Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 4—Rapid Prototyping Using the Zynq SDR Kit and Simulink Code Generation Workflow

By Mike Donovan, Andrei Cozma, and Di Pu

Introduction

The previous parts of this article series introduced the Zynq SDR rapid prototyping platform,[1] presented the steps of using MATLAB and Simulink to develop an algorithm that can successfully process and decode ADS-B transmissions,[2] and showed how to verify the algorithm both in simulation and with live data acquired from the SDR platform.[3] The ultimate goal of all stages is to create a verified model that can be translated into C and HDL code and is ready to be integrated in the SDR platform's software and hardware infrastructure.

The Simulink model discussed in Part 2 of the series ("Mode S Detection and Decoding Using MATLAB and Simulink")[2] is a simulation model with enough hardware specific fidelity to verify that the design will successfully decode ADS-B messages. Using that model as a starting point, the final steps required to produce a working receiver design that runs on the Zynq SDR Rapid Prototyping Platform will be discussed. As in the previous articles in this series, the skills needed to develop this working design include: proficiency in MATLAB and Simulink, knowledge of the Zynq radio hardware, and software/hardware integration skills.

The steps to follow in this article include:

- Partition the Simulink model into functions that will target the FPGA fabric and the ARM® processing system on the Zynq SoC.
- Introduce design changes to the Simulink model to improve the performance of the generated HDL code.
- Generate the source HDL and C code for the ADS-B receiver algorithm.
- Integrate the generated source code in the Zynq radio platform design.
- Test the embedded design on the target hardware with live aircraft signals.

At the end of this process, a fully verified SDR system will be produced, running C and HDL code automatically generated from a Simulink ADS-B model and receiving and decoding live commercial aircraft signals in real time.

Partitioning a Model into Hardware and Software Components

The first step in the process of generating the implementation code is to partition the design into the functionality that will run on the programmable logic and the ARM processing system of the Zynq SoC.

Partitioning usually begins by identifying the processing requirements of the different components of the design and the required execution rates and times. Components (such as data modulation/demodulation algorithms) that are computationally intensive and need to run in real time at the sample rate are best suited to be implemented in the programmable logic. Less intensive processing tasks (such as data decoding and rendering, and system monitoring and diagnosis), are better suited for software implementation. Some other aspects to consider are: the data types and complexity of the operations and the precision of the input and output data. All the operations that target the programmable logic work on fixed-point, integer, or Boolean data types. In the case of more complex operations such as trigonometric functions or square root, approximations are used to implement them efficiently using the available hardware resources. All these constraints result in precision loss that can adversely affect system functionality if not properly assessed and implemented. However, the components that target the processing system can work on floating-point numbers and implement operations of any complexity with the highest degree of fidelity, but usually at the expense of slower execution speed.

Using those constraints as a guideline, the partitioning of the ADS-B decoding algorithm is fairly obvious. The functionality in the Detector block in the ModeS_Simulink_Decode.slx model, which includes the front-end processing of the I/Q samples all the way through to the checksum computation, is well suited for implementation on the programmable logic of the Zynq SoC (Figure 1). The decoding of the message bits, which is implemented in the Modified Buffer and Decode and Display blocks, is easily implemented in the processing system.

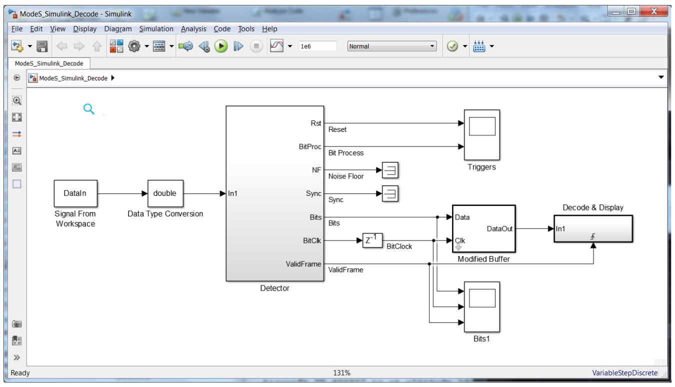


Figure 1. ModeS_Simulink_Decode.slx: FPGA and ARM processor partition.

Readers interested in following along with the Simulink model can find the files on the Analog Devices GitHub repository.[4]

Generating HDL Code from a Simulink Model

The Detector block in the Mode S Decoder model (Figure 2) is comprised of several subsystems: CalcSyncCorr, CalcNF, SyncAndControl, BitProcess, CalcCRC, and FameDetect. HDL Coder from MathWorks[5] is used to produce the source HDL code for this design.

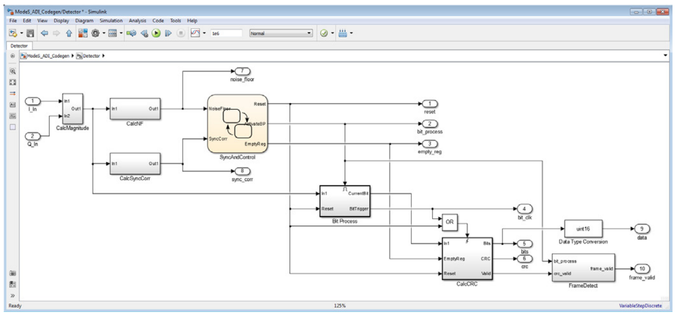


Figure 2. Detector block used for HDL code generation.

A Simulink model must satisfy several conditions to successfully generate HDL code using HDL Coder. A few of the most significant requirements are:

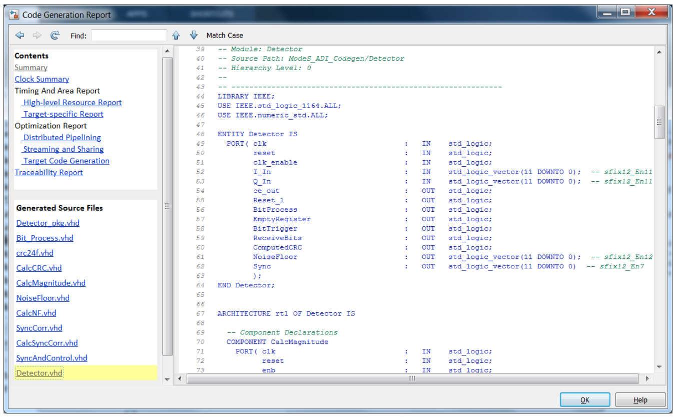
- Use blocks that support HDL code generation. HDL Coder supports code generation for approximately 200 Simulink blocks.[6] In the detector design, all the blocks, including the Stateflow diagram and the Digital Filter blocks, support HDL code generation.
- Use fixed-point data types. In the detector design, the signals use 12-bit, 24-bit, and Boolean data types. The 12-bit data type matches the bit width of the analog-to-digital converters on the Analog Devices AD9361 transceiver.
- Use scalar or vector signals. Vector signals can be used for multichannel signals or resource sharing.
- Avoid algebraic loops in the model. The HDL Coder software does not support HDL code generation for models in which algebraic loop conditions exist.

The ModeS_Simulink_Decode.slx model did not satisfy all these conditions, so the part of the CalcCRC block that compares the received bits to the computed checksum was moved outside the Detector block and ultimately implemented in C. The resulting model, ModeS_ADI_CodeGen.slx, was used to generate the HDL code. In contrast to a manual coding process, it only takes a couple minutes to generate several thousand lines of HDL code. The source code produced by HDL Coder is a bit true, cycle accurate version of the Simulink model. This is one of the major productivity gains in using model-based design; the generated code is an accurate translation of the Simulink model.

In addition, the code is designed to be readable and traceable so engineers can easily map the generated code to their design model. This is achieved in several ways (Figure 3):

- The hierarchy of the model is preserved in the HDL code files that get generated. In this example, the top level block is named Detector.vhd, and the subsystems at the next level of hierarchy are named CalcNF.vhd, Bit_Process.vhd, and so on.
- The block names, port names, signal names, data types, and complexity used in the model are preserved in the generated code.

Links between the model and source code allow a designer to click on a block in the Simulink model and automatically navigate to the generated HDL code. Similarly, there are hyperlinks in the generated code that will open the Simulink model and highlight the block associated with that segment of code.



```
-- Module: Detector
-- Source Path: ModeS_ADI_Codegen/Detector
-- Hierarchy Level: 0
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Detector IS
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        clk_enable                        :   IN    std_logic;
        I_In                              :   IN    std_logic_vector(11 DOWNTO 0);  -- sfix12_En11
        Q_In                              :   IN    std_logic_vector(11 DOWNTO 0);  -- sfix12_En11
        ce_out                            :   OUT   std_logic;
        Reset_1                           :   OUT   std_logic;
        BitProcess                        :   OUT   std_logic;
        EmptyRegister                     :   OUT   std_logic;
        BitTrigger                        :   OUT   std_logic;
        ReceiveBits                       :   OUT   std_logic;
        ComputedCRC                       :   OUT   std_logic;
        NoiseFloor                        :   OUT   std_logic_vector(11 DOWNTO 0);  -- sfix12_En12
        Sync                              :   OUT   std_logic_vector(11 DOWNTO 0)  -- sfix12_En7
        );
END Detector;


ARCHITECTURE rtl OF Detector IS

  -- Component Declarations
  COMPONENT CalcMagnitude
    PORT( clk                             :   IN    std_logic;
          reset                           :   IN    std_logic;
          enb                             :   IN    std_logic;
```

Figure 3. Source HDL code for ModeS_ADI_CodeGen.slx.

Optimizing the ADS-B Model to Produce HDL Code with a Higher Clock Speed

Although the ModeS_ADI_CodeGen.slx model successfully generates HDL code, it is rare that a designer will not want to improve the initial results. Designers typically need to meet speed and area constraints, which usually involves optimizing the initial Simulink model to achieve the desired results. A major advantage of Simulink and code generation is that the designer can make those optimizations in the model, run a simulation to ensure the changes do not break the algorithm, and then re-generate the HDL code. This is usually much simpler and less error prone than making changes in the HDL source code and potentially breaking the algorithm.

In the case of this design, the HDL code generated by the model easily fit on the available FPGA fabric, but ran at a relatively low clock rate. This is common in many initial designs. A built-in analysis tool in HDL Coder shows that the critical path in the model extended from the I/Q sample input to the first register in the CalcCRC subsystem. Inserting pipeline registers in the design is one common method to increase the clock speed (Figure 4). Pipelining shortens the path between signal operations at the expense of adding delay to the overall processing. This trade-off is usually acceptable since a slight delay is typically a small price to pay for higher clock rates.

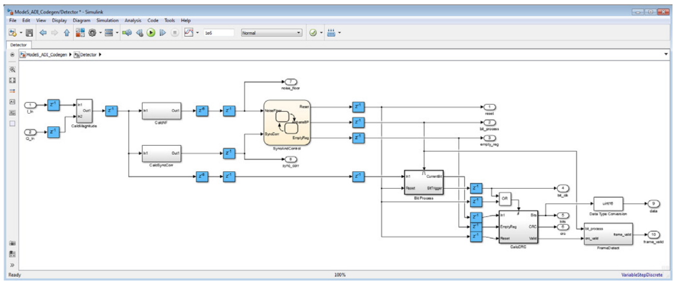
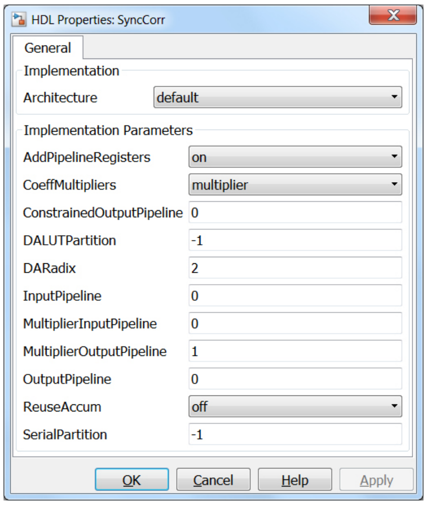


Figure 4. Pipeline registers inserted into detector design.

The pipeline registers in between the subsystems help improve the clock rate of the design, but better clock rates can be achieved by making favorable architecture choices for the Digital Filter blocks. Many of the Simulink blocks have architecture choices that enable a designer to optimize the design for speed or area. In the case of the digital filters used for the calculation of the noise floor and the preamble correlation (Figure 5), pipelining the output multipliers can shorten the critical path within the digital filter and improve the design clock rate.



Implementation

Architecture	default

Implementation Parameters

AddPipelineRegisters	on
CoeffMultipliers	multiplier
ConstrainedOutputPipeline	0
DALUTPartition	-1
DARadix	2
InputPipeline	0
MultiplierInputPipeline	0
MultiplierOutputPipeline	1
OutputPipeline	0
ReuseAccum	off
SerialPartition	-1

Figure 5. HDL block choices for the Digital Filter block.

After making these two simple pipeline changes, the clock rate of the generated HDL code exceeded 140 MHz. This is a useful lesson for engineers using code generation tools: applying a little knowledge of hardware design principles to the code generation models can have a significant impact on the results of the generated code. Further optimization of this design was possible, but deemed unnecessary, as the HDL code easily met the relatively simple timing and resource objectives for this design.

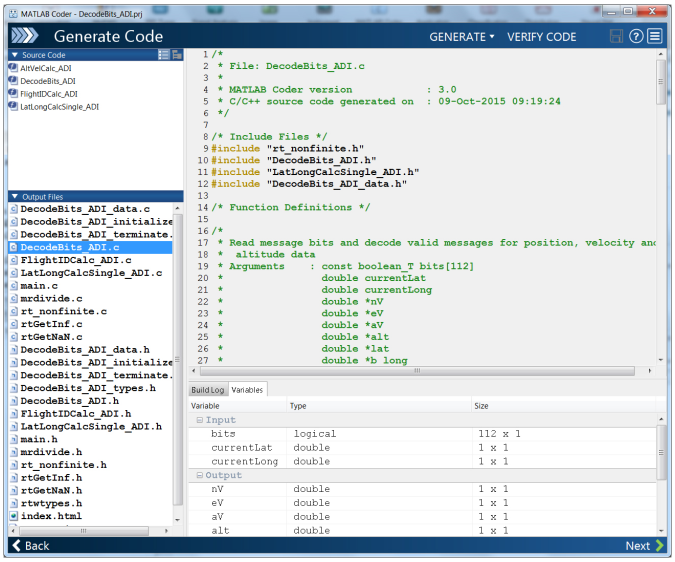
In a traditional radio design process, a large percentage of the development time is spent testing and debugging the HDL code. In the model-based design approach, used in this example, more time was spent on developing the simulation and code generation models. However, there was a significant savings in development time because the generated source code identically matched the validated behavior of the simulation; only a minimal amount of debugging had to be performed on the embedded hardware.

Generating C Code with MATLAB Coder[7]

Similar to HDL code generation, there are several conditions that must be satisfied in order to generate C code for the decoding functionality of this design. The two most important requirements are:

- Use functions supported by MATLAB Coder. MATLAB Coder supports most of the MATLAB language and a wide range of toolboxes,[8] but you may unknowingly use functions that are not supported for code generation. MATLAB Coder provides tools, such as the Code Readiness Tool,[9] to help find any unsupported functions.
- Ensure that once a MATLAB variable is declared, its size and type do not change. This is necessary to make sure that memory allocations are made correctly in the generated code.

The easiest way to generate C code from MATLAB is to open a new MATLAB Coder Project, which can be accessed from the Apps tab on the MATLAB Toolstrip. The final output of the MATLAB Coder Project can be seen in Figure 6.



```
/*
 * File: DecodeBits_ADI.c
 *
 * MATLAB Coder version            : 3.0
 * C/C++ source code generated on  : 09-Oct-2015 09:19:24
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "DecodeBits_ADI.h"
#include "LatLongCalcSingle_ADI.h"
#include "DecodeBits_ADI_data.h"

/* Function Definitions */

/*
 * Read message bits and decode valid messages for position, velocity an
 *  altitude data
 * Arguments    : const boolean_T bits[112]
 *                double currentLat
 *                double currentLong
 *                double *nV
 *                double *eV
 *                double *aV
 *                double *alt
 *                double *lat
 *                double *b long
```

Variable	Type	Size
Input		
bits	logical	112 x 1
currentLat	double	1 x 1
currentLong	double	1 x 1
Output		
nV	double	1 x 1
eV	double	1 x 1
aV	double	1 x 1
alt	double	1 x 1

Figure 6. MATLAB Coder project for DecodeBits_ADI.m.

In this project, the top level MATLAB function is DecodeBits_ADI.m. The user needs to specify the data types and sizes required by this function as input arguments. Figure 6 shows that the input arguments of this function are 112 Boolean data bits and two double precision values (to provide the user's current latitude and longitude). The output sizes and data types for DecodeBits_ADI.m (such as *nV for North Velocity, *eV for East Velocity, and *alt for altitude) are automatically determined by MATLAB Coder. MATLAB Coder finds all other functions called by the top level entry point file DecodeBits_ADI.m, including AltVelCalc_ADI.m and LatLongCalc_ADI.m, and then generates the source C code for the entire decoding algorithm.

The C code generated by MATLAB Coder is a fairly straightforward translation of the MATLAB functionality to the C language. As in the case of HDL code generation, the source code produced by MATLAB Coder is readable and traceable, so engineers can easily identify the relationship between the original MATLAB code and the generated C code. The C code from this example can be produced from the MATLAB command prompt and compiled by any ANSI C compiler.

HDL Code Platform Deployment

After partitioning the design into the functionalities that will run on the programmable logic and processing system of the Zynq, optimizing the design for HDL and C code generation, and verifying in simulation that the optimized design is functional and meets the performance criteria, it is now time to deploy the design on to the actual SDR hardware platform and verify the system's functionality under real-world conditions.

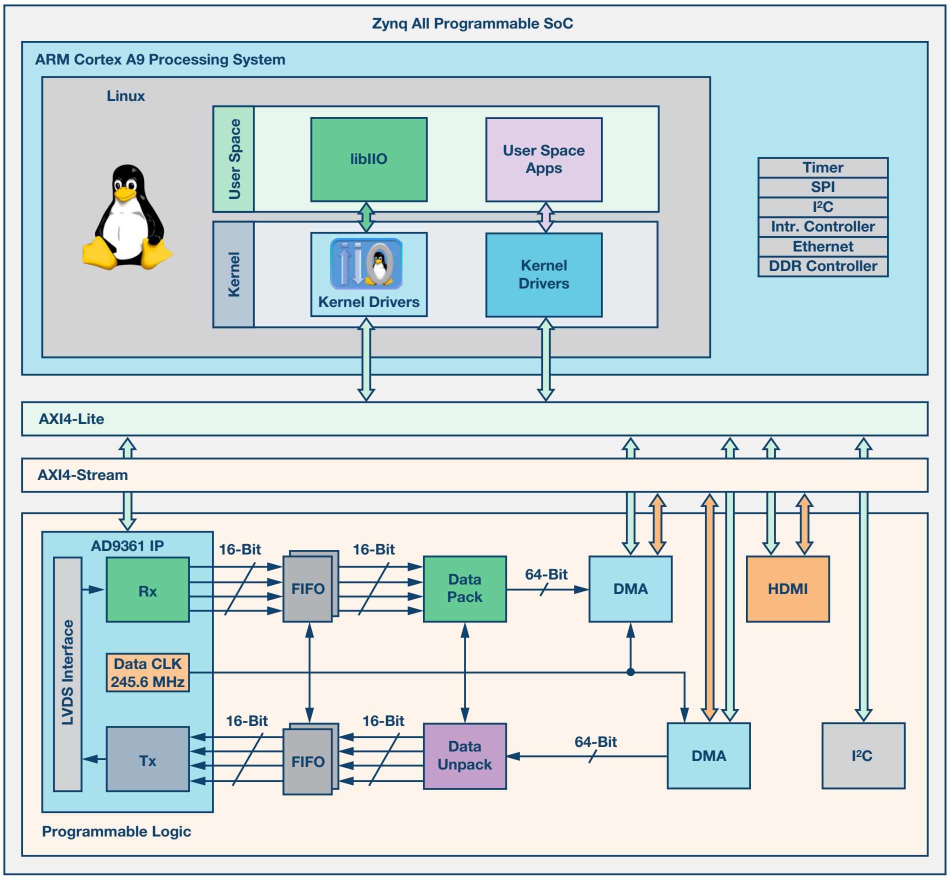


Figure 7. HDL reference design block diagram.

For this purpose, an Analog Devices AD-FMCOMMS3-EBZ SDR platform[10] connected to a Xilinx ZC706 board[11] running the Analog Devices Linux distribution is used.

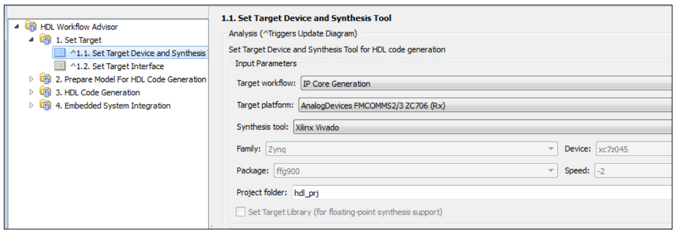
The AD-FMCOMMS3-EBZ board is accompanied by an open-source Vivado HDL reference design provided by Analog Devices.[12] This reference design contains all the IP blocks needed to configure and transfer data to and from the AD9361 transceiver on the AD-FMCOMMS3-EBZ board. Figure 7 presents a block diagram of the HDL reference design.

The AD9361 IP core implements the LVDS receive and transmit data interfaces between the AD9361 transceiver chip and the Zynq device, as well as the data interfaces to the rest of the design. DMA blocks are used for high speed data transfer between the AD9361 IP and the DDR memory. The data interface to the AD9361 IP block consists of four data lines for receive and four data lines for transmit, corresponding to the I&Q data for the two receive and two transmit channels of the AD9361. Each data line is 16 bits wide. To make the data transfers inside the system more efficient, the receive and transmit data is packed into 64-bit wide buses that are managed by the DMA blocks. Pack and unpack blocks are used to connect the 16-bit parallel data lines of the AD9361 IP to the DMAs.

Deploying the HDL code of the ADS-B model into the existing HDL infrastructure of the SDR platform requires creating an IP core that can be inserted into the data path; this is done to process the received data in real time and pass the processed data to the software layer. The deployment process can prove to be a difficult and time consuming task because it requires deep understanding of the HDL design's functionality and also adequate HDL programming skills. To simplify these steps, MathWorks includes a utility in HDL Coder called HDL Workflow Advisor, and Analog Devices provides a board support package (BSP) for the AD-FMCOMMS2-EBZ/AD-FMCOMMS3-EBZ SDR platform and Xilinx ZC706 board.[13]

The HDL Workflow Advisor guides the user through the steps needed to generate HDL code from a Simulink model. The user can choose from a selection of several different Target Workflows, including "ASIC/FPGA," "FPGA-in-the-Loop," and "IP Core Generation." Target Platform selections include Xilinx Evaluation Boards, Altera Evaluation Boards, or the FMCOMMS2/3 ZC706 SDR Platform. The rest of the code generation and target integration process can then be automated by the HDL Workflow Advisor.

The BSP provided by Analog Devices is a collection of board definitions and reference designs[14] that provide the HDL Workflow Advisor the required information and tools to generate an IP block compatible with the existing HDL reference design, and also insert the generated IP into the HDL reference design. Figure 8 shows how to configure the Workflow Advisor to generate the IP core for the ADS-B model. Please note that the IP Core Generation workflow must be selected, targeting the Analog Devices AD-FMCOMMS3-EBZ SDR platform and the Xilinx ZC706 board.


HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis
^1.2. Set Target Interface
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. Embedded System Integration

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: IP Core Generation

Target platform: AnalogDevices FMCOMMS2/3 ZC706 (Rx)

Synthesis tool: Xilinx Vivado

Family: Zynq   Device: xc7z045

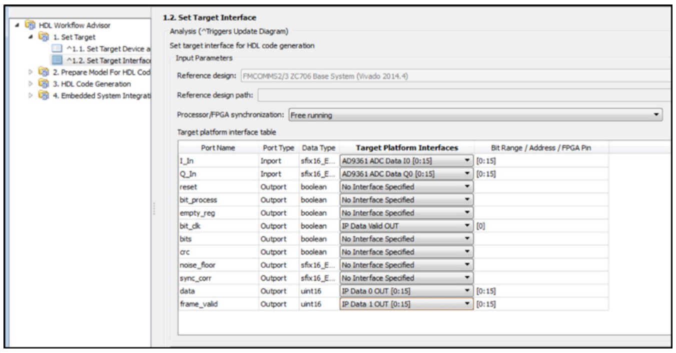
Package: ffg900   Speed: -2

Project folder: hdl_prj

☐ Set Target Library (for floating-point synthesis support)

Figure 8. Workflow Advisor configuration.

The next step is to configure the interfaces between the IP and the reference design. On the input side, the model accepts raw I&Q samples; this connects the model's input ports directly to the AD9361 receiver data ports. Of all the model's output signals, the only ones of interest at this stage are the data, frame_valid, and bit_clk signals. The data and frame_valid are 16 bits wide and are clocked by the bit_clk signal. These signals can be connected to the "DUT Data x Out" interfaces of the BSP, which means they will receive direct access to the DMA blocks; data can then be transferred into the DDR, which is accessible by the software layer. The bit_clk signal is connected to the "DUT Data Valid Out" BSP interface and controls the DMA sampling rate. Figure 9 shows how the HDL interface must be configured.



HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device a
^1.2. Set Target Interfac
2. Prepare Model For HDL Cod
3. HDL Code Generation
4. Embedded System Integrati

1.2. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Reference design: FMCOMMS2/3 ZC706 Base System (Vivado 2014.4)

Reference design path:

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
I_In	Inport	sfix16_E...	AD9361 ADC Data I0 [0:15]	[0:15]
Q_In	Inport	sfix16_E...	AD9361 ADC Data Q0 [0:15]	[0:15]
reset	Outport	boolean	No Interface Specified	
bit_process	Outport	boolean	No Interface Specified	
empty_reg	Outport	boolean	No Interface Specified	
bit_clk	Outport	boolean	IP Data Valid OUT	[0]
bits	Outport	boolean	No Interface Specified	
crc	Outport	boolean	No Interface Specified	
noise_floor	Outport	sfix16_E...	No Interface Specified	
sync_corr	Outport	sfix16_E...	No Interface Specified	
data	Outport	uint16	IP Data 0 OUT [0:15]	[0:15]
frame_valid	Outport	uint16	IP Data 1 OUT [0:15]	[0:15]

Figure 9. HDL interface configuration.

Once the target interface has been defined, Step 2 and Step 3 of the HDL Workflow Advisor can be left in their default state and the project generation process can be started by running Step 4.1 (Create Project). The result of this step is a Vivado project that has the ADS-B IP core integrated into the Analog Devices HDL reference design. Figure 10 depicts the connections between the ADS-B IP core and the rest of the blocks in the design.

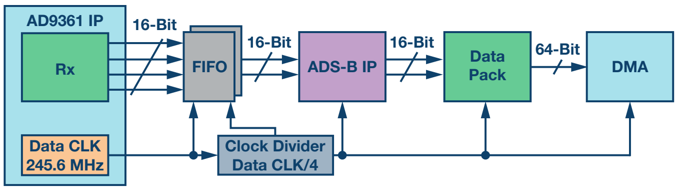


Figure 10. ADS-B IP connections in the HDL reference design.

Generating the bitstream from the Vivado project concludes the HDL integration process, but the final goal is to have Linux running on the system. For this purpose, after generating the bitstream, a Linux boot file can be created by following the standard Xilinx SDK first stage boot loader (fsbl) and Linux boot file creation process. The Linux device tree and image files corresponding to the newly created HDL design are distributed with the AD-FMCOMMS3-EBZ BSP. All files must be copied together with the Linux boot file on the boot partition of the SD card; this is used to store all files needed to run the Analog Devices Linux Distribution on the Xilinx ZC706 board.

C Code Platform Deployment

Now that the ADS-B HDL IP has been integrated into the SDR platform's HDL design, and the Linux SD card is created, it is time to implement the software application that decodes the ADS-B data. This application is based on the C code generated in Section 5 and performs the following tasks:

- Configure the AD9361 for ADS-B signals reception.
- Read the data from the ADS-B IP core.
- Detect the valid ADS-B frames in the read data.
- Decode and display the ADS-B information.

The easiest way to implement Task 1 and Task 2 is to use the functionality provided by the libiio library.[15] This library provides interface functions that enable users to easily configure the AD9361 as well as receive and transmit data. The configuration sequence sets the following system parameters:

- LO frequency—1.09 GHz
- Sampling rate—12.5 MHz
- Analog bandwidth—4.0 MHz
- AGC—fast attack mode

Besides the parameters mentioned above, a digital FIR filter with data rate of 12.5 MSPS, a pass band frequency of 3.25 MHz, and a stop band frequency of 4 MHz is loaded into the AD9361 to ensure that the received data contains only the band of interest. The system parameters and the design methodology of this FIR filter are described in Part 3 of this article series.[3]

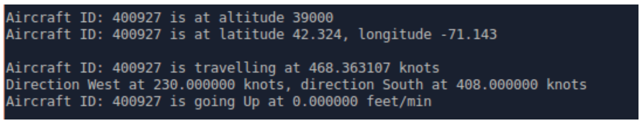
The output data of the ADS-B IP is transferred into the system's DDR memory by the DMA block. The libiio library provides the following functions: position the data acquired from the ADS-B IP into a memory buffer with a specified size; wait for the buffer to be filled; gain access the buffer through pointers. Once the buffer is filled, the ADS-B decoding algorithm can process the data. The ADS-B IP core has two output channels: one channel corresponding to the ADS-B bitstream, and the other channel indicating where a valid data frame ends in the bitstream. Both channels contain the same data rate and are synchronized with each other. A sample equal to "1" in the valid channel denotes the last bit of a valid frame in the data channel. By parsing both channels, the software can extract the valid ADS-B data frames from the bitstream and pass the data to the decoding function generated by MATLAB Coder. The decoding function uses the ADS-B data frame and the latitude and longitude of the current location as input when computing the aircraft's coordinates. The current latitude and longitude are specified as parameters of the application. The decoded ADS-B data is displayed similarly to the Simulink model.

The ADS-B data decoding application is built under Linux using a makefile. The source code of the application and the makefile can be found on the Analog Devices github repository.[16]

This completes the platform deployment steps for both the HDL and C code generated from the ADS-B model using HDL Coder and MATLAB Coder from MathWorks. The next step is to verify the system's functionality and evaluate the results.

System Validation

To validate the system's functionality, begin by creating a loopback connection between one receive and one transmit port of the AD-FMCOMMS3-EBZ board and transmit the same ADS-B signal that was used during simulation. By receiving and decoding this data, it can be verified that the output of the algorithm running on the SDR platform matches the simulation results. Figure 11 displays the output of the ADS-B data decoding application; the results are identical to those shown in Part 3 of the article series for HIL simulation using precaptured data. This provides confidence that the system is running as expected and is ready to be used with real-world data.

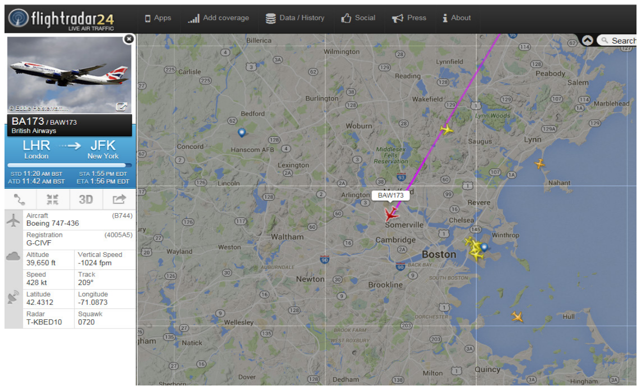


```
Aircraft ID: 400927 is at altitude 39000
Aircraft ID: 400927 is at latitude 42.324, longitude -71.143

Aircraft ID: 400927 is travelling at 468.363107 knots
Direction West at 230.000000 knots, direction South at 408.000000 knots
Aircraft ID: 400927 is going Up at 0.000000 feet/min
```

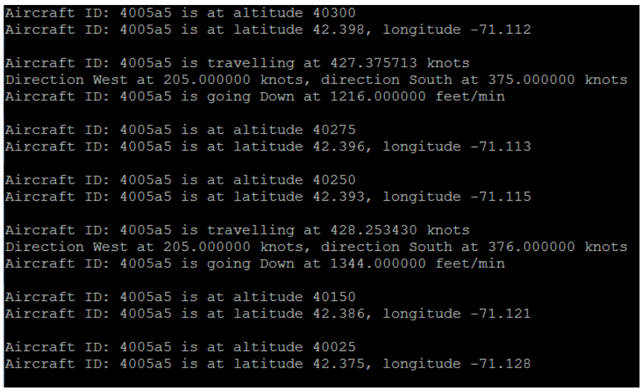
Figure 11. Loopback results.

For the actual field test, the SDR receiver was placed outside the MathWorks headquarters in Natick, MA and compared against ADS-B information decoded by the system with the data provided by airplane live tracking websites (such as flightradar24.com). It was observed that the system was able to decode data received from the airplanes within the antenna's line of sight. Figure 12 shows a comparison between the aircraft information detected by the system and the online airplane tracking data; the decoding algorithm displays the correct aircraft ID, altitude, speed, and latitude/longitude coordinates.



BA173 / BAW173
British Airways
LHR London → JFK New York
STD 11:20 AM BST STA 1:55 PM EDT
ATD 11:42 AM BST ETA 1:56 PM EDT

Aircraft (B744) Boeing 747-436	
Registration (4005A5) G-CIVF	
Altitude 39,650 ft	Vertical Speed -1024 fpm
Speed 428 kt	Track 209°
Latitude 42.4312	Longitude -71.0873
Radar T-KBED10	Squawk 0720



```
Aircraft ID: 4005a5 is at altitude 40300
Aircraft ID: 4005a5 is at latitude 42.398, longitude -71.112

Aircraft ID: 4005a5 is travelling at 427.375713 knots
Direction West at 205.000000 knots, direction South at 375.000000 knots
Aircraft ID: 4005a5 is going Down at 1216.000000 feet/min

Aircraft ID: 4005a5 is at altitude 40275
Aircraft ID: 4005a5 is at latitude 42.396, longitude -71.113

Aircraft ID: 4005a5 is at altitude 40250
Aircraft ID: 4005a5 is at latitude 42.393, longitude -71.115

Aircraft ID: 4005a5 is travelling at 428.253430 knots
Direction West at 205.000000 knots, direction South at 376.000000 knots
Aircraft ID: 4005a5 is going Down at 1344.000000 feet/min

Aircraft ID: 4005a5 is at altitude 40150
Aircraft ID: 4005a5 is at latitude 42.386, longitude -71.121

Aircraft ID: 4005a5 is at altitude 40025
Aircraft ID: 4005a5 is at latitude 42.375, longitude -71.128
```

Figure. 12 Live data results.

Conclusion

This article concludes the four part article series demonstrating how model-based design can be used to take an SDR system all the way from simulation to production. The series addressed all the stages of developing a "hardware ready" ADS-B Simulink model. We designed a simulation model to prove we could decode recorded ADS-B messages, and then validated the model with live data acquired from the SDR hardware platform. This validated not only the model but also the SDR platform's settings for the analog front end and digital receiver chain; it also gave us confidence that the platform was properly tuned for receiving ADS-B signals. Afterward, we partitioned the model into the functionalities that run on the Zynq processing system and programmable logic, and optimized the model for automatic C and HDL code generation. Finally, we integrated the C and HDL code into the SDR design and validated the system's functionality with live commercial air traffic. The end result is a design process that uses modelling and code generation tools from MathWorks, together with the Zynq SDR platform, to create a fully functional SDR system.

This example system shows that the model-based design workflow in combination with the Analog Devices AD9361/AD9364 integrated RF Agile Transceiver™ programmable radio hardware can help design teams develop working radio prototypes more quickly and less expensively than using traditional design methodologies. This prototype was built by the authors in a relatively short time with minimal obstacles, drawing on the following resources:

- The ability to build a model of an ADS-B receiver in MATLAB and Simulink that can generate usable C and HDL source code.
- Functions within HDL Workflow Advisor to automate many of the hardware/software integration steps.
- Libraries (such as libiio) that assist in the remaining integration steps to deploy the SDR prototype.
- Product help and technical support that are available from MathWorks and Analog Devices.

ADS-B is a relatively simple standard and provides a good test case to demonstrate this approach to building an SDR prototype. Engineers who adopt model-based design and the Zynq SDR platform should be able to follow the workflow presented in this series of articles to develop much more complex and powerful QPSK-, QAM-, and LTE-based SDR systems.

References

[1] Di Pu, Andrei Cozma, and Tom Hill. "Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio. Part 1—the Analog Devices/Xilinx SDR Rapid Prototyping Platform: Its Capabilities, Benefits, and Tools." *Analog Dialogue*, Volume 49, Number 3.

[2] Mike Donovan, Andrei Cozma, and Di Pu. "Four Quick Steps to Production Using Model-Based Design for Software-Defined Radio. Part 2—Mode S Detection and Decoding Using MATLAB and Simulink." *Analog Dialogue*, Volume 49, Number 4.

[3] Di Pu, Andrei Cozma. "Four Quick Steps to Production Using Model-Based Design for Software-Defined Radio. Part 3—Mode S Signals Decoding Algorithm Validation Using Hardware in the Loop." *Analog Dialogue*, Volume 49, Number 4.

[4] Analog Devices GitHub repository.

[5] HDL Coder.

[6] HDL Coder Block Support.

[7] MATLAB Coder.

[8] MATLAB Toolboxes.

[9] MATLAB Code Generation Readiness Tool.

[10] AD-FMCOMMS3-EBZ User Guide.

[11] Xilinx Zynq-7000 All Programmable SoC ZC706 Evaluation Kit.

[12] AD-FMCOMMS2-EBZ/AD-FMCOMMS3-EBZ/AD-FMCOMMS4-EBZ HDL/AD-FMCOMMS5-EBZ HDL Reference Design.

[13] Analog Devices BSP for MathWorks HDL Workflow Advisor.

[14] Board and Reference Design Registration System.

[15] What Is Libiio?

[16] MathWorks Targeting Models—ADSB.

Mike Donovan [mike.donovan@mathworks.com] is a manager in the Application Engineering Group at MathWorks. He has a B.S.E.E. from Bucknell University and an M.S.E.E. from the University of Connecticut. Prior to joining MathWorks, Mike worked on radar and satellite communications systems and in the broadband telecommunications industry.

Mike Donovan

Andrei Cozma [andrei.cozma@analog.com] is an engineering manager for ADI, supporting the design and development of system level reference designs. He holds a B.S. degree in industrial automation and informatics and a Ph.D. in electronics and telecommunications. He has been involved in the design and development of projects from different industry fields such as motor control, industrial automation, software-defined radio, and telecommunications.

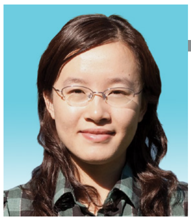
Andrei Cozma

Also by this Author:

FPGA-Based Systems Increase Motor-Control Performance

Volume 49, Number 1

Di Pu [di.pu@analog.com] is a system modeling applications engineer for ADI, supporting the design and development of software-defined radio platforms and systems. She has been working closely with MathWorks to solve mutual end customer challenges. Prior to joining ADI, she received her B.S. degree from Najing University of Science and Technology (NJUST), Nanjing, China, in 2007 and her M.S. and Ph.D. degrees from Worcester Polytechnic Institute (WPI), Worcester, MA, U.S.A., in 2009 and 2013—all in electrical engineering. She is a winner of the 2013 Sigma Xi Research Award for Doctoral Dissertation at WPI.



Di Pu