**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **06-June-2020** | **Name:** | **Raziya Banu** |
| **Course:** | **HDL** | **USN:** | **4AL16EC058** |
| **Topic:** | **FPGA Embedded Design** | **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 Embedded Design **Making FPGAs work for embedded developers:**    FPGA technology has been available since the mid 1980s. In embeddedsystems development, FPGAs have a long history of use as a containerfor the plethora of glue logic surrounding microprocessor applications.  While FPGA technology has the potential to speed embeddeddevelopment and allow new approaches to be taken to design in general,most embedded software developers still tend to work just as before –selecting a discrete hardware processor at the beginning of the designcycle, creating the physical platform then writing the software to makeuse of this platform.  This lack of exploitation of 'soft' hardware in embedded systemsdevelopment can largely be attributed to the lack of tools to allow Cprogrammers to use their skills at the programmable hardware level. Thedesign of programmable devices must be better integrated with currentembedded design flows and made accessible to the mainstream of embeddedengineers and software developers in order for the full potential ofFPGAs to be realized in the embedded world.  So traditional FPGA design flows are not attractive to embeddeddevelopers because, even though they are software based, they stillrely on the designer understanding and being willing to design at thehardware level. What embedded developers want is to be able to do isremain at the algorithm level.  This can include the processor functionality itself, and today softprocessors hosted in FPGAs are becoming more common as embeddedplatforms. At the fundamental level, moving to FPGA-hosted softprocessors will deliver advantages in terms of architecture flexibilityplus smaller, simpler boards. But there is a deep level of utilizationthat offers even more compelling benefits.  https://www.embedded.com/wp-content/uploads/media-1043024-altiumfpgafig1.jpg  Figure: Large scale FPGAs provide a versatile reconfigurable hardwareplatform for embedded systems design that can provide generic interfaceto both soft and discrete system devices and allow easy interchange ofprocessors and peripherals.  The advantage of such a system is that the processor choice wouldnot need to be made 'up front'. The system could be developed using oneprocessor, but moved to a faster device if, during development, it wasfound that more performance was needed. **8:3 Encoder**  `timescale 1ns / 1ps  module Encoder(d0,d1,d2,d3,d4,d5,d6,d7,a,b,c);  input d0,d1,d2,d3,d4,d5,d6,d7;  output a,b,c;  or(a,d4,d5,d6,d7);  or(b,d2,d3,d6,d7);  or(c,d1,d3,d5,d7);  endmodule |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **06-June-2020** | **Name:** | **Raziya Banu** | |
| **Course:** | **Udemy** | **USN:** | **4AL16EC058** | |
| **Topic:** | **\*args and \*\*kwargs in Python** | **Semester & Section:** | **8th sem & ‘B’ section** | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Python \*args and \*\*kwargs:**Passing Multiple Arguments to a Function **\*args** and **\*\*kwargs** allow you to pass multiple arguments or keyword arguments to a function. Consider the following example. This is a simple function that takes two arguments and returns their sum:  def my\_sum(a, b):  return a + b  This function works fine, but it’s limited to only two arguments. Using the Python args Variable in Function Definitions There are a few ways you can pass a varying number of arguments to a function. The first way is often the most intuitive for people that have experience with collections. You simply pass a list or a [set](https://realpython.com/python-sets/) of all the arguments to your function. So for my\_sum(), you could pass a list of all the integers you need to add:  # sum\_integers\_list.py  def my\_sum(my\_integers):  result = 0  for x in my\_integers:  result += x  return result  list\_of\_integers = [1, 2, 3]  print(my\_sum(list\_of\_integers))  This implementation works, but whenever you call this function you’ll also need to create a list of arguments to pass to it. This can be inconvenient, especially if you don’t know up front all the values that should go into the list.  This is where \*args can be really useful, because it allows you to pass a varying number of positional arguments. Take the following example:  # sum\_integers\_args.py  def my\_sum(\*args):  result = 0  # Iterating over the Python args tuple  for x in args:  result += x  return result  print(my\_sum(1, 2, 3))  In this example, you’re no longer passing a list to my\_sum(). Instead, you’re passing three different positional arguments. my\_sum() takes all the parameters that are provided in the input and packs them all into a single iterable object named args.  Note that **args is just a name.** You’re not required to use the name args. You can choose any name that you prefer, such as integers:  # sum\_integers\_args\_2.py  def my\_sum(\*integers):  result = 0  for x in integers:  result += x  return result  print(my\_sum(1, 2, 3))  The function still works, even if you pass the iterable object as integers instead of args. All that matters here is that you use the **unpacking operator** (\*).  Bear in mind that the iterable object you’ll get using the unpacking operator \* is [not a list but a tuple](https://realpython.com/python-lists-tuples/). A tuple is similar to a list in that they both support slicing and iteration. However, tuples are very different in at least one aspect: lists are [mutable](https://realpython.com/courses/immutability-python/), while tuples are not. To test this, run the following code. This script tries to change a value of a list:  # change\_list.py  my\_list = [1, 2, 3]  my\_list[0] = 9  print(my\_list)  The value located at the very first index of the list should be updated to 9. If you execute this script, you will see that the list indeed gets modified:  $ python change\_list.py  [9, 2, 3]  The first value is no longer 0, but the updated value 9. Now, try to do the same with a tuple:  # change\_tuple.py  my\_tuple = (1, 2, 3)  my\_tuple[0] = 9  print(my\_tuple)  Here, you see the same values, except they’re held together as a tuple. If you try to execute this script, you will see that the Python interpreter returns an [error](https://realpython.com/python-exceptions/):  $ python change\_tuple.py  Traceback (most recent call last):  File "change\_tuple.py", line 3, in <module>  my\_tuple[0] = 9  TypeError: 'tuple' object does not support item assignment  This is because a tuple is an immutable object, and its values cannot be changed after assignment. Keep this in mind when you’re working with tuples and \*args. Using the Python kwargs Variable in Function Definitions what \*args is for, but what about \*\*kwargs? \*\*kwargs works just like \*args, but instead of accepting positional arguments it accepts keyword (or **named**) arguments. Take the following example:  # concatenate.py  def concatenate(\*\*kwargs):  result = ""  # Iterating over the Python kwargs dictionary  for arg in kwargs.values():  result += arg  return result  print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))  When you execute the script above, concatenate() will iterate through the Python kwargs [dictionary](https://realpython.com/python-dicts/) and concatenate all the values it finds:  $ python concatenate.py  RealPythonIsGreat!  Like args, kwargs is just a name that can be changed to whatever you want. Again, what is important here is the use of the **unpacking operator** (\*\*).  So, the previous example could be written like this:  # concatenate\_2.py  def concatenate(\*\*words):  result = ""  for arg in words.values():  result += arg  return result  print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))  Note that in the example above the iterable object is a standard dict. If you [iterate over the dictionary](https://realpython.com/iterate-through-dictionary-python/) and want to return its values, like in the example shown, then you must use .values().  In fact, if you forget to use this method, you will find yourself iterating through the **keys** of your Python kwargs dictionary instead, like in the following example:  # concatenate\_keys.py  def concatenate(\*\*kwargs):  result = ""  # Iterating over the keys of the Python kwargs dictionary  for arg in kwargs:  result += arg  return result  print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))  Now, if you try to execute this example, you’ll notice the following output: | | | |