Reconfigurable Virtual Instrumentation Design for Radar using Object-Oriented Techniques and Open-Source Tools

Ryan Seal and Julio Urbina
The Pennsylvania State University
USA

1. Introduction

Over the years, instrumentation design has benefited from technological advancements, and, simultaneously suffered adverse effects from increasing design complexity. Among these advancements has been the shift from traditional instrumentation, defined by a physical front-panel interface controlled and monitored by the user; to modern instrumentation, which utilize specialized hardware and provide interactive, software-based interfaces, from which the user interacts using a general-purpose computer (GPC). The term virtual instrumentation (VI) (Baican & Necsulescu, 2000) is used to describe such systems, which generally use a modular hardware-based stage to process or generate signals, and a GPC to provide the instrument with a layer of reprogrammable software capabilities. A variety of instruments can be emulated in software, limited only by capabilities of the VI's output stage hardware. In more recent years, the popularity of reconfigurable hardware (Hsiung et al., 2009), more specifically field programmable gate arrays (FPGAs) (Maxfield, 2004), have revolutionized capabilities of electronic instrumentation. FPGAs are complex integrated circuits (ICs) that provide an interface of digital inputs and outputs that can be assembled, using specialized development tools, to create customized hardware configurations. By incorporating FPGA technology into traditional VI, the instrument's capabilities are expanded to include an additional layer of hardware-based functionality.

In this chapter we explore the use of such enhanced instruments, which are further augmented by the use of object-oriented programming techniques and the use of open-source development tools. Using low-cost FPGAs and open-source software we describe a design methodology enabling one to develop state-of-the-art instrumentation via a generalized instrumentation core that can be customized using specialized output stage hardware. Additionally, using object-oriented programming (OOP) techniques and open-source tools, we illustrate a technique to provide a cost-effective, generalized software framework to uniquely define an instrument's functionality through a customizable interface, implemented by the designer. The intention of the proposed design methodology is to permit less restrictive collaboration among researchers through elimination of proprietary licensing; making hardware and software development freely available while building upon an existing open-source structure. Beginning with section 2, the concept of a virtual instrument, its inherent properties, and improvements through reconfigurable hardware will be discussed. Section 3 will cover object-oriented software design principles and its application to instrumentation. Finally, section 4 will provide an application of these techniques applied to radar instrumentation, where we present the design of a radar pulse generator used to synchronize transmit and receive stages for the 430 MHz radar at the Arecibo Observatory (AO).

2. Virtual instrumentation

In this section, an overview of VI and related concepts will be provided in conjunction with an introduction to reconfigurable hardware and tools necessary for design.

2.1 Virtual instruments

VI, like traditional instrumentation, is used for systematic measurement and control of physical phenomena; but, unlike its counterpart, VI contains additional software layer which defines the VI's functionality. This additional functionality is used to emulate traditional instrumentation and is bounded solely by physical limitations imposed by the instrument's output stage hardware. A block diagram illustrating the virtual instrument is shown in Figure 1. Popularity of VI has flourished in scientific and research-oriented environments, largely driven by the necessity and ubiquitousness of general-purpose computing. Many commercially available (Baican & Necsulescu, 2000) systems exist today and all provide some form of modular, specialized hardware configurations to augment the number of emulable instruments available to the user. These specialized hardware units commonly utilize some form of analog-to-digital (A/D) or digital-to-analog (D/A) circuitry combined with a standard interface to communicate information to and from the GPC.

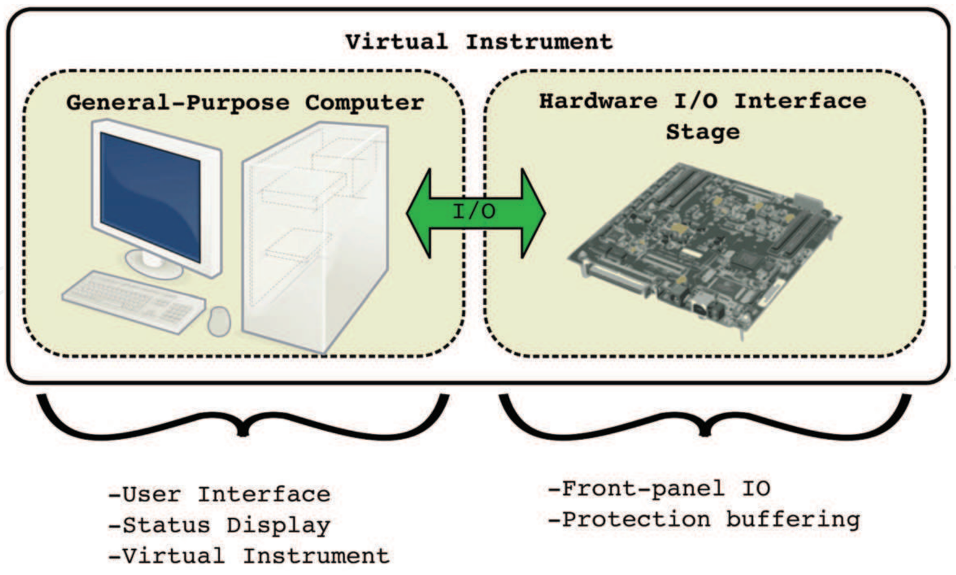


Fig. 1. A block diagram illustrating the concept of the virtual instrument

2.2 Reconfigurable logic

Reconfigurable, or programmable, logic devices come in many forms including: simple programmable logic devices (SPLDs), complex programmable logic devices (CPLDs), field

programmable logic devices (FPGAs), application-specific integrated circuits (ASICs), and application-specific standard parts (ASSPs). The general term, programmable logic device (PLD), is used to describe both SPLDs and CPLDs, which contain coarsely defined groups of logic, determined by the manufacturer, that can be customized by the end-user. These devices are more beneficial for special-purpose hardware applications, and are typically chosen when the design engineer has a well-defined set of specifications to work from, but the potential for variability in specifications or the instrument's functionality exists. At the opposite end of the programmable logic spectrum lies both ASICs and ASSPs. These devices are also used in special-purpose hardware applications, but these applications require stringent design specifications, allowing the design engineer to finalize and send the design to a special-purpose ASIC/ASSP manufacturer. Any engineering changes made to an ASIC/ASSP require retooling by the manufacturer, thus making this technology more costly in design environments with variability. FPGAs fall somewhere between the more complex CPLD design and the rigidness of the ASIC design, providing smaller groups of internal logic that can be combined to create CPLD-like designs and even larger, complex designs like that of the typical ASIC application. This amount of flexibility, combined with costs much lower than ASIC design, and comparable to CPLDs, make the FPGA a strong candidate for reconfigurable VI design.
In addition to these advantages, many FPGA vendors provide free software design tools that can be used with many of the vendor's low-cost devices, which are suitable for most VI applications. FPGA designs follow a well-defined design structure, known as the *design flow* (Navabi, 2006), which consists of several stages including:

1. Design Entry

In this stage the design engineer translates the design requirements into a language recognized by FPGA vendor tools. There are several languages to choose from and the field of FPGA language design is growing at a rapid rate. The five most common of these are: 1) Verilog, 2) SystemVerilog, 3) VHDL, 4) SystemC, and 5) pure C/C++ (Maxfield, 2004). Future trends appear to be moving towards higher-level (i.e. object-oriented) languages to better accommodate the ever-growing size and complexity of FPGAs.

2. Design Validation

After entering the design, the design engineer must create a simulation test-bench to ensure that the design entry functions as expected. A number of tools are available for test-bench simulation, and, depending on the language used for design entry, many of these tools are available in open-source, including: iverilog and gplcver for Verilog; ghdl for VHDL; and SystemC.

3. Synthesis

The Synthesis stage compiles the given design entry into a bitstream file format that can be used to program the device. Several sub-stages in the synthesis process optimize the design to meet requirements using vendor-specific tools.

4. Post Synthesis Verification

During the synthesis stage, timing information and loading effects for each electrical path in the design are collected. This information is contained in the bitstream file generated at the final stage of synthesis and is used to perform secondary simulations to ensure that timing delays and loading effects, not apparent in the design entry phase, are understood.

2.3 Reconfigurable virtual instruments

Combining the concepts presented in sections 2.1 and 2.2, an even more powerful form of VI can be constructed, providing the designer with the ability to utilize a reconfigurable hardware module to assist in time-critical applications or application-specific algorithms (e.g. FFTs, image processing, filtering, etc...) more suitable for hardware-based platforms. An illustration of reconfigurable virtual instrumentation (RVI) is depicted in Figure 2 along with each component's responsibilities. RVIs distribute the virtual instrument component among the GPC and reconfigurable hardware module, with the partitioning ratio dependent upon the application and the intentions of the design engineer. RVIs provide opportunities for areas of instrumentation development unachievable using standard VI, due to the GPC's computational inefficiencies and scheduling delays inherent in GPC-based systems. High-performance computing is now commonplace using GPC platforms assisted by reconfigurable hardware (Bishof et al., 2008), with results often tens to hundreds of times faster than equivalent multiprocessor systems. The benefits of such systems can also be applied to radar-based problems, including computational problems and instrumentation design; an example of the latter will be detailed in section 4.

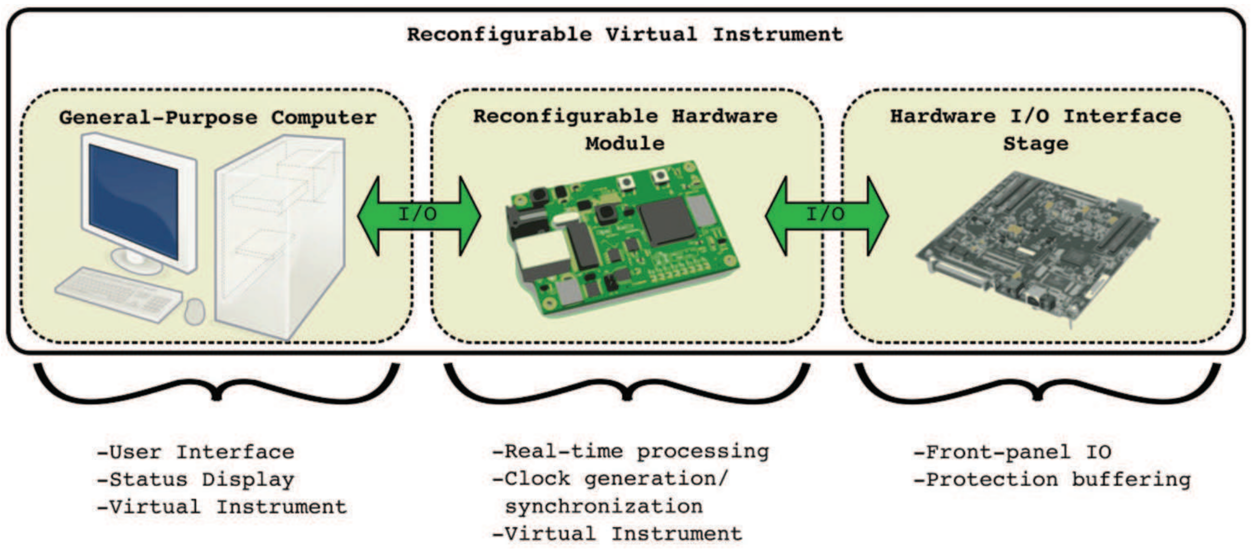


Fig. 2. A block diagram illustrating the introduction of a reconfigurable hardware module to enhance the basic VI concept.

3. RVI design methodology using a generic framework

In this section, additional specifications are given to the RVI definition provided in section 2.3 to further generalize the design and alleviate difficulties typically encountered in the implementation phase. Furthermore, relevant details concerning principles of software re-use and object-oriented principles will be discussed.

RVIs provide a basic instrumentation platform for a user to interface with the instrument through a software interface, typically supplied by a Personal Computer (PC), which also controls the reconfigurable hardware module and output stage hardware through a common communication peripheral. Although useful, in this section we present a more robust solution providing a stand-alone instrumentation core that can be customized to a variety of applications, including radar instrumentation.

3.1 Small form factor general-purpose computers

In recent years, GPC technology has become an imposing competitor to more traditional standards (e.g. PC104 and Single Board Computers) in the embedded market, primarily due to price/performance ratios, but also from the surge in small, custom hardware modules supporting standard communication interfaces (e.g. PCI, Ethernet, USB 1.0/2.0). Small form factor (SFF) GPCs provide four form factors, shown in Figure 3, suitable for compact, stand-alone instrumentation. All SFF GPCS are commercial-off-the-shelf (COTS) products and

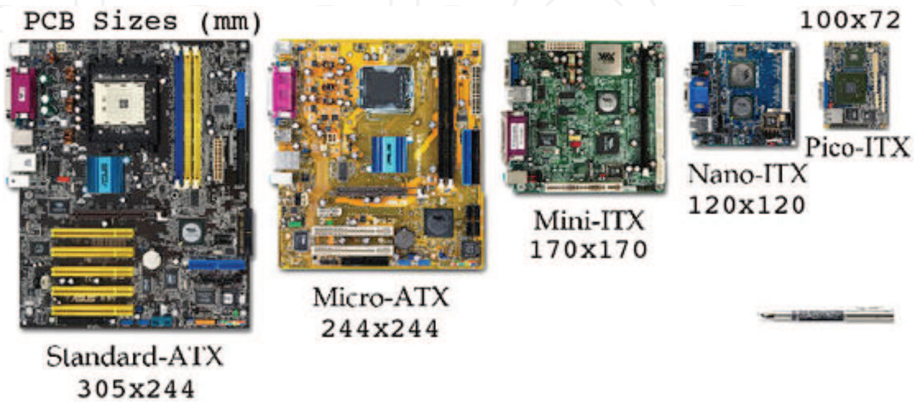


Fig. 3. Comparison of Small Form Factor Motherboards available today

incorporate standard peripherals, such as integrated video cards, keyboard/mouse connections, USB 1.0/2.0, IDE connectors, RS-232, and 100Mbps Ethernet ports. These systems are competively priced, provide impressive processing capabilities, and can be readily replaced in the event of failure. Capitalizing on these factors, we propose a generic RVI core that can be integrated into a standard rack-mount chassis; providing a packaged instrument accessible directly by keyboard, video, and mouse; or remotely accessible using an Ethernet-based communication protocol. The instrument provides a generalized framework from which customizations can be made to meet specifications determined by the designer. The overall design is intended to provide a packaged, customizable system, that can operate in remote locations accessible to Internet. This concept is illustrated in Figure 4

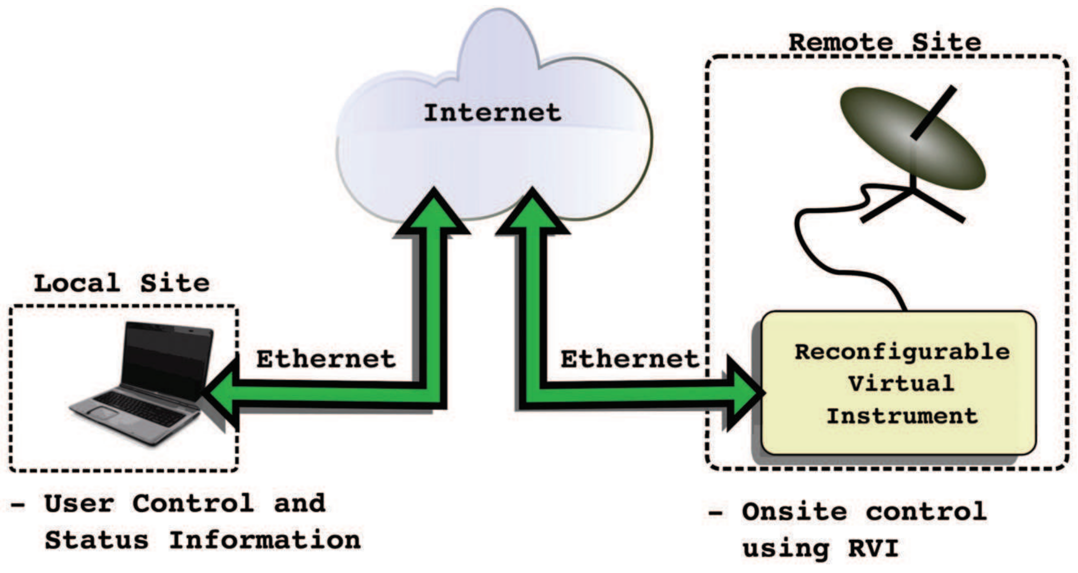

Internet
Remote Site
Local Site
Ethernet
Ethernet
Reconfigurable Virtual Instrument
- User Control and Status Information
- Onsite control using RVI

Fig. 4. An illustration of a reconfigurable virtual instrument with a remote interface.

3.2 Software description and structure overview

Designing a generalized software framework conducive to customization is a crucial component in VI designs. In this section, we will provide a framework for generic, object-oriented interfaces that can be easily extended to meet requirements of custom applications.

3.3 Object-oriented programming and open-source software

Object-oriented programming, popularized in the mid 1980s, has long been considered a method for managing ever-increasing software complexity through a structuring process known as *object-oriented decomposition* (Booch, 1994). *Objects*, with respect to software design, can be considered tangible entities providing an interface through which interactions take place. The description, or definition, of the *object* is usually referred to as a *class*. In a program, a *class* is *instantiated* to create an *object*. Details of these actions are hidden behind the interface using a technique referred to as *encapsulation*, which separates the behavior of the *object* from its interface. Concerning instrumentation development, we can use these techniques to define software utilizing well-defined interfaces (Pugh, 2007), and then subsequently define its behavior to emulate that of the instrument being designed.

An instrument is defined by its inputs, outputs, control parameters, and its function. Mathematically, an instrument can be represented by

$$\mathbf{y} = T(\mathbf{x}, \mathbf{c}), \qquad \mathbf{y} \in \mathbb{R}^l, \mathbf{x} \in \mathbb{R}^m, \mathbf{c} \in \mathbb{R}^n, \tag{1}$$

where $\mathbf{y}$ is the output vector, $\mathbf{x}$ represents the input vector, and $\mathbf{c}$ is the control vector. Design of the instrument requires definition of the instrument's function $T(\cdot, \cdot)$ and $\mathbf{c}$. Additionally, careful consideration must be given to users of the instrument by ensuring that the instrument's control parameters use familiar terminology and provide expected behavior. These criteria are typically provided in the instrument's planning stages, known as design requirements. Modern VI designs commonly use graphical user interfaces (GUIs) to simplify the instrument's input interface. Although useful, designs utilizing command-line interfaces provide important advantages over GUIs, primarily relating to system debugging, upgrades, and, in some cases, performance. In addition to these benefits, the command-line system can be easily augmented to accommodate a GUI after completing the initial design, which also removes the tendency for the designer to couple code describing system functionality with the GUI framework, leading to difficulties when making necessary revisions. For this reason, focus will be given to a command-line driven system.

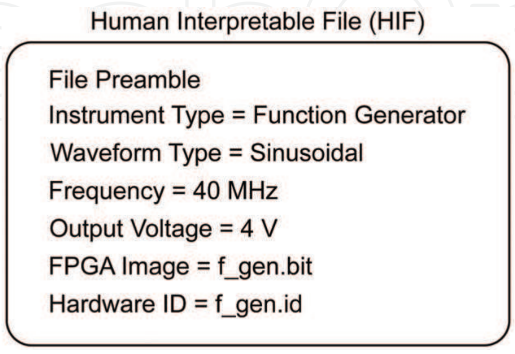
To provide a generic framework for customization, the instrument's design is divided into two categories:

- Instrument configuration
  Describes the instrument's static configuration, which includes the instrument's type and its initial control parameter values.
- Instrument run-time operation
  Defines the dynamic portion of the instrument, namely its dynamic command-set (e.g. start, stop, reset, etc...) as well as control parameter values that can be changed dynamically, while the system is in operation.

3.4 Instrument configuration design

The instrument's configuration is defined by the user through a plain-text input file, either located directly on the instrument, or, in the case of a remote-interface, the user's local machine. Reasons (Hunt & Thomas, 1999) for using plain-text input files are numerous, including:

- Universally understood by all Operating Systems (OS) and programming languages.
- Consists of printable characters interpretable by people with the use of special tools.
- Can be made more verbose, more representative of human language, thus making it self-describing (i.e. no documentation required) and obsolescent-proof.
- Expedites debugging and development.



Human Interpretable File (HIF)

```
File Preamble
Instrument Type = Function Generator
Waveform Type = Sinusoidal
Frequency = 40 MHz
Output Voltage = 4 V
FPGA Image = f_gen.bit
Hardware ID = f_gen.id
```

Fig. 5. Sample Human Interpretable File illustrating simplicity and ease of use.

This plain-text file is referred to as a Human Interpretable File (HIF) and contains a number of parameters necessary to define the instrument and its configuration. These parameters include, but are not limited to: instrument type, control parameters, FPGA bitstream file, and, if necessary, the output stage hardware. An example HIF is shown in Figure 5. The HIF structure and content are chosen at the designer's discretion, but the instrument's type must be consistent, as it is required to load the proper interface for parsing and translating the remaining portion of the file, which contain control parameters particular to the specified instrument type. This interface is referred to as the Instrument Definition Interface (IDI) and its responsibilities include:

- Contains the instrument's control parameter specifications, which are used to verify settings requested in the HIF.
- Formats the HIF into an intermediate format more suitable for use in the verification and post-verification processes.
- Creates an Instrument Interpretable File (IIF) formatted for direct use by the reconfigurable hardware.

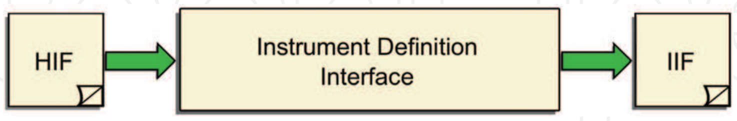


Fig. 6. Block diagram illustrating the Instrument Definition Interface

A block diagram of the IDI is shown in Figure 6. The IDI, in terms of OOP-style programming, defines an interface that accepts an input file and produces an output file. The implementation details are coded by the instrument's designer, who must provide a customized instrumentation *class* that implements the IDI. The resulting output file, or IIF, is structured to provide information concerning the required bitstream file, output stage hardware, and a custom data structure directly readable by the reconfigurable hardware. A sample IIF file is illustrated in Figure 7. As an example, a class diagram, depicted in Figure 8, is used to illustrate how a specialized instrument *class* implements the IDI. Additionally,

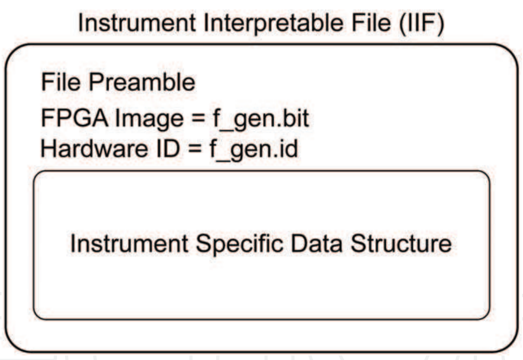

Instrument Interpretable File (IIF)
File Preamble
FPGA Image = f_gen.bit
Hardware ID = f_gen.id
Instrument Specific Data Structure

Fig. 7. Sample Instrument Interpretable File illustrating structure.

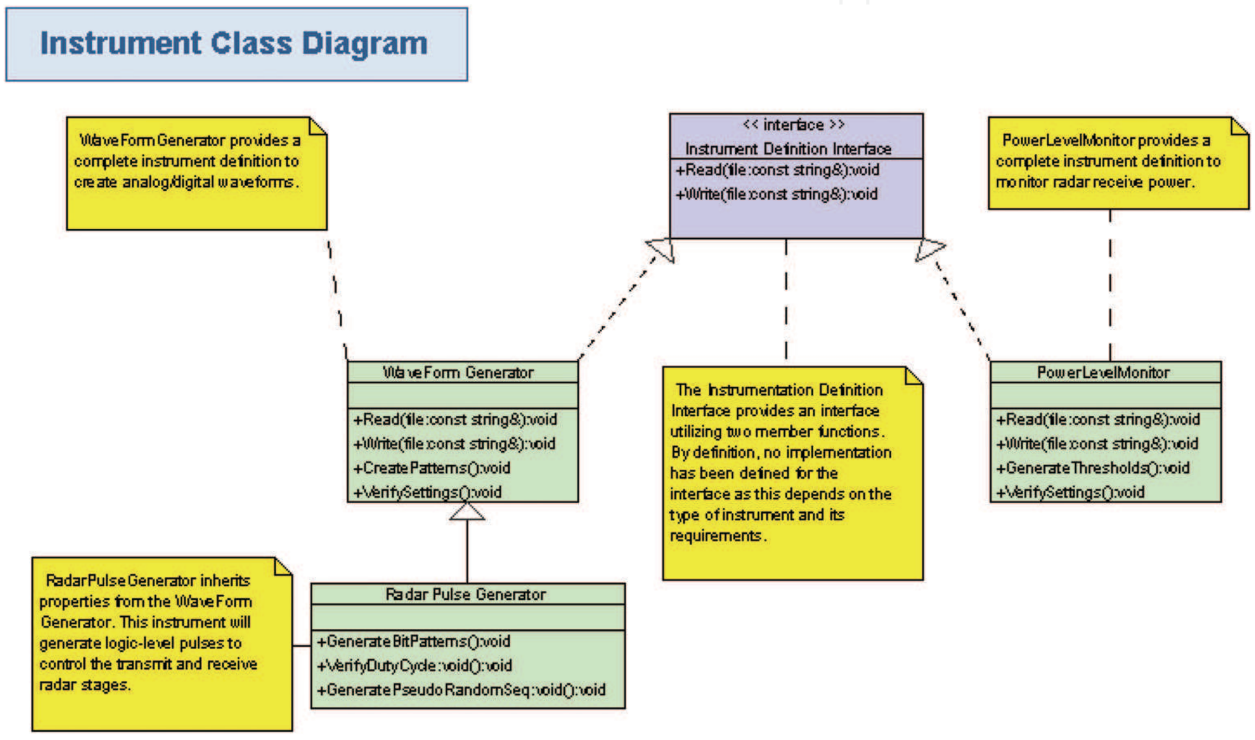

Instrument Class Diagram
WaveForm Generator provides a complete instrument definition to create analog/digital waveforms.
<< interface >>
Instrument Definition Interface
+Read(file:const string&):void
+Write(file:const string&):void
PowerLevelMonitor provides a complete instrument definition to monitor radar receive power.
WaveForm Generator
+Read(file:const string&):void
+Write(file:const string&):void
+CreatePatterns():void
+VerifySettings():void
The Instrumentation Definition Interface provides an interface utilizing two member functions. By definition, no implementation has been defined for the interface as this depends on the type of instrument and its requirements.
PowerLevelMonitor
+Read(file:const string&):void
+Write(file:const string&):void
+GenerateThresholds():void
+VerifySettings():void
RadarPulseGenerator inherits properties from the WaveForm Generator. This instrument will generate logic-level pulses to control the transmit and receive radar stages.
Radar Pulse Generator
+GenerateBitPatterns():void
+VerifyDutyCycle:void():void
+GeneratePseudoRandomSeq:void():void

Fig. 8. Block diagram illustrating the Instrument Definition Interface

the concept of inheritance can also be seen, where the *WaveFormGenerator class* is further specialized into a *RadarPulseGenerator class*, which, in turn, is a derivative of the *WaveFormGenerator class*. Several instrumentation *classes* can coexist in the RVI's storage and a particular *class* is chosen by the instrument type, which is defined by the HIF. It should be noted that all information provided by the IDI is considered *static* information, meaning this file is used to initialize the system; including the instrument's type, the instrument's control parameters, and the FPGA bitstream file. These concepts are realized through creation of a *generator* program, which is a command-line program used to read an HIF, validate its inputs, and create an IIF readable by the *run-time* program described in section 3.5.

3.5 Instrument run-time operation design

After defining the instrument's specifications and control parameters, a mechanism is needed from which the user can select and load the virtual instrument and reconfigurable hardware definitions into the system. All information necessary to accomplish this task is contained in the IIF, which was produced by the configuration program presented in section 3.4. These operational tasks are performed by the *run-time* program, which provides a command-line interface through which the user interacts with the system. The *run-time* program begins with a minimal interface, and, after the user chooses an IIF, this interface is expanded to include functionality provided by the instrument described in the file. A specialized helper *class* is used to parse the IIF, determine the instrument's type, and load the specified Instrument Operating Interface (IOI) into the system. Next, the system programs the reconfigurable hardware with a generalized bitstream file, whose purpose is to retrieve identification information (e.g. reading an EEPROM device) from the output stage hardware and verify compatibility with the user's requested instrumentation type. After verification, the system initializes the reconfigurable hardware with the bitstream file provided in the requested IIF. Finally, the system can accept commands provided by the loaded virtual instrument. A diagram illustrating this process is shown in Figure 9. Software encompassing these concepts is referred to as the *run-time* program, which operates in a shell environment incorporating a command-line user interface. As previously mentioned, these concepts can easily be expanded to include a GUI-based display system, depending on the user's requirements.

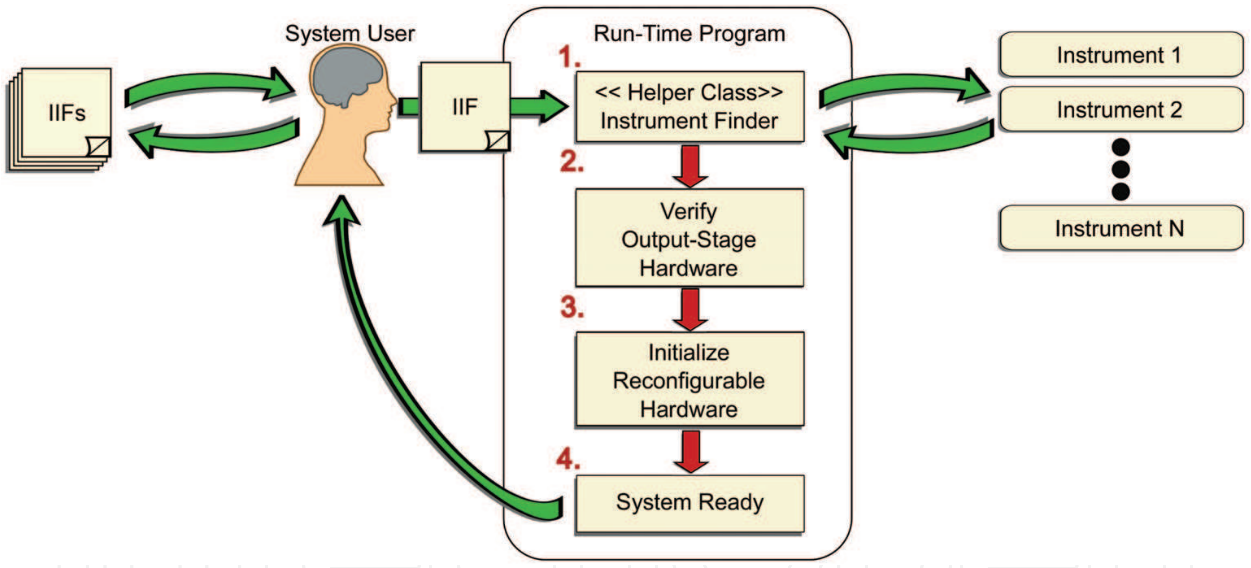

IIFs
System User
IIF
Run-Time Program
1.
<< Helper Class>> Instrument Finder
2.
Verify Output-Stage Hardware
3.
Initialize Reconfigurable Hardware
4.
System Ready
Instrument 1
Instrument 2
Instrument N

Fig. 9. Block diagram illustrating the interactive run-time interface.

4. Implementation of a radar pulse generator

In this section, discussion of a specialized radar instrument, referred to as the radar pulse generator (RPG), is developed using concepts defined in earlier sections. The sections that follow will provide details of hardware components, software, and all tools necessary for implementation. The RPG (Seal, 2008) was designed for use at AO and served as a starting point for concepts developed in sections 2 and 3.

RPGs provide logic-level signals to control and synchronize a radar's transmit and receive stages in pulsed radar applications as shown in Figure 10. These pulses ensure the

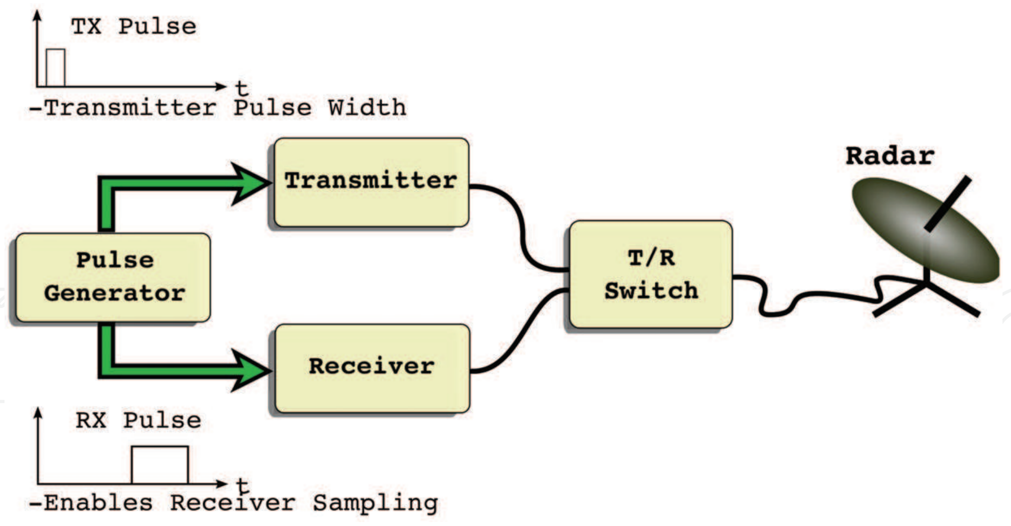


Fig. 10. Monostatic radar overview illustrating the pulse generator used to control both transmitter and receiver.

transmitter operates within a safe regime while simultaneously controlling the receiver's sampling mechanism to record user-defined ranges. The transmitted pulse is given by

$$p(t) = \Re\{m(t)e^{j\omega_0 t}\}, \tag{2}$$

where ω_0 is the transmitter's carrier frequency and $m(t)$ is a periodic waveform that modulates the carrier signal with pulse width δ_t and period T_t. For monostatic radars, a pulse is transmitted, and after a specified delay, the radar switches into receive mode. The RPG, in its most basic form, supplies the receiver's gating circuitry with a periodic pulse (gating pulse) of width δ_r and period T_t, as specified by $m(t)$. The rising edge of δ_r represents the first range sample r_0. The radar's received signal is given by

$$r(t) = \alpha p(t-\tau) + w(t), \tag{3}$$

where α is a generalized attenuation factor reflecting power loss in the returned signal, $w(t)$ represents additive noise, and τ is a specified delay determined by

$$\tau = 2r/v_p, \tag{4}$$

where r is the range in meters, and v_p is the wave's velocity in meters per second. In addition to timing control between transmit and receive modes, the RPG must also generate the transmitter's modulating waveform $m(t)$, which can potentially consist of moderately complex pulse modulation patterns. The RPG's design must accommodate a large number of these patterns, as well as an efficient, user-friendly interface to create, modify, and store them.

4.1 Hardware

Specifications for the design were taken from existing hardware in operation at AO and new capabilities were suggested from the scientific staff. Primary suggestions included: an efficient, easy-to-use interface for designing moderately complex bit patterns; the ability to store and cycle bit patterns without clocking glitches; flexibility, with minimal hardware modifications, to expand system functionality for new research as well as future needs; and the ability to monitor, configure, and control the system remotely. Given these specifications and requirements, an RVI-based system, utilizing a GPC and COTS FPGA module, was chosen, and a block diagram illustrating these components is shown in Figure 11.

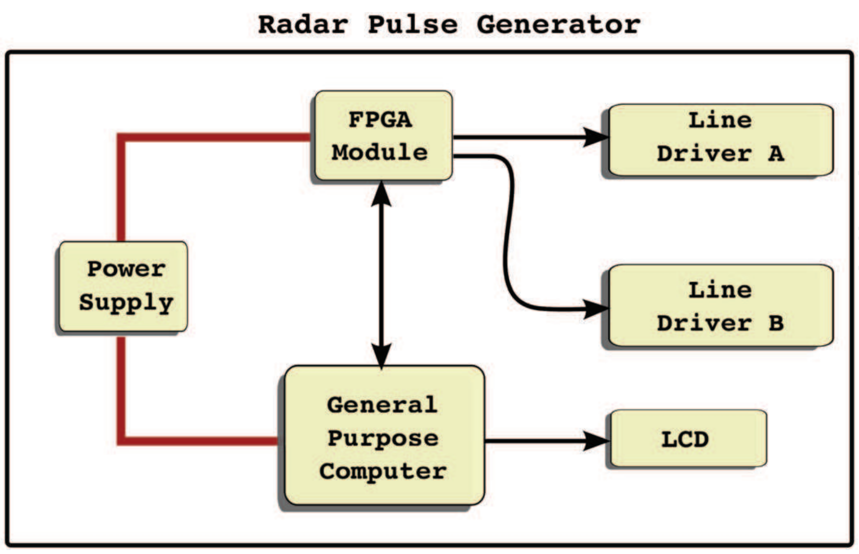


Fig. 11. Block diagram illustrating radar pulse generator components.

Small form factor general-purpose computer
Remote configuration and control were an important aspect of the system; allowing engineers to operate, diagnose, and monitor the instrument's status from an off-site location (e.g. home). To accomplish this, a small on-board computer, supplied with an Ethernet interface, was chosen. The computer, a Mini-ITX form factor mainboard, depicted in Figure 12, contains a number of standard peripherals (e.g. USB, RS-232, video, etc...) commonly found in PCs. The Linux Operating System was chosen for the design due to scalability and availability of existing open-source tools (e.g. VNC, ssh, sockets) for remote access.

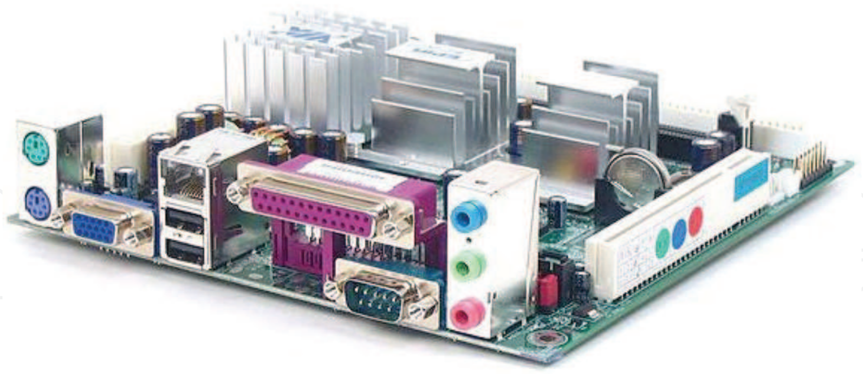


Fig. 12. VIA ML6000 Mini-ITX form factor mainboard

On-site system status
To assist on-site monitoring, debugging, and maintenance; a USB-controlled liquid crystal display (LCD) was used. Software was written to display a number of parameters to indicate the system's operational state.
System power
The instrument's mainboard and FPGA carrier are powered from a standard PC power supply. Although smaller, more-efficient solutions were available, these supplies are more

readily available and less costly to replace. LCD power is derived directly from the mainboard's USB interface, and output stage hardware is powered from the FPGA carrier. A momentary pushbutton switch mounted on the rear panel is used to power the system.

FPGA module

For experiments requiring sub-microsecond timing, reconfigurability, and the ability to produce lengthy bit patterns, a low-cost COTS FPGA module was chosen. The module, shown in Figure 13, uses a consumer-grade Xilinx Spartan-3A FPGA and provides an on-board clock oscillator and phase lock loop (PLL) circuitry. The module is controlled using a USB 2.0 interface with greater than 32 MBPS downstream (GPC to FPGA) transfer rates and nearly 18 MBPS upstream data rates. *Verilog HDL* was used for coding and verification was performed using *icarus* and *gtkwave*. FPGA synthesis was performed using the Xilinx Webpack development suite, which is freely available and supports a number of Xilinx FPGAs. Further details of the FPGA code implementation is described in section 4.3

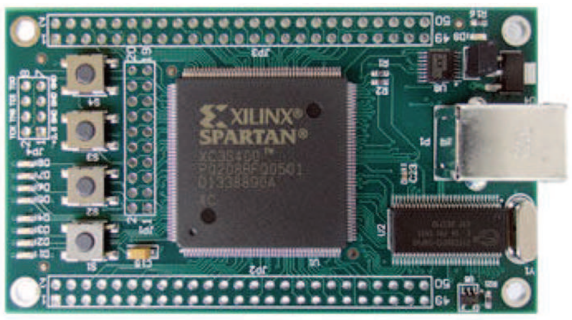


Fig. 13. XEM3001V2 Opal Kelly FPGA module

FPGA carrier

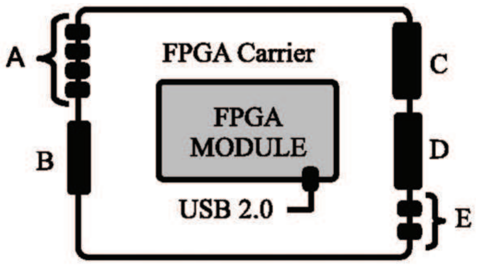
The FPGA carrier board was designed to provide both signal buffering and access using 20-pin dual-header connectors. In total, 32 signal outputs, divided into two 16-bit ports labeled *PORTA* and *PORTB*, are provided, along with external clock inputs, synchronization sources, and an external trigger. These modifications allow the FPGA module to synchronize with an external system clock, and provide precision timing through an on-site atomic clock. A 256Kx16 SRAM module is mounted on-board, allowing *PORTA* to optionally function as a dedicated SRAM controller. Additionally, *PORTB* can be dedicated to use a quadrature 10-bit AD9761 DAC. A block diagram of the board is shown in Figure 14. Schematic capture, board layout, and overall design were completed using the GPL'd Electronic and Design Automation software (gEDA), a popular open-source development suite for schematic capture and printed circuit board (PCB) design.

Output stage hardware

The output stage hardware is composed of two boards, each providing 8 signals, to drive a 50-ohm coaxial transmission line using TTL-level signals. Each board mounts to the instrument's front-panel using BNC pcb-to-panel connectors and provides socketed ICs for quick, in-field replacement. Power and signals are supplied from ribbon cables connected to the FPGA carrier board's 20-pin port connectors.

4.2 System design and operation

Software design consists of two independent programs: 1) the system's *generator* program, which is responsible for bit pattern creation and system configuration; and 2) the *run-time*



A: Ext. Clock / Sync Inputs
B: ATX Power Conn.
C: 16 Channel PORT A Digital Outputs
D: 16 Channel PORT B Digital Outputs
E: 10-bit DAC Quadrature Outputs

Fig. 14. FPGA carrier board block diagram

shell, which allows the user to control the system via command-line. These programs follow design approaches presented in sections 3.4 and 3.5 and all software was written using the C++ program language (Stroustrup, 2000).

Generator program

The RPG is capable of producing any number of arbitrary, indefinitely repeatable, bit patterns. Generation of such patterns must account for operating limits of the radar system, and an effective, efficent entry method who use the system. To lessen complexity, the approach discussed in section 3.4 was utilized by designing an HIF to store common system parameters specific to the radar's transmitter. These parameters are passed to the IDI, where they are analyzed using a language parser. The parser translates the data into a bit-vector format which passes through rule-checking and verification stages that contain custom-defined transmitter specifications. If verification succeeds, an IIF is written to the system's hard drive; otherwise the system exits and reports the error to the user. This particular IIF contains an ASCII-based structure of 1's and 0's, representing digital logic levels. The FPGA's bitstream file is configured to parse this structure and instruct the hardware of the requested bit pattern sequence, clock source, and synchronization method.

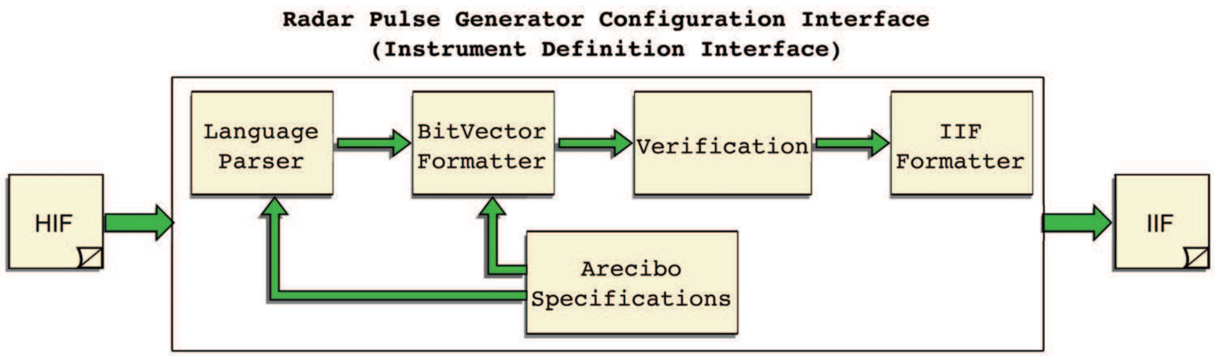


Fig. 15. The Configuration Interface implements the Instrument Definition Interface.

Run-time shell program

After bit pattern generation and verification, a *run-time shell* program is used to operate the instrument; communicating directly with the FPGA through a library-based interface provided by the manufacturer. This library is free and currently available in the following languages: C/C++, Python, Ruby, Matlab, and LabView. Although we present an implementation using C++, any of these languages can be substituted to independently develop a customized system that may be more suitable for a particular development group, depending on the developer's experience and skill set. The program resides in the on-board, Linux-based computer and provides the following control commands: 1) load and unload any number of bit patterns; 2) start and stop the system; 3) request hardware status; 4) switch bit patterns; and 5) select the desired clock and trigger source. Using a separate thread of execution (i.e. multi-threading), system status is polled from the FPGA at a user-defined rate (25 ms default) and an overall status is displayed on the instrument's LCD. Figure 16 illustrates an overview of program operation.

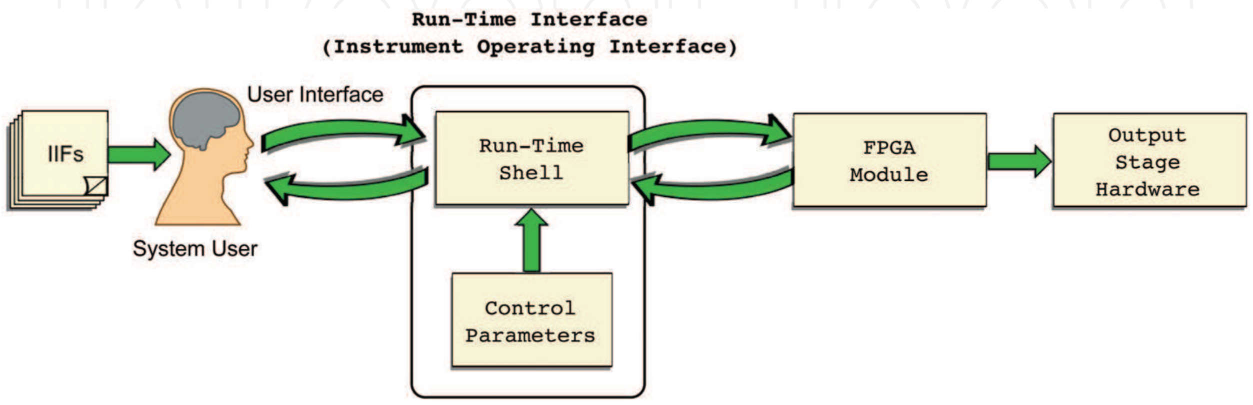


Fig. 16. Block diagram illustrating interaction between the user and the radar pulse generator's run-time shell.

4.3 HDL design and operation

Code for FPGA implementation was written in *Verilog HDL* and functionality was divided into a number of small, well-defined modules designed to improve readability and alleviate code maintenance. The Opal Kelly XEM3001v2 FPGA module, used in this design, utilizes a Xilinx xc3s400 FPGA containing 400,000 gates, 208 I/O, and 288 kbits of on-board RAM. Other features provided by the XEM3001v2 limit the potential number of I/O to 90. Considering the system's requirements, it was determined that the memory structure implemented in the design and clock routing choices would be the primary factors determining performance. Design of the FPGA module began with an analysis of communication methods provided by the Opal Kelly library's Application Programming Interface (API). The Opal Kelly API operates using firmware to establish FPGA/PC communication and a small HDL module integrates into the user's FPGA design to provide communication with the host API. For data transfers implemented in this design, two types of communication were chosen: 1) multi-byte data transfers using the Opal Kelly *PIPE* modules, and 2) simple status/control commands using the Opal Kelly *WIRE* modules. Opal Kelly *PIPE* modules are designed to efficiently transfer a known number of 16-bit wide integers between the host PC and FPGA module while the *WIRE* modules are more suitable for controlling or monitoring a single 16-bit state. An overview of the FPGA's HDL data flow design is depicted in Figure 17 and a description of relevant modules is given in the sections that follow.

InputControl module

This module was designed to act as a simple state machine and makes use of a single Opal Kelly *PIPE* input module to transfer data into the FPGA. When data is present on the *PIPE*

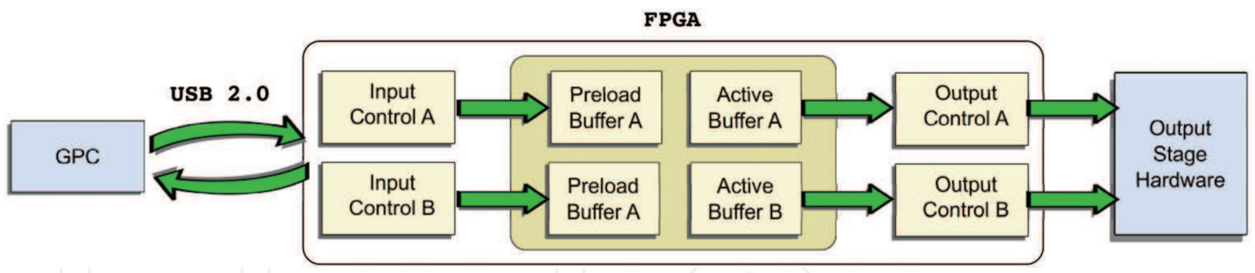


Fig. 17. Overview of FPGA HDL model and illustration of data flow from the module's input to output.

port, the *InputControl* module routes data to the proper memory buffer and maintains a count of bytes transferred. When the proper number of bytes have been transferred, the module sends a signal back to the host PC and returns to the idle state. The module also communicates with other modules in the FPGA to perform synchronization and hand-shaking internal to the system.

OutputControl module

After system initialization is complete (i.e. bit patterns are loaded), the user can send a *start* command to the FPGA. This signal is sent to the *OutputControl* module and the first state of the active buffer's bit pattern is loaded onto the output ports. An internal state counter is decremented at a rate determined by *clock divider* variable until the count reaches zero. At this point, the next state is loaded onto the output ports and the state counter is initialized with the new state count. When the process reaches the last state in the bit pattern, the system checks for a *switch* or *stop* signal. The *switch* signal tells the module to switch buffers and continue with the new bit pattern. Additionally, the host is alerted of the successful switch and prompted to load a new bit pattern into the *preload* buffer. This dual-layer buffering allows the system to cycle, glitch free, through any number of bit patterns without delay. The *stop* signal will halt the module's execution and go into an idle state. If neither signal is present, the module will reload the first state in the current buffer and repeat the bit pattern. Since this module monitors the *switch* signal, it controls the buffering system, alternating the designation of *preload* and *active* buffers. The *OutputControl* module is responsible for sending bit patterns to the output ports, incrementing through the states via the state counter, and selecting the active and preload buffers, which change when signaled from the PC using a command. Additionally, the module synchronizes the states to the user selected source and implements a clock divider module for seamless switching of the state machine's operating rate.

ControlSignals module

The *ControlSignals* module relies on a 16-bit word that is responsible for controlling the FPGA's changeable states. The word uses 14 bits, with each bit controlling states on the port's input and output modules. The 2 remaining bits provide custom functionality specific to the 430 MHz transmitter at AO. This behavior is easily customizable to match needs for any system. Among the various controls are: 1) system start, 2) system stop, 3) switch patterns, and 4) sync source selection.

StatusSignals module

Similar to the *ControlSignals* module, the *StatusSignals* module monitors the status of both *InputControl* and *OutputControl* modules. This 16-bit word is read by the calling program as often as necessary. Currently, the shell program creates a separate thread and reads the status word at 25 ms intervals. The module's primary function is to provide feedback to the

PC and synchronize data transfers; preventing any aliasing of data buffers and providing verification of the control commands sent the FPGA.

Results

System performance can be characterized in terms of resource usage, clock speed, and GPC-to-FPGA communication rates. At the start of the design, specifications were given requiring the FPGA module to accept 3 possible clock sources: an on-board oscillator and 2 external clock sources. The on-board oscillator was provided by the FPGA module and is used for isolated system debugging. The 2 external clock sources both required a maximum reference frequency of 20 MHz, which specified a minimum pulse resolution of 100 ns (i.e. one half the clock rate). The memory requirements were taken from the existing equipment and specified by 4kx16 blocks for each port. Loading new bit patterns into the machine required real-time scheduling and had to be scheduled precisely with the site's clock reference (i.e. atomic clock). After the initial design entry and functional verification, synthesis and various optimizations were performed using the provided Xilinx Webpack tools. All of the system's timing requirements were met and clock inputs up to ≈ 68 MHz were achieved; well above the specified 20 MHz clock rate. According to generated reports, approximately 25% of the FPGA's resources were utilized. All available block RAM was allocated to pattern storage and communication rates were limited by the 25 ms refresh rate imposed on the status and control lines. The FPGA employed a dual-stage buffer to provide real-time mode switching. Precision of the pattern switching was accomplished through

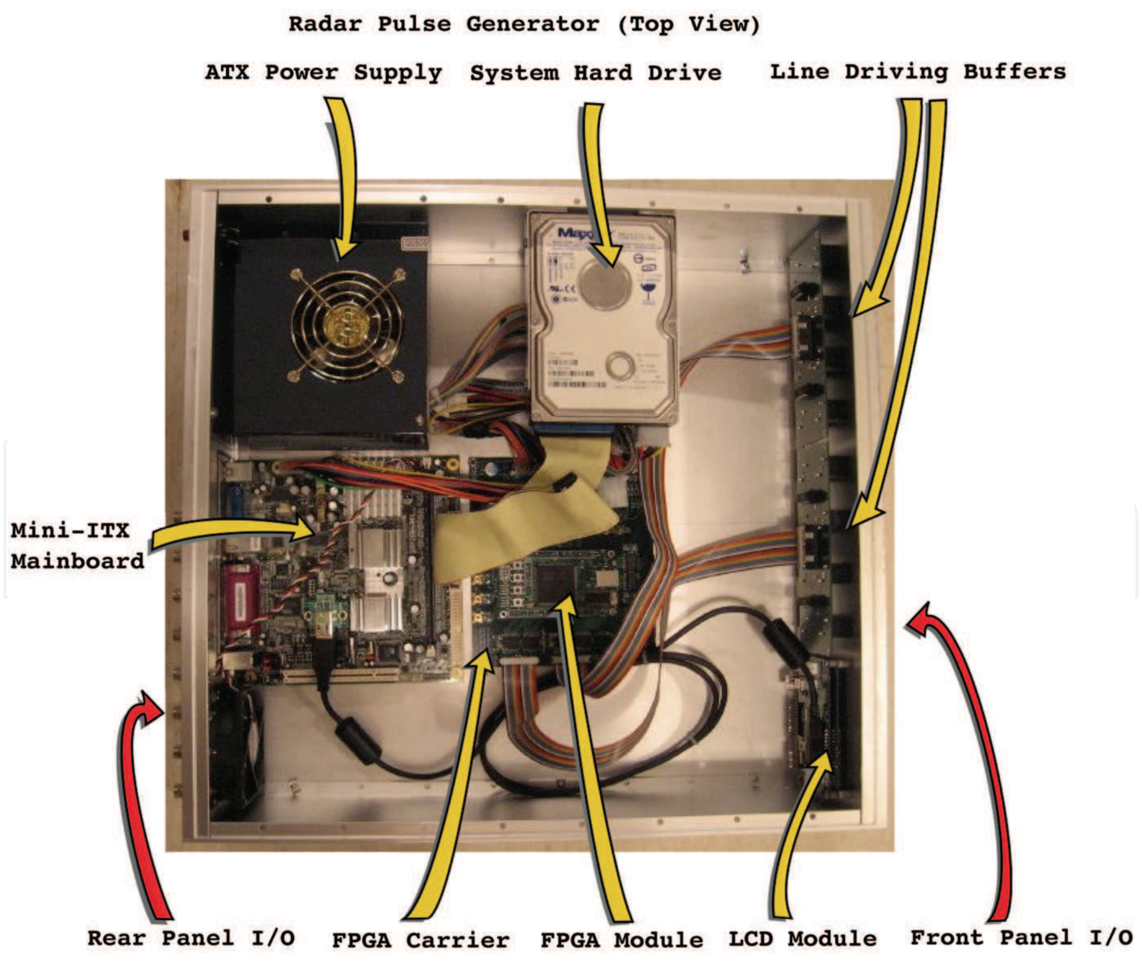


Fig. 18. Radar Pulse Generator prototype top view with component labeling

synchronization with a 1 PPS timing signal derived from the on-site atomic clock. This enabled the user to request a new pattern from the non-real-time *run-time shell* program, and still maintain real-time scheduling requirements. Machining of the prototype took place in-house at AO and the FPGA carrier and output stage hardware boards were sent to a manufacturing facility. The device is contained in a 3U size rack-mount chassis and three different views of the instrument are illustrated in Figures 18, 19 and 20.

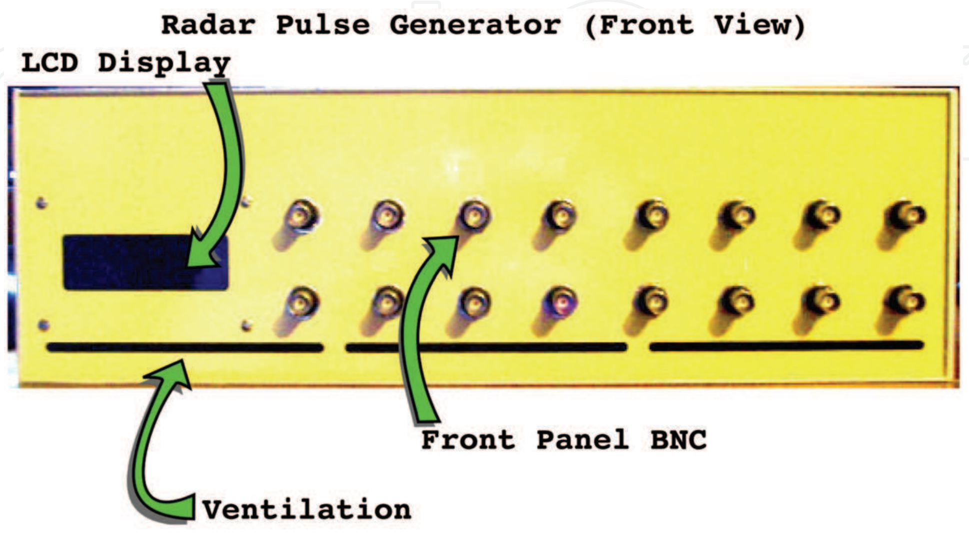


Fig. 19. Radar Pulse Generator prototype front view with component labeling

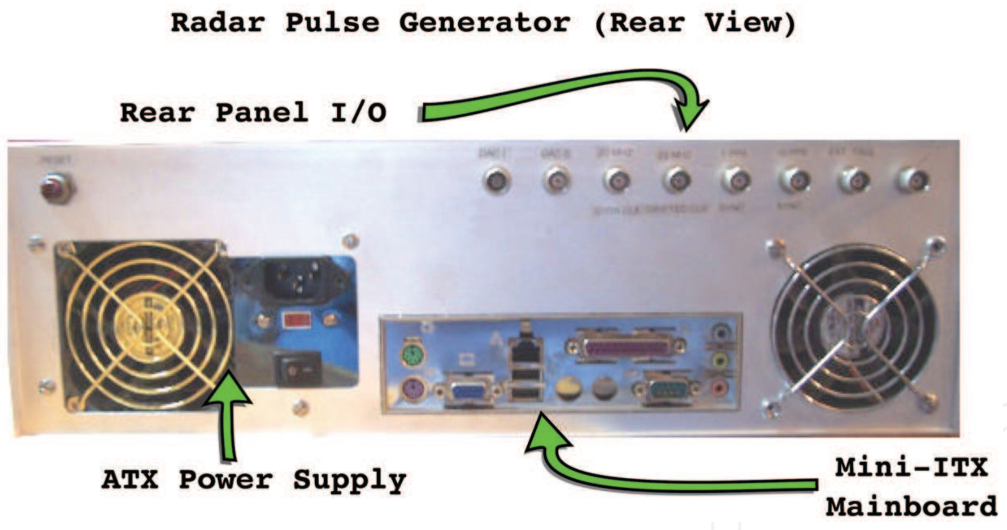


Fig. 20. Radar Pulse Generator prototype rear view with component labeling

5. Conclusion

In this chapter, we have presented principles and techniques enabling the design and implementation of high-speed, low-cost instrumentation employing COTS components and reconfigurable hardware. Several open-source tools were introduced to provide viable alternatives to expensive, proprietary software requiring maintenance fees and limiting the ability to promote code sharing due to licensing restrictions. Emphasis was also given to object-oriented programming techniques, which help reduce design complexity and promote software reuse through interface-oriented techniques. Finally, a radar-based application was presented, demonstrating a number of concepts developed throughout the chapter.

6. References

Baican, R. & Necsulescu, D. (2000). *Applied Virtual Instrumentation*,WIT Press, UK.

Bishof, C., Bucker,M., Gibbon, P., Joubert, G., Lippert, T.,Mohr, B. & Peters, F. (2008). *Advances in Parallel Computing*, IOS Press.

Booch, G. (1994). Object-Oriented Analysis and Design, The Benjamin/Cummings Publishing Company, Inc. Hsiung, P., Santambrogio, D. & Huang, C. (2009). Reconfigurable System Design and Verification, CRC Press, USA.

Hunt, A. & Thomas, D. (1999). *The Pragmatic Programmer*, Addison-Wesley Professional, USA.

Maxfield, C. (2004). *The Design Warrior's Guide to FPGAs*, Newnes, USA.

Navabi, Z. (2006). *Verilog Digital System Design*, McGraw-Hill Publishing Companies, Inc., USA. Pugh, K. (2007). *Interface-Oriented Design*, The Pragmatic Programmers LLC., USA.

Seal, R. (2008). Design and implementation of a multi-purpose radar controller using opensource tool, *Proceedings of the IEEE Radar Conference 2008*, Rome, May 2008, pp. pp. 1–4.

Stroustrup, B. (2000). *The C++ Programming Language*, Addison-Wesley, USA.

Radar Technology
Edited by Guy Kouemou

ISBN 978-953-307-029-2
Hard cover, 410 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

In this book “Radar Technology”, the chapters are divided into four main topic areas: Topic area 1: “Radar Systems” consists of chapters which treat whole radar systems, environment and target functional chain. Topic area 2: “Radar Applications” shows various applications of radar systems, including meteorological radars, ground penetrating radars and glaciology. Topic area 3: “Radar Functional Chain and Signal Processing” describes several aspects of the radar signal processing. From parameter extraction, target detection over tracking and classification technologies. Topic area 4: “Radar Subsystems and Components” consists of design technology of radar subsystem components like antenna design or waveform design.