FM demodulator

Yiming Ren A59005465

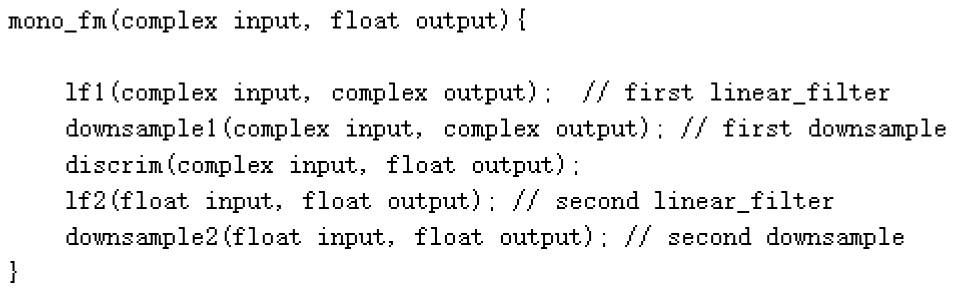
y5ren@ucsd.edu

- **Outline:**

  FM demodulator on Pynq is an integrated IP core. It can process the input signal from RTL2832 USB tuner. In this report, I'm going to explain my HLS implementation and optimization.

- **Mono FM demodulator:**

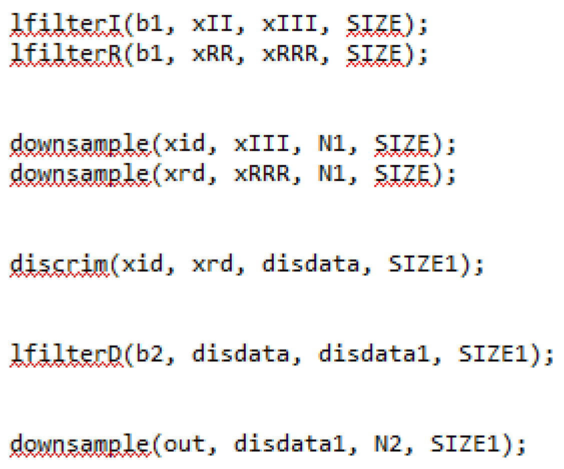
  A mono FM demodulator consists of 3 parts, including downsampler, linear filter, and discriminator. It processes the signal by calling and reorganizing these parts in a certain way.



```
mono_fm(complex input, float output){

    lf1(complex input, complex output);  // first linear_filter
    downsample1(complex input, complex output); // first downsample
    discrim(complex input, float output);
    lf2(float input, float output); // second linear_filter
    downsample2(float input, float output); // second downsample
}
```

Sample workflow of mono FM demodulator

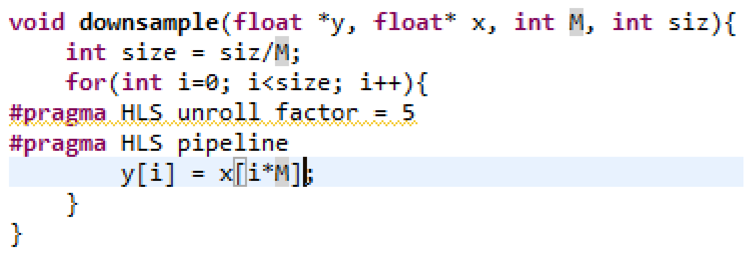
In HLS, I use Axistream multiple DMAs to pass the data to IP. With multiple DMAs, I can use 2 independent arrays of type *axis_t* for real and imaginary part of the complex data. Therefore, I have to split the complex type parameters each to 2 float type parameters, alternatively, splitting the function with complex parameter to 2 functions with float parameters. Also, to achieve target latency, I use dataflow to optimize the performance, which requires additional parameters being used. My frame of mono FM demodulator is like:



```
lfilterI(b1, xII, xIII, SIZE);
lfilterR(b1, xRR, xRRR, SIZE);


downsample(xid, xIII, N1, SIZE);
downsample(xrd, xRRR, N1, SIZE);


discrim(xid, xrd, disdata, SIZE1);


lfilterD(b2, disdata, disdata1, SIZE1);


downsample(out, disdata1, N2, SIZE1);
```

For linear filter I and R, xII and xRR are read from input, obviously representing the imaginary and real part of input. xIII and xRRR are the result from linear filter. Xid and xrd are the downsampled xIII and xRRR. They are later discriminated to disdata, which then is linear filtered and downsampled to the final output. SIZE is 1000 because 1000 elements are contained in each write, so SIZE1 is 100 since 1000 elements are downsampled to 100 elements.

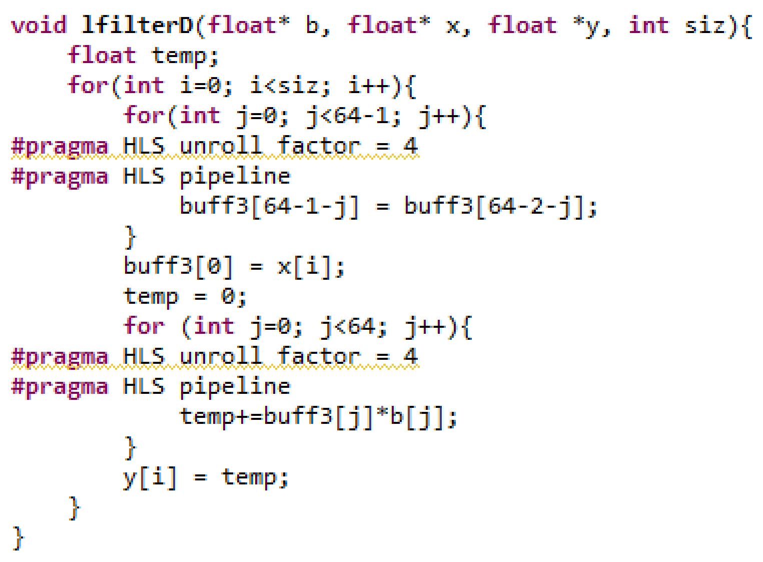
- **Downsampler:**



```
void downsample(float *y, float* x, int M, int siz){
    int size = siz/M;
    for(int i=0; i<size; i++){
#pragma HLS unroll factor = 5
#pragma HLS pipeline
        y[i] = x[i*M];
    }
}
```

Integer M indicates the downsample factor. It is 10 in first downsampling and 5 in second. Downsampler catches every i*M element in given input. In mono FM we don't consider offset.

- **Linear filter:**



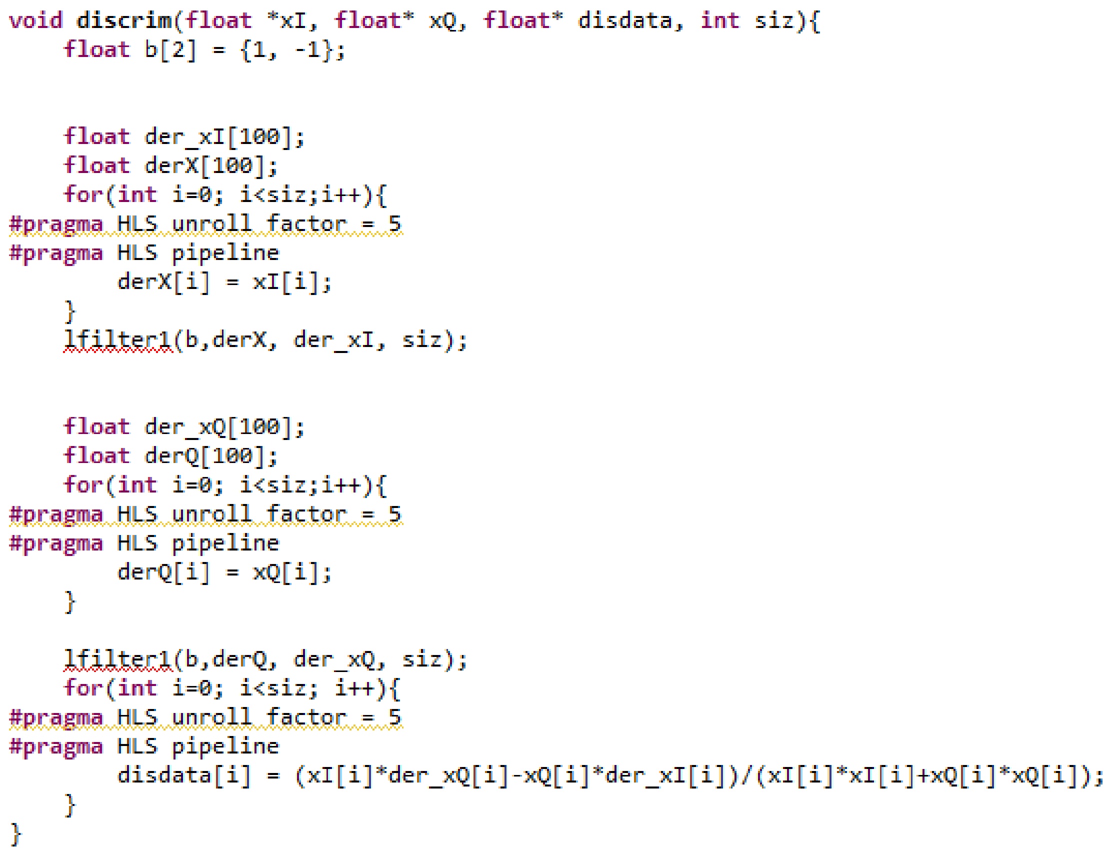
```
void lfilterD(float* b, float* x, float *y, int siz){
    float temp;
    for(int i=0; i<siz; i++){
        for(int j=0; j<64-1; j++){
#pragma HLS unroll factor = 4
#pragma HLS pipeline
            buff3[64-1-j] = buff3[64-2-j];
        }
        buff3[0] = x[i];
        temp = 0;
        for (int j=0; j<64; j++){
#pragma HLS unroll factor = 4
#pragma HLS pipeline
            temp+=buff3[j]*b[j];
        }
        y[i] = temp;
    }
}
```

Take linear filter D as an example. Linear filter functions are implemented follow the equation. Buffer arrays should be global to make sure IP could utilize the data from last function call. Therefore, 3 global buffers are needed to identify buffers for xII, xRR, and disdata. I create 3 linear filter functions with same function body but different global buffer name. For example, linear filter D uses ‘buff3’, while others use ‘buff2’ and ‘buff1’.

- **Discriminator:**

Since we use C code in HLS, we pass pointers for array. To get der_xQ and der_xI without changing data in xI and xQ, additional array named derQ and derX are created. They can also be used for dataflow. Since discriminator is only called once in mono FM demodulator, the size of above arrays is constant 100. Linear filter 1 is different to other linear filters because size of parameter ‘b’ is

2, while size of ‘b’ is 64 in other linear filters.

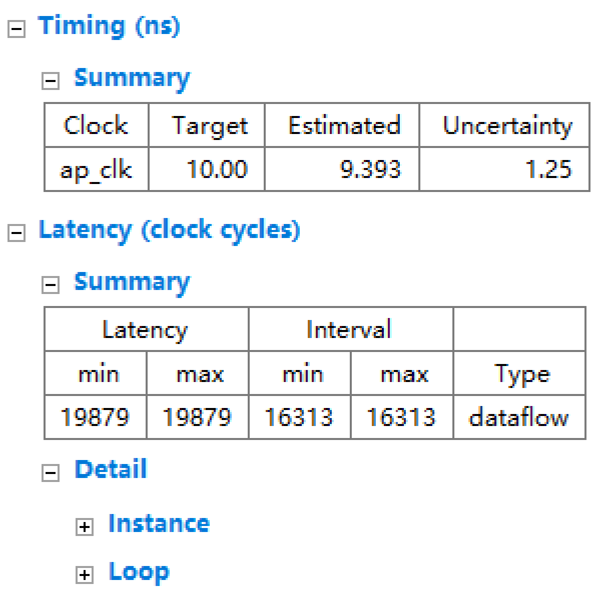


```
void discrim(float *xI, float* xQ, float* disdata, int siz){
    float b[2] = {1, -1};


    float der_xI[100];
    float derX[100];
    for(int i=0; i<siz;i++){
#pragma HLS unroll factor = 5
#pragma HLS pipeline
        derX[i] = xI[i];
    }
    lfilter1(b,derX, der_xI, siz);


    float der_xQ[100];
    float derQ[100];
    for(int i=0; i<siz;i++){
#pragma HLS unroll factor = 5
#pragma HLS pipeline
        derQ[i] = xQ[i];
    }

    lfilter1(b,derQ, der_xQ, siz);
    for(int i=0; i<siz; i++){
#pragma HLS unroll factor = 5
#pragma HLS pipeline
        disdata[i] = (xI[i]*der_xQ[i]-xQ[i]*der_xI[i])/(xI[i]*xI[i]+xQ[i]*xQ[i]);
    }
}
```

- **Optimization:**
  - **Dataflow**
  - **Loop pipeline**
  - **Array partitioning**
  - **Loop unroll**

Loop unroll factors are corresponding to number of iterations, determined usually by size of array. For example, loops in the linear filter I and linear filter R with unroll factor as 4 give best efficiency since amount of iterations is 64, size of ‘b’. For loop in the beginning where the *axis_t* inputs are read, loop factor as 20 is suitable since size of ‘xI’ is 1000.

- ## Performance and resource utilization:



⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.393	1.25

⊟ **Latency (clock cycles)**

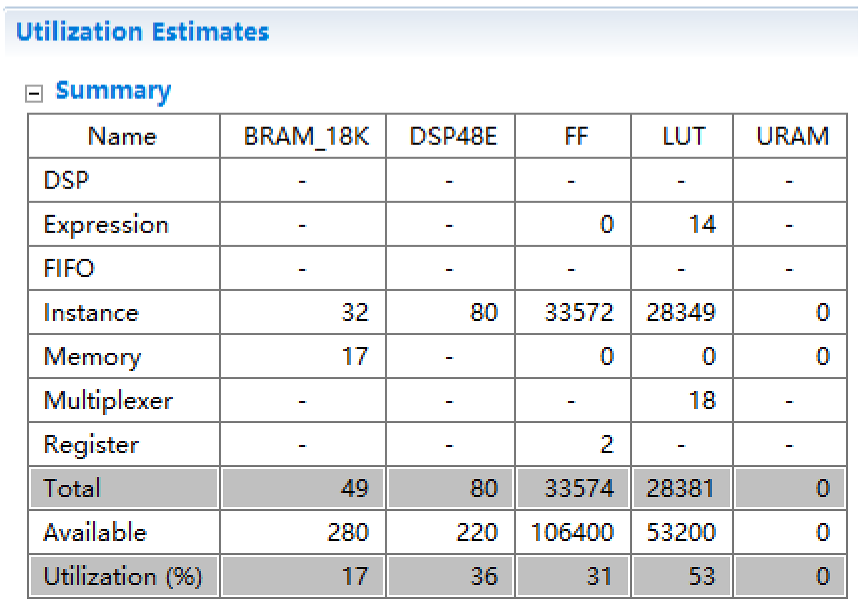
⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
19879	19879	16313	16313	dataflow

⊟ **Detail**

⊞ **Instance**

⊞ **Loop**



Utilization Estimates

⊟ **Summary**

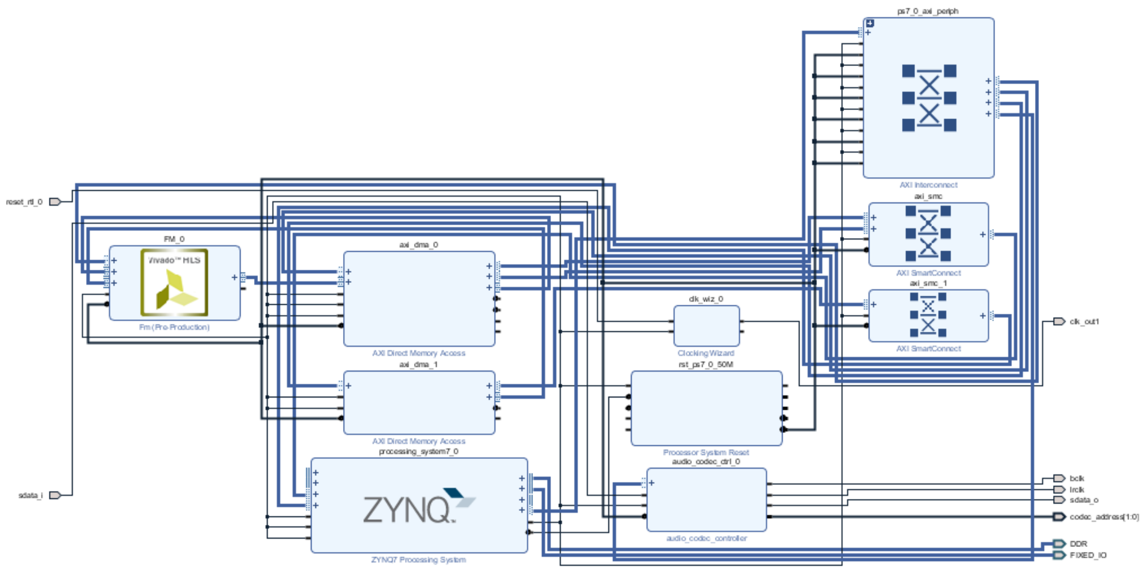
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	14	-
FIFO	-	-	-	-	-
Instance	32	80	33572	28349	0
Memory	17	-	0	0	0
Multiplexer	-	-	-	18	-
Register	-	-	2	-	-
Total	49	80	33574	28381	0
Available	280	220	106400	53200	0
Utilization (%)	17	36	31	53	0

Throughput = 1000 / 19879 / 9.393 = 5.36 kHz

Latency = 19879 * 2400 = 0.48s < 1s

- ## Adopting HP audio port

We can employ the FM demodulator in pynq, and after that use audio jack to play the processed signal. The connection between PL and audio jack is based on the ADAU1761 codec controller on pynq. The IP of the codec controller in base Overlay can be found at https://github.com/Xilinx/PYNQ. After downloading and adding the repo in vivado project, the IP named 'audio_codec_ctrl' will be available. Note for Vivado 2019, pynq 2.5 is needed. This IP can be connected with my multi DMA design plus a clock wizard. Next, by setting up the modules following the downloaded base overlay, I got the block design in the figure:



Rough block design

In notebook, theoretically, loading the 'z_out' to internal audio buffer and then playing it can make the headphone work. However, due to the limitation of time and available resources, I'm not be able to finish the demo. A problem is that in the official audio introduction notebook 'audio_playback.ipynb', the user uses an instances in the base overlay called ‘audio’. Specifically, the notebook assigns ‘base.audio’ to the controller,

```
  from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
pAudio = base.audio
```

while I cannot find any IP or hierarchy called ‘audio’ in the base overlay block design. In the future, the further works on my project is solving this problem, adding more IPs or modules if needed, and testing the demodulator on headphone.