**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **13-06-2020** | **Name:** | **Karthik J** |
| **Course:** | VLSI | **USN:** | **4AL16EC030** |
| **Topic:** | MOSFET by IIT-G | **Semester & Section:** | **8TH A** |
| **GitHub Repository:** | Karthik-J |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| MOSFET Introduction The metal–oxide–semiconductor field-effect transistor (MOSFET) is a transistor used for amplifying or switching electronic signals. In MOSFETs, a voltage on the oxide-insulated gate electrode can induce a conducting channel between the two other contacts called source and drain. The channel can be of n-type or p-type, and is accordingly called an nMOSFET or a pMOSFET. Figure 1 shows the schematic diagram of the structure of an nMOS device before and after channel formation.    Figure 2 shows symbols commonly used for MOSFETs where the bulk terminal is either labeled (B) or implied (not drawn).    Fig. (2): Circuit symbols for nMOS and pMOS respectively Output Characteristics MOSFET output characteristics plot ID versus VDS for several values of VGS.    The characteristics of an nMOS transistor can be explained as follows. As the voltage on the top electrode increases further, electrons are attracted to the surface. At a particular voltage level, which we will shortly define as the threshold voltage, the electron density at the surface exceeds the hole density. At this voltage, the surface has inverted from the p-type polarity of the original substrate to an n-type inversion layer, or inversion region, directly underneath the top plate as indicated in Fig. 1(b). This inversion region is an extremely shallow layer, existing as a charge sheet directly below the gate. In the MOS capacitor, the high density of electrons in the inversion layer is supplied by the electron–hole generation process within the depletion layer. The positive charge on the gate is balanced by the combination of negative charge in the inversion layer plus negative ionic acceptor charge in the depletion layer. The voltage at which the surface inversion layer just forms plays an extremely important role in field-effect transistors and is called the threshold voltage Vtn. The region of output characteristics where VGStn and no current flows is called the cutt-off region. When the channel forms in the nMOS (pMOS) transistor, a positive (negative) drain voltage with respect to the source creates a horizontal electric field moving the electrons (holes) toward the drain forming a positive (negative) drain current coming into the transistor. The positive current convention is used for electron and hole current, but in both cases electrons are the actual charge carriers. If the channel horizontal electric field is of the same order or smaller than the vertical thin oxide field, then the inversion channel remains almost uniform along the device length. This continuous carrier profile from drain to source puts the transistor in a bias state that is equivalently called either the non-saturated, linear, or ohmic bias state. The drain and source are effectively short-circuited. This happens when VGS > VDS + Vtn for nMOS transistor and VGS < VDS +Vtp for pMOS transistor. Drain current is linearly related to drain-source voltage over small intervals in the linear bias state.  But if the nMOS drain voltage increases beyond the limit, so that VGS < VDS + Vtn, then the horizontal electric field becomes stronger than the vertical field at the drain end, creating an asymmetry of the channel carrier inversion distribution shown in Figure 4.    Fig. 4: Channel pinchoff for (a) nMOS and (b) pMOS transistor devices.  If the drain voltage riseswhile the gate voltage remains the same, then VGD can go below the threshold voltage in the drain region. There can be no carrier inversion at the drain-gate oxide region, so the inverted portion of the channel retracts from the drain, and no longer “touches” this terminal. The pinched-off portion of the channel forms a depletion region with a high electric field. The n-drain and p-bulk form a pn junction. When this happens the inversion channel is said to be “pinched-off” and the device is in the saturation region. The characteristics can be loosely modelled by the following equations.   Transfer Characteristics The transfer characteristic relates drain current (ID) response to the input gate-source driving voltage (VGS). Since the gate terminal is electrically isolated from the remaining terminals (drain, source, and bulk), the gate current is essentially zero, so that gate current is not part of device characteristics. The transfer characteristic curve can locate the gate voltage at which the transistor passes current and leaves the OFF-state. This is the device threshold voltage (Vtn). Figure 5 shows measured input characteristics for an nMOS and pMOS transistor with a small 0.1V potential across their drain to source terminals.    The transistors are in their non-saturated bias states. As VGS increases for the nMOS transistor in Figure 5a, the threshold voltage is reached where drain current elevates. For VGS between 0V and 0.7V, ID is nearly zero indicating that the equivalent resistance between the drain and source terminals is extremely high. Once VGS reaches 0.7V, the current increases rapidly with VGS indicating that the equivalent resistance at the drain decreases with increasing gate-source voltage. Therefore, the threshold voltage of the given nMOS transistor is about Vtn ≈ 0.7V. The pMOS transistor input characteristic in Figure 5b is analogous to the nMOS transistor except the ID and VGS polarities are reversed.  4x1 Multiplexer Introduction A multiplexer or mux is a combinational circuits that selects several analog or digital input signals and forwards the selected input into a single output line. A multiplexer of 2n inputs has n selected lines, are used to select which input line to send to the output.    Figure 2 shows how a 4:1 MUX can be constructed out of two 2:1 MUXs.   Design using pass-transistor logic A multiplexer can be designed using various logics. Fig.3 shows how a 2:1 MUX is implemented using a pass-transistor logic.GS.    The pass-transistor logic attempts to reduce the number of transistors to implement a logic by allowing the primary inputs to drive gate terminals as well as source-drain terminals. The implementation of a 2:1 MUX requires 4 transistors (including the inverter required to invert S), while a complementary CMOS implementation would require 6 transistors. The reduced number of devices has the additional advantage of lower capacitance. Design using transmission gate logic A transmission gate is an electronic element and good non mechanical relay built with CMOS technology. It is made by parallel combination of nMOS and pMOS transistors with the input at the gate of one transistor (C) being complementary to the input at the gate () of the other. The symbol of a transmission gate is shown below in fig.4.    The transmission gate acts as a bidirectional switch controlled by the gate signal C. When C=1, both MOSFETs are on, allowing the signal to pass through the gate. In short, A=B, if C=1. On the other hand, C=0, places both transistors in cut-off, creating an open circuit between nodes A and B. Fig.5 shows the implementation of a 2:1 MUX using transmission gate logic.    Here, the transmission gates selects input A or B on the basis of the value of the control signal S. When S=0, Z=A and when S=1, Z=B. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date:** | | **13-06-2020** | **Name:** | **Karthik J** |  |
| **Course:** | | CNN for Computer Vision with Keras and TensorFlow in Python | **USN:** | **4AL16EC030** |  |
| **Topic:** | |  | **Semester & Section:** | **8th A** |  |
|  | **AFTERNOON SESSION DETAILS** | | | | |
|  | **My work:** <https://github.com/Karthikjsannakki/CNN>  **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 | | | | |