Week 4: Logic Applications and Introduction to Sequential Logic

4.1 Binary Arithmetic

4.1.0 How can a circuit add up?

The logical circuits we have seen so far have just a single output with many inputs. If we wanted a circuit that would add up, the circuit must

- Receive pattern of bits at one set of inputs which represents one number.
- Receive pattern of bits at a second set of inputs which represents the second number.
- Implement logic that produces the correct outputs for these numbers at a set of outputs.

Consider the addition of just two 1-digit binary numbers. The circuit is called a *binary adder* and its truth table is fairly easy to write out…

A	*B*	*A* plus B	
0	0	0	
0	1	1	
1	0	1	
1	1	0	plus 1 to carry.

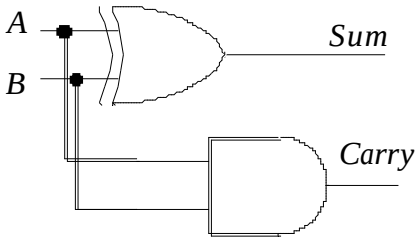
Notice that the circuit needs two outputs: a “sum” output and a “carry”. So we can write out the specification for each of the outputs in a truth table.

SUM

A	*B*	*Sum*
0	0	0
0	1	1
1	0	1
1	1	0

CARRY

A	*B*	*Carry*
0	0	0
0	1	0
1	0	0
1	1	1



To make a circuit which adds these numbers we need merely to make logic which implements these truth tables, and we can add. You will hopefully recognise these truth tables. The *Carry* table is simply an AND gate and *Sum* is an XOR gate. So a logical circuit which adds is shown right. There are many other variations of logic which will produce the same effect.

4.1.1 The Half Adder and the Full Adder

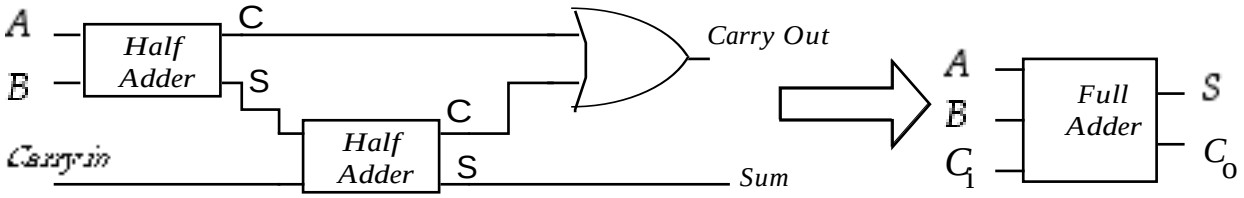
The circuit shown above is known as a *half adder*. Why? Because when adding two binary digits we need to consider the possibility that there would be a “carry digit” from a calculation on the less significant digits. A circuit taking account of this possibility now must have three inputs and two outputs. The truth table required for a full adder is shown below:

A	*B*	*Carry in*	*Sum*	*Carry out*
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

We could now proceed to work out the logic for these functions using a *K*-map as we have done previously (although we would find that no minimisation is possible).

Exercise 4.1 Work out and minimise the logic for a full adder using a *K*-map

However we can see that a full adder can be made from half adders. A circuit showing this is sketched below.



Notice how we are using a box to represent a previously designed circuit element. This is helpful because there are different ways of implementing the logic of a full adder and often we are not interested in which implementation has been chosen.

4.1.2 Bits and Bytes: Adding bigger numbers

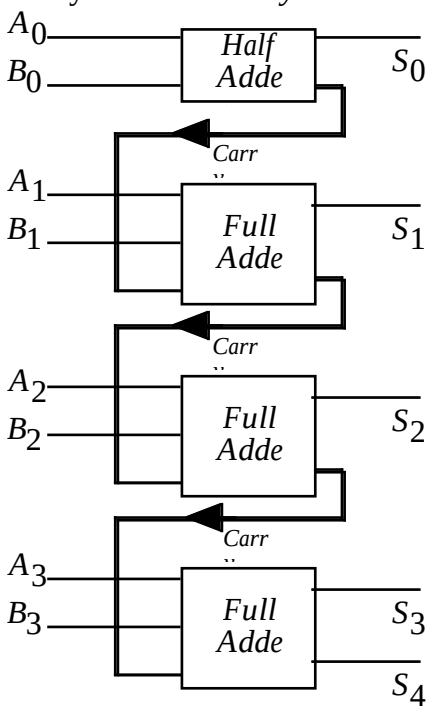
Some terminology: When logic levels are used to represent binary numbers, each input or output line represents a single binary digit. This is generally abbreviated to *bit* (**b**inary dig**it**). A collection of (usually 8) bits forming a number is generally known as a *byte*.

Suppose we want to add two numbers, say $A = 7$ and $B = 5$. How could we make a circuit which would add these numbers? We would need to add together binary representations of these numbers bit-by-bit, starting with the least significant bit and working along to the most significant bit, carrying over bits as we went. i.e.

A	0	1	1	1
B	0**(1)**	1**(1)**	0**(1)**	1
A plus B	1	1	0	0

(the number in brackets represented carried over bits)

Notice that the adding for the first bit requires only a half adder because there is no bit carried over from a previous calculation. The circuit for a full multi-bit adder is shown right. Notice that it is possible that the sum of two 4 bit numbers may need 5 bits to be represented fully (e.g. 1111 + 1111 = 11110)



Exercise 4.2 Verify that the multi-bit adder shown will indeed successfully add 7 and 5. Use the truth table of the full adder to aid you.

4.1.3 Two's complement encoding: subtraction

Numbers may be subtracted in a similar way to the way in which they are added. Again, there are many possible strategies. We could of course write out truth tables, deduce canonical forms and then minimise functions and implement a subtractor circuit. However we will not do this. There exists a simple way to subtract two binary numbers using only adders. It is not however the most trivial thing to explain

First of all notice that in order to represent negative numbers or subtraction (they amount to the same thing) we need somehow to mark out a number as negative. There are several possible ways to do this. Let's consider 8-bit bytes. Normally we use such bytes to represent numbers between 0 (00000000) and 255 (11111111). If we wish to represent negative numbers then we need to use one of the bits to represent the sign, which leaves only 7 bits left to represent the magnitude of the number. Perhaps the simplest code for representing negative numbers would be to use the most significant bit (MSB) for the sign. In this case then 00000001 would represent 1 and 10000001 would represent –1. The range of number that could be represented in this way would be from $-(2^7-1)$ through 0 to $+(2^7-1)$, i.e. –127 to +127. However with that scheme in order to subtract numbers (or "add a negative number") special logic would be required.

Here is how it is in fact done. Suppose we wish to subtract *B* from *A* (where both are assumed to be positive). First the number to be subtracted (*B*) is converted into a code known as the *two's complement* of the original number. This is our representation of negative numbers. Then this coded version of the negative number is added to *A* using normal adding circuitry and the answer produced. If the answer is negative, one again takes its two's complement to translate it into normally readable form (i.e. a positive number with assumed minus sign).

The two's complement of a number is formed as follows.

- First we use a representation where the MSB is reserved as a sign bit, i.e. 0 represents positive and 1 represents negative. Thus for an 8-bit byte using two's complement arithmetic, the maximum positive number that may be represented is + 127 (01111111).
- Now we take the 1's complement of the magnitude of the number (i.e. turn all 1's to zero and *vice versa*)
- To form the 2's complement of the number add 1 to the 1's complement

Example

What is the two's complement representation of (a) 27 and (b) –27?

The number +27 is represented in two's complement arithmetic as **0**0011011. Notice that the initial zero shown in bold does not represent 0×2^7; it represents the sign of the number.

The number –27 is formed from the two's complement representation of +27, **0**0011011.
First we form the 1's complement: **1**1100100. Then we add 1, i.e. **1**1100100 + 1 = **1**1100101. Notice that the sign bit is "set" (≡1) indicating a negative number, but the seven remaining bits cannot be interpreted in the normal way. So the two's complement representation of –27 is **1**1100101

Exercise 4.3

Perform the following sums using two's complement arithmetic, showing all the steps in your answers, and verify that the technique works correctly.

(a) 80 – 5
(b) 5 – 80
(c) 113 – 113.

Now show that –(–5) = +5 in 2's complement arithmetic.

4.2 Coding

We have already seen how binary codes can be used to represent (or "code") states in the real world (e.g. A = 1 means "the door is open") and how binary codes can also be used to represent numbers (e.g. 1001 represent $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^2 + 1 \times 2^0 = 8 + 1 = 9$). We have also seen that there are different ways of representing binary numbers when we want to have the possibility of using negative numbers. There are yet more possibilities

4.2.1 BCD: Binary Coded Decimal

Suppose we want to encode a number in such a way that it may be easily converted back into a decimal form for display. To do this we can use the binary coded decimal system which allocates the "natural codes" to the decimal digits 0 to 9, but discards the other codes, i.e. they are "can't happen" states. Then to represent a number such as 217 we can use BCD code to encode each digit of the decimal number separately. i.e.

$$217 \Rightarrow 2\ 1\ 7 \Rightarrow 0010\ 0001\ 0111$$

Notice that this is not quite as efficient in use of bits as raw binary code which would yield

$217 \Rightarrow 1\times128 + 1\times64 + 0\times32 + 1\times16 + 1\times8 + 0\times4 + 0\times2 + 1\times1$
$217 \Rightarrow 1\times2^7 + 1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^2 + 1\times2^0$
$217 \Rightarrow 11011001$

This uses only 8 bits to carry the same information as the 12 bits of the BCD code. However for many purposes the BCD code is more convenient (but not for doing arithmetic!).

4.2.2 Gray Codes

Gray codes are yet another way of representing binary numbers. They are just as efficient as the natural binary codes but avoid the possibility of transient errors. In this sense the use of a Gray code is for the same purpose as the special logic we employed to avoid static hazards. The key feature of a Gray code is that only a single bit changes as the code is incremented. There is no unique Gray code. Below is an example of a 3-bit Gray code.

Exercise 4.4 Construct your own 3-bit Gray Code

MY GRAY CODE

N	*A*	*B*	*C*
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	0	0
6	1	0	1
7	1	1	1

YOUR GRAY CODE

N	*A*	*B*	*C*
0			
1			
2			
3			
4			
5			
6			
7			

Can you see how the use of a Karnaugh map helps in deriving a Gray code?

Gray Codes are most commonly encountered on shaft encoders, devices that produce a binary coded output that changes as a shaft turns within a bearing.

4.2.3 Error Detection codes

Error detection code is crucial when binary digits are sent from one circuit (chip, device, computer, country, planet...) to another. How can one tell whether an error has occurred during transmission? To give some idea of how critical this is, a CD contains around 640 MB of "information", i.e. $\approx 640 \times 1048576$ bytes $\times 8 = 5.3 \times 10^9$ bits. While it might be acceptable that the CD be read inaccurately when the information encodes music, it is not acceptable that there be an error of even a single bit if the CD contains, say, computer software. CD error detection is extremely clever and sophisticated using a so-called *Reed Soloman* code. However we can see how the idea works if we consider sending a byte of information and then sending an additional bit of information which depends on the data just sent. The simplest code is produced as follows.

Let the 8 bits of information to be sent be A_0 , A_1, A_2, etc. First A_0 and A_1 are fed into a device with the following truth table

A_0	A_1	*output*
0	0	0
0	1	1
1	0	1
1	1	0

You may recognise this as an XOR function. Then the output of the XOR operation is XORed with the next bit, A_2, and so on until the end of the byte when the result is sent as a final additional "check" bit. This operation is determining whether there have so far been an even number of 1's in the data (*output* = 0) or an odd number of 1's (*output* = 1) The final output is called an *odd parity check bit*.

At the receiving end the same operation can be performed on the incoming bits and if the "check bit" is found to be the same then one can be sure that only an even number of errors have occurred (i.e. 0 , 2, 4…) – taking "error" to mean that the bit is received, not missed, but with the wrong value. If the communications are reasonably reliable then one can assume that 2 errors are unlikely and so one can detect whether an error has occurred. (However the error cannot be corrected – it would be necessary to request that the data be sent again.)

4.2.4 Encoding and Decoding

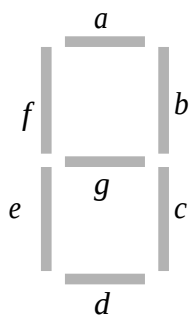
Frequently one would like to encode and decode numbers into some other form, usually non-binary data encoded into a binary number or vice versa. Suppose for example that on receipt of a 3-bit binary number we would like a circuit to select one of 8 different output lines.

Since we now have 8 output lines we need to use 8 different truth tables, 8 different *K*-maps and minimise 8 different logic functions, one for each of the output lines. Similar *decoders* can be made to select 1 of 10 different outputs on receipt of a BCD coded decimal digit (a "4 to 10" decoder)

Encoders will similarly produce a numerical binary code depending which of (say) 10 different input lines have been selected.

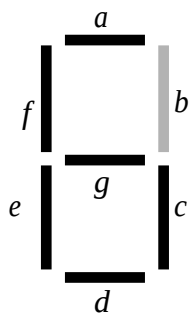
4.2.5 BCD to 7-Segment display

A seven segment display is one of the more primitive but still very common forms of alphanumeric display. Each of the segments lights when instructed to by decoding circuitry. For example when the code for the number "6" arrives, the decoder would select elements *a, f, e, d, c* and *g* for illumination as shown below right.



Exercise 4.5. What is the required decoding logic for each element of a 7 segment display subject to a BCD input. This is a fairly tedious (but important) problem that you can solve in the normal way:
truth table ⇒ *K*-map ⇒ minimised logic.

The solution for element *a* is shown in *Beard* on page 368-369



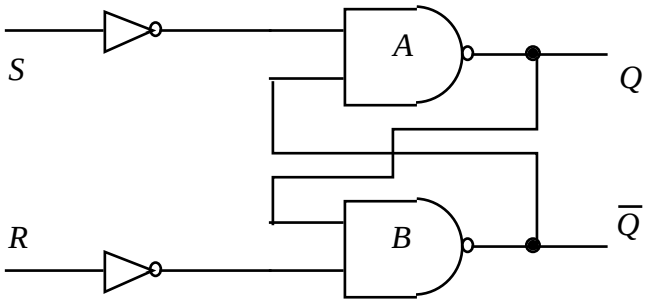
Sequential Logic

So far we have been studying combinatorial logic that enables us to perform such tasks as adding or subtracting two numbers, encoding, etc. Now we look at *sequential* logic circuits, i.e. circuits with an output that depends not only on the current state of its inputs but also on the previous states.

4.3 Latches and Flip Flops

4.3.1 The SR Latch

Consider the circuit shown right and its behaviour when the inputs S and R take on values 00, 01, 10, 11. Since the output values Q and $\overline{Q}$ are fed back to the inputs of NAND gates A and B we need also to know the states Q and $\overline{Q}$ in order to predict what the values Q and $\overline{Q}$ will be! In fact the analysis is not as complicated as it might seem. Recall that the output of a NAND gate is 1 unless all the inputs are 1 in which case the output is zero. Let's analyse the circuit assuming $Q = 0$ and then look again at what happens when $Q = 1$.



Assume $Q = 0$ (this means that $\overline{Q}$ must be 1, hence we must check $Q = \text{NOT}\,\overline{S} = S$):

S	R	$\overline{S}$	$\overline{R}$	$\overline{Q}$	$Q?$
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	?
1	1	0	0	1	?

The last two rows are inconsistent with our initial assumption.

Assume $Q = 1$ (this means $\overline{Q} = \text{NOT}\,\overline{R} = R$):

S	R	$\overline{S}$	$\overline{R}$	$\overline{Q}$	$Q?$
0	0	1	1	0	1
0	1	1	0	1	?
1	0	0	1	0	1
1	1	0	0	1	1

The second row is inconsistent with our initial assumption.

We see that certain states are inconsistent with our assumptions and they are marked with a "?". Notice that making SR = 10 **S**ets the latch to $Q = 1$ (so inconsistent with Q=0 initial assumption) and making SR = 01 **R**esets the latch to $Q = 0$ (inconsistent with Q=1 initial assumption). The latch has stored 1 bit of information because the output state doesn't change if the inputs revert to 00, whether from 01 or from 10. However some aspects of this circuit are unsatisfactory from a practical point of view.

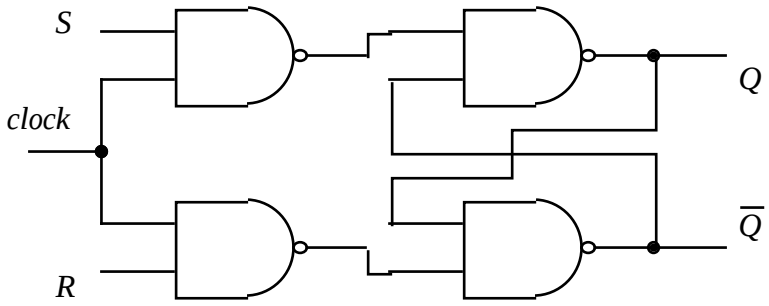
Notice that the two outputs Q and $\overline{Q}$ are appropriately labelled as complementary, unless SR =11 occurs. SR=11 is a forbidden state since if the input condition 00 follows a 11 condition then the desired latching property of this circuit is lost. We do not know what state the circuit will adopt next (it will depend on whether S or R reaches zero first).

Exercise 4.6 Implement an *SR* latch in hardware and in software and verify that it performs as predicted. Notice its latching property.

4.3.2 The SR Flip Flop

An SR flip flop is a development of an SR latch. The key change is the use of a so-called "clock" input to *gate* the inputs to the SR latch. The SR inputs are only enabled (made effective) when Clock takes the value 1. In normal use the clock input is 0 unless we wish the outputs to be updated, in which case Clock is momentarily set to 1 and then back to 0: a "clock pulse".

Note. This idea of "enabling" an input is an important concept in digital electronics. It can be implement using an AND or a NAND gate. E.g. for an AND gate if one input is held at 0 then no matter what the value of the other input the output will not change from being 0. However when an "enabling" input has the value 1 then the other input affects the output: an input of 1 produces an output of 1 and an input of 0 produces an output of 0. We can think of that input being allowed through to the output. If we use a NAND gate then one input enables the other input to pass through but complemented.



Once again the circuit output when SR=11 during the clock does not give reliable results and must be avoided, even transiently. The truth table is summarised below

S	R	Q_{n+1}	$\overline{Q}_{n+1}$
0	0	Q_n	$\overline{Q}_n$
0	1	0	1
1	0	1	0
1	1	?	?

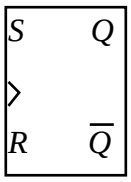
There is a new notation used here that is appropriate to clocked circuits. The subscripts $_n$ and $_{n+1}$ signify "the current value of an output" and "the subsequent value after the next clock pulse" respectively.

The key points are that

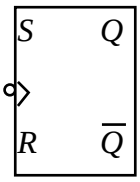
- S and R still set and reset Q and with SR =00 Q (whatever its state) is "memorised" by the circuit, i.e. $Q_{n+1} = Q_n$, and that
- S and R only affect Q when they are enabled by a clock pulse going $0 \rightarrow 1 \rightarrow 0$. The flip flop captures the state of SR at the instant the clock goes back to 0, like taking a photograph.

Depending on the particular circuitry at the clock input, the "enabled" state may correspond to a 1 (as above) or to a 0 at the Clock input (the terms "positive edge triggered" and "negative edge triggered" used by Beards do not properly apply here – see the JK flip-flop later). The symbols for SR flip flops with normal and complementary clock inputs are shown below where the wedge symbol (>) is used to indicate a clock input (Ck).

Normal clock SR Flip Flop



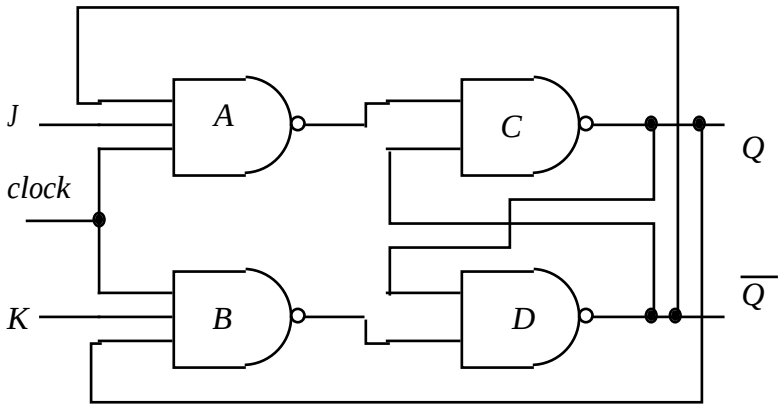
Complementary clock SR Flip Flop



Exercise 4.7 Implement an *SR* flip flop in hardware and in software and verify that it performs as predicted.

4.3.3 The non-existent "JK Flip Flop according to Beards"

The circuit discussed in section 14.3 of Beards (shown below) which he calls the "JK flip flop" does not actually behave as he states and is in fact never used: it would work in exactly the same manner as the ordinary clocked SR flip flop. You can see that J is directly equivalent to S and K is equivalent to R and the extra feedback connections are redundant because $Q=K$ and $\overline{Q}=J$ always. For the state $JK = 11$ both outputs would go to 1 during the clock, just as for the SR flip flop, and so the state following the clock would similarly be unpredictable.



The only kind of useful JK flip flop is the *master slave JK flip flop* described in the next section.

Exercise 4.8 Implement a Beards-type "*JK* flip flop" in software and verify that it does *not* perform as he states, i.e. that it does not reliably toggle for $J=K=1$. (It may oscillate or go to $Q=\overline{Q}=1$)