**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **12-06-2020** | **Name:** | **Karthik J** |
| **Course:** | VLSI | **USN:** | **4AL16EC030** |
| **Topic:** | CMOS Inverter basics | **Semester & Section:** | **8TH A** |
| **GitHub Repository:** | Karthik-J |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **CMOS Inverter** For the investigation of circuit-level degradation a CMOS (complementary MOS) inverter is analyzed. A major advantage of CMOS technology is the ability to easily combine complementary transistors, n-channel and p-channel, on a single substrate. The CMOS inverter consists of the two transistor types which are processed and connected, as seen schematically in Figure [7.10](https://www.iue.tuwien.ac.at/phd/entner/node33.html#f:cmos-inverter-dopings).   |  | | --- | | Image cmos-inverter-dopings | | **Figure 7.10:** Schematic of a CMOS inverter as processed on a p-type silicon substrate. The effect of NBTI mainly impacts the p-channel MOSFET (right hand side transistor). |   The p-channel MOSFET relies on an n-type substrate. As commonly p-type wafers are used for processing, an additional n-type well implant is necessary. In this well, which is a deep region of n-type doping, the p-channel MOSFET is placed. As the p-substrate and the n-well junction is reverse biased, no significant current flows between these regions and the two transistors are isolated.  The output current of the p-channel MOSFET is typically much lower than the current of an n-channel MOSFET with similar dimensions and dopings. This is due to the lower carrier mobility of holes compared to electrons. As the characteristics of the complementary transistors should be as equal as possible, the width of the p-channel MOSFET is typically made larger to compensate the difference. In our example device the necessary geometry factor is $3.5$to obtain equal drain currents for equal gate biases.   |  | | --- | | \includegraphics[width=8cm]{figures/cmos-inverter-circuit} | | **Figure 7.11:** Schematic of a CMOS inverter circuit. In the stationary case the circuit does not consume any power when assuming perfect devices without leakage current. NBT stress is imposed on the p-channel device at $\ensuremath {V_\textrm {in}}= \ensuremath {V_\textrm {low}}$. |   Figure [7.11](https://www.iue.tuwien.ac.at/phd/entner/node33.html#f:cmos-inverter-circuit) gives the schematic of the CMOS inverter circuit. It can be seen that the gates are at the same bias \ensuremath {V_\textrm {in}}which means that they are always in a complementary state. When \ensuremath {V_\textrm {in}}is high, $\ensuremath{V_\textrm{in}}\approx \ensuremath{V_\textrm{dd}}$, the voltage between gate and substrate of the nMOS transistor is also approximately \ensuremath{V_\textrm{dd}}and the transistor is in on-state. The gate-substrate bias at the pMOS on the other side is nearly zero and the transistor is turned off. The output voltage \ensuremath {V_\textrm {out}}is pulled to ground, which is the low state. When the input voltage is in a high-state, the complementary situation occurs and the pMOSFET is turned on while the nMOSFET is turned off. The output voltage is therefore pulled to \ensuremath{V_\textrm{dd}}which is the high-state. It is important to note that in both states, high and low, no static current flows through the inverter. This is of course only valid when assuming ideal devices with zero off- and leakage-currents.  Considering negative bias temperature instability, the worst stress conditions are imposed on the p-channel MOSFET at $\ensuremath {V_\textrm {in}}= \ensuremath {V_\textrm {low}}$. At this bias condition the pMOSFET is turned on, with approximately the same potential at the source and the drain $\ensuremath{V_\textrm{gs}}= \ensuremath{V_\textrm{gd}}= \ensuremath{V_\textrm{dd}}$and negative gate to substrate voltage $\ensuremath{V_\textrm{gsub}}= -\ensuremath{V_\textrm{dd}}$. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date:** | | **12-06-2020** | **Name:** | **Karthik J** |  |
| **Course:** | | CNN for Computer Vision with Keras and TensorFlow in Python | **USN:** | **4AL16EC030** |  |
| **Topic:** | |  | **Semester & Section:** | **8th A** |  |
|  | **AFTERNOON SESSION DETAILS** | | | | |
|  | **Image of session** | | | | |
|  | NumPy NumPy is a Python package used for numerical computation. NumPy is one of the foundational packages for scientific computing with Python. NumPy's core data type is the array and NumPy functions operate on arrays. Installing NumPyInstalling NumPy Before NumPy's functions and methods can be used, NumPy must be installed. Depending on which distribution of Python you use, the installation method is slightly different. Install NumPy on Anaconda If you installed the Anaconda distribution of Python, NumPy comes pre-installed and no further installation steps are necessary.  If you use a version of Python from python.org or a version of Python that came with your operating system, the **Anaconda Prompt** and **conda** or **pip** can be used to install NumPy. Install NumPy with the Anaconda Prompt To install NumPy, open the **Anaconda Prompt** and type:  > conda install numpy  Type y for yes when prompted. Install NumPy with pip To install NumPy with **pip**, bring up a terminal window and type:  $ pip install numpy  This command installs NumPy in the current working Python environment. Verify NumPy installation To verify NumPy is installed, invoke NumPy's version using the Python REPL. Import NumPy and call the .\_\_version\_\_ attribute common to most Python packages.  In [1]:  import numpy as np  np.**version**  Out[1]:  '1.16.4'  A version number like '1.16.4' indicates a successful NumPy installation. Python Lists and NumPy Arrays NumPy is a Python package used for numerical calculations, working with arrays of homogeneous values, and scientific computing. This section introduces NumPy arrays then explains the difference between Python lists and NumPy arrays. Python Lists NumPy is used to construct homogeneous arrays and perform mathematical operations on arrays. A NumPy array is different from a Python list. The data types stored in a Python list can all be different.  python\_list = [ 1, -0.038, 'gear', True]  The Python list above contains four different data types: 1 is an integer, -0.038 is a float, 'gear' is a string, and 'True' is a boolean.  The code below prints the data type of each value store in python\_list.  In [1]:  python\_list = [1, -0.038, 'gear', True]  for item in python\_list:  print(type(item))  <class 'int'>  <class 'float'>  <class 'str'>  <class 'bool'> NumPy Arrays The values stored in a NumPy array must all share the same data type. Consider the NumPy array below:  np.array([1.0, 3.1, 5e-04, 0.007])  All four values stored in the NumPy array above share the same data type: 1.0, 3.1, 5e-04, and 0.007 are all floats.  The code below prints the data type of each value stored in the NumPy array above.  In [2]:  import numpy as np  for value in np.array([1.0, 3.1, 5e-04, 0.007]):  print(type(value))  <class 'numpy.float64'>  <class 'numpy.float64'>  <class 'numpy.float64'>  <class 'numpy.float64'>  If the same four elements stored in the previous Python list are stored in a NumPy array, NumPy forces all of the four items in the list to conform to the same data type.  In the next code section, all four items are converted to type '<U32', which is a string data type in NumPy (the U refers Unicode strings; all strings in Python are Unicode by default).  In [3]:  np.array([1, -0.038, 'gear', True])  Out[3]:  array(['1', '-0.038', 'gear', 'True'], dtype='<U32')  NumPy arrays can also be two-dimensional, three-dimensional, or up to n-dimensional. In practice, computer resources limit array size. Remember that regardless of size, all elements in a NumPy array must be the same type. NumPy arrays are useful because mathematical operations can be run on an entire array simultaneously. If numbers are stored in a regular Python list and the list is multiplied by a scalar, the list extends and repeats- instead of multiplying each number in the list by the scalar.  The code below demonstrates list repetition using the multiplication operator, \*.  In [4]:  lst = [1, 2, 3, 4]  lst\*2  Out[4]:  [1, 2, 3, 4, 1, 2, 3, 4]  To multiply each element in a Python list by the number 2, a loop can be used:  In [5]:  lst = [1, 2, 3, 4]  for i, item in enumerate(lst):  lst[i] = lst[i]\*2  lst  Out[5]:  [2, 4, 6, 8]  The method above is relatively cumbersome and is also quite computationally expensive. An operation that is computationally expensive is an operation that takes a lot of processing time or storage resources like RAM and CPU bandwidth.  Another way to complete the same operation in the loop above is to use a NumPy array. Array Multiplication An entire NumPy array can be multiplied by a scalar in one step. The scalar multiplication operation below produces an array with each element multiplied by the scalar 2.  In [6]:  nparray = np.array([1,2,3,4])  2\*nparray  Out[6]:  array([2, 4, 6, 8])  If we have a very long list of numbers, we can compare the amount of time it takes each of the two computation methods above, a list with a loop compared to array multiplication to complete the same operation. This comparison highlights an advantage of arrays compared to lists- speed. Timing Arrays Jupyter notebooks have a nice built-in method to time how long a line of code takes to execute. In a Jupyter notebook, when a line starts with %timeit followed by code, the kernel runs the line of code multiple times and outputs an average of the time spent to execute the line of code.  We can use %timit to compare a mathematical operation on a Python list using a for loop to the same mathematical operation on a NumPy array.  In [7]:  lst = list(range(10000))  %timeit for i, item in enumerate(lst): lst[i] = lst[i]\*2  3.21 ms ± 958 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)  In [8]:  nparray = np.arange(0,10000,1)  %timeit 2\*nparray  7.11 µs ± 200 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)  With 10,000 integers, the Python list and for loop takes an average of single milliseconds, while the NumPy array completes the same operation in tens of microseconds. This is a speed increase of over 100x by using the NumPy array (1 millisecond = 1000 microseconds).  For larger lists of numbers, the speed increase using NumPy is considerable. Array Slicing Multiple values stored within an array can be accessed simultaneously with array slicing. To pull out a section or slice of an array, the colon operator : is used when calling the index. The general form is:  <slice> = <array>[start:stop]  Where <slice> is the slice or section of the array object <array>. The index of the slice is specified in [start:stop]. Remember Python counting starts at 0 and ends at n-1. The index [0:2] pulls the first two values out of an array. The index [1:3] pulls the second and third values out of an array.  An example of slicing the first two elements out of an array is below.  In [1]:  import numpy as np  a = np.array([2, 4, 6])  b = a[0:2]  print(b)  [2 4] Array Indexing Elements in NumPy arrays can be accessed by indexing. Indexing is an operation that pulls out a select set of values from an array. The index of a value in an array is that value's location within the array. There is a difference between the value and where the value is stored in an array.  An array with 3 values is created in the code section below.  In [1]:  import numpy as np  a = np.array([2,4,6])  print(a)  [2 4 6]  The array above contains three values: 2, 4 and 6. Each of these values has a different index.  **Remember counting in Python starts at 0 and ends at n-1.**  The value 2 has an index of 0. We could also say 2 is in location 0 of the array. The value 4 has an index of 1 and the value 6 has an index of 2. The table below shows the index (or location) of each value in the array.   | **Index (or location)** | **Value** | | --- | --- | | 0 | 2 | | 1 | 4 | | 2 | 6 |   Individual values stored in an array can be accessed with indexing.  The general form to index a NumPy array is below:  <value> = <array>[index]  Where <value> is the value stored in the array, <array> is the array object name and [index] specifies the index or location of that value.  In the array above, the value 6 is stored at index 2.  In [2]:  import numpy as np  a = np.array([2,4,6])  print(a)  value = a[2]  print(value)  [2 4 6]  6  Python.org downloads page showing download for Windows button | | | | |