Chapter 3

Integrated circuits

Aims

The aim of this chapter is to cover the basics of how computer chips function and are built. The story we tell uses a stack of abstractions at different levels, running from the basic physics of semiconductors up through how transistors work and their use as switches to building logic gates and then components such as memory cells and basic adders.

Component
Gate
Transistor
Semiconductors
Quantum physics

Each level of this is built in some way on the next level down. We aim to give a broad overview rather than great detail.

Putting this into the broader context, we will see later that everything in a computer is represented in binary, 1's and 0's. Inside a chip these correspond to high and low voltages, but conceptually they also correspond to the **true** and **false** of **boolean logic**. The computer needs to do calculations with the boolean values. These calculations can be represented as logical operations made up from the standard boolean connectives (**and**, **or**, **not**) and are implemented in the computer through a switching network made using **transistors** as switches.

The aim of this chapter is to show how transistors are made from semiconductors (and why they work), how they can be put together to compute logical operations (gates), and how those gates can be put together to do other computations, such as making a simple one-bit adder. We will also show how to make a simple one-bit memory.

The designs we show are not necessarily the ones used in real integrated circuits. Particularly when we come to the adder and the memory, actual designs are optimised. But they do show that the concept can be made to work.

Learning Objectives

By the end of this chapter you should understand:

- ☐ 1. how transistors are constructed from semiconductors and how they function as switches
- ☐ 2. how to use transistors to build basic logic circuits called gates
- ☐ 3. how to put units of logic together to construct other circuits such as basic adders
- ☐ 4. how circuits with feedback can be used to construct simple memory cells.

3.1 Semiconductors

This section is not examinable.

The physical reason an electric current flows in a conductor is that (negatively charged) electrons are moving in the opposite direction. Metals, such as copper and iron, are conductors.

But the quantum physics underlying this is more complicated, and some substances (like silicon) have some of the properties of metals that would make them conductors, but not all (there are issues with the energy levels of electron bands).

These substances will conduct electricity, but not readily. They are **semiconductors**, neither good conductors nor good resistors, but unlike metals their conductivity increases if they are heated.

The best known of these is silicon. Silicon is an element with atomic number 14, and valency 4. It forms crystals in which each atom is surrounding by four others at the corners of a tetrahedron (a diamond cubic lattice, the same structure as carbon when forming diamonds).

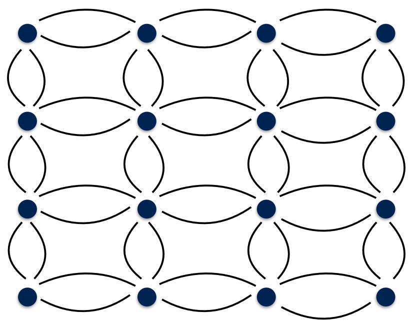
Silicon is very common, it is a basic component of rocks, including sand. Glass is made from Silicon Dioxide.

For use in chips, impurities are added to the silicon. These are called **dopants**, and they fall into two classes: electron **acceptors** or **donors**.

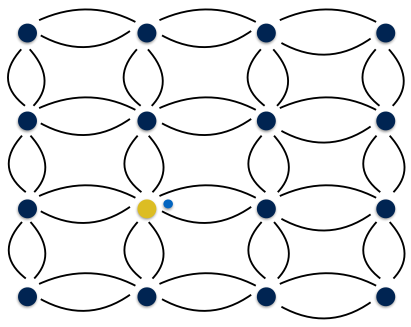
Donors produce **n-type** semiconductors. A commonly used donor is **arsenic**. **Acceptors** produce **p-type** semiconductors. A commonly used acceptor is **boron**.

Silicon crystal structure

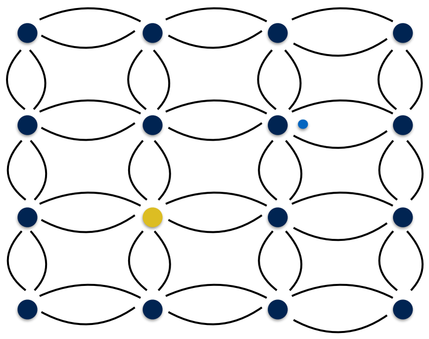
Here is a flattened out picture of the silicon crystal structure (the connectiivity is wrong, but that is not important for our current purposes). The arcs represent standard covalent bonds. In other words they are a pair of electrons shared between the outer shells of the atoms at the two ends.



Doping replaces one of the silicon atoms in the structure with an atom of the dopant (here arsenic). So this gives you a substance with the same crystal structure, but the wrong atoms at some random points. Now arsenic has an extra electron in its outer shell compared with silicon, and so what we end up with is the same basic structure as a pure silicon crystal, but with some extra electrons at various points.

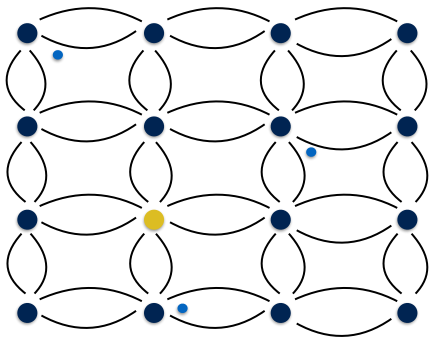


The critical thing is that these additional electrons are not tied to the arsenic atoms, but can move about the crystal structure. This allows the silicon structure to conduct electricity. Remember that metals conduct electricity because electrons are free to move through the metal, so transporting electric charge and thus making an electric current.



Understanding electrical conductivity properly, and understanding the conductivity in semiconductors properly takes some quite serious quantum physics. But the upshot is that the mobile electrons in an n-semiconductor behave as an ideal gas. This is not simply a loose analogy, there is a precise technical meaning to this, in terms of the statistical behaviour of the electrons in terms of their position and motion. As a result we can regard an n-semiconductor as electrically neutral, but holding a negatively charged electron gas at positive pressure.

The story for a p-semiconductor is similar, except that the dopant has fewer electrons in the outer shell than sillcon. This means that there is an electron missing where the crystal structure expects there to be one. This is called an electron **hole**. An electron from a neighbouring atom can be borrowed to fill that hole. If we think in terms of holes, not electrons, that means the hole moves from its original atom to a neighbouring one. As a result, the holes can move about the crystal structure in the same way as electrons, and, like the electrons in an n-semiconductor, they behave as an ideal gas. We can therefore think of the p-semiconductor as filled with a positively charged gas made out of electron holes. Alternatively we can think of it as filled with an electron gas at negative pressure.



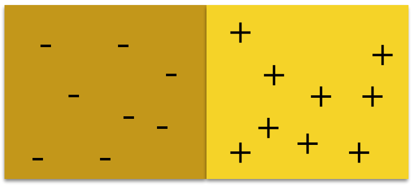
There is a symmetry between n-semiconductors and p-semiconductors.

n-semiconductor	**p-semiconductor**
standard diamond cubic lattice	standard diamond cubic lattice
dopant gives additional electrons	dopant gives additional holes (absences of electrons)
overall electrically neutral	overall electrically neutral
conducts electricity through movement of additional electrons round crystal	conducts electricity through movement of holes round crystal
electrons behave as ideal gas	holes behave as ideal gas
filled with electron gas at positive pressure	filled with electron gas at negative pressure

Extra Reading: Start with Wikipedia on semiconductors.

Diodes

Interesting things happen when you put p-semiconductors and n-semiconductors together. The simplest example of this is when you just put them next to each other. This is called a **diode**.



If we try to pass a current through this from left to right, then this would be given by electrons travelling in the opposite direction. We connect the right hand side to a negative potential (electron source) and the left hand side to a positive one (electron sink).

Now, the p-semiconductor wants to be filled with an electron gas at negative pressure. If electrons come in they increase that pressure. But the electrons cannot flow from right to left across the join between the two semiconductors because the gas on the other side is at a higher pressure, and flows take place from high pressure to low, not vice versa. The result is that the possible current is blocked at the join between the two semiconductors, there is no overall electron flow, and so no current.

The situation is different if we try to pass a current from right to left. In this case the electron source is next to the n-semiconductor and it increases the pressure of the electron gas in it. That electron gas flows easily into the adjacent low-pressure (in fact negative pressure) region in the p-semiconductor, and then across it to the electron sink. The result is that a current flows easily.

So the diode allows a current to flow easily in one direction, (from p to n), but not the other.

FACT TO REMEMBER: current can flow from a p-type semiconductor to an n-type semiconductor. Current can NOT flow from n-type to p-type.

Of course a diode can be forced to allow a current to flow in the "wrong" direction. If we put enough potential on it, then this forces an electron gas into the p-semiconductor at high enough pressure to overcome the differential. But chips are designed so that this does not happen.

3.2 Transistors

This section, and the rest of the chapter, is examinable.

The standard technology to make integrated circuits is **CMOS** (**Complementary Metal-Oxide Semiconductor**). This uses a particular sort of transistor, called a **MOSFET** (**Metal-Oxide Semiconductor Field Effect Transistor**). These are used as switches in the chip. We begin by describing what they are and how they work.

MOSFET's have four connections:

gate: which acts for us as the control on the switch

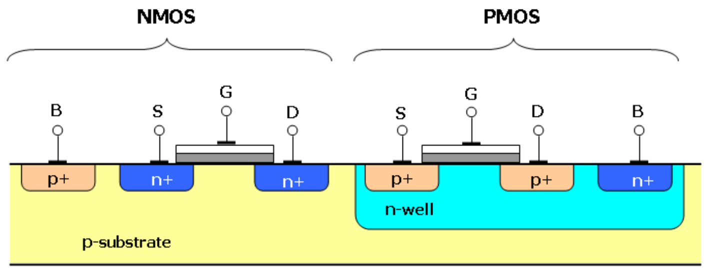
source: a connection for the wire into (or out of) the transistor

drain: a connection for the wire out of (or into) the transistor

body: which we shall ignore

The **gate** controls whether there is a connection between the **source** and the **drain**.

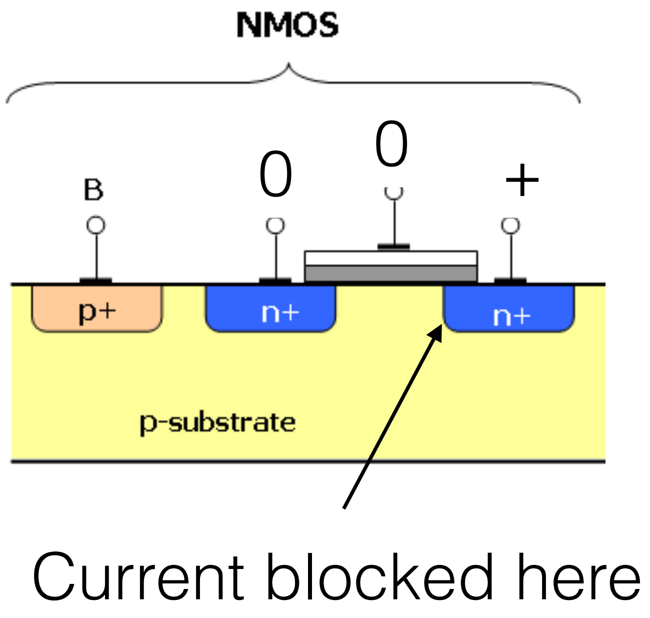
There are two sorts of MOSFETS, **nMOS** and **pMOS** (or **NMOS** and **PMOS**, or **n-channel** and **p-channel**).



We begin by looking at the nMOS transistor.

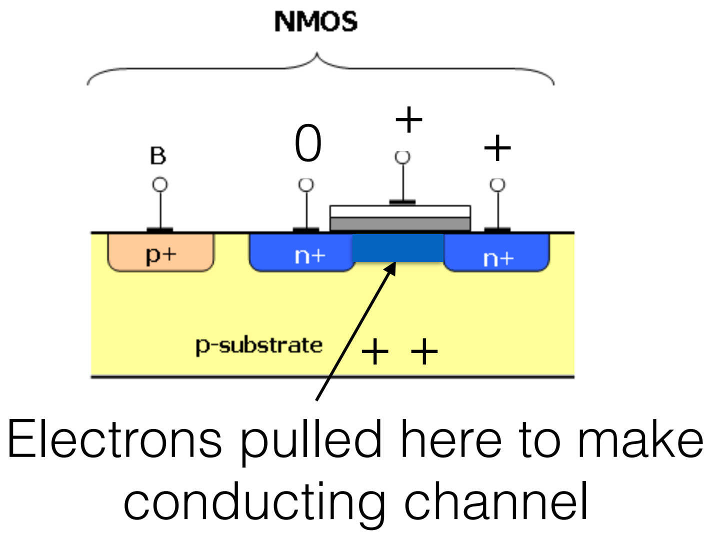
The basic transistor substrate is made out of p-type semiconductor. The gate sits on an insulating layer, so that no current can flow from the gate into the transistor. The source and drain both sit on pieces of n-type semiconductor which themselves are connected to the p-type semiconductor substrate.

If no potential is applied at the gate, then any current from the source to the drain (or from the drain to the source) would have to flow from an n-type semiconductor into the p-type semiconductor, through that and into another n-type semiconductor. But we have seen that current can not flow from an n-type semiconductor into a p-type, and so there is no electrical connection between source and drain.



However, remember that the p-type semiconductor is filled with a "gas" made out of positively charged electron holes. If we apply a potential at the gate, that is positive compared with the potential of the substrate, then that gas is pushed away from the gate, leaving a strip in which there are free electrons, but not holes. This strip is marked blue in the diagram below. It changes the p-type substrate so that there is a strip of what is effectively n-type semiconductor running between the source and the drain, and hence a continuous line of conducting n-type semiconductor. This allows current to flow between the source and the drain (in either direction).

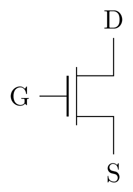
Notice that if we apply a negative potential at the gate of an nMOS transistor, then, holes are attracted to the gate, making the area a stronger p-type semiconductor, and thus ensuring that, as with the neutral gate, there is no connection.



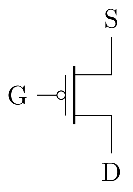
The pMOS transistors are similar, with n- and p-type semiconductors exchanged. The diagram above shows a pMOS transistor built using a large n-type well in a p-type substrate. Source and drain are connected to small p-type wells. So far as positive and negative potentials are concerned, the pMOS transistor is a mirror image of the nMOS. As a result it allows current when there is a negative potential at the gate compared with the potential of the substrate, and does not when there is a neutral or positive.

In implementations, the body of an nMOS is connected to the most negative voltage available (often ground, 0), while the body of the pMOS is connected to the highest voltage available. This means that for the pMOS 0 switches the transistor on, forming a connection, while a high voltage turns it off.

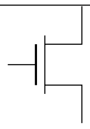
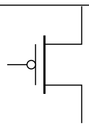
MOSFET's on circuit diagrams In circuit diagrams, an nMOS transistor is represented by the symbol



And a pMOS transistor is represented by



Summary We will use **0** for **low potential (ground)** and **1** for **high potential**. And we will say the transistor is **on** if htere is a connection between source and drain, and **off** if there is not.

	nMos	**pMOS**
AKA	n-channel	p-channel
Symbol		
On	Gate=1	Gate=0
Off	Gate=0	Gate=1
Substrate	p	n
Channel	n	p
Substrate potential	0	1

Additional reading: Find out more about how Intel actually makes these transistors on chips on their website: http://www.intel.com/content/www/us/en/silicon-innovations/standards-22nm-explained-video.html and http://www.intel.com/content/www/us/en/silicon-innovations/standards-14nm-explained-video.html.

3.3 Gates

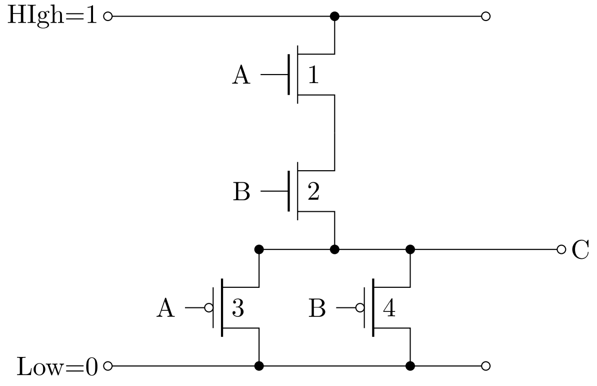
Computers work in binary, 0's and 1's, and in the context of integrated circuits, these are represented by different voltage levels. We are going to take **0** as being a low voltage (ground) and **1** as being high (perhaps 5v). One of the things a cpu does at its heart is to manipulate and do calculations with these 0's and 1's, taking some number of bits as inputs and producing a number of bits as output.

The simplest basic calculations are carried out by standard pieces of circuitry called **gates**. The standard picture is for gates to have one or two inputs and a single output. What they do in essence is calculate a formula in boolean logic.

Example: **(And gate)** The function of an **and gate** is to compute the boolean operation **and** as given by the following truth table (**0** is **false**, and **1** is **true**):

Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

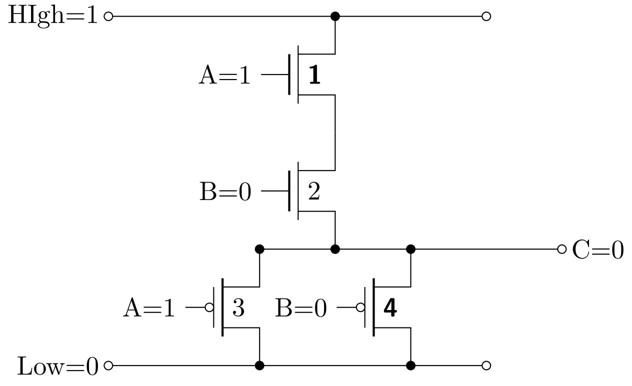
This is computed by the following circuit:



In this circuit, inputs are at A and B, and the output is at C.

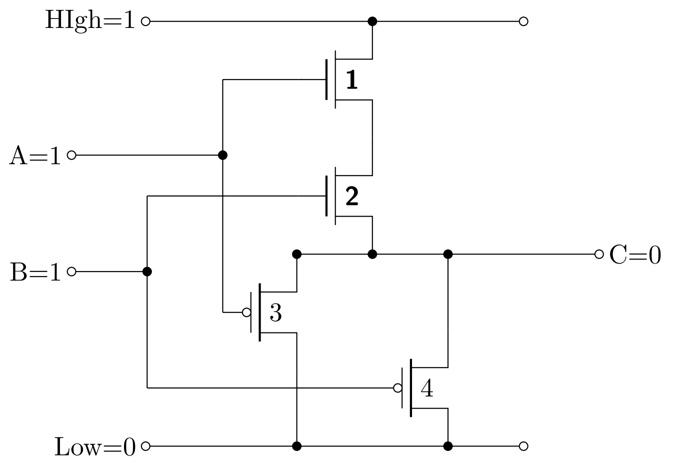
Part of the circuit design is that there is a high potential rail (at the top) and a low potential rail (at the bottom). High potential is never directly connected to low potential, so there is no significant current flow. But the output at C is always connected to one (and only one) of the potential rails at the top and bottom. So C takes the value of the rail that it is connected to.

For example, if A=1 and B=0, then the pMOS transistor at 4 is on, so C is connected to the low rail and the output is 0. The transistors that are on are shown in bold (nMOS are on when their gate is 1, pMOS are on when their gate is 0). Since 2 is off, there is no connection between the high rail and C.



In order to make the diagram simpler, we've left out the links between inputs A at 1 and 3, and inputs B at 2 and 4. In the next diagram we've put them in.

There is a connection between High and C if and only if both of the transistors 1 and 2 are on. This happens exactly when both A and B are 1.



We can summarise the working of the gate in the following table:

A	B	1	2	3	4	C
0	0	off	off	on	on	0
0	1	off	on	on	off	0
1	0	on	off	off	on	0
1	1	on	on	off	off	1

Notice that this table includes the truth table for **and** as the input-output behaviour, and that is why the gate is said to be computing it.

There is a lot of structure to this design:

- top rail as source of High=1
- bottom rail as source of Low=0
- inputs A and B
- output C
- inputs A and B used to control a switching network that connects C to exactly one of HIgh and Low.

In fact there is more structure. The switching network splits into a top half and a bottom half:

- top half connects C to High=1 exactly when (**A and B**) is **true**
- bottom half connects C to Low=0 exactly when (**A and B**) is **false**.

- top and bottom half are **duals** of each other.

We'll explain this last part. The top half contains transistors used as switches in **series**. So there is a circuit only when both switches are closed (both transistors are on). This tests that both of the gates have the right value and so corresponds to an "and" of the gate values.

The bottom half contains transistors used as switches in **parallel**. So there is a circuit when either switch is closed (at least one transistor is on). This tests that at least one of the gates has the right value and so corresponds to an "or" of the gate values.

Moreover, there is an exact correspondence between the transistors in the top half and the transistors in the bottom half, in which both transistors are controlled by the same input and one is an nMOS and the other a pMOS.

FACT: Any boolean function can be implemented by such a gate.

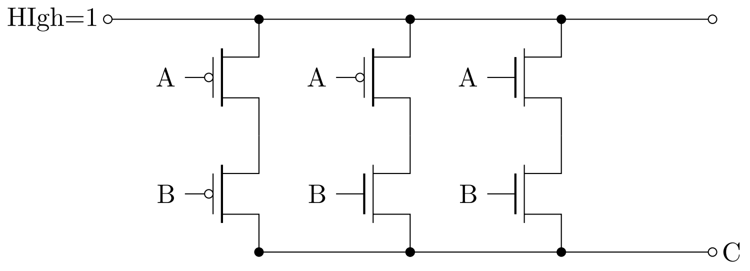
Justification: Start by writing out the input-output behaviour of the function as a table. Let's take for example:

Inputs		Output
A	B	C
0	0	1
0	1	1
1	0	0
1	1	1

(This is (**A implies B**), but we will not use that.)

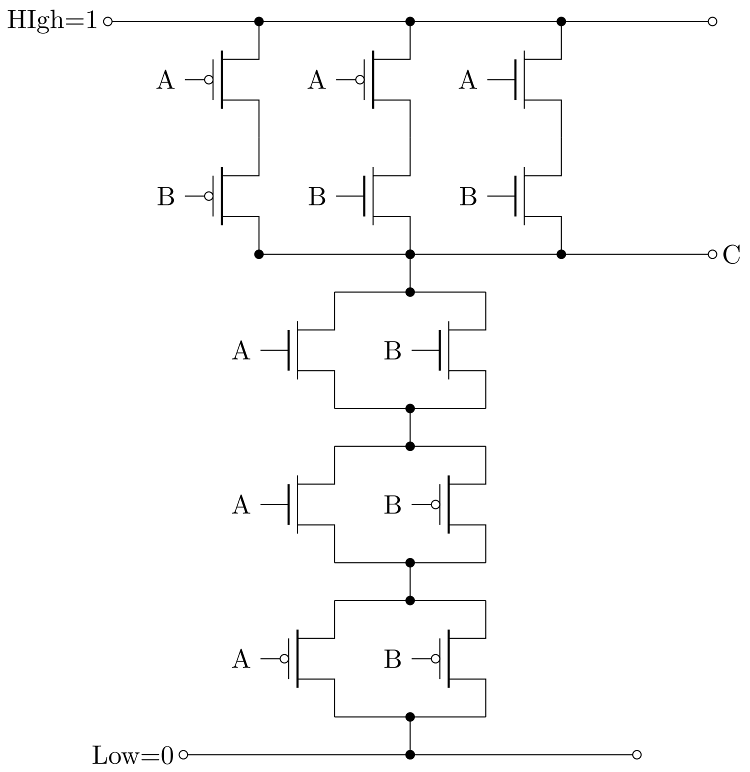
We can now construct the top half of a gate by taking all of the rows that produce 1. Here we have three such rows. For each of these rows we construct a chain with a transistor for each input, where the transistor is an nMOS if the input is 1, and a pMOS if the input is 0. We put these in parallel as the top half of the gate.

In our example we get chains corresponding to rows 1,2, and 4 of the table:



We can now construct a bottom half by dualising the top. We change the type of the transistors, and where we have things in parallel we put them in series, and vice versa.

This gives us:



In fact, what we have in this example is unnecessarily complicated (and in general this algorithm for producing gates will give you circuits that have more transistors than necessary). It would be simpler to take the single line that produces 0, use this to generate the bottom half of the gate, and then dualise that to get the top half.

Self-test

☐ 1. What is the simpler gate that results from doing this?

☐ 2. Follow the algorithm above to produce an **inverter** (a logic gate for the boolean function **not**). (Start with the lower half).

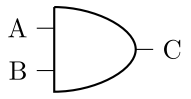
Input	Output
A	C
0	1
1	0

☐ 3. Follow the algorithm above to produce a **nand gate** (a logic gate for the boolean function **nand** = **not and**).

Input		Output
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

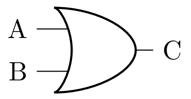
When it comes to designing larger scale circuits, these gates are treated as basic components, and there are symbols for them. We will make particular use of gates for **and**, **or**, **nand**, and **exclusive or**.

and



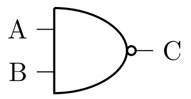
C = A **and** B

or



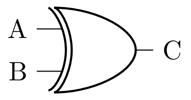
C = A **or** B

nand



C = A **nand** B

exclusive or

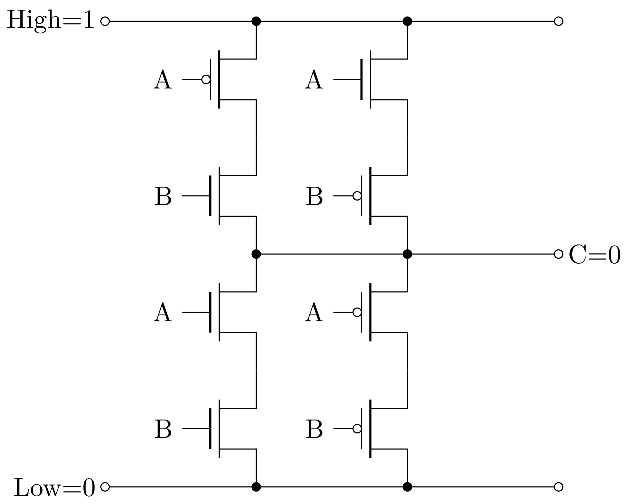


C = A **exclusive or** B

Both **and** and **or** are standard logical operations (or connectives). **Nand** is not, but it is simply the result of applying **not** to A **and** B: A **nand** B = **not** (A **and** B). **Exclusive or** is also unfamiliar, and less simple to write in terms of standard connectives. A **exclusive or** B is true when exactly one of A and B is true, but not both. **Exclusive or** is sometimes abbreviated to **xor**. The truth tables corresponding to these logical operations are:

A	B	A **and** B	A **or** B	A **nand** B	A **exclusive or** B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

The gate that we have not seen a circuit diagram for is **exclusive or**. This diagram does not make use of the algorithm given previously:



Self-test

☐ 1. Verify that the diagram above is indeed an **exclusive or** gate.

3.4 Components: adder

In this section we will find out how to build a simple adder. First we will build a very simple one-bit adder, and then we will show how to put one-bit adders together to make an n-bit adder. This is a demonstration in principle. Actual adders are not necessarily made like this, but we will see how the technology can be used to add unsigned boolean numbers.

If you are not sure about addition in binary arithmetic, skip this section and come back to it after we have covered it in chapter 10.

Here is the simple boolean single digit addition table:

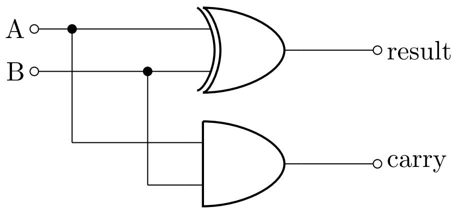
+	0	1
0	00	01
1	01	10

We have given all the results to two digits. You can think of the lower order digit as the result and the higher order as a carry.

The next stage is to look at these as boolean functions:

Input		Output	Logic	
A	B	A+B	A **and** B	A **exclusive or** B
0	0	00	0	0
0	1	01	0	1
1	0	01	0	1
1	1	10	1	0

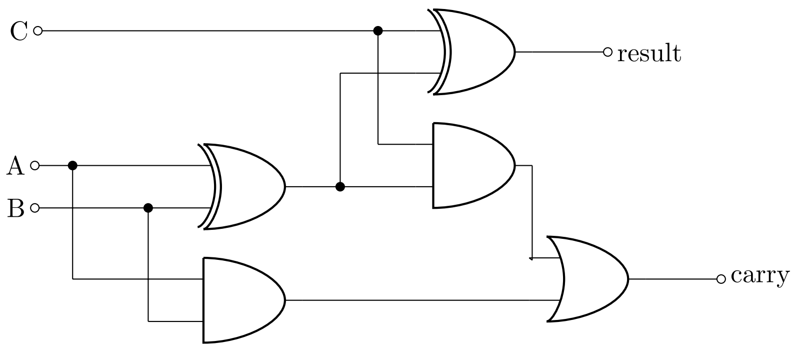
This means that the following circuit can be used to implement this addition operation:



This circuit is called **half adder**. That is because it really only does half the job it needs to. A **full adder** needs to account for a carry coming in as well as its two arguments. It can be implemented as basically two half adders, the first to add the two arguments, and the second to add in the carry. Here is the truth table for the logic you need (using **xor** for **exclusive or**):

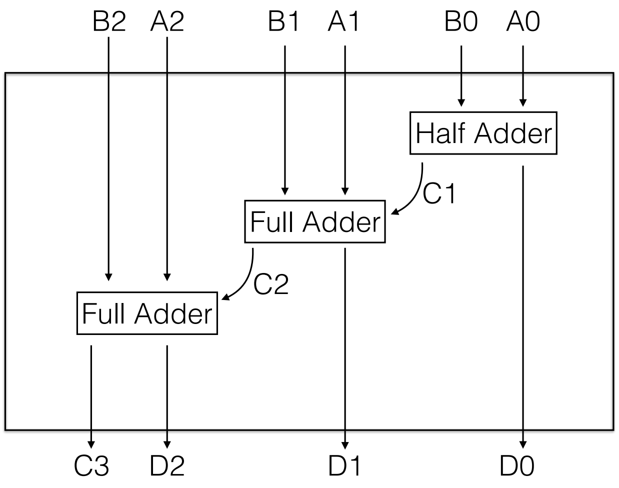
Inputs		Carry	Output	Logic				
A	B	C	A+B	A **xor** B	(A **xor** B) **xor** C	A **and** B	(A **xor** B) **and** C	(A **and** B) **or** ((A **xor** B) **and** C)
0	0	0	00	0	0	0	0	0
0	0	1	01	0	1	0	0	0
0	1	0	01	1	1	0	0	0
0	1	1	10	1	0	0	1	1
1	0	0	01	1	1	0	0	0
1	0	1	10	1	0	0	1	1
1	1	0	10	0	0	1	0	1
1	1	1	11	0	1	1	0	1

Comparing the first and second digits for the result with the logical expressions, we can see that the first (lowest order) digit of the result is (A **xor** B) **xor** **C**, and the second digit (the final carry) can be calculated as (A **and** B) **or** ((A **xor** B) **and** C). That means it can be calculated using the following circuitry:



Once we have a full adder, we can chain them together to make an n-bit adder. Here is a 3-bit adder built from a half adder and two full adders. The inputs are A's and B's, and outputs D's. The C's are carries. So if we are adding 011 to 110, then the inputs are

A2	A1	A0	B2	B1	B0
0	1	1	1	1	0



Self-test

☐ 1. Verify that you understand how this 3-bit adder works by adding 011 and 110 using binary long addition, and then checking how the adder given below does it. You should see a digit by digit correspondence between the paper and pencil method and the working of this adder.

☐ 2. Draw the diagram of an 8-bit adder.

At this point we have seen how to construct a general n-bit adder, at least in principle. It is not very efficient as the calculation of each bit has to wait on the previous ones. Actual hardware adders do a better job of controlling this wait and doing things in parallel, at least in the average case.

We are not going to cover how to multiply unsigned binary numbers, though you should now be in a position to see that it is at least possible. We are also not going to cover arithmetic operations on binary floating point, but again you should be able to see that they are possible, at least once we have covered the relevant material on floating point representation.

3.5 Flipflops

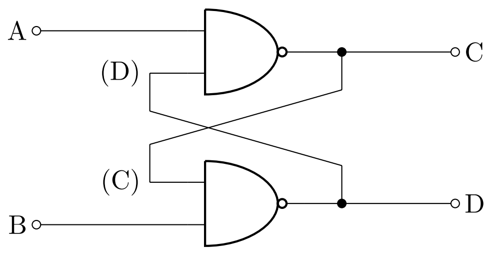
So far we have looked at how digital circuits can be used to do calculations. The circuits we have looked at take inputs and give outputs. Given the same set of inputs they always give the same outputs. The terminology for this is that they are **combinational systems**.

Definition [combinational circuit]: A logic circuit whose outputs at a specified time are a function only of the inputs at that time. (source: encyclopedia.com)

The next circuit we look at does not have this behaviour. Its outputs depend on what happened to the circuit previously. They depend on what **state** the circuit was in when its inputs were supplied. It is a **sequential circuit**.

Definition [sequential circuit]: A circuit whose outputs at a specified time depend not only on the inputs at that time, but on the sequence and ordering of previous inputs.

A basic flipflop In the literature this is also referred to as a **latch**. There are many type of flipflops (latches), but this is the simplest. It is made out of two **nand** gates. It has two inputs (A and B) and two outputs (C and D). Each input is fed into a nand gate, as one of its inputs, and the output of that nand gate is fed back to the other as the nand gate's second input.



This design is symmetric about a horizontal line through the middle (exchange A with B and C with D, and we get the same circuit).

We now find out what happens to this circuit when we give it various inputs.

First, nand gates have the property that if either input is 0, then the output is 1 independent of the other input.

Let's suppose that at least one of A and B is 0. We'll use the example A=0, but the argument when B=0 is similar. If A=0 then the output of the top nand gate is 1: C=1. The inputs to the bottom nand gate are therefore C=1 and B. If B=0, then the output is D=1, and if B=1, the output is D=0.

This means that if either A=0 or B=0 then the circuit behaves in a combinational way, with its outputs depending only on its inputs.

The remaining possibility is that A=1 and B=1. In this case we don't immediately know what value either of C or D takes. Let's try D=1. In this case the output of the top nand gate is C=0. The bottom nand gate has inputs

C=0 and B=1, so its output D=1, which is consistent with our assumption. So this is a possible stable configuration of the circuit.

Now let's try D=0. In this case the output of the top nand gate is C=1. The bottom nand gate has inputs C=1 and B=1, so its output D=0, which again is consistent with our assumption. So this is also possible stable configuration of the circuit.

Since we have covered all the possibilities for D, these are the only two stable configurations of the flipflop when A=1 and B=1.

This is summarised in the table below listing all of the possible configurations of the flipflop.

A	B	C	D
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	1	1	0

The result is that when A=0 or B=0, the circuit behaves combinationally. But when A=1 and B=1, then the circuit behaves sequentially. Its state does not depend simply on the current inputs. In fact it depends on the previous state of the circuit as follows:

A	B	C	D	next state on A=1, B=1
0	0	1	1	uncertain: flipflop can go to either C=0,D=1 or C=1,D=0
0	1	1	0	flipflop stays in C=1, D=0
1	0	0	1	flipflop stays in C=0, D=1
1	1	0	1	flipflop stays in C=0, D=1
1	1	1	0	flipflop stays in C=1, D=0

If we give the flipflop the input (A=0, B=0), then its behaviour will be unpredictable. But if we avoid that, and only give it inputs (A=0, B=1), (A=1, B=0), and (A=1, B=1), then its behaviour will be perfectly predictable. However, its behaviour for (A=1,B=1) will depend on the previous state. This means that the flipflop has a very simple memory.

3.6 Components: memory

In this section we show how the flipflop we studied in the previous section can be used to implement a simple 1-bit memory cell. This is a proof of concept. These kinds of techniques are used for some of the faster memories, but actual RAM uses a different approach which we will describe later.

We start by describing what a 1-bit memory cell does. It provides four functions:

store 0: stores a **0** in the cell, erasing what was there before

store 1: stores a **1** in the cell, erasing what was there before

hold: holds the value currently stored

read: gives the value currently held in the cell

At each time-point the cell is given one of these instructions. For it to operate as a memory cell, **read** should always produce the value placed into the cell at the most recent **store**.

The flipflop can provide these four functions if it is operated according to the correct protocol.

In this protocol, the flipflop is always given inputs (A=1,B=1) except when a store instruction is issued. The **store 0** instruction is issued by giving the flipflop inputs (A=0,B=1). The **store 1** instruction is issued by giving the flipflop inputs (A=1,B=0). The **hold** instruction is issued by giving the flipflop inputs (A=1,B=1). **read** is implemented by giving the flipflop inputs (A=1,B=1) and taking the value from D, which is always set at the current value stored in the flipflop. The flipflop is never given inputs (A=0,B=0).

Example: Consider the sequence of states the flipflop goes through corresponding to the input sequence: **store 0; store 1; read; hold; store 0; hold; read;**

instruction	A	B	C	D
store 0	0	1	1	0
store 1	1	0	0	1
read	1	1	0	1
hold	1	1	0	1
store 0	0	1	1	0
hold	1	1	1	0
read	1	1	1	0

Notice that the value at D is always that of the last store, and in particular, at the two reads it is correctly first 1 and then 0.

Self-test

☐ 1. Check the flipflop behaves correctly as a memory cell for a different sequence of input commands, e.g. **store 1; read; store 0; hold; read; store 1; store 1; hold; hold; read;**

When we studied the adder, we first built a simple 1-bit adder, and then showed that by replicating it, and connecting the copies together in the right way, we could construct an n-bit adder for any n we liked. A similar thing is true here. In order to construct a large memory, we can build it from replicas of this single cell, connected together in the appropriate way. The details are complicated, but it should be obvious that it can be done by using the appropriate switching network to send the right signals to the right cells. This is how some kinds of memory are built, but not the RAM that is used to build main memory.

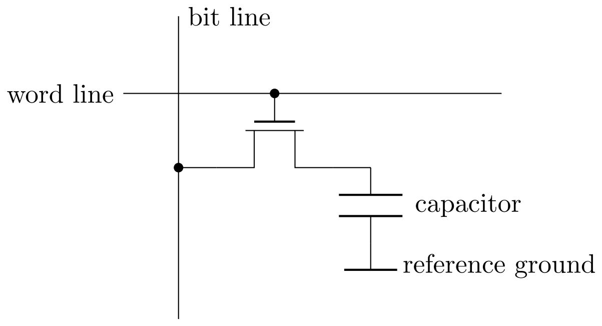
3.7 Memory: RAM

Actual random access memory is not simply built from transistors. It uses **capacitors** to store the data.

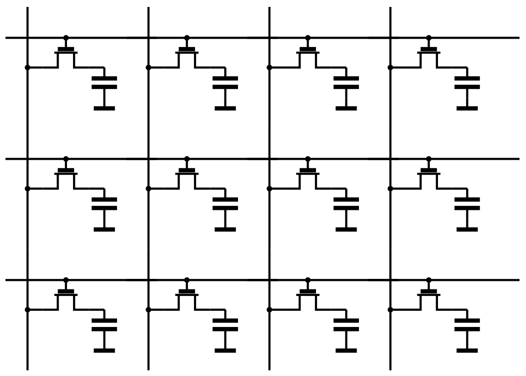
Definition [capacitor]: A **capacitor** is a device that stores an electric charge.

A single RAM device is built as a two-dimensional array of cells, each of which stores one bit. The axis in one direction gives addresses. The axis in the other gives an array of bits corresponding to a single memory read. Even though memory may conceptually be organised as an array of 32-bit "words", a single hardware read can give more than this.

The diagram below shows a single cell. When there is a high voltage on the word line, the transistor is switched on, allowing the potential in the bit line to be stored in the capacitor, or, conversely allowing the potential stored in the capacitor to be read at the bit line.



These cells are arranged in an array:



The basic idea is that when a single word line is high, then the values stored in the RAM can be accessed on the bit lines. But there are a number of difficulties in making this practical, such as:

- In order to get as much RAM onto a chip, everything is as small as possible. In particular the capacitors (which are built using the same kinds of semiconductor materials as the transistors) are small and do not store much charge.
- Reads reduce the charge stored in the capacitor, so the contents need to be put back.
- The capacitance of the bit lines is as much or more than the capacitance of the capacitors, so the potential that can be read is low and has to be amplified.
- Charge leaks from the capacitors, and so the data stored has to be periodically refreshed.

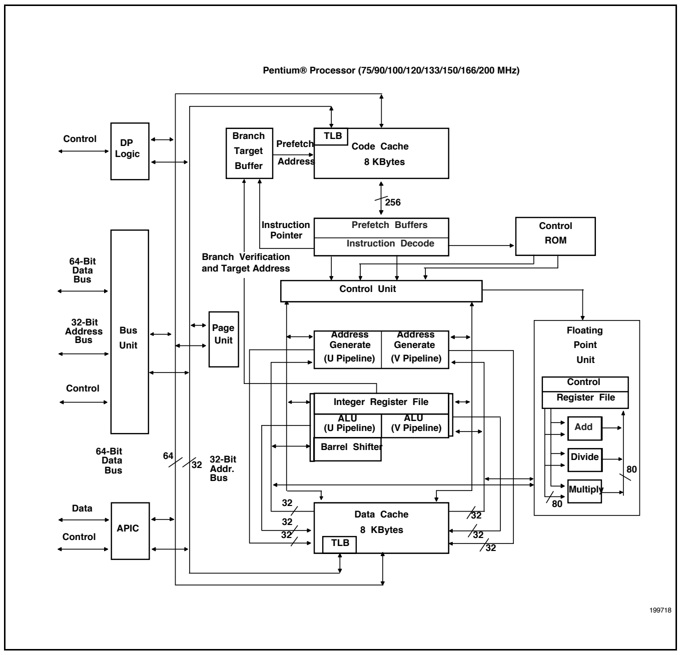
How these problems are solved is beyond the scope of this course. For more information see Bruce Jacob's lectures: `http://www.ece.umd.edu/class/enee759h.S2003/lectures/Lecture2.pdf` or the book by Keeth and Baker: "DRAM Circuit Design: A Tutorial" (`https://www.researchgate.net/publication/293483172_DRAM_Circuit_Design_A_Tutorial`).

3.8 CPU overall design and physical layout

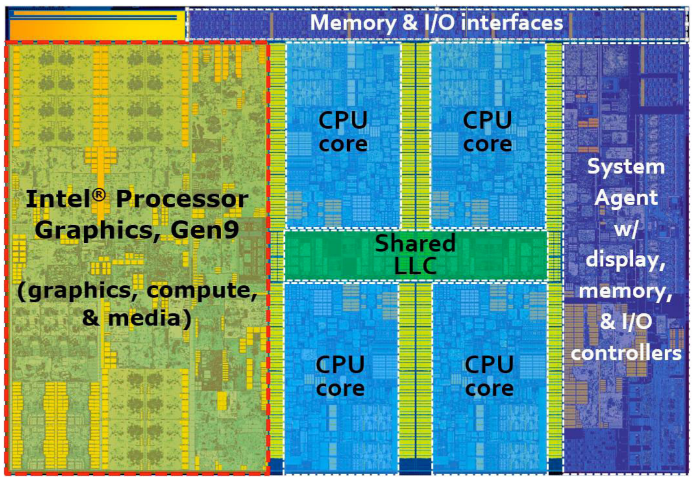
The point we have reached is that we can see roughly how basic computational functionaility can be implemented as circuits that can then be used as components to put on chips.

Complex integrated circuits, such as cpu's have a lot of these components, and are designed at various levels of abstraction. Lower level details, such as the physical layout are often handled by computer programs. One result of this is that layouts have become more complex, with the physical grouping of components less closely linked to the logical structure. Reasons for doing this include improving communications and spreading the areas of activity in order to reduce heat generation. However, major components can still be seen on pictures of the dies used to produce chips.

The following block diagram shows a single core cpu from the 1990's (an Intel Pentium). It is easy to see the different functional parts (for example control, cache, floating point unit, . . .).



Here is the actual physical layout for a more recent chip, a four-core Intel Skylake processor, which, along with the four cores incorporates a GPU:


Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

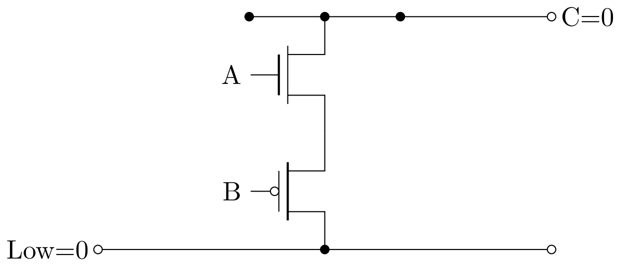
Considerable detail about how this chip is structured can be found at: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake`.

The most famous chip design companies are currently ARM and Intel. Intel has a classic business strategy that involves the company carrying out all levels from chip design through to fabrication. ARM, on the other hand, only sells designs and specifications, and it does this at different levels from a functional specification through to basic designs. It is useful to learn a little about this different approach (as it exposes some aspects of the production that are hidden inside the company by Intel's overall approach). See `http://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works` for an accessible introduction.

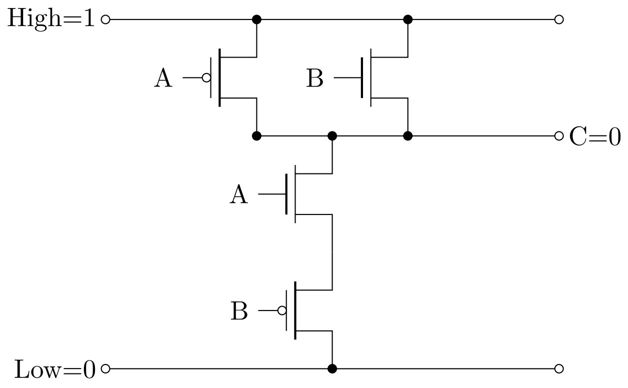
3.9 Self-test solutions

☐ 1. What is the simpler gate that results from doing this?

Solution: We look at the single line in the table that produces 0. This has inputs A=1 and B=0. That gives us a single chain for the bottom half of the gate with A controling an nMOS and B controlling a pMOS.



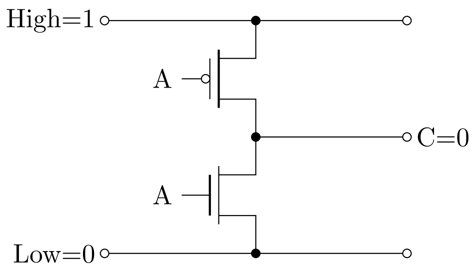
The two transistors are in series, so to dualise, we interchange pMOS and nMOS and put them in parallel.



☐ 2. Follow the algorithm above to produce an **inverter** (a logic gate for the boolean function **not**). (Start with the lower half).

Input	Output
A	C
0	1
1	0

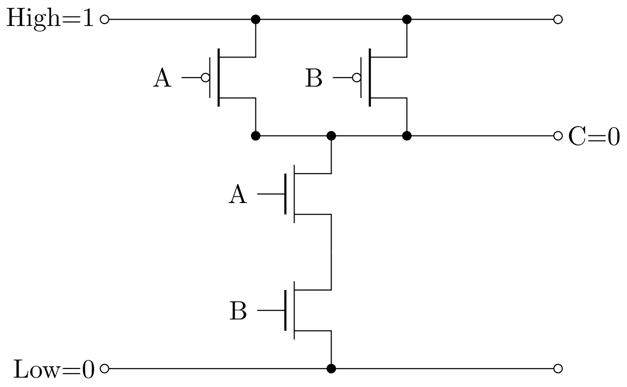
Solution: The top half is a simple pMOS and the bottom is an nMOS.



☐ 3. Follow the algorithm above to produce a **nand gate** (a logic gate for the boolean function **nand** = **not and**).

Input		Output
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Solution: Start with the single line 4 to give the bottom half, constructed from two nMOS transistors in series.


High=1
A
B
C=0
A
B
Low=0