Key concepts covered in Pre-recorded lectures L1 – L12

Pre-recorded lecture L1 (3.1 – 3.25)

<Introduction>
3.1
In this lesson, we will be talking about Logic Gates and Boolean algebra.
Digital circuits can be found in all the devices that we use, for example smartphones, computers, washing machines, cars and etc.
Logic Gates are the basic building blocks of such digital circuits.
Boolean algebra will help us to describe, design and simplify digital circuits.

3.2
First of all we would like to introduce the two Boolean constants.
There are only two constants in Boolean algebra: TRUE or FALSE.
Sometimes we call them by other names, such as logic High, logic Low; or simply HI, LO; or even more commonly, we call them 1 (one) and 0 (zero).
In all the digital circuits that we are going to look at, all the inputs and outputs are considered as Boolean variables, or logic variables.
Such variables can only take on these two values: either 0 or 1.
This makes digital logic fairly simple to analyze.

3.3
In the physical world, how do we actually represent different logic levels?
The most common way to represent logic levels is to make use of electrical voltage.
In terms of electrical voltage, you can visualize simply using say, batteries.
Using batteries, you can create different voltages.

On this diagram, what you see are two different types of devices.
One of them is called the TTL device, the other one is called the CMOS device.
You will get to know more about these two devices later on in this course.

If you look at the two rectangles, you find that we have partitioned the voltage between 0 to 5 volts.
The main partition: you can see a small voltage range between 0 to 0.8V allocated to Logic 0; and then you have another voltage range from 2 to 5 volts allocated to Logic 1. However, this is only true for the TTL device.
In the case of the CMOS device, you have a different voltage range for logic 0 and you also have a different voltage range for logic 1.
This is not going to be a big problem, but it is something that we should be aware of.
In other words, when different devices work together, it is important that we take note of their voltage range to make sure that they can work together.
The other point that you want to take note of is: there is a middle-range in which it is known as indeterminate.
That means it is neither logic 1 nor logic 0.
In a properly designed digital circuit, the voltage level should not be in the indeterminate range.
It should always be either in the logic 0 range or the logic 1 range.
You will also notice that typically we use low voltages to represent logic 0, and we use high voltages to represent logic 1.

3.4
Now what does a typical logic circuit look like?
Usually such a circuit will have one or more logic inputs; similarly it should also have one or more logic outputs; and the outputs are related to the inputs by logic functions.
If you look at this very simple diagram: a rectangle showing that we have three inputs coming into the circuit, and one output coming out from the circuit.
The output is dependent on the inputs in a determinate fashion; it is not a random fashion.
Within this logic circuit, depending on the complexity of the circuit, it may be made up of smaller circuit blocks.
In other words, you can create simple digital circuits and then you can interconnect them together to form a complex circuit that performs complex functions.

3.5
Typically we will use a truth table to describe the logic function of any digital circuit.
What do we see on a truth table?
The truth table will tabulate the output of a logic circuit in response to all the possible logic inputs.
Depending on the number of inputs you have for the circuit, the truth table will have different combinations.
On a truth table, all the possible input combinations will be listed out.

To make the truth table easy to read, the input combinations should be arranged according to the binary counting sequence.

3.6
Now we take a look at a typical truth table.
On this truth table there are N inputs.
N could be an integer, anywhere from 3 to 5, or even to 8, depending on the complexity of the circuit.
We will arrange each of the inputs, one per column.
Since there are N inputs, we expect to see N columns on the truth table.
Assuming that this circuit only has one output, then you will only see one column for the output.
What you will notice is: because there are N-number of inputs, you will see altogether 2^N (2 raised to the power of N; also 2^N) rows on the truth table.
If you examine closely, every row represents a different input combination.
For example, if you look at the first row, it represents the situation when all the inputs are 0.
If you take the other extreme end, this (last) row will represent all the input combinations when they are all 1.
So altogether you will find that there are 2^N different input combinations; and for each of these combinations the output will be produced accordingly.
This is the purpose of a truth table.

<3 basic logic gates>
3.7
Now we will proceed to introduce the three basic logic operations.
Three operations that we want to learn about, the first one is called Logical Addition, or simply the OR operation.
The second one is called Logical Multiplication, or simply the AND operation.
The third operation is Logical Complement, or inversion, or otherwise known as the NOT operation.
In digital circuits, these operations are realized by electronic devices which are known as logic gates.

3.8
We will look at the first type of logic gate which is called an OR gate.
There are a few things that we need to know about the OR gate.
The first thing is we must be able to recognize the logic symbol.
In this case what we can see is there are two inputs, A and B, going into the logic gate and one output X coming out from the logic gate.

To describe this logic gate we can make use of what we call a logic expression.
We can write the expression in this form: X = A + B
or we can also write it in this form: X = A OR B.
You will find that it is more common to use this expression (X = A+B).
So if you look at a 2-input OR gate, the output is related to the inputs by this expression.

3.9
Now we look at the truth table for this 2-input OR gate.
On the left are the two inputs A and B.
Since the OR gate only has one output it has one (output) column.
Since there are two inputs, therefore we see $2^2 = 4$ rows on the truth table.
In each of these rows, you will find that the inputs A and B will change in the manner of, say when A=0, B=0;
or in the second condition A=0, B=1;
in the third condition A=1, B=0;
in the last condition A=1, B=1.
So altogether there are only four different possible input conditions.
For each of these input conditions, the output is specified.
When A=1, B=1 then the output X is 1.
When either A or B is a 1, the output is also 1.
The only time when the output is 0, is when A=0, B=0.
This is the characteristic of OR gate.
We can summarize it by saying: the output X is 1 if at least one of the inputs is equal to 1.

3.10
An OR gate is not limited to having only two inputs.
It can have three or even more inputs and the behavior is the same.
If you look at the logic symbol, you find that the shape of the OR symbol is the same.
The only thing that has changed is the number of inputs.
The way that we write the logic expression is also the same.
For the 3-input OR gate, the output will be equal to 1 as long as at least one of the inputs is 1.

3.11
This can be easily seen from the truth table.
Now that we have three inputs, 2^3 is 8 and therefore altogether we will see eight rows in the truth table.
For each of the input combinations, the output X is listed.
It is not surprising that the output X is only 0 provided that all three inputs are 0.

Otherwise, the OR gate will return a 1 in the output.

3.12
Now we are ready to look at the second operator, which is called the AND operator or simply the AND gate.
If you look at the logic symbol, you will find that it looks somewhat different from the OR symbol.
Therefore, when we draw the OR gate or the AND gate, we have to draw the correct shape so as to express its logical function clearly.
On the 2-input AND gate, what we can write as a logical expression is this form (A●B) or we can also use this form (A AND B), and most often we will use this form (AB).
So we will not put a "●", we will also not write the word "AND" between the two variables A and B.
It is understood that we are doing an AND operation between A and B.
Similarly if we have a 3-input AND gate, which is what we see here, we can simply write the Boolean expression as X=ABC.
The AND gate behaves such that the output will only be 1 provided that all the inputs are equal to 1.

3.13
We can see this very clearly from the truth table.
In this case for the 2-input truth table, we find that the output X is only equal to 1 when A=B=1; otherwise the output X equals to 0.

3.14
This is a 3-input AND gate and therefore we have 8 possible input combinations.
Because it is an AND gate, we will only get a 1 in the output provided that inputs A=B=C=1; otherwise the output will return a 0.

3.15
We come to the third operator which is the NOT operator, or very commonly we refer to it as the inverter.
The inverter is the simplest gate to understand because it has only one input and one output.
The output is exactly opposite to the input.
This is the logic symbol, and we put a bubble at the output to indicate that the output is simply the opposite of the input.
How do we write the logic expression?
You will find in many textbooks that it is written in this form: $\overline{\mathrm{A}}$ to show that it is an inversion of A.

In this course, we will use this notation (A’).
(you may write $\overline{\mathrm{A}}$ if you prefer)

3.16
The truth table for the NOT gate is really simple.
There is only one input and only one output.
If the input is 0, the output is 1.
If the input is 1, the output is 0.

3.17
Now I would like to quickly mention that there is a related logic device, which is known as the buffer.
The buffer does not perform any logic operation, as you can see from the truth table.
The output is exactly the same as the input.
We also do not see any bubble at the output of the logic symbol.
Therefore the buffer does not change the logic of the input (before reproducing it at the output).
Why would we want to use a buffer?
Usually a buffer is used not so much because of logic purpose, but it could be due to electrical requirement, which we will come to know more about later on in the course.

<Timing diagrams>
3.18
Now that we have learnt the 3 basic logic gates, AND, OR and NOT, we want to put them together to build a logic circuit.
In this simple circuit, we have an inverter whose output is connected to an input of this OR gate.
The inverter’s input is A, so this signal is A’.
B is the other input of the OR gate.
Therefore the Boolean expression for output X can be written as this (X=A’+B).
The value of X is determined by A and B.
A and B are input signals of the circuit.
They are expected to change between 0 and 1 at various time instances, which will cause output X to change.
Note that a circuit has no control over how its inputs may change.
These waveforms are given to us:
<Animation>
This is how we read the waveforms:
A and B are 0 at first.
A changes to 1 at time t1.

B changes to 1 at time t2,
then A changes to 0 at time t3.
Given these input waveforms, we would like to sketch the waveform for output X.
To do this, we may use logical reasoning.
Look at the expression: X=A' OR B.
If you recall:
the OR gate output is 1 whenever either input is 1.
In other words, the OR gate output is 0 only when both inputs are 0.
Therefore X is 0 only when A=1 and B=0.
On the waveforms, we look for the time instances when A is 1 and B is 0.
This happens only during this time.
Therefore, output X will be 0 only during this time;
and 1 otherwise.
This is how we draw the timing diagram.
The output waveform shows that X is 1 at first, at time t1 it changes to 0; and at time t2 it changes to 1.
This type of simplified timing waveforms are what we will usually sketch by hand.

3.19
However, in technical documents, timing diagrams will show more details.
Let's look at one such example.
<Animation>
These are 3 input waveforms of a circuit.
This includes the output waveform.
I'd like you to take note of 3 things.
First: rise time.
The signal X is changing from 0 to 1, but a slope - instead of a vertical edge - shows that the transition takes time to happen.
This short amount of time taken by a signal to rise from Low to High is called "rise time".
Second: fall time.
Similarly, for the signal to fall from High to Low, the transition will require a short amount of time.
This is called "fall time".
Third: propagation delay.
The arrows highlight the changes in output F, which are caused by one or more changes in the inputs.
Notice that after the input has changed, it takes a short time delay before the output would change.
This time delay is called "propagation delay".

Rise time, fall time and propagation delay are all very small, in the range of nanoseconds (ns), but they are not zero.
You will learn more about them later in this course.

<Order of precedence>

3.20
Now we have come to the part where we want to talk about Boolean algebra.
Boolean algebra is helpful in analyzing logic circuits.
It allows us to describe or express logic operations mathematically.
Boolean algebra is very much similar to normal algebra, but it is actually simpler because it does not have fraction, and it does not have negative numbers.

3.21
Similar to normal algebra, when we evaluate Boolean algebra, we also need to observe a certain order of precedence.
So if we have a logical expression that involves different logical operations, we need to evaluate them in this order.
The highest precedence would be complement over single variable;
followed by any expression that we keep within parentheses;
then followed by the AND operation;
and finally by the OR operation.
We will look at some examples.
Look at the first example: what we will evaluate first would be the single variable inversion (C’), thereafter we will evaluate this term, BC’, because it involves the AND operation. Lastly we evaluate the entire expression, Y=A+BC’, which involves the OR operator.

Look at the second example, we will once again evaluate C’, because it is a complement over a single variable; but we also have the parentheses.
So we will evaluate (A+B), and then finally we will evaluate the entire expression that involves the AND gate.

In the third example, we do not have complement over a single variable.
Instead, we have an expression within a parentheses, so we will evaluate (BC) first, after that we will apply the inversion.
Finally we will evaluate the entire expression, because the OR operator has the lowest precedence.

So you will find that the order of precedence is somewhat similar to the order that we encounter in ordinary algebra.

3.22
There are often times that we have to describe logic circuits.
Of course we have learnt that we can use a truth table to describe a logic circuit, but you also recognize the limitations of using the truth table.
The truth table will grow very long when you increase the number of inputs.
Therefore, sometimes it is better to describe a circuit using an algebraic expression.
We look at this example.
We have a diagram here comprising an inverter, an OR gate and an AND gate.
If I want to describe this logic circuit, I can draw this diagram.
Alternatively, I can also write the Boolean expression.
Starting from the left hand side where we see the inputs, we know that this is input A going through an inverter, so this would be A'.
We have the input B and the input A' going into the OR gate, so what we have here would be A'+B.
We have the AND gate with the input C. So therefore the final expression for X is simply (A'+B)C.
This is what we mean by describing a logic circuit using an algebraic expression.

3.23
There are also times that we are given a Boolean expression and we want to find out what is the logic level of the output, given the input values.
If we want to evaluate the logical expression, all we need to do is to substitute the input values into the expression, observe the order of precedence, and we will be able to find out what is the output. We look at some examples.

Let us consider the first scenario.
If at this point in time, the inputs A, B, C, D are such that A=0, B=1, C=1, D=0, what logic level do we expect to see in the output X?
All we need to do is substitute the values of A, B, C, D into the Boolean expression.
What we have is: 0'**.**(1+1)**.**(0+0)' = 1.1.1 = 1. Therefore in this case, X=1.

Let us consider the second example, where A, B, C, D are all equal to 1.
Plug in the values into the Boolean expression: the first step we do is A=1 and we take the inversion. 1' is 0.
We know that 0 AND with anything is going to be 0. Therefore there is actually no need to plug in the values of B, C, D in the remaining expression; because regardless of what is this, this term is already 0. So we can simply conclude that X=0.

In the last example, if A, B, C, D are all equal to 0, then what we have is A'=0' and then (B+C) = (0+0) which is of course 0.

So once again we come to the situation where we have 0 AND with something else. Therefore the conclusion is X=0.

3.24
In this case, we are given a logic circuit diagram where we have inverters, OR gates and an AND gate, connected in this manner with 4 different inputs: A, B, C, D.
Sometimes we just want to find out very quickly what will be the output X if the input values of A,B,C,D are given.
We can simply evaluate it directly on the diagram without writing any Boolean expression.
Since A=1, after the inverter, it will be a 0.
We are told that B=1; 1 OR with 0 is 1.
We are told that C=0, D=0, C OR with D is still 0; after the inversion we get a 1.
We know that 1 AND 1 = 1. Therefore X=1.

<Implementation of Boolean expression>
3.25
Sometimes we are given a Boolean expression, and we want to build up the circuit.
When we translate a logic expression into a physical circuit, it is known as implementation.
If one has to implement such a logic expression, what it means is: we need to draw up the logic circuit diagram to show how we intend to connect up the diagram.
This kind of activity - you will need to do especially when you go for your lab classes. Before you go to the lab class, you will have to draw up your connection circuit diagram, so that when you are in the lab you will know how to connect up.
Usually we start by drawing the inputs on the left hand side.
Since we have 3 inputs, we will draw them as A, B, C.
Here we have the first term (AC) that we need to create; this can be done by taking the signal A and the signal C, and we send them together into an AND gate. That will give us AC.
The next expression that we need to implement is BC’.
We will need an inverter to invert C; this will give us C’. Then we take the signal B and send it into an AND gate to give us BC’.
Now we come to the very last term (A’BC), we need an inverter for A; after that, we connect the signal B and we also connect the signal C, and we send them into another AND gate, and we get A’BC.
Finally, we need to OR them together, so we will just send them into an OR gate.
This is what we mean by drawing a circuit diagram.
However, when you look at a diagram like this, you find that there are places where wires are crisscrossing each other, and it can be confusing.
Because when wires cross you do not know which ones are connected and which ones are not.

To make things clear, what we will do is: if the wires are connected together, we will indicate it very clearly.
For example in this case, because these 2 wires are connected, we will put a blob ● there to show that they are connected; similarly over here.
This will clearly show the wires that are connected.
The other wires that crisscross each other (without ●), they are not connected.
When we draw logic circuit diagram, we do not have to draw it every neatly.
What is important: the signals must be clearly labeled, the shape of the symbols must be accurate, and also the output is clearly indicated.

Pre-recorded lecture L2 (3.26 – 3.46)

<Boolean theorems>
3.26
Now we come to topic of Boolean theorems.
Boolean theorems will be used to help us to simplify Boolean expression.
You will find that many of these theorems are similar to those that you have encountered in normal algebra.
The reason why we want to simplify logic expressions is to obtain simpler logic circuits.
Obviously simpler circuits cost less to build and are also less prone to failure.

3.27
We will start with the axioms.
In the digital world, if a logic signal is not 1, then it must be 0, or vice versa.
Anything AND with 0 is 0.
1 AND with 1 is 1.
We have seen it from the AND gate truth table.
Anything AND with 0 is 0; this we have also seen from the AND gate truth table.
Similarly these are the observations we have made from the OR gate truth table.

3.28
Now we will look at the single variable theorems.
In these expressions, when you see the variable X; X refers to a logic variable, or a Boolean variable, and therefore X can take on the value of 0 or 1.
So if you look at the expression like this: X can be 0 or 1 and it does not matter. If you AND X with 0, you will obtain 0.
Similarly, if you take a Boolean variable X which can be 0 or 1, and you AND X with 1, then the result will be X itself. This can be easily verified by referring back to the AND gate truth table.
If you take a variable X and AND with itself, you will get back the same value X.
For example, 1 AND 1 is still 1; 0 AND 0 is still 0.
If you take a variable X and AND it with the inverted copy X’, then you will obviously get 0. Because if X is one, then X’ will be 0 and vice versa.
Similarly, the theorems on the right hand side can be easily understood by referring back to the OR gate truth table.
Essentially if we have any variable X and we OR X with 1, we will definitely get 1.
If we OR X with 0, we will get back the variable X itself.
The variable X OR with itself is still the same X.
If we OR X with its opposite, X’, we are guaranteed that 1 of them will be 1.
The last theorem, (X’)’=X, is very useful.

If you have a variable, if it is 0 and you invert it, you get 1; if you invert it again, you get back 0.
In other words, when you take a signal, or a variable, and you invert it twice, you get back the same thing.
The other thing you probably notice from these two sets of single variable theorems, will be that they enjoy the property of duality.
If you take any theorem on the left hand side, and then if you switch around using this (swap 0 with 1 and vice-versa, swap AND with OR and vice-versa), you find that you obtain the theorem on the right hand side.

3.29
These are the same set of single variable theorems, just that they are now expressed in diagrammatic form.

3.30
Now we will look at the multivariable theorems.
Multivariable means that there will be 2 or more variables involved.
These 3 theorems are not new to us; we have been using these theorems or laws whenever we do algebraic manipulations.
First thing (commutative) is it does not matter whether we perform A+B or B+A. Similarly, whether we perform A AND B or B AND A, it is the same.
Look at the associative laws, if we try to add 3 variables together, it does not matter which 2 you add first.
You can add B and C first, then you add it with A; or you can add A and B first, before you add it with C.
The order does not matter, and it is the same for the AND operation.
Look at the distributive laws, we have also been applying this law frequently in manipulating algebraic expressions.
If we have an expression like this, we can simply ‘so call’ multiply in.
So we have A multiply into B and A multiply into C to give us this (AB+AC)=A(B+C).
Same thing, if we have multiple variables being multiplied in this manner, (A+B)(C+D), we can just multiply in one-by-one.
These laws are not new to us. It is the same set of laws that you have been using in ordinary algebra.

<Multivariable theorems>

3.31
Now we have come to a set of laws that are applicable to Boolean algebra, but they are not applicable to normal algebra.
The first set of laws are called the absorption laws.

There are 2 versions.
Let us look at the first version which is on the left.
The expression with these 2 terms, A+AB, can be simplified such that it becomes A.
The advantage is very obvious: B is no longer in the expression.
What it means is, if we design a circuit like this, effectively we don't need the input B. We can take just the input A.
The proof is given here using distributive law, and one of the single variable theorems, which tells us that 1+X = 1.
Now look at the law on the right hand side, this is a slightly different version.
You will find that it looks similar to the expression on the left, but this part is different.
If I have A+A'B, I can simplify it to A+B.
I am not able to get rid of B, but I am able to get rid of A'; that means I do not need an inverter.
The proof is given here, you can follow it step by step.
You will find that this particular application of the theorem where we combine (A+A') into 1, this is one of the most common techniques that we use, whenever we use algebra to simplify Boolean expression.

3.32
Here the same set of absorption laws are presented in the form of Venn diagrams.
If you look at the Venn diagrams, you will find that it is quite easy to understand the two theorems.
Look at the expression on the left.
You have the set A, which is highlighted in pink, and you're going to OR with the intersection A and B. You find that it is effectively still the same set A.
Look at the expression on the right.
You begin with the set A which is the circle in pink, and then you are going to OR it with this term which is NOT A, but B (i.e. A'B).
A'B would be represented by this partially hidden green square, and you find that if you OR the 2 sets together, effectively it is the same as set A union with Set B.
I hope these diagrams will help you to remember the absorption laws.

3.33
We come to another law, which is called the consensus law.
What we have are 3 terms on the left: AB, A'C, BC, and we can actually simplify it.
If you compare the expression on the left and the expression on the right, what you notice is this term, BC, has been eliminated.
This is a very useful theorem, because effectively you are reducing 3 terms to 2 terms. So you are simplifying the expression.
The proof is given here so you can take it step by step and understand.

I would like to point out that this expression is related to a circuit that you will learn later on in this course. That circuit is known as a multiplexer.
More specifically, I would say it is a 2-input multiplexer.
At this point in time you may not know what a 2-input multiplexer is.
Later on in the course after you have learnt this, you may come back and visit this slide, and you will have a better appreciation.

3.34
We will move on to the final set of theorems - DeMorgan's Theorems.
There are 2 versions of the DeMorgan's Theorems, this is the first version.
If you look at the Boolean expression, a simple way to describe this theorem is:
I have 2 variables and I OR them together; after that I put an inverter over the whole expression.
DeMorgan's Theorem tells us that what I can do is, I can break up the 2 terms: I can convert the OR operator into the AND operator.
But at the same time, I must apply the inversion to the 2 terms that I have separated.
The proof is given in a simple form using a truth table.

3.35
The other version of DeMorgan's Theorem is shown here.
This time what we have is:
we have 2 terms that are AND together, and then we apply the inversion.
DeMorgan's theorem tells us that we can split up the 2 variables, so we will have 1 variable here and 1 variable here; and we have converted the AND operation into the OR operation.
But at the same time, we must apply the inversion to each of the 2 terms.
Again, the proof is given here.

3.36
DeMorgan's theorems are not limited to 2 variables only. It can be generalized to as many variables as is necessary.
Look at the first expression, what you see is: there are many variables and they are OR together; and the entire expression is subjected to an inverter.
DeMorgan's theorem allows us to split up each term: so we will convert each OR operator into a corresponding AND operator; at the same time we must apply the inversion to each variable.
Similarly for the other version of DeMorgan's Theorem.
You will find that DeMorgan's Theorems are useful when we want to simplify expression.

3.37
Now that we have learnt many theorems, we can look at this example.
Let’s say we are given this Boolean expression, and we are told to simplify it.
At this point in time, the word “simplify” may not be very clear to you, because you may not fully comprehend the objective of simplification.
We will be able to talk more about the criteria and objective of simplification in a later part of this course.
Let’s say we are given this expression and we want to simplify.
One of the first thing we want to do is to break up the individual terms.
The theorem we can apply would be DeMorgan’s Theorem.
If we apply DeMorgan’s Theorem, we will be able to separate out the variables, but we have to remember to change the AND operation into the OR operation, and we also need to remember to apply the inverter to each of the terms that we have split up.
So we need to apply it here, need to apply here, and we need to apply here.
We know that when we take a variable and invert it twice, effectively it is back to the same variable.
Therefore, this expression can simply be written as A’+B+C’+D’.
This is the final expression which cannot be further simplified.

<NOR and NAND>

3.38
So far we have looked at the 3 basic logic gates, that is AND, OR and NOT.
We say that they are basic logic gates because AND, OR, and NOT can be used to create any digital circuit.
But we want to introduce two other gates and they are known as the NOR gate and the NAND gate.
These are actually built upon the basic logic gates.
Let us first look at the NOR gate.
If you look at the logic symbol, you find that it is effectively taking A OR B, and then we do an inversion.
That is exactly what this Boolean expression tells us.
The word NOR comes from combining the word N̲OT with O̲R̲.
Similarly, the word NAND comes from combining the word N̲OT and A̲N̲D̲.
Therefore what we are actually doing is taking A and B, send it through an AND gate, and then pass it through the NOT gate.
This is also what the logic expression tells us.

3.39
We look at the truth table for a 2-input NOR gate.
Once again we have the two inputs on the left, and we have the output X.

If you apply the OR gate mentally, and then apply the NOT gate, it should not be difficult for you to visualize this truth table.
You find that the only time you get a 1 in the output, is when A and B are both 0. Otherwise, you will get 0 in the output of the NOR gate.

3.40
For the NAND gate, we can similarly draw up the truth table.
We just need to take the inputs A with B, AND them together, then we invert them, and we will obtain the output X.
It is not difficult to see that the only time we will get a 0 in the output is when A and B are both 1. Otherwise, we will get a 1 in the output.
So we can summarise the NAND gate behaviour such that its output is only equal to 0 when all the inputs are 1.

3.41
The same logic operation can be extended to more than 2 inputs. So you can have 3 inputs, 4 inputs, etc.
Here we are showing a 3-input NOR gate.
Once again, the only way to get a 1 in the output in the NOR gate is when all the inputs are 0.

3.42
For the 3-input NAND gate, the only time you can get a 0 in the output is when all the 3 inputs are 1.

3.43
In summary, we have looked at the 3 basic logic gates (AND, OR, NOT), and we have also looked at 2 other gates (NAND, NOR).
Although we can describe each of them using a truth table, it is helpful if mentally we can understand how each of these gates behave.
Look at the OR gate, a good way to summarise the OR gate is: the output of the OR gate is 1 when any input is 1.
For the AND gate, the output is only 1 provided that all the inputs are 1.
Obviously for the NOT gate, it is very straightforward; because the output is always the opposite of the input.
The NOR gate is exactly opposite to the OR gate; and the NAND gate is opposite to AND.

<Universal gates>

3.44
We have said that AND, OR and NOT are basic logic gates: with AND, OR and NOT alone, you can build any digital circuit that you want.
Why then do we need to learn about NAND gate and NOR gate?
NAND gate and NOR gate have a very useful property, and that useful property is known as universality.
NAND gates and NOR gates are known as universal gates.
What does it mean to say that they are universal gates?
Using NAND gates alone, you can form AND gate, you can form OR gate, and you can form NOT gate.
Therefore, you can create any digital circuit with purely NAND gates.
In other words, you can have a digital circuit that is built 100% purely of NAND gates.
The same can be said for NOR gates.
NOR gates are also universal gates: NOR gates can be used to form AND, OR and NOT gates.
Therefore you can also build a digital circuit using 100% NOR gates.
The equivalence can be easily proved by DeMorgan's theorems.

3.45
On this diagram, we show you how to use a single NAND gate to form an inverter.
All you need to do is to connect the inputs together.
If you want to use NAND gates to form the AND gate, all you need to remember is the NAND gate is actually the AND gate plus an inverter.
Therefore if you have a NAND gate, all you need to do is to invert it back and you will get AND.
If you want to use the NAND gate to form the OR gate, it is a bit more complicated; but you can make use of DeMorgan's Theorem.
Essentially, take A and invert it, take B and invert it, then NAND them together.
So in conclusion, 1 NAND gate can form 1 inverter; 2 NAND gates can form 1 AND gate; 3 NAND gates can form 1 OR gate.

3.46
Similar observations can be made about the NOR gate.
To obtain the inverter from a NOR gate, just tie the inputs together.
To obtain the OR gate from the NOR gate, all you need to do is use an inverter.
To obtain the AND gate from the NOR gates, again we make use of DeMorgan's Theorem.
Take input A and invert it, take input B and invert it, and then NOR them together.

In summary, 1 NOR gate can be used to form an inverter; 2 NOR gates can form an OR gate; and 3 NOR gates can form an AND gate.
If you need to replace your circuit, which comprises AND, OR and NOT gates, with 100% NAND, you will just replace them as illustrated in 3.45.
Alternatively, you can replace your circuit with 100% using NOR gates as illustrated in 3.46.

What is the reason that we want to replace the circuit with 100% NAND or 100% NOR? This is purely from a cost point of view.
If you can build a circuit 100% from the same type of gates, usually the manufacturing cost is lower.
So if you can build a circuit from 100% NAND or 100% NOR, it is usually cheaper compared to building the same circuit but with a certain quantity of inverters, and another quantity of OR gates, and then another quantity of AND gates.
This is the main reason why NAND gates and NOR gates have been used in the industry.

Pre-recorded lecture L3 (3.47 – 3.61)

<Alternate symbols>
3.47
We shall introduce an alternate way of drawing logic symbols.
Listed on this table are 4 common logic gates (AND, OR, NAND, NOR);
and these are the logic symbols we have been using so far.
If we take each Boolean expression, and apply DeMorgan's theorem, we will obtain these expressions.
Based on each expression, the alternate symbol is derived.
Notice each "prime" in a Boolean expression corresponds to a bubble on the symbol.
Look at each pair of symbols, you will notice these patterns:
Bubble at an input or output: if it is present on one symbol, then it will be absent on the other symbol.
The symbol shape: if it is "AND" shape on one symbol, then it will be "OR" shape on the other symbol.
These patterns visually describe DeMorgan's theorems.

3.48
We can use these steps to easily switch between alternate and standard symbol:
Add a bubble if there is none;
remove the bubble if there is one;
and change the shape.
A pair of standard and alternate symbols describes the same logic gate.
In other words, using a standard or alternate symbol only changes the drawing, it does not change the circuit.

3.49
The NOT gate also has standard and alternate symbols.
The only difference is the placement of the bubble; there is no change in the symbol shape.
The two symbols focus on two different parts of the NOT gate truth table.
The symbol makes us interpret, or verbally describe the logic flow in a certain way.
A bubble looks like zero, so we say "zero" when there is a bubble, and we say "one" when there is no bubble.
This symbol says "output is zero when input is one".
The focus is on this row of the truth table.
This symbol says "output is one when input is zero".
The focus is on this row of the truth table.

Therefore the choice of standard or alternate symbol does not change the circuit; it only helps us to focus on a specific logic flow or logic behaviour.

3.50
We can also use the AND gate to illustrate.
The standard symbol says this: both A and B must be 1 to produce 1 at the output.
Conversely, the alternate symbol says this: either A=0 or B=0 will produce 0 at the output.
Now you may ask, "Which symbol should we use?"
We mainly use the standard symbols because they are more familiar to us.
However, there may be occasions to draw standard and alternate symbols in the same circuit diagram to make it easier for a person to describe the logic flow.
When doing so, we follow two simple rules:
Bubble-to-bubble matching;
and say 0 when we see a bubble.

3.51
Let's look at the first example.
We are given this logic circuit diagram.
This is the AND gate alternate symbol and this is the NOR gate standard symbol.
Here we have a bubble output connected to a non-bubble input.
This is a mismatch.
We should get rid of the mismatch.
We may redraw it this way.
We only change the NOR symbol.
Notice the bubble output connects to a bubble input.
This is meant by bubble-to-bubble matching.
We may also redraw it differently.
This time we only change the AND symbol.
Notice the non-bubble output connects to a non-bubble input.
This is also bubble-to-bubble matching.
What is the difference between the two diagrams?
They focus on two different parts of the circuit's truth table.

This diagram says:
Output X=1 when inputs C=0 and at the same time either A=0 or B=0.
This diagram says:
Output X=0 when inputs C=1 or A and B are both 1.
So which diagram should we use?
We use this (left) if we want to show how the circuit produces 1 at output X.
But we will use this (right) if we want to show how the circuit produces 0 at output X.

3.52
Let's look at one more example.
We are given this diagram.
This is a mismatch: non-bubble output connected to a bubble input.
To get rid of the mismatch,
We may draw it this way.
We only change the OR symbol.
Notice the bubble output connects to a bubble input.
This is bubble-to-bubble matching.
We may also draw it a different way.
This time we change two symbols.
Notice the non-bubble outputs connect to non-bubble inputs.
This is also bubble-to-bubble matching.
This diagram says:
X=1 when inputs C=1 or both A and B=0.
This diagram says:
X=0 when C=0 and either A or B is 1.
We use this diagram to show how the circuit produces 1 at output X.
But we will use this to show how the circuit produces 0 at output X.

Alternate symbols are very useful for describing logic flow in an intuitive manner.
For example, we can draw a symbol like this:
c is 1 when a=0 and b=1 (c= a'b)
Which is equivalent to this:
c is 0 when a=1 or b=0 (c' = a+b')

<XOR and XNOR>

3.53
Now we want to introduce 2 other gates.
The exclusive-OR gate and the exclusive-NOR gate.
The exclusive-OR gate is somewhat similar to the OR gate, but it is not the same.
From the logic symbol, you can tell that it is different, because of this additional line.
From the Boolean expression, you can also tell that the exclusive-OR gate is not the same as the ordinary OR gate that we have seen.
Now the exclusive-OR gate is used quite frequently in digital circuits, and therefore it is also given an operator symbol.
The truth table of the Ex-OR gate tells us very clearly that the output X=1 when A=0 and B=1, or when A=1 and B=0.
Notice that Ex-OR is different from OR in this last scenario: if A=B=1, then output X=0.
This is what we mean by the word "exclusive".

That means either A or B is 1, but not both - this is what we mean by exclusive.
What you see here is known as the IEEE symbol.
This is a different way of drawing a logic symbol for the Ex-OR gate.
Next we will talk about the exclusive-NOR gate.

3.54
The XNOR gate is similar to the XOR gate, except for the bubble at the output.
In other words, take A XOR with B, and then you invert it.
It should not be difficult for you to see what happens on the XNOR gate truth table. Whenever the 2 inputs are the same, the output is 1.
Whenever the 2 inputs are different, the output will be 0.

3.55
I have mentioned that XOR and XNOR are very commonly used in digital circuits.
One application of XOR is to compare 2 numbers and to conclude whether they are same or different.
This kind of comparator circuit is called bit-wise comparator.
Let us take a very simple scenario, say we have 2 numbers, a and b.
When we do bit-wise comparison, what it means is: a as well as b, are multi-bit numbers.
Multi-bit could be 2 bits, 3 bits, 4 bits etc.
Let's say in this case a and b are each 3-bit.
So a itself is made up of 3-bits, which we will name them a2, a1, a0.
Similarly b is made up of 3 bits and we call them b2, b1, b0.
I have 2 inputs coming in, a and b, and I want to compare whether they are same or different.
In fact very specifically, I want to design a circuit that will output a 1 when the 2 inputs are different.
I can make use of the XOR gate.
What I am going to do is: compare a and b, bitwise.
I will compare a2 with b2, so I can use an XOR gate to compare them.
Then I will compare a1 with b1, also using an XOR gate; and then I will compare a0 with b0.
From what we have understood of the XOR gate, we know that so long as the 2 bits are different, we will get a 1 in the output.
As long as any one of the 3 comparisons give us a 1, we know that the 2 inputs a and b are different.
Therefore, we can easily combine the information using an OR gate. When this is 1, it means that a is not the same as b.
Take an example, let's say a is the number 101; b is the number 110.
You feed it into the circuit, so you have 1, 0, 1, and you have 1, 1, 0.

So the output of this gate is 0 because a2=b2; because a1 and b1 are different, it will be a 1.
Similarly over here, a0 and b0 are different.
The output here will be a 1, therefore we know that a ≠ b, which is very obvious when we look at it.
This is how the digital circuit will determine for you that a is not the same as b.

3.56
In the event that we come across an XOR gate with multiple inputs, essentially it is an odd-function generator.
Therefore among all the inputs, so long as there is an odd number of 1's, the output will be 1.
Take the case of a 3-input XOR gate, the output is 1 if there is 1 bit at the input which is a 1, or all 3 bits at the inputs are 1s.
Of course we can extend a similar concept to XOR gates with even more inputs. Whenever you feel that you are confused, let's say you have many inputs that you have to XOR together and you are somewhat confused, you can always break it down in this form, progressively XOR 2 items at a time, and you will be able to obtain the output.

<Logic circuits connections>
3:57
So far we have learnt about logic gates, we have learnt how to draw logic symbols, we have looked at Boolean algebra, we know how to describe circuits using truth tables, and it is time that we get to know a little more about the physical logic devices.
In the real physical world, when we want to build a digital circuit, we need to collect the electrical components and connect them up in a meaningful way - to create a physically functioning logic circuit.
There are many options these days to build digital circuits.
For example, we may be using standard logic ICs; or we could be using what we call application-specific ICs; alternatively we may also use programmable logic devices. Depending on the purpose, e.g. it could be for experiments inside a lab, or it could be for development purpose, or it could be for mass production.
Different options can be used depending on the circumstances.
What we would like to focus on would be some small scale integrated logic devices, based on the operators that you have learnt so far; and these are also the components that you will be using in your lab experiment 1.

3.58
This diagram shows us the different views of an IC component.
Usually on the IC, what you find would be a plastic or ceramic protective casing.

If you're able to open up the casing, you will be able to see the actual silicon chip inside.
To be able to use the silicon chip, there must be connections made, and these will be made through the metal pins.
As you can see, on this device there are 14 pins; obviously the 14 pins are not identical.
Therefore it is important that we know how to read the pin number.
If you look at the device from the top view, you identify the location of the notch, you will be able to read the pin numbers in an anti-clockwise manner.
Beginning from the bottom left corner, it will always be pin number 1, and then you just go anti-clockwise, and the pin number will increment.
This is externally how an IC would look like - but this is not how all ICs look like, because ICs can come in different packaging.

3.59
This diagram shows the different kinds of IC packaging that are available.

3.60
We will look at an example.
Let's say we have to implement this Boolean expression in the lab.
So what do we need to do?
First of all, we must identify the components that we can use.
The components that are available to us, the one of the left is called the 7404 hex-NOT.
The word "hex" means 6, which is why you see 6 NOT gates on the IC.
On the right, we have the 7408 quad-AND.
The word "quad" means 4, so you have 4 independent AND gates on the IC.
In order to use these ICs, first thing we have to remember is they are electrical components, so they can only function if there is electricity.
The first thing we must do before we can use them, is to connect them to the power supply.
Different ICs may require different supply voltage, but one of the most common voltage that IC devices use would be 5 volts.
In the lab experiment, you will also be connecting the IC components to 5 volts.
Now we must identify the correct pin on the IC to connect to 5V.
On an IC, the power supply pin is labeled as Vcc.
If you look at this diagram, you find that Vcc is located on pin 14.
So we will connect pin 14 to 5V.
Since these 2 are independent ICs, both of them must be connected to 5V.
You've probably also learnt from your high school physics, that if you are connecting up a circuit, it must be in a closed loop.
Without a closed loop, current will not flow.
In order to complete the closed loop, we have to connect the ground.

On these 2 ICs, we have to identify the ground pin - which is located on pin 7; we will connect them to ground.
Now the 2 ICs have the necessary power supply, and we can then proceed to implement the logic expression, Y=AB’.
The first thing we need to do it to invert the signal B.
There are 6 inverters on the IC, we are free to choose any one of them.
In this case I will choose to use this inverter so I will send signal B into the NOT gate (pin 9), and the output (pin 8) would be B’.
I will now proceed to use the AND gate and there are 4 AND gates; I can choose to use any one of them.
In this case I would choose to use the 1st one (pin 1), and I will send the signal A into pin2 of this AND gate.
Quite clearly the output is available here (pin 3), so this is output Y, which is equal to (A AND B’).

This diagram helps you to visualize the kind of connections that you will make when you go to the lab.
However, we don’t usually draw logic circuit diagrams in this manner.
If you look at this diagram, there are 2 ICs, but on each of the ICs we are only using 1 gate, yet we have to include the entire IC - it can be confusing.
Oftentimes we would like to draw a logic circuit connection diagram before we go to the lab, so that we know exactly how to connect up the circuit.

3.61
Here I will show you what it means when we want to draw a circuit connection diagram.
For the circuit that we want to implement (Y=AB’), we know that we need an inverter, we need an AND gate.
We have the inverter, then we also have the AND gate, and this is the output.
Remember that the first thing we need is to connect the devices to 5V and ground.
To help us know which pins we should connect to 5V and ground, we will write the pin numbers.
Then we make sure that the inputs are labeled, and the output is also labeled.
Depending on the gates that we have chosen, we have to fill in the pin numbers.
So if I am using the same configuration on the previous page, then I am connecting input B to pin 9; I’m obtaining the output from pin 8; and I’m sending it into pin 1 and the other input (A) is from pin 2.
The final output is taken from pin 3.
The additional information that we can fill in is the component number.
This would be 7404, which is the hex-NOT; this would be 7408, which is the quad-AND.
If you draw a circuit connection diagram like this and you bring it to the lab, it will be very

easy for you to connect up the circuit which will accomplish what you have set out to do, which is this one on slide 3.60.

Pre-recorded lecture L4 (4.1 – 4.22)

<Half adder and full adder>

4.1

In this lesson, we will be looking at digital arithmetic.

In digital circuits such as digital computers, electronic calculators, arithmetic operations are carried out on binary numbers.

Examples of arithmetic operations would include addition, subtraction, multiplication and division.

Similar to decimal addition and subtraction, binary addition and subtraction both begin with the LSB, i.e. the least significant bit.

4.2

Let us do a quick revision by looking at 2 examples: decimal addition and subtraction.

We will begin with the addition example.

Notice that when we add a pair of numbers, we always begin with the least significant digit.

In this case, we will add 4 with 3, which gives us 7.

Then we move on to the next position: as we add 6 with 8, we realise that the result exceeds 9; therefore we need a carry; and so on and so forth, and this is how we get the result of 1347 in decimal.

Similarly for subtraction, we also begin with the least significant digit.

4 minus 3 is 1; 6 cannot minus 8; therefore we need to borrow.

We continue with the subtraction and this is how we arrive at the answer.

You will find that in binary addition and subtraction, we will follow a similar process.

4.3

We look at this example using binary addition.

We are given these 2 numbers; the decimal equivalent is given for us so that we can easily check our addition.

Beginning with the least significant bit, we proceed to add: 1+0=1; then we go on to the next position: 1+1=2.

But in the binary number system, there are only 2 symbols: 0 and 1.

We are not able to write down the digit “2”, instead we have to recall that 2(decimal) = 10(binary).

Therefore we need to carry over.

We have a carry over and we have a 0.

Then we continue to add so here we have 1+1=2 again; therefore we have a carry.

Then we continue with the addition and finally we get this.

You can easily do a conversion from binary to decimal to verify that this is 33(decimal).

4.4
Now that we have looked at how to add 2 binary numbers together, we will explore digital circuits that actually can help us to add numbers together.
These circuits are known as adders.
We will be considering first the half adder.
The half adder is a combinational logic circuit that is able to add 2 bits together.
Look at this truth table, the 2 inputs that we want to add together: A and B.
As we add 2 bits together, we know what the possible results are.
If we add 0 with 0, of course we get 0.
0+1=1; 1+1=2.
But as you recall we are not able to write 2, instead we have to write 10: 1 is the MSB, 0 is the LSB.
To be able to distinguish them, we actually give them different names.
For the MSB, we will call it the "carry"; for the LSB, we will call it the "sum".
Therefore if you look at the truth table, 0+0=0; 0+1=1; 1+1=10.
Now this truth table is very simple, and therefore we will be able to write down the Boolean expressions for the outputs, simply by observation.
If you observe for the carry, it is simply an AND gate.
As for the sum, we have learnt in an earlier lesson, this is actually the XOR function.

4.5
Although a half adder is useful - it is able to add 2 bits together, but it has its limitations.
Sometimes, when we add 2 bits together we get a carry.
When that carry is carried over to the higher position, it needs to be added to 2 other bits.
So in such a case, we will need to add 3 bits together.
A half adder cannot be used to add 3 bits.
In order to add 3 bits, we need a different circuit.
This circuit we call it the full adder.
Looking at the truth table, you can see that there are 3 bits that we want to add together.
Actually it doesn't matter what we name the 3 input bits, but in order to make things very clear, we would like to name the 2 bits that we are usually adding together as A and B, and then the 3rd input we actually name it as the Carry input.
Now that we have 3 bits that we need to add together, there will be different results.
Apart from getting 0, 1 or 2(decimal) in the result, now we will also get the result 3(decimal).
But again, because we are not able to write "3" in the binary system, we have to write it as "11" in the binary system.
So this is the MSB, which is the carry out; and this is the LSB, which is the sum.

We have seen 2 circuits: the half adder which is able to add 2 bits together; and the full adder which is able to add 3 bits together.

4.6
The truth table for the full adder is also fairly simple.
Therefore we can obtain the Boolean expressions of the outputs, simply by observation.
If you look carefully at the sum column, you will realise that it can be easily obtain by XOR.
If you look at carefully at the carry out column, you find that so long as any 2 bits of the inputs are 1, we will definitely have a carry.
So getting these 2 expressions can be fairly intuitive.
With these 2 expressions, we will then be able to implement the circuit.

4.7
These are the 2 expressions that we have obtained for the sum and carry out of the full adder.
These are the inputs to the full adder.
You can now see that the full adder circuit is made up of 2 XOR gates, 3 AND gates, and 1 OR gate.
You will encounter the full adder circuit for the rest of this topic because we will be using it to perform addition.

4.8
We would like to do a rearrangement for the Boolean expression for the carry out.
This was the expression that we have obtained earlier.
Now we will replace B with this (A’B+AB); and we will replace A with this (AB’+AB).
Based on the Boolean theorems that you have learnt, you know that this is valid.
After that, we will collect this term (Cin.A.B) and this term (Cin.A.B) together to form this term (Cin.A.B).
We are able to rearrange the expression to this form.
The purpose of arranging the expression into this form, is so that we can implement the full adder using half adders.
So this is an alternate circuit implementation.

4.9
In other words, a full adder can also be implemented using 2 half adders and an OR gate.
This is the OR gate.
You may not recognise this symbol.
This is known as the IEEE symbol - we will be talking about it later on in the course.

<4-bit parallel adder>

4.10

We have seen how a full adder is able to add 3 bits together, we would like to make use of full adders to perform addition of binary numbers.
Usually the numbers we need to add together will not be 1 bit alone.
It is quite common that we need to add, for example, in this case, two 4-bit numbers.
If we have to add two 4-bit numbers together, we will need 4 full adders.
We will lay out the two numbers: this is the first number that we want to add; and this is the other number.
We will line them up in this manner, and we will make use of full adders to add them up.
4 rectangles will represent the 4 full adders.
If you recall: the full adder has 3 inputs.
It is able to add 2 bits together, as well as a carry input.
Beginning at the LSB, we do not have any carry bit (input).
So the carry bit that we are adding in is actually 0.
Now the full adder will proceed to add the 3 bits together.
It will be adding the carry bit, and this bit, and this bit: 0+1+1=10. 1 is the carry out, 0 is the sum.
The carry out will go into the next full adder, and the sum will be at the output.
Now we go onto the next full adder, which will then be adding the carry, and the 2 bits.
Once again we get the result 10; we have a carry-over (1).
The next full adder will proceed to add these 3 bits together, the result would be 11.
It has a carry out of 1, and a sum bit of 1.
At the last full adder, it will add these 3 bits together and the result is simply 1.
You can easily verify this addition by converting to decimal.
It is actually 5 (decimal) + 7 (decimal) = 12 (decimal).

4.11

Now that we have a clearer idea of how full adders can be used to add a pair of numbers, we want to take a closer look at the circuit we can build using full adders.
When we have to add a pair of numbers, which are of multiple bits, we will need more than 1 full adder.
In fact to be specific, if we want to add a pair of numbers, that are each of N-bit, then we will need N full adders.
The full adders, individually, are identical; we need to connect them up in a specific manner.
Since we are linking one to the next, we say that we are cascading the full adders.
This kind of cascaded parallel adder is also known as a ripple adder.

Now when we make use of such an adder, the pair of numbers that we are to going to add together - known as the augend and the addend - are fed into the adder circuits at the same time.
The full adders will then proceed to perform the addition; and the addition is fairly fast.
The only thing that limits the speed of addition would be the propagation delay.
We will talk more about propagation delay later in this course.
In the case of the full adder, later on when you look at the circuit, you will find that the propagation delay will contribute to what is known as carry propagation.

4.12
On this diagram, we will illustrate how 4 full adders can be cascaded to add a pair of 4-bit numbers.
Before we proceed to show the addition, I would like to talk about the naming convention of multi-bit signal.
If you look at the value, there are 4 bits. In order to distinguish the 4 bits, we give them an index number.
By convention, we would like to use the index number 0 for the LSB, and we would like to use a larger number for the MSB.
In the case of these 2 numbers, one of them is called A, the other is called B.
Each of the bit in these 2 numbers can be uniquely identified by referring to it as either the bit B[3] or the other one which is A[1].
Each of the 8 bits can be uniquely identified - this is a very common convention when we name multi-bit signal.
Even at the outputs, you will find that we are also following a similar convention.
The LSB will have an index of 0, whereas the MSB will have the larger index.
In this course, as far as possible - if you are naming your variables which have multiple bits - please try to adhere to this naming convention.
What you will observe next will be the addition process carried out by this 4-bit ripple adder.
Pay attention to the propagation delay especially when for example, these 3 bits (B[0], A[0], C_in) are being added together; there will be a short delay before the carry output and the Sum[0] output will be available.
That is what we mean by carry propagation.
<Animation>
The values are now being added. There is a short delay, another delay, and finally the result is available.
You will have noticed that it takes time for each full adder to produce the carry and the sum output.

If we say that at time t=0, these 3 bits (B[0], A[0], C_in) are available; it will take a short time of propagation delay, let's call it t_{pd}, before this sum bit (Sum[0]) and the carry bit output will be available.
Then the next full adder will be adding this bit (B[1]) and this bit (A[1]) with the carry.
That once again will require t_{pd} before the carry and the Sum[1] outputs are available.
In this manner, what you can see is: every time there is a carry, you will have an additional delay.
Therefore by the time these 2 outputs (C_out and Sum[3]) are available, you would have accumulated 4 times t_{pd} with respect to time t=0.

<Sign-magnitude representation of signed numbers>

4.13
We would like to talk about representing signed numbers.
So far, all the numbers we have encountered are called unsigned numbers.
An unsigned number is a value that has only magnitude; it does not have a positive or a negative sign.
From this point onward, we will be focusing on signed numbers.
In order to distinguish positive numbers from negative numbers, we need an additional bit and that bit is typically known as the sign bit.
There are different ways to represent a signed number.
In this course, we will consider 2 systems.
The first system that we want to look at is called the sign-magnitude system.
The sign-magnitude system, as its name suggests, is very simple.
The number is represented in 2 portions: 1 portion is called the sign, the other portion is called the magnitude.
So for example, we want to represent the signed number +14. 14 is in decimal.
We will first convert 14(decimal) to binary, which is 1110 - this is the magnitude.
Whether it is +14 or -14, the magnitude is the same.
The only difference between them would be the sign bit.
For positive values, the sign bit is 0; for negative values, the sign bit will be 1.

4.14
In the sign-magnitude system, you will find that there are equal numbers of positive and negative values.
More specifically, if you make use of N bits to represent a number in sign-magnitude, you will be able to represent any number within this range.
These are some examples of sign-magnitude representation.
You have +85 and -85, notice that the only thing different between them is the sign bit.
Interestingly, there is a positive representation for 0; there is also a negative representation for 0.

This actually gives rise to some inconvenience if we use sign-magnitude system to carry out arithmetic operations.
Therefore there is a need to look for a different system to represent signed numbers.

<Two's complement representation of signed numbers>

4.15
Now we would consider the second type of representation for signed numbers.
This is known as the two's complement system or two's complement representation. There is one similarity between the 2's complement system and the sign-magnitude system: i.e. the sign bit will tell us clearly whether the number is positive or negative.
In the case of a positive number, the sign bit will be 0.
In the case of a negative number, the sign bit will be 1.
This is exactly the same as the sign-magnitude system.
Now let us look at the same example again.
If we need to represent the value +14, as well as -14 in the 2's complement system, what do we need to do?
For a positive value, it is exactly the same as the sign-magnitude system.
First of all, convert 14(decimal) to binary; that will give us the magnitude (1110 in binary).
We just need to write down the magnitude, and make sure that the sign bit is 0; because it is a positive value.
For a negative value, we know that the sign bit must be 1.
But what do we fill into this portion?
This portion will not be the magnitude; instead we will need to carry out some simple steps to obtain that value.

4.16
We will introduce the 2's complement operation.
Given any binary number, we can always perform a 2's complement operation on it by following these steps.
In the first step, we will invert every bit of the binary number.
This very operation of inverting every bit is actually also known as "one's complement".
So starting from the value 1110, if we invert every bit, this is what we obtain.
We perform 1's complement on 1110, and we obtain this (0001).
The second step requires to add a 1 to this number (0001).
It is important to distinguish arithmetic addition from logical addition.
Logical addition is the OR operation - we also use the + sign.
Therefore it is important for us to be careful in distinguishing between logical addition and arithmetic addition.
Here we will perform arithmetic addition, so adding this number (0001) to this number (1), we will get this (0010).

We say that the 2’s complement of 1110 is 0010.
Interestingly, you find that the relation between this number (1110) and this number (0010) is reciprocal.
In other words, (1110) is the 2’s complement of (0010); (0010) is also the 2’s complement of (111).
Now that we have obtained the 2’s complement of (1110), then (0010) is the number we will use to fill up the 2’s complement representation for -14.

4.17
This is the number (0010) that we have obtained (from slide 4.16).
If you recall, when we take this number (1110) and we do a 2’s complement, we get this (0010).

<2’s complement>
4.18
In digital arithmetic, it is very frequent that we have to perform a 2’s complement operation.
Sometimes you find that doing - step one: 1’s complement, followed by step two: adding 1 - can be quite troublesome.
So it is a good news that there is a shortcut method that enables us to easily obtain the 2’s complement of a binary number.
The steps are given here:
Starting from the least significant bit, we will copy down the bit if it is ‘0’, and we continue to do so until we reach the first bit of ‘1’.
Thereafter, we will invert the remaining bits.
Notice that it is a sequential process that we have to start from the rightmost bit and move towards the leftmost bit.
This method works on all binary numbers.

4.19
Take a look at these examples.
If we have the value ‘+85’ in 2’s complement representation, and we want to convert it to ‘-85’, all we need to do is to copy down the first bit until we reach the first ‘1’.
Subsequently for the remaining bits, we just invert every one of them.

4.20
In the 2’s complement number system, you will find that there is one more negative value than the positive ones.
This is not the case in the sign magnitude system.
If you use N bits to represent signed numbers in the 2’s complement system, this will be the range of values that can be represented.
So in the case of the 4-bit system, you will be able to represent -8, -7, ..., 0, 1, 2, 3, … all the way up to +7.
Similarly if you have 8 bits, you will be able to represent -128, up to +127.

Please note that in a 4-bit system, you will not be able to represent the value +8.
Similarly in an 8-bit system, you will not be able to represent the value +128.
This will become clearer as we proceed.

4.21
In summary, if we need to represent signed numbers in the 2's complement system, the first thing we want to check is whether the number is zero or positive.
If it is the case, then we know that the way we represent it is exactly the same as the sign-magnitude system, so all we need to do is to first obtain the magnitude in binary and then we will just put a sign bit of "0" in front of the MSB, if the MSB is "1".
Remember for a positive value, the MSB must be "0".
Now on the other hand, if the number is negative, then the first thing we would do is to obtain the magnitude in binary, and then we will need to do a 2's complement on that binary number, and the last thing we would do would be to put a sign bit of "1" in front of this MSB, if the MSB is "0".
Now also remember that in a signed number, a negative number must have "1" in the MSB.

4.22
Let's look at 2 examples.
We are given the following signed decimal numbers, and we want to represent them in 2's complement.
In the case of +5, we will first obtain its binary equivalent, which is 101.
Since we know it is a positive value, it cannot have '1' as its MSB, therefore we must put a '0' in front. So +5 is 0101.
In the case of -7, we know that it is a negative value, so we will first obtain the binary equivalent of the magnitude, which is 111.
Then we have to do a 2's complement.
If you recall how we do the 2's complement by the shortcut method: we will copy the first '1', and then we will invert the remaining bits.
So this (001) will be the 2's complement of 111.
Now as we have mentioned, a negative value cannot have '0' in its MSB, therefore we must put a sign bit '1' in front to show that it is -7.

Pre-recorded lecture L5 (4.23 – 4.41)

<2's complement number system>
4.23
What we are showing here is the 2's complement number system if we are using 4 bits.
What we see on the left most column are the decimal values.
So we have all the positive values on the bottom and then we have the negative values on top.
For each of these values, we can then carry out the representation in the 2's complement format.
So for positive values, we know all we need to do is to obtain the magnitude which are all here (the middle column), and then we just need to make sure that the sign bit, which is the MSB, is a '0'.
So in the last column, we have the 2's complement representation for the corresponding decimal number.
So for example, +3 is represented as 0011; +6 is represented as 0110.
Similarly for the negative values, we will first obtain the magnitude.
So for example, if we want to represent -5, we will first obtain the magnitude of 5 which is here (the middle column) and then we do a 2's complement.
When you do a 2's complement, remember you just start from the right and copy down the bits until you reach the first '1'.
Subsequently, you invert every bit so this is what you get, and this (1011) is the 2's complement representation of -5.
If you do that for all the decimal numbers, you will obtain this table which shows us the 2's complement representation in 4 bits.
There are a few things which we would like to observe from this table:

- all the negative values must have sign bit equal to '1'
- all the positive values and zero must have sign bit equal to '0'
- The most negative value is -8 in this case
- the most positive value is +7

Notice that we are not able to represent +8 using a 4-bit system.
In other words, if you want to represent +8, you will have to use a 5-bit system.
Now the other thing that you will also take note of is, the most negative value will have this pattern: the sign bit is '1', and the remaining bits are all '0'.
The value -1 will always have an entire string of '1's.
Obviously, zero will be all '0's. The most positive value will have a sign bit of '0' and the remaining bits will be all '1's.
The other thing you can notice from this table is that if you start from the top, you notice that every time you add '1' to it, you will get to the next number, and if you add '1' to it, you will go on to the next number.
And if you go all the way to this number (1111), and you add '1', you will actually come to here (0000).
So adding a number is equivalent to moving in steps (down the table) which can be easily illustrated later on.

4.24
Now the same table can actually be generalised to an N-bit system.
A 4-bit system is quite small.
In modern day computers, it is more common to find 32-bit data or even 64-bit data.
So the same idea can be extended whether it is 32-bit system or 64-bit system.
You will find that the most negative value will always have a sign bit of ‘1’, remaining bits will be ‘0’.
-1 will be always a string of ‘1’s;
zero will be a string of ‘0’s;
the most positive value will have a sign bit of ‘0’, and the remaining bits are all ‘1’s.

4.25
By now you have noticed that there are several special bit patterns.
We have the most negative values; we have -1; we have zero and we have the most positive value.

4.26
We have started off by saying that given a decimal number, we can represent it in the 2’s complement format.
Now sometimes we have to do the reverse.
Let’s say we have a signed number already represented in the 2’s complement format, how do we determine its decimal equivalent?
If the number is a positive value, it is very straightforward.
All we need to do is perform a binary-to-decimal conversion.
If the number is negative, then what we will do is: we will first perform a 2’s complement to convert that number to its positive equivalent, and thereafter we will perform the binary-to-decimal (conversion) to get its decimal equivalent.

4.27
Let us now look at 2 examples.
We are given these binary numbers.
We are told that they are in 2’s complement format, which means that they are signed numbers.
Looking at the sign bit, we know that the first number is positive and the second number is negative.
So in the case of the positive number, it is very straightforward, all we need to do is to find the magnitude in decimal, and straightaway we can conclude that this value is +9.
Now in the case of the negative number, what we will do is: we will first perform a 2’s complement on this number, meaning that we will copy the bit from the right until we hit the first bit of ‘1’, and then we invert every subsequent bit.
So this is what we get.
Now notice that effectively, what we have done is: we have converted a negative value to its positive equivalent.
So now all we need to do is a binary-to-decimal conversion and we know that the magnitude is 13.

From there we conclude that this negative value that we started off with (10011) is actually -13.

<Observations on the 2’s complement number system>
4.28
Now we would like to make some observations on the 2’s complement number system.
If we are given an N-bit binary number, we can perform 2’s complement on it, and the end result is also N-bit.
To represent a binary number that has N significant bits in the magnitude, then we would actually need (N+1) bits.
Obviously the extra bit is the sign bit.
Now we have highlighted an exception here.
You will find that there are a few exceptions, and all these exceptions would apply to the most negative value in the system.

4.29
Sometimes there are more bits available than necessary to represent a binary number in the 2’s complement system.
In such a case, we will fill up the more significant bits with the sign bit.
So we have to remember that the sign bit for positive value is ‘0’ and ‘1’ for negative.
This action of duplicating the sign bit is known as “sign extension”.
In other words, if I have this negative value, it is a 3-bit representation.
Now if I want to represent the same value, the same negative number using 4 bits, then all I need to do is to duplicate the sign bit.
Now if I want to represent the same number using 5 bits, I would then further duplicate the sign bit.
It is the same for positive values.
This is a positive value, and if I want to represent using 4 bits, I would duplicate the sign bit; and if I want to represent it in 5 bits, I would duplicate the sign bit.
Now the other observation that we know is: a 2’s complement operation will actually change a positive number to negative, and vice versa, and there is no change in the magnitude.
Now once again there is an exception, and the exception applies to the most negative value in the system.

4.30
So as I have mentioned, the exception arises when dealing with the most negative value that can be represented given a number of bits.
For example in a 4-bit system, -8 is the most negative value.
The exception arises because we cannot represent +8 (with only 4 bits).
Similarly in a 5-bit system, we will not be able to represent +16 (it needs 6 bits at least).
In an 8-bit system, the maximum we can represent would be +127, we are not able to represent +128 (it needs 9 bits at least).
So of course the same statement can be generalised to an N-bit system.

4.31
So why do we want to use the 2’s complement system?
If you consider the sign-magnitude system, I’m sure you agree that the sign-magnitude system is easier, so why do we want to use the 2’s complement system?
Now the main reason would be that the 2’s complement system representation allows us to perform subtraction in the same way as an addition.
So what is the significance?
The significance of this statement is: if we are able to build a digital circuit for carrying out addition, we can use the same set of circuit to perform subtraction.
Now the advantage is very obvious, because it means that you do not need to build 2 separate circuits: one of them just to cater to addition, and the other to cater to subtraction.
You can actually build 1 single circuit and use it for both addition and subtraction.
Building one circuit is certainly cheaper than building two circuits; and one circuit will also occupy less space than two circuits.

<2’s complement addition and subtraction>

4.32
Here we will consider the addition of a pair of numbers that are already represented in the 2’s complement system.
So we have to bear in mind that all the numbers that we are looking at are signed numbers.
If the sign bit is ‘0’, that means they are positive values; if the sign bit is ‘1’, that means they are negative values.
The first example shows us that 2 positive numbers can be added easily.
You can visualize that if you have 5 full adders, the addition can be easily carried out.
In the addition of numbers, we must make sure that the sign bit is properly aligned.
All the sign bits must be properly aligned in the same position as we add up the numbers.
So in this case, adding +10 with +3 gives us +13.
Notice that the sign bits are actually added in the same manner as the remaining bits.

4.33
In this case, we are adding a positive number, which is +10, and we are adding it to -3.
We will proceed to add from the LSB and then we proceed to add all the way up to the sign bit.
The sign bits are added together just like the other bits.
And in this case, there is a carry out, but we are to ignore the carry.
In other words, when we perform addition, we will only look at the answer which is of the same number of bits as the numbers we are adding together.
So in this case, we are adding 2 numbers that are 5 bits each.
So in the result, we will also only look at 5 bits.
Any additional carry out must be ignored.

4.34
In this case, we are adding -10 with +3, and the important thing is we align the sign bits, and the result is -7.

4.35
The final example shows that we can add 2 negative numbers together.
Once again all we need to do is to make sure that the sign bits are properly aligned, and we will also ignore any carry out.
Now what these examples have shown us is that: the same addition procedure can be applied regardless of the values that we are adding.
Whether we are adding positive numbers or negative numbers, or whether which one is larger or smaller, makes no difference.
In other words, when we design a circuit that adds numbers together, and the numbers are represented in 2's complement format, we do not need to worry whether we are adding positive values or negative values together.
The circuit will perform in exactly the same manner.

4.36
What about subtraction?
We mentioned that the 2's complement system has an advantage, in a sense that subtraction can be carried out in the same way as addition.
If we look at the situation, let's say A and B are two binary values, and we want to carry out the subtraction, A - B.
Now A - B can be converted to A + (-B).
But what is -B?
In the 2's complement representation, -B is simply the 2's complement of B.
This is true regardless whether B is positive or negative.
In other words, it does not matter whether A or B is positive or negative.
Whenever we have to perform A - B, we will always perform it as A + (-B), or A + (2's complement of B).

4.37
Let us look at these examples.
In this case, we want to perform 10 - 3, and we convert it to 10 + (-3).
What is -3? -3 is simply the 2's complement of 3.
So what you see here, this is +10, and this is -3.
So 10 - 3 can be performed as 10 + (-3), and this will be the result, which of course we know is +7.
Remember that we have to ignore the carry out.

In the second example, we want to perform -10 - (-3), so we will perform -10 + (2's complement of -3).
2's complement of -3 of course is +3.
So what we have here is, we have -10, and we have +3, and this is the result.
Notice that the sign bit is '1', so it is a negative result.
The answer is -7.

<Arithmetic overflow>
4.38
This is a reminder that whenever we want to add a pair of numbers, the two numbers should have the same size, that is, the same number of bits.
So a 4-bit number will add with another 4-bit number, and the answer will still be 4-bit.
An 8-bit number will add with another 8-bit number, and the answer is also 8-bit.
Similarly, whether it is 16-bit or 32-bit, the pair of numbers must have the same number of bits, and the result must also have the same number of bits.

4.39
On this diagram, we will be able to illustrate how addition and subtraction actually takes place in the 2's complement system.
In this case, we have a 4-bit system.
Therefore you can see that the values that we are able to represent would be from -8 up to +7.
We arrange the numbers in a circular manner, so that it will be easy for you to see that if we want to perform any addition of positive numbers, we are simply moving in steps in the clockwise direction.
If we are subtracting positive numbers, we are simply moving in the anticlockwise direction.
<Animation>
For example, 1 + 2 is 3, so you can see the arrows moving. 3 + 4 will give us 7. Now it will move anticlockwise because it is a subtraction.
Now you take note of the next scenario, -4 minus 5, gives us the answer +7.
Why is this so? Why would -4 minus 5 actually give us an answer of +7?
Now we all know that -4 minus 5 is actually -9.
But remember that this is a 4-bit system.
In a 4-bit system, it can only represent the range of values -8 to +7.
-9 is a result that cannot be represented in 4 bits.
This is a situation that we want to take note of whenever we perform addition or subtraction.
In other words, whenever we add a pair of numbers or subtract numbers together, there is a potential that the result cannot be represented in that system.
This kind of situation is called arithmetic overflow.

4.40
Arithmetic overflow happens when we perform an arithmetic operation between 2 numbers and each number is N-bit, but then we produce a result that can no longer be represented in N bits.
Now we illustrate the situation of arithmetic overflow using a number line.
On the left, we have the most negative value that we can represent.
On the right, we have the most positive value we can represent.
Let's say we start off with a negative value, and we try to add a positive value to it, notice that the result will always be within the range.
However, if we have a positive value, and we try to add another positive value to it, we may have a situation where the result exceeds the range that can be represented.

So in this case, we say that arithmetic overflow has happened.
Now the above is the scenario for adding positive values.
Now for subtraction, it is also possible to have arithmetic overflow.
If we start off with a positive value, and we subtract a positive value from it, effectively, this is what happens (moving towards the left).
Now you find that the result will always be within the range.
Therefore, starting with a positive value and subtracting a positive value will never result in arithmetic overflow.
However, if we have a negative value, and we subtract it further (move towards the left), this may potentially exceed the range that can be represented.
Now in this case, overflow has happened.
Whenever we use a circuit to perform arithmetic addition or subtraction for us, we need to be aware that arithmetic overflow may happen, and we must be able to detect it if it happens.

4.41
These are the rules that we use to detect arithmetic overflow.
If the 2 numbers that we are adding have opposite signs, that is if we add a positive value with a negative value, overflow will never happen.
Now however, if we are adding a pair of values of the same sign, for example, positive add with positive, or negative add with negative, then there is a potential of arithmetic overflow.
How do we detect it? We look at the sign bit of the result.
If we are adding a positive value with another positive value, and the result is negative, then that means arithmetic overflow has happened.
Similarly, if we add a negative value with another negative value, yet the result is positive, then we know that arithmetic overflow has happened.
If you can understand the rules (of overflow detection) for addition, then you will find that in the case of subtraction, the reverse is true.
Meaning that, if you subtract 2 numbers of the same sign (equivalent to adding 2 numbers of opposite signs), you will never have overflow.
But if you subtract 2 numbers of opposite signs (equivalent to adding 2 numbers of the same sign), then there may be an overflow.
The way to detect it is to look at the sign bit.
If you look at this example, we are trying to add 2 numbers, which are positive, we can see that they are positive from the sign bit and yet upon adding up, we see that the sign bit of the result is '1'.
Meaning that we have added 2 positive numbers, but it gives us a negative result.
Obviously, arithmetic overflow has happened.

Pre-recorded lecture L6 (4.42 – 4.58)

<Circuit to add and subtract>
4.42
We have earlier said that one main advantage of using the 2's complement representation for signed numbers is that addition and subtraction can be carried out in the same manner, meaning that the circuit is the same, whether we are performing addition or subtraction.
Here we will look at such a combined circuit.
If you look at the circuit on figure 6.12, you will find that there are 4 full adders.
It will be able to perform the addition of two 4-bit numbers, X and Y.
It will also be able to perform the subtraction, X minus Y.
We will look at both scenarios, and the first scenario that we will look at is the addition.
In the case of addition, what we need to do is to set the signal, Add/Sub, to be 0.
Notice that Add/Sub is connected to the carry input.
Therefore the carry input C0 is also 0.
Now on the same circuit you will also notice that there are 4 XOR gates and their purpose is to invert Y when we perform subtraction.
Therefore in the case of addition, they will not be inverting the Y inputs.
Once the inputs X and Y and the carry in are provided to the full adders, the full adders will proceed with the addition.
In this example, we will be looking at adding the number 1(dec) with 3(dec), which we know that will get the result 4(dec).

4.43
If you look at this circuit (Fig 6-12), this is the Add/Sub signal.
Notice that it is connected to C_0, and it should be connected to logic 0 when we are performing addition.
These are the 4 full adders (FA).
The inputs and output with the lowest subscript (x_0, y_0, s_0), these are the least significant bits.
The ones with the highest subscripts (x_3, y_3, s_3), are the most significant bits.
These are the 4 XOR gates, they are drawn here using the IEEE logic symbol.
As I have mentioned, in the case of addition, the XOR gate does not do anything.
In other words, the signal Y will simply flow through the XOR gate, and appear at the input of the full adder.
So the full adder will simply perform X + Y.
We will now look at the animation.
<Animation>
Notice the carry propagation at each full adder (i.e. the time delay incurred when the carry output of one FA goes into the carry input of the next FA).
As you can see, this signal (Add/Sub) is set to 0 because we are performing an addition.
Therefore there is a 0 bit in the carry input (C_0), and also because of these 0s at the XOR gate inputs, the signal Y will not be inverted.
In fact it will just go through directly to the input of the full adder.

Therefore, the circuit simply performs X + Y, and the answer is available at the sum outputs (S_3, S_2, S_1, S_0).

4.44
Now we will proceed to show how the same circuit can be used to perform subtraction.
We recall when we perform the subtraction X minus Y, it is effectively the same as X + (-Y), and if you recall, -Y is actually the 2’s complement of Y.
How do we obtain the 2's complement of Y?
We will first invert every bit of Y (1’s complement of Y) and then we add a 1 to it.
(Refer to slide 4.16)
By using the XOR gates, we will then be able to invert Y and then we make use of the carry input (C_0) into the circuit to add a 1.
Effectively, when we want to perform 1 - 4, we are actually doing 1 + 2's complement of 4, and of course we expect the result to be -3.

4.45
We will now show how the subtraction is being carried out in this circuit.
<Animation>
Notice how the Y bits are being inverted (by the XOR gates) before entering the full adders.
At the end of the subtraction, the result that we obtain (S_3, S_2, S_1, S_0 = 1101) is the 2’s complement representation of -3.

<Addition with registers>
4.46
We have seen how a circuit can be used to perform both addition and subtraction.
Now we would like to look at adder circuits in a more realistic scenario.
For example, in an electronic calculator, a typical situation would be for us to key in the first number and then we will key in the second number and then we will add the two numbers together.
Now on a typical device like this, we would expect the values that we have keyed in to be temporarily stored in some memory locations.
Therefore in a typical digital circuit, there would be a necessity to provide some memory storage to store the values before they are added.
In digital circuits, such storage elements are known as registers.
A register is usually a memory storage for storing multiple bits.
You find that in this example, there will be timing signals at different instances.
For example at (time instances) t1, t2 or t3, different timing signals will be used to control the events happening in the circuit.
For example, at t1, it will clear the content of register A; at t2, it will load the first value to be added; and then at t3, it will actually transfer a temporary value into A register.

4.47
Then at t4, it will load the second value.
Finally after the addition is completed, at t5, the result will be stored into the A register.

Now we'd like to once again highlight that there is carry propagation for addition.
Therefore, there must be sufficient time allowed for the full adders to carry out the addition before we store the final result.
Otherwise, the result may not be correct.
In this example, we will be looking at adding these two values, which is 1(dec) + 2(dec), and the final result, which is 3(dec) will be stored in A register.

4.48
If you look at this circuit, once again, we can recognise the full adders, there are 4 of them.
They will be responsible for adding the (pair of) 4 bits together.
Here we have a 4-bit memory called B register, which will store the values that we are going to add.
We have another 4-bit memory which is called A register, which will store the intermediate result as well as the final result.
In this scenario, we expect the values that we are going to add are already available somewhere else in the system memory.
Therefore, we will be fetching the values from the system memory and loading them into the registers here before the addition.
<Animation>
(At time t1 the content of A register is cleared to zero)
(At time t2) The first value (0001) is being loaded (from memory), and added to 0.
Notice the carry propagation. Now X + 0 is X.
(At time t3) So now the value X (0001) is being stored in A register.
(At time t4) Now the second value Y (0010) is now transferred into register B, and now X and Y are being added together.
Once again, notice the carry propagation.
(At time t5) Eventually when the result of X + Y (0011) is ready, it will be transferred and stored into A register.
That completes the whole process of adding X + Y.

4.49
We have noticed the effect of carry propagation.
Obviously the more full adders there are, the longer would be the carry propagation.
Therefore in very large circuits, the speed of addition could be affected.
To overcome such a problem, there is a special purpose circuit, which is known as the carry-look-ahead circuit.
This helps to produce the carry bit in a shorter time and therefore this can help to speed up the entire process of addition.
Many addition circuits actually have such built-in carry-look-ahead circuits.

<Binary multiplication>
4.50
So far we have been looking at addition and subtraction, now we would like to talk a bit about multiplication and division.

If we have to do binary multiplication, the way we carry it out is similar to decimal multiplication.
Look at this example, we are multiplying 7 with 12.
In this case, we are actually looking at unsigned multiplication.
Therefore, we are only looking at multiplying the values, without having to pay attention to the sign.
<Animation>
You find that the entire multiplication process is actually a series of addition.
In this case, when we multiply 7 with 12, we begin with the LSB, and we produce the first result (0000).
By the time we multiply with the next bit, we actually have to shift it by one bit position to the left.
Now this is similar to what we would have done in decimal multiplication.
However, I'd like to point out that in decimal multiplication, when we shift left by one position, or one digit, it is multiplying by 10 (decimal).
But in the binary system, when we shift left by one bit position, effectively we are multiplying by 2 (decimal).
So in this case, we have 4 intermediate products where we will add together.
What we have shown here is we are adding the 4 intermediate values in one go.
In a more realistic situation, where we have digital circuits to perform the addition, typically what will happen is the first 2 numbers will be added to give an intermediate result.
That intermediate result will be added to the next value, and the new intermediate result will then be added with the next value and so on and so forth.
In other words, the circuit will only be adding a pair of numbers at any time, it will not accumulate 4 or 5 numbers and add them in one go.

4.51
Now we will look at 2's complement multiplication.
When we have to multiply signed numbers, there are different ways of handling it.
One way of course is to first convert the signed number into unsigned, and then we just carry out the (unsigned) multiplication as before.
Finally, we convert the result into signed representation.
There is an alternative way that we would like to illustrate here.
Depending on whether the multiplier is positive or negative, we do it in a slightly different manner.
If the multiplier is positive, then we will just multiply the 2 numbers the same way as unsigned multiplication.
If the multiplier is negative, then we would like to take care of the negative value in two parts.
We will treat this number, this negative value, as a sum of two parts.
The first part would be what we call the negative part, and the other part would be the positive part.
For example, if we look at the value -3(dec), it is 1101 in 2's complement representation.

This effectively is made up of 2 parts, which is the negative part, which is actually -8, and the positive part, which is +5.
(1100 = 1000 + 0101 or -3 = -8 + 5)
So in other words, if we want to multiply a value by -3, we will treat it as:
We will multiply that value by -8, and then we will add that result with multiplying the value by +5.
(value x (-3) = value x (-8 + 5) = value x (-8) + value x (+5))

4.52
We will look at this example.
<Animation>
We begin by multiplying the value on top (1011), which we call the multiplicand, with the positive part of the multiplier (0101).
Therefore, this is what we get.
Notice that we have to do sign extension to make sure that we preserve the sign.
In the last step, when we multiply by this value (1000), we have to note that this is actually a negative value, therefore there will be a change in sign.
In 2's complement representation, to change the sign, we can easily perform 2's complement.
Therefore from this (1011) to this (0101), it is actually 2's complement. Then we also make sure that we do sign extension.
Eventually, we just need to add up all the values to give us the final product (0000 1111).

4.53
The explanation (for the steps in 4.52) is provided here.
Essentially, we are treating the negative value, in this case, -3, as (-8 + 5).
Over here when we perform a multiplication with -8 (1000), we need to take care of the sign change, and therefore we do a 2's complement.
Over here, we are multiplying with a positive value (0101), there is no change in sign.
Therefore it is simply done by shift and sign extension.
Generally when we need to multiply two numbers together, if the two numbers are of m bits each, then we would expect the result to be contained in 2m bits.
In other words, the result of multiplying 2 numbers that are m bits each, will not exceed 2m bits.
Therefore the result will be able to be contained using 2m bits.
Therefore, 4 bits multiplied by 4 bits, the answer is definitely within 8 bits.

4.54
Now we have a quick word on binary division.
If we have to do unsigned division, the way it is done is similar to what we would do by what we call the long division method in decimal arithmetic.
<Animation>
In this case we perform 9 divide by 3.
Effectively, it is a series of shifts and subtraction.

Obviously, by now we know that subtraction can be easily carried out using addition, especially if the numbers are represented in 2’s complement.

4.55
Now in the event that there is a signed division, the most straightforward method would be to first convert the signed numbers to unsigned, perform the division as before, and then eventually convert the final result using the correct sign representation.

<BCD addition>
4.56
We would like to talk about BCD addition.
We have previously mentioned that BCD, or Binary Coded Decimal, is not suitable for arithmetic operations because we know that in BCD, we use 4 bits to represent a digit.
Using 4 bits, we have 16 possible symbols.
But in BCD we are only using the digits 0 to 9, therefore out of the 16 symbols, we are only using 10 symbols and there will be 6 illegal codes.
(These 4-bit codes 1010, 1011, 1100, 1101, 1110 and 1111 are not BCD)
When we use BCD, if we do have to add a pair of BCD digits together, if the result exceeds 9, it will go into one of the illegal codes.
Now when that happens, we will have to perform a correction, and the correction is carried out by adding the value 6(dec) because we want to skip over the 6 invalid codes.

4.57
The correction involves two steps.
The first step is to make sure that a carry of 1 is brought over to the more significant digit and added to it.
The second step would be adding 6 to the less significant digit.
We will look at an example to illustrate this.

4.58
In this example, we want to add 24(dec) with 47(dec), and of course we already know that the result should be 71(dec).
But in the process of addition, you will find that the digit “4” plus the digit “7” will result in a digit (1011 in binary) that will exceed 9.
Therefore, a correction will be needed.
<Animation>
In this case, because 4 plus 7 exceeds 9, we need to perform a correction.
The correction is done by adding 6 (0110bin) to it (1011bin), and this will be the result (0001bin).
At the same time, we must carry forward “1” to add to the more significant digits, which gives us 7 (2+4+1=7).

Note that for every pair of BCD digits that we are adding together, we must check whether the result has exceeded 9 and therefore needs correction.

For example, if the result of this addition (0010 + 0100 + 1) had exceeded 9, then it would need to be corrected by adding 6 to it.

Pre-recorded lecture L7 (5.1 – 5.23)

<Minterms and maxterms>

5.1
In this topic, we will be looking at combinational logic circuits.
This is the most common type of digital circuits, it is made up of a combination of logic gates, such as AND, OR and NOT.
At any point in time, the output of such a circuit will only depend on the logic inputs.
For example, if there are N inputs in the circuit, then the output at any point in time will only depend on the logic values of all the inputs at that same point in time.
The significance is that it means that combinational logic circuits have no memory.
This is entirely different from another class of circuits, which are known as sequential circuits.
Therefore in comparison, combinational logic circuits are easy to analyse and design, as opposed to sequential circuits.

5.2
So how do we proceed to design and implement a combinational logic circuit?
Usually, the function of a required logic circuit is described in a truth table.
Based on the truth table, we will obtain the Boolean expression for each output that we need to design.
Once we have the Boolean expression, we will be able to implement it, using different types of logic gates.
So now the question is, from a truth table, how do we obtain the Boolean expression?

5.3
A Boolean expression is also known as a Boolean equation, or sometimes we call it a logic function.
It can fully describe a logic circuit's output in response to every possible input condition.
For very simple circuits, we can obtain the Boolean expression easily by observation.
If you recall, when we were looking at the full adder (slide 4.6), that was exactly what we did.
It was from observation that we obtained the Boolean expression for the sum and the carry output of a full adder.
However, we may encounter more complex circuits, and in such cases, it may not be possible to just observe and obtain the expression.
Therefore what we want to do is to be able to obtain the expression in a systematic manner.

5.4
There are different forms of Boolean expressions that we can write.
In fact there are 2 broad classes.
The first class is called canonical form, and the other class is called the standard form.
We will first talk about the canonical form.
In the canonical form, there are 2 subtypes.

The first type is what we call the sum of minterms expression; and the other type is what we call the product of maxterms expression.

5.5
There are some terminologies that we first need to understand.
Minterms are all possible combinations of a given set of Boolean variables formed by the AND operation.
On the other hand, maxterms are all possible combinations of a given set of Boolean variables that this time, are formed by the OR operation.
So now you want to have the impression that minterms are formed by AND, and maxterms are formed by OR.

5.6
This table will show us what we mean by minterms and maxterms.
This circuit has 2 inputs, X and Y.
We are very familiar with this part of the truth table.
Now in this truth table, there is no output.
We are only looking at the inputs, because we want to first understand how to write minterms and maxterms from the inputs.
Let us first look at the minterms.
Remember that minterms are formed by the AND operation, which you can see from here.
So in the first case, X is 0, Y is 0 and we will write the minterm in this manner: X' AND Y'.
Now why do we want to write X'•Y'?
If you substitute the values X=0 and Y=0 into this minterm (X'•Y'), you find that it will return you a value of 1 (X'•Y'=0'•0'=1).
Now that is what a minterm is.
It is written in such a way that it will return the value 1.
Now if this is the input, if X=0 and Y=1, then this (X'•Y) will be the corresponding minterm.
Substituting the values of X and Y (X=0, Y=1) into this minterm (X'•Y) will return you a 1 (X'•Y=0'•1=1).
Same thing for this, if you substitute these values (X=1, Y=0) into this expression (X•Y'), you will get a 1 (X•Y'=1•0'=1).
Finally, if X and Y are both 1, all you need to do is AND them together (X•Y), and you'll get 1 (X•Y= 1•1 =1).
These are the corresponding minterms.
So what does a minterm actually describe?
Each of these minterms describes a specific input condition.
A unique input condition will have a corresponding unique minterm.
Therefore, when we have 4 different input combinations we will have 4 different minterms.
Sometimes we use a short form to represent the minterms, and we write them in this manner, and we call them m0, m1, m2 and m3.

The values 0, 1, 2 and 3 are simply obtained by interpreting these (X, Y in binary) numbers as decimal.
So if you convert this number (10 in binary) to decimal, it is 2.
If you convert this number (11 in binary) to decimal, it is 3.
Therefore we call it minterm number 2 (i.e. m2), minterm number 3 (i.e. m3).
That is for the minterms.

Now we will look at the maxterms.
Remember the maxterms are written using the OR operator.
If X is 0, Y is 0, and we pluck in the values into this expression (X+Y), we will get 0.
0 + 0 = 0.
So that is the way we write a maxterm.
Maxterms are written in such a way that we get a 0.
So if X is 0, Y is 1, then we pluck into this maxterm (X+Y’), we will get 0.
If X is 1 and Y is 0, the maxterm (X’+Y) will return us 0.
If X and Y are both 1, and we are inverting them (X’+Y’), then we will get 0.
So maxterms are written in such a way that we will get a 0.
Similar to the minterms, every input combination that we see here has a corresponding maxterm.
Since there are 4 different input combinations, we will have 4 corresponding maxterms.
We can also write them using short forms.
The numbering system is the same as the minterms’.
However, we will use the lower case ‘m’ to represent a minterm; and we use the uppercase “M” to represent a maxterm.
The purpose of writing minterms and maxterms is such that they uniquely describe the input combination.

<Canonical expressions>
5.7
Now we’ll look at a logic circuit with 3 inputs, X, Y and Z.
Now since there are 3 inputs, of course we expect to have 8 different input combinations.
Therefore we will have 8 corresponding minterms, as well as 8 corresponding maxterms.
If you remember how a minterm is supposed to be written: minterms are supposed to be written with AND gate, and they are supposed to be written to give you a 1.
Take an example, let’s say we take this example, X = 0, Y and Z = 1, then the corresponding minterm would be this (X’YZ).
So the minterm (X’YZ) will return us a 1 when we substitute the values of X, Y, Z (0,1,1) into the minterm (X’YZ=0’•1•1=1).
Similarly, if we take another example, X = 1, Y = 0, Z = 1, the corresponding maxterm (X’+Y+Z’) would be written in such a way that when you pluck in the values of X, Y and Z (1,0,1), it will return a value of 0 (X’+Y+Z’=1’+0+1’=0).
Similarly, we can make use of the short form to write minterms as well as maxterms.

5.8
So for any circuit that has N number of inputs, there will be 2^N minterms.
Let us look at this example where we have 4 inputs named a, b, c and d.
If you look at the number 13 in decimal, if you convert it to binary, it will be 1101.
Therefore, minterm 13 will represent the situation where inputs a=1, b=1, c=0, d=1.
If you pluck in those values into this expression (abc'd), the minterm will return a value of 1.
Now on the other hand, the corresponding maxterm is written as M13, maxterm number 13.
If you pluck in the values of a=1, b=1, c=0, d=1 into (a' + b' + c + d'), you will have 1' + 1' + 0 + 1' which will return a value of 0.
So this is the corresponding maxterm, which is M13.
If we take another example, we take 2 in decimal.
2 (decimal) is 0010 in binary.
That means this is now representing the situation where a=0, b=0, c=1, and d=0.
Therefore minterm m2 will be written in this format (m2 = a' b' c d').
If you pluck in these values, minterm m2 will return a value of 1.
On the other hand, maxterm M2 will return a value of 0 if we pluck in this set of values (a=0, b=0, c=1, and d=0).

5.9
To summarise, minterms are written in such a way that the corresponding minterm will give a logic 1.
Now you will notice that at any point in time, only one of the input conditions can happen, and therefore only one minterm can return the value of 1.
The other minterms will return a value of 0.
Look at the example, if the 3 inputs are such that x is 0, y is 1, and z is 1, then the corresponding minterm is called m3 and it is written in this manner (m3 = x'yz).
The minterm m3 will return you a value of 1 (when x=0, y=1, z=1).

Just now I have mentioned that when we name the minterm number, we are actually arranging the input variables and interpreting it as a decimal number, in order to denote the minterm number.
For example, if I arrange the variables in the order of x, y, z, then 011 = 3 in decimal.
Therefore we call it minterm m3.
However, if you decide to rearrange the variables in this order (z, y, x), without changing their values, so you have 110, then this becomes minterm number 6.
So do be careful whenever you use a short form to denote a minterm.
It is important to specify the ordering of the variables.
Whether you are using this ordering (x,y,z), or whether you are using this ordering (z,y,x), you need to specify clearly if you want to use the minterm number or the maxterm number.

5.10
The corresponding maxterms are written in such a way that they will yield a value of 0.
So if you look at these values, x=1, y=0, z=1, the corresponding maxterm will be written in this manner (x'+y+z'), and it would be called M5.
If you substitute the values (x=1, y=0, z=1), you will get 0 in the maxterm.
Once again when we use a maxterm number we must specify the ordering of the variables.
Are we reading the value in the order of x,y,z? Or are we reading it in a different order?

5.11
We have learnt about minterms, and we have learnt about maxterms.
The reason why we want to learn minterms and maxterms is so that we are able to write Boolean expressions from truth tables.
From a truth table, we will be able to write 2 types of Boolean expressions.
The first type is called the sum of minterms expression, the second type would be called the product of maxterms expression.
We will first look at the sum-of-minterms expression.
For each combination of the input variables that produces a logic 1 in the output, we will collect the corresponding minterms and OR them together.
This is the method we will use to write the sum of minterms expression.

5.12
From the same truth table, we are also able to write the product of maxterms expression.
The method is slightly different.
We will now look for each combination of the inputs that will produce a 0 in the output, and we collect the corresponding maxterm and we AND them together.
Conversion between the two forms is very easy.
Now why do we have two forms of expression?
It has to do with the association of active HIGH outputs and active Low outputs.
Sum of minterms expressions are typically used for active High outputs; product of maxterms expressions are usually used to describe active Low outputs.
We will talk more about active High and active Low later.

5.13
We will now look at an example and understand how to write the sum of minterms expression as well as the product of maxterms expression.
We have a 3-input circuit in front of us, it is presented in the form of a truth table.
There is only one output in this circuit.
Depending on which one of the input combinations, the output may sometimes be 0, or it may sometimes 1.
So now our objective is to write the sum of minterms expression for the output F and we also want to write the product of maxterms expression for the output F.

5.14
Using the given truth table, we have now inserted two columns.
The two columns that we have inserted are the minterms column and the maxterms column.
Now when we are required to obtain Boolean expressions, we do not need to insert these two columns.
These two columns are inserted here so that it is easier for you to see how the minterms or the maxterms are written.
We will proceed to write the sum of minterms expression.
To write the sum of minterms expression, what we need to do is go to the output, look for all the occasions where we see a 1 in the output and we look for the corresponding input combinations.
So there are 4 input combinations, under which the output F is to give a logic 1 in response.
Then we will proceed to collect the corresponding minterms and we will OR them together.
This is how we write the sum of minterms expression for the output F.
If you look at the minterms we have collected, this is the first one (X'Y'Z), this is the second one (X'YZ'), this is the third one (XY'Z'), and this is the fourth one (XYZ).
We can write in the short form, and remember we have to specify the ordering of the variables in order to interpret this decimal number correctly.

Now we will proceed to write the product-of-maxterms expression.
To do that, we will go to the output, and look for all the 0s, then we will collect the corresponding maxterms. Then we will AND them together.
This is the first maxterm (X+Y+Z) that we collect, this is the second one (X+Y'+Z') that we have collected, this is the third one (X'+Y+Z'), and this is the final one (X'+Y'+Z).
So it is a product of maxterms expression and we can also write it in short form.
Once again, we need to specify the order of the variables.

Now we would like to interpret the two expressions.
If you look at the sum of minterms expression, what the expression is trying to tell us is: how do I get a 1 in the output F?
To get a 1 in the output F, this is the first possibility, this is the second possibility, this is the third possibility, and this is the fourth possibility.
So in other words, out of the 8 possible input combinations, these are the 4 input combinations that will return a 1 in the output F.
Now on the other hand, you can look at the same truth table from a different point of view, and that is looking at the product of maxterms expression.
What the expression is trying to tell us is: how do I get a 0 in the output F?
If I want to get a 0 in the output F, there are once again 4 possible conditions and what are these 4 conditions?
They are described by the corresponding maxterms.
In other words, the sum-of-minterms and the product-of-maxterms expression are two views of the same circuit.
Remember that in a digital circuit, the output is either 0 or 1.

So we can always choose to describe the circuit from the output = 1 point of view, which will be the sum-of-minterms; or we can describe the same circuit from the output = 0 point of view, and that will be the product-of-maxterms expression.
So the two expressions describe the same circuit but they give different views.
It is equivalent to looking at the same coin but from two different sides.

5.15
There are other ways of writing the canonical expressions.
These are what we call the short form, so we can write it in this manner.
We use the summation sign ∑ for the sum of minterms expression.
Notice that it is important to list the order of the variables if we are using the minterm numbers.
For the product of maxterms expression, we will use the pi sign **π** to indicate that it is a product.
Once again, it is important that we list the order of the variables.
We say that conversion between the two forms is very easy.
If you notice, the minterm numbers that happen here are mutually exclusive with the maxterm numbers, which is obvious.
This is because these are the input conditions that return a 1 in the output F.
Obviously, they cannot be the input conditions that return a 0 in the output F.

<A quick glimpse of active high and active low>
5.16
Now we would like to interpret the output F with regard to the concepts of active HIGH and active LOW.
In the above example, we notice that the output F is 1 when there is an odd number of 1's among the 3 inputs X, Y, Z.
We may give F a more meaningful name, e.g. ODD, which is TRUE when it is High.
We say the logic signal ODD is active High.
On the other hand, the output F is 0 when there is an even number of 1's among the 3 inputs X, Y, Z.
We may give F a more meaningful name, e.g. EVEN*, which is TRUE when it is Low.
We say the logic signal EVEN* is active Low.
Most signals are active High,
We usually write sum-of-minterms expressions for such signals.
Active Low signals are less common,
We usually write product-of-maxterms expressions for them.
We will talk more about active high and active low later in this course

<SOP and POS>
5.17
We have looked at the canonical form of Boolean expressions, now we would like to look at the standard form of Boolean expressions.
In the standard form, there are two subtypes.
The first one is called the sum-of-products or SOP, the other is called the product-of-sum, or POS.

SOP is the simplified form from the sum of minterms; POS is the simplified form from the product of maxterms.
Simplified expressions will lead to simpler logic circuits.
So when we have a canonical expression, we will try to simplify it, and this process is known as minimisation.
The objective of minimisation is to minimise the number of logic gates that we need to use, as well as to minimise the number of inputs on each logic gate.
You'll find that these are related to the number of product terms or sum terms that we write in the expression, and also related to the number of variables that we write in the sum term or product term.

5.18
Now we are looking at the sum of minterms canonical expression.
If we simplify it by looking for the common factor xy (xyz'+xyz); and we simplify this part (x'y'z+xy'z) by looking for the common factor y'z, we are able to simplify the expression to this form.
So we have simplified a sum-of-minterms canonical expression to a sum-of-products expression.
What is the difference between a minterm and a product term?
If you look at the minterm, you find that in every minterm, the variables x, y and z must appear, either in an inverted form (x', y', z') or non-inverted form (x, y, z).
However in a product term, you find that some of the variables might not be present. So in this case (xy), z is not there.
In this case (y'z), x is not there.
So a product term does not have to contain all the input variables, but a minterm must always contain all the input variables.

5.19
Now we will begin with a product-of-maxterms expression, and we can simplify in this manner, and obtain the product-of-sums expression.
So what is the difference between the sum term and the maxterm term?
If you look at the sum term, this is called a sum term, not all the variables need to be present in the sum term.
But if you look at the maxterm, every variable must be present, either in the inverted form (x', y', z') or the non-inverted form (x, y, z).

5.20
Now we are looking at some expressions that are neither SOP nor POS.
If you look at the first expression, it may appear to be a SOP expression.
However, this term (xy)'z, if you look carefully, is not a product term.
A product term should not have any parentheses.
In the second expression, it may look like a POS.
However this term (x'+z)' is not a sum term.
A sum term should not have an inverter outside the parentheses.
If you look at the last expression, it may look like a POS, but this part of the expression (xy+z) is not a sum term.

A sum term should be formed by variables ORed together.
There should not be a mixture of AND and OR within a sum term.

Whenever possible, we will write a Boolean expression in the standard form, that is either SOP or POS.
You'll find that most of the time, we will be writing expressions in SOP form.

5.21
If we recall the process of combinational logic circuit design, we say that the behaviour of a circuit is described in the form of a truth table.
From the truth table, we will obtain the canonical expression; and from there we will want to simplify and obtain the standard expression.
So from the canonical form to the standard form, this is a simplification process, and there are different methods that can be used for simplification.
We can make use of the algebraic method, and that will require our knowledge of Boolean theorems.
We will need to make use of the single-variable theorems and the multi-variable theorems.
Alternatively, we can use some other methods such as the Karnaugh map, the Q-M method, or other Heuristic methods.
In this course, we will only focus on these two methods (algebraic & K-map).
The other 2 methods are very powerful.
However, they are usually implemented as computer algorithms.

5.22
Why do we want to simplify Boolean expressions?
Quite obviously, with a simpler expression, we will be able to build simpler circuits, meaning that we will use fewer logic gates.
Therefore, we will need to make fewer circuit connections, and that will lead to lower costs.
Also with fewer components, the failure rate is also lower, and therefore the reliability of the circuit will be improved.
Now we have previously learned the algebraic method to simplify Boolean expressions, and we also noticed that in order to do that, we need to be very familiar with the Boolean theorems.
We also need a lot of practice to improve our skills.

5.23
For example, if we have these two Boolean expressions, we can attempt to simplify them using Boolean theorems; and you can see that the simplified expression can be simply implemented.
However, algebraic simplification can sometimes be fairly confusing, especially for someone who is not experienced.
Therefore, we would like to look at the other method of simplification, which is the Karnaugh map.

Pre-recorded lecture L8 (5.24 – 5.48)

<Karnaugh maps>
5.24
Today's topic will be on the Karnaugh map.
The Karnaugh map is a graphical method that can help us simplify Boolean expressions.
Compared to the algebraic method, the Karnaugh map is much easier to use.
The Karnaugh map is based on the 2 Boolean theorems.
In AB+ AB', we can take out the common factor which is A.
Since we know that B + B' is 1, we can easily simplify it to A.
We have eliminated B in the process.
Similarly, when we have 2 terms and we can also eliminate the variable B, that will be useful for writing Product of Sums expression.
As we know, the truth table gives the value of the output X for every combination of input values.
The Karnaugh map or K-map gives exactly the same information but in a different format.

5.25
When we draw up the K-map, we need to label the squares such that the adjacent squares differ only in one variable.
Why is this important?
If we take a look at the theorems, the basis of using this theorem is because the terms AB and AB' differ in only one variable, which is B.
Therefore, we are able to eliminate it.
Likewise for this term.
Therefore in order for the Karnaugh map to work, we must label adjacent squares such that they differ in exactly one variable.
On the K-map, we can easily obtain the SOP or Sum-of-products expressions by circling all the squares that contain a 1, and then we will OR the terms together.
You find that we can also use the K-map to write POS, or Product-of-sums expressions by circling the squares that have 0's and then we AND the terms together.
You will notice that when we write SOP, we will look out for 1's in the output, which is similar to the way that we were writing the sum of minterms expression.
Similarly, when we write the product-of-sums expression, we will be looking out for outputs that are 0's, and this is consistent with the way that we have written product of maxterms expression.

5.26
Now we will show how to transform a two-input truth table into a K-map.
Usually we will not encounter a 2-input truth table where we have to simplify because a 2-input truth table is usually in a very simple form.
There is hardly any simplification required.
Nevertheless, we'd like to show you how a truth table can be easily transformed into a K-map.

On the left, we have a truth table with 2 minterms: A'B' and AB, which you can easily locate on the truth table.
Now what is going to happen next is we will transform the truth table into the K-map and we have labelled the K-map in the format of a grid.
So on this row, A=0, on this row, A=1.
In this column, B=0, and in this column, B=1.
So if you compare the truth table and the K-map, the truth table is in the form of a table whereas the K-map is in the form of a grid.
Now you will see how we can transfer the values from the truth table into the K-map.
<Animation>
As you can see, the transfer from the truth table to the K-map is very straightforward.
One of the quick checks you can make while transferring is to of course to note down the number of '1's and '0's.
This is to minimise any transfer error.

5.27
Now we will show you how to transform a 3-input truth table into a K-map.
The Boolean expression is given to us, it comprises 4 minterms, and you can easily identify the 4 minterms on the truth table.
On the right, we have the K-map, notice how the rows are being labelled.
In the 1st row, A=0, B=0.
In the 2nd row, A=0, B=1.
In the 3rd row, A=1, B=1.
Now you will notice that this is somewhat different from the usual binary sequence that we tend to use.
This is simply because we remember that on a K-map, we must make such that adjacent squares can only be different in one variable.
Therefore when we change from this row to this row, you find that only A has changed from 0 to 1, B has not changed.
This is something that you have to remember whenever you construct a K-map.
When it comes to the final row, you will find that A=1, B=0.
We should not label the K-map such that in this row, A=1, B=0. This will be wrong and it will make the K-map useless for this purpose.
You will now see how the values from the truth table are transferred into the K-map.
<Animation>
After the transfer is completed, it is always good practice to check the number of '1's or number of '0's that have been transferred.
This is to minimise any error.

5.28
Now we will show how to transform a 4-input truth table into a K-map.
On the left, we have the truth table.
The Boolean expression is given to us.
There are 4 minterms which you can easily identify on the truth table.
On the right is the K-map.

We have arranged the 4 variables A, B, C and D in this manner and once again, you can see that the K-map is laid out in the grid format.
The other thing that you also notice is the way we have labelled each row.
Notice that as we move from one row to the next, only one variable has changed.
For example, from this row, only variable B has changed.
When you move from this row to this row, only variable A has changed.
As we move from this row to this row, only variable B has changed.
In fact, if we compare the top row and the bottom row, you find that only the variable A is different.
This is the characteristic of a K-map and this characteristic must be maintained whenever we draw a K-map.
You will find the same characteristic when we label the columns.
Now you will see how the values are transferred from the truth table to the K-map.
<Animation>
Now if we do a quick check, we will find that we have transferred 4 squares of '1' to a K-map.
Although a K-map can be used for 5 variables or even up to 6 variables, it will become more and more difficult to use as the number of variables increases.
Therefore for most practical reasons, we are using 4-input K-maps.

5.29
In using K-maps, we find that a lot of times, we have to transfer truth tables with 4 inputs into a K-map.
So if we know the sequence of the transfer, it can make the transfer faster and it can also prevent any errors during the transfer.
Therefore if you arrange the K-map (inputs) in this order, where the truth table (inputs) are in the order of A, B, C, D, and we arrange the K-map in the same order, A, B, C, D, then you'll find that the decimal number for each of the entries here will correspond with the minterm or maxterm number, and they will appear in this sequence.
So if we are aware of this sequence, it will make the transfer easier and faster.

<Loops on K-map>
5.30
We have seen how to transform a truth table into a K-map, now we will proceed to see how we make use of the K-map to perform Boolean expression simplification.
The purpose of using the K-map is to loop neighbouring squares that contain '1's so that we can eliminate a variable.
Here we will begin by looking at what are the neighbouring '1's that we can loop together.
We are given this K-map with some of the squares filled with '1', and the others filled with '0'.
Our objective here is to find loops of 2 neighbouring '1's.
This is one loop that contains 2 neighbouring 1s, this is another loop.
This is another loop, and this is another loop.
After we have obtained the loops, we will need to write down the product term.
Remember when we loop '1's, we will write product terms.

How do we proceed to write the product term?
If we take a look at the green loop, we find that it covers 2 rows and 1 column.
If you look at these 2 rows, we find that what is common in these 2 rows is that A=0.
B=0 in one row, B=1 in the other row, meaning that B does not matter.
If we now look at this column, C=0, and D has to be 1.
Therefore when we write the product term for the green loop, we will write it as A'C'D.
We write A' because we know that A has to be 0.
We do not write B because B does not matter.
We write C'D because C has to be 0, D has to 1.
So if we substitute the values of A, C and D (0, 0, 1) into this product term (A'C'D), we know we will get 1, which will correspond to these '1's that we are looping.
So this is the method that we will write the product term for each of the loops on the K-map.
This is A'C'D for the green loop.
For the pink loop, A has to be 1, B has to be 1, C does not matter and D has to be 1.
Therefore the product term is ABD.
For the red loop, A has to be 1, B has to be 0, C does not matter, D has to be 0.
Therefore the product term is AB'D'.
Finally for the blue loop, A does not matter, B has to be 0, C has to be 1, D has to be 0.
Therefore we write the product term as B'CD'.
So from these examples, what you have seen is whenever we find 2 neighbouring squares that are filled with 1, we can loop them together and we will be able to eliminate one variable.
If you look at this K-map, it has 4 input variables, therefore every minterm that we have will have 4 variables.
But because we are able to loop 2 neighbouring '1's, we are able to eliminate one variable.
Therefore in all the product terms that you see, we will only find 3 variables because 1 variable has already been eliminated in each case.
Now therefore it should not be difficult to see that if we are able to form bigger loops, we will be able to eliminate more variables, and that is the objective of using K-maps.

5.31
Now we have another K-map, which is also 4-input.
This time, we will try to look for 4 neighbouring squares that contain '1' so we can loop them together.
This is one loop; this is another loop; and these 4 corners form a loop too.
If you examine the 4 corners carefully, we find that if we move from this corner to this corner, there is no difference in the variables A and B, there is only one difference and that is in variable C.
There is also no difference in the variable D.
Therefore it meets the requirement of neighbouring squares that differ in only 1 variable.
Therefore we are able to loop them together.
Similarly, if we move from this corner to this corner, you will find that the only variable that is different would be in variable A.

Thus, again it fulfils the requirement that the two squares differ in only one variable.
Therefore, we are able to loop them together.
So when we find ‘1’s in all four corners, we can loop them together as a group of 4 neighbouring ‘1’s.
After forming the loops, we will need to write down the product terms.
For the pink loop, it will simply be BD because the loop covers these 2 rows which tell us that B has to be 1, but A does not matter.
The loop also covers these two columns, which tell us that C does not matter, but D has to be 1.
For the red loop, it is simply A’B’ because the loop occupies the top row and in the top row, we need A to be 0, and B to be 0.
For the blue loop, it will be B’D’ because it occupies the 4 corners, which tell us that B has to be 0 and D has to be 0.
A and C do not matter.

5.32
Now in this case we have a K-map that has 8 neighbouring ‘1’s.
This is seldom the case.
Nevertheless, if we encounter a truth table or a K-map that has 8 neighbouring ‘1’s, we can loop them together and write the Boolean expression.
This is simply D because the loop occupies over 4 rows, meaning that A and B do not matter.
The loop covers these 2 columns, which require D to be 1, but C does not matter.
Now what you have seen is that if we are able to loop four ‘1’s together, we eliminate 2 variables.
If we are able to loop eight ‘1’s together, we eliminate 3 variables.
Therefore, the bigger the loop, the more variables are eliminated.
In other words, whenever we find loops on a K-map, we should always look for the biggest loop.

5.33
When we use K-maps for Boolean expression simplification, we must make sure that we comply with these rules.
If we want to obtain the SOP expression from a K-map, we must loop the ‘1’s and we must not loop any ‘0’s.
As we have seen, we can only loop 2 neighbouring squares, 4 neighbouring squares or 8 neighbouring squares.
So we can only loop 2^N, where N is an integer.
We cannot loop 6 neighbouring ‘1’s together, neither can we loop an odd number of neighbouring ‘1’s together.
We have also seen what we have meant by neighbours.
Neighbours are next to each other on the same row or on the same column.
Therefore, we cannot loop any 2 squares along diagonals.
If we are looping the ‘1’s to write SOP expressions, we must make sure that all the ‘1’s are looped and none of them are left behind.

In order to arrive at the most simplified Boolean expression, we should always form the biggest loops and we use the fewest loops to include all the ‘1’s.
In some cases, you find that a square may be looped more than once.
However, we do not loop a square more than once if it is not necessary.
Whenever we loop ‘1’s together, every loop will give us a product term.
Similarly, we can use a K-map to obtain POS expressions but we must remember to loop ‘0’s instead of ‘1’s.
The rules that apply to looping ‘1’s also apply to looping ‘0’s.
Every loop of ‘0’s will result in a sum term.

<Simplified expressions>
5.34
Now we will look at an example to show us how we can make use of K-maps to simplify Boolean expressions for 4-input circuits.
On this truth table, we find that the inputs are A, B, C, D and the output is Z.
So the very first step that we would do is to transform this truth table into a K-map.

5.35
This is what we get after transforming the truth table into a K-map.
Notice that the K-map must be properly labelled for each row, as well as each column.
We can also perform a quick check to make sure that we have transferred the correct number of ‘0’s and ‘1’s.
Once we are satisfied that there is no transfer error, we can then proceed to form the loops.
If we look carefully at the K-map, we find that this is the optimal way to loop all the ‘1’s without leaving any, and using the fewest loops.
We manage to find 8 neighbouring ‘1’s and we notice that there is a ‘1’ here which we can loop with this ‘1’ to form a loop of 2.
If we do not loop the 2 of them together, then we would need to write a minterm just for this square.
A minterm would contain 4 variables, but because we have looped it in this manner, we have eliminated 1 variable and this gives us a more simplified expression.
Therefore this illustrates the case where sometimes it may be of advantage to loop a square more than once.
After forming the loops, we proceed to write the product term for each of these loops.
For the big rectangle, it is simply A; because B, C and D do not matter.
For the blue loop, it will be BCD; because A does not matter.
Now on this K-map, there are only 2 loops and therefore we will only have 2 product terms.
Therefore the final step is to write the SOP expression which requires us to OR the product terms together.
So this is the Boolean expression that we have obtained for this K-map and it is in the form of SOP, sum-of-product.

5.36
Now here we will demonstrate that on the same K-map, we can also obtain the product-of-sums expression.
We do not need to change anything on the K-map.
The only thing that is different now is we must loop ‘0’s instead of ‘1’s.
So once again, we will examine the K-map carefully, and we will loop all the ‘0’s with the biggest possible loop, and the fewest number of loops, and we must not leave any ‘0’ out.
So in this case, this is the best way to loop all the ‘0’s.
Notice that all the loops we have formed in this case are loops of 4.
This means that we are able to eliminate 2 variables in each of the loops.
Now we will proceed to write the sum term that corresponds to each loop.
Remember that we are writing the product of sums expression and we are looping ‘0’s.
Therefore each term is a sum term and we must write sum terms correctly.
For the red loop, it will be A+B.
Now why is it A+B?
If we look at the loop, it occupies the top row.
In the case of the top row, A is 0 and B is 0.
We know that we are writing a sum term, and when we write a sum term, we remember the maxterm.
If you recall, when we write the maxterm, we have to write it in such a way that it gives us a 0.
Therefore the correct way to write the sum term in this case will be A+B, because when we pluck in the values of A=0, B=0, this expression will return a 0 (A+B = 0+0 = 0).
This is how a sum term is written.
The sum term for the other loops are written in similar way.
This will be for the blue loop.
This expression tells us that A is 0, D is 0; B and C do not matter.
For the green loop, it is simply A+C because A has to be 0, C has to be 0, but B and D do not matter.
Now that we have written all the sum terms, we will need to write down the final expression.
Since we are writing POS, product-of-sums, that means we will take these 3 sum terms and AND them together.
So this is the simplified POS expression in this example.

<Don’t cares>
5.37
Now we would like to consider what we would call don’t care conditions.
In some situations, certain logic circuits can be designed such that there are certain input conditions that we do not have any specified output levels.
This means that it doesn't matter whether a particular output is 0 or 1.
Now this is possible because sometimes such input conditions will not happen at all; or even in the event that such input conditions happen, the corresponding output does not matter, whether it is 0 or 1.
So we call this type of situations “Don’t care” situations.

Now when we have a "don't care" condition, what the circuit designer can do is to make a choice for the output.
It can be 0, or it can be 1, and the decision is usually based on producing the simplest output expression.

5.38
We will now look at an example that involves don't care conditions.
Here we are required to design a logic circuit whose input is a BCD digit (BCD is Binary Coded Decimal) and this circuit's output will go HIGH if the input is smaller than 6 (decimal).
We will first draw up the truth table for this circuit.
BCD inputs are made up of 4 bits; and 4 bits can represent 16 symbols.
But we also know that a BCD digit is only from 0 to 9.
In other words, if the circuit input is a BCD digit, only 0 to 9 are expected.
The rest of the inputs (1010 – 1111) will never happen.
Or even if they do happen, we are not interested in the outputs and therefore we declare them as "don't care".
So when we fill up the truth table for this circuit, we will make sure that as long as the input is smaller than 6, meaning that it is 0 to 5, then the output must be 1.
If the input is 6 to 9, then the output must be 0.
We have fulfilled the requirement of this circuit.
What happens (to the output) over here (when input is 1010 – 1111) is what we would say "don't care".

5.39
Now if we are designing a circuit that has "don't care" conditions, we should take advantage of it.
So we will transform the truth table into a K-map.
Notice that on this K-map, we have '1's, we have '0's, and we also have "don't cares" (usually marked with "X").
You will find that when there are don't cares on a K-map, it means that there are different options to loop the '1's.
If there are different options to loop the '1's, that means we will arrive at different Boolean expressions, and that means we arrive at different designs.
Each of these designs will be different in terms of the Boolean expression, and also in terms of the actual circuit connections.
However, each of these designs will fulfil the requirement that the input is a 4-bit BCD, and the output will only go to 1 if the value is 6 or smaller.
So now we will look at the first possible design.
Here we will attempt to loop the '1's and write the SOP.
So if we loops the '1's in this manner, this is the Boolean expression that we obtain.

5.40
In this case, we will loop the '1's in a different way, so we will arrive at a different Boolean expression, and so it is a different design.

We are also looping 4 squares together, and this time, take note that we have looped the “don’t cares”.
In other words, in this case, I am treating these 2 "don't cares" as ‘1’.
The corresponding expression would be (Z = A’B’ + BC’).
If you compare this expression with design 1, they are different, but both of them are made up of loops of 4, and there are exactly 2 loops.
Therefore we say that design 1 is as good as design 2 because both of them only involve 2 product terms, and in each product term, there are only 2 variables.

5.41
Now we consider a third possible design.
Once again, we have formed loops of 4, and this time, we are treating these 2 "don't cares" as ‘1’s, and the corresponding Boolean expression is this (B’C+A’C’).
This expression is different from the previous 2, therefore it constitutes a different circuit, a different design.
But it also comprises 2 product terms, and in each product term, only 2 variables.
Therefore in terms of simplification, so far design 1, 2 and 3 are as good as one another.

5.42
In this case, we will try to loop the ‘0’s, and write the POS expression.
We have looped the two ‘0’s, but we have not looped any "don't cares".
Effectively, this means that we have treated the "don't cares" as ‘1’s.
The corresponding Boolean expression is this.
Notice that in each sum term, you will find 3 variables, because these are loops of 2 squares.
We only manage to eliminate 1 variable from each of the sum terms.

5.43
In this design, we will also loop ‘0’s to write POS.
However, we will loop some of the "don't cares" to take advantage of them.
Notice that we have looped all these "don't cares", meaning that we have treated them as ‘0’s.
There is an advantage of treating them as ‘0’s and looping them because it gives us a much bigger loop.
When we have a bigger loop, we eliminate more variables and that is the objective of Boolean expression simplification.
The corresponding Boolean expression would be [Z = A’ (B’+C’)].
If you compare this with the previous POS expression, it is definitely simpler.
Therefore design 5 is better than design 4.

5.44
In summary, the way we treat "don't cares" on a K-map is similar to other squares.
They can be looped just like the other squares.

We can loop the "don't cares" with ‘1’s when we are writing SOP expressions.
Effectively, we are treating such "don't cares" as ‘1’s; the "don't cares" that we did not loop are effectively treated as ‘0’s.
Conversely, if we are writing POS, we can loop the "don't cares" with ‘0’s, and those that we have looped are treated as ‘0’s; those we have decided not to loop are treated as ‘1’s.
Notice that we are not required to loop "don't cares".
Therefore "don't cares" should only be looped if it helps to simplify a Boolean expression, and as you have seen, that is when it helps to form a bigger loop.

<Enable and disable>

5.45
In summary, if we are required to design a Combinational Logic Circuit, the following steps will typically take place.
Given the problem specifications, we will identify the inputs, outputs, and we need to derive the relationship between them.
We can represent the relationship in the form of a truth table.
From the truth table, we can obtain the Boolean expression, and that would be in the canonical form.
However, we would prefer to perform Boolean expression simplification, so that we can arrive at simpler expressions and therefore simpler circuits.
So the final expression that we obtain can be either SOP or POS, although most of the time SOP is used.
When it comes to expression simplification, we have learnt the algebraic method, we have also learnt the K-map method.
There are also other methods available but usually they are in the form of computer algorithms.
After we have designed the circuit, that is obtain the expression, we will then be able to implement the circuit, and typically it means connecting up the circuit using logic gates.
Sometimes we are not restricted to the type of gates we can use but there are occasions that we may have to be restricted to using only one particular type of gate.
For example, two-input NAND gates, which we know are universal gates.
Or similarly, we could be asked to use NOR gates only because NOR and NAND are both universal gates.

5.46
Here we would like to introduce the concept of enable and disable.
In digital circuits, after we have designed and created a circuit, obviously we expect it to function as long as there is a power supply.
However, there are certain times that we would like to be able to control the circuit, such that it will behave in one way and under other circumstances it will behave in a different way.
So the flexibility of being able to control the circuit in this manner is generally termed as enable and disable.
When we enable a circuit, what it means is the output of the circuit is allowed to change in response to the changes in the inputs.

On the other hand, if we disable or inhibit a circuit, what it means is that the circuit output is not able to change even though there are changes in the inputs.
In such a case, you find that the output of a disabled circuit will be fixed at either logic 0 or logic 1 depending on whether the output is active high or active low.
Now this may sound a little bit puzzling because we will think that whatever circuit that we have designed, we obviously want the output to respond to the inputs.
So what does it mean when we say we do not allow the output to change?

5.47
This diagram will help us to understand what it means by enable and disable.
On the left, we have 4 logic gates: AND, NAND, OR and NOR.
On the right hand side, we have the same gates.
The difference between the left and the right is on the left, the gates are said to be enabled and on the right, the gates are said to be disabled.
So let us take a closer look as to what it means to be enabled or disabled.
Let us look at the AND gate.
To enable the AND gate, the input B is connected to 1.
Now from the Boolean theorem, we have learned that anything, AND with 1, is simply that same variable itself ($x \cdot 1 = x$).
Therefore, in this case, since this is our input A, then the output will simply be A AND 1, which is A.
Effectively, what this means is by connecting B to 1, I have allowed the signal at input A to show up at the output x exactly the same way.
So this is what we mean by enable.
We are now allowing the output x to respond to the changes in the input A.
Now what if we decide to disable the gate?
If I disable the gate, I will put logic 0 to B and of course based on our understanding of Boolean theorems, and the behaviour of an AND gate, we know that the output is simply A AND 0, which will always be 0.
Therefore, no matter what happens to input A here, the output x is stuck at logic 0. This is what we mean by disable.
In other words, the output x no longer responds to any changes in input A because it has been disabled.
If you apply the same concept to the rest of the gates, you will find a similar behaviour.
Let us look at the NAND gate.
We can enable it by putting B=1, and we know that the output will be a copy of the input A. However, it will be an inverted copy ($x = A'$)
We can disable the NAND gate by connecting B to 0 and we find that regardless of what happens to the input A, the output x will be stuck at logic 1.
So if we now compare AND and NAND, we see a similarity.
That is, to enable AND or NAND, we need a logic 1 (at input B).
To disable it, we need a logic 0 (at input B).
When the AND gate is enabled, we get an exact copy of the input ($x=A$) but over at the NAND gate, we get an inverted copy ($x=A'$).
As for a disabled AND gate, the output will be stuck at 0; but for a disabled NAND, the output will be stuck at logic 1.

Now a similar analysis can be made to the OR gate as well as the NOR gate.
If you summarise everything together, you will find that these 4 different arrangements offer 4 different ways of enabling and disabling; and to also achieve either an inverted copy of the output or a non-inverted copy.

5.48
Let's look at a simple example.
This truth table describes a circuit that controls a door with child safety.
ChildSafe acts as the enable input.
When ChildSafe is 1, the circuit is enabled.
Therefore the door can be closed or opened by controlling the input Unlock.
When ChildSafe is 0, the circuit is disabled.
The door is closed regardless of the input Unlock.
You should be able to implement this circuit using an AND gate.
Note that enable/disable is a design feature.
A disabled circuit is not a faulty circuit.
You will see more examples on enable and disable later on in this course.

Pre-recorded lecture L9 (6.1 – 6.22)

<IC basics>
6.1
We will now begin with the topic on digital circuits which will enable us to have a better understanding of the circuits from the electrical point of view, as well as from the physical point of view.
This is important because building digital circuits is not a theoretical subject.
The actual circuits need to be constructed and they must be able to function correctly.
So there are certain concerns and issues, as well as characteristics of the digital circuits that we ought to be aware of.
Integrated digital circuits are typically made up of a collection of resistors, diodes and transistors.
These collections of components are fabricated on a single piece of semiconductor material, the most common type would be silicon, and that material is called a substrate, sometimes it is also called a die, or a chip.
So collectively, the entire piece of material will be able to carry out the logic functions that we have designed.

6.2
This is a picture showing us what resistors look like.
Now I'd like to point out that what we are looking at are discrete components.
Discrete components are slightly bigger sized components, but on an integrated circuit you will not see such resistors because they are too large to fit onto a semiconductor material.
However, if you are a hobbyist, and you would like to build some circuits yourself, these types of resistors are easily available and you can use them to build circuits.
These are the electrical symbols for resistors.

6.3
Now we are looking at some discrete components known as diodes.
Diodes are different from resistors in the sense that current can only flow in one direction, and you can see which direction based on the symbol of the diode.
The sharp pointing direction is where the current would flow.
Now in the case of the resistor, current can flow in either direction.

6.4
This picture shows us different types of discrete component transistors.
Transistors are probably the most important item that we need to find on digital circuits.
You will notice from this picture that all the transistors have 3 legs or 3 metal pins, so this makes them quite easily identifiable.
The role played by transistors in digital circuits is essentially a switch.

6.5
Here we will look at a very simple circuit as an example, just so that everyone is clear about how a basic circuit functions.
What we see on this diagram is a 5V battery, and we have a resistor, and we have what we call an LED.
An LED means Light-Emitting Diode, which is a type of diode; therefore current can only flow in one direction.
When there is sufficient current flowing in the diode, the diode will light up.
Here we have a switch, which can be easily implemented using a transistor.
So the function of the switch is to close this circuit or open this circuit.
In the current situation, the switch is open, therefore no current will be flowing through, and therefore the LED will not light up.
Now if we close the switch, the circuit will be closed, and current will flow from high voltage to low voltage, and then the LED will light up.
So now that the switch is closed, the entire circuit has a closed loop, current will flow from a high voltage, to the low voltage through this loop, and therefore now the LED lights up.
You can easily extend this idea to all the digital circuits, essentially the transistors that you find will be like the switches.
We will control the transistors such that they can turn on or turn off.
When they turn on, they will close a circuit.
When they turn off, they will open a circuit, and therefore they are used to control whether there is current flow or there is no current flow.
With that, we will be able to obtain high voltage or low voltage output which in turn will be used to represent Logic 1 and Logic 0.

6.6
On this table, it shows us that in the physical world, there are many physical properties that can be used to represent Logic 0 and Logic 1.
For example, we can use low fluid pressure to represent 0 and high fluid pressure to represent 1.
Or in the case of magnetic storage, we can use the "north" direction magnetic field to represent 0, and we can use the "south" magnetic field to represent 1.
Now in the case of digital circuits, the most common representation for Logic 0 and 1 would be electrical voltages.
Therefore, we will be looking at this part of the chart.
Listed here are 2 different types of logic technologies and they operate in slightly different voltage ranges but the idea behind them is similar.
Typically, low voltage will represent Logic 0, and higher voltage will represent Logic 1.

6.7
In digital circuits, the complexity of the circuit is usually measured in terms of the number of logic gates you can find in the circuit.
So if you are able to pack a large number of logic gates into one single circuit, then this is a very high scale of integration, which we can call the Giga-scale integration. At the

other end of the chart, if we are only packing fewer than 12 logic gates into say an IC chip, then we are looking at very small-scale integration.

6.8
These are some examples of the small-scale integration devices and we have used some of them in our laboratory experiments.

6.9
This is an example of a very highly integrated device, typically a microprocessor.

<TTL and CMOS>
6.10
In digital circuits, we need to represent logic 0 and logic 1 and we say that we use voltages to represent them and therefore we must have a way to switch between high and low voltages.
To achieve that, we make use of components known as transistors.
There are 2 broad classes of transistors and logic technology.
One class is known as the TTL family, and the other class is known as the CMOS family.
They are made from different types of transistors and therefore they have different behaviours.

6.11
Here we will take a quick look at the transistors.
You notice that all transistors will have 3 terminals. 2 of the terminals will be used to make the connection.
So for example I need to make a connection from this path to this path, and the third terminal will be used to control the switch.
So whether I want to open the switch or to close the switch, it will be controlled by this input.
When the switch is open, there is no current flow.
When we close the switch, there will be current flow in this manner.
So all you need to visualise is every transistor that you are able to control, if you want to turn off the switch, you will just need to control the terminal B or terminal G, depending on what type of transistor is being used.
You will be able to turn off the transistor in which case there is no current flow between the 2 terminals.
Or you can turn on the transistor, and there will be current flow between the terminals.
So this is simply the function of a transistor in digital circuits.

6.12
We will now briefly talk about some of the characteristics within the TTL family.
It is called a family because a certain logic function component can come with different prefixes and the different prefixes are to identify the difference in their electrical characteristics.
We will learn more about this characteristic later on.

So as far as the logic function or the physical pin layout is concerned, the different prefixes have no effect.

6.13
For example, if we are looking at these hex inverters, they can come in different prefixes, but all of them are hex inverters as we can identify by the part number.
But because they have different prefixes, they will have different characteristics.
For example, in this case, it would be a high speed version and in this case, it would be a low power version.

6.14
The CMOS family also has different prefixes for different characteristics.
The old series CMOS devices are not commonly used.
The HC series CMOS devices are pin-compatible with the TTL family, meaning they have the same pin layout.
For example,
This shows the pin layout of a quad 2-input NOR
The HCT series is electrically-compatible with TTL, meaning their voltage and current requirements allow us to connect the CMOS output to a TTL input, or vice versa with no problem.

6.15
These are some examples of CMOS devices so we are looking at quad 2-input NOR and they can come with different prefixes and they will have different electrical characteristics.

6.16
Since we are talking about electrical compatibility, we would like to look at the voltage range of logic devices.
For the TTL family, the power supply VCC is nominally fixed at 5V. This is the case that you have encountered in the lab experiments.
For the CMOS family, the power supply labelled as VDD has a much wider range. Now if we want to mix TTL and CMOS devices within the same circuit, then we would most likely use a 5V power supply.

6.17
Now we would like to illustrate why some CMOS devices are not (electrically) compatible with TTL devices.
What we see here on the left is a TTL device, and on the right is a CMOS device from the 74AC family.
This family is not (electrically) compatible with TTL.
We are operating both devices at the same power supply voltage, which is 5V.
However, if we look at the TTL device, the voltage range for Logic 0 is 0V-0.8V, while the voltage range for Logic 1 is 2V-5V.
On the other hand, for the CMOS device, even though it is running at the same power supply voltage, the voltage range for Logic 0 is 0V-1.5V and for 1, is 3.5V-5V. Now

remember we say that the circuit should never operate in the indeterminate range because in the indeterminate range, it is neither 0 nor 1.
So now if we visualise this case, where we send the output from a TTL device to a CMOS device, the voltage for Logic 1 coming out from the TTL is perfectly legitimate because it is within the 2V-5V range, but as it enters the CMOS, it would be in the indeterminate range which will cause problems in the circuit function.
Vice versa, if we send the output from a CMOS device into a TTL device, and the output happens to be Logic low, which is perfectly OK coming from the CMOS, but as it enters the TTL device, it will fall in the indeterminate range, and once again this will give rise to problems.
So this is what we mean when we say that the two families in this case have incompatible voltage range and therefore are not electrically compatible.

6.18
Here we would like to briefly describe what happens to inputs that are not connected to anything.
On a typical IC, there are many pins.
We will connect up the power supply and the ground pins and we will connect up whatever other pins that we need to use.
So there may be some pins that we do not need to use.
The question here is what if we leave them unconnected?
Unconnected inputs are also known as floating inputs.
In the case of the TTL family, if you leave an input floating, internal to the device, it behaves as if it is a Logic 1.
Now if you take a multimeter and measure the voltage at the unconnected pin, it will fall within the indeterminate range.
While this practise does not lead to any serious problem, it is not recommended because if a pin is left floating, it can be easily affected by other signals in the surrounding circuit.
This kind of unwanted influence is known as noise pick up.
Therefore, it is good practice to connect up any unused inputs to either VCC or ground.
Now in the case of the CMOS, it is a very different scenario.
If the input pin of an unused CMOS device is left floating, it can also pick up noise.
When it does pick up noise, it can lead to a situation where there is a huge amount of current flowing through the IC.
When that happens, the IC can become overheated, and the IC will become damaged.
Therefore, for CMOS devices, all unused input pins must be connected to either the power supply, the ground, or perhaps another input.

<Active high and active low>

6.19
Now we would like to introduce the concept of active levels.
We have mentioned these terms before, but now we will explain what they mean.
A signal is said to be active high if it performs the named function when it is Logic 1. On the other hand, we say that a signal is active low if it performs the named function when it is Logic 0.

6.20
This example will help us to understand what active high means.
We have a signal which we call OPEN, and this signal is used to open a door, and it will open the door when the logic signal OPEN=1.
Therefore, when the signal is 1, it does the named function, which is known as OPEN.
Now the question is what happens when OPEN=0?
Quite obviously, when OPEN=0, the door will not be opened.

Now we look at the second example.
We have an active low signal now, and we call it UNLOCK*.
Notice that we use the label asterisk (*) to highlight the fact that it is an active low signal.
Now this signal UNLOCK* is used to unlock a safe when UNLOCK*=0.
Therefore, the signal UNLOCK* is considered as active low, because when the signal is low, it does what it says which is to unlock.
So the same question is what happens when UNLOCK*=1?
Quite obviously, it will not unlock.
So these two examples help us to understand what it means to be active high - that means the signal will do what it says it is supposed to do when it is Logic 1.
Whereas an active low signal will do what it is supposed to do when it is Logic 0.

6.21
In many digital circuits, you may come across a mixture of active high and active low signals.
Hence, it is helpful if we can distinguish them.
Therefore, there are different conventions to distinguish between the 2 types of signals.
So on this side we have the active low convention, and here we have the active high convention.
So depending on which textbooks or which data sheets you might be looking at, they could be using different conventions.
In this course, we will use this convention (*), and that is - if it is an active low signal, we will indicate it with an asterisk (*).
Otherwise, it will not have any asterisk (*).

6.22
When we talk about active high and active low, we also need to learn about 2 other terms, which are asserted and negated.
A logic signal is said to be asserted when it is in its active level.
Otherwise, it is said to be negated.
Now when we say a signal is active high, we have to remember that it does not mean that the signal can only be high.
A logic signal can be high at any time, and it can be low at some other time. Therefore, we have to remember that logic signals will change between high and low.
If it is an active high signal, and at this moment it is in a logic high state, then we would describe it as asserted.
Now for the same signal that is active high, it may be at Logic 0 at a different instant. In such a case, we will say that it is negated.

Similarly for a logic signal which is said to be active low.
An active low signal may be low at some time, and high at other times.
When an active low signal is in the Logic low state, we say that it is asserted.
If it is in the Logic high state, we say that it is negated.

Pre-recorded lecture L10 (6.23 – 6.39)

<Circuits connections>
6.23
Now we would like to spend some time looking at logic circuit connection diagrams.
It is useful to include more information on a circuit connection diagram especially when we intend to connect up the circuit in a lab, for example.
This is because the information will help us to connect up the circuit quickly with no error, and also in the event that the circuit does not function as expected, it will be easy for us to troubleshoot and find out where the problem is.
So on a well-drawn logic circuit connection diagram, you should expect to find this information clearly indicated.
Apart from that, we also try to conform to bubble-to-bubble design rules, as well as clearly indicating the active high or active low signals.

6.24
On this diagram, we can see very clearly that two logic devices are being used.
We have a hex inverter and a quad Nand.
The pin numbers of the circuit are clearly shown.
For example, we have VCC and ground, and all the input pins and output pins are clearly indicated with the pin numbers.
If you look at the inputs, they have been labelled with the asterisk to highlight that they are active low signals.
Similarly for the outputs, they are also labelled as active low signals.
As for the bubble-to-bubble matching, what we see is a non-bubble output will connect to a non-bubble input, which is why you find that the alternate logic symbol of the inverter is used.
If you had used the standard symbol, then we would have created a situation where a bubble output is mismatched with a non-bubble input, and that is not the way to draw logic circuit diagrams.

6.25
On this diagram, it illustrates the advantage of drawing logic circuit diagrams that follow the bubble-to-bubble logic convention.
If you look at the top circuit, we have a case of mismatched output with a bubble, going into input without a bubble.
While this diagram is functionally correct, it is difficult to interpret its logic behaviour.
If you look at the diagram, Fig. 6.11(b), you find that what we have done is to replace the standard symbol of a NAND with the alternate logic symbol, and this will ensure that a bubble output is matched with a bubble input.
Similarly, over here.
The advantage is that it makes the understanding of this circuit easier.
To illustrate, let us look at the output called DATA.
This diagram (b) will help me to interpret it in this manner.
In order to get DATA to be a 1, there are 2 possibilities.
One possibility is I get a 0 here (ADATA_L=0), or I get a 0 here (BDATA_L=0).

If I were to get a 0 here (ADATA_L=0), it means I need to get a 1 here (A=1), and a 1 here (ASEL).
So what it means is, if I get a 1 here (DATA=1), it is possible because A=1 and ASEL is 1, which is indicated here (ASEL▪A).
Alternatively, it is because B is 1, and BSEL is 1.
So in other words, the output DATA is 1 either because A is 1, and I am selecting it (ASEL=1), or because B is 1, and I am selecting it (BSEL=1).
So it becomes very easy to see how this circuit is behaving.
This circuit is actually known as a 2-input multiplexer, which you will get to learn later on in this course.
As far as possible, we will try to draw logic circuit diagrams such that that bubble outputs will match with bubble inputs, and not in this manner.

6.26
Now we would like to quickly talk about troubleshooting.
Whenever we build up a digital circuit, while we can be as careful as possible, there are times that the circuit that we build may not function as expected.
Therefore, we must be able to detect the fault, and we must be able to isolate the fault, meaning knowing where the problem is, and then proceed to correct the fault.
You may refer to the supplementary lab manual, which gives you some tips as to how you can troubleshoot such digital circuits.
Sometimes the problems from the digital circuit arise because of some internal IC problems.
So these are the possible problems which can be easily solved by replacing the IC.

6.27
Sometimes it is also possible to have problems with a circuit because of some external faults.
For example, we might have made some wrong connections; or maybe we did not check whether the power supply is functioning properly; or we might have accidentally connected 2 wires together when they're not supposed to; or we have 2 wires that are supposed to be connected but then we have not connected them together.
Now here I would like to highlight the point that whenever we are connecting digital circuits, there are certain things that we must be mindful of, and that is: a circuit output should not be connected to another circuit output unless you know exactly what you are doing.
In other words, there are occasions where it is possible, permissible to connect 2 outputs together.
You will learn about that later on, but for the time being, if you do not know what you are connecting to, make sure that you do not connect 2 outputs together.
Similarly, a circuit output should be determined by the circuit input and therefore, you should never connect the output directly to VCC to make it go high, neither should you connect it to ground to make it go low, or to any other fixed voltage level.

<Transistor switching>
6.28
Now we would start to look at the transistor in a digital circuit.
As far as the transistor is concerned, we can treat it as a switch.
Essentially, when we close a switch, current will flow; but when we open a switch, there will be no current flow.
If we look at these 2 diagrams, they are identical.
The only difference is on the left, we have the bottom switch open and the top switch is closed.
In this diagram (left), we show that the input (V_{IN}) is logic 0.
On the diagram here (right), the input (V_{IN}) is actually logic high; and what happens is this effectively controls the switch such that the top switch is open and the bottom switch is closed.
Now if we recall that when a switch is closed, current flows, and when the switch is opened, current does not flow, then what we will be able to see is on the left, we will have current flowing from the 5V power supply.
We have to remember that current always flows from high voltage to low voltage.
In a circuit the power supply is the highest voltage available and therefore current will be flowing from the 5V power supply in this case.
On the other hand, if we look at the diagram on the right, because the bottom switch is closed, and the top switch is opened, we will have current flowing in this direction. Now in a digital circuit, the ground represents 0V, and it is the lowest voltage you can find in the entire circuit, therefore if there is any current flow, the current will be flowing to the ground.
So in other words, in digital circuits, current flows from the power supply and then it goes through various paths, depending on what kind of circuit we have, and then eventually the current will flow into ground.
Now what do we manage to achieve from this circuit?
On the left, because we have a current flow from the power supply, and also because the switch is closed, when the switch is closed, the resistance is very low, therefore the output voltage here (V_{OUT}) is very close to that voltage (V_{DD}).
Therefore, it will be a high voltage, and this is how we obtain a logic high output.
On the other hand, when the current is flowing to ground, in this case, and also because the switch is closed, it has a very low resistance.
Therefore, the output voltage here (V_{OUT}) would be very close to ground.
So it will be a very low voltage, and therefore this gives us a logic low output.
So on this diagram what we have seen is that effectively, we are able to achieve a simple inverter circuit.
When the input is low and we control the transistors such that when the top transistor closes, then we will have a logic high output.
On the other hand, if we have a logic high input, and it controls such that the bottom transistor conducts, then we will have a logic low output.
So effectively, this is the behaviour of all the inverters that you will encounter.

6.29
Now we show a few examples to help us to see how physical inverters can be obtained.
In this example, we will use what we call a BJT transistor, and we connect it up with two resistors.
If we put a logic 0 at the input, we will not be able to turn on the transistor and therefore there will be current flow from the top resistor and this will give a logic 1 output.

6.30
On the other hand, if we put a logic 1 input, then we will be able to turn on the transistor and then current will be flowing to ground in this manner, and therefore the output voltage is low.
This is once again how we achieve an inverter.

6.31
This is another example of an inverter circuit.
In this case it is making use of what we call the TTL technology, transistor-transistor logic.
Now the circuit is a little bit complicated, but we do not need to worry too much about that.
All we need to know is if I put a logic 0 at the input, some of the transistors will turn on, some of the transistors will turn off; but the overall effect is the top transistor turns on but the bottom transistor is turned off.
Therefore, current will flow in this manner, and that is how we get logic 1 at the output.

6.32
Conversely, if we put a logic 1 at the input, then this time we find that the transistors will turn on in a slightly different manner.
Therefore there will be a current path flowing into ground and then we have a logic 0 output.
So this is how we achieve the inverter function.

6.33
Now we will look at the CMOS inverter.
The CMOS inverter circuit is fairly easy to understand.
What we have here are two transistors.
The one on top is called the PMOS, the one below is called the NMOS
If you look at the symbols of the PMOS and the NMOS, you find that the PMOS has a bubble whereas the NMOS does not have a bubble.
Now the purpose of drawing a bubble is the same as what you have seen so far in our digital circuit drawings.
In other words, when you see a bubble, it’s telling you that you need a logic 0 to turn on (the PMOS).
So to turn on a PMOS, we need a logic 0 because the bubble says so.
On the other hand, to turn on an NMOS, we need a logic 1, because there is no bubble here.

Whenever we encounter any CMOS logic circuits, we need to remember that a bubble means we need a logic 0 to turn on (the transistor), no bubble means we need a logic 1 to turn on (the transistor).
In this case, if we make the input X=0, then obviously we know the PMOS will turn on but the NMOS will turn off.
Therefore the top transistor is turned on, and we have output Y=1.
So this is the CMOS inverter.

6.34
On the other hand, if we make input X=1, from the symbols we can tell that the bottom transistor will turn on but the top transistor will turn off.
Therefore now we have a current path to ground, output Y will be of low voltage.
This is how we achieve the CMOS inverter circuit.
You'll find that CMOS logic circuits are fairly easy to analyse, because just be looking at the bubble or non-bubble input we know very clearly whether we need a logic 1 or a logic 0 to turn it on.

From the last few examples that you have seen, you have also noticed a common behaviour.
Basically there will be two transistors, one on top and one below (the output), if the top transistor turns on, then we have a logic 1 output.
If the bottom transistor turns on, then we have a logic 0 output.
You find that in none of the situation, both the top and bottom transistors turn on.
We cannot design a circuit where both the top and bottom transistors turn on because that will not give us a valid logic output.
Therefore we have to remember, only either the top or the bottom can turn on at any point in time, but never both turn on.

<CMOS logic circuits>
6.35
Before we go on to look at some examples of CMOS logic circuits, I'd like to go through some basic concepts.
In simple circuits, you are likely to come across circuits in series, and circuits in parallel.
What do they mean?
Let me give a simple example to illustrate.
Let's say there is a river, and we want to cross the river from this side to the other side.
I may have a bridge which ends somewhere in the middle of the river, and another bridge that continues the path.
In this case, bridge A and bridge B are said to be in series with each other.
If I want to cross the river, both bridge A and bridge B must be working.
If bridge B is broken then I won't be able to cross.
Similarly, if B is working but A is not, then I also cannot cross the river.
In other words, if a circuit has series components, then every individual series component must be working, otherwise we do not have a path.

On the other hand, you may have two bridges across the same river.

In this case, we say they are parallel to each other.
If I want to cross the river, I can use either bridge C or bridge D.
If bridge C is broken, I can use bridge D, and vice-versa.
If both bridges C and D are broken, then of course I will not be able to cross.
Therefore, in a circuit that has parallel components, we need at least one of the components to be working, otherwise we do not have a path.
Using these concepts, it will be easier for us to look at some CMOS logic circuits.

6.36
The first example we want to look at.
If you look at the circuit on top, you will notice that the transistors on top are in parallel.
Why do we say they are in parallel?
If this transistor (PMOS connected to input A) turns on, then current will be able to flow (from V_{DD} to X).
Similarly, if this transistor (PMOS connected to input B) turns on current will be able to flow (from V_{DD} to X).
Now I only need one of the two (PMOS) transistors to turn on, and I will be able to get current to flow from V_{DD} to output X.
Therefore the two (PMOS) transistors are said to be in parallel.

On the other hand, if you look at the two (NMOS) transistors below, they are said to be in series.
This is because if I want current to flow from X to ground, I must make sure this transistor (NMOS connected to input A) is conducting or turned on, and at the same time, this transistor (NMOS connected to input B) must also turn on.
So now we can see very clearly the effect of putting transistors in parallel, and putting transistors in series.
If they are in parallel, either one of them needs to turn on (to form a current path).
If they are in series, then both of them need to turn on, if we want current to flow through.
Now we look at the circuit below, you find that the arrangement is somewhat different.
In this case, the two (PMOS) transistors on top are in series.
If we want current to flow from V_{DD} to output X, we must make sure both (PMOS) transistors are turned on.
On the other hand the two transistors at the bottom are in parallel.
If I want current to flow from output X to ground, then I need to make sure that at least one of them is turned on, but I don't need both of them to turn on.

6.37
With this understanding of parallel and series transistors, it should not be difficult for us to see that if I want output X=1, then I must form a current path from V_{DD} to X, that means I need to turn on this transistor (PMOS connected to A) or this transistor (PMOS connected to B).
Looking at the symbols, I need a logic 0 to turn it on.
Therefore I conclude that, either A=0 or B=0 will be able to form a current path from V_{DD} to output X.

Notice that if A=0 or B=0, then one of these two (NMOS) transistors will be turned off.
Therefore there will be no current flow from X to ground.
This meets the requirement that we have stated earlier.
Only the top or the bottom transistors can turn on at any time, they cannot be both turned on at the same time.
If we look at it from the other point of view, if I want output X=0 it means I must have a current path from X to ground.
Then I must make sure that both of these (NMOS) transistors turn on.
The logic symbols tell me that I have to make A=1 and B=1.
Therefore A=B=1 will cause X=0.
If we combine these two pieces of information together, it is not difficult for us to see that this is effectively a NAND gate.

Now we will look at the circuit at the bottom.
If you look at output X, if we want X=1 we must form a current path from V_{DD} to output X.
Therefore we must make sure that both these (PMOS) transistors turn on.
The logic symbols tell me that I need logic 0 (at A and at B) to turn on the transistors.
Therefore we have to make A=B=0 in order to turn on these two transistors.
At the same time, you find that since A=B=0, neither of the (NMOS) transistors will turn on.
Therefore there is no current path from X to ground.
This meets our requirement that only the top transistors are turned on but not the bottom ones.
Now we look at the bottom transistors.
If I need X=0, I must form a current path from X to ground.
That means either this transistor (NMOS connected to A) or that transistor (NMOS connected to B) must turn on.
The logic symbols tell me that either A=1 or B=1.
Putting the two pieces of information together, it is quite straight forward for us to tell that this is actually a NOR gate.
In a NOR gate, if you want the output to be 0, you need at least one of the inputs to be 1.

<More CMOS logic circuits>
6.38
Now we look at another CMOS logic circuit.
If you compare this with the one that we just looked at, there is a great deal of similarities.
The 3 transistors on top, Q2, Q4 and Q6 are parallel to one another.
The 3 transistors below, Q1, Q3 and Q5 are in series with one another.
If we analyse this circuit, we will say that in order for Z=1, we need a current path from V_{DD} to output Z.
That can be accomplished by turning on Q2, or Q4 or Q6; because the 3 transistors are parallel to one another.
Therefore either A=0, or B=0 or C=0 will be sufficient to cause Z=1.

On the other hand, for Z to be 0, there must be a current path from Z to ground.
This will require all the 3 transistors Q1, Q3, and Q5 to turn on.
In order to turn on all 3 of them, the inputs A, B and C must all be 1.
Therefore, combining these two pieces of information together, we can conclude that this is a 3-input NAND gate.
(Output Z=0 only when A=B=C=1 thus Z = (ABC)')

6.39
Now we look at another example.
This example may seem a little complicated compared to the first few.
Maybe it's easier to consider the circuit in 2 parts.
The first observation we would like to make is this:
Either A=0 or B=0 will cause Z=0.
How is that so?
Now let us look at A=0 or B=0.
Effectively, this means either Q2 or Q4 will turn on.
Q2 and Q4 are in parallel.
If either one of them turn on, we have a current path from V_{DD} to this point (the input of Q5 and Q6).
Therefore at this point it will be logic 1.
What is the effect of this logic 1?
If you look at this part of the circuit (right side of the blue dashes), if this (the input of Q5 and Q6) is 1, Q5 will turn on but Q6 will turn off, therefore we have a current path from Z to ground, which makes Z=0.

Now let us analyse the other part of the circuit.
It says here: A=B=1 will cause Z=1.
If A, B are both 1, Q1 and Q3 will both turn on.
When both Q1 and Q3 are turned on, we have a current path from this point to ground.
Therefore this point (the input of Q5 and Q6) will be at logic 0.
Now when this point is logic 0, it will turn on transistor Q6 but turn off transistor Q5.
We now have a current path from V_{DD} to output Z.
Therefore we get Z=1.
So this is how the circuit works.

In fact if you take a closer look, this circuit is actually made up of 2 parts.
The first stage (left side of the blue dashes) is actually a 2-input NAND.
The second stage (right side of the blue dashes) is actually a NOT gate.
In other words, to obtain an AND gate, we actually have to begin with a NAND gate and then invert the output.
This may also help you better understand when we were looking at the universal NAND gate earlier.
We said there are advantages of building circuits with purely NAND.
This is a clear indication.
It is less costly to build the NAND gate compared to the AND gate.
The NAND gate actually requires fewer transistors than the AND gate.

Pre-recorded lecture L11 (6.40 – 6.63)

<Voltage parameters>
6.40
Now we'd like to start looking at some characteristics of digital circuits.
What we have here is the transfer characteristic of a CMOS inverter.
What we have on the horizontal axis is the input voltage (V_{IN}), and over there on the vertical axis is the output voltage (V_{OUT}).
What you can recognise from this diagram is that: when input voltage is around 0V, the output voltage is around 5V.
In this case, we assume the power supply V_{DD} is 5V.
As the input voltage increases, you find that initially there is little change in the output voltage.
However, as the input voltage enters this range, you find that the output voltage has a sharp drop.
Then as the input voltage gradually goes higher and higher, the output voltage is only around 0V.
What you are able to see here is: apart from the usual characteristic that we understand of an inverter, i.e. when input is low, output is high and when input is high, output is low; we also can see what happens when the input voltage is between logic high and logic low.
This is what we mean by transfer characteristic.
What does typical mean?
Typical means it is not guaranteed, in the sense that not every single CMOS inverter you pick up will exhibit this exact behaviour.
There can be slight variations among different components.
There can be slight differences in the voltage levels.

6.41
Now we would like to see how the time-based signals would look like.
In other words, if the input voltage is changing over a period of time, how would the corresponding output voltage changes be following this transfer characteristic?
<Animation>
As you can see, in the beginning, the input of the inverter is logic 0.
If you look at this point (red dot on V_{IN} graph) it corresponds to this (red dot on transfer function), and therefore the output is logic 1 (red dot on V_{OUT} graph) which is 5V.
As time goes by, the input voltage rises and as it cross this point (yellow dot on V_{IN} graph), using the information (yellow dot) from the transfer characteristic, we will be able to obtain the corresponding output voltage (yellow dot on V_{OUT} graph) which is still quite close to 5V.
As the input voltage goes higher and approaches this point (green dot on V_{IN} graph), then we find the output voltage has dropped quite significantly (green dot on transfer characteristic) which gives rise to this green point (on V_{OUT} graph).
Eventually as the input voltage rises further and reaches this point (purple dot on V_{IN} graph), we get the output voltage (purple dot on transfer characteristic) which is very close to 0V (purple dot on V_{OUT} graph).

6.42
Now we would like to talk about some voltage parameters.
Based on our experience with digital circuits, we know that a high voltage represents logic 1 and a low voltage represents logic 0.
However, how high is high and how low is low?
In order for the circuits to work properly, whatever high voltage we want to use must meet certain requirements.
Similarly, the low voltage that we intend to use to represent logic 0 must also be within a certain specified range.
The voltage parameters we are looking at specify the correct range for the input and output voltages.
Let us look at the 4 parameters 1-by-1.
The first parameter which is labelled as V_{OH} (min), O stands for output, H stands for 1.
So V_{OH} (min) is actually the minimum output voltage produced for logic 1.
In other words, a digital circuit which is supposed to produce a logic 1 output, under correct operating conditions, it must produce an output voltage which is at least this value.
That means, it must be at least this minimum value.
In other words, if let's say VOH (min) is equal to 3V, then if the circuit is producing a logic 1 output, this output voltage will guarantee be at least 3V.
It may be 3.5V, 4V or 5V, but it cannot be lower than 3V.

Similarly we have a parameter V_{IH} (min), I stands for input.
This is the minimum input voltage to be recognised as logic 1.
In other words, if you send a logic 1 input into a circuit, you must ensure that the voltage is at least this value.
If it is lower than this value, then it may not be recognised as logic 1.
For example, if V_{IH} (min) is 2V, if you want the circuit to recognise an input as logic 1, you must give it at least 2V.
It can be higher than 2V but it cannot be lower.

The two parameters we have looked at apply to logic high outputs or inputs.
Similar parameters also exist for logic low.
If you want a certain input voltage to be recognised as logic 0, then it must not exceed V_{IL} (max).
Similarly, if a logic device produces a logic 0 output, then that output voltage must not exceed V_{OL} (max).
So a logic low voltage cannot be too high, and a logic high voltage cannot be too low.

<DC noise margin>
6.43
What is the significance of the voltage parameters that we have just talked about?
The voltage parameters are related to noise margin.
What is noise margin?
In all digital circuits, apart from the digital signals that we want, there could also be other signals that we do not want; but they exist anyway.

Signals that exist but are not part of what we want are generally known as noise.
Any unwanted signals are known as noise.
Whenever we have a logic circuit that produces a logic high output like in this case, by the time it reaches the input of another logic circuit, of course it should still be logic high.
However, what we see here is a series of noise.
That noise will be able to affect the signal coming from the output.
If the noise affects the output signal but it is still within the V_{IH} (min) then it is OK.
But if it falls below V_{IH} (min) then we will have a problem.
If you look at this part of the diagram, we have difficulty seeing the signal clearly.
The same problem will happen in the digital circuit.
If you send a signal which is supposed to be logic high into a digital circuit, but part of the signal voltage is actually lower than what is permitted for logic high, then the circuit may not be able to recognise it correctly as logic 1.
This is what we mean by noise margin:
How much noise are we able to tolerate?
The larger the noise we can tolerate, the better it is.
Therefore the higher the noise margin, the better it is.
More specifically, in the output high state, the noise margin is given by these 2 parameters: V_{OH} (min) and V_{IH} (min).
In other words, the minimum logic high voltage coming out of a device, i.e. V_{OH} (min), compared to the minimum voltage a circuit is willing to accept, i.e. V_{IH} (min), that difference will represent the amount of noise the circuit is able to tolerate.
For example, if V_{OH} (min) = 3V, and V_{IH} (min) = 2V, then the circuit has a 1V noise margin (NM = 3-2 =1).

6.44
The same analysis can be made for a logic low output.
What we have here is a circuit that produces a voltage low enough to be recognised as logic 0. i.e. not higher than V_{OL} (max).
But it is affected by noise in between.
Some of the signals become higher and higher.
By the time the signal reaches this circuit's input, its voltage must not be too high.
If it is higher than what the input expects it to be, i.e. V_{IL} (max), then it will not be recognised as logic low.
Therefore under the effect of noise, if it is still below V_{IL} (max), then there is no problem.
However, if the noise affects the signal so much so that it exceeds V_{IL} (max), then this circuit will have a problem in recognising the signal as logic 0.
Specifically we can say that for the logic low situation, the noise margin is given by these 2 parameters: V_{IL} (max) and V_{OL} (max).
Once again, a higher noise margin is better because it means that the circuit can withstand a larger amount of noise.

6.45
On this chart, we are able to see some voltage parameter values for different types of devices.

For example, there is 5-VLSTTL, 5-V CMOS, 3.3-V LVTTL and CMOS operating at slightly lower power supply voltages (2.5V, 1.8V).
What I would like you to look at are these values, which represent the noise margins.
What you can generalise from this diagram is:
If you have a higher power supply voltage, for the same type of device, e.g. TTL, it tends to give a better noise margin.
This you can also see in the case of CMOS.
The CMOS with 5V power supply will have better noise margin than a CMOS working at 1.8V power supply voltage.

The other thing we can learn from this chart is:
If we have different types of circuits working in different operating conditions, are they able to work together?
Look at this part of the diagram, we have a driving gate (D) which sends its output to a receiving gate (R).
On this table, the different types of driving gate and receiving gate are listed.
Along the diagonal, the driving gate and receiving gate are of the same type.
Therefore there is no problem in using them together.
How do we decide whether they are able to work together?
These are the criteria that we need to look at:
If D’s V_{OH} is higher than R’s V_{IH}, and D’s V_{OL} is lower than R’s V_{IL}, then the circuits can work together.
Otherwise, they will not be able to work together.
If you look at this table, you notice that it is not symmetrical along the diagonal.
For example, if D is 5CMOS and R is 5TTL, they can work together.
But if D is 5TTL and R is 5CMOS, then they cannot work together.

<Other characteristics>

6.46
We have looked at the voltage parameters, now we would like to quickly mention the current parameters.
Similar to the voltage parameters, we have current parameters pertaining to logic 1 and logic 0.
We also have current that flows into an input, and current that flows from an output.
Generally there is specification on the maximum amount of current flowing into an input or from an output.
We need to meet such current requirements in order for the circuit to work correctly.

In the case of CMOS circuits, the currents are very little, especially at the input.
For the output, depending on the CMOS family, there is a range of output current capability.

6.47
What is the significance of the current parameters?
It is actually related to another characteristic of digital ICs, known as Fan-out.

Fan-out specifies the number of standard loads, or gates, that a particular output can drive.
In other words, how many gates can be connected to a particular output?
The number of gates that can be connected depends on the amount of current available.
Therefore current parameters are directly related to fan-out.
The more gates you need to drive, the more current you need to have.
Apart from that, when we connect more gates to a certain output, it will have other implications; such as it can reduce the noise margin, which is not a good thing, because we have said earlier that a higher noise margin is better.
It can also end up reducing the switching speed.
The switching speed of an IC essentially measures the time taken for an output to switch between logic 0 and logic 1.
Obviously, the short amount of time needed the better it is, and the higher switching speed it is.

6.48
There are some other characteristics that we should be aware of:
Power dissipation.
Every electrical component will consume power.
When we have a logic gate or logic device, we connect it to a power supply and turn on the circuit, we expect it to consume power.
For CMOS circuits, most of the power is consumed when the output is switching between 0 and 1.
If you have a CMOS circuit whose output is stable, e.g. constantly staying at logic 0 (or logic 1) most of the time, then it is not consuming much power.
But if it is switching frequently between 0 and 1, it will have a much higher power consumption, which is proportional to the switching frequency, as well as the square of the power supply voltage.
With a higher power dissipation it also means the device gets heated up.
Obviously, the lower the power dissipation the better it is.

6.49
We would like to talk about propagation delay.
We have mentioned propagation delay before.
In every logic gate, whenever the input changes, the output will change correspondingly.
However, there will be a short delay before the output would change.
This average time delay for a signal to propagate from the input to the output, is called propagation delay.
Depending on the type of devices we are looking at, the propagation delay can vary.
The typical value will be in the nanosecond range.
Propagation delay can be further distinguished by specifying t_{PHL} and t_{PLH}.
t_{PHL} applies when the output is changing from high (H) to low (L).
On the other hand, t_{PLH} applies when the output changes from low (L) to high (H).
These two values may not be the same even on the same device.

Obviously, the shorter is the propagation delay, the faster is the switching speed, and the better it is.
However, we need to remember that we do not have zero propagation delay.
No matter how short the delay is, maybe as short as 9ns, it is not zero.

6.50
I would like to show how input changes will lead to a change in the output with the effect of propagation delay exaggerated so that we can see it clearly.
In normal circuits that we operate in the lab the propagation delay is very small and we may not be able to observe it.
<Animation>
What we are looking at is an inverter with its input and output.
Obviously, when an inverter's input goes low, we expect its output to go high.
That happens during this part of the timing diagram.
We also notice there is a propagation delay, $t_{PLH.}$
How is this propagation delay measured?
It is measured from the 50% point of the input transition to the 50% point of the output transition.
What does this 50% point mean?
It refers to the halfway point of the 1-to-0 transition.
Same thing here, the 50% point refers to the halfway point of the 0-to-1 transition.
In this case, because the output is changing from 0 to 1, the propagation delay is labelled as $t_{PLH.}$
On the other hand, when the input changes from low to high, we expect the output to change from high to low.
Once again, there is a propagation delay.
It is measured from the 50% point of the input transition to the 50% point of the output transition.
The 50% point refers to the halfway point of the 0-to-1 input transition, and the halfway point of the 1-to-0 output transition.
Since the output is changing from 1 to 0, the propagation delay is labelled as $t_{PHL.}$

The other parameters that we'd like you to look at are rise time (t_r) and fall time (t_f).
As the name suggests, rise time measures the time taken for a signal to rise from 0 to 1.
As this signal rises from 0 to 1, before it reaches the full range, it will first cross the 10% point, and eventually cross the 90% point.
Using these two points, we are able to measure the amount of time the output takes to rise from 10% to 90% of the final voltage. We call it the rise time (t_r).
Conversely, we can also measure the fall time, which is the time taken for a signal to fall from logic 1 to logic 0.
Once again, using this entire voltage range, we look for the 10% point and the 90% point.
We use them to quantify the amount of time the signal takes to change from high to low, and we call it the fall time (t_f).

In general, rise time and fall time should be as short as possible.

But similar to propagation delay, they are not zero.
In some cases, if a signal has very long rise time or fall time, it can actually lead to some problems.

<Tristate output>

6.51
Now we have come to the topic of tristate outputs.
So far in all the digital circuits that we have encountered, we say that the output state is either logic 0 or logic 1.
However, now we would like to introduce a third state, which is called High impedance (Hi-Z).
In this state, the output is neither 0 nor 1; effectively it behaves like an open circuit.
In order to achieve the high impedance output, we must add something to a logic device, which usually is in the form of an enable input.
In other words, a device with tristate output capability will have an enable input; such that when the device is enabled, the output state is as per normal, i.e. either 0 or 1.
But when the device is disabled, the output will be in high impedance state.
You can find tristate buffers and tristate inverters available.

6.52
We will try to understand how to achieve a tristate output.
What we are looking at is a tristate inverter.
Look at the diagram on the left (a), this inverter has an enable input.
There are two transistors, one on top and one below.
In this scenario, the inverter is enabled and the input is logic low.
The effect is: the top transistor is turned on but the bottom transistor is turned off.
Based on what we have learned, we know that current will flow from V_{DD} to the output, and that will give a logic high output.

Using the same inverter in (b) but the input is changed to high.
What happens now is: the bottom transistor is turned on but the top transistor is turned off.
We have a current path from the output to ground; so we get a logic low output.
What we have seen so far from diagram (a) and (b) is what we have encountered in ordinary inverters.
So if I have a tristate inverter which is enabled, then it behaves just like an ordinary inverter.
Note however, on an ordinary inverter you will not find an enable input.

On a tristate inverter, there is an enable input.
We can choose to disable it (diagram c).
When we disable a tristate inverter, what actually happens is: both the top and the bottom transistors are turned off.
This is true regardless of the inverter input.
In other words, simply by disabling the inverter, we are able to turn off both the top and bottom transistors.

Therefore now the output does not have a current path to V_{DD} nor to ground.
We say the output is in high impedance state.

Once again, I would like to remind you that in any logic circuit, only either the top transistor is turned on, or the bottom transistor is turned on, or both of them are turned off.
Under no circumstances should we turn on both the top and bottom transistors.

6.53
Now that we have understood how a tristate inverter works, we will look at an example of a tristate buffer.
Look at the tristate buffer logic symbol in diagram (c).
It is just a triangle with no bubble at the output because it is a buffer, not an inverter.
Since it is a tristate device, it must have an enable input.

Diagram (b) gives us the truth table of the circuit.
We can easily analyse how it works.
The circuit's inputs are: enable, and buffer input A.
The other signals are internal: B, C and D; they are labelled here to help us understand how the circuit works.
Eventually what we see will be the output (OUT), which will be high, low or H-Z.

Let us first look at the scenario when the device is disabled.
The enable input (EN) needs logic 0 to disable the device.
In other words, this enable input is active high because we need logic high to enable it.
To disable the device, we need to make the input signal EN=0.
We follow the circuit in diagram (a): EN=0, it means the input at the NAND gate is 0; therefore the output of the NAND gate, C=1.
Over here, since EN=0, the inverter output B=1; therefore the output of the NOR gate, D=0.
These are listed out for us in the truth table (b).
When EN=0, then B=1, C=1 and D=0 regardless of the value of input A.
Effectively, Q2 will turn off since C=1; and Q1 will also turn off since D=0.
Q1 and Q2 are both turned off and so we achieve Hi-Z at the output.

Now we look at the scenario when we have enabled the device.
We tie input EN to 1 and connect input A to 0.
Effectively, the NAND gate inputs are 1 and 0, therefore its output C=1.
The NOR gate inputs are both 0, therefore its output D=1.
So Q1 turns on and Q2 turns off.
We have a path from OUT to ground, therefore OUT=0.

In the final scenario, we enable the device (EN=1) and connect input A to 1.
Effectively, the NAND gate inputs are both 1, therefore its output C=0.
The NOR gate inputs are 1 and 0, therefore its output D=0.
So Q1 turns off and Q2 turns on.

We have a path from V_{cc} to OUT, therefore OUT=1.
So this is how the tristate buffer works.

<Tristate application>
6.54
Now that we have understood how tristate devices work, we would like to understand why we need them.
What is the advantage of using such devices?
The advantage is: we are able to connect two or more outputs together.
If you recall, earlier in this course you have been reminded not to connect outputs together, unless you know what you are doing.
So this is one of those situations where it is permissible to tie two or more outputs together.
If you have tristate outputs, you can tie them together, but at any point in time you must make sure at the most only one of the outputs is enabled.
It is OK to have none of them enabled; but you cannot enable two or more outputs at any time.
The maximum number of output you can enable is one; otherwise you have a serious condition known as bus contention; meaning that two or more outputs are trying to put data on the same signal line, and the end result will be incorrect.
One very common use of tristate output capability is in connecting memory devices together.
In a typical system, there will be multiple memory devices.
All of them will be putting out data to the same set of signal lines.
Therefore their outputs need to be connected together.
These memory devices have tristate output capability, therefore the outputs can be connected together to share the same data bus.

6.55
Here we will look at an example to illustrate the scenario.
On the left is a CPU, Central Processing Unit.
On the right are some memory devices.
We have a mixture of ROM (Read Only Memory) and RAM (Random Access Memory).
You notice that all of them share the same data bus.
In other words, all of them can put data on to the bus, or take data from the bus.
In the current scenario, what we see is ROM-A is enabled, therefore it outputs data on to the data bus.
At the same time, the data on the data bus is also being read by the CPU, as well as by the RAM.
These 4 components, i.e. CPU and 3 memory devices, are sharing the same data bus; but only ROM-A has its output enabled.
Only 1 of them is enabled to put data on to the data bus.

6.56
Now the scenario has changed.
This time ROM-B wants to put data on to the data bus.

Then we must make sure ROM-A is disabled and only ROM-B is enabled.

6.57
In this instance, the RAM wants to put data on to the data bus.
The RAM is enabled, but at the same time we must make sure ROM-B is disabled.
So at any time, only one of them is putting data on to the data bus.

6.58
Similarly in this case.
Now the CPU is putting data on to the data bus.
But it must make sure the other devices are disabled and therefore they cannot put data on to the data bus.

<Open-drain output>

6.59
We have seen how tristate outputs can be connected together.
Now we would like to introduce a different type of outputs which can also be connected together.
It is called open-collector or open-drain output.
This is a 2-input NAND gate.
This symbol indicates that it is an open-drain output;
meaning that it can be connected to another open-drain output.
Let's look at the circuit.
The 2 transistors: Q1 and Q2 are controlled by A and B.
When A and B are both 1, both transistors will turn on, providing a current path from Z to ground.
Thus output Z=0.
This is described by this row of the truth table.
What happens when A=0 or B=0?
You notice that there is no transistor inside the NAND gate between Vcc and output Z.
This type of output is called "open drain".
If you look at the truth table, the NAND gate output Z is open.
For the output to produce logic 1, we need to connect a pull-up resistor between Vcc and output Z.
This is an external resistor connected outside the NAND gate.
With this connection, there is now a current path from Vcc to Z via the resistor.
Therefore output Z is 1 when either A or B is 0.

6.60
One common application of an open-drain output is to light up an LED.
When sufficient current flows through an LED, it will light up.
In this circuit, R is the external pull-up resistor for the open-drain NAND gate.
When inputs A=B=1, both transistors will turn on, providing a path for sufficient current, e.g. 10 mA to flow from Vcc, through the external resistor, the LED, Q1, Q2 and then to ground. The LED will light up.
If either input A=0 or B=0,

there will be hardly any current flow through the LED and so it will not light up.

6.61
We say that open-drain outputs can be connected together.
Now let's see what happens when we do that.
In this example, we have 3 units of 2-input open-drain NAND gates.
Their outputs U, V and W are connected together to form the common output Z.
Instead of connecting 3 external pull-up resistors like this;
the outputs U, V and W share the same external pull-up resistor R connected between Z and Vcc.
Let's look at output Z.
R provides the only current path from Vcc to output Z.
However, there are 3 parallel current paths from output Z to ground:
So whether current flows this way, that way, or all 3 ways, Z will be 0.
Z can only be 1 if all 3 current paths are off.
Therefore, we say that the common point Z behaves as "wired-AND".
Z is effectively = U AND V AND W.
Note that this is achieved by wire connections, indicated by the dotted AND symbol, not by using an AND gate.

6.62
This table summarises the circuit behaviour.
The common point Z will go high only if all 3 outputs U, V and W are high.
Individually, U, V or W may go low for 2 reasons:
e.g. W may go low because inputs E=F=1;
W may also be pulled low by U or V, since all three of them are connected together.

6.63
The wired-AND behaviour may be visualised as an elastic band like this.
Let's say there are 3 hooks on the elastic band: U, V and W.
If you pull down the hook U, you will pull down the elastic band;
The other 2 hooks will also slide down.
If you pull down hook V or W, the effect is the same, i.e. the elastic band as well as the other 2 hooks will be pulled down.
This illustrates the wired-AND behaviour.
The only time the elastic band stays high is when none of the hooks U, V or W is pulled down.

Pre-recorded lecture L12 (7.1 – 7.15)

<Schmitt-trigger input>
7.1
We have previously looked at the digital circuit characteristics of a standard CMOS inverter.
Its transfer characteristic looks something like this.
As the input voltage increases from 0 to 5V, the output voltage will drop from 5V to 0V, since it is an inverter.
We will now introduce a different type of inverter which has a somewhat different transfer characteristic.
Its characteristic is not symmetrical.
What does it mean?
If we look at this transfer characteristic, it does not matter whether the input voltage increases from 0 to 5V, or decreases from 5V to 0V.
It will still follow the same transfer function.
However, for a different type of inverter, the input-output behaviour is not symmetrical.

7.2
The device we would like to introduce is called the Schmitt-trigger inverter.
A Schmitt-trigger inverter is characterised by 2 threshold voltages: V_{T-} and V_{T+}.
V_{T-} is used when the input voltage is reducing from high to low.
V_{T+} is used when the input voltage increases from 0 to 5V.
The behaviour when input increases will not be the same when input decreases.
Thus we say this behaviour is not symmetrical, and this type of behaviour is called hysteresis.
To indicate that this type of inverter behaves differently from an ordinary inverter, we use a symbol like this ▱ to highlight that it is a Schmitt-trigger input.

7.3
What we are looking at now is a Schmitt-trigger inverter.
Look at the logic symbol in diagram (b).
The shape of the symbol is the same as an ordinary inverter.
But the symbol ▱ is included to tell us that it is the Schmitt-trigger type.

Now look at the input-output transfer characteristic in diagram (a).
Notice that it is made up of 2 paths.
It will take one path when the input voltage V_{IN} rises from 0 to 5V.
It will take another path when the input voltage V_{IN} changes from 5V to 0V.
You find that in the two different paths, the two different threshold voltages will be used to determine the inverter output.
<Animation>
This happens when input voltage rises (yellow path); and this happens when input voltage reduces (green path).
As you can clearly see, the behaviour of the circuit is different, depending on whether the input voltage V_{IN} is increasing from 0-5V, in which case it will follow the yellow path

If the input voltage is decreasing from 5V to 0V, the output will follow the green path.
This is the non-symmetrical, hysteresis behaviour which is a characteristic of Schmitt-trigger device.

7.4
As a result of the Schmitt-trigger input characteristic, you find that when an input voltage rises slowly, the Schmitt-trigger inverter will behave differently from an ordinary inverter.
In the case of an ordinary inverter, the output may be unstable and fluctuating.
But in the case of a Schmitt-trigger inverter, the output will have clean transitions and no fluctuations.

7.5
Waveform (a) is a logic signal used as input for 2 different inverters:
Waveform (b) shows the output of an ordinary inverter; and waveform (c) shows the output of a Schmitt-trigger inverter.
Do take note of how the output in (b) will fluctuate, and (c) will not.
<Animation>
What you have just seen is this:
As the input voltage rises into this range (around 2.1V to 2.9V) and fluctuates, the output of the ordinary inverter also fluctuates.
Similarly over here, when the input voltage fluctuates as it decreases.
However, for the Schmitt-trigger inverter, it needs V_{IN} to rise above V_{T+} before the output will switch to logic 0.
On the other hand, when the input is dropping from high voltage to low voltage, V_{IN} has to drop below V_{T-} before the output will switch to logic 1.
Therefore the Schmitt-trigger inverter can produce an output with clean transitions between logic 0 and 1.
What is the significance?
In some digital circuits, we make use of high-to-low transitions or low-to-high transitions to trigger some events. You will encounter this in the second half of this course.
In such situations, if you subject the circuit input to such fluctuations, obviously there are many transitions which will accidentally trigger the events multiple times. This would not be intended.
In such cases, the Schmitt-trigger device will be useful because even though the input signal is somewhat noisy with fluctuations, the output from a Schmitt-trigger inverter will have clean transitions.

<PLA example>
7.6
So far all the digital circuits and logic gates we have seen, they have specific logic functions.
For example, when we look at a NAND gate or an OR gate, they each has a fixed logic function.
We do not change an OR gate into an AND gate, we also do not change an XOR gate into a NOT gate.

While such fixed-logic gates have allowed us to produce digital circuits, they lack flexibility.
What we would like to introduce now are programmable logic devices.
The logic functions of such devices can be modified by programming.
Therefore these devices provide a flexible way to implement digital logic circuits.
There are many different types of PLDs.
For example, PLA which is Programmable Logic Array; PAL which is Programmable Array Logic; complex PLD; and also FPGA.
In the lab experiments, you have used the FPGA and you have seen how it can be programmed to perform different logic functions.
In terms of complexity, PLA and PAL are much simpler compared to CPLD and FPGA.
We would like to introduce a simple device to help you understand how to achieve a programmable logic device.
The idea behind it is actually quite simple.
If you recall, a lot of digital circuits can be described using Boolean expressions.
One of the most common types of Boolean expressions we have used would be the sum-of-minterms or the sum-of-product expressions.
If we are able to describe a logic function using such expressions, we can easily implement it on a PLA.

7.7
We are looking at an example of a PLA.
This PLA has 4 inputs (I1, I2, I3, I4) and 3 outputs (O1, O2, O3); therefore it is known as a 4x3 PLA.
Within this PLA, it is able to support 6 product terms: P1, P2, … and P6.
How do we make use of this PLA?
Let's look at the inputs, we are able to obtain the non-inverted inputs as well as the inverted inputs.
In other words, for input I1, I will be able to obtain I1 and I1'.
These will help me to form different Boolean expressions using the non-inverted input or the inverted input.
Look at the AND gate structures.
Each AND gate can accept 8 inputs, which can be traced to the 4 inputs I1, I2, I3, I4 and their inverted form, I1', I2', I3' and I4'.
Therefore we have a choice of using any one or more of these inputs and feed them to any one of these AND gates.
Subsequently, we can collect the AND gate outputs P1, …, P6 and connect them to the inputs of the OR gate.
If you look at these 3 OR gates, each has 6 inputs.
Thus each OR gate can take in up to 6 product terms.
What you also see on this diagram are the connections (marked by x).
The x marked out all the programmable connections.
In other words, when we program a PLA, we specify to make or break a connection.
By selectively connecting some and disconnecting the rest (at the AND gate inputs), we will be able to form different product terms; and over here (at the OR gate inputs) by

selectively connecting some and disconnecting the rest, we can select the product terms to be summed together.
As this is a somewhat complicated diagram, it has been simplified.

7.8
This is a simplified diagram of the same PLA that we were looking at.
We can still see the 8 possible choices of inputs going into each AND gate.
There are 6 possible choices of inputs going into each OR gate.
We have not done any programming yet.
In other words, this 4x3 PLA is ready for us to program it to suit our purpose.
(x marks all the available connections for programming)

7.9
Now we look at this example where the PLA has been programmed in this manner.
By programming, we are actually selecting the connections that we want to make.
The selected connections are indicated by the crosses in this diagram.
Where a cross is present, it means there is a connection.
If there is no cross, it means there is no connection.
For example, in this case, inputs I1 and I2 are connected to the inputs of the AND gate producing output P1.
The rest are not connected to P1, although they may be connected to the other AND gate inputs.
Similarly, at the OR gate inputs we have chosen to connect the products P1 and P2 to produce O1.
To produce O2, we have chosen to use P3, P4 and P5.
You can see now, by selectively programming the connections, we will be able to obtain different product terms, and collect different product terms to form sum-of-product expressions.
If you look carefully at the diagram, the product terms are:
P1 = A B
P2 = A’ B’ C’ D’
P3 = A C’
P4 = A’ C D
P5 = B
P6 = A’ B’ D’
After generating the product terms, we will collect the relevant ones and feed them into the OR gates.
The selected product terms are summed to produce these outputs:
O1 = P1 + P2 = A B + A’ B’ C’ D’
O2 = P3 + P4 + P5 = A C’ + A’ C D + B
O3 = P1 + P3 + P6 = A B + A C’ + A’ B’ D’
Each output implements an SOP expression.
In other words, the PLA offers a simple way to implement different Boolean expressions by forming different product terms, and summing product terms together to implement an SOP expression.

7.10
Using the programmed PLA example, we can generate the truth table of the implemented circuit.
The inputs are A, B, C, D and the outputs are O1, O2 and O3.
You find that this table can also be easily programmed if you have memory devices.
In other words, if you treat each row as a memory location, you can then program the output bits, 1 or 0, into each of these memory locations.
This is another way of implementing programmable logic devices, which is usually known as Lookup Tables (LUT).
LUTs are used in FPGA.

<Fixed-point and floating-point numbers>

7.11
Now we'd like to talk about fixed-point numbers versus floating-point numbers.
Sometimes in programming, you declare a variable as integer, but there are times you declare a variable as float or double.
When you declare a variable as float or double, you are using floating-point numbers.
Let us first look at fixed-point numbers.
What does it mean to be a fixed-point number?
Essentially we have a certain number of bits or digits available to represent a numerical value.
In the case of decimal system, maybe we have 10 decimal digits; and maybe we want to allocate all 10 digits to be in front of the decimal point.
In such a case, it means it is an integer because we have 10 digits, and we use all of them to represent the integer portion, and do not leave any digit for the fractional portion.
Maybe in a different scenario, we want to allocate the digits differently.
For example, we may allocate 3 digits in front of the decimal point, and 3 digits behind the decimal point.
We will achieve a representation like this: 123.456_{10}
The same concept can be applied to other number bases.
For example, base 16 or base 2, i.e. binary.
In the case of binary, in this example, we have 8 bits.
We have allocated 4 bits for the integer part, and 4 bits for the fractional part.
So because we have fixed the number of digits here (the integer portion) and the number of digits here (the fractional portion), therefore this point, the radix point, has a fixed position.
Therefore we call these fixed-point numbers.
The concept of fixed-point numbers is quite easy to understand, but it has certain disadvantages.

7.12
You find that fixed-point numbers are usually used for representing integers.
Fixed-point numbers can certainly be used to representing non-integers, i.e. numbers with a fractional portion.
However when you do that, there could be certain disadvantages.

Firstly because of the inflexibility of the radix point, only a relatively small range of numbers can be represented.
The other disadvantage is, there may be a loss of significant digits when we perform arithmetic operations.
For example, let's say we have a system where we use 1 bit for the integer portion, and 2 bits for the fractional portion.
Using this, we are able to represent 0.01 with no problem.
But if I take the number and multiply with itself.
0.01 x 0.01 = 0.0001
But since I have already decided to allocate only 2 bits for the fractional portion - which cannot be changed in a fixed-point system – so I will only have 2 bits for the fractional portion of the result.
In other words, the information contained in this part of the answer is lost.
Therefore fixed-point numbers are not very suitable for this kind of situations.

7.13
To overcome the problem, we can make use of floating-point numbers
Floating-point numbers are simply the machine/computer equivalent of the scientific notation.
The notation is usually written in this form: $S \times R^E$

7.14
The different parts of the representation are given different names.
S is called the significand.
R is the radix.
E is the exponent.
The significand is usually a signed, fractional, fixed-point number.
The radix is an integer (10 for decimal; 2 for binary).
The exponent is usually a signed, fixed-point integer.
So a floating-point number is actually made up of 3 parts: significand, radix and exponent.

7.15
In a lot of computers, there are standard ways of representing floating-point numbers.
They are typically based on the IEEE standard.
We will just take a quick look at this standard so that we have an idea of what it means to be a floating-point number.
To represent a floating-point number, we can use either 32 bits or 64 bits.
With 64 bits, we can represent a number with greater precision, therefore it is known as double precision.
With 32 bits it is only single precision.
These charts show how the 32 bits or 64 bits are allocated to represent different parts of a floating-point number.
For example in single precision, bits 22 to 0 are used for the significand, the magnitude portion only.
Bits 30 to 23 are used to represent the exponent.

This bit, bit 31, is used to represent the sign of the significand.
Notice that the significand in this standard is represented in sign-magnitude notation, which we have learned before.
The exponent is represented in 2's complement notation which we have also learned.

<End of transcript>