CMPEN417 Final Project
Rajeswara Racharla
Reema Dhar
Arsalan Saif
5-1-2022

1. A brief introduction of the project, your approach to designing each function, what is the goal for each function and the structure of your entire design

In this project we are creating a cordic based signal processor. There are two main blocks to this function, which are the M-tap Complex filter and the other being the cordic rotator. These two blocks work hand and hand to complete the operation. The N-Tap Complex FIR filter was first passed in a complex signal, which is represented in the cartesian coordinate system. The FIR provides computation efficiency in our implementation as it is suitable for multi-rate applications. The complex FIR filter program will take the complex signal and filter the impulse response. In the FIR function we first shift the register by 1 to the right and then multiply the output separately. The output signal is determined by computing previous inputs and current inputs using the FIR function. Once this process is complete and the signal is fully filtered, it is passed into the 2nd block, which is the Cordic rotator. The cordic rotator will take the vector and convert them from cartesian form to polar form. From here it will take the vector and rotate it from whatever position it is at until it reaches the y value of zero. This would ensure that the theta is equal to zero due to sin (0) = 0. This rotator only rotates between values of 0 and pi/2 so there will be no values which are outside of quadrant 1. If it is found to be outside of this range, then the cordic rotator will execute a while loop which will keep on rotating the vector by 90 degrees until it is in in the range of the 1st quadrant. The purpose of this process is to accurately calculate the magnitude and angle/phase of the vector and then outputting the values every time theta changes to keep track of the changes. The scaling factor of K will be calculated through every theta change and the final summed K value will be printed at the end of the operation.

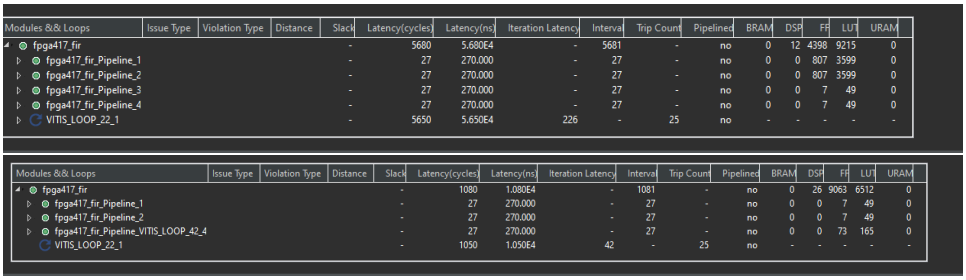
2. What pragma optimization are applied at which place on which function(FIR or CORDIC)? Showing some code snippets, figures and tables is preferred

In the FIR function, we used pragma array_partition and pragma unroll and pragma pipeline. A cordic function is constructed with the usage of arrays to represent the vectors. To improve latency and parallelism.

The outer loop pipeline was switched off since it causes interval errors with memory dependencies.

```
void fpga417_fir(int* input_real, int* input_img, int* output_real, int* output_img){

    int shift_reg_r[KERNEL_SIZE]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int shift_reg_i[KERNEL_SIZE]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int result_r[KERNEL_SIZE]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int result_i[KERNEL_SIZE]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    int input;
#pragma HLS ARRAY_PARTITION variable=shift_reg_r type=complete
#pragma HLS ARRAY_PARTITION variable=shift_reg_i type=complete
#pragma HLS ARRAY_PARTITION variable=coeffr type=complete
#pragma HLS ARRAY_PARTITION variable=coeffi type=complete


    for(input=0;input<KERNEL_SIZE;input++){
#pragma HLS PIPELINE off

        for(int j=KERNEL_SIZE-1;j>0;j--){
#pragma HLS unroll

            shift_reg_r[j]=shift_reg_r[j-1];
            shift_reg_i[j]=shift_reg_i[j-1];
        }
        shift_reg_r[0]=input_real[input];
        shift_reg_i[0]=input_img[input];


        for(int i=0;i<KERNEL_SIZE;i++){
#pragma HLS unroll

            result_r[i]=((shift_reg_r[i]*coeffr[i])-(shift_reg_i[i]*coeffi[i]));
            result_i[i]=((shift_reg_r[i]*coeffi[i])+(shift_reg_i[i]*coeffr[i]));
        }

        for(int k=0;k<KERNEL_SIZE;k++){
#pragma HLS PIPELINE
            output_real[input]+=result_r[k];
            output_img[input]+=result_i[k];
        }
```

a. **Latency cycles before and after**:



Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	5680	5.680E4	-	5681	-	no	0	12	4398	9215	0
fpga417_fir_Pipeline_1				-	27	270.000	-	27	-	no	0	0	807	3599	0
fpga417_fir_Pipeline_2				-	27	270.000	-	27	-	no	0	0	807	3599	0
fpga417_fir_Pipeline_3				-	27	270.000	-	27	-	no	0	0	7	49	0
fpga417_fir_Pipeline_4				-	27	270.000	-	27	-	no	0	0	7	49	0
VITIS_LOOP_22_1				-	5650	5.650E4	226	-	25	no	-	-	-	-	-

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	1080	1.080E4	-	1081	-	no	0	26	9063	6512	0
fpga417_fir_Pipeline_1				-	27	270.000	-	27	-	no	0	0	7	49	0
fpga417_fir_Pipeline_2				-	27	270.000	-	27	-	no	0	0	7	49	0
fpga417_fir_Pipeline_VITIS_LOOP_42_4				-	27	270.000	-	27	-	no	0	0	73	165	0
VITIS_LOOP_22_1				-	1050	1.050E4	42	-	25	no	-	-	-	-	-

Latency improvement: 5680->1080

b. **Resource utilization before and after**:

12 DSP->26 DSP, LUT 9215-> 6512, FF 4398->9063

c. **Analysis of before and after change, why this happen**:

With the array_partition, the array goes from a memory with one read/write port to having a different register for each array elements so they can be accessed in parallel, the pragma unroll is used to take advantage of this, it can access these separate memories in parallel. This leaves very little idle time. The third loop has pipelining since they add on top of each other and cannot happen parallelly so it cannot be unrolled.

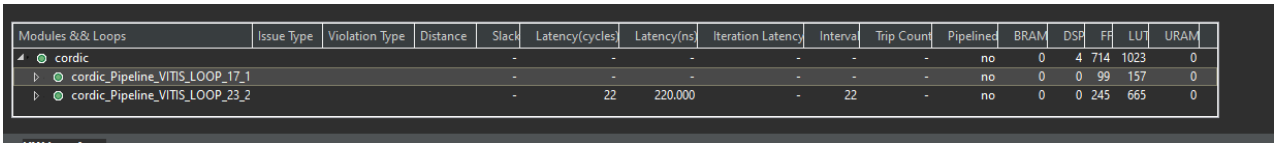
3. **If any arbitrary precision data type are used, report**:
   a. **The range of numbers your data type supported**:

The arbitrary precision data type as ap_fixed. We used (32,20) each data type is 32 bits. The 20 bits are used to represent the int and 12 bits are used to represent the fraction part, as this avoids the overflow/underflow issue.

b. **Why do you choose such ranges**:

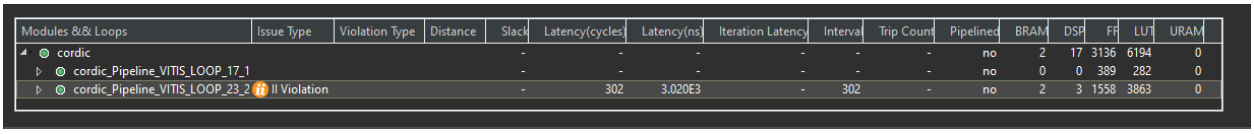
The number of iterations is 20, and the value of 20 in the phase array is 0.00000381469726560650. ln(0.00000381469726560650) is –12.5 so the digits needed for fraction part is 13. 12 is good enough for our purposes and 32 bits total is good for alignment.

c. **Resource utilization compared with standard double floating-point**:



Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cordic				-	-	-	-	-	-	no	0	4	714	1023	0
cordic_Pipeline_VITIS_LOOP_17_1				-	-	-	-	-	-	no	0	0	99	157	0
cordic_Pipeline_VITIS_LOOP_23_2				-	22	220.000	-	22	-	no	0	0	245	665	0

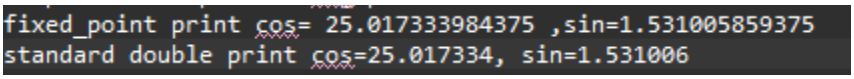
This is with fixed point.



Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cordic				-	-	-	-	-	-	no	2	17	3136	6194	0
cordic_Pipeline_VITIS_LOOP_17_1				-	-	-	-	-	-	no	0	0	389	282	0
cordic_Pipeline_VITIS_LOOP_23_2	II Violation			-	302	3.020E3	-	302	-	no	2	3	1558	3863	0

This is double, latency is higher, with more DSP. Also has violation.

d. **Precision comparison with standard double floating-point**:



```
fixed_point print cos= 25.017333984375 ,sin=1.531005859375
standard double print cos=25.017334, sin=1.531006
```

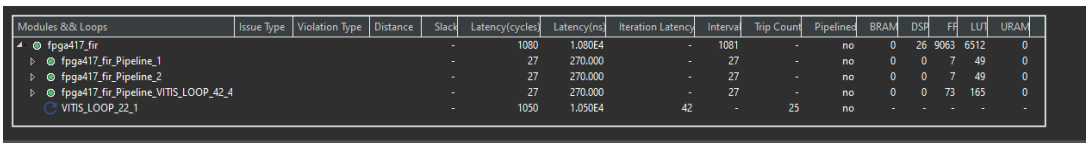
Precision is much higher than standard double.

4. A summarization of your design, how did you achieve the final optimal design, what is the best running frequency, latency cycles, resource utilization you achieved

We take in an array of real and imaginary numbers; in the fir we take in both these arrays then use a shift register in a loop to take in the inputs one by one shifting them in. Then multiply them with the complex coefficients. Then we sum all the multiplication results in a different loop. This is done for every input in the array, streaming them in. Then we use the CORDIC rotator, we first get it within 0 to pi/2 by rotating by 90 if negative x and y values. The cordic then rotates the vector till the degree is 0, we know this when sin theta is 0, we get magnitude by multiplying cos theta with cordic_gain fixed.

Optimize FIR with array_partition, unroll and pipeline.

To test it, we ran CORDIC separately from the FIR. For Cordic we changed the number of iterations from 32 to 20 but kept the fixed-point values are the same. Other resource utilizations can we seen in the figure below for the FIR.



Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	1080	1.080E4	-	1081	-	no	0	26	9063	6512	0
fpga417_fir_Pipeline_1				-	27	270.000	-	27	-	no	0	0	7	49	0
fpga417_fir_Pipeline_2				-	27	270.000	-	27	-	no	0	0	7	49	0
fpga417_fir_Pipeline_VITIS_LOOP_42_4				-	27	270.000	-	27	-	no	0	0	73	165	0
VITIS_LOOP_22_1				-	1050	1.050E4	42	-	25	no	-	-	-	-	-