

CIRCUIT CELLAR®

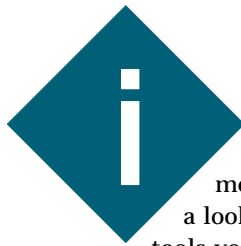
THE MAGAZINE FOR COMPUTER APPLICATIONS

LEARNING THE ROPES

Ingo Cyliax

The Foundation Environment

Ingo continues his FPGA tutorial by looking at the design tools needed to begin designing with FPGAs and CPLDs. He'll get you started with Xilinx's Foundation environment.



In this installment, I want to take a look at the design tools you will need to start designing with FPGAs and CPLDs. This is actually a pretty complicated topic, possibly even more involved than the architecture overview I covered last time.

DESIGN TOOLS

FPGA/CPLD vendors are primarily interested in developing what is called the back end. It's the pieces of software you'd need to take a design that has been captured either in schematics or high-level description language (HDL) into a form that can be used to program a part.

However, because the EDA tool industry is fairly dynamic and FPGA/CPLD parts keep evolving, the software developers for back-end tools fight a battle on two fronts. They not only have to develop libraries for different EDA tools and simulators, they have to come up with better fitters and routers for new parts that have more resources and complex architectures. All while maintaining

an image that it's "easy."

Evolving interchange standards like EDIF and Verilog/VHDL help standardize interfaces to CAD tools and simulators. For example, a CAD vendor can now provide an EDIF-compatible library of design elements that use Verilog or VHDL to implement the models necessary for simulation environments. Of course, there are still some glitches, but things are getting better.

Realizing that there is a potentially large learning curve, FPGA/CPLD vendors are also offering cost-effective, entry-level design environments. These are complete packages with design entry, simulators, libraries, and the back end. Later, I will show you one of these packages.

These design environments are certainly good for learning FPGA/CPLD development and, in many cases, actually designing fairly large designs. However, if your organization already standardizes on one CAD environment, these packages may not be of much help, and you're back to the integration game. If you're one of these organizations, you probably have the resources to handle the integration to a specific FPGA/CPLD back end.

For the rest, going with a FPGA/CPLD design environment from the FPGA/CPLD vendor is probably one of the best ways to go. There's a lot of support on the vendor's web site for these packages, because if you get good at designing for their parts, they hope to sell a large quantity of them.

The particular package I want to look at with you this month is Xilinx's Foundation environment. Foundation is a representative of other environments. It's also very complete. Finally, if you're a student or someone that just wants to learn this stuff, Xilinx has a student edition

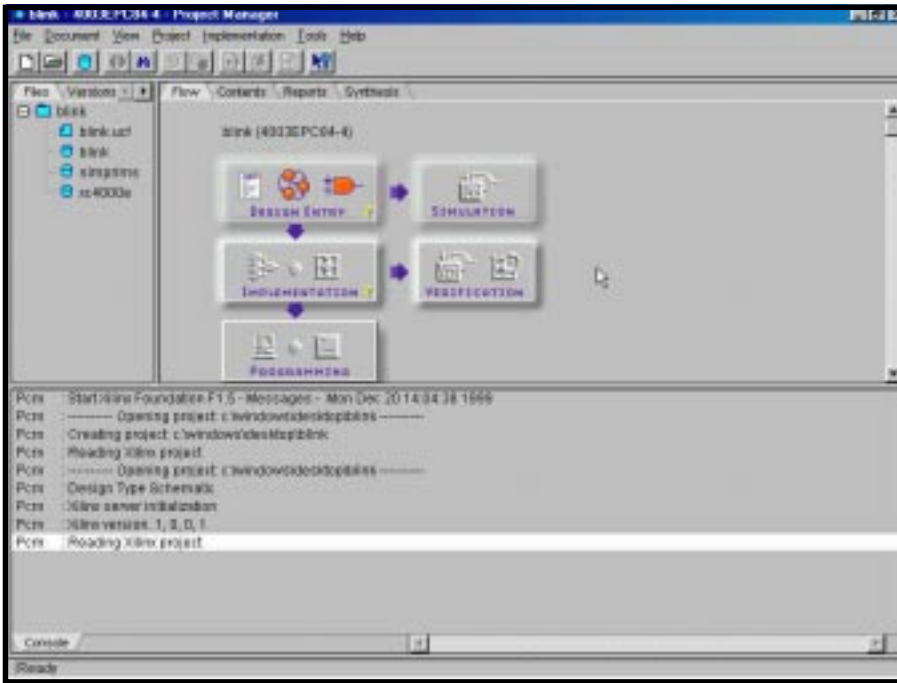


Photo 1—Each block has a small status icon to the bottom right. When you have completed the step, a check mark will show you what's done and where to go next.

of this package, which sells for under \$100 and includes a good tutorial style text book that will work you through the tool.

The student edition is not a crippled version of Foundation, except that it only supports some of the small/medium-sized devices. After you come up to speed and want to try a commercial design, you should go ahead and purchase the production version. It comes with a support contract, where the student version doesn't.

Let's get started.

DESIGN FLOW

Starting a FPGA design might be a little overwhelming at first. Some of you have been using simple PLDs. With a PLD, you create a file that contains a pin-definition section, equations that implement what the output pins do, and a section with some test vectors for testing your design. The PLD assembler will then generate a JEDEC file from your description. The JEDEC file, including the test vectors, would then be loaded into a PLD programmer that programs and tests the PLD.

Well, designing for FPGA/CPLD with Foundation is not much different. The details are all different, of

course, but the tool does a good job of guiding you through the process.

When I first started the project with Foundation, I was presented with a dialog box about the kind of design I wanted to implement. This included questions about whether I wanted to do a schematics- or HDL-based design. I will walk you through a schematics-based design, and then in later articles, I'll revisit doing an

HDL-based design (VHDL/Verilog) and go over its issues.

When I chose a schematics-based design, I also needed to decide what kind of FPGA or CPLD I would be implementing this design on. At this point, because the tool needs to know what device libraries to pre-load with your schematics editor, you only need to get the device family right. You can always change your mind later about the device size and package type.

Xilinx has several device families, and if you're just starting out, you'll likely be designing with XC4000E FPGAs or XC9500 CPLDs. The XC4000E is a 5-V SRAM-based FPGA, and the XC9500 is a flash memory-based CPLD. The architectures at the chip level are different between the two. (You should refer to last month's column to see why and how.) Even though the devices have different architecture, Foundation does a good job isolating the user from the difference.

Xilinx uses a universal library scheme. The same design components are available at the schematics level. This is great for the beginning designer. If I need a 16-bit adder, I'll chose the 16-bit adder block and won't worry about how it's implemented. Of course, if you're doing high-performance designs, then you

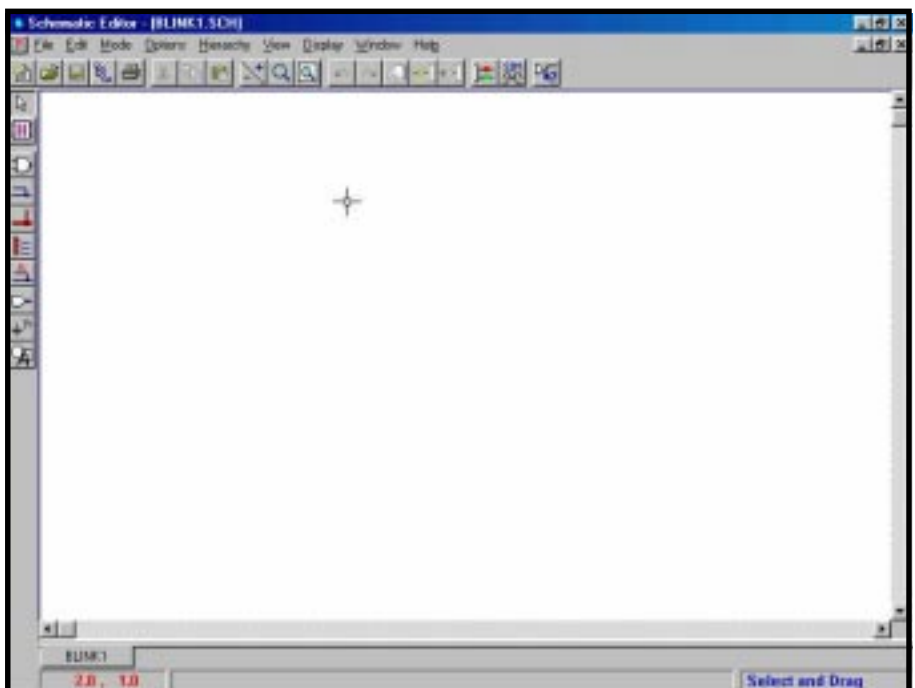


Photo 2—By selecting the schematic option, you get a blank page to start from.

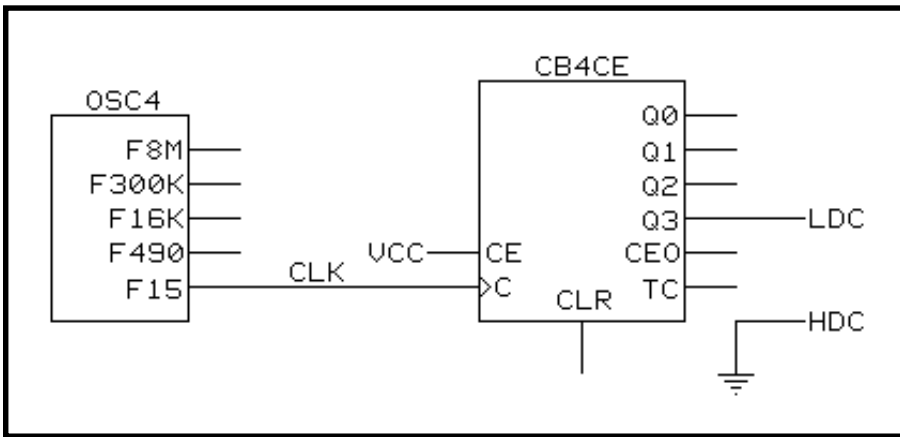


Photo 4—The schematic editor lets you select components to drop into the drawing from a symbol library or a component palette.

might want to use more device-specific architectural features.

Once you have chosen the device family and design style, the project manager will display a top-level situation display. On the left, there is a browser view of the design components, which shows the schematics sheets and libraries. On the right, you'll see a block-level overview of the tool chain (see Photo 1).

Each block has a small status icon to the bottom right. When you have completed the step, a check mark will show you what's done and where to go next. At the top of the flow, there is a design entry button and a simulation button. The design entry button actually has three sub-buttons—one for schematics capture, one for HDL, and one for a graphical state-machine editor.

Even though I chose to do a schematic project, I can still use HDL and a state-machine editor with specific sub-components of the design. In effect, I can mix and match design entry, as long as the top-level design is a schematic. The project manager keeps track of what components are needed to implement a design.

When I click on the schematics button, the project manager starts a schematics editor for me, which at this point is blank (see Photo 2). If you have room, you can also install sample projects, which are a good collection of FPGA/CPLD designs in both schematics- and HDL-design entry. There are also some mixed-entry projects.

Any FPGA or CPLD design needs

some I/O. (These are the pins that get connected to the outside world.) You could design a project, which only implements a function internally to the chip, with no I/O pins. However, if you do this, the Xilinx implementation tool will optimize the design out. The tool figures that because there are no external signals, the design doesn't really do anything and all of the logic isn't necessary. This is not productive, so let's do something really simple.

A design that I usually implement for the first time for many projects, is an LED blinker. This is a simple design that takes an internal oscillator (OSC4) symbol and prescales one of the outputs with a counter. This is a

XC4000-type design, and on XC4000-series devices there are two pins high during configure (HDC) and low during configure (LDC). When the SRAM-based FPGA is in its configuration state, these pins will have a high on HDC and a low on LDC. I usually connect a bicolored LED (red/green) to these pins with a 300-ohm resistor, so that it will light the red LED when it's in configuration state.

I then connect the most significant bit of the counter to the LDC output and set the HDC to a logic zero in the design.

The LED will be red whenever the FPGA is not configured. Once I load the blink design into the FPGA, it will blink green whenever the MSB of the counter is zero. The oscillator has a 16-Hz output, so a divide by 16 counter will make the LED toggle at a 1-Hz rate.

We'll talk more about actually programming the devices next time. Let's now look at what it takes to implement the blink design.

As I mentioned, when I first start the schematics editor, the sheet is blank. I first load the OSC4 module. To do this, I bring up the library list with the button that looks like a simple gate. I browse and select the OSC4 symbol and drop it on my sheet. Then, I go back and find a 4-bit counter and

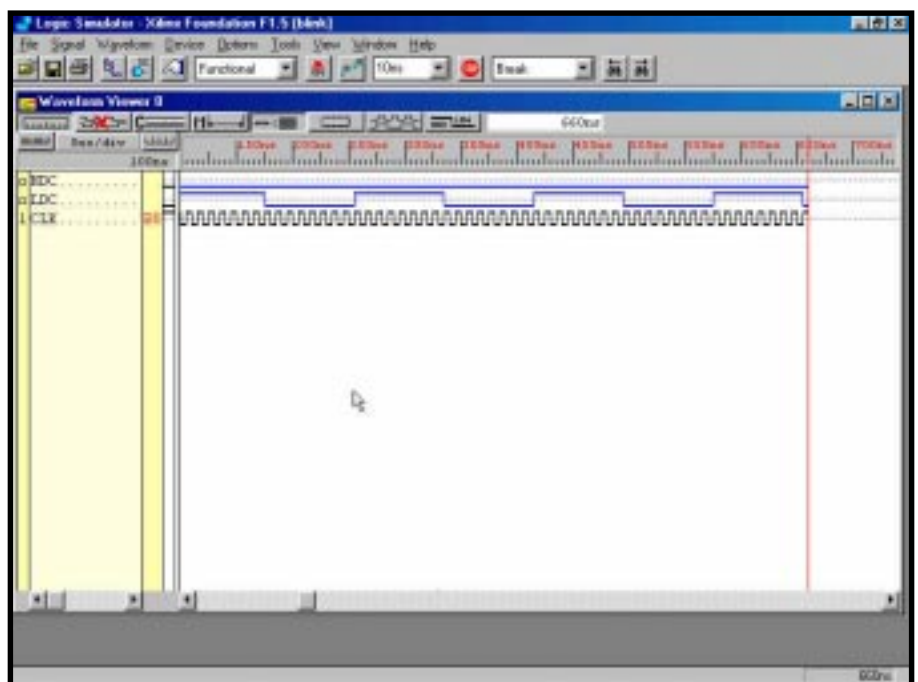


Photo 3—The Functional Simulation button will produce a simulation like this if everything was done right.

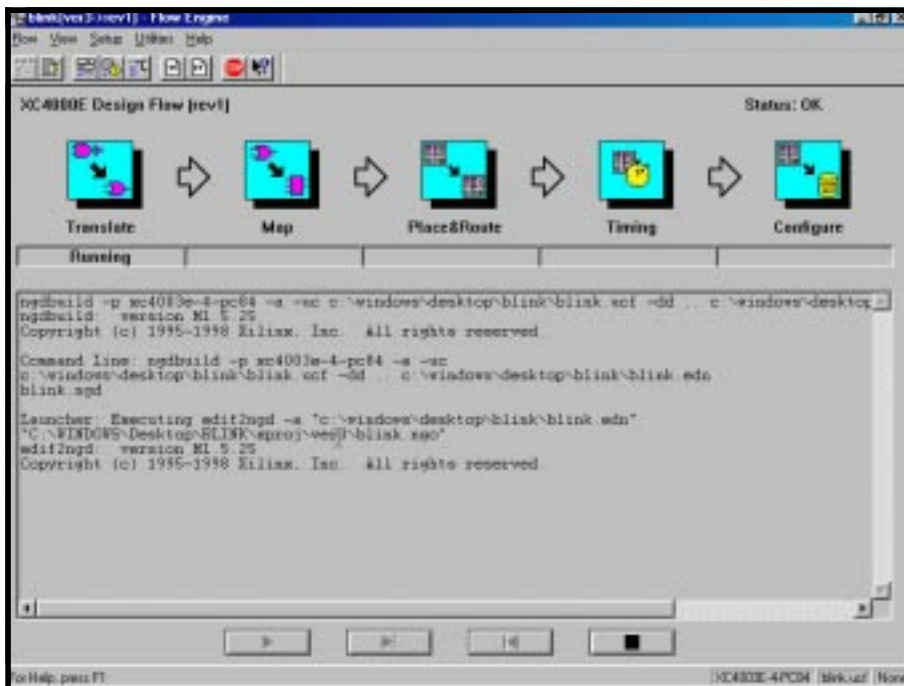


Photo 4—Here's a look at the design manager's feedback as the steps are finished.

drop it on the sheet as well. Then, I draw a wire from the F16 output to the clock input of the counter. I also need to wire a logic one to the counter enable. I can do this either with a library symbol for V_{CC} or by selecting a V_{CC} from the palette.

Then I wire the MSB bit of the counter to an I/O port. There are a couple of ways to do this. The easiest is to use a hierarchical port from the palette and drop it on the design. The implementation software can infer an I/O port from any hierarchical port at the top level of a design. I connect the hierarchical port to the counter output. I also add another port and wire it to ground, which comes from the palette or the library. Now, we highlight each port in turn and give it a name. The port connected to the counter is named LDC, and the other port HDC. Figure 1 shows the schematic for the FPGA design.

The other way to do this is to select an OPORT and an OBUF from the library and wire them up to the design. The advantage with this method is that I can label the port's pin numbers in the design with an attribute. Labeling the pin numbers in the design is not the preferred way of doing this. It's better done in the user constraint file for the design. I also

use the user constraint file when using hierarchical ports for the I/O pins.

The user constraint file is a text file that is used to give hints to the implementation software about how we want our design routed. To add a pin constraint, I simply add a couple of lines that say:

```
NET hdc LOC=P36;
NET ldc LOC=P37;
```

If I don't specify where the pins go, the fitter will automatically assign pins. This is fine during the initial design phase when you don't have the design wired to anything specific. However, the HDC/LDC pins are always at a specific pin and need to be "tied down". (Please note that the above pin numbers are from a specific package type—84 pin PLCC, and you'll have to look at the datasheet or book to find out where they go on the part you want to use. The Xilinx databook is available at www.xilinx.com.)

So, now that I have entered the design, I'll want to simulate it. I click on the Functional Simulation button, and it will check the design and extract the data necessary for the simulator. If it finds a problem in the schematics, it will print a warning or

error message in the log (bottom of screen) and mark the schematics in the browser with an error icon. Of course, everything will be OK and the simulation tool will start up.

Although you can interactively drive the simulation, it's better to write a simulation script. To do this, you launch the script editor (under Tools). This editor will start a wizard for you and guide you through creating a script file for your design. Just select HDC and LDC as the signals you want to watch in the wizard. After creating and saving the simulation script, run it. That can be done through the script editor tool or from the File tab in the simulation tool. You should end up with a simulation like that in Photo 3.

Now that I'm sure the design does what I want it to do, it's time to implement. I click on the implementation tool button. This causes a net list to be extracted from the design and the design manager to be launched. A net list is a file that represents the design after it has been flattened and all of the library symbols have been resolved. It enumerates the low-level components, and now they are connected to each other. For schematics-based designs, it will generate a EDIF net list. (If you want, you can look at this file with a text editor.) Foundation also deals with other net list formats. XNF, which is Xilinx's text-based net list format, is one of them.

In the dialog box just before launching, we can adjust the parameters that effect how our design is implemented. We can also change the specific part type we are using. Once we accept all of the dialog parameters, the design manager launches and starts working on the net list. Photo 4 shows the design manager at work.

The first step the implementation process does is read the net list, call up any components that have been referred to in the net list, and convert everything into an internal database format. It also does some sanity checks on your design. For example, it will check to see if you have more than one component driving a signal, which could be a problem. It also

checks to make sure that all of the subcomponents are present.

The next component in the tool chain is the mapper. This does some optimization on (flattens) the design. It will try to eliminate logic if it's redundant or inverted. The object at this point is to fit all of the logic in the design into a hierarchy of 4- and 5-input LUTs (or product terms in CPLDs). The mapper can fail, if your design is too big to fit a specific device. Once this is done, we go to the place-and-route phase.

Here the tool will figure out which physical LUT to program and what wires to use to connect everything together. At this point, if the design is very complex, it may decide that it needs to use an extra LUT or extra I/O resources as a routing resource. Also, it uses the constraint file to lock down where the designer might want a LUT or an I/O pin. Furthermore, the constraint file can be used to tell the router which networks need to have priority routing to meet specific timing constraints. The router will fail if it can't figure out how to route your design, and routing can take a long time if your design is complex.

Once the design has been mapped and routed, we can extract timing information from the design. The tools can give fine timing information about every possible CLB, flip flop, and route in the chip. In many cases, however, we just want to know the critical feedback delay in a synchronous design. In effect, how fast it will run. The results for the timing analysis are saved in a file, as is the information from the other phases.

The design then gets converted into a configuration file. This is called the "bit" file, even though it also contains symbolic information that a hardware debugger can use. We'll look at the Xilinx hardware debugger next time.

Once we have finished implementing the design, we can go back to the simulator and simulate the actual implementation, including the timing information. The simulator can also generate vector files in the functional simulation that we can run against

the timing simulation to verify that our design will meet whatever timing requirements we have.

Next time, I'll show you how to program a device with the blink design. ☒

Ingo Cyliax has written for Circuit Cellar on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego-based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. You may reach him at cyliax@derivation.com.

SOURCE

Xilinx, Inc.
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com