Pearson BTEC Level 4 Higher Nationals in Engineering (RQF)

Unit 20: Digital Principles

Unit Workbook 1 of 1

1 INTRODUCTION

This Workbook guides you through the learning outcomes related to:

LO1. Explain and analyse simple combinational logic circuits.

Concepts of combinational logic:

Simple logic circuits implemented with electro-mechanical switches and transistors. Circuits built from AND, OR, NAND, NOR, XOR gates to achieve logic functions, e.g. majority voting, simple logical controls, adders.

Number systems, and binary arithmetic:

Binary, Decimal, Hexadecimal number representation, converting between, applications and relative advantages. Addition and subtraction in binary, range of n-bit numbers.

Analysis of logic circuits:

Truth Tables, Boolean Algebra, de Morgan's theorem, Karnaugh Maps.

Simplification and optimisation of circuits using these techniques.

LO2. Explain and analyse simple sequential logic circuits.

Sequential logic elements and circuits:

SR latch built from NAND or NOR gates.

Clocked and edge-triggered bistables, D and JK types.

Simple sequential circuits, including shift registers and counters.

Timing Diagrams.

Memory technologies:

Memory terminology, overview of memory technologies including Static RAM, Dynamic RAM and Flash memory cells.

Relative advantages in terms of density, volatility and power consumption.

Typical applications, e.g. in memory stick, mobile phone, laptop.

LO3. Describe and evaluate the technologies used to implement digital electronic circuits.

Logic values represented by voltages:

Sequential logic elements and circuits:

The benefit of digital representation of information.

The concept of logic input and output values and thresholds.

Digital technologies:

Introduction to discrete logic families, CMOS and TTL, relative advantages in terms of speed, power consumption, density.

Programmable logic, FPGAs, relative advantages and applications.

LO4. Describe and analyse a range of digital subsystems, hence establishing the building blocks for larger systems.

User interface:

Examples to include switches, light emitting diodes and simple displays.

Digital subsystems:

Examples to be drawn from adders (half, full, n-bit), multiplexers and demultiplexers, coders and decoders, counters applied as timers, shift registers applied to serial data transmission, elements of the ALU (Arithmetic Logic Unit). Emphasis on how these can be applied, and how they might fit into a larger system.

IMPORTANT NOTE:

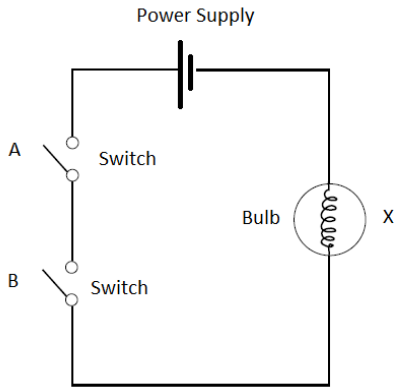
It is recommended that in conjunction with this Workbook, that you also read "round your subject". This means accessing and reading the references and other material given in this workbook as well undertaking your own research.

It is also recommended that you consult the following, which are related units:
Unit 19: Electrical and Electronic Principles
Unit 22: Electronic Circuits and Devices
Unit 52: Further Electrical, Electronic and Digital Principles

2 CONCEPTS OF COMBINATIONAL LOGIC

2.1 Simple Logic Circuits.

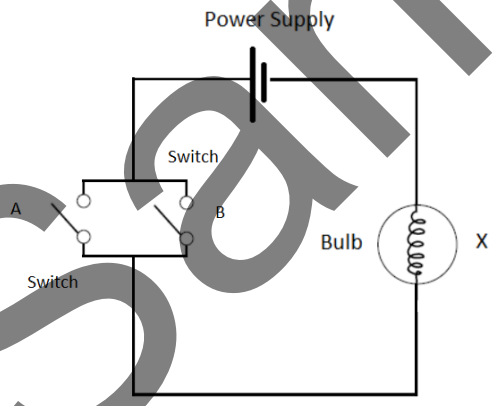
Consider a simple electrical circuit consisting of a battery, a bulb, and two switches connected in series. The switches can be considered as inputs to the circuit and the bulb as the output. A truth table is a convenient method of describing the operation of the circuit (See Figure 1).



A	B	X
Open	Open	Off
Open	Closed	Off
Closed	Open	Off
Closed	Closed	On

Figure 1 Switch representation of a 2-input AND function and its associated Truth Table

As the bulb is only lit (ON) when *both* the A and B switches are Closed (ON), then this circuit can be said to perform a 2-input AND function. In fact, the results depend on the way in which the switches are connected; consider another circuit in which the two switches are connected in parallel (Figure 2).



A	B	X
Open	Open	Off
Open	Closed	On
Closed	Open	On
Closed	Closed	On

Figure 2 Switch representation of a 2-input OR function and its associated Truth Table

In this case, as the light is ON when either A or B are Closed (ON), this circuit can be said to provide a 2-input OR function.

These circuits are performing simple logical functions, which can also be implemented in many other ways (transistor, pneumatic, hydraulic, and magnetic devices for example, where ON and OFF are represented by different mechanisms)

Thus, it is possible to specify a particular logic function without necessarily specifying its final physical realisation. To facilitate this, special symbols are employed to represent logic functions, and the associated

truth table assignments are specified using more abstract terms, specifically; TRUE and FALSE. The representation of the logic function is usually called a GATE.

2.2 NOT Gate

The simplest of all Logic Gates is the NOT Gate. It's Symbol, Truth Table, and Timing diagram representation is shown in Figure 3. The small circle on the output of the gate represents an inverting function.

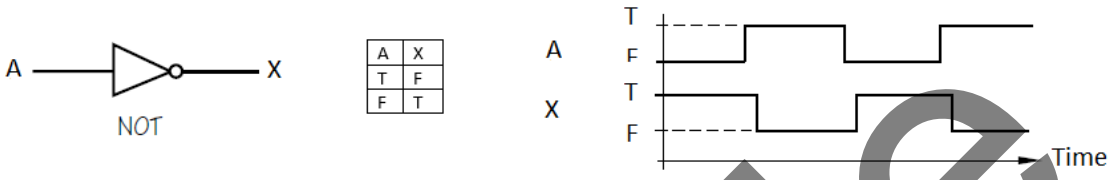


Figure 3 Logic NOT representation:
Symbol, Truth Table and Timing diagram representation

As a reminder that these abstract functions will eventually have physical realisations, the timing diagram waveforms shows a delay between transitions on the inputs and corresponding responses at the outputs. The actual values of these delays depend on the technology used to implement the functions, but it is important to note that in any physical implementation there will always be some element of delay.

2.3 AND, OR and XOR Gates

Three slightly more complex functions are known as AND, OR, and XOR. These are shown in Figures xxxx, below.

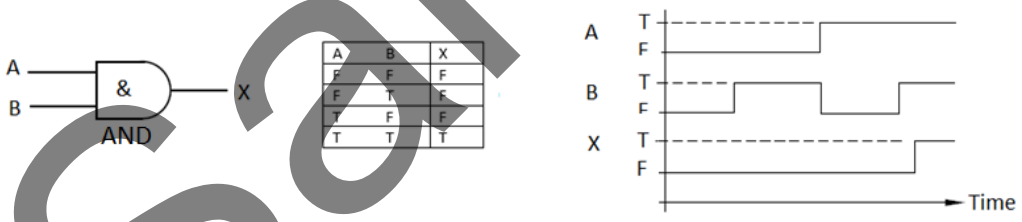


Figure 4 Logic AND representation:
Symbol, Truth Table and Timing diagram representation

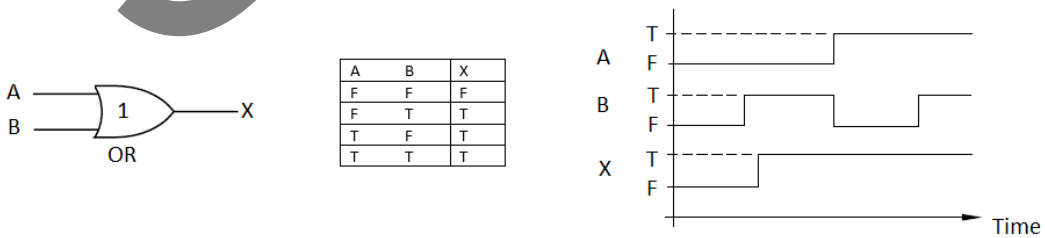


Figure 5 Logic OR representation:
Symbol, Truth Table and Timing diagram representation

3 NUMBER SYSTEMS

3.1 Decimal and Binary number systems

Almost certainly because humans have ten fingers, the decimal, or base-10 numbering system has been adopted universally. The value of particular digit in a decimal number depends on the **value** of the digit and its **position** within the number. The decimal system is, therefore, a **place-value** system.

Many other numbering systems are used for varying purposes, but it is not surprising that the logic systems that we introduced in the Chapter 4 infer a Binary, or Base-2 numbering system, the relevant values of the digits used being 0 (representing False, for example) and 1 (representing True, for example).

Each column in a binary number has a weight derived from the base, and each digit is combined with its column's weight to determine the final value of the number. This is shown in Figure 7.

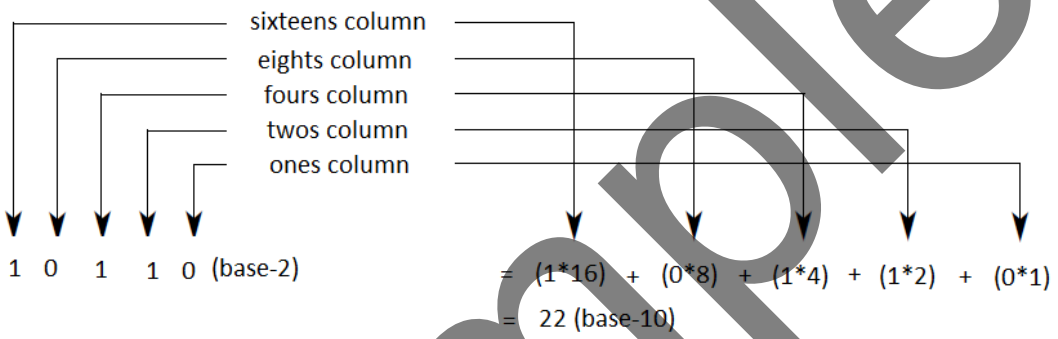


Figure 7 Combining digits with column weights in binary

The 1 or 0 in a binary number is a binary digit, which neatly contracts to become **bit**. The binary value 10110_2 is said to be 5 bits wide.

A group of 4 bits is known as a nybble (nibble) and a group of 8 bits is known as a byte.

Did you know that there are only ten types of people in the world; those that understand binary and those that don't! (Think about it!)

3.2 Octal (Base-8) and Hexadecimal (Base-16)

Any number system having a base that is a power of two (2, 4, 8, 16, 32, etc.) can be easily mapped into its binary equivalent and vice versa. For this reason, electronics engineers typically make use of either the octal (base-8) or hexadecimal (base-16) systems.

As a base-8 system, octal requires eight individual symbols to represent all of its digits. This isn't a problem because we can simply use the symbols 0 to 7. However, as a base-16 system, hexadecimal requires sixteen individual symbols to represent all of its digits; (0 to 9) and (A to F).

The rules for counting in octal and hexadecimal are the same as for any other place-value system – when all the digits in a column are exhausted, the next count sets that column to zero and increments the column to the left (see Figure 8).

Decimal	Binary	Octal	Hexadecimal
0	00000000	000	00
1	00000001	001	01
2	00000010	002	02
3	00000011	003	03
4	00000100	004	04
5	00000101	005	05
6	00000110	006	06
7	00000111	007	07
8	00001000	010	08
9	00001001	011	09
10	00001010	012	0A
11	00001011	013	0B
12	00001100	014	0C
13	00001101	015	0D
13	00001110	016	0E
15	00001111	017	0F
16	00010000	020	10
17	00010001	021	11
18	00010010	022	12
:	:	:	:
Etc.	Etc.	Etc.	Etc.

Figure 8 Counting in Octal and Hexadecimal

Although not strictly necessary, binary, octal and hexadecimal numbers are often prefixed by leading zeros to pad them to whatever width is required. This padding is usually performed to give some indication as to the physical number of bits used to represent that value within a computer. Each octal digit can be directly mapped onto three binary digits, and each hexadecimal digit can be directly mapped onto four binary digits (Figure 9).

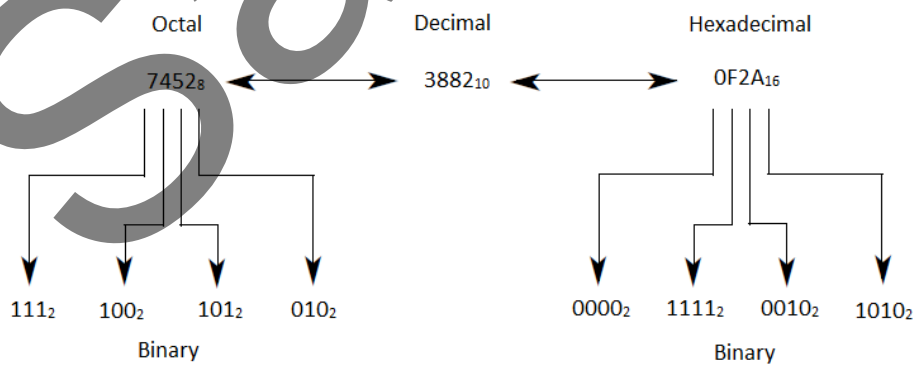


Figure 9 Mapping Octal and Hexadecimal to Binary

In the early digital computers, data paths were often 9 bits, 12 bits, 18 bits, or 24 bits wide, which explains the original popularity of octal notation. Due to the fact that each octal digit maps directly to three binary bits, these data-path values were easily represented in octal. More recently, digital computers have standardized on data-path widths that are integer multiples of 8 bits; for example, 8 bits, 16 bits, 32 bits,

4 BINARY ARITHMETIC

Due to the fact that digital computers are constructed from logic gates that can represent only two states, they must make use of the binary number system.

4.1 Unsigned Binary Numbers

Unsigned binary numbers can only be used to represent positive values. Consider the range of numbers that can be represented using 8 bits (see Figure 10).

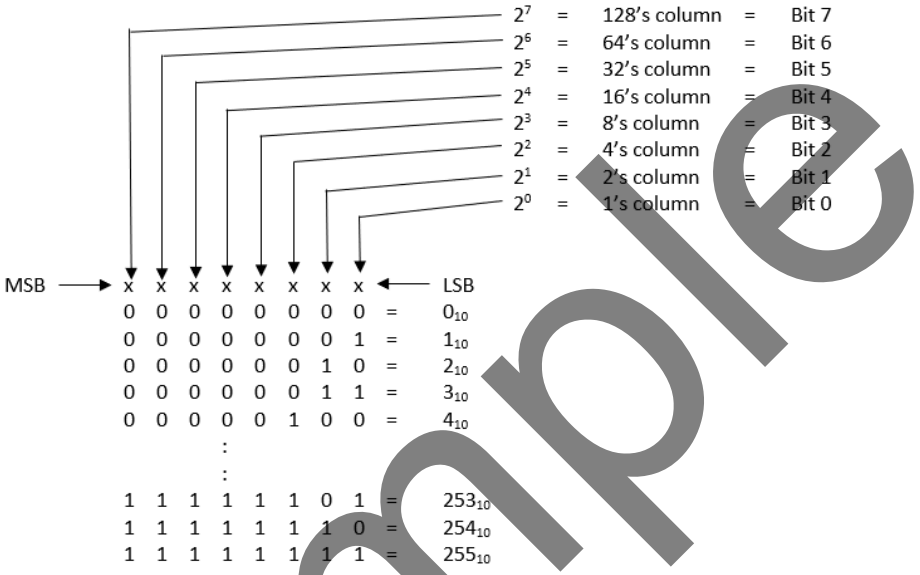


Figure 10 Unsigned Binary Numbers

Each 'x' character represents a single bit; the right-hand bit is known as the *least significant bit (LSB)* because it represents the smallest value. Similarly, the left-hand bit is known as the *most significant bit (MSB)* because it represents the largest value.

In computing it is usual to commence indexing things from zero, so the least significant bit is referred to as *bit 0*, and the most significant bit (of an 8-bit value) is referred to as *bit 7*. Every bit can be individually assigned a value of 0 or 1, so a group of 8 bits can be assigned $2^8 = 256$ unique combinations of 0s and 1s. This means that an 8-bit unsigned binary number can be used to represent values in the range 0_{10} through $+255_{10}$.

4.2 Binary Addition

Two binary numbers may be added together using an identical process to that used for decimal addition. First, the two least significant bits are added together to give a sum and, possibly, a carry-out to the next stage. This process is repeated for the remaining bits progressing towards the most significant. For each of the remaining bits, there may be a carry-in from the previous stage and a carry-out to the next stage. To fully illustrate this process, consider the step-by-step addition of two 8-bit binary numbers (Figure 11).

subtracted from a 9. The resulting nines complement value is added to the minuend, and then an end-around-carry operation is performed. The advantage of the nines complement technique is that it is never necessary to perform a borrow operation.
Now consider the same subtraction performed using the tens complement technique (Figure 13).

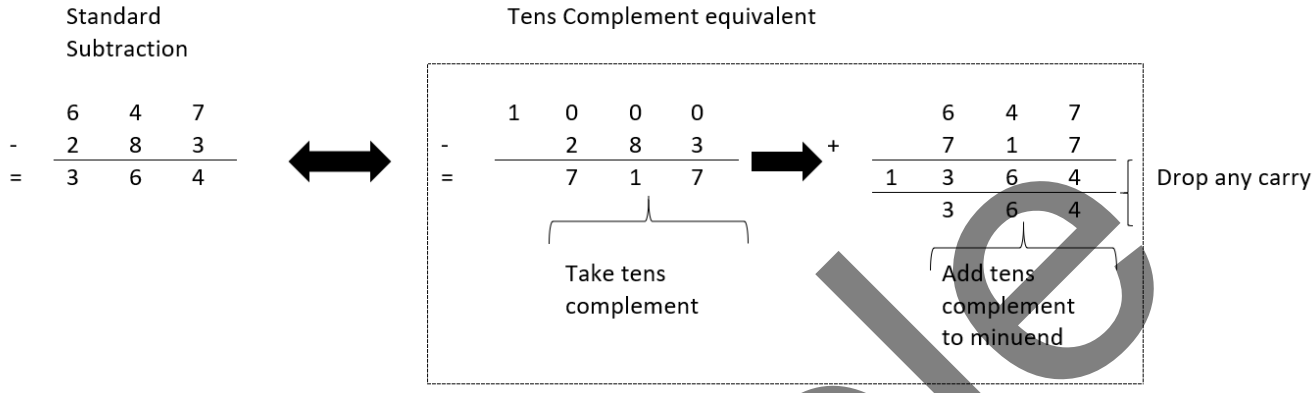


Figure 13 Tens complement decimal subtraction

The advantage of the tens complement is that it is not necessary to perform an end-around-carry; any carry-out resulting from the addition of the most significant digits is simply dropped from the final result. The disadvantage is that, during the process of creating the tens complement, it is necessary to perform a borrow operation for every digit in the subtrahend. This problem could be solved by first taking the nines complement of the subtrahend, adding one to the result, and then performing the remaining operations as for the tens complement.

Similar techniques may be employed with binary (base-2) numbers, where the radix complement is known as the *twos complement* and the diminished radix complement is known as the *ones complement*. First, consider a binary subtraction performed using the ones complement technique (Figure 14).

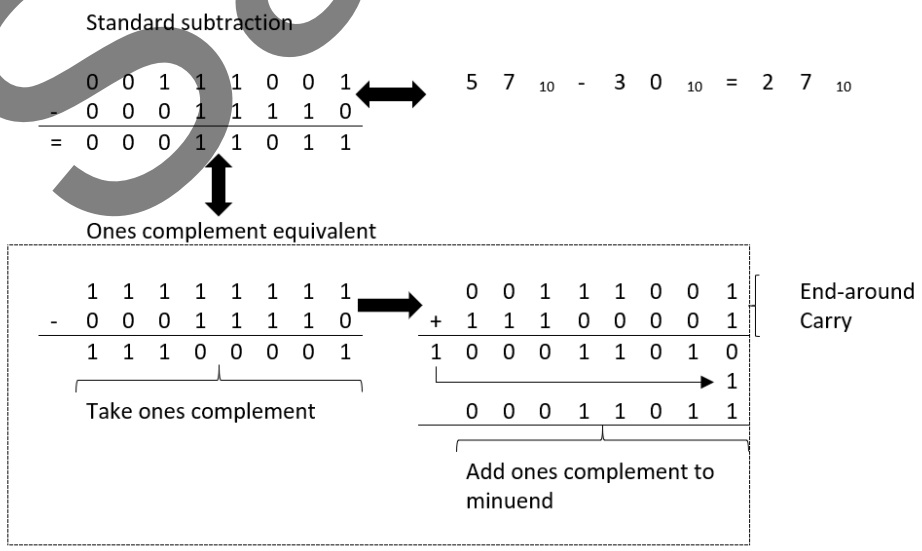


Figure 16 Shortcut for generating a twos complement

Unfortunately, all of the previous examples will return incorrect results if a larger value is subtracted from a smaller value; that is, for these techniques to work, the final result must be greater than or equal to zero. In the case of unsigned binary numbers, the reason is clear because, by definition, an unsigned binary number can only be used to represent a positive value but subtracting a larger value from a smaller value results in a negative value. Obviously, it would be useless if computers could only be used to generate positive values, so we need some way to represent negative numbers.

4.4 Signed Binary Numbers

Signed binary numbers can be used to represent both positive and negative values, and they do this in a rather cunning way. In standard decimal arithmetic, negative numbers are typically represented in a form known as *sign-magnitude*, which means prefixing values with plus or minus signs. However, for reasons of efficiency, computers rarely employ the sign-magnitude form, and instead use the *signed binary* format, in which the most significant bit is also called the *sign bit* (Figure 17).

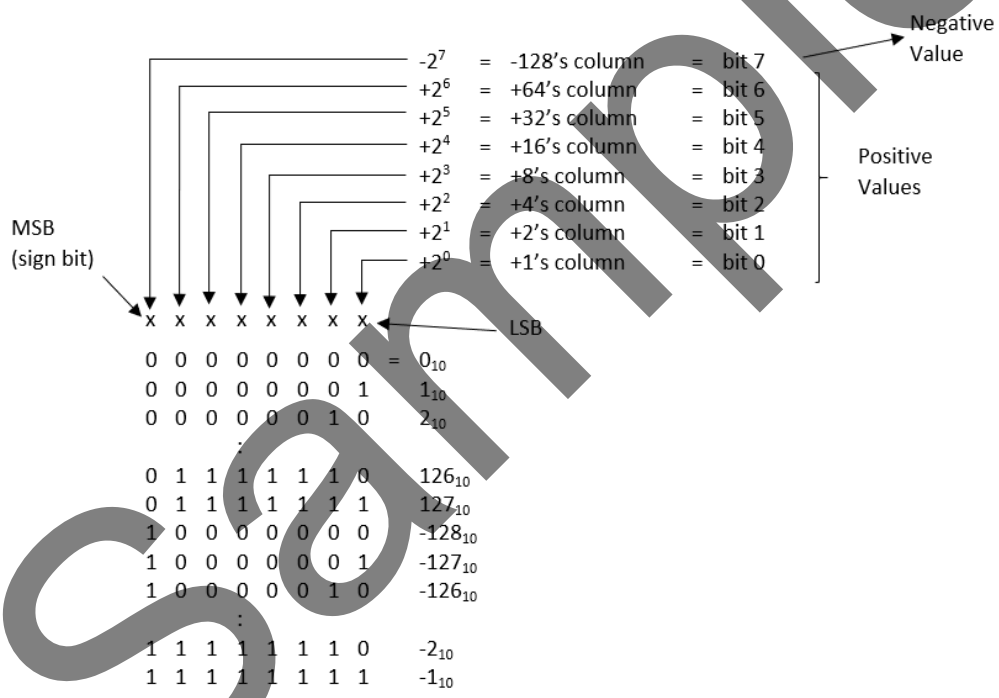


Figure 17 Signed binary numbers

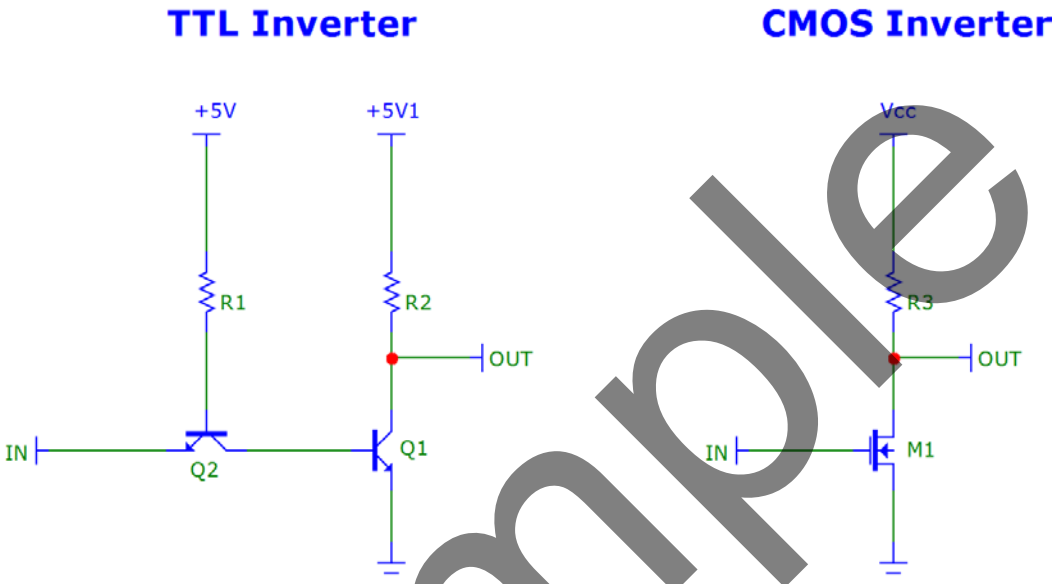
The least significant bits continue to represent the same positive quantities as for unsigned binary numbers, but the sign bit is used to represent a *negative quantity*. In the case of a signed 8-bit number, a 1 in the sign bit represents $-2^7 = -128$, and the remaining bits are used to represent positive values in the range 0_{10} to $+127_{10}$. Thus, when the value represented by the sign bit is combined with the values represented by the remaining bits, an 8-bit signed binary number can be used to represent values in the range -128_{10} to $+127_{10}$.

To illustrate the differences between the sign-magnitude and signed binary formats, first consider a positive sign-magnitude decimal number and its negative equivalent: for example, +27 and –27. As we see,

9 CHARACTERISTICS

9.1 TTL and CMOS Technologies

TTL stands for Transistor-Transistor Logic and CMOS stands for Complementary Metal Oxide Semiconductor. TTL devices are made using bipolar transistors (NPN and PNP types) and are current controlled. CMOS devices use nMOS and pMOS types and are voltage (field) controlled. The circuits below feature an inverter made from both types of technology.



The TTL inverter uses two NPN transistors. When the input is low (0) this draws current from +5V through R1 and turns Q2 ON. With Q2 being ON there is a low voltage on the base of Q1, meaning Q1 is OFF (high impedance). No current flows through Q1 (except for a negligible leakage current) and all of the supply voltage (+5V) is presented to the output. So, a logic 0 into this gate gives a logic 1 at the output. On the other hand, if a logic 1 (+5V) is presented to the input then Q2 will go OFF and a high will be presented to the base of Q1. This makes Q1 conduct and pulls the output low. So, a logic 1 into this gate gives a logic 0 at the output. An inverter is produced. Commercial circuits use modified designs to enhance performance and cost.

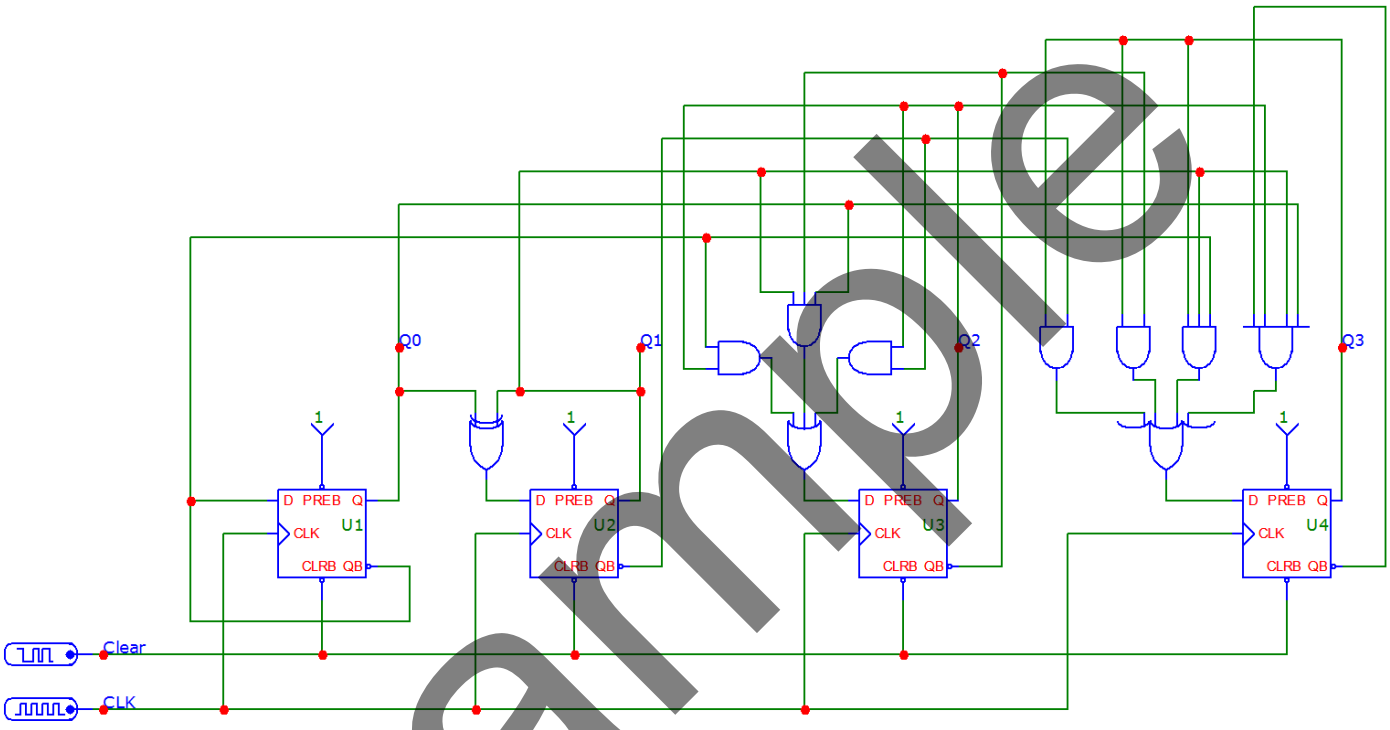
If the CMOS inverter sees a logic 0 at the input then M1 will be off and all of the supply voltage will appear at the output. So, a logic 0 input gives a logic 1 output. On the other hand, if a logic 1 is presented to the input then M1 conducts and brings the output low. Again, we have an inverter.

There are many designs for all types of logic gate, using TTL variants and CMOS variants, but the basics are given above. Let's now compare the characteristics of the two technologies.

Counter

A hexadecimal counter counts from 0 to 15. Because we don't use the numbers 10 through to 15 in hexadecimal we use the letters A through to F instead.

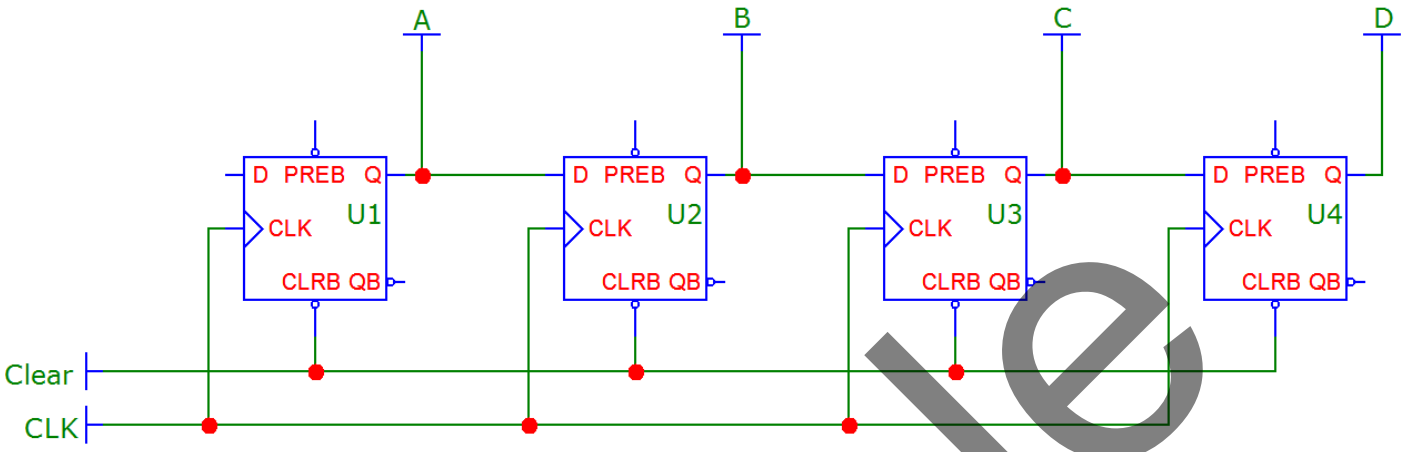
A hex counter will count from binary 0 to 15 and then re-start at 0 and carry on counting. You will see later how to implement formal design techniques for your sequential circuit designs. These techniques have been implemented to design the hex counter below, which uses D-type FF's and combinational logic…



A Transient Analysis will show the signals in the time domain…

Shift Register

A shift register will accept input bits and shift these along the outputs, bit-by-bit, after each clock pulse. Take a look at the D-FF implementation of a Serial-In Parallel-Out (SIPO) shift register below…



Serial data is injected into the D pin of U1 and this is transferred to the Q output of U1 (A) on the next rising edge of the CLK. Since Q of U1 is connected to D of U2 then the same bit of data is transferred to Q of U2 (B) after the next CLK etc.

If a system uses a 4 bit word then 4 CLK pulses will place this word on the lines ABCD for subsequent processing.

11.2 Design of Sequential Circuits

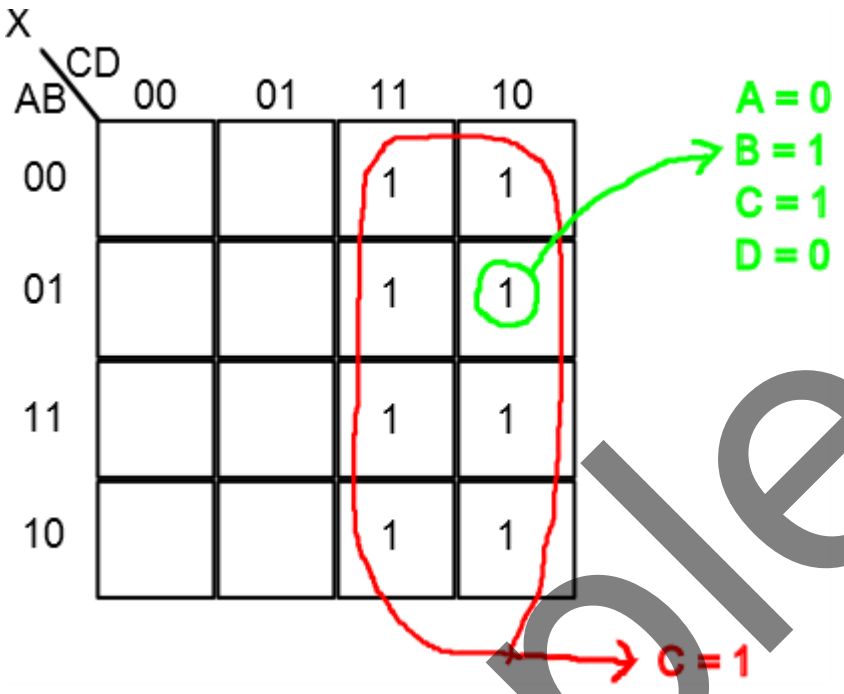
Minimisation

Minimisation of Boolean statements is critical if efficient and fast logic circuits are to be implemented in a system. Consider the case where a circuit output X is subject to combinations of 4 different inputs (ABCD), as follows…

$$X = \bar{A}\bar{B}CD + \bar{A}\bar{B}C\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D} + ABCD + ABC\bar{D} + A\bar{B}CD + A\bar{B}C\bar{D}$$

That's some horrible looking expression. Without any skills in Minimisation we would be implementing the combinational logic as we read it. That would involve an 8-input OR gate, eight 4-input AND gates and twelve inverters, right? That's going to be expensive, slow and costly on silicon space – 21 gates used up. Let's see if we can minimise this expression. To do so normally involves the use of a Karnaugh Map.

A Karnaugh Map lays out all the possible combinations of the inputs into squares. If we have four digital inputs then there will be $2^4 = 16$ squares. Let's look at a Karnaugh Map for the four inputs ABCD…



The Karnaugh Map marks the possible logic levels for each pair of signals in the form of a Gray Code. A Gray Code only allows one input to change at any one time, so we mark as ***00, 01, 11, 10*** rather than 00, 01, 10, 11 because going from 01 to 10 involves two changes at the same time, which is illegal if we wish to minimise efficiently.

The eight terms in our expression for X are marked with 1's on the Karnaugh Map, as shown. Minimisation involves grouping terms in 2's, 4's or 8's. The terms in this example all form a nice group of eight, as indicated in red. Looking above the red highlighted area we see that what's in common with all of those terms is when C is 1. The startling fact which hits us here is that…

We don't need 21 gates, we just need to connect X to C

You will see more examples of minimisation using Karnaugh Maps by using formal design techniques, as demonstrated in the following worked example.

Worked Example 1

Use formal design techniques to produce the D-Type flip-flop equations for a 0 to 5 Up-Counter.

Our formal design technique firstly requires writing the specification for the system in the form of a ***Modified State Transition Table***. This table lists all possible present states and the desirable next states to achieve our specification. Here's the Modified State Transition Table for our example…

Q2	Q1	Q0	Q2'	Q1'	Q0'	D2	D1	D0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0

The table is split into three areas, as shown. The first three columns show all possible current states for a set of three D-type flip flops. Here there must be $2^3 = 8$ rows, as indicated. Notice that the Q0 column toggles 01010101 and the Q1 column is half as fast i.e. 00110011. The Q2 column is half as fast again i.e. 00001111. This is a standard way to construct the input patterns for Truth Tables.

The central area of the table contains the dashed data. The dashes mean 'required next state'. For example, if the current state Q2, Q1, Q0 is 000 (i.e. a count of zero) then we would like the next count to be 001 (i.e. a count of 1). This wish/requirement is indicated in the columns headed Q2', Q1', Q0'.

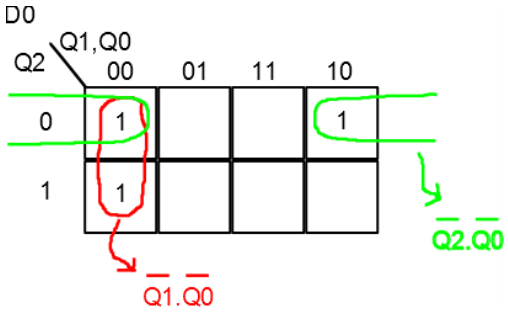
You might be wondering why the columns headed D2, D1, D0 are there, since they are exactly the same as the dashed (next state) columns. Well, we know that the D input to a D-type FF will become the next state after the next clock pulse. Those D columns are simply there to help you to form the FF input equations.

To form those FF input equations what we are really asking is '***what/how do I connect to D0 or D1 or D2 to make this circuit work?***'

This where our Karnaugh Maps come in really handy – they will tell us what gates we need for each D input and how they should be connected.

Let's try to work out the requirement for D0 first of all. Take a look at the D0 column. Wherever there is a 1 in the D0 column we need to make a note of the corresponding Q states and enter them into our Truth Table. The first 1 in the D0 column happens to be in the first row so we look across to see the states of Q, Q1 and Q0 in that row. These states are 000 so we must enter a 1 into the Truth Table. Further 1's are

noticed in rows 3 and 5, so we must enter 1's into the Truth Table at 010 and 100. Here's the completed Truth Table for our D0 input…



Those three 1's have been entered. We now try to group the data in 2's, 4's or 8's, where possible. The two 1's highlighted in red both lie in the column where Q1 and Q0 are both 0. We also notice for the red group that Q2 can be either 0 or 1. We always try to look for common states in Truth Tables, so we ignore Q2 altogether and record this red group as Q1 being 0 ***and*** Q0 being 0 i.e. $\overline{Q1}.\overline{Q0}$ is the term in red.

We now need to turn our attention to the solitary 1 over on the right hand side of the Truth Table. It would be very bad practice to call this $\overline{Q2}.Q1.\overline{Q0}$. It would still be valid and produce a working circuit, but that circuit would be wasteful of gates and slower than it need to be. We need to remember that we're in the business of ***grouping*** when we analyse our Karnaugh Maps. If we were to get some scissors and cut out the Karnaugh Map we could then fold the left edge of the paper and the right edge of the paper to form a tube. Having made the tube we see that the 1 we are looking at is not solitary after all – it's next to the 1 in the top left hand corner. This fact is highlighted by the green grouping in the diagram. For this green grouping we see that what's common is Q2 being 0 ***and*** Q0 being 0 i.e. $\overline{Q2}.\overline{Q0}$ is the term in green.

We have now minimised the equation for D0 and this can finally be expressed as…

$$\boldsymbol{D0 = \overline{Q1}.\overline{Q0} + \overline{Q2}.\overline{Q0}}$$

To connect the groups to form the full expression for D0 we must use an OR gate, indicated by the '+' sign above.

That's all there is really for minimisation with Karnaugh Maps. The only other thing to remember is that when you cut out the Karnaugh map you could just as easily have connected to top and bottom edges to form groups also – that task is not needed in this example though.

One final thought: If you are analysing a Karnaugh Map and you cannot see any groupings then no minimisation is possible and you must represent each 1 in its full form. This is something you come across when trying to find the input equations for D1 and D2, which are…

$$\boldsymbol{D1 = \overline{Q2}.\overline{Q1}.Q0 + \overline{Q2}.Q1.\overline{Q0}}$$

$$\boldsymbol{D2 = Q2.\overline{Q1}.\overline{Q0} + \overline{Q2}.Q1.Q0}$$

Worked Example 2

Use formal design techniques to produce the D-Type flip-flop equations for a 4 to 0 Down-Counter.

Here's the Modified State Transition Table for our example…

Q2	Q1	Q0	Q2'	Q1'	Q0'	D2	D1	D0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	1
0	1	1	0	1	0	0	1	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	0	1	0	0
1	1	0	1	0	0	1	0	0
1	1	1	1	0	0	1	0	0

(*See Worked Example 1 for detailed guidance on this current example.*)

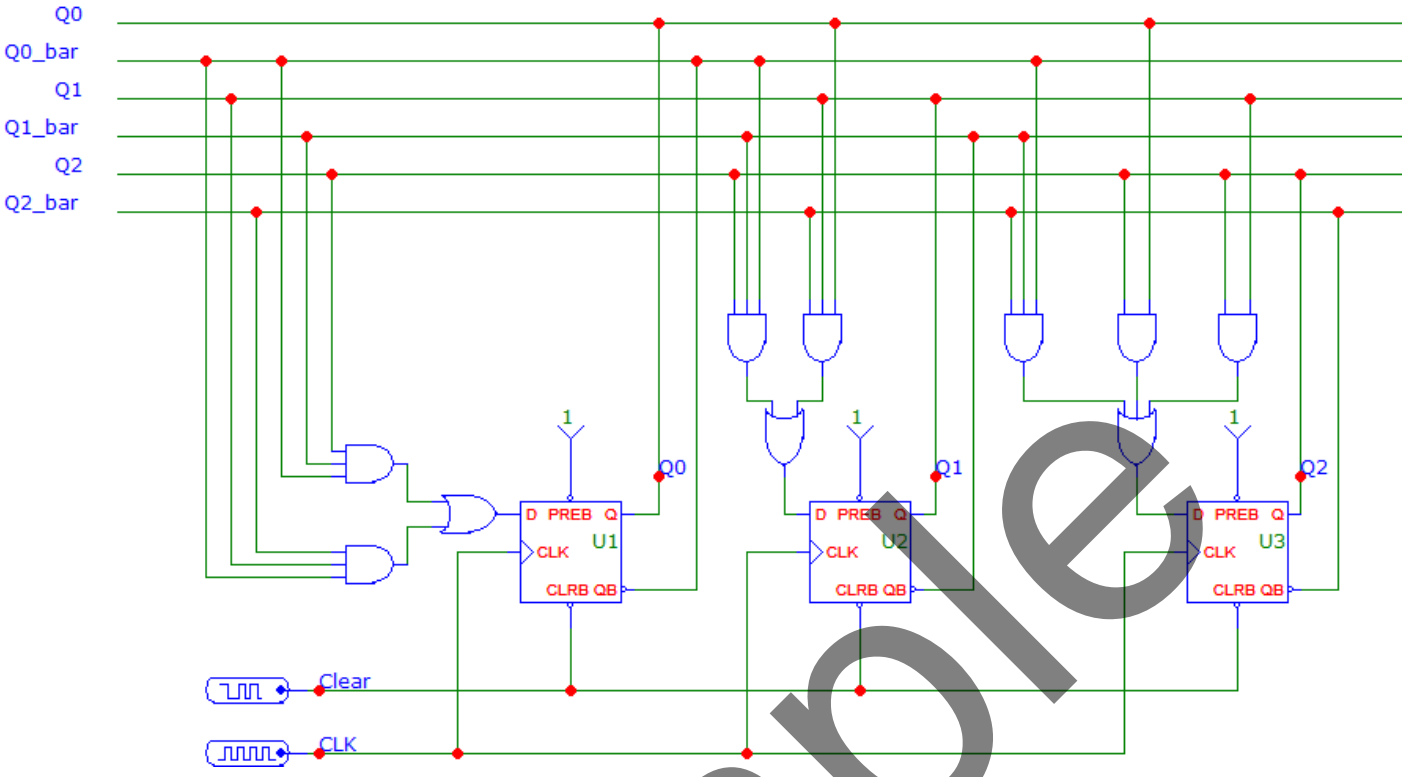
Karnaugh Map analysis of this table yields the following FF input equations (try them yourself for confirmation)…

$$D0 = Q2.\overline{Q1}.\overline{Q0} + \overline{Q2}.Q1.\overline{Q0}$$

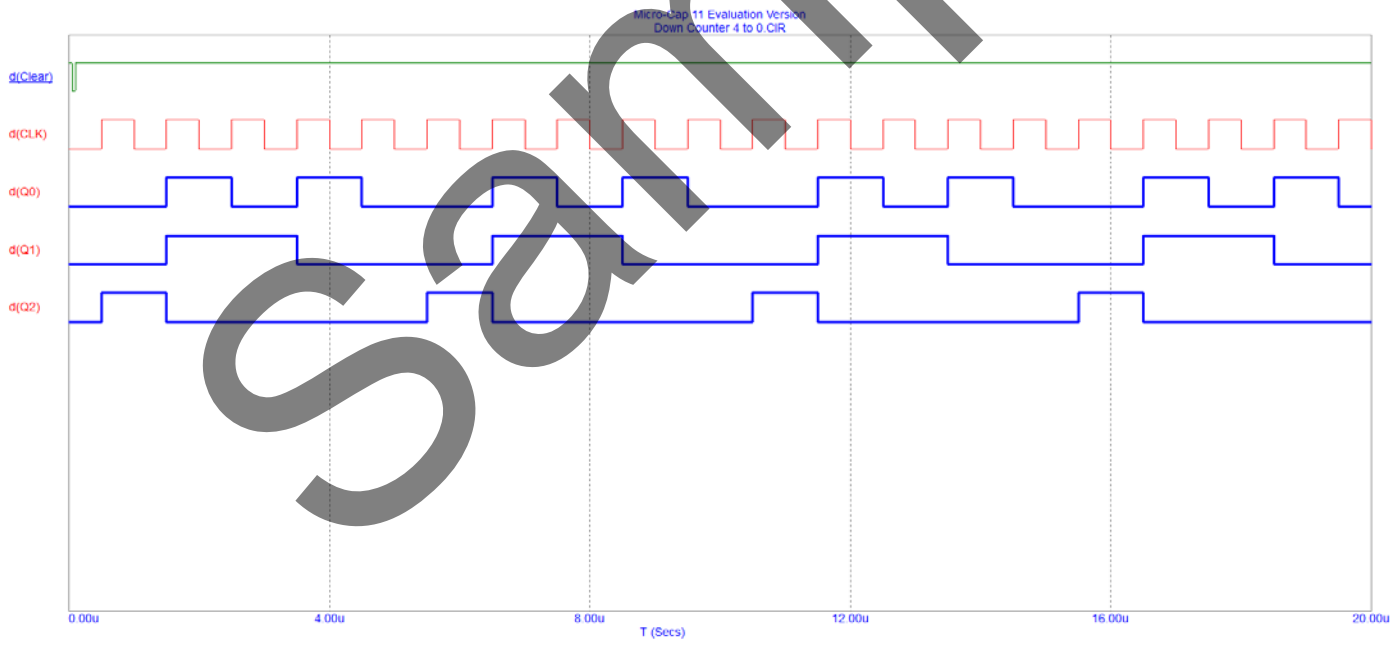
$$D1 = Q2.\overline{Q1}.\overline{Q0} + \overline{Q2}.Q1.Q0$$

$$D2 = \overline{Q2}.\overline{Q1}.\overline{Q0} + Q2.Q0 + Q2.Q1$$

These equations for the D inputs may be turned into combinational logic gates and the whole circuit may be constructed in MicroCap as follows…



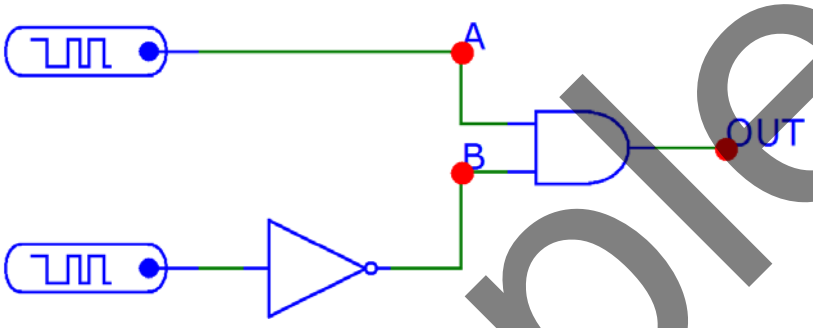
The resulting Transient Analysis confirms that the circuit does indeed repeatedly count from 4 to 0…



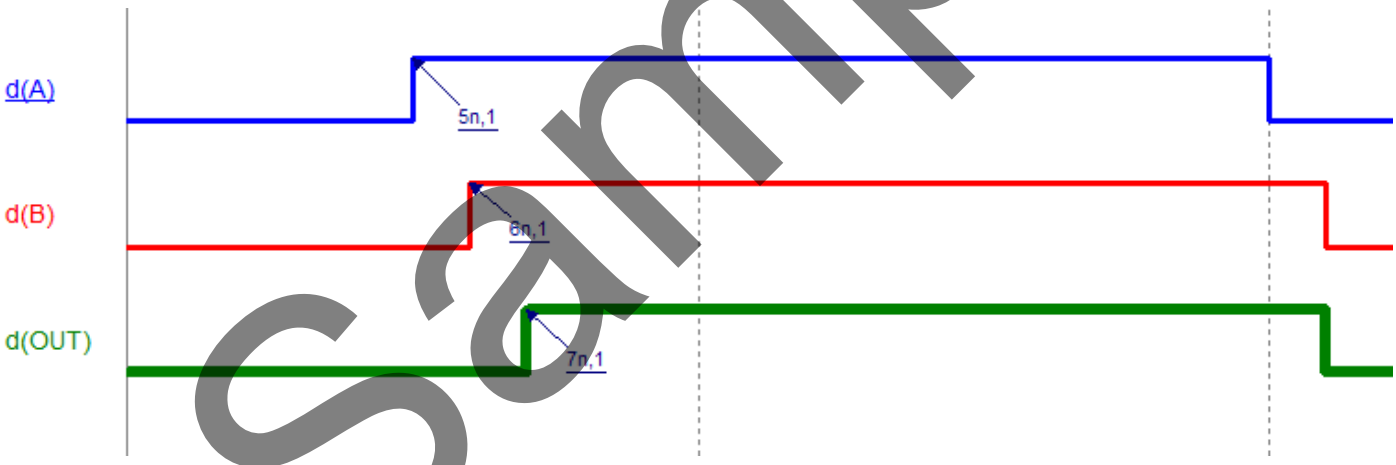
Race Hazards

Every logic gate has a built in delay. That means that when data is presented to a gate the output does not respond immediately – there is a delay, caused by internal parasitic capacitances which are unavoidable. The more gates you have in a circuit then the more delay there will be (that's why it's a good idea to minimise with Karnaugh Maps).

If we have an AND gate with one input fed directly by a logic 1 and the other input fed via an inverter then the two inputs '***race***' each other to present their logic levels to the AND gate. The directly fed input will arrive before the other one since it needs to negotiate the delay of the inverter before it meets the AND gate. The situation is illustrated below…



A transient analysis (using non-ideal components) produces the following timing diagram…



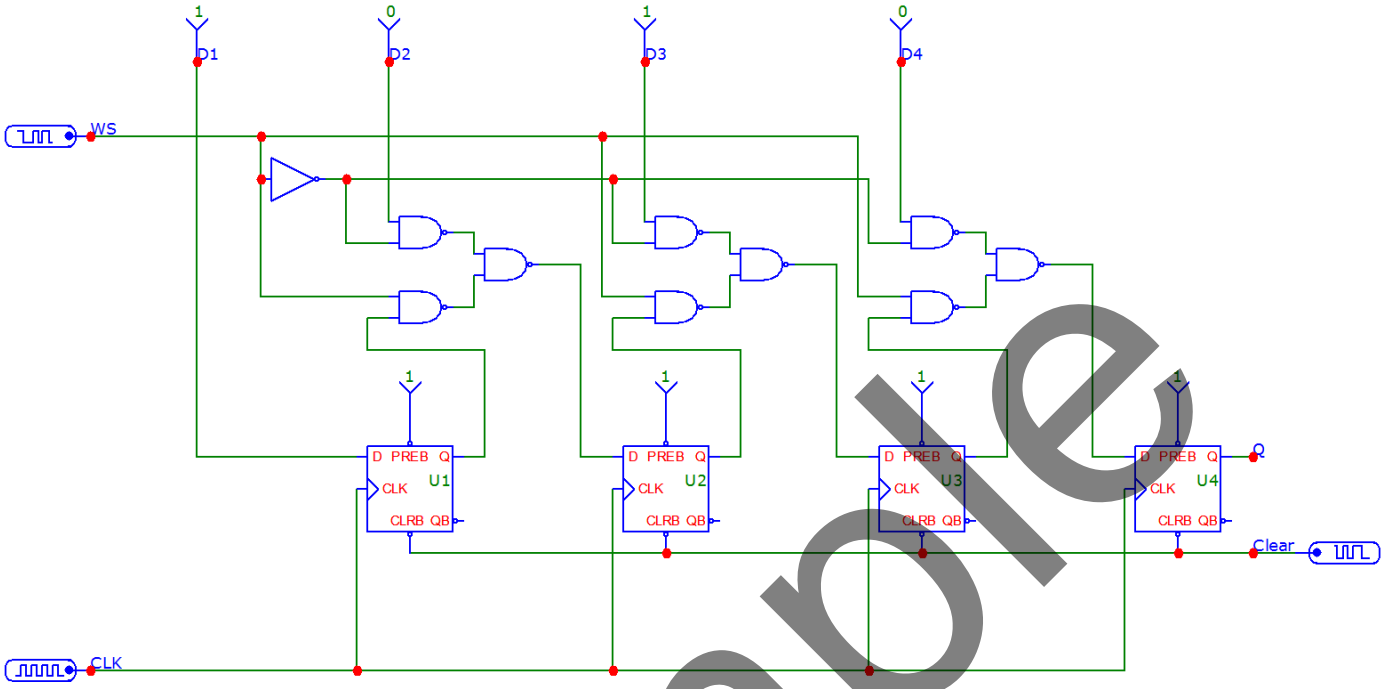
Here we see that the immediate signal (A) decides to go high after 5ns (blue). The inverter is presented with a logic 0 at 5ns but, because it has a built in delay, this will not be presented to the AND gate until 6ns (red). The AND gate too has its own built in delay, so, rather than go high after 6ns (when A and B are both high) it goes high a further 1ns after the wanted time. Such Race hazards present great challenges to circuit designers working at very high clock rates.

Clock Speed

As just mentioned, high clock speeds can cause race hazards in sequential circuits. High clock speeds also bring problems in terms of both additional parasitic capacitances (causing delays/glitches) and higher power requirements (quicker battery drain in modern devices).

Parallel to Serial Converter

This circuit takes four parallel bits of data (ABCD in the diagram below)



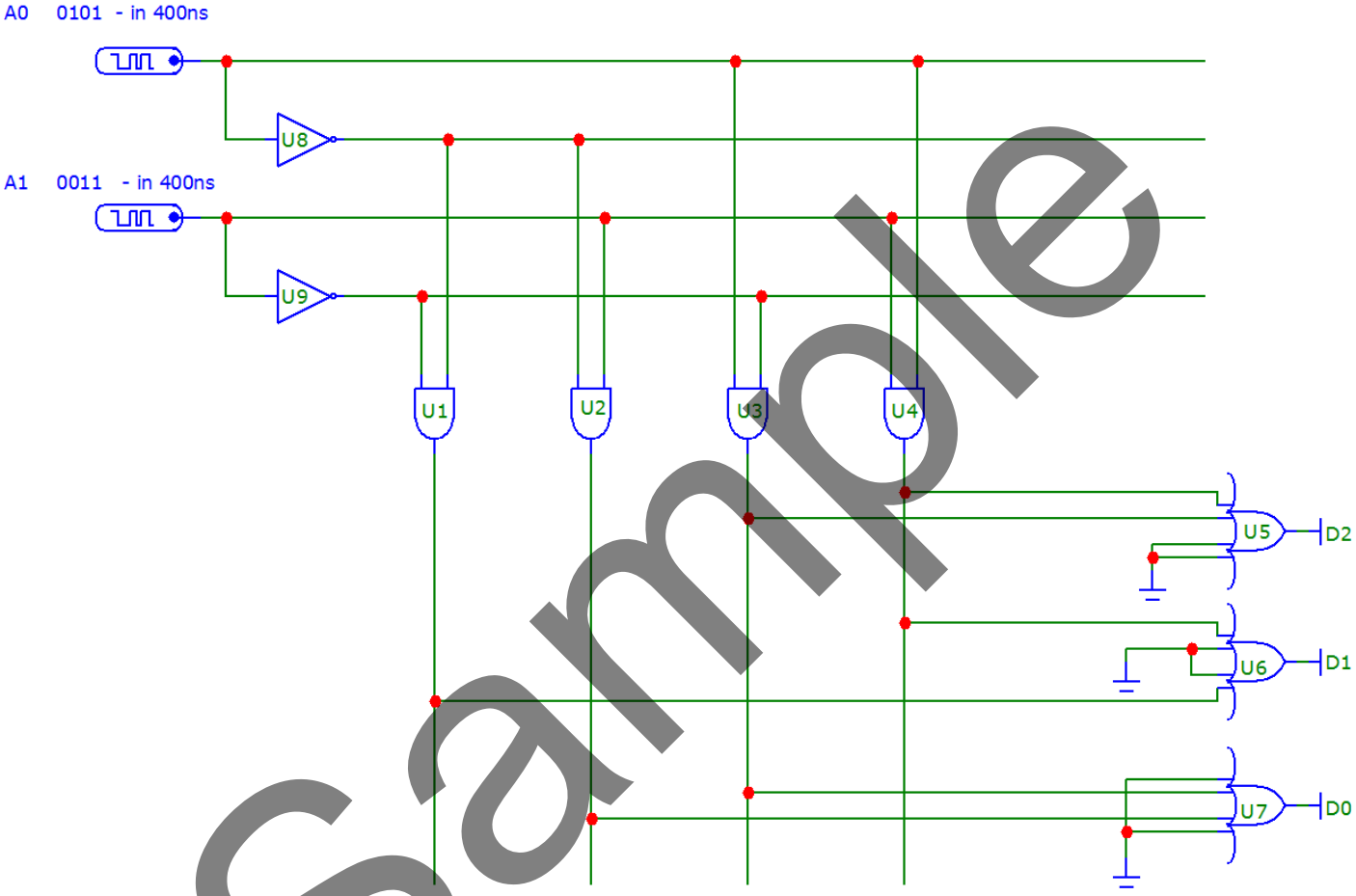
Assume we present the circuit with parallel data DCBA = 1010, as shown. We would like to see the Q output from U4 (gate 4) change state with this 1010 pattern, thus giving us a serial form of the parallel input data.

The first thing to do is to keep the dual-function line WS (Write/Shift) low so that we may write the data to each D input via the combinational logic. We then turn the WS line high (i.e. request Shift mode) which moves the data to the right on each of 4 successive clock pulses. We have ourselves a PISO (parallel in serial out) shift register, as confirmed by the Transient Analysis below…

12 PROGRAMMABLE LOGIC DEVICES

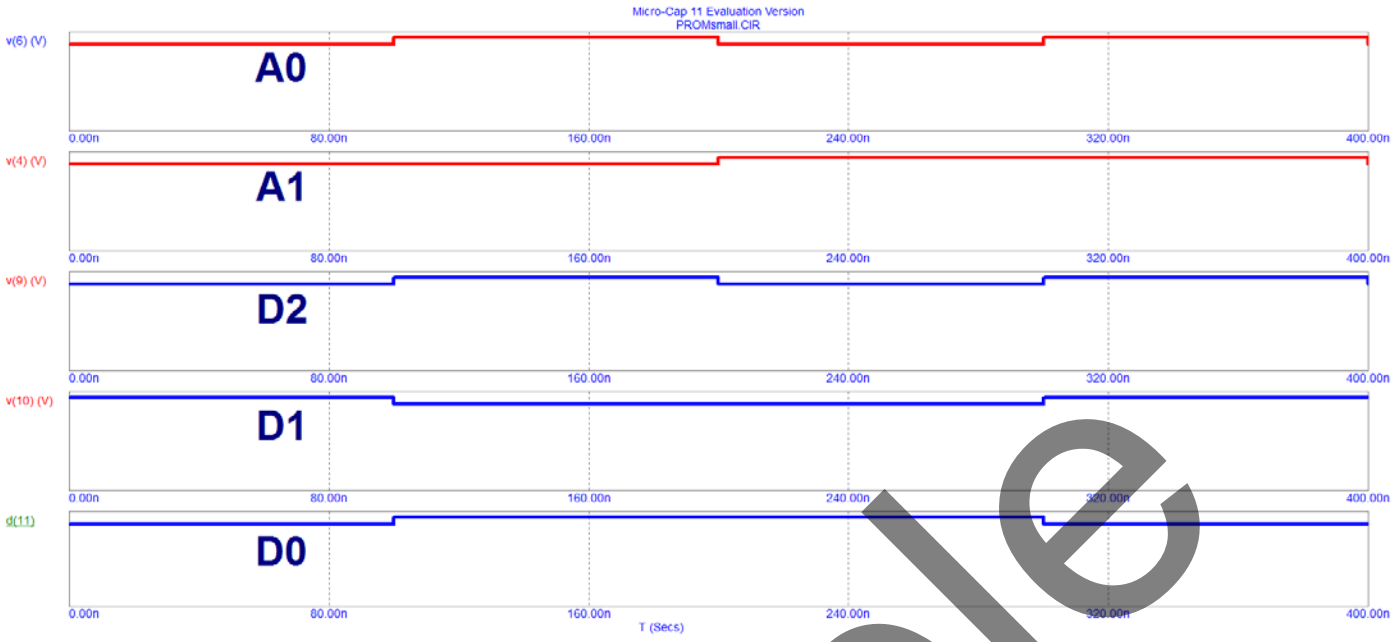
12.1 Programmable Read-Only Memory (PROM)

The figure below shows a simple PROM constructed in the MicroCap simulator. The memory location you wish to read is entered into A0 and A1. The output from the PROM is given by data bits D0, D1 and D2.



We could have any number of 4-input OR gates here but have chosen to keep the circuit simple (the Student Version of MicroCap limits the components and nodes, but it is still a great free simulator). The PROM has fixed AND wiring but flexible OR wiring. For example, U5 has its lower two inputs connected to ground, but these could easily have been connected to the outputs of U1/U2. These OR connections are usually ALL in place in a chip from the factory. We use a PROM programmer to remove the lines we wish to erase (little fuses are blown with high currents and the result is permanent).

The Transient Analysis of this circuit is as follows (zoom in for fine detail)…



Challenge

See if you can follow through the logic levels from the circuit diagram for this PROM.

12.2 Programmable Logic Devices

There are several extensively used types of programmable logic device. In Assignment 3 you get the chance to use one of these when designing a circuit. For now though, a quick list of types and parameters will suffice…

- PROM – we have just examined this
- PLA – programmable logic array, programmable AND with programmable OR
- PAL – programmable array logic, fixed OR with programmable AND
- GAL – generic array logic, similar to PAL but erasable/reprogrammable
- CPLD – complex programmable logic device, larger versions of PALs/GALs
- FPGA – field programmable gate array, meant for the most complex designs