There's no end in sight... now we have micros with the power of minis, "superminis" that rival mainframes, minis on a chip... and there's talk of reducing components to molecular size using recombinant DNA technology...

There seems to be no such thing as a computer with too much computing power. No matter the speed or capacity, computers always find jobs to do... and no wonder: this is the age of excess information!

Part II
Logical Spaghetti
Computers are like elephants: there are a lot of ways to describe them...

A powerful calculator!
Made of switches!

It follows instructions!

An input-output device!

Exotic hardware!

How does one get to the heart of the matter?

With an elephant cleaver?
IF THERE'S ONE IDEA WE'VE TRIED TO DRUM IN, IT'S THAT THE COMPUTER IS ESSENTIALLY AN INFORMATION PROCESSOR. SO FORGET THE ELEPHANT...

TO UNDERSTAND INFORMATION PROCESSING, IT HELPS TO COMPARc IT WITH A MORE FAMILIAR PROCESS: COOKING. SO Step INTO GRANDMOTHER EGGNOGE'S KITCHEN, AS SHE PREPARES BASIC SPAGHETTI...

**HERE'S THE WORLD FAMOUS RECIPE:**

1. **Bring a kettle of salted water to boil.**
2. **Add 8 oz. of raw spaghetti.**
3. **Boil for 10 minutes.**
4. **Drain through a sieve.**

**Serve...**

YOU CAN'T EAT INFORMATION!

THIS SPAGHETTI IS BETTER ANALYZED THAN EATEN!
It's not hard to distinguish a few components in this process:

First, the ingredients, or input.

- Dry spaghetti
- Water
- Salt

Next, the equipment which does the cooking: hands, kettle, stove, saltshaker, sieve, plate, spoon.

These form the processing unit.

Less obviously, there is a part of the cook's brain which controls the process. It monitors and directs the step-by-step unfolding of the recipe. This is referred to as the control unit.

And of course the completed dish, or output.

Of course, spaghetti is nothing special! Any recipe could be processed by the same basic structure:

Ingredients or input $\Rightarrow$ A processing unit under control $\Rightarrow$ Output

Or, more abstractly:

Input $\Rightarrow$ Processing $\Rightarrow$ Output unit

White arrows (→) are the flow of food.
Gray arrow (––) is the flow of information.
Black arrow (––→) is the flow of control.

Ugh! What is this?!
With computers, the diagram is slightly different:

There are two reasons for this: one is the fact that input and output are information, not food—so the gray arrow is the same as the white ones.

The other is the great importance of memory, which forms the fifth and final component. In computers, all information passes into memory first! Here's the diagram:

\[\text{INPUT} \rightarrow \text{MEMORY} \rightarrow \text{OUTPUT}\]

\[\Rightarrow \text{information flow} \quad \Rightarrow \text{control flow}\]

Von Neumann's idea:

In the case of computers, the input consists of all the "raw" data to be processed—as well as the entire "recipe," or program, which specifies what's to be done with them.

The memory stores the input and results from the processing unit:

Control reads the program and translates it into a sequence of machine operations.

The processing unit performs the actual additions, multiplication, counting, comparison, etc., on information received from memory.

The output consists of the processing unit's results, stored in memory and transmitted to an output device.
Here's the real thing (an IBM Personal Computer), just to give one example of how these components may actually look:

Control, processing unit, and memory are housed in one small box.

Input is entered from keyboard. Disk drives provide extra memory storage.

Other common input/output devices (not pictured) are a modem for sending and receiving signals over the phone, and a printer, for producing output on paper.

Let's start in the middle, with the Processing Unit:

In the kitchen, a chef may display a rich repertoire of processing possibilities:

- Braise
- Broil
- Sauté
- Roast
- Steam
- Poach
- Fry
- Sauté

But, as the great Escoffier himself has remarked, all cooking techniques are combinations of simpler steps: the application of more or less heat, wet or dry, etc...

Likewise, all the power of the computer depends on a couple of elementary operations.
O.K... O.K... no more beating around the bush with culinary metaphors...

The computer's elementary operations are logical...

What's a logical operation, you ask? A logical question, considering how much easier it is to think of illogical operations, like amputation of the thumbs or getting out of bed on Mondays...

Nothing is logical on Mondays...

To everyone's good fortune, logic isn't as hard as it used to be. In Aristotle's time, the subject was divided into inductive and deductive branches, inductive logic being the art of inferring truths by observing nature, while deductive logic deduced truths from other truths:

1. You are a man.
2. All men are mortal.
3. Therefore, you are mortal.

Anem? How do you know all men are mortal??

Medieval

Logicians compounded the confusion with six "modes": a statement was either true, false, necessary, contingent, possible, or impossible.

Necessary is to contingent as true is to false... possibly...

Their reasoning grew so mindless that the medieval logician Duns Scotus has been immortalized in the word "dunce"!

Their reasoning grew so mindless that the medieval logician Duns Scotus has been immortalized in the word "dunce"!
The subject was stretched to absurd lengths by Lewis Carroll:

1. Gentiles have no objection to pork.
2. Nobody who admires poetry ever reads Hogg's poems.
3. No mandarin knows Hebrew.
4. Everyone who does not object to pork admires turnstiles.
5. No Jew is ignorant of Hebrew. Therefore, no mandarin ever reads Hogg's poems. * *

* FROM SYMBOLIC LOGIC

Clearly, it was time to simplify the subject...

The step was taken by George Boole (1815 - 1864), an English mathematician who built an "algebra" out of logic.

That is, he made logic fully symbolic, just like math. Sentences were denoted by letters and connected by algebraic symbols — an idea going back to Leibniz, who had dreamed of "justice by algebra."

We can't possibly describe Boole's algebra in its entirety. We'll limit ourselves to three words:

Boole looked at the very connective tissue of language: the words "and", "or", and "not".

(1-x)(1-y) = 1-x-y+xy. Therefore, 50 years!
Suppose \( P \) is any statement. For example,

\[ P \equiv \text{"The pig has spots."} \]

According to Boole, this sentence is either true (T) or false (F). No other option is allowed! *  

Now let \( Q \) be another statement—likewise true or false:

\[ Q \equiv \text{"The pig is glad."} \]

Now form the compound sentences:

- \( P \) AND \( Q \): The pig is spotted AND the pig is glad.
- \( P \) OR \( Q \): The pig is spotted OR the pig is glad.

When are these statements true?  

* In some versions of logic, more than two truth values are permissible.

There are four possible combinations of truth and falsehood for \( P \) and \( Q \):

- \( P \) true, \( Q \) true
- \( P \) false, \( Q \) true
- \( P \) true, \( Q \) false
- \( P \) false, \( Q \) false

**AND**  
"The pig is glad AND has spots."

This is true only in the one case in which \( P \) and \( Q \) are both true. This is summarized in a truth table:

\[
\begin{array}{ccc}
P & Q & P \text{ AND } Q \\
T & T & T \\
T & F & F \\
F & T & F \\
F & F & F \\
\end{array}
\]

**OR**  
"The pig is glad OR has spots."

This is true in the three cases for which either one of the statements \( P \), \( Q \) is true.

\[
\begin{array}{ccc}
P & Q & P \text{ OR } Q \\
T & T & T \\
T & F & T \\
F & T & T \\
F & F & F \\
\end{array}
\]
AND ONE MORE LOGICAL OPERATOR—

**NOT**

**NOT**\(_P\) = The pig is NOT spotted.

This operator simply turns a statement into its opposite.

<table>
<thead>
<tr>
<th>(P)</th>
<th><strong>NOT</strong>(_P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Boole made this look algebraic in something like the following way:

\* Denote **T** by 1
\* Denote **F** by 0
\* Denote **AND** by \(\cdot\)
\* Denote **OR** by \(\oplus\)
\* Denote **NOT** by \(\neg\)

Then the truth tables become:

\[
\begin{array}{c|c|c|c}
  P & Q & P \& Q & P \lor Q \\
  \hline
  1 & 1 & 1 & 1 \\
  1 & 0 & 0 & 1 \\
  0 & 1 & 1 & 1 \\
  0 & 0 & 0 & 0 \\
\end{array}
\]

EXCEPT FOR THE ONE \(\oplus\) \(\oplus\) EQUATION \(\{\oplus\} = 1\), THESE LOOK LIKE ORDINARY ARITHMETIC... WITH **AND** PLAYING THE ROLE OF **TIMES** AND **OR** IN THE ROLE OF **PLUS.**

AND “NOT” IN THE ROLE OF \(\neg\) PLUSED?

**DOO.**

WE'RE NEVER GOING TO USE THE SYMBOLS **AND** \(\oplus\)... YOU CAN FORGET ABOUT THEM... BUT USING 1 AND 0 TO REPRESENT TRUE AND FALSE IS VERY USEFUL... SO FROM NOW ON WE'LL WRITE TRUTH TABLES LIKE THIS:

\[
\begin{array}{c|c|c|c|c|c|c|c}
  P & Q & P \& Q & P \lor Q & P \oplus Q & P \neg Q \\
  \hline
  1 & 1 & 1 & 1 & 0 & 0 \\
  1 & 0 & 0 & 1 & 1 & 0 \\
  0 & 1 & 0 & 1 & 1 & 0 \\
  0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

From these relationships, Boole built up an entire algebra, using only the numbers 0 and 1... Today this Boolean algebra is used all the time by computer engineers—only they express it as electrical circuits...
The key is the **automatic switch**, which is either open or closed, as a logical proposition is either true or false.

**An automatic switch has two wires coming in and one going out**.

This is the **input wire**, which signals the switch to close.

This is the **output wire**.

**This wire acts solely as a power supply.**

- **When no current flows through the input wire, the switch remains open**, as pictured above. When an input signal arrives, however, the electronic equivalent of a miniature boxing glove "punches" the switch closed, resulting in an output signal.

What is the output when two switches (A, B) are arranged in series, one after the other? [In our diagram, please note the rearrangement of wires, made for convenience of illustration.]

The current can flow only if both switches are closed—i.e., when input signals arrive simultaneously at A and B.

Writing 1 for current and 0 for no current, we can then write this input-output table. Look familiar? It should! It's identical to the truth table for AND!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

That's why this arrangement of switches is called an **AND-GATE**.

AND—it has its very own symbol. →
TWO SWITCHES CONNECTED IN PARALLEL BEHAVE LIKE LOGICAL OR: CURRENT CAN PASS FROM POWER TO OUTPUT IF EITHER SWITCH A, B IS CLOSED (OR IF BOTH ARE).

---

**This is the OR-GATE**

AND ITS SYMBOL IS: \[ \text{A} \quad \bigcirc \quad \text{B} \quad \Rightarrow \quad \text{A OR B} \]

---

NOT IS NOT ANY MORE DIFFICULT... IT USES A SPECIAL SWITCH THAT REMAINS CLOSED UNTIL AN INPUT SIGNAL OPENS IT — JUST THE REVERSE OF AN ORDINARY SWITCH:

0 \rightarrow \text{POWER} \rightarrow 1 \rightarrow \text{POWER} \rightarrow A \rightarrow \text{OUTPUT} \rightarrow 0

---

THIS KIND OF SWITCH IS CALLED AN INVERTER, AND IT HAS A SYMBOL, TOO:

\[ \text{A} \quad \bigtriangleup \quad \text{NOT-A} \]

---

AN EVERYDAY EXAMPLE SHOWS HOW THESE SIMPLE GATES CAN MAKE LOGICAL DECISIONS.

YOU KNOW THOSE BUZZERS THAT GO OFF WHEN YOU START YOUR CAR AND YOUR SEAT BELT ISN'T FASTENED? THE KIND THAT'S SPECIALLY DESIGNED TO PENETRATE HUMAN BONE?

---

WELL, THAT'S BECAUSE THE SEAT BELT AND IGNITION ARE CONNECTED BY AN AND-GATE! LIKE SO:

\[ \text{IGNITION} \quad \Rightarrow \quad \text{BUZZER} \quad \text{SEAT BELT} \]

THAT IS, IF THE IGNITION IS ON AND THE SEAT BELT IS NOT, THE BUZZER SOUNDS. PRETTY LOGICAL, NO?

---

CAN YOU THINK OF ANY EXAMPLES OF OR-GEATES IN DAILY LIFE?

(HEY ABOUT A SMOKE ALARM TRIGGERED BY EITHER OF TWO DIFFERENT DETECTORS?)

---

NOT AT THE MOMENT... I'M BUSY!
Here are a few warm-up exercises for chasing through logic diagrams:

**Do the input-output (I/O) tables:**

1. ![Diagram 1]
2. ![Diagram 2]
3. ![Diagram 3]
4. ![Diagram 4]
5. ![Diagram 5] (Note: Only one input!)
6. ![Diagram 6] (Ditto)

(7) What is output when A=1, B=0, C=1?

(8) Complete the I/O table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design logic diagrams with these I/O tables.

(9) ![Diagram 9]
10. ![Diagram 10]
11. ![Diagram 11]
12. ![Diagram 12]

Logic gates have only one or two inputs and a single output—but computer components have many inputs and outputs with complicated input/output behavior.

The wonderful fact is that any input/output table can be produced by a combination of logic gates!

To do it, you need multiple-input logic gates.

Here's a 4-input AND-gate:

- ![Diagram 13]
- ![Diagram 14]

This means E=1 if A=B=C=D=1, and E=0 otherwise. The gate can be made with four switches in series:

- ![Diagram 15]

Similarly, there's a multiple-input OR-gate:

- ![Diagram 16]

It can actually be made from an AND-gate and some inverters:

- ![Diagram 17]
As an example of how to produce a given input/output table, let's solve problem #12:

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Begin by finding all rows where \( C = 1 \).

The table says \( C = 1 \) if \( A = 1 \) and \( B = 0 \) or \( A = 0 \) and \( B = 1 \). \( C = 0 \) otherwise.

Writing \( \bar{A} \) for \( \text{NOT-A} \), this amounts to saying:

\( C = 1 \) if \( A = 1 \) and \( B = 1 \) or \( \bar{A} = 1 \) and \( B = 1 \).

\( C = 0 \) otherwise.

In other words,

\[ C = (A \land B) \lor (\bar{A} \land B) \]

To draw the circuit, run the input wires and their negatives in one direction —

A and B

— and attach the gates to the appropriate wires.

Exactly the same method works for more inputs. For example:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Again, find all rows with output = 1.

Note all possible input combinations!

In this case:

\[ D = (A \land B \land C) \lor (\bar{A} \land B \land \bar{C}) \lor (\bar{A} \land B \land C) \lor (A \land B \land C) \]

Run the inputs and their negatives across the page, attach and-gates, then run them through an or-gate!

To repeat: by the same method, you can produce any input/output table!
**The question:***

Is there some natural way to represent numbers using only 0's and 1's? Can the operations of arithmetic be built out of logic?

**The answer:***

(Which goes back to our old pal Leibniz!)

As sure as I didn't steal calculus from Newton!

Our decimal system, based on ten, was a result of our having ten fingers — an accident of nature! Binary numbers are what would have evolved if we'd been born with two fingers, like the tree sloth.

I'd count by fours, but I only have one free paw!

Tree sloths always count in binary!
LOOK AT THE SYMBOL "10" — "ONE-ZERO." FORGET THAT IT USUALLY MEANS TEN! FORGET IT! STOP CALLING IT THAT! IS THERE ANYTHING THERE THAT SAYS "TEN"? NO!! IT'S JUST A ONE FOLLOWED BY A ZERO — IN AND OF ITSELF, IT HAS NOTHING TO DO WITH TEN!!!

THE SYMBOL ONLY MAKES "TEN" FLASH THROUGH YOUR MIND BECAUSE YOU'VE ALWAYS CALLED IT THAT... IT'S LIKE A RITUAL: PERFORM IT OVER AND OVER AND IT BECOMES AUTOMATIC!

IN ACTUALITY, "10" MEANS:

<table>
<thead>
<tr>
<th>1 (ONE) HANDFUL* AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (ZERO) FINGERS LEFT OVER</td>
</tr>
</tbody>
</table>

*REMEMBER - ON P. 24, WE AGREED TO CALL TEN FINGERS, NOT FIVE, A HUMAN HANDFUL!

Since we humans have ten fingers, our "10" is ten... but to an organism with, say, eight fingers, 10 would mean eight!

In the case at hand, with just two fingers in a handful... 10 means two!

So we can write:

\[ 10_{\text{binary}} = 2_{\text{decimal}} \]

**Note:** Do not read this as "Ten equals two." "One-zero in binary"

Two Two Two

Two Two Two
LIKEWISE, 100—"ONE-ZERO-ZERO"—MEANS
A HANDFUL OF HANDFULS.

IN DECIMAL, THAT'S 10 x 10,
OR A HUNDRED. WELL, IN
BINARY IT'S 10 x 10 ALSO—but
THAT ONLY AMOUNTS TO
FOUR!

1000 IS

\[ 10 \times 10 \times 10 = 2 \times 2 \times 2 = 8 \]

AND GENERALLY,

1 FOLLOWED BY N ZEROS IS:

\[ 2^n \]

N TIMES

("TWO TO THE NTH POWER").

IN THE COMPUTER AGE, EVERYONE WILL
BE REQUIRED BY LAW TO MEMORIZE
THE POWERS OF TWO, UP TO 2^64.
P Don't WAST! AVOID JAIL AND
DO IT NOW!

\[ \begin{align*}
1 &= 2^0 = 1 \\
10 &= 2^1 = 2 \\
100 &= 2^2 = 4 \\
1000 &= 2^3 = 8 \\
10000 &= 2^4 = 16 \\
100000 &= 2^5 = 32 \\
1000000 &= 2^6 = 64 \\
10000000 &= 2^7 = 128 \\
100000000 &= 2^8 = 256 \\
1000000000 &= 2^9 = 512 \\
10000000000 &= 2^{10} = 1024
\end{align*} \]

ALL OTHER BINARY NUMBERS—101, 1100, 111110, AND EVERY OTHER
PATTERN OF 0S AND
1S—is A SUM OF SUCH
POWERS OF TWO! IT'S COMPLETELY ANALOGOUS TO DECIMAL.

\[
\begin{align*}
\text{IN DECIMAL:} & \quad \text{IN BINARY:} \\
497 &= 111110001 \\
400 &= 100000000 \\
+ 90 &= 10000000 \\
+ 7 &= 100000 \\
& + 10000 \\
& + 1000 \\
& + 100 \\
& + 10 \\
& + 1
\end{align*}
\]

\[ 256 \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \]

\[ \frac{497}{4} = \]


\[
\begin{array}{cccccccccc}
& & & & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[ 256 + 64 + 32 + 16 + 8 + 2 = 282 \]

NOW YOU DO IT: CONVERT TO DECIMAL:

110101010110
TO MAKE THIS A BIT MORE CONCRETE —
HERE'S HOW TO COUNT UP FROM 1 IN BINARY.
IT'S JUST LIKE COUNTING IN DECIMAL, ONLY EASIER.
IN DECIMAL, TO COUNT PAST A 9,
YOU WRITE 0 AND CARRY 1. IN BINARY,
YOU HAVE TO CARRY 1 EVERY OTHER NUMBER!!

EASY AS FALLING OUT OF BED!

BINARY
0
1
10
11
100
101
110
111
1000
1001
1010
1011
1100
1101
1110
1111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

ETC!

AS YOU MAY HAVE NOTICED, BINARY NUMBERS
GET LONGNNNNNG VERY FAST!

THIS MAKES THEM HARD FOR US HUMANS TO USE WITHOUT MAKING MISTAKES — BUT FOR COMPUTERS THEY'RE IDEAL!!

BINARY CALCULATION IS SIMPLE.
THERE ARE ONLY FIVE RULES TO REMEMBER:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10
AND THE HANDY FIFTH RULE:
1 + 1 + 1 = 11

AS OPPOSED TO 100 SUMS IN DECIMAL: 9+6, 7+5, 9+3, 8+4, 4+6, ETC ETC ETC!!

TO ADD TWO BINARY NUMBERS, PROCEED PLACE BY PLACE FROM RIGHT TO LEFT, CARRYING A 1 WHEN NECESSARY.
HERE'S A STEP-BY-STEP EXAMPLE:

THE CARRIES

A FEWSUMS TO PRACTICE ON:

100   11   11001   11011   111111111
+ 1   + 1  + 1001  + 1011  + 1111111

WHAT IS THE RESULT OF ADDING A BINARY NUMBER TO ITSELF?

[21]
Another wonderful fact about binary:

**Subtraction is done by adding!!**

The method is called using "two's complement." First you invert the number to be subtracted, so that all its 1's become 0's and vice versa. Then add the two numbers and add 1 to the sum. Ignore the final carry and that's the answer!

![Binary Subtraction Example]

Binary multiplication—and any multiplication—may also be done by repeated addition: To multiply A x B, just add A to itself B times. Likewise, division can be done by repeated subtraction.

The computer can do all arithmetic by adding!!

---

**The Adder**

Before showing how to combine logic gates into a binary adder, we need a bit of terminology.

**Bit** is an abbreviation of "binary digit." It refers to a single 0 or 1.

Is it a binary digit or a binary digit?

It's very common to group bits eight at a time; and any string of eight bits is called a byte. There are 2^8, or 256, possible bytes, from 00000000 to 11111111.
NOW LET'S SEE WHAT AN ADDER MIGHT LOOK LIKE.

TO SAVE DRAWING, WE'LL MAKE IT A FOUR-BIT ADDER, CAPABLE OF ADDING TWO 4-BIT NUMBERS, OR "NIBBLES." (YES, THEY'RE REALLY CALLED THAT!)

\[
\begin{array}{c}
A = 1110 \\
B = 1011 \\
\hline
11001
\end{array}
\]

THE INPUT OF OUR ADDER MUST CONSIST OF EIGHT BITS, FOUR FOR EACH Nibble. THE OUTPUT MUST BE FIVE BITS, THAT IS, A Nibble PLUS ONE BIT FOR A POSSIBLE CARRY. LIKE SO:

\[
\begin{array}{c}
A \\
B \\
A + B
\end{array}
\]

ANY NUMBER FROM ZERO TO FIFTEEN

HOW TO PROCEED? ONE WAY IS TO MAKE A GIANT TRUTH TABLE, MATCHING EVERY POSSIBLE COMBINATION OF INPUTS WITH THE CORRECT OUTPUT, AND CONSTRUCTING A HUGE STEW OF ANDS AND ORS TO FORCE A SOLUTION. THIS IS POSSIBLE, BUT THE COMPLEXITY OF THE JOB MIGHT MAKE YOU THROW UP YOUR HANDS.

OR JUST THROW UP, IF YOU HAVE NO HANDS!

INSTEAD, RECALL HOW ADDITION WORKS IN PRACTICE: COLUMN BY COLUMN, WITH A CARRY BIT CARRYING OUT OF ONE COLUMN AND INTO THE NEXT:

\[
\begin{array}{c}
1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
\hline
1 & 1 & 0 & 0 & 1
\end{array}
\]

SO IT SHOULD BE POSSIBLE TO MAKE A 4-BIT ADDER OUT OF FOUR 1-BIT ADDERS!

THE 1-BIT ADDER MUST HAVE THREE INPUTS - ONE FOR EACH OF THE TWO SUM AND CARRY IN - AND TWO OUTPUTS - ONE SUM BIT AND ONE CARRY-OUT BIT.

FOUR OF THESE CAN THEN BE HOOKED UP TO PRODUCE A 4-BIT ADDER.

NOTE: 8 INPUTS AND 5 OUTPUTS, AS PROMISED!
The input/output table for the 1-bit adder:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Carry In</th>
<th>Carry Out</th>
<th>Sum Bit</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Now there's nothing to it! Remember, logic gates can be rigged up to produce any input/output table. In this case, just treat each output column separately:

A

B

Carry In

Carry Out

Sum Bit

You can add two numbers of any length by hooking together enough 1-bit adders.

The implication of the last two sections is that binary is the "natural" system for encoding numbers in a machine made of on/off switches. Even so, computers use several variations on the basic idea.

Integers, or whole numbers — if they aren't too large — are encoded in straight binary. For instance, 185 would become 10111001.

Floating point representation is for large or fractional numbers. For example, 19.700, 030.2 would be encoded as the binary equivalent of 197.5, meaning 197 × 10^2. Floating point representation often involves rounding off.

Binary coded decimal represents a number in decimal, but with each digit encoded in binary. 967, for instance, would become 1001 0110 0111 9 6 7.

126
AND WHAT ABOUT NON-NUMERICAL INFORMATION—THE ALPHABET, PUNCTUATION MARKS, OTHER SYMBOLS, AND EVEN THE BLANK SPACE??

SINCE THERE IS NO NATURAL WAY TO ENCODE THESE INTO 0S AND 1S, COMPUTER SCIENTISTS INVENTED AND ADOPTED A STANDARD CODE BY MUTUAL AGREEMENT:

ASCII,
THE AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE.

(Actually, ASCII is used by everyone but IBM, which has its own code, called EBCDIC.)

↑ Thus, the letter “t” is encoded as 1010100... etc!
↑ The first two columns contain symbols for such things as "start of heading" (SCH) and other textual directions.

TO ENCODE AND DECODE DATA, COMPUTERS USE LOGIC DEVICES CALLED, NATURALLY ENOUGH, ENCODERS AND DECODERS.

AN ENCODER
USUALLY HAS MANY INPUTS AND A FEW OUTPUTS. A SINGLE INPUT SIGNAL PRODUCES A PATTERN OF OUTPUTS. FOR EXAMPLE, A COMPUTER KEYBOARD IS ATTACHED TO AN ENCODER WHICH TRANSLATES A SINGLE KEYSTROKE INTO ITS ASCII CODE.

A DECODER
WORKS THE OTHER WAY AROUND, TRANSLATING A PATTERN OF BITS INTO A SINGLE OUTPUT SIGNAL. ONE DECODER CONVERTS A BINARY NIBBLE INTO A DECIMAL DIGIT. ANOTHER TRANSFORMS A SPECIFIED LOCATION, OR ADDRESS, INTO MEMORY INTO A SIGNAL TO THAT MEMORY CELL. (SEE P. 155.)
Once alphanumeric information is encoded in binary strings, it is ready to be processed by the computer's most elaborate combination of logic gates, the **Arithmetic Logic Unit** (or ALU, for short).

**Data In** → **Data Out**

- **Byte #1**
- **Byte #2**

**Function Select**

- **IT CONTAINS AN ADDER!**
- **It contains an adder!**

**Carry Bit**

- **Output Byte**

The function select inputs determine which arithmetic or logical function the ALU is to perform, each function having its own binary code. For example, 0001 applied to function select might mean **ADD**, in which case another function (0101, say) might mean **COMPARE** two bytes, bit by bit, and output a 1 wherever they agree. (Meanwhile, the adder takes a nap.)

You can get an idea of a fancy ALU's capabilities from the list on page 181.
THE ALU WOULD BE A COMPLETE CENTRAL PROCESSING UNIT, EXCEPT FOR ONE THING: IT'S UNABLE TO STORE RESULTS. RETURNING TO THE COOKING ANALOGY, WE MIGHT SAY THE ALU LACKS "COUNTER SPACE." WHERE WOULD GRANDMA BAGGAGE BE Without SOMEPLACE TO SET DOWN HER SPAGHETTI?

ALTHOUGH THE ALU CAN PERFORM MIRACLES OF INPUT/OUTPUT, IT CAN'T REMEMBER ANYTHING, AND THAT'S WHERE FLIP-FLOPS COME IN...

VERSATILE AS THEY MAY BE, THE LOGICAL COMBINATIONS WE'VE BEEN SKETCHING STILL HAVE NO MEMORY. THEIR OUTPUT CONTINUES ONLY AS LONG AS THE INPUT IS APPLIED.

I CAN'T REMEMBER A THING!

AND YET— THERE IS A WAY TO HOOK THESE LOGICAL BUT SEMI-GATES TOGETHER INTO A GADGET THAT HOLDS AN OUTPUT INDEFINITELY: THE FLIP-FLOP. STARE AT THIS A MINUTE!!
Besides the strange way a flip-flop eats its own tail, please note the unfamiliar gate used in the construction. It's called a **NAND gate**, which is merely an abbreviation of "not-and."

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>NAND</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Now what happens when the input changes?

Supposing we begin with the input \((s=1, r=0)\), what does changing it to \((s=1, r=1)\) do to the flip-flop's output?

The answer is: **nothing!** The lower NAND gate's input becomes \((0, 1)\), so its output \(\bar{q}\) is still 1, so \(q\) remains 0.

Now for the flip-flop in action:

Suppose the input is \(s=1, r=0\).

<table>
<thead>
<tr>
<th>S</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>(\bar{q})</td>
</tr>
</tbody>
</table>

Then \(q\) must be 1, because NAND outputs 1 if either input is 0. Coupling this back to the upper gate gives \(q=0\).

1. OK, great! But where's the memory?
2. Yeah, what?

1. Great. But where's the memory?
2. Oh, it's the flip-flop remembers!

A little weird, isn't it? The same input \((s=r=1)\) can produce two different outputs, depending on the previous input!
The way a flip-flop is used is this: It begins by setting there with a constant input of \((S=1, R=1)\) and an output of god-knows-what:

\[
\begin{array}{c}
S & 1 & 1 \\
R & 1 & 0 \\
\hline
Q & ? & ?
\end{array}
\]

And I'm damned if I'll tell!

You set the flip-flop [i.e., make \(Q=1\)] by flashing a 0 momentarily down the S-wire, and then returning it to 1:

\[
\begin{array}{c}
S & 1 & 1 \\
R & 1 & 0 \\
\hline
Q & 0 & 1
\end{array}
\]

Or you can reset it [make \(Q=0\)] by flashing a 0 down the R-wire, then returning it to 1:

\[
\begin{array}{c}
S & 1 & 1 \\
R & 1 & 0 \\
\hline
Q & 0 & 1
\end{array}
\]

In either case, as long as \((1,1)\) keeps coming in, the flip-flop will maintain its output until it's changed with another incoming 0.

The only input combination we haven't checked is \((R=1, S=0)\). It's easy to verify that it produces output of \(Q=Q=0\):

\[
\begin{array}{c}
\text{what happens when the input returns to \((1,1)\)?} \\
\text{0} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1}
\end{array}
\]

The answer is not so clear: It depends on which output happens to flop first! (One of them must.)

If \(Q\) is first to change, we get:

\[
\begin{array}{c}
S & 1 & 0 \\
R & 1 & 1 \\
\hline
Q & 0 & 1
\end{array}
\]

If \(Q\) flops first, however:

\[
\begin{array}{c}
S & 1 & 0 \\
R & 1 & 1 \\
\hline
Q & 1 & 0
\end{array}
\]

Since there is no way of knowing which of these will actually happen, and we don't want our flip-flops in random states, the input \((1,0, R=0)\) is disallowed.

We can summarize the basic "RS" flip-flop like so:

\[
\begin{array}{c|c|c}
S & R & Q \\
\hline
1 & 1 & \text{no change} \\
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & \text{disallowed}
\end{array}
\]

Flip-flop inputs are always arranged to make certain the disallowed state cannot arrive.
A LITTLE EXERCISE:

A NOR-GATE is a shorthand way of writing "NOT OR," i.e.

A BASIC RS FLIP-FLOP may also be made out of NOR-GATES:

1. What is the output when R=0, S=1?
   When S=0, R=1?

2. What happens when each of these input conditions changes to R=0, S=0?

3. What is the output when R=1, S=1? What happens when this changes to R=0, S=0?

4. What input combination must be disallowed?

5. If R=0, S=0, how do you set this flip-flop (i.e., make Q=1)? How do you reset it?

BY THE WAY, A FLIP-FLOP IS ALSO CALLED A LATCH, BECAUSE IT "LOCKS IN" DATA.

REGISTERS, COUNTERS, & GLITCHES:

IF THE FLIP-FLOP IS A DEVICE FOR STORING ONE BIT, A REGISTER STORES SEVERAL BITS SIMULTANEOUSLY. IT'S LIKE A ROW OF BOXES, EACH HOLDING ONE BIT.

A ROW OF FLIP-FLOPS SHOULD DO THE JOB....

...SORT OF! BUT IF YOU TRY AND MAKE THIS WORK BY HOOKING UP SOME INPUTS TO RS FLIP-FLOPS, YOU MAY FIND YOURSELF GROWING CONFUSED!
The solution is to add a "gating network" to the basic R-S flip-flop.

Here "D" stands for data, and "E" stands for enable. Note that the gating network makes it impossible for R and S to be zero simultaneously.

When E=1, then R=D and S=\overline{D} (NOT-D). Hence, the value of D is stored at Q. In other words, E=1 enables the bit D to be loaded into the flip-flop.

When E=0, S and R both become 1, and the flip-flop does not change. That is, E=0 blocks the arrival of more data.

Computers are black boxes made of black boxes made of black boxes...

So—in the spirit of ignoring the inner workings once they're understood or even without ever understanding them—we incorporate the gating network into the box, and draw the gated latch like so...

Then here's a parallel register: not the only kind of register, but a genuine member of the breed!

Now what controls the "enable" input?
AS SOON AS YOU BEGIN STORING DATA, QUESTIONS OR TIMING ARISE: HOW LONG DO YOU STORE IT? WHEN DO YOU MOVE IT? HOW DO YOU SYNCHRONIZE SIGNALS? THESE ISSUES ARE SO CRITICAL THAT LOGIC WITH MEMORY IS CALLED SEQUENTIAL, TO DISTINGUISH IT FROM THE PURELY COMBINATIONAL LOGIC OF MEMORY-LESS NETWORKS. TO KEEP THE SEQUENTIAL LOGIC IN STEP, ALL COMPUTERS HAVE CLOCKS!

THE CLOCK'S PULSE IS THE COMPUTER'S HEARTBEAT—ONLY INSTEAD OF A WARM, RAGGED HUMAN HEARTBEAT, LIKE THIS:

\[ \begin{align*}
&\text{ONE PULSE} \rightarrow \text{ONE CYCLE} \\
\end{align*} \]

ONE CLOCK PULSE IS THE BURST OF CURRENT WHEN CLOCK OUTPUT = 1. ONE CYCLE IS THE INTERVAL FROM THE BEGINNING OF A PULSE TO THE BEGINNING OF THE NEXT. DEPENDING ON THE COMPUTER, THE CLOCK FREQUENCY MAY BE HUNDREDS OF THOUSANDS TO BILLIONS OF CYCLES PER SECOND!

SLOW COMPUTER:

\[ \begin{align*}
&\text{FAST COMPUTER:} \\
\end{align*} \]

THE IDEA OF USING A CLOCK IS THAT THE COMPUTER'S LOGICAL STATE SHOULD CHANGE ONLY ON THE CLOCK PULSE. IDEALLY, WHEN THE CLOCK HITS 1, ALL SIGNALS MOVE; THEN STOP ON CLOCK = 0. THEN GO... THEN STOP... THEN GO...

\[ \begin{align*}
&\text{GO STOP GO STOP...} \\
\end{align*} \]

A TYPICAL EXAMPLE IS TO ATTACH THE CLOCK TO THE "ENABLE" INPUT OF A GATED LATCH, IN WHICH CASE THE LATCH BECOMES KNOWN AS A "D FLIP-FLOP."

\[ \begin{align*}
&\text{THEN A NEW BIT OF DATA IS LOADED AT EVERY CLOCK PULSE!} \\
\end{align*} \]

UNfortunately things are rarely ideal! IT TAKES A NON-ZERO TIME FOR A SIGNAL TO PASS ALONG A WIRE, SO THINGS ARE NEVER PERFECTLY SYNCHRONIZED. FOR EXAMPLE, SUPPOSE AN AND GATE ONE INPUT IS CHANGING FROM 1 TO 0, AND THE OTHER FROM 0 TO 1:

\[ \begin{align*}
&\text{IF A CHANGES AFTER B, THE OUTPUT WILL HAVE AN UNWANTED PULSE:} \\
&\text{THAT PULSE IS A GLITCH, AND GRIEVE AS IT IS, IT CAN CAUSE A FLIP-FLOP TO FLOP!} \\
\end{align*} \]

\[ \begin{align*}
&\text{WE'RE UNAVOIDABLE!} \\
\end{align*} \]
The glitch is defeated by the **master-slave** flip-flop:

- DATA
- CLOCK

The inverted clock signal to the slave flip-flop delays the data input from arriving at the slave until the **end** of a clock pulse, after all glitches have died out. For example, suppose we want to load the bit 1 into the flip-flop.

As usual, we draw the whole thing as a single box!

---

**Shift Registers**

Stringing a number of master-slave flip-flops together makes a **shift register**

- DATA
- CLOCK

Data enter a shift register one bit at a time, shifting to the right with each new clock pulse.

**For example, the nibble 1101 would enter the shift register like this:**

- Data enters slave at clock = 0.

Each clock pulse brings a new bit into the register. (Why doesn’t the bit travel all the way through on one pulse? Because of the master-slave flip-flops!)

Likewise, the nibble shifts out one bit at a time.

**Shift registers are useful when information is to be transmitted serially, or one bit at a time.**
Finally, a special kind of register: the **counter**.

Is that like the counter Monte Carlo?

A counter is just what it sounds like: something that counts. In other words, it's a register that increments itself—adds 1 to its contents—whenever a "count" signal arrives.

---

**Diagram:**

- **ON E=1, Q goes in at D:**
  - ![Diagram of circuit with E and Q](image)
  - Then Q reverses once for each cycle of E—in other words, Q **toggles** just half as often as E.

---

**Text:**

Described in that way, a counter sounds easy to make: just combine an adder with a register! This would in fact work, but there's an even slicker way, based on another fancy flip-flop. Consider this master-slave flip-flop, coupled back on itself:

- **Diagram:**
  - ![Diagram of master-slave flip-flop](image)

As usual, we abbreviate the whole circuit by this simpler box. The "T" is for toggle, to indicate that the flip-flop toggles whenever T=1. Then here's our counter: each flip-flop toggles at half the rate of the one to its left.

- **Table:**
  - | COUNT IN | D | C | B | A |
  - | --- | --- | --- | --- |
  - | 0 | 0000 | 0000 | 0000 | 0000 |
  - | 1 | 0001 | 0001 | 0001 | 0001 |
  - | 2 | 0010 | 0010 | 0010 | 0010 |
  - | 3 | 0011 | 0011 | 0011 | 0011 |
  - | 4 | 0100 | 0100 | 0100 | 0100 |
  - | 5 | 0101 | 0101 | 0101 | 0101 |
  - | 6 | 0110 | 0110 | 0110 | 0110 |
  - | 7 | 0111 | 0111 | 0111 | 0111 |
  - | 8 | 1000 | 1000 | 1000 | 1000 |

---

**Text:**

Friller than I am!

It counts from 0000 to 1111!
A FEW ITEMS OF NOTE:

1. This counter is called an "asynchronous ripple counter," because the count ripples through from one flip-flop to the next. This causes a slight delay before the count is registered.

2. When the 16th count pulse arrives, the counter returns to 0; to go higher than 15, more flip-flops are needed.

3. The Nth flip-flop in a ripple counter divides the incoming pulse by 2^N. This is the principle on which digital watches are based: a high-frequency internal clock pulse is divided to a rate of precisely 1 cycle per second.

4. Internal output:

   ![Clock Signal](image)

   There are also synchronous counters, which register all bits simultaneously, and counters which return to 0 on any preassigned number. In any case, from now on, a counter is just another black box!

EXERCISES

The Amazing NAND:

1. Show that
   - A is the same as
   - B is the same as
   - A and B is the same as

Conclude that all logic can be derived from the single relation NAND!!

2. Can the same be done with NOR?

3. Show that
   - A is the same as
   - B is the same as
   - A and B is the same as

Redraw the adder on p. 146 using only NAND gates.

4. Given a 4-bit shift register,
   - Show its contents after each of four clock pulses as the nibble 0101 is entered.

5. How would you attach a buzzer to a counter to sound when the count hits 21 (10101 in binary)? Hint: look at the shift register on p. 109.

6. Convince yourself that attaching inverters to the outputs makes a counter count backwards.
Now in case you're feeling strangled by spaghetti—

The tangled diagrams on the preceding pages were never intended to trace the complete wiring diagram of any computer. Rather, they are meant to demonstrate how the computer's essential functions—math, comparison, decoding, data selection and storage—all depend on simple logic.

Now that you presumably believe in the power of logic, no more wiring diagrams are needed!

Outward to higher levels!
In the infancy of electronic computing, memory was always more expensive than sheer computing power. Plenty of processing could be done with relatively few components, but every increase in memory simply meant more—more actual, physical places to store things!

Since then, research into memory technology has brought down the cost considerably. For a few hundred dollars you can buy a micro with over 64,000 bytes of memory, compared with ENIAC’s memory of about 100 numbers—at a cost of millions!!

And a human’s billions of neurons costing? *ENIAC did not compute in binary.*

The same research effort, however, has produced a bewildering array of memory types and technologies!!

Card memories, tape memories, drum, disk, bubble, optical, core, charge-coupled device, and semiconductor memories; volatile and non-volatile, dynamic and static, destructive and non-destructive; read-write, read-only, programmable read-only, erasable programmable read-only...pant...ruff.

Have I forgotten anything? Have I forgotten anything? I don’t remember.

Well, one has to begin somewhere!!
An important distinction exists between electronic and electromechanical memory devices.

Electronic memories, with no moving parts, are as fast as the rest of the computer. Electromechanical memories have moving parts, like disks or reels of tape. This makes them slow—how slow depending on the type of memory.

You found that file yet?

Electronic memories' speed makes them ideal for the computer's main or internal memory, while electromechanical memories are used for auxiliary storage outside the machine.

Electromagnetic memories compensate for their slowness with a gigantic capacity. One hard disk can store up to ten million bytes, compared with a typical micro's main memory of 65,536 (2^16) bytes.

Internal memory can be thought of as a simple grid, with a cell at each intersection. Depending on the computer, each cell can hold one byte, two bytes, or more.

Every cell has a unique address, specifying where it sits in the grid.

In practice, there may be many such grids, in which case the address specifies the grid number, as well as the row and column within it.

Note: Do not confuse a cell's address with its contents!!
WHAT IS THE MAXIMUM NUMBER OF CELLS THE COMPUTER CAN ADDRESS? THIS DEPENDS ON THE LENGTH AND STRUCTURE OF THE COMPUTER'S "WORDS." FOR EXAMPLE, A 32-BIT MACHINE MAY INTERPRET THE FIRST 8 BITS AS AN INSTRUCTION...

...AND THE REMAINING 24 BITS AS AN ADDRESS.

IN THAT CASE, ADDRESSES CAN BE ANYTHING BETWEEN

\[ 00000 \ldots 0 \]

AND

\[ 111\ldots1 = 2^{24} - 1 \]

giving \(2^{24}\) possible memory cells.

16,777,216, to be exact!

AN 8-BIT MICRO, ON THE OTHER HAND, MIGHT PROCESS THREE BYTES IN SUCCESSION:

\[ \text{O1111101} \text{ AN INSTRUCTION} \]

\[ \text{10101010} \text{ THE FIRST HALF OF AN ADDRESS,} \]

\[ \text{00010100} \text{ AND THE SECOND HALF OF AN ADDRESS.} \]

HERE THE ADDRESS IS 16 BITS LONG, GIVING \(2^{16} = 65,536\) possible addresses.

TO MAKE ADDRESSES SHORTER AND MORE READABLE, THEY'RE OFTEN EXPRESSED IN HEXADECIMAL, OR BASE-16, NUMERALS.

\[ 10_{\text{hex}} = 16_{\text{decimal}} \]

\[ 100_{\text{hex}} = 16^2 = 256 \]

\[ 1000_{\text{hex}} = 16^3 = 4096 \]

\[ \ldots \]

ETC!

JUST AS BASE-10 NUMBERS REQUIRE THE DIGITS 0-9, SO HEXADECIMAL NEEDS DIGITS FROM 0 TO FIFTEEN. THE EXTRA DIGITS ARE REPRESENTED BY THE LETTERS A-F:

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEX</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

FOR EXAMPLE:

\[ 4A0D_{\text{hex}} = \]

\[ 4 \times 16^3 + 10 \times 16^2 + 0 \times 16 + 13 \times 1 \]

18,957 \text{ decimal}

TO CONVERT BINARY TO HEX:

GROUP THE BINARY NUMBER INTO NIBBLES, STARTING FROM THE RIGHT. CONVERT EACH Nibble TO A HEX DIGIT:

\[ 101110 \quad 0101 \quad 1011 \]

5 C B

TO CONVERT HEX TO BINARY, JUST REVERSE THE PROCESS.
FROM THE HARDWARE POINT OF VIEW, THERE ARE THREE MAIN TYPES OF INTERNAL MEMORY.

**CORE**

Memories use little magnetic doughnuts — "cores." Each core can be electrically magnetized in one of two directions, representing 0 and 1.

AND TWO SEMICONDUCTOR MEMORIES:

**RAM**

Uses flip-flops to store bits — so each memory cell is essentially a (parallel) register.

**ROM**

Indicates a 1 or 0 at each grid point by the presence or absence of an electric connection there.

**RAM STANDS FOR** "RANDOM ACCESS MEMORY," MEANING THAT ANY CELL CAN BE ACCESSED DIRECTLY. ROM AND CORE MEMORIES ALSO PROVIDE RANDOM ACCESS, BUT FOR SOME REASON RAM HOGGED THE NAME!

**A CASE OF SPECIES CONFUSION...**

**ROM STANDS FOR** "READ-ONLY MEMORY."

A practical difference between them is that you can only read what's in ROM, while with RAM you can read things out or write them in with equal ease. When you load a program into the computer, it is stored in RAM.

RAM 1

ROM
Unfortunately, 
RAM IS 
VOLATILE.

적이면, RAM은 
버전이 
아니면.

IT FORGETS EVERYTHING WHEN THE POWER 
IS TURNED OFF.

For example, I own a battery-powered pocket 
computer with 1680 bytes of RAM. It can store up 
to ten programs even when you turn it off. 
Because it keeps some electricity running through 
memory.

But when 
the battery 
dies... 
EVE-EVE, 
PROGRAMS!

RAM VOLATILITY IS ONE 
REASON THAT THE 
MAGNIFICENT, INFALLIBLE 
COMPUTER IS VULNERABLE 
TO THE VACABIES OF 
OUTMODED, ERRATIC POWER 
GENERATING STATIONS!

ROM — "READ-ONLY MEMORY"— 
ONCE ITS CONTENTS ARE ENTERED, 
CAN NEVER BE REWRITTEN.* 
ORDINARILY, ROM IS PROGRAMMED 
AT THE FACTORY, BUT THERE ARE 
NOW ALSO PROMS — PROGRAMMABLE 
ROMS — WHICH CAN BE CUSTOM- 
PROGRAMMED TO THE USER’S 
SPECIFICATIONS.

*EXCEPT FOR 
EPROM — 
ERASABLE 
PROGRAMMABLE 
ROMS — BUT WE 
DON'T GET 
INTO THAT!

WHAT ARE 
YOU DOING 
OUT PROM 
IGHT? 
GOING 
TO RADD 
SHACK 
TO GET 
FIXED.

UNLIKE RAM, ROM 
IS NON-VOLATILE: 
IT KEEPS ITS 
CONTENT EVEN 
WITHOUT POWER. 
AFTER ALL, IT’S 
NOTHING BUT A 
HUGE GRID OF WIRES 
WITH PHYSICAL 
CONNECTIONS AT SOME 
INTERSECTIONS, 
THE CONNECTIONS 
REMAIN, REGARDLESS 
OF ELECTRIC CURRENT.

AND WHEN I SAY 
"HUGE," I MEAN 
"THAT."
Some typical uses of ROM:

Most video game cartridges are programmed in ROM. Just plug it in and it's ready to go! But of course, it can't be reprogrammed either...

Many personal computers have thousands of bytes of ROM to store the program which, in turn, allows the machine to "understand" the language called BASIC.

And, as we'll see, ROM plays an important role in the computer's control section.

Behind the explosive growth of RAM and ROM is... the incredible shrinking technology!

Etched on silicon chips, the density of components per chip has been doubling every year!

The standard measure of chip storage is the K, short for "kilo" ("chilo" is Greek for 1000). In computerese it means 2^10, the power of two closest to 1000:

\[ K = 1024 \]

Almost Greek for almost 1000!

The first RAM chip with 1K bits of storage was a sensation — but now 64K is common, and the 256K chip has arrived! What's next?

KkK?

KKk?
Despite the growth of RAM capacity, sometimes it is not the answer to every prayer!!

Show us the way to store more than internal RAM can hold!

Let us protect our data from power losses!

Grant us a program library of frequently used routines!

Discum vobiscum!

Mass storage.

As the name implies, mass storage is memory that can store a lot!! Almost all mass storage devices are non-volatile and have a mechanical component that makes them much slower than electronic random access memories.

For example:

Punch cards. The cards of Jacquard, Babbage, and Hollerith are still in use!

Paper tape. Same idea as punch cards: a hole represents 1, a non-hole 0.

Magnetic tape. Stores bits as small magnetic regions, which may be magnetized in one of two directions, representing 1 or 0.
Faster, less bulky, and the current storage of choice is the **Magnetic Disk**. Disks also store bits as tiny magnetized regions—up to 10 million bytes per disk!

A big computer system usually has multiple disk drives, with phonograph-arm-like read/write heads darting back and forth across the whirling platters.

**Floppies** are small, low-cost magnetic disks made of plastic. They always stay in their jackets, because a speck of dust can create a monster **glitch**!

Like internal memory, mass storage must be organized, or “formatted.” Take the floppy disk for example:

Floppies are formatted into rings and sectors—three rings and eight sectors in this very oversimplified disk. (It's more like 26 sectors and 77 rings in a genuine disk.)

To access a particular block of data, you specify the ring number and sector number. Then the disk drive:

1. Spins the disk until that sector lies under the read/write head
2. Move the head in or out to the proper ring.

This process takes milliseconds—an eternity to a computer!
SOME TYPICAL USES OF MASS STORAGE:

A GERBIL RANCHER, USING A MICROCOMPUTER TO IMPROVE PRODUCTIVITY, BUYS THE APPROPRIATE PROGRAMS (FROM BERBYTE, INC.) STORED ON FLOPPIES.

A GOVERNMENT AGENCY (TAKE YOUR PICK) MAINTAINS FILES ON THE COMPUTER, STORED ON HARD DISK.

AND EVERYBODY'S ON THIS LIST. SO WHO AM I?

A GOVERNMENT AGENCY (TAKE YOUR PICK) MAINTAINS FILES ON THE COMPUTER, STORED ON HARD DISK.

THE PHONE COMPANY STORES IN BURBEE MEMORY THE MESSAGE: "THE NUMBER YOU HAVE REACHED IS NOT IN SERVICE..."

AND EVERYBODY'S ON THIS LIST. SO WHO AM I?

WELL, YOU GET THE PICTURE... NOW IT'S TIME TO MOVE ON...
Along with input/output, memory, and the arithmetic/logic unit, control is the computer's final, critical ingredient. Our old schematic diagram shows the flow of control (→) and information (←).

It helps to redraw this diagram in a way that better reflects a genuine computer design known as "bus architecture."

The vertical arrows, representing electrical pathways a byte or more wide, are the buses. According to signals passed along the control bus, addresses and data get on and off the data/address bus. With the proviso that only one "passenger" can ride the bus at a time.

How are we to imagine this control, from which all dark arrows point away?

As a megalomaniacal robot that can't keep its electronic fingers out of anything?

I must maintain control at all costs.

A wise ruler who judiciously chooses the time for every act?

Go ye, and multiply!

A relentless tyrant who wields a whip hand over repellous glitches?

Well, at least the buses run on time.

Note that all the arrows on the control bus point away from the control section.
Like anyone else, control reveals its character by its behavior... so let's follow what happens in this oversimplified computer, which fleshes out the diagram of two pages back with some essential counters and registers.

Here's what they're for:

<table>
<thead>
<tr>
<th>Program Counter:</th>
<th>Instruction Register:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticks off the instructions one by one.</td>
<td>Holds an encoded version of the instruction being performed.</td>
</tr>
<tr>
<td><strong>“Boil spaghetti ten minutes.”</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address Register:</th>
<th>Accumulator:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holds the address of whatever is to enter or leave memory.</td>
<td>The ALU's main register, keeping a running total of ALU operations.</td>
</tr>
<tr>
<td><strong>“Get me byte 0101!”</strong></td>
<td><strong>“Couldn't solve 1+1 without it!”</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B Register:</th>
<th>C Register:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An auxiliary register to hold numbers on their way to ALU.</td>
<td>Holds data on the way to output.</td>
</tr>
<tr>
<td><strong>Like a motel that rents rooms by the microsecond!</strong></td>
<td><strong>Is there control in the outside world?</strong></td>
</tr>
</tbody>
</table>

In fact, control spends most of its time just moving the contents of these registers around!
To see how control works, let's follow what happens when the computer *adds two numbers*—our very first program!

Like everything about computers, programs can be described at various levels. We begin with **Assembly Language**, which specifies the computer's actual moves, but omits the fine details. At this level, here's how to add two numbers:

0. **Load the first number into the accumulator.**

1. **Add the second number (holding the sum in the accumulator).**

2. **Output the contents of the accumulator.**

3. **Halt.**

To express this in proper assembly language, we must specify the precise location in memory of the two numbers to be added, and condense the wordy statements into mnemonic abbreviations. Suppose, for example, that the numbers are stored at addresses 1E and 1F (hexadecimal). Our program becomes:

0. **LDA 1E**  
   (**Load accumulator with contents of 1E**)

1. **ADD 1F**  
   (**Add contents of 1F**)

2. **OUT**  
   (**Output contents of accumulator**)

3. **HALT**

In general, assembly-language statements have two parts:

- **The operator**, which describes the step to be performed
- **The operand**, which gives the address on which the operator acts

**LDA 1E**

Note however! Some operators don't need an explicit operand. "OUT", for instance, is understood to apply to the accumulator.
Now that we have an assembly-language program, how do we feed it to the machine—which only understands 0's and 1's?

The answer is clear: within the machine, each operator is encoded as a string of bits called its "op-code." Some simple samples:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>OP-CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>001</td>
</tr>
<tr>
<td>ADD</td>
<td>010</td>
</tr>
<tr>
<td>OUT</td>
<td>110</td>
</tr>
<tr>
<td>HALT</td>
<td>111</td>
</tr>
</tbody>
</table>

Then a machine instruction consists of an op-code segment, or "field," followed by an address field giving the operand in binary:

LDA 1E = 001 111110

Op-code field

Address field

So here's our program translated into machine language:

0. LDA 1E 001 11110
1. ADD 1F 010 11111
2. OUT 110 XXXXX?
3. HALT 111 XXXXX?

Any 5 bits are ok for these address fields, as they'll be ignored!

Now, (assuming an input device)

The program steps are read into consecutive memory addresses, beginning with 0. The contents of memory are then:

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001 11110</td>
</tr>
<tr>
<td>1</td>
<td>010 11111</td>
</tr>
<tr>
<td>2</td>
<td>110 00000</td>
</tr>
<tr>
<td>3</td>
<td>111 00000</td>
</tr>
</tbody>
</table>

Note that the program step number is the address where it's stored!

And we also need to enter the data: the two numbers to be added, any two numbers will do, say 5 and 121. They go in addresses 1E and 1F:

1E 00000101
1F 01111001

How can the computer distinguish data from instructions? By assuming everything is an instruction, unless instructed to do otherwise!
Once the program is stored, control can begin execution. In a series of even more primitive steps called microinstructions, one microinstruction occurring with each clock pulse. Are you ready for the gory details?

Control begins by fetching the first instruction: it—

0.0. Moves contents of program counter (00000000 to begin with) to register

0.1. Moves contents of that memory address to instruction register

The instruction register now holds the first instruction. Control "reads" it and—

0.2. Moves the address of the instruction register's address field to register

0.3. Moves contents of that memory address to accumulator

The accumulator is now loaded with the first piece of data. One microinstruction remains

0.4. Increment program counter

A bit confused? Let's go through it again with the next step, ADD.

Again control begins with a "fetch phase":

1.0. Move contents of program counter (now 00000001) to register

1.1. Move contents of that address to instruction register

The instruction in the instruction register, 01011111, causes control to:

1.2. Move address field from instruction register to register

1.3. Move contents of that memory address to B register

1.4. Signal the ALU to ADD and put the sum

Again, there's one more step:

1.5. Increment program counter...
AND FINALLY?

Well, luckily the last two instructions are easier:

2.0 AND 2.1 ARE THE SAME FETCH INSTRUCTIONS AS BEFORE, PUTTING INSTRUCTION 2 ("OUT") IN THE INSTRUCTION REGISTER.

This op-code (110) CAUSES CONTROL TO:

2.2. MOVE CONTENTS OF ACCUMULATOR TO C REGISTER

2.3. INCREMENT PROGRAM COUNTER

Finally, control fetches the instruction 111 ("HALT"), which CAUSES CONTROL TO:

3.2. DO NOTHING

ARE YOU BEGINNING TO SEE WHAT THE BEAST CONTROL REALLY IS??

I guess it has to GIVE UP... SOB!!
IN REAL LIFE THE SITUATION IS MORE COMPLICATED IN DETAIL BUT THE SAME IN PRINCIPLE. THERE ARE MORE REGISTERS, AND OP-CODES ARE LONGER THAN THREE BITS, ALLOWING CONTROL TO RESPOND TO A MUCH LARGER SET OF INSTRUCTIONS. HERE'S THE INSTRUCTION SET OF A GENUINE PROCESSOR, THE MOTOROLA 6800.

ARITHMETIC
ADD
ADD WITH CARRY
SUBTRACT
SUBTRACT WITH CARRY
INCREASE
DECREASE
COMPARE
NEGATE

LOGICAL
AND
OR
EXCLUSIVE OR
NOT
SHIFT RIGHT
SHIFT LEFT
SHIFT RIGHT ARITHMETIC
ROTATE RIGHT
ROTATE LEFT
TEST

DATA TRANSFER
LOAD
STORE
MOVE
CLEAR
CLEAR CARRY
CLEAR OVERFLOW
SET CARRY
SET OVERFLOW

BRANCH
BRANCH IF ZERO
BRANCH IF NOT ZERO
BRANCH IF EQUAL
BRANCH IF NOT EQUAL
BRANCH IF CARRY
BRANCH IF NO CARRY
BRANCH IF POSTIVE
BRANCH IF NEGATIVE
BRANCH IF OVERFLOW
BRANCH IF NO OVERFLOW
BRANCH IF GREATER THAN
BRANCH IF GREATER THAN OR EQUAL
BRANCH IF LESS THAN
BRANCH IF LESS THAN OR EQUAL
BRANCH IF HIGHER
BRANCH IF HIGHER OR EQUAL
BRANCH IF NOT HIGHER
BRANCH IF NOT HIGHER OR EQUAL
BRANCH IF LOWER
BRANCH IF LOWER OR EQUAL

SUBROUTINE CALL
CALL SUBROUTINE

SUBROUTINE RETURN
RETURN FROM SUBROUTINE
RETURN FROM INTERRUPT

MISCELLANEOUS
NO OPERATION
PUSH
POP
WAIT
ADJUST DECIMAL ENABLE INTERRUPT
DISABLE INTERRUPT
BREAK

ONE GROUP OF THESE INSTRUCTIONS DESERVES SPECIAL MENTION: THE BRANCH, OR JUMP, INSTRUCTIONS.

As we'll see, these give the computer a lot of its "intelligence." Their effect is to transfer control to another part of the program. The simplest jump instruction is just plain "jump," as in:

⇒ "JMP 123" CAUSES CONTROL TO ENTER 123 IN THE PROGRAM COUNTER... AND PROCEED WITH THE PROGRAM FROM THERE.

But "smarter" are conditional jumps. They transfer control if some condition is satisfied: For instance, "JMP IF ZERO" MEANS JUMP IF THE ACCUMULATOR HOLDS 0.

⇒ OTHERWISE, DON'T JUMP!

⇒ "JZ 321"
So you see, control is no tyrant at all. It only does what it's told—completely automatically!!

If you really want to imagine the control section's personality, think of a perfectly efficient bureaucrat, acting in strict obedience to the computer's real boss: the program!

"Go to... go to..."
—Shakespeare