# Getting Started
# Computer-Aided Design of Digital Circuits
## VHDL Modeling and FPGA Synthesis of Digital Circuits
### ENTITY, ARCHITECTURE, and Simple Signal Assignments

**FPGA** stands for *field-programmable gate array*. We may call *programmable logic arrays* (PLAs), introduced in the early 1970s, the ancestor of today's sophisticated FPGAs, which have undergone amazing changes over the decades. For more information on this and other predecessors just Google it!

Using the 1960s technique to implement the circuit shown in Figure 1, you would need to manually put two SSI chips on the board, and wire them up. This is called *manual* implementation using *discrete* logic. It is manual (as opposed to automated) because it is carried out by hand. Multi-million-transistor circuits cannot be designed by hand, let alone multi-billion-transistor circuits! So, design automation is a must for large circuits. It is also discrete (as opposed to integrated). Even for a medium-sized circuit, you would need many chips should you choose to use discrete logic. It is much more economical in terms of space, performance, and power consumption, to integrate all the chips into one single (possibly huge) chip. Again, computer support cannot be overlooked to reach such an ambitious goal cost-effectively.

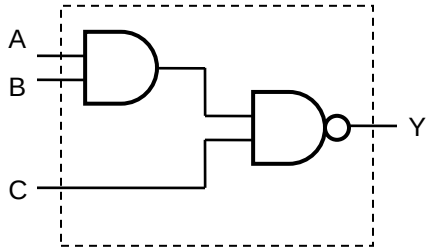



**Figure 1. Logic circuit using two chips**

Powerful CAD tools are now available to handle the full design cycle with different end products namely *full-custom* chips, *semi-custom* chips, and FPGAs. In this book, we will use FPGAs, which are much less expensive for prototyping purposes. Furthermore, the other two options have a long turnaround time, while FPGA synthesis is only one click away! In return, FPGAs are normally slower than full-custom or semi-custom chips. Additionally, FPGAs may not be large enough for very large applications. But do not worry! Our designs in Digital Systems I will be small enough to fit in one FPGA chip!

FPGAs are ready-to-use *programmable* chips. Soon, you will see what we mean by *programmability* in this context. The main parts of an FPGA are *programmable* logic elements (LEs) and *programmable* interconnects to be explained shortly. Each LE (usually) has a look-up table or LUT for short. A medium-sized FPGA has tens of thousands of LEs hence tens of thousands of LUTs. An LUT with $n$ inputs has $2^n$ single-bit memory cells and one output. (A typical value for $n$ in medium-sized FPGAs is 4.) A 3-input LUT ($n = 3$), as an example, has $2^3$ or 8 single-bit storage cells as shown in Figure 2. Every 3-bit input combination specifies (addresses) one unique cell. In other words, each cell in this LUT has its unique address in the range of 000 to 111. By programming an LUT, we mean seating appropriate logic values into these storage cells. Then each input combination (8 in total) will place the content of the corresponding cell at the output of the LUT. So, if you apply 101, as an example, to the LUT of Figure 2, the output will be logic 1.

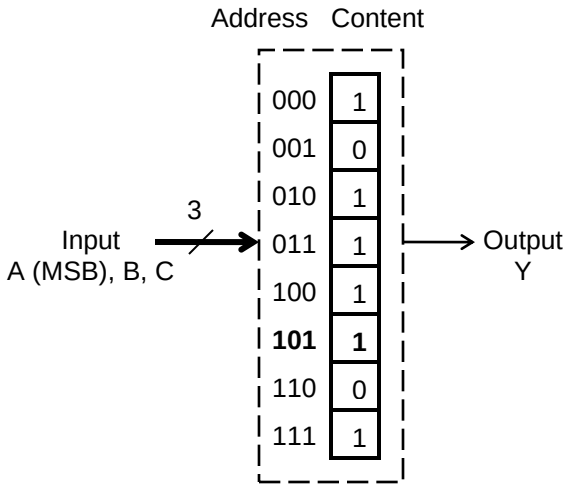
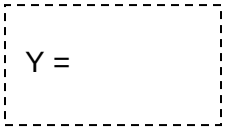


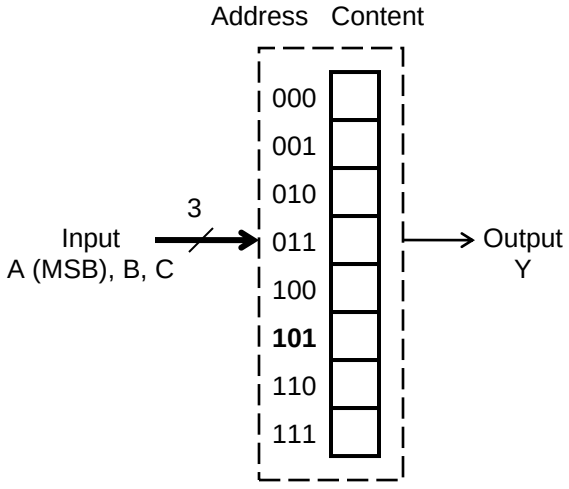

**Figure 2. Three-input LUT**

**Exercise:** In the LUT of Figure 2, what would appear at the output if ABC = 001?




An n-input LUT can be programmed to realize an n-bit logic function. The function mapped into (or realized by) the LUT of Figure 2 is A' . B + A . C + B' . C'. Take a close look. **The content of an LUT is an exact copy of the output column of the truth table of the desired function.**

**Exercise:** Program the following LUT to realize a 3-input XOR gate with inputs A and B.




Programmed LUTs in an FPGA are tied together properly through open-ended reconfigurable interconnects (made up of metal wires and switches) running between LUTs to reach the complete (and possibly very large) digital circuit. Figure 3 shows a simplified view of an FPGA with 3-bit LUTs and four interconnects per routing channel.

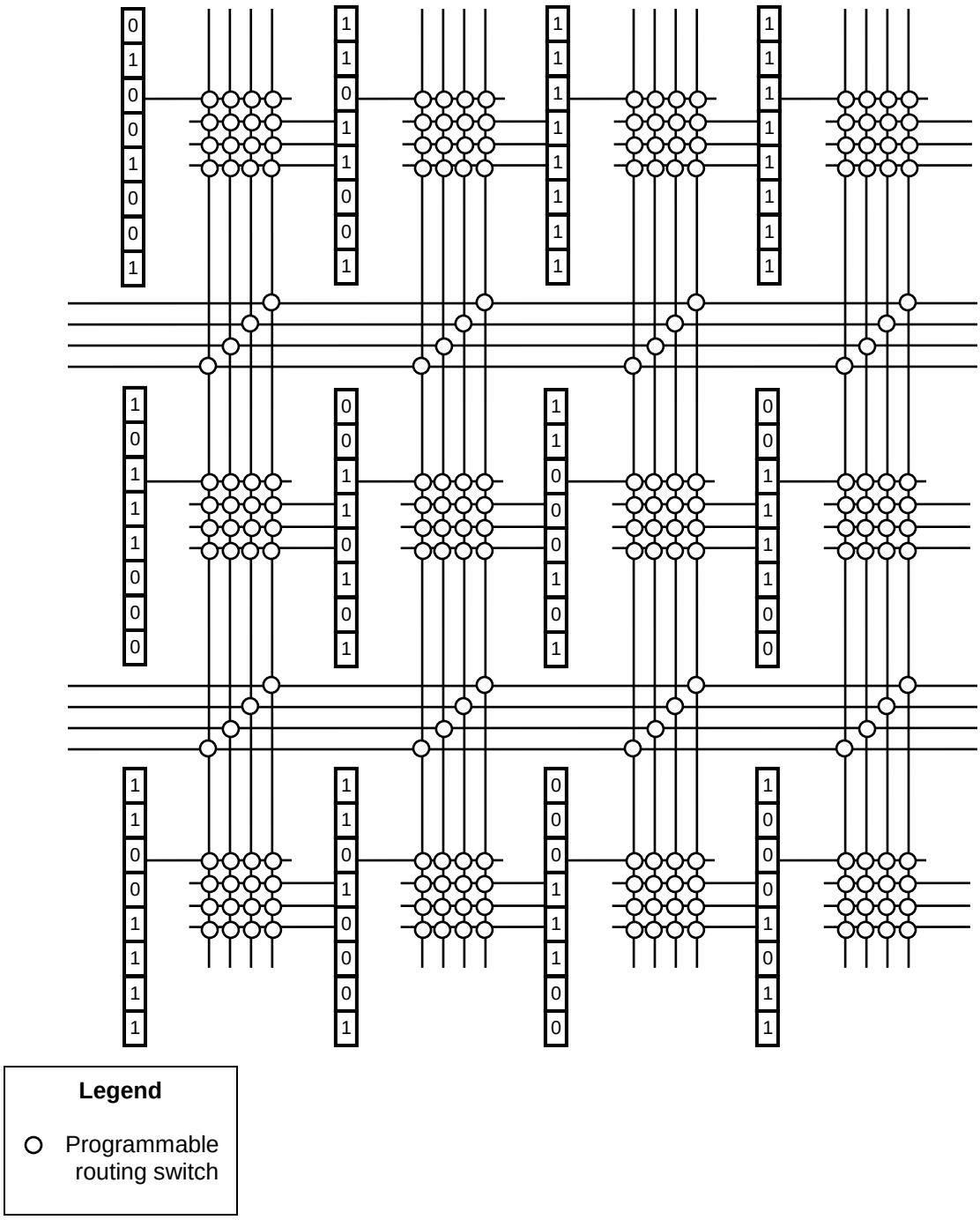



**Figure 3. Simplified view of an FPGA**

In addition to LUTs, switches in routing channels have storage cells. Reconfiguration of the LUTs' interconnection network is facilitated by storing appropriate logic values in these cells. The stored logic values turn the switches on or off to make or break contacts between the interconnecting wires as needed and eventually set up an interconnection network that works for all the circuit components. These wires/switches can also provide the circuit with necessary connections to the input and output pins of the FPGA to get connected to the outside world. Therefore, to get an FPGA operating as intended, the storage cells of LUTs as well as the storage cells of switches must receive appropriate logic values. The process of

placing appropriate logic values into these storage cells is called FPGA *programming or configuration*. The Quartus software to be used in this book takes care of FPGA programming among other things.

On the hardware side, we will use the Altera DE10-Lite Development and Education board, which features a powerful but low-cost Max10 FPGA chip with 4-input LUTs. The board has to be connected to the computer that runs the Quartus software.

The first step in an automated design cycle is to *describe* or *model* the intended circuit in a format that is understandable to the CAD tool. This description is called a *design entry*, which can be in graphical format (logic diagram), textual format, or a combination of the two. In this book, we will focus on the textual format. A hardware description language (HDL) must be used to prepare a textual design entry. VHDL and Verilog are the two most frequently used languages. In this book, we will use VHDL, which stands for "very-high-speed integrated circuits hardware description language". The Quartus software takes the VHDL code and programs the FPGA chip accordingly. We may also say the Quartus software *maps* the (VHDL) design entry into the FPGA chip.

Now you are familiar with two different technologies to build logic gates, hence logic circuits: In this chapter, you learned how to build logic gates using LUTs. In Chapter 2, you learned how to build standard CMOS logic gates using transistors. Remember that CAD tools can translate a VHDL code into either of these two technologies. As an example, consider the following VHDL code line, which describes a 2-input NAND gate with inputs A and B, and output Y:

```
Y <= A NAND B;
```

Figure 4 illustrates the two different implementations. In Figure 4*a*, a 2-input LUT is used. Figure 4*b* shows the standard CMOS implementation of this gate.

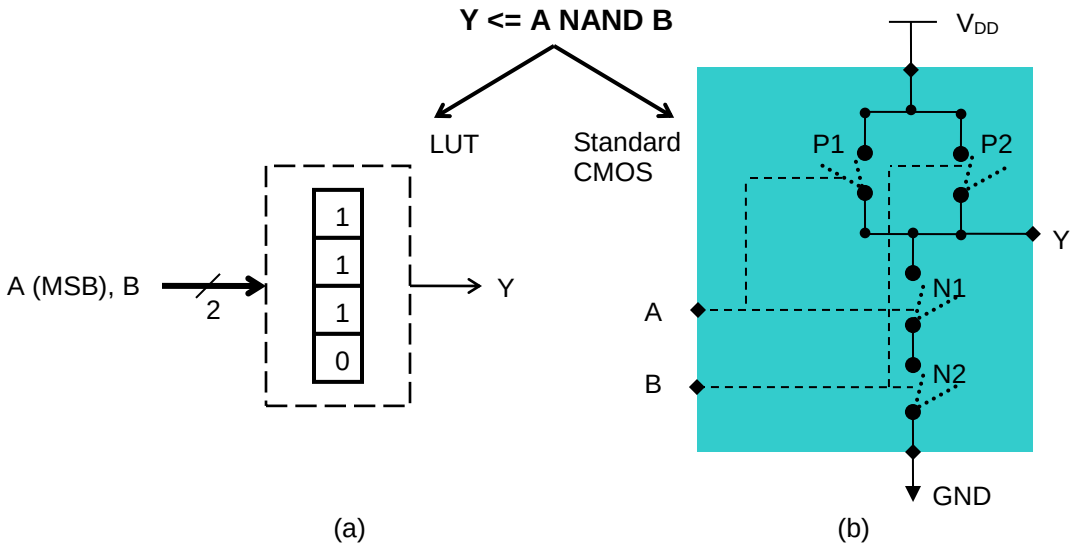



**Figure 4. Two target technologies for a VHDL code describing a 2-input NAND gate**

It is very important to understand that LUTs and routing switches are made up of transistors as well. However, their transistor-level architecture is out of the scope of this book.

**Example:** Use a 3-input LUT to implement a 2-input NAND.

Figure 5*a* illustrates the programmed LUT. The logic symbol of the gate is depicted in Figure 5*b*.

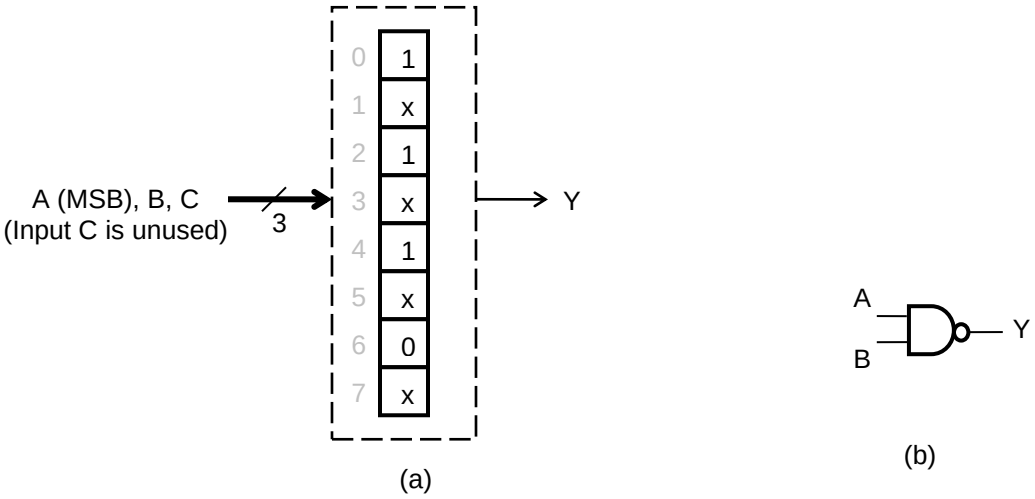



**Figure 5. A 2-input NAND: (a) 3-input LUT implementation, (b) logic symbol**

Note that the 3-input LUT in this design has one too many inputs. Let us assume that the unused input, C, is tied to logic 0. This makes the contents of odd-address cells inaccessible. This is why each of these cells is marked with a don't care.

**Example:** Use a 3-input LUT to implement the diagram shown in Figure 6.

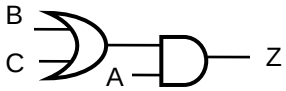



**Figure 6. Logic diagram to be implemented with a 3-input LUT**

In Chapter 3, Part I, you learned how to obtain a logic expression for output Z and then how to draw a truth table for Z:

$$Z = A \cdot (B + C)$$

The truth table of Z is shown in Figure 7*a* and its LUT implementation is depicted in Figure 7*b*.

| Row | A B C | Z |
|---|---|---|
| 0 | 0 0 0 | 0 |
| 1 | 0 0 1 | 0 |
| 2 | 0 1 0 | 0 |
| 3 | 0 1 1 | 0 |
| 4 | 1 0 0 | 0 |
| 5 | 1 0 1 | 1 |
| 6 | 1 1 0 | 1 |
| 7 | 1 1 1 | 1 |

(a)

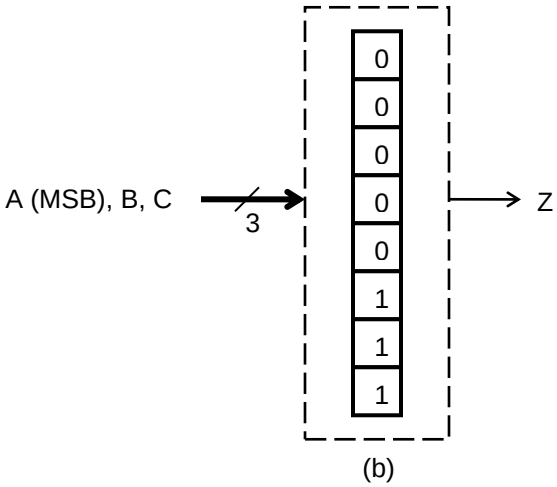


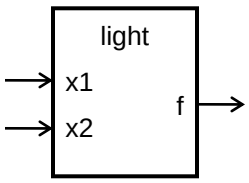

**Figure 7. Function Z = A . (B + C): (a) truth table, (b) LUT implementation**

**Introduction to VHDL Modeling**

A VHDL code describing a logic circuit has two parts: *Entity* and *Architecture*. The entity only tells us what the inputs and outputs of the circuit are. It does not say anything about the inside of the circuit. So, we may call a logic symbol (with no specific shape) a graphical version of the entity. As an example, a textural ENTITY is shown in Figure 8*a*. Its graphical version (logic symbol) is illustrated in Figure 8*b*. The underlying circuit has two inputs, x1 and x2, and one output, f.

```
ENTITY light IS
PORT (x1, x2 : IN     STD_LOGIC;
          f  : OUT    STD_LOGIC
      );
END light;
```

(a)




(b)

**Figure 8. An ENTITY: (a) textual, (b) graphical**

Every ENTITY has an arbitrary name. You would of course prefer/choose meaningful names. In this example, "light" is the ENTITY's name. You will soon find out why this ENTITY is called "light". As shown in the first line of the code, the ENTITY's name is typed between two keywords, namely ENTITY and IS.

Keyword END followed by the ENTITY's name indicates the end of ENTITY.

Keyword PORT is used to introduce the inputs and outputs. Take a close look at the code and see what the syntax is: x1 and x2 are followed by the keyword IN, and f is followed by the keyword OUT. So, x1 and x2 are the two inputs and f is the only output of this circuit. Additionally, the keyword STD_LOGIC signifies that each input is **one** bit wide and so is the output. The way that parentheses, colons, and semicolons are used in an ENTITY is clearly illustrated in Figure 8*a*. Follow the same format in your codes.

Let us assume that the circuit under consideration is a 2-input XOR gate. As you know, two binary switches along with such a gate can independently turn on a light if it is off, and turn it off if it is on. This is why we call the ENTITY "light". The ARCHITECTURE of such a circuit is shown in Figure 9.

```
ARCHITECTURE LogicFunction OF light IS
BEGIN
    f <= (x1 AND NOT x2) OR (x2 AND NOT x1);
END LogicFunction;
```

**Figure 9. ARCHITECTURE for ENTITY light**

Every ARCHITECTURE has an arbitrary name. You would of course prefer/choose meaningful names. In this example, "LogicFunction" is the ARCHITECTURE's name, which follows the keyword ARCHITECTURE. The name "LogicFunction" tells us that the circuit is described *algebraically*. In the coming chapters, you will learn other techniques to describe circuits in VHDL. Look at the first line again

and see how the keyword OF, the ENTITY's name (light) and the keyword IS follow the ARCHITECTURE's name.

The logic function that describes the circuit is located between the Keywords BEGIN and END, as shown in Figure 9. Note that the END keyword is followed by the ARCHITECTURE's name.

The function described in Figure 9 is a 2-input exclusive OR, which is shown here again for ease of reference:

```
f <= (x1 AND NOT x2) OR (x2 AND NOT x1);
```

You may wish to reword it as

```
f <=  x1 XOR x2;
```

The way that a logic expression is assigned to a signal in Figure 9 is called a *simple signal assignment*. Keep in mind that the assignment operator in VHDL, <=, consists of two characters, namely the "less than sign" followed by the "equal sign". The logic expression sits on the right side of the assignment operator. On the left side, you will place the signal that represents the output of the corresponding function or circuit (signal f in Figure 9). You may algebraically describe digital circuits using simple signal assignments. You will learn more complex (hence powerful) signal assignments in the coming chapters.

**Vectors versus Bits**

A signal can be wider than one bit. Such a signal is called a *vector*. Keyword STD_LOGIC_VECTOR is used to define a vector. In the following example, A, B, and Y each are defined as an 8-bit vector:

```
ENTITY vector_operation IS
PORT (   A, B :     IN        STD_LOGIC_VECTOR (7 DOWNTO 0);
            Y    :     OUT       STD_LOGIC_VECTOR (15 DOWNTO 8)
        );
END vector_operation;
```

Pay close attention to see how the size of a vector is specified and how indexes are assigned to different bits in a vector. Following the keyword STD_LOGIC_VECTOR, two numbers are written and separated by the keyword DOWN in parentheses. From left to right, these two numbers are the upper and lower bounds (limits) of the vector, respectively. The bounds also indicate the vector's length. For example, the upper bound of vector Y is 15 and the lower bound is 8. So, vector Y is 8-bits long (15 – 8 + 1 = 8) and consists of the following single bits from left to right: Y(15), Y(14), Y(13), Y(12), Y(11), Y(10), Y(9) and Y(8). As you see, every single bit of vector Y may explicitly be referenced by the vector's name, Y, followed by the bit's index inside parentheses.

Look at the following architecture and see how vectors may make hardware programming much easier:

```
ARCHITECTURE Algebraic OF vector_operation IS
BEGIN
    Y <=  A XOR B;
END Algebraic;
```

You should note that the vector assignment Y <= A XOR B is equivalent to the following eight single-bit assignments:

```
Y(15) <= A(7) XOR B(7);
Y(14) <= A(6) XOR B(6);
```

```
Y(13) <= A(5) XOR B(5);
Y(12) <= A(4) XOR B(4);
Y(11) <= A(3) XOR B(3);
Y(10) <= A(2) XOR B(2);
Y(9) <= A(1) XOR B(1);
Y(8) <= A(0) XOR B(0);
```