Implementation of a bank of correlators in FPGA

Manideep Dunna
Department of ECE
mdunna@eng.ucsd.edu

Sanjeev Anthia Ganesh
Department of ECE
santhiag@eng.ucsd.edu

1 Problem Definition

We are working on a research project in our lab that uses radar to detect and identify specific objects by tagging them using custom made backscatter tags. The custom tags are designed to modulate the incident radar signal with a characteristic signature and reflect the RF signals. We have a working implementation of the algorithm in MATLAB to detect the tag signatures. Our goal in this course project is to facilitate faster processing of radar's ADC samples on an FPGA in an effort to perform real-time identification of these tags. For this purpose, we want to explore the ways to parallelize our algorithm.

First, we will start with a brief description of the radar and the reflected RF signals. We use Frequency modulated continuous wave(FMCW) radars [1], where a radar sends a chirp signal. The RF signal reflections that arrive back to the radar after bouncing off the objects in the surroundings are down-converted and are sampled by the ADC. In our work, custom tags are used to modulate these reflected signals in a specific manner to identify if the objects of interest are present in the vicinity of the radar. The down-converted signals are of the form $s(t)e^{j2\pi f_0 t} + n(t)$ where f_0 is a known constant and t indicates the time, $n(t)$ is the radar reflections from all the objects and $s(t)$ is the modulation pattern. The modulation pattern is chosen to be an unknown sequence of the set of Gold code sequences [2] which have favourable correlation property. Now, our main goal is to identify which one of the gold sequence $s(t)$ is used as the modulation pattern by analyzing the ADC samples.

Now, we describe the detection algorithm. The main idea here is to find the unknown sequence $s(t)$ is by computing the cross-correlation of all the ADC samples with every sequence in the set of gold codes. The sequence that results in maximum cross-correlation value is declared as the unknown sequence $s(t)$. This technique is commonly used in code division multiple access (CDMA) wireless systems to multiplex multiplex users. In our hardware implementation, we obtain radar samples at 1Msps rate implying the 1MHz bandwidth of the digital signal. According to our design constraints, the modulation pattern has a bandwidth of only 500KHz. The Gold code sequence is chosen to be of length 31 bits, for which the cardinality of the Gold code set is 33. So, we have to perform cross-correlations with 33 different sequences.

2 Proposed Approach:

Keeping the above facts in mind, we want to down sample the input samples by a factor of 2 before performing the cross-correlations. To ensure no aliasing during down-sampling operation, we perform low pass filter to preserve only the spectrum occupied by the modulation pattern and then down sample the digital signal. This is a multi-rate signal processing problem and there is an efficient method called a Polyphase filter implementation to reduce the computational load and save resources. We will implement this optimized architecture in the first step of the FPGA algorithm.

In the next step, we have a bank of 33 correlators that take input samples at 500Ksps from the previous polyphase filter stage. Each correlator is entirely independent of each other and we can exploit the FPGA resources to do a cross correlation with each sequence in a parallel manner without any data dependency. This will be second optimization we want to perform in our parallel implementation.

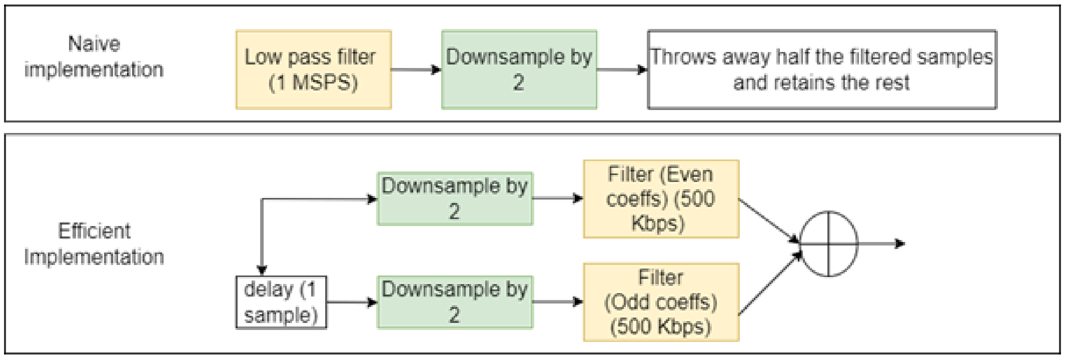

Figure 1: Naive implementation vs Polyphase implementation of the filter

This approach will reduce the algorithm latency by 30 times if the FPGA resources allow us to completely execute all these cross-correlations in parallel.

In section 4, we discuss about the filter designed and why we go for polyphase implementation. And in section 5, we discuss about the bank of correlators and the results we get from the various optimization performed on FPGA. In section 6, the end to end system implementation is discussed in detail. The demo is performed in the PYNQ board and the corresponding DMA stream implementation can be seen in the Git repository[3].

3 Filter design

The input signal to the FPGA is the received signal from the radar. The received signal from the radar contains a sequence of code that we wish to figure out. The received signal is upsampled. Therefore, our aim is to filter, downsample the signal and send the filtered signal to a bank of correlators.

The main purpose of a filter stage is to avoid aliasing after downsampling. So, this is why we need a filter stage before downsampling. The problem with having a filter stage before downsampling by 2 stage is that half the signal that has been filtered is thrown away while the remaining signal is sent ahead. This is problem in terms of computation because we add and multiply the weights with the samples only to see the samples being thrown away. One way to avoid unnecessary computation is if we know what is the signal that is thrown away before downsampling. If we could somehow avoid those computations, then we would be able to improve the latency of the operation as well.

$H(z) = a + bz^{-1} + cz^{-2} + dz^{-3}$, H(z) is the z-transform of the filter.
$S(z) = A + Bz^{-1} + Cz^{-2} + Dz^{-3}$, S(z) is the z-transform of the signal.
$H(z)S(z) = aA + (aB + Ab)z^{-1} + (bB + aC + Ac)z^{-2} + (bC + cB + Ad + aD)z^{-3} + ...$, is the output of the filter

If we see from the equation above, during downsampling, we remove the samples with z^{-1}, z^{-3}. Basically, they are the odd components. So, to avoid computations of these odd components, we notice something here. Whenever the even index of the signal is multiplied with the even index of the filter as well as whenever the odd index of the signal is multipled with the odd index of the filter, we try to retain those coefficients. So, for example, the coefficient aA is obtained from multiplying z^0 of the filter (a) with z^0 of the signal A. Similarly, we obtain other coefficients of z^{-2}, z^{-4}, .. by multiplying even coefficients or odd coefficients of the signal with even and odd coefficients of the filter respectively.

Therefore, whenever the odd coeffcients of the signal are multiplied with the even coefficients of the filter and vice versa, we see that these samples are thrown away. For example, $az^0 * Bz^{-1}$ and $Az^{-1} * bz^0$ is multiplying 0th index of the signal (even) with the first index of the filter (odd) and vice versa. This corresponds to z^{-1} of the output of the filter which is thrown away.

By avoiding such operations, we try to achieve a version known as the Polyphase implementation of the filter. Block diagram in figure 1 shows how we can split one full-rate filter into two half-rate filters and operate them simultaneously.

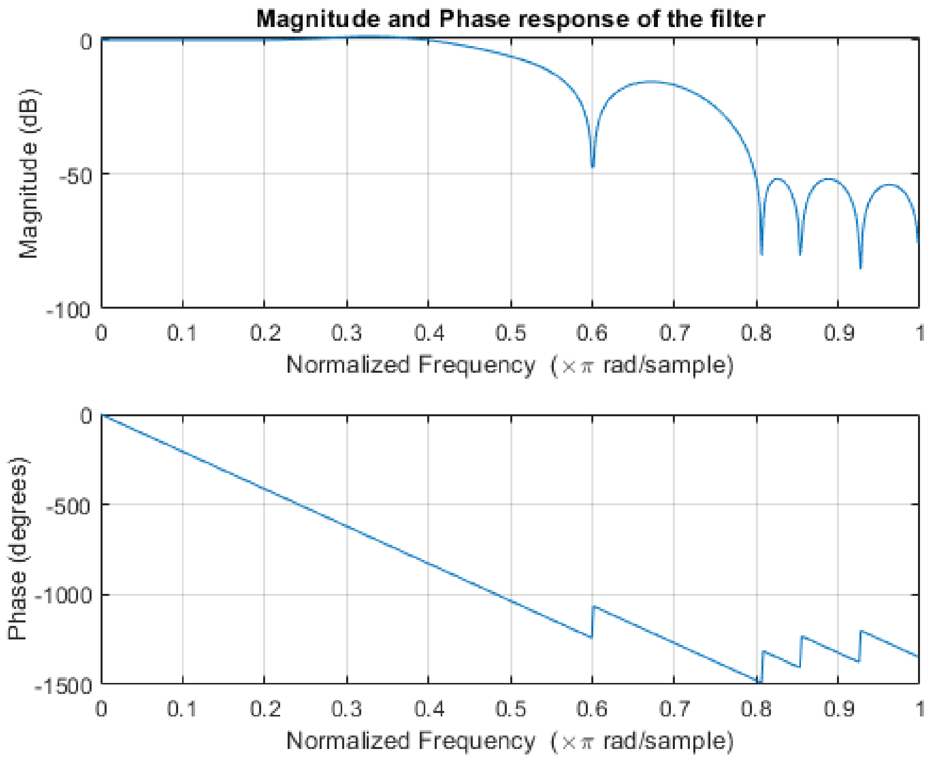

Figure 2: Magnitude and Phase response of the low pass filter

3.1 Choice of filter:

The Polyphase implementation works only for FIR filters and the filter we designed was a low-pass filter derived from Parks–McClellan algorithm. This is a 24 tap linear phase filter whose response is shown in figure 2. The fft response of the input signal and the filtered input signal is shown in figure 3.

3.2 Results from the Optimizations in FPGA

For this implementation, we consider a block of 340 input samples that are generated at 1MSPS rate. We have set the FPGA estimated clock rate to 100MHz. FPGA implementation of the naive version and the polyphase version of the filter gave the results we expected. By allowing two half rate filters to run in parallel, we reduced the total number of computations as well as the latency to perform the filter operation. Table 1 shows the resource utilization and latency of the three filter optimizations.

	BRAM	**DSP**	**FF**	**LUT**	**Latency (# of cycles)**
Baseline	1	7	3067	2692	8130
Polyphase baseline	0	15	3949	3925	1614
Polyphase optimized	0	50	7039	4519	189

Table 1: Comparison of Filter & Downsampler Implementation with and without polyphase decomposition

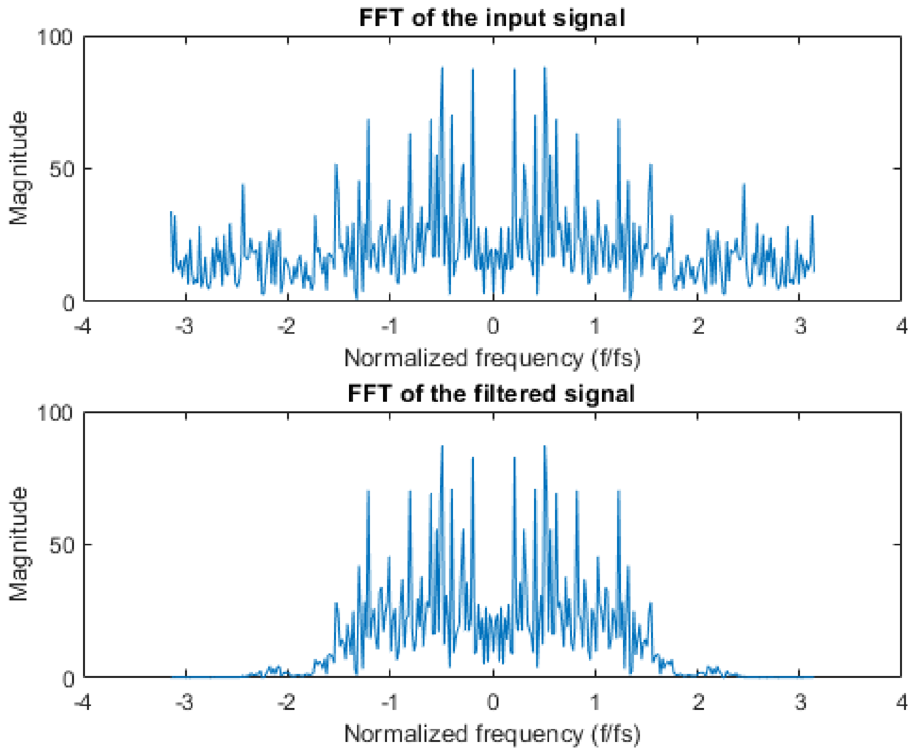

Figure 3: Frequency spectra of the raw input signal and the filtered input signal

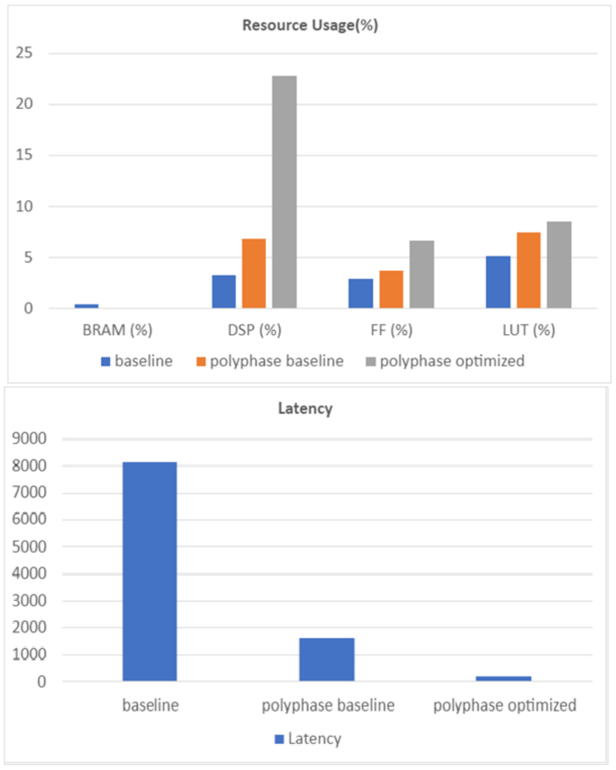

Figure 4: Top: Resource utilization from the filter design. Bottom: Latency from different optimizations

For the baseline, we implemented 24 tap FIR filter & down sampler with the input data and filter coefficients represented as floats. This implementation takes 8130 cycles = 81 us to process the block of 340 samples. To show the effectiveness of the the poly-phase decomposition, we implemented polyphase baseline where we did not perform any optimization other than using 2 parallel filters and observed the latency and resource usage. As expected the DSP usage doubled due to the usage of 2 parallel filters. The Flip Flop and LUT usage remained almost the same as baseline. The latency here decreased approximately by a factor of 4. Factor of 4 improvement is expected because, 2x improvement should come from the parallel filter implementation(reduction in filter taps) and another 2x improvement comes from the fact that the input data is also split into two blocks that are independently processed by two filters.

Next, we made optimizations in terms of data bitwidth(fixed point representation). The input data samples and the filter coefficients lie in the range -1 to 1 with 9 decimal point accuracy. Since we need 10 bits to achieve a 3 decimal point accuracy, we used 30 bits to represent the fractional part of the data. Then we unrolled the shift register and the accumulation loop completely. This increased the DSP usage by approximately 3 times. Considering the binary adder tree structure, atleast a factor of 3 increase is expected for a full unroll of 12 additions. To take advantage of full unrolling, the memory partitioning is set to complete resulting in the increase of the LUT and FF usage. Finally, the latency decreased to 189 cycles = 1.9 us with all the optimizations.

4 Bank of correlators

The filtered signal from the previous stage is given as an input to the bank of correlators. There are 33 correlators with 31 taps each in our design. Correlation can be understood as a FIR filtering operation with the coefficients reversed in order. So, a correlator can be implemented as an FIR filter here. We correlate the filtered input signal with the all these correlators and look at the peaks. We can take advantage of parallel implementation of the FPGA and process all the correlators simultaneously and find out the peaks as shown in figure 5. In the baseline implementation, we ran each correlator sequentially and find out the peaks. In the baseline, the correlator coefficients, input and output samples are represented as floats. Table 2 contains the resource usage and latency for each of the optimizations we have done.

In the baseline, the correlator coefficients are implemented as floats but the correlator coefficients are binary valued numbers taking either +1 or -1. So, we used 2 bit signed integer to represent the correlator coefficients. This data-width optimization reduced the massive space requirement as shown in table 2. Another observation here is that, all the correlators use the same input and so it is enough to maintain the same shift register for all the correlators. This shift register is partitioned completely for full access by all the correlators simultaneously. Also, since the correlator-coefficients are just +/-1, accumulation operation can be performed with just additions and subtractions which can be directly implemented using LUTs and FFs without any DSP utilization at all. This way, the FF and LUT usage increased slightly compared to baseline but the latency decreased by 4 times.

Next, we focused on the nested loops where the outer loop runs over all the correlators and inner loop runs over the shift register and accumulator. We varied the loop order to see if we can pipeline the outer loop with a smaller initiation interval but contrary to our expectation, the latency has increased along with the resource usage. Finally, we observed that while finding the maximum peak in each correlator output, we were storing the result in the function output variable and accessing it in every iteration which is increasing the latency. So, we used a temporary variable array to store the intermediate results for the computation and finally assigning the output variable and it reduced the latency to 5600 cycles(56 us). We called this optimization "Function Interface Changes" in the table 2. Figure. 6 shows the comparison of the bank of correlator implementation among different optimizations.

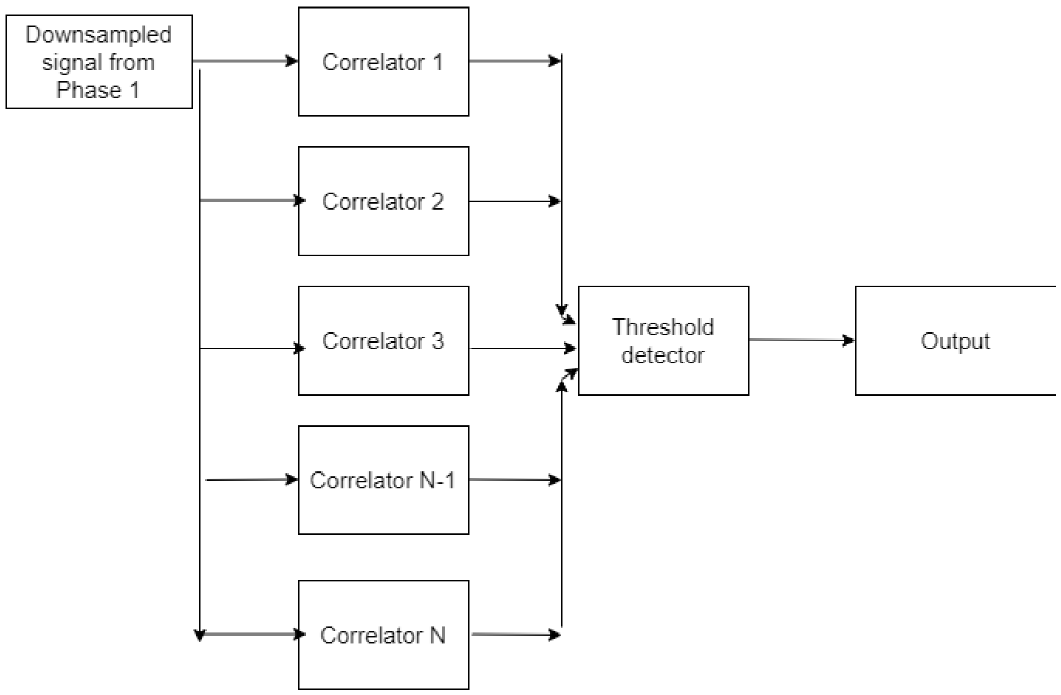

Figure 5: Implementation of the bank of correlators

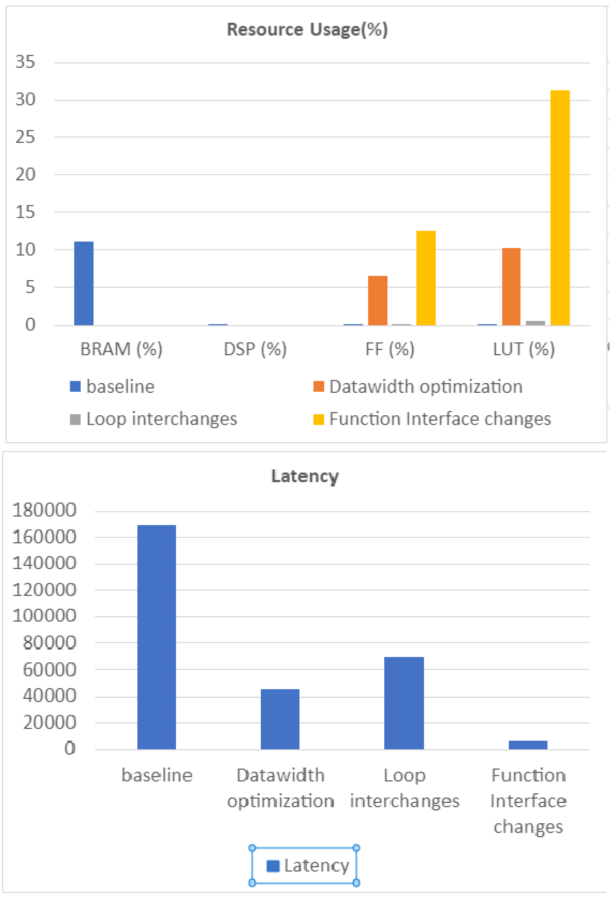

Figure 6: Top: Resource utilization from the Bank of correlators optimization. Bottom: Latency from different optimizations

	BRAM	DSP	FF	LUT	Latency (# of cycles)
Baseline	31	10	4681	3387	168499
Datawidth optimization	0	0	6951	5386	44898
Loop interchanges	0	0	14333	29003	69364
Function Interface changes	0	0	13263	16650	5673

Table 2: Comparison of correlator Resource usage for different optimizations

5 End to End system implementation

In the integration process, one can consider the filter and bank of correlators belong to separate IPs and after the execution of filter, we could pass the data to the correlators. This framework doesn't pipeline the two operations. So, for example, in this project, we have the length of the input signal as 340 samples. The output of the filter and downsampler contains 170 samples. In the baseline implementation, we see that after the filter stage processes the 170 samples from the input signal, the correlator takes those 170 samples as input and starts its operation. So, assuming the correlator and filter take 1 cycle per sample, we see that the baseline implementation takes 340 cycles on the whole.

So, the first step of optimization is we want the correlator to start right after the filter outputs 1 sample. This can be achieved by two different methods - Using pipelining as well as utilizing the dataflow implementation. By this method, we see the correlator and the filter perform the operations simultaneously.

From the resource utilization and latency information, we see that pipelining completed the operation around 6000 cycles, but the resources utilized by the pipelining operation were high compared to the dataflow implementation. But these optimizations fared well compared to the baseline implementation which consumed over 15000 cycles to complete the operation.

End to End system	BRAM	DSP	FF	LUT	Latency
Baseline	5	92	20348	11388	15256
Optimization_1 (with dataflow)	2	46	17442	9563	9276
Optimization_2 (with pipelining)	7	58	27216	49103	6439

Table 3: Comparison of End to End system Resource usage for different optimizations

6 Demo:

For the demo[3], we implemented a streaming interface to send the input data and read the output data from the IP using a DMA on PYNQ board. It is enough to use just a single DMA with two channels since all the processing is happening on just real samples. The input sent to the IP is a block of 340 real samples as a float array. The output is an array of 33 element float array corresponding to the peak values at each of the correlators. For the streaming interface, we used ap-axiu struct to encapsulate the array data that has to be sent and received from the IP. The Jupyter notebook compares the obtained output array from the FPGA with the ground truth and plots the errors.

References

[1] "FMCW Radar tutorial." `https://www.radartutorial.eu/02.basics/Frequency%20Modulated%20Continuous%20Wave%20Radar.en.html`.

[2] E. H. Dinan and B. Jabbari, "Spreading codes for direct sequence cdma and wideband cdma cellular networks," *IEEE communications magazine*, vol. 36, no. 9, pp. 48–54, 1998.

[3] "Project Repository." `https://github.com/Sanjeev2697UCSD/CSE237C_FinalProject`.