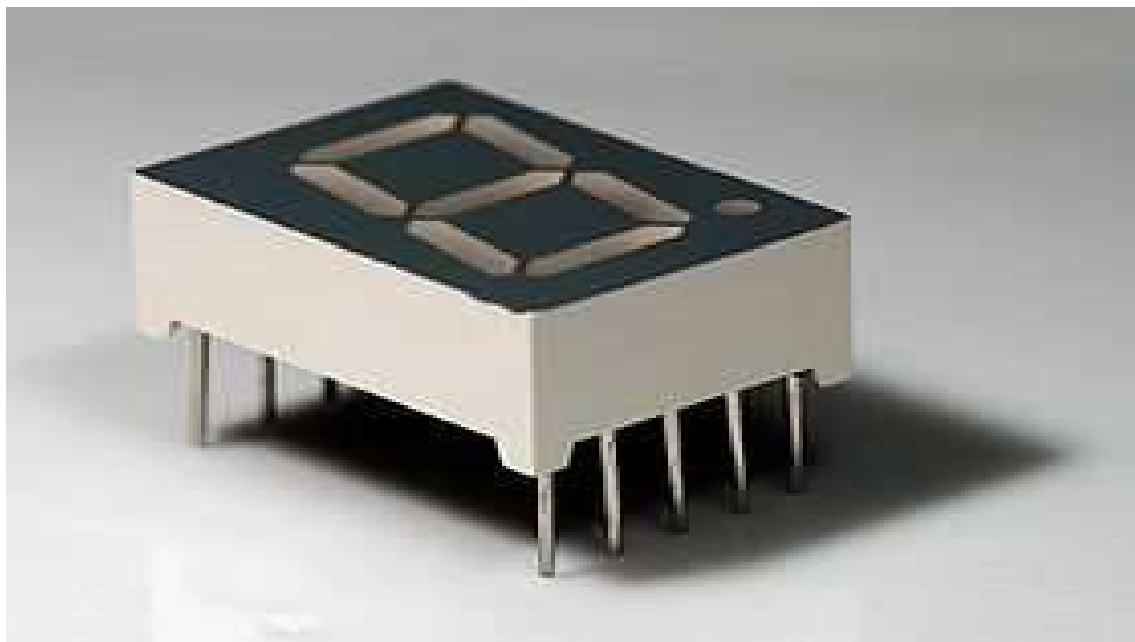




UNESCO-NIGERIA TECHNICAL &
VOCATIONAL EDUCATION REVITALISATION
PROJECT-PHASE II



NATIONAL DIPLOMA IN COMPUTER TECHNOLOGY



Introduction to Digital Electronics

**COURSE CODE: COM 112
PRACTICAL BOOK**

YEAR I, SEMESTER I

Version 1: December 2008

WEEK 1	Using Excel to convert numbers to different number systems	3
	Convert a binary number to decimal.....	4
	Convert a binary number to hexadecimal	5
	Convert a binary number to octal.....	7
WEEK 2	Converts a decimal number to binary	9
	Convert a decimal number to hexadecimal.....	11
	Convert a decimal number to octal	13
	Convert a hexadecimal number to binary	15
WEEK 3	Convert a hexadecimal number to decimal.....	16
	Convert a hexadecimal number to octal.....	18
WEEK 4	Understanding Boolean Logic	20
	An Introduction to TKGate.....	20
WEEK 5	The DeMorgan's laws	24
WEEK 6	Design and Implementation of a logical circuit with 4 inputs.....	26
WEEK 7	Implementing Boolean Logic Equations	28
WEEK 8	Designing a Half-Adder to compute a 1-bit sum	30
	Design of a One-Bit Full-Adder:	32
	Implementing a Full-Adder with Half-Adders	32
WEEK 9	Understanding AND Gates	35
	Understanding OR Gates.....	38
	NOT Gates (Inverters).....	39
	NAND Gates	39
WEEK 10	Technological Advances in the Manufacture of Gates.....	40
WEEK 11	Understand the basic concepts of SSI, MSI, LSI, VLSI	41
WEEK 12	Designing a Transparent Latch.....	42
WEEK 13	Designing a D Flip-Flop	44
WEEK 14	Counters	46
WEEK 15	A Binary Counter	48

Week 1 Practical

Objectives:

1. Ability to develop formulas using Excel spread sheet to convert Binary numbers, into other number systems.
2. Convert from one code to another.

Using Excel to convert numbers to different number systems

A number system is a systematic way to represent numbers with symbolic characters and uses a base value to conveniently group numbers in compact form. The most common number system is decimal, which has a base value of 10, and a symbolic character set of 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. However, there are other number systems, and they can be more efficient to use for some specific purposes. For example, because computers use Boolean logic to perform calculations and operations, they use the binary number system, which has a base value of 2.

Microsoft Office Excel has several functions that you can use to convert numbers to and from the following number systems:

Number system	Base value	Symbolic character set
Binary	2	0,1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, and 9
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Convert a binary number to decimal

To do this task, use the **BIN2DEC function**.

Syntax

BIN2DEC(number)

Number is the binary number you want to convert. Number cannot contain more than 10 characters (10 bits). The most significant bit of number is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Remark

If number is not a valid binary number, or if number contains more than 10 characters (10 bits), BIN2DEC returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select cell A1, and type =BIN2DEC(1100100).

Result

Converts binary 1100100 to decimal (100)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the BIN2DEC function to binary 111111111

Result

Converts binary 111111111 to decimal (-1)

Practice with as many binary numbers as possible.

Convert a binary number to hexadecimal

To do this task, use the **BIN2HEX** function.

Syntax

BIN2HEX(number,places)

Number is the binary number you want to convert. Number cannot contain more than 10 characters (10 bits). The most significant bit of number is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

“Places” is the number of characters to use. If places is omitted, BIN2HEX uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

Remarks

- If number is not a valid binary number, or if number contains more than 10 characters (10 bits), BIN2HEX returns the #NUM! error value.
- If number is negative, BIN2HEX ignores places and returns a 10-character hexadecimal number.
- If BIN2HEX requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.
- If places is nonnumeric, BIN2HEX returns the #VALUE! error value.
- If places is negative, BIN2HEX returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select cell A1, and type =BIN2HEX (11111011, 4).

Result

Converts binary 11111011 to hexadecimal with 4 characters (00FB)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the BIN2HEX function to binary 1110

Result

Converts binary 1110 to hexadecimal (E)

Example 3

Using the above procedure convert binary 111111111 to Hexadecimal. The result will be Hexadecimal (FFFFFFF)

Convert a binary number to octal

To do this task, use the BIN2OCT function.

Syntax

BIN2OCT(number,places)

Number is the binary number you want to convert. Number cannot contain more than 10 characters (10 bits). The most significant bit of number is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places is the number of characters to use. If places is omitted, BIN2OCT uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

Remarks

- If number is not a valid binary number, or if number contains more than 10 characters (10 bits), BIN2OCT returns the #NUM! error value.
- If number is negative, BIN2OCT ignores places and returns a 10-character octal number.
- If BIN2OCT requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.
- If places is nonnumeric, BIN2OCT returns the #VALUE! error value.
- If places is negative, BIN2OCT returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select cell A1, and type =BIN2OCT (1001, 3).

Result

Converts binary 1001 to octal with 3 characters (011)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the BIN2OCT function to binary 1100100

Result

Converts binary 1100100 to octal (144)

Exercise:

1. Using the above procedure convert binary 111111111 to Octal
2. Convert Binary 1010 to Decimal.
3. Convert Binary 0000001 to Hexadecimal.

Week 2 Practical

Objectives:

1. Ability to develop formulas using Excel spread sheet to convert Decimal numbers, into other number systems.
2. Convert from one code to another.

Converts a decimal number to binary

To do this task, use the **DEC2BIN** function.

Syntax

DEC2BIN(number,places)

Number is the decimal integer you want to convert. If number is negative, **places** is ignored and DEC2BIN returns a 10-character (10-bit) binary number in which the most significant bit is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places is the number of characters to use. If **places** is omitted, DEC2BIN uses the minimum number of characters necessary. **Places** is useful for padding the return value with leading 0s (zeros).

Remarks

- If number < -512 or if number > 511, DEC2BIN returns the #NUM! error value.
- If number is nonnumeric, DEC2BIN returns the #VALUE! error value.
- If DEC2BIN requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.
- If places is nonnumeric, DEC2BIN returns the #VALUE! error value.
- If places is negative, DEC2BIN returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select a cell, and type =DEC2BIN (9, 4)

Result

Converts decimal 9 to binary with 4 characters (1001)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the DEC2BIN function to decimal -100.

Result

Converts decimal -100 to binary (1110011100)

Convert a decimal number to hexadecimal

To do this task, use the DEC2HEX function.

Syntax

DEC2HEX(number,places)

Number is the decimal integer you want to convert. If **number** is negative, **places** is ignored and DEC2HEX returns a 10-character (40-bit) hexadecimal number in which the most significant bit is the sign bit. The remaining 39 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places is the number of characters to use. If **places** is omitted, DEC2HEX uses the minimum number of characters necessary. **Places** is useful for padding the return value with leading 0s (zeros).

Remarks

- If number < -549,755,813,888 or if number > 549,755,813,887, DEC2HEX returns the #NUM! error value.
- If number is nonnumeric, DEC2HEX returns the #VALUE! error value.
- If DEC2HEX requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.
- If places is nonnumeric, DEC2HEX returns the #VALUE! error value.
- If places is negative, DEC2HEX returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select a cell, and type = DEC2HEX(100, 4)

Result

Converts decimal 100 to hexadecimal with 4 characters (0064)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the **DEC2HEX** function to decimal -54.

Result

Converts decimal -54 to hexadecimal (FFFFFFFFCA)

Convert a decimal number to octal

To do this task, use the DEC2OCT function.

Syntax

DEC2OCT(number, places)

Number is the decimal integer you want to convert. If **number** is negative, **places** is ignored and DEC2OCT returns a 10-character (30-bit) octal number in which the most significant bit is the sign bit. The remaining 29 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places is the number of characters to use. If **places** is omitted, DEC2OCT uses the minimum number of characters necessary. **Places** is useful for padding the return value with leading 0s (zeros).

Remarks

- If number < -536,870,912 or if number > 536,870,911, DEC2OCT returns the #NUM! error value.
- If number is nonnumeric, DEC2OCT returns the #VALUE! error value.
- If DEC2OCT requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.
- If places is nonnumeric, DEC2OCT returns the #VALUE! error value.
- If places is negative, DEC2OCT returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select a cell, and type = DEC2OCT(58, 3)

Result

Converts decimal 58 to octal (072)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the **DEC2OCT** function to decimal -100

Result

Converts decimal to octal (7777777634)

Exercise

1. Using MS Excel, convert the Decimal number 7 to binary with 4 characters.
2. Convert Decimal 7 to Octal with 4 places.

Week 3 Practical

Objectives:

1. Ability to develop formulas using Excel spread sheet to convert Hexadecimal numbers, into other number systems.
2. Convert from one code to another.

Convert a hexadecimal number to binary

To do this task, use the HEX2BIN function.

Syntax

HEX2BIN(number,places)

Number is the hexadecimal number you want to convert. Number cannot contain more than 10 characters. The most significant bit of number is the sign bit (40th bit from the right). The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places is the number of characters to use. If places is omitted, HEX2BIN uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

Remarks

- If number is negative, HEX2BIN ignores places and returns a 10-character binary number.
- If number is negative, it cannot be less than FFFFFFFE00, and if number is positive, it cannot be greater than 1FF.
- If number is not a valid hexadecimal number, HEX2BIN returns the #NUM! error value.
- If HEX2BIN requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.

- If places is nonnumeric, HEX2BIN returns the #VALUE! error value.
- If places is negative, HEX2BIN returns the #NUM! error value.

Example 1

3. Create a blank workbook or worksheet.
4. In the worksheet, select a cell, and type = HEX2BIN("F", 8)

Result

Converts hexadecimal F to binary, with 8 characters (00001111)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the HEX2BIN function to B7.

Result

Converts hexadecimal B7 to binary (10110111)

Exercise

Convert Hexadecimal FFFFFFFFFF to Binary

Convert a hexadecimal number to decimal

To do this task, use the HEX2DEC function.

Syntax

HEX2DEC(number)

Number is the hexadecimal number you want to convert. **Number** cannot contain more than 10 characters (40 bits). The most significant bit of number is the sign bit. The remaining 39 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Remark

If number is not a valid hexadecimal number, HEX2DEC returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select a cell, and type = HEX2DEC("A5")

Result

Converts hexadecimal A5 to decimal (165)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the HEX2DEC function to "FFFFFFFF5B".

Result

Converts hexadecimal FFFFFFFF5B to decimal (-165)

Exercise:

Insert a formula in a worksheet to convert the Hexadecimal number 3DA408B9 to Decimal.

Convert a hexadecimal number to octal

To do this task, use the HEX2OCT function.

Syntax

HEX2OCT(number,places)

Number is the hexadecimal number you want to convert. **Number** cannot contain more than 10 characters. The most significant bit of number is the sign bit. The remaining 39 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places is the number of characters to use. If **places** is omitted, HEX2OCT uses the minimum number of characters necessary. **Places** is useful for padding the return value with leading 0s (zeros).

Remarks

- If number is negative, HEX2OCT ignores places and returns a 10-character octal number.
- If number is negative, it cannot be less than FFE000000, and if number is positive, it cannot be greater than 1FFFFFFF.
- If number is not a valid hexadecimal number, HEX2OCT returns the #NUM! error value.
- If HEX2OCT requires more than places characters, it returns the #NUM! error value.
- If places is not an integer, it is truncated.
- If places is nonnumeric, HEX2OCT returns the #VALUE! error value.
- If places is negative, HEX2OCT returns the #NUM! error value.

Example 1

1. Create a blank workbook or worksheet.
2. In the worksheet, select a cell, and type =HEX2OCT("F", 3)

Result

Converts hexadecimal F to octal with 3 characters (017)

Example 2

Repeat steps 1 and two of example 1 but this time change the argument for the HEX2OCT function to "3B4E".

Result

Converts hexadecimal 3B4E to octal (35516)

Exercise:

- 1. What is the worksheet function for converting Octal numbers to:**
 - a. Binary**
 - b. Decimal**
 - c. Hexadecimal**

- 2. Convert the following Octal numbers to Binary**
 - a. 3**
 - b. 7777777**
 - c. Convert the Octal number above to a Binary number with three characters.**

Week 4 Practical

Objectives:

1. Introduction to the digital logic simulator TKGate
2. Introduction to Boolean logic and its relation to circuits
3. Design and implement Boolean logic equations

Understanding Boolean Logic

This lab exercise is intended to give you an introduction to logic, and its relation to digital circuits and logic gates. All computers are implemented using (a huge number of) logic gates. While we will not be trying to design a computer in this lab, you will first use the logic gate simulator program *TKGate* to simulate a logic circuit.

Logic has only two conditions: TRUE and FALSE. TRUE is often represented by the term HIGH or the number 1, and FALSE is represented by the term LOW or the number 0. Boolean algebra consists of a set of laws that govern logical relationships. Unlike ordinary algebra, where an unknown can take any numeric value, all elements of a boolean expression are either TRUE or FALSE.

An Introduction to TKGate

The following steps lead you through an introduction to TKGate.

Starting `tkgate`

1. Click on the TKGate button to start TKGate. This will bring up a window that looks like Figure1.

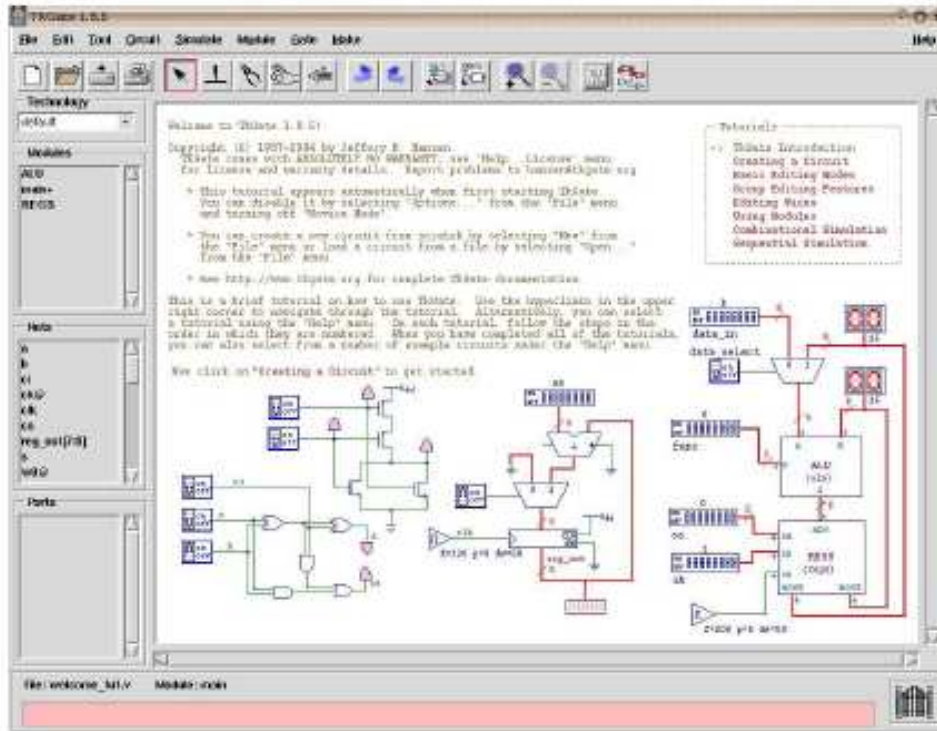


Figure 1: The TKGate window

2. Read the information that is being displayed (the TKGate introduction). Continue with the tutorial by clicking on **Creating a Circuit** in the **Tutorials** box on the right. Follow the instructions to create the simple circuit that TKGate will ask you to create. This will help you get used to the way in which gates and wires are drawn in TKGate.

Starting the Simulator

The simulator controls can be accessed either via the "Simulate" menu, or the button bar. Start a simulation by selecting "Begin Simulation" from the "Simulate" menu, or by pressing the "play" button on the button bar. A scope trace window will appear when you start the simulator, as well as text windows for any "tty" devices in your circuit. If there are any auto execute script files (see Simulation Scripts for details) these will be executed too.






The simulation will be performed with the designated root module at the top-level. The simulator internally expands any module instances in your circuit. Since the path you take to get to a module is significant to the simulator, you cannot jump directly to submodules but must "navigate" your way to them by selecting a module at the current level and by opening it using the menu or the '>' keyboard command. You can leave a module you are in with the '<' keyboard command.

Gate and wire names in sub-modules are referenced by prepending a dot-separated "path" of module instance names. For instance, suppose there are two instances of a module

named "foo" in the root module with instance names "g1" and "g2". Now suppose the "foo" module contains a wire named "w1". These wire names are referenced by the simulator as "g1.w1" and "g2.w1" to distinguish the two instances.

TkGate is an event-driven simulator. Time is measured in discrete units called "epochs". Each gate has a delay of a certain number of epochs. Some complex gates have multiple delay constants. In addition, some gates such as registers and memories have additional delay parameters which affect internal state changes.

The basic simulator commands are:

Function	Button	Description
Run		Enters continuous simulation mode. The simulation will continue as long as there are events in the event queue. If there are any clock gates in your circuit, this will mean the simulation will continue indefinitely. If the circuit is combinational, the simulation will continue until the circuit reaches quiescence.
Pause		Causes a continuously running simulation to stop.
Step Epoch		Causes the simulation to advance a fixed number of epochs. The number of epochs to advance can be set on the simulation options menu. You can also invoke this command with the spacebar.
Step Cycle		Causes the simulation to advance to the rising edge of a clock. You can set the number of clock cycles to simulate and the number of epochs past the designated cycle to step (to allow time for registers to change value). The default is to trigger on any clock, but you can designate a specific clock in the simulator options menu. You can also invoke this command with the tab key.
End Simulation		Causes the simulation to be terminated and all probes to be deleted.

Other simulator commands will be discussed in the following sections.

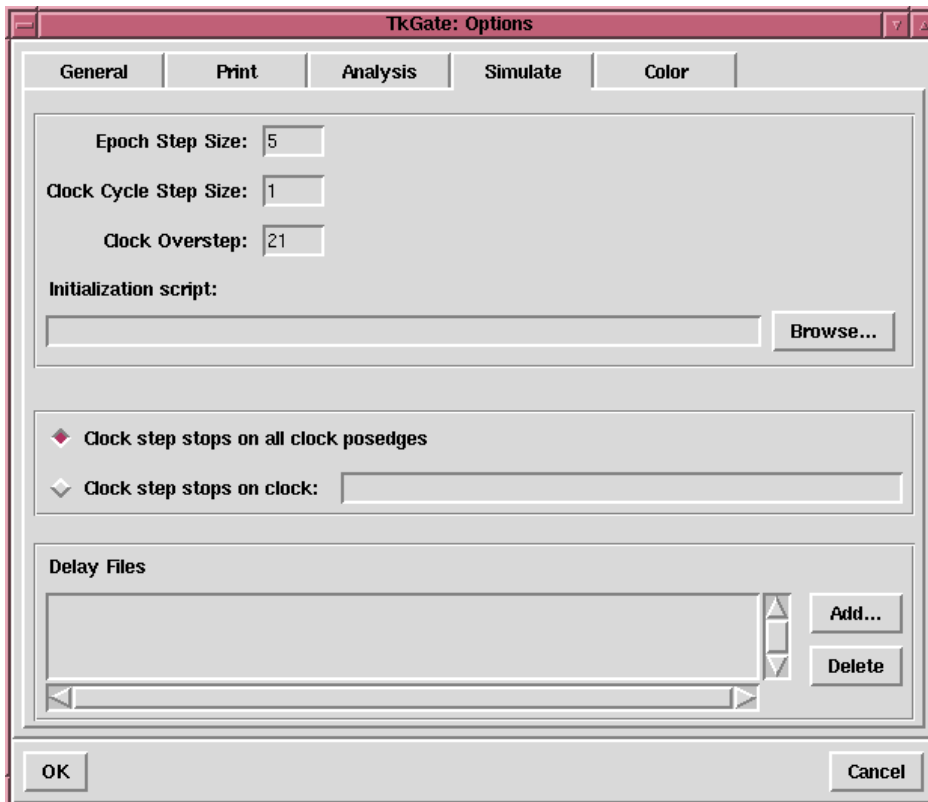
Simulator Options

Several simulation options can be set through the options dialog box. To edit the simulator options, select "Options..." from the "File" menu, and then select the "Simulator" tab from the tab box.

The simulator options are:

- **Epoch Step Size:** Specifies the number of epochs to step each time the simulator is stepped by using the step button or by pressing the space bar.

- **Clock Cycle Step Size:** Specifies the number of clock cycles to step each time the clock is stepped by using the clock step button or by pressing the tab key.
- **Clock Overstep:** Specifies the number of epochs to simulate past the clock edge when doing a clock step. This can be used to advance the simulator enough for registers to change value.
- **Initialization Script:** Specifies a simulation script to automatically execute when starting the simulator. The script file specified here is a global property and applies to any circuit that has been loaded into TkGate. To specify scripts specific to a particular circuit see circuit initialization scripts.
- **Clock step stops on all clock posedges:** Indicates that the clock step command should trigger on positive edges on all clocks in the circuit as.
- **Clock step stops on clock:** Indicates that the clock step command should trigger on positive edges only on the specified clock. This option is only useful for circuits with multiple clocks.
- **Delay Files:** Specifies additional files from which to load gate delay specifications. More on writing gate delay specifications can be found in Gate Delay Files



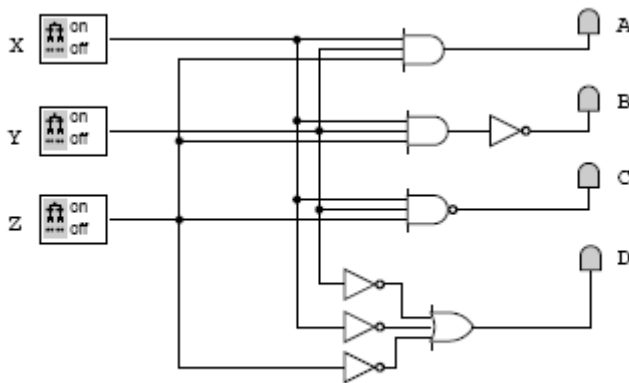
Week 5 Practical

Objectives:

1. Implement Boolean Logic Equations
2. Implement DeMorgan's Laws

The DeMorgan's laws

The instructor is to assist the students in designing the circuit shown in the figure below using TkGate or another digital simulator program.



In addition to the gates already discussed in class (the picture contains one OR gate, two AND gates, three NOT gates (inverters), and one NAND gate), this circuit contains two additional types of components:



Switches: When set to the `on` position, the output is high (i.e. true). When set to the `off` position, the output is low (i.e. false).



LEDs (Light Emitting Diodes): when the signal connected to the LED is HIGH, the LED glows (turns red on the screen). When the signal connected to the LED is LOW, the LED is off (pink on the screen).

1.2 To test this circuit, we need to activate the simulator. From the `Simulate` menu, choose `Begin Simulation`. Alternately, you can click the play icon in the toolbar at the top of the TKGate window.

Now from the `Simulate` menu, choose `Run`, or, alternately, click the play button once more. Now the power is on. You should see little AND gates marching along in the bottom right corner. This means we are in `Run` mode. The switches are sending a 0 signal. The two LEDs at the right should be pink, representing a 0 signal.

The purpose of this circuit is to demonstrate a slightly extended version of one of the DeMorgan's laws of logic that states that

$$\text{not } (X \text{ and } Y \text{ and } Z) \equiv (\text{not } X) \text{ or } (\text{not } Y) \text{ or } (\text{not } Z)$$

In the circuit, B and C both take on the value of

$\text{not } (X \text{ and } Y \text{ and } Z)$ while D is $(\text{not } X) \text{ or } (\text{not } Y) \text{ or } (\text{not } Z)$

Fill in the following truth table for each possible combination of inputs (use 0 to represent OFF, a pink LED; and use 1 for ON, a red LED). You can flip a switch from OFF to ON (or from ON to OFF) in the simulation by clicking on it.

Once you are done filling in the table go back to Edit mode. The `tkgate` logo, should appear in the lower right corner of the `tkgate` window.

X	Y	Z	B	C	D
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Truth table for lab1-1.v

Week 6 Practical

Objectives:

1. Design and implement Boolean logical equations.
2. Design and Implement a logical circuit with 4 inputs.



Design and Implementation of a logical circuit with 4 inputs

In this lab exercise you will design a circuit that will be used to choose between two values: its input will be the two logical values a and b, and two *controls* c1 and c2 that determine which of a and b it should choose. If c1 is true, then the circuit should output a AND b. If c1 and c2 are both false, then the circuit should output a OR b. Otherwise the circuit should output 1, regardless of the values of a and b. Design a circuit that achieves this goal on a piece of paper.

Now, implement your circuit using TKGate. Your circuit should have an LED at the output. The 4 inputs and the output should look like this:



You can now add the gates needed to achieve your design, and then connect them up. Recall that to create a gate in TKGate, you should first make sure that `tkgate` is in “move/connect” mode.

Now, left-click at the location where you want the gate; a  should appear where you clicked. Go to the Make menu, select the Gate submenu, and then select the gate you want to create. It should appear where the  was. You can left-click on the gate and adjust its position if necessary.

To wire your gates up, you should hold the left mouse button down on the inputs or outputs of the components until the soldering iron cursor appears and then drag to draw the wires. Let go of the mouse button near other wires to make a connection. To add comments to the diagram, for example to indicate which expressions are calculated by which gates, select Comment from the Make menu. Now, simulate your circuit, and verify that it works as expected. Once you believe that this is the case, show both your design on paper and your TKGate simulation to your instructor.

1.1 Exercise

The Figure below shows “chains” and “trees” of gates for five common gates.

1. Add the gates and connections for the NOR chain and XOR tree.
2. For which gates do the chain and tree compute the same result?
3. Which chains and/or trees compute the function (A and B) or (C and D)
4. Which of the chains and/or trees compute the function that is true if and only if an odd number of A, B, C, and D are true?

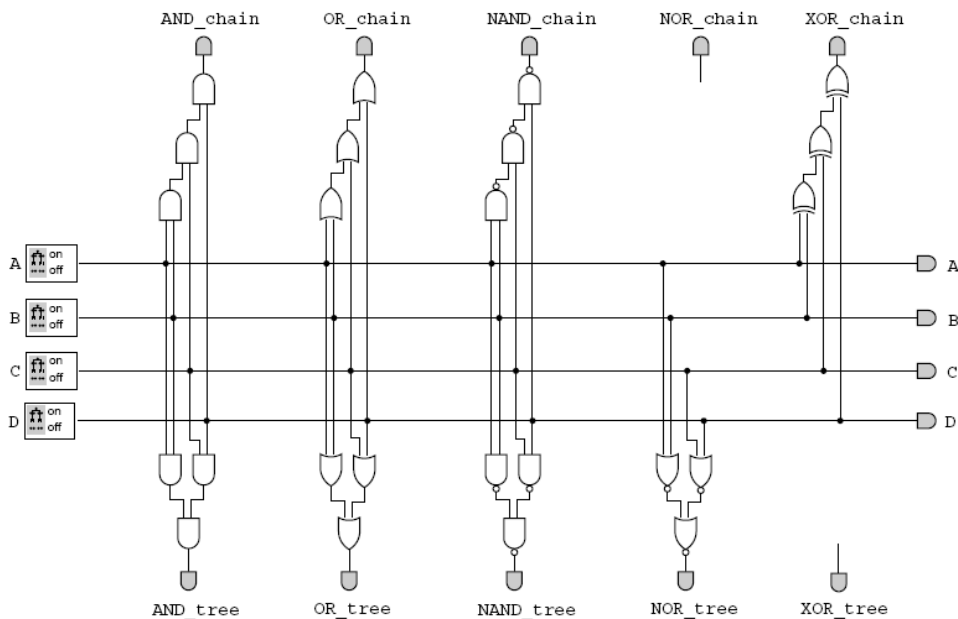


Figure 2: Another circuit

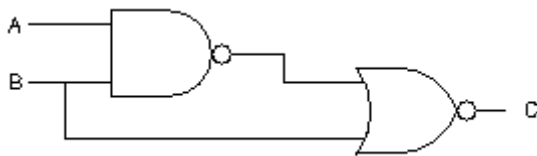
Week 7 Practical

Objectives:

1. Implementing Boolean Logic Equations
2. Compare between the outputs of logical circuits and logical equations

Implementing Boolean Logic Equations

In Digital Electronics, circuits can be designed from logical equations and vice versa. In this lab exercise, we will use Boolean algebra to simplify a logical equation and you are required to implement a circuit based on the equation using TKGate or any other digital simulator package. Test your design and compare your result with that of the Boolean simplification.



$$C = (B + \overline{(A * B)}) \quad [\text{given formula}]$$

$$C = \overline{B} * \overline{(A * B)} \quad [\text{DeMorgan } \overline{(A + B)} = \overline{A} * \overline{B}]$$

$$C = \overline{B} * (A * B) \quad [A = \overline{\overline{A}} \quad \{\text{double negative}\}]$$

$$C = (A * B) * \overline{B} \quad [A * B = B * A]$$

$$C = A * (B * \overline{B}) \quad [(A * B) * C = A * (B * C)]$$

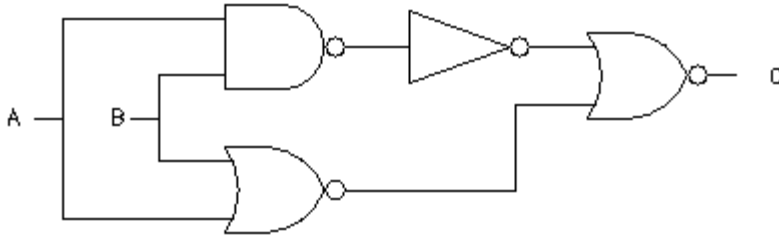
$$C = A * 0 \quad [A * \overline{A} = 0]$$

$$C = 0 \quad [0 * A = 0]$$

Using Boolean Algebra, the above formula is proven to always result in a logical 0 output. No matter what A and B are, C is always a logical 0.

Now simulate the circuit as shown above and see if it always give 0.

Exercise 2:



$$\begin{aligned}
 C &= \overline{\overline{(A * B)} + \overline{(A + B)}} && \text{[given formula]} \\
 C &= \overline{\overline{(A * B)} + \overline{(A + B)}} && \text{[A = A \{double negative\}]} \\
 C &= \overline{(A * B) * \overline{(A + B)}} && \text{[DeMorgan } \overline{(A + B)} = \overline{A} * \overline{B}\text{]} \\
 C &= \overline{(A * B) * (A + B)} && \text{[A = A \{double negative\}]} \\
 C &= \overline{(\overline{A} + \overline{B}) * (A + B)} && \text{[DeMorgan } \overline{(A * B)} = \overline{A} + \overline{B}\text{]} \\
 C &= ((\overline{A} + \overline{B}) * A) + ((\overline{A} + \overline{B}) * B) && \text{[A * (B + C) = (A * B) + (A * C)]} \\
 C &= (A * (\overline{A} + \overline{B})) + (B * (\overline{A} + \overline{B})) && \text{[A * B = B * A]} \\
 C &= ((A * \overline{A}) + (A * \overline{B})) + ((B * \overline{A}) + (B * \overline{B})) && \text{[A * (B + C) = (A * B) + (A * C)]} \\
 C &= ((0 + (A * \overline{B})) + ((B * \overline{A}) + 0)) && \text{[A * } \overline{A} = 0\text{]} \\
 C &= (A * \overline{B}) + (B * \overline{A}) && \text{[0 + A = A]}
 \end{aligned}$$

This formula indicates that C is true if A is true and B is false, or when B is true and A is false. This function is called the XOR (exclusive OR). Remember that in an OR gate, one input true will create a true output, and that if both inputs are true the output is still true. The XOR function will give a false output if both inputs are true as well as both false, and will only give a true output when the inputs are different.

Now simulate the circuit as shown above and compare the result.

Week 8 Practical

Objectives:

Implement the Half-Adder and the Full-Adder using a logic simulator software (TKgate).

Designing a Half-Adder to compute a 1-bit sum

Let us now consider the design for a simple adder that takes two bits, x and y as inputs and produces two outputs, **sum** and **carry**. For example, if the inputs are 1 and 1, the sum is 2, or in binary 10. Since our adder only creates a one bit sum, that's a sum of 0 with a carry of 1. The decimal equivalent would be a one digit adder that might take inputs like 5 and 8 to produce a sum of 3 and a carry of 1 (in other words, 13).

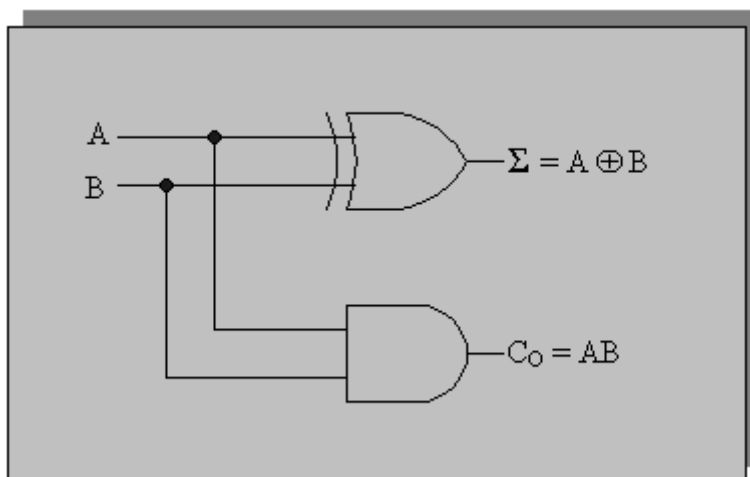
Start by filling in the table below for computing **sumBit** and **carryBit** from x and y . Determine what logic gates match these truth tables (treating 1 as true and 0 as false). Then, draw a circuit that takes x and y as inputs and produces the **sumBit** and **carryBit** bits as outputs.

x	y	sumBit	carryBit
0	0	_____	_____
0	1	_____	_____
1	0	_____	_____
1	1	_____	_____

Your circuit is commonly called a “half-adder.”

In order to implement the above truth table, i.e. one that returns a sum bit of 1 only when the two inputs are not the same, and a 0 when they are, an XOR gate is required. This will work perfectly for the first three entries of the truth table because there are no carry bits. As for the fourth entry, which happens to be a 0 sum bit with a carry bit of 1, an XOR gate alone cannot achieve that until an AND gate is added to take care of the carry bit.

This is illustrated in the figure below;



Now, start TKGate and design the circuit above.

Depending on the order in which you make the connections, TKGate might rename some of the input and output ports to names like w0, or w1, etc. Check the names of the ports – they should be x, y, sum, and carry. If TKGate changed any of them, select the altered port with a left mouse-click, then get the “Properties” menu by holding down the right mouse button over the port, selecting properties, and releasing the button. You can fill-in the correct name in the “Net name” entry and then click “OK.”

Now, you can simulate your circuit. It is a good idea to save your work at this point before you start the simulator. Run the simulator and verify that your half-adder works.

Design of a One-Bit Full-Adder:

The one-bit half-adder is great if all you want to do is add 0 or 1 to 0 or 1. However, most situations require working with larger numbers. We could make a bigger addition table, but a 1-billion by 1-billion addition table is rather impractical. Instead, we will take another look at the way we learned to add in elementary school: one digit at a time. We can build hardware that works on the same principle.

The half-adder that you designed in the last Lab exercise is a good start. However, it only adds the bits from x and y, and does not take into account any carry generated by the previous bit. Thus, we need an adder that takes three input bits: one for a bit from x, one for a bit from y, and one for the carry from the preceding bit. A circuit that does this is called a *full-adder*. We will describe two ways to build a full-adder. In this exercise we will build a full-adder out of half-adders.

Implementing a Full-Adder with Half-Adders

Suppose that we have combined a bit from x with a bit from y using the half-adder that you designed in the last Lab exercise. Now, we have an input carry, so the situation looks like:

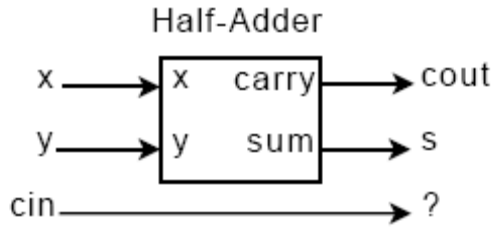


Figure 1: A half-adder with an extra input carry.

where cin is the input carry, and cout is the output carry. What should we do with cin? From the pencil and paper method for adding, we know that we should add the input carry to our result. But how should we do this? Consider the following computation:

$$\begin{array}{r}
 x \\
 + y \\
 \hline
 \text{cout} s \\
 + \text{cin} \\
 \hline
 ? ?
 \end{array}$$

If the sum $s + \text{cin}$ is 0 or 1, then everything is fine, and we can compute the answer using two half-adders:

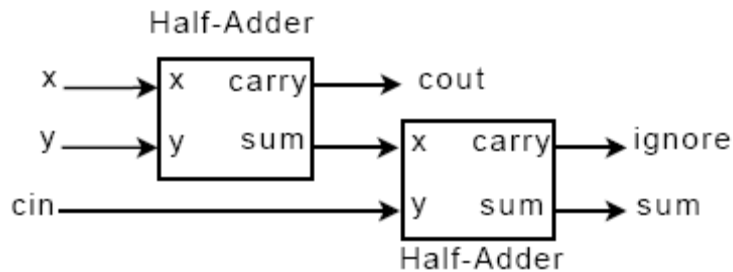


Figure 2: two half-adders to (almost) add two bits and a carry.

But what if the sum $s + \text{cin}$ is 2? In this case, it will generate its own carry (the bit labeled ignore in the figure), which needs to be added to the carry cout from the first half-adder (since both of them represent the value 2). This suggests that we use one more half-adder as follows:

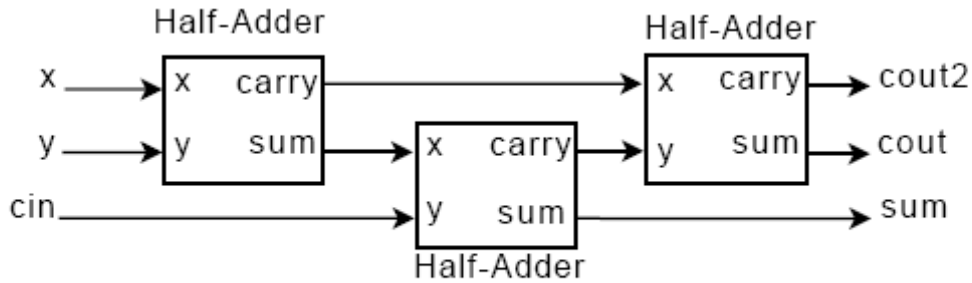
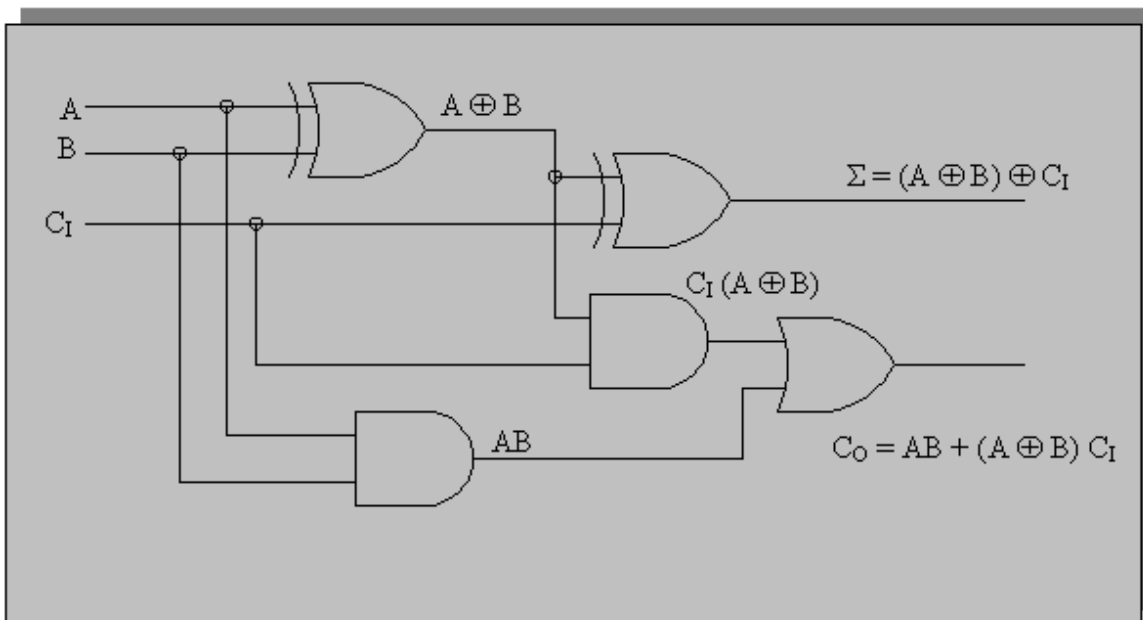


Figure 3: three half-adders to add two bits and a carry.

What should we do with the additional carry cout2? Nothing, since as we are only adding three bits together, the largest possible sum is 3, so that last carry will always be 0. This means that we can ignore cout2 and we do not need the part of the rightmost half-adder in Figure 3 that produces cout2. Draw the part of that last half-adder that produces cout from x and y.



Logic Circuit for the full-adder

Week 9 Practical

Objectives:

To understand the characteristics of various logical gates.

Understanding AND Gates

If we think of two signals, A and B, as representing a **truth value** of two different propositions, then A could be either TRUE (a logical 1) or FALSE (a logical 0). B can take on the same values. Now consider a situation in which the output, C, is TRUE only when both A is TRUE and B is TRUE. We can construct a **truth table** for this situation. In that truth table, we insert all of the possible combinations of inputs, A and B, and for every combination of A and B we list the output, C.

A	B	C
False	False	False
False	True	False
True	False	False
True	True	True

An AND Example

Let's imagine a physician prescribing two drugs. For some conditions drug A is prescribed, and for other conditions drug B is prescribed. Taken separately each drug is safe. When used together dangerous side effects are produced.

Let

- A = Truth of the statement "Drug 'A' is prescribed."
- B = Truth of the statement "Drug 'B' is prescribed."
- C = Truth of the statement "The patient is in danger."

Then, the **truth table** below shows when the patient is in danger.

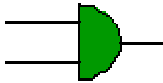
A	B	C
False	False	False
False	True	False

True	False	False
True	True	True

Notice that C is TRUE when both A AND B are true and only then!

AND GATES

An AND function can be implemented electrically using a device known as an AND gate. You might imagine a system in which zero (0) is represented by zero (0) volts, and one (1) is represented by three (3) volts, for example. If we are going to use electrical devices we need some sort of symbolic representation. There is a standard symbol for an AND gate shown below.

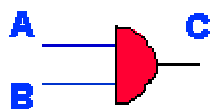


Often in lab work it's helpful to use an LED to show when a signal is 0 or 1. Usually a 1 is indicated with an LED that is ON (i.e. glowing).

- To get a logical zero, connect the input of the gate to ground to have zero (0) volts input.
- To get a logical one, connect the input of the gate to a five (5) volts source to have five volts at the input.
- Each button controls one switch (two buttons - two switches) so that you can control the individual inputs to the gate.
- Each time you click a button, you toggle the switch to the opposite position.

Exercise

1. You have an AND gate. Both inputs are zero. What is the output?
2. Assume you have an AND gate with two inputs, A and B. Determine the output, C, for the following cases.



P1. A = 1, B = 0

P2. A = 0, B = 1

P3. If either input is zero, what is the output?

P4. $A = 1, B = 1$

Understanding OR Gates

Consider a case where a pressure can be high and a temperature can be high. Let's assume we have two sensors that measure temperature and pressure. The first sensor has an output, T, that is **1** when a temperature in a boiler is too high, and **0** otherwise. The second sensor produces an output, P, that is **1** when the pressure is too high, and **0** otherwise. Now, for the boiler, we have a dangerous situation when either the temperature or the pressure is too high. It only takes one. Let's construct a truth table for this situation. The output, D, is **1** when danger exists.

T	P	D
False	False	False
False	True	True
True	False	True
True	True	True

What we have done is defined an **OR** gate. An OR gate is a gate for which the output is **1** whenever one or more of the inputs is **1**. The output of an OR gate is **0** only when all inputs are **0**. Shown below is a schematic symbol for an OR gate, together with the simulated LEDs and input buttons so that you can explore OR gate behavior.

Exercise

Assume you have an OR gate with two inputs, A and B. Determine the output, C, for the following cases.

1. A = 1, B = 0
2. A = 0, B = 1
3. If either input is one, what is the output?

NOT Gates (Inverters)

A third important logical element is the inverter. An inverter does pretty much what it says. If the input is **0**, the output is **1**. Conversely, if the input is **1**, the output is **0**.

Exercise

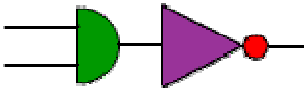
You need to control two pumps that supply two different concentrations of reactant to a chemical process. The strong reactant is used when pH is very far from the desired value, and the weak reactant when pH is close to desired.

You need to ensure that only one of the two pumps runs at any time. Each pump controller responds to standard logic signals, that is when the input to the pump controller is 1, the pump operates, and when that input is 0, the pump does not operate.

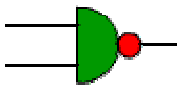
Design a circuit to operate this process using a NOT gate.

NAND Gates

There is another important kind of gate, the **NAND** gate. Actually, the way to start thinking about a NAND gate is to think of it as an AND gate with an inverter on the output. That's shown below.



Actually, however, the symbol for a NAND gate compresses the inverter down to a dot at the output of the NAND gate as shown below.



Week 10 Practical

Objectives:

To trace the Technological Advances in the Manufacture of Gates

Technological Advances in the Manufacture of Gates

In electronics, an integrated circuit (also known as IC, microcircuit, microchip, silicon chip, or chip) is a miniaturized electronic circuit (consisting mainly of semiconductor devices, as well as passive components) that has been manufactured in the surface of a thin substrate of semiconductor material. Integrated circuits are used in almost all electronic equipment in use today and have revolutionized the world of electronics.

1. Conduct research on the Internet to investigate the sequence of events that led to the manufacture of the first micro chip by Jack Kilby of Texas Instruments. Your write up should include;

- 1.1 Vacuum Tube Computers
- 1.2 Transistor-based Computer
- 1.3 Comparison between Vacuum Tube computers and Transistor-based computers.
- 1.4 The drawbacks of using Vacuum Tubes and Transistors in computer architectural design.

Week 11 Practical

Objectives:

Understand the basic concepts of Small Scale Integration (SSI), Medium Scale Integration (MSI), Large Scale Integration (LSI), and Very Large Scale Integration (VLSI).

Understand the basic concepts of SSI, MSI, LSI, VLSI

Discuss Integrated Circuits under the following headings

1. Small Scale Integration (SSI)
2. Medium Scale Integration (MSI)
3. Large Scale Integration (LSI)
4. Very Large Scale Integration (VLSI)

Week 12 Practical

Objectives:

1. Implement sequential circuits
2. Design and test a transparent latch

This lab introduces circuits that can perform sequences of steps and remember information from previous steps. These circuits are basically made of **flip-flops** that can remember values from one step to the next.

Designing a Transparent Latch

Figure 1 shows the first circuit that we will consider.

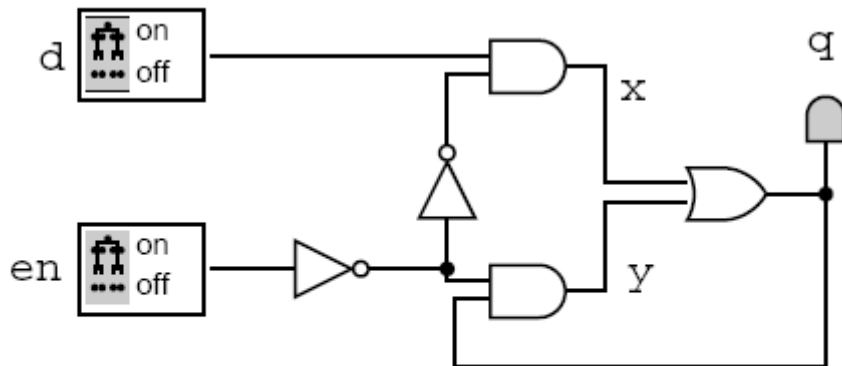


Figure 1: A Transparent Latch

When we write down the logic equations for this circuit, we get:

$$x = d \text{ AND } en$$

$$y = q \text{ AND } _en$$

$$q = x _ y$$

This acts as a two-input multiplexer with output q that selects d when en is true and selects q when en is false. Writing out these two cases we get:

Case en :

$$x = d$$

$$y = F$$

$$q = x$$

Substituting $x = d$ into $q = x$ we get $q = d$. In other words, when en is true, the output q matches the data input d . The other case is:

Case $_en$:

$x = F$
 $y = q$
 $q = y$

The equations for q and y give us two equations, $y = q$ and $q = y$ in two unknowns, q and y . Because q and y are Boolean valued, each must be either true or false.¹ We note that $q = y = T$ and $q = y = F$ are both solutions.

How does the circuit know which solution to choose? When en is true, q takes on the value of d . When en is false, q retains whatever value it had when en went from true to false. Thus, q “remembers” the value that d had when en was true. While en is false, q retains its value and is unaffected by changes of d . This circuit is called a transparent latch.

Start up TKGate and design the circuit in figure 1. Then verify the operation of the latch. For example, change d when en is low, and then again when en is high. Show that you can store both high and low values of d within the latch.

You may observe that if you start the simulator with the switch for en in the off (i.e. low) position, then the LED for q is yellow. This indicates that the value for q is “undefined;” in other words, TKGate can’t figure out if q should be high or low. When you set en high, q will take on the value of d and will no longer be undefined.

Week 13 Practical

Objectives:

1. Implement sequential circuits
2. Design and Test a D-Flip Flop

Designing a D Flip-Flop

Now, let us look at an extension of the previous idea: the D Flip-Flop. Flip-flops are circuits that combine a pair of latches along with circuitry to generate a clock (a signal that is first high, then low, then high, then low, then high, then low, etc). Figure 2 shows the circuit that is known as a D flip-flop.

In the circuit in Figure 2, the pair of NOR gates and the inverter at the bottom of the diagram produce two control signals, such that **phase-1** is low whenever **phase-2** is high, and the other way around. These two signals come from a single input **clk**. The **clk** input is referred to as the “clock.” It is the clock that tells the flip-flops to update their outputs. In other words, with each clock event, the hardware moves to the next step of its computation. This is the same clock that appears on advertisements for computers. For example, if a computer has a 3.2GHz clock, that means that it the hardware performs 3.2 billion of these basic steps per second.

Note that some of the gates at the bottom of the figure are needed to make sure the signals go low or high at exactly the right time.

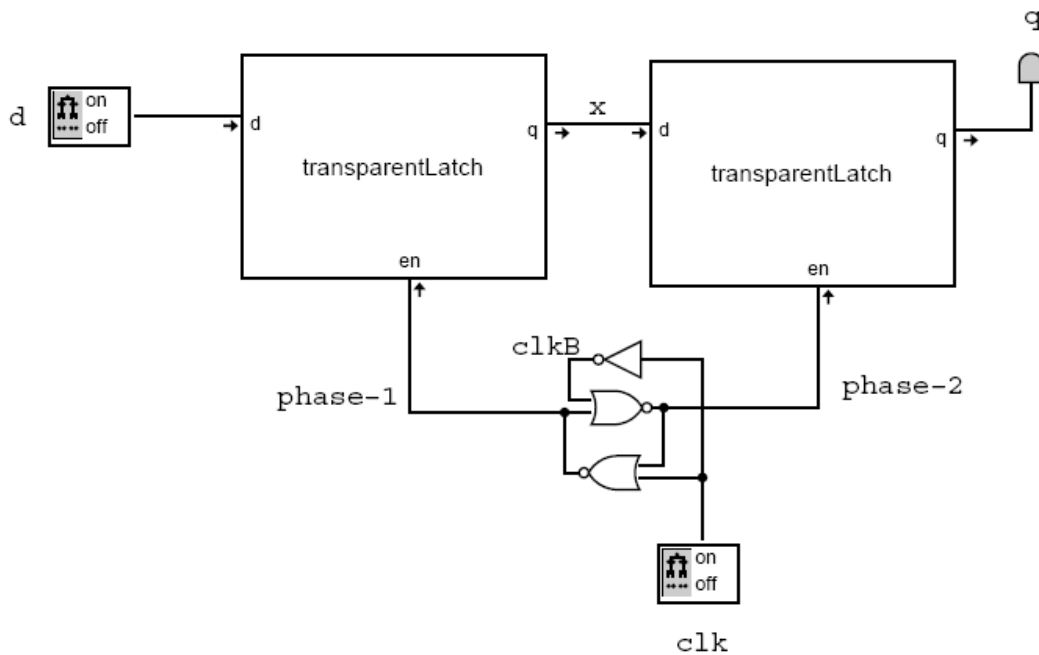


Figure 2: A D Flip-Flop

Consider the situation when `clk` is low. In this case, `clkB` is high; therefore, `phase-2` is low; and finally, `phase-1` is high. This means that the value on `d` is transferred to node `x`. When `clk` goes high, `clkB` and `phase-1` go low. This in turn causes `phase-2` to go high. Likewise, when `clk` goes low, `clkB` will go high; `phase-2` will go low; and `phase-1` will go high.

The combined effect of the two latches is that the value of `d` is copied to the `q` when the clock makes a low-to-high transition. At all other time, `q` retains its value, independent of any changes to `d`. This circuit is called a positive-edge-triggered D flip-flop. The term “positive-edge-triggered” refers to the property that `q` changes when the clock goes high (i.e. on the rising edge of the clock). It’s called a “D flip-flop” because the value of `q` is determined by the “data” input (i.e. `d`).

Now, simulate this circuit and verify that whenever the `clk` signal goes from low to high, the `q` output takes on the current value of `d`. Show that you can change `d` at other times, whether the clock is high or low, without affecting `q`.

WEEK 14 PRACTICAL

Objectives:

1. Ability to design counter circuits

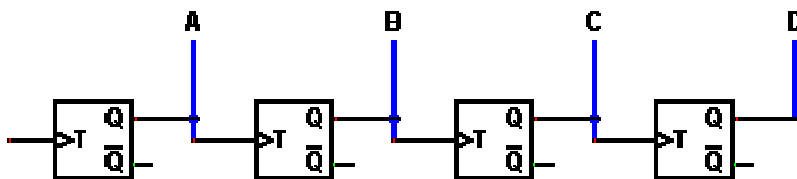
Counters

One common requirement in digital circuits is *counting*, both forward and backward. Digital clocks and watches are everywhere, timers are found in a range of appliances from microwave ovens to VCRs, and counters for other reasons are found in everything from automobiles to test equipment.

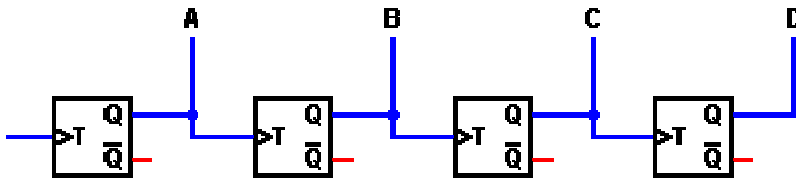
Although we will see many variations on the basic counter, they are all fundamentally very similar. The demonstration below shows the most basic kind of binary counting circuit.

In the 4-bit counter below, we are using edge-triggered master-slave flip-flops. The output of each flip-flop changes state on the falling edge (1-to-0 transition) of the T input.

The count held by this counter is read in the reverse order from the order in which the flip-flops are triggered. Thus, output D is the high order of the count, while output A is the low order. The binary count held by the counter is then DCBA, and runs from 0000 (decimal 0) to 1111 (decimal 15). The next clock pulse will cause the counter to try to increment to 10000 (decimal 16). However, that 1 bit is not held by any flip-flop and is therefore lost. As a result, the counter actually reverts to 0000, and the count begins again.



Use a different input scheme, as shown in the figure below. Instead of changing the state of the input clock with each click, you will send one complete clock pulse to the counter when you click the input button. The button image will reflect the state of the clock pulse, and the counter image will be updated at the end of the pulse. For a clear view without taking excessive time, each clock pulse has a duration or pulse width of 300 ms (0.3 second). The demonstration system will ignore any clicks that occur within the duration of the pulse.



A major problem with the counters shown here is that the individual flip-flops do not all change state at the same time. Rather, each flip-flop is used to trigger the next one in the series. Thus, in switching from all 1s (count = 15) to all 0s (count wraps back to 0), we don't see a smooth transition. Instead, output A falls first, changing the apparent count to 14. This triggers output B to fall, changing the apparent count to 12. This in turn triggers output C, which leaves a count of 8 while triggering output D to fall. This last action finally leaves us with the correct output count of zero. We say that the change of state "ripples" through the counter from one flip-flop to the next. Therefore, this circuit is known as a "ripple counter."

This causes no problem if the output is only to be read by human eyes; the ripple effect is too fast for us to see it. However, if the count is to be used as a selector by other digital circuits (such as a multiplexer or demultiplexer), the ripple effect can easily allow signals to get mixed together in an undesirable fashion. To prevent this, we need to devise a method of causing all of the flip-flops to change state at the same moment. That would be known as a "synchronous counter" because the flip-flops would be synchronized to operate in unison.

Week 15 Practical

Objectives:

Ability to design counter circuits

A Binary Counter

Using the basic building blocks of gates and flip-flops, we can build every digital function. In this exercise, we will be designing a four bit counter. The Figure below shows its design:

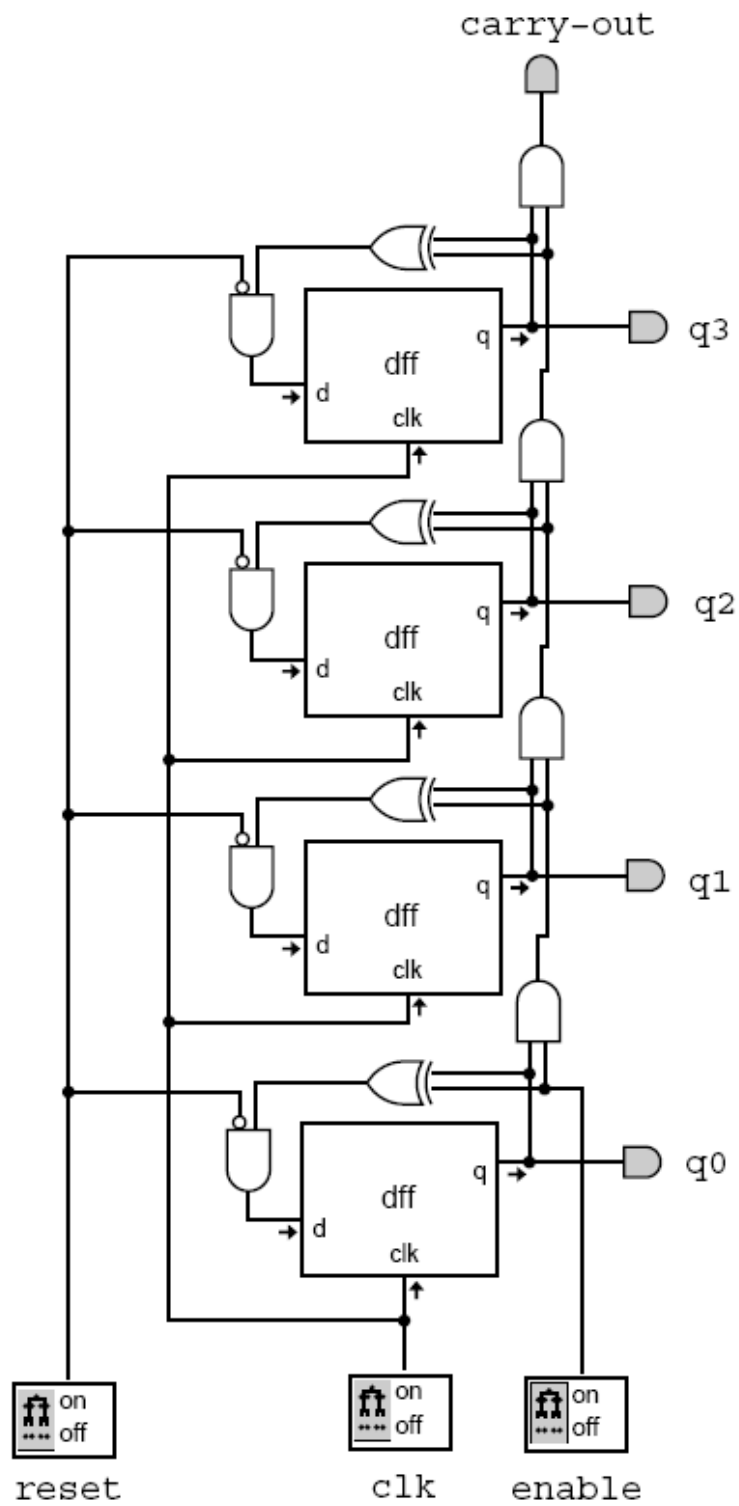


Figure 4: A Four-Bit, Binary Counter

As with the toggle element, the **d** inputs of all of the flip-flops are forced low when **reset** is high. Note that each **AND** gate on the right has an output that is true if-and-only-if the **q** outputs for all flip-flops below the gate are true. Thus, each **XOR** gate inverts the **q** output if-and-only-if all of the **q** outputs below that stage are true. This is exactly the situation when a carry is generated by the lower bits when adding one to the current value.

Implement this module and start the simulator. Set the **reset** signal to high. Toggle the switch for the **clk** signal and all of the **q** outputs should go low. Now, set the **reset** signal to low. Toggle the switch for the **clk** signal several times and observe how the **q** signals count in binary.