# Serial FIR Filter

## User's Guide

October 2005
ipug13_02.0

## Introduction

The Serial FIR Filter core is one of two FIR cores supported by Lattice. This core is an area-efficient implementation that uses serial arithmetic elements to achieve compact size.

This user's guide explains the functionality of the Serial FIR Filter core and how it can be configured to support different filter types.

The Serial FIR Filter core comes with the documentation and files listed below:

- Data sheet
- Lattice gate level netlist
- RTL simulation model
- Core instantiation template

## Features

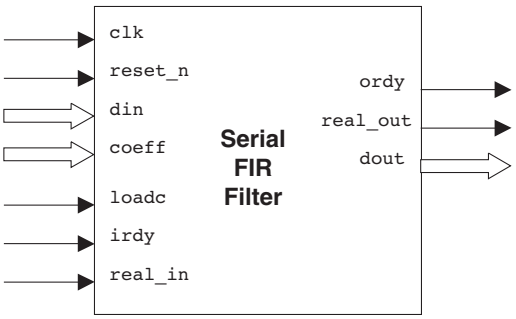
- Serial arithmetic for reduced resource utilization
- Variable number of taps up to 64
- Data and coefficients up to 32 bits
- Output size consistent with data size
- Signed or unsigned data and coefficients
- Full arithmetic precision
- Fixed or loadable coefficients
- Decimation and interpolation
- Real or complex data
- Selectable rounding
- Scalable outputs
- Fully serial implementation
- Multi-cycle modes for area/time tradeoff
- Optimization based on symmetry of filter

## General Description

Many digital systems use filters to remove noise, provide spectral shaping, or perform signal detection. The two common types of filters that provide these functions are: infinite impulse response (IIR) and finite impulse response (FIR) filters. IIR filters are used in systems that can tolerate phase distortion. FIR filters have an inherently stable structure and are used in systems that require linear phase. For this reason, FIR filters are more commonly used.

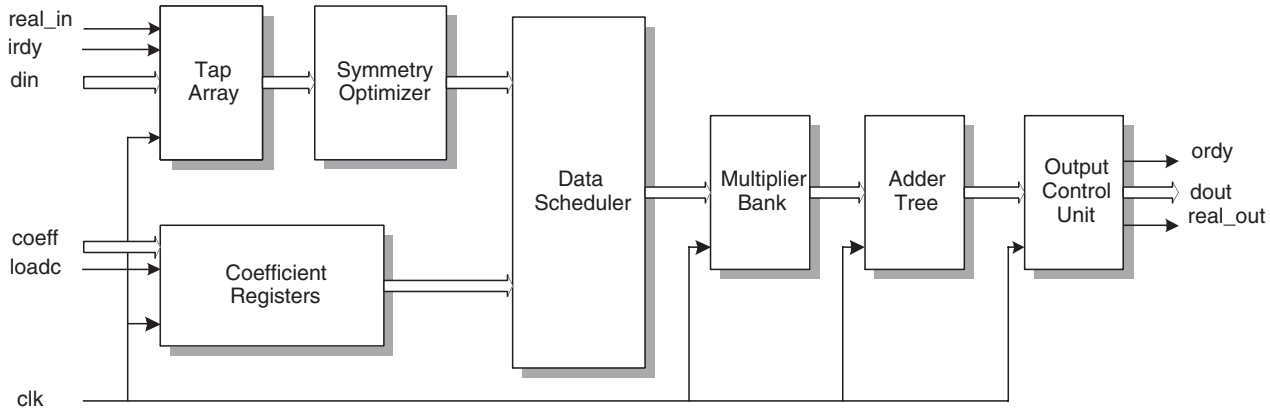
The Lattice Serial FIR Filter uses serial arithmetic elements and is well suited for area-crucial FPGA applications. The core supports a wide range of taps, data lengths and architectures. It supports single cycle mode as well as multi cycle mode. It can be used to implement decimation and interpolation filters. Additionally, the core can operate with complex or real data types. The coefficients can either be fixed at the time of core generation or dynamically loaded during runtime, thus allowing use in adaptive filtering. The architecture is optimized depending on the symmetry type of the filter. Symmetric odd and symmetric even types are recognized in addition to the non-symmetric types. Figure 1 shows the Serial FIR Filter interface diagram.

***Figure 1. Serial FIR Filter Interface Diagram***




## Block Diagram

***Figure 2. Serial FIR Filter Internal Block Diagram***




## Functional Description

Figure 2 shows the internal block diagram of the serial FIR filter core. A brief description of each functional module is given below. The following notations are used in this document:

- *n* - number of taps
- *w* - width of input data and coefficients
- *c* - number of cycles for multi-cycle operation
- *d* - decimation ratio
- *u* - interpolation ratio
- *m* - number of multipliers, determined as: *m* = next highest integer to (*n/c*)
- *ow* - output width
- *ofw* - output full width
- PP - processing period, one PP is composed of *ofw* + 1 clock cycles

### Tap Array

The Tap Array module stores delayed versions, or taps, of input data. The number of taps of the FIR filter and the data width are user-defined parameters fixed at the time of core generation. The Tap Array consists of *n* taps, each of width *w*, organized as shift registers. All data registers are reset when signal `reset_n` is asserted. At the end of every processing cycle, the data values are shifted into the next sequential shift register inside the tap array, with the first register getting the value from input data port `din`.

## Coefficient Registers

The Coefficient Registers module stores the FIR filter coefficients. These coefficients can either be loaded during run time or fixed at core generation. If the user chooses to fix the coefficients, then the register values are hard-coded and the `coeff` and `loadc` ports are not used. If the user chooses to load the coefficients, they are loaded into the coefficient registers sequentially at every clock edge. The coefficient loading starts at the first clock edge after `loadc` goes high and continues as long as `loadc` is active.

## Symmetry Optimizer

This module is used for symmetric filters and contains the adders/subtractors necessary for implementing symmetry. A serial adder is used for realizing even-symmetric filters and a serial subtractor is used for odd-symmetric filters.

## Data Scheduler

The Data Scheduler schedules the tap and coefficient data to the multiplier bank for multi-cycle computations. This module has the necessary multiplexers to supply the tap and coefficient data to the multiplier bank in batches. For a multi-cycle implementation with *c* cycles, the number of multipliers, *m,* is equal to (*n/c*) rounded to the next higher integer. For a single-cycle implementation (*c* = 1), the data scheduler reduces to a direct connection. The data scheduler is also used to multiplex data for optimizing decimation and interpolation filters.

## Multiplier Bank

The Multiplier Bank has *m* number of *w* bit serial multipliers, where *m* is the number of taps *n* divided by the number of computational cycles *c* rounded to the next higher integer (*m* = ceil (*n/c*)). For single cycle implementations, the number of multipliers is equal to the number of taps. The input to the Multiplier Bank comes from the data scheduler, and the output goes to the adder tree. The maximum delay through the Multiplier Bank is equal to the delay of a singe multiplier.

## Adder Tree and Output Control Unit

The Adder Tree contains a binary tree of serial adders, all operating in parallel. The Output Control Unit contains the scaling and rounding logic required for output scalability and selectable rounding. This block also contains data registers to provide synchronous registered output from the filter core. For multi-cycle or decimation filtering, an adder is present in the block, which when combined with the output registers, makes an accumulator.

## Parameter Descriptions

User configuration parameters such as filter type, data width, number of taps and data type, which are configurable, are described in Table 1. These parameters are configured using IPexpress™, included with Lattice's ispLEVER® design tools.

***Table 1. Serial FIR Filter Parameter Definitions***

| Parameter | Default Value | Value | Description |
|---|---|---|---|
| `Filter Type` | Single-Cycle | Single-Cycle, Multi-Cycle, Decimation or Interpolation | Type of filter selected by the user. This determines the rest of the parameter options. |
| `Data Width` | 8 bits | Real: 4 to 32 bits Complex: 4 to 16 bits | Width of input data (*w*) in bits. The width of the coefficients is also equal to this parameter. For complex data types, the `Data Width` is equal to the width of the real part and the range is from 4 to 16 bits. |
| `Number of Taps` | 16 | 4 to 64 | Number of taps (*n*) in the filter. |
| `Computational Cycles` | 2 | 2 to 32 | Number of cycles (*c*) for multi-cycle filters. Number of processing cycles (PP) to perform the filtering process. The output is computed once in *c* PPs. |
| `Decimation Ratio` | 2 | 2 to 32 | Decimation is downsampling of the bit stream. In a decimation filter with decimation ratio '*d*', the output data rate is 1/*d* of the input rate. |
| `Interpolation Ratio` | 2 | 2 to 32 | For interpolation filters. Interpolation is the reverse of decimation. The input rate is 1/*u* of the output rate. |
| `Rounding Method` | Nearest | Truncation or Nearest | Types of rounding available. |
| `Arithmetic Type` | Signed | Signed or Unsigned | Specifies the type of arithmetic modules for the core. If the symmetry of the core is even or odd, then the arithmetic type is always signed. |
| `Data Type` | Real | Real or Complex | Specifies the data type of the inputs (`din` and `coeff`) and the output (`dout`) of the Serial FIR core. When complex `data type` mode is selected, the arithmetic type is always signed. |
| `Complex I/O Mode` | Parallel | Parallel or Serial | In the parallel I/O mode, real and imaginary parts are applied on the data bus in the same clock cycle. In the serial mode, real data is applied in the first PP cycles, followed by the imaginary data in the next PP cycles. |
| `Output Width` | Full Precision | 4 to 96 | Width of output data (*w*) in bits. If the width is less than the maximum output width determined by the core generator, the outputs are scaled. |
| `Coeffs Loadable` | Run-time Loadable | Fixed or Run-time Loadable | Determines if the coefficients are run-time loadable. If the coefficients are run-time loadable, the core has two additional input `ports`, `coeff` and `loadc`, for loading purposes. If the coefficients are fixed during core configuration, no additional input ports are used. |
| `Coefficients Format` | Hexadecimal | Hexadecimal or Decimal | The coefficient values can be entered in either in hexadecimal or decimal format. |
| `Symmetricity` | Even | None, Even, or Odd | Specifies the impulse response of the filter. Even Symmetricity applies to symmetric impulse response, while Odd Symmetricity applies to anti-symmetric impulse response. Decimation and Interpolation filters do not have Symmetricity (the value None should be selected). If the Symmetricity of the core is even or odd, then the arithmetic type is always signed. |

## Signal Descriptions

Table 2 shows the signal descriptions for the input and output ports of the Serial FIR Filter IP.

***Table 2. Serial FIR Filter Signal Definitions***

| Port Name | I/O Type | Active State | Signal Description |
|---|---|---|---|
| clk | Input | Rising edge | **Clock**. Master clock input to the Serial FIR FILTER core. |
| din[31..3:0] | Input | N/A | **Data Input**. Data to be processed. In the complex parallel I/O mode, the din bus includes both the real and imaginary parts |
| dout[127..3:0] | Output | N/A | **Data Output**. The data is the filter output. In the complex parallel I/O mode, the dout bus includes both the real and imaginary parts |
| reset_n | Input | Low | **Reset**. This signal resets all the delayed data signals to 0. |
| coeff[31..3:0] | Input | N/A | **Coefficient**. Coefficients for the filter are loaded sequentially while asserting the loadc signal. |
| loadc | Input | High | **Load Coefficient**. This signal is asserted to load the filter coefficients (coefficient values on the coeff bus). |
| irdy | Input | High | **Input Ready**. irdy is asserted to indicate the availability of valid input data. |
| real_in | Input | High | **Real Part Input**. real_in is asserted to indicate that the real part of the complex data is being input at din. This signal is available only in complex-serial mode. |
| ordy | Output | High | **Output Ready**. ordy is asserted by the core to signify the availability of valid data at dout. |
| real_out | Output | High | **Real Part Output**. real_out is asserted to indicate that the real part of the complex data is being presented at dout. This signal is available only in complex-serial mode. |

## Core Operation and Timing

There are four distinct implementations of Serial FIR Filter: Single-Cycle, Multi-Cycle, Decimation and Interpolation. This section describes these implementation types in detail. A note on rounding and truncation is also given in this section.

Complex data type is supported in all these implementations. For complex data type, the complex input data can be either supplied all at once (complex-parallel) or in two stages, real data followed by imaginary data (complex-serial).

### Single Cycle

This is the simplest of all implementations, in that it assumes availability of sufficient resources for simultaneous computations. For an *n*-tap filter, a Single Cycle implementation uses *n* multipliers and *n*-1 adders. A new output is available once every processing period (PP). The timing diagrams for the single-cycle implementations are given in Figures 3 and 4. As seen in the timing diagram, real and imaginary parts of the input are supplied in successive PP in complex serial mode. The input irdy should be asserted high to coincide with the first clock cycle of every valid input data at the din port. For complex serial mode, the signal real_in should be asserted high at the first clock cycle of every valid real data. Similarly, the core asserts the output real_out when the real part of the output data is placed on the output bus.

***Figure 3. Timing for Single-Cycle, Real or Complex-Parallel Mode***

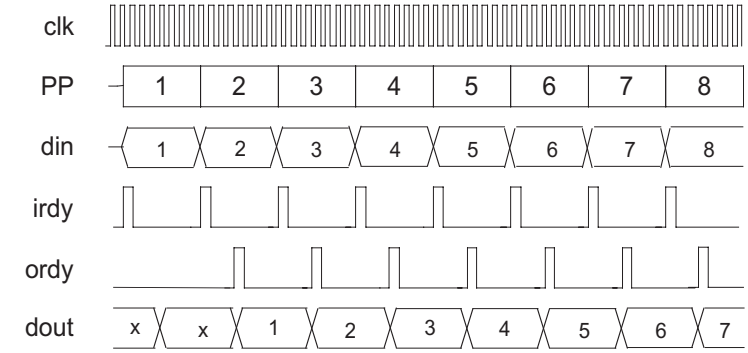
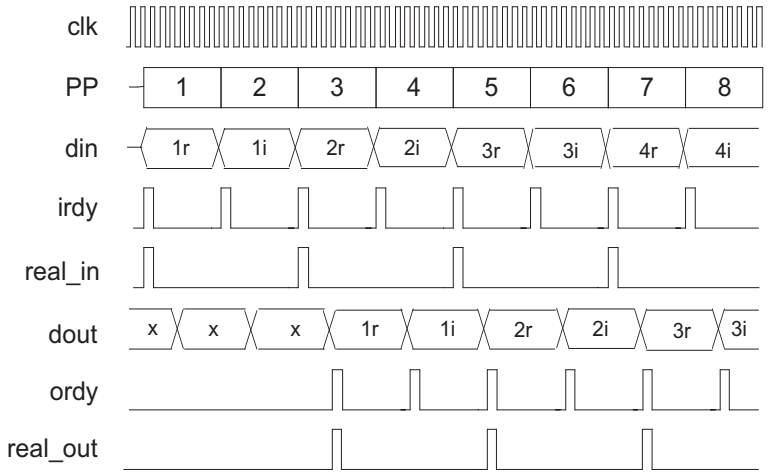



***Figure 4. Timing for Single-Cycle, Complex-Serial Mode***



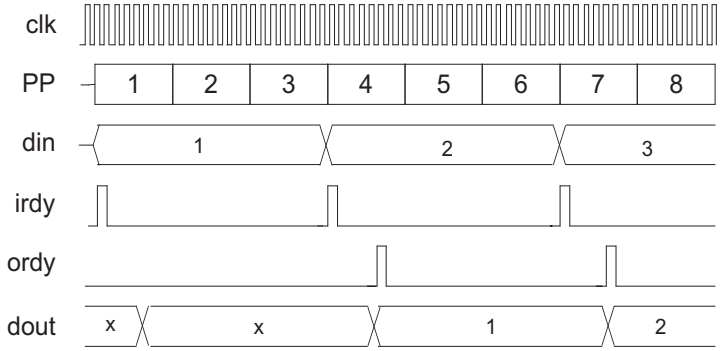

## Multi Cycle

In a multi-cycle implementation, each output is computed over a period of *c* PPs. The implementation is similar to the single-cycle implementation, except that fewer resources are used over multiple processing cycles. The number of multipliers and adders used is not more than 1/*m* of those used in a single cycle implementation. In a multi-cycle implementation, there is an additional accumulator (an adder and a register combination) to accumulate the final sum through the *c* PPs. The timing diagrams for multi-cycle implementations are given in Figures 5 and 6.

## Real and Complex-Parallel Modes

In both the Real and Complex-Parallel modes, the signal `irdy` is asserted during the first clock cycle of a multi-cycle operation. The data output of the core changes every *c* PPs. The output `ordy` goes high during the first clock cycle of every *c* PPs. This operation is shown in Figure 5.

***Figure 5. Timing for Multi-cycle, Real or Complex-Parallel Mode***

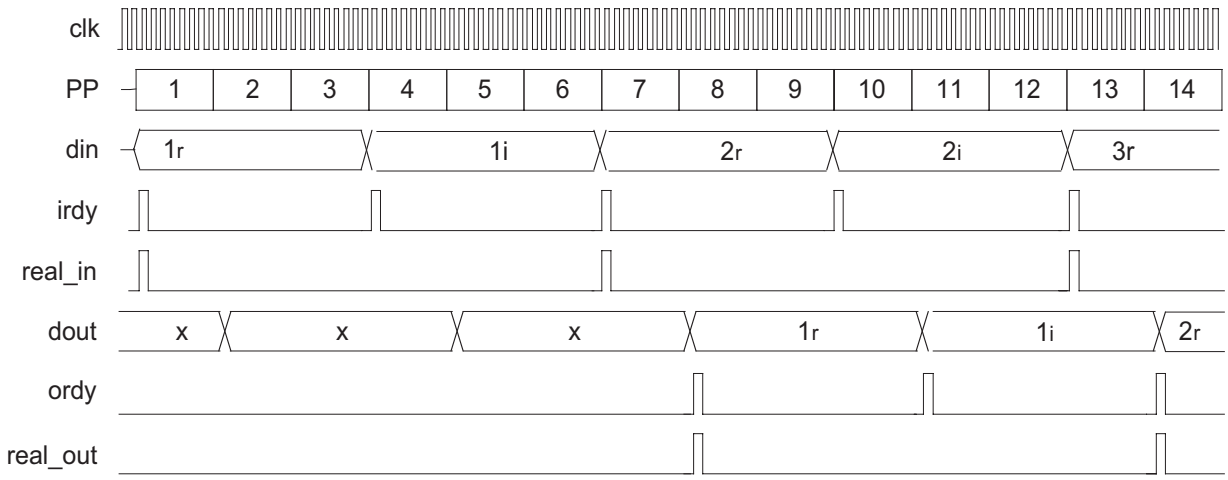



## Complex-Serial Mode

The data and handshake signals for a typical complex-serial mode configuration (*c* = 3) are shown in Figure 6. In this mode, every input data cycle contains a real data cycle followed by an imaginary data cycle. Each of these real and imaginary data cycles are *c* PPs wide. The `irdy` input signal must be asserted high during the first cycle of every real or imaginary input data cycle. The `real_in` input signal must be asserted high during the first cycle of every real input data cycle.

The output data cycles also contain a real data cycle followed by an imaginary data cycle. The `ordy` output signal goes high during the first clock cycle of every real data cycle or imaginary data cycle. The `real_out` output signal goes high to indicate valid data during the real data cycle and is only active for one clock cycle.

***Figure 6. Timing for Multi-cycle, Complex Serial Mode***




## Decimation

Decimation is down-sampling of the data stream. In a simple decimation filter with decimation ratio *d,* every $d^{th}$ sample of the input is sent to the output. The danger with down sampling is that aliasing can occur if the input signal is not band-limited to 1/*d* of the original bandwidth. To prevent aliasing, it is necessary to do a low pass filtering of the input data before down sampling. The decimation filter is, therefore, a cascade of a low pass filter and a down sampler. The implementation of this is similar to a normal FIR, except that *d* - 1 samples are skipped at the output after every valid output. The output data rate is 1/*d* of the input rate. Fewer arithmetic resources are required for decimation filter implementations, as it is not necessary to compute an output for every input sample.

The output signal ordy goes high during the first cycle of each data output. For, complex-serial mode there is an additional output, `real_out`, which goes high during the first cycle of every real part of the complex data.

The timing diagrams for two decimation filter implementations are shown in Figures 7 and 8.

***Figure 7. Timing for Real or Complex-Parallel, Decimation Mode***

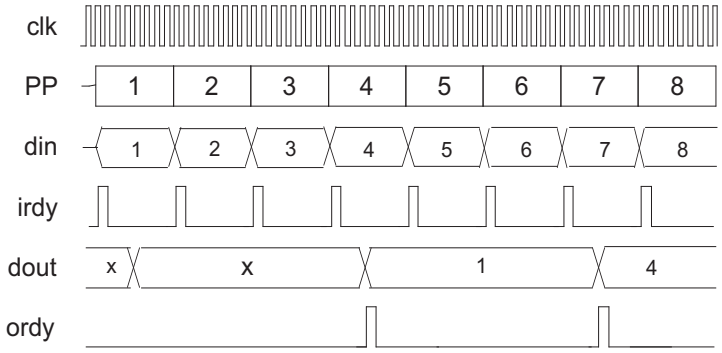
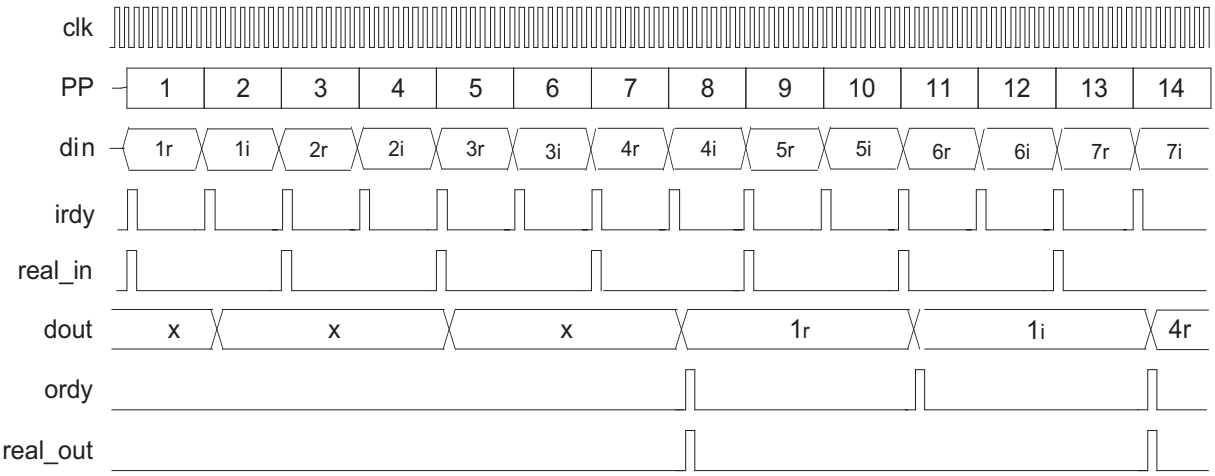



***Figure 8. Timing for Complex-Serial, Decimation Mode***




## Interpolation

Interpolation is the reverse process of decimation. In this mode, the data is upsampled. For an interpolation ratio *u, u* - 1 'zeros' are introduced between any two consecutive samples and the resulting expanded stream is passed through a low pass filter. The operational environment of an interpolation filter would be similar to a regular FIR filter, except that the input data rate is reduced by *u* and 0's are introduced in the taps. The timing diagrams for two interpolation filter implementations are shown in Figures 9 and 10.

***Figure 9. Timing for Real or Complex-Parallel, Interpolation Mode***

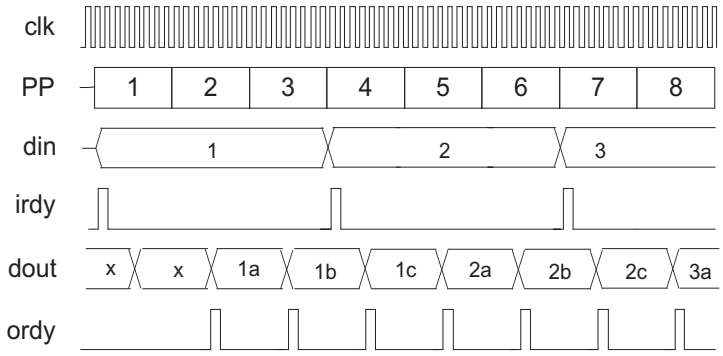
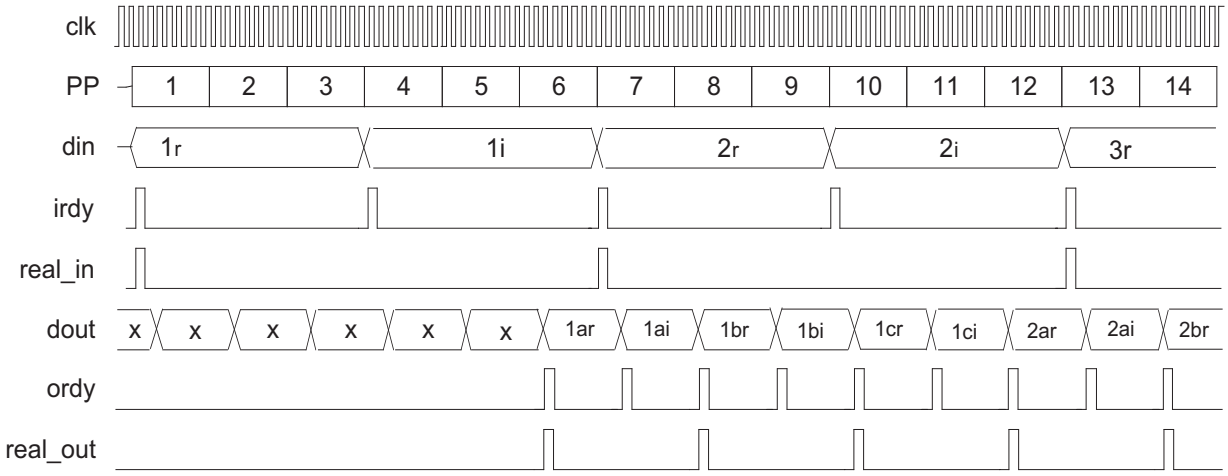


***Figure 10. Timing for Complex-Serial, Interpolation Mode***




## Output Scaling and Rounding

When the user defined output width (*ow*) of the filter output is less than the full output width of the filter (*ofw*), the outputs are scaled using a rounding scheme based on the parameter "rounding method". If the rounding method is defined as "truncation", the least significant *ofw-ow* bits are simply discarded and the most significant *ow* bits are retained in the output. If the rounding method is selected to be "nearest", the least significant *ofw-ow* bits are discarded, and the most significant *ow* bits are rounded based on the value of the least significant bits that were discarded.

Truncation takes the value to the next step towards negative infinity, and rounding nearest takes the value to the nearest step in either direction.

Table 3 gives an example that illustrates the output scaling and rounding for two numbers using integer, fixed point, signed and unsigned representations. In the example, the full output width (*ofw*) is 8 and the desired output width (*ow*) is 6. Output scaling in this case is equivalent to a division by 4.

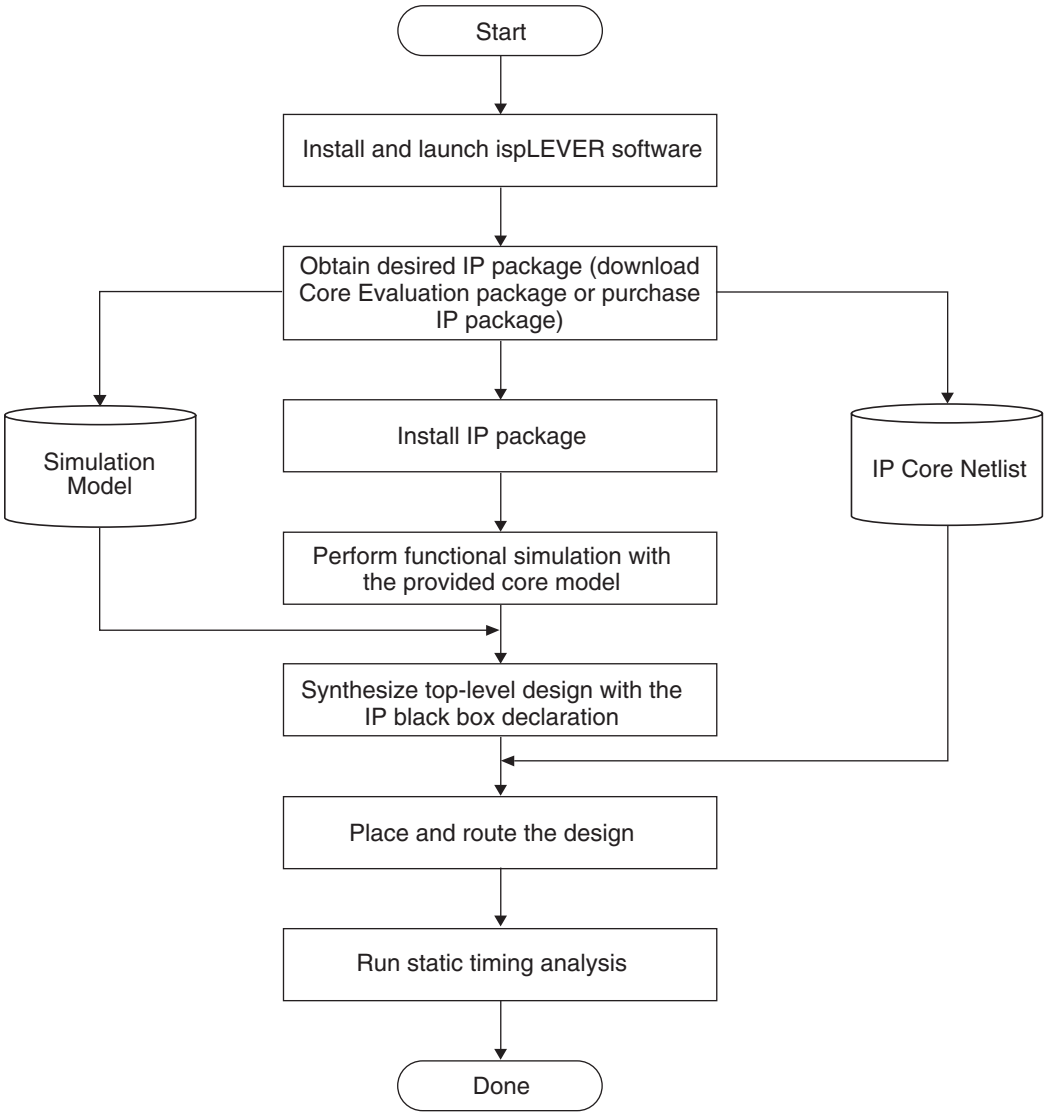
***Table 3. Example Description of Output Scaling and Rounding***

| Binary | Mode | Decimal | Full Precision Divide by 4 | Truncation | Nearest |
|---|---|---|---|---|---|
| 1010 1001 | Unsigned, integer | 169 | 42.25 | 42 | 42 |
| | Signed, Integer | -87 | -21.75 | -22 | -22 |
| | Unsigned, FP between bit 3 and bit 4 | 10.5625 | 2.640625 | 2.625 | 2.625 |
| | Signed, FP between bit 3 and bit 4 | -5.4375 | -1.359375 | -1.375 | -1.375 |
| 1010 1011 | Unsigned, integer | 171 | 42.75 | 42 | 43 |
| | Signed, Integer | -85 | -21.25 | -22 | -21 |
| | Unsigned, FP between bit 3 and bit 4 | 10.6875 | 2.671875 | 2.625 | 2.6875 |
| | Signed, FP between bit 3 and bit 4 | -5.3125 | -1.328125 | -1.375 | -1.3125 |

## Serial FIR Filter IP Core Design Flow

The Serial FIR Filter IP Core can be implemented using the push-button Graphical User Interface (GUI) flow accessed through the IP Manager tool in the ispLEVER design tool. Figure 11 illustrates the software flow used when evaluating the Serial FIR Filter Core.

***Figure 11. Lattice IP Core Evaluation Flow***




## IPexpress

The Lattice IP configuration tool, IPexpress, is incorporated in the ispLEVER software. IPexpress includes a GUI for entering the required parameters to configure the core. For more information on using IPexpress and the ispLEVER design software, refer to the software help and tutorials included with ispLEVER. For more information on ispLEVER, see the Lattice web site at: www.latticesemi.com/software.

### Functional RTL Simulation Under ModelSim (PC Platform)

Once the serial FIR core has been downloaded and unzipped to the designated directory, the core is ready for evaluation.

A simulation script file is provided in the "eval" directory for RTL simulation. The script file vsim_fir_serial.do uses pre-compiled models provided with this package. The pre-compiled library of models is located in the directory:

- fir_ser_xp_1_00x/xpga/ver1/lib/modelsim/work.

**Simulation Procedures**

1. Launch ModelSim
2. Using the main GUI, change the directory location:
   Select: File > Change Directory > fir_ser_xp_1_00x/xpga /ver1/eval/simulation
3. Execute <Modelsim macro name>.d
   Select: Macro > Execute Macro > scripts/ vsim_fir_serial.do

The functional simulation for IP Cores is not currently applicable with the OEM version of ModelSim embedded in the ispLEVER 3.0 software. For more information on how to use ModelSim, please refer to the *ModelSim User's Manual.*

## Core Implementation

Users can instantiate the IP Core to implement into their system design. The following Verilog source files for Serial FIR Filter core are provided:

- fir_ser_xp_1_00x.v for the Serial FIR Filter core-top RTL source
- sfir_top.v for top-level source

Users can use the core-top RTL as a black box to the system designs. All default signal names in the top-level RTL source file must be replaced with real signal names from the system design.

## Black Box Consideration

Since the core is delivered as a gate-level netlist, the synthesis software will not re-synthesize the internal nets of the core. In the synthesis process, the instantiated core must be declared as a black box. The ispLEVER software automatically detects the provided netlist of the instantiated IP core in the design. For more detailed information regarding Synplify's black box declaration, please refer to the following sections of the Synplify reference manual:

The core implementation consists of synthesis and place and route sections. Each of the sections is described below.

Two synthesis tools, Synplicity® Synplify® and LeonardoSpectrum™, are included in the ispLEVER software for seamless processing of designs. The current IP cores are being tested with EDIF flow. The following is a step-by-step procedure for each synthesis tool to generate an EDIF netlist containing the IP core as a black box.

## Synthesis Using Synplify

The step-by-step procedure provided below describes how to run synthesis using Synplify.

1. Create a new working directory for synthesis
2. Launch the Synplify synthesis tool
3. Start a new project and add the specified files in the following order:
   ~/source/fir_ser_xp_1_00x_params.v
   ~/source/fir_ser_xp_1_00x.v
   ~/source/sfir_top.v
4. In the Implementation Options select a target device LFX1200B

5. Specify an EDIF netlist filename and EDIF netlist output location in the Implementation Options. This top-level EDIF netlist will be used during Place and Route.

6. In the Implementation Options, set the following:
   - Enable FSM Compiler
   - Disable Resource Sharing
   - Set the global frequency constraint to 200 Mhz.

7. Select Run

## Synthesis Using LeonardoSpectrum

The step-by-step procedure provided below describes how to run synthesis using LeondardoSpectrum.

1. Create a new working directory for synthesis

2. Launch the LeonardoSpectrum synthesis tool

3. Start a new project and open the specified files in the following order
   ~/source/fir_ser_xp_1_00x_params.
   ~/source/fir_ser_xp_1_00x.v
   ~/source/sfir_top.v

4. Select Lattice device technology LFX1200B

5. Set the working directory to the path where you would like to save the output netlist.

6. Specify an EDIF netlist filename for the output file. This top-level EDIF netlist will be used during Place and Route.

7. Select Run Flow

Note: <top-level RTL source> could be the user's top-level design or the top-level source (sfir_top.v) file in the source directory of the downloaded package.

## Place and Route

Once the EDIF netlist is generated, the next step is to import the EDIF into the Project Navigator. The step-by-step procedure provided below describes how to perform place and route in ispLEVER for an ispXPGA™ device:

1. Create a new working directory for Place and Route

2. Start a new project, assign a project name and select the project type as EDIF

3. Select an ispXPGA LFX1200B device, with -4 speed grade and 680 fpBGA package

4. Copy the following files to the Place and Route working directory:
   a) ..\..\par\fir_ser_xp_1_00x.ld2
   b) ..\..\par\fir_ser_xp_1_00x.lct
   c) The top-level EDIF netlist generated from Running Synthesis

5. Rename the fir_ser_xp_1_00x.lct file (in step 4) to match the project name. For example, if the project name is "demo", then the .lct file must be renamed to demo.lct. The preference file name must match that of the project name.

6. Import EDIF netlist into the project

7. Select the Post-Route Timing Report in the Project Navigator to execute Place and Route and generate a timing report for ispXPGA.

## References

- ispLEVER Software Online Help Manual

## Technical Support Assistance

Hotline: 1-800-LATTICE (North America)
+1-503-268-8001 (Outside North America)
e-mail: techsupport@latticesemi.com
Internet: www.latticesemi.com

# Appendix for ispXPGA® FPGAs

***Table 4. Performance and Resource Utilization***

| Parameter | LUT4s[2] | PFUs[3] | Registers | External Pins | System EBRs | $f_{MAX}$[1] |
|---|---|---|---|---|---|---|
| fir_ser_xp_1_002.lpc[4] | 260 | 115 | 382 | 41 | None | 185 |

1. Performance and utilization characteristics are generated using LFX1200B-04FE680C in Lattice ispLEVER 3.x software. The evaluation version of this IP core only works on this specific device density, package, and speed grade.
2. Look-Up Table (LUT) is a standard logic block of Lattice devices. LUT4 is a 4-input LUT. For more information check the data sheet of the device.
3. Programmable Function Unit (PFU) is a standard logic block of some Lattice devices. For more information, check the data sheet of the device.
4. Configuration fir_ser_xp_1_002.lpc has the following settings: Number of Taps (*n*) = 16, Data Width(*w*) = 8, Coeff Width = 8, Output Width = 20, Signed, Single-Cycle, Real, Symmetric Coeff, Loadable Coeff (8 Coeffs).

# Supplied Netlist Configurations

The Ordering Part Number (OPN) for the Lattice Serial FIR Filter IP Core is FIR-SER-XP-N1. Table 5 lists the Lattice-specific netlists that are available in the Evaluation Package, which can be downloaded from the Lattice web site at www.latticesemi.com.

You can use the IPexpress software tool to help generate new configurations of this IP core. IPexpress is the Lattice IP configuration utility, and is included as a standard feature of the ispLEVER design tools. Details regarding the usage of IPexpress can be found in the IPexpress and ispLEVER help system. For more information on the ispLEVER design tools, visit the Lattice web site at: www.latticesemi.com/software.

***Table 5. Parameter Values for Evaluation Configuration(s)***

| Parameter File Name | Input Data Width (Bits) | Number of Taps | FIR Type | Symmetry | Arithmetic Type | Data Type | Output Data Width (Full Data Width)[2] |
|---|---|---|---|---|---|---|---|
| fir_ser_xp_1_002.lpc[1] | 8 | 16 | Single cycle | Symmetric | Signed | Real | Full (20) |

1. The latency for the fir_ser_xp_1_002 configuration is (6 + Output_Full_Width + 1) or 27.
2. The Output Data Width is the same as the Full Data Width.

# X-ON Electronics

Largest Supplier of Electrical and Electronic Components

***Click to view similar products for*** Development Software ***category:***

***Click to view products by*** Lattice ***manufacturer:***

Other Similar products are found below :

RAPPID-560XBSW RAPPID-567XFSW DG-ACC-NET-CD SRP004001-01 SW006021-1NH SW163052 SYSWINEV21 Core429-SA SW500006-HPA CWP-BASIC-FL W128E13 CWP-PRO-FL SYSMACSE210L SYSMACSE203L AD-CCES-NODE-1 NT-ZJCAT1-EV4 CWA-BASIC-FL RAPPID-567XKSW CWA-STANDARD-R SW89CN0-ZCC CWA-LS-DVLPR-NL VDSP-21XX-PCFLOAT RAPPID-563XMSW IPS-EMBEDDED SWR-DRD-L-01 SDAWIR-4532-01 SYSMAC-SE201L MPROG-PRO535E AFLCF-08-LX-CE060-R21 WS02-CFSC1-EV3-UP SYSMAC-STUDIO-EIPCPLR LIB-PL-PC-N-1YR-DISKID SYSMACSE2XXL LS1043A-SWSP-PRM 1120270005 1120270006 MIKROBASIC PRO FOR FT90X (USB DONGLE) MIKROC PRO FOR AVR (USB DONGLE LICENSE) MIKROC PRO FOR FT90X (USB DONGLE) MIKROBASIC PRO FOR AVR (USB DONGLE LICEN MIKROBASIC PRO FOR FT90X MIKROC PRO FOR DSPIC30/33 (USB DONGLE LI MIKROC PRO FOR FT90X MIKROC PRO FOR PIC32 (USB DONGLE LICENSE 52202-588 MIKROPASCAL PRO FOR ARM (USB DONGLE LICE MIKROPASCAL PRO FOR FT90X MIKROPASCAL PRO FOR FT90X (USB DONGLE) MIKROPASCAL PRO FOR PIC32 (USB DONGLE LI SW006021-2H