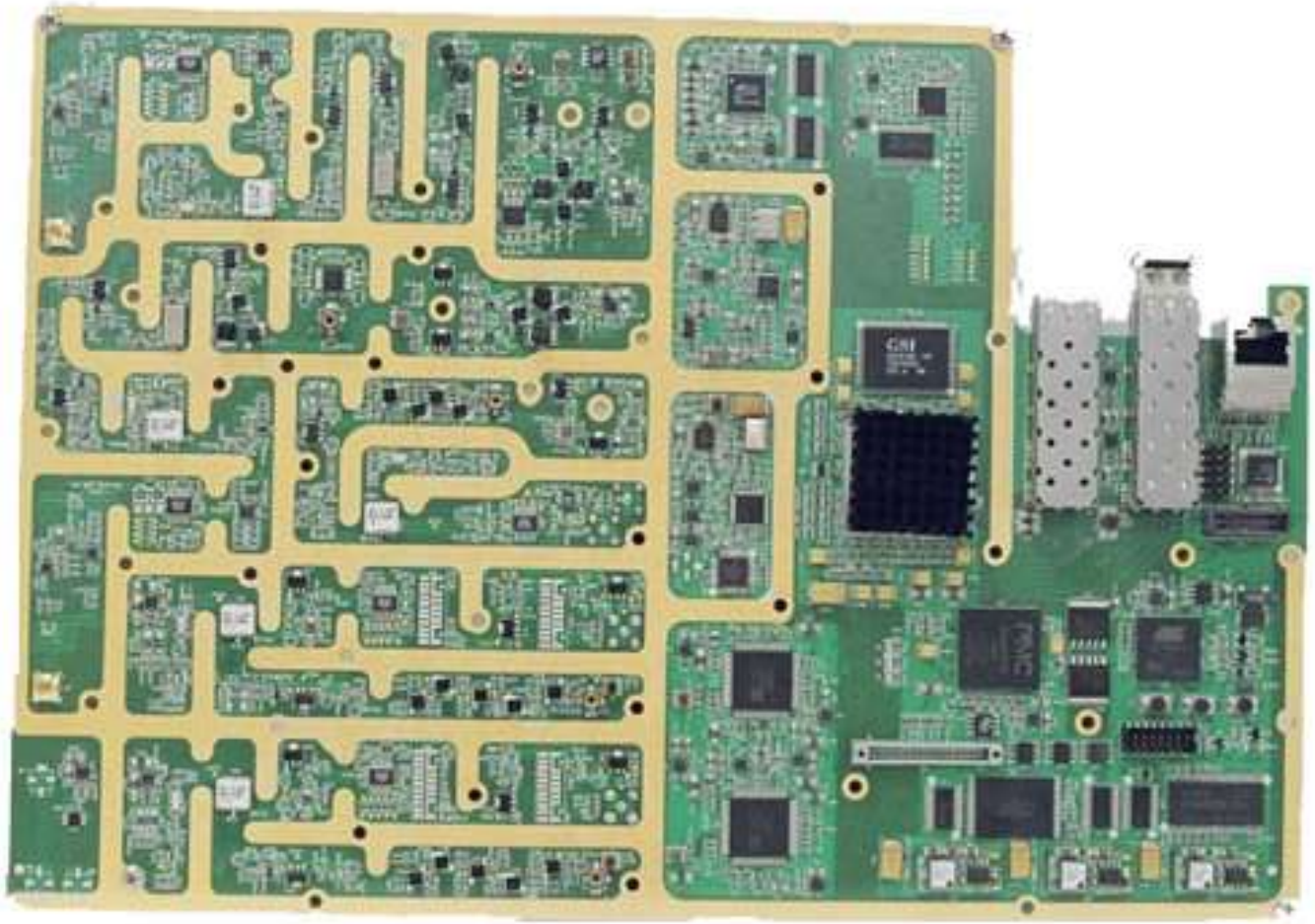
Digital Circuits



Jasmin Malley

First Edition, 2012

ISBN 978-81-323-2845-2

Published by:
Orange Apple
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

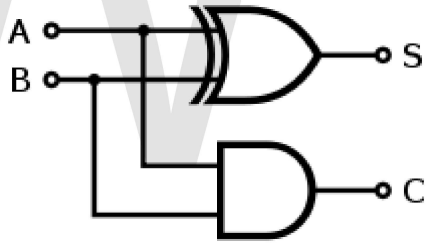
Table of Contents

Chapter- 1

Adder

In electronics, an **adder** or **summer** is a digital circuit that performs addition of numbers. In modern computers adders reside in the arithmetic logic unit (ALU) where other operations are performed. Although adders can be constructed for many numerical representations, such as Binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where two's complement or one's complement is being used to represent negative numbers, it is trivial to modify an adder into an adder-subtractor. Other signed number representations require a more complex adder.

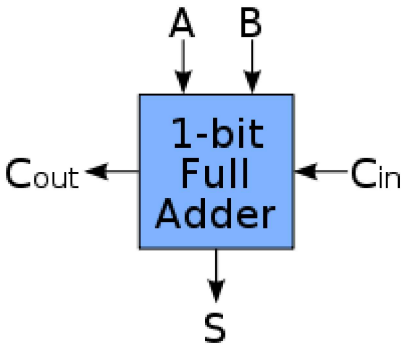
Half adder



Example half adder circuit diagram

A **half adder** adds two one-bit binary numbers A and B. It has two outputs, S and C (the value theoretically carried on to the next addition); the final sum is $2C + S$. The simplest half-adder design, pictured on the right, incorporates an XOR gate for S and an AND gate for C. Half adders cannot be used compositely, given their incapacity for a carry-in bit.

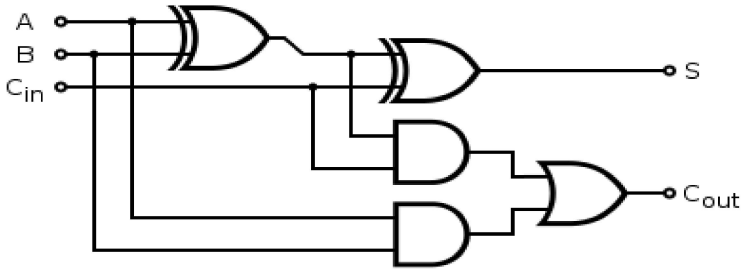
Full adder



Schematic symbol for a 1-bit full adder with C_{in} and C_{out} drawn on sides of block to emphasize their use in a multi-bit adder

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as A, B, and C_{in}; A and B are the operands, and C_{in} is a bit carried in (in theory from a past addition). The circuit produces a two-bit output sum typically represented by the signals C_{out} and S, where $\text{sum} = 2 \times C_{out} + S$. The one-bit full adder's truth table is:

Inputs			**Outputs**	
A	***B***	C_{in}	C_{out}	***S***
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



Example full adder circuit diagram; the AND and OR gates can be replaced with NAND gates for the same results

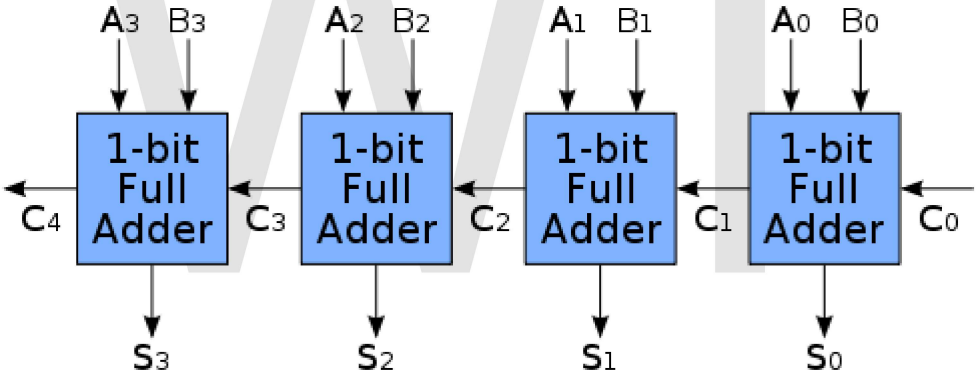
A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with $S = A \oplus B \oplus C_{in}$ and $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$.

In this implementation, the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. Using only two types of gates is convenient if the circuit is being implemented using simple IC chips which contain only one gate type per chip. In this light, C_{out} can be implemented as $C_{out} = (A \cdot B) \oplus (C_{in} \cdot (A \oplus B))$.

A full adder can be constructed from two half adders by connecting *A* and *B* to the input of one half adder, connecting the sum from that to an input to the second adder, connecting C_i to the other input and OR the two carry outputs. Equivalently, *S* could be made the three-bit XOR of *A*, *B*, and C_i, and C_o could be made the three-bit majority function of *A*, *B*, and C_i.

More complex adders

Ripple carry adder

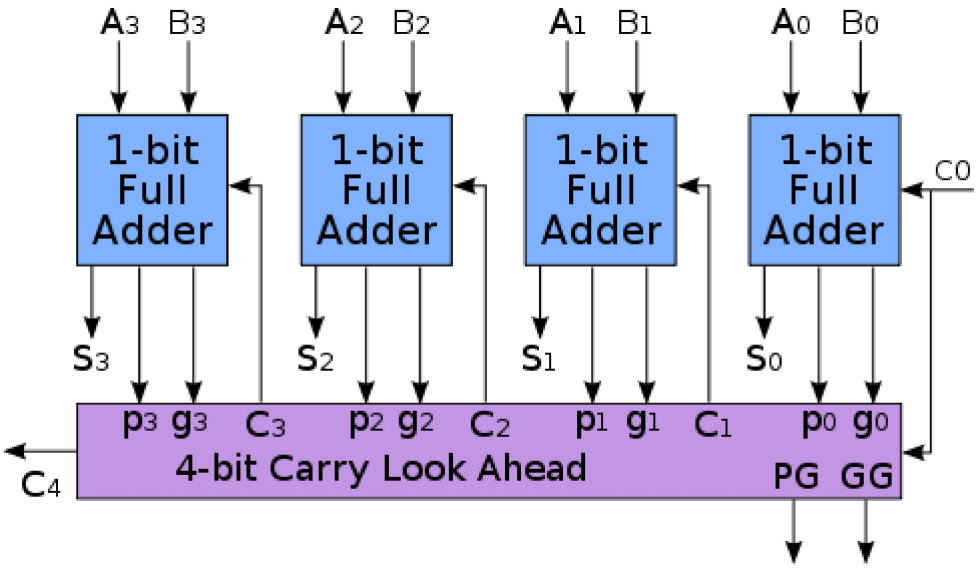


4-bit adder with logic gates shown

It is possible to create a logical circuit using multiple full adders to add *N*-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a *ripple carry adder*, since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder.

The layout of a ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is 31 * 2(for carry propagation) + 3(for sum) = 65 gate delays.

Carry look-ahead adders



4-bit adder with Carry Look Ahead

To reduce the computation time, engineers devised faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals (*P* and *G*) for each bit position, based on if a carry is propagated through from a less significant bit position (at least one input is a '1'), a carry is generated in that bit position (both inputs are '1'), or if a carry is killed in that bit position (both inputs are '0'). In most cases, *P* is simply the sum output of a half-adder and *G* is the carry output of the same adder. After *P* and *G* are generated the carries for every bit position are created. Some advanced carry look ahead architectures are the Manchester carry chain, Brent-Kung adder, and the Kogge-Stone adder.

Some other multi-bit adder architectures break the adder into blocks. It is possible to vary the length of these blocks based on the propagation delay of the circuits to optimize computation time. These block based adders include the carry bypass adder which will determine *P* and *G* values for each block rather than each bit, and the carry select adder which pre-generates sum and carry values for either possible carry input to the block.

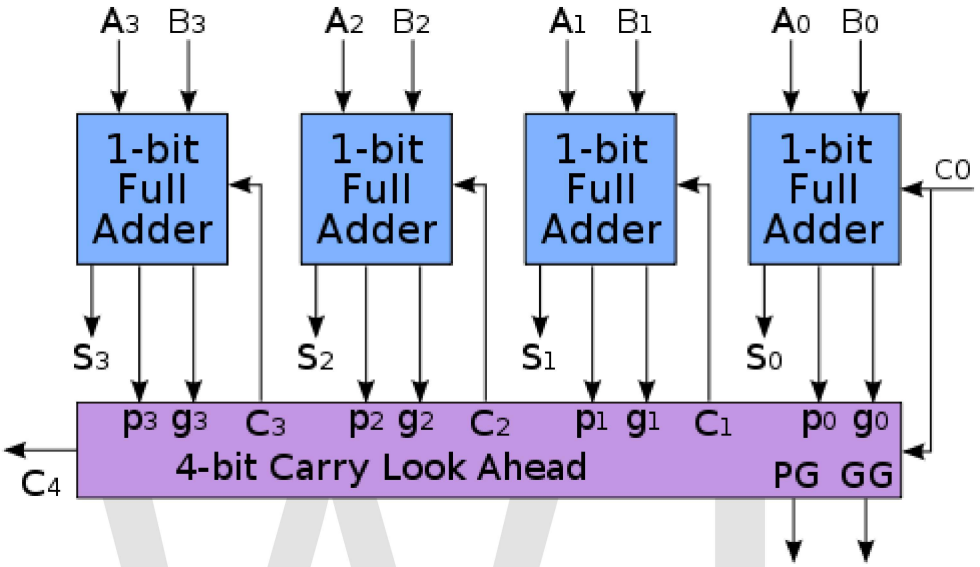
Other adder designs include the conditional sum adder, carry skip adder, and carry complete adder.

Look-ahead carry unit

A **Lookahead Carry Unit** (**LCU**) is a logical unit in digital circuit design used to decrease calculation time in adder units and used in conjunction with carry look-ahead adders (CLAs).

16-bit adder

By combining four 4-bit CLAs, a 16-bit adder can be created but additional logic is needed in the form of an LCU. A single 4-bit CLA is shown below:



4-bit adder with Carry Look Ahead (CLA)

The LCU accepts the group propagate (*PG*) and group generate (*GG*) from each of the four CLAs. The LCU then generates the carry input for each CLA.

Assume that P_i is *PG* and G_i is *GG* from the i[th] CLA then the output carry bits are

$$
\begin{aligned}
C_4 &= G_0 + P_0 \cdot C_0 \\
C_8 &= G_4 + P_4 \cdot C_4 \\
C_{12} &= G_8 + P_8 \cdot C_8 \\
C_{16} &= G_{12} + P_{12} \cdot C_{12}
\end{aligned}
$$

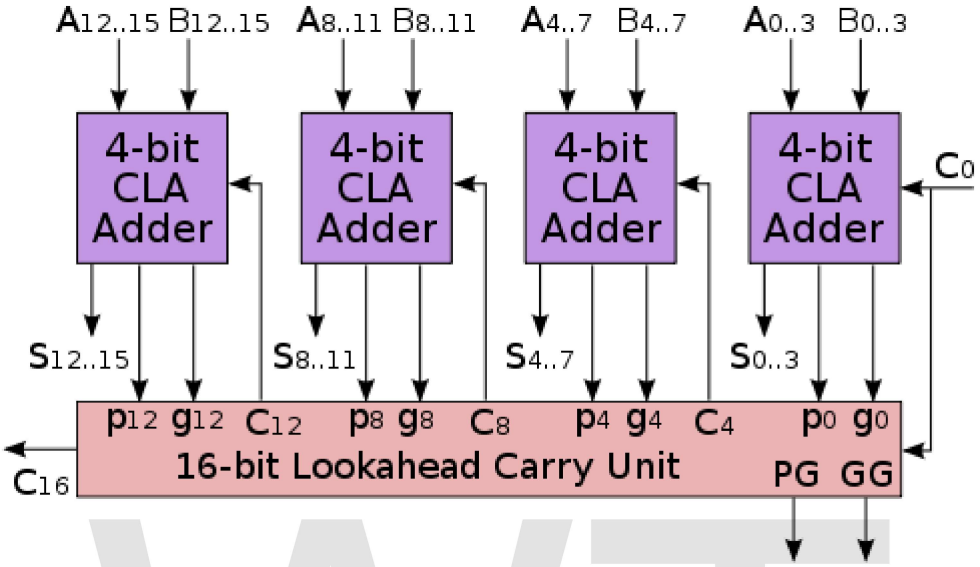
Substituting C_4 into C_8, then C_8 into C_{12}, then C_{12} into C_{16} yields the expanded equations:

$$
\begin{aligned}
C_4 &= G_0 + P_0 \cdot C_0 \\
C_8 &= G_4 + G_0 \cdot P_4 + C_0 \cdot P_0 \cdot P_4 \\
C_{12} &= G_8 + G_4 \cdot P_8 + G_0 \cdot P_4 \cdot P_8 + C_0 \cdot P_0 \cdot P_4 \cdot P_8 \\
C_{16} &= G_{12} + G_8 \cdot P_{12} + G_4 \cdot P_8 \cdot P_{12} + G_0 \cdot P_4 \cdot P_8 \cdot P_{12} + C_0 \cdot P_0 \cdot P_4 \cdot P_8 \cdot P_{12}
\end{aligned}
$$

C_4 corresponds to the carry input into the second CLA; C_8 to the third CLA; $C12$ to the fourth CLA; and C_{16} to overflow carry bit.

In addition, the LCU can calculate its own propagate and generate:

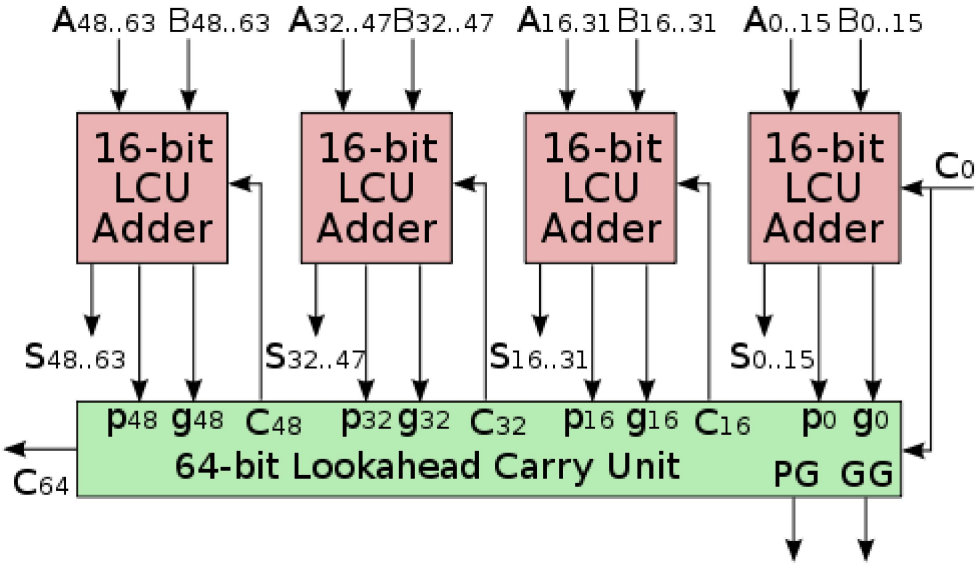
$$P_{LCU} = P_0 \cdot P_4 \cdot P_8 \cdot P_{12}$$
$$G_{LCU} = G_{12} + G_8 \cdot P_{12} + G_4 \cdot P_{12} \cdot P_8 + G_0 \cdot P_{12} \cdot P_8 \cdot P_4$$



16-bit adder with LCU

64-bit adder

By combining 4 CLAs and an LCU together creates a 16-bit adder. Four of these units can be combined to form a 64-bit adder. An additional (second-level) LCU is needed that accepts the propagate (P_{LCU}) and generate (G_{LCU}) from each LCU and the four carry outputs generated by the second-level LCU are fed into the first-level LCUs.



64-bit adders with a second-level LCU

3:2 compressors

We can view a full adder as a *3:2 compressor*: it sums three one-bit inputs, and returns the result as a single two-bit number. Thus, for example, a binary input of *101* results in an output of *1+0+1=10* (decimal number '2'). The carry-out represents bit one of the result, while the sum represents bit zero. Likewise, a half adder can be used as a *2:2 compressor*.

3:2 compressors can be used to speed up the summation of three or more addends. If the addends are exactly three, the layout is known as the carry-save adder. If the addends are four or more, more than one layer of compressors is necessary and there are various possible design for the circuit: the most common are Dadda and Wallace trees. This kind of circuit is most notably used in multipliers, which is why these circuits are also known as Dadda and Wallace multipliers.

Serial binary adder

The **serial binary adder** is a digital circuit that performs binary addition bit by bit. The serial full adder has three single bit inputs for the numbers to be added and the carry in. There are two single bit outputs for the sum and carry out. The carry in signal is the previously calculated carry out signal. The addition is performed by adding each bit, lowest to highest, once each clock cycle.

Serial Binary Addition

Serial Binary addition is done by, in simplest terms, a flip-flop and a full adder as stated above. However, there are slight nuances to the addition that can be confusing. When a serial adder performs its addition, it is partially dependent on the clock cycle as a flip-flop is asynchronous and the full adder is not. Thus, when a timing diagram is done, the sum output will change as the inputs are changed, relative to the previous clock cycle which was used to determine the carry in bit. In a serial binary adder of, for example, 8 bits one bit each of data is loaded and given to the full adder during one clock cycle and the resulting sum is stored in a register. In this way the 8 bit sum is obtained after 8 clock cycles . Design is divided in two parts controller and architecture. The controller part consists of control signal like load inputs. A counter should also be used in the architecture that tells that 8 clock cycles are over and the resulting sum is obtained .

Serial binary subtracter

The serial binary subtracter operates the same as the serial binary adder, except the subtracted number is in 2's complement.

Example of operation

Denary

5+9=14

- X=5, Y=9, Sum=14

Binary

0101+1001=1110

Addition of each step

Inputs			Outputs	
Cin	**X**	**Y**	**Sum**	**Cout**
0	1	1	0	1
1	0	0	1	0
0	1	0	1	0
0	0	1	1	0

**addition starts from lowest*

Result=1110 or 14

Chapter- 2

Carry Look-ahead Adder and Subtractor

Carry look-ahead adder

A **carry look-ahead adder** is a type of adder used in digital logic. A carry look-ahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, *ripple carry adder* for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits. The carry look-ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits. The Kogge-Stone adder and Brent-Kung adder are examples of this type of adder.

Theory of Operation

A ripple-carry adder works in the same way as pencil-and-paper methods of addition. Starting at the rightmost (least significant) digit position, the two corresponding digits are added and a result obtained. It is also possible that there may be a carry out of this digit position (for example, in pencil-and-paper methods, "9+5=4, carry 1"). Accordingly all digit positions other than the rightmost need to take into account the possibility of having to add an extra 1, from a carry that has come in from the next position to the right.

This means that no digit position can have an absolutely final value until it has been established whether or not a carry is coming in from the right. Moreover, unless the sum without a carry is 9 (in pencil-and-paper methods) or 1 (in binary arithmetic), it is not even possible to tell whether or not a given digit position is going to pass on a carry to the position on its left. At worst, when a whole sequence of sums comes to ...99999999... (in decimal) or ...11111111... (in binary), nothing can be deduced at all until the value of the carry coming in from the right is known, and that carry is then propagated to the left, one step at a time, as each digit position evaluated "9+1=0, carry 1" or "1+1=0, carry 1". It is the "rippling" of the carry from right to left that gives a ripple-carry adder its name, and its slowness. When adding 32-bit integers, for instance, allowance has to be made for the possibility that a carry could have to ripple through every one of the 32 one-bit adders.

Carry lookahead depends on two things:

1. Calculating, for each digit position, whether that position is going to propagate a carry if one comes in from the right.
2. Combining these calculated values so as to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Supposing that groups of 4 digits are chosen. Then the sequence of events goes something like this:

1. All 1-bit adders calculate their results. Simultaneously, the lookahead units perform their calculations.
2. Suppose that a carry arises in a particular group. Within at most 3 gate delays, that carry will emerge at the left-hand end of the group and start propagating through the group to its left.
3. If that carry is going to propagate all the way through the next group, the lookahead unit will already have deduced this. Accordingly, *before the carry emerges from the next group* the lookahead unit is immediately (within 1 gate delay) able to tell the *next* group to the left that it is going to receive a carry - and, at the same time, to tell the next lookahead unit to the left that a carry is on its way.

The net effect is that the carries start by propagating slowly through each 4-bit group, just as in a ripple-carry system, but then move 4 times as fast, leaping from one lookahead carry unit to the next. Finally, within each group that receives a carry, the carry propagates slowly within the digits in that group.

The more bits in a group, the more complex the lookahead carry logic becomes, and the more time is spent on the "slow roads" in each group rather than on the "fast road" between the groups (provided by the lookahead carry logic). On the other hand, the fewer bits there are in a group, the more groups have to be traversed to get from one end of a number to the other, and the less acceleration is obtained as a result.

Deciding the group size to be governed by lookahead carry logic requires a detailed analysis of gate and propagation delays for the particular technology being used.

It is possible to have more than one level of lookahead carry logic, and this is in fact usually done. Each lookahead carry unit already produces a signal saying "if a carry comes in from the right, I will propagate it to the left", and those signals can be combined so that each group of (let us say) four lookahead carry units becomes part of a "supergroup" governing a total of 16 bits of the numbers being added. The "supergroup" lookahead carry logic will be able to say whether a carry entering the supergroup will be propagated all the way through it, and using this information, it is able to propagate carries from right to left 16 times as fast as a naive ripple carry. With this kind of two-level implementation, a carry may first propagate through the "slow road" of individual

adders, then, on reaching the left-hand end of its group, propagate through the "fast road" of 4-bit lookahead carry logic, then, on reaching the left-hand end of its supergroup, propagate through the "superfast road" of 16-bit lookahead carry logic.

Again, the group sizes to be chosen depend on the exact details of how fast signals propagate within logic gates and from one logic gate to another.

For very large numbers (hundreds or even thousands of bits) lookahead carry logic does not become any more complex, because more layers of supergroups and supersupergroups can be added as necessary. The increase in the number of gates is also moderate: if all the group sizes are 4, one would end up with one third as many lookahead carry units as there are adders. However, the "slow roads" on the way to the faster levels begin to impose a drag on the whole system (for instance, a 256-bit adder could have up to 24 gate delays in its carry processing), and the mere physical transmission of signals from one end of a long number to the other begins to be a problem. At these sizes carry-save adders are preferable, since they spend no time on carry propagation at all.

Carry lookahead method

Carry lookahead logic uses the concepts of *generating* and *propagating* carries. Although in the context of a carry lookahead adder, it is most natural to think of generating and propagating in the context of binary addition, the concepts can be used more generally than this. In the descriptions below, the word *digit* can be replaced by *bit* when referring to binary addition.

The addition of two 1-digit inputs *A* and *B* is said to *generate* if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition 52 + 67, the addition of the tens digits 5 and 6 *generates* because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit does not carry (2+7=9)).

In the case of binary addition, $A \cdot B$generates if and only if both *A* and *B* are 1. If we write *G*(*A*,*B*) to represent the binary predicate that is true if and only if $A \cdot B$generates, we have:

$$G(A, B) = A \cdot B$$

The addition of two 1-digit inputs *A* and *B* is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

In the case of binary addition, $A + B$ propagates if and only if at least one of A or B is 1. If we write $P(A,B)$ to represent the binary predicate that is true if and only if $A + B$ propagates, we have:

$$P(A, B) = A + B$$

Sometimes a slightly different definition of *propagate* is used. By this definition $A + B$ is said to propagate if the addition will carry whenever there is an input carry, but will not carry if there is no input carry. It turns out that the way in which generate and propagate bits are used by the carry lookahead logic, it doesn't matter which definition is used. In the case of binary addition, this definition is expressed by:

$$P'(A, B) = A \oplus B$$

For binary arithmetic, *or* is faster than *xor* and takes fewer transistors to implement. However, for a multiple-level carry lookahead adder, it is simpler to use $P'(A, B)$.

Given these concepts of generate and propagate, when will a digit of addition carry? It will carry precisely when either the addition generates *or* the next less significant bit carries and the addition propagates. Written in boolean algebra, with C_i the carry bit of digit i, and P_i and G_i the propagate and generate bits of digit i respectively,

$$C_{i+1} = G_i + (P_i \cdot C_i)$$

Implementation details

For each bit in a binary sequence to be added, the Carry Look Ahead Logic will determine whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. Then, when the actual addition is performed, there is no delay from waiting for the ripple carry effect (or time it takes for the carry from the first Full Adder to be passed down to the last Full Adder). Below is a simple 4-bit generalized Carry Look Ahead circuit that combines with the 4-bit Ripple Carry Adder we used above with some slight adjustments:

For the example provided, the logic for the generate (g) and propagate (p) values are given below. Note that the numeric value determines the signal from the circuit above, starting from 0 on the far left to 3 on the far right:

$$\begin{aligned} C_1 &= G_0 + P_0 \cdot C_0 \\ C_2 &= G_1 + P_1 \cdot C_1 \\ C_3 &= G_2 + P_2 \cdot C_2 \\ C_4 &= G_3 + P_3 \cdot C_3 \end{aligned}$$

Substituting C_1 into C_2, then C_2 into C_3, then C_3 into C_4 yields the expanded equations:

$$C_1 = G_0 + P_0 \cdot C_0$$
$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1$$
$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2$$
$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

To determine whether a bit pair will generate a carry, the following logic works:

$$G_i = A_i \cdot B_i$$

To determine whether a bit pair will propagate a carry, either of the following logic statements work:

$$P_i = A_i \oplus B_i$$
$$P_i = A_i + B_i$$

The reason why this works is based on evaluation of $C_1 = G_0 + P_0 \cdot C_0$. The only difference in the truth tables between ($A \oplus B$) and ($A + B$) is when both A and B are 1. However, if both A and B are 1, then the G_0 term is 1 (since its equation is $A \cdot B$), and the $P_0 \cdot C_0$ term becomes irrelevant. The XOR is used normally within a basic full adder circuit; the OR is an alternate option (for a carry lookahead only) which is far simpler in transistor-count terms.

The Carry Look Ahead 4-bit adder can also be used in a higher-level circuit by having each CLA Logic circuit produce a propagate and generate signal to a higher-level CLA Logic circuit. The group propagate (PG) and group generate (GG) for a 4-bit CLA are:

$$PG = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$GG = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

Putting 4 4-bit CLAs together yields four group propagates and four group generates. A Lookahead Carry Unit (LCU) takes these 8 values and uses identical logic to calculate C_i in the CLAs. The LCU then generates the carry input for each of the 4 CLAs and a fifth equal to C_{16}.

The calculation of the gate delay of a 16-bit adder (using 4 CLAs and 1 LCU) is not as straight forward as the ripple carry adder. Starting at time of zero:

- calculation of P_i and G_i is done at time 1
- calculation of C_i is done at time 3
- calculation of the PG is done at time 2
- calculation of the GG is done at time 3
- calculation of the inputs for the CLAs from the LCU are done at
  - time 0 for the first CLA
  - time 5 for the second CLA
  - time 5 for the third & fourth CLA
- calculation of the S_i are done at

    - time 4 for the first CLA
    - time 8 for the second CLA
    - time 8 for the third & fourth CLA
- calculation of the final carry bit (C_{16}) is done at time 5

The maximum time is 8 gate delays (for $S_{[8-15]}$). A standard 16-bit ripple carry adder would take 31 gate delays.

Manchester carry chain

The Manchester carry chain is a variation of the carry look-ahead adder that uses shared logic to lower the transistor count. As can be seen above in the implementation section, the logic for generating each carry contains all of the logic used to generate the previous carries. A Manchester carry chain generates the intermediate carries by tapping off nodes in the gate that calculates the most significant carry value. Not all logic families have these internal nodes, however, CMOS being a major example. Dynamic logic can support shared logic, as can transmission gate logic. One of the major downsides of the Manchester carry chain is that the capacitive load of all of these outputs, together with the resistance of the transistors causes the propagation delay to increase much more quickly than a regular carry look-ahead. A Manchester carry chain section generally won't exceed 4 bits.

Subtractor

In electronics, a **subtractor** can be designed using the same approach as that of an adder. The binary subtraction process is summarized below. As with an adder, in the general case of calculations on multi-bit numbers, three bits are involved in performing the subtraction for each bit of the difference: the minuend (X_i), subtrahend (Y_i), and a borrow in from the previous (less significant) bit order position (B_i). The outputs are the difference bit (D_i) and borrow bit B_{i+1}.

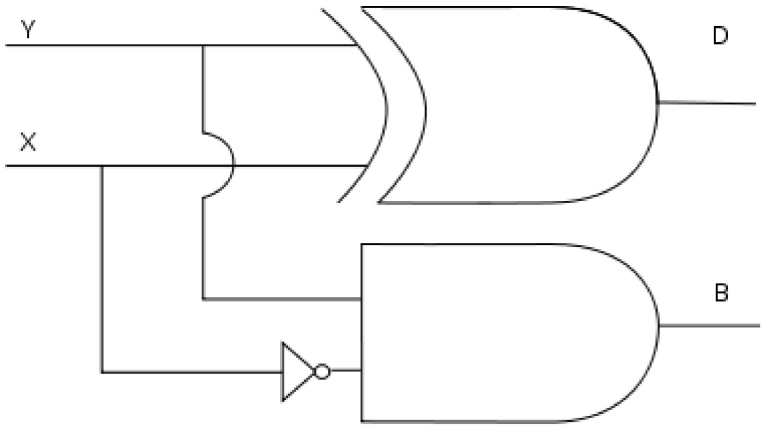
$$D_i = X_i \oplus Y_i \oplus B_i$$

K-map $B_i(1,2,4,7)$

Subtractors are usually implemented within a binary adder for only a small cost when using the standard two's complement notation, by providing an addition/subtraction selector to the carry-in and to invert the second operand.

$$-B = \bar{B} + 1 \text{(definition of two's complement negation)}$$
$$\begin{aligned} A - B &= A + (-B) \\ &= A + \bar{B} + 1 \end{aligned}$$

Half subtractor



Logic diagram for a half subtractor

The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow).

Truth table

The truth table for the half subtractor is given below.

X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

From the above table one can draw the Karnaugh map for "difference" and "borrow".

So, Logic equations are:

$$D = X \oplus Y$$
$$B = \overline{X} \cdot Y$$

Full subtractor

The Full_subtractor is a combinational circuit which is used to perform subtraction of three bits. It has three inputs, X (minuend) and Y (subtrahend) and Z (subtrahend) and two outputs D (difference) and B (borrow).
Easy way to write truth table
D=X-Y-Z (don't bother about sign)
B = 1 If X<(Y+Z)

Truth table

The truth table for the full subtractor is given below.

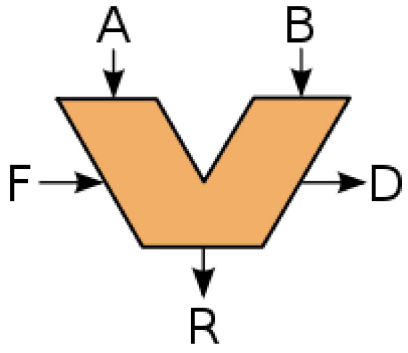
X	Y	Z	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

So, Logic equations are:

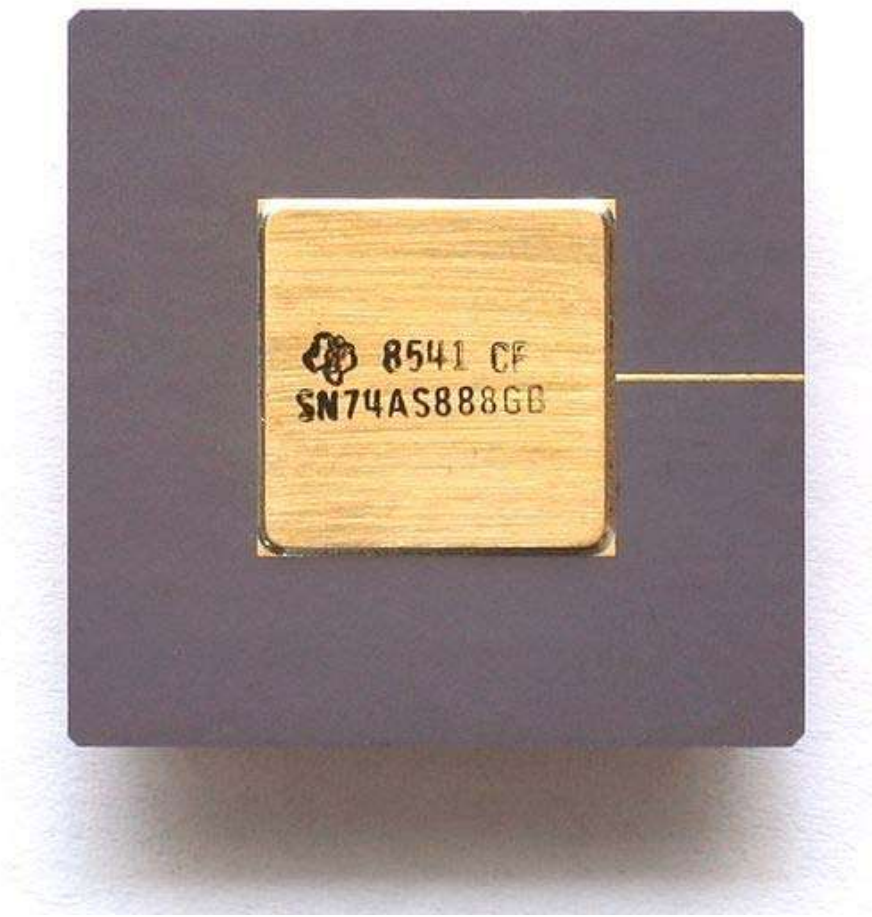
$$D = X \oplus Y \oplus Z$$
$$B = Z \cdot (\overline{X \oplus Y}) + \overline{X} \cdot Y$$

Chapter- 3

Arithmetic Logic Unit



Arithmetic Logic Unit schematic symbol



Cascadable 8 Bit ALU Texas Instruments SN74AS888

In computing, an **arithmetic logic unit** (**ALU**) is a digital circuit that performs arithmetic and logical operations. The ALU is a fundamental building block of the central processing unit (CPU) of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. The processors found inside modern CPUs and graphics processing units (GPUs) accommodate very powerful and very complex ALUs; a single component may contain a number of ALUs.

Mathematician John von Neumann proposed the ALU concept in 1945, when he wrote a report on the foundations for a new computer called the EDVAC. Research into ALUs remains an important part of computer science, falling under **Arithmetic and logic structures** in the ACM Computing Classification System.

Numerical systems

An ALU must process numbers using the same format as the rest of the digital circuit. The format of modern processors is almost always the two's complement binary number representation. Early computers used a wide variety of number systems, including ones' complement, two's complement sign-magnitude format, and even true decimal systems, with ten tubes per digit.

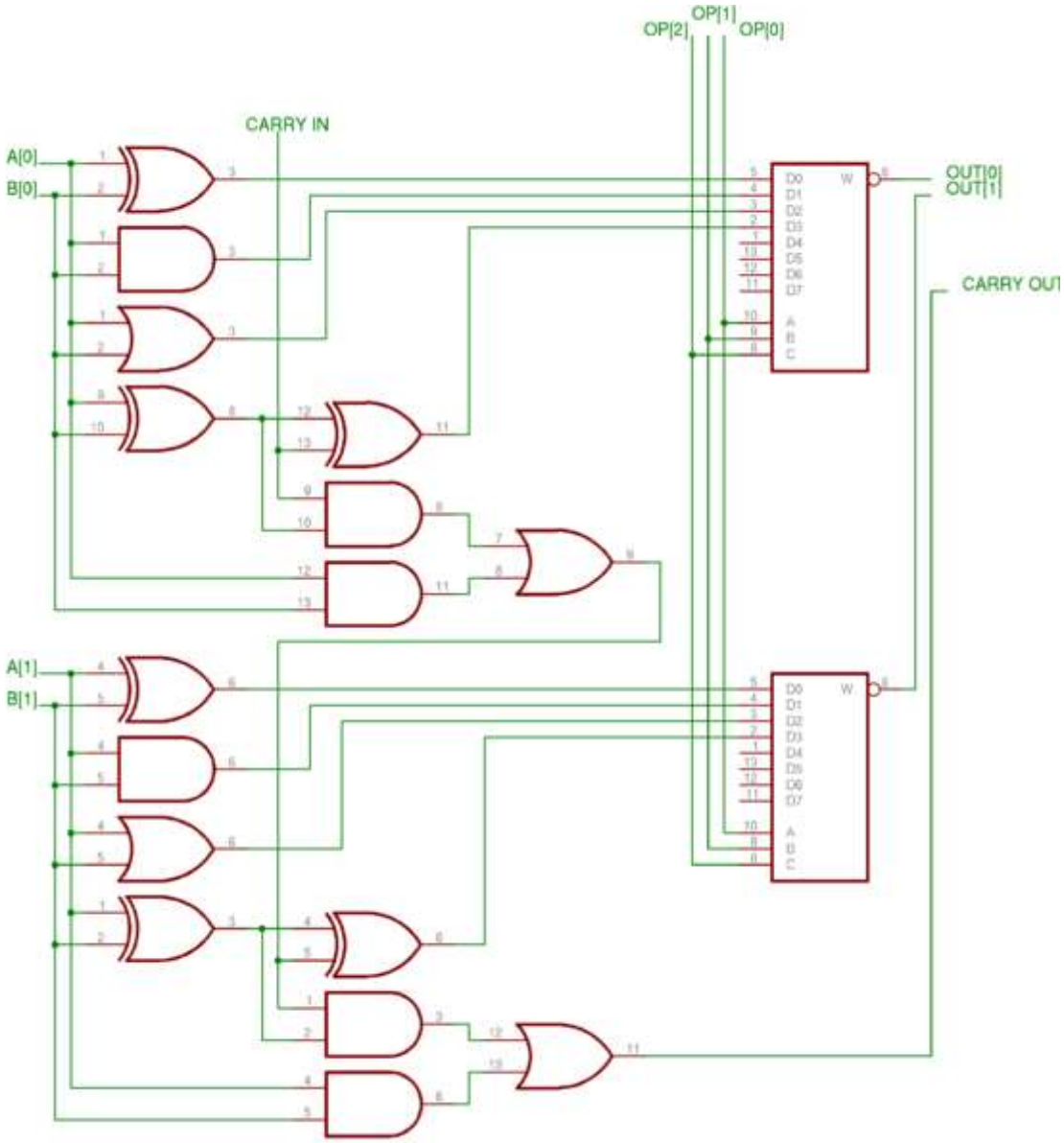
ALUs for each one of these numeric systems had different designs, and that influenced the current preference for two's complement, as this is the representation that makes it easier for the ALUs to calculate additions and subtractions.

The ones' complement and two's complement number systems allow for subtraction to be accomplished by adding the negative of a number in a very simple way which negates the need for specialized circuits to do subtraction; however, calculating the negative in two's complement requires adding a one to the low order bit and propagating the carry. An alternative way to do two's complement subtraction of A−B is to present a one to the carry input of the adder and use ¬B rather than B as the second input.

Practical overview

Most of a processor's operations are performed by one or more ALUs. An ALU loads data from input registers, an external Control Unit then tells the ALU what operation to perform on that data, and then the ALU stores its result into an output register. The Control Unit is responsible for moving the processed data between these registers, ALU and memory.

Simple operations



A simple example arithmetic logic unit (2-bit ALU) that does AND, OR, XOR, and addition

Most ALUs can perform the following operations:

- Integer arithmetic operations (addition, subtraction, and sometimes multiplication and division, though this is more expensive)
- Bitwise logic operations (AND, NOT, OR, XOR)
- Bit-shifting operations (shifting or rotating a word by a specified number of bits to the left or right, with or without sign extension). Shifts can be interpreted as multiplications by 2 and divisions by 2.

Complex operations

Engineers can design an Arithmetic Logic Unit to calculate any operation. The more complex the operation, the more expensive the ALU is, the more space it uses in the processor, the more power it dissipates. Therefore, engineers compromise. They make the ALU powerful enough to make the processor fast, but yet not so complex as to become prohibitive. For example, computing the square root of a number might use:

1. **Calculation in a single clock** Design an extraordinarily complex ALU that calculates the square root of any number in a single step.
2. **Calculation pipeline** Design a very complex ALU that calculates the square root of any number in several steps. The intermediate results go through a series of circuits arranged like a factory production line. The ALU can accept new numbers to calculate even before having finished the previous ones. The ALU can now produce numbers as fast as a single-clock ALU, although the results start to flow out of the ALU only after an initial delay.
3. **interactive calculation** Design a complex ALU that calculates the square root through several steps. This usually relies on control from a complex control unit with built-in microcode.
4. **Co-processor** Design a simple ALU in the processor, and sell a separate specialized and costly processor that the customer can install just beside this one, and implements one of the options above.
5. **Software libraries** Tell the programmers that there is no co-processor and there is no emulation, so they will have to write their own algorithms to calculate square roots by software.
6. **Software emulation** Emulate the existence of the co-processor, that is, whenever a program attempts to perform the square root calculation, make the processor check if there is a co-processor present and use it if there is one; if there isn't one, interrupt the processing of the program and invoke the operating system to perform the square root calculation through some software algorithm.

The options above go from the fastest and most expensive one to the slowest and least expensive one. Therefore, while even the simplest computer can calculate the most complicated formula, the simplest computers will usually take a long time doing that because of the several steps for calculating the formula.

Powerful processors like the Intel Core and AMD64 implement option #1 for several simple operations, #2 for the most common complex operations and #3 for the extremely complex operations.

Inputs and outputs

The inputs to the ALU are the data to be operated on (called operands) and a code from the control unit indicating which operation to perform. Its output is the result of the computation.

In many designs the ALU also takes or generates as inputs or outputs a set of condition codes from or to a status register. These codes are used to indicate cases such as carry-in or carry-out, overflow, divide-by-zero, etc.

ALUs vs. FPUs

A Floating Point Unit also performs arithmetic operations between two values, but they do so for numbers in floating point representation, which is much more complicated than the two's complement representation used in a typical ALU. In order to do these calculations, a FPU has several complex circuits built-in, including some internal ALUs.

In modern practice, engineers typically refer to the ALU as the circuit that performs integer arithmetic operations (like two's complement and BCD). Circuits that calculate more complex formats like floating point, complex numbers, etc. usually receive a more specific name such as FPU.

Chapter- 4

Binary Multiplier

A **binary multiplier** is an electronic circuit used in digital electronics, such as a computer, to multiply two binary numbers. It is built using binary adders.

A variety of computer arithmetic techniques can be used to implement a digital multiplier. Most techniques involve computing a set of *partial products*, and then summing the partial products together. This process is similar to the method taught to primary schoolchildren for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system.

History

Until the late 1970s, most minicomputers did not have a multiply instruction, and so programmers used a "multiply routine" which repeatedly shifts and accumulates partial results, often written using loop unwinding. Mainframe computers had multiply instructions, but they did the same sorts of shifts and adds as a "multiply routine".

Early microprocessors also had no multiply instruction. The Motorola 6809, introduced circa 1977-78, was one of the earliest microprocessors with a dedicated hardware multiply instruction. It apparently did the same sorts of shifts and adds as a "multiply routine", but implemented in the microcode of the MUL instruction.

As more transistors per chip became available (Moore's law), it became possible to put enough adders on a single chip to sum all the partial products at once, rather than re-use a single adder to handle each partial product one at a time.

Because some common digital signal processing algorithms spend most of their time multiplying, people who design digital signal processors sacrifice a lot of chip area in order to make the "multiply" as fast as possible—a single-cycle multiply-accumulate unit often used up most of the chip area of early DSPs.

Multiplication basics

The method taught in school for multiplying decimal numbers, is based on calculating partial products, shifting them to the left and then adding them together. The most difficult part is to obtain the partial products, as that involves multiplying a long number by one digit (from 0 to 9):

```
    123
  x 456
  =====
    738  (this is 123 x 6)
   615   (this is 123 x 5, shifted one position to the left)
+ 492    (this is 123 x 4, shifted two positions to the left)
  =====
  56088
```

A binary computer does exactly the same, but with binary numbers. In binary encoding each long number is multiplied by one digit (either 0 or 1), and that is much easier than in decimal, as the product by 0 or 1 is just 0 or the same number. Therefore, the multiplication of two binary numbers comes down to calculating partial products (which are 0 or the first number), shifting them left, and then adding them together (a binary addition, of course):

```
      1011   (this is 11 in binary)
    x 1110   (this is 14 in binary)
    ======
      0000   (this is 1011 x 0)
     1011    (this is 1011 x 1, shifted one position to the left)
    1011     (this is 1011 x 1, shifted two positions to the left)
+ 1011       (this is 1011 x 1, shifted three positions to the left)
=========
 10011010   (this is 154 in binary)
```

This is much simpler than in the decimal system, as there is no table of multiplication to remember: just shifts and adds.

This method is mathematically correct, but it has two serious engineering problems. The first is that it involves 32 intermediate additions in a 32-bit computer, or 64 intermediate additions in a 64-bit computer. These additions take a lot of time. The engineering implementation of binary multiplication consists, really, of taking a very simple mathematical process and complicating it a lot, in order to do fewer additions; a modern processor can multiply two 64-bit numbers with 16 additions (rather than 64), and can do several steps in parallel—but at a cost of making the process almost unreadable.

The second problem is that the basic school method handles the sign with a separate rule ("+ with + yields +", "+ with - yields -", etc.). Modern computers embed the sign of the number in the number itself, usually in the two's complement representation. That forces the multiplication process to be adapted to handle two's complement numbers, and that complicates the process a bit more. Similarly, processors that use one's complement,

sign-and-magnitude, IEEE-754 or other binary representations require specific adjustments to the multiplication process.

Engineering approach: an unsigned example

For example, suppose we want to multiply two unsigned eight bit integers together: *a*[7:0] and *b*[7:0]. We can produce eight partial products by performing eight one-bit multiplications, one for each bit in multiplicand *a*:

```
p0[7:0] = a[0] × b[7:0] = {8{a[0]}} & b[7:0]
p1[7:0] = a × b[7:0] = {8{a}} & b[7:0]
p2[7:0] = a × b[7:0] = {8{a}} & b[7:0]
p3[7:0] = a × b[7:0] = {8{a}} & b[7:0]
p4[7:0] = a × b[7:0] = {8{a}} & b[7:0]
p5[7:0] = a × b[7:0] = {8{a}} & b[7:0]
p6[7:0] = a × b[7:0] = {8{a}} & b[7:0]
p7[7:0] = a × b[7:0] = {8{a}} & b[7:0]
```

where {8{a[0]}} means repeating a[0] (the 0th bit of a) 8 times (Verilog notation).

To produce our product, we then need to add up all eight of our partial products, as shown here:

```
                                                    p0[7] p0[6] p0[5] p0[4]
p0[3] p0[2] p0[1] p0[0]
                                              + p1[7] p1[6] p1[5] p1[4] p1[3]
p1[2] p1[1] p1[0] 0
                                        + p2[7] p2[6] p2[5] p2[4] p2[3] p2[2]
p2[1] p2[0] 0     0
                                  + p3[7] p3[6] p3[5] p3[4] p3[3] p3[2] p3[1]
p3[0] 0     0     0
                            + p4[7] p4[6] p4[5] p4[4] p4[3] p4[2] p4[1] p4[0]
0     0     0     0
                      + p5[7] p5[6] p5[5] p5[4] p5[3] p5[2] p5[1] p5[0] 0
0     0     0     0
                + p6[7] p6[6] p6[5] p6[4] p6[3] p6[2] p6[1] p6[0] 0     0
0     0     0     0
          + p7[7] p7[6] p7[5] p7[4] p7[3] p7[2] p7[1] p7[0] 0     0     0
0     0     0     0
-------------------------------------------------------------------------
--------------------
P[15] P[14] P[13] P[12] P[11] P[10]  P[9]  P[8]  P[7]  P[6]  P[5]  P[4]
P[3]  P[2]  P[1]  P[0]
```

In other words, *P*[15:0] is produced by summing *p*0, *p*1 << 1, *p*2 << 2, and so forth, to produce our final unsigned 16-bit product.

Engineering approach: signed integers

If *b* had been a signed integer instead of an unsigned integer, then the partial products would need to have been sign-extended up to the width of the product before summing. If

a had been a signed integer, then partial product *p7* would need to be subtracted from the final sum, rather than added to it.

The above array multiplier can be modified to support two's complement notation signed numbers by inverting several of the product terms and inserting a one to the left of the first partial product term:

```
                                                                  1  -p0[7]  p0[6]
p0[5]  p0[4]  p0[3]  p0[2]  p0[1]  p0[0]
                                                          -p1[7] +p1[6] +p1[5]
+p1[4] +p1[3] +p1[2] +p1[1] +p1[0]   0
                                                  -p2[7] +p2[6] +p2[5] +p2[4]
+p2[3] +p2[2] +p2[1] +p2[0]   0      0
                                          -p3[7] +p3[6] +p3[5] +p3[4] +p3[3]
+p3[2] +p3[1] +p3[0]   0      0      0
                                  -p4[7] +p4[6] +p4[5] +p4[4] +p4[3] +p4[2]
+p4[1] +p4[0]   0      0      0      0
                          -p5[7] +p5[6] +p5[5] +p5[4] +p5[3] +p5[2] +p5[1]
+p5[0]   0      0      0      0      0
                    -p6[7] +p6[6] +p6[5] +p6[4] +p6[3] +p6[2] +p6[1] +p6[0]
0      0      0      0      0      0
   1  +p7[7] -p7[6] -p7[5] -p7[4] -p7[3] -p7[2] -p7[1] -p7[0]   0
0      0      0      0      0      0
 ------------------------------------------------------------------------
-------------------------------------
P[15]  P[14]  P[13]  P[12]  P[11]  P[10]   P[9]   P[8]   P[7]   P[6]
P[5]   P[4]   P[3]   P[2]   P[1]  P[0]
```

Do not be misled by the minus sign (-) notation above. It does not mean arithmetic negation (-(7) = -7), but instead binary complementation, more commonly denoted x'(read x prime), $\bar{x}$(x bar) or ~*x* (tilde x), achieved by flipping all the bits. In two's complement to get the negation of a number, you complement the number then add 1, so they are NOT equivalent.

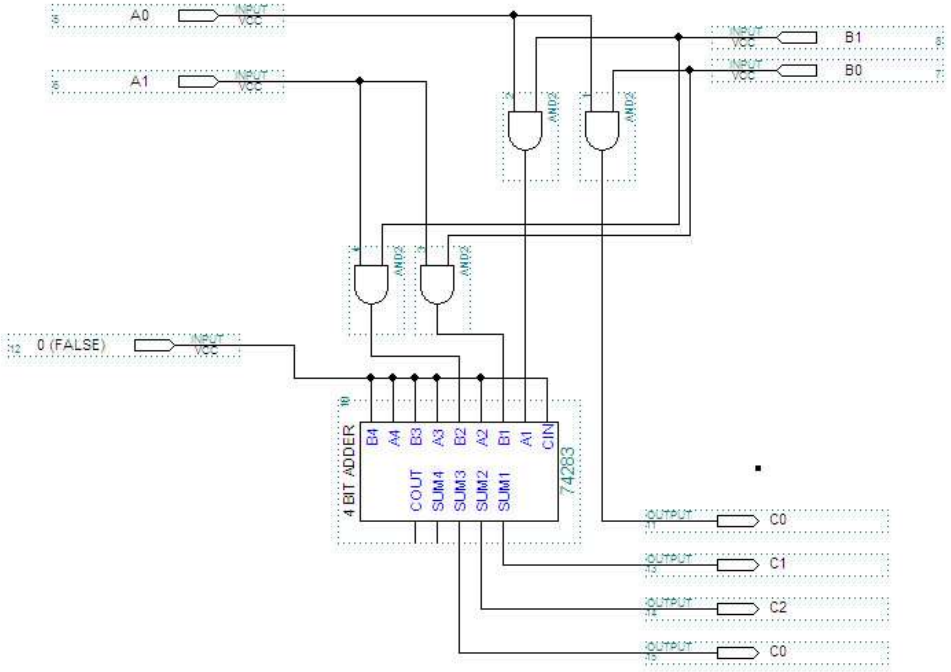
There are a lot of simplifications in the bit array above that are not shown and are not obvious. The sequences of one complemented bit followed by noncomplemented bits are implementing a two's complement trick to avoid sign extension. The sequence of p7 (noncomplemented bit followed by all complemented bits) is because we're subtracting this term so they were all negated to start out with (and a 1 was added in the least significant position). For both types of sequences, the last bit is flipped and an implicit -1 should be added directly below the MSB. When the +1 from the two's complement negation for p7 in bit position 0 (LSB) and all the -1's in bit columns 7 through 14 (where each of the MSBs are located) are added together, they can be simplified to the single 1 that "magically" is floating out to the left.

Implementations

Older multiplier architectures employed a shifter and accumulator to sum each partial product, often one partial product per cycle, trading off speed for die area. Modern multiplier architectures use the Baugh–Wooley algorithm, Wallace trees, or Dadda

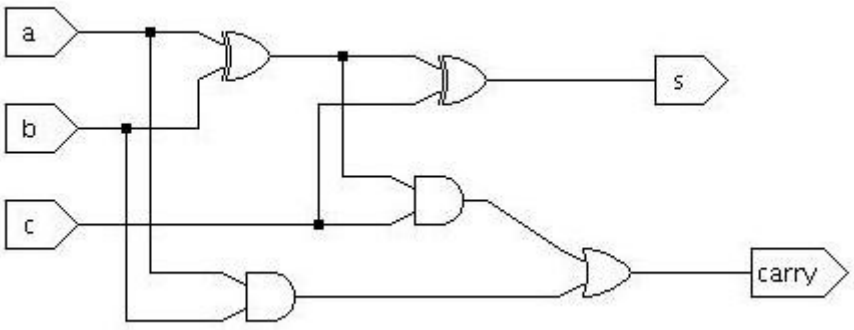
multipliers to add the partial products together in a single cycle. The performance of the Wallace tree implementation is sometimes improved by *modified* Booth encoding one of the two multiplicands, which reduces the number of partial products that must be summed.

Example

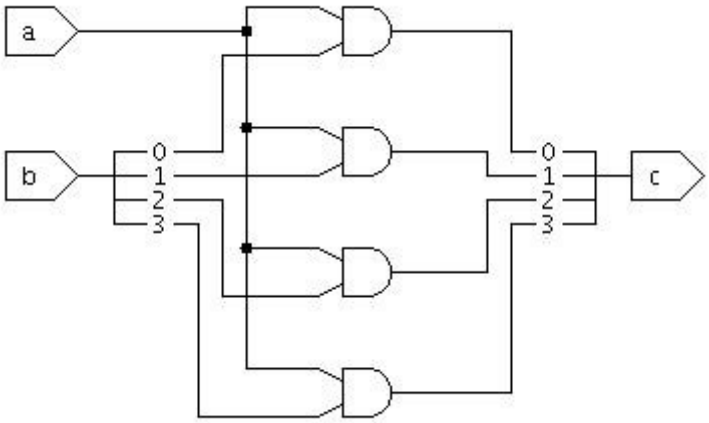


2 Bit by 2 Bit Binary Multiplier
Using a 4 Bit + 4 Bit Adder

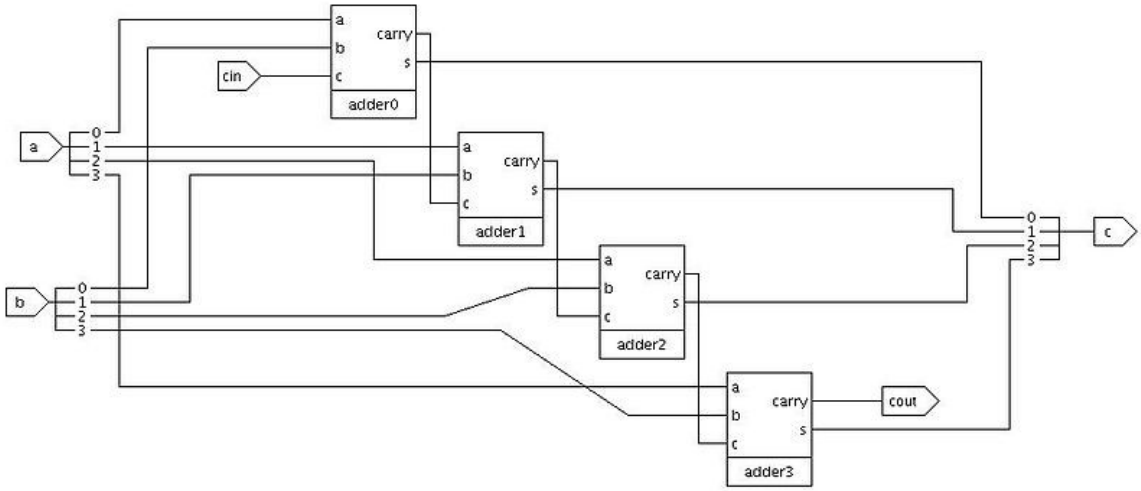
(The following are to clarify each level of abstraction)



A simple adder
1-bit adder



"Ander"
A 1x4 bit ander

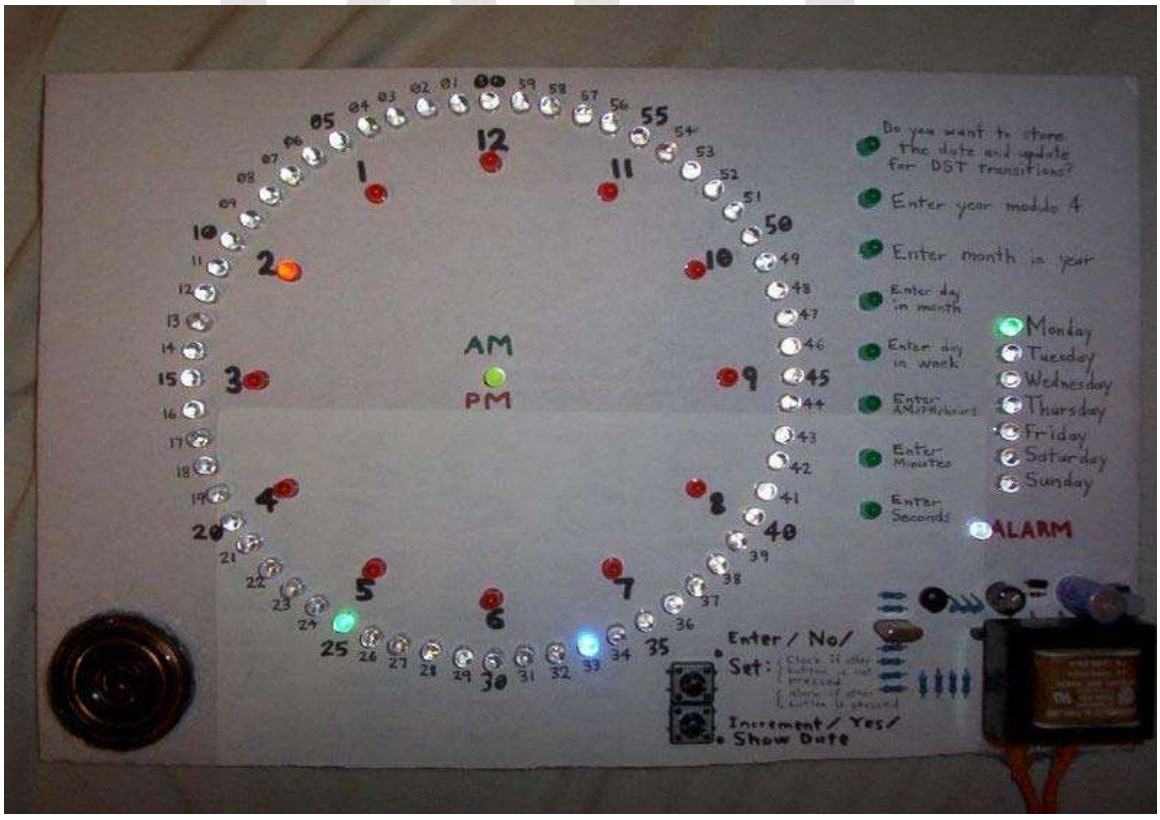


4-bit Adder
Using 4 1-bit adders

Chapter- 5

Charlieplexing

Charlieplexing is a technique proposed in early 1995 by Charlie Allen at Maxim Integrated Products for driving a multiplexed display in which relatively few I/O pins on a microcontroller are used to drive an array of LEDs. The method utilizes the tri-state logic capabilities of microcontrollers in order to gain efficiency over traditional multiplexing. Although it is more efficient in its use of I/O, there are issues that cause it to be more complicated to design and render it impractical for larger displays. These issues include duty cycle, current requirements and the forward voltages of the LEDs.



A Charlieplexed digital clock which controls 90 LEDs with 10 pins of a PIC16C54 microcontroller.

Traditional multiplexing

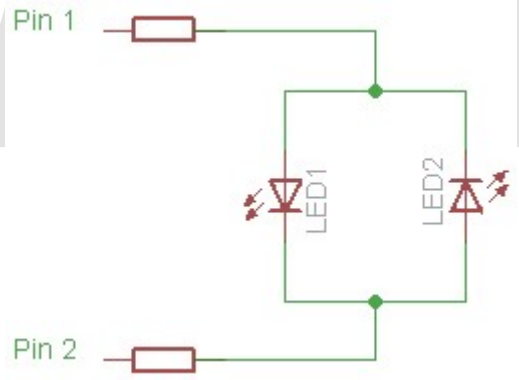
Display multiplexing is very different from multiplexing used in data transmission, although it has the same basic principles. In display multiplexing, the data lines of the displays are connected together in parallel to a common bus on the microcontroller. Then, the displays are turned on individually and addresses when they turn on. This allows one to use fewer I/O pins than it would normally take to drive the same number of displays directly.

When using charlieplexing, *n* drive pins can drive *n* digits with *n-1* segments. When simplified, it equates to *n* pins being able to drive n^2-*n* segments or LEDs. Traditional multiplexing takes many more pins to drive the same number of LEDs; *2n* pins must be used to drive n^2 LEDs (though a 1-of-n decoder chip can be used to reduce the number of microcontroller I/O pins to $n + \lceil \log_2 n \rceil$).

Charlieplexing

Complementary drive

Charlieplexing, in its simplest form, works using a matrix of complementary pairs of LEDs. The simplest possible charlieplexed matrix would look like this:

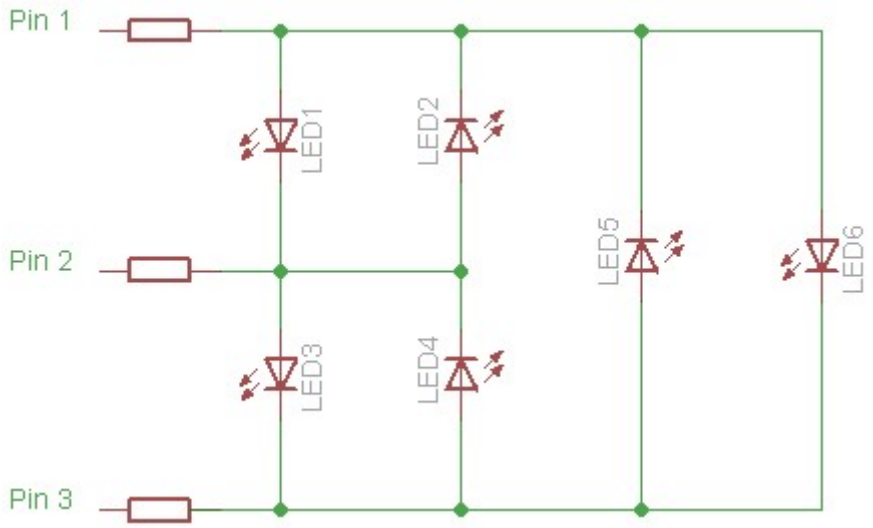


By applying a positive charge to pin 1 and grounding pin 2, LED 1 will light. Since current cannot flow through LEDs in reverse bias, LED 2 will remain unlit. If the charges on pin 1 and pin 2 are reversed, LED 2 will light and LED 1 will be unlit.

The Charlieplexing technique does not actually make a larger matrix possible when only using two pins, because two LEDs can be driven by two pins without any matrix connections, and without even using tri-state mode. However, the 2-pin circuit serves as a simple example to show the basic concepts before moving on to larger circuits where Charlieplexing actually shows an advantage.

Expanding: tri-state logic

If we were to expand this circuit to accommodate 3 pins and 6 LEDs, it would look like this:



This presents a problem however. In order for this circuit to act like the previous one, we must disconnect one of the pins before applying charge to the remaining two. This can be solved by utilizing the tri-state logic properties of microcontroller pins. Microcontroller pins generally have three states, 5 V, 0 V and input. Input mode puts the pin into a high impedance state which, electrically speaking, “disconnects” that pin from the circuit, meaning little or no current will flow through it. This allows for the circuit to see any number of pins connected at any time, simply by changing the state of the pin. In order to drive the 6 LED matrix above, the two pins corresponding to the LED we wish to light are connected to 5 V and 0 V while the third pin is set in its input state. In doing so, we prevent current leakage out of the third pin, ensuring that the LED we wish to light is the only one lit.

By using tri-state logic, the matrix can theoretically be expanded to any size, as long as pins are available. For n pins, you can have up to n(n-1) LEDs in the matrix. Any LED can be lit by applying 5 V and 0 V to its corresponding pins and setting all of the other pins connected to the matrix to input mode.

Problems with charlieplexing

Refresh rate

Because only a single set of LEDs, all having a common anode or cathode, can be lit simultaneously without turning on unintended LEDs, charlieplexing requires frequent output changes, through a method known as flickering. When flickering is done, not all LEDs are lit quite simultaneously, but rather one set of LEDs is lit briefly, then another set, then another, and eventually the cycle repeats. If it is done fast enough, they will

appear to all be on, all the time, to the human eye. In order for a display to not have any noticeable flicker, the refresh rate for each LED must be greater than 50 Hz. Suppose 8 bits are used to control 56 LEDs via charlieplexing, which is enough for 8 7-segment displays (without decimal points). Typically 7-segment displays are made to have a common cathode, sometimes a common anode, but without loss of generality suppose it is a common cathode. All LEDs in all 8 7-segment displays cannot be turned on simultaneously in any desired combination via charlieplexing. It is impossible to get 56 bits of information directly from 8 bits of information. Instead, the human eye must be fooled by use of a flicker. Only one 7-segment display, one set of 7 LEDs can be active at any time. The way this would be done is for the 8 common cathodes of the 8 displays to each get assigned to its own unique pin among the 8 I/O ports. At any time, one and only one of the 8 controlling I/O pins will be actively low, and thus only the 7-segment display with its common cathode connected to that actively low pin can have any of its LEDs on. That is the active 7-segment display. The anodes of the 7 LED segments within the active 7-segment display can then be turned on in any combination by having the other 7 I/O ports either high or in high-impedance mode, in any combination. They are connected to the remaining 7 pins, but through resistors (the common cathode connection is connected to the pin itself, not through a resistor, because otherwise the current through each individual segment would depend on the number of total segments turned on, as they'd all have to share a single resistor). But to show a desired number using all 8 digits, only one 7-segment display can be shown at a time, so all 8 must be cycled through separately, and in a 50th of a second for the entire period of 8. Thus the display must be refreshed at 400 Hz for the period-8 cycle through all 8 segments to make the LEDs flash no slower than 50 times per second. This requires constant interruption of whatever additional processing the controller performs, 400 times per second.

Peak current

Due to the decreased duty cycle, the current requirement of a charlieplexed display increases much faster than it would with a traditionally multiplexed display. As the display gets larger, the average current flowing through the LED must be (roughly) constant in order for it to maintain constant brightness, thus requiring the peak current to increase proportionally. This causes a number of issues that limit the practical size of a charlieplexed display.

- LEDs often have a maximum peak current rating as well as an average current rating.
- If the microcontroller code crashes, the LED left lit is under much higher stress than in a traditionally multiplexed display increasing the risk of a failure before the fault is spotted.

Requirement for tristate

All the outputs used to drive a charlieplexed display must be tristate. If the current is low enough to drive the displays directly off the I/O pins of the microcontroller this is no problem but if external tristates must be used then each tristate will generally require two

output lines to control eliminating most of the advantage of a charlieplexed display. Since the current from microcontroller pins is typically limited to 20 mA or so this severely restricts the practical size of a charlieplexed display. However, it can be done enabling one segment at the time.

Complexity

Charlieplex matrices are significantly more complicated, both in the required PC Board layout and microcontroller programming, than are traditional multiplex matrices. This increases design time.

Forward voltage

When using LEDs with different forward voltages, like when using different color LEDs, problems can exist where other LEDs will light when not wanted to.

If we look at the diagram above we notice that if LED 6 has a four volt forward voltage, and LED's 1 and 3 have forward voltages of two volts or less, they will light when LED 6 is intended to, as their current path is shorter. This issue can easily be avoided by checking the forward voltages of the LEDs used in the matrix and checking for compatibility issues. Or, more simply, using LEDs that have the same forward voltage.

LED failure

If a single LED fails, either by becoming an open circuit, by becoming a short-circuit, or becoming leaky (developing a parasitic parallel resistance which allows current in both directions), the impact will be catastrophic for the display as a whole and furthermore the actual problem LED may be notoriously difficult to identify, as not just a single but potentially large set of LEDs which should not be lit may all come on together, and without detailed knowledge of the circuit, the relation between which LED is bad and what set of LEDs all come on together cannot be easily established.

If the failed LED becomes an open circuit, the voltage between the LEDs 2 electrodes may build up until it finds a path through two LEDs. There are as many such paths as there are pins used to control the array minus 2; if the LED with anode at node m and cathode at node n fails in this way, it may be that every single pair of LEDs in which one's anode is node m, cathode is p for any value of p (with the exceptions that p cannot be m or n, so there are as many possible choices for p as the number of pins controlling the array minus 2), along with the LED whose anode is p and cathode is n, will all light up.

If there are 8 I/O pins controlling the array, this means there will be 6 parasitic paths through pairs of 2 LEDs, and 12 LEDs may be unintentionally lit, but fortunately this will only happen when the one bad LED is supposed to come on, which may be a small fraction of the time, and will exhibit no deleterious symptoms when the problem LED is not supposed to be lit. If the problem is a short between nodes x and y, then every time

any LED U with either x or y as its anode or cathode and some node z as its other electrode is supposed to come on (without loss of generality, suppose U's cathode is connected to x), the LED V with cathode y and anode z will light as well, so any time EITHER node x or y is activated as an anode OR a cathode, two LEDs will come on instead of one. In this case, it lights only one additional LED unintentionally, but it does it far more frequently; not merely when the failed LED is supposed to come on, but when ANY LED which has a pin in common with the failed LED is supposed to come on.

The problem elements become especially difficult to identify if there are two or more LEDs at fault. What this means is that unlike most methods in which the loss of a single LED merely causes a single burned-out segment, when charlieplexing is used, one or two burned-out LEDs, whatever the mode of failure, will almost certainly cause a catastrophic cascade of unintended lightings of the LEDs that still work, very likely rendering the entire device completely and immediately unusable. This must be taken into account when considering the required lifetime and failure characteristics of the device being designed.

Input data multiplexing

Charlieplexing can also be used to multiplex digital input signals into a microcontroller. The same diode circuits are used, except a switch is placed in series with each diode. To read whether a switch is open or closed, the microcontroller configures one pin as an input with an internal pull-up resistor. The other pin is configured as an output and set to logic-low. If the input pin reads low then the switch is closed, and if the input pin reads high then the switch is open.

One potential application for this is to read a standard (4x3) 12-key numeric keypad using only 4 I/O lines. The traditional row-column scan method requires 4 + 3 = 7 I/O lines. Thus charlieplexing saves 3 I/O lines; however it adds the expense of 12 diodes, (since the diodes are only free when LEDs are used). This reference shows a variation on the circuit that only needs 4 diodes, however that method qualifies as lossy compression, because when certain combinations of buttons are pressed simultaneously, those signals interfere with the microcontroller's ability to read certain other buttons. The microcontroller can always detect when the data is corrupt, but there is no guarantee it can sense the original key presses, unless only one button is pressed at a time. (However, it is probably possible to arrange the circuit so that if at most any two adjacent buttons are pressed, then no data loss will occur.) Basically though, the input is only loss-less on the 4 diode circuit if only one button is pressed at a time, or if certain problematic multiple key presses are avoided. In the 12 diode circuit, this is not an issue, and there is always a one-to-one correspondence between button presses and input data. However, there are so many diodes that are required to use the method (especially for larger arrays) that there is generally no cost savings over the traditional row-column scan method, unless for some reason the cost of a diode is only a fraction of the cost of an I/O pin, where that fraction is one over the number of I/O lines.

Chapter- 6

Counter

In digital logic and computing, a **counter** is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal.

Electronic counters

In electronics, counters can be implemented quite easily using register-type circuits such as the flip-flop, and a wide variety of classifications exist:

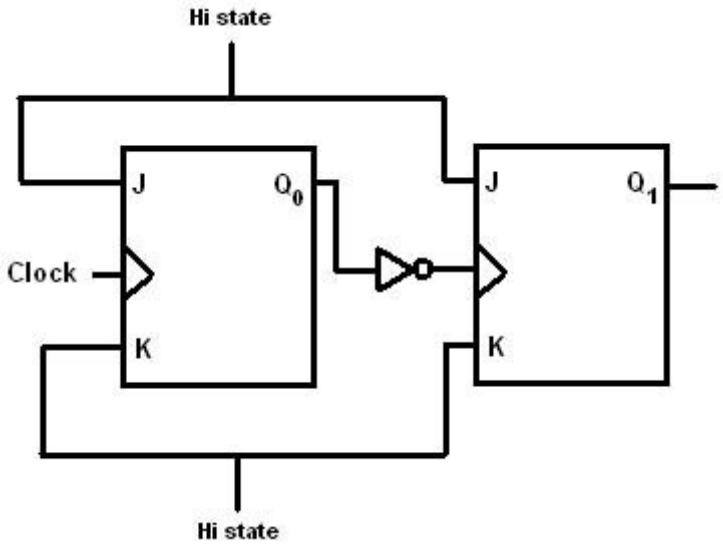
- Asynchronous (ripple) counter – changing state bits are used as clocks to subsequent state flip-flops
- Synchronous counter – all state bits change under control of a single clock
- Decade counter – counts through ten states per stage
- Up/down counter – counts both up and down, under command of a control input
- Ring counter – formed by a shift register with feedback connection in a ring
- Johnson counter – a *twisted* ring counter
- Cascaded counter

Each is useful for different applications. Usually, counter circuits are digital in nature, and count in natural binary. Many types of counter circuits are available as digital building blocks, for example a number of chips in the 4000 series implement different counters.

Occasionally there are advantages to using a counting sequence other than the natural binary sequence—such as the binary coded decimal counter, a linear feedback shift register counter, or a Gray-code counter.

Counters are useful for digital clocks and timers, and in oven timers, VCR clocks, etc.

Asynchronous (ripple) counter



Asynchronous counter created from two JK flip-flops

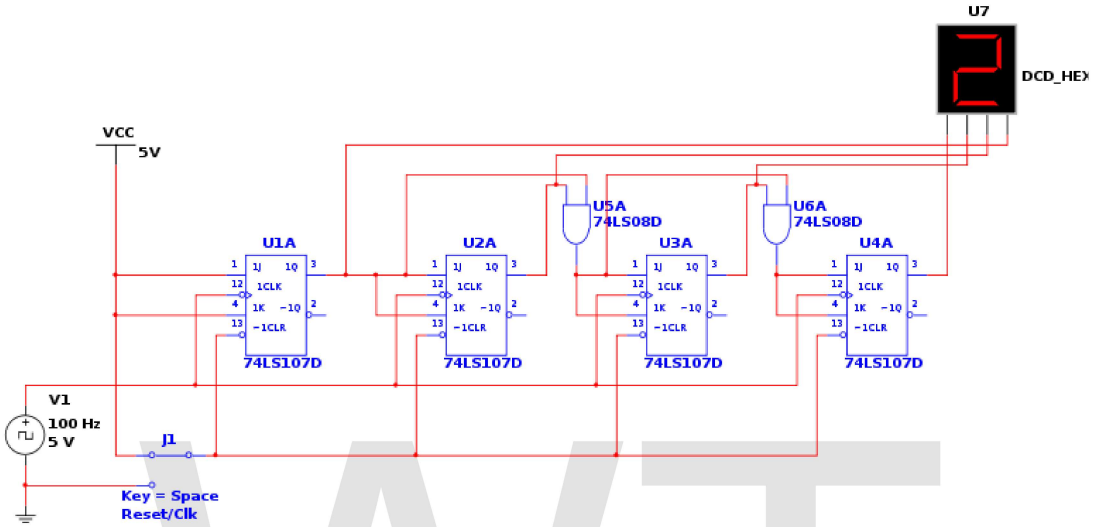
An asynchronous (ripple) counter is a single K-type flip-flop, with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock. If this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), you will get another 1 bit counter that counts half as fast. Putting them together yields a two-bit counter:

Cycle	Q1	Q0	(Q1:Q0)dec
0	0	0	0
1	0	1	1
2	1	0	2
3	1	1	3
4	0	0	0

You can continue to add additional flip-flops, always inverting the output to its own input, and using the output from the previous flip-flop as the clock signal. The result is called a ripple counter, which can count to 2^n-1 where n is the number of bits (flip-flop stages) in the counter. Ripple counters suffer from unstable outputs as the overflows "ripple" from stage to stage, but they do find frequent application as dividers for clock signals, where the instantaneous count is unimportant, but the division ratio overall is (to clarify this, a 1-bit counter is exactly equivalent to a divide by two circuit; the output frequency is exactly half that of the input when fed with a regular train of clock pulses).

The use of flip-flop outputs as clocks leads to timing skew between the count data bits, making this ripple technique incompatible with normal synchronous circuit design styles.

Synchronous counter



A 4-bit synchronous counter using SR flip-flops

A simple way of implementing the logic for each bit of an ascending counter (which is what is depicted in the image to the right) is for each bit to toggle when all of the less significant bits are at a logic high state. For example, bit 1 toggles when bit 0 is logic high; bit 2 toggles when both bit 1 and bit 0 are logic high; bit 3 toggles when bit 2, bit 1 and bit 0 are all high; and so on.

Synchronous counters can also be implemented with hardware finite state machines, which are more complex but allow for smoother, more stable transitions.

Hardware-based counters are of this type.

Decade counter

A decade counter is one that counts in decimal digits, rather than binary. A decade counter may have each digit binary encoded (that is, it may count in binary-coded decimal, as the 7490 integrated circuit did) or other binary encodings (such as the bi-quinary encoding of the 7490 integrated circuit). Alternatively, it may have a "fully decoded" or one-hot output code in which each output goes high in turn (the 4017 is such a circuit). The latter type of circuit finds applications in multiplexers and demultiplexers, or wherever a scanning type of behavior is useful. Similar counters with different numbers of outputs are also common.

The decade counter is also known as a mod-counter.

Up/down counter

A counter that can change state in either direction, under the control of an up/down selector input, is known as an up/down counter. When the selector is in the up state, the counter increments its value. When the selector is in the down state, the counter decrements the count.

Ring counter

A ring counter is a shift register (a cascade connection of flip-flops) with the output of the last one connected to the input of the first, that is, in a ring. Typically, a pattern consisting of a single bit is circulated so the state repeats every n clock cycles if n flip-flops are used. It can be used as a cycle counter of n states.

Johnson counter

A Johnson counter (or switchtail ring counter, twisted-ring counter, walking-ring counter, or Moebius counter) is a modified ring counter, where the output from the last stage is inverted and fed back as input to the first stage. A pattern of bits equal in length to twice the length of the shift register thus circulates indefinitely. These counters find specialist applications, including those similar to the decade counter, digital-to-analog conversion, etc. It can be established by D flip-flop and JK flip-flop.

Computer science counters

In computability theory, a **counter** is considered a type of memory. A counter stores a single natural number (initially zero) and can be arbitrarily many digits long. A counter is usually considered in conjunction with a finite-state machine (FSM), which can perform the following operations on the counter:

- Check whether the counter is zero
- Increment the counter by one
- Decrement the counter by one (if it's already zero, this leaves it unchanged).

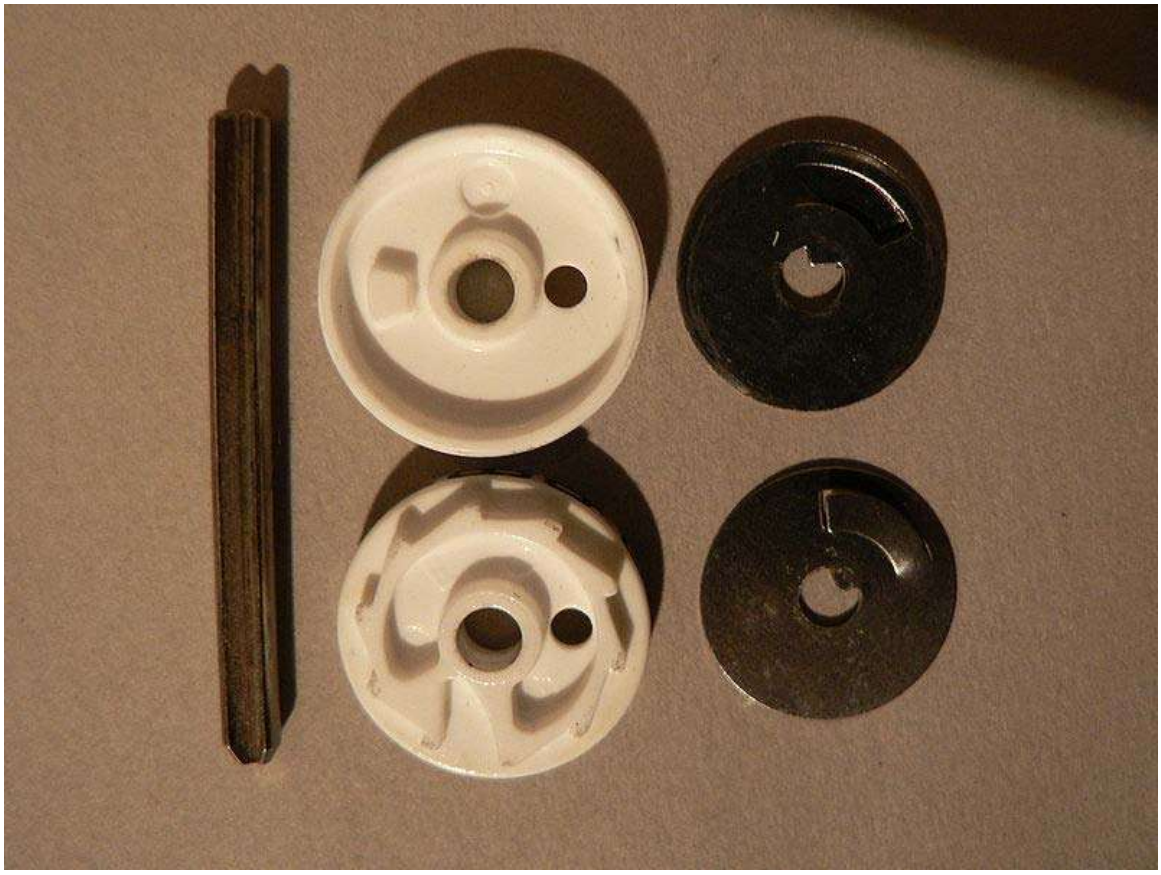
The following machines are listed in order of power, with each one being strictly more powerful than the one below it:

1. Deterministic or non-deterministic FSM plus two counters
2. Non-deterministic FSM plus one stack
3. Non-deterministic FSM plus one counter
4. Deterministic FSM plus one counter
5. Deterministic or non-deterministic FSM

For the first and last, it doesn't matter whether the FSM is a deterministic finite-state machine or a nondeterministic finite-state machine. They have equivalent power. The first two and the last one are levels of the Chomsky hierarchy.

The first machine, an FSM plus two counters, is equivalent in power to a Turing machine.

Mechanical counters



Mechanical counter wheels showing both sides. The bump on the wheel shown at the top engages the ratchet on the wheel below every turn.

Several mechanical counters

Long before electronics became common, mechanical devices were used to count events. These typically consist of a series of disks mounted on an axle, with the digits 0 through 9 marked on their edge. The right most disk moves one increment with each event. Each disk except the left-most has a protrusion that, after the completion of one revolution, moves the next disk to the left one increment. Such counters were originally used to control manufacturing processes, but were later used as odometers for bicycles and cars and in fuel dispensers. One of the largest manufacturers was the Veeder-Root company, and their name was often used for this type of counter.

Chapter- 7

Dual-modulus Prescaler and Propagation Delay

Dual-modulus prescaler

A **dual modulus prescaler** is an electronic circuit used in high-frequency synthesizer designs to overcome the problem of generating narrowly-spaced frequencies that are nevertheless too high to be passed directly through the feedback loop of the system. The modulus of a prescaler is its frequency divisor. A dual-modulus prescaler has two separate frequency divisors, usually M and M+1.

The problem

A frequency synthesizer produces an output frequency, f, which divided by the modulus is the reference frequency, f_r:

$$\frac{f_o}{N} = f_r \Rightarrow f = N f_r$$

The modulus, N, is is generally restricted to integral values, as the comparator will match when the waveform is in phase. Typically, the possible frequency multiples will be the channels for which the radio equipment is designed for, so f_r will usually be equal to the channel spacing. For example, on narrow-band radiotelephones, a channel spacing of 12.5 kHz is typical.

Suppose that the programmable divider, using N, is only able to operate at a maximum clock frequency of 10 MHz, but the output f is in the hundreds of MHz range; . Interposing a fixed prescaler, which can operate at this frequency range, with a value M of say, 40, drops the output frequency into the operating range of the programmable divider. However, a factor of 40 has been introduced into the equation, so the output frequency is now:

$f_o = 40Nf_r$

If f_r remains at 12.5 kHz, only every 40th channel can be obtained. Alternatively, if f_r is reduced by a factor of 40 to compensate, it becomes 312.5 Hz, which is much too low to give good filtering and lock performance characteristics. It also means that programming the divider becomes more complex, as the modulus needs to be verified so that only those that give true channels are used, not every 1/40th of a channel that is available.

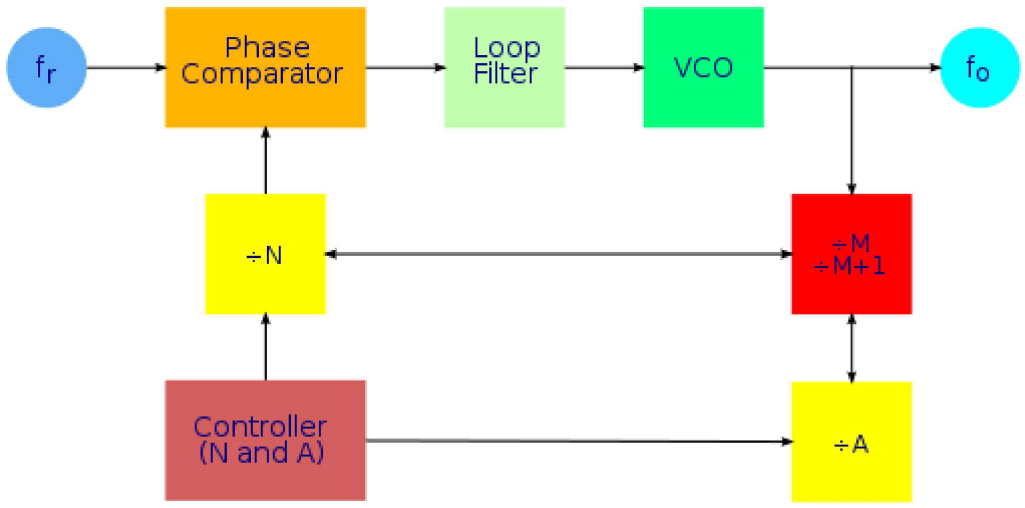
The solution

The solution is the dual modulus prescaler. The main divider is split into two parts, the main part N and an additional divider A which is strictly lesser than N. Both dividers are clocked from the output of the dual-modulus prescaler, but only the output of the N divider is fed back to the comparator. Initially, the prescaler is set to divide by M + 1. Both N and A count down until A reaches zero, at which point the prescaler is switched to a division ratio of M. At this point, the divider N has completed A counts. Counting continues until N reaches zero, which is an additional N - A counts. At this point the cycle repeats.

$$f_o = f_r\left[M(N-A)+(M+1)A\right]$$
$$\Rightarrow f_o = f_r\left(MN+A\right)$$

So while we still have a factor of M being multiplied by N, we can add an additional count, A, which effectively gives us a divider with a fractional part. Only the prescaler needs to be constructed from high-speed parts, and the reference frequency can remain equal to the desired output frequency spacing.

The diagram below shows the elements and arrangement of a frequency synthesizer with dual-modulus prescaler. (Compare with diagram on main synthesizer page).



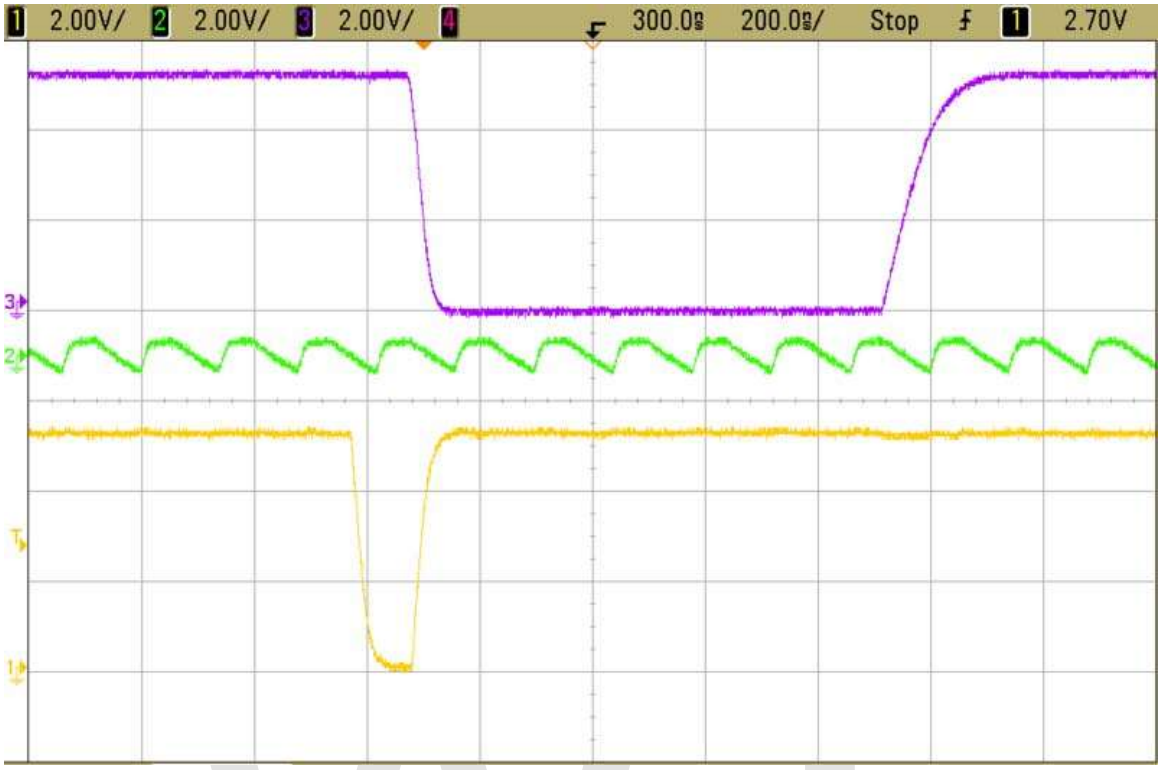
One can compute A and N from the formulae:

$$N = \left\lfloor \frac{V}{M} \right\rfloor$$
$$A = V - MN$$

where V is the combined division ratio V = MN+A. For this to work properly, A must be strictly less than M, as well as less than or equal to N. These restrictions on values of A imply that you can't get every division ratio V. If V falls below M(M - 1), some channels will be missing.

Example

Dual modulus prescaler waveform with a 10 microsecond scale.



Dual modulus prescaler waveform with a 20 nanosecond scale.

Today, most dual-modulus prescalers exist inside of PLL chips, making it impossible to probe actual signals during operation. The first dual-modulus prescalers were discrete ECL devices, separate from the PLL chips. Here is an example of a dual-modulus prescaler in use. This circuit happens to use a Motorola MC145158 with a Fujitsu MB-501 dual-modulus prescaler operating in the 128/129 mode. The PLL is locked at 917.94 MHz (f_o) with a channel spacing frequency of 30 kHz (f_r). The total integer count therefore is 30,598. Dividing this by 128 (M) yields a quotient of 239 with a remainder of 6, N and A respectively. The result of this frequency choice is that the prescaler spends most of its time counting at 128, and just a brief period at 129.

This is shown by the upper purple trace, the modulus control, A, counter output. These two screen captures differ only in the horizontal scale. The lower, yellow trace is the N counter output whose frequency corresponds to the channel spacing frequency of 30 kHz. The green trace is the output from the dual-modulus prescaler, which happens to correspond to 7.1714 MHz in the case that the prescaler is at 128 and 7.1158 when it is at 129. It is plainly obvious that the modulus control is low for precisely 6 cycles of the prescaler output. What is not obvious is the fact that the frequency changes by less than one percent between the two states of the modulus control. There will be cases where A = 0, resulting in the dual-modulus prescaler counting only by 128. This would happen at 906.24, 910.08, 913.92, 917.76, 921.60 MHz and so on.

Propagation delay

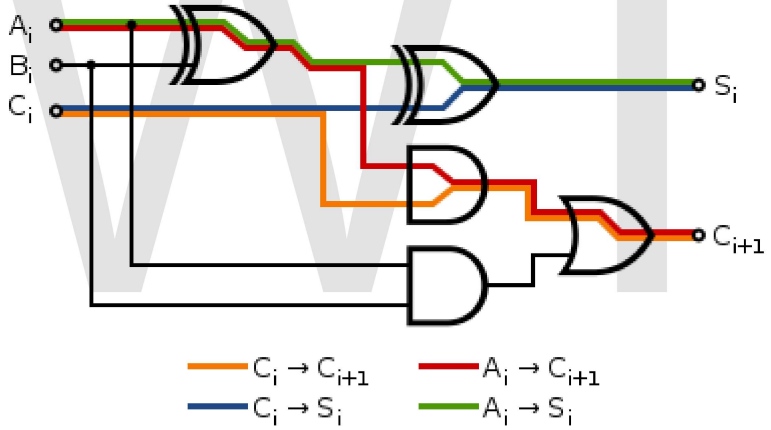
Propagation delay is a technical term that can have a different meaning depending on the context. It can relate to networking, electronics or physics. In general it is the length of time taken for the quantity of interest to reach its destination.

Networking

In computer networks, **propagation delay** is the amount of time it takes for the head of the signal to travel from the sender to the receiver over a medium. It can be computed as the ratio between the link length and the propagation speed over the specific medium.

Propagation delay = d/s where d is the distance and s is the wave propagation speed. In wireless communication, $s=c$, i.e. the speed of light. In copper wire, the speed s generally ranges from .59 to .77. This delay is the major obstacle in the development of high-speed computers and is called the interconnect bottleneck in IC systems.

Electronics



A full adder has an overall gate delay of 3 logic gates from the inputs A and B to the carry output C_{out} shown in red

In electronics, digital circuits and digital electronics, the **propagation delay**, or **gate delay**, is the length of time starting from when the input to a logic gate becomes stable and valid, to the time that the output of that logic gate is stable and valid. Often this refers to the time required for the output to reach from 10% to 90% of its final output level when the input changes. Reducing gate delays in digital circuits allows them to process data at a faster rate and improve overall performance.

The difference in *propagation delays* of logic elements is the major contributor to glitches in asynchronous circuits as a result of race conditions.

The principle of logical effort utilizes propagation delays to compare designs implementing the same logical statement.

Propagation delay increases with operating temperature, marginal supply voltage as well as an increased output load capacitance. The latter is the largest contributor to the increase of propagation delay. If the output of a logic gate is connected to a long trace or used to drive many other gates (high fanout) the propagation delay increases substantially.

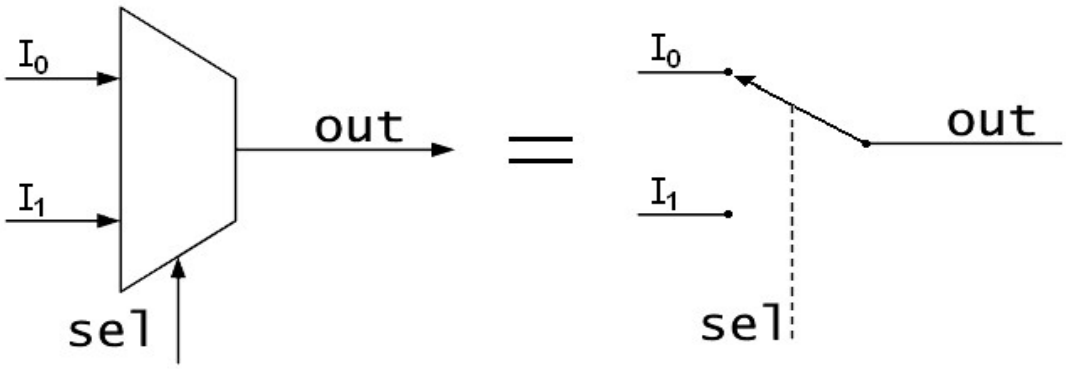
Wires have an approximate propagation delay of 1 ns for every 6 in of length. Logic gates can have propagation delays ranging from more than 10 ns down to the picosecond range, depending on the technology being used.

Physics

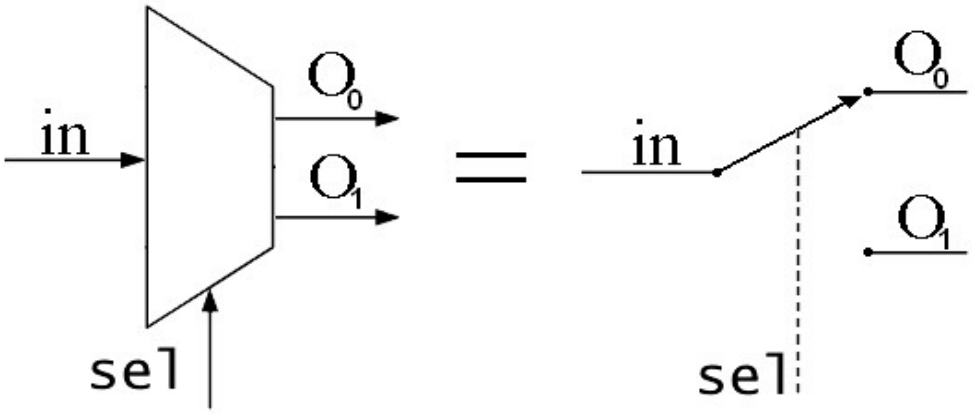
In physics, particularly in the electromagnetism field, the **propagation delay** is the length of time it takes for a signal to travel to its destination. For example, in the case of an electric signal, it is the time taken for the signal to travel through a wire.

Chapter- 8

Multiplexer



Schematic of a 2-to-1 Multiplexer. It can be equated to a controlled switch.



Schematic of a 1-to-2 Demultiplexer. Like a multiplexer, it can be equated to a controlled switch.

In electronics, a **multiplexer** or **mux** is a device that selects one of several analog or digital input signals and forwards the selected input into a single line. A multiplexer of 2^n inputs has n select lines, which are used to select which input line to send to the output.

An electronic multiplexer makes it possible for several signals to share one device or resource, for example one A/D converter or one communication line, instead of having one device per input signal.

On the other end, a demultiplexer (or **demux**) is a device taking a single input signal and selecting one of many data-output-lines, which is connected to the single input. A multiplexer is often used with a complementary demultiplexer on the receiving end.

An electronic multiplexer can be considered as a multiple-input, single-output switch, and a demultiplexer as a single-input, multiple-output switch. The schematic symbol for a multiplexer is an isosceles trapezoid with the longer parallel side containing the input pins and the short parallel side containing the output pin. The schematic on the right shows a 2-to-1 multiplexer on the left and an equivalent switch on the right. The *sel* wire connects the desired input to the output.

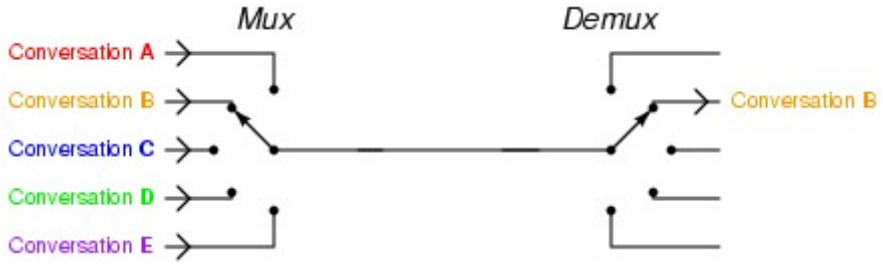
Telecommunications

In telecommunications, a multiplexer is a device that combines several input information signals into one output signal, which carries several communication channels, by means of some multiplex technique. A **demultiplexer** is in this context a device taking a single input signal that carries many channels and separates those over multiple output signals.

In telecommunications and signal processing, an analog time division multiplexer (TDM) may take several samples of separate analogue signals and combine them into one pulse amplitude modulated (PAM) wide-band analogue signal. Alternatively, a digital TDM multiplexer may combine a limited number of constant bit rate digital data streams into one data stream of a higher data rate, by forming data frames consisting of one timeslot per channel.

In telecommunications, computer networks and digital video, a statistical multiplexer may combine several variable bit rate data streams into one constant bandwidth signal, for example by means of packet mode communication. An inverse multiplexer may utilize several communication channels for transferring one signal.

Cost savings



The basic function of a multiplexer: combining multiple inputs into a single data stream. On the receiving side, a demultiplexer splits the single data stream into the original multiple signals.

One use for multiplexers is cost savings by connecting a multiplexer and a **demultiplexer** (or **demux**) together over a single channel (by connecting the multiplexer's single output to the demultiplexer's single input). The image to the right demonstrates this. In this case, the cost of implementing separate channels for each data source is more expensive than the cost and inconvenience of providing the multiplexing/demultiplexing functions. In a physical analogy, consider the merging behaviour of commuters crossing a narrow bridge; vehicles will take turns using the few available lanes. Upon reaching the end of the bridge they will separate into separate routes to their destinations.

At the receiving end of the data link a complementary *demultiplexer* is normally required to break single data stream back down into the original streams. In some cases, the far end system may have more functionality than a simple demultiplexer and so, while the demultiplexing still exists logically, it may never actually happen physically. This would be typical where a multiplexer serves a number of IP network users and then feeds directly into a router which immediately reads the content of the entire link into its routing processor and then does the demultiplexing in memory from where it will be converted directly into IP packets.

Often, a multiplexer and demultiplexer are combined together into a single piece of equipment, which is usually referred to simply as a "multiplexer". Both pieces of equipment are needed at both ends of a transmission link because most communications systems transmit in both directions.

A real world example is the creation of telemetry for transmission from the computer/instrumentation system of a satellite, space craft or other remote vehicle to a ground-based system.

In analog circuit design, a multiplexer is a special type of analog switch that connects one signal selected from several inputs to a single output.

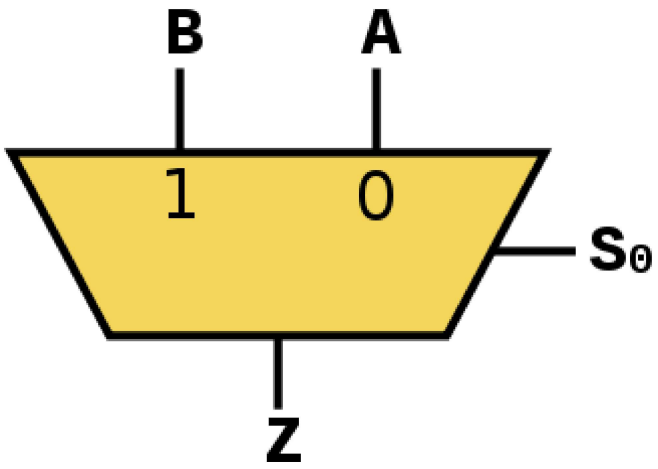
Digital multiplexers

In digital circuit design, the selector wires are of digital value. In the case of a 2-to-1 multiplexer, a logic value of 0 would connect I_0 to the output while a logic value of 1 would connect I_1 to the output. In larger multiplexers, the number of selector pins is equal to $\lceil \log_2(n) \rceil$ where n is the number of inputs.

For example, 9 to 16 inputs would require no fewer than 4 selector pins and 17 to 32 inputs would require no fewer than 5 selector pins. The binary value expressed on these selector pins determines the selected input pin.

A 2-to-1 multiplexer has a boolean equation where A and B are the two inputs, S is the selector input, and Z is the output:

$$Z = (A \cdot \overline{S}) + (B \cdot S)$$



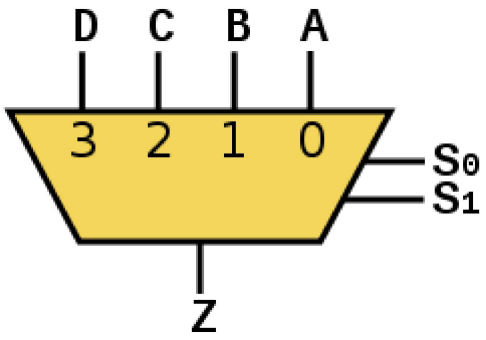
A 2-to-1 mux

Which can be expressed as a truth table:

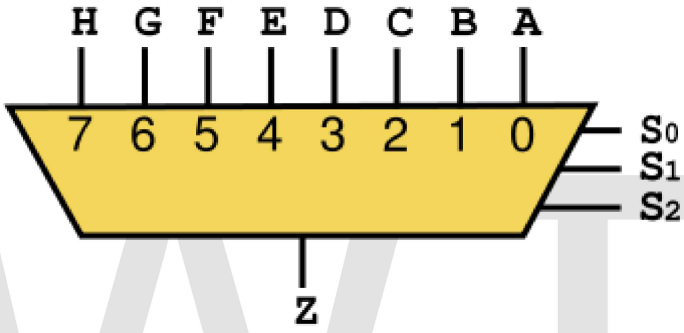
S	A	B	Z
0	1	1	1
	1	0	1
	0	1	0
	0	0	0
1	1	1	1
	1	0	0
	0	1	1
	0	0	0

This truth table shows that when $S=0$ then $Z=A$ but when $S=1$ then $Z=B$. A straightforward realization of this 2-to-1 multiplexer would need 2 AND gates, an OR gate, and a NOT gate.

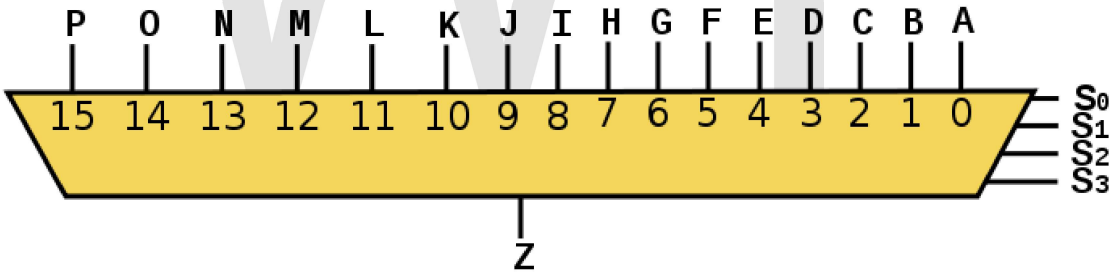
Larger multiplexers are also common and, as stated above, requires $\lceil \log_2(n) \rceil$ selector pins for *n* inputs. Other common sizes are 4-to-1, 8-to-1, and 16-to-1. Since digital logic uses binary values, powers of 2 are used (4, 8, 16) to maximally control a number of inputs for the given number of selector inputs.



4-to-1 mux



8-to-1 mux

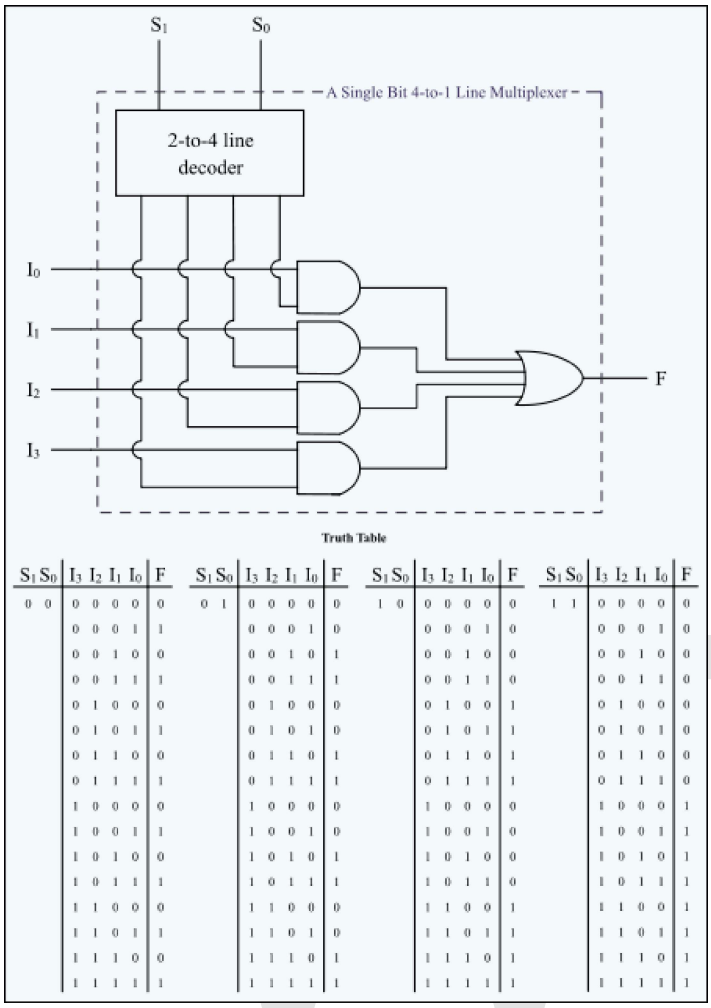


16-to-1 mux

The boolean equation for a 4-to-1 multiplexer is:

$$F = (A \cdot \overline{S_0} \cdot \overline{S_1}) + (B \cdot \overline{S_0} \cdot S_1) + (C \cdot S_0 \cdot \overline{S_1}) + (D \cdot S_0 \cdot S_1)$$

Two realizations for creating a 4-to-1 multiplexer are shown below:



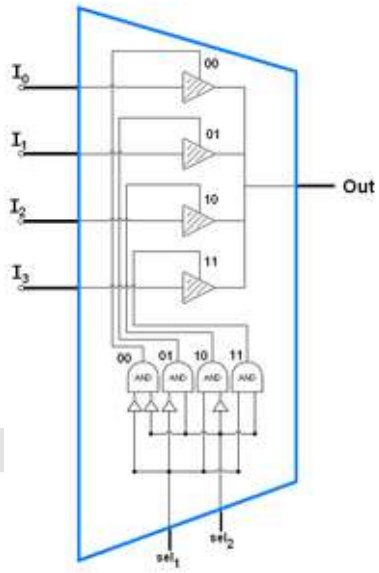
Truth Table

S_1 S_0	I_3	I_2	I_1	I_0	F
0 0	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	0
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	1
	0	1	1	0	0
	0	1	1	1	1
	1	0	0	0	0
	1	0	0	1	1
	1	0	1	0	0
	1	0	1	1	1
	1	1	0	0	0
	1	1	0	1	1
	1	1	1	0	0
	1	1	1	1	1

S_1 S_0	I_3	I_2	I_1	I_0	F
0 1	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	1
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	1
	0	1	1	1	1
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	1
	1	0	1	1	1
	1	1	0	0	0
	1	1	0	1	0
	1	1	1	0	1
	1	1	1	1	1

S_1 S_0	I_3	I_2	I_1	I_0	F
1 0	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	0
	0	1	0	0	1
	0	1	0	1	1
	0	1	1	0	1
	0	1	1	1	1
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	0
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	1

S_1 S_0	I_3	I_2	I_1	I_0	F
1 1	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	0
	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	0
	0	1	1	1	0
	1	0	0	0	1
	1	0	0	1	1
	1	0	1	0	1
	1	0	1	1	1
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	1



These are two realizations of a 4-to-1 multiplexer:

- one realized from a decoder, AND gates, and an OR gate
- one realized from 3-state buffers and AND gates (the AND gates are acting as the decoder)

Note that the subscripts on the I_n inputs indicate the decimal value of the binary control inputs at which that input is let through.

Chaining multiplexers

Larger multiplexers can be constructed by using smaller multiplexers by chaining them together. For example, an 8-to-1 multiplexer can be made with two 4-to-1 and one 2-to-1 multiplexers. The two 4-to-1 multiplexer outputs are fed into the 2-to-1 with the selector pins on the 4-to-1's put in parallel giving a total number of selector inputs to 3, which is equivalent to an 8-to-1.

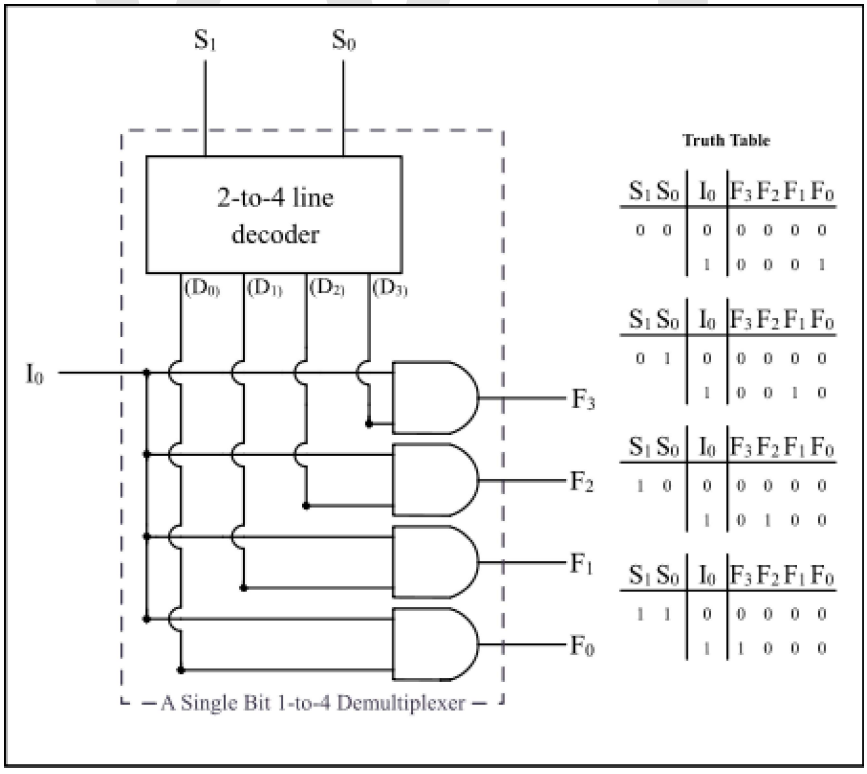
List of ICs which provide multiplexing

The 7400 series has several ICs that contain multiplexer(s):

S.No.	IC No.	Function	Output State
1	74157	Quad 2:1 mux.	Output same as input given
2	74158	Quad 2:1 mux.	Output is inverted input
3	74153	Dual 4:1 mux.	Output same as input
4	74352	Dual 4:1 mux.	Output is inverted input
5	74151A	8:1 mux.	Both outputs available (i.e., complementary outputs)
6	74151	8:1 mux.	Output is inverted input
7	74150	16:1 mux.	Output is inverted input

Digital demultiplexers

Demultiplexers take one data input and a number of selection inputs, and they have several outputs. They forward the data input to one of the outputs depending on the values of the selection inputs. Demultiplexers are sometimes convenient for designing general purpose logic, because if the demultiplexer's input is always true, the demultiplexer acts as a decoder. This means that any function of the selection bits can be constructed by logically OR-ing the correct set of outputs.



Example: A Single Bit 1-to-4 Line Demultiplexer

List of ICs which provide demultiplexing

The 7400 series has several ICs that contain demultiplexer(s):

S.No.	IC No.	Function	Output State
1	74139	Dual 1:4 demux.	Output is inverted input
3	74156	Dual 1:4 demux.	Output is open collector
4	74138	3:8 demux.	Output is inverted input
5	74154	1:16 demux.	Output is inverted input
6	74159	1:16 demux.	Output is open collector and same as input

Multiplexers as PLDs

Multiplexers can also be used as programmable logic devices. By specifying the logic arrangement in the input signals, a custom logic circuit can be created. The selector inputs then act as the logic inputs. This is especially useful in situations when cost is a factor and for modularity.

Chapter- 9

Multivibrator

A **multivibrator** is an electronic circuit used to implement a variety of simple two-state systems such as oscillators, timers and flip-flops. It is characterized by two amplifying devices (transistors, electron tubes or other devices) cross-coupled by resistors or capacitors. The name "multivibrator" was initially applied to the free-running oscillator version of the circuit because its output waveform was rich in harmonics. There are three types of multivibrator circuit depending on the circuit operation:

- **astable**, in which the circuit is not stable in either state —it continually switches from one state to the other. It does not require an input such as a clock pulse.
- **monostable**, in which one of the states is stable, but the other state is unstable (transient). A trigger causes the circuit to enter the unstable state. After entering the unstable state, the circuit will return to the stable state after a set time. Such a circuit is useful for creating a timing period of fixed duration in response to some external event. This circuit is also known as a **one shot**.
- **bistable**, in which the circuit is stable in either state. The circuit can be flipped from one state to the other by an external event or trigger.

Multivibrators find applications in a variety of systems where square waves or timed intervals are required. For example, before the advent of low-cost integrated circuits, chains of multivibrators found use as frequency dividers. A free-running multivibrator with a frequency of one-half to one-tenth of the reference frequency would accurately lock to the reference frequency. This technique was used in early electronic organs, to keep notes of different octaves accurately in tune. Other applications included early television systems, where the various line and frame frequencies were kept synchronized by pulses included in the video signal.

History

The classic multivibrator circuit (also called a *plate-coupled multivibrator*) is first described by H. Abraham and E. Bloch in *Publication 27* of the French *Ministère de la Guerre*, and in *Annales de Physique 12, 252 (1919)*. It is a predecessor of Eccles-Jordan trigger derived from this circuit a year later on.

Astable multivibrator

An astable multivibrator is a regenerative circuit consisting of two amplifying stages connected in a positive feedback loop by two capacitive-resistive coupling networks. The amplifying elements may be junction or field-effect transistors, vacuum tubes, operational amplifiers, or other types of amplifier. The example diagram shows bipolar junction transistors.

The circuit is usually drawn in a symmetric form as a cross-coupled pair. Two output terminals can be defined at the active devices, which will have complementary states; one will have high voltage while the other has low voltage, (except during the brief transitions from one state to the other).

Operation

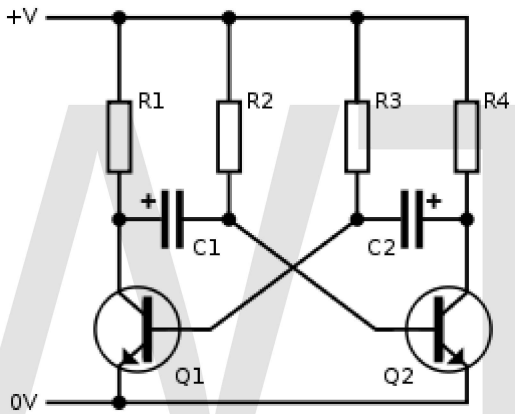


Figure 1: Basic BJT astable multivibrator

The circuit has two stable states that change alternatively with maximum transition rate because of the "accelerating" positive feedback. It is implemented by the coupling capacitors that instantly transfer voltage changes because the voltage across a capacitor cannot suddenly change. In each state, one transistor is switched on and the other is switched off. Accordingly, one fully charged capacitor discharges (reverse charges) slowly thus converting the time into an exponentially changing voltage. At the same time, the other empty capacitor quickly charges thus restoring its charge (the first capacitor acts as a time-setting capacitor and the second prepares to play this role in the next state). The circuit operation is based on the fact that the forward-biased base-emitter junction of the switched-on bipolar transistor can provide a path for the capacitor restoration.

State 1 (Q1 is switched on, Q2 is switched off):
In the beginning, the capacitor C1 is fully charged (in the previous State 2) to the power supply voltage *V* with the polarity shown in Figure 1. Q1 is *on* and connects the left-hand positive plate of C1 to ground. As its right-hand negative plate is connected to Q2 base, a maximum negative voltage (-*V*) is applied to Q2 base that keeps Q2 firmly *off*. C1 begins discharging (reverse charging) via the high-resistive base resistor R2, so that the voltage

of its right-hand plate (and at the base of Q2) is rising from below ground (-*V*) toward +*V*. As Q2 base-emitter junction is backward-biased, it does not impact on the exponential process (R2-C1 integrating network is unloaded). Simultaneously, C2 that is fully discharged and even slightly charged to 0.6 V (in the previous State 2) quickly charges via the low-resistive collector resistor R4 and Q1 forward-biased base-emitter junction (because R4 is less than R2, C2 charges faster than C1). Thus C2 restores its charge and prepares for the next State 2 when it will act as a time-setting capacitor. Q1 is firmly saturated in the beginning by the "forcing" C2 charging current added to R3 current; in the end, only R3 provides the needed input base current. The resistanse R3 is chosen small enough to keep Q1 (not deeply) saturated after C2 is fully charged.

When the voltage of C1 right-hand plate (Q2 base voltage) becomes positive and reaches 0.6 V, Q2 base-emitter junction begins diverting a part of R3 charging current. Q2 begins conducting and this starts the avalanche-like positive feedback process as follows. Q2 collector voltage begins falling; this change transfers through the fully charged C2 to Q1 base and Q1 begins cutting off. Its collector voltage begins rising; this change transfers back through the almost empty C1 to Q2 base and makes Q2 conduct more thus sustaining the initial input impact on Q2 base. Thus the initial input change circulates along the feedback loop and grows in an avalanche-like manner until finally Q1 switches off and Q2 switches on. The forward-biased Q2 base-emitter junction fixes the voltage of C1 right-hand plate at 0.6 V and does not allow it to continue rising toward +*V*.

State 2 (Q1 is switched off, Q2 is switched on):
Now, the capacitor C2 is fully charged (in the previous State 1) to the power supply voltage *V* with the polarity shown in Figure 1. Q2 is *on* and connects the right-hand positive plate of C2 to ground. As its left-hand negative plate is connected to Q1 base, a maximum negative voltage (-*V*) is applied to Q1 base that keeps Q1 firmly *off*. C2 begins discharging (reverse charging) via the high-resistive base resistor R3, so that the voltage of its left-hand plate (and at the base of Q1) is rising from below ground (-*V*) toward +*V*. Simultaneously, C1 that is fully discharged and even slightly charged to 0.6 V (in the previous State 1) quickly charges via the low-resistive collector resistor R1 and Q2 forward-biased base-emitter junction (because R1 is less than R3, C1 charges faster than C2). Thus C1 restores its charge and prepares for the next State 1 when it will act again as a time-setting capacitor...and so on... (the next explanations are a mirror copy of the second part of Step 1).

Multivibrator period (frequency)

Derivation

The duration of state 1 (low output) will be related to the time constant R_2C_1 as it depends on the charging of C1, and the duration of state 2 (high output) will be related to the time constant R_3C_2 as it depends on the charging of C2. Because they do not need to be the same, an asymmetric duty cycle is easily achieved. The extended content below contains a derivation of the multivibrator period (frequency).

Summary

The total period of oscillation is given by:

$T = t_1 + t_2 = \ln(2) R_2 C_1 + \ln(2) R_3 C_2$

$$f = \frac{1}{T} = \frac{1}{\ln(2) \cdot (R_2 C_1 + R_3 C_2)} \approx \frac{1}{0.693 \cdot (R_2 C_1 + R_3 C_2)}$$

where...

- f is frequency in hertz.
- R_2 and R_3 are resistor values in ohms.
- C_1 and C_2 are capacitor values in farads.
- T is the period (In this case, the sum of two period durations).

For the special case where

- $t_1 = t_2$ (50% duty cycle)
- $R_2 = R_3$
- $C_1 = C_2$

$$f = \frac{1}{T} = \frac{1}{\ln(2) \cdot 2RC} \approx \frac{0.721}{RC}$$

The duration of each state also depends on the initial state of charge of the capacitor in question, and this in turn will depend on the amount of discharge during the previous state, which will also depend on the resistors used during discharge (R1 and R4) and also on the duration of the previous state, *etc*. The result is that when first powered up, the period will be quite long as the capacitors are initially fully discharged, but the period will quickly shorten and stabilise.

The period will also depend on any current drawn from the output and on the supply voltage.

Output pulse shape

The output voltage has a shape that approximates a square waveform. It is considered below for the transistor Q1.

During State 1, Q2 base-emitter junction is backward-biased and the capacitor C1 is "unhooked" from ground. The output voltage of the switched-on transistor Q1 changes rapidly from high to low since this low-resistive output is loaded by a high impedance load (the series connected capacitor C1 and the high-resistive base resistor R2).

During State 2, Q2 base-emitter junction is forward-biased and the capacitor C1 is "hooked" to ground. The output voltage of the switched-off transistor Q1 changes exponentially from low to high since this relatively high resistive output is loaded by a low impedance load (the capacitance C1). This is the output voltage of R_1C_1 integrating circuit.

To approach the needed square waveform, the collector resistors have to be low resistance. The base resistors have to be low enough to make the transistors saturate in the end of the restoration ($R_B < \beta.R_C$).

Initial power-up

When the circuit is first powered up, neither transistor will be switched on. However, this means that at this stage they will both have high base voltages and therefore a tendency to switch on, and inevitable slight asymmetries will mean that one of the transistors is first to switch on. This will quickly put the circuit into one of the above states, and oscillation will ensue. In practice, oscillation always occurs for practical values of *R* and *C*.

However, if the circuit is temporarily held with both bases high, for longer than it takes for both capacitors to charge fully, then the circuit will remain in this stable state, with both bases at 0.6 V, both collectors at 0 V, and both capacitors charged backwards to −0.6 V. This can occur at startup without external intervention, if *R* and *C* are both very small.

Frequency divider

An astable multivibrator can be synchronized to an external chain of pulses. A single pair of active devices can be used to divide a reference by a large ratio, however, the stability of the technique is poor owing to the variability of the power supply and the circuit elements; a division ratio of 10, for example, is easy to obtain but not dependable. Chains of bistable flip-flops provide more predictable division, at the cost of more active elements.

Protective components

While not fundamental to circuit operation, diodes connected in series with the base or emitter of the transistors are required to prevent the base-emitter junction being driven into reverse breakdown when the supply voltage is in excess of the V_{eb} breakdown voltage, typically around 5-10 volts for general purpose silicon transistors. In the monostable configuration, only one of the transistors requires protection.

Monostable multivibrator circuit

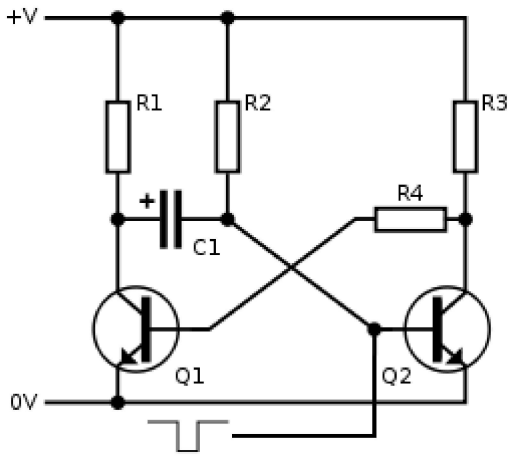


Figure 2: Basic BJT monostable multivibrator.

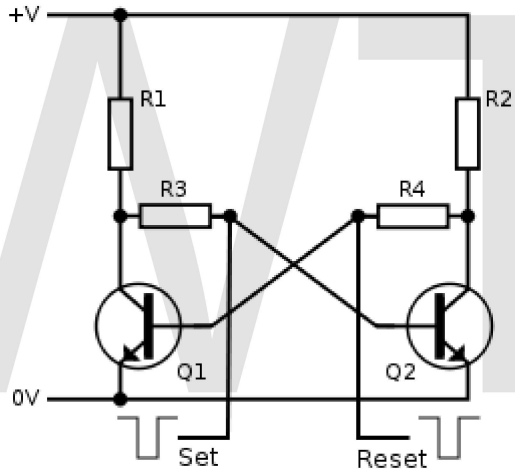


Figure 3: Basic BJT bistable multivibrator (suggested values: *R1, R2* = 1 kΩ *R3, R4* = 10 kΩ).

In the monostable multivibrator, the one resistive-capacitive network (C_2-R_3 in figure 1) is replaced by a resistive network (just a resistor). The circuit can be thought as a 1/2 astable multivibrator. Q2 collector voltage is the output of the circuit (in contrast to the astable circuit, it has a perfect square waveform since the output is not loaded by the capacitor).

When triggered by an input pulse, a monostable multivibrator will switch to its unstable position for a period of time, and then return to its stable state. The time period monostable multivibrator remains in unstable state is given by $t = \ln(2)R_2C_1$. If repeated application of the input pulse maintains the circuit in the unstable state, it is called a *retriggerable* monostable. If further trigger pulses do not affect the period, the circuit is a *non-retriggerable* multivibrator.

For the circuit in Figure 2, in the stable state Q1 is turned off and Q2 is turned on. It is triggered by zero or negative input signal applied to Q2 base (with the same success it can be triggered by applying a positive input signal through a resistor to Q1 base). As a result, the circuit goes in State 1 described above. After elapsing the time, it returns to its stable initial state.

Bistable multivibrator circuit

In the bistable multivibrator, both the resistive-capacitive network are replaced by resistive networks (just resistors or direct coupling).

This latch circuit is similar to an astable multivibrator, except that there is no charge or discharge time, due to the absence of capacitors. Hence, when the circuit is switched on, if Q1 is on, its collector is at 0 V. As a result, Q2 gets switched off. This results in more than half $+V$ volts being applied to R4 causing current into the base of Q1, thus keeping it on. Thus, the circuit remains stable in a single state continuously. Similarly, Q2 remains on continuously, if it happens to get switched on first.

Switching of state can be done via Set and Reset terminals connected to the bases. For example, if Q2 is on and Set is grounded momentarily, this switches Q2 off, and makes Q1 on. Thus, Set is used to "set" Q1 on, and Reset is used to "reset" it to off state.

Chapter- 10

Application-specific Integrated Circuit

An **application-specific integrated circuit** (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed solely to run a cell phone is an ASIC. Application-specific standard products (ASSPs) are intermediate between ASICs and industry standard integrated circuits like the 7400 or the 4000 series.

As feature sizes have shrunk and design tools improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5,000 gates to over 100 million. Modern ASICs often include entire 32-bit processors, memory blocks including ROM, RAM, EEPROM, Flash and other large building blocks. Such an ASIC is often termed a SoC (system-on-a-chip). Designers of digital ASICs use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

Field-programmable gate arrays (FPGA) are the modern-day technology for building a breadboard or prototype from standard parts; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications. For smaller designs and/or lower production volumes, FPGAs may be more cost effective than an ASIC design even in production. The non-recurring engineering cost of an ASIC can run into the millions of dollars.

History

The initial ASICs used gate array technology. Ferranti produced perhaps the first gate-array, the ULA (Uncommitted Logic Array), around 1980. An early successful commercial application was the ULA circuitry found in the 8-bit ZX81 and ZX Spectrum low-end personal computers, introduced in 1981 and 1982. These were used by Sinclair Research (UK) essentially as a low-cost I/O solution aimed at handling the computer's graphics. Some versions of ZX81/Timex Sinclair 1000 used just four chips (ULA, 2Kx8 RAM, 8Kx8 ROM, Z80A CPU) to implement an entire mass-market personal computer with built-in BASIC interpreter.

Customization occurred by varying the metal interconnect mask. ULAs had complexities of up to a few thousand gates. Later versions became more generalized, with different base dies customised by both metal and polysilicon layers. Some base dies include RAM elements.

Standard cell design

In the mid 1980s, a designer would choose an ASIC manufacturer and implement their design using the design tools available from the manufacturer. While third-party design tools were available, there was not an effective link from the third-party design tools to the layout and actual semiconductor process performance characteristics of the various ASIC manufacturers. Most designers ended up using factory-specific tools to complete the implementation of their designs. A solution to this problem, which also yielded a much higher density device, was the implementation of Standard Cells. Every ASIC manufacturer could create functional blocks with known electrical characteristics, such as propagation delay, capacitance and inductance, that could also be represented in third-party tools. Standard Cell design is the utilization of these functional blocks to achieve very high gate density and good electrical performance. Standard cell design fits between Gate Array and Full Custom design in terms of both its NRE (Non-Recurring Engineering) and recurring component cost.

By the late 1990s, logic synthesis tools became available. Such tools could compile HDL descriptions into a gate-level netlist. Standard-cell Integrated Circuits (ICs) are designed in the following conceptual stages, although these stages overlap significantly in practice.

1. A team of design engineers starts with a non-formal understanding of the required functions for a new ASIC, usually derived from Requirements analysis.
2. The design team constructs a description of an ASIC to achieve these goals using an HDL. This process is analogous to writing a computer program in a high-level language. This is usually called the RTL (Register transfer level) design.
3. Suitability for purpose is verified by functional verification. This may include such techniques as logic simulation, formal verification, emulation, or creating an equivalent pure software model. Each technique has advantages and disadvantages, and often several methods are used.
4. Logic synthesis transforms the RTL design into a large collection of lower-level constructs called standard cells. These constructs are taken from a standard-cell library consisting of pre-characterized collections of gates (such as 2 input nor, 2 input nand, inverters, etc.). The standard cells are typically specific to the planned manufacturer of the ASIC. The resulting collection of standard cells, plus the needed electrical connections between them, is called a gate-level netlist.
5. The gate-level netlist is next processed by a placement tool which places the standard cells onto a region representing the final ASIC. It attempts to find a placement of the standard cells, subject to a variety of specified constraints.
6. The routing tool takes the physical placement of the standard cells and uses the netlist to create the electrical connections between them. Since the search space is large, this process will produce a “sufficient” rather than “globally-optimal”

solution. The output is a file which can be used to create a set of photomasks enabling a semiconductor fabrication facility (commonly called a 'fab') to produce physical ICs.

7. Given the final layout, circuit extraction computes the parasitic resistances and capacitances. In the case of a digital circuit, this will then be further mapped into delay information, from which the circuit performance can be estimated, usually by static timing analysis. This, and other final tests such as design rule checking and power analysis (collectively called signoff) are intended to ensure that the device will function correctly over all extremes of the process, voltage and temperature. When this testing is complete the photomask information is released for chip fabrication.

These steps, implemented with a level of skill common in the industry, almost always produce a final device that correctly implements the original design, unless flaws are later introduced by the physical fabrication process.

The design steps (or flow) are also common to standard product design. The significant difference is that Standard Cell design uses the manufacturer's cell libraries that have been used in potentially hundreds of other design implementations and therefore are of much lower risk than full custom design. Standard Cells produce a design density that is cost effective, and they can also integrate IP cores and SRAM (Static Random Access Memory) effectively, unlike Gate Arrays.

Gate array design

Gate array design is a manufacturing method in which the diffused layers, i.e. transistors and other active devices, are predefined and wafers containing such devices are held in stock prior to metallization — in other words, unconnected. The physical design process then defines the interconnections of the final device. For most ASIC manufacturers, this consists of from two to as many as nine metal layers, each metal layer running perpendicular to the one below it. Non-recurring engineering costs are much lower, as photolithographic masks are required only for the metal layers, and production cycles are much shorter, as metallization is a comparatively quick process.

Gate array ASICs are always a compromise as mapping a given design onto what a manufacturer held as a stock wafer never gives 100% utilization. Often difficulties in routing the interconnect require migration onto a larger array device with consequent increase in the piece part price. These difficulties are often a result of the layout software used to develop the interconnect.

Pure, logic-only gate array design is rarely implemented by circuit designers today, having been replaced almost entirely by field-programmable devices, such as field-programmable gate arrays (FPGAs), which can be programmed by the user and thus offer minimal tooling charges (non-recurring engineering (NRE)), only marginally increased piece part cost, and comparable performance. Today, gate arrays are evolving into structured ASICs that consist of a large IP core like a CPU, DSP unit, peripherals,

standard interfaces, integrated memories SRAM, and a block of reconfigurable, uncommited logic. This shift is largely because ASIC devices are capable of integrating such large blocks of system functionality and "system-on-a-chip" requires far more than just logic blocks.

In their frequent usages in the field, the terms "gate array" and "semi-custom" are synonymous. Process engineers more commonly use the term "semi-custom", while "gate-array" is more commonly used by logic (or gate-level) designers.

Full-custom design

By contrast, full-custom ASIC design defines all the photolithographic layers of the device. Full-custom design is used for both ASIC design and for standard product design.

The benefits of full-custom design usually include reduced area (and therefore recurring component cost), performance improvements, and also the ability to integrate analog components and other pre-designed — and thus fully verified — components, such as microprocessor cores that form a system-on-chip.

The disadvantages of full-custom design can include increased manufacturing and design time, increased non-recurring engineering costs, more complexity in the computer-aided design (CAD) system, and a much higher skill requirement on the part of the design team.

For digital-only designs, however, "standard-cell" cell libraries, together with modern CAD systems, can offer considerable performance/cost benefits with low risk. Automated layout tools are quick and easy to use and also offer the possibility to "hand-tweak" or manually optimize any performance-limiting aspect of the design.

Structured design

Structured ASIC design (also referred to as "platform ASIC design"), is a relatively new term in the industry, resulting in some variation in its definition. However, the basic premise of a structured ASIC is that both manufacturing cycle time and design cycle time are reduced compared to cell-based ASIC, by virtue of there being pre-defined metal layers (thus reducing manufacturing time) and pre-characterization of what is on the silicon (thus reducing design cycle time). One definition states that

> In a "structured ASIC" design, the logic mask-layers of a device are predefined by the ASIC vendor (or in some cases by a third party). Design differentiation and customization is achieved by creating custom metal layers that create custom connections between predefined lower-layer logic elements. "Structured ASIC" technology is seen as bridging the gap between field-programmable gate arrays and "standard-cell" ASIC designs. Because only a small number of chip layers must be custom-produced, "structured ASIC" designs have much smaller non-

recurring expenditures (NRE) than "standard-cell" or "full-custom" chips, which require that a full mask set be produced for every design.

This is effectively the same definition as a gate array. What makes a structured ASIC different is that in a gate array, the predefined metal layers serve to make manufacturing turnaround faster. In a structured ASIC, the use of predefined metallization is primarily to reduce cost of the mask sets as well as making the design cycle time significantly shorter. For example, in a cell-based or gate-array design the user must often design power, clock, and test structures themselves; these are predefined in most structured ASICs and therefore can save time and expense for the designer compared to gate-array. Likewise, the design tools used for structured ASIC can be substantially lower cost and easier (faster) to use than cell-based tools, because they do not have to perform all the functions that cell-based tools do. In some cases, the structured ASIC vendor requires that customized tools for their device (e.g., custom physical synthesis) be used, also allowing for the design to be brought into manufacturing more quickly.

One other important aspect about structured ASIC is that it allows intellectual property (IP) that is common to certain applications or industry segments to be "built in", rather than "designed in". By building the IP directly into the architecture the designer can again save both time and money compared to designing IP into a cell-based ASIC.

Cell libraries, IP-based design, hard and soft macros

Cell libraries of logical primitives are usually provided by the device manufacturer as part of the service. Although they will incur no additional cost, their release will be covered by the terms of a non-disclosure agreement (NDA) and they will be regarded as intellectual property by the manufacturer. Usually their physical design will be pre-defined so they could be termed "hard macros".

What most engineers understand as "intellectual property" are IP cores, designs purchased from a third-party as sub-components of a larger ASIC. They may be provided as an HDL description (often termed a "soft macro"), or as a fully routed design that could be printed directly onto an ASIC's mask (often termed a hard macro). Many organizations now sell such pre-designed cores — CPUs, Ethernet, USB or telephone interfaces — and larger organizations may have an entire department or division to produce cores for the rest of the organization. Indeed, the wide range of functions now available is a significant factor in the phenomenal increase in electronics in the late 1990s and early 2000s; as a core takes a lot of time and investment to create, its re-use and further development cuts product cycle times dramatically and creates better products. Additionally, organizations such as OpenCores are collecting free IP cores paralleling the open source movement in software.

Soft macros are often process-independent, i.e., they can be fabricated on a wide range of manufacturing processes and different manufacturers. Hard macros are process-limited and usually further design effort must be invested to migrate (port) to a different process or manufacturer.

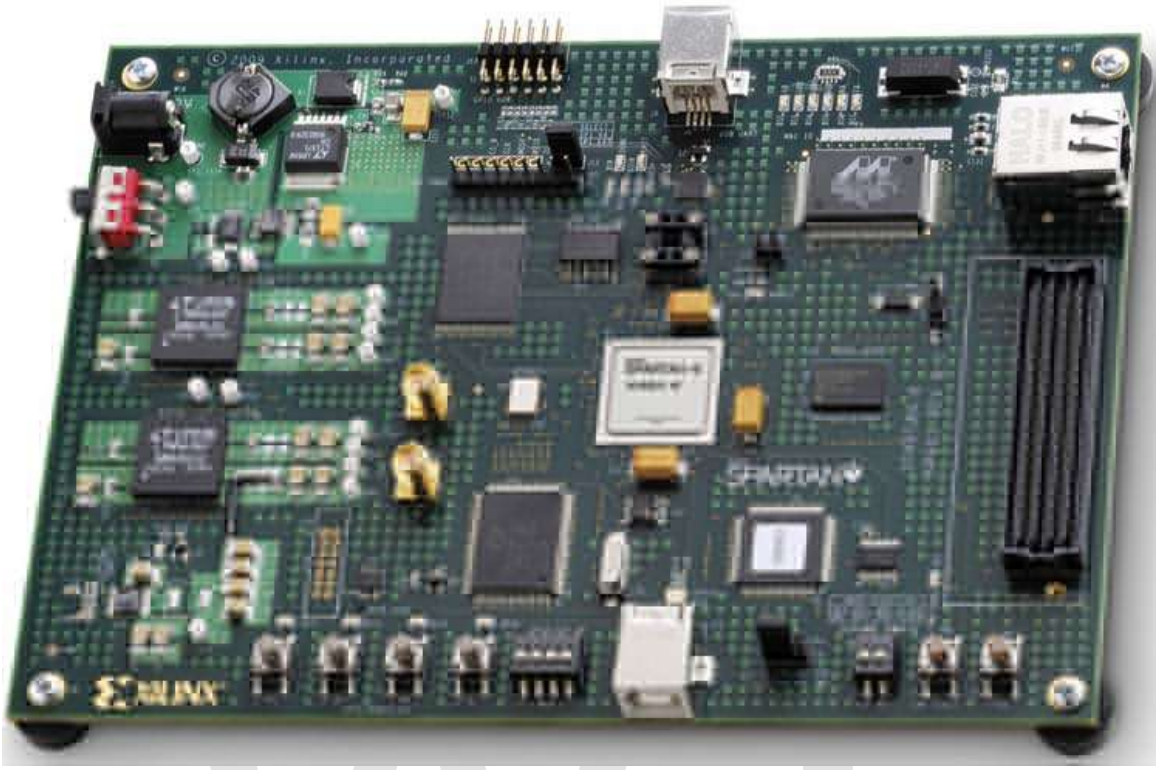
Multi-project wafers

Some manufacturers offer Multi-Project Wafers (MPW) as a method of obtaining low cost prototypes. Often called shuttles, these MPW, containing several designs, run at regular, scheduled intervals on a "cut and go" basis, usually with very little liability on the part of the manufacturer. The contract involves the assembly and packaging of a handful of devices. The service usually involves the supply of a physical design data base i.e. masking information or Pattern Generation (PG) tape. The manufacturer is often referred to as a "silicon foundry" due to the low involvement it has in the process. .

Chapter- 11

Field-programmable Gate Array

An Altera Stratix IV GX FPGA



An example of a Xilinx Spartan 6 FPGA programming/evaluation board

A **Field-programmable Gate Array** (**FPGA**) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications.

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations . Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

In addition to digital functions, some FPGAs have analog features. The most common analog feature is programmable slew rate and drive strength on each output pin, allowing the engineer to set slow rates on lightly loaded pins that would otherwise ring unacceptably, and to set stronger, faster rates on heavily loaded pins on high-speed channels that would otherwise run too slow. Another relatively common analog feature is

differential comparators on input pins designed to be connected to differential signaling channels. A few "mixed signal FPGAs" have integrated peripheral Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) with analog signal conditioning blocks allowing them to operate as a system-on-a-chip. Such devices blur the line between an FPGA, which carries digital ones and zeros on its internal programmable interconnect fabric, and field-programmable analog array (FPAA), which carries analog values on its internal programmable interconnect fabric.

History

The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both had the option of being programmed in batches in a factory or in the field (field programmable), however programmable logic was hard-wired between logic gates.

In the late 1980s the Naval Surface Warfare Department funded an experiment proposed by Steve Casselman to develop a computer that would implement 600,000 reprogrammable gates. Casselman was successful and a patent related to the system was issued in 1992.

Some of the industry’s foundational concepts and technologies for programmable logic arrays, gates, and logic blocks are founded in patents awarded to David W. Page and LuVerne R. Peterson in 1985.

Xilinx Co-Founders, Ross Freeman and Bernard Vonderschmitt, invented the first commercially viable field programmable gate array in 1985 – the XC2064. The XC2064 had programmable gates and programmable interconnects between gates, the beginnings of a new technology and market. The XC2064 boasted a mere 64 configurable logic blocks (CLBs), with two 3-input lookup tables (LUTs). More than 20 years later, Freeman was entered into the National Inventors Hall of Fame for his invention.

Xilinx continued unchallenged and quickly growing from 1985 to the mid-1990s, when competitors sprouted up, eroding significant market-share. By 1993, Actel was serving about 18 percent of the market.

The 1990s were an explosive period of time for FPGAs, both in sophistication and the volume of production. In the early 1990s, FPGAs were primarily used in telecommunications and networking. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications.

FPGAs got a glimpse of fame in 1997, when Adrian Thompson, a researcher working at the University of Sussex, merged genetic algorithm technology and FPGAs to create a sound recognition device. Thomson’s algorithm configured an array of 10 x 10 cells in a Xilinx FPGA chip to discriminate between two tones, utilising analogue features of the digital chip. The application of genetic algorithms to the configuration of devices like FPGA's is now referred to as Evolvable hardware

Modern developments

A recent trend has been to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". This work mirrors the architecture by Ron Perlof and Hana Potash of Burroughs Advanced Systems Group which combined a reconfigurable CPU architecture on a single chip called the SB24. That work was done in 1982. Examples of such hybrid technologies can be found in the Xilinx Virtex-II PRO and Virtex-4 devices, which include one or more PowerPC processors embedded within the FPGA's logic fabric. The Atmel FPSLIC is another such device, which uses an AVR processor in combination with Atmel's programmable logic architecture. The Actel SmartFusion devices incorporate an ARM architecture Cortex-M3 hard processor core (with up to 512kB of flash and 64kB of RAM) and analog peripherals such as a multi-channel ADC and DACs to their flash-based FPGA fabric.

An alternate approach to using hard-macro processors is to make use of soft processor cores that are implemented within the FPGA logic.

As previously mentioned, many modern FPGAs have the ability to be reprogrammed at "run time," and this is leading to the idea of reconfigurable computing or reconfigurable systems — CPUs that reconfigure themselves to suit the task at hand. The Mitrion Virtual Processor from Mitrionics is an example of a reconfigurable soft processor, implemented on FPGAs. However, it does not support dynamic reconfiguration at runtime, but instead adapts itself to a specific program.

Additionally, new, non-FPGA architectures are beginning to emerge. Software-configurable microprocessors such as the Stretch S5000 adopt a hybrid approach by providing an array of processor cores and FPGA-like programmable cores on the same chip.

Gates

- 1987: 9,000 gates, Xilinx
- 1992: 600,000, Naval Surface Warfare Department
- Early 2000s: Millions

Market size

- 1985: First commercial FPGA technology invented by Xilinx
- 1987: $14 million
- ~1993: >$385 million
- 2005: $1.9 billion
- 2010 estimates: $2.75 billion

FPGA design starts

- 10,000
- 2005: 80,000
- 2008: 90,000

FPGA comparisons

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. A study has shown that designs implemented on FPGAs need on average 18 times as much area, draw 7 times as much dynamic power, and are 3 times slower than the corresponding ASIC implementations.

An Altera Cyclone II FPGA, on an Altera teraSIC DE1 Prototyping board.

Advantages include the ability to re-program in the field to fix bugs, and may include a shorter time to market and lower non-recurring engineering costs. Vendors can also take a middle road by developing their hardware on ordinary FPGAs, but manufacture their final version so it can no longer be modified after the design has been committed.

Xilinx claims that several market and technology dynamics are changing the ASIC/FPGA paradigm:

- Integrated circuit costs are rising aggressively
- ASIC complexity has lengthened development time
- R&D resources and headcount are decreasing
- Revenue losses for slow time-to-market are increasing
- Financial constraints in a poor economy are driving low-cost technologies

These trends make FPGAs a better alternative than ASICs for a larger number of higher-volume applications than they have been historically used for, to which the company attributes the growing number of FPGA design starts.

Some FPGAs have the capability of partial re-configuration that lets one portion of the device be re-programmed while other portions continue running.

Versus complex programmable logic devices

The primary differences between CPLDs (Complex Programmable Logic Devices) and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also far more complex to design for.

Another notable difference between CPLDs and FPGAs is the presence in most FPGAs of higher-level embedded functions (such as adders and multipliers) and embedded memories, as well as to have logic blocks implement decoders or mathematical functions.

Security considerations

With respect to security, FPGAs have both advantages and disadvantages as compared to ASICs or secure microprocessors. FPGAs' flexibility makes malicious modifications during fabrication a lower risk. For many FPGAs, the loaded design is exposed while it is loaded (typically on every power-on). To address this issue, some FPGAs support bitstream encryption.

Applications

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. One such area is code breaking, in particular brute-force attack, of cryptographic algorithms.

FPGAs are increasingly used in conventional high performance computing applications where computational kernels such as FFT or Convolution are performed on the FPGA instead of a microprocessor.

The inherent parallelism of the logic resources on an FPGA allows for considerable computational throughput even at a low MHz clock rates. The flexibility of the FPGA allows for even higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has driven a new type of processing called reconfigurable computing, where time intensive tasks are offloaded from software to FPGAs.

The adoption of FPGAs in high performance computing is currently limited by the complexity of FPGA design compared to conventional software and the turn-around times of current design tools.

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

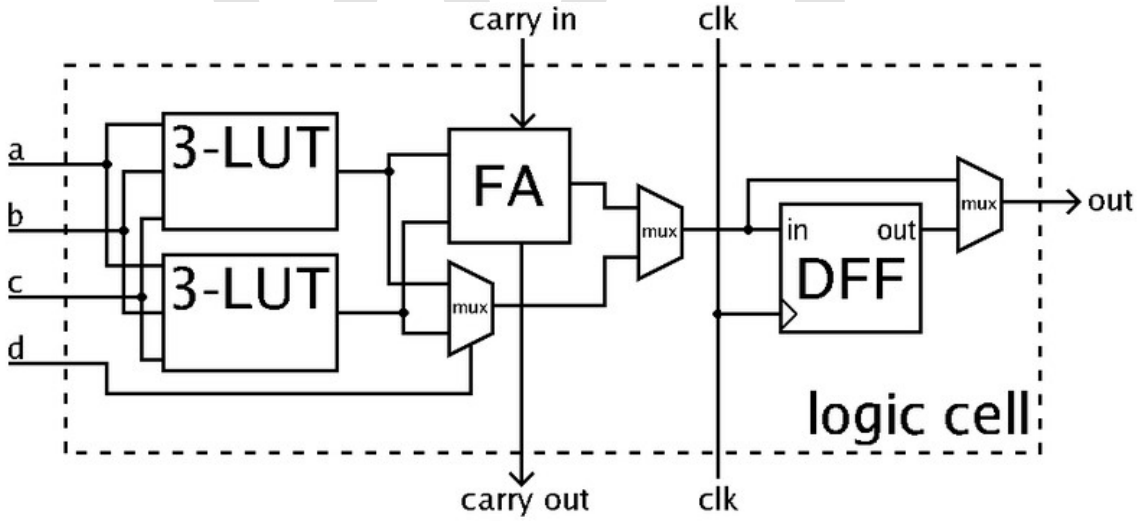
Architecture

The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. Generally, all the routing channels have the same width

(number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs/LABs and I/Os required is easily determined from the design, the number of routing tracks needed may vary considerably even among designs with the same amount of logic. For example, a crossbar switch requires much more routing than a systolic array with the same gate count. Since unused routing tracks increase the cost (and decrease the performance) of the part without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs that will fit in terms of LUTs and IOs can be routed. This is determined by estimates such as those derived from Rent's rule or by experiments with existing designs.

In general, a logic block (CLB or LAB) consists of a few logical cells (called ALM, LE, Slice etc). A typical cell consists of a 4-input Lookup table (LUT), a Full adder (FA) and a D-type flip-flop, as shown below. The LUTs are in this figure split into two 3-input LUTs. In *normal mode* those are combined into a 4-input LUT through the left mux. In *arithmetic* mode, their outputs are fed to the FA. The selection of mode is programmed into the middle mux. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space.



Simplified example illustration of a logic cell

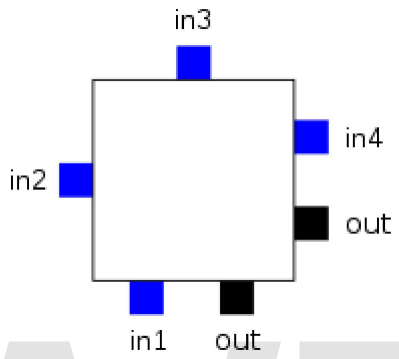
ALMs and Slices usually contains 2 or 4 structures similar to the example figure, with some shared signals.

CLBs/LABs typically contains a few ALMs/LEs/Slices.

In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.

Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed.

For this example architecture, the locations of the FPGA logic block pins are shown below.



Logic Block Pin Locations

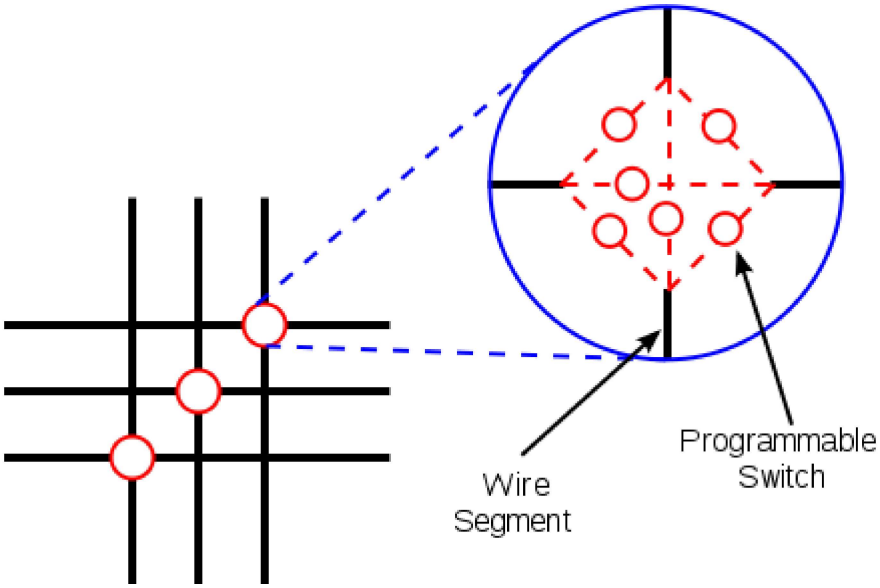
Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block.

Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

Whenever a vertical and a horizontal channel intersect, there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure below illustrates the connections in a switch box.



Switch box topology

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories.

FPGAs are also widely used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time-to-market.

FPGA design and programming

To define the behavior of the FPGA, the user provides a hardware description language (HDL) or a schematic design. The HDL form is more suited to work with large structures because it's possible to just specify them numerically rather than having to draw every piece by hand. However, schematic entry can allow for easier visualisation of a design.

Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA.

Going from schematic/HDL source files to actual configuration: The source files are fed to a software suite from the FPGA/CPLD vendor that through different steps will produce

a file. This file is then transferred to the FPGA/CPLD via a serial interface (JTAG) or to an external memory device like an EEPROM.

The most common HDLs are VHDL and Verilog, although in an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level through the introduction of alternative languages. National Instrument's LabVIEW graphical programming language (sometimes referred to as "G") has an FPGA add-in module available to target and program FPGA hardware. The LabVIEW approach drastically simplifies the FPGA programming process.

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly called *IP cores*, and are available from FPGA vendors and third-party IP suppliers (rarely free, and typically released under proprietary licenses). Other predefined circuits are available from developer communities such as OpenCores (typically released under free and open source licenses such as the GPL, BSD or similar license), and other sources.

In a typical design flow, an FPGA application developer will simulate the design at multiple stages throughout the design process. Initially the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. Then, after the synthesis engine has mapped the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.

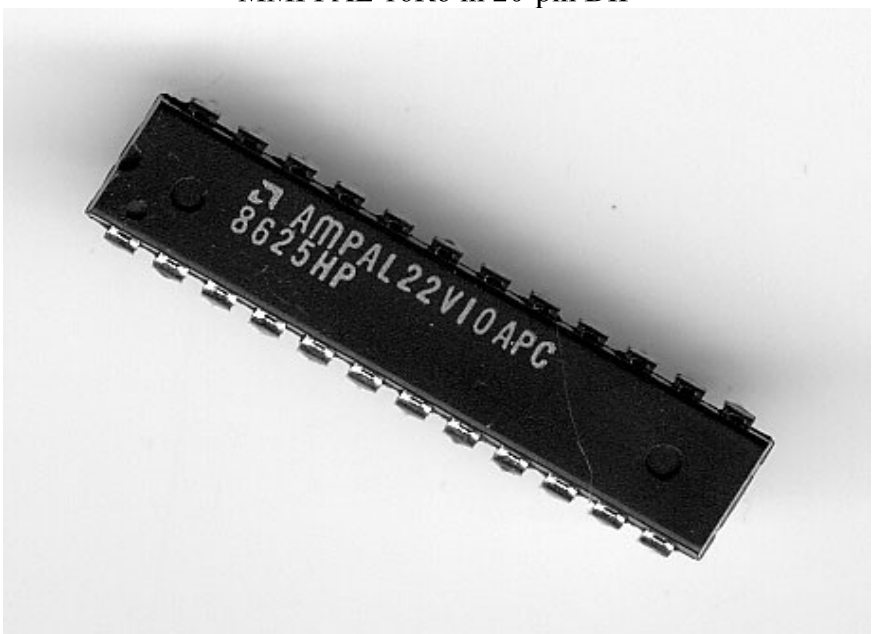
Basic process technology types

- SRAM - based on static memory technology. In-system programmable and re-programmable. Requires external boot devices. CMOS.
- Antifuse - One-time programmable. CMOS.
- PROM - Programmable Read-Only Memory technology. One-time programmable because of plastic packaging.
- EPROM - Erasable Programmable Read-Only Memory technology. One-time programmable but with window, can be erased with ultraviolet (UV) light. CMOS.
- EEPROM - Electrically Erasable Programmable Read-Only Memory technology. Can be erased, even in plastic packages. Some but not all EEPROM devices can be in-system programmed. CMOS.
- Flash - Flash-erase EPROM technology. Can be erased, even in plastic packages. Some but not all flash devices can be in-system programmed. Usually, a flash cell is smaller than an equivalent EEPROM cell and is therefore less expensive to manufacture. CMOS.
- Fuse - One-time programmable. Bipolar.

Chapter- 12

Programmable Array Logic

MMI PAL 16R6 in 20-pin DIP



AMD 22V10 in 24-pin DIP

The term **Programmable Array Logic** (PAL) is used to describe a family of programmable logic device semiconductors used to implement logic functions in digital circuits introduced by Monolithic Memories, Inc. (MMI) in March 1978. MMI obtained a registered trademark on the term PAL for use in "Programmable Semiconductor Logic Circuits". The trademark is currently held by Lattice Semiconductor.

PAL devices consisted of a small PROM (programmable read-only memory) core and additional output logic used to implement particular desired logic functions with few components.

Using specialized machines, PAL devices were "field-programmable". Each PAL device was "one-time programmable" (OTP), meaning that it could not be updated and reused after its initial programming. (MMI also offered a similar family called HAL, or "hard array logic", which were like PAL devices except that they were mask-programmed at the factory.)

KB01VG1

Early history

Before PALs were introduced, designers of digital logic circuits would use small-scale integration (SSI) components, such as those in the 7400 series TTL (transistor-transistor logic) family; the 7400 family included a variety of logic building blocks, such as gates (NOT, NAND, NOR, AND, OR), multiplexers (MUXes) and demultiplexers (DEMUXes), flip flops (D-type, JK, etc.) and others. One PAL device would typically replace dozens of such "discrete" logic packages, so the SSI business went into decline as the PAL business took off. PALs were used advantageously in many products, such as minicomputers, as documented in Tracy Kidder's best-selling book "The Soul of a New Machine."

PALs were not the first commercial programmable logic devices; Signetics had been selling its field programmable logic array (FPLA) since 1975. These devices were completely unfamiliar to most circuit designers and were perceived to be too difficult to use. The FPLA had a relatively slow maximum operating speed (due to having both programmable-AND and programmable-OR arrays), was expensive, and had a poor reputation for testability. Another factor limiting the acceptance of the FPLA was the large package, a 600-mil (0.6", or 15.24 mm) wide 28-pin dual in-line package (DIP).

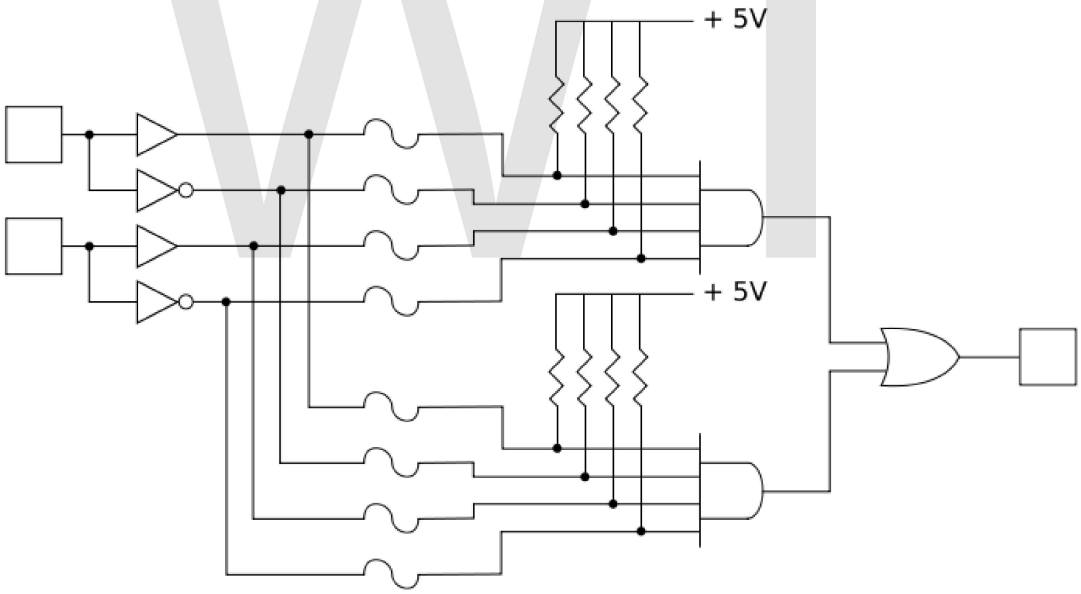
The project to create the PAL device was managed by John Birkner and the actual PAL circuit was designed by H. T. Chua. In a previous job, Birkner had developed a 16-bit processor using 80 standard logic devices. His experience with standard logic led him to believe that user programmable devices would be more attractive to users if the devices were designed to replace standard logic. This meant that the package sizes had to be more typical of the existing devices, and the speeds had to be improved. The PAL met these requirements and was a huge success and were "second sourced" by National Semiconductor, Texas Instruments, and Advanced Micro Devices.

Process technologies

Early PALs were 20-pin DIP components fabricated in silicon using bipolar transistor technology with one-time programmable (OTP) titanium-tungsten programming fuses. Later devices were manufactured by Lattice Semiconductor and Advanced Micro Devices using CMOS technology.

The original 20 and 24-pin PALs were described by MMI as medium-scale integration (MSI) devices.

PAL architecture



Simplified programmable logic device

The programmable elements (shown as a fuse) connect both the true and complemented inputs to the AND gates. These AND gates, also known as *product terms*, are ORed together to form a *sum-of-products* logic array.

The PAL architecture consists of two main components: a logic plane and output logic macrocells.

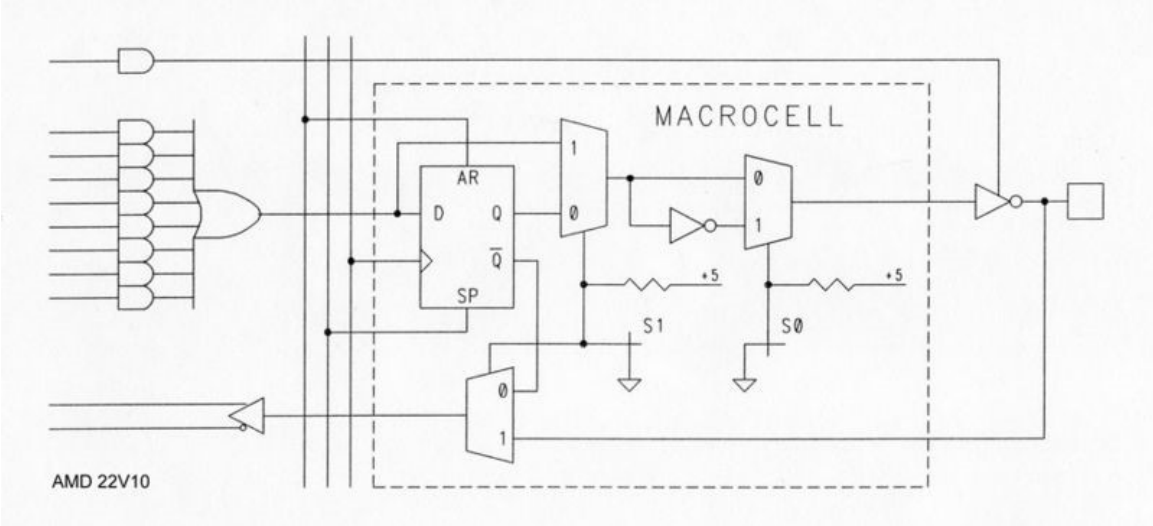
Programmable logic plane

The programmable logic plane is a programmable read-only memory (PROM) array that allows the signals present on the devices pins (or the logical complements of those signals) to be routed to an output logic macrocell.

PAL devices have arrays of transistor cells arranged in a "fixed-OR, programmable-AND" plane used to implement "sum-of-products" binary logic equations for each of the outputs in terms of the inputs and either synchronous or asynchronous feedback from the outputs.

Output logic

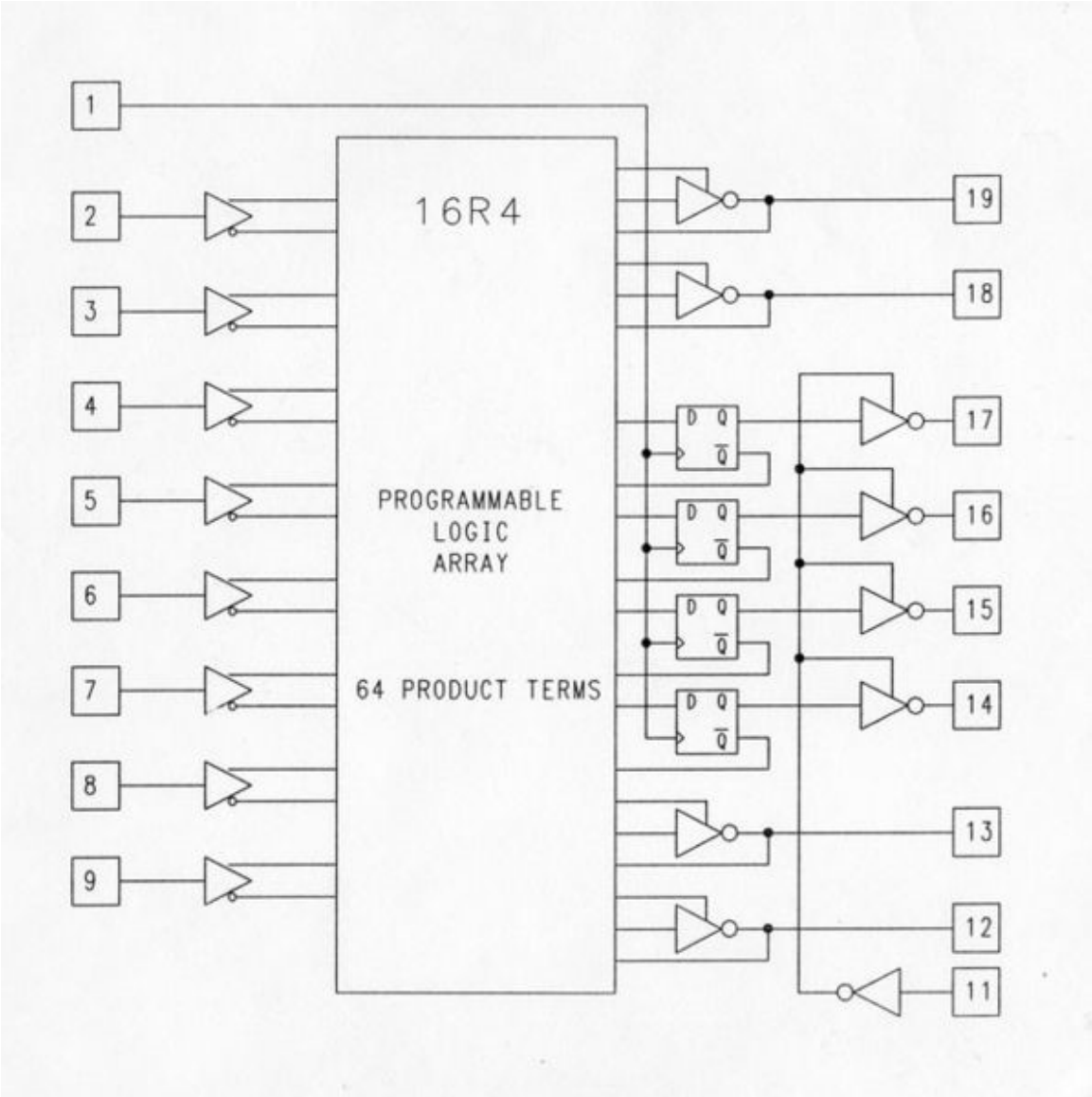
The early 20-pin PALs had 10 inputs and 8 outputs. The outputs were active low and could be registered or combinational. Members of the PAL family were available with various output structures called "output logic macrocells" or OLMCs. Prior to the introduction of the "V" (for "variable") series, the types of OLMCs available in each PAL were fixed at the time of manufacture. (The PAL16L8 had 8 combinational outputs and the PAL16R8 had 8 registered outputs. The PAL16R6 had 6 registered and 2 combinational while the PAL16R4 had 4 of each.) Each output could have up to 8 product terms (effectively AND gates), however the combinational outputs used one of the terms to control a bidirectional output buffer. There were other combinations that had fewer outputs with more product term per output and were available with active high outputs. The 16X8 family or registered devices had an XOR gate before the register. There were also similar 24-pin versions of these PALs.



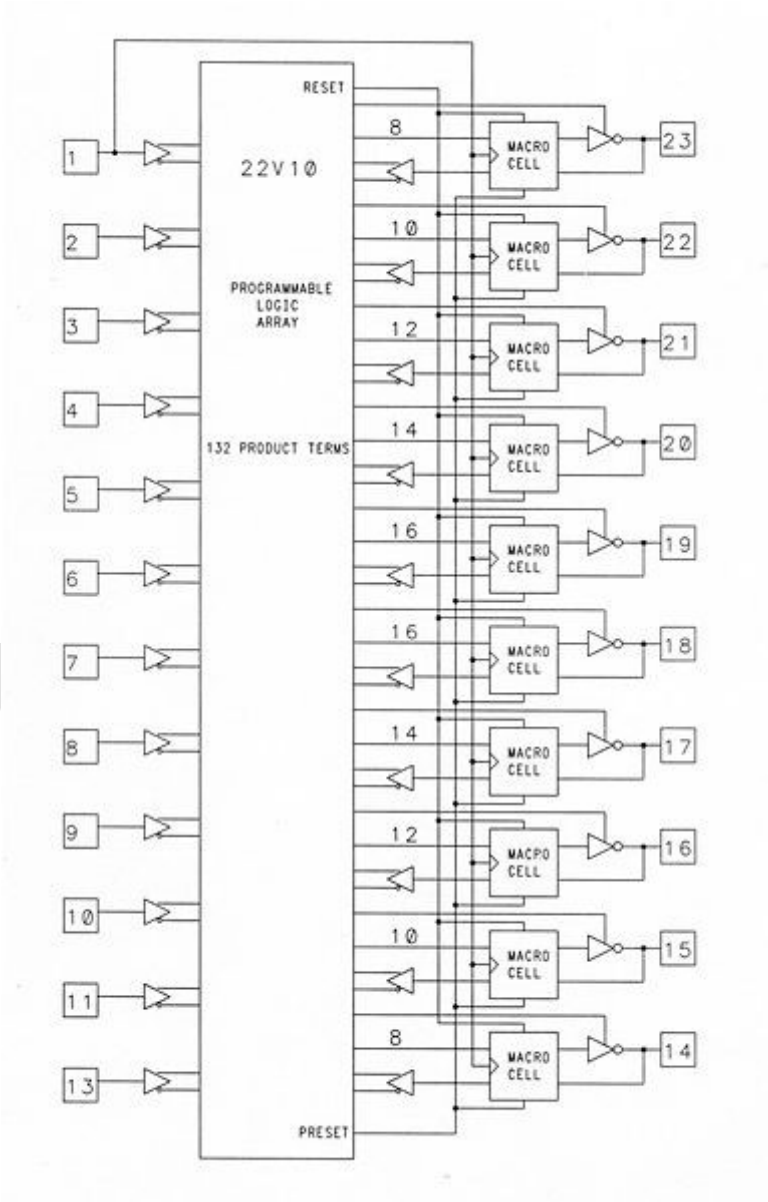
AMD 22V10 Output Macrocell

This fixed output structure often frustrated designers attempting to optimize the utility of PAL devices because output structures of different types were often required by their applications. (For example, one could not get 5 registered outputs with 3 active high

combinational outputs.) So, in June 1983 AMD introduced the 22V10, a 24 pin device with 10 output logic macrocells. Each macrocell could be configured by the user to be combinational or registered, active high or active low. The number of product term allocated to an output varied from 8 to 16. This one device could replace all of the 24 pin fixed function PAL devices. Members of the PAL "V" ("variable") series included the PAL16V8, PAL20V8 and PAL22V10.



PAL 16R4 Block Diagram



AMD 22V10 Block Diagram

Programming PALs

PALs were programmed electrically using binary patterns (as JEDEC ASCII/hexadecimal files) and a special electronic programming system available from either the manufacturer or a third-party, such as DATA/IO. In addition to single-unit device programmers, device feeders and gang programmers were often used when more than just a few PALs needed to be programmed. (For large volumes, electrical programming costs could be eliminated by having the manufacturer fabricate a custom metal mask used to program the customers' patterns at the time of manufacture; MMI used the term "hard array logic" (HAL) to refer to devices programmed in this way.)

Programming languages

```
PAL16R4 PAL              PAL DESIGN SPECIFICATION
CNT4SC
4 bit counter with synchronous clear
Michael Holley and Dave Pellerin
Clk  Clear  NC  NC  NC  NC  NC  NC  NC  GND
 OE  NC     NC /Q3 /Q2 /Q1 /Q0  NC  NC  VCC

  Q3 :=  Clear
       + /Q3 * /Q2 * /Q1 * /Q0
       +  Q3 *  Q0
       +  Q3 *  Q1
       +  Q3 *  Q2

  Q2 :=  Clear
       + /Q2 * /Q1 * /Q0
       +  Q2 *  Q0
       +  Q2 *  Q1

  Q1 :=  Clear
       + /Q1 * /Q0
       +  Q1 *  Q0

  Q0 :=  Clear
       + /Q0

FUNCTION TABLE
OE Clear Clk   /Q0 /Q1 /Q2 /Q3
------------------------------
 L   H    C     L   L   L   L
 L   L    C     H   L   L   L
 L   L    C     L   H   L   L
 L   L    C     H   H   L   L
 L   L    C     L   L   H   L
 L   H    C     L   L   L   L
------------------------------
```

PALASM design of a 4-bit counter.

Though some engineers programmed PAL devices by manually editing files containing the binary fuse pattern data, most opted to design their logic using a hardware description language (HDL) such as Data I/O's ABEL, Logical Devices' CUPL, or MMI's PALASM. These were computer-assisted design (CAD) (now referred to as "design automation") programs which translated (or "compiled") the designers' logic equations into binary fuse map files used to program (and often test) each device.

PALASM

The PALASM (from "PAL assembler") language was used to express boolean equations for the outputs pins in a text file which was then converted to the 'fuse map' file for the programming system using a vendor-supplied program; later the option of translation from schematics became common, and later still, 'fuse maps' could be 'synthesized' from an HDL (hardware description language,) such as Verilog.

The PALASM compiler was written by MMI in FORTRAN IV on an IBM 370/168. MMI made the source code available to users at no cost. By 1983, MMI customers ran versions on the DEC PDP-11, Data General NOVA, Hewlett-Packard HP2100, MDS800 and others.

ABEL

Data I/O Corporation released ABEL.

CUPL

Logical Devices, Inc. released the Universal Compiler for Programmable Logic (CUPL), which ran under MSDOS on the IBM PC and is currently available as an integrated development package for Microsoft Windows.

Device programmers

Popular device programmers included Data I/O Corporation's Model 60A Logic Programmer and Model 2900.

One of the very first PAL Programmers was the Structured Design "SD-20". They had the PALASM software built-in and only required a CRT terminal to enter the equations and view the fuse plots. After fusing, the outputs of the PAL could be verified if test vectors were entered in the source file.

Successors

After MMI succeeded with the 20-pin PAL parts introduced circa 1978, AMD introduced the 24-pin 22V10 PAL with additional features. After buying out MMI (circa 1987), AMD spun off a consolidated operation as Vantis, and that business was acquired by Lattice Semiconductor in 1989.

Altera introduced the EP300 (first CMOS PAL) in 1983 and later moved into the FPGA business.

Lattice Semiconductor introduced the generic array logic (GAL) family in 1985, with functional equivalents of the "V" series PALs that used reprogrammable logic planes based on EEPROM (electrically eraseable programmable read-only memory) technology.

National Semiconductor was a "second source" of GAL parts. AMD introduced a similar family called PALCE. In general one GAL part is able to function as any of the similar family PAL devices. For example the 16V8 GAL is able to replace the 16L8, 16H8, 16H6, 16H4, 16H2 and 16R8 PALs (and many others besides).

ICT (International CMOS Technology) introduced the PEEL 18CV8 in 1986. The 20-pin CMOS EEPROM part could be used in place of any of the registered-output bipolar PALs and used much less power.

Larger-scale programmable logic devices were introduced by Atmel, Lattice Semiconductor, and others. These devices extended the PAL architecture by including multiple logic planes and/or burying logic macrocells within the logic plane(s). The term "complex programmable logic device" (CPLD) was introduced to differentiate these devices from their PAL and GAL predecessors, which were then sometimes referred to as "simple programmable logic devices" or SPLDs.

Another large programmable logic device is the "field-programmable gate array" or FPGA. This term is often used to describe devices currently made by Altera and Xilinx.

Chapter- 13

Programmable Logic Device

A **programmable logic device** or PLD is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed, that is, reconfigured.

Using a ROM as a PLD

Before PLDs were invented, read-only memory (ROM) chips were used to create arbitrary combinational logic functions of a number of inputs. Consider a ROM with *m* inputs (the address lines) and *n* outputs (the data lines). When used as a memory, the ROM contains 2^m words of *n* bits each. Now imagine that the inputs are driven not by an *m*-bit address, but by *m* independent logic signals. Theoretically, there are 2^m possible Boolean functions of these *m* signals, but the structure of the ROM allows just 2^n of these functions to be produced at the output pins. The ROM therefore becomes equivalent to *n* separate logic circuits, each of which generates a chosen function of the *m* inputs.

The advantage of using a ROM in this way is that any conceivable function of the *m* inputs can be made to appear at any of the *n* outputs, making this the most general-purpose combinational logic device available. Also, PROMs (programmable ROMs), EPROMs (ultraviolet-erasable PROMs) and EEPROMs (electrically erasable PROMs) are available that can be programmed using a standard PROM programmer without requiring specialised hardware or software. However, there are several disadvantages:

- they are usually much slower than dedicated logic circuits,
- they cannot necessarily provide safe "covers" for asynchronous logic transitions so the PROM's outputs may glitch as the inputs switch,
- they consume more power,
- they are often more expensive than programmable logic, especially if high speed is required.

Since most ROMs do not have input or output registers, they cannot be used stand-alone for sequential logic. An external TTL register was often used for sequential designs such

as state machines. Common EPROMs, for example the 2716, are still sometimes used in this way by hobby circuit designers, who often have some lying around. This use is sometimes called a 'poor man's PAL'.

Early programmable logic

In 1969, Motorola offered the XC157, a mask-programmed gate array with 12 gates and 30 uncommitted input/output pins.

In 1970, Texas Instruments developed a mask-programmable IC based on the IBM read-only associative memory or ROAM. This device, the TMS2000, was programmed by altering the metal layer during the production of the IC. The TMS2000 had up to 17 inputs and 18 outputs with 8 JK flip flop for memory. TI coined the term Programmable Logic Array for this device.

In 1971, General Electric Company (GE) was developing a programmable logic device based on the new PROM technology. This experimental device improved on IBM's ROAM by allowing multilevel logic. Intel had just introduced the floating-gate UV erasable PROM so the researcher at GE incorporated that technology. The GE device was the first erasable PLD ever developed, predating the Altera EPLD by over a decade. GE obtained several early patents on programmable logic devices.

In 1973 National Semiconductor introduced a mask-programmable PLA device (DM7575) with 14 inputs and 8 outputs with no memory registers. This was more popular than the TI part but cost of making the metal mask limited its use. The device is significant because it was the basis for the field programmable logic array produced by Signetics in 1975, the 82S100. (Intersil actually beat Signetics to market but poor yield doomed their part.)

In 1974 GE entered into an agreement with Monolithic Memories to develop a mask-programmable logic device incorporating the GE innovations. The device was named the 'Programmable Associative Logic Array' or PALA. The MMI 5760 was completed in 1976 and could implement multilevel or sequential circuits of over 100 gates. The device was supported by a GE design environment where Boolean equations would be converted to mask patterns for configuring the device. The part was never brought to market.

PAL

MMI introduced a breakthrough device in 1978, the Programmable Array Logic or PAL. The architecture was simpler than that of Signetics FPLA because it omitted the programmable OR array. This made the parts faster, smaller and cheaper. They were available in 20 pin 300 mil DIP packages while the FPLAs came in 28 pin 600 mil packages. The PAL Handbook demystified the design process. The PALASM design software (PAL Assembler) converted the engineers' Boolean equations into the fuse pattern required to program the part. The PAL devices were soon second-sourced by National Semiconductor, Texas Instruments and AMD.

After MMI succeeded with the 20-pin PAL parts, AMD introduced the 24-pin 22V10 PAL with additional features. After buying out MMI (1987), AMD spun off a consolidated operation as Vantis, and that business was acquired by Lattice Semiconductor in 1999.

There are also PLA's : Programmable Logic Array.

GALs

Lattice GAL 16V8 and 20V8

An innovation of the PAL was the **generic array logic** device, or **GAL**, invented by Lattice Semiconductor in 1985. This device has the same logical properties as the PAL but can be erased and reprogrammed. The GAL is very useful in the prototyping stage of a design, when any bugs in the logic can be corrected by reprogramming. GALs are programmed and reprogrammed using a PAL programmer, or by using the in-circuit programming technique on supporting chips.

Lattice GALs combine CMOS and electrically erasable (E^2) floating gate technology for a high-speed, low-power logic device.

A similar device called a **PEEL** (**programmable electrically erasable logic**) was introduced by the International CMOS Technology (ICT) corporation.

CPLDs

PALs and GALs are available only in small sizes, equivalent to a few hundred logic gates. For bigger logic circuits, complex PLDs or CPLDs can be used. These contain the equivalent of several PALs linked by programmable interconnections, all in one integrated circuit. CPLDs can replace thousands, or even hundreds of thousands, of logic gates.

Some CPLDs are programmed using a PAL programmer, but this method becomes inconvenient for devices with hundreds of pins. A second method of programming is to solder the device to its printed circuit board, then feed it with a serial data stream from a personal computer. The CPLD contains a circuit that decodes the data stream and configures the CPLD to perform its specified logic function.

Each manufacturer has a proprietary name for this programming system. For example, Lattice Semiconductor calls it "in-system programming". However, these proprietary systems are beginning to give way to a standard from the Joint Test Action Group, JTAG.

FPGAs

While PALs were busy developing into GALs and CPLDs (all discussed above), a separate stream of development was happening. This type of device is based on gate array technology and is called the field-programmable gate array (FPGA). Early examples of FPGAs are the 82s100 array, and 82S105 sequencer, by Signetics, introduced in the late 1970s. The 82S100 was an array of AND terms. The 82S105 also had flip flop functions.

FPGAs use a grid of logic gates, and once stored, the data doesn't change, similar to that of an ordinary gate array. The term "field-programmable" means the device is programmed by the customer, not the manufacturer.

FPGAs are usually programmed after being soldered down to the circuit board, in a manner similar to that of larger CPLDs. In most larger FPGAs the configuration is volatile, and must be re-loaded into the device whenever power is applied or different functionality is required. Configuration is typically stored in a configuration PROM or EEPROM. EEPROM versions may be in-system programmable (typically via JTAG).

The difference between FPGAs and CPLDs is that FPGAs are internally based on Look-up tables (LUTs) whereas CPLDs form the logic functions with sea-of-gates (e.g. sum of products). CPLDs are meant for simpler designs while FPGAs are meant for more complex designs. In general, CPLDs are a good choice for wide combinational logic applications, whereas FPGAs are more suitable for large state machines (i.e. microprocessors).

Other variants

At present, much interest exists in reconfigurable systems. These are microprocessor circuits that contain some fixed functions and other functions that can be altered by code running on the processor. Designing self-altering systems requires engineers to learn new methods, and that new software tools be developed.

PLDs are being sold now that contain a microprocessor with a fixed function (the so-called *core*) surrounded by programmable logic. These devices let designers concentrate on adding new features to designs without having to worry about making the microprocessor work.

How PLDs retain their configuration

A PLD is a combination of a logic device and a memory device. The memory is used to store the pattern that was given to the chip during programming. Most of the methods for storing data in an integrated circuit have been adapted for use in PLDs. These include:

- Silicon antifuses
- SRAM
- EPROM or EEPROM cells
- Flash memory

Silicon antifuses are the storage elements used in the PAL, the first type of PLD. These are connections that are made by applying a voltage across a modified area of silicon inside the chip. They are called antifuses because they work in the opposite way to normal fuses, which begin life as connections until they are broken by an electric current.

SRAM, or static RAM, is a volatile type of memory, meaning that its contents are lost each time the power is switched off. SRAM-based PLDs therefore have to be programmed every time the circuit is switched on. This is usually done automatically by another part of the circuit.

An EPROM cell is a MOS (metal-oxide-semiconductor) transistor that can be switched on by trapping an electric charge permanently on its gate electrode. This is done by a PAL programmer. The charge remains for many years and can only be removed by exposing the chip to strong ultraviolet light in a device called an EPROM eraser.

Flash memory is non-volatile, retaining its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for PLD memory.

As of 2005, most CPLDs are electrically programmable and erasable, and non-volatile. This is because they are too small to justify the inconvenience of programming internal SRAM cells every time they start up, and EPROM cells are more expensive due to their ceramic package with a quartz window.

PLD programming languages

Many PAL programming devices accept input in a standard file format, commonly referred to as 'JEDEC files'.They are analogous to software compilers. The languages used as source code for logic compilers are called hardware description languages, or HDLs.

PALASM and ABEL are frequently used for low-complexity devices, while Verilog and VHDL are popular higher-level description languages for more complex devices. The more limited ABEL is often used for historical reasons, but for new designs VHDL is more popular, even for low-complexity designs.

PLD programming devices

A device programmer is used to transfer the boolean logic pattern into the programmable device. In the early days of programmable logic, every PLD manufacturer also produced a specialized device programmer for its family of logic devices. Later, universal device programmers came onto the market that supported several logic device families from different manufacturers. Today's device programmers usually can program common PLDs (mostly PAL/GAL equivalents) from all existing manufacturers. Common file formats used to store the boolean logic pattern (fuses) are JEDEC, Altera POF (Programmable Object File), or Xilinx BITstream.

Chapter- 14

Logic Gate

A **logic gate** is a physical model of a Boolean function, that is, it performs a logical operation on one or more logic inputs and produces a single logic output. Logic gates are primarily implemented electronically using diodes or transistors, but can also be constructed using electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements.

With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

Background

The simplest form of electronic logic is diode logic. This allows AND and OR gates to be built, but not inverters, and so is an incomplete form of logic. Further, without some kind of amplification it is not possible to have such basic logic operations cascaded as required for more complex logic functions. To build a functionally complete logic system, relays, valves (vacuum tubes), or transistors can be used. The simplest family of logic gates using bipolar transistors is called resistor-transistor logic (RTL). Unlike diode logic gates, RTL gates can be cascaded indefinitely to produce more complex logic functions. These gates were used in early integrated circuits. For higher speed, the resistors used in RTL were replaced by diodes, leading to diode-transistor logic (DTL). Transistor-transistor logic (TTL) then supplanted DTL with the observation that one transistor could do the job of two diodes even more quickly, using only half the space. In virtually every type of contemporary chip implementation of digital systems, the bipolar transistors have been replaced by complementary field-effect transistors (MOSFETs) to reduce size and power consumption still further, thereby resulting in complementary metal–oxide–semiconductor (CMOS) logic.

For small-scale logic, designers now use prefabricated logic gates from families of devices such as the TTL 7400 series by Texas Instruments and the CMOS 4000 series by RCA, and their more recent descendants. Increasingly, these fixed-function logic gates

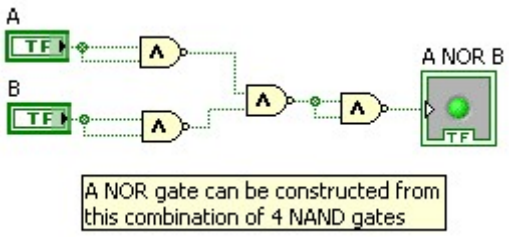
are being replaced by programmable logic devices, which allow designers to pack a large number of mixed logic gates into a single integrated circuit. The field-programmable nature of programmable logic devices such as FPGAs has removed the 'hard' property of hardware; it is now possible to change the logic design of a hardware system by reprogramming some of its components, thus allowing the features or function of a hardware implementation of a logic system to be changed.

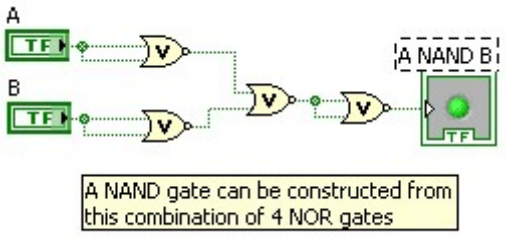
Electronic logic gates differ significantly from their relay-and-switch equivalents. They are much faster, consume much less power, and are much smaller (all by a factor of a million or more in most cases). Also, there is a fundamental structural difference. The switch circuit creates a continuous metallic path for current to flow (in either direction) between its input and its output. The semiconductor logic gate, on the other hand, acts as a high-gain voltage amplifier, which sinks a tiny current at its input and produces a low-impedance voltage at its output. It is not possible for current to flow between the output and the input of a semiconductor logic gate.

Another important advantage of standardised integrated circuit logic families, such as the 7400 and 4000 families, is that they can be cascaded. This means that the output of one gate can be wired to the inputs of one or several other gates, and so on. Systems with varying degrees of complexity can be built without great concern of the designer for the internal workings of the gates, provided the limitations of each integrated circuit are considered.

The output of one gate can only drive a finite number of inputs to other gates, a number called the 'fanout limit'. Also, there is always a delay, called the 'propagation delay', from a change in input of a gate to the corresponding change in its output. When gates are cascaded, the total propagation delay is approximately the sum of the individual delays, an effect which can become a problem in high-speed circuits. Additional delay can be caused when a large number of inputs are connected to an output, due to the distributed capacitance of all the inputs and wiring and the finite amount of current that each output can provide.

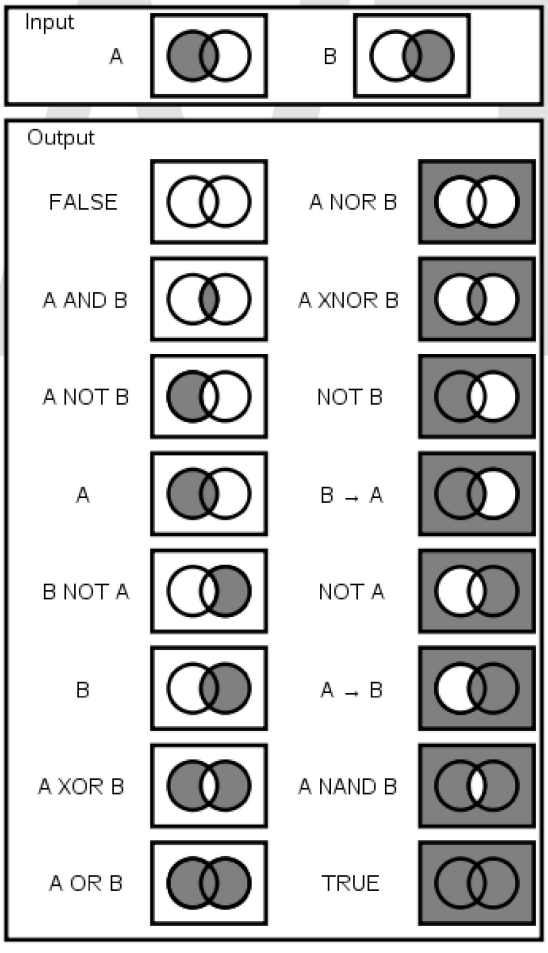
Logic gates





All other types of Boolean logic gates (i.e., AND, OR, NOT, XOR, XNOR) can be created from a suitable network of NAND gates. Similarly all gates can be created from a network of NOR gates. Historically, NAND gates were easier to construct from MOS technology and thus NAND gates served as the first pillar of Boolean logic in electronic computation.

For an input of 2 variables, there are 16 possible boolean algebraic functions. These 16 functions are enumerated below, together with their outputs for each combination of inputs variables.



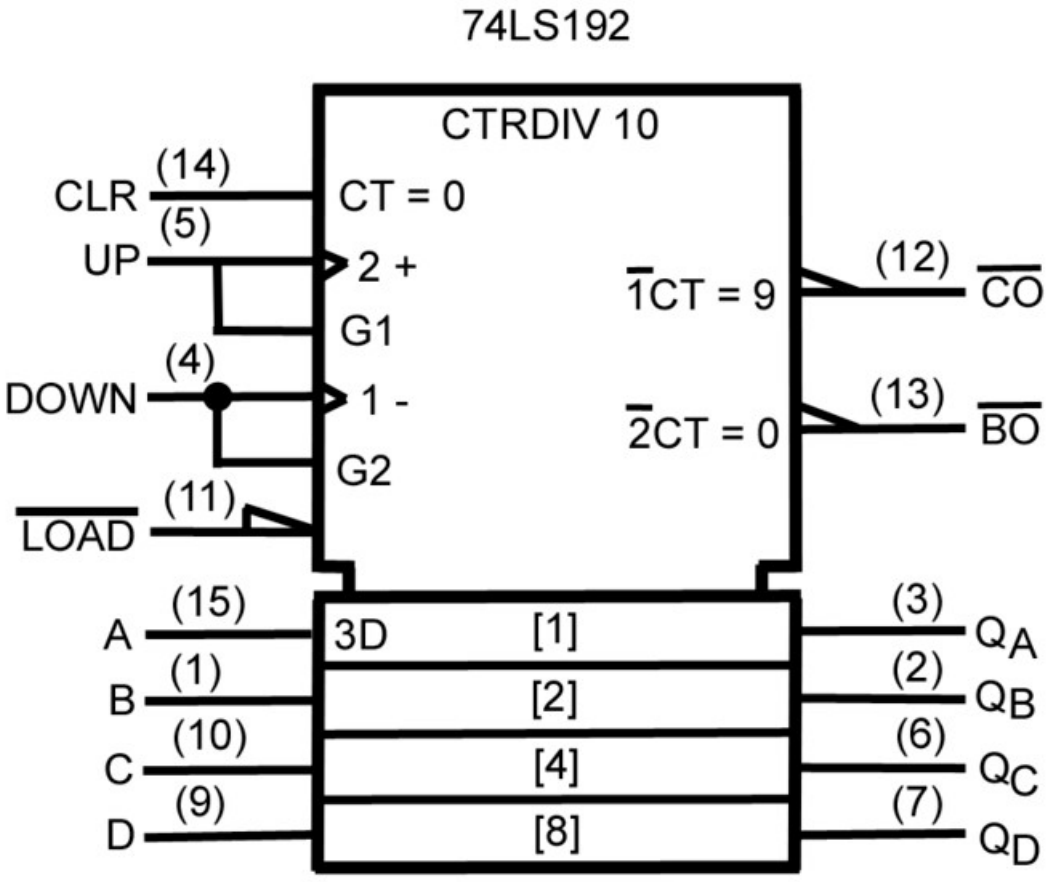
Venn Diagrams for Logic Gates

	A	**0**	**0**	**1**	**1**	
INPUT						**Meaning**
	B	**0**	**1**	**0**	**1**	
	FALSE	0	0	0	0	Whatever *A* and *B*, the output is false. Contradiction.
	A AND B	0	0	0	1	Output is true if and only if (iff) both *A* and *B* are true.
	A $\not\rightarrow$ B	0	0	1	0	*A* doesn't imply *B*. True iff *A* but not *B*.
	A	0	0	1	1	True whenever *A* is true.
	A $\not\leftarrow$ B	0	1	0	0	*A* is not implied by *B*. True iff not *A* but *B*.
	B	0	1	0	1	True whenever *B* is true.
OUTPUT	A XOR B	0	1	1	0	True iff *A* is not equal to *B*.
	A OR B	0	1	1	1	True iff *A* is true, or *B* is true, or both.
	A NOR B	1	0	0	0	True iff neither *A* nor *B*.
	A XNOR B	1	0	0	1	True iff *A* is equal to *B*.
	NOT B	1	0	1	0	True iff *B* is false.
	A $\leftarrow$ B	1	0	1	1	*A* is implied by *B*. False if not *A* but *B*, otherwise true.
	NOT A	1	1	0	0	True iff *A* is false.

A ⟶ B	1	1	0	1	*A* implies *B*. False if *A* but not *B*, otherwise true.
A NAND B	1	1	1	0	*A* and *B* are not both true.
TRUE	1	1	1	1	Whatever *A* and *B*, the output is true. Tautology.

The four functions denoted by arrows are the logical implication functions. These functions are not usually implemented as elementary circuits, but rather as combinations of a gate with an inverter at one input.

Symbols

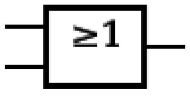
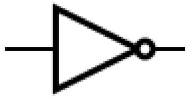
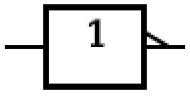


A synchronous 4-bit up/down decade counter symbol (74LS192) in accordance with ANSI/IEEE Std. 91-1984 and IEC Publication 60617-12.

There are two sets of symbols in common use, both now defined by ANSI/IEEE Std 91-1984 and its supplement ANSI/IEEE Std 91a-1991. The "distinctive shape" set, based on traditional schematics, is used for simple drawings, and derives from MIL-STD-806 of the 1950s and 1960s. It is sometimes unofficially described as "military", reflecting its origin. The "rectangular shape" set, based on IEC 60617-12, has rectangular outlines for all types of gate, and allows representation of a much wider range of devices than is possible with the traditional symbols. The IEC's system has been adopted by other standards, such as EN 60617-12:1999 in Europe and BS EN 60617-12:1999 in the United Kingdom.

The goal of IEEE Std 91-1984 was to provide a uniform method of describing the complex logic functions of digital circuits with schematic symbols. These functions were more complex than simple AND and OR gates. They could be medium scale circuits such as a 4-bit counter to a large scale circuits such as a microprocessor. IEC 617-12 and its successor IEC 60617-2 do not include the "distinctive shape" symbols. These are, however, included in ANSI/IEEE 91 (and 91a) with this note: "The distinctive-shape symbol is, according to IEC Publication 617, Part 12, not preferred, but is not considered to be in contradiction to that standard." This compromise was reached between the respective IEEE and IEC working groups to permit the IEEE and IEC standards to be in mutual compliance with one another.

In the 1980s, schematics were the predominant method to design both circuit boards and custom ICs known as gate arrays. Today custom ICs and the field-programmable gate array are typically designed with Hardware Description Languages (HDL) such as Verilog or VHDL.

Type	Distinctive shape	Rectangular shape	Boolean algebra between A & B	Truth table
AND			$A \cdot B$	**INPUT** **OUTPUT** A B A AND B 0 0 0 0 1 0 1 0 0 1 1 1
OR			$A + B$	**INPUT** **OUTPUT** A B A OR B 0 0 0 0 1 1 1 0 1 1 1 1
NOT			$\overline{A}$	**INPUT** **OUTPUT** A NOT A

0	1
1	0

In electronics a NOT gate is more commonly called an inverter. The circle on the symbol is called a *bubble*, and is used in logic diagrams to indicate a logical inversion between the external logic state and the internal logic state (1 to 0 or vice versa). On a circuit diagram it must be accompanied by a statement asserting that the *positive logic convention* or *negative logic convention* is being used (high voltage level = 1 or high voltage level = 0, respectively). The *wedge* is used in circuit diagrams to directly indicate an active-low (high voltage level = 0) input or output without requiring a uniform convention throughout the circuit diagram. This is called *Direct Polarity Indication*. Both the *bubble* and the *wedge* can be used on distinctive-shape and rectangular-shape symbols on circuit diagrams, depending on the logic convention used. On pure logic diagrams, only the *bubble* is meaningful.

NAND

$\overline{A \cdot B}$

INPUT		OUTPUT
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

NOR

$\overline{A + B}$

INPUT		OUTPUT
A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

XOR

$A \oplus B$

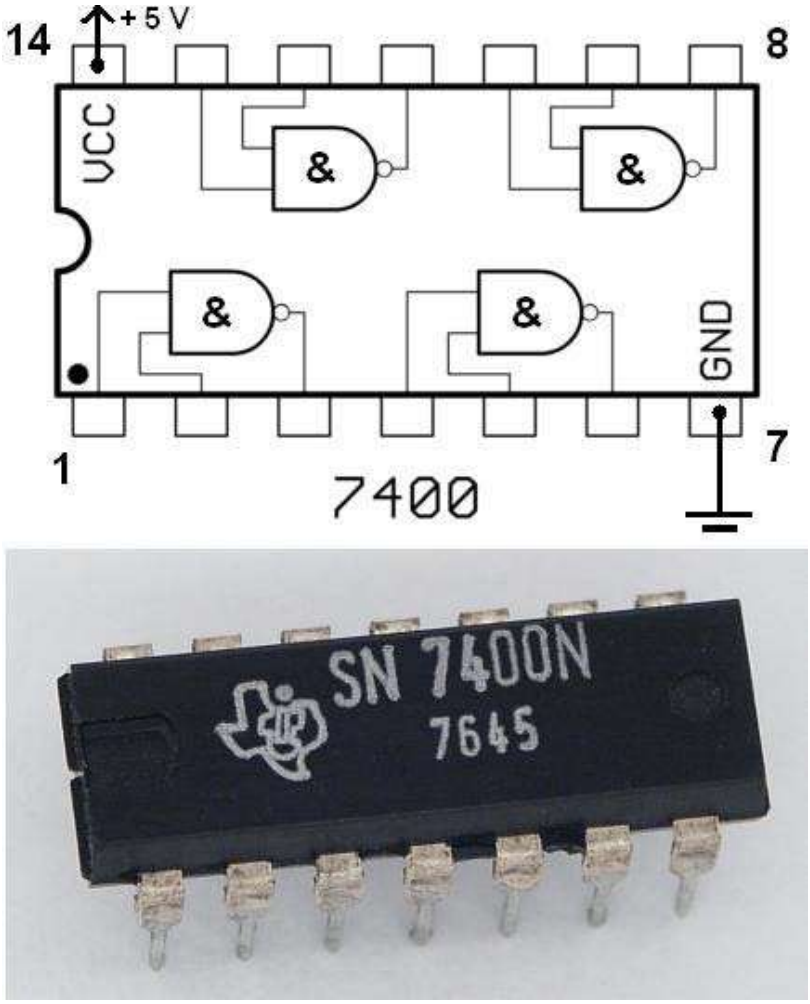
INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

$\overline{A \oplus B}$ or $A \odot B$

INPUT		OUTPUT
A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Charles Sanders Peirce (1880) showed that NAND gates alone (or alternatively NOR gates alone) can be used to reproduce the functions of all the other logic gates, but his work on it was unpublished until 1935. The first published proof was by Henry M. Sheffer in 1913.



The 7400 chip, containing four NANDs. The two additional pins supply power (+5 V) and connect the ground.

Two more gates are the exclusive-OR or XOR function and its inverse, exclusive-NOR or XNOR. The two input Exclusive-OR is true only when the two input values are *different*, false if they are equal, regardless of the value. If there are more than two inputs, the gate generates a true at its output if the number of trues at its input is *odd* (). In practice, these gates are built from combinations of simpler logic gates.

De Morgan equivalent symbols

By use of De Morgan's theorem, an *AND* gate can be turned into an *OR* gate by inverting the sense of the logic at its inputs and outputs. This leads to a separate set of symbols

with inverted inputs and the opposite core symbol. These symbols can make circuit diagrams for circuits using active low signals much clearer and help to show accidental connection of an active high output to an active low input or vice-versa.

Symbolically, a NAND gate can also be shown using the OR shape with bubbles on its inputs, and a NOR gate can be shown as an AND gate with bubbles on its inputs. The bubble signifies a logic inversion. This reflects the equivalency due to De Morgans law, but it also allows a diagram to be read more easily, or a circuit to be mapped onto available physical gates in packages easily, since any circuit node that has bubbles at both ends can be replaced by a simple bubble-less connection and a suitable change of gate. If the NAND is drawn as OR with input bubbles, and a NOR as AND with input bubbles, this gate substitution occurs automatically in the diagram (effectively, bubbles "cancel"). This is commonly seen in real logic diagrams - thus the reader must not get into the habit of associating the shapes exclusively as OR or AND shapes, but also take into account the bubbles at both inputs and outputs in order to determine the "true" logic function indicated.

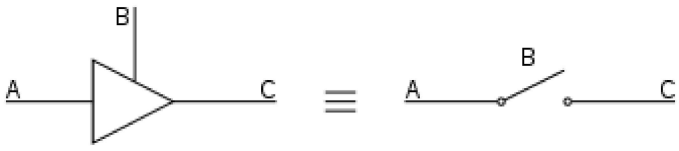
All logic relations can be realized by using NAND gates (this can also be done using NOR gates). De Morgan's theorem is most commonly used to transform all logic gates to NAND gates or NOR gates. This is done mainly since it is easy to buy logic gates in bulk and because many electronics labs stock only NAND and NOR gates.

Data storage

Logic gates can also be used to store data. A storage element can be constructed by connecting several gates in a "latch" circuit. More complicated designs that use clock signals and that change only on a rising or falling edge of the clock are called edge-triggered "flip-flops". The combination of multiple flip-flops in parallel, to store a multiple-bit value, is known as a register. When using any of these gate setups the overall system has memory; it is then called a sequential logic system since its output can be influenced by its previous state(s).

These logic circuits are known as computer memory. They vary in performance, based on factors of speed, complexity, and reliability of storage, and many different types of designs are used based on the application.

Three-state logic gates



A tristate buffer can be thought of as a switch. If *B* is on, the switch is closed. If B is off, the switch is open.

Three-state, or 3-state, logic gates are a type of logic gates that have three states of the output: high (H), low (L) and high-impedance (Z). The high-impedance state plays no role in the logic, which remains strictly binary. These devices are used on buses also known as the Data Buses of the CPU to allow multiple chips to send data. A group of three-states driving a line with a suitable control circuit is basically equivalent to a multiplexer, which may be physically distributed over separate devices or plug-in cards.

In electronics, a high output would mean the output is sourcing current from the positive power terminal (positive voltage). A low output would mean the output is sinking current to the negative power terminal (zero voltage). High impedance would mean that the output is effectively disconnected from the circuit.

'Tri-state', a widely-used synonym of 'three-state', is a trademark of the National Semiconductor Corporation.

Miscellaneous

Logic circuits include such devices as multiplexers, registers, arithmetic logic units (ALUs), and computer memory, all the way up through complete microprocessors, which may contain more than 100 million gates. In practice, the gates are made from field-effect transistors (FETs), particularly MOSFETs (metal–oxide–semiconductor field-effect transistors).

Compound logic gates AND-OR-Invert (AOI) and OR-AND-Invert (OAI) are often employed in circuit design because their construction using MOSFET's is simpler and more efficient than the sum of the individual gates.

In reversible logic, Toffoli gates are used.

History and development

Starting in 1898, Nikola Tesla filed for patents of devices containing logic gate circuits. Eventually, vacuum tubes replaced relays for logic operations. Lee De Forest's modification, in 1907, of the Fleming valve can be used as AND logic gate. Ludwig Wittgenstein introduced a version of the 16-row truth table, which is shown above, as proposition 5.101 of *Tractatus Logico-Philosophicus* (1921). Claude E. Shannon introduced the use of Boolean algebra in the analysis and design of switching circuits in 1937. Walther Bothe, inventor of the coincidence circuit, got part of the 1954 Nobel Prize in physics, for the first modern electronic AND gate in 1924. Active research is taking place in molecular logic gates.

Implementations

As of 2008, most logic gates are made of CMOS transistors. Often millions of logic gates are packaged in a single integrated circuit.

There are several logic families with different characteristics (power consumption, speed, cost, size) such as: RDL (resistor-diode logic), RTL (resistor-transistor logic), DTL (diode-transistor logic), TTL (transistor-transistor logic) and CMOS (complementary metal oxide semiconductor).

Many early electromechanical digital computers, such as the Harvard Mark I, were built from relay logic gates, using electro-mechanical relays.

It is also possible to make logic gates out of pneumatic devices, such as the Sorteberg relayor mechanical logic gates, including on a molecular scale. Logic gates have been made out of DNA and used to create a computer called MAYA.

Additionally, logic gates can be made from quantum mechanical effects (though quantum computing usually diverges from boolean design).

It is also possible to make photonic logic gates using non-linear optical effects.