Four Best Practices for Prototyping MATLAB and Simulink Algorithms on FPGAs

by Stephan van Beek, Sudhir Sharma, and Sudeepa Prakash, MathWorks

Chip design and verification engineers often write as many as ten lines of test-bench code for every line of RTL code that is implemented in silicon. They can spend 50% or more of the design cycle on verification tasks. Despite this level of effort, nearly 60% of chips contain functional flaws and require re-spin[i]. Because HDL simulation is not sufficient to catch system-level errors, chip designers now employ FPGAs to accelerate algorithm creation and prototyping.

Using FPGAs to process large test data sets enables engineers to rapidly evaluate algorithm and architecture tradeoffs quickly. They can also test designs under real-world scenarios without incurring the heavy time penalty associated with HDL simulators. System-level design and verification tools such as MATLAB® and Simulink® help engineers realize these benefits by rapidly prototyping algorithms on FPGAs.

This article describes best practices for creating FPGA prototypes with MATLAB and Simulink. The best practices are listed below and highlighted in Figure 1.

(1) Analyze the effects of fixed-point quantization early in the design process and optimize the word length to yield smaller and more power-efficient implementations
(2) Use automatic HDL code generation to produce FPGA prototypes faster
(3) Reuse system-level test benches with HDL co-simulation to analyze HDL implementations using system-level metrics
(4) Accelerate verification with FPGA-in-the-loop simulation

WHY PROTOTYPE ON FPGAS?

Prototyping algorithms on FPGAs gives engineers increased confidence that their algorithm will behave as expected in the real world. In addition to running test vectors and simulation scenarios at high speed, engineers can use FPGA prototypes to exercise software functionality and adjacent system-level functions, such as RF and analog subsystems. Moreover, because FPGA prototypes run faster, larger data sets can be used, potentially exposing bugs that would not be uncovered by a simulation model.

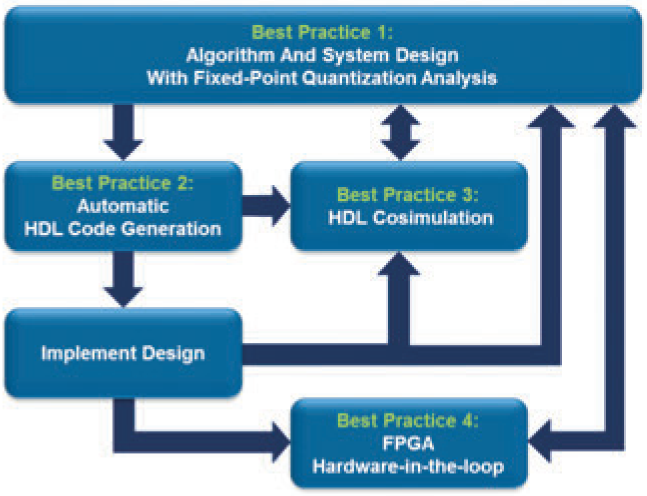


Figure 1 - Model-Based Design best practices for FPGA prototype development.

Model-Based Design using HDL code generation enables engineers to efficiently produce FPGA prototypes, as shown in Figure 2. This figure illustrates the practical reality that engineers often abbreviate the detailed design phase in an attempt to begin the hardware development phase to meet development schedules. In practice, engineers will revisit the detailed-design phase during the HDL creation phase as they discover that the fixed-point algorithm is not meeting system requirements. This overlap contributes to an elongated HDL creation phase as depicted by the long purple bar and may result in design compromises, such as glue logic or design patches.

Because automatic HDL code generation is a faster process than hand-coding, engineers can invest some of the time savings to produce higher quality fixed-point algorithms in the detailed design phase. This approach enables engineers to produce higher quality FPGA prototypes faster than with a manual workflow.

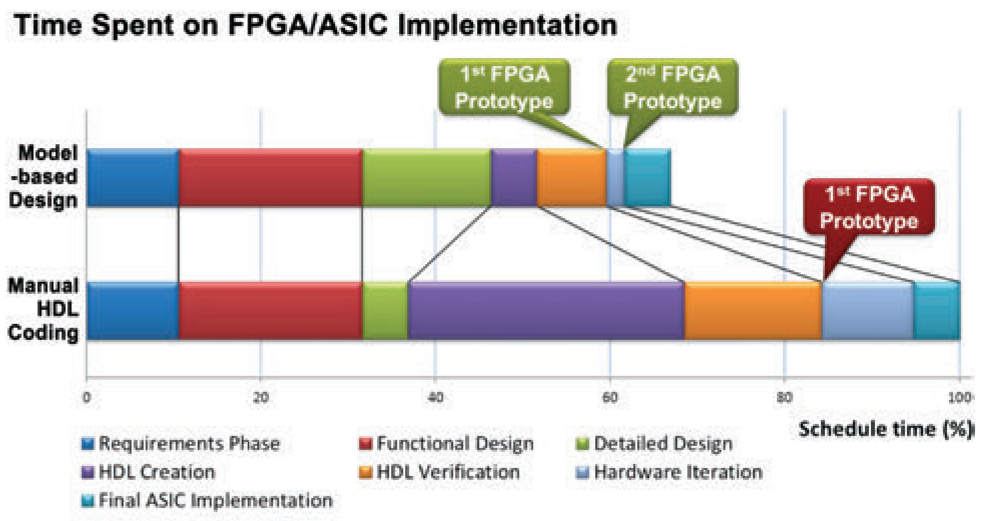


Figure 2 - Comparison of Model-Based Design and manual workflow timelines for FPGA prototyping and ASIC implementation.

DIGITAL DOWN CONVERTER CASE STUDY

To illustrate best practices for FPGA prototyping using Model-Based Design, a digital down converter (DDC) serves as a useful case study. A DDC is a common building block in many communications systems (see Figure 3). It is used to transform high-rate passband input into low-rate baseband output, which can be processed using lower sample rate clocks. This results in lower-power, lower-resource hardware implementation.

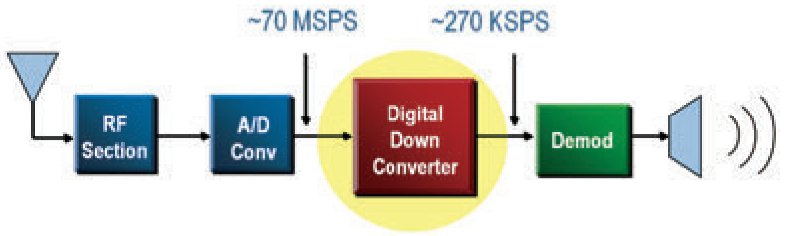


Figure 3 - Communications system employing a digital down converter.

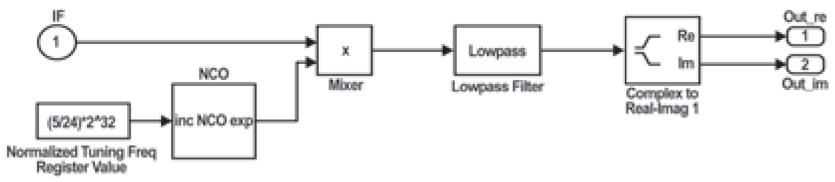


Figure 4 - System model of a digital down converter.

The main components of a DDC are those shown in Figure 4:

- Numerical controlled oscillator (NCO)
- Mixer
- Digital filter chain

BEST PRACTICE #1 – ANALYZE THE EFFECT OF FIXED-POINT QUANTIZATION EARLY IN THE DESIGN PROCESS

Engineers typically test new ideas and develop initial algorithms using floating-point data types. Hardware implementation in FPGAs and ASICs, however, requires conversion to fixed-point data types, which often introduces quantization errors. In a manual workflow, fixed-point quantization is usually performed during the HDL coding process. In this workflow, an engineer cannot easily quantify the effect of fixed-point quantization by comparing the fixed-point representation to a floating-point reference. Nor is it easy to analyze the HDL implementation for overflows.

To make intelligent decisions on the required fraction lengths, engineers need a way to compare the floating-point simulation results against fixed-point simulation results *before* starting the HDL coding process. Increasing the fraction length reduces quantization errors;

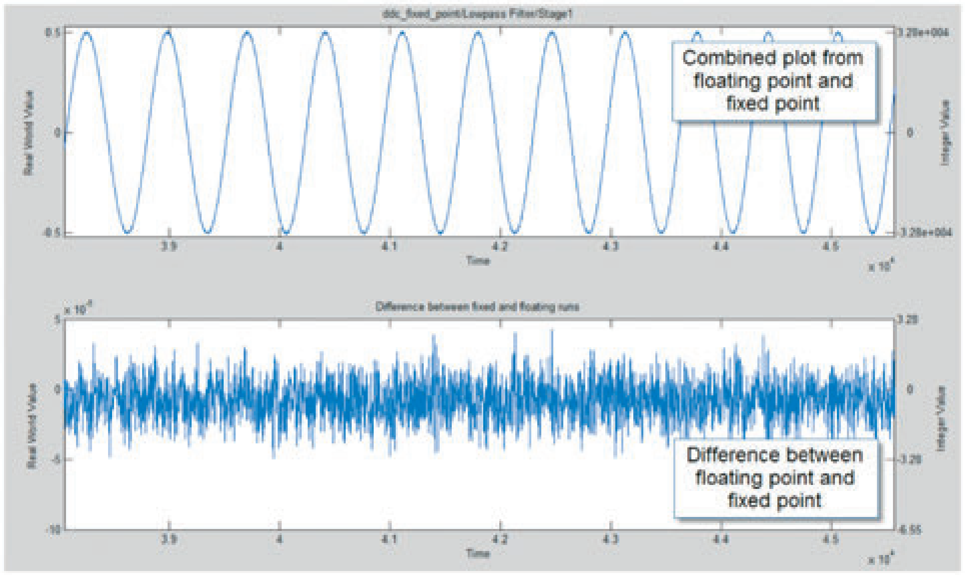
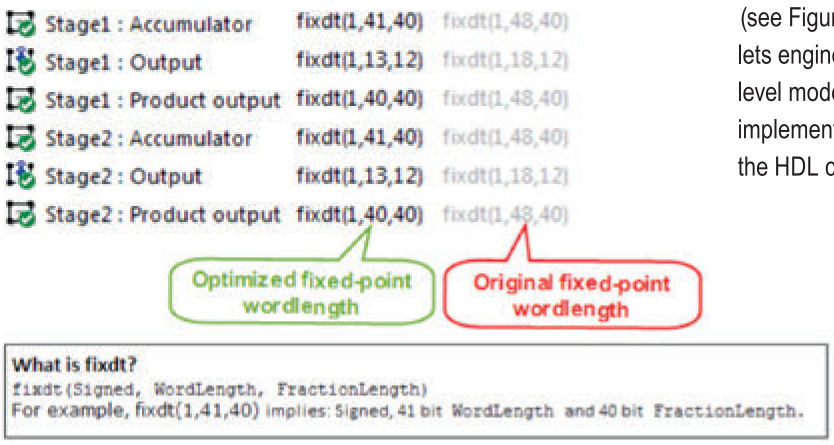


Figure 5 - Quantifying the effect of fixed-point quantization using Simulink Fixed Point.

however, such increases mean that word length needs to be increased (for more area and more power consumption).

For example, Figure 5 illustrates the differences between the floating-point and fixed-point simulation results for stage one of the low-pass filter in the DDC filter chain. These differences are due to fixed-point quantization. The top graph shows an overlay of floating-point and fixed-point simulation results. The bottom graph shows the quantization error at every point in the plot. Depending on the design specification, engineers may need to increase fraction lengths to reduce the introduced quantization error.



Stage1 : Accumulator	fixdt(1,41,40)	fixdt(1,48,40)
Stage1 : Output	fixdt(1,13,12)	fixdt(1,18,12)
Stage1 : Product output	fixdt(1,40,40)	fixdt(1,48,40)
Stage2 : Accumulator	fixdt(1,41,40)	fixdt(1,48,40)
Stage2 : Output	fixdt(1,13,12)	fixdt(1,18,12)
Stage2 : Product output	fixdt(1,40,40)	fixdt(1,48,40)

Optimized fixed-point wordlength
Original fixed-point wordlength

What is fixdt?

```
fixdt(Signed, WordLength, FractionLength)
```

For example, fixdt(1,41,40) implies: Signed, 41 bit WordLength and 40 bit FractionLength.

In addition to selecting a fraction length, engineers must optimize the word length to achieve low-power and area-efficient designs.

In the DDC case study, engineers use Simulink Fixed Point™ to reduce the word length of parts of the digital filter chain by as many as 8 bits (see Figure 6).

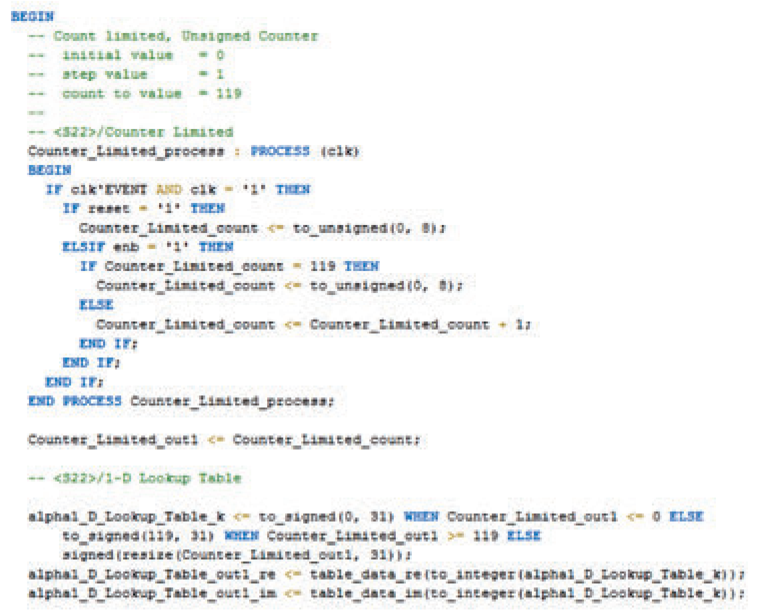
BEST PRACTICE #2 – USE AUTOMATIC HDL CODE GENERATION TO PRODUCE FPGA PROTOTYPES FASTER

HDL code is required to produce an FPGA prototype. Engineers have written Verilog or VHDL code by hand. As an alternative, generating HDL code automatically using HDL Coder™ offers several important benefits. Engineers can:

- Quickly assess if the algorithm can be implemented in hardware
- Rapidly evaluate different algorithm implementations and choose the best one
- Prototype algorithms on FPGAs faster

For the DDC case study, we generated 5,780 lines of HDL code within 55 seconds. Engineers can read and readily understand the code (see Figure 7). Automatic code generation lets engineers make changes in the system-level model, and produce an updated HDL implementation in minutes by regenerating the HDL code.

Figure 6 - Optimizing fixed-point data types using Simulink Fixed Point.



```
BEGIN
  -- Count limited, Unsigned Counter
  --  initial value   = 0
  --  step value      = 1
  --  count to value  = 119
  --
  -- <S22>/Counter Limited
  Counter_Limited_process : PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      IF reset = '1' THEN
        Counter_Limited_count <= to_unsigned(0, 8);
      ELSIF enb = '1' THEN
        IF Counter_Limited_count = 119 THEN
          Counter_Limited_count <= to_unsigned(0, 8);
        ELSE
          Counter_Limited_count <= Counter_Limited_count + 1;
        END IF;
      END IF;
    END IF;
  END PROCESS Counter_Limited_process;

  Counter_Limited_out1 <= Counter_Limited_count;

  -- <S22>/1-D Lookup Table

  alpha1_D_Lookup_Table_k <= to_signed(0, 31) WHEN Counter_Limited_out1 <= 0 ELSE
      to_signed(119, 31) WHEN Counter_Limited_out1 >= 119 ELSE
      signed(resize(Counter_Limited_out1, 31));
  alpha1_D_Lookup_Table_out1_re <= table_data_re(to_integer(alpha1_D_Lookup_Table_k));
  alpha1_D_Lookup_Table_out1_im <= table_data_im(to_integer(alpha1_D_Lookup_Table_k));
```

Figure 7 - Sample of HDL code generated using HDL Coder with Simulink.

BEST PRACTICE #3 – REUSE SYSTEM LEVEL TEST BENCHES WITH CO-SIMULATION FOR HDL VERIFICATION

Functional Verification

HDL co-simulation enables engineers to reuse Simulink models to drive stimuli into the HDL simulator and perform system-level analysis of the simulation output interactively (Figure 8).

While HDL simulation provides only digital waveform output, HDL co-simulation provides complete visibility into the HDL code, as well as access to the full suite of system-level analysis tools of Simulink. When engineers observe a difference between expected results and HDL simulation results, co-simulation helps them to better understand the system-level effect of the mismatch.

For example, in Figure 9, on the following page, the spectrum scope view enables an engineer to make an informed decision to ignore the mismatch between the expected results and HDL simulation results because the differences lie in the stop-band. The digital waveform output, in contrast, merely flags the mismatch in expected results and HDL simulation results as an error. The engineer might eventually arrive at the same conclusion, but it would take more time to complete the required analysis.

Test Coverage

Engineers can use HDL Verifier, Simulink Design Verifier, and Questa /ModelSim to automate code coverage analysis. In this workflow, Simulink Design Verifier produces a suite of test cases for model coverage. HDL Verifier automatically runs Questa / ModelSim with this test suite to gather code coverage data for a complete analysis of the generated code.

BEST PRACTICE #4 – ACCELERATE VERIFICATION WITH FPGA-IN-THE-LOOP SIMULATION

Having verified the DDC algorithm using system-level simulations and HDL co-simulations, you can now deploy the DDC algorithm on an FPGA target platform. FPGA-based verification (also referred to as FPGA-in-the-loop simulation) of the algorithm increases confidence that the algorithm will work in the real world. This enables engineers to run test scenarios faster than with host-based HDL simulation.

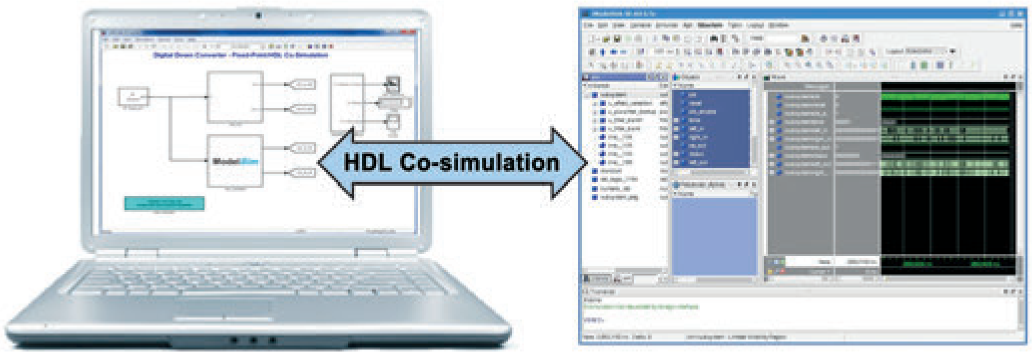
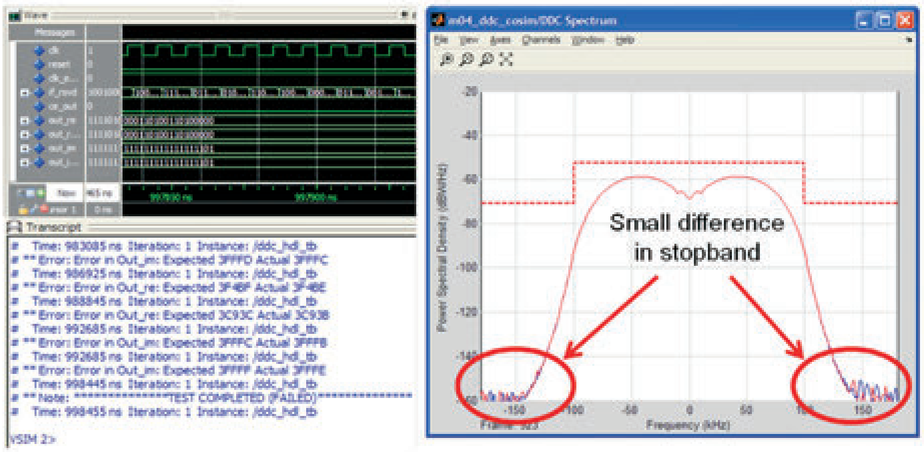


Figure 8 - HDL co-simulation between Simulink and Questa/ ModelSim.



```
#   Time: 983085 ns  Iteration: 1  Instance: /ddc_hdl_tb
# ** Error: Error in Out_im: Expected 3FFFD Actual 3FFFC
#   Time: 986925 ns  Iteration: 1  Instance: /ddc_hdl_tb
# ** Error: Error in Out_re: Expected 3F4BF Actual 3F4BE
#   Time: 988845 ns  Iteration: 1  Instance: /ddc_hdl_tb
# ** Error: Error in Out_re: Expected 3C93C Actual 3C93B
#   Time: 992685 ns  Iteration: 1  Instance: /ddc_hdl_tb
# ** Error: Error in Out_im: Expected 3FFFC Actual 3FFFB
#   Time: 992685 ns  Iteration: 1  Instance: /ddc_hdl_tb
# ** Error: Error in Out_im: Expected 3FFFF Actual 3FFFE
#   Time: 998445 ns  Iteration: 1  Instance: /ddc_hdl_tb
# ** Note: ***************TEST COMPLETED (FAILED)***************
#   Time: 998455 ns  Iteration: 1  Instance: /ddc_hdl_tb

VSIM 2>
```

Figure 9 - Using a domain-specific scope to analyze system-level metrics and assess behavior of HDL implementation.

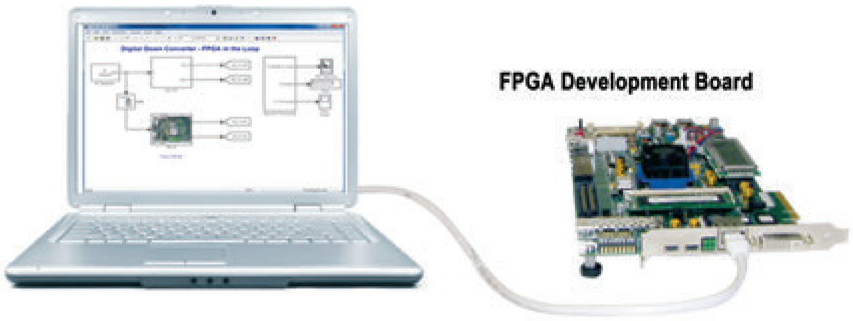


Figure 10 - FPGA-in-the-loop simulation using Simulink and FPGA hardware.

For the DDC algorithm, you use a Simulink model to drive FPGA input stimuli and to analyze the output of the FPGA (Figure 10). As with HDL co-simulation, the results are always available in Simulink for analysis.

Figure 11 compares the two verification methods, HDL co-simulation and FPGA-in-the-loop simulation, used for the DDC design. In this case, FPGA-in-the-loop simulation is 23 times faster than HDL co-simulation. Such speed increases enable engineers to run more extensive sets of test cases and to perform regression tests on their designs. This lets them identify potential problem areas that need more detailed analysis.

Although it is slower, HDL co-simulation provides more visibility into the HDL code. It is, therefore, well suited for more detailed analysis of the problem areas found during FPGA-in-the-loop simulation.

SUMMARY

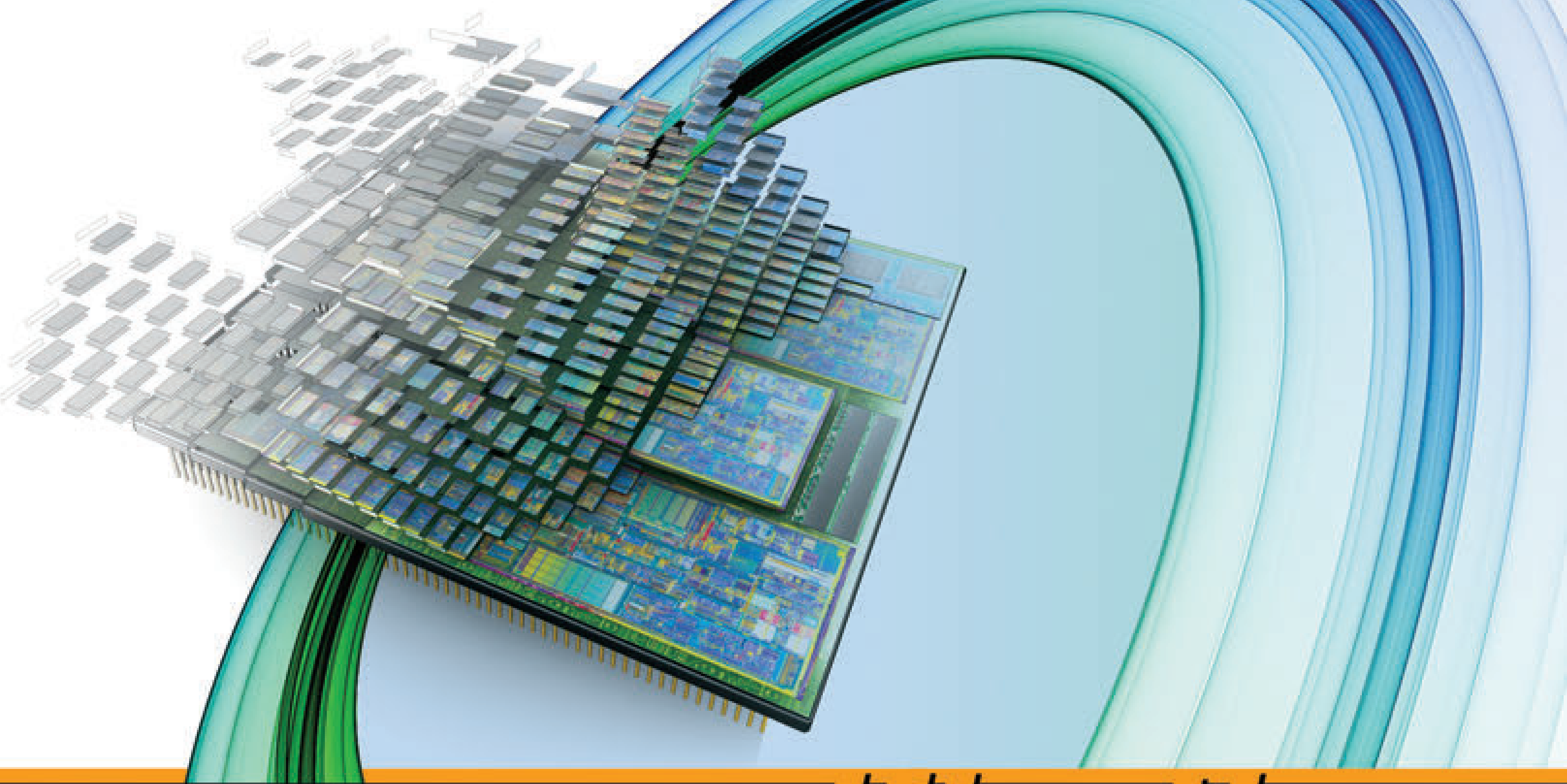
Following the four best practices outlined in this article enables engineers to develop FPGA prototypes much faster and with a greater degree of confidence than a traditional, manual workflow. In addition, engineers can continue to refine their models throughout development and rapidly regenerate code for FPGA implementation. This capability enables much shorter design iterations than a traditional workflow that relies on hand-written HDL. To learn more about the workflow outlined here or to download a technical kit, visit www.mathworks.com/programs/techkits/techkit_asic_response.html

ENDNOTES

[1] Hardware/Software Co-verification, by Dr. Jack Horgan. http://alturl.com/gfzsf

Verification Method	Simulation Performance	Visibility into HDL Code	System-Level Analysis Possible
HDL Co-simulation	46 sec	Yes	Yes
FPGA-in-the-Loop	2 sec	No	Yes

Figure 11 - HDL co-simulation vs. FPGA-in-the-loop for DDC design verification.



verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Mentor Graphics®

www.mentor.com