1 Introduction: What Is an FPGA, What Is High-Level Synthesis or HLS?

Abstract

This chapter shows what an FPGA is and how it is structured from Configurable Logic Blocks or CLB (in the Xilinx terminology, or LAB, i.e. Logic Array Blocks in Altera FPGAs). It also shows how a hardware is mapped on the CLB resources and how a C program can be used to describe a circuit. An HLS tool transforms the C source code into an intermediate code in VHDL or Verilog and a placement and routing tool builds the bitstream to be sent to configure the FPGA.

1.1 What Hardware to Put in an FPGA?

A processor is a hardware device (an *electronic circuit*, or a *component*), able to calculate everything which is *computable*, i.e. any calculation for which there is an *algorithm*. It is enough to transform this algorithm into a program and apply the processor to this program with data to obtain the result of the calculation.

For example, a processor can run a program turning it into a tremendous calculator, capable of carrying out all the mathematical operations. For this, the processor contains a central component, the Arithmetic and Logic Unit (ALU). As its name suggests, this electronic circuit unit within the processor performs basic arithmetic and logic operations. In the ALU, the calculation is the propagation of signals passing through *transistors* (a transistor is a kind of small switch or tap which you open or close to give way or block the current; a transistor has three ends: the input, the output, and the open/close command).

The ALU itself contains an *adder*, i.e. a circuit to do binary additions.

How can transistors be assembled in a way which, by supplying the 0s and 1s of the binary representation of two integers as input, obtain the 0s and 1s of the binary representation of the result at the output of the transistor network?

When the solution to the problem is not directly apparent, we must break it down to bring us back to a simpler problem. In school, we learned that to add two decimal

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_1

```
 1 110          1 1100 111
   8231           1010 0111
 + 1976         + 0111 0001
 1 0207         1 0001 1000
```

Fig. 1.1 Decimal addition (left), binary addition (right). The two additions are applied to different numbers

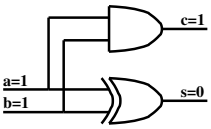

Fig. 1.2 A pair of gates representing a binary adder circuit

numbers is to go from right to left—from units to tens, hundreds, etc.—by adding the digits of the same rank and by propagating the carry to the left. In doing so, two digits are produced at each step (each rank). One is the sum modulo 10 and the other is the carry (see the left part of Fig. 1.1; the line in red is the carries one; the two numbers to be added are in brown; the result is composed of the sum in blue and the final carry in green).

We thus reduced our general problem of the addition of two numbers to the simpler problem of the addition of two digits.

If the digits are binary ones (these are *bits*, abbreviation of *binary digits*), the addition process is identical. The sum is calculated modulo 2 and there is a carry as soon as at least two of the three input bits are 1s (see the right part of Fig. 1.1).

The modulo 2 sum of two bits and their carry can be defined as Boolean operators.

The modulo 2 sum is the exclusive OR (or XOR) of the two bits (the symbol $\oplus$) and the carry is the AND (the symbol $\wedge$).

Notice that $a \oplus b$ is 1 if and only if a and b are different (one of the two is a 1 and the other is a 0). Similarly, $a \wedge b$ is 1 if and only if a and b are both 1s.

The transition to Boolean operators is a decisive step towards a hardware representation of the binary addition because we know how to build, based on transistors, small constructions which form logic *gates*, i.e. Boolean operators. Thus, we find NOT, AND, OR, XOR gates, but also NAND, NOR, and all the Boolean functions of two variables.

Our binary adder becomes the pair of gates in Fig. 1.2. The top gate is an AND and the bottom gate is an XOR. The figure shows the operation of the circuit when two inputs are provided with the same value 1.

To build a circuit, we must draw the gates which compose it or write a program in a hardware description language (HDL) like VHDL or Verilog. The code in Listing 1.1 is the VHDL interface and implementation of the adder shown in Fig. 1.2.

Listing 1.1 A VHDL function defining a 1-bit adder

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity BIT_ADDER is
  port(a, b: in  STD_LOGIC;
       s, c: out STD_LOGIC);
end BIT_ADDER;
architecture BHV of BIT_ADDER is
begin
```

```
  s <= a xor b;
  c <= a and b;
end BHV;
```

A succession of softwares transform these gates or this program into transistors then organize them on a *layout*. Finally, a founder physically builds the circuit.

Instead of relying on an electronic foundry, we can use an FPGA. Its name indicates that it is more or less an array of gates which can be *programmed*. The programming consists of linking these gates into subsets forming small circuits within the large circuit which houses them.

1.2 Look-Up Table (LUT): A Piece of Hardware to Store a Truth Table

We want to add two 1-bit words, i.e. $s = a + b$, where a and b are Boolean variables. The sum s is a 2-bit word, composed of the modulo 2 sum and the carry bit.

For example, if the pair (a, b) is (1, 1), their sum s in binary is 10, which is the concatenation of the carry bit (1) and the modulo 2 sum (0).

Let us first concentrate on the modulo 2 sum.

We can define the modulo 2 sum as a Boolean function of two variables, which truth table is presented as Table 1.1, where the arguments of the function are in blue and the values of the function are in red.

For example, the last line of the table says that if $a = b = 1$ then $s = 0$, i.e. $s(1, 1) = 0$.

A LUT (acronym of Look-Up Table) is a hardware device which is similar to a memory. This memory is filled with the truth table of a Boolean function. For example, a 4-bit LUT (or a LUT-2), can store the truth table of a Boolean function of two variables (the red values in Table 1.1).

More generally, a 2^n-bit LUT (or a LUT-n), can contain the truth table of a Boolean function of n variables (where you can fit 2^n truth values).

For example, NOT is a single-variable Boolean function. It is represented by a two line truth table and a LUT-1, i.e. two truth values (NOT(0)=**1** and NOT(1)=**0**).

The two-variable Boolean function AND can be extended to three variables (a AND b AND c). The truth table has eight lines, as shown in Table 1.2.

Table 1.1 The truth table of the modulo 2 sum of two 1-bit words

a	b	s
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.2 The truth table of the operator a AND b AND c

a	b	c	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 1.3 The truth table of the operator "IF a THEN b ELSE c"

a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The operator "IF a THEN b ELSE c" is another example of a three-variable Boolean function. It is represented by the eight truth values in Table 1.3. The function "IF a THEN b ELSE c" is c if a is 0 (the values in cyan), b otherwise (the values in orange).

The LUT, like the truth table, is a kind of universal Boolean function. By filling it, you define the Boolean function it contains.

In an FPGA, a LUT is represented by a memory, i.e. an addressable hardware device. By addressing the memory, we obtain what it contains at the provided address.

Figure 1.3 shows how, from $a = 1$ and $b = 1$ inputs, meaning the LUT address 3 (**11** in binary), the bit contained in the memory cell at the bottom right is accessed. The address is split into two halves. One half (a input) serves as a row selector and the other half (b input) as a column selector.

In Fig. 1.3, the memory is addressed with $a = 1$, which selects the line framed in red. It is also addressed with $b = 1$, which selects the column framed in green. The intersection of the chosen row and column provides the output s of the LUT.

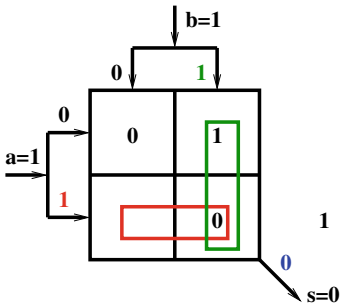

Fig. 1.3 Accessing the address 3 cell (**11** in binary) of the LUT to compute $s = (1 + 1)$ modulo 2

1.3 Combining LUTs

Let us continue our construction of an adder. This time, let us try to build a *full adder*, that is, a hardware cell calculating the modulo 2 sum of two bits and an input carry. The *carry* is a new input variable which extends the truth table from four to eight rows. The modulo 2 sum of the three input bits a, b, and c_i is their XOR ($a \oplus b \oplus c_i$).

Rather than being built with a LUT-3, the full adder is built with two LUT-2. It calculates not only the modulo 2 sum of its three inputs but also their carry.

For example, if $a = 0$ and $b = 1$, the modulo 2 sum is $s = 1$. But if there is a carry in, say $c_i = 1$, then the modulo 2 sum is $s = 0$ and there is a carry out $c_o = 1$.

A first LUT-2 is used to store the truth table of the Boolean function *generate*. As its name suggests, the *generate* function value is 1 when the sum of the two sources a and b generates a carry, i.e. when $a = b = 1$, meaning $a \wedge b$ is true (a AND b).

A second LUT-2 stores the truth table of the Boolean function *propagate*. The *propagate* function value is 1 when both sources a and b can propagate an inbound carry but cannot generate one, which happens when either of the two sources is a 1 but not both simultaneously. The *propagate* function is the XOR ($a \oplus b$).

We link the two tables as shown in Fig. 1.4. In the figure, the addressed table entries are shown in red. The values out of the LUTs are propagated to the mux box and to the XOR gate on the right.

The box labeled mux is a *multiplexer*, that is, equivalent to the Boolean function "IF x THEN y ELSE z". The input coming from the left side of the box is the x selector. The entries coming from the lower edge of the box are the choices y (the right input) and z (the left input). If the selector x is a 0, the choice on the left (z) is found at the output. Otherwise, it is the right choice which crosses the multiplexer (y).

Thus, if $a = 0$ and $b = 1$, the *propagate* function value is 1 and the *generate* function value is 0 (these values are in red in the figure). If the incoming carry c_i is 1, the multiplexer selector (1) passes the right choice c_i and the outgoing carry is $c_o = 1$.

The structure of the prefabricated circuit composed of the two LUTs and the two gates brings out the *propagation* of the input carry c_i towards the output c_o when the multiplexer chooses its right input, i.e. when *propagate* is 1. This carry propagation

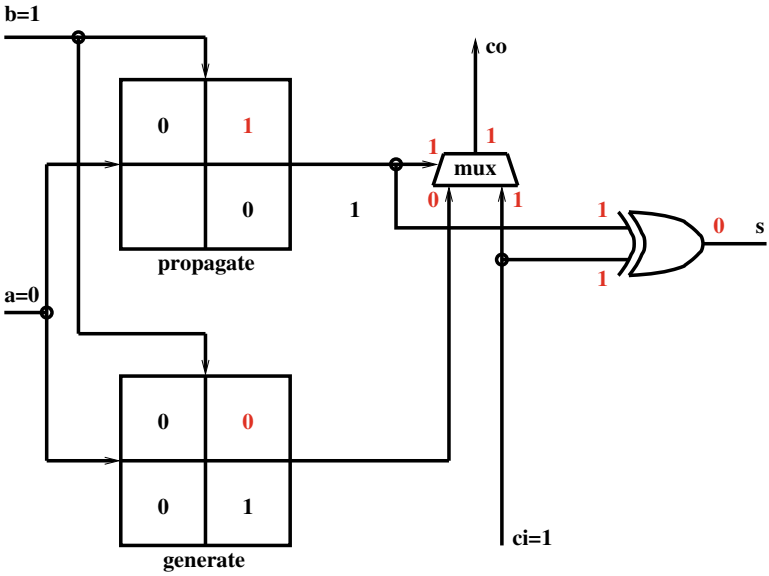

Fig. 1.4 A full adder with two LUT-2

mode is very efficient because the input signal c_i is found at the output after a single gate crossing.

The gate on the right of the figure is an XOR. It produces the modulo 2 sum of *propagate* and c_i. Since *propagate* is itself an XOR, the output s is the modulo 2 sum of the three inputs a, b, and c_i ($a \oplus b \oplus c_i$).

In the example presented, the combination of the two LUTs, the multiplexer and the XOR gate calculates two bits, one representing the modulo 2 sum of the three inputs and the other being the outgoing carry. By sticking these two bits together, we form the 2-bit sum of the three inputs ($0 + 1 + 1 = 10$ in binary).

The organization of the full adder proposed above corresponds to what is found in a CLB, i.e. a Configurable Logic Block, which is the basic building block of the FPGA (in [1], pages 19 and 20, you have the exact description of the CLBs you find in the FPGA used in this book).

A CLB combines a LUT with a fast carry propagation mechanism. It is a kind of Swiss army knife, which computes logic functions with the LUT and arithmetic functions with the carry propagation.

The programming or *configuration* of the CLB is the filling of the LUTs with the truth values of the desired Boolean functions (in the example, the *propagate* and *generate* functions; the Swiss army knife may divide the LUT in two halves to install two Boolean functions).

1.4 The Structure of an FPGA

Let us try to extend our adder from a 1-bit word adder to a 2-bit word adder.

Let $A = a_1a_0$ and $B = b_1b_0$, for example $A = 10$, with $a_1 = 1$ and $a_0 = 0$ (i.e. $1 * 2^1 + 0 * 2^0$, or 2 in decimal), and $B = 01$ (i.e. $0 * 2^1 + 1 * 2^0$, or 1 in decimal). The sum $A + B + c_i$ is the 3-bit word $c_os_1s_0$ (for example $10 + 01 + 1 = 100$ or in decimal $2 + 1 + 1 = 4$).

By combining two CLBs configured as a full adder, with the output of the first (c_{o0} in Fig. 1.5) connected to the input of the second (c_{i1} in Fig. 1.5) and inputs a_0 and b_0 for the first and a_1 and b_1 for the second, three bits are output, forming the $c_os_1s_0$ sum of the two 2-bit words $A = a_1a_0$ and $B = b_1b_0$ and an incoming carry c_i.

Figure 1.5 shows this 2-bit adder.

An FPGA contains a matrix of CLBs (see the left part of Fig. 1.6).

For example, the Zynq XC7Z020 from Xilinx is a SoC (System-on-Chip, i.e. a circuit containing several components: processors, memories, USB and Ethernet interfaces, and an FPGA of course) whose programmable part (the FPGA) contains 6650 CLBs.

One can imagine that the CLBs are organized in a square of more or less 80 columns and 80 rows (the exact geometry is not described). For a detailed presentation of FPGAs (including their history), you can refer to the Hideharu Amano book [2].

Each CLB contains two identical, parallel and independent SLICEs (centre part of Fig. 1.6). Each slice mainly consists of four LUT-6 and eight flip-flops (the red squares labeled FF—for Flip-Flop—in the centre and right part of Fig. 1.6). Each flip-flop is a 1-bit clocked memory point, which collects the output of the LUT.

Each LUT-6 can represent a six-variable Boolean function or can be split into two LUT-5, each representing a five-variable Boolean function.

The LUTs of the same slice are linked together by a carry propagation chain identical to that of Figs. 1.4 and 1.5 (including a multiplexer and an XOR gate).

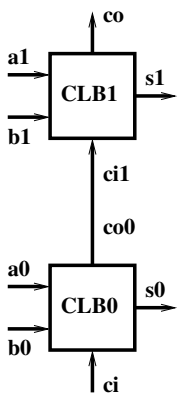

Fig. 1.5 A 2-bit adder built from two CLBs

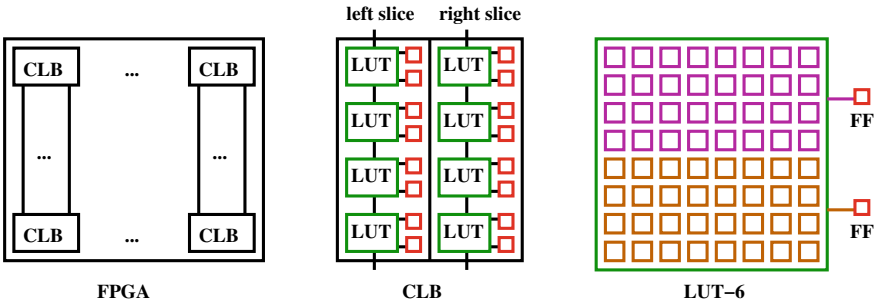

Fig. 1.6 The structure of an FPGA

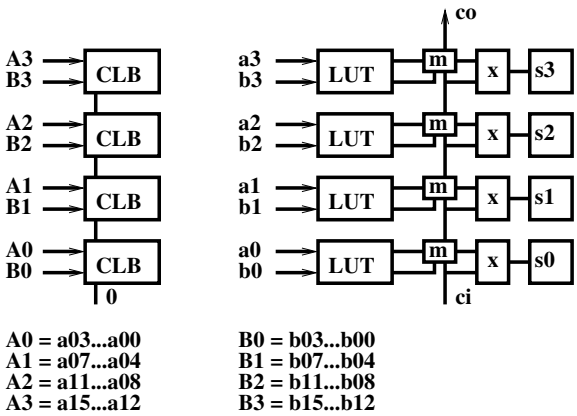

Fig. 1.7 A 16-bit adder built with four CLBs

A LUT can be partially filled. It may contain only four useful bits out of the 64 available to represent a Boolean function with two variables. But it is better not to waste this resource.

By continuing to extend the 2-bit adder, one can build an adder of any size.

A 16-bit adder links four CLBs of a single column, using 16 LUTs in series, each LUT containing the two tables in Fig. 1.4 (notice that the adder uses only one of the two available slices in each CLB).

The left part of Fig. 1.7 shows how the two 16-bit words to be added are distributed by nibbles in the four CLBs (for exemple the $A_3 = a_{15}a_{14}a_{13}a_{12}$ nibble inputs the highest CLB in the figure). The first CLB in the chain (the lowest in the figure) receives an input carry entry $c_i = 0$.

The right part of the figure shows the addition of the first nibbles $(a_3a_2a_1a_0 + b_3b_2b_1b_0)$ in the LUTs (each LUT is split into two half LUTs, the first containing the *generate* function and the second containing the *propagate* function). The multiplexers (boxes labeled m) propagate the carry from the input c_i to the output c_o. The XOR gates (boxes labeled x) provide the modulo 2 sum bits, which may be stored in the flip-flops (rightmost boxes, labeled s_0 through s_3).

1.5 Programming an FPGA

An FPGA is a *programmable* circuit. How do you program a circuit?

A circuit is programmable when it is given two successive operating modes.

The first mode is the initialization. Storing structures are filled with initial values. Once this phase has been completed, the second operating mode starts. It is the computation, which uses the values installed during the initialization.

The initialization phase of an FPGA fills the LUTs with the truth values of the Boolean functions which they represent.

It is also necessary to link the CLBs participating in the same calculation (for example, clear the c_i input of the first CLB of the adder, and link its c_o output to the next CLB c_i input).

This link phase is done with multiplexers (for example to choose c_i between 0 and c_o). The initialization of the links sets the selection bit of the multiplexer.

All the resources of the FPGA (LUTs and inter CLB links) are thus initialized from a sequence of bits which constitutes a *bitstream*.

This sequence of bits is sent from a programming station (i.e. your computer). It comes in the FPGA bit by bit (known as *serial transmission*). Each bit takes its corresponding place in the FPGA.

Once this phase is completed, the FPGA enters the computation phase which performs the function assigned to it by the configuration phase.

In practice, the programming phase lasts a few seconds.

One question remains: how do you go from a function to its hardware implementation on the FPGA?

In the early days of FPGAs (i.e. in the mid-80s), gates were drawn as in Fig. 1.2. A translator was in charge of placing these gates in LUTs.

Soon enough, FPGAs became big and complex enough to implement units which became difficult (and unsafe) to define in schematics.

Hardware description languages (HDL) were proposed (VHDL, Very high speed integrated circuits HDL, and Verilog are the two most used languages).

Rather than drawing gates, we describe a circuit behaviour with a hardware-specific language and a compiler transforms the source description into a bitstream.

An HDL program exactly says how a circuit should behave, from its inputs to its outputs, through binary operations or subroutine calls. In addition to the calculation, the program also expresses the temporality of the signals—i.e. the variables of the program—related to their propagation time in the circuit.

In the mid-90s, a higher level method was proposed, based on classic programming languages (like C or C++). The temporality is left as a job for a translator. The idea was to define a circuit by a program and let the translator implement the corresponding component with CLBs. This method is HLS (High-Level Synthesis; *synthesis* is the name given to the construction of the circuit from a transformation of the source program; the translator is a *synthesizer*).

For example, the C function shown in Listing 1.2 builds a 32-bit adder.

Listing 1.2 A function defining a 32-bit adder

```
void adder_ip(unsigned int  a,
              unsigned int  b,
              unsigned int *c){
  *c = a + b;
}
```

HLS transforms this C code into an intermediate representation (one of them is the RTL or Register Transfer Level representation, which is used in HDL programs; from the RTL, a VHDL or a Verilog program can be built).

A placement and routing software maps the RTL on CLBs (placement phase), then associates these CLBs with those of the FPGA by minimizing the propagation times and sets up the necessary links (routing phase).

The placement and routing leads to the constitution of the bitstream which is lastly transmitted to the FPGA through a USB link (Universal Serial Bus).

The next chapter is devoted to the installation of the HLS, placement and routing softwares, and their application to the adder example.

References

1. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
2. H. Amano, *Principles and Structures of FPGAs* (Springer, 2018)