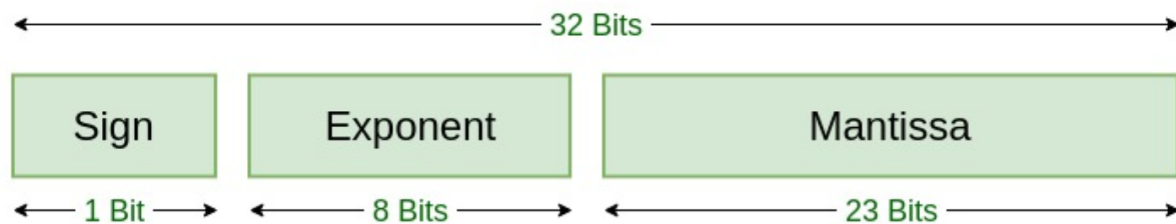


Lab 7

Tasks to be completed:

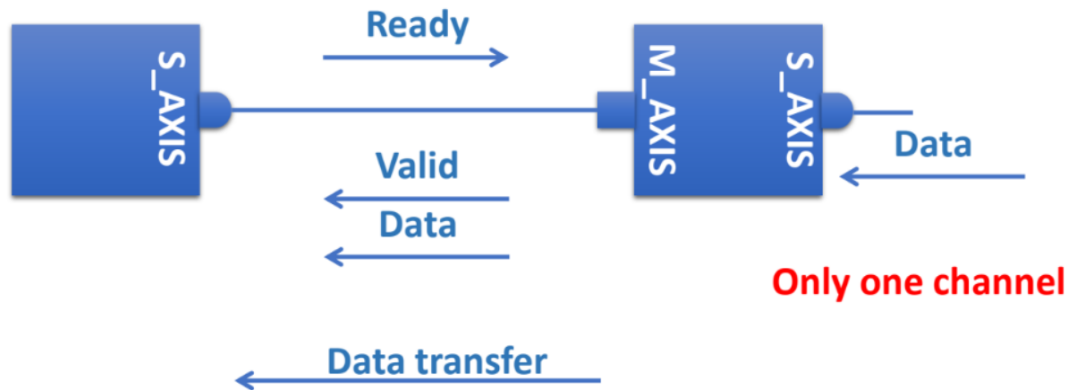
- AXI Stream Protocol Use
- Floating Point IP usage
- FFT IP Implementation and test bench.

Before starting with the implementation, let's revisit the single-precision and double-precision floating-point representation. IEEE 754 is a technical standard for the representation and computation of floating-point numbers. Two representation is possible according to this standard, namely, single precision and double precision. The format for both is given below:



Single Precision IEEE 754 Floating-Point Standard

The Floating-point IP will be using the AXI Protocols (AXI Stream in particular). Two concepts that will be useful in this Lab are Master-Slave and Valid Ready Signals. In AXI, the transactions take place between Master and Slave, as shown below:



The AXI Master initiates the transactions, and the slave responds to it. Many signals are associated with a transaction out of which the valid and ready signals are useful for this Lab. More information on these signals is covered in the Lab tutorial video.

Once the Vivado Project is made, add a top file.

Now we will add the FFT IP, go to the IP Catalog and add the Fast Fourier Transform IP.

Project Summary x IP Catalog x

Cores | Interfaces

Search: (5 matches)

Name	AXI4	Status	License	VLNV
Vivado Repository				
Communication & Networking				
Telecommunications				
LTE Fast Fourier Transform		Production	Purchase	xilinx.com:ip:lfe_fft.2.1
Wireless				
LTE Fast Fourier Transform		Production	Purchase	xilinx.com:ip:lfe_fft.2.1
Digital Signal Processing				
Transforms				
FFTs				
Fast Fourier Transform	AXI4-Stream	Production	Included	xilinx.com:ip:fft.9.1
LTE Fast Fourier Transform		Production	Purchase	xilinx.com:ip:lfe_fft.2.1

Details

Name: **Fast Fourier Transform**

Version: 9.1 (Rev. 4)


Interfaces: AXI4-Stream


Description: The Fast Fourier Transform (FFT) is a computationally efficient algorithm for computing the Discrete Fourier Transform (DFT). The FFT Core can compute 8 to 65536-point forward or inverse complex transforms on up to 12 parallel channels. The input data is a vector of complex values represented as two's-complement numbers 8 to 34 bits wide or single precision floating point numbers 32 bits wide. The phase factors can be 8 to 34 bits wide. All memory is on-chip using either Block RAM or Distributed RAM. Three arithmetic types are available: full-precision unscaled, scaled fixed-point, and block-floating point. Several parameters are run-time configurable: the point size, the choice of forward or inverse transform,

Configuring the FFT IP

We first need to change the transform length to 16, as we are doing a 16 point FFT

We can leave the Architecture as it is.

 Customize IP ✕

Fast Fourier Transform (9.1) 


[Documentation](#) [IP Location](#) [Switch to Defaults](#)

IP Symbol

Implementation Details

Latency

☒ Show disabled ports



Component Name

Configuration

Implementation

Detailed Implementation

Number of Channels

Transform Length

Architecture Configuration

Target Clock Frequency (MHz) [1 - 1000]

Target Data Throughput (MSPS) [1 - 1000]

Architecture Choice

☒ Automatically Select

☐ Pipelined, Streaming I/O

☐ Radix-2, Burst I/O

☐ Radix-2 Lite, Burst I/O

☐ Run Time Configurable Transform Length


OK


Cancel

Natural Order keeps the bits in the same way it was sent.

Cancel

For Detail Implementation, we make no changes.

 Customize IP ✕

Fast Fourier Transform (9.1) 

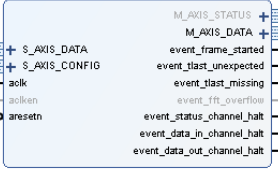
[Documentation](#) [IP Location](#) [Switch to Defaults](#)

IP Symbol

Implementation Details

Latency

☒ Show disabled ports



Component Name FFT ✕

Configuration

Implementation

Detailed Implementation

Memory Options

Data

Phase Factors

☒ Block RAM
☐ Distributed RAM

☒ Block RAM
☐ Distributed RAM

Number of stages using Block RAM for Data and Phase Factors

Reorder Buffer

☒ Block RAM ☐ Distributed RAM

☐ Optimize Block RAM Count Using Hybrid Memories

Optimize Options

Complex Multipliers

Butterfly Arithmetic

OK

Cancel

We can have a look at the Implementation Details and see that the input data is 64bit data with the Upper 32 bits for imaginary numbers and Lower 32 bits for Real numbers.

IP Symbol

Implementation Details

Latency

Information

implementation :

Pipelined, Streaming I/O

Transform Size

Largest :16

Smallest :16

Output Data Width :32

Resource Estimates Group

DSP48 Slices :6

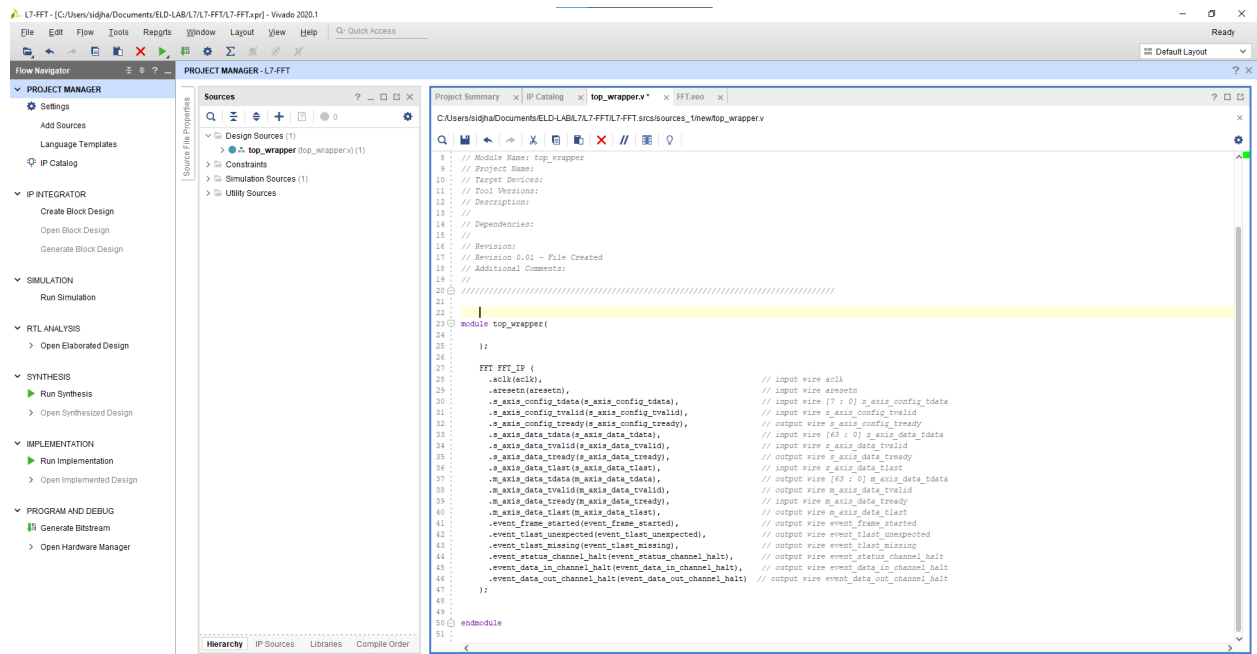
Block RAMs :2

AXI4 Stream Port Structure

S_AXIS_DATA - TDATA

Transaction	Field	Type
0	CHAN_0_XN_IM_0(63:32)	float_single
	CHAN_0_XN_RE_0(31:0)	float_single
1	CHAN_0_XN_IM_1(63:32)	float_single
	CHAN_0_XN_RE_1(31:0)	float_single
2	CHAN_0_XN_IM_2(63:32)	float_single
	CHAN_0_XN_RE_2(31:0)	float_single
3	CHAN_0_XN_IM_3(63:32)	float_single
	CHAN_0_XN_RE_3(31:0)	float_single
...		
15	CHAN_0_XN_IM_15(63:32)	float_single

We can then copy the instantiation template to the top_wrapper we created earlier.



Now the top_wrapper code:

```
23 module top_wrapper(  
24     input aclk,  
25     input aresetn,  
26  
27     input [31:0] in_data, // Input Data Stream  
28     input in_valid,  
29     input in_last,  
30     output in_ready,  
31  
32     input [7:0] config_data, // Config Data Stream  
33     input config_valid,  
34     output config_ready,  
35  
36     output [31:0] out_data, // Output Data Stream  
37     output out_valid,  
38     output out_last,  
39     input out_ready  
40 );  
41  
42 wire [63:0] data_fft; // FFT IP takes 64bit data, so padding the upper 32-bits with 0's.  
43 assign data_fft[63:32] = 32'd0;  
44 assign data_fft[31:0] = in_data; // Real Data from the Input Stream  
45  
46 wire [63:0] out_fft; // Output of FFT is 64 bit  
47  
48 wire  
49 event_frame_started,  
50 event_tlast_unexpected,  
51 event_tlast_missing,  
52 event_status_channel_halt,  
53 event_data_in_channel_halt,  
54 event_data_out_channel_halt; // Additional Event Signals  
55
```

Creating the interfaces for Input, Configuration and Output.

Each AXI Stream has data, valid and ready. While the input & output ports have last signals.

We can also create separate wires which are of 64bit length for communication with the FFT IP.

Here the upper 32bits will be for imaginary, as we have no imaginary data we can make that 0

Store the real data in the lower 32bits

We can then connect the ports to FFT IP itself.

```
56 FFT FFT_IP (  
57     .aclk(aclk),                                // input wire aclk  
58     .aresetn(aresetn),                          // input wire aresetn  
59  
60     .s_axis_config_tdata(config_data),           // input wire [7 : 0] s_axis_config_tdata  
61     .s_axis_config_tvalid(config_valid),         // input wire s_axis_config_tvalid  
62     .s_axis_config_tready(config_ready),         // output wire s_axis_config_tready  
63  
64     .s_axis_data_tdata(data_fft),               // input wire [63 : 0] s_axis_data_tdata  
65     .s_axis_data_tvalid(in_valid),              // input wire s_axis_data_tvalid  
66     .s_axis_data_tready(in_ready),              // output wire s_axis_data_tready  
67     .s_axis_data_tlast(in_last),               // input wire s_axis_data_tlast  
68  
69     .m_axis_data_tdata(out_fft),                // output wire [63 : 0] m_axis_data_tdata  
70     .m_axis_data_tvalid(out_valid),             // output wire m_axis_data_tvalid  
71     .m_axis_data_tready(out_ready),             // input wire m_axis_data_tready  
72     .m_axis_data_tlast(out_last),              // output wire m_axis_data_tlast  
73  
74     .event_frame_started(event_frame_started),  // output wire event_frame_started  
75     .event_tlast_unexpected(event_tlast_unexpected), // output wire event_tlast_unexpected  
76     .event_tlast_missing(event_tlast_missing),  // output wire event_tlast_missing  
77     .event_status_channel_halt(event_status_channel_halt), // output wire event_status_channel_halt  
78     .event_data_in_channel_halt(event_data_in_channel_halt), // output wire event_data_in_channel_halt  
79     .event_data_out_channel_halt(event_data_out_channel_halt) // output wire event_data_out_channel_halt  
80 );  
81  
82 assign out_data = out_fft[31:0]; // Last 32-bits being displayed. As real part is needed.  
83 endmodule  
84
```

And then assign the output which will be 32bits the lower half(bottom 32bits) of the FFT IPs output..

Next we need to make the testbench:

```
23 module tb_FFT();
24
25     reg aclk; // Making reg type for Inputs
26     reg aresetn;
27
28     reg [31:0] in_data;
29     reg in_valid;
30     reg in_last;
31     wire in_ready;
32
33     reg [7:0] config_data;
34     reg config_valid;
35     wire config_ready; // Wire for outputs for all Groups of Stream Signals
36
37     wire [31:0] out_data;
38     wire out_valid;
39     wire out_last;
40     reg out_ready;
41
42     reg [31:0] input_data [15:0]; // Creating a ROM, for the input data to the FFT IP
43
44     integer i;
45
46     top_wrapper tb_in( // Wrapper for the Top Module
47         .aclk(aclk),
48         .aresetn(aresetn),
49
50         .in_data(in_data),
51         .in_valid(in_valid),
52         .in_ready(in_ready),
53         .in_last(in_last),
54
55         .config_data(config_data),
56         .config_valid(config_valid),
57         .config_ready(config_ready),
58
59         .out_data(out_data),
60         .out_valid(out_valid),
61         .out_last(out_last),
62         .out_ready(out_ready)
63     );
64
```

We start by making regs for inputs and wires for outputs and instantiate the top module with these regs and wires.

We also need to create a ROM, which will store the input data we want to send to the FFT. As we have a vector of length 16, each word having 32bits of data we can create the input data register array.

Additionally you have to initialize the various registers you created.

As the reset is active low, setting that to 0 will enable the reset functionality.

We can set the out_ready signal to be 1 here so that the FFT puts output data on the data bus as soon as it is ready.

```
65 always
66 begin
67     #5 aclk = ~aclk; // CLK with 10 unit period
68 end
69
70 initial begin
71     aclk = 0;
72     aresetn = 0;
73
74     in_valid = 1'b0;
75     in_data = 32'd0;
76     in_last = 1'b0;
77
78     out_ready = 1'b1; // Initializing OUT_READY to 1, inorder to tell the FFT
79                       // that outputs can be generated whenever ready
80                       // Failure to do so leads to "back-pressure"
81
82     config_data = 8'd0;
83     config_valid = 1'b0;
84 end
85
86 initial begin
87     #70 // As Reset needs to be activated for atleast 2 cycles we have given 70 units of delay
88     aresetn = 1;
89
90     input_data[0] = 32'b00100101100011010011000100110010;
91     input_data[1] = 32'b00111111001111100011111010111101;
92     input_data[2] = 32'b00111111011111101001100011111101;
93     input_data[3] = 32'b00111111000101100111100100011000;
94     input_data[4] = 32'b10111110010101001110011011001101;
95     input_data[5] = 32'b10111111010111011011001111010111;
96     input_data[6] = 32'b10111111011100110111100001110001;
97     input_data[7] = 32'b1011111011010000001111111001001;
98     input_data[8] = 32'b0011111011010000001111111001001;
99     input_data[9] = 32'b00111111011100110111100001110001;
100    input_data[10] = 32'b00111111010111011011001111010111;
101    input_data[11] = 32'b00111110010101001110011011001101;
102    input_data[12] = 32'b10111111000101100111100100011000;
103    input_data[13] = 32'b10111111011111101001100011111101;
104    input_data[14] = 32'b10111111001111100011111010111101; // Input Data generated from python
105    input_data[15] = 32'b10100101100011010011000100110010;
106 end
107
108
```

Now in order to create the AXI Handshake we can do the following

Put the data on the Data bus and then set valid to be HIGH

While the READY signal is LOW keep VALID HIGH, this situation for the signals triggers the handshake and completes the transaction.

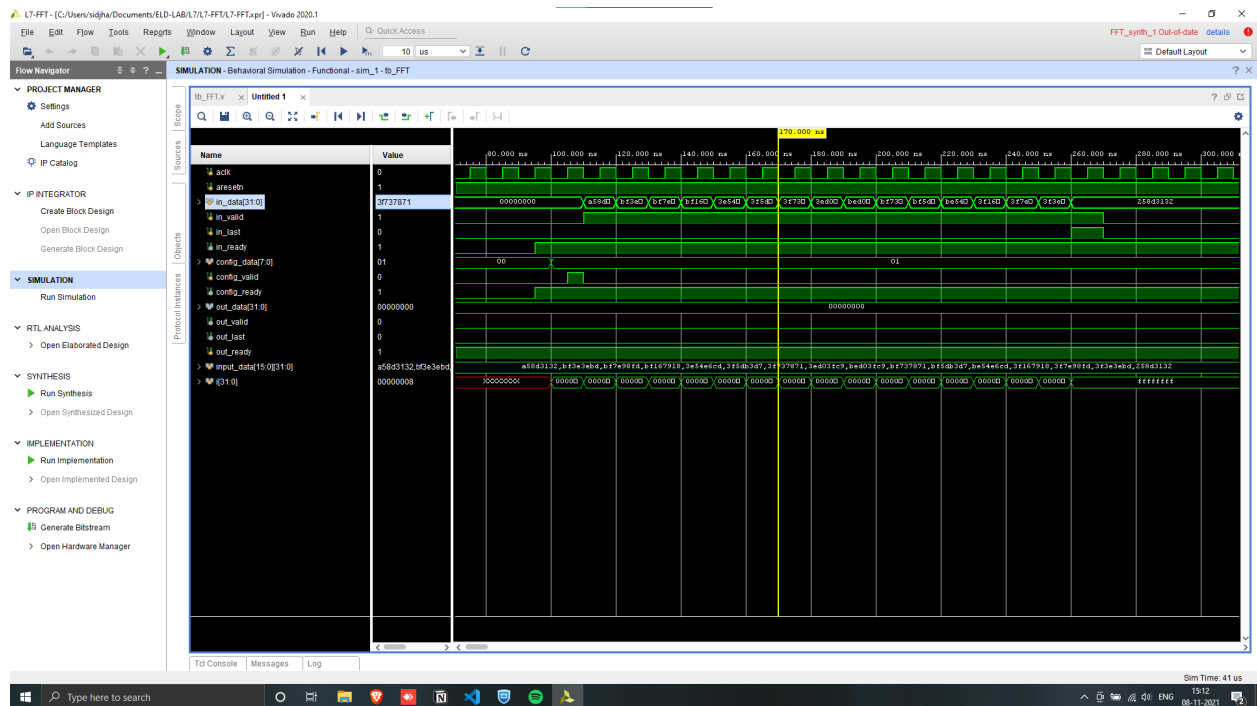
We do the same thing for the Input data as well. For every element in the register array, using a for loop we put data on the bus and assert valid HIGH. Wait for the READY signal to be HIGH, this completes the transaction.

```
108 initial begin // Config Data initial block
109     #100
110     config_data = 1;
111     #5 config_valid = 1;
112
113     while(config_ready == 0) begin
114         config_valid = 1;
115     end
116     #5 config_valid = 0;
117 end
118
119 initial begin // Input Port Initial Block
120     #100
121     for (i = 15; i>=0; i=i-1) begin
122         #10
123         if (i==0) begin // Last signal needs to be generated once the last data is sent
124             // In this case once we reach the 0th position we can assert
125             // last signal to be 1.
126             in_last = 1'b1;
127         end
128
129         in_data = input_data[i]; // Passing data stored in memory to in_data port
130         in_valid = 1'b1; // Once data is put on the bus make the valid HIGH.
131
132         while (in_ready == 0) begin // Waiting for AXI Handshake, for the in_ready to be 1.
133             in_valid = 1'b1;
134         end
135     end
136     #10
137     in_valid = 1'b0;
138     in_last = 1'b0;
139     // Once all the transactions are completed assert the valid and last to 0.
140 end
141
142 initial begin // Output Port Initial Block
143     #100 // Giving delay so that all the input data can be stored in ROM
144
145     wait(out_valid == 1);
146     #300 out_ready = 1'b0; // Adding a 300 unit delay so that all the data that needs to be sent can be put on the Data Bus
147 end
148
149
```

While for the output port the test bench is the slave port in that case so here we have to wait for the output valid to be HIGH this would signify that data has been put on the bus. We made the READY signal for this port HIGH initially the handshake takes place instantly then gives us the output data we want.

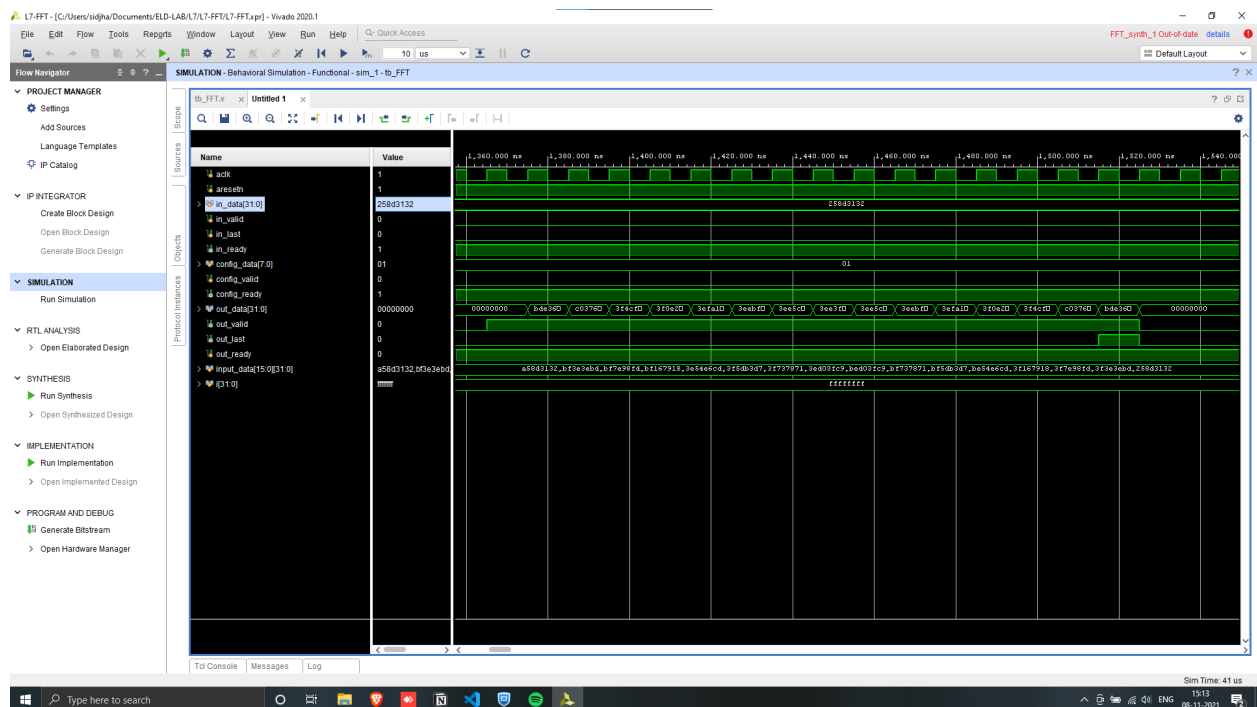
Simulation Results.

Here we can see the input data being transmitted.



Here we can see the output data being transmitted.

The last signals in both the simulations are generated as soon as the last data is transmitted.



You can choose any vector and check the functionality using the tools mentioned in the lab video.

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>