UG9001

FPGA FFT Library

User guide

v0.1

DRAFT

Table of Contents

Introduction

The OwOComm FFT library implements several FFT architectures including Single Path Delay Feedback (SPDF) and Bailey's 4-step FFT. Multiple architectures may be combined to form a hybrid FFT core. Large FFTs in external memory are supported. Cores are generated using a code generator.

Features

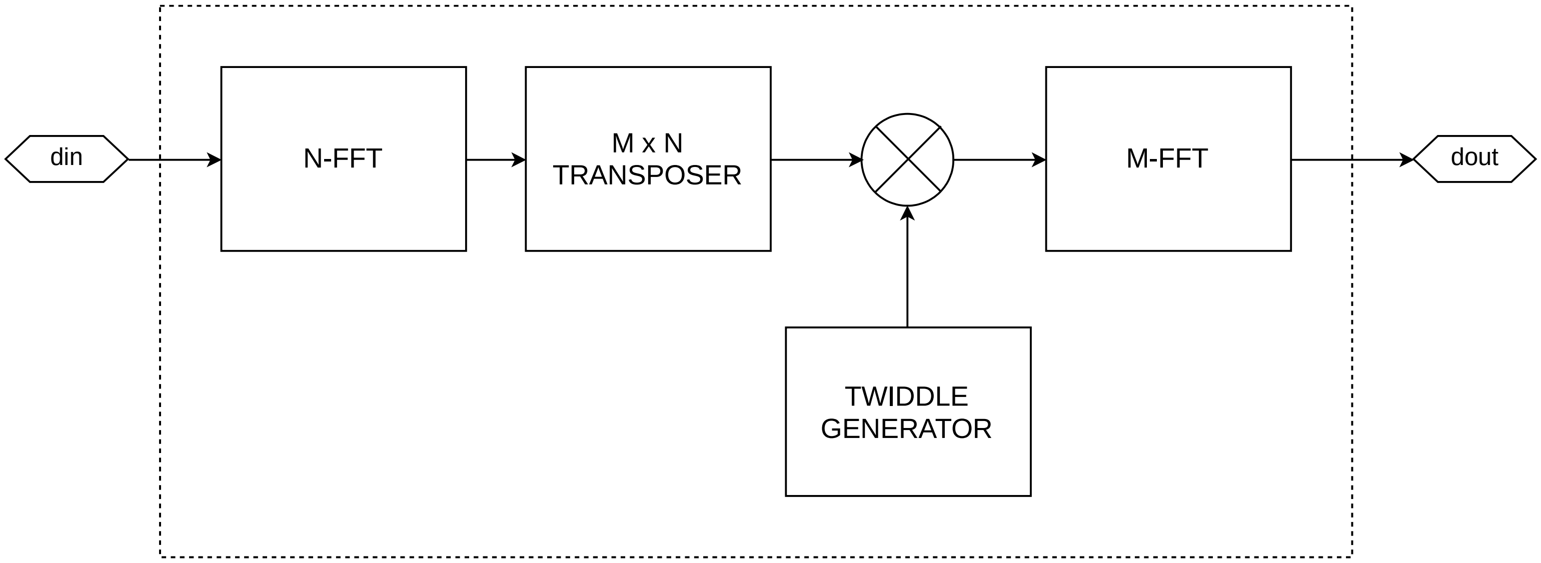
- Power of 2 FFT sizes
- Fixed point
- Scaling options: unscaled, divide by N, divide by sqrt(N)
- Rounding or truncation after each butterfly
- Large FFT sizes (up to 1G points) using external memory via AXI interface
- Automatically generated wrappers for natural order input/output
- Support for multiple interleaved channels in generated wrapper
- Optimized for data width up to 35 bits and twiddle width up to 24 bits
- Optimized for Xilinx® 7-series, Ultrascale, and Ultrascale+ FPGA families

Architectural overview

Bailey’s 4-step FFT

The Cooley-Tukey algorithm can be most generally described as subdividing a size N*M FFT into sub-FFTs of size N and M. M FFTs of size N are performed on the input data, the results are rotated by various angles (multiplication by roots of unity or twiddle factors), and finally N FFTs of size M are performed.

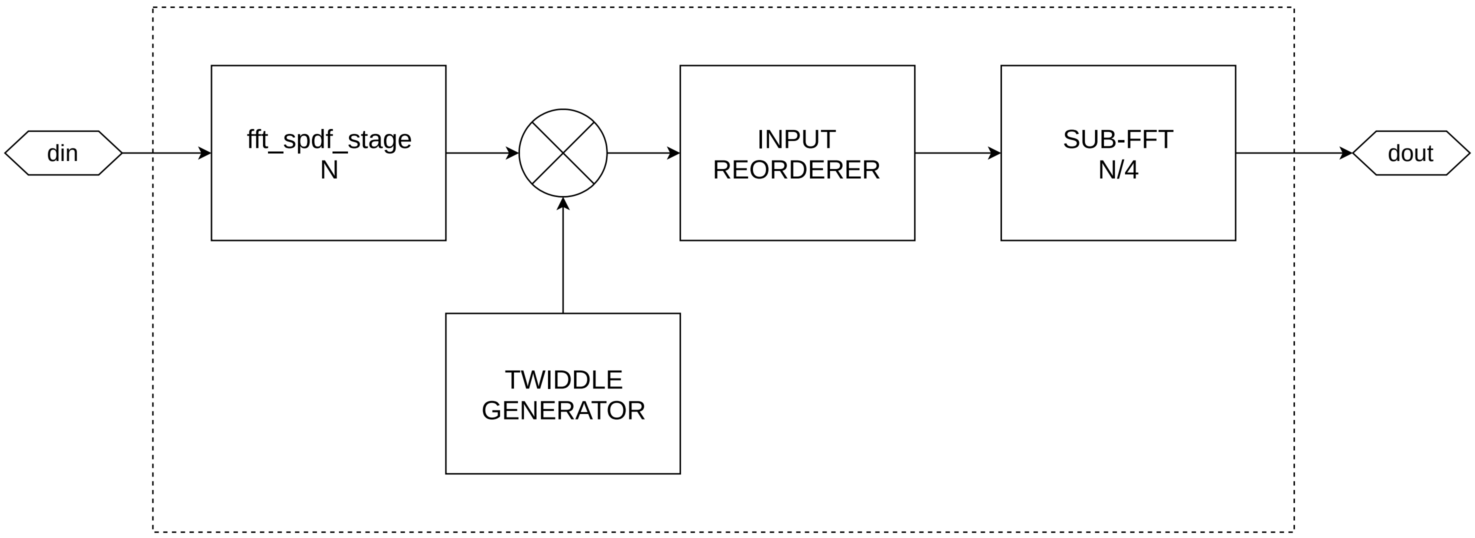
More precisely, the input data in natural order is put into a M columns by N rows matrix in row-major order. FFTs are performed on the columns, the resulting matrix is multiplied by twiddle factors, and FFTs are then performed on the rows. The results (in natural order) are read from the matrix in column major order.



Input and output transpose steps are omitted and data input/output are in transposed order. However because the sub-FFTs may take data in non-natural order, the final input and output order is best described as an address bit permutation. Generated FFT cores will have this permutation in the code comments, and generated reorderers will take care of correct data ordering.

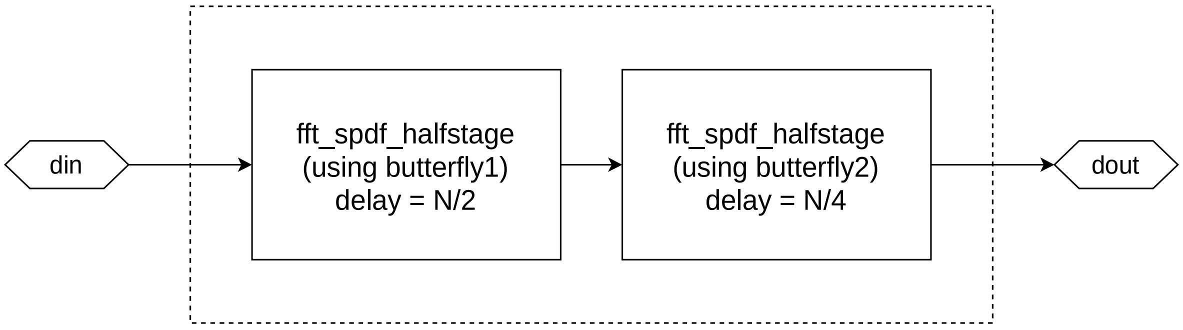
Single Path Delay Feedback (SPDF)

The SPDF FFT architecture is conceptually similar to the Bailey's formulation in that input data is viewed as a M x 4 matrix, butterfly operations are performed on the columns, and a size M sub-FFT is performed.



fft_spdf_stage

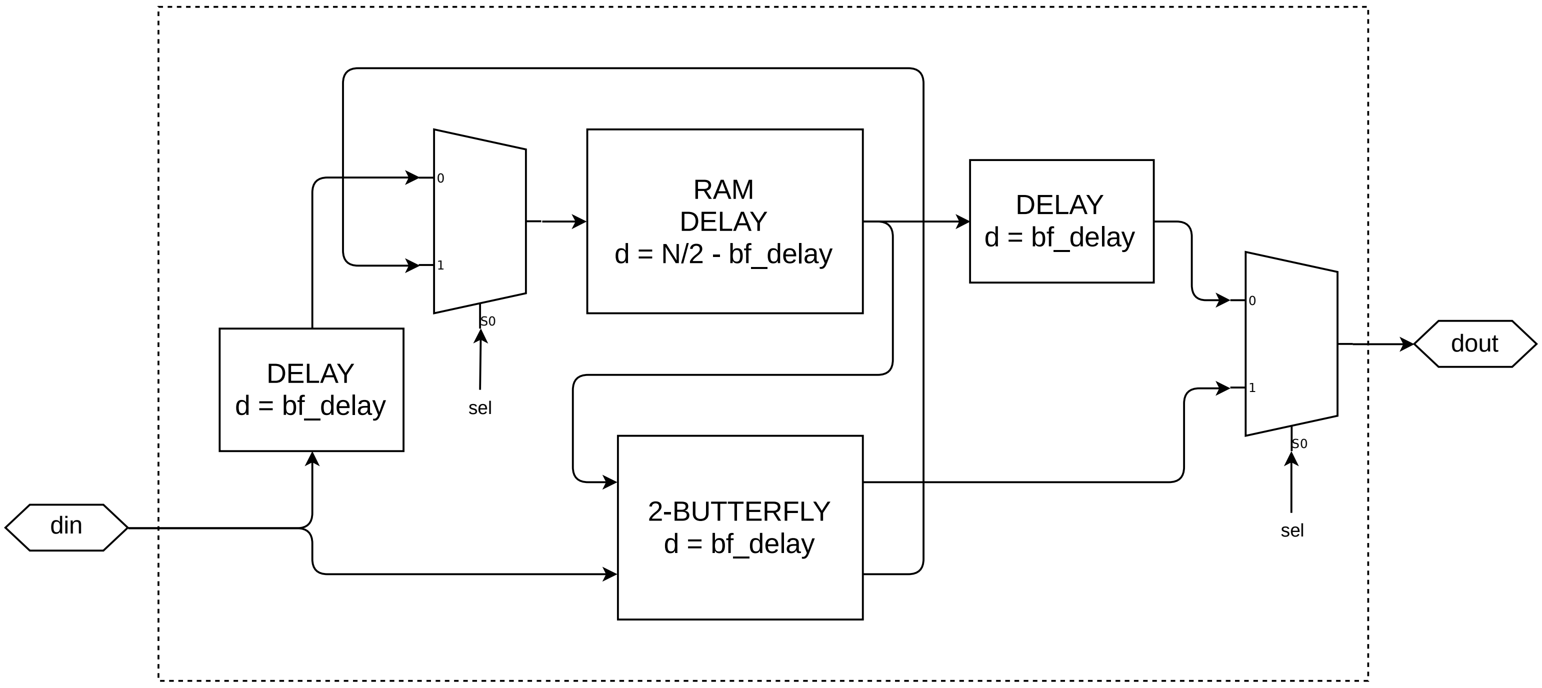
The SPDF butterfly stage is split into two stages, each regrouping its input data into a M x 2 matrix, performing 2-butterfly operations on its columns, and reading out the rows.



- Functionally equivalent to Nx4 transposer followed by 4-FFT followed by 4xN transposer

fft_spdf_halfstage

The SPDF half stage performs a 2-butterfly operation on the columns of a M x 2 matrix input in row major order, while also outputting the results in row major order.



During the first half of the frame (first N/2 cycles), sel is set to 0. Old data is flushed out while the first row of input fills the delay memory.

During the second half (next N/2 cycles) sel is 1 and the butterfly is active, reading the first row from the delay memory concurrently with the second row being input from din. The upper butterfly output (logically first row) is output to dout while the lower output (second row) is stored into the delay line, to be flushed out next frame.

Designing with the core

Library primitives

To use the library it is required to import the `fft_types` package:

```
use work.fft_types.all;
```

Data types

Complex values throughout the library are represented using the `complex` type, which is defined as:

```
-- maximum supported bit width of the signed integers in a complex value.
constant COMPLEXWIDTH: integer := 48;

type complex is record
      re: signed(COMPLEXWIDTH-1 downto 0);
      im: signed(COMPLEXWIDTH-1 downto 0);
end record;
```

`COMPLEXWIDTH` limits the maximum precision of a complex value. In FFT cores the precision is further limited by a configurable data bits parameter.

The following operators are defined for the `complex` type:

```
function "+" (Left, Right: complex) return complex;
function "-" (Left, Right: complex) return complex;
function "-" (Right: complex) return complex;
function "*" (Left: complex; Right: integer) return complex;
function "/" (Left: complex; Right: integer) return complex;
function to_complex (re,im: integer) return complex;
function to_complex (re,im: signed) return complex;
function complex_re(val: complex; bits: integer) return signed;
function complex_im(val: complex; bits: integer) return signed;
```

Clocking and data framing

The generated cores run from a single clock and has no clock enable input. To gate the operation of the core it is required to use clock gating (e.g. using a BUFGCE).

Data input is synchronized with and framed by the phase signal, which is required to be a monotonically increasing counter that wraps around at one frame length. When phase = 0, din should be set to the 0th complex value in this frame.

Data output follows data input after a fixed delay. The delay is documented in the code comments in all generated cores.

Phase is required to be monotonic for at least 32 cycles before the input of the first value in a frame, and remain monotonic until the output for that frame is complete for that frame to be considered valid. Violating this requirement can lead to incorrect data output but the core can be reset by restarting the phase counter at N-32.

Generating cores

gen_fft.py

This is the main FFT core generator which generates cores, reorderers, and wrappers.

To generate the code for one of the layout definitions, run:

```
./gen_fft.py fft INSTANCE_TO_GENERATE
```

For example:

```
./gen_fft.py fft fft1024_wide
```

To generate input/output reorderers for a FFT core instance, run:

```
./gen_fft.py reorderer INSTANCE_TO_GENERATE
```

To generate natural order wrappers, run:

```
./gen_fft.py wrapper INSTANCE_TO_GENERATE
```

Composing FFT layouts

There are several pre-defined layouts in gen_fft_layouts.py. You can add new FFT sizes or custom layouts by editing gen_fft_layouts.py.

An example of a layout definition:

```
fft1024_wide = \
     FFT4Step(1024,
          FFT4Step(64,
               FFT4Step(16,
                    fft4_scale_none,
                    fft4_scale_none),
               fft4_scale_div_sqrt_n),
          FFT4Step(16,
               fft4_scale_div_n,
               fft4_scale_div_n));

fft1024_wide.setOptionsRecursive(rnd=True, largeMultiplier=True)
```

An FFT layout definition is simply an instance of one of the composite FFT classes, which are FFT4Step and FFTSPDF.

FFT4Step

FFT4Step represents a Bailey's 4-step FFT, which combines two sub-FFTs of size M and N to form a larger FFT of size M * N.

Constructor arguments

- `N` – integer – FFT size.
- `sub1` - object - sub-FFT instance 1. Can be a base FFT, FFT4Step, or FFTSPDF.
- `sub2` - object - sub-FFT instance 2. Can be a base FFT, FFT4Step, or FFTSPDF.
- `twiddleBits` – precision of the twiddle generator. It is recommended to leave this unset. The default value is 'twBits' and twiddle precision takes on the value of the top level entity parameter 'twBits'.
- `rnd` – whether to enable rounding after twiddle multiplication. Does not affect sub-FFTs or inner butterflies. It is recommended to leave this unset and to set this value recursively using setOptionsRecursive().
- `largeMultiplier` – whether to use the double width multiplier implementation (using 2 DSP48E1s for a 25x35 multiplier). It is recommended to leave this unset and to set this value recursively using setOptionsRecursive().

Methods

- `setOptions(rnd, largeMultiplier)` – sets options non-recursively. See above for a description of the options.
- `setOptionsRecursive(rnd, largeMultiplier)` – sets options recursively. See above for a description of the options.
- `delay()` - returns the total delay in clock cycles of this FFT core.
- `inputBitOrder()` - returns the input data order as an address bit permutation (list of integers)
- `outputBitOrder()` - returns the output data order as an address bit permutation (list of integers)

FFTSPDF

FFTSPDF represents a Single Path Delay Feedback FFT. The sub-FFT must have a size of N/4. The internal 4-butterfly does not do any scaling. If scaling is required it must be done in the sub-FFT.

Constructor arguments

- `N` – integer – FFT size.
- `sub1` - object - sub-FFT instance 1. Can be a base FFT, FFT4Step, or FFTSPDF.
- `twiddleBits` – see FFT4Step
- `rnd` – see FFT4Step
- `largeMultiplier` – see FFT4Step

Methods

- `setOptions(rnd, largeMultiplier)` – sets options non-recursively. See FFT4Step for a description of the options.
- `setOptionsRecursive(rnd, largeMultiplier)` – sets options recursively. See FFT4Step for a description of the options.
- `delay()` - returns the total delay in clock cycles of this FFT core.
- `inputBitOrder()` - returns the input data order as an address bit permutation (list of integers)
- `outputBitOrder()` - returns the output data order as an address bit permutation (list of integers)

Base FFT instances

These are to be used as the base cases when composing a larger FFT core.

The base FFT instances are:

- fft2_scale_none – A size 2 FFT. No scaling is performed.
- fft2_scale_div_n – A size 2 FFT. Scales by ½.
- fft4_scale_none – A size 4 FFT. No scaling is performed.
- fft4_scale_div_sqrt_n – A size 4 FFT. Scales by ½.
- fft4_scale_div_n - A size 4 FFT. Scales by ¼.

Methods

These methods apply to all base FFT instances:

- `delay()` - returns the delay in clock cycles of this FFT butterfly.
- `inputBitOrder()` - returns the input data order as an address bit permutation (list of integers)
- `outputBitOrder()` - returns the output data order as an address bit permutation (list of integers)

Generated FFT core description

```
-- data input bit order: XXXXX
-- data output bit order: XXXXX
-- phase should be 0,1,2,3,4,5,6,...
-- delay is XXX
entity INSTANCENAME is
     generic(dataBits: integer := 24;
                twBits: integer := 12);
     port(clk: in std_logic;
          din: in complex;
          phase: in unsigned(...);
          dout: out complex
          );
end entity;
```

Parameters

- **dataBits** – integer
  The input and output precision. A dataBits of 12 means 12 bits real and 12 bits imaginary.
- **twBits** – integer
  The twiddle ROM precision.
  The internal twiddle multipliers see a width of twBits + 1. If targeting a A*B size multiplier, set twBits to A-1. For example to target a dual DSP48E1 25x35 multiplier implementation, dataBits should be <= 35, and twBits should be <= 24 (or vice versa).

Ports

- **clk** – std_logic – The input clock that all ports are synchronous to.
- **din** – complex – Input data
- **phase** – unsigned – A monotonically increasing counter that is synchronized with din
- **dout** – complex – Output data

Generated reorderer description

```
-- phase should be 0,1,2,3,4,5,6,...
-- delay is XXX
entity INSTANCENAME_ireorderer1 is
     generic(dataBits: integer := 24;
               twBits: integer := 12);
     port(clk: in std_logic;
          din: in complex;
          phase: in unsigned(...);
          dout: out complex
          );
end entity;

-- phase should be 0,1,2,3,4,5,6,...
-- delay is XXX
entity INSTANCENAME_oreorderer1 is
     generic(dataBits: integer := 24;
               twBits: integer := 12);
     port(clk: in std_logic;
          din: in complex;
          phase: in unsigned(...);
          dout: out complex
          );
end entity;
```