specification document.

Once the problem has been stated clearly, often designers start thinking about the basic blocks necessary to accomplish what is specified. Designers often think of standard building blocks, such as adders, shift-registers, counters, and the like. Traditional design methodology splits a design into a "data path" and a "controller." The term "data path" refers to the hardware that actually performs the data processing. The controller sends control signals or commands to the data path, as shown in Figure 4-1. The controller can obtain feedback in the form of status signals from the data path.

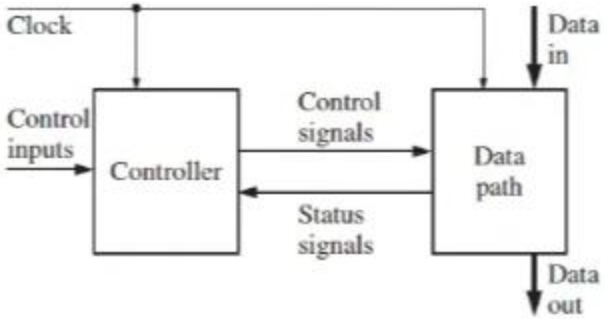


FIGURE 4-1: Separation of a Design into Data Path and Controller

In the context of a microprocessor, the data path is the ALU (Arithmetic and Logic Unit) that performs the core of the processing. The controller is the control logic that sends appropriate control signals to the data path, instructing it to

In the context of a microprocessor, the data path is the ALU (Arithmetic and Logic Unit) that performs the core of the processing. The controller is the control logic that sends appropriate control signals to the data path, instructing it to

perform addition, multiplication, shifting, or whatever action is called for by the instruction. Many users have a tendency to mistakenly consider the term "data path" to be synonymous with the data bus, but "data path" in traditional design terminology refers to the actual data processing unit.

Maintaining a distinction between data path and controller helps in debugging (i.e., finding errors in the design). It also helps while modifying the design. Many modifications can be accomplished by changing only the control path because the same data path can still support the new requirements. The controller can generate the new sequence of control signals to accomplish the functionality of the modified design. Design often involves refining the data path and controller in iterations.

the modified design. Design often involves refining the data path and controller in iterations.

In this chapter, we will discuss various design examples. Several arithmetic and non-arithmetic examples are presented. Non-arithmetic examples include a 7-segment decoder, a traffic light, a scoreboard, and a keypad scanner. Arithmetic circuits such as adders, multipliers, and dividers are also presented.

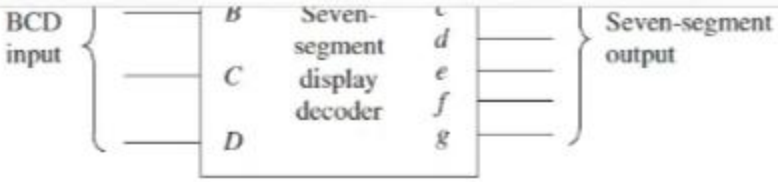
4.1 BCD to 7-Segment Display Decoder

Seven segment displays are often used to display digits in digital counters, watches, and clocks. A digital watch displays time by turning on a combination of the segments on a 7-segment display. For this example, the segments are labeled as follows, and the digits have the forms as indicated in Figure 4-2.

IGURE 4-2: 7-Segment
)isplay

URA R

Let us design a BCD to 7-segment display decoder. BCD stands for "binary coded decimal." In this format, each digit of a decimal number is encoded into



We will create a behavioral Verilog architectural description of this BCD to 7-segment decoder by using a single process with a case statement to model this

combinational circuit, as in Figure 4-4. The sensitivity list of the process consists of the BCD number (4 bits).

FIGURE 4-4: Behavioral Verilog Code for BCD to 7-Segment Decoder

```
module bcd_seven (bcd, seven);
  input [3:1] bcd;
  output[7:1] seven;
```

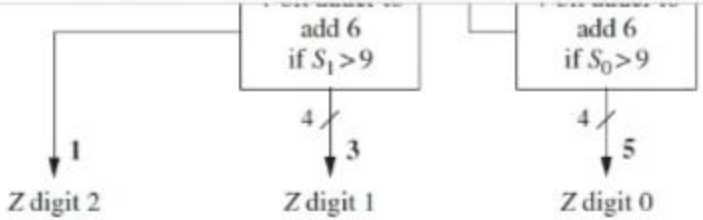
```
module bcd_seven (bcd, seven);
  input [3:0] bcd;
  output[7:1] seven;

  reg   [7:1] seven;

  always @(bcd)
  begin
    case (bcd)
      4'b0000 : seven = 7'b0111111 ;
      4'b0001 : seven = 7'b0000110 ;
      4'b0010 : seven = 7'b1011011 ;
      4'b0011 : seven = 7'b1001111 ;
      4'b0100 : seven = 7'b1100110 ;
      4'b0101 : seven = 7'b1101101 ;
      4'b0110 : seven = 7'b1111101 ;
      4'b0111 : seven = 7'b0000111 ;
      4'b1000 : seven = 7'b1111111 ;
      4'b1001 : seven = 7'b1101111 ;
      default : seven = 7'b0000000 ;
    endcase
  end
endmodule
```

4.2 A BCD Adder

In this example, we design a 2-digit BCD adder, which will add two BCD numbers and produce the sum in BCD format. In BCD representation, each decimal digit is encoded into binary. For instance, decimal number 97 will be represented as 1001 0111 in the BCD format, where the first 4 bits represent digit 9 and the next 4 bits represent digit 7. One may note that the BCD representation is different from the binary representation of 97, which is 110001. It takes 8 bits to represent 97 in BCD, whereas the binary representation of 97 (110001) only requires 6 bits. The 4-bit binary combinations 1010, 1011, 1100, 1101, 1110, and 1111 corresponding to hexadecimal numbers *A* to *F* are not used in the BCD representation. Since 6 out of 16 representations possible with 4 binary bits are skipped, a BCD number will take more bits than the corresponding binary representation.

When BCD numbers are added, each sum digit should be adjusted to skip the six unused codes. For instance, if 6 is added with 8, the sum is 14 in decimal form. A binary adder would yield 1110, but the lowest digit of the BCD sum should read 4. In order to obtain the correct BCD digit, 6 should be added to the sum whenever it is greater than 9. Figure 4-5 illustrates the hardware that will be required to

URA R



perform the addition of 2 BCD digits. A binary adder adds the least significant digits. If the sum is greater than 9, an adder adds 6 to yield the correct sum digit along with a carry digit to be added with the next digit. The addition of the higher digits is performed in a similar fashion.

The Verilog code for the BCD adder is shown in Figure 4-6. The input BCD numbers are represented by *X* and *Y*. The BCD sum of two 2-digit BCD numbers can exceed two digits and hence three BCD digits are provided for the sum, which is represented by *Z*. The compiler directive 'define' can be used to denote each digit of each BCD number. For example, the upper digit of *X* can be denoted by *Xdig1* by using the Verilog statement:

```
`define      Xdig1      X[7:4]
```

This statement allows us to use the name *Xdig1* whenever we wish to refer to the upper digit of *X*. If BCD numbers 97 and 38 are added, the sum is 135; hence, *Zdig2*

```
input[7:0] X;
input[7:0] Y;
output[11:0] Z;
```

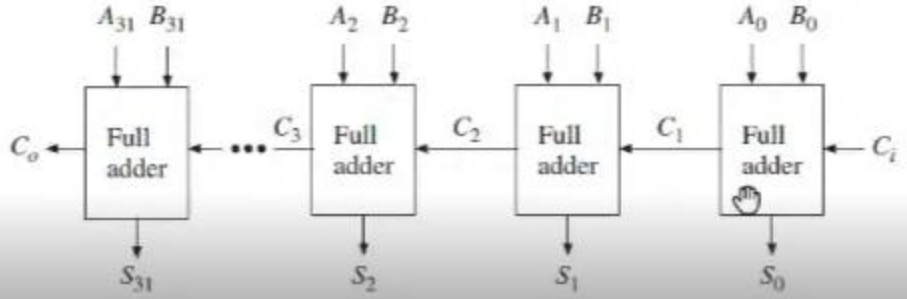


```
wire[4:0] S0;
wire[4:0] S1;
wire C;

assign S0 = `Xdig0 + `Ydig0 ;
assign `Zdig0 = (S0 > 9) ? S0[3:0] + 6 : S0[3:0] ;
assign C = (S0 > 9) ? 1'b1 : 1'b0 ;

assign S1 = `Xdig1 + `Ydig1 + C ;
assign `Zdig1 = (S1 > 9) ? S1[3:0] + 6 : S1[3:0] ;
assign `Zdig2 = (S1 > 9) ? 4'b0001 : 4'b0000 ;
```

4.3 32-Bit Adders

Let us assume that we have to design a 32-bit adder. A simple way to construct an adder is to build a **ripple-carry adder**, as shown in Figure 4-7. In this type of adder, 32 copies of a one-bit full adder are connected in succession to create the 32-bit adder. The carry "ripples" from the least significant bit to the most significant bit. If gate delays are t_g, a one-bit adder delay is $2*t_g$ (assuming a sum-of-products expression for sum and carry, and ignoring delay for inverters), and a 32-bit ripple-carry adder will take approximately 64 gate delays. For instance, if gate delays are 1 ns, the maximum frequency at which the 32-bit ripple-carry adder can operate is approximately 16 MHz! This is inadequate for many applications. Hence, designers often resort to faster adders.

-7: A 32-Bit
arry Adder



RA R

Carry Look-Ahead Adders