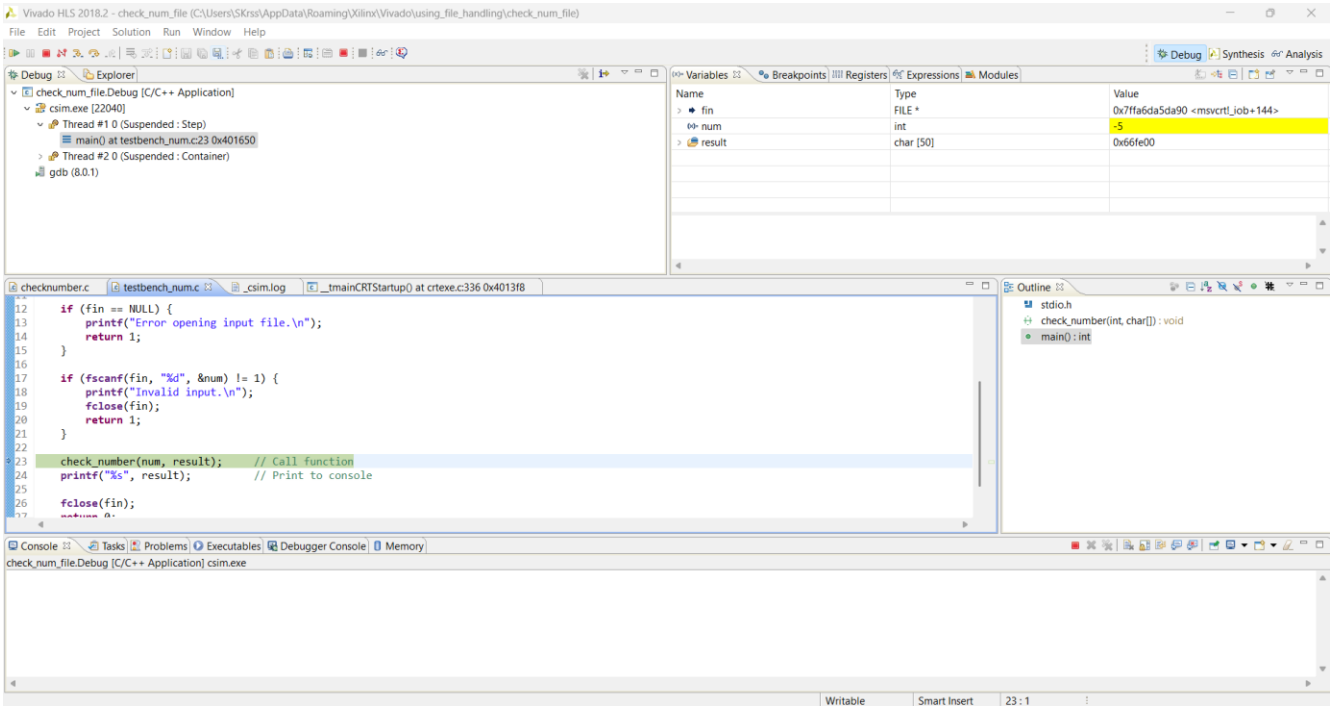
```c
#include <stdio.h>

void check_number(int a, char result[50]){
    if(a>0)
        sprintf(result, "The Number %d is positive.\n", a);
    else if(a<0)
        sprintf(result, "The Number %d is negative.\n", a);
    else
        sprintf(result, "The Number %d is zero.\n", a);
}
```

Testbench file:-

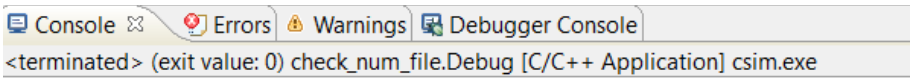
checknumber.c | testbench_num.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8

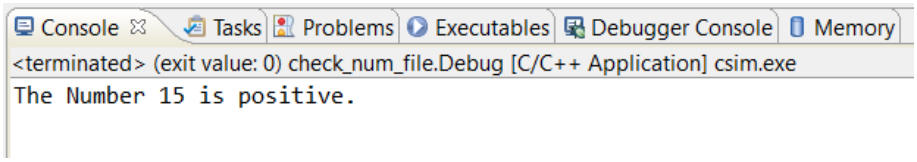
```c
#include <stdio.h>

// Declare the function
void check_number(int a, char result[50]);

int main() {
    FILE *fin = fopen("input_file.txt", "r");

    int num;
    char result[50] = {0};  // Initialize buffer to avoid garbage

    if (fin == NULL) {
        printf("Error opening input file.\n");
        return 1;
    }

    if (fscanf(fin, "%d", &num) != 1) {
        printf("Invalid input.\n");
        fclose(fin);
        return 1;
    }

    check_number(num, result);     // Call function
    printf("%s", result);          // Print to console

    fclose(fin);
    return 0;
}
```

Debugger Window:-



Outputs:-

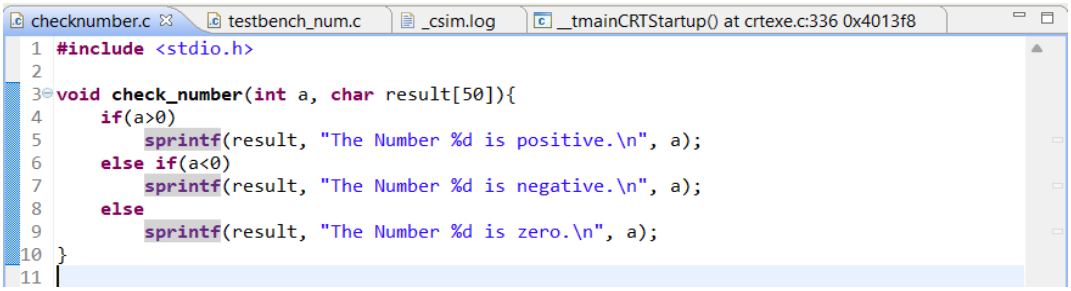




Input.txt file:-

2. Read the inputs from the file named "input_file.txt" and display the output in the console and “output_file.txt”:

Source File:-



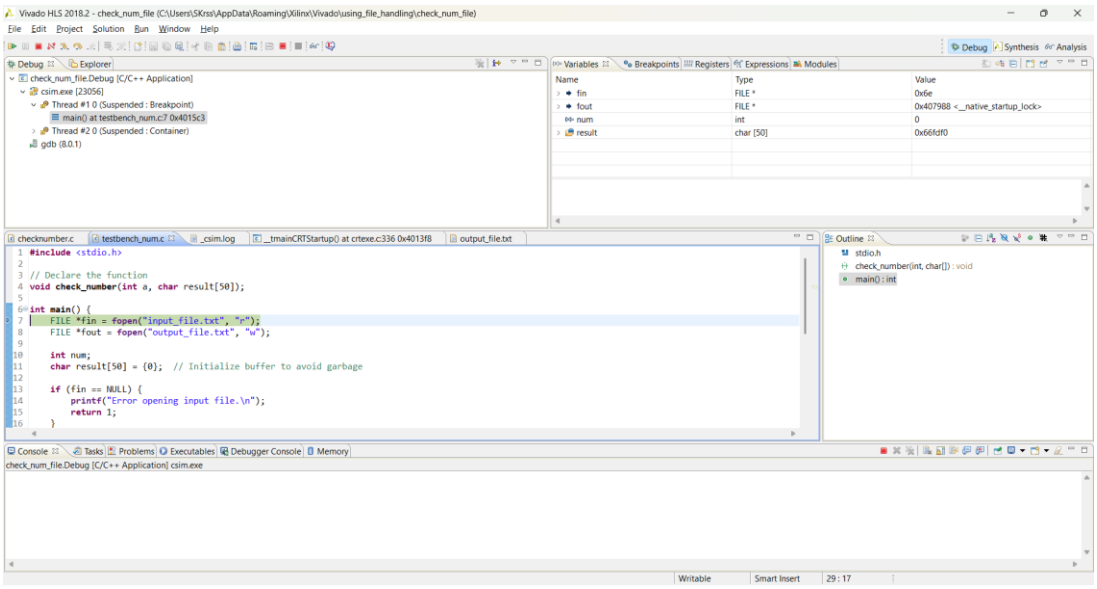
Testbench File:-

checknumber.c | testbench_num.c | _csim.log | __tmainCRTStartup() at crte | output_file.txt

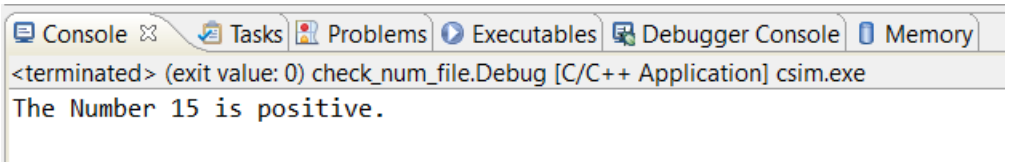
```c
#include <stdio.h>

// Declare the function
void check_number(int a, char result[50]);

int main() {
    FILE *fin = fopen("input_file.txt", "r");
    FILE *fout = fopen("output_file.txt", "w");

    int num;
    char result[50] = {0};  // Initialize buffer to avoid garbage

    if (fin == NULL) {
        printf("Error opening input file.\n");
        return 1;
    }

    if (fscanf(fin, "%d", &num) != 1) {
        printf("Invalid input.\n");
        fclose(fin);
        fclose(fout);
        return 1;
    }

    check_number(num, result);     // Call function
    printf("%s", result);          // Print to console
    fprintf(fout, "%s", result);    //Write to output file

    fclose(fin);
    fclose(fout);
    return 0;
}

```

Input.txt file:

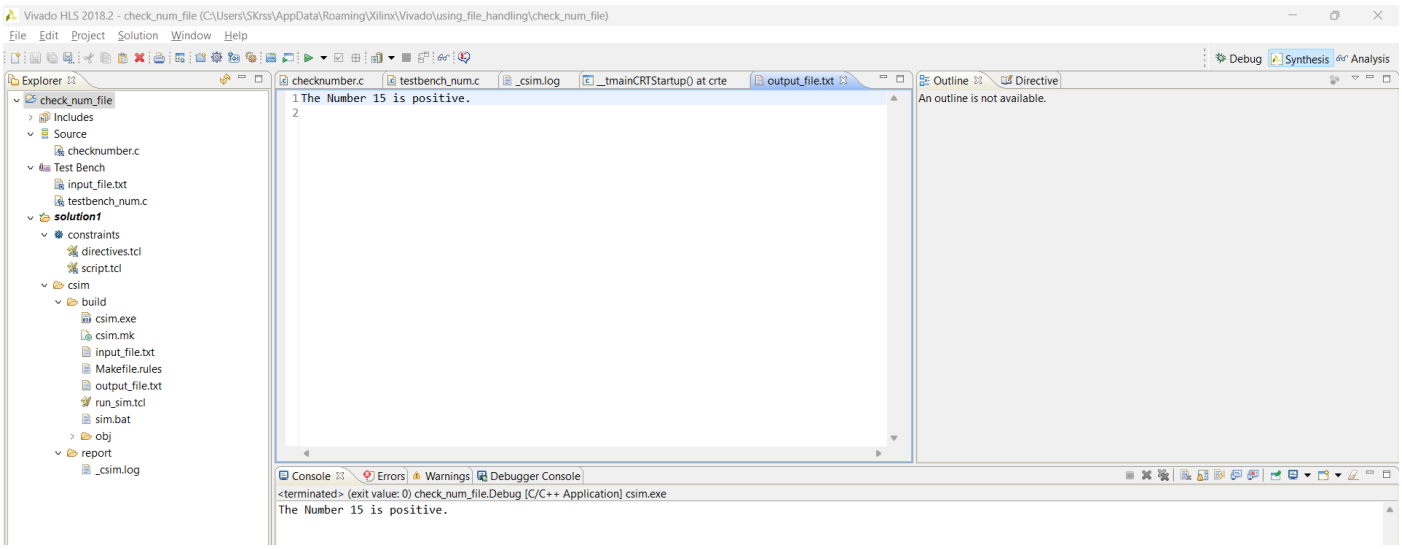
Debugger Window:-



Outputs(in console):-



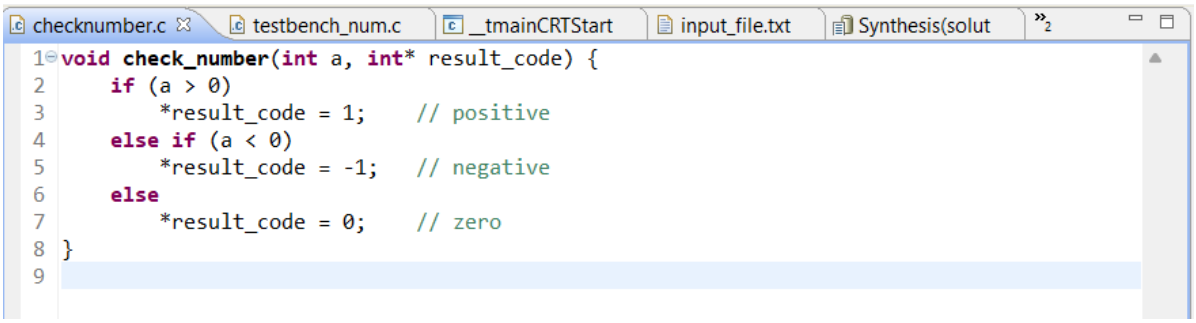
Output_file.txt :-



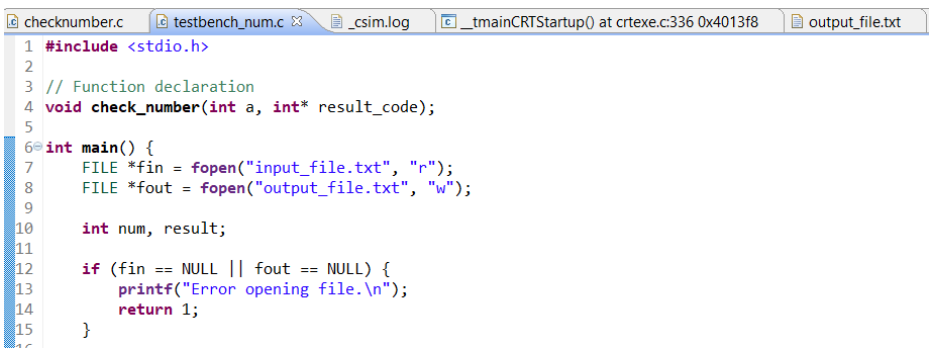
While running synthesis, the console displays that the 'sprintf' function is unsupported in this software.

So the modified source and testbench files are:-

Source File:-



Testbench File:-



checknumber.c | testbench_num.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8 | output_file.txt

```c
#include <stdio.h>

// Function declaration
void check_number(int a, int* result_code);

int main() {
    FILE *fin = fopen("input_file.txt", "r");
    FILE *fout = fopen("output_file.txt", "w");

    int num, result;

    if (fin == NULL || fout == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    // Read input number
    if (fscanf(fin, "%d", &num) != 1) {
        printf("Invalid input.\n");
        fclose(fin);
        fclose(fout);
        return 1;
    }

    // Call HLS-compatible function
    check_number(num, &result);

    // Determine message
    if (result == 1) {
        printf("The Number %d is positive.\n", num);
        fprintf(fout, "The Number %d is positive.\n", num);
    } else if (result == -1) {
        printf("The Number %d is negative.\n", num);
        fprintf(fout, "The Number %d is negative.\n", num);
    } else {
        printf("The Number %d is zero.\n", num);
        fprintf(fout, "The Number %d is zero.\n", num);
    }

    fclose(fin);
    fclose(fout);
    return 0;
}
```

Input_file.txt :-

checknumber.c | testbench_num.c | __tmainCRTStart | input_file.txt | Synthesis(solut

```
50
```

Outputs:-

Console | Errors | Warnings | DRCs | Debugger Console

```
<terminated> (exit value: 0) check_num_file.Debug [C/C++ Application] csim.exe
The Number 50 is positive.
```

Output_file.txt:-

checknumber.c | testbench_num.c | __tmainCRTStart | output_file.txt | input_file.txt

```
The Number 50 is positive.
```

Synthesis Reports:-

Synthesis Report for 'check_number'

General Information

Date:	Wed Jun 4 14:15:16 2025
Version:	2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project:	check_num_file
Solution:	solution1
Product family:	artix7
Target device:	xc7a35tcpg236-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	3.431	1.25

Latency (clock cycles)

Summary

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

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	24
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	24
Available	100	90	41600	20800
Utilization (%)	0	0	0	~0

Verilog Code:-

checknumber.c | testbench_num.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8 | output_file.txt | input_file.txt | Synthesis(solution1) | check_number.v

```
// ==============================================================
// RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2018.2
// Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
//
// ==============================================================

`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="check_number,hls_ip_2018_2,{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_CLOC

module check_number (
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        a,
        result_code,
        result_code_ap_vld
);


input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [31:0] a;
output  [31:0] result_code;
output   result_code_ap_vld;

reg result_code_ap_vld;

wire   [0:0] tmp_fu_39_p2;
wire   [0:0] tmp_1_fu_45_p3;
wire   [0:0] tmp_3_fu_61_p2;
wire   [1:0] a_lobit_fu_53_p3;
wire   [1:0] p_a_lobit_fu_67_p3;

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

assign a_lobit_fu_53_p3 = ((tmp_fu_39_p2[0:0] === 1'b1) ? 2'd1 : 2'd3);

assign ap_done = ap_start;

assign ap_idle = 1'b1;

assign ap_ready = ap_start;

assign p_a_lobit_fu_67_p3 = ((tmp_3_fu_61_p2[0:0] === 1'b1) ? a_lobit_fu_53_p3 : 2'd0);

assign result_code = $signed(p_a_lobit_fu_67_p3);

assign tmp_1_fu_45_p3 = a[32'd31];

assign tmp_3_fu_61_p2 = (tmp_fu_39_p2 | tmp_1_fu_45_p3);

assign tmp_fu_39_p2 = (($signed(a) > $signed(32'd0)) ? 1'b1 : 1'b0);

endmodule //check_number

```

To enhance the experience with Xilinx Vivado, leverage Verilog code to effectively execute simulation, synthesis, and more. This approach will unlock valuable insights and streamline design process for the best results.

Design File (checknumber.v):

checknumber.v | checknumber_tb.v | Untitled 6

C:/Users/SKrss/checknum.srcs/sources_1/new/checknumber.v

```
22
23  module check_number (
24          ap_start,
25          ap_done,
26          ap_idle,
27          ap_ready,
28          a,
29          result_code,
30          result_code_ap_vld
31  );
32
33
34  input   ap_start;
35  output   ap_done;
36  output   ap_idle;
37  output   ap_ready;
38  input  [31:0] a;
39  output  [31:0] result_code;
40  output   result_code_ap_vld;
41
42  reg result_code_ap_vld;
43
44  wire   [0:0] tmp_fu_39_p2;
45  wire   [0:0] tmp_1_fu_45_p3;
46  wire   [0:0] tmp_3_fu_61_p2;
47  wire   [1:0] a_lobit_fu_53_p3;
48  wire   [1:0] p_a_lobit_fu_67_p3;
49
50  always @ (*) begin
51      if ((ap_start == 1'b1)) begin
52          result_code_ap_vld = 1'b1;
53      end else begin
54          result_code_ap_vld = 1'b0;
55      end
56  end
57
58  assign a_lobit_fu_53_p3 = ((tmp_fu_39_p2[0:0] === 1'b1) ? 2'd1 : 2'd3);
59
60  assign ap_done = ap_start;
61
62  assign ap_idle = 1'b1;
63
64  assign ap_ready = ap_start;
65
66  assign p_a_lobit_fu_67_p3 = ((tmp_3_fu_61_p2[0:0] === 1'b1) ? a_lobit_fu_53_p3 : 2'd0);
67
68  assign result_code = $signed(p_a_lobit_fu_67_p3);
69
70  assign tmp_1_fu_45_p3 = a[32'd31];
71
72  assign tmp_3_fu_61_p2 = (tmp_fu_39_p2 | tmp_1_fu_45_p3);
73
74  assign tmp_fu_39_p2 = (($signed(a) > $signed(32'd0)) ? 1'b1 : 1'b0);
75
76  endmodule //check_number
```

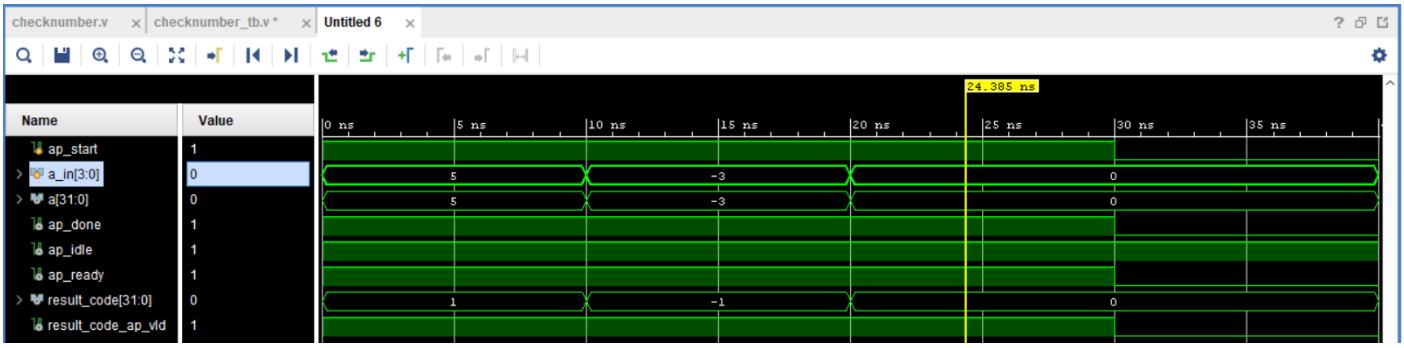
Testbench File (checknumber_tb.v):

checknumber.v | checknumber_tb.v | Untitled 6

C:/Users/SKrss/checknum.srcs/sim_1/new/checknumber_tb.v

```
22  module tb_check_number;
23
24    // Testbench 4-bit input
25    reg ap_start;
26    reg [3:0] a_in;
27
28    // Extended 32-bit signal to connect to DUT
29    wire [31:0] a = {{28{a_in[3]}}, a_in};  // Sign-extend to 32-bit
30
31    // DUT outputs
32    wire ap_done;
33    wire ap_idle;
34    wire ap_ready;
35    wire [31:0] result_code;
36    wire result_code_ap_vld;
37
38    // DUT instantiation
39    check_number uut (
40      .ap_start(ap_start),
41      .ap_done(ap_done),
42      .ap_idle(ap_idle),
43      .ap_ready(ap_ready),
44      .a(a),
45      .result_code(result_code),
46      .result_code_ap_vld(result_code_ap_vld)
47    );
48
49    initial begin
50      // Enable waveform dump
51      $dumpfile("wave.vcd");
52      $dumpvars(0, tb_check_number);
53
54      // Print header
55      $display("Time\tap_start\ta_in\tsigned(a)\tresult_code\tresult_code_ap_vld");
56      $monitor("%0t\t%b\t\t%b\t%0d\t\t%0d\t\t%b",
57               $time, ap_start, a_in, $signed(a), result_code, result_code_ap_vld);
58
59      // Start simulation
60      ap_start = 1;
61
62      // Test 1: Positive (5)
63      a_in = 4'b0101; #10;
64
65      // Test 2: Negative (-3 in 4-bit = 1101)
66      a_in = 4'b1101; #10;
67
68      // Test 3: Zero
69      a_in = 4'b0000; #10;
70
71      // Stop
72      ap_start = 0; #10;
73      $finish;
74    end
75
76  endmodule
77
```

Simulation Graph:-



Observations:-

- At the start of the simulation, ap_start is high, which begins the operation of the module.
- When the input a_in = 5, the output result_code = 1, showing the module detects a positive number.
- When the input changes to a_in = -3, the output becomes -1, meaning the module identifies a negative number.
- When the input changes to a_in = 0, the output also becomes 0, confirming the module detects zero correctly.
- The control signals ap_done, ap_idle, ap_ready, and result_code_ap_vld change in step with the inputs, ensuring the output is valid only after processing.
- Overall, the module works as expected by identifying positive, negative, and zero inputs while keeping proper synchronization with the control signals.

Limitations of This Code (for Vivado HLS)

Limitation	Explanation
sprintf()	Not synthesizable in Vivado HLS — dynamic formatting like sprintf can't be mapped to hardware logic.
FILE*, fopen, fscanf, fprintf	File I/O operations are not supported in hardware synthesis — can be used only in testbench for C simulation.
Dynamic string manipulation (char result[50])	HLS synthesis does not support dynamic string handling or heap-based memory access.
Console printing (printf)	Can be used for **C simulation only**, but is **ignored during synthesis**.

Explaination

This C program is designed to check whether a given number is positive, negative, or zero, and then output the result both to the console and to a file. The core logic is implemented in the **check_number() function**, which takes an integer input and uses **sprintf()** to store a corresponding message in a character buffer. The **main()** function acts as a testbench — it reads an integer from an input file **(input_file.txt),** calls the **check_number()** function, and then prints the resulting message to the console and writes it into an output file **(output_file.txt).** The file handling is done using standard C functions like **fopen(), fscanf(), fprintf(), and fclose().**

However, this code has important limitations if used in a hardware synthesis context, such as with Vivado HLS. Functions like sprintf(), printf(), and all file operations are not synthesizable, meaning they cannot be implemented in hardware. These are only suitable for simulation and debugging purposes. Additionally, dynamic string manipulation (like using character buffers for messages) is not supported in hardware logic synthesis. To make this code synthesizable, the logic should return a simple status code (e.g., 1 for positive, 0 for zero, -1 for negative), and all message formatting and file operations should be done only in the testbench during simulation.

In Verilog, a hardware description language (HDL), the code is written to describe the behavior or structure of digital circuits. A Verilog module typically includes inputs, outputs, and logic to perform a specific task, such as arithmetic or control operations. When you write a Verilog module in Xilinx Vivado (not Vivado HLS), you are directly describing how the circuit should behave at the Register Transfer Level (RTL). The process in Vivado begins with writing the Verilog source files and a corresponding testbench that stimulates the design with specific input patterns and checks the outputs. The testbench is used only for simulation and is not synthesized.

Once the Verilog design is written, you use Vivado to run **RTL simulation** to verify functionality. If the simulation is correct, you proceed to **synthesis**, where Vivado translates the RTL code into a gate-level netlist, representing how the logic will be implemented in the actual hardware (FPGA fabric). After synthesis, the **implementation** step maps the design to the resources of the target FPGA, performs placement and routing, and checks timing. Finally, Vivado generates a **bitstream file (.bit)**, which can be used to program the FPGA.

Unlike Vivado HLS (which translates C/C++ into HDL), Verilog code in standard Vivado is already in the correct abstraction level for direct hardware synthesis. It doesn't support C-like features such as printf or dynamic memory, and the logic is implemented using always blocks, continuous assignments, and instantiations of submodules. The flow is purely HDL-based, making it more predictable and efficient for low-level hardware control.

Date:- 5-6-2025

To check whether datatypes(int, float, char, double) are mapping in vivado:-

Ex:- Program to find subtraction of two numbers using HLS and check in vivado.

❖ **In Vivado HLS:- (using 'int' as datatype)**

Source Code:-

subtract.c | subtract_tb.c | _csim.log | __tmainCRTStartup() a | Synthesis(solution1) | subtract.v

```c
#include <stdio.h>

// Vivado HLS-compatible top-level function
int subtract(int a, int b) {
    int result = a - b;
    return result;
}

```

Testbench Code:-

subtract.c | subtract_tb.c | _csim.log | __tmainCRTStartup() a | Synthesis(solution1) | subtract.v

```c
#include <stdio.h>

// Declare the subtract function
int subtract(int a, int b);

int main() {
    int a = 15;
    int b = 7;
    int result;

    result = subtract(a, b);

    printf("Subtraction of %d - %d = %d\n", a, b, result);

    return 0;
}

```

Output Console:-

Console | Tasks | Problems | Executables

```
<terminated> (exit value: 0) Subtract_operation.Debug [C/C++ Application] csim.exe
Subtraction of 15 - 7 = 8
```

Synthesis Report:-

Synthesis Report for 'subtract'

General Information

Date:	Thu Jun 5 13:01:33 2025
Version:	2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project:	Subtract_operation
Solution:	solution1
Product family:	artix7
Target device:	xc7a35tcpg236-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.702	1.25

Latency (clock cycles)

Summary

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

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

Verilog Code:-

subtract.c | subtract_tb.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8 | Synthesis(solution1) | subtract.v

```
// ==============================================================
// RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2018.2
// Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
//
// ==============================================================

`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="subtract,hls_ip_2018_2,{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.

module subtract (
        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 = (a - b);

endmodule //subtract

```

❖ In Xilinx Vivado:-

Design File:-

subtract.v | subtract_tb.v | Untitled 9*

C:/Users/SKrss/check_subtract_2.srcs/sources_1/new/subtract.v

```
//////////////////////////////////////////////////////////////////////////////////


module subtract (
        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 = (a - b);

endmodule //subtract
```

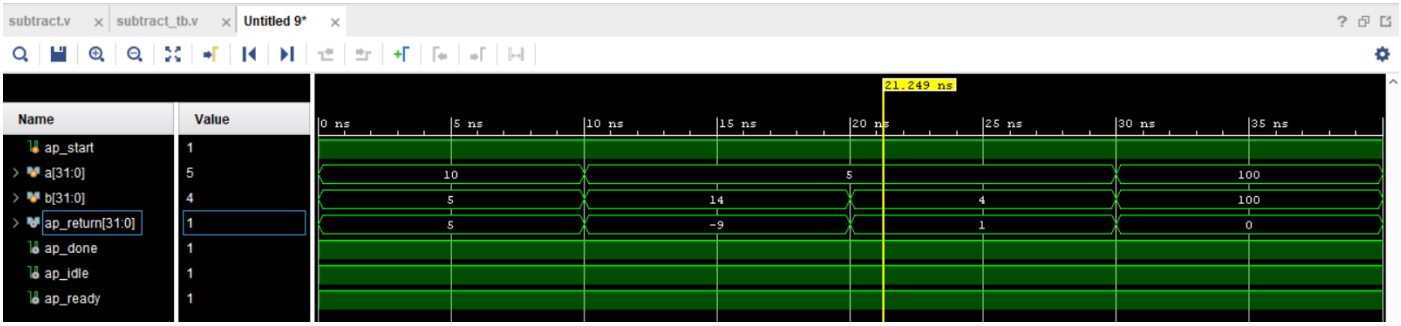
Testbench File:-

subtract.v | subtract_tb.v | Untitled 9*

C:/Users/SKrss/check_subtract_2.srcs/sim_1/new/subtract_tb.v

```
module subtract_tb;

    // Inputs
    reg ap_start;
    reg [31:0] a;
    reg [31:0] b;

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

    // Instantiate the subtract module
    subtract uut (
        .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...");

        // Test case 1: 10 - 5
        a = 10;
        b = 5;
        ap_start = 1;
```

Simulation Graph:-



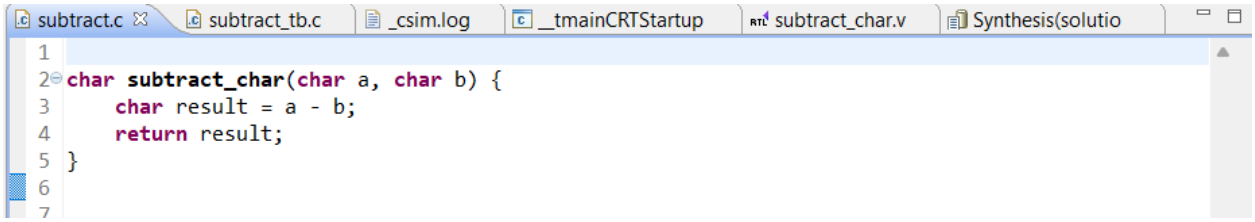
Observations:-

- The signal ap_start stays high during the simulation, so the subtractor is always active and ready to work.
- The input signals a[31:0] and b[31:0] change at different times, and the output ap_return updates right after each change.
- In the first case, when a = 10 and b = 5, the result is 5, which shows subtraction works correctly when a is greater than b.
- In the second case, when a = 5 and b = 14, the result is -9, which proves the design can handle negative results.
- In the third case, when a = 4 and b = 1, the result is 3, showing another correct positive subtraction.
- In the fourth case, when a = 100 and b = 100, the result is 0, confirming that the design handles equal numbers properly.
- The output always matches the expected result immediately after the inputs change, which means the subtractor behaves like a combinational circuit.
- The control signals ap_done, ap_idle, and ap_ready remain high, showing that the module is always ready to take new inputs and complete operations without delay.
- The testbench checks different cases such as greater input, smaller input, and equal inputs to fully test the subtraction logic.
- Overall, the simulation proves that the subtractor works correctly for positive, negative, and zero results while keeping the control signals in sync.

2.Using ‘char’ as datatype:-

❖ In Vivado HLS:-

Source File:-



Testbench File:-

subtract.c | subtract_tb.c | _csim.log | __tmainCRTStartup | subtract_char.v | Synthesis(solutio

```c

#include <stdio.h>

char subtract_char(char a, char b);

int main() {
    char a = 25;
    char b = 10;
    char result;

    result = subtract_char(a, b);

    printf("Subtraction of %d - %d = %d\n", a, b, result);
    return 0;
}

```

Output Console:-

Console | Tasks | Problems | Executables | Debugger Console

```
<terminated> (exit value: 0) Subtract_operation.Debug [C/C++ Application] csim.exe
Subtraction of 25 - 10 = 15
```

Verilog Code:-

subtract.c | subtract_tb.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8 | subtract_char.v | Synthesis(solution1)

```
// === Subtract_operation/source/subtract.c ==============================
// RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2018.2
// Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
//
// ==========================================================

`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="subtract_char,hls_ip_2018_2,{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_CLO

module subtract_char (
        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 = (a - b);

endmodule //subtract_char

```

Synthesis Report:-

Synthesis Report for 'subtract_char'

General Information

Date:	Thu Jun 5 14:39:55 2025
Version:	2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project:	Subtract_operation
Solution:	solution1
Product family:	artix7
Target device:	xc7a35tcpg236-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.115	1.25

Latency (clock cycles)

Summary

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

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	15
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	15
Available	100	90	41600	20800
Utilization (%)	0	0	0	~0

❖ In Xilinx Vivado:-

Design Code:-

char_subtract.v | char_subtract.v (2) | Untitled 10

C:/Users/SKrss/subtract_char.srcs/sources_1/new/char_subtract.v

```
20  //////////////////////////////////////////////////////////////////////////////////
21
22
23  module subtract_char (
24          ap_start,
25          ap_done,
26          ap_idle,
27          ap_ready,
28          a,
29          b,
30          ap_return
31  );
32
33
34  input   ap_start;
35  output   ap_done;
36  output   ap_idle;
37  output   ap_ready;
38  input  [7:0] a;
39  input  [7:0] b;
40  output  [7:0] ap_return;
41
42  assign ap_done = ap_start;
43
44  assign ap_idle = 1'b1;
45
46  assign ap_ready = ap_start;
47
48  assign ap_return = (a - b);
49
50  endmodule //subtract_char
```

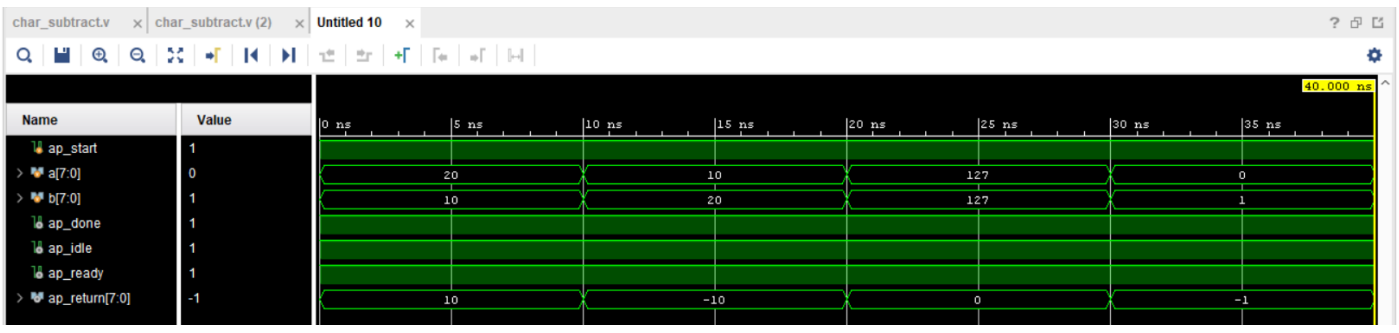
Testbench File:-

char_subtract.v | char_subtract.v (2) | Untitled 10

C:/Users/SKrss/subtract_char.srcs/sim_1/new/char_subtract.v

```
23  `timescale 1ns / 1ps
24
25  module subtract_char_tb;
26
27      // Inputs
28      reg ap_start;
29      reg [7:0] a;
30      reg [7:0] b;
31
32      // Outputs
33      wire ap_done;
34      wire ap_idle;
35      wire ap_ready;
36      wire [7:0] ap_return;
37
38      // Instantiate the subtract_char module
39      subtract_char uut (
40          .ap_start(ap_start),
41          .ap_done(ap_done),
42          .ap_idle(ap_idle),
43          .ap_ready(ap_ready),
44          .a(a),
45          .b(b),
46          .ap_return(ap_return)
47      );
48
49      initial begin
50          $display("Testbench for subtract_char module");
52          // Test case 1: 20 - 10 = 10
53          a = 8'd20;
54          b = 8'd10;
55          ap_start = 1;
56          #10;
57          $display("Test 1: a = %0d, b = %0d, ap_return = %0d", a, b, ap_return);
58
59          // Test case 2: 10 - 20 = -10 (wraps around to 8-bit)
60          a = 8'd10;
61          b = 8'd20;
62          ap_start = 1;
63          #10;
64          $display("Test 2: a = %0d, b = %0d, ap_return = %0d (interpreted as signed: %0d)", a, b, ap_return, $signed(ap_return));
65
66          // Test case 3: 127 - 127 = 0
67          a = 8'd127;
68          b = 8'd127;
69          ap_start = 1;
70          #10;
71          $display("Test 3: a = %0d, b = %0d, ap_return = %0d", a, b, ap_return);
72
73          // Test case 4: 0 - 1 = -1 (wraps to 255)
74          a = 8'd0;
75          b = 8'd1;
76          ap_start = 1;
77          #10;
78          $display("Test 4: a = %0d, b = %0d, ap_return = %0d (signed: %0d)", a, b, ap_return, $signed(ap_return));
79
80          $display("Testbench completed.");
81          $finish;
82      end
```

Simulation Graph:-



Observations:-

- The signal ap_start is kept high throughout the simulation, which enables the subtractor module to continuously perform operations on the given inputs.
- When the input values are a = 20 and b = 10, the output ap_return is 10, which is correct since 20 – 10 = 10.
- When the inputs change to a = 10 and b = 20, the output becomes -10, which shows that the subtractor correctly handles negative results using signed 8-bit representation.
- For the case a = 127 and b = 127, the output is 0, which confirms that the design produces zero when both operands are equal.
- When the inputs are a = 0 and b = 1, the output is -1, which validates the correct handling of subtraction that results in a negative single value.
- The results are updated immediately after the inputs change, which indicates that the subtractor behaves like a combinational circuit with no extra delay.
- The control signals ap_done, ap_idle, and ap_ready remain asserted, confirming that the module is always ready to accept inputs and provide results without stalling.
- The simulation demonstrates that the char subtractor correctly performs subtraction for different conditions, including positive difference, negative difference, zero result, and boundary cases for signed 8-bit values.

Date:- 6-6-2025

3.Using ‘float’ as Datatype:-

Source code:-

datatypes1.cpp | datatypes1_tb.c | datatype

```cpp
#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;
}

```

Testbench Code:-

datatypes1.cpp | *datatypes1_tb. | datatypes1_csim | Synthesis(solut

```cpp
#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 Source Code and Testbench Code:-

datatypes1.v * | datatypes1_tb.v | Untitled 3

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

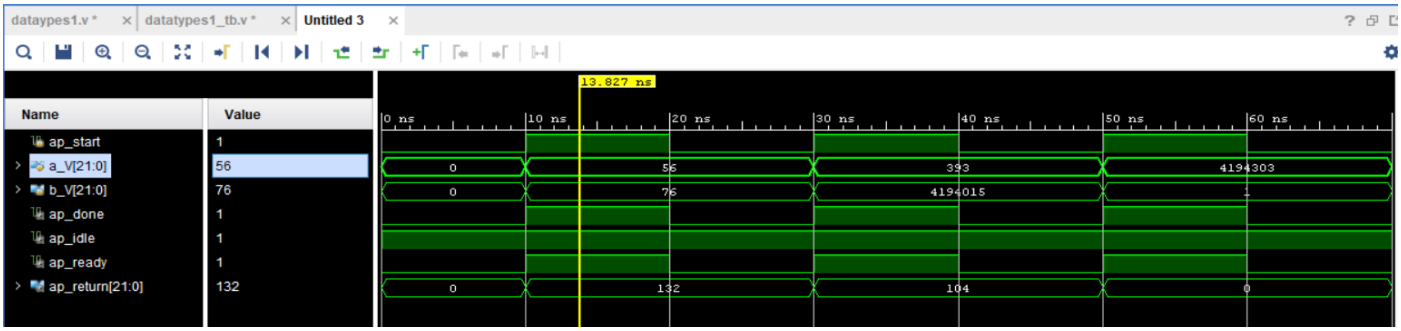
```
`timescale 1 ns / 1 ps

(* 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
```

datatypes1_tb.v *

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 Q8.17
    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
```

Simulation Graph:-



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.

Date:- 9-6-2025

Design and Implementation of Synchronous D-Flipflop using vivado HLS and Xilinx vivado:

❖ In Vivado HLS:

Source Code:-

d_flipflop.c | d_flipflop_tb.c | __tmainCRTStart | Synthesis(solut | d_flip_flop.v

```c

void d_flip_flop(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;
}

```

Testbench Code:-

d_flipflop.c | d_flipflop_tb.c | _csim.log | __tmainCRTStartup() at crtexe.c:336 0x4013f8

```c
#include <stdio.h>

// Function prototype
void d_flip_flop(int clk, int rst, int d, int *q);

int main() {
    int clk = 0;
    int rst = 1;
    int d = 0;
    int q = 0;

    // Apply reset
    d_flip_flop(clk, rst, d, &q);
    printf("After reset: Q = %d\n", q);

    // Release reset
    rst = 0;

    // First rising edge with d = 1
    d = 1;
    clk = 1;
    d_flip_flop(clk, rst, d, &q);
    printf("Rising edge 1 (D=1): Q = %d\n", q);

    // Falling edge (no capture)
    clk = 0;
    d = 0;
    d_flip_flop(clk, rst, d, &q);
    printf("Falling edge (D=0): Q = %d\n", q);

    // Second rising edge with d = 0
    clk = 1;
    d = 0;
    d_flip_flop(clk, rst, d, &q);
    printf("Rising edge 2 (D=0): Q = %d\n", q);

    return 0;
}

```

Output Console:-

Console | Tasks | Problems | Executables | Debugger Console | Memory

<terminated> (exit value: 0) D_flipflop_using_clk.Debug [C/C++ Application] csim.exe

```
After reset: Q = 0
Rising edge 1 (D=1): Q = 1
Falling edge (D=0): Q = 1
Rising edge 2 (D=0): Q = 0
```

Synthesis Report:-

Synthesis Report for 'd_flip_flop'

General Information

Date:	Mon Jun 9 10:18:59 2025
Version:	2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project:	D_flipflop_using_clk
Solution:	solution1
Product family:	artix7
Target device:	xc7a35tcpg236-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.438	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	1	1	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	36
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	24
Register	-	-	34	-
Total	0	0	34	60
Available	100	90	41600	20800
Utilization (%)	0	0	~0	~0

❖ In Xilinx Vivado:-

Design File:-

Package | Device | d_ff.v | d_ff_tb.v

C:/Users/SKrss/d_flipflop/D_flipflop.srcs/sources_1/new/d_ff.v

```
22
23  module d_flip_flop (
24          ap_clk,
25          ap_rst,
26          ap_start,
27          ap_done,
28          ap_idle,
29          ap_ready,
30          clk,
31          rst,
32          d,
33          q,
34          q_ap_vld
35  );

37  parameter    ap_ST_fsm_state1 = 2'd1;
38  parameter    ap_ST_fsm_state2 = 2'd2;
39
40  input   ap_clk;
41  input   ap_rst;
42  input   ap_start;
43  output   ap_done;
44  output   ap_idle;
45  output   ap_ready;
46  input   clk;
47  input  rst;
48  input  [7:0] d;
49  output  [7:0] q;
50  output   q_ap_vld;
51
```

```
52  reg ap_done;
53  reg ap_idle;
54  reg ap_ready;
55  reg q_ap_vld;
56
57  (* fsm_encoding = "none" *) reg   [1:0] ap_CS_fsm;
58  wire    ap_CS_fsm_state1;
59  reg   [7:0] q_reg;
60  wire   [0:0] tmp_fu_47_p2;
61  wire   [0:0] tmp_1_fu_59_p2;
62  wire    ap_CS_fsm_state2;
63  reg   [1:0] ap_NS_fsm;
64
65  // power-on initialization
66  initial begin
67  #0 ap_CS_fsm = 2'd1;
68  #0 q_reg = 8'd0;
69  end
70
71  always @ (posedge ap_clk) begin
72      if (ap_rst == 1'b1) begin
73          ap_CS_fsm <= ap_ST_fsm_state1;
74      end else begin
75          ap_CS_fsm <= ap_NS_fsm;
76      end
77  end
78
79  always @ (posedge ap_clk) begin
80      if (((ap_start == 1'b1) & (1'b1 == ap_CS_fsm_state1))) begin
81          if (((tmp_1_fu_59_p2 == 1'd0) & (tmp_fu_47_p2 == 1'd1))) begin
82              q_reg <= d;
```

```
82              q_reg <= d;
83          end else if ((tmp_fu_47_p2 == 1'd0)) begin
84              q_reg <= 8'd0;
85          end
86      end
87  end
88
89  always @ (*) begin
90      if ((1'b1 == ap_CS_fsm_state2)) begin
91          ap_done = 1'b1;
92      end else begin
93          ap_done = 1'b0;
94      end
95  end
96
97  always @ (*) begin
98      if (((ap_start == 1'b0) & (1'b1 == ap_CS_fsm_state1))) begin
99          ap_idle = 1'b1;
100     end else begin
101         ap_idle = 1'b0;
102     end
103 end
104
105 always @ (*) begin
106     if ((1'b1 == ap_CS_fsm_state2)) begin
107         ap_ready = 1'b1;
108     end else begin
109         ap_ready = 1'b0;
110     end
111 end
112
```

```
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[0];
assign ap_CS_fsm_state2 = ap_CS_fsm[1];
assign q = q_reg;
assign tmp_1_fu_59_p2 = ((clk == 8'd0) ? 1'b1 : 1'b0);
assign tmp_fu_47_p2   = ((rst == 8'd0) ? 1'b1 : 1'b0);


endmodule // d_flip_flop
```

Testbench File:-

Package | Device | d_ff.v | d_ff_tb.v

C:/Users/SKrss/d_flipflop/D_flipflop.srcs/sim_1/new/d_ff_tb.v

```
module tb_d_flip_flop;

    // Inputs
    reg ap_clk;
    reg ap_rst;
    reg ap_start;
    reg clk;
    reg  rst;
    reg [7:0] d;

    // Outputs
    wire ap_done;
    wire ap_idle;
    wire ap_ready;
    wire [7:0] q;
    wire q_ap_vld;

    // Instantiate the DUT
    d_flip_flop dut (
      .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)
    );
```

```
53
54      // Generate ap_clk
55      initial begin
56        ap_clk = 0;
57        forever #5 ap_clk = ~ap_clk; // 10ns period
58      end
59
60      // Task to apply a start pulse
61      task apply_start;
62        begin
63          ap_start = 1;
64          #10;
65          ap_start = 0;
66        end
67      endtask
68
69      // Test sequence
70      initial begin
71        ap_rst = 1;
72        clk = 0;
73        rst = 1;
74        d = 0;
75        ap_start = 0;
76
77        #20;
78        ap_rst = 0;
```

```
79
80      // Test Case 1: reset active
81      clk = 0; rst = 1; d = 1;
82      apply_start(); #10;
83      $display("After reset: q = %d", q);
84
85      // Test Case 2: capture D = 1
86      clk = 1; rst = 0; d = 1;
87      apply_start(); #10;
88      $display("Rising clk, D=1: q = %d", q);
89
90      // Test Case 3: falling clk
91      clk = 0; d = 0;
92      apply_start(); #10;
93      $display("Falling clk, D=0: q = %d", q);
94
95      // Test Case 4: rising clk, D = 0
96      clk = 1; d = 0;
97      apply_start(); #10;
98      $display("Rising clk, D=0: q = %d", q);
99
100     // Test Case 5: reset again
101     clk = 1; rst = 1; d = 1;
102     apply_start(); #10;
103     $display("Async reset: q = %d", q);
104
105       $finish;
106     end
107
108 endmodule
```

Simulation Graph:-

Truth Table:-

Clock Edge	Reset (rst)	Input (D)	Output (Q)	Observation
Rising Edge	1	X	0	Reset active, output forced to 0 regardless of input.
Rising Edge	0	0	0	Output follows input D = 0.
Rising Edge	0	1	1	Output follows input D = 1.
Falling Edge / No Edge	X	X	Q (no change)	No update; output holds previous value.

Observations:-

- At the beginning of the simulation, the reset signal (ap_rst / rst) is low, allowing the D flip-flop to start in its normal operating mode.
- The output signal q[7:0] depends on the input signal d[7:0], but it only updates its value at the rising edge of the clock (clk).
- When the input d is 0 at a rising clock edge, the output q also becomes 0, showing that the flip-flop stores the input correctly.
- Later, when the input d changes to 1, the output q does not update immediately but changes to 1 at the next rising edge of the clock, confirming that the circuit is edge-triggered.
- Similarly, when the input d changes back to 0, the output q changes to 0 only after the next rising clock edge, holding the previous value until that time.
- This behavior demonstrates the data storage property of the D flip-flop, where the input is sampled only on the rising edge and stored until the next clock cycle.

- If the reset (rst) signal is asserted high during operation, the output q is forced to 0, overriding the stored value, which shows the proper reset functionality of the flip-flop.
- The control signals ap_done, ap_idle, ap_ready, and q_ap_vld toggle along with the main operation, ensuring that the module correctly indicates when the output is valid and when it is ready for new data.
- Throughout the simulation, it is observed that the flip-flop maintains stable output between clock edges, meaning the output q does not follow the input d directly but only updates on clock events.
- Overall, the waveform confirms that the D flip-flop behaves correctly by latching the input on the rising edge of the clock, holding it stable until the next clock edge, and resetting properly when required.

Digital Predistortion Circuits

❖ *CONSTELLATION MAPPER:*

C code:-

*constellation5.cpp Synthesis(solution1)

```
#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.

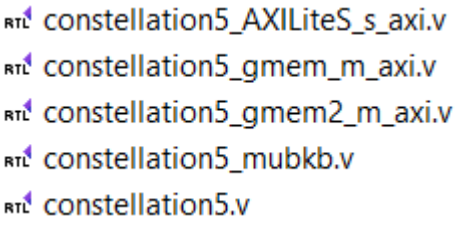
- **Process:**
  - For each symbol, extract 2 bits from the input.
  - Use these bits as an index into I/Q lookup tables.
  - Store the corresponding I and Q values in the output arrays.

- **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.

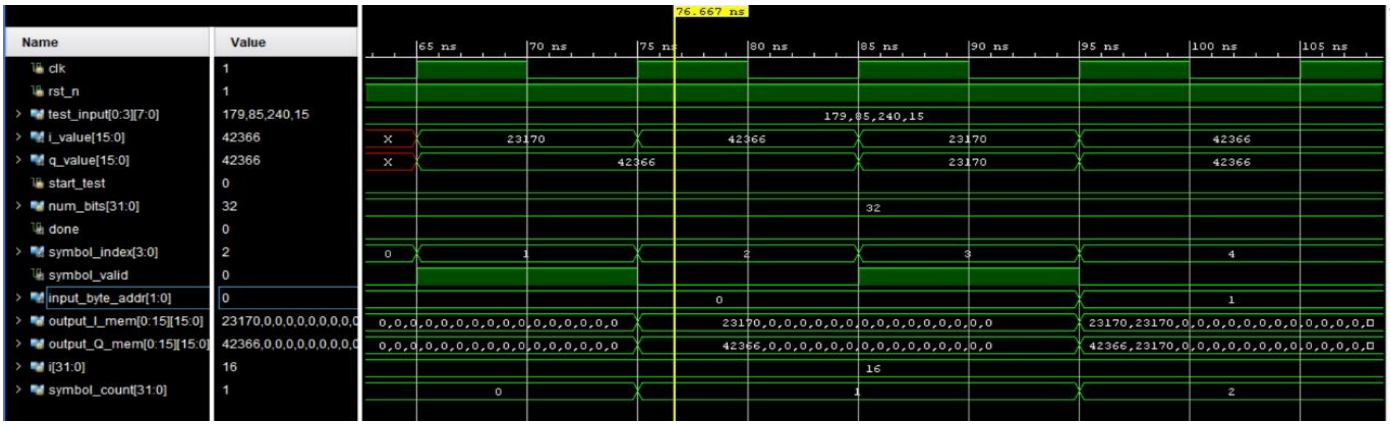
Verilog Code:-

The C code generates 5 Verilog files as shown:



The main file is constellation5.v .

Simulation Graph:



❖ *PULSE SHAPING FILTER:*

C code:-

Synthesis(solution1) | pulse_shape.cpp | pulse_shape_tb.cpp

```cpp
#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

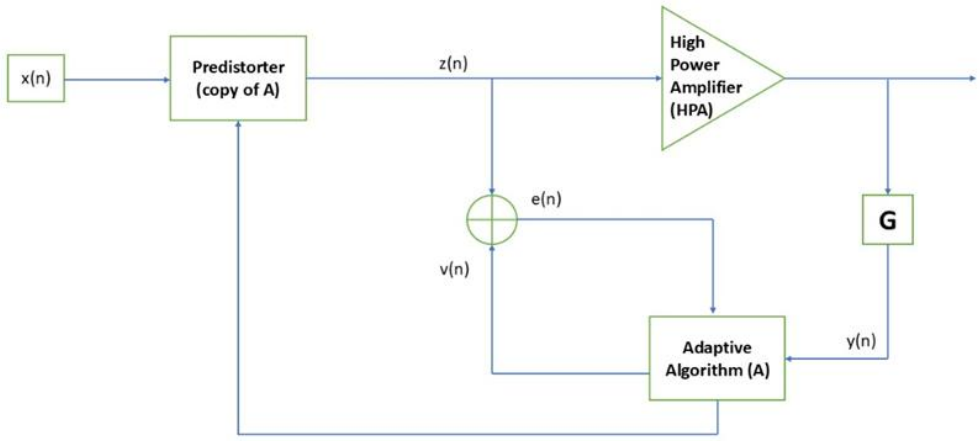
Efficiency:

- 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

❖ *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:-*

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;
    }
}
```

❖ *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:*

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 UP CONVERTER:*

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;


// 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];
```