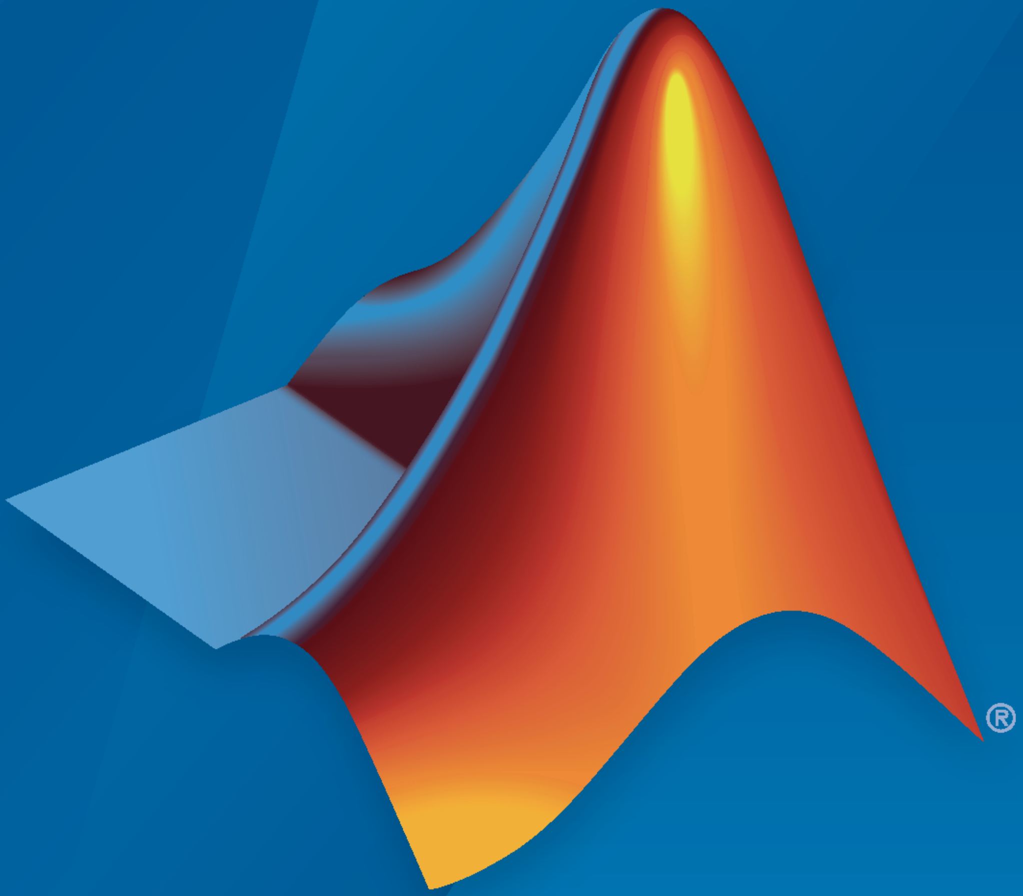
DSP HDL Toolbox™

Getting Started Guide



MATLAB® & SIMULINK®

R2022b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

DSP HDL Toolbox™ Getting Started Guide

Revision History

March 2022	Online only	New for Version 1.0 (Release 2022a)
September 2022	Online only	Revised for Version 1.1 (Release 2022b)

Contents

Product Overview

DSP HDL Toolbox Product Description

Design digital signal processing applications for FPGAs, ASICs, and SoCs

DSP HDL Toolbox™ provides pre-verified, hardware-ready Simulink® blocks and subsystems for developing signal processing applications such as wireless, radar, audio, and sensor processing. It includes templates for interfacing MATLAB® and Simulink as well as reference examples.

Using the toolbox you can model, explore, and simulate hardware architecture options for DSP algorithms. The IP blocks enable implementation for serial and parallel processing so you can explore the design space between resource usage, power, and gigasample-per-second (GSPS) throughput performance.

The toolbox algorithms let you generate readable, synthesizable code in VHDL® and Verilog® (with HDL Coder™). You can also generate SystemVerilog DPI verification components from designs that use these algorithms (with HDL Verifier™).

2

Concepts

Digital Signal Processing Design for FPGAs and ASICs

In this section...

Deploying algorithmic models to FPGA hardware makes it possible to do real time testing and verification. However, designing digital signal processing systems for hardware requires design tradeoffs between hardware resources and throughput. You can speed up hardware design and deployment by using HDL-optimized blocks that have hardware-suitable interfaces and architectures, application examples that implement common DSP algorithms, and automatic HDL code generation. You can also use hardware support packages to assist with deploying and verifying your design on real hardware.

MathWorks® HDL products, such as DSP HDL Toolbox, allow you to start with a mathematical model, such as MATLAB code from DSP System Toolbox™, and design a hardware implementation of that algorithm that is suitable for FPGAs and ASICs.

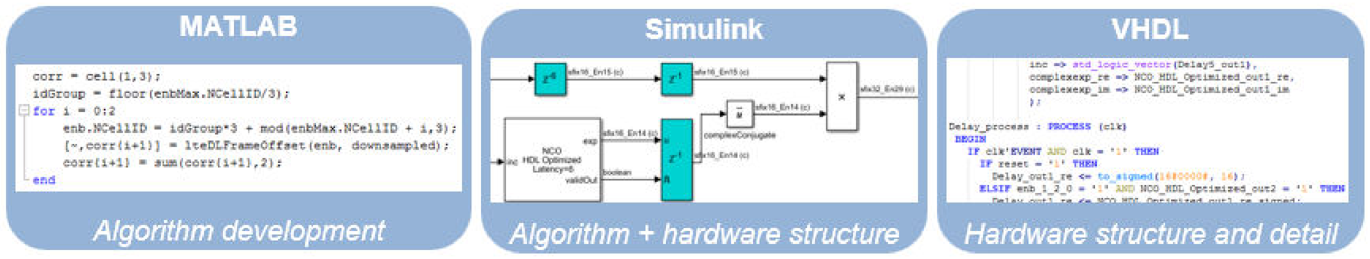
From Mathematical Algorithm to Hardware Implementation

DSP design often starts with algorithm development and testing using MATLAB functions. MATLAB code, which usually operates on matrices of floating-point data, is good for developing mathematical algorithms, manipulating large data sets, and visualizing data.

Hardware engineers typically receive a mathematical specification from an algorithm team, and reimplement the algorithm for hardware. Hardware designs require tradeoffs of resource usage for clock speed and overall throughput. Usually this tradeoff means operating on streaming data, and using some logic to control the storage and flow of data. Hardware engineers usually work in hardware description languages (HDLs), like VHDL and Verilog, that provide cycle-based modeling and parallelism.

To bridge this gap between mathematical algorithm and hardware implementation, use the MATLAB algorithm model as a starting point for hardware implementation. Make incremental changes to the design to make it suitable for hardware, and progress towards a Simulink model that you can use to automatically generate HDL code by using HDL Coder.

This diagram shows the design progression from mathematical algorithm in MATLAB, to hardware-compatible implementation in Simulink, and then the generated VHDL code.

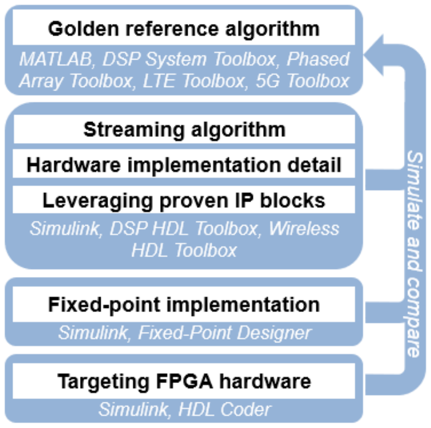


```
corr = cell(1,3);
idGroup = floor(enbMax.NCellID/3);
for i = 0:2
    enb.NCellID = idGroup*3 + mod(enbMax.NCellID + i,3);
    [~,corr(i+1)] = lteDLFrameOffset(enb, downsampled);
    corr(i+1) = sum(corr(i+1),2);
end
```

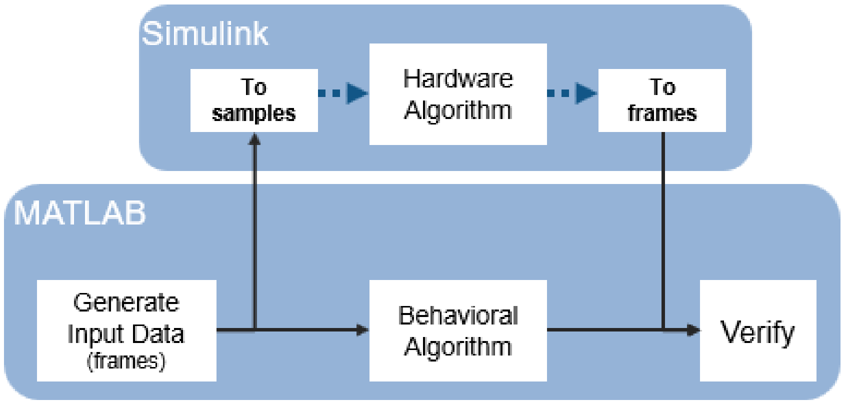
```
            inc => std_logic_vector(Delay5_out1),
            complexexp_re => NCO_HDL_Optimized_out1_re,
            complexexp_im => NCO_HDL_Optimized_out1_im
            );

Delay_process : PROCESS (clk)
 BEGIN
   IF clk'EVENT AND clk = '1' THEN
     IF reset = '1' THEN
       Delay_out1_re <= to_signed(16#0000#, 16);
     ELSIF enb_1_2_0 = '1' AND NCO_HDL_Optimized_out2 = '1' THEN
```

While both MATLAB and Simulink support automatic generation of HDL code, you must construct your design with hardware requirements in mind, and Simulink is better-suited for cycle-based modeling for hardware. It can represent parallel data paths and streaming data with control signals to manage the timing of the data stream. To aid in fixed-point type choices, it clearly visualizes data type propagation in the design. It also allows for easy pipelining of mathematical operations to improve maximum clock frequency in hardware.

While you create your hardware-ready design, use the MATLAB algorithm as a "golden reference" to verify that each version of the design still meets the mathematical requirements. The workflow shown in the diagram uses MATLAB and Simulink as collaboration and communication tools between the algorithm and hardware design teams.


Golden reference algorithm
MATLAB, DSP System Toolbox, Phased Array Toolbox, LTE Toolbox, 5G Toolbox
Streaming algorithm
Hardware implementation detail
Leveraging proven IP blocks
Simulink, DSP HDL Toolbox, Wireless HDL Toolbox
Fixed-point implementation
Simulink, Fixed-Point Designer
Targeting FPGA hardware
Simulink, HDL Coder
Simulate and compare

For instance, when designing digital filters, you can use DSP System Toolbox functions to create a golden reference in MATLAB. Then transition to Simulink and create a hardware-compatible implementation by using library blocks from DSP HDL Toolbox and Wireless HDL Toolbox™. You can reuse test and data generation infrastructure from MATLAB by importing data from MATLAB to your Simulink model and returning the output of the model to MATLAB to verify it against the "golden reference".



HDL-Optimized Blocks

Library blocks from DSP HDL Toolbox implement filters, FFTs, decimators, interpolators, and an NCO for use in digital signal processing systems. These blocks use a standard streaming data interface for hardware. This interface makes it easy to connect parts of the algorithm together, and includes control signals that manage the flow of data and mark frame boundaries. These blocks support automatic HDL code generation with HDL Coder. You can also use blocks from Communications Toolbox™ and DSP System Toolbox that support HDL code generation.

The blocks provide hardware-suitable architectures that optimize resource use, such as including adder and multiplier pipelining to fit well into FPGA DSP slices. They also support automatic and configurable fixed-point data types. Using predefined blocks also allows you to try different parameter configurations and architectures without changing the rest of the design.

For lists of blocks that support HDL code generation, see DSP HDL Toolbox Block List (HDL Code Generation), Wireless HDL Toolbox Block List (HDL Code Generation), Communications Toolbox Block List (HDL Code Generation), and DSP System Toolbox Block List (HDL Code Generation).

Generate HDL Code and Prototype on FPGA

DSP HDL Toolbox provides blocks that support HDL code generation. To generate HDL code from designs that use these blocks, you must have an HDL Coder license. HDL Coder produces device-independent code with signal names that correspond to the Simulink model. HDL Coder also provides a tool to drive the FPGA synthesis and targeting process, and enables you to generate scripts and test benches for use with third-party HDL simulators.

To assist with the setup and targeting of programmable logic on a prototype board, and to verify your wireless communications system design on hardware, download a hardware support package such as HDL Coder Support Package for Xilinx® RFSoC Devices.

See Also

Related Examples

- "Implement FFT Algorithm for FPGA" on page 3-2
- "Implement Digital Downconverter for FPGA"
- "Implement Digital Upconverter for FPGA"

More About

- "Hardware Control Signals"
- "High-Throughput HDL Algorithms"

Tutorials

Implement FFT Algorithm for FPGA

This example shows how to implement a hardware-targeted FFT by using DSP HDL Toolbox™ blocks.

Signal processing functions and blocks from DSP System Toolbox™ provide frame-based, floating-point algorithms and can serve as behavioral references for hardware designs. However, efficient hardware designs must use streaming data interfaces and fixed-point data types. Hardware designs also often require control signals such as *valid*, *reset*, and *backpressure*.

The blocks in DSP HDL Toolbox libraries provide hardware-optimized algorithms that model streaming data interfaces, hardware latency, and control signals in Simulink®. The blocks can process a number of samples in parallel to achieve high throughput such as gigasample-per-second (GSPS) rates. You can change the block parameters to explore different hardware implementations. These blocks support HDL code generation and deployment to FPGAs with HDL Coder™.

This example introduces the hardware-friendly streaming data interface and control signals used by DSP HDL Toolbox blocks, and shows how to use the two hardware architectures provided by the FFT block. Then, it shows how to generate HDL code for your design.

The DSP HDL Toolbox FFT block provides two architectures optimized for different use cases. You can set the **Architecture** parameter on the block to one of these options.

- `Streaming Radix 2^2` — Use this option for high throughput applications. The architecture achieves gigasamples per second (GSPS) when you use vector input.
- `Burst Radix 2` — Use this option for low area applications. The architecture uses only one complex butterfly.

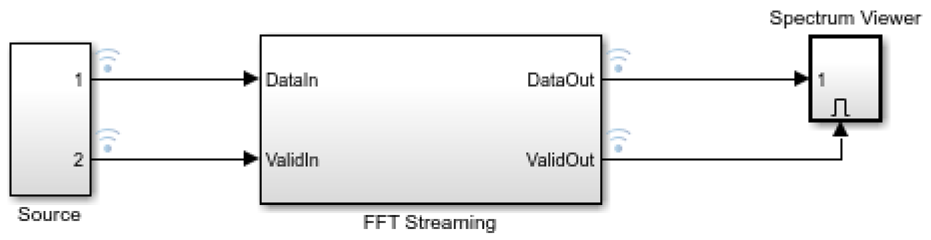
This example includes two models, which show how to use the streaming and burst architectures of the FFT block. The streaming model shows how to use the input and output *valid* control signals to model data rate independently from the clock rate. The burst model shows how to use the *valid* control signal to model bursty data streams and how to use a *ready* signal that indicates when the algorithm can and cannot accept new data samples.

Streaming Radix 2^2 Architecture

Modern ADCs are capable of sampling signals at sample rates up to several gigasamples per second. However, clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. One way to perform GSPS processing on an FPGA is to process multiple samples at the same time at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface which accepts scalar input at a GHz clock rate, and produces a vector of samples at a lower clock rate. Therefore, modern signal processing requires vector processing.

The `Streaming Radix 2^2` architecture is designed to support high-throughput applications. This example model uses an input vector size of 8 and the **Architecture** parameter of the FFT block is set to `Streaming Radix 2^2`. For a timing diagram, supported features, and FPGA resource usage, see FFT.

```
modelname = 'FFTHDLOptimizedExample_Streaming';
open_system(modelname);
```



The InitFcn callback function (Model Properties > Callbacks > InitFcn) sets parameters for the model. In this example, the parameters control the size of the FFT and the input data characteristics.

```
FFTLength  = 512;
```

The input data is two sine waves, 200 KHz and 250 KHz, each sampled at 1*2e6 Hz. The input vector size is 8 samples.

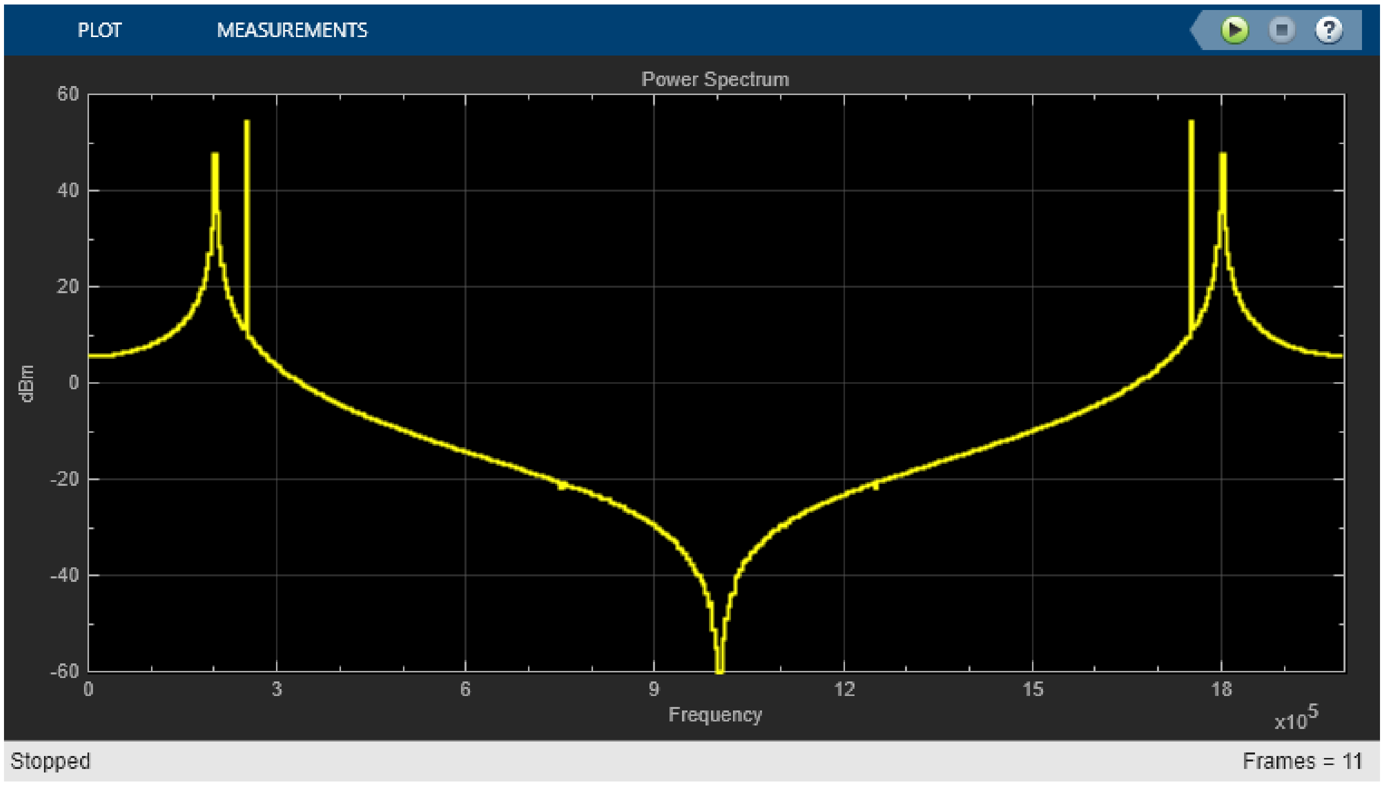
```
FrameSize    = 8;
Fs           = 1*2e6;
```

To demonstrate the use of a **valid** control signal for noncontinuous input data, this example applies valid input every other cycle.

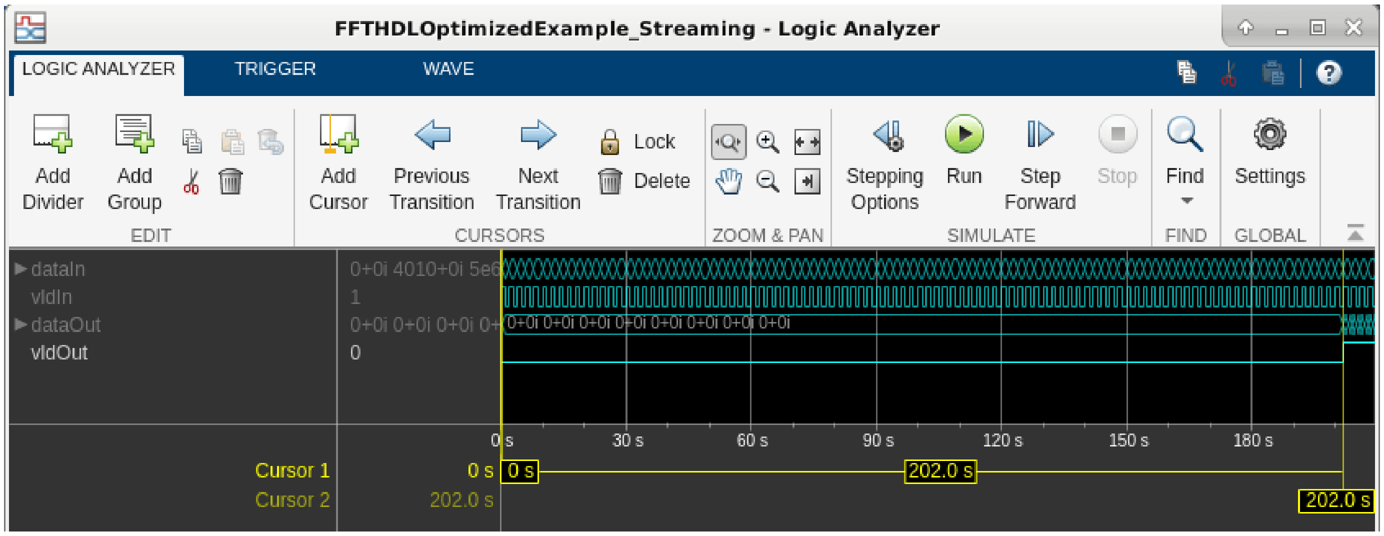
```
ValidPattern = [1,0];
```

Open the Spectrum Viewer and run the example model.

```
open_system('FFTHDLOptimizedExample_Streaming/Spectrum Viewer/Power Spectrum viewer');
set_param(modelname,'SimulationCommand','start')
```



Use the Logic Analyzer to view the input and output signals of the `FFT Streaming` subsystem. The waveform shows that the input valid signal is high every second cycle, and that there is some latency before the block returns the first valid output sample. The block mask displays the latency from the first valid input to the first valid output, assuming no gaps in input valid samples. In this case, the actual latency is longer than the displayed latency because of the gaps in the input stream. The block returns the output data with no gaps in the valid samples.



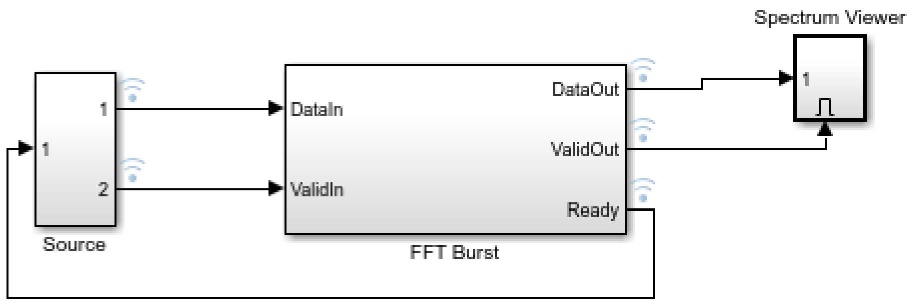
Burst Radix 2 (Minimum Resource) Architecture

Use the `Burst Radix 2` architecture for applications with limited FPGA resources, especially when the FFT length is large. This architecture uses only one complex butterfly to calculate the FFT. In this model, the **Architecture** parameter of the FFT block is set to `Burst Radix 2`.

When you select this architecture, the block has an output control signal, **ready**, that indicates when the block can accept new input data. The block sets the **ready** signal to 1 (`true`) when it can accept data and starts processing once the whole FFT frame is saved into the memory. While processing, the block cannot accept data, so the block sets the **ready** signal to 0 (`false`). You must apply data only when the **ready** signal is 1. The block ignores any data applied while the **ready** signal is 0.

For a timing diagram, supported features, and FPGA resource usage, see FFT.

```
modelname = 'FFTHDLOptimizedExample_Burst';
open_system(modelname);
```



Copyright 2017-2021 The MathWorks, Inc.

The InitFcn callback function (Model Properties > Callbacks > InitFcn) sets parameters for the model. In this example, the parameters control the size of the FFT and the input data characteristics.

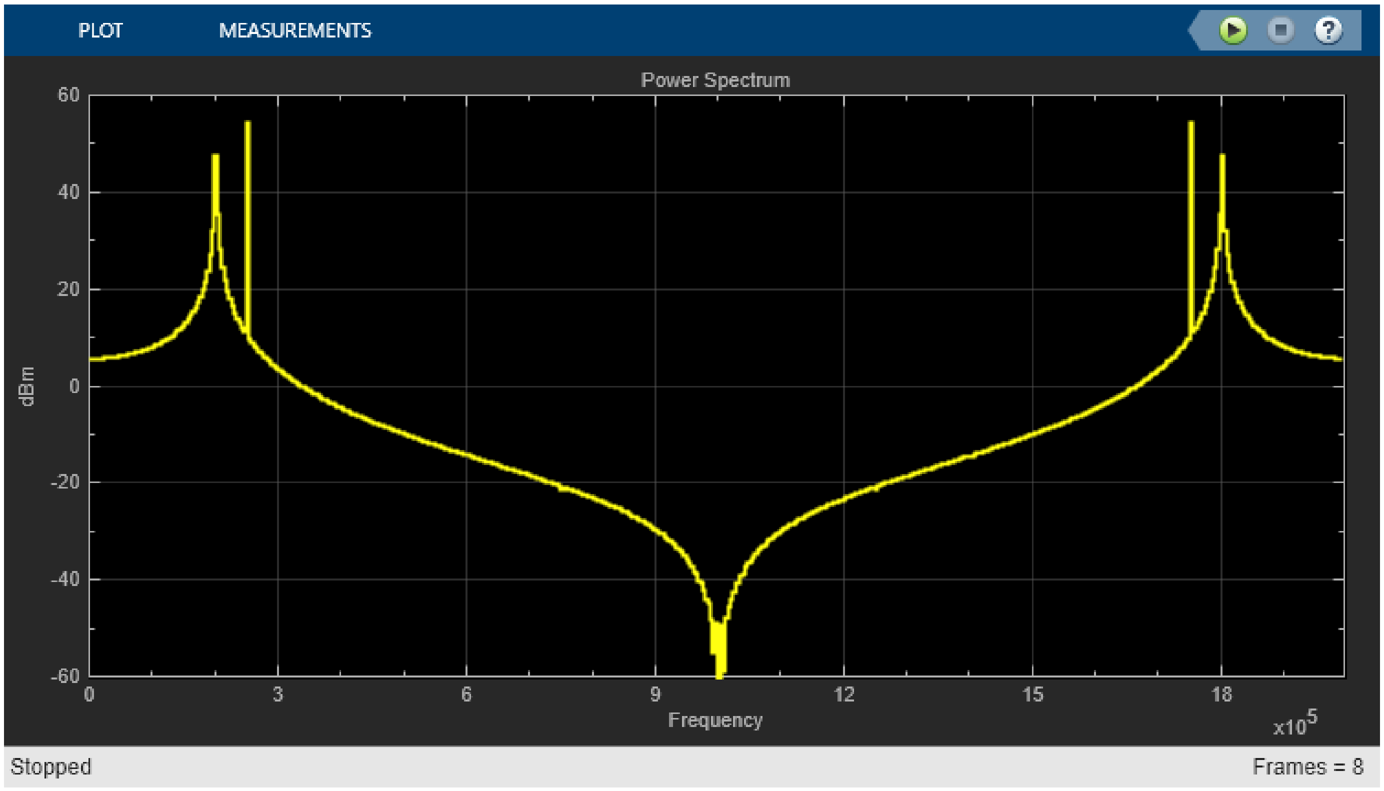
```
FFTLength  = 512;
```

The input data is two sine waves, 200 KHz and 250 KHz, each sampled at 1*2e6 Hz. Data is valid every cycle.

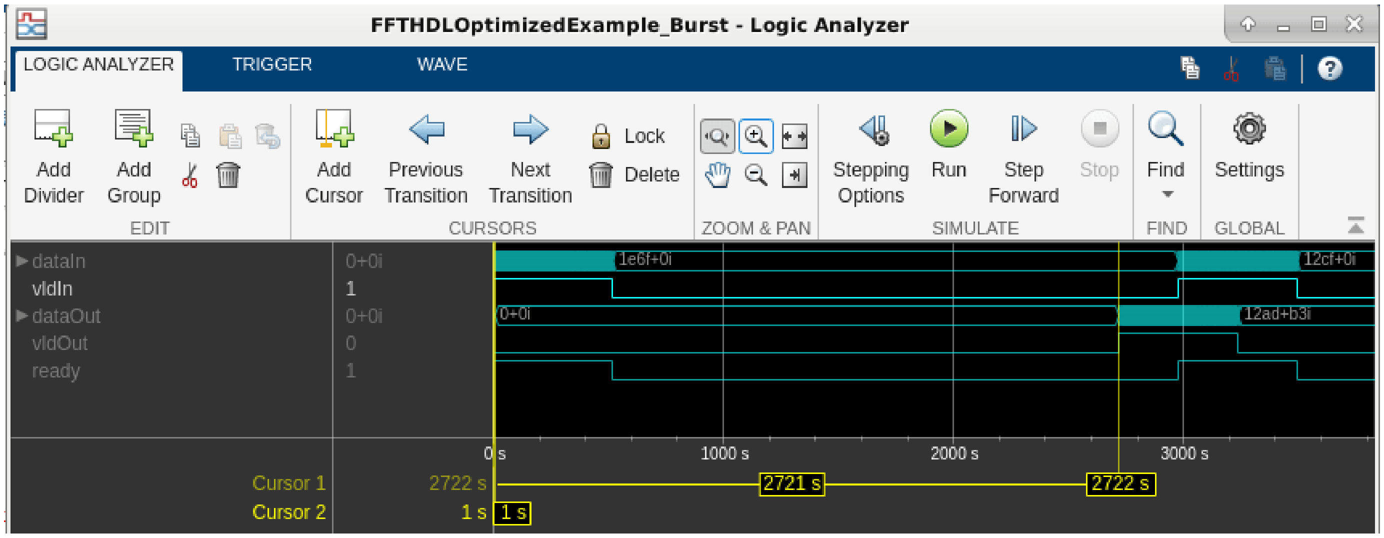
```
Fs           = 1*2e6;
ValidPattern = 1;
```

Open the Spectrum Viewer and run the example model.

```
open_system('FFTHDLOptimizedExample_Burst/Spectrum Viewer/Power Spectrum viewer');
set_param(modelname,'SimulationCommand','start')
```



Use the Logic Analyzer to view the input and output signals of the `FFT Burst` subsystem. The waveform shows that the input data arrives in bursts of valid samples, and that there is some latency before the block returns the valid output samples. The latency in the waveform matches the latency displayed on the block mask. The block also returns an output **ready** signal that indicates when it has room to start accepting the next burst of input data. To ensure that the next burst of input data is not applied before the block sets the **ready** signal to 1 (true), the model uses the ready signal as an enable signal for the input data generation.



Generate HDL Code and Test Bench

You must have the HDL Coder product to generate HDL code for this example. Your model must have a subsystem that is targeted for HDL code generation.

Choose one of the models to generate HDL code and test bench for the FFT subsystem.

```
systemname = 'FFTHDLOptimizedExample_Burst/FFT Burst';
```

or

```
systemname = 'FFTHDLOptimizedExample_Streaming/FFT Streaming';
```

Then, use this command to generate HDL code for that subsystem. The generated code can be used for any FPGA or ASIC target.

```
makehdl(systemname);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb(systemname);
```

See Also

FFT | IFFT

Related Examples

- “High-Throughput Channelizer for FPGA” on page 3-8

More About

- “Digital Signal Processing Design for FPGAs and ASICs” on page 2-2
- “Hardware Control Signals”

High-Throughput Channelizer for FPGA

This example shows how to implement a high-throughput, gigasamples-per-second (GSPS), channelizer for hardware by using a polyphase filter bank.

High speed signal processing is a requirement for applications such as radar, broadband wireless, and backhaul communication. Modern ADCs can sample signals at sample rates up to several GSPS, but the clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. To perform GSPS processing on an FPGA, you can move from scalar processing to vector processing and process multiple samples in parallel at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface, which accepts scalar input at a GHz clock rate and produces a vector of samples at a lower clock rate.

This example shows how to design a signal processing application that supports GSPS throughput. These Simulink® models assume that the input data is vectorized by using a JESD204B interface, and is available at a lower clock rate in the FPGA. The algorithm processes four samples at a time. Both models have a polyphase filter bank that consists of a filter and an FFT. The polyphase filter bank technique minimizes leakage and scalloping loss from the FFT. For more information about polyphase filter banks, see “High Resolution Spectral Analysis in MATLAB”.

The first part of the example uses the Channelizer block from the DSP HDL Toolbox™ library, configured for a 12-tap filter. The Channelizer block uses the polyphase filter bank technique and automatically chooses data types, applies pipelining, and uses other optimizations for hardware performance and resource use. With this block, you can easily explore variations on your design.

The second part of the example implements a polyphase filter bank that has a 4-tap filter. This second part uses basic Simulink blocks. It shows the polyphase filter bank architecture, and the challenges of filter design for hardware, such as choosing data types and inserting pipeline stages.

Implement Channelizer with 12-Tap Filter

This model uses the Channelizer block, configured with a 12-tap filter, that results in good spectrum performance. Using the Channelizer block from the DSP HDL Toolbox library makes it easy to change design parameters such as coefficients, vector size, and FFT length. The block also automatically shares multipliers, calculates fixed-point data types, and pipelines the filter.

The model uses these workspace variables to configure the FFT and filter. The input vector size for this model is 4 samples. The model uses a 512-point FFT and a 12-tap filter for each band. The number of coefficients for the channelizer is 512 frequency bands times 12 taps per frequency band. Generate all the coefficients by using the `tf(h)` function.

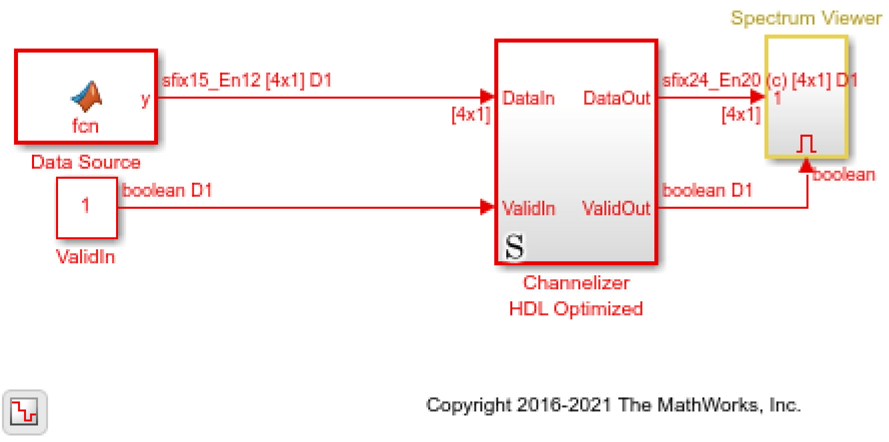
```
FFTLength = 512;
InVect = 4;
hc = dsp.Channelizer;
hc.NumTapsPerBand = 12;
hc.NumFrequencyBands = FFTLength;
hc.StopbandAttenuation = 60;
coef12Tap = tf(hc);
```

The input data consists of two sine waves, 200 kHz and 206.5 kHz. The frequencies are close to each other to illustrate the spectrum resolution of the channelizer.

The `Channelizer HDL Optimized` subsystem contains the Channelizer block and a synchronous State Control (HDL Coder) block that generates hardware-optimized code for the enable logic within the channelizer.

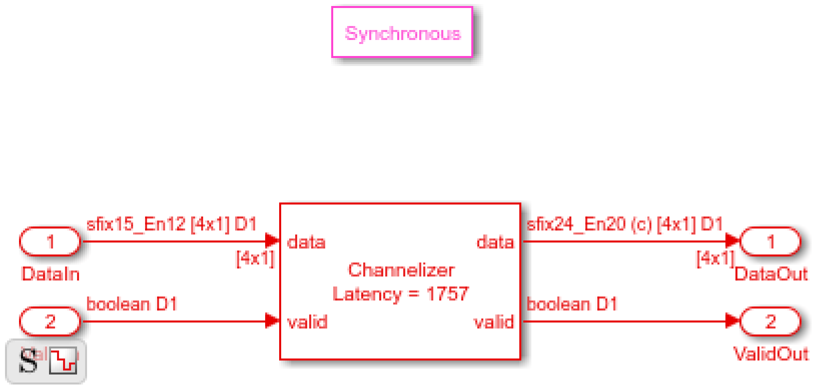
The block implements pipeline stages around the multipliers so that the logic fits into FPGA DSP blocks, and implements the coefficient banks inside the channelizer with ROM blocks. These hardware-friendly options are the default behavior for the DSP HDL Toolbox blocks.

```
modelname = 'PolyphaseFilterBankHDLExample_HDLChannelizer';
open_system(modelname);
set_param(modelname,'SimulationCommand','Update');
```

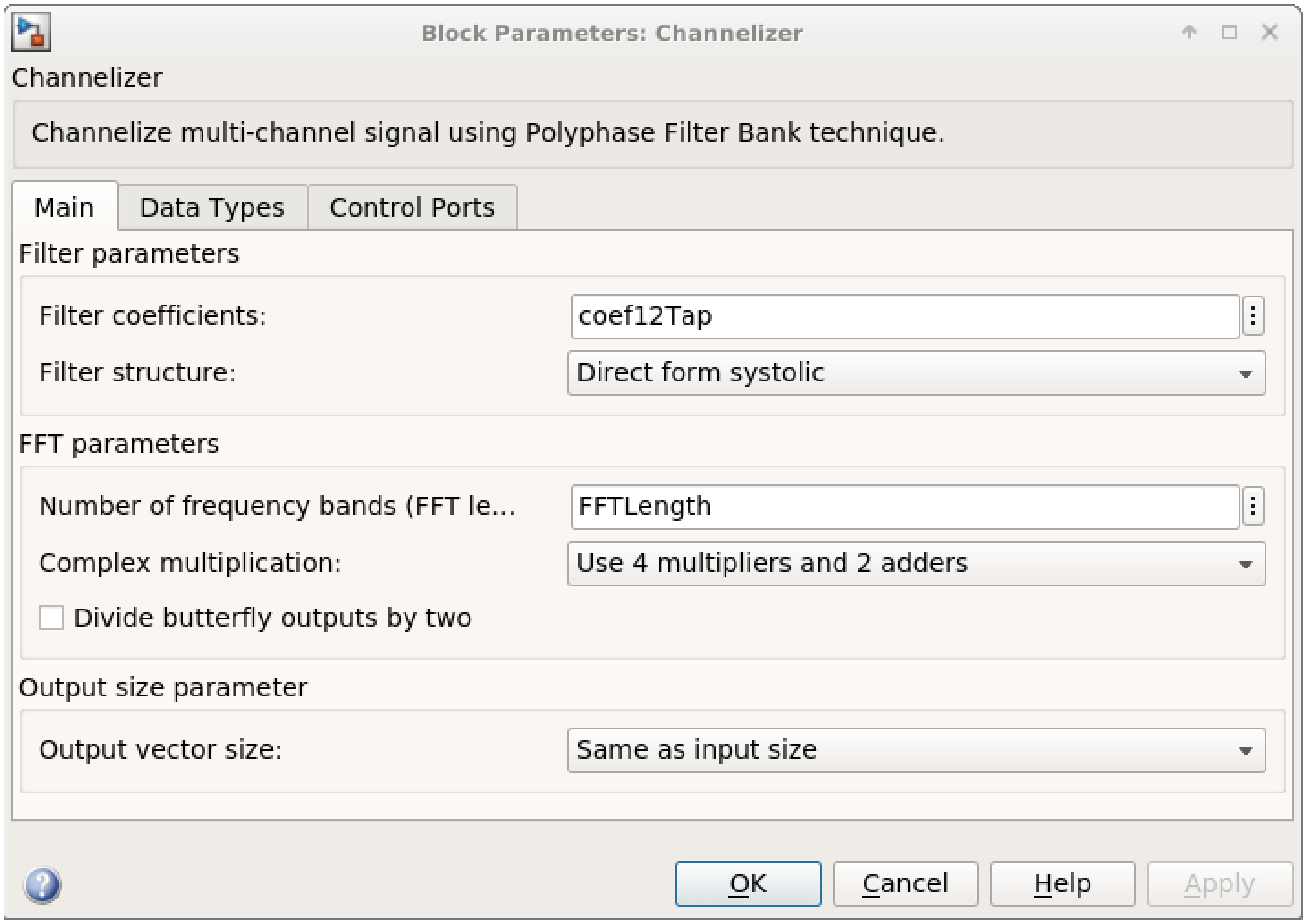


This model diagram shows the Channelizer block inside the subsystem.

```
open_system([modelname,'/Channelizer HDL Optimized'])
```



The block has parameters to configure the filter coefficients, FFT length, and other settings that enable you to explore different hardware implementations of the algorithm.



Channelizer

Channelize multi-channel signal using Polyphase Filter Bank technique.

Filter parameters

Filter coefficients: coef12Tap

Filter structure: Direct form systolic

FFT parameters

Number of frequency bands (FFT le... FFTLength

Complex multiplication: Use 4 multipliers and 2 adders

☐ Divide butterfly outputs by two

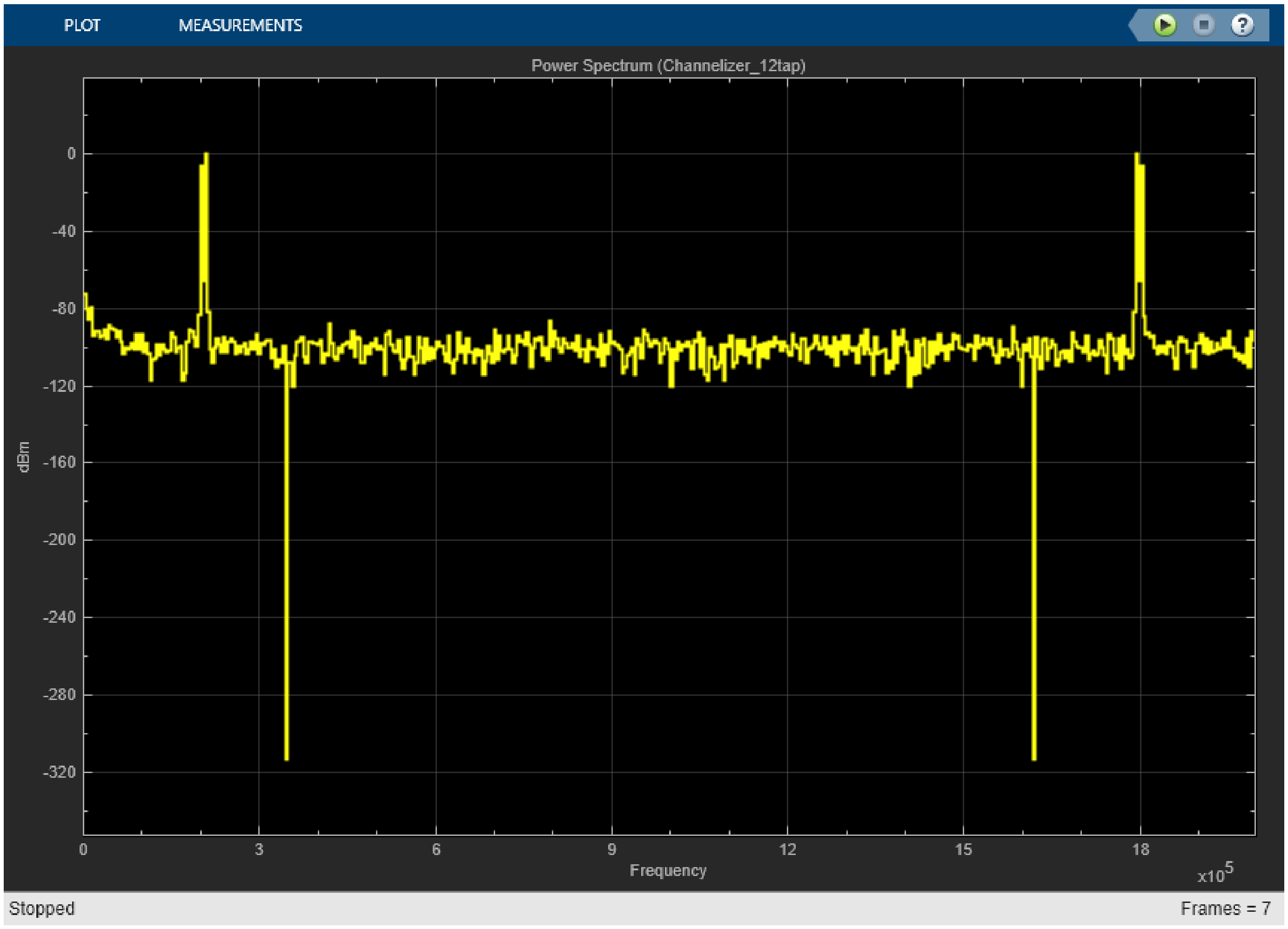
Output size parameter

Output vector size: Same as input size

View Simulation Results

To visualize the spectrum result, open the spectrum viewer and run the model.

```
open_system([modelname,'/Spectrum Viewer/Power Spectrum viewer (Channelizer_12tap)']);
sim(modelname);
```



The spectrum viewer shows that the 12-tap filter separates the spectrum of the two signals. Zoom in between 100 kHz and 300 kHz to see where the channelizer detects two peaks. Two peaks is the expected result because the input signal has two frequency components.

Generate HDL Code

You must have the HDL Coder™ product to generate HDL code for this example model. Use this command to generate HDL code.

```
makehdl([modelname,'/Channelizer HDL Optimized']);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb([modelname,'/Channelizer HDL Optimized']);
```

The design was synthesized for Xilinx Virtex 7 (xc7vx550t-ffg1158, speed grade 2). The design achieves a clock frequency of 361 MHz. At 4 samples per clock, this frequency results in 1.4 GSPS throughput.

The **Minimize clock enable** HDL code generation option is on in this model. The clock enable signal is a global signal, which is not recommended for high speed designs. In the model Configuration

Parameters, choose **HDL Code Generation**, **Global settings**, **Ports**, and then select **Minimize clock enable**. This option is supported when the model is single rate.

The FFT block uses 56 DSP blocks on the FPGA, and the filter uses 48 DSP blocks.

```
 T = table(...
    categorical({'LUT'; 'LUT RAM'; 'FF'; 'BRAM'; 'DSP'}), ...
    [26641; 14585; 24816; 2; 104], ...
    'VariableNames',{'Resource','Usage'});

disp(T);
```

```
    Resource    Usage
    ________    _____

    LUT         26641
    LUT RAM     14585
    FF          24816
    BRAM            2
    DSP           104
```

Implement 4-Tap Polyphase Filter Bank

To show the internal implementation of a polyphase filter bank, the second example model implements a 512-point FFT by using the DSP HDL Toolbox FFT block and a 4-tap filter for each band by using basic Simulink blocks. The 4-tap filter has lower frequency resolution than the 12-tap filter in the first model, but it is easier to see the structure of the filter. These MATLAB variables configure the blocks in the model. Use the `dsp.Channelizer` System object™ to generate the coefficients. The `polyphase` method of the channelizer object generates a 512-by-4 matrix. Each row represents the coefficients for one band. Cast the coefficients to fixed-point types that have the same word length as the input signal.

```
h = dsp.Channelizer;
h.NumTapsPerBand = 4;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef4Tap = fi(polyphase(h),1,15,14,'RoundingMethod','Convergent');
```

The algorithm requires 512 filters (one filter for each band). For a vector input of 4 samples, the model implements four parallel 4-tap filters. Each filter applies 128 sets of coefficients.

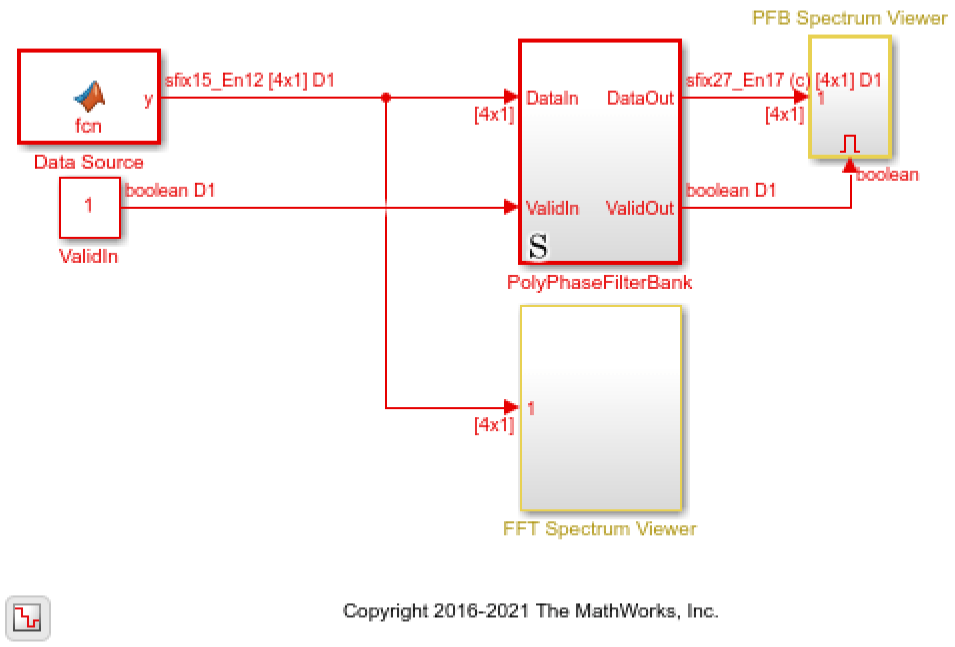
```
ReuseFactor = FFTLength/InVect;
```

These variables configure the model to pipeline the multipliers and the coefficient bank to fit the logic into DSP blocks on the FPGA. Using the DSP blocks enables synthesis to a higher clock rate.

```
Multiplication_PipeLine = 2;
CoefBank_PipeLine       = 1;
```

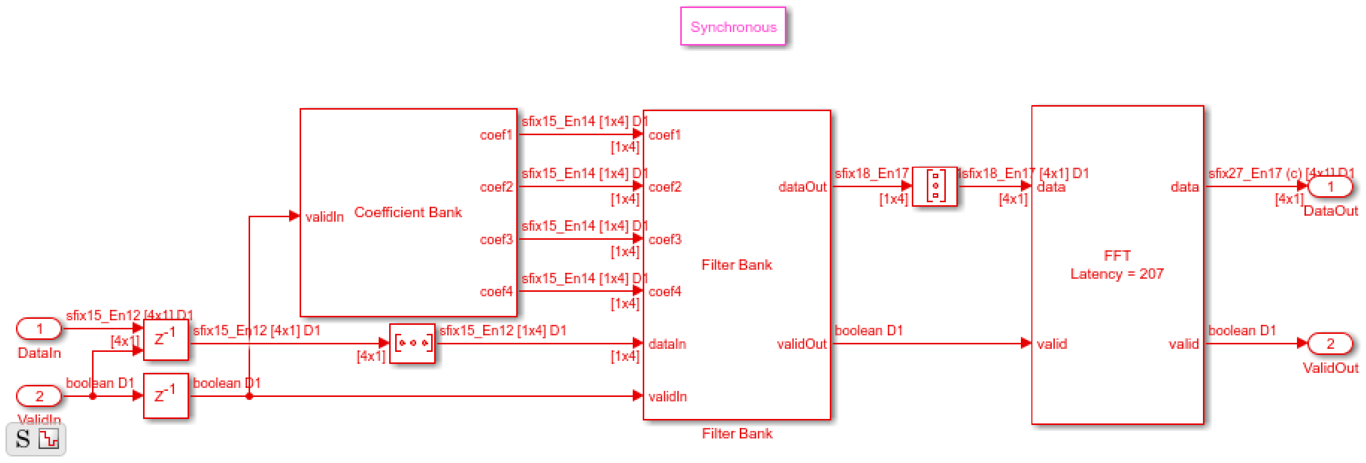
The input data consists of two sine waves, 200 kHz and 250 kHz. These two frequencies are farther apart than the previous model because the smaller filter has lower spectrum performance. The input and output of the `PolyPhaseFilterBank` subsystem are 4-by-1 vectors.

```
modelname = 'PolyphaseFilterBankHDLExample_4tap';
open_system(modelname);
set_param(modelname,'SimulationCommand','Update');
```



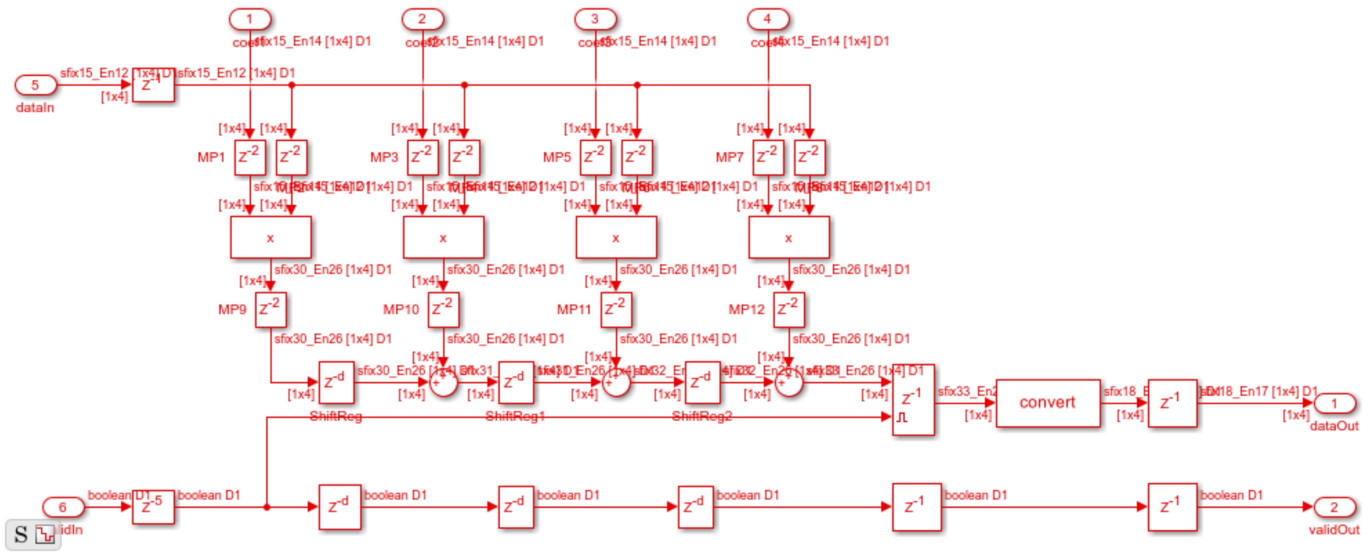
The `PolyphaseFilterBank` subsystem contains the `Coefficient Bank` subsystem that rotates over the coefficient sets. The `Filter Bank` subsystem accepts vectors of coefficients and data and returns a vector of filtered data. The FFT block also accepts and returns a vector of 4 samples, and implements a hardware-optimized architecture.

```
open_system([modelname,'/PolyPhaseFilterBank'])
```



This model diagram shows the pipelined 4-tap filter implementation.

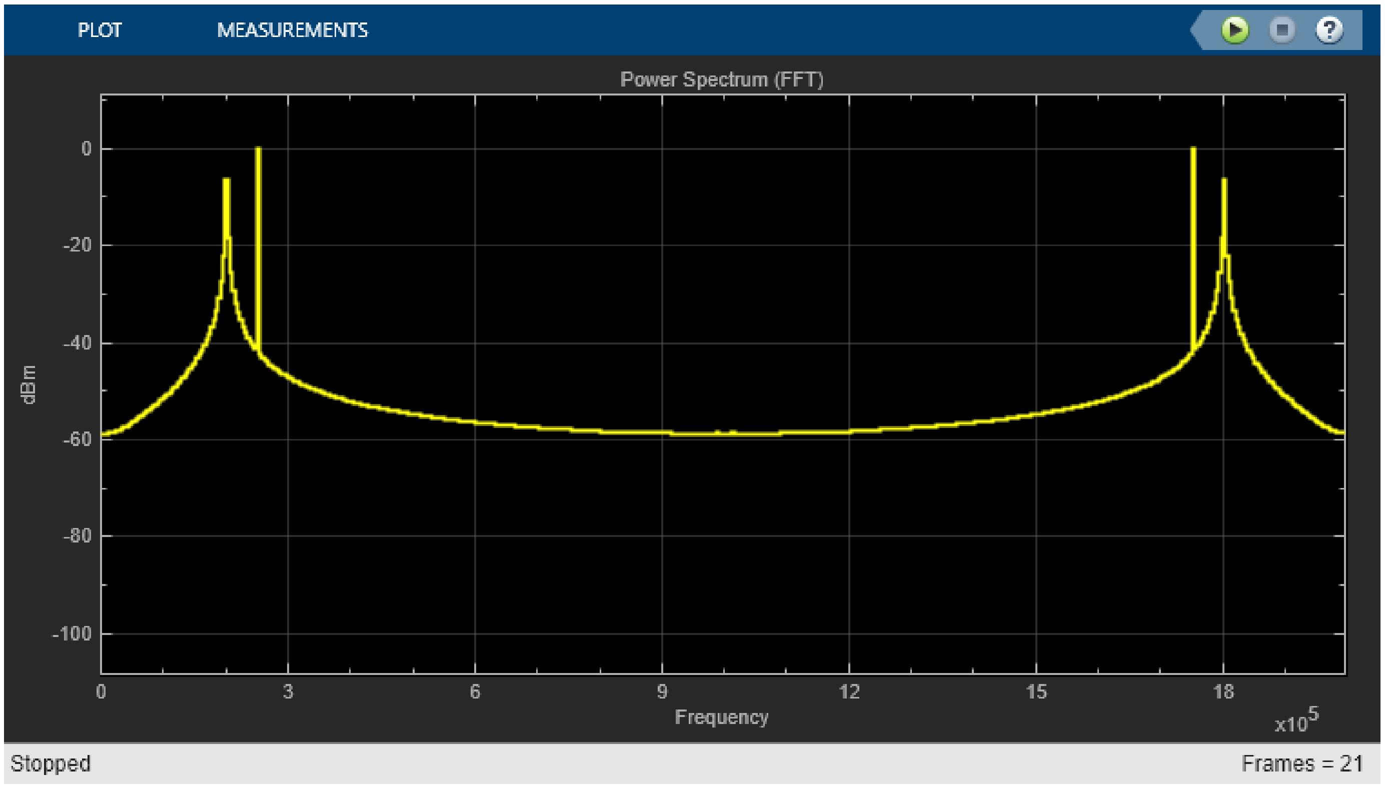
```
open_system([modelname,'/PolyPhaseFilterBank/Filter Bank'])
```

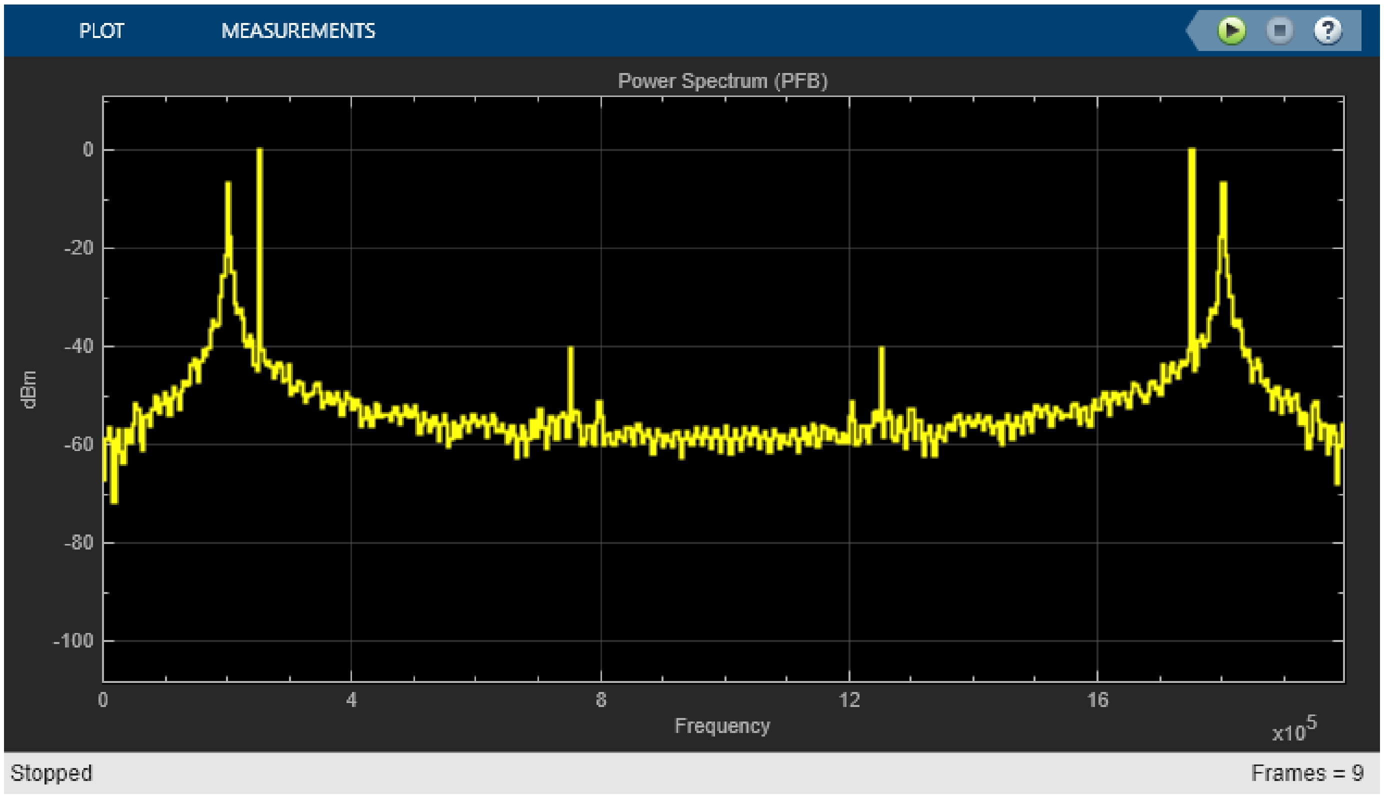


View Simulation Results

To visualize the result of the simulation, open the spectrum viewers and run the model.

```
open_system([modelname,'/FFT Spectrum Viewer/Power Spectrum viewer (FFT)']);
open_system([modelname,'/PFB Spectrum Viewer/Power Spectrum viewer (PFB)']);
sim(modelname);
```





The polyphase filter bank spectrum viewer shows the improvement in the power spectrum and minimization of frequency leakage and scalloping compared with using only an FFT. By comparing the two spectrums, and zooming in between 100 kHz and 300 kHz, you can see that the polyphase filter bank has fewer peaks over –40 dB than the classic FFT.

Considerations for Optimized Hardware

Data Type

Data word length affects both the accuracy of the result and the resources used in the hardware. This 4-tap filter uses full precision. With an input data type of `fixdt(1,15,13)`, the output is `fixdt(1,18,17)`. The absolute values of the filter coefficients are all smaller than 1, so the data does not grow after each multiplication operation. Each addition adds one bit to the data type. To keep the accuracy in the FFT, the data type grows one bit for each stage. This growth makes the twiddle factor multiplication larger at each stage. For many FPGAs, a multiplication size smaller than 18-by-18 is desirable. Because a 512-point FFT has 9 stages, the input of the FFT cannot be more than 11 bits. The first 8 binary digits of the maximum coefficient in this case are zero. Therefore, this example casts the coefficients to `fixdt(1,7,14)` instead of `fixdt(1,15,14)`. Also, the maximum value of the Datatype block output inside the polyphase filter bank has 7 leading zeros after the binary point, so the model casts the filter output to `fixdt(1,11,17)`. These adjustments keep the FFT internal multiplier size smaller than 18-by-18 and save hardware resources.

Design for Speed

The model uses these settings to enable the generated HDL code to synthesize to a faster clock rate.

1 Synchronous State Control block — Implements hardware-friendly enable logic for Delay blocks.

2 Minimize clock enable — Avoids implementing a global clock enable signal that could decrease the synthesized clock rate.
3 Use DSP block in FPGA — Maps multipliers into DSP blocks in the FPGA by including 2 delays before each multiplier and 2 delays after. These pipeline registers cannot have a reset signal. Set the reset type to none for each pipeline by right-clicking the Delay block and selecting **HDL Code** > **HDL Block Properties**, then setting **Reset Type** to `None`.
4 Use ROM in FPGA — Map the combinatorial logic inside the Coefficient MATLAB Function block (inside the `Coefficient Bank` subsystem) to a ROM by adding a register after the block. The delay length is set by `CoefBank_PipeLine`. Set the reset type for these delays to none.

Generate HDL Code

You must have the HDL Coder™ product to generate HDL code for this example model. Use this command to generate HDL code.

```
makehdl([modelname,'PolyPhaseFilterBank']);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb([modelname,'PolyPhaseFilterBank']);
```

When the design is synthesized for Xilinx® Virtex 7 (xc7vx550t-ffg1158, speed grade 2), the design achieves a clock frequency of 324 MHz (before place and route). At 4 samples per clock, this frequency results in 1.3 GSPS throughput.

The FFT block uses 56 DSP blocks in the FPGA, and the filter uses 16 DSP blocks.

```
 T = table(...
    categorical({'LUT'; 'LUT RAM'; 'FF'; 'BRAM'; 'DSP'}), ...
    [14713; 3570; 21514; 2; 72], ...
    'VariableNames',{'Resource','Usage'});

disp(T);
```

```
    Resource    Usage
    ________    _____

    LUT         14713
    LUT RAM      3570
    FF          21514
    BRAM            2
    DSP            72
```

Conclusion

The first part of the example shows how the Channelizer block from the DSP HDL Toolbox library makes it easy to implement an algorithm for hardware, and provides quick exploration of design options.

The second part of the example shows design considerations when implementing filters for hardware without using a hardware-optimized library block, such as choosing data types and inserting pipeline stages. When you use the library blocks from DSP HDL Toolbox, you do not have to consider these factors yourself. The blocks implement hardware-optimized algorithms that are ready for HDL code generation and deployment to FPGAs or ASICs.

See Also

FFT | IFFT | Channelizer | Channel Synthesizer

Related Examples

- “HDL Implementation of Four Channel Synthesizer and Channelizer”
- “Implement FFT Algorithm for FPGA” on page 3-2

More About

- “Digital Signal Processing Design for FPGAs and ASICs” on page 2-2
- “Hardware Control Signals”
- “High-Throughput HDL Algorithms”