# Digital Electronics

Supplement to Engs 31 / CoSc 56

# Digital Electronics

Supplement to Engs 31 / CoSc 56

Eric W. Hansen
Dartmouth College

December 17, 2024

Eric W. Hansen (BS, Caltech; PhD, Stanford) is Associate Professor of Engineering Sciences, Emeritus, at Dartmouth. He is also author of the textbook "Fourier Transforms: Principles and Applications" (Wiley, 2014). He holds amateur radio license KB1VUN and continues to experiment with electronics, signal processing, and communications.

**Edition**: Annual Edition 2024

**Website**: http://www.dartmouth.edu/~engs31book/

# Acknowledgements

Thanks to Robert Beezer and all the dedicated members of the PreTeXt community for creating the PreTeXt tools (http://pretextbook.org[1]), which make it possible to quickly output print, web, PDF versions and more from the same source.

[1] pretextbook.org

# Contents

# Part I

# Logic Design Basics

# Chapter 1

# The world of 1s and 0s

## 1.1 Bits, codes, and numbers

**Simple binary choices.** The most basic idea in digital electronics is to represent the world using two values, 0 and 1. Here are some examples:

- A claim, such as "It is raining outside" or "Jupiter is bigger than Mars", is either TRUE (1) or FALSE (0).
- The state of a light (or motor, or alarm) is either ON (1) or OFF (0).
- On early computer displays, each pixel was either BLACK (0) or WHITE (1).
- A clock display includes a light that is off (0) for AM, and on (1) for PM.

A two-alternative choice (1 or 0) is said to be **binary**. A single 1 or 0 is called a **bit** (a contraction of *binary* and *digit*). Representing the state of a light by 0 for OFF and 1 for ON uses one bit.

**Checkpoint 1.1.1** What other examples of binary states in the world can you think of?

**Binary vectors.** When we need to represent more than two alternatives we can use more than one bit:

- A thermostat could express the temperature of a room as TOO HOT (10), TOO COLD (01), or JUST RIGHT (00). (The remaining combination "11" could perhaps be assigned to signal a malfunction.)
- A pair of bits can be used to control the headlights on a car: OFF (00), LOW BEAM (01), HIGH BEAM (11). The high beam is always on together with the low beam; the combination "10" would be unused.
- Four shades of gray on a computer screen—WHITE, LIGHT GRAY, DARK GRAY, BLACK—can be represented by "11", "10", "01", and "00", respectively.
- With three bits we can represent colors as mixtures of the primary colors red, green, and blue. Define RED by "100", GREEN by "010", and BLUE by "001". The combination of RED and BLUE is "101". Eight colors are possible in all.

| Color | Binary code (RGB) |
|---|---|
| BLACK | 000 |
| BLUE | 001 |
| GREEN | 010 |
| CYAN | 011 |
| RED | 100 |
| MAGENTA | 101 |
| YELLOW | 110 |
| WHITE | 111 |

**Figure 1.1.2** Binary representations for a common color palette.

Groups of bits are called binary **vectors**, or bit vectors.

In the table above, the vector "100" is the **binary code** for the color RED. In general, if you have a vector consisting of $N$ bits, you can represent $2^N$ different binary codes and encode up to $2^N$ different possibilties.

**Checkpoint 1.1.3** What other examples of multi-bit states in the world can you think of?

**Checkpoint 1.1.4** If you have an image display where each pixel is encoded by an 8-bit vector, how many shades of gray, from white to black, can you represent?

**Answer**. With 8 bits, there can be $2^8 = 256$ shades of gray.

**Checkpoint 1.1.5** If you have an image display where each pixel is encoded by 24 bits (eight for red, eight for green, and eight for blue), how many different colors can you make?

**Answer**. With 24 bits, there can be $2^{24} = 2^4 \cdot 2^{10} \cdot 2^{10}$ colors. Now, $2^{10} = 1024 \approx 1000$, so the number of colors is approximately 16 million.

**Checkpoint 1.1.6** How many bits should be required to encode all the alphanumeric characters 0..9, a..z, and A..Z? (We aren't counting punctuation marks or special characters like "%".)

**Answer**. There are 62 alphanumeric characters. The next power of two after 62 is $64 = 2^6$. Thus, six bits should be necessary. To include punctuation requires 7 bits. If you want to know how it's really done, punctuation and all, visit the website www.asciitable.com[1].

**Binary numbers.** Binary codes can be used to represent numbers. We'll just do nonnegative numbers here, and introduce codes for negative numbers later. With three bits, we can represent the decimal numbers from 0 to 7, like this:

| Decimal | Binary |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

**Figure 1.1.7** Binary representations for the numbers 0–7

[1] `www.asciitable.com`

The digits in a decimal number, from right to left, correspond to powers of ten (1, 10, 100...), and the system of decimal numbers is called "base 10". Binary numbers obey place value just like decimal numbers, but based on powers of two rather than ten: from right to left, the bits correspond to powers of two (1, 2, 4...), and the system of binary numbers is called "base 2".

In a decimal number, the leftmost digit, the one with the highest place value, is called the most significant digit. The rightmost digit, with the lowest place value, is called the least significant digit. Likewise, in a binary number, the leftmost bit is the **most significant bit** (msb), and the rightmost bit is the **least significant bit** (lsb). The bits of a number can be enumerated according to their significance, e.g., $\mathbf{y} = (y_3, y_2, y_1, y_0)$: $y_3$ is the msb, and $y_0$ is the lsb. Unlike the usual vector or n-tuple you may have seen in algebra or geometry, the indices of the bits in a binary number descend from 3 down to 0 rather than ascend from 0 to 3.

To facilitate interpretation of binary numbers, it's a good idea to memorize a few powers of two.

| Power | Decimal | Power | Decimal |
|---|---|---|---|
| $2^0$ | 1 | $2^{10}$ | 1024, "1K" |
| $2^1$ | 2 | $2^{11}$ | 2048, "2K" |
| $2^2$ | 4 | $2^{12}$ | 4096, "4K" |
| $2^3$ | 8 | $2^{13}$ | 8192, "8K" |
| $2^4$ | 16 | $\vdots$ | |
| $2^5$ | 32 | $2^{20}$ | $1024^2$, "1M" |
| $2^6$ | 64 | $\vdots$ | |
| $2^7$ | 128 | $2^{30}$ | $1024^3$, "1G" |
| $2^8$ | 256 | $\vdots$ | |
| $2^9$ | 512 | $2^{40}$ | $1024^4$, "1T" |

**Figure 1.1.8** Some powers of two. $2^{10} = 1024$ is approximately a thousand, and abbreviated "1K" (kilo). Likewise, $2^{20}$, $2^{30}$, and $2^{40}$ are approximately a million, a billion, and a trillion, respectively, and are often abbreviated "1M" (mega), "1G" (giga), and "1T" (tera).

Knowing these (or having the appropriate functions on your calculator), you can convert numbers from decimal to binary and back.

**Checkpoint 1.1.9** What decimal number is represented by the binary number 101101?

**Answer**. $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32 + 8 + 4 + 1 = 45$

**Checkpoint 1.1.10** What is the binary representation of the decimal number 75?

**Answer**.

$$\begin{aligned} 75 &= 64 + 8 + 2 + 1 \\ &= 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \end{aligned}$$

$\implies$ 75 decimal = 1001011 binary

**Binary, decimal, hexadecimal.** It can become tedious to write out all the 1s and 0s for even a moderately-sized binary number, not to mention being prone to error. A more compact way to write long binary numbers is to collect the bits into groups of four, and assign each four-bit code a unique character.

The digits 0-9 are fine for "0000" through "1001". After that we use the letters A-F to denote ten ("1010") through fifteen ("1111"), respectively. A character in this system has 16 possibilities, and each place in a number represents a power of 16: $16^0 = 1$, $16^1 = 16$, $16^2 = 256$... . This base-16 number system is called **hexadecimal**. For example, decimal 108 is

$$108 = 64 + 32 + 8 + 4 = 1101100 = 0110\ 1100 = 6C$$

in hexadecimal. Going the other way, A5 in hexadecimal is $10 \cdot 16 + 5 \cdot 1 = 165$ in decimal.

When there is potential for confusion among binary, decimal, and hexadecimal, we can use subscripts, e.g., $36_{16} = 110110_2 = 54_{10}$. Another way is to precede a hexadecimal number by x or 0x, a decimal number by d or 0d, and a binary number by b or 0b: 0x36 = 0b00110110 = 0d54.

## 1.2 Data bits, control bits

When we use bits to encode (that is, represent) information— e.g., text, images, music, measurements— we say those bits are encoding **data**. Data can come from many sources—a camera, a microphone, a thermometer, a keyboard. It can have more or less personal significance: it could be a voicemail with a job offer from that company you want to work for, a letter from a dear friend, an e-card from your sister on your birthday, or... it could be spam. But at the level of bits, it's all data.

We also use bits to turn lights on and off, to open and close windows, to lock and unlock doors, to start and stop motors. We call these **control** bits. Control bits also come from a variety of sources. At its simplest, we can flip a switch to **assert** a control bit (set it to 1). Control bits may also be a result of some processing of data.

- A digital thermostat measures the room temperature (data) and asserts a control bit if the room is too warm (thereby actuating the air conditioner).
- A camera acquires images of the road in front of you (data), and a driver-assist computer uses that data to assert a control signal that sounds an alarm or nudges the steering wheel if it detects you are drifting out of your lane.

This concept of data and control may be familiar from computer programming. Consider the following simple program.

```
if reset==1 then
    x = 0;
else
    x = x+1;
end if;

if x > 15 then
    alarm = 1;
else
    alarm = 0;
end if;
```

Here, the variable `x` is either assigned the value 0 or it is incremented. We consider `x` to be data. Which operation is performed on the data depends on the input `reset`, whether it is 1 or 0. Thus, we consider `reset` to be a control

bit. In the second part of the program, `x` is tested to see if it has exceeded a limiting value, 15. This is a data operation, to determine the status of `x`. Then, based on the result of the test, an output bit `alarm` is set to 1 or 0 (asserted or deasserted). Again, `x` is data, and `alarm` is a control bit. The example shows that control bits can either be inputs to a system that direct the processing of data (reset), or outputs of a system (alarm), resulting from the processing of data.

The "view from 20,000 feet" of a digital processor is a box that takes in data and control bits, and operates on the data to produce new data and control bits.

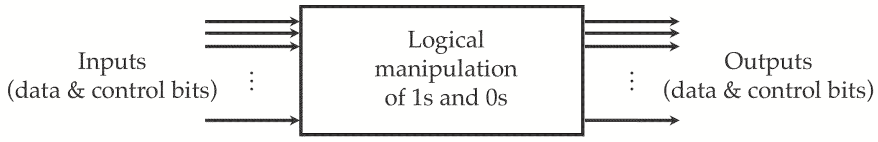



**Figure 1.2.1** A digital processor takes in data and control bits and produces data and control bits.

The processor contains four broad functions that we'll develop in detail as we proceed through the course:

1. Datapath, consisting of:
   (a) Memory: Where data is stored during computation
   (b) Operations: Arithmetic and logical manipulations of the data
2. Controller, which coordinates the operations of the datapath
3. Input/Output (I/O), which connect the processor to the outside world via sensors and actuators.

Depending on the system, some functions may be more or less prominent, but you will (almost) always be able to identify them, and they provide a valuable framework for conceptualizing and realizing a design. It is always good practice in digital design to maintain a clear distinction between data signals and control signals, and between data operations and control functions.

**Checkpoint 1.2.2** Pick a simple everyday activity, like waking up to your alarm clock, or buying breakfast at KAF. Can you identify data and control, maybe multiple instances of data and control, in the activity?

**Checkpoint 1.2.3** Consider this simple program,

```
if x>15 then
   x = 0;
else
   x = x+1;
end if;
```

There is only the data signal `x`. Identify the control functions and rewrite the program to use an explicit control signal.

**Answer**.

```
if x>15 then
   reset = 1;
```

```
else
   reset = 0;
end if;

if reset == 1 then
   x = 0;
else
   x = x+1;
end if;
```

## 1.3 Operating on bits with truth tables

The simplest digital processor simply takes in one or more input bits and maps them to an output bit. For example, take a two-bit number $x = (x_1, x_0)$ and assert the output, $y$, if $x < 2$. Using mathematical notation, we can express the input-output relationship as a function, $y = f(x)$, and write down what the function does in a table.

| $x$ (decimal) | $x_1$ | $x_0$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 |

**Figure 1.3.1** Representing the comparison operation $x < 2$ with a table.

A tabular representation of the mapping from input bits to output bits is called a **truth table**. Truth tables can have any number of inputs and outputs. Here is one with a two-bit input $n = (n_1, n_0)$ and a four-bit output $y = (y_3, y_2, y_1, y_0)$.

| $n$ (decimal) | $n_1$ | $n_0$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 0 | 0 |

**Figure 1.3.2** A truth table with two input bits and four output bits.

The function $y = f(n)$ asserts the $n^{th}$ output bit, $y_n$. It is called a **one-hot decoder** (because only one output is asserted, or "hot", at a time).

**Checkpoint 1.3.3** Write a truth table for a function $y = f(x)$ that asserts a single output bit if the two-bit input is an odd number (zero is neither even nor odd).

**Answer**.

| $x$ (decimal) | $x_1$ | $x_0$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 |

**Checkpoint 1.3.4** Write a truth table for a function $g = f(x, y)$ that has two two-bit inputs. The output bit $g$ is asserted if $x > y$.

**Answer**.

| $x_1$ | $x_0$ | $y_1$ | $y_0$ | $g$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

With this brief introduction, watch the beginning (up to time 2:21) of the following video on truth tables.

When you have both control inputs and data inputs, it is usually good practice to organize the truth table's input columns with the control inputs first, followed by the data inputs. For example, a small digital system, called a **multiplexer**, has two one-bit data inputs, $(A, B)$, a control bit $S$ ("select"), and a one-bit output, $y$. The control bit $S$ selects whether $y = A$ or $y = B$. The function of the multiplexer is described by the following if-then-else:

```
if S=='0'
    then y=A;
    else y=B;
end if;
```

The full truth table for a two-input multiplexer has $2^3 = 8$ rows. Following the recommended practice, the input columns are organized with with the select bit first, followed by the two data bits.

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

A truth table with $N$ inputs has $2^N$ rows. Exhaustively tabulating every row can rapidly become unwieldy as the number of inputs increases. Here is one way that truth tables can occasionally be compressed. Look again at the third and fourth rows of the multiplexer truth table:

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| | ⋮ | | ⋮ |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| | ⋮ | | ⋮ |

The output will be TRUE if $S$ is FALSE and $A$ is TRUE, whether $B$ is TRUE or FALSE. Because we don't care what the value of $B$ is, we merge the two rows, like this, marking the entry for $B$ with a dash (—):

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| | ⋮ | | ⋮ |
| 0 | 1 | — | 1 |
| | ⋮ | | ⋮ |

Additional rows can be merged using "don't care" entries, simplifying the truth table to four rows.

**Checkpoint 1.3.5** Complete the simplification of the truth table for the two-input multiplexer.

**Answer**. In addition to the third and fourth rows, the first and second, fifth and seventh, and sixth and eighth can be merged:

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| 0 | 0 | — | 0 |
| 0 | 1 | — | 1 |
| 1 | — | 0 | 0 |
| 1 | — | 1 | 1 |

Note how the if-then-else behavior is concisely captured by the compact truth table.

**Checkpoint 1.3.6** A digital system has four one-bit inputs, $(x_0, x_1, x_2, x_3)$, a two-bit binary data output $n = (n_1, n_0)$, and a one-bit control output $v$.

- If all the inputs are 0, then $v = 0$ and $n = 00$.
- If one or more of the inputs is 1, then $v = 1$ and $n$ is the largest index of the asserted inputs.

For example, if $x_2$ and $x_3$ are asserted, then $v = 1$ and $n = 11$ (3). Write a compact truth table for this function.

**Answer**.

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $v$ | $n_1$ | $n_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | — | 1 | 0 | 1 |
| 0 | 1 | — | — | 1 | 1 | 0 |
| 1 | — | — | — | 1 | 1 | 1 |

This function is called a **priority encoder**. The compact truth table reveals a nested if-then-else structure:

```
if x3==1 then
```

```
    n=3; v=1;
elsif x2==1 then
    n=2; v=1;
elsif x1==1 then
    n=1; v=1;
elsif x0==1 then
    n=0; v=1;
else
    n=0; v=0;
end if;
```

## 1.4 Operating on bits with logic equations

Truth tables are naturally suited to expressing if-then-else operations on bits, but they are not particularly convenient and it is perhaps hard to visualize how truth tables alone can ever enable the design of a complex digital processor like a calculator or a phone. As is often the case in engineering and science, mathematics supplies a toolkit to help us work efficiently. The toolkit is **Boolean algebra**, part of the larger discipline of mathematical logic.

> Named after English mathematician George Boole (1815-1864)

A **logical statement** in Boolean algebra consists of variables (truth table inputs) that are combined into an expression (truth table output) that is either TRUE (1) or FALSE (0) depending on the truth or falsity of the inputs.

The fundamental operations for combining variables in a logical statement are **AND**, **OR**, and **NOT**. The AND of two or more bits is TRUE if and only if all the bits are true. The OR of two or more bits is TRUE if at least one of the bits is true. The NOT operation applies only to a single bit, inverting the state of the bit from TRUE to FALSE or from FALSE to TRUE.

| $a$ | $b$ | $a$ AND $b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $a$ | $b$ | $a$ OR $b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $a$ | NOT $a$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Figure 1.4.1** Truth tables for the AND, OR, and NOT operations.

We will illustrate the AND, OR, and NOT operations by deriving a logical statement for the $x < 2$ function from its truth table.

| $x$ (decimal) | $x_1$ | $x_0$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 |

One way for the output $y$ to be true is if $x = (x_1, x_0) = 00$, i.e., if $x_1 = 0$ AND $x_0 = 0$. Another way for $y$ to be true is if $x = 01$, $x_1 = 0$ AND $x_0 = 1$. Either of these statements—one or the other—is sufficient for the truth of $y$. We can combine them into one expression with OR:

$$y = (x_1 = 0 \text{ AND } x_0 = 0) \text{ OR } (x_1 = 0 \text{ AND } x_0 = 1)$$

The AND and OR operations combine two or more Boolean statements to make a more complex statement. The NOT operation, on the other hand, operates on a single statement, changing it from TRUE to FALSE or from FALSE to TRUE. Using the NOT operation, we may replace "$x_1 = 0$" by "NOT( $x_1 = 1$)", replace "$x_0 = 0$" by "NOT( $x_0 = 1$)", and rewrite the expression in terms of positive assertions about the inputs.

$$y = (\text{NOT}(x_1 = 1) \text{ AND } \text{NOT}(x_0 = 1)) \text{ OR } (\text{NOT}(x_1 = 1) \text{ AND } x_0 = 1)$$

Now note: if $x$ is TRUE, then the statement "$x = 1$" is TRUE, and if $x$ is FALSE, the statement "$x = 1$" is FALSE. The additional effort of writing $x_0 = 1$ and $x_1 = 1$ in our expression is redundant. We can instead just write $x_0$ and $x_1$, and the expression for $x < 2$ becomes simpler:

$$y = (\text{NOT}(x_1) \text{ AND } \text{NOT}(x_0)) \text{ OR } (\text{NOT}(x_1) \text{ AND } x_0)$$

There are simple notations for AND, OR, and NOT that enable us to dispense with the words. We denote AND and OR by the symbols $\cdot$ and $+$, respectively, like the ordinary algebraic symbols for multiplication and addition. We denote NOT by an overbar, $\overline{x}$, or a prime, $x'$. With this notation, we can express any complex logical operation as a **logic equation**.

For example, for the $x < 2$ function we began with the wordy

$$y = (x_1 = 0 \text{ AND } x_0 = 0) \text{ OR } (x_1 = 0 \text{ AND } x_0 = 1)$$

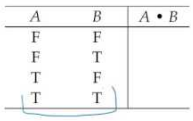
and now have the more concise algebraic expression:

$$y = (\overline{x_1} \cdot \overline{x_0}) + (\overline{x_1} \cdot x_0)$$

Check this for yourself by plugging in 0s and 1s for $x_1$ and $x_0$, and recreating the truth table.

Now that you have been introduced to the AND, OR, and NOT operations, watch the truth table video all the way through.

[redacted]



| A | B | A • B |
|---|---|---|
| F | F | |
| F | T | |
| T | F | |
| T | T | |

[redacted]

The AND and OR operations are defined for any number of inputs, e.g., $a \cdot b \cdot c$, $a + b + c + d$. We call the AND of two or more logical variables, $a \cdot b \cdot c$, a **product term**. The OR of two or more product terms, like $a \cdot b + c \cdot d$, is called a **sum of products**. Writing a logic equation from the rows of a truth table naturally results in a sum of products.

The method is summarized here and in the video.

- Identify all the rows in which the output is TRUE.
- For each of these rows, form a product term from the inputs. For each input, $x$,
    - Write $x$ in the product if $x = 1$ in the row.

- Write $\overline{x}$ in the product if $x = 0$ in the row.
- Omit $x$ from the product if it is marked as "don't care" (–) in the row.

- OR the product terms together.

**Practice.** Repeat the truth table exercises in the previous section, writing logic equations from the truth tables

**Checkpoint 1.4.2** Write a logic equation for a function $y = f(x)$ that asserts a single output bit if the two-bit input is an odd number (zero is neither even nor odd). The truth table is

| $n$ (decimal) | $x_1$ | $x_0$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 |

**Solution**. Either $x = 01$ or $x = 11$ will make $y = 1$. Write $\overline{x_1} \cdot x_0$, $x_1 \cdot x_0$, and combine them with OR,

$$y = (\overline{x_1} \cdot x_0) + (x_1 \cdot x_0)$$

**Checkpoint 1.4.3** Convert this truth table for a one-hot decoder into four logic equations.

| $n$ (decimal) | $n_1$ | $n_0$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 0 | 0 |

**Answer**.

$$y_0 = \overline{n_1} \cdot \overline{n_0}$$
$$y_1 = \overline{n_1} \cdot n_0$$
$$y_2 = n_1 \cdot \overline{n_0}$$
$$y_3 = n_1 \cdot n_0$$

**Checkpoint 1.4.4** The compressed truth table for the multiplexer function was

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| 0 | 0 | — | 0 |
| 0 | 1 | — | 1 |
| 1 | — | 0 | 0 |
| 1 | — | 1 | 1 |

Write a logic equation $y = f(S, A, B)$ for this truth table.

**Answer**. $y = \overline{S} \cdot A + S \cdot B$

## 1.5 Electrical logic and logic diagrams

Truth tables and logic equations are mathematical descriptions. To actually build a digital processor, we must physically represent the 1s and 0s and the logical operations.

In digital electronics, logical TRUE (1) and FALSE (0) are represented by HIGH and LOW voltage levels. Commonly, LOW is about 0.5V and HIGH is about 3V. The usual pairing is HIGH with TRUE and LOW with FALSE, although there are occasions when the pairings are reversed.

The logical operations of AND, OR, and NOT are realized by electronic circuits called **logic gates**. A network of interconnected logic gates is called a **logic circuit**, represented graphically by a **logic diagram**.

Each type of gate has a distinct symbol, shown in the figure below.

AND gates and OR gates may have any number of inputs, but each may have only one output. The NOT gate, or **inverter**, has only one input and one output. The number of inputs a gate has is also called its **fan-in**. A signal connected to the input of a gate or to a system output is called the **driver** for that input or output.

The process of translating a logic equation into a logic diagram or circuit is called **logic synthesis**.

There are conventions for drawing clear logic diagrams. Among them are these:

- Logic diagrams are drawn with system inputs, also called input **ports**, on the left and system outputs (output ports) on the right. We imagine the logic signals to be flowing from the inputs through the gates to the outputs. As much as possible, logic gates are oriented with their outputs pointing to the right.
- Paths (wires) that intersect are marked with a dot at the intersection. Non-intersecting wires merely cross.

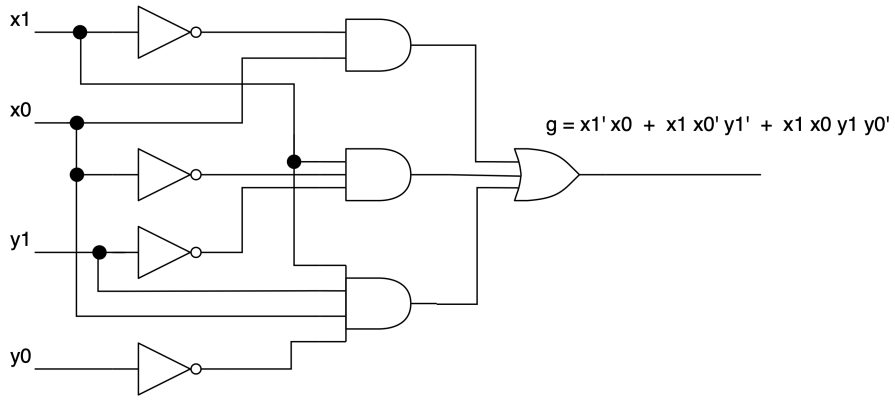
There are also hard-and-fast **design rules** that must obeyed when synthesizing logic circuits. When you have drawn a logic diagram, check it against these design rules. If you discover any errors, go back and examine your design process to see what went wrong, and correct the errors.

- Each input of a gate may have only one driver. Each gate input is connected either to one (and only one) input port or to one (and only one) output of another gate.
- A gate may not drive itself. The output of a gate may not be connected back to an input of the same gate.
- Every gate output may be connected to (**fan out** to) an output port or to one or more inputs of other gates.
- Two gate outputs may not be connected together.

**Example 1.5.1** The logic equation for the two-bit compararison $x > y$ (Checkpoint 1.3.4, p. 8)is

$$g = \overline{x_1} \cdot x_0 + x_1 \cdot \overline{x_0} \cdot \overline{y_1} + x_1 \cdot x_0 \cdot y_1 \cdot \overline{y_0}.$$

The logic diagram is shown below




The circuit has four NOT gates, one each of two, three, and four-input AND gates, and a three-input OR gate. Verify for yourself that the circuit obeys the design rules. □

**Practice.** Make logic diagrams from the logic equations derived in previous checkpoints. Check your circuits against the design rules.

**Checkpoint 1.5.2** Two-bit odd number detector

$$y = (\overline{x_1} \cdot x_0) + (x_1 \cdot x_0)$$

**Checkpoint 1.5.3** One-hot decoder.

$$y_0 = \overline{n_1} \cdot \overline{n_0}$$
$$y_1 = \overline{n_1} \cdot n_0$$
$$y_2 = n_1 \cdot \overline{n_0}$$
$$y_3 = n_1 \cdot n_0$$

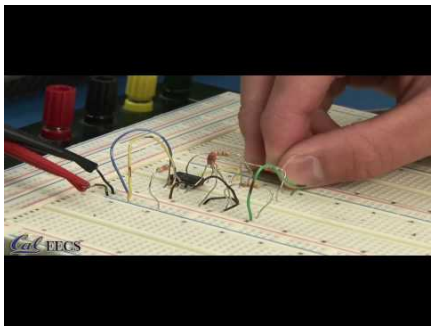
**Checkpoint 1.5.4** Two-input multiplexer

$$y = \overline{S} \cdot A + S \cdot B$$

## 1.6 Implementing a logic circuit

In this section we're going to see how to implement a **prototype**, or experimental version, of a logic circuit with **integrated circuit** logic gates on a **solderless breadboard**. This method is no longer used for industrial logic prototyping, but the technique is still important for other kinds of electronic experimentation and worth mastering.

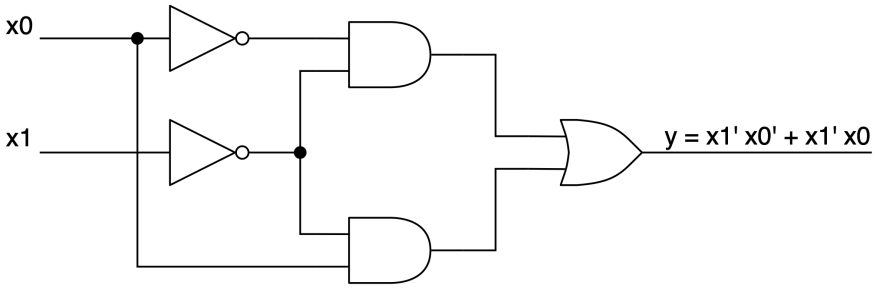
Watch the video from U.C. Berkeley to learn about how solderless breadboards work



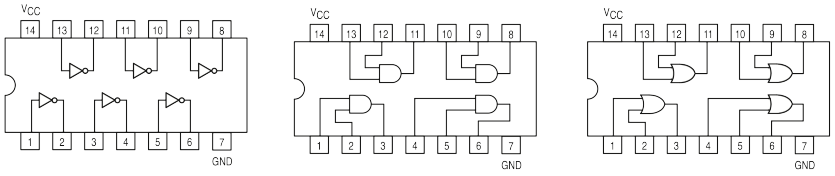
The steps for converting a logic diagram to a prototype are:

1. Plan: Select components and prepare your logic diagram for hardware implementation.
2. Place: Plug the components into the breadboard, leaving room for wiring between them.
3. Route: Connect wires from pin to pin on the components.

We illustrate the procedure using the "$x < 2$" circuit which was designed in an earlier example.




The first *planning* step is to select the necessary integrated circuits. A little browsing of a parts catalog (such as this list on Wikipedia[1]) will turn up three appropriate components:




74LS04 hex NOT gate | 74LS08 quad AND gate | 74LS32 quad OR gate

Here is how to read the part numbers:

- The final digits, "04", "08", "32", uniquely identify the circuits as "hex NOT gate", "quad AND gate", and "quad OR gate", respectively.
- The "LS" denotes a particular type of transistor used to make the logic circuits. There are several other types with different real-world characteristics such as speed and power consumption, but regardless of the technology, a '08 part is a quad AND gate. The LS logic family was once quite common and popular for undemanding applications.

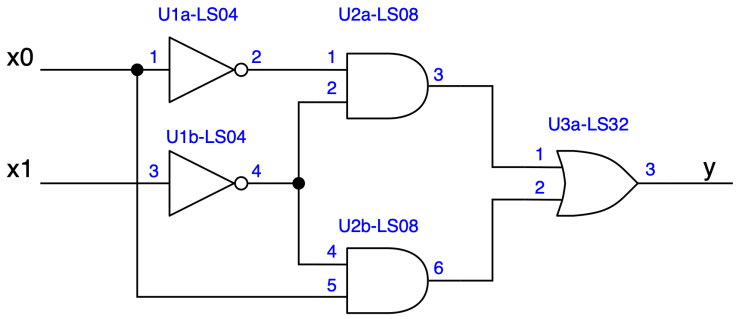
[1] en.wikipedia.org/wiki/List_of_7400-series_integrated_circuits

- Finally, the "74" denotes commercial-grade components, as distinguished from the more rugged military-grade components, whose numbers begin with "54".

As you hold an integrated circuit (IC) in your fingertips so you can see the labeling on top, rotate it so that there is either a notch at the left end or a dot at the lower left corner. This orients the IC so that Pin 1 is at the lower left corner. As shown in the diagrams above, the pin numbers increase from left to right along the bottom edge of the IC and then from right to left along the upper edge. The first and last pins are above each other at the left end of the IC; in all three of these, that's Pin 1 and Pin 14.

The power and ground pins (labeled VCC and GND, respectively, on the diagrams above) are how the logic gates inside the integrated circuit are supplied with electric current so they can function. They are not shown on logic diagrams, but must not be neglected when you wire up a circuit!

The next *planning* step is to annotate the logic diagram, as shown below:



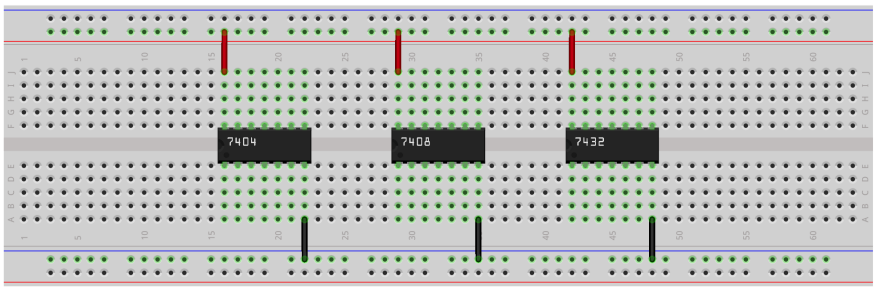

Each logic gate is identified by a unique **reference designator**, or **REFDES**; here, U1a and U1b denote two of the six NOT gates in the LS04, U2a and U2b denote two of the four AND gates in the LS08, and U3a denotes one of the four OR gates in the LS32. The designation of a gate as "a", "b", etc, is arbitrary, but usually the letters are assigned as you go around the chip in a counter-clockwise direction from pin 1 to pin 14 (or whatever the last pin is). The gate inputs and outputs are labeled on the diagram by their respective pin numbers on the integrated circuit.

Each wire (or branching tree of wires) from an input port or gate output to ßoutput ports or other gate inputs is called a **net**. From the annotated logic diagram you may optionally create a table called a **netlist** that shows all the nets in the circuit. For this circuit, the netlist looks like this (you are invited to compare it with the annotated logic diagram):

| Source (from) | Destination (to) |
|---|---|
| *Power and ground nets* | |
| Vcc | U1-14, U2-14, U3-14 |
| GND | U1-7, U2-7, U3-7 |
| *Input and output nets* | |
| Input x0 | U1-1, U2-5 |
| Input x1 | U1-3 |
| U3-3 | Output y |
| *Signal nets* | |
| U1-2 | U2-1 |
| U1-4 | U2-2, U2-4 |
| U2-3 | U3-1 |
| U2-6 | U3-2 |

In the netlist, "U$m$-$n$" denotes pin $n$ of integrated circuit U$m$.

The *placing* step consists of arranging the integrated circuits on the breadboard to make wiring as convenient as possible. Here, since the signals move cleanly from U1 to U2 to U3 in the logic diagram, it makes sense to arrange the LS04, LS08, and LS32 in order from left to right on the breadboard. The layout is diagrammed below. The three ICs are oriented identically, with Pin 1 at the lower left corner. Power and ground wires have also been installed. Power is always wired with red wire and ground is always wired with black wire. Red and black are never used for anything other than power and ground.

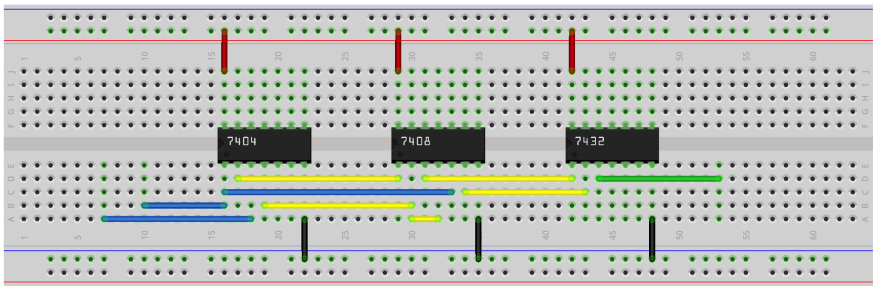



In the *routing* step, the connections tabulated in the netlist are made. We have already seen the power and ground connections. They should be short and low to the board so they are out of the way and cannot accidentally be pulled loose.

Following the power and ground, the input and output connections are made. It is unlikely that these will need to be changed once the circuit is wired, so they should also be kept low to the surface of the board. Consider using colors of wire to identify inputs and outputs (but don't use red or black!).

Finally, route the signal nets between the gate inputs and outputs. These really define the logic function implemented by the circuit, and are the most likely to be changed as you debug the circuit and correct logic errors or wiring errors. It's OK for these wires to have a little extra slack for convenience.

The routed prototype is shown in the diagram below. Compare the breadboard with the netlist to verify the wiring. Can you identify which wire corresponds to each input, and which wire is the output?




An actual breadboard, wired in the lab, is shown below. Compare the photograph with the previous diagram.

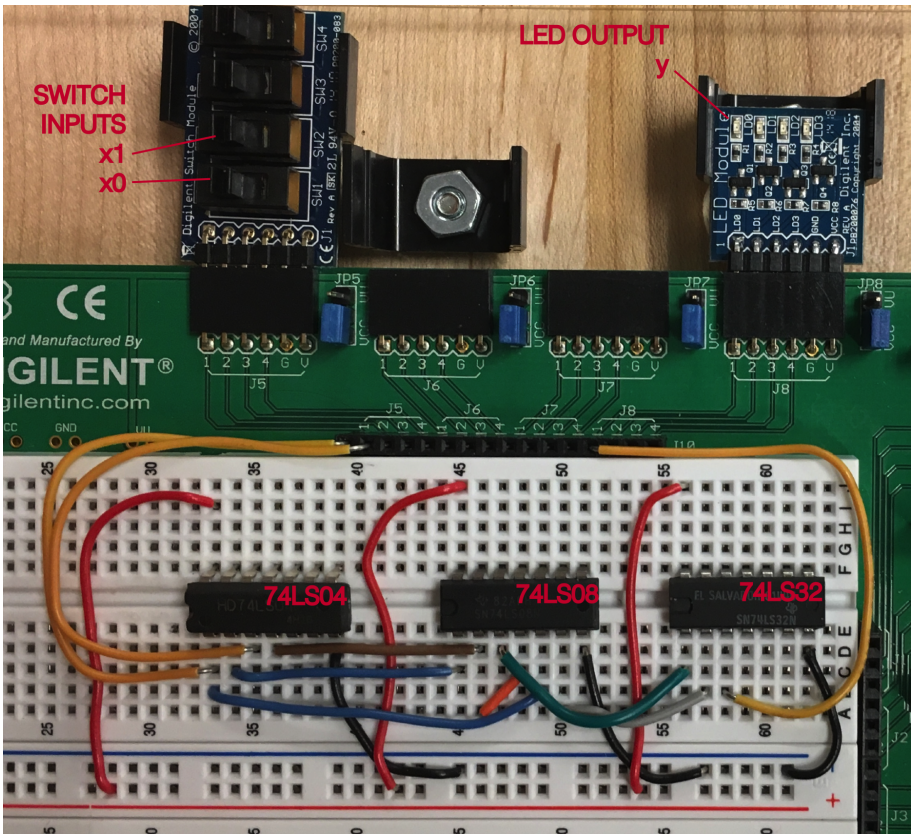



**Figure 1.6.1** The $x < 2$ circuit prototyped on a solderless breadboard.

The final step in developing a prototype is testing: Does the circuit behave as desired? For the circuit in Figure 1.6.1, p. 19, switches connect the inputs $x_0$ and $x_1$ to power (1) or ground (0). The output $y$ is wired to a light-emitting diode (LED). Testing consists simply of applying the inputs $00, 01, 10, 11$ sequentially via the switches and observing whether the LED is appropriately on or off.

There are ways to build a digital system other than prototyping on a solderless breadboard, but they all follow the same basic steps introduced in this chapter:

1. Capture (Section 1.3, p. 8 and Section 1.4, p. 11): Expressing the desired system behavior, e.g., in a truth table and logic equations.

2. Synthesis (Section 1.5, p. 14 and Section 1.6, p. 15): Translating the captured design to a logic diagram and/or a list of required components and a netlist.

3. Implementation (Section 1.6, p. 15): Placing components in a physical layout and routing the connections specified in the netlist.

4. Testing: Verifying that the prototype functions as intended.

**Practice.** For each of the logic diagrams you developed in the last section, select the necessary 7400-series integrated circuits, annotate the logic diagram, and write a netlist.

**Checkpoint 1.6.2** Two-bit odd number detector

$$y = (\overline{x_1} \cdot x_0) + (x_1 \cdot x_0)$$

**Checkpoint 1.6.3** One-hot decoder.

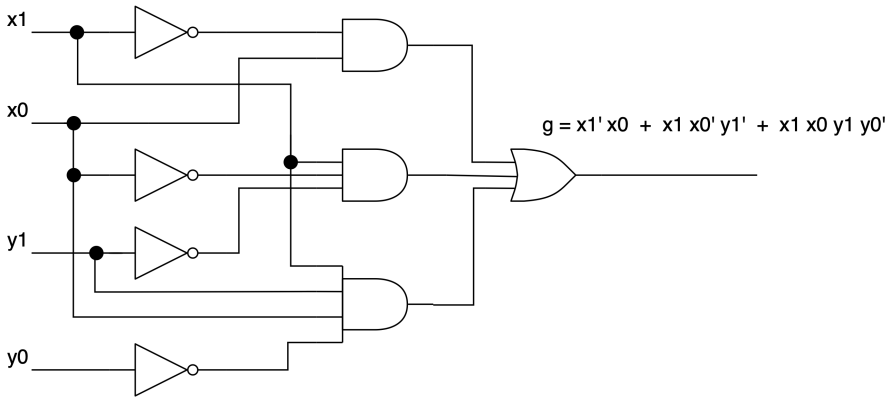
$$y_0 = \overline{n_1} \cdot \overline{n_0}$$
$$y_1 = \overline{n_1} \cdot n_0$$
$$y_2 = n_1 \cdot \overline{n_0}$$
$$y_3 = n_1 \cdot n_0$$

**Checkpoint 1.6.4** Two-input multiplexer

$$y = \overline{S} \cdot A + S \cdot B$$

**Checkpoint 1.6.5** Two-bit comparator

$$g = \overline{x_1} \cdot x_0 + x_1 \cdot \overline{x_0} \cdot \overline{y_1} + x_1 \cdot x_0 \cdot y_1 \cdot \overline{y_0}.$$




Coming up in the next chapter, we will learn about the electronics inside logic gates, and some real-world properties of these gates like their speed, size, and power consumption. We will then learn some techniques for simplifying logic equations so that implemented designs are faster, smaller, and consume less power, all of which are highly desirable in a practical system.

# Chapter 2

# Electronic Logic

In Section 1.5, p. 14 we made the following points about the electrical representation of logic:

- Logical TRUE (1) and FALSE (0) are electrically represented by HIGH and LOW voltage levels; commonly, LOW is about 0.5V and HIGH is about 3V.
- The usual pairing is HIGH with TRUE and LOW with FALSE, although there are occasions when the pairings are reversed.
- The logical AND, OR, and NOT operations are realized by electronic circuits called logic gates, each with a distinct symbol, as illustrated below

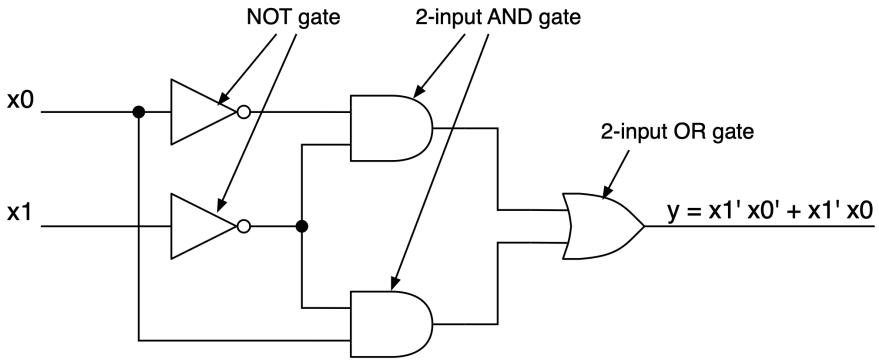


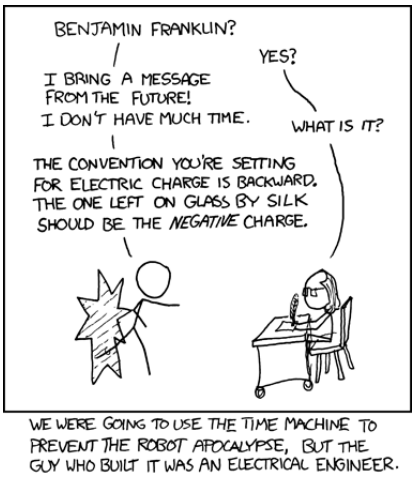

**Figure 2.0.1** AND, OR, and NOT symbols on a logic diagram for the $x < 2$ function.

In this chapter we will expand each of these points, addressing the following questions:

- How are AND, OR, and NOT gates constructed?
- What are the actual characteristics of logic gates?
- Why would anyone deviate from the pairing of HIGH with TRUE and LOW with FALSE?

## 2.1 Electric current and voltage

**Current.** Electric **current** is the flow of electric charge. While many electrically charged particles exist in nature (e.g., electrons, protons, alpha particles, chemical ions), the one of fundamental importance in electronics is the **electron**, which has a charge of approximately $-1.602 \times 10^{-19}$ coulombs. Electric current is measured in **amperes**, or amps (A). Officially, the ampere is a fundamental unit and the coulomb is derived from it, defined to be the amount of charge flowing in a one ampere current in one second. Thus, a 1 amp current is a flow of about $6.24 \times 10^{18}$ electrons per second. Until the discovery of the electron in the twentieth century, it was believed that electric current consisted of positive charges, and engineers continue to follow a **positive current convention**, imagining positive charges flowing in the direction opposite the actual electron flow.




https://xkcd.com/567/

In digital electronics we typically work with currents smaller than an ampere, and use the more convenient units milliampere (mA, $10^{-3}$A) or microampere ($\mu$A, $10^{-6}$A).

**Voltage, resistance, Ohm's law.** An electric current will flow wherever charges can be made to move: in vacuum (e.g., as part of the solar wind, or inside a vacuum tube), in liquids (salt water), gases (a lightning strike in the air, or the current producing the glow in a fluorescent lamp), in metals (wires), and in the special materials called semiconductors, used to fabricate the transistors at the heart of modern electronics. These media have a natural **resistance** to the flow of current, which must be overcome in some way. Resistance is measured in units of ohms , abbreviated $\Omega$. Larger units, commonly used, are the kilohm (K$\Omega$ or K, $10^3$ ohms) and the megohm (M$\Omega$ or M, $10^6$ ohms).

The necessary impetus for current flow is called **voltage**, and is measured in units of volts (V). In electronics, voltage is typically supplied by a battery or a power converter plugged into a wall outlet.

The relationship between current, voltage, and resistance is given by **Ohm's law**, written

$$V = IR,$$

where $V$ is voltage (V), $I$ is current (A), and $R$ is resistance ($\Omega$). Ohm's law is illustrated by the circuit below.

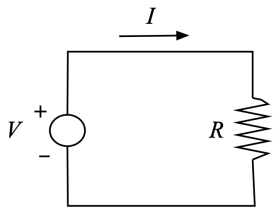



**Figure 2.1.1** Illustrating Ohm's law. A voltage source causes current to flow through a resistance.

The two terminals of the voltage source are labeled positive (+) and negative (-). Electrons flow out of the negative terminal, around the circuit, and back to the positive terminal; however, according to the positive current convention, we write the current as an arrow pointing the opposite direction, from the positive terminal toward the negative. The zigzag line in the circuit is the symbol for a resistance, either naturally occurring (e.g., the resistance of the wire) or deliberately introduced via a discrete component called a **resistor**.

Ohm's law says, for example, that a 5V source is needed to produce a 5mA current in a 1000 Ω (1K) resistor.

**Checkpoint 2.1.2** What current flows in a circuit with a 3.3V source and 100 Ω resistor?

**Answer**. Rewrite Ohm's law to solve for current, then substitute the voltage and resistance values: $I = V/R = 3.3\text{V}/100\Omega = 33\text{mA}$

**Power consumption.** The current that flows in a circuit carries electrical energy, which is converted within the circuit either to useful output (e.g., the light emitted by a light emitting diode (LED)), or to heat, which is not useful unless you're building an oven. Many circuits are battery powered, and we want to minimize the energy consumption of the circuit to operate as long as possible before recharging or replacing the battery. Even if a circuit is powered by a external supply, the energy consumed by the circuit has consequences for your electric bill and for the environment. Again, minimizing energy consumption is important.

The rate of energy consumption is called **power**, and has units of watts (which happens to be volts × amps). We can calculate the power consumption of the circuit by multiplying the supply voltage by the current flowing into the circuit:

$$P = VI$$

Using this expression with Ohm's law, we can also calculate how much electrical energy from the supply is converted to heat by a resistor:

$$P = I^2R = V^2/R$$

where $V$is the **voltage drop** measured across the resistor, and $I$ is the current flowing through the resistor. This is called **Ohmic heating**, and is not a good thing if you are trying to minimize energy consumption.

**Checkpoint 2.1.3** In a circuit with a 3.3V source and 100 Ω resistor, how much power is dissipated by the resistor?

**Answer**. Using $P = V^2/R$ and substituting values, $P = 3.3^2/100 \approx 0.11\text{W} = 110\text{mW}$


https://xkcd.com/643/

## 2.2 Electric circuits

A complex electrical circuit may have several branches that the current can follow in its journey from positive to negative (Figure 2.2.1, p. 24)

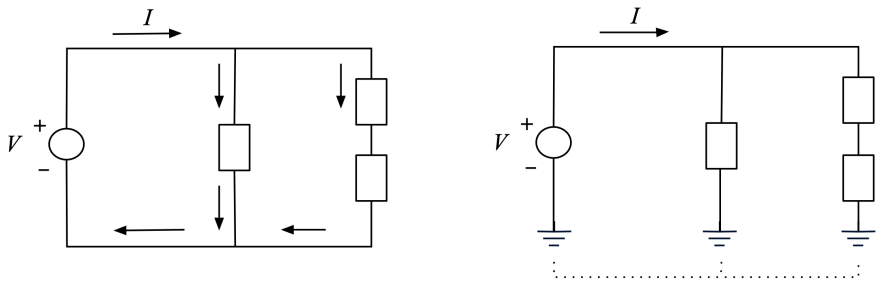



**Figure 2.2.1** A circuit with two current branches (left). The common return path to the negative terminal of the supply is abbreviated by the use of the ground symbol (right). There is an implicit wire (dotted) connecting all the ground symbols together.

It is frequently convenient to abbreviate the common return path to the negative power supply terminal by drawing the "common" or "ground" symbol at the end of each branch and on the negative terminal, as shown in the figure. It is understood that there is a wire connecting all the ground terminals together. The ground symbolizes an actual earth connection in some electrical systems, but in digital electronics it is simply used to represent a common connection to the negative supply terminal.

Voltage and current are measured in a circuit with a voltmeter and ammeter, respectively. The resistance of a component is measured with an ohmmeter, outside the circuit. (Do not try to measure resistance of a component in a circuit with an ohmmeter, especially if the circuit has power applied!) In the lab, these measurement functions are bundled into one instrument called a **multimeter**. Figure 2.2.2, p. 25, below, illustrates how voltage, current, and resistance are directly measured with a multimeter.

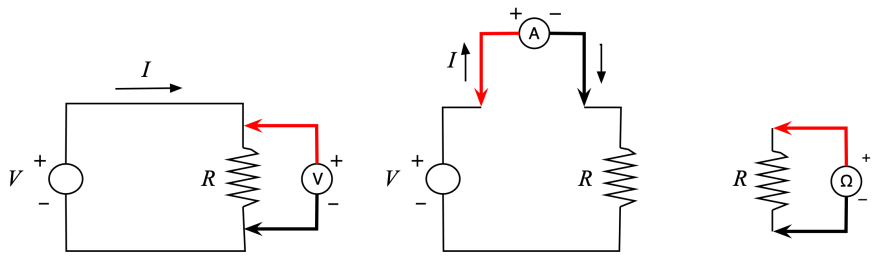



**Figure 2.2.2** Electric measurements with a multimeter. Voltage drop across a component is measured with a voltmeter, with the positive (red) lead toward the positive supply (left). Current is measured by opening the circuit so the current flows through the ammeter, again with the positive (red) lead toward the positive supply (center). Resistance is directly measured by removing the component from the circuit and using the ohmmeter. The polarity of the leads does not matter when measuring resistance.

In addition to direct measurements, any of these three quantities can be indirectly measured using the other two, via Ohm's law. Rather than breaking a circuit to insert an ammeter, for example, the voltage drop across a known resistor can be measured, and the current calculated from $I = V/R$.

Here is a video, from U.C. Berkeley, with more information about the multimeter.

**Voltage divider.** Here is a simple circuit with two resistors, connected in **series**, called a **voltage divider**.

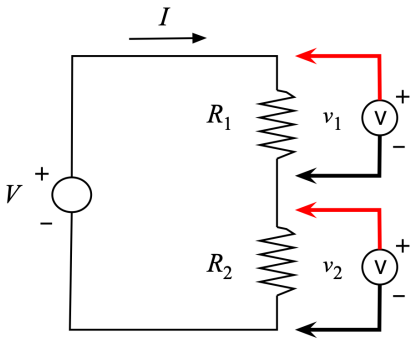



**Figure 2.2.3** A circuit with resistors in series. A voltmeter measures the voltage drop across each resistor. The positive (red) lead of the meter is placed toward the positive terminal of the power supply, and the negative (black) lead toward the negative terminal of the supply.

Before making measurements in the lab, let's do a thought experiment using Ohm's law. The current through resistor $R_1$ is $v_1/R_1$, and the current through $R_2$ is $v_2/R_2$. The same current passes through both resistors; the current flowing out of $R_1$ is equal to the the current flowing into $R_2$, a consequence of the principle known as **Kirchoff's current law**, so we have

$$I = \frac{v_1}{R_1} = \frac{v_2}{R_2}$$

It is also true that $V = v_1 + v_2$, a consequence of the principle known as **Kirchoff's voltage law**. We can combine these two expressions and obtain expressions for the voltage drops $v_1$ and $v_2$:

$$v_1 = \frac{R_1}{R_1 + R_2} V$$
$$v_2 = \frac{R_2}{R_1 + R_2} V$$

The circuit is called a **voltage divider**, because the supply voltage $V$ is divided between the two resistors.

**Checkpoint 2.2.4** Fill in the algebraic steps leading to the voltage divider equations.

**Checkpoint 2.2.5** A voltage divider has two resistors, with $R_1 > R_2$. Which of the following statements is true?

1. $v_1 > v_2$
2. $v_1 = v_2$
3. $v_1 < v_2$

**Answer**. The larger resistor has the larger voltage drop. Hence, (1) is true.

**Checkpoint 2.2.6** You have a 5V supply and want to use a voltage divider to reduce the voltage to $v_2 = 1$V. What should the resistor ratio $R_1/R_2$ be?

**Solution**. Use the equation $v_2 = \frac{R_2}{R_1+R_2} V$. Then,

$$\frac{R_1 + R_2}{R_2} = \frac{R_1}{R_2} + 1 = \frac{V}{v_2}$$
$$\frac{R_1}{R_2} = \frac{V}{v_2} - 1 = \frac{5}{1} - 1 = 4:1$$

**Checkpoint 2.2.7** According to the voltage divider equations, what is the approximate value of $v_2$ in the cases where $R_1 \gg R_2$ (including $R_2 = 0$), on the one hand, and $R_1 \ll R_2$ (including $R_1 = 0$), on the other hand?

**Answer**. When $R_1 \gg R_2$,

$$v_2 = \frac{R_2}{R_1 + R_2} V \approx \frac{R_2}{R_1} V \approx 0$$

and when $R_1 \ll R_2$,

$$v_2 = \frac{R_2}{R_1 + R_2} V \approx \frac{R_2}{R_2} V = V$$

**Experiments.** Here are two experiments you can do to practice making voltage measurements. You will need a breadboard, power supply (a battery

will do), multimeter, and an assortment of resistors. You can use the multimeter to measure the resistor values, or learn to read the resistor color code[1].

1. Wire up the voltage divider circuit, Figure 2.2.3, p. 25, with any two resistors. Measure the resistor voltages with the voltmeter leads attached in the normal way, according to the figure. Then reverse the voltmeter leads (red and black) and repeat the measurements. What do you observe?

| | $v_1$ | $v_2$ |
|---|---|---|
| Leads normal | | |
| Leads reversed | | |

2. Wire up the voltage divider circuit and fill in this table for several resistor pairs. (Be sure to attach the voltmeter leads in the normal way, according to Figure 2.2.3, p. 25). Using the voltage divider equations, predict the voltage drops, then confirm them with measurements.

| Resistors | | Predicted | | Measured | |
|---|---|---|---|---|---|
| $R_1$ | $R_2$ | $v_1$ | $v_2$ | $v_1$ | $v_2$ |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

**Light emitting diodes.** When a circuit has something to say to you, it often turns on a **light emitting diode (LED)**. An LED is an electronic device that emits light in response to the current flowing through it. A simple LED circuit is shown below. The resistor controls the amount of current flowing through the LED, and hence the brightness of the light.

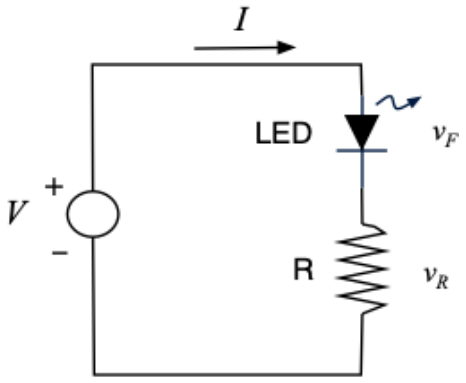



**Figure 2.2.8** A light emitting diode, connected in series with a current-limiting resistor.

The terminals of the LED are called the **anode** (the triangle, in the symbol) and the **cathode** (the horizontal bar, in the symbol). The LED has three salient properties:

- Current only flows in one direction, from the anode to the cathode. The anode is connected toward the positive power supply, and the cathode toward the negative, or ground.

[1] www.digikey.com/en/resources/conversion-calculators/conversion-calculator-resistor-color-code-4-band

- Current will flow when the anode voltage (relative to ground) exceeds the cathode voltage (relative to ground) by an amount called the **forward voltage**. Once current is flowing, the voltage across the LED remains relatively constant over a wide range of currents (the LED does not obey Ohm's law).
- The LED has a maximum allowed current. If this current is exceeded, the LED will overheat and burn out.

You can read more about LEDs in this tutorial[2] from SparkFun Electronics.

The value of an LED's forward voltage is given in its datasheet. Otherwise, you can measure it with a voltmeter in a test circuit. Once you know it, you can use Ohm's law to calculate the resistor value needed to obtain a particular current. This is important, because operating an LED without a current-limiting resistor will likely result in burning it out. Here is how we derive an expression for $R$:

- Using Kirchoff's voltage law, $V = v_F + v_R$.
- Using Ohm's law, the voltage drop across the resistor is $v_R = IR$.
- Combine these two equations to solve for $R$ in terms of the other parameters:

$$\begin{aligned} & V = v_F + IR \\ \implies \quad & R = \frac{V - v_F}{I} \end{aligned}$$

**Example 2.2.9** In Figure 2.2.8, p. 27, suppose the LED's forward voltage $v_F$ can range from a minimum 3.0V to maximum 3.4V. The maximum allowed steady current is 20mA, and the suggested operating current is 16-18 mA (datasheet[3]). The power supply voltage is $V = 5$V.

Calculate the resistor value required to limit the LED current to not exceed the 20mA maximum, for any forward voltage between 3.0V and 3.4V. This is called a **worst-case design**.

**Solution**. Use the equation for $R$, with $V = 5$V, $v_F = 3.0$V, and $I = 20\text{mA} = 0.02$A:

$$R = \frac{V - v_F}{I} = \frac{5 - 3}{0.02} = 100\Omega$$

Repeat with $v_F = 3.4$V:

$$R = \frac{V - v_F}{I} = \frac{5 - 3.4}{0.02} = 80\Omega$$

Choose the larger of the two values to be safe. □

**Checkpoint 2.2.10** Visit the SparkFun or Adafruit websites and look at their single LEDs, e.g.,

- SparkFun, 5mm green LED[4], also available red, blue, yellow
- Adafruit, 5mm superbright red LED[5], also available green, blue, yellow

and note any differences in forward voltage with color.

[2] learn.sparkfun.com/tutorials/light-emitting-diodes-leds

[3] cdn.sparkfun.com/datasheets/Components/LED/10mm%20green%20LED%20YSL-R1047G5D-D2-1.png

[4] www.sparkfun.com/products/9592

[5] www.adafruit.com/product/297

**Experiment.** Here is an experiment you can do to become familiar with LEDs. You will need a breadboard, 5V power supply, LED, and three resistors spread between 100 and 800 ohms. Wire up the circuit in Figure 2.2.8, p. 27. For each of your resistor values, measure $v_F$ and $v_R$, calculate the current, and note the brightness in the table below:

| $R$ | $v_F$ | $v_R$ | $I$ | comment on brightness |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |

**Safety note.** While the voltages and currents in digital circuits are usually low, there are situations that can result in damage to components and, more importantly, injury to you. It is a common joke that electronic devices run on magic smoke. If you ever let the magic smoke out of a device, it will stop working. What is not funny is that the release of smoke is usually preceded by serious overheating that can burn skin and, occasionally, an explosive release of flame and molten material.

Before you ever apply power to a circuit, check that you have correctly wired the power and ground pins of your integrated circuits. Wiring them backwards will almost always result in overheating and damage. Always make sure that your power supply is set to the correct voltage (usually, either 3.3V or 5V) with proper polarity (positive to the red wire / power pins, negative to the black wire / ground pins). Never run an LED without a current limiting resistor. If you notice a "hot" smell, turn the power off immediately and be very careful about touching anything in your circuit until it has had a chance to cool. Then check your wiring again.

Here is a brief video with some spectacular examples of overheated electronic components.

We now have enough of the basic principles of electricity to start seeing how logic gates work on the inside.

## 2.3 Electronic switches (transistors)

You have observed that the output of the voltage divider circuit can be controlled by changing the resistances. With reference to Figure 2.3.1, p. 30, below, the voltage divider equation derived earlier says

$$V_{\text{out}} = V \times \frac{R_2}{R_1 + R_2}.$$

Now, two extreme cases are of primary interest for logic design.

- $R_2 \gg R_1$. In particular, if $R_1 \approx 0$ and $R_2 \to \infty$, the output "sees" a low resistance path to the supply voltage, and the output voltage is $V$, or HIGH.
- $R_2 \ll R_1$. In particular, if $R_1 \to \infty$ and $R_2 \approx 0$, the output sees a low resistance path to ground, and the output voltage is 0, or LOW.

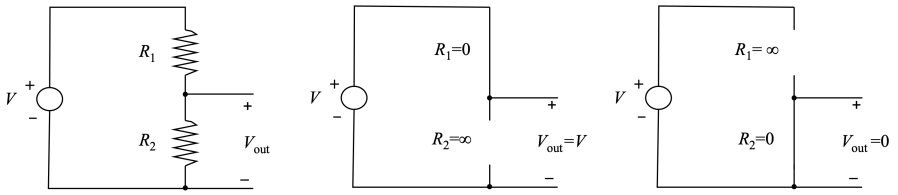



**Figure 2.3.1** Voltage divider. The output, $V_{\text{out}}$, is the voltage across $R_2$. In extreme cases, the output is HIGH ($V$) or LOW (0).

These extreme cases of resistance are identical to the function of a switch. A closed switch has very low resistance to current flow, and an open switch has infinite resistance (Figure 2.3.2, p. 30). (Unlike a door, a closed switch allows current to flow.)

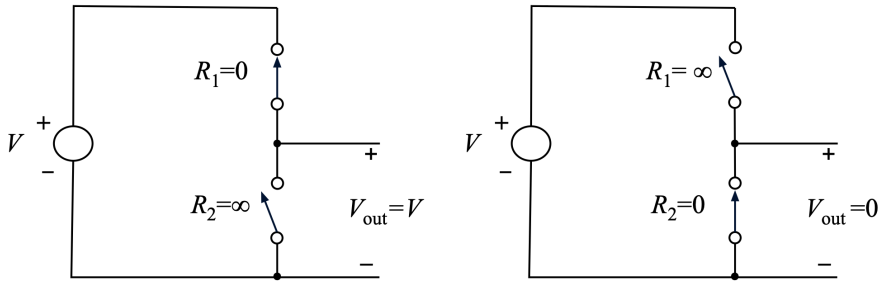



**Figure 2.3.2** Voltage divider, with extreme values of resistance replaced by switches.

The key step is to make the resistances/switches controllable electrically by HIGH and LOW input voltages. These electrically-controlled resistors/ switches are called **transistors**. The particular transistors we will use are fabricated using so-called MOS ("metal-oxide-semiconductor") technology. For our purposes we can skip over the physics of how they work and simply regard them as voltage controlled resistors or, even simpler, as voltage controlled switches.

There are two types of MOS transistor, called **pMOS** and **nMOS**. Both have three terminals, called the **gate**, **source**, and **drain**. They are distinguished in a circuit diagram by slightly different symbols, shown in Figure 2.3.3, p. 31. The lower pair of symbols has the advantage of identifying the source by an arrow, and distinguishing pMOS from nMOS by the direction of the arrow. However, the upper pair of symbols is more common in digital design and will be used here.

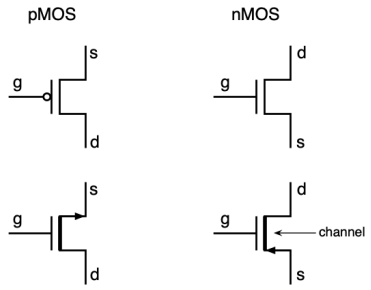



**Figure 2.3.3** Common symbols for pMOS transistor (left) and nMOS transistor (right). The transistor's silicon channel carries current from source to drain (pMOS) and from drain to source (nMOS). Voltage applied to the gate controls the resistance of the channel.

The operation of the two kinds of transistors can be observed by inserting them in place of one of the resistors in a voltage divider circuit, Figure 2.3.4, p. 32. The curves in the figure are produced by simulating the circuits with the LTSpice software, a free download from Analog Devices [8]. These simulated voltage curves correctly illustrate general behavior. The physical parameters of a transistor's design can be varied to affect the detailed shape of the curve for particular applications.

- When the gate voltage of the pMOS transistor is at or above the source voltage, the resistance of the channel is very high, approximately infinite. When the gate is lower than the source, the channel resistance decreases as the gate voltage decreases, to a very small value when the gate voltage is near zero. Consequently, the output voltage of the circuit *rises* as the gate voltage *decreases*. We say that the pMOS transistor "pulls the output up".

- When the gate voltage of the nMOS transistor is at or below the source voltage, the resistance of the channel is very high, approximately infinite. When the gate is higher than the source, the resistance decreases as the gate voltage increases, to a very small value when the gate voltage is near the supply voltage. Consequently, the output voltage of the circuit *decreases* as the gate voltage *increases*. We say that the nMOS transistor "pulls the output down".

The gate voltage, by controlling the resistance of the transistor channel, switches the output voltage between a HIGH and LOW level, with a narrow transition region.

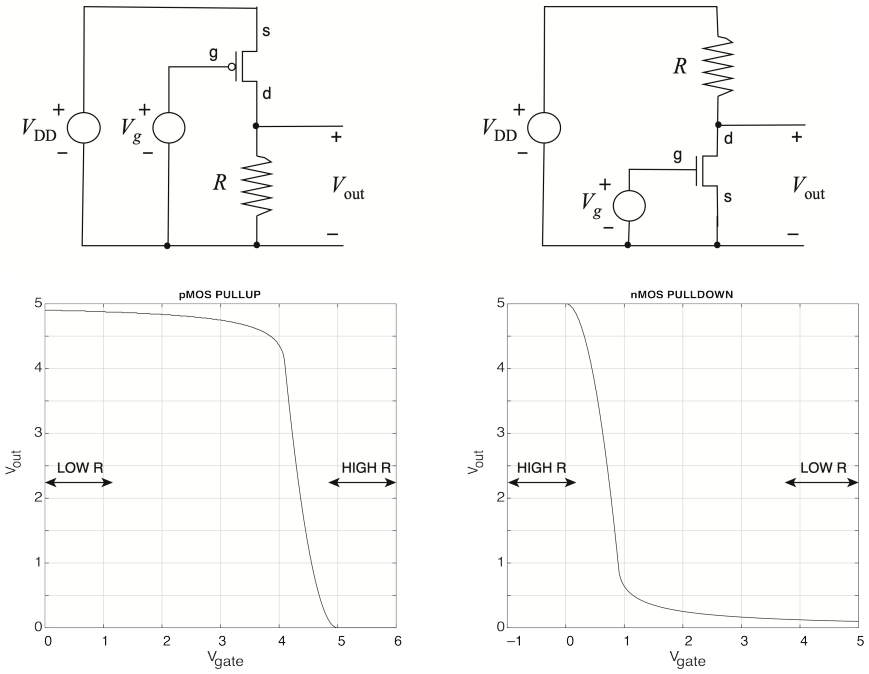



**Figure 2.3.4** pMOS pull-up and nMOS pull-down circuits simulated with LTSpice. In both circuits, $R = 500\text{K}\Omega$. The gate voltage controls the resistance of the transistor channel, and switches the output voltage between a HIGH and LOW level, with a narrow transition region. Compare Figure 2.3.1, p. 30.

The pMOS and nMOS transistors are always used as shown in Figure 2.3.4, p. 32, the pMOS transistor with its source connected to the supply voltage $V_{DD}$, and the nMOS transistor with its source connected toward ground. The physical reasons for this are beyond the scope of this course, but are explained in many electronics texts.

Another way to visualize the operation of the two transistors is to compute and plot the resistance of the channel as a function of the gate voltage. This is shown in Figure 2.3.5, p. 33.

- The resistance of the pMOS transistor is low when the gate voltage is LOW, and very high when the gate voltage is HIGH.
- Conversely, the resistance of the nMOS transistor is very high when the gate voltage is LOW, and low when the gate voltage is HIGH.

When the transistor has an approximately infinite resistance, it is like an open (non-conducting) switch, and in comparison, when it has a low resistance, it is like a closed (conducting) switch. Thus, we can see how the transistors fulfill the roles of approximately ideal zero and infinite resistances illustrated earlier, in Figure 2.3.1, p. 30.

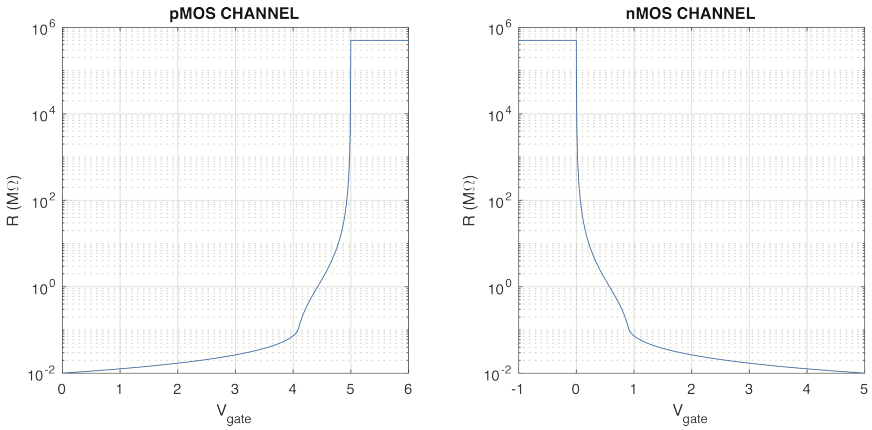



**Figure 2.3.5** Resistance of the pMOS and nMOS channels, calculated with LTSpice (see Figure 2.3.4, p. 32).

**Checkpoint 2.3.6** Review Figure 2.3.2, p. 30, Figure 2.3.4, p. 32, and Figure 2.3.5, p. 33, summarizing for yourself how the nMOS and pMOS transistor circuits work.

## 2.4 Building gates from transistors

**CMOS inverter.** Look again at Figure 2.3.4, p. 32. In both the pMOS and nMOS circuits, when the input voltage, $V_{gate}$, is LOW, the output voltage is HIGH, and when the input is HIGH, the output is LOW. Either circuit could be used to implement a NOT gate. Indeed, for many years logic circuits were fabricated solely from nMOS transistors. Today, the dominant technology is complementary MOS, or **CMOS**, which uses both nMOS and pMOS transistors. A CMOS NOT gate is shown in Figure 2.4.1, p. 33, together with its voltage curve, obtained by LTspice simulation.

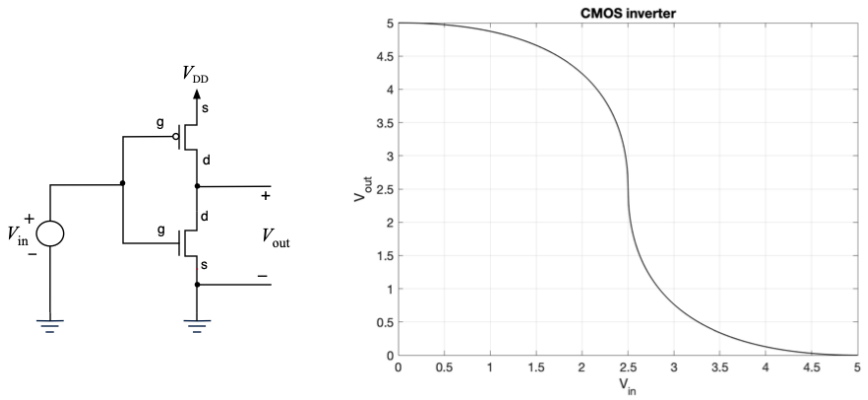



**Figure 2.4.1** The CMOS inverter (left) and its simulated voltage characteristic curve (right).

The CMOS inverter consists of a pMOS transistor stacked on top of an nMOS transistor. When the input voltage is LOW, the pMOS transistor is ON and the nMOS transistor is OFF. The output is connected via the low-resistance pMOS transistor to $V_{DD}$, creating a HIGH output. When the input voltage is HIGH, the pMOS transistor is OFF and the nMOS transistor is ON, and the

output is connected via the low-resistance nMOS transistor to ground, creating a LOW output.

As the input increases from LOW to HIGH, there is a point in the middle where the output rapidly switches from HIGH to LOW. In practice, input voltages around this transition are avoided. Figure 2.4.2, p. 34, below, shows why. The current, $I_{DD}$, that flows from power to ground through the two transistors peaks midway between the LOW and HIGH extremes of the input voltage $V_{in}$. When the input is LOW, the nMOS transistor is off and little current flows. When the input is HIGH, the pMOS transistor is off and, again, little current flows. But in the middle between LOW and HIGH, both transistors are partially on and a considerable amount of current is able to flow. The bottom graph in the figure plots the power consumption, $P = V_{DD}I_{DD}$, and shows that power consumption is also maximized between the extreme LOW and HIGH input voltages. To minimize power consumption, then, the circuit is operated so that the transistors function as switches, snapping rapidly between ON and OFF, spending negligible time in the mid-range, high power zone.

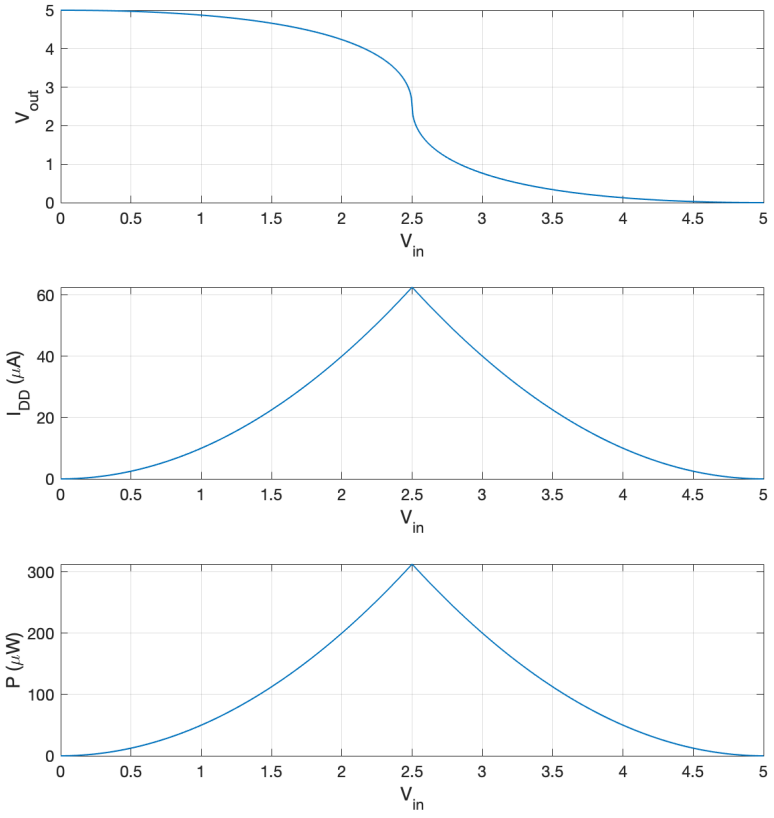



**Figure 2.4.2** (Top) Output voltage of the simulated CMOS inverter. (Middle) The current flowing through the two transistors, from power to ground, is lowest at the extremes of $V_{in}$. (Bottom) The power dissipated in the transistors is lowest at the extremes of $V_{in}$.

**CMOS AND and OR gates.** We can build out from the CMOS inverter circuit to construct AND and OR gates, but the path will turn out to be rather indirect. For an AND gate, we want two HIGH inputs to result in a HIGH

output. However, pMOS transistors, which connect the output to power, are only ON when their gate inputs are LOW. Likewise, for an OR gate, we want two LOW inputs to result in a LOW output, but nMOS transistors, which connect the output to ground, are only ON when their gate inputs are HIGH. The transistors appear to be backwards from what we need them to be.

On the other hand, we can easily construct inverting AND and OR gates, known as NAND and NOR.

**Checkpoint 2.4.3** Two other interesting logical operations are NAND (NOT AND) and NOR (NOT OR). Write truth tables for these two operations.

**Answer**.

| $a$ | $b$ | $a$ NAND $b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $a$ | $b$ | $a$ NOR $b$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Opposite the definition of the AND gate, the NAND gate has a LOW output when both of its inputs are HIGH, and a HIGH output when either of its inputs is LOW. This fits with the way that pMOS and nMOS transistors work. Likewise, opposite the definition of the OR gate, the NOR gate has a LOW output when either of its inputs is high, and a HIGH output when both inputs are LOW. Again, this matches the behavior of the pMOS and nMOS transistors. The NAND and NOR circuits are shown, with their logic symbols, in Figure 2.4.4, p. 35, below.

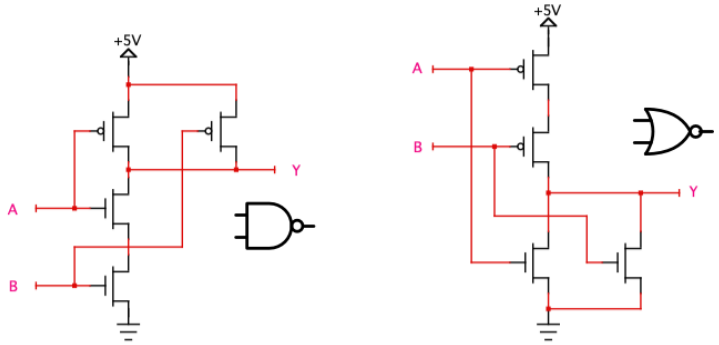



**Figure 2.4.4** The CMOS NAND (left) and NOR (right) gates, symbols and circuits.

In the NAND circuit, the two nMOS transistors are in series. When the inputs are both HIGH, both of the nMOS transistors are ON, providing a low-resistance path from the output to ground, as desired. The two pMOS transistors are off. When either input is LOW, one of the pMOS transistors will be ON, providing a low-resistance path from the output to power, and one of the nMOS transistors will be OFF, blocking the path from output to ground. Consequently, the output is HIGH. Apply the same reasoning to the NOR circuit, and convince yourself that it works as it is supposed to.

**Checkpoint 2.4.5** To review, explain, transistor-by-transistor, how the NAND and NOR circuits work.

A NAND gate with more than two inputs is constructed by adding one pMOS-nMOS pair for each additional input. The pMOS transistor is added in parallel with the existing pMOS transistors between the output and power, and the nMOS transistor is added in series with the existing nMOS transistors

between the output and ground. Similarly, the NOR gate may be expanded by adding one pMOS-nMOS pair for each additional input.

**Checkpoint 2.4.6** Draw a three-input NAND circuit and a three-input NOR circuit.

**Answer**.

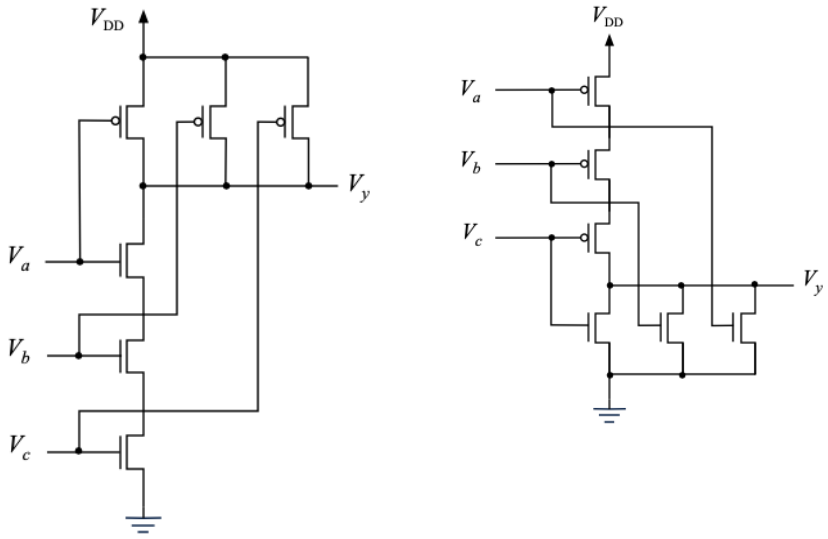



**Figure 2.4.7** Three-input CMOS NAND gate (left). Three-input CMOS NOR gate (right). When all three inputs are HIGH, the three nMOS transistors in the NAND are all ON, pulling the output LOW. When all three inputs are LOW, the three pMOS transistors in the NOR are all ON, pulling the output HIGH.

A true AND or OR gate is constructed by driving an inverter circuit with the output of a NAND or NOR gate, respectively. That is, an AND gate is a "NOT NAND" and an OR gate is a "NOT NOR".

**Checkpoint 2.4.8** Draw the circuit for a two-input OR gate.

**Answer**.

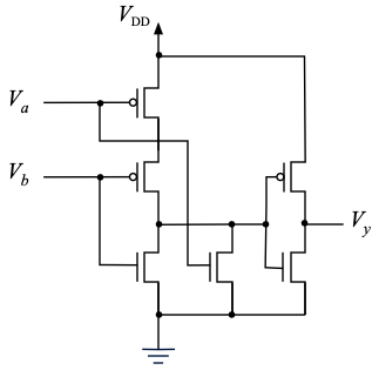



**Figure 2.4.9** Two-input CMOS OR gate is a NOR followed by a NOT.

**Checkpoint 2.4.10** Count the transistors: Figure out a general expression for the number of transistors in an $n$-input NAND or NOR gate. How does this compare with the transistor count for an AND or OR gate with the same number of inputs?

**Answer**. NAND or NOR: Two transistors per input, or $2n$ transistors in all.

AND or OR: AND and OR are NAND and NOR, followed by inverters.

Add two transistors for the inverter at the output, or $2(n+1)$ transistors in all.

**Logic families.** Since the introduction of digital integrated circuits in the 1960s, logic circuit designs have evolved, becoming smaller, faster, and more power-efficient. Today you are likely only to encounter logic families based on CMOS technology. An older technology, known as **bipolar**, or **transistor-transistor logic (TTL)**, is still occasionally seen. We use a bipolar family denoted LSTTL in the early labs for this course, because bipolar devices are generally more durable for breadboarding by beginning students. Other important practical distinctions between CMOS and TTL will be made in the following sections. A broad spectrum of discrete integrated circuit logic families is described in the Texas Instruments Logic Guide [14].

## 2.5 Electrical characteristics of logic circuits

**Voltage.** Under normal operation, the inputs and outputs of a logic gate circuit are either LOW -- close to ground -- or HIGH -- close to the supply voltage. The valid ranges for LOW and HIGH inputs and outputs are specified by four parameters:

- A valid LOW input is required be less than a specified value denoted $V_{IL}$ (voltage-in-low).
- valid HIGH input is required to be greater than a specified value, denoted $V_{IH}$ (voltage-in-high).
- A valid LOW output is required to be less than a specified value denoted $V_{OL}$ (voltage-out-low).
- A valid HIGH output is required to be greater than a specified value denoted $V_{OH}$ (voltage-out-high).

The actual values of these parameters in a particular circuit depend on the physical design of the transistors, but are consistent across integrated circuits in the same family, e.g., the LS devices used in Section 1.6, p. 15. The relevant portions of a 74LS00 (quad two-input NAND) datasheet [11], showing the input and output voltage limits, are shown in Figure 2.5.1, p. 37, below. LSTTL devices operate with a nominal supply voltage ($V_{CC}$) of 5V. Certain CMOS logic devices may also operate from a 5V supply with compatible input and output voltage levels. Many other CMOS devices operate at lower supply voltages—3.3V or less—and are not TTL-compatible without special precautions being taken.

| | | | MIN | NOM | MAX | UNIT |
|---|---|---|---|---|---|---|
| $V_{CC}$ | Supply voltage | SN54xx00 | 4.5 | 5 | 5.5 | V |
| | | SN74xx00 | 4.75 | 5 | 5.25 | |
| $V_{IH}$ | High-level input voltage | | 2 | | | V |
| $V_{IL}$ | Low-level input voltage | SNx400, SN7LS400, and SNx4S00 | | | 0.8 | V |
| | | SN54LS00 | | | 0.7 | |

| | | | | | |
|---|---|---|---|---|---|
| $V_{OH}$ | $V_{CC}$ = MIN, $V_{IL}$ = 0.8 V, and $I_{OH}$ = –1 mA | 2.5 | 3.4 | | V |
| $V_{OL}$ | $V_{CC}$ = MIN, $V_{IH}$ = 2 V, and $I_{OL}$ = 20 mA | | | 0.5 | V |

**Figure 2.5.1** Excerpts from the data sheet for a 74LS00 quad two-input NAND gate integrated circuit (Texas Instruments), showing the input and output voltage limits.

From the table in the figure, we read:

- $V_{IL} = 0.8$V

- $V_{IH} = 2.0\text{V}$
- $V_{OH}$ is typically 3.4V, but could be as low as 2.5V.
- $V_{OL}$ is no larger than 0.5V.

The minimum and maximum voltages are also known as the **worst case** values.

Usually, the output of one gate will be connected to drive the input of another gate. If the output is a valid LOW (less than 0.5V), it will certainly be low enough to satisfy the input requirement of the following gate (less than 0.8V). Likewise, if the output is HIGH (at least 2.5V), it will be certainly be high enough to satisfy the input requirement of the following gate (higher than 2.0V). It is always the case that $V_{OL} < V_{IL}$ and $V_{OH} > V_{OL}$ (Figure 2.5.2, p. 38). The gaps between the output and input voltage specifications are called the **noise margin** of the circuit. With reference to the figure, even if some noise causes a LOW output to be slightly larger than 0.4V, it will still be recognized as a valid LOW by the following gate input. Similarly, a HIGH output corrupted by noise can still be high enough to be recognized as a valid HIGH by the following gate input.

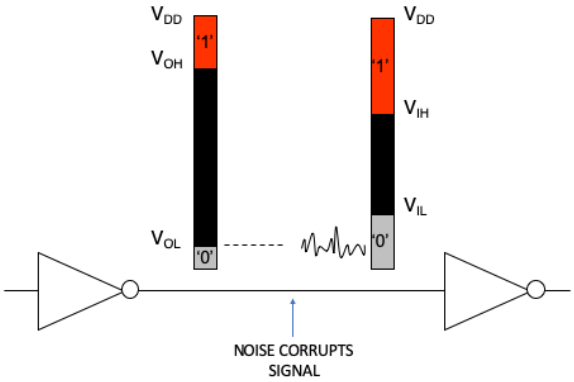



**Figure 2.5.2** Noise margin. The LOW output of the first gate is corrupted by noise, but is still received as a valid LOW input by the second gate.

**Current.** In a typical circuit, the output of a logic gate drives another circuit or component, which is called a **load**. (Figure 2.5.3, p. 39). All loads require current, and that current must be supplied by the driver. As shown in the figure, current may flow either direction between the driver and the load.

- When the driver is HIGH, current flows from the power supply, through the gate output, into the load and then to ground. The driver is said to **source** the load current.
- When the driver is LOW, current flows from the power supply, through the load into the gate output and then to ground. The driver is said to **sink** the load current.

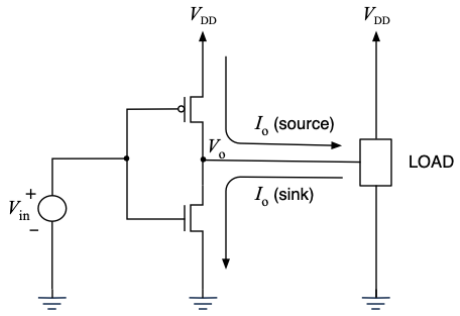



**Figure 2.5.3** Logic gate output current. When the output is HIGH, current flows from the power supply, through the output, into the load. When the output is LOW, current flows from the load into the output, to ground.

The amount of current a driver may source or sink is limited. Current flowing out of the gate passes through the channel of a pMOS transistor, which has a finite resistance even then the transistor is ON. By Ohm's law, $V = IR$, there is a voltage drop across the transistor that increases as more current is drawn by the load. This voltage drop lowers $V_O$, and since $V_O$ is guaranteed to be higher than $V_{OH}$, $I_O$ must not exceed a certain maximum value. This maximum output current is denoted $I_{OH}$ (current-out-high). Similarly, current flowing into the gate passes through the channel of the nMOS transistor, which also has a finite resistance when the transistor is ON. Increasing current increases the voltage drop across the nMOS channel, and since $V_O$ is guaranteed to be lower than $V_{OL}$, there is also a limit to how much current may be sunk by the gate output. This value is denoted $I_{OL}$ (current-out-low).

The datasheet for a typical HC-series device [13] is excerpted in Figure 2.5.4, p. 39. According to the table in the figure, when the power supply is 4.5V, as much as 4 mA can be sourced or sinked, while the output voltage remains within acceptable HIGH and LOW limits.

| PARAMETER | TEST CONDITIONS | | $V_{CC}$ | $T_A$ = 25°C | | | SN54HC04 | | SN74HC04 | | UNIT |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | MIN | TYP | MAX | MIN | MAX | MIN | MAX | |
| $V_{OH}$ | $V_I$ = $V_{IH}$ or $V_{IL}$ | $I_{OH}$ = –20 µA | 2 V | 1.9 | 1.998 | | 1.9 | | 1.9 | | V |
| | | | 4.5 V | 4.4 | 4.499 | | 4.4 | | 4.4 | | |
| | | | 6 V | 5.9 | 5.999 | | 5.9 | | 5.9 | | |
| | | $I_{OH}$ = –4 mA | 4.5 V | 3.98 | 4.3 | | 3.7 | | 3.84 | | |
| | | $I_{OH}$ = –5.2 mA | 6 V | 5.48 | 5.8 | | 5.2 | | 5.34 | | |
| $V_{OL}$ | $V_I$ = $V_{IH}$ or $V_{IL}$ | $I_{OL}$ = 20 µA | 2 V | | 0.002 | 0.1 | | 0.1 | | 0.1 | V |
| | | | 4.5 V | | 0.001 | 0.1 | | 0.1 | | 0.1 | |
| | | | 6 V | | 0.001 | 0.1 | | 0.1 | | 0.1 | |
| | | $I_{OL}$ = 4 mA | 4.5 V | | 0.17 | 0.26 | | 0.4 | | 0.33 | |
| | | $I_{OL}$ = 5.2 mA | 6 V | | 0.15 | 0.26 | | 0.4 | | 0.33 | |

**Figure 2.5.4** Excerpt from the data sheet for a 74HC04 CMOS hex inverter integrated circuit (Texas Instruments), showing the output current limits.

In contrast, the TTL implementation of the same inverter function [13] has asymmetric output current specifications Figure 2.5.5, p. 40: $I_{OH}$ for the LS04 is 400$\mu$A, but $I_{OL}$ is considerably larger, 8mA. (The physical reasons for the asymmetry are beyond the scope of this course; a good explanation may be found in the text by Wakerly [16], pp. 160-169.)

| | | SN54LS04 | | | SN74LS04 | | | UNIT |
|---|---|---|---|---|---|---|---|---|
| | | MIN | NOM | MAX | MIN | NOM | MAX | |
| $V_{CC}$ | Supply voltage | 4.5 | 5 | 5.5 | 4.75 | 5 | 5.25 | V |
| $V_{IH}$ | High-level input voltage | 2 | | | 2 | | | V |
| $V_{IL}$ | Low-level input voltage | | | 0.7 | | | 0.8 | V |
| $I_{OH}$ | High-level output current | | | −0.4 | | | −0.4 | mA |
| $I_{OL}$ | Low-level output current | | | 4 | | | 8 | mA |
| $T_A$ | Operating free-air temperature | −55 | | 125 | 0 | | 70 | °C |

**Figure 2.5.5** Excerpt from the data sheet for a 74LS04 TTL hex inverter integrated circuit (Texas Instruments), showing the output current limits.

In both technologies, the output current of a logic gate is more than sufficient to drive the inputs of many logic gates at once; the input current requirements of logic gates are quite modest, on the order of nanoamps for CMOS devices and microapmps for TTL. A single LED requires more current, up to 10mA or more, and additional circuitry is required to go between a logic gate and a motor or large LED display. These design problems are explored in the lab.

## 2.6 Timing characteristics of logic circuits

An ideal gate represents a truth table, which is simply an input-output relationship. The output is presumed to change simultaneously with a causative change in an input. In an actual logic circuit, however, the output changes a brief time after the input changes. This time is called the **propagation delay** of the circuit. Propagation delay is at the root of what we are talking about when we say a digital system is "fast" or "slow".

Propagation delay is illustrated in Figure 2.6.1, p. 41, below. An inverter was wired up on a solderless breadboard with its input driven by a function generator, and its output viewed on an oscilloscope. The small oscillations around the LOW and HIGH voltage levels are typical of real logic circuits, and are within the limits specified by $V_{IH}$ and $V_{IL}$ for correct operation. It is evident that the output transitions occur a measurable time after the input transitions.

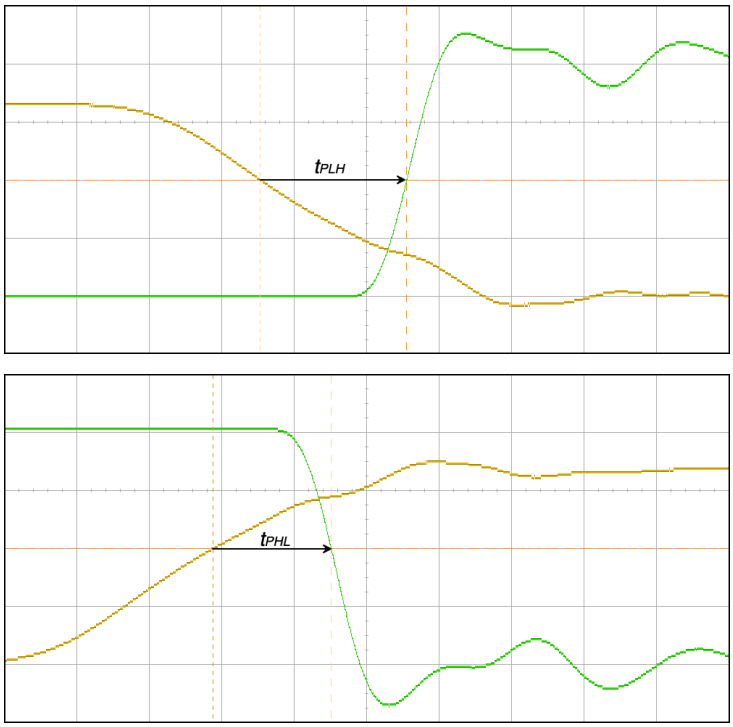



**Figure 2.6.1** Propagation delay. A function generator drives a CMOS inverter (74HC04) and the output is viewed on an oscilloscope. The horizontal scale is 5ns/division, the vertical scale is 1V/division. The orange trace is the input, and the green trace is the output. (Top) The LOW-to-HIGH delay, $t_{PLH}$, is about 10ns. (Bottom) The HIGH-to-LOW delay, $t_{PHL}$, is about 8ns.

A transition in a signal from LOW to HIGH is called a **rising edge** or **positive edge**. A transition from HIGH to LOW is called a **falling edge** or **negative edge**. Propagation delays are measured from an edge of the input to the next edge of the output, as shown in Figure 2.6.2, p. 42. The delay from an input edge of either polarity to a rising edge at the output is called $t_{PLH}$ (LOW-to-HIGH). If the output edge is falling, the delay is called $t_{PHL}$ (HIGH-to-LOW). The designation of a delay as $t_{PLH}$ or $t_{PHL}$ depends only on the polarity of the output edge.

Propagation delays are defined between particular points on the rising or falling edges. For LS-series devices, they are measured between the point where the input crosses 1.3V and the point where the output crosses 1.3V. For the HC-series CMOS family, delays are measured between the points where input and output cross $V_{DD}/2$ (in Figure 2.6.1, p. 41. the supply voltage is 4V and the threshold is 2V). Other logic families may define these reference points in yet other ways, so always check the datasheet.

Historically, the supply voltage for bipolar logic, incuding LSTTL, is denoted $V_{CC}$. The $V_{CC}$ notation is also often used instead of $V_{DD}$ to denote the positive supply voltage in HC-series (CMOS) logic.

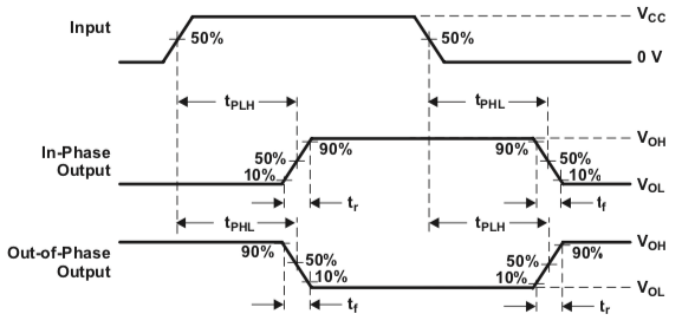



**Figure 2.6.2** A portion of the data sheet for a Texas Instruments SN74HC04 CMOS hex inverter integrated circuit [13], showing how the propagation delays $t_{PLH}$ and $t_{PHL}$ are measured.

Typical and maximum (worst case) values for propagation delay are tabulated in datasheets (for example, Figure 2.6.3, p. 42). Sometimes, no distinction is made between $t_{PLH}$ and $t_{PHL}$, and the delay is denoted by a single symbol such as $t_d$, $t_p$, or $t_{pd}$. According to the table in Figure 2.6.3, p. 42, propagation delay depends on the supply voltage, $V_{CC}$. The typical values for $t_{PLH}$ and $t_{PHL}$ are about 9ns for the conditions ($V_{CC} = 4$V) used for the measurements in Figure 2.6.1, p. 41. The measured values, 8ns and 10ns, are within the typical range for this device, according to the datasheet.

| PARAMETER | FROM (INPUT) | TO (OUTPUT) | $V_{CC}$ | $T_A$ = 25°C | | |
|---|---|---|---|---|---|---|
| | | | | MIN | TYP | MAX |
| $t_{pd}$ | A | Y | 2 V | | 45 | 95 |
| | | | 4.5 V | | 9 | 19 |
| | | | 6 V | | 8 | 16 |

**Figure 2.6.3** A portion of the data sheet for a Texas Instruments SN74HC04 hex inverter integrated circuit, showing values for the propagation delays $t_{PLH}$ and $t_{PHL}$ (denoted $t_{pd}$).

The propagation delay for a circuit consisting of one gate driving another is the sum of their individual propagation delays. For a combination of two HC04 inverters, the typical delay would be 18ns, and the maximum (worst case) delay would be 38ns.

Propagation delays in a circuit are a combination of logic delays, related to how fast a transistor can switch between ON and OFF, and wire delays, which are the time required for a signal to propagate from gate to gate within an integrated circuit, and from package to package on a circuit board. The velocity of propagation on a wire is less than the speed of light, and can be measured in the lab.

The physical origins of logic delays and wire delays are beyond the scope of this course, but may be understood using physical principles covered in Phys 14 and Engs 22. Sections 4.2 and 5.1 of Reference [2] provide more details for the interested reader.

## 2.7 Power consumption of logic circuits

(∗) Static and dynamic power

## 2.8 Logic and voltage, again

(∗) High-true and low-true signals

# Chapter 3

# Efficient Logic

In Section 1.3, p. 8 and Section 1.4, p. 11 we saw how truth tables and logic equations are used to capture the behavior of a digital design, and then, in Section 1.5, p. 14, how the equations are translated into logic diagrams for gate-level implementation. Chapter 2, p. 21 described logic gate circuits and their important electrical characteristics: input and output voltages and currents, propagation delay, and power consumption.

It is, of course, of primary importance that a design be logically correct and perform the function it is designed to do. Following closely behind are three very practical specifications:

- Speed: Does the system perform its functions in an acceptable amount of time?
- Power: Does the system consume an acceptable amount of power?
- Size: Does the system fit into a package of acceptable size?

Your phone, for example, must be very fast in order to do things like stream video, it must be powered by a battery and give off very little waste heat, and it must be small enough to fit into your pocket. Speed, power, and size are not independent. Higher speeds tend to require more power. On the other hand, simpler circuits require fewer transistors and less wiring, reducing size and increasing speed. Making the tranistors themselves smaller increases speed and enables more functionality to be packed into a smaller space.

This chapter focuses on efficient gate-level design: how to go from truth table to logic equation to the simplest possible logic diagram. We will see two methods for "by-hand" logic minimization based on Boolean algebra. Modern EDA tools, based on the same principles, automate logic minimization for complex designs.

## 3.1 Truth tables, again

What we know, so far (Section 1.4, p. 11):

- A logical variable $x$ is either 0 (FALSE) or 1 (TRUE).
- Logical variables are combined into logical expressions with AND ($\cdot$), OR ($+$), and NOT ($'$, —) operations, e.g., $x \cdot y \cdot \overline{z}$. The "$\cdot$" symbol in the AND of two variables may be omitted, e.g., $xy$ is equivalent to $x \cdot y$.

- An OR of ANDs, e.g., $w \cdot x + \overline{y} \cdot z$, is called a sum of products. Each row of a truth table is represented by a product term (AND), and the logic equation for a truth table is a sum (OR) of products.

A truth table with $N$ inputs has $2^N$ rows, and logic expression with $N$ variables can become quite lengthy, up to $2^N$ product terms. We saw in Section 1.3, p. 8 that sometimes, a long truth table can be simplified by merging rows with "don't care" entries. This also leads to simpler logic equations. The process of simplifying a logic equation is called **logic minimization**.

To illustrate, we will revisit Checkpoint 1.3.5, p. 10, the two-input multiplexer. The full truth table is

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

and using don't care entries, the truth table simplifies to four rows.

| $S$ | $A$ | $B$ | $y$ |
|---|---|---|---|
| 0 | 0 | — | 0 |
| 0 | 1 | — | 1 |
| 1 | — | 0 | 0 |
| 1 | — | 1 | 1 |

The logic equations for the full and simplified truth tables are, respectively,

$$y = \overline{S}A\overline{B} + \overline{S}AB + S\overline{A}B + SAB$$
$$y = \overline{S}A + SB$$

The first equation requires four three-input AND gates, a four-input OR gate, three NOTs, and all the wiring to connect them together. The second equation requires only two two-input AND gates, two-input OR gate, and one NOT. That is quite a simplification!

**Logical adjacency.** Two binary codes are called **logically adjacent** if they differ in only one bit. In the multiplexer example, the binary codes for the third and fourth rows of the full truth table, "010" and "011" are logically adjacent. Two product terms are logically adjacent if they differ only by inverting one variable, e.g., $\overline{S}A\overline{B}$ and $\overline{S}AB$. The value of $B$ is irrelevant, since $y$ is TRUE whether $B$ is 0 or 1, and the sum $\overline{S}A\overline{B} + \overline{S}AB$ simplifies to $\overline{S}A$. Likewise, the binary codes for the sixth and eight rows, "101" and "111", are logically adjacent, differing only in the $A$ bit. Their sum of products simplifies, $S\overline{A}B + SAB = SB$.

**Checkpoint 3.1.1** Identify all the pairs of rows in the multiplexer truth table that are logically adjacent and have the same output. Write simplified logical expressions for each pair.

**Answer**. Producing output $y = 0$: "000" and "001", $\overline{SA}$; "000" and "100", $\overline{AB}$; "010" and "110", $A\overline{B}$; "100" and "110", $S\overline{B}$.

Producing output $y = 1$: "010" and "011", $\overline{S}A$; "011" and "111", $AB$; "101" and "111", $SB$.

Discovering and exploiting logical adjacency is the key to logic minimization. However, it rapidly becomes unwieldy for even moderate-sized truth tables, as this next example shows. In the $x > y$ function you designed in Checkpoint 1.3.4, p. 8, the truth table was:

| $x_1$ | $x_0$ | $y_1$ | $y_0$ | $g$ | | $x_1$ | $x_0$ | $y_1$ | $y_0$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | | 1 | 1 | 1 | 1 | 0 |

**Figure 3.1.2** Truth table for a two-bit comparison function, $x > y$.

Observe that if $x = 00$, then $g$ will be zero regardless of the value of $y$. We may compress all of the rows with $x = 00$, as shown below. Again, the dashes indicate that we don't care what the values of those inputs are.

| $x_1$ | $x_0$ | $y_1$ | $y_0$ | $g$ |
|---|---|---|---|---|
| 0 | 0 | — | — | 0 |
| 0 | 1 | 0 | 0 | 1 |
| ⋮ | | | | |

Likewise, if $x = 01$ and $y_1 = 1$, then $g$ will be zero regardless of $y_0$, and another pair of rows can be compressed:

| $x_1$ | $x_0$ | $y_1$ | $y_0$ | $g$ |
|---|---|---|---|---|
| 0 | 0 | — | — | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | — | 0 |
| ⋮ | | | | |

Compressing the remainder of the rows is left as an exercise.

**Checkpoint 3.1.3** Complete the compression of the truth table for the two-bit comparison by pairing rows and using "don't care" entries. Then write a logic equation.

**Answer**. There are a few ways to do this. Here is one.

| $x_1$ | $x_0$ | $y_1$ | $y_0$ | $g$ |
|---|---|---|---|---|
| 0 | 0 | — | — | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | — | 0 |
| 1 | 0 | 0 | — | 1 |
| 1 | 0 | 1 | — | 0 |
| 1 | 1 | 0 | — | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

The logic equation is

$$g = x_1' x_0 y_1' y_0' + x_1 x_0' y_1' + x_1 x_0 y_1' + x_1 x_0 y_1 y_0'.$$

As the exercise showed, some logical adjacencies are harder to spot from a truth table than others. It gets even harder as the number of variables increases. There is a better way.

## 3.2 Algebraic minimization

A logic equation is an algebraic representation of a truth table. Like a truth table, a logic equation is simplified by identifying logically adjacent product terms and combining them into simplified terms, e.g., $XYZ' + XYZ = XY$.

**Boolean algebra.** Logical variables (e.g., $A, B, C$), the logic values (0 and 1), and the operations (AND, OR, NOT) have a systematic structure called **Boolean algebra**. Like high school algebra, Boolean algebra has a number of rules that allow logical expressions to be manipulated and simplified. They are listed in Table 3.2.1, p. 48, below, and may all be verified with truth tables. Some are familiar from high school algebra, others are unique to Boolean algebra. The final item in the list, **DeMorgan's Rules**, is included here for completeness but will be described in a later section.

**Table 3.2.1 Rules of Boolean Algebra**

| | | |
|---|---|---|
| Identities with 0 and 1 | $A \cdot 0 = 0$<br>$A \cdot 1 = A$ | $A + 0 = A$<br>$A + 1 = 1$ |
| Double negative | $(A')' = A$ | |
| Idempotent | $A \cdot A = A$ | $A + A = A$ |
| Complementary | $A \cdot A' = 0$ | $A + A' = 1$ |
| Commutative | $A \cdot B = B \cdot A$ | $A + B = B + A$ |
| Associative | $A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$ | $A + B + C = (A + B) + C = A + (B$ |
| Distributive | $A \cdot (B + C) = A \cdot B + A \cdot C$ | $A + (B \cdot C) = (A + B) \cdot (A + C)$ |
| DeMorgan's Rules | $\overline{A \cdot B} = \overline{A} + \overline{B}$ | $\overline{A + B} = \overline{A} \cdot \overline{B}$ |

The basic simplification of logically adjacent terms, $AB + AB'$, follows from an application of the distributive law, to factor out the A, and then complementarity, and finally, identity:

$$AB + AB' = A(B + B') = A \cdot 1 = A$$

All logic minimization, fundamentally, comes down to this one move: factor and combine.

**Examples of algebraic simplification.** The other rules of Boolean algebra are applied in more complex situations to reveal the basic move. See if you can spot the rules that are applied in the following simplification (parentheses for emphasis):

$$ABC + ABC' = (AB)C + (AB)C' = AB(C + C') = (AB) \cdot 1 = AB$$

This one has a few more steps:

$$\begin{aligned} ABC + AB'C &= A(BC) + A(B'C) = A(CB) + A(CB') = ACB + ACB' \\ &= AC(B + B') = AC \cdot 1 = AC \end{aligned}$$

Of course, all the steps between $ABC + AB'C$ and $AC(B + B')$ are intuitive by analogy with high school algebra, and don't need to be written out in such detail (unless you need to in order to be confident of your work).

A trickier situation arises when you have three terms, like this: $ABC + ABC' + AB'C$. You could combine the first two terms, giving

$$ABC + ABC' + AB'C = AB + AB'C$$

or the first and third terms, resulting in

$$ABC + ABC' + AB'C = AC + ABC'$$

In hardware, both would require the same gates, and neither is preferable to the other. However, there is something special about Boolean algebra that enables an even simpler form to be found. The idempotency rule, $A + A = A$, permits a term in an expression to be copied and used twice:

$$\begin{aligned} ABC + ABC' + AB'C &= (ABC + ABC) + ABC' + AB'C \\ &= (ABC + ABC') + (ABC + AB'C) = AB(C + C') + AC(B + B') \\ &= AB + AC \end{aligned}$$

How about that!

The complementarity rule is helpful when you're given something like this: $A + A'B$, which, on the face of it, doesn't look like it can be simplified. But note:

$$\begin{aligned} A + A'B &= A \cdot 1 + A'B = A(B + B') + A'B = AB + AB' + A'B \\ &= (AB + AB) + AB' + A'B = (AB + AB') + (AB + A'B) \\ &= A(B + B') + (A + A')B = A + B \end{aligned}$$

Finally, simplifications can lead to more simplifications, as in this example:

$$\begin{aligned} ABC + ABC' + AB'C + AB'C' &= A(BC + BC' + B'C + B'C') \\ &= A(B(C + C') + B'(C + C')) \\ &= A(B + B') = A \end{aligned}$$

Once you've factored out the $A$, every combination of $B$ and $C$ is there, and for any choice of $B$ and $C$, one of them will be true!

All of these examples are variations on the same theme: to simplify a logic expression, look for ways to factor and combine, expanding (with idempotency and complementarity) if necessary to reveal additional combinations. And if you are ever in doubt as to whether your results are correct, you can always resort to a truth table.

**Checkpoint 3.2.2** Minimize the logic equation for the two-bit odd number detector, Checkpoint 1.4.2, p. 13.

**Solution**. Apply the basic factor-combine move,

$$y = (\overline{x_1} \cdot x_0) + (x_1 \cdot x_0) = (\overline{x_1} + x_1) \cdot x_0 = x_0$$

Of course, a number is odd if its least=significant bit is 1!

**Checkpoint 3.2.3** Write the full logic equation for the two-input multiplexer, then minimize it using Boolean algebra.

**Solution**. The full equation, from the truth table, Checkpoint 1.3.5, p. 10, is

$$y = S'AB + S'AB' + SA'B + SAB.$$

The minimized equation is

$$y = S'A(B + B') + SB(A + A') = S'A + SB$$

**Checkpoint 3.2.4** Write the full logic equation for the two-bit $x > y$ function, Checkpoint 1.3.4, p. 8, then minimize it using Boolean algebra. Compare your result with the compressed truth table, Checkpoint 3.1.3, p. 47

**Solution**. From the truth table, the unreduced equation is

$$g = x_1'x_0y_1'y_0' + x_1x_0'y_1'y_0' + x_1x_0'y_1'y_0 + x_1x_0y_1'y_0' + x_1x_0y_1'y_0 + x_1x_0y_1y_0'$$

Look for factorizations, copying terms that can be used twice:

$$g = x_1y_1'(x_0'y_0' + x_0'y_1 + x_0y_1' + x_0y_1) + (x_1' + x_1)x_0y_1'y_0' + x_1y_1'y_0'(x_0' + x_0)$$

After combining,

$$g = x_1y_1' + x_0y_1'y_0' + x_1y_1'y_0'$$

For comparison, writing an equation from the compressed truth table gives:

$$g = x_1'x_0y_1'y_0' + x_1x_0'y_1' + x_1x_0y_1' + x_1x_0y_1y_0'$$

It isn't as simple as the one derived from Boolean algebra, and in fact, there appear to be no common terms! But both can be confirmed by going back to the original truth table.

The last exercise was fairly complicated. If you didn't get my answer, you might have a sinking feeling that Boolean algebra is a difficult path to minimized logic. You might also be dismayed that the result did not match up with the compressed truth table derived previously. Your feelings are justified:

- There are multiple ways to pair rows in a truth table to reveal don't care entries.
- It can be very difficult to discover the best simplification by mere inspection of the truth table.
- Following the steps of Boolean algebra can also result in multiple results, all logically equivalent, but differing in complexity.

What to do? Once again, it must be emphasized that by-hand manipulation of truth tables and logic equations, while instructive, do not guarantee optimal results and are not how complex digital systems are designed. But before leaving the subject, there is one more by-hand method, that points more directly to computational simplification, and is also kind of fun.

## 3.3 Karnaugh maps

What we now know about logic minimization:

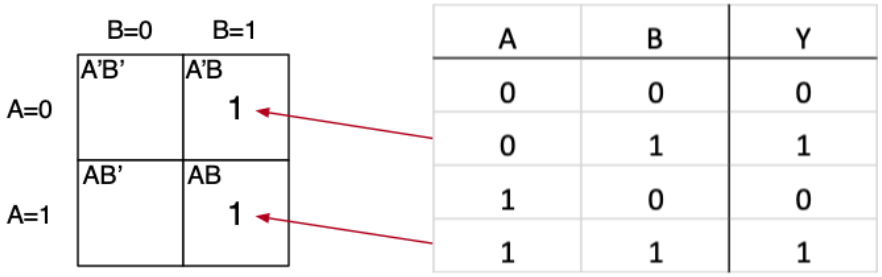
- Logically adjacent rows of a truth table can be merged into a single row containing a "don't care" entry for one of the variables.
- Equivalently, a logically adjacent pair of terms in an equation can be merged into single term with one fewer variable.
- Potential simplifications can be hard to spot, and results are not unique.

And some things we wish we knew:

- How can we systematically find all possible simplifications?
- How can we know if an equation is in its simplest possible form?

We will answer these questions, at least for fairly simple systems, in this section.

**The basic Karnaugh map.** The **Karnaugh map** is a graphical representation of a truth table. It consists of an array of cells, one for each row of the table, Figure 3.3.1, p. 51




| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Figure 3.3.1** Karnaugh map and truth table for $Y = AB + A'B = B$. Each cell of the Karnaugh map corresponds to one row of the truth table.

Each cell of the Karnaugh map represents one row of the truth table, and one possible combination of the input variables $A$ and $B$ in a logic equation. The cells are all logically adjacent: on the top row, $A'B'$ is adjacent to $A'B$, and on the bottom row, $AB'$ is adjacent to $AB$. In the left column, $A'B'$ is adjacent to $AB'$, and in the right column, $A'B$ is adjacent to $AB$. The important difference between the truth table and the Karnaugh map is that, in a Karmaugh map, logically adjacent cells are also physically adjacent, whereas in the truth table they may or or may not be. In the figure, for example, the second and fourth rows of the truth table are logically adjacent but not physically adjacent, but their cells in the Karnaugh map are both.

With the practice you've had so far, it isn't hard to inspect the truth table and see that as long as $B$ is TRUE, $Y$is TRUE, regardless of the value of $A$. Also, you know how to write the logic equation from the truth table and simplify it by Boolean algebra: $A'B + AB = (A' + A)B = 1 \cdot B = B$. Here is how the Karnaugh map reveals the same simplification:

- The truth table is transferred to the Karnaugh map by writing a 1 in each cell that corresponds to a row of the truth table for which $Y = 1$. The unreduced logic equation is the OR of the terms represented by each cell. Here, $Y = A'B + AB$.
- Two cells are merged if they contain 1s and are physically adjacent. The two cells in the right column, holding 1s, $A'B$ and $AB$, are merged into a larger group.
- The truth of the larger group depends on fewer variables. If $B$ is TRUE, then one of these cells will be TRUE; thus, $B$ suffices to define the group. The simplified expression for the larger group is $B$, and the simplified equation is $Y = B$.

The simplification is accomplished simply by spotting adjacent 1s on the map and collecting them into larger groups. No algebra required.

Recall an example from the previous section, the nonintuitive simplification $A + A'B = A + B$. It hinged on expanding $A = A(B + B')$, and then copying

$AB$ so it could be used in two combinations:

$$\begin{aligned} A + A'B &= A \cdot 1 + A'B = A(B + B') + A'B = AB + AB' + A'B \\ &= (AB + AB) + AB' + A'B = (AB + AB') + (AB + A'B) \\ &= A(B + B') + (A + A')B = A + B \end{aligned}$$

Figure 3.3.2, p. 52 shows the same simplification, on a Karnaugh map. The term $A$ is TRUE in the two cells in the bottom row. $A'B$ is in the upper right corner, where it is visually apparent that it is adjacent to $AB$ and can be grouped into a pair, using the cell for $AB$ twice. $B$ is TRUE for these two cells. If either $A = 1$ or $B = 1$, then $Y = 1$. The simplified expression $Y = A + B$. Much easier than the algebraic simplification.

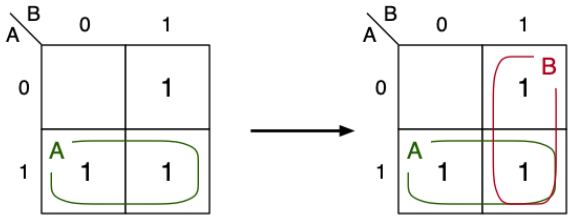



**Figure 3.3.2** Karnaugh map for $Y = A + A'B$ (left), regrouped into the simplified expression $A + B$ (right). The cell for $AB$ is used twice.

**Larger Karnaugh maps.** The Karnaugh map for a function of three variables is shown in Figure 3.3.3, p. 52

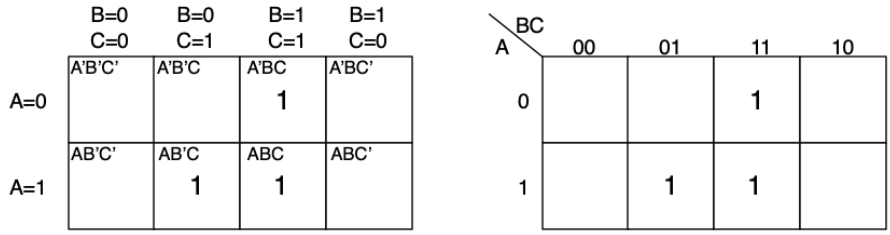



**Figure 3.3.3** Karnaugh map for $Y = AB'C + ABC + A'BC$. The map on the right is drawn in a standard way.

First, look at the map on the left, which is a three-variable extension of the version in Figure 3.3.1, p. 51. With three variables, there are $2^3 = 8$ possible combinations of the input variables $A, B, C$. Consequently, there are eight cells in the Karnaugh map. They are organized as two rows of four. Each cell in the top row is logically adjacent to the cell below it in the bottom row. Moving across the top row, each cell is logically adjacent to the cell to either side in the row, and the cells at each end of the row are also logically adjacent. To accomplish this, the columns are ordered in a non-binary sequence: $BC = 00, 01, 11, 10$. This is important. If you set up a grid with the columns ordered in a binary sequence, $BC = 00, 01, 10, 11$, the second and third columns will not be logically adjacent, the first and fourth columns will not be logically adjacent, and you will obtain incorrect results.

The map on the right is drawn the way Karnaugh maps typically appear. With practice, it is no harder to use, and considerably faster to draw. But of course, you can always add the additional notation if it helps you.

The individual 1s in the figure represent the logic equation $Y = AB'C + ABC + A'BC$, which we earlier saw has the simplified form $Y = AB + AC$.

Here we will use the Karnaugh map to accomplish the simplification without algebra. We look for sets of logically adjacent cells, and find two, shown in Figure 3.3.4, p. 53.

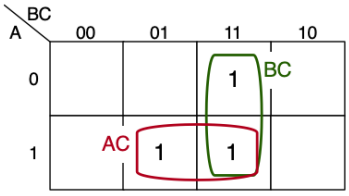



**Figure 3.3.4** Karnaugh map for $Y = AB'C + ABC + A'BC$. Cells are grouped into the simplified expression $AC + BC$. The $ABC$ cell is used twice.

The cell for $ABC$ is used twice, just as the term $ABC$ was used twice in the algebraic simplification. Each pair of cells is represented by a term made of only two variables: $AB'C + ABC = AC$, and $A'BC + ABC = BC$. The simplified equation is the OR of the two groups, $y = AC + BC$.

**Checkpoint 3.3.5** Group cells in these three-variable Karnaugh maps and write the reduced logic expression

**Answer**. Lorem ipsum

A four-variable Karnaugh map is a $4 \times 4$ array of cells, as shown in Figure 3.3.6, p. 53

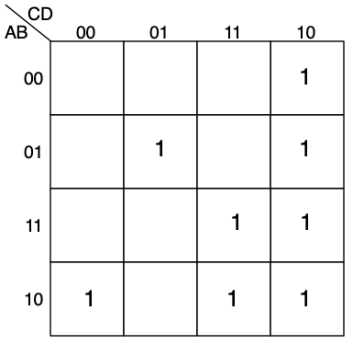



**Figure 3.3.6** Karnaugh map for a function of four variables.

Cells are logically adjacent to their north-south-east-west neighbors, including wrap-around at the edges. With the larger playing field, larger groups are possible: sixteen, eight, four, two, and one. The example in Figure 3.3.7, p. 54 shows some of these.

- There is a clear group of four in the rightmost column, simplifying to $CD'$. Algebraically, $A'B'CD' + A'BCD' + ABCD' + AB'CD' = (A'B' + A'B + AB + AB') \cdot CD' = CD'$. Without using algebra, note that one of the four cells will be TRUE if $CD'$ is TRUE, so $CD'$ suffices to define the group.
- The cells for $ABCD'$ and $AB'CD'$ are reused, to expand the pair of $ABCD$ and $AB'CD$ into a group of four in the lower right corner, simplifying to $AC$.
- The cell in the lower right corner, $AB'C'D'$, is logically adjacent to the other corner $AB'CD'$, forming a wraparound group of two, $AB'D'$.

- Finally, the single cell $A'BC'D$ is not adjacent to any other cells and cannot be simplified.

The simplified expression is the OR of these simpler terms, $AC+CD'+AB'D'+A'BC'D$.

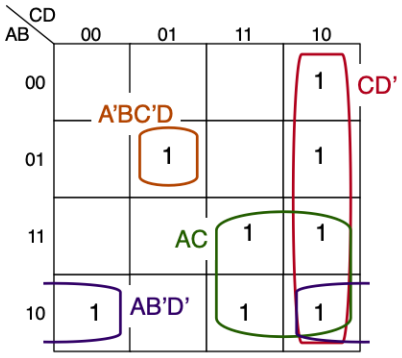
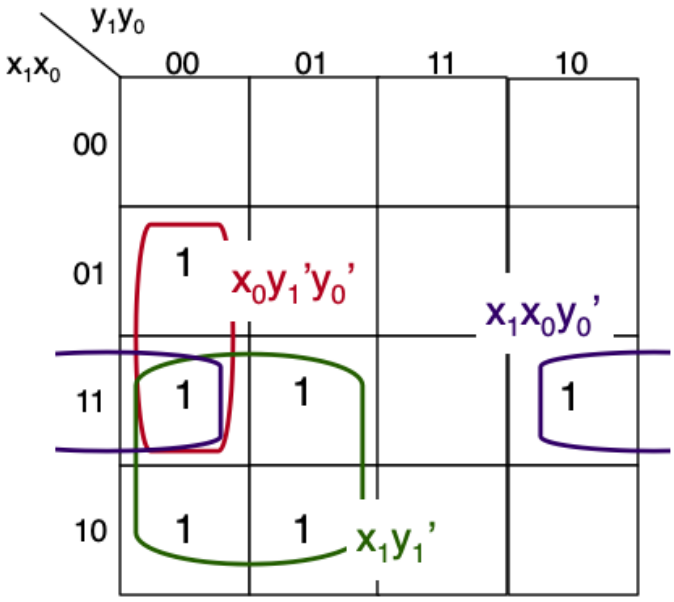



**Figure 3.3.7** Karnaugh map for a function of four variables, with groups of 1s collected.

Always try to reuse cells to create the largest possible groups, since large groups are represented by simple product terms.

**Checkpoint 3.3.8** Return again to the two-bit $x > y$ circuit, Checkpoint 3.2.4, p. 50. Transfer the truth table to a Karnaugh map and derive a simplified logic equation.

**Solution**.




The reduced logic equation is $g = x_1y_1' + x_0y_1'y_0' + x_1x_0y_0'$.

**Checkpoint 3.3.9** We are always seeking groupings of size 1, 2, 4, 8, or even 16, but never 3, 5, 7, 9, 10, etc. Why is this?

**Solution**. For example, on a three-variable map, try to simplify $AB'C' + AB'C + ABC$ into a single product term.

**Checkpoint 3.3.10** Why can't two cells on a diagonal be grouped?

**Answer**. They are not logically adjacent, but are separated by two bits, e.g., $AB' + A'B$.

Finding logically adjacent terms to combine is easier on a Karnaugh map---just look for physically adjacent cells. We now have a method for simplifying logic, but aren't quite finished. In a complex situation, how can you be sure you've found the simplest set of groups, leading to the simplest logic?

## 3.4 DeMorgan's Rules

We return now to the final item in the list of Boolean rules Table 3.2.1, p. 48, known as DeMorgan's rules :

$$\overline{A \cdot B} = \overline{A} + \overline{B} \qquad \text{and} \qquad \overline{A + B} = \overline{A} \cdot \overline{B}$$

They can be formally confirmed with truth tables, but also understood intuitively. We know that $A$ and $B$ must both be true in order for $A \cdot B$ to be true. If either $A$ or $B$ is false, therefore, $A \cdot B$ will be false. Translated into algebra, we have the first rule, $\overline{A \cdot B} = \overline{A} + \overline{B}$. For the second rule, if both $A$ and $B$ are false, then $A + B$ will be false. Translating into algebra, we have $\overline{A + B} = \overline{A} \cdot \overline{B}$.

**Checkpoint 3.4.1** Confirm DeMorgan's rules with a truth table

**Solution**.

| $A$ | $B$ | $\overline{AB}$ | $\overline{A+B}$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

DeMorgan's rules can also be visualized on a Karnaugh map, Figure 3.4.2, p. 55.

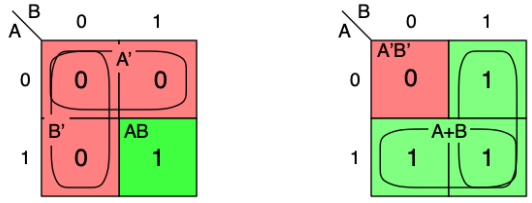



**Figure 3.4.2** Karnaugh maps illustrating DeMorgan's rules. (Left:) The red cells, where $AB$ is FALSE, are covered by $A' + B'$. (Right:) The red cell, where $A + B$ is FALSE, is $A'B'$.

**Checkpoint 3.4.3** DeMorgan's rules apply to more than two variables. Confirm, by Boolean algebra, that they work for three or more variables:

$$\overline{ABC\ldots} = \overline{A} + \overline{B} + \overline{C} + \ldots \quad \text{and} \quad \overline{A + B + C + \ldots} = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \ldots$$

**Solution**. Here is the algebraic proof for the first rule, with three variables. Proof for the second rule is similar.

$$\begin{aligned}(A \cdot B \cdot C)' &= ((A \cdot B) \cdot C)' = (A \cdot B)' + C' \\ &= A' + B' + C'\end{aligned}$$

Once you see how it's done for three variables, four or more follow the same pattern.

**DeMorgan equivalent gates.** DeMorgan's rules give a valuable interpretation of NAND and NOR gates, Figure 3.4.4, p. 56. The bubble at the output of a NOT gate indicates logical inversion. A bubble may also be placed on an AND or OR gate to invert an input or output. Thus, a NAND gate is an AND gate with a bubble on the output, and a NOR gate is an OR gate with a bubble on the output.

Carrying the "bubble = NOT" symbolism farther, the expression $A' + B'$ represents an OR gate with inverting bubbles on its inputs. DeMorgan's rule, $(AB)' = A' + B'$, says that this is equivalent to a NAND gate. We say that the AND with bubbled output and the OR with bubbled inputs are **DeMorgan equivalent** symbols for the NAND function. In the same way, the other DeMorgan's rule, $(A + B)' = A'B'$, says that the OR with bubbled output and the AND with bubbled inputs are DeMorgan equivalents for the NOR function.

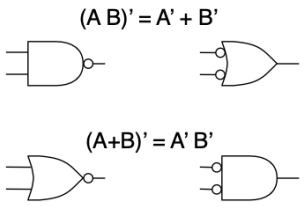



**Figure 3.4.4** DeMorgan equivalent gate symbols. (Top:) By DeMorgan's rule, $(A \cdot B)' = A' + B'$, an AND gate with an output bubble is equivalent to an OR gate with input bubbles. (Bottom:) By DeMorgan's rule, $(A + B)' = A' \cdot B'$, an OR gate with an output bubble is equivalent to an AND gate with input bubbles.

**Sum of products.** The DeMorgan equivalent representations of NAND and NOR enable a sum of products, $Y = AB + CD$, to be implemented solely with NAND gates, Figure 3.4.5, p. 56. In the figure, the sum of products $Y = AB + CD$ is initially implemented with two AND gates and one OR gate. The logic is unaffected by placing an inverting bubble at the end of each of the wires connecting the AND gates to the OR gate. The inversion at the output of the AND is undone by the inversion at the input of the OR. The output bubble converts the two ANDs to NANDs. Also, recognize that the OR with bubbled inputs is the DeMorgan equivalent symbol for a NAND.

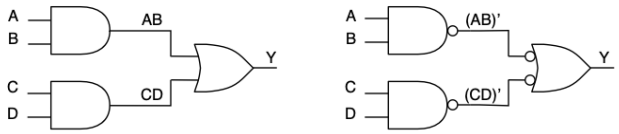


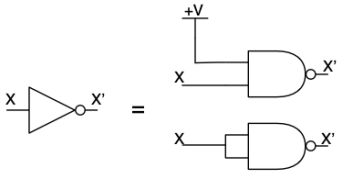

**Figure 3.4.5** Using DeMorgan's rule, $(A \cdot B)' = A' + B'$, to implement a sum-of-products with NAND gates only. (Left:) A sum of products, $Y = AB + CD$, implemented with two AND gates and one OR gate. (Right:) Placing inverting bubbles at the outputs of the AND gates and the inputs of the OR gate change all three gates to NANDs, but does not change the logic.

What difference does this make? Two things: First, the circuit is constructed from only one kind of circuit, NAND, rather than two, AND and OR. This simplifies the layout of the circuit. Second, we have seen that a two-input NAND gate requires only four transistors, whereas the two-input AND and OR each require six (Section 2.4, p. 33). The transistor count is reduced from eighteen to twelve, a signficant reduction in complexity and area.

When we first met the CMOS gate circuits in Section 2.4, p. 33, it seemed

awkward for the simplest gate circuits to be NAND and NOR. Now, thanks to DeMorgan's rules, we see that they are actually superior to AND and OR. Any logic equation can be implemented using only NAND gates by first converting it to a sum of products. If it is necessary to invert a signal, e.g., $y = A'B + AC$, a NOT gate may be used.

**Checkpoint 3.4.6** In a pinch, a NOT gate may also be replaced by a NAND. Show that both of the circuits below are equivalent to a NOT gate.




**Solution**. In the upper circuit, $X \cdot 1 = X$ (identity), therefore $\overline{X \cdot 1} = \overline{X}$. In the lower circuit, $X \cdot X = X$ (idempotency), therefore $\overline{X \cdot X} = \overline{X}$.

A final note about writing a sum of products using NAND gates. Using the DeMorgan equivalents, you can see that the circuit is a sum of products, and that the bubbles at the ends of the wires "cancel". Beginners will often convert the OR with bubbled inputs back to the other form, the AND with bubbled output, resulting in the circuit shown in Figure 3.4.7, p. 57 This does not affect the underlying transistor circuit, but the diagram no longer visually communicates that the circuit is a sum of products. Don't do this.

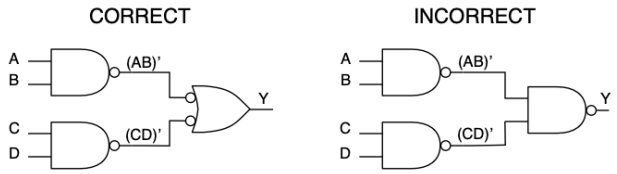



**Figure 3.4.7** The right way and wrong way to write a sum of products using NAND gates. (Left:) The DeMorgan equivalent symbols visually communicate that the circuit is a sum of products. This is the correct way to draw the circuit. (Right:) Replacing the final gate with the other DeMorgan form does not change the underlying circuit, but it obscures the meaning of the logic diagram.

**Checkpoint 3.4.8** Draw the logic diagram for $y = A'B + AC$, using three NAND gates (with correct DeMorgan symbols) and a NOT gate.

**Solution**.

**Checkpoint 3.4.9** Redraw the simplified two-bit $x > y$ equation, Checkpoint 3.3.8, p. 54, using NAND gates only (with correct DeMorgan symbols), and NOT gates, as needed.

**Solution**.

## 3.5 Optimum minimization

**Implicants.** The question, "What is the simplest reduced form of a logic equation?" does not have a unique answer. There can be more than one good solution, requiring the same amount of circuitry. So what constitutes even one simplest solution? To answer this precisely, it helps to introduce some terminology, illustrated in Figure 3.5.1, p. 58.

- A **minterm** is a product term that occupies only one cell on a Karnaugh map. A minterm also corresponds to one row of a truth table. In the figure, $AB'C'D'$ and the other cells containing 1s are minterms.
- An **implicant** is any product term (a single minterm is a trivial product term) that makes an expression TRUE. In the expression $A + A'B$, $A$ and $A'B$ are implicants. In the figure, $AB'C'D'$. $BC'D$, $ABD$, and $AC$ are implicants.
- A **prime implicant** is an implicant that cannot be combined with another implicant to make a product term with fewer variables. In the figure, $BC'D$, $ABD$, and $AC$, are all prime implicants. While $AB'C'D'$ is an implicant, it is not a prime implicant, since it can be grouped with $AB'CD'$ to make $AB'D'$ (which is prime).
- An **essential prime implicant** is a prime implicant that covers at least one cell not covered by any other implicant. It is "essential" because removing it would leave at least one cell uncovered. In the figure, $BC'D$ and $AC$ are essential prime implicants, because they cover cells not covered by any other groups. However, $ABD$ is not essential. If it were removed, its cells would still be covered by $BC'D$ and $AC$.

There is no such thing as an optimus prime implicant.

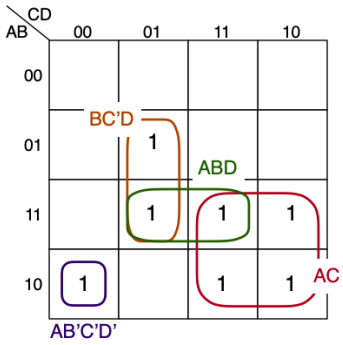



**Figure 3.5.1** Illustrating "minterm", "implicant", "prime implicant", and "essential prime implicant".

A minimized logic equation is a sum of essential prime implicants. In Figure 3.5.1, p. 58, the minimized logic equation is $Y = AB'D' + BC'D + AC$.

**Algorithmic minimization.** The process of logic minimization is to find the smallest number of essential prime implicants that will cover the 1s on the Karnaugh map. With practice, it isn't hard to use a Karnaugh map to identify essential prime implicants for functions of four or fewer variables. For five and six variables, the Karnaugh map goes three-dimensional, becoming a stack of two or four planar four-variable maps with groups that extend vertically as well as horizontally. Beyond that, the Karnaugh map is in hyperspace and no longer visually tractable, though it can be represented by a data structure in a computer program.

The systematic nature of logic minimization invites algorithmic solutions that search for all the prime implicants and from that set identify the essential prime implicants. It has been found, however, that tabulating all possible implicants and then searching for the prime ones rapidly becomes computationally unwieldy.

A better algorithm, called **Espresso**, is described in the textbook by Katz [1] and in lecture notes by Nowick. [2] Some version of Espresso is used in EDA tools to simplify a logic equations.

Espresso starts with one cell in the (hyperspatial) Karnaugh map and grows a prime implicant by expanding into neighboring cells. Once a prime implicant has been identified the expansion process begins again with a new, uncovered cell. In this way a set of overlapping prime implicants is assembled. It may not be an optimal set, however. Espresso searches for a better covering by shrinking (reducing) some of the prime implicants so that none of them overlap, and expanding them again in different directions to create a different overlapping covering of the map. If this new covering is better (fewer, larger prime implicants) than the first one, it is retained and the reduce-expand step repeats from there (though in such as way as to not repeat old solutions). If not, it goes back and tries again. The quality and efficiency of results depend on the algorithm's choices of where to start and how to expand and reduce, and Espresso includes various heuristics for generating good results in reasonable time.

[1] R.H. Katz, Contemporary Logic Design. Benjamin/Cummings (1994), pp 85-91.
[2] Columbia University CSEE 6861 (2016), accessed June 3, 2020.

# Chapter 4

# Combinational Logic

A truth table enumerates all the possible combinations of a system's input bits (e.g., "0000", "0001", ... "1111"), and the corresponding outputs. For this reason, the logic described by a truth table is also called **combinational**. Truth tables, logic equations, and logic gates are fundamental ways to capture the behavior of a digital system, but they are inefficient ways to capture the behavior of a complex digital system. In hardware, as in software, modularity and hierarchy are essential to good design. This chapter discusses several combinational logic functions that are commonly used building blocks for constructing digital systems.

## 4.1 Multiplexers

In an earlier exercise, Checkpoint 1.3.5, p. 10, you were introduced to the two-input multiplexer, which has two one-bit data inputs, $(A, B)$, a control bit $S$ ("select"), and a one-bit output, $y$. The control bit $S$ selects whether $y = A$ or $y = B$. The function of the multiplexer is described by the following if-then-else:

```
if S=='0'
    then y=A;
    else y=B;
end if;
```

which can be expressed as a logic equation,

$$y = S'A + SB.$$

There are numerous applications of multiplexers in hardware design, as there are for "if-then-else" statements in software. So that the gate-level sum-of-products circuit doesn't have to be drawn every time a designer wants to use a multiplexer, the component has its own logic symbol Figure 4.1.1, p. 62.

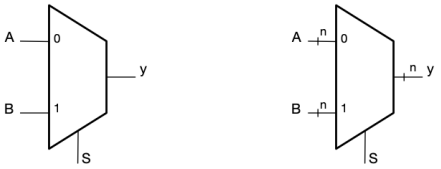



**Figure 4.1.1** Two-input (or "two-to-one") multiplexer symbol. (Left) Selecting one of two single-bit signals; (Right) Selecting one of two $n$-bit bundles.

The same symbol may be used to indicate that the choice is between single-bit or multi-bit signals, just by labeling the input and output lines with the number of bits.

A multiplexer with $n$-bit inputs is constructed from $n$ simple multiplexers as shown in Figure 4.1.2, p. 62.

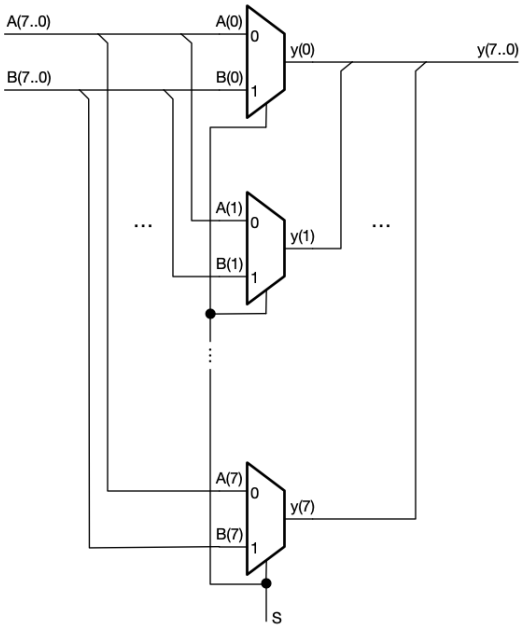



**Figure 4.1.2** An 8-bit multiplexer is constructed from 8 simple multiplexers, with a common select bit. The inputs and output must have the same number of bits. The notation `A(7..0)` indicates an 8-bit bundle, with individual bits denoted `A(0)` through `A(7)`. Each multiplexer selects between a pair of single inputs. A perpendicular junction between two wires, as on the select line, indicates a connection, as usual. An angled junction indicates a **bus tap**, where one or more wires branch out of a bundle, as shown on the multiplexer inputs, or merge into a bundle, as on the output.

The multi-bit multiplexer notation, Figure 4.1.1, p. 62, packs a lot of logic into a compact symbol!

Any time you have the need to select between two alternatives, think "multiplexer". For example, suppose a temperature sensor outputs an 8-bit number. One such sensor is placed outside the window in your room, and the other sensor is on your desk. An 8-wire bundle runs from each sensor to a multiplexer, and you can select which temperature to read with a signal `TemperatureSelect` applied to the select bit, Figure 4.1.3, p. 63.

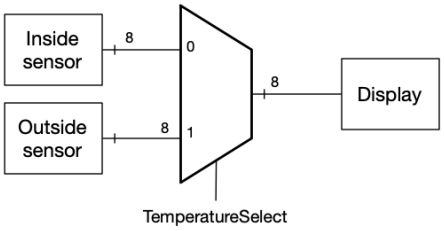



**Figure 4.1.3** Using an 8-bit, two-input multiplexer to select which of two temperature sensors to display.

**Checkpoint 4.1.4** Here is a selection process that frequently occurs in practice. The system has two control bits, `G` and `S`, two data inputs, `A` and `B`, and one data output, `y`. When `G='0'`, the output is 0, regardless of the values of the inputs. When `G='1'`, typical multiplexer behavior results, with `A` or `B` selected, depending on the value of `S`. Show, via a logic diagram, how to implement this with two two-input multiplexers.

**Multi-input multiplexers.** Multiplexers can also be devised that select among more than two input signals. A four-to-one multiplexer, Figure 4.1.5, p. 63, selects one of four inputs. The inputs may be one or $n$ bits each. To choose among four inputs requires two select bits, which together encode the numbers 0, 1, 2, 3 (binary "00", "01", "10", "11").

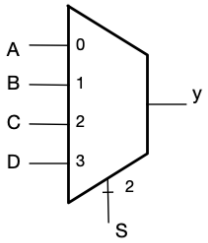



**Figure 4.1.5** A multiplexer with four inputs requires two select bits. The select bits may be drawn as a two-bit bundle, as shown here, or as two individual wires, whichever is more convenient for the designer.

A multi-input multiplexer functions like a "case" or "switch" statement in software:

```
case S is
    when "00" => y <= A;
    when "01" => y <= B;
    when "10" => y <= C;
    when "11" => y <= D;
end case;
```

**Checkpoint 4.1.6** Write a truth table for a four-to-one multiplexer, with four one-bit inputs $A$ through $D$, output $y$, and two select bits $S_0$ and $S_1$. The truth table can be simply written with three columns, for the two select bits and the output. Also write a logic equation, $y = f(A, B, C, D)$.

**Answer**.

| $S_1$ | $S_0$ | $y$ |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

The logic equation is $y = S_1'S_0'A + S_1'S_0B + S_1S_0'C + S_1S_0D$.

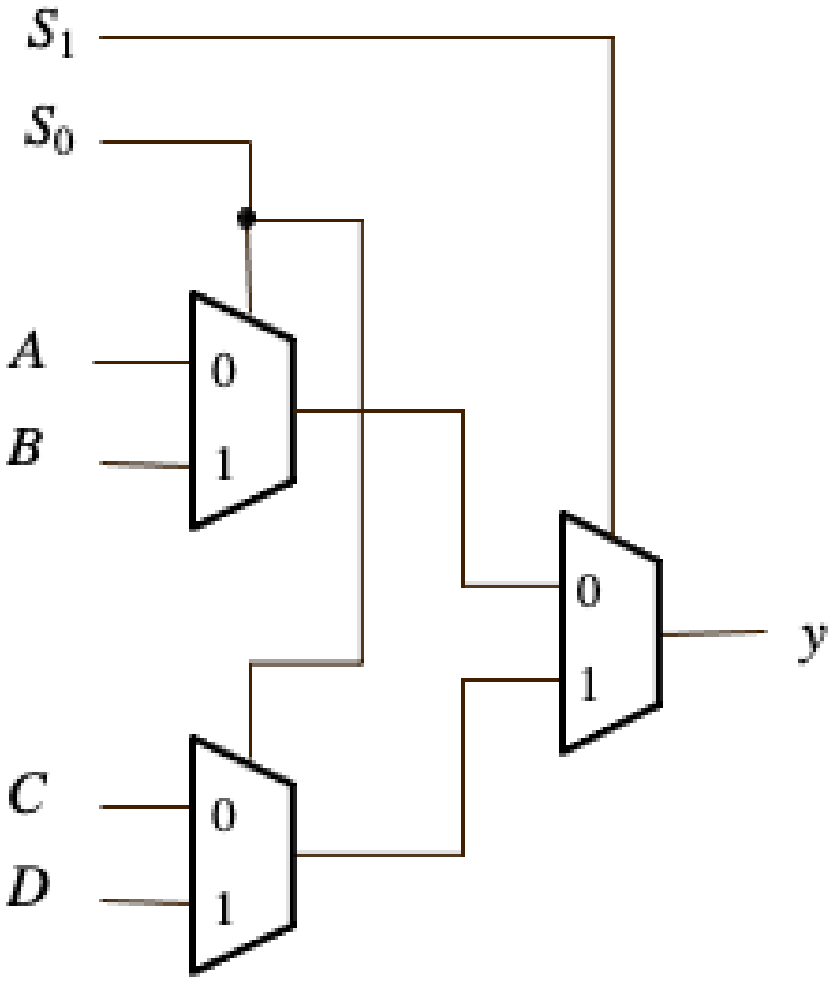
The logic equation may be factored into a combination of three two-to-one multiplexers, like this:

$$y = S_1'(S_0'A + S_0B) + S_1(S_0'C + S_0D).$$

This form is of practical interest because of a special two-input multiplexer circuit that requires only six CMOS transistors, making an implementation based on this form quite small and fast. Learn about it in this video:

**Checkpoint 4.1.7** Draw a logic diagram for the four-input multiplexer using three two-input multiplexers.

**Answer**.




**Checkpoint 4.1.8** Calculate and compare the number of transistors required for (a) the "case-like" four-input multiplexer made with NAND and NOT gates, (b) the four-input multiplexer constructed from two-input multiplexers with NAND and NOT gates, and (3) the four-input multiplexer constructed from six-transistor two-input multiplexers.

**Answer**. (a) 2 NOT, 4 3-input NAND, and 1 4-input NAND, requiring 36 transistors. (b) 2 NOT, 9 2-input NAND, requiring 40 transistors. (c) 3 2-input multiplexers, requiring 18 transistors (perhaps 16?).

A multiplexer that selects one of four multi-bit bundles is made by connecting single-bit multiplexers together, similarly to Figure 4.1.2, p. 62. Even larger (e.g., 8-to-1, 16-to-1) multiplexers (with three and four select bits, respectively), may also be compactly constructed from two-input CMOS multiplexers. The circuits have more than two layers of logic, and consequently are slower direct than sum-of-products circuits, but they are much smaller and use less power.

## 4.2 One-hot decoders

A multiplexer selects one of two or more inputs, according to the value of one or more select bits. The one-hot decoder, introduced previously in Figure 1.3.2, p. 8 and Checkpoint 1.4.3, p. 13, goes the other way: it activates one of its outputs, according to the value of its select bits, Figure 4.2.1, p. 66. The

additional enable input, G, must be '1' in order for any output to be activated. If it is '0', then all outputs are off. Although, as shown in the figure, the decoder can be thought of as a demultiplexer, potentially routing a multi-bit data input to one of several multi-bit outputs, the more common application is to use the decoder with only single-bit outputs, to provide control signals to other components.

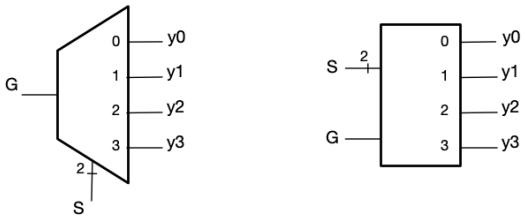



**Figure 4.2.1** A two-to-four one-hot decoder, drawn two ways. (Left) As a "demultiplexer", with select inputs S and data input G. (Right) As a decoder block, with with select inputs S enable input G. The notation on the right is preferred in this class. The select bits may be drawn as a two-bit bundle, as shown here, or as two individual wires, whichever is more convenient for the designer.

**Checkpoint 4.2.2** Revisit Figure 1.3.2, p. 8 and Checkpoint 1.4.3, p. 13, adding the enable input $G$. Also draw a logic diagram, using NAND and NOT gates.

**Answer**. The truth table, with enable input added, is:

| $G$ | $n_1$ | $n_0$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

And the logic equations are:

$$y_0 = G \cdot \overline{n_1} \cdot \overline{n_0}$$
$$y_1 = G \cdot \overline{n_1} \cdot n_0$$
$$y_2 = G \cdot n_1 \cdot \overline{n_0}$$
$$y_3 = G \cdot n_1 \cdot n_0$$

(Spoiler alert) Please attempt the preceding exercise before watching this video.

**Checkpoint 4.2.3** Draw a logic diagram using a one-hot decoder block (not gate-level logic) to activate one of four LEDs — RED, YELLOW, GREEN, BLUE —, depending on a binary input — "00", "01", "10", "11", respectively. An additional control signal disables all four LEDs.

## 4.3 Seven-segment display decoder

Not all decoders are of the one-hot variety. Here is an example of a more complex decoder.

A popular way to display numerals uses seven LED-illuminated segments, arranged as shown in Figure 16.1.1, p. 165. The segments are labeled $a$ through $g$. The display can easily show the decimal numerals 0 through 9, and with a little imagination, the additional hexadecimal characters A through F (as A, b, c, d, E, F). A display may also include a decimal point as an eighth "segment".


**Figure 4.3.1** A single-digit seven-segment display can show 0-9 and A-F (as A, b, c, d, E, F).

The signals to drive the segments are produced by a special 4-to-7 decoder that follows the truth table shown in Figure 4.3.2, p. 67.

| x | $(x_3\ x_2\ x_1\ x_0)$ | $(a\ b\ c\ d\ e\ f\ g)$ | x | $(x_3\ x_2\ x_1\ x_0)$ | $(a\ b\ c\ d\ e\ f\ g)$ |
|---|---|---|---|---|---|
| 0 | 0 0 0 0 | 1 1 1 1 1 1 0 | 8 | 1 0 0 0 | 1 1 1 1 1 1 1 |
| 1 | 0 0 0 1 | 0 1 1 0 0 0 0 | 9 | 1 0 0 1 | 1 1 1 0 0 1 1 |
| 2 | 0 0 1 0 | 1 1 0 1 1 0 1 | 10($A$) | 1 0 1 0 | 1 1 1 0 1 1 1 |
| 3 | 0 0 1 1 | 1 1 1 1 0 0 1 | 11($b$) | 1 0 1 1 | 0 0 1 1 1 1 1 |
| 4 | 0 1 0 0 | 0 1 1 0 0 1 1 | 12($C$) | 1 1 0 0 | 1 0 0 1 1 1 0 |
| 5 | 0 1 0 1 | 1 0 1 1 0 1 1 | 13($d$) | 1 1 0 1 | 0 1 1 1 1 0 1 |
| 6 | 0 1 1 0 | 0 0 1 1 1 1 1 | 14($E$) | 1 1 1 0 | 1 0 0 1 1 1 1 |
| 7 | 0 1 1 1 | 1 1 1 0 0 0 0 | 15($F$) | 1 1 1 1 | 1 0 0 0 1 1 1 |

**Figure 4.3.2** Truth table for hexadecimal to seven-segment decoder. A TRUE (1) output, $a - g$, activates the respective segment.

Read more about seven-segment decoders in Chapter 16, p. 165.

**Checkpoint 4.3.3** With four inputs and seven outputs, writing logic equations for the seven-segment decoder is a formidable task (seven four-variable Karnaugh maps!). However, just to keep your skills sharp, pick a couple of outputs and derive reduced logic equations in minimum sum-of-products form.

**Checkpoint 4.3.4** In the special case of a binary coded decimal (BCD) to seven-segment decoder, rows 10-15 of the truth table are assumed never to be used, and we don't care what the outputs are in these cases. On the Karnaugh map, the cells corresponding to these rows can be marked as '1' if that enables you to create larger groups. Try this in your solution to the previous exercise and see if it results in simpler logic.

## 4.4 Encoders

A one-hot decoder, Figure 4.2.1, p. 66, activates one of its outputs based on the input binary code. An **encoder** outputs a binary code to indicate which of several inputs has been asserted. You might, for example, have four pushbutton switches labelled 0 through 3, and you would like to display the binary code "10" when button 2 is presseed.

The potential problem, of course, is that more than one button could be pressed at the same time. Or, if you had four peripheral devices, each with data to communicate to you, more than one could be vying for your attention. The solution is for the encoder to prioritize its inputs, e.g., lower numbered inputs have higher priority: if inputs 1 and 3 are asserted at the same time, the output is "01". This device is called a **priority encoder**.

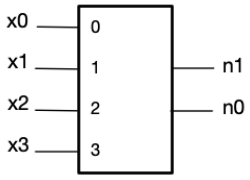



**Figure 4.4.1** An encoder assigns a binary output code indicating which of its inputs is asserted. Input `x0` may be omitted so that output "00" indicates no inputs are asserted.

**Checkpoint 4.4.2**

1. Write the truth table for a four-input, two-output, priority encoder. Input 0 has highest priority.

2. A potential problem is what to output when no inputs are asserted. One solution would be to have only three inputs, 1 through 3, and assign the output code "00" when none of the three inputs is asserted. Modify your truth table accordingly.

3. Write logic equations for the encoder

**Checkpoint 4.4.3** Repeat the previous exercise, truth table and logic equations, with priority reversed, so that input 3 has highest priority. Instead of sacrificing one input so that "00" can indicate that nothing has been asserted, add an addtional output bit that is '1' if at least one of four inputs is asserted, and '0' if no inputs are asserted.

(Spoiler alert) Please attempt the preceding exercises before watching this video.

## 4.5 Lookup tables

Any combinational logic function is completely described by a truth table. From there, we have seen how logic equations may be derived, minimized, and finally implemented as gate-level circuits. When integrated circuits contained only small numbers of gates (see Section 1.6, p. 15), even a system of modest complexity could require a large number of packages and a correspondingly large circuit board. In response, designers searched for other ways to implement logic at higher levels of integration.

One clever discovery was that a four-to-one multiplexer could be used to implement any function of two variables. Recall that the logic equation for a four-to-one multiplexer is:

$$y = S_1' S_0' A + S_1' S_0 B + S_1 S_0' C + S_1 S_0 D$$

Each term contains one of the four possible combinations of the select bits $S_1$ and $S_0$. The remaining inputs $A$ through $D$ are used to select which combinations will yield a TRUE output. For example, to implement $y = S_1' S_0 + S_1 S_0'$, you would wire inputs $A$ and $D$ to '0', and inputs $B$ and $C$ to '1'. An eight-to-one multiplexer could be wired to implement any function of three variables, and a sixteen-to-one multiplexer could be wired to implement any function of four variables. If you are counting transistors, this may not compare favorably with a minimum sum-of-products constructed from individual gates. But, if the individual gates require several packages on a circuit board, and the equivalent multiplexer circuit can be built with only one package, you save on area and maybe power and speed as well.

This is an old idea, that, surprisingly, lives on in the programmable logic devices used in this course (see Section 11.1, p. 123). Figure 4.5.1, p. 70 shows an eight-to-one multiplexer, with each input connected to a small block of logic that can be programmed to store either '1' or '0' (we'll learn about these in a future chapter). The combination of storage and multiplexer is called a **lookup table**, or **LUT**. (This isn't exactly how they are made, but it is a very convenient model.) Depending on the programming, the LUT can implement any function of three variables. Contemporary programmable devices contain thousands of small LUTs, each with as many as six inputs. Rather than changing the gates and wiring of a gate-level circuit to modify a function, one may simply change the programming of its LUT.

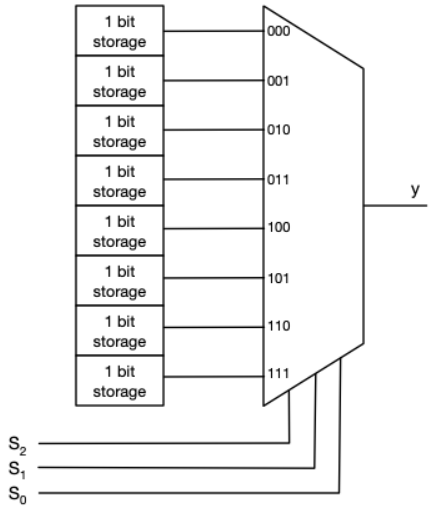



**Figure 4.5.1** Model of a three-input lookup table (LUT) as eight bits of storage and an 8-to-1 multiplexer. The LUT can store the truth table for any function of three logic variables, $Y = f(S_2, S_1, S_0)$.

**Checkpoint 4.5.2** What logic function is implemented by the LUT in Figure 4.5.1, p. 70, if the stored bits are (read from top to bottom), 1, 1, 0, 0, 0, 1, 0, 1?

**Answer**. $y = S_2'S_1'S_0' + S_2'S_1'S_0 + S_2S_1'S_0 + S_2S_1S_0 = S_2'S_1' + S_2S_0$

# Part II

# VHDL-based design

# Chapter 5

# Top level modeling with VHDL

## 5.1 Preview of (more) modern logic design

Recall, from Section 1.6, p. 15, the four steps in prototyping a digital system:

1. Capture: Expressing the desired system behavior, e.g., in a truth table and logic equations.
2. Synthesis: Translating the captured design to a logic diagram and/or a list of required components and a netlist.
3. Implementation: Placing components in a physical layout and routing the connections specified in the netlist.
4. Testing: Verifying that the prototype functions as intended.

We are now going to up the sophistication of our design approach with **computer-aided design (CAD)**, also called **electronic design automation (EDA)**. EDA tools take advantage of computing power to enable designers to do three things better:

- We can capture more sophisticated designs at a higher level of abstraction than truth tables and logic equations.
- We can test a design before implementing it, through logic simulation.
- We can synthesize and implement larger designs with programmable hardware, rather than hand wiring a breadboard with discrete components.

Early EDA tools enabled designers to perform schematic capture, drawing their logic diagrams on a computer screen. The tools would then translate the schematic into an internal representation that could be analyzed for consistency (e.g., making sure that two outputs aren't connected together), simulated (is the design logically correct, does it meet timing requirements), and then translated into a printed circuit layout for fabrication. In the early 1980s, design capture through **hardware description language (HDL)** was introduced. A hardware description language looks like an ordinary programming language, but it contains features designed to model hardware that operates in real time. Two of these early HDLs, **VHDL** and **Verilog**, have continued, with periodic updates, to the present day, and are the most commonly used in universities and industry. More sophisticated HDLs, including so-called **high level synthesis (HLS)** methods that convert C code into hardware prototypes, are evolving rapidly and may supplant VHDL and Verilog in the future.

As with most arguments about programming languages (C? C++? Java? Python? ...), VHDL and Verilog each have their passionate fans and detractors. Each language has advantages and disadvantages relative to the other, and industry sectors that prefer one over the other. My opinion is that VHDL is easier to learn than Verilog (though of course there are people who believe the opposite). And, because I'm the boss in this course, we're going to use VHDL.

## 5.2 Input and output ports (entity block)

VHDL was designed for the purpose of modeling hardware. Going back to the beginning, recall that a digital system takes in control and data bits through its input ports and and produces control and data bits through its output ports Figure 5.2.1, p. 74.

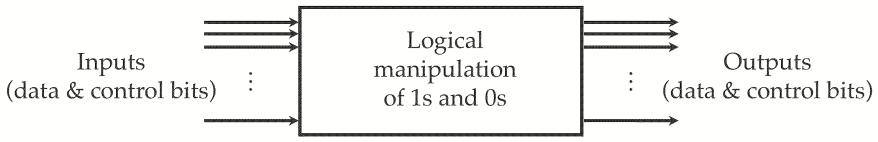



**Figure 5.2.1** A digital processor takes in data and control bits through its input ports and produces data and control bits through its output ports.

In VHDL, the external description of a system, showing its input and output ports, is called an **entity**. A port may represent a single bit, or a bundle of bits (bit vector, or bus). Consider, for example, the four-bit up-down counter depicted in Figure 5.2.2, p. 74. There are two single-bit input ports, CLK and UP/DOWN, a single-digit seven-segment output bus, and a single-bit output port for the terminal count signal (TC).

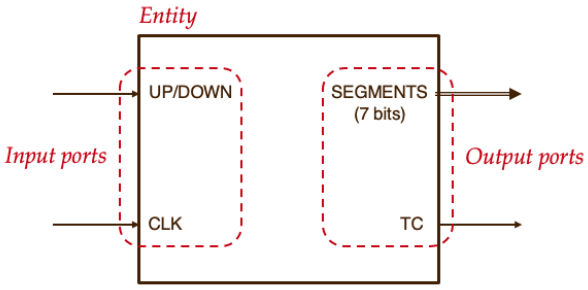



**Figure 5.2.2** Input and output ports of a four-bit up-down counter.

A VHDL model for the counter begins with the entity block in Listing 5.2.3, p. 75. Each port is declared with a name, a direction (in or out), a data type (always "std_logic" for single bits or "std_logic_vector" for bundles of bits), and an optional (but always advisable) comment. The notation may seem cumbersome at first, but so did your first programming language, and in time, you mastered its rules. The good news is that all entity blocks are structured like this. There are almost no additional features to learn.

```
entity counter4_updown is
        port(clk:       in  std_logic;                --
            clock
             updown:    in  std_logic;                -- 0
                for up, 1 for down
             segments:  out std_logic_vector(0 to 6); --
                segments a-g
             TC:        out std_logic );              --
                terminal count
end counter4_updown;
```

**Listing 5.2.3** Entity block for four-bit up-down counter, Figure 5.2.2, p. 74.

The declaration `std_logic_vector(0 to 6)` for `segments` merits an additional comment. This is a vector of seven bits, indexed from left to right. `segments(0)` corresponds to segment `a`, `segments(1)` is segment `b`, up to `segments(6)`, which is segment `g`. This is one way to index a bit vector in VHDL. The other way, used when the vector must be indexed with its most significant bit at the left end, is, e.g., `std_logic_vector(7 downto 0)` for an eight-bit byte. The `downto` form is always used for numbers.

**Checkpoint 5.2.4** Based on your prior experience with other programming languages, what other observations do you have about the entity block in Listing 5.2.3, p. 75? What questions do you have?

**Checkpoint 5.2.5** Write an entity block for the odd number detector, Checkpoint 1.3.3, p. 8. Write the entity so that the input has eight bits rather than two.

**Answer**.

```
entity odd_number is
   port(x:   in  std_logic_vector(7 downto 0); -- 8-bit
       input
        y:   out std_logic );                -- 0 if even,
            1 if odd
end odd_number;
```

**Checkpoint 5.2.6** Write an entity block for the "$x > y$" function. Write the entity so that the inputs have four bits rather than two.

**Answer**.

```
entity x_gt_y is
   port(x:   in  std_logic_vector(3 downto 0); -- 4-bit
       input
        y:   in  std_logic_vector(3 downto 0); -- 4-bit
            input
        g:   out std_logic );                  -- 1 if x >
            y, 0 otherwise
end x_gt_y;
```

## Exercises

1. **VHDL models.** For additional practice, write entity blocks for the earlier example systems:

   a Multiplexer, Checkpoint 1.3.5, p. 10.

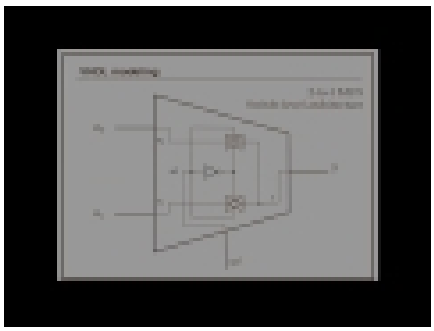
   b One-hot decoder, Figure 1.3.2, p. 8.

   c Priority encoder, Checkpoint 1.3.6, p. 10.

## 5.3 Internal logic (architecture block)

Every VHDL model has an entity block that defines its interface to the outside world. This is analogous to a function prototype in a traditional programming language like C or Java. The entity declares the inputs and outputs, but says nothing about how the system actually processes the inputs to produce the outputs. The internal functionality of the system is described by the system's **architecture block**. Because the architecture is where the actual work gets done, it is more complex than the entity block. However, we will see that VHDL is a highly structured and stylized language, almost template-like. It will not be difficult to write architectures for simple systems, and the architectures for more complex systems follow the same patterns.

Before moving ahead, you may want to watch this video (up to time 5:18), which reviews the entity block and gives a preview of the architecture block.



As a working example, let's consider the two-input multiplexer, Checkpoint 1.4.4, p. 13. It has two data inputs, $A$ and $B$, select bit $S$, and output $y$. The logic equation is $y = \overline{S} \cdot A + S \cdot B$. The complete VHDL model for the multiplexer, based on the logic equation, is shown in Listing 5.3.1, p. 76, below.

```
-- Two-input multiplexer model
   library IEEE;
   use IEEE.std_logic_1164.all;

   entity mux2to1 is
       port( A, B:  in  std_logic;     -- inputs
             S:     in  std_logic;     -- select bit
             y:     out std_logic );   -- output
   end mux2to1;

   architecture gatelevel of mux2to1 is
   begin
       y <= (not(S) and A) or (S and B);
   end gatelevel;
```

**Listing 5.3.1** VHDL model for a two-input multiplexer, based on the logic equation $y = \overline{S} \cdot A + S \cdot B$.

**Checkpoint 5.3.2** Based on your prior experience with other programming languages, what observations do you have about the architecture block in Listing 5.3.1, p. 76? What questions do you have?

The listing includes a header line describing the model. Multiple header lines (they are just comments) are helpful for the reader, to provide information about the project, engineer, date created, revisions, etc. The next two lines, before the entity block, must appear in every VHDL model you write. They are library statements that, among other things, make the std_logic and std_logic_vector

datatypes available for use. Next is the entity block; note here that the data inputs, $A$ and $B$, are declared in the same line. Sometimes this is convenient, if the the ports are closely related in function.

The architecture block proper is set off by the line `architecture gatelevel of mux2to1 is`. You can think of it as saying "this is a gate-level (logic equation) model for the mux2to1 entity". The label `gatelevel` is purely descriptive, and can be any word you find useful, though only certain labels are commonly used in practice. The statements comprising the architecture are bracketed by `begin ... end`.

The logic equation itself, `y <= (not(S) and A) or (S and B);` looks much as it would in any computer language, with the particular note that the VHDL assignment operator is `<=`. (The plain `=` operator is reserved for testing equality, like `==` in C.) The assignment operator is usually read "gets", e.g., `y <= (not(S) and A)` is read "y gets not S and A".

Now, to make a point, we'll write the architecture again, making a literal translation of the logic diagram into multiple logic equations. This is not an efficient way to write the model for the multiplexer, but it provides a simple introduction to an important feature of VHDL models. The gate-level logic diagram is shown in Figure 5.3.3, p. 77. The nets between the AND gate outputs and OR gate inputs are named asn and bs, denoting "a and not s" and "b and s", respectively.

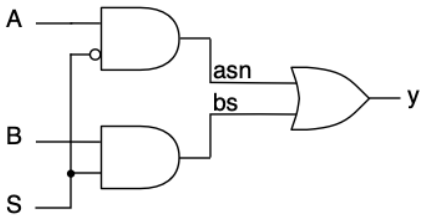



**Figure 5.3.3** Logic diagram for a two-input multiplexer. Intermediate signals are labeled asn and bs.

A VHDL model using these intermediate signals is shown in Listing 5.3.4, p. 78. The key feature is the way the intermediate nets are declared and used:

- They are declared with the keyword `signal`, in the location shown, within the architecture but before the `begin...end` containing the logic itself.
- Signals are local to the architecture and unknown to the entity and the outside world.
- Like ports, they are declared with a datatype (here, `std_logic`).
- Signals and ports may be mixed in a logic equation, as long as they are of the same type (which, for now, is easy).

```
-- Two-input multiplexer model, gate-by-gate
   library IEEE;
   use IEEE.std_logic_1164.all;

   entity mux2to1 is
       port( A, B:  in  std_logic;     -- inputs
             S:     in  std_logic;     -- select bit
             y:     out std_logic );   -- output
   end mux2to1;

   architecture gatelevel of mux2to1 is
       signal asn, bs: std_logic;      -- intermediate nets
   begin
       asn <= not(S) and A;
       bs <= S and B;
       y <= asn or bs;
   end gatelevel;
```

**Listing 5.3.4** VHDL model for a two-input multiplexer, gate-by-gate.

As with the first listing, the equations themselves look the way they would if they were written in a conventional computing language. The three statements would be executed in order, first computing the two intermediate values and then combining them to make the result. In VHDL, however, we are modeling hardware, and in an electronic circuit all the components are operating, simultaneously, all the time. VHDL captures this property of hardware with the idea of **concurrency**.

Unlike a conventional programming language, in VHDL these two arrangements of the equations are equivalent:

```
asn <= not(S) and A;
bs <= S and B;

y <= asn or bs;
```

```
y <= asn or bs;

asn <= not(S) and A;
bs <= S and B;
```

They both express the concurrent logic of the circuit diagram: two AND gates combining the inputs, and an OR gate combining the outputs of the AND gates. Concurrency is essential to the way that VHDL is able to model hardware. In the next chapter we will begin to see how it works.

**Example 5.3.5** Here is an example of using std_logic_vectors to simplify the entity and architecture. Recall the logic equations for the one-hot decoder, Checkpoint 1.4.3, p. 13,

$$y_0 = \overline{n_1} \cdot \overline{n_0}$$
$$y_1 = \overline{n_1} \cdot n_0$$
$$y_2 = n_1 \cdot \overline{n_0}$$
$$y_3 = n_1 \cdot n_0$$

We could write the entity with two one-bit inputs and four one-bit outputs, but it will be more compact to use a two-bit std_logic_vector input and a four-bit std_logic_vector output. Try it yourself, then check the answer.

**Hint**. Individual elements of a vector are referenced by their indices, e.g., `y(0)` is the least significant bit of the vector `y` declared as `std_logic_vector(3 downto 0)`.

**Answer**. The VHDL model for this one-hot decoder is

```
-- Two-input one-hot decoder model, gate level
   library IEEE;
   use IEEE.std_logic_1164.all;

   entity decoder_onehot2 is
       port( n:  in  std_logic_vector(1 downto 0);     --
           input
             y:  out std_logic_vector(3 downto 0) );   --
                 output
   end decoder_onehot;

   architecture gatelevel of decoder_onehot is
   begin
       y(0) <= not(n(1)) and not(n(0));
       y(1) <= not(n(1)) and n(0);
       y(2) <= n(1) and not(n(0));
       y(3) <= n(1) and n(0);
   end gatelevel;
```

□

## Exercises

1. **VHDL models.** Complete the VHDL models for these earlier examples. Use std_logic_vectors when appropriate.

   a Two-bit odd number circuit, Checkpoint 1.4.2, p. 13.

   b Two-bit $x > y$ circuit, Checkpoint 1.3.4, p. 8.

   c Priority encoder, Checkpoint 1.3.6, p. 10.

# Chapter 6

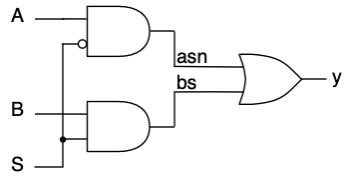
# VHDL processes

## 6.1 Events

The architecture for the simple two-input multiplexer model, Listing 5.3.1, p. 76, consists of a single statement, `y <= (not(S) and A) or (S and B);`. We begin this chapter by unpacking how this statement models the multiplexer hardware. The multiplexer has three input ports, $A, B, S$, and one output port, $y$. While the inputs are steady, 1 or 0, the output is steady, 1 or 0. The output only changes in response to a change on one of the inputs. Such a change, from 0 to 1 or from 1 to 0, is called an **event**. An event causes transistors in the circuit to switch on or off in response to the new input values, which result in a new output value in accordance with the logic of the circuit. Between events the circuit is in steady state.

One of the uses of a VHDL model is to test the logical function of a design. A software tool called a **simulator** creates a sequence of input events and computes outputs according to the statements in the VHDL model. The results may then be examined to see if they agree with the expected behavior of the system. Just as in a real circuit, a statement in a VHDL model, e.g., `y <= (not(S) and A) or (S and B);`, is normally inactive, waiting for an event on one or more of the inputs $A, B$ and $S$. In response to an event, the simulator executes the statement and assigns a new value of $y$ appropriately. After this, execution is **suspended** until the next event.

This starts to get interesting when more than one statement is involved. Consider the other multiplexer model, Listing 5.3.4, p. 78, which has three statements:



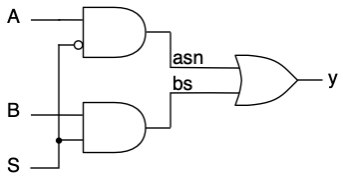

```
asn <= not(S) and A;
       -- 1
bs <= S and B;
              -- 2
y <= asn or bs;
             -- 3
```

Suppose, in response to earlier events, $A = 0, B = 1, S = 0$, so $y = A = 0$. The circuit is in steady state and all three statements in the model are suspended. Now, let $A$ change from 0 to 1. This event activates the upper AND gate, and its output changes from 0 to 1. This makes an event on the input of the OR gate, and it activates, changing its output from 0 to 1. The lower AND gate is unaffected and remains inactive.

An analogous chain of events occurs in the execution of the VHDL model.

The event on port A activates Statement 1, and it is evaluated. The value of the signal asn is changed, then the statement suspends, pending a new event. The change in asn is an event which activates Statement 3. The statement is evaluated and a new value is assigned to the port y. Then the statement suspends pending a new event. There are no events on ports S or B, so Statement 2 remains suspended. If a new event occurs, on B, say, execution of the model will resume with activation and evaluation of Statement 2, which causes an event on signal bs, activating Statement 3. Statement 1 will remain inactive, because it does not depend on B or bs. Note how the VHDL exactly mimics the operation of the circuit.

Suppose we reorder the statements, like this:




```
y <= asn or bs;
            -- 3
asn <= not(S) and A;
       -- 1
bs <= S and B;
             -- 2
```

If we set up the same initial conditions as before, $A = 0, B = 1, S = 0$, and $y = A = 0$, and repeat the same sequence of events, the statements will be executed identically: Statement 1, then 3, then 2, then 3. The execution of the statements in the model depends not on the order in which they are written, but on the sequence of events. For this reason, VHDL is said to be an **event-driven** language.

A circuit operates, processing its inputs and generating outputs, as long as power is applied. A VHDL simulator processes events, in an implicit "do forever" loop, until its execution is terminated.

> VHDL is capable of modeling propagation delays, as well, but this is a more advanced topic beyond the scope of this course. The interested reader is referred to Reference [1], pp. 141ff.

**Checkpoint 6.1.1** Work through the multiplexer model again, with an event on the select bit. What potential problem(s) do you see?

## 6.2 Processes

VHDL models based just on logic equations are straightforward, but very limited. We will now introduce a major feature of the language which enables more complex models to be written with greater ease.

A statement like `y <= a and b;` represents a series of steps in the simulation of the model:

1. Wait, suspended, for an event on a port or signal.
2. When an event occurs, evaluate the statement.
3. Assign the result and suspend, pending another event.

The statement is, in fact, a shorthand for the central VHDL construct, called a **process**, that explicitly includes these steps. Here is the basic process syntax:

```
process
begin
   y <= a and b;  -- 2 (evaluate)
   wait on a, b;     -- 3 (assign, suspend)
                     -- 1 (wait)
end process;
```

In simulation the process waits, inactive, pending an event on `a` or `b`. When an event occurs, the simulator activates the process and executes the statements, in order. The first one, `y <= a and b;`, is evaluated. The new value of `y` is noted, but not assigned. We say that the output is **scheduled**. When the simulator executes the second statement, `wait on a, b;`, the scheduled output is assigned, and the process suspends. It then waits for the next event on either input.

The same statement, `y <= a and b;`, means different things when it stands alone in an architecture and when it is included inside a process:

- As a standalone statement, `y <= a and b;` is shorthand for a process; Waiting, activating, scheduling, assigning, and suspending are all implicit.
- Inside a process, `y <= a and b;` merely schedules the assignment of a new value to `y`. Suspension, waiting, and activation are made explicit by the `wait a, b` statement.

A more common way to write a process, which we shall use from now on, is:

```
process(a, b)
begin
   y <=a and b;
end process;
```

The parenthesized list `(a, b)` is called the **sensitivity list** for the process. The process is said to be **sensitive** to `a` and `b`. The items in the sensitivity list look like function arguments in a conventional programming language, but they are not and should not be mistaken for such. Rather, the sensitivity list replaces the explicit `wait` statement in the previous model, naming the ports or signals whose events will activate the process.

Here is the three-statement multiplexer model, written with explicit processes. Like the logic gates they represent, they are concurrent. Each operates only in response to events on its inputs, interacting with other gates / processes only through interconnecting ports and signals.

```
-- Two-input multiplexer model, gate-by-gate, with processes
   library IEEE;
   use IEEE.std_logic_1164.all;

   entity mux2to1 is
       port( A, B:  in  std_logic;
             S:     in  std_logic;
             y:     out std_logic );
   end mux2to1;

   architecture gatelevel of mux2to1 is
       signal asn, bs: std_logic;      -- intermediate nets
   begin

   and_gate_1: process(A, S)           -- labels are
       optional, but helpful
      begin
         asn <= not(S) and A;
      end process and_gate_1;

   and_gate_2: process(B, S)
      begin
         bs <= S and B;
      end process and_gate_2;

   or_gate: process(asn, bs)
      begin
         y <= asn or bs;
      end process or_gate;

   end gatelevel;
```

**Listing 6.2.1** Gate-level model for the two-input multiplexer, with each gate represented by a process.

**Checkpoint 6.2.2** Walk through the model in Listing 6.2.1, p. 84, for events on `A`, `B`, and `S`, making sure you understand the seqence of events and activations leading to the output.

## 6.3 Sequential statements in processes

A VHDL model is a collection of concurrent processes. So far, we have used processses to represent individual logic gates. It must be stressed that this is only illustrative. One would rarely, in practice, use this apparatus to model something as simple as a logic gate. For the two-input multiplexer we have been considering, the single equation `y <= (not(s) and A) or (S and B);` is far more compact. However, as we proceed we'll gradually approach situations where we can see the superiority of concurrent process models.

First of all, processes may contain multiple statements. For example, the multiplexer architecture may be written

```
architecture gatelevel of mux2to1 is
          signal asn, bs: std_logic;      -- intermediate
              nets
      begin

      process(A, B, S, asn, bs)
         begin
            asn <= not(S) and A;   -- 1
            bs <= S and B;         -- 2
            y <= asn or bs;        -- 3
      end process;

      end gatelevel;
```

with the equations for all three gates in one process. The execution sequence for this process is a more complicated than we've seen, so we will take it a step at a time. Suppose $S = 0$ and the A port changes value. The following cascade of events and process activations ensues:

1. The event on A activates the process. The three statements are evaluated, in order, using the values that are in effect at the time of activation. In Statement 1, a new value for asn is scheduled, but not assigned. In Statement 2, the scheduled value of bs is unchanged, because neither S nor B changed. In Statement 3, the scheduled value of y is also unchanged, because the new value of asn has not yet been assigned, only scheduled. Statement 3 is still working with the old values.

2. When the end of the process is reached, the scheduled values are assigned; asn receives its new value. The process then suspends, pending a new event.

3. Because asn changed, there is an event, and the process activates. The three statements are evaluated, in order, using the current values. This time, y will receive a new scheduled value.

4. The scheduled values are assigned, and the process suspends, pending further events.

The result, after two activations of the process, is that y is updated with the new value of A. If you compare this with the logic diagram, Figure 5.3.3, p. 77, you will see that it matches the way one gate drives another.

**Checkpoint 6.3.1** Walk through the activations of the process following a change in S from 0 to 1. Compare with the flow of signals in time through the logic diagram.

In a conventional programming language, if you had two statements like this:

```
y = a;
y = b;
```

you would see y change twice. First, y would take the value of a, then it would take the value of b. In VHDL, if you wrote an architecture like this:

```
architecture gatelevel of foo is
begin
   y <= a;
   y <= b;
end gatelevel;
```

The port y would concurrently be assigned a and b. This is a "multiple driver error", like wiring two gate outputs to the same gate input. On the other hand, if these statements are placed inside a process,

```
architecture gatelevel of foo is
begin
   process(a,b)
   begin
      y <= a;   -- 1
      y <= b;   -- 2
    end process;
end gatelevel;
```

execution proceeds as follows, in response to an event on one of the inputs:

1. The process activates, and Statement 1 is evaluated. The value of y is scheduled to be the value of a, but as usual, it is not to be assigned until the process suspends.
2. Statement 2 is evaluated. The value of y is now scheduled to be the value of b, overriding the previously scheduled assignment.
3. y is assigned its new value, and the process suspends pending further events.

There is no multiple driver error. And unlike the conventional programming language, y is only assigned once. This feature permits more sophisticated logic to be placed inside processes.

### Exercises

Here is a good place to pause and do some practice problems...

**1.** (∗)

## 6.4 If-then-else statements

The various VHDL models we have seen for the multiplexer are all based on the logic equation or logic diagram. They are all correct, but none of them capture the essence of the multiplexer's behavior, which is really "if-then-else". VHDL provides this construction for processes. With it, the multiplexer architecture is:

```
architecture behavioral of mux2to1 is
begin

mux2: process(A, B, S)
begin
   if S='0' then
      y <= A;
   else
      y <= B;
   end if;
end process mux2;

end behavioral;
```

**Listing 6.4.1** Behavioral model for the two-input multiplexer, using if-then-else.

This model expresses the multiplexer's behavior more directly than the logic equation. Consequently, it is more readable, and in VHDL as in software engineering, readable code is much less prone to bugs. Furthermore, this construction is easily reused in any architecture where a multiplexer appears in the block diagram, without pausing to render the multiplexer into a logic equation.

Another benefit of the if-then-else version of the multiplexer model is that it is easily extended to vector inputs. Suppose you need to select one of two 8-bit bundles. After declaring the two data inputs and the output as `std_logic_vector(7 downto 0)`, the rest of the model is identical to the single-bit version! (Imagine doing this with logic equations, one for each bit.)

Frequently, we want to use nested if-then-else statements, for a hierarchical choice. For example, consider a multiplexer with an enable (EN) bit. When EN is false, the output is "0...0" regardless of what the other inputs are doing. But, if EN is true, then input A or B is selected as usual, based on the value of the select (S) bit. This would be modeled as so:

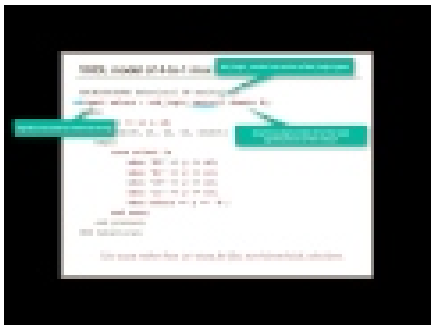
```
architecture behavioral of mux2to1_en is
begin

mux2en: process(A, B, S, EN)
begin
   if EN='0' then
      y <= "00000000";  -- assuming 8-bit vectors
   elsif S='0' then
      y <= A;
   else
      y <= B;
   end if;
end process mux2en;

end behavioral;
```

**Listing 6.4.2** Behavioral model for a two-input multiplexer, with enable.

## 6.5 Case statements

An if-then-else construction is used for two-alternative choices or, nested, for hierarchical choices, where the second decision depends on the outcome of the first. There are many uses for multi-alternative choices, which are "flat" rather than hierarchical. For these, an if-then-else can be inelegant and inefficient. A pure truth table is another example of a multi-alternative flat choice. VHDL provides the case construction for these situations.

Here is a video that summarizes the use of the case statement. Watch it (up to 9:18) before or after you read this section.



For a working example, consider the four-bit binary coded decimal (BCD) to seven-segment decoder. The truth table is shown below:

| x(3..0) | segs(0...6) |
|---|---|
| 0000 | 1111110 |
| 0001 | 0110000 |
| 0010 | 1101101 |
| 0011 | 1111001 |
| 0100 | 0110011 |
| 0101 | 1011011 |
| 0110 | 1011111 |
| 0111 | 1110000 |
| 1000 | 1111111 |
| 1001 | 1110011 |

**Figure 6.5.1** Truth table for a BCD to seven-segment decoder. Outputs are high-true.

This is easily translated into VHDL with the case statement:

```
-- x is std_logic(3 downto 0)
-- segs is std_logic(0 to 6)

sevenseg: process(x)
begin
   case x is
      when x"0" =>   segs <= "1111110";   -- abcdefg
      when x"1" =>   segs <= "0110000";
      when x"2" =>   segs <= "1101101";
      when x"3" =>   segs <= "0110000";
      when x"4" =>   segs <= "0110000";
      when x"5" =>   segs <= "0110000";
      when x"6" =>   segs <= "0110000";
      when x"7" =>   segs <= "0110000";
      when x"8" =>   segs <= "0110000";
      when x"9" =>   segs <= "1110011";
      when others => segs <= "1001001";   -- bar stack for
          invalid
   end case;
end process sevenseg;
```

**Listing 6.5.2** VHDL model for a BCD to seven-segment decoder, implementing the truth table in Figure 6.5.1, p. 88. The input values are written in hexadecimal, for readability. The last line, "when others", covers the invalid cases, x"a" through x"f", making a stack of three horizontal bars, ≣.

It is sometimes necessary to drive a seven-segment display with low-true signals, sinking current rather than sourcing. In this case, it is preferable to preserve the high-true nature of the table and simply invert `segs` before sending it to the output, e.g., with a statement like `segs_l <= not(segs);`.

**Checkpoint 6.5.3** Extend the truth table and VHDL model from BCD to hexadecimal, including the cases x"a" through x"f"

You now have what you need to model any combinational digital system with VHDL.

## Exercises

**1.** Use a case statement to write a model for a four-to-one multiplexer (two select bits, four inputs, one output).

**Answer**.

```
-- s is std_logic_vector(1 downto 0)

mux4to1: process(a, b, c, d, s)
begin
   case s is
      when x"0" =>   y <= a;
      when x"1" =>   y <= b;
      when x"2" =>   y <= c;
      when x"3" =>   y <= d;
      when others => null;        -- "null" means "do
          nothing"
   end case;
end process mux4to1;
```

**2.** Use a case statement to write a model for a two-to-four one-hot decoder. For added challenge, include an enable (EN) input.

**Hint**. It is OK to mix if-then-else and case statements

**Answer**.

```
-- x is std_logic_vector(1 downto 0)

dec2to4_en: process(x, en)
begin
    if en='0' then                        -- disabled,
        all off
       y <= "0000";
    else                                  -- enabled
       case x is
          when x"0" =>   y <= "0001";
          when x"1" =>   y <= "0010";
          when x"2" =>   y <= "0100";
          when x"3" =>   y <= "1000";
          when others => y <= "0000";     -- all off, if
              illegal input
       end case;
    end if;
end process dec2to4_en;
```

# Chapter 7

# Simulating VHDL models

## 7.1 Introduction

Capturing a design with a hardware description language enables the designer to then test the design, by simulation, before implementing it. It is easier, and less expensive, to debug a design in simulation rather than discovering design errors in hardware. Simulation does not remove the need for physical test, especially in this class, but it will get you closer to a correct design.

Prior to the advent of simulation, physical test looked like Figure 7.1.1, p. 91. A design, captured in a logic diagram, was prototyped on a hand-wired circuit board. Stimuli (inputs) were supplied by switches, function generators, or other digital subsystems. Of course, there would be bugs—logic errors or wiring errors. Debugging was carried out by probing signals of interest with an oscilloscope or a more powerful tool called a logic analyzer, and deducing causes from the observed waveforms. The circuit would be rewired to correct the error, and the process would repeat until all the bugs were fixed.

> "Oscilloscopes were used by cavement to debug fire." An engineer, quoted in Tracy Kidder, "The Soul of a New Machine.".

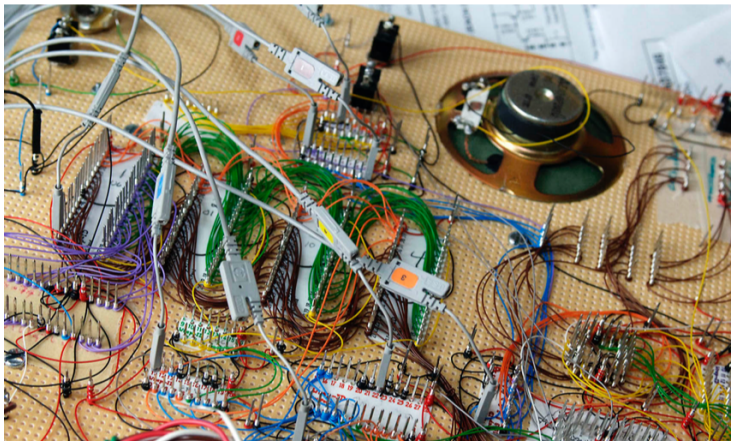


**Figure 7.1.1** Physical test of a hand-wired prototype. The gray probe wires connect to an oscilloscope.

Test by simulation is similar in spirit, as shown in Figure 7.1.2, p. 92, below. The prototype system, whether hardware or VHDL model, is called the **device**

**under test (DUT)** or **unit under test (UUT)**. Physical stimuli from a function generator or switches are replaced by an additional VHDL model called a **testbench**. The testbench generates an appropriate sequence of stimuli and applies them to the input ports of the UUT. The role of the oscilloscope or logic analyzer in displaying the outputs of the UUT is played by additional software that can display signals in the UUT model as waveforms on a screen. The simulator is usually able to watch not just the model's output ports, but also its internal signals. It is also possible to write a testbench in such a way that it not only generates stimuli but also compares the outputs with known correct responses to the stimuli. We shall use only waveform observation in this class.

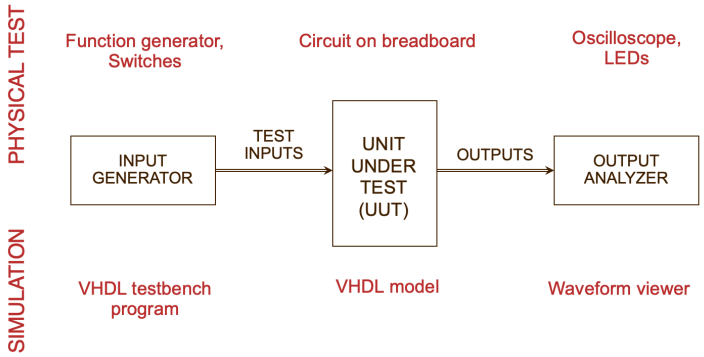



**Figure 7.1.2** Physical test compared with test by simulation.

## 7.2 VHDL testbench structure

Testbenches are generally quite simple, because they always have the same five parts, written in nearly the same way. To demonstrate, we will write a testbench for the two-input multiplexer model Listing 5.3.1, p. 76. The model has a control input, `S`, two data inputs, `A` and `B`, and an output, `y`. Every testbench has the same parts, as shown in the listing, below. The same testbench will apply to all of the two-input multiplexer models with the same ports, regardless of their architectures.

```
library ieee;
use ieee.std_logic_1164.all;

-- (1) EMPTY ENTITY DECLARATION
-- Testbenches never have I/O ports
entity mux2to1_tb is
end mux2to1_tb;

architecture testbench of mux2to1_tb is

-- (2) DECLARE THE UUT AS A COMPONENT
-- Must match the entity declaration in the UUT's model
      component mux2to1 is
         port( A, B:  in  std_logic;
               S:     in  std_logic;
               y:     out std_logic );
       end component;

-- (3) DECLARE SIGNALS TO CONNECT TO THE UUT
-- Customary to give them the same names as the UUT's ports
      signal a, b, s: std_logic := '0';            --
          inputs intialized to '0'
      signal y:       std_logic;

begin

-- (4) INSTANTIATE THE UUT IN THE TESTBENCH
-- Analogous to wiring a physical device in to the test
    equipment
      uut: mux2to1 port map
              a => a,                           -- "=>"
                  means "port maps to signal"
              b => b,                           -- separated
                  by commas, not semicolons
              s => s,
              y => y );

-- (5) GENERATE STIMULI
-- Systematically generate all combinations of input
stim_proc:  process                             -- No
    sensitivity list
      begin
           s <= '0';  a <= '0';  b <= '0';  -- May place
               all on one line, for compactness
           wait for 100ns;                        --
               Process suspends, resumes after 100ns

           s <= '0';  a <= '0';  b <= '1';  -- Each line is
               one row of truth table
           wait for 100ns;

           s <= '0';  a <= '1';  b <= '0';
           wait for 100ns;

           ...

           s <='1';  a <= '1';  b <= '1';
           wait;

           -- Process halts here
      end process stim_proc;

   end testbench;
```

**Listing 7.2.1** VHDL testbench for two-input multiplexer model, Listing 5.3.1, p. 76

Notes on the five testbench sections:

1. The entity may be named anything, but I find it helpful to construct a name by appending "_tb" to the name of the entity of the unit under test. The testbench's entity does not have ports. It is self-contained.

2. The VHDL model for the UUT must be declared to the testbench so that it can be used, like declaring function prototypes in a conventional language. The designation **component** indicates that the UUT is a separate unit that is "plugged into" the testbench.

3. These signals are the nets that "wire" the UUT to the testbench. It is conventional to give them the same names as their corresponding UUT ports. The " :='0' " appended to each line is a special assignment that gives the signals an initial value of '0'.

4. The UUT is **instantiated** into the testbench. The notation "a => a," indicates that port `a` of the UUT is mapped, or connected, to signal `a` of the testbench. Later, the testbench will drive signal `a` with '0' or '1', which will, in turn, drive port `a` of the UUT.

5. The stimulus process asserts and deasserts the signals driving the UUT ports. In the first line, all three inputs are scheduled to be '0'. In the next line, the process suspends, assigning '0' to each input. The process resumes after 100ns, and the simulator moves to the next line, changing one of the inputs. In this way, all eight combinations of the three inputs are applied to the UUT, followed by a delay to observe the system's response. The final wait statement has no criterion for resuming execution, so the process and the simulation halt here.

**Checkpoint 7.2.2** Using Listing 7.2.1, p. 93 as a template, write a testbench for the two-input one-hot decoder, Exercise 6.5.2, p. 89.

**Answer**.

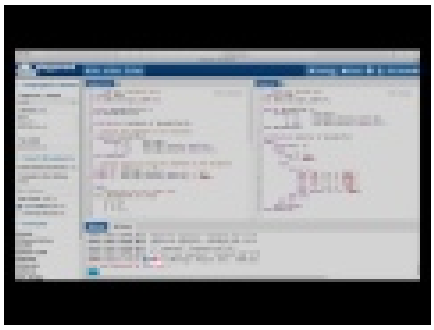
```
-- Testbench for two-input one-hot decoder, with enable
library ieee;
use ieee.std_logic_1164.all;

entity dec2to4_en_tb is
end dec2to4_en_tb;

architecture testbench of dec2to4_en_tb is

   -- Declare UUT
   component dec2to4_en is
      port( en:    in  std_logic;
            x:     in  std_logic_vector(1 downto 0);
            y:     out std_logic_vector(3 downto 0) );
    end component;

   -- Declare and initialize signals
   signal en: std_logic := '0';
   signal x:  std_logic_vector(1 downto 0) := "00";
   signal y:  std_logic_vector(3 downto 0);

begin

   -- Instantiate UUT and wire to signals
   uut: dec2to4_en port map
           en => en,
           x  => x,
           y  => y );

   -- Generate stimuli
stim_proc:  process
   begin
        en <= '0';
        x <= "00";  wait for 100ns;

        x <= "01";  wait for 100ns;   -- en is still '0'
            until it is changed
        x <= "10";  wait for 100ns;
        x <= "11";  wait for 100ns;

        en <= '1';                    -- Now enable the
            decoder
        x <= "00";  wait for 100ns;

        x <= "01";  wait for 100ns;
        x <= "10";  wait for 100ns;
        x <= "11";  wait for 100ns;

        en <= '0'; wait;             -- Disable the decoder
            and halt
   end process stim_proc;

   end testbench;
```

**Listing 7.2.3** VHDL testbench for enabled two-to-four decoder model, Exercise 6.5.2, p. 89

## 7.3 Simulating with EDA Playground

Once you have a VHDL model and a testbench, you need a simulator to execute them and to display the results. One simple simulator that we have used in this course is EDA Playground [5], a free online tool by Doulos, Inc. In the lab we use the Vivado simulator, supplied by Xilinx. Vivado integrates design capture and simulation with tools for synthesis and implementation. At this time, EDA Playground is fine to use with these readings and for in class exercises, but for all lab and project work you will use the Xilinx tools.

Watch this video to learn how to get started with EDA Playground.



**Checkpoint 7.3.1 Simulation: Two-input multiplexer.** Try simulating the **two-input multiplexer**[1].

**Checkpoint 7.3.2 Simulation: Two-input decoder.** Try simulating the **two-to-four decoder**[2].

### Exercises

1. **Simulation: Four-input multiplexer.** Write a testbench for, and simulate, the four-input multiplexer[3]

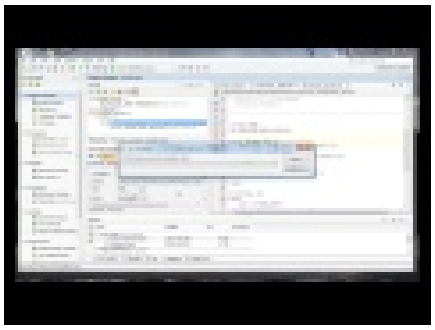
   **Answer**. Here is just the stimulus process. The four inputs `d, c, b, a` are packed into a vector `x(3 downto 0)`. Single input ports are extracted from the vector. The testbench sets the select bits and asserts the data bit of interest. Then it toggles the inputs, to demonstrate that the unselected inputs do not pass.

[1] `www.edaplayground.com/x/4ZGB`
[2] `www.edaplayground.com/x/4ctJ`

```
a <= x(0);
b <= x(1);
c <= x(2);
d <= x(3);

stim_proc: process
begin
    s <= "00";
    x <= "0001";  wait for T;
    x <= not(x);  wait for T;

    s <= "01";
    x <= "0010";  wait for T;
    x <= not(x);  wait for T;

    s <= "10";
    x <= "0100";  wait for T;
    x <= not(x);  wait for T;

    s <= "11";
    x <= "1000";  wait for T;
    x <= not(x);  wait for T;

    wait;
end process stim_proc;
```

## 7.4 Simulating with Xilinx Vivado

Here is a video showing how to get started with the Vivado simulator:



In both the Vivado simulator and in EDA Playground, errors are indicated by changing the color of the waveform and also by labeling the logic value something other than '0' or '1'. VHDL, as a language originally developed for modeling and simulation, actually defines nine possible values for a signal or port of type std_logic. The most important ones for us, so far, are:

- 'U': Uninitialized. When a signal is declared, it has this value until it is assigned '1' or '0'.
- 'X': Unknown. The simulator is unable to figure out what value to assign. This happen, for instance, if you try to operate on an uninitialized signal. If a signal `A` is uninitialized, then `y <= A and B;` will return 'X' for `y`. Many "red X errors" can be avoided by being careful to initialize all the signals in an architecture, e.g., `signal A: std_logic := '0';`.

[3] `www.edaplayground.com/x/2GeL`

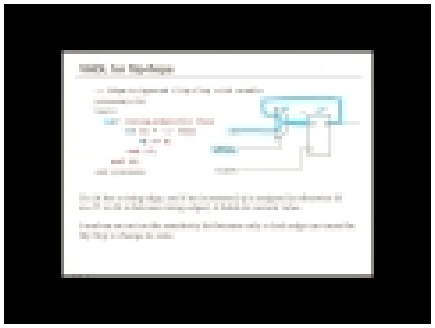
- '0': Logic 0
- '1': Logic 1

With experience, you will learn to trace occurrences of 'U' and 'X' back from a simulation to the coding error that produced them, and make the necessary corrections.

# Chapter 8

# Modeling synchronous systems

This video gives an overview of the topics in this chapter: how to model flip-flops and registers in VHDL.



## 8.1 VHDL flip-flop model

The basic D flip-flop changes its state in response to the rising edge of a clock signal. This is modeled with a process, like this:

```
process(clk)                    -- 1
begin
   if rising_edge(clk) then     -- 2, 5
      Q <= D;                   -- 3
   end if;
end process;                    -- 4, 6
```

**Listing 8.1.1** Basic process for an edge-triggered flip flop. Comments are keyed to the explanation in the text.

Let's walk through the operation of this process and see how it models the operation of a flip-flop:

1. The process is sensitive only to the `clk>` input. Between events on `clk` (edges), the process is suspended. The `D` input relies on a rising clock edge; it cannot, on its own, effect a change in the state of the flip-flop.

2. On a rising clock edge, the process activates. Inside, `if rising_edge(clk)` checks if the event is a rising edge or a falling edge. A rising edge allows

the next statement, `Q <= D;`, to be evaluated.

3. The assignment of `D` to `Q` is scheduled.

4. When the end of the process is reached, `Q` is assigned and the process suspends.

5. Later, the clock has a falling edge. This event also activates the process. However, because the event is a falling edge, the `rising_edge(clk)` function returns FALSE. The assignment statement `Q <= D;` is not evaluated.

6. At the end of the process, no change is made to to `Q`. The process idles, pending the next clock event.

This basic template can be embellished in several ways to add features. For example, here is a model for a flip-flop with enable:

```
process(clk)
begin
   if rising_edge(clk) then
      if CE = '1' then
         Q <= D;
      end if;
   end if;
end process;
```

**Listing 8.1.2** VHDL model for an enabled edge-triggered flip flop.

In this model, the rising clock edge allows the inner if-then to be evaluated. Then, if `en` is TRUE, `Q` is updated. Otherwise, an assignment of `Q` is not made, and the flip-flop holds its state.

Do not write `if rising_edge(clk) and (CE='1') then...`. While it is logically equivalent to the nested if statements, it implies that the clock is ANDed with the enable signal, making a gated clock. After simulation, when the VHDL model is synthesized to a circuit, an intervening AND gate will cause the flip-flop's clock to be delayed, potentially introducing timing errors. The correct way is shown in Listing 8.1.2, p. 100. In general, the VHDL model for a synchronous system must always be of the form:

```
if rising_edge(clk) then
   -- Everything else
end if;
```

**Checkpoint 8.1.3** Write a VHDL model for an edge-triggered flip-flop with synchronous reset.

**Answer**.

```
process(clk)
begin
   if rising_edge(clk) then
      if reset = '1' then
         Q <= '0';
      else
         Q <= D;
      end if;
   end if;
end process;
```

**Listing 8.1.4** VHDL model for an edge-triggered flip flop with synchronous reset.

In the solution of preceding exercise, Listing 8.1.4, p. 100, the reset function is the first to be evaluated, in keeping with the usual priority of reset over loading. In principle, one could also write the model like this:

```
process(clk)
begin
   if rising_edge(clk) then
      Q <= D;
      if reset='1' then
         Q <= '0';
      end if;
   end if;
end process;
```

or even this:

```
process(clk)
begin
   if rising_edge(clk) then
      Q <= '0';
      if reset='0' then
         Q <= D;
      end if;
   end if;
end process;
```

However, neither of these, in my opinion, is as clear as the original solution in expressing the priority of reset over load. Always strive for readable code.

### Exercises

1. **Flip-flop with two controls.** Write a VHDL model for an edge-triggered flip-flop with synchronous enable and reset controls. Reset has priority over enable.

   **Answer**.

   ```
   process(clk)
   begin
      if rising_edge(clk) then
         if reset = '1' then
            Q <= '0';
         elsif CE = '1' then
            Q <= D;
         end if;
      end if;
   end process;
   ```

   **Listing 8.1.5** VHDL model for an edge-triggered flip flop with synchronous reset and enable.

   Note how the nested if-then statements express the priority of reset over enable.

## 8.2 Registers

It is easy to extend the model for a single flip-flop to a multi-bit register.

```
-- D and Q are previously declared std_logic_vector(7
    downto 0)

process(clk)
begin
   if rising_edge(clk) then
      if CE = '1' then
         Q <= D;
      end if;
   end if;
end process;
```

**Listing 8.2.1** VHDL model for an eight-bit parallel-load register with enable.

Notice that, except for the vector nature of `D` and `Q`, the model is identical to Listing 8.1.2, p. 100! Including a synchronous reset is equally straightforward.

**Checkpoint 8.2.2** Write a VHDL model for an eight-bit register with synchronous enable and reset. As usual, reset has priority over enable.

**Hint**. See Listing 8.1.5, p. 101

## 8.3 Counters

A counter is a register combined with an incrementer, Figure 8.3.1, p. 102.

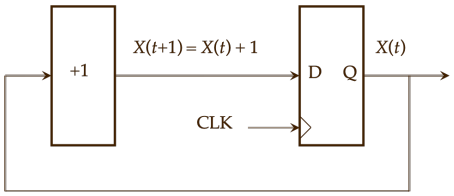



**Figure 8.3.1** A counter consists of a register to hold the current state, and an incrementer to calculate the next state.

We can model this in VHDL with a register and logic equations for the incrementer. Using previously derived equations, the model for a two-bit binary counter can be written like this, with two concurrent processes:

```
-- D and X are previously declared std_logic_vector(1
    downto 0)

reg: process(clk)                -- State update (clocked)
begin
   if rising_edge(clk) then
            X <= D;
      end if;
   end if;
end process reg;

incr: process(X)                 -- Next state
    (asynchronous)
begin
   D(0) <= not(X(0));
   D(1) <= X(1) xor X(0);
end process incr;
```

**Listing 8.3.2** VHDL model for a two-bit binary counter.

This model clearly separates the register that stores the counter's state from

the incrementer logic. The `reg` process updates on every rising clock edge. The update to `X` causes an event that activates the `incr` process, which computes new inputs for the register. This matches the behavior of the block diagram, Figure 8.3.1, p. 102.

While it is often advantageous to separate the state register and the next state logic in this way, for something as simple as a counter it is needlessly complicated. We can accomplish the same thing, without any loss of clarity, with one process:

```
-- X is previously declared std_logic_vector(1 downto 0)

ctr2: process(clk)
begin
   if rising_edge(clk) then
      X(0) <= not(X(0));
      X(1) <= X(1) xor X(0);
   end if;
end process ctr2;
```

**Listing 8.3.3** VHDL model for a two-bit binary counter, written with one synchronous process.

A rising clock edge activates the process and causes the two expressions for `X` to be evaluated. The new value of `X(0)` is scheduled to be the inverse of the current value of `X(0)`. The new value of `X(1)` is scheduled to be the exclusive-OR of the two current values of `X(0)` and `X(1)`. At the end of the process, the two `X` bits are simultaneously updated to their new values and execution suspends, pending the next clock event.

This is a more compact expression for the counter, but it is not easily extended to more than two bits. We would need a new logic equation for each additional bit, and successive equations would be more complex, depending on all the less-significant bits.

VHDL provides numeric data types with arithmetic operations defined. Here is how the same counter is written, using these capabilities.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;                              -- 1

entity counter2 is
   port( clk:    in  std_logic;
         X:      out std_logic_vector(1 downto 0) );
end counter2;

architecture behavior of counter2 is
   signal uX:    unsigned(1 downto 0);                 -- 2

begin
ctr2: process(clk)
   begin
      if rising_edge(clk) then
        uX <= uX + 1;                                  -- 3
      end if;
   end process ctr2;

   X <= std_logic_vector(uX);                          -- 4
end behavior;
```

**Listing 8.3.4** VHDL model for a two-bit binary counter, using arithmetic operations. Comments are keyed to the explanation in the text.

> Libraries have been devised that define arithmetic for std_logic_vectors, but these are not IEEE standard and should be avoided. They are not used in this course.

Here are the important new VHDL features in this model:

1. The additional library, **ieee.numeric_std** [6], [1], pp 298ff, and [15], supplies the arithmetic data types and operations for unsigned (nonnegative) and signed (positive and negative) arithmetic. We need the numeric_std library because the std_logic_vector type does not support arithmetic.

2. The counter's state is held by a signal of type `unsigned`.

3. The state update is a simple arithmetic operation. The numeral `1` is an integer, and by the VHDL rules of unsigned arithmetic, an integer may be added to an unsigned number. Without this, the expression would be written uX <= uX + "01", explicitly writing the increment as a two-bit binary number. The two-bit addition is modulo-4: $3 + 1 = 0$.

4. By convention, all entity ports in a VHDL model must be std_logic or std_logic_vector. This line a type cast, converting the internal unsigned count to std_logic_vector type.

The biggest advantage of this version of the counter is that it easily extends to larger numbers by simply increasing the lengths of `X` and `uX`.

## Exercises

1. **Counter with two controls.** Modify the counter in Listing 8.3.4, p. 104 to have eight bits, with synchronous reset and enable. As usual, reset has priority over enable.

**2. Up-down counter.** Modify the counter in Listing 8.3.4, p. 104 to be an eight-bit up-down counter. When control input UPDOWN is '0', the counter counts up, and when it is '1', the counter counts down. Counting up, 255 wraps around to 0, and counting down, 0 wraps around to 255.

## 8.4 Testbenches for synchronous systems

The new feature that must be added to a testbench for a synchronous system is a clock generator. Here is an example, a testbench for the simple two-bit counter Listing 8.3.4, p. 104.

```
-- Testbench for two-bit binary counter
library ieee;
use ieee.std_logic_1164.all;

entity counter2_tb is
end counter2_tb;

architecture testbench of counter2_tb is

-- Declare UUT
component counter2 is
   port( clk:  in  std_logic;
         X:    out std_logic_vector(1 downto 0) );
 end component;

-- Declare and initialize signals
constant CLK_PERIOD: time := 100ns;               -- 1
signal clk: std_logic := '0';
signal X:   std_logic_vector(1 downto 0) := "00";

begin

-- Instantiate UUT and wire to signals
uut: counter2 port map
        clk => clk,
        X   => X );

-- Generate stimuli
clk_proc:  process                                 -- 2
begin
     clk <= '1';
     wait for CLK_PERIOD/2;
     clk <= '0';
     wait for CLK_PERIOD/2;
end process clk_proc;

--stim_proc:  process                              -- 3
--begin
--end process stim_proc;

end testbench;
```

**Listing 8.4.1** VHDL testbench for two-bit counter model, Listing 8.3.4, p. 104. Comments are keyed to explanations in the text.

Three features of this testbench distinguish it from previous testbenches:

1. The clock period is declared as a named constant. As in conventional

programming languages, it is always advisable to use a symbolic name for numerical constants. Then, if the value needs to be updated in the future, it is only necessary to change the numerical value once, rather than at every appearance of the value in the code. VHDL includes a datatype `time`, which can be expressed in seconds (s), milliseconds (ms), microseconds (us), and nanoseconds (ns). Here, assigning `CLK_PERIOD` the value `100ns` establishes a 10MHz clock.

2. A separate process is used to generate the clock. It sets the `clk` signal HIGH, waits for half the clock period (50ns), then sets it LOW, and waits for half the clock period. The process resumes from the top when the wait time is up, and repeats without end, generating a clock with 100ns period.

3. There are no stimuli for this counter other than the clock, so the normal stimulus process is not used in this testbench.

**Checkpoint 8.4.2 Simulation: Four-bit binary counter.** Here is a VHDL model and testbench for a **four-bit binary counter**[1]. Run the simulation and observe the behavior in the timing diagram.

If we want to simulate a counter with reset and enable, Exercise 8.3.1, p. 104, we need to generate the enable and reset signals in the testbench. We can do this with the stimulus process.

When you design a testbench, you want to make sure that you cover the cases of interest and not miss important edge cases. Here, it is important to demonstrate that the counter can be reset to zero. An exhaustive test would ensure that the counter could be reset from every state, e.g., count to 1, reset, count to 2, reset, count to 3, reset, ..., but this is too much. We can be confident, from the way the model is written, that if reset works for an arbitrary count value, it will work for all values. Likewise, with the enable, we want to show that the counter can be stopped, then started back up from where it left off. We of course want to show that the counter will still roll over from its maximum count to zero. And, finally, we want to demonstrate the priority of reset over enable; the counter should be resettable whether it is running or stopped. Study how these tests are implemented in the testbench shown in Listing 8.4.3, p. 107.

[1] `www.edaplayground.com/x/2ww2`

```
-- Testbench for binary counter with reset and enable
library ieee;
use ieee.std_logic_1164.all;

entity counter_reseten_tb is
end counter_reseten_tb;

architecture testbench of counter_reseten_tb is

-- Declare UUT
component counter_reseten is
   port( clk:   in  std_logic;
         reset: in  std_logic;
         CE:    in  std_logic;
         y:     out std_logic_vector(7 downto 0) );
 end component;

-- Declare and initialize signals
constant CLK_PERIOD: time := 100ns;
signal clk:       std_logic := '0';
signal reset, CE: std_logic := '0';
signal y:         std_logic_vector(7 downto 0) := x"00";

begin

-- Instantiate UUT and wire to signals
uut: counter_reseten port map
        clk   => clk,
        reset => reset,
        CE    => CE,
        y     => y );

-- Generate stimuli
-- These processes run concurrently.
clk_proc:  process
begin
     clk <= '1';
     wait for CLK_PERIOD/2;
     clk <= '0';
     wait for CLK_PERIOD/2;
end process clk_proc;

stim_proc:  process
begin
     CE <= '0';  reset <= '1';     -- Hold in reset
     wait for 2*CLK_PERIOD;        -- for two clock cycles

     reset <= '0';                 -- Release reset, but
         still holding
     wait for 2*CLK_PERIOD;

     CE <= '1';                    -- Enable counting
     wait for 5*CLK_PERIOD;

     reset <= '1';                 -- Reset while counting
     wait for 2*CLK_PERIOD;

     reset <= '0';                 -- Start counting again
     wait for 4*CLK_PERIOD;
     CE <= '0';                    -- Hold while counting
     wait for 4*CLK_PERIOD;

     CE <= '1';                    -- Re-enable and
     wait;                            -- let it run
end process stim_proc;

end testbench;
```

**Checkpoint 8.4.4 Simulation: Counter with enable and reset.** Run your solution to Exercise 8.3.1, p. 104 with this testbench, in EDA Playground[2].

[2] www.edaplayground.com/x/4WH9

# Chapter 9

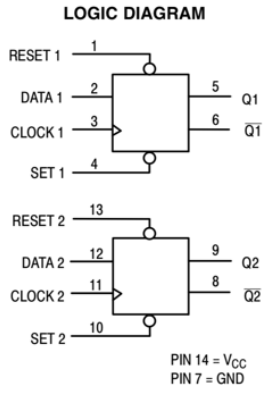
# Modeling mixed synchronous-asynchronous systems

A purely combinational circuit like a multiplexer or decoder is modeled by a VHDL process that sensitive to all the circuit's inputs. The circuit responds immediately (after a propagation delay) to changes in its inputs. A system like this, which operates without a clock, is said to be **asynchronous**. On the other hand, the flip-flops, registers, and counters we have seen so far respond only to clock edges. Their other inputs, like data (D) or enable (CE), do not have the ability to activate the circuit. Systems like these, whose activations are synchronized to a clock, are called **synchronous**.

There are also systems that have both synchronous and asynchronous parts. This chapter is about how to model such mixed circuits in VHDL.

## 9.1 Flip-flop model with asynchronous controls

Some classic discrete integrated circuits containing flip-flops have the ability to set and clear the state at any time, independent of the clock. A simple example, the HC74A dual flip-flop [9], is shown below.




**FUNCTION TABLE**

| Inputs | | | | Outputs | |
|---|---|---|---|---|---|
| Set | Reset | Clock | Data | Q | $\overline{Q}$ |
| L | H | X | X | H | L |
| H | L | X | X | L | H |
| L | L | X | X | H* | H* |
| H | H | ↑ | H | H | L |
| H | H | ↑ | L | L | H |
| H | H | L | X | No Change | |
| H | H | H | X | No Change | |
| H | H | ↓ | X | No Change | |

*Both outputs will remain high as long as Set and Reset are low, but the output states are unpredictable if Set and Reset go high simultaneously.

**Figure 9.1.1** Logic diagram (left) and function (truth) table (right) for the HC74A dual D flip-flop with asynchronous set and clear [9].

Each flip-flop has its own clock, data, set and reset inputs. SET and RESET are low-true (shown by the bubbles). On the truth table, SET and RESET inputs are both HIGH, the flip-flop operates in the usual edge-triggered mode.

When SET is HIGH and RESET is LOW, the output, Q, immediately (after a small propagation delay) becomes LOW (0). The complementary output, Q', becomes HIGH (1). These transitions, in response to the SET and RESET inputs without regard to the clock, are asynchronous. Similarly, when SET is LOW and RESET is HIGH, Q immediately becomes HIGH (1), and Q' becomes LOW (0), asynchronously.

The HC74A has an additional, problematic "feature". If SET and RESET are both LOW (that is, TRUE), the circuit enters a paradoxical state where Q and Q' are simulaneously HIGH. After this, when one of the inputs goes HIGH, the output resolves to one of the other defined outcomes. In the very rare and undesirable case of SET and RESET becoming HIGH or LOW simultaneously, the circuit's behavior is said to be unpredictable. This strange situation must be avoided, by using only one of the control inputs and tying the other HIGH to disable it, or by tying both inputs HIGH and using the flip-flop as a purely synchronous device. There are good applications of asynchronous inputs, but in this course you are instructed not to use them.

That said, here is how the HC74A would be modeled...almost. Rather than trying to accurately account for the SET=RESET=LOW situation, this model gives SET priority over RESET. Additionally, according to the truth table, SET and RESET have priority over the clock.

```
-- clock1, data1, set1, reset1, Q1, Qbar1 are std_logic
HC74A_1: process(clk, set1, reset1)
   begin
      if set1='0' then               -- low-true
         Q1 <= '1';
         Qbar1 <= '0';
      elsif reset1='0' then          -- low-true
         Q1 <= '0';
         Qbar1 <= '1';
      elsif rising_edge(clock1) then   -- Asynch inputs
          have priority over synch
         Q1 <= data1;
         Qbar1 <= not(data1);
      end if;
   end process HC74A_1;
```

**Listing 9.1.2** VHDL model of one half of the MC74HC74A dual flip-flop [9].

### Exercises

1. **Simulation: HC74A.** Write a testbench for the HC74A model, Listing 9.1.2, p. 110, that verifies all the functions specified in the truth table, Figure 9.1.1, p. 109. Then carry out a simulation.

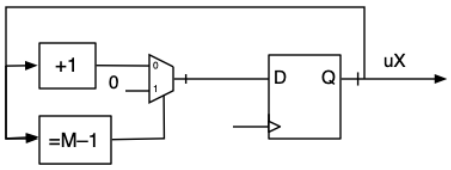
## 9.2 Counters with terminal count output

Another important application mixing asynchronous and synchronous functionality is a counter that asserts an output when its terminal count is reached. We'll approach the design in two stages: first, modeling a counter with an arbitrary maximum count, and second, asserting a terminal count signal when that maximum is reached.

**Counter with arbitrary maximum count.** Recall the basic four-bit counter model (see Listing 8.3.4, p. 104).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter4 is
   port( clk:    in  std_logic;
         X:      out std_logic_vector(3 downto 0) );
end counter4;

architecture behavior of counter4 is
   signal uX:    unsigned(3 downto 0);

begin
ctr8: process(clk)
   begin
      if rising_edge(clk) then
        uX <= uX + 1;
      end if;
   end process ctr4;

   X <= std_logic_vector(uX);
end behavior;
```

This counter rolls over to 0 when it reaches 15. A more flexible counter can be designed with an arbitrary upper limit, $M - 1$. To do this, we have to detect that the count is $M - 1$, with a comparator, and then use that signal to clear (load zeros into) the counter. A block diagram looks like this:




The comparator output is the select bit for a multiplexer, choosing between the incremented count and zero. The multiplexer and the register are combined into a single process in this model (for counting from 0 to 9, also known as a **decade counter**):

```
architecture behavior of counter4 is
   constant M: integer := 10;          -- 1
   signal uX:  unsigned(3 downto 0);

begin
ctr4: process(clk)
   begin
      if rising_edge(clk) then
         if uX = M-1 then              -- 2
            uX <= "0000";
         else
            uX <= uX + 1;              -- 3
         end if;
      end if;
   end process ctr4;

   X <= std_logic_vector(uX);
end behavior;
```

**Listing 9.2.1** VHDL model for a decade (0-9) counter. Comments are keyed to explanations in the text.

New features in this model are:

1. The upper limit of the counter is defined via a named integer constant, following good programming practice. `M` is 10, the number of counter states (0, 1, ..., 9).

2. The numeric_std library includes all the arithmetic comparison operators, such as = [6]. A signal of unsigned (or signed) type may be compared with a constant of integer type. If the count is seen to be $M - 1$, then it is reset to 0.

3. Otherwise, the count is incremented.

**Asserting the terminal count signal.** Next, we need to assert the terminal count signal, `TC`, when the count equals $M - 1$. We already have a comparator in the logic diagram, above, so we just need to assign the comparator's output to `TC`.

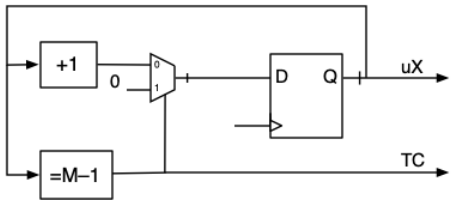



**Figure 9.2.2** Logic diagram of a counter with terminal count output.

One way model this in VHDL is to put the counter update logic in one synchronous process, and the comparator in a concurrent asynchronous process. The complete VHDL model is shown below.

```
ctr4: process(clk)
begin
   if rising_edge(clk) then
      if iTC = '1' then
         uX <= "0000";
      else
         uX <= uX + 1;
      end if;
  end if;
end process ctr4;

TCcomp: process(uX)
begin
  if uX = M-1 then
      iTC <= '1';
  else
      iTC <= '0';
  end if;
end process TCcomp;
```

This is logically and syntactically correct, and will produce the desired behavior in simulation. Its disadvantage, stylistically, is that the separation of the counter into two processes makes the code less readable. The model is made more compact and readable if all the counter's functionality is packed into a single process, like this:

```
entity counter4 is
      port( clk:    in  std_logic;
            X:      out std_logic_vector(3 downto 0);
            TC:     out std_logic; );
   end counter4;

   architecture behavior of counter4 is
      constant M:   integer := 10;     -- decade counter
      signal uX:    unsigned(3 downto 0) := "0000";
      signal iTC:   std_logic := '0';  -- 1

   begin
      ctr4: process(clk, uX)           -- 2,5,8
      begin
      -- Synchronous count update
         if rising_edge(clk) then      -- 3,6,9
            if iTC = '1' then
               uX <= "0000";
            else
               uX <= uX + 1;
            end if;
        end if;

       -- Asynchronous TC comparator
        if uX = M-1 then               -- 4,7
            iTC <= '1';
        else
            iTC <= '0';
        end if;
      end process ctr4;

      X <= std_logic_vector(uX);
      TC <= iTC;
   end behavior;
```

**Listing 9.2.3** VHDL model of a decade counter with terminal count output. See Figure 9.2.4, p. 114 for timing diagram. Comments are keyed to notes in the text.

Follow the steps in the execution of the process and see how the synchronous and asynchronous parts mesh. Assume an initial count of 8.

1. Declare internal signals, `uX` and `iTC`, that may be read and written. The ouptut ports `X` and `TC` may only be written, not read.

2. A rising clock edge causes the process to activate.

3. Because `uX` is 8, (internal) terminal count `iTC` was previously set to 0, so the counter will update from 8 to 9.

4. The current value of the count (8) does not yet equal 9, so `iTC` will be '0'.

5. The process suspends with `uX`=9 and `iTC` = '0'. These values are mapped to the output ports `X` and `TC`. The change in `uX` from 8 to 9 makes an event on `uX` and reactivates the process (`uX` is on the sensitivity list in addition to `clk`).

6. The synchronous part does not evaluate, because there is no clock edge.

7. The current value of the count (9) equals 9, and `iTC` will be asserted when the process suspends.

8. The next rising edge will activate the process.

9. Because `iTC` is now 1, the counter will reset when the process suspends.

Simulation, Figure 9.2.4, p. 114, shows that the counter resets and the terminal count signal is asserted at the appropriate time.

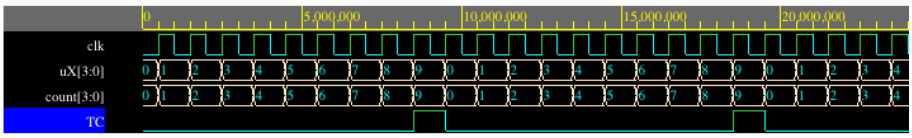



**Figure 9.2.4** Timing diagram for counter with terminal count (TC) output, produced by simulation of the VHDL model, Listing 9.2.3, p. 113. The counter resets and the TC output is asserted when the count is 9.

## Exercises

1. **Safe decade counter.** The decade counter design, Listing 9.2.1, p. 111, has one vulnerability. If a glitch were to cause it to skip to a value greater than 9, it would keep counting until it naturally rolled over at 15. Modify the design to avoid this problem.

   **Answer**. Change the comparator from $=$ to $\geq$:

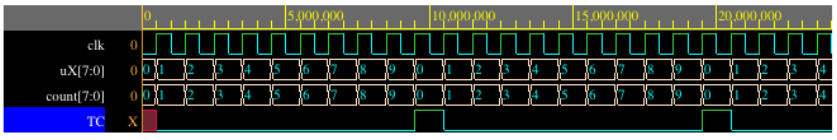
```
ctr4: process(clk)
   begin
      if rising_edge(clk) then
         if uX >= M-1 then
            uX <= "0000";       -- reset
         else
            uX <= uX + 1;       -- increment
         end if;
   end process ctr4;
```

2. **An incorrectly modeled terminal count.** A common mistake is to write the VHDL model for a counter with terminal count output like this:

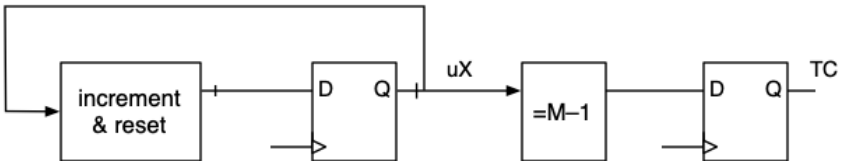
```
ctr4: process(clk)
begin
   if rising_edge(clk) then
      if uX = M-1 then
         uX <= "0000";
         TC <= '1';
      else
         uX <= uX + 1;
         TC <= '0';
      end if;
   end if;
end process ctr4;
```

   Simulate this design and observe that the TC signal is asserted at the wrong time. Reverse engineer the VHDL model to a logic diagram, and use it to explain what went wrong.

   **Answer**. The simulation is:



The TC signal is asserted when the count is 0, one clock cycle delayed. The logic diagram represented by this model is:




In the VHDL model, the TC signal is updated on the rising edge, so it is stored in a flip-flop. When the count, `uX`, is 9, the `TC` signal is not set until the next clock edge, at which time `uX` is also set to 0. The extra flip-flop following the comparator is introducing the erroneous delay. This is a very common design pitfall.

## 9.3 A template for mixed synchronous and asynchronous processes

```
process(clk, A, B, ...)           -- 1
begin
   if rising_edge(clk) then       -- 2
      -- synchronous logic
   end if;

   -- asynchronous logic          -- 3
end process;
```

**Listing 9.3.1** A template for combining synchronous and asynchronous logic in the same VHDL process.

1. The sensitivity list includes the clock and all inputs to the asynchronous part.

2. The synchronous logic is inside the `if rising_edge(clk)...end if` block.. Every signal assignment in this block is registered (stored in flip-flops).

3. The asynchronous logic is outside the `if rising_edge(clk)...end if` block. These signal assignments are not registered.

# Chapter 10

# Counters

Counters serve multiple purposes in a digital system. They can keep time, divide clock frequencies, index arrays, and do simple event sequencing. This chapter explains how to design counters for these and other purposes.

## 10.1 Overview of simple counters

The most basic counter is a register to hold the current value of the count, and an incrementer to generate the next value in the count sequence (Figure 8.3.1, p. 102). Enhancements to this basic setup include:

- Enable (count-hold control)
- Clear (reset control)
- Direction (up-down control)
- Variable step size (count increment)
- Load (initial value or override value)
- Arbitrary count limit (terminal count)
- Count sequence (straight binary, Gray code...)

This more capable counter includes the current state register, and combinational logic to generate the counter's next state, based on the current state and the external inputs, as in Figure 10.1.1, p. 117, below.

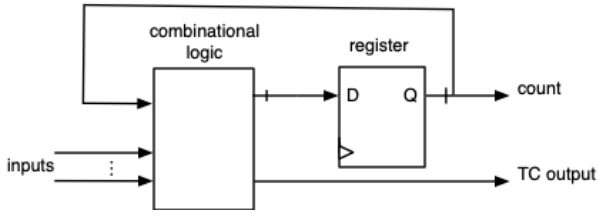



**Figure 10.1.1** A counter consists of a register to hold the current value, and combinational logic to determine the next value of the count based on the current count and external inputs.

The combinational logic is made from arithmetic elements and multiplexers controlled by inputs. An example, from an earlier section, is a counter that

runs from 0 to a maximum value $M - 1$, and generates a terminal count (TC) signal when the maximum value is reached, Figure 9.2.2, p. 112. The VHDL model for a counter may either be purely synchronous or a mixed synchronous-asynchronous process, Listing 9.2.3, p. 113.

## 10.2 Counter controls

The simplest counter is just a register and an incrementer. On each rising clock edge, the count advances, and when it reaches the capacity of the register ($2^N - 1$ for an $N$-bit counter), on the next edge the count resets (or "rolls over") naturally to zero. Control inputs modify this natural sequence, e.g., by halting the count (enable) or forcing it to zero (clear). In this course, all counter controls are designed to be synchronous, acting on the rising clock edge. They are never asynchronous.

When there is more than one control input, the designer must decide how to prioritize them. Typically, the clear function has the highest priority, followed by the enable. These functions are often built into the register itself. In hardware, the priority of control signals is expressed through a sequence of multiplexers, with the highest priority control placed closest to the register Figure 10.2.1, p. 118

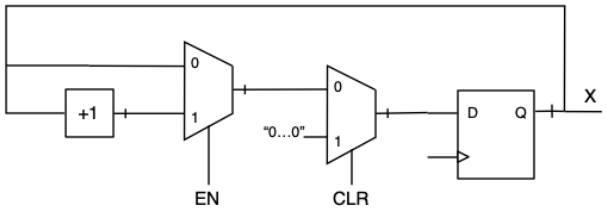



**Figure 10.2.1** Counter with synchronous clear (CLR) and enable (EN) controls. CLR is given priority over EN by placing it closer to the register.

The clear-over-enable priority is modeled in VHDL by nested if-then-else statements.

```
-- Signal X is previously declared as unsigned; all other
    signals are std_logic

   process(clk)
   begin
      if rising_edge(clk) then
         if CLR = '1' then                -- 1
            X <= (others => '0');         -- 2
         elsif EN = '1' then              -- 3
            X <= X+1;
         end if;
      end if;                             -- 4
   end process;
```

**Listing 10.2.2** VHDL model for the counter in Figure 10.2.1, p. 118. Comments are keyed to notes in the text.

The important features of this model (compare to the logic diagram, Figure 10.2.1, p. 118) are:

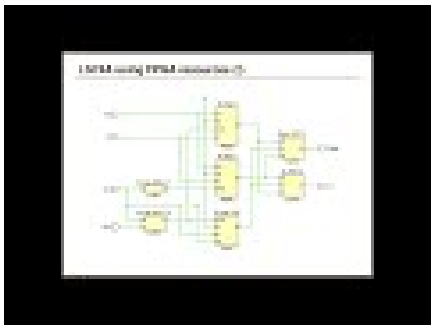
1. By testing `CLR` first, it is given priority over `EN`. The first decision to be made is whether we will clear the counter.

2. The VHDL construction `(others => '0')` conveniently makes a vector "0...0" whose length matches to size of `X`.

3. If `CLR` is not asserted, `EN` is tested and if it is TRUE, the counter is incremented.

4. If neither `CLR` nor `EN` is asserted, no action is taken, and `X` is unchanged. There will be no latch warning (Section 13.3, p. 140) because the process is clocked; we intend for a register to be synthesized.

Watch the following video for more about nested control statements.



## Exercises

1. **Up-down counter with terminal count output.** Modify the counter in Figure 9.2.2, p. 112 to have a single control input, UPDOWN, and to assert the TC signal on $M - 1$ when counting up, and 0 when counting down. Draw a block diagram, and then write a VHDL model (see Listing 9.2.3, p. 113). Validate the model with a testbench and simulation.

2. **Preloadable counter.** Design a counter with a single control input, LOAD, and a (vector) data input, D. When LOAD is asserted, the data at D is synchronously loaded into the counter's register, overriding the current count value. When LOAD=0, the counter counts normally. Draw a block diagram and write a VHDL model based on the block diagram.

   **Answer**. Block diagram:

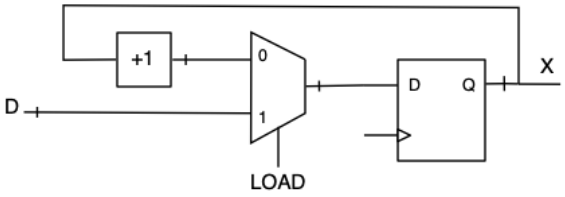



**Figure 10.2.3** Counter with parallel preload inputs.

VHDL model:

```
-- Signals X and D are previously declared as unsigned;
-- all other signals are std_logic.

   process(clk)
   begin
      if rising_edge(clk) then
         if LOAD = '1' then
            X <= D;
         else
            X <= X+1;
         end if;
      end if;
   end process;
```

**Listing 10.2.4** VHDL model for the counter with parallel preload inputs.

3. **Countdown timer.** Design a counter with a single control input, LOAD, a (vector) data input, D, and a terminal count signal, TC. When LOAD is asserted, the data at D is synchronously loaded into the counter's register. When LOAD=0, the counter counts normally, down to zero. When zero is reached, TC is asserted and the counter holds at zero until LOAD is asserted again. Draw a block diagram and write a VHDL model based on the block diagram.

## 10.3 Clock dividers

A counter with maximum count $M-1$ and a terminal count output, TC, counts in the sequence $0, 1, 2, ..., M-1, 0, ...$ and asserts TC once every $M$ clock cycles. The period of the TC signal is $M$ times the clock period, and its frequency is $1/M$ times the clock frequency. This property can be used to make a slower clock signal, e.g., converting a 100 MHz clock to 1 MHz. A circuit which performs this function is called a **clock divider**.

A clock signal is usually required to have a 50% **duty cycle**, that is, it is HIGH for 50% of its period and LOW for the other half-period. The TC output of a counter has a lower duty cycle, because it is only asserted for one clock cycle out of $M$. We cannot just use the TC output as a clock. To obtain a 50% duty cycle output, we use the setup shown in Figure 10.3.1, p. 120.

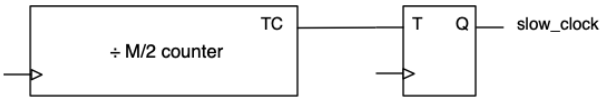



**Figure 10.3.1** Clock divider constructed from a counter and a toggle flip-flop (T-flop). The frequency of the slow_clock signal is 1/M the frequency of the master clock, and it has a 50% duty cycle.

The **toggle flip-flop**, or **T-flop**, is enabled by its T input. When T is TRUE, the T-flop changes its state from 0 to 1 or from 1 to 0 on every rising clock edge. When T is FALSE, the T-flop holds its state. In this circuit, the counter is designed to assert TC every $M/2$ clock cycles ($M$ must be even). For one clock cycle out of every $M/2$, TC is asserted and the T-flop toggles. Thus, the T-flop's output is 1 for $M/2$ clock cycles and 0 for $M/2$ clock cycles. It has a frequency of $1/M$ times the original clock frequency, and its duty cycle is 50%.

The VHDL model for the clock divider process is given below. Compare with Listing 9.2.3, p. 113.

```
-- Signal X is previously declared as unsigned.
-- Constant M is declared as an integer, intialized to an
    even number.
-- Signal slow_clock is std_logic, initialized to '0'

   clock_divider: process(clk)
   begin
      if rising_edge(clk) then
         if X = M/2-1 then                    -- 1
            X <= (others => '0');             -- 2
            slow_clock <= not(slow_clock);    -- 3
         else
            X <= X+1;                         -- 4
         end if;
      end if;
   end process clock_divider;
```

**Listing 10.3.2** VHDL model for a $1/M$ clock divider with 50% duty cycle output. Comments are keyed to explanations in the text.

Note the following about the VHDL model:

1. The numeric_std library, which is included whenever there is arithmetic, allows an unsigned to be compared with an integer. During synthesis, the integer quotient `M/2` is computed and the result is "hardwired" into the synthesized hardware.

2. The VHDL construct `(others => '0')` makes a bit vector "0...0" that matches the length of `X`.

3. The slow_clock signal is toggled every $M/2$ clock cycles. This line results in the synthesis of the T-flop from a D-flip and an inverter.

4. If `X` is not at its terminal count, it is incremented at every rising edge.

### Exercises

1. **Sizing a clock divider.** What is the minimum size, in bits, of the register in a clock divider that reduces a clock from 100MHz to 1MHz?
2. **Clock divider simulation.** Expand the clock divider process, Listing 10.3.2, p. 121, to a full VHDL entity and architecture (I suggest $M = 6$ for compactness), write a testbench, and simulate.
3. **Alternative clock divider.** Design a clock divider using a $\div M$ counter and a comparison operation ($=$, $>$, $<$, ...). Which design do you prefer, and why?

   **Answer**. The counter runs from 0 to M-1. The count X is compared with the constant M/2. If X < M/2, the slow_clock output is asserted. Otherwise, slow_clock is 0.

   Both designs require the same number of flip-flops. The second design requires an extra comparator to produce the slow_clock signal. Thus, the first design may be more efficient.

## 10.4 Counter with variable step size

Sometimes we need a counter with a step size that can be input via a port or signal. One example is shown in Figure 10.4.1, p. 122, below. The count, $X$, and the increment, $\Delta$, are $N$-bit (unsigned) numbers. The adder lacks both a

carry-in and a carry-out; its output is the sum $X + \Delta$, modulo $2^N$. For example, if $N = 4$ and $\Delta = 3$, the count sequence will be 0, 3, 6, 9, 12, 15, 2, 5, 8, 11, 14, 1,...

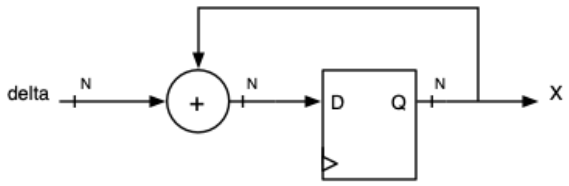



**Figure 10.4.1** Counter with externally supplied step size.

The VHDL model for this counter is also straightforward.

```
-- numeric_std library is included.
-- Signals X and delta are previously declared unsigned(N-1
    downto 0).

   process(clk)
   begin
      if rising_edge(clk) then
         X <= X + delta;
      end if;
   end process;
```

**Listing 10.4.2** VHDL model for an $N$-bit counter with external step size.

The basic counter can be embellished in the usual ways, with reset and/or enable, or adding an output signal that detects when the counter has rolled over.

## Exercises

1. **Variable-step counter with overflow.** Redesign the variable step-size counter so that it asserts an overflow signal, OV, for one clock cycle when the count rolls over, e.g., from 15 to 2. Draw a block diagram, write a VHDL model from the block diagram, and validate the design through simulation.

# Chapter 11

# Synthesis and implementation

Chapter 1, p. 3 introduced the design flow for a digital system:

1. Capture (Section 1.3, p. 8 and Section 1.4, p. 11): Expressing the desired system behavior, e.g., in a truth table and logic equations.
2. Synthesis (Section 1.5, p. 14 and Section 1.6, p. 15): Translating the captured design to a logic diagram and/or a list of required components and a netlist.
3. Implementation (Section 1.6, p. 15): Placing components in a physical layout and routing the connections specified in the netlist.
4. Testing: Verifying that the prototype functions as intended.

This chapter shows how the same design flow works with contemporary EDA tools. Tutorial videos for the Xilinx Vivado toolset are available on the Canvas site for Engs 31.

## 11.1 Programmable logic

VHDL modeling enables the capture of a digital system's behavior and also the verification of a design through simulation, before the design is committed to hardware. EDA tools also perform logic synthesis and implementation, moving from the VHDL model to a hardware prototype.

The hardware prototypes we saw in Section 1.6, p. 15 used so-called small-scale integration (SSI) and medium-scale integration (MSI) technology: individual packages containing the familiar combinational logic (gates, multiplexers, decoders, adders, etc) and memory elements (flip-flops, registers). Modern integrated systems put orders of magnitude more capability into a single small package. This is true not just of finished systems (e.g., microcontrollers), but also prototypes. **Programmable logic** is the name given to devices containing a multitude of combinational and storage elements, with interconnects that can be configured and reconfigured by the user, enabling the flexible creation, testing, and revising of large prototypes.

The dominant programmable logic device in use today is the **field programmable gate array (FPGA)**. The basic structure of an FPGA is an array of thousands of **configurable logic blocks (CLBs)**, each of which contains

one or more **logic slices**, which are pairings of a small block of combinational logic with a flip-flop, Figure 11.1.1, p. 124.

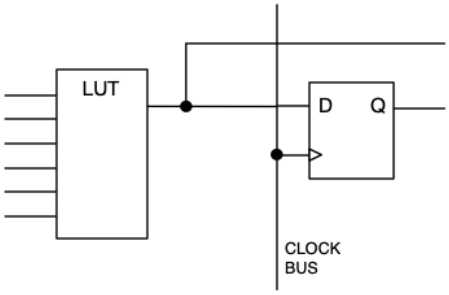



**Figure 11.1.1** A basic FPGA logic slice consists of a lookup table (LUT) for combinational logic and a flip-flop. The slice outputs both the unregistered and registered LUT output. A configurable logic block (CLB) usually contains several slices, and an FPGA contains thousands of CLBs.

The combinational logic in the slice is not constructed from gates, but is itself a small block of programmable memory called a **lookup table (LUT)** Figure 11.1.2, p. 124.

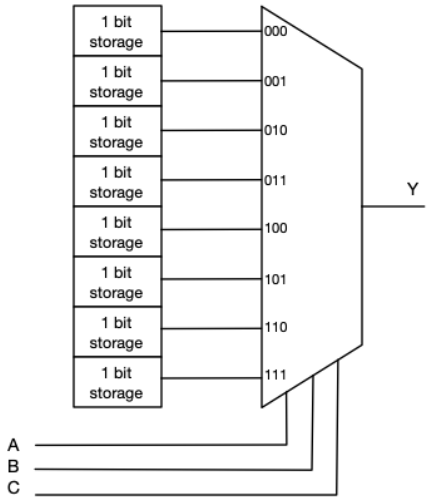



**Figure 11.1.2** Model of a three-input lookup table (LUT) as eight bits of storage and an 8-to-1 multiplexer. The LUT can store the truth table for any function of three logic variables, $Y = f(A, B, C)$.

As shown in the figure, a three-input LUT, storing $2^3 = 8$ bits, can hold the truth table for any logical function of three inputs, $Y = f(A, B, C)$. The appropriate 1s and 0s are stored in the LUT when the FPGA is configured (programmed). The model shown in the figure is a simplification: LUTs used in contemporary FPGAs have four or six inputs. The LUT output may be routed either to a flip-flop, for storage, or to an input of another LUT, to make a more complex logical function, as shown in Figure 11.1.1, p. 124.

The array of CLBs is enmeshed in a network of wires resembling the traffic lanes in a city. Signals are routed from CLB to CLB along these wires with switches that are set when the FPGA is configured. The clock lines for the flip-flops have their own special paths designed so that, as much as possible, the clock arrives at each flip-flop at the same time. Special circuits, called **buffers**, interface the internal wiring to pins on the FPGA package and the

outside world.

## 11.2 Mapping ports to pins

The ports of a VHDL entity are the system inputs and outputs. Physical connection of the system ports to the outside world are made through pins on the FPGA package. In Xilinx Vivado, the mapping from ports to pins is specified by a **constraints file** that is processed along with the VHDL source files.

In this class, the FPGA is mounted on a **development board** with switches, LEDs, and a variety of input/output connections, Figure 11.2.1, p. 125.

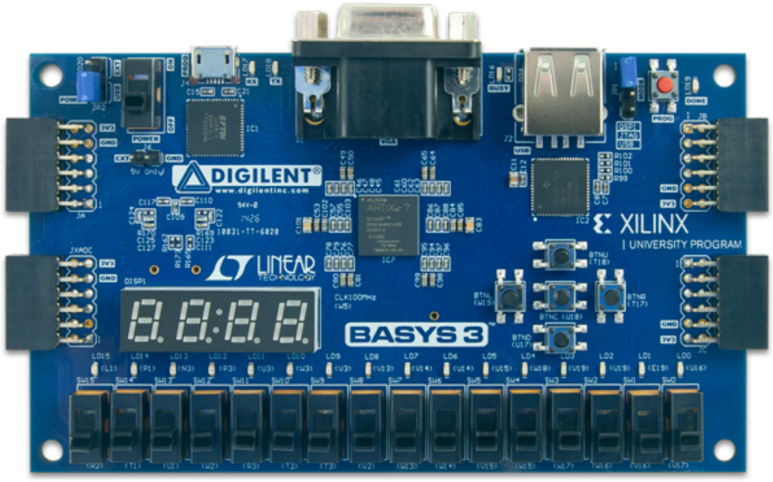



**Figure 11.2.1** Top view of the Digilent Basys3 development board. For a more detailed view, see Figure 1 and Table 1 of the reference manual, [3].

**Checkpoint 11.2.2 How big is our FPGA?** According to the Basys3 documentation, what FPGA is used on the board? Then use the Xilinx website[1] to find out how many LUTs and flip-flops this FPGA possesses.

Each switch, LED, etc, on the development board is wired to a pin on the FPGA, as described in the documentation [3]. These connections are listed in a corresponding constraints file, available for download[2] from the Digilent website. As an example of how the constraints file is written, three entries are shown below.

[1] www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf

[2] github.com/Digilent/digilent-xdc/blob/master/Basys-3-Master.xdc

```
set_property PACKAGE_PIN W5 [get_ports clk]
    set_property IOSTANDARD LVCMOS33 [get_ports clk]
    create_clock -add -name sys_clk_pin -period 10.00
        -waveform {0 5} [get_ports clk]

# Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]

#set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
```

**Listing 11.2.3** Three entries from the constraints (XDC) file for the Basys3 board, showing mappings from the FPGA to the board clock and two slide switches. The third entry is commented out by the # character.

The XDC file is not written in VHDL, and consequently the entries have non-VHDL syntax. In particular, port names are case-sensitive ("clk" and "CLK" are read as different names). The Basys3 board's clock signal is supplied by a chip that is wired to pin W5 on the FPGA, as can be confirmed by referring to the Basys3 schematic diagram, [4]. The first entry in the XDC file, `set_property PACKAGE_PIN W5 [get_ports clk]`, indicates that pin W5 will be routed inside the FPGA to an entity port called "clk". In this way the board clock signal is connected through pin W5 to the clock input of your design. If the clock port of your entity is named something other than `clk`, edit this line and the following two lines to use the correct name, taking care to write the name with the same case in the entity and the XDC file. (The two lines following can be ignored for now, except to note that the name of the clock port must be the same in all three lines.)

Likewise, the second entry indicates that one of the sixteen slide switches on the board is wired to FPGA pin V17. Again, look at the schematic [4] to see how this is done. To use this switch as an input to your design, replace the default name `sw[0]` with the name of the appropriate port in your design's entity. This port name must be used in both lines of the entry, and the case must match the case in your entity.

Unused lines in the downloaded XDC file are preceded by the XDC comment character, #, as shown in the third entry.

## 11.3 Logic synthesis for FPGAs

The translation of a VHDL model into an FPGA prototype occurs in two steps: synthesis and immplementation. In logic synthesis, the EDA tool analyzes the VHDL source code to identify the logical resources needed by the prototype. For example, a simple four-bit binary counter contains the process:

```
process(clk)
begin
   if rising_edge(clk) then
      uX <= uX+1;           -- uX is unsigned(3 downto 0)
   end if;
end process;

X <= std_logic_vector(uX);  -- map to output port
```

The synthesis tool infers from this statement that a four-bit register is required to hold `uX`, and an incrementer is needed to update the count. A netlist is created from this decomposition, which can be displayed as a schematic

diagram.

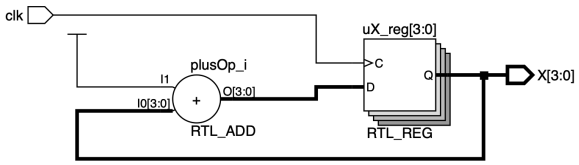



**Figure 11.3.1** Schematic diagram elaborated from the VHDL model for a four-bit counter. The counter consists of a register and an incrementer.

The register, in turn, is decomposed into four flip-flops and the incrementer is decomposed into four lookup tables.

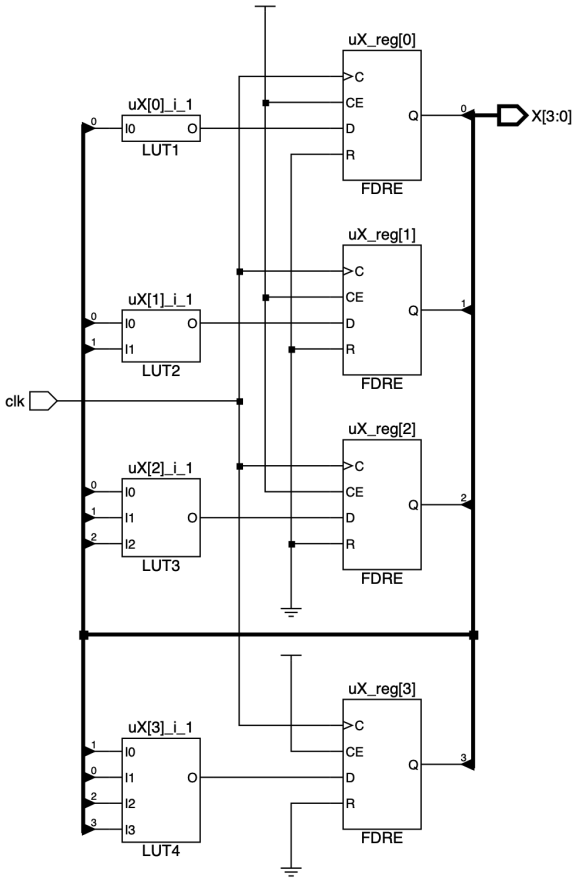



**Figure 11.3.2** Schematic diagram synthesized from VHDL model for a four-bit counter. There are four flip-flops for the counter state and four LUTs for the next-state logic (incrementer). The label "FDRE" denotes a D flip-flop with reset and enable, one of several types of flip-flop that can be synthesized in this FPGA [21].

The Xilinx Vivado tool, which we use in this class, also makes the netlist available in tabular form, with hyperlinking between the table and the schematic:

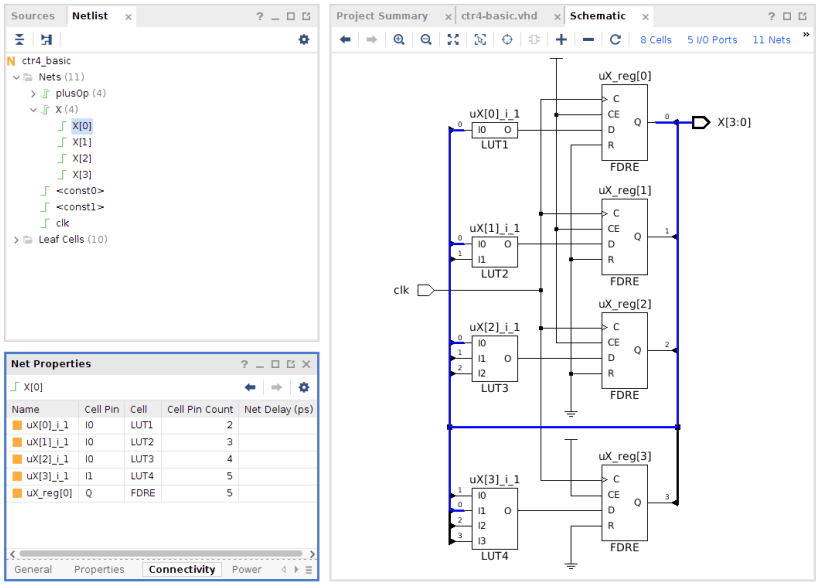



**Figure 11.3.3** Schematic diagram and netlist synthesized from the VHDL model for the four-bit counter. Clicking a net in the netlist highlights the net in the schematic, and vice-versa.

Additionally, the tool enables the designer to peek inside the LUTs to see their truth tables:

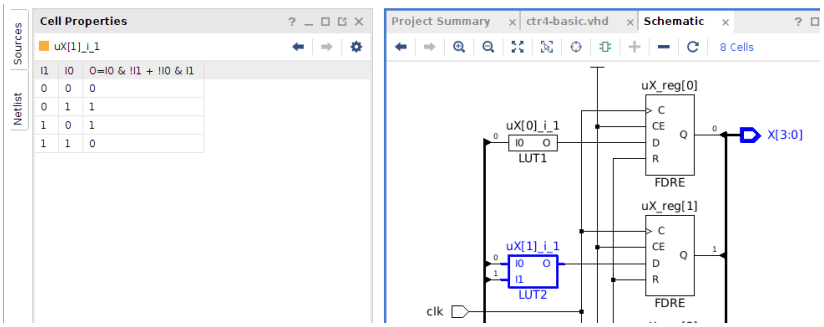



**Figure 11.3.4** Schematic diagram and netlist synthesized from the VHDL model for the four-bit counter. Double-clicking a LUT in the schematic reveals its truth table. The truth table shown implements a two-input exclusive-OR function.

Compare this with logic synthesis for a breadboarded prototype, in which the designer analyzes the logic diagram to determine an effective set of IC packages, then makes a netlist in tabular form or by annotating the logic diagram.

During synthesis, the EDA tool will note syntax errors in the VHDL, issue warnings (e.g., Chapter 13, p. 137), and report other useful information like estimates of propagation delays and resource usage.

## 11.4 Logic implementation on FPGAs

After synthesis, we know what resources (LUTs and flip-flops) we will need and how they will be connected. The task of the implementation step is to choose particular LUTs and flip-flops from the thousands available on the target FPGA, (the “place” step), and then to select the wiring paths between them

and from ports to pins, following the XDC file (the "route" step). Placing and routing is frequently an iterative process, to keep the routes short while avoiding congestion (running out of "lanes" on the highway).

Following place-and-route, the designer can see the final netlist in the form of a schematic. For the four-bit counter, the post-implementation schematic looks the same as the post-synthesis schematic, Figure 11.3.2, p. 127, and is omitted here.

The very last implementation step is to incorporate the programming for the LUTs, the switch settings for the routing, and other settings, into a binary file called a **bitstream** that is loaded into the FPGA to accomplish the configuration of the device.

## 11.5 Physical testing

The final step is to physically test the prototype: does the hardware actually work? Does it produce the expected outputs in response to its inputs? In large scale digital designs, creating a testing procedure is a design problem in itself, carried out in parallel with the system design. It is not an afterthought. In this class our designs are quite simple, but it is no less important to think carefully about how to test a design, and then to do the physical testing. No matter how confident you are of your simulations, until you carry out a successful physical test, you do not have a working design.

In order of increasing complexity, here are some of the tools available to you for physical test.

- Voltmeter, for power supply and constant logic levels
- LEDs, for constant and very slowly changing logic levels
- Seven-segment display, for constant and very slowly changing hexadecimal signals
- Toggle switches, for applying constant logic levels.
- Pushbutton switches, for applying momentary logic levels.
- Function generator, for applying high-speed time-varying stimuli, digital and analog
- Terminal emulator, for applying serial text input (see Chapter 18, p. 175)
- Analog oscilloscope, for observing the detailed shape of a digital signal
- Digital oscilloscope, for observing multiple time-varying logic signals at once.

In addition, you can use previously tested subsystems as input or output devices. As you develop a project, always be thinking about how you can use these tools to test it.

FPGA development boards, like the Basys3, often provide a rich set of ports for input and output. For example, the Basys3 has 16 slide switches, five pushbutton switches, sixteen single LEDs, four seven-segment displays, a VGA video port, and four twelve-pin peripheral module (Pmod) sockets. You can use eight of the twelve pins of any available Pmod socket to bring out signals from your design for observation by an oscilloscope. Just route the signal to an entity port and map it to the Pmod pin through the XDC file.

The philosophy we want you to adopt in this course is "top-down design, bottom-up testing". By this, we mean that you think carefully about the design, as a whole, and have a solid paper design (block diagram) before writing any VHDL code. Break the design into smaller, testable modules. Write the VHDL for each module in turn, simulating, implementing, and physically testing, module-by-module. It is never a good idea to write a bunch of VHDL for an entire project and hope it will work. There will be bugs, and by building and testing one small module at a time, you will have the bugs confined to a small space where they are easier to find and correct. Moreover, "modular build-and-test" does not mean "modular code and simulate". Until you carry out a successful physical test, you do not have a working design!

To wrap this chapter up, the following table summarizes the correspondence between the steps of a gate oriented design flow, Section 1.6, p. 15, and the FPGA design flow.

| Step | Gate level flow | FPGA flow |
|---|---|---|
| Capture | Logic diagram | HDL model |
| Verification | Schematic simulator | Testbench |
| Synthesis | Select ICs<br><br>Translate logic diagram to netlist | Infer resources from HDL model<br>Generate netlist |
| Implementation | Place ICs on breadboard<br>Route wiring between ICs | Select specific FPGA resources<br>Determine routing paths<br>Generate bitstream and configure FPGA |
| Verification | Physical test | |

**Figure 11.5.1** Gate level design flow compared with computer-assisted FPGA design flow.

# Chapter 12

# State machines in VHDL

## 12.1 State machines

Structurally, a state machine is very much like a counter: a register to hold the current state and combinational logic to determine the next state, based on the current state and any external inputs. Additionally, there may be combinational logic to assert outputs in particular states, such as the terminal count signal from a counter.

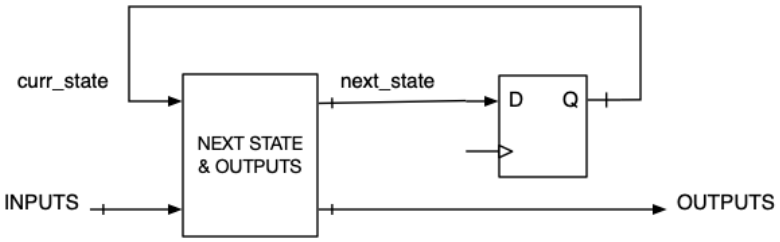



**Figure 12.1.1** A state machine consists of a register to hold the current state and combinational logic to generate the next state.

In a counter, the bits of the state register represent a binary number, and the next state logic increments or decrements the value of the number, or modifies the count sequence based on the inputs. The "state" of a state machine is more general than a numerical count, and the next state logic allows, in principle, for any state to be reached from any other state.

A state diagram is a graphical representation of the sequence of states that enables the designer to step away from the details of the logic and think at a higher level.

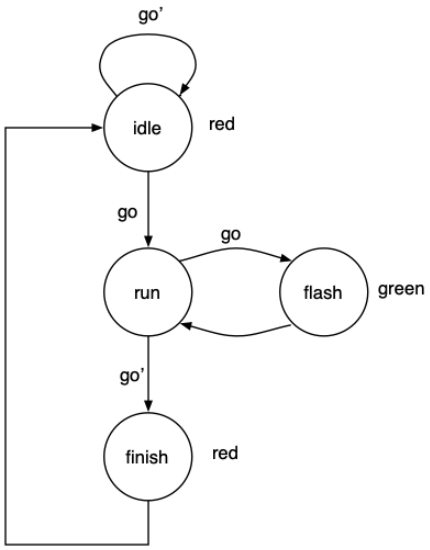



**Figure 12.1.2** A state diagram with one input and two outputs.

In a classical design, each state is assigned a binary code, e.g., idle="00", run="01", flash="11", finish="10". Then, a truth table is derived with the current state and inputs on one side and the next state and the outputs on the other. This truth table is reduced to logic equations for the combinational logic, and the state machine is implemented with gates and flip-flops.

With VHDL, we go directly from the state diagram to a VHDL model, followed by simulation, synthesis, implementation, and physical test.

## 12.2 VHDL template for state machines

A state machine, Figure 12.1.1, p. 131, always has a state register and a block of combinational logic. In VHDL, we will represent these by two processes. A state diagram, e.g., Figure 12.1.2, p. 132, is a very regular structure. Each state contains transitions from that state to other states, which may or may not be conditional on inputs, and outputs that are asserted while in that state. The regularity lends itself to a very regular structure for the two processes in the VHDL model.

```
architecture behavior of example is
    type state_type is (idle, run, flash, finish);  -- 1
    signal curr_state, next_state: state_type;      -- 2

    -- Input go and outputs red and green are declared as
        std_logic.

begin

FSM_reg: process(clk)                                -- 3
begin
    if rising_edge(clk) then
       curr_state <= next_state;
    end if;
end process FSM_reg;

FSM_logic: process(curr_state, go)
begin
    next_state <= curr_state;                       -- 4
    red <= '0';
    green <= '0';

    case curr_state is
       when idle => red <= '1';                     -- 5
          if go = '1' then
             next_state <= run;
          end if;                                   -- 6

       when run =>
          if go = '1' then                          -- 7
             next_state <= flash;
          else
             next_state <= finish;
          end if;

       when flash => green <= '1';                  -- 8
          next_state <= run;

       when finish => red <= '1';
          next_state <= idle;
    end case;                                       -- 9
end process FSM_logic;
end behavior;
```

**Listing 12.2.1** VHDL model derived from the state diagram in Figure 12.1.2, p. 132. Comments are keyed to explanations in the text.

The only new VHDL in the model are noted by comments 1 and 2. Everything else is a construct we have used before.

1. Rather than assign binary codes to each state, VHDL allows us to work with the symbolic names of the states. The line `type state_type is (idle, run, flash, finish);` declares an **enumerated type**. The name of the type is `state_type` (you can call it whatever you want, but this is descriptive). An item declared to be of type `state_type` may take on one of four values: `idle`, `run`, `flash`, and `finish`.

2. In particular, we want two signals, `curr_state` and `next_state`, to be of type `state_type`.

3. The state register process always looks like this, for any state machine.

4. The combinational logic begins by assigning default values to `next_state` and to the outputs, `red` and `green`. These are the values that these signals will take if they are not changed by logic farther down the process. Using defaults simplifies the rest of the logic.

5. Each state gets its own `when` line in the `case` statement. It is conventional first to assert the outputs that are active in the state, and then to write the logic for the next state assignment.

6. Because `next_state` is assigned `curr_state` by default, we don't have to explicitly write the logic for staying in state `idle` if input `go` is false.

7. Outputs `red` and `green` are `'0'` by default, so we don't have to explicitly deassert them in state `run`. There are two paths out of the state. Both are written into the `if-then-else` logic.

8. State `flash` transitions unconditionally to state `run`. Likewise, state `finish` transitions unconditionally to state `idle`.

9. A `when others` is not needed. The only possible values for `curr_state` are accounted for by their own `when` lines.

You can use this state machine as a guide for any other state machine that you write in this course. The VHDL model for a state machine will always have this structure.

```
architecture behavior of my_design is
    type state_type is ( [list of states] );
    signal curr_state, next_state: state_type;
    ...

begin

FSM_reg: process(clk)
begin
    if rising_edge(clk) then
       curr_state <= next_state;
    end if;
end process FSM_reg;

FSM_logic: process(curr_state, [list of inputs])
begin
    next_state <= curr_state;
    -- Assign defaults for all outputs

    case curr_state is
       -- Write a when line for each state
       when [state name] => -- assert outputs
          -- if-then-else logic for next_state assignments

       ...

    end case;
end process FSM_logic;
```

**Listing 12.2.2** Template for writing a state machine in VHDL.

## 12.3 Simulating state machines

The key point in simulating a state machine in VHDL is to write a stimulus process that asserts the inputs so that all state transitions can be tested. The testbench must do this without explicit knowledge of what state the system is in. Its knowledge of the state machine is restricted to the outputs, and its influence on the state machine is solely through the inputs. For example, here is a stimulus process for the state machine in Listing 12.2.1, p. 133.

```
stim_proc: process
begin
    go <= '0';  wait for 4*CLK_PERIOD;  -- Wait in idle
        state for a while

    go <= '1';  wait for 4*CLK_PERIOD;  -- Watch the green
        light flash

    go <= '0';  wait for 4*CLK_PERIOD;  -- Give it time to
        get back to idle

    go <= '1';  wait for 5*CLK_PERIOD;  -- Odd number of
        clock periods this time

    go <= '0';  wait for 4*CLK_PERIOD;  -- Back to the idle
        state

    wait;                                   -- All done
end process stim_proc;
```

Here is a simulation with EDA Playground[1]. EDA Playground does not show the states in a VHDL simulation. Because of this, you are advised to use Vivado instead from now on. It is very difficult to debug a state machine without direct knowledge of what state you are in!

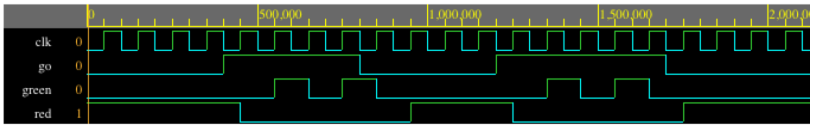



**Figure 12.3.1** Waveform generated by EDA Playground simulation of Listing 12.2.1, p. 133. The waveform does not display the states, only the inputs and outputs. (Right-click "View image" (Firefox) or Right-click "Open image in new window" (Safari) to enlarge the image.)

Here is the same simulation, performed with Vivado. By showing the states as well as the inputs and outputs, it is easier to compare the simulation with the state diagram.

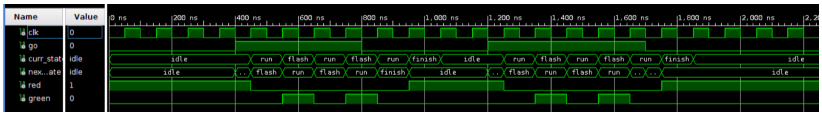



**Figure 12.3.2** Waveform generated by Vivado simulation of Listing 12.2.1, p. 133. The waveform displays the states as well as the inputs and outputs. (Right-click "View image" (Firefox) or Right-click "Open image in new window" (Safari) to enlarge the image.)

[1] www.edaplayground.com/x/26S4

## 12.4 State machine synthesis

When the state machine Listing 12.2.1, p. 133 is passed to the Vivado synthesis tool (Section 11.3, p. 126), a report is generated with interesting details about how the state machine is processed. This report includes a listing of the binary codes assigned to the states:

```
-----------------------------------------------------
   State |     New Encoding |     Previous Encoding
-----------------------------------------------------
    idle |                00 |                   00
     run |                01 |                   01
   flash |                10 |                   10
  finish |                11 |                   11
-----------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register
'curr_state_reg' using encoding 'sequential' in module
'FSM_redgreen'
```

Also, a table of the resources used:

```
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |LUT2 |     3|
|2     |LUT3 |     1|
|3     |FDRE |     2|
+------+-----+------+
```

This table is corroborated by the post-synthesis schematic, Figure 12.4.1, p. 136. The two flip-flops hold the two state bits. Two of the lookup tables (Section 11.1, p. 123) are used for next state logic, and the other two are used to generate the outputs.

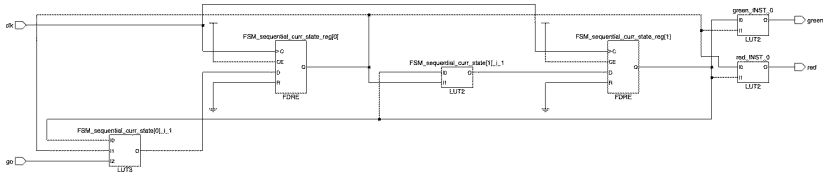
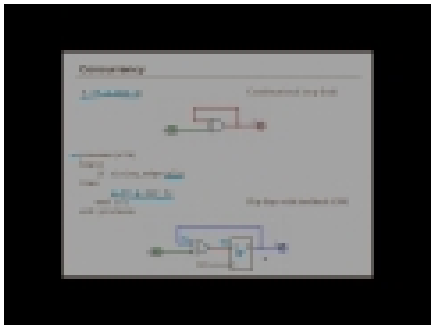



**Figure 12.4.1** Post-synthesis schematic of the state machine in Listing 12.2.1, p. 133.

# Chapter 13

# VHDL modeling pitfalls

This video gives an overview of some of the topics in this chapter.



## 13.1 Multiple drivers

You know that in hardware, an input to a logic circuit (gate, flip-flop, multiplexer, etc) may have only one driver, either a system input port or the output of another logic circuit within the system. If an input is wired with multiple drivers, the drivers will "fight" each other electrically, giving an indeterminate voltage level at the input and/or causing excessive current flow in the drivers leading to a burned-out chip.

In VHDL, as in hardware, multiple drivers are not allowed. As noted earlier, in Section 6.3, p. 84, you cannot have two concurrent statements, or processes, driving the same signal. The following is allowed:

```
process(a, b)
   begin
      y <= a;
      y <= b;
end process;
```

because the two assignments are inside a process, and evaluated in order, with the final value of `y` assigned at the end of the process. But this is not allowed:

```
process(a)
   begin
      y <= a;
end process;

process(b)
   begin
      y <= b;
end process;
```

The two processes concurrently attempt to assign a value to `y`. The simulator will not be able to resolve the conflict and an error will be thrown.

A very common instance of the multiple driver error occurs when resetting a counter:

```
process(a)
   begin
      if a='1' then
        x <= "0000";
        -- other logic
      else
        -- other logic
      end if;
end process;

process(clk)
   begin
      x <= x+1;
end process;
```

The designer's intent here is to use the input `a` to reset the counter and to control other functions. But, by assigning `x` in both processes, `x` has been given two drivers. The correct way to write this is to put the reset function into the counter's process and leave the other logic as-is:

```
process(a)
   begin
      if a='1' then
        -- other logic
      else
        -- other logic
      end if;
end process;

process(clk)
   begin
      if a='1' then
         x <= "0000";
      else
         x <= x+1;
      end if;
end process;
```

Now all the assignments to `x` occur in the same process, and `x` has only one driver.

## 13.2 Combinational loops

When you are programming in a conventional language, you frequently want to update the value of a variable by logically masking it with a bit vector, e.g., `x = x & "00001111";`, or by adding it with another variable or constant, e.g., `x = x + 1`. It is implicit in these expressions that `x` is stored in memory. The update is performed by reading the value of `x` from memory, doing the logical or arithmetic operation, and storing the result back into the same memory location, overwriting the original value. In VHDL, because we are modeling hardware, memory must explicitly be represented by a clocked process. If you forget this (and everyone does, sometime), modeling errors will result.

For example, consider this trivial unclocked process,

```
process(y)
   begin
      y <= not(y);
end process;
```

In a conventional language, this would simply flip the bit `y`. But not in VHDL. An event on `y` activates the process. The bit is flipped, from '0' to '1', say, and the process suspends. But the flipping of `y` reactivates the process, and the bit is flipped from '1' to '0'. This event again reactivates the process, and the bit is flipped from '0' to '1', and...you get the picture. The bit will not flip just once, but will oscillate between '0' and '1'. In hardware, this process models connecting the output of an inverter back to its input. The output will oscillate at a rate determined by the propagation delay of the gate.

This kind of modeling error, in which the output of a combinational element is fed directly back to the input, is called a **combinational loop**. In this course there is no good reason for a combinational loop. It will always lead to a bug, and must always be avoided.

The correct way to replace `y` with its complement is to store it in a flip-flop. The Q output of the flip-flop is passed through an inverter, placing the complement of the bit at the D input of the flip-flop. On the rising clock edge, the value at D is stored at Q. In VHDL,

```
process(clk)
   begin
      if rising_edge(clk) then
         y <= not(y);
      end if;
end process;
```

Often, your design tool will give you a warning when it encounters a combinational loop. It will not throw an error, because combinational loops are not forbidden by the language. Thus, it is incumbent on the designer (that's you) to read and heed the warning messages. If you fail to do so (and everyone does, sometime), a real error may occur farther along, during implementation, or the final hardware will exhibit a bug during physical testing, which is a bad place to discover a modeling error.

Figure 13.2.1, p. 140 shows a variety of combinational loops, and their VHDL models. These are examples to be avoided, not emulated! But it is instructive to study, even simulate, a few of them to observe how they behave. It may help you spot a hardware error in the future.

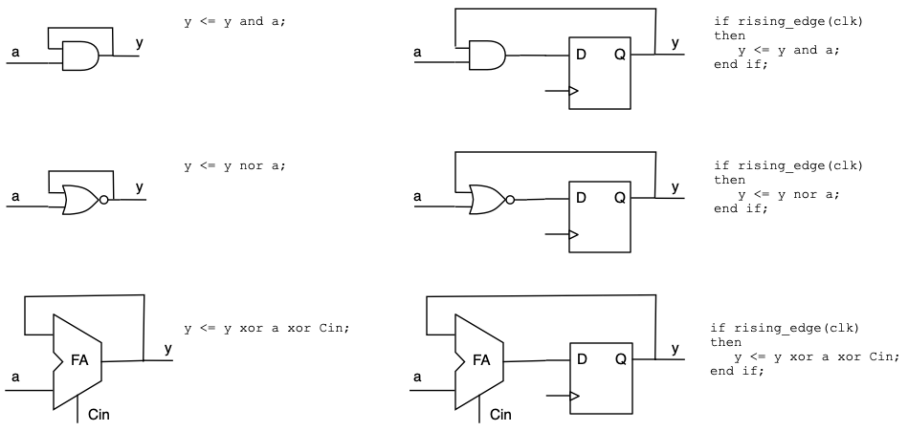



**Figure 13.2.1** Examples of combinational loops (left), and how they are repaired by inserting a storage element (right).

**Checkpoint 13.2.2 Simulation: Combinational loop.** Try simulating, with EDA Playground[1], the combinational loops in Figure 13.2.1, p. 140. Note the addition of a propagation delay in the VHDL model, which makes the simulation more closely resemble what would be observed in hardware. Modify the model to simulate and compare the combinational loop and the loop with flip-flop.

## 13.3 Inferred latches

This section focuses on an all-too-common mistake, when a designer writes an if-then statement and neglects to include the else clause. Here is a simple example:

```
process(a, x)        -- 1
begin
   if a='1' then     -- 2
      y <= x;        -- 3
   end if;
end process;         -- 4
```

**Listing 13.3.1** A VHDL model with an incomplete if-then-else construction. Comments are keyed to notes in the text.

Let's trace its function step-by-step, assuming an initial event on `x`:

1. The event on `x`, a change from '0' to '1' or from '1' to '0', activates the process.
2. The value of `a` is tested.
3. If `a` is '1', this statement is evaluated, and `y` is scheduled to be assigned the value of `x` when the process suspends. Otherwise, this statement is skipped.
4. The process suspends, and `y` is either assigned the value of `x`, or, because there is no `else` clause, `y` is unchanged (implicitly, `y <= y`).

Note that as long as `a` is '1', `y` will follow any changes in `x`, but if `a` changes to '0', the value of `y`holds at whatever its last assignment was. This property of VHDL

[1] `www.edaplayground.com/x/4ULn`

is called **implied memory**. The model behaves somewhat like a flip-flop, but without an edge trigger. It is called a **latch**, **D latch**, or **transparent latch**. We can write the model in a standard form by renaming the signals.

```
latch: process(EN, D)
begin
   if EN='1' then    -- While the latch is enabled,
      Q <= D;        -- the output follows the input
   end if;
end process latch;
```

**Listing 13.3.2** VHDL model for a transparent latch.

The control input is called `EN`, "enable". While the latch is enabled (`EN` is '1'), it is **transparent**, in that the data input, `D`, passes right through to the output, `Q`. When the latch is disabled (`EN` is '0') the latch holds its value.

**Checkpoint 13.3.3** Write a truth table for the transparent latch, with inputs EN and D, and output Q.

**Answer**.

| EN | D | Q |
|---|---|---|
| 0 | — | Q (hold) |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Prior to the invention of edge-triggered flip-flops, latches were used for bit storage. In hardware, a transparent latch requires about half the transistors of an edge-triggered flip-flop. However, edge-triggered flip-flops are easier to design with, and we use then exclusively in this course. The presence of a latch in a design is an error to be avoided.

While you will not intentionally use a latch in a design, they will sneak in if you are not careful when you write if-then-else and case statements, and very often the unexpected presence of a latch in a design will show up as a bug in physical test.

Latches are avoided by remembering always to cover all the alternatives in a decision:

- Make sure all your if-then statements include an else clause. The exception is a clocked process (`if rising_edge(clk) then...end if`), where you intend to have (edge-triggered) storage.
- Make sure your case statements cover all their alternatives. The easiest way to do this is to include `when others => ...` as the last statement in the case. You may either write `when others => null;` which causes no action to be taken, or put in a default action, like `when others => y <= '0';`.

As with combinational loops, an incomplete if-then statement is not a violation of the language, and will not throw an error. However, the synthesis tool will give you a warning stating that a latch has been synthesized for a signal. Make it a habit to always check your synthesis warnings. If a latch warning turns up for some signal, go back through your code and find where the signal is assigned. Figure out why a latch is being synthesized (depending on how complex your code is, it might not be immediately obvious), and correct the problem before proceeding. It is much easier to fix at this stage than to discover it at physical test.

**Example 13.3.4 Incomplete if-then, simulated.** Here is the full VHDL model for an incomplete if-then statement, written like a multiplexer.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity latch_mux is
    Port ( sel : in STD_LOGIC;
           x : in STD_LOGIC;
           y : out STD_LOGIC);
end latch_mux;

architecture Behavioral of latch_mux is
begin

mux: process(sel, x)
begin
    if sel='1' then
        y <= x;
    end if;
end process mux;

end Behavioral;
```

And here is a testbench to exercise its functionality:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity latch_mux_tb is
end latch_mux_tb;

architecture Behavioral of latch_mux_tb is

component latch_mux is
    Port ( sel : in STD_LOGIC;
           x : in STD_LOGIC;
           y : out STD_LOGIC);
end component;

signal sel, x: std_logic := '0';
signal y: std_logic := '0';
constant T: time := 100ns;

begin

dut: latch_mux port map(
    sel => sel,
    x => x,
    y => y );

stim_proc: process
begin
    sel <= '1';
    x <= '0';   wait for T;
    x <= '1';   wait for T;

    sel <= '0';
    x <= '0';   wait for T;
    x <= '1';   wait for T;

    sel <= '1'; x <= '0'; wait for T;
    sel <= '0'; x <= '0'; wait for T;
                x <= '1'; wait for T;
    sel <= '1'; x <= '1'; wait for T;
    sel <= '0'; x <= '0'; wait for T;
                x <= '1'; wait for T;
    wait;
end process stim_proc;

end Behavioral;
```

Running this simulation with the Xilinx Vivado tool resulted in the following waveform. Observe that the system is transparent from `x` to `y` when `sel` is '1', and latched when `sel` is '0'.

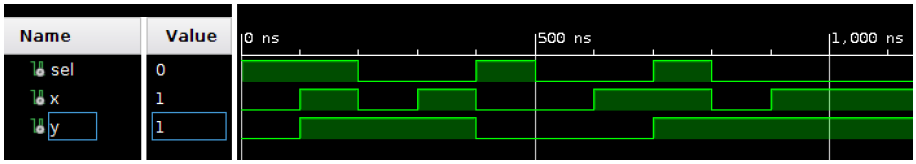
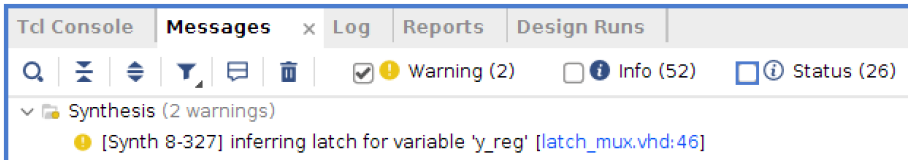


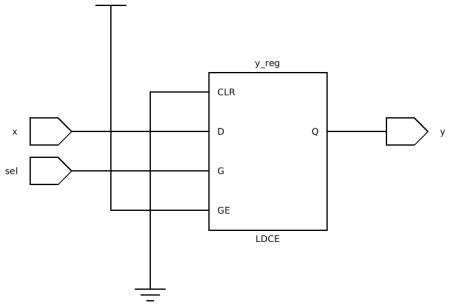

**Figure 13.3.5** Vivado simulation of the incomplete if-then model. Observe the transparent latch action when SEL is alternately '1' and '0'.

□

**Example 13.3.6 Incomplete if-then, synthesized.** The VHDL model in Example 13.3.4, p. 142 was synthesized in Vivado. The following warning was generated:




The warning explains that a latch, denoted `y_reg`, was "inferred" by the synthesis tool from the VHDL model at line 46, which is the if-then statement. The schematic of the synthesized design reveals the same:




A latch element (LDCE) is synthesized, with SEL connected to the enable (G) input, X connected to the data (D) input, and Y connected to the output (Q). The additional inputs, CLR and GE, are explained in the Vivado documentation [21]. □

## 13.4 Gated clocks

See Section 8.1, p. 99.

When modeling any clocked circuit (flip-flop, register, etc), do not include a clock enable (CE) control by writing `if rising_edge(clk) and (CE='1') then...`. While it is logically equivalent to the nested if statements, it implies that the clock is ANDed with the enable signal, making a gated clock. After simulation, when the VHDL model is synthesized to a circuit, an intervening AND gate will cause the flip-flop's clock to be delayed, potentially introducing timing errors.

The correct way to include a clock enable is shown in Listing 8.1.2, p. 100. And, in general, the VHDL model for a synchronous system must always be of the form:

```
if rising_edge(clk) then
   -- Everything else
end if;
```

That said, clock gating is often used in digital systems, especially microcontrollers, to reduce power consumption and extend battery life. When a subsystem is not being used, it can be "put to sleep" by stopping its clock. This requires additional control circuitry to know when to start and stop the clock, and careful design to avoid timing errors between subsystems.

## 13.5 Incomplete sensitivity lists

The simple rules for remembering what signals/ports to put in a process sensitivity list are (Section 9.3, p. 115)

- For clocked logic, the clock signal must be on the sensitivity list. Any inputs inside `if rising_edge(clk) ... end if;` may be omitted from the sensitivity list.
- For unclocked (combinational) logic, all inputs must be on the sensitivity list.

Though these rules are simple, they are often neglected. What can go wrong as a result?

**Simulation.** To illustrate what can happen, consider a very simple process:

```
-- a, b, y are declared std_logic
process(a,b)
begin
  y <= a and b;
end process;
```

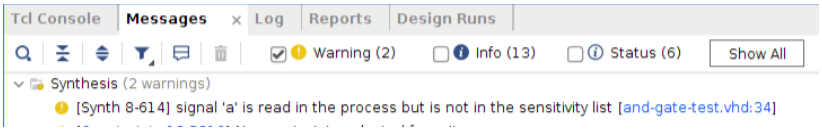
This process was simulated three times: with both `a` and `b` on the sensitivity list, with `a`alone, and with `b` alone. The results are displayed in Figure 13.5.1, p. 145, below.


**Figure 13.5.1** Simulations of an AND gate process. (Top) Inputs a and b are on the sensitivity list. (Middle) Input a, alone, on the sensitivity list. (Bottom) Input b, alone, on the sensitivity list.

In the first case, the waveforms show `y` is '1' only when `a` and `b` are '1', as expected. In the second case, when input `b` changes from '1' to '0' at 300ns, the output remains high. Since the process is insensitive to `b`, it does not activate on this event and the output holds at its previous value, which was set at 200ns when `a` changed from '0' to '1'. In the third case, because the process is insensitive to `a` and the output does not change to '1' at 200ns. The output is always 0.

Leaving inputs off the sensitivity list causes incorrect and misleading simulation results.

**Synthesis.** Continuing with the AND gate toy problem, the Vivado synthesis tool detects the missing input on the sensitivity list and throws a warning.




However, Vivado's synthesis tool assumes that the omission was accidental, and adds missing signals to the sensitivity list ([20], p.195). Consequently, for the AND gate example, all three cases synthesize to the same logic, Figure 13.5.2, p. 146.

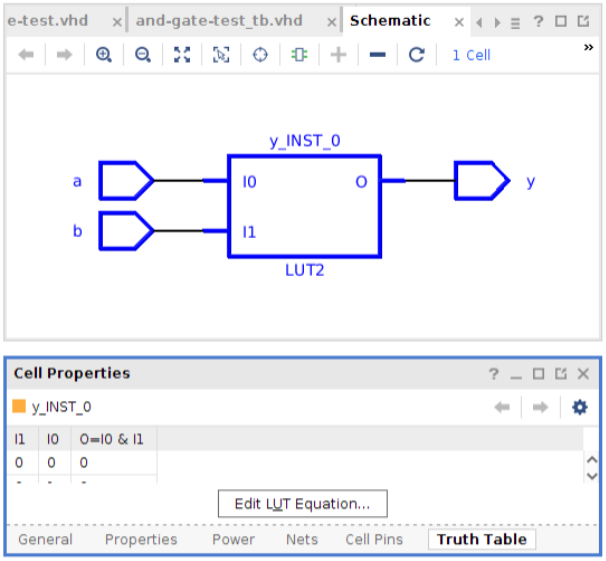



**Figure 13.5.2** Synthesis of AND gate process. The logic equation for the LUT is "O = I0 & I1" even if one of the inputs is left off the sensitivity list.

This is the behavior of the Vivado tool. Other synthesis tools may respond differently, so always be careful to read the synthesis warnings and correct any incomplete sensitivity lists. Omissions will, at the least, result in incorrect simulations, and could result in incorrect logic as well.

## 13.6 Summary

You will avoid a lot of debugging if you are careful to follow these VHDL guidelines:

1. Always read the synthesis warnings and correct the problems they point to.
2. Multiple drivers: Keep all assignments for a signal inside one process. Do not assign a signal in more than one process.
3. Combinational loops: Rewrite them, or insert storage.
4. Inferred latches: Make sure that every if-then has an else, and every case statement covers all possibilities or has a default ("when others =>").

5. Never gate the clock.
6. Sensitivity lists: be sure to follow the rules, and obey the synthesis warnings.

# Chapter 14

# Scalable VHDL models

## 14.1 Parametrized designs

The VHDL model for a counter depends on two **parameters**: the width of the register and the maximum count value. For example, in a decade counter process that is part of an architecture:

```
architecture behavior of example is
    signal uy: unsigned(3 downto 0) := "0000";   -- 1, 2
begin
    process(clk)
    begin
       if rising_edge(clk) then
          if uy < 9 then                         -- 3
             uy <= uy+1;
          else
             uy <= "0000";                       -- 4
          end if;
       end if;
    end process;
...
end behavior;
```

There are four points in this model, marked by comments, where constant values are used:

1. The length of the counter register (4 bits, indexed `3 downto 0`)
2. The initial state of the counter, `"0000"`
3. The maximum value of the counter, `9`
4. The reset value, `"0000"`

All four of these numerical constants must be entered correctly in order for the counter to work as desired. If, at some future time, you want to change the counter from counting to 9 to counting to 11, you will have to change one number. If you want to change the maximum count to 23, you must modify all four points in the model: the length of `uy` from 4 to 5, the initial and reset values from `"0000"` to `"00000"`, and the upper count limit from 9 to 23. If you miss one of these changes you'll have a syntax error or a functional bug.

You know from prior programming experience that it is dangerous to have unnamed constants, sometimes called "magic numbers", in your code. They often go undocumented and when changes have to be made, they are often

overlooked, leading to costly rounds of debugging. These risks are mitigated by replacing the magic numbers with documented named constants. Here is how that would be done in the decade counter:

```
architecture behavior of example is
    constant REGSIZE: integer := 4;  -- register length
    constant MAXVAL: integer := 9;   -- maximum count

    signal uy: unsigned(REGSIZE-1 downto 0) :=
        (others=>'0');
begin
    process(clk)
    begin
       if rising_edge(clk) then
          if uy < MAXVAL then
             uy <= uy+1;
          else
             uy <= (others=>'0');
          end if;
       end if;
    end process;
end behavior;
```

In this version, the register length and the maximum count parameters are assigned to named constants, REGSIZE and MAXVAL at the beginning of the architecture. These named parameters replace the instances of the magic numbers in the model. Additionally, we have used the `(others=>'0')` construction to replace the `"0000"`. This represents a string "0...0" of length that matches the vector it is assigned to. With these modifications, if you want to change the maximum count from 9 to 23, you only have to change REGSIZE and MAXVAL, and their values will propagate through the model. These two changes make the code easier to read, thus less prone to bugs, and also **scalable**, by changing the parameter values rather than by rewriting the code. Large designs often contain many registers of the same size. Making the register size a named constant parameter, and using that parameter throughout the design, makes it easy to scale the design to different register sizes in the future.

## 14.2 Computed parameters

Referring again to the counter, parameters REGSIZE and MAXVAL are closely connected, since MAXVAL cannot exceed the largest number held in the register, which is $2^{\text{REGSIZE}}-1$. Solving this inequality for REGSIZE, we obtain REGSIZE $\geq \log_2(\text{MAXVAL}+1)$. Thus, if we begin with MAXVAL, we can calculate the minimum required REGSIZE by rounding $\log_2(\text{MAXVAL}+1)$ up to the next integer. For example, for MAXVAL=9, we calculate $\log_2(10) = 3.32$; rounding up gives 4, as expected. For MAXVAL=23, $\log_2(24) = 4.59$, which rounds up to 5.

VHDL has a library, **ieee.math_real**, that enables us to make this calculation in the architecture. The functions in this library are not synthesizable in hardware, but they can be used to do arithmetic on parameters during simulation or synthesis. The math_real library can also be used to compute more complex stimuli, such as sine waves, in a testbench. All the available functions in the math_real library are helpfully described in References [1], pp 443ff, and [15].

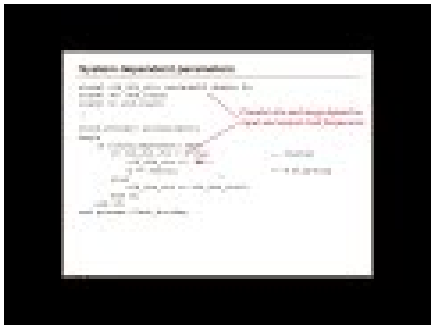
Here is how to write the calculation of REGSIZE from MAXVAL:

```
--  Include the math_real library
    use ieee.math_real.all;

--  In the architecture
architecture behavior of example is
    constant MAXVAL: integer := 9;   -- maximum count
    constant REGSIZE: integer := integer( ceil( log2 (
        real(MAXVAL+1) )));
    ...
```

Unpacking the expression `integer( ceil( log2 ( real(MAXVAL+1) )))` from the inside out,

- `real(MAXVAL+1)` casts the integer value MAXVAL+1 to type real.
- `log2( ... )` calculates the base-2 logarithm.
- `ceil( ... )`, "ceiling", rounds the base-2 logarithm up to the next integer value.
- `integer( ... )` converts the result to match the integer type of REGSIZE.

This video presents another common example of a parametrized architecture:



## 14.3 Counters: integer or unsigned?

Another way to make a scalable counter is to use a signal of integer type rather than unsigned. The parametrized decade counter looks like this:

```
library IEEE;
use IEEE.std_logic_1164.all;                      -- 1

...

architecture behavior of example is
    constant MAXVAL: integer := 9;
    signal iy: integer range 0 to MAXVAL := 0;  -- 2, 3

begin
    process(clk)
    begin
       if rising_edge(clk) then
          if iy < MAXVAL then
             iy <= uy+1;
          else
             iy <= 0;                             -- 3
          end if;
       end if;
    end process;
end behavior;
```

**Listing 14.3.1** Decade counter using integer datatype. Comments are keyed to explanations in the text.

The differences from the unsigned counter, indicated by comments, are:

1. The numeric_std library is not needed. The integer data type and all its arithmetic operations are part of original VHDL.

2. The count is declared to be of integer type. The size of the integer is specified by its numeric range rather than by a number of bits.

3. The initial value and reset value of `iy` are the integer constant `0` rather than the bit string `"0000"`.

The integer counter is easily scaled by changing the value of MAXVAL, without the complication of calculating the size of a register. An integer counter can, in principle, perform the same tasks as an unsigned counter. However, there are reasons to prefer one over the other in particular situations:

- If a counter is used as a timer or clock divider, where the only output is a terminal count signal, then the integer counter is preferable.

- If the counter's value is to be used only as an array index, e.g., generating sequential memory addresses, then the integer counter is preferable. Array indices are also of integer type and no type conversion is required. An unsigned count will have to be converted to integer type to be used as an index: `to_integer(uy)`.

- If the actual value of the count is needed, e.g., to display the time in a stopwatch, then the integer count must be converted to a std_logic_vector with an explicit length: `std_logic_vector(to_unsigned(iy, REGSIZE))`. In this case, it seems cleaner to hold the count in an unsigned vector, with size calculated from the maximum value as shown earlier.

If you are using an integer counter you must be careful not to omit the range specification in the declaration of the count signal. By default, a signal of integer type is 32 bits long, and without a range specification it will be synthesized as a 32-bit register, likely wasting resources. With the range restriction, when the

counter is synthesized, `iy` will be assigned a register of length appropriate to hold MAXVAL.

It is also important to note that unsigned and integer counters behave differently in simulation. An integer counter with range 0 to 9 will not naturally roll over from 9 to 0, but will throw an "out of range" error when it tries to increment from 9 to 10. A four-bit unsigned counter will naturally increment from 9 to 10 and roll over from 15 to 0 if the count is not compared to the maximum value. This only applies to simulation. In synthesis, both counters will be synthesized with four-bit registers and, in the absence of an explicit maximum value check, both will naturally roll over from 15 to 0.

The uses and inter-relationships of the integer, signed/unsigned, and std_logic_vector types are illustrated in the graphic shown in Figure 14.3.2, p. 153.

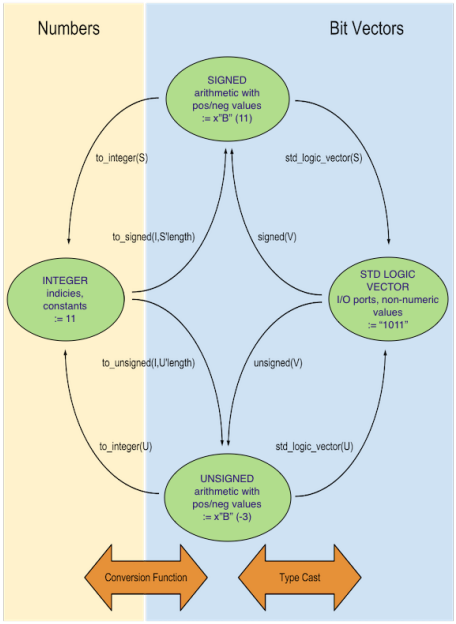



**Figure 14.3.2** Relationships among integer, signed/unsigned, and std_logic_vector data types [22].

**Checkpoint 14.3.3 Integer counter with TC output.** Modify the integer counter, Listing 14.3.1, p. 152, to generate a terminal count (TC) output. Verify your design by simulation. Here is a design and testbench[1], in EDA Playground, to get you started.

**Checkpoint 14.3.4 Integer counter vs unsigned.** Using Xilinx Vivado, write complete VHDL models and testbenches for the integer and unsigned decade counters. Compare their behavior in simulation. Run both models through synthesis and compare the resource usage. Remove the range constraint from the integer counter declaration and observe the effect on simulation and synthesis.

[1] `www.edaplayground.com/x/24AE`

# Part III

# Components and Subsystems

# Chapter 15

# Subsystem design

In Section 1.2, p. 6 we identified the four principle subdivisions of all digital processors:

1. The datapath, consisting of:
   (a) Memory: Where data is stored during computation
   (b) Operations: Arithmetic and logical manipulations of the data
2. The controller, which coordinates the operations of the datapath
3. Input/Output (I/O), which connect the processor to the outside world via sensors and actuators.

Memory functions are handled by flip-flops, registers, and other elements we haven't seen yet. Arithmetic and logical operations include multiplexers, decoders, and adders. Registers and incrementers are combined into counters, which are used in a variety of ways. The heart of any controller is a state machine. I/O functions, so far, are restricted to simple switch inputs and LED outputs.

We will now begin to assemble these blocks into subsystems that perform more complex operations, and finally assemble subsystems into complete system designs. The distinction between a subsystem and a full system can be a little fuzzy, but for our purposes, we will use **subsystem** to mean something that performs a single function as part of a larger whole. Examples include: a multi-digit seven-segment display decoder; a subsystem that displays data on a video screen; and a keypad interface that converts button presses to usable data. None of these would be regarded as a complete processor, but all could be important parts of one.

This chapter begins conceptually, with some new design principles, and ends with some new VHDL constructs that facilitate the construction of processors from subsystems.

## 15.1 System architecture

The structure of a large digital system is shown in Figure 15.1.1, p. 158.

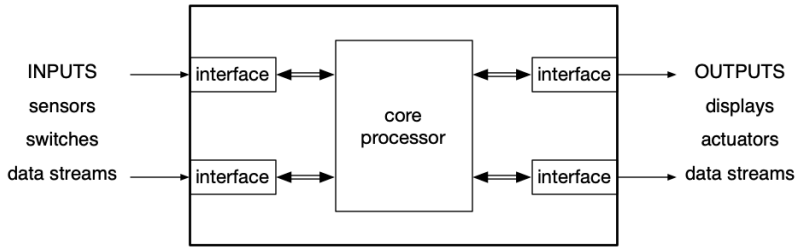



**Figure 15.1.1** A large digital system has a core processor and input/output subsystems that interface the processor to peripheral devices for input and output.

Signals from external sources (sensors, switches, data streams like WiFi or Bluetooth) supply data to the core processor, which performs logical and arithmetic operations on the data. The results flow to output devices (displays, actuators, data streams).

At the beginning of a system design process, the requirements for the system are identified -- what is purpose of the system, what are its inputs and outputs, how must input data be processed to produce output data? Separating the I/O from the core processor enables modular design, construction, and testing.

Input and output devices may be quite simple, like a pushbutton or an LED. These are either ON or OFF, and their connections to the core processor can be simple and direct. Other inputs and outputs are more complex -- e.g., analog-to-digital and digital-to-analog converters, GPS receivers, video displays, motors. Some sensors have analog outputs, others are digital, and digital sources output their data in different formats. An **interface** takes the data as it comes from the input device and communicates it to the core processor in a consistent way. Likewise, on the output side, a motor is very different from a video display, and the interface adapts the data as it comes from the processor to the requirements of the output device. Once an interface for a particular device has been designed and tested, it may be reused in a future system design.

## 15.2 Register transfer level (RTL)

As a general rule, any digital processor will have a datapath and a controller Section 1.2, p. 6. This is certainly true of the core processor for a large system, and many subsystems and I/O interfaces are themselves processing units with their own datapaths and controllers.

In this class we adopt a common datapath methodology called **register transfer level (RTL)** design. The layout of an RTL module is shown in Figure 15.2.1, p. 159. This is a good design pattern to follow for any module of a digital design. Beginning at the boundary of the module, input data is received and stored in an input register. The module's output data is stored in an output register. Between these registers, the module's datapath processes the input data and produces the output data. The input, output, and datapath registers are controlled by signals generated by the module's controller, which is a state machine. The controller also receives control signals from other modules that source data, and outputs control signals to modules that receive data. Like most general block diagrams, this seems complicated at first, but will become clear as we work through several concrete examples in subsequent chapters.

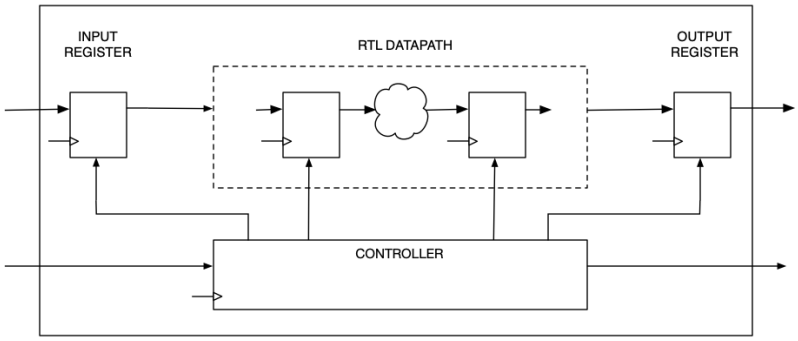



**Figure 15.2.1** RTL module. Input and output data are stored in registers. The datapath, which processes the input data into output data, consists of one or more register transfer operations. A register transfer itself has one or more input registers, a block of combinational logic that operates on the data, and a register to store the result. The controller includes a state machine that asserts enable inputs for the registers, and also communicates outside the module via control signals.

The functionality of the datapath is organized as a set of **register transfers**, each consisting of one or more source registers, a block of combinational logic that operates on the data, and a destination register to store the result, Figure 15.2.2, p. 159. Registers in an RTL design almost always are equipped with enable inputs to control whether the register updates, performing the operation, or holds. Optionally, they may also have clear inputs.

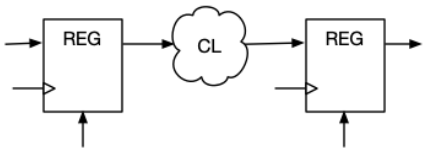



**Figure 15.2.2** The basic register transfer operation consists of one or more source registers, a block of combinational logic (CL) to operate on the data, and a destination register. Load enable lines for the registers connect to the controller.

To make this a little more concrete, Figure 15.2.3, p. 160 shows three common register transfer operations.

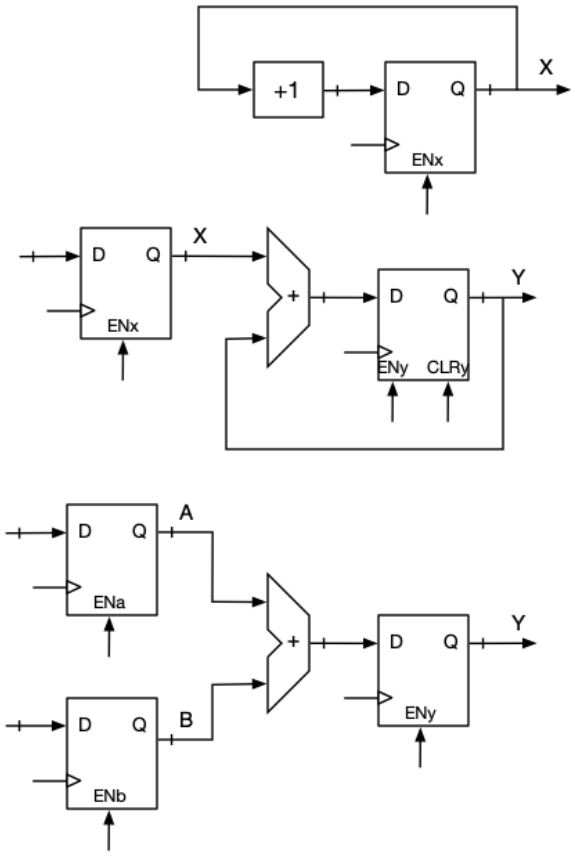



**Figure 15.2.3** Some register transfer operations. (Top) Incrementer, `X <= X+1`; (Middle) Accumulator, `Y <= Y+X`; (Bottom) Addition, `Y <= A+B`.

In the incrementer, the source and destination are the same register. The combinational logic adds 1 to X, and the result is written back into X on the rising edge of the clock if ENx is TRUE. The VHDL model for this RTL is:

```
if rising_edge(clk) then
   if ENx = '1' then
      X <= X+1;
   end if;
end if;
```

In the accumulator, the X and Y registers are both sources, and the Y register is also the destination. The combinational logic is an adder, calculating the sum of X and Y, which is then loaded back into Y on the rising clock edge if ENy is TRUE. If values $x_1, x_2, ...$ are sequentially stored in X, after N clock cycles the Y register holds the accumulated sum $x_1 + x_2 + ... + x_N$. The Y register can be cleared as well as enabled; as usual, we assume that in the physical register, clear has priority over enable. The VHDL model for this RTL is:

```
if rising_edge(clk) then
   if CLRy = '1' then
      Y <= (others=>'0');
   elsif ENy = '1' then
      Y <= Y + X;
   end if;
end if;
```

Loading the X register is a separate register transfer, controlled by the ENx input.

In the third example, there are two source registers, A and B, and one destination register, Y. Again, the combinational logic is an adder, calculating the sum of A and B. The sum is stored back into Y on the rising edge of the clock, if ENy is TRUE. The VHDL model for this RTL is:

```
if rising_edge(clk) then
   if ENy = '1' then
      Y <= A + B;
   end if;
end if;
```

Loading A and B are also register transfers, controlled by ENa and ENb, respectively.

The control inputs -- enable and clear -- for the registers are connected to the controller. The state machine is then designed so that it asserts the control signals in the right order to load the source and destination registers and correctly sequence the datapath's RTL operations.

Again, don't worry if this feels very abstract at this point. As we go through different subsystem designs in the coming chapters, watch for the register transfer operations and how the controllers are designed. Your understanding will increase as you see examples and also begin to design your own subsystems in lab.

## 15.3 Structural VHDL

When you make a breadboarded prototype with discrete integrated circuits Section 1.6, p. 15, each integrated circuit is a self-contained and reusable unit; using VHDL vocabulary, each IC has its own entity and architecture. The ICs are placed on the breadboard and connected together with wires. There is an analogous feature in VHDL for developing reusable modules and connecting them together. This is called **structural VHDL**.

We'll illustrate the idea with a simple, almost trivial, example. We have previously seen that a full adder implements the addition operation for single bits. The VHDL model for a full adder looks like this:

```
library ieee;
use ieee.std_logic_1164.all;

entity full_addder is
   port( a, b, cin: in  std_logic;
         sum, cout: out std_logic );
end full_adder;

architecture gate_level of full_adder is
begin
   sum <= a xor b xor cin;
   cout <= (a and b) or (a and cin) or (b and cin);
end gate_level;
```

**Listing 15.3.1** VHDL model for a full adder.

A multi-bit, say four-bit, adder is constructed by chaining together, or cascading, full-adder modules. You may recognize the syntax from testbenches (e.g., Listing 7.2.1, p. 93).

```
library ieee;
use ieee.std_logic_1164.all;

entity adder4 is
    port( X, Y: in   std_logic_vector(3 downto 0);
          Cin : in   std_logic;
          Z:    out  std_logic_vector(3 downto 0);
          Cout: out  std_logic);
end adder4;

architecture structural of adder4 is

component full_adder                                -- 1
   port( a, b, cin: in  std_logic;
         sum, cout: out std_logic );
   end component;

   signal c0, c1, c2: std_logic;                    -- 2

begin

bit0: full_adder port map(                          -- 3
         a => X(0),
         b => Y(0),
         cin => Cin,                                -- 4
         sum => Z(0),
         cout => c0 );                              -- 5

bit1: full_adder port map(                          -- 6
         a => X(1), b => Y(1), cin => c0,           -- 5
         sum => Z(1), cout => c1 );

bit2: full_adder port map(
         a => X(2), b => Y(2), cin => c1,
         sum => Z(2), cout => c2 );

bit3: full_adder port map(
         a => X(3), b => Y(3), cin => c2,
         sum => Z(3), cout => Cout );               -- 4

end structural;
```

**Listing 15.3.2** Structural VHDL model of a four-bit adder. Comments are keyed to explanations in the text.

The top-level entity of the four-bit adder is what one would expect: two four-bit inputs X and Y, a four-bit output Z, and single-bit carry-in and carry-out ports. The structural VHDL features of the architecture are marked by the comments:

1. The full adder, previously described by its own entity and architecture, is declared as a **component** within the four-bit adder's architecture. This is identical to the way a unit under test is declared in a testbench. The component declaration matches the full adder's entity block.

2. Signals `c0, c1, c2` will wire the carry chain between the full adders.

3. The four-bit adder is constructed from four **instances** of the full_adder component. This is like pulling four full-adder ICs out of a parts drawer. It is always a good idea to give each instance a descriptive label (here,

`bit0`, etc) as an aid to the reader. The **port map** of the instance specifies how the component is wired up. Here, `a => X(0)` says that the `a` port of the full adder is wired to bit 0 of the top-level `X` port.

4. Very often, a component port and a mapped signal will have the same name. The `cin` (carry in) port of full adder `bit0` is wired to the `Cin` port of the top-level entity, and the `cout` (carry out) port of full adder `bit3` is wired to the `Cout` port of the top-level entity. To avoid errors, remember the syntax: `component port => top-level port, or signal`.

5. Wiring between components is always done with declared signals. You cannot directly map port-to-port between components. The desired connection of `bit0`'s carry-out port to `bit1`'s carry-in port is accomplished by mapping port `cout` of `bit0` and port `cin` of `bit1` to the same signal, `c0`.

6. If it does not compromise clarity, multiple port mappings can be put on the same line to save space, as shown for three of the full adder instances.

Using the numeric_std library, a four-bit adder can be modeled much more simply, and implemented more efficiently, on an FPGA. This is why I said the example is "almost trivial". Where structural VHDL really shines is in a modular design as diagrammed in Figure 15.1.1, p. 158. There, each of the interfaces, and the core processor, may be separately designed and validated as components, and then instantiated and wired together in a top-level structural model. The modules are also available for reuse in a subsequent design. This is a very good way to plan and execute a large project.

**Checkpoint 15.3.3** Write a behavioral architecture, using the numeric_std library, for the four-bit adder in Listing 15.3.2, p. 162. You can do it with a single combinational process.

**Hint**. When you add two four-bit numbers, with or without a carry-in, you must allow five bits for the result. The `Cout` bit is bit 4 of this result.

## 15.4 Parametrized components: generics

(∗)

# Chapter 16

# Multiplexed seven-segment display

## 16.1 Seven-segment displays

A popular way to display numerals uses seven LED-illuminated segments, arranged as shown in Figure 16.1.1, p. 165. The segments are labeled $a$ through $g$. The display can easily show the decimal numerals 0 through 9, and with a little imagination, the additional hexadecimal characters A through F (as A, b, c, d, E, F). A display may also include a decimal point as an eighth "segment".


**Figure 16.1.1** A single-digit seven-segment display can show 0-9 and A-F (as A, b, c, d, E, F).

The signals to drive the segments are produced by a special 4-to-7 decoder that follows the truth table shown in Figure 16.1.2, p. 165.

| **x** | $(x_3\ x_2\ x_1\ x_0)$ | $(a\ b\ c\ d\ e\ f\ g)$ | **x** | $(x_3\ x_2\ x_1\ x_0)$ | $(a\ b\ c\ d\ e\ f\ g)$ |
|---|---|---|---|---|---|
| 0 | 0 0 0 0 | 1 1 1 1 1 1 0 | 8 | 1 0 0 0 | 1 1 1 1 1 1 1 |
| 1 | 0 0 0 1 | 0 1 1 0 0 0 0 | 9 | 1 0 0 1 | 1 1 1 0 0 1 1 |
| 2 | 0 0 1 0 | 1 1 0 1 1 0 1 | 10($A$) | 1 0 1 0 | 1 1 1 0 1 1 1 |
| 3 | 0 0 1 1 | 1 1 1 1 0 0 1 | 11($b$) | 1 0 1 1 | 0 0 1 1 1 1 1 |
| 4 | 0 1 0 0 | 0 1 1 0 0 1 1 | 12($C$) | 1 1 0 0 | 1 0 0 1 1 1 0 |
| 5 | 0 1 0 1 | 1 0 1 1 0 1 1 | 13($d$) | 1 1 0 1 | 0 1 1 1 1 0 1 |
| 6 | 0 1 1 0 | 0 0 1 1 1 1 1 | 14($E$) | 1 1 1 0 | 1 0 0 1 1 1 1 |
| 7 | 0 1 1 1 | 1 1 1 0 0 0 0 | 15($F$) | 1 1 1 1 | 1 0 0 0 1 1 1 |

**Figure 16.1.2** Truth table for hexadecimal to seven-segment decoder. A TRUE (1) output, $a - g$, activates the respective segment.

Inside the display, the LEDs are wired in one of two ways. In a **common-cathode** display, the cathodes are wired together. There is a single common cathode connection for all the segments, and they have separate anode con-

nections. The cathode terminal is wired to ground, and a segment is turned on by driving its anode HIGH, through a current-limiting resistor, sourcing current through the LED. In a **common-anode** display, on the other hand, the anodes are wired together and the segments have separate cathode connections. The anode terminal is wired to Vcc, and a segment is turned on by driving its cathode LOW through a current-limiting resistor, sinking current from Vcc through the LED (Figure 16.1.3, p. 166).

For a common-cathode display, the segment signals $a - g$ in the truth table, Figure 16.1.2, p. 165, drive the anodes with a **high true** voltage convention, meaning that a logical TRUE (1) is represented electrically by a HIGH voltage. For a common-anode display, the cathodes must be driven low to activate the segments. The same truth table applies, but the segment signals are **low true**, meaning that a logical TRUE (1) is represented electrically by a LOW voltage. Rather than rewrite the truth table swapping 1s and 0s, it is preferred to use the same truth table and just run $a - g$ through NOT gates to the cathodes. In this way, '1' and '0' still mean TRUE and FALSE, and the inverter is interpreted as changing the sense of the signal from high true to low true.

Often, the current requirements of the LEDs, particularly the common connections, exceed the source/sink capacity of a logic circuit, and additional buffers or transistor switches with higher current capability are used between the logic and display.

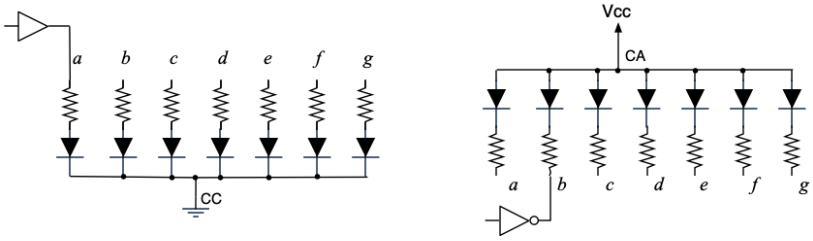



**Figure 16.1.3** (Left) A common-cathode seven-segment display. Segment $a$ is shown being activated by driving its anode HIGH through a current-limiting resistor. (Right) A common-anode seven-segment display. Segment $b$ is shown being activated by driving its cathode LOW, through a current-limiting resistor.

**Checkpoint 16.1.4** Why should each segment have its own current-limiting resistor --- why not wire one current-limiting resistor between the common terminal and Vcc or ground?

**Hint**. How much current can flow through the current-limiting resistor (Ohm's Law)? What will you see if this current is divided among all the active LEDs?

**Checkpoint 16.1.5** Write a VHDL process, using a `case` statement, that models the seven-segment decoder's truth table.

## 16.2 Multi-digit display

A single display has eight or nine terminals---seven segments, maybe a decimal point, and a common. The Basys3 board has a four-digit display, but each segment does not have its own terminal (which would use up 28 or 32 of the FPGA's pins). Rather, the four digits are wired together as shown in Figure 16.2.1, p. 167. Each digit has its own common anode terminal. The cathodes of the four A segments are wired together to a common "A" terminal. Likewise, the cathodes of the B segments are wired together to a common "B" terminal, and so forth. The C segment of Digit 2, say, is activated by pulling

the Digit 2 anode terminal (pin 9) HIGH, and pulling the C segment terminal (pin 4) low.

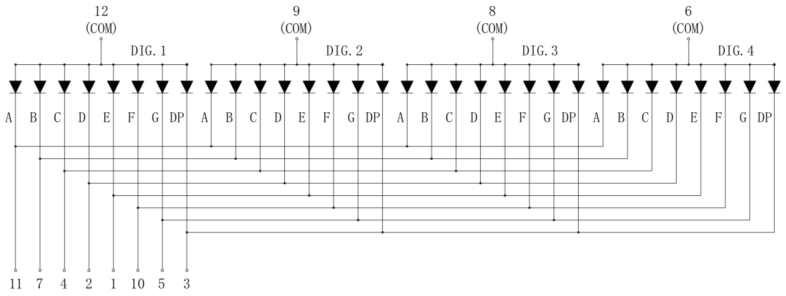



**Figure 16.2.1** The four-digit display on the Basys3 (Luckylight KW4-281ASB). Each digit has a common anode. The cathodes of all the A segments are wired together, and similarly for segments B through G and the decimal points. A single segment is activated by pulling its digit's anode high, and pulling its cathode low. (Reference: Luckylight Electronics)

The wiring of the display to the FPGA is shown in Figure 16.2.2, p. 167. The cathode terminals are wired through 100-ohm current limiting resistors directly to pins on the FPGA (not shown). These wires are labeled CA-CG and DP. With the current-limiting resistor, the current draw of a segment is within the capacity of the FPGA's output. The anode terminals (A1-A4), on the other hand, must be able to carry enough current for the entire digit, which is 8 times the current of a single segment. This is too much current for an FPGA pin to source directly, and so transistor drivers (Q1A, Q1B, Q2A, Q2B) with greater current-carrying capacity are used. The wires labeled AN0-AN3 are wired to the FPGA. When one of them, say AN0, is pulled LOW, its transistor is turned on, and current flows from the 3.3V power supply, denoted VCC3V3, into Pin 6, the digit's common anode terminal (Digit 4, as it turns out; the labeling isn't perfectly consistent).

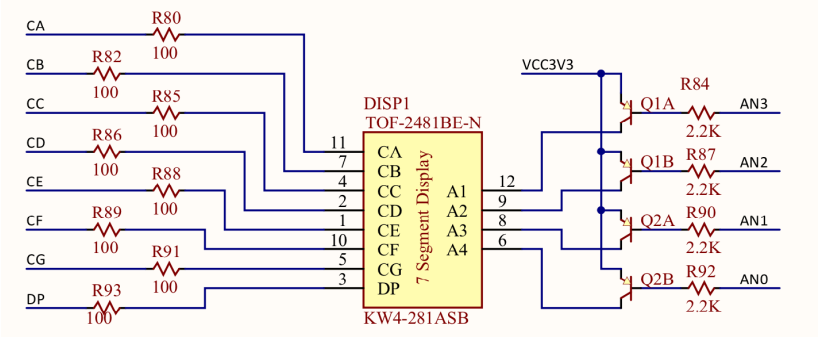



**Figure 16.2.2** How the seven-segment display module is wired to the FPGA on the Basys3. The cathodes (CA-CG and DP) are connected directly to pins on the FPGA, through 100-ohm current-limiting resistors. The common anode terminals are wired to the 3.3V power supply, VCC3V3, through transistor drivers (Q1A, Q1B, Q2A, Q2B). Pulling ANx low turns the respective transistor on, and current flows from VCC3V3 into the digit's anodes. (Reference: Digilent)

This explains how the display works, electrically. We will need some logic to display a four-digit number.

## 16.3 Multiplexed display driver

A number is displayed on a digit by pulling one of AN0-AN3 LOW, and also pulling appropriate cathodes LOW. For example, to display the number 3 on Digit 2, pull AN2 LOW, and pull cathodes CA, CB, CC, CD, and CG LOW.

A four-digit number is displayed one digit at a time. Each digit is four bits, hexadecimal 0 through F. The logic selects one of the four-bit numbers in turn, converts it to seven segments with a combinational hex-to-seven-segment decoder, which is connected to the display, CA-CG. In synchrony with the digit selection, the appropriate anode (AN0-AN3) is activated. Then the next number is selected and displayed, and the next, cycling through the four numbers rapidly enough that the flashing digits appear to be on simultaneously. Sixty hertz is a good cycle rate to avoid perceptible flicker in the entire display. The display must move from digit to digit at four times that rate, or 240 Hz. This is demonstrated in the video, Figure 16.3.1, p. 168.

**Figure 16.3.1** Demonstrating a multiplexed seven-segment display at four sweep rates: 1, 10, 20, and 60 sweeps/second (Hz).

A **multiplexed seven-segment display** driver is a useful component to have in your personal parts library. You can pull it out and add it to a design whenever needed.

### Exercises

1. Following the ideas described above, design the block diagram for a four-digit multiplexed display driver. Your available components are a single-digit decoder, counter, one-hot decoder, and multiplexer(s).
2. Write a VHDL model (entity and architecture) for a multiplexed dislay driver component.
3. Modify your block diagram and VHDL model with an additional control signal, BLANK. When BLANK is asserted, leading zeros are suppressed, i.e., "0070" is displayed as " 70".

# Chapter 17

# Shift registers

## 17.1 Shift register logic

As we have seen, a register is a set of flip-flops for holding a binary vector. The flip-flops in the register frequently share common enable and reset controls, as shown in Figure 17.1.1, p. 169.

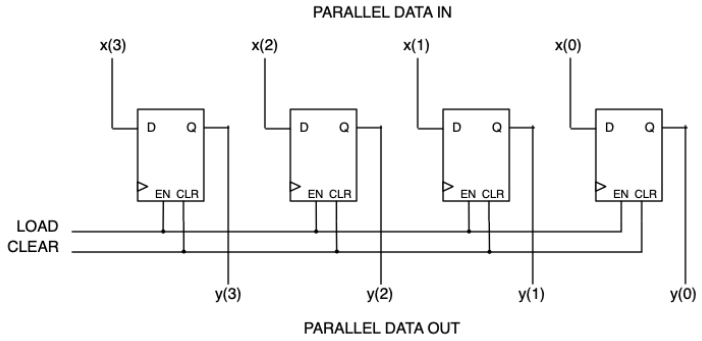



**Figure 17.1.1** A four-bit parallel-in, parallel-out register. The register loads on the rising clock edge when LOAD is asserted, and clears to zero when CLEAR is asserted. CLEAR has priority over LOAD. If LOAD and CLEAR are both FALSE, the register holds its contents.

A different kind of register is created by chaining flip-flops together, as shown in Figure 17.1.2, p. 169.

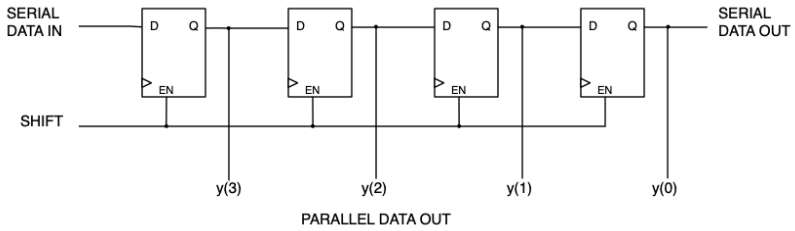



**Figure 17.1.2** A four-bit serial-in, parallel-out shift register. The bits in the register are shifted right by one flip-flop on each rising clock edge, while SHIFT is asserted. If SHIFT is FALSE, the register holds its contents.

When SHIFT is asserted, the flip-flops are all enabled to load. On the rising clock edge, then, the SERIAL DATA IN signal is loaded into the first flip-flop

(y(0)), the contents of the first flip-flop are loaded into the second flip-flop (y(1)), etc. If the register holds "1011" and the SERIAL input is '1' prior to the clock edge, after the clock edge the register will hold "1101". If SERIAL is changed to '0', after the next clock edge the register will hold "0110". If SHIFT is deasserted, the register will continue to hold "0110" after the next clock edge. A register with this structure is called a **shift register**.

**Checkpoint 17.1.3 A left-shifting register.** Draw the logic diagram for a shift register that shifts left, rather than right.

The register in Figure 17.1.2, p. 169 has serial input and parallel output. It is also possible to have a register that loads in parallel and outputs serially. This is shown in Figure 17.1.4, p. 170. This is a bare-bones version of the parallel-to-serial register that is always shifting except when it is being loaded. An additional input can be added to enable or disable the shifting.

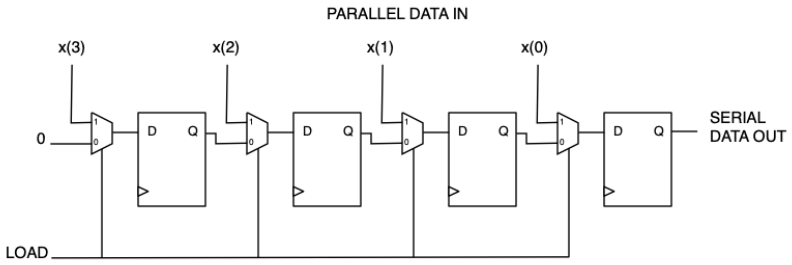



**Figure 17.1.4** A four-bit parallel-in, serial-out, shift register. The bits in the register are shifted right on each rising clock edge. If LOAD is asserted, the parallel data bits are loaded into the flip-flops. If LOAD is not asserted, the register holds "0000" after four clock cycles.

**Checkpoint 17.1.5 Parallel-to-serial shift register.** A more flexible version of the parallel-to-serial shift register includes an input to control whether the register shifts or not. Draw the logic diagram for a register with both LOAD and SHIFT controls. LOAD has priority over SHIFT. If LOAD and SHIFT are both 0, the register holds its contents.

**Hint**. Expand the multiplexers to have more inputs.

## 17.2 VHDL models for shift registers

Recall that the parallel-in, parallel-out register is modeled by a VHDL process, like this:

```
-- x and y are vector signals of the same type and size
process(clk)
begin
    if rising_edge(clk) then
       if clear='1' then
          y <= (others=>'0');
       elsif load='1' then
          y <= x;
       end if;
    end if;
end process;
```

The four-bit parallel-in, serial-out register in Figure 17.1.2, p. 169 could be modeled like this, continuing to use `y` to denote the contents of the register.

```
-- x and y are four-bit vector signals
serial_out <= y(0);
process(clk)
begin
    if rising_edge(clk) then
       if load='1' then
          y <= x;
       else
          y(3) <= serial_in;
          y(2) <= y(3);
          y(1) <= y(2);
          y(0) <= y(1);
       end if;
    end if;
end process;
```

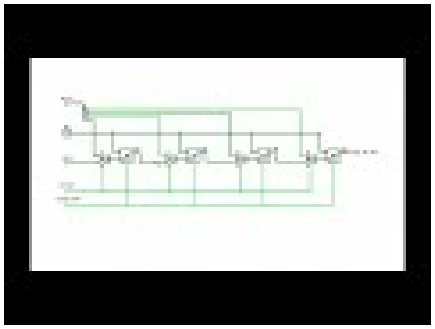
This is logically correct, but it rapidly becomes unwieldy for modeling large registers. A compact way of writing the same model is like this:

```
-- x and y are four-bit vector signals
serial_out <= y(0);
process(clk)
begin
   if rising_edge(clk) then
      if load='1' then
         y <= x;
      else
         y <= serial_in & y(3 downto 1);
      end if;
   end if;
end process;
```

**Listing 17.2.1** VHDL process modeling a right-shifting parallel-to-serial register. The `&` symbol denotes concatenation.

This model introduces a new VHDL language feature, the **concatenation** operator, `&`. The expression `serial_in & y(3 downto 1);` says that the `serial_in` bit is concatenated with the upper three bits of `y`, making a new vector with `serial_in` in the most-significant place and `y(1)` in the least-significant place. Then, the assignment of this new vector to `y` on the rising clock edge causes the contents of the register to be replaced by this new vector, i.e., the register has been shifted. This construction is easily expanded to longer registers simply by changing the upper index of `y`, e.g., from 3 to 7, 15, 31, etc.

This video complements the material presented so far:



### Exercises

**1.** Modify Listing 17.2.1, p. 171 to model a left-shifting register.

**2.** Write a VHDL model for the parallel-to-serial register in Checkpoint 17.1.5, p. 170.

## 17.3 Serial communication

The most important application of shift registers is data communication. Imagine you want to send 8-bit vectors (bytes) from a source device to a destination device. These vectors could be text (8 bits per character), or data from an analog-to-digital converter. If the bits are sent in parallel, one wire is required for each bit. The area required on a circuit board is eight times that required by one wire. The cost of the copper for a cable is eight times the cost for a single wire, and the cable is bulkier than a single wire. Moreover, a device with eight parallel input/output pins rather than a single serial pin is going to require a larger package (an integrated circuit package is much larger than the chip within because of the pins).

Alternatively, using a shift register, each byte can be converted to a stream of eight serial bits. The bits are sent through a single wire to a shift register at the other end, which converts the data back to eight-bit bytes. The first shift register **serializes** the data, and the second shift register **deserializes** it (Figure 17.3.1, p. 172). The tradeoff for using less wire is the additional hardware for the shift registers and associated controls, and an 8× higher clock speed on the wire between the serializer and the deserializer. Except for the very highest data rates, this tradeoff is usually considered acceptable.

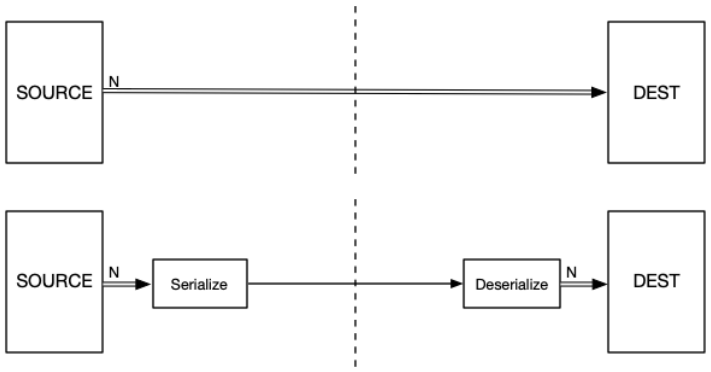



**Figure 17.3.1** Source and destination devices connected by an N-bit parallel data link (top), and connected via a one-bit serial link (bottom). Shift registers serialize and deserialize the data. The bit rate on the serial link is N× the rate on each wire of the N-bit bus.

While I said the serial link requires only one wire, in reality any data link requires some kind of clock or other signal to maintain synchronization of the source and destination. For short wiring runs, it is not too costly to use a second wire to send a clock signal alongside the data wire. But for a long run, the extra clock line becomes contly or is prohibited by the system architecture (e.g., a telephone line does not have a wire that can be used to carry a clock). These design problems have led, over time, to the development of a variety of serial data protocols to solve the synchronization problem. We will learn about two of these is the next two chapters. All serial methods, however, will make

use of the shift registers developed here.

# Chapter 18

# Asynchronous serial communication (UART)

## 18.1 Serial communication

In broad outline, a serial data link is shown in Figure 18.1.1, p. 175, below. Serial data transmission has the great advantage of requiring less wiring and less space on a circuit board. The cost is the additional circuitry to serialize and deserialize the data, and the need to accommodate an N× higher bit rate over the link. In many applications, the benefits outweigh the costs. Many integrated circuits, like analog-to-digital and digital-to-analog converters, send and receive data serially. Nearly all wired (e.g., USB) and wireless (e.g., Bluetooth) communications are serial.

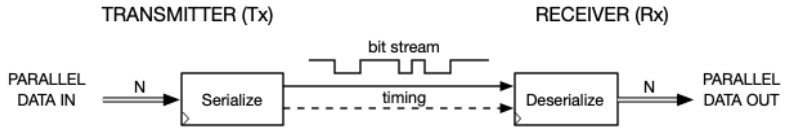



**Figure 18.1.1** Serial data communication. A block (e.g., a byte) of parallel data is serialized and transmitted as a bit stream over a single wire. In some protocols, additional wires carry timing signals to synchronize the receiver with the transmitter. The received bit stream is deserialized and output as parallel data.

The serializer includes a parallel-to-serial shift register to convert a data byte to a packet of bits in a serial stream. The deserializer has a serial-to-parallel shift register to reconstruct the byte from the received stream. Both transmitter and receiver have state machines to manage control signals and the shifting of the registers. The design of a serial link requires two problems to be addressed. Both concern the synchronization of the receiver with the transmitter.

1. The deserializer's register must shift at the proper times, once per bit, and safely within each bit.

2. The deserializer must know when a new packet is arriving, so that the stream is properly **framed** into bytes.

There are many methods, or **protocols**, for serial communication, which solve these two problems in different ways. For example, the first problem can be solved by sending a clock signal from the transmitter to the receiver, at the cost

of an additional wire. Alternatively, if the transmitter and receiver "agree" on a bit rate, then they can have their own clocks running at the same frequency. Then, the problem changes to making sure the receiver's clock is in phase with the transmitter's clock.

The framing problem can be solved by sending a parallel synchronization signal that is asserted with the first bit of a new packet. Alternatively, as we shall soon see, the framing signal can be embedded in the bitstream. A state machine inside the receiver sees this assertion and initiates shifting in response. The framing signal can also be used to adjust the phase of the receiver's clock to make sure the register is shifted at the right time.

Serial protocols that send timing signals alongside the data are said to be **synchronous**. Other protocola embed the timing information in the bitstream and do not require synchronization signals. They are said to be **asynchronous**, though "self-synchronous" might be a better way to think about it.

The rest of this chapter introduces a classic asynchronous protocol, called the Universal Asynchronous Receiver and Transmitter, or **UART**. For most applications, the UART protocol has been superseded by faster methods, but it still used when simplicity is more valuable than speed. Many microcontrollers include pins to support UART communication, some wireless transceiver chips have UART interfaces, and the MIDI (Musical Instrument Digital Interface) protocol for keyboards and synthesizers uses UART communication.

## 18.2 UART protocol

In the UART protocol, an 8-bit byte (typically an ASCII[1]-encoded character) is transmitted as shown in Figure 18.2.1, p. 176, below. When nothing is being sent (the line is quiescent), the signal is at a logical '1' level. The beginning of a byte is signalled by the line going from from '1' to '0' for one bit time. This first bit is called the **start bit**. It is followed by the eight data bits, least-significant bit first, and then a **stop bit**, which is always a '1'. The line then idles at '1' until the next byte. A so-called parity bit may optionally be inserted between the most significant data bit and the stop bit, and there may be an additional stop bit. But this packet format, "8 bits, no parity, 1 stop bit" (sometimes abbreviated 8N1, 8-N-1, or 8-None-1), is the most common and the one that we shall use here.

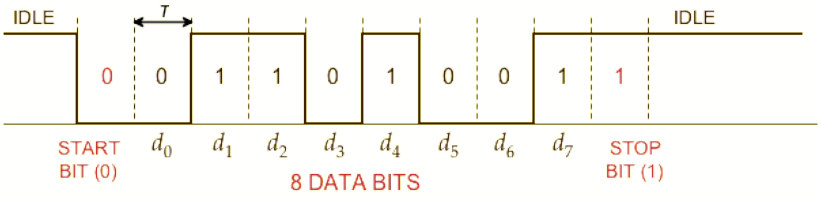



**Figure 18.2.1** Structure of a typical UART data packet. The line is '1' when idle. The beginning of a packet is signalled by the line going from '1' to '0' (start bit). The eight data bits follow, LSB-first. The end of the packet is marked by the line going to '1' (stop bit) and then back to idling. The baud rate is the reciprocal of the bit time, $T$.

In a UART, the reciprocal of the temporal width of a single bit is called the **baud rate**. Common baud rates include 9600, 19,200, and 115,200. The UART receiver and transmitter operate at an agreed-upon baud rate so that the transmitter's clock does not have to be sent alongside the data. (Old

[1] `www.asciitable.com`

dialup modems, which are based the UART protocol, would negotiate a baud rate by sending and receiving test signals before actually commencing data transmission.) With a known baud rate, all that is required for synchronization is for the receiver to know when a new byte is beginning, and that is conveniently taken care of by the start bit.

**Checkpoint 18.2.2 Baud rate.** The average keyboard user can type at a rate of 52 words per minute (a standard word is five letters and a space character). A trained typist can go as high as 90 words per minute. Assuming 90 words per minute, and 8 bits per character (ASCII code), what is a reasonable baud rate (110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200) for communicating typed text?

**Answer**. Remarkably, 110 baud is sufficient even for a fast typist!

**Checkpoint 18.2.3 Baud rate.** A 4 megapixel color image (24 bits/pixel) can be JPEG-compressed to about 8 bits/pixel. How long would it take to download this image over a serial link using a UART protocol at 57,600 baud (the fastest dialup modem, back in the day)?

**Answer**. Approximately 700 seconds, or 12 minutes. Be grateful for WiFi and the cell phone network!

## 18.3 UART system design

**Transmitter.** The transmitter datapath is simple. A block diagram is shown in Figure 18.3.1, p. 177.

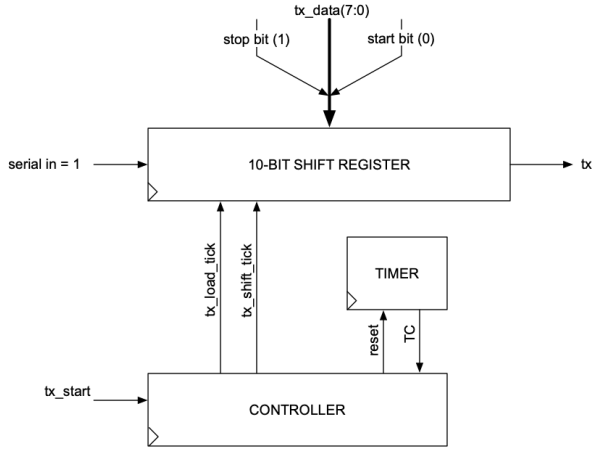



**Figure 18.3.1** A UART transmitter consists of a 10-bit shift register for parallel-to-serial conversion, a bit time counter to mark the time between shifts, and a controller (state machine).

The controller is diagrammed in Figure 18.3.2, p. 178. The state machine is normally in a loop governed by the timer, which is designed to assert TC for one clock cycle, and restart itself, every bit time. When it times out, the register is shifted by asserting tx_shift_tick for one clock cycle. When idle, the output of the shift register is '1'. To initiate a transmission, the user sets up the tx_data byte and asserts the tx_start signal for one clock cycle. This causes the state machine to assert tx_load_tick for one clock cycle and also reset the timer. The state machine then returns to the normal shift loop. The load operation immediately places the start bit on the tx line, and subsequent

assertions of tx_shift_tick shift the data bits and, finally, the stop bit, out of the register. The register is then shifting '1's out until the next load.

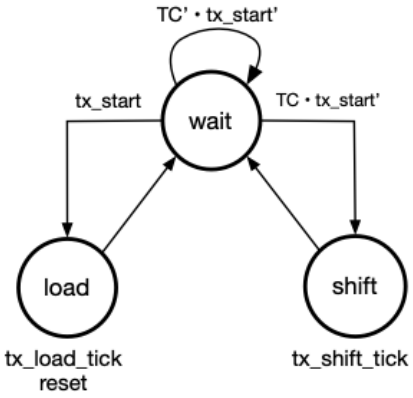



**Figure 18.3.2** State machine for the UART transmitter controller.

**Checkpoint 18.3.3 UART bit timer.** If the system clock is 10MHz, how high does the bit timer have to count for a 115,200 baud rate?

**Answer.** $N = \dfrac{10,000,000}{115,200} \approx 87$

**Receiver.** Receivers are more complex than transmitters, because they don't know what's coming at them or when it's coming. The receiver's datapath is straightforward, just a 10-bit serial-to-parallel shift register, an 8-bit parallel-load output register, and a double-flop synchronizer, because the input stream is asynchronous to the receiver's clock, Figure 18.3.4, p. 178. The shift register is equipped with an rx_shift_tick control (and optionally, a clear control), and the output register has an rx_data_load control. Additionally, the controller outputs a signal rx_done_tick when a byte has been completely received.

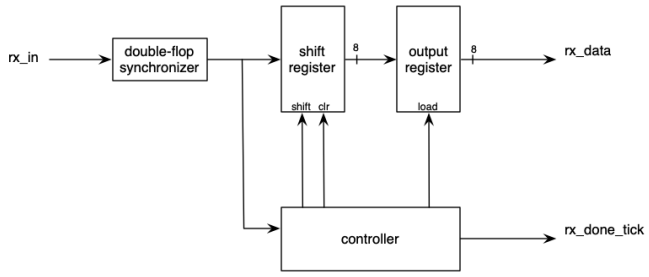



**Figure 18.3.4** Block diagram of the UART receiver.

The interesting part is the controller. We assume that the receiver knows the transmitter's baud rate. The controller has a timer that divides the clock down to the bit time. It also has a counter to keep track of how many times the register has been shifted (how many bits have been received in the current packet). The controller's state machine moves through these steps:

1. Watch the serial input line, waiting for it to drop from '1' to '0', signifying the arrival of the start bit. While in this state, the shift counter and the bit timer are held in reset.

2. Find the center of the start bit. This is done by having the timer count for half a bit time (see Figure 18.2.1, p. 176).

3. Assert rx_shift_tick for one clock cycle, bringing the start bit into the register.
4. Wait for the timer to count one full bit time, then assert rx_shift_tick to bring the first data bit into the register. Repeat this step until the stop bit is shifted into the register (count the shifts).
5. After ten shifts, the middle eight bits of the register (8 down to 1) contain the data byte. Transfer this byte into the output register, rx_data by asserting rx_data_load. This second register ensures that the output remains steady while the next packet is being received.
6. Assert the rx_done_tick pulse for one clock cycle to signal that the data byte is available, then return to the initial state and wait for the next start bit.

When a long line connects the transmitter and receiver, noise can corrupt the received signal and cause the receiver not to function properly. A common error check that can be build into a UART is to make sure that the received stop bit is, in fact, '0', confirming that the state machine didn't just react to a glitch on the line. Another is to observe the final contents of the shift register to make sure that the LSB is '0' and the MSB is '1'. If this is not observed, then a framing error has occurred.

### Exercises

1. **UART transmitter design.** Convert the UART transmitter block diagram, Figure 18.3.1, p. 177, and state diagram, Figure 18.3.2, p. 178, into a VHDL model, write a testbench, and confirm that the model works correctly.
2. **UART receiver design.** Draw a state diagram for the UART receiver. Convert the block diagram, Figure 18.3.4, p. 178, and the state diagram to a VHDL model, write a testbench, and confirm that the model works correctly. You may also use your transmitter model to generate a simulated bitstream and test the combination end-to-end.
3. **UART prototype.** Take your transmitter and receiver designs through synthesis and implementation. Test your FPGA prototype in one of two ways (or both):

    - Set up a data byte for the transmitter with slide switches, and use a pushbutton to assert the tx_start signal. Wire the transmitter output to the receiver input (this is called a "loopback") and display the rx_data on LEDs.
    - Set up a terminal emulator, like PuTTy, and use it to send bytes from a computer to the UART receiver and receive them from the UART transmitter (another loopback). You will need to uncomment the entries for the RsRx and RsTx ports in the XDC file, which map to the USB-to-serial interface on the Basys3 board. Loop rx_data back to tx_data, and use rx_done_tick to trigger tx_start. Check that what you type into PuTTy is displayed in the terminal window via the UART loopback.

## 18.4 The RS-232 standard

The UART protocol is sometimes called "RS-232". While they are closely related

historically, they are not synonymous and should not be confused. RS-232 is a standard for the wiring of a serial interface and the voltages on those wires. It is independent of the way the bits are formatted, which is the UART protocol. RS-232 specifies, at a minimum, one wire for the transmitted signal, one for the received signal, and a ground. Data can move in both directions at the same time, so called "full duplex" operation. The voltage levels on the data wires are allowed to be between +3V and +15V for logical '0', and between -3V and -15V for logical '1'. Special interface chips, such as the MAX232[1], have been developed which convert the RS-232 voltages to normal logic levels. While the standard also defines additional wires for special control features, many UART protocol applications today, in microcontrollers, MIDI, etc, use only the transmit, receive, and ground wires. If you ever need to work with an actual RS-232 interface, you can find ample information about it online.

[1] www.ti.com/lit/ds/symlink/max232.pdf

# Chapter 19

# Serial Peripheral Interface (SPI)

## 19.1 Serial communication

Chapter 18, p. 175 introduced serial communications and the two problems that must be solved in any serial link design (Figure 19.1.1, p. 181):

1. The receiver's serial-to-parallel register must shift at the proper times, once per bit, and safely within each bit.
2. The receiver must know when a new packet is arriving, so that the stream is properly **framed** into bytes.

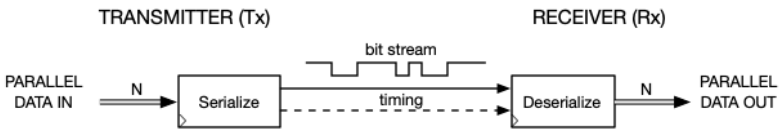



**Figure 19.1.1** Serial data communication. A block (e.g., a byte) of parallel data is serialized and transmitted as a bit stream over a single wire. In some protocols, additional wires carry timing signals to synchronize the receiver with the transmitter. The received bit stream is deserialized and output as parallel data.

In the UART protocol for asynchronous serial communication, the synchronization is embedded in the data stream (Section 18.2, p. 176). In this chapter we will see a different approach, a synchronous protocol called the **Serial Peripheral Interface (SPI)** bus. It solves the synchronization problem by sending a clock signal and a framing signal alongside two (transmit and receive) data lines. A state machine in the transmitter generates the timing signals, and a state machine in the receiver uses the timing signals to control the receiver's shift register.

The SPI bus is used for package-to-package communication on a single circuit board. It is capable of full duplex (bidirectional) communication at high speeds, up to 10 million bits per second. Many of the peripheral modules, or Pmods[1], available for the Digilent FPGA development boards use a SPI interface.

[1] `store.digilentinc.com/by-communication-protocol/spi/`

## 19.2 SPI protocol

The SPI bus is diagrammed in Figure 19.2.1, p. 182. The bus is designed to have one master device that controls the communication, and one or more peripheral devices. The single peripheral setup is most commonly encountered in this course.

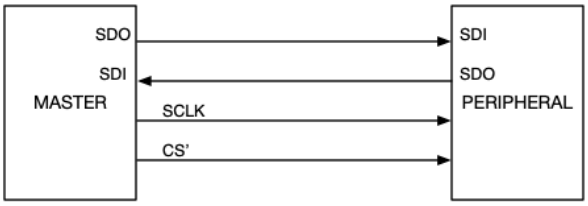



**Figure 19.2.1** SPI bus nomenclature. Communication is initiated and synchronized by the master device. The synchronization signals are: Serial Clock (SCLK) and Chip Select (CS', low-true). The data signals are Serial Data Out (SDO) and Serial Data In (SDI).

Four wires are defined between the master device and the peripheral device. Some peripherals are receive-only (e.g., digital-to-analog converters) and some are send-only (e.g., analog-to-digital converters). These will have both timing lines but only one data line. The designations in the figure for the SPI bus signals are SCLK, CS', SDO, and SDI. Many variations exist, however (e.g., this list in a Wikipedia article[1]).

The master initiates a SPI bus transaction by asserting the CS' signal (LOW); it then holds it LOW for the duration of the transaction. One bit of data is sent (SDO) or received (SDI) in each SCLK cycle until the full packet (e.g., 16 bits) has been moved. The master concludes the transaction by releasing CS'. Each device has only one data (shift) register. When a peripheral device has both SDI and SDO ports, the received data is shifted into the data register as the transmitted data is shifted out of the same register.

This is straightforward enough, but the devil is in the details. Referring to Figure 19.2.2, p. 183, the serial clock and the data streams are permitted by the SPI bus definition to have four possible timing relationships. The first SCLK edge following the CS' transition from HIGH to LOW is called the "leading edge" (marked by a red line in the figure), and the opposite edge is called the "trailing edge" (marked by a blue line). The leading edge may either be rising ("positive polarity", denoted CPOL=0) or falling ("negative polarity", CPOL=1). Data bits are transmitted, or launched, on one SCLK edge and received, or captured, on the opposite edge. And, there are two possibilities for this: launch on the leading edge and capture on the trailing edge (CPHA=1), or vice versa (CPHA=0).

[1] en.wikipedia.org/wiki/Serial_Peripheral_Interface#Interface

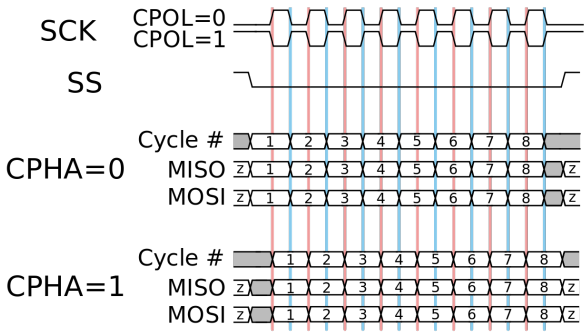



**Figure 19.2.2** SPI bus timing, from [17]. There are two possible relationships between the SCLK polarity and the assertion of CS', and two possible phase relationships between SCLK and the launch/capture of the data.

The good news is that a particular peripheral device will implement only one of these four possibilities, and it will be documented in the device's data sheet. An example is the timing diagram for the ADCS7476 analog-to-digital converter [10], below.

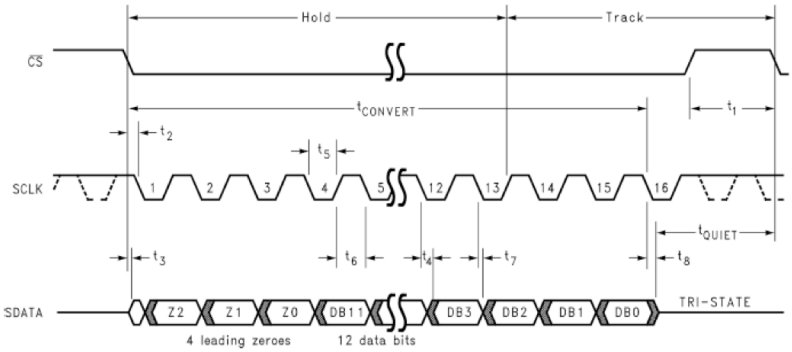



**Figure 19.2.3** SPI bus timing for the ADCS7476 analog-to-digital converter, excerpted from [10]. The data bits are launched from the '7476 on the falling edge of the serial clock, and captured by the master device on the rising edge. The indicated timing specifications are described in the data sheet.

Following the assertion of CS' on the falling edge of the serial clock (SCLK), the serializing register inside the ADCS7476 shifts, launching a data bit onto the SDATA (SDO) line. On the next rising edge of SCLK, the deserializing register inside the master shifts, capturing the data bit. According to the timing diagram and the data sheet [10], the received data will be valid (clearly '1' or '0') and safe to capture no later than $t_4 = 20$ns after the falling edge. The maximum SCLK frequency is specified to be 20MHz (50ns period). Thus, the data is guaranteed to be valid for at least 5ns before the rising edge captures it.

## 19.3 SPI interface design

We will illustrate the process for designing a SPI interface by continuing to work with the ADCS7476 analog-to-digital converter. This is the chip used by the Digilent Pmod AD1[1] two-channel analog-to-digital converter module,

[1] store.digilentinc.com/pmod-ad1-two-12-bit-a-d-inputs/

and also the Pmod MIC3[2] microphone module. Both are popular devices for student projects in this class.

The core of the receiver, of course, is a deserializing (serial-to-parallel) shift register. The ADCS7476 is a 12-bit converter, converting a voltage between 0 and 3.3V to an integer between 0 and $2^{12}-1=4095$. There are three or four preamble bits, extending the '7476 output to 15 or 16 bits, depending on the relative timing of the CS' and SCLK signals (see the data sheet for details). Here, a 16-bit register will be used. The preamble bits are shifted out, followed by the data, from the MSB (bit 11) to the LSB (bit 0). To keep the bits in the right order, the register must shift left rather than right. The datapath will also need a 12-bit parallel-load output register to hold the result. The shift register will have a shift control input, and the output register will have a load control input.

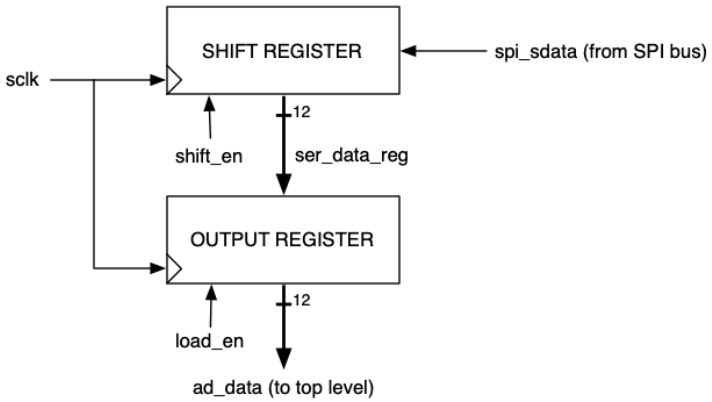



**Figure 19.3.1** Datapath for ADCS7476 SPI bus interface.

The A/D converter's maximum sampling rate is specified to be 1 MSPS (million samples per second) with a maximum 20MHz SCLK frequency. This 20:1 ratio holds for other sampling rates, i.e., if the sampling rate is 48kHz (typical for high quality audio), the SCLK frequency must be at least 960kHz (round up to 1MHz). If minimizing power consumption is a design goal (and it usually is), the FPGA and the SPI bus should be run at the lowest practical speeds. If we determine that 1MHz is also an acceptable speed for the FPGA, then it will be convenient to divide the 100MHz master clock down to 1MHz and use it for both the FPGA and the SPI bus. The FPGA and the SPI bus can use the same clock up to 20MHz, but for a higher FPGA speed we have to separate the clocks. Multi-clock systems are routinely built, but are beyond the scope of this course. Let's assume, therefore, that 1MHz is a good clock speed for both the FPGA and the SPI bus.

The controller initiates a conversion when it sees an assertion of an input called take_sample. The controller supplies two signals to the SPI bus: spi_cs (CS') and spi_sclk (SCLK). It also supplies the control signals for the two registers: shift_en and load_en (Figure 19.3.2, p. 185). The diagram shows SCLK simply being forwarded from the FPGA to the SPI bus. This is actually not permitted by the Xilinx EDA tools (clock signals are treated specially), but for simplicity we'll defer this detail to the lab.

[2] store.digilentinc.com/pmod-mic3-mems-microphone-with-adjustable-gain/

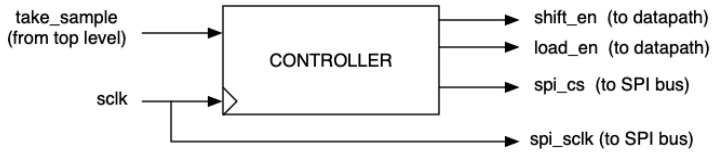



**Figure 19.3.2** Controller signals for ADCS7476 SPI bus interface.

Inside the controller, we need a state machine to manage the control signals and a counter to keep track of the register shifts. The state machine idles, waiting for take_sample. During this time the shift counter is held in its clear state. When take_sample is asserted, the state machine moves through its sequence:

1. CS' is asserted (LOW) to start the conversion and the shift register is enabled with shift_enable, which also enables the shift counter.

2. Since SCLK is also the system clock, we can count shifts by just letting the counter run until enough shifts have occurred. Read the ADCS7476 timing diagram and datasheet [10] carefully to make sure that you don't stop shifting one clock cycle too early or one cycle too late.

3. Deassert CS' and shift_en. Assert load_en to transfer the lower 12 bits of the shift register to the output register.

4. Return to the idle state.

## Exercises

**1.** Write a state diagram for the SPI bus controller.

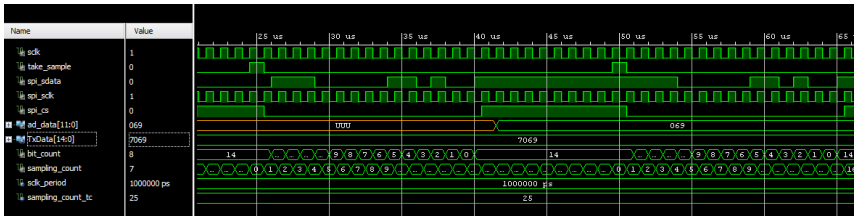
**2.** Translate the block diagram and the state diagram for the SPI interface into a VHDL model. Organize your design with these five processes:

- Shift register
- Output register
- Controller: state update and next-state/output logic, as usual
- Shift counter

**3.** Here are the key declarations and stimulus processes for a testbench that simulates the output of the A/D converter. Complete the testbench and use it to validate your design.

```
--  Complete testbench with UUT declaration,
    instantiation, and clock process,
--  entity/architecture, libraries, etc.

--  Clock period definitions
    constant sclk_period : time := 1 us;           -- 1
        MHz serial clock
    constant sampling_count_tc : integer := 25;    --
        40 kHz sampling rate, for testing

--  Data definitions
    constant TxData : std_logic_vector(14 downto 0) :=
        "111000001101001";
    signal bit_count : integer := 14;

--  Internal definitions
    signal sampling_count : integer := 0;

--  Stimulus process:  assert take_sample periodically
    to initiate conversions
    stim_proc_1: process(sclk)
    begin
       if rising_edge(sclk) then
          if sampling_count < sampling_count_tc-1 then
             sampling_count <= sampling_count + 1;
             take_sample <= '0';
          else
             sampling_count <= 0;
             take_sample <= '1';
          end if;
       end if;
    end process stim_proc_1;

-- Stimulus process:  testbench pretends to be the A/D
    converter
    stim_proc_2: process(spi_sclk)
    begin
       if falling_edge(spi_sclk) then  -- clock data
           out on falling edge, MSB first
          if spi_cs = '0' then         -- watch for SPI
              interface to activate the A/D
             spi_sdata <= TxData(bit_count);
             if bit_count = 0 then bit_count <= 14;
             else bit_count <= bit_count - 1;
             end if;
          end if;
       end if;
    end process stim_proc_2;
```

Your simulation results should look like this:



# Chapter 20

# Video graphics (VGA)

## 20.1 Digital images

We live in a sea of digital images—phones, laptops, watches, televisions, cameras, games, automobile dashboards, advertising... how many more can you think of? They vary widely in size (height, width), resolution (number of pixels, number of colors, number of frames per second), and technologies (LCD, LED, and OLED are among the abbreviations you may have encountered while shopping for a new device). In this chapter we will focus on one particular type of video display, called **Video Graphics Array**, or **VGA**. Since its introduction by IBM in 1987 for its PS/2 line of personal computers, it has undergone many modifications and extensions. We shall be concerned with the particular implementation supported on the Digilent Basys 3 board. You can read more about the VGA standard, including the original IBM specifications, beginning with the Wikipedia article [18]. We shall also refer to the Basys 3 reference manual [3]. Another interesting source is the "Project F" website [7].

A two-dimensional digital image is a rectangular array of picture elements, or **pixels**. Image size is conventionally expressed as "width × height", e.g., a VGA image, which has 640 columns and 480 rows, is expressed "640 × 480". This is opposite the way arrays are sized in mathematics—a matrix with 640 columns and 480 rows is said to be "480 × 640". However, both the elements of a matrix and the pixels of a VGA image are indexed in the same way, by (row, column), from coordinates $(0, 0)$ at the upper left corner to $(479, 639)$ at the lower right corner.

Each pixel is represented by a $B$-bit number encoding its color. For simple black and white, one bit is sufficient. Eight bits will encode 256 gray levels. Four bits, in the original VGA specification, encoded 16 colors. The Basys 3 supports up to 12 bits (4096 colors) per pixel: 4 bits of red, 4 bits of green, and 4 bits of blue.

**Checkpoint 20.1.1** How many bits of memory are required to store a 12-bit VGA image? How many bits of block RAM does the Artix 7 FPGA on the Basys 3 board contain [3]?

**Answer**. $640 \times 480 \times 12 = 3,686,400$ bits. And according to the reference manual, the FPGA contains 1,800 Kbits. Evidently, there isn't enough memory to store this image. Something will have to be done!

A video image is a time sequence of single images, or **frames**. The speed at which individual still frames are displayed is called the **frame rate**, expressed in **frames per second (fps)**. The frame rate must be high enough to create the illusion of smooth, flicker-free motion in the mind of the viewer. Typical

frame rates are 24 fps for movies, 30 fps for television, and 60 fps for VGA graphics.

**Checkpoint 20.1.2** Why do you think a computer screen has a higher frame rate than a movie or television program?

**Hint**. It may have something to do with the eye's sensitivity to flicker in stationary vs moving images. Read up on it!

**Checkpoint 20.1.3** How many pixels per second are sent to a VGA display operating at 60 fps? How does this compare with the clock speed of the Basys 3 board [3]?

**Answer**. $640 \times 480 \times 60 = 18,432,000$ pixels per second. This is a less than than the 100 MHz Basys 3 clock speed.

## 20.2 VGA image displays

We have seen that a VGA image, for our purposes, is a 640 (columns) $\times$ 480 (rows) array of one- to twelve-bit pixels, updating at 60 frames per second. All that data must be communicated from the source (in our case, the Artix 7 FPGA on the Basys 3 board) to a VGA-compatible monitor.

Originally (think old TV sets), the glass screen of a "picture tube" was coated with a phosphorescent material that lit up in response to excitation by a beam of electrons. As the electron beam was swept in a **raster** pattern, line-by-line from top to bottom, the image was "painted" on the phosphorescent screen (see this Wikipedia article[1] for more information about this vintage technology). Color images required three electron beams and more complex three-phosphor coatings to make red, green, and blue. Three analog voltages controlled the intensities of the beams as they moved, to create different mixtures of the three colors at each pixel. Modern monitors use either light-emitting diodes or backlit liquid crystal shutters to create the pixels, and the raster scanning is accomplished by counters that address the pixels sequentially, from left to right and from top to bottom.

On the Basys 3, the values of red, green, and blue are each represented by a four-bit number. These are converted to analog voltages between 0 and 0.7 volts for compatibility with the monitor's video inputs. In addition to the red, green, and blue video signals, two synchronization signals are required so that the counters in the monitor address the pixels in time with the sending of the data from the source. Horizontal sync pulses of a specified width produced by the source indicate the end of a single line, reset the horizontal counter and increment the vertical counter to the beginning of the next line. A vertical sync pulse of specified width at the end of a full frame resets the vertical counter back to the first line. A standard 15-pin connector brings these five video and sync signals from the source to the monitor, Figure 20.2.1, p. 188

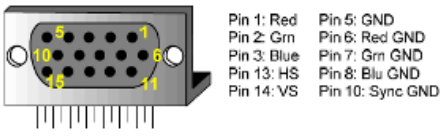



**Figure 20.2.1** DB-15 VGA connector [3]. The five signals are the Red, Green, and Blue pixel voltages, each with its own ground, and the horizontal (HS) and vertical (VS) sync pulses, which share a ground.

[1] en.wikipedia.org/wiki/Cathode-ray_tube

**Checkpoint 20.2.2** What are the five signals required to display an image on a VGA monitor? Which are analog, which are digital?

**Answer**. Red, green and blue video (analog); horizontal and vertical sync (digital).

For historical reasons (back to those analog video displays), there are blank pixels before and after a line of video, and blank lines above and below a frame of video. This has the effect of embedding the actual $640 \times 480$ image in a larger $800 \times 525$ field. Over sixty frames, this comes to $800 \times 525 \times 60 = 25.2$ million pixels per second. It is more convenient for us to use a 25 MHz pixel clock, since it is simply one fourth of the 100 MHz Basys 3 master clock. For modern monitors, the slower clock works fine if the number of lines is reduced from 525 to 521 ($800 \times 521 \times 60 = 25.008$ MHz).

A timing diagram for the video signals is shown in Figure 20.2.3, p. 189. Beginning at the left edge of the active area (column 0), there are 640 active pixels. The rest of the pixels, from 640 to 799, are blanked (zero), as indicated by the horizontal blanking (HB) waveform. The blanked region is divided into the "front porch", the horizontal sync pulse (HS), and the "back porch", which immediately precedes the start of the next active line. From the top of the active area (row 0), there are 480 active lines, followed by the vertical blanking region (lines of all zeros) from line 480 to line 520. Within this region are the bottom border, the vertical sync pulse, and the top border. The horizontal and vertical time scales are not the same. Although the vertical sync pulse appears quite thin in the drawing, just two lines, because each line consists of 800 pixels, it is actually quite long.

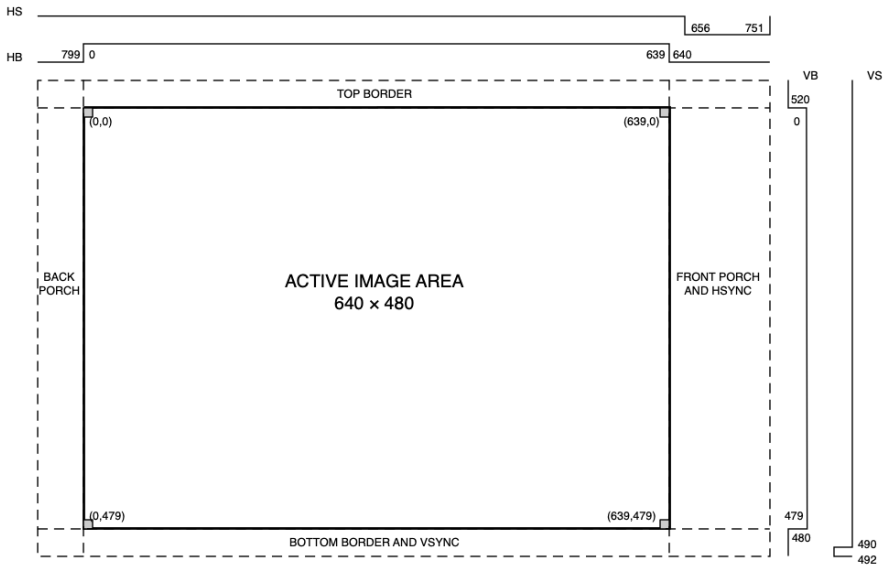



**Figure 20.2.3** Horizontal and vertical VGA timing. The horizontal time scale is in pixels. The vertical scale is in lines (one line is 800 pixels). HB and HS: Horizontal blanking, horizontal sync. VB and VS: Vertical blanking, vertical sync. Each pixel is one clock cycle at 25 MHz (40 ns). Each line is 800 pixels (32 $\mu$s).

## 20.3 VGA interface: Controller

A VGA display interface must do two things: Produce the stream of pixels from the data to be displayed, and generate the horizontal and vertical sync

pulses that synchronize the monitor with the pixel stream Figure 20.3.1, p. 190. We can designate the part that produces the pixel stream as the datapath for the interface, and the part that makes the sync pulses the controller. We will begin with the controller in this section and then present a couple of options for datapath design in the next section.

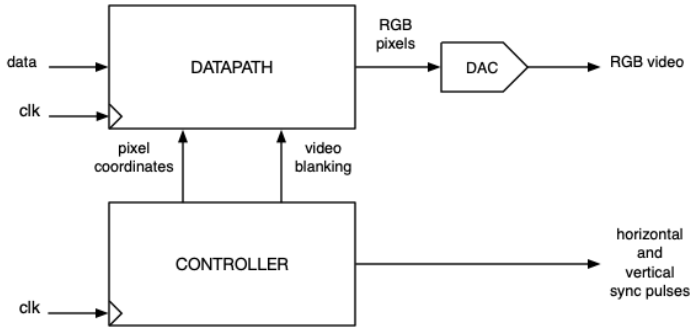



**Figure 20.3.1** Block diagram of VGA interface. The controller generates horizontal and vertical sync pulses to synchronize the monitor. The datapath uses pixel coordinates and blanking signals from the controller to produce the stream of red, green, and blue pixel (RGB) pixel values, which are converted to three analog red, green, and blue video signals by digital-to-analog converters (DACs).

The timing of the blanking and sync signals was shown in Figure 20.2.3, p. 189. The same information, in a tabular format, is shown in Figure 20.3.2, p. 190.

| | Horizontal (pixels) | Vertical (lines) |
|---|---|---|
| Active area | 640 | 480 |
| Front porch / bottom border | 16 | 10 |
| Sync pulse | 96 | 2 |
| Back porch / top border | 48 | 29 |
| Total | 800 | 521 |

**Figure 20.3.2** VGA timing parameters. For actual times, multiply the number of pixels by 40 ns, and the number of lines by 32 $\mu$s.

Ordinarily, we think of control signals being generated by a state machine that interacts with the outside world and with the datapath. Making VGA sync signals is a much simpler problem. There are no exteral control inputs, and no status signals from the datapath, that would alter the controller's state sequence. We just need to assert four signals (horizontal and vertical sync, horizontal and vertical blanking) at regular intervals. This is easily done by a pair of counters (horizontal and vertical), plus some decoding logic. The horizontal counter is clocked by the 25 MHz pixel clock, and counts from 0 to 799. The vertical counter is also clocked by the 25 MHz pixel clock, but is only enabled every 800 clock cycles by the terminal count of the horizontal counter, and counts from 0 to 520. These counters supply the indexing for the pixels in the datapath. Comparators (`if-then-else` logic, in VHDL) convert the horizontal and vertical counts into the sync and blanking signals, using the values in Figure 20.3.2, p. 190.

**Checkpoint 20.3.3** Design a block diagram and VHDL model for the VGA controller using two counters and comparator logic, as described in this section. Here is a suggested list of input and output ports:

```
clk:       in    -- 25 MHz pixel clock
pixel_x:   out   -- horizontal index
pixel_y:   out   -- vertical (line) index
h_sync:    out   -- horizontal sync pulse
v_sync:    out   -- vertical sync pulse
h_blank:   out   -- horizontal blanking signal
v_blank:   out   -- vertical blanking signal
```

It is straightforward to testbench the controller simply by supplying a 25 MHz clock and comparing the outputs to Figure 20.2.3, p. 189 and Figure 20.3.2, p. 190. Physical test with a VGA monitor requires a datapath, which will be described in the next section.

## 20.4 VGA interface: Datapath

Figure 20.3.1, p. 190 in the previous section showed the VGA interface organized as the familiar datapath and controller. It was shown how the controller may be constructed from a pair of counters, one for the horizontal ($x$) pixel address, and one for the vertical ($y$) pixel address, plus logic to decode these addresses into sync and blanking signals. The function of the sync pulses is to synchronize the raster scanning counters inside the monitor with the incoming video stream. The datapath may be imagined as a lookup table that takes in the pixel coordinates and outputs the corresponding RGB values. Additionally, the blanking signals are used to ensure that pixel values are zero outside the active image area. How the lookup table is constructed differentiates two approaches to designing the datapath.

**Block RAM lookup table.** More typically called a **video RAM** or **frame buffer**, this is a block of read-write memory that contains the pixel values to be displayed. This is diagrammed in Figure 20.4.1, p. 191. The memory has independent read and write ports, and in Xilinx terminology, is called a "simple dual-port block RAM" (for a complete description, see [19]). In this setup, the VGA controller is constantly accessing the RAM and streaming pixels to the display. Simultaneously, new data may be written into the RAM to update the display, although one must be careful to avoid, or at least mitigate the consequences of, reading and writing the same pixel in the same clock cycle.

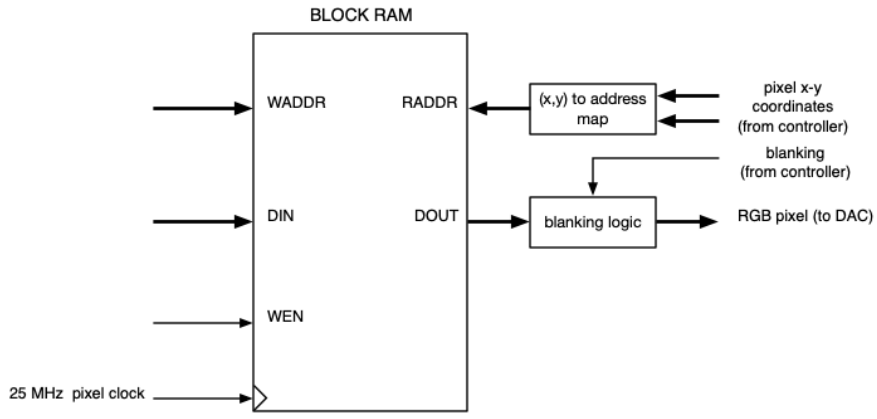



**Figure 20.4.1** Simple dual-port block RAM used as video memory.

Pixels in the image field are indexed by their $(x, y)$ coordinates, but the RAM is indexed by a single address. The mapping from pixel coordinates to

memory address is given by the expression

$$\text{RADDR} = x + y \times \#\text{columns}$$

where the number of columns for a VGA image is 640. The inactive pixels in the image field are not stored in the RAM. They are always zero.

A previous calculation showed that a 640 × 480 image with full 12-bit RGB pixels requires more memory than is contained in the FPGA on the Basys 3. Two strategies for reducing the required memory are:

- Use fewer colors. For most of the graphics applications in this course, the full 12-bit RGB supported by the Basys 3 is not needed. A black-and-white display needs only one bit per pixel. Four colors (e.g., red, green, blue, and black or white) can be encoded with only two bits, reducing memory requirements by a factor of six. Sixteen colors require four bits, yielding a threefold reduction in memory.
- Use a smaller image. Your application may not really need the full 640 × 480 resolution (think about classic video games, which looked "blocky" but were still fun to play). Reducing the displayed image to 320 × 240 will reduce the required memory by a factor of four. To fill the screen at this lower resolution means that the displayed pixels will be 4× larger. These larger pixels are drawn by modifying the mapping from the VGA pixel coordinates to block RAM address as follows:

  $$\text{RADDR} = (x/2) + (y/2) \times (\#\text{columns}/2)$$

  where #columns/2 = 320, and division by two is integer division (0/2 = 0, 1/2 = 0, 2/2 = 1, 3/2 = 1, etc), e.g., a one-bit right shift. In this way, the four pixels at screen coordinates (0,0), (0,1), (1,0), and (1,1) all have the color stored at memory location 0, and appear as a single larger pixel.

**Checkpoint 20.4.2** Check, using the 2 × 2 pixel formula above, the following mappings from $(x, y) \longrightarrow$ RADDR:

- (638,0), (639,0), (638,1), (639,1) $\longrightarrow$ 319
- (0,2), (0,3), (1,2), (1,3) $\longrightarrow$ 320
- (638,478), (639,478), (638, 479), (639,479)$\longrightarrow$ 320×240−1

**Checkpoint 20.4.3** With 640×480 resolution, what is largest number of colors that can be displayed within the memory limits of the FPGA on the Basys 3?

**Answer**. The simple calculation is (1800 Kbits)/(640×480) = 5 bits (rounded down). We can display 32 colors. Considering how memory is actually organized (read the Xilinx documentation, [19]), we'll be limited to four bits, or 16 colors.

**Checkpoint 20.4.4** With larger pixels, 320×240 resolution, what is largest number of colors that can be displayed within the memory limits of the FPGA on the Basys 3?

**Answer**. The simple calculation is, again, (1800 Kbits)/(320×240) = 23 bits (rounded down). We'll have no problem storing full 12-bit RGB.

The FPGA block RAMs are not pure combinational lookup tables. They may include an input register to hold the address, and an output register to hold the memory value fetched from that address. Thus, there may be a one or two clock cycle **latency** between the application of the address and the availability of the data. The design tool for the block RAM will tell you how much latency to expect, and you must be careful to delay the control signals by the same amount, so that they remain properly aligned with the pixel stream.

**Logic-defined lookup table.** Sometimes, what is to be displayed is quite simple, just a few basic shapes against a constant background. (Think, for example, of a video game like the classic "Pong".) In this case, it can be inconvenient to use a full block RAM. Instead, simple logic can be used: "If the current $(x, y)$ is within the boundaries of an object, set the pixel to the color of the object."

For example, a good test of the correctness of your VGA controller design is a classic set of color bars[1], used to calibrate televisions, Figure 20.4.5, p. 193.

**Figure 20.4.5** Society of Motion Picture and Television Engineers (SMPTE) television test pattern.

Each bar is a rectangular area defined by minimum and maximum values of $x$ and $y$. The lookup table is modeled in VHDL by nested `if-then-else` statements, e.g.,

[1] `en.wikipedia.org/wiki/SMPTE_color_bars`

```
entity vga_test_pattern is
   port( pixel_x, pixel_y: in  std_logic_vector(9 downto
       0);   -- enough bits for 800
         color:            out std_logic_vector(11 downto
             0)   -- 12-bit RGB
   );
end vga_test_pattern;

architecture behavioral of vga_test_pattern is
-- Predefine RGB mixtures for the colors
-- SMPTE 75% color bars (12/16 = 75%)
constant WHITE:     std_logic_vector(11 downto 0) :=
    x"ccc";  -- red, green, blue
constant YELLOW:    std_logic_vector(11 downto 0) :=
    x"cc0";  -- red, green, no blue
...

signal urow, ucolumn: unsigned(9 downto 0);

begin

urow <=  unsigned(pixel_y); ucolumn <= unsigned(pixel_x);

pixel_color: process(urow, ucolumn)
begin
   color <= BLACK;              -- default

   -- Large vertical bars
   if (urow >= 0) and (urow < 320) then
      if    (ucolumn >= 0)  and (ucolumn < 92)  then color
          <= WHITE;
      elsif (ucolumn >= 92) and (ucolumn < 184) then color
          <= YELLOW;
      ...
      end if;

   -- Middle row of bars
   elsif ...

   -- Bottom row of bars
   elsif ...

   end if;
end process pixel_color;
end behavioral;
```

**Checkpoint 20.4.6** Download the complete file vga_test_pattern_12.vhd from Canvas. Create a top-level with the test pattern and your VGA controller as components. Create a bitstream file, connect the Basys 3 to a VGA monitor, download your bitstream file to the Basys 3, and check that the test pattern appears correctly on the monitor.

This example used a static lookup table. In the next section we will see how this method can also be used to create simple moving objects, like bouncing balls.

## 20.5 Moving objects and simple games

If you have access to a VGA monitor, build and test your own bouncing ball system as you work through this section.

**Draw a ball.** A 4 × 4 red square "ball", whose upper-left coordinate is (x0,y0), drawn on a white background, can be generated by the following VHDL process:

```
draw_ball: process(pixel_x, pixel_y, x0, y0)
begin
   if (pixel_x >= x0) and (pixel_x < x0+4)
         and (pixel_y >= y0) and (pixel_y < y0+4)
   then
      color <= RED;
   else
      color <= WHITE;
   end if;
end process draw_ball;
```

where `pixel_x` and `pixel_y` are coordinates from the controller, and `RED` and `WHITE` are 12-bit RGB constants (e.g., x"F00" and x"FFF").

**Move the ball.** The ball's position can be dynamically updated with another VHDL process:

```
move_ball: process(clk)
begin
   if rising_edge(clk) then
      if move_ball_en = '1' then
         x0 <= x0 + dx;
         y0 <= y0 + dy;
      end if;
   end if;
end process move_ball;
```

where `dx` and `dy` are the horizontal and vertical increments in the square's position. The enable signal, `move_ball_en`, is asserted for only one clock cycle in every frame. The ball is drawn in a new position in each successive frame, and moves across the screen. The speed at which the ball moves will be $60\sqrt{(dx)^2 + (dy)^2}$ pixels/second.

The best time to assert `move_ball_en` is during the inactive blanking period between the bottom of the frame and the top of the next frame. A state machine can be designed to watch for the v_blank signal to go low, enable the update for one clock cycle, and then watch for v_blank to go high again.

**Checkpoint 20.5.1** What do you think would happen if `move_ball_en` were asserted while the image was being displayed, instead of while the image is inactive?

**Checkpoint 20.5.2** What do you think would happen if `move_ball_en` were asserted for more than one clock cycle in each frame?

**Checkpoint 20.5.3** Draw a state diagram and write a VHDL model for a state machine to update the position of the ball

**Answer**. Here is a sketch of the state machine's combinational logic:

```
FSM_comb: process(curr_state, v_blank)
begin
   next_state <= curr_state;  -- defaults
   move_ball_en <= '0';

   case curr_state is
      when v_on =>
         if v_blank = '0' then
            next_state <= update;
         end if;

      when update =>  move_ball_en <= '1';
         next_state <= v_off;

      when v_off =>
         if v_blank = '1' then
            next_state <= v_on;
         end if;
    end case;
end process FSM_comb;
```

**Bounce the ball.** Once the ball is moving, its coordinates (x0, y0) will increment indefinitely, and the ball will disappear off the edge of the screen, at least until the values of x0 and y0 roll over. A more interesting behavior is to bounce the ball off the "walls" at the edge of the screen. If the ball is moving to the right and it collides with the right wall, its horizontal increment, `dx`, should change sign so that the ball will reverse direction. The vertical increment does not change; if the ball is moving up and to the right when it hits the wall, it will be moving up and to the left after it bounces. Detecting a collision must take into account the size of the ball, so that the ball doesn't move beyond the edge of the screen before bouncing.

Here is a skeleton of a process to detect wall collisions and change directions:

```
ball_hit_wall: process(clk)
begin
   if rising_edge(clk) then
      -- Hit the right wall
      if (dx  > 0) and (x0 >= 640-4) then
         dx <= -dx;

      -- Check the other walls

      end if;
   end if;
end process ball_hit_wall;
```

The literal `640-4` should, of course, be replaced by named integer constants for the location of the wall and the size of the ball. The remainder of the process is left as an exercise.

**Checkpoint 20.5.4** Complete the `ball_hit_wall` process to include collisions with the other three walls.

Combining `draw_ball`, `move_ball`, and `ball_hit_wall` with the update state machine and a VGA controller will draw a ball bouncing around the screen. Once you have this, you can imagine drawing and moving other objects, controlling the movement of objects with user input (like pushbuttons), and colliding one moving object with another. A good project to try is the classic "Pong" game.

A block RAM lookup table can also be dynamically updated. Imagine, for example, making a system that can draw on the screen under user control, like the classic "Etch-A-Sketch" toy. In every frame, the block RAM is read and the contents displayed on the screen by the VGA controller. During the vertical blanking period, the user's controls can be read, move a cursor, and update the color of the pixel that the cursor points to in the block RAM. The screen is cleared by writing the background color to every memory location.

With these principles in mind, a number of VGA based projects can be designed.

## 20.6 Streamlining VGA arithmetic

Managing bricks, checking interiors...

# References


[1] P.J. Ashenden, *A Student's Guide to VHDL*, second edition, Morgan Kaufmann, 2008.

[2] W.J. Dally, R.C. Harding, T.M. Aamodt, *Digital Design using VHDL: A Systems Approach*, Cambridge University Press, 2016.

[3] Digilent, *Basis 3 Reference Manual*, https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf, Accessed November 2019.

[4] Digilent, *Basis 3 Schematic Diagrams*, https://reference.digilentinc.com/_media/basys3:basys3_sch.pdf, Accessed November 2019.

[5] Doulos, *EDA Playground*, https://www.edaplayground.com/, Accessed October 2019.

[6] Doulos, *VHDL Vector Arithmetic using Numeric_std*, https://www.doulos.com/knowhow/vhdl_designers_guide/numeric_std/, Accessed October 2019.

[7] W. Green, *Project F - FPGA Development*, https://projectf.io/sitemap/#fpga-graphics, Accessed August 2019.

[8] Linear Technology, *LTSpice*, https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html, Accessed September 2019.

[9] ON Semiconductor, *MC74HC74A data sheet*, August 2014 Rev. 14, https://www.onsemi.com/pub/Collateral/MC74HC74A-D.PDF, Accessed October 2019.

[10] Texas Instruments, *ADCS747x 1-MSPS, 12-Bit, 10-Bit, and 8-Bit A/D Converters data sheet*, http://www.ti.com/lit/ds/symlink/adcs7476.pdf, Accessed November 2019.

[11] Texas Instruments, *SNx400, SNx4LS00, and SNx4S00 Quadruple 2-Input Positive-NAND Gates data sheet*, http://www.ti.com/lit/ds/symlink/sn74ls00.pdf, Accessed October 2019.

[12] Texas Instruments, *SN5404, SN54LS04, SN54S04, SN7404, SN74LS04, SN74S04 Hex Inverters*, http://www.ti.com/lit/ds/symlink/sn74ls04.pdf, Accessed October 2019.

[13] Texas Instruments, *SNx4HC04 Hex Inverters data sheet*, http://www.ti.com/lit/ds/symlink/sn74hc04.pdf, Accessed October 2019.

[14] Texas Instruments, *Logic Guide* (2017), http://www.ti.com/lit/sg/sdyu001ab/sdyu001ab.pdf, Accessed October 2019.

[15] *STANDARD VHDL PACKAGES*, https://www.csee.umbc.edu/portal/help/VHDL/stdpkg.html, University of Maryland, Baltimore County Department

of Computer Science and Electrical Engineering, Accessed November 2019.

[**16**] J.F. Wakerly, *Digital Design: Principles and Practices*, 4th edition, Prentice-Hall, 2006.

[**17**] Wikipedia, *Serial Peripheral Interface*, https://en.wikipedia.org/wiki/Serial_Peripheral_Interface, Accessed November 2019.

[**18**] Wikipedia, *Video Graphics Adapter*, https://en.wikipedia.org/wiki/Video_Graphics_Array, Accessed August 2021.

[**19**] Xilinx, *UG473: 7 Series FPGAs Memory Resources User Guide*, v1.14, https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf. Accessed September 2021.

[**20**] Xilinx, *UG901: Vivado Design Suite User Guide, Synthesis*, v2018.3, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug901-vivado-synthesis.pdf. Accessed November 2019.

[**21**] Xilinx, *UG953: Vivado Design Suite 7-Series Libraries Guide*, v2018.3, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug953-vivado-7series-libr pdf. Accessed October 2019.

[**22**] *VHDL type conversion chart.* This chart came to me from a former student. Nearly identical versions are found, without attribution, at various places online, e.g., http://www.bitweenie.com/listings/vhdl-type-conversion/, Accessed November 2019.

# Index

## Colophon

This book was authored in PreTeXt.