Designing Filters for High Performance
Volker Mauer, Principal Design Engineer, Altera

Introduction
Stratix® 10 devices contain a number of innovations that deliver breakthrough DSP performance. This paper describes a complex-valued FIR filter, implemented in VHDL, with a 1 GHz target maximum clock frequency ($f_{\text{MAX}}$). The discussion begins with the theoretical analysis that leads to an efficient implementation, describes implementation constraints, and discusses how the design efficiently uses Stratix 10 device features such as the DSP block. The FIR filter code is written to yield optimized results using Altera’s Quartus® Prime design environment. To conclude, the paper discusses the results and points to further optimizations for efficient filters.

Theoretical Analysis
Before implementing the design, we performed theoretical analysis to derive the minimum number of multipliers and adders to implement the required structure. Generally, filters can be described mathematically as convolution, performing the equation:

$$y_t = \sum_{n=0}^{k-1} x_{t-n} \cdot c_n$$

The filter used for this investigation is a 64-tap filter with 16 bit wide complex data and coefficients. There is no coefficient symmetry, and the coefficients are fixed. The input and output sample rates are 125 MHz.

For real valued samples and coefficients and 64 coefficients ($k$), this calculation requires 64 multiplications and 63 additions. For complex coefficients and samples, this equation still holds. The equation’s multiplication is complex valued multiplication, which can be performed as 4 real valued multiplications using the equation:

$$(x_r + jx_i) \cdot (c_r + jc_i) = x_r \cdot c_r - (x_i \cdot c_i) + j(x_r \cdot c_i + x_i \cdot c_r)$$

This multiplication requires 4 multipliers, one adder, and one subtractor. The entire 64-tap filter requires 256 multipliers, 64 adders, and 64 subtractors to implement the 64 taps, and 126 adders that sum the results of the individual taps.
However, there is another way to describe a complex multiplication, which is called a canonical implementation:

**Equation 3.**

\[(x_r + jx_i) \cdot (c_r + jc_i) = (c_r - c_i) \cdot x_r + (x_r - x_i) \cdot c_i + j((c_r + c_i) \cdot x_i + (x_r - x_i) \cdot c_i)\]

Note that the term \((x_r - x_i) \cdot c_i\) is used twice in this formula. That means we only need to implement it once, and we can reuse the result. This equation allows a complex multiplication using 3 real multipliers 2 subtractors and 3 adders. The entire filter with 64 coefficients now requires 192 multipliers, 128 subtractors, and 192 adders for the 64 taps, and 126 adders to sum the individual taps. Given that multipliers are larger and more power hungry than adders, this method improves the overall efficiency.

We can achieve additional improvement by considering the complex filter in its entirety. Starting with equation 1, we can write the entire filter as:

**Equation 4.**

\[y_t = \sum_{n=0}^{k-1} x_{r,t-n} \cdot C_{r,n} - \sum_{n=0}^{k-1} x_{i,t-n} \cdot C_{i,n} + j\left(\sum_{n=0}^{k-1} x_{r,t-n} \cdot C_{i,n} + \sum_{n=0}^{k-1} x_{i,t-n} \cdot C_{r,n}\right)\]

Effectively, we have replaced the complex filter with 4 real filters, one addition, and one subtraction. Our 64-tap complex filter needs 256 multipliers and 252 adders for the 4 real FIR filters, and one adder and one subtractor to combine the results of the filters.

![Figure 1. Complex FIR Filter Built with Four Real Filters](image)

We can also derive an alternative structure based on the canonical form:

**Equation 5.**

\[y_t = \sum_{n=0}^{k-1} x_{r,t-n} \cdot (C_{r,n} - C_{i,n}) - \sum_{n=0}^{k-1} (x_{r,t-n} - x_{i,t-n}) \cdot C_{i,n} + \left\{\sum_{n=0}^{k-1} x_{r,t-n} \cdot (C_{r,n} + C_{i,n})_{i,n} + \sum_{n=0}^{k-1} (x_{r,t-n} - x_{i,t-n}) \cdot C_{r,n}\right\}\]

Again, we can see that the last FIR filter is identical to the second, so we can implement the complex FIR filter as 3 real FIR filters, 2 subtractions, and 3 additions. If the number of taps, \(k\), is 1, equations 4 and 5 reduce to equations 2 and 3. The main advantage of this structure is that the 2 subtractors and 3 adders can be shared between the real filters, and therefore, are only required once per filter and not once.
per tap. Our 64-tap filter now requires 192 multiplications and 189 additions for the 3 real FIR filters, 2 subtractions and 1 addition at the input of the filters, and 2 additions to sum the outputs of the filters. We can achieve further savings for filters that have fixed coefficients, because we can pre-calculate the sum and difference of the real and imaginary components of the coefficients.

**Figure 2. Complex FIR Filter Built with Three Real FIR Filters**

![Complex FIR Filter Diagram](image)

**Table 1** summarizes the results of our analysis. The complex canonical FIR filter requires the fewest multiplications and the fewest additions/subtractions. Therefore, we choose that implementation.

**Table 1. Resources for complex FIR implementations**

<table>
<thead>
<tr>
<th>8-Tap Complex FIR</th>
<th>Multiplications</th>
<th>Additions</th>
<th>Subtractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR with complex taps</td>
<td>256</td>
<td>190</td>
<td>64</td>
</tr>
<tr>
<td>FIR with complex taps, canonical</td>
<td>192</td>
<td>318</td>
<td>128</td>
</tr>
<tr>
<td>Complex FIR built with 4 real FIR filters</td>
<td>256</td>
<td>253</td>
<td>1</td>
</tr>
<tr>
<td>Complex FIR, canonical, built with 3 real FIR filters</td>
<td>192</td>
<td>192</td>
<td>2</td>
</tr>
</tbody>
</table>

The actual number of multipliers needed is the number of multiplications (shown in **Table 1**) divided by 8, because our clock rate exceeds the sample rate by a factor of 8, allowing us to timeshare the multipliers. The same is true for the adders and subtractors.

So far we have not taken into account bit widths. Each addition or subtraction result is one bit wider than the largest of its inputs, and the width of a multiplication result requires the sum of the widths of its inputs. As a first approximation, we can assume that adders/subtractors after the multipliers need at least twice as many hardware resources as the adders/subtractors in front of the multipliers.

**Implementation**

**HyperFlex Architecture**

Altera’s Stratix 10 devices have been designed to provide 2X the performance of previous-generation, high-performance device families, and can achieve clock up to 1 GHz frequencies. Most high-performance FPGA designs are limited by routing. Furthermore, traditional performance enhancement techniques, such as pipelining and register retiming, cause significant routing congestion in conventional architectures. The Stratix 10 HyperFlex™ architecture distributes bypassable registers (called Hyper-Registers) throughout the routing fabric. These registers allow designers to pipeline data paths as deeply as desired, enabling designs to run at a
significantly higher clock frequency—up to the performance limit of hardened DSP blocks and embedded memory. Understanding How the New HyperFlex Architecture Enables Next-Generation High-Performance Systems (1) describes the HyperFlex architecture in more detail, however, one common performance enhancing technique is to insert additional pipeline stages using Hyper-Registers. Using Hyper-Registers to add pipelining, designers can achieve a major performance increase without using logic array registers, saving significant area compared to pipelining in conventional architectures. A relatively small increase in register count can result in a significant speed increase, often without increasing latency.

**Resets**

Stratix 10 Hyper-Registers do not have control signals (such as reset) because these signals would increase the size and power consumption of the entire FPGA. Instead, we use a synchronous reset strategy.

In filter chains, the first samples at the output are invalid anyway, and a value of ‘0’ is equally likely to be wrong as any other value. With this in mind, we can usually to remove the reset on the entire data path. Any wrong values flush out during the system startup. However, we cannot use the same assumption for the control domain: if we implement timesharing, or if the same filter time-interleaves different channels, it is essential that the channel counter and timeshare counters start at a specified time. The clock cycle that releases the reset is critical. We can implement this reset with a synchronous reset.

**Altera DSP Blocks**

Many DSP-centric devices are used in RF cards, in which the majority of multipliers are used in filter chains. The Altera® Arria® and Stratix series devices contain DSP blocks that are highly optimized for efficient filter implementations. We can think of DSP blocks as mini-filters that can be chained together to act as larger filters, rather than multipliers, and make use of the DSP block’s advanced features. Typically, it is possible to map adders and subtractors that precede or follow the multipliers into the DSP blocks. DSP blocks can be configured to use pre-adders, which are commonly used in symmetrical filters. In a systolic implementation, the addition across taps can also be mapped into the DSP blocks. The DSP block contains an accumulator as well, and an interesting extra feature is the ability to add a constant instead of the feedback value in the first cycle of an accumulation, which allows the addition of half an LSB (the most common rounding operation). An additional DSP block feature is that one multiplier input can be connected to the input of the DSP block, or it can be connected to an internal memory, which can store up to 8 predefined values. This feature is ideal for fixed coefficient filters, and absorbs more functionality from general logic into the highly power efficient DSP block. The FIR filter discussed in this paper uses several DSP block features.

**Internal Coefficient Storage**

Altera’s DSP blocks have a limited amount of ROM that can be connected to one of the multiplier inputs. This feature is extremely useful for fixed coefficient filters, where one input is connected to a single coefficient or in the case where the sample rate is a fraction of the clock rate, a small set of possible values. The ROM address can be provided from outside of the DSP block.
**Systolic Mode**

We used the systolic mode to move the summation over taps into the DSP blocks. In systolic mode, pipeline registers are put into the output cascade chain. This implementation requires matching delays in the sample chain. Figure 3 illustrates this functionality with the additional pipeline registers highlighted in red. The adders in the cascade chain are wide, because they operate on the multiplier outputs, therefore, we can achieve significant savings by mapping them into the DSP blocks. In particular, wide adders need extra pipelining to work at clock rates of 1 GHz, so absorbing these adders into the DSP block is highly desirable.

![Figure 3. Systolic FIR with Internal Coefficient Storage](image)

**Accumulator**

Because our input sample rate is 125 MHz, the device can perform the calculations over 8 clock cycles, which requires an accumulation of partial filter results. We use the internal accumulator of the last DSP block in the chain.

**Rounding and Truncation**

There are different methods to reduce FIR filter output precision. The easiest way is truncation. Based on data widths and knowledge of the coefficients, we can calculate the largest possible value at the filter output. This knowledge can be exploited to drop MSBs without negatively affecting the results. In this design, we truncated MSBs. The 44 bit DSP block output width exceeds the maximum value based on the data widths, which is 36 bits (multiplying 16 bit data and 16 bit coefficients results in 32 bits, summing 8 of these multiplications adds another 3 bits, and the addition/subtraction to produce the real and imaginary components adds another bit). We can further optimize by exploiting the fact that we know the coefficients. We did not use this optimization in this example, instead, we produced a filter that can be re-synthesized with different coefficients without re-aligning the outputs.
It is also possible to reduce LSBs. If the LSBs are below the noise level, this reduction comes without penalty and results in fewer in resources and less power in the last filter stages. More importantly, it also reduces resources and power in the stages that follow this filter. Imagine a filter chain containing 5 filters with 16 bits coefficients. If the first filter had 16 bits at the input, the output of the fifth filter would have more than 100 bits. Typically, the data width is controlled and kept similar throughout a filter chain, which requires rounding or truncation at the filter output. A popular rounding method is to add half the LSB. This operation adds a small DC bias and is a relatively inexpensive rounding method.

Altera’s DSP blocks allow half DSP adding internally. You can switch the accumulator feedback path to use a predefined constant instead of the accumulator feedback. In this design, this operation is performed during the first of the 8 accumulation cycles. The effect is that in the first cycle, the accumulator input is added with the preload constant, while in the remaining 7 cycles the accumulator input is added to the previous value of the accumulator. This technique has the advantage that no cycle is wasted to preload a value or to clear the accumulator, therefore, it is highly efficient.

Coding the Filter

The first step in writing the filter is to write the logic that goes into the DSP blocks. We can then verify the logic in isolation, and we can check that it really does only use DSP blocks and runs at 1 GHz.

VHDL Templates

The easiest way to begin is to start with existing templates. When you open a new VHDL file, right-click in the empty file and select Add template. The software provides a choice of templates. To exploit advanced features of the DSP block for Stratix 10 devices, the VHDL > Full Designs > Arithmetic > DSP Features > DSP features for 20-nm Device category provides the most comprehensive list of relevant templates.

Of the available templates, none fit our purpose exactly, but the M18x19systolic with Preadder and Coefficient template was relevant because it matched the output of the multipliers, and the M18x18_systolic with Input Cascade, Output Chaining, Accumulator, Double Accumulator and Preload Constant template had the features we needed after the multipliers. We selected the M18x19systolic with Preadder and Coefficient template and copied features from the other template into it to obtain the desired functionality. See Figure 3.
Next, we replaced the dummy multipliers in the template (which demonstrate how to handle odd numbers of multipliers) with real multipliers. We reduced the double accumulator to a single accumulator. On the input side, we assigned the preserve attribute to all input signals. This step is important because if inputs are not needed synthesis optimizes registers away, which can prevent the recognition of the DSP blocks and can cause other logic that should be inside the DSP blocks to be pulled out into general logic. In this particular case, we decided to leave the pre-adders in the DSP block so that the module could be re-used for symmetrical filters. Then we connected the b input of the pre-adder to 0 when instantiating the module. We also made the coefficients a generic that can be passed into the module, rather than hardcoding the coefficients into the filter module. This implementation allows the filter to be re-used with different coefficients.

**Sample Chain**

The next module is the sample chain. While Altera devices have small memory blocks, in this particular case it was best to build the sample chain entirely from registers. If we had used memory, we would have needed additional pipeline registers between the memories and DSP blocks to achieve high $f_{\text{MAX}}$. At the same time, short shift registers can use the HIPI registers, so it was very efficient to code the sample chain as a set of short, specialized shift registers, one per multiplier, that shift in a new sample every 8 clock cycles and rotate around the existing samples for the remaining clock cycles. We introduced an additional delay to match the DSP block systolic delay.
We then combined the sample chains and the DSP module into the module `fir_ns.vhd`, which contains a useful, standalone non-symmetrical FIR filter. At this level, unused inputs of the DSP module were tied off. The module also includes the control logic, i.e., a counter that counts from 0 to 7, and we added decoding logic to control the shift register functionality (load new value or rotate) and the accumulators (accumulate or preload constant). The counter values are used as addresses to the coefficient registers.

Because of the filter’s systolic nature, the coefficient addresses for the individual coefficient memories need to be delayed. However, instead, we chose to perform this delay more effectively by shifting the coefficients inside the memories.

**Pre- and Post-Adder Modules**

To move from a real valued filter to a complex filter, we created two more components:

- The pre-adder module takes the real and imaginary inputs and feeds them through a subtractor. It also feeds them through unchanged, but with matching pipeline delay, so that the inputs to the 3 filters are available.
- The post-adder module takes the output from the three filters and adds the output from filter 2 to the outputs or both filter 1 and filter 3, thereby generating the real and imaginary output of the complex filter.

**Testbench**

A design would not be complete without a testbench. For debugging filters, the most useful input pattern is an impulse. If the filter works correctly, the coefficients can be observed in order at the output of the filter. For a complex filter, an impulse on the real input, followed by an impulse on the imaginary input, is ideal to trace any control timing misalignments or data mismatches. A second useful debug pattern is setting the sample input to the maximum value and keeping it at this level for as many samples as the filter has taps. This scenario identifies any overflows that are not visible with the impulse response. Random patterns are useful for verification, but make it harder to predict what value is expected at any point within the design at any given time.
Figure 5 shows the output of a simulation with the coefficients programmed to \((1 - i) \cdot 1\) to 64. We can see an impulse on the real input, followed by an impulse on the imaginary input, and an impulse on both real and imaginary input, resulting in the expected outputs ramping through the coefficient values.

**Optimizing in the Quartus Prime Software**

Throughout the process, the design and its components were run through the Quartus Prime software. In particular, we compiled the component that uses DSP blocks standalone, to ensure ideal mapping onto the existing DSP blocks. The Quartus Prime software is optimized to take advantage of Hyper-Registers in the HyperFlex architecture—the compilation flow is aware of post place-and-route retiming using Hyper-Registers. When the entire design was run through the Quartus Prime software, the Hyper-Retimer module, a new module in the compilation flow, moves register locations to the optimum locations to optimize performance and timing margin. Furthermore, an additional compilation step, called Fast Forward Compile, provides additional guidance on where to insert additional pipeline stages or perform other code changes to optimize performance. This design flow is explained in *Using Quartus II Software to Maximize Performance in the HyperFlex Architecture.*

At the end of the Hyper-Retiming operation, the Quartus Prime software displays the design \(f_{\text{MAX}}\) but more importantly, it generates a “what-if” scenario analysis on performance that could be achieved with this design when following the performance Fast Forward Compile optimization recommendations. You can find these numbers in the Hyper-Retimer report, under **Clock \(f_{\text{MAX}}\)** Summary. The **Fast Forward Details for Clock Domain clk** section provides precise guidance about where additional pipelining is required and gives the source and destination points of the paths that would benefit from additional pipelining. Following the recommendations step by step moves the achieved \(f_{\text{MAX}}\) closer to the theoretically achievable \(f_{\text{MAX}}\).
Our design, at an intermediate development stage, achieved 728 MHz. Fast Forward Compile suggested adding registers to fully pipeline DSP blocks to increase the \( f_{\text{MAX}} \) to an estimated 987 MHz. It also identified three additional places where extra pipelining would increase the \( f_{\text{MAX}} \) to 1,183 MHz, 1,215 MHz, and 1,236 MHz. In the **Recommendations for Critical Chain**, the start and end points of the critical paths are identified to guide us through the process of pipelining the designs optimally. Stratix 10 performance tops out at 1 GHz. Therefore, performance listed above 1 GHz represents additional timing margin in the design, which is often desirable.

**Seed Sweep**

At a target \( f_{\text{MAX}} \) of 1 GHz, even a single additional routing hop can have a large effect on the achieved \( f_{\text{MAX}} \) and seed variations can be significant. It is advisable to run seed sweeps on the entire design from time to time. For individual components, the \( f_{\text{MAX}} \) variation across seeds is a good indication whether there is room for further improvement: if the results are low and all seeds result in similar performance, then there is probably a fundamental limit that we should address in the VHDL code. If several seeds pass the target \( f_{\text{MAX}} \), further improvements may be possible, and the underlying cause is probably placement related. It is worth looking for high fanout signals and connections between memories and DSP blocks, because they can be hard to place optimally. If the majority of seeds pass, there is no need for further optimization.
Altera’s Design Space Explorer (DSE) (see Figure 7) is a convenient tool to use to explore variations across seeds and optimization criteria. It allows you to specify what and how many seeds to use, and to investigate multiple exploration modes, such as **High Performance Effort**, **Aggressive Performance**, **High Power**, etc. If high performance is of most interest, **Aggressive Performance** gives the best results.

**Figure 7. Design Space Exploration**

Further recommendation can be found in *Methods for implementation of feedback loops in high speed FPGA applications.* (3)

**Results**

Figure 8 shows the results achieved across 10 seeds using the Quartus Prime software version 15.1.
This design achieves 1 GHz across all seeds, unless we selected the Aggressive Area optimization (this option created a more densely packed but lower performance design). The consistent high performance indicates that there is no benefit from further code optimizations.

Seed sweeps are useful when fine-tuning a design. The critical path in the best seed typically points to an area where the code can be improved. In contrast, a bad seed points out a badly placed component, and does not necessarily relate to a fundamental, code-based limitation. It is not necessary to run a full seed sweep across different optimization criteria every time. However, it is worth doing the sweep once to identify which optimization provides the best results. Later, using a reduced sweep for a single optimization technique and covering 3 to 5 seeds is a good compromise.

In this example, the resources needed varied slightly across seeds; common to all seeds are 12 DSP blocks. Because each DSP block contains two multipliers, this requirement is consistent with our expectation shown in Table 1 on page 3 (once these numbers are divided by 8 to reflect that we have timeshared the filter by 8, as the clock date exceeds the sample rate by a factor of 8).

**Design Variations and Future Work**

The filter structure developed in the theoretical analysis sections works well for data and coefficient widths for up to 17 bits. However, data and coefficient widths larger than 18 bits require larger multipliers. We can extend the filter up to 18 bit data and coefficients with similar resources, but the design requires some code changes. The main problem is that the pre-adders required for this structure increase the data width to 19 bits at the multiplier input. This bit width is possible using the built-in DSP block pre-adders, but we would need to change the code: for filters 1 and 3, the
coefficients are 19 bits wide. Therefore, we cannot use the internal coefficient storage. Instead, the real and imaginary coefficients must be connected to the DSP block pre-adder inputs, and the 18 bit samples must connect to the third input of the DSP block. For filter 2, the coefficients are 18 bits, so we can use the coefficient storage, but we must add the samples inside the DSP block. Therefore, instead of one pre-adder that feeds into a sample pipeline containing the difference between the real and imaginary samples, we need two sample pipelines (one for the real, one for imaginary part of the sample) that feed the DSP block pre-adder inputs. The filter 2 sample pipelines can be shared with filters 1 and 3, which is resource efficient, but may complicate the placement.

The subcomponent `fir_ns.vhd` can be used as a standalone non-symmetrical FIR filter. To build a symmetric FIR filter, we do not need to change the DSP block subcomponent because it already contains pre-adders, but we would need to extend the sample pipeline to contain the reverse samples, and modify the shift registers for the second half of the sample pipe to provide matching samples to the pre-adder inputs.

For programmable coefficients, we would remove the internal coefficient storage from the DSP block template, and add a `c` input. In this case, you should start with a new DSP block template.

Except for the accumulator, which is placed inside a DSP block, this design contains no feedback loops. Feedback loops need attention, because adding pipeline registers alters the loop’s functional behavior. If a design contains feedback loops, you can apply the techniques described in (3) to add pipelining without changing functionality.

## Conclusion

Stratix 10 DSP blocks support up to 1 GHz operation and take advantage of the device’s new HyperFlex capabilities. Our complex valued FIR filter design achieved a target $f_{\text{MAX}}$ of 1 GHz. By re-ordering the equations, we were able to optimize the algorithm to reduce the number of operations. Additionally, using the Stratix 10 DSP block features minimized the general logic required, and we were able to implement functions for a high target $f_{\text{MAX}}$. Using the Quartus Prime software design templates helped us code the filter quickly, and the software guided us through design optimization, step by step, to achieve our required target $f_{\text{MAX}}$ across a large number of random seeds.

## References

Document Revision History

Table 2 shows the revision history for this document.

Table 2. Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2015</td>
<td>1.0</td>
<td>Initial release</td>
</tr>
</tbody>
</table>