Firmware Development Methodologies for Synthetic Test Instrumentation

Michael Fluet and Joseph Manzi
Defense and Aerospace, Systems Test Group
Teradyne
North Reading, MA, USA
michael.fluet@teradyne.com
joseph.manzi@teradyne.com

Abstract—Traditional test equipment typically offers a fixed set of pre-determined functionality that is exposed to the test engineer by one or more fixed application programming interfaces. When building a general purpose test system using traditional test equipment, it is common to include a separate piece of test equipment for each supported protocol or bus type. As a result, these purpose-built test systems tend to have a large footprint, are difficult to maintain, and have a high obsolesce risk. To combat these issues, many system integrators are moving towards synthetic test instrumentation that can provide the functionality of several traditional test instruments.

With the increasing popularity of synthetic test instrumentation, many instrument vendors allow test engineers to develop their own FPGA firmware to create custom runtime-defined test instruments better suited to the specific needs of each application. Custom firmware allows a single synthetic instrument to support multiple standard and proprietary protocols, add non-standard functionality to standard protocols, and insert application specific test capability. Custom FPGA firmware development has not traditionally been the responsibility of test engineers. As a result, developing FPGA firmware for synthetic test instrumentation often presents a new development challenge that must be met.

Many synthetic test instrument manufacturers have acknowledged this challenge and provide various methods to assist the test engineer in creating custom firmware. Methods include development tools and applications, graphical programming environments, integration with third party development tools, firmware template applications, and example source code. Each firmware development methodology has its own benefits and limitations, and provides varying levels of ease of use, flexibility, portability, and functionality. When choosing synthetic test instrumentation, it is important that a test engineer take into account the available firmware development methodologies and ensure that they are a good match for their current and future development requirements. This paper compares and contrasts some of the various development methodologies currently available.

Keywords—synthetic; instrument; FPGA; firmware; development

I. Introduction

As electronic equipment increases in complexity, the automated test systems used for production testing and performance verification of electronic equipment must also increase in complexity. It is becoming progressively more common that generic test equipment is not capable of testing all interfaces on a unit under test (UUT) and that more specialized or application specific test equipment is required. While application specific test equipment can often be found, it is sometimes space or cost prohibitive to add new test instrumentation to an automated test system for each new interface, and each custom test instrument increases the maintenance effort and obsolescence risk associated with the test system.

Test engineers are challenged to determine how to provide application specific test coverage using a test system that is flexible enough to be used with several test program sets, compact in size, cost-effective, and reliable. Modern test instrumentation provides a solution in the form of synthetic instruments. With synthetic instrumentation, a single test instrument can be reconfigured as needed to support multiple buses or protocols, meeting the application specific needs of each test program set. Early synthetic instruments allowed users to vary the functionality of the instrument across several fixed modes of operation. Modern and next generation synthetic instruments often use programmable FPGAs to vary the functionality of the test instrument to meet the UUT demands, and several instrument vendors expose the FPGA programming to the end user. With the flexibility of these new open FPGA synthetic instruments comes myriad new development challenges and several development approaches to overcome these challenges.

II. TRADITIONAL TEST SYSTEM CHALLENGES

In a traditional automated test system, each bus or protocol is tested with a test instrument suited to that specific bus or protocol. If a single test program set hosted on an automated test system requires a protocol, then a test asset is added to the system to support that protocol. As multiple test program sets are hosted on an automated test system of this type, several application specific test instruments may be required. With each new instrument added to the test system, the direct and maintenance costs of the test system increase, the footprint of the test system increases, the reliability of the test system decreases, and the obsolescence risk associated with the test system increases. As UUTs become higher speed and more complex, the need for application specific testing will continue to increase.

In order to avoid using application specific instrumentation in a general purpose test system or to support buses and protocols for which dedicated test equipment is not available, many test engineers add application specific hardware and mission specific FPGA firmware to test fixtures. This practice can accelerate short-term development and provide a functional test solution, but application specific hardware and firmware can present short-term development and long-term support challenges.

III. SYNTHETIC TEST INSTRUMENTS

Many of the problems with application specific test instrumentation in a general purpose test system or application specific hardware and firmware in each test fixture are mitigated by use of synthetic test instrumentation. Synthetic test instruments allow a single test instrument to be leveraged across multiple test program sets with varied bus and protocol requirements. Consolidating multiple application specific test instruments into a single synthetic test instrument has the added benefits of reducing cost of ownership, reducing footprint, increasing reliability, and reducing obsolescence risk.

Synthetic test instruments can be binned into two groups – fixed logic synthetic test instruments and open FPGA synthetic test instruments. Fixed logic synthetic test instruments do not expose any programmable logic to the test engineer. Open FPGA synthetic test instruments allow the test engineer to develop their own application specific FPGA firmware.

A. Fixed Logic Synthetic Test Instruments

Fixed logic synthetic test instruments provide flexible operation and the ability to create custom buses and protocols without allowing or requiring the test engineer to generate any programmable logic firmware. Available functionality is typically exposed via an application programming interface similar to those used with fixed function test instruments. While this limits the flexibility of the synthetic instrument to only exposed functionality, it provides a familiar programming environment to the test engineer and there is typically little apprehension about using a fixed logic synthetic test instrument. Fixed logic synthetic test instruments can address the needs of multiple buses and protocols, but are not as well suited for semi-custom or custom bus implementations as instruments with an open FPGA architecture.

B. Open FPGA Synthetic Test Instruments

Synthetic test instruments that provide an open FPGA architecture can support standard buses and protocols as well, but they also offer additional flexibility by allowing the test engineer to write their own custom FPGA firmware. By writing application specific FPGA firmware for a synthetic test instrument, a test engineer can support standard buses and protocols, non-standard variations on standard protocols, and fully custom interfaces that may have previously required custom fixture hardware or test assets. Open FPGA synthetic test instruments are also very well suited for testing proprietary or classified protocols that are not available to test instrument vendors as the protocols can be loaded at runtime and the test instrument can be sanitized afterwards.

IV. FPGA FIRMWARE DEVELOPMENT METHODOLOGIES

Since the majority of test instruments are fixed function and expose functionality via an application programming interface or graphical user interface, many test engineers have little or no experience programming synthetic instruments and FPGA firmware. Developing FPGA firmware for synthetic test instruments can at first seem like an overwhelming task. Synthetic instrument vendors that provide open FPGA architectures understand the challenges associated with programming these instruments and offer guidance and software tools to assist the test engineer in creating custom FPGA firmware.

A. Graphical Programming Tools

One increasingly popular method of FPGA programming amongst test engineers is using graphical programming tools provided by instrument vendors. The graphical programming tools used for FPGA development are often very similar to the graphical programming tools used for test program development and integration between the two domains is almost seamless. A test engineer could possibly write their entire test program, software and firmware, in a single environment.

One major advantage to graphical programming tools is that these tools present an easy to use, intuitive interface to the test engineer. Graphical programming tools provide IP libraries for basic functionality and provisions for creating custom IP libraries from hardware description language (HDL) such as VHDL or Verilog. Simulation is done completely within the tool. Graphical programming environments often wrap the entire FPGA development process and tool chain, allowing a test engineer with little or no FPGA development experience to quickly begin to develop simple FPGA designs.

The simplicity of this programming method also presents limitations to the test engineer. Since most graphical programming tools make use of FPGA vendor specific tool chains, they must create interim VHDL or Verilog source files. However, graphical programming tools typically do not allow the user access to directly view or edit the VHDL or Verilog source code generated by the graphical programming tool. More often than not, the programmatically generated source code is encrypted or not human readable, and is deleted after synthesis. Graphical programming tools also may not expose all options in the underlying synthesis tools. In cases where the graphical programming tools do not make the most efficient use of FPGA resources, it is impossible to re-factor the design without access to the source code and the ability to change build options.

Simulation for graphical programming is done completely within the graphical programming environment. While this provides convenience and prevents the test engineer from having to learn another software tool for simulation, the embedded simulation tools only provide a small subset of those features provided with dedicated simulation software. Graphical programming simulation is often single-cycle and cannot simulate more complex FPGA firmware abstractions such as registers, DMA, FIFOs, or multiple cycle paths. Since the programmatically generated VHDL or Verilog source code is not available to the end user, using supplementary simulation software is not an option.

Another drawback to using graphical programming tools is that the tools are closed source and target only the tool vendor's test instrumentation and specific versions of a FPGA vendor's tool chain. Any test programs generated using graphical programming tools are not portable to other programming environments or other vendors' hardware. Since simulations and constraint management are done within the same graphical programming environment, they suffer the same lack of portability.

B. *Model-Based Tool Integration*

Model-based design tools are available from some third-party software vendors. These tools have traditionally been used for mathematical modeling and simulation, however extensions now exist to support graphical editing, automatic code generation, simulation, and verification, making them suitable for FPGA firmware development for open FPGA synthetic instruments. Synthetic instrument vendors can create libraries that define the firmware interfaces to their instrument and distribute them with their products. A test engineer can then combine generic blocks provided by the tool vendor, instrument specific blocks provided by the instrument vendor, and their own custom blocks into a design ready for synthesis.

Graphical editing with model-based design tools provides a simple method for editing and interconnecting complex designs by arranging firmware blocks. The same tool provides the capability to do dynamic simulation with complex simulation models and proven analysis software. Model-based design tools are well suited for applications where digital filtering and signal processing will be performed in a synthetic instrument, as the design can be mathematically simulated and adjusted until it is correct and then committed to a FPGA firmware design.

Code generated from model-based design is non-encrypted synthesizable VHDL or Verilog and adheres to industry standard coding guidelines and principles. This VHDL or Verilog source can be used with any FPGA synthesis tool, allowing resource optimization, modification outside of the model-based design tools, and portability to any capable hardware. While the generated code is portable, the models, graphical interconnect diagrams, and simulations created within the model-based design tool are not portable to other vendors' software. Since VHDL or Verilog source code is user accessible, additional simulation software outside of the model-based design tools can be leveraged.

One major drawback to third-party model based design tools is cost; since the software vendors creating model based design tools are not targeting their own hardware, this software is often expensive and requires a regular maintenance fee.

C. *Standard Firmware Development Tools*

Test engineers experienced in FPGA firmware development often prefer to write their own VHDL or Verilog source rather than generate it with graphical programming environments or model-based design tools. Writing the VHDL or Verilog source allows a test engineer to better understand what is being synthesized in their firmware, to make efficient use of the underlying FPGA resources, and to tailor the firmware development process to their specific needs. These test engineers would use standard firmware development tools for simulation and synthesis.

A test engineer using standard firmware development tools is not necessarily on his own when programming synthetic test instruments. Instrument vendors can provide resources to assist with development and integration, including:

1) Firmware templates in VHDL or Verilog, including key interfaces to instrument hardware and areas for application specific VHDL or Verilog development.
2) Ready-to-synthesize example applications, including VHDL or Verilog source code, simulations, project files, and build scripts, that allow a test engineer to successfully build firmware and confidence.
3) In-depth documentation describing how to build firmware targeting their synthetic test instrument.
4) Tools or scripts that assist with easy integration of user-defined firmware with vendor provided firmware and hardware.

There are many reasons why a test engineer may want to use standard firmware development tools. The benefits include:

1) User readable and editable VHDL or Verilog source code that allows a test engineer to see exactly what is being implemented
2) Ability to select a familiar tool chain for ease of use or integration with existing FPGA firmware development environments and processes.
3) Ability to adjust the tool chain to make the most efficient use of FPGA resources.
4) Portable code that can be used with any development tools and can target any hardware.
5) Full chip simulation capability using best-in-class simulation software
6) Low cost of ownership, assuming a FPGA development tool suite is already owned.

The drawback to using standard firmware development tools only is that it requires an understanding of a hardware description language, simulation and synthesis concepts and tools, and hardware integration that is not required when using graphical programming or model-based programming. The learning curve for someone unfamiliar with FPGA firmware development is steeper using this method than it would be with development tools that provide a graphical interface and pre-built libraries. For this reason, it is imperative that vendors offer firmware templates and example code that ranges from simple to complex and includes detailed documentation. Inexperienced FPGA firmware designers require straightforward templates and examples to ease development with the unfamiliar development process. Experienced FPGA firmware designers can use the templates as a starting point and refer to the example applications as necessary.

V. Case Study

A case study was performed to compare and contrast development using three commonly used FPGA firmware development methodologies. For this case study, we chose to assess National Instruments LabVIEW FPGA, Mathworks Matlab/Simulink and HDL Coder, and the Xilinx Vivado Design Suite. The application developed during the evaluation uses a RS-485 serial interface to transmit and receive data between various test instruments and a LinkSprite LS-Y201-RS485 camera.

A. Labview FPGA

LabVIEW FPGA, an add-on to National Instruments' LabVIEW development environment, allows users to graphically program their RIO product line of synthetic test instruments. Hardware and software used for this portion of case study included LabVIEW 2013 with the FPGA add-on, and a National Instruments Flex-RIO 7966R with a 6584 RS-485 adapter.

LabVIEW FPGA proved very easy to use and set up. If one is already familiar with LabVIEW, use of the LabVIEW FPGA add-on is intuitive. There is no need to know any HDL coding, and a very large repository of code samples are available online at ni.com that can be used to get started quickly. For this project, an RS-232 Interface Reference example created for an older RIO instrument was used [1]. Importing an existing VI and modifying it to work with the FlexRIO saved significant development time, as did the simplicity with which the UART could be embedded into a sub-VI that could be replicated across multiple channels (Figure 1). One drawback to this approach is that it is very difficult for multiple test engineers to work on multiple parts of the same FPGA design at the same time.

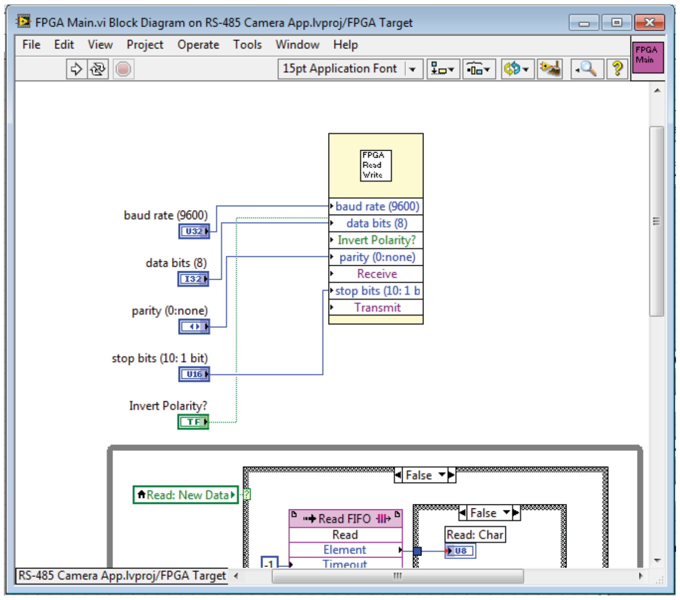


Fig. 1. Main Block Diagram VI with Embedded SubVI

The learning curve is fairly low, especially if one is already proficient with LabVIEW. The FPGA add-on introduces additional FPGA-Specific options to the LabVIEW environment like the Single-Cycle timed loop, targeted instrument-specific functions, and Xilinx Coregen IP integration. These options were useful to make sure the UART was running at the proper bit-rate in reference to the base clock of the FPGA. LabVIEW FPGA also allows the import of VHDL-coded "modules" that can be integrated into the FPGA's block diagram. This feature was not needed for the simple case study design, but it would be useful when implementing more complex designs.

When creating the Front-end application using PC-level VIs, additional tools are available via the "FPGA Interface" function that allow the VI to interact with the FPGA on the RIO instrument. It was convenient to be able to write the front-end for the application and design the FPGA in the same environment. At compile time, the graphical design "code" was converted to VHDL and passed into the Xilinx tool chain. As this process is time consuming, the FPGA design was also simulated in LabVIEW FPGA to verify that everything was wired correctly.

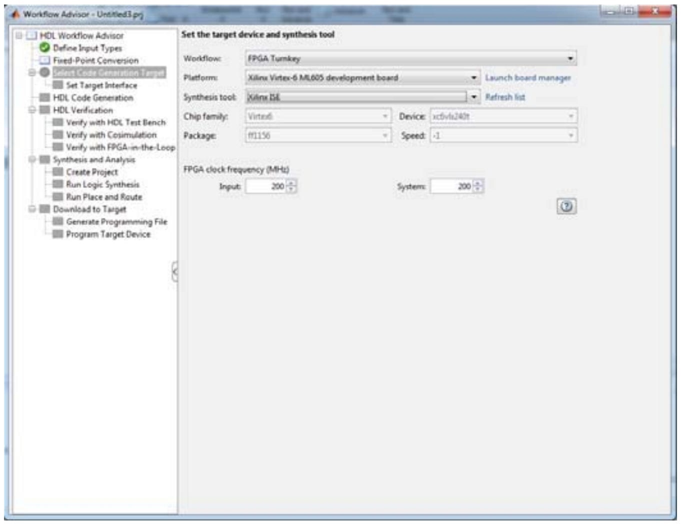
National Instruments Flex-RIO hardware had to be used for this part of the case study. LabVIEW FPGA cannot generate code or binary files for any hardware from other vendors. Application specific Flex-RIO adapter modules were used as well. Had there not been a RS-485 adapter module available, a separate integration kit is available for purchase to develop and integrate with the Flex-RIO hardware.

B. Matlab/Simulink and HDL Coder

Mathworks makes an add-on for their Matlab and Simulink suite called HDL Coder. It converts models created in Simulink or code written in Matlab to VHDL or Verilog. Matlab and Simulink with HDL Coder were used to generate VHDL, which was then synthesized using Xilinx ISE and Matlab Workflow Advisor. The resulting firmware was executed on a Xilinx ML-605 Evaluation board. For this example, Matlab Workflow Advisor was used, so a full installation of Xilinx ISE was also required.

Coding in Matlab is reminiscent of programming in Python, Windows PowerShell, or another similar scripting language. Programs can be interactively written and executed line by line in the Command Window, or they can be programmed into scripts with functions for batch execution. Many code samples were available for advanced signal processing and Memory access, however none were available for a pre-written UART. Using notes found online [2], a simple UART was implemented in Matlab code. Matlab excelled with their simulation environment, taking a list of pre-defined values and calculating the results to validate the design.

The Workflow Advisor tool was used to generate the actual HDL code (Figure 2). The tool produced the best results when the Workflow was set to “Generic ASIC/FPGA”. The output was VHDL code that could be manually imported into the Xilinx ISE tools. Getting the “FPGA Turnkey” and the “IP Core Generation” workflows to run proved rather difficult as it would never recognize the ISE installation. Every time Matlab was opened, the ISE installation needed to be manually registered. The “Set Target Interface” portion of the workflow proved to be very finicky. One time, it wouldn’t recognize ports on a function and required a restart to allow remapping of the ports. Another time, the wrong “Target Platform Interfaces” was selected for a port and the incorrect setting persisted even after being fixed, requiring a restart of Matlab to clear the setting. Using the “Specify FPGA Pin” option required the user to figure out what FPGA IO Pins to map to and how to specify them. There is no way to instantiate a list or import a UCF file. The remainder of the process flow went as expected, creating a bit file that could be downloaded to the target FPGA.



Set the target device and synthesis tool

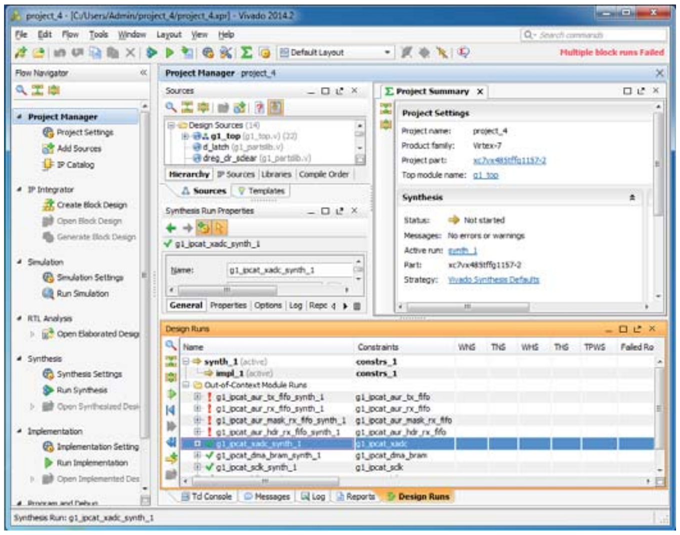
Workflow: FPGA Turnkey
Platform: Xilinx Virtex-6 ML605 development board
Synthesis tool: Xilinx ISE
Chip family: Virtex6
Device: xc6vlx240t
Package: ff1156
Speed: -1
FPGA clock frequency (MHz)
Input: 200
System: 200

Fig. 2. Mathworks Workflow Advisor Tool

Overall, creating a bitstream proved to be somewhat difficult for a beginner. Additional models are needed from synthetic instrument vendors, as the HDL Coder “Workflow Advisor” is by default limited to a small list of target hardware. There is no obvious way to import a user constraints file; everything must be typed in manually when creating a new board. Nonetheless, model-based design appears to be a good option for using graphical programming that is portable between target hardware from multiple vendors or for cases where Matlab or Simulink modeling already exists for integral parts of a design.

C. Vivado Design Suite

Xilinx makes an ADE for their FPGA devices called Vivado Design Suite (Figure 3). This suite provides standard design tools for Xilinx FPGAs and allows users to program Xilinx’s Series 7 FPGAs using VHDL or Verilog. They also include some templates to get started with basic programming. The hardware used for the evaluation was the Teradyne High Speed Subsystem Flexible IO Expansion Instrument with a RS-485 Physical Interface Module. Teradyne High Speed Subsystem software drivers and firmware templates were also used.



Project Settings
Project name: project_4
Product family: Virtex-7
Project part: xc7vx485tffg1157-2
Top module name: g1_top

Synthesis
Status: Not started
Messages: No errors or warnings
Active run: synth_1
Part: xc7vx485tffg1157-2
Strategy: Vivado Synthesis Defaults

Synthesis Run Properties: g1_ipcat_xadc_synth_1
Name: g1_ipcat_xadc_synth_1

Design Runs

Name	Constraints	WNS	TNS	WHS	THS	TPWS	Failed Ro
synth_1 (active)	constrs_1						
impl_1 (active)	constrs_1						
Out-of-Context Module Runs							
g1_ipcat_aur_tx_fifo_synth_1	g1_ipcat_aur_tx_fifo						
g1_ipcat_aur_rx_fifo_synth_1	g1_ipcat_aur_rx_fifo						
g1_ipcat_aur_mask_rx_fifo_synth_1	g1_ipcat_aur_mask_rx_fifo						
g1_ipcat_aur_hdr_rx_fifo_synth_1	g1_ipcat_aur_hdr_rx_fifo						
g1_ipcat_xadc_synth_1	g1_ipcat_xadc						
g1_ipcat_dma_bram_synth_1	g1_ipcat_dma_bram						
g1_ipcat_sdk_synth_1	g1_ipcat_sdk						

Synthesis Run: g1_ipcat_xadc_synth_1

Fig. 3. Xilinx Vivado Design Suite

The Xilinx Vivado Design Suite is easy to install. Everything needed to run the design suite exists standalone, so a test engineer can conveniently install multiple versions side-by-side. Once started, it was easy to set up a new project targeting the FPGA to be used. The Vivado Design Suite provides an “IP Integrator” feature that allows one to add multiple IP integration “Blocks”, graphically wire them up, and then expose the completed blocks to the main VHDL or Verilog code. This feature was not evaluated in this case study.

The Vivado Design Suite supports either VHDL or Verilog code, as well as mixing them in a single project. Basic programing in Verilog or VHDL is fairly easy to pick up. VHDL is a strongly-typed language, requiring everything to be clearly defined and wired. Verilog is a weakly-typed language, much more forgiving and “C-like”. The HDL used is often defined by the project requirements, or may be simply the preference of the test engineer.

Using the Verilog application template provided with the High Speed Subsystem Test Development Kit, all IP blocks required to communicate with the instrument were imported. Finding a UART in the Xilinx IP catalog and adding it to the project was very simple. The FIFO in the UART was then "wired" to the appropriate FPGA IO pins and registers. Simulation files were manually created to verify that the new additions to the FPGA application template were functioning as expected. After upgrading some modules, synthesis and implementation were able to run successfully and a bitstream was created. Instrument specific APIs were then used to load the FPGA firmware into the High Speed Subsystem Flexible IO Expansion Instrument.

Starting with no Vivado or VHDL experience, it took a long time to get comfortable with the Vivado Design Suite and learn VHDL. Using this approach, the test engineer is responsible for everything from creating the proper algorithms to exposing the right endpoints so that the instrument APIs can load the generated bitstream. There are a lot of subtleties and programming choices that can make a difference between whether or not your HDL code successfully compiles. With all the flexibility provided by directly using the Vivado Design Suite or other standard firmware development tools to program a synthetic test instrument, it can be difficult to get started. Having lots of example code, documentation, and environment templates available from the test instrument manufacturer is a huge help in getting over this initial hurdle.

This case study involved relatively simple firmware development that could be done on any hardware. Had this been a larger, more complex application, the full chip simulation available with standard FPGA firmware development tools would have offered much more value. This methodology is also suitable for designs targeting multiple hardware targets or requiring portability.

VI. Conclusion

Synthetic test instrument vendors provide various methods for programming FPGA firmware in their instruments. All methods of FPGA firmware development discussed in this paper have their benefits and drawbacks. The case study in this paper shows that various FPGA firmware development methodologies may be better or worse suited for a particular firmware development task. For simple designs, a graphical programming environment like LabVIEW provides ease of use without a requirement for HDL coding. Model-based design tools like Matlab/Simulink with HDL Coder are available for firmware integration with powerful modeling software. And for complex designs requiring powerful simulation and portability, standard FPGA firmware development tools, such as the Vivado Design Suite, excel. A test engineer using open architecture synthetic test instrumentation needs to consider what is most important to their applications – ease of use, flexibility, portability, or functionality – and choose a development methodology that aligns with their current and future development requirements.

References

[1] "RS-232 Interface Reference Example for LabVIEW FPGA."- *National Instruments.* March 8, 2010. http://www.ni.com/example/27164/en/

[2] "FPGA Implementation of Universal Asynchronous Receiver and Transmitter (UART)." – *Haibo Wang, ECE Department, Southern Illinois University.* March 29, 2012. http://www.engr.siu.edu/~haibo/ece428/notes/ece428_uart.pdf