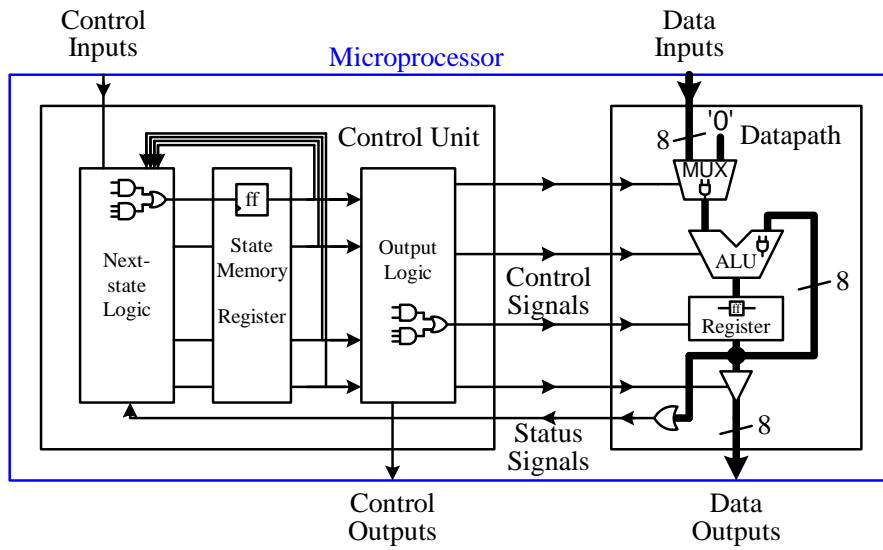


Contents

Designing Microprocessors	2
1.1 Overview of a Microprocessor	3
1.2 Design Abstraction Levels.....	6
1.3 Examples of a 2-to-1 Multiplexer	6
1.3.1 Behavioral Level.....	7
1.3.2 Gate Level.....	8
1.3.3 Transistor Level.....	9
1.4 Introduction to VHDL and Verilog.....	10
1.5 Synthesis.....	13
1.6 Going Forward.....	13
1.7 Summary Checklist.....	14
1.8 Problems	14
Index.....	16

Chapter 1

Designing Microprocessors



Being a computer science or electrical engineering student, you probably have assembled a PC before. You may have gone out to purchase the motherboard, central processing unit (CPU), memory, disk drive, video card, sound card, and other necessary parts, assembled them together, and have made yourself a state-of-the-art working computer. But have you ever wondered how the circuits inside those integrated circuit (IC) chips are designed? You know how the PC works at the system level by installing the operating system and seeing your machine come to life. But have you thought about how your PC works at the circuit level, how the memory is designed, or how the CPU circuit is designed?

In this book, I will show you from the ground up, how to design the digital circuits for **microprocessors**, also known as CPUs. When we hear the word “microprocessor,” the first thing that probably comes to many of our minds is the Intel Pentium® CPU, which is found in most PCs. However, there are many more microprocessors that are not Pentiums, and many more microprocessors that are used in areas other than the PCs.

Microprocessors are the heart of all “smart” devices, whether they be electronic devices or otherwise. Their smartness comes as a direct result of the decisions and controls that microprocessors make. For example, we usually do not consider a car to be an electronic device. However, it certainly has many complex smart electronic systems, such as the anti-lock brakes and the fuel-injection system. Each of these systems is controlled by a microprocessor. Yes, even the black, hardened blob that looks like a dried-up and pressed-down piece of gum inside a musical greeting card is a microprocessor.

There are generally two types of microprocessors: **general-purpose microprocessors** and **dedicated microprocessors**. General-purpose microprocessors, such as the Pentium CPU, can perform different tasks under the control of software instructions. General-purpose microprocessors are used in all personal computers.

Dedicated microprocessors, also known as **application-specific integrated circuits (ASICs)**, on the other hand, are designed to perform just one specific task. For example, inside your cell phone, there is a dedicated microprocessor that controls its entire operation. The embedded microprocessor inside the cell phone does nothing else but control the operation of the phone. Dedicated microprocessors are, therefore, usually much smaller and not as complex as general-purpose microprocessors. However, they are used in every smart electronic device, such as the musical greeting cards, electronic toys, TVs, cell phones, microwave ovens, and anti-lock break systems in your car. From this short list, I’m sure that you can think of many more devices that have a dedicated microprocessor inside them. Although the small dedicated microprocessors are not as powerful as the general-purpose microprocessors, they are being sold and used in a lot more places than the powerful general-purpose microprocessors that are used in personal computers.

Designing and building microprocessors may sound very complicated, but don’t let that scare you, because it is not really all that difficult to understand the basic principles of how microprocessors are designed. We are not trying to design a Pentium microprocessor here, but after you have learned the material presented in this book, you will have the basic knowledge to understand how it is designed.

This book will show you in an easily understandable approach, starting with the basics and leading you through to the building of larger components, such as the arithmetic logic unit (ALU), register, datapath, control unit, and finally to the building of the microprocessor—first dedicated microprocessors, and then general-purpose microprocessors. Along the way, there will be many sample circuits that you can try out and actually implement in hardware using the optional Altera UP2 development board. These circuits, forming the various components found inside a microprocessor, will be combined together at the end to produce real, working microprocessors. Yes, the exciting part is that at the end, you actually can implement your microprocessor in a real IC and see that it really can execute software programs or make lights flash!

1.1 Overview of a Microprocessor

The Von Neumann model of a computer, shown in Figure 1.1, consists of four main components: the input, the output, the memory, and the microprocessor (or CPU). The parts that you purchased for your computer can all be categorized into one of these four groups. The keyboard and mouse are examples of input devices. The CRT (cathode ray tube) and speakers are examples of output devices. The different types of memory (cache, read-only memory (ROM), random-access memory (RAM), and disk drive) are all considered part of the memory box in the model. In this book, the focus is not on the mechanical aspects of the input, output, and storage devices. Rather, the

focus is on the design of the digital circuitry of the microprocessor, the memory, and other supporting digital logic circuits.

The logic circuit for the microprocessor can be divided into two parts: the **datapath** and the **control unit**, as shown in Figure 1.1. Figure 1.2 shows the details inside the control unit and the datapath. The datapath is responsible for the actual execution of all data operations performed by the microprocessor, such as the addition of two numbers inside the arithmetic logic unit (ALU). The datapath also includes registers for the temporary storage of your data. The functional units inside the datapath, which in our example includes the ALU and the register, are connected together with multiplexers and data signal lines. The data signal lines are for transferring data between two functional units. Data signal lines in the circuit diagram are represented by lines connecting two functional units. Sometimes, several data signal lines are grouped together to form a **bus**. The width of the bus (that is, the number of data signal lines in the group) is annotated next to the bus line. In the example, the bus lines are thicker and are 8-bits wide. Multiplexers, also known as MUXs, are for selecting data from two or more sources to go to one destination. In the sample circuit, 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. The tri-state buffer is used to control the output of the data from the register.

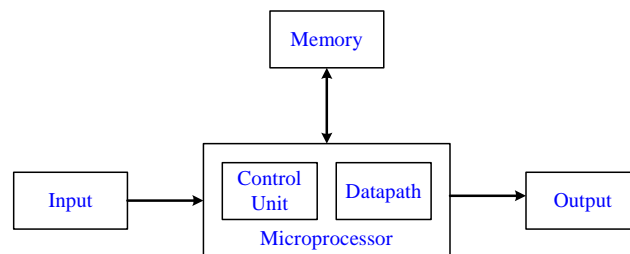


Figure 1.1. Von Neumann model of a computer.

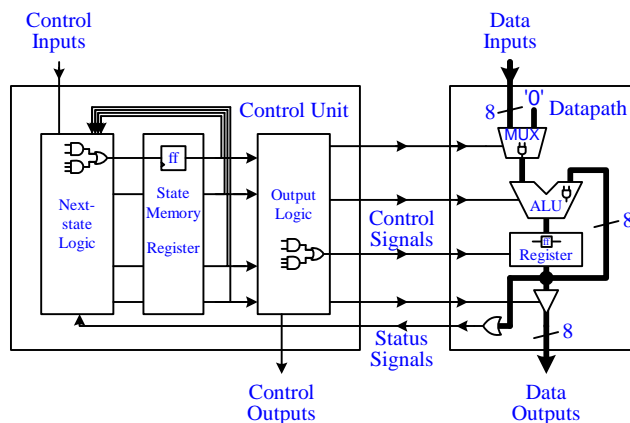


Figure 1.2. Internal parts of a microprocessor.

Even though the datapath is capable of performing all of the data operations of the microprocessor, it cannot, however, do it on its own. In order for the datapath to execute the operations automatically, the 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 a **finite state machine (FSM)** because it is a machine that executes by going from one state to another and that 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 for determining what the next state should be for the machine. The output logic is the circuit for generating the actual control signals for controlling the datapath.

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. For example, an adder circuit is a combinational circuit. It takes two numbers as inputs. The adder evaluates the sum of these two numbers and outputs the result.

A sequential circuit, on the other hand, is dependent not only on the current inputs, but also on all the previous inputs. In other words, a sequential circuit has to remember its past history. For example, the up-channel button on a TV remote is part of a sequential circuit. Pressing the up-channel button is the input to the circuit. However, just having this input is not enough for the circuit to determine what TV channel to display next. In addition to the up-channel button input, the circuit must also know the current channel that is being displayed, which is the history. If the current channel is channel 3, then pressing the up-channel button will change the channel to channel 4.

Since sequential circuits are dependent on the history, they must therefore contain memory elements for remembering the history; whereas combinational circuits do not have memory elements. Examples of combinational circuits inside the microprocessor include the next-state logic and output logic in the control unit, and the ALU, multiplexers, tri-state buffers, and comparators in the datapath. Examples of sequential circuits include the register for the state memory in the controller and the registers in the datapath. The memory in the Von Neuman computer model is also a sequential circuit.

Regardless of whether a circuit is combinational or sequential, they are all 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 just electronic binary switches that can be turned on or off. The on and off states of a transistor are used to represent the two binary values: 1 and 0.

Figure 1.3 summarizes how the different parts and components fit together to form the microprocessor. From transistors, the basic logic gates are built. Logic gates are combined 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. Latches and flip-flops are the simplest forms of sequential circuits, and they provide the basic building blocks for more complex sequential circuits. Certain combinational circuits and sequential circuits are used as standard building blocks for larger circuits, such as the microprocessor. These standard combinational and sequential components usually are found in standard libraries and serve as larger building blocks for the microprocessor. Different combinational components and sequential components are connected together to form either the datapath or the control unit of a microprocessor. Finally, combining the datapath and the control unit together will produce the circuit for either a dedicated or a general-purpose microprocessor.

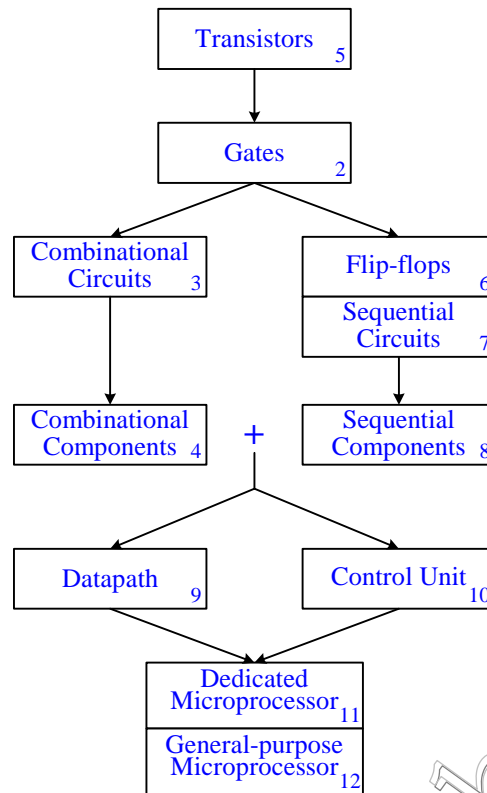


Figure 1.3. Summary of how the parts of a microprocessor fit together. The numbers in each box denote the chapter number in which the topic is discussed.

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. At the gate level, you also can specify the circuit using either a truth table or a Boolean equation. 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** in building 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, which is the **behavioral level**, we construct the circuit by describing the behavior or operation of the circuit using a hardware description language. This is very similar to writing a computer program using a programming language.

1.3 Examples of a 2-to-1 Multiplexer

As an example, let us look at the design of the 2-to-1 multiplexer from the 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 is intended just to give you an idea of what the description of the circuits look like at the different abstraction levels. We will get to the details in the rest of the book.

An important point to gain from these examples is to see 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 size (how big the circuit is or how many transistors 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, besides being functionally correct, there will always be economic versus performance tradeoffs that we need to consider.

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 .

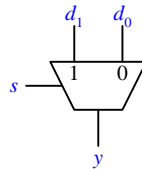


Figure 1.4. Logic symbol for the 2-to-1 multiplexer.

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) to describe a 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. There are two popular hardware description languages, VHDL and Verilog. Figure 1.5 shows the behavioral description of the 2-to-1 multiplexer using VHDL.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY multiplexer IS PORT (
    d0, d1, s: 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;

```

```

    END IF;
  END PROCESS;
END Behavioral;

```

Figure 1.5. Behavioral level VHDL description of the 2-to-1 multiplexer.

The LIBRARY and USE statements are similar to the “#include” preprocessor command in C. 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. The ARCHITECTURE section defines the actual operation of the circuit. The operation of the multiplexer is defined in the conditional IF-THEN-ELSE statement:

```

IF (s = '0') THEN
  y <= d0;
ELSE
  y <= d1;
END IF;

```

The two signal assignment statements, which uses 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 .

Figure 1.6 shows the behavioral description of the same 2-to-1 multiplexer but using the Verilog hardware description language. Comparing between the VHDL and the Verilog version of the code, one can quickly see the similarities. The PROCESS block in the VHDL code is replaced by the ALWAYS block in the Verilog. The tilde symbol (~) is the NOT operator.

```

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

  always @(s, d0, d1) begin
    if (~s)
      y = d0;
    else
      y = d1;
    end
  endmodule

```

Figure 1.6. Behavioral level Verilog description of the 2-to-1 multiplexer.

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 operation.

1.3.2 Gate Level

At the gate level, you can draw a **schematic diagram**, which is a diagram showing how the logic gates are connected together. Two schematic diagrams of a circuit are shown in Figure 1.7(a) and (b). In Figure 1.7(a), the circuit uses three INVERTERS (\neg), four 3-input AND gates (\exists), and one 4-input OR gate (\exists). In Figure 1.7(b), only one INVERTER, 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.

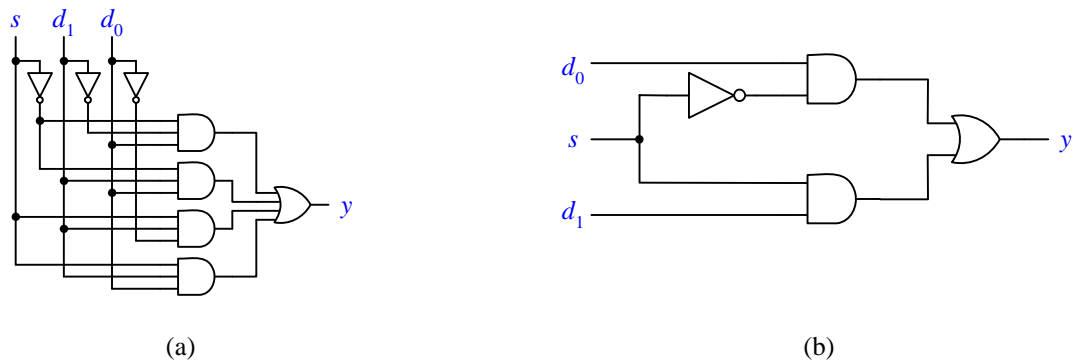


Figure 1.7. Gate level circuit diagram for the 2-to-1 multiplexer: (a) circuit using eight gates; (b) circuit using four gates.

At the gate level, you can also 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 (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 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.

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)

$$\begin{aligned}
 y &= s' d_1' d_0 + s' d_1 d_0 + s d_1 d_0' + s d_1 d_0 \\
 &= s' d_0 + s d_1
 \end{aligned}$$

(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 PMOS ($\overline{\text{P}}$), and three are NMOS (N). The pair of transistors on the left forms an inverter 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 .

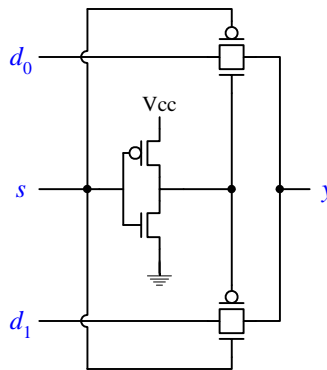


Figure 1.9. Transistor circuit for the 2-to-1 multiplexer.

1.4 Introduction to VHDL and Verilog

The popularity of using hardware description languages (HDL) for designing digital circuits began in the mid-1990s when commercial synthesis tools became available. Two popular hardware design languages, used by many engineers today are VHDL and Verilog. VHDL, which stands for VHSIC Hardware Description Language (VHSIC, in turn, stands for Very High Speed Integrated Circuit), was sponsored and developed jointly by the U.S. Department of Defense and the 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. In this book, we will show many of the examples in both VHDL and Verilog.

Both VHDL and Verilog, in many respects, is 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 VHDL and Verilog, 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.

You saw in Section 1.3.1 how we used VHDL and Verilog to describe the 2-to-1 multiplexer at the behavioral level. You can also use either languages to describe a circuit at other levels. Figure 1.10 shows the VHDL code for the multiplexer written at the **dataflow** level, and the corresponding Verilog code in Figure 1.11. The main difference between the behavioral VHDL code shown in Figure 1.5 and the dataflow VHDL code in Figure 1.10 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 like 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 \leftarrow , 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(
  d0, d1, s: 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.

In the Verilog version shown in Figure 1.11, the ASSIGN keyword is used for the signal assignment in conjunction with the Verilog built-in logical operators &, |, and ~ for the operation and, or, and not respectively.

```
module multiplexer
  (input d0, d1, s,
   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 can also write HDL code at the **structural** level. Figure 1.13 and Figure 1.14 show the VHDL and Verilog code, respectively, for the multiplexer written at the structural level. The code is based on the circuit shown in Figure 1.12. In the VHDL code, the three different gates (*and2gate*, *or2gate*, and *notgate*) used in the circuit first are declared and defined 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 *multiplexer2*. First of all, the COMPONENT statements specify what components are used in the circuit. The SIGNAL statement declares 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 also specify how they are connected using the external and internal signals.

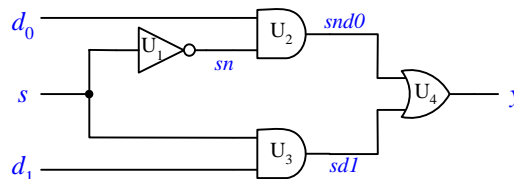


Figure 1.12. 2-to-1 multiplexer circuit.

```
----- 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(
  d0, d1, s: 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 1.13. Structural level VHDL description of the 2-to-1 multiplexer.

```

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

  wire sn, snd0, sd1;

  not
  U1(sn, s);

```

```
and
  U2(snd0,d0,sn),
  U3(sd1,d1,s);
or
  U4(y,snd0,sd1);
endmodule
```

Figure 1.14. Structural level Verilog description of the 2-to-1 multiplexer.

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 1.5, just like what computer programmers do. The question then 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; the only difference is that a different language is used. A synthesizer is then 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**.

Furthermore, the netlist from the output of the synthesizer can be used directly to implement the actual circuit in a programmable logic device (PLD) chip, such as a field programmable gate array (FPGA). With this final step, the creation of a digital circuit that is implemented fully in an integrated circuit (IC) chip can be done easily. The Appendices give a tutorial of the complete process: from writing the HDL code to synthesizing the circuit and uploading the netlist to the FPGA chip using Altera's development system.

1.6 Going Forward

We will now embark upon a journey that will take you from a simple transistor to the building 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 beginning of each chapter, I will refresh your memory with this map by highlighting the components in the map that the chapter will cover.

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, but you certainly will have the knowledge of how a microprocessor is designed, because you actually will have designed and implemented a working microprocessor yourself.

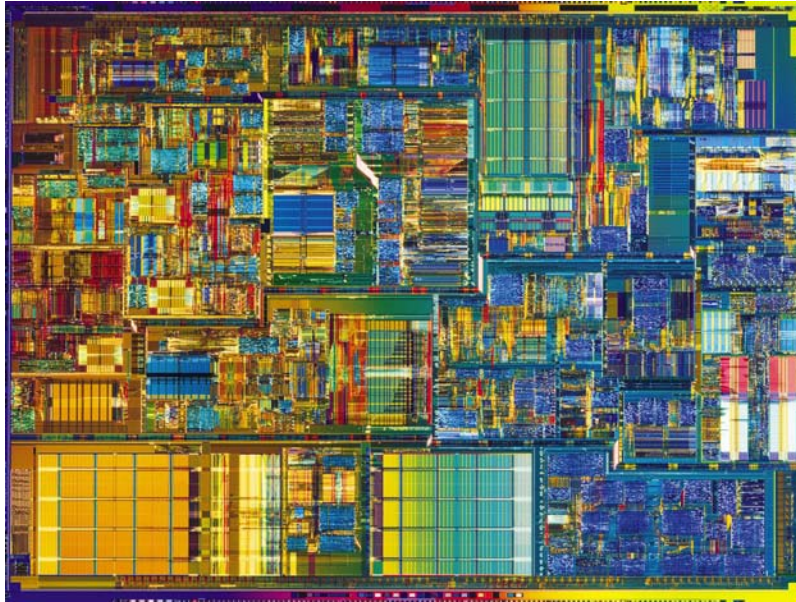


Figure 1.15. The internal circuitry of the Intel P4 CPU.

1.7 Summary Checklist

- Microprocessor
- General-purpose microprocessor
- Dedicated microprocessor
- ASIC
- Datapath
- Control unit
- Finite state machine (FSM)
- Next-state logic
- State memory
- Output logic
- Combinational circuit
- Sequential circuit
- Transistor level design
- Gate level design
- Register-transfer level design
- Behavioral level design
- Logic symbol
- VHDL
- Synthesis
- Netlist

1.8 Problems

- 1.1. Find out the approximate number of general-purpose microprocessors sold in the US in a year versus the number of dedicated microprocessors sold.
- 1.2. Compile a list of devices that you use during one regular day that are controlled by a microprocessor.
- 1.3. Describe what your regular daily routine will be like if there is no electrical power (including battery power)

available.

- 1.4. Apply the Von Neumann model of a computer system, as shown in Figure 1.1, to the following systems. Determine what parts of the system correspond to the different parts of the model.
 - 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?
- 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.

Answer

CPU	Year Introduced	Clock Speed	Number of Transistors
8086	1978	4.7 – 10 MHz	29,000
80286	1982	6 – 12 MHz	134,000
80386	1985	16 – 33 MHz	275,000
80486	1989	25 – 100 MHz	1.2 million
Pentium	1993	60 – 200 MHz	3.3 million
Pentium Pro	1995	150 – 200 MHz	5.5 million
Pentium II	1997	234 – 450 MHz	7.5 million
Celeron	1998	266 – 800 MHz	19 million
Pentium III	1999	400 MHz – 1.2 GHz	28 million
Pentium 4	2000	1.4 – 3 GHz	42 million

- 1.8. 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).
- 1.9. Using Figure 1.13 as a template, write the structural VHDL code for the 2-to-1 multiplexer circuit shown in Figure 1.7(a).
- 1.10. Do Tutorial 1 in Appendix A.
- 1.11. Do Tutorial 2 in Appendix B.
- 1.12. Do Tutorial 3 in Appendix C.

Index**A**

Abstraction level. *See* Design abstraction levels.
 Application-specific integrated circuit, 3
 ASIC. *See* Application-specific integrated circuit

B

Behavioral level (VHDL), 6, 7
See also Design abstraction levels. ,
 Bus, 4

C

Combinational circuit, 5
 Control unit. *See* Finite state machine.

D

Datapath, 4
 Dedicated microprocessor, 3
 Design abstraction levels, 6
 behavioral level, 6
 gate level, 6
 register-transfer level, 6
 RTL. *See* Register-transfer level.
 transistor level, 6

F

Field programmable gate array, 13
 Finite state machine, 4
 FPGA. *See* Field programmable gate array.
 FSM. *See* Finite state machine.

G

Gate, 5
 Gate level, 6, 8
See also Design abstraction levels.
 General-purpose microprocessor, 3

H

Hardware description language, 7, 10, 13
 Verilog, 10
 VHDL, 10
 HDL. *See* Hardware description language

L

Logic gate, 5

Logic symbol, 7

M

Microprocessor, 3
 dedicated, 3
 general-purpose, 3

N

Netlist, 10, 13
 Next-state logic, 4
See also Finite state machine.

O

Output logic, 4
See also Finite state machine.

R

Register-transfer level, 6
See also Design abstraction levels.
 RTL. *See* Register-transfer level.

S

Schematic diagram, 8
 Sequential circuit, 5
 State memory, 4
See also Finite state machine.
 Structural level (VHDL), 11
See also Design abstraction levels.
 Synthesis, 13
 Synthesizer, 10, 13

T

Transistor, 5
 Transistor level, 6, 9
See also Design abstraction levels.

V

Verilog, 10
 VHDL, 7, 10
 behavioral level, 8
 dataflow level, 11
 structural level, 12, 13