**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **02-June-2020** | **Name:** | **Raziya Banu** |
| **Course:** | **HDL** | **USN:** | **4AL16EC058** |
| **Topic:** | **FPGA Basics: Architecture, Applications and Uses** | **Semester & Section:** | **8th sem & ‘B’ section** |
| **Github Repository:** |  |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report –**  In my first session today I have studied about the FPGA Basics: Architecture, Applications and Uses.  The field-programmable gate array (FPGA) is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. ****What is FPGA?****   The [**field-programmable gate array (FPGA)**](https://www.arrow.com/en/categories/programmable-devices/programmable-logic-devices/fpgas) is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.    The FPGA has its roots in earlier devices such as programmable read-only memories (PROMs) and programmable logic devices (PLDs). These devices could be programmed either at the factory or in the field, but they used fuse technology (hence, the expression “burning a PROM”) and could not be changed once programmed. In contrast, FPGA stores its configuration information in a re-programmable medium such as static RAM (SRAM) or flash memory. FPGA manufacturers include [**Intel**](https://www.arrow.com/en/manufacturers/intel), Xilinx, [**Lattice Semiconductor**](https://www.arrow.com/en/manufacturers/lattice-semiconductor), [**Microchip Technology**](https://www.arrow.com/en/manufacturers/microchip-technology) and **[Microsemi](https://www.arrow.com/en/manufacturers/microsemi)**. ****FPGA Architecture**** A basic FPGA architecture (Figure 1) consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.  Depending on the manufacturer, the CLB may also be referred to as a logic block (LB), a logic element (LE) or a logic cell (LC).  An individual CLB (Figure 2) is made up of several logic blocks. A lookup table (LUT) is a characteristic feature of an FPGA. An LUT stores a predefined list of logic outputs for any combination of inputs: LUTs with four to six input bits are widely used. Standard logic functions such as multiplexers (mux), full adders (FAs) and flip-flops are also common.  The number and arrangement of components in the CLB varies by device; the simplified example in Figure 2 contains two three-input LUTs (1), an FA (3) and a D-type flip-flop (5), plus a standard mux (2) and two muxes, (4) and (6), that are configured during FPGA programming.  This simplified CLB has two modes of operation. In normal mode, the LUTs are combined with Mux 2 to form a four-input LUT; in arithmetic mode, the LUT outputs are fed as inputs to the FA together with a carry input from another CLB. Mux 4 selects between the FA output or the LUT output. Mux 6 determines whether the operation is asynchronous or synchronized to the FPGA clock via the D flip-flop.  Current-generation FPGAs include more complex CLBs capable of multiple operations with a single block; CLBs can combine for more complex operations such as multipliers, registers, counters and even digital signal processing (DSP) functions. ****CPLD vs FPGA**** Originally, FPGAs included the blocks in Figure 1 and little else, but now designers can choose from products with a large range of features. Less complex devices such as simple programmable logic devices (SPLDs) and complex programmable logic devices (CPLDs) bridge the gap between discrete logic devices and entry-level FPGAs.  Entry-level FPGAs emphasize low power consumption, low logic density and low complexity per chip. Higher-function devices add functional blocks dedicated to specific functions: Examples include clock management components, phase-locked loops (PLLs), high-speed serializers and deserializers, Ethernet MACs, PCI express controllers and high-speed transceivers. These blocks can either be implemented with CLBs—termed soft IP—or designed as separate circuits; i.e., hard IP. Hard IP blocks gain performance at the expense of reconfigurability.  At the high end, the FPGA product family includes complex system-on-chip (SoC) parts that integrate the FPGA architecture, hard IP and a microprocessor CPU core into a single component. Compared to separate devices, a SoC FPGA provides higher integration, lower power, smaller board size and higher-bandwidth communication between the core and other blocks. ****SoC FPGAs**** SoC FPGAs include a wide range of processing capabilities to suit different applications. A low-cost, low-power SoC FPGA such as Intel’s [**Cyclone V**](https://www.arrow.com/en/research-and-events/articles/cyclone-v-soc-fpga-development-kits-enable-software-design), for example, targets high-volume applications such as industrial motor control drives, protocol bridging, video processing cards and handheld devices. The device (Figure 3) has two distinct parts: the FPGA portion and a hard processor system (HPS) based around a single- or dual-core 32-bit Arm Cortex-A9 MPCORE running at 925 MHz. Each part contains its own set of peripherals, which includes hard IP from third-party vendors.  At the other end of the scale, the Stratix 10 SX targets high-performance applications in communications, data center acceleration, high-performance computing (HPC), radar processing and ASIC prototyping; that FPGA includes a quad-core 64-bit Arm Cortex-A53 running at up to 1.5 GHz. FPGA Design How do we transform this collection of thousands of hardware blocks into the correct configuration to execute the application? An FPGA-based design begins by defining the required computing tasks in the development tool, then compiling them into a configuration file that contains information on how to hook up the CLBs and other modules. The process is similar to a software development cycle except that the goal is to architect the hardware itself rather than a set of instructions to run on a predefined hardware platform.  Designers have traditionally used a hardware description language (HDL) such as VHDL (Figure 4) or Verilog to design the FPGA configuration.  Once the FPGA design has been created and verified using HDL, the compiler takes the text-based file and generates a configuration file that contains information on how the components should be wired together. Even if the HDL code has no errors, choosing the wrong FPGA may still cause the compilation to fail—for example, the FPGA runs out of a specific resource type or the compiler cannot create the required routes between components. ****FPGA Uses: An Attractive Choice for Certain Applications**** The ability to configure the hardware of the FPGA, reconfigure it when needed and optimize it for a particular set of functions makes the FPGA an attractive option in many applications.  FPGAs are often used to provide a custom solution in situations in which developing an ASIC would be too expensive or time-consuming. An FPGA application can be configured in hours or days instead of months. Of course, the flexibility of the FPGA comes at a price: An FPGA is likely to be slower, require more PCB area and consume more power than an equivalent ASIC.  **Implement a 4:1 MUX and write the test bench code to verify the module**  **module** mux\_4to1\_assign ( **input** [3:0] a, // 4-bit input called a  **input** [3:0] b, // 4-bit input called b  **input** [3:0] c, // 4-bit input called c  **input** [3:0] d, // 4-bit input called d  **input** [1:0] sel, // input sel used to select between a,b,c,d  **output** [3:0] out); // 4-bit output based on input sel    // When sel[1] is 0, (sel[0]? b:a) is selected and when sel[1] is 1, (sel[0] ? d:c) is taken  // When sel[0] is 0, a is sent to output, else b and when sel[0] is 0, c is sent to output, else d  **assign** out = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);    **endmodule** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **02-June-2020** | **Name:** | **Raziya Banu** | |
| **Course:** | **Udemy** | **USN:** | **4AL16EC058** | |
| **Topic:** | **Functions in Python** | **Semester & Section:** | **8th sem & ‘B’ section** | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Python Functions:** A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.  As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions. Defining a Function You can define functions to provide the required functionality. Here are simple rules to define a function in Python.   * Function blocks begin with the keyword def followed by the function name and parentheses ( ( ) ). * Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses. * The first statement of a function can be an optional statement - the documentation string of the function or docstring. * The code block within every function starts with a colon (:) and is indented. * The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.  Syntax def functionname( parameters ):  "function\_docstring"  function\_suite  return [expression]  By default, parameters have a positional behavior and you need to inform them in the same order that they were defined. Example The following function takes a string as input parameter and prints it on standard screen.  def printme( str ):  "This prints a passed string into this function"  print str  return Calling a Function Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.  Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −  #!/usr/bin/python  # Function definition is here  def printme( str ):  "This prints a passed string into this function"  print str  return;  # Now you can call printme function  printme("I'm first call to user defined function!")  printme("Again second call to the same function")  When the above code is executed, it produces the following result −  I'm first call to user defined function!  Again second call to the same function Pass by reference vs value All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −  #!/usr/bin/python  # Function definition is here  def changeme( mylist ):  "This changes a passed list into this function"  mylist.append([1,2,3,4]);  print "Values inside the function: ", mylist  return  # Now you can call changeme function  mylist = [10,20,30];  changeme( mylist );  print "Values outside the function: ", mylist  Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −  Values inside the function: [10, 20, 30, [1, 2, 3, 4]]  Values outside the function: [10, 20, 30, [1, 2, 3, 4]]  There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.  #!/usr/bin/python  # Function definition is here  def changeme( mylist ):  "This changes a passed list into this function"  mylist = [1,2,3,4]; # This would assig new reference in mylist  print "Values inside the function: ", mylist  return  # Now you can call changeme function  mylist = [10,20,30];  changeme( mylist );  print "Values outside the function: ", mylist  The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result −  Values inside the function: [1, 2, 3, 4]  Values outside the function: [10, 20, 30] Function Arguments You can call a function by using the following types of formal arguments −   * Required arguments * Keyword arguments * Default arguments * Variable-length arguments  Required arguments Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.  To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows −  #!/usr/bin/python  # Function definition is here  def printme( str ):  "This prints a passed string into this function"  print str  return;  # Now you can call printme function  printme()  When the above code is executed, it produces the following result −  Traceback (most recent call last):  File "test.py", line 11, in <module>  printme();  TypeError: printme() takes exactly 1 argument (0 given) Keyword arguments Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.  This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways −  #!/usr/bin/python  # Function definition is here  def printme( str ):  "This prints a passed string into this function"  print str  return;  # Now you can call printme function  printme( str = "My string")  When the above code is executed, it produces the following result −  My string  The following example gives more clear picture. Note that the order of parameters does not matter.  #!/usr/bin/python  # Function definition is here  def printinfo( name, age ):  "This prints a passed info into this function"  print "Name: ", name  print "Age ", age  return;  # Now you can call printinfo function  printinfo( age=50, name="miki" )  When the above code is executed, it produces the following result −  Name: miki  Age 50 Default arguments A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −  #!/usr/bin/python  # Function definition is here  def printinfo( name, age = 35 ):  "This prints a passed info into this function"  print "Name: ", name  print "Age ", age  return;  # Now you can call printinfo function  printinfo( age=50, name="miki" )  printinfo( name="miki" ) | | | |