CSE237C Final Project Report

Henrik Larsson Hestnes

December 2021

Contents

1 Introduction

For the final project in this course in this course, it was implemented an FM-demodulator in Vitis HLS 2021.1, and then generated a bitstream of the code using Vivado 2021.1. This bitstream was then loaded as an overlay onto a PYNQ-Z2 board. This PYNQ-Z2 board had an RTL2832 connected to it, which was used to sample RF-signals. The board then hardware-accelerated the FM-demodulation by using the bitstream created in Vivado on the FPGA, to make it possible to demodulate this signal in real-time(demodulate 1 second of RF-signals in under 1 second), which was the main objective of this project.

2 HLS implementation

Briefly explained, an FM-demodulation consists of the following processes on the raw complex RF-signal:

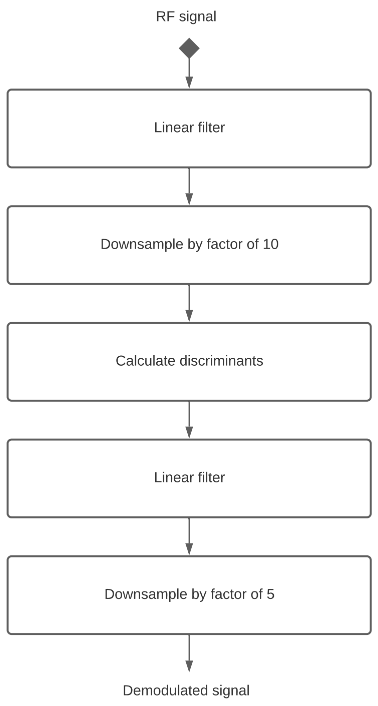

Figure 1: Dataflow of an FM-demodulator

2.1 Linear filter

To make the process depicted in fig. 1 as a C++ code, all of the 5 blocks were implemented as stand-alone C++ functions. The linear filters were implemented as direct II transposed structure FIR low-pass filters on the following form

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] \quad = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{1}$$

The number of previous input states(x) used to calculate the output y were set to 64, and thus the number of taps in the FIR-filter were also set to 64. The values for the taps were found using the $scipy.signal.firwin$ library in Python[1]. The cut-off frequency of the first filter was set to 200 KHz, while the second filter had a cutoff frequency of 12 KHz. The implemented code was inspired by the $scipy.signal.lfilter$ code, which can be found here[1]. The biggest difference from this implementation is that every a-coefficient is set explicitly to 0 except a[0] which is set to 1 to reflect eq. (1). The whole a-array is therefore removed from the parameter list of the function.

2.2 Downsample

The downsample functions were simply implemented as for-loops passing on every N'th element of the original datasample, discarding the rest.

2.3 Discriminant

The purpose of the discriminant is to calculate the derivative of the modulated phase, which provides the frequency of the signal. This turns the complex samples into real numbers, and is calculated by the following formula

$$\phi'(t) = \frac{S_R(t)\left(S_I(t) - S_I(t-1)\right) - \left(S_R(t) - S_R(t-1)\right) S_I(t)}{S_R^2(t) + S_I^2(t)} \tag{2}$$

where S_R denotes the real part of a sample and S_I denotes the imaginary part. The implementation of this function were split into two separate functions, one which calculates the derivative and one which calculates the discriminant. This facilitates for dataflow between the two functions.

2.4 Testbench

Why a testbench is of the essence for a project like this is obvious. Since we already have every one of these functions well working in Python from the "Project: FM Demodulator" in this course, this code was utilized to find a "correct" output of every function with a given input[2]. This input-output relationship was again utilized to find the RMS-deviation of the implemented code against the given Python code. At first, every single function was tested separately to make sure they met the accuracy requirements, before the functions were put together in a main function called FM_demodulator, which also was tested against the "correct" I/O-relationship.

2.5 Optimization

The first optimization done was to change all inputs in every function to a FIFO stream using the hls::stream datatype from the hls_stream.h library. Every input sample of the demodulation gets passed in the same order throughout the whole demodulation phase, and only depends on the 64 previous samples in the linear filters and the prevoius sample in the discriminant function. The whole design is therefore very suited for streams. Since Vitis 2021.1 already takes care of the loop pipelining in a good manner, these opimizations have not explored the effect different manually set initiation intervals would have on the latency.

2.5.1 Dataflow

The second optimization done was to apply dataflow in the "FM_demodulaton" function. This function only consists of other functions, and is therefore highly suited for dataflow. Dataflow was also added to the discriminator function, since this function also only consists of two other sub-functions. After these optimizations, the synthesis gave the following results on a sample size of 24000 samples, which is 1/100 of the number of samples per second

	Latency	**BRAM**	**DSP**	**FF**	**LUT**
Total	15.0 ms	163	45	23561	18563
Utilization on PYNQ-Z2	-	58%	20%	22%	34%

It is important to remember that this performance is only on 1/100 of the amount of samples per second. This does not mean that this latency necessarily will get increased by a factor of 100 as long as the interface between the processing system and the programmable is implemented in a good way, preferably by streaming both the inputs and output. Even if this interface is implemented in the most optimal manner, one should still expect a quite significant increase in total latency due to resource limitations on the FPGA and that this interface will consume time. It is also important to remember that the time this interface uses is not included in this latency, an thus will contribute to another increase in the total latency. It can therefore not be expected that this implementation already will manage to perform real-time demodulation of the RF-signals.

2.5.2 Array partitioning and loop unrolling

As one of the undoubted bottlenecks of this implementation is the first linear low-pass FIR filter, this optimization was mainly focused on this filter-function. In fact, this function stands for a total of 100% of the latency of the entire implementation according to the synthesis report, which in itself is quite interesting, and really emphasize why this optimization should be focused here. The optimal solution with regards to latency would of course be to completely partition all the arrays in the function, and then unroll all of it. This is however not even close to feasible due to resource limitations on the FPGA, and therefore this has to be optimized more carefully. By looking closer at the code of this function, it is clear to se that the two inner for loops, the shift register loop and the MAC loop contributes to a lot of these latency cycles. By partitioning both the shift registers in the first linear filter in a cyclic manner by a factor of 16, and unrolling this loop by a factor of 16, it is actually the second linear filter which becomes the bottleneck. By also partitioning the shift register of the second linear filter cyclic with a factor of two, and then unrolling the shift register loop by a factor of 2, we get the following performance

	Latency	**BRAM**	**DSP**	**FF**	**LUT**
Total	1.37 ms	161	195	35000	39120
Utilization on PYNQ-Z2	-	57%	88%	32%	73%

This is really starting to look like something which could manage to demodulate the RF-signals in real-time, at least if the interface between the PS and PL is implemented as a streaming interface. This latency is calculated on 1/100 of the total number of samples per second. If you multiply the latency by 100 you get a total latency of 0.137 seconds, which should be more than good enough as long as the interface is implemented right. However, after trying to implement this as an axi-streaming interface, making sure the TLAST-port was set correctly and generating the bitstream with two DMAs, this never worked as it should on the actual PYNQ-Z2 board. For some reason the execution always stalled on the last receive-port. After spending quite too many hours trying to debug this without any results, the interface was changed to an axi-burst interface. When this was run on the FPGA, the code managed to demodulate one second of RF-samples in approximately 1.2 seconds, which is too slow for the real-time requirements. It was therefore time to look at other options regarding optimization.

2.5.3 Change number of samples per function call

Another big bottleneck of this implementation is the communication interface between the PS and PL. A good idea is therefore to make each function call handle more samples, which would decrease the amount of times the PS has to call the PL. Changing the number of samples handled per function call from 24000 to 32000 yielded the following performance

	Latency	**BRAM**	**DSP**	**FF**	**LUT**
Total	1.83 ms	162	195	35002	39120
Utilization on PYNQ-Z2	-	57%	88%	32%	73%

This is barely different with regards to resource usage. The latency of the function goes up as expected, but considering that this function only has to be run 75 times to demodulate 1 second of RF-samples, the total latency of 1 second is still on 0.137. When trying to run this implementation on the FPGA, the resulting demodulation time was still over real-time performance, but it was reduced from around 1.2 seconds to around 1.05, which is really close to the real-time requirements.

2.5.4 Reducing number of taps in filter

As there are no more obvious optimizations that can be done within the resources available at the PYNQ-Z2 board, this optimization is to reduce the number of taps in the first FIR-filter, which might be on the expense of the accuracy of the filter and thus compromise the sound quality. However, the overall goal of this project was to demodulate the signals within the real-time requirement, so this optimization was seen on as an improvement as long as the sound quality did not get noticeably worse. By reducing the number of taps in the first linear filter from 64 to 54, and thus also reducing the size of the shift register correspondingly, we get the opportunity to do a cyclic array partitioning of the shift register with a factor of 18, which is 1/3 of the size of the total shift register. By unrolling the shift register loop with the same factor, we got the following results

	Latency	**BRAM**	**DSP**	**FF**	**LUT**
Total	1.52 ms	162	215	36145	42018
Utilization on PYNQ-Z2	-	57%	97%	33%	78%

By multiplying the latency of this function with 75, which is as many times as this function has to be called to demodulate a whole second of samples, this yields a total latency of 0.114 seconds, which is quite a significant decrease from the previous implementations. The resource usage is also just within the bounds of the FPGA. By making a bitstream out of this code and running it on the device, the total demodulation time became around 0.96 for one second of samples every time the code was run, which is just within the bounds of the real-time requirements, and which was the goal of this project. There is also no noticeable difference in the sound generated by this code versus using the $scikit - dsp - comm$ library in Python, at least as far as the ears of the author of this paper is concerned.

3 PYNQ implementation

The PYNQ implementation of this project is more straightforward, and can be found in the PYNQ_code folder.This code is also heavily inspired by the code handed out in the "Project: FM Demodulator" in this course[2]. The biggest difference is that it is a more compact version, and the length of the sample-array which gets sent to the IP-core every iteration is changed which also changes the number of iterations in the for-loop. The first part of this code samples 3 seconds of RF-signals, before demodulating the signal using the $scikit - dsp - comm$ library in Python to see how long time the process takes on optimized code only running on the CPU[3]. This code used 15-16 seconds to demodulate 3 seconds of RF-samples, which of course is not even close to real-time. After this, the bitstream is imported as an overlay on the board, and the same 3 seconds of samples are demodulated using the hardware-accelerated FPGA code. This demodulation takes around 2.92 seconds, which is just within the real-time requirements, which was the overall objective of this project. A demo of this code can be seen here.

4 Conclusion

This project went out on making an FM-demodulator work in real-time on a PYNQ-Z2 board. As the first linear filter is the undoubted bottleneck of this demodulation, applying optimizations such ass array partitioning and loop unrolling to this part of the code is necessary to make it work in real-time. Streaming inputs and outputs between the functions internally is also a good approach. Preferably the interface between the PS and PL should be implemented as an axi-streaming interface, but it is also possible to make it work in real-time with an axi-burst interface by making a small compromise on the sound quality.

References

[1] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[2] Ryan Kastner. Project: FM Demodulator. https://pp4fpgas.readthedocs.io/en/latest/project7.html, 2021.

[3] Mark Wickert and Chiranth Siddappa. scikit-dsp-comm. https://readthedocs.org/projects/scikit-dsp-comm/downloads/pdf/latest/, 2021.