Experiment no 1: Design and Implementation of a Universal Gates

AND Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity AND_Gate is
port(
   A: in std_logic;
   B: in std_logic;
   Y: out std_logic
);
end AND_Gate;

architecture AND_Logic of AND_Gate is
begin
   Y <= A AND B;
end AND_Logic;
```

OR Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity OR_Gate is
port(
   A: in std_logic;
   B: in std_logic;
   Y: out std_logic
);
end OR_Gate;

architecture OR_Logic of OR_Gate is
```

```
begin
   Y <= A OR B;
end OR_Logic;
```

NOT Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity NOT_Gate is
port(
   A: in std_logic;
   Y: out std_logic
);
end NOT_Gate;

architecture NOT_Logic of NOT_Gate is
begin
   Y <= NOT A;
end NOT_Logic;
```

NAND Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity NAND_Gate is
port(
   A: in std_logic;
   B: in std_logic;
   Y: out std_logic
);
end NAND_Gate;
```

```
architecture NAND_Logic of NAND_Gate is
begin
   Y <= NOT (A AND B);
end NAND_Logic;
```

NOR Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity NOR_Gate is
port(
   A: in std_logic;
   B: in std_logic;
   Y: out std_logic
);
end NOR_Gate;

architecture NOR_Logic of NOR_Gate is
begin
   Y <= NOT (A OR B);
end NOR_Logic;
```

XOR Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity XOR_Gate is
port(
   A: in std_logic;
   B: in std_logic;
```

```
    Y: out std_logic
);
end XOR_Gate;

architecture XOR_Logic of XOR_Gate is
begin
    Y <= A XOR B;
end XOR_Logic;
```

Experiment no 2: Design and Implementation of Full Adder

A full adder is a digital circuit that performs the addition of two binary numbers and an input carry. It serves as the fundamental building block for more complex arithmetic operations like adders, subtractors, and multipliers.

Design of a Full Adder:

A full adder can be designed using two half adders and an OR gate. Half adders perform the addition of two single-bit binary numbers, while the OR gate resolves any carry generated from the half adders.

Truth Table:

The truth table for a full adder is as follows:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Implementation using Hardware Gates:

The full adder can be implemented using logic gates as follows:

```
Sum = (A XOR B) XOR Cin
Cout = (A AND B) OR (A XOR B) AND Cin
```

Implementation using HDL (Hardware Description Language):

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Full_Adder is
port(
   A: in std_logic;
   B: in std_logic;
   Cin: in std_logic;
   Sum: out std_logic;
   Cout: out std_logic
);
end Full_Adder;

architecture FA_Logic of Full_Adder is
begin
   Sum <= (A XOR B) XOR Cin;
   Cout <= (A AND B) OR (A XOR B) AND Cin;
end FA_Logic;
```

Experiment no 3: Design and Implementation of Decoder.

The design of a decoder depends on the number of input bits (n) and the number of output lines (2^n). For instance, a 2-to-4 decoder has two input bits and four output lines, while a 3-to-8 decoder has three input bits and eight output lines.

Truth Table:

The truth table for a decoder specifies the output signals for each combination of input bits. For example, the truth table for a 2-to-4 decoder is as follows:

A	B	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

A decoder is a digital circuit that converts a binary code into a set of output signals, where each output signal corresponds to a specific combination of input bits. They are essential components in digital systems for selecting, enabling, or controlling various circuits based on the input code.

Design of a Decoder:

The design of a decoder depends on the number of input bits (n) and the number of output lines (2^n). For instance, a 2-to-4 decoder has two input bits and four output lines, while a 3-to-8 decoder has three input bits and eight output lines.

Truth Table:

The truth table for a decoder specifies the output signals for each combination of input bits. For example, the truth table for a 2-to-4 decoder is as follows:

A	B	D0	D1	D2	D3
0	0	1	0	0	0

0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

drive_spreadsheetExport to Sheets

Implementation using Hardware Gates:

Decoders can be implemented using logic gates, such as AND, OR, and NOT gates. The specific gate configuration depends on the number of input bits and output lines.

Implementation using HDL (Hardware Description Language):

Decoders can also be implemented using a hardware description language like VHDL or Verilog. Here's an example of VHDL code for a 2-to-4 decoder:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Two_to_Four_Decoder is
port(
   A: in std_logic;
   B: in std_logic;
   D0: out std_logic;
   D1: out std_logic;
   D2: out std_logic;
   D3: out std_logic
);
end Two_to_Four_Decoder;

architecture Decoder_Logic of Two_to_Four_Decoder is
begin
   D0 <= NOT(A) AND NOT(B);
```

```
    D1 <= NOT(A) AND B;
    D2 <= A AND NOT(B);
    D3 <= A AND B;
end Decoder_Logic;
```

Experiment no 4: Design and Implementation of J-K Latch

Design:

The design of a J-K latch involves a combination of logic gates, typically NAND gates, arranged in a feedback loop. The feedback loop allows the latch to maintain its state even when the clock signal is low. The J and K inputs control the behavior of the latch at the rising edge of the clock signal. When J is high and K is low, the latch sets its output to 1. When J is low and K is high, the latch resets its output to 0. When both J and K are low, the latch holds its current state. When both J and K are high, the latch toggles its output, meaning it changes from 0 to 1 or from 1 to 0.

Implementation:

The J-K latch can be implemented using various logic gates, but NAND gates are commonly used due to their universal property. The following schematic shows a J-K latch implementation using NAND gates:

here is the VHDL code for a J-K latch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity JKLatch is
port(
   J: in std_logic;
   K: in std_logic;
   Clk: in std_logic;
   Q: out std_logic;
);
end JKLatch;


architecture JK_Latch of JKLatch is
begin
   process(Clk)
   begin
      if rising_edge(Clk) then
         if J = '1' and K = '0' then
```

```
            Q <= '1';
        elsif J = '0' and K = '1' then
            Q <= '0';
        elsif J = '1' and K = '1' then
            Q <= NOT Q;
        else
            Q <= Q;
        end if;
    end if;
  end process;
end JK_Latch;
```

Experiment no 5: Design and Implementation of D-Latch.

Design of a D-Latch:

A D-latch, also known as a data latch, is a simple and fundamental sequential logic circuit that stores data. It is an asynchronous edge-triggered flip-flop, meaning that the data held by the latch is updated only at the rising or falling edge of the clock signal. Unlike a J-K latch, a D-latch has only one data input, D, which determines the next state of the output, Q, based on the presence or absence of the clock signal.

Implementation using Hardware Gates:

D-latches can be implemented using various logic gates, but NAND gates are commonly used due to their universal property. The following schematic shows a D-latch implementation using NAND gates:

The truth table for a D-latch is as follows:

D	Clock	Q
0	0	Q
0	1	0
1	0	Q
1	1	1

Here is the VHDL code for a D-latch:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DLatch is
port(
   D: in std_logic;
   Clk: in std_logic;
   Q: out std_logic;
);
```

```
end DLatch;


architecture DLatch of DLatch is
begin
   process(Clk)
   begin
      if rising_edge(Clk) then
         Q <= D;
      end if;
   end process;
end DLatch;
```

Experiment no 6: Design and Implementation asynchronous counter.

Asynchronous counters, also known as ripple counters, are sequential digital circuits that increment or decrement a count based on a clock signal. They employ a cascading structure of flip-flops, where the output of one flip-flop triggers the next, creating a chain of propagation delays. Unlike synchronous counters, asynchronous counters do not rely on a global clock signal for synchronization, making them simpler and less susceptible to clock skew issues.

Design Principles:

1. Flip-Flop Chain: Asynchronous counters consist of a chain of flip-flops, typically D flip-flops, connected in a cascade manner. The output of each flip-flop feeds the input of the next flip-flop, creating a ripple effect.

2. Flip-Flop Control: The control logic for each flip-flop determines the behavior of the counter, whether it is incrementing or decrementing. The control logic is typically implemented using combinational logic gates based on the desired count sequence.

3. Clocking: Asynchronous counters are triggered by the output of the previous flip-flop, creating a self-clocking mechanism. The propagation delay of each flip-flop determines the minimum clock period for the counter to operate reliably.

Implementation:

1. Up-Counter: An up-counter increments the count with each clock pulse. The control logic for each flip-flop is designed to toggle its state when the previous flip-flop's output is high, creating a ripple-up count sequence.

2. Down-Counter: A down-counter decrements the count with each clock pulse. The control logic for each flip-flop is designed to complement its state when the previous flip-flop's output is high, creating a ripple-down count sequence.

3. Multi-Bit Counters: Multi-bit asynchronous counters can be constructed by cascading multiple flip-flops, each representing a different bit of the count. The control logic for each flip-flop is designed to handle the carry or borrow propagation between bits.

Advantages:

1. Simplicity: Asynchronous counters are relatively simple in design, requiring fewer components compared to synchronous counters.

2. Clockless Operation: The self-clocking mechanism eliminates the need for a global clock signal, reducing complexity and clock skew issues.

3. Flexibility: Asynchronous counters can be easily customized to handle various count sequences and bit widths.

Disadvantages:

1. Speed Limitations: The propagation delay through the flip-flop chain limits the maximum operating frequency of asynchronous counters.
2. Glitches and Hazards: The asynchronous nature of the counter can lead to glitches and hazards in the output signal, requiring additional logic to eliminate them.
3. Synchronization Challenges: Combining asynchronous counters with synchronous systems can require synchronization circuitry to avoid metastability issues.

Applications:

1. Frequency Dividers: Asynchronous counters can be used to divide the frequency of a clock signal, generating lower-frequency clock pulses.
2. Binary Counters: Asynchronous counters are commonly used as binary counters for counting events or generating address sequences.
3. Timing Control: Asynchronous counters can be used to generate timing signals and control the sequencing of operations in digital systems.
4. Pulse Generation: Asynchronous counters can be used to generate pulses of varying widths, useful for timing control and synchronization.
5. Code Conversion: Asynchronous counters can be used to convert binary codes into other formats, such as BCD (Binary-Coded Decimal) or Gray codes.

Experiment no 7: Design and verify the circuit CMOS Inverter using transient analysis. Obtain VTC curve and threshold voltage of inverter for a specific parameter, verify with the value of threshold voltage obtained using formula. Create symbol of this inverter for further application.

Experiment no 8: Create a Layout of CMOS Inverter.

Experiment no 9: Design NAND and NOR gate. Perform all the analysis.

Experiment no 10: Design 1-bit half adder and verify the circuit using transient analysis.

Experiment no 11: Design Full adder and verify the circuit using transient analysis.

Experiment no 12: Design a multiplexer technology and perform all the analysis to verify its characteristics.

Experiment no 13: Design 1-bit half adder and verify the circuit using transient analysis:

Experiment no 14: Design a MOS based SRAM cell and verify its characteristics and create a layout of the same