XAPP1236 (v2.0) December 15, 2016

Multi-Channel Fractional Sample Rate Conversion Filter Design Using Vivado High-Level Synthesis

Author: Matt Ruan

Summary

This application note focuses on the design of a multi-channel fractional sample rate conversion (SRC) filter using the Vivado High-Level Synthesis (HLS) tool, which takes the source code in C++ programming language and generates highly efficient synthesizable Verilog or VHDL code for FPGA. When there is a need to change filter parameters, e.g., the number of channels, the number of filter taps, or sample rate conversion ratios, only simple modification to the C++ header file is needed. The example SRC filter has a generic architecture from which filters of other types can be easily obtained by modifying the C++ source code.

Introduction

Sample rate conversion (SRC) filters are widely employed in digital signal processing systems which need to handle multiple data rates. For instance, on a music compact disk (CD) 44.1K sound samples are recorded every second. However, the sound track of digital video disk (DVD) needs to play back 48K samples per second. Sample rate conversion from 44.1Ksps to 48Ksps must be performed by the video editing tool before the CD music can be added to a DVD sound track.

Another important application of SRC filters is in the area of data compression to minimize the number of samples required for the representation of certain waveform. In the case of 3GPP Long-Term Evolution (LTE) systems [Ref 1], the nominal sample rate of an LTE 20MHz signal is 30.72Msps, while the useful signal bandwidth is only 18.015MHz. It means that up to 30%-40% transportation bandwidth between the baseband channel card and the remote radio unit can be saved by reducing the sample rate. As a result, the operator will be able to serve 30%-40% more subscribers using existing optical fibers.

The wide application of SRC filters calls for a method to design flexible, scalable, and resource-efficient filters that can run on programmable logic devices. This application note will explain how Xilinx Vivado Design Suite can nicely address this need by allowing the designers to describe the filters in C++ programing language, which is then synthesized into hardware description language (HDL) and implemented on FPGAs. The Vivado HLS (HLS) tool [Ref 2], which is integrated in the Vivado Design Suite, can automatically generate the HDL with optimized pipeline architecture according to the given constraints, and create test benches to ensure the behaviors of HDL and C++ code are identical. In many cases, the Vivado HLS synthesized code has similar performance to that of a hand coded HDL design performed by an experienced logic engineer. When there is a need to change the clock rate, the target FPGA part number, or filter parameters like the number of taps, fractional conversion ratio, etc., only slight modification to the C++ header file is required. The C++ filter design literately becomes an IP that can be easily customized for new applications.

Theory of Operation

This section explains the basic theory of sample rate conversion and how SRC filters work. Denote the ratio between the input and output data sample rates of the filter as P/Q where *P* and *Q* are integers. The sample rate conversion operation consists of the following steps:

1. Interpolation: Inserting $(Q\text{-}1)$ zeroes between every two input samples.
2. Low-pass Filtering: Using a low-pass filter to eliminate the aliasing.
3. Decimation: Outputting a sample every *P* low-pass filtered samples.

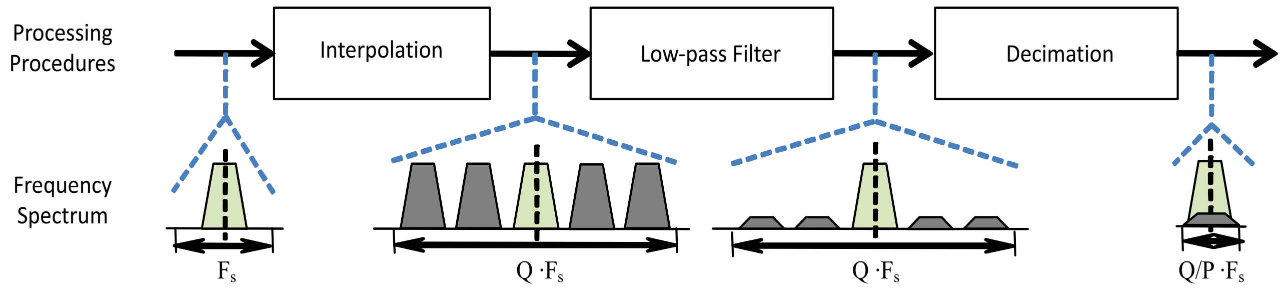


Figure 1: **Conceptual Procedures of Fractional Sample Rate Conversion**

Figure 1 shows the signal spectrum after each step. The wanted signal and its undesirable aliasing are shown in light green and dark grey, respectively. The net effect of the SRC filtering is that the total spectrum of the signal is compressed or expanded to $(Q/P \cdot F_S)$, and the original signal is slightly contaminated by its aliasing, which needs to be controlled within an acceptable level by a carefully designed low-pass filter at the second step.

To have a closer look at the operations, the computation procedures of a SRC filter are illustrated in Figure 2. The first row of boxes represents the input data stream with inserted zeros that are colored in white. The following rows of boxes represent the filter coefficients sliding from left to right to realize convolutional multiplication. Each row corresponds to one filter coefficient position from which one low-pass filtered output can be computed. For the rows with dashed lines "--" in the front, the low-pass filtered results should be decimated and only those with "y_n" in the front are selected as the SRC filter's final output.

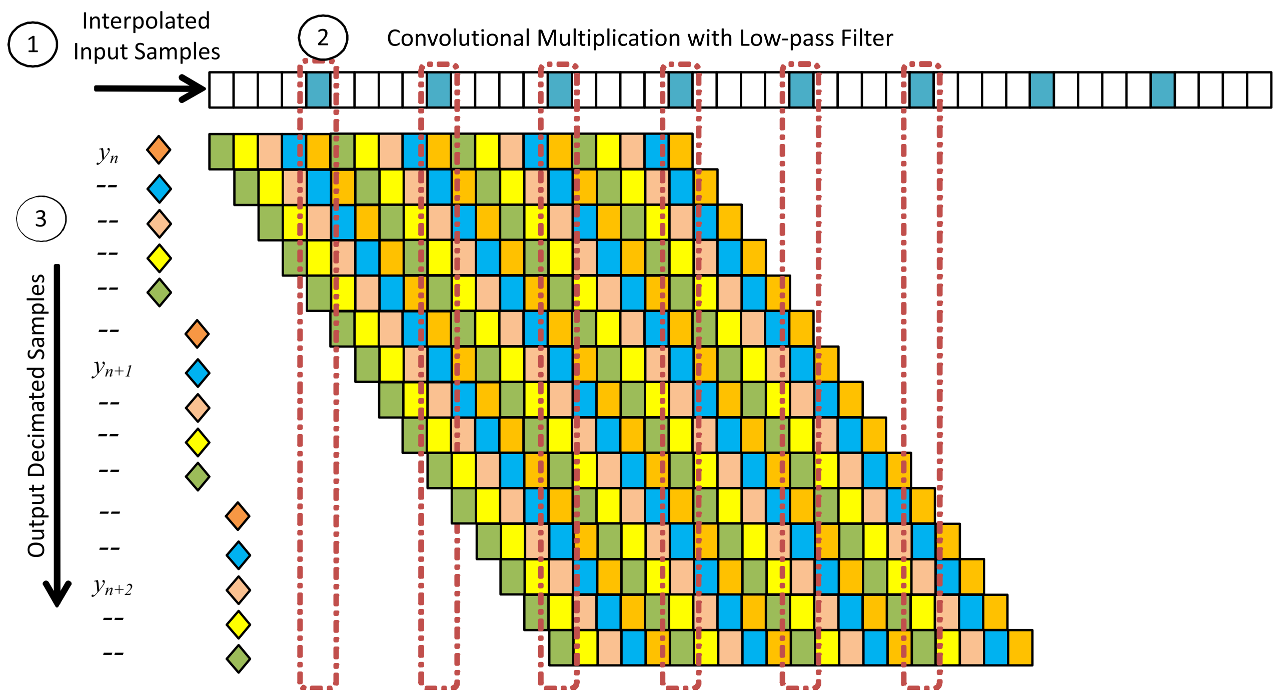

1
Interpolated Input Samples
2
Convolutional Multiplication with Low-pass Filter
3
Output Decimated Samples
y_n
y_{n+1}
y_{n+2}

Figure 2: **Computation Procedures of a Fractional Sample Rate Conversion Filter**

In the example shown in Figure 2, the SRC filter has 20 taps and the fractional ratio is 5/6. To compute every low-pass filter output, only the most recent 4 input samples are needed. Moreover, the coefficients are dot multiplied with the non-zero samples and can be divided into 5 sets such that the coefficients of different sets are never used to compute any single output sample. These observations can be generalized into the following facts:

- Since only one out of P filtered samples are outputted, there is no need to compute the other (P-1) low-pass filtered samples.
- Denote the number of filter taps as L. The filter coefficients can be evenly divided into Q phases, each of which contains at most $\mathrm{ceil}(L/Q)$ coefficients that are needed for the computation of one low-pass filter output.
- The most recent $\mathrm{ceil}(L/Q)$ input samples need to be kept in the registers for the dot multiplication with one phase of coefficients when computing one low-pass filter output.

These facts lead to a much simplified SRC filter architecture as shown in Figure 3. The filter coefficients are saved in $\mathrm{ceil}(L/Q)$ ROMs, and only those needed for the dot multiplication with the input samples are read out to compute one filter output. The architecture shown in Figure 3 is similar to that of a conventional FIR filter but the controller needs to be redesigned to generate coefficient ROM addresses and manage the shift registers appropriately.

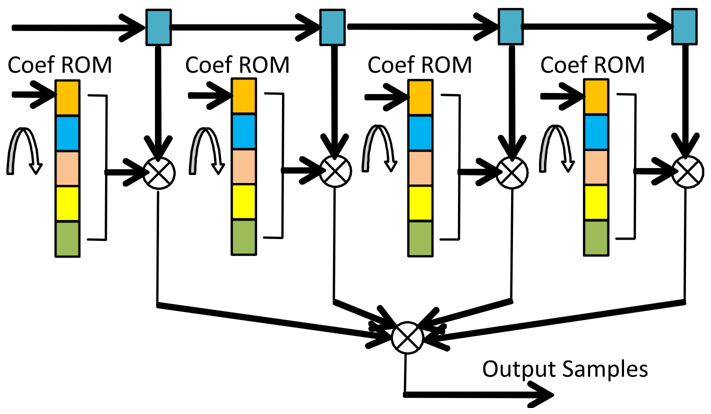


Figure 3: **Hardware Efficient Fractional Sample Rate Conversion Filter Architecture**

Multi-Channel SRC Filter Architecture

Wideband multiple-antenna LTE systems need to process multiple streams of data, each of which can have independently selected data rate and sample rate conversion ratio. This application note gives an example of designing such a multi-channel multi-rate SRC filter on Xilinx 7 Series FPGAs that typically run at 400MHz or higher. The requirements of the fractional resample filters are summarized in the following table.

Table 1: **Multi-Channel Multi-Rate SRC Requirements**

	Requirements	Notes
Number of Channels	16	16 channels correspond to 8 LTE 20MHz carriers in 160MHz signal bandwidth.
Input Sample Rate	Up to 30.72Msps	Nominal sample rate of LTE 20MHz signal
Output Sample Rate	Up to 30.72Msps	Nominal sample rate of LTE 20MHz signal
Sample Conversion Ratios	Bypass, 3/4, 5/8, 5/6, 4/3, 8/5, 6/5	3/4, 5/8, 5/6 are decimation ratios, 4/3, 8/5, 6/5 interpolation ratios. One filter supports all ratios.

For such a multi-channel FIR, it is recommended [Ref 3] to use the hardware efficient systolic multi-MAC architecture as shown in Figure 4. The high efficiency is achieved by storing the input data stream in shift registers which can be implemented in look-up-tables (LUT), abundant in FPGAs. One level pipeline is inserted to the output of each MAC so that the multiplication and summation operations for each tap can be realized in one single DSP48E unit. This saves a great amount of FPGA resources and improves timing performance.

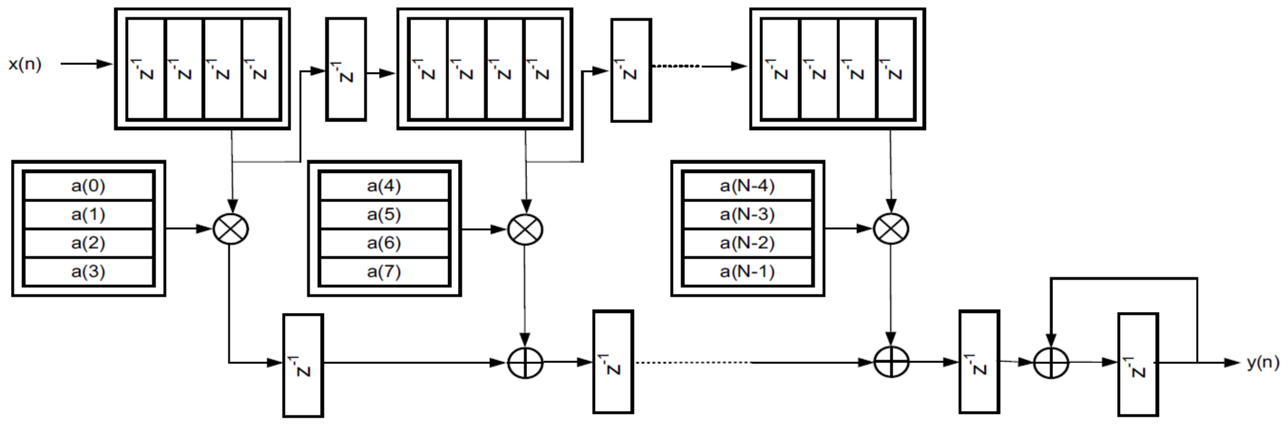


Figure 4: **Recommended Systolic Multi-MAC Architecture (Figure 3-10 of [Ref 3])**

The generic systolic multi-MAC architecture can be modified for various applications including SRC filters, since, as shown in Figure 3, the main calculation module of a SRC filter is still a systolic MAC block. The tricky part of SRC filters is always the control logic for coefficient ROM address generation and data shift register management.

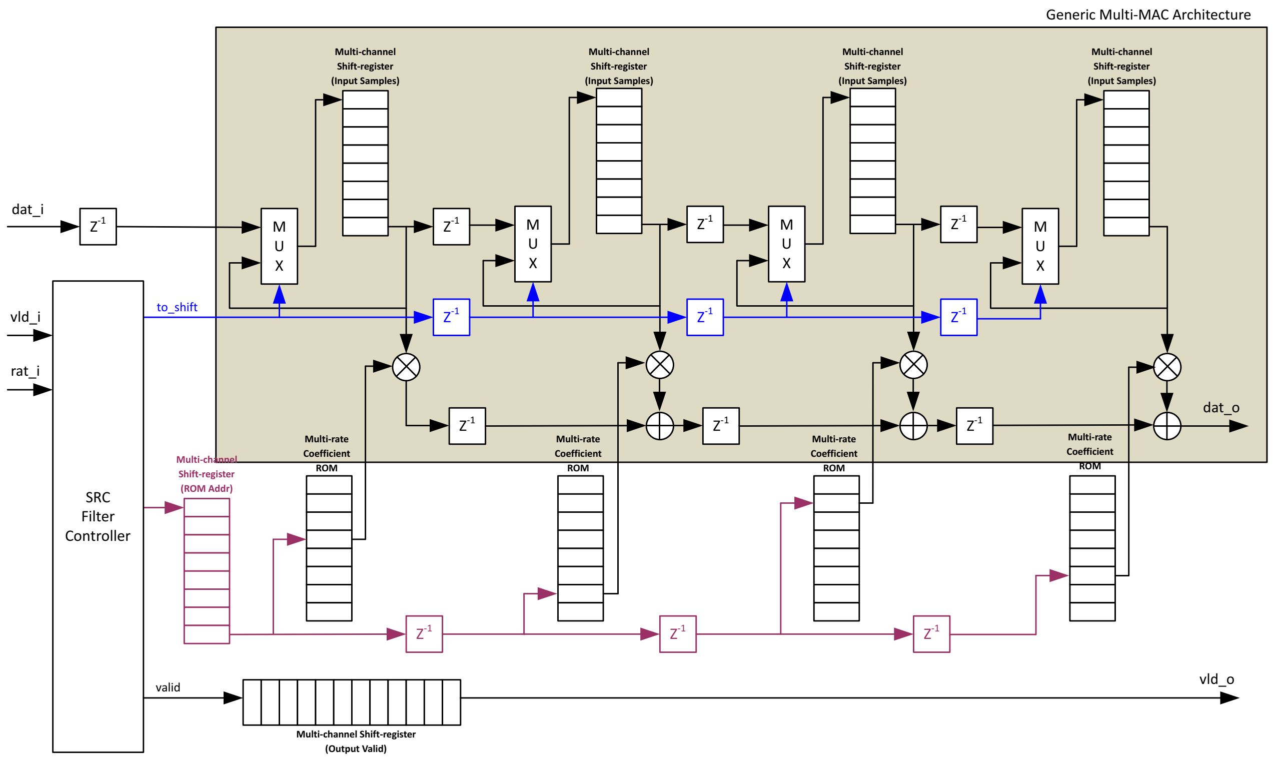

Generic Multi-MAC Architecture
Multi-channel Shift-register (Input Samples)
dat_i
vld_i
rat_i
to_shift
MUX
SRC Filter Controller
Multi-channel Shift-register (ROM Addr)
Multi-rate Coefficient ROM
dat_o
valid
vld_o
Multi-channel Shift-register (Output Valid)

Figure 5: **Simplified Block Diagram of an 8-Channel 4-Tap Multi-Rate SRC Filter**

Figure 5 shows a block diagram of a multi-channel multi-rate SRC filter. The grey shaded area shows a generic multi-MAC calculation unit with shift registers storing the data of independent channels that are processed in a round-robin time-multiplexed manner. When valid_i is asserted, the shift register takes the new input data and shifts out the oldest one. When no new input data for the current channel is available, the oldest data of the shift register is written back such that the state of the channel remains unchanged. This enables the SRC filter to deal with various sample rates for each channel not exceeding an upper limit given by F_c/N_{ch}, where F_c is the main clock frequency and N_{ch} is the number of channels.

The organization of the coefficient ROM is detailed in Figure 6 such that each address corresponds to one set of coefficients for the given sample conversion ratio. Every coefficient ROM contains all the coefficients of one filter tap. Due to the pipelined architecture, only the first ROM address needs to be computed and it is then delayed for the other filter taps.

Addr	ROM 0	ROM 1	ROM 2	ROM M
0	Rat1_Coef_0	Rat1_Coef_3	Rat1_Coef_6	Rat1_Coef_L-2
1	Rat1_Coef_1	Rat1_Coef_4	Rat1_Coef_7	Rat1_Coef_L-1
2	Rat1_Coef_2	Rat1_Coef_5	Rat1_Coef_8	Rat1_Coef_L
3	Rat2_Coef_0	Rat2_Coef_5	Rat2_Coef_10	Rat2_Coef_L-4
4	Rat2_Coef_1	Rat2_Coef_6	Rat2_Coef_11	Rat2_Coef_L-3
5	Rat2_Coef_2	Rat2_Coef_7	Rat2_Coef_12	Rat2_Coef_L-2
6	Rat2_Coef_3	Rat2_Coef_8	Rat2_Coef_13	Rat2_Coef_L-1
7	Rat2_Coef_4	Rat2_Coef_9	Rat2_Coef_14	Rat2_Coef_L

Figure 6: **Diagram of Coefficient ROM Data Structure**

It is shown in Figure 5 that once the input data, to_shift flag, valid flag, and ROM address are computed by the SRC filter controller, the calculation is straightforward and can be performed by simple arithmetic blocks. However, it is not trivial to design the logic for the controller to accommodate various sample rates and conversion ratios for multiple channels. High-level programing languages like C++ are much more convenient for the description of such controllers.

Implementation Details

Figure 7 shows the data flow of C++ functions. The top-level function is MultiSRC, and the four sub-functions, srcCtrl, srcMac, srcRnd, and srcCoef, correspond to the controller, MAC calculation unit, rounding, and coefficient ROMs of Figure 5. The shift register is realized by the HLS data structure ap_shift_reg [Ref 2]. The C++ source code of the srcMac and srcCtrl functions is described in the following sections, as well as the Vivado HLS directives used to obtain the desired synthesis results.

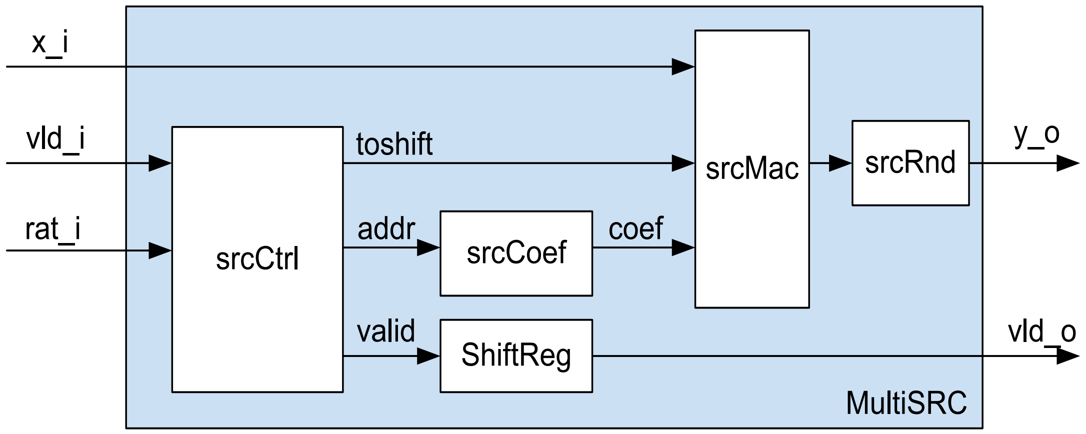


Figure 7: **C++ Function Data Flow**

srcMac.cpp

This function implements a generic multi-channel systolic MAC unit. Figure 8 shows the MAC calculation block for one filter tap supporting multiple channels. A number of such MAC modules can be instantiated to generate a multi-tap filter. As it is assumed that the filter coefficients are available to the MAC as inputs, the generic MAC architecture can be used for various types of filters including but not limited to SRC filters

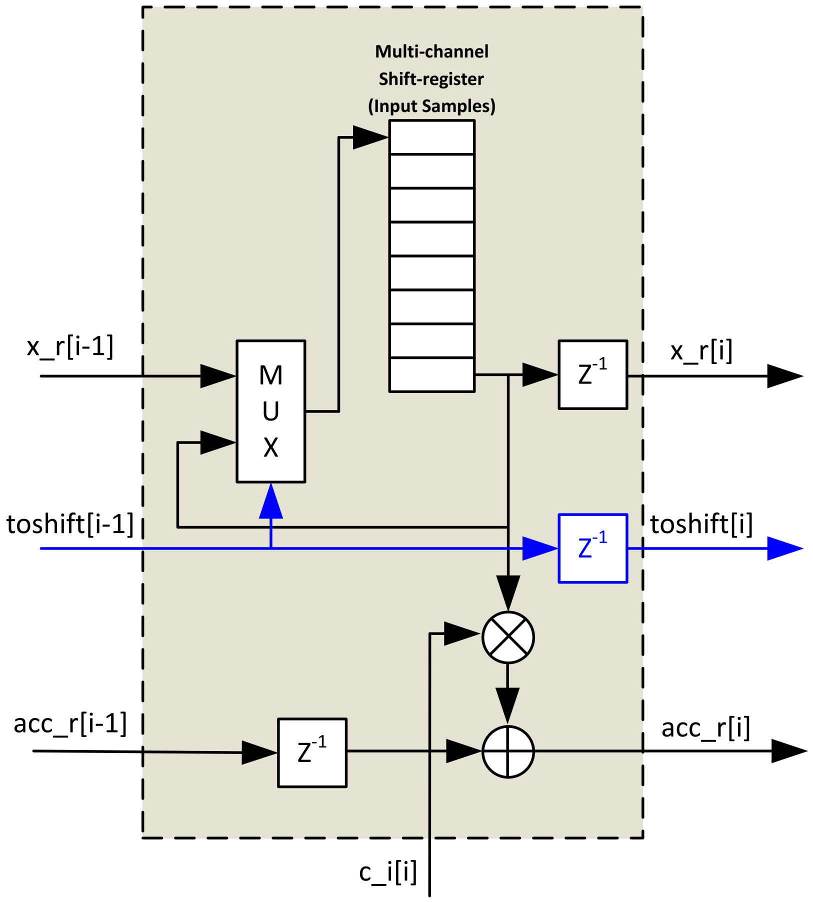


Figure 8: **MAC Block for One Filter Tap**

The multi-channel MAC can be described by the following C++ code:

```
// loop for all the taps
MULTMACLOOP: for(int i=NumTap-1; i>=1; i--){

  // multiplier with registered output
  mult_t mul_temp = x_r[i] * c_i[i];

  // acc_o is expected to have one clock delay here
  acc_r[i] = mul_temp + acc_r[i-1];

  // read out data from shift register
  x_r[i] = shift_reg[i].read(ChMux-1);

  // mux to determine whether shift or not
  // if not shift, then write back the data read from shift register
  x_mux = toshift_r[i-1]? x_r[i-1] : x_r[i];

  // shift in and out
  shift_reg[i].shift(x_mux);
  toshift_r[i]=toshift_r[i-1];
}

// multiply for the first shift reg
acc_r[0] = x_r[0] * c_i[0];

// read out data for next
x_r[0] = shift_reg[0].read(ChMux-1);


// mux to determine whether shift or not
// if not shift, then write back the data read from shift register
toshift_r[0]=toshift_i;
x_mux = toshift_r[0]? x_i : x_r[0];
shift_reg[0].shift(x_mux);
```

srcCtrl.cpp

This function implements the control logic of the multi-channel multi-rate SRC filter. A block diagram is shown in Figure 9. When a channel is served, its status is read out and processed according to the input parameters like sample rate conversion ratio, data valid flag, etc. Upon completion of a new phase and coefficient ROM address calculation, the status memory of the channel needs to be updated before the next access.

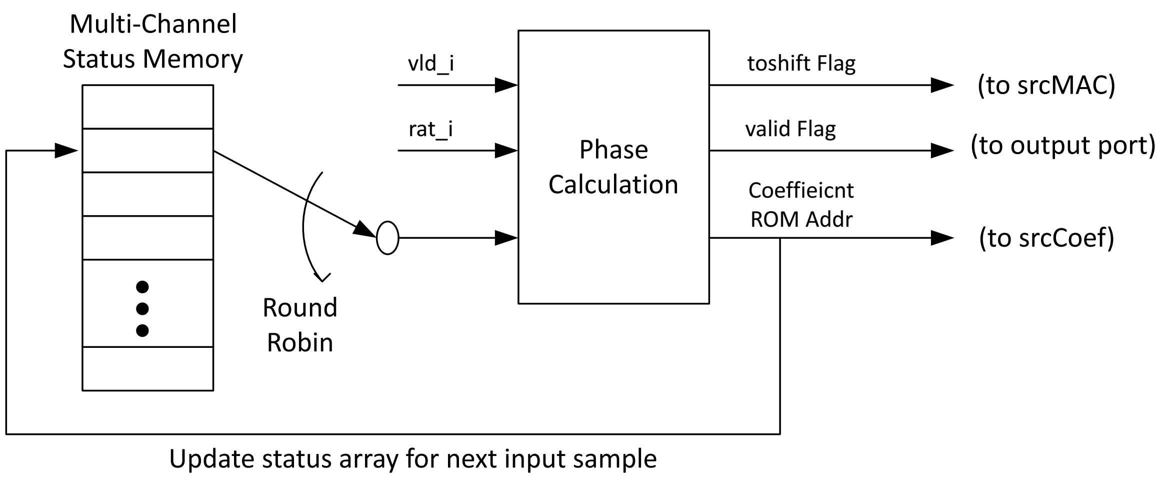
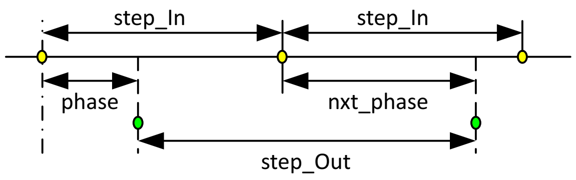
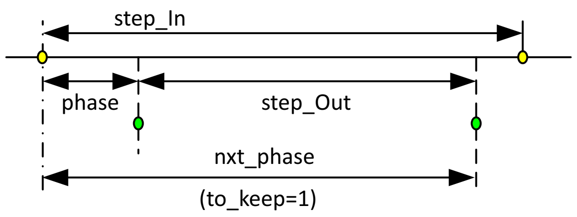


Figure 9: **Multiple Channels Served by One Phase Calculation Engine in Time-Multiplexed Manner**

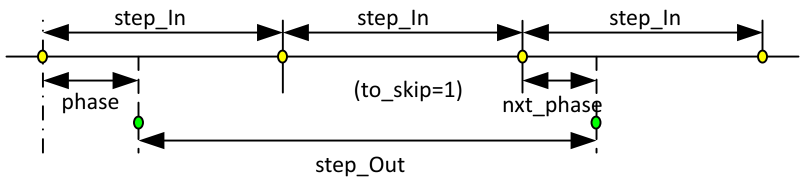
Figure 10 illustrates the phase calculation process. The input sample period is defined by step_in, and the output one by step_out. The yellow dots represent input samples, green ones are outputs. The phases and steps are defined relative to the position of the current input sample. Figure 10(a) shows a normal phase update process where the next output occurs between the next two inputs. In case where the next output still occurs before the next input, as illustrated in Figure 10(b), the to_keep flag needs to be asserted to force the shift registers of the MAC unit to keep their current status. In case of a decimation the next output happens after the next two input samples, as shown in Figure 10(c), and the to_skip flag needs to be asserted to indicate that no output needs to be computed for the next input sample.



(a) Normal Condition, to_keep=false, to_skip=false



(b) When (phase+step_Out<step_In), to_keep=true, to_skip=false



(c) When (phase+step_Out>2 step_In), to_keep=false, to_skip=true

Figure 10: : **Phase Calculation and Flag Setting Conditions**

The above control logic can be described by the following C code.

```
// update phase and flag for next input
    if (vld_o){
    phase+=step_out;
    if(phase<step_in){ // it means that no new data is required for next output
      tokeep=true;
      isskip=false;
    }else{ // new data is needed
      tokeep=false;
      isskip=( (phase>>1) >=step_in);
      phase-=step_in;
    }
  }else if (vld_i){ // a new input is available, but it is to be skipped
    tokeep=false;
    isskip=( (phase>>1) >=step_in);
    phase-=step_in;
  }
```

Directives

When converting the C++ code into HDL, Vivado HLS needs some side information to describe parameters like the number of clock cycles available to complete a loop, whether the module can accept new inputs before old ones are all processed, etc. These directives are an integral part of the design, and specify how the C++ code is to be synthesized into HDL with expected behavior. When porting an existing design to a new application, quite often only slight modifications of the directives are needed, without touching the C++ code.

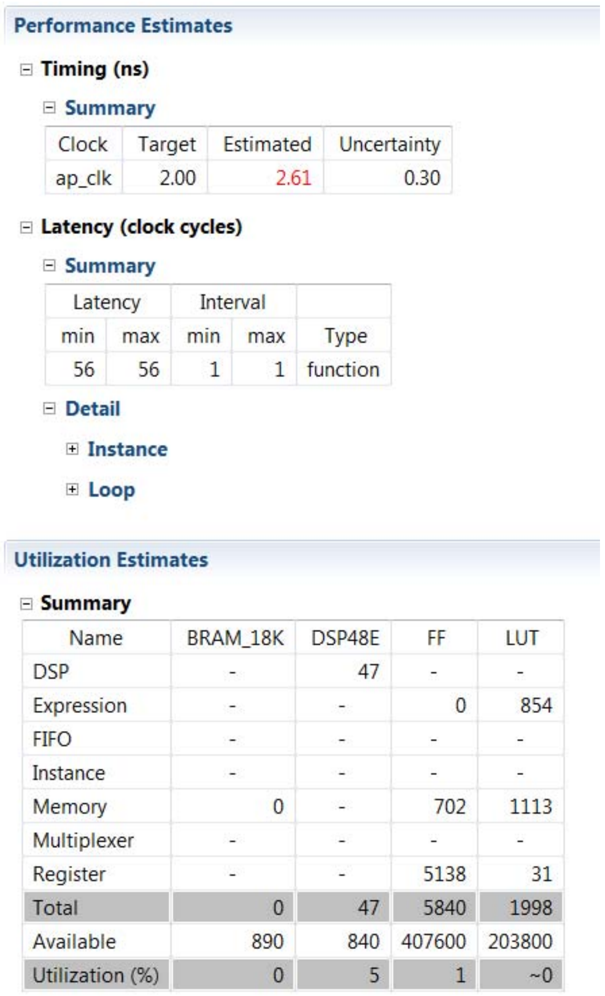
For the SRC filter, the following directives are used to realize the desirable behavior.

```
# IO interface
set_directive_interface -mode ap_none multiSRC x_i
set_directive_interface -mode ap_none multiSRC vld_i
set_directive_interface -mode ap_none multiSRC rat_i
set_directive_interface -mode ap_none multiSRC vld_o
set_directive_interface -mode ap_none multiSRC y_o

# The function has a pipelined architecture and accepts new inputs every clock cycle
set_directive_pipeline -II 1  multiSRC

# Reset the counters to ensure a deterministic phase
set_directive_reset srcCtrl cnt_r
set_directive_reset srcCtrl cnt_w

# Inform the tool of the variables not inter dependent
set_directive_dependence -variable vec_phase  -type inter -dependent false srcCtrl
set_directive_dependence -variable vec_tokeep -type inter -dependent false srcCtrl
set_directive_dependence -variable vec_isskip -type inter -dependent false srcCtrl

# Inline the functions for highest performance
set_directive_inline -region -recursive multiSRC

# Partition the arrays to meet the bandwidth requirement
set_directive_array_partition -dim 1 srcCoef coef_rom
set_directive_array_partition -type complete srcCoef addr
set_directive_array_partition -type complete srcMac c_i
set_directive_array_partition -type complete srcMac x_r
set_directive_array_partition -type complete srcMac acc_r
set_directive_array_partition -type complete srcMac toshift_r

# Print scheduling information for debugging
config_schedule -verbose
```

Synthesis Results

Xilinx Vivado HLS analyzes all the design files and then automatically selects the appropriate hardware architecture to meet the target clock speed and data throughput specified by the designer in the form of synthesis directives. Once the C compilation is completed, the basic information about the synthesized HDL can be reviewed to check against the targets. Figure 11 shows the SRC filter synthesis report generated by Vivado HLS.

As expected, the 47-tap multi-channel SRC filter uses 47 DSP48Es to achieve the throughput of 1 clock cycle per input data. The design is estimated to run at 1/2.61ns=383.1MHz, which is lower than the 500MHz (2ns) target. The clock speed estimate on the C synthesis report is not fully accurate because the timing of the design is not fixed until FPGA place and route is completed. According to the report, the filter coefficient ROMs and base address tables are implemented in distributed memory.


Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	2.00	2.61	0.30

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
56	56	1	1	function

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	47	-	-
Expression	-	-	0	854
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	702	1113
Multiplexer	-	-	-	-
Register	-	-	5138	31
Total	0	47	5840	1998
Available	890	840	407600	203800
Utilization (%)	0	5	1	~0

Figure 11: **C Synthesis Report**

Verification Results

Within the Vivado HLS design flow, functional verification consists of two steps.

The first step is C++ functional verification to validate the C++ code. The testbench needs to be manually coded in C++. However, with the rich file I/O functions provided by the C++ library, it is quite straightforward to code up a testbench based on pre-stored input and output test vectors. For the multi-channel SRC, the test bench reads the input test vector, calls the C++ function to process the data, and then compares the C++ function output with the pre-stored golden output test vector. One test case featuring 16 independent channels with different sample rates and conversion ratios has been constructed for verification purpose (see Table 2).

Table 2: **Constructed SRC Filter Test Case for Verification Purpose**

Channel ID	Carrier Sample Rate (Msps)	Conversion Ratio
0	30.72	5/8
1	23.04	5/6
2	15.36	3/4

Table 2: **Constructed SRC Filter Test Case for Verification Purpose**

Channel ID	Carrier Sample Rate (Msps)	Conversion Ratio
3	5.76	Bypass
4	15.36	4/3
5	7.68	8/5
6	11.52	6/5
7	19.2	Bypass
8	30.72	3/4
9	11.52	4/3
10	5.76	6/5
11	15.36	8/5
12	23.04	4/3
13	19.2	8/5
14	11.52	5/6
15	19.2	5/8

Once the C++ behavior has been verified, and the C++ functions are synthesized into HDL, Vivado HLS can automatically generate an HDL testbench according to the C++ test code. This step is referred to as "C/RTL Co-simulation", and ensures the HDL behavior matches the C++ functionality. Vivado HLS supports various simulators and HDL code formats for C/RTL co-simulation, as shown in Figure 12.

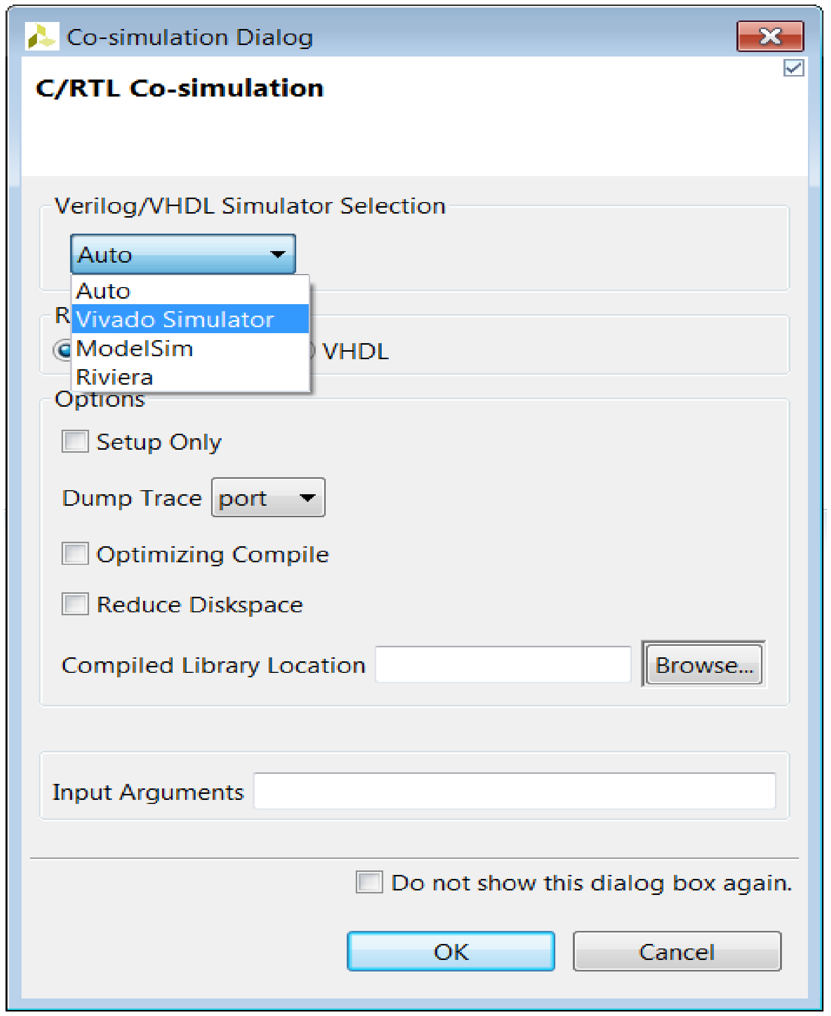


Figure 12: **C and HDL Co-Simulation Wizard**

The outputs of the HDL design are compared to that of the C reference model to ensure the functionality is correct. At the end of the simulation, the tool prints the post checking results, which look like the following:

```
......

## save_wave_config multiSRC.wcfg
## run all
////////////////////////////////////////////////////////////////////////////////////
// Inter-Transaction Progress: Completed Transaction / Total Transaction
// Intra-Transaction Progress: Measured Latency / Latency Estimation * 100%
////////////////////////////////////////////////////////////////////////////////////
// RTL Simulation : 0 / 16753 [0.00%] @ "105000"
// RTL Simulation : 16753 / 16753 [0.00%] @ "33725000"
////////////////////////////////////////////////////////////////////////////////////
$finish called at time : 33733 ns : File
"C:/xapp1236-multi-src-hls/ProjMultiSRC/SolutionX/sim/verilog/multiSRC.autotb.v" Line
445
```

```
run: Time(s): cpu = 00:00:00 ; elapsed = 00:00:08 . Memory (MB): peak = 59.340 ; gain =
0.281
## quit
INFO: [Common 17-206] Exiting xsim at Wed Jun 01 14:38:17 2016...
[0]         Total 625 Output Samples        0 Errors.
[1]         Total 625 Output Samples        0 Errors.
[2]         Total 375 Output Samples        0 Errors.
[3]         Total 188 Output Samples        0 Errors.
[4]         Total 667 Output Samples        0 Errors.
[5]         Total 400 Output Samples        0 Errors.
[6]         Total 450 Output Samples        0 Errors.
[7]         Total 625 Output Samples        0 Errors.
[8]         Total 750 Output Samples        0 Errors.
[9]         Total 500 Output Samples        0 Errors.
[10]        Total 226 Output Samples        0 Errors.
[11]        Total 800 Output Samples        0 Errors.
[12]        Total 1000 Output Samples       0 Errors.
[13]        Total 1000 Output Samples       0 Errors.
[14]        Total 313 Output Samples        0 Errors.
[15]        Total 391 Output Samples        0 Errors.
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
```

There is also an option to dump traces for manual debugging on the waveforms. Below are some simulation waveforms for the 16-channel test case with various sample rates and conversion ratios. The simulator used is Vivado Simulator 2016.1. From these waveforms shown in Figure 13 it can be checked that the latency of the multi-channel SRC is 238ns/2ns = 119 clock cycles, which matches the expectation that 47 (taps) + 16 (channels) + 56 (filter delay reported in synthesis report) = 119.

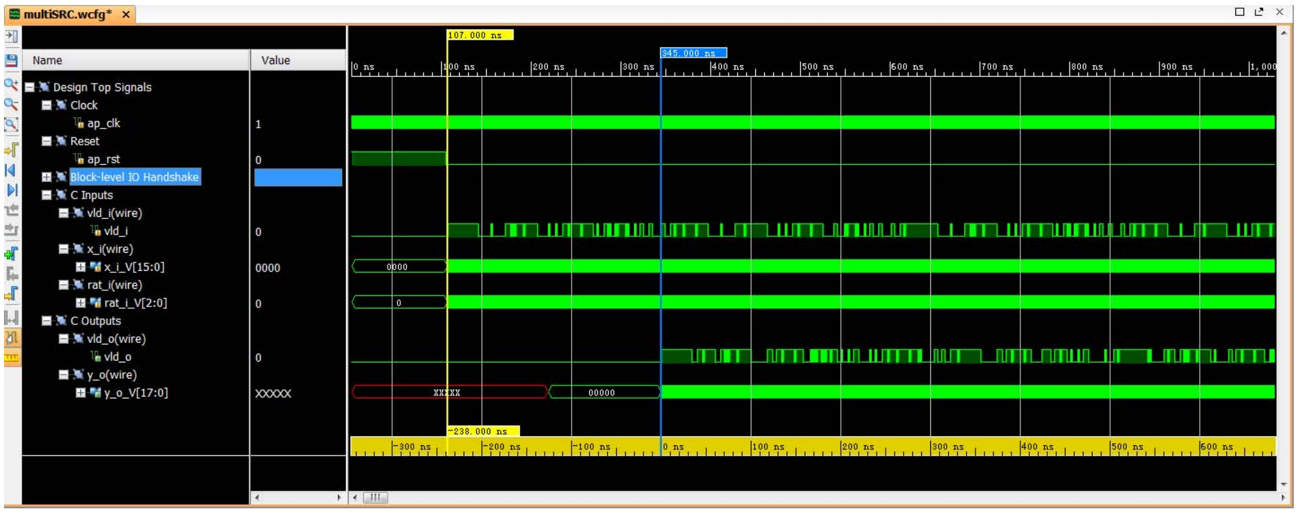


Figure 13: **C and HDL Co-Simulation Waveform**

Implementation Results

Xilinx Vivado HLS not only generates the HDL code of the C++ function, but also provides a number of options to package the HDL into an IP ready for integration into a larger design using the Vivado Design Suite, e.g., System Generator, EDK, and IP Integrator. For illustration purposes, IP Catalog has been used for the example reference design.

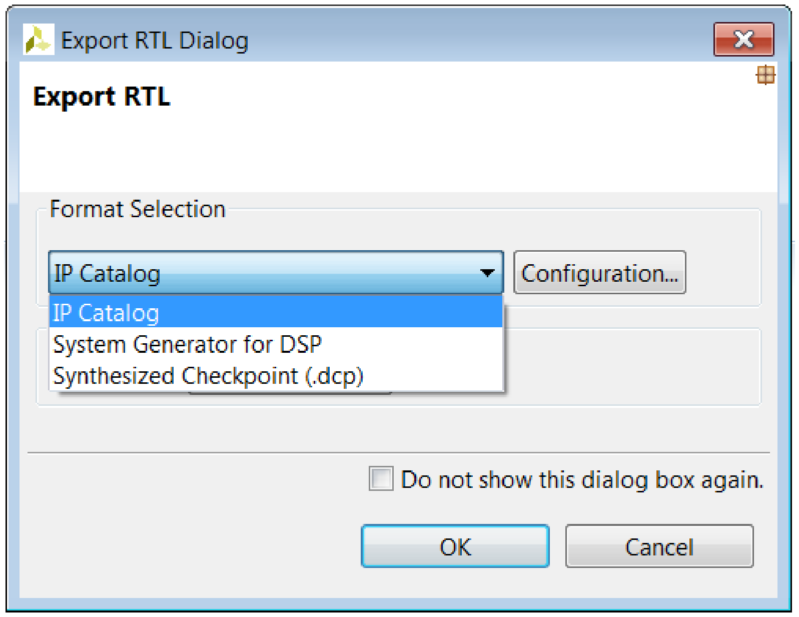


Figure 14: **HDL Export Dialog Box**

The Vivado HLS tool automatically creates a Vivado project and synthesizes all the HDL code to validate the implementation performance. Below is the final implementation report for the SRC filter:

```
Implementation tool: Xilinx Vivado v.2016.1
Project:             ProjMultiSRC
Solution:            SolutionX
Device target:       xc7k325tffg900-2
Report date:         Wed Jun 01 14:43:30 +0800 2016

#=== Resource usage ===
SLICE:         1440
LUT:           1905
FF:            5150
DSP:             47
BRAM:             0
SRL:            392
#=== Final timing ===
CP required:    2.000
CP achieved:    1.967
Timing met
```

Two benchmark filters, based on Xilinx FIR Complier v7.2, have been built to evaluate the quality of the synthesis results. The first benchmark is a multi-channel polyphase FIR filter supporting 16 independent channels, 32 phases, and 47 taps for each phase. The MAC architecture of such a polyphase filter is similar to that of the SRC, while the control logic is much simpler because it does not support changing sampling rates in real time as the proposed SRC filter does. As shown in Table 3, the resource of the SRC filter implemented in HLS is higher than that of the FIR filter in FF but lower in LUT and DSP, which means that the Vivado HLS synthesis result is reasonably efficient in FPGA resource.

The second benchmark filter comprises 16 instances of a single-path polyphase FIR which takes 16 clock cycles to compute one output sample. The filter coefficients are forced to be stored into distributed memories for fair comparison with the other two solutions. The advantage of the multi-channel pipelined architecture is clearly illustrated in Table 3 where the SRC implemented in Vivado HLS is much more efficient in LUT. This is simply because the multi-channel architecture only needs one set of coefficient ROMs, while the single-channel architecture requires seven additional sets. Similarly, one additional DSP is required per filter for phase accumulation, hence the lower efficiency compared to a fully parallel architecture.

Table 3: **Benchmark Filter Synthesis Results**

	LUT	FF	DSP	BRAM18
SRC Implemented in Vivado HLS	1905	5150	47	0
Multi-channel Polyphase FIR	2344	1817	48	0
16 Instances of Single-channel Polyphase FIR	8736	3568	80	0

Conclusion

This application note demonstrates a method of building multi-carrier multi-channel SRC filters using the Vivado HLS tool chain which takes C++ code on input and generates HDL code synthesizable on FPGAs. The C++ source code is easy to maintain and scalable to various FPGA parts, input and output sample rates, and system clock frequencies. By modifying the control logic coded in C++, it is simple to build other types of filters out of the generic multi-MAC architecture.

Reference Design

You can download the Reference Design Files for this application note from the Xilinx website.

Table 4 shows the reference design matrix.

Table 4: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Matt Ruan (Xilinx)
Target devices	7K325T, 7Z045, etc.
Source code provided	Yes
Source code format	C++, test vectors, MATLAB models and scripts, and HLS/Vivado synthesize script
Design uses code and IP from existing Xilinx application note and reference designs or third party	No
Simulation	
Functional simulation performed	Yes
Timing simulation performed	No
Test bench used for functional and timing simulations	Yes
Test bench format	C++ and Verilog
Simulator software/version used	Vivado Simulator 2016.1
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/versions used	Vivado HLS 2016.1
Implementation software tools/versions used	Vivado Design Suite 2016.1
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	Xilinx ZC706 EVB

Design File Hierarchy

The directory structure underneath the top-level folder is described below.

```
\m
| This folder contains MATLAB reference model and test vector generation scripts.
|
\src
|  This folder contains C++ design files and header files.
|
\tb
|  This folder contains the C++ test bench.
|
\tv
|  This folder contains the input and output golden test vectors for
|  verification purpose.
|
\boardtest
|
+----- \src
|       This folder contains Vivado project files for onboard testing.
```

Installation and Operating Instructions

You can download the reference design onto a ZC706 evaluation board [Ref 4], which contains a Zynq-7000 AP SoC device. The download configures the Zynq-7000 AP SoC to run the example design.

To configure the Zynq-7000 AP SoC to run the example design:

1. Install Xilinx Vivado Design Suite 2016.1 or later.
2. Unzip the design files into a clean directory.
3. In the Vivado HLS command line window:
   a. `cd` to the root of the design directory.
   b. Enter `vivado_hls run.tcl`.
   c. Check the synthesized design meets expectation.
4. In the Vivado Tcl command window:
   a. `cd` to the `boardtest` directory.
   b. Enter `source boardtest.tcl`.
   c. Check the implementation result meets expectation.
5. Run the reference design on the Zynq-7000 AP SoC.
   a. Download the design onto the ZC706 evaluation board.
   b. Press the middle pushbutton (SW9) to reset the design.

c. Make sure the heartbeat LED on the righthand side is blinking.

d. Press the rightmost pushbutton (SW8) to run the test.

e. Check whether LED2 (the middle LED) is solid on.

f. Repeat steps d-e several of times to confirm the test passes.

References

1. 3GPP TS 36.211, "3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Channels and Modulation (Release 12)", July 2014.
2. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
3. *FIR Compiler v7.2 LogiCORE IP Product Guide* (PG149)
4. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide* (UG954)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/15/2016	2.0	Updated the software version to 2016.1, which achieves 491.52MHz performance with the same C++ design files. Updated reports, output values, resource counts, waveforms, and dialog box displays throughout document to reflect results when implementing the SRC filter in the 2016.1 Vivado release.
06/15/2015	1.0	Initial Xilinx release.

Please Read: Important Legal Notices