Instrument ShAred Artifact for Computing (ISAAC)

Polyphase Decimation Filter Spec

FPGA Specification for firdecim

ISAAC Document Number: D-xxxxxx

Revision: 1.0

Code Rev: 1.0

Date: **May 18, 2009**

FPGA Authors: Kayla Nguyen
Jason Zheng

Custodian: Kayla Nguyen

Summary: This document serves as the documentation for the FPGA design for ISAAC Polyphase Decimation Filter. This document only has the information relating to the FPGA and its internal modules. This does not provide interface information between the FPGA and its surroundings.

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099

Revision History

Revision	Description	Author	Date	Approval
1.0	Initial draft for ISAAC	Kayla Nguyen	May 18, 2009	

Contents

List of Figures

List of Tables

1 Features

Table 1: Core Facts

Core Specifics	
Supported Devices	Xilinx, Actel, ASIC
Provided With Core	
Documentation	Product Specification
Design File Formats	Verilog
Verification	Verilog Test Bench
Generated Files	Verilog Filter Core Verilog Filter Test Bench Verilog MAC Core (if R=1) Verilog Frequency Tuner Core (if chosen) Verilog Filter Wrapper Core (if chosen) Makefile User Constraint File (ucf) Input Data File

2 Theory of Polyphase Decimation Filter

The Polyphase Decimation Filter is a Finite Impulse Response Filter (FIR) which decimates the incoming signal by a factor of M.

2.1 Plain FIR + Decimation

In this design of the Polyphase Decimation Filter, the inputs are filtered through a traditional FIR filter, then every M filtered data are taken as the output. This scheme assumes that there are input buffers, and most of the calculations are thrown away due to decimation. If the multiplier clock (internal clock) to input data rate ratio is R, and there are N taps in the filter, there needs to be ceil(N/R) multipliers.

2.2 Traditional Polyphase Filter

In the traditional polyphase filter design, the inputs are decimated first, then those are filtered through an FIR. This scheme breaks the N-tap filter into M sub-filters, and there are no wasted calculations. If there are M sub-filters, and there are N taps in the filter, there needs to be max$(N/M, M)$ multipliers with at least one multiplier per sub-filter. There are two schemes in this design method shown in Figures 1 and 2.

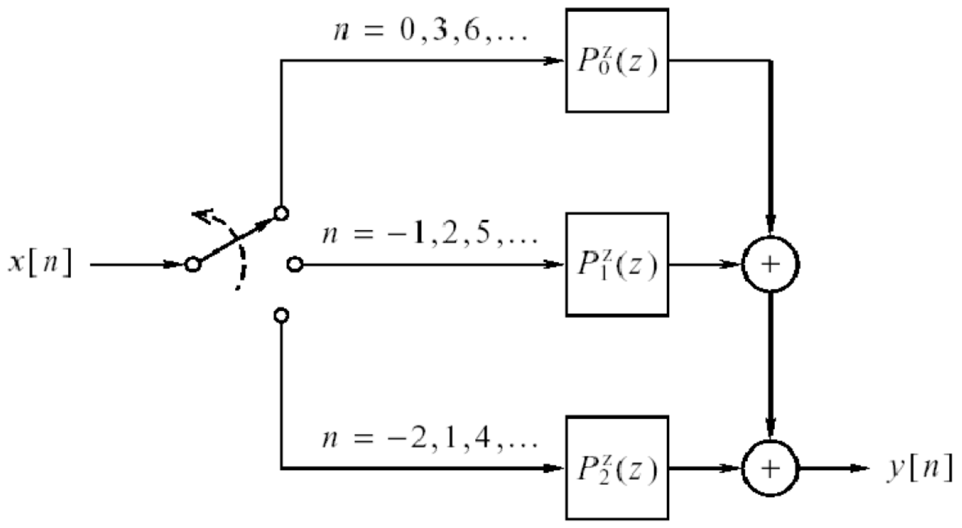

Figure 1: Traditional Switched Input Design

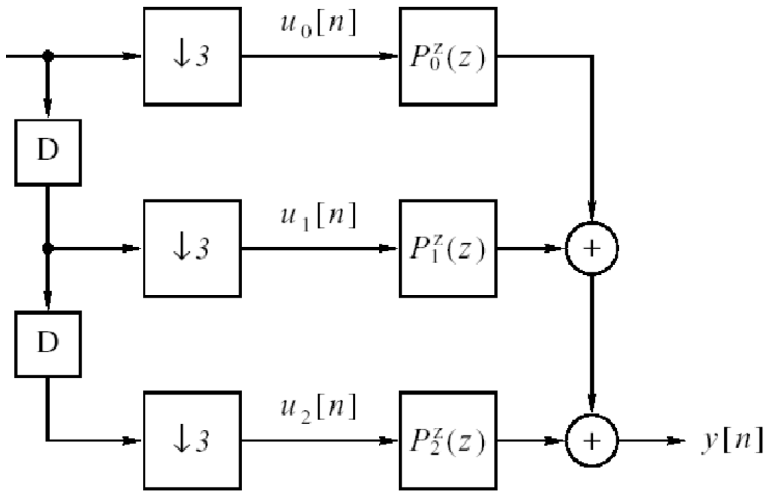

Figure 2: Traditional Delayed Input Design

2.3 Serial FIR + Decimation

The Serial FIR + Decimation design is based on White's design [1]. This scheme calculates N multiplications at once in bit-wise fashion. The latency of the filter depends on the fixed-point width K, in which multipliers are replace by LUTs with 2^N depth, where N is the number of taps. The internal clock rate is K∗InputRate, which limits the input rate if the multiplier is not able to run K times faster. For high input rates, input buffers are needed to temporarily store the incoming inputs while the filter is running previous input data.

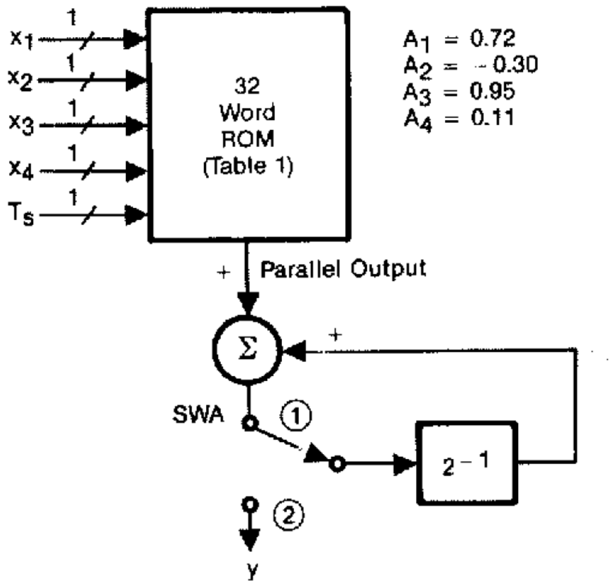

Figure 3: Serial FIR Design

2.4 FIR Decimation Methods Comparison

A comparison table for the three methods described in the above sections is given in Table 2.

Table 2: FIR Decimation Comparison

Method	Resource Required	Pros	Cons
Plain FIR + Decimation	(N/R) Multipliers	Straight forward implementation	Wasted calculations due to decimation
Traditional Polyphase Filter	(N/M) Multipliers	No wasted calculations	Cannot take advantage of high R
Serial FIR + Decimation	2^N Entries LUT	Small, compact FIR	Slow, need buffer for high input rates

3 Theory of ISAAC Polyphase Decimation Filter (firdecim)

The ISAAC approach to the Polyphase Decimation Filter is based on the combination of Traditional Polyphase Filter and the Plain FIR + Decimation filter. By incorporating the best of both designs, the

ISAAC design generalizes multiplier sharing with decimation factor M and clock ratio R. The number of multipliers needed is ceil($N/(M*R)$), which is a combination of the Plain FIR + Decimation (N/R) and the Traditional Polyphase Filter (N/M) number of multipliers. Since R is typically very high in a multi-stage polyphase decimation design, this design suits well for "bottom-heavy" filters with single clock domain, even though early stages of the design can also utilize the ISAAC firdecim core.

The ISAAC firdecim design is divided into three categories of ratio R:

1) R is greater than 1

2) R is equal to 1

3) R is less than 1

where:

$$R = \frac{\text{input data rate}}{\text{internal clock rate } (or \text{ multiplier clock rate})}$$

3.1 Limitations of the firdecim Design

1. For R >1, the decimation factor must be equal or greater than the R, and must be a multiple of R. This implies that there is only one output data stream. If the decimation factor is less than 1, the Verilog design is incorrect.

4 ISAAC firdecim Parameterize Script

The ISAAC firdecim Verilog RTL code is generated by a Python script. This script is able to take in a set of options in order to parameterize the RTL code to fit a specific need. The options are listed in Table 3.

Table 3: ISAAC firdecim Python Options

Command Line Option	Alternate Option	Details	Section Reference
-f	--datafreq	Input data frequency (MHz) default=60MHz	4.1.1
-l	--clockfreq	Internal clock/multiplier frequency (MHz), default=60MHz	4.1.2
-d	--decimation	Decimation factor, default=5	4.1.3
-t	--taps	Number of taps, default=15	4.1.4
-i	--inwidth	Data input bit width, default=16	4.1.5
-c	--coefwidth	Coefficient bit width, default=16	4.1.6
-o	--outwidth	Output bit width, default=16	4.1.7
-a	--accumwidth	Accumulator/multiplier bit width, default=32	4.1.8
-v	--vfile	Output Verilog file name, default=firdecim.v	4.1.9
-r	--coeffile	Coefficient file which contains the coefficients, default=coef15.txt	4.1.10
-n	--infile	Text file name for the test input data storage, default=in.txt	4.1.11
-m	--outfile	Test file name for the test output data storage after filtering, default=out.txt	4.1.12
-u	--tune	Should the firdecim module be tunable? Yes or no, default=no	4.1.13
-e	--inctuner	Should be frequency tuner be generated? Yes or no, default=no	4.1.14
-s	--tunestep	Stepping frequency for the frequency tuner in MHz, default=1	4.1.15
-p	--tunesampling	Sampling frequency for the frequency tuner in MHz, default=240	4.1.16
-x	--tunecenter	Center frequency of input data to use for testing in MHz, default=65	4.1.17
-h	--help	Show help message	None
	--version	Show program's version	None

4.1 Explanation of Options

4.1.1 Option: Input Data Frequency

The option -f or --datafreq is the frequency of the raw data coming into the firdecim. This number is read in as a floating number.

If the data rate is reduced by a Mux to parallel streams, this option must be data rate before the Mux. For example, if the raw data rate is 240MHz and is Muxed down to 4 streams of 60MHz data, then option -f must be 240.

Note that this number must be a multiple (or multiplicative inverse) of the internal clock frequency. For example:

1) "-f 240 -l 60" is acceptable, but "-f 200 -l 60" is NOT acceptable

2) "-f 2.4 -l 60" is acceptable, but "-f 2.6 -l 60" is NOT acceptable

In other words: $\frac{\text{input data frequency}}{\text{clock frequency}}$ or $\frac{\text{clock frequency}}{\text{input data frequency}}$ must be a whole number.

4.1.2 Option: Internal Clock/Multiplier Frequency

The user is able to choose any internal clock frequency (option -l or --clockfreq) they choose. This number is read in as a floating number. Keep in mind that this clock frequency is physically limited to the device in which this design is implemented, the placing of the module, the routing of the signal lines, and any other modules that are placed on the same device.

4.1.3 Option: Decimation Factor

The decimation factor (-d or --decimation) determines the ratio between the output data frequency and the input data frequency. If the input data frequency is 60MHz, and the decimation factor is 5, then the output data frequency is 2.4MHz. This also determines the amount of output data vs. the amount of input data. If the amount of input data is 10,000 points, then the amount of output data is 2,000 points. This number is read in as an integer, so an input of 2.5 is not acceptable.

4.1.4 Option: Number of Taps

The number of taps (-t or --taps) is determined by the design of the filter. This number is read in as an integer, so an input of 10.5 is not acceptable.

There are certain rules that this number must follows:

1) This number "should" be a multiple of the decimation factor. If the actual number of taps is not a multiple of the decimation factor, the user should add zeros at the end of the coefficients list and choose the next closest number of taps that is a multiple of the decimation factor.

4.1.5 Option: Data Input Bit Width

The user is able to choose any arbitrary data input bit width (option -i or --inwidth). This number is read in as an integer, so an input of 16.5 is not acceptable. Keep in mind that the module assumes that the most significant 2 bits are integer bits (2^1 and 2^0) and the rest of the bits are the fractions. This is significant in the truncation of the bit width in the case that the output bit width is less than the accumulator/multiplier bit width. The module also assumes that the input data is signed data. This means that if the most significant bit is a 0, the data is positive data. If the most significant bit is a 1, the data is negative data.

4.1.6 Option: Coefficient Bit Width

The user is able to choose any arbitrary coefficient bit width (option -c or --coefwidth). This number is read in as an integer, so something like "16.5" is not acceptable. Keep in mind that the module assumes that the most significant 2 bits are integer bits (2^1 and 2^0) and the rest of the bits are the fractions. This is significant in the truncation of the bit width in the case that the output bit width is less than the accumulator/multiplier bit width. The module also assumes that the input is signed data. This means that if the most significant bit is a 0, the coefficient is positive. If the most significant bit is a 1, the coefficient is negative.

4.1.7 Option: Data Output Bit Width

The user is limited to choosing the output bit width that is less than or equal to the accumulator/multiplier bit width. This number is read in as an integer, so something like "16.5" is not acceptable. If the output bit width is less than the accumulator/multiplier bit width, the output data is truncated starting with the 3rd bit from the most significant bits. This is because the module assumes that both the input data and the coefficients has 2 bits of integer.

For example,

1	1	1	1	1	0	1	0	0	1	1	0	1	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0
Accumulator/multiplier data																															
Spare		Output data																Spare													

Accumulator/multiplier data: 1111_1010_0110_1000_0110_1010_1000_0000
Output data: 1110_1001_1010_0001

4.1.8 Option: Accumulator Bit Width

The user is able to choose any arbitrary accumulator bit width (-a or --accumwidth). This number is read in as an integer, so something like "16.5" is not acceptable. Note that in order to get a good resolution output, the accumulator bit width should be at least the bit width of the input data plus the bit width of the coefficient data.

4.1.9 Option: Verilog Data File Name

The user is able to choose any arbitrary Verilog data file name (-v or --vfile), as long as it ends with ".v".

4.1.10 Option: Coefficient File Name

The coefficient file name (-r or --coeffile) is the name of the file which stores the coefficients. The file must contains the coefficients in BINARY format, with the correct bit widths according to the coefficient bit width (-c or –ntaps), and the file must contain at least the number of coefficients that matches the number of taps (-t or -ntaps).

Another use of the coefficient file is in the Frequency Tuner, see Section 4.1.14.

4.1.11 Option: Input Data File Name

The input data file name (-n or --infile) is the name of the file to be read by the test bench as input data. The Python code generates pseudo data using the cosine function with the bit width given in the Input Bit Width option (-i or --inwidth). If the user has a different set of inputs, replace the file with the data being tested. Note that the data is in Binary, and each point of data is entered in separate lines. Look at the generated data by the Python code to follow the format.

4.1.12 Option: Output Data File Name

The output data file name (-m or --outwidth) is the name of the file that the filtered data is the be stored. The data is stored in Binary format with new data points every line.

4.1.13 Option: Tunability

The tunability option (-u or --tune) takes in a "yes" or "no". If "yes" is chosen, the Python code creates a port for the coefficients to be loaded. If "no" is chosen, the Python code writes the coefficients from the Coefficient File into parameters, one for each coefficient.

4.1.14 Option: Include Frequency Tuner

If Tunability is chosen as "yes", the option for Include Frequency Tuner (-e or --inctuner) is able to be answered "yes" or "no". Note that this option only allows "yes" if Tunability is "yes". If "yes" is chosen, the Python code generates a frequency tuner module, which is able to generate two sets of coefficients based on the following equation:

$$\begin{aligned} \text{coef\_r} &= \text{constant} * \cos\left(\frac{2\pi \cdot \text{fcenter}}{\text{fsampling}}\right) \\ \text{coef\_i} &= \text{constant} * \sin\left(\frac{2\pi \cdot \text{fcenter}}{\text{fsampling}}\right) \end{aligned}$$

where:

1) **coef_r** is the "real" coefficients based on cosine.
2) **coef_i** is the "imaginary" coefficients based on sine.
3) **constant** is a set of constants, one for each calculated coefficient, given in Coefficient File. If the coefficients are meant to be only the sine and cosine, the Coefficient File should contain decimal 1's (but in Binary Form, ex: 01000000 with 8-bit signed).
4) **fcenter** is the center frequency of the input data. This value is given in Input Center Frequency (see Section 4.1.17.
5) **fsampling** is the sampling frequency of the input data. This value is given in Input Sampling Frequency (see Section 4.1.16).

This frequency tuner can to be used to tune 2 firdecim filters if the user chooses to use the tuner that way. Otherwise, the user can choose to use one channel of coefficients only, and the synthesizer should remove the other coefficient calculation block.

4.1.15 Option: Frequency Tuner Stepping Size

This option is only used when the filter has a Tunability of "yes". The frequency tuner stepping size (-s or --tunestep) is the smallest change in frequency that the filter needs to be tuned to. This number is read in as a floating point number. For example, if the filter needs to be tuned to 66MHz, then 67MHz, then 68MHz, the number to input in this option is "1". If this is the case, the user cannot tune the filter to 65.5MHz. If it is needed that the filter needs to be tuned to 65.5MHz, the input to this option is "0.5". The input to this option must be a multiple, or multiplicative inverse, of 1. Example of acceptable inputs: 0.5 (which is 1/2), 10, 0.04 (which is 1/25). Example of NON-acceptable input: π, 1/11 (which really is $0.\overline{09}$). Since the size of the look-up table depends on the inverse of the stepping size, smaller stepping size requires a larger look-up table. The size of the look-up table is constraint by the availability of resource on the device of implementation.

4.1.16 Option: Input Sampling Frequency

This option is only used when the filter has a Tunability of "yes". The input sampling frequency (-p or --tunesampling) is used in the frequency tuner to calculate the coefficients of the filter. This number is read in as a floating point number, so technically any number input is acceptable. Note that in order for the frequency tuner to be accurate, it is recommended that this number is a whole number.

4.1.17 Option: Input Center Frequency

The input center frequency (-x or --tunecenter) is used in the test bench of the RTL code. This is used to test the filter at a certain frequency. Users are allowed to change this frequency inside the test bench after it is generated to test different frequencies. The variable in the test bench is "freq".
Currently, the user is able to only choose frequencies of whole number. Fractional frequencies is not recognized by the core, e.g. 65.5MHz. Future expansion of this capability is possible.

5 Use of Makefile

A Makefile is generated for easy synthesis and simulation in a Linux environment. The default FPGA device specified by the Makefile is the Virtex-II 3000, XQR2V3000-4CG717.

5.1 RTL Simulation using NCVerilog

The Makefile allows user to quickly simulate the design using the pre-generated test bench and data input file. By typing “make test” on the command line, the Makefile calls for NCVerilog simulation tool to simulate the design using the pre-generated test bench. In order to run simulation with NCVerilog, the Linux server must have NCVerilog installed.

5.2 Synthesis using Xilinx XFlow

The Makefile allows user to quickly synthesize the design by typing on the command line “make all”. The file calls for Xilinx XFlow tool to synthesize, place and route, and create a timing module of the design for gate level simulation. The synthesis tool uses the user constraint file generated by the Python script (gatefile.ucf) to constraint the timing of the design. The constraint is the clock speed that the user inputs as one of the options of the Python script. The user is free to change this constraint, and add any other constraints that are necessary, such as pin placement, implementation, or routing constraints. To learn about the user constraint files and the types of constraints available, read the Xilinx Constraints Guide [2].

To synthesize the design using a different FPGA device, simply change the device in Makefile under “dev”. To change any of the XFlow parameters, read the Xilinx XFlow Overview guide [3].

5.3 Cleaning the Synthesis Environment

To clean the synthesis workspace (for easy re-synthesis, or to remove all of the synthesis files), simply type “make clean” in the command prompt. This command removes the following files:

- /work folder
- .log files
- .his files
- .ncd files
- gatefile.v file (this file is used for synthesis, created from the top module Verilog files)

6 Design Timing

6.1 Timing for R =1

The filter design for R =1 is such that the first valid output is $N + 3$ valid clocks (this means that if there are any invalid data where the Input Valid strobe is low, this clock does not count in the number of "valid clocks") after the first input, where N is the number of taps in the filer. There is an initial latency of 3 clocks due to the calculation latency of the filter. After the initial output data, there is a valid output every M clocks, where M is the decimation factor.

The following example is a filter with $N = 10$ taps, and decimation factor of $M = 2$. The timing diagram is shown in Figure 4.

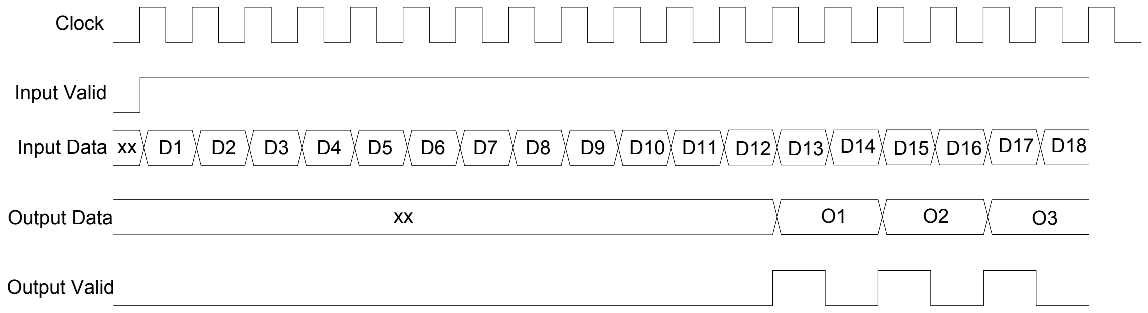

Figure 4: Timing Diagram Example for $R = 1$

From Figure 4, notice that each inputs are valid for one clock, and that another one is valid the next clock. It is not a requirement that the inputs are valid one clock after another, but the output frequency of decimation is only guaranteed when the inputs are valid consistently in time. Also notice that the first output data happens in clock 13, which is $N + 3$. The Output Valid strobe is only valid for the first clock of the output data, this is true even if the output data is valid for 100 clocks. The output data should be latched on the rising edge of the clock, when Output Valid is high.

6.2 Timing for R <1

The filter design for R <1 is such that the first valid output is 5 clocks after the first valid input. This is the latency in the calculation of the multiplications and additions of the filter. This latency is always 5 clocks and is not influenced by the ratio R. After this initial latency, there is a valid output every M/R clocks (where M is the decimation factor and $M/R > M$ since R <1).

The following timing example is a filter with $R = 1/3$, $N = 4$ taps, and decimation factor of $M = 2$. The timing diagram is shown in Figure 5.

From Figure 5, notice that each inputs are valid for three clocks, and the next data is valid the very next clock afterwards. It is not a requirement that the inputs are valid one after another in with perfect frequency, but the output frequency of decimation is only guaranteed when the inputs are valid consistently in time. The output data is valid starting 5 clocks after the first valid input data. Since there are 4 taps, the first two outputs are pseudo-data (O1x and O2x in Figure 5) since the inputs have not had the chance to travel through the whole filter. The first real valid output is O1. For filters with different number of

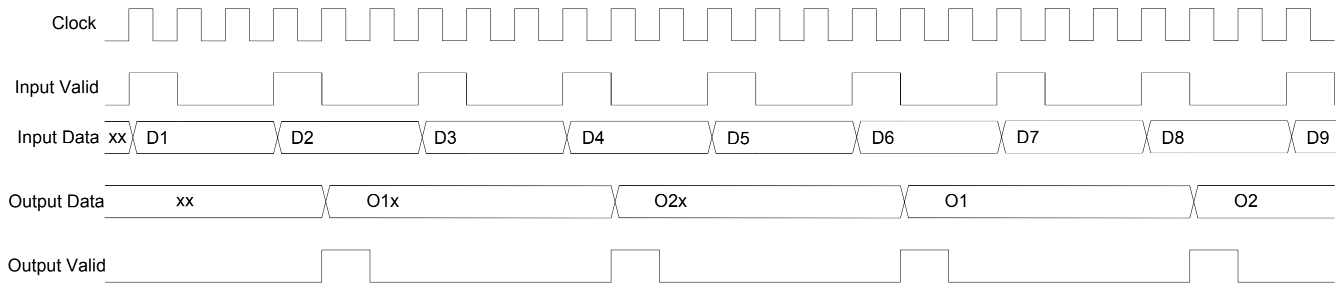

Figure 5: Timing Diagram Example for R <1

taps, the first real output data is output number:

$$\frac{\text{number of taps } (N)}{\text{decimation factor } (M)} + 1$$

The Output Valid strobe is high for exactly one clock at the beginning of the change in data. It is guaranteed that the data is held valid until the next valid data is output, but the data should be latched when the Output Valid strobe is high on the rising edge of the clock.

6.3 Timing for R>1

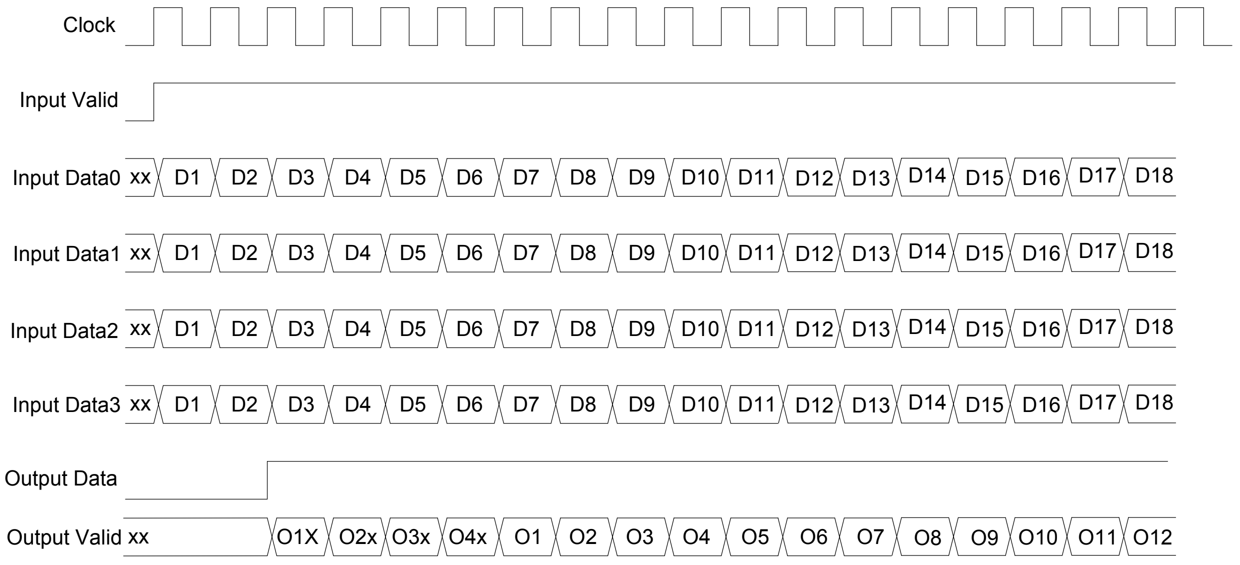

Figure 6: Timing Diagram Example for R >1

The filter design for R >1 is such that the first valid output is 2 clocks after the first valid input. This is the latency in the calculation of multiplications and additions of the filter. This latency is always 2 clocks and is not influenced by the ratio R. After this initial latency, there is a valid output every R/M clocks (where M is the decimation factor and $M/R \geq M$ since $R > 1$ and $M \geq R$).

Figure 6 shows the timing diagram of a filter example with $R = 4$, $N = 16$ taps, and decimation factor of $M = 4$. Notice that there are 4 parallel inputs SigIn0-SigIn3 into the filter, with data valid every clock.

Even though it is not a requirement that data be valid every clock, but the output frequency of decimation is only guaranteed when the inputs are valid consistently in time. The output data is valid starting 2 clocks after the first valid input data. Since there are 16 taps, the first 4 outputs are pseudo-data (O1x-O4x) since the inputs have not had the chance to travel through the whole filter. The first real valid output is O1. For filter with different number of taps, the first real output data is output number:

$$\frac{\text{number of taps } (N)}{\text{decimation factor}(M)} + 1$$

For cases where the output data is not valid every clock, the Output Valid strobe is high for exact one clock at the beginning of the change in data. It is guaranteed that the data is helf valid until the next valid data is output, but the data should be latched when the Output Valid strobe is high on the rising edge of the clock.

6.4 Frequency Tuning Timing

When the option Tunability is “yes” and the Frequency Tuner option is also “yes”, the Python script generates a Frequency Tuner Verilog module along with a wrapper to connect the tuner and the filter. In order to tune the filter, the user needs to input the frequency in MHz into input port “freq”, and assert the “newFreq” strobe. The frequency tuner module starts to calculate the coefficients on the rising edge of the newFreq strobe. As long as another rising edge of newFreq is not seen by the module, the module stops calculating the coefficients as soon as it meets the number of taps in the filter. This implies that signal newFreq can be held high for as long as the user wishes. If the filter is tuned to a new frequency, newFreq should be deasserted then assert again to generate the rising edge.

The Frequency Tuner takes N+2 clocks to generate a complete set of coefficients, where N is the number of taps in the filter. This implies that for at least N+2 clocks after the rising edge of newFreq, the filter is in an unstable condition and no input data should be valid during this time.

Currently, the user is able to choose a stepping size fractional of 1, but the frequency input is not able to recognize this fractional frequency, e.g. 65.5MHz. Therefore, only frequencies of whole numbers may be input at this time. Future expansion of this capability is possible.

7 Examples

7.1 Filter with R=1 and No Frequency Tuner

7.1.1 Design of firdecim_m5_n15

The firdecim_m5_n15 is a filter with 15 taps which filters the input signal and decimates the input by a factor of 5. The core of the firdecim_m5_n15 design is the multiply-accumulate (MAC) unit. The basic idea behind the design for firdecim_m5_n15 is given in figure (7).

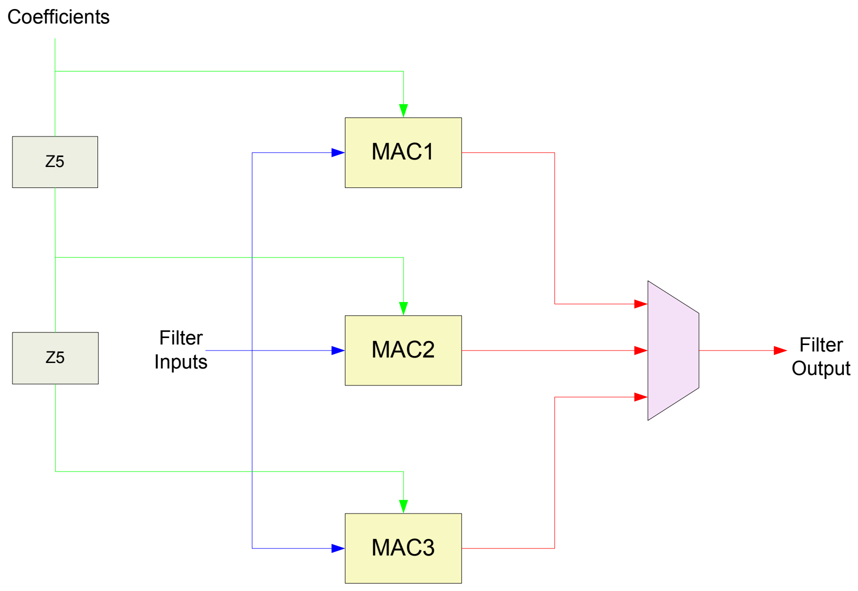

Figure 7: firdecim_m5_n15 Design Block Diagram

Table 4 shows the design parameters for firdecim_m5_n15.

Table 4: firdecim_m5_n15 Design Parameters

Design Parameter	Requirement
Tunable	No
Input clock rate	60 MHz
Input data rate	60 MHz
Output data rate	12 MHz
Number of taps	15
Decimation factor	5

7.1.2 Interface

Figure (8) shows the inputs and outputs of the firdecim_m5_n15 module.

7.1.3 firdecim_m5_n15 Module

Figure (9) shows how the inputs to the MACs are generated inside the module, and how the outputs of the MACs are generated to be the outputs of the firdecim_m5_n15 core.

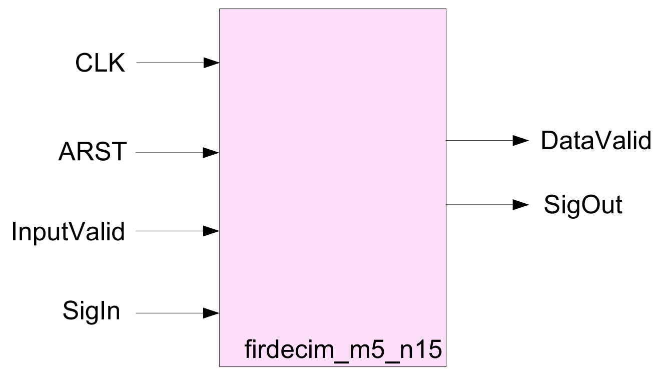

Figure 8: firdecim_m5_n15 Inputs and Outputs

Table 5: Interface Signals for firdecim_m5_n15

Signal	Direction	Size	Active Level	Description
CLK	Input	1-bit	Rising Edge	Clock input value.
ARST	Input	1-bit	High	Asynchronous reset. All module registers are cleared when this signal is high
InputValid	Input	1-bit	High	Indicate that SigIn is valid.
SigIn	Input	IWIDTH	N/A	Input data.
DataValid	Output	1-bit	High	Indicate that SigOut is valid.
SigOut	Output	OWIDTH	N/A	Output data.

There is an internal counter inside the firdecim_m5_n15 block that controls when each MAC gets initialized. Each input signal to the MACs come straight from the inputs of the firdecim_m5_n15 core. The coefficients to the MAC are generated this way:

MAC1	MAC filterCoef comes from coe0
	When the core gets ARST, the initialized value of coe0 is the parameter e0
MAC2	MAC filterCoef comes from coe10
	When the core gets ARST, the initialized value of coe10 is the parameter e10
MAC3	MAC filterCoef comes from coe5
	When the core gets ARST, the initialized value of coe5 is the parameter e5

After initialization, the coefficients gets rotated every clock that there is a valid input data. This way, the inputs gets multiplied by the correct coefficient.

7.2 Filter with R<1 and No Frequency Tuner

7.2.1 Design of firdecim_m5_n25

The firdecim_m5_n25 is a filter with 25 taps which filters the incoming data and decimates the input by a factor of 5. The core of the firdecim_m5_n25 design is the 5 accumulators. The basic idea behind the design for firdecim_m5_n25 is given in Figure (10).

Table 6 is a list of the design parameters for firdecim_m5_n25.

Since the Input clock rate vs. Input data rate ratio is 5:1, this design is able to take into advantage the face that the input data is valid for multiple numbers of clocks. By using this advantage, the design

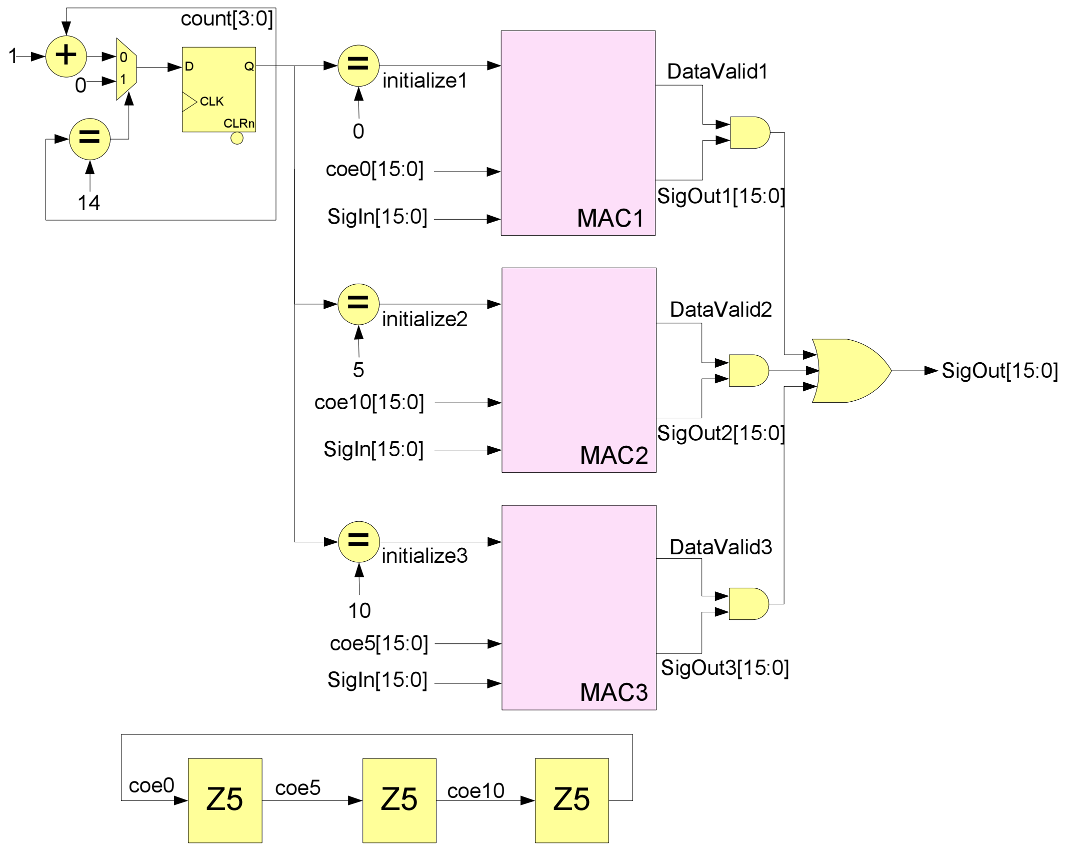

Figure 9: firdecim_m5_n15 Logic

Table 6: firdecim_m5_n25 Design Parameters

Design Parameter	Requirement
Tunable	No
Input clock rate	60 MHz
Input data rate	12 MHz
Output data rate	2.4 MHz
Number of taps	25
Decimation factor	5

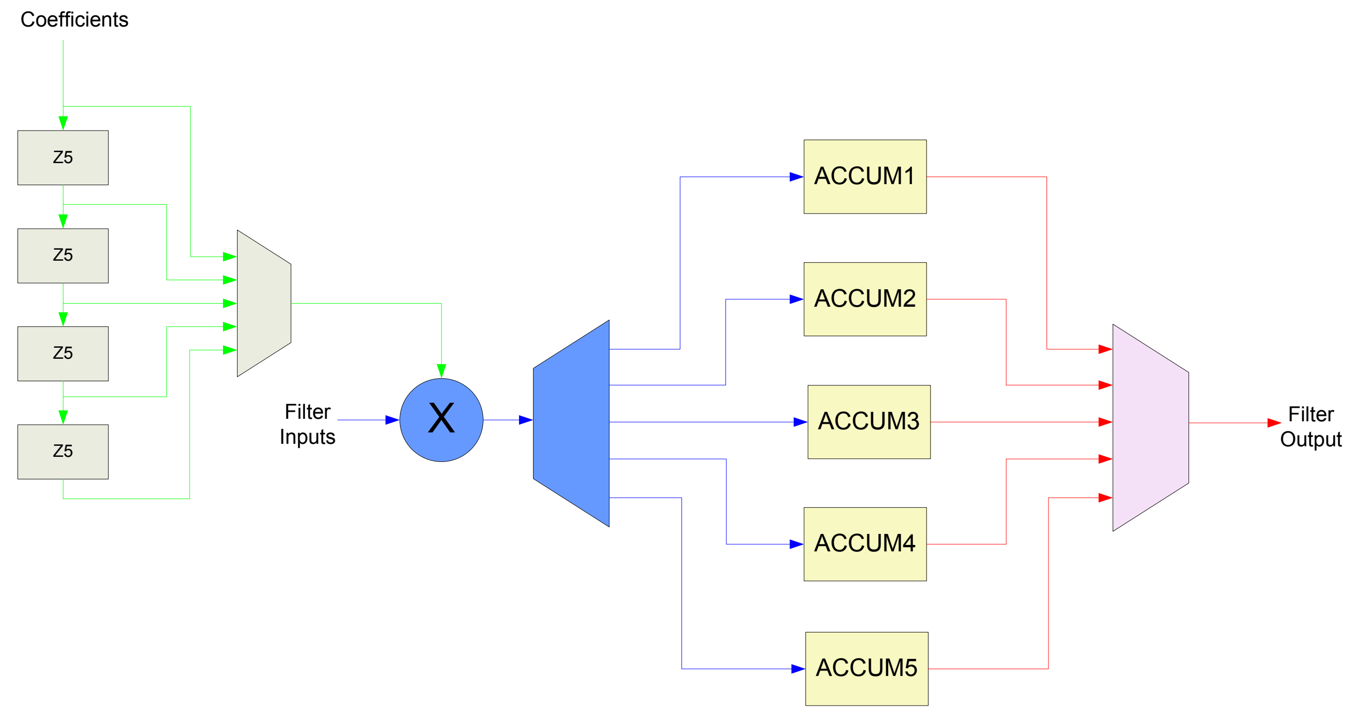

Figure 10: firdecim_m5_n25 Design Block Diagram

for firdecim_m5_n25 utilized one single multiplier with 5 accumulators.

7.2.2 Interface

Figure (11) shows the inputs and outputs of the firdecim_m5_n25 module.

Table 7: Interface Signals for firdecim_m5_n25

Signal	Direction	Size	Active Level	Description
CLK	Input	1-bit	Rising Edge	Clock input value.
ARST	Input	1-bit	High	Asynchronous reset. All module registers are cleared when this signal is high
InputValid	Input	1-bit	High	Indicate that SigIn is valid.
SigIn	Input	IWIDTH	N/A	Input data.
DataValid	Output	1-bit	High	Indicate that SigOut is valid.
SigOut	Output	OWIDTH	N/A	Output data.

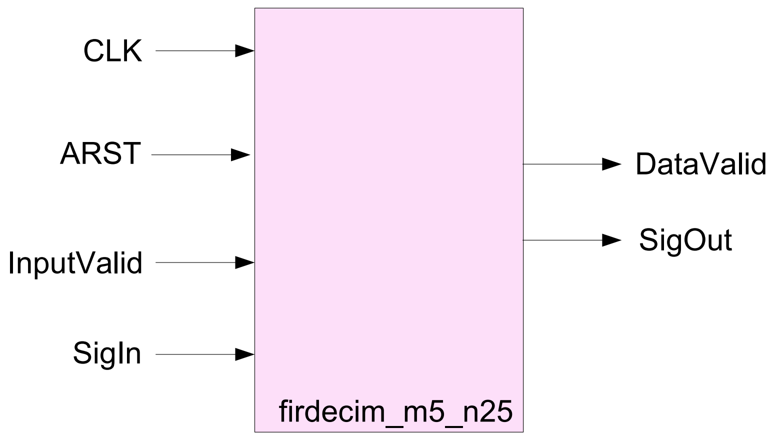

Figure 11: firdecim_m5_n25 Inputs and Outputs

7.2.3 Multiply-Accumulator

The Multiply-Accumulator in firdecim_m5_n25 is different compared to the MAC in firdecim_m5_n15. The function of the multiplier is to act as a filter with 25 taps. The function of the accumulator is to decimate the input down by 5. The multiplier in firdecim_m5_n25 runs at the internal clock speed, but only gets a new input data every 5 clocks. The advantage of this is the design can use the same input data to multiply with different coefficients every clock. This allows the design to use one single multiplier.

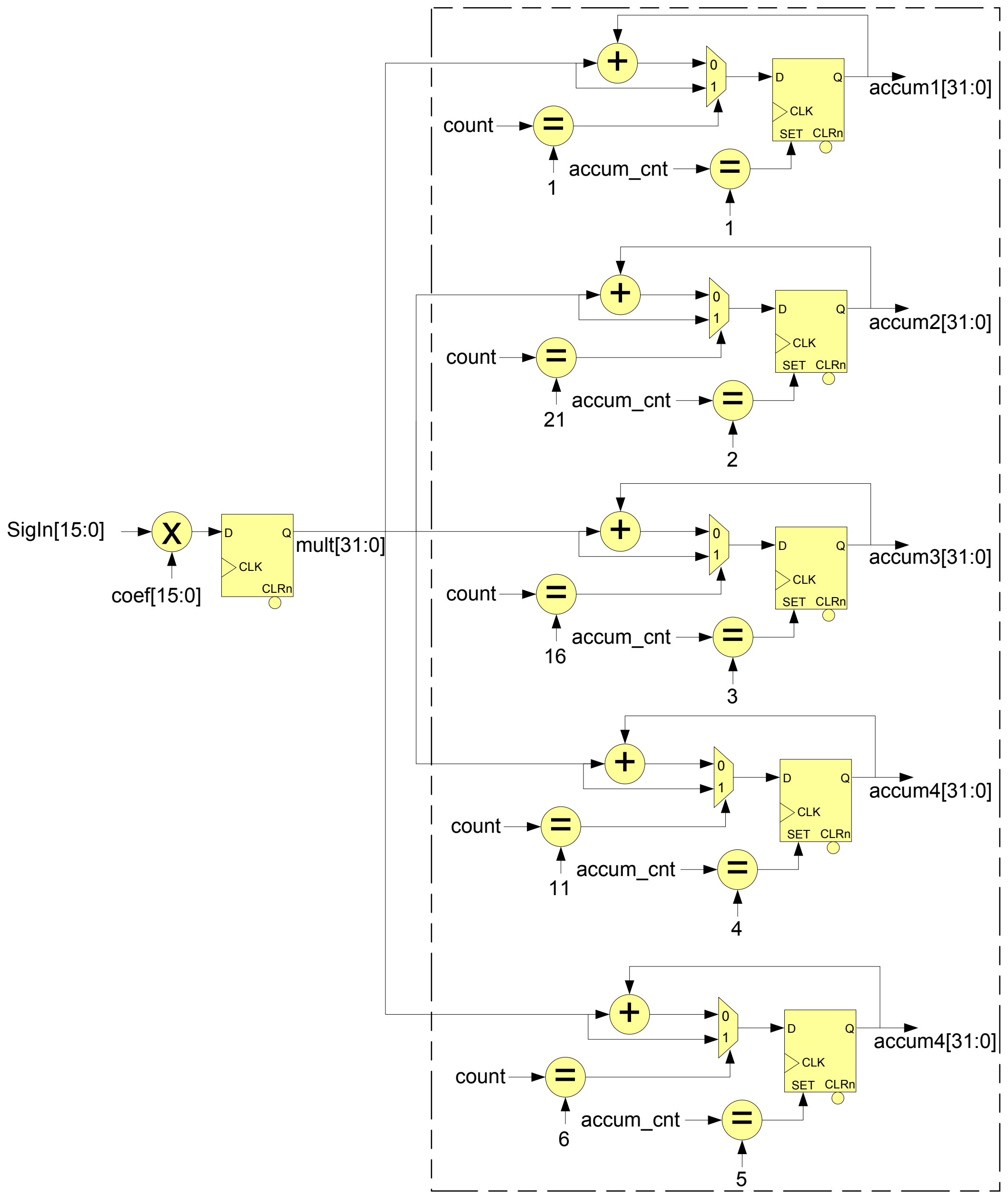

Figure 12: firdecim_m5_n25 Multiply Accumulate

7.2.4 firdecim_m5_n25 Module

Figure (13) shows how the output signals are generated with the inputs to the Multiply-Accumulator shown in the previous section.

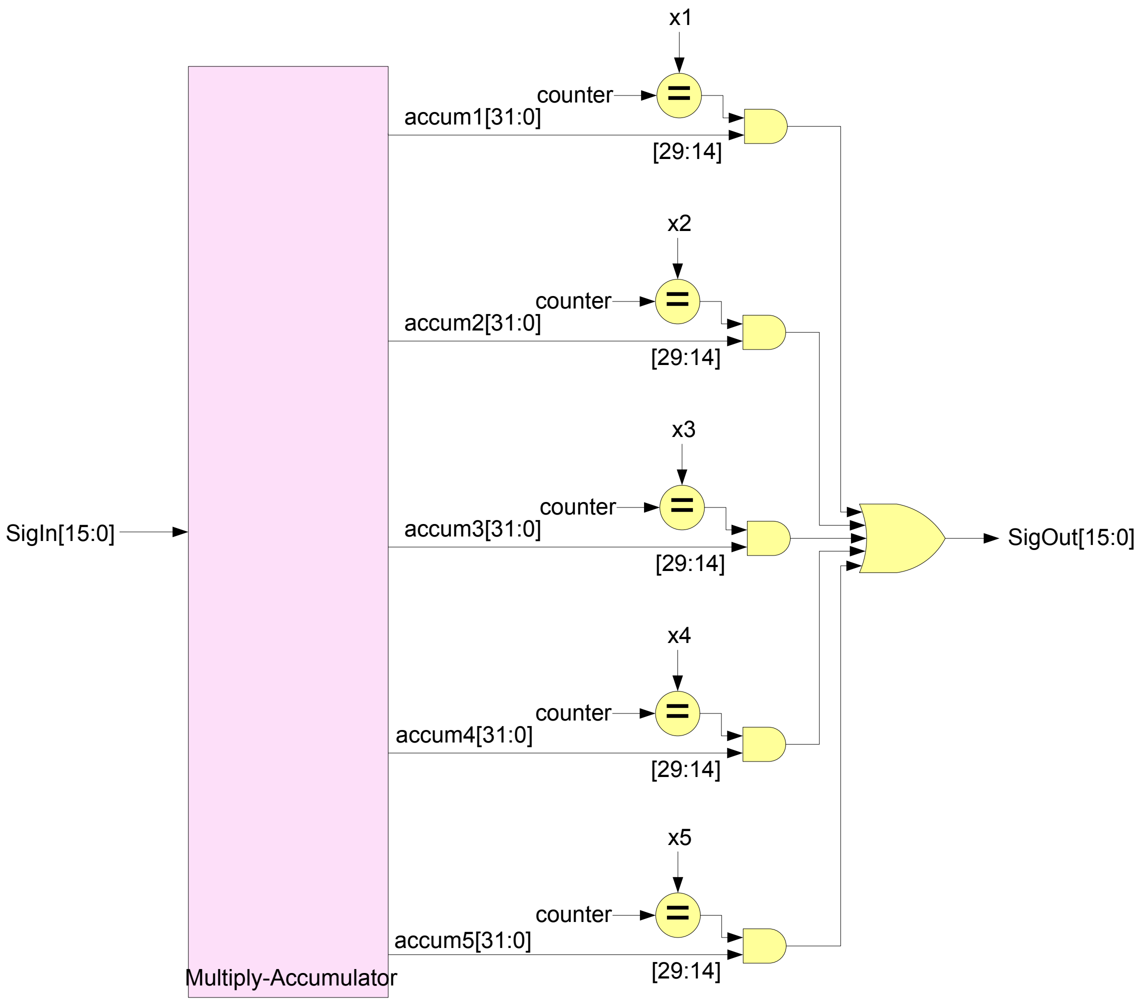

Figure 13: firdecim_m5_n25 Logic

7.3 Filter with R>1 and Frequency Tuner

7.3.1 Design of firdecim_m4_n12

The firdecim_m4_n12 is a filter with 12 taps which filters the input signal and decimates the input by a factor of 4. Table 8 is a list of the design parameters for firdecim_m4_n12.

From Figure 14, the number of multipliers utilized in the design is the same number as the number of taps on the filter. Since the input data rate is larger than the internal clock rate, there is no possible resource sharing schemes. The coefficients are stored in an internal ROM, and rotated every clock. Since this filter is able to tune to different frequency, the coefficients come from an external module which calculates the coefficients.

Table 8: firdecim_m4_n12 Design Parameters

Design Parameter	Requirement
Tunable	Yes
Input clock rate	60 MHz
Input data rate	240 MHz
Output data rate	60 MHz
(Decimation)	4
Number of taps	12
Decimation factor	4

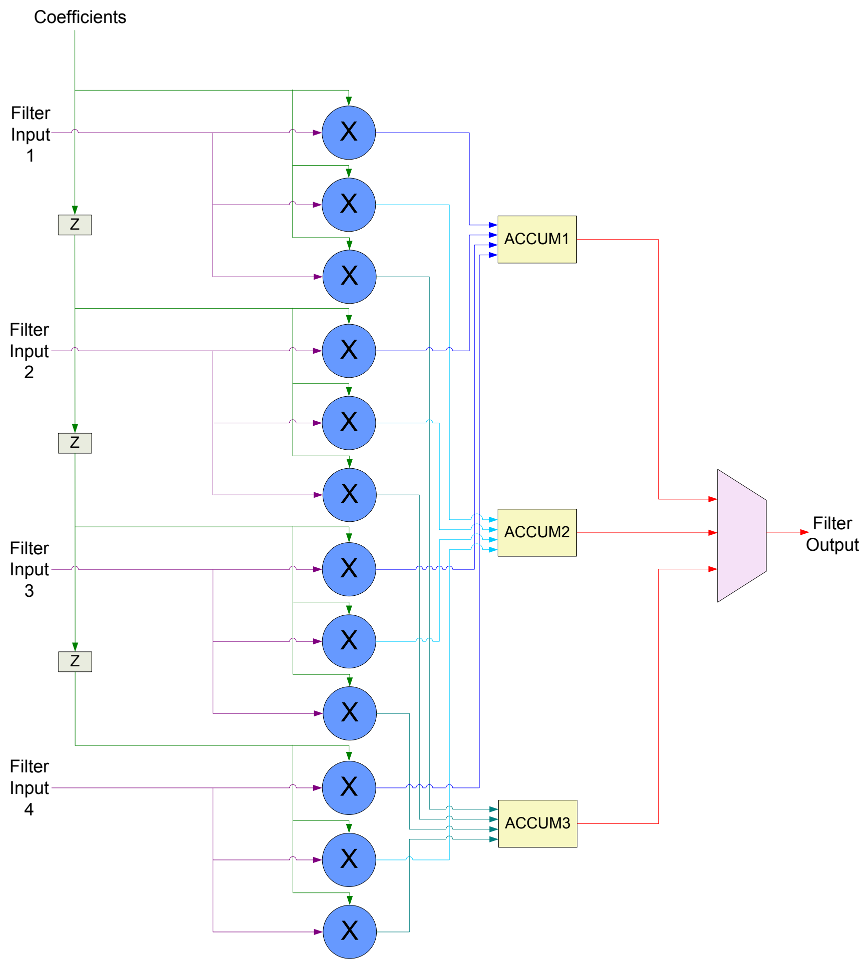

Figure 14: firdecim_m4_n12 Design Block Diagram

7.3.2 Interface

Figure 16 shows the inputs and outputs of the firdecim_m4_n12 module.

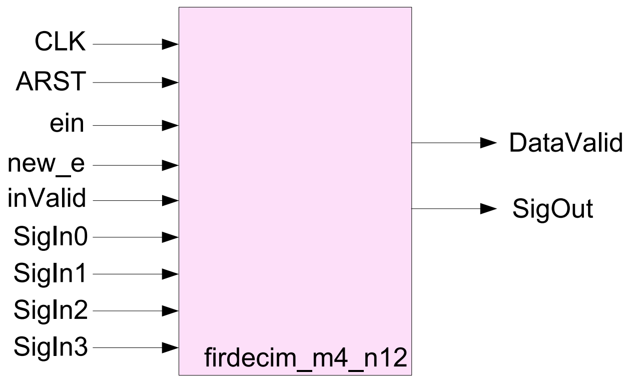

Figure 15: firdecim_m4_n12 Inputs and Outputs

Table 9: Interface Signals for firdecim_m4_n12

Signal	Direction	Size	Active Level	Description
CLK	Input	1-bit	Rising Edge	Clock input value.
ARST	Input	1-bit	High	Asynchronous reset. All module registers are cleared when this signal is high
ein	Input	16-bit	N/A	Serialized coefficient inputs. The values come from firdecim_m4_n12_freqcalc module. This is valid when new_e is high, and keeps on shifting in new values until new_e is deasserted.
new_e	Input	1-bit	High	Indicates that ein is valid. Number of clocks high is proportional to the number of taps in the filter.
InputValid	Input	1-bit	High	Indicate that SigIn is valid.
SigIn0	Input	IWIDTH	N/A	Parallel input data 0.
SigIn1	Input	IWIDTH	N/A	Parallel input data 1.
SigIn2	Input	IWIDTH	N/A	Parallel input data 2.
SigIn3	Input	IWIDTH	N/A	Parallel input data 3.
DataValid	Output	1-bit	High	Indicate that SigOut is valid.
SigOut	Output	OWIDTH	N/A	Output data.

7.3.3 firdecim_m4_n12 Frequency Tuner

Interface

Figure 16 shows the inputs and outputs of the firdecim_m4_n12_freqcalc module.

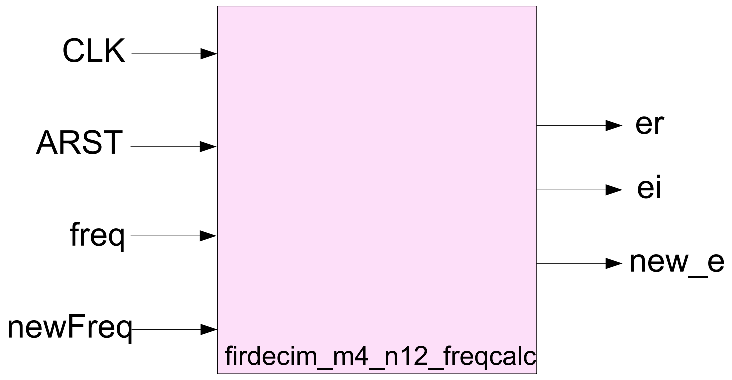

Figure 16: firdecim_m4_n12_freqcalc Inputs and Outputs

Table 10: Interface Signals for firdecim_m4_n12_freqcalc

Signal	**Direction**	**Size**	**Active Level**	**Description**
CLK	Input	1-bit	Rising Edge	Clock input value.
ARST	Input	1-bit	High	Asynchronous reset. All module registers are cleared when this signal is high
freq	Input	7-bit	N/A	Center frequency input in MHz. Range: 20MHz - 100MHz
new_Freq	Input	1-bit	High	Indicates the freq is new and this module needs to calculate a new set of coefficients.
er, ei	Output	16-bit	N/A	Serialized coefficient outputs. er is real (multiplied by Cosine), ei is imaginary (multiplied by Sine).
new_e	Output	1-bit	High	Indicates that er and ei are valid. Should be asserted for a number of clocks that matches the number of filter taps of the filter design.

7.3.4 firdecim_m4_n12 Wrapper

Interface

Figure 17 how the frequency tuner interacts with the filter to form the wrapper called firdecim_m4_n12_wrapper.

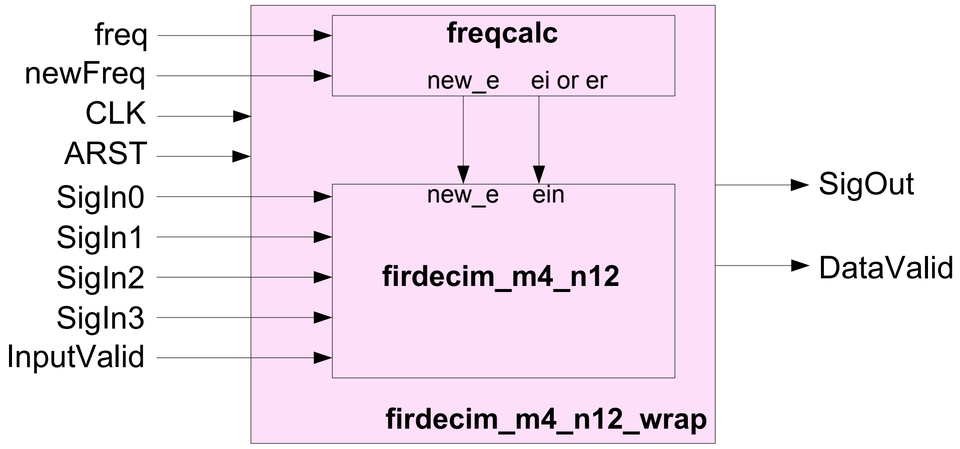

Figure 17: firdecim_m4_n12_freqcalc Wrapper Module

8 Repository Location

The firdecim design files are located on the ISAAC server SVN:

//isaacdev/proj/isaac/repo/edk_repository/iCore/DSP/Filter/FIRDecim/firdecim_ISAAC

References

[1] S. Smith, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, vol. 6, pp. 4–19, July 1989.

[2] [Online]. Available: http://www.cis.upenn.edu/~milom/cse372-Spring06/xilinx/cgd.pdf

[3] [Online]. Available: http://www.xilinx.com/itp/xilinx8/books/data/docs/dev/dev0190_28.html