XILINX®

XAPP1173 (v1.0) May 2, 2013

Implementing Carrier Phase Recovery Loop Using Vivado HLS

Author: Alex Paek, Duncan Mackay

Summary

This application note describes the implementation of a carrier phase recovery loop algorithm for a single carrier QAM demodulator using Vivado® High-Level Synthesis (HLS). The algorithm, described in C/C++, is compiled to synthesizable RTL by Vivado HLS, added as a block to System Generator for DSP for verification, and then implemented using the Vivado Design Suite.

The automatic resource assignment, sharing, and scheduling capability of Vivado HLS is instrumental in compiling a complex algorithm into a solution optimized for both performance and resource usage.The proposed design methodology can be applied to expedite the implementation of other common baseband signal processing algorithms, such as timing recovery loops, MIMO decoders, and adaptive equalizers.

Introduction

The physical layer structure of any digital communication receiver can be generalized by Figure 1. The details of each block depends on the type of receiver, — a single carrier QAM receiver versus multi-carrier receiver like OFDM, single versus multiple TX/RX antennas, continuous stream versus burst stream type of receiver, and so on. The carrier phase recovery loop is for processing a continuous stream single-carrier QAM receiver with a single antenna.

Such a system has three major components:

- Digital down conversion block, which converts an Intermediate Frequency (IF) data stream coming from the ADC down to the baseband (this part is skipped if the data stream is already in the baseband). Its sample rate is reduced by the use of multiple stages of filters.
- The baseband processing includes:
  - The timing recovery loop for determining the optimum symbol time, and which trigger all its subsequent processing,
  - Carrier phase recovery loop: Determines the phase offset present in the received data stream,
  - Adaptive equalizer: Compensates for any channel impairments by finding the optimum inverse response of the channel response.
- The Forward Error Correction (FEC) decode block: typically there are two levels of FEC. Viterbi code for the inner FEC and Reed Solomon for the outer FEC.

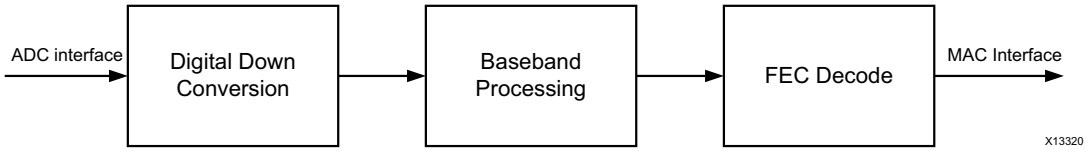


Figure 1: **Typical Demodulator Functional Block**

It is critical to choose the right design methodology not just for fast time to market, but also for design reuse, verification, and test/debug capability. During this decision process, you might select more than one design entry method depending on the type of blocks.

For example,

- If there is a block that is control intensive, for example a state machine, you might choose VHDL or Verilog for the design entry.
- If there are optimized IPs available for a particular function you might choose to utilize those blocks instead of designing them from scratch. Blocks such as filters, FFT, digital direct synthesis (DDS) are readily available in the System Generator toolbox. Using those blocks, you can quickly put together a digital down converter (DDC). Similarly, for the FEC Decode block, many common FEC decoders are available in the System Generator toolbox.
- If there are blocks that are algorithmically complex and there are no available IPs, one can describe the algorithm in C/C++ and use Vivado HLS to convert them into synthesizable RTL, and that is the focus of this application note.

For the top level environment, System Generator is chosen because there are available mechanisms to easily integrate various sources — RTL, System Generator blockset or from C/C++/SystemC. Also, MATLAB and Simulink® allows one to build a sophisticated verification environment using built-in functions and blocks.

This application note focuses on the design methodology and process for quickly realizing the system described above. Although this document provides less emphasis on the algorithmic aspects of the solution, it is worth reviewing the algorithm in an overview of the solution.

Receiver Specification

The QAM receiver specification, of which the carrier phase recovery loop block is part, is as follows:

- Programmable QAM setting: QPSK, 16, 64, 256 QAM with a variable symbol rate of up to 6.25 Msps
- An IF input stream with 8 times the baud rate (50 Msps)
- Timing Recovery loop
- Carrier Phase Recovery loop
- LMS algorithm based adaptive equalizer
- Blind acquisition and tracking for carrier loop and equalizer
- The target device:
  - Artix®-7 device
- Clock rates:
  - 100 MHz for baseband processing
  - 200 MHz for digital down conversion

An overview of the QAM Receiver block is shown in Figure 2.

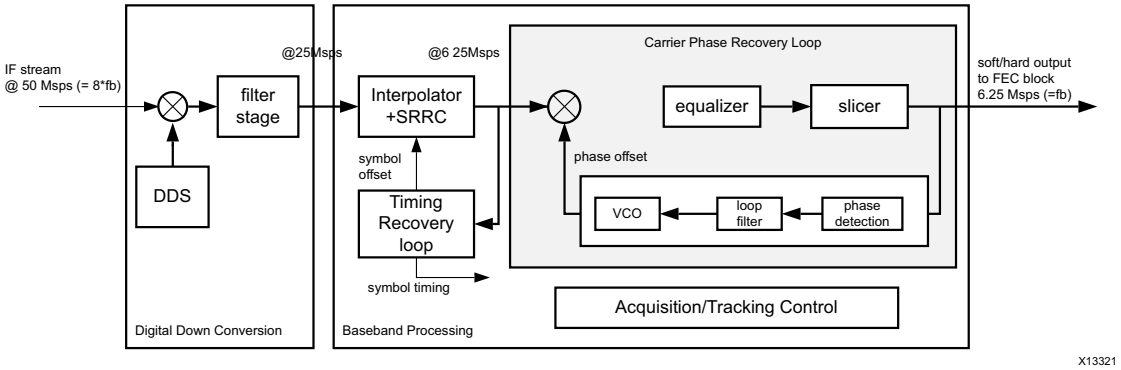


Figure 2: **QAM Receiver Block Diagram**

The algorithm for the whole receiver chain is captured and verified in Simulink environment.

For the implementation, the digital down conversion block is implemented using FIR Compiler [Ref 1] and DDS Compiler [Ref 2] in System Generator while the baseband processing blocks are implemented using Vivado HLS.

This application note describes the detail algorithm and implementation for the carrier phase recovery loop block implementation (shaded region in Figure 2). The equalizer and the remainder of the baseband block is not covered in this application note.

Algorithm

There are two major synchronizations performed in a typical continuous-transmission single carrier QAM receiver to correct for any offset in the symbol timing and carrier phase between the transmitter and the receiver. These two offsets, caused largely by an inherent offset between the Local Oscillator (LO) on the transmitter and the receiver, are recovered and tracked by the timing recovery loop and the carrier phase recovery loop (CR). These are the critical elements to perform a coherent demodulation of the received data stream.

Figure 3 shows the Simulink model of the CR block.

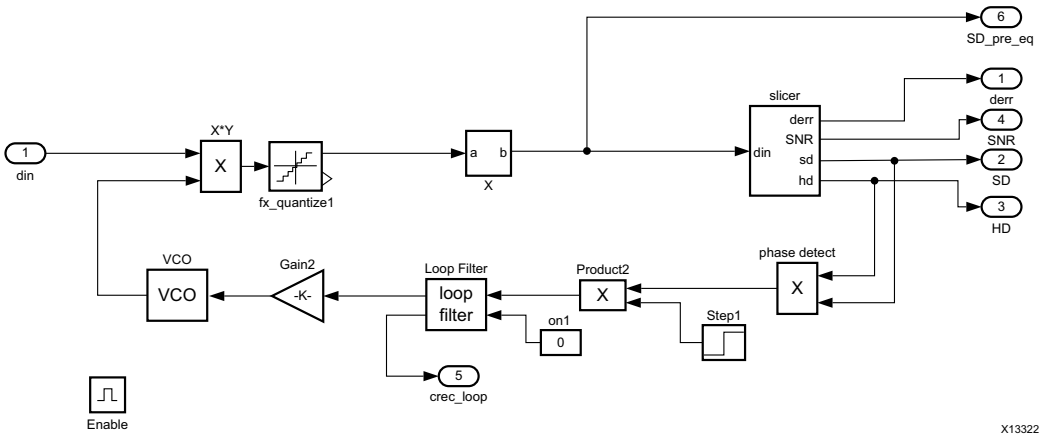


Figure 3: **Simulink Model of Carrier Phase Recovery Loop**

The input to the CR block is I/Q data coming from a Square Raised Root Cosine (SRRC) filter at a symbol rate and sampled at an optimum symbol time.

The timing recovery block calculates this optimum symbol time from the received data stream and produces the strobe signal which is distributed to all the blocks that are processed at every symbol time — including the CR block and adaptive equalizer. The timing recovery block also computes the optimum symbol by the use of variable interpolation whose phase is determined during the computation of symbol time [Ref 3] [Ref 4].

The function of the CR block is easily seen at the input and output of the block, shown in Figure 4. The input is a rotating constellation due to carrier offset plus some constant phase offset and the output is phase corrected and shown as a stable fixed constellation.

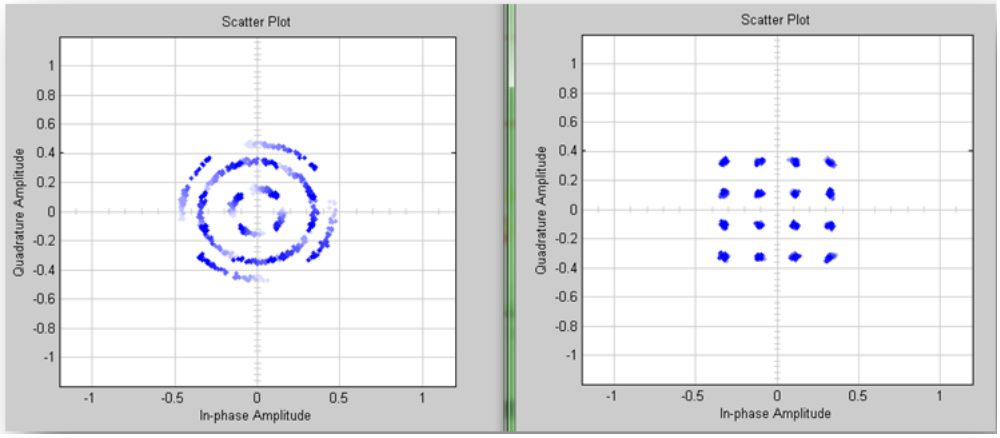


Figure 4: **Input and Output of the Carrier Phase Recovery**

The carrier phase recovery loop algorithm is based on a scheme commonly known as a "Costas loop". This consists of the following elements [Ref 5]:

- Phase detector
- PI loop filter
- Digital VCO (voltage controlled oscillator)
- Derotator
- Slicer

The phase detector determines the instantaneous phase offset by approximating the angle between the de-rotated constellation (y_k, I/Q input to the CR block) and the expected constellation (c_k, which is hard decision), using the approximation Im{y_k*c_k^* }. Both y_k, c_k are the output of the slicer, which maps the y_k, soft decision into the hard decision, c_k.

The PI loop filter takes the instantaneous phase offset and averages over a length of time; and thus, controls the loop response of the CR closed loop. The origin of loop filter comes from an analog phase-locked loop. The loop filter has a proportional term and an integrator term, which determine the characteristic of the loop response such as loop bandwidth and damping factor. Whether the CR loop converged or not can be determined by looking at the integral term of the loop filter.

The output of the loop filter is the carrier offset: after the CR loop enters into a locked state, this value stays relatively constant. This offset becomes the input to the VCO.

The VCO generates the sine/cosine terms whose frequency is controlled by the carrier offset. The major components are the accumulator and the look-up table (LUT). To reduce the size of the LUT to store sine/ cosine terms, only one quadrant of the sine term is stored using the symmetry of the cosine terms. (This can even be halved by storing only the half of the quadrant.) The number of words and the bit width of the cosine LUT and the bit width of the accumulator are parameters which need to be determined from the system simulation.

An example of the output is provided in Figure 5.

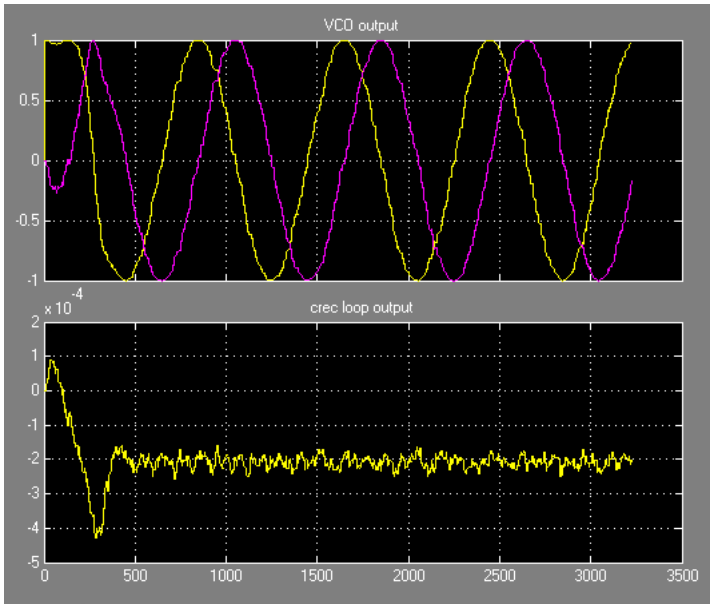


Figure 5: **Output of VCO and the Integral Term of Loop Filter**

The final part of the entire system is the adaptive equalizer (EQ). The EQ is not covered in this application note, but how both the CR and EQ interact with each other in a closed loop system and how both converge and stay converged is a topic called acquisition and tracking. Blind acquisition, where CR-EQ converges without the use of any known data stream is a well-researched topic. There are several widely used blind algorithms (RCA-reduced constellation algorithm, MMA-multi-modulus algorithm) and these are described in the reference section [Ref 6] [Ref 10]. These algorithms essentially modify how the error terms, which are phase offset for CR, and decision error for the EQ, are computed but change very little from the proposed CR block structure.

Design Strategy

The first step in the design of this system is to create the carrier phase recovery loop algorithm with fixed-point accuracy in Simulink, and verify it with the remainder of the QAM receiver. The QAM receiver is designed according to the aforementioned specification: the input source is an IF stream at 50 Msps with some timing offset and a carrier offset of approximately 40 KHz.

The CR algorithm can then be captured in C++. Using C++ offers a few advantages over standard ANSI C when designing complex systems to be realized in an FPGA. Notable advantages are:

- The ability to use classes and templates: this simplifies design capture and reuse.
- Support for arbitrary precision fixed-point types, allowing bit accurate specification of both the integer part and fractional part of variables.

Vivado HLS is an integrated design environment (IDE) and contains a C/C++ code debugger and compiler, a synthesis tool to synthesize C, C++ or SystemC to RTL (VHDL or Verilog) and IP packaging functionality which allows the output to be easily used by other tools in the Xilinx design environment. [Ref 8]

A test bench is created to verify the functionality of the C++ code matches the functionality of the Simulink model. The test bench code can easily be created using the input source from the Simulink model. After the C++ code has been functionally verified, Vivado HLS can be used to perform C/C++ to RTL synthesis.

Like any design process implemented within a large system, a sensible hierarchy is encouraged for debugging, implementation and the integration of blocks into the larger system, regardless of what design method is used (RTL, C/C++ or model based design). Following that philosophy, the CR code is partitioned into several submodules, shown in Figure 6.

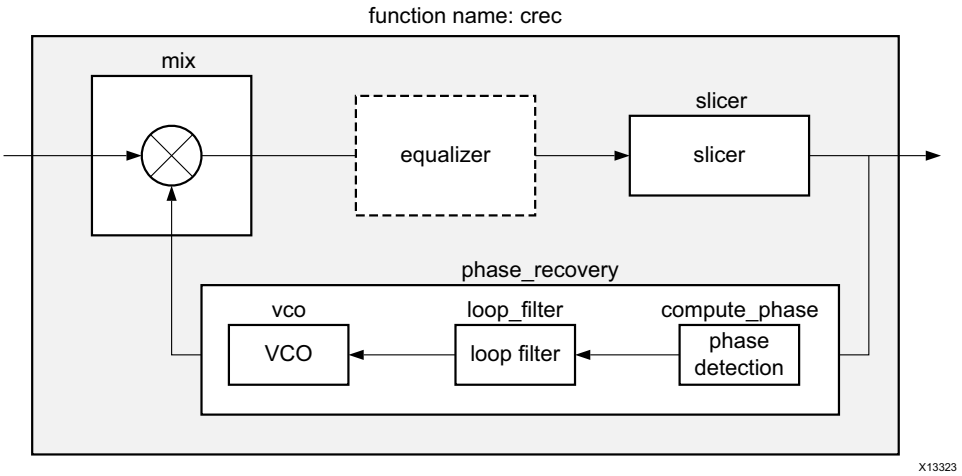


Figure 6: **Partitioning of the CR Code**

The same hierarchy as the Simulink model is used, and each submodule is processed to ensure the resource usage, design throughput, and design latency are reasonable before the top module is synthesized.

In Vivado HLS, all source files can be imported into a "source" directory. When a submodule is designated as the top-level for synthesis, the rest of the submodules become a part of test bench: they are not synthesized, but used to confirm (test) the correct operation of the top-level for synthesis.

After each of the individual submodules is processed and the resource and performance of each has been confirmed, the entire design can be synthesized as a single block. It is possible to synthesize each submodule in isolation, package it as IP and integrate each submodule into a top-level design inside System Generator but most likely there would be some cross-functional optimization performed when processing several functions together and thus, producing a final design with less resources and better throughput/latency than those of the sum of the individual blocks.

Throughput Goal

Before the implementation starts, it is important to set the throughput requirement. With Vivado HLS, the throughput is specified by the interval: the number of clock cycles between new input samples. This is derived as follows:

- The processing clock rate for the CR block is 100 MHz.
- The CR block is processing the symbol data coming in at a symbol rate which is nominally at a rate of 6.25 MHz. For a 100Mhz clock, there are 16 clock cycles between each new data sample. This is a nominal rate, meaning it is 16 clock cycles most of the time but it can be less or more depending on whether the receiver is running at a slower or faster rate than that of the transmitter.
- The timing recovery loop block is using the 100 MHz clock but operating at the data rate of 25 MHz because its incoming data from DDC is 25 MHz, thus, the output changes at the resolution of 25MHz: 4 clock cycles (with a 100 MHz clock).
- Because the timing recovery block continuously keeps track of the optimum symbol time, its strobe time varies accordingly: it would nominally be 6.25 MHz, but can shift earlier or later by 4 clock ticks, making the strobe time 12, 16 or 20 clock cycles.
- Intuitively, if the receiver is running at a slightly slower rate than the transmitter, most of time the strobe time would be 16 clock cycles, but occasionally it can see 12 cycles between the strobes.
- Therefore, the CR block must be specified to run at an interval of 12 clock cycles.

It is worth noting the design reuse aspect of using Vivado HLS. As long as the algorithm stays constant, even though the data/symbol rate requirements might change, the exact same source code can be used. New design requirements can have a different throughput goal but new goals can be met by using different synthesis constraints (change performance or use more or less resources): the same validated C++ algorithm, the source code, does not need to be changed.

Closing the Loop

Because the CR block is a closed-loop system, it imposes a few implementation challenges. It cannot simply be pipelined to meet the timing as you would do with a non-closed loop design because pipelining the feedback path changes the overall system response.

While the CR block can tolerate a few changes in symbol latency, some closed loop systems are more sensitive or even prohibitive to this latency change than others. For example, IIR filters, least mean squares (LMS) adaptive equalizers cannot tolerate any latency (Note however, that there are algorithms that allows pipelining in LMS [Ref 7]).

The other consideration is the error propagation in the closed loop. To be sure of this, one should insert a way to clear the feedback term.

Whether Vivado HLS or any design in general can easily meet the throughput goal depends on the symbol duration: the clock rate/symbol rate is basically how many clock cycles are there in a symbol time.

If one has difficulty meeting the throughput goal using Vivado HLS, the following strategy can be applied to implement any closed loop design.

1. Break the loop in the design. Figure 7 shows the same CR block as Figure 6 except that the feedback loop is broken. Therefore:
   - Ns = the max number of tolerable sample/symbol latency in the loop: it can be 0 to a few samples.
   - Ts = the number of clock cycles per sample.
   - Tp = the number of clock cycles from A to B.
   - Np = the number of symbol latency from A to B = floor(Tp/Ts).

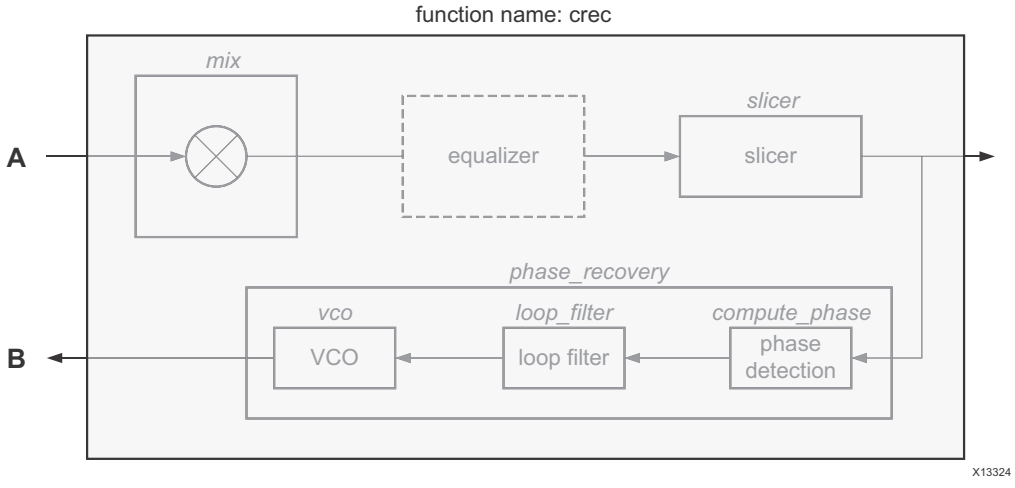


Figure 7: **CR BLOCK Code as an Open Loop**

2. Use Vivado HLS to synthesize the design.
3. Is the resulting value Np <= Ns?
   - Yes: Go to step 4.
   - No: Return to step 2 and apply optimization directives until the condition is met. If this is not achievable, an algorithmic change might be required.

4. Connect the loop outside the module as shown in Figure 8 and be sure to perform the step noted in the sub-bullet before implementing the design in the FPGA.
   - Before committing the design to hardware, insert an initialize (int) or `clear` signal, to zero out the feedback path. This ensures the unknown is not propagated from the initial power up.

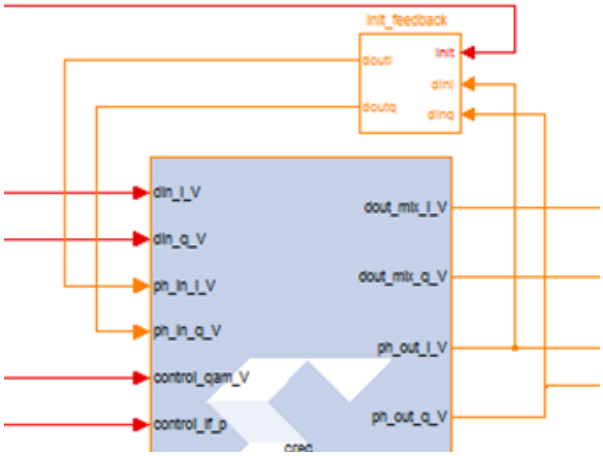


Figure 8: **Closed Loop System from an Open-Loop CR block**

VHLS Implementation

With the system design and methodology steps addressed, the implementation details of the CR block in Vivado HLS can now be reviewed. This section discusses all aspects of Vivado HLS as it pertains to the CR block implementation. This includes a review of:

- source code and coding style
- arbitrary precision data-types
- use of Synthesis Directives to perform optimizations
- Synthesis Report
- RTL verification

Further details on the optimizations are also provided later.

Coding Style

For the most part, the code can be written in a very behavioral manner without any particular hardware implementation in mind. A few exceptions to this are:

- System calls like printf, and calls to operating systems like getc(), time() are not allowed as these require access to the operating system. See the sections on Unsupported C Constructs and Unsupported C++ Constructs in the *Vivado High-Level Synthesis User Guide* [Ref 8].
- Dynamic memory allocation and functions such as malloc() and free() are not allowed. All operations which result in a hardware resource, like memory, must be allocated during compile/synthesis time.
- Pointer casting in the general case is not supported: it is supported between native C types but not when using user defined types.
- The C++ Standard Template Library (STL) is not supported.
- A register or memory in hardware can be inferred from C by use of the static qualifier. See the *Coding Style Guide* in the *Vivado High-Level Synthesis User Guide* [Ref 8].

The top level CR code is shown in Figure 9 showing the top-level and sub-functions highlighted (only an overview of the sub-functions is provided).

```
//_____________________________________________________________________________
void mix (
    cdata_t din,
    cdata_t *dout_mix,
    cphase_t ph_offset) {

....

};
//_____________________________________________________________________________
void slicer (
    qam_t   qam,
    cdata_t dout_mix,
    hd_t    *hd_out,
    cdata_t *sd_out,
    error_t *err ) {

....
}

//_____________________________________________________________________________
void phase_recovery (
  hd_t       hd_out,
  cdata_t    sd_out,
  loop_int_t *loop_integ,
  loop_out_t *loop_out,
  cphase_t   *ph_offset,
  ctl_crec_t control ) {

....
}

//_____________________________________________________________________________
// crec loop
//
void crec (
    cdata_t     din,
    cdata_t    *dout_mix,
    cphase_t    ph_in,
    cphase_t   *ph_out,
    loop_int_t *loop_integ,
    ctl_crec_t  control ) {

  qam_t     qam = control.qam;
  hd_t      hd_out;
  cdata_t   sd_out;
  error_t   err;

  cphase_t   t_ph_out;
  cdata_t    t_dout_mix;
  loop_int_t t_loop_integ;
  loop_out_t t_loop_out;

  mix( din, &t_dout_mix, ph_in );

  slicer ( qam, t_dout_mix, &hd_out, &sd_out, &err);

  phase_recovery ( hd_out, sd_out, &t_loop_integ, &t_loop_out, &t_ph_out, control);

  *ph_out     = t_ph_out;
  *dout_mix   = t_dout_mix;
  *loop_integ = t_loop_integ;

}
```

Figure 9: **Top-Level of the CR Code: crec.cpp**

Arbitrary Precision

Vivado HLS provides a C++ template class which implements arbitrary precision data-types. The data-type used in this example, **ap_fixed**, has both integer and fraction components plus a variety of rounding/truncation, and saturation modes. See the chapter on C++ Arbitrary Precision Fixed Point Types in *Vivado High-Level Synthesis User Guide* [Ref 8] for more details.

Figure 10 shows an example of using this ap_fixed data type.

```
typedef ap_fixed<14,2,AP_TRN,AP_SAT> phase_t;    // phase in radian

typedef struct cphase_t {  // phase in x, y
    ap_fixed<12,1> i,q;
} cphase_t;

typedef ap_fixed<28,2> loop_int_t;
typedef ap_fixed<28,2> loop_out_t;

typedef struct hd_t {
  ap_fixed<5,1> i,q;
} hd_t;

typedef ap_fixed<16,1,AP_RND_INF,AP_SAT_SYM> cos_t;
typedef ap_fixed<16,1> lut_out_t;
```

Figure 10: **Examples of Arbitrary Precision Fixed Point Types (partial crec.h)**

The first data-type shown in Figure 10 defines a data type of 14-bits, of which 2-bits are integers (thus the remaining 12-bits are fractional bits) and which uses quantization mode `AP_TRN` (truncation) and an overflow mode `AP_SAT` (saturation). The underlying class

automatically handles the fractional arithmetic, quantization and overflow when variables of different specifications are used together.

The data-types can also be declared as floating point types, either single or double precision, in which case Vivado HLS assigns synthesizes a design which uses Xilinx floating point operators for the arithmetic. Floating-point operations require more hardware resources and clock cycles to compute the result. In many cases, such as this application, fixed-point types can be used to reduce the hardware and cycle costs while still achieving the required accuracy.

Synthesis Directives

The C++ code by itself describes the behavior of the algorithm but there is no notion of throughput, latency, or resources, which are integral specifications of the desired hardware.

Synthesis directive commands are used to specify those design attributes and others. See chapter 5 of *Vivado High-Level Synthesis User Guide* [Ref 8] for a full list. Only the synthesis directives used to implement the CR block are described here. These directives can be embedded in the code or kept in a separate file as Tcl commands shown in Figure 11.

```
1 ############################################################
2 ## This file is generated automatically by Vivado HLS.
3 ## Please DO NOT edit it.
4 ## Copyright (C) 2012 Xilinx Inc. All rights reserved.
5 ############################################################
6 set_directive_allocation -limit 1 -type operation "mix" mul
7 set_directive_interface -mode ap_none -register "crec" dout_mix
8 set_directive_interface -mode ap_none -register "crec" ph_out
9 set_directive_interface -mode ap_none -register "crec" loop_integ
10 set_directive_dataflow "crec"
```

Figure 11: **Synthesis Directives for crec.cpp (directives.tcl)**

As seen from the list of synthesis directives, it does not take much effort to optimize the implementation of the CR block to achieve the throughput goal of 12 clock cycles with reasonable resources.

Each directive is described below:

- Line 6: The `set_directive_allocation` command limits the use of multipliers in the "mix" function to only a single multiplier. This function is essentially a complex multiplication (Figure 12). Without this directive, this function is implemented using 4 multipliers: by default, Vivado HLS seeks to minimize the latency. By using only 1 multiplier, the latency of this module increases to 4 but even with this increase, the overall throughput goal of 12 is still met.

```
//_____________________________________
// apply phase correction
void mix (
    cdata_t din,
    cdata_t *dout_mix,
    cphase_t ph_offset) {

    cdata_t t_dout;

    t_dout.i =  din.i * ph_offset.i;
    t_dout.i -= din.q * ph_offset.q;
    t_dout.q =  din.i * ph_offset.q;
    t_dout.q += din.q * ph_offset.i;

    *dout_mix = t_dout;

};
```

Figure 12: **C code for the mix function (crec.cpp)**

- Lines 7-9: The `set_directive_interface` command specifies that the RTL ports to implement. These function arguments should be registered and use I/O protocol **ap_none**. By default, output ports for pointers have an associated output valid signal: the ap_none protocol ensures there is only a data output port, and thus eliminates the valid signal associated with each output. Because they are all registered, it is safe to strobe

them with `ap_done` signal (which indicates when the function has completed). Figure 13 shows these ports highlighted in the output RTL.

```
module crec (
        din_i_V,
        din_q_V,
        dout_mix_i_V,
        dout_mix_q_V,
        ph_in_i_V,
        ph_in_q_V,
        ph_out_i_V,
        ph_out_q_V,
        loop_integ_V,
        control_qam_V,
        control_lf_p,
        control_lf_i,
        control_lf_out_gain,
        control_reg_clr,
        control_reg_init_V,
        ap_clk,
        ap_rst,
        ap_start,
        ap_done,
        ap_idle,
        ap_ready
);
```

Figure 13: **IO ports shown in Vivado HLS RTL (Verilog example: rec.v)**

- Line 10: The `set_directive_dataflow` command improves the overall throughput of the CR block by enabling concurrent execution of the sub-functions at the top level. Conceptually, it is similar to pipelining except that it works on the function level. The example in Figure 14 shows how concurrent operation of the functions improves both when the top-level function can start a new operation (every three cycles versus every eight cycles) and the time it take to create an output (five versus eight clock cycles).

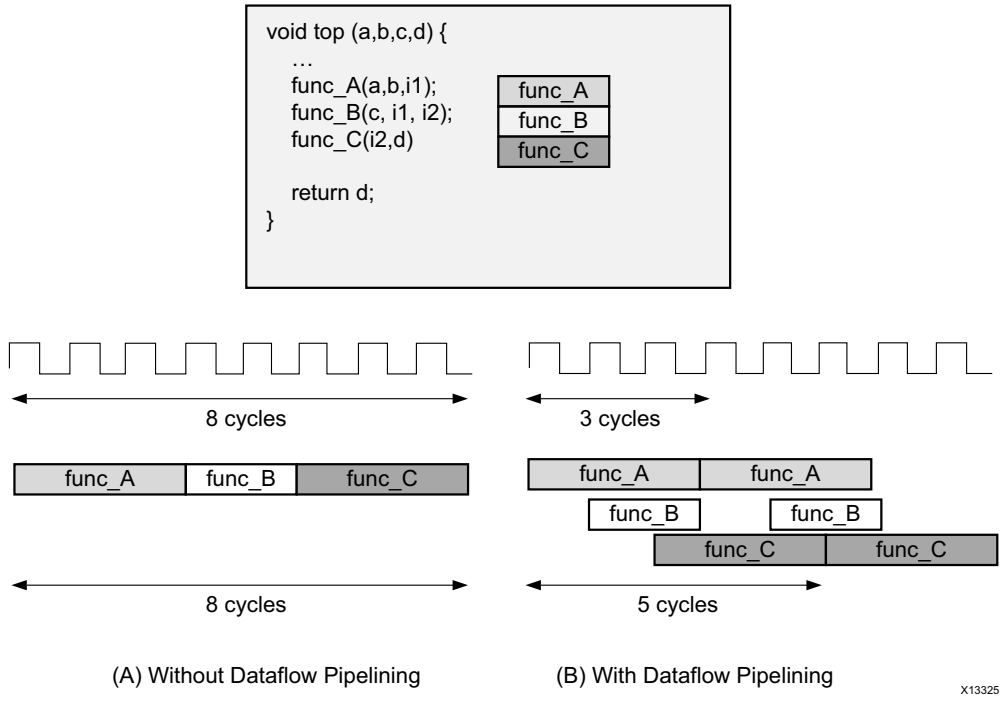


Figure 14: **Dataflow Optimization Example**

Dataflow optimization can be explained in more detail by addressing it's implications on the CR function. With respect to the throughput of the CR module, Figure 15 shows multiple executions of the three sub-functions: "mix" then "slicer" then "phase recovery".

The overall operation is dominated by the longest latency, from the "phase recovery" function. Without dataflow optimization, the next invocation of the "mix" function could not start until the "phase recovery" function is finished. By using the dataflow directive, concurrent operation is

enabled, the "mix" operation/function can start before the phase recovery operation is complete and an initiation internal (II) between new inputs of 12 clock cycles is achieved.

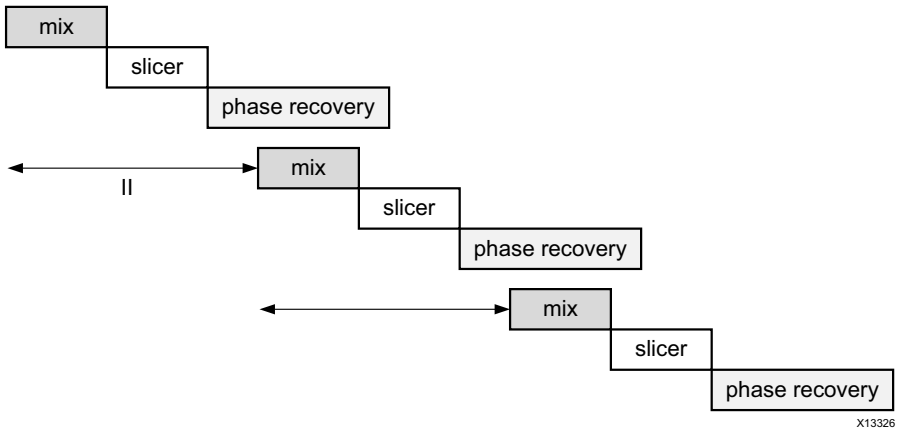
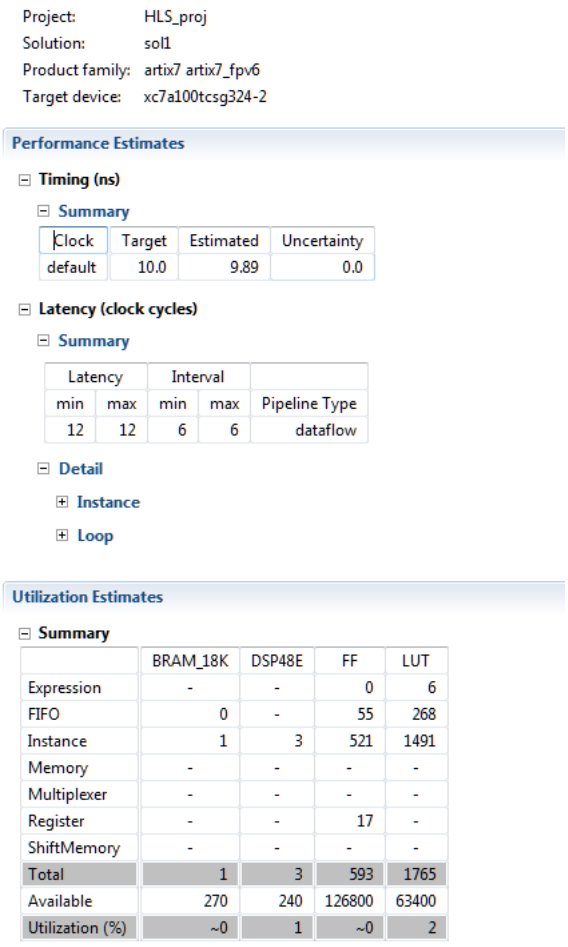


Figure 15: **Dataflow Pipeline Behavior of the CR Block**

Synthesis Report

At the end of C to RTL synthesis, the synthesis report, which shows the estimated timing, latency/throughput and resource utilization is generated (Figure 16).

The C to RTL synthesis typically takes a few minutes and the overall optimization process, which include creating a new solution, setting any new synthesis directives, synthesizing to RTL, and checking the results, is relatively fast.



Project: HLS_proj
Solution: sol1
Product family: artix7 artix7_fpv6
Target device: xc7a100tcsg324-2

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.0	9.89	0.0

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Pipeline Type
12	12	6	6	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	6
FIFO	0	-	55	268
Instance	1	3	521	1491
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	17	-
ShiftMemory	-	-	-	-
Total	1	3	593	1765
Available	270	240	126800	63400
Utilization (%)	~0	1	~0	2

Figure 16: **HLS Report for the CR Block**

There is a strong correlation between the reported estimates and the results after the FPGA implementation. The results of HLS are estimates and factors such as net delays and logic-level optimizations (LUT packing, register balancing) cannot be fully known until RTL synthesis is performed and the design placed and routed on the FPGA.

Simulation

The RTL generated from the C to RTL synthesis process can be simulated using the same C test bench within the Vivado HLS environment. There are various simulator supported (ISim, XSim, ModelSim, VSC). See *Vivado High-Level Synthesis User Guide* [Ref 8] for a complete list. You can also turn on the **Dump Trace** option to generate the simulation dump file — such as VCD file — to view the waveform later.

Figure 17 shows the RTL Co-simulation dialog box and the RTL simulation report. The dialog box shows Verilog RTL selected for simulation with the Xsim simulator and the Dump Trace option selected. The simulation report confirms the latency, initiation interval and pass/fail result for Verilog.

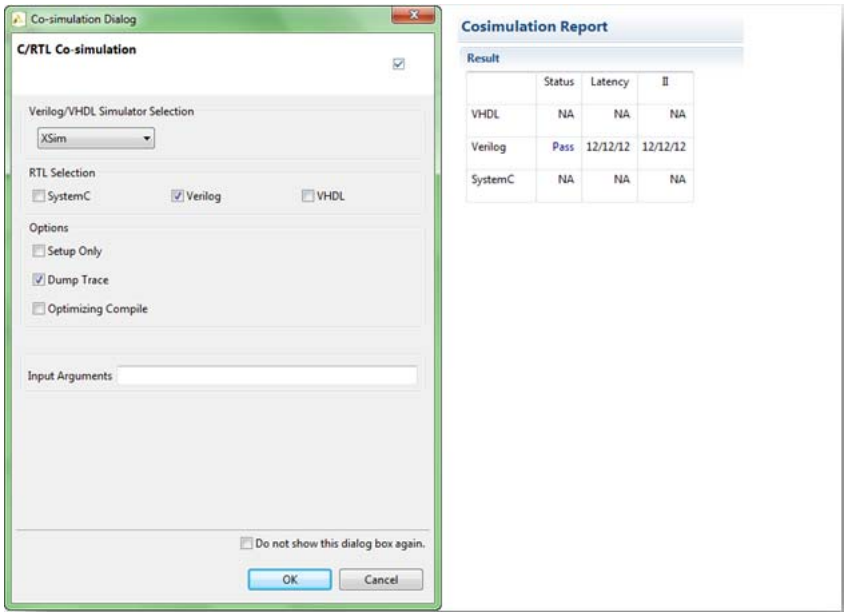


Figure 17: **RTL Co-simulation Dialog Box and Simulation Results**

Figure 18 shows an example trace file from the RTL co-simulation. In this case, the RTL ports `dout_mix_q_V` and `dout_mix_i_V` are highlighted and are valid when the `ap_done` signal is asserted.

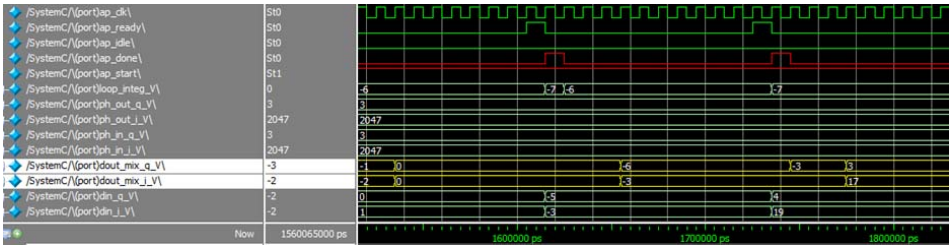


Figure 18: **Example Trace File showing Co-simulation Results**

Optimization

Using optimizations to achieve the design goals is the critical part in using Vivado HLS in the design process. Here are some general guidelines:

- Understand the default behavior of the tool (for example, for loop results in a serial implementation).
- How to use synthesis directives: these control the number/type of resource used, how loops are implemented (serial/parallel implementation), what I/O ports and protocols are used in the RTL (RAM, FIFO or scalar interface), and can specify explicit targets for the latency and throughput.
- Utilize **Analysis Perspective** (Figure 19), which graphically shows in which states operations where scheduled and how the resources are used (shared or not). The operations shown in the **Analysis Perspective** can be cross-referenced to the source code.
- If necessary, to understand the behavior of the RTL design better, examine the simulation waveform from RTL co-simulation.
- Generate a comparison report (Figure 20) to compare different implementations (different directives, clock rates, devices, and so on). A comparison report can be generated automatically by **Comparison Report** on the GUI toolbar.

Figure 19 shows the Analysis Perspective.

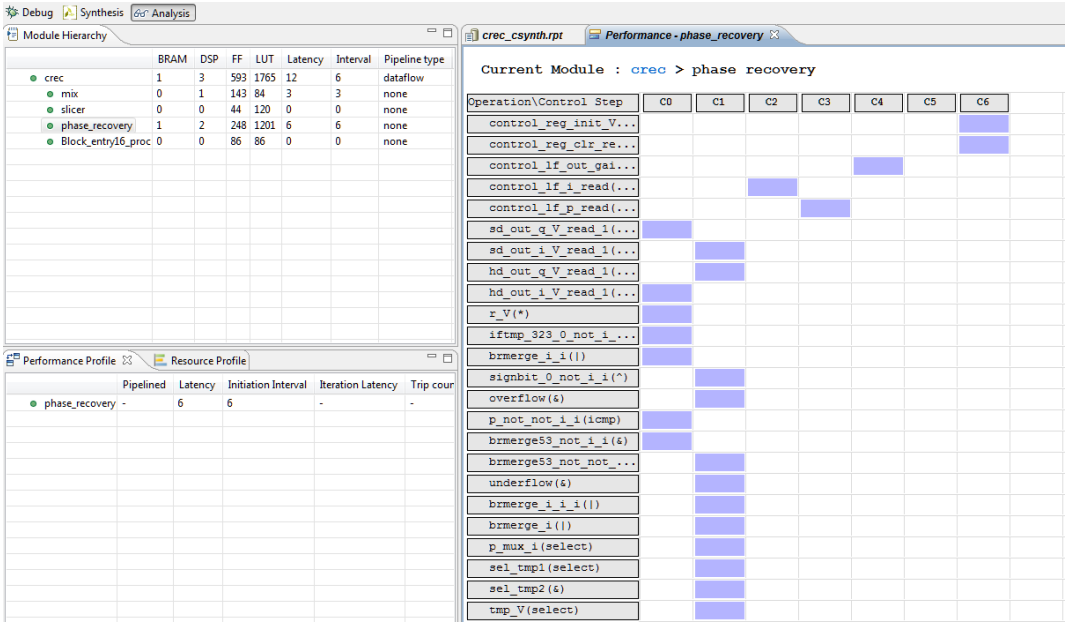


Figure 19: **The Analysis Perspective**

Figure 20 shows an example of the comparison report where three different solutions are compared. The difference in the fastest clock rate, latency, interval as well as the number of BRAMs, DSP48s, LUTs, and flip-flops can be reviewed to determine the most optimal solution.

Vivado HLS Report Comparison

All Compared Solutions

sol1 : xc7a100tcsg324-2
sol2 : xc7a100tcsg324-2
sol3 : xc7a100tcsg324-2

Performance Estimates

Timing (ns)

Clock		sol1	sol2	sol3
default	Target	10.0	10.0	10.0
	Estimated	9.89	9.89	9.89

Latency (clock cycles)

		sol1	sol2	sol3
Latency	min	12	13	10
	max	12	13	10
Interval	min	6	14	11
	max	6	14	11

Utilization Estimates

	sol1	sol2	sol3
BRAM_18K	1	1	1
DSP48E	3	3	6
FF	593	469	345
LUT	1765	1337	1255

Export the report(.html) using the Export Wizard

Figure 20: **Comparison Report: Three Solutions**

FPGA Implementation

After the design goals are met, the produced RTL can be packaged into an IP block for use in other Xilinx design tools, such as Vivado IP catalog, System Generator for DSP, or the Xilinx ISE environment.

During the IP packaging process, the results of the FPGA implementation can optionally be evaluated by selecting **Evaluate**. This evaluation options launches RTL synthesis and P&R from within Vivado HLS and provides the final implementation results. These implementation results are not part of the IP package but simply allows you to confirm that the IP meets the post-implementation timing and resource estimates before releasing the IP package.

Figure 21 shows dialog box for packing the final design into an IP block for use in System Generator.

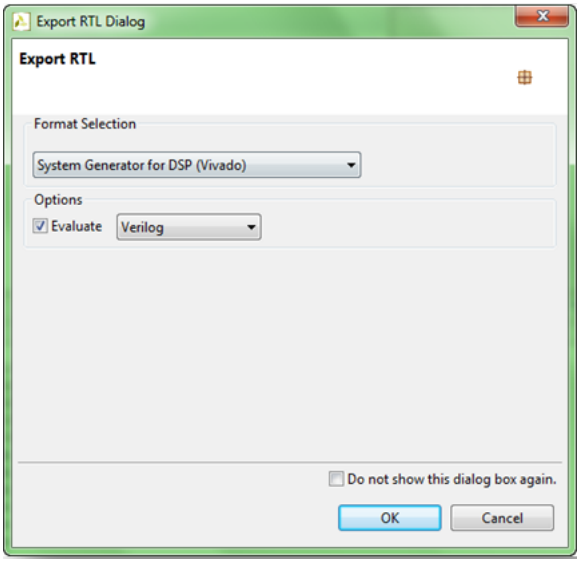


Figure 21: **Packaging the IP for System Generator**

Figure 22 shows the results after the evaluation step. Here, the actual number of resources which are used (including slices and SRL) are listed along with the final timing after place and route.

```
Implementation tool: Xilinx Vivado v2012.4
Device target:       xc7a100tcsg324-3
Report date:         Wed Jan 30 19:37:52 EST 2013

#=== Resource usage ===
SLICE:          304
LUT:            987
FF:             322
DSP:              5
BRAM:             2
SRL:             86
#=== Final timing ===
CP required:    10.000
CP achieved:    7.149
Timing met
```

Figure 22: **Results from RTL Evaluation**

Verification in System Generator For DSP

After the Vivado HLS synthesized design has been packaged as a block for System Generator [Ref 9], it can be instantiated seamlessly into the MathWorks Simulink model-based design environment, simulated, and implemented with the remainder of the System Generator blocks.

The System Generator environment is an ideal place to integrate all the modules from various sources (System Generator for DSP blockset, MathWorks HDL coder generated RTL, custom RTL, subset of m-code, etc.) and co-simulate and generate the top level design.

To add the Vivado HLS module to System Generator, instantiate a Vivado HLS block from the System Generator toolbox. Double-click the **Vivado HLS** block to open the Vivado HLS dialog box (Figure 23) and specify the location of the Vivado HLS solution directory.

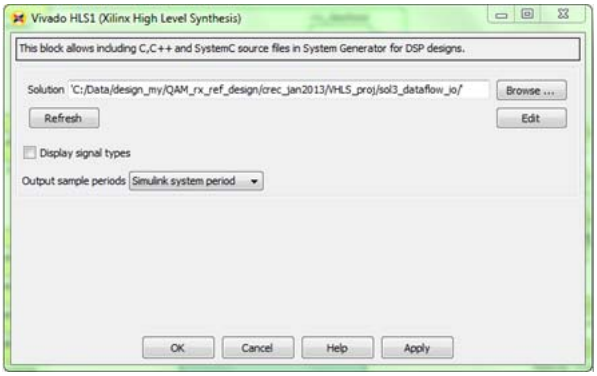


Figure 23: **System Generator Vivado HLS Dialog Box**

The IP block can then be connected to the rest of the system, as shown in Figure 24.

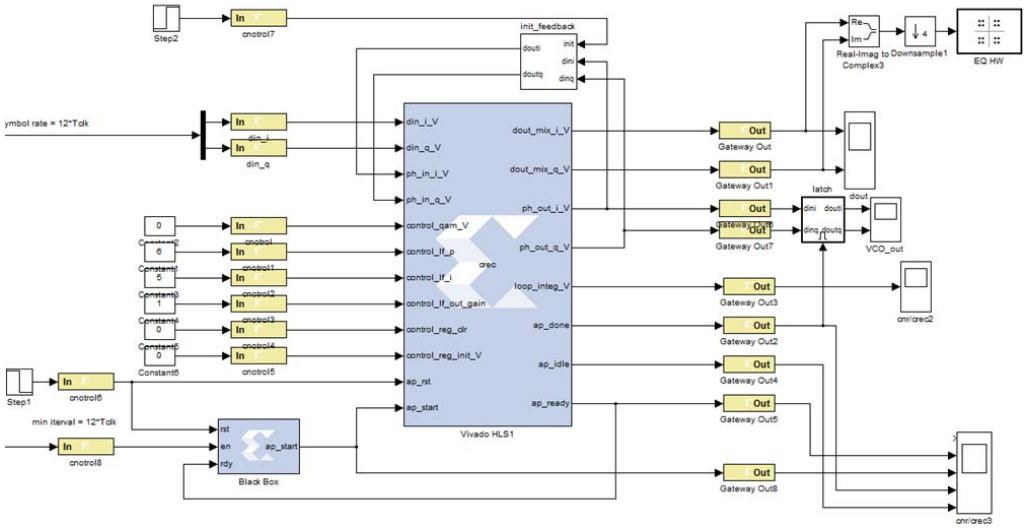


Figure 24: **CR Block inside System Generator**

Taking advantage of the System Generator simulation environment, results can easily be generated to verify the output of the VCO and loop filter (Figure 25) and confirm the expected behavior of carrier phase recovery loop (Figure 26).

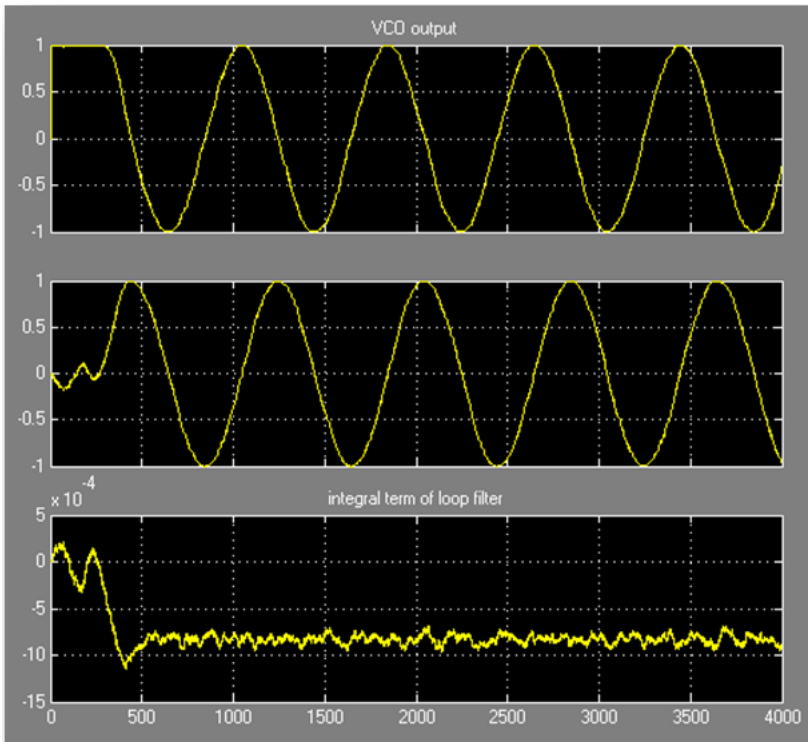


Figure 25: **Simulation Output (VCO and Loop Filter) in System Generator**

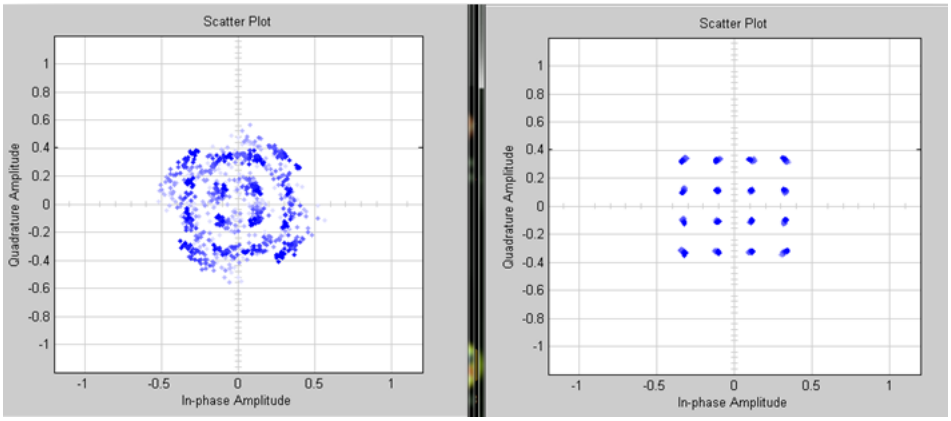


Figure 26: **Constellation Output (Input/Output of the CR Block) in System Generator**

Design Source Files

Version of the tool: Vivado Design Suite tool, Vivado HLS, System Generator For DSP Vivado version, all in 2013.1 version.

The test case for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=343245

The list of design files and directories is as follows:

- src: has all the C++ sources (crec.cpp has all the functions), header file (`crec.h`), test bench (`crec_tb.cpp`).
- VHLS_proj: is the directory for VHLS project. There are several solutions. One that is presented in this application note is "`sol3_dataflow_io`".
- System Generator For DSP: has a System Generator For DSP top level model (`crec_v1a.mdl`) containing the CR module from VHLS.

Conclusion

Even though the benefits of a HLS design methodology has been well known, the use of the HLS tool has not been widely adopted largely due to its quality of results in utilized resource and speed. The quality of results and the amount of effort to reach those design goals and the ease of integrating the results into the rest of the Xilinx design flow are what distinguish Vivado HLS from a lot of HLS tools of the past, as seen from this carrier phase recovery loop design.

The design process of taking the carrier phase recovery loop algorithm written in C++, synthesized and optimized by Vivado HLS, simulated and verified in System Generator and implemented into an FPGA is described in detail in this application note.

References

1. *FIR Compiler v6.3* (DS795).
2. *DDS Compiler v5.0* (DS794).
3. F. Gardner, “Interpolation in digital modems – part 1, fundamentals,” *IEEE Transactions on Communications,* Vol. 41, 1993.
4. L. Erup, Interpolation in digital modems – part 2, implementation and performance,”, *IEEE Transactions on Communications,* Vol. 41, 1993.
5. Mengali, *Synchronization Techniques in Digital Receiver.*
6. J Yang, “The multimodulus blind equalizer and its generalized algorithms,” *IEEE Journals on selected areas on communication*, Vol. 20, No. 5, 2002.
7. G Long, “The LMS algorithm with delayed coefficient adaptation,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, Issue 9, 1989.
8. *Vivado High-Level Synthesis User Guide* (UG902).
9. *System Generator for DSP User Guide* (UG640).
10. H Besbes, A. Paek, “A low complexity blind QAM receiver,” *IEEE GLOBCOM*, Vol. 4, 2003.

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
05/02/2013	1.0	Initial Xilinx release.

Notice of Disclaimer