1 Controlling complexity

Technical skill is mastery of complexity while creativity is mastery of simplicity.

E. Christopher Zeeman, Catastrophe Theory, 1977

The goal of this text is to teach you how to design a processor from scratch. In a step-by-step process, we will teach you how to specify, design, and test a processor as an example of a complex digital system. We will use the commercially important Verilog hardware description language (HDL) as the basis for this design process.

In particular, we will develop the VeSPA (*Very Small Processor Architecture*) processor as a vehicle for demonstrating the overall design process. We show how the instruction set for this processor is defined, how to build an assembler for the processor, how to develop a behavioral simulator in Verilog to test the instruction set and the assembler, and how to develop a complete Verilog structural model of a pipelined implementation of the processor. We also describe the synthesis process for automatically translating this structural model into a real piece of silicon hardware. We end by demonstrating several techniques that can be used to verify the correctness of the processor design.

1.1 Hierarchical design flow

The development of any type of digital computing system is fundamentally a problem of controlling complexity. The designer of a large-scale digital system, such as a processor, begins with a high-level idea of what tasks the system is to perform. To realize this system in some physical technology, such as a collection of VLSI (Very Large-Scale Integrated circuit) silicon chips, the designer must determine how millions of individual transistors should be interconnected to perform the desired operations. The need to translate from a high-level conceptual view of the system to a specification of the complex interconnections among a virtual sea of transistors is referred to as the designer's *abstraction gap*, as suggested in Figure 1.1.

To bridge this gap between system concept and physical realization, we can use the *hierarchical design flow* process shown in Figure 1.2. The *instruction set architecture*

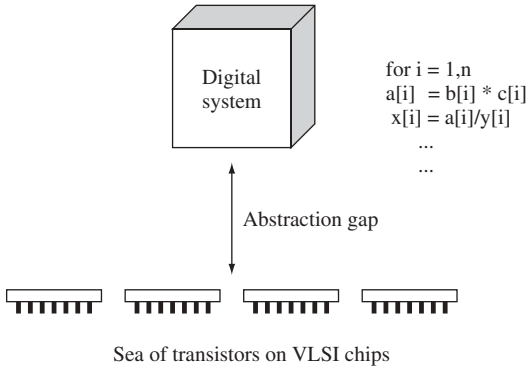

Figure 1.1. The need to translate from the high-level conceptual view of a digital system, such as a processor capable of executing the program shown above, to its physical realization in VLSI chips leads to the designer's abstraction gap.

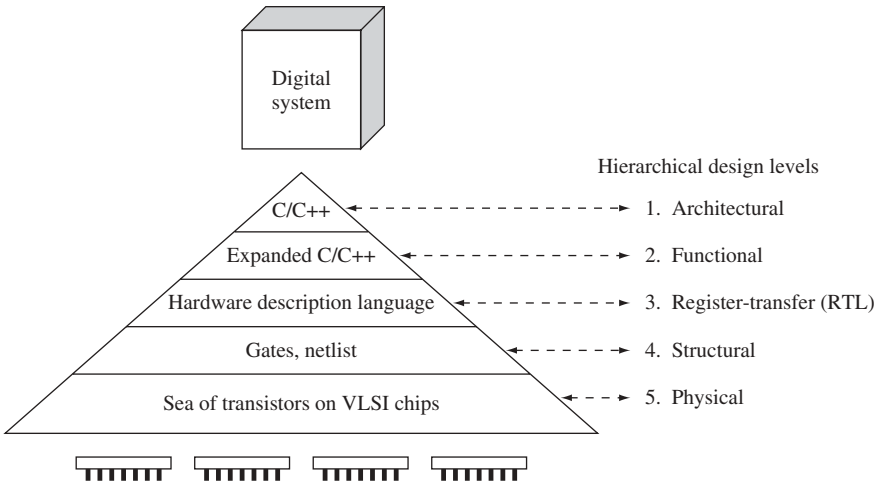

Figure 1.2. The hierarchical design flow used to bridge the abstraction gap between the high-level view of a digital system and its physical realization in VLSI chips.

(ISA) specification at the highest level in this design hierarchy provides an abstract description of what functions the system is capable of performing. In the case of the design of a processor, this level specifies the instructions available to the assembly language programmer and the programmer-visible architectural storage elements, such as the general-purpose registers, the program counter, and the processor status register. This level of the hierarchy is typically described using a written assembly language programmer's manual.

The *behavioral* level of the design hierarchy is a logical refinement of the ISA specification. This level provides precise functional information about how the system's state is affected by each of the operations specified in the ISA. The behavioral Verilog model developed in this step can actually execute machine language programs written for the processor. However, it typically contains no timing information showing how long each instruction takes to execute, nor does it specify how the operations are implemented. It is used to verify that the ISA is defined correctly, and to provide a simulator on which programmers, such as compiler writers and operating system programmers, can begin developing and testing their code.

This *behavioral* level in the design hierarchy is sometimes referred to as the *register-transfer level* (RTL) since it describes transformations that occur to the contents of registers as they are moved among the storage elements defined by the ISA. However, it does not typically specify how the transformation itself is implemented. For instance, the subtraction of the contents of register `rs2` from the contents of register `rs1` with the results stored in register `rdst` may be specified at this level in Verilog as `R[‘rdst] = R[‘rs1] − R[‘rs2]`. This RTL or behavioral description shows *what* happens in a subtraction operation, but it does not specify *how* it happened or how much time was required.

The next level in the design hierarchy is the *structural* level. This level begins to answer the questions of how a function is actually implemented. It also begins to define the number of cycles required to execute each operation, the number of buses that interconnect registers and functional units, the size of internal memory buffers, and so forth. This level represents a mapping of the behavioral model into a more specific implementation. For example, this level defines how the subtractor actually performs a subtraction operation.

Finally, the *physical* design level specifies the detailed chip-level floor-planning, layout, and transistor-level timing. It defines a mapping of the structural level description on to a specific technology, such as a CMOS application-specific integrated circuit (ASIC). This final stage in the design hierarchy often can be produced automatically from the structural Verilog description using an appropriate logic synthesis design automation tool. The designer may choose to translate certain portions of the design from the behavioral or structural level to the next level by hand, however, to optimize specific design criteria, such as power consumption, chip area, or signal delays, for instance.

Since each level in this design hierarchy is an incremental refinement of the previous level, this hierarchical design flow provides a technique for managing the complexity of designing a large digital system. The hardware description language is a means for precisely capturing the details at the behavioral and structural levels of the design hierarchy. These behavioral and structural models can be compiled and simulated to verify the correctness of the design at each level. The structural model then provides the input for a synthesis tool that will make the final transformation of the hardware description language model into a piece of silicon.

1.2 Designing hardware with software

Using the above design flow, it can often seem that designing a processor is very much like writing a piece of software. Indeed, both the behavioral and structural models of a processor are written in the hardware description language and can be changed, compiled, and executed (actually, simulated) in a manner very similar to writing, compiling, and executing a program written in a high-level programming language, such as C, C++, Java, or Fortran, for instance. However, it is important to distinguish the fundamental differences in this hardware design process from the process of writing software to run on a processor.

Figure 1.3 shows how a hardware designer begins with the ISA, develops a functional and behavioral model, refines these models into a structural model, and, finally, synthesizes that model into the actual processor. This processor consists of hardware storage elements for the ISA-defined registers, logic circuits that implement the ISA-specified instructions, and the memory system. In the final step, it is a real, tangible piece of hardware. To change the processor, for instance, to add a new instruction, every step in this chain of events must be repeated. The result is a new silicon chip with the additional logic circuits necessary to implement this new instruction. While it may be quite simple to make the changes necessary to implement this new instruction in the ISA specification manual and the Verilog models, it can be a slow and expensive process to actually fabricate the new chip.

The analogous process for compiling and executing a program written in a high-level language is compared to this hardware design process in Figure 1.4. A programmer begins with a logical description of the task to be accomplished by the program. This description is refined into an algorithm that describes the steps

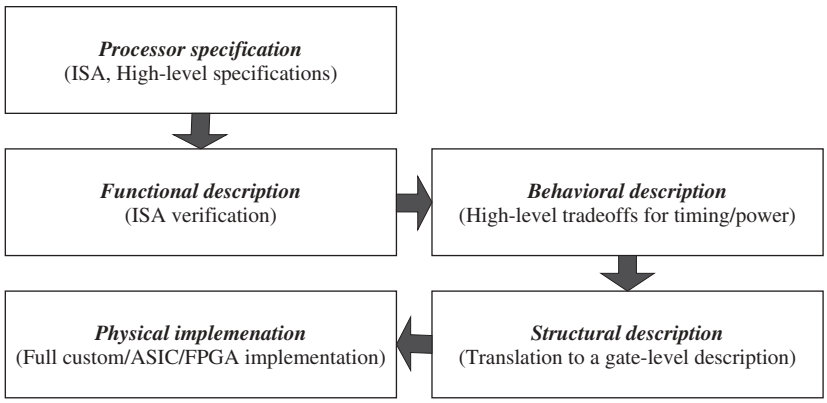

Figure 1.3. The process of refining the behavioral and structural models in a hardware description language to produce a new processor.

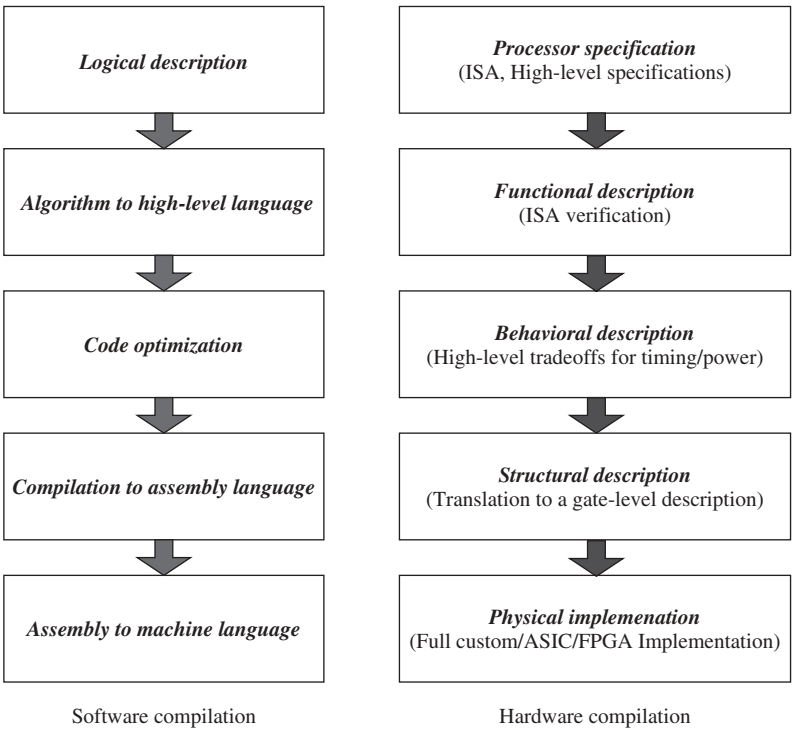

Figure 1.4. The process of compiling and executing a program to run on an existing processor has strong similarities to the process of designing a digital system using a hardware description language. However, the final results of the two processes are quite different.

necessary to complete the desired task. This algorithm then is written in a textual format in the syntax of some high-level language, such as C/C++. A compiler reads this text file and transforms the program into another text file containing an equivalent program in the target processor's assembly language. Finally, this text file is read by an assembler and converted into a string of bits that can be linked with other precompiled library functions and loaded into the processor's memory. At this point, the processor can execute the program stored in its memory.

Changing anything in the program requires each of these steps to be repeated beginning with the text file that contains the modified high-level program. However, in contrast to the need to produce a new silicon chip, the recompiled program can simply be loaded into the processor's memory where it is ready to be re-executed. While the steps required to produce the processor chip are analogous to those required to compile and execute a program, the last step in each process produces completely different results. The software compilation process produces a string of bits that are stored in the processor's memory. The hardware development process, however, ultimately produces a new artifact in the form of a new piece of silicon.

1.3 Summary

While shrinking process technologies have permitted the possibility of building large integrated circuits, they have also forced designers to battle with the task of managing this complexity under pressing time-to-market constraints on the design cycle. As a natural result, there has been an increased amount of automation throughout the entire design process. One manifestation of this change has been an evolution from building circuits at the gate or transistor level to developing an ability to specify circuits at a reasonably high level, from where a circuit implementation can be obtained in a push-button manner. Interestingly, a similar evolution was seen in software: in the early years, software was written at the assembler level, but as programs became more complex, there was a move towards writing programs in a high-level language that could be translated by a compiler into machine code.

HDLs have played an important role in this process, as they represent a medium for designers to specify a circuit at a level that is comparable to a high-level language. Once a design is specified in such a standardized format, there are a number of computer-aided design (CAD) tools that will compile this code to translate the specification into a hardware implementation. In today's world, Verilog and VHDL are the two most widely used HDLs. Since the subject of which of these is the better language can prompt fierce partisan reactions among the believers of either sect, we will prudently avoid that debate.[1] In this book, we use the Verilog HDL as a vehicle for processor design, and our rationale for this choice is threefold. Firstly, we feel that Verilog, being more C-like, is easier for the novice to learn, and is less encumbered with syntactic niceties than VHDL. Secondly, it is arguably the most widely used HDL in industry today. Thirdly, and we believe, most convincingly, learning Verilog provides an easy path to learning any other HDL, and the major goal of this book is to profess the ideas behind processor design using HDLs, rather than to evangelize any specific HDL.

[1] A third prong to the debate may be added in the near future with the advent of newer hardware description languages based on C, although these are in the early stages of use at this time.

2 A Verilogical place to start

Let's start at the very beginning.
A very good place to start.
When you read, you begin with A-B-C,
when you sing, you begin with Do-Re-Mi.

from Rogers and Hammerstein's The Sound of Music

In this chapter, we will present an elementary introduction to Verilog, with the primary aim of permitting the reader to learn enough of the language to carry out a competent design. Due to the scope of this text, we do not attempt to present complete coverage of the language; for example, we will not cover switch-level modeling concepts that are typically at the transistor-level, since the design of our processor does not require that level of design detail. For these and other details, the interested reader may refer to sources such as those shown in the *Further reading* section at the end of this chapter.

2.1 My Veri first description

In teaching an English-speaker a new tongue such as Spanish or Japanese or Marathi, two extreme approaches may be attempted. A more structured approach would lead the student through a rigorous path that first teaches the alphabet, followed by words, sentences and grammar; an alternative 'immersive' or 'communicative' approach places the student in an environment where the language is extensively spoken, in the hope that this may motivate learning in a more natural environment. In practice, of course, an intermediate approach is often the most effective, and we will use a similar philosophy in presenting a first exposition to the admittedly nonhuman language that is Verilog.

In this section, we will present a simple Verilog description of a very simple module. The module that we present is a full adder that inputs three bits, the addend

bits, a and b, and the input carry bit, c_{in}. The full adder outputs two bits, the sum bit, s, and the output carry bit, c_{out}, and these are related by the elementary equations

$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

A Verilog description is shown in Figure 2.1, and is fairly self-explanatory. The description is encapsulated within a `module` that is parameterized by its *ports*, or connections to the external world. The body of the module first declares these parameters as inputs or outputs, followed by a description that defines the functionality of the module in terms of its logic equations; note that the symbols ^, & and | refer to the XOR, AND and OR logical operators, respectively.

```
module full_adder(a,b,cin,s,cout);

  input a,b,cin;     // declaration of the list of inputs
  output s, cout;    // declaration of the list of outputs

  assign s = a ^ b ^ cin;
  assign cout = (a & b) | (a & cin) | (b & cin);

endmodule
```

Figure 2.1. Verilog module for a full adder.

This module may be used as a building block for the hierarchical description of a more complex module, such as the four-bit ripple carry adder of Figure 2.2 whose Verilog description is shown in Figure 2.3. As before, the module description begins with a listing and then a declaration of all inputs and outputs. An interesting feature is in the definition of several vectors. For instance, the inputs `p` and `q` are defined as four-bit vectors, with bit 3 being the most significant bit; alternatively, if '[3:0]' were to be substituted by '[0:3]', then bit 0 would be the most significant bit. The next declaration of a `wire` corresponds, in this example, to an internal connection within the module. Finally, the description of the adder instantiates four copies of `full_adder` defined in Figure 2.1. Note that the order of the port connections corresponds exactly to the order used in the definition of `full_adder`[1].

Through the examples shown above, it is easy to see that the specification of the adder is very similar to writing a program in a high-level programming language. A Verilog compiler has the capability of translating this description to a hardware

[1] Verilog does indeed permit an out-of-order specification of the list of ports, but we prefer to avoid its use in this book.

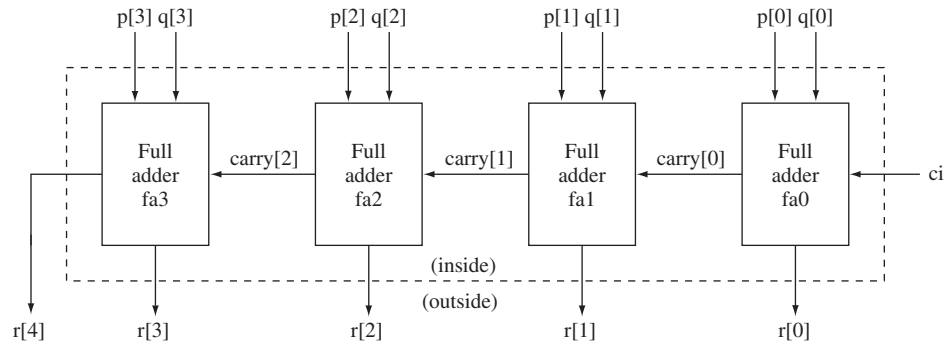

Figure 2.2. Schematic of a four-bit ripple-carry adder.

```
module RCA(p,q,ci,r);

input [3:0] p, q;        // Declaration of two four-bit inputs
input ci;                // and the one-bit input carry

output [4:0] r;          // Declaration of the five-bit outputs

wire [2:0] carry;        // Declaration of internal carry wires

full_adder fa0(p[0],q[0],ci,r[0],carry[0]);
full_adder fa1(p[1],q[1],carry[0],r[1],carry[1]);
full_adder fa2(p[2],q[2],carry[1],r[2],carry[2]);
full_adder fa3(p[3],q[3],carry[2],r[3],r[4]);

endmodule
```

Figure 2.3. Verilog module for a four-bit ripple-carry adder.

implementation to a specified target. This target could be a field programmable gate array (FPGA) implementation or a semicustom implementation using a standard cell library.

2.2 A more formal introduction to the basics

2.2.1 Modules and ports

As exemplified by the discussion in Section 2.1, a typical Verilog description is contained within a module, and this module may instantiate one or more copies of other modules. The body of each module contains a list of port declarations, followed by the description of the module, and terminated by the keyword `endmodule`.

The ports of the module may be of type `input`, `output`, or `inout`, where the last corresponds to a bidirectional connection.

2.2.2 Nets and registers

Apart from the port declarations, it is also possible to make declarations that are entirely internal to the module. These declarations may correspond to:

Net variables correspond to structural connections between two or more elements within the module. These are most often declared within a model using the keyword `wire`. Alternative specifications of nets also exist to define, for example, tristable nets (e.g., `triand`, `trior`), nets that can implement wired AND or wired OR logic functions (e.g., `wand`, `wor`), and power/ground nets (`supply1` and `supply0`). The default value of a `wire` data type is the tristate value **z**.

Register variables correspond to elements with memory that can store their value until they are next updated. The most commonly used register data type is denoted by the keyword `reg`, and its default value is the unknown value **x**. The differences between a register declaration and a hardware implementation of a register are subtle and include the fact that the former may or may not be clocked and may be changed asynchronously. In many cases, though, there is a one-to-one relationship between the two: the burden of ensuring this is placed on the designer, who can force synchronicity by updating the register variable in the Verilog description only on the onset of a clock.

Unless specified in terms of an array, both the `wire` and `reg` keywords represent a single bit. Particularly in the case of stored variables, more compact representations are often convenient. To facilitate this, Verilog permits register variables of type `integer` and `real` to store integer and real variables, respectively. For example, in specifying the value of a counter, an integer variable may be more convenient to use, and may make the description more readable than the use of an array of bits.

Another stored variable that is provided for convenience is the integer `time` data type, which is most commonly used in conjunction with the system function `$time` that provides the current time point of the simulation as an integer. A corresponding data type `realtime` can be used to store time in the form of a real number.

Verilog imposes stringent requirements on the mapping from the internals of a module to its ports, and these are summarized below. Since a port defines the connection to a module to the external world, it may be viewed from the inside of a module as well as from the outside. For the example of the full adder module that was