Schematic-based flows grind to a halt

Toward the end of the 1980s, as designs grew in size and complexity, schematic-based ASIC flows began to run out of steam. Visualizing, capturing, debugging, understanding, and maintaining a design at the gate level of abstraction became increasingly difficult and inefficient when juggling 5,000 or more gates and reams of schematic pages.

In addition to the fact that capturing a large design at the gate level of abstraction is prone to error, it is also extremely time-consuming. Thus, some EDA vendors started to develop design tools and flows based on the use of *hardware description languages*, or HDLs.

The advent of HDL-based flows

The idea behind a hardware description language is, perhaps not surprisingly, that you can use it to describe hardware. In a wider context, the term *hardware* is used to refer to any of the physical portions of an electronics system, including the ICs, printed circuit boards, cabinets, cables, and even the nuts and bolts holding the system together. In the context of an HDL, however, “hardware” refers only to the electronic portions (components and wires) of ICs and printed circuit boards. (The HDL may also be used to provide limited representations of the cables and connectors linking circuit boards together.)

In the early days of electronics, almost anyone who created an EDA tool created his or her own HDL to go with it. Some of these were analog HDLs in that they were intended to rep-
resent circuits in the analog domain, while others were focused on representing digital functionality. For the purposes of this book, we are interested in HDLs only in the context of designing digital ICs in the form of ASICs and FPGAs.

**Different levels of abstraction**

Some of the more popular digital HDLs are introduced later in this chapter. For the nonce, however, let’s focus more on how a generic digital HDL is used as part of a design flow. The first thing to note is that the functionality of a digital circuit can be represented at different levels of abstraction and that different HDLs support these levels of abstraction to a greater or lesser extent (figure 9.1).

![Figure 9.1. Different levels of abstraction.](image)

The lowest level of abstraction for a digital HDL would be the **switch level**, which refers to the ability to describe the circuit as a netlist of transistor switches. A slightly higher level of abstraction would be the **gate level**, which refers to the ability to describe the circuit as a netlist of primitive logic gates and functions. Thus, the early gate-level netlist formats gener-
ated by schematic capture packages as discussed in the previous chapter were in fact rudimentary HDLs.

Both switch-level and gate-level netlists may be classed as structural representations. It should be noted, however, that "structural" can have different connotations because it may also be used to refer to a hierarchical block-level netlist in which each block may have its contents specified using any of the levels of abstraction shown in Figure 9-1.

The next level of HDL sophistication is the ability to support functional representations, which covers a range of constructs. At the lower end is the capability to describe a function using Boolean equations. For example, assuming that we had already declared a set of signals called Y, SELECT, DATA-A, and DATA-B, we could capture the functionality of a simple 2:1 multiplexer using the following Boolean equation:

\[ Y = (\text{SELECT} \& \text{DATA-A}) \mid (!\text{SELECT} \& \text{DATA-B}); \]

Note that this is a generic syntax that does not favor any particular HDL and is used only for the purposes of this example. (As we discussed in chapter 3, the "&" character represents a logical AND, the "!" character represents an OR, and the "\mid" character represents a NOT.)

The functional level of abstraction also encompasses register transfer level (RTL) representations. The term RTL covers a multitude of manifestations, but the easiest way to wrap one's brain around the underlying concept is to consider a design formed from a collection of registers linked by combinational logic. These registers are often controlled by a common clock signal, so assuming that we have already declared two signals called CLOCK and CONTROL, along with a set of registers called REGA, REGB, REGC, and REGD, then an RTL-type statement might look something like the following:
when CLOCK rises
  if CONTROL == "1"
    then REGA = REGB & REGC;
  else REGA = REGB | REGD;
  end if;
end when;

In this case, symbols like when, rises, if, then, else, and the like are keywords whose semantics are defined by the owners of the HDL. Once again, this is a generic syntax that does not favor any particular HDL and is used only for the purposes of this example.

The highest level of abstraction supported by traditional HDLs is known as behavioral, which refers to the ability to describe the behavior of a circuit using abstract constructs like loops and processes. This also encompasses using algorithmic elements like adders and multipliers in equations; for example:

\[ Y = (\text{DATA-A} + \text{DATA-B}) \times \text{DATA-C} \]

We should note that there is also a system level of abstraction (not shown in figure 9-1) that features constructs intended for system-level design applications, but we'll worry about this level a little later.

Many of the early digital HDLs supported only structural representations in the form of switch or gate-level netlists. Others such as ABEL, CUPL, and PALASM were used to capture the required functionality for PLD devices. These languages (which were introduced in chapter 3) supported different levels of functional abstraction, such as Boolean equations, text-based truth tables, and text-based finite state machine (FSM) descriptions.

The next generation of HDLs, which were predominantly targeted toward logic simulation, supported more sophisticated levels of abstraction such as RTL and some behavioral constructs. It was these HDLs that formed the core of the first true HDL-based design flows as discussed below.
A simple (early) HDL-based ASIC flow

The key feature of HDL-based ASIC design flows is their use of logic synthesis technology, which began to appear on the market around the mid-1980s. These tools could accept an RTL representation of a design along with a set of timing constraints. In this case, the timing constraints were presented in a side-file containing statements along the lines of "the maximum delay from input X to output Y should be no greater than N nanoseconds" (the actual format would be a little drier and more boring).

The logic synthesis application automatically converted the RTL representation into a mixture of registers and Boolean equations, performed a variety of minimizations and optimizations (including optimizing for area and timing), and then generated a gate-level netlist that would (or at least, should) meet the original timing constraints (Figure 9-2).

There were a number of advantages to this new type of flow. First of all, the productivity of the design engineers rose dramatically because it was a lot easier to specify, understand,
discuss, and debug the required functionality of the design at the RTL level of abstraction as opposed to working with reams of gate-level schematics. Also, logic simulators could run designs described in RTL much more quickly than their gate-level counterparts.

One slight glitch was that logic simulators could work with designs specified at high levels of abstraction that included behavioral constructs, but early synthesis tools could only accept functional representations up to the level of RTL. Thus, design engineers were obliged to work with a synthesizable subset of their HDL of choice.

Once the synthesis tool had generated a gate-level netlist, the flow became very similar to the schematic-based ASIC flows discussed in the previous chapter. The gate-level netlist could be simulated to ensure its functional validity, and it could also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist could then be used to drive the place-and-route software, following which a more accurate timing analysis could be performed using extracted resistance and linefeed capacitance values.

**A simple (early) HDL-based FPGA flow**

It took some time for HDL-based flows to flourish within the ASIC community. Meanwhile, design engineers were still coming to grips with the concept of FPGAs. Thus, it wasn’t until the very early 1990s that HDL-based flows featuring logic synthesis technology became fully available in the FPGA world (Figure 9-3).

As before, once the synthesis tool had generated a gate-level netlist, the flow became very similar to the schematic-based FPGA flows discussed in the previous chapter. The gate-level netlist could be simulated to ensure its functional validity, and it could also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist could then be used to drive the FPGA’s mapping, packing, and place-and-route software, following
which a more accurate timing report could be generated using real-world (physical) values.

**Architecturally aware FPGA flows**

The main problem besetting the original HDL-based FPGA flows was that their logic synthesis technologies were derived from the ASIC world. Thus, these tools “thought” in terms of primitive logic gates and registers. In turn, this meant that they output gate-level netlists, and it was left to the FPGA vendor to perform the mapping, packing, and place-and-route functions.

Sometime around 1994, synthesis tools were equipped with knowledge about different FPGA architectures. This meant that they could perform mapping—and some level of packing—functions internally and output a LUT/CLB-level netlist. This netlist would subsequently be passed to the FPGA vendor’s place-and-route software. The main advantage of this approach was that these synthesis tools had a better idea about timing estimations and area utilization, which allowed them to generate a better *quality of results* (QoR). In real terms, FPGA designs generated by architecturally aware synthesis tools were
1878: Ireland. Denis Redmond demonstrates capturing an image using selenium photocells.

15 to 20 percent faster than their counterparts created using traditional (gate-level) synthesis offerings.

**Logic versus physically aware synthesis**

We’re jumping a little bit ahead of ourselves here, but this is as good a place as any to briefly introduce this topic. The original logic synthesis tools were designed for use with the multimicron ASIC technologies of the mid-1980s. In these devices, the delays associated with the logic gates far outweighed the delays associated with the tracks connecting those gates together. In addition to being relatively small in terms of gate-count (by today’s standards), these designs featured relatively low clock frequencies and correspondingly loose design constraints. The combination of all of these factors meant that early logic synthesis tools could employ relatively simple algorithms to estimate the track delays, but that these estimations would be close enough to the real (post-place-and-route) values that the device would work.

Over the years, ASIC designs increased in size (number of gates) and complexity. At the same time, the dimensions of the structures on the silicon chip were shrinking with two important results:

- Delay effects became more complex in general.
- The delays associated with tracks began to outweigh the delays associated with gates.

By the mid-1990s, ASIC designs were orders of magnitude larger—and their delay effects were significantly more sophisticated—than those for which the original logic synthesis tools had been designed. The result was that the estimated delays used by the logic synthesis tool had little relation to the final post-place-and-route delays. In turn, this meant that achieving *timing closure* (tweaking the design to make it achieve its original performance goals) became increasingly difficult and time-consuming.
For this reason, ASIC flows started to see the use of physically aware synthesis somewhere around 1996. The ways in which physically aware synthesis performs its magic are discussed in more detail in chapter 19. For the moment, we need only note that, during the course of performing its machinations, the physically aware synthesis engine makes initial placement decisions for the logic gates and functions. Based on these placements, the tool can generate more accurate timing estimations.

Ultimately, the physically aware synthesis tool outputs a placed (but not routed) gate-level netlist. The ASIC’s physical implementation (place-and-route) tools use this initial placement information as a starting point from which to perform local (fine-grained) placement optimizations followed by detailed routing. The end result is that the estimated delays used by the physically aware synthesis application more closely correspond to the post-place-and-route delays. In turn, this means that achieving timing closure becomes a less taxing process.

“But what of FPGAs,” you cry. Well, these devices were also increasing in size and complexity throughout the 1990s. By the end of the millennium, FPGA designers were running into significant problems with regard to timing closure. Thus, around 2000, EDA vendors started to provide FPGA-centric, physically aware synthesis offerings that could output a mapped, packed, and placed LUT/CLB-level netlist. In this case, the FPGA’s physical implementation (place-and-route) tools use this initial placement information as a starting point from which to perform local (fine-grained) placement optimizations followed by detailed routing.

**Graphical design entry lives on**

When the first HDL-based flows appeared on the scene, many folks assumed that graphical design entry and visualization tools, such as schematic capture systems, were poised to exit the stage forever. Indeed, for some time, many design engineers prided themselves on using text editors like VI
1879: America Thomas Alva Edison invents an incandescent light bulb (a year after Sir Joseph Wilson Swan in England).

(from Visual Interface) or EMACS as their only design entry mechanism.

But a picture tells a thousand words, as they say, and graphical entry techniques remain popular at a variety of levels. For example, it is extremely common to use a block-level schematic editor to capture the design as a collection of high-level blocks that are connected together. The system might then be used to automatically create a skeleton HDL framework with all of the block names and inputs and outputs declared. Alternatively, the user might create a skeleton framework in HDL, and the system might use this to create a block-level schematic automatically.

From the user’s viewpoint, “pushing” down into one of these schematic blocks might automatically open an HDL editor. This could be a pure text-and-command-based editor like VI, or it might be a more sophisticated HDL-specific editor featuring the ability to show language keywords in different colors, automatically complete statements, and so forth.

Furthermore, when pushing down into a schematic block, modern design systems often give you a choice between entering and viewing the contents of that block as another, lower-level block-level schematic, raw HDL code, a graphical state diagram (used to represent an FSM), a graphical flowchart, and so forth. In the case of the graphical representations like state diagrams and flowcharts, these can subsequently be used to generate their RTL equivalents automatically (Figure 9-4).

Furthermore, it is common to have a tabular file containing information relating to the device’s external inputs and outputs. In this case, both the top-level block diagram and the tabular file will (hopefully) be directly linked to the same data and will simply provide different views of that data. Making a change in any view will update the central data and be reflected immediately in all of the views.
**A positive plethora of HDLs**

Life would be so simple if there were only a single HDL to worry about, but no one said that living was going to be easy. As previously noted, in the early days of digital IC electronics design (circa the 1970s), anyone who created an HDL-based design tool typically felt moved to create his or her own language to accompany it. Not surprisingly, the result was a morass of confusion (you had to be there to fully appreciate the dreadfulness of the situation). What was needed was an industry-standard HDL that could be used by multiple EDA tools and vendors, but where was such a gem to be found?

**Verilog HDL**

Sometime around the mid-1980s, Phil Moorby (one of the original members of the team that created the famous HIL0 logic simulator) designed a new HDL called Verilog. In 1985, the company he was working for, Gateway Design Automation, introduced this language to the market along with an accompanying logic simulator called Verilog-XL.
One very cool concept that accompanied Verilog and Verilog-XL was the Verilog programming language interface (PLI). The more generic name for this sort of thing is application programming interface (API). An API is a library of software functions that allow external software programs to pass data into an application and access data from that application. Thus, the Verilog PLI is an API that allows users to extend the functionality of the Verilog language and simulator.

As one simple example, let's assume that an engineer is designing a circuit that makes use of an existing module to perform a mathematical function such as an FFT. A Verilog representation of this function might take a long time to simulate, which would be a pain if all the engineer really wanted to do was verify the new portion of the circuit. In this case, the engineer might create a model of this function in the C programming language, which would simulate, say, 1,000 times faster than its Verilog equivalent. This model would incorporate PLI constructs, allowing it to be linked into the simulation environment. The model could subsequently be accessed from the Verilog description of the rest of the circuit by means of a PLI call providing a bidirectional link to pass data back and forth between the main circuit (represented in Verilog) and the FFT (captured in C).

Yet one more really useful feature associated with Verilog and Verilog-XL was the ability to have timing information specified in an external text file known as a standard delay format (SDF) file. This allowed tools like post-place-and-route timing analysis packages to generate SDF files that could be used by the simulator to provide more accurate results.

As a language, the original Verilog was reasonably strong at the structural (switch and gate) level of abstraction (especially with regard to delay modeling capability); it was very strong at the functional (Boolean equation and RTL) level of abstraction; and it supported some behavioral (algorithmic) constructs (Figure 9-5).
In 1989, Gateway Design Automation along with Verilog (the HDL) and Verilog-XL (the simulator) were acquired by Cadence Design Systems. The most likely scenario at that time was for Verilog to remain as just another proprietary HDL. However, with a move that took the industry by surprise, Cadence put the Verilog HDL, Verilog PLI, and Verilog SDF specifications into the public domain in 1990.

This was a very daring move because it meant that anybody could develop a Verilog simulator, thereby becoming a potential competitor to Cadence. The reason for Cadence’s largesse was that the VHDL language (introduced later in this section) was starting to gain a significant following. The upside of placing Verilog in the public domain was that a wide variety of companies developing HDL-based tools, such as logic synthesis applications, now felt comfortable using Verilog as their language of choice.

Having a single design representation that could be used by simulation, synthesis, and other tools made everyone’s life a lot easier. It is important to remember, however, that Verilog was originally conceived with simulation in mind; applications like
synthesis were something of an afterthought. This means that when creating a Verilog representation to be used for both simulation and synthesis, one is restricted to using a *synthesizable subset* of the language (which is loosely defined as whatever collection of language constructs your particular logic synthesis package understands and supports).

The formal definition of Verilog is encapsulated in a document known as the *language reference manual* (LRM), which details the syntax and semantics of the language. In this context, the term *syntax* refers to the grammar of the language—such as the ordering of the words and symbols in relation to each other—while the term *semantics* refers to the underlying meaning of the words and symbols and the relationships between the things they denote ... phew!

In an ideal world, an LRM would define things so rigorously that there would be no chance of any misinterpretation. In the real world, however, there were some ambiguities with respect to the Verilog LRM. Admittedly, these were corner-case conditions along the lines of “if a control signal on this register goes inactive at the same time as the clock signal triggers, which signal will be evaluated by the simulator first?” But the end result was that different Verilog simulators might generate different results, which is always somewhat disconcerting to the end user.

Verilog quickly became very popular. The problem was that different companies started to extend the language in different directions. In order to curtail this sort of thing, a nonprofit body called *Open Verilog International* (OVI) was established in 1991. With representatives from all of the major EDA vendors of the time, OVI’s mandate was to manage and standardize Verilog HDL and the Verilog PLI.

The popularity of Verilog continued to rise exponentially, with the result that OVI eventually asked the IEEE to form a working committee to establish Verilog as an IEEE standard. Known as IEEE 1364, this committee was formed in 1993. May 1995 saw the first official IEEE Verilog release, which is
formally known as IEEE 1364-1995, and whose unofficial designation has come to be Verilog 95.

Minor modifications were made to this standard in 2001; hence, it is often referred to as the Verilog 2001 (or Verilog 2K1) release. At the time of this writing, the IEEE 1364 committee is working feverishly on a forthcoming Verilog 2005 offering, while the design world holds its breath in dread anticipation (see also the section on “Superlog and SystemVerilog” later in this chapter).

**VHDL and VITAL**

In 1980, the U.S. Department of Defense (DoD) launched the *very high speed integrated circuit* (VHSIC) program, whose primary objective was to advance the state of the art in digital IC technology.

This program sought to address, among other things, the fact that it was difficult to reproduce ICs (and circuit boards) over the long life cycles of military equipment because the function of the parts wasn’t documented in a rigorous fashion. Furthermore, different components forming a system were often designed and verified using diverse and incompatible simulation languages and design tools.

In order to address these issues, a project to develop a new hardware description language called VHSIC HDL (or VHDL for short) was launched in 1981. One unique feature of this process was that industry was involved from a very early stage. In 1983, a team comprising Intermetrics, IBM, and Texas Instruments was awarded a contract to develop VHDL, the first official release of which occurred in 1985.

Also of interest is the fact that in order to encourage acceptance by the industry, the DoD subsequently donated all rights to the VHDL language definition to the IEEE in 1986. After making some modifications to address a few known problems, VHDL was released as official standard IEEE 1076 in 1987. The language was further extended in a 1993 release and again in 1999.
Initially, VHDL didn’t have an equivalent to Verilog’s PLI. Today, different simulators have their own ways of doing this sort of thing, such as ModelSim’s foreign language interface (FLI). We can but hope that these diverse offerings will eventually converge on a common standard.

DAC may be pronounced to rhyme with “sack,” or it may be spelled out as “D-A-C.”

As a language, VHDL is very strong at the functional (Boolean equation and RTL) and behavioral (algorithmic) levels of abstraction, and it also supports some system-level design constructs. However, VHDL is a little weak when it comes to the structural (switch and gate) level of abstraction, especially with regard to its delay modeling capability.

It quickly became apparent that VHDL had insufficient timing accuracy to be used as a sign-off simulator. For this reason, the VITAL initiative was launched at the Design Automation Conference (DAC) in 1992. VHDL Initiative toward ASIC Libraries (VITAL) was an effort to enhance VHDL’s abilities for modeling timing in ASIC and FPGA design environments. The end result encompassed both a library of ASIC/FPGA primitive functions and an associated method for back-annotating delay information into these library models, where this delay mechanism was based on the same underlying tabular format used by Verilog (Figure 9-6).

---

**Figure 9-6. Levels of abstraction (Verilog versus VHDL).**
Mixed-language designs

Once upon a time, it was fairly common for an entire design to be captured using a single HDL (Verilog or VHDL). As designs increased in size and complexity, however, it became more common for different portions of the design to be created by different teams. These teams might be based in different companies or even reside in different countries, and it was not uncommon for the different groups to be using different design languages.

Another consideration was the increasing use of legacy design blocks or third-party IP, where the latter refers to a design team purchasing a predefined function from an external supplier. As a general rule of thumb related to Murphy's Law, if you were using one language, then the IP you wanted was probably available only in the other language.

The early 1990s saw a period known as the HDL Wars, in which the proponents of one language (Verilog or VHDL) stridently predicted the imminent demise of the other... but the years passed and both languages retained strong followings. The end result was that EDA vendors began to support mixed-language design environments featuring logic simulators, logic synthesis applications, and other tools that could work with designs composed from a mixture of Verilog and VHDL blocks (or modules, depending on your language roots).

UDL/I

As previously noted, Verilog was originally designed with simulation in mind. Similarly, VHDL was created as a design documentation and specification language that took simulation into account. As a result one can use both of these languages to describe constructs that can be simulated, but not synthesized.

In order to address these problems, the Japan Electronic Industry Development Association (JEIDA) introduced its own HDL, the unified design language for integrated circuits (UDL/I) in 1990.
The key advantage of UDL/I was that it was designed from the ground up with both simulation and synthesis in mind. The UDL/I environment includes a simulator and a synthesis tool and is available for free (including the source code). However, by the time UDL/I arrived on the scene, Verilog and VHDL already held the high ground, and UDL/I never really managed to attract much interest outside of Japan.

**Superlog and SystemVerilog**

In 1997, things started to get complicated because that's when a company called Co-Design Automation was formed. Working away furiously, the folks at Co-Design developed a "Verilog on steroids" called Superlog.

Superlog was an amazing beast that combined the simplicity of Verilog with the power of the C programming language. It also included things like temporal logic, sophisticated design verification capabilities, a dynamic API, and the concept of assertions that are key to the formal verification strategy known as model checking. (VHDL already had a simple assert construct, but the original Verilog had nothing to boast about in this area.)

The two main problems with Superlog were (a) it was essentially another proprietary language, and (b) it was so much more sophisticated than Verilog 95 (and later Verilog 2001) that getting other EDA vendors to enhance their tools to support it would have been a major feat.

Meanwhile, while everyone was scratching their heads wondering what the future held, the OVI group linked up with their equivalent VHDL organization called VHDL International to form a new body called Accellera. The mission of this new organization was to focus on identifying new standards and formats, to develop these standards and formats, and to foster the adoption of new methodologies based on these standards and formats.

In the summer of 2002, Accellera released the specification for a hybrid language called SystemVerilog 3.0 (don't even ask me about 1.0 and 2.0). The great advantage to this
language was that it was an incremental enhancement to the existing Verilog, rather than the death-defying leap represented by a full-up Superlog implementation. Actually, SystemVerilog 3.0 featured many of Superlog’s language constructs donated by Co-Design. It included things like the assertion and extended synthesis capabilities that everyone wanted and, being an Accellera standard, it was well placed to quickly gain widespread adoption.

The current state of play (at the time of this writing) is that Co-Design was acquired by Synopsys in the fall of 2002. Synopsys maintained the policy of donating language constructs from Superlog to SystemVerilog, but no one is really talking about Superlog as an independent language anymore. After a little pushing and pulling, all of the mainstream EDA vendors officially endorsed SystemVerilog and augmented their tools to accept various subsets of the language, depending on their particular application areas and requirements. SystemVerilog 3.1 hit the streets in the summer of 2003, followed by a 3.1a release (to add a few enhancements and fix some annoying problems) around the beginning of 2004. Meanwhile, the IEEE is set to release the next version of Verilog in 2005. In order to avert a potential schism between Verilog 2005 and SystemVerilog, Accellera has promised to donate their SystemVerilog copyright to the IEEE by the summer of 2004.

**SystemC**

And then we have SystemC, which some design engineers love and others hate with a passion. SystemC—discussed in more detail in chapter 11—can be used to describe designs at the RTL level of abstraction.¹ These descriptions can subsequently be simulated 5 to 10 times faster than their Verilog or VHDL counterparts, and synthesis tools are available to convert the SystemC RTL into gate-level netlists.

¹ SystemC can support higher levels of abstraction than RTL, but those levels are outside the scope of this chapter; instead, they are discussed in more detail in chapter 11.
One big argument for SystemC is that it provides a more natural environment for hardware/software codesign and co-verification. One big argument against it is that the majority of design engineers are very familiar with Verilog or VHDL, but they are not familiar with the object-orientated aspects of SystemC. Another consideration is that the majority of today’s synthesis offerings represent hundreds of engineer years of development in translating Verilog or VHDL into gate-level netlists. By comparison, there are far fewer SystemC-based synthesis tools, and those that are available tend to be somewhat less sophisticated than their more traditional counterparts.

In reality, SystemC is more applicable to a system-level versus an RTL design environment. Having said this, SystemC seems to be gaining a lot of momentum in Asia and Europe, and the debate on SystemC versus SystemVerilog versus VHDL will doubtless be with us for quite some time.

**Points to ponder**

**Be afraid, be very afraid**

Most software engineers throw up their hands in horror when they look at another programmer’s code, and they invariably start a diatribe as to the lack of comments, consistency, whatever … you name it, and they aren’t happy about it.

They don’t know how lucky they are because the RTL source code for a design often sets new standards for awfulness! Sad to relate, the majority of designs described in RTL are almost unintelligible to another designer. In an ideal world, the RTL description of a design should read like a book, starting with a “table of contents” (an explanation of the design’s structure), having a logical flow partitioned into “chapters” (logical breaks in the design), and having lots of “commentary” (comments explaining the structure and operation of the design).
It’s also important to note that coding style can impact performance (this typically affects FPGAs more than ASICs). One reason for this is that, although they might be logically equivalent, different RTL statements can yield different results. Also, tools are part of the equation because different tools can yield different results.

The various FPGA vendors and EDA vendors are in a position to provide their customers with reams of information on particular coding styles and considerations with regard to their chips and tools, respectively. However, the following points are reasonably generic and will apply to most situations.

**Serial versus parallel multiplexers**

When creating RTL code, it is useful to understand what your synthesis tool is going to do in certain circumstances. For example, every time you use an if-then-else statement, the result will be a 2:1 multiplexer. This becomes interesting in the case of nested if-then-else statements, which will be synthesized into a priority structure. For example, assume that we have already declared signals Y, A, B, C, D, and SEL (for select) and that we use them to create a nested if-then-else (Figure 9-7).

```vhdl
if SEL == 00 then Y = A;
elseif SEL == 01 then Y = B;
elseif SEL == 10 then Y = C;
else Y = D;
end if;
```

![Figure 9-7. Synthesizing nested if-then-else statements.](image-url)
As before, the syntax used here is a generic one that doesn’t really reflect any of the mainstream languages. In this case, the innermost if-then-else will be the fastest path, while the outermost if-then-else will be the critical signal (in terms of timing). Having said this, in some FPGAs all of the paths through this structure will be faster than using a case statement. Speaking of which, a case statement implementation of the above will result in a 4:1 multiplexer, in which all of the timing paths associated with the inputs will be (relatively) equal (Figure 9-8).

```plaintext
case SEL of;
  00": Y = A;
  01": Y = B;
  10": Y = C;
otherwise: Y = D;
end case;
```

![4:1 MUX](image)

**Figure 9-8. Synthesizing a case statement.**

**Beware of latch inference**

Generally speaking, it’s a good idea to avoid the use of latches in FPGA designs unless you really need them. One other thing to watch out for: If you use an if-then-else statement, but neglect to complete the “else” portion, then most synthesis tools will infer a latch.

**Use constants wisely**

Adders are the most used of the more complex operators in a typical design. In certain cases, ASIC designers sometimes employ special versions using combinations of half-adders and full-adders. This may work very efficiently in the case of a gate array device, for example, but it will typically result in a very bad FPGA implementation.

When using an adder with constants, a little thought goes a long way. For example, “A + 2” can be implemented more
efficiently as “A + 1 with carry-in,” while “A – 2” would be better implemented as “A – 1 with carry-in.”

Similarly, when using multipliers, “A * 2” can be implemented much more efficiently as “A SHL 1” (which translates to “A shifted left by one bit”), while “A * 3” would be better implemented as “(A SHL 1) + A.”

In fact, a little algebra also goes a long way in FPGAs. For example, replacing “A * 9” with “(A SHL 3) + A” results in at least a 40 percent reduction in area.

**Consider resource sharing**

Resource sharing is an optimization technique that uses a single functional block (such as an adder or comparator) to implement several operators in the HDL code.

If you do not use resource sharing, then each RTL operation is built using its own logic. This results in better performance, but it uses more logic gates, which equates to silicon real estate. If you do decide to use resource sharing, the result will be to reduce the gate-count, but you will typically take a hit in performance. For example, consider the statement illustrated in Figure 9-9.

Note that frequency values shown in Figure 9-9 are of interest only for the purposes of this comparison, because these values will vary according to the particular FPGA architecture, and they will change as new process nodes come online.

The following operators can be shared with other instances of the same operator or with related operators on the same line:

*    
+    
-    
>    
<    
>=   
<=   

For example, a + operator can be shared with instances of other + operators or with – operators, while a * operator can be shared only with other * operators.
For not-so-technical readers, the circles with “>” symbols indicate comparators (circuits that compare two numbers to determine which is the larger); the circles with “+” symbols indicate adders; and the wedge-shaped blocks are 2:1 multiplexers that select between their inputs based on the value of the control signals coming out of the comparators.

if \( B > C \)  
then \( Y = A + B \);  
else \( Y = A + C \);  
end if;

![Diagram](image.png)

Total LUTs = 32  
Clock frequency = 87.7 MHz

Resource Sharing = ON  
(one adder)

Resource Sharing = OFF  
(two adders)

Total LUTs = 64  
Clock frequency = 133.3 MHz (+52%)

Figure 9-9. Resource sharing.

If nothing else, it’s a good idea to check whether or not your synthesis application has resource sharing enabled or disabled by default. And one final point is that resource sharing in ASICs can alleviate routing congestion, but it may actually cause routing problems in FPGAs.

**Last but not least**

Internal tri-state buses are slow in most FPGAs and should be avoided unless you are 100 percent confident that you know what you’re doing. If at all possible, use tri-state buffers only at the top-most level of the design. If you do wish to use internal tri-state buffers, then in the case of FPGA families that don’t support these gates, the majority of today’s synthesis tools provide automatic tri-state-to-multiplexer conversion (this basically involves converting the tri-state buffers specified in the RTL into corresponding LUT/CLB-based logic.)
Also, bidirectional buffers can cause timing loop problems, so if you use them, make sure that any false paths are clearly marked.
Silicon Virtual Prototyping for FPGAs

Just what is an SVP?

Before we leap headfirst into the concept of silicon virtual prototyping for FPGAs, it’s probably worth reminding ourselves how the silicon virtual prototype (SVP) concept originated in the ASIC world, some of the alternative SVP manifestations one might see in that world, and some of the problems associated with those manifestations.

As high-end ASIC devices containing tens of millions of logic gates appeared on the scene, capacity and complexity issues associated with these megadesigns caused design flows to become a little wobbly around the edges.

The problem is that, with traditional flows, many design issues do not become apparent until accurate timing analysis can be performed following extraction of realistic physical values (capacitance, resistance, and sometimes inductance), based on the results from place-and-route. This requires the engineers to go all of the way through the flow (including synthesis and place-and-route) before they discover a major problem that would have been better detected and resolved earlier in the process.

This is extremely irritating, and the end result often involves numerous time-consuming iterations that can so delay a design that it completely misses its time-to-market window. (In many cases there is only room in the market for the winner, and there’s no such thing as second place!)

One solution is to create an SVP, which is a representation of the design that can be generated relatively quickly, but which (hopefully) contains sufficient information to allow the
designers to identify and address a large proportion of potential problems before they undergo the time-consuming portions of the design flow. In theory, the time taken to iterate a design using an SVP can be measured in hours, as opposed to days or weeks using conventional design flows.

**ASIC-based SVP approaches**

As was discussed in the previous chapter, the role of logic synthesis is to accept an RTL representation of a design along with a set of timing constraints. The logic synthesis application automatically converts this RTL representation into a mixture of registers and Boolean equations, performs a variety of minimizations and optimizations (including optimizing for area and timing), and then generates a gate-level netlist that hopefully meets the original timing constraints.

Conventional logic synthesis solutions operate in the gate-size versus delay plane, which means they are constantly making trade-offs with regard to the size of gates and the delays associated with those gates. Due to their underlying modus operandi, these tools perform tremendous amounts of compute-intensive, time-consuming evaluations. Even worse, many of the optimization decisions performed by the synthesis tool are often rendered meaningless when the design is handed over to the physical implementation (place-and-route) portion of the flow.

**Gate-level SVPs (from fast-and-dirty synthesis)**

One key aspect of an SVP is the ability to generate it quickly and easily. The majority of current ASIC SVPs are based on the use of a gate-level netlist representation of the design that is subsequently placed using a rough-and-ready placement algorithm. Unfortunately, conventional synthesis tools consume too much time and computational resources to meet the speed demands of prototyping. Thus, some ASIC-based SVP flows make use of a fast-and-dirty synthesis engine (Figure 10-1).
This fast-and-dirty engine is typically based on completely different algorithms from the main synthesis application, for example, direct RTL mapping. Thus, the ensuing gate-level netlist used to form the SVP is not as accurate a representation of the design’s final implementation as one might hope for.

In turn, this means that once the SVP has been used to perform RTL exploration and timing analysis, engineers still have to perform a full-up logic synthesis (or physically aware synthesis) step using a completely different synthesis engine in order to generate the real netlist to be passed on to the physical implementation (place-and-route) tools.

So, the big problem with this SVP-based approach is that the prototyping tools and their methodologies are separate and distinct from the implementation tools and their methodologies. This leads to unpredictability of design convergence due to lack of correlation, which can result in time-consuming back-end-to-front-end iterations, which sort of defeats the whole purpose of using an SVP in the first place!

**Gate-level SVPs (from gain-based synthesis)**

As opposed to conventional logic synthesis that is based in the gate-size versus delay plane, a concept known as gain-based
synthesis\(^1\) is a kettle of fish of a different color (I never metaphor I didn’t like).

This form of synthesis is derived from ideas put forward by Ivan Sutherland, Bob Sproull, and David Harris in their 1999 book *Logical Effort: Designing Fast CMOS Circuits*.\(^2\) In this case, the synthesis engine uses logical effort concepts to establish a fixed-timing plane, and the physical implementation (place-and-route) tools subsequently work within this plane.

This means that all timing optimizations are completed and all circuit delays are determined and frozen by the end of the synthesis step. When the placement engine performs its task, it uses a size-driven algorithm in which all of the cells are dynamically sized to meet their timing budgets based on the actual loads they see. Following placement, a load-driven routing engine is used to tune the width and spacing of the tracks so as to maintain the original timing budgets and to ensure signal integrity.

One interesting point with regard to the gain-based approach is that the amount of computer memory and computational effort required to perform this type of synthesis are a fraction of that demanded by conventional synthesis tools. This means that a gain-based synthesis engine claims an order of magnitude increase in capacity over conventional synthesis approaches.

Another interesting point is that the gain-based synthesis engine automatically uses up any slack in path delays. This means that the smallest possible sizes are used for each gate that will just meet the timing budget. Thus, the resulting implementation occupies the smallest amount of silicon real estate, which significantly reduces congestion, power consumption, and noise problems.

\(^1\) At the time of this writing, one of the chief proponents of gain-based synthesis is Magma Design Automation (www.magma-da.com).

\(^2\) Ivan Sutherland is internationally renowned for his pioneering work on logic design.
"But," you cry, "what does all of this have to do with SVPs?" Well, the speed and capacity inherent to gain-based synthesis means that the same synthesis engine can be used for both prototyping and implementation (Figure 10-2).

![Figure 10-2. SVP based on gain-based synthesis.](image)

Basing both the prototyping and implementation environments on the same algorithms, tools, and methodologies provides high correlation and predictable design convergence and significantly reduces time-consuming back-end-to-front-end iterations.

**Cluster-level SVPs**

As discussed earlier, the majority of today’s SVPs are based on full-blown gate-level netlist representations of the design. Even though these representations may be generated using fast-and-dirty synthesis, they can still contain millions upon millions of logic gates, which can strain the capacity of the SVP’s placement and analysis engines.

One solution is to use the concept of _clustering_ as a basis for the SVP’s placement decisions and track-delay estimations. In this case the cells (gates and registers) generated by fast-and-dirty or gain-based synthesis are automatically gathered into groups called _clusters_. Each cluster typically consists of tens to
hundreds of cells, which means that they are small enough to preserve overall placement quality; however, the number of clusters is orders of magnitude smaller than the number of cells, providing extremely significant run-time improvements.

The actual number of cells may vary from cluster to cluster so as to keep the areas of the clusters as uniform as possible. In order to streamline computational complexity and capacity requirements, optimization and analysis are performed on the clustered data. Furthermore, in cases where two clusters are linked by multiple wires (which is a common occurrence), these wires are considered to be a single "weighted" wire for the purposes of estimating routing resource utilization, which has an effect on cluster placement.

**RTL-based SVPs**

A well-accepted engineering rule of thumb states that detecting, isolating, and resolving a problem at any stage of the design, implementation, or deployment process costs 10 times more than addressing the same problem at the previous stage in the process. In the case of digital ICs, there are three major breakpoints in the design flow with respect to analyzing area, timing, and so forth. (Figure 10-3)

![Figure 10-3. Major breakpoints with respect to analyzing area, timing, and so forth.](image-url)
The term timing closure refers to analyzing a design or architecture to detect and correct any problematic timing paths. Irrespective of the level it is performed at, timing closure is an iterative process, which means that the analyze-detect-correct steps typically need to be run a number of times in order to achieve convergence.

With regard to the levels of abstraction shown in Figure 10-3, postlayout timing analysis is the most accurate by far, but it is extremely expensive with regard to cost and time. Iterating at the postlayout level is a painful proposition, and design teams try very hard to avoid making changes at this level.

In the case of conventional flows, the first breakpoint for relatively accurate timing analysis occurs at the gate level following synthesis and in-place optimization (IPO). The problem is that getting to this post-IPO breakpoint using conventional flows requires the use of physically aware synthesis to provide a placed gate-level netlist. This approach is therefore extremely compute-intensive and time-consuming, and large blocks can take days to go through the full physical synthesis and timing analysis process. Not only does this stretch out the design and timing closure process, but it also ties up expensive EDA tools that could be being used for implementing chips rather than analyzing their timing.

One alternative is to use a gate-level SVP as discussed above; but, once again, these representations have their own problems, including requiring the use of some form of compute-intensive and time-consuming synthesis and placement.

Another approach is to work with an RTL-based SVP, which allows engineers to quickly identify and address paths that will cause downstream timing problems. In order to wrap one’s brain around how this works, it’s first necessary to understand that there’s a related application that takes the logical

---

3 At the time of this writing, one of the chief proponents of RTL-based SVPs is InTime Software (www.intimesw.com).
and physical (LEF and DEF) definition files associated with an
ASIC cell library and generates a corresponding design kit
database to be used by the RTL-based SVP (Figure 10-4).

![Diagram](image)

**Figure 10-4. Generating a design kit.**

It's important to note that such a design kit is not a library
of characterized gates, but is instead a database of character-
ized logical functions (such as counters, XOR trees, etc.). The
design kit generator captures the behavior of these logical
functions, including timing and area estimations.

The RTL-based SVP generator and analysis engine subse-
quently accepts the RTL code for the design, the time
constraints associated with the design block (in industry-
standard SDF format), and the design kit associated with the
target cell library. As the SVP generator reads in the RTL, it
converts it into a netlist of entities called work functions. Each
work function is an abstraction that directly maps onto an
equivalent function in the design kit.

Once the RTL has been converted into a netlist of work
functions, the SVP generator performs identical logical oper-
ations to those that are typically performed at the gate level,
including common subexpression elimination, constant
propagation, loop unraveling, the removal of all redundant
functional computations, and so forth.

The SVP generator and analysis engine uses the resulting
minimal irredundant network of work functions to perform a
"virtual placement" of these functions. This placement is
then used to generate accurate area estimates, which are
subsequently used to generate accurate time estimates. In con-
junction with the design kit, the SVP generator and analysis engine understands how the various synthesis engines will weight various factors and modify their implementation strategies (such as swapping counter realizations) in order to meet the specified timing constraints. All of these factors are taken into account when performing the analysis.

Proponents of RTL-based SVPs claim a 40-fold speed increase as compared to generating a post-IPO, pre-place-and-route gate-level netlist using a physically aware synthesis approach. In an example 4.5-million-gate design circa 2003, this equated to a 2.5-hour iteration to generate and analyze an RTL-based SVP as compared to 99 hours to generate and analyze a post-IPO gate-level netlist.

Of course the big question is, just how accurate are RTL-based SVPs? The supporters of this form of SVP claim that its timing analysis results typically correlate to post-IPO delays with an error of 20 percent or less (worst-case errors may rise to 30 percent). Although this may sound pretty dire, the latest generation of synthesis tools is capable of closing timing on RTL that is within 20 to 30 percent of the desired timing (it’s the paths that are off by 80, 150, 200 percent, and higher that cause problems). Thus, RTL-based SVPs are accurate enough to allow design engineers to generate RTL code that can subsequently be fully implemented by the downstream syntheses and layout engines.

I know, I know. We’ve digressed again, although you have to admit that this is all interesting stuff! But now let’s return to pondering FPGAs.

**FPGA-based SVPs**

Not surprisingly, multimillion-gate FPGA designs are hitting the same problems that befell ASICs, including the fact that it takes ages to place, route, and perform timing analysis on the little rascals.

One particularly painful aspect of this process is that, although the original RTL representation of the design is
almost invariably hierarchical, \(^4\) the FPGA's place-and-route tools typically end up working on flattened representations of the design. This means that even if you make the smallest of changes to a single block of RTL code and resynthesize only that block, you end up having to rerun place-and-route on the entire design. In turn, this means that you can grow to be old and gray before you finally get to achieve timing closure on your design.

In order to address these problems, some EDA vendors have started to provide tools that support the concept of an FPGA SVP by providing a mixture of floor planning and pre-place-and-route timing analysis. This is coupled with the ability to perform place-and-route on individual design blocks, which dramatically speeds up the implementation process. \(^5\)

This form of SVP commences with a graphical top-down view of the target FPGA device showing all of the internal logical resources, such as LUTs, registers, slices, CLBs, embedded RAMs, multipliers, and so forth.

Following the logic synthesis step (using the synthesizer of your choice), the SVP generator loads the ensuing hierarchical LUT/CLB-level netlist, along with any associated timing and physical constraints, and automatically creates an initial floor plan. This auto-generated floor plan shows a collection of square and/or rectangular blocks, each of which corresponds to a top-level module in the design. Furthermore, if any of these top-level modules itself contains submodules, then these are shown as embedded blocks in the floor plan (and so on down through the hierarchy).

The SVP generator performs its own initial placement of the resources (LUTs, registers, RAMs, multipliers, etc.) used by each block. These resources are also shown in the top-down view of the device, along with graphical representations

\(^4\) By “hierarchical,” we mean that the top level of the design is typically formed from a number of functional modules, which may themselves call submodules and so forth.

\(^5\) At the time of this writing, one of the chief proponents of FPGA SVPs, in the form described here, is Hier Design (www.hierdesign.com).
as to the amount of routing resources required to link the various blocks together.

**Interactive manipulation**

The initial placement of the design in the SVP allows it to provide accurate timing estimations on a block-by-block basis prior to running place-and-route. If any potential problem areas are detected, you can interactively modify the floor plan in order to address them.

The simplest form of manipulation is to reshape the rectangular blocks in the floor plan by pulling their sides to make them taller, thinner, shorter, or fatter. Alternatively, you can create more complex outlines such as “L,” “U,” and “T” shapes (pretty much any contour you can form out of squares and rectangles).

Next, you can move the blocks around. When you grab a block and start to drag it across the face of the device, the system will provide a graphical indication as to whether or not there are the necessary resources required to implement that block at its current location (you can only drop the block in an area where there are sufficient resources). Furthermore, as you manipulate a block by reshaping it or moving it around, the system dynamically displays the utilization of resources (LUTs, registers, RAMs, multipliers, etc.) inside that block relative to the total amount of each resource type currently encompassed by that block.

You can also split existing blocks into two or more sub-blocks, which you can then manipulate independently. Alternatively, you can merge two or more blocks into a single block. Also, in some cases (say, areas of control logic), you might wish to pull one or more subblocks out of their parent blocks and move them up to the top level of the design, at which point you can reshape them, merge them together, move them around, and so forth. Much of this reflects a different philosophy of how one might use an ASIC floor-planning tool. In the case of an ASIC, for example, if you have two
blocks with lots of interconnect between them, you would typically place them side by side. By comparison, in the case of an FPGA, merging the blocks (thereby allowing the place-and-route tools to do a much better job of optimization using local versus global routing resources) might provide a more efficacious solution.

Furthermore, you aren’t limited to manipulating blocks only as described in the original RTL hierarchy. You can actually manipulate individual FPGA resources like LUTs, registers, slices, CLBs, and the like. This includes dragging them around and repositioning them within their current hierarchical block, dragging them from one hierarchical block to another, creating new blocks, and dragging groups of LUTs from one or more existing blocks into this new block, and so forth.

Where things start to get really clever is that, if you go back to make changes to your original RTL and resynthesize those modules, then when you reimport the resulting LUT/CLB-level netlist(s), the SVP generator sorts everything out for you and loads the right logic into the appropriate floor-plan blocks. (How do they do it? I don’t have a clue!)

**Incremental place-and-route**

As soon as you are ready to rock and roll, you can select one or more floor-plan blocks and kick off the FPGA vendor’s place-and-route software. Each block is treated as an individual entity, so once you’ve laid out a block, it will remain untouched unless you decide you want to change it. This has a number of advantages. First of all, place-and-route run times for individual blocks are extremely small compared to the traditional times associated with full-up multimillion-gate designs.

And even if you add up the place-and-route times for running all of the blocks individually, the total elapsed time is much less than it would be if one were performing place-and-route on the design in its entirety. This is because the complexity (and associated run times) of place-and-route increases
in a nonlinear manner as the size of the block being processed increases. Furthermore, once you’ve run place-and-route on all of the blocks, you can make changes to individual blocks and rerun place-and-route only on those blocks without affecting the rest of the chip.

An additional advantage associated with this SVP approach is that it lends itself to creating and preserving IP. That is, once a block has undergone place-and-route, you can lock it down and export it as a new structural LUT/CLB-level netlist along with its associated physical and timing constraints. This block can subsequently be used in other designs (its placement is relative, which means that it can be dragged around the chip and relocated as discussed above).

**RTL-based FPGA SVPs**

In an ideal world, it would be nice to be able to work with RTL-based FPGA SVPs. The various FPGA and EDA vendors do provide RTL-level floor-planning tools with varying degrees of sophistication. At the time of this writing, however, there is no FPGA equivalent to the state-of-the-art in RTL-based ASIC SVP technology (but we will doubtless see such a beast in the not-so-distant future).
Problems with traditional HDL-based flows

With regard to the traditional HDL-based flows introduced in chapter 9, a design commences with an original concept, whose high-level definition is determined by system architects and system designers. It is at this stage that macro-architecture decisions are made, such as partitioning the design into hardware and software components (see also chapter 13).

The resulting specification is then handed over to the hardware design engineers, who commence their portion of the development process by performing microarchitecture definition tasks such as detailing control structures, bus structures, and primary data path elements. These microarchitecture definitions, which are often performed in brainstorming sessions on a whiteboard, may include performing certain operations in parallel versus sequential, pipelining portions of the design versus nonpipelining, sharing common resources (for example, two operations sharing a single multiplier, versus using dedicated resources) and so forth.

Eventually, the design intent is captured by writing RTL VHDL/Verilog. Following verification via simulation, this RTL is then synthesized down to a structural netlist suitable for use by the target technology’s place-and-route applications (Figure 11-1).

At the time of this writing, these VHDL or Verilog-based flows account for around 95 percent of all ASIC and FPGA designs; however, there are a number of problems associated with these flows:

Note that this chapter focuses on C/C++ flows in the context of generic digital designs. Considerations such as quantization (commencing with floating-point representations which are subsequently coerced into their fixed-point counterparts) are covered in the discussions on DSP-centric designs in chapter 12.
In the case of an FPGA target, the LUT/CLB-level netlist may be presented in EDIF, VHDL, or Verilog depending on the FPGA vendor.

With regards to physically aware synthesis-based flows, EDIF remains the “netlist of choice.” In this case, the placement information may be incorporated in the EDIF itself or presented in an external “constraints” side-file.

**Figure 11-1. Traditional (simplified) HDL-based flows.**

- **Capturing the RTL is time-consuming**: Even though Verilog and VHDL are intended to represent hardware, it is still time-consuming to use these languages to capture the functionality of a design.
- **Verifying RTL is time-consuming**: Using simulation to verify large designs represented in RTL is computationally expensive and time-consuming.
- **Evaluating alternative implementations is difficult**: Modifying and reverifying RTL to perform a series of what-if evaluations of alternative microarchitecture implementations is difficult and time-consuming. This means that the number of evaluations the design team can perform may be limited, which can result in a less-than-optimal implementation.
- **Accommodating specification changes is difficult**: If any changes to the specification are made during the course of the project, folding these changes into the RTL and performing any necessary reverification can be painful and time-consuming. This is a significant consideration in certain application areas, such as wireless projects, because broadcast standards and protocols are constantly evolving and changing.
- **The RTL is implementation specific**: Realizing a design in an FPGA typically requires a different RTL coding style from that used for an ASIC implementation (see also the discussions in Chapters 7, 9, and 18). This
means that it can be extremely difficult to retarget a complex design represented in RTL from one implementation technology to another. This is of concern when one is migrating an existing ASIC design into an FPGA equivalent or creating an FPGA design to be used as a prototype for a future ASIC implementation.

One way to view this is that all of the implementation intelligence associated with the design is hard-coded into the RTL, which therefore becomes implementation specific. It’s important to understand that this implementation specificity goes beyond the coarse ASIC-versus-FPGA boundary, which dictates that RTL intended for an FPGA implementation is not suitable for an optimal ASIC realization, and vice versa. Even assuming a single target device architecture, the way in which a set of algorithms is used to process data may require a number of different micro-architecture implementations, depending on the target application areas.

Actually, to be scrupulously fair, we should probably note that the same RTL may be used to drive both ASIC and FPGA implementations. The reason for doing this is to avoid the risk of introducing a functional bug into the RTL when retargeting the code, but there is typically a penalty to be paid. That is, if code originally targeted toward an FPGA implementation is subsequently used to drive an ASIC implementation, the resulting ASIC will typically require more silicon real estate and have higher power consumption as compared to using RTL created with an ASIC architecture in mind. Similarly, if code originally targeted toward an ASIC implementation is subsequently used to drive an FPGA implementation, the ensuing FPGA will typically take a significant performance hit as compared to using RTL created with an FPGA architecture in mind.

- **RTL is less than ideal for hardware-software codesign:** System-on-chip (SoC) devices are generally under-

---

1895: America.
Dial telephones go into Milwaukee’s city hall.
stood to be those that include microprocessor cores. Irrespective of whether these designs are to be realized using ASICs or FPGAs, today’s SoCs are exhibiting an ever-increasing amount of software content. When coupled with increased design reuse on the hardware side, in many cases it is necessary to verify the software and hardware concurrently so as to completely validate such things as the system diagnostics, RTOS, device drivers, and embedded application software. Generally speaking, it can be painful verifying (simulating) the hardware represented in VHDL or Verilog in conjunction with the software represented in C/C++ or assembly language.

One approach that addresses the issues enumerated above is to perform the initial design capture at a higher level of abstraction than can be achieved with RTL VHDL/Verilog. The first such level is to use some form of C/C++, but as usual nothing is simple because there are a variety of alternatives, including SystemC, augmented C/C++, and pure C/C++.

**C versus C++ and concurrent versus sequential**

Before we leap into the fray, we should tie down a couple of points to ensure that we’re all marching in step to the same beat. First, there is a wide variety of programming languages available, but—excepting specialist application areas—the most commonly used by far are traditional C and its object-oriented offspring C++. For our purposes here, we will refer to these collectively as C/C++.

The next point of import is that, by default, statements in languages like C/C++ are executed sequentially. For example, assuming that we have already declared three integer variables called $a$, $b$, and $c$, then the following statements

```plaintext
a = 6; /* Statement in C/C++ program */
b = 2; /* Statement in C/C++ program */
c = 9; /* Statement in C/C++ program */
```
would, perhaps not surprisingly, occur one after the other. However, this has certain implications; for example, if we now assume that the following statements occur sometime later in the program

```c
a = b;  /* Statement in C/C++ program */
b = a;  /* Statement in C/C++ program */
```

then `a` (which initially contained 6) will be loaded with the value currently stored in `b` (which is 2). Next, `b` (which initially contained 2) will be loaded with the value currently stored in `a` (which is now 2), so both `a` and `b` will end up containing the same value.

The sequential nature of programming languages is the way in which software engineers think. However, hardware design engineers have quite a different view of the world. Let’s assume that a piece of hardware contains two registers called `a` and `b` that are driven by a common clock signal. Let’s further assume that these registers have previously been loaded with values of 6 and 2, respectively. Finally, let’s assume that at some point in the HDL code, we see the following statements:

```vhdl
a = b;  /* Statement in VHDL/Verilog Code */
b = a;  /* Statement in VHDL/Verilog Code */
```

As usual, the above syntax doesn’t actually represent VHDL or Verilog; it’s just a generic syntax used only for the purposes of this example. Generally speaking, hardware engineers would expect both of these statements to be executed *concurrently* (at the same time). This means that `a` (which initially contained 6) will be loaded with the value stored in `b` (which was 2) while—at the same time—`b` (which initially contained 2) will be loaded with the value stored in `a` (which was 6). The end result is that the initial contents of `a` and `b` will be exchanged.

As usual, of course, the above is something of a simplification. However, it’s fair to say that HDL statements will
SystemC is "managed" by the Open SystemC Initiative (OSCI). This is an independent not-for-profit organization composed of companies, universities, and individuals dedicated to promoting SystemC as an open-source standard for system-level design.

The code for SystemC—along with an integrated simulator and design environment—is available from www.systemc.org.

execute concurrently by default, unless sequential behavior is forced by means of techniques like blocking assignments. Thus, by default, RTL-based logic simulators will execute the statements shown above in this concurrent manner; similarly RTL-based logic synthesis tools will generate hardware that handles these two activities simultaneously. By comparison, unless explicitly directed to do otherwise (by means of the techniques introduced later in this chapter), C/C++ statements will execute sequentially.

**SystemC-based flows**

**What is SystemC (and where did it come from)?**

Before we come to consider SystemC-based flows, it is probably a good idea to briefly summarize just what SystemC is, because there is typically some confusion on this point (not the least in the mind of the author).

**SystemC 1.0**

One of the underlying concepts behind SystemC is that it is an open-source environment to which everyone contributes. As an example, consider Linux, which was rough around the edges at first. Based on contributions from different folks, however, Linux eventually became a real operating system (OS) with the potential to challenge Microsoft.

In this spirit, a relatively undocumented SystemC 1.0 was let loose to roam wild and free circa 2000. SystemC 1.0 was a C++ class library that facilitated the representation of notions such as concurrency (things happening at the same time), timing, and I/O pins. By means of this class library, engineers could capture designs at the RTL level of abstraction.

One advantage of this early incarnation was that it facilitated hardware/software codesign environments. Another was that SystemC representations at the RTL level of abstraction might simulate 5 to 10 times faster than their VHDL and Ver-

On the downside, it was harder and more time-consuming to capture an RTL-level design in SystemC 1.0 than with VHDL or Verilog. Furthermore, there was a scarcity of design tools that could synthesize SystemC 1.0 representations into netlist-level equivalents with any degree of sophistication.

**SystemC 2.0**

Later, in 2002, SystemC 2.0 arrived on the scene. This augmented the 1.0 release with some high-level modeling constructs such as FIFOs (a form of memory that can accept and subsequently make available a series of words of data and that operates on a first-in first-out principle). The 2.0 release also included a variety of behavioral, algorithmic, and system-level modeling capabilities, such as the concepts of transactions and channels (which are used to describe the communication of data between blocks at an abstract level).

In order to gain a little perspective on all of this, let's first consider a typical scenario of how things would have worked using the original SystemC 1.0. As a simple example, let's assume that we have two functions called $f(x)$ and $g(x)$ that have to communicate with each other. (Figure 11-2).

![Diagram](image-url)

**Figure 11-2. Interfacing in SystemC 1.0.**

---

1 This is design-dependent; in reality, some SystemC RTL-level simulation run times are at parity with their HDL counterparts.
In this case, the interface between the blocks would have to be defined at the pin level. The real problem with this approach occurs when you are in the early stages of a design, because you are already defining implementation details such as bus widths. This makes things difficult to change if you wish to experiment with different what-if architectural scenarios. This aspect of things became much easier with SystemC 2.0, which allowed abstract interfaces to be declared between the blocks (Figure 11-3).

![Figure 11-3. Interfacing in SystemC 2.0.](image)

Now, the interfacing between the blocks can be performed at the level of abstract records on the basis that, in the early stages of the design cycle, we don’t really care how data gets from point a to point b, just that it does get there somehow.

These abstract interfaces facilitate performing architectural evaluation early in the design cycle. Once the architecture starts to firm up, you can start refining the interface by using high-level constructs such as a FIFO to which one would assign attributes like width and depth and characteristics like blocking write, nonblocking read, and how to behave when empty or full. Still later, this logical interface can be replaced by a completely specified (pin-level) interface that binds the functional blocks together at a more physical level.
**Levels of abstraction**

Truth to tell, this is where things start to become a little fuzzy around the edges, not the least because one runs into different definitions depending on to whom one is talking. As a first pass, however, we might take a stab at capturing the different levels of SystemC abstraction, as shown in Figure 11-4.

![Figure 11-4. Levels of SystemC abstraction.](image)

This is why things become confusing, because SystemC can mean all things to all people. To some it's a replacement for RTL VHDL/Verilog, while to others it's a single language that can be used for system-level specification, algorithmic and architectural analysis, behavioral design, and testbenches for use in verification.

One area of confusion comes when you start to talk about behavioral synthesis. This encompasses certain aspects of both the algorithmic and transactional levels (in the latter case, however, you have to be careful as to how to define your transactions).

**SystemC-based design-flow alternatives**

This is a tricky one because one might go various ways here. For example, many of today's designs begin life as com-
plex algorithms. In this case, it is very common to start off by creating a C or C++ representation. This representation can be used to validate the algorithms by compiling it into a form that can be run (simulated) 1,000 or more times faster than an RTL equivalent.

In the case of the HDL-based flows discussed in chapter 9, this C/C++ representation of the algorithms would then be hand-translated into RTL VHDL/Verilog. The C/C++ representation will typically continue to be used as a golden model, which means it can be linked into the RTL simulator and run in parallel with the RTL simulation. The results from the C/C++ and RTL models can be compared so as to ensure that they are functionally equivalent.

Alternatively, in one flavor of a SystemC-based flow, the original C/C++ model could be incrementally modified by adding timing, concurrency, pin definitions, and so forth to transform it to a level at which it would be amenable to SystemC-based RTL or behavioral synthesis.

In another flavor of a SystemC-based flow, the design might be initially captured in SystemC using system, algorithmic, or transaction-level constructs that could be used for verification at a high level of abstraction. This representation could then be incrementally modified to bring it down to a level at which it would be amenable to SystemC-based RTL or behavioral synthesis.

Irrespective of the actual route by which one might get there, let’s assume that we are in possession of a SystemC representation of a design that is suitable for SystemC-based behavioral or RTL synthesis. In this case, there are two main design-flow alternatives, which are (1) to translate the System C into RTL VHDL/Verilog automatically and then to use conventional RTL synthesis technology, or (2) to use SystemC-based synthesis to generate an implementation-level netlist directly.

There are two schools of thought here. One says that synthesizing the SystemC directly into the implementation-level netlist offers the cleanest, fastest, and most efficient route.
However, another view is that it's better to translate the SystemC into RTL VHDL/Verilog first because RTL is the way design engineers really visualize their world; that this level is a natural staging point for integrating design blocks (including third-party IP) originating from multiple sources; and that Verilog/VHDL synthesis technology is extremely mature and powerful (as compared to SystemC-based synthesis technology).

But we digress. Both of these flows can be applied to ASIC or FPGA targets (Figure 11-5).

![Figure 11-5. Alternative SystemC flows.](image)

The first SystemC synthesis applications were predominantly geared toward ASIC flows, so they didn’t do a very good job at inferring FPGA-specific entities such as embedded RAMs, embedded multipliers, and so forth. More recent incarnations do a much better job of this, but the level of sophistication exhibited by different tools is a moving target, so the prospective user is strongly advised to perform some in-depth evaluations before slapping a bundle of cash onto the bargaining table.

Note that figure 11-5 shows the use of implementation-specific SystemC to drive the ASIC versus FPGA flows. As soon as you start coding at the RTL level and adding timing
concepts, be it in VHDL, Verilog, or SystemC, then achieving an optimal implementation requires that the code be written with a specific target architecture in mind.

Once again, having said this, the same SystemC can be used to drive both ASIC and FPGA flows, but there is typically a penalty to be paid. If SystemC code originally targeted toward an FPGA implementation is subsequently used to drive an ASIC flow, the resulting ASIC will typically require more silicon real estate and have higher power consumption as compared to using code created with an ASIC architecture in mind. Similarly, if code originally targeted toward an ASIC implementation is subsequently used to drive an FPGA flow, the ensuing FPGA will typically take a significant performance hit as compared to using code created with an FPGA architecture in mind. This is primarily a result of hard-coding the microarchitecture definition in the source.

Love it or loath it

Depending on whom you are talking to, folks either love SystemC or they loath it. Most would agree that SystemC 2.0 is very promising and that there’s no other language that provides the same capabilities (some of these capabilities are being added into SystemVerilog, but not all of them).

On the downside, a lot of design engineers are reasonably proficient at writing C, but most of them are significantly less familiar with the object-oriented aspects of C++. So requiring them to use SystemC means giving them more power on the one hand, while thrusting them into a world they don’t like or understand on the other. It’s also true that while SystemC can be very useful for verification and high-level system modeling, in some respects it’s still relatively immature toolwise with regard to actual implementation flows.

One school of thought says that, although SystemC is difficult to write by hand and also difficult to synthesize, which makes it a somewhat clumsy specification language, it does provide a powerful framework for simulation across languages and levels of abstraction.
At the time of this writing, a number of companies that were strong supporters of SystemC in the United States have grown somewhat less vocal over the last few years. On the other hand, SystemC is gaining some ground in Europe and Asia. What does the future hold? Wait a few years, and I'll be happy to tell you!

**Augmented C/C++-based flows**

*What do we mean by augmented C/C++?*

There are two ways in which standard C/C++ can be augmented to extend its capabilities and the things it can be used to represent. The first is to include special comments, known by some as commented directives or pragmas, into the pure C/C++ code. These comments can subsequently be recognized and interpreted by parsers, precompilers, compilers, and other tools and used to add constructs to the code or modify the way in which it is processed.\(^2\) One significant drawback to this approach is that simulation requires the use of proprietary C/C++ compilers as opposed to using standard off-the-shelf compilers. This limits the options customers have and is only viable if standards are developed for multiple EDA vendors to leverage.

The other way in which C/C++ can be augmented is to add special keywords and statements into the language. This is a very popular technique, and there are a veritable plethora of such language variants roaming wild and free around the world, each tailored toward a different application area. One downside of this approach is that, once again, it requires proprietary C/C++ compilers; otherwise, tools such as simulators that have not been enhanced to understand these new keywords and statements will crash and burn. A common solution

---

\(^2\) One example of this form of C/C++ augmentation is demonstrated by 0-In Design Automation (www.0in.com) for use with its *assertion-based verification* (ABV) technology. Another example of particular relevance here is Future Design Automation (www.future-da.com), which employs this technique with its C/C++ to RTL synthesis engine.
to this problem is to wrap standard #ifdef directives around the new keywords and statements such that a precompiler can be used to discard them as required (this is somewhat inelegant, but it works).

In the case of capturing the functionality of hardware for ASIC and FPGA designs, it is necessary to augment standard C/C++ with special statements to support such concepts as clocks, pins, concurrency, synchronization, and resource sharing.\(^3\)

Assuming that you have an initial model represented in pure C/C++, the first step would be to augment it with clock statements, along with interface statements used to define the input and output pins. You could then use an appropriate synthesis tool to generate an implementation (as discussed below). However, because C/C++ is by nature sequential, the resulting hardware can be horribly slow and inefficient if the synthesis tool is not capable of locating potential parallelisms and exploiting them.

For example, assume that we have the following statements in a C/C++ representation of the design:

```c
a = 6;    /* Standard C/C++ statement */
b = 2;    /* Standard C/C++ statement */
c = 9;    /* Standard C/C++ statement */
d = a + b; /* Standard C/C++ statement */
```

By default, each = sign is assumed by the synthesis application to represent one clock cycle. Thus, if the above code were left as is, the augmented C/C++ synthesis tool would generate hardware that loaded variable (register) \(a\) with 6 on the first clock, then \(b\) with 2 on the next clock, then \(c\) with 9 on the next clock, and so forth. Thus, by hardware standards, this would run horribly slowly.

\(^3\) A big player in this form of C/C++ augmentation for ASIC and FPGA design capture, simulation, and synthesis is Celoxica (www.celoxica.com) with its Handel-C language.
Of course, most synthesis tools would be capable of locating and exploiting the potential parallelisms in the above example, but they might well miss more complex cases that require human consideration and intervention. For the purposes of these discussions, however, we shall continue to work with this simple test case. The point is that an augmented C/C++ language will have keywords like “parallel” (or “par”) and “sequential” (or “seq”) that will instruct the downstream synthesis application as to which statements should be executed in parallel, and so forth. For example:

```c
parallel;       /* Augmented C/C++ statement */
a = 6;          /* Standard C/C++ statement */
b = 2;          /* Standard C/C++ statement */
c = 9;          /* Standard C/C++ statement */
sequential;    /* Augmented C/C++ statement */
d = a + b;      /* Standard C/C++ statement */
```

In this case, the `parallel` statement instructs the synthesis tool that the following statements can be implemented concurrently, while the `sequential` statement implies that the preceding operations must occur prior to any subsequent actions taking place. Of course, these parallel and sequential statements can be nested as required.

Things become more complex in the case of loops, depending on whether the designer wishes to unravel them partially or fully. Just to give a point of reference, we might visualize a loop as being something like “for i = 1 to 10 in increments of 1 do xxxxx, yyyy, and zzzzz”. In some cases, it may be possible to simply associate a parallel or sequential statement with the loop, but if more subtlety is required, the designer may be obliged to completely rewrite these constructs.

It may also be necessary to add “share” statements if resource sharing is required, and “channel” statements to share signals between expressions, and the list goes on.
**Augmented C/C++ design-flow alternatives**

As usual, one might go various ways here. As we previously discussed, in the case of a design that begins life as a suite of algorithms, it is very common to start off by creating a C or C++ representation. Following verification, this C/C++ model can be incrementally modified by adding statements for clocks, pins, concurrency, synchronization, and resource sharing so as to make the model suitable for the appropriate synthesis utility. Alternatively, the design might be captured using the augmented C/C++ language from the get-go.

Irrespective of the actual route we might take to get there, let’s assume that we are in possession of an augmented C/C++ representation of a design that is suitable for synthesis. Once again, there are two main design-flow alternatives, which are (1) to translate the augmented C/C++ into Verilog or VHDL at the RTL level of abstraction automatically and to then use conventional RTL synthesis technology, or (2) to use an appropriate augmented RTL synthesis engine.

And, once again, one school of thought says that synthesizing the augmented C/C++ directly into the implementation-level netlist offers the cleanest, fastest, and most efficient route. Others say that the RTL Verilog/VHDL level is the natural staging post for design integration and that today’s RTL synthesis technology is extremely mature and powerful.

Both of these flows can be applied to ASIC or FPGA targets (Figure 11-6). The first augmented C/C++ synthesis applications were predominantly geared toward ASIC flows. This meant that these early incarnations didn’t do a tremendous job when it came to inferring FPGA-specific entities such as embedded RAMs, embedded multipliers, and so forth. More recent versions of these tools do a much better job at this, but, as usual, the prospective user is strongly advised to perform some in-depth evaluations before handing over any hard-earned cash.

Note that figure 11-6 shows the use of implementation-specific code to drive the ASIC versus FPGA flows because
achieving an optimal implementation requires that the code be written with a specific target architecture in mind. In reality, the same code can be used to drive both ASIC and FPGA flows, but there is usually a penalty to be paid (see the discussions on SystemC for more details).

**Pure C/C++-based flows**

Last, but not least, we come to pure C/C++-based flows. In reality, the term pure C/C++ actually refers to industry-standard C/C++ that is minimally augmented with SystemC data types to allow specific bit widths to be associated with variables and constants.

Although relatively new, pure C/C++-based flows offer a number of advantages as compared to other C-based flows and traditional Verilog-/VHDL-based flows:

- **Creating pure C/C++ is fast and efficient**: Pure untimed C/C++ representations are more compact and easier to

---

At the time of this writing, perhaps the best example of a pure C/C++ based flow is provided by Precision C Synthesis from Mentor (www.mentor.com). Also of interest is the SPARK C-to-VHDL synthesis tool developed at the Center for Embedded Computer Systems, University of California, San Diego and Irvine (www.cecs.uci.edu/~spark).
create and understand than equivalent SystemC and augmented C/C++ representations (and they are much more compact than their RTL equivalents, requiring perhaps 1/10th to 1/100th of the code).

- **Verifying C/C++ is fast and efficient**: A pure untimed C/C++ representation will simulate significantly faster than a timed SystemC or augmented C/C++ model and 100 to 10,000 times faster than an equivalent RTL representation. In fact, pure C/C++ models are already widely created and used by system designers for algorithm and system validation.

- **Evaluating alternative implementations is fast and efficient**: Modifying and reverifying pure untimed C/C++ to perform a series of what-if evaluations of alternative microarchitecture implementations is fast and efficient. This facilitates the design team’s ability to arrive at fundamentally superior microarchitecture solutions. In turn, this can result in significantly smaller and faster designs as compared to flows based on traditional hand-coded RTL methods.

- **Accommodating specification changes is relatively easy**: If any changes to the specification are made during the course of the project, it’s relatively easy to implement and evaluate these changes in a pure untimed C/C++ representation, thereby allowing the changes to be folded into the resulting implementation.

Furthermore, as noted earlier in this chapter, one of the most significant problems associated with existing SystemC and augmented C/C++-based design flows is that the implementation intelligence associated with the design has to be hard-coded into the model, which therefore becomes implementation specific.

A key aspect associated with a pure untimed C/C++-based design flow is that the code presented to the synthesis engine is just what someone would write if he or she didn’t have any
preconceived hardware implementation or target device architecture in mind. This means that the C/C++ code that system designers write today is an ideal input to this form of synthesis. The only modification typically required to use a pure C/C++ model with the synthesis engine is to add a single special comment to the source code to indicate the top of the functional portion of the design (anything conceptually above this point is considered to form part of the testbench).

As opposed to adding intelligence to the source code (thereby locking it into a target implementation), all of the intelligence is provided by the user controlling and guiding the synthesis engine itself (Figure 11-7).

![Diagram of a pure untimed C/C++-based design flow.](image)

**Figure 11-7. A pure untimed C/C++-based design flow.**

Once the synthesis engine has parsed the source code, the user can use it to perform microarchitecture trade-offs and evaluate their effects in terms of size and speed. The synthesis engine analyzes the code, identifies its various constructs and operators, along with their associated data and memory dependencies, and automatically provides for parallelism wherever possible. The engine also provides a graphical interface that allows the user to specify how different elements should be handled. For example, the interface allows the user to associate ports with registers or RAM blocks; it identifies constructs like loops and allows the user to specify on an
individual basis whether they should be fully unraveled, partially unraveled, or left alone; it allows the user to specify whether or not loops and other constructs should be pipelined; it allows the user to perform resource sharing on specific entities; and so forth.

These evaluations are performed on the fly, and the synthesis engine reports total size/area and latency in terms of clock cycles and I/O delays (or throughput time/cycles in the case of pipelined designs). The user-defined configuration associated with each what-if scenario can be named, saved, and reused as required (it would be almost impossible to perform these trade-offs in a timely manner using a conventional hand-coded RTL-based flow).

The fact that the pure untimed C/C++ source code used by the synthesis engine is not required to contain any implementation intelligence and that all such intelligence is supplied by controlling the engine itself means that the same source code can be easily retargeted to alternative microarchitectures and different implementation technologies.

Once the user’s evaluations are completed, clicking the “Go” button causes the synthesis engine to generate corresponding RTL VHDL. This code can subsequently be used by conventional logic synthesis or physically aware synthesis applications to generate the netlist used to drive the downstream implementation (place-and-route, etc.) tools.

As usual, it would be possible to synthesize the pure untimed C/C++ directly into a gate-level netlist (this alternative is not shown in figure 11-7). However, generating the intermediate RTL provides a comfort zone for the engineers by allowing them to check that they are satisfied with the implementation decisions that have been made during the course of the C/C++ to RTL translation.

Furthermore, generating intermediate RTL is useful because this is the level of abstraction where hardware design engineers generally stitch together the various functional blocks forming their designs. Large portions of today’s designs
are typically presented in the form of IP blocks represented in RTL. This means that the intermediate RTL step shown in figure 11-7 is a useful point in the design flow for integrating and verifying the entire hardware system. The design engineers can then take full advantage of their existing RTL synthesis technology, which is mature, robust, and well understood.

**Different levels of synthesis abstraction**

The fundamental difference between the various C/C++-based flows presented in this chapter is the level of synthesis abstraction each can support. For example, although SystemC offers significant system-level, algorithmic, and transaction-level modeling capabilities, its synthesizable subset is at a relatively low level of abstraction. Similarly, although augmented C/C++ representations are closer to pure C/C++ than are their SystemC counterparts, which means that they simulate much more quickly, their synthesizable subset remains significantly lower than would be ideal.

This lack of synthesis abstraction causes the timed SystemC and augmented C/C++ representations to be implementation specific. In turn, this makes them difficult to create and modify and significantly reduces their flexibility with regard to performing what-if evaluations and retargeting them toward alternative implementation technologies (Figure 11-8).

By comparison, the latest generation of pure untimed C/C++ synthesis technology supports a high level of synthesis abstraction. Non-implementation-specific C/C++ models are very compact and can be quickly and easily created and modified. By means of the synthesis engine itself, the user can quickly and easily perform what-if evaluations and retarget the design toward alternative implementation technologies. The end result is that a pure C/C++-based design flow can dramatically speed implementation and increase design flexibility as compared to other C/C++-based flows.

Before anyone starts to pen irate letters claiming the author is anti-SystemC, it should be reiterated that the discussions presented here are focused on the use of the various flavors of C/C++ in the context of FPGA implementation flows.

In this case, the toolchain used to progress SystemC representations through to actual implementations is relatively immature and unsophisticated.

When it comes to system-level modeling and verification applications, however, SystemC can be extremely efficacious (many observers see SystemC and SystemVerilog being used in conjunction with each other, with SystemC being employed for the initial system-level design representation, and then SystemVerilog being used to “flesh out” the implementation-level details).
Similarly, if one is coming from a software background and is working on embedded software applications and hardware/software co-design and co-verification, then SystemC is considered by many to be "the bees knees" as it were.

![Diagram](image)

**Figure 11-8. Different levels of C/C++ synthesis abstraction.**

**Mixed-language design and verification environments**

Last, but not least, we should note that a number of EDA companies can provide mixed-level design and verification environments that can support the cosimulation of models specified at multiple levels of abstraction.

In some cases, this may simply involve linking a C/C++ model to a Verilog simulator via its *programming language interface (PLI)* or to a VHDL simulator via its *foreign language interface (FLI)*. Alternatively, one might find a SystemC environment with the ability to accept blocks represented in Verilog or VHDL.

And then there are very sophisticated environments that start off with a graphical block-based editor showing the design’s major functional units, where the contents of each block can be represented using the following:

- VHDL
- Verilog
- SystemVerilog
- SystemC
- Handel-C
- Pure C/C++

The top-level design might be in a traditional HDL that calls submodules in the various HDLs and in one or more flavors of C/C++. Alternatively, the top-level design might be in one of the flavors of C/C++ that calls submodules in the various languages.

In this type of environment, the VHDL, Verilog, and SystemVerilog representations are usually handled by a single-kernel simulation engine. This engine is then cosimulated with appropriate engines for the various flavors of C/C++. Furthermore, this type of environment will incorporate source-code debuggers that support the various flavors of C/C++; it will allow testbenches to be created using any of the languages; and supporting tools like graphical waveform displays will be capable of displaying signals and variables associated with any of the language blocks.\(^5\)

In reality, the various mixed-language design and verification environment solution combinations and permutations change on an almost weekly basis, so you need to take a good look at what’s out there before you leap into the fray.

---

\(^5\) A good example of a mixed-language simulation and verification environment of this type that is focused on FPGA—and, to a lesser extent, ASIC—designs is offered by Aldec Inc. (www.aldec.com). Another good example is ModelSim® from Mentor; this includes native SystemC support, thereby allowing single-kernel simulation between VHDL, Verilog, and SystemC.
**Introducing DSP**

The term *digital signal processing*, or DSP, refers to the branch of electronics concerned with the representation and manipulation of signals in digital form. This form of processing includes compression, decompression, modulation, error correction, filtering, and otherwise manipulating audio (voice, music, etc.), video, image, and similar data for such applications as telecommunications, radar, and image processing (including medical imaging).

In many cases, the data to be processed starts out as a signal in the real (analog) world. This analog signal is periodically sampled, with each sample being converted into a digital equivalent by means of an *analog-to-digital* (A/D) converter (Figure 12-1).

![Diagram of analog-to-digital conversion](image)

**Figure 12-1. What is DSP?**

These samples are then processed in the digital domain. In many cases, the processed digital samples are subsequently...
The term CODEC is often bandied around by folks working in the DSP arena. This sometimes stands for COmpressor/DECompressor; that is, something that compresses and decompresses data.

In telecommunications, however, it typically stands for COder/DECoder; that is, something that encodes and decodes a signal.

CODECs can be implemented in software, hardware, or as a mixture of both.

converted into an analog equivalent by means of a digital-to-analog (D/A) converter.

DSP occurs all over the place—in cell phones and telephone systems; CD, DVD, and MP3 players; cable desktop boxes; wireless and medical equipment; electronic vision systems; ... the list goes on. This means that the overall DSP market is huge; in fact, some estimates put it at $10 billion in 2003!

**Alternative DSP implementations**

**Pick a device, any device, but don’t let me see which one**

As usual, nothing is simple because DSP tasks can be implemented in a number of different ways:

- **A general-purpose microprocessor (μP):** This may also be referred to as a central processing unit (CPU) or a microprocessor unit (MPU). The processor can perform DSP by running an appropriate DSP algorithm.

- **A digital signal processor (DSP):** This is a special form of microprocessor chip (or core, as discussed below) that has been designed to perform DSP tasks much faster and more efficiently than can be achieved by means of a general-purpose microprocessor.

- **Dedicated ASIC hardware:** For the purposes of these discussions, we will assume that this refers to a custom hardware implementation that executes the DSP task. However, we should also note that the DSP task could be implemented in software by including a microprocessor or DSP core on the ASIC.

- **Dedicated FPGA hardware:** For the purposes of these discussions, we will assume that this refers to a custom hardware implementation that executes the DSP task. Once again, however, we should also note that the DSP functionality could be implemented in software by means of an embedded microprocessor.
core on the FPGA (at the time of this writing, dedicated DSP hard cores do not exist for FPGAs).

**System-level evaluation and algorithmic verification**

Irrespective of the final implementation technology (µP, DSP, ASIC, FPGA), if one is creating a product that is to be based on a new DSP algorithm, it is common practice to first perform system-level evaluation and algorithmic verification using an appropriate environment (we consider this in more detail later in this chapter).

Although this book attempts to avoid focusing on companies and products as far as possible, it would be rather coy of us not to mention that—at the time of this writing—the de facto industry standard for DSP algorithmic verification is MATLAB\(^1\) from The MathWorks (www.mathworks.com).\(^2\)

For the purposes of these discussions, therefore, we shall refer to MATLAB as necessary. However, it should be noted that there are a number of other very powerful tools and environments available to DSP developers. For example, Simulink\(^8\) from The MathWorks has a certain following; the Signal Processing Worksystem (SPW) environment from CoWare\(^3\) (www.coware.com) is very popular, especially in telecom markets; and tools from Elanix (www.elanix.com) also find favor with many designers.

**Software running on a DSP core**

Let’s assume that our new DSP algorithm is to be implemented using a microprocessor or DSP chip (or core). In this case, the flow might be as shown in Figure 12-2.

---

\(^1\) MATLAB and Simulink are registered trademarks of The MathWorks Inc.

\(^2\) It should be noted that MATLAB and Simulink can be used for a wide range of tasks, including control system design and analysis, image processing, financial modeling, and so forth.

\(^3\) EDA is a fast-moving beast. For example, SPW came under the auspices of Cadence when I first started penning this chapter, but it fell under the purview of CoWare by the time I was half-way through!
1906: America. First radio program of voice and music is broadcast.

The process commences with someone having an idea for a new algorithm or suite of algorithms. This new concept typically undergoes verification using tools such as MATLAB as discussed above. In some cases, one might leap directly from the concept into handcrafting C/C++ (or assembly language).

Once the algorithms have been verified, they have to be regenerated in C/C++ or in assembly language. MATLAB can be used to generate C/C++ tuned for the target DSP core automatically, but in some cases, design teams may prefer to perform this translation step by hand because they feel that they can achieve a more optimal representation this way. As yet another alternative, one might first auto-generate C/C++ code from the algorithmic verification environment, analyze and profile this code to determine any performance bottlenecks, and then recode the most critical portions by hand. (This is a good example of the old 80:20 rule, in which you spend 80 percent of your time working on the most critical 20 percent of the design.)

Once you have your C/C++ (or assembly language) representation, you compile it (or assemble it) into the machine code that will ultimately be executed by the microprocessor or DSP core.

This type of implementation is very flexible because any desired changes can be addressed relatively quickly and easily by simply modifying and recompiling the source code. However, this also results in the slowest performance for the DSP algorithm because microprocessor and DSP chips are both
classed as *Turing machines*. This means that their primary role in life is to process instructions, so both of these devices operate as follows:

- Fetch an instruction.
- Decode the instruction.
- Fetch a piece of data.
- Perform an operation on the data.
- Store the result somewhere.
- ...
- Fetch another instruction and start all over again.

Of course, the DSP algorithm actually runs on hardware in the form of the microprocessor or DSP, but we consider this to be a software implementation because the actual (physical) manifestation of the algorithm is the program that is executed on the chip.

**Dedicated DSP hardware**

There are myriad ways in which one might implement a DSP algorithm in an ASIC or FPGA—the latter option being the focus of this chapter, of course. But before we hurl ourselves into the mire, let’s first consider how different architectures can affect the speed and area (in terms of silicon real estate) of the implementation.

DSP algorithms typically require huge numbers of multiplications and additions. As a really simple example, let’s assume that we have a new DSP algorithm that contains an expression something like the following:

\[ Y = (A \times B) + (C \times D) + (E \times F) + (G \times H); \]

As usual, this is a generic syntax that does not favor any particular HDL and is used only for the purposes of these discussions. Of course, this would be a miniscule element in a horrendously complex algorithm. But, at the end of the day, DSP algorithms tend to contain a lot of this type of thing.
The point is that we can exploit the parallelism inherent in hardware to perform DSP functions much more quickly than can be achieved by means of software running on a DSP core. For example, suppose that all of the multiplications were performed in parallel (simultaneously) followed by two stages of additions (Figure 12-3).

![Diagram of parallel implementation of a function]

**Figure 12-3. A parallel implementation of the function.**

Remembering that multipliers are relatively large and complex and that adders are sort of large, this implementation will be very fast, but will consume a correspondingly large amount of chip resources.

As an alternative, we might employ resource sharing (sharing some of the multipliers and adders between multiple operations) and opt for a solution that is a mixture of parallel and serial (Figure 12-4).

This solution requires the addition of four 2:1 multiplexers and a register (remember that each of these will be the same multibit width as their respective signal paths). However, multiplexers and registers consume much less area than the
two multipliers and adder that are no longer required as compared to our initial solution.

On the downside, this approach is slower, because we must first perform the \((A \times B)\) and \((C \times D)\) multiplications, add the results together, add this total to the existing contents of the register (which will have been initialized to contain zero), and store the result in the register. Next, we must perform the \((E \times F)\) and \((G \times H)\) multiplications, add these results together, add this total to the existing contents of the register (which currently contains the results from the first set of multiplications and additions), and store this result in the register.

As yet another alternative, we might decide to use a fully serial solution (Figure 12-5).

This latter implementation is very efficient in terms of area because it requires only a single multiplier and a single adder. This is the slowest implementation, however, because we must first perform the \((A \times B)\) multiplication, add the result to the existing contents of the register (which will have been initial-
Figure 12-5. A serial implementation of the function.

ized to contain zero), and store the total in the register. Next, we must perform the \((C \times D)\) multiplication, add this result to the existing contents of the register, and store this new total in the register. And so forth for the remaining multiplication operations. (Note that when we say “this is the slowest imple-
mentation,” we are referring to these hardware solutions, but even the slowest hardware implementation remains much, much faster than a software equivalent running on a micro-
processor or DSP.)

**DSP-related embedded FPGA resources**

As previously discussed in chapter 4, some functions like multipliers are inherently slow if they are implemented by connecting a large number of programmable logic blocks together inside an FPGA. Because these functions are required by a lot of applications, many FPGAs incorporate special hard-wired multiplier blocks. (These are typically located in close proximity to embedded RAM blocks because these functions are often used in conjunction with each other.)

Similarly, some FPGAs offer dedicated adder blocks. One operation that is very common in DSP-type applications is called a multiply-and-accumulate. As its name would suggest,
this function multiplies two numbers together and adds the result into a running total stored in an accumulator (register). Hence, it is commonly referred to as a MAC, which stands for multiply, add, and accumulate (Figure 12-6).

![Diagram of a MAC](image)

Figure 12-6. The functions forming a MAC.

Note that the multiplier, adder, and register portions of the serial implementation of our function shown in figure 12-5 offer a classic example of a MAC. If the FPGA you are working with supplies only embedded multipliers, you would be obliged to implement this function by combining the multiplier with an adder formed from a number of programmable logic blocks, while the result would be stored in a block RAM or in a number of distributed RAMs. Life becomes a little easier if the FPGA also provides embedded adders, and some FPGAs provide entire MACs as embedded functions.

**FPGA-centric design flows for DSPs**

Arrggh! I'm quivering with fear (but let's call it anticipation) as I'm poised to pen these words. This is because, at the time of this writing, the idea of using FPGAs to perform DSP is still relatively new. Thus, there really are no definitive design flows or methodologies here—everyone seems to have
his or her unique way of doing things, and whichever option you choose, you’ll almost certainly end up breaking new ground one way or another.

**Domain-specific languages**

The way of the world is that electronic designs increase in size and complexity over time. In order to manage this problem while maintaining—or, more usually, increasing—productivity, it is necessary to keep raising the level of abstraction used to capture the design’s functionality and verify its intent.

For this reason the gate-level schematics discussed in chapter 8 were superceded by the RTL representations in VHDL and Verilog discussed in chapter 9. Similarly, the drive toward C-based flows as discussed in chapter 11 is powered by the desire to capture complex concepts quickly and easily while facilitating architectural analysis and exploration.

In the case of specialist areas such as DSPs, system architects and design engineers can achieve a dramatic improvement in productivity by means of *domain-specific languages* (DSLs), which provide more concise ways of representing specific tasks than do general-purpose languages such as C/C++ and SystemC.

One such language is MATLAB, which allows DSP designers to represent a signal transformation, such as an FFT, that can potentially take up an entire FPGA, using a single line of code4 along the lines of

\[ y = \text{fft}(x); \]

Actually, the term MATLAB refers both to a language and an algorithmic-level simulation environment. In order to avoid confusion, it is common to talk about M-code (meaning "MATLAB code") and M-files (files containing MATLAB code). Some engineers in the trenches occasionally refer to

---

4 Note that the semicolon shown in this example MATLAB statement is optional. If present, it serves to suppress the output display.
the "M language," but this is not argot favored by the folks at The MathWorks.

In addition to sophisticated transformation operators like the FFT shown above, there are also much simpler transformations like adders, subtractors, multipliers, logical operators, matrix arithmetic, and so forth. The more complex transformations like an FFT can be formed from these fundamental entities if required. The output from each transformation can be used as the input to one or more downstream transformations, and so forth, until the entire system has been represented at this high level of abstraction.

One important point is that such a system-level representation does not initially imply a hardware or software implementation. In the case of DSP core, for example, it could be that the entire function is implemented in software as discussed earlier in this chapter. Alternatively, the system architects could partition the design such that some functions are implemented in software, while other performance-critical tasks are implemented in hardware using dedicated ASIC or FPGA fabric. In this case, one typically needs to have access to a hardware or software codesign environment (see also chapter 13). For the purposes of these discussions, however, we shall assume pure hardware implementations.

**System-level design and simulation environments**

System-level design and simulation environments are conceptually at a higher level than DSLs. One well-known example of this genre is Simulink from The MathWorks. Depending on whom one is talking to, there may be a perception that Simulink is simply a graphical user interface to MATLAB. In reality, however, it is an independent dynamic modeling application that works with MATLAB.

If you are using Simulink, you typically commence the design process by creating a graphical block diagram of your system showing a schematic of functional blocks and the connections between them. Each of these blocks may be user-
defined, or they may originate in one of the libraries supplied with Simulink (these include DSP, communications, and control function block sets). In the case of a user-defined block, you can “push” into that block and represent its contents as a new graphical block diagram. You can also create blocks containing MATLAB functions, M-code, C/C++, FORTRAN ... the list goes on.

Once you’ve captured the design’s intent, you use Simulink to simulate and verify its functionality. As with MATLAB, the input stimulus to a Simulink simulation might come from one or more mathematical functions, such as sine-wave generators, or it might be provided in the form of real-world data such as audio or video files. In many cases, it comes as a mixture of both; for example, real-world data might be augmented with pseudorandom noise supplied by a Simulink block.

The point here is that there’s no hard-and-fast rule. Some DSP designers prefer to use MATLAB as their starting point, while others opt for Simulink (this latter case is much rarer in the scheme of things). Some folks say that this preference depends on the user’s background (software DSP development versus ASIC/FPGA designs), but others say that this is a load of tosh. And it really doesn’t matter, because, if the truth is told, the reasons behind who does what in this regard pale into insignificance compared to the horrors that are to come.

**Floating-point versus fixed-point representations**

Irrespective as to whether one opts for Simulink or MATLAB (or a similar environment from another vendor) as a starting point, the first-pass model of the system is almost invariably described using floating-point representations. In the context of the decimal number system, this refers to numbers like $1.235 \times 10^3$ (that is, a fractional number raised to some power of 10). In the context of applications like MATLAB, equivalent binary values are represented inside the computer using the IEEE standard for double-precision floating-point numbers.
Floating-point numbers of this type have the advantage of providing extremely accurate values across a tremendous dynamic range. However, implementing floating-point calculations of this type in dedicated FPGA or ASIC hardware requires a humongous amount of silicon resources, and the result is painfully slow (in hardware terms). Thus, at some stage, the design will be migrated over to use fixed-point representations, which refers to numbers having a fixed number of bits to represent their integer and fractional portions. This process is commonly referred to as quantization.

This is totally system/algorithmd dependent, and it may take some considerable amount of experimentation to determine the optimum balance between using the fewest number of bits to represent a set of values (thereby decreasing the amount of silicon resources required and speeding the calculations), while maintaining sufficient accuracy to perform the task in hand. (One can think of this trade-off in terms of how much noise the designer is willing to accept for a given number of bits.) In some cases, designers may spend days deciding “should we use 14, 15, or 16 bits to represent these particular values?” And, just to increase the fun, it may be best to vary the number of bits used to represent values at different locations in the system/algorithm.

Things start to get really fun in that the conversion from floating-point to fixed-point representations may take place upstream in the system/algorithmic design and verification environment, or downstream in the C/C++ code. This is shown in more detail in the “System/algorithmic level to C/C++” section below. Suffice it to say that if one is working in a MATLAB environment, these conversions can be performed by passing the floating-point signals through special transformation functions called quantizers. Alternatively, if one is working in a Simulink environment, the conversions can be performed by running the floating-point signals through special fixed-point blocks.
System/algorithmic level to RTL (manual translation)

At the time of this writing, many DSP design teams commence by performing their system-level evaluations and algorithmic validation in MATLAB (or the equivalent) using floating-point representations. (It is also very common to include an intermediate step in which a fixed-point C/C++ model is created for use in rapid simulation/validation.) At this point, many design teams bounce directly into hand-coding fixed-point RTL equivalents of the design in VHDL or Verilog (figure 12-7a). Alternatively, they may first transition the floating-point representations into their fixed-point counterparts at the system/algorithmic level, and then hand-code the RTL in VHDL or Verilog (Figure 12-7b).

![Diagram]

Figure 12-7. Manual RTL generation.

There are, of course, a number of problems with this flow, not the least being that there is a significant conceptual and representational divide between the system architects working at the system/algorithmic level and the hardware design engineers working with RTL representations in VHDL or Verilog.
Because the system/algorithmic and RTL domains are so different, manual translation from one to the other is time-consuming and prone to error. There is also the fact that the resulting RTL is implementation specific because realizing the optimal design in an FPGA requires a different RTL coding style from that used for an optimal ASIC implementation.

Another consideration is that manually modifying and reverifying RTL to perform a series of what-if evaluations of alternative microarchitecture implementations is extremely time-consuming (such evaluations may include performing certain operations in parallel versus sequential, pipelining portions of the design versus nonpipelining, sharing common resources—for example, two operations sharing a single multiplier—versus using dedicated resources, etc.)

Similarly, if any changes are made to the original specification during the course of the project, it’s relatively easy to implement and evaluate these changes in the system-/algorithmic-level representations, but subsequently folding these changes into the RTL by hand can be painful and time-consuming.

Of course, once an RTL representation of the design has been created, we can assume the use of the downstream logic-synthesis-based flows that were introduced in chapter 9.

**System/algorithmic level to RTL (automatic-generation)**

As was noted in the previous section, performing system-/algorithmic-level-to-RTL translation manually is time-consuming and prone to error. There are alternatives, however, because some system-/algorithmic-level design environments offer direct VHDL or Verilog RTL code generation (Figure 12-8).

As usual, the system-/algorithmic-level design would commence by using floating-point representations. In one version of the flow, the system/algorithmic environment is used to migrate these representations into their fixed-point counter-
1909:
Leo Baekeland patterns an artificial plastic that he calls Bakelite.

![Diagram showing the flow of design and verification processes.](image)

**Figure 12-8. Direct RTL generation.**

parts and then to generate the equivalent RTL in VHDL or Verilog automatically (Figure 12-8a)\(^5\).

Alternatively, a third-party environment might be used to take the floating-point system/algorithmic-level representation, autointeractively quantize it into its fixed-point counterpart, and then automatically generate the equivalent RTL in VHDL or Verilog (figure 12-8b)\(^6\).

As before, once an RTL representation of the design has been created, we can assume the use of the downstream logic-synthesis-based flows that were introduced in chapter 9.

---

\(^5\) A good example of this type of environment is offered by Elanix Inc. (www.elanix.com).

\(^6\) An example of this type of environment is offered by AccellChip Inc. (www.accelchip.com), whose environment can accept floating-point MATLAB M-files, output their fixed-point equivalents for verification, and then use these new M-files to auto-generate RTL.
**System/algorithmic level to C/C++ etc.**

Due to the problems associated with exploring the design at the RTL level, there is an increasing trend to use a stepping-stone approach. This involves transitioning from the system/algorithmic-level domain into some sort of C/C++ representation, which itself is subsequently migrated into an RTL equivalent. One reason this is attractive is that the majority of DSP design teams already generate a C/C++ model for use as a golden (reference) model, in which case this sort of comes for free as far as the downstream RTL design engineer is concerned.

Of course, the first thing to decide is when and where in the flow one should transition from floating-point to fixed-point representations (Figure 12-9).

![Diagram showing the transition from system/algorithmic level to C/C++ and then to fixed-point representation.]

**Figure 12-9. Migrating from floating point to fixed point.**

Frighteningly enough, Figure 12-9 shows only a subset of the various potential flows. For example, in the case of the handcrafted options, as opposed to first hand-coding the
C/C++ and then gradually transmogrifying this representation into Handel-C or SystemC, one could hand-code directly into these languages.

However, the main thing to remember is that once we have a fixed-point representation in one of the flavors of C/C++, we can assume the use of the downstream C/C++ flows introduced in chapter 11 (one flow of particular interest in this area is the pure untimed C/C++ approach used by Precision C from Mentor).

**Block-level IP environments**

Nothing is simple in this world because there is always just one more way to do things. As an example, one might create a library of DSP functional blocks at the system/algorithmic level of abstraction along with a one-to-one equivalent library of blocks at the RTL level of abstraction in VHDL or Verilog.

The idea here is that you could then capture and verify your design using a hierarchy of functional blocks specified at the system/algorithmic level of abstraction. Once you were happy with your design, you could then generate a structural netlist instantiating the RTL-level blocks, and use this to drive downstream simulation and synthesis tools. (These blocks would have to be parameterized at all levels of abstraction so as to allow you to specify such things as bus widths and so forth.)

As an alternative, the larger FPGA vendors typically offer IP core generators (in this context, the term core is considered to refer to a block that performs a specific logical function; it does not refer to a microprocessor or DSP core). In several cases, these core generators have been integrated into system-/algorithmic-level environments. This means that you can create a design based on a collection of these blocks in the system-/algorithmic-level environment, specify any parameters associated with these blocks, and perform your system-/algorithmic-level verification.
Later, when you’re ready to rock and roll, the core generator will automatically generate the hardware models corresponding to each of these blocks.⁷ (The system-/algorithmic-level models and the hardware models ensuing from the core generator are bit identical and cycle identical.) In some cases the hardware blocks will be generated as synthesizeable RTL in VHDL or Verilog. Alternatively, they may be presented as firm cores at the LUT/CLB level of abstraction, thereby making the maximum use of the targeted FPGA’s internal resources.

One big drawback associated with this approach is that, by their very nature, IP blocks are based on hard-coded micro-architectures. This means that the ability to create highly tuned implementations to address specific design goals is somewhat diminished. The end result is that IP-based flows may achieve an implementation faster with less risk, but such an implementation may be less optimal in terms of area, performance, and power as compared to a custom hardware implementation.

**Don’t forget the testbench!**

One point that the folks selling you DSP design tools often neglect to mention is the test bench. For example, let’s assume that your flow involves taking your system-/algorithmic-level design and hand-translating it into RTL. In that case, you are going to have to do the same with your testbench. In many cases, this is a nontrivial task that can take days or weeks!

Or let’s say that your flow is based on taking your floating-point system-/algorithmic-level design and hand-translating it into floating-point C/C++, at which point you will wish to verify this new representation. Then you might take your floating-point C/C++ and hand-translate it into fixed-point C/C++, at which point you will wish to verify this representation. And then you might take your fixed-point C/C++ and

---

⁷ A good example of this type of approach is the integration of Simulink with the System Generator utility from Xilinx (www.xilinx.com).
Radio distress signals save 1900 lives after two ships collide.

(hopefully) automatically synthesize an equivalent RTL representation, at which point ... but you get my drift.

The problem is that at each stage you are going to have to do the same thing with your testbench (unless you do something cunning as discussed in the next (and last—hurray!) section.

**Mixed DSP and VHDL/Verilog etc. environments**

In the previous chapter, we noted that a number of EDA companies can provide mixed-level design and verification environments that can support the cosimulation of models specified at multiple levels of abstraction. For example, one might start off with a graphical block-based editor showing the design’s major functional units, where the contents of each block can be represented using

- VHDL
- Verilog
- SystemVerilog
- SystemC
- Handel-C
- Pure C/C++

In this case, the top-level design might be in a traditional HDL that calls submodules represented in the various HDLs and in one or more flavors of C/C++. Alternatively, the top-level design might be in one of the flavors of C/C++ that calls submodules in the other languages.

More recently, integrations between system-/algorithmic-level and implementation-level environments have become available. The way in which this works depends on who is doing what and what that person is trying to do (sorry, I don’t...

---

8 Don’t laugh, because I personally know of one HUGE system house that does things in just this way!

9 With regards to the C/C++ to RTL stage of the process, even if you have a C/C++ to RTL synthesis engine, your testbench will typically contain language constructs are aren’t amenable to synthesis, which means that you’re back to doing things by hand.
mean to be cryptic). For example, a system architect working at the system/algorithmic level (e.g., in MATLAB) might decide to replace one or more blocks with equivalent representations in VHDL or Verilog at the RTL level of abstraction. Alternatively, a design engineer working in VHDL or Verilog at the RTL level of abstraction might decide to call one or more blocks at the system/algorithmic level of abstraction.

Both of these cases require cosimulation between the system-/algorithmic-level environment and the VHDL/Verilog environment, the main difference being who calls whom. Of course, this sounds easy if you say it quickly, but there is a whole host of considerations to be addressed, such as synchronizing the concept of time between the two domains and specifying how different signal types are translated as they pass from one domain to the other (and back again).

This really is a case of treating any canned demonstration with a healthy amount of suspicion. If you are planning on doing this sort of thing, you need to sit down with the vendor’s engineer and work your own example through from beginning to end. Call me an old cynic if you will, but my advice is to let their engineer guide you, while keeping your hands firmly on the keyboard and mouse. (You’d be amazed how much activity can go on in just a few seconds should you turn your head in response to the age-old question, “Good grief! Did you see what just flew by the window?”)
**Introduction**

For the purposes of this book, we are concerned only with electronic systems that include one or more FPGAs on the **printed circuit board (PCB)**. The vast majority of such systems also make use of a general-purpose microprocessor, or µP, to perform a variety of control and data-processing applications.\(^1\) This is often referred to as the **central processing unit (CPU)** or **microprocessor unit (MPU)**.

Until recently, the CPU and its peripherals typically appeared in the form of discrete chips on the circuit board. There are an almost infinite number of possible scenarios here, but the two main ones involve the way in which the CPU is connected to its memory (Figure 13-1).

---

\(^1\) Alternatively, one might use a microcontroller (µC) device, which combines a CPU core with selected peripherals and specialized inputs and outputs.

---

**Figure 13-1. Two scenarios at the circuit board level.**
In both of these scenarios, the CPU is connected to an FPGA and some other stuff via a general-purpose processor bus. (By “stuff” we predominantly mean peripheral devices such as counter timers, interrupt controllers, communications devices, etc.)

In some cases, the main memory (MEM) will also be connected to the CPU by means of the main processor bus, as shown in figure 13-1a (actually, this connection will be via a special peripheral called a memory controller, which is not shown here because we’re trying to keep things simple). Alternatively, the memory may be connected directly to the CPU by means of a dedicated memory bus, as shown in Figure 13-1b).

The point is that presenting the CPU and its various peripheral devices in the form of dedicated chips on the circuit board costs money and occupies real estate. It also impacts the reliability of the board because every solder joint (connection point) is a potential failure mechanism.

One alternative is to embed the CPU along with some of its peripherals in the FPGA itself (Figure 13-2).²

![Figure 13-2. Two scenarios at the FPGA level.](image)

² Another alternative would be to embed a microprocessor core in an ASIC, but that’s a tale for another book!
It is common for a relatively small amount of memory used by the CPU to be included locally in the FPGA. At the time of this writing, however, it is rare for all of the CPU's memory to be included in the FPGA.

Creating an FPGA design of this type brings a whole slew of new problems to the table. First of all, the system architects have to decide which functions will be implemented in software (as instructions to be executed by the CPU) and which functions will be implemented in hardware (using the main FPGA fabric). Next, the design environment must support the concept of coverification, in which the hardware and embedded software portions of the system can be verified together to ensure that everything works as it should. Both of these topics are considered in more detail later in this chapter.

**Hard versus soft cores**

**Hard cores**

A hard microprocessor core is one that is implemented as a dedicated, predefined (hardwired) block (these cores are only available in certain device families). Each of the main FPGA vendors has opted for a particular processor type to implement its hard cores. For example, Altera offer embedded ARM processors, QuickLogic have opted for MIPS-based solutions, and Xilinx sports PowerPC cores.

Of course, each vendor will be delighted to explain at great length why its implementation is far superior to any of the others (the problem of deciding which one actually is better is only compounded by the fact that different processors may be better suited to different tasks).

As noted in chapter 4, there are two main approaches for integrating such cores into the FPGA. The first is to locate it in a strip to the side of the main FPGA fabric (Figure 13-3).

In this scenario, all of the components are typically formed on the same silicon chip, although they could also be formed on two chips and packaged as a multiple chip module (MCM).
One advantage of this implementation is that the main FPGA fabric is identical for devices with and without the embedded microprocessor core, which can make things easier for the design tools used by the engineers. The other advantage is that the FPGA vendor can bundle a whole load of additional functions in the strip to complement the microprocessor core, such as memory and special peripherals.\(^3\)

An alternative is to embed one or more microprocessor cores directly into the main FPGA fabric. One, two, and even four core implementations are currently available at the time of this writing (Figure 13-4).

In this case, the design tools have to be able to take account of the presence of these blocks in the fabric; any memory used by the core is formed from embedded RAM blocks, and any peripheral functions are formed from groups of general-purpose programmable logic blocks. Proponents of this scheme can argue that there are inherent speed advan-

\(^3\) This approach is favored by vendors such as Altera (www.altera.com) and QuickLogic (www.quicklogic.com).
Figure 13-4. Bird’s-eye view of chips with embedded cores inside the main fabric.

tages to be gained from having the microprocessor core in intimate proximity to the main FPGA fabric.\(^4\)

**Soft microprocessor cores**

As opposed to embedding a microprocessor physically into the fabric of the chip, it is possible to configure a group of programmable logic blocks to act as a microprocessor. These are typically called “soft cores,” but they may be more precisely categorized as either soft or firm, depending on the way in which the microprocessor's functionality is mapped onto the logic blocks. For example, if the core is provided in the form of an RTL netlist that will be synthesized with the other logic, then this truly is a soft implementation. Alternatively, if the core is presented in the form of a placed and routed block of LUTs/CLBs, then this would typically be considered a firm implementation.

In both of these cases, all of the peripheral devices like counter timers, interrupt controllers, memory controllers, communications functions, and so forth are also implemented as

---

\(^4\) This approach is favored by Xilinx (www.xilinx.com), who also provide a multitude of peripherals in the form of soft IP cores.
The Nios is based on a SPARC architecture using the concept of register windows, while the MicroBlaze is based on a classical RISC architecture.

IDE is pronounced by spelling it out as “I-D-E.”

Depending on who you are talking to and the FPGA or RTOS vendor in question, the ‘D’ in IDE can stand for “design” or “development.”

QuickLogic offer a 9-bit soft microcontroller that goes under the catchy name of Q90C1xx. (Having a 9-bit data word can be useful for certain communication functions.)

soft or firm cores (the FPGA vendors are typically able to supply a large library of such cores).

Soft cores are slower and simpler than their hard-core counterparts (of course they are still incredibly fast in human terms). However, in addition to being practically free, they also have the advantages that you only have to implement a core if you need it and that you can instantiate as many cores as you require until you run out of resources in the form of programmable logic blocks.

Once again, each of the main FPGA vendors has opted for a particular processor type to implement its soft cores. For example, Altera offers the Nios, while Xilinx sports the MicroBlaze. The Nios has both 16-bit and 32-bit architectural variants, which operate on 16-bit or 32-bit chunks of data, respectively (both variants share the same 16-bit-wide instruction set). By comparison, the MicroBlaze is a true 32-bit machine (that is, it has 32-bit-wide instruction words and performs its magic on 32-bit chunks of data). Once again, each of the vendors will be more than happy to tell you why its soft core rules and how its competitors’ offerings fail to make the grade (sorry, you’re on your own here).

One cool thing about the integrated development environment (IDE) fielded by Xilinx is that it treats the PowerPC hard core and the MicroBlaze soft core identically. This includes both processors being based on the same CoreConnect processor bus and sharing common soft peripheral IP cores. All of this makes it relatively easy to migrate from one processor to the other.

Also of interest is the fact that Xilinx offers a small 8-bit soft core called the PicoBlaze, which can be implemented using only 150 logic cells (give or take a handful). By comparison, the MicroBlaze requires around 1,000 logic cells.\(^5\)

\(^5\) For the purposes of these discussions, a logic cell can be assumed to contain a 4-input LUT, a register element, and various other bits and pieces like multiplexers and fast carry logic.
Embedded Processor-Based Design Flows ■ 245

(which is still extremely reasonable for a 32-bit processor implementation, especially when one is playing with FPGAs that can contain 70,000 or more such cells.)

Partitioning a design into its hardware and software components

As noted in chapter 4, almost any portion of an electronic design can be realized in hardware (using logic gates and registers, etc.) or software (as instructions to be executed on a microprocessor). One of the main partitioning criteria is how fast you wish the various functions to perform their tasks:

- **Picosecond and nanosecond logic**: This has to run insanely fast, which mandates that it be implemented in hardware (in the FPGA fabric).
- **Microsecond logic**: This is reasonably fast and can be implemented either in hardware or software (this type of logic is where you spend the bulk of your time deciding which way to go).
- **Millisecond logic**: This is the logic used to implement interfaces such as reading switch positions and flashing light-emitting diodes, or LEDs. It's a pain slowing the hardware down to implement this sort of function (using huge counters to generate delays, for example). Thus, it's often better to implement these tasks as microprocessor code (because processors give you lousy speed—compared to dedicated hardware—but fantastic complexity).

The trick is to solve every problem in the most cost-effective way. Certain functions belong in hardware, others cry out for a software realization, and some functions can go either way depending on how you feel you can best use the resources

---

6 This 70,000 value was true when I ate my breakfast this morning, but it will doubtless have increased by the time you come to read this book.
(both chip-level resources and hardware/software engineers) available to you.

It is possible to envisage an “ideal” electronic system level (ESL) environment in which the system architects initially capture the design via a graphical interface as a collection of functional blocks that are connected together. Each of these blocks could then be provided with a system-/algorithmic-level SystemC representation, for example, and the entire design could be verified prior to any decisions being made as to which portions of the design were to be implemented in hardware and software.

When it comes to the partitioning process itself, we might dream of having the ability to tag each graphical block with the mouse and select a hardware or software option for its implementation. All we would then have to do would be to click the “Go” button, and the environment would take care of synthesizing the hardware, compiling the software, and pulling everything together.

And then we return to the real world with a resounding thud. Actually, a number of next-generation design environments show promise, and new tools and techniques are arriving on an almost daily basis. At the time of this writing, however, it is still very common for system architects to partition a design into its hardware and software portions by hand, and to then pass these top-level functions over to the appropriate engineers and hope for the best.

With regard to the software portion of the design, this might be something as simple as a state machine used to control a human-level interface (reading the state of switches and controlling display devices). Although the state machine itself may be quite tricky, this level of software is certainly not rocket science. At the other end of the spectrum, one might have incredibly complex software requirements, including

- System initialization routines and a hardware abstraction layer
- A hardware diagnostic test suite

RTOS is pronounced by spelling it out as “R-T-O-S.”

Real-time systems are those in which the correctness of a computation or action depends not only on how it is performed but also when it is performed.
- A real-time operating system (RTOS)
- RTOS device drivers
- Any embedded application code

This code will typically be captured in C/C++ and then compiled down to the machine instructions that will be run on the processor core (in extreme cases where one is attempting to squeeze the last drop of performance out of the design, certain routines may be handcrafted in assembly code).

At the same time, the hardware design engineers will typically be capturing their portions of the design at the RTL level of abstraction using VHDL or Verilog (or SystemVerilog).

Today's designs are so complex that their hardware and software portions have to be verified together. Unfortunately, wrapping one's brain around the plethora of verification alternatives and intricacies can make a grown man (well, me actually) break down and weep.

**Hardware versus software views of the world**

One of the biggest problems to overcome when it comes to the verification of the hardware and software portions of a design is the two totally different worldviews of their creators.

The hardware folks typically visualize their portion of the design as blocks of RTL representing such things as registers, logical functions, and the wires connecting them together. When hardware engineers are debugging their portion of the design, they think in terms of an editor showing their RTL source code, a logic simulator, and a graphical waveform display showing signals changing values at specific times. In a typical hardware design environment, clicking on a particular event in the waveform display will automatically locate the corresponding line of RTL code that caused this event to occur.

By comparison, the software guys and gals think in terms of C/C++ source code, of registers in the CPU (and in the
peripherals), and of the contents of various memory locations. When software engineers are debugging a program, they often wish to single-step through the code one line at a time and watch the values in the various registers changing. Or they might wish to set one or more breakpoints (this refers to placing markers at specific points in the code), run the program until they hit one of those breakpoints, and then pause to see what’s going on. Alternatively, they might wish to specify certain conditions such as a register containing a particular value, then run the program until this condition is met, and once again pause to see what’s happening.

When a software developer is writing application code such as a game, he or she has the luxury of being reasonably confident that the hardware (say, a home computer) is reasonably robust and bug-free. However, it’s a different ball game when one is talking about a software engineer creating embedded applications intended to run on hardware that’s being designed at the same time. When a problem occurs, it can be mega tricky determining if it was a fault in the software or if the hardware was to blame. The classic joke is a conversation between the two camps:

**Software Engineer:** “I think I may have hit a hardware problem while running my embedded application.”

**Hardware Engineer:** “At what time did the error occur? Can you give me a test case that isolates the problem?”

**Software Engineer:** “The error occurred at 9:30 this morning, and the test case is my application!”

In the case of today’s state-of-the-art verification environments, the hardware and software worlds are tightly coupled. This means that if the software engineers detect a potential hardware bug, identifying the particular line of code being executed will take the hardware engineers directly to the corresponding simulation time frame in the graphical waveform display. Similarly, if the hardware engineers detect a potential software bug (such as code requesting an illegal
hardware transaction), they can use their interface to guide the software team to the corresponding line of source code. Unfortunately, this type of environment can cost a lot of money, so sometimes you have to opt for a less sophisticated solution.

**Using an FPGA as its own development environment**

Perhaps the simplest place to start is the scenario where the FPGA is used as its own development environment. The idea here is that you have an SRAM-based FPGA with an embedded processor (hard or soft) mounted on a development board that’s connected to your computer. In addition to the FPGA, this development board will also have a memory device that will be used to store the software programs that are to be run by the embedded CPU (figure 13-5).

![Figure 13-5. Using an FPGA as its own development environment.](image)

Once the system architects have determined which portions of the design are to be implemented in hardware and software, the hardware engineers start to capture their RTL blocks and functions and synthesize them down to a LUT/CLB-level netlist. Meanwhile, the software engineers start to capture their C/C++ programs and routines and compile them down to machine code. Eventually, the LUT/CLB-level netlist will be loaded into the FPGA via a configuration file, the linked machine code image will be loaded into the memory device, and then you let the system run wild and free (Figure 13-6).
1912:
The *Titanic* sends out radio distress signals when it collides with an iceberg and sinks on its maiden voyage.

Also, any of the machine code that is to be embedded in the FPGA’s on-chip RAM blocks would actually be loaded via the configuration file.

**Improving visibility in the design**

The main problem with the scenario discussed in the previous section is lack of “visibility” as to what is happening in the hardware portion of the design. One way to mitigate this is to use a virtual logic analyzer to observe what’s happening in the hardware (this is discussed in more detail in Chapter 16).

Things can be a little trickier when it comes to determining what’s happening with the software. One point to
remember is that—as discussed in chapter 5—an embedded CPU core will have its own dedicated JTAG boundary scan chain (Figure 13-7).

![JTAG diagram]

**Figure 13-7. Embedded processor JTAG boundary scan chain.**

This is true of both hard cores and the more sophisticated soft cores. In this case, the verification environment can use the scan chain to monitor the activity on the buses and control signals connecting the CPU to the rest of the system. The CPU’s internal registers can also be accessed via the JTAG port, thereby allowing an external debugger to take control of the device and single-step through instructions, set breakpoints, and so forth.

**A few verification alternatives**

If you really want to get visibility into what’s happening in the hardware portions of design, one approach is to use a logic simulator. In this case, the majority of the system will be modeled and simulated in VHDL or Verilog/SystemVerilog at the RTL level of abstraction. When it comes to the CPU core, however, there are various ways in which to represent this (Figure 13-8).
Irrespective of the type of model used to represent the CPU, the embedded software (machine code) portion of the design will be loaded into some form of memory—either embedded memory in the FPGA or external memory devices—and the CPU model will then execute those machine code instructions.

Note that figure 13-8 shows a high-level representation of the contents of the FPGA only. If the machine code is to be stored in external memory devices, then these devices would also have to be part of the simulation. In fact, as a general rule of thumb, if the software talks to any stuff, then that stuff needs to be part of the verification environment.

**RTL (VHDL or Verilog)**

Perhaps the simplest option here is when one has an RTL model of the CPU, in which case all of the activity takes place in the logic simulator. One disadvantage of this approach is that a CPU performs tremendous numbers of internal operations in order to perform the simplest task, which equates to incredibly slow simulation runs (you’ll be lucky to be able to simulate 10 to 20 system clocks per second in real time).

The other disadvantage is that you have no visibility into what the software is doing at the source code level. All you’ll
be able to do is to observe logic values changing on wires and inside registers.

And there's always the fact that whoever supplies the real CPU doesn't want you to know how it works internally because that supplier may be using cunning proprietary tricks and wish to preserve their IP. In this case, you may well find it very difficult to lay your hands on an RTL model of the CPU at all.

**C/C++, SystemC, etc.**

As opposed to using an RTL model, it is very common to have access to some sort of C/C++ model of the CPU. (The proponents of SystemC have a vision of a world in which the CPU and the main peripheral devices all have SystemC models provided as standard for use in this type of design environment.)

The compiled version of this CPU model would be linked into the simulation via the *programming language interface (PLI)* in the case of a Verilog simulator or the *foreign language interface (FLI)*—or equivalent—in the case of a VHDL simulator.

The advantages of such a model are that it will run much faster than its RTL counterpart; that it can be delivered in compiled form, thereby preserving any secret IP; and that, at least in FPGA circles, such a model is usually provided for free (the FPGA vendors are trying to sell chips, not models).

One disadvantage of this approach is that the C/C++ model may not provide a 100 percent cycle-accurate representation of the CPU, which has the potential to cause problems if you aren't careful. But, once again, the main disadvantage of such a model is that its only purpose is to provide an engine to execute the machine code program, which means that you have no visibility into what the software is doing at the source code level. All you'll be able to do is observe logic values changing on wires and inside registers.

Way back in the mists of time, the Logic Modeling Corporation (LMC)—which was subsequently acquired by Synopsys—defined an interface for connecting behavioral models of hardware blocks to logic simulators. This is known as the SWIFT interface, and models—such as CPUs—that comply with this specification may be referred to as SWIFT models.
**Physical chip in hardware modeler**

Yet another possibility is to use a physical device to represent a hard CPU core. For example, if you are using a PowerPC core in a Xilinx FPGA, you can easily lay your hands on a real PowerPC chip. This chip can be installed in a box called a hardware modeler, which can then be linked into the logic simulation system.

The advantage of this approach is that you know the physical model (chip) is going to functionally match your hard core as closely as possible. Some disadvantages are that hardware modelers aren’t cheap and they can be a pain to use.

The majority of hardware-modeler-based solutions don’t support source-level debugging, which, once again, means that you have no visibility into what the software is doing at the source code level. All you’ll be able to do is to observe logic values changing on wires and inside registers.

**Instruction set simulator**

As previously noted, in certain cases, the role of the software portion of a design may be somewhat limited. For example, the software may be acting as a state machine used to control some interface. Alternatively, the software’s role may be to initialize certain aspects of the hardware and then sit back and watch the hardware do all of the work. If this is the case, then a C/C++ model or a physical model is probably sufficient—at least as far as the hardware design engineer is concerned.

At the other extreme, the hardware portions of the design may exist mainly to act as an interface with the outside world. For example, the hardware may read in a packet of data and store it in the FPGA’s memory, and then the CPU may perform huge amounts of complex processing on this data.

---

1SS is pronounced by spelling it out as "I.S.S."
cases like these, it is necessary for the software engineer to have sophisticated source-level debugging capabilities. This requires the use of an instruction set simulator (ISS), which provides a virtual representation of the CPU.

Although an ISS will almost certainly be created in C/C++, it will be architected very differently from the C/C++ models of the CPU discussed earlier in this section. This is because the ISS is created at a very high level of abstraction; it thinks in terms of transactions like “get me a word of data from location x in the memory,” and it doesn’t concern itself with details like how signals will behave in the real world. The easiest way to explain how this works is by means of an illustration (Figure 13-9).

![Diagram of the simulation process](image)

**Figure 13-9. How an ISS fits into the picture.**

First of all, the software engineers capture their program as C/C++ source code. This is then compiled using the `-d` (debug) option, which generates a symbol table and other debug-specific information along with the executable machine code image.
When we come to perform the coverification, there are a number of pieces to the puzzle. At one end we have the source-level debugger, whose interface is used by the software engineer to talk to the environment. At the other end we have the logic simulator, which is simulating representations of the memory, stuff like peripheral devices, general-purpose logic, and so forth (for the sake of simplicity, this illustration assumes that all of the program memory resides in the FPGA itself).

In the case of the CPU, however, the logic simulator essentially sees a hole where this function should be. To be more precise, the simulator actually sees a set of inputs and outputs corresponding to the CPU. These inputs and outputs are connected to an entity called a bus interface model (BIM), which acts as a translator between the simulator and the ISS.

Both the source code and the executable image (along with the symbol table and other debug-centric information) are loaded into the source-level debugger. At the same time, the executable image is loaded into the MEM block. When the user requests the source-level debugger to perform an action like stepping through a line of source code, it issues commands to the ISS. In turn, the ISS will execute high-level transactions such as an instruction fetch, or a memory read/write, or an I/O command. These transactions are passed to the BIM, which causes the appropriate pins to “wiggle” in the simulation world.

Similarly, when something connected to the processor bus in the FPGA attempts to talk to the CPU, it will cause the pins driving the BIM to “wriggle.” The BIM will translate these low-level actions into high-level transactions that it passes to the ISS, which will in turn inform the source-level debugger what’s happening. The source-level debugger will then display the state of the program variables, the CPU registers, and other information of this ilk.
There are a variety of incredibly sophisticated (often frighteningly expensive) environments of this type on the market. Each has its own cunning tricks and capabilities, and some are more appropriate for ASIC designs than FPGAs or vice versa. As usual, however, this is a moving target, so you need to check around to see who is doing what before putting any of your precious money on the table.

**A rather cunning design environment**

As far as possible (and insofar as makes sense), this book attempts to steer away from discussing specific companies and products. But there’s an exception to every rule, and this is it, because a company called Altium Ltd. (www.altium.com) has come up with a rather cunning FPGA design environment called Nexar that deserves mention.

It’s difficult to know where to start, so let’s kick off by saying that we’re talking about a complete FPGA hardware/software co-design and verification environment for around $7,995. This environment targets engineers designing things like simple controllers for domestic appliances like washing machines and is based on the fact that you can now purchase FPGAs containing more than 1 million system gates for around $20.

Nexar includes a hardware development board that plugs into the back of your PC. This development board comes equipped with two daughter cards: one carrying a Xilinx FPGA and the other equipped with an Altera device. Nexar also features a number of soft microprocessor cores that replicate the functionality of industry-standard 8-bit devices like the 8051, Z80, and PIC microcontrollers (a range of 16-bit and 32-bit processor and DSP cores are planned for the future). Also

---

8 For example, Seamless from Mentor (www.mentor.com), Incisive from Cadence (www.cadence.com), and XoC from Axis Systems (www.axissystems.com).

9 This price was true circa November 2003.

10 Again, this gate-count and price are circa November 2003.
included are a library of peripheral devices, a library of around 1,500 component blocks that range from simple gates to more complex functions such as counters, and a small RTOS.

By means of a schematic capture interface, the user places blocks representing the processors, peripherals, and various logic functions and wires them together. All of the blocks supplied with Nexar are provided royalty-free. These blocks have been presynthesized, so when you are ready to rock and roll, they can be directly downloaded into the FPGA on the development board. (If necessary, you can also create your own blocks and capture their contents in RTL. These will subsequently be processed by the synthesis engine bundled with Nexar.)

Clicking on a processor block allows you to enter the C/C++ source code program to be associated with that processor. This will subsequently be processed by one of the compilers bundled with Nexar.

The idea is that everything associated with the design—hardware and software—will be downloaded into the FPGA on the development board. In order to see what's happening in the hardware, you can include a variety of virtual instrument blocks in your schematic, including logic analyzers, frequency counters, frequency generators, and so forth. When it comes to the software, Nexar provides a source-level debugger that allows you to perform all of the usual tasks like setting breakpoints, specifying watch expressions, single-stepping, stepping over, stepping into, and so on.

What can I say? I've actually seen one of these little rascals performing its magic, and I was impressed. I really like the fact that this is essentially a turnkey solution, and you get everything (no costly add-ons required) in a package the size of a shoebox. And for the class of designs it is targeting, I personally think that Nexar is going to be a hard act to follow for some time to come.