CS 577: Project Report

Group Number	6
Name of the top modules	ADPCM_MAIN()
Link for Git-Hub Repo	https://github.com/VLSI-PROJECT5/ADPCM.git

Group Members	Roll Numbers
Deep Bhuniya	204101021
Nishanth Galeti	204101023
Naga Siva Kumar Reddy Nallagatla	204101037
Navjot Singh	204101038
Kaushal Dewangan	204101071

Introduction

Audio coding is used to compress digital representation of high precision audio signals for the purpose of efficient transmission of signal. Normally, There will be more redundant bits when we are transmitting the signal. So, If we share less data, then efficient transmission of signal is possible. The signal with minimum number of bits in audio coding representation when getting transparent signal reproduction can't be different from the original input.

Differential Pulse Code Modulation (DPCM) is a technique, in which we can remove the redundant bits. Which will reduce the overall bit rate and increase the transmission efficiency. In DPCM, we are going to use a concept called "Prediction". That is, Based on the knowledge of the past sample, We will predict the value of the present sample. This will be done by "Predictor"(In ADPCM, it will be done by Adaptive Predictor). Here, the prediction may not be same, But it will be very close to the actual sample value.

DPCM encoder has two components, Quantizer and Predictor And DPCM decoder has Predictor components.

Predicted_Error = Actual_Input_Sample - Predicted_Sample

Then this, Predicted_Error will be given as input to Quantizer, then it will output Quantized_Error. This Quantized_Error will be given as input to Encoder, which will given encoded signal.

This encoded signal will be given as input for DPCM Decoder. Now, this decoder will decode the signal and will produce the Quantized_Error. Now, again we are going to Predict the Output with the help of the Predictor.

Now,

Quantized_Version_Signal = Quantized_Error + Predicted_Output.

We will consider this " Quantized_Version_Signal " as the output signal.

Note :- Even though we are able to remove the redundant bits from the signal, but still we are not getting good efficiency. We can get this by, ADPCM.

Adaptive differential pulse code modulation (ADPCM) is a technique that is converts analog signals to binary signal by taking frequent samples size of the sound and represents as sampled modulation in bits. The ADPCM technique is used in multimedia data compression because the method makes it possible to reduce bit flow without compromising quality.

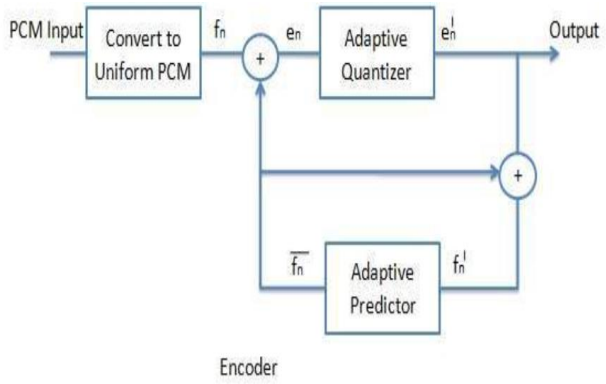
The Quantizer and Predictor are adaptive in ADPCM. This means that the stepsize of the Quantizer and Predictor will vary according to the input signal.

ADPCM Quantization:-

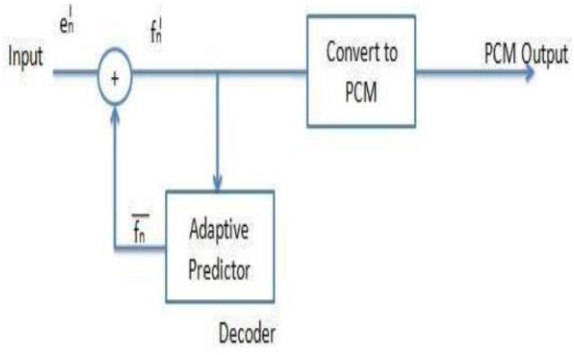
ADPCM adapts the quantizer step size according to the rms value of the input signal. Inorder to do this, we need to estimate the standard deviation of the signal. We are going to do this by 2 ways.

1) Adaptive Quantization Forward Estimation.
2) Adaptive Quantization Backward Estimation.

Both of the above ways are effective in producing the result. But Mostly we will go with " Adaptive Quantization Backward Estimation ", because we don't need buffers to store here.

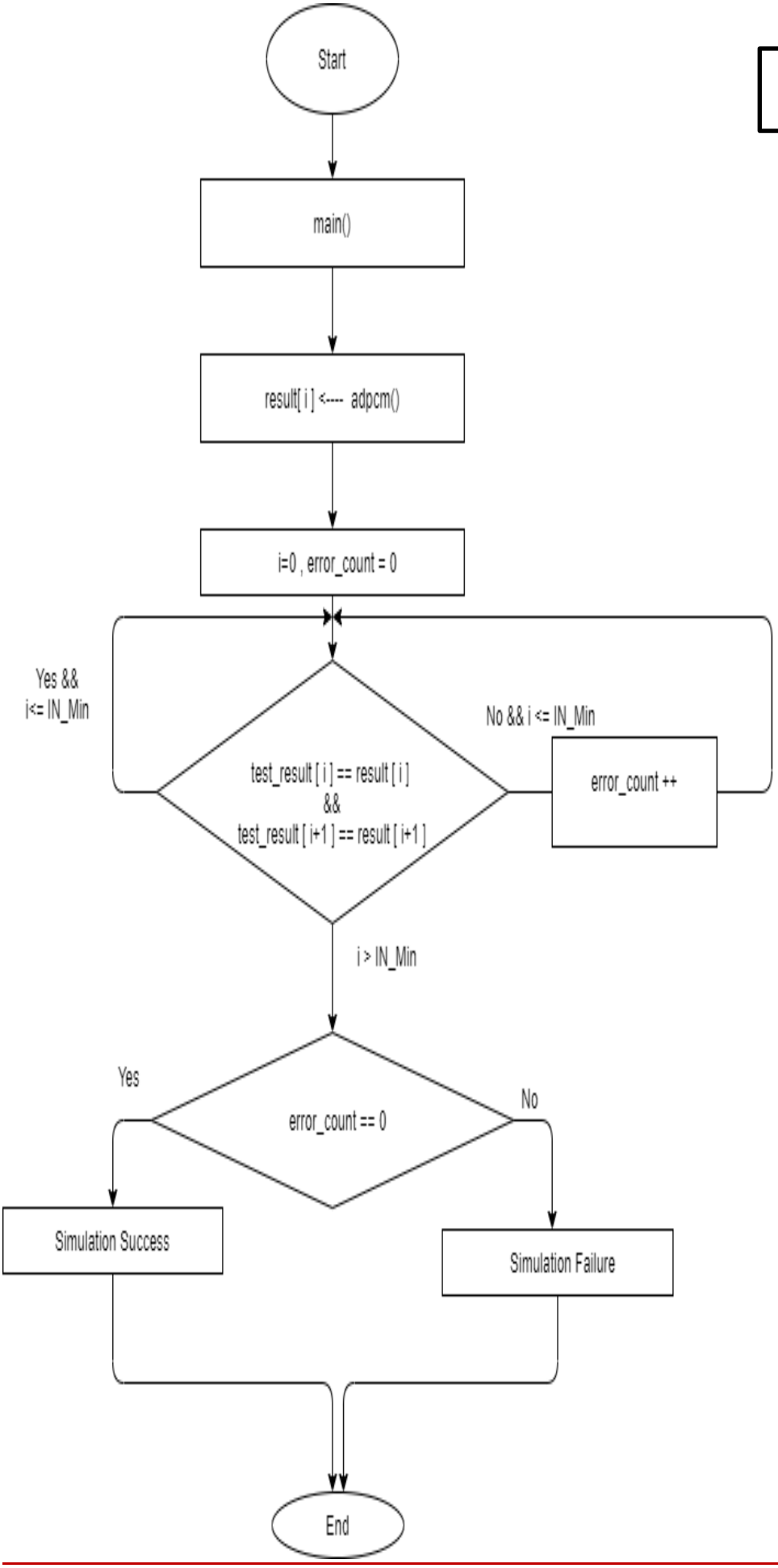


ADPCM Encoder

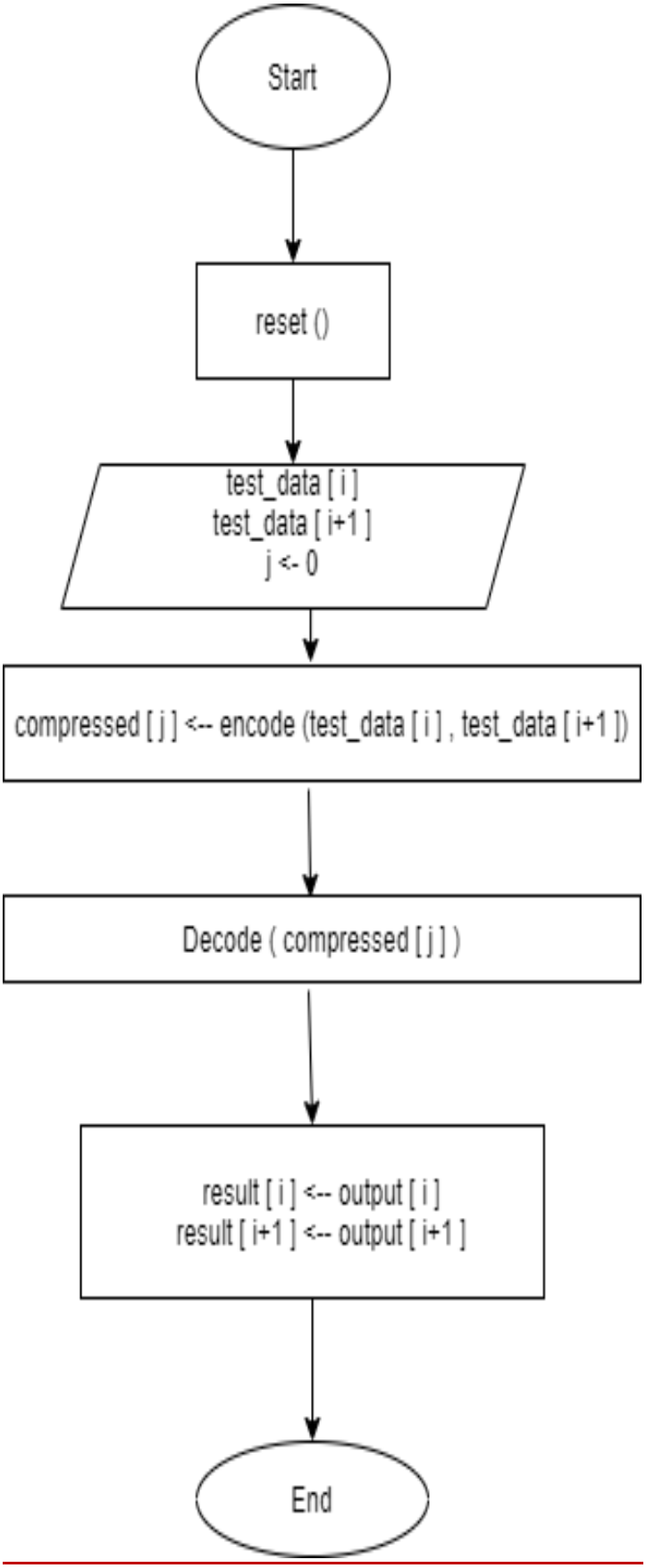


ADPCM Decoder

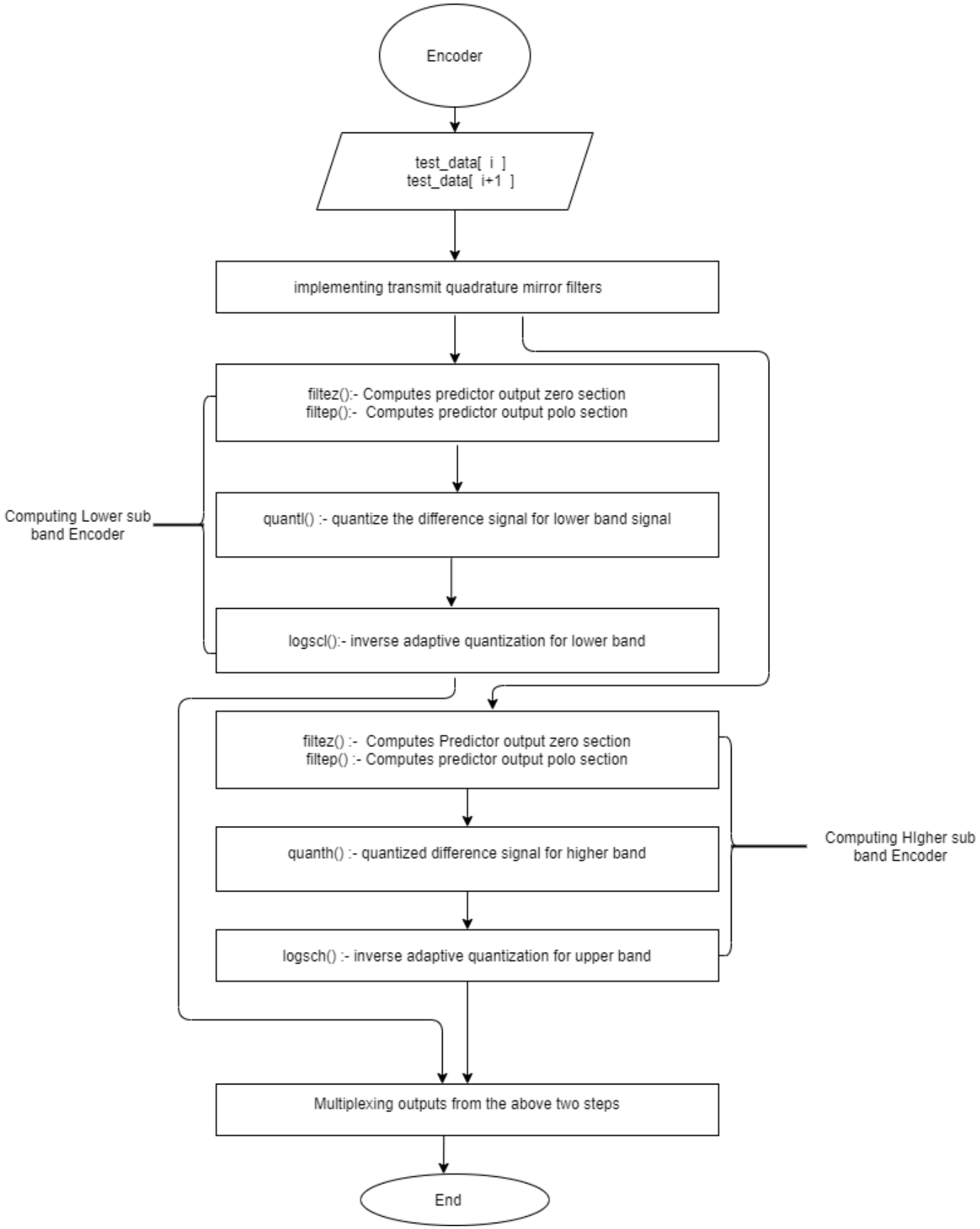
Main () Function



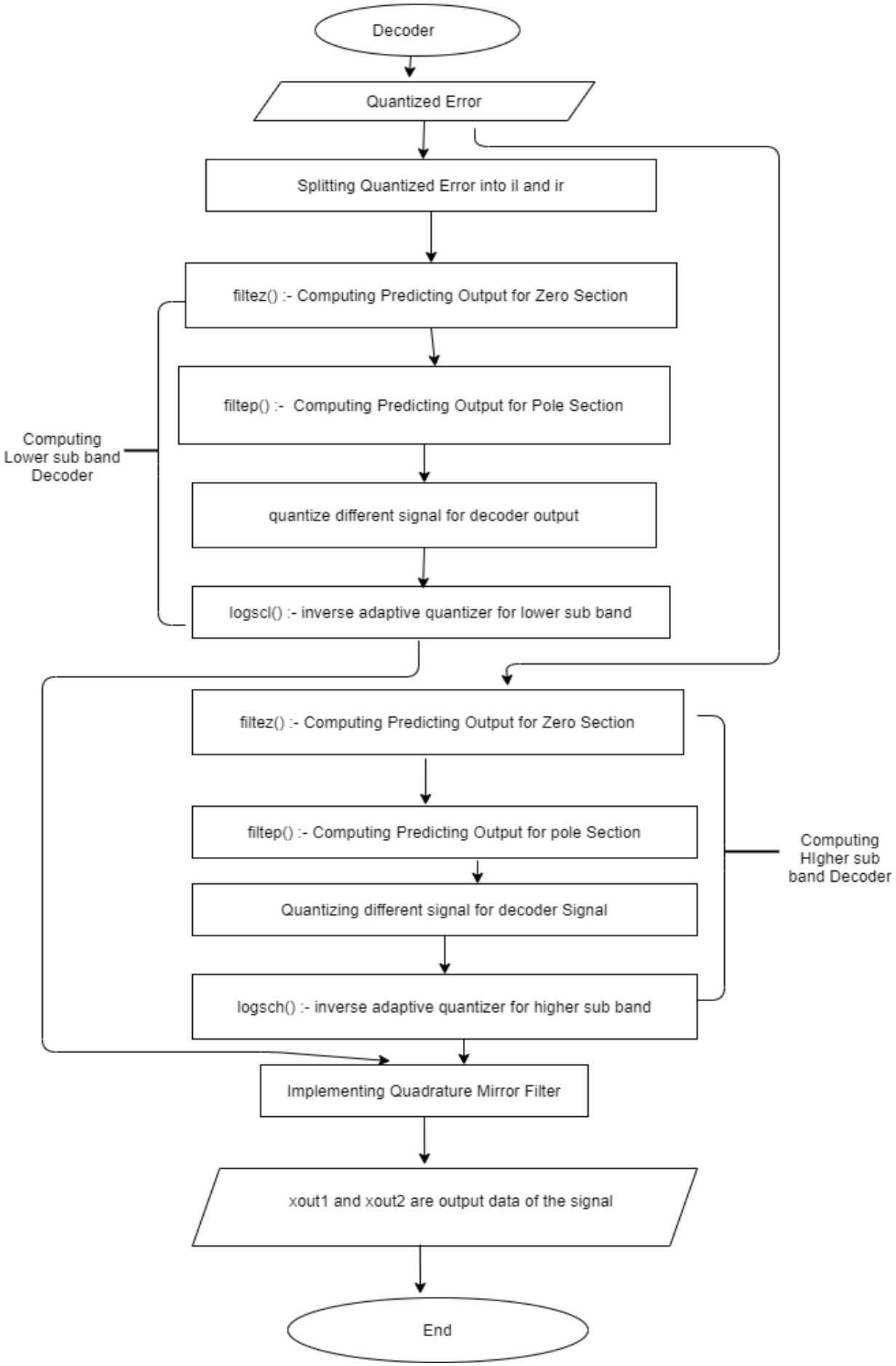
ADPCM() function flow



Flow Chart For Encoder :-



Flow Chart For Decoder :-



Phases :-

1) Screenshots of final optimized code

1.1 Synthesis screenshot :

1.2 C/RTL co-simulation screenshot :

2) Type of Optimizations performed

Phase 1:-

1.1 Simulation Screenshot

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	66	-
FIFO	-	-	-	-	-
Instance	1	103	5130	9336	-
Memory	21	-	14	11	0
Multiplexer	-	-	-	1697	-
Register	-	-	1005	-	-
Total	22	103	6149	11110	0
Available	280	220	106400	53200	0
Utilization (%)	7	46	5	20	0

- **Detail**
  - **Instance**

Instance	Module	BRAM_18K	DSP48E	FF	LUT	URAM
grp_decode_fu_258	decode	1	66	2951	4461	0
grp_encode_fu_326	encode	0	37	2167	4219	0
grp_reset_fu_409	reset	0	0	12	656	0
Total	3	1	103	5130	9336	0

  - **DSP48E**

Cosimulation Report for 'adpcm_main'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	8534	8534	8534	NA	NA	NA

Phase 2

Benchmark	Type(Area /Latency)	Resource Utilization				Latency (no of clock cycles)		Optimizations
		LUT	FF	DSP	BRAM	Min	Max	
Baseline	_	9114	5408	95	10	17504	25904	_
Optimization 1	Latency	12515	6145	120	22	8114	12614	Loop Unroll, Code Rearranging
Optimization 2	Latency	12571 (23% utilization)	6196	120	22	7214	10714	Optimization 1 + Pipelining
Optimization 3	Area	11110 (20% utilization)	6149	103	22	7664 (Dec By 56%)	11164 (Dec By 56.9 %)	Allocation Directive + Function In-lining + Array Mapping

Final Co-Simulation Passed with 8534 Clock Cycles (Where each clock cycle = 10 ns)

Optimization 1 :

By going through entire code, we observed that by re-writing some parts of code, as well as by unrolling the loops partially or completely, we were able to decrease latency involved in the FOR loops by **53.6%**. And also by using some optimization techniques in IF-ELSE parts, we were able to reduce the difference between minimum and maximum latencies.

Pragma used for unrolling the loops :-

Example :- For complete loop unrolling = **#pragma HLS unroll**

Example :- For partial loop unrolling = **#pragma HLS unroll factor =2**

Optimization 2 :

After Optimization1, we observed that in some functions , between each iteration there isn't any dependency. So we applied pipeline pragma there and achieved overall latency drop by 58.7%. But in doing so, we observed that area utilization is increased , LUT's utilization increased to 23% from 17%.

Pragma used for applying pipeline concept :-

Example :- **#pragma HLS pipeline**

Optimization 3 :

Here, we found out following scenarios to optimize area,

1) So many functions are repeatedly called and by default each function call is taking different memory map, thus LUT's utilization is increasing. So function in-lining seems to be a good option to reduce LUT's, and INLINE is the directive for implementing this.

2) With the help of resource profile and schedule viewer, we identified the operators utilizing the highest number of LUT's. so by using pragma **ALLOCATION,** we limited the number of LUT's by some extent.

3) We used vertical mapping where bit width is increased but number of flip-flops going to decrease as two arrays mapping into same word location. For this we used **ARRAY MAP** directive to implement this technique.

Hence by applying all 3 techniques in optimization 3, we were able to reduce LUT's utilization from 23% to 20%.