ARITHEMATIC
VERILOG ASSIGNMENT QUESTIONS

#ripple carry adder

1.Implement a 8-bit ripple carry adder module in Verilog. The module should have two 8-bit inputs (A and B), a 1-bit carry-in input (Cin), an 8-bit output (Sum), and a 1-bit carry-out output (Cout). Use full adder modules to design the 8-bit ripple carry adder.

2.Design a full adder module in Verilog. The module should have three 1-bit inputs (A, B, and Cin), two 1-bit outputs (Sum and Cout). Use this full adder module to construct a 16-bit ripple carry adder.

3.Create a parameterized ripple carry adder module in Verilog. The module should allow the user to specify the bit width (N) of the inputs and outputs. Implement the adder using the parameterized bit width, and include appropriate parameterized testbench code to verify its functionality.

#carry look ahead adder

4.Design a 4-bit Carry Look-Ahead Adder module in Verilog that takes two 4-bit inputs (A and B) and produces a 4-bit output Sum. Assume that the carry input is provided separately. Implement the logic for the carry generation and propagation using the Carry Look-Ahead approach.

5.Design an 8-bit Carry Look-Ahead Adder module in Verilog that takes two 8-bit inputs (A and B) and produces an 8-bit output Sum. Also, include an input Cin for the carry-in. Write a testbench to simulate the   behaviour of the module and test it with various input combinations.

6.Design a 8-bit adder module that combines a 4-bit carry look-ahead adder and a 4-bit ripple carry adder. The module should take two 8-bit inputs (A and B) and produce an 8-bit output Sum. Additionally, provide a Cin input for the carry-in.

7.Enhance the performance of a 16-bit ripple carry adder by using carry look-ahead techniques for the higher-order bits (8 to 15) and a ripple carry adder for the lower-order bits (0 to 7). Implement this hybrid adder in Verilog

8.Develop an overflow detection circuit for a 12-bit addition operation. Use a carry look-ahead adder for the addition and a ripple carry adder for detecting the overflow condition. Implement the circuit in Verilog and show how the overflow detection signal is generated based on the carry and sum outputs.

#carry skip adder

9.Design a 16-bit carry-skip adder module in Verilog that takes two 8-bit inputs (A and B) and produces an 8-bit output Sum. Also, include an input Cin for the carry-in. Write a testbench to simulate the behaviour of the module and test it with various input combinations.

10.Design a 8-bit adder circuit in Verilog that combines a 4-bit carry skip adder and a 4-bit carry look-ahead adder. Provide inputs A and B for the adder and output signals for the sum and carry-out.

11.Create a hierarchical Verilog design for a 16-bit adder using an 8-bit carry skip adder and an 8-bit carry look-ahead adder. Instantiate the sub-modules, connect inputs and outputs, and provide stimulus to verify the functionality.

#carry select adder

12.Design a 4-bit Carry-Select Adder module in Verilog that takes two 4-bit inputs (A and B) and produces a 4-bit output Sum. Implement a 2-to-1 multiplexer (mux2to1) as a sub-module to select between two possible sum results based on the carry input.

13.Design a parameterized Carry-Select Adder module in Verilog that can handle N-bit inputs. The module should include a parameter N for the bit width and take two N-bit inputs (A and B). Implement the module to generate the carry-select adder for the given bit width.

14.Design a 16-bit Carry-Select Adder module in Verilog by instantiating two 8-bit Carry-Select Adders. Write a testbench to verify the functionality of the 16-bit adder.

15.Design a 8-bit carry select adder module in Verilog. The module should have two 8-bit inputs (A and B) and a 1-bit input Cin. It should output an 8-bit sum Sum and a 1-bit carry-out Cout. Use two 4-bit ripple carry adders and a multiplexer to implement the carry select adder.

#conditional sum and prefix adder

16.Design a 4-bit conditional sum adder module in Verilog that takes two 4-bit inputs (A and B) and produces a 4-bit output Sum. The module should also have a carry-in input Cin and a control input Ctrl. When Ctrl is set to 1, the adder should perform addition with the carry; otherwise, it should perform subtraction with the borrow.

17.Implement a 8-bit conditional sum adder in Verilog using conditional if-else statements. The adder should accept two 8-bit inputs (A and B), a carry-in input Cin, and a control input Ctrl. Depending on the value of Ctrl, the adder should either add or subtract the inputs along with the carry.

18.Design an 8-bit conditional sum adder module using the ternary operator (? :) in Verilog. The module should take two 8-bit inputs (A and B), a carry-in input Cin, and a control input Ctrl. Implement the adder logic using the ternary operator for conditional selection of addition or subtraction.

19.Create a 16-bit conditional sum adder using Verilog generate statements. The adder should support both addition and subtraction based on a control input Ctrl. Use generate statements to instantiate 4 instances of a 4-bit CSA module.

20.Write a testbench for a conditional sum adder module that generates random 8-bit inputs, random Cin and Ctrl values, and compares the output with the expected result. Use Verilog's $random function to generate random inputs and assertions to verify correctness.

#prefix adders

21.Design an 8-bit prefix adder module in Verilog using a Brent-Kung prefix structure. The module should take two 8-bit inputs (A and B) and produce an 8-bit output Sum.

22.Design a 16-bit Kogge-Stone prefix adder module in Verilog. The module should take two 16-bit inputs (A and B) and produce a 16-bit output Sum

23.Design a 16-bit Han-Carlson prefix adder module in Verilog. The module should take two 16-bit inputs (A and B) and produce a 16-bit output Sum.

.

#carry save multiplier

24. Design a 4-bit Carry Save Multiplier module in Verilog that takes two 4-bit inputs (A and B) and produces an 8-bit product P. Assume that the inputs are in two's complement representation. Write the module and test it with appropriate test cases.

25. Design an 8-bit parallel Carry Save Multiplier module in Verilog that takes two 8-bit inputs (A and B) and produces a 16-bit product P.Implement the multiplication using multiple 4-bit carry save multiplier modules. Write the module and test it with appropriate test cases.

26. Design of the 4-bit Carry Save Multiplier that includes an additional input Acc that accumulates the product. The module should accumulate the product of multiple multiplications. Write the module and test it with appropriate test cases.

#wallace multiplier

27. Design a Wallace Tree Multiplier module in Verilog that takes two 4-bit inputs (A and B) and produces an 8-bit output Product.

28. Implement the partial product generation step within the WallaceTreeMultiplier module. Given inputs A and B, generate the partial product for each bit of A

29. Complete the WallaceTreeMultiplier module by implementing the Wallace Tree structure using the partial products and compressors. Assume that you have 4-to-2 compressors available.

#booth's algorithm implementation

30. Design a Verilog module that implements Booth's algorithm for signed integer multiplication. The module should take two 4-bit signed numbers (A and B) as inputs and produce an 8-bit product P.

31. Design a Verilog module for a bit-pairing multiplier. The module should accept two 8-bit inputs (A and B) and generate a 16-bit product P. Use the bit-pairing technique to reduce the number of partial products.

.

#restoring and non restoring division

32. Design a Verilog module that implements the restoring division algorithm. The module should take two 8-bit unsigned numbers ('Dividend' and 'Divisor’) as inputs and produce an 8-bit quotient 'Quotient' and an 8-bit remainder 'Remainder'.

33. Create a Verilog module for the non-restoring division algorithm. The module should accept two 8-bit signed numbers (Dividend and Divisor) as inputs and generate an 8-bit quotient Quotient and an 8-bit remainder Remainder.

34. Implement a Verilog module for non-restoring division with rounding. The module should accept two 8-bit signed inputs (Dividend and Divisor) and generate a 8-bit quotient Quotient and a 8-bit remainder Remainder. Include logic to round the quotient to the nearest integer.

35. Design a Verilog module for radix-4 non-restoring division. Take two 16-bit unsigned inputs (Dividend and Divisor) and generate a 16-bit quotient Quotient and a 16-bit remainder Remainder.

#barrel shifter

36.Design a 4-bit right barrel shifter in Verilog. Create a module that takes a 4-bit input (A), a 2-bit shift amount (Shift), and a control signal (Direction) to specify left or right shift. Implement the barrel shifter using multiplexers.

37.Implement a variable-width barrel shifter in Verilog. Design a module that accepts a 6-bit input (A), a 3-bit shift amount (Shift), and a control signal (Direction). If Direction is 0, the shifter should perform a left shift; if it's 1, it should perform a right shift. Generate a 6-bit output (Result) accordingly.

38.Design a circular barrel shifter module in Verilog. This module should take a 5-bit input (A) and a 3-bit rotation amount (Rotate). Implement the rotation such that the input bits are shifted circularly by the specified number of positions.

39.Implement a cascaded barrel shifter in Verilog. Create a module that accepts a 16-bit input (A), two 3-bit shift amounts (Shift1 and Shift2), and a control signal (Direction). The module should perform two successive shifts: first by Shift1 positions and then by Shift2 positions, in the specified direction. Output the final 16-bit result.

#truncation and rounding off

40.Design a Verilog module that takes a 32-bit floating-point number in IEEE 754 format as input and truncates it to a 16-bit floating-point number. Implement the truncation process according to the IEEE 754 standard.

41.Implement a Verilog module that rounds off a 12-bit fixed-point number to the nearest integer. The module should include an input flag indicating whether to round up when the fractional part is exactly halfway between two integers.

42.Design a Verilog module that takes a 4-element vector of 16-bit signed integers as input and truncates each component to an 8-bit signed integer. Ensure that the truncation process retains the sign bit and discards the least significant bits.

43.Create a Verilog module that rounds a 32-bit binary-coded decimal (BCD) number to a specified number of decimal places. The module should support rounding up or down based on the fractional part of the BCD representation.

44.Implement a Verilog module that performs combined truncation and rounding of a 24-bit fixed-point number. The module should take a control input indicating whether to truncate or round the number. Provide test cases to demonstrate the different behaviours of the module.

#floating point arithematic

45.Design a Verilog module that performs single-precision floating-point addition according to the IEEE 754 standard. The module should take two 32-bit single-precision floating-point inputs (A and B) and produce a 32-bit single-precision floating-point result Result.

46.Implement a Verilog module for double-precision floating-point multiplication. The module should take two 64-bit double-precision floating-point inputs (A and B) and generate a 64-bit double-precision floating-point product Product. Use the IEEE 754 double-precision format.

47. Design a Verilog module that converts a 32-bit single-precision floating-point number to its equivalent 16-bit binary representation. The module should take the single-precision input F and produce the 16-bit binary output Binary.

48. Design a Verilog module for single-precision floating-point division according to the IEEE 754 standard. The module should take two 32-bit single-precision floating-point inputs (Dividend and Divisor) and produce a 32-bit single-precision floating-point result Quotient.

49. Implement a Verilog module that compares two 32-bit single-precision floating-point numbers (A and B) and produces a 1-bit output indicating whether A is greater than or equal to B. Handle special cases such as NaN values.

50. Design a Verilog module that converts a 32-bit single-precision floating-point number (FloatValue) to a 16-bit signed fixed-point representation. The module should take the input and produce the fixed-point output Fixed Value