Building a Computer
Outline

1 Numbers

2 Letters and Strings

3 Structured Information

4 Boolean Algebra and Functions

5 Logic Using Electrical Circuits

6 Computing With Logic

7 Memory

8 von Neumann Architecture
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, \( 135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2 \)

In the base 8 (octal) system, \( 135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8 \)

In the base 10 (decimal) system, \( 135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0 \)

In general, if we choose some base \( b \geq 2 \), every positive integer between 0 and \( b^d - 1 \) can be uniquely represented using \( d \) digits, with coefficients having values 0 through \( b - 1 \).

A modern 64-bit computer can represent integers up to \( 2^{64} - 1 \).
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, $135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2$

In the base 8 (octal) system, $135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8$

In the base 10 (decimal) system, $135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$

In general, if we choose some base $b \geq 2$, every positive integer between 0 and $b^d - 1$ can be uniquely represented using $d$ digits, with coefficients having values 0 through $b - 1$.

A modern 64-bit computer can represent integers up to $2^{64} - 1$. 
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system:

In the base 2 (binary) system, $135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2$

In the base 8 (octal) system, $135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8$

In the base 10 (decimal) system, $135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$

In general, if we choose some base $b \geq 2$, every positive integer between 0 and $b^d - 1$ can be uniquely represented using $d$ digits, with coefficients having values 0 through $b - 1$.

A modern 64-bit computer can represent integers up to $2^{64} - 1$.
Numbers

At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, $135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2$

In the base 8 (octal) system, $135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8$

In the base 10 (decimal) system, $135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$

In general, if we choose some base $b \geq 2$, every positive integer between 0 and $b^d - 1$ can be uniquely represented using $d$ digits, with coefficients having values 0 through $b - 1$.

A modern 64-bit computer can represent integers up to $2^{64} - 1$.
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, \(135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2\)

In the base 8 (octal) system, \(135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8\)

In the base 10 (decimal) system, \(135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0\)

In general, if we choose some base \(b \geq 2\), every positive integer between 0 and \(b^d - 1\) can be uniquely represented using \(d\) digits, with coefficients having values 0 through \(b - 1\).

A modern 64-bit computer can represent integers up to \(2^{64} - 1\).
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, \(135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2\)

In the base 8 (octal) system, \(135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8\)

In the base 10 (decimal) system, \(135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0\)

In general, if we choose some base \(b \geq 2\), every positive integer between 0 and \(b^d - 1\) can be uniquely represented using \(d\) digits, with coefficients having values 0 through \(b - 1\).

A modern 64-bit computer can represent integers up to \(2^{64} - 1\).
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, $135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2$

In the base 8 (octal) system, $135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8$

In the base 10 (decimal) system, $135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$

In general, if we choose some base $b \geq 2$, every positive integer between 0 and $b^d - 1$ can be uniquely represented using $d$ digits, with coefficients having values 0 through $b - 1$.

A modern 64-bit computer can represent integers up to $2^{64} - 1$. 
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, $135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2$

In the base 8 (octal) system, $135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8$

In the base 10 (decimal) system, $135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$

In general, if we choose some base $b \geq 2$, every positive integer between 0 and $b^d - 1$ can be uniquely represented using $d$ digits, with coefficients having values 0 through $b - 1$.

A modern 64-bit computer can represent integers up to $2^{64} - 1$. 
At the most fundamental level, a computer manipulates electricity according to specific rules.

To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use.

To represent integers (for example, 135), computers use the binary system.

In the base 2 (binary) system, \(135_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000111_2\)

In the base 8 (octal) system, \(135_{10} = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 207_8\)

In the base 10 (decimal) system, \(135_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0\)

In general, if we choose some base \(b \geq 2\), every positive integer between 0 and \(b^d - 1\) can be uniquely represented using \(d\) digits, with coefficients having values 0 through \(b - 1\).

A modern 64-bit computer can represent integers up to \(2^{64} - 1\).
Arithmetic in any base is analogous to arithmetic in base 10

Examples of addition in base 10 and base 2

\[
\begin{array}{c}
1 \\
1 \ 7 \\
+ \ 2 \ 5 \\
\hline
4 \ 2
\end{array}
\quad \begin{array}{c}
1 \ 1 \\
1 \ 1 \ 1 \\
+ \ 1 \ 1 \ 0 \\
\hline
1 \ 1 \ 0 \ 1
\end{array}
\]

To represent a negative integer, a computer typically uses a system called two’s complement, which involves flipping the bits of the positive number and then adding 1

For example, on an 8-bit computer, \(3 = 00000011\), so \(-3 = 11111101\); note that just like how \(3 + (-3) = 0\), we have \(00000011 + 11111101 = 100000000\), which is 0 since we are on an 8-bit computer
Numbers

Arithmetic in any base is analogous to arithmetic in base 10

Examples of addition in base 10 and base 2

```
1
1 7
+ 2 5
---
4 2
```

```
1 1
1 1 1 1
+ 1 1 0
---
1 1 0 1
```

To represent a negative integer, a computer typically uses a system called two’s complement, which involves flipping the bits of the positive number and then adding 1

For example, on an 8-bit computer, $3 = 00000011$, so $-3 = 11111101$; note that just like how $3 + (-3) = 0$, we have $00000011 + 11111101 = 100000000$, which is 0 since we are on an 8-bit computer
Arithmetic in any base is analogous to arithmetic in base 10

Examples of addition in base 10 and base 2

\[
\begin{array}{c}
1 \\
\hline
1 \\
\hline
1 \\
\hline
+ \\
\hline
2 \\
\hline
5 \\
\hline
4 \\
\hline
2 \\
\end{array}
\quad
\begin{array}{c}
1 \\
\hline
1 \\
\hline
1 \\
\hline
+ \\
\hline
1 \\
\hline
1 \\
\hline
0 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
0 \\
\hline
1 \\
\end{array}
\]

To represent a negative integer, a computer typically uses a system called two’s complement, which involves flipping the bits of the positive number and then adding 1.

For example, on an 8-bit computer, \(3 = 00000011\), so \(-3 = 11111101\); note that just like how \(3 + (-3) = 0\), we have \(00000011 + 11111101 = 100000000\), which is 0 since we are on an 8-bit computer.
Arithmetic in any base is analogous to arithmetic in base 10

Examples of addition in base 10 and base 2

\[
\begin{array}{cccc}
1 \\
1 & 7 \\
+ & 2 & 5 \\
\hline
4 & 2
\end{array}
\quad \begin{array}{cccc}
1 & 1 \\
1 & 1 & 1 & 1 \\
+ & 1 & 1 & 0 \\
\hline
1 & 1 & 0 & 1
\end{array}
\]

To represent a negative integer, a computer typically uses a system called two’s complement, which involves flipping the bits of the positive number and then adding 1

For example, on an 8-bit computer, \(3 = 00000011\), so \(-3 = 11111101\); note that just like how \(3 + (-3) = 0\), we have \(00000011 + 11111101 = 100000000\), which is 0 since we are on an 8-bit computer.
Arithmetic in any base is analogous to arithmetic in base 10

Examples of addition in base 10 and base 2

\[
\begin{align*}
1 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 1 \\
1 & \quad 7 \quad \quad \quad \quad \quad \quad \quad \quad \quad + \quad 1 \quad 1 \\
+ & \quad 2 \quad 5 \quad \quad \quad \quad \quad \quad \quad \quad \quad + \quad 1 \quad 1 \\
\hline
4 & \quad 2 \quad \quad \quad \quad \quad \quad \quad \quad \quad 0 \quad 1 \\
\end{align*}
\]

To represent a negative integer, a computer typically uses a system called two’s complement, which involves flipping the bits of the positive number and then adding 1

For example, on an 8-bit computer, \(3 = 00000011\), so \(-3 = 11111101\); note that just like how \(3 + (-3) = 0\), we have \(00000011 + 11111101 = 100000000\), which is 0 since we are on an 8-bit computer.
If we are using base 10 and only have eight digits to represent our numbers, we might use the first six digits for the fractional part of a number and last two for the exponent.

For example, 31415901 would represent $0.314159 \times 10^1 = 3.14159$

Computers use a similar idea to represent fractional numbers.
If we are using base 10 and only have eight digits to represent our numbers, we might use the first six digits for the fractional part of a number and last two for the exponent.

For example, 31415901 would represent $0.314159 \times 10^1 = 3.14159$

Computers use a similar idea to represent fractional numbers.
If we are using base 10 and only have eight digits to represent our numbers, we might use the first six digits for the fractional part of a number and last two for the exponent.

For example, 31415901 would represent $0.314159 \times 10^1 = 3.14159$

Computers use a similar idea to represent fractional numbers.
If we are using base 10 and only have eight digits to represent our numbers, we might use the first six digits for the fractional part of a number and last two for the exponent.

For example, 31415901 would represent $0.314159 \times 10^1 = 3.14159$

Computers use a similar idea to represent fractional numbers.
Letters and Strings

In order to represent letters numerically, we need a convention on the encoding.

The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange).

ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters.

ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn’t provide for characters used in many languages.

The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more.
In order to represent letters numerically, we need a convention on the encoding.

The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange).

ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters.

ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn’t provide for characters used in many languages.

The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more.
In order to represent letters numerically, we need a convention on the encoding.

The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange).

ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters.

ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn’t provide for characters used in many languages.

The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more.
Letters and Strings

In order to represent letters numerically, we need a convention on the encoding.

The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange).

ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters.

ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn’t provide for characters used in many languages.

The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more.
Letters and Strings

In order to represent letters numerically, we need a convention on the encoding.

The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange).

ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters.

ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn’t provide for characters used in many languages.

The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more.
Letters and Strings

In order to represent letters numerically, we need a convention on the encoding.

The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange).

ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters.

ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn’t provide for characters used in many languages.

The International Standards Organization’s (ISO) 16-bit Unicode system can represent every character in every known language, with room for more.
Letters and Strings

A string is represented as a sequence of numbers, with a “length field” at the very beginning that specifies the length of the string.

For example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to the string “chocolate”, with the length field set to 9.
A string is represented as a sequence of numbers, with a “length field” at the very beginning that specifies the length of the string.

For example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to the string “chocolate”, with the length field set to 9.
A string is represented as a sequence of numbers, with a “length field” at the very beginning that specifies the length of the string.

For example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to the string “chocolate”, with the length field set to 9.
Structured Information

We can represent any information as a sequence of numbers

Examples

• A picture can be represented as a sequence of pixels, each represented as three numbers giving the amount of red, green, and blue at that pixel
• A sound can be represented as a temporal sequence of “sound pressure levels” in the air
• A movie can be represented as a temporal sequence of individual pictures, usually 24 or 30 per second, along with a matching sound sequence
We can represent any information as a sequence of numbers

Examples

• A picture can be represented as a sequence of pixels, each represented as three numbers giving the amount of red, green, and blue at that pixel
• A sound can be represented as a temporal sequence of “sound pressure levels” in the air
• A movie can be represented as a temporal sequence of individual pictures, usually 24 or 30 per second, along with a matching sound sequence
Structured Information

We can represent any information as a sequence of numbers

Examples
- A picture can be represented as a sequence of pixels, each represented as three numbers giving the amount of red, green, and blue at that pixel
- A sound can be represented as a temporal sequence of “sound pressure levels” in the air
- A movie can be represented as a temporal sequence of individual pictures, usually 24 or 30 per second, along with a matching sound sequence
Boolean variables are variables that take the value `True` (1) or `False` (0)

With booleans 1 and 0 we could use the operations (functions) `AND`, `OR`, and `NOT` to build up more interesting boolean functions

A truth table for a boolean function is a listing of all possible combinations of values of the input variables, together with the result produced by the function

Truth tables for `AND`, `OR`, and `NOT` functions

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x AND y</th>
<th>x</th>
<th>y</th>
<th>x OR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>NOT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Algebra and Functions

Boolean variables are variables that take the value True (1) or False (0)

With booleans 1 and 0 we could use the operations (functions) \texttt{AND}, \texttt{OR}, and \texttt{NOT} to build up more interesting boolean functions

A truth table for a boolean function is a listing of all possible combinations of values of the input variables, together with the result produced by the function

Truth tables for \texttt{AND}, \texttt{OR}, and \texttt{NOT} functions

\[
\begin{array}{ccc}
  x & y & x \ \text{AND} \ y \\
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
  x & y & x \ \text{OR} \ y \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{cc}
  x & \text{NOT} \ x \\
  0 & 1 \\
  1 & 0 \\
\end{array}
\]
Boolean Algebra and Functions

Boolean variables are variables that take the value True (1) or False (0)

With booleans 1 and 0 we could use the operations (functions) AND, OR, and NOT to build up more interesting boolean functions.

A truth table for a boolean function is a listing of all possible combinations of values of the input variables, together with the result produced by the function.

**Truth tables for AND, OR, and NOT functions**

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \text{ AND } y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \text{ OR } y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>NOT $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Algebra and Functions

Boolean variables are variables that take the value True (1) or False (0)

With bools 1 and 0 we could use the operations (functions) AND, OR, and NOT to build up more interesting boolean functions

A truth table for a boolean function is a listing of all possible combinations of values of the input variables, together with the result produced by the function

Truth tables for AND, OR, and NOT functions

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x AND y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x OR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>NOT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean variables are variables that take the value True (1) or False (0)

With booleans 1 and 0 we could use the operations (functions) AND, OR, and NOT to build up more interesting boolean functions

A truth table for a boolean function is a listing of all possible combinations of values of the input variables, together with the result produced by the function

**Truth tables for AND, OR, and NOT functions**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x AND y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x OR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>NOT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Any function of boolean variables, no matter how complex, can be expressed in terms of **AND**, **OR**, and **NOT**

Consider the proposition “if you score over 93% in this course, then you will get an A”

The truth values for the above proposition is given by the “implication” function \((x \implies y)\) having the following truth table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(x \implies y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The function can be compactly written as \(\overline{x} \lor x \land y\) (or \(\overline{x} + xy\))
Boolean Algebra and Functions

Any function of boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT

Consider the proposition “if you score over 93% in this course, then you will get an A”

The truth values for the above proposition is given by the “implication” function \((x \implies y)\) having the following truth table

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(x \implies y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The function can be compactly written as NOT \(x\) OR \(x\) AND \(y\) (or \(\bar{x} + xy\))
Boolean Algebra and Functions

Any function of boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT.

Consider the proposition “if you score over 93% in this course, then you will get an A”.

The truth values for the above proposition is given by the “implication” function \( x \implies y \) having the following truth table:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x \implies y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The function can be compactly written as NOT \( x \) OR \( x \) AND \( y \) (or \( \bar{x} + xy \)).
Any function of boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT.

Consider the proposition “if you score over 93% in this course, then you will get an A”

The truth values for the above proposition is given by the “implication” function \((x \implies y)\) having the following truth table

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(x \implies y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The function can be compactly written as NOT \(x\) OR \(x\) AND \(y\) (or \(\overline{x} + xy\))
Boolean Algebra and Functions

Any function of boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT

Consider the proposition “if you score over 93% in this course, then you will get an A”

The truth values for the above proposition is given by the “implication” function $(x \implies y)$ having the following truth table

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \implies y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The function can be compactly written as NOT $x$ OR $x$ AND $y$ (or $\overline{x} + xy$)
Boolean Algebra and Functions

The minterm expansion algorithm, due to Claude Shannon, provides a systematic approach for building boolean functions from truth tables.

Minterm expansion algorithm

1. Write down the truth table for the boolean function under consideration.
2. Delete all rows from the truth table where the value of the function is 0.
3. For each remaining row, create something called a “minterm” as follows:
   a. For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to NOT it.
   b. Now AND all of these variables together.
4. Combine all of the minterms for the rows using OR.
The minterm expansion algorithm, due to Claude Shannon, provides a systematic approach for building boolean functions from truth tables.

Minterm expansion algorithm

1. Write down the truth table for the boolean function under consideration
2. Delete all rows from the truth table where the value of the function is 0
3. For each remaining row, create something called a “minterm” as follows
   a. For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to NOT it
   b. Now AND all of these variables together
4. Combine all of the minterms for the rows using OR
The minterm expansion algorithm, due to Claude Shannon, provides a systematic approach for building boolean functions from truth tables.

Minterm expansion algorithm

1. Write down the truth table for the boolean function under consideration.
2. Delete all rows from the truth table where the value of the function is 0.
3. For each remaining row, create something called a “minterm” as follows:
   a. For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to NOT it.
   b. Now AND all of these variables together.
4. Combine all of the minterms for the rows using OR.
For the implication function, the minterm expansion algorithm applied as follows

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \implies y$</th>
<th>minterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\bar{xy}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$xy$</td>
</tr>
</tbody>
</table>

produces the boolean function $\bar{x}y + \bar{xy} + xy$, which is equivalent to the simpler $\bar{x} + xy$ function.

Finding the simplest form of a boolean function is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science.
For the implication function, the minterm expansion algorithm applied as follows

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \implies y$</th>
<th>minterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$xy$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$xy$</td>
</tr>
</tbody>
</table>

produces the boolean function $\bar{x}y + \bar{x}y + xy$, which is equivalent to the simpler $\bar{x} + xy$ function.

Finding the simplest form of a boolean function is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science.
For the implication function, the minterm expansion algorithm applied as follows

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>x → y</th>
<th>minterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\bar{x}y$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$xy$</td>
</tr>
</tbody>
</table>

produces the boolean function $\bar{x}y + \bar{x}y + xy$, which is equivalent to the simpler $\bar{x} + xy$ function.

Finding the simplest form of a boolean function is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science.
Logic Using Electrical Circuits

An electromechanical switch is a device in which when the input is off, the output is “low” (0), and when the input is on, the output is “high” (1).

The NOT gate constructed using a switch that conducts only when the input is off.

The AND and OR gates for computing $x \text{ AND } y$ and $x \text{ OR } y$, constructed using electromechanical switches.
Logic Using Electrical Circuits

An electromechanical switch is a device in which when the input is off, the output is “low” (0), and when the input is on, the output is “high” (1)

The **NOT** gate constructed using a switch that conducts only when the input is off

The **AND** and **OR** gates for computing $x \text{ AND } y$ and $x \text{ OR } y$, constructed using electromechanical switches
Logic Using Electrical Circuits

An electromechanical switch is a device in which when the input is off, the output is “low” (0), and when the input is on, the output is “high” (1).

The **NOT** gate constructed using a switch that conducts only when the input is off.

The **AND** and **OR** gates for computing \( x \text{ AND } y \) and \( x \text{ OR } y \), constructed using electromechanical switches.
Logic Using Electrical Circuits

An electromechanical switch is a device in which when the input is off, the output is “low” (0), and when the input is on, the output is “high” (1).

The **NOT** gate constructed using a switch that conducts only when the input is off.

The **AND** and **OR** gates for computing $x \text{ AND } y$ and $x \text{ OR } y$, constructed using electromechanical switches.
Computers today are built with much smaller, much faster, more reliable, and more efficient transistorized switches.

Since the details of the switches aren’t terribly important at this level of abstraction, we represent, or “abstract”, the gates using the following symbols:

\[ \overline{x \overline{y} + \overline{x}y + xy} \]

A logical circuit for the implication function \( \overline{x \overline{y} + \overline{x}y + xy} \)
Computers today are built with much smaller, much faster, more reliable, and more efficient transistorized switches.

Since the details of the switches aren’t terribly important at this level of abstraction, we represent, or “abstract”, the gates using the following symbols:

A logical circuit for the implication function $\bar{x}\bar{y} + \bar{y} + xy$
Logic Using Electrical Circuits

Computers today are built with much smaller, much faster, more reliable, and more efficient transistorized switches.

Since the details of the switches aren’t terribly important at this level of abstraction, we represent, or “abstract”, the gates using the following symbols:

- **AND**
- **OR**
- **NOT**

A logical circuit for the implication function $\overline{x}y + \overline{x}y + xy$.
Logic Using Electrical Circuits

Computers today are built with much smaller, much faster, more reliable, and more efficient transistorized switches.

Since the details of the switches aren’t terribly important at this level of abstraction, we represent, or “abstract”, the gates using the following symbols:

- AND
- OR
- NOT

A logical circuit for the implication function $\bar{x} \bar{y} + \bar{y}x + xy$

\[
\begin{align*}
x & \quad y \\
\bar{x} & \quad \bar{y} \quad x \\
\text{AND} & \quad \text{OR} \quad \text{NOT}
\end{align*}
\]

$\Rightarrow \quad x \Rightarrow y$
A truth table describing the addition of two two-bit numbers to get a three-bit result.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x + y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>

Building a corresponding circuit using the minterm expansion algorithm is infeasible — adding two 16-bit numbers, for example, will result in a circuit with several billion gates.
A truth table describing the addition of two two-bit numbers to get a three-bit result

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x + y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>

Building a corresponding circuit using the minterm expansion algorithm is infeasible — adding two 16-bit numbers, for example, will result in a circuit with several billion gates.
A truth table describing the addition of two two-bit numbers to get a three-bit result

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x + y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>

Building a corresponding circuit using the minterm expansion algorithm is infeasible — adding two 16-bit numbers, for example, will result in a circuit with several billion gates.
Computing With Logic

We build a relatively simple circuit called a full adder (FA) that does just one column of addition

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$c_{in}$</th>
<th>$z$</th>
<th>$c_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The minterm expansion principle applied to the truth table for the FA circuit yields the following boolean functions

$$ z = \bar{x}\bar{y}c_{in} + \bar{x}y\bar{c}_{in} + x\bar{y}\bar{c}_{in} + xy\bar{c}_{in} $$

$$ c_{out} = \bar{x}yc_{in} + x\bar{y}c_{in} + xy\bar{c}_{in} + xyc_{in} $$
Computing With Logic

We build a relatively simple circuit called a full adder (FA) that does just one column of addition.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>cin</th>
<th>z</th>
<th>cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The minterm expansion principle applied to the truth table for the FA circuit yields the following boolean functions:

\[
z = \bar{x}y\bar{cin} + xy\bar{cin} + x\bar{y}\bar{cin} + xycin
\]

\[
c_{out} = \bar{x}ycin + xy\bar{cin} + xy\bar{cin} + xycin
\]
We build a relatively simple circuit called a full adder (FA) that does just one column of addition.

The minterm expansion principle applied to the truth table for the FA circuit yields the following boolean functions:

\[
\begin{align*}
    z &= \overline{x} \overline{y} c_{in} + \overline{x} y \overline{c}_{in} + x \overline{y} c_{in} + x y c_{in} \\
    c_{out} &= \overline{x} y c_{in} + x \overline{y} c_{in} + x y \overline{c}_{in} + x y c_{in}
\end{align*}
\]
We can “chain” $n$ full adders together to add two $n$-bit numbers, and the resulting circuit is called a ripple-carry adder.

A 2-bit ripple-carry adder
We can “chain” $n$ full adders together to add two $n$-bit numbers, and the resulting circuit is called a ripple-carry adder.

A 2-bit ripple-carry adder
We can “chain” \( n \) full adders together to add two \( n \)-bit numbers, and the resulting circuit is called a ripple-carry adder.

A 2-bit ripple-carry adder
We can “chain” $n$ full adders together to add two $n$-bit numbers, and the resulting circuit is called a ripple-carry adder.

A 2-bit ripple-carry adder
Truth table for a **NOR** gate (OR followed by NOT)

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \text{ NOR } y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A latch is a device that allows us to “lock” a bit and retrieve it later.

By aggregating millions of latches we have the Random Access Memory (RAM).

A latch can be constructed from two **NOR** gates as shown below.

where the input $S$ is known as “set” while the input $R$ is known as “reset”
Truth table for a \texttt{NOR} gate (\texttt{OR} followed by \texttt{NOT})

\begin{tabular}{c c | c}
\textit{x} & \textit{y} & \textit{x} \texttt{NOR} \textit{y} \\
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\end{tabular}

A latch is a device that allows us to “lock” a bit and retrieve it later.

By aggregating millions of latches we have the Random Access Memory (RAM).

A latch can be constructed from two \texttt{NOR} gates as shown below.

where the input \textit{S} is known as “set” while the input \textit{R} is known as “reset”
Memory

Truth table for a \texttt{NOR} gate (\texttt{OR} followed by \texttt{NOT})

\begin{center}
\begin{tabular}{c|c|c}
  $x$ & $y$ & $x \ \texttt{NOR} \ y$ \\
  \hline
  0 & 0 & 1 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 0 \\
\end{tabular}
\end{center}

A latch is a device that allows us to “lock” a bit and retrieve it later

By aggregating millions of latches we have the Random Access Memory (RAM)

A latch can be constructed from two \texttt{NOR} gates as shown below

where the input $S$ is known as “set” while the input $R$ is known as “reset”
Memory

Truth table for a NOR gate (or followed by NOT)

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x$ NOR $y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A latch is a device that allows us to “lock” a bit and retrieve it later

By aggregating millions of latches we have the Random Access Memory (RAM)

A latch can be constructed from two NOR gates as shown below

where the input $S$ is known as “set” while the input $R$ is known as “reset”
Memory

Truth table for a \texttt{NOR} gate (\texttt{OR} followed by \texttt{NOT})

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \texttt{ NOR} y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A latch is a device that allows us to “lock” a bit and retrieve it later

By aggregating millions of latches we have the Random Access Memory (RAM)

A latch can be constructed from two \texttt{NOR} gates as shown below

![NOR gate diagram]

where the input $S$ is known as “set” while the input $R$ is known as “reset”
In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
von Neumann Architecture

In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
von Neumann Architecture

In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
In a modern computer, the CPU (Central Processing Unit) is where all the computation takes place.

The CPU has devices such as ripple-carry adders, multipliers, etc., for doing arithmetic, and a small amount of (scratch) memory called registers.

The computer’s main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer’s circuit board.

A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution.

The CPU has two special registers: a program counter that keeps track of the location in memory where it will find the next instruction and an instruction register that stores the next instruction to execute.

This way of organizing computation is due to the famous mathematician and physicist, John von Neumann, and is known as the von Neumann architecture.
Instructions, like data, can be encoded as numbers.

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide.

Each of the instructions will need a number, called an operation code (or opcode), to represent it.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells.

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands.

For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`. 
Instructions, like data, can be encoded as numbers

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide

Each of the instructions will need a number, called an operation code (or opcode), to represent it

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands

For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`
Instructions, like data, can be encoded as numbers

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide

Each of the instructions will need a number, called an operation code (or opcode), to represent it

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands

For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`
von Neumann Architecture

Instructions, like data, can be encoded as numbers

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide

Each of the instructions will need a number, called an operation code (or opcode), to represent it

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands

For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`
von Neumann Architecture

Instructions, like data, can be encoded as numbers

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide

Each of the instructions will need a number, called an operation code (or opcode), to represent it

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands

For example, the instruction add 3 0 2 (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as 00110010
von Neumann Architecture

Instructions, like data, can be encoded as numbers

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide

Each of the instructions will need a number, called an operation code (or opcode), to represent it

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands

For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`
von Neumann Architecture

Instructions, like data, can be encoded as numbers

For example, let’s assume an 8-bit computer with only four instructions: add, subtract, multiply, and divide

Each of the instructions will need a number, called an operation code (or opcode), to represent it

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Multiply</td>
</tr>
<tr>
<td>11</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Next, let’s assume that our computer has four registers, numbered 0 (00), 1 (01), 2 (10), and 3 (11); and 256 8-bit memory cells

An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands

For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`
Our computer operates by repeatedly performing the following procedure

1. Send the address in the program counter (commonly called the PC) to the memory, asking it to read that location
2. Load the value from memory into the instruction register
3. Decode the instruction register to determine what instruction to execute and which registers to use
4. Execute the requested instruction, which involves reading operands from registers, performing arithmetic, and sending the results back to the destination register
5. Increment PC so that it contains the address of the next instruction in memory
von Neumann Architecture

Our computer operates by repeatedly performing the following procedure

1. Send the address in the program counter (commonly called the PC) to the memory, asking it to read that location
2. Load the value from memory into the instruction register
3. Decode the instruction register to determine what instruction to execute and which registers to use
4. Execute the requested instruction, which involves reading operands from registers, performing arithmetic, and sending the results back to the destination register
5. Increment PC so that it contains the address of the next instruction in memory

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Instruction Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00100001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register 0</th>
<th>Register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101</td>
<td>00000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000111</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Central Processing Unit (CPU)

<table>
<thead>
<tr>
<th>Location (Binary)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

Memory