# DSP RTL Library (DRL): A Verilog Approach

Ahmed Shahein

August 30, 2019

# Contents

# Preface

I am delighted and keen to share my passion towards Digital Signal Processing (DSP) with other passionate and learners. I am grateful to the open source community and to my professors, instructors, and colleagues who helped me through the last decade and a half to learn and fall in love with DSP.

I have been focusing on the hardware side of DSP, in particular front-end or RTL design and implementation. Nevertheless, I have digested Matlab/Octave and SystemC sufficient enough to model the developed DSP components, models and even systems as well. I would like to share my humble experience with you, hoping, to help or even assist you to learn, discover, facilitate either or both DSP and Verilog.

I have very ambitious plans for the DSP RTL Library (DRL), and I hope to meet them all, if not alone may be with your appreciated help and support. I am looking forward to provide the most commonly used DSP components in generic parameterizable and synthesizable RTL code which can be used for either ASIC and/or FPGA development. Moreover, I am planning to provide a GRM "mostly Octave". The purpose of these models to be used for bit-true verification and building time-lossly systems. I have a wish to validate these modules on an FPGA, but I am a bit skeptical on that point due to time constraints. I am aiming to develop scripts which will take design parameters and generate the RTL. This is planned once the library is alpha release, i.e. stable and mature enough.

The DRL is targeting

- Verilog beginners and learners
- DSP beginners and learners
- System engineers seeking early area (gate count), power, and latency figures
- Software engineers seeking performance check comparing their algorithm on DSP processor to HW
- ASIC/FPGA designers seeking rapid prototyping

The DRL only focuses on implementation. Refer to DSP reference text books for theory and background, such as; [1], [2], [3], [4], [5], [6], [7], and [8]. Refer to reference text books for RTL coding such as [9] and [10].

In addition to text books, the following websites have been of a great value for me during the complete development cycle:

- www.dsprelated.com
- www.zipcpu.com

- www.dspguru.com
- www.wikipedia.com
- www.edaplayground.com

## *Let's Have Fun!*

# List of Abbreviations

| Abbreviation | Description |
|---|---|
| FIR | Finite Impulse Response |
| IIR | Infinite Impulse Response |
| CIC | Cascade Integrator Comb |
| MAC | Multiply ACCumulator |
| TF | Transposed-form |
| DF | Direct-form |
| PPD | PolyPhase Decimation |
| PPI | PolyPhase Interpolation |
| NCO | Numerically Controlled Oscillator |
| FCW | Frequency Control Word |
| SNR | Signal-to-Noise Ratio |
| SFDR | Spurious Free Dynamic Range |
| CORDIC | COordinate Rotation DIgital Computer |
| LFSR | Linear Feedback Shift Register |
| RTL | Register Transfer Logic |
| DSP | Digital Signal Processing |
| DUT | Device Under Test |
| STA | Static Time Analysis |
| CCW | Counter ClockWise |
| CW | ClockWise |
| HW | HardWare |
| SW | SoftWare |

Table 1: List of abbreviations for DRL.

# Chapter 1

# Overview

The DSP-RTL-Library (DRL) will provide synthesized RTL developed in Verilog 2001, test-bench developed in SystemVerilog 2012, and Octave golden reference models (GRM). The DRL will provide conventional implementation, i.e., unoptimized for a specific budget (area, power, or latency).

## 1.1 Coding Style

I recommend the following coding style which I used during the development of the DRL. The coding style is given in 1.1. The coding style is proposed to facilitate debugging and troubleshooting during both functional and formal verification. In addition to documenting the code for easier readability. Therefore, please, stick to it!

| **VHDL** | **Verilog** | **Name** | **Label** |
|---|---|---|---|
| in | input | i_ | • |
| out | output | o_ | • |
| inout | inout | io_ | • |
| generic | parameter | gp_ | |
| constant | localparam | c_ | • |
| signal | wire/reg | w_ | combinatorial |
| signal | wire/reg | r_ | sequential |
| variable | • | v_ | • |
| type | • | t_ | • |
| process | always | • | • |
| generate | generate | | g_ |
| function | function | • | func_ |
| procedure | task | • | proc_ |
| • | • | • | • |
| • | • | • | • |

Table 1.1: RTL (VHDL/Verilog) coding style.

The RTL coding is carried out in a flatten (1D) methodology. In other words, I have avoided fancy RTL features such as 2D arrays. This is due to the

| Status | Description | Abbreviation |
|---|---|---|
| Stable | Synthesized and verified | S |
| Validate | FPGA proven via lab measurements | V |
| Integrate | Tested within a complete functioning system | I |

Table 1.2: RTL status explanation.

fact that most of the open source tools do not support that feature for synthesis yet. It might made understanding the RTL code not as straight forward as it would be using the 2D arrays within the code, but it is fun to do it that way as well.

Some text books recommend avoiding using the (‘include) directive. Nevertheless, I decided to use it since it is quite convenient during development cycles.

It shall be noted that, my implementations are latency driven, i.e., the time required for an input sample to propagate from input port to output is minimized.

The following coding practices are applicable for all designs, unless explicitly stated otherwise:

- Asynchronous active low reset.
- Synchronous active high enable.
- Rising edge flip-flops are only used.
- Arithmetic computations are carried out based on fixed-point 2's complement data type.

The following design tips is viable for you:

- In case of single-bit input, set it to 2-bits for preserving the sign bit.
- In case of failed test-case (built or developed by you) during verification. Please check the test-bench especially the enabling and clocking scheme. Generating a synchronous enable with multiple clocks (in case of sample rate converters such as; decimation and interpolation) might be tricky.

The DRL offers the following status for each developed design, as given in 1.2:

## 1.2 Tool Chain

A set of open source tools have been used during development and verification of the DRL. In addition to commercially free limited versions such as Xilinx WebPack and Intel ModelSim Standard Edition versions. The associated open source tools are given in 1.3. The commercial free packages are shown in 1.4.

If there is a need to digest digital filtering and correlate between the various domains (time, frequency, and $z$), I would absolutely recommend the ASN Filter Designer provided by Advanced Solutions Nederland. It is very intuitive tool and it has a free version as well.

If you are fortunate and have access to Matlab,I would recommend the MSD-Toolbox for multi-stage decimation filter design and optimization. In addition

| Tool | Usage |
|---|---|
| Yosys | Front-end synthesis |
| Qflow | Back-end and STA |
| Verilator | Linting |
| GTKwave | Waveform viewer |
| Octave | Modeling |
| Latex | Documentation |
| GNU Radio | System design and integration |
| TexStudio | LaTeXword processor |
| draw.io | Drawing shapes and figures |

Table 1.3: Open source tool chain.

to the CVX convex optimization toolbox. And last but not least the GLPK the open source linear programming solver.

If you are more into system design and exploration I would recommend the GNU Radio the open source software defined radio (tool and hardware). Moreover, the CppSim open source simulator for analog and digital components.

I personally have the following development boards. The ADALM-Pluto from Analog Device. The Spartan-3E starter kit from Digilent. I would definitely recommend them, price wise and quality wise, they are quite reliable.

| Tool | Usage |
|---|---|
| ModelSim SE (Intel) | RTL front-end simulation |
| Vivado Webpack (Xilinx) | RTL front-end synthesis |

Table 1.4: Commercial free tool chain.

## 1.3 Modules

I have ambitious vision to the DRL, very ambitious may be, but I am looking forward to achieve as much as possible with the limited resources I have at the moment. The DRL offers and supports the current DSP modules, as shown in 1.5. In addition to these modules, I am planning to have the IIR and DC blocker filters, and recursive sinusoidal generator.

| Module | Description | Status |
|---|---|---|
| | *Filters* | |
| FIR | This implementation supports both TF and DF typologies for both symmetrical and asymmetrical coefficients sets. | SVI |
| MAC | This module offers a DF FIR topology based on a single MAC unit for both symmetrical and asymmetrical coefficients sets with parameterizable filter length, coefficient quantization bit-width and IO bit-width. | SVI |
| CICD | This is a CIC decimation filter with parameterizable decimation factor, order, differential delay length, decimation phase, and IO bit-width. | SVI |
| CICP | This is a CIC interpolation filter with parameterizable interpolation factor, order, differential delay length, interpolation phase offset, and IO bit-width. | SVI |
| PPD | This is a polyphase decimation filter supports both TF and DF filter typologies, in addition to counter-clock-wise and clock wise orientation for both filter coefficient set and commutator. In addition to parameterizable filter length, coefficient quantization bit-width, decimation phase, and IO bit-width. | SVI |
| PPI | This is a polyphase interpolation filter supports both TF and DF filter typologies, in addition to counter-clock-wise and clock wise orientation for both filter coefficient set and commutator.In addition to parameterizable filter length, coefficient quantization bit-width, interpolation phase offset, and IO bit-width. | SVI |
| | *Signal Generators* | |
| NCO | This is a ROM based sinusoidal signal generator based on one-eighth (1/8) sinusoidal symmetry. The module generates complete sine and cosine signals based on 1/8 sine and 1/8 cosine stored signals only. | SVI |
| CORDIC | This is a ROM less sinusoidal signal generator based on binary search 'like' algorithm. | SVI |

Table 1.5: DRL supported modules and their development status.

# Chapter 2

# DSP Modules

## 2.1 Digital Filters

This section summarizes the implemented modules, their interfaces, the desired hardware (HW), and detailed design example "for just some selected modules".

The following design parameters is used within the document, as shown in 2.1.

### 2.1.1 Finite Impulse Response - FIR

The digital FIR filter itself can be designed by Octave/Matlab, ASN Filter Designer, GPLK, or any other aid. The outcome of the design step shall be the filter coefficients (*c_filt_coeff*), the filter length[1] (*gp_coeff_length*), and the quantization bit-width (*gp_coeff_width*) for the filter coefficients. As a designer you would have to choose the topology either transposed-form (TF) or direct-form (DF). As a rule of thumb, DF is preferred for smaller input bit-widths, refer to [6] for more details.

[1] ≡ number of filter taps ≡ filter order + 1

| **Parameter** | **Description** | RTL |
|---|---|---|
| $N$ | Filter length | *gp_coeff_length* |
| $M$ | Decimation factor | *gp_decimation_factor* |
| $L$ | Interpolation factor | *gp_interpolation_factor* |
| $D$ | Differntial delay length | *gp_diff_delay* |
| $P$ | Phase offset for decimation or interpolation | *gp_phase* |
| $Q$ | Quantization bit-width | *gp_coeff_width* |
| $W_i$ | Input bit-width | *gp_idata_width* — *gp_inp_width* |
| $W_o$ | Output bit-width | *gp_odata_width* — *gp_oup_width* |
| $C_x$ | Filter coefficient | *c_filt_coeff[x]* |
| $n$ | Phase accumulator bit-width | *gp_phase_accu_width* |
| $k$ | ROM address bit-width | *gp_rom_depth* |
| $m$ | ROM data bit-width | *gp_rom_width* |

Table 2.1: Design parameters.

The following OCTAVE command shall be used to generate the reference data/response for verification of the design.

```
o_data= filter(c_filt_coeff,1,i_data);
```

The desired HW interface is shown in 2.2. Filter configurations for either TF or DF typologies is shown in 2.1 and 2.2, respectively.

| Generic Parameters | |
|---|---|
| *gp_data_width* | Bit-width for signed input data |
| *gp_coeff_length* | Filter coefficient length 'number of filter taps' |
| *gp_coeff_width* | Filter coefficient quantization bit-width |
| *gp_tf_df* | Configure either TF (1) or DF (0) topology |
| *gp_symm* | Configure either symmetric (1) or asymmetric (0) filter coefficient set |
| IO Ports | |
| i_rst_an | Active low asynchronous reset |
| i_ena | Active high synchronous enable |
| i_clk | Rising edge clock |
| i_data | Signed fixed-point data |
| o_data | Signed fixed-point data |

Table 2.2: IO interface for FIR filter - ***filt_fir.v***.

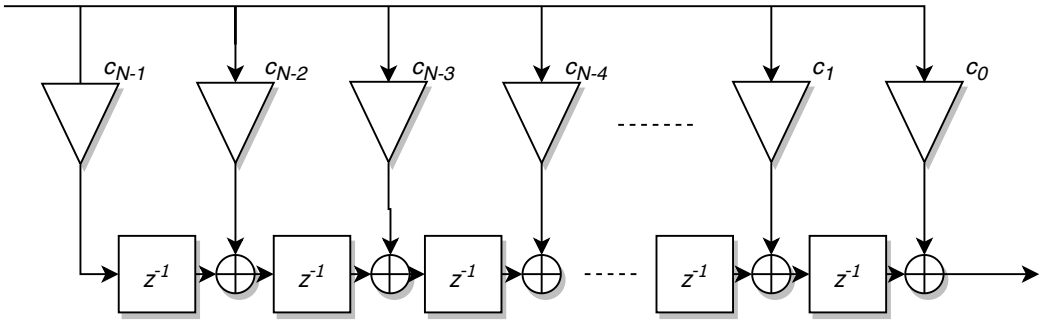


Figure 2.1: FIR implementations using TF topology.

### 2.1.2 Multiply Accumulate - MAC

The MAC is an implementation for FIR DF filters only based on a single MAC[2] unit. The following OCTAVE command shall be used to generate the reference data/response for verification of the design.

```
o_data= filter(c_filt_coeff,1,i_data);
```

The desired HW interface is shown in 2.3.

[2]single multiplier and adder unit together with a feedback flip-flop

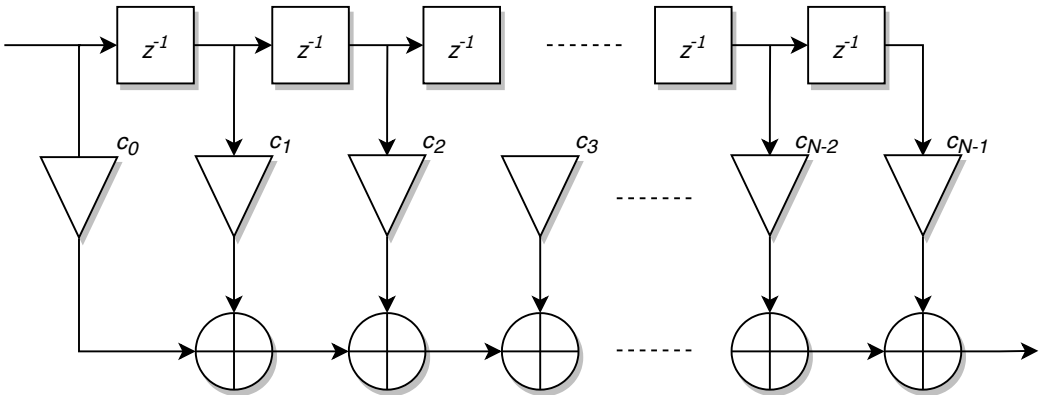


Figure 2.2: FIR implementations using TF topology.

| Generic Parameters | |
|---|---|
| *gp_data_width* | Bit-width for signed input data |
| *gp_coeff_length* | Filter coefficient length 'number of filter taps' |
| *gp_coeff_width* | Filter coefficient quantization bit-width |
| *gp_symm* | Configure either symmetric (1) or asymmetric (0) filter coefficient set |
| IO Ports | |
| i_rst_an | Active low asynchronous reset |
| i_ena | Active high synchronous enable |
| i_clk | Rising edge clock |
| i_data | Signed fixed-point data |
| o_data | Signed fixed-point data |

Table 2.3: IO interface for FIR MAC filter - ***filt_mac.v***.

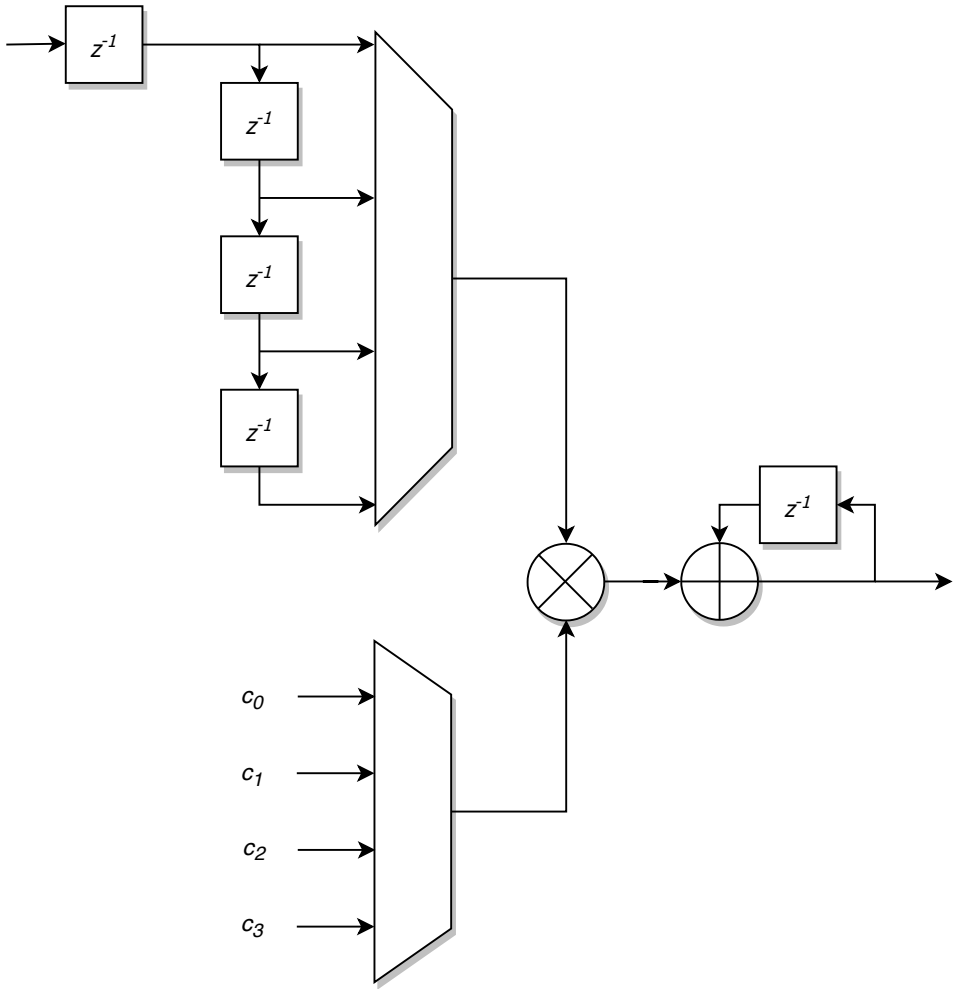


Figure 2.3: FIR filter based on MAC unit for asymmetric filter.

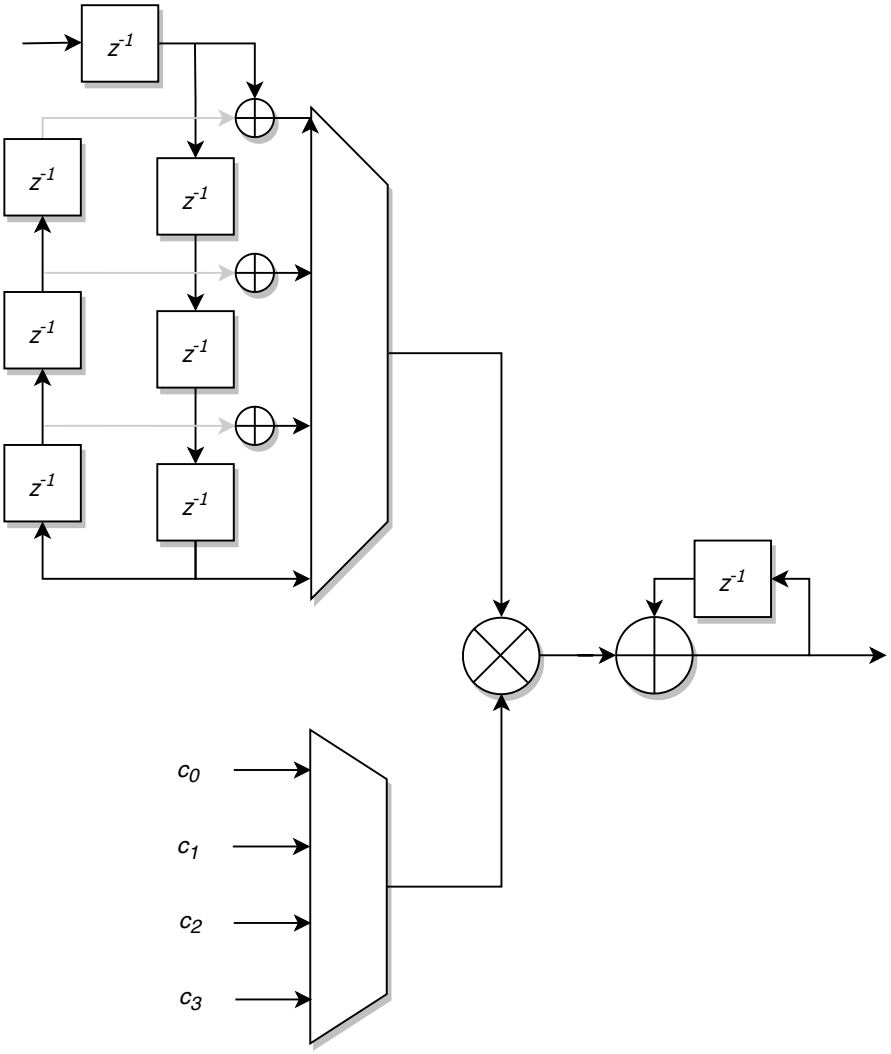


Figure 2.4: FIR filter based on MAC unit for symmetric filter.

### 2.1.3 PolyPhase Decimation - PPD

The following OCTAVE command shall be used to generate the reference data/response for verification of the design.

```
o_data = downsample( filter(c_filt_coeff, 1, i_data), gp_decimation_factor,
gp_decimation_factor-1-gp_comm_phase);
```

| Generic Parameters | |
|---|---|
| *gp_idata_width* | Bit-width for signed input data |
| *gp_decimation_factor* | Decimation factor for downsampler |
| *gp_coeff_length* | Filter coefficient length 'number of filter taps' |
| *gp_coeff_width* | Filter coefficient quantization bit-width |
| *gp_tf_df* | Configure either TF (1) or DF (0) topology |
| *gp_comm_reg_oup* | Configure commutator to either have registered output (1) or non-registered output (0) |
| *gp_comm_ccw* | Configure commutator to either distribute data in CCW (1) or CW (0) directions |
| *gp_comm_phase* | Configure commutator phase offset |
| *gp_odata_width* | Bit-width for signed output data |
| IO Ports | |
| i_rst_an | Active low asynchronous reset |
| i_ena | Active high synchronous enable |
| i_clk | Rising edge clock |
| i_data | Signed fixed-point data |
| o_data | Signed fixed-point data |
| o_sclk | $f_s/M$ |

Table 2.4: IO interface for PPD filter - ***filt_ppd.v***.

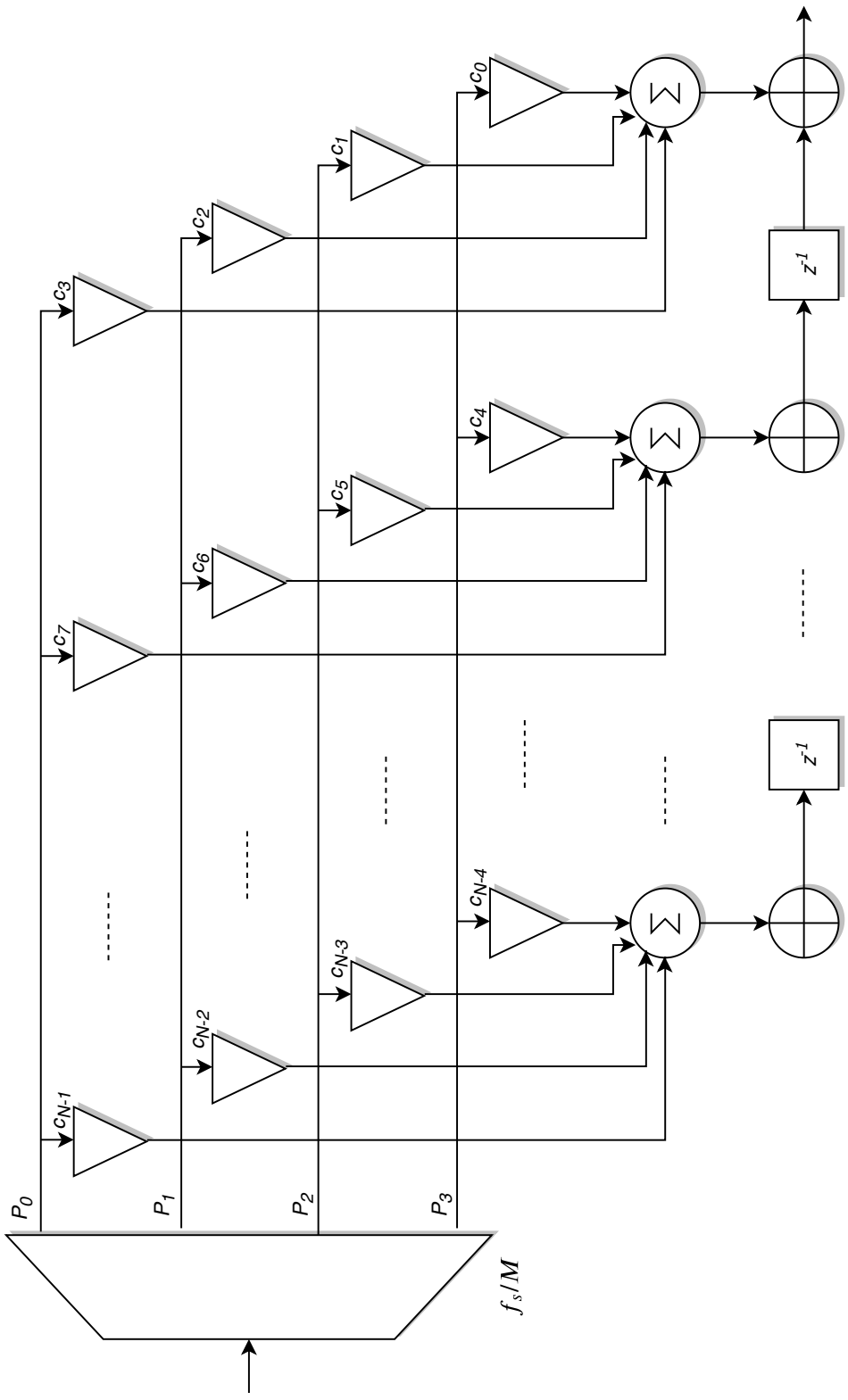


Figure 2.5: Polyphase decimation filter based on TF topology, with CCW multiplier, and CW commutator configuration.

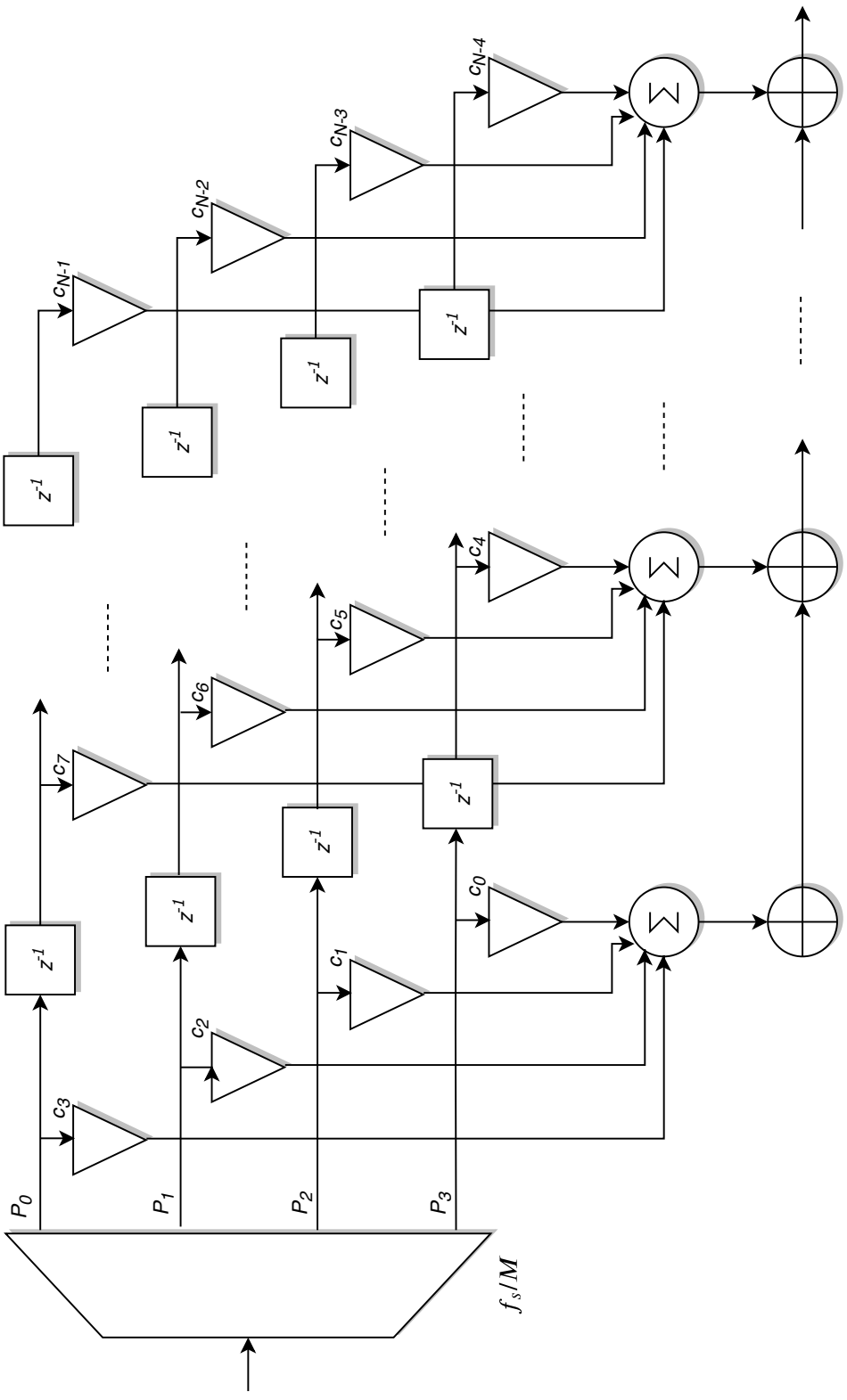


Figure 2.6: Polyphase decimation filter based on DF topology, with CCW multiplier, and CW commutator configuration.

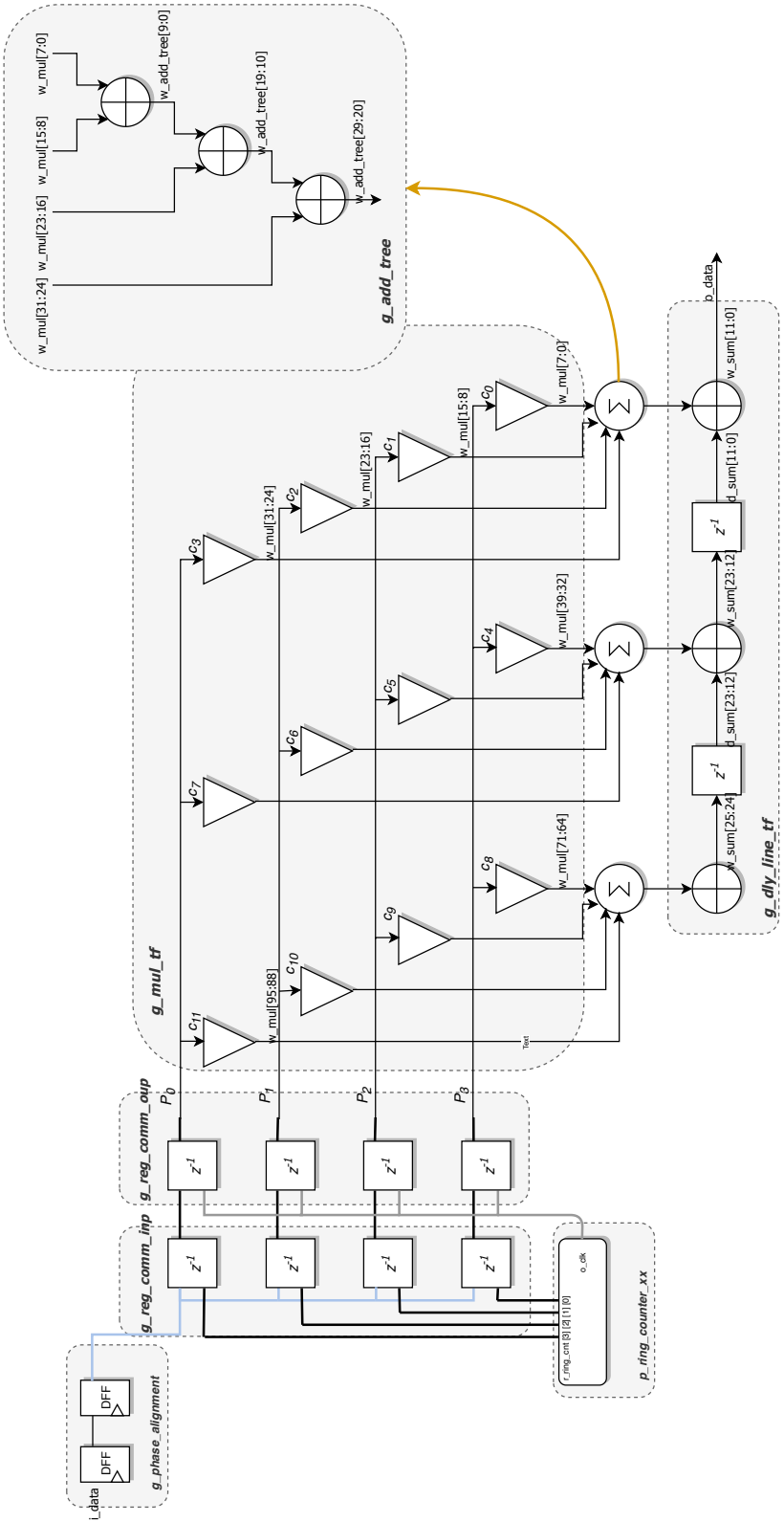


Figure 2.7: PPD with $M = 4$, $N = 12$, $W_i = 2$, $Q = 6$, in TF topology, with CCW commutator, CW multiplier matrix orientation, and commutator registered output.

### 2.1.4 PolyPhase Interpolation - PPI

The following OCTAVE command shall be used to generate the reference data/response for verification of the design.

```
o_data= filter( c_filt_coeff, 1, upsample(i_data, gp_interpolation_factor,
gp_comm_phase) ) ;
```

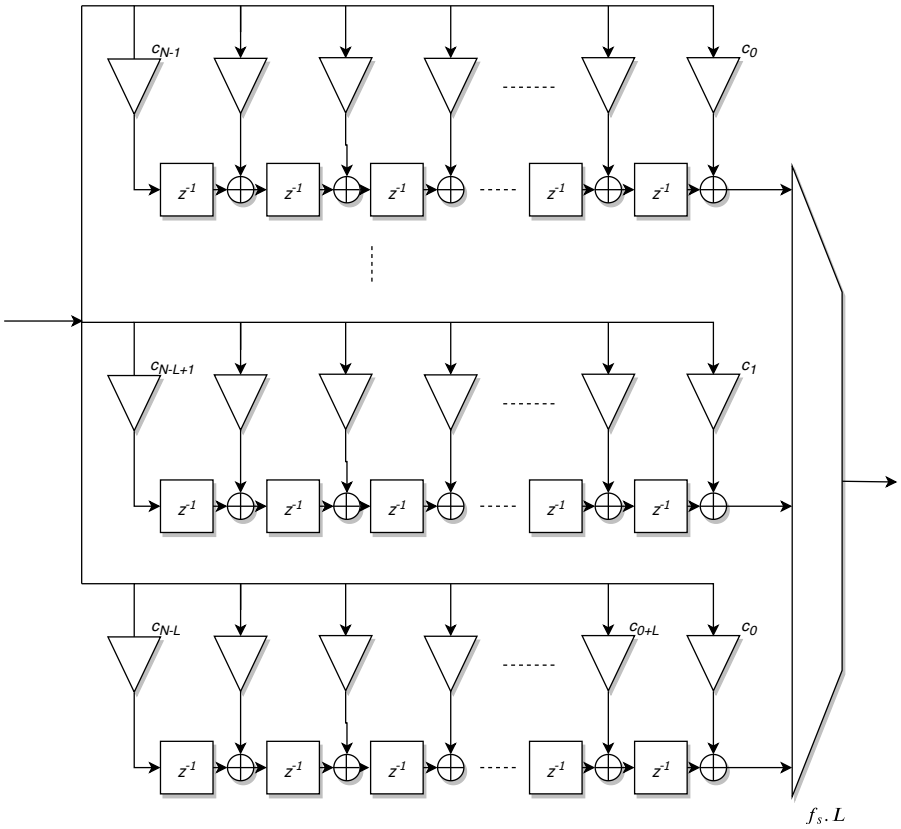


Figure 2.8: Polyphase interpolation filter in TF topology.

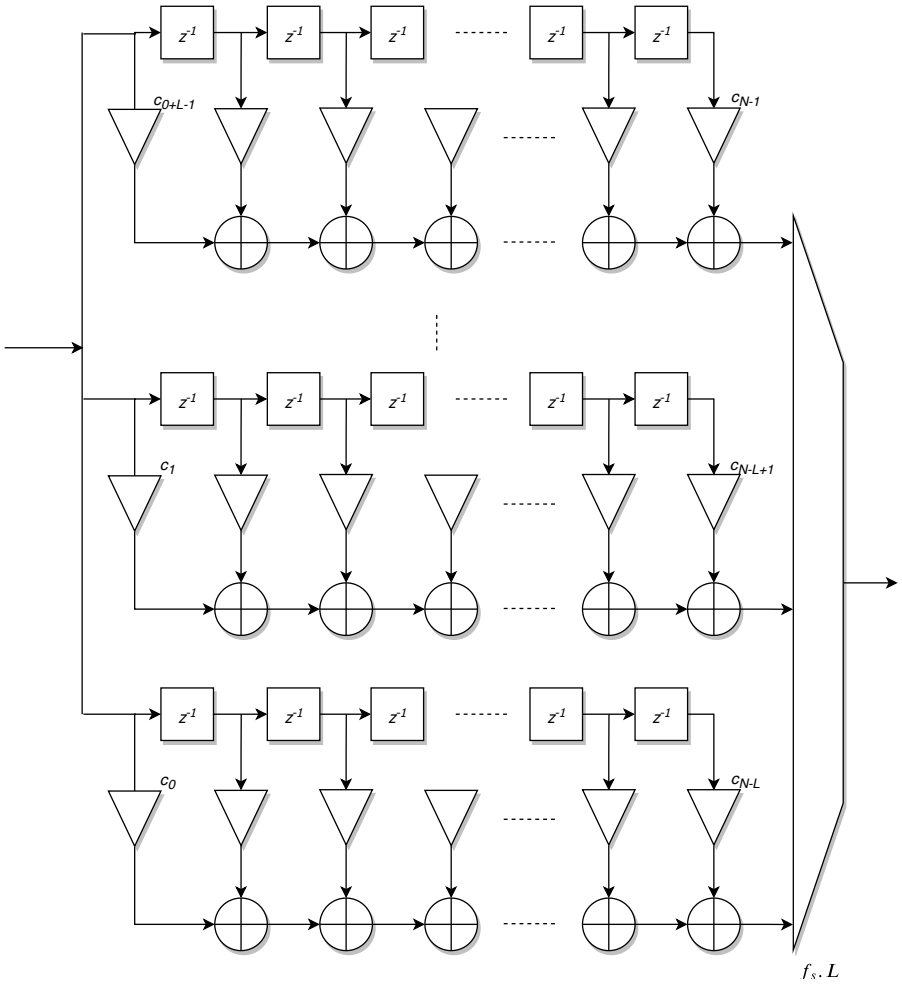


Figure 2.9: Polyphase interpolation filter in DF topology.

| Generic Parameters | |
|---|---|
| *gp_idata_width* | Bit-width for signed input data |
| *gp_interpolation_factor* | Decimation factor for downsampler |
| *gp_coeff_length* | Filter coefficient length 'number of filter taps' |
| *gp_coeff_width* | Filter coefficient quantization bit-width |
| *gp_tf_df* | Configure either TF (1) or DF (0) topology |
| *gp_comm_reg_oup* | Configure commutator to either have registered output (1) or non-registered output (0) |
| *gp_comm_ccw* | Configure commutator to either distribute data in CCW (1) or CW (0) directions |
| *gp_mul_ccw* | Configure commutator to either distribute data in CCW (1) or CW (0) directions |
| *gp_comm_phase* | Configure commutator phase offset |
| *gp_odata_width* | Bit-width for signed output data |
| IO Ports | |
| i_rst_an | Active low asynchronous reset |
| i_ena | Active high synchronous enable |
| i_clk | Rising edge clock |
| i_fclk | Rising edge clock $f_s.L$ |
| i_data | Signed fixed-point data |
| o_data | Signed fixed-point data |
| o_sclk | Signed fixed-point data |

Table 2.5: IO interface for PPI filter - ***filt_ppi.v***.

### 2.1.5 Cascaded Integrator Comb Decimation - CICD

The following OCTAVE command shall be used to generate the reference data/response for verification of the design.

```
[o_data Hcic] = CICFilter(gp_diff_delay, gp_order, gp_decimation_factor,
gp_phase, 1, i_data, 'd');
```

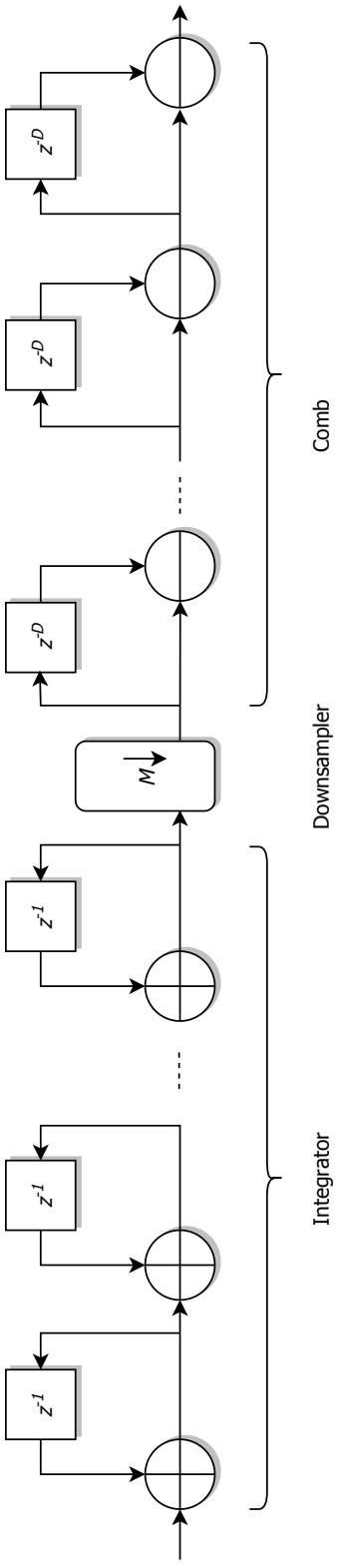


Figure 2.10: CIC decimation filter.

| Generic Parameters | |
|---|---|
| *gp_decimation_factor* | Decimation factor, positive integer larger than one |
| *gp_order* | CIC filter order or number of stages, positive integer larger than one |
| *gp_diff_delay* | Comb delay-line length, positive integer larger than zero |
| *gp_phase* | Downsampler phase, positive integer larger than zero |
| *gp_inp_width* | Input data-width, positive integer |
| *gp_oup_width* | Output data-widt, positive integer |
| IO Ports | |
| i_rst_an | Active low asynchronous reset |
| i_ena | Active high synchronous enable |
| i_clk | Rising edge clock |
| i_data | Signed fixed-point data |
| o_data | Signed fixed-point data |

Table 2.6: IO interface for CICD filter - ***filt_cicd.v***.

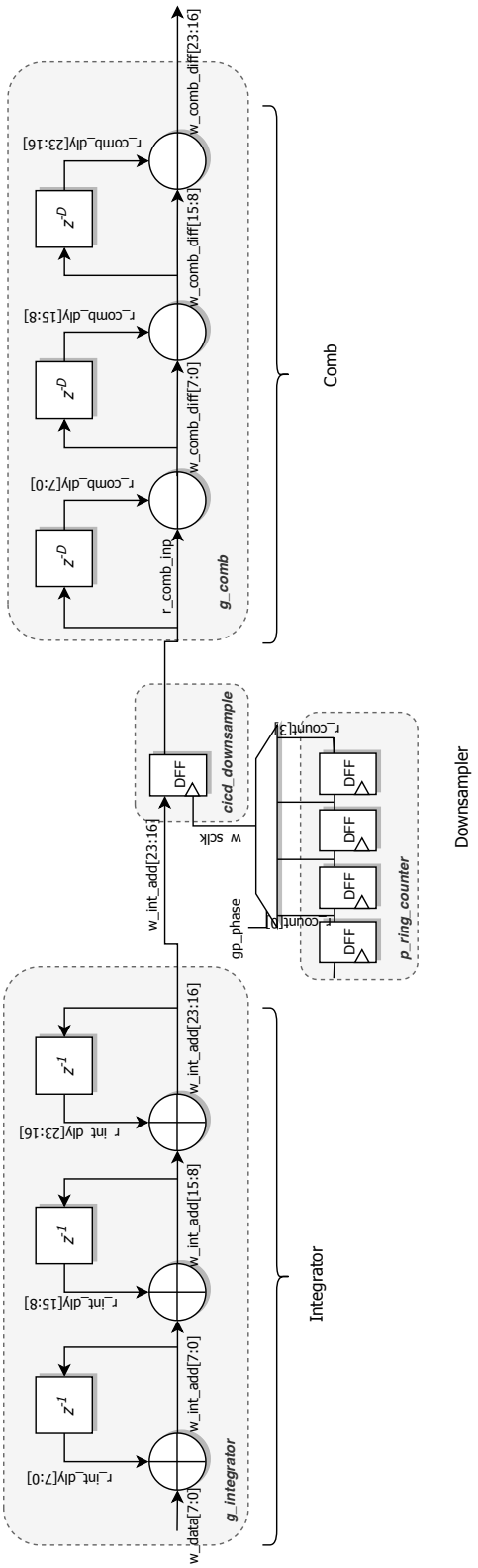


Figure 2.11: CIC decimation filter with $N = 3$, $D = 1$, $M = 4$, $P = 3$, and $W_i = 2$.

### 2.1.6 Cascaded Integrator Comb Interpolation - CICI

The following OCTAVE command shall be used to generate the reference data/response for verification of the design.

```
[o_data Hcic] = CICFilter(gp_diff_delay, gp_order, gp_interpolation_factor,
gp_phase, 1, i_data, 'i');
```

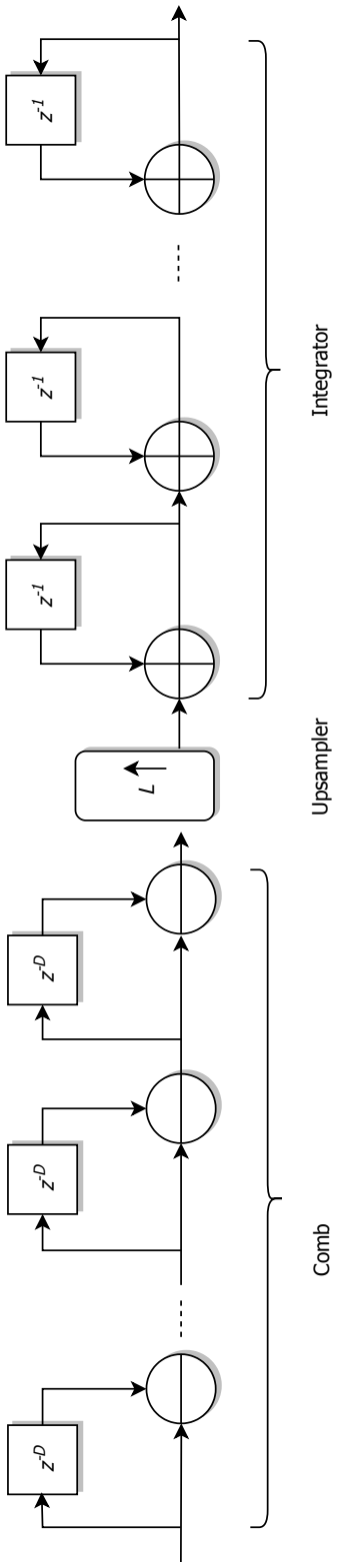


Figure 2.12:

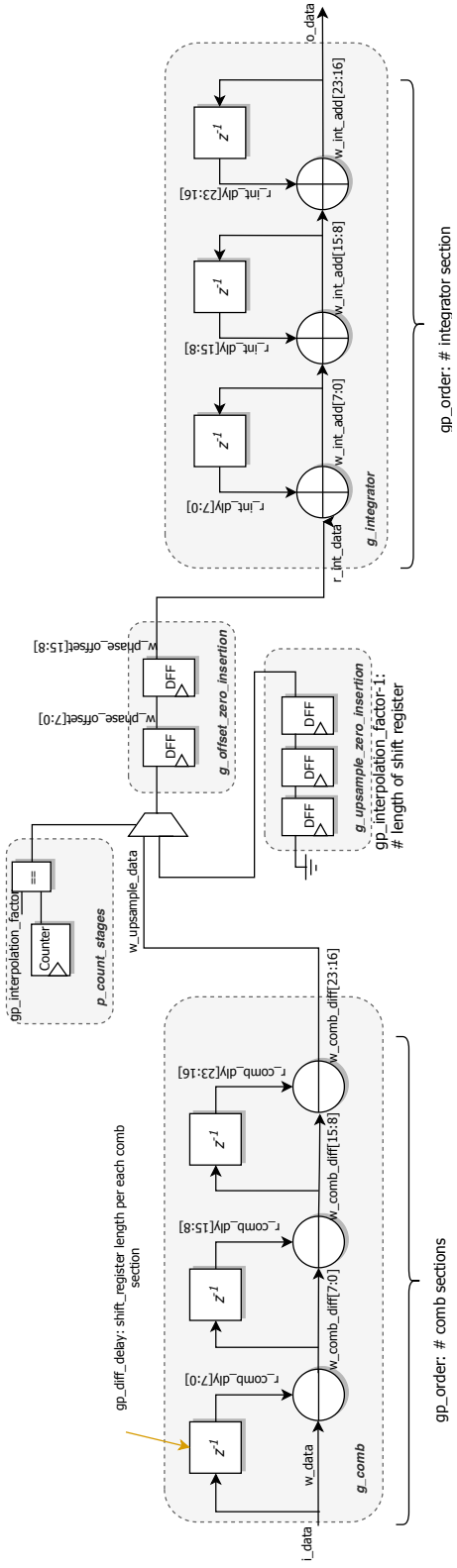
i_data
o_data
g_comb
w_data
z-1
r_comb_div[7:0]
w_comb_diff[7:0]
r_comb_div[15:8]
w_comb_diff[15:8]
r_comb_div[23:16]
w_comb_diff[23:16]
gp_diff_delay: shift_register length per each comb section
gp_order: # comb sections
p_count_stages
gp_interpolation_factor
Counter
=
w_upsample_data
g_upsample_zero_insertion
DFF
gp_interpolation_factor-1:
# length of shift register
g_ofset_zero_insertion
w_phase_offset[7:0]
w_phase_offset[15:8]
g_integrator
r_int_data
r_int_div[7:0]
w_int_add[7:0]
r_int_div[15:8]
w_int_add[15:8]
r_int_div[23:16]
w_int_add[23:16]
gp_order: # integrator section

Figure 2.13:

## 2.2 Signal Generators

### 2.2.1 Numerical Controlled Oscillator - NCO

The NCO is a ROM-based signal generator. The implementation of the proposed NCO is based on sinusoidal one-eighth symmetry which is area efficient.

As a design tip, in order to improve the SFDR of the NCO, it is recommended to add a dithering circuitry for the LSBs of the phase accumulator.

The OCTAVE model has been reused from DSPrelated which has been developed by Kadhiem Ayob. Refer to OCTAVE folder for further details.

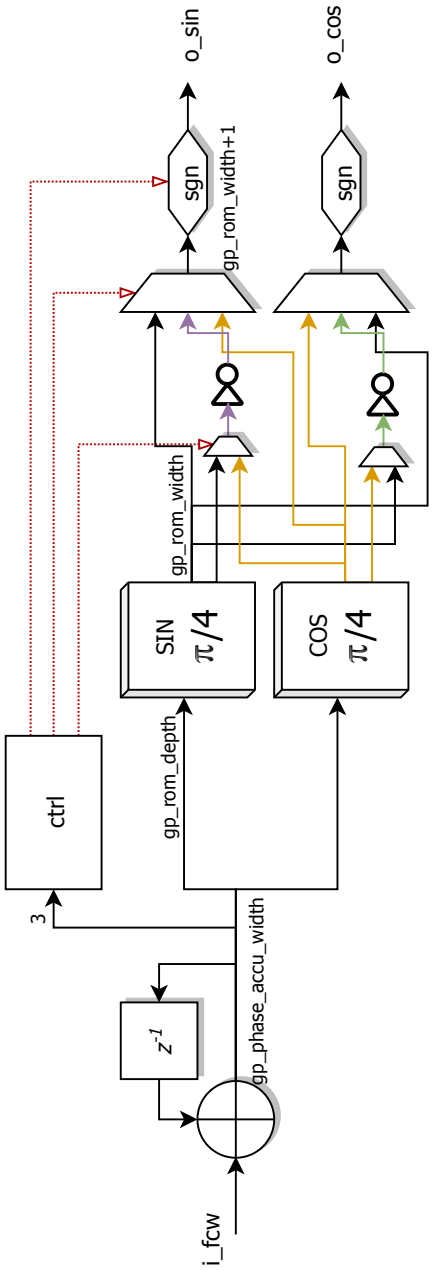


Figure 2.14: NCO implementation based on one-eighth symmetry.

### 2.2.2 COordinate Rotation DIgital Computer - CORDIC

The designed CORDIC supports various implementation and functional choices. The CORDIC supports both unrolled and iterative implementations. The unrolled is a pure combinatorial implementation which is area consuming but

offers the lowest latency. The iterative implementation is successive iterative implementation which offers smaller area but latency overhead. The latency is directly correlated and equivalent to the number of iteration. The iterative implementation employs resource sharing as well for area efficient implementation.

Moreover, the CORDIC can operate in either rotation or vectoring mode. The *atan* LUT is generated based on an Octave model.

It shall be noted that, the gain compensation module is developed and attached within the same RTL file. However, it is not used since it shall be calibrated by the designer as intended.

| Generic Parameters | |
|---|---|
| *gp_mode_rot_vec* | Configure CORDIC operational mode to either rotation (1) or vectoring (0) |
| *gp_impl_unrolled_iterative* | Configure CORDIC implementation to either unrolled (1) or iterative (0) |
| *gp_nr_iter* | Configure the CORDIC number of internal iterations, must be positive integer larger than 1 |
| *gp_angle_width* | ATAN LUT quantization bit-width |
| *gp_angle_depth* | ATAN LUT address/pointer bit-width, imply the LUT depth |
| *gp_xy_width* | IO bit-width for x and y |
| *gp_z_width* | IO bit-width for z |
| IO Ports | |
| i_rst_an | Active low asynchronous reset |
| i_ena | Active high synchronous enable |
| i_clk | Rising edge clock |
| i_x | Signed fixed-point data |
| i_y | Signed fixed-point data |
| i_z | Signed fixed-point data |
| o_x | Signed fixed-point data |
| o_y | Signed fixed-point data |
| o_z | Signed fixed-point data |
| o_done | Signed fixed-point data |

Table 2.7: IO interface for CORDIC signal generator - ***sgen_cordic.v***.

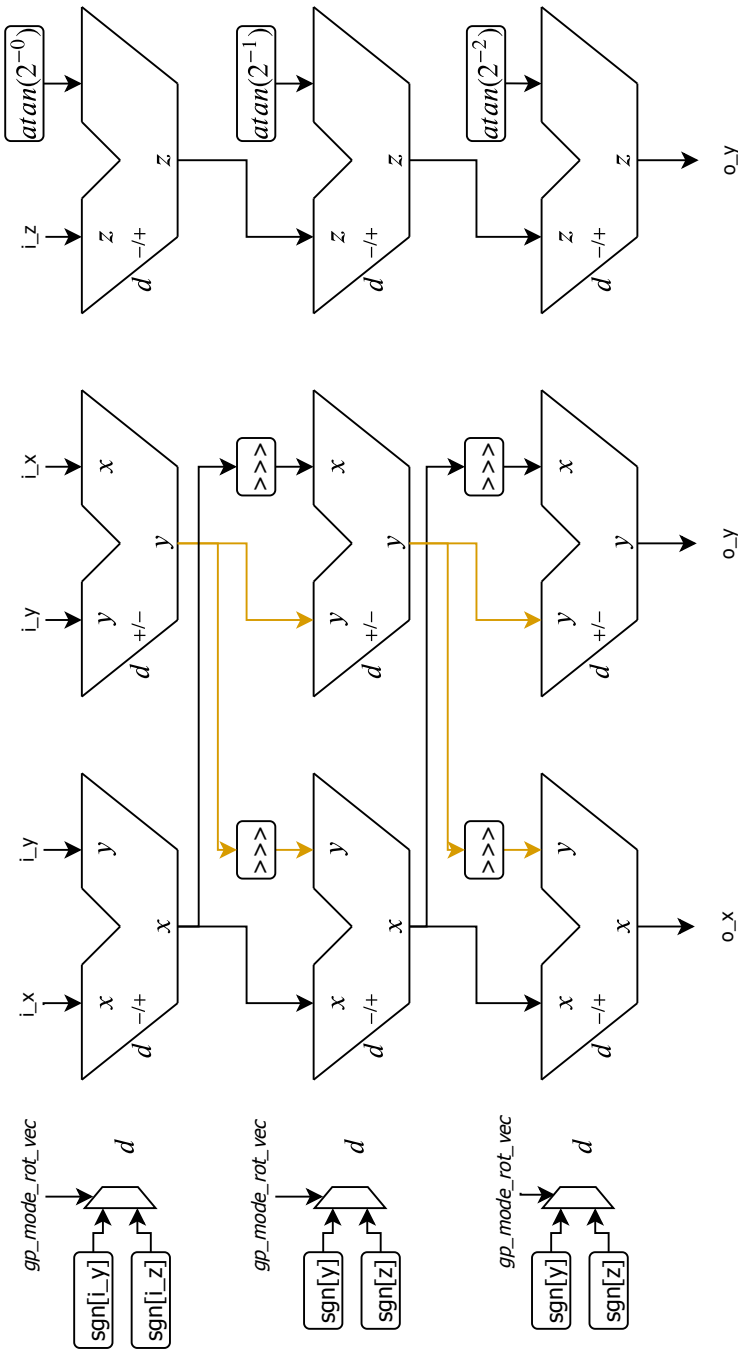


Figure 2.15: CORDIC unrolled implementation.

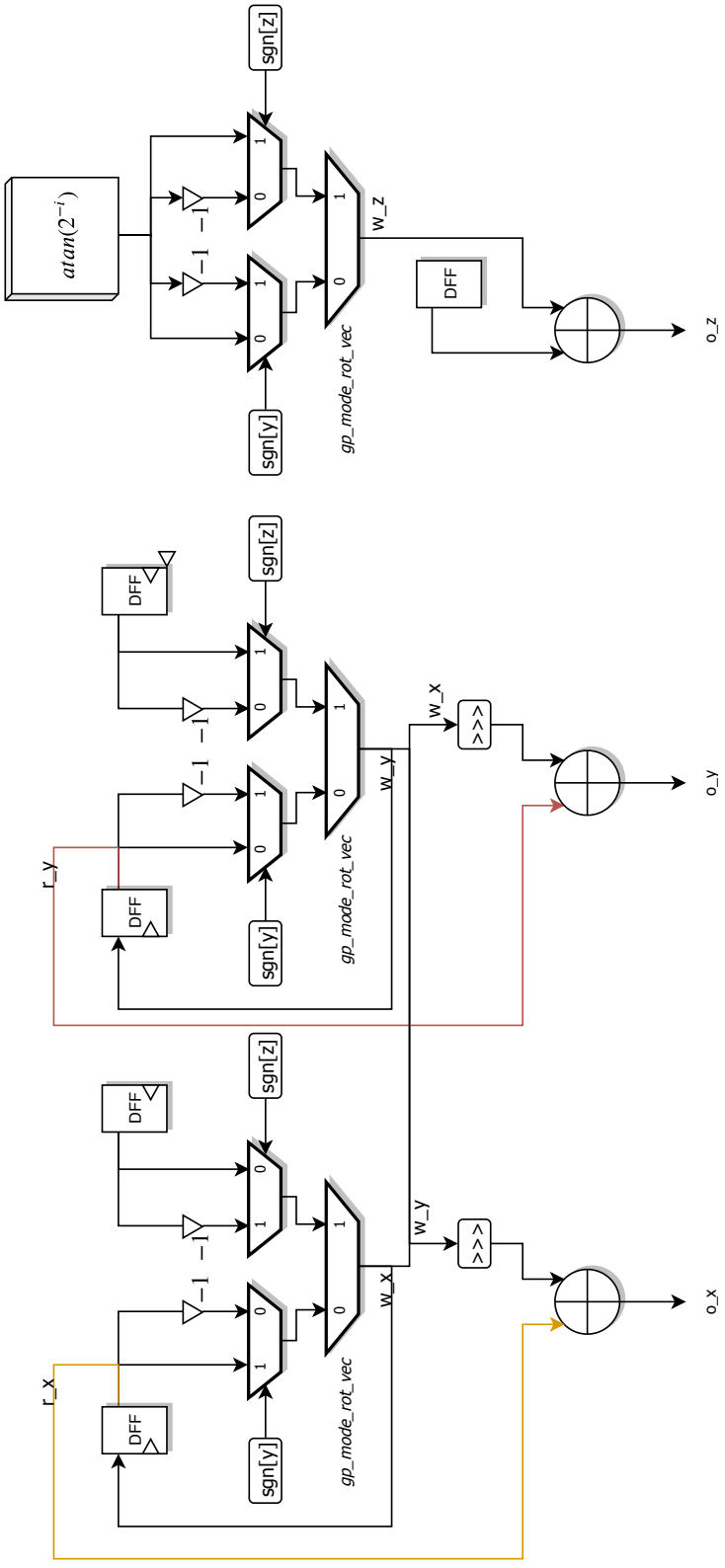


Figure 2.16: CORDIC iterative implementation employing resource sharing.

# Chapter 3

# Verification & Validation

This chapter is to present the used verification methodology. Moreover, the validation setup using evaluation boards.

## 3.1 Verification

The verification has been carried out in a simple straight forward way. Simply a test-bench invoking the targeted module (DUT). The test-bench loads the stimuli and response data from ASCII files which have been generated from OCTAVE using the golden reference models from OCTAVE as well.

It shall be noted that, test-case 4 for the NCO is failing but it is waived. This is due to the fact that the Octave model resets the internal counter whenever a new frequency control word (FCW) is received. However, the RTL is continuing with the new FCW in harmony. My personal opinion that the RTL is more desirable. Nevertheless, if needed other update feel free to ask.

The CORDIC automated verification is waived as well.

## 3.2 Validation

Place holder planned for later!

# Bibliography


[1] R. G. Lyons, *Understanding Digital Signal Processing.* Pearson, 3rd ed., 2012.

[2] A. Shahein, *Power Optimization Methodologies for digital FIR decimation filters.* PhD thesis, Freiburg University, 2014.

[3] A. Shahein, "Design and implementation in vhdl of a decimator for band-bass sigma-delta converters," Master's thesis, Ain Shams University, 2007.

[4] A. Shahein, Q. Zhang, N. Lotze, and Y. Manoli, "A novel hybrid monotonic local search algorithm for fir filter coefficients optimization," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, pp. 616–627, March 2012.

[5] A. Shahein, M. Afifi, M. Becker, N. Lotze, and Y. Manoli, "A power-efficient tunable narrow-band digital front end for bandpass sigma–delta adcs in digital fm receivers," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, pp. 883–887, Nov 2010.

[6] A. Shahein, M. Becker, N. Lotze, and Y. Manoli, "Power aware combination of transposed-form and direct-form fir polyphase decimators for sigma-delta adcs," in *2009 52nd IEEE International Midwest Symposium on Circuits and Systems*, pp. 607–610, Aug 2009.

[7] A. Shahein, M. Becker, N. Lotze, M. Ortmanns, and Y. Manoli, "Optimized scheme for power-of-two coefficient approximation for low power decimation filters in sigma delta adcs," in *2008 51st Midwest Symposium on Circuits and Systems*, pp. 787–790, Aug 2008.

[8] P. Diniz, S. Netto, and E. D. Silva, *Digital Signal Processing: System Analysis and Design.* New York, NY, USA: Cambridge University Press, 2002.

[9] R. Zimmermann, *Binary Adder Architecture for Cell-Based VLSI and their Synthesis.* PhD thesis, Swiss Fedral Institute of Technology Zurich, 1997.

[10] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems.* New York, NY, USA: Wiley-Interscience, 2006.