



XAPP1299 (v1.0) December 10, 2016

Designing a Digital Up-Converter using Modular C++ Classes in Vivado High Level Synthesis

Author: Alex Paek, Jim Wu

Summary

This application note describes the implementation of digital up-converter (DUC) design using the Vivado® High-Level Synthesis (HLS) tool, which produces synthesizable RTL from C++ source code. It details the methods used, such as HLS optimization techniques and coding styles/guidelines in constructing template classes and functions to build a hierarchical design.

The automatic resource assignment, sharing and scheduling capability of the Vivado HLS tool is instrumental in mapping a complex algorithm into a solution optimized for both performance and resource usage. The proposed design methodology and recommendation can be applied in general to expedite the implementation of an algorithmically complex block such as a software defined radio.

Introduction

The DUC design consists of multi-stage finite impulse rate (FIR) filters, a direct digital synthesizer (DDS) and a mixer. One important aspect of this design is that we want to make a parameterize-able design where filter parameters like coefficients, fixed point precision, and data sample rate can be easily changed, by utilizing template classes and functions.

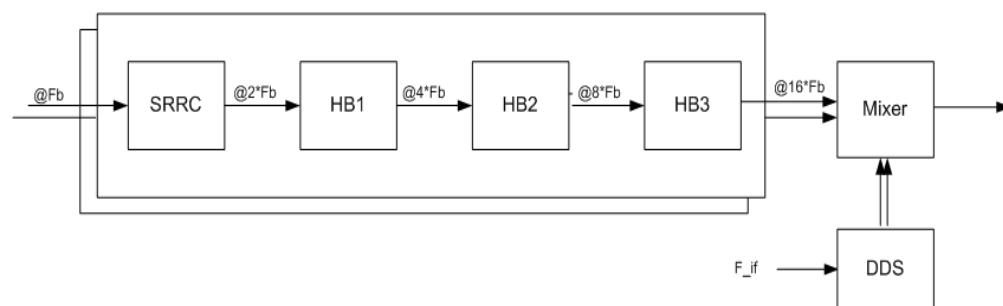


Figure 1: DUC Block Diagram

The DUC specification is as follows:

- The input is a complex data input (I/Q) sample, which can be a QAM symbol, and the output is a continuous stream of real sample.
- The output rate of the DUC (F_o) is set to the processing clock rate (F_{clk}). The relationship from the input symbol rate (F_b) to the output rate is: $F_o = 16 * F_b = F_{clk}$.
- 4 stages of FIR filter, with interpolation ratio of 2 in each stage and the overall interpolation ratio of 16.
- The first stage is a 64 tap square raised root cosine (SRRC) filter, and the next three stage filters are half-band (HB) interpolate by 2 FIR filter. The overall filter response is shown in [Figure 2](#).
- The DDS has the frequency resolution of $F_{clk}/2^{32}$ (for example, 0.07Hz for 300 MHz F_{clk}), and spurious free dynamic range (SFDR) is greater than 120 dB.

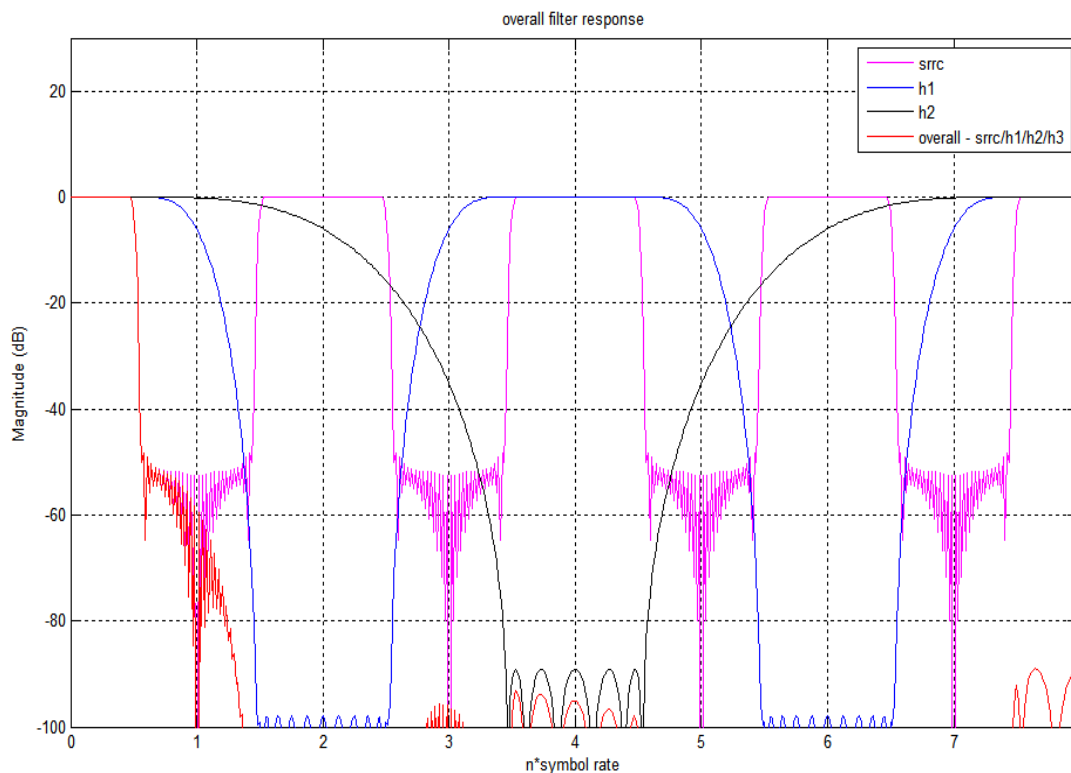


Figure 2: Overall Filter Response

Designing the FIR Block

Since the FIR block is one of the main building blocks in the DUC design, we need to make sure the HLS tool can produce an efficient implementation in terms of resource and throughput. Depending on the data sample rate and processing clock rate, a FIR can have a serial, parallel or semi-parallel structure. Some of the key important questions in designing an efficient FIR implementation are the following:

- Is the number of multiplier accumulators (MACs) used in the design optimum - that is, close to the theoretical number:

$$\text{Minimum number of MACs required} = \frac{\text{OutputDataRate} \times \text{NumberOfTops} \times \text{NumberOfChannels}}{\text{ClockRate}}$$

- There are several factors that can further reduce the number of MACs, such as: exploiting coefficient properties, such as even/odd symmetry, zero coefficients (in halfband filter), coefficient bit width and the target device timing.
- Is the processing clock rate close to reasonable FPGA clock rate - such as greater than 300 MHz if you are targeting K7 device with -1 speed grade?
- Are there any particular C/C++ coding styles that produce efficient RTL?
- What is the optimization process in using HLS tool?

Coding the FIR Class

The C++ class is an object oriented programming (OOP) concept that encapsulates both program data and methods in the class definition. The encapsulation allows you to build a complex design using basic building blocks in a modular way. Once a base class is defined, you can easily extend or instantiate the base class in a new class and use it in the provided APIs without having to know the implementation details of the base class.

In addition, C++ class and functions can be template-ized, which allows designers to pass the constants, variable types and HLS optimization parameters through template arguments - making customizable C++ class and functions without having create a whole new class.

Below is an example of class definition for non-symmetric coefficient FIR filter:

```
template<int l_WHOLE, int l_SAMPLE, int II_GOAL>
class nosym_class {
    DATA_T sr[l_WHOLE];
    ACC_T acc;
    COEF_T coeff[l_WHOLE];
public:
    // MAC engine
    ACC_T MAC( DATA_T din, COEF_T coef, ACC_T acc );
};
```

```

// filter
void process ( DATA_T din, DATA_T* dout);
void process_frame(DATA_T din[l_SAMPLE], DATA_T dout[l_SAMPLE]);
void init(const COEF_T cin[l_WHOLE]);
}; // nosym_class

```

The FIR class definition includes the following three major parts:

- Data members
 - coeff: Coefficient memory
 - sr: Shift registers
 - acc: Accumulator storage
- Methods:
 - MAC(): Multiply-and-accumulate function
 - process(): Process data in series of MAC operation.
 - process_frame(): Process one frame of data
 - init(): Initialize coefficient memory
- Template parameters:
 - l_WHOLE: The number of taps or coefficients (= FIR order +1)
 - l_SAMPLE: The number of data samples to process
 - II_GOAL: Target HLS initiation interval (= number of clock cycles between consecutive input samples, input sample rate = clock rate/II_GOAL)

It is worth noting that in HLS, you can use an arbitrary number of bits and fractional number of bits when you define variables, instead of typical int/short/char type of declaration. The convention is as follows:

```

ap_fixed<total number of bits, number of integer bits, truncation/rounding,
wrap/saturate when overflow>

```

There are several data types used throughout the FIR classes. They are, DATA_T (data type for FIR input and output), COEF_T (data type for coefficients), and ACC_T (data type for MAC accumulator). This allows you to easily do trade-offs among resource utilization, precision, and dynamic ranges.

```

typedef ap_fixed<3+NFRAC, 3, AP_TRN, AP_WRAP> DATA_T;
typedef ap_fixed<10+NFRAC, 10, AP_TRN, AP_WRAP> ACC_T;
typedef ap_fixed<2+NFRAC, 2, AP_RND_CONV, AP_WRAP> COEF_T;

```

Multi-Stage Filter Class

With C++ classes and OOP methodology, a multi-stage FIR filter can be constructed simply by instantiating objects from the base classes for each processing stage and then calling the processing methods associated with each object.

Below is an abstracted definition of the multi-stage filter design which includes a Square-Root Raised Cosine (SRRC) filter stage and three stages of half-band filters. Each stage up-converts the data sample rate by 2.

```

template<int l_INPUT>
class filterStageclass {

public:

interp2_class<Lsrrc_WHOLE,  l_INPUT,  II_SRRC> srrc;
interp2_hb_class<Lhb1_WHOLE, 2*l_INPUT, II_HB1> hb1;
interp2_hb_class<Lhb2_WHOLE, 4*l_INPUT, II_HB2> hb2;
interp2_hb_class<Lhb3_WHOLE, 8*l_INPUT, II_HB3> hb3;

void process(DATA_T din[l_INPUT], DATA_T dout[16*l_INPUT] ) {
#pragma HLS INLINE
#pragma HLS dataflow
    DATA_T srrc_dout[2*l_INPUT];
    DATA_T hb1_dout[4*l_INPUT];
    DATA_T hb2_dout[8*l_INPUT];

    srrc.process_frame(din, srrc_dout);
    hb1.process_frame(srrc_dout, hb1_dout);
    hb2.process_frame(hb1_dout, hb2_dout);
    hb3.process_frame(hb2_dout, dout);
}
};

```

Note: Each filter stage has a different value passed to II_GOAL template parameter as defined below. "II_GOAL" specifies the number of clock cycles between each input sample, such that the input sample rate of filter equals the processing clock rate / II_GOAL. Using this method, HLS produces an efficient HW structure that uses the resources that is equal or close to the theoretical minimum number of MAC resources.

- II_SRRC: 16
- II_HB1: 8
- II_HB2: 4
- II_HB3: 2

The HLS synthesis results for each filter stage and the overall DUC design are shown in the following figures:

The synthesis report contains the following information:

- Estimated clock period
- Latency: The number of clock cycles to process and generate all the outputs
- Interval: The number of clock cycles before the function can accept new input data
- Estimated FPGA resources

Performance Estimates								
▣ Timing (ns)								
▣ Summary								
Clock	Target	Estimated	Uncertainty					
ap_clk	4.00	3.86	0.50					
▣ Latency (clock cycles)								
▣ Summary								
Latency		Interval		Type				
min	max	min	max					
3270	3271	3200	3200	loop rewind				
▣ Detail								
▣ Instance								
▣ Loop								
		Latency		Initiation Interval				
Loop Name		min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L_process_frame		3270	3270	87	16	16	200	yes
Utilization Estimates								
▣ Summary								
Name	BRAM_18K	DSP48E	FF	LUT				
DSP	-	-	-	-				
Expression	-	-	0	14				
FIFO	-	-	-	-				
Instance	-	3	345	222				
Memory	-	-	-	-				
Multiplexer	-	-	-	567				
Register	-	-	1248	324				
Total	0	3	1593	1127				
Available	650	600	202800	101400				
Utilization (%)	0	~0	~0	1				

Figure 3: Synthesis Report for SRRC Block

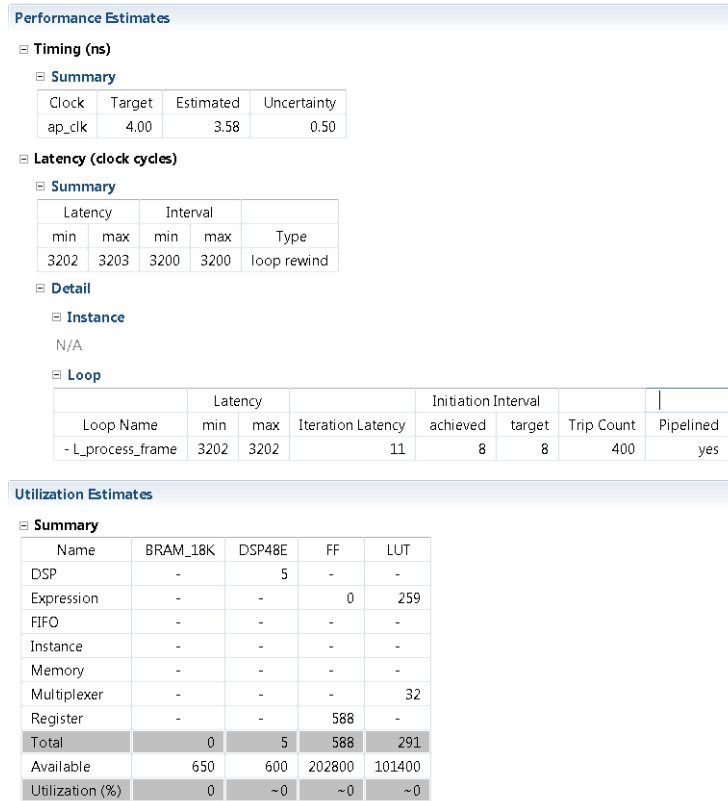


Figure 4: Synthesis Report for the First Halfband Filter Block

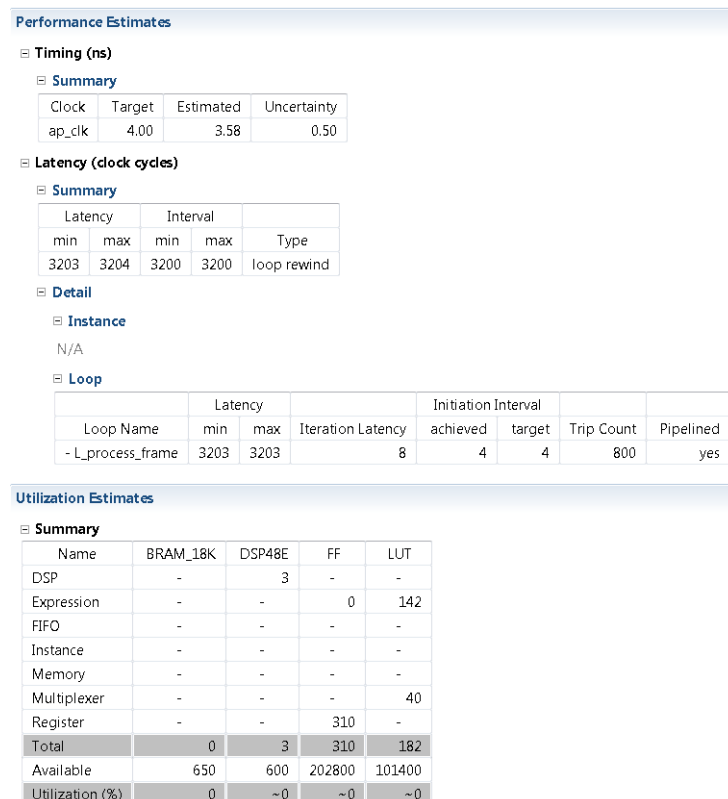


Figure 5: Synthesis Report for the Second Halfband Filter Block

Performance Estimates

▣ **Timing (ns)**

▣ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.58	0.50

▣ **Latency (clock cycles)**

▣ **Summary**

Latency		Interval		Type
min	max	min	max	
3204	3205	3200	3200	loop rewind

▣ **Detail**

▣ **Instance**

N/A

▣ **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- L_process_frame	3204	3204	7	2	2	1600	yes

Utilization Estimates

▣ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	0	110
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	43
Register	-	-	202	18
Total	0	2	202	171
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

Figure 6: Synthesis Report for the Third Halfband Filter Block

Performance Estimates

▣ **Timing (ns)**

▣ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.86	0.50

▣ **Latency (clock cycles)**

▣ **Summary**

Latency		Interval		Type
min	max	min	max	
3312	3313	3200	3200	dataflow

▣ **Detail**

▣ **Instance**

▣ **Loop**

Utilization Estimates

▣ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	60	298
Instance	2	30	6369	4479
Memory	-	-	-	-
Multiplexer	-	-	-	2
Register	-	-	20	-
Total	2	30	6449	4779
Available	650	600	202800	101400
Utilization (%)	~0	5	3	4

Figure 7: Synthesis Report for Two Channel Multi-Stage Filter Block

Note: The "Interval" is 3200 in the synthesis report. This is defined as a number of clock cycles to process one frame input samples, which is defined as 200 samples at the input of the DUC module. Since each I/Q input sample comes in at every 16 clock cycles; thus, the interval time is 3200. During the 3200 clock cycles, there are 3200 samples of output being generated out of the DUC module, for every 200 input sample pair, since the DUC interpolates the input sample by factor of 16.

DDS Class

The DDS block has two main components - phase accumulator and sine/cosine block, shown in [Figure 8](#). The two most important specifications for DDS are:

- Frequency/phase Resolution: Determines how the fine adjustment frequency can be made. The higher the resolution the bigger the look up table would be.
- Spurious Free Dynamic Range (SFDR): Achieving the maximum SFDR with the least amount of resource (mostly the size of the lookup table), is probably the most important design goal.

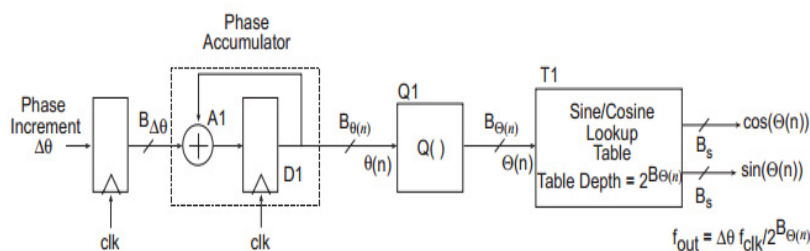


Figure 8: DDS Block Diagram

One common technique in maximizing SFDR without having to increase the sine/cosine lookup table, is to use dithering of the output of the phase accumulator ("Q1" block in [Figure 8](#)). It is effective in suppressing the major spurs but the overall noise floor is raised (middle figure in [Figure 9](#)).

A different method is used here, with the goal of lowering both major spurs and the noise floor. The sine/cosine lookup table is split into two tables - one coarse table and one fine table. The coarse table has the phase resolution of $2\pi/N$, where $N=4096$, while the phase resolution for the fine table is $2\pi/(N*M)$, where $M=512$. The output of the both the tables are combined to produce sine and cosine output.

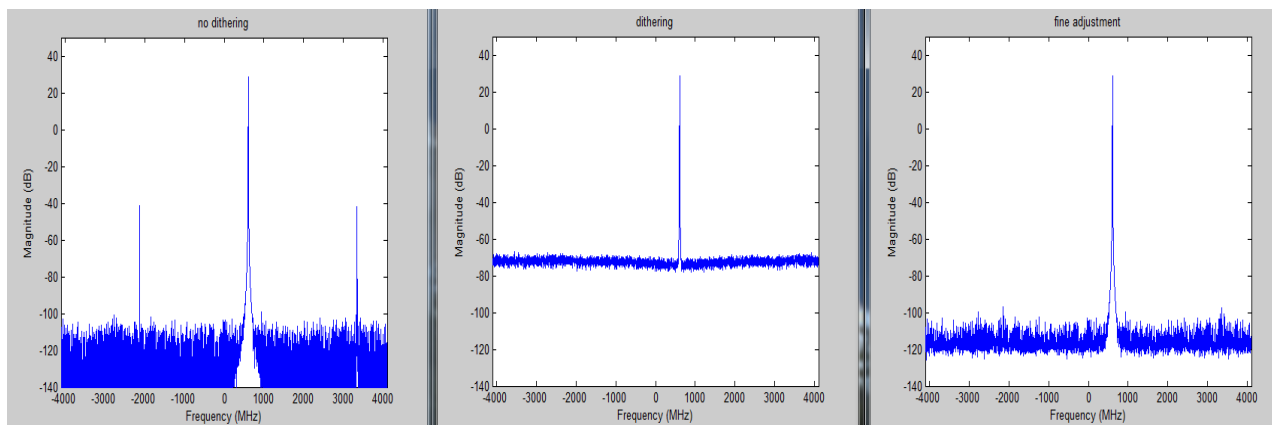


Figure 9: PSD Plot for DDS

A: Original - no method of maximize SFDR

B: Using dithering

C: Using split table

The phase output of the accumulator (32-bit) is truncated to 21-bit. The 12 MSBs looks up the coarse table, and the 9 LSBs looks up the fine table. By partitioning the 21 bits into two groups of address, a 2M word (2^{21}) lookup table is reduced to ~4.5K word table ($2^{12} + 2^9$). The two outputs of the lookup table are combined using the following equation:

$$\begin{aligned}
 e^{j*\theta} &= e^{j*(\phi+\epsilon)} = e^{j*\phi} e^{j*\epsilon} = (\cos(\phi) + j*\sin(\phi)) * (\cos(\epsilon) + j*\sin(\epsilon)) \\
 &= (\cos(\phi)\cos(\epsilon) - \sin(\phi)\sin(\epsilon)) + j(\sin(\phi)*\cos(\epsilon) + \cos(\phi)\sin(\epsilon)) \\
 &= (\cos(\phi) - \sin(\phi)\sin(\epsilon)) + j(\sin(\phi) + \cos(\phi)\sin(\epsilon)),
 \end{aligned}$$

Since, $\cos(\epsilon)$ is almost unity.

This solution is slightly more expensive (BRAM, 2 mults, 2 add/sub) than the dithering technique (LFSR, adder). But it does not suffer from the high noise floor as in the case of dithering.

The top level DDS code is shown in [Figure 10](#). The two functions, `init_cos_lut` and `init_fine_lut`, initialize the two tables, `cos_lut` (the coarse table) and `fine_lut` (the fine table) respectively. See [Figure 11](#) for more details. It is important to note that the HLS tool recognizes that these two tables are read only and hence implements them into a ROM and does not build any logic to compute the table content. In the `cos_lut` table, in order to save the memory size, only the quarter quadrant of a circle is stored and the sign is changed as the angle goes around the unit circle.

```

53 //
54 // N:      number of sample to process
55 // DDS_T:  cos/sin data type
56 // OUTPUT_T: output data type
57 //
58
59 template < int N, typename DDS_T, typename OUTPUT_T >
60 class dds_class {
61
62 // tables
63 lut_word_t  cos_lut[LUTSIZE];
64 fine_word_t fine_lut[FINESIZE];
65
66 // accumulator
67 acc_t acc;
68
69 dds.h [RO]
70
71 void process ( incr_t incr, DDS_T* cos_out, DDS_T* sin_out ) {
72
73 fine_adr_t fine_adr;
74 fine_word_t fine_word;
75
76 lut_adr_t full_adr;      // cover full quadrant
77 quad_adr_t lsb;         // cover 1/4 quadrant
78 quad_adr_t cos_adr, sin_adr;
79
80 ap_uint<2> msb;         // specify which quadrant
81 lut_word_t  cos_lut_word;
82 lut_word_t  sin_lut_word;
83
84 // initialize tables
85 init_cos_lut( cos_lut, LUTSIZE );
86 init_fine_lut( fine_lut, FINESIZE, DELTA );
87
88 // phase accumulator
89 acc += incr;
90
91 // look up cos/sine table
92 full_adr = acc(31,20);
93 msb      = full_adr(11,10);
94 lsb      = full_adr(9,0);
95
96

```

Figure 10: DDS Class (showing “dds::process”)

```

53 //
54 // N:      number of sample to process
55 // DDS_T:  cos/sin data type
56 // OUTPUT_T: output data type
57 //
58
59 template < int N, typename DDS_T, typename OUTPUT_T >
60 class dds_class {
61
62 // tables
63 lut_word_t  cos_lut[LUTSIZE];
64 fine_word_t fine_lut[FINESIZE];
65
66 // accumulator
67 acc_t acc;
68
69 dds.h [RO]
70
71 void init_cos_lut( lut_word_t cos_lut[LUTSIZE], const int LUTSIZE ) {
72
73 double cos_double;
74 //ofstream fp_dout ("debug.txt");
75
76 // #define FULL
77 //-----
78 #ifndef MIDPOINT
79 // store single quadrant
80 for (int i=0; i<LUTSIZE; i++) {
81 //cos_double = cos(2*M_PI*(0.0+(double)i)/(4*LUTSIZE));
82 cos_double = cos(2*M_PI*(0.5+(double)i)/(4*LUTSIZE));
83 cos_lut[i] = cos_double;
84 fp_dout << scientific << cos_double << endl;
85 }
86
87 #ifdef FULL
88 // store full quadrant
89 ofstream fp_ideal ("ideal.txt");
90 for (int i=0; i<4*LUTSIZE; i++) {
91 cos_double = cos(2*M_PI*(0.5+(double)i)/(4*LUTSIZE));
92 fp_ideal << scientific << cos_double << endl;
93 }
94 #endif
95
96 #else
97 //-----
98 // not the mid point
99
100 // store single quadrant
101 for (int i=0; i<LUTSIZE; i++) {
102 cos_double = cos(2*M_PI*(0.0+(double)i)/(4*LUTSIZE));
103 cos_lut[i] = cos_double;
104 //fp_dout << scientific << cos_double << endl;
105 }
106
107 #ifdef FULL
108 // store full quadrant
109 ofstream fp_ideal ("ideal.txt");
110 for (int i=0; i<4*LUTSIZE; i++) {
111 cos_double = cos(2*M_PI*(0.0+(double)i)/(4*LUTSIZE));
112 fp_ideal << scientific << cos_double << endl;
113 }
114 #endif
115
116 #endif
117
118
119
120
121
122
123
124 #endif
125
126 #else
127 //-----
128 // not the mid point
129
130 // store single quadrant
131 for (int i=0; i<LUTSIZE; i++) {
132 cos_double = cos(2*M_PI*(0.0+(double)i)/(4*LUTSIZE));
133 cos_lut[i] = cos_double;
134 //fp_dout << scientific << cos_double << endl;
135 }
136
137 #ifdef FULL
138 // store full quadrant
139 ofstream fp_ideal ("ideal.txt");
140 for (int i=0; i<4*LUTSIZE; i++) {
141 cos_double = cos(2*M_PI*(0.0+(double)i)/(4*LUTSIZE));
142 fp_ideal << scientific << cos_double << endl;
143 }
144 #endif
145
146
147 #endif
148
149

```

Figure 11: DDS Class (showing “DDS::init_cos_lut”)

Before the DDS block is integrated into the top level DUC, it is synthesized standalone by the HLS tool and checked against the following design goals:

- Both sine and cosine output are produced at every clock cycle. This is verified by checking the initiation interval is 1. The HLS synthesis directive used for this is "pipeline".
- Resource budgets are met. The both look up tables are implemented in BRAM and not LUT (using the "resource" HLS directive) and HLS does not build any logic to initialize those tables.
- Clock rate is reasonable - greater than 300 MHz.

The HLS synthesis report is shown in the figures below:

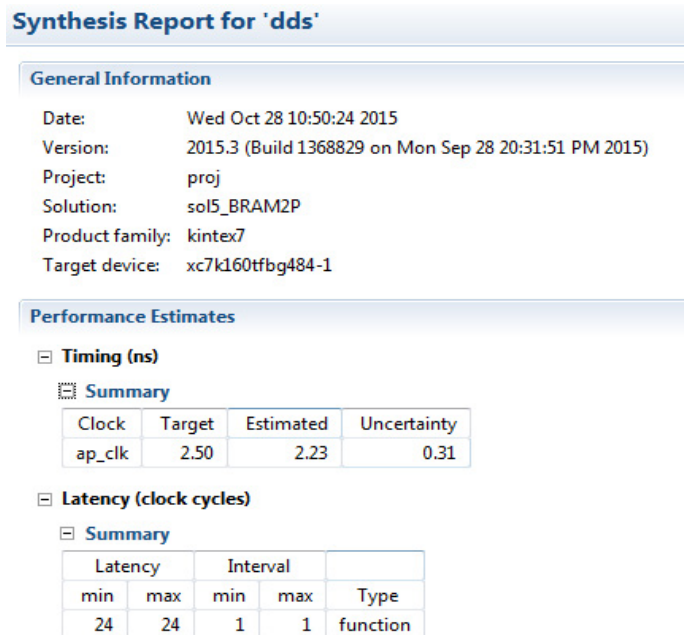


Figure 12: DDS HLS Synthesis Report (a)

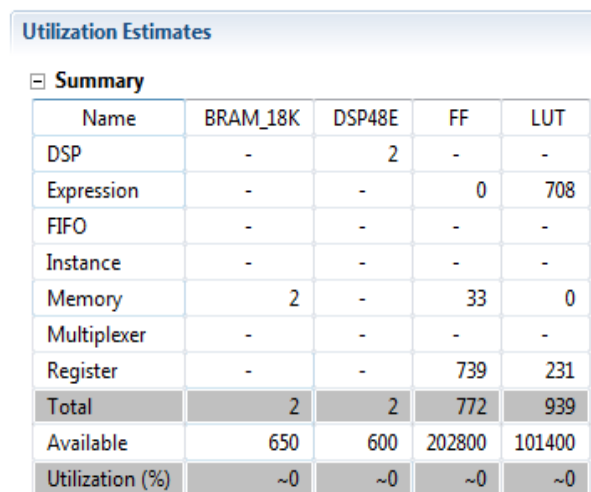


Figure 13: DDS HLS Synthesis Report (b)

Top Level Design

Once all the submodules have been synthesized and the design goals (desired initiation intervals and resource budgets) are met, they are integrated into the top module as shown in the code in [Figure 14](#).

```

4 //
5 void duc( DATA_T din_i[L_INPUT], DATA_T din_q[L_INPUT], DATA_T dout[L_OUTPUT],
6         incr_t incr ) {
7
8 #pragma HLS interface axis depth=L_INPUT port=din_i
9 #pragma HLS interface axis depth=L_INPUT port=din_q
10 #pragma HLS interface axis depth=L_OUTPUT port=dout
11
12 #pragma HLS interface ap_stable port=incr
13
14 #pragma HLS dataflow
15
16 static filterStageClassTwoChan <L_INPUT> f0;
17 static dds_class <L_OUTPUT, dds_t, DATA_T> dds_0;
18
19 DATA_T dout_i[L_OUTPUT];
20 DATA_T dout_q[L_OUTPUT];
21 dds_t dds_cos[L_OUTPUT];
22 dds_t dds_sin[L_OUTPUT];
23
24 f0.process(din_i, dout_i, din_q, dout_q);
25 dds_0.process_frame(incr, dds_cos, dds_sin);
26 dds_0.mix(dds_cos, dds_sin, dout_i, dout_q, dout);
27
28
29 }
30

```

Figure 14: Top Level DUC Code

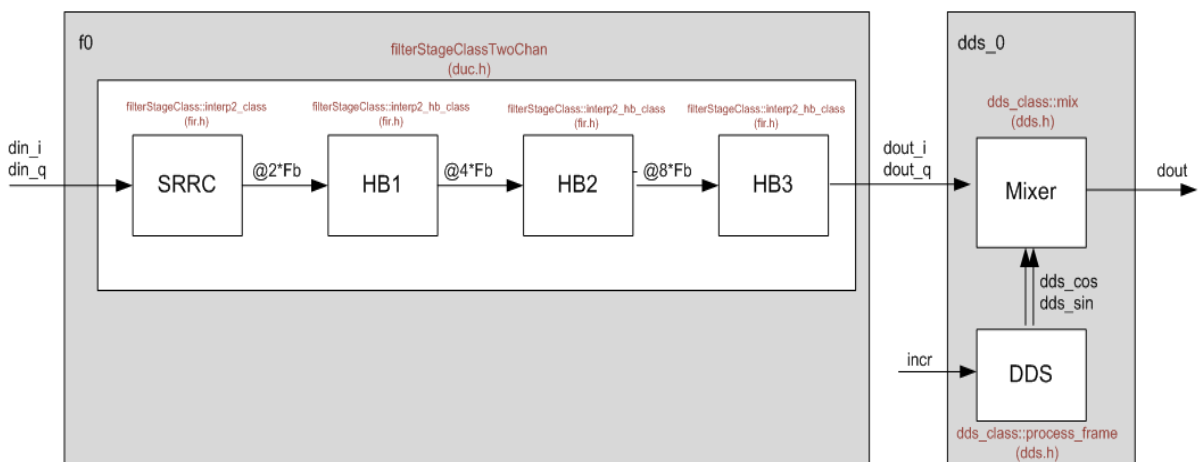


Figure 15: Top Level Block Diagram - Showing the Class Definitions

It is important to note the following:

- The DUC design is a frame based, processing array of data inputs and outputs.
- These arrays of inputs and outputs are synthesized into an AXI streaming interface, using the 'interface' HLS directive (line 8-10).
- The data rate of the inputs and the outputs are different. The output rate is 16 times the input data rate, therefore, $L_OUTPUT = L_INPUT * 16$.
- It is expected that the output of the DUC drives a DAC. Therefore, there should not be any gap or idle between the frames of the output data stream.
- In a pipelined system like DUC, where the data is processed by one block and passed to the next block, it is critical to process the data stream in a pipeline manner. This results in a higher throughput, less overall latency and use of minimal buffer storage. The "dataflow" directive enables this pipelining capability (line 14).
- One benefit of using the "dataflow" directive is that all the output storage of each processing block can be optimized down to minimum size. For example, in this code, all the output storages are declared as "L_OUTPUT" samples (defined to 3200), but all those storages can be implemented as a fifo of depth = 8, since all the access is done in a stream fashion. This is enabled in the Tcl file ("run.tcl"): "config_dataflow -default_channel fifo -fifo_depth 8".
- In a typical HW processing block, there will be a programmable setting to control the behavior of the HW block. In the DUC design, it is the "incr" input, to control the modulating frequency of the DUC. To specify that this input is part of the control word and not the real time signal, the interface directive is set as "interface ap_stable" (line 12).
- Another option for handling programmable setting is by using the AXI4-Lite interface. All the control words can be lumped into the same bundle, and each word will be address-mapped. This is done by the interface directive - "interface s_axilite".

Top Level Synthesis / Implementation

The DUC synthesis report is shown in the below figures. It is important to note that both the performance goal of Initiation Interval of 3200 is met, along with a reasonable resource budget.

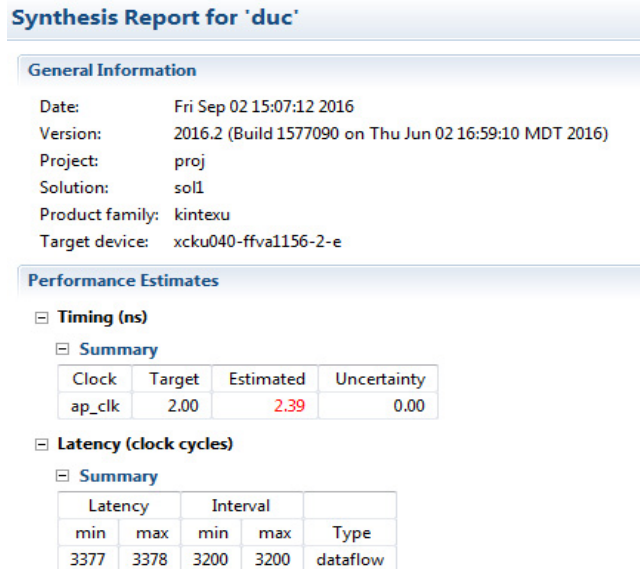


Figure 16: DUC HLS Synthesis Report (a)

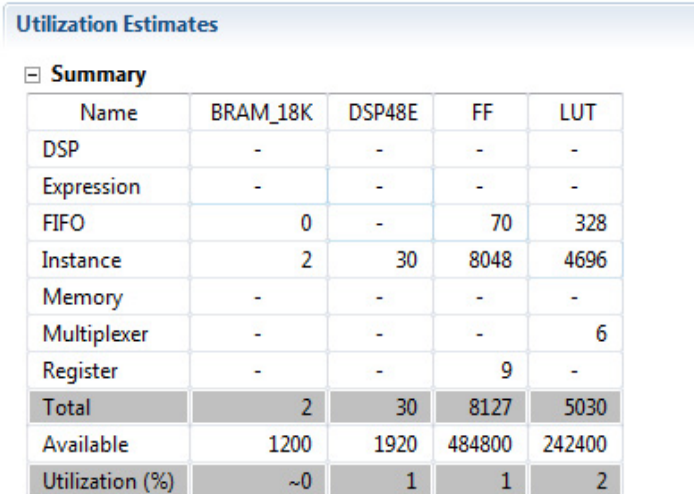


Figure 17: DUC HLS Synthesis Report (b)

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
din_i_V_TDATA	in	24	axis	din_i_V	pointer
din_i_V_TVALID	in	1	axis	din_i_V	pointer
din_i_V_TREADY	out	1	axis	din_i_V	pointer
din_q_V_TDATA	in	24	axis	din_q_V	pointer
din_q_V_TVALID	in	1	axis	din_q_V	pointer
din_q_V_TREADY	out	1	axis	din_q_V	pointer
dout_V_TDATA	out	24	axis	dout_V	pointer
dout_V_TVALID	out	1	axis	dout_V	pointer
dout_V_TREADY	in	1	axis	dout_V	pointer
incr_V	in	32	ap_stable	incr_V	scalar
ap_clk	in	1	ap_ctrl_hs	duc	return value
ap_rst_n	in	1	ap_ctrl_hs	duc	return value
ap_start	in	1	ap_ctrl_hs	duc	return value
ap_done	out	1	ap_ctrl_hs	duc	return value
ap_ready	out	1	ap_ctrl_hs	duc	return value
ap_idle	out	1	ap_ctrl_hs	duc	return value

Figure 18: DUC HLS Synthesis Report (c)

After the top level DUC design has gone through C-to-RTL synthesis successfully, Vivado implementation - FPGA synthesis and place and route, is performed. When Vivado implementation is invoked within the HLS tool, it uses a default implementation strategy and its main goal is to confirm that the estimated resources and the timing estimate from C-to-RTL synthesis are highly correlated to each other. In this DUC design, the clock is intentionally over-constrained to 2 ns (=500 MHz) to see how close implementation can meet that timing constraint. In this design, C-to-RTL synthesis estimated the clock period to be 2.39 ns while the implementation achieved the 2.171 ns, targeting KU040 -2 device. This timing can potentially improve by using a different strategy in Vivado Synthesis/Map/Place and Route.

General Information

Report date: Fri Sep 02 15:59:14 -0400 2016
 Project: proj
 Solution: sol1
 Device target: xcku040-ffva1156-2-e
 Implementation tool: Xilinx Vivado v.2016.2

Resource Usage

	Verilog
CLB	928
LUT	3600
FF	7471
DSP	30
BRAM	2
SRL	689

Final Timing

	Verilog
CP required	2.000
CP achieved	2.171

Timing not met

Figure 19: DUC FPGA Implementation Report

Verification

The HLS tool allows you to verify your design in the following two stages:

1. csim compile/link/simulate your C/C++ design code and testbench code,
2. RTL csim invokes the RTL simulator to verify the HLS generated RTL using the same testbench used in csim.

Figure 20 shows the output of RTL co-simulation.

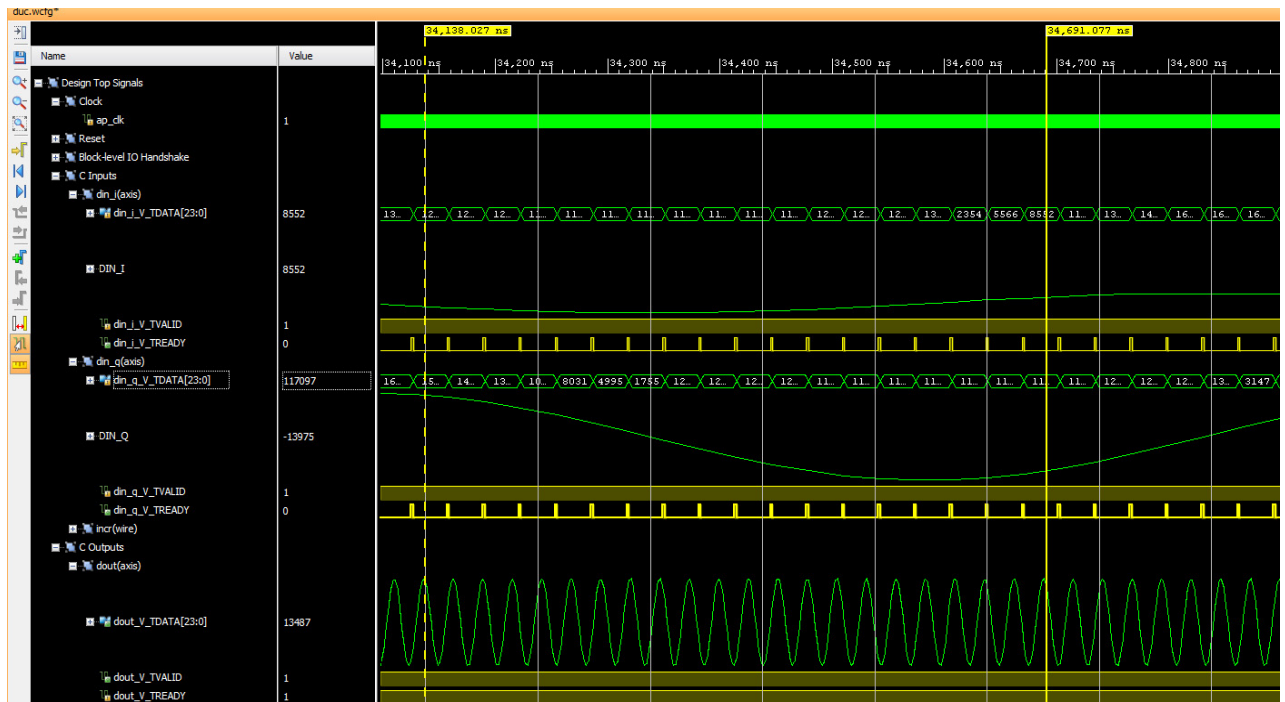


Figure 20: RTL Cosimulation Output

After the verification is complete, the HLS produces the RTL that is exported as a Sysgen block. Sysgen is based on Matlab Simulink tool which is a model based design environment. Once exported as a Sysgen block from the HLS tool, it can be represented as a block within the Sysgen environment and the HLS block is simulated and verified with the rest of the Sysgen/Simulink blocks, thus, making an ideal verification environment for the signal processing block/algorithm. Refer to Figure 20 for more details.

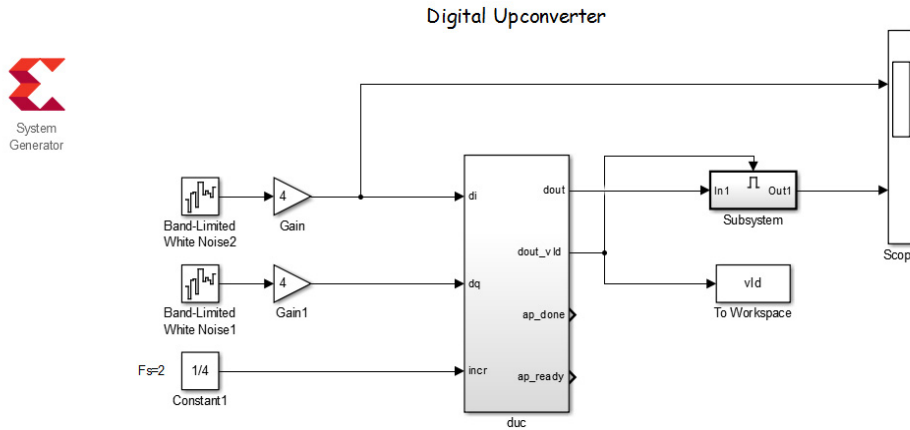


Figure 21: Sysgen Verification of the DUC Block (a)

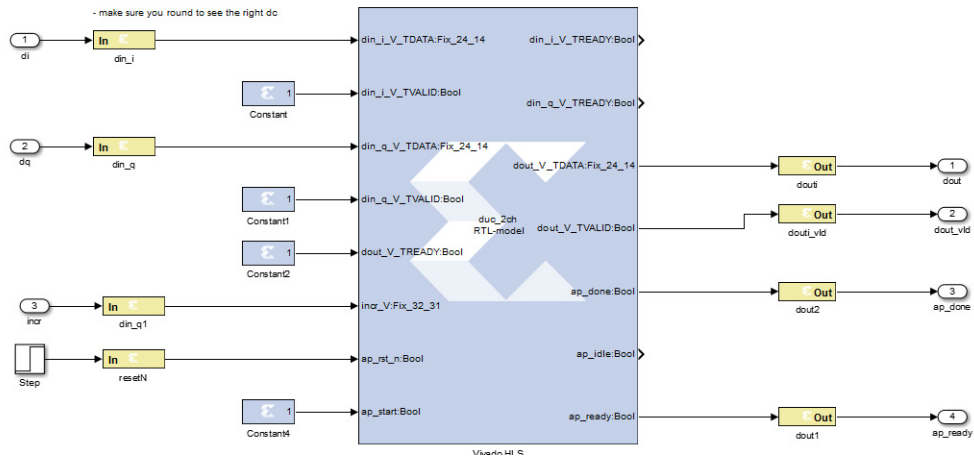


Figure 22: Sysgen Verification of the DUC Block (b)

In this Sysgen design environment, a white random signal is used as an input to both I/Q symbol input sources - these can be white random QAM symbols as well. The modulating frequency is chosen to be $F_s/8$, where F_s is the sampling frequency of the output data stream, and the DUC output stream is captured and its frequency spectrum plot is shown in Figure 19.

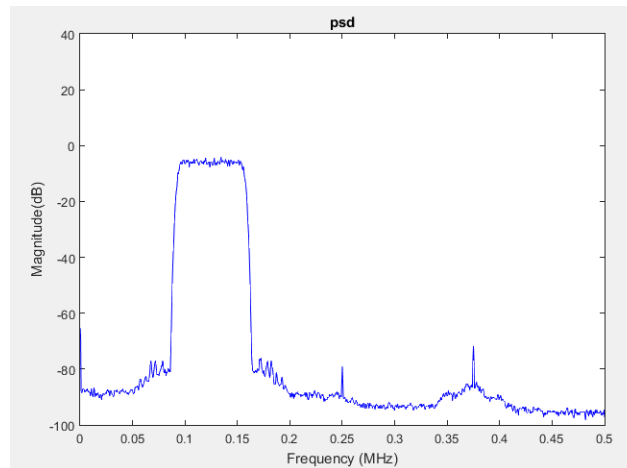


Figure 23: PSD Plot of the DUC Output

Reference Design

Vivado System Edition version 2016.2 is used in this application note design. For Sysgen verification, Matlab/Simulink version 2015b is used.

Download the [reference design files](#) for this application note from the Xilinx website.

Here are the steps to produce the HLS project:

1. Open Vivado HLS command line window.
2. Cd to the design directory.
3. Type "vivado_hls -f run.tcl". This will go through all the process - C-simulation, C to RTL synthesis, export to Sysgen, invoke Vivado implementation and RTL cosimulation.
4. Open Vivado HLS GUI and open the HLS project. Or open the GUI by typing "vivado_hls -p proj" in the command line window.
5. Check out the synthesis report and implementation report - highlighted in yellow in Figure 20.

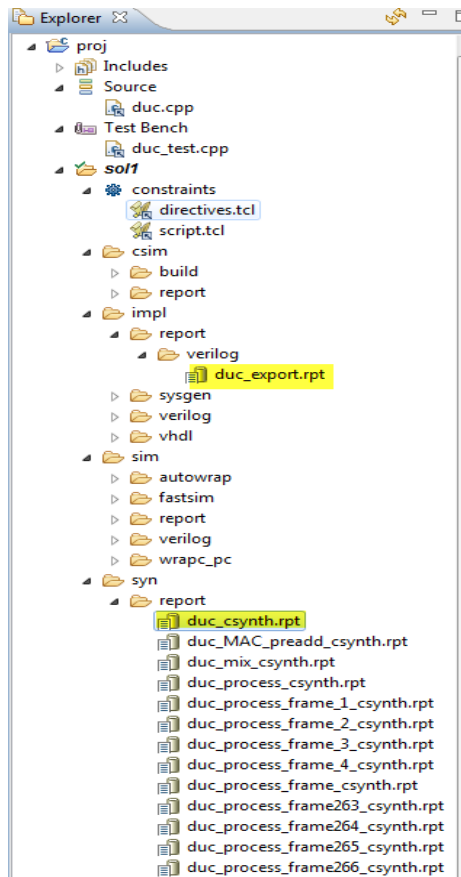


Figure 24: HLS GUI Showing Project Directories

Table 1 shows the reference design matrix.

Table 1: Reference Design Matrix

Parameter	Description
General	
Developer name	Alex Paek, Jim Wu
Target devices	KCKU040 -2 -e
Source code provided	Yes
Source code format	C++, HLS 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++
Simulator software/version used	Vivado simulator

Table 1: Reference Design Matrix (Cont'd)

Parameter	Description
SPICE/IBIS simulations	N/A
Implementation	
Synthesis software tools/versions used	Vivado HLS 2016.2
Implementation software tools/versions used	Vivado Design Suite 2016.2
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	No
Hardware platform used for verification	N/A

Conclusion

This application note walks through the process of designing a digital up-converter (DUC) using the Vivado HLS tool. The process starts with writing C++ code, verifying the design in C simulation, synthesizing C code to produce synthesizable RTL code, verifying the RTL code using RTL/C cosimulation, exporting the RTL to Sysgen environment, verifying the RTL in Sysgen environment, and implemented into FPGA using Vivado implementation tool.

This application note describes in detail how the DUC design is constructed using template classes, and optimization techniques to produce a portable/generic solution where you can easily change the parameters for the design and establishing a basis for building a library of components and for constructing a design with a more complex hierarchy.

References

1. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))
2. *System Generator for DSP User Guide* ([UG948](#))
3. *Implementing Carrier Phase Recovery Loop Using Vivado HLS* ([XAPP1173](#))

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/10/2016	1.0	Initial Xilinx release.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.