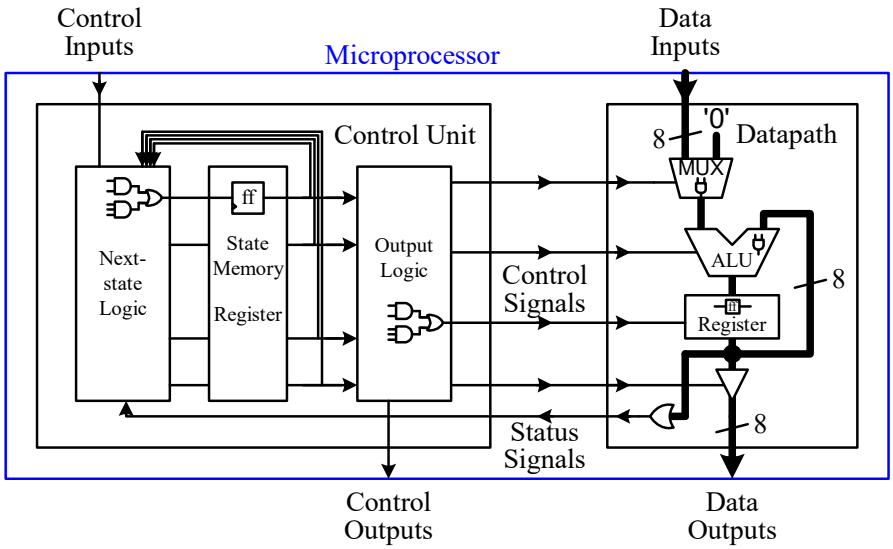
Contents

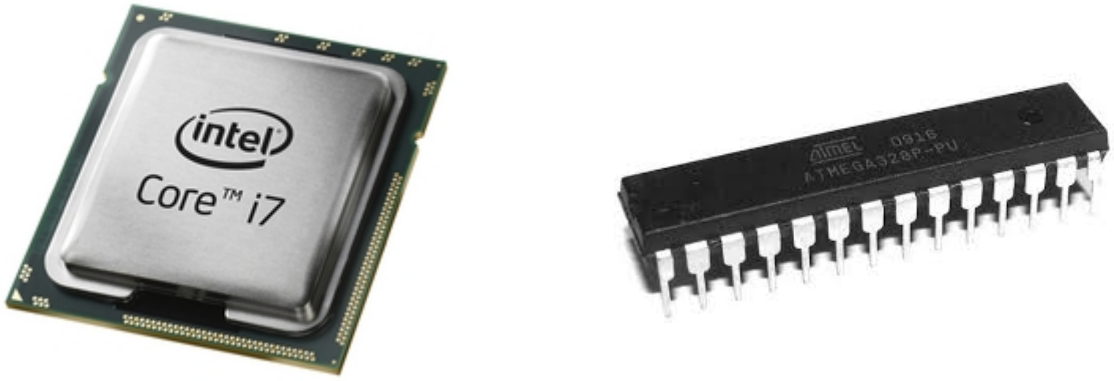
Chapter 1

1 Introduction to Microprocessor Design



Electronic devices are an integral part of our lives. Every day and everywhere we see and use electronic devices, from cellular telephones to electronic billboards, cars, toys, TVs, elevators, musical greeting cards, personal computers, traffic lights, and many more. Inside each and every one of them, there is a microprocessor that controls their operations. Microprocessors are at the heart of all of these "smart" devices. Their smartness is a direct result of the work of the microprocessor, without which none of these electronic devices would be able to operate as they do.

There are generally two types of microprocessors: **general-purpose microprocessors** and **dedicated microprocessors**. General-purpose microprocessors, such as the Intel Core™ i7 CPU shown in Figure 1.1(a), can perform different tasks under the control of different software programs. General-purpose microprocessors typically are much more powerful in terms of processing power and speed. However, they usually require external components for their memory and supporting input/output (I/O) peripherals. They are used in all personal computers.



(a) (b)

Figure 1.1 Microprocessors: (a) General-purpose Intel Core™ i7 CPU; (b) Dedicated Atmel 328P microcontroller.

Dedicated microprocessors, also known as **microcontrollers** or **application-specific integrated circuits** (ASICs), on the other hand, are designed to perform just one specific task. For example, inside your cell phone is a dedicated microcontroller that does nothing else but controls its entire operation. Microcontrollers therefore are usually not as powerful (because they do not need to perform so many tasks) as a microprocessor and much smaller in size. However, they usually will have the memory and supporting I/O peripherals included inside the chip, hence the entire system can be on a single chip. For example, the Atmel ATMEGA328P microcontroller shown in Figure 1.1(b) has built-in flash memory, electrically erasable programmable read-only memory (EEPROM), static random-access memory (SRAM), general purpose I/O's, timers, serial interface, and analog-to-digital converters (ADC). Dedicated microcontrollers are used in almost all smart electronic devices. Although the small dedicated microcontrollers are not as powerful and are slower in speed as compared to general-purpose microprocessors, they are being sold much more and are used in a lot more places than general-purpose microprocessors.

In this book, I will show you in detail, how to design, implement, and interface a microprocessor. At the end, you will be able to design your own custom microprocessor and use it to control your own electronic device**.** I will use a hands-on approach to guide you step-by-step through the entire design process with complete circuits that you actually can implement in hardware. The exciting part is that at the end, you can actually, very easily and inexpensively, implement your own custom microprocessor in a real integrated circuit (IC) and see that it really can execute software programs, make lights flash or do whatever you have designed it to do.

We will start with the fundamentals of digital logic circuit design in Chapter 2, which will provide you with a good foundation and basic building blocks for creating larger and more complex digital circuits. Chapters 3 and 4 will discuss the design of simple digital circuits and common circuits that are used as building blocks for larger circuits. Chapter 5 talks about the design of memory circuits. Typically, an introduction to digital logic design course will cover the materials from Chapters 1 to 5 only. Moving on to more advanced digital logic design, Chapter 6 talks about control unit circuits. Chapter 7 talks about the datapath and how to connect it with the control unit to produce a dedicated microcontroller. Chapter 8 extends the dedicated microcontroller from Chapter 7 to produce a

general-purpose microprocessor. Finally, Chapter 9 concludes with examples of how to interface these microprocessors and microcontrollers in the real world.

1.1 Overview of Microprocessor Design

The microprocessor or microcontroller is an electronic digital logic circuit that is implemented inside an IC chip. Any digital electronic circuit at the lowest physical level understands only whether there is electricity or no electricity, which is typically represented by the use of a 1 or a 0. The question is how do we design a microprocessor so that it can understand the 1s and 0s, and then do something meaningful with that understanding? To design a microprocessor is to design its logic circuit to do whatever it is intended to do. To implement the microprocessor is to put the logic circuit of the microprocessor onto an IC chip.

Previously, making an IC chip with any circuit was a long and expensive process. With the advance of very large-capacity field-programmable gate array (FPGA) chips, digital circuits of almost any size can be implemented in a chip easily and quickly. Moreover, because FPGA chips are erasable, you can use the same FPGA chip over and over again to implement different circuits. If you put an adder circuit in the FPGA chip, that chip will be an adder, and if you put a traffic light controller circuit in the FPGA chip, that chip will be a traffic light controller. So implementing any digital circuit in a FPGA chip is quite simple. The challenge now is how to design the circuit; how do we design the adder circuit or the traffic light controller circuit?

A block diagram of a microprocessor circuit is shown in Figure 1.2. As you can see, it is divided into two main parts: the **control unit** and the **datapath**. The datapath is responsible for the execution of all of the microprocessor's data operations, such as the addition of two numbers inside the arithmetic logic unit (ALU). The datapath also includes registers for the temporary storage of data and comparators for testing data values. These and many other functional units are connected together with multiplexers and data signal lines. The data signal lines are for transferring data between two functional units. Sometimes, several data signal lines are grouped together to form a **bus**. The width of the bus (i.e., the number of data signal lines in the group) is annotated next to the bus line. In the figure, the bus lines are thicker and are 8-bit wide. Multiplexers, also known as MUXs, are for selecting data from two or more sources to go to one destination. In the figure, a 2-to-1 multiplexer is used to select between the input data and the constant "0" to go to the left operand of the ALU. The output of the ALU is connected to the input of the register. The output of the register is connected to three different destinations: (1) the right operand of the ALU; (2) an OR gate used as a comparator for the test "not equal to 0"; and (3) a tri-state buffer, which is used to control the output of the data from the register.

Even though the datapath is capable of performing all of the microprocessor's data operations, it cannot, however, do it on its own. In order for the datapath to execute the operations automatically and correctly, a control unit is required. The control unit, also known as the **controller**, controls all of the operations of the datapath and therefore, the operations of the entire microprocessor. The control unit is also called a **finite-state machine** (FSM) because it is a machine that executes by going from one state to another, and there are only a finite number of states for the machine to go to. The control unit is made up of three parts: (1) the **next-state logic;** (2) the **state memory;** and (3) the **output logic**. The purpose of the state memory is to remember the current state that the FSM is in. The next-state logic is the circuit that determines what the machine's next state should be. The output logic is the circuit that generats the actual control signals for controlling the datapath and/or external devices.

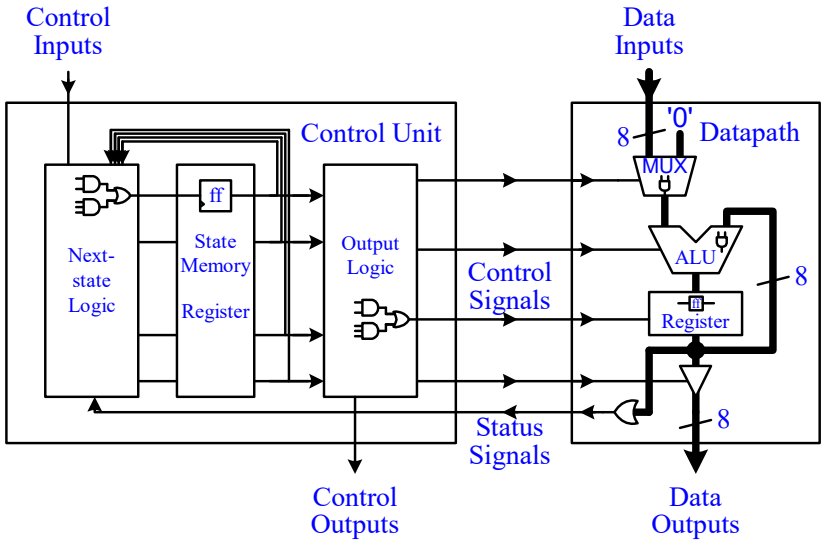


Figure 1.2 Internal parts of a microprocessor.

Every digital logic circuit, regardless of whether it is part of the control unit or the datapath, is categorized as either a **combinational circuit** or a **sequential circuit**. A combinational circuit is one where the output of the circuit is dependent only on the current inputs to the circuit, and therefore has no memory about what has happened before. For example, an adder is a combinational circuit because it will produce a sum when given any two input numbers.

A sequential circuit, on the other hand, is dependent not only on the current inputs, but also on all of the previous inputs. In other words, a sequential circuit has to remember its past history. For example, a register is a sequential circuit because it can remember a value indefinitely. Because sequential circuits are dependent on the history, they must contain memory elements to remember that history. Combinational circuits, on the other hand, do not need to remember the history, and so they do not have memory elements.

An analogy of the difference between a combinational circuit and a sequential circuit is the combination lock that we are familiar with. There are actually two different types of combination locks as shown in **Error! Reference source not found.**. For the lock in Figure 1.3(a), you just turn the three number dials in any order you like to the correct number and the lock will open. For the lock in Figure 1.3(b), you also have three numbers that you need to turn to, but you need to turn to these three numbers in the correct sequence. If you turn to these three numbers in the wrong sequence the lock will not open even if you have the numbers correct. The lock in Figure 1.3(a) is like a combinational circuit where the order in which the inputs are entered into the circuit does not matter, whereas, a sequential circuit is like the lock in Figure 1.3(b) where the sequence of the inputs does matter.

(a) (b)

Figure 1.3 Two types of combination locks: (a) the order in which you enter the numbers does not matter; (b) the order in which you enter the numbers does matter.

Examples of combinational circuits inside the microprocessor include the ALU, multiplexers, tri-state buffers, and comparators in the datapath, and the next-state logic and output logic circuits in the control unit. Examples of sequential circuits include the register for the state memory in the control unit and the registers in the datapath.

All digital logic circuits, whether they are combinational or sequential, are made up of the three basic logic gates: **AND**, **OR**, and **NOT** gates. From these three basic gates, the most powerful computer can be made. Furthermore, these basic gates are built using transistors—the fundamental building blocks for all digital logic circuits. Transistors are simply electronic binary switches that can be turned on and off. The 1s and 0s that we, as computer scientists, often talk about are used to represent the on and off states of a transistor.

To summarize, transistors, as the lowest-level building blocks, are used to build the basic logic gates. Logic gates are connected together to form either combinational circuits or sequential circuits. The difference between these two types of circuits is only in the way the logic gates are connected together. Certain combinational circuits and sequential circuits are used as standard building blocks for larger circuits and so are kept in standard libraries. These standard combinational and sequential components are connected together to form either the datapath or the control unit. Finally, combining the datapath and the control unit together will produce the circuit for either a dedicated or a general-purpose microprocessor.

1.2 Design Abstraction Levels

Digital circuits can be designed at any one of several abstraction levels. When designing a circuit at the **transistor level**, which is the lowest level, you are dealing with discrete transistors and connecting them together to form the circuit. The next level up in the abstraction is the **gate level**. At this level, you are working with logic gates to build the circuit. In using logic gates, a designer usually creates standard combinational and sequential components for building larger circuits. In this way, a very large circuit, such as a microprocessor, can be built in a hierarchical fashion. Design methodologies have shown that solving a problem hierarchically is always easier than trying to solve the entire problem as a whole from the ground up. These combinational and sequential components are used at the **register-transfer level** to build the datapath and the control unit in the microprocessor. At the register-transfer level, we are concerned with how the data is transferred between the various registers and functional units to realize or solve the problem at hand. Finally, at the highest level, called the **behavioral level**, we can describe the behavior or operation of the circuit using a high-level hardware description language, and we can use a synthesizer, which is equivalent to a compiler, to automatically generate the logic circuit for it. Designing at this level does not require knowledge of the underlying logic gates and circuits because the synthesizer will automatically create the logic circuit for you. This is very similar to writing a computer program using a high-level programming language, and then using the compiler to automatically translate the program into machine language that the computer can execute.

An important point to realize is that there are many different ways to create the same functional circuit. Although they are all functionally equivalent, they are different in other respects, such as the actual circuit (how the transistors or gates are connected together), size (how big the circuit is or how many transistors or gates it uses), speed (how long it takes for the output result to be valid), cost (how much it costs to manufacture), and power usage (how much power it uses). Hence, when designing a circuit, in addition to being functionally correct, we also should consider the economic versus performance tradeoffs. In this book, we will focus mainly on how to design a functionally correct circuit with some discussion about how to optimize the circuit size.

1.3 Examples of a 2-to-1 Multiplexer

As an introduction example, let us look at the design of the 2-to-1 multiplexer from different abstraction levels. At this point, don't worry too much if you don't understand the details of how all of these circuits are built. This example is intended just to give you an idea of what the circuit looks like at the different abstraction levels. We will get to the details in the rest of the book.

The multiplexer is a component that is used a lot in the datapath. An analogy for the operation of the 2-to-1 multiplexer is similar in principle to a railroad switch in which two railroad tracks are to be merged onto one track. The switch controls which one of the two trains on the two separate tracks will move onto the one track. Similarly, the 2-to-1 multiplexer has two data inputs, d_1 and d_0, and a select input, s. The select input determines which data from the two data inputs will pass to the output, y.

Figure 1.4 shows the graphical symbol, also referred to as the **logic symbol**, for the 2-to-1 multiplexer. From looking at the logic symbol, you can tell how many signal lines the 2-to-1 multiplexer has, and the name or function designated for each line. For the 2-to-1 multiplexer, there are two data input signals, d_1 and d_0, a select input signal, s, and an output signal, y.

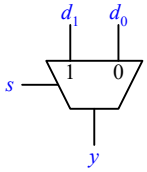


Figure 1.4 Logic symbol for the 2-to-1 multiplexer.

1.3.1 Behavioral Level

We can describe the operation of the 2-to-1 multiplexer simply (using the same names as in the logic symbol) by saying that

if $s = 0$ then d_0 passes to y,

otherwise

d_1 passes to y.

Or more precisely, the value that is at d_0 passes to y if $s = 0$, and the value that is at d_1 passes to y if $s = 1$.

We use a hardware description language (HDL), which is quite similar to many high-level computer programming languages, to describe the circuit at the **behavioral** level. When describing a circuit at this level, you would write basically the same thing as in the description, except that you have to use the correct syntax required by the hardware description language. Figure 1.5 shows the description of the 2-to-1 multiplexer using the hardware description language called **Verilog**, and Figure 1.6 shows the description of the same 2-to-1 multiplexer using another hardware description language called **VHDL**, which stands for VHSIC Hardware Description Language (VHSIC, in turn stands for Very High Speed Integrated Circuit). Verilog and VHDL are two standard hardware description languages used for digital logic design.

```
module multiplexer (
  input s,
  input d0,
  input d1,
  output reg y
);

  always @ (s or d0 or d1) begin
    if (s == 0) begin
      y = d0;
    end else begin
      y = d1;
    end
  end

endmodule
```

Figure 1.5 Behavioral Verilog code for a 2-to-1 multiplexer.

In the Verilog code shown in Figure 1.5, the declaration of the component begins with the keyword `module` followed by the name of the component, which in the example, is the user identifier `multiplexer`. All of the words used in Verilog are case sensitive. The input and output interface signals are listed next using the keywords `input` and `output`. For ease of reference, the user-defined names used for these signals match those shown earlier in the logic symbol. The `always` block is followed by its sensitivity list of signals inside the parentheses. The `always` block is executed each and every time when any one of the signals in the sensitivity list changes value. The statements inside the `always` block (bracketed by the `begin` and `end` keywords) are executed sequentially. In the

example, there is only one `if-then-else` statement inside the block. Like any `if` statement in other programming languages, the assignment statement `y = d0` is executed when the condition "s equals to 0" is true, otherwise the assignment statement `y = d1` is executed. For the two assignment statements, the value for the expression on the right side of the equal sign is assigned to the signal on the left side of the equal sign. Notice that the output signal `y` on the left side of the equal sign is declared as a `reg` because assignment statements inside the `always` block cannot drive a `wire` data type, but can only drive a register or an integer data type. Finally, the module is terminated with the keyword `endmodule`. A summary of the Verilog language can be found in Appendix C.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY multiplexer IS PORT (
  s, d0, d1: IN STD_LOGIC;
  y: OUT STD_LOGIC);
END multiplexer;

ARCHITECTURE Behavioral OF multiplexer IS
BEGIN
  PROCESS(s, d0, d1)
  BEGIN
    IF (s = '0') THEN
      y <= d0;
    ELSE
      y <= d1;
    END IF;
  END PROCESS;

END Behavioral;
```

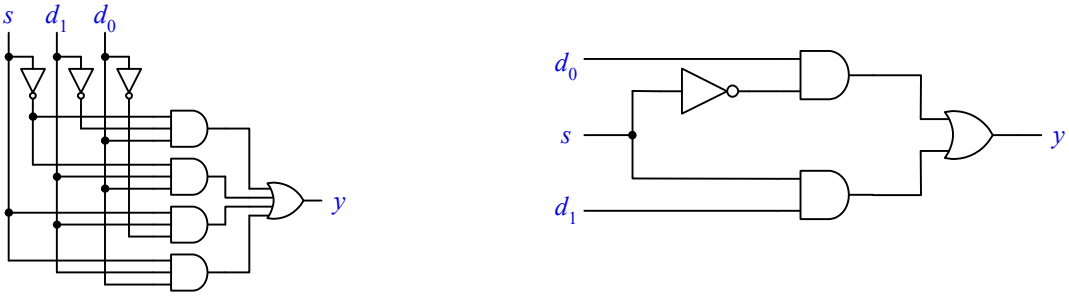
Figure 1.6. Behavioral VHDL code for a 2-to-1 multiplexer.

In the VHDL code shown in Figure 1.6, the LIBRARY and USE statements are similar to the "#include" and "using namespace" preprocessor commands in C++. None of the words used in VHDL are case sensitive, however, in the examples, the keywords are shown in upper case. The IEEE library contains the definition for the STD_LOGIC type used in the declaration of signals. The ENTITY section declares the interface for the circuit by specifying the input and output signals of the circuit. In this example, there are three input signals of type STD_LOGIC and one output signal also of type STD_LOGIC. Again, the names used for these signals match those shown earlier in the logic symbol. The ARCHITECTURE section defines the actual operation of the circuit. The ARCHITECTURE keyword is followed by a user identifier name and the entity that it is for. The PROCESS block with its sensitivity list is like the `always` block in Verilog. The operation of the multiplexer is defined in the conditional IF-THEN-ELSE statement. The two signal assignment statements, which use the symbol <= to denote the signal assignment, in conjunction with the IF-THEN-ELSE statement, says that the signal y gets the value of d_0 if s is equal to 0; otherwise, y gets the value of d_1. The PROCESS block is terminated by the END PROCESS statement, and the ARCHITECTURE block is terminated by the END keyword followed by the name of this architecture. A summary of the VHDL language can be found in Appendix D.

Having written the behavioral code, either in Verilog or VHDL, we will use a synthesizer to automatically construct the netlist (which is the circuit connections) that will operate according to the description of the code. As you can see, when designing circuits at the behavioral level, we do not need to know what logic gates are needed or how they are connected together. We only need to know their interface and functional operation, and then describe it using an HDL.

1.3.2 Gate Level

At the gate level, you can draw a **schematic diagram** or **circuit diagram**, which shows how the logic gates are connected together. Two different schematic diagrams of a 2-to-1 multiplexer circuit are shown in Figure 1.7(a) and (b). In Figure 1.7(a), the circuit uses three NOT gates (), four 3-input AND gates (), and one 4-input OR gate (). In Figure 1.7(b), only one NOT gate, two 2-input AND gates, and one 2-input OR gate are needed. Although one circuit is larger (in terms of the number of gates needed) than the other, both of these circuits realize the same 2-to-1 multiplexer function. Therefore, when we want to actually implement a 2-to-1 multiplexer circuit, we will want to use the second, smaller circuit rather than the first.



(a) (b)

Figure 1.7 Gate level circuit diagram for the 2-to-1 multiplexer: (a) circuit using eight gates; (b) circuit using four gates.

At the gate level, you also can describe the 2-to-1 multiplexer using a **truth table** or with a **Boolean equation** as shown in Figure 1.8(a) and (b), respectively. For the truth table, we list all possible combinations of the binary values for the three inputs, s, d_0, and d_1, and then determine what the output value y should be based on the functional description of the circuit. We see that for the first four rows of the table when $s = 0$, y has the same values as d_0; whereas, in the last four rows when $s = 1$, y has the same values as d_1.

The Boolean equation in Figure 1.8(b) can be derived from either the schematic diagram or the truth table. The first equality in Figure 1.8(b) matches the truth table in Figure 1.8(a) and also the schematic diagram in Figure 1.7(a). The second equality in Figure 1.8(b) matches the schematic diagram in Figure 1.7(b). To derive the first equality equation from the truth table, we look at all of the rows where the output y is a 1. Each of these rows results in a term in the equation. For each term, the variable is primed (′) when the value of the variable is a 0, and unprimed when the value of the variable is a 1. For example, the first term $s'd_1'd_0$ in the equation is obtained from the first row in the truth table where y is a 1, since s is a 0, d_1 is a 0 and d_0 is a 1.

s	d_1	d_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a)

$$y = s'd_1'd_0 + s'd_1d_0 + sd_1d_0' + sd_1d_0$$
$$= s'd_0 + sd_1$$

(b)

Figure 1.8 Gate level description of the 2-to-1 multiplexer: (a) using a truth table; (b) using a Boolean equation.

1.3.3 Transistor Level

The 2-to-1 multiplexer circuit at the transistor level is shown in Figure 1.9. It contains six transistors, three of which are P-type metal-oxide semiconductor (PMOS) transistors (⏊), and three are N-type metal-oxide semiconductor (NMOS) transistors (⊥). The pair of transistors on the left forms a NOT gate for the signal *s*, while the two pairs of transistors on the right form two transmission gates. The transmission gate allows or disallows the data signal d_0 or d_1 to pass through, depending on the control signal *s*. The top transmission gate is turned on when *s* is a 0, and the bottom transmission gate is turned on when *s* is a 1. Hence, when *s* is 0, the value at d_0 is passed to *y*, and when *s* is 1, the value at d_1 is passed to *y*.

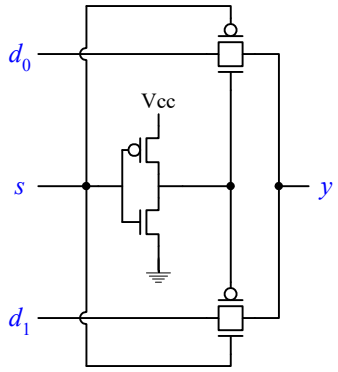


Figure 1.9 Transistor circuit for the 2-to-1 multiplexer.

A more detailed discussion about how to design digital circuits at the transistor level can be found in the online resources.

1.4 Introduction to Hardware Description Language

The popularity of using hardware description languages (HDL) to design digital circuits began in the mid-1990s when commercial synthesis tools became available. Two popular HDLs used by many engineers today are VHDL and Verilog. VHDL was sponsored and developed jointly by the U.S. Department of Defense and the Institute of Electrical and Electronic Engineers (IEEE) in the mid-1980s. It was standardized by the IEEE in 1987 (VHDL-87), and later extended in 1993 (VHDL-93). Verilog, on the other hand, was first introduced in 1984, and again later in 1988, as a proprietary hardware description language by two companies: Synopsys and Cadence Design Systems.

Both Verilog and VHDL, in many respects, are similar to a regular computer programming language, such as C. For example, it has constructs for variable assignments, conditional statements, loops, and functions (just to name a few). In a computer programming language, a compiler is used to translate the high-level source code to machine code. In HDL, however, a synthesizer is used to translate the source code to a description of the actual hardware circuit that implements the code. From this description, which we call a **netlist**, the actual, physical digital device that realizes the source code can be made automatically. Accurate functional and timing simulation of the code is also possible in order to test the correctness of the circuit.

We saw in the previous section how we used Verilog and VHDL to describe the 2-to-1 multiplexer at the behavioral level. HDL can also be used to describe a circuit at other levels. Figure 1.10 shows the VHDL code for the multiplexer written at the **dataflow** level. The main difference between the behavioral VHDL code shown in Figure 1.6 and the dataflow VHDL code is that, in the behavioral code, there is a PROCESS block statement; whereas, in the dataflow code, there is no PROCESS statement. Statements within a PROCESS block are executed sequentially as in a computer program, while statements outside a PROCESS block (including the PROCESS block itself) are executed concurrently or in parallel. The signal assignment statement, using the symbol <=, is derived directly from the Boolean equation for the multiplexer, as shown in Figure 1.8(b), using the built-in VHDL operators: AND, OR, and NOT.

```
LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY multiplexer IS PORT(
  s, d0, d1: IN STD_LOGIC;
  y: OUT STD_LOGIC);
END multiplexer;

ARCHITECTURE Dataflow OF multiplexer IS
BEGIN
  y <= ((NOT s) AND d0) OR (s AND d1);
END Dataflow;
```

Figure 1.10. Dataflow level VHDL description of the 2-to-1 multiplexer.

The corresponding Verilog version of the 2-to-1 multiplexer written at the dataflow level is shown in Figure 1.11. For Verilog, the `assign` keyword is used for the signal assignment, and the symbols &, |, and ~ are used for the logical operators AND, OR, and NOT, respectively.

```
module multiplexer (
  input s, d0, d1,
  output y
);

  assign y = ((~s) & d0) | (s & d1);

endmodule
```

Figure 1.11. Dataflow level Verilog description of the 2-to-1 multiplexer.

In addition to the behavioral and dataflow levels, we also can write HDL code at the **structural** level. Figure 1.13 shows the Verilog code for the multiplexer written at the structural level and Figure 1.14 shows the VHDL code. The code is based on the circuit shown in Figure 1.7(b) and duplicated here in Figure 1.12.

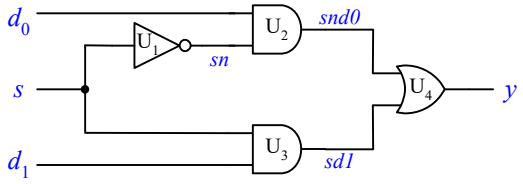


Figure 1.12. 2-to-1 multiplexer circuit.

```
module multiplexer (
  input s, d0, d1,
  output y
);

  wire sn,snd0,sd1;

  // first parameter is the output; remaining parameters are the inputs
  not U1(sn,s);
  and U2(snd0,sn,d0);
  and U3(sd1,s,d1);
  or  U4(y,snd0,sd1);

endmodule
```


Figure 1.13 Structural Verilog code for the 2-to-1 multiplexer.

In the structural Verilog code shown in Figure 1.13, the keywords (`not`, `and`, `or`) for the various gates are used. The first parameter for these gate statements is the output from the gate. The remaining parameters in the statements are the inputs to the gate. The `wire` keyword defines user identifiers using the wire data type for connecting the gates together based on the schematic diagram. For example, looking at the schematic diagram, the NOT gate labeled U_1 has *s* as the input and *sn* as the output. Hence in the corresponding code, we have the statement `not U1(sn,s)`. The user-identifier name U1 in the statement is optional, and is added only for easy identification of the gate in the circuit.

The structural VHDL code shown in Figure 1.14 looks more complicated than the Verilog code but it actually does the same thing. The three different gates (*notgate*, *and2gate*, and *or2gate*) used in the circuit are declared and defined first using the ENTITY and ARCHITECTURE statements, respectively. After this, the multiplexer is declared (also with the ENTITY statement). The actual, structural definition of the multiplexer is in the ARCHITECTURE section for *multiplexer*. First of all, the COMPONENT statements specify what components are used in the circuit. The SIGNAL statement declares the three internal signals (*sn*, *snd0*, and *sd1*) that will be used in the connection of the circuit. Finally, the PORT MAP statements declare the instances of the gates used in the circuit and specify how they are connected using the external and internal signals. So if you ignore all of the preliminary declaration stuff, the last few statements in the VHDL code matches the statements in the Verilog code.

```
----------------- NOT gate ----------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY notgate IS PORT(
  i: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END notgate;
ARCHITECTURE Dataflow OF notgate IS
BEGIN
  o <= NOT i;
END Dataflow;

----------------- 2-input AND gate ---------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY and2gate IS PORT(
  i1, i2: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END and2gate;
ARCHITECTURE Dataflow OF and2gate IS
BEGIN
  o <= i1 AND i2;
END Dataflow;

----------------- 2-input OR gate ----------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY or2gate IS PORT(
  i1, i2: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END or2gate;
ARCHITECTURE Dataflow OF or2gate IS
BEGIN
  o <= i1 OR i2;
END Dataflow;
```

```
---------------- 2-to-1 multiplexer ------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY multiplexer IS PORT(
  s, d0, d1: IN STD_LOGIC;
  y: OUT STD_LOGIC);
END multiplexer;
ARCHITECTURE Structural OF multiplexer IS
  COMPONENT notgate PORT(
    i: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT and2gate PORT(
    i1, i2: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT or2gate PORT(
    i1, i2: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;

  SIGNAL sn, snd0, sd1: STD_LOGIC;

BEGIN
  U1: notgate PORT MAP(s, sn);
  U2: and2gate PORT MAP(d0, sn, snd0);
  U3: and2gate PORT MAP(d1, s, sd1);
  U4: or2gate PORT MAP(snd0, sd1, y);
END Structural;
```

Figure 1.14 Structural VHDL code for the 2-to-1 multiplexer.

The focus of this book is not to teach the details of how to write Verilog or VHDL codes. Because these two hardware description languages are quite similar to high-level computer languages such as C, which you already should be familiar with, we will take the approach of learning by examples. Throughout the book, there are Verilog and VHDL codes for all of the circuits discussed, and in the appendix there is a syntax summary of the two languages for your reference. By looking at the examples and the summary references, you should be able to write codes using these languages to describe your digital circuits.

1.5 Synthesis

Given a gate-level circuit diagram, such as the one shown in Figure 1.7, you actually can get some discrete logic gates and manually connect them together with wires on a breadboard. Traditionally, this is how electronic engineers actually designed and implemented digital logic circuits. However, this is not how electronic engineers design circuits anymore. They write programs, such as the one in Figure 1.6, just like what computer programmers do. The question is how does the program that describes the operation of the circuit actually gets converted to the physical circuit?

The problem here is similar to translating a computer program written in a high-level language to machine language for a particular computer to execute. For a computer program, we use a compiler to do the translation. For translating a digital logic circuit, we use a **synthesizer**. Instead of using a high-level computer language to describe a computer program, we use a hardware description language (HDL) to describe the operations of a digital logic circuit. Writing a description of a digital logic circuit is similar to writing a computer program except that a different language is used. A synthesizer then is used to translate the HDL program into the circuit **netlist**. A netlist is a description of how a circuit actually is realized or connected using basic gates. This translation process from an HDL description of a circuit to its netlist is referred to as **synthesis**.

The netlist from the output of the synthesizer can be used directly to implement the actual circuit in a FPGA IC chip. With this final step, the creation of a digital circuit that is implemented fully in an IC chip can be done easily. The appendices give a tutorial of the complete process: from writing the Verilog or VHDL code to synthesizing the circuit and uploading the netlist to a FPGA chip using an FPGA development board.

1.6 Going Forward

We will now embark upon a journey that will take you from a simple transistor to the construction of a microprocessor. Figure 1.2 will serve as our guide and map. If you get lost on the way, and do not know where a particular component fits in the overall picture, just refer to this map. At the end, you will be able to design your own custom microprocessor. The exciting part is that this is not just talk and theories. You will be able to implement and try out all of the circuits on a real FPGA chip. You will be able to make lights flash, motors run, and execute your own software program on your own custom microprocessor.

Figure 1.15 is an actual picture of the circuitry inside an Intel Pentium 4 CPU. When you reach the end of this book, you still may not be able to design the circuit for the P4 because of lack of resources, but you certainly will know how it is designed, because you actually will have designed and implemented a real working microprocessor yourself.

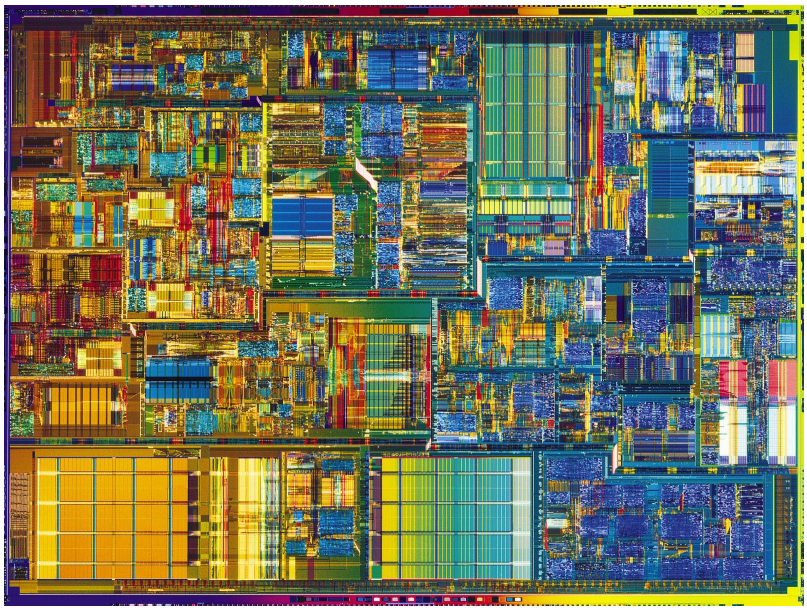


Figure 1.15. The internal circuitry of the Intel Pentium 4 CPU.

1.7 Problems

1.1. Find out the approximate number of general-purpose microprocessors sold in your country in the most recent year versus the number of dedicated microprocessors sold.

1.2. Compile a list of devices controlled by a microprocessor that you use during one regular day.

1.3. Describe what your regular daily routine would be like if no electrical power (including battery power) were available.

1.4. What are the inputs and outputs for the following systems?

a) Traffic light
b) Heart pacemaker
c) Microwave oven
d) Musical greeting card
e) Hard disk drive (not the entire personal computer)

1.5. The speed of a microprocessor is often measured by its clock frequency. What is the clock frequency of the fastest general-purpose microprocessor available today?

1.6. Compare some typical clock speeds between general-purpose microprocessors versus dedicated microprocessors.

1.7. Summarize the mainstream generations of the Intel general-purpose microprocessors used in personal computers starting with the 8086 CPU. List the year introduced, the clock speed, and the number of transistors in each.

1.8. The first-generation PC uses the Intel 8088 with approximately 29,000 transistors. Approximately how many transistors are in the Intel Core i7 (Quad) CPU? Approximately how many transistors are in the Xilinx Virtex-7 FPGA? Approximately how many transistors are in the Altera Stratix V FPGA?

1.9. Using Figure 1.11 as a template, write the dataflow Verilog code for the 2-to-1 multiplexer circuit shown in Figure 1.7(a).

1.10. Using Figure 1.13 as a template, write the structural Verilog code for the 2-to-1 multiplexer circuit shown in Figure 1.7(a).

1.11. Using Figure 1.10 as a template, write the dataflow VHDL code for the 2-to-1 multiplexer circuit shown in Figure 1.7(a).

1.12. Using Figure 1.14 as a template, write the structural VHDL code for the 2-to-1 multiplexer circuit shown in Figure 1.7(a).

1.13. Do the Xilinx ISE tutorial in Appendix A.

1.14. Do the Altera Quartus tutorial in Appendix B.

Index

V

