Using FPGAs to interface with digital communication protocols

Vineet Aggarwal, National Instruments - June 06, 2007

Custom or proprietary digital protocols are commonly used in today's world for device or sub-system communication in everything from Aerospace to Consumer Electronics. Many of these applications also use popular standards like SPI, I2C, I2S and S/PDIF. FPGAs can be easily configured to interface with a wide variety of protocols, and – using high-level programming tools – engineers are able to design, prototype, and test faster than ever before.

Over time, digital protocol specifications can change and evolve, and ASIC-based interfaces face challenges with regard to maintenance and forward compatibility. FPGA chips, however, are reconfigurable to keep up with future modifications that might be necessary. This how-to article discusses the implementation a custom digital communication protocol using FPGAs.

When implementing a digital communication protocol with FPGAs, it's important to understand the bigger picture. A typical specifications document for a digital protocol will include everything from individual byte translation all the way to the type of connector to use. There are several layers of abstraction that eventually work their way down to the physical electrical characteristics associated with each I/O pin. For example, digital logic levels could be simple 5V or 3.3V TTL lines (transistor-transistor logic) with single-ended signaling that references a common ground. Ground referenced signals are used to reduce the number of required pins per data line; however, while this might reduce the cost of cabling or line drivers, they are more susceptible to noise and therefore can’t be used for long distances. Digital standards like RS-485 use differential signaling between -7 and 12 volts, for communication up to 4000 feet. Depending on the protocol being used, it might be necessary to implement additional circuitry for signal conditioning, and to amplify or attenuate electrical signals when interfacing to an FPGA chip.

Once digital logic levels adhere to the protocol specifications, an FPGA application can be written to control I/O lines and group individual data bits into bytes. The transfer of digital data from one device to another could be synchronous (ie: referencing the same clock) or asynchronous (ie: using independent clocks or handshaking). Digital protocols commonly describe a group of bytes as a frame, and each frame of a communication sequence could contain bytes for error correction, device identification, or addressing information. Not every communication protocol has these complexities, but it's important to understand the various layers of abstraction and decide how much should be implemented on the FPGA and how much should be handled by some other system component like a microprocessor (of course the microprocessor could also be implemented on the FPGA). Since FPGAs operate at the physical hardware level, controlling individual digital I/O lines, this article will primarily focus on clocking data lines and interpreting byte commands.

Custom protocol definition
Now that we've had a brief introduction to the different types of digital protocols, let's delve into an implementation example. This tutorial will reference a simple example that uses five digital I/O lines
to serially communicate information from a slave device to a master device. To make this example a little more interesting, let's imagine that this is a video game system, in which the slave device is a gamepad controller and the master device is the gaming console. The FPGA chip will reside in the gaming console as the master device. Before looking at the programming involved, we'll have to understand how this digital protocol works.

The 5 digital I/O lines are defined as: COMMAND, DATA, SELECT, ACK, and CLOCK.

The COMMAND line is used to send commands from the gaming console to the controller, typically asking for the controller device ID and current state of all buttons and joysticks. Each command is made up of 8 bits, or a single byte, and will be shown here in hexadecimal representation. The DATA line then sends back the requested information through a series of 8 bytes. Each byte holds information about the current state of all buttons and joysticks. The third line used is the SELECT line, which simply gets the attention of the controller by going low, and then stays low during the entire transmission. And the fourth line is the ACK line, which sends an acknowledgement from the controller to the console after every byte on the DATA line. Lastly, the CLOCK line is generated by the controller to keep all communication synchronized.

When implementing any digital communication protocol, it's very useful to have timing diagrams to help understand how all of the digital lines operate.

1. Timing diagram for example digital protocol.

Fig 1 shows a timing diagram for the first 5 bytes of data transmission. All values are written on the falling edge of the CLOCK line, and then read on the rising edge of the CLOCK line. The sequence begins with the SELECT line going low and the game console sending the first byte. The COMMAND line sends the start command with a 0x01 (hexadecimal 1) which corresponds to 0000 0001 in binary. All bytes are transmitted with the least significant bit first, so the actual COMMAND output is 1000000, with one bit for every falling edge of the CLOCK line. The gamepad controller reads each bit on every rising edge of the CLOCK line and responds to the start command (0x01) by sending its device ID on the DATA line. In this example, the device ID is represented by 0x41 (hexadecimal 41) which corresponds to 0100 0001 in binary. Since all bytes are transmitted with the least significant bit first, the actual DATA output is 10000010.

The second byte sent by the game console to the gamepad controller is 0x42, which asks for the current state of all buttons and joysticks. The actual COMMAND output is 01000010, which is then answered by the controller with a series of 7 bytes that include information about which buttons are being pressed. For example, a typical response from the controller might be 0x5A 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF, where the actual bits latched-in off the DATA line would be:

01011010 11111111 11111111 11111111 11111111 11111111 11111111

The first of these bytes (0x5A) only indicates that 6 bytes that follow it will be button information
After each byte (or every 8 bits) is sent, the controller also toggles the ACK line low for a single clock cycle to acknowledge the transfer of every section of data. After the final DATA byte is read, the SELECT goes back to high and the controller waits for the next sequence of commands from the game console.

Once we’ve understood how this example protocol works, we can begin to look at what would be involved for implementation. **FPGA implementation of our custom protocol**

As with any type of programming, there are several ways to implement digital communications protocols with several different architectures to choose from. For simplicity, we will use nested loop architecture, but a state machine architecture is also quite common. We will use a While loop as the main program loop, with double-nested For loops within it.

Once the five I/O lines have been instantiated in the FPGA application, they should all be initialized to logical high. Next, the outer-most While loop is defined, and all protocol related code can reside within this main application loop. As depicted by the timing diagram, the first action in the entire sequence of events is setting the SELECT line to low, and leaving it low until the entire transmission has taken place. The program will then enter double nested For loops, for which the outer For loop will execute 9 times for every byte being transferred between the game console and game pad controller, while the inner For loop will execute 16 times, with a single iteration for every CLOCK line transition.

On the first falling edge (high to low transition) of the CLOCK line, the first bit of the first COMMAND byte is written from the game console to the gamepad controller. On the second iteration of the inner For loop, the CLOCK line transitions from low back to high and the DATA line is read by the game console. By the end of the eight complete CLOCK periods, the gamepad controller has transferred the first complete COMMAND byte and has read a complete DATA byte. This sequence of events happens nine times for every iteration of the outer For loop; the final step is setting the SELECT line back to logical high to complete the overall transmission. Below is the graphical block diagram of the double nested For loops written in LabVIEW FPGA (see also the "sidebar" on LabVIEW FPGA at the end of this article).

![Diagram](image)  

2. Double-nested For loops for our digital protocol presented in LabVIEW FPGA  
(the falling-edge case is shown here).

Fig 2 depicts the double-nested For loops just described (the outermost While loop is not shown). The byte commands defined in the previous section are stored in the COMMAND register shown on the left-most part of Fig 2. The first element in this 9 element array is 0x01 (hexadecimal 1), followed by 0x41 (hexadecimal 41). The remaining seven elements in the COMMAND register are all 0xFF, to hold the COMMAND line high for the remaining part of the transmission. The gamepad
controller, which is the slave device reading the **COMMAND** line, only responds to the first two bytes in command sequence and simply ignores remaining seven bytes. The COMMAND register is wired to a tunnel on the outer *For* loop structure, where it is automatically indexed for each iteration of the *For* loop. Since this is a 9-element array, the *For* loop executes 9 times with each iteration corresponding to a single byte of transferred data.

The inner *For* loop will execute sixteen times, as indicated by the constant 16 wired to the count terminal on the top left-hand side. A green Boolean shift register is initialized to *FALSE* and then toggled back and forth to generate the CLOCK output values. A numeric shift register is initialized by a 0 and then used to index through each bit value for DATA and COMMAND bytes to be transferred. Each byte of the COMMAND register is converted to an 8 element array of Boolean data which is indexed and written to the **COMMAND** digital output line on every false case of the Case Structure.

The individual I/O lines are represented by digital I/O nodes, which are labeled accordingly. The delay times between each iteration of these double nested *For* loops is specified by a timer function, and can be specified in units of milliseconds, microseconds, or even individual ticks of the compiled FPGA clock rate.

![Diagram](image)

3. Inner *For* loop (*the rising-edge case is shown here*).

When the **CLOCK** line goes from low to high, the *TRUE* case of the Case Structure executes and a single bit of the incoming DATA byte is read from the **DATA** line. The DATA bit from every iteration is inserted into an 8-element Boolean array and the value of the numeric shift register is incremented to index the next bit.

Once 8 bits of a single byte are written to the **COMMAND** line and 8 bits of a single byte are read from the **DATA** line, the 8 bits in the Boolean array for DATA are converted to a single U8 number to be stored in DATA register.

The DATA register, shown on the right-most part of *Fig 2*, is used to store each byte of data read from the gamepad controller after every transmission sequence.

Once the **SELECT** line has returned to *TRUE*, all communication is over until the next time the value of **SELECT** changes. This application ultimately polls the state of the game controller at the execution rate of the outer *While* loop.

**Conclusion**
The ability to create a custom digital interface with FPGAs can completely alleviate the time and
expense associated with developing an ASIC-based custom hardware solution. FPGA-based hardware provides the flexibility and performance needed to communicate with almost any digital protocol and can interface with multiple protocols simultaneously. In addition, FPGA chips are reprogrammable and can adapt with any changes that might occur in a digital protocol specification.

**About LabView**

National Instruments LabVIEW is a graphical programming environment, optimized for engineering applications in control, design, and test. The LabVIEW FPGA Module compiles LabVIEW graphical block diagrams to execute on FPGA-based hardware, providing a visual representation of I/O control and data processing. Graphical programming uses visual structures like While Loops, For Loops, Sequence Frames and Case Structures to control program execution, and involves passing data between various functions using graphically drawn wires. This block diagram approach to FPGA programming is well-suited, in that it offers an intuitive depiction of the inherent parallelism that FPGAs provide.

**Compilation summary**

Device Utilization Summary:

- Number of BUFGMUXs: 2 out of 16 (12%)
- Number of External IOBs: 242 out of 484 (50%)
- Number of LOCed IOBs: 242 out of 242 (100%)
- Number of SLICEs: 1121 out of 14336 (7%)

Clock Rates: (Requested rates are adjusted for jitter and accuracy)

- Base clock: 40 MHz
- Requested Rate: 40.408938 MHz
- Theoretical Maximum: 73.877069 MHz

**Vineet Aggarwal** is a Data Acquisition Product Manager at National Instruments, focusing on implementations and features of high-performance PCI and PXI products. Vineet holds a Bachelor of Science in Electrical Engineering from The Ohio State University.