**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **04-June-2020** | **Name:** | **Raziya Banu** |
| **Course:** | **HDL** | **USN:** | **4AL16EC058** |
| **Topic:** | **Hardware modelling** | **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 Hardware modelling using verilog.  **ABSTRACT HARDWARE MODELLING USING AN OBJECTORIENTED LANGUAGE EXTENSION TOVHDL**  Guido Schumacher, Wolfgang Nebel Department of Computer Science, Carl von Ossietzky University Oldenburg, 26111 Oldenburg, Germany ABSTRACT Reusability of hardware models becomes more and more important in the design of complex systems. This is explained for different modelling problems. Methods for re-use, which successfully use abstraction as a key to manage the design complexity in the software domain, are analysed with respect to their applicability to hardware design. Object-oriented modelling is stated as a technique which potentially increases design productivity. Attempts to adapt such techniques for hardware specification and design are presented The lack of support for object-oriented modelling in the current version of VHDL is explained. Possible subjects for object orientation are discussed We illustrate how an object-oriented extension to the hardware description language VHDL could be used to describe hardware systems at a high level of abstraction. An explanation is given that it is not sufficient to introduce object-oriented language constructs into a parallel hardware description language, it is also necessary "to provide high-level communication constructs for the data exchange between the parallel processes.  **INTRODUCTION**  The non-saturating integration density of integrated circuits (despite of earlier assumptions) provides sufficient motivation to investigate further possibilities to increase design productivity. The increase of complexity can be estimated at about ten every seven years. It cannot be compensated by the increase of computing power, which on the one hand does not achieve an increase in intellectual value addition without a change in design methodology, while on the other is consumed mostly by complex data management systems and user-friendly graphical interfaces.  As keys to manage the design complexity and the increase of design effort typically the terms hierarchy and abstraction are mentioned. A hierarchical, structural decomposition of the design problem reduces the complexity by isolating subproblems and hence makes the global design problem accessible for a solution. This achievement has to be paid for with a suboptimal total solution due to the local optimization of the substructures. Abstraction should be looked at from different points of view. First, it allows us to encapsulate existing components which can be described by an abstract model containing only that information which is required at the higher level of abstraction. Examples of this kind of abstraction are cell libraries of ASIC vendors. Here abstraction allows the re-use of hardware components. They do not need to be redesigned for each application, i.e. the design cost (measured in terms of transistors) is reduced.  On the other hand, abstraction can be used for reducing design effort if design detail can automatically be attributed to less detailed design specifications using synthesis tools. Application examples are: logic synthesis, technology mapping, place and route. Here the reduction in design effort is due to a re-use of automated design strategies and architectures. At the system level, abstraction is required to describe a first specification without confusing and often unknown implementation details. It allows a first check of the system description for inconsistencies and errors and therefore reduces the risk of redesign steps in further design phases. 6.1.1. Traditional Approaches of Re-Use In the past, the method of design data encapsulation was developed in an evolutionary way in the direction of higher levels of abstraction. Prototyping LDPC Codes in Hardware Any hardware design that is intended for practical applications requires implementation of prototype models on the hardware for testing [13]. Field Programmable Gate Array (FPGA) is a very flexible and widely used platform for [rapid prototyping](https://www.sciencedirect.com/topics/engineering/rapid-prototyping), and is also a low-cost approach compared to [ASIC](https://www.sciencedirect.com/topics/engineering/application-specific-integrated-circuits) implementation.  FPGA is an integrated circuit that contains large numbers of identical logic cells that can be interconnected by a matrix of wires using programmable switch boxes [14]. A design can be implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. The array of logic cells and the mesh of interconnecting wires form the basic building block of an FPGA. Complex designs can be implemented by programming these basic building blocks [15].  FPGAs offer a number of benefits over other implementation flows such as ASIC and off-the-shelf DSP and microcontroller chips. Some of the benefits of using FPGA are as follows :   1. Performance: FPGAs offer logic structures that provide the advantage of incorporating [parallelism](https://www.sciencedirect.com/topics/engineering/parallelism) in designs and thereby significantly enhance the computational speed compared to processor-based platforms. 2. Reliability: Processor-based designs operate on instructions to perform a particular task using shared hardware resources. However, FPGA-based designs consist of dedicated hardware for performing such tasks with predictable delays. Hence, increasing the reliability of real-time systems.   3. Long-term maintenance: FPGAs provide flexibility in upgrading the design in case of a change in specification of an application over time. The time spent in redesigning/enhancing a FPGA-based design is much less compared to that of ASIC design.  4.  Cost: The Non-Recurring Engineering (NRE) cost for designing a custom ASIC is huge compared to FPGA-based solutions. FPGAs bypass the backend physical design, [fabrication](https://www.sciencedirect.com/topics/engineering/fabrication), and packaging cost as well.  5.  Time to Market: FPGA technology provides flexibility for rapid prototyping of the design by avoiding fabrication and other processing delays, thus facilitating quicker “time to market” solutions.  The major limitation of FPGA-based design is in its overall performance compared to ASIC solutions. However, it provides the best methodology for quick prototyping and testing of the designs including the above listed advantages.  Implement a simple T Flipflop and test the module using a compiler.    module tff ( input clk,  input rstn,  input t,  output reg q);    always @ (posedge clk) begin  if (!rstn)  q <= 0;  else  if (t)  q <= ~q;  else  q <= q;  end  endmodule |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **04-June-2020** | **Name:** | **Raziya Banu** | |
| **Course:** | **Udemy** | **USN:** | **4AL16EC058** | |
| **Topic:** | **Return Functions in Python** | **Semester & Section:** | **8th sem & ‘B’ section** | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Python return statement:**Parameters and Arguments A function or procedure usually needs some information about the environment, in which it has been called. The interface between the environment, from which the function has been called, and the function, i.e. the function body, consists of special variables, which are called parameters. By using these parameters, it's possible to use all kind of objects from "outside" inside of a function. The syntax for how parameters are declared and the semantics for how the arguments are passed to the parameters of the function or procedure depends on the programming language.  Very often the terms parameter and argument are used synonymously, but there is a clear difference. Parameters are inside functions or procedures, while arguments are used in procedure calls, i.e. the values passed to the function at run-time.  "call by value" and "call by name"  The evaluation strategy for arguments, i.e. how the arguments from a function call are passed to the parameters of the function, differs between programming languages. The most common evaluation strategies are "call by value" and "call by reference":   * Call by Value The most common strategy is the call-by-value evaluation, sometimes also called pass-by-value. This strategy is used in C and C++, for example. In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, its value will be assigned (copied) to the corresponding parameter. This ensures that the variable in the caller's scope will stay unchanged when the function returns. * Call by Reference In call-by-reference evaluation, which is also known as pass-by-reference, a function gets an implicit reference to the argument, rather than a copy of its value. As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed. By using Call by Reference we save both computation time and memory space, because arguments do not need to be copied. On the other hand this harbours the disadvantage that variables can be "accidentally" changed in a function call. So, special care has to be taken to "protect" the values, which shouldn't be changed. Many programming languages support call-by-reference, like C or C++, but Perl uses it as default.   In ALGOL 60 and COBOL there has been a different concept called call-by-name, which isn't used anymore in modern languages.  and what about Python?  There are some books which call the strategy of Python call-by-value, and some call it call-by-reference. You may ask yourself, what is right.  Humpty Dumpty supplies the explanation:  --- "When I use a word," Humpty Dumpty said, in a rather a scornful tone, "it means just what I choose it to mean - neither more nor less."  --- "The question is," said Alice, "whether you can make words mean so many different things."  --- "The question is," said Humpty Dumpty, "which is to be master - that's all."  Lewis Carroll, Through the Looking-Glass  To come back to our initial question what evaluation strategy is used in Python: The authors who call the mechanism call-by-value and those who call it call-by-reference are stretching the definitions until they fit.  Correctly speaking, Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".  Paramterübergabe  If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place within the function. If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.  First, let's have a look at the integer variables below. The parameter inside the function remains a reference to the argument's variable, as long as the parameter is not changed. As soon as a new value is assigned to it, Python creates a separate local variable. The caller's variable will not be changed this way:  def ref\_demo(x):  print("x=",x," id=",id(x))  x=42  print("x=",x," id=",id(x))  Parameterübergabe  In the example above, we used the id() function, which takes an object as a parameter. id(obj) returns the "identity" of the object "obj". This identity, the return value of the function, is an integer which is unique and constant for this object during its lifetime. Two different objects with non-overlapping lifetimes may have the same id() value.  If you call the function ref\_demo() of the previous example - like we do in the green block further below - we can check what happens to x with the id() function: We can see that in the main scope, x has the identity 140709692940944. In the first print statement of the ref\_demo() function, the x from the main scope is used, because we can see that we get the same identity. After we assigned the value 42 to x, x gets a new identity 140709692942000, i.e. a separate memory location from the global x. So, when we are back in the main scope x has still the original value 9 and the id 140709692940944.  In other words, Python initially behaves like call-by-reference, but as soon as we change the value of such a variable, i.e. as soon as we assign a new object to it, Python "switches" to call-by-value. That is, a local variable x will be created and the value of the global variable x will be copied into it.  x = 9  id(x)  Output::  140709692940944  ref\_demo(x)  x= 9 id= 140709692940944  x= 42 id= 140709692942000  id(x)  Output::  140709692940944  Side effects  A function is said to have a side effect, if, in addition to producing a return value, it modifies the caller's environment in other ways. For example, a function might modify a global or static variable, modify one of its arguments, raise an exception, write data to a display or file etc.  There are situations, in which these side effects are intended, i.e. they are part of the function's specification. But in other cases, they are not wanted , they are hidden side effects. In this chapter, we are only interested in the side effects that change one or more global variables, which have been passed as arguments to a function. Let's assume, we are passing a list to a function. We expect the function not to change this list. First, let's have a look at a function which has no side effects. As a new list is assigned to the parameter list in func1(), a new memory location is created for list and list becomes a local variable.  def no\_side\_effects(cities):  print(cities)  cities = cities + ["Birmingham", "Bradford"]  print(cities)  locations = ["London", "Leeds", "Glasgow", "Sheffield"]  no\_side\_effects(locations)  ['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']  ['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg', 'Birmingham', 'Bradford']  print(locations)  ['London', 'Leeds', 'Glasgow', 'Sheffield']  This changes drastically, if we increment the list by using augmented assignment operator +=. To show this, we change the previous function rename it as "side\_effects" in the following example:  def side\_effects(cities):  print(cities)  cities += ["Birmingham", "Bradford"]  print(cities)    locations = ["London", "Leeds", "Glasgow", "Sheffield"]  side\_effects(locations)  ['London', 'Leeds', 'Glasgow', 'Sheffield']  ['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham', 'Bradford']  print(locations)  ['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham', 'Bradford']  We can see that Birmingham and Bradford are included in the global list locations as well, because += acts as an in-place operation.  The user of this function can prevent this side effect by passing a copy to the function. A shallow copy is sufficient, because there are no nested structures in the list. To satisfy our French customers as well, we change the city names in the next example to demonstrate the effect of the slice operator in the function call:  def side\_effects(cities):  print(cities)  cities += ["Paris", "Marseille"]  print(cities)    locations = ["Lyon", "Toulouse", "Nice", "Nantes", "Strasbourg"]  side\_effects(locations[:])  print(locations)  ['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']  ['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg', 'Paris', 'Marseille']  ['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']  print(locations)  ['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']  We can see that the global list locations has not been effected by the execution of the function.  Command Line Arguments  If you use a command line interface, i.e. a text user interface (TUI) , and not a graphical user interface (GUI), command line arguments are very useful. They are arguments which are added after the function call in the same line.  It's easy to write Python scripts using command line arguments. If you call a Python script from a shell, the arguments are placed after the script name. The arguments are separated by spaces. Inside the script these arguments are accessible through the list variable sys.argv. The name of the script is included in this list sys.argv[0]. sys.argv[1] contains the first parameter, sys.argv[2] the second and so on. The following script (arguments.py) prints all arguments:  # Module sys has to be imported:  import sys Iteration over all arguments: for eachArg in sys.argv:  print(eachArg)    Example call to this script:  python argumente.py python course for beginners  This call creates the following output:  argumente.py  python  course  for  beginners  Variable Length of Parameters | | | |