ED KLINGMAN

feature

FPGA Design from the Outside In

FPGAs enable everyone to be a chip designer. This installment shows how to design the bus interface for a generic peripheral chip.

When designing with an embedded microprocessor, you always have to take into account, if not begin with, the actual pinout of the device. Each pin on a given microprocessor is uniquely defined by the manufacturer and must be used in a specific manner to achieve a specific function. Part of learning to design with embedded processors is learning the pin definitions. In contrast, field programmable gate array (FPGA) devices come to the design with pins completely undefined (except for power and ground). You have to define the FPGA's pins yourself. This gives you incredible flexibility but also forces you to think through the use of each pin.

This article describes ways you can define FPGA pins and gives you an example of a straightforward project—an 8051-compatible peripheral bus. The project results in an FPGA-based peripheral device, the F51, that connects directly to an 8051 microcontroller and supports a variety of 8051 peripheral functions.

We can start designing the project using the Universal Design Methodology (UDM). Bob Zeidman wrote a Beginner's Corner on how to design an FPGA using UDM, so I thought we'd just plug our parameters into his outline. The result, shown in the sidebar, is the first step in our project.

The ins and outs of an FPGA

FPGAs can contain millions of logic gates yet every FPGA, regardless of the complexity of its internal design, must communicate with external devices through its I/O pins. Although this statement is trivially true, it doesn't follow that the I/O assignments are trivial. Indeed, the I/O pins on some FPGAs are the most complex part of the chip.

For all practical purposes, the internal logic gates are idealized; they accept 1's and 0's as input and produce a 1 or 0 as output. I/O pins, on the other hand, provide various drive levels, slew capabilities, and often specific interface ability for, among others, PCI and Low Voltage Differential Signaling (LVDS). Several glob-

al buffers exist that allow an I/O pin to drive "global" lines that span the FPGA (in contrast with shorter lines for connecting local elements). Such global lines are normally dedicated to clock lines or reset lines that typically span the device. Most other pins are connected to general-purpose I/O buffers.

Review of 8051 I/O

Although the 32 I/O pins of a typical 8051 microprocessor provide a complex set of capabilities, we'll focus on the 8051 data bus and associated control signal as shown in Figure 1.
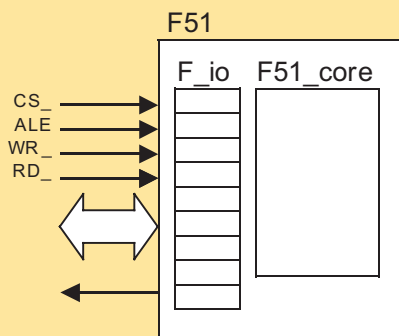
The multiplexed address and data bus operates in a straightforward manner. At the beginning of a bus

---

A UDM outline for our F51 project*

- An external block diagram showing how device fits into system:
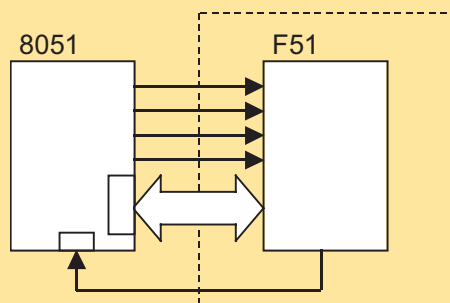
- An external block diagram showing each functional section:

- A description of I/O pins, including output drive capabilities and input threshold levels: **information is in the text of this article**
- Timing estimate, initialization setup and hold-times for input pins, propagation times for output pins, and clock cycle times: **20 to 100MHz typical I/O speeds**
- Logic estimate—gate count or chip count: **one chip, gates as needed**
- Physical specification; package type, physical size, connector requirements, etc.: **208 PQFP (208 pin—plastic quad flat pack)**
- Power consumption target: **unspecified**
- Price target: **$5 to $20 depending on quantity and gate count**
- Test Procedures: **to be specified in later article**

*   Outline is based on Zeidman, Bob, "Universal Design Methodology," Beginner's Corner, *Embedded Systems Programming*, Dec. 2003, p.55.

FIGURE 1 8051 data bus and control signals
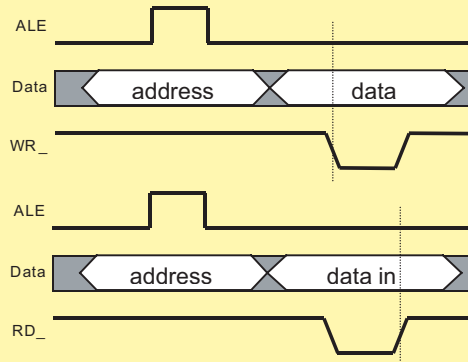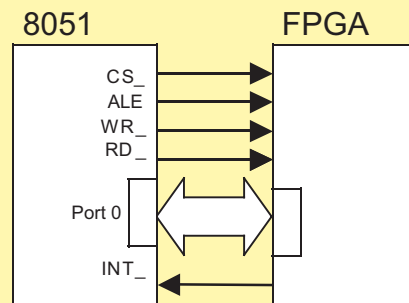

FIGURE 2 An 8051-compatible FPGA interface that allows an 8051 to read or write to an FPGA using 8051 data bus and control strobes

transaction the address latch enable (ALE) signal allows the 8-bit address to be latched from the data bus, then the bus is freed for data to be written or read by the 8051.

If the 8051 is writing data, it appears on the bus following the address information, and the active-low write signal, WR_, indicates that the data is available. The falling edge of WR_ is typically used to capture the data from the bus.

If data is to be read into the 8051, then, after the address appears, the 8051 leaves the bus in a high-impedance state, allowing some external device to drive its data onto the bus. The falling edge of the active-low read strobe signal, RD_, indicates that the bus is available; the 8051 drives the rising edge of RD_ to indicate that the data has been captured.

As Figure 2 shows, the simplest way an 8051 can interface to an FPGA is through its data bus (port 0), and the RD_, WR_, and ALE control signals. In addition, we assume that there are other peripherals, so we add a chip-select signal, CS_, which will tell our FPGA to pay attention to the data bus and control signals only when it (the FPGA) is selected. If CS_ is not low (active), the FPGA must assume that another peripheral device is being read or written and ignore all other 8051 signals. Finally, we provide an

It's easy to design an FPGA with pins that are always inputs or always outputs; bidirectional pins are much trickier.

output signal from the FPGA that will typically provide a "ready" indicator that may be used to interrupt the 8051. The 8051 signals will fit the FPGA like a hand in a glove. Each 8051 control signal will have a corresponding FPGA I/O pin "shaped" to respond to the signal. When we're done, we'll have an 8051-compatible peripheral interface that will serve for an almost infinite variety of possible FPGA-based peripherals.

Bidirectional I/O

It's easy to design an FPGA with pins that are always inputs or always outputs; bidirectional pins are much trickier. You'll need to know when to drive the pins, when to read from the pins, and when to leave the pins in a neutral high-impedance state. As a general rule, tri-state circuitry is not implemented inside FPGAs, for manufacturing process reasons. However, I/O pins typically offer tri-state capability. The desired effect of tri-stating internal signals is usually handled via multiplexers.

In order to construct a bidirectional I/O, we need to consider the
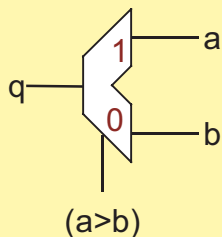
FIGURE 3 IF THEN ELSE construct

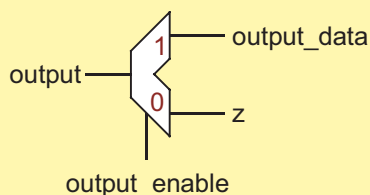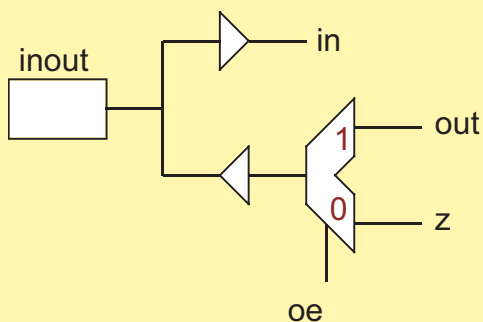**FIGURE 4** The same Verilog construct with signals renamed

FIGURE 5 The inout line is a bidirectional signal

major elements of FPGA logic: nets and registers. A *net* is a connecting element that follows a driving voltage. A *register* is a driving element that produces and maintains a driving voltage or logic level. Nets convey values, while registers store values.

The Verilog hardware-design language contains a construct familiar to all C programmers, the IF THEN ELSE construct. Brian Kernighan and Dennis Ritchie (*The C Programming Language, Second Edition*, Prentice Hall Software Series, 1988) introduce the conditional expression, written with the ternary operator "`?:`" via:

```
expr1 ? expr2 : expr3
```

as an alternative means of treating the IF THEN ELSE construct, where `expr1` is a condition selecting either `expr2` or `expr3`. If the condition is true, select the second expression; otherwise select the third expression.

The Kernighan and Ritchie example considers the statements:

```
if ( a > b )
            q = a;
else
            q = b;
```

and rewrites them as:

```
q = ( a > b ) ? a : b ;
    / * q = max (a,b) */
```

Verilog provides the same construct, but Verilog programmers tend to think in terms of the more graphical interpretation as shown in Figure 3. The meaning is exactly the same as in C. If the selector, $a > b$ is true, output q is assigned value a, else value b.
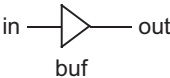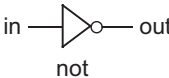
Now let's take the above construct shown in Figure 3 and replace the condition ($a > b$) with the `output_enable` signal and rename variable a to `output_data`. This is simple enough, but now let's do something useful and let variable b become the high-impedance state, z. If q is the output, our diagram changes to Figure 4.
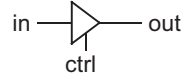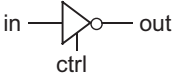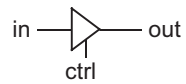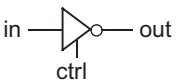
Now when the `output_enable` is true, the output data appears at `output`, else the output is in the high-
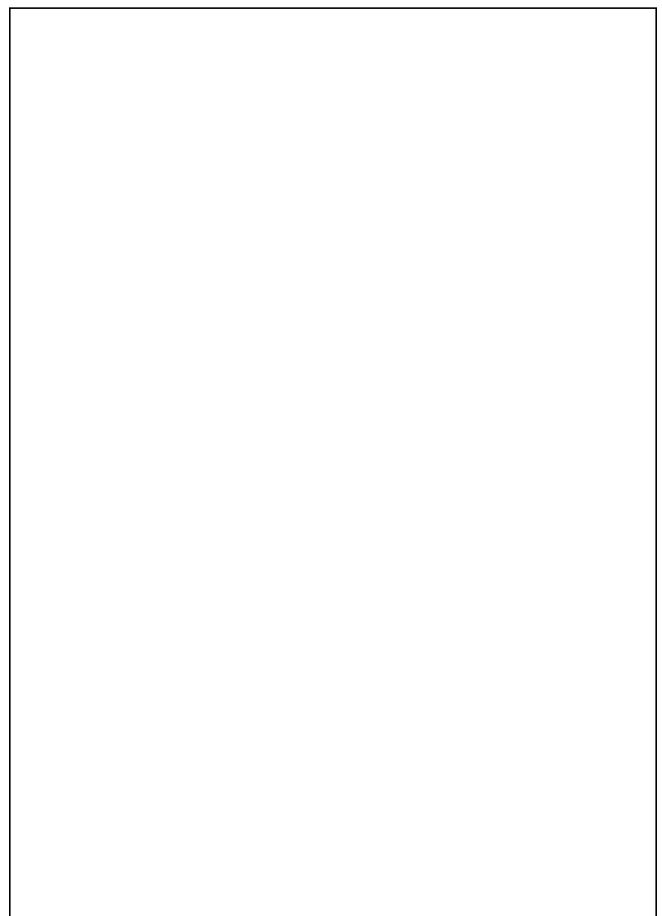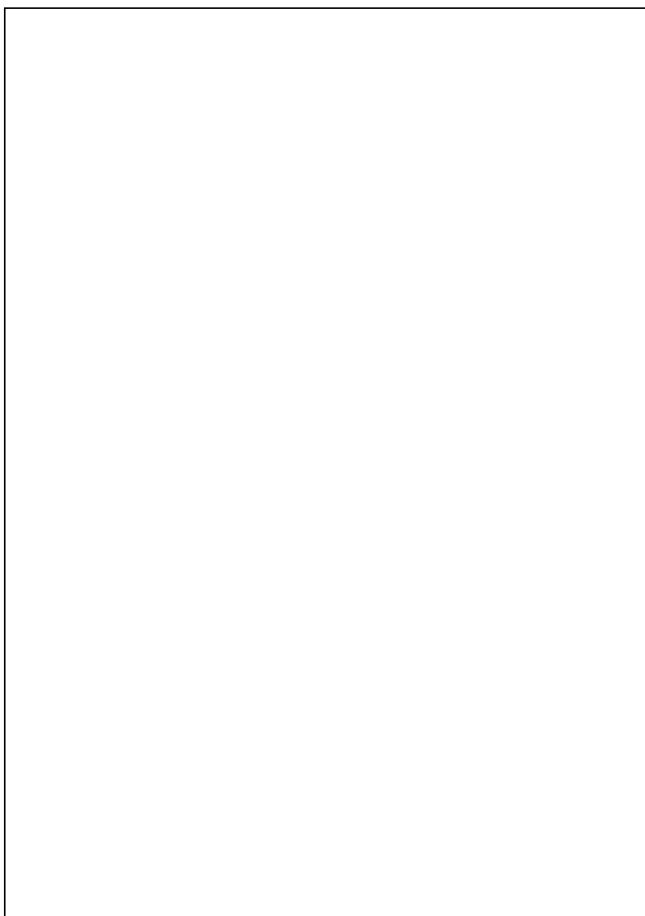
For those who aren't sure about what high-impedance signals do, here's a primer. Only one source at a time should drive a signal wire or else they'll fight with one another.

FIGURE 6   Verilog primitive buffers

FIGURE 6   Verilog primitive buffers

impedance state. For those who aren't sure about what high-impedance signals do, here's a primer. Only one source at a time should drive a signal wire or else they'll fight with one another. You then get into conflicting signals, questions of relative drive strength, and other issues best avoided. A high-impedance signal has effectively zero strength, so a wire in the high-impedance state can be driven by another signal.

We can now tie the `output` wire in Figure 4 to another wire; let's call it `input` and add a buffer as shown in Figure 5. Notice that we've renamed the `output_enable` symbol `oe`.

The *inout* line is a bidirectional signal that can be driven from somewhere outside the chip when `oe` is false or by output data—simply called `out` in Figure 5—when `oe` is true. This is exactly what we need for our bidirectional 8051 data bus. When the 8051 writes to us, it will drive data into the FPGA. When the 8051 reads from us the `out` data should appear. This suggests that the output-enable signal, oe, should be some function of the 8051's read-strobe signal, `RD_`. The rectangle or "pad" on the schematic represents an I/O pin that will be connected to a corresponding 8051 data bus pin.

Fortunately for us, the elements described above are available as Verilog primitives, specifically `buf` and `bufif1`. The simple buffer, `buf`, simply passes its input to its output.

The `bufif1` primitive has two inputs, control and data, and one output. If the control input is "1" the output passes the same value as the data input. When the control input is "0", the output is *Z* (high-impedance).

Figure 6 shows the Verilog symbols are straightforward, as are the corresponding truth tables. Note that Verilog variables can assume four states; 0, 1, *X*, and *Z*. Of course 0 and 1 are the usual logic states, also known as low and high or false and true. The two new state states are *X* and *Z*. *Z* is the high-impedance state, which can be overridden by any driven signal, 0

LISTING 1   BusPin module

```
module BusPin ( IN, PIN, OUT, OE );

    output IN;              // signal from module into FPGA

    inout PIN;              // bi-directional port represented by PIN

    input OUT;              // signal from FPGA into BusPin module

    input OE;               // control signal determining BusPin direction

    buf i_buf ( IN, PIN )   // IN = PIN

    bufif1 o_buf ( PIN, OUT, OE );          // PIN = OE ? OUT : Z

endmodule
```

While actual hardware is always in some state, the concept of the unknown state is extremely useful during hardware simulations.

FIGURE 7 Symbolic representation of bidirectional I/O pin
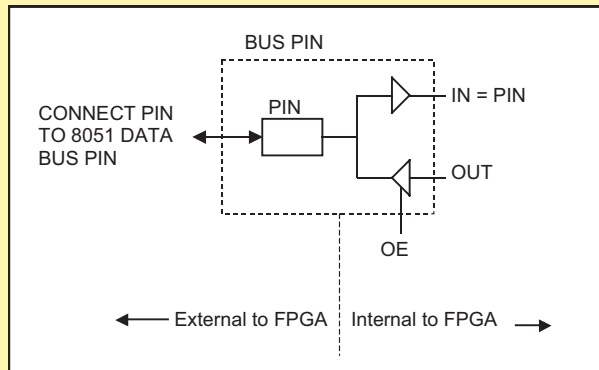
or 1. Thus, if a wire in a high-impedance state is tied to (wire-or'ed with) another wire driven high or low, the result is determined by the other wire state. This technique provides a convenient means of tying multiple signals together in such a way that they will not fight. Of course, the control signals must be manipulated such that all but one of the signals is forced to the high-impedance state at any time.

The states 0, 1, and Z are generally familiar, but what is X? X is the unknown state. If logic is poorly designed, a system can get into an unknown state, and such states generally propagate as unknowns. While actual hardware is always in some state, the concept of the unknown state is extremely useful during hardware simulations. For example, if a simulator shows valid signals in green and unknown signals in red, then it's

LISTING 2 Using Verilog's high-level keyword "assign"

```
module  IN_BUF ( OUT, IN );
        output          OUT;
        input           IN;
        assign          OUT = IN;

endmodule

module  OUT_BUF ( OUT, IN );
        output          OUT;
        input           IN;
        assign          OUT = IN;

endmodule
```

LISTING 3 Format for instantiating modules

```
module_name   instance_name (
      .port1_name ( wire1_name ),
      .port2_name ( wire2_name ),
      .
      .
      .portN_name ( wireN_name )

      );
```

easier to determine just when and where a logic error occurs. Because unknown inputs produce unknown outputs, the red waveforms typically spread quickly as the design falls apart.
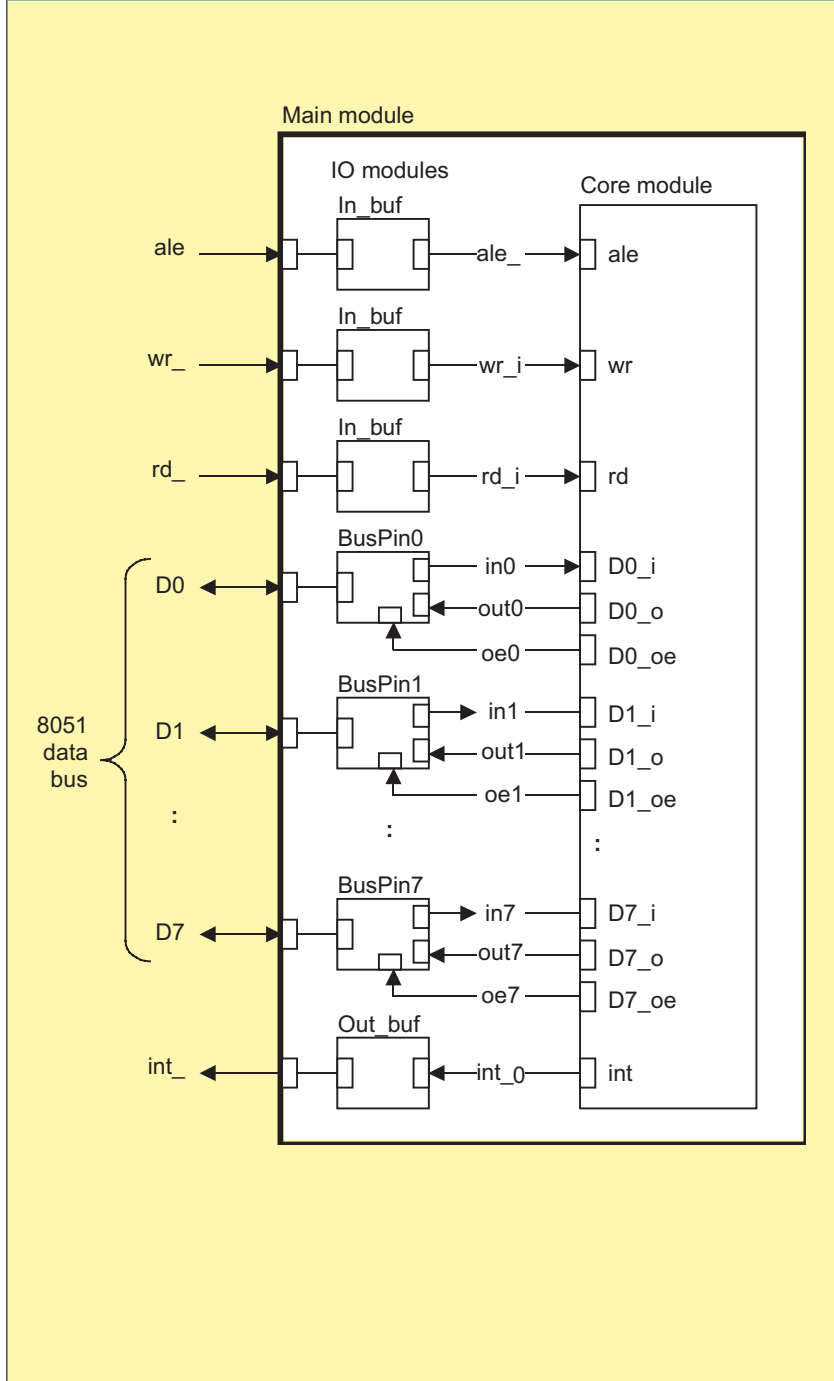
Instantiation of elements

When designing systems from primitive elements or combinations of primitive elements, we need to instantiate, or make instances of, our elements. This is done by specifying the *type of element*, followed by the *name of the element*, followed by a *list of arguments* in parenthesis. The output of primitive elements is typically listed first. For example, we can instantiate:

```
buf     i_buf   ( out, in );

bufif1  o_buf   ( out, in, oe );
```

User-designed subsystems, or modules, have ports for communications with the world and are defined by specifying the type `module`, followed by the user-specified module name, followed by a port list in parenthesis, followed by port definitions, port names, and other module specifications, all terminated by the keyword `endmodule`. If we assume that "PIN" is an actual I/O pin of the FPGA and we wish it to be one of the bidirectional pins connected to the 8051 data bus, our description of a "BusPin" module can be achieved as shown in Listing 1.

Because it's expressed in terms of the Verilog primitives `buf` and `bufif1` any Verilog synthesizer can map this module into FPGA elements. (I'm indebted to Muzzafer Kal for this particularly clear implementation of a bidirectional I/O pin shown in Figure 7.) Note that, in addition to being able to handle this generic bidirectional I/O pin, most FPGA manufacturers provide specific I/O pin primitives that can be used in place of the generic module. Because the keywords associated with manufacturer-specific I/O differ for each manufacturer, we'll use the generic *BusPin*.

FIGURE 8 Illustrating the FPGA main module, I/O modules, and core module

Regardless of the programming language you use, a subroutine that's never called is a meaningless piece of code. In the same sense, a module that's not connected to anything is a meaningless module.

In similar fashion, we can define input pins and output pins as generic modules. In these cases, because we don't need to handle the tricky bidirectionality, we don't even need to specify the low-level primitive buffers, but can simply use the high-level Verilog keyword "assign" as shown in Listing 2.

The first thing you might think, when looking at these module definitions is, "why use two modules that have identical definitions but different names?" Because we'll have input signals from the 8051 and output signals to the 8051, things will get very confusing if we try to use the same module name for both. As you'll see, things get confusing enough as it is.

Remember that although we're defining modules for use with the FPGA design, the modules will provide INPUT and OUTPUT channels to and from the FPGA, and should be named from this perspective, that is, IN and OUT are defined from the FPGA's point of view.

Modules, connectivity

Regardless of the programming language you use, a subroutine that's never called is a meaningless piece of code. In the same sense, a module that's not connected to anything is a meaningless module. Subroutines that are actually called maintain a connection in the form of a return address on a stack. For subroutines this is essentially hidden information and changes dynamically as the routine is called from different locations. For hardware modules the information is static and not hidden. Each port of a module must be associated with a connection, and the connection is typically described via the keyword "wire." When we want to include an instance of a module in a design, we specify each port name with the associated wire name in parenthesis. In general, modules are instantiated in the format shown in Listing 3.

This is a very important format, and you should return to study it if you get confused.

Tying things together

We now have enough pieces defined to begin describing the F51 device that forms the basis of our FPGA-based 8051 peripheral. Before proceeding, let's mention one more convention. It's considered good form to define the "main" module and include only the I/O pins and a core module, with no logic, per se, outside of the core module. Figure 8 illustrates the FPGA main module, I/O modules, and core module. Listing 4 shows the format for instantiation modules.

Little project, big payoff

Although this may seem like a lot of work for little result, that's not really the case. In order to reach this point, we first decided upon a goal, which, in this case, was to create an 8051-compatible peripheral interface based on an FPGA. Next we reviewed the 8051 bus and control protocol. Realizing that we needed a bidirectional data bus, we investigated Verilog primitives from which such a bus could be built. Then we described the manner in which such primitives could be combined into a module representing an I/O pin. Because we need multiple pin modules and will need other functional modules, we then described a key formalism by which any main module is decomposed into pin modules

Although this may seem like a lot of work for little result, that's not really the case. . . . I can sincerely say that I wish someone had given me this design for free!

LISTING 4 Format for instantiating modules

```
module F51      (    ale,
                     wr_,
                     rd_,
                     data,
                     int_
                );

    input    ale;
    input    wr_;
    input    rd_;
    inout    [7:0] data;
    output   int_;

    wire     ale_i;
    wire     wr_i;
    wire     rd_i;
    wire     int_o;

            IN_BUF              ale_in ( ale_i, ale );
            IN_BUF              wr_in ( wr_i, wr_ );
            IN_BUF              rd_in ( rd_i, rd_ );

                OUT_BUF             int_out ( int_, int_o );

                BUS_PIN             data0 ( in0, D0, out0, oe );
                BUS_PIN             data1 ( in1, D1, out1, oe );
                                    .
                                    .
                BUS_PIN             data7 ( in7, D7, out7, oe );

    F51_core F51_core ( .ale ( ale_i ),
                .wr ( wr_i );
                .rd  ( rd_i ),

                DATA_BUS

                .int ( int_o ))

endmodule
```

---

ules, representing connection to the outside world (that is, external to the FPGA) and a core module that will perform all logical functions. This required us to review the inter-module connection formalism by which ports of different modules are connected by wires using port names and wire names. We ended up with a Verilog implementation of a universal FPGA peripheral to an 8051 microprocessor. While the internal logic hasn't yet been developed, that will be the easy part.

Note that we've ignored several pins that will always be present. These include ground and supply voltage pins, as well as a clock-input pin and a reset-signal input pin. In addition, unless the peripheral we're designing is purely computational (accepting data from the 8051 and returning a result), we'll also need more I/O pins to interface to a controlled subsystem. These pins would, of course, be specific to the device being monitored or controlled, and therefore can't be universally specified. The addition of a clock buffer and a reset buffer should provide a nice exercise for you to implement as a review of the principles involved.

In a future article we'll look into the F51 core module to see how to actually use these signals, as well as describe a test bench to exercise the F51 chip.

Finally, if you still feel that we have reached a minimal result, I can sincerely say that I wish someone had given me this design for free! **esp**

Ed Klingman worked as a research physicist at NASA for seven years, before founding Cybernetic Micro Systems, Inc, now celebrating its 25th anniversary. He's the author of the Prentice-Hall textbooks Microprocessor Systems Design, Vols. I *and* II *and numerous technical papers, and has been awarded 20 U.S. patents. You can reach him at klingman@geneman.com.*