OwOComm

Intellectual communism reinvented!

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

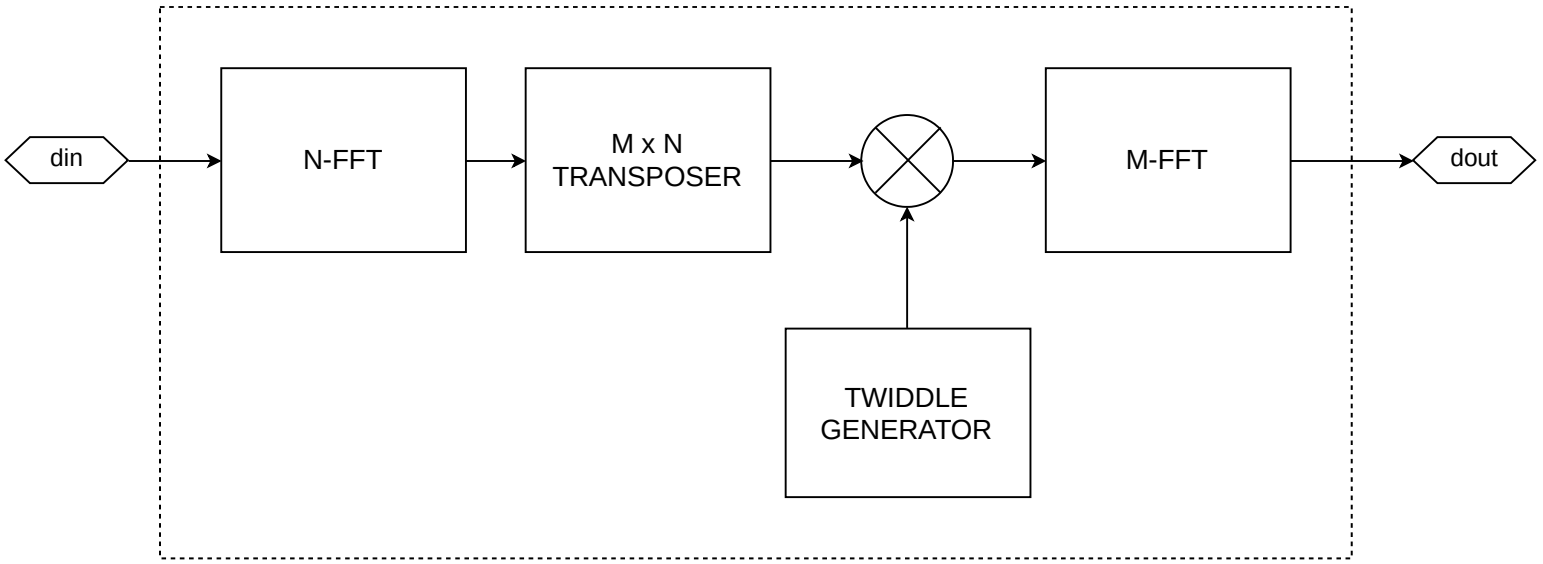
Architectural overview

The OwOComm FPGA FFT library implements two main algorithms which are the Bailey's 4-step and the Single Path Delay Feedback (SPDF) formulations. Both are instances of the Cooley-Tukey algorithm. Generated cores can use a mix of these two formulations for a tradeoff between logic area and block RAM usage.

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.

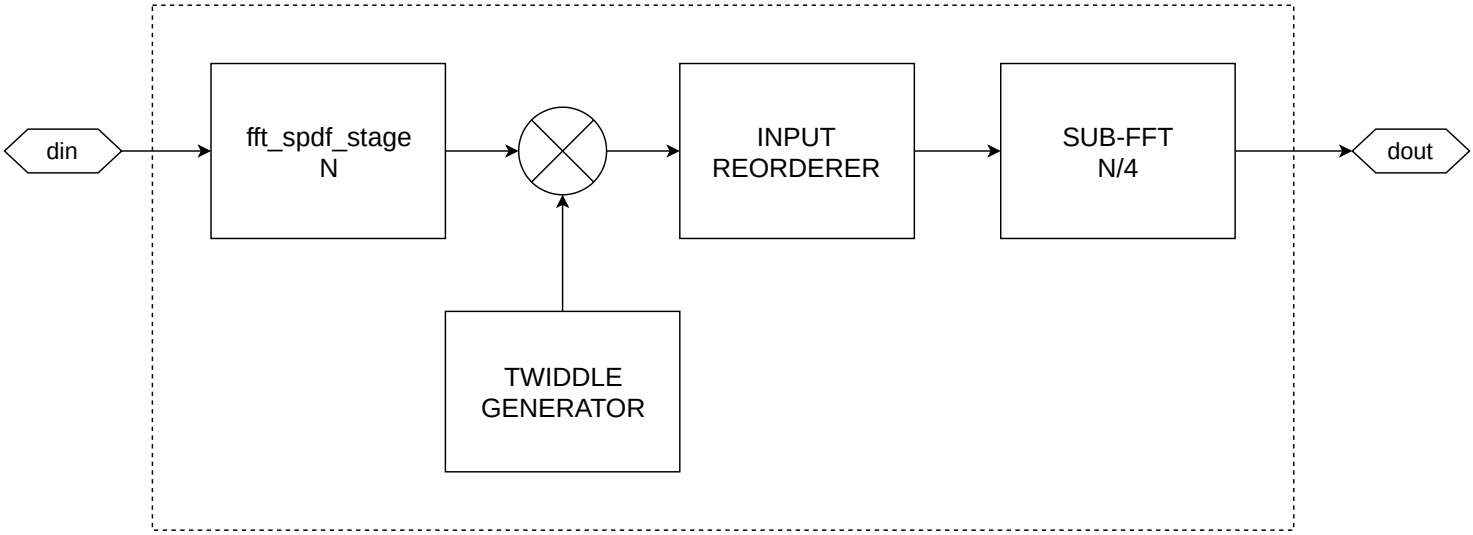
More precisely, the input data in natural order is put into a M columns by N rows matrix in row-major order. FFTs are performed on the columns, the resulting matrix is multiplied by twiddle factors, and FFTs are then performed on the rows. The results (in natural order) are read from the matrix in column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

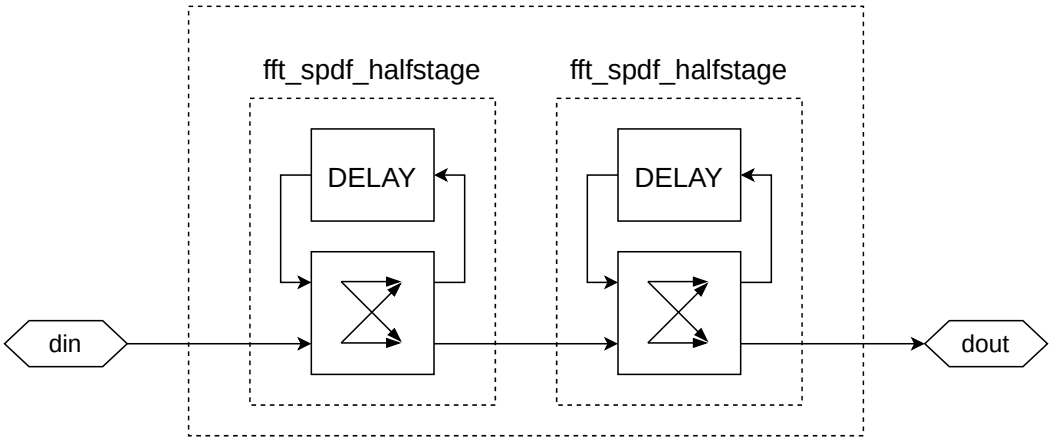
Single Path Delay Feedback (SPDF)

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

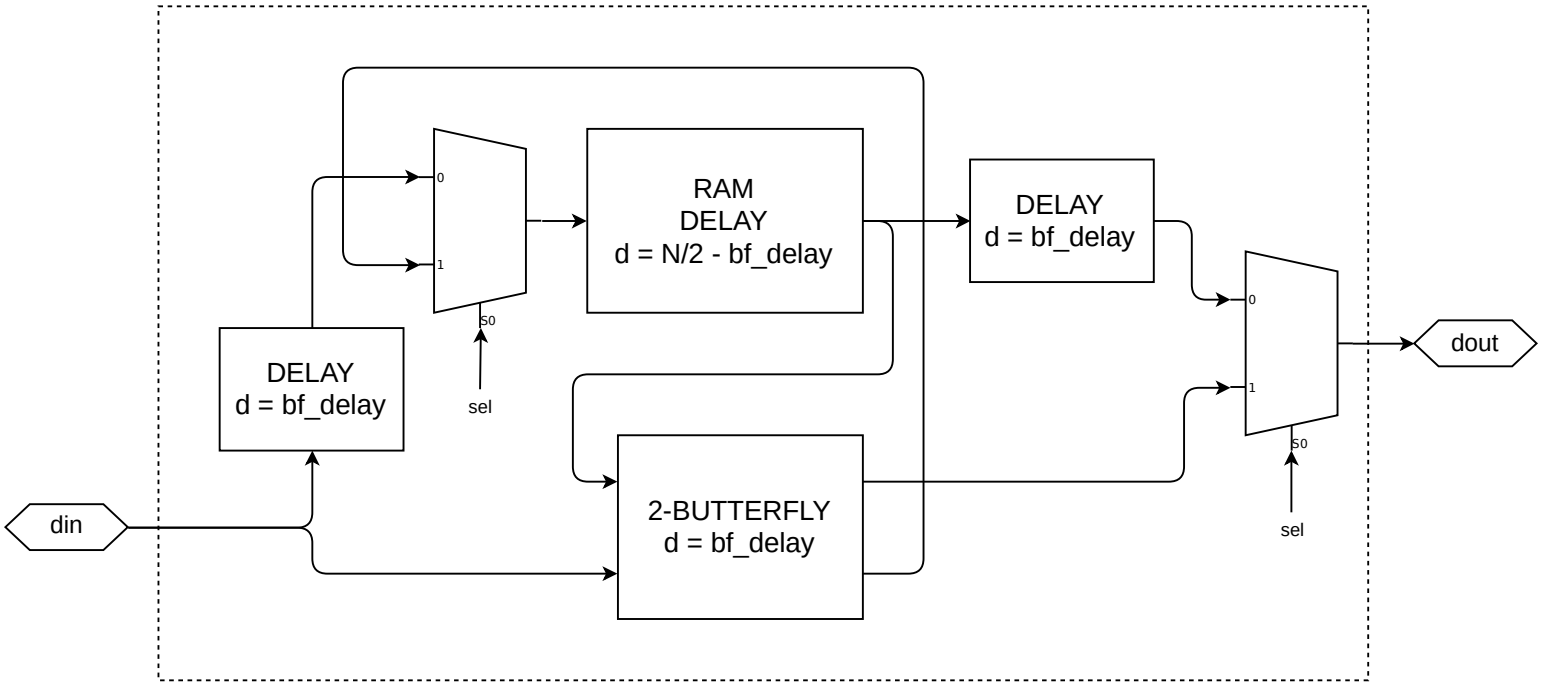
The SPDF butterfly stage is split into two stages, each regrouping its input data into a M x 2 matrix, performing 2-butterfly operations on its columns, and reading out the rows.



- 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 LSBs of `re` and `im` are used when the data width is less than `COMPLEXWIDTH`.

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).

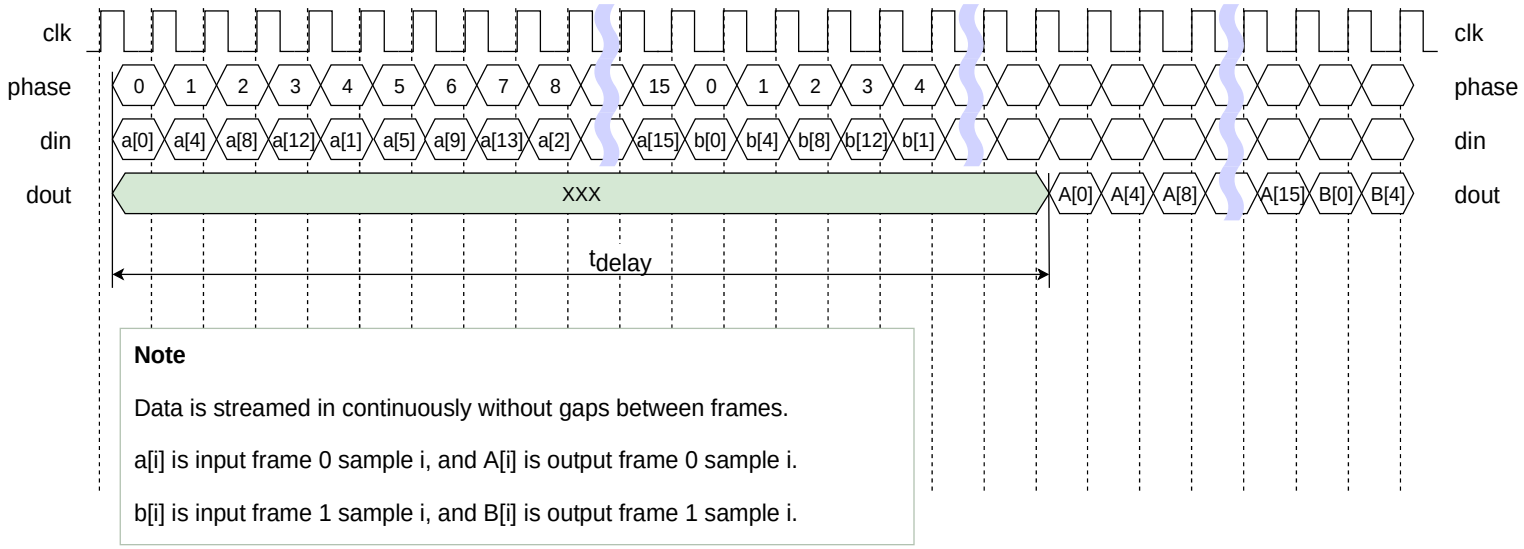
Data input is synchronized with and framed by the phase signal, which is required to be a monotonically increasing counter that wraps around at one frame length. When phase = 0, din should be set to the 0^{th} complex value in this frame.

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

Data input/output order are defined by a bit permutation. The exact permutation is documented in the comments at the top of each generated core.

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.

16-FFT timing example



Note

Data is streamed in continuously without gaps between frames.

a[i] is input frame 0 sample i, and A[i] is output frame 0 sample i.

b[i] is input frame 1 sample i, and B[i] is output frame 1 sample i.

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 predefined layouts in gen_fft_layouts.py. You can add new FFT sizes or custom layouts by editing this file.

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.setMultiplier(largeMult)
```

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.
- `multiplier` – object – multiplier type to use for twiddle multiplication. Must be an instance of `gen_fft_modules.Multiplier`. Built in instances are `defaultMult` (generic inferred RTL multiply) and `largeMult` (uses 2 DSP48E1s for a 25x35 multiplier). It is recommended to leave this unset and to set this value recursively using `setMultiplier()`.
- `twiddleBits` – integer – 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'.

Methods

- `setMultiplier(multiplier, recursive=True)` – sets the twiddle multiplier. See above for a description of the multiplier argument. If recursive is true this will affect sub-FFT instances as well.
- `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.
- `multiplier` – object – see FFT4Step
- `twiddleBits` – object – see FFT4Step

Methods

- `setMultiplier(multiplier, recursive=True)` – sets the twiddle multiplier. See FFT4Step for a description of the multiplier argument.
- `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)

Layout recommendations

When composing a new layout, the following should be considered:

- Is natural order input required? If both input and output are allowed to be in arbitrary order (defined by an address bit permutation) it is recommended to use FFT4Step throughout.
- Are multiple (interleaved) channels required? If so, reorder memory is required on both input and output side, and input/output reordering comes at no additional memory usage. Therefore using FFT4Step may still be indicated.
- FFTSPDF provides the lowest memory usage when input must be in natural order and is single channel.
- FFT4Step provides the lowest LUT usage but both data input and output are in non-linear order, therefore reorderers may be required.

When composing a natural order input single channel FFT, use one or two layers of FFTSPDF and switch to FFT4Step for the next sub-FFT.

```
fft1024_spdf_wide = \
     FFTSPDF(1024,
          FFTSPDF(256,
               FFT4Step(64,
                    FFT4Step(16,
                         fft4_scale_div_sqrt_n,
                         fft4_scale_div_n),
                    fft4_scale_div_n)));
```

When composing a layout using FFT4Step, the following should be considered:

- Keep the size of sub1 and sub2 close to sqrt(N) for minimal memory usage.
- If sub1 and sub2 must be different sizes (e.g. non-perfect-square N), sub2 should be the smaller sub-FFT. There is a reorderer before sub-FFT2 because the FFT4Step twiddle address generator requires linear column order (to avoid using a multiplier).

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
                inverse: boolean := true);
     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).
- **inverse** – boolean
  Set to true for inverse transform (twiddle phase 2πij/N) or false for forward transform (twiddle phase -2πij/N).

Ports

- **clk** – std_logic – The 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

Entity naming: `INSTANCENAME_ireordererN`, where INSTANCENAME is the name of the FFT layout instance in the python file and N is the number of interleaved channels. ireorderer is the input reorderer and oreorderer is the output reorderer.

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

Parameters

- **dataBits** – integer
  The data width. A dataBits of 12 means 12 bits real and 12 bits imaginary.

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.

Large FFTs in external memory

This chapter describes the OwOComm DRAM-optimized external memory FFT implementation.

Overview

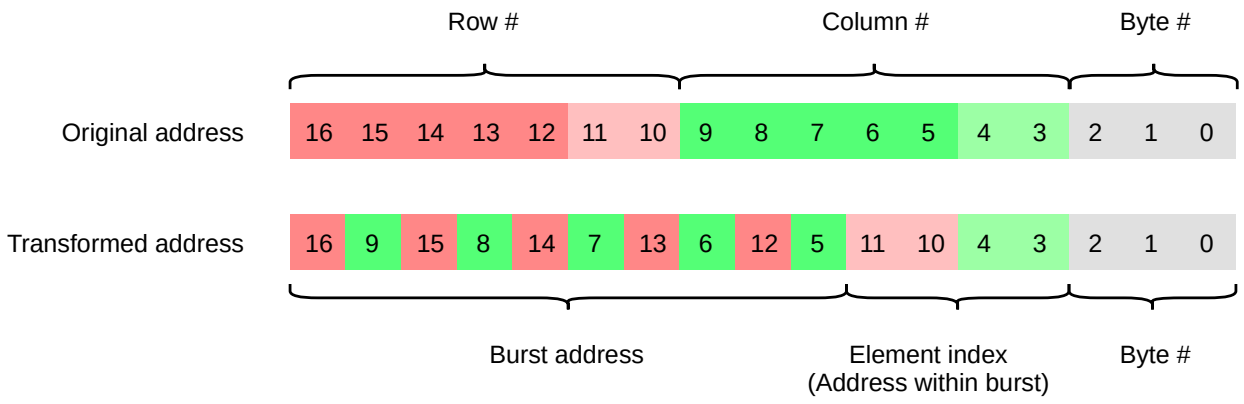
Using the Bailey's 4-step FFT algorithm it is possible to perform a size N^2 FFT using a single FFT core of size N and two passes over the data. See section Bailey's 4-step FFT for an overview of the algorithm. The algorithm requires several transpose steps which can be omitted if the data matrix can be read in transposed order efficiently.

Traditional matrix orderings (row or column major order) result in optimal memory access patterns for only one direction of traversal. The 2-pass variant of the Bailey's FFT algorithm requires accessing the matrix in both row-first and column-first order of traversal. We introduce a matrix ordering that results in more optimal memory access patterns on DRAM (maximizing the number of read/write bursts per open row).

In benchmarks with matrix size 1024 x 1024 on a Zynq-7010 we have found the achieved memory bandwidth with this ordering to be around 90% of the sequential access memory bandwidth.

Row/column interleaved order

In traditional row-major order a DRAM page occupies a portion of a matrix row or a few rows. This results in cross-page strided accesses for column-first order of traversal. For efficient DRAM access each DRAM page must occupy a sub-block of the matrix. This can be achieved by simply reordering bits in the row-column address. In the OwOComm implementation we have adopted the row-column interleaved order which achieves optimal ordering independent of page size. The ordering is most simply described as interleaving the bits of the row and column addresses:



The resulting matrix order can be visualized as follows:

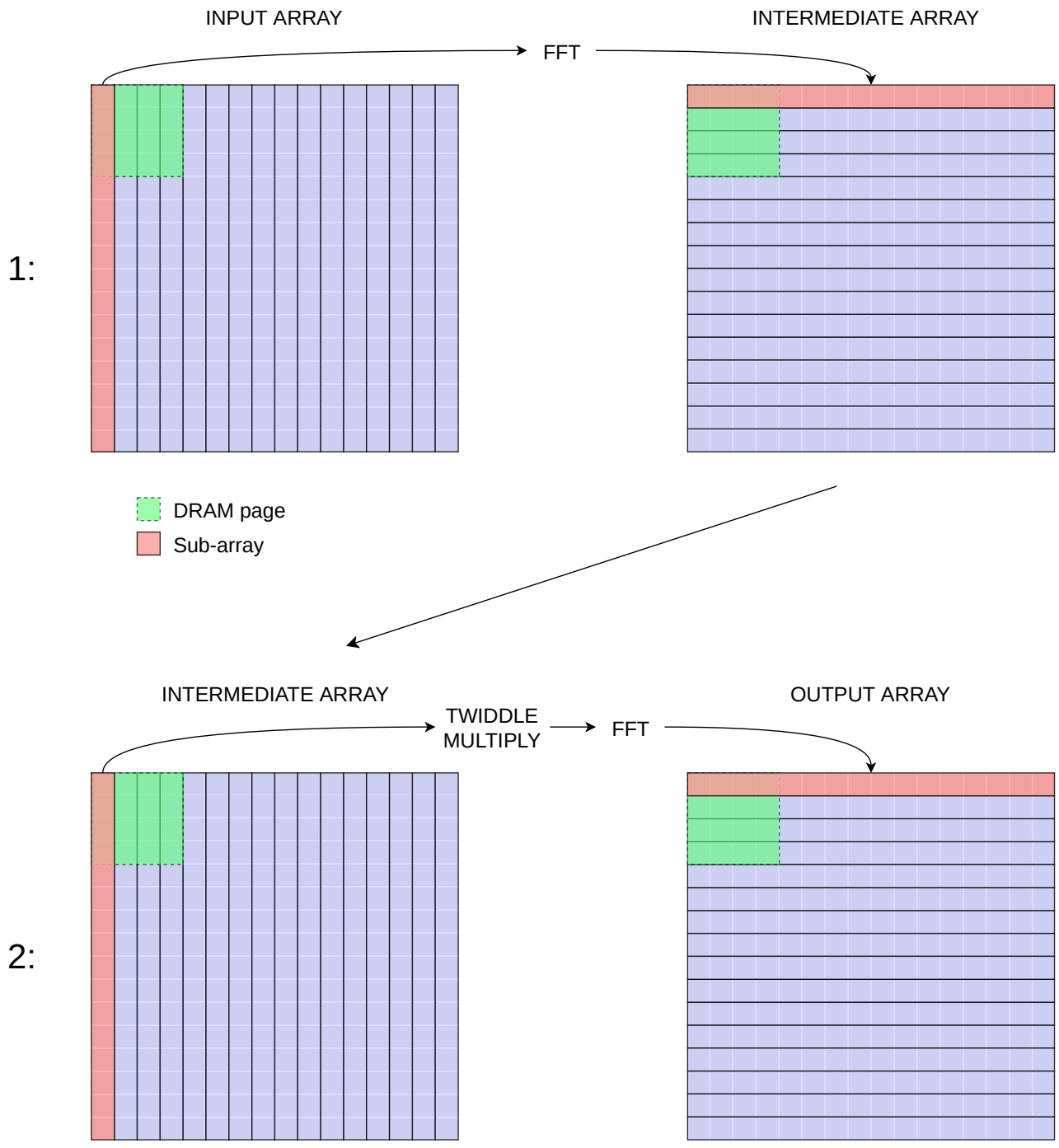
0	1	4	5	16	17	20	21	64	65	68	69	80	81	84	85
2	3	6	7	18	19	22	23	66	67	70	71	82	83	86	87
8	9	12	13	24	25	28	29	72	73	76	77	88	89	92	93
10	11	14	15	26	27	30	31	74	75	78	79	90	91	94	95
32	33	36	37	48	49	52	53	96	97	100	101	112	113	116	117
34	35	38	39	50	51	54	55	98	99	102	103	114	115	118	119
40	41	44	45	56	57	60	61	104	105	108	109	120	121	124	125
42	43	46	47	58	59	62	63	106	107	110	111	122	123	126	127
128	129	132	133	144	145	148	149	192	193	196	197	208	209	212	213
130	131	134	135	146	147	150	151	194	195	198	199	210	211	214	215
136	137	140	141	152	153	156	157	200	201	204	205	216	217	220	221
138	139	142	143	154	155	158	159	202	203	206	207	218	219	222	223
160	161	164	165	176	177	180	181	224	225	228	229	240	241	244	245
162	163	166	167	178	179	182	183	226	227	230	231	242	243	246	247
168	169	172	173	184	185	188	189	232	233	236	237	248	249	252	253
170	171	174	175	186	187	190	191	234	235	238	239	250	251	254	255

DRAM burst

The number in each cell is the memory address, and the layout on screen represents the logical matrix layout. It can be seen that a page of size P always occupies a sqrt(P) x sqrt(P) sub-block (for perfect square P) or a 2Q x Q sub-block (for $P = 2Q^2$) of the matrix.

Algorithm details

The FFT implementation takes an array in the form of a matrix already in row-column interleaved order and performs two passes over the data. The two passes are illustrated as follows:



The first step performs sub-FFTs on each input column and writes rows to the intermediate matrix.

The second step takes each column from the intermediate matrix, performs twiddle multiplication and sub-FFTs, then writes the rows of the output matrix. The results can be read out by reading the columns of the output matrix in order.

VHDL library reference guide

This chapter describes the modules contained in the VHDL library. Some of these are used by generated cores.

List of modules

Name	Description
complexMultiply2	Complex multiplier
complexRam	Dual port RAM of complex values
complexRamLUT	Dual port RAM of complex values, LUTRAM implementation
complexRamDelay	Delay line of complex values
dsp48e1_complexMultiply	Complex multiplier using 2 x DSP48E1
fft2_serial	2-FFT butterfly
fft4_serial7	4-FFT butterfly
fft_spdf_halfstage	SPDF 2-butterfly stage
fft_spdf_stage	SPDF 4-butterfly stage
multiplyAdd	Inferred signed integer multiply-add
reorderBuffer	Data reorderer
transposer	Data interleaver
transposer4	2x2 data interleaver

complexMultiply2 complex_multiply2.vhd

Complex multiplier using inferred multiply-adds. Optimal for signed multiplies that fit into a single DSP48E1 or DSP48E2. Delay from in1/in2 to out1 is 6 cycles.

Data is scaled such that when in1 is equal to “01000...”, output is equal to the most significant bits of in2. For example “01000...” * “00011...” = “00011...”.

```
entity complexMultiply2 is
     generic(in1Bits,in2Bits,outBits: integer := 8;
                round: boolean := true);
     port(clk: in std_logic;
                in1,in2: in complex;
                out1: out complex
                );
end entity;
```

Parameters

- **in1Bits, in2Bits** – integer – The width of the components of the input data.
- **outBits** – integer – The width of the components of the output data.
- **round** – boolean – This parameter is ignored and rounding is always performed.

Ports

- **clk** – std_logic – Clock that all ports are synchronous to.
- **in1, in2** – complex – Input operands.
- **out1** – complex – Output product.

complexRam complex_ram.vhd

This module infers a dual port dual clock RAM of complex values.

The read delay is 2 cycles and the write delay is 1 cycle.

If rdclk and wrclk are the same clock, reads and writes may not be to the same address simultaneously after accounting for read/write delay.

```
entity complexRam is
      generic(dataBits: integer := 8;
                      -- real depth is 2^depth_order
                      depthOrder: integer := 9);
      port(rdclk,wrclk: in std_logic;
                  -- read side; synchronous to rdclk
                  rdaddr: in unsigned(depthOrder-1 downto 0);
                  rddata: out complex;

                  --write side; synchronous to wrclk
                  wren: in std_logic;
                  wraddr: in unsigned(depthOrder-1 downto 0);
                  wrdata: in complex);
end entity;
```

Parameters

- **dataBits** – integer – The width of the real and imaginary parts.
- **depthOrder** – integer – Log2 of the depth of the RAM.

Ports

- **rdclk, wrclk** – std_logic – Read and write clock.
- **rdaddr** – unsigned – Read address.
- **rddata** – complex – Read data.
- **wren** – std_logic – Write enable.
- **wraddr** – unsigned – Write address.
- **wrdata** – complex – Write data.

complexRamLUT complex_ram_lut.vhd

Same as complexRam, but always implemented in LUTRAM.

complexRamDelay complex_ram_delay.vhd

A delay line of complex values. May be implemented in shift registers for small lengths or RAM for longer lengths.

```
entity complexRamDelay is
      generic(dataBits, delay: integer);
      port(clk: in std_logic;
           din: in complex;
           dout: out complex);
end entity;
```

Parameters

- **dataBits** – integer – The width of the real and imaginary parts.
- **delay** – integer – Number of clock cycles to delay the data.

Ports

- **clk** – std_logic – Input and output clock.
- **din** – complex – Input data.
- **dout** – complex – Output data.

dsp48e1_complexMultiply dsp48e1_multadd.vhd

Complex multiplier using inferred multiply-adds. Optimal for signed multiplies that fit into two DSP48E1 or DSP48E2.

Delay from in1/in2 to out1 is 9 cycles.

Data scaling is identical to complexMultiply2.

Maximum input width is 25 bits for in1 and 35 bits for in2.

```
entity dsp48e1_complexMultiply is
     generic(in1Bits, in2Bits, outBits: integer := 8;
                round: boolean := true);
     port(clk: in std_logic;
                in1,in2: in complex;
                out1: out complex
                );
end entity;
```

Parameters

- **in1Bits, in2Bits** – integer – The width of the components of the input data.
- **outBits** – integer – The width of the components of the output data.
- **round** – boolean – Whether to perform rounding when scaling the output.

Ports

- **clk** – std_logic – Clock that all ports are synchronous to.
- **in1, in2** – complex – Input operands.
- **out1** – complex – Output product.

fft4_serial7 fft4_serial7.vhd

This module implements a size 4 FFT. Input is in natural order and output is in bit reversed order (0, 2, 1, 3).

Delay is 11 cycles.

```
entity fft4_serial7 is
     generic(dataBits: integer := 18;
               scale: scalingModes := SCALE_NONE;
               round: boolean := true;
               inverse: boolean := true);

     port(clk: in std_logic;
          din: in complex;
          phase: in unsigned(1 downto 0);
          dout: out complex);
end entity;
```

Parameters

- **dataBits** – integer – The input and output precision.
- **scale** – enum – The scaling of the output data. Valid values are SCALE_DIV_N, SCALE_DIV_SQRT_N, and SCALE_NONE.
- **round** – boolean – Whether to perform rounding when scaling the output.
- **inverse** – boolean – Set to true for inverse transform (twiddle phase 2πij/N) or false for forward transform (twiddle phase -2πij/N).

Ports

- **clk** – std_logic – The 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.

Revision history

Date, version	Author	Description
2019/07/15 – v0.1	gabu-chan	Initial version.