CM100194: Systems Architecture I
Lecture Notes, part II

Jim Laird

Semester I: 2012/13

Contents

1 Boolean Logic and Circuits

We may use circuits constructed from logic gates to implement logic (e.g. conditionals), arithmetic, data storage and control cycles. Our main tool for analysing the behaviour of these circuits is Boolean logic, which allow us to represent their input/output behaviour as Boolean formulas (representing the output state) of Boolean variables (representing the input state). The logical operations on which Boolean algebra is based correspond to logic gates. They have operands and produce results whose values can be only two-fold, namely: TRUE and FALSE. These are called *Boolean* values and the operations on them are *Boolean* operations.

There are three fundamental logical operations, though they are not actually independent. A fourth operation is also commonly used.

- Logical AND:

  $$A.B$$

  Here A and B are Booleans, namely TRUE and FALSE, *not* numbers.

  We write AND using the multiplication operator, but it is important to realise this is *not* arithmetic multiplication of A and B. We might also see

  $$A \wedge B \quad A\&B \quad A \cap B \quad AB$$

  for AND.

  The result is TRUE if both A and B are TRUE; otherwise the result is FALSE.

- Logical (inclusive) OR:

  $$A + B$$

  This is *not* addition, we are just using the plus sign in a different way. We might also see

  $$A \vee B \quad A|B \quad A \cup B$$

  for OR.

  The result is TRUE if either A or B are TRUE; otherwise the result is FALSE.

- Logical NOT:

  $$A'$$

  We also find

  $$\sim A \quad \neg A \quad -A \quad \overline{A} \quad !A$$

  The result is TRUE if A is FALSE; the result is FALSE if A is TRUE.

- Logical Exclusive OR (EOR or XOR):

  $$A \text{ XOR } B$$

  We might also see

  $$A \oplus B \quad A \wedge B$$

  for XOR.

  The result is TRUE if A is TRUE and B is FALSE; the result is TRUE if A is false and B is TRUE; otherwise the result is FALSE.

  A XOR B is TRUE when A and B are *different*.

It is conventional and convenient to represent TRUE by the bit 1 and FALSE by the bit 0. **These are not numbers.** We can now write

A	0	0	1	1
B	0	1	0	1
$A.B$	0	0	0	1
$A+B$	0	1	1	1
A XOR B	0	1	1	0

A	0	1
A'	1	0

These are called *Truth Tables*.

These are also commonly written as

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

So we have, for example, $1.1 = 1$, $1.0 = 0$, $1 + 0 = 1$, $1 + 1 = 1$. **This is not arithmetic.**

1.1 Boolean Algebra

Combinatorial logic circuits (i.e. without feedback) correspond directly to formulas of *Boolean algebra*. An *algebra* is nothing complicated, just some class of data plus some operations on them. Thus arithmetic is an algebra on numbers. We are interested in an algebra on Boolean values, namely TRUE and FALSE. Our operations are AND, NOT, OR and so on.

What are the properties of this algebra?

We have already seen the truth tables

Operands		AND	OR
A	B	$A.B$	$A+B$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Operand	NOT
A	A'
0	1
1	0

Thus we can see that $A+B=B+A$, $A+0=A$ and so on.

	AND rule	OR rule	NOT rule
zero	$A.0=0$	$A+0=A$	$0'=1$
unit	$A.1=A$	$A+1=1$	$1'=0$
idempotent	$A.A=A$	$A+A=A$	$(A')'=A$
inverse	$A.A'=0$	$A+A'=1$	
commutative	$A.B=B.A$	$A+B=B+A$	
associative	$A.(B.C)=(A.B).C$	$A+(B+C)=(A+B)+C$	
distributive	$A.(B+C)=A.B+A.C$	$A+(B.C)=(A+B).(A+C)$	
DeMorgan	$(A.B)'=A'+B'$	$(A+B)'=A'.B'$	

The way we prove these assertions is to look at the truth tables. For example, $A+1=1$.

A	B	$A+B$	
0	0	0	
0	1	1	$\leftarrow$
1	0	1	
1	1	1	$\leftarrow$

in the indicated rows we have $A+1$ and the result column is 1 for all values of A. Thus $A+1=1$.

Another example, $A+A=A$.

A	B	$A+B$	
0	0	0	$\leftarrow$
0	1	1	
1	0	1	
1	1	1	$\leftarrow$

in the indicated rows we have $A=B$ and the result column is the same as A for all values of A. Thus $A+A=A$.

This is a proof by enumeration of cases, which is quite common for this kind of Boolean algebra.

Notice that $(A'.B')' = (A')' + (B')' = A + B$ by application of DeMorgan and idempotent. Thus we can define OR in terms of AND and NOT (or vice versa!).

It is very important to realise that A, B and C in the above rules stand for general formulae, so, for example, the unit rule says that $1 + A.B.C' = 1$ and the zero rule says that $0.A.B.(B + C + 1) = 0$ amongst an infinite number of other things.

1.2 Proving Boolean Identities by Algebra

Now we know these basic identities, we can use them to prove things. For example, to show that $A.(A+B) = A$ we can apply the Boolean equalities:

$$\begin{array}{rcll} A.(A+B) & = & A.A + A.B & \text{distributive} \\ & = & A + A.B & \text{idempotent} \\ & = & A.1 + A.B & \text{unit} \\ & = & A.(1+B) & \text{distributive} \\ & = & A.1 & \text{unit} \\ & = & A & \text{unit} \end{array}$$

Notice that we **do not have cancellation rules**. So $A + 1 = B + 1$ **does not imply** $A = B$. For example, $0 + 1 = 1 + 1$ but $0 \neq 1$.

1.3 Proving Boolean Identities by Truth Tables

The alternative to algebra is to use truth tables:

A	B	$A + B$	$A.(A + B)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

We see that the column for $A.(A + B)$ is identical to the column for A. Thus, $A.(A + B) = A$.

Another algebra proof:

$$\begin{array}{rcll} A + A.B & = & A.1 + A.B & \text{unit} \\ & = & A.(1+B) & \text{distributive} \\ & = & A.1 & \text{unit} \\ & = & A & \text{unit} \end{array}$$

So we have proved $A + A.B = A$. Proof by truth table is also easy.

A bigger proof:

$$\begin{array}{rcl} (A + B' + C).(A + B) & = & A.(A + B) + B'.(A + B) + C.(A + B) \\ & = & A.A + A.B + B'.A + B'.B + C.A + C.B \\ & = & A + A.B + A.B' + A.C + B.B' + B.C \\ & = & A.(1 + B + B' + C) + 0 + B.C \\ & = & A.1 + B.C \\ & = & A + B.C \end{array}$$

We are now led to ask whether there is a way to show that this is indeed the *simplest* form of the original Boolean function $(A + B' + C).(A + B)$.

To do this we need to develop the notion of a *standard form* of expression of a Boolean function.

1.4 Standard Forms

Given a function of n Boolean variables, it is always possible to reduce the function to a form in which there is series of terms joined by OR operations and in which each of the terms contains a combination of *all* n variables (or their complements) joined by AND operations.

Let us take an example and put it into this form:

$$\begin{aligned}
A + B.C &= A.(B + B') + B.C \\
&= A.B + A.B' + B.C \\
&= A.B.(C + C') + A.B' + B.C \\
&= A.B.C + A.B.C' + A.B' + B.C \\
&= A.B.C + A.B.C' + A.B'.(C + C') + B.C \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C.(A + A') \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + A.B.C + A'.B.C \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C
\end{aligned}$$

This is a standard form. It is known somewhat misleadingly as the *standard sum of products* form, even though we know they are *not* sums and products. It is also called *disjunctive normal form*, or DNF.

There is also a *standard product of sums* form, where we have a series of terms joined by AND operations and in which each of the terms contains a combination of *all* n variables (or their complements) joined by OR operations. This is also called *conjunctive normal form*, or CNF.

The above example becomes

$$A + B.C = (A + B + C).(A + B + C').(A + B' + C)$$

as a standard product of sums.

Exercise: show this.

The SofP form is generally used in preference to the PofS forms purely because they are psychologically easier to compute given our background in the rules of arithmetic. There is nothing to separate them mathematically.

1.5 Standard Forms and Truth Tables

There is an easy way to compute standard forms using truth tables (for small numbers of variables). For example

A	B	C	$B.C$	$A+B.C$		
0	0	0	0	0		$A+B+C$
0	0	1	0	0		$A+B+C'$
0	1	0	0	0		$A+B'+C$
0	1	1	1	1	$A'.B.C$	
1	0	0	0	1	$A.B'.C'$	
1	0	1	0	1	$A.B'.C$	
1	1	0	0	1	$A.B.C'$	
1	1	1	1	1	$A.B.C$	

In the truth table for $A+B.C$ we look at each row that has a 1 for $A+B.C$. Write down $A.B.C$ and put a prime against each variable that has a 0 in its column. Thus 011 becomes $A'.B.C$. Join all these terms with OR. Therefore we get

$$A'.B.C + A.B'.C' + A.B'.C + A.B.C' + A.B.C,$$

as before.

It is just as easy to get the PofS form: we look at each row that has a 0 for $A+B.C$. Write down $A+B+C$ and put a prime against each variable that has a 1 in its column. Thus 001 becomes $A+B+C'$. Join all these terms with AND. This time we get

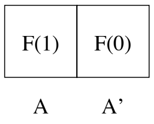
$$(A+B+C).(A+B+C').(A+B'+C)$$

for the PofS form.

1.6 Karnaugh Maps

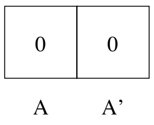
So far we have only made things more complicated, but this is just the first step on the path to the simplest form.

Given a function, F, of one Boolean variable, A, we can plot the function $F(A)$, whatever it is, on a map with two regions:

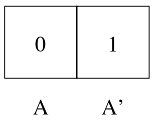


We put the value of $F(1)$ (be it 0 or 1) in the box marked A, and the value of $F(0)$ in the box marked A'.

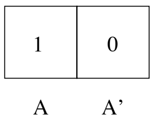
Here are the four possible function of one variable:



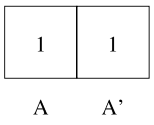
F(A) = 0



F(A)=A'



F(A)=A



F(A)=1

These are the four possible functions in truth table format:

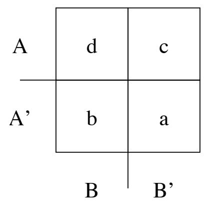
A	$F(A)$
0	0
1	0

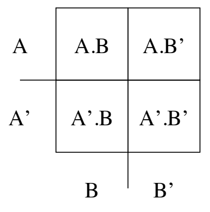
A	$F(A)$
0	1
1	0

A	$F(A)$
0	0
1	1

A	$F(A)$
0	1
1	1

We can extend this to two variables:





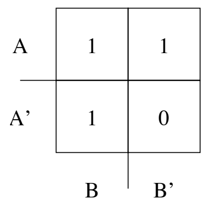
corresponding to the function

A	B	$F(A,B)$
0	0	a
0	1	b
1	0	c
1	1	d

So, for example, the function

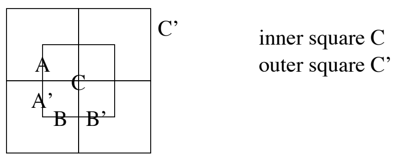
A	B	$F(A,B)$
0	0	0
0	1	1
1	0	1
1	1	1

is mapped as

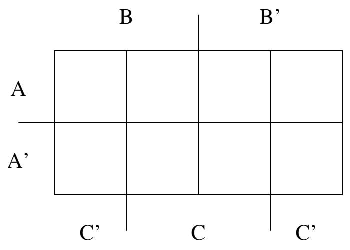


These are called *Karnaugh Maps*, after Maurice Karnaugh who popularised them in the early 1950s.

Also known as *Carroll Diagrams*, after Lewis Carroll, who was interested in using these diagrams to solve problems in logic. His form of drawing the diagrams was slightly different, but equivalent.



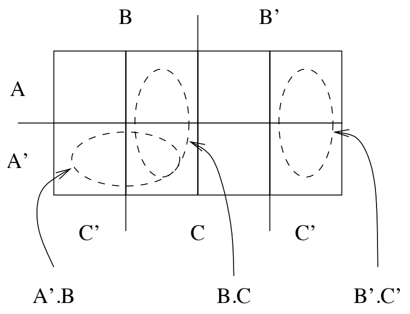
For three variables we get:

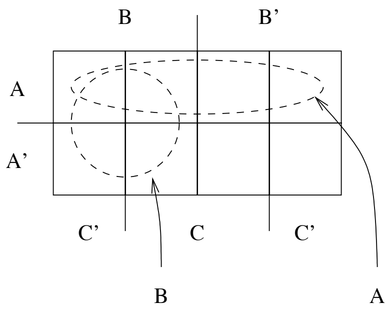


We should view this as a $2 \times 2 \times 2$ diagram unrolled flat.

The point of note is that *each cell in the diagram corresponds to one term in the standard sum of products form*. (And to the PofS form.) The cell marked by A, B, C' corresponds to the term $A.B.C'$. (And $(A' + B' + C)$.)

Moreover, pairs and quads of cells correspond to other terms: the OR of the cells.



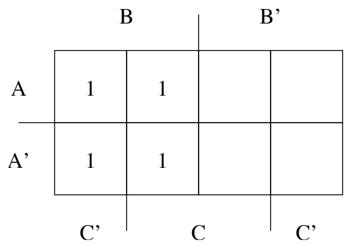


The whole of the diagram corresponds to 1, while the empty diagram corresponds to 0. The larger the are, the smaller the term.

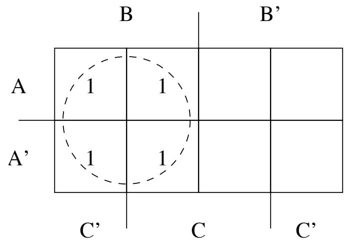
It is easy to find the term associated with an area. If the area lies completely within the region defined by a label (such as A or A'), write down that label. If the area covers a label and its complement (such as both A and A') do not include it.

So an area covering $A'.B.C'$ and $A'.B.C$ corresponds to the term $A'.B$. This, of course, is just using pictures to compute $A'.B.C' + A'.B.C = A'.B.(C' + C) = A'.B.1 = A'.B$.

So suppose we have a function $F = A.B.C + A'.B.C + A.B.C' + A'.B.C'$ in standard form. We plot on a Karnaugh map:



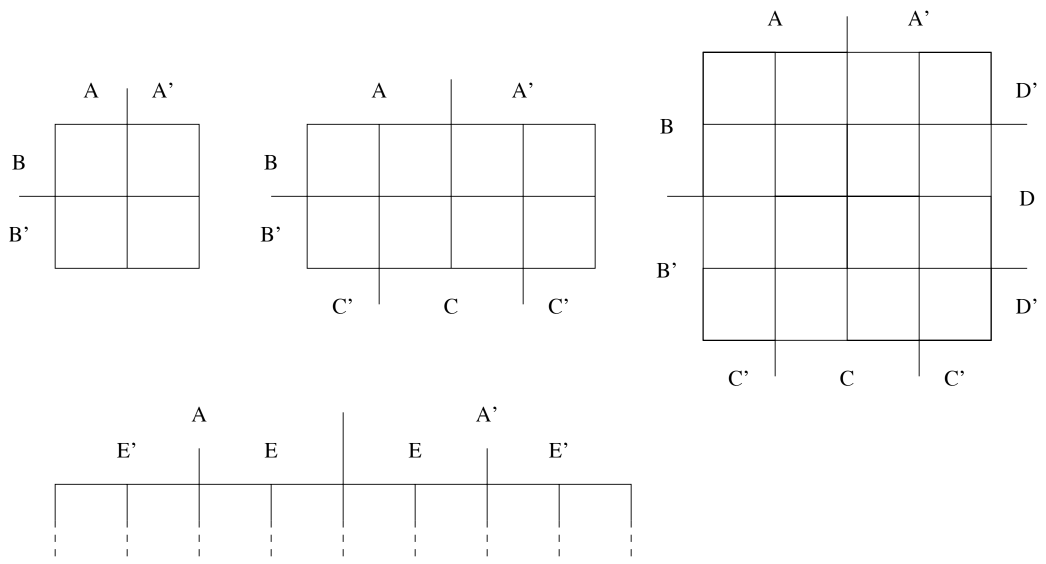
(the convention is not to bother writing in the 0s). Next, draw a region about the 1s:



and then determine what term this corresponds to. In this case it is just B. Therefore we have shown that $F = A.B.C + A'.B.C + A.B.C' + A'.B.C' = B$.

Karnaugh maps are our way of finding simplest forms.

The maps extend to more variables and more:



1.7 Simplifying a Boolean Function

Thus we can

1. Write it in the form of a standard sum of products;
2. Plot the product terms on a Karnaugh map;
3. Find the *smallest* number of the *biggest* valid groupings that will encompass all of the marked regions of the Karnaugh map; and
4. Read off the function that results.

To clarify: a *grouping* is a rectangle containing a power of 2 cells. Overlapping rectangles is fine, and we need to make them as big as possible (remember the bigger the area the smaller the term it represents).

Also, the rectangles might wrap around the edge of the diagram in the case of three or more variables.

lsls

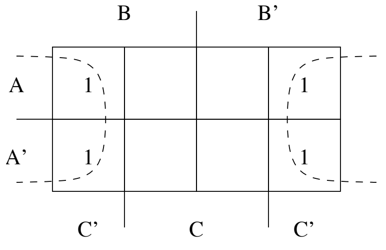
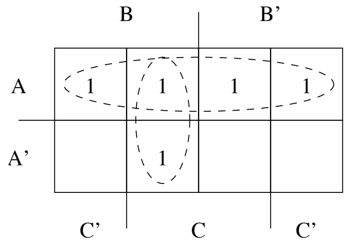


Figure 1: Wrapping in Karnaugh Maps

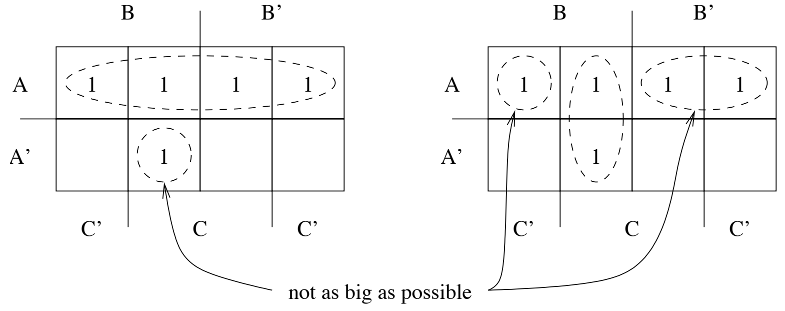
We can try this out on our function $F = A + B.C = A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C$. The map is



The two areas correspond to A (upper area) and $B.C$ (other area). Thus, the simplest form for F is

$$F = A + B.C$$

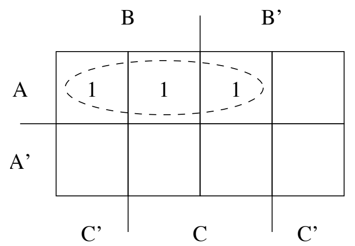
Note that other putative groupings



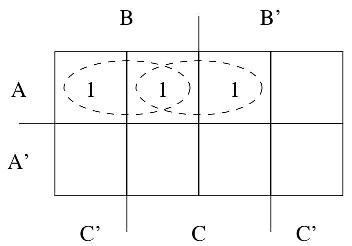
are not valid as they are not as large as possible.

Of course, they *do* correspond to valid formulae for F, but not *simplest* formulae. The first is $A + A'.B.C$, the second $A.B.C' + B.C + A.B'$. Both simplify to $A + B.C$

In this example:



Incorrect



Correct

the grouping is not a power of 2 elements. It must be a power of 2 to be able to determine the corresponding SofP term.

Exercise: Go through all 16 functions of two variables A and B and plot their Karnaugh maps. Then find their simplest forms.

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1
		0	$A.B$	$A.B'$	A	$A'.B$	B	$A.B'+$ $A'.B$	$A+B$

A	B	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1
		$A'.B'$	$A.B+$ $A'.B'$	B'	$A+B'$	A'	$A'+B$	$A'+B'$	1

We should, however, note that the simplest form of a function yielded by this method may not in fact be the most efficient in terms of the total number of logic gates needed to implement it. For example, the form $M = A.B + A.C$ which requires two AND gates and one OR gate, can be factorised into $M = A.(B + C)$ which needs only one AND gate and one OR gate.

The Karnaugh maps approach will find the simplest form *as a sum of products.*

Everything follows through for the product of sums form and similarly allows us to determine the simplest PofS form for a given formula. Sometimes it will be simpler (in count of logic gates) than the SofP form, sometimes not. In the case of $M = A.B+A.C$, the SofP simplest form is $A.B+A.C$, while the PofS simplest form is $A.(B+C)$.

1.8 Can't Happen and Don't Care Terms

Can't happen (CH) terms: some combinations of values of the input variables in Boolean functions may not occur in certain problems.

Don't care (DC) terms: the output of a Boolean function may be immaterial for certain combinations of the values of the input variables.

By marking these cases with an "X" in Karnaugh maps, we can choose to *use* or *ignore* them when simplifying a Boolean function.

They act like blank tiles in Scrabble: we can choose to think of them as either 1 or 0 as we wish. If they can't happen, we don't care what outputs they might give.

1.9 Simplifying Functions with "Can't Happen" Terms

Let us take a problem in which Boolean variables A, B, C and D are used to represent the digits of a decimal counter as shown below:

$$A'B'C'D' = 0000; \quad A'B'C'D = 0001; \quad A'B'CD' = 0010$$

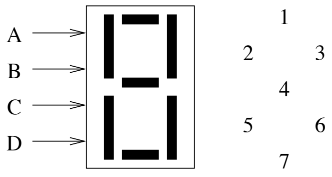
and so on, up to:

$$A'BCD = 0111; \quad AB'C'D' = 1000; \quad AB'C'D = 1001$$

The patterns 1010, 1011, 1100, 1101, 1110 and 1111 will not occur, as they are not patterns for decimal digits; they are the *can't happen* inputs in this function.

This encoding for the numbers 0-9 is called *Binary Coded Decimal*, or BCD. The number 42 would be represented by the eight bits 01000010. This encoding is used occasionally for simplicity, for example the track numbers on a CD are encoded in 8-bit BCD. This means the maximum number of tracks on a CD is 99.

We could use these four inputs to control a decimal LCD display:



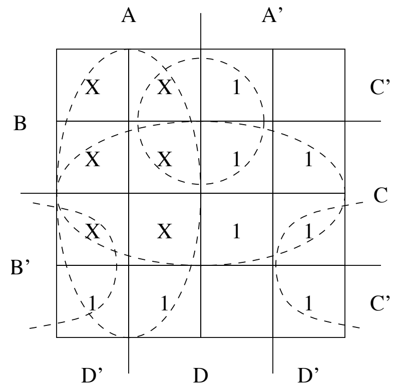
Each combination of inputs would light up a selection of the LED segments. We can regard this as seven different functions of the four input bits. So segment number 1 need to be on for inputs corresponding to decimal values 0, 2, 3, 5, 6, 7, 8 and 9. The truth table for segment 1 is

	A	B	C	D	on?
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1

	A	B	C	D	on?
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	CH
11	1	0	1	1	CH
12	1	1	0	0	CH
13	1	1	0	1	CH
14	1	1	1	0	CH
15	1	1	1	1	CH

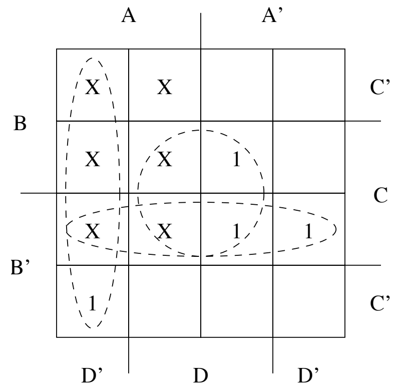
Similarly for the other six segment functions. So what is the simplest function that lights segment 1 for just the correct combination of inputs?

Here is the Karnaugh map



This corresponds to $A + C + B.D + B'.D'$.

In this next example we don't use all the Xs: construct the simplest Boolean function that will give an output if a decimal digit 2, 3, 7 or 8 occurs (but will give no output for any other decimal digit).



This corresponds to $A.D' + B'.C + C.D$.

1.10 Simplifying Functions with "Don't Care" Terms

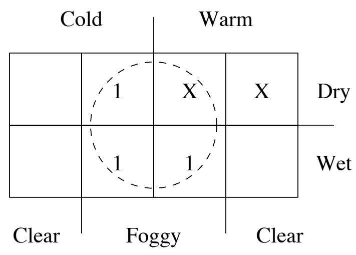
The nationalised electricity generation industry in the small and unimportant nation of Dementia has been ordered by the government to have its generation plant ready to run if the weather is

- both cold and foggy; or
- both foggy and wet.

The government has, however, told the industry that, as a concession, the state of readiness of the generation plant is immaterial if the weather is both warm and dry.

What is the simplest way to determine the conditions under which the plant should the plant be ready?

The map:



We see that, taking the "Don't Care" terms into account, the conditions under which the generation plant must be ready to run is, quite simply, when the weather is foggy!

1.11 Implementing Arithmetic

We shall now consider how we may design combinatorial logic circuits for implementing arithmetical operations on (binary representations of) signed and unsigned integers. First, we consider the technicalities of binary arithmetic in a little more detail. We observed that the results of arithmetic, like addition, with signed integers could lead to overflow. For example, in 6 bits, 2's complement:

$$\begin{array}{c c} 010000 & 16 \\ 011010 & 26 \\ \hline 101010 & -22 \end{array}$$

Adding two positive integers, as in this example, leads to an apparently negative result.

We need to be able to detect this kind of *overflow*.

Overflow can be detected if, within the ALU we add an additional, hidden bit, the *guard bit*, to the most significant end of the integer representations. This lengthens the internal representation by extending their most significant bits to the left. Arithmetic is now performed internally, not on n-bit operands, but on $n+1$ bit operands. Using the same example:

$$\begin{array}{r|l} 0 & 010000 \\ 0 & 011010 \\ \hline 0 & 101010 \end{array}$$

Note that the hidden bit (0) in the result is different from the most significant bit (1) of the 6-bit signed integer representation. This indicates *arithmetic overflow.*

There is often a condition code flag, the *overflow flag*, that is set when the guard bit disagrees with the top bit. So by testing the this flag we can tell if there was an overflow.

Some examples of addition with the guard bit, using 6-bit 2's complement representations:

$$\begin{array}{cc|c} 0 & 010111 & 23 \\ 0 & 001000 & 8 \\ \hline 0 & 011111 & 31 \end{array} \qquad \begin{array}{cc|c} 0 & 011011 & 27 \\ 0 & 010000 & 16 \\ \hline 0 & 101011 & ov \end{array}$$

$$\begin{array}{cc|c} 0 & 011011 & 27 \\ 1 & 100001 & -31 \\ \hline 1 & 111100 & -4 \end{array} \qquad \begin{array}{cc|c} 0 & 010001 & 17 \\ 1 & 100000 & -32 \\ \hline 1 & 110001 & -15 \end{array}$$

$$\begin{array}{cc|c} 1 & 101111 & -17 \\ 1 & 101111 & -17 \\ \hline 1 & 011110 & ov \end{array} \qquad \begin{array}{cc|c} 1 & 101111 & -17 \\ 1 & 111101 & -3 \\ \hline 1 & 101100 & -20 \end{array}$$

1.12 Multiplication of Positive Integers

We can multiply by adding repeatedly. Thus binary 000110 × 000011 (decimal 6 × 3) is just:

$$\begin{array}{cc} 000110 & 6 \\ 000110 & 6 \\ 000110 & 6 \\ \hline 010010 & 18 \end{array}$$

However, multiplication can also be performed by the familiar tabular method. Consider the decimal example 324 × 216:

$$\begin{array}{r} 324 \\ 216 \\ \hline 1944 \\ 3240 \\ 64800 \\ \hline 69984 \end{array}$$

In effect we can are doing this:

$$(324 \times 200) + (324 \times 10) + (324 \times 6)$$

namely

$$(324 \times 2 \times 10^2) + (324 \times 1 \times 10^1) + (324 \times 6 \times 10^0)$$

So we are doing multiplications by a single digit, then multiplying by powers of 10.

And we can do the same in binary, by noticing that

- multiplying by a single digit is easy as the only digits are 0 and 1
- multiplying by integral powers of 2 is simply a case of *shifting a bit pattern to the left.*

Example.

$$\begin{array}{cc}
14 \times 1 & 1110 \times 1 = 1110 \\
14 \times 2 & 1110 \times 10 = 11100 \\
14 \times 4 & 1110 \times 100 = 111000 \\
14 \times 8 & 1110 \times 1000 = 1110000 \\
14 \times 16 & 1110 \times 10000 = 11100000 \\
14 \times 32 & 1110 \times 100000 = 111000000
\end{array}$$

To multiply by 2^n we shift left by n bits.

This means the tabular method is extra-easy: 010111×001011 (decimal 23×11) becomes

$$\begin{array}{ccrr}
010111 & \times & 1000 & 10111000 \\
010111 & \times & 10 & 101110 \\
010111 & \times & 1 & 10111 \\
\cline{4-4}
 & & & 11111101
\end{array}$$

For each 0 we write down nothing; for each 1 we write down the number shifted left by a suitable number of places; then we add. Binary multiplication is no more than a few shifts and a few adds!

Notice that, in general, if we multiply 2 n-bit integers, we could produce a result of up to $2n$ bits in length. In this case, 11111101 is equivalent to decimal 253, which is the correct result.

1.13 Summary of Shift Operations

Shifts can move bit patterns both left and right, by one or more places. They also come in various forms, including:

- Logical shifts: bits moved out are lost and zeroes fill the vacated positions.

  A left shift

  $$11100101 \xrightarrow{\text{left } 3} 00101000$$

  A right shift

  $$11100101 \xrightarrow{\text{right } 3} 00011100$$

- Arithmetic right shifts: are like logical right shifts, except that *copies of the sign bit are copied into the vacated bit positions on the left.* A right shift

  $$01100101 \xrightarrow{\text{right } 3} 00001100$$

  The 0 top bit is copied. Another

  $$11100101 \xrightarrow{\text{right } 3} 11111100$$

  The 1 top bit is copied.

An arithmetic left shift is the same as a logical left shift.

More interesting is the effect these shifts have when we regard the bits as integers:

- a logical left shift is a multiplication of an *unsigned* value by a power of 2

- a logical right shift is a division of an *unsigned* value by a power of 2
- an arithmetic left shift is the same as a logical left shift
- an arithmetic right shift is a division of an *signed 2s complement* value by a power of 2

Divisions are *integer divisions*, namely the quotient discarding the remainder. Multiplications are subject to overflow very often.

Examples.

$$\begin{array}{lll} 00000101 \xrightarrow{\text{left } 3} 00101000 & 5 \times 2^3 = & 40 \\ 11100101 \xrightarrow{\text{right } 3} 00011100 & 229 \div 2^3 = & 28 \text{ (rem 5)} \\ 11100101 \xrightarrow{\text{right } 3} 11111100 & -27 \div 2^3 = & -4 \text{ (rem 5)} \end{array}$$

There is also

- *Circular shifts* or *rotates*: bits moved out of one end of a register are moved in at the opposite end of the register.

but these are not related to arithmetical operations.

A left rotate

$$11100101 \xrightarrow{\text{left } 3} 00101111$$

A right rotate

$$11100101 \xrightarrow{\text{right } 3} 10111100$$

Exercise: Show how to construct a rotate from a pair of logical shifts and an OR.

In summary: to do arithmetic we need (at least) shifts and (unsigned) add operations. Let us turn to the add operations.

1.14 Addition in Decimal and Binary

Decimal addition: sum digits from pairs of addends are obtained as follows (c is a carry):

Digits	0	1	2	...	7	8	9
0	0	1	2	...	7	8	9
1	1	2	3	...	8	9	0c
2	2	3	4	...	9	0c	1c
3	3	4	5	...	0c	1c	2c
⋮				⋮			
7	7	8	9	...	4c	5c	6c
8	8	9	0c	...	5c	6c	7c
9	9	0c	1c	...	6c	7c	8c

lsls

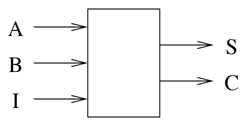


Figure 2: Half Adder

1.15 Binary Addition

The sum digit table for binary addition is very simple indeed, as we would expect from a system with only two digits (again, c is a carry):

Digits	0	1
0	0	1
1	1	0c

In fact, both the decimal and binary tables of the previous slides are really twice as big, because when addition is performed, there are not simply two addend digits. There may also be a *carry-in* digit. For example, the binary table is thus:

Carry-in	Digits	0	1
0	0	0	1
	1	1	0c
1	0	1	0c
	1	0c	1c

Our addition is actually a *three* input operation: two summands and a carry from the previous column. It also has *two* outputs: the sum and a carry to the next column.

1.16 Addition and Logic

We are now in a position to see the connection between arithmetic and logic.

At any general stage in the process of adding two binary numbers, there will be:

- Three inputs: the two addend digits, A and B and the carry-in, I; and
- Two outputs: the sum digit, S, and the carry out, C.

When the two least significant digits of the two binary numbers are added, the initial carry-in $I = 0$.

The full table is:

I	A	B	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Compare the addition table and the XOR table (ignore the carry for now)

+	0	1
0	0	1
1	1	0

XOR	0	1
0	0	1
1	1	0

Thus, s, the sum ignoring carry is

$$s = A \text{ XOR } B$$

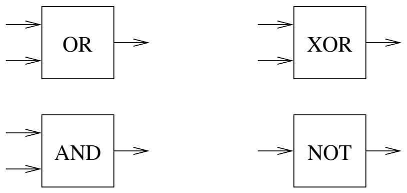
Similarly, look at the carry and the AND tables:

c	0	1
0	0	0
1	0	1

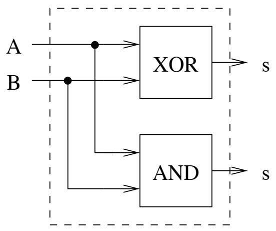
AND	0	1
0	0	0
1	0	1

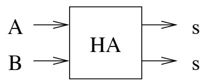
Therefore, c, the carry ignoring the previous carry is

$$c = A.B$$



We can therefore draw the logic gate diagram for the *partial sum* and *carry* digits (we have yet to deal with the carry-in, I) as follows:





lsls

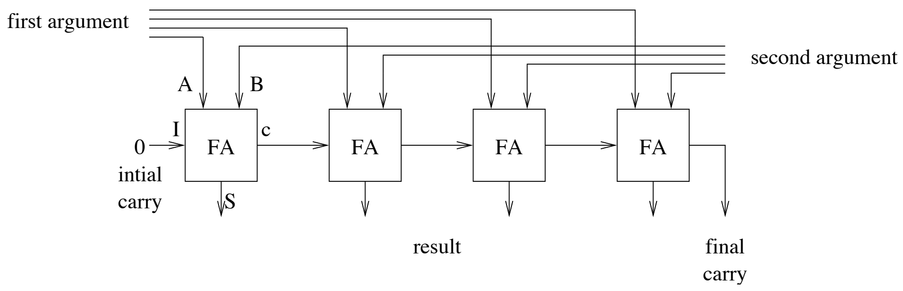


Figure 3: Addition using Full Adders

This is a *half adder*, as it is about half of what we need to make a full adder.

We now need to deal with the carry-in, I.

If the partial sum digit, s (i.e. the half-adder sum) and the carry-in, I, differ, then the final sum digit S will be 1. Otherwise, it will be 0. That is, it is just the sum of I and s, or $S = I$ XOR s.

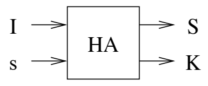
The carry-out, C, will be 1 if either:

1. the partial carry digit, c (i.e. the half-adder carry) is 1, or
2. both the partial sum digit, s and the carry-in digit, I, are 1:

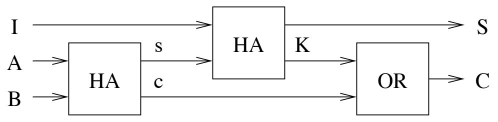
$$K = I.s$$

These conditions (1 and 2) cannot in reality occur together.

Notice that to compute S and K, we can use a half adder whose inputs are s (the partial sum digit) and I:



So the whole single stage addition process can be computed using two half adders and an OR gate:



These comprise a *full adder*. Full adders can be strung together to add multi-bit numbers. See Figure 3.

The full adders are used *serially*, as each must wait for the previous to provide its result before it can compute its own result. The the bits of the answer are produced one at a time.

A larger example. Add $011100 + 010110$ $(28 + 22)$. Read this table right to left, down the columns.

A	0	1	1	1	0	1	
B	0	1	0	1	1	0	
I	1	1	1	0	0	0	previous carry C
s	0	0	1	0	1	0	half sum: A XOR B
c	0	1	0	1	0	0	half carry: $A.B$
K	0	0	1	0	0	0	half carry: $I.s$
S	1	1	0	0	1	0	sum: I XOR S
C	0	1	1	1	0	1	carry: K+C

The result is 110010 (50).

1.17 A Parallel Scheme

If the bit patterns for A and B are available in parallel, addition can be accomplished by logic as follows (A and B now stand for sequences of bits, e.g., a byte or word):

1. while B not all zeros do
   - $C \leftarrow A.B$ (carries)
   - $A \leftarrow A$ XOR B (sums)
   - $B \leftarrow C << 1$ (shift carries up one place)
2. return A.

In this "$A.B$" means the *bitwise* AND of the bits from A and B, namely ANDing each pair of bits in parallel.

In this, C contains the carries of the individual bit sums. We add the bits of b into A; shifting C up by one place puts the carries into the right place to be added in the next loop. When B is all zeros, that is there are no more carries, we are finished.

Example.

A	011100	
B	010110	
C	010100	bitwise AND
A	001010	A XOR B
B	101000	C shifted
C	001000	
A	100010	
B	010000	
C	000000	
A	110010	
B	000000	

Result is 110010, as before.

So, given hardware to do *bitwise parallel* operations on binary words, we can add together much faster than the serial full adder solution.

Exercise: Given n bits, what is the *maximum* number of times we might go around the loop?

1.18 Bitwise Logical Instructions

It will by now come as no surprise to learn that there is a functional class of operations on most computers that performs the logical operations between bit patterns.

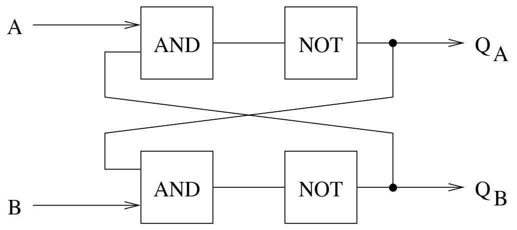
Thus, if $A = 01101010$ and $B = 11010000$

AND will generate the pattern	01000000
INCLUSIVE OR will generate	11111010
EXCLUSIVE OR will generate	10111010
NOT on A will generate	10010101

2 Sequential Logic

The output of a logic gate in a *combinatorial* logic system may form the input to a gate "further down" the system. However, in a *sequential* system, the output of a logic gate can also be fed back so as to become the input to a gate "earlier" in the system.

The simple (inverted SR, or $\overline{SR}$) latch is an example:



Here we have the simultaneous equations

$$\begin{aligned} Q_A &= (A.Q_B)' \\ Q_B &= (B.Q_A)' \end{aligned}$$

This has solutions

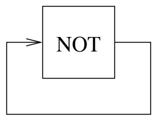
A	B	Q_A	Q_B
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	1	1	0

But these solutions are not particularly useful in a dynamic system.

It is quite difficult to work out what happens in a system like this one. Let us make some helpful assumptions (which are valid for real implementations of such a system), namely:

- That there is a delay before a logic gate responds to an input;
- Separate logic gates cannot switch output simultaneously;
- When the power to the gates is switched on, we have no knowledge about the states of the outputs from the logic gates.
- We can therefore think in terms of the output one moment after the input, for all possible starting states of the system.

The requirement for a delay between input and output is essential if circuits are to make physical sense. Otherwise what does



do? The output would have to be simultaneously 0 and 1. With a small delay, the circuit could oscillate between the two values, which is at least physically possible.

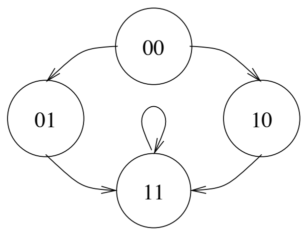
Let us assume that A and B are both set to zero. By chasing the values around the circuit we find:

Old Outputs		Inputs		New Outputs	
Q_A	Q_B	A	B	Q_A^{new}	Q_B^{new}
0	0	0	0	1	1
0	1	0	0	1	1
1	0	0	0	1	1
1	1	0	0	1	1

Thus, if $A = B = 0$ and we start with $Q_A = Q_B = 0$, we have, a moment later, $Q_A = Q_B = 1$.

2.1 State Transition Diagrams

We can also represent this table as a *state transition diagram.* The possible outputs of the system, Q_A and Q_B, are represented in each circle and the arrowed arcs show the changes from an old output to a new one. Here is the state transition diagram when A and B are both 0:



Point of note:

- state 00 does not go directly to state 11 as the table might suggest. Slight asymmetries in the system means the state will either go through 01 or 10 first.
- we can't tell in advance which way it will jump: this is a *non-deterministic* change of state
- the transitions from 01 and 10 are *deterministic*, as we *do* know what will happen.

Old Outputs		Inputs		New Outputs	
Q_A	Q_B	A	B	Q_A^{new}	Q_B^{new}
0	0	0	1	1	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	0	1	1	0
0	0	1	0	1	1
0	1	1	0	0	1
1	0	1	0	1	1
1	1	1	0	0	1
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	1	1	0
1	1	1	1	0	0

lsls

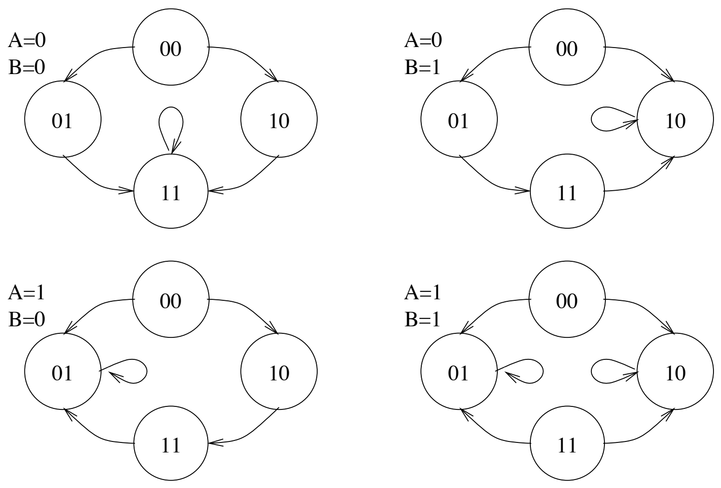


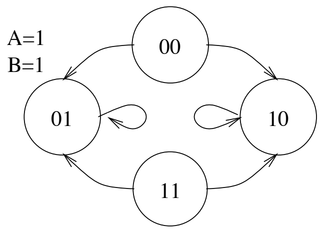
Figure 4: State Transitions of the Latch

The full set of state transition diagrams is in figure 4.

The state transition diagrams for all possible input combinations of A and B can be summarised by specifying the new state of Q_A (that is, Q_A^{new}).

A	B	Q_A^{new}
0	0	1
0	1	1
1	0	0
1	1	Q_A

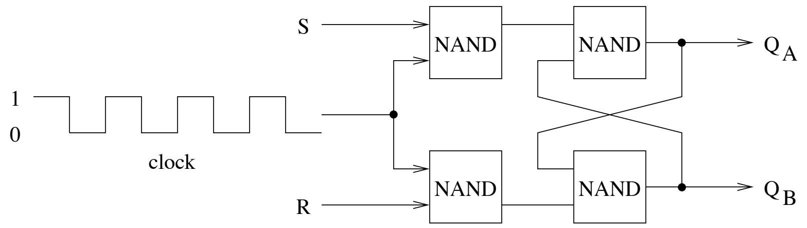
The last line needs some explanation.

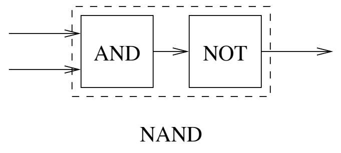


We saw from the state transition diagrams that Q_A and Q_B will settle in different states, unless $A = B = 0$. If A and B both started at 0, they would not both become 1 simultaneously, but would pass through $A = 1$ and $B = 0$ or $A = 0$ and $B = 1$ on the way. Even in this case, therefore, Q_A and Q_B would settle into different states. Thus, when $A = B = 1$, Q_A and Q_B will always stay as they were.

2.2 The SR Flip-Flop

The simple latch needs extending to become a useful logic system. It needs a *clock* signal so that the latch responds to inputs only when the clock signal is present, rather than at any time (to simplify the diagram, a NAND gate is shown. It is simply an AND gate followed by a NOT gate):





Truth table for NAND:

A	B	AND	NAND
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

The inputs are traditionally called R and S, and the while thing an *SR flip-flop*, or and *SP gated latch.*

The purpose of the clock signal is this: when the signals R and S are changing the circuit can act in unpredictable ways (witness the non-deterministic transitions). Much better is to put the system into a *known* state, change the inputs, and only then look at the outputs.

This is what the clock does. When the clock is 0, the latch is held in a stable state. We can play with R and S as much as we like, as long as they have settled when the clock changes to 1. When the clock is 1, we then read off the new state. So

- clock 0 change inputs
- clock 1: read outputs

This minimises the problems of unstable transitions.

We can do exactly the same kind of analysis on this system as we did on the simple latch. In fact only four *distinguishable* state transition diagrams result:

Inputs			Diagram
S	R	C	type
0	0	0	a
0	0	1	a
0	1	0	a
0	1	1	b
1	0	0	a
1	0	1	c
1	1	0	a
1	1	1	d

When there is a clock signal only, or no clock signal, the system settles into one of the two stable states 01 and 10 (it is a *flip-flop* or *bistable* system).

Diagram (d) is important. It shows that if S and R are both set and the clock pulse is then removed, the system will transition to one of two possible steady states (01 or 10) and we cannot know which: it will be a non-deterministic transition.

lsls

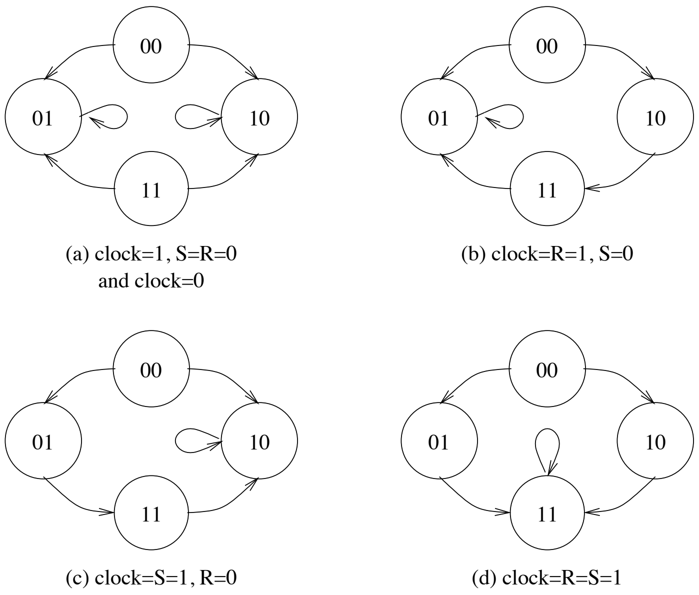


Figure 5: State Diagrams for SR Latch

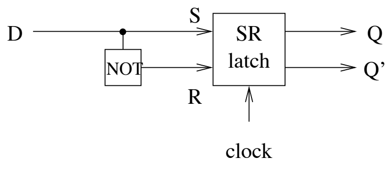
2.3 Summary of the Behaviour Of the SR Flip-Flop

The table below summarises our findings by showing the new output state (Q_A^{new}) of the flip-flop, given the S and R inputs (C is not included, as the flip-flop can only change while C is set):

S	R	Output (Q_A^{new})
0	0	Q_A
0	1	0
1	0	1
1	1	?

2.4 D-type Flip-Flop

One way to overcome the defect in the SR flip-flop (that is, the uncertainty of the state resulting from dropping the clock signal when R and S are both 1) is to ensure that the two inputs are always different. This is ensured in the *D-type flip-flop*:



The table showing the behaviour of the D-type flip-flop is thus very simple:

D	Q^{new}
0	0
1	1

This is equivalent, in effect, to an SR flip-flop which is allowed only the two input states:

$$\begin{array}{lll} S = 0 & \text{and} & R = 1 \\ S = 1 & \text{and} & R = 0 \end{array}$$

What is the point of the D-type flip-flop? It is a *one clock cycle delay.* That is, the Q^{new}, which appears as output one cycle after D was input is equal to that D.

2.5 JK Flip-Flop

This type of flip-flop makes positive use of the fourth input combination, the one that is deficient in the SR flip-flop. It is used to *toggle*, or change the output state, of the flip-flop. The table describing the behaviour of the JK flip-flop is thus as follows:

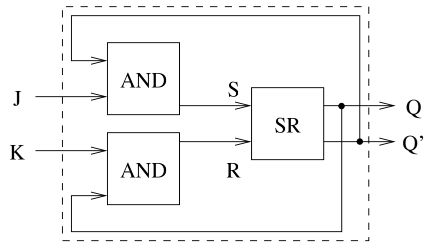
J	K	Q^{new}
0	0	Q
0	1	0
1	0	1
1	1	Q'

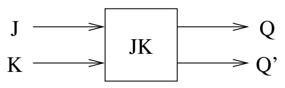
From the table showing JK flip-flop behaviour we can see, for example, that if the old JK output, Q, was 0 and the new output is to be 1, then we can achieve this by setting $J = 1$ and $K = 0$, or $J = 1$ and $K = 1$. The table for all output changes is:

Q^{old}	Q^{new}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

X's are "don't care". So $J, K = 0, \text{X}$ stands for both $J, K = 0, 0$ and $J, K = 0, 1$.

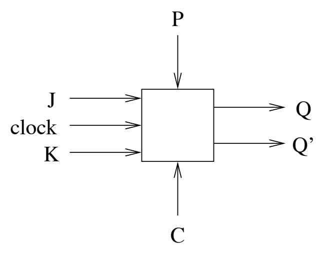
The *unclocked JK flip-flop* (or latch) can be constructed with an SR flip-flop plus two AND gates (the analysis is not given here):





The more common type of J-K flip-flop is the *clocked J-K flip-flop* which changes in response to the *lowering* of the clock signal.

As well as having a clock signal input, the flip-flop may also be shown with two additional inputs, namely, *unclocked set* (P) and *clear* (C) inputs:



2.6 Traffic Light Controller

A traffic light cycles through four states repeatedly, namely: red; red and amber; green; and amber. Let is represent these four states by two Boolean variables, X and Y:

State	X	Y
Red	0	0
Red & amber	0	1
Green	1	0
Amber	1	1

We can now show the successor states for each state in this cycle:

Starting		Successor	
X	Y	X	Y
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

We shall use one clocked JK flip-flop to generate X and another to generate Y.

Here is the control table we need:

Old		New		JK controls			
X	Y	X	Y	J_X	K_X	J_Y	K_Y
0	0	0	1	0	X	1	X
0	1	1	0	1	X	X	1
1	0	1	1	X	0	1	X
1	1	0	0	X	1	X	1

As before, X denotes a "don't care" term.

We can now draw Karnaugh maps of the JK controls and simplify them in the usual way.

The J-K controls are thus:

$$J_X = K_X = Y$$
$$J_Y = K_Y = 1$$

The control system is therefore:

lsls

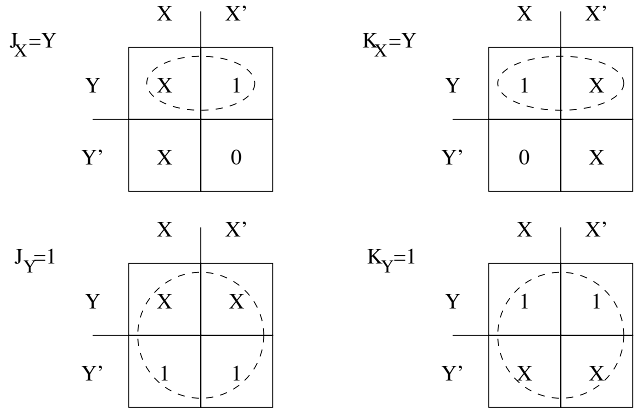
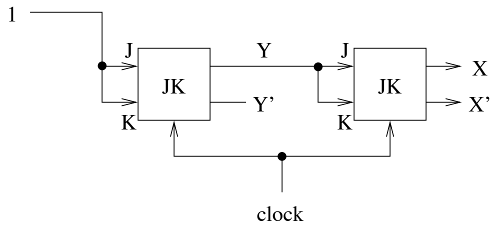


Figure 6: Karnaugh Maps for Traffic Lights



We now need to connect the outputs X and Y to the lights.

X	Y	Red
0	0	1
0	1	1
1	0	0
1	1	0

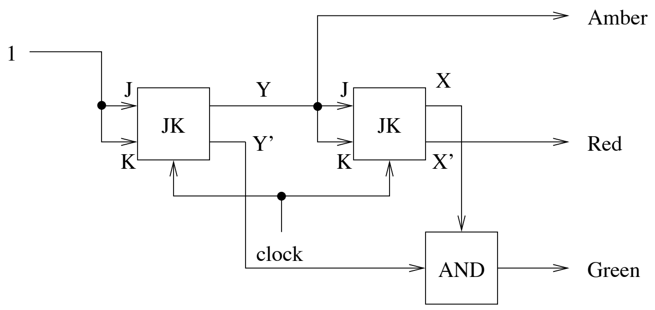
X	Y	Amber
0	0	0
0	1	1
1	0	0
1	1	1

X	Y	Green
0	0	0
0	1	0
1	0	1
1	1	0

Thus

$$\begin{aligned} R &= X'.Y' + X'.Y = X' \\ A &= X'.Y + X.Y = Y \\ G &= X.Y' \end{aligned}$$

So the final control system is



As the clock ticks, so the lights cycle though the expected sequence.