Getting a handle on HDLs

BRIAN DIPERT, TECHNICAL EDITOR

Programmable-logic chips and designs are growing more complex. As a result, you'll sooner or later need to add HDL expertise to your skills if you want to keep hitting those project deadlines. For this Hands-On Project, I learn VHDL, complete a mixed logic and embedded-memory design in an FPGA, and share observations along the way.

After hearing innumerable hardware-description-language (HDL) pitches from silicon and software vendors and watching the schematic-versus-synthesis debates at industry conferences and on newsgroup comp.arch. fpga, I decided to take the plunge, learn an HDL, and form my own opinions (see **sidebar** "Synthesis strengths and shortcomings"). The resulting project, which I cover in a two-part article series beginning here and concluding in *EDN*'s Sept 11, 1998, issue, comprises several steps:

1. Refresh schematic skills last used during an ASIC-design project several years ago and develop HDL expertise.
2. Use that expertise to create schematic and synthesis versions of a programmable-logic design, both based on a common specification.
3. Ensure functional compatibility between the two designs, although they might differ in maximum operating frequency and sustained transfer rate.
4. Compare efficiency and performance results of the two alternatives and quantify time to completion when possible.
5. Run the resultant VHDL source code through multiple vendors' synthesis compilers and compare their results.
6. Give the code to the respective vendors and challenge them to improve my designs.

I had also hoped to compare power consumption for the design alternatives. After further research, however, I realized that today's tools provide only rough estimates and that I'd need to bench-test each design with an oscilloscope if I wanted to obtain numbers that were accurate and, therefore, meaningful. Realizing that this effort was beyond the scope of the available time and not wanting to base conclusions on questionable approximations, I shelved this part of the project (see **sidebar** "Current status, future plans").

My first task was selecting an HDL. I decided on VHDL, because it seems to be the most common language that PLD, FPGA, and synthesis vendors advocate, especially in their low-cost, entry-level tool suites. Therefore, I hoped that this choice would be directly applicable to more readers. The choice between Verilog and VHDL usually comes down to personal coding preference, although each approach has its secondary merits.

Now for the silicon platform. I targeted a design having at least 10,000 gates. Indus-

try pundits call this gate count the schematic "pain threshold" at which HDLs begin to become attractive, also depending on desired design efficiency and performance. This gate-count complexity rules out CPLDs. I settled on Xilinx's XC4000XL FPGA architecture, because I hoped that its pass-transistor-based interconnection and mix of short and long routing networks would expose any design-compilation differences between schematic- and synthesis-design versions, as well as among synthesis vendors' implementations. Xilinx's XC4000XL FPGAs also provided a range of gate counts in case my design ended up larger or smaller than I had initially estimated. If I had extra time on my hands, I could even port my design to Xilinx's XC4000E or Spartan FPGAs, which differ from the XC4000XL in the amount of available routing. This experiment would let me modify the routing variable and hold all other factors essentially constant.

I'm a newcomer to HDLs, excluding Abel and Palasm work, and my programmable-logic experience is restricted to much less complicated devices and designs. Realizing, therefore, that I represented only one end of the spectrum of *EDN* readers' expertise, the answer was to ask the programmable-logic consultant company HighGate Design to join me in the project.

Principal Engineer Stephen Wasson has completed hundreds of FPGA designs, many using Xilinx devices and extending back to that company's earliest days. Stephen is also the initial developer of Xilinx's PCI core. His primary motivation in working on this project was to satisfy his intellectual curiosity, because he'd encountered the same synthesis-versus-schematic de-bates I had. The project output would also provide him a testbench for future EDA-tool and PLD benchmarking at his company.

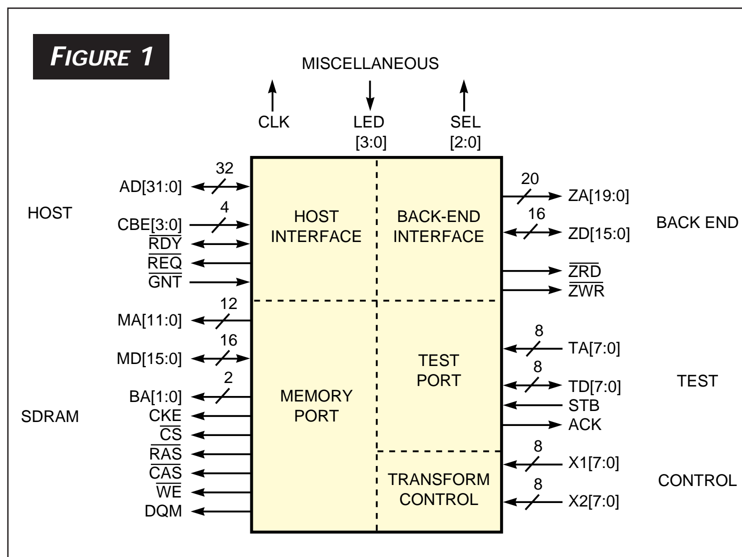
Stephen and I began discussing this project soon after the January 1997 DesignCon conference. He agreed that, as I had planned, HighGate Design would target development of both schematic and synthesis designs. He also accepted my challenge to create our common architecture specification. Stephen and I both wanted the design to contain a mix of functions—decoding logic, counters and state ma-chines, arithmetic units, and memory blocks—to level the playing field. Because Stephen doesn't use HDLs, that design task would fall to fellow HighGate Design Engineer David Holmes. By making the VHDL de-signs modular, we could also compile design subsets and identify logic-dependent differences among vendors.

Next, we considered tools. I'd used Viewlogic's WorkView Office in past ASIC designs, and Stephen also prefers it. Moreover, Viewlogic wanted to participate in my project, so that part of the decision was easy. We used version 7.40, along with software updates we obtained via the company's Web update feature. Synthesis was a tougher challenge. Paranoia about having the results published drove one large vendor to immediately respond, "Thanks, but no thanks." Another company bounced back and forth several times over a multiweek period before definitively declining. Synopsys, however, was enthusiastic about the project, so we decided to use the company's FPGA Express 2.01.

FPGA Express contains an HDL text editor—sort of. If compilation creates errors and warnings, you can click on each message to bring up a window that displays the offending line and enables you to edit the file. However, you can't create an HDL file from scratch. Instead, I used EditPad from Jan Goyvaerts (www.tornado.be/~johnfg), an enhanced alternative to Windows 95's Notepad. EditPad worked well for both my coding and my article-

@a glance

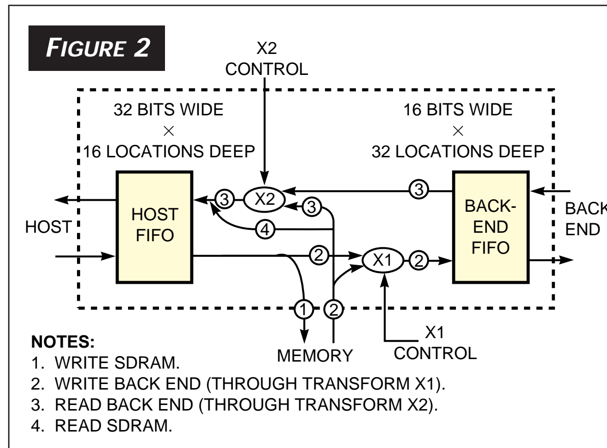
- Hardware-description languages (HDLs) bring numerous advantages to programmable-logic design, with a corresponding set of trade-offs.
- Synthesis results depend on the type of circuits you're designing, the device you're targeting, and the degree of software and silicon independence you want to maintain.
- Time you spend up-front learning HDL fundamentals will noticeably improve your subsequent design productivity.
- Modifying completed logic versus creating your own circuits for your design doesn't necessarily save time.
- A solid software background doesn't automatically ensure success when you first attempt to synthesize hardware.



The high-level block diagram shows major functional blocks and system interfaces.

writing needs. The only feature I missed, which other synthesis vendors' editors support, is color-coded VHDL keywords. This capability would have minimized the keystrokes and the typographical errors.

HDL editors also commonly provide generic entity and architecture templates, as well as preconstructed libraries of functions. Third-party HDL editors include HDL Turbo Writer from Saros Technology, ED-4W-HDL from Silicon System Solutions, and EALE/HDL from Translogic. The back-end place-and-route software in all cases was Xilinx's Alliance version 1.4.



Multiple internal datapaths test design-tool routing efficiency and bus-contention avoidance.

Because I'd already done ASIC-design work using WorkView Office and hoped that a brief refresher would get me up to speed, I started with the synthesis-based design. Ideally, I'd develop a solid VHDL foundation before complicating the picture by using VHDL to implement circuits for this project. However, like many of you, I didn't have the time to read a book cover to cover without distraction or attend a multiday off-site class.

Last summer, I began to occasionally review a few pages or a chapter from **Reference 1** when I had a spare few hours or to compile an example or two on the accompanying Cypress Semiconductor (www.cypress.com) Warp2 software. (I recommend both of these resources, along with **References 2 through 4**, for engineers learning VHDL.) Version 4.1 of Warp2Sim, so named because it includes a postfit waveform simulator, is especially useful because it covers both Cypress' CPLDs and now-defunct antifuse FPGAs. This support means that I could run the same source code through a PLD fitter and an FPGA place and router for comparison.

Last fall, I attended part of a seminar sponsored by Lattice Semiconductor (www.lattice.com) and taught by Mark Santoro of Santoro System Engineering (Encino, CA). I also attended QuickLogic's (www.quicklogic.com) session with Synplicity (www.synplicity.com) and beginner and intermediate VHDL sessions taught by Escalade, both at January's DesignCon '98. I've also heard great things about sessions by Esperan and TM Associates, and I encourage you to take one if you have the time. I also encourage you not to limit your VHDL training to elementary designs, such as two-input NAND gates and simple, clocked logic circuits with asynchronous and synchronous resets, as I did.

Fortunately, research and review of previous programmable-logic articles has made me aware that coding style has a measurable impact on final results, so this knowledge found its way into my design from the beginning (**References 5 through 8**). My final step before starting was to use the Quick Tour and tutorial that comes with FPGA Express. Although both the Quick Tour and tutorial improved my user interface and overall design-flow understanding, the prewritten source files meant that my overall VHDL expertise didn't grow as a result.

Stephen Wasson lives and works just south of Silicon Valley, whereas I'm roughly three hours north in Sacramento. My preferred work schedule starts at around 5 am and, most days, wraps up by 6 pm or so. Stephen, on the other hand, follows a more traditional engineering regimen, beginning in the late morning or early afternoon hours and continuing till 3 or 4 the next morning. As a result, Stephen and I met face to face only once or twice through the course of the project and spoke on the phone only a few more times. Most of our interaction was over e-mail, which had both its good and its bad points. Design teams that span multiple geographical locations and time zones can learn a lesson from our experience.

E-mail positives included the ability to polish the wording of questions and answers before sending them on and to combine multiple comments into one communication instead of relying on a dozen or so phone calls each day. E-mail also bridged our work schedules; for example, I could send a message to him at the end of my day and would find a reply waiting for me (often composed past midnight) when I stumbled into the office the next morning. E-mail's conciseness sometimes caused confusion and delay, however. I particularly remember one multiday, multiple-e-mail debate about which synchronous-DRAM (SDRAM) density and architecture we should target, which ended only when we realized we were advocating the same part but just using different terminology to describe it.

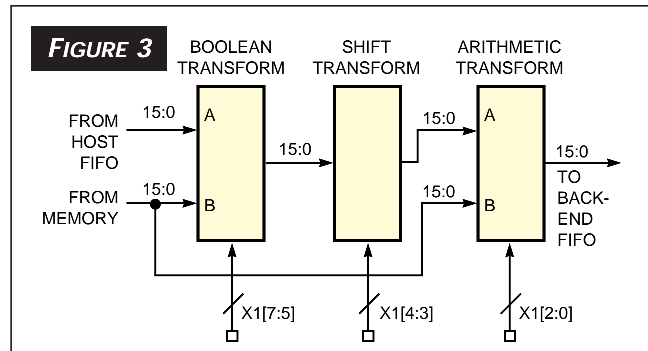
Stephen wanted to design a multi-FPGA neural network "fabric," or parallel-processing scheme. Although the idea is technically intriguing, I was unsure about how much practical application the architecture would have for *EDN*'s readers. As I continued negotiating with software vendors, he went back to the drawing board and in early December 1997 delivered a Revision 0.3 specification for a data-transform controller design that includes two system buses, a DRAM controller, and various internal arithmetic functions. Feedback and negotiation resulted in an almost-complete Revision 0.4 in early January.

Design details

The FIFO buffer on either end of our design optimally would contain both memory and logic (**Figures 1 and 2**). The design is also notable for its integrated DRAM controller, its large number of datapaths, and the two transform blocks (**Figures 3, 4, and 5**). If the design tools took advantage of the FPGA's FlexRAM and the natural data-

flow direction that the carry chains and other architecture elements encourage, we could fit everything into a XC-4013XL, Stephen estimated.

The design includes status, ID, and control registers, accessible via host-driven I/O reads and writes. Memory reads and writes can access the 64-Mbit SDRAM or the back-end bus. Back-end reads and writes always flow through the X2 and X1 transforms, respectively. By manipulating the transform-control inputs, you can make the design pass unchanged data from the back-end bus through the transforms, or you can modify the data with SDRAM contents and arithmetic and logical functions.



The X1 transform performs logical and arithmetic operations on data passing from the host bus to the back-end bus.

The host bus approximates but doesn't functionally match a 32-bit multiplexed PCI bus. The initiator and target both drive the open-collector-ready signal, a conceptual combination of PCI's initiator- and target-ready signals. Our design does not support burst suspend or early termination and does not use request or grant lines. The FPGA decodes the upper 3 address bits to determine host selection, allowing for as many as eight targets on one bus. SDRAM and back-end 32-bit data bursts have a fixed length of 16, which an address/control

SYNTHESIS STRENGTHS AND SHORTCOMINGS

These days, programmable-logic companies seem obsessed with hardware-description-language (HDL) synthesis. The vendors are running seminars, churning out press releases and glossy brochures, and giving presentations to get you to learn Verilog or VHDL—and quickly. Some of this promotion is self-serving, because they'd prefer that you buy their newest, biggest chips that could take forever to fill with logic if you used schematic-entry techniques. However, remember that PLDs and FPGAs are subject to the same Moore's Law dynamics as any other semiconductor device, which means that in the long term, the newest chips will probably be the cheapest per gate.

HDLs have other strengths. Like C++ and other object-oriented software languages, they lend themselves nicely to modular-design approaches, with logic segmented in a number of interconnected sections. You can even simulate and debug each module as a distinct entity before wiring them all together. Conceptually at least, this partitioning also allows you to reuse your or another's already-proven circuits in your next project. However, you can do much the same thing with multilevel schematics, although you must use the same vendor's tool suite each time. Modifying a module can sometimes take as long as or longer than developing a new one from scratch.

Taking the comparison a step further, the debate between schematic and synthesis advocates reminds me of the old assembly-versus-high-level-software-language battles. Synthesis generally produces less efficient results than do schematics. However, balance this factor against HDL's time-to-completion and time-to-modify advantages, and realize that no hard and fast rules exist regarding relative efficiency and performance. Synthesis products from different companies produce varying results based on what types of logic the companies are working on and what silicon platform you're targeting. Just as improving CPU performance masks high-level software inefficiencies, larger, cheaper, and more routing-rich programmable-logic chips probably will reduce future high-level-synthesis-efficiency concerns.

One of Verilog and VHDL's key advantages over schematics is HDLs' device portability, but here too the reality is sometimes less compelling than the hype. You can write silicon architecture- and vendor-independent HDL code and even port your design to an ASIC, something that Technical Editor Jim Lipman plans to tackle for his Oct 22, 1998, Hands-On Project. The output, however, can have worse efficiency and performance results than a less portable HDL design that instantiates vendor-specific functions.

Equally significant, but less commonly discussed, is synthesis-vendor portability. If you use functions and subroutines from a non-industry-standard library, you can't run the code on another vendor's synthesis tools without modification. Other synthesis problems are more practical. Odds are, you're approaching the point at which schematic designs are becoming unmanageable, but your company's unwilling to allocate the necessary time and money to enable you to properly develop HDL-simulation and -synthesis expertise.

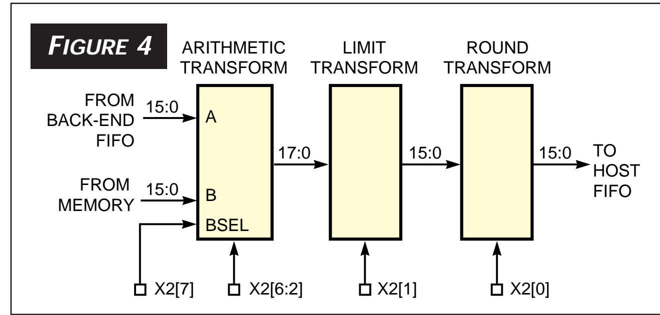
The uniqueness of various programmable-logic architectures and the subsequent reliance on vendor-developed schematic libraries have caused a slower migration of designers to synthesis compared with past migrations of "sea-of-NAND-gate" ASICs. To synthesis vendors, programmable-logic and ASIC designers are as different as night and day, with programmable-logic designers representing many more potential customers but averaging much smaller tool budgets. Software prices *have* fallen significantly from a few years ago, whereas tool capabilities have risen. However, an HDL-synthesis and -simulation package using a few back-end tool suites is still an expensive proposition for many companies, especially if they want to continue to use both schematics and synthesis for portions of the design.

cycle precedes. The target has as many as 16, originally eight, host clocks to respond to an initiator read request with first data. All I/O- and memory-read transactions include a single-clock turnaround cycle between an address and the first data.

The DRAM controller supports 64-Mbit, self-refresh SDRAM with a 16-bit data bus and four-bank internal array. The design uses explicit bank precharging at the end of each 32-access-long read or write burst, which also terminates the SDRAM internal address burst counter. Automatic FPGA-driven writes on power-up and reset configure the SDRAM for column-access-strobe-to-read data latencies of two clocks and full-page burst length.

The back-end bus provides separate 20-bit address and 16-bit data buses. The FPGA specifies the desired operation by driv-ing the ZRD# or ZRW# outputs along with the address and data transfers. Data bursts have a fixed length of 32 accesses. Al-though the design specification describes an optional asynchronous test port to manipulate data within the FPGA, such as the output of each stage of the X1 and X2 transforms, neither Stephen nor I have as yet implemented this feature. However, we provide four control register outputs suitable for monitoring with an oscilloscope or logic analyzer, or to drive debug LEDs.

A 17-bit host-configurable length counter decrements for each host-bus burst cycle. The SDRAM address counter increments for each SDRAM read or write burst cycle, including back-end reads and writes, which use SDRAM data in the transforms. The back-end address counter increments for each back-end read or write cycle.



The X2 transform manipulates data in transit from the back-end to the host bus.

Although we could have alternatively implemented the FIFO as several-stage pipelines or made them narrower, we stuck with the original 16×32- and 32×16-bit definitions. This depth allows for flexibility in operating frequency for the logic-core, host, and back-end buses. It also lets you and us add host-bus hold-off and pipelining capabilities in the future.

Although I had intended to immediately begin the design, my execution of this plan was not so good. Other work obligations, plus unanticipated business travel and a few ill-timed computer crashes (unrelated to the FPGA-design software) distracted me. As a result, I wrote my first line of VHDL on

CURRENT STATUS, FUTURE PLANS

So far in this project, I've finished and commented all the design modules and run them through FPGA Express to error-check the syntax. I also briefly simulated them in Viewlogic's SpeedWave and interconnected them with a top-level structural-VHDL entity and architecture. In completing these steps, I've encountered two more of the common VHDL stumbling blocks: the fact that design-module interconnection and state-machine design are both visually oriented tasks and the problem of type definition.

The visual-orientation problem makes it difficult except in the simplest cases to directly create design-module interconnections and state-machine designs in a text-based language. I did lots of sketches on paper before committing my design to VHDL. A graphical tool, such as Translogic's Ease/HDL, would have been useful for automatically generating the necessary VHDL code in these cases.

I also struggled with type definitions. I might first declare a signal as an *OUT* and then realize that I also needed to access it within the entity, prompting a shift to type *BUFFER*. However, I'd then forget to also define interconnecting signals in other entities as *BUFFER*, resulting in error messages. Although I could have avoided this problem by declaring these signals as type *INOUT*, **Reference 12** discourages this technique because it reduces code readability, preferring instead to reserve *INOUT* for true bidirectional signals.

For these reasons and others, some engineers advocate a blend of HDL and schematic techniques for programmable-logic designs. Although the need for this two-phase approach probably decreases with increased HDL expertise, I can't disagree with the intent of their suggestion. Stephen Wasson, my partner on this project, believes that schematics are best for data-flow circuits, and synthesis is preferable for data control and thereafter for the testbench and simulation.

I still face much additional work before I can call this project "done." First, I have to map the logic to an XC4000XL device and run the netlist through the back-end tools. Stephen has to finish his schematic design, and we'll then run Viewlogic Fusion simulation vectors through both to ensure functional compatibility. HighGate Design still plans to complete a VHDL design, but I'm unsure if I can finish my schematic version in time to meet the publishing deadlines for Part 2 of this project. Alternatively, we may create a second version of his design, eliminating the timing and layout constraints and removing extra circuitry, such as X1 and X2 transform pipelines.

You can find the documentation as well as source files for both my and HighGate Design's schematic and synthesis implementations on *EDN*'s Web site, www.ednmag.com. Check back often, because we'll upload new information as we complete it. I encourage you to download our files, modify them to fit your needs, and use them to benchmark programmable-logic devices and design software for your future projects.

FPGA DESIGN WITH VHDL

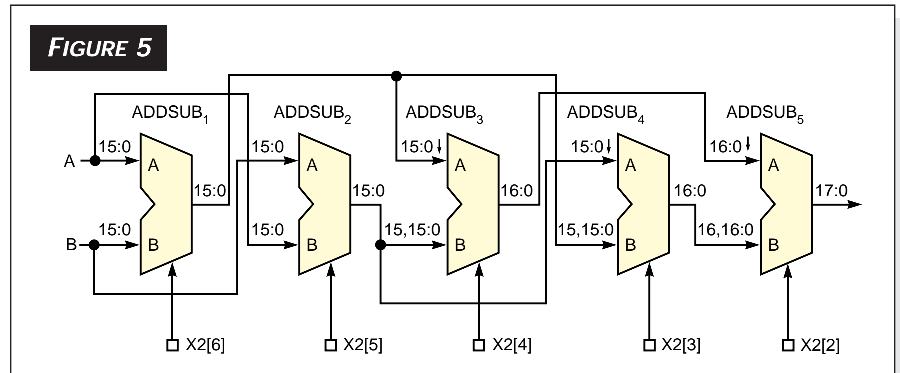
Sunday afternoon, February 22. The following discussion assumes that you have some familiarity with VHDL command syntax and entity, architecture, and component structures. If not, you might want to begin by examining any of the materials listed in **References 1** through **4** and **8**.

Two engineers: two approaches

From the beginning of the design, I strove to test one of the key selling points of synthesis: vendor independence, which means not using libraries of parameterizable modules (LPMs), direct instantiation, or EDA vendor-proprietary routines. Stephen, however, would instantiate whenever possible in his schematic and synthesis designs. He also planned to insert timing and layout constraints to guide compilation and the back-end tools, neither of which I intended to do. If necessary, he would predefine the pinout to guide the data flow through the part, whereas I let the tools pick the pins they thought best. My approach would also theoretically let a designer migrate my design to an ASIC without significant rewrites. Unintentionally, however, I broke my vendor-independency vow.

I declared all signals of type *std_logic* and *std_logic_vector*, whose possible states extend beyond '1' and '0' to include simulation-friendly 'Z,' 'X,' 'U,' and other values. This approach also comes in handy when you create tristates and bidirectional buffers. Direct manipulation of *std_logic* signals worked fine for logical functions, such as ANDs and ORs, shifts, and rotates. However, FPGA Express flagged as an error every attempt to add or subtract a number and signal or two signals. VHDL is a strongly typed language. Indiscriminately mixing signals or variables of different type or length causes error messages, which is both good and bad. Type enforcement keeps you from making mistakes that would be difficult to catch and debug during simulation, but this enforcement can be frustrating when you know what you want to do and are looking for a syntax that the synthesis tool accepts.

I was already declaring IEEE libraries *ieee.std_logic_1164.ALL* and *ieee.std_logic_arith.ALL* in my VHDL files. Synopsys recommended that, when necessary, I also include *ieee.std_logic_signed.ALL* or *ieee.std_logic_unsigned.ALL*. I could use either for most arithmetic functions but had to be careful to use the right one for comparisons (**Listing 1**).



By controlling the polarity of the X2 transform-control signals, you can pass back-end data through unchanged or arithmetically combine it with SDRAM contents.

Assuming that *data_ina* has value "10000000" and *data_inb* has value "00000000" when a rising-edge clock transition causes the process to run, *data_out* would be '0' after the process conclusion. If, however, you changed the line

USE ieee.std_logic_unsigned.ALL;

to

USE ieee.std_logic_signed.ALL;,

you'd end up with an opposite result.

Using the signed and unsigned libraries solved the mixed-logic and arithmetic-function problems, but I later found that their "IEEE" label was deceptive. Synopsys developed the *ieee.std_logic_unsigned* and *ieee.std_logic_signed* libraries and put them into the IEEE directory. Synopsys assured me that simulation vendors support these libraries, and other synthesis companies have developed their own versions, although the files may not exist in the same place in the directory structure or have the same name. I'll revisit this topic after I've had a chance to run my code through multiple vendors' synthesis software and will report my findings.

How could I have avoided this type-incompatibility problem? According to David Pellerin from Accolade Design Automation (www.acc-eda.com), an alternative technique uses the VHDL-93 *ieee.numeric_std* library. With this approach, I'd convert the *std_logic* and *std_logic_vector* signals or variables to type *signed* or *unsigned* for arithmetic functions and then back to *std_logic* and *std_logic_vector* before performing additional logical operations or passing them to other entities. FPGA Express, however, doesn't support VHDL-93 syntax, which is not a big problem in my design. Although commands such as *XNOR*, *SLL* (shift left logical), and *SRL* (shift right logical) would have been nice to have, I developed workarounds without much effort.

All my research before coding indicated that HDLs are conceptually similar to high-level software languages. Because I consider myself a decent C programmer, I felt reasonably confident that developing VHDL expertise meant just learning another language syntax. In retrospect, I see that this attitude was somewhat naive, and I'm not sure whether software experience helps or hinders an HDL novice. The simple explanation of the difference is that software executes sequentially, whereas most HDL statements, which create hardware, execute concurrently.

All concurrent statements between the *BEGIN* and *END* commands of all architectures in a design, as well as all processes initiated by transitions of signals in their sensitivity lists, evaluate in parallel. This fact also means that the order of statements and processes within the file is irrelevant. Within a process, statements execute sequentially, but signals don't update until after the process terminates. As long as I kept

these facts in mind, I was OK. However, because the command syntax was similar to C or Pascal (*IF-THEN-ELSE*, *CASE*, and *FOR-LOOP* structures, etc), I easily fell back into a software mentality. This attitude can cause typos, such as missing semicolons and *ELSE IF* instead of the intended *ELSIF*. It can also result in incorrect logic functions, as **Listing 2**, which I modified from an example in **Reference 1**, demonstrates.

Looking at this function with software-biased eyes, you might think that the process runs each time a bit in the 8-bit *a_bus* variable changes. You'd be right. You might also think that the process terminates with x='1' if *a_bus*="11111111." Here, unfortunately, you'd be wrong. Remember that VHDL updates all signals only at the conclusion of the process. Therefore, the statement x≤'1' has no influence on the first iteration of *x≤a_bus(i) AND x;* within the loop. Likewise, any evaluation of *x≤a_bus(i) AND x;* also affects the next loop iteration. Assuming that x initializes to '0' on power-up and reset, x=0 at the process conclusion, regardless of what value of *a_bus* causes the process to run. **Listing 3** shows one possible fix.

Both **Listings 2 and 3** use a variable, *i*, for looping, whereas **Listing 3** adds variable *tmp*. **Listing 3** assigns a value to signal *x* only at the end of the process. Variables differ from signals in that the simulator—and, therefore, the synthesis compiler—immediately updates variables. However, **Reference 1** cautions against indiscriminate use of variables because of nonstandard support among simulator and synthesis tools, so I avoided them whenever possible. A good rule of thumb: In any process, the last statement that assigns a value to a signal is the one that takes precedence. You can also selectively make assignments using *IF-THEN* and *CASE* statements.

Coding style influences results

Placing multiple design entities in separate source files is another habit of software engineers who are used to writing modular code and numerous subroutines. A modular, hierarchical approach provides many benefits over one large design entity: easier reuse of portions of the architecture in future designs; step-by-step simulation as you develop each module; and an simpler-to-understand documentation style, which is analogous to multiple chapters in a book. In our case, this technique will also let us test vendors' synthesis tools not only on the entire design, but also on circuits within it. However, modularity has one significant downside: Because the tools sequentially compile each module and merge the netlists afterward, these tools cannot identify resource-sharing opportunities that span multiple modules.

IF-THEN versus *CASE* usage can impact the efficiency and performance of the resulting logic. *IF-THEN* statements create priority-evaluated logic (**Listing 4**), whereas *CASE* statements result in logic that judges all defined conditions in parallel. Each XC4000XL look-up table has four inputs. The logic that **Listing 5** creates fits within one look-up table, producing a version of the desired logic function that is more efficient and faster than the multi-look-up-table equivalent in **Listing 4**.

Synthesis attempts to fit additional logic functions into the available portions of partially used look-up tables, but success depends on your design and isn't guaranteed. Notice too that **Listings 4 and 5** define an outcome for all possible input combinations. This technique has two benefits. First, if you don't define all possible input conditions, synthesis infers and inserts a latch to hold signals at previous values when these undefined conditions occur. This result is acceptable if an inferred latch is what you intend, but otherwise you end up at least with unnecessary logic or, worse, a circuit that functions incorrectly. Second, because I used the *std_logic* type, my *IF-THEN* and *CASE* statements cover 'X,'

FOR MORE INFORMATION...

For information on products such as those discussed in this article, circle the appropriate numbers on the Information Retrieval Service card or use *EDN*'s Express Request service. When you contact any of the following manufacturers directly, please let them know you read about their products in *EDN*.

Esperan Ltd
Ramsbury, Wilts, UK
+44 0 1672 520101
fax +44 0 1672 521039
www.esperan.com
Circle No. 385

HighGate Design Inc
Saratoga, CA, USA
+1-408-255-7160
fax +1-408-255-7162
www.highgatedesign.com
Circle No. 386

Saros Technology
Stevenage, Herts, UK
+44 01438-746433
fax +44-01438-310093
www.saros.co.uk
Circle No. 387

Silicon System Solutions
Canterbury, Victoria, Australia
+61-3-9888-4774
fax +61-3-9888-4224
www.silicon-systems.com
Circle No. 388

Synopsys Inc
Munich, Germany
+49 89 99 320 130
fax +49 89 99 320 132
www.synopsys.com
Circle No. 389

TM Associates
Oregon City, OR, USA
+1-503-656-4457
fax +1-503-656-4475
www.europa.com/~twille
Circle No. 390

Translogic USA Corp
Enschede, The Netherlands
+31-318-642076
www.translogiccorp.com
Circle No. 391

Viewlogic Systems Inc
Berkshire, UK
+44 1344 303737
fax +44 1344 304747
www.viewlogic.com
Circle No. 392

Xilinx Inc
San Jose, CA, USA
+1-408-559-7778
fax +1-408-879-4780
www.xilinx.com
Circle No. 393

VOTE . . .
Please also use the Information Retrieval Service card to rate this article (circle one):
High Interest 582
Medium Interest 583
Low Interest 584

Super Circle Number
For more information on the products available from all of the vendors listed in this box, circle one number on the reader service card.
Circle No. 394

'Z,' and other signal and variable states.

Even the manner in which you use parentheses can impact the results. According to Xilinx's documentation, many synthesis tools implement the expression *A+B+C+D* as three series adders. By including two sets of parentheses, resulting in the expression *(A+B)+(C+D)*, you end up with two parallel adders feeding a third adder, a higher performance configuration.

Standing on others' shoulders

As I dug deeper into the design, I often used **Reference 1** and my seminar notes. Instead of creating my own FIFOs from scratch, I quickly modified an example in **Reference 1**. This task involved adding internal logic that simplified the external interface to read, write, clock, and reset inputs and adding empty and full flags. Modifications were straightforward, and I completed them in just over an hour. The host-bus FIFOs were a challenge, because I was interfacing between 32- and 16-bit ports, but resolving this issue was relatively easy. I also came across several other useful resources (**References 9** through **11**).

My experiences with the other "core" went less smoothly. Christian Green from MoSys (www.mosys.com) had developed a simple synchronous-graphics-RAM (SGRAM) controller (**Reference 12**), and I thought that migrating it to my SDRAM design would be simple. Wrong. My difficulty had little to do with the design itself, which was commented and made efficient use of silicon. However, I had to tear apart and redo more of the states and state transitions than I had anticipated.

Reference 12's design targeted 8-Mbit, low-latency SGRAM, with a burst length of four accesses and autoprecharge. Mine, on the other hand, interfaced with 64-Mbit standard SDRAM, with a full page burst terminated after 32 accesses and no autoprecharge. The original design also used a modified Mealy state-machine design technique, which was a challenge to decipher. I also blame the fact that I dived into the conversion with an incomplete understanding of either state-machine coding in VHDL or SDRAM operating details.

After a week of work, averaging three hours per day on this project, I'd created the VHDL code for four unidirectional FIFOs, both transform-logic blocks, and the SDRAM controller. The transform functions were extremely fast and simple to code, and I imagined that both creating and modifying them in VHDL was much easier than what my partner was experiencing with his schematic designs. I'd also sketched out the remainder of the state machines, driven by host-bus input transitions. At this point, I decided to step back and see how well I was addressing chip- and system-level concerns. I'd moved ahead of Stephen's progress, and I encountered some minor spec discrepancies and missing details that we jointly needed to resolve before it made sense to proceed. I found several problems.

Performance, size issues

When I began modifying **Reference 12's** DRAM controller, I remembered that SDRAM access requests and in-progress row-refresh operations could occasionally collide. Because refresh takes priority, the result would be a delay in the first read- or write-data transfer. Adding up

LISTING 1—COMPARE ROUTINE USING UNSIGNED LIBRARY

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY synccompare_e IS PORT (
       clk:         IN std_logic;
       data_ina:    IN std_logic_vector(7 DOWNTO 0);
       data_inb:    IN std_logic_vector(7 DOWNTO 0);
       data_out:    OUT std_logic);
END synccompare_e;
ARCHITECTURE synccompare OF synccompare_e IS
BEGIN
       PROCESS (clk)
       BEGIN
              IF clk'EVENT AND clk = '1' THEN
                     IF data_ina < data_inb THEN
                            data_out <= '1';
                     ELSE
                            data_out <= '0';
                     END IF;
              END IF;
       END PROCESS;
END synccompare;
```

LISTING 2—INCORRECT AND ROUTINE CODING

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY my_and IS PORT(
       a_bus:       IN std_logic_vector (7 DOWNTO 0);
       x:     BUFFER std_logic);
END my_and;
ARCHITECTURE wont_work OF my_and IS
BEGIN
anding:
       PROCESS (a_bus)
       BEGIN
              x <= '1';
              for i IN 7 DOWNTO 0 LOOP
                     x <= a_bus(i) AND x;
              END LOOP;
       END PROCESS;
END wont_work;
```

LISTING 3—CORRECT AND ROUTINE CODING

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY my_and IS PORT(
       a_bus:       IN std_logic_vector (7 DOWNTO 0);
       x:     OUT std_logic);
END my_and;
ARCHITECTURE will_work OF my_and IS
BEGIN
anding:
       PROCESS (a_bus)
       VARIABLE temp:     std_logic;
       BEGIN
              tmp:= '1';
              for i IN 7 DOWNTO 0 LOOP
                     tmp := a_bus(i) AND tmp;
              END LOOP;
       x <= tmp;
       END PROCESS;
END will_work;
```

clock cycles, I realized that my design would be unable to satisfy the original maximum eight-clock delay from host-read request to first-data output. Then, Stephen re-minded me that, although the host bus used 32-bit data, everything else was 16 bits. The answer to the perform-ance problem was to run most of the chip at twice the clock frequency of the host-bus logic.

If the XC4000XL had internal PLLs, I could have doubled an external host-bus clock. I could also halve a fast input clock with a T-flip-flop circuit, but then the design might be out of phase with the half-frequency clock that other host-bus devices created and use. Driving this derived clock *to* the other peripherals would solve the phase problem but cause unwanted timing skew. The con-cept of a target device's creating the host-bus clock also seemed awkward. The technique I chose instead employed two clock inputs from an external precision PLL. The inputs selectively drive por-tions of the FPGA.

I defined the FIFO arrays follow-ing my "no-instantiation" rule (**Listing 6**). I created four 16-bit-wide, 32-element-deep, unidirec-tional, dual-ported FIFO buffers: two for the host bus and two for the back-end bus. I used separate logic to handle the 32- to 16-bit host-bus translation. However, I had a hunch that FPGA Express might not deci-pher that I was creating a FIFO buffer and use FlexRAM (Xilinx's look-up-table memo-ry), instead of flip-flops, for FIFO stor-age elements. A quick e-mail to Synop-sys confirmed my suspicion.

Out of fairness to Synopsys, I'm not sure whether any other synthesis prod-uct can implement generic array defin-itions into RAM, either. Creating FIFO arrays from look-up tables or flip-flops means employing different logic usage (**Table 1**). The **table** omits the logic around the array and any logic blocks for signal routing. Unlike Xilinx's XC4000XL family, other PLDs and ASICs that provide no onboard RAM usable as embedded memory *must* use flip-flops to create FIFO elements.

I had to do something to make the design smaller. One option would be to reduce the FIFO depth, but this approach would limit future design flexibility. I also could have used sepa-rate FIFO buffers outside the FPGA. However, I'd probably also have to partition the host, back-end, and other design logic into multiple PLDs with un-known cost, power, perform-ance, and board-space impacts. The design specification allowed for no simultaneous reads and writes for either the host or the back-end bus. So, I next con-verted the four 16-bit×32 unidi-rectional FIFO buffers into two bidirectional alternatives—one for the host bus and the other for the back-end bus. This step partially solved the array-size problem, but a few issues remained.

The back-end FIFO buffer had separate *clka* and *clkb* inputs, cor-responding to the two data I/O ports. However, FPGA Express warned that I had defined a single FIFO-array-write process with both clocks in the sensitivity list. Breaking the FIFO writes into two processes solved this problem but synthesized a circuit with two 512-element arrays, one for each clock. Equally disturbing, FPGA Express' warning list reported that the outputs of each *clka* and *clkb* array flip-flop pair were dri-ving a common node. Redesign-ing the back-end FIFO buffer for one clock input and write process got me to the desired 512 ele-ments and eliminated the inter-nal short circuits. Because the host FIFO buffer's two ports need-ed to run at different frequencies for perform-ance rea-sons, I employed two 512-element unidirec-tional FIFO buffers in this case.

Other improvements for both the back-end and host included explicitly initializing array flip-flops on reset, which allowed FPGA Express to use the XC-4000XL global reset network. I could have also multiplexed both ports for each FIFO buffer onto a common set of I/O signals, which would have reduced the routing requirements. However, for perfor-mance reasons, I wanted to allow the state machines feeding data into one end of the FIFO buffer and out the other to run in parallel. The lesson? When

LISTING 4—IF-THEN ROUTINE

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY ifthenex_e IS PORT(
      a:    IN std_logic_vector (3 DOWNTO 0);
      x:    OUT std_logic);
END ifthenex_e;
ARCHITECTURE ifthenex OF ifthenex_e IS
BEGIN
      PROCESS (a)
      BEGIN
            IF a = "0001" THEN
                  x <= '1';
            ELSIF a = "0010" THEN
                  x <= '1';
            ELSIF a = "0100" THEN
                  x <= '1';
            ELSE
                  x <= '0';
      END PROCESS;
END ifthenex;
```

LISTING 5—CASE ROUTINE

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY caseex_e IS PORT(
      a:    IN std_logic_vector (3 DOWNTO 0);
      x:    OUT std_logic);
END caseex_e;
ARCHITECTURE caseex OF caseex_e IS
BEGIN
      PROCESS (a)
      BEGIN
            CASE a IS
                  WHEN "0001" =>
                        x <= '1';
                  WHEN "0010" =>
                        x <= '1';
                  WHEN "0100" =>
                        x <= '1';
                  WHEN OTHERS =>
                        x <= '0';
            END CASE;
      END PROCESS;
END caseex;
```

LISTING 6—IF-THEN AND CASE LOGIC IMPLEMENTATIONS

```
TYPE fifo_array IS ARRAY(31 DOWNTO 0) OF std_logic_vector(15 DOWNTO 0);
SIGNAL fifo:              fifo_array;
```

TABLE 1—DEVICE USAGE COMPARISON: LOOK-UP-TABLE- VERSUS FLIP-FLOP-BASED FIFO BUFFERS

FIFO size (bits)	No. of FIFOs	No. of FIFO array elements	XC4000XL series four-input look-up-table size (number of elements)	No. of four-input look-up tables per logic block	No. of four-input look-up-table elements per logic block	No. of flip-flops per logic block	No. of logic blocks: FIFO array from look-up tables	No. of logic blocks: FIFO array from flip-flops
16×32 or 32×16	One	512	16	Two	32	Two	16	256
16×32 or 32×16	Two	1024	16	Two	32	Two	32	512
16×32 or 32×16	Three	1536	16	Two	32	Two	48	768
16×32 or 32×16	Four	2048	16	Two	32	Two	64	1024

your design can take advantage of available embedded RAM, the amount of work to retarget the HDL code to a different FPGA or an ASIC may be an acceptable trade-off for significantly im-proved silicon efficiency.

The reference design implements considerable data movement throughout the FPGA with the resulting potential for bus contention (**Figure 2**). For example, the host FIFO drives the SDRAM's data buffers during writes, and SDRAM reads can drive the X1 and X2 transform inputs as well as contend with the X2 transform output for the host FIFO buffer. I first controlled which source would drive a node using internal tristate buffers. However, rereading various VHDL and programmable-logic literature, I realized that unless I closely controlled the internal timing and logic placement, I might easily end up with transitional bus contention, increasing power consumption and reducing device reliability.

The alternative strategy, which I implemented wherever possible, used multiplexers. This approach consumes additional logic and routing resources. However, my chosen strategy also meant I wouldn't have to create layout and timing constraints to ensure that two data sources weren't simultaneously driving a common internal node. The performance impacts of my multiplexer choice were unclear; I'd heard conflicting data in the past from engineers about whether internal tristates resulted in faster or slower designs.

One performance improvement that I decided to omit involves placing one or several pipeline-register sets within the X1 and X2 transforms. Although this approach would increase first-data latency through the transforms, it might also boost the maximum operating frequency. However, this technique would also increase the on-chip logic and complicate the state machines. Stephen plans to investigate adding pipelines to his designs, and I'll be interested to see whether they significantly impact performance or size.

References

1. Skahill, Kevin, *VHDL for Programmable Logic*, Addison-Wesley Publishing, 1996, ISBN 0-201-89586-2.

2. Pellerin, David and Douglas Taylor, *VHDL made easy!*, Prentice-Hall, 1997, ISBN 0-13-650763-8.

3. Smith, Douglas J, *HDL Chip Design*, Doone Publications, 1996, ISBN 0-9651934-3-8.

4. "Putting the design back in HDL design," QuickLogic and Synplicity, 1998 DesignCon, On-Chip System Design Conference.

5. Dipert, Brian, "Moving beyond programmable logic: if, when, how?," *EDN*, Nov 20, 1997, pg 77.

6. Dipert, Brian, "Programmable logic: Beat the heat on power consumption," *EDN*, Aug 1, 1997, pg 57.

7. Dipert, Brian, "Shattering the programmable-logic speed barrier," *EDN*, May 22, 1997, pg 36.

8. Conner, Doug, "Making the jump to HDL-based programmable-logic design," *EDN*, Sept 1, 1997, pg 181.

9. Steve Knapp's "Programmable Logic Jump Station," www.optimagic.com.

10. "VHDL Reference Manual," Synopsys.

11. "HDL synthesis for FPGAs," Xilinx.

12. Green, Christian, "Analyzing and implementing SDRAM and SGRAM controllers," *EDN*, Feb 2, 1998, pg 155.

Acknowledgments

It's been a pleasure working with Stephen Wasson from HighGate Design on this project; the reference-design concept and specification he developed are comprehensive in breadth and depth. I'd also like to acknowledge the technical assistance of David Pellerin of Accolade Design Automation, Kevin Skahill of Cypress Semiconductor, Christian Green of MoSys, Ron Plyer and Ramine Roane of Synopsys, Tom Barber of Viewlogic, and Loren Lacy of Xilinx.

You can reach Technical Editor Brian Dipert at 1-916-454-5242, fax 1-916-454-5101, e-mail edndipert@worldnet.att.net, http://members.aol.com/bdipert.

VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest	Medium Interest	Low Interest
582	583	584