Field Programmable Gate Array

Muhammed Fahri UNLERSEN
Necmettin Erbakan University

What is an FPGA?

FPGA is a term formed by combining the first letters of the word Field-Programmable Gate Array. The reason for using the term "field programming" is that the function of the FPGA integrated circuit (IC) is not programmed at factory output and is an IC that can be changed while in the field. The function mentioned here is a task created with the hardware architecture of IC. It has grown very rapidly since the FPGA term was introduced. While growing at a high rate in terms of capacity and performance, the decrease in cost per unit operation has made FPGAs remarkable (Unlersen, 2015). In Figure 1, an FPGA IC is presented.

Figure 1. An FPGA IC belongs Xilinx Company

Although Xilinx presented the first hardware that can be called FPGA in 1984, the term FPGA became popular in 1988 with the company Actel. The non-recurring engineering cost required for application-specific integrated circuit (ASIC) fabrication does not exist in FPGAs. But, this situation made FPGAs advantageous only in the use of a low number of units. In this process, ASICs were more popular because they were very low cost compared to FPGAs in high production. However, according to Moore's law, the prediction that the number of units that FPGA will be advantageous will increase in the future has prevented the interest in FPGAs from decreasing. Today, performance, I/O capacity, power consumption, time to market and other capabilities are more important than device cost in FPGA-ASIC comparison (*FPGA Designs with VHDL Documentation*,

n.d.; Trimberger, 2015; *What Is an FPGA? Field Programmable Gate Array*, n.d.).

Some of the application areas of FPGAs can be listed as follows (Rajewski, 2017):

- Aerospace
- Defense
- Automotive
- High Performance Computing and Data Storage
- Data Center
- Industrial
- ASIC Prototyping
- Broadcast
- Video and Image Processing
- Wired and Wireless Communications
- Medical Imaging
- Security

In the design of an embedded system, the question of which platform should be designed first comes to mind. Because for the designer, there are many different hardware such as microcontrollers, ASIC, microcomputer, FPGA. Actually, FPGA is not a one-to-one alternative to other microprocessor-built platforms. On an FPGA, a hardware to perform the required operation can be designed. However, in systems created with a microprocessor, commands that will perform a desired operation are executed on a fixed hardware. Additionally, it is also possible to design a microprocessor with an FPGA.

The designer's choice of FPGA among these alternatives depends on the needs of the system to be designed rather than a matter of whim. For example, in a hardware where the algorithm to be used will change frequently and operations such as multiplication and division with complex numbers will be made frequently, using a DSP produced for this purpose may be more logical than using an FPGA. Because it will be very simple and flexible to make calculations on this DSP using a high-level language such as C. In a platform that should be cheap rather than high performance, choosing a microcontroller can be a fast, simple and satisfying solution. However, if the process requires high performance and speed, then FPGA will be more suitable for this type of applications

(Leong et al., 1998; Rajewski, 2017).

FPGA Structure

FPGAs are semi-ready silicon devices that can be electrically programmed to be part of a digital circuit or system. Its structure can be defined in three main parts: programmable logic blocks, input and output blocks surrounding this block array, and interconnections (Chu, 2008; Gunes & Ors, n.d.).

The basic FPGA structure consists of thousands of basic elements called Configurable Logic Blocks (CLB). These basic structures can be called Logic Blocks (LB), Logic Elements (LE) or Logic Cells (LC) according to the manufacturer (Gunes & Ors, n.d.). CLBs are formed by combining a set of logic elements such as a LookUp Table (LUT) and flip flops (FF). The hardware architecture of the FPGA consists of the data stored in these LUTs. The working principle of a 4-input LUT is illustrated in Figure 2.

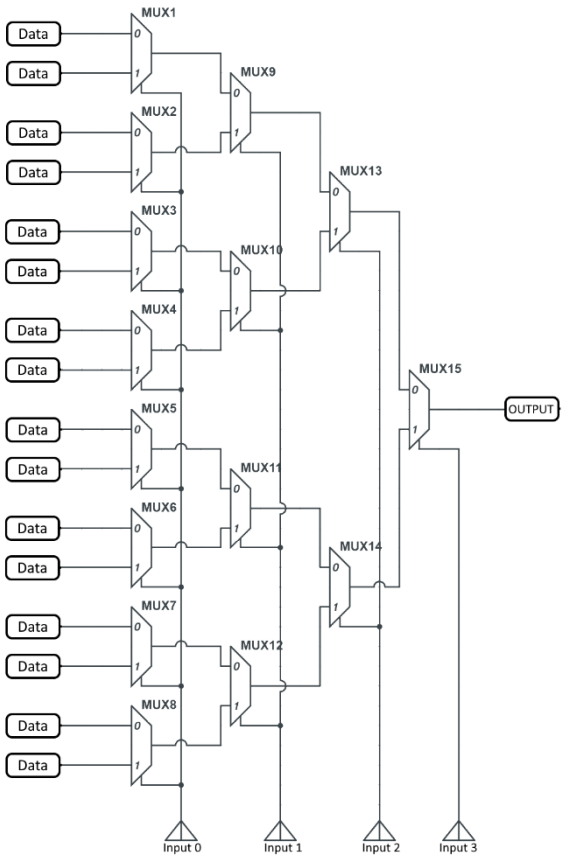

Figure 2. A 4-Input LUT Structure

The values specified here, as Data are the data loaded during FPGA programming. According to this loaded data, the value applied to the inputs is selected and transferred to the output. Thus, this LUT fulfills its special mission.

Although the LUT and the used logic elements that make up the CLB differ from company to company, an example CLB structure is presented in Figure 3.

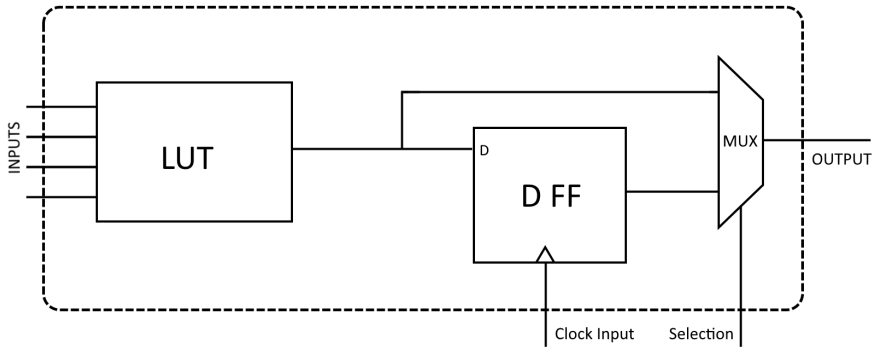

Figure 3. An example CLB Structure

In the CLB shown, a LUT output is selected with a multiplexer according to synchronous or asynchronous structure, while a D-type FF is activated in synchronous use.

The term programmability in FPGAs means that LUT information and other logic elements can be controlled according to the designed system after production. The programming mentioned here is not the commands operated by a hardware as in microcontrollers, but the codes that define the hardware.

There are interconnections around CLBs. Interconnects allow these blocks to be programmed and communicated with other blocks (Chu, 2008).

The communication of FPGAs with the outside world is provided by the input and output (I/O) blocks. These blocks can be configured in different directions as input or output. Today, the I/O blocks of FPGAs support 500MHz operating frequency. Also, some of the I/O blocks have the ability to read data on both falling and rising edges (*What Is an FPGA? Field Programmable Gate Array*, n.d.; *What Is an FPGA? Programming and FPGA Basics - INTEL*, n.d.).

Pins on FPGA IC are divided into 2 categories (Chu, 2008; Gunes & Ors, n.d.; Rajewski, 2017).

- Dedicated pins: Pins with special tasks on the FPGA are called dedicated pins. They are divided into three groups according to their functions.
    - Power pins: They provide the power needed for the IC to work.
    - Configuration pins: They are used to download the program created in the PC software to the IC.
    - Clock pins: These are the pins specially set to receive the simultaneous clock signals in IC.
- User pins: These are the standard pins that can be set as Input, Output or Input-

Output. Each pin has its own I/O cell. This cell determines in which mode a pin will work.

Synchronous design is an important area in logic circuit designs. Such designs are clock-based. This is also true for FPGA. However, the most important issue in synchronous design is that the clock signal reaches all logic elements simultaneously. Otherwise, timing problems and even electrical problems will arise. FPGA manufacturers have overcome such problems with a connection structure called Global Routing or Global Line for clock signals. Thanks to these connections, the clock signal reaches all CLBs simultaneously. For this reason, clock feeds must be made from the pins of the FPGA that are reserved as clock (Chu, 2008; Gunes & Ors, n.d.; Rajewski, 2017; Unlersen et al., 2018).

There are PLL blocks that can generate the high frequencies which are needed internally within the FPGA. These blocks can generate the necessary operating frequencies up to 500MHz, taking the frequencies from the clock input of the FPGA as reference, usually around 50MHz (Unlersen, 2015).

FPGAs contain RAM units in blocks. These memories are used for data storage processes that CLBs will need during their operations. These RAMs support both single access and multiple access. Multiple applications accessing the same RAM at the same time is called multiple access. While these block RAMs meet the large memory needs of CLBs, there are distributed RAMs interspersed around the CBLs for small memory needs (Chu, 2008; Gunes & Ors, n.d.; Rajewski, 2017).

Programming FPGAs

VHDL and Verilog are used in FPGA programming. The prepared program is loaded onto the FPGA IC with the Joint Test Action Group (JTAG) protocol. JTAG is an IEEE Standard 1149.1-1990 that was created in the 1980s to eliminate errors in the production of electronic cards (IEEE Standards Board. & IEEE Computer Society. Test Technology Technical Committee., 1993). Most FPGAs do not have an internal EEPROM. The loaded program is stored in SDRAM cells. In other words, the program loaded with JTAG is not permanent. Therefore, the program must be reloaded each time the FPGA is re-energized. For this reason, they are designed with an external EEPROM right next to the point where the FPGA ICs are located (*What Is an FPGA? Field Programmable Gate Array*, n.d.; *What Is an FPGA? Programming and FPGA Basics - INTEL*, n.d.).

VHDL - Very High-Speed Integrated Circuit Hardware Description Language

Hardware description languages are used in FPGA programming. These are VHDL and Verilog. Verilog uses a textual format to describe electronic systems. In the field of

electronic design, Verilog can be used for verification through simulation for testability analysis, error grading, logic synthesis and timing analysis. Verilog has the IEEE 1364 standard. Design is performed with fewer commands. In terms of structure, it is a language that is often compared to C. However, Verilog is not as wordy as VHDL due to its nature. Therefore, VHDL is more capable of creating hierarchical structures (Gunes & Ors, n.d.; Nageswaran, 1997).

Details of VHDL will be given here. VHDL is a widely used hardware description language for designing and testing digital circuits. VHDL stands for Very high speed integrated circuit Hardware Description Language (Unlersen, 2015).

The most important feature of VHDL is that designs can be divided into components in a hierarchical way. Each design element should have a well-defined interface. There must be faultless behavior design in architecture. VHDL supports synchronous and asynchronous circuit design. The time behavior of functions can be observed by simulation. It thus allows the behavior of the underlying system to be verified and modeled before the design is translated into actual gates and cables ("IEEE Standard for VHDL Language Reference Manual," 2019; *VHDL Tutorial: Learn by Example*, n.d.; Nageswaran, 1997). In addition, programs prepared in VHDL are portable structures. A component prepared for a previous project can be integrated into subsequent projects (Baker, n.d.; Pak, n.d.).

There are many advantages of using VHDL (*VHDL Mini-Reference*, n.d.).

- Has independent design definitions
- Applicable to many manufacturer ICs
- The design can be updated when necessary
- Allows a standard documentation
- It shortens the design process
- Accelerates the commercialization of design
- Reduces research and development costs
- Increases final product quality
- Enables detailed control of its functions
- Re-use of previous designs as components

A program prepared with VHDL consists of three basic parts;

1. Entity
2. Architecture
3. Procedure

The input and output pins of the entity to be designed are defined in the Entity section. An example for “OR GATE” of Entity is given below.

```
entity OR_GATE is                -- Comment line
 generic(
            data_width: integer :=4;
            delay_time: time := 10ns
        );
 port    (
            D : in std_logic_vector( data_width-1 downto 0);
            Q : out std_logic
        );
 signal ln01 : std_logic;
end OR_GATE;
```

where OR_GATE is the project name. The name of the vhd file with all the codes must be the same as the project name. Expressions defined in the generic field are used to define presets that can be set without changing the code structure. Here, the width of inputs the OR gate will have is defined as generic. Here, two expressions are defined inside the generic(…) expression. There should be no markup at the end of the last statement. Generic values can be updated as needed during synthesis and simulation. But the hardware is not hot-swappable (Gunes & Ors, n.d.; “IEEE Standard for VHDL Language Reference Manual,” 2019; Nageswaran, 1997).

In the Port section, there are pin definitions that the designed hardware will use when communicating with the outside world. Here, there is a lot of information such as port names, port direction, data width, data type. All items must be separated with a semicolon. There is no sign at the end of the last item (Baker, n.d.; *VHDL Tutorial: Learn by Example*, n.d.).

In this section, after the port definition, signal definition can be made if needed. Detailed information about this will be given in the definitions section.

The “- -” (double dash) symbol in the first line is used to add a comment line. This sign is like // in C language, % in Matlab, or # in Python. Expressions after the double dash are ignored by the compiler.

It is the unit in which the type and direction of the input and output pins of the Entity

design are defined in detail. However, it does not contain any information about the functions of the design. Input-output definitions and directions (in, out, inout, buffer) are made in this section (Baker, n.d.; “IEEE Standard for VHDL Language Reference Manual,” 2019).

The descriptions of pin directions used here are as follows (Pak, n.d.);

- In: It is a read-only structure used for input pins.
- Out: It is the structure used for the output pins and can only be given a value but cannot be read.
- Buffer: It is the only driver accepting structure that can be read and written for bidirectional pins.
- Inout: It is a pin structure that can have more than one driver and can both read and write values.

A procedure is a construct used within the designed entity to avoid duplicating an operation that is often repeated. It is not an indispensable structure. A procedure that increments the value in a variable by one is presented below.

```
procedure one_incrementer (variable vr: inout int8) is
begin
    if (vr <= MAKSIMUM) then
        vr := vr + 1;
    end if;
end
```

Architecture is the area where the structure of the designed asset is determined. The architecture required for the OR_GATE design defined in the Entity section is shown below.

```
architecture behaviour of OR_GATE is
begin

    Q <= D(0) or D(1) or D(2) or D(3);

end behaviour;
```

The term “behaviour” as used here is an arbitrary term for the programmer. However, special terms of the programming language cannot be used here.

The function of the designed system is defined in Architecture. This design can be done in three different ways (Gunes & Ors, n.d.; Pak, n.d.). They are:

- Behavioral
- Data Flow
- Structural

Let's examine these identification forms with a full adder design.

First, the entity required for the 1-bit full adder design must be defined. The entity required for this structure is given below.

```
entity FULL_ADDER is
port   (
        A : in std_logic;
        B : in std_logic;
        Carry_in : in std_logic;
        Q : out std_logic;
        Carry_out : out std_logic
        );
end FULL_ADDER;
```

Behavioral Style

Here, A, and B are the two numbers to add, while Carry_in is the carry input. The truth table for this full adder will be as in Table 1.

Table 1. Truth Table for Full Adder

A	B	Carry_in	Carry_out	Q
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

In line with this table, a code can be created for the behavioral style as follows.

```
architecture Behaviour of FULL_ADDER is
begin
    process (A, B, Carry_in)
    begin
        if (A='0' and B='0' and Carry_in='1') or
                (A='0' and B='1' and Carry_in='0') or
                (A='1' and B='0' and Carry_in='0') then
            Q <= '1';
            Carry_out <= '0';

        elsif (A='0' and B='0' and Carry_in='0') then
            Q <= '0';
            Carry_out <= '0';

        elsif (A='0' and B='1' and Carry_in='1') or
                (A='1' and B='0' and Carry_in='1') or
                (A='1' and B='1' and Carry_in='0') then
            Q <= '0';
            Carry_out <= '1';
        else
            Q <= '1';
            Carry_out <= '1';
        end if;
    end process;
end Behaviour;
```

The process structure and if, elsif, else structure used here will be mentioned in the following stages. Behavioral style is designed using processes. In this style, typing is performed in a program-like manner. However, it is not clear what kind of formulation the operations to be performed have. It's just like constructing a truth table with conditions (Gunes & Ors, n.d.; Pak, n.d.).

Data Flow Style

In the data flow style, the operations are presented more clearly. Results are obtained by arithmetic and/or logic operations with control signals and data. In order for the full adder to be designed in a data flow architecture, logic operations should be introduced by passing from the truth table to the Karnaugh diagram. Karnaugh diagrams and logic formula of the truth table given above are given in Table 2.

Table 2. Karnaugh Diagrams for Q and Carry out in Full Adder Design

Q

A \ B C	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Q = (A xor B) xor C

Carry_out

A \ B C	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Carry_out = (A and B) or (A and C) or (B and C)

The codes in the data flow style created according to the obtained formulas of the full adder circuit are presented below.

```
architecture BEH of FULL_ADDER is
    signal L : std_logic;
begin
  Carry_out<=(A and B) or (A and Carry_in) or (B and Carry_in);
  L <= A xor B;
  Q <= L xor Carry_in;
end BEH;
```

Structural Style

It is formed as a result of the organization of sub-modules working simultaneously in a structural style. For the full adder, this can be achieved by combining the half adders. For this reason, a half adder structure is needed first for this example. First, let's examine the half adder structure.

```
entity HALF_ADDER is
port    (
         A : in std_logic;
         B : in std_logic;
         Carry_out : out std_logic;
         Q : out std_logic
         );
end HALF_ADDER;

architecture DATAFLOW of HALF_ADDER is
begin
    Q <= A xor B;
    Carry_out <= A and B;
end;
```

As can be seen, A and B are defined as input, Carry_out and Q are defined as output. These values are simply obtained with an EXCLUSIVE-OR and an AND operation.

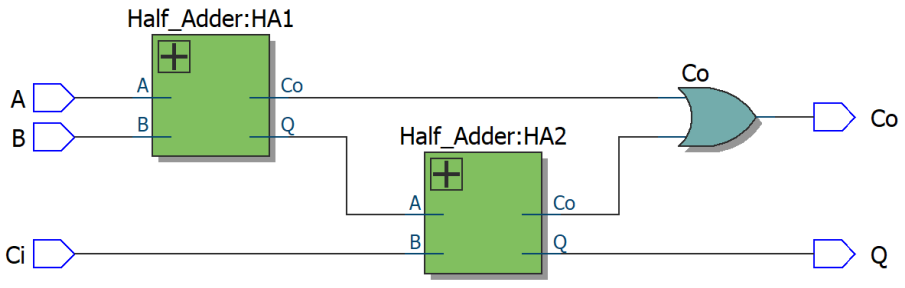

Figure 4. Full Adder Construction with Half Adders

Using half adders, it is possible to obtain a full adder with the scheme given in Figure 4. Defining this schema in architecture is what we call structural style. The architecture

created with the definition of this schema is shown below.

```
architecture STRUCTURE of FULL_ADDER is

    component HALF_ADDER
        port (
            In1, In2 : in std_logic;
            Out1, Out2 : out std_logic
        );
    end component;

    signal line1, line2, line3 : std_logic;
begin

    HA1 : HALF_ADDER port map (A, B , line1, line2);
    HA2 : HALF_ADDER port map (line2, Carry_in, line3, Q);
    Carry_out <= line1 or line3;

end STRUCTURE;
```

Components are one of the most important elements of VHDL. Thanks to its component definition feature, VHDL offers the opportunity to include previous studies into new studies. A defined component can be used repeatedly within the architecture. The line1, line2, line3 specified here is a signal object. It can also be understood from the code that these provide the electrical connection between two points (Gunes & Ors, n.d.; Pak, n.d.).

Data Objects

The data objects used in VHDL are:

- Signal
- Constant
- Variable

Signals, the most important data object, provide connections within the design. Constant and Variable are less frequently used data objects.

There are a number of standard details to consider when specifying names for data objects.

- When creating names for data objects, characters, numbers, and the underscore are permitted for use.
- Characters not used in English cannot be used.

- Keywords used in VHDL cannot be specified as data object name.
- The first character of the data object must be a letter.
- The last character of the data object cannot be an underscore.

Assigning a Value to a Data Object

While assigning a single-bit value to the signal data object, it is written as 1 or 0 in single quotes. But for assigning a multiple bit value, it is written as a sequence of 1 and 0 in double quotes. For example, if beep is a 1 bit signal and dt is a 4 bit signal:

```
beep <= '0';
dt <= "1011";
```

The signal object is defined as shown below.

```
SIGNAL data_obj_name : type [:= default_value];
```

```
signal line1, line2, line3 : std_logic;
Signal Beep : Bit := '0';
Signal data : integer;
```

In this definition, it is not obligatory to include the expression given in square brackets. Sample signal definitions are given above.

Constants are expressions that are initially assigned a value and cannot be changed afterwards. The definition and examples are given below.

```
CONSTANT data_obj_name : type := value;
```

```
Constant Yes : Boolean := true;
Constant No : Boolean := false;
```

Variable stores temporary information. They are expressed in the process and subprogram, and the information they store is accessible within the respective process. The definition of a variable is given below. It is not mandatory to use the part in square brackets.

```
VARIABLE data_obj_name : type [:= default_value];
```

```
VARIABLE X,Y : std_logic_vector(7 downto 0);
variable a : integer := 0;
```

Data Types

Data objects are always defined using a data type. The data types that can be synthesized are as follows.

- BIT
- BIT_VECTOR
- STD_LOGIC
- STD_LOGIC_VECTOR
- SIGNED
- UNSIGNED
- INTEGER
- ENUMERATION
- BOOLEAN

BIT and BIT_VECTOR

BIT defines a single bit object. This object can take a value of '0' or '1'. BIT_VECTOR is used to define the BIT type object with the specified width.

```
SIGNAL x1 : BIT;
SIGNAL C : BIT_VECTOR (1 to 4);
SIGNAL VR1 : BIT_VECTOR (7 downto 0);
SIGNAL VR2 : BIT_VECTOR (0 to 7);
```

As defined above, x1 is a single-bit object while C is a 4-bit object. VR1 and VR2 are both 8-bit objects. The difference between these two is in the order of the bits. In VR1, the most significant bit is in the 7 indexed bit and the least significant bit is in the 0 indexed bit. In VR2, on the other hand, the most significant bit is in the 0 indexed bit, while the least significant bit is in the 7 indexed bit. This can be seen in the data assignment and results below.

```
VR1 <= "10110110";
    -- VR1(0) -> 0
    -- VR1(1) -> 1
    -- VR1(2) -> 1
    -- VR1(3) -> 0
    -- VR1(4) -> 1
    -- VR1(5) -> 1
    -- VR1(6) -> 0
    -- VR1(7) -> 1
```

```
VR2 <= "10110110";
    -- VR2(0) -> 1
    -- VR2(1) -> 0
    -- VR2(2) -> 1
    -- VR2(3) -> 1
    -- VR2(4) -> 0
    -- VR2(5) -> 1
    -- VR2(6) -> 1
    -- VR2(7) -> 0
```

STD_LOGIC and STD_LOGIC_VECTOR

STD_LOGIC type, which has a more flexible structure than BIT type, can take Z, 0, 1, L, H, U, -, X and W values. The explanations of these values are shown in Table 3 ("IEEE Standard for VHDL Language Reference Manual," 2019; *VHDL Tutorial: Learn by Example*, n.d.; Nageswaran, 1997). Objects with this data type can perform AND, NAND, OR, NOR, XOR, XNOR, NOT logic operations. As with the BIT structure, adding _VECTOR defines the width of the object. To use it, it is necessary to install the std_logic_1164 package in the ieee library.

Table 3. Possible Values of STD_LOGIC type

Value	Explanation
0	Logic 0
1	Logic 1
Z	High Impedance
W	Weak signal (unable to say 0 or 1)
L	Weak 0
H	Weak 1
-	Don't Care
U	Uninitialized
X	Unknown

```
VARIABLE x : std_logic_vector (7 downto 0);

VARIABLE btn: std_logic;
```

Here, the variable x is 8 bits wide, while the btn variable is 1 bit wide.

Signed - Unsigned

It is a data type used for signed and unsigned data. In addition to the ability to access the value of each bit separately, such as std_logic_vector, it also supports a number of operations such as arithmetic (+, - *), comparison and shifting. To use it, the std_logic_arith package from the ieee library must be installed.

Objects of UNSIGNED data type always store positive values. The value is defined by all bits. For example, an 8-bit data object stores values between 0 and 255.

Objects of SIGNED data type can have positive or negative values. The sign of the data object is determined by the most significant bit. If the most significant bit is 0, it is positive. Contrarily, if the most significant bit is 1, it is negative. The absolute value (magnitude) of a negative value is found by inverting the remaining bits and adding 1.

```
SIGNAL in1: unsigned (3 downto 0);
SIGNAL in2: signed (7 downto 0);
```

Integer

Generally, it can take values in the range of -2^{31} to 2^{31} -1. Depending on the user, this range of values can be changed. If a value outside the definition range is given, an error will occur. It can store negative values. In addition to standard operations such as addition, subtraction, multiplication, division, and modulus, they can also be used in comparison operations such as greater than, greater than or equal, less than, less than or equal, equal, and unequal.

```
VARIABLE a : integer;
VARIABLE b : integer range -100 to 200;
```

The variable a, defined here is defined to be used in the natural range of the integer. B, on the other hand, is set to store values in the range of minimum -100 and maximum 200.

Boolean

This data type has two values as TRUE and FALSE.

```
Constant Yes : Boolean := true;
Constant No : Boolean := false;
```

Enumeration

It is a data type whose values can be defined by the programmer. In order to use this data type, the data type must be defined first. All values to be used in this type must be given during type definition. For example, let's define a color variable.

```
TYPE color is (red, orange, yellow, blue, black, white);
SIGNAL pxl : color;
```

Here, the pxl variable cannot take a value other than the defined colors. While the pxl <= blue; operation is a valid assignment, the pxl <= green; operation is an invalid assignment. Because the color green is not included in the type definition.

Operators

The operators used in VHDL are grouped and presented in Table 4.

Table 4. Operators Used in VHDL

	Symbol of Operator	
Miscellaneous	MOD	Modulus
	ABS	Absolute value
	**	Power
Multiplication, Division	*	Multiplication
	/	Division
Addition, Subtraction	+	Addition
	-	Subtraction
Sign	+	Positive
	-	Negative
Logic	NOT	Not
	AND	And
	NAND	Not and
	OR	Or
	NOR	Not or
	XOR	Exclusive or
	XNOR	Not exclusive or

Sequential Operations

The simultaneous assignment statements in the architectural design have no priority or order. Changing their order does not affect the function of the final structure. However, VHDL also provides another type of statement, called sequential statements. These are statements such as if statement, case statement, loop statements. The order of these will affect its function. So the order is important. Separation of concurrent statements that do not change the result of their sequencing and sequential statements whose ordering is important is provided by the PROCESS structure (Gunes & Ors, n.d.; "IEEE Standard for VHDL Language Reference Manual," 2019; *VHDL Mini-Reference*, n.d.).

The PROCESS statement is included in the architecture. The variable type is defined in PROCESS as shown. Data in a variable defined in a PROCESS can only be exported by transferring it to another signal type object.

```
[Name :] PROCESS (Sensitivity List)
 VARIABLE var_name : var_type [range . to .][:=default_value];
begin
                -- codes
    ....        -- codes
end process [Name];
```

The expressions in square brackets specified here are not mandatory. The expression indicated by Name is used to define PROCESS. It is not a mandatory statement, but naming is necessary to create a regular code. The sensitivity list is important. Here, the signals to be used in PROCESS are listed. Thus, it is defined that this PROCESS is affected by the change of signals in the specified list. If there is a variable to be used in the sequential code, its definition should be added to the presented location as VARIABLE. The scope of the defined variable is limited in PROCESS.

The most frequently used expression in the process is the IF structure. The IF structure is a structure used when the activation or deactivation of some codes depends on certain conditions. Here is how the IF structure is in VHDL.

```
If comparison then        -- mandatory line
    VHDL statements;
Elsif comparison then     -- not mandatory, in case of need
    VHDL statements;
Else                      -- not mandatory, in case of need
    VHDL statements;
End if;                   -- mandatory line
```

An example code is shown below. With this code, x1 or x2 is transferred to the f signal according to the state of the Sel signal. In addition, the structure of the process and the sensitivity list can be seen. As can be understood, a multiplexer structure is presented here.

```
PROCESS (Sel, x1, x2)
BEGIN
    IF Sel = '0' THEN
        f <= x1;
    ELSE
        f <= x2;
    END IF;
END PROCESS;
```

From the expression given here, it is not clear how many bits wide or type x1, x2, and f are. However, all 3 signals must be of the same type and width.

The case statement can be viewed as an alternative to the nested if (elsif) statement. The fact that VHDL is a fully defined language shows itself here. Here, all states for

the signal (or variable) that is the input of the case structure should be given in options. Otherwise, synthesis will not be possible. In order to provide this situation easily, there is an option called **others** in VHDL. This is a term used to denote all alternatives other than those defined. The Case structure used in VHDL is given below.

```
case statement is
    when value_1 =>
        VHDL statement;
    when value_2 =>
        VHDL statement;
    when value_3 =>
        VHDL statement;
    when others =>
        VHDL statement;
end case;
```

Here, if the expression given between the “case” and “is” is a value other than value_1, value_2 and value_3, which is compared in case options, the “when others” tab will be active. The “when others” statement is not needed if all possible values are defined. Below is a multiplexer structure with 2 bit select inputs.

```
case Sel is
    when "00" =>
        f <= x0;
    when "01" =>
        f <= x1;
    when "10" =>
        f <= x2;
    when "11" =>
        f <= x3;
end case;
```

The “When others” are omitted as all possible states are mentioned here.

When we talk about loops in VHDL, we encounter for and while loops. A for loop is a loop used to execute a specific group of commands a specified times. The structure of the “for loop” is given below.

```
[Name :] for arbitrary name in start to stop loop
    VHDL statements;
end loop [Name];
```

For example, the design of a structure that finds the number of bits that are 1 in an A signal coming from outside is given below.

```
process(A)  -- Defination of Process
variable V1:integer range 0 to 7;
begin
    V1:=0;
    for i in 0 to A'length-1 loop
        if ( A(i) = '1' ) then
            V1 := V1 + 1;
        end if;
    end loop;
    X <= V1;
end process;
```

Another loop structure in VHDL is the while loop. The general structure of the while loop is given below.

```
[Name :] while Comparison Statement loop

    VHDL ifadeleri;

end loop [Name];
```

The design of a structure realized with a while loop that finds the number of bits that are 1 in an incoming A signal is given below.

```
process(A)
 variable V1 : integer range 0 to 7;
 variable i : integer := 0;
begin
    V1 := 0;
    while i < A'length loop
        if ( A(i) = '1' ) then
            V1 := V1 + 1;
        end if;
        i := i + 1;
    end loop;
    X <= V1;
end process;
```

The necessity of performing synchronized operations with the clock pulse in VHDL is a very common situation. In these cases, the detection of the pulse moment of the clock signal (as a rising edge or falling edge) becomes important. This process can be defined in VHDL with PROCESS and is controlled by the ‘EVENT statement. Below are two different examples that increase the value of the counter signal by one on the rising edge and by the other one the falling edge. The clk used here is a signal received from the global clock line, and the counter is an integer type signal that can be mathematically processed.

The counter design triggered on the rising edge is as follows.

```
PROCESS (clk)
Begin
    If (clk'EVENT and clk='1') then
        counter <= counter + 1;
    End if;
End process;
```

The counter design triggered on the falling edge is as follows.

```
PROCESS (clk)
Begin
    If (clk'EVENT and clk='0') then
        counter <= counter + 1;
    End if;
End process;
```

The only difference between the two processes is that the clk in the second comparison in the if statement is equal to 1 in one and 0 in the other. Because the 'EVENT statement is triggered on both rising edge and falling edge. However, the value of the triggered signal determines which edge it is. If there has been a trigger and the signal value is 1, it means a rising edge change. If there has been a trigger and the signal value is 0, it means a falling edge change.

An Example of VHDL Application

Let's make an example application using what we have mentioned so far. For example, let's design one 32kHz and one 1MHz signal generator using the 50 MHz clock signal. Let's do this on a physical card. The card we will use in this example is the Altera DE2-115 FPGA board. This board has slide switches, buttons, leds, 7 segment displays, 2-line LCD screen, SDRAM, SRAM, FLASH memory, 50MHz oscillator, DAC and VGA output ports and many ports and hardware. There is a Cyclone IV FPGA IC on this board. This is a development board and the datasheet details which pin of the FPGA IC is connected to which hardware. The picture of the development board is shown in Figure 5.

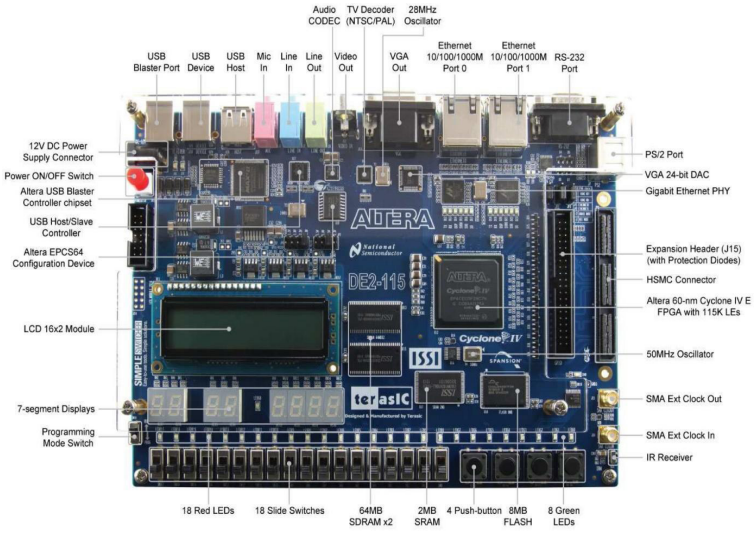

Figure 5. Altera Cyclone IV EP4CE115F29C7 DE2-115 Development Board

This board contains the JTAG programming structure on its own. It is programmed by connecting to a PC via USB with Quartus software web edition, which Altera company distributes free of charge. It is possible to write code in both VHDL and Verilog hardware description languages with Quartus software. Here, after installing the Quartus software, it will be explained step by step how to create a project, how to simulate it and how to download to the FPGA IC. The Quartus software, whose images are given here, is v13.1 version.

The screen shown in Figure 6 appears first. On this screen, there is the Project Navigator section in the upper left corner. Here is the information of the project being worked on. Under the Project Navigator section, items that we may want to see related to the project such as Hierarchy, files, design units etc. are listed in tabs. On this screen, creating a new project starts by clicking New Project Wizard or clicking New Project Wizard from the File menu.

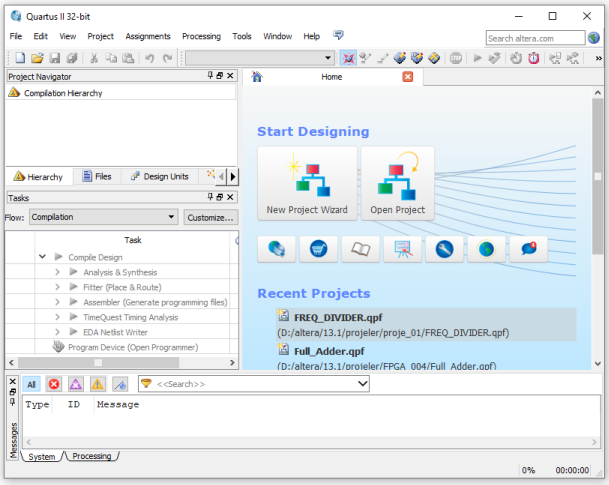

Figure 6. Quartus Software Welcome Interface

After clicking on the New Project Wizard, the Introduction page first appears. We proceed on this page with the Next button. On the next page, Directory, Name, Top-Level Entity information should be entered. This page view is given in Figure 7. Here, in the first line, the folder where our project will be saved should be selected. The second line contains the project name. The name we will give here is automatically transferred to the third line. Let's give FREQ_DIVIDER as the name of this project. Let's press the Next button to move to the next page.

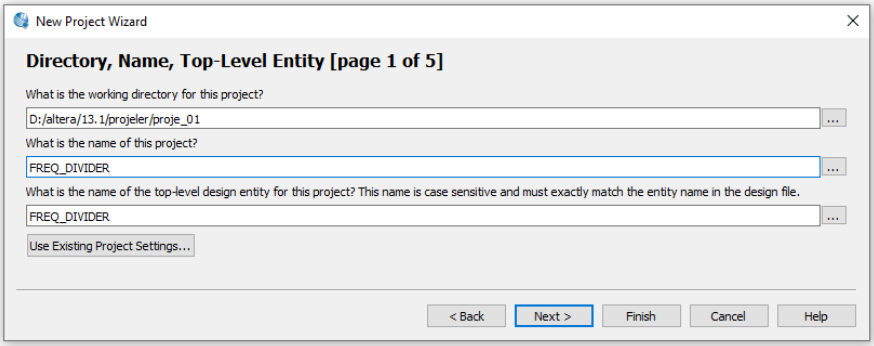

Figure 7. New Project Creation Steps; Folder and Project Name Entry

On the next page, if you have VHD files that you have prepared before and need to use in the new project, a page will open for you to add them. Since there is no such need for this project, we can proceed with the Next button without taking any action.

Then, on the page shown in Figure 8, it will ask you to specify the FPGA IC you will use. There is Cyclone IV EP4CE115F29C7 IC on the development board we will use in this project. This device must be selected from the list.

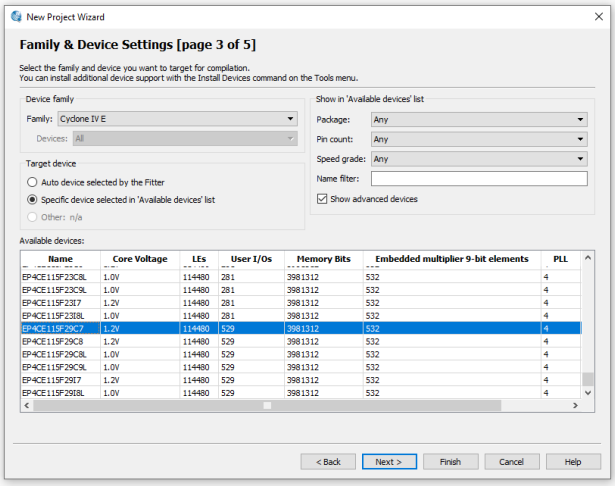

Figure 8. Determining the Family and Name of the FPGA IC

Since we will not make any changes on the next pages, we can complete the process by

clicking Finish after this point. On the page that opens, the name of the FPGA IC used in the project and the name of the project can now be seen in Project Navigator.

For the operations to be done in this project, we need to add a new VHDL file. For this, clicking New from the File menu will open a list containing many file types. Select VHDL File from this list and click OK.

In the editor page that opens, we can start writing the codes of the frequency divider. Here, we need to write the *library ieee* command to indicate that we will use the ieee library first. Later, *the std_logic_1164* and *std_logic_arith* packages that we will use in this project are added.

After this point, the Entity should be created. In this project, 1 input pin for 50MHz input, 1 input pin for reset and 2 output pins for 32kHz and 1MHz outputs are required.

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;

Entity FREQ_DIVIDER is
    Port(
            clk     :in std_logic;
            rst     :in std_logic;
            Q_32kHz :out std_logic;
            Q_1MHz  :out std_logic
    );
    signal cntr_32k : unsigned (9 downto 0) := (others=>'0');
    signal cntr_1M  : unsigned (4 downto 0) := (others=>'0');
    signal sclk_32k :std_logic :='0';
    signal sclk_1M  :std_logic :='0';
end FREQ_DIVIDER;
```

After this point, the architecture should be designed. The basic logic here would be: There is 20ns between two rising pulses of the 50MHz signal. Since 1MHz is 0 for 500ns and 1 for 500ns, 1MHz signal output should change its level for every 25 pulses of 50MHz signal. The 0 and 1 periods of the 32kHz signal are 15.63µs. Approximately 781 pulses (781x20ns =15.62µs and f=32.01kHz) are needed to obtain this period. A 5-bit unsigned counter is needed to count 25 pulses, and a 10-bit unsigned counter is needed to count 781 pulses.

The logic in this architecture is to create 2 different counters. When these counters reach the specified numbers, they should toggle their output and reset their own counter. The created entity and architecture are as follows.

```
Architecture Beh of FREQ_DIVIDER is
Begin
    process (clk,rst)
    Begin
        if (rst='1') then
            cntr_32k <= (others=>'0');
            cntr_1M <= (others=>'0');
            sclk_1M <='0';
            sclk_32k <='0';
        elsif ( clk'Event AND clk='1' ) then
            cntr_32k <= cntr_32k + 1;
            cntr_1M <= cntr_1M + 1;
            if (cntr_1M >= 25) then
                sclk_1M <= not sclk_1M;
                cntr_1M <= (others=>'0');
            end if;
            if (cntr_32k >= 781) then
                sclk_32k <= not sclk_32k;
                cntr_32k <= (others=>'0');
            end if;
        end if;
    end process;
    Q_32kHz<=sclk_32k;
    Q_1MHz<=sclk_1M;
end Beh;
```

In this architecture, the frequencies created such as sclk_32k and sclk_1M are first transferred to an object of signal type and from there to the output. This is because the final value is inverted after a certain number of clock pulses. In order to perform an inversion operation, it is necessary to read the value of the signal first. However, since it cannot be read from the pins defined as out, firstly, the data is transferred to the signal object and then to the output.

The created file can be saved with the CTRL+S key combination. The project name automatically appears as the file name during recording. Here it is necessary to save without modification.

After saving, the project should be compiled. For this, click Start Compilation from the Processing menu. Compilation stages can be followed in the Task menu under Project Navigator on the left. The entire build process should be completed in green. Red errors can be observed in the warning messages pane at the bottom. The errors specified in this menu should be corrected and recompiled.

After a successful compilation, the simulation can be started. To start the simulation, select Run Simulation Tool from the Tools menu and RTL Simulation from the drop-down menu when hovering over it. Then Quartus opens ModelSim software and adds a library called work. On the page that opens, the library is located on the far left. If the

mentioned menus are not visible, it is possible to open them from the Window menu of ModelSim.

When the Work in the library is expanded, the project name freq_divider appears under it. Double-clicking on the Freq_divider displays all signals defined in the project in the “Object list”. These signals (clk, rst, Q_32kz and Q_1MHz) are dragged and dropped into Wave windows, from which we will intervene and observe.

Right click clk in the Wave window and select Clock. Since the period is in picoseconds in the window that opens, enter 20000 and press the OK button.

Right click rst in the Wave window and click Force. The value of U is set to 0 in the window that opens.

After this point, we need to determine how long the simulation should be run. Since the longest period belonging to the 32kHz signal is 31.25µs, simulation duration can be 50µs . For this, 50us is written in the upper middle area of ModelSim that is 100ps on the start up. Instead of the symbol for micro µ, the u letter is used. Simulation is performed by clicking “Run 100 F9” in the Run tab from the Simulate menu. Right click on the simulation screen and select Zoom Full. The simulation result will appear as shown in Figure 9.

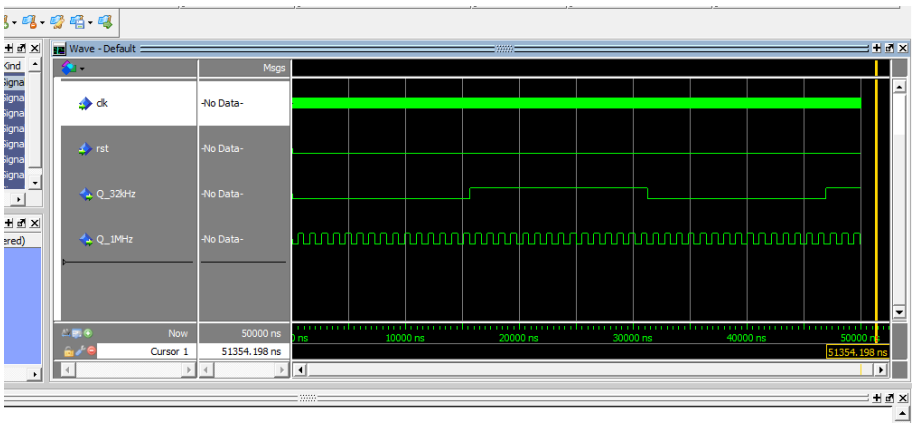

Figure 9. Wave Window Simulation Results of ModelSim Software

For the application, the pins of the FPGA IC must be associated with the pins defined in the entity. For this, click on Pin Planner in the Assignments menu. The Pin Planner interface shown in Figure 10 will open. There is an All Pins section at the bottom of this interface. Here are the pins defined in Entity. Here, the relevant pin numbers should be entered according to the datasheet on the Location tab. For example, Y2 pin is given for 50MHz. For reset operation, a push button connected to R24 can be selected. The outputs can be transferred to the outside world from the header ends or given to the LEDs for monitoring. Here, E21 and E22, which are the connections of the green LEDs, are used. Although their flashing is not obvious due to the high frequency, these LEDs on the board should be able to be observed at low brightness like a PWM with 50% duty.

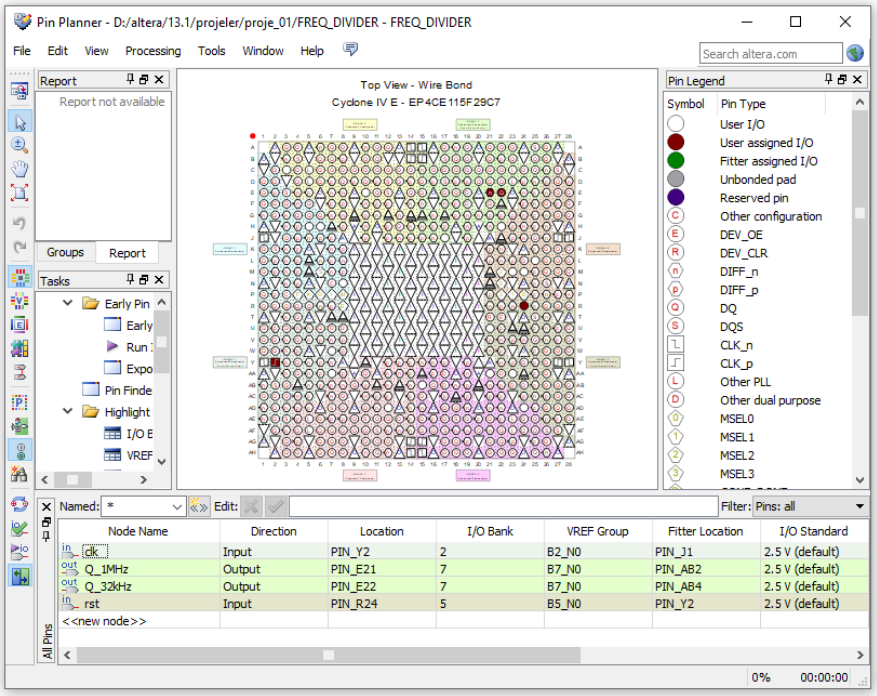

Figure 10. Pin Planner Interface

After this process, Pin Planner is closed. The project is compiled again. As a result of error-free compilation, it can be passed to the programming phase. The DE2-115 development board is connected to the computer and its drivers are installed. For drivers, the Altera folder containing the installed version of the Quartus program should be searched. Then click Programmer in the Tools menu and the interface in Figure 11 will open.

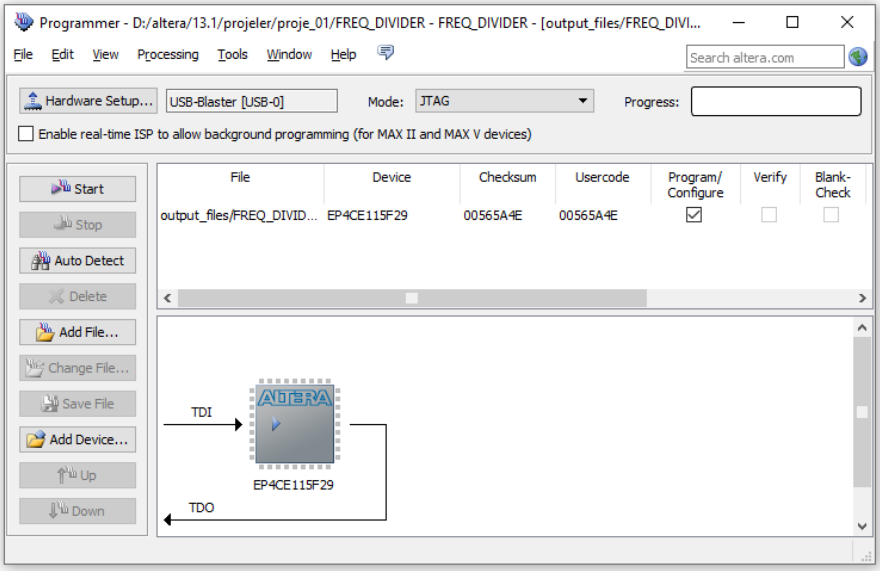

Figure 11. The Programmer Interface

The freq_divider of the prepared project should be seen on the opened page. If this

file does not exist, the add file button is pressed. In the opened interface, the FREQ_DIVIDER.sof file is selected in the output_files folder in the folder where the project is located. When the start button is pressed, the program will be loaded into the FPGA.

FPGAs are integrated circuits with very powerful processing capacity. However, creating an embedded system with an FPGA is a very laborious and detailed work. According to the needed system, How much speed the needed how much processing load it has, how much hardware it needs to be solved with logic elements, the EEPROM capacity required for the program, etc. should be carefully determined, the FPGA to be used in line with these needs should be determined and the necessary PCB design should be made. All these issues should be carefully examined for determining the system structure to consist of an FPGA or microcontroller.

References

Baker, G. (n.d.). *VHDL references*. https://www2.cs.sfu.ca/~ggbaker/reference/

Chu, P. P. (2008). *FPGA prototyping by VHDL examples*. A John Wiley & Sons Inc. Publication.

FPGA designs with VHDL documentation. (n.d.). Retrieved July 11, 2021, from https://vhdlguide.readthedocs.io/en/latest/index.html

Gunes, E. O., & Ors, S. B. (n.d.). *VHDL notes*. Retrieved June 5, 2021, from https://web.itu.edu.tr/~ayhant/dersler/sstu/vhdl/VHDL_sunumI.pdf

IEEE standard for VHDL language reference manual. (2019). In *IEEE Std 1076-2019* (pp. 1–673). https://doi.org/10.1109/IEEESTD.2019.8938196

IEEE Standards Board., & IEEE Computer Society. Test Technology Technical Committee. (1993). *1149.1-1990 - IEEE standard test access port and boundary-scan architecture*.

Leong, P. H. W., Tsang, P. K., & Lee, T. K. (1998). A FPGA based forth microprocessor. *IEEE Symposium on FPGAs for Custom Computing Machines*, *1998-April*, 1–2. https://doi.org/10.1109/FPGA.1998.707903

Nageswaran, J. M. (1997). *VHDL reference manual*. Synario Design Automation. www.synario.com

Pak, B. (n.d.). *VHDL lecture notes*.

Rajewski, J. (2017). *Learning FPGAs : digital design for beginners with Mojo and Lucid HDL*. O'Reilly Media, Inc.

Trimberger, S. M. (2015). Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, *103*(3), 318–331. https://doi.

org/10.1109/JPROC.2015.2392104

Unlersen, M. F., Yaldiz, E., & Imeci, S. T. (2018). FPGA based fast bartlett DoA estimator for ULA antenna using parallel computing. *Applied Computational Electromagnetics Society Journal*, *33*(4).

Unlersen, M. F. (2015). FPGA Kullanılarak Dizi Anten Performansının İyileştirilmesi - Improving of array antenna performance using FPGA. In *Institute of Science and Technology - Electrical and Electronics Engineering Department*. Selcuk University.

VHDL mini-reference. (n.d.). The University of California, Irvine. Retrieved June 5, 2021, from https://www.ics.uci.edu/~jmoorkan/vhdlref/vhdl.html

VHDL tutorial: Learn by example. (n.d.). Retrieved July 11, 2021, from http://esd.cs.ucr.edu/labs/tutorial/

What is an FPGA? Field Programmable Gate Array. (n.d.). Retrieved June 11, 2021, from https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

What is an FPGA? Programming and FPGA basics - INTEL. (n.d.). Retrieved June 11, 2021, from https://www.intel.com/content/www/us/en/products/details/fpga/resources/

About the Authors

Muhammed Fahri UNLERSEN received B.Sc., MSEE and Ph.D. degrees in Electrical and Electronics Engineering from Selcuk University, Konya, Turkey in 2004, 2007 and 2015, respectively. He was a Lecturer in Doganhisar Vocational School at Selcuk University from 2005 to 2016. He has been working as an Assistant Professor in the Department of Electrical and Electronics Engineering at Necmettin Erbakan University since 2016. His current research areas are the array antennas, electromagnetic field theory, field programmable gate arrays and optimization techniques. He also has interests in control theory, inverse kinematics of 6 DoF Stewart Platform and experimental motion platforms capable of generating high g for simulators.

Email: mfunlersen@erbakan.edu.tr, Orcid: https://orcid.org/0000-0001-7850-6712

Similarity Index

The similarity index obtained from the plagiarism software for this book chapter is 13%.

To Cite This Chapter:

Unlersen, M. F. (2021). Field Programmable Gate Array. In S. Kocer, O. Dundar & R. Butuner (Eds .), *Programmable Smart Microcontroller Cards* (pp. 179–207). ISRES Publishing.