VIDYA PRATISHTHAN'S

KAMALNAYAN BAJAJ INSTITUTE OF ENGINEERING AND TECHNOLOGY, BARAMATI

Vidyanagari, Bhigwan Road, Baramati. Dist. Pune -413133

Academic Year: 2024-25

LABORATORY MANUAL

Name of the Student:		
Class:	**Division:**	**Roll No.:**
Subject: VLSI Design & Technology (2019 Course) [404182]		**Exam Seat No.:**
Department of Electronics and Telecommunication Engineering		

VIDYA PRATISHTHAN'S
KAMALNAYAN BAJAJ INSTITUTE OF ENGINEERING AND TECHNOLOGY, BARAMATI

INDEX

Department: Electronics and Telecommunication Engineering **Class:** BE

Sr. No.	Name of the Experiment	Date of Conduction	Date of Checking	Page No.	Sign	Remark
Part A: To write VHDL code, simulate with test bench, synthesis, implement on PLD						
1	4 bit ALU for Add, Subtract, AND, NAND, OR, XOR & XNOR.					
2	Universal shift register with mode selection input for SISO, SIPO, PISO, & PIPO.					
3	Mod - N Counter					
4	FIFO Memory					
5	Keypad interface					
Part B: To prepare CMOS layout in selected technology, simulate with & without capacitive load, comment on rise & fall times.						
6	Inverter, NAND, NOR gates					
7	Half Adder & Full Adder					
8	2:1 Mux using logic gates & transmission gates					
9	One bit SRAM Cell					

CERTIFICATE

This is to certify that Mr./Miss. _____of Class B.E.E&TC Roll No._____has satisfactory completed term work of the subject "VLSI Design & Technology" for ____ semester of academic year _____.

Date: / /20 Staff Member In-Charge HoD Principal

Experiment No: 1

To design 4 bit ALU for add, subtract, AND, NAND, XOR, XNOR, OR

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: To design 4 bit ALU for add, subtract, AND, NAND, XOR, XNOR, OR operations.

OBJECTIVE:

1. Write, synthesize and simulate VHDL code of ALU for add, subtract, AND, NAND, XOR, XNOR, OR& ALU Pass operations.
2. Implement on FPGA.

KEYWORDS: FPGA, ALU

PRE-LAB:

1. Study various VHDL modeling techniques
2. Study hardware level structure of ALU
3. Study various Arithmetic and Logical operations.

EXPLANATION:

In ECL, TTL and CMOS, there are available integrated packages which are referred to as arithmetic logic units (ALU). The logic circuitry in this unit is entirely combinational (i.e. consists of gates with no feedback and no flip-flops).The ALU is an extremely versatile and useful device since, it makes available, in single package, facility for performing many different logical and arithmetic operations.Arithmetic Logic Unit (ALU) is a critical component of a microprocessor and is the core component of central processing unit.ALU's comprise the combinational logic that implements logic operations such as AND, OR and arithmetic operations, such as ADD, SUBTRACT.

Functionally, the operation of typical ALU is represented as shown in diagram below,

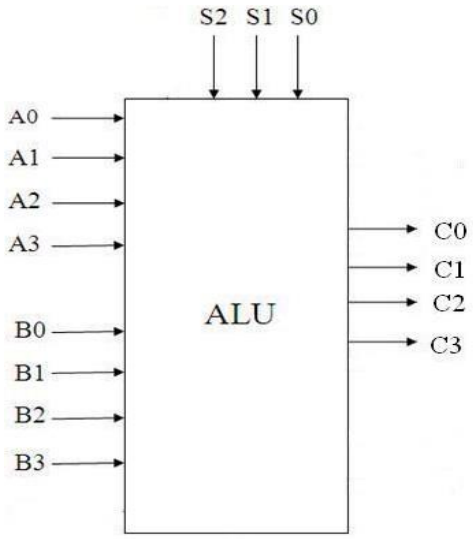


Figure 1: Functional representation of Arithmetic Logic Unit

Controlled by the three function select inputs (S0 to S2), ALU can perform all the 8 possible logic operations or 8 different arithmetic operations on active HIGH or active LOW operands.The function table lists the arithmetic operations that are performed in ALU.

Table 1: Function Table for ALU

Mode Select Inputs			Function
0	0	0	A + B
0	0	1	A - B
0	1	0	A **AND** B
0	1	1	A **NAND** B
1	0	0	A **XOR** B
1	0	1	A **XNOR** B
1	1	0	A **OR** B

Examples for arithmetic operations in ALU:

The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations. 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10. The first three operations produce a sum of one digit, but when the both augends and addend bits are equal 1, the binary sum consists of two digits. The higher significant bit of the result is called carry.When the augends and addend number contains more significant digits, the carry obtained from the addition of the two bits is called half adder. One that performs the addition of three bits (two significant bits and a previous carry) is called full adder. The name of the circuit is from the fact that two half adders can be employed to implement a full adder.

A binary adder-subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers. Connecting n full adders in cascade produces a binary adder for two n-bit numbers.

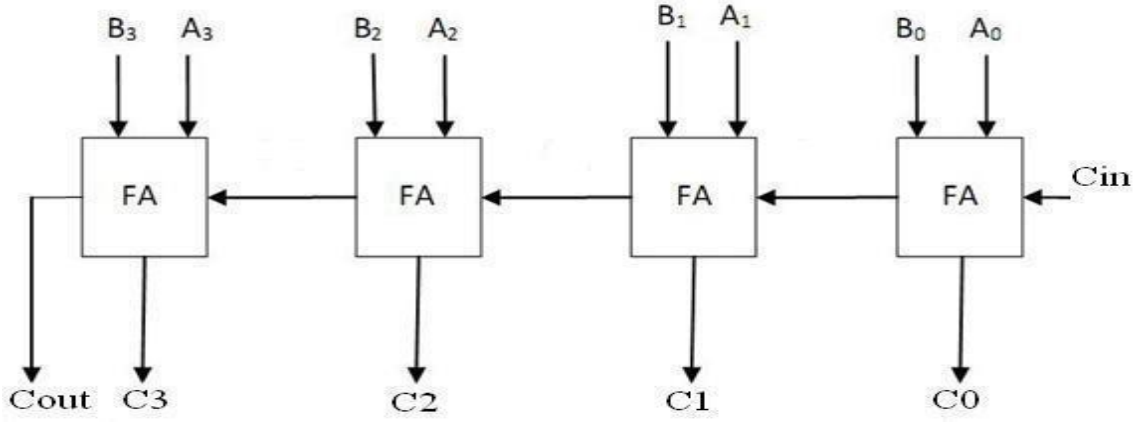


Figure 2 :4-Bit Adder

The subtraction circuit is included by providing a complementing circuit.The subtraction of unsigned binary numbers can be done most conveniently by means of complement. Subtraction A–B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be

obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with the inverters and a one can be added to the sum through the input carry.The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an EX-OR gate with each full adder.A 4-bit adder-subtractor circuit is shown in fig 3. The mode input M controls the operation. When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor.

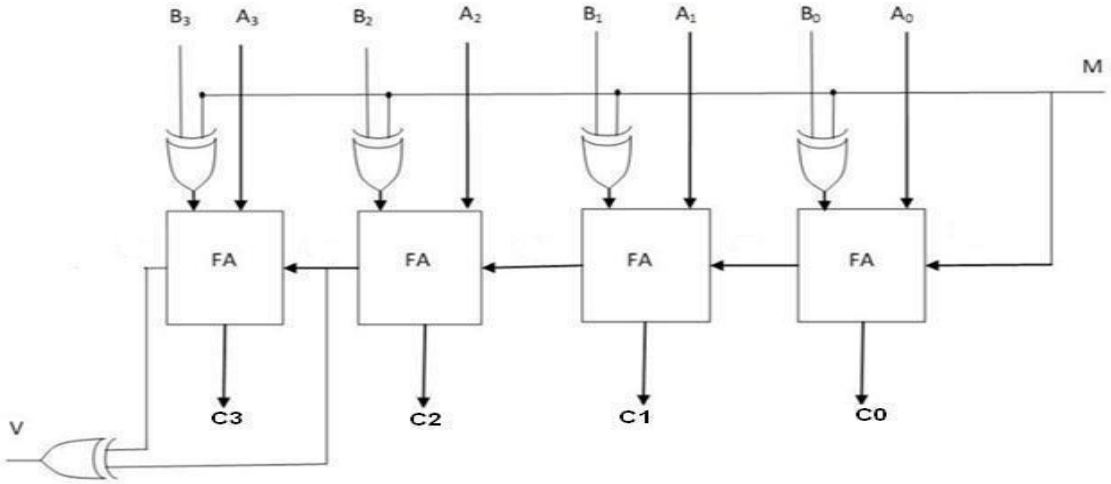


Figure 3 :4-Bit Adder Subtractor

Examples for Logical operations in ALU:

In a 4-bit Arithmetic Logic Unit, logical operations are performed on individual bits.

EX-OR - In a 4 bit ALU, the inputs given are A0, A1, A2, A3 and B0, B1, B2, B3. Operations are performed on individual bits. Thus, as shown in fig.4, inputs A0 and B0 will give output C0.

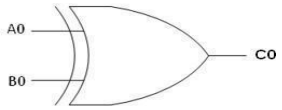


Figure 4 : Ex-OR Gate

Table 2 : Truth table for Ex-Or gate

Inputs		Outputs
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

APPARATUS: Basys 3 FPGA development board.

EXPERIMENT PROCEDURE:

1. Define output directory location.
2. Setup design sources and constraints.
3. Run synthesis, report utilization and timing estimates, write checkpoint design.
4. Run placement and logic optimization, report utilization and timing estimates, write checkpoint design.

5. Run router, report actual utilization and timing, write checkpoint design, run drc, write verilog and xdc out.
6. Generate a bitstream.
7. Refer “Create your first Project in Vivado” document available on Classroom for more details

POST-LAB:

1. Design two ALUs which operate parallel.
2. Design ALU using structural modeling technique.

APPLICATION:

_____

_____

_____

_____

OBSERVATION TABLE:

Logic Utilization	Used	Available	% Utilization
Number of Slices			
Number of Slice Flip Flops			
Number of 4 input LUTs			
Number of bonded IOBs			
Number of BRAMs			
Number of GCLKs			

RESULTS:

Print outs showing RTL schematic of the design and implementation on FPGA are attached at the end of experiment**.**

CONCLUSION:

_____

_____

_____

VIVA QUESTIONS:

1. Which modeling technique is best suitable for ALU design? Why?
2. What is the application of ALU?
3. How to implement ALU design on an FPGA board?
4. Is ALU sequential or combinational hardware?
5. What is the difference between signal and variable in VHDL?

Experiment No: 2

To design Universal shift register with mode selection input for SISO, SIPO, PISO& PIPO modes

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: To design Universal shift register with mode selection input for SISO, SIPO, PISO& PIPO modes.

OBJECTIVE:

1. Write, synthesize and simulate VHDL code of Universal shift register with mode selection input for SISO, SIPO, PISO & PIPO modes.
2. Implement on FPGA.

KEYWORDS:SISO, SIPO, PISO & PIPO.

PRELAB:

1. Study different operating modes of Shift register
2. Study hardware of Universal shift register

APPARATUS: Basys 3 FPGA development board.

THEORY:

Shift registers, like counters, are a form of *sequential logic*. Sequential logic, unlike combinational logic, is not only affected by the present inputs, but also, by the prior history. In other words, sequential logic remembers past events. Shift registers produce a discrete delay of a digital signal or waveform. A waveform synchronized to a *clock*, a repeating square wave, is delayed by **"n"** discrete clock times, where**"n"** is the number of shift register stages. Thus, a four stage shift register delays "data in" by four clocks to "data out". The stages in a shift register are *delay stages*, typically type **"D"** Flip-Flops or type **"JK"** Flip-flops.Basic shift registers are classified by structure according to the following types:

- Serial-in/serial-out
- Parallel-in/serial-out
- Serial-in/parallel-out
- parallel-in/parallel-out

Universal Shift Register: A shift register that can perform with any combination of serial and parallel inputs and outputs (i.e. Serial-in/serial-out, Parallel-in/serial-out, Serial-in/parallel-out, parallel-in/parallel-out). A universal shift register is often a bi-directional as well.

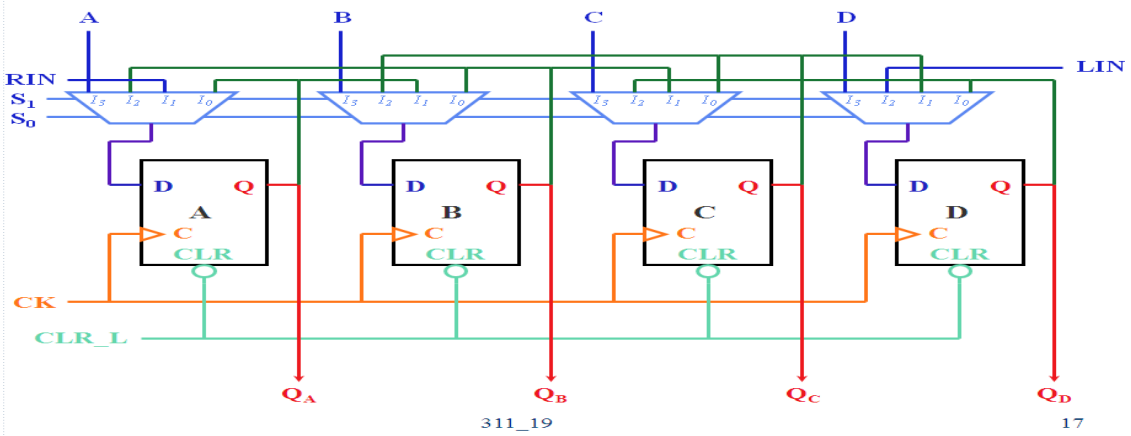


Figure 1: Universal shift register

Serial-in/serial-out shift register: Ithas a clock input, a data input, and a data output from the last stage. In general, the other stage outputs are not available otherwise, it would be a serial-in, parallel-out shift register. Three type **D** Flip-Flops are cascaded Q to D and the clocks paralleled to form a three stage shift register as shown in following figure.

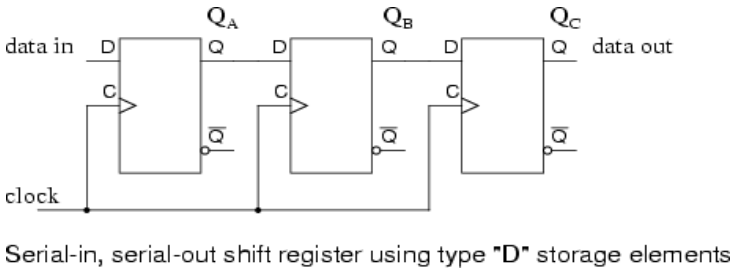


Figure 2: In-Out Shift Register using type “D” Storage Elements

The waveforms below are applicable to serial-in, serial-out shift register. The three pairs of arrows show that a three stage shift register temporarily stores 3-bits of data and delays it by three clock periods from input to output.

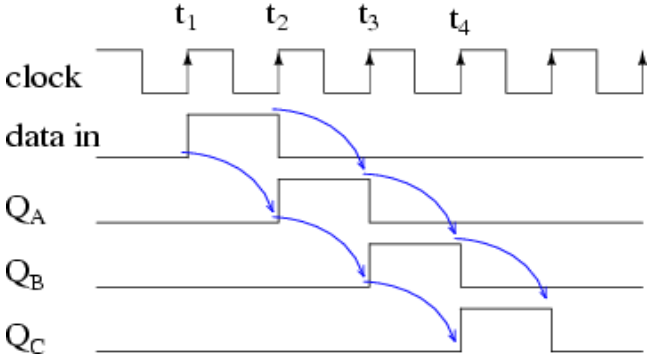
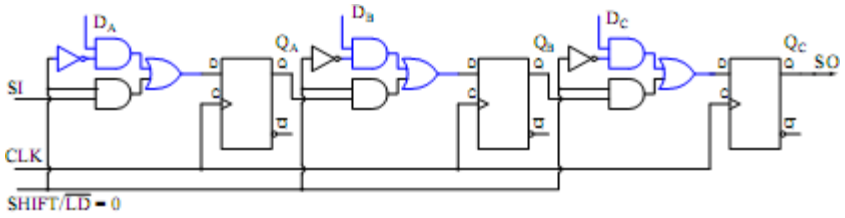


Figure 3: Waveforms

Parallel-in/ serial-out shift register: It does everything that the previous serial-in/ serial-out shift registers does plus input data to all stages simultaneously. The parallel-in/ serial-out shift register

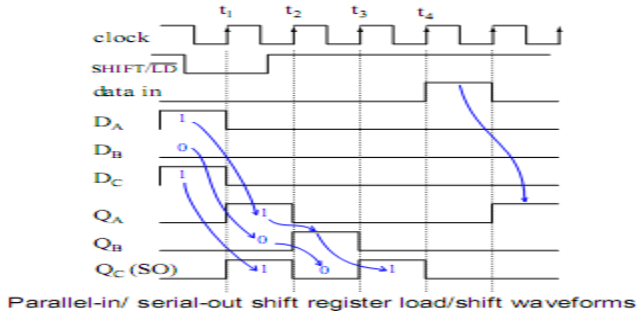
stores data, shifts it on a clock by clock basis, and delays it by the number of stages times the clock period. In addition, parallel-in/ serial-out really means that we can load data in parallel into all stages before any shifting ever begins



Parallel-in/ serial-out shift register showing parallel load path

Figure 4: Parallel-in, Serial-out Shift Register showing Parallel Load Pattern

In the above figure, we show the parallel load path when SHIFT/LD' is logic low. The upper NAND gates serving DA DB DC are enabled, passing data to the D inputs of type D Flip-Flops respectively. At the next positive going clock edge, the data will be clocked from D to Q of the three FFs. The shift path is shown above when SHIFT/LD' is logic high. The lower AND gates of the pairs feeding the OR gate are enabled giving us a shift register connection of SI to DA , QA to DB , QB to DC , QC to SO. Clock pulses will cause data to be right shifted out to SO on successive pulses. The waveforms below show both parallel loading of three bits of data and serial shifting of this data. Parallel data at DA DB DC is converted to serial data at SO.

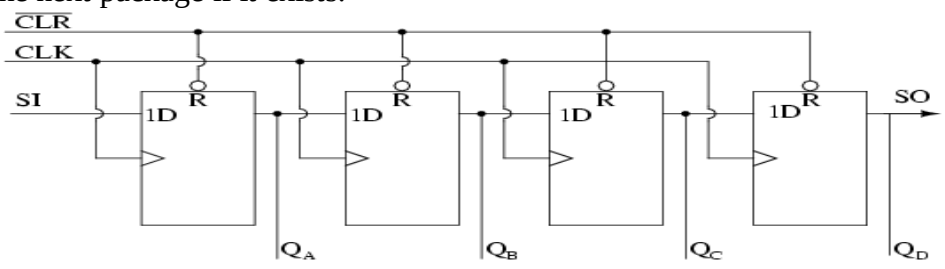


Parallel-in/ serial-out shift register load/shift waveforms

Figure 5: Parallel-in, Serial-out Shift Register Load / Shift Waveforms

Serial-in/parallel-out shift register:

The details of the serial-in/parallel-out shift register are fairly simple. It looks like a serial-in/ serial-out shift register with taps added to each stage output. Serial data shifts in at **SI** (Serial Input). After a number of clocks equal to the number of stages, the first data bit in appears at SO (Q_D) in the figure. In general, there is no SO pin. The last stage (Q_D above) serves as SO and is cascaded to the next package if it exists.



Serial-in/ Parallel out shift register details

Figure 6: Serial-in, Parallel-out Shift Register Details

The shift register has been cleared prior to any data by CLR', an active low signal, which clears all type D Flip-Flops within the shift register. Following figure shows waveforms for this mode. Note that serial data 1011 pattern is presented at the SI input. This data is synchronized with the clock CLK.

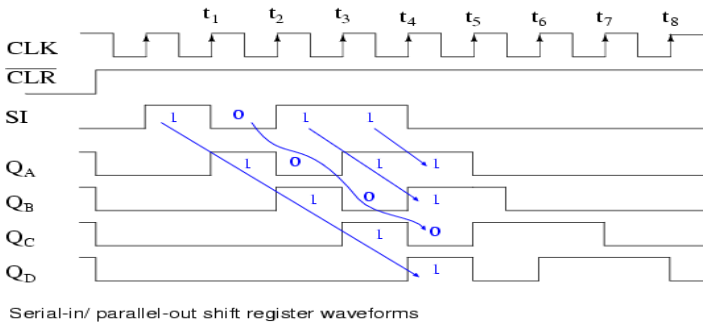


Figure 7: Serial-in, Parallel-out Shift Register Waveform

Parallel-in/ parallel-out shift register: The internal details of a right shifting parallel-in/ parallel-out shift register are shown below. The tri-state buffers are not strictly necessary to the parallel-in/ parallel-out shift register, but are part of the real-world device shown below

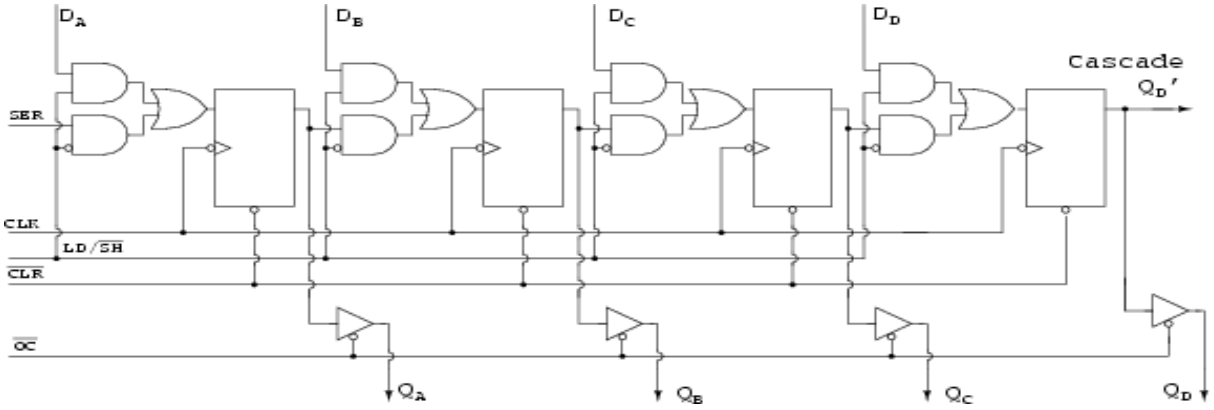


Figure 8: Parallel-in, Parallel-out Shift Register with tri-state Output

If **LD/SH'=1**, the upper four AND gates are enabled allowing application of parallel inputs **D_A D_B D_C D_D** to the four FF data inputs. The four bits of data will be clocked in parallel from **D_A D_B D_C D_D** to **Q_A Q_B Q_C Q_D**at the next negative going clock. In this "real part", **OC'** must be low if the data needs to be available at the actual output pins as opposed to only on the internal FFs.

EXPERIMENT PROCEDURE:

1. Define output directory location.
2. Setup design sources and constraints.
3. Run synthesis, report utilization and timing estimates, write checkpoint design.
4. Run placement and logic optimization, report utilization and timing estimates, write checkpoint design.

5. Run router, report actual utilization and timing, write checkpoint design, run drc, write verilog and xdc out.
6. Generate a bitstream.
7. Refer “Create your first Project in Vivado” document available on Classroom for more details

POST-LAB:

1. Design barrel shifter
2. Design shift register using JK Flip flop

APPLICATION:

_____

_____

_____

_____

_____

OBSERVATION TABLE:

Logic Utilization	Used	Available	% Utilization
Number of Slices			
Number of Slice Flip Flops			
Number of 4 input LUTs			
Number of bonded IOBs			
Number of BRAMs			
Number of GCLKs			

RESULTS:

Print outs showing RTL schematic and implementation on FPGA are attached at the end of experiment**.**

CONCLUSION:

_____

_____

_____

_____

_____

Viva Questions:

1. What is the need of shifting operation?
2. What is the application of SIPO and PISO mode?
3. What is the difference between memory and shift register?
4. What is the difference between shift register and barrel shifter?
5. Why is the clock signal needed in the shift register?

Experiment No: 3

To design MOD N Counter

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: To design Universal shift register with mode selection input for SISO, SIPO, PISO& PIPO modes.

OBJECTIVE:

1. Write, synthesize and simulate VHDL code of Universal shift register with mode selection input for SISO, SIPO, PISO & PIPO modes.
2. Implement on FPGA.

KEYWORDS:SISO, SIPO, PISO & PIPO.

PRELAB:

1. Study different operating modes of Shift register
2. Study hardware of Universal shift register

APPARATUS: Basys 3 FPGA development board.

THEORY:

The job of a counter is to count by advancing the contents of the counter by one count with each clock pulse. Counters which advance their sequence of numbers or states when activated by a clock input are said to operate in a "count-up" mode. Likewise, counters which decrease their sequence of numbers or states when activated by a clock input are said to operate in a "count-down" mode. Counters that operate in both the UP and DOWN modes, are called bidirectional counters.

Counters are sequential logic devices that are activated or triggered by an external timing pulse or clock signal. A counter can be constructed to operate as a synchronous circuit or as an asynchronous circuit. With synchronous counters, all the data bits change synchronously with the application of a clock signal. Whereas an asynchronous counter circuit is independent of the input clock so the data bits change state at different times one after the other.

Then counters are sequential logic devices that follow a predetermined sequence of counting states which are triggered by an external clock (CLK) signal. The number of states or counting sequences through which a particular counter advances before returning once again back to its original first state is called the **modulus** (MOD). In other words, the modulus (or just modulo) is the number of states the counter counts and is the dividing number of the counter.

Modulus Counters, or simply *MOD counters*, are defined based on the number of states that the counter will sequence through before returning back to its original value. For example, a 2-bit counter that counts from 00_2 to 11_2 in binary, that is 0 to 3 in decimal, has a modulus value of 4 (00

→ 1 → 10 → 11, and return back to 00) so would therefore be called a modulo-4, or mod-4, counter. Note also that it has taken four clock pulses to get from 00 to 11.

As in this simple example there are only two bits, (n = 2) then the maximum number of possible output states (maximum modulus) for the counter is: $2^n = 2^2$ or 4. However, counters can be designed to count to any number of 2^n states in their sequence by cascading together multiple counting stages to produce a single modulus or MOD-N counter.

Therefore, a "Mod-N" counter will require "N" number of flip-flops connected together to count a single data bit while providing 2^n different output states, (n is the number of bits). Note that N is always a whole integer value.

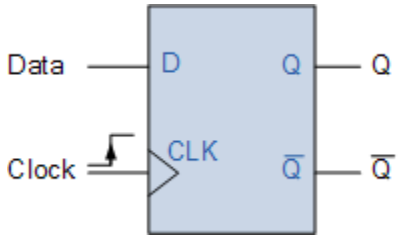
The we can see that MOD counters have a modulus value that is an integral power of 2, that is, 2, 4, 8, 16 and so on to produce an n-bit counter depending on the number of flip-flops used, and how they are connected, determining the type and modulus of the counter.

D-type Flip-flops

MOD counters are made using "flip-flops" and a single flip-flop can produce a count of 0 or 1, giving a maximum count of 2. There are different types of flip-flop designs we could use, the S-R, the J-K, J-K Master-slave, the D-type or even the T-type flip-flop to construct a counter. But to keep things simple, we will use the D-type flip-flop, (DFF) also known as a Data Latch, because a single data input and external clock signal are used, and is also positive edge triggered.

The D-type flip-flop, such as the TTL 74LS74, can be made from either S-R or J-K based edge-triggered flip-flops depending on whether you want it to change state either on the positive or leading edge (0 to 1 transition) or on the negative or trailing edge (1 to 0 transition) of the clock pulse. Here we will assume a positive, leading-edge triggered flip-flop. You can find more information in the following link about **D-type flip-flops**.

D-type Flip-flop and Truth Table



CLK	D	Q	$\overline{Q}$
↑	H	H	L
↑	L	L	H
L	X	no change	

The operation of a D-type flip-flop, (DFF) is very simple as it only has a single data input, called "D", and an additional clock "CLK" input. This allows a single data bit (0 or 1) to be stored under the control of the clock signal thus making the D-type flip-flop a synchronous device because the data on the inputs is transferred to the flip-flops output only on the triggering edge of the clock pulse.

So from the truth table, if there is a logic "1" (HIGH) on the Data input when a positive clock pulse is applied, the flip-flop SET's and stores a logic "1" at "Q", and a complimentary "0" at Q. Likewise, if there is a LOW on the Data input when another positive clock pulse is applied, the flip-flop RESET's and stores a "0" at "Q", and a resulting "1" at Q.

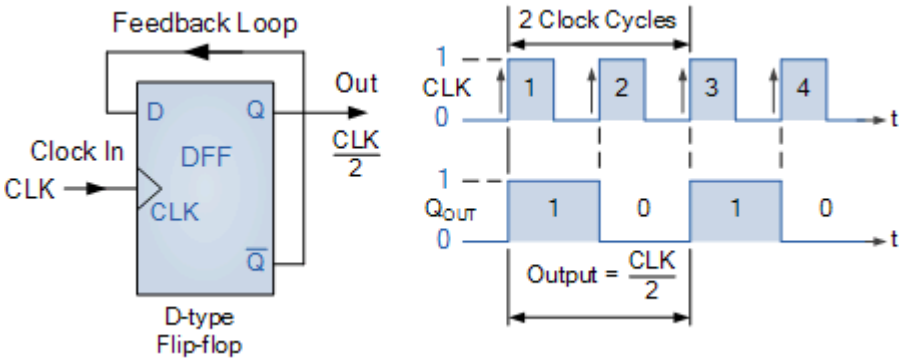
Then the output "Q" of the D-type flip-flop responds to the value of the input "D" when the clock (CLK) input is HIGH. When the clock input is LOW, the condition at "Q", either "1" or "0" is held until the next time the clock signal goes HIGH to logic level "1". Therefore the output at "Q" only changes state when the clock input changes from a "0" (LOW) value to a "1" (HIGH) making it a positive edge triggered D-type flip-flop. Note that negative edge-triggered flip-flops work in exactly the same way except that the falling edge of the clock pulse is the triggering edge.

So now we know how an edge-triggered D-type flip-flop works, lets look at connecting some together to form a MOD counter.

Divide-by-Two Counter

The edge-triggered D-type flip-flop is a useful and versatile building block to construct a MOD counter or any other type of sequential logic circuit. By connecting the Q output back to the "D" input as shown, and creating a feedback loop, we can convert it into a binary divide-by-two counter using the clock input only as the Q output signal is always the inverse of the Q output signal.

Divide-by-two Counter and Timing Diagram



The timing diagrams show that the "Q" output waveform has a frequency exactly one-half that of the clock input, thus the flip-flop acts as a frequency divider. If we added another D-type flip-flop so that the output at "Q" was the input to the second DFF, then the output signal from this second DFF would be one-quarter of the clock input frequency, and so on. So for an "n" number of flip-flops, the output frequency is divided by 2n, in steps of 2.

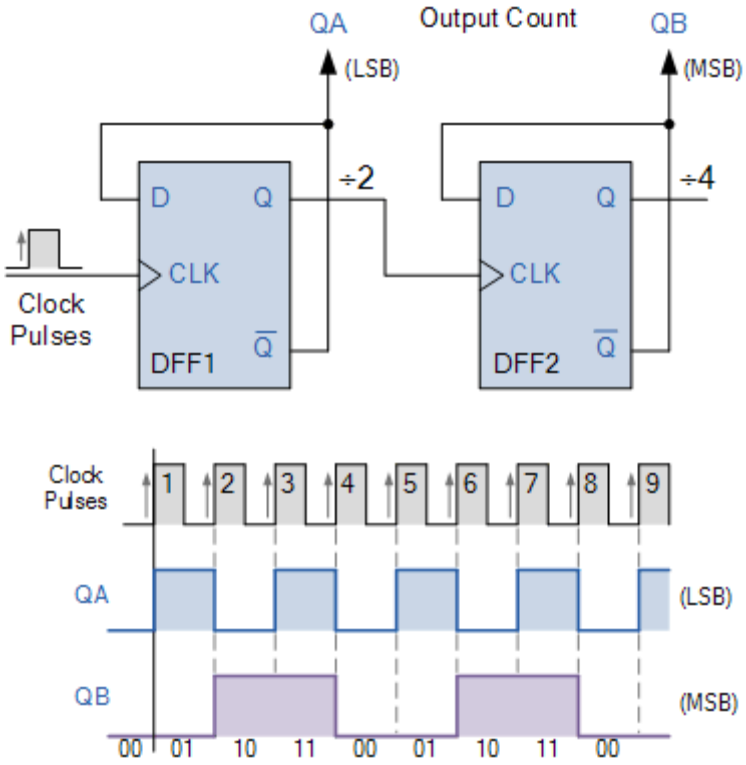
Note that this method of **frequency division** is very handy for use in sequential counting circuits. For example, a 60Hz mains frequency signal could be reduced to a 1Hz timing signal by using a divide-by-60 counter. A divide-by-6 counter would divide the 60Hz down to 10Hz which is then feed to a divide-by-10 counter to divide the 10Hz down to a 1Hz timing signal or pulse, etc.

MOD-4 Counter

Technically as well as being a 1-bit storage device, a single flip-flop on its own could be thought of as a MOD-2 counter, as it has a single output resulting in a count of two, either a 0 or 1, on the application of the clock signal. But a single flip-flop on its own produces a limited counting sequence, so by connecting together more flip-flops to form a chain, we can increase the counting capacity and construct a MOD counter of any value.

If a single flip-flop can be considered as a modulo-2 or MOD-2 counter, then adding a second flip-flop would give us a MOD-4 counter allowing it to count in four discrete steps. The overall effect would be to divide the original clock input signal by four. Then the binary sequence for this 2-bit MOD-4 counter would be: 00, 01, 10, and 11 as shown.

MOD-4 Counter and Timing Diagram



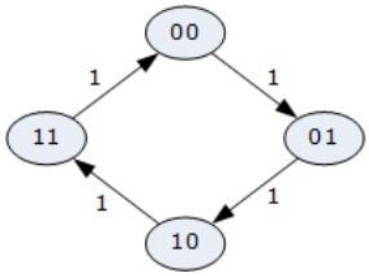
Note that for simplicity, the switching transitions of QA, QB and CLK in the above timing diagram are shown to be simultaneous even though this connection represents an asynchronous counter. In reality there would be a very small switching delay between the application of the positive going clock (CLK) signal, and the outputs at QA and QB.

We can show visually the operation of this 2-bit asynchronous counter using a truth table and state diagram.

MOD-4 Counter State Diagram

We can see from the truth table of the counter, and by reading the values of QA and QB, when QA = 0 and QB = 0, the count is 00. After the application of the clock pulse, the values become QA = 1, QB = 0, giving a count of 01. After the arrival of the next clock pulse, the values change and become QA = 0, QB = 1, giving a count of 10. Finally the values become QA = 1, QB = 1, giving a count of 11. The application of the next clock pulse causes the count to return back to 00, and thereafter it counts continuously up in a binary sequence of: 00, 01, 10, 11, 00, 01 …etc.

Clock Pulse	Present State			Next State		State Diagram
	Q_B	Q_A		Q_B	Q_A	
0 (start)	0	0	⇒	0	1	
1	0	1	⇒	1	0	
2	1	0	⇒	1	1	
3	1	1	⇒	0	0	
4 (repeat)	0	0	⇒	0	1	



Then we have seen that a MOD-2 counter consists of a single flip-flop and a MOD-4 counter requires two flip-flops,allowing it to count in four discrete steps. We could easily add another flip-flop onto the end of a MOD-4 counter to produce a MOD-8 counter giving us a 23 binary sequence of counting from 000 up to 111, before resetting back to 000. A fourth flip-flop would make a MOD-16 counter and so on, in fact we could go on adding extra flip-flops for as long as we wanted.

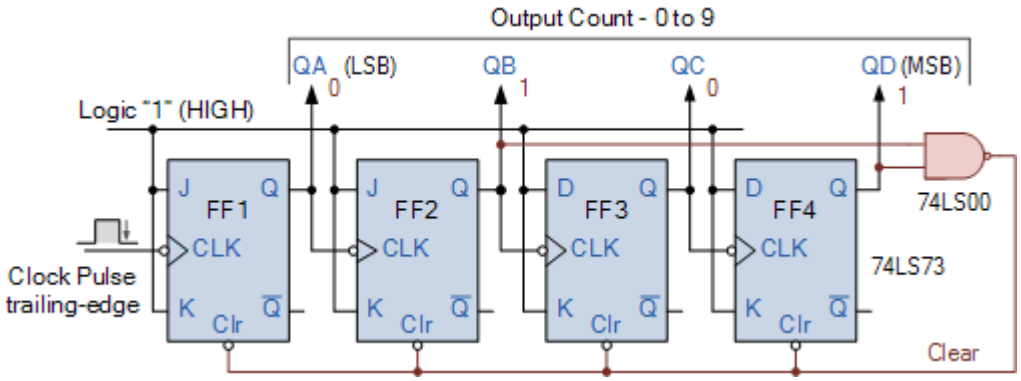
Modulus 10 Mod Counters

A good example of a modulo-m counter circuit which uses external combinational circuits to produce a counter with a modulus of 10 is the Decade Counter. Decade (divide-by-10) counters such as the TTL 74LS90, have 10 states in its counting sequence making it suitable for human interfacing where a digital display is required.

The decade counter has four outputs producing a 4-bit binary number and by using external AND and OR gates we can detect the occurrence of the 9th counting state to reset the counter back to zero. As with other mod counters, it receives an input clock pulse, one by one, and counts up from 0 to 9 repeatedly.

Once it reaches the count 9 (1001 in binary), the counter goes back to 0000 instead of continuing on to 1010. The basic circuit of a decade counter can be made from JK flip-flops (TTL 74LS73) that switch state on the negative trailing-edge of the clock signal as shown.

MOD-10 Decade Counter



MOD Counters Summary

We have seen in this tutorial about **MOD Counters** that binary counters are sequential circuits that generate binary sequences of bits as a result of a clock signal and the state of a binary counter is determined by the specific combination formed by all the counters outputs together.

The number of different output states a counter can produce is called the modulo or modulus of the counter. The Modulus (or MOD-number) of a counter is the total number of unique states it passes through in one complete counting cycle with a mod-n counter being described also as a divide-by-n counter.

The modulus of a counter is given as: 2n where n = number of flip-flops. So a 3 flip-flop counter will have a maximum count of 23 = 8 counting states and would be called a MOD-8 counter. The maximum binary number that can be counted by the counter is 2n–1 giving a maximum count of (111)2 = 23–1 = 710. Then the counter counts from 0 to 7.

Common MOD counters include those with MOD numbers of 2, 4, 8 and 16 and with the use of external combinational circuits can be configured to count to any predetermined value other than one with a maximum 2n modulus. In general, any arrangement of a “m” number of flip-flops can be used to construct any MOD counter.

A common modulus for counters with truncated sequences is ten (1010), called MOD-10. A counter with ten states in its sequence is known as a decade counter. Decade counters are useful for interfacing to digital displays. Other MOD counters include the MOD-6 or MOD-12 counter which have applications in digital clocks to display the time of day.

EXPERIMENT PROCEDURE:

1. Define output directory location.
2. Setup design sources and constraints.
3. Run synthesis, report utilization and timing estimates, write checkpoint design.
4. Run placement and logic optimization, report utilization and timing estimates, write checkpoint design.
5. Run router, report actual utilization and timing, write checkpoint design, run drc, write verilog and xdc out.

6. Generate a bitstream.
7. Refer “Create your first Project in Vivado” document available on Classroom for more details

POST-LAB:

1. Design MOD “Roll No.” Counter

APPLICATION:

OBSERVATION TABLE:

Logic Utilization	Used	Available	% Utilization
Number of Slices			
Number of Slice Flip Flops			
Number of 4 input LUTs			
Number of bonded IOBs			
Number of BRAMs			
Number of GCLKs			

RESULTS:

Print outs showing RTL schematic and implementation on FPGA are attached at the end of experiment**.**

CONCLUSION:

Viva Questions:

1. How to design MOD 5 Counter?

Experiment No: 4

To design FIFO Memory

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: To design FIFO memory

OBJECTIVE:

1. Write, synthesize and simulate VHDL code of FIFO memory.
2. Implement on FPGA.

KEYWORDS: FIFO, SRAM, etc.

PRE-LAB:

1. Study different types of memories
2. Study hardware structure of memories

APPARATUS: Basys 3 FPGA development board.

EXPLANATION:A FIFO is a special type of buffer. The name FIFO stands for first in first out and means that the data written into the buffer first comes out of it first.FIFOs are commonly used in electronic circuits for buffering and flow control between hardware and software. In its hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be SRAM, flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size, a dual-port SRAM is usually used, where one port is dedicated to writing and the other to reading.

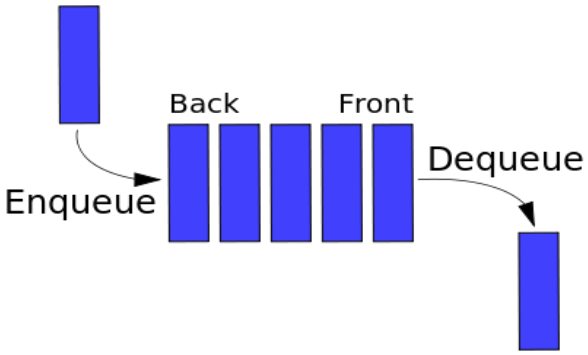


Figure 1: Representation of a FIFO (first in, first out) queue

A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing. Asynchronous FIFOs introduce metastability issues. A common implementation of an asynchronous FIFO uses a Gray code (or any unit distance code) for the read and write pointers to ensure reliable flag generation. One further note concerning flag generation is that one must necessarily use pointer arithmetic to generate flags for asynchronous FIFO implementations.

Examples of FIFO status flags include: full, empty, almost full, almost empty, etc.

A brief summary of the FIFO's operation is as follows. The write and read pointers are essentially 4-bit registers whose outputs are processed by 4:16 decoders to select one of the sixteen words in the memory array. The **reset** input is used to initialize the device, primarily by clearing the

write and read pointers such that they both point to the same memory word. The initialization also causes the **empty** output to be placed in its active state and the **full**output to be placed in its inactive state.

The write and read pointers chase each other around the memory array in an endless loop. An active edge on the **write** input causes any data on the **data-in[3:0]** bus to be written into the word pointed to by the write pointer, the **empty** output is placed in its inactive state (because the device is no longer empty) and the write pointer is incremented to point to the next empty word.

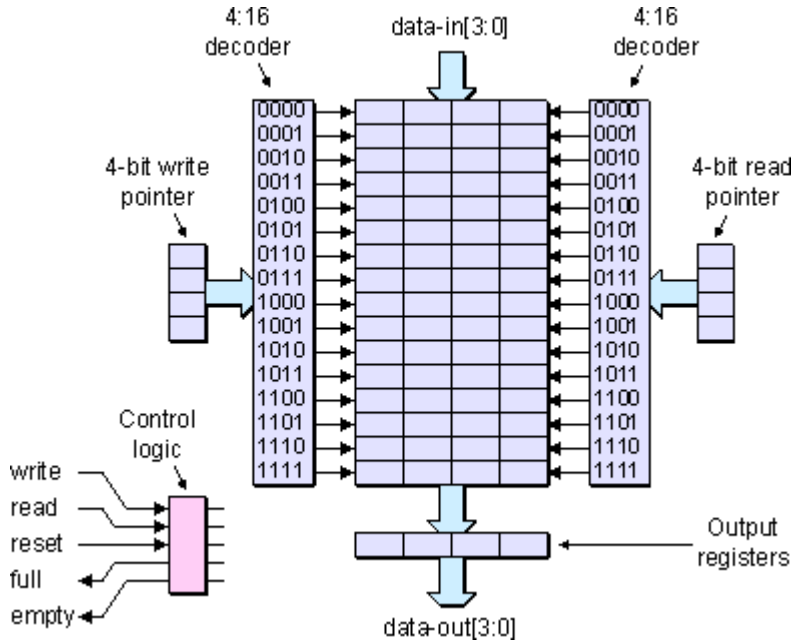


Figure 2: First-in first-out (FIFO) memory.

Data can be written into the FIFO until all the words in the array contain values. When the write pointer catches up to the read pointer, the **full** output is placed in its active state (indicating that the device is full) and no more data can be written into the device.

An active edge on the **read** input causes the data in the word pointed to by the read pointer to be copied into the output register; the **full** output is placed in its inactive state and the read pointer is incremented to point to the next word containing data. (These discussions assume *write-and-increment* and *read-and-increment techniques*, but it should be noted that some FIFOs employ an increment-and-write and *increment-and-read* approach.)

Data can be read out of the FIFO until the array is empty. When the read pointer catches up to the write pointer, the **empty** output is placed in its active state, and no more data can be read out of the device.

EXPERIMENT PROCEDURE:

1. Define output directory location.
2. Setup design sources and constraints.
3. Run synthesis, report utilization and timing estimates, write checkpoint design.

4. Run placement and logic optimization, report utilization and timing estimates, write checkpoint design.
5. Run router, report actual utilization and timing, write checkpoint design, run drc, write verilog and xdc out.
6. Generate a bitstream.
7. Refer “Create your first Project in Vivado” document available on Classroom for more details

POST-LAB:

1. Write VHDL code for stack.
2. Implement RAM memory on FPGA

APPLICATION:

OBSERVATION TABLE:

Logic Utilization	Used	Available	% Utilization
Number of Slices			
Number of Slice Flip Flops			
Number of 4 input LUTs			
Number of bonded IOBs			
Number of BRAMs			
Number of GCLKs			

RESULT:

Print outs showing RTL schematic and implementation on FPGA are attached at the end of experiment**.**

CONCLUSION:

VIVA QUESTIONS:

1. What is the application of FIFO?
2. What are different scheduling algorithms?
3. How to optimize hardware of FIFO?
4. How to avoid metastability in Flip flops?
5. How is logic implemented in FPGA?

Experiment No: 5

To interface keypad with FPGA

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: To interface keypad with FPGA

OBJECTIVE:

1. Write, synthesize and simulate VHDL code to interface keypad with FPGA.
2. Implement on FPGA.

KEYWORDS: Matrix, Row, Col, etc.

PRE-LAB:

1. Study keypad scanning algorithm
2. Study state diagram and VHDL code for it

APPARATUS: Basys 3 FPGA development board.

EXPLANATION:

The Keypad Pmod has a row/column matrix circuit to reduce the number of IO required to use it. Figure 3 shows the schematic of the Pmod's relevant circuitry (except Digilent labeled the switch names in row 4 incorrectly).

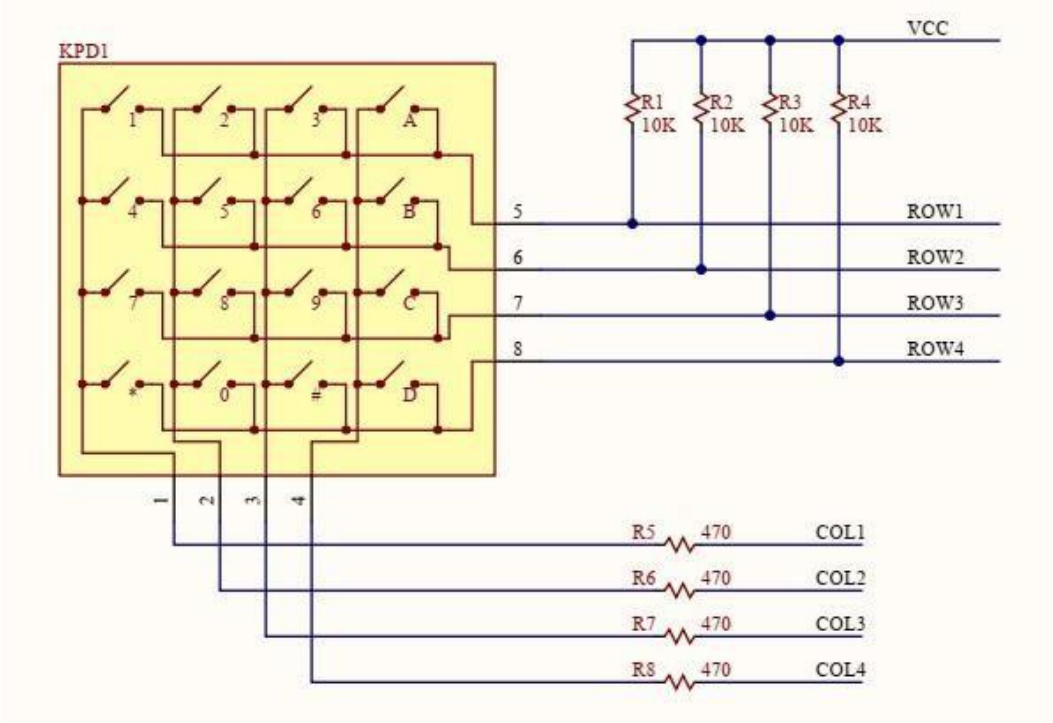
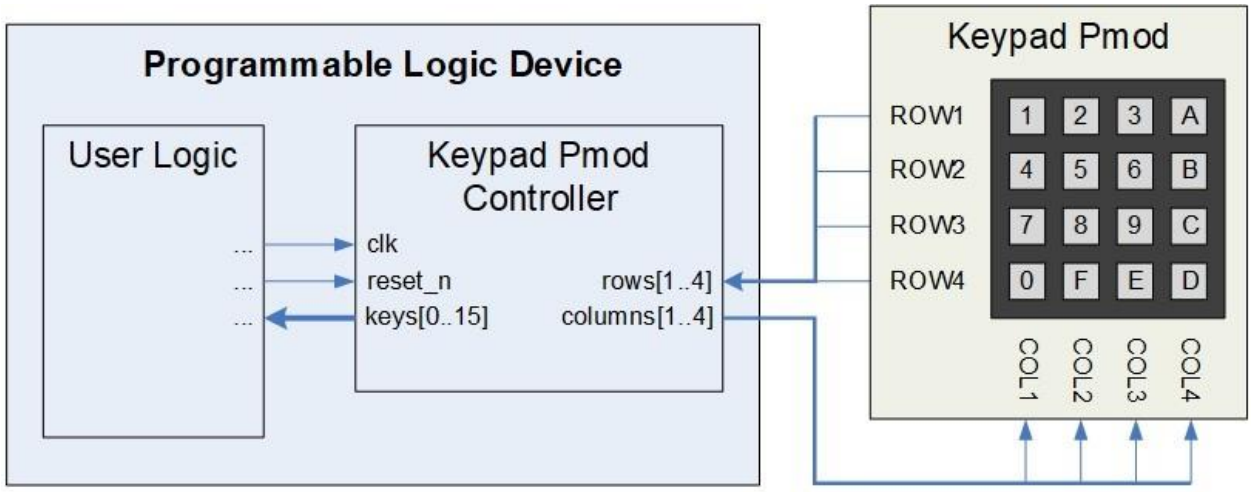


Figure : Keypad Pmod Circuit



Detecting Single Key Presses

With no keys pressed, the row outputs are pulled up to a logic high through the 10 kΩ resistors. A single key press forms a voltage divider between the row's 10 kΩ resistor and the connected column's 470 Ω resistor. If the column is pulled low by the FPGA, then the row output drops below the logic low threshold voltage and is read by the FPGA as a '0'. Single key presses are reliably detected by reading the rows while pulling the columns low one at a time to determine which key in the row is pressed.

This approach reliably detects that a particular key was pressed *if only one key was pressed.*

However, additional steps still need to be taken to eliminate both false positive and false negative detections that can result if multiple keys were pressed simultaneously.

Detecting Two Simultaneous Key Presses

The single key press approach described above also detects most dual key presses but not all. It reliably detects dual key presses *if only two keys are pressed and if those two keys are not in the same row.*

Suppose two keys in the same row are pressed simultaneously. Since one of their columns is held high by the FPGA, that column's 470 Ω resistor is then in parallel with the row's 10 kΩ resistor. This changes the voltage divider circuit such that the row output is not below the logic low threshold voltage, so the FPGA input incorrectly reads that no keys are pressed (a false negative for each of the two keys).

The Keypad Pmod Controller corrects for this dual key press problem. It checks whether two keys in the same row are pressed simultaneously by pulling those two columns low at the same time. Figure 4 depicts the resulting circuit. If both keys are pressed, the row output falls below the logic low voltage threshold. Note, the row output also goes low if either of the keys is pressed alone.

Therefore, Keypad Pmod Controller only accepts the result as a dual key press if neither key was independently detected by the single key press approach.

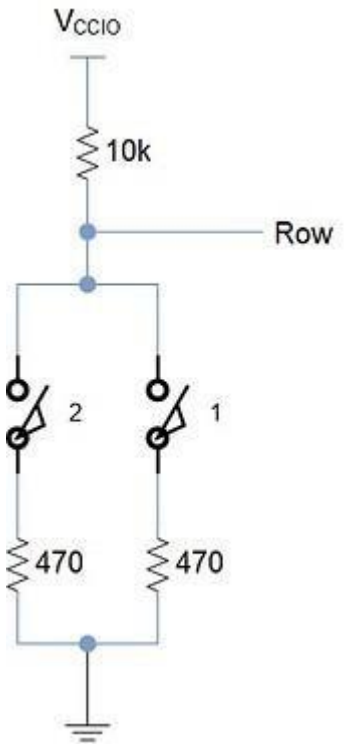


Figure: One Row with Two Columns Pulled Low

Eliminating False Negatives and False Positives

Three or more simultaneous key presses make the above approaches unreliable. Problems develop because resistors are usually then being connected in parallel. Many such combinations lead to either false negative or false positive results, because they drastically alter the voltage divider circuits.

After cycling through the columns and polling the rows for each single and dual key press scenario, the Keypad Pmod Controller counts the key presses detected during that polling cycle before outputting the results. If more than two keys were pressed, all results are suspect and unreliable, so no results are accepted or outputted.

In this manner, all output values are definite, reliable key presses. If more than two keys are pressed simultaneously, they are all ignored as operator error, since it is then impossible to reliably determine which keys are pressed or which are not.

Detect Time

The Keypad Pmod Controller also calculates the number of input clock cycles needed for each polling check. Once a column is pulled low, a row with a key press has an appreciable time

constant before that input reaches the logic low threshold. To meet this requirement, the Controller waits 300 ns before reading the values.

The Controller performs this function using the GENERIC parameter *clk_freq*, which must be set to the frequency of the system clock input *clk*.

Debounce

Once the polling cycle completes and the detected key presses are counted to verify their legitimacy, the Controller debounces the results before outputting them. The GENERIC parameter *stable_time* specifies how many milliseconds a key value must remain stable before being counted as a single press.

Configuring the Controller

The Keypad Pmod Controller is configured by setting the GENERIC parameters in the ENTITY. Table 1 describes the parameters.

Table 1. Generic Parameter Descriptions

Generic	Data Type	Default	Description
clk_freq	integer	50_000_000	Frequency of the system clock input (PORT clk) (Hertz)
stable_time	integer	10	Time a key press input must remain stable to be considered valid and debounced (milliseconds)

Port Descriptions

Table 2 describes the Keypad Pmod Controller's ports.

Port	Width	Mode	Data Type	Interface	Description
clk	1	in	standard logic	user logic	System clock
reset_n	1	in	standard logic	user logic	Asynchronous active low reset
rows	4	in	standard logic vector	keypad pmod	Reads the keypad pmod rows to determine if the key(s) in the selected column(s) is pressed
columns	4	buffer	standard logic vector	keypad pmod	Drives each keypad pmod column high (not selected) or low (selected) to select which column(s) in each row is being polled
keys	16	out	standard logic vector	user logic	Outputs which keys are being pressed ('1' = pressed, '0' = not pressed), keys(0) to keys(15) correspond to keypad keys 0 to F respectively

Connections

This Pmod has one 12-pin connector. Table 3 provides the pinout. The Keypad Pmod Controller's ports need to be assigned to the FPGA pins that are routed to these connectors as listed.

Table 3. Keypad Pmod Pinout and Connections to Keypad Pmod Controller

Pmod Connector	Pmod Pin Number	Pmod Port	Keypad Pmod Controller Port
J1	1	COL4	columns(4)
J1	2	COL3	columns(3)
J1	3	COL2	columns(2)
J1	4	COL1	columns(1)
J1	5	GND	-
J1	6	VCC	-
J1	7	ROW4	rows(4)
J1	8	ROW3	rows(3)
J1	9	ROW2	rows(2)
J1	10	ROW1	rows(1)
J1	11	GND	-
J1	12	VCC	-

Reset

The *reset_n* input port must have a logic high for the Keypad Pmod Controller component to operate. A low logic level on this port asynchronously resets the component. During reset, the component sets the *columns* outputs high, clears the *keys* outputs and clears all relevant internal registers. Once released from reset, the Keypad Pmod Controller resumes operation.

Conclusion

This Keypad Pmod Controller is a programmable logic component that interfaces to Digilent's Keypad Pmod. It handles all processes necessary to reliably detect and debounce up to 2 simultaneous key presses and output the results on a parallel interface. The Controller also determines if more than 2 keys are simultaneously pressed and outputs all zeros in this case to indicate that no definite, reliable key presses can be determined, thereby eliminating the false positives and false negatives inherent in the keypad design.

EXPERIMENT PROCEDURE:

1. Define output directory location.
2. Setup design sources and constraints.
3. Run synthesis, report utilization and timing estimates, write checkpoint design.
4. Run placement and logic optimization, report utilization and timing estimates, write checkpoint design.
5. Run router, report actual utilization and timing, write checkpoint design, run drc, write verilog and xdc out.
6. Generate a bitstream.
7. Refer "Create your first Project in Vivado" document available on Classroom for more details

POST-LAB:

1. Write VHDL code for interfacing keypad with multiple key press detection

APPLICATION:

_____

_____

_____

_____

_____

RESULTS:

Print outs showing RTL schematic and implementation on FPGA are attached at the end of experiment**.**

CONCLUSION:

_____

_____

_____

_____

VIVA QUESTIONS:

1. What are different types of switches?
2. What is the difference between mealy and moore machines in terms of VHDL code?
3. What is key debouncing? Have you considered this assignment?
4. Why have you preferred the moore machine in this experiment?
5. Can you write VHDL code by using structural modeling technique?

Experiment No: 6

Simulation of CMOS Inverter, NAND, NOR gates

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: Simulation of CMOS Inverter, NAND, NOR gates

OBJECTIVES:

1. To design and simulate CMOS Inverter, NAND, NOR gates

KEYWORDS: n- well, diffusion layer, polysilicon, etc.

PRE-LAB:

1. Study MOSFET level implementation of basic gates.
2. Study calculation of area.
3. Study various modes of power dissipation.

TOOLS REQUIRED: Microwind 3.1

THEORY:

Inverter:

As you can see from Figure 1, a CMOS circuit is composed of two MOSFETs. The top FET is a PMOS type device while the bottom FET is an NMOS type. Both gates are connected to the input line. The output line connects to the drains of both FETs.

Truth table	
Input (V_{IN})	Output(V_{OUT})
0	1
1	0

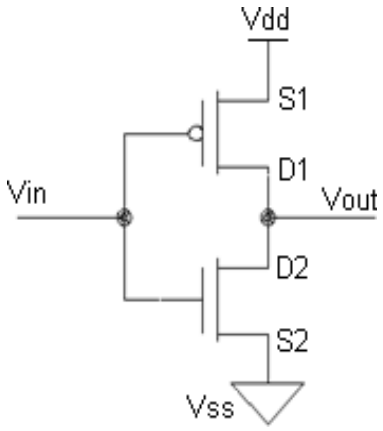


Figure 1: CMOS Inverter

Following figure shows the stick diagram of the inverter.

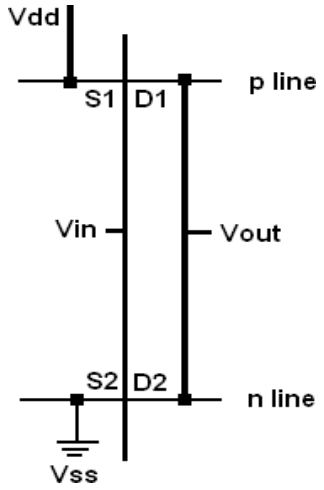


Figure 2: Stick diagram ofCMOS Inverter

The operation of the CMOS inverter can be divided into five regions indicated on Figure 3(c). The state of each transistor in each region is shown in Table 2. In region *A*, the nMOS transistor is OFF so the pMOS transistor pulls the output to V_{DD}. In region *B*, the nMOS transistor starts to turn ON, pulling the output down. In region *C*, both transistors are in saturation. Notice that ideal transistors are only in region *C* for $V_{in} = V_{DD}/2$ and thatthe slope of the transfer curve in this example is $-\infty$ in this region, corresponding to infinite gain. Real transistors have finite output resistances on account of channel length modulation and thus have finite slopes over a broader region *C*. In region *D*, the pMOS transistor is partially ON and in region *E*, it is completely OFF, leaving the nMOS transistor to pull the output down to GND. Also notice that the inverter's current consumption is ideally zero, neglecting leakage, when the input is within a threshold voltage of the V_{DD}or GND rails. This feature is important for low-power operation.

Table 2: Summary of CMOS inverter operation

Region	Condition	p-device	n-device	Output
A	$0 \leq V_{in} < Vtn$	linear	cutoff	$V_{out} = V_{DD}$
B	$Vtn \leq V_{in} < V_{DD}/2$	linear	saturated	$V_{out} > V_{DD}/2$
C	$V_{in} = VDD/2$	saturated	saturated	V_{out} drops sharply
D	$V_{DD}/2 < V_{in} \leq V_{DD} - \|Vtp\|$	saturated	linear	$V_{out} < V_{DD}/2$
E	$V_{in} > V_{DD} - \|Vtp\|$	cutoff	linear	$V_{out} = 0$

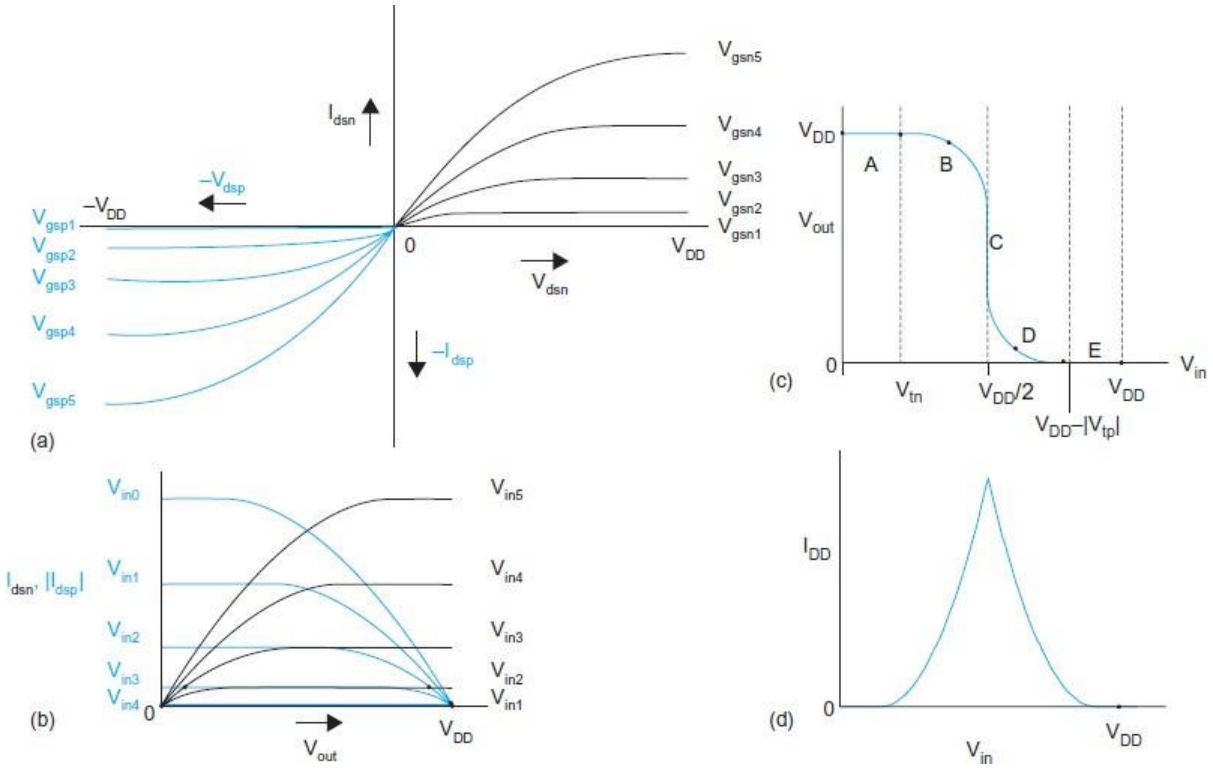
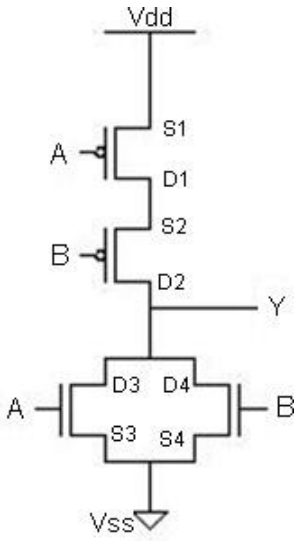


Figure 3: Graphical derivation of CMOS inverter DC characteristic

Nor Gate:



Truth Table		
A	**B**	**Y**
0	0	1
0	1	0
1	0	0
1	1	0

Figure 4: CMOS NOR Gate

When either one or both inputs are high, i.e., when the n-net creates a conducting path between the output node and the ground, the p-net is cut-off. On the other hand, if both input voltages are low, i.e., the n-net is cut-off, and then the p-net creates a conducting path between the output node and the supply voltage V_{DD}. Thus, the dual or complementary circuit structure allows that, for any given input combination, the output is connected either to V_{DD}or to ground via a low-resistance path. A DC current path between the $V_{DD}$and ground is not established for any of the input combinations. This results in the fully complementary operation mode already examined for the simple CMOS inverter circuit.

The output voltage of the CMOS NOR gate will attain a logic-low voltage of V_{OL}= 0 and a logic-high voltage of V_{OH}= V_{DD}. For circuit design purposes, the switching threshold voltage *Vth* of the CMOS gate emerges as an important design criterion. Following figure shows a stick diagram of NOR gate.

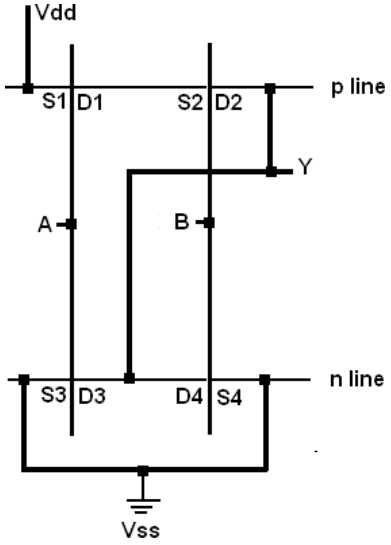
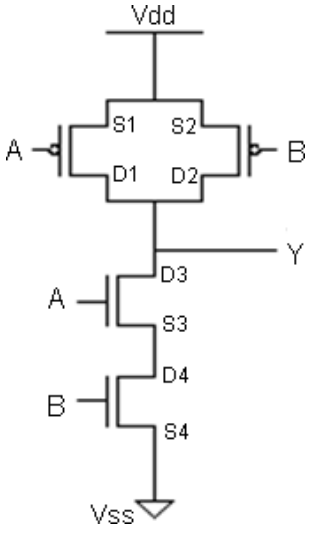


Figure 5: Stick diagram of NOR gate

Nand Gate:



Truth Table		
A	**B**	**Y**
0	0	1
0	1	1
1	0	1
1	1	0

Figure 6: CMOS NAND Gate

The two-input NAND gate shown on the left is built from four transistors. The series-connection of the two n-channel transistors between GND and the gate-output ensures that the gate-output is only driven low (logical 0) when both gate inputs A or B are high (logical 1). The complementary parallel connection of the two transistors between Vdd and gate-output means that the gate-output is driven high (logical 1) when one or both gate inputs are low (logical 0). The net result is the logical NAND function. Following figure shows stick diagram of NAND gate.

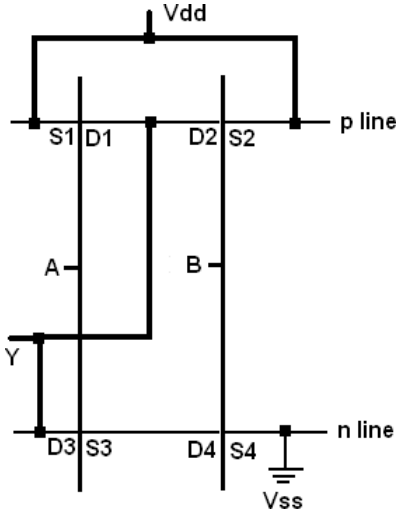


Figure 7: Stick diagram of NAND gate

POST-LAB:

1. Calculate power dissipation in all above circuits.
2. Calculate area of all the circuits.

APPLICATION:

_____

_____

_____

_____

OBSERVATIONS: Euler's Path

RESULT:

Print outs showing layout of the circuits and their simulation results are attached at the end of experiment**.**

CONCLUSION:

_____

_____

_____

_____

VIVA QUESTIONS:

1. What are different sources of power dissipation?
2. What is the W/L ratio?
3. What are the steps to design layout?
4. Design 2:1 Mux using this method.
5. What are the steps for manufacturing a MOSFET?

Experiment No: 7

Simulation of Half Adder and Full Adder

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: Simulation of half adder and full adder

OBJECTIVES:

1. To design half adder and full adder
2. To draw a circuit diagram of a half adder and full adder
3. To draw a stick diagram of a half adder and full adder

KEYWORDS: Half Adder, Full adder, n- well, diffusion layer, polysilicon, etc.

PRE-LAB:

1. Study different configurations of half adder and full adder

TOOLS REQUIRED: Microwind 3.1

THEORY:

Half Adder:

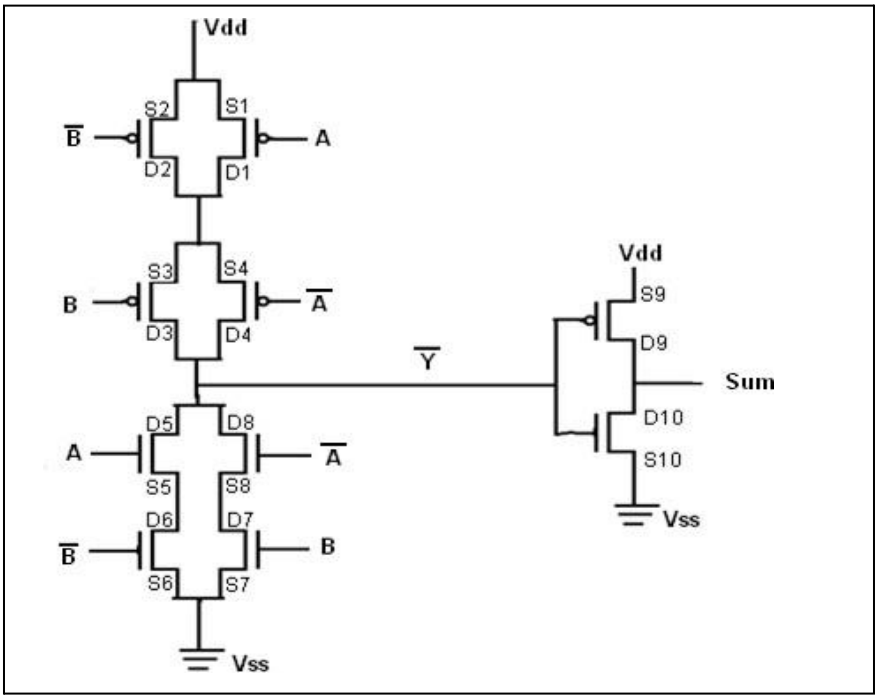


Figure 8: Circuit diagram to generate Sum of two inputs

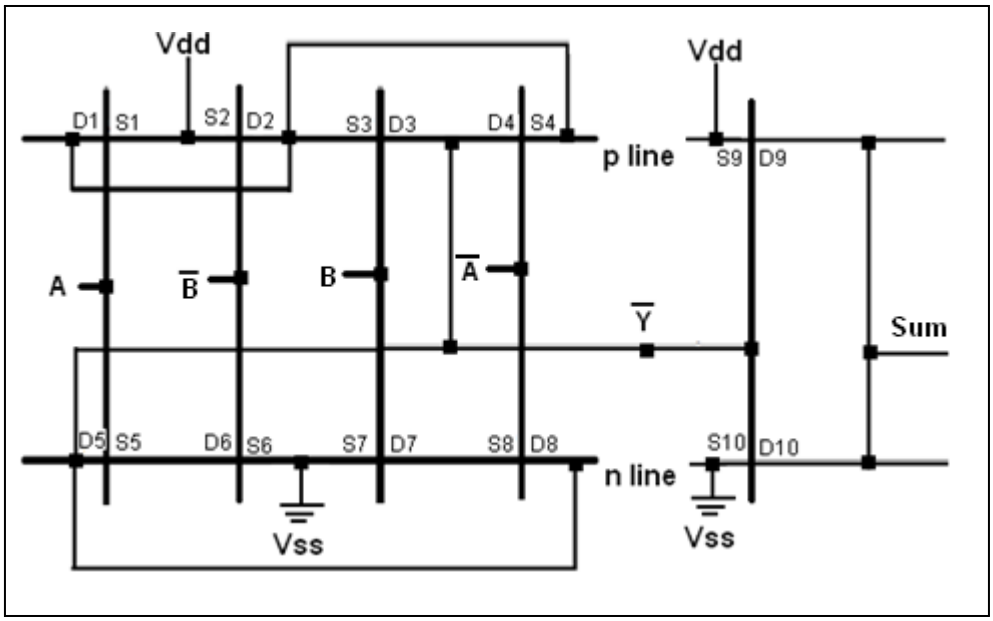


Figure 9: Stick diagram of Sum

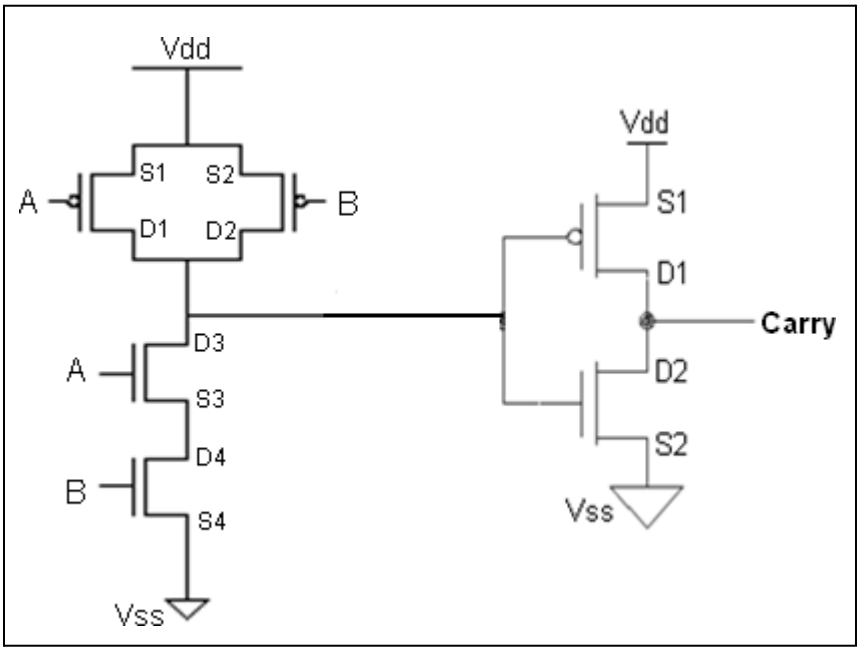


Figure 10: Circuit diagram to generate Carry of two inputs

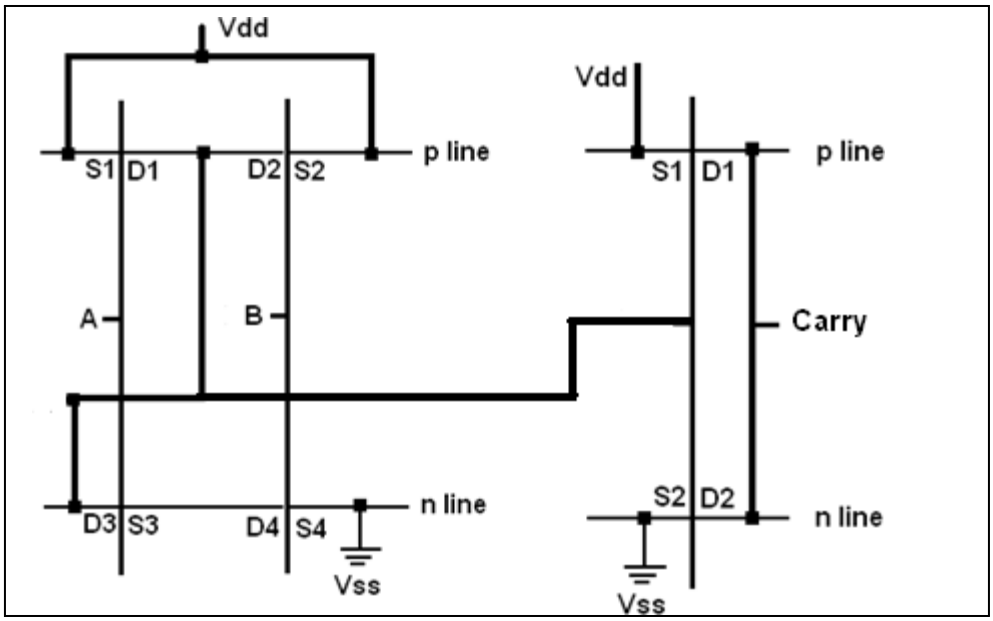


Figure 11: Stick diagram of Carry

POST LAB:

Design similar circuits by referring boolean equations

APPLICATION:

_____

_____

_____

_____

CALCULATIONS: (W/L) ratios of circuits

_____

_____

_____

_____

_____

OBSERVATIONS: Euler's Path

RESULT:

Print outs showing layout of the circuits and their simulation results are attached at the end of experiment**.**

CONCLUSION:

_____

_____

_____

_____

VIVA QUESTIONS:

1. Compare other ways of design half adder and full adder

Experiment No: 8

Simulation of 2:1 Mux by using Logic Gates and by using Transmission gates

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: Simulation of 2:1 Mux by conventional method and by using Transmission gates.

OBJECTIVES:

1. To design 2:1 Mux by conventional method.
2. To draw a circuit diagram of 2:1 Mux by conventional method.
3. To draw a stick diagram of 2:1 Mux by conventional method.
4. To draw a circuit diagram of 2:1 Mux by using Transmission gates.
5. To draw the layout of the 2:1 Mux directly using Transmission gates.

KEYWORDS: Pass Transistor,n- well, diffusion layer, polysilicon, etc.

PRE-LAB:

1. Study concept of pass transistors
2. Overview hardware of mux.

TOOLS REQUIRED: Microwind 3.1

THEORY: The CMOS TG operates as a bidirectional switch between the nodes A and B which is controlled by signal C. If the control signal C is logic-high, i.e., equal to V_{DD}, then both transistors are turned on and provide a low resistance current path between the nodes A and B. If, on the other hand, the control signal C is low, then both transistors will be off, and the path between the nodes A and B will be an open circuit. This condition is also called the high-impedance state. Note that the substrate terminal of the nMOS transistor is connected to ground and the substrate terminal of the pMOS transistor is connected to V_{DD}**.**Thus, we must take into account the substrate-bias effect for both transistors, depending on the bias conditions. Figure 1 also shows three other commonly used symbolic representations of the CMOS transmission gate.

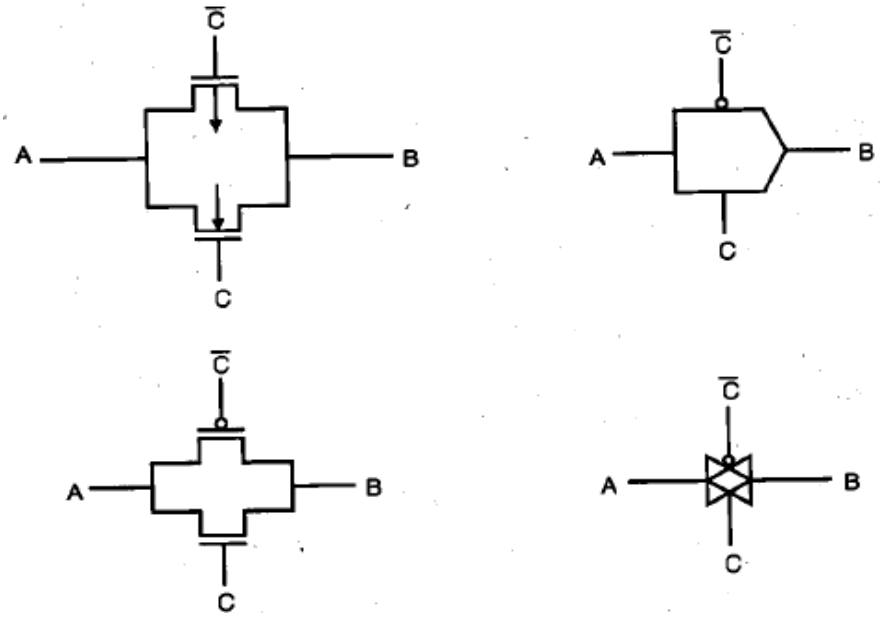


Figure 1: Four different representations of CMOS Transmission gate(TG)

For a detailed DC analysis of the CMOS transmission gate, we will consider the following bias condition, shown in Figure 2. The input node (A) is connected to a constant logic-high voltage, *Vin*= V_{DD}. The control signal is also logic-high, thus ensuring that both transistors are turned on. The output node (B) may be connected to a capacitor,which represents capacitive loading of the subsequent logic stages driven by the transmission gate. We will now investigate the input-output current-voltage relationship of the CMOS TG as a function of the output voltage *Vout*.

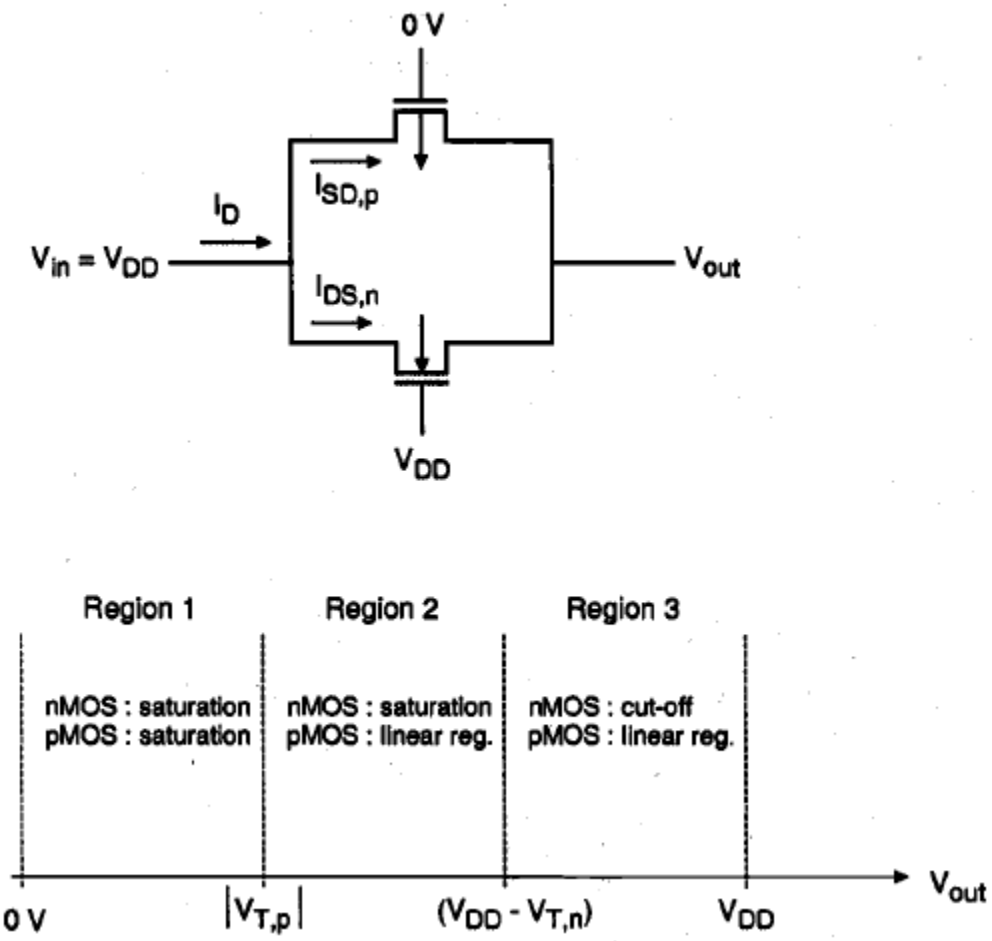


Figure 2: Bias conditions and operating regions of CMOS Transmission gate, shown as functions of the output voltage

It can be seen from Figure 2 that the drain-to-source and the gate-to-source voltages of the nMOS transistor are

$$V_{DS,n} = V_{DD} - Vout$$
$$V_{GS,n} = V_{DD} - Vout$$

Thus, the nMOS transistor will be turned off for $Vout > V_{DD} - V_{Tn}$ and will operate in the saturation mode for $Vout < V_{DD} - V_{Tn}$. The V_{DS} and V_{GS} voltages of the pMOS transistor are

$$V_{DS,p} = Vout - V_{DD}$$

$V_{GS,p} = -V_{DD}$

Consequently, the pMOS transistor is in saturation for $V0ut < |V_{TP}|$, and it operates in the linear region for $Vout > |V_{TP}|$. Note that, unlike the nMOS transistor, the pMOS transistor remains turned on, regardless of the output voltage level *Vout*, This analysis has shown that we can identify three operating regions for the CMOS transmission gate, depending on the output voltage level. These operating regions are depicted in Figure 2 as functions of *Vout*. The total current flowing through the transmission gate is the sum of the nMOS drain current and the pMOS drain current.

$$ID = I_{DS,n} + I_{SD,p}$$

At this point, we may devise an *equivalent resistance* for each transistor in this structure, as follows.

$$R_{eq,n} = \frac{V_{DD} - V_{out}}{I_{DS,n}}$$

$$R_{eq,p} = \frac{V_{DD} - V_{out}}{I_{DS,p}}$$

Combining the equivalent resistance values found for the three operating regions, we can now plot the total resistance of the CMOS transmission gate as a function of the output voltage V_{OUT} as shown in Figure 3.

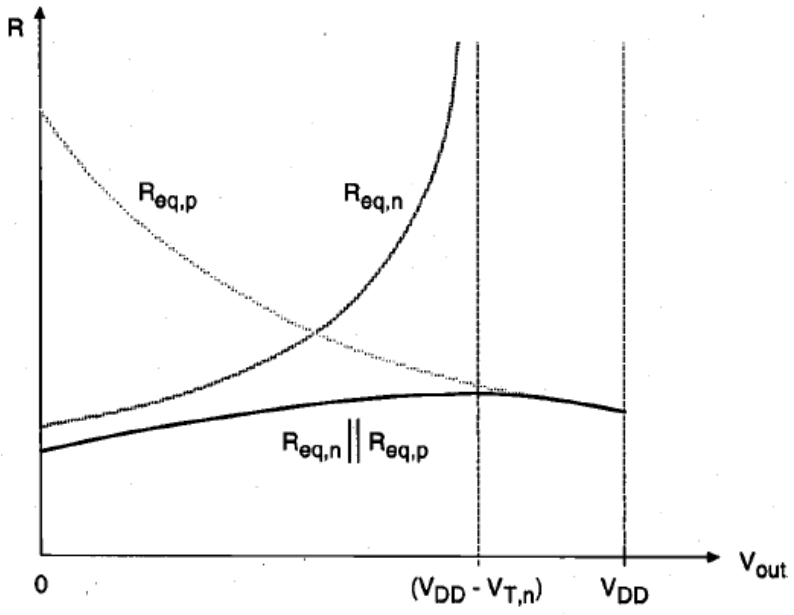
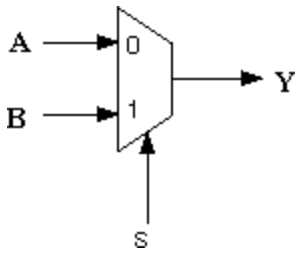


Figure 3: Equivalent resistance of the CMOS transmission gate, plotted as a function of the output voltage.

Mux:



$$Y = (A.\bar{S} + (B.S)\ Y = (A.\bar{S} + (B.S)$$

Truth Table:

S	Y
0	A
1	B

The MUX has property that it selects the input for output according to status of select line.

Circuit diagram:

1. **Using conventional method:** Following figure shows circuit diagram of 2:1 mux by using conventional method

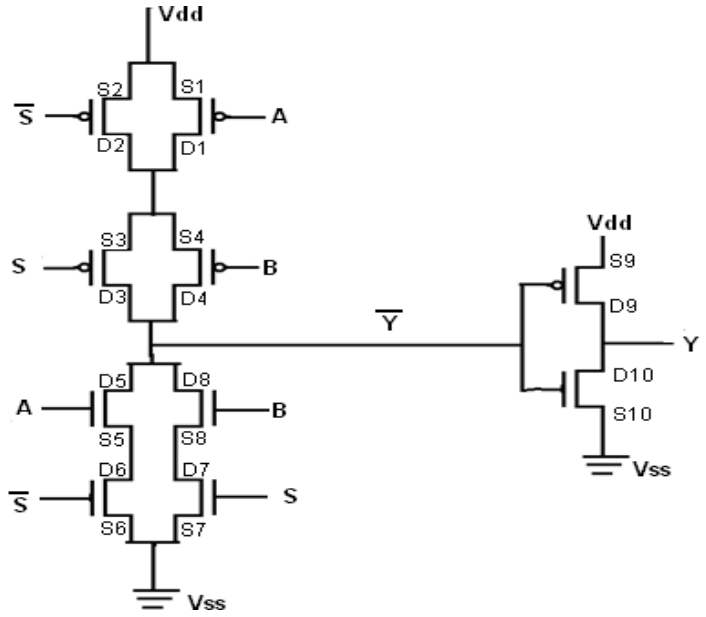


Figure 4: Circuit diagram of 2:1 mux by using conventional method

Stick diagram:

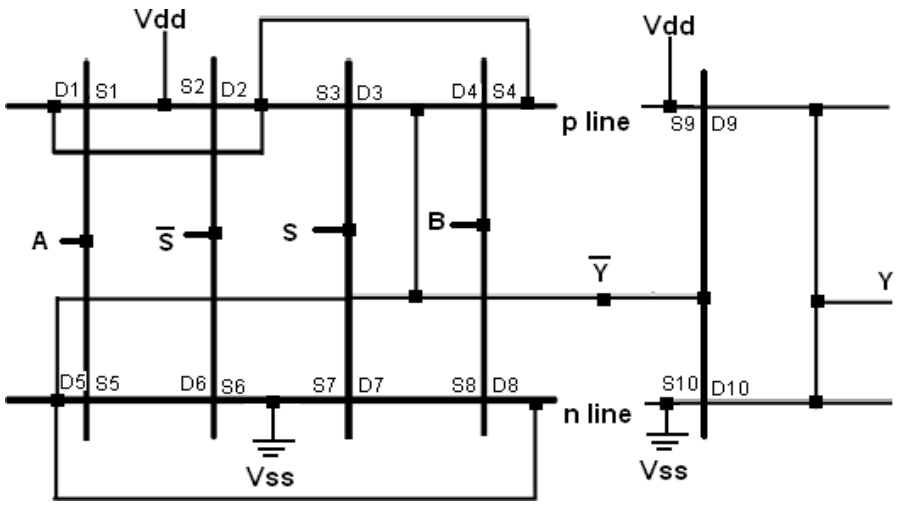


Figure 5: Stick diagram of 2:1 mux by using conventional method

2. **Using Transmission Gate:** Following figure shows circuit diagram of 2:1 mux by using Transmission Gate method.

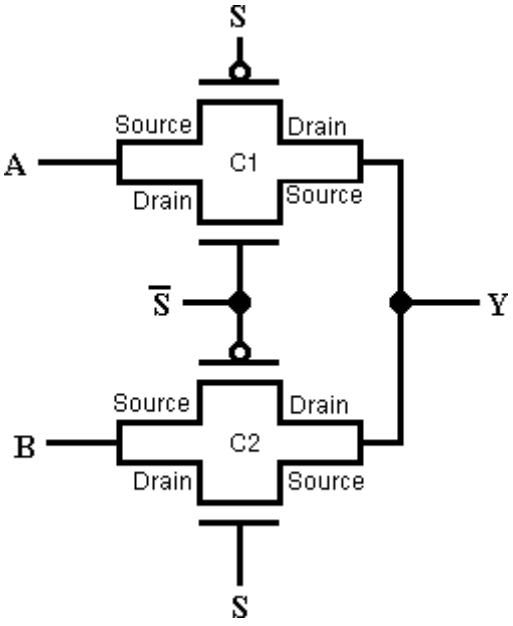


Figure 6: Circuit diagram of 2:1 mux by using Transmission Gate method.

POST LAB:

Design given Boolean equations by using Transmission gates

APPLICATION:

_____

_____

_____

_____

CALCULATIONS: (W/L) ratios of circuits

OBSERVATIONS: Euler's Path

RESULT:

Print outs showing layout of the circuits and their simulation results are attached at the end of experiment**.**

CONCLUSION:

VIVA QUESTIONS:

1. What are the drawbacks of pass transistors?
2. What are the applications of Transmission gates?
3. Explain different operating regions of CMOS Transmission gate.
4. What is the difference between 2:1 Mux by conventional method and by using Transmission gates?
5. Design XOR gate using TG.

Experiment No: 9

Simulation of single bit SRAM cell

Name of the Student: _____

Class: _____ Batch: _____

Date of Performance: _____

Signature of the Staff: _____

TITLE: Simulation of single bit SRAM cell

OBJECTIVES:

1. To design single bit SRAM cell.
2. To draw a circuit diagram of a single bit SRAM cell.
3. To draw a stick diagram of a single bit SRAM cell.

KEYWORDS: SRAM,DRAM,n- well, diffusion layer, polysilicon, etc.

PRE-LAB:

1. Study different configurations of SRAM cell.
2. Difference between DRAM and SRAM in terms of area, power and speed.

TOOLS REQUIRED: Microwind 3.1

THEORY: SRAM has become a major component in many VLSI Chips due to their large storage density and small access time. SRAM has become the topic of substantial research due to the rapid development for low power, low voltage memory design during recent years due to increase demand for notebooks, laptops, IC memory cards and hand held communication devices. SRAMs are widely used for mobile applications as both on chip and off-chip memories, because of their ease of use and low standby leakage.Low power on-chip memories have become the topic of substantial research as they can account for almost half of total CPU dissipation, even for extremely power-efficient designs As the technology increases integration density of transistors increases, power consumption has become a major concern in today's processors and SoC designs. Considerable attention has been paid to the design of low power and high-performance SRAMs as they are critical components in both handheld devices and high performance processors.

6T static random-access memory is a type of semiconductor memory that uses bistable latching circuitry to store each bit. The term static differentiates it from dynamic RAM which must be periodically refreshed. SRAM exhibits data remembrance, but is still volatile in conventional sense, that data is eventually lost when memory is not powered.

6T SRAM Celloperation:

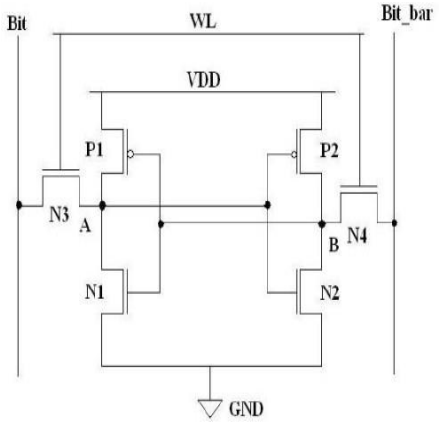


Figure 1: Schematic of 6T SRAM cell

1. Standby Mode (the circuit is idle)
In standby mode word line is not asserted (word line=0), so pass transistors N3 and N4 which connect 6T cell from bit lines are turned off. It means that the cell cannot be accessed. The two cross coupled inverters formed by N1-N2 will continue to feed back each other as long as they are connected to the supply, and data will hold in the latch.

2. Read Mode (the data has been requested)
In read mode word line is asserted (word line=1), Word line enables both the access transistor which will connect cell from the bit lines. Now values stored in nodes (node A and B) are transferred to the bit lines. Assume that 1 is stored at node A so bit line bar will discharge through the driver transistor (N1) and the bit line will be pull up through the Load transistors (P1) toward VDD, a logical 1. Design of the SRAM cell requires read stability (do not disturb data when reading).

3. Write Mode (updating the contents)
Assume that the cell is originally storing a 1 and we wish to write a 0. To do this, the bit line is lowered to 0V and the bit bar is raised to VDD, and the cell is selected by raising the word line to VDD. Typically, each of the inverters is designed so that PMOS and NMOS are matched, thus the inverter threshold is kept at VDD/2. If we wish to write 0 at node A, N3 operates in saturation. Initially, its source voltage is 1. Drain terminal of N2 is initially at 1 which is pulled down by N3 because the access transistor N3 is stronger than N1. Now N2 turns on and P1 turns off, thus a new value has been written which forces the bit line lowered to 0V and bit bar to VDD. SRAM to operate in write mode must have write-ability which is the minimum bit line voltage required to flip the state of the cell.

POST LAB:

1. Observe the effect of Rise time and fall time.
2. Design 4T SRAM cell.
3. Design ROM cell.

APPLICATION:

_____

_____

_____

_____

RESULT: Printouts showing layout of the circuits and their simulation results are attached at the end of experiment**.**

CONCLUSION:_____

_____

_____

_____

VIVA QUESTIONS:

1. What is the difference between SRAM and DRAM?
2. What is the application of SRAM?
3. Can we use SRAM as cache memory?
4. How to calculate the area of the SRAM cell in this assignment?
5. What do you understand from this SRAM structure?

Appendix

```
## This file is a general .xdc for the Basys3 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

# Clock signal
set_property PACKAGE_PIN W5 [get_ports CLK100MHZ]

        set_property IOSTANDARD LVCMOS33 [get_ports CLK100MHZ]
        create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports CLK100MHZ]

# Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
#set_property PACKAGE_PIN W16 [get_ports {sw[2]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
#set_property PACKAGE_PIN W17 [get_ports {sw[3]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {sw[3]}]
#set_property PACKAGE_PIN W15 [get_ports {sw[4]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {sw[4]}]
#set_property PACKAGE_PIN V15 [get_ports {sw[5]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {sw[5]}]
#set_property PACKAGE_PIN W14 [get_ports {sw[6]}]
```

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[6]}]

#set_property PACKAGE_PIN W13 [get_ports {sw[7]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[7]}]

#set_property PACKAGE_PIN V2 [get_ports {sw[8]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[8]}]

#set_property PACKAGE_PIN T3 [get_ports {sw[9]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[9]}]

#set_property PACKAGE_PIN T2 [get_ports {sw[10]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[10]}]

#set_property PACKAGE_PIN R3 [get_ports {sw[11]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[11]}]

#set_property PACKAGE_PIN W2 [get_ports {sw[12]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[12]}]

#set_property PACKAGE_PIN U1 [get_ports {sw[13]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[13]}]

#set_property PACKAGE_PIN T1 [get_ports {sw[14]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[14]}]

#set_property PACKAGE_PIN R2 [get_ports {sw[15]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {sw[15]}]

 # LEDs

set_property PACKAGE_PIN U16 [get_ports {LED[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]

set_property PACKAGE_PIN E19 [get_ports {LED[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]

set_property PACKAGE_PIN U19 [get_ports {LED[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]

set_property PACKAGE_PIN V19 [get_ports {LED[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]

set_property PACKAGE_PIN W18 [get_ports {LED[4]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[4]}]

set_property PACKAGE_PIN U15 [get_ports {LED[5]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[5]}]

set_property PACKAGE_PIN U14 [get_ports {LED[6]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[6]}]

set_property PACKAGE_PIN V14 [get_ports {LED[7]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[7]}]

set_property PACKAGE_PIN V13 [get_ports {LED[8]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[8]}]

set_property PACKAGE_PIN V3 [get_ports {LED[9]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[9]}]

set_property PACKAGE_PIN W3 [get_ports {LED[10]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[10]}]

set_property PACKAGE_PIN U3 [get_ports {LED[11]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[11]}]

set_property PACKAGE_PIN P3 [get_ports {LED[12]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[12]}]

set_property PACKAGE_PIN N3 [get_ports {LED[13]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[13]}]

set_property PACKAGE_PIN P1 [get_ports {LED[14]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[14]}]

set_property PACKAGE_PIN L1 [get_ports {LED[15]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[15]}]

#7 segment display

set_property PACKAGE_PIN W7 [get_ports {seg[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]

set_property PACKAGE_PIN W6 [get_ports {seg[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]

set_property PACKAGE_PIN U8 [get_ports {seg[2]}]

```
        set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]

set_property PACKAGE_PIN V7 [get_ports dp]
        set_property IOSTANDARD LVCMOS33 [get_ports dp]

set_property PACKAGE_PIN U2 [get_ports {an[0]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
set_property PACKAGE_PIN U4 [get_ports {an[1]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
set_property PACKAGE_PIN V4 [get_ports {an[2]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
set_property PACKAGE_PIN W4 [get_ports {an[3]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]


##Buttons
#set_property PACKAGE_PIN U18 [get_ports btnC]
        #set_property IOSTANDARD LVCMOS33 [get_ports btnC]
#set_property PACKAGE_PIN T18 [get_ports btnU]
        #set_property IOSTANDARD LVCMOS33 [get_ports btnU]
#set_property PACKAGE_PIN W19 [get_ports btnL]
        #set_property IOSTANDARD LVCMOS33 [get_ports btnL]
#set_property PACKAGE_PIN T17 [get_ports btnR]
```

#set_property IOSTANDARD LVCMOS33 [get_ports btnR]

#set_property PACKAGE_PIN U17 [get_ports btnD]

#set_property IOSTANDARD LVCMOS33 [get_ports btnD]

##Pmod Header JA

##Sch name = JA1

#set_property PACKAGE_PIN J1 [get_ports {JA[0]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[0]}]

##Sch name = JA2

#set_property PACKAGE_PIN L2 [get_ports {JA[1]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[1]}]

##Sch name = JA3

#set_property PACKAGE_PIN J2 [get_ports {JA[2]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[2]}]

##Sch name = JA4

#set_property PACKAGE_PIN G2 [get_ports {JA[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[3]}]

##Sch name = JA7

#set_property PACKAGE_PIN H1 [get_ports {JA[4]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[4]}]

##Sch name = JA8

#set_property PACKAGE_PIN K2 [get_ports {JA[5]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[5]}]

##Sch name = JA9

#set_property PACKAGE_PIN H2 [get_ports {JA[6]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[6]}]

##Sch name = JA10

#set_property PACKAGE_PIN G3 [get_ports {JA[7]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JA[7]}]

##Pmod Header JB

##Sch name = JB1

#set_property PACKAGE_PIN A14 [get_ports {JB[0]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[0]}]

##Sch name = JB2

#set_property PACKAGE_PIN A16 [get_ports {JB[1]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[1]}]

##Sch name = JB3

#set_property PACKAGE_PIN B15 [get_ports {JB[2]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[2]}]

##Sch name = JB4

#set_property PACKAGE_PIN B16 [get_ports {JB[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[3]}]

##Sch name = JB7

#set_property PACKAGE_PIN A15 [get_ports {JB[4]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[4]}]

##Sch name = JB8

#set_property PACKAGE_PIN A17 [get_ports {JB[5]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[5]}]

##Sch name = JB9

#set_property PACKAGE_PIN C15 [get_ports {JB[6]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[6]}]

##Sch name = JB10

#set_property PACKAGE_PIN C16 [get_ports {JB[7]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JB[7]}]

##Pmod Header JC

##Sch name = JC1

#set_property PACKAGE_PIN K17 [get_ports {JC[0]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[0]}]

##Sch name = JC2

#set_property PACKAGE_PIN M18 [get_ports {JC[1]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[1]}]

##Sch name = JC3

#set_property PACKAGE_PIN N17 [get_ports {JC[2]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[2]}]

##Sch name = JC4

#set_property PACKAGE_PIN P18 [get_ports {JC[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[3]}]

##Sch name = JC7

#set_property PACKAGE_PIN L17 [get_ports {JC[4]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[4]}]

##Sch name = JC8

#set_property PACKAGE_PIN M19 [get_ports {JC[5]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[5]}]

##Sch name = JC9

#set_property PACKAGE_PIN P17 [get_ports {JC[6]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[6]}]

##Sch name = JC10

#set_property PACKAGE_PIN R18 [get_ports {JC[7]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {JC[7]}]

#Pmod Header JXADC

#Sch name = XA1_P

set_property PACKAGE_PIN J3 [get_ports {vauxp6}]

set_property IOSTANDARD LVCMOS33 [get_ports {vauxp6}]

```
#Sch name = XA2_P
set_property PACKAGE_PIN L3 [get_ports {vauxp14}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxp14}]
#Sch name = XA3_P
set_property PACKAGE_PIN M2 [get_ports {vauxp7}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxp7}]
#Sch name = XA4_P
set_property PACKAGE_PIN N2 [get_ports {vauxp15}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxp15}]
#Sch name = XA1_N
set_property PACKAGE_PIN K3 [get_ports {vauxn6}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxn6}]
#Sch name = XA2_N
set_property PACKAGE_PIN M3 [get_ports {vauxn14}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxn14}]
#Sch name = XA3_N
set_property PACKAGE_PIN M1 [get_ports {vauxn7}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxn7}]
#Sch name = XA4_N
set_property PACKAGE_PIN N1 [get_ports {vauxn15}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxn15}]



##VGA Connector
#set_property PACKAGE_PIN G19 [get_ports {vgaRed[0]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[0]}]
#set_property PACKAGE_PIN H19 [get_ports {vgaRed[1]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[1]}]
#set_property PACKAGE_PIN J19 [get_ports {vgaRed[2]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[2]}]
```

#set_property PACKAGE_PIN N19 [get_ports {vgaRed[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[3]}]

#set_property PACKAGE_PIN N18 [get_ports {vgaBlue[0]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[0]}]

#set_property PACKAGE_PIN L18 [get_ports {vgaBlue[1]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[1]}]

#set_property PACKAGE_PIN K18 [get_ports {vgaBlue[2]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[2]}]

#set_property PACKAGE_PIN J18 [get_ports {vgaBlue[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[3]}]

#set_property PACKAGE_PIN J17 [get_ports {vgaGreen[0]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[0]}]

#set_property PACKAGE_PIN H17 [get_ports {vgaGreen[1]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[1]}]

#set_property PACKAGE_PIN G17 [get_ports {vgaGreen[2]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[2]}]

#set_property PACKAGE_PIN D17 [get_ports {vgaGreen[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[3]}]

#set_property PACKAGE_PIN P19 [get_ports Hsync]

#set_property IOSTANDARD LVCMOS33 [get_ports Hsync]

#set_property PACKAGE_PIN R19 [get_ports Vsync]

#set_property IOSTANDARD LVCMOS33 [get_ports Vsync]

##USB-RS232 Interface

#set_property PACKAGE_PIN B18 [get_ports RsRx]

#set_property IOSTANDARD LVCMOS33 [get_ports RsRx]

#set_property PACKAGE_PIN A18 [get_ports RsTx]

#set_property IOSTANDARD LVCMOS33 [get_ports RsTx]

##USB HID (PS/2)

#set_property PACKAGE_PIN C17 [get_ports PS2Clk]

#set_property IOSTANDARD LVCMOS33 [get_ports PS2Clk]

#set_property PULLUP true [get_ports PS2Clk]

#set_property PACKAGE_PIN B17 [get_ports PS2Data]

#set_property IOSTANDARD LVCMOS33 [get_ports PS2Data]

#set_property PULLUP true [get_ports PS2Data]

##Quad SPI Flash

##Note that CCLK_0 cannot be placed in 7 series devices. You can access it using the ##STARTUPE2 primitive.

#set_property PACKAGE_PIN D18 [get_ports {QspiDB[0]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[0]}]

#set_property PACKAGE_PIN D19 [get_ports {QspiDB[1]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[1]}]

#set_property PACKAGE_PIN G18 [get_ports {QspiDB[2]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[2]}]

#set_property PACKAGE_PIN F18 [get_ports {QspiDB[3]}]

#set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[3]}]

#set_property PACKAGE_PIN K19 [get_ports QspiCSn]

#set_property IOSTANDARD LVCMOS33 [get_ports QspiCSn]