SIEMENS EDA

Algorithmic C (AC) DSP Library **Reference Manual**

Software Version v3.5.0
August 2023

Table of Contents

Table of Contents

Table of Contents

Chapter 1: Introduction to ac_dsp

The Algorithmic C DSP Library (*ac_dsp*) contains synthesizable C++ functions commonly used in Digital Signal Processing operations like filters and Fast Fourier Transforms. The functions use a C++ class-based object design so that it is easy to instantiate multiple variations of objects into a more complex subsystem and utilizes the AC Datatypes for true bit-accurate behavior. The filter block set includes filters for CIC Interpolation/Decimation, FIR, Moving Average and Polyphase Interpolation/Decimation designs. The FFT block set includes various FFT architectures based on Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF), various radix modes and various buffer architectures (memory-exchange, single-delay-feedback, in-place).

The input and output arguments of these functions are parameterized so that arithmetic may be performed at the desired fixed point precision and provide a high degree of flexibility on the area/performance trade-off of hardware implementations obtained during Catapult synthesis.

The following sections provide a summary of the *ac_dsp* library:

- Using the ac_dsp Library
- Summary of Classes
- Installing the ac_dsp Library

1.1.1. Using the ac_dsp Library

In order to utilize any of the top-level ac_dsp classes, locate the required *ac_** header file from the ac_dsp directory, and include that in your code. A summary of the files and classes within the ac_dsp directory is given in Summary of Classes. A line that includes the required file for the ac_fft_dit_r2_inpl class is given below.

```
#include <ac_fft_dit_r2_inpl.h>
```

1.2. Summary of Classes

The following tables summarize the classes currently supported in the ac_dsp library. Each class has a public *run()* interface function that is synthesized as the top-level block by catapult and is called by the user in their design, in accordance with the requirements of class-based hierarchy.

Filter Blocks

Filter Implementation	Class Name	Header File
CIC Decimator (Full Precision)	ac_cic_dec_full	ac_cic_dec_full.h
CIC Interpolator (Full Precision)	ac_cic_intr_full	ac_cic_intr_full.h
FIR Filter – Constant Coefficients	ac_fir_const_coeffs	ac_fir_const_coeffs.h
FIR Filter – Loadable Coefficients	ac_fir_load_coeffs	ac_fir_load_coeffs.h
FIR Filter – Programmable Coefficients	ac_fir_prog_coeffs	ac_fir_prog_coeffs.h
FIR Filter – Register Sharing	ac_fir_reg_share	ac_fir_reg_share.h
Integrate and Dump	ac_intg_dump	ac_intg_dump.h
1-D Moving Average Filter	ac_mv_avg	ac_mv_avg.h
Polyphase Decimation Filter	ac_poly_dec	ac_poly_dec.h
Polyphase Interpolation Filter	ac_poly_intr	ac_poly_intr.h

FFT Blocks

FFT Implementation	Class Name	Header File
DIF Radix-2 In-place	ac_fft_dif_r2_inpl	ac_fft_dif_r2_inpl.h
DIF Mix-Radix Single-Delay Feedback	ac_fft_dif_r2m2p2_sdf	ac_fft_dif_r2m2p2_sdf.h
DIF Radix-22 Single-Delay-Feedback	ac_fft_dif_r2p2_sdf	ac_fft_dif_r2p2_sdf.h
DIF Radix-2X Block Floating Point In-place	ac_fft_dif_r2pX_bfp_inpl	ac_fft_dif_r2pX_bfp_inpl.h
DIF Radix-2X Dynamic In-place	ac_fft_dif_r2pX_dyn_inpl	ac_fft_dif_r2pX_dyn_inpl.h
DIF Radix-2X In-place	ac_fft_dif_r2pX_inpl	ac_fft_dif_r2pX_inpl.h
DIF Radix-2 Single-Delay-Feedback	ac_fft_dif_r2_sdf	ac_fft_dif_r2_sdf.h
DIT Radix-2 In-place	ac_fft_dit_r2_inpl	ac_fft_dit_r2_inpl.h
DIT Radix-2 Single-Delay-Feedback	ac_fft_dit_r2_sdf	ac_fft_dit_r2_sdf.h

1.3. Installing the ac_dsp Library

The library consists of the directories and files shown here:

```
.
|-- include
|   `-- ac_dsp
|       |-- ac_cic_dec_full.h
|       |-- ac_cic_full_core.h
|       |-- ac_cic_intr_full.h
```

```
|       |-- ac_fft_dif_r2_inpl.h
|       |-- ac_fft_dif_r2_sdf.h
|       |-- ac_fft_dif_r2m2p2_sdf.h
|       |-- ac_fft_dif_r2p2_sdf.h
|       |-- ac_fft_dif_r2pX_bfp_inpl.h
|       |-- ac_fft_dif_r2pX_dyn_inpl.h
|       |-- ac_fft_dif_r2pX_inpl.h
|       |-- ac_fft_dit_r2_inpl.h
|       |-- ac_fft_dit_r2_sdf.h
|       |-- ac_fir_const_coeffs.h
|       |-- ac_fir_load_coeffs.h
|       |-- ac_fir_prog_coeffs.h
|       |-- ac_fir_reg_share.h
|       |-- ac_intg_dump.h
|       |-- ac_mv_avg.h
|       |-- ac_poly_dec.h
|       |-- ac_poly_intr.h
|       |-- twiddlesR_64bits.h
|       |-- twiddles_20bits.h
|       `-- twiddles_64bits.h
|-- pdfdocs
|   `-- ac_dsp_ref.pdf
```

In order to utilize this library you must have the AC Datatypes package and the AC Math package installed and configure your software environment to provide the path to the “include” directories of these packages as part of your C++ compilation arguments.

Chapter 2: Filter Functions

The *ac_dsp* library include the following filter blocks:

- Cascaded Integrator-Comb (CIC) Full Precision Filters (ac_cic_dec_full and ac_cic_intr_full)
- Integrate and Dump Filter (ac_intg_dump)
- FIR Filters (ac_fir_const_coeffs, ac_fir_load_coeffs, ac_fir_prog_coeffs and ac_fir_reg_share)
- 1-D Moving Average (ac_mv_avg)
- Poly-Phase Interpolation (ac_poly_intr)
- Poly-Phase Decimation (ac_poly_dec)

For all the filter designs, the input and output are passed in the form of an ac_channel streaming interfaces. The design itself uses class-based hierarchy for its implementation with the top level member function in each class being implemented as a separate hierarchical block. The core class, which contains all the computation, always has its member functions inlined.

2.1. Cascaded Integrator-Comb (CIC) Full Precision Filters (ac_cic_dec_full and ac_cic_intr_full)

CIC filters have cascaded stages of integrators and comb filters, one after the other. Depending upon whether the integrator section comes before or after the comb section, we have two broad classes of CIC filters: CIC decimators and CIC interpolators. CIC decimators have the integrator section preceding the comb section, with a down-sampler separating the two. Conversely, CIC interpolators have the comb section preceding the integrator section, with an up-sampler separating the two.

Both integrator and comb sections have equal number of stages without a multiplier and without the need to store coefficients, making the CIC filters highly hardware-efficient and useful for high rate change applications. The response of the filters is determined by the number of stages, the differential delay and the rate change factor. Illustration 1 shows a high-level block diagram of the CIC interpolation filter.

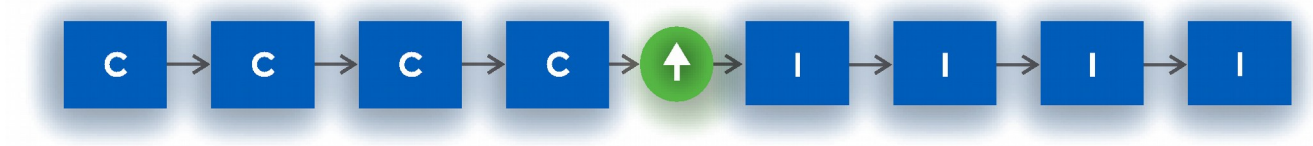


Illustration 1: CIC Interpolation Filter

2.1.1. Features

- C++ implementation provided.
- Configurable number of stages.
- Configurable differential delay.
- Configurable rate change factor.
- Configurable input/output data width.
- Full precision bitwidth for integrator and comb stages.

2.1.2. Limitations

- No multi-channel support.
- Configuration datatypes must support the following operators: =, +

2.1.3. Functional Description

An integrator filter is a single pole accumulator with a transfer function:

$$H_I(Z) = \frac{1}{1 - Z^{-1}}$$

A comb filter is a differentiator with a transfer function:

$$H_C(z) = 1 - Z^{-M}$$

In this equation, M is the differential delay, and is usually limited to 1 or 2. In a CIC filter, the integrators operate at a high sampling frequency (fS), and the comb filters operate at low frequency (fS/R). Therefore, the transfer function of CIC decimators as derived from the previous two equations is:

$$H(z) = {H_I}^N(z){H_c}^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N$$

Where:

- HI is the transfer function of the integrator part of the filter.
- HC is the transfer function of the comb part of the filter.
- N is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part or the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the decimation factor.
- M is the differential delay.

Full Precision Bitwidth

For the full-precision CIC designs, the ac_fixed input and output data-types are passed by the user as template parameters. Based on the input bitwidth as well as the N, R and M filter parameters, the full-precision bitwidth for the intermediate stages of the CIC decimation filter is calculated as:

$$B_{max} = ceil(N\ log_2(RM) + B_{in})$$

The full-precision bitwidth for the intermediate stages of the CIC interpolation filter is calculated as:

$$B_{max} = ceil(log_2(R^{N-1}M^N) + B_{in})$$

In the above two equations, Bin is the input bitwidth.

CIC as a Moving Average Filter

The recursive form of a moving average filter can be implemented as CIC filter. The difference equation of a moving average filter is given as:

$$y[n] = \sum_{k=0}^{N-1} x[n-k]$$

The previous equation can also be rewritten as:

$$y[n] = x[n] - x[n-N] + y[n-1]$$

This difference equation resembles CIC interpolator filter with one stage and a differential delay of N as shown in Illustration 2.

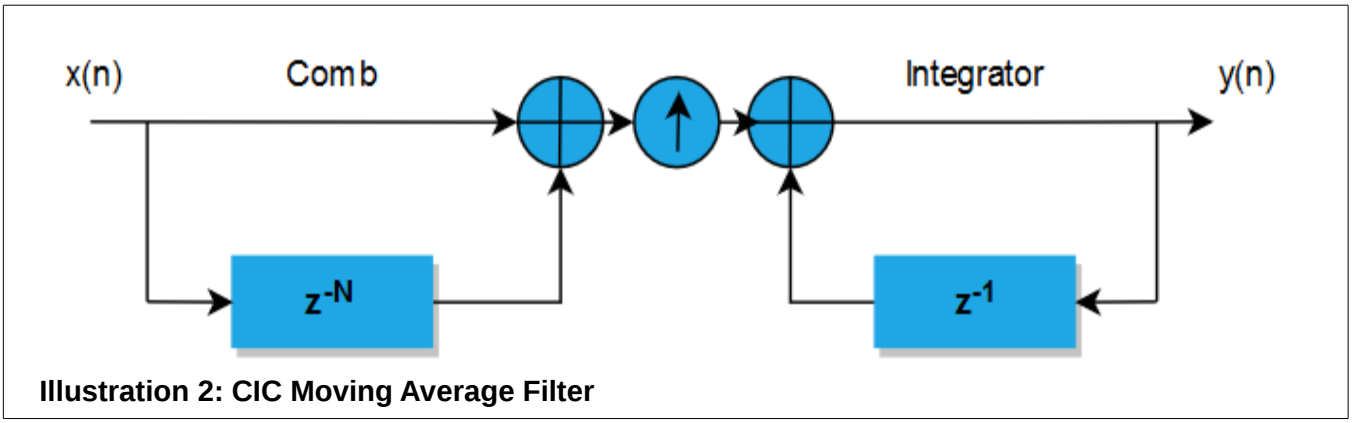


Illustration 2: CIC Moving Average Filter

The *ac_dsp CIC* filter can hence be used as a moving average filter.

2.1.4. Model Parameters / Function Signature

The IP for the cic filter designs is implemented as a class template in C++ design, using class-based hierarchy. It depends upon the core class for the actual computation, with a member function of the aforementioned class template being synthesized as the top-level interface.

The template signature for both the decimation and interpolation filter classes is shown below:

```
// Decimation filter:
template < class IN_TYPE, class OUT_TYPE, unsigned R_, unsigned M_, unsigned N_
>
class ac_cic_dec_full
{
// code
}

// Interpolation filter:
template < class IN_TYPE, class OUT_TYPE, unsigned R_, unsigned M_, unsigned N_
>
class ac_cic_intr_full
{
// code
}
```

A description of the various class template parameters used is given below:

Parameter	Description
IN_TYPE	Input type. Must be an ac_fixed datatype.
OUT_TYPE	Output type. Must be an ac_fixed datatype.
R_	Decimation Factor.
M_	Differential delay for comb sections.
N_	Number of stages.

The prototype of the top-level member functions for both the interpolation and decimation filters is given below:

```
// Decimation filter:
void run(ac_channel <IN_TYPE> &data_in, ac_channel <OUT_TYPE> &data_out)
{
// code
}

// Interpolation filter:
void run(ac_channel <IN_TYPE> &data_in, ac_channel <OUT_TYPE> &data_out)
{
// code
}
```

2.1.5. Class Hierarchy

Integrator and comb sections are designed as blocks, through class-based hierarchy. The top-level member function is specified as the top-level design through the hls_design interface pragma. Illustration 3 shows the instance hierarchy for a sample CIC decimator design.

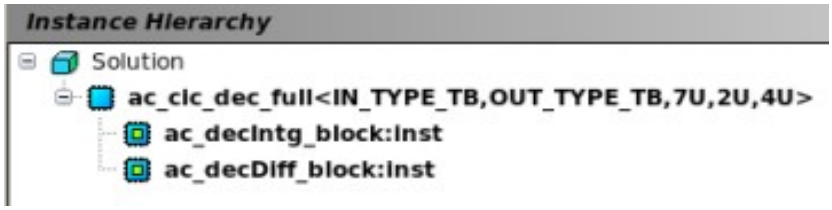


Illustration 3: Instance Hierarchy for ac_cic_dec_full design.

2.1.6. Synthesis Considerations

The following table specifies the different types of loops in the design and synthesis constraint options for them.

Loops	Description
INTG_STG	Integrator loop should be fully unrolled for throughput 1 and can be partially unrolled based on requirements.
COMB	Comb loop, can be partially unrolled for adder reuse.
COMB_B	Differential delay loop, should be fully unrolled. Design has two loops for interpolation and decimation configurations. COMB_B loop is visible only when differential delay is greater than one and should be fully unrolled.

The *INTG_STG* loop should be fully unrolled to implement parallel integrator stages, however it can also be partially unrolled for adder sharing between the stages at the cost of throughput.

Differentiators in the comb section remain idle for the time equal to rate change factor. So for better hardware utilization, adders in the comb section can be shared among the differentiators. The *COMB* loop can be

partially unrolled to do this. The bitwidth of all intermediate stages is the same, further allowing the reuse of low rate comb adders.

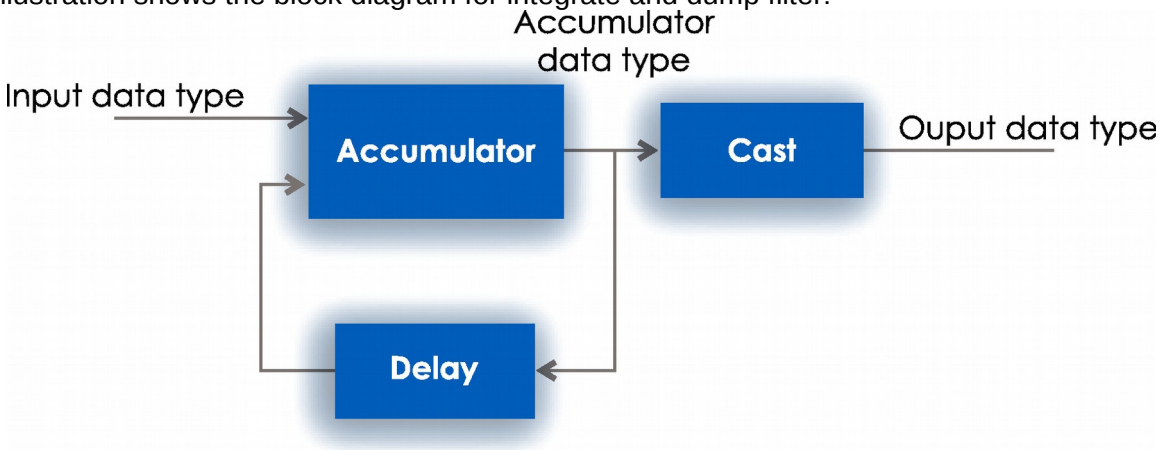
2.1.7. Usage Example

The following example shows the steps to instantiate the decimation filter and call the top-level member function. The interpolation filter can be instantiated in a similar manner.

```
// Include the filter header
#include <ac_dsp/ac_cic_dec_full.h>

// Define configuration parameters and I/O data types
enum {
    R_TB       = 7,     // Rate change factor
    M_TB       = 2,     // Differential Delay
    N_TB       = 4      // Number of stages
  };
typedef ac_fixed <32, 16, true> IN_TYPE_TB;
typedef ac_fixed <48, 32, true> OUT_TYPE_TB;

// Declare channel inputs and outputs
ac_channel<IN_TYPE_TB> fixed_in;
ac_channel<OUT_TYPE_TB> fixed_out;

// Instantiate the filter class object
ac_cic_dec_full < IN_TYPE_TB, OUT_TYPE_TB, R_TB, M_TB, N_TB > filter;

// Push values into input channel
// for (...) {
//   ... fixed_in.write(...)
// }

// Call the filter function
filter.run(fixed_in, fixed_out);

// Process outputs
// while (fixed_out.available(1)) {
//   ... fixed_out.read();
// }
```

2.2. Integrate and Dump Filter (ac_intg_dump)

The Integrate and Dump block creates a cumulative sum of a given number of samples N and resets the sum to zero after every N input samples, where N is the integration period parameter value. The reset occurs in the same cycle that the block produces its output allowing a continuous stream of data at the input. A following illustration shows the block diagram for integrate and dump filter:



2.2.1. Features

- C implementation
- Streaming input and output
- Configurable number of maximum input data samples
- Runtime setting for number of samples to be integrated in current burst
- Multiple channel implementation with interleaved input/output
- Configurable Input/Output, accumulation data width
- Serial or parallel architecture generated by controlling loop constraints

2.2.2. Limitations

- The input N is registered and must be valid in the cycle when the first input is to be accumulated
- Configuration datatypes must support the following operators: =, +

2.2.3. Model Parameters / Function Signature

The Integrate and Dump filter block is implemented as a C++ class. The template signature for the class is:

```
template < class IN_TYPE, class ACC_TYPE, class OUT_TYPE, class N_TYPE,
```

```
          int NS, int CHN>
class ac_intg_dump
{
// code
}
```

A description of the various class template parameters used is given below:

Parameter	Description
IN_TYPE	Data type of input samples
ACC_TYPE	Data type of internal accumulator
OUT_TYPE	Data type of output samples
N_TYPE	Data type for number of samples configuration
NS	Maximum number of samples to be integrated in an implementation
CNH	Number of channels

The prototype of the top-level member functions for both the interpolation and decimation filters is given below:

```
void run(ac_channel <IN_TYPE> &data_in, ac_channel <OUT_TYPE> &data_out,
         ac_channel <N_TYPE> &n_sample)
{
// code
}
```

2.2.4. Synthesis Considerations

specifies the different types of loops in the design and how they are treated.

Loops

Loops	Description
ACC_LOOP	Accumulation loop accumulates incoming samples for N times defined in the port. Loop can be pipelined with an II of 1 for parallel architecture. However input data width should be increased accordingly.
CHN_LOOP	Channel loop iterates for a number defined in configuration. Loop can be pipelined with an II of 1 for parallel architecture. However input data width should be increased accordingly.

2.2.5. Usage Example

The following steps show an example of how to instantiate a filter.

```
// Include the filter header
#include <ac_dsp/ac_intg_dump.h>
```

```
// Define configuration parameters and I/O types
       enum {
          CONFIG = 25, //No of configurations
          NS  = 1024,  //Max no of samples that should be accumulated in a
configuration
          CHN = 4      //No of channels configured
       };
 typedef ac_int < 16, false > N_TYPE;
 typedef ac_fixed < 25, 15, false > IN_TYPE;
 typedef ac_fixed < 60, 15, false > ACC_TYPE;
 typedef ac_fixed < 30, 15, false > OUT_TYPE;

// Declare channel inputs and outputs
  ac_channel < IN_TYPE > fixed_in;
  ac_channel < OUT_TYPE > fixed_out;
  ac_channel < ac_int < 16, false > > n_sample;

// Instantiate the filter class object
ac_intg_dump < IN_TYPE, ACC_TYPE, OUT_TYPE, N_TYPE, NS, CHN> filter;

// Push values into input channel
// for (...) {
//   ... fixed_in.write(...)
// }

// Call the filter function
filter.run(data_in, data_out, n_sample);
```

2.3. FIR Filters (ac_fir_const_coeffs, ac_fir_load_coeffs, ac_fir_prog_coeffs and ac_fir_reg_share)

Finite impulse response (FIR) filters are used in a wide range of applications, including communications and video applications. They come in a wide range of different architectures, each with a different trade-off in area, performance and power. The source for this FIR filter targets the most common architectures and allows a designer to switch between architectures, if needed. Illustration 4 shows the conventional tapped delay line realization of a shift register FIR filter for N = 3 where N equals the number of taps in the FIR filter.

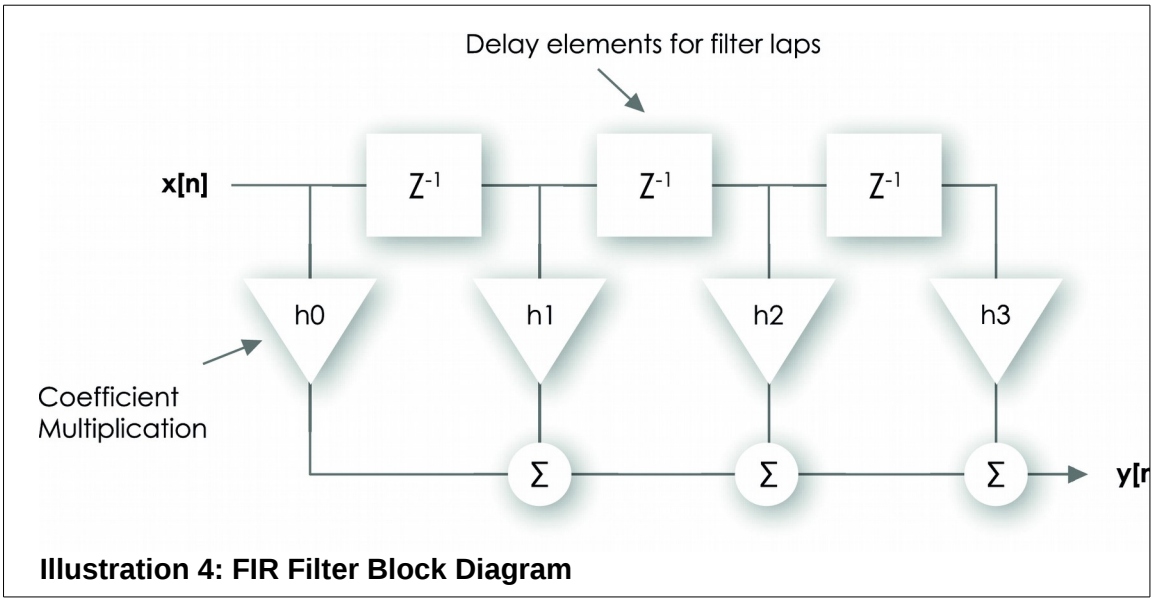


Illustration 4: FIR Filter Block Diagram

2.3.1. Features

- C++ implementation provided.
- Multiple coefficient options available:
  - External Coefficients – Coefficients are programmed outside the design.
  - Constant Coefficients – Coefficients remain constant in the design and multipliers are synthesized into logic rather than utilizing dedicated multipliers.
  - Loadable coefficients- Coefficient are loaded in register/memory internal to the design
- Multiple architectural options available:
  - Direct form
  - Transpose form
  - Register based implementation
  - Circular buffer implementation
  - Rotational shift implementation
- Coefficients optimized for symmetric filters.
- Configurable number of Taps
- Configurable Input/output and accumulation data width

- Hand shake implementation of I/O interfaces
- Serial or parallel architecture options by controlling the loops

2.3.2. Limitations

- Configuration datatypes must support the following operators: =, + and *

2.3.3. Functional Description

The conventional single-rate FIR can be expressed by the difference equation as:

$$y(n) = \sum_{k=0}^{N-1} h(k)\, x(n-k)$$

Where N is the number of taps/filter coefficients. y(n) and x(n) are the output and input sequences respectively. The following sections describe the available architectural configurations for the FIR filter.

2.3.4. Filters based on Shift Register Implementations

Register Based Direct Form Filters

In these filters, delay elements are mapped to registers and are easy to express in C++. These types of filters have an architecture that is highly explorable via loop unrolling and pipelining. Illustration 5 shows a filter where MAC loop is fully rolled and coefficients are external to the design.

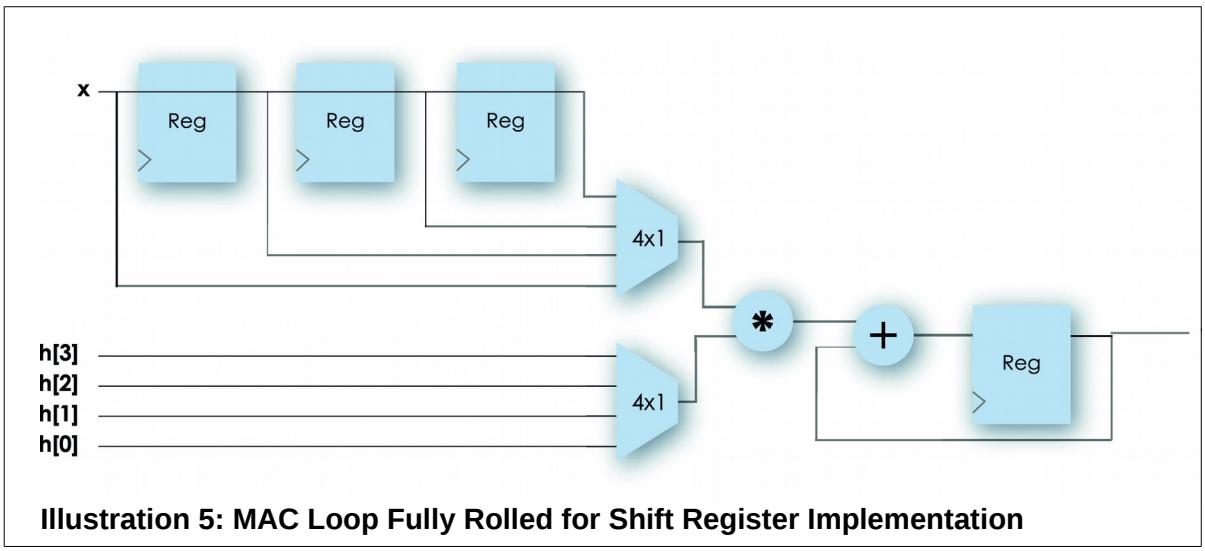


Illustration 5: MAC Loop Fully Rolled for Shift Register Implementation

Because the MAC loop is kept rolled, the multiplier and adder can be shared to compute the filter output. The filter throughput, however, equals four in this case since each multiply and accumulate takes one clock cycle. The MAC loop can be partially or fully unrolled to increase performance. Illustration 6 shows the design with the loops fully unrolled. It is assumed that the clock is slow enough to allow the multipliers and adder tree to be scheduled in the same clock cycle. HLS automatically inserts additional pipeline registers as needed.

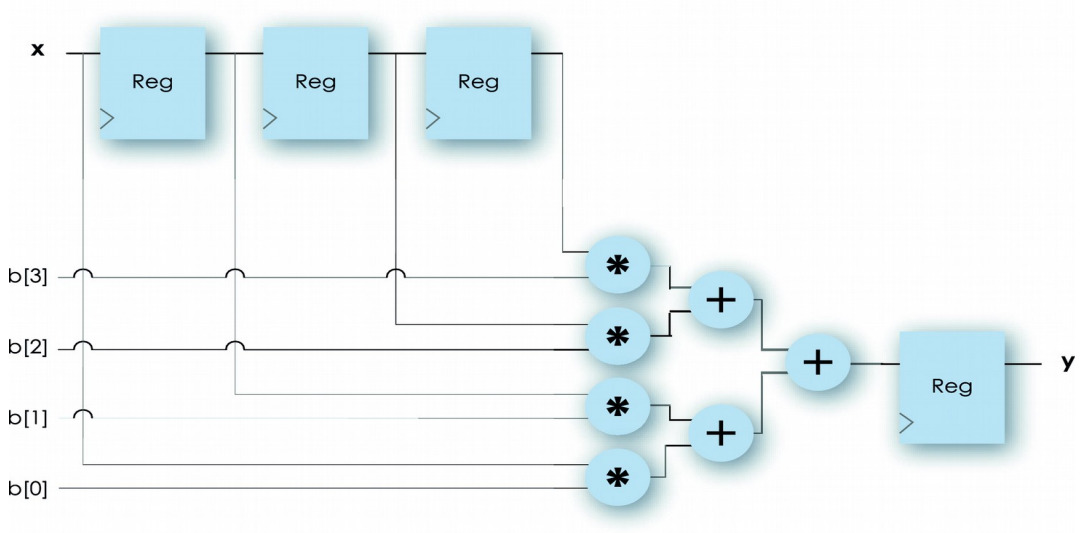


Illustration 6: MAC Loop Fully Unrolled for Shift Register Implementation

Memory based Filters

When number of taps of the filter is significantly large, it may be required to map the shift register into memory to reduce area and power. The most efficient implementation for a memory-based shift register is to use the same approach that is used when writing code for a microprocessor. In this case the read and write pointers are implemented with a circular buffer. The write and read pointer locations move as new data is shifted in, rather than moving the data for each tap. Illustration 7 shows how the read and write pointers circulate. Note that the read pointer runs in the opposite direction as the write pointer.

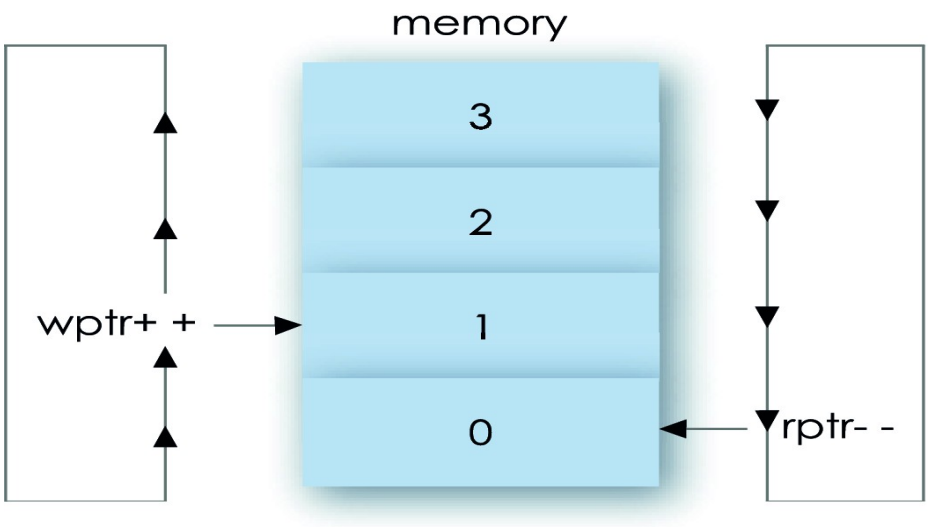


Illustration 7: Read and Write Pointer Circulation for Memory-based Filters

Rotational shift

Rotational shift is an intermediate solution that removes the MUX feeding the multiplier. The MUX becomes a bottleneck as for a large number of taps. Rotation occurs as part of a MAC loop after the +=.

2.3.5. Transposed Form Filters

The previous discussion of FIR filters was based on the direct form FIR, where the tap-values are multiplied against the coefficients and accumulated. For the fully parallel implementations, this results in an adder tree. Some of the disadvantages of an adder tree are that it can have multi-cycle latency for high clock speeds and large number of taps. It may also be more difficult to route due to the large number of interconnects between the adders. An alternate implementation is the transposed form of the FIR, which has single cycle latency independent of the number of taps. However, it can be limited in terms of Fmax due to fan-out. Illustration 8 presents a block diagram that shows the general structure and data flow of a transposed FIR. The following illustration shows that rather than shifting the input data, the partial sums of products of the current input “x” and the coefficients are shifted and accumulated.

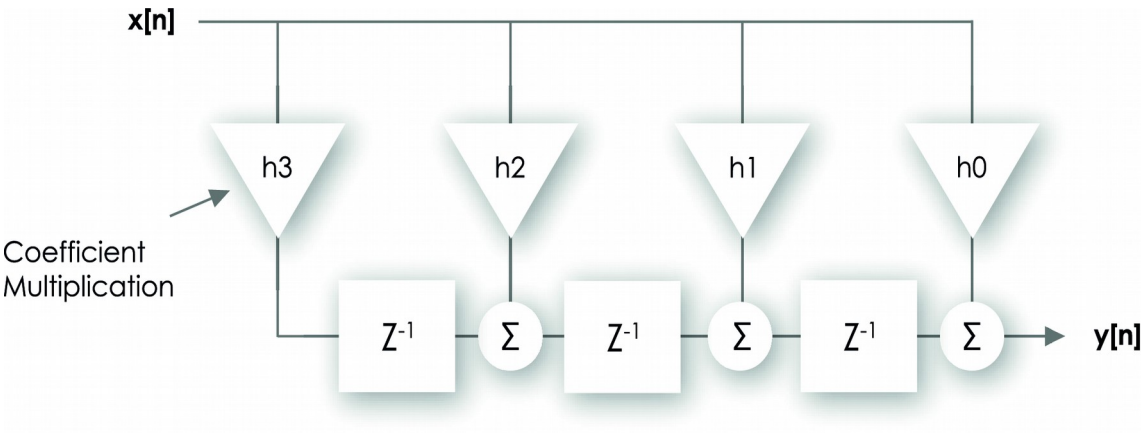


Illustration 8: Transposed Form of FIR Fiilters

2.3.6. Symmetric Filters

The impulse response for many filters possesses significant symmetry where half of the coefficients are exact replica of other half. This symmetry can be exploited to reduce the number of multipliers. Illustration 9 shows the impulse response for a 7-tap symmetric FIR filter.

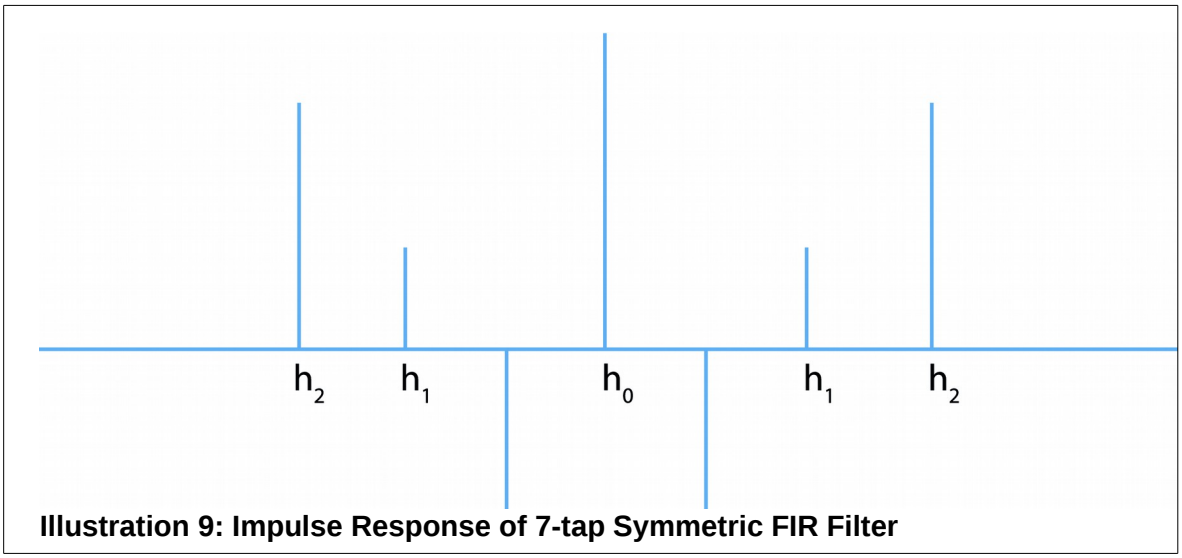


Illustration 9: Impulse Response of 7-tap Symmetric FIR Filter

Instead of implementing the filter as initially shown, a more hardware efficient approach may be used to reduce number of multipliers in the design. This can be achieved by folding the data delay elements and then adding the opposite ends of the delay elements together. We only need to store half the coefficients this way, thanks to the redundancy and repetition of coefficients present in symmetric filters.

In general, the former approach requires N multiplications and (N-1) additions. In contrast, the architecture in the following illustration requires only approximately [N/2] multiplications and N additions. The architecture of such a form is shown in Illustration 10.

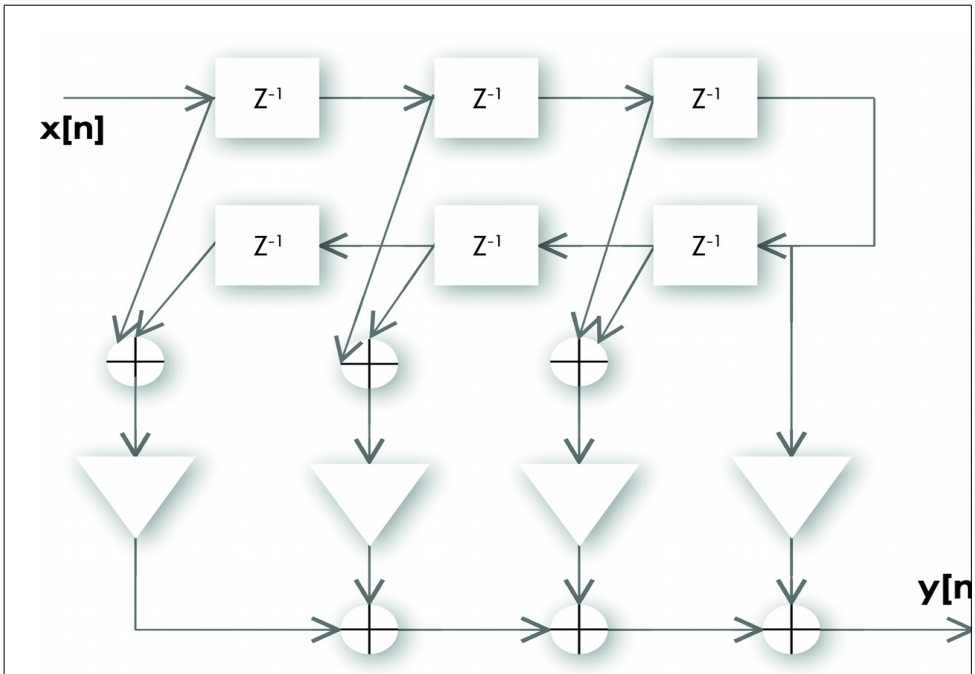


Illustration 10: Folded Form of Symmetric FIR Filter

2.3.7. Coefficient Options

The IP for the FIR filters is classified into separate designs based on how the coefficients are utilized. The following sections discuss the different implementations for FIR filter designs with regards to the model parameters used by each as well as calling each top-level function in the C++ design. The IP for the filter designs is implemented as a class template, using class-based hierarchy. They depends upon the core class for the actual computation, with a member function of the aforementioned class template being synthesized as the top-level interface.

In all cases, different implementations are chosen based on the FTYPE enum which has the following values:

- SHIFT_REG: Direct form shift register implementation.
- C_BUFF: Circular buffer based shift register.
- ROTATE_SHIFT: Rotate Shift Implementation.
- FOLD_EVEN: Symmetric filter with even number of taps.
- FOLD_ODD: Symmetric filter with odd number of taps.
- TRANSPOSED: Transposed form filter.
- FOLD_EVEN_ANTI: Anti-Symmetric filter with even number of taps.
- FOLD_ODD_ANTI: Anti-Symmetric filter with odd number of taps.

2.3.8. Constant Coefficients FIR Filter (ac_fir_const_coeffs)

This filter design is to be used when the coefficients do not change at all. The coefficient values are hard-coded into the design, which allows HLS to perform constant propagation and optimize the multipliers to be constant multipliers.

Model Parameters

A summary of the different template parameters used in the design, as well as a description of each is given in the following table.

Parameter	Description
IN_TYPE	Input data type.
OUT_TYPE	Output data type.
COEFF_TYPE	Coefficient data type.
ACC_TYPE	Accumulator data type.
N_TAPS	Number of taps/coefficients used.
ftype	Parameter from FTYPE enum that helps choose different filter types. Specialized implementations for anti-symmetric filter types are not available for this design.

The *ac_fir_const_coeffs* templatized class is not directly called by the user in their testbench. This is because the class does not have any member data variables that contain coefficient values. Instead, it generates the coefficients based on an array pointer passed to it by another wrapper class, called *ac_fir_const_coeffs_wrapper*, which derives the *ac_fir_const_coeffs* class and contains the *coeffs[]* array as a member variable. This wrapper class is declared with the *#pragma hls_design top* directive, and should be declared by the user in their testbench. Guidelines to do so are given in the Calling the Top-level Function section. The user must make sure that the value for the N_TAPS parameter is the same as the number of elements in the coeffs[] array, to avoid any compile-time errors.

A snippet of both the *ac_fir_const_coeffs* class and is given below:

```
template < class IN_TYPE, class OUT_TYPE, class COEFF_TYPE, class ACC_TYPE,
           unsigned N_TAPS, FTYPE ftype >
class ac_fir_const_coeffs
{
// Code
};
```

Calling the Top-level Function

The following example shows the steps to instantiate the constant coefficients filter and call the top-level member function.

1. Include <ac_dsp/ac_fir_const_coeffs.h>

2. Declare the ac_fir_const_coeffs_wrapper class with the coeffs[] array in it, an example of which shown as follows for a sample coeffs[] array with 5 elements in it.

```
 #pragma hls_design top
 template < class IN_TYPE, class OUT_TYPE, class COEFF_TYPE, class ACC_TYPE,
            unsigned N_TAPS, FTYPE ftype >
 class ac_fir_const_coeffs_wrapper : public ac_fir_const_coeffs<IN_TYPE,
OUT_TYPE,
 COEFF_TYPE, ACC_TYPE, N_TAPS, ftype>
 {
 private:
   const COEFF_TYPE coeffs[N_TAPS] = {1, 2, 3, 4, 5};
public:
   ac_fir_const_coeffs_wrapper() : ac_fir_const_coeffs<IN_TYPE, OUT_TYPE,
COEFF_TYPE, ACC_TYPE, N_TAPS, ftype> (coeffs) {  }
 };
```

3. Define Configuration parameters and I/O datatype in testbench header.

```
enum {
  // Number of taps. Make sure that this number is the same as the number of
  // elements in the wrapper class coeffs array.
  TAPS = 5
};
typedef ac_fixed <32, 16, true> IN_TYPE_TB;
typedef ac_fixed <64, 32, true> OUT_TYPE_TB;
typedef ac_fixed <32, 16, true> COEFF_TYPE_TB;
typedef ac_fixed <64, 32, true> MAC_TYPE_TB;
```

4. Declare ac_channels for input and output in the CCS_MAIN() function with the I/O type above.

```
  ac_channel < IN_TYPE_TB > input;
  ac_channel < OUT_TYPE_TB > output_test;
```

5. Write all input values to the *input* ac_channel. Create an object of the wrapper class with template parameters and call the top function of the filter in the CCS_MAIN() design with the ac_channels passed as arguments to the function. In this case, we are specifically choosing a folded implementation of a symmetric filter with an odd number of coefficients.

```
ac_fir_const_coeffs_wrapper<IN_TYPE_TB, OUT_TYPE_TB, COEFF_TYPE_TB,
MAC_TYPE_TB,
                            TAPS, FOLD_ODD> filter;
filter.run(input, output_test);
```

2.3.9. Loadable Coefficients FIR Filter (ac_fir_load_coeffs)

In this design, the coefficient values are updated, one-per-clock-cycle into the top-level function. The design also has an ac_channel which carries an *ld* flag. This flag is a boolean value. Whenever the ld value is set to true externally, the design knows that it must update the coefficient values.

Model Parameters

A summary of the different template parameters used in the design, as well as a description of each is given in the following table.

Parameter	Description
IN_TYPE	Input data type.
OUT_TYPE	Output data type.
COEFF_TYPE	Coefficient data type.
ACC_TYPE	Accumulator data type.
N_TAPS	Number of taps/coefficients used.
ftype	Parameter from FTYPE enum that helps choose different filter types. Specialized implementations for anti-symmetric filter types are not available for this design.

A snippet for the *ac_fir_load_coeffs* class, showing all the template parameters, is given below:

```
template < class IN_TYPE, class OUT_TYPE, class COEFF_TYPE, class ACC_TYPE, un-
signed N_TAPS, FTYPE ftype >
class ac_fir_load_coeffs
{
// Code
}
```

Calling the Top-level Function

The following example shows the steps to instantiate the loadable coefficients filter and call the top-level member function.

1. Include <ac_dsp/ac_fir_load_coeffs.h>

2. Define Configuration parameters and I/O datatype in the testbench header.

```
enum {
  // Number of taps. Make sure that this number is the same as the number of
  // elements in the wrapper class coeffs array.
  TAPS = 29
};
typedef ac_fixed <32, 16, true> IN_TYPE;
typedef ac_fixed <64, 32, true> OUT_TYPE;
typedef ac_fixed <32, 16, true> COEFF_TYPE;
typedef ac_fixed <64, 32, true> MAC_TYPE;
```

3. Declare ac_channels for input, output, coefficients and loading flag in the CCS_MAIN() function with the I/O types above.

```
    ac_channel<IN_TYPE_TB> input;
```

```
    ac_channel<OUT_TYPE_TB> output_test;
    ac_channel<COEFF_TYPE_TB> coeffs_ch;
    ac_channel<bool> load;
```

4. Load all the coefficient values: Write all coefficient values to the coeffs_ch ac_channel. Create an object of the ac_fir_load_coeffs class with template parameters. Write the value “true” to the load ac_channel. Call the top function of the filter in the CCS_MAIN() design with the ac_channels passed as arguments to the function. Once that is done, write the value “false” to the load ac_channel in order to signal to the design that you are done loading coefficient values. In this case, we are specifically choosing a folded implementation of a symmetric filter with an odd number of coefficients.

```
ac_fir_load_coeffs<IN_TYPE_TB, OUT_TYPE_TB, COEFF_TYPE_TB, MAC_TYPE_TB,
TAPS,
                   FOLD_ODD> filter;
load.write(true);
filter.run(input, coeffs_ch, output_test, load);
load.write(false);
```

5. Write all input values to the *input* ac_channel and coefficient values to the *coeffs_ch* ac_channel. Call the run function again to process all the input values.

```
filter.run(input, coeffs_ch, output_test, load);
```

2.3.10. External Coefficients FIR Filter (ac_fir_prog_coeffs)

This filter design is used when the coefficients are mapped to register or memory outside the design.

Model Parameters

A summary of the different template parameters used in the design, as well as a description of each is given in the following table.

Parameter	Description
IN_TYPE	Input data type.
OUT_TYPE	Output data type.
COEFF_TYPE	Coefficient data type.
ACC_TYPE	Accumulator data type.
N_TAPS	Number of taps/coefficients used.
ftype	Parameter from FTYPE enum that helps choose different filter types. Specialized implementations for anti-symmetric filter types are not available for this design.

A snippet for the *ac_fir_prog_coeffs* class, showing all the template parameters, is given below:

```
template < class IN_TYPE, class OUT_TYPE, class COEFF_TYPE, class ACC_TYPE, un-
signed N_TAPS, FTYPE ftype >
class ac_fir_prog_coeffs
```

```
{
// Code
}
```

Calling the Top-level Function

The following example shows the steps to instantiate the external coefficients filter and call the top-level member function.

1. Include <ac_dsp/ac_fir_prog_coeffs.h>

2. Define Configuration parameters and I/O datatype in the testbench header.

```
const int TAPS = 27;
typedef ac_fixed < 28,  6, true, AC_TRN, AC_WRAP > IN_TYPE;
typedef ac_fixed < 64, 32, true, AC_TRN, AC_WRAP > OUT_TYPE;
typedef ac_fixed < 23,  7, true, AC_TRN, AC_WRAP > COEFF_TYPE;
typedef ac_fixed < 64, 32, true, AC_TRN, AC_WRAP > MAC_TYPE;
```

3. Declare ac_channels for input, output, coefficients in the CCS_MAIN() function with the I/O types above.

```
  ac_channel<IN_TYPE_TB> input;
  ac_channel<OUT_TYPE_TB> output_test;
  ac_channel<COEFF_TYPE_TB> coeffs_ch;
```

4. Create the object for the top-level class and call the top-level function: Write all coefficient values to the coeffs_ch ac_channel. Create an object of the ac_fir_load_coeffs class with template parameters. Call the top function of the filter in the CCS_MAIN() design with the ac_channels passed as arguments to the function. In this case, we are specifically choosing a folded implementation of a symmetric filter with an odd number of coefficients.

```
ac_fir_prog_coeffs<IN_TYPE, OUT_TYPE, COEFF_TYPE, MAC_TYPE, TAPS,
                   FOLD_ODD> filter;
filter.run(input, coeffs_ch, output_test);
```

2.3.11. Register Share FIR Filter (ac_fir_reg_share)

In this design, a shift register is shared between two filter instances.

Model Parameters

A summary of the different template parameters used in the design, as well as a description of each is given in the following table.

Parameter	Description
IN_TYPE	Input data type.
OUT_TYPE	Output data type.

COEFF_TYPE	Coefficient data type.
ACC_TYPE	Accumulator data type.
N_TAPS	Number of taps/coefficients used.
MEM_WORD_WIDTH	Memory Word Width
BLK_SZ	Block Size
BLK_OFFSET	Block Offset
ftype	Parameter from FTYPE enum that helps choose different filter types. Specialized implementations for anti-symmetric filter types are not available for this design.

A snippet for the *ac_fir_reg_share* class, showing all the template parameters, is given below:

```
template < int N_TAPS, class IN_TYPE, class OUT_TYPE, class COEFF_TYPE, class
ACC_TYPE, int MEM_WORD_WIDTH, int BLK_SIZE, int BLK_OFFSET, FTYPE ftype >
class ac_fir_reg_share
{
// Code
}
```

Calling the Top-level Function

The following example shows the steps to instantiate the external coefficients filter and call the top-level member function.

1. Include <ac_dsp/ac_fir_reg_share.h>

2. Define Configuration parameters and I/O datatype in the testbench header.

```
const int TAPS_1 = 48;
const int TAPS_2 = 32;
const int TAPS_3 = 2 * TAPS_1;

#define TYPE_1 SHIFT_REG
#define TYPE_2 FOLD_EVEN_ANTI
#define TYPE_3 FOLD_EVEN

typedef ac_int < 12, true > IN_TYPE;
typedef ac_fixed < 12, 12, true, AC_RND, AC_SAT > OUT_TYPE;
typedef ac_fixed < 16, 1, true, AC_RND, AC_SAT > COEFF_TYPE;
typedef ac_fixed < 33, 17, true, AC_TRN, AC_WRAP > ACC_TYPE;

const int MEM_WIDTH = 16;

const int BLK_SZ_1 = 12;
const int BLK_OFFSET_1 = 0;
```

```
const int BLK_SZ_2 = 4;
const int BLK_OFFSET_2 = 12;

const int BLK_SZ_3 = 16;
const int BLK_OFFSET_3 = 0;
```

3. Declare the input, output, coefficients in the CCS_MAIN() function with the I/O types above.

```
  ac_channel<IN_TYPE> input_d;
  ac_channel<IN_TYPE> input_i;
  ac_channel<OUT_TYPE> output_test_d;
  ac_channel<OUT_TYPE> output_test_i;
  COEFF_TYPE coeffs;
```

4. In the top-level function, declare the registers which are going to be used as shift registers. Create the objects for the top-level classes. Read the inputs one by one and pass them as arguments to the run function.

```
static IN_TYPE reg_d_1[TAPS_3];    /* Create a shift register for filter 1
and                                          filter 3, shared */
static IN_TYPE reg_d_2[TAPS_1];    /* Create a shift register for Filter 2
*/

// Filter objects with pointer of required shift rgister to the constructors
static ac_fir_reg_share < TAPS_1, IN_TYPE, OUT_TYPE, COEFF_TYPE, ACC_TYPE,
MEM_WIDTH, BLK_SZ_1, BLK_OFFSET_1, TYPE_1 > filter_d1(reg_d_1);
static ac_fir_reg_share < TAPS_2, IN_TYPE, OUT_TYPE, COEFF_TYPE, ACC_TYPE,
MEM_WIDTH, BLK_SZ_2, BLK_OFFSET_2, TYPE_2 > filter_d2(reg_d_2);
static ac_fir_reg_share < TAPS_3, IN_TYPE, OUT_TYPE, COEFF_TYPE, ACC_TYPE,
MEM_WIDTH, BLK_SZ_3, BLK_OFFSET_3, TYPE_3 > filter_i(reg_d_1);

input_d_t = input_d.read();   /* Read for Filer 1 FIR */
input_i_t = input_i.read();   /* Read for Filter 2 */
filter_d1.run(input_i_t, coeffs, output_d_t);          /*Call MAC for fil-
ter 1*/
filter_d2.run(input_d_t, coeffs, output_e_t);          /*Call MAC for fil-
ter 1*/
output_i.write(output_d_t);
output_d.write(output_e_t);

input_i_t = input_i.read();   /* Read for filter 3 */
filter_i.run(input_i_t, coeffs, output_e_t);           /*Call MAC for fil-
ter 3*/
output_i.write(output_e_t);
```

2.4. 1-D Moving Average (ac_mv_avg)

A 1-D moving average filter replaces each data point with the average of the neighboring data points defined within the span. The moving average filter can be thought of as a low-pass filter, but this filter is designed to work with finite data sets.

2.4.1. Features

- Streaming input and output
- Configurable number of input data samples
- Configurable window type clip or mirror
- Configurable Input/Output, accumulation and weights data width
- Configurable window size
- Weights are implemented as constants and provided as const array in the constructor
- Hand shake implementation of I/O interfaces

2.4.2. Limitations

- The window span must be odd
- The window span must be less than the number of samples
- The data point to be smoothed must be at the center of the span
- Only clipping and mirroring are supported at the edge of the data

2.4.3. Functional Description

A moving average filter is equivalent to low pass filtering with the response of the smoothing given by the difference equation:

$$y_S(i) = \frac{1}{2N+1}\left(C_0 y(i+N) + C_1\left(y(i+N-1) + \ldots\ldots\ldots + C_{2N} y(i-N)\right)\right)$$

where YS(i) is the smoothed value for the ith data point, N is the number of neighboring data points on either side of YS(i), and 2N+1 is the window size and Ci is the weights associated with the data points. The previous equation resembles symmetric moving average filter where data to be averaged is at the center of the window.

Data points at the ends of the stream can not be smoothed because end points are not defined. At the end points either clipping or mirroring can be used as shown in the following illustration:

CIC Filter as a Moving Average Filter

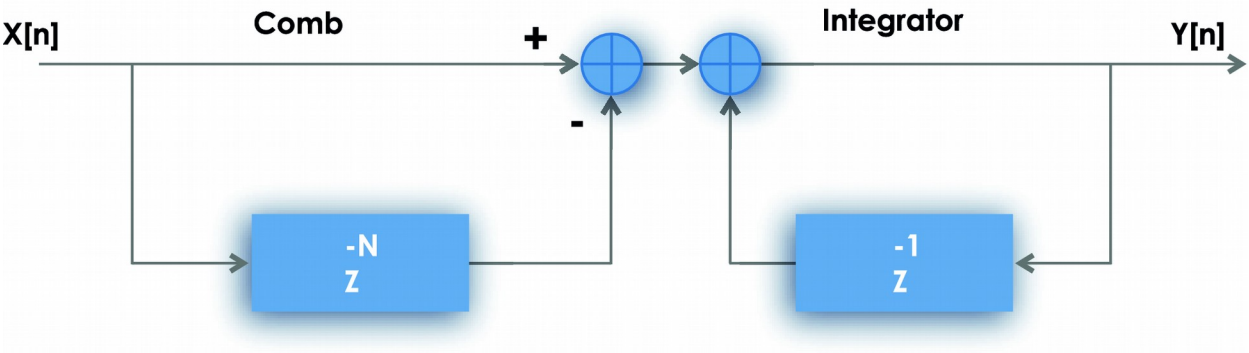
Recursive form of moving average filter can be implemented as CIC filter. Difference equation of a moving average filter can also be written without scaling as:

$$y[n] = \sum_{k=0}^{N-1} x[n-k]$$

The previous equation can also be expressed in recursive form as

$$y[n] = x[n] - x[n-N] + y[n-1]$$

This difference equation resembles CIC filter with number of stages equals to 1 and differential delay equals to N. A following illustration shows the blck diagram of this implementation:



From the previous equation, the comb and integrator can also be arranged in decimation format.

***Note**: The CIC filter can be used as a moving average filter, however, the scaling should be done outside the design.*

2.4.4. Model Parameters / Function Signature

Filter operates on finite set of data. The maximum number of sample data in the bursts has to be configured and number of samples in the current burst need to be provided as an input.

The template signature for the filter classes is shown below:

```
template < unsigned MAX_SAMPLE, unsigned TAPS, ac_window_mode WIN_TYPE,
           class IN_TYPE, class OUT_TYPE, class ACC_TYPE,
```

```
          class COEFF_TYPE, class S_TYPE >
class ac_mvg_avg
{
// code
}
```

The following are the class template parameters for the filter:

Parameter	Description
MAX_SAMPLE	Maximum number of sample in the input data burst
TAPS	Window size (must be odd)
WIN_TYPE	Type of windowing applied (AC_MIRROR, AC_CLIP)
IN_TYPE	Data type for input samples
OUT_TYPE	Data type for filter outputs
ACC_TYPE	Data type for internal accumulator
COEFF_TYPE	Data type of the filter coefficients
S_TYPE	Data type of the n_samples input

The prototype of the top-level member functions for both the interpolation and decimation filters is given below:

```
void run(ac_channel <IN_TYPE> &data_in, ac_channel <OUT_TYPE> &data_out,
         ac_channel <S_TYPE> &n_sample)
{
// code
}
```

2.4.5. Synthesis Considerations

specifies the different types of loops in the design and how they are treated.

Loops

Loops	Description
SAMPLE Loop	iterates for number equal to MAX_SAMPLE configured and breaks when number of samples as given in the port is read. The loop can be fully unrolled.
ACC loop	multiplies data with the weights and accumulates. Loop iterates for window size and can be fully unrolled.

2.4.6. Usage Example

The following steps show an example of how to instantiate this filter. Note that a wrapper class is used to contain the coefficients for the filter.

```
// Include the filter header
```

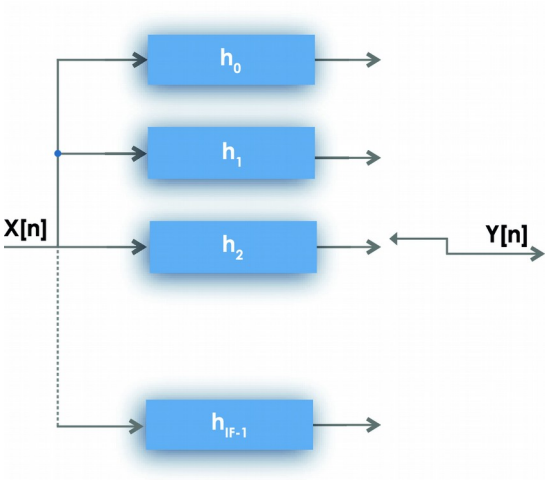
```
#include <ac_dsp/ac_mv_avg.h>

// Define configuration parameters
  enum {
    CONFIG     = 50,
    MAX_INPUT1 = 500,
    MAXSZ      = 1024,
    TAPS       = 7,
    AC_TYPE_TB = 0
  };

typedef ac_fixed < 16, 2, true > IN_TYPE;
typedef ac_fixed < 16, 2, true > OUT_TYPE;
typedef ac_fixed < 16, 2, true > ACC_TYPE;
typedef ac_fixed < 16, 2, true > COEFF_TYPE;
typedef ac_int < 10, false > S_TYPE;

// Create a wrapper class to contain the coefficient array
#pragma hls_design top
  template < int MAX_SAMPLE, int TAPS, ac_window_mode WIN_TYPE, class
 IN_TYPE,
             class OUT_TYPE, class ACC_TYPE, class COEFF_TYPE, class S_TYPE
 >
  class ac_mv_avg_wrapper : public ac_mv_avg< MAX_SAMPLE, TAPS, WIN_TYPE,
 IN_TYPE,
             OUT_TYPE, ACC_TYPE, COEFF_TYPE, S_TYPE >
{
public:
  // Coefficients here:
  const COEFF_TYPE coeffs[TAPS] = { .5, .125, .5, 1, 1, .25, 1 };

  // Pass a pointer to the const coeffs array as a parameter in the
               ac_fir_const_coeffs constructor.
  ac_mv_avg_wrapper() : ac_mv_avg< MAX_SAMPLE, TAPS, WIN_TYPE, IN_TYPE,
OUT_TYPE,    ACC_TYPE, COEFF_TYPE, S_TYPE > (coeffs) {  }
};

// ...
// Main

// Declare channels for inputs and outputs
ac_channel < IN_TYPE > data_in;
ac_channel < OUT_TYPE > data_out;
ac_channel < S_TYPE > n_sample;
```

```
// Instantiate the filter object (derived class)
ac_mv_avg_wrapper < MAXSZ,      TAPS,
                    AC_TYPE,    IN_TYPE,
                    OUT_TYPE,   ACC_TYPE,
                    COEFF_TYPE, S_TYPE > filter;
// Loop...
{
  // Push values into input channel
  // for (...) {
  //   ... data_in.write(...)
  // }

  // Call the filter function
  filter.run(data_in, data_out, n_sample);

  // Process outputs
  // while (fixed_out.available(1)) {
  //   ... data_out.read();
  // }
}
```

2.5. Poly-Phase Interpolation (ac_poly_intr)

Some digital applications require an increase in input sample rate. This is achieved by inserting zeros between the samples and low pass filtering the resulting data stream. An efficient way to do this is to decompose the low pass filter using a poly-phase structure.

The poly-phase filter implements an efficient one to IF (Interpolation Factor) interpolation filter, where sample rate of the input stream is padded with IF-1 zeros. The following illustration shows structure of a poly phase interpolation filter.



Coefficients C0, C1, C2, - - - - - - - - - -CN-1 of the N tap filter are mapped into sets of coefficients according to the following equation.

$$h_{i(n)} = (C_i + IF * r) \;\; where\; i = 0,1 \ldots \ldots \ldots IF - 1 \; and \; r = 0,1, \ldots \ldots \ldots N/IF - 1$$

Each input sample is accessed by all sub filters in parallel and for each input sample, one output sample from each sub filters is delivered to the output.

2.5.1. Features

- C compatible class
- Configurable Number of the Taps
- Configurable Input Data Type (integer or fixed with configurable width)
- Configurable Output Data Type (integer or fixed with configurable width)
- Configurable accumulator Data Type (integer or fixed with configurable width)
- Configurable Integer Interpolation Factor
- Multiple implementation options, serial or parallel by Catapult.
- Coefficient optimization for symmetric filters using symmetric pair's technique

2.5.2. Limitations

- Configuration datatypes must support the following operators: =, +,

2.5.3. Functional Description

When an interpolation filter is decomposed in poly-phase structure, not all the resultant sub-filters remain symmetric. However, by arranging the coefficients of the sub filters, symmetry can be achieved. The symmetric pair technique observes when coefficients of two non-symmetric sub filters are added and subtracted; the resulting sub filters exhibit symmetry.

The following example of 15 tap and interpolated by 3 filter demonstrates this technique. {a, b, c, d, e, f, g, h, g, f, e, d, c, b, a} produce the following sub filters:

$h_0 = a, d, g, f, c$

$h_1 = b, e, h, e, b$

$h_2 = c, f, g, d, a$

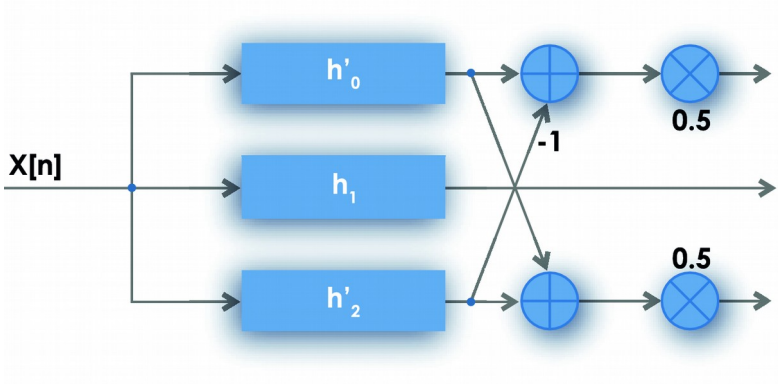
Sub filters h0 and h2 are not symmetric. Applying the symmetric pair's technique produces the following sub filters:

$h'_0 = a+c, d+f, g+g, f+d, c+a$

$h_1 = b, e, h, e, b$

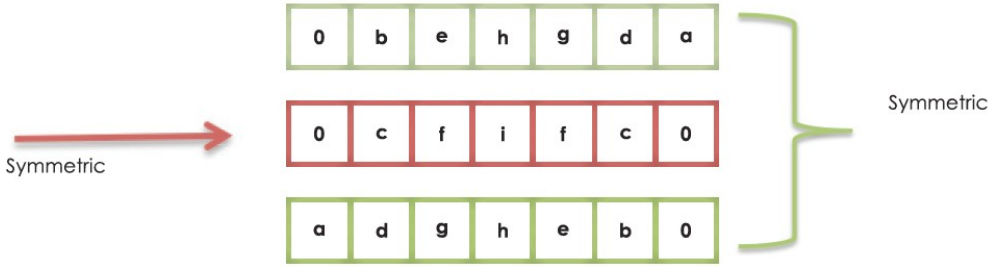
$h'_2 = c-a, f-d, g-g, d-f, a-c$

Now both h'0 and h'2 are symmetric and negative symmetric respectively. The filter can now be implemented utilizing symmetry, giving the associated resource savings. The output from sub filters h0 and h2 should be added and subtracted and then scaled by a factor of 0.5 to produce the original filter output. The following illustration shows the resulting structure.



Coefficient Padding

Decomposing a FIR filter may result in in a sub filter which is not fully populated with coefficients, if interpolation factor is not integer multiple of number of taps. In such cases zero padding is required and it may be padded either in the end or anywhere to exploit symmetric pair technique. The reorganization of the filter coefficients results in a change in the phase response of the filter. In a filter with 17 taps and interpolate by 3, padding a zero in between the coefficients would be required to align the coefficients such that symmetry can be exploited. This results in a smaller implementation, but with different phase response for the filter. If such a change can not be accommodated in the application system, user can either force non-symmetric structure implementation or make use of the extra coefficients to use symmetric pair technique. The following illustration shows this example.



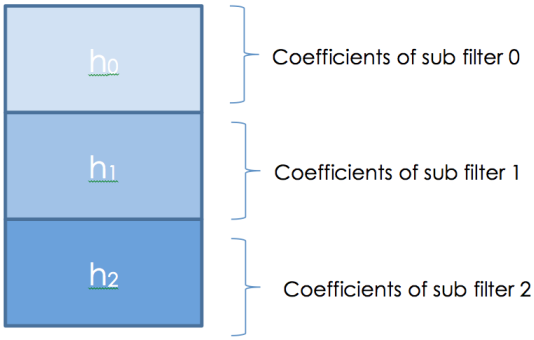
2.5.4. Arrangement of Input Coefficients

Design requires coefficient to be arranged in sub filter order as

```
{h0(n), h1(n), h2(n)……..hIF-1(n) }
```

where hi(n) is as defined in previously mentioned equation

When utilizing symmetry of the filter, as previously discussed, modified coefficients should be provided and should be arranged in sub-filter order.



Symmetric sub filters do not require all coefficients as input. Even symmetric sub-filters requires only NATPS/2 coefficients, while odd symmetric sub-filters require NTAPS/2+1 coefficients and non-symmetric filters require NTAPS coefficients where

- NTAPS is number of taps of a sub-filter.
- COEFFSZ parameter of the design specifies size of the coefficient array
    - For non-symmetric Filters COEFFSZ = NTAPS*IF
    - For even symmetric filters COEFFSZ = NTAPS*IF/2
    - For odd symmetric Filters COEFFSZ = (NTAPS/2+1)*IF

sign[ ] and corr[ ] inputs

The *sign[ ]* and *corr[ ]* inputs of the design are required when symmetry of the sub-filters is utilized by adding and subtracting coefficients of non-symmetric sub-filters. These inputs are array of size equal to the number of sub-filters (interpolation factor).

If the filter will not be symmetric, then two pointers need to be declared and passed to the function. The pointers are not used and will be optimized away:

```
bool *sign;
ac_int<8,false> *corr;
```

sign[IF] input indicates symmetric nature of sub-filters, “true” defines symmetric and “false” defines anti-symmetric.

The *corr[IF]* input defines relation between the sub-filters, that is, which sub-filters are added and subtracted.

For example, as described in the previous example sub-filter 0 and sub-filers 2 used symmetric pair’s technique and resultant h’0 and h’2 are symmetric and anti-symmetric respectively.

Values and respective definitions of sign[IF] and corr[IF] arrays are shown in the following table.

`sign[IF] = {1,1,0}`	Sub-filter 0 is symmetric Sub-filter 1 is symmetric Sub-filter 2 is anti-symmetric
`corr[IF] = {2,1,0}`	Sub-filter 0 is modified by sub-filter 2 Sub-filter 1 is unmodified Sub-filter 2 is modified by sub-filter 0

2.5.5. Model Parameters / Function Signature

The template signature for the filter classes is shown below:

```
template < class IN_TYPE, class COEFF_TYPE, class ACC_TYPE, class OUT_TYPE,
           class STR_CTRL_TYPE, class STR_COEFF_TYPE,
           int NTAPS, int COEFFSZ, int IF, FTYPE ftype >
class ac_poly_intr
{
// code
}
```

The following are the class template parameters for the filter:

Parameter	Description
IN_TYPE	Data type for input samples
COEFF_TYPE	Data type of the filter coefficients
ACC_TYPE	Data type for internal accumulator
OUT_TYPE	Data type for filter outputs
STR_CTRL_TYPE	Data type of the control channel input
STR_COEFF_TYPE	Data type of the coefficient stream input
NTAPS	Number of taps of the poly-phase sub filters
COEFFSZ	Number of coefficients in the input array / memory: • For non-symmetric Filters COEFFSZ = NTAPS*IF • For even symmetric filters COEFFSZ = NTAPS*IF/2 • For odd symmetric Filters COEFFSZ = (NTAPS/2+1)*IF
IF	Interpolation factor
ftype	Specifies filter type: • EVEN_SYMMETRIC – Symmetric sub-filters with even number of taps. • ODD_SYMETRIC – Symmetric sub-filters with odd number of taps. • NON_SYMETIRC - Non symmetric sub-filters

The prototype of the top-level member functions for the filters is given below:

```
void run(ac_channel <IN_TYPE> &data_in, ac_channel <OUT_TYPE> &data_out,
         ac_channel <STR_CTRL_TYPE> &ctrl_st,
         ac_channel <STR_COEFF_TYPE> &coeffs_st)
{
// code
}
```

2.5.6. Synthesis Considerations

The following table specifies the different types of loops in the design and how they are can be utilized.

Loops	Description
SHIFT_REG	Implements shift register and can be fully unrolled
INTR_F	Interpolation loop
MAC_x	Multiply and Accumulate loop, can be fully rolled or unrolled

MAC loop control in the design is optional. If input coefficient array is mapped to registers, loop can be partially unrolled however if array is mapped to memory loop can be either fully rolled or fully unrolled. When fully unrolled data width of the memory should be increased by factor of loop iteration. Based on the requirements design can be pipelined with initiation interval with maximum throughput of 1.

2.5.7. Usage Example

The following steps show an example of how to instantiate a filter and call the run function:

1. Define Configuration parameters and data types. The following code shows an example of even symmetric filter

```
// Include the filter header
#include <ac_dsp/ac_poly_intr.h>

const int NTAPS = 16;
const int IR = 3;
const int COEFFSZ = 24;
#define TYPE EVEN_SYMMETRIC

typedef ac_fixed < 16, 1, true, AC_RND > IN_TYPE;
typedef ac_fixed < 17, 1, true, AC_RND > COEFF_TYPE;
typedef ac_fixed < 32, 1, true, AC_RND > ACC_TYPE;
typedef ac_fixed < 32, 1, true > OUT_TYPE;
typedef struct {
  COEFF_TYPE coeffs[COEFFSZ];
} STR_COEFF_TYPE;
```

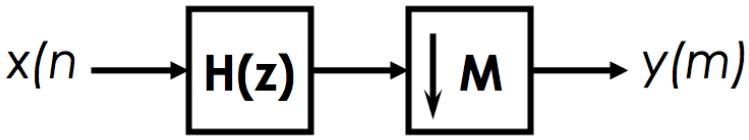
```
typedef struct {
  bool sign[IR];
  ac_int < 8, false > corr[IR];
} STR_CTRL_TYPE;
```

2. Declare ac_channels for input, output, coefficients. Create an object of the class with template parameters and call the run function.

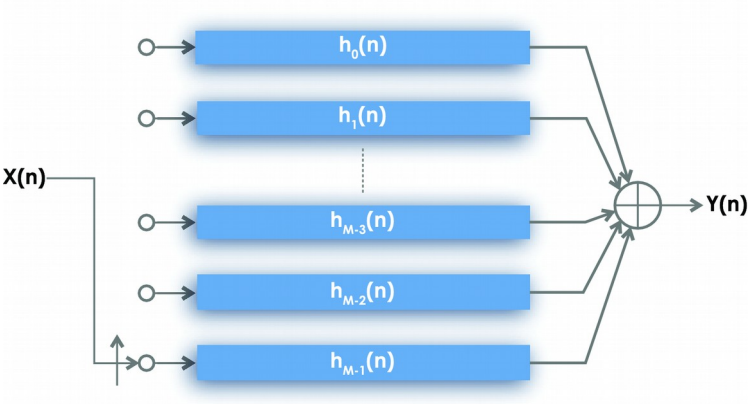
```
  ac_channel < IN_TYPE >         &in;
  ac_channel < OUT_TYPE >        &out;
  ac_channel < STR_CTRL_TYPE >   &ctrl_st;
  ac_channel < STR_COEFF_TYPE > &coeffs_st;
  static ac_poly_intr < IN_TYPE,  COEFF_TYPE,    ACC_TYPE,
                        OUT_TYPE, STR_CTRL_TYPE, STR_COEFF_TYPE,
                        NTAPS,    COEFFSZ,       IR, TYPE > filter_inst;
  filter_inst.ac_polyIntr_top(in, out, ctrl_st,coeffs_st);
```

2.6. Poly-Phase Decimation (ac_poly_dec)

Some digital applications require a decrease in the input sample rate. Down sampling by an integer factor, M, can be achieved by two step process. Reduce high-frequency components with a low pass filter and then decimate the output sequence, keeping only every Mth output sample.



An efficient way to do this is to decompose the low pass filter using a poly-phase structure. The following illustration shows the poly-phase decimation filter which implements the computationally efficient M-to-1 poly-phase decimating filter.



A set of N prototype filter coefficients a0, a1, a2, - - - - aN-1 is mapped to the M poly-phase sub-filters h0(n), h1(n), h2(n) - - - - - - - hM-1(n)according to following equation

$$h_i(r) = a(i + Mr) \quad i = 0, 1, \ldots, M-1 \quad r = 0, 1, \ldots, \frac{N}{M}$$

Input samples x(n) are delivered to the poly-phase segments via an input commutator which starts at the segment index and decrements to index 0. After the commutator has executed one cycle and delivered M input samples to the filter, one input to the one sub filter and so on, a single output is taken as the summation of the outputs from the poly-phase segments. The output sample rate is M times lower than the sample rate of the input data stream.

2.6.1. Features

- C compatible class
- Configurable Number of the Taps
- Configurable Input Data Type (integer or fixed with configurable width)
- Configurable Output Data Type (integer or fixed with configurable width)
- Configurable accumulator Data Type (integer or fixed with configurable width)
- Configurable Integer Decimation Factor
- Multiple implementation options, serial or parallel by Catapult.

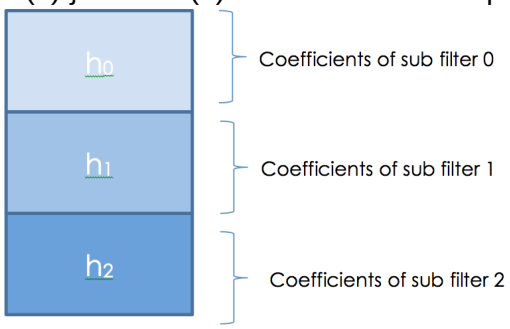
2.6.2. Limitations

- Configuration datatypes must support the following operators: =, +, *

2.6.3. Functional Description

Design requires coefficient to be arranged in sub filter order as

{h0(n), h1(n), h2(n)……..hIF-1(n) }where hi(n) is as defined in the previous equation



2.6.4. Model Parameters / Function Signature

The template signature for the filter classes is shown below:

```
template < class IN_TYPE, class COEFF_TYPE, class STR_COEFF_TYPE,
```

```
          class ACC_TYPE, class OUT_TYPE, int NTAPS, int DF >
class ac_poly_dec
{
// code
}
```

The following are the class template parameters for the filter:

Parameter	Description
IN_TYPE	Data type for input samples
COEFF_TYPE	Data type of the filter coefficients
STR_COEFF_TYPE	Data type of the coefficient stream input
ACC_TYPE	Data type for internal accumulator
OUT_TYPE	Data type for filter outputs
NTAPS	Number of taps of the poly-phase sub filters
DF	Decimation factor

The prototype of the top-level member functions for the filters is given below:

```
void run(ac_channel <IN_TYPE> &data_in, ac_channel <OUT_TYPE> &data_out,
         ac_channel <STR_COEFF_TYPE> &coeffs_st)
{
// code
}
```

2.6.5. Synthesis Considerations

The following table specifies the different types of loops in the design and how they ar e can be utilized.

Loops	Description
SHIFT	Implements shift register and can be fully unrolled
DF	Decimation loop
MAC	Multiply and Accumulate loop, can be fully rolled or unrolled

MAC loop control in the design is optional. If input coefficient array is mapped to registers, loop can be partially unrolled however if array is mapped to memory loop can be either fully rolled or fully unrolled. When fully unrolled data width of the memory should be increased by factor of loop iteration. Based on the requirements design can be pipelined with initiation interval with maximum throughput of 1.

2.6.6. Usage Example

The following steps show an example of how to instantiate a filter.

1. Define Configuration parameters and data types.

```
// Include the filter header
#include <ac_dsp/ac_poly_dec.h>

const int NTAPS = 26;
const int DF = 2;

typedef ac_fixed < 16, 1, true, AC_RND, AC_SAT > IN_TYPE;
typedef ac_fixed < 16, 1, true, AC_RND, AC_SAT > COEFF_TYPE;
typedef ac_fixed < 36, 4, true, AC_RND, AC_SAT > ACC_TYPE;
typedef ac_fixed < 15, 1, true , AC_RND, AC_SAT> OUT_TYPE;
typedef struct {
  COEFF_TYPE coeffs[NTAPS * DF];
} STR_COEFF_TYPE;
```

2. Declare ac_channels for inputs, outputs and coefficients. Create an object of the class with template parameters and call the run function.

```
ac_channel < IN_TYPE > &in;
ac_channel < OUT_TYPE > &out;
ac_channel < STR_COEFF_TYPE > &coeffs_st;
static ac_poly_dec < IN_TYPE,  COEFF_TYPE, STR_COEFF_TYPE,
                     ACC_TYPE, OUT_TYPE,   NTAPS, DF> filter_inst;
filter_inst.run(in, out, coeffs_st);
```

Chapter 3: Fast Fourier Transform (FFT) Functions

The *ac_dsp* library include the following FFT implementations:

- Radix-2 (DIF) Inplace
- Radix-2/2 ^{2}Mix (DIF) Single Delay Feedback
- Radix-2^{2} (DIF) Single Delay Feedback
- Radix-2^{X} (DIF) In-place with Block Floating Point
- Radix-2^{X} (DIF) Dynamic In-place
- Radix-2^{X} (DIF) with Automated Radix Mixing
- Radix 2 (DIF) Single Delay Feedback
- Radix-2 (DIT) In-place
- Radix-2 (DIT) Single Delay Feedback

We make the following assumptions while using these FFT implementations:

- The number of FFT points as well as the radix is always a power of two. For more details on the number of FFT points, please refer to Details and Limitations.
- The inputs are always appropriately decimated as per the architecture of the design. For instance, the DIT architectures will have their inputs decimated in time and outputs appear in natural order. DIF designs will assume that the input appears in natural order. Some of the DIF designs allow the user to configure whether the output appears in a natural or bit-reversed order through a template parameter.
- The input and output are passed to the design in the form of *ac_channel* streaming interfaces. The design itself uses class-based hierarchy for its implementation with the top level member function in each class being implemented as a separate hierarchical block. The core class, which contains all the computation, always has its member functions inlined.
- Scaling by half occurs at the output of each FFT stage in the design. For more details on this and the precision of each stage please refer to Details and Limitations.

For a description of the features, testing and architecture options for these FFT implementations, refer to Overview of FFT Implementations.

3.1. Overview of FFT Implementations

The following section provides a general overview of the implementations of the FFT functions in the *ac_dsp* library.

- Functional Description of the FFT Function
- Catapult Architectural Exploration Options
- Details and Limitations
- Testing and Verification

3.1.1. Functional Description of the FFT Function

The FFT is a computationally efficient algorithm for computing a Discrete Fourier Transform (DFT) of sample sizes that are a positive integer power of 2. The DFT X (k) of a sequence x (n) as well as the Inverse Discrete Fourier Transform (IDFT) are defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n).e^{-\frac{j2\pi kn}{N}}$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k).e^{\frac{j2\pi kn}{N}}$$

Where N is the number of FFT points. The two equations given above are the reverse of each other. The top equation is for DFT and the bottom equation is for IDFT.

3.1.2. Catapult Architectural Exploration Options

Different RTL implementations of CatWare can be explored using catapult “Architectural constraint editor”.

NOTE*: The following screenshots are examples from the Radix-2 (DIF) In-place function. Other FFT function may have slightly different names for the classes, functions and loops but the general concept is the same.*

- Unroll and/or pipelining the nested loop structure. User can explore various parallelism of FFT hardware using loop unrolling. The “main” loop for in-place FFTs generally cannot be pipelined with an II of 1, but the “main” loop of SDF FFTs can generally be fully pipelined (default setting for loops are shown in Illustration 11: Loop Options).

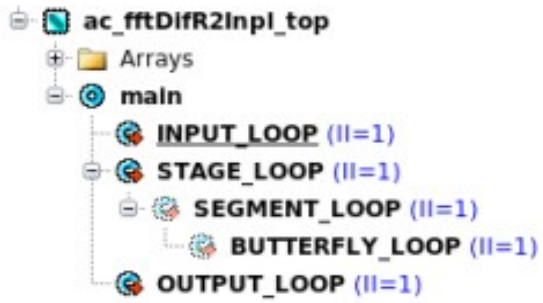


Illustration 11: Loop Options

- Choose to map arrays to registers or RAM memory.

Illustration 12: Array Mapping Options

- Map Constant Phase Factors to different types of ROMs

Illustration 13: ROM Mapping Options

3.1.3. Details and Limitations

- ***Configurable Number of FFT Points***: IP supports FFT points from 2 to 4096* in integer powers of 2.
- ***Configurable Bit Precision***: Bit precision for input and output can be configured via class template arguments. Internal bit precision will be optimally calculated based on the configurations. The input and output precision will the be same as the stage precision.
- ***Scaling by Half for All Stages***: The designs are hard-coded to scale by half on all stage outputs in order to prevent overflow. The combined scaling that occurs from input to output as a result is (1/N), where N is the number of FFT points. If the user does not wish to scale by half on the output, they can refer to the code comments to help them modify the code appropriately.
- ***Simple Addressing Scheme***: Use of simple logical expressions for address generation reduces area and improves maximum supported frequency.

- ***Class-Based Hierarchy:*** *All the designs use class-based hierarchy for their implementation. The top-level block is synthesized from a C-compatible function within the class, as mentioned earlier. The core classes have their member functions inlined.*

- ***No need for Additional Memory Bank for Reshuffling***: Reshuffling mechanism is performed during Butterfly operation; the need for an extra memory bank is eliminated.

- ***Constant Twiddle Implementation***: Pre-calculated twiddle factors are used. These can be mapped to a register bank or a ROM. For more details, please refer to Catapult Architectural Exploration Options.

- ***Configurable Twiddle Factor precision:*** This is also passed as a template parameter.

- Hardware Efficient computation.

- Configurable parallelism.

* ***Note*** - *Value can be further increased if required.*

The following list describes the limitations of the FFT functions:

- Separate read/write ports or a dual memory port needed: Some of the architectures require separate Read-Write or Dual Memory Ports to simultaneously read and write butterfly inputs and outputs. Catapult will fail to pipeline if a single port memory is used.

- Block supports SISO (Single input- single output) only: Block is limited by SISO. There is no MIMO support.

- The FFT inputs and outputs can currently only be of signed *ac_complex<ac_fixed>* type.

3.1.4. Testing and Verification

Verification of IPs goes through number of phase, broadly we can divide testing and verification into two Phases:

3. CatWare vs. Matlab Output (Phase 1)

4. Untimed C++ vs. RTL Output (Phase 2)

CatWare vs. Matlab Output (Phase 1)

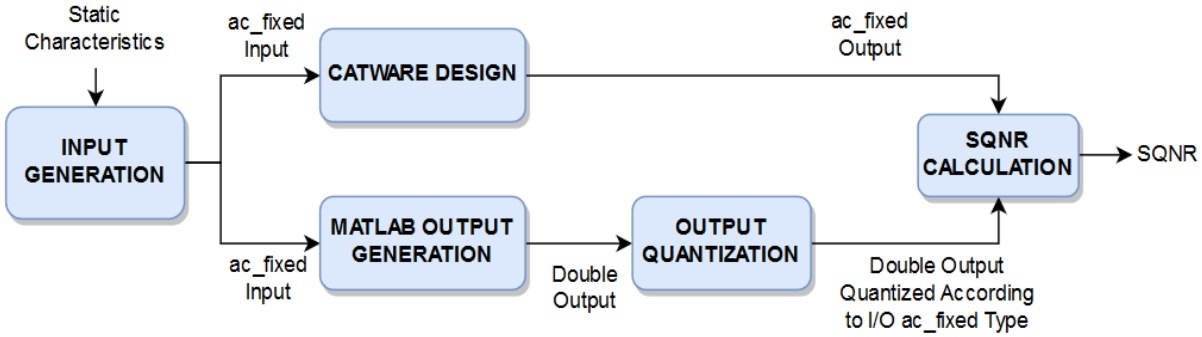


Illustration 14: CatWare vs. Matlab

This phase uses a C++ design and tests it against the reference fft function in MATLAB. A mixed-tone sine wave input is generated along with the corresponding reference output in MATLAB, based on the static characteristics as defined in the C++ testbench header (one notable exception to this is the dynamic inplace FFT, which depends upon both static and dynamic characteristics for reference I/O generation). The C++ testbench exercises the Catware Design and also compares the CatWare output to the reference output after quantizing the latter, while making sure that the SQNR doesn't drop below a pre-defined threshold.

Untimed C++ vs. RTL Output (Phase 2)

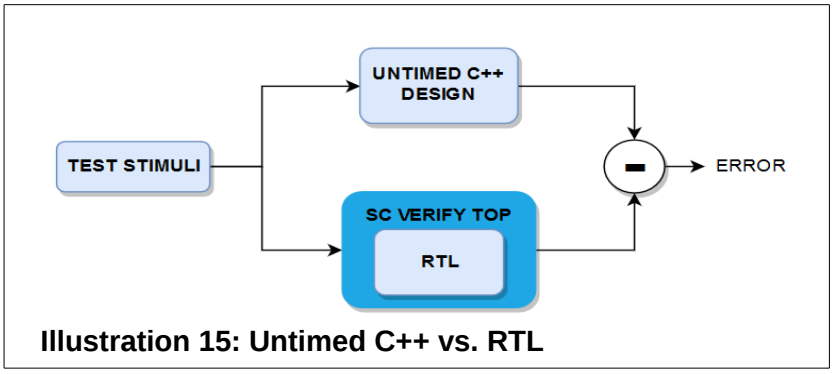


Illustration 15: Untimed C++ vs. RTL

In this phase Output of RTL and Untimed C++ will be compared bit-wise in SCVerify.

3.2. Radix-2 (DIF) Inplace

This Fast Fourier Transform (FFT) implementation utilizes a Radix 2 FFT with Decimation in Frequency (DIF) In-place architecture. The FFT has a parameterized and high performance implementation in C++. The FFT is synthesizable by Catapult to produce the corresponding RTL implementation constrained by the user. Illustration 16 shows the system block diagram for this function.

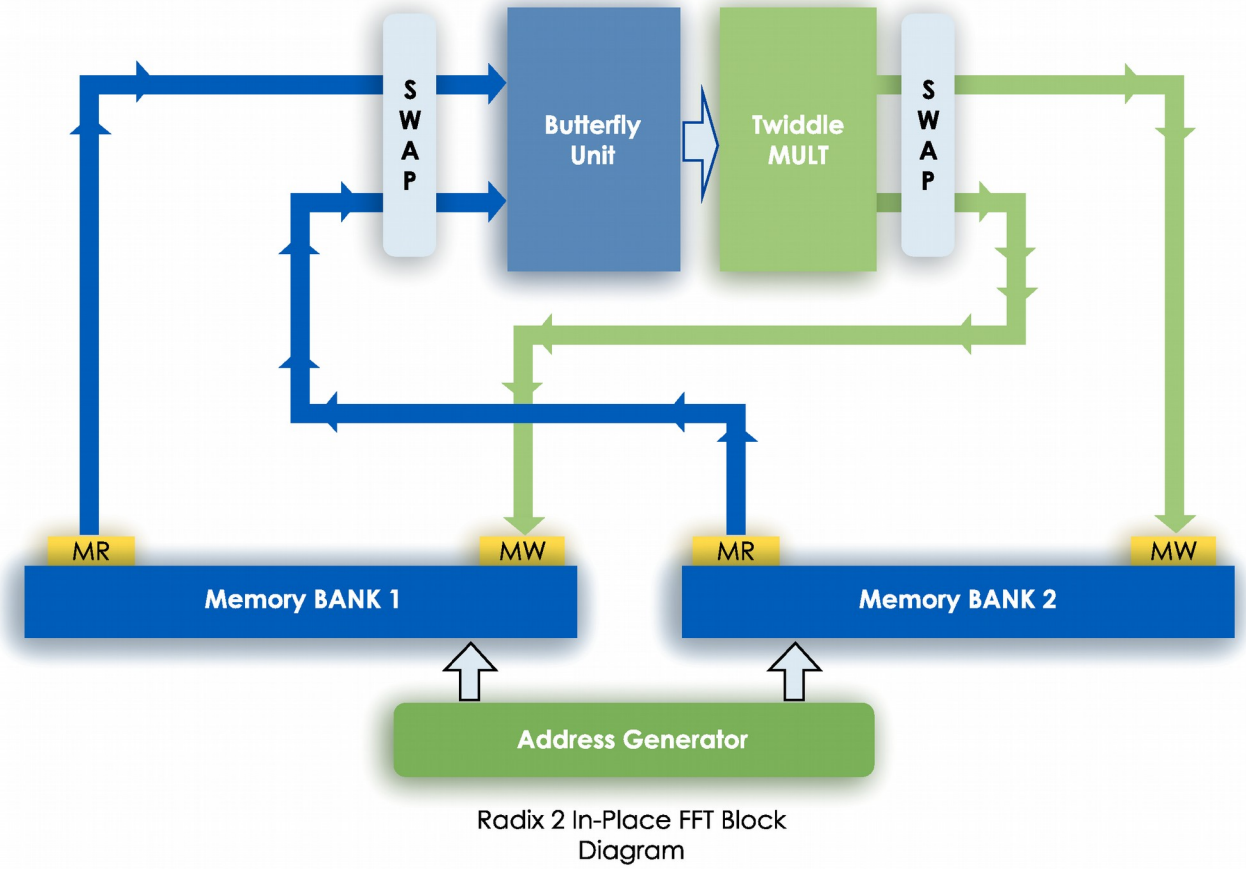


Illustration 16: Radix-2 DIF In-place FFT Block Diagram

(Note: This is only an explanatory diagram. The actual hardware will be different)

FFT In-place is a single Butterfly Architecture that occupies less area on silicon or fewer resources on FPGA. User has to provide serial input to the block. After fetching N Inputs input channel will stall for the duration FFT will be computed. Memory banks will be used to store intermediate results. Architecture implements Radix-2 DIF FFT so, Input is considered to be 'Natural' order and output will be in 'Bit – reversed' stream. This architecture shows that:

- The Memory Bank can be implemented using Dual Port Memories.
- Two memory banks will be used to read and write for every clock cycle.
- Butterfly unit will be used for computation in each clock cycle.
- Two swapping units will be used to swap the inputs and outputs of the butterfly, enabling inputs of the next stage to be simultaneously read from two banks.
- Address Generator inside FSM and used to access input and output butterfly calculation

3.2.1. Model Parameters

The C++ FFT Radix-2 In-place (DIT) IP is implemented as a class template named *ac_fft_dit_r2_inpl*. It depends upon a core class for the actual FFT computation, while a member function of the class, called *run(..) acts as a C++ wrapper and implements the in-place architecture. This member function is also synthesized as the top-level design block.*

Given below is snippet of *ac_fft_dif_r2_inpl* class.

```
template <INS_ID, N_FFT, TW_w, STAGE_w, STAGE_i, STAGE_r>
    class ac_fft_dif_r2_inpl
    {
      //code
    }
```

Class will have a member function named *run(..)* that would implement In-place structure.

***Note**: In C simulation, Input channel must contain N valid values where N equals to FFT points otherwise simulation asserts.*

Definition of various template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
TWIDDLE_PREC	Specifies the twiddle factor bitwidth. 2-bits will be automatically assigned to the integer part.
DIT_D_P	Specifies width of the stages, inputs and outputs, in bits
DIT_D_I	Specifies the integer width of the stages, inputs and outputs, in bits

3.2.2. Calling the *ac_fft_dif_r2_inpl* Top-level function

Consider DIF Radix-2 In-place FFT with the following static characteristics:

- 128 FFT points.
- 19-bit twiddle factors.
- 18-bit input/output/stage width, with 2 bits reserved for the integer part.

The steps required to instantiate the *ac_fft_dif_r2_inpl* object and calling the top-level member function are as follows:

1. Include <ac_dsp/ac_fft_dif_r2_inpl.h>
2. Define I/O data type in testbench header file.

```
typedef ac_fixed<18,2,true> data_real_type
```

```
typedef ac_complex<data_real_type> IO_complex_type
```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

```
ac_channel<IO_complex_type> instream;
ac_channel<IO_complex_type> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dif_r2_inpl* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dif_r2_inpl<128, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

3.2.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 17. The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

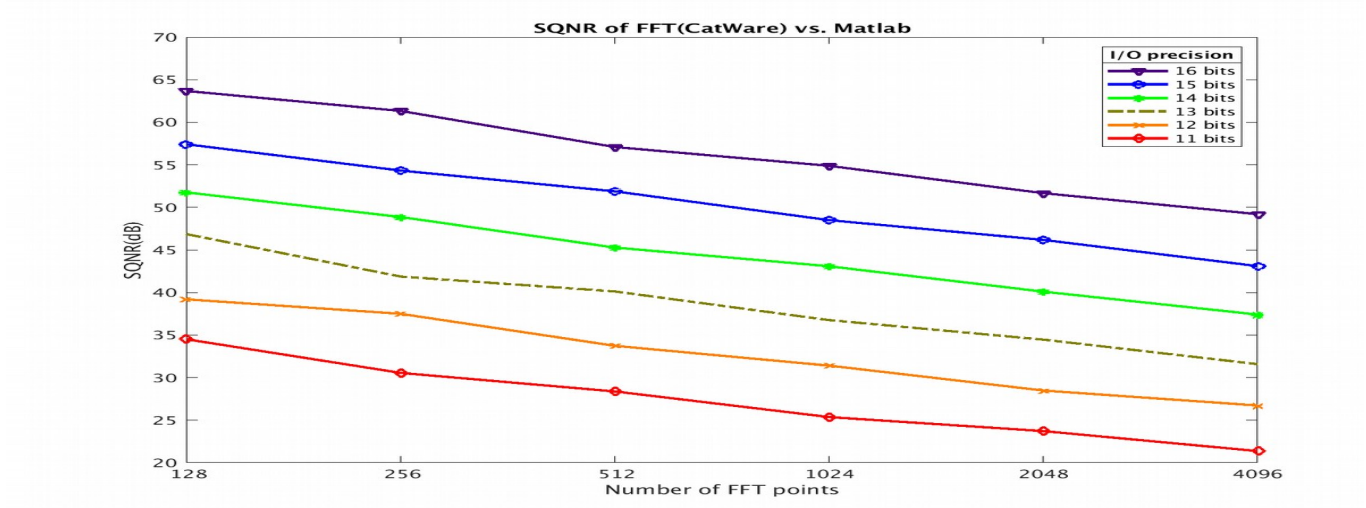


Illustration 17: SQNR Plot for Different Static Characteristics

3.3. Radix-2/2^2 Mix (DIF) Single Delay Feedback

This Fast Fourier Transform (FFT) is a Mix Radix 2/2^2 FFT with a decimation in frequency (DIF) Single Delay Feedback (SDF) architecture. The FFT has a parameterized and high performance FFT implementation in both C++ . The architecture is synthesizable by Catapult to produce the corresponding RTL implementation constrained by user and is design to process a continuous stream of data.

Illustration 18 shows the system block diagram for this function.

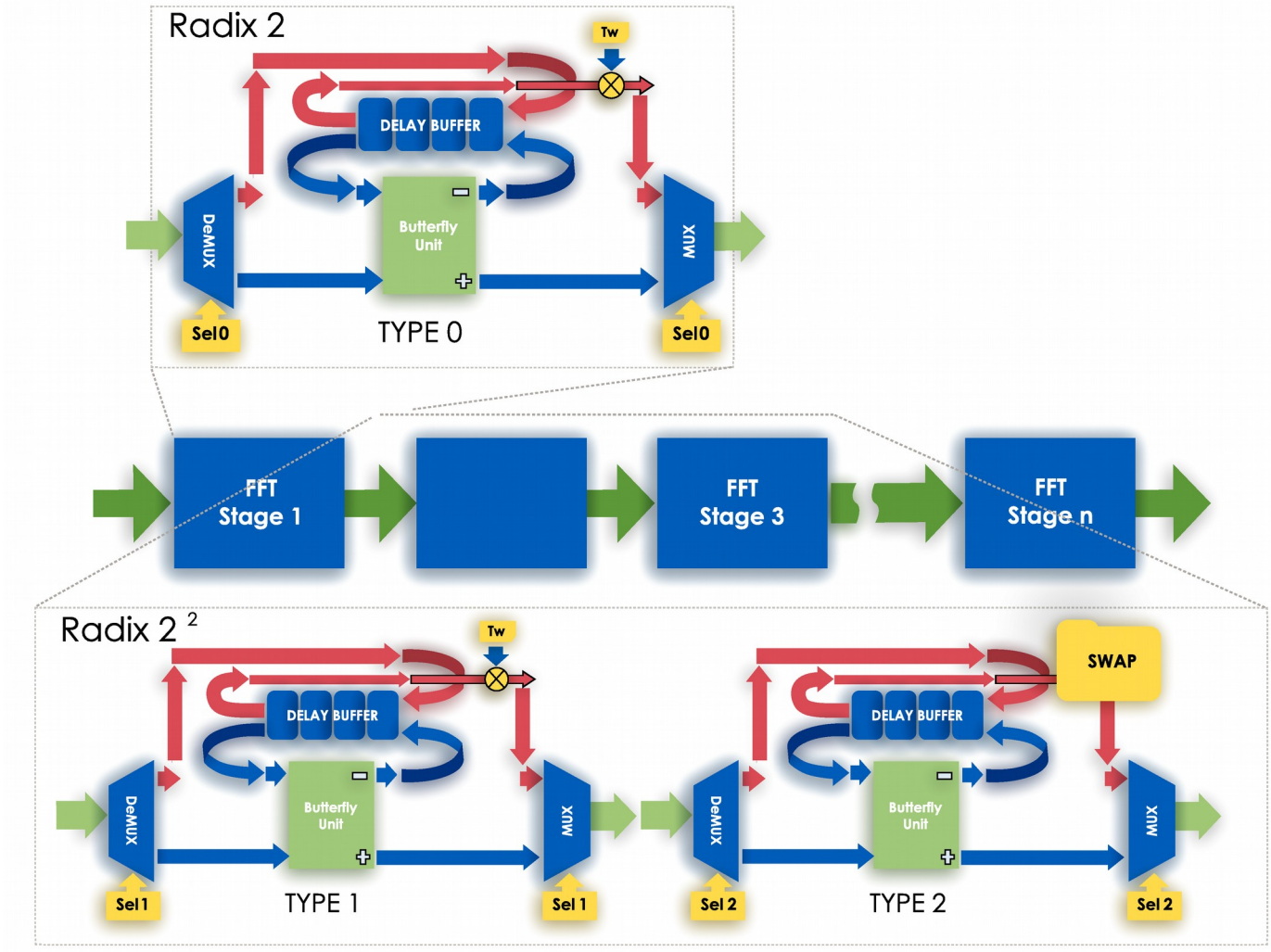


Illustration 18: Mix-Radix SDF System Block Diagram

(Note: This is only an explanatory diagram actual hardware will be different)

A block diagram representation of the FFT DIF mix Radix-2/22 SDF Architecture is shown in Illustration 18. FFT SDF Architecture is designed to provide high throughput. CatWare provides streaming input and output. For a N=(2n) Point FFT, implementation of n stages, will be done inside the CatWare. Architecture implements Radix-2/22 Single Delay Feedback DIF FFT. Output will be bit-reversed.

The above block diagram shows:

- ‘log2N’ Cascaded stages with each stages containing a Butterfly unit.
- Three type of Butterfly computation units are used Type0(Radix2), Type 1 (with twiddle multiplier) and Type 2 (with swapping unit).

- If numbers of FFT Points are not Power of 4 then, first stage will be carrying Type 0 butterfly unit. Second stage from input will be with Type 2 butterfly then stage 3 with Type 1 and likewise alternate butterfly Type will be used for each stage.
- If numbers of FFT Points are Power of 4 then, first stage from input will be with Type 2 butterfly then stage 2 with Type 1 and likewise alternate butterfly Type will be used for each stage.
- Each stage will implements a FIFO (circular buffer or shift register) depth of which depends on the which stage it is.
- Cascading of stage is done to maintain the highest throughput.
- MUX will be used to switch between the FIFO and butterfly Unit (switching is shown with Red and Blue colour in figure:1).
- Twiddles will be mapped to ROMs.

3.3.1. Model Parameters

Core functionality of the FFT function is implemented as a C++ class template. The object of the core class is wrapped in another class for C++ implementation. Template parameters are used to configure the FFT function for a specific HW implementation.

Given below is a snippet of the class.

```
template<unsigned N_FFT, int MEM_TH, int TWID_PREC, int DIF_D0_P, int DIF_D0_I>
class ac_fft_dif_r2m2p2_sdf {
  //code
}
```

The class has a member function named *run(..)* that calls the core design functions. This member function is synthesized as the top-level block in catapult synthesis using class-based hierarchy. The core class member functions are inlined. Every stage is hard-coded to have its output scaled by half to avoid overflow. To avoid this, the user can refer to comments in the code and change it accordingly to have the output passed as it is, without any scaling.

***Note**: In C simulation, Input channel must contain N valid values where N is the number of FFT points. Without these values, the simulation asserts.*

Definition of the template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
MEMORY_THRESHOLD	Memory threshold parameter. If the FIFO size is greater than MEM_TH, it is synthesized as a circular buffer. If not, it is synthesized as a shift register.
TWID_PREC	Specifies the twiddle factor bitwidth. 2-bits will be automatically assigned to the integer part.
DIF_D0_P	Specifies width of the stage 0 in bits. The width of every stage thereafter, as well

	as the input and output, will be the same.
DIF_D0_I	Specifies integer width of the stage 0 in bits. The integer width of every stage thereafter, as well as the input and output, will be the same.

3.3.2. Calling the ac_fft_dif_r2m2p2_sdf Top-level function

Let us consider an FFT implementation with the following static characteristics:

- 128 FFT points.
- Memory Threshold of 32.
- 19-bit twiddle factor precision.
- Stage, input and output bitwidth of 18 bits, with 2 bits being allocated for the integer part.

The following are the steps to initialize an object of the *ac_fft_dif_r2m2p2_sdf* class with the above mentioned static characteristics:

1. Include <ac_dsp/ac_fft_dif_r2m2p2_sdf.h>
2. Define input/output data type in the testbench header file.

```
typedef ac_fixed<18, 2, true> dif_fxp_in;
typedef ac_complex<dif_fxp_in> dif_inp, dif_out;
```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

```
ac_channel<dif_inp> instream;
ac_channel<dif_out> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dif_r2m2p2_sdf* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dif_r2m2p2_sdf <128, 32, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

3.3.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 19. The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 18-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

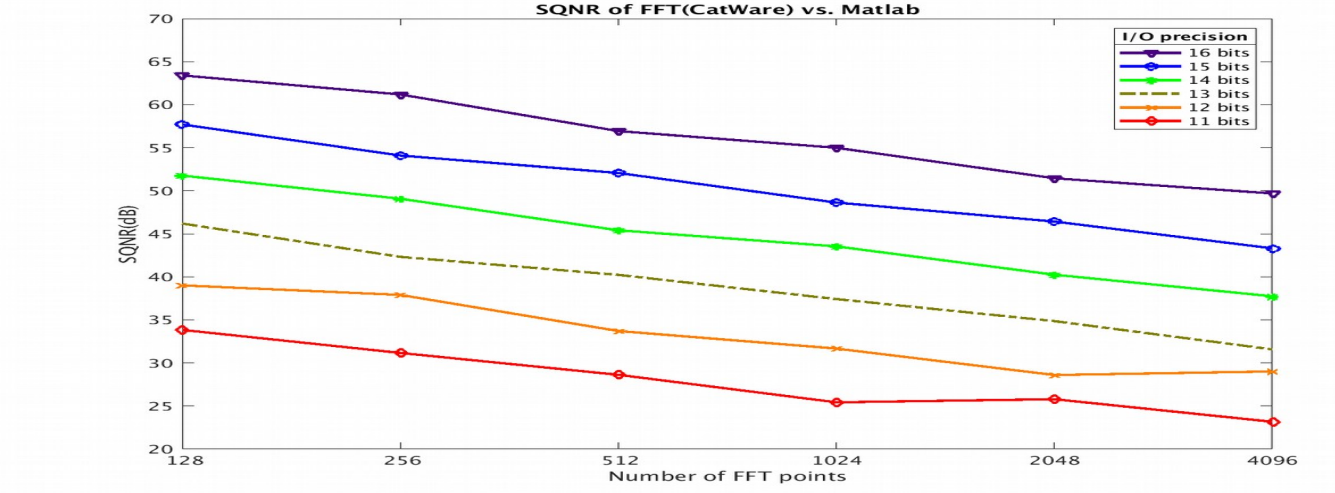


Illustration 19: SQNR Plots for Different Static Characteristics

3.4. Radix-2^2 (DIF) Single Delay Feedback

This Fast Fourier Transform (FFT) is a Radix2^2 FFT with a decimation in frequency (DIF) Single Delay Feedback (SDF) architecture. The FFT has a parameterized and high performance FFT implementation in C++. The architecture is synthesizable by Catapult to produce the corresponding RTL implementation constrained by user and is design to process a continuous stream of data.

Illustration 20 shows the system block diagram for this function.

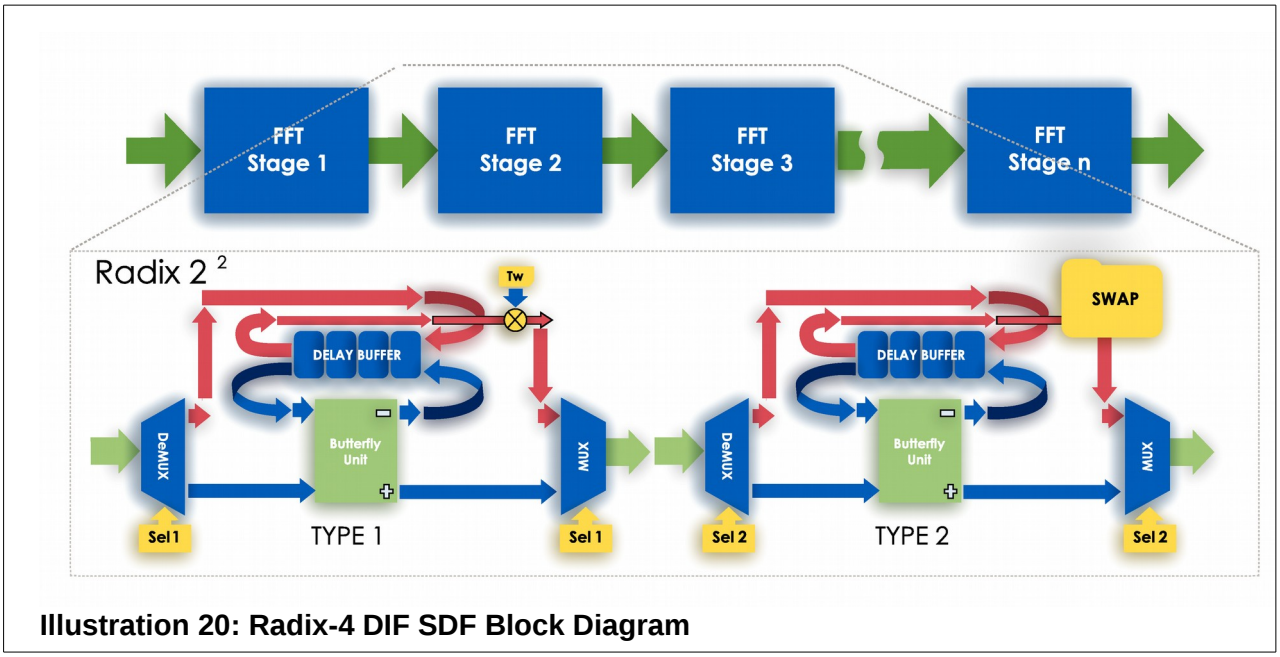


Illustration 20: Radix-4 DIF SDF Block Diagram

***Note**: This is only an explanatory diagram actual hardware will be different.*

A block diagram representation of the FFT DIF Radix 22 SDF Architecture is shown in the Figure 1. FFT SDF Architecture is designed to provide high throughput. FFT function provides streaming input and output. For a N=(4n) Point FFT, implementation of 2n stages, will be done inside the FFT function. Architecture implements Radix-2^2 Single Delay Feedback DIF FFT. Output will be bit - reversed.

The following list describes some of the architectural decisions that were made for this implementation:

- 'log2N' Cascaded stages with each stages containing a Butterfly unit.
- Radix - 2 type of Butterfly computation units are used
- Each stage will implements a FIFO (circular buffer or shift register) depth of which depends on the which stage it is.
- Cascading of stage is done to maintain the highest throughput.
- MUX will be used to switch between the FIFO and butterfly Unit (switching is shown with Red and Blue colour in figure:1).
- Twiddles will be mapped to ROMs.

3.4.1. Model Parameters

Core functionality of the FFT function is implemented as a C++ class template. The object of the core class is wrapped in another class for C++ implementation. Template parameters are used to configure the FFT function for a specific HW implementation.

Given below is a snippet of the class.

```
template<unsigned N_FFT, int MEM_TH, int TWID_PREC, int DIF_D0_P, int DIF_D0_I>
class ac_fft_dif_r2p2_sdf {
  //code
}
```

The class has a member function named *run(..)* that calls the core design functions. This member function is synthesized as the top-level block in catapult synthesis using class-based hierarchy. The core class member functions are inlined. Every stage is hard-coded to have its output scaled by half to avoid overflow. To avoid this, the user can refer to comments in the code and change it accordingly to have the output passed as it is, without any scaling.

***Note**: In C simulation, Input channel must contain N valid values where N is the number of FFT points. Without these values, the simulation asserts.*

Definition of the template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
MEMORY_THRESHOLD	Memory threshold parameter. If the FIFO size is greater than MEM_TH, it is synthesized as a circular buffer. If not, it is synthesized as a shift register.

TWID_PREC	Specifies the twiddle factor bitwidth. 2-bits will be automatically assigned to the integer part.
DIF_D0_P	Specifies width of the stage 0 in bits. The width of every stage thereafter, as well as the input and output, will be the same.
DIF_D0_I	Specifies integer width of the stage 0 in bits. The integer width of every stage thereafter, as well as the input and output, will be the same.

3.4.2. Calling the ac_fft_dif_r2p2_sdf Top-Level Function

Let us consider an FFT implementation with the following static characteristics:

- 128 FFT points.
- Memory Threshold of 32.
- 19-bit twiddle factor precision.
- Stage, input and output bitwidth of 18 bits, with 2 bits being allocated for the integer part.

The following are the steps to initialize an object of the *ac_fft_dif_r2p2_sdf* class with the above mentioned static characteristics:

1. Include <ac_dsp/ac_fft_dif_r2p2_sdf.h>
2. Define input/output data type in the testbench header file.

```
typedef ac_fixed<18, 2, true> dif_fxp;
typedef ac_complex<dif_fxp_in> dif_inp, dif_out;
```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

```
ac_channel<dif_inp> instream;
ac_channel<dif_out> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dif_r2p2_sdf* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dif_r2p2_sdf <128, 32, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

3.4.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 21. The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

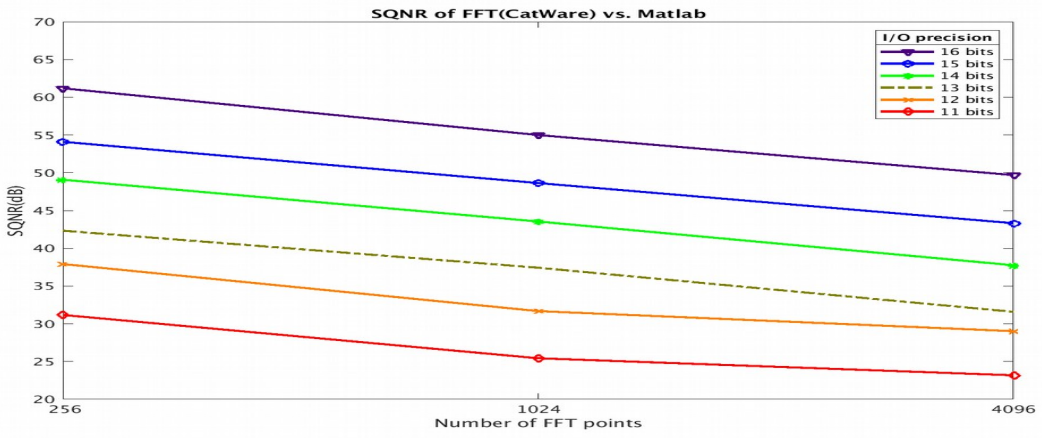


Illustration 21: SQNR Plots for Different Static Characteristics

3.5. Radix-2^X (DIF) In-place with Block Floating Point

This Fast Fourier Transform (FFT) is a Block Floating Point generic Radix 2X with Automated Radix Mixing feature implementing Decimation in Frequency (DIF) In-place architecture. The FFT has a parameterized and high performance implementation in both C++ and System C. The FFT is synthesizable by Catapult to produce the corresponding RTL implementation constrained by user.

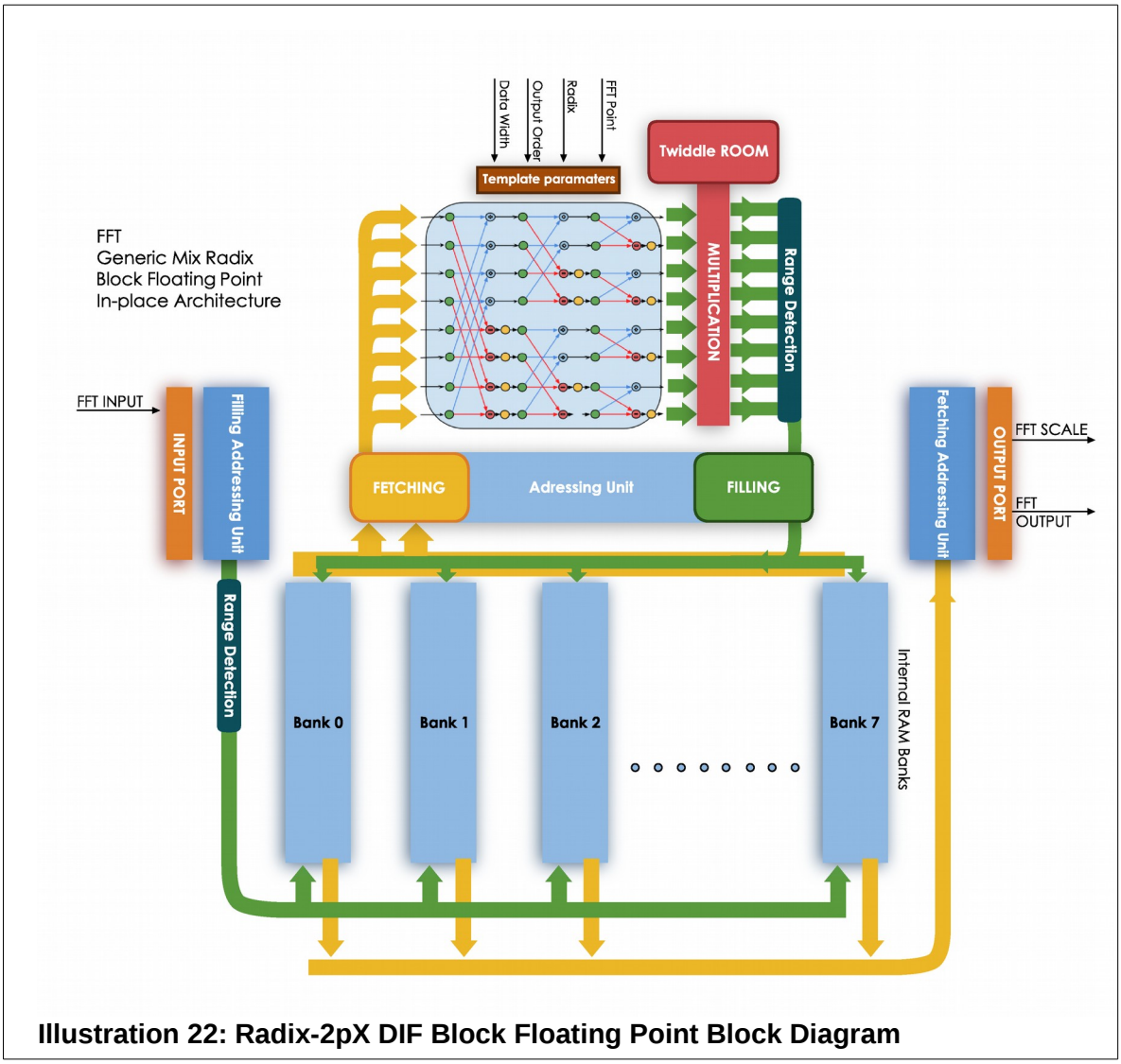


Illustration 22: Radix-2pX DIF Block Floating Point Block Diagram

(Note: This is only an explanatory diagram actual hardware will be different)

3.5.1. Model Parameters

The IP for this block is implemented as a C++ class template named *ac_fft_dif_r2pX_bfp_inpl*. Given below is a snippet of the *ac_fft_dif_r2pX_bfp_inpl* class.

```
template<int N_FFT,int RADIX,int ORDER,int TWIDDLE_PREC,int DIF_D_P,int
DIF_D_I, class out_struct>
class  ac_fft_dif_r2pX_bfp_inpl
{
  //code
}
```

The class has a member function named run*(..)* that acts as the top-level block. Note: The input channel must contain N valid values where N is the number of FFT points while simulating in C. This is done in order to prevent an assertion due to an empty channel read.

Definition of template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
RADIX	Specifies Radix engine size
ORDER	Specifies Output order (0 for Bit reversed/DIF order, 1 for Natural)
TWIDDLE_PREC	Specifies twiddle factor bitwidth. 2 bits of this will automatically be assigned to the integer part.
DIF_D_P	Specifies width of the stages, in bits
DIF_D_I	Specifies the width of each stage's integer part, in bits

3.5.2. Calling the ac_fft_dif_r2px_bfp_inpl Top-level function

Let us consider an FFT implementation with the following static characteristics:

- 128 FFT points.
- Memory Threshold of 32.
- Bit reversed output order
- 19-bit twiddle factor precision.
- Stage, input and output bitwidth of 18 bits, with 2 bits being allocated for the integer part.

The following are the steps to initialize an object of the *ac_fft_dif_r2pX_bfp_inpl* class with the above mentioned static characteristics:

1. Include <ac_dsp/ac_fft_dif_r2pX_bfp_inpl.h>
2. Define input/output data type in the testbench header file.

```
typedef ac_fixed<18, 2, true> dif_fxp;
typedef ac_complex<dif_fxp_in> dif_inp, dif_out;
typedef ac_int < ac::log2_ceil < ac::log2_ceil < F_N >::val >::val +1, 1 >
scale_typ;
struct out_struct {
  dif_out manti;
  scale_typ expo;
};
```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

```
ac_channel<dif_inp> instream;
ac_channel<dif_out> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dif_r2pX_bfp_inpl* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dif_r2pX_bfp_inpl <128, 32, 0, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

3.5.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 23. The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

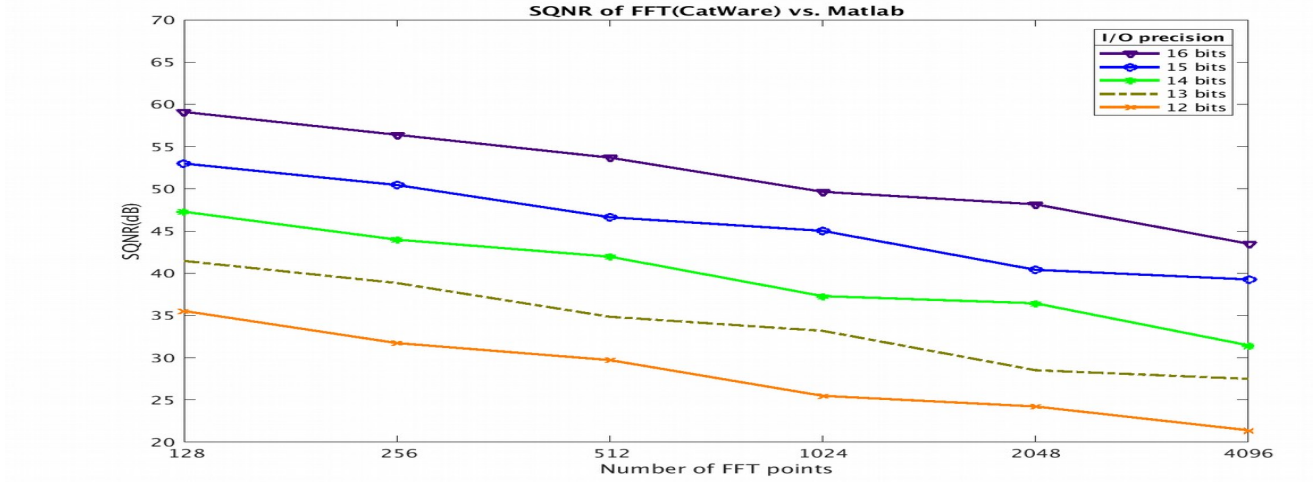


Illustration 23: SQNR Plots for Different Static Characteristics

** SQNR is dependent on type of input and number of other parameters. Results shown here are only valid for specific Input.*

3.6. Radix-2^X (DIF) Dynamic In-place

This is a generic Mix-Radix 2^X FFT which is dynamic in number of points and instances. It implements an in-place, Decimation in Frequency (DIF) architecture. The FFT has a parameterized and high performance implementation in C++. The FFT implementation can be synthesized by Catapult to produce the corresponding RTL implementation as constrained by the user. The system block diagram for this function is shown in Illustration 24.

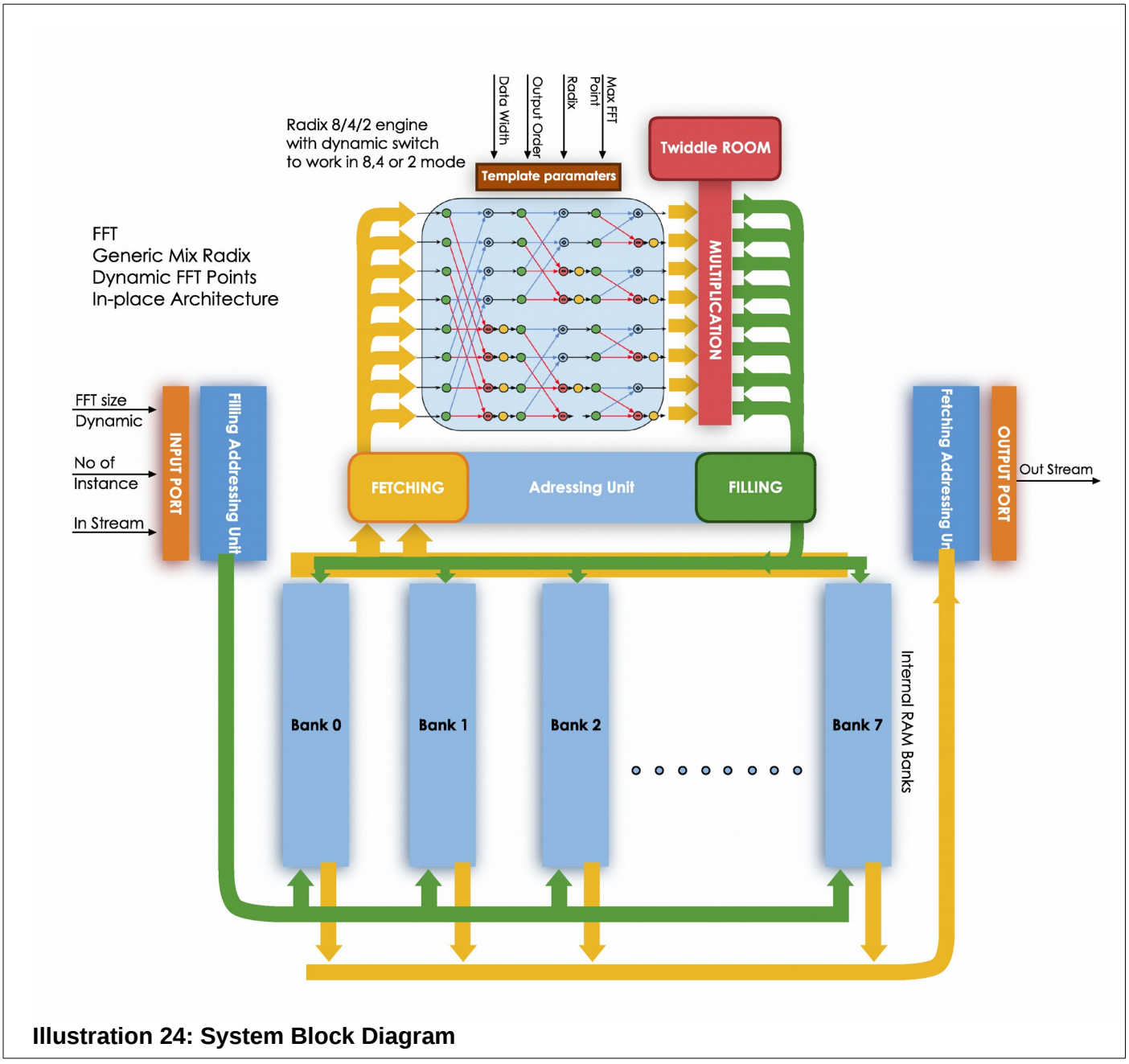


Illustration 24: System Block Diagram

3.6.1. Model Parameters

The IP for this block is implemented as a C++ class template named *ac_fft_dif_r2pX_dyn_inpl*. Given below is a snippet of the *ac_fft_dif_r2pX_dyn_inpl* class.

```
template<unsigned N_FFT,int RADIX,int ORDER,int TWIDDLE_PREC,int DIF_D_P,int
DIF_D_I>
class ac_fft_dif_r2pX_dyn_inpl
{
```

```
  //code
}
```

The class has a member function named *run(..)* that acts as the top-level block. Note: The input channel must contain N valid values where N is the number of FFT points while simulating in C. This is done in order to prevent an assertion due to an empty channel read.

Definition of template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
RADIX	Specifies Radix engine size
ORDER	Specifies Output order (0 for Bit reversed/DIF order, 1 for Natural)
TWIDDLE_PREC	Specifies twiddle factor bitwidth. 2 bits of this will automatically be assigned to the integer part.
DIF_D_P	Specifies width of the stages, in bits
DIF_D_I	Specifies the width of each stage's integer part, in bits

3.6.2. Calling the ac_fft_dif_r2pX_dyn_inpl Top-level Function

Let us consider an FFT with the following static and dynamic characteristics:

Static characteristics

- 128-point FFT.
- Radix-4 engine.
- Input, Output and Stage precision of 18 bits, with a 2-bit integer part.
- Bit Reversed/DIF Output order.
- Twiddle precision of 19 bits.

Dynamic Characteristics

- First burst has 10 instances of an 8 point FFT input.
- Second burst has 3 instances of a 32 point FFT input.

The following are the steps required to instantiate an FFT with these characteristics:

1. Include <ac_dsp/*ac_fft_dif_r2pX_dyn_inpl.h*>
2. Define input data type in the testbench header file.

```
typedef ac_fixed<18,2,true> dif_fxp_in;
```

```
typedef ac_complex<dif_fxp_in> dif_inp, dif_out;
```

3. Declare *ac_channel* objects in the CCS_MAIN() function of the corresponding testbench .cpp file. Keep in mind that the widths of the *ac_int* types encapsulated in the *ac_channel*s for dynamic reduction and instance count depend upon the number of FFT points and radix, as shown below.

```
const int dyn_W = ac::log2_ceil<ac::log2_ceil<128>::val -
                                ac::log2_ceil<  4>::val + 1>::val;
const int ins_W = ac::log2_ceil<128>::val - ac::log2_ceil<4>::val;
ac_channel<dif_inp> instream;
ac_channel<dif_out> outstream;
ac_channel<ac_int<dyn_W, false> > dyn_port;
ac_channel<ac_int<ins_W, false> > ins_port;
```

4. Write the FFT inputs, dynamic reduction and instance count parameters to the relevant *ac_channel*s using loops. Following that, instantiate an object of the *ac_fft_dif_r2pX_dyn_inpl* class and call the top level member function with the *ac_channel*s defined above.

```
ac_fft_dif_r2pX_dyn_inpl <128, 4, 0, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream, dyn_port, ins_port);
```

fft_design1 is an object of the *ac_fft_dif_r2pX_dyn_inpl* class with the static characteristics mentioned above.

The values on the relevant input ports will configure the block for the dynamic no. of FFT points and no. of instances, as shown in the table below, for the dynamic characteristics listed above.

Dynamic Reduction Port	Dynamic FFT Points	Instance Count Port	No. of Complex Inputs
4	8 (128 >> 4)	10	80 (8 x 10)
2	32 (128 >> 2)	3	96 (32 x 3)

3.6.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 25. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part. A Radix-4 implementation is used.

The input passed to measure the SQNR is a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR.

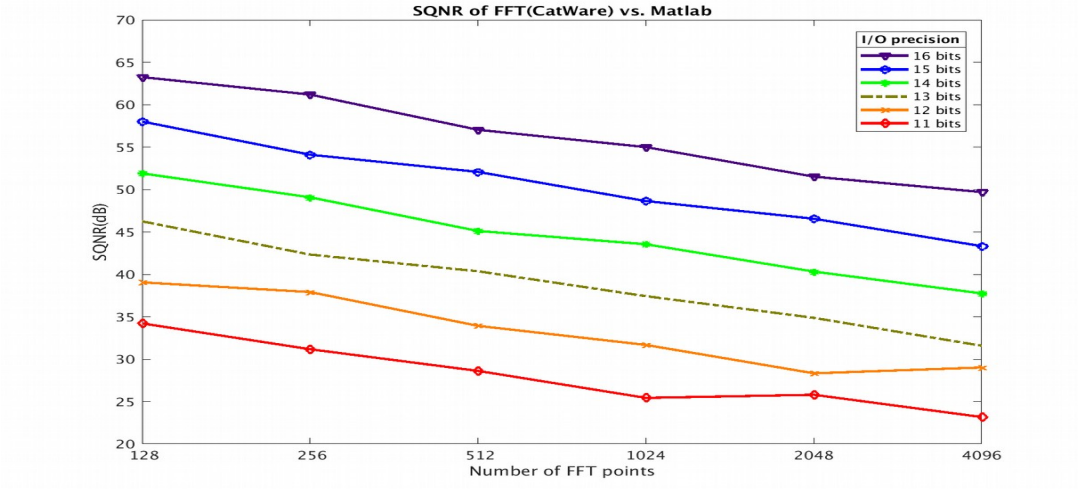


Illustration 25: SQNR Plots for Different Static Characteristics

3.7. Radix-2^X (DIF) with Automated Radix Mixing

This Fast Fourier Transform (FFT) is a generic Radix 2^X with Automated Radix Mixing, implementing a Decimation in Frequency (DIF), In-place architecture. The FFT has a parameterized and high performance implementation in C++. The implementation is synthesizable by Catapult to produce the corresponding RTL implementation as constrained by the user.

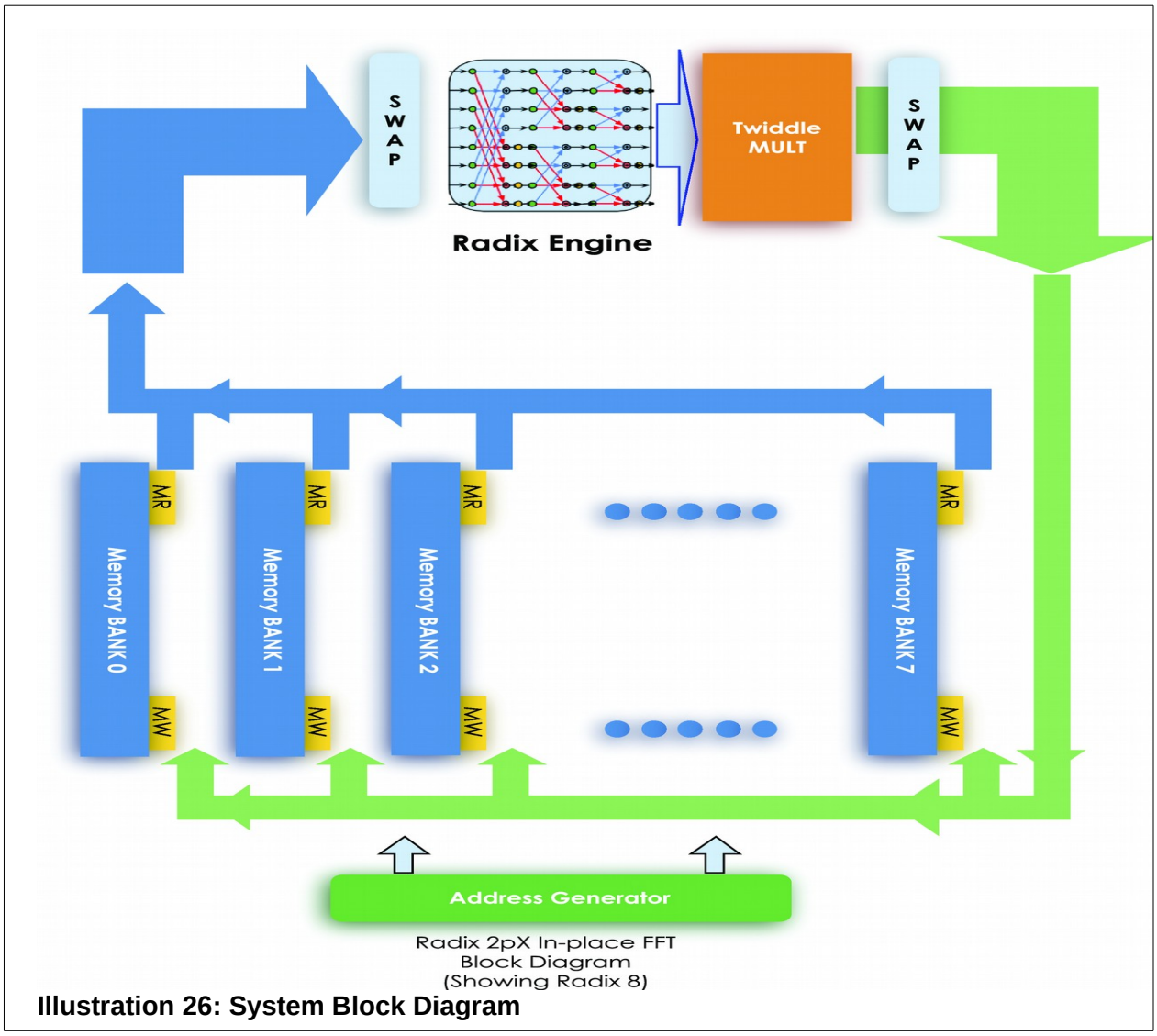


Illustration 26: System Block Diagram

Note: This is only an explanatory diagram. The actual hardware will be different.

Illustration 26 shows an implementation having 8 Dual port Memory Banks and Radix engine having a Radix-2 flow graph structure to occupy less area on silicon or fewer resources on FPGA. User has to provide a serial input to the block. After fetching N Inputs input channels will stall until the FFT is completely computed. Memory banks will be used to store intermediate results. Since the architecture implements a DIF FFT so, the input is considered to be in natural order. The output will be in a bit–reversed or natural order depending on the template parameter 'ORDER'. This architecture shows that:

- The Memory Bank can be implemented using Dual Port Memories.
- 8 memory banks will be used to read and write eight values every clock cycle.
- Radix engine will take eight inputs and provide eight outputs every clock cycle.
- Two swapping units will be used to swap the input and output vectors of the butterfly, enabling inputs of the next stage to be simultaneously read from eight banks without any address conflict.

- The Address Generator, controlled by the FSM, is used to access butterfly inputs and outputs.

3.7.1. Model Parameters

FFT Radix-2pX In-place (DIF) IP is implemented as C++ class template named *ac_fft_dif_r2pX_inpl*.

Given below is snippet of *ac_fft_dif_r2pX_inpl* class.

```
template<unsigned N_FFT,int RADIX,int ORDER,int TWIDDLE_PREC,int DIF_D_P,int
DIF_D_I>
class ac_fft_dif_r2pX_inpl
{
//code
}
```

This class will have a member function named *run(..)* that would implement In-place structure. This function is synthesized as a top-level hierarchical block.

***Note**: In C simulation, Input channel must contain N valid values where N equals to FFT points otherwise simulation asserts.*

The definition of the template parameters used is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
RADIX	Specifies Radix engine size
ORDER	Specifies Output order =0 for Bit reverse =1 for Natural
TWIDDLE_PREC	Specifies the twiddle factor in bits. 2-bits will be automatically assigned to the integer part(STAGE_i)
DIF_D_P	Specifies width of the stages, in bits
DIF_D_I	Specifies the stages integer part, in bits

3.7.2. Calling the ac_fft_dif_r2pX_inpl Top-level Function

Let us consider an FFT implementation with the following static characteristics:

- 128 FFT points.
- Radix-4 implementation.
- Output in DIF/Bit-reversed order.
- 19 bit wide twiddle factors.
- Input, output and stage width set to 18 bits, with 2 bit being allocated for the integer part.

1. Include <ac_dsp/ac_fft_dif_r2pX_inpl.h>

2. Define input/output data type in the testbench header file.

```
typedef ac_fixed<18, 2, true> dif_fxp_in;
typedef ac_complex<dif_fxp_in> dif_inp, dif_out;
```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

```
ac_channel<dif_inp> instream;
ac_channel<dif_out> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dif_r2pX_inpl* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dif_r2pX_inpl <128, 4, 0, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

3.7.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 27. The x axis represents number of FFT points and y axis shows SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part. A Radix-4 implementation is used.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

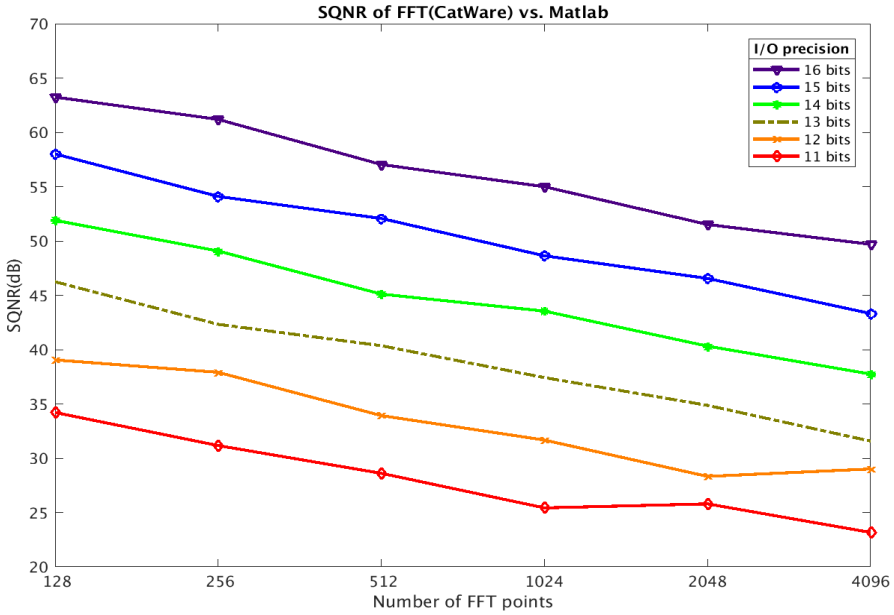


Illustration 27: SQNR Plots for Different Static Characteristics

3.8. Radix-2 (DIF) Single Delay Feedback

This FFT implementation uses a Radix-2, Decimation In Frequency (DIF) Single Delay Feedback (SDF) architecture, with a parameterized and high performance implementation in C++. The architecture is synthesizable by Catapult to produce the corresponding RTL implementation constrained by the user. It is designed to process a continuous stream of data and can be pipelined with an II of 1.

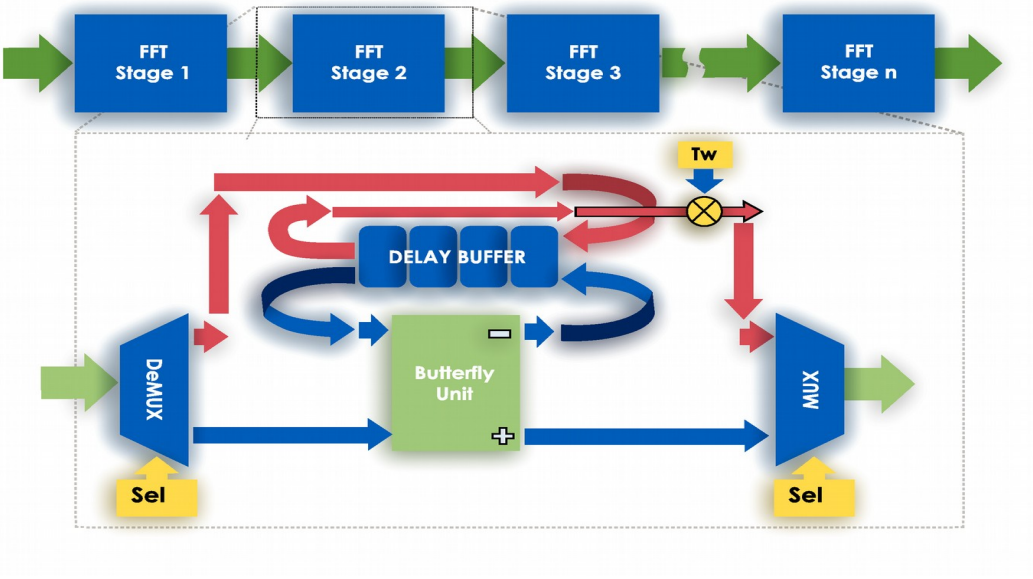


Illustration 28: System Block Diagram

A block diagram representation of the FFT DIF Radix 2 SDF Architecture is shown above. It is designed to provide high throughput, with streaming interfaces at both the input and output implemented as *ac_channel*s. For an $N = 2^n$ Point FFT, implementation of *n* stages will be done inside the FFT function. Architecture implements Radix-2 Single Delay Feedback DIF FFT. The output will be in bit-reversed order.

The following are the features of the architecture, as seen in Illustration 28:

- *n* cascaded stages with each stage containing a Butterfly unit.
- Radix-2 butterfly computation units are used
- Each stage implements a FIFO (circular buffer or shift register), the depth of which depending on the which stage it is.
- Cascading of stages is done to maintain the highest throughput.
- MUX will be used to switch between the FIFO and butterfly Unit (switching datapaths are shown in red and blue in the block diagram)

3.8.1. Model Parameters

Core functionality of the FFT function is implemented as a C++ class template. The object of the core class is wrapped in another class for C++ implementation. Template parameters are used to configure the FFT function for a specific HW implementation.

Given below is a snippet of the class.

```
template<unsigned N_FFT, int MEM_TH, int TWID_PREC, int DIF_D0_P, int DIF_D0_I>
class ac_fft_dif_r2_sdf {
```

```
  //code
}
```

The class has a member function named *run(..)* that calls the core design functions. This member function is synthesized as the top-level block in catapult synthesis.

***Note**: In C simulation, Input channel must contain N valid values where N is the number of FFT points. Without these values, the simulation asserts.*

Definition of the template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
MEM_TH	Memory threshold parameter. If the FIFO size is greater than MEM_TH, it is synthesized as a circular buffer. If not, it is synthesized as a shift register.
TWID_PREC	Specifies the twiddle factor bitwidth. 2-bits will be automatically assigned to the integer part.
DIF_D0_P	Specifies width of the stage 0 in bits. The width of every stage thereafter, as well as the input and output, will be the same.
DIF_D0_I	Specifies integer width of the stage 0 in bits. The integer width of every stage thereafter, as well as the input and output, will be the same.

3.8.2. Calling the ac_fft_dif_r2_sdf Top-level Function

Let us consider an FFT implementation with the following static characteristics:

- 128 FFT points.
- Memory Threshold of 32.
- 19-bit twiddle factor precision.
- Stage, input and output bitwidth of 18 bits, with 2 bits being allocated for the integer part.

The following are the steps to initialize an object of the *ac_fft_dif_r2_sdf* class with the above mentioned static characteristics:

1. Include <ac_dsp/ac_fft_dif_r2_sdf.h>
2. Define input/output data type in the testbench header file.

   ```
   typedef ac_fixed<18, 2, true> dif_fxp_in;
   typedef ac_complex<dif_fxp_in> dif_inp, dif_out;
   ```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

   ```
   ac_channel<dif_inp> instream;
   ```

```
ac_channel<dif_out> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dif_r2pX_inpl* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dif_r2_sdf <128, 32, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

5. Keep in mind that in order to fully flush out the pipeline, the user must also write in an addition 128 placeholder values (same as the number of FFT points) and call the run function again.

3.8.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 29. The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

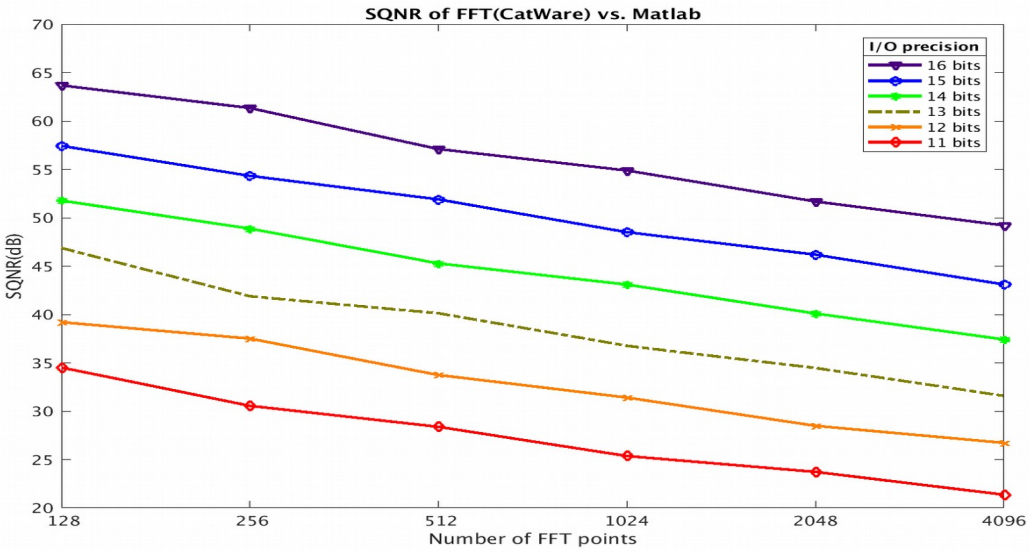


Illustration 29: SQNR Plots for Different Static Characteristics

3.9. Radix-2 (DIT) In-place

This Fast Fourier Transform (FFT) implementation utilizes a Radix-2, Decimation in Time (DIT) In-place architecture. The FFT has a parameterized and high performance implementation in C++. The FFT is synthesizable by Catapult to produce the corresponding RTL implementation constrained by the user.

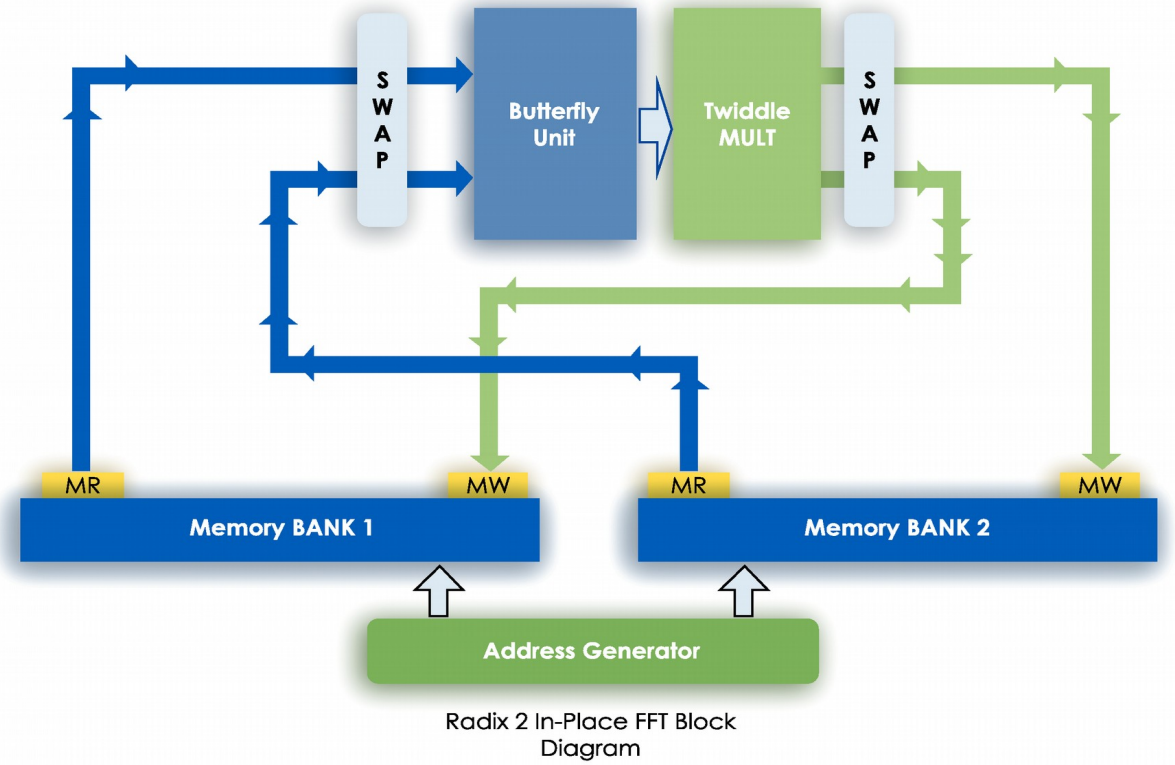


Illustration 30: System Block Diagram

A block diagram representation of the Radix-2 “In Place DIT” Architecture is shown in Illustration 30. A single Butterfly Architecture is used to occupy less area on silicon or fewer resources on FPGA. The user has to provide serial input to the block through the input *ac_channel*. After fetching N Inputs, the input channel will stall until the FFT is fully computed. Memory banks will be used to store intermediate results. Since this is a DIT FFT architecture, the input is assumed to arrive in a bit-reversed order. This architecture shows that:

- The Memory Bank can be implemented using Dual Port Memories.
- Two memory banks will be used to read and write for every clock cycle.
- Butterfly units will be used for computation in each clock cycle.
- Two swapping units will be used to swap the inputs and outputs of the butterfly, enabling inputs of the next stage to be simultaneously read from two banks.
- The address generator, controlled by the FSM, is used to read inputs and write butterfly outputs.

3.9.1. Model Parameters

The C++ FFT Radix-2 In-place (DIT) IP is implemented as a class template named *ac_fft_dit_r2_inpl*. It depends upon a core class for the actual FFT computation, while a member function of the class, called

run(..) acts as a C++ wrapper and implements the in-place architecture. This member function is also synthesized as the top-level design block.

```
template<unsigned N_FFT, int TWIDDLE_PREC, int DIT_D_P, int DIT_D_I>
class ac_fft_dit_r2_inpl
{
   //code
}
```

Class will have a member function named *run(..)* that would implement In-place structure.

Note: *In C simulation, Input channel must contain N valid values where N equals to FFT points otherwise simulation asserts.*

Definition of various template parameters is given below:

Parameters	Description
N_FFT	Specifies the number of FFT points
TWIDDLE_PREC	Specifies the twiddle factor bitwidth. 2-bits will be automatically assigned to the integer part.
DIT_D_P	Specifies width of the stages, inputs and outputs, in bits
DIT_D_I	Specifies the integer width of the stages, inputs and outputs, in bits

3.9.2. Calling the ac_fft_dit_r2_inpl Top-level Function

Consider DIT Radix-2 In-place FFT with the following static characteristics:

- 128 FFT points.
- 19-bit twiddle factors.
- 18-bit input/output/stage width, with 2 bits reserved for the integer part.

The steps required to instantiate the *ac_fft_dit_r2_inpl* object and calling the top-level member function are as follows:

1. Include <ac_dsp/ac_fft_dit_r2_inpl.h>
2. Define I/O data type in testbench header file.

   ```
   typedef ac_fixed<18,2,true> data_real_type
   typedef ac_complex<data_real_type> IO_complex_type
   ```

3. Declare *ac_channel*s with the I/O type defined above in the CCS_MAIN() function.

   ```
   ac_channel<IO_complex_type> instream;
   ac_channel<IO_complex_type> outstream;
   ```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dit_r2_inpl* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dit_r2_inpl<128, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

3.9.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 31.The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

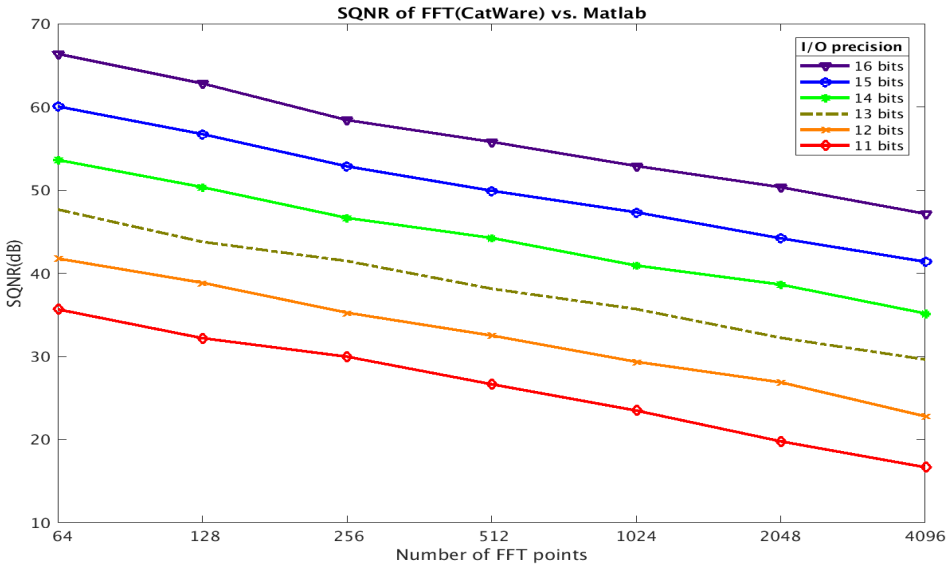


Illustration 31: SQNR Plots for Different Static Characteristics

3.10. Radix-2 (DIT) Single Delay Feedback

This Fast Fourier Transform (FFT) is a Radix 2 FFT with a Decimation in Time (DIT) Single-Delay Feedback(SDF) architecture. It has a parameterized, high-performance implementation in C++ that can be synthesized in Catapult to produce the corresponding RTL implementation as constrained by the user.

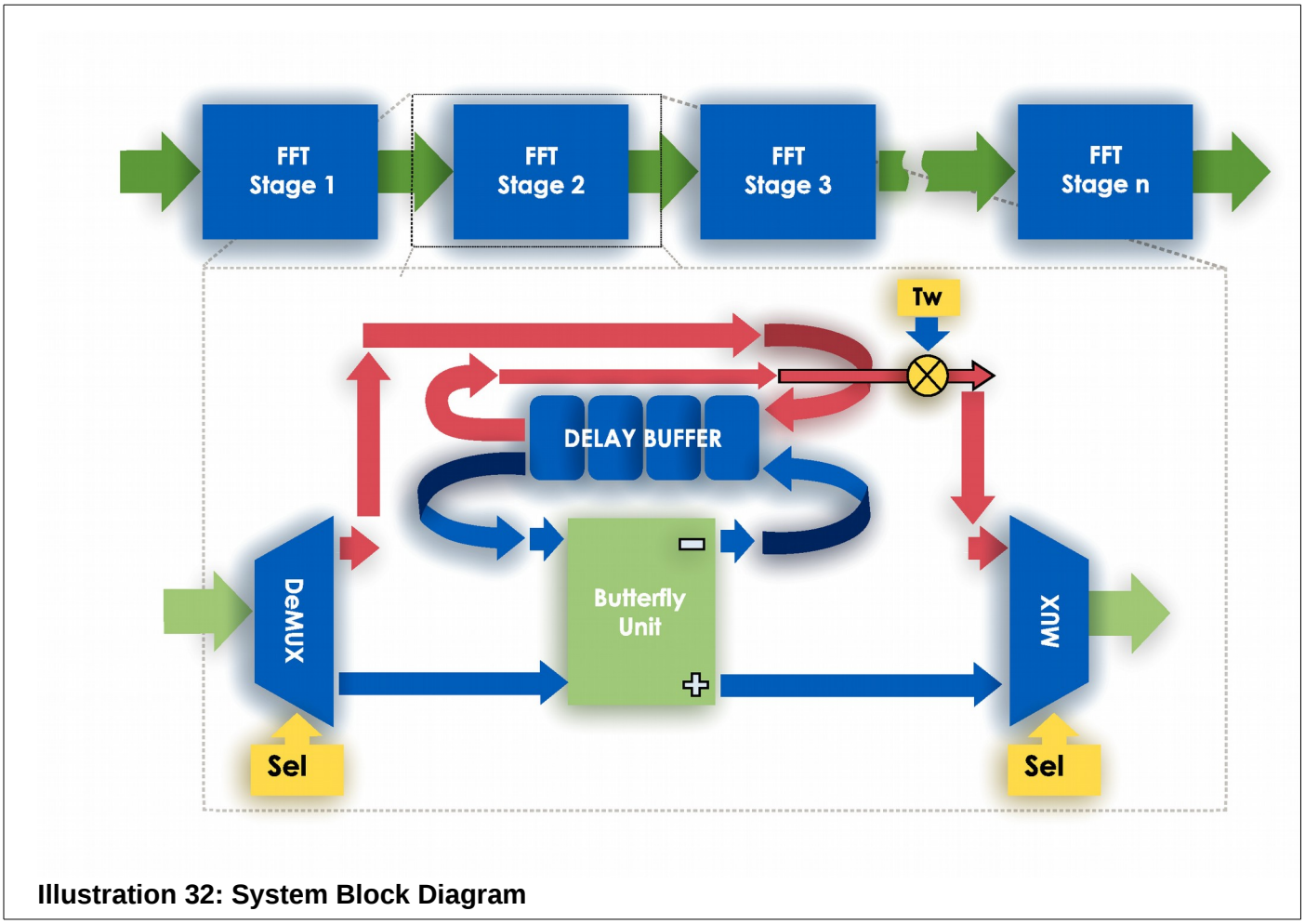


Illustration 32: System Block Diagram

A block diagram representation of the FFT DIT Radix- 2 SDF Architecture is shown in Illustration 32. FFT SDF Architecture is designed to provide a high throughput implementation that can be pipelined with and II of 1. The input and output have streaming, *ac_channel* interfaces. For an $N = 2^n$ Point FFT, n stages are implemented via the block. Architecture implements Radix-2 Single Delay Feedback DIT FFT.

The following features can be seen in the block diagram above:

- *n* Cascaded stages with each stages containing a Radix-2 Butterfly unit.
- Each stage will implements a FIFO (circular buffer or shift register), the depth of which depends on which stage it is.
- Cascading of stages is used to maintain the highest throughput.
- MUX will be used to switch between the FIFO and butterfly Unit (the switching datapaths are shown in red and blue in the above block diagram).

3.10.1. Model Parameters

The implementation is largely configured using template parameters, which can be seen in the below snippet of the *ac_fft_dit_r2_sdf* class.

```
template <unsigned N_FFT, int MEM_TH, int TWID_PREC, int DIT_D0_P, int
DIT_D0_I>
class ac_fft_dit_r2_sdf
{
//code
}
```

The class has a top-level member function named *run(..)* that implements an SDF structure.

***Note**: In C simulation, Input channel must contain N valid values where N equals to the number of FFT points, in order to avoid a simulation assertion.*

Definition of template parameters is given below:

Parameter	Description
N_FFT	Specifies the number of FFT points
MEM_TH	Specifies the memory threshold. Based on the FIFO size, memory threshold and stage number in the design, the design decides on whether to use a shift register or circular buffer for that particular stage's FIFO.
TWID_PREC	Specifies the twiddle factor bitwidth. 2-bits will be automatically assigned to the integer part.
DIT_D0_P	Specifies width of the first stage in bits. The inputs, outputs, and all other stages will also have the same width.
DIT_D0_I	Specifies integer width of the first stage in bits. The inputs, outputs, and all other stages will also have the same integer width.

3.10.2. Calling the ac_fft_dit_r2_sdf Top-Level Function

Consider a Radix-2 DIT SDF implementation using the following static characteristics:

- 128 FFT points.
- Memory threshold of 32.
- 19-bit twiddle factors.
- Inputs, outputs and stages that are 18 bits wide, with 2 bits being assigned to the integer part.

The following are the steps required to instantiate an object of the *ac_fft_dit_r2_sdf* class with the above static characteristics and call its top-level member function in the C++ testbench.

1. Include <ac_dsp/ac_fft_dit_r2_sdf.h>
2. Define IO datatype in testbench header file

```
typedef ac_fixed<18, 2, true> data_real_type
typedef ac_complex<data_real_type> IO_complex_type
```

3. Declare input and output *ac_channel*s in the CCS_MAIN() function.

```
ac_channel<IO_complex_type> instream;
ac_channel<IO_complex_type> outstream;
```

4. Write all input values to the instream *ac_channel*. Instantiate an object of the *ac_fft_dit_r2_sdf* class with the above static characteristics and call the top level member function with the instream and outstream channels.

```
ac_fft_dit_r2_sdf<128, 32, 19, 18, 2> fft_design1;
fft_design1.run(instream, outstream);
```

5. Keep in mind that in order to fully flush out the pipeline, the user must also write in an addition 128 placeholder values (same as the number of FFT points) and call the run function again.

3.10.3. Signal to Quantization Noise Ratio (SQNR) Results

SQNR plots are shown in Illustration 33. The x axis represents number of FFT points and y axis represents the SQNR in decibel (dB), for different I/O precision values. 19-bit twiddle factors are used, and 2 bits are always allocated for the I/O integer part.

The input passed to the Catware Blocks to measure the SQNR for this design is in the form of a mixed tone sine wave with 3 frequency components. For this input, reducing the I/O precision by one bit results in approximately 6dB reduction in SQNR. Bear in mind that this may not hold true for other kinds of inputs.

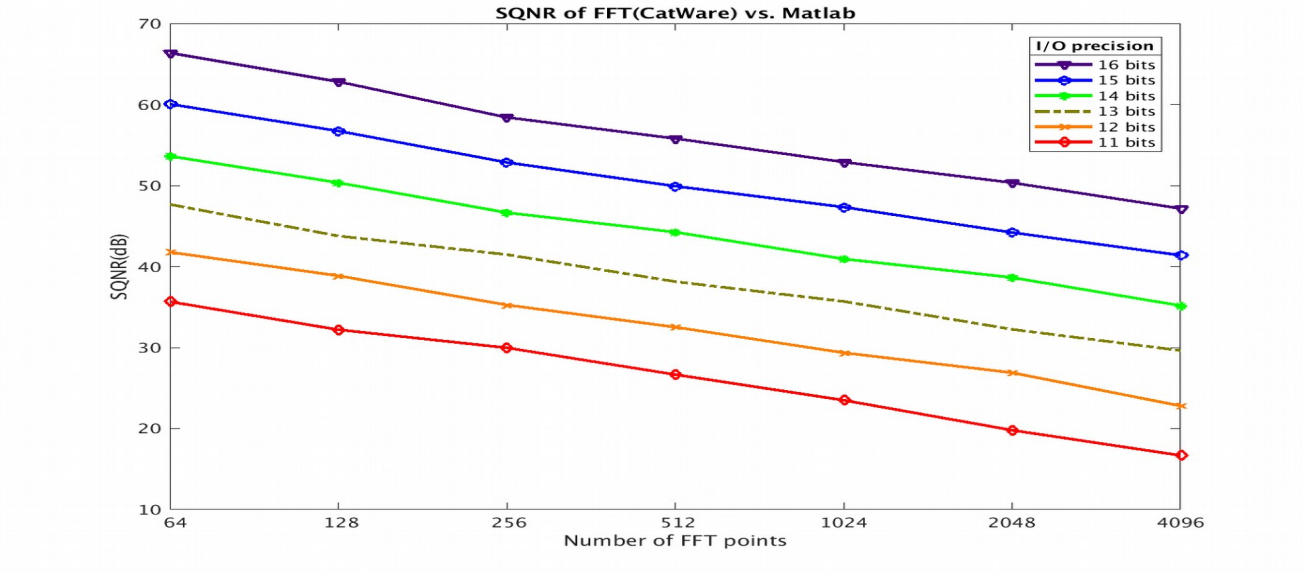


Illustration 33: SQNR Plots for Different Static Characteristics