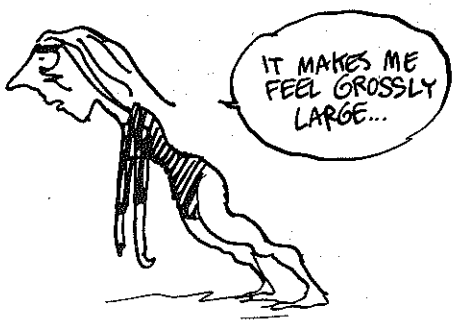
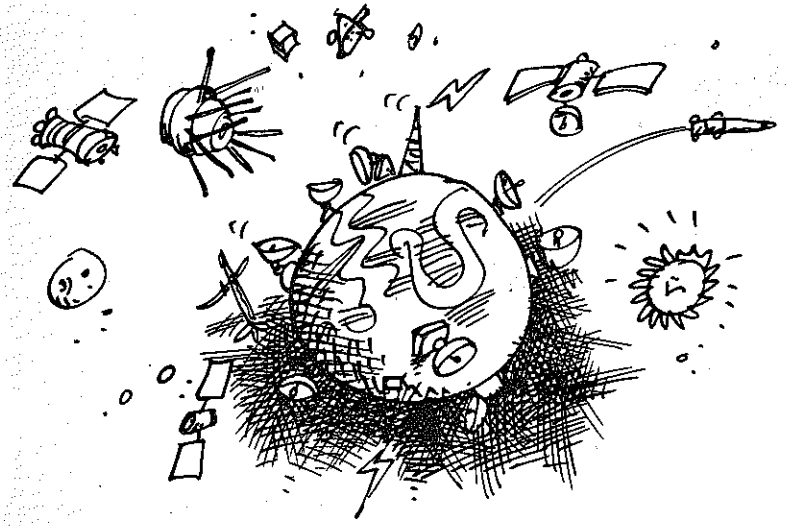
THERE'S NO END IN SIGHT.... NOW WE HAVE MICROS WITH THE POWER OF MINIS, "SUPERMINIS" THAT RIVAL MAINFRAMES, MINIS ON A CHIP... AND THERE'S TALK OF REDUCING COMPONENTS TO MOLECULAR SIZE USING RECOMBINANT DNA TECHNOLOGY...

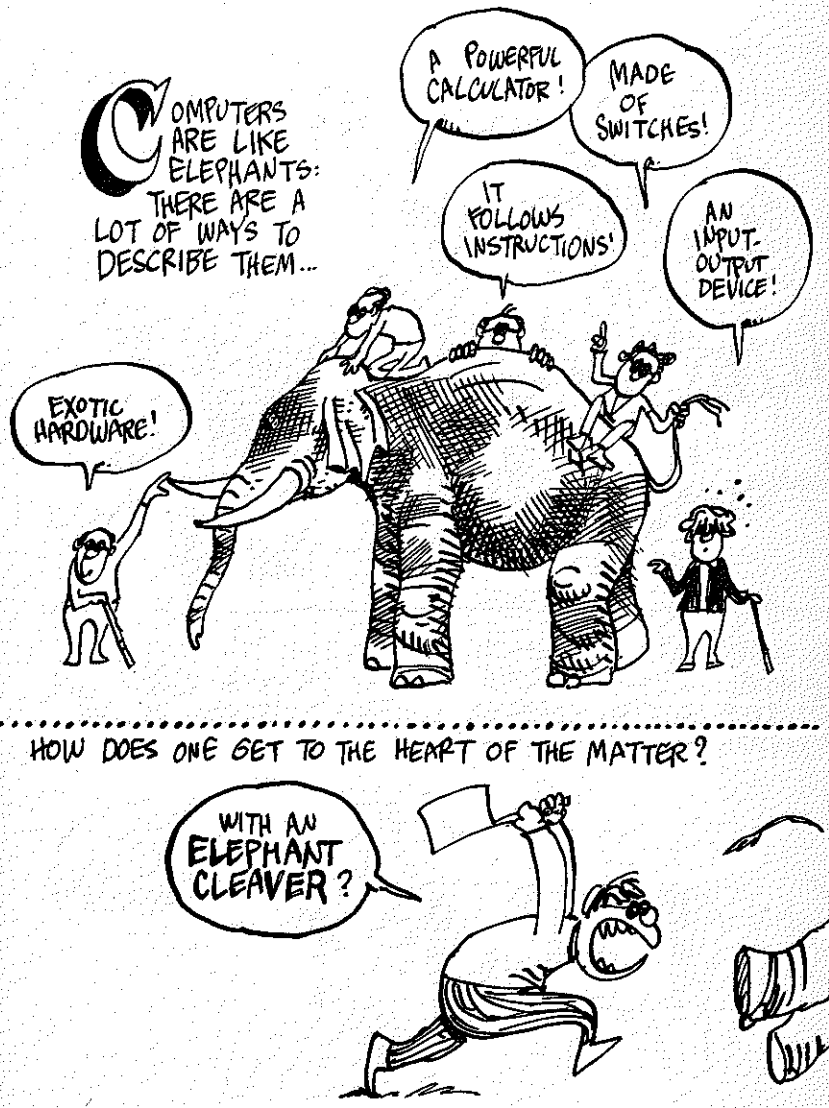


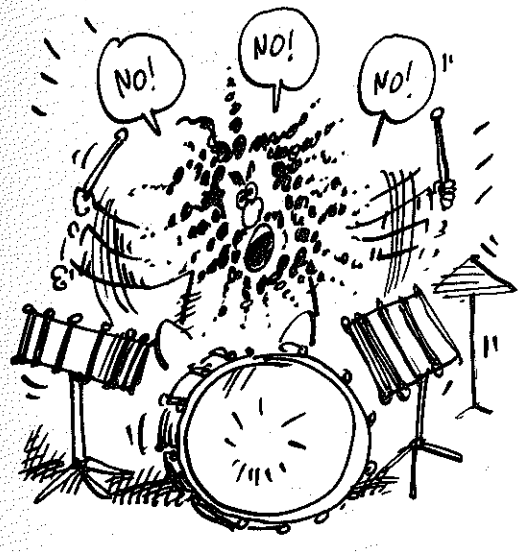
THERE SEEMS TO BE NO SUCH THING AS A COMPUTER WITH TOO MUCH COMPUTING POWER. NO MATTER THE SPEED OR CAPACITY, COMPUTERS ALWAYS FIND JOBS TO DO... AND NO WONDER: THIS IS THE AGE OF EXCESS INFORMATION!



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 COMPARE IT WITH A MORE FAMILIAR PROCESS: **COOKING.** SO STEP INTO GRANDMOTHER BABBAGE'S KITCHEN, AS SHE PREPARES BASIC SPAGHETTI...

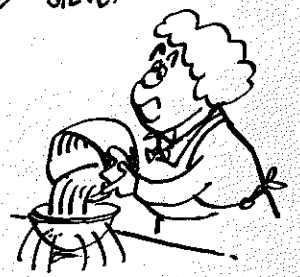
HERE'S THE WORLD FAMOUS RECIPE:

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.



5. SERVE..

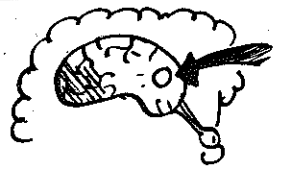
IT'S NOT HARD TO DISTINGUISH A FEW COMPONENTS IN THIS PROCESS:

FIRST, THE INGREDIENTS, OR **INPUT.**

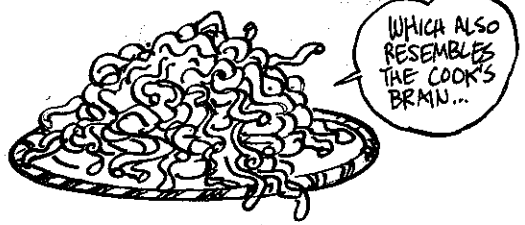
NEXT, THE EQUIPMENT WHICH DOES THE COOKING: HANDS, KETTLE, STOVE, SALTSHAKER, SIEVE, PLATE, SPOON.

THESE FORM THE **PROCESSING UNIT.**

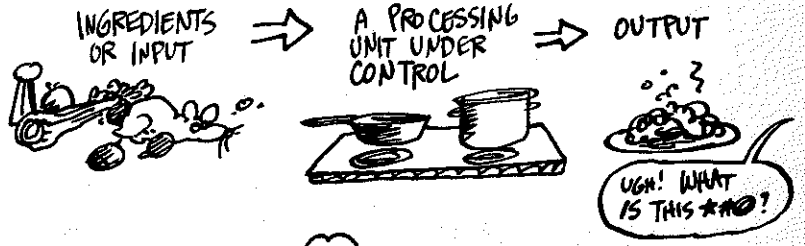
LESS OBVIOUSLY, THERE IS A *PART OF THE COOK'S BRAIN* WHICH **CONTROLS** THE PROCESS. IT MONITORS AND DIRECTS THE STEP-BY-STEP UNFOLDING OF THE RECIPE. THIS IS REFERRED TO AS THE **CONTROL UNIT.**

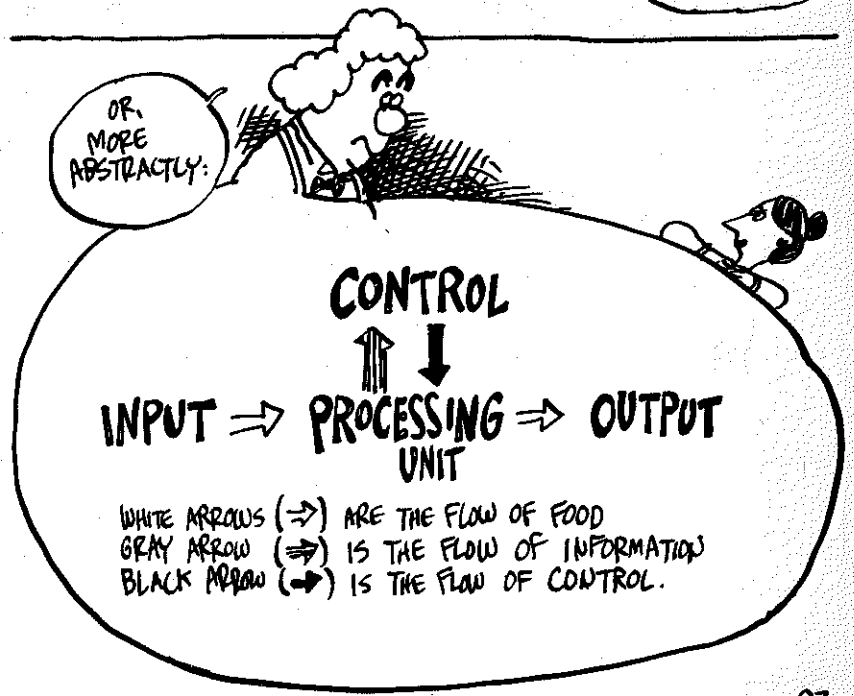


AND OF COURSE THE COMPLETED DISH, OR **OUTPUT.**



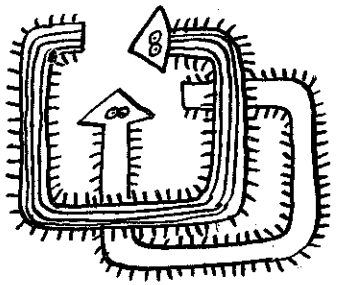
OF COURSE, SPAGHETTI IS NOTHING SPECIAL! ANY RECIPE COULD BE PROCESSED BY THE SAME BASIC STRUCTURE:



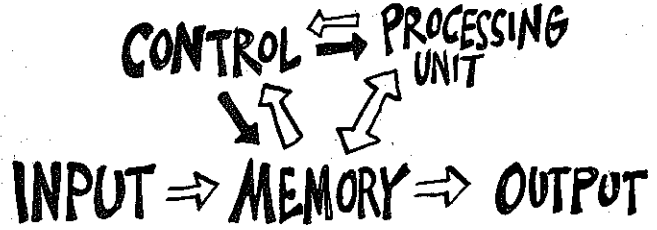


WITH COMPUTERS, THE DIAGRAM IS SLIGHTLY DIFFERENT:

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:

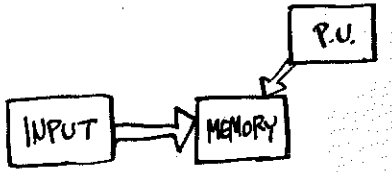


⇒ = INFORMATION FLOW

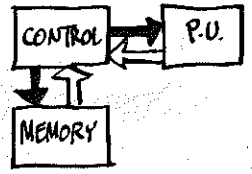
➡ = CONTROL FLOW

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.

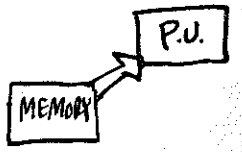
THE **MEMORY** STORES THE INPUT AND RESULTS FROM THE PROCESSING UNIT:



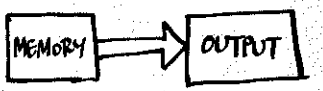
CONTROL READS THE PROGRAM AND TRANSLATES IT INTO A SEQUENCE OF MACHINE OPERATIONS.



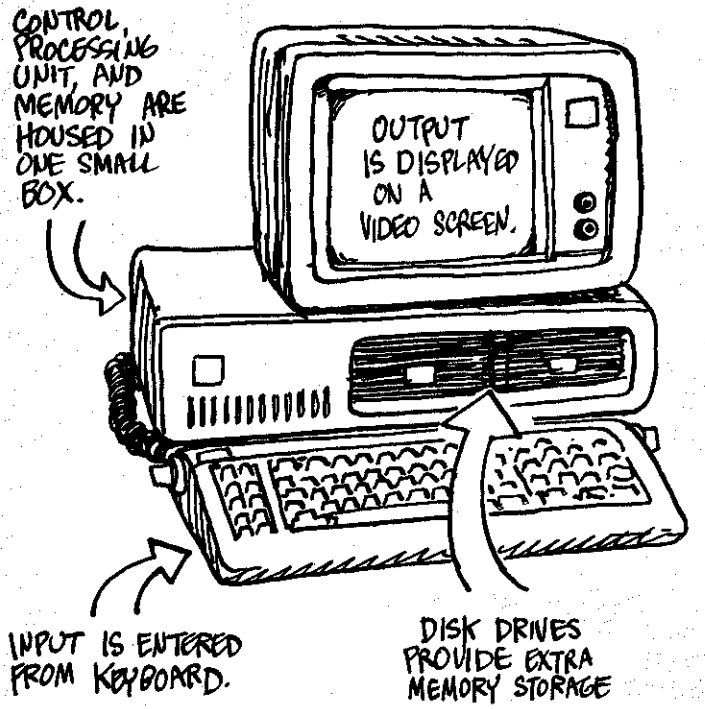
THE **PROCESSING UNIT** PERFORMS THE ACTUAL ADDITIONS, MULTIPLICATION, COUNTING, COMPARISON, ETC, ON INFORMATION RECEIVED FROM MEMORY.



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:

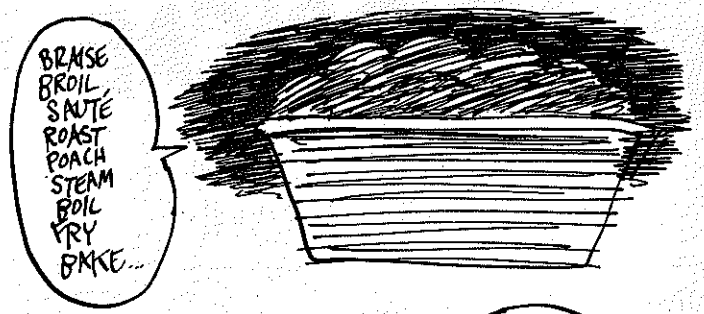


OTHER COMMON INPUT/OUTPUT DEVICES (NOT PICTURED) ARE A ***MODEM***, FOR SENDING AND RECEIVING SIGNALS OVER THE PHONE, AND A ***PRINTER***, FOR PRODUCING OUTPUT ON PAPER.

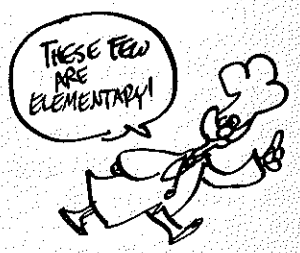
LET'S START IN THE MIDDLE, WITH THE

PROCESSING UNIT:

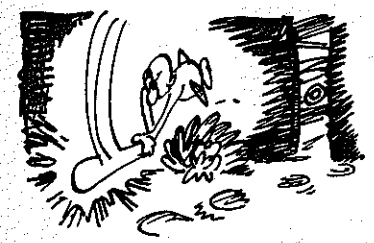
IN THE KITCHEN, A CHEF MAY DISPLAY A RICH REPERTOIRE OF PROCESSING POSSIBILITIES:



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.

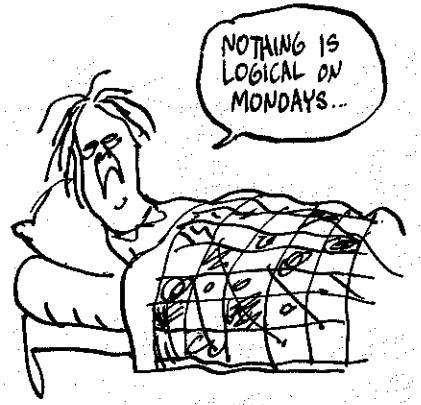


O.K... O.K... NO MORE BEATING AROUND THE BUSH WITH CULINARY METAPHORS...

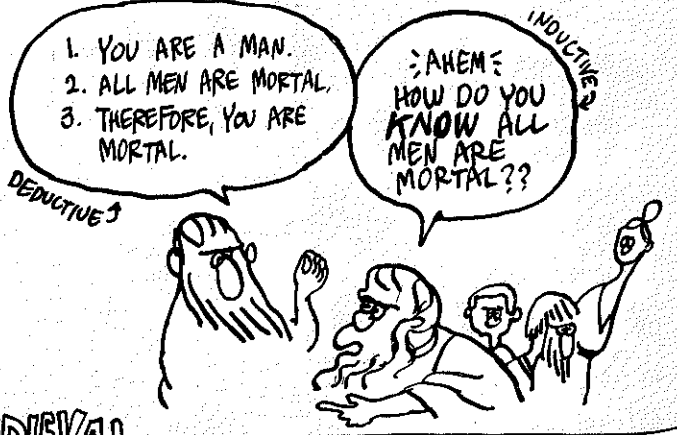
THE COMPUTER'S ELEMENTARY OPERATIONS ARE

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...

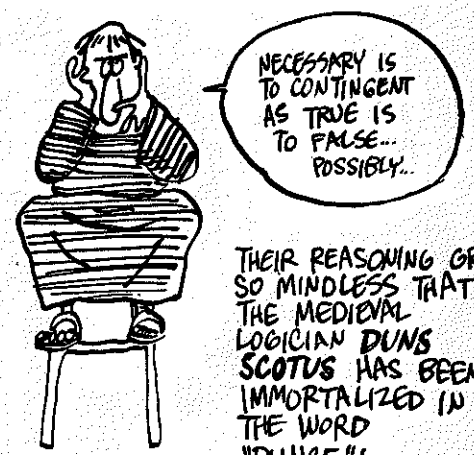


TO EVERYONE'S GOOD FORTUNE, LOGIC ISN'T AS HARD AS IT USED TO BE. IN ARISTOTLE'S TIME, THE SUBJECT WAS DIVIDED INTO INDUCTIVE AND DEDUCTIVE BRANCHES, INDUCTIVE LOGIC BEING THE ART OF INFERRING TRUTHS BY OBSERVING NATURE, WHILE DEDUCTIVE LOGIC DEDUCED TRUTHS FROM OTHER TRUTHS:



MEDIEVAL

LOGICIANS COMPOUNDED THE CONFUSION WITH SIX "MODES": A STATEMENT WAS EITHER TRUE, FALSE, NECESSARY, CONTINGENT, POSSIBLE, OR IMPOSSIBLE.



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 PIGSTIES 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." *

CLEARLY, IT WAS TIME TO SIMPLIFY THE SUBJECT...

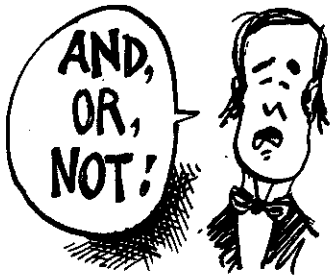
* FROM SYMBOLIC LOGIC

THIS STEP WAS TAKEN BY GEORGE BOOLE (1815-1864), AN ENGLISH MATHEMATICIAN WHO BUILT AN "ALGEBRA" OUT OF LOGIC.

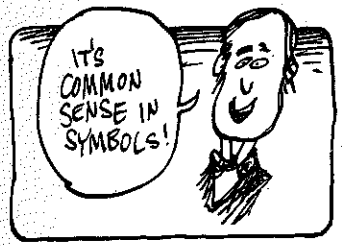
THAT IS, HE MADE LOGIC FULLY SYMBOLIC, JUST LIKE MATH. SENTENCES WERE DENOTED BY LETTERS AND CONNECTED BY ALGEBRAIC SYMBOLS — AN IDEA GOING BACK TO LEIBNIZ, WHO HAD DREAMED OF "JUSTICE BY ALGEBRA."

$(1-x)\cdot(1-y) = 1-x-y+xy$. THEFORE, 30 YEARS!

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".



SUPPOSE P IS ANY STATEMENT... FOR EXAMPLE,

P = "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:

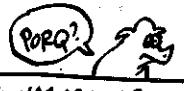
Q = "The pig is glad."

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 SENTENCES 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, Q ARE BOTH TRUE. THIS IS SUMMARIZED IN A *TRUTH TABLE*:

P	Q	P AND Q
T	T	T
T	F	F
F	T	F
F	F	F

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.

P	Q	P OR Q
T	T	T
T	F	T
F	T	T
F	F	F

AND ONE MORE LOGICAL OPERATOR—

NOT

NOT-P = The pig is NOT spotted.

THIS OPERATOR SIMPLY TURNS A STATEMENT INTO ITS OPPOSITE.

P	NOT-P
T	F
F	T

BOOLE MADE THIS LOOK ALGEBRAIC IN SOMETHING LIKE THE FOLLOWING WAY:

- DENOTE *T* BY 1
- DENOTE *F* BY 0
- DENOTE *AND* BY ·
- DENOTE *OR* BY ⊕
- DENOTE *NOT* BY 1−

THEN THE TRUTH TABLES BECOME:

1·1 = 1	1⊕1 = 1	1−1 = 0
1·0 = 0	1⊕0 = 1	1−0 = 1
0·1 = 0	0⊕1 = 1	
0·0 = 0	0⊕0 = 0	

EXCEPT FOR THE ONE WEIRD EQUATION 1⊕1 = 1, THESE LOOK LIKE ORDINARY ARITHMETIC... WITH "AND" PLAYING THE ROLE OF "TIMES" AND "OR" IN THE ROLE OF "PLUS."

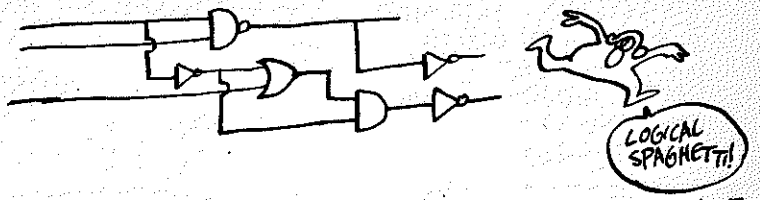
WE'RE NEVER GOING TO USE THE SYMBOLS · AND ⊕... YOU CAN FORGET ABOUT THEM... BUT USING 1 AND 0 TO REPRESENT TRUE AND FALSE IS VERY USEFUL... SO FROM NOW ON WE'LL WRITE TRUTH TABLES LIKE THIS:

P	Q	P AND Q
1	1	1
1	0	0
0	1	0
0	0	0

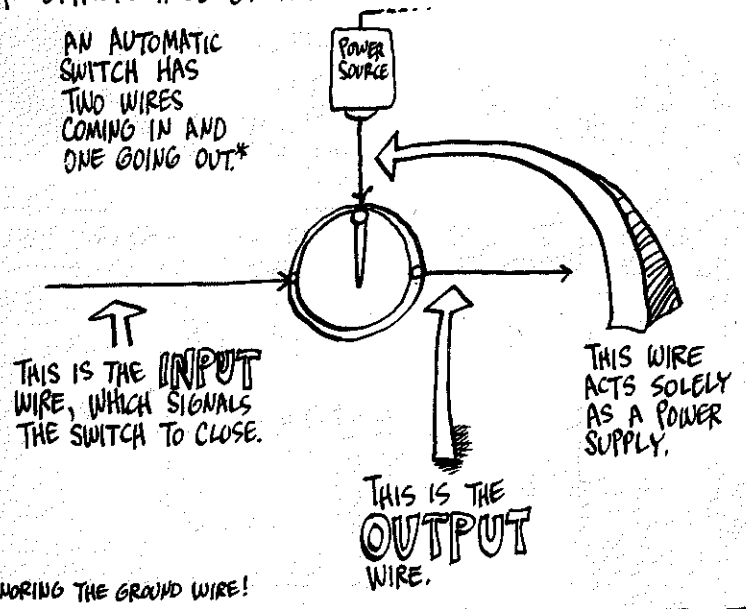
P	Q	P OR Q
1	1	1
1	0	1
0	1	1
0	0	0

P	NOT-P
1	0
0	1

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*...

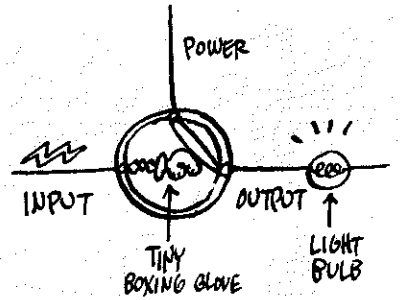


THE KEY IS THE **AUTOMATIC SWITCH**, WHICH IS EITHER OPEN OR CLOSED, AS A LOGICAL PROPOSITION IS EITHER TRUE OR FALSE.

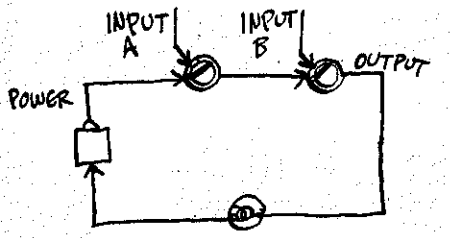


*IGNORING THE GROUND WIRE!

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.

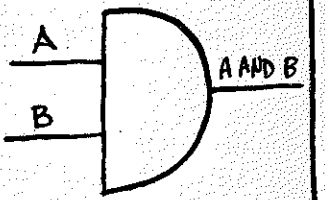
WRITING 1 FOR CURRENT AND 0 FOR NO CURRENT, WE CAN THEN WRITE THIS **INPUT-OUTPUT** TABLE. LOOK FAMILIAR? IT SHOULD! IT'S IDENTICAL TO THE TRUTH TABLE FOR **AND!**

A	B	OUTPUT
1	1	1
1	0	0
0	1	0
0	0	0

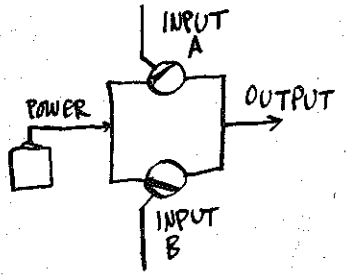
THAT'S WHY THIS ARRANGEMENT OF SWITCHES IS CALLED AN

AND-GATE

AND IT HAS ITS VERY OWN SYMBOL

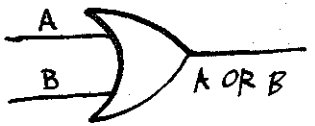


TWO SWITCHES CONNECTED IN PARALLEL BEHAVE LIKE LOGICAL **OR**: CURRENT CAN PASS FROM POWER TO OUTPUT IF EITHER SWITCH A, B IS CLOSED (OR IF BOTH ARE).

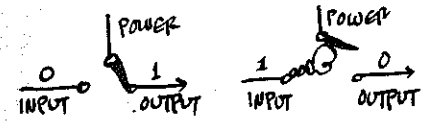


A	B	OUTPUT
1	1	1
1	0	1
0	1	1
0	0	0

THIS IS THE **OR-GATE** AND ITS SYMBOL IS:

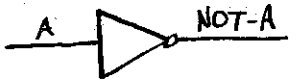


NOT IS **NOT** ANY MORE DIFFICULT... IT USES A SPECIAL SWITCH THAT REMAINS CLOSED UNTIL AN INPUT SIGNAL OPENS IT — JUST THE REVERSE OF AN ORDINARY SWITCH:



A	OUTPUT
1	0
0	1

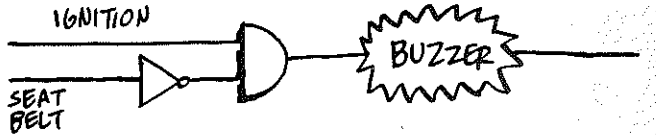
THIS KIND OF SWITCH IS CALLED AN **INVERTER**, AND IT HAS A SYMBOL, TOO:



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:



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

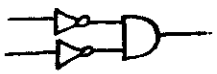
CAN YOU THINK OF ANY EXAMPLES OF OR-GATES IN DAILY LIFE?

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

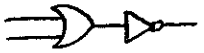
HERE ARE A FEW WARM-UP EXERCISES FOR CHASING THROUGH LOGIC DIAGRAMS:

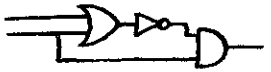
DO THE INPUT-OUTPUT (I/O) TABLES:

(1) (2)



(3) (4)



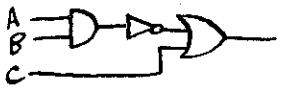


(5) (6)

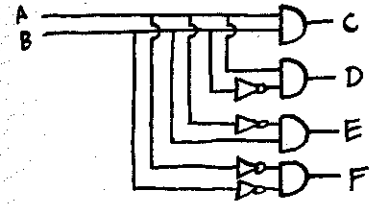
(NOTE: ONLY ONE INPUT!)

(DITTO!)

(7) WHAT IS OUTPUT WHEN A=1, B=0, C=1 ?



(8) COMPLETE THE I/O TABLE:



A	B	C	D	E	F
1	1	1	0	0	0
1	0				
0	1				
0	0				

DESIGN LOGIC DIAGRAMS WITH THESE I/O TABLES.

(9)

IN	OUT
1	0
0	1

(10)

IN	OUT
1	0
0	1

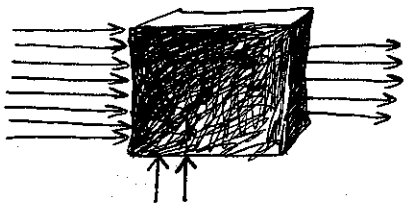
(11)

IN	OUT
1	1
0	0

(12)

IN	OUT
1	0
0	1

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:



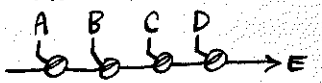
THE WONDERFUL FACT IS THAT **ANY** INPUT/OUTPUT TABLE CAN BE PRODUCED BY A COMBINATION OF LOGIC GATES!

TO DO IT, YOU NEED MULTIPLE-INPUT LOGIC GATES. HERE'S A 4-INPUT **AND**-GATE:

A	B	C	D	E
1	1	1	1	1
1	1	1	0	0
1	1	0	1	0
				⋮
0	0	0	0	0

ALL 0's

THIS MEANS E=1 IF A=B=C=D=1, AND E=0 OTHERWISE. THE GATE CAN BE MADE WITH FOUR SWITCHES IN SERIES:

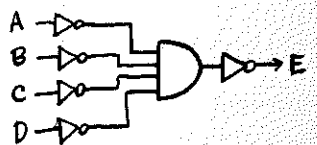


SIMILARLY, THERE'S A MULTIPLE-INPUT **OR**-GATE:

A	B	C	D	E
1	1	1	1	1
1	1	1	0	1
				⋮
0	0	0	1	1
0	0	0	0	0

ALL 1's

IT CAN ACTUALLY BE MADE FROM AN **AND**-GATE AND SOME INVERTERS:



AS AN EXAMPLE OF HOW TO PRODUCE A GIVEN INPUT/OUTPUT TABLE, LET'S SOLVE PROBLEM #12:

IN A	IN B	OUT C
1	1	0
1	0	1
0	1	1
0	0	0

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.

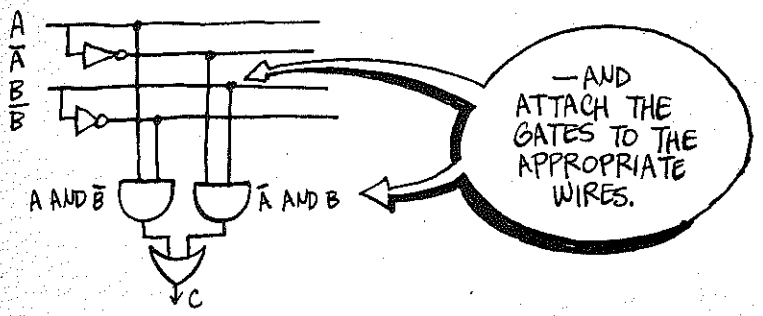
WRITING $\bar{A}$ FOR NOT-A, THIS AMOUNTS TO SAYING

C=1 IF A=1 AND $\bar{B}$=1 OR $\bar{A}$=1 AND B=1.
C=0 OTHERWISE.

IN OTHER WORDS,

$$C = (A \text{ AND } \bar{B}) \text{ OR } (\bar{A} \text{ AND } B)$$

TO DRAW THE CIRCUIT, RUN THE INPUT WIRES AND THEIR NEGATIVES IN ONE DIRECTION —



— AND ATTACH THE GATES TO THE APPROPRIATE WIRES.

EXACTLY THE SAME METHOD WORKS FOR MORE INPUTS. FOR EXAMPLE:

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

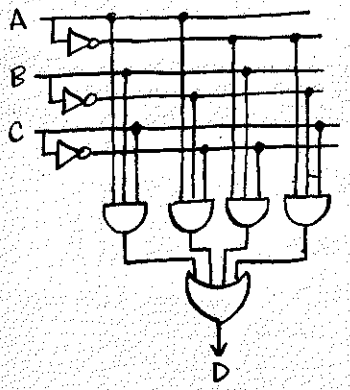
AGAIN, FIND ALL ROWS WITH OUTPUT = 1.

NOTE ALL POSSIBLE INPUT COMBINATIONS!

IN THIS CASE,

$$D = (A \text{ AND } B \text{ AND } C) \text{ OR } (A \text{ AND } \bar{B} \text{ AND } \bar{C}) \text{ OR } (\bar{A} \text{ AND } B \text{ AND } \bar{C}) \text{ OR } (\bar{A} \text{ AND } \bar{B} \text{ AND } C).$$

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

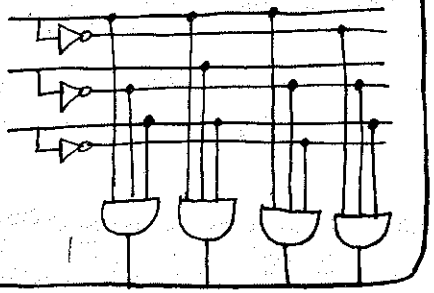
INPUT/OUTPUT TABLE!!

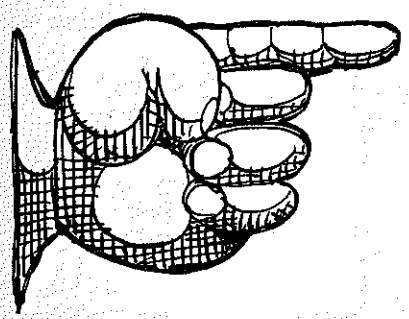
WHERE DOES THIS LEAVE US?

WITH ONE FOOT IN THE SWAMP!

IGNORANCE

BY NOW YOU MAY BE GETTING THE IDEA THAT INFORMATION IS ENCODED INSIDE COMPUTERS AS STRINGS OF 1'S AND 0'S, WHICH CAN BE TRANSFORMED IN ANY WAY WE LIKE BY THE RIGHT COMBINATION OF LOGIC GATES.





BUT WE HAVEN'T REALLY SEEN HOW LOGIC GATES CAN DO THE JOB COMPUTERS WERE DESIGNED FOR:

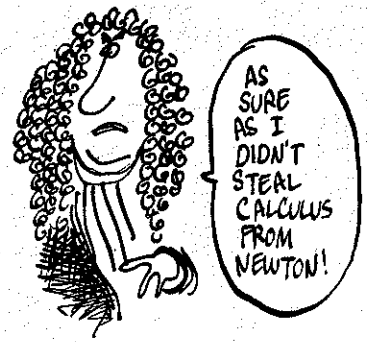
NAMELY: HOW DO COMPUTERS COMPUTE?

The questions:

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):



THE SYSTEM IS CALLED

BINARY NUMBERS.

THEY'RE BASED ON *TWO!*

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!

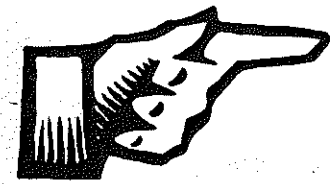
TREE SLOTHS ALWAYS COUNT IN BINARY!

10

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!

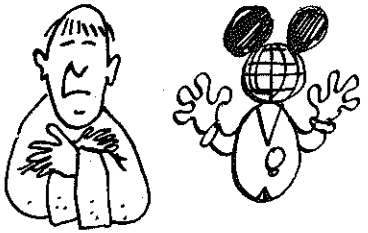
IN ACTUALITY, "10" MEANS:



1 (ONE) HANDFUL* AND

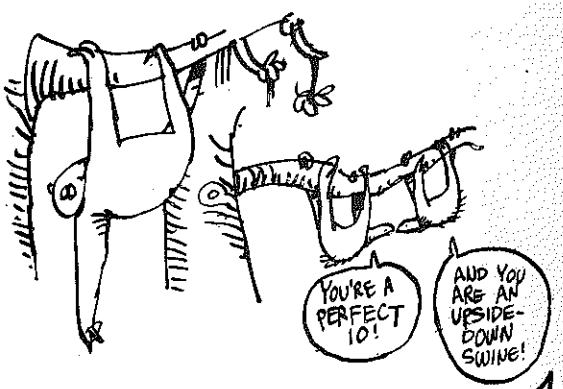
0 (ZERO) FINGERS LEFT OVER

*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!

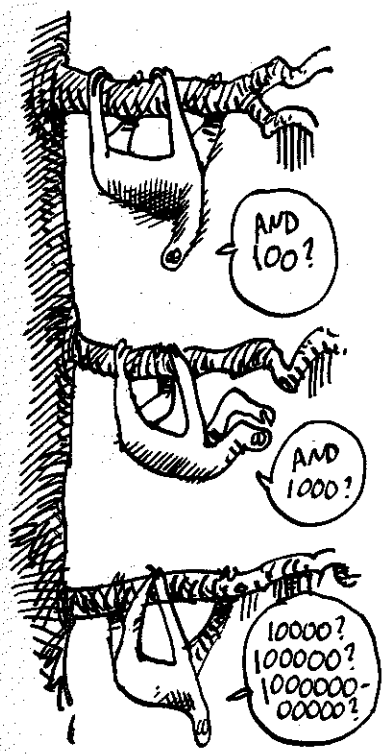
IN THE CASE AT HAND, WITH JUST **TWO** FINGERS IN A HANDFUL... 10 MEANS **TWO!!**



SO WE CAN WRITE:

$$10_{\text{BINARY}} = 2_{\text{DECIMAL}}$$

NOTE: DO **NOT** READ THIS AS "TEN EQUALS TWO." TEN DOES NOT EQUAL TWO!! "ONE-ZERO IN BINARY" EQUALS TWO!!



LIKEWISE, **100** — "ONE-ZERO-ZERO" — MEANS

1 HANDFUL OF HANDFULS.

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

1000 IS

$10\times10\times10 = 2\times2\times2 = 8$

⋮

AND GENERALLY,

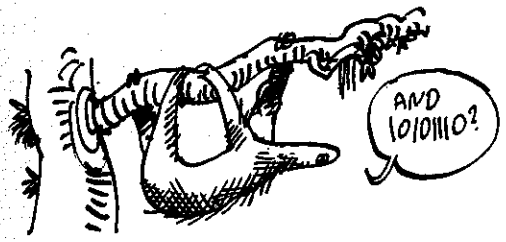
1 FOLLOWED BY N ZEROES IS:

$\underbrace{2\times\cdots\times2}_{N\text{ TIMES}} = 2^N$

("TWO TO THE N^{TH} POWER").

IN THE COMPUTER AGE, EVERYONE WILL BE REQUIRED BY LAW TO MEMORIZE THE POWERS OF TWO, UP TO 2^{10}. BETTER NOT WAIT! AVOID JAIL AND DO IT NOW!

$1 = 2^0 = 1$
$10 = 2^1 = 2$
$100 = 2^2 = 4$
$1000 = 2^3 = 8$
$10000 = 2^4 = 16$
$100000 = 2^5 = 32$
$1000000 = 2^6 = 64$
$10000000 = 2^7 = 128$
$100000000 = 2^8 = 256$
$1000000000 = 2^9 = 512$
$10000000000 = 2^{10} = 1024$



ALL OTHER BINARY NUMBERS — 101, 1111, 11000, AND EVERY OTHER PATTERN OF 0's AND 1's — IS A SUM OF SUCH POWERS OF TWO! IT'S COMPLETELY ANALOGOUS TO DECIMAL.

IN DECIMAL:	IN BINARY:	
497 =	111110001 =	
400	100000000	256
+ 90	+ 10000000	128
+ 7	+ 1000000	64
	+ 100000	32
	+ 10000	16
	+ 1	1
		497

TO TRANSLATE A BINARY NUMBER INTO THE DECIMAL SYSTEM, LIST THE POWERS OF TWO OVER THE CORRESPONDING PLACES, AND ADD THOSE LYING OVER A 1.

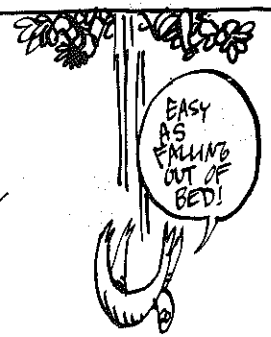
$\cdots$	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	1	0	0	0	1	1	0	1	0		

256 + 16+8 + 2 = 282

NOW YOU DO IT. CONVERT TO DECIMAL:

(1) 11 (2) 101 (3) 1111111 (4) 1101010101101

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!!



BINARY	DECIMAL
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15
10000	16
10001	17
10010	18
10011	19
10100	20
⋮	⋮
ETC!	ETC!

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

1000110101101011000110100111010110110

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:

$0 + 0 = 0$

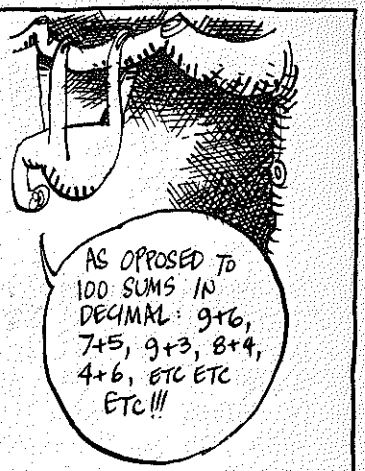
$0 + 1 = 1$

$1 + 0 = 1$

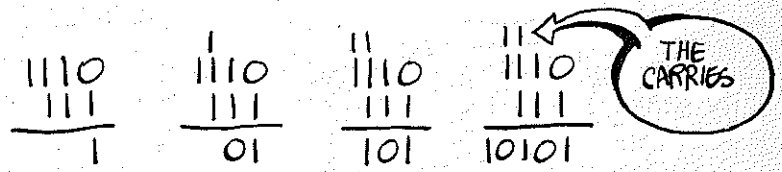
$1 + 1 = 10$

AND THE HANDY FIFTH RULE:

$1 + 1 + 1 = 11$



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:



A FEW SUMS TO PRACTICE ON:

100	11	11001	11011	1111111111
+ 1	+ 1	+ 1100	+ 11011	+ 1111111111

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

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!

E.G. 1101 − 1100

```
  1101
  0011   ← INVERTED
 10000   SUM
+    1   ADD 1
 10001
 ↑ IGNORE
  0001   ← ANSWER.
```

BINARY MULTIPLICATION — AND ANY MULTIPLICATION — MAY ALSO BE DONE BY REPEATED ADDITION: TO MULTIPLY A × B, JUST ADD A TO ITSELF B TIMES. LIKEWISE, DIVISION CAN BE DONE BY REPEATED SUBTRACTION.

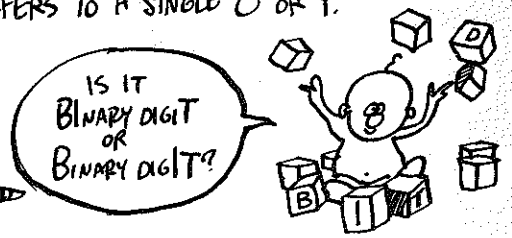
```
 110 × 11 =

  110
 +110   } 11 TIMES
 +110
 1001 0
```

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.

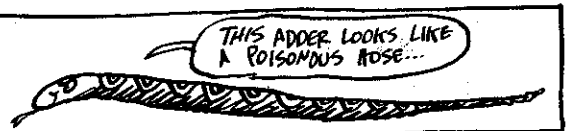
BIT IS AN ABBREVIATION OF "BINARY DIGIT." IT REFERS TO A SINGLE 0 OR 1.



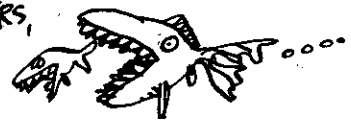
IT'S VERY COMMON TO GROUP BITS EIGHT AT A TIME, AND ANY STRING OF EIGHT BITS IS CALLED A **BYTE**.

THERE ARE 2^8, OR 256, POSSIBLE BYTES, FROM 00000000 TO 11111111.

NOW LET'S SEE WHAT AN ADDER MIGHT LOOK LIKE.

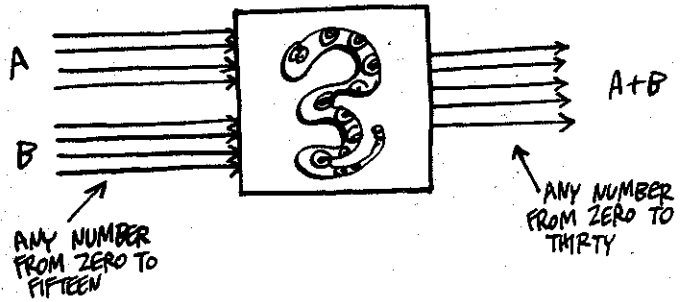


TO SAVE DRAWING, WE'LL MAKE IT A FOUR-BIT ADDER, CAPABLE OF ADDING TWO 4-BIT NUMBERS, OR "NIBBLES." (YES, THEY'RE REALLY CALLED THAT!)



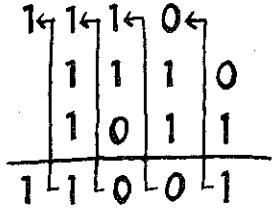
A = 1110
B = 1011
11001

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:



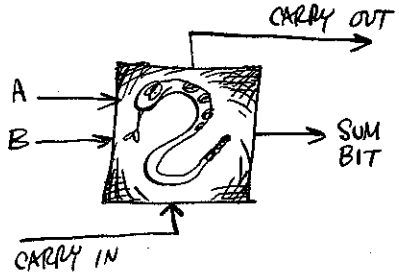
HOW TO PROCEED? ONE WAY IS TO MAKE A GIANT TRUTH TABLE, MATCHING EVERY POSSIBLE COMBINATION OF INPUTS WITH THE CORRECT OUTPUT, AND CONSTRUCTING A HUGE STEW OF ANDs AND NOTs TO FORCE A SOLUTION. THIS IS POSSIBLE, BUT THE COMPLEXITY OF THE TASK MIGHT MAKE YOU THROW UP YOUR HANDS.

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

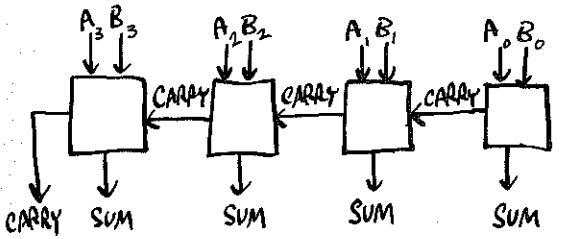


SO IT SHOULD BE POSSIBLE TO MAKE A 4-BIT ADDER OUT OF FOUR 1-BIT ADDERS!

THE 1-BIT ADDER MUST HAVE THREE INPUTS — ONE FOR EACH OF THE TWO SUMMAND BITS AND ONE FOR THE BIT CARRIED 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:

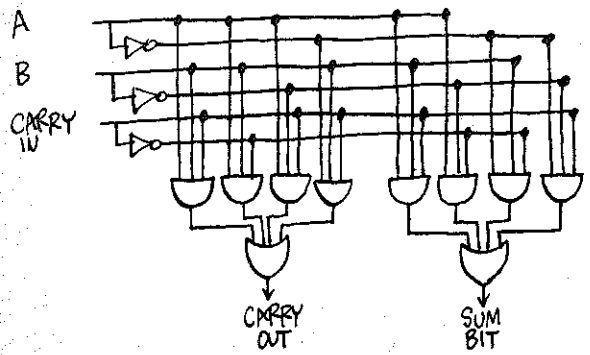


NOTE: 8 INPUTS AND 5 OUTPUTS, AS PROMISED!

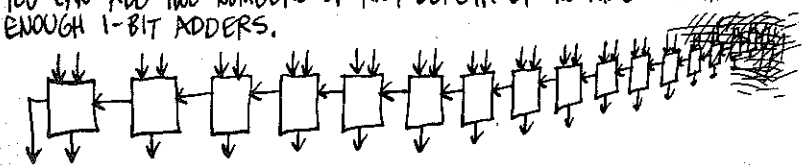
THE INPUT/OUTPUT TABLE FOR THE 1-BIT ADDER:

A	B	CARRY IN	CARRY OUT	SUM BIT
1	1	1	1	1
1	1	0	1	0
1	0	1	0	0
1	0	0	0	1
0	1	1	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

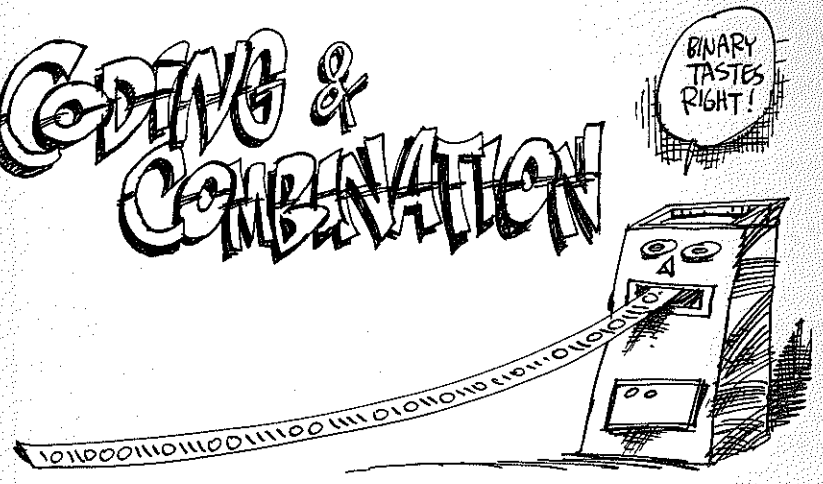
NOW THERE'S NOTHING TO IT! REMEMBER, LOGIC GATES CAN BE RIGGED UP TO PRODUCE ANY INPUT/OUTPUT TABLE. IN THIS CASE, JUST TREAT EACH OUTPUT COLUMN SEPARATELY:



YOU CAN ADD TWO NUMBERS OF ANY LENGTH BY HOOKING TOGETHER ENOUGH 1-BIT ADDERS.



CODING & COMBINATION



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

1	0	1	1	1	0	0	1

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 197×10^5.

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

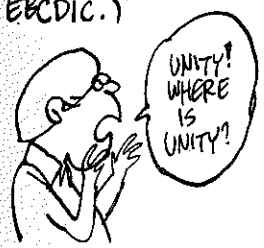
AND WHAT ABOUT *NON*-NUMERICAL INFORMATION — THE ALPHABET, PUNCTUATION MARKS, OTHER SYMBOLS, AND EVEN THE BLANK SPACE ??

SINCE THERE IS NO NATURAL WAY TO ENCODE THESE INTO 0's AND 1's, 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.)



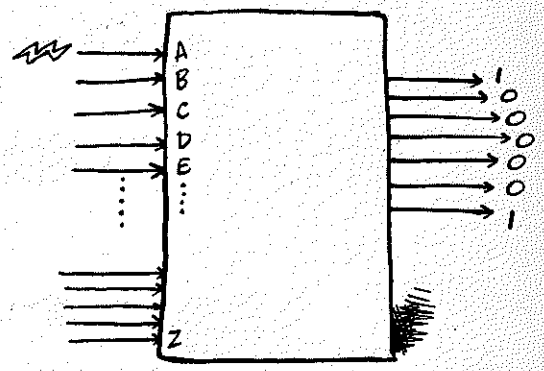
NEXT FOUR BITS \ FIRST THREE BITS	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	¦
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

☆ THUS, THE LETTER "T" IS ENCODED AS 101 0100... ETC!

☆ THE FIRST TWO COLUMNS CONTAIN SYMBOLS FOR SUCH THINGS AS "START OF HEADING" (SOH) AND OTHER TEXTUAL DIRECTIONS.

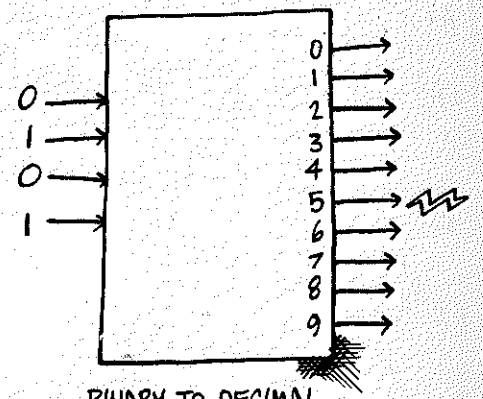
TO ENCODE AND DECODE DATA, COMPUTERS USE LOGIC DEVICES CALLED, NATURALLY ENOUGH, ***ENCODERS*** AND ***DECODERS***.

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.



ASCII ENCODER

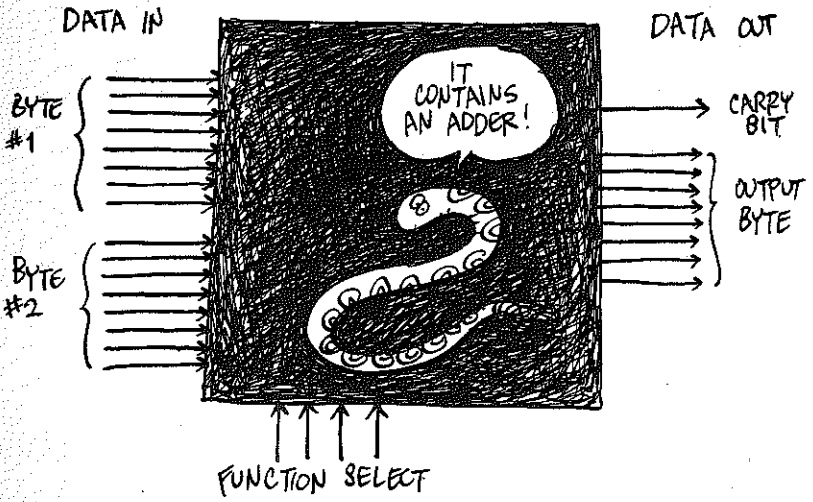
A **DECODER** WORKS THE OTHER WAY AROUND, TRANSLATING A PATTERN OF BITS INTO A SINGLE OUTPUT SIGNAL. ONE DECODER CONVERTS A BINARY NIBBLE INTO A DECIMAL DIGIT. ANOTHER TRANSFORMS A SPECIFIED LOCATION, OR ***ADDRESS***, IN MEMORY INTO A SIGNAL TO THAT MEMORY CELL. (SEE P. 155.)



BINARY TO DECIMAL DECODER

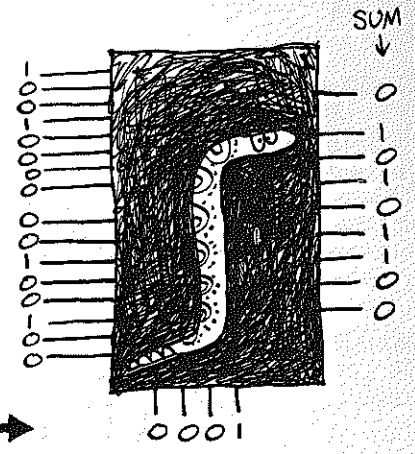
ONCE ALPHANUMERIC INFORMATION IS ENCODED IN BINARY STRINGS, IT IS READY TO BE PROCESSED BY THE COMPUTER'S MOST ELABORATE COMBINATION OF LOGIC GATES, THE

ARITHMETIC LOGIC UNIT (OR **ALU**, FOR SHORT).

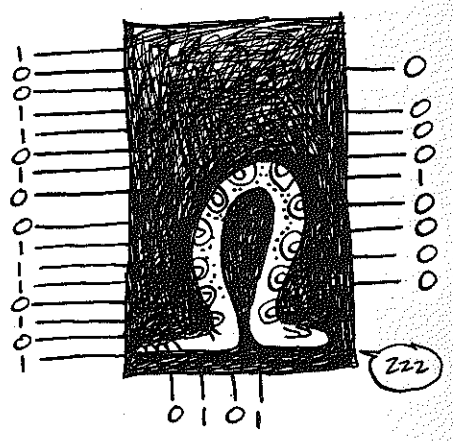


THIS IS THE MACHINE'S CENTRAL PROCESSOR, WHICH CAN ADD, SUBTRACT, MULTIPLY, COMPARE, SHIFT, AND PERFORM A WEALTH OF OTHER LOGICAL FUNCTIONS. THE DRAWING ABOVE REPRESENTS AN 8-BIT ALU, BUT THEY CAN RANGE FROM FOUR TO SIXTY BIT CAPABILITY, DEPENDING ON THE COMPUTER.

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 →

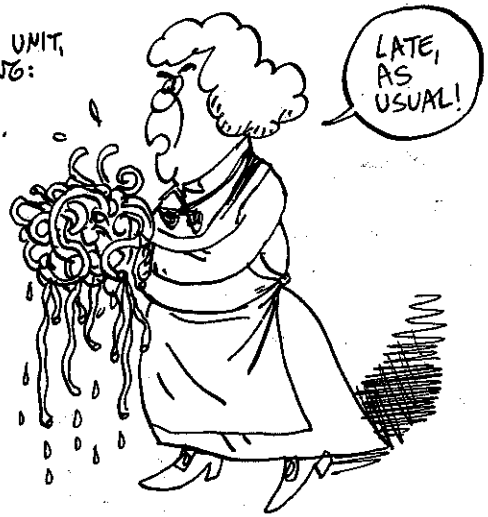


ANOTHER FUNCTION (0101, SAY) MIGHT ***COMPARE*** TWO BYTES, BIT BY BIT, AND OUTPUT A 1 WHEREVER THEY AGREE. (MEANWHILE, THE ADDER TAKES A NAP.)



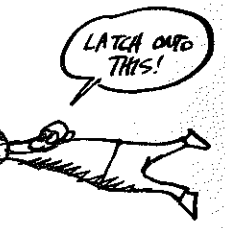
YOU CAN GET AN IDEA OF A FANCY ALU'S CAPABILITIES FROM THE LIST ON PAGE 182.

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 BABBAGE BE WITHOUT SOMEPLACE TO SET DOWN HER SPAGHETTI?

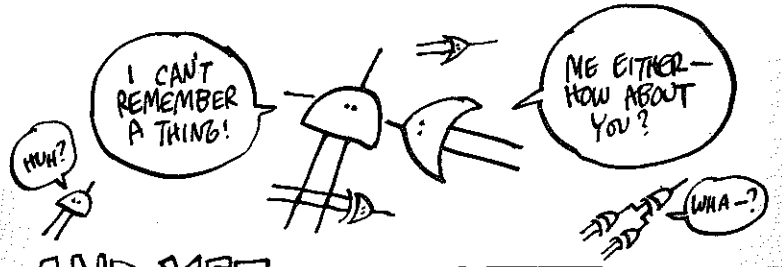


ALTHOUGH THE ALU CAN PERFORM MIRACLES OF INPUT/OUTPUT, IT CAN'T **REMEMBER** ANYTHING — AND THAT'S WHERE *FLIP-FLOPS* COME IN...

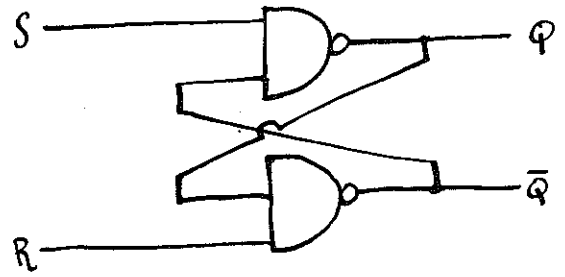
FLIP-FLOPS FLIP-FLOPS

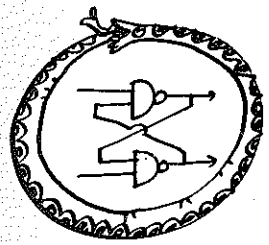


VERSATILE AS THEY MAY BE, THE LOGICAL COMBINATIONS WE'VE BEEN SKETCHING STILL HAVE NO MEMORY. THEIR OUTPUT CONTINUES ONLY AS LONG AS THE INPUT IS APPLIED.



AND YET — THERE IS A WAY TO HOOK THESE LOGICAL BUT SENILE GATES TOGETHER INTO A GADGET THAT ***HOLDS*** AN OUTPUT INDEFINITELY: THE ***FLIP-FLOP***. STARE AT THIS A MINUTE!!

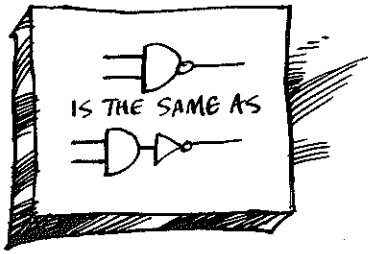




BESIDES THE STRANGE WAY A FLIP-FLOP EATS ITS OWN TAIL, PLEASE NOTE THE UNFAMILIAR GATE USED IN THE CONSTRUCTION. IT'S CALLED A

NAND GATE

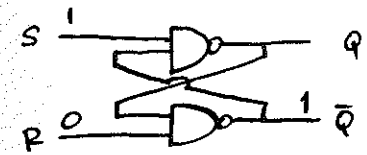
WHICH IS MERELY AN ABBREVIATION OF "NOT-AND."



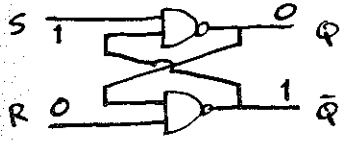
A	B	NAND
1	1	0
1	0	1
0	1	1
0	0	1

NOW FOR THE FLIP-FLOP IN ACTION:

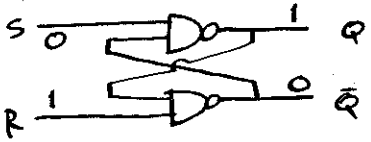
SUPPOSE THE INPUT IS S=1, R=0



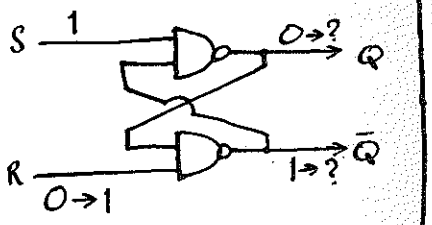
THEN $\bar{Q}$ MUST BE 1, BECAUSE *NAND* OUTPUTS 1 IF EITHER INPUT IS 0. COUPLING THIS BACK TO THE UPPER GATE GIVES Q=0:



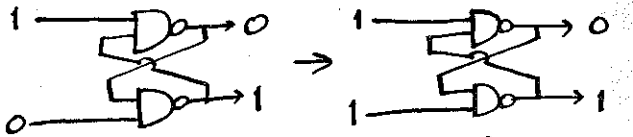
AND IF S=0, R=1 ? WELL, THAT'S JUST THE PREVIOUS DIAGRAM TURNED UPSIDE DOWN:



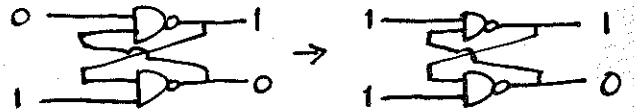
NOW WHAT HAPPENS WHEN THE INPUT **CHANGES?** SUPPOSING WE BEGIN WITH THE INPUT (S=1, R=0), WHAT DOES CHANGING IT TO (S=1, R=1) DO TO THE FLIP-FLOP'S OUTPUT?



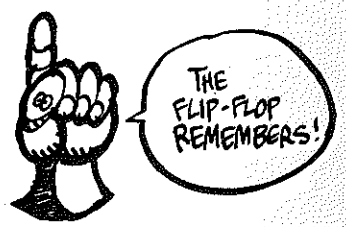
THE ANSWER IS: **NOTHING!** THE LOWER *NAND*-GATE'S INPUT BECOMES (0, 1), SO ITS OUTPUT $\bar{Q}$ IS STILL 1, SO Q REMAINS 0.



BUT PRECISELY THE SAME LINE OF REASONING SHOWS NO CHANGE IN OUTPUT WHEN INPUT CHANGES TO (S=1, R=1) FROM (S=0, R=1):

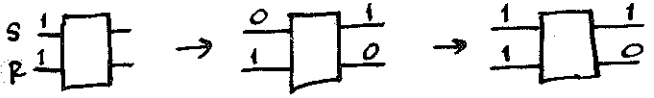


A LITTLE WEIRD, ISN'T IT? THE SAME INPUT (S=R=1) CAN PRODUCE TWO DIFFERENT OUTPUTS, DEPENDING ON THE PREVIOUS INPUT!

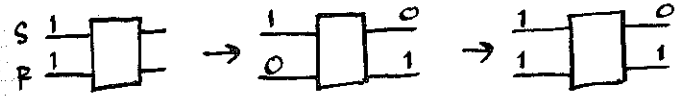


THE WAY A FLIP-FLOP IS USED IS THIS: IT BEGINS BY SITTING THERE WITH A CONSTANT INPUT OF (S=1, R=1) AND AN OUTPUT OF GOD-KNOWS-WHAT:

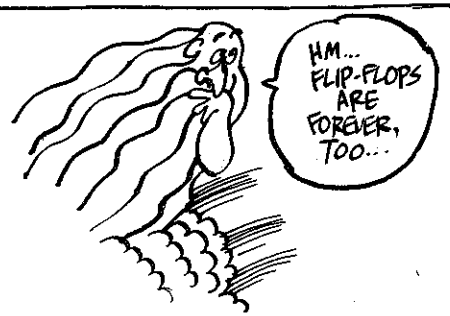
YOU **SET** THE FLIP-FLOP [I.E., MAKE Q=1] BY FLASHING A 0 MOMENTARILY DOWN THE S-WIRE, AND THEN RETURNING IT TO 1:



OR YOU CAN **RESET** IT [MAKE Q=0] BY FLASHING A 0 DOWN THE R-WIRE, THEN RETURNING IT TO 1:

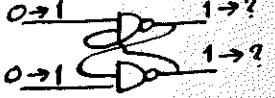


IN EITHER CASE, AS LONG AS (1,1) KEEPS COMING IN, THE FLIP-FLOP WILL MAINTAIN ITS OUTPUT UNTIL IT'S CHANGED WITH ANOTHER INCOMING 0.



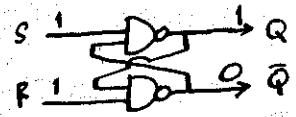
THE ONLY INPUT COMBINATION WE HAVEN'T CHECKED IS (R=S=0). IT'S EASY TO VERIFY THAT IT PRODUCES OUTPUT OF $Q=\bar{Q}=1$:

WHAT HAPPENS WHEN THE INPUT RETURNS TO (1,1)?

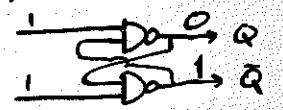


THE ANSWER IS NOT SO CLEAR: IT DEPENDS ON WHICH OUTPUT HAPPENS TO FLOP FIRST!! (ONE OF THEM MUST.)

IF $\bar{Q}$ IS FIRST TO CHANGE, WE GET:



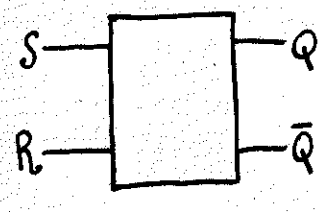
IF Q FLOPS FIRST, HOWEVER:



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 (S=0, R=0) IS

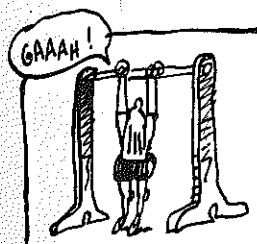
DISALLOWED.

WE CAN SUMMARIZE THE BASIC "R-S" FLIP-FLOP LIKE SO:



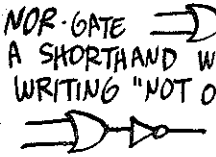
S	R	Q	$\bar{Q}$
1	1	NO CHANGE	
1	0	0	1
0	1	1	0
0	0	DISALLOWED!	

FLIP-FLOP INPUTS ARE ALWAYS ARRANGED TO MAKE CERTAIN THE DISALLOWED STATE CANNOT ARRIVE.



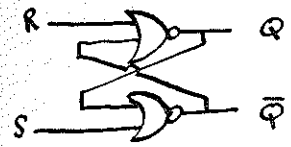
A LITTLE EXERCISE:

A NOR-GATE IS A SHORTHAND WAY OF WRITING "NOT OR": I.E.



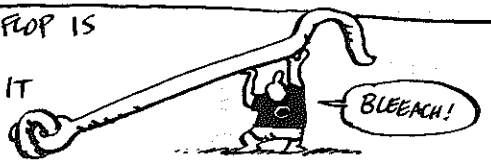
A	B	NOR
1	1	0
1	0	0
0	1	0
0	0	1

A BASIC R-S FLIP-FLOP MAY ALSO BE MADE OUT OF NOR-GATES:

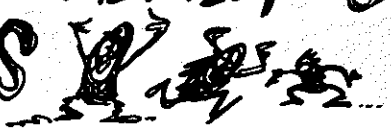


1. WHAT IS THE OUTPUT WHEN R=0, S=1 ? WHEN S=0, R=1 ?
2. WHAT HAPPENS WHEN EACH OF THESE INPUT CONDITIONS CHANGES TO R=0, S=0?
3. WHAT IS THE OUTPUT WHEN R=1, S=1 ? WHAT HAPPENS WHEN THIS CHANGES TO R=0, S=0 ?
4. WHAT INPUT COMBINATION MUST BE DISALLOWED?
5. IF R=0, S=0, HOW DO YOU SET THIS FLIP-FLOP (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.

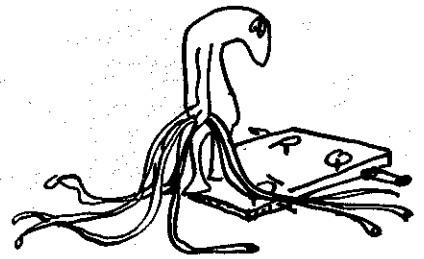


REGISTERS, COUNTERS, & GLITCHES



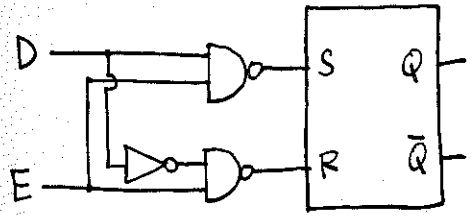
IF THE FLIP-FLOP IS A DEVICE FOR STORING ONE BIT, A **REGISTER** STORES SEVERAL BITS SIMULTANEOUSLY. IT'S LIKE A ROW OF BOXES, EACH HOLDING ONE BIT.

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

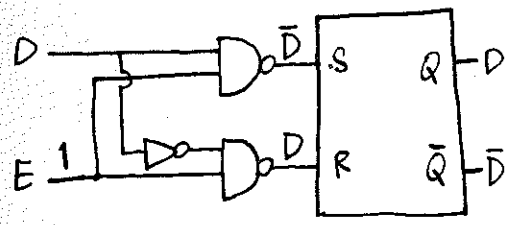


...SORT OF! **BUT** IF YOU TRY AND MAKE THIS WORK BY HOOKING UP SOME INPUTS TO R-S FLIP-FLOPS, YOU MAY FIND YOURSELF GROWING CONFUSED!

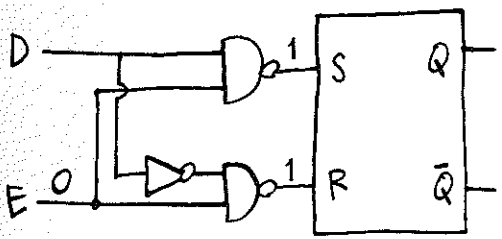
THE SOLUTION IS TO ADD A "GATING NETWORK" TO THE BASIC R-S FLIP-FLOP.



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.

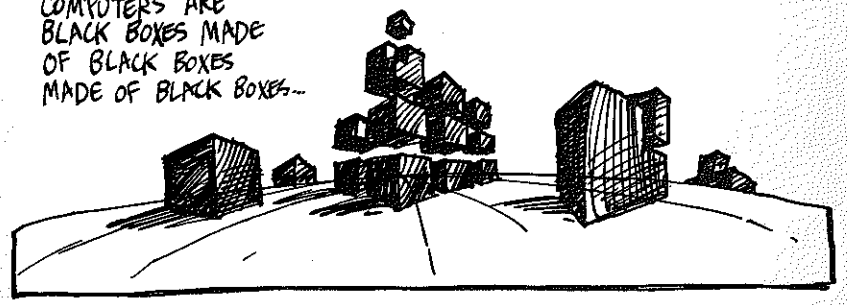


WHEN E=1, THEN R=D AND S=$\bar{D}$ (NOT-D). HENCE, THE VALUE OF D IS STORED AT Q. IN OTHER WORDS, E=1 **ENABLES** THE BIT D TO BE LOADED INTO THE FLIP-FLOP.

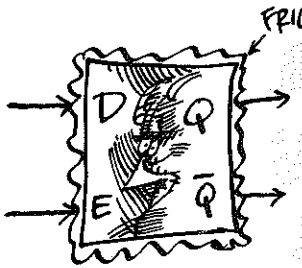


WHEN E=0, S AND R BOTH BECOME 1, AND THE FLIP-FLOP DOES NOT CHANGE. THAT IS, E=0 BLOCKS THE ARRIVAL OF MORE DATA.

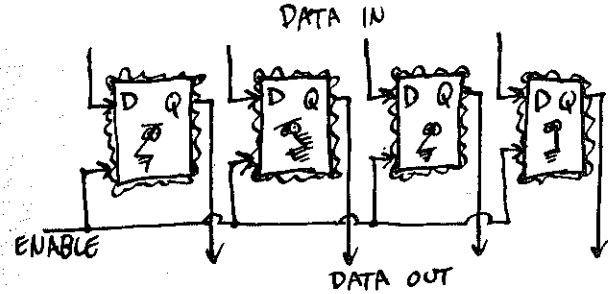
COMPUTERS ARE BLACK BOXES MADE OF BLACK BOXES MADE OF BLACK BOXES...



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

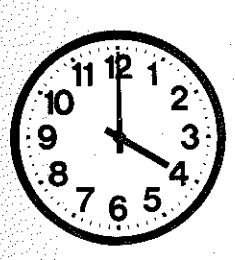


THEN HERE'S A **PARALLEL REGISTER**: NOT THE ONLY KIND OF REGISTER, BUT A GENUINE MEMBER OF THE BREED!



ON E=1, FOUR BITS ARE SIMULTANEOUSLY LOADED INTO THE LATCHES!

NOW WHAT CONTROLS THE "ENABLE" INPUT?



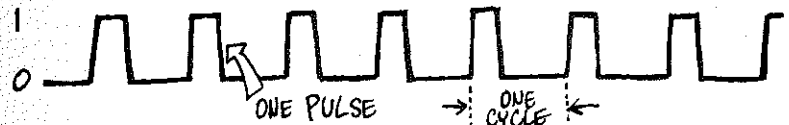
A BASIC FACT OF COMPUTER LIFE:

AS SOON AS YOU BEGIN STORING DATA, QUESTIONS OF *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,

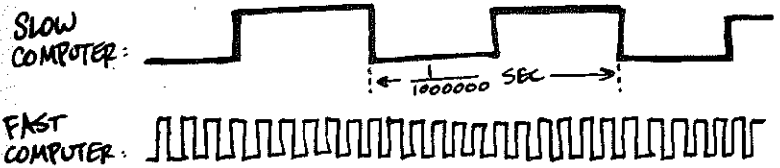
⇨ ALL COMPUTERS HAVE CLOCKS!

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

THE COMPUTER'S PULSE IS SQUARE AND COLD:

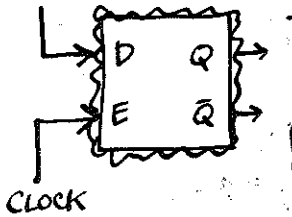


ONE **CLOCK PULSE** IS THE BURST OF CURRENT WHEN CLOCK OUTPUT = 1. ONE **CYCLE** IS THE INTERVAL FROM THE BEGINNING OF A PULSE TO THE BEGINNING OF THE NEXT. DEPENDING ON THE COMPUTER, THE CLOCK FREQUENCY MAY BE HUNDREDS OF THOUSANDS TO ***BILLIONS*** OF CYCLES PER SECOND!



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...

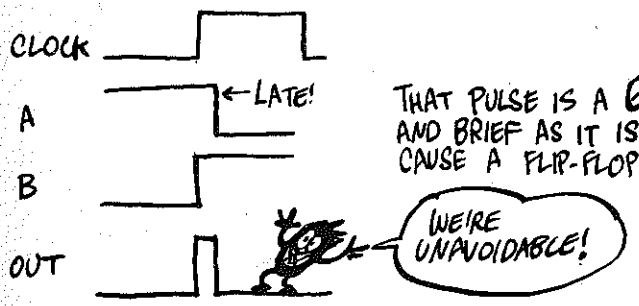
A TYPICAL EXAMPLE IS TO ATTACH THE CLOCK TO THE "ENABLE" INPUT OF A GATED LATCH, IN WHICH CASE THE LATCH BECOMES KNOWN AS A "D FLIP-FLOP."



THEN A NEW BIT OF DATA IS LOADED AT EVERY CLOCK PULSE!

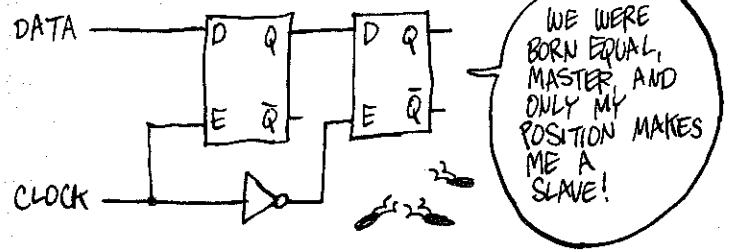
UNFORTUNATELY, THINGS ARE RARELY IDEAL! IT TAKES A NON-ZERO TIME FOR A SIGNAL TO PASS ALONG A WIRE, SO THINGS ARE NEVER PERFECTLY SYNCHRONIZED. FOR EXAMPLE, SUPPOSE AT AN *AND* GATE, ONE INPUT IS CHANGING FROM 1 TO 0, AND THE OTHER FROM 0 TO 1:

IF A CHANGES AFTER B, THE OUTPUT WILL HAVE AN UNWANTED PULSE:

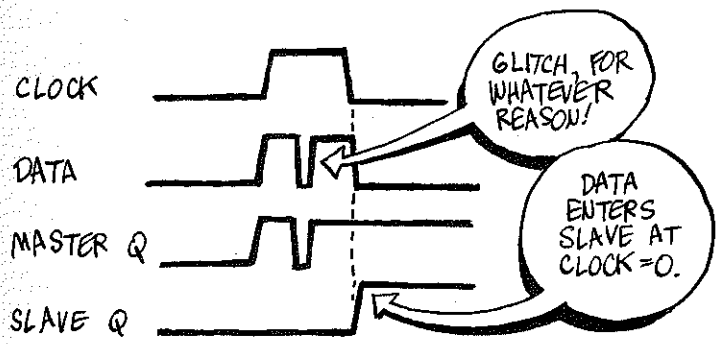


THAT PULSE IS A **GLITCH**, AND BRIEF AS IT IS, IT CAN CAUSE A FLIP-FLOP TO FLOP!

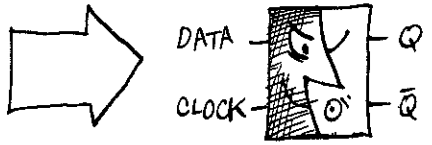
THE GLITCH IS DEFEATED BY THE **MASTER-SLAVE** FLIP-FLOP:



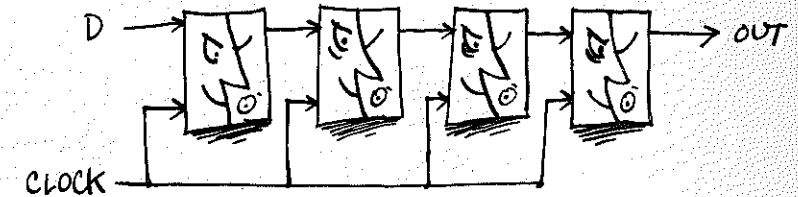
THE INVERTED CLOCK SIGNAL TO THE SLAVE FLIP-FLOP DELAYS THE DATA INPUT FROM ARRIVING AT THE SLAVE UNTIL THE ***END*** OF A CLOCK PULSE, AFTER ALL GLITCHES HAVE DIED OUT. FOR EXAMPLE, SUPPOSE WE WANT TO LOAD THE BIT **1** INTO THE FLIP-FLOP.



AS USUAL, WE DRAW THE WHOLE THING AS A SINGLE BOX!

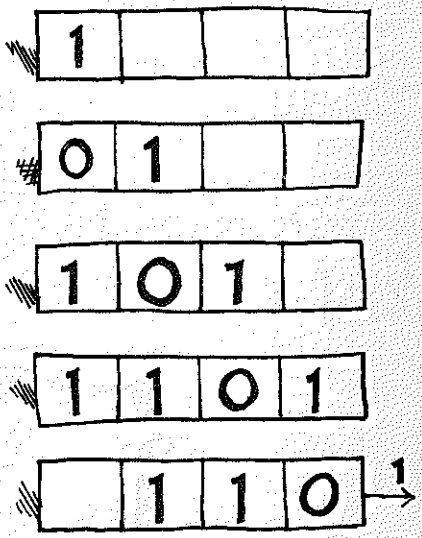


STRINGING A NUMBER OF MASTER-SLAVE FLIP-FLOPS TOGETHER MAKES A **SHIFT REGISTER:**



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:



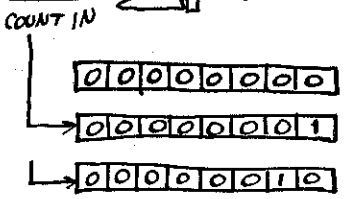
EACH CLOCK PULSE BRINGS A NEW BIT INTO THE REGISTER. (WHY DOESN'T THE BIT TRAVEL ALL THE WAY THROUGH ON ONE PULSE? BECAUSE OF THE MASTER-SLAVE FLIP-FLOPS!)

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.

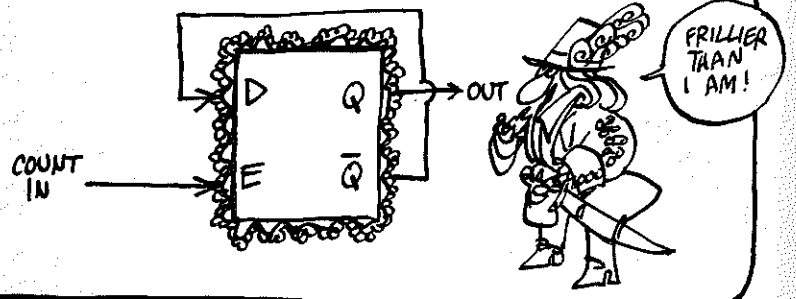
FINALLY, A SPECIAL KIND OF REGISTER: THE **COUNTER.**

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:

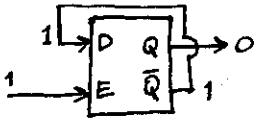


ETC!

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:

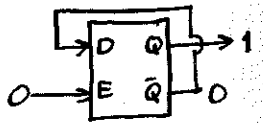


ON E=1, $\bar{Q}$ GOES IN AT D:



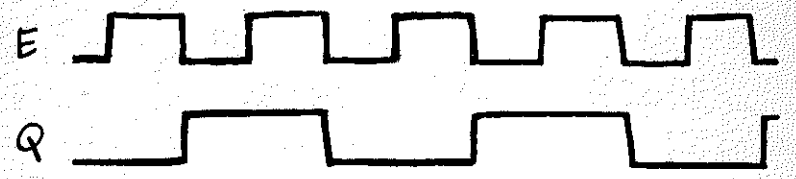
E: ⌐‾

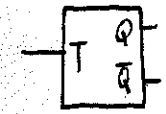
ON E=0, D PASSES TO Q, AND THE OUTPUTS REVERSE.



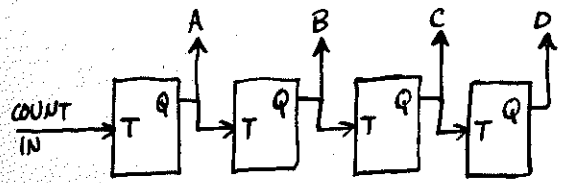
E: ‾⌐

THEN Q REVERSES ONCE FOR EACH CYCLE OF E—IN OTHER WORDS, Q **TOGGLES** JUST HALF AS OFTEN AS E:





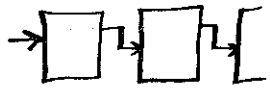
AS USUAL, WE ABBREVIATE THE WHOLE CIRCUIT BY THIS SIMPLER BOX. THE "T" IS FOR TOGGLE, TO INDICATE THAT THE FLIP-FLOP TOGGLES WHENEVER T=1. THEN HERE'S OUR COUNTER: EACH FLIP-FLOP TOGGLING AT HALF THE RATE OF THE ONE TO ITS LEFT:



COUNT IN	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
⋮			⋮	

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.

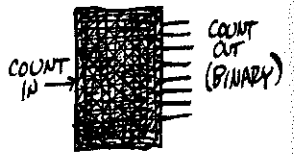


2 WHEN THE 16TH COUNT PULSE ARRIVES, THE COUNTER RETURNS TO 0. TO GO HIGHER THAN 15, MORE FLIP-FLOPS ARE NEEDED.

THIS 14-BIT COUNTER CAN GO FROM 0 TO $2^{14}-1 = 16,383$

3 THE N^{TH} 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 ONE CYCLE PER SECOND.

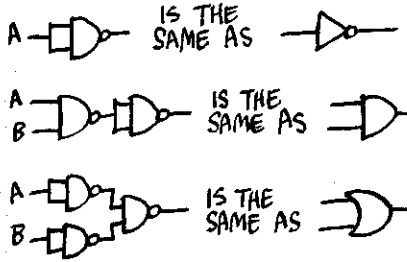
4 THERE ARE ALSO ***SYNCHRONOUS*** COUNTERS, WHICH REGISTER ALL BITS SIMULTANEOUSLY, AND COUNTERS WHICH RETURN TO 0 ON ANY PREASSIGNED NUMBER. IN ANY CASE, FROM NOW ON, A COUNTER IS JUST ANOTHER BLACK BOX !!



EXERCISES

THE AMAZING *NAND*:

1. SHOW THAT

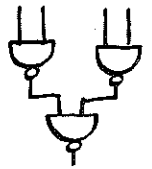


CONCLUDE THAT ⇒ ALL LOGIC CAN BE DERIVED FROM THE SINGLE RELATION *NAND* !!!

2. CAN THE SAME BE DONE WITH *NOR*?

3. SHOW THAT

IS THE SAME AS



REDRAW THE ADDER ON P. 126 USING ONLY *NAND* GATES.

4. GIVEN A 4-BIT SHIFT REGISTER,

SHOW ITS CONTENTS AFTER EACH OF FOUR CLOCK PULSES AS THE NIBBLE 0011 IS ENTERED.

5. HOW WOULD YOU ATTACH A BUZZER TO A COUNTER TO SOUND WHEN THE COUNT HITS NINE (=1001 IN BINARY)?
HINT: LOOK AT THE SEAT BELT BUZZER ON P. 109.

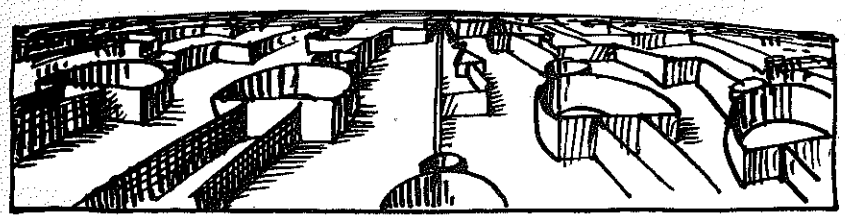
6. CONVINCE YOURSELF THAT ATTACHING INVERTERS TO THE OUTPUTS MAKES A COUNTER COUNT BACKWARDS.

NOW IN CASE YOU'RE FEELING STRANGLED BY SPAGHETTI—

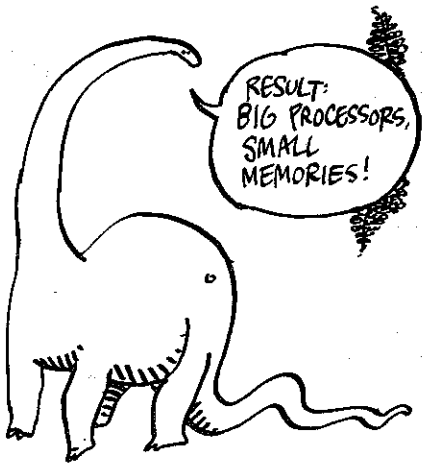
THE TANGLED DIAGRAMS ON THE PRECEDING PAGES WERE NEVER INTENDED TO TRACE THE COMPLETE WIRING DIAGRAM OF ANY COMPUTER. RATHER, THEY ARE MEANT TO DEMONSTRATE HOW THE COMPUTER'S ESSENTIAL FUNCTIONS— MATH, COMPARISON, DECODING, DATA SELECTION AND STORAGE— ALL DEPEND ON SIMPLE LOGIC.

NOW THAT YOU PRESUMABLY BELIEVE IN THE POWER OF LOGIC, NO MORE WIRING DIAGRAMS ARE NEEDED!

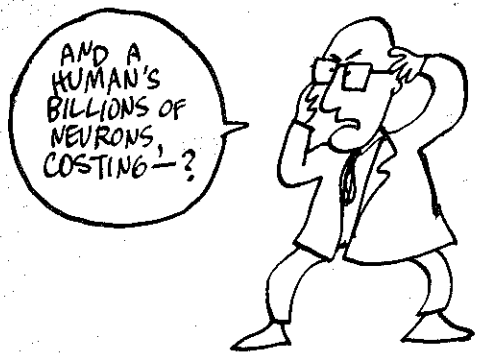
ONWARD, TO HIGHER LEVELS!



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!!



*ENIAC DID NOT COMPUTE IN BINARY.

THE SAME RESEARCH EFFORT, HOWEVER, HAS PRODUCED A BEWILDERING ARRAY OF MEMORY TYPES AND TECHNOLOGIES!!

WELL, ONE HAS TO BEGIN SOMEWHERE!!

AN IMPORTANT DISTINCTION EXISTS BETWEEN

ELECTRONIC

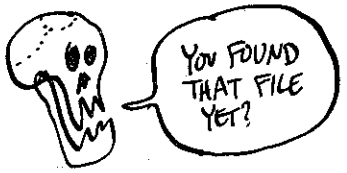
AND

ELECTRO-MECHANICAL

MEMORY DEVICES.

ELECTRONIC MEMORIES, WITH NO MOVING PARTS, ARE AS FAST AS THE REST OF THE COMPUTER.

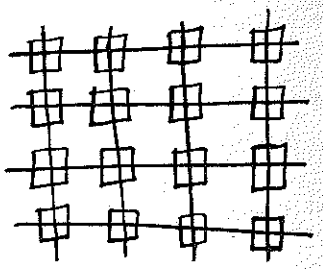
ELECTROMECHANICAL MEMORIES HAVE MOVING PARTS, LIKE DISKS OR REELS OF TAPE. THIS MAKES THEM SLOW— HOW SLOW DEPENDING ON THE TYPE OF MEMORY.



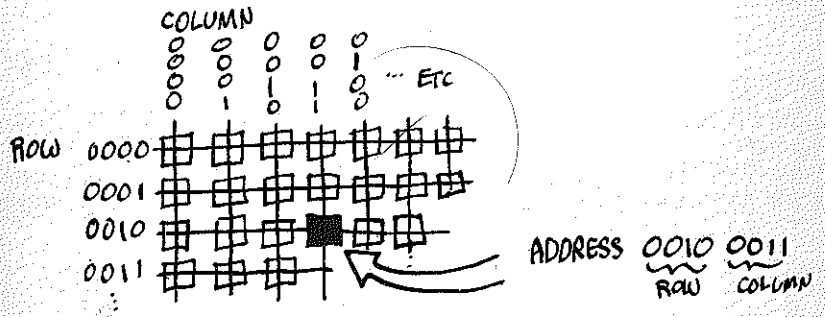
ELECTRONIC MEMORIES' SPEED MAKES THEM IDEAL FOR THE COMPUTER'S MAIN, OR INTERNAL MEMORY, WHILE ELECTRO-MECHANICAL MEMORIES ARE USED FOR AUXILIARY STORAGE OUTSIDE THE MACHINE.

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.

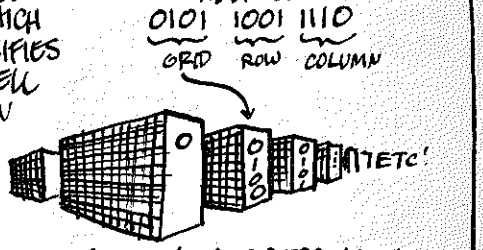
INTERNAL MEMORY CAN BE THOUGHT OF AS A SIMPLE GRID, WITH A CELL AT EACH INTERSECTION. DEPENDING ON THE COMPUTER, EACH CELL CAN HOLD ONE BYTE, TWO BYTES, OR MORE.



EVERY CELL HAS A UNIQUE **ADDRESS**, SPECIFYING WHERE IT SITS IN THE GRID.

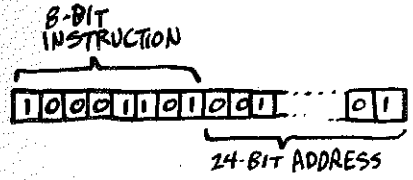


IN PRACTICE, THERE MAY BE MANY SUCH GRIDS, IN WHICH CASE THE ADDRESS SPECIFIES THE GRID NUMBER, AS WELL AS THE ROW AND COLUMN WITHIN IT.



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

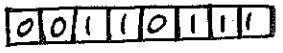
$$00000\cdots0$$

AND

$$1111\cdots1 = 2^{24}-1$$

GIVING 2^{24} POSSIBLE MEMORY CELLS.

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

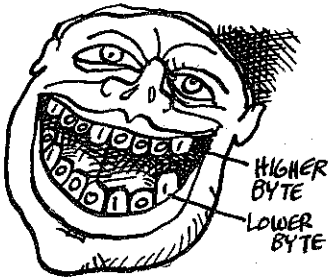


AN INSTRUCTION,

THE FIRST HALF OF AN ADDRESS,

AND THE SECOND HALF OF AN ADDRESS.

HERE THE ADDRESS IS 16 BITS LONG, GIVING $2^{16}=65,536$ POSSIBLE ADDRESSES.



16-BIT WORDS ARE OFTEN SPLIT LIKE THIS INTO HIGHER-LEVEL AND LOWER-LEVEL BYTES.

10001101 00010010
HIGHER LOWER

TO MAKE ADDRESSES SHORTER AND MORE READABLE, THEY'RE OFTEN EXPRESSED IN

HEXADECIMAL,

OR BASE-16, NUMERALS.

$10_{HEX} = 16_{DECIMAL}$

$100_{HEX} = 16^2 = 256$

$1000_{HEX} = 16^3 = 4096$

⋮

ETC!

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

DECIMAL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

FOR EXAMPLE:

$$4A0D_{HEX} =$$

$$4\times16^3$$
$$+\ 10\times16^2$$
$$+\ 0\times16$$
$$+\ 13\times1$$
$$\overline{\qquad\qquad}$$
$$18,957_{DECIMAL}$$

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

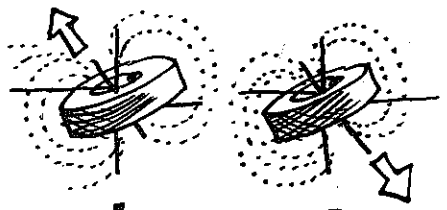
1011 1110 0101 1011
5 C 5 B

TO CONVERT HEX TO BINARY, JUST REVERSE THE PROCESS.

FROM THE HARDWARE POINT OF VIEW, THERE ARE THREE MAIN TYPES OF INTERNAL MEMORY.

CORE

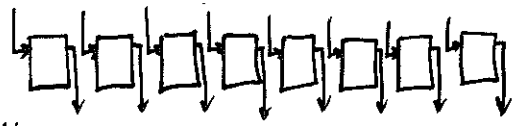
MEMORIES USE LITTLE MAGNETIC DOUGHNUTS – "CORES." EACH CORE CAN BE ELECTRICALLY MAGNETIZED IN ONE OF TWO DIRECTIONS, REPRESENTING 0 AND 1.



AND TWO SEMICONDUCTOR MEMORIES:

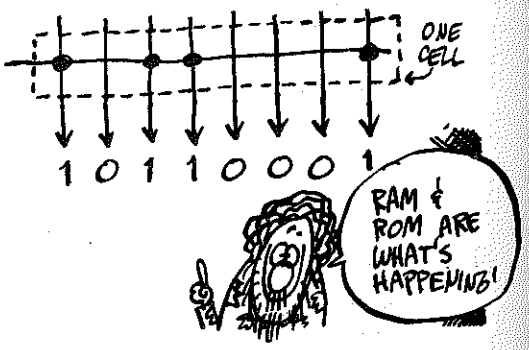
"RAM"

USES FLIP-FLOPS TO STORE BITS — SO EACH MEMORY CELL IS ESSENTIALLY A (PARALLEL) REGISTER!



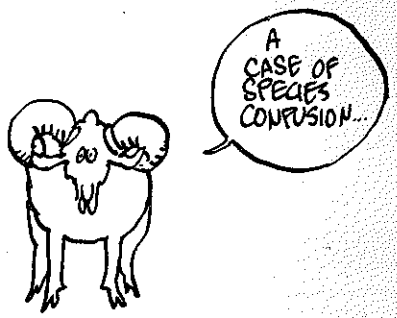
"ROM"

INDICATES A 1 OR 0 AT EACH GRID POINT BY THE PRESENCE OR ABSENCE OF AN ELECTRIC CONNECTION THERE.



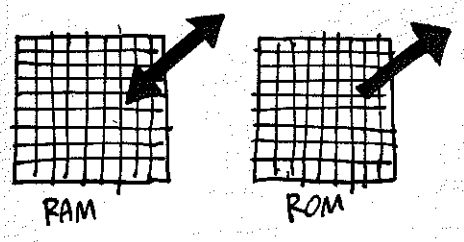
RAM STANDS FOR

"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!



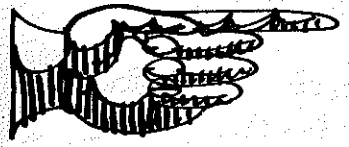
ROM STANDS FOR "READ-ONLY MEMORY."

ROM-AN STYLE LETTERING



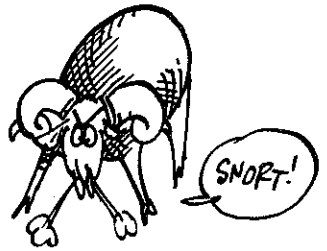
THE PRACTICAL DIFFERENCE BETWEEN THEM IS THAT YOU CAN ONLY READ WHAT'S IN ROM, WHILE WITH RAM YOU CAN READ THINGS OUT OR WRITE THEM IN WITH EQUAL EASE.

IN GENERAL!



WHEN YOU LOAD A PROGRAM INTO THE COMPUTER, IT IS STORED IN **RAM.**

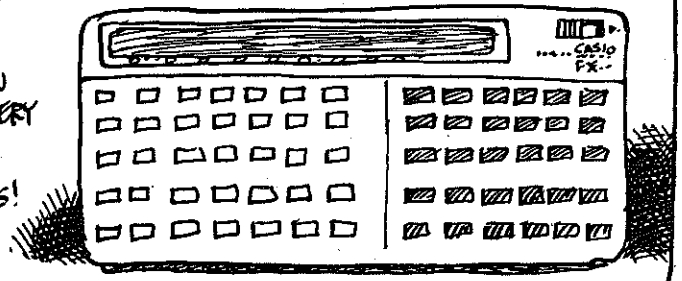
UNFORTUNATELY, RAM IS VOLATILE,



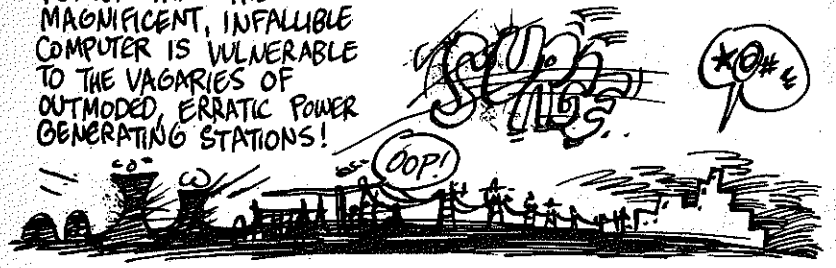
MEANING IT FORGETS EVERYTHING WHEN THE POWER IS TURNED OFF.

FOR EXAMPLE, I OWN A BATTERY-POWERED POCKET COMPUTER WITH 1680 BYTES OF RAM. IT CAN STORE UP TO TEN PROGRAMS ***EVEN WHEN I TURN IT OFF***, BECAUSE IT KEEPS SOME ELECTRICITY RUNNING THROUGH MEMORY.

BUT WHEN THE BATTERY DIES... BYE-BYE, PROGRAMS!

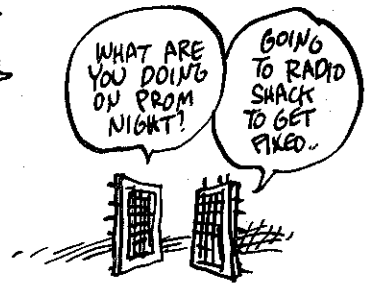


RAM VOLATILITY IS ONE REASON THAT THE MAGNIFICENT, INFALLIBLE COMPUTER IS VULNERABLE TO THE VAGARIES OF OUTMODED, ERRATIC POWER GENERATING STATIONS!

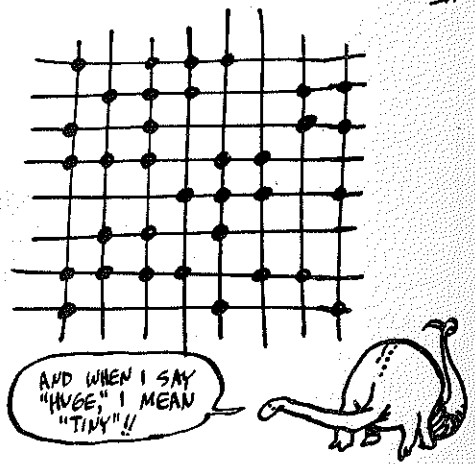


ROM — "READ-ONLY MEMORY" — ONCE ITS CONTENTS ARE ENTERED, CAN NEVER BE REWRITTEN.* ORDINARILY, ROM IS PROGRAMMED AT THE FACTORY, BUT THERE ARE NOW ALSO PROMS — PROGRAMMABLE ROMS — WHICH CAN BE CUSTOM-PROGRAMMED TO THE USER'S SPECIFICATIONS.

*EXCEPT FOR EPROM — ERASABLE PROGRAMMABLE ROM — BUT WE WON'T GET INTO THAT!



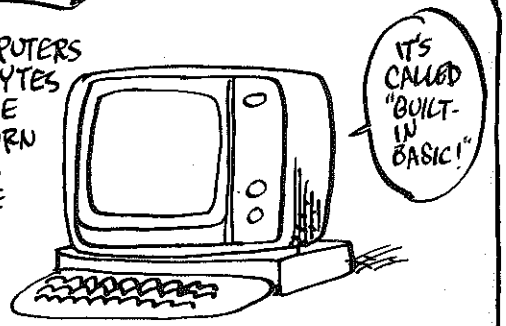
UNLIKE RAM, ROM IS NON-VOLATILE: IT KEEPS ITS CONTENTS EVEN WITHOUT POWER. AFTER ALL, IT'S NOTHING BUT A HUGE GRID OF WIRES WITH PHYSICAL CONNECTIONS AT SOME INTERSECTIONS. THE CONNECTIONS REMAIN, REGARDLESS OF ELECTRIC CURRENT.



SOME TYPICAL USES OF ROM:

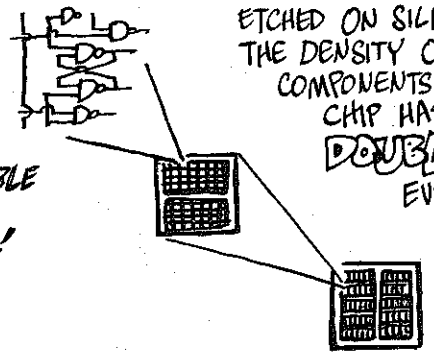
MOST VIDEO GAME CARTRIDGES ARE PROGRAMMED IN ROM. JUST PLUG IT IN AND IT'S READY TO GO! BUT OF COURSE, IT CAN'T BE REPROGRAMMED EITHER...

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!*

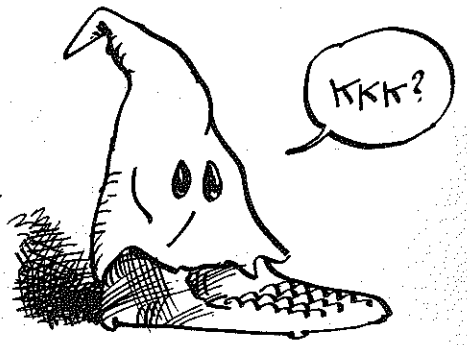


ETCHED ON SILICON CHIPS, THE DENSITY OF COMPONENTS PER CHIP HAS BEEN **DOUBLING** EVERY YEAR!

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

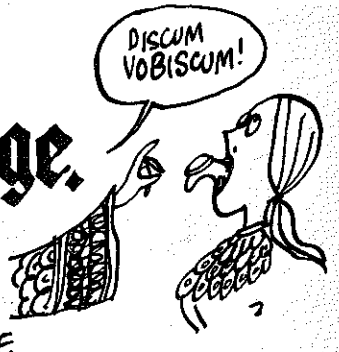
THE FIRST RAM CHIP WITH 1K BITS OF STORAGE WAS A SENSATION — BUT NOW 64K IS COMMON, AND THE 256K CHIP HAS ARRIVED! WHAT'S NEXT?



DESPITE THE GROWTH OF RAM CAPACITY, SOMETIMES IT IS NOT THE ANSWER TO EVERY PRAYER !!

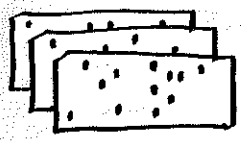
THE ANSWER?

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

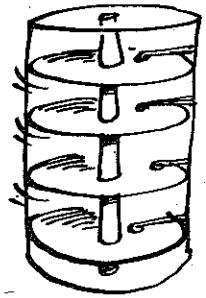
SAME IDEA AS PUNCH CARDS: A HOLE REPRESENTS 1, A NON-HOLE 0.

MAGNETIC TAPE

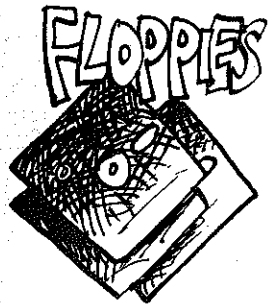
STORES BITS AS SMALL MAGNETIC REGIONS, WHICH MAY BE MAGNETIZED IN ONE OF TWO DIRECTIONS, REPRESENTING 1 OR 0.

FASTER, LESS BULKY, AND THE CURRENT STORAGE OF CHOICE IS THE

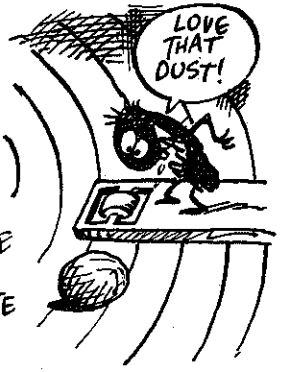
DISKS ALSO STORE BITS AS TINY MAGNETIZED REGIONS — UP TO 10 MILLION BYTES PER DISK!



A BIG COMPUTER SYSTEM USUALLY HAS MULTIPLE DISK DRIVES, WITH PHONOGRAPH-ARMLIKE READ/WRITE HEADS DARTING BACK AND FORTH ACROSS THE WHIRLING PLATTERS.

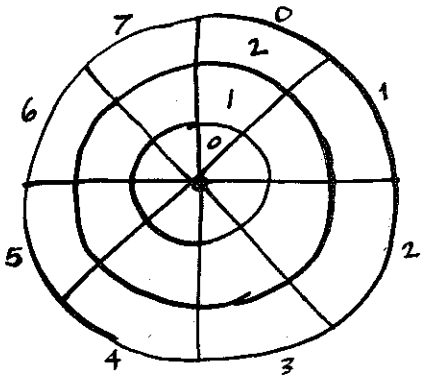


ARE SMALL, LOW-COST MAGNETIC DISKS MADE OF PLASTIC. THEY ALWAYS STAY IN THEIR JACKETS, BECAUSE A SPECK OF DUST CAN CREATE A MONSTER *GLITCH!*



OTHER, MORE EXOTIC MASS STORAGE TECHNOLOGIES INCLUDE **BUBBLE** MEMORIES, **CHARGE-COUPLED DEVICES**, AND **OPTICAL DISKS** READ BY LASERS.

LIKE INTERNAL MEMORY, MASS STORAGE MUST BE ORGANIZED, OR "FORMATTED." TAKE THE FLOPPY DISK FOR EXAMPLE:

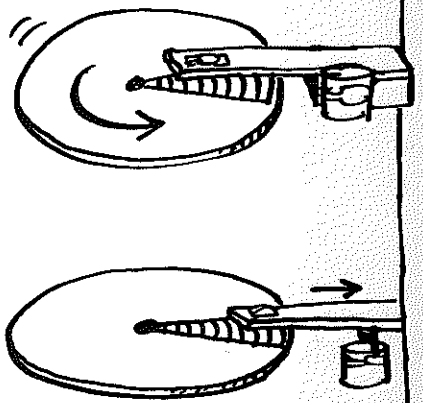


FLOPPIES ARE FORMATTED INTO RINGS AND SECTORS — THREE RINGS AND EIGHT SECTORS, IN THIS VERY OVER-SIMPLIFIED DISK. (IT'S MORE LIKE 26 SECTORS AND 77 RINGS IN A GENUINE DISK.)

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

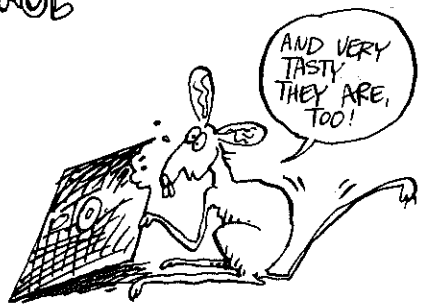
2) MOVES THE HEAD IN OR OUT TO THE PROPER RING.



THIS PROCESS TAKES MILLISECONDS — AN ETERNITY TO A COMPUTER!

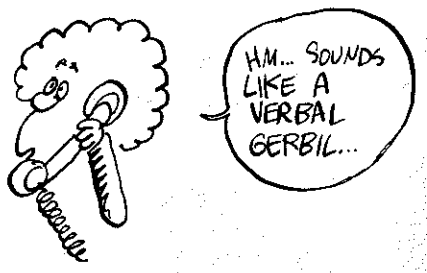
SOME TYPICAL USES OF MASS STORAGE:

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



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

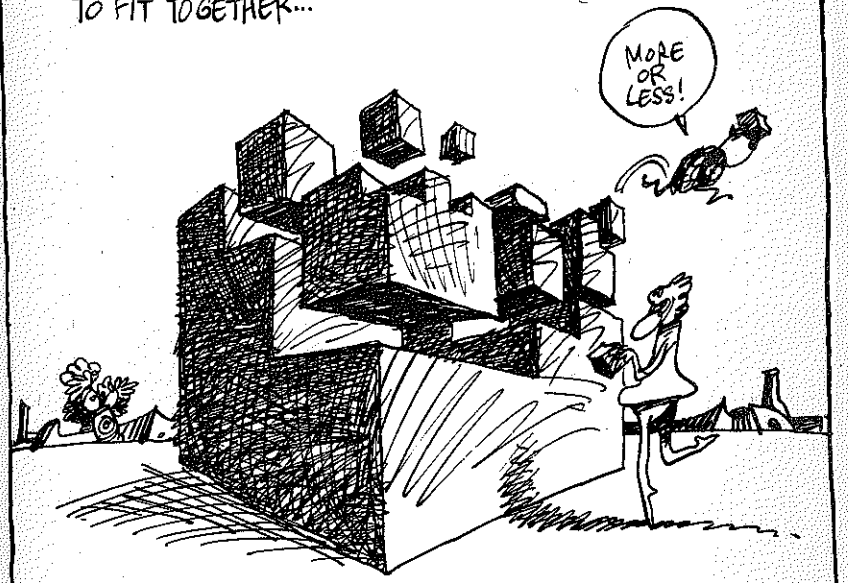
THE PHONE COMPANY STORES IN BUBBLE MEMORY THE MESSAGE: "THE NUMBER YOU HAVE REACHED IS NOT IN SERVICE..."



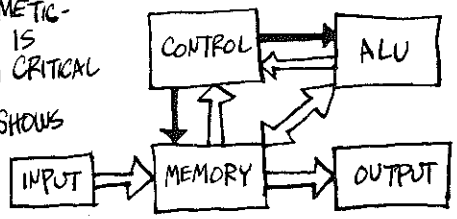
WELL, YOU GET THE PICTURE... NOW IT'S TIME TO MOVE ON...

GETTING EVERYTHING UNDER CONTROL

IN WHICH ALL THE BLACK BOXES ARE FINALLY SEEN TO FIT TOGETHER...



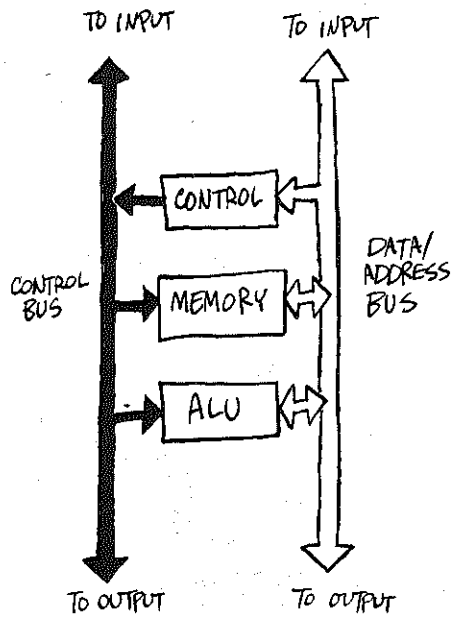
ALONG WITH INPUT/OUTPUT, MEMORY, AND THE ARITHMETIC-LOGIC UNIT, CONTROL IS THE COMPUTER'S FINAL, CRITICAL INGREDIENT. OUR OLD SCHEMATIC DIAGRAM SHOWS THE FLOW OF CONTROL (➡) AND INFORMATION (⇨).



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**.

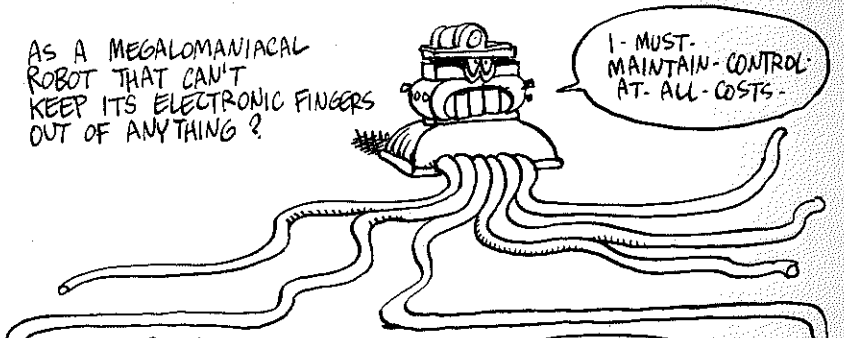
ACCORDING TO SIGNALS PASSED ALONG THE CONTROL BUS, ADDRESSES AND DATA GET ON AND OFF THE DATA/ADDRESS BUS, WITH THE PROVISO THAT ONLY ONE "PASSENGER" CAN RIDE THE BUS AT A TIME.



NOTE THAT ALL THE ARROWS ON THE CONTROL BUS POINT ***AWAY*** FROM THE CONTROL SECTION.

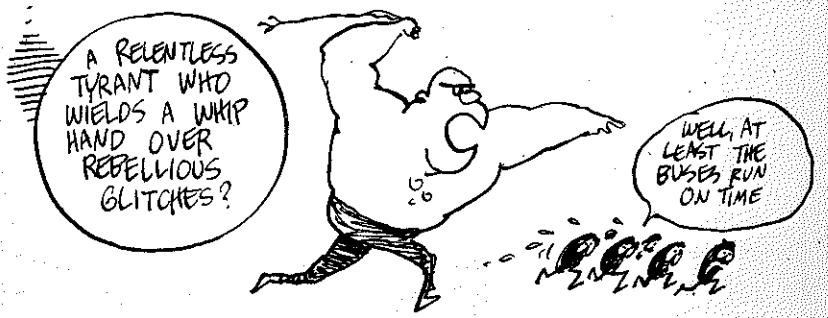
HOW ARE WE TO IMAGINE THIS CONTROL, FROM WHICH ALL DARK ARROWS POINT AWAY ??

AS A MEGALOMANIACAL ROBOT THAT CAN'T KEEP ITS ELECTRONIC FINGERS OUT OF ANYTHING ?

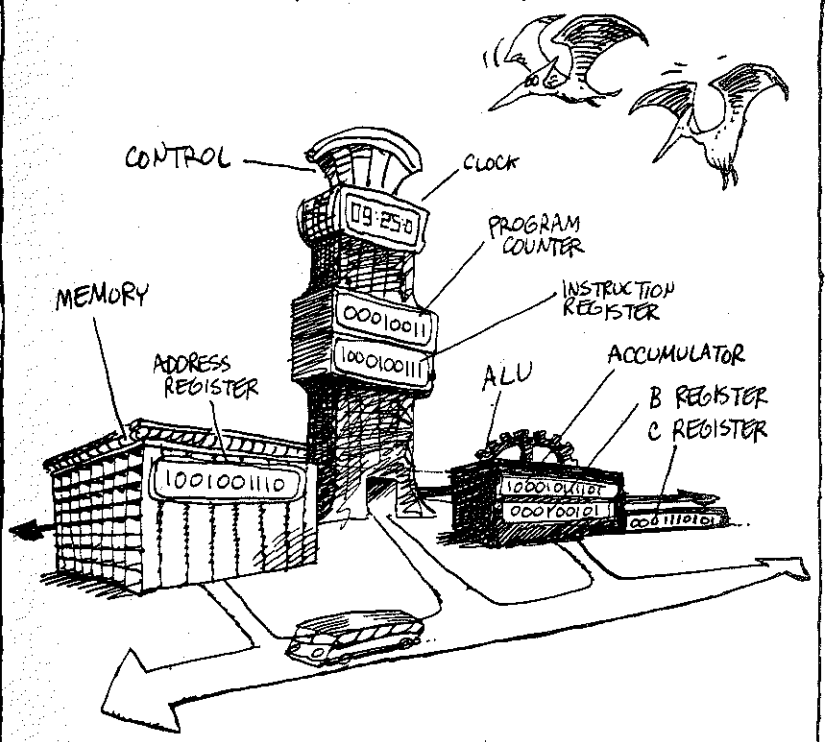


A WISE RULER WHO JUDICIOUSLY CHOOSES THE TIME FOR EVERY ACT?

A RELENTLESS TYRANT WHO WIELDS A WHIP HAND OVER REBELLIOUS GLITCHES?



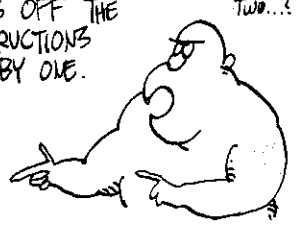
LIKE ANYONE ELSE, CONTROL REVEALS ITS CHARACTER BY ITS BEHAVIOR... SO LET'S FOLLOW WHAT HAPPENS IN THIS OVERSIMPLIFIED COMPUTER, WHICH FLESHES OUT THE DIAGRAM OF TWO PAGES BACK WITH SOME ESSENTIAL COUNTERS AND REGISTERS.



THIS IS A MINIMAL COLLECTION OF EQUIPMENT. A TYPICAL COMPUTER HAS MORE REGISTERS AND COUNTERS, BUT ALL COMPUTERS HAVE THE ONES SHOWN HERE.

HERE'S WHAT THEY'RE FOR:

PROGRAM COUNTER: TICKS OFF THE INSTRUCTIONS ONE BY ONE.



INSTRUCTION REGISTER: HOLDS AN ENCODED VERSION OF THE INSTRUCTION BEING PERFORMED.

ADDRESS REGISTER: HOLDS THE ADDRESS OF WHATEVER IS TO ENTER OR LEAVE MEMORY.

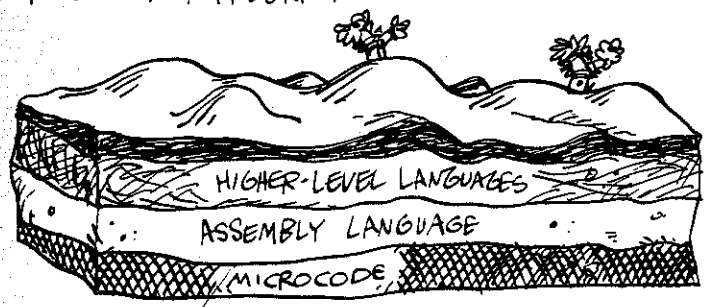
ACCUMULATOR: THE ALU'S MAIN REGISTER, KEEPING A RUNNING TOTAL OF ALU OPERATIONS.

B REGISTER: AN AUXILIARY REGISTER TO HOLD NUMBERS ON THEIR WAY TO ALU.

C REGISTER: HOLDS DATA ON THE WAY TO OUTPUT.

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!

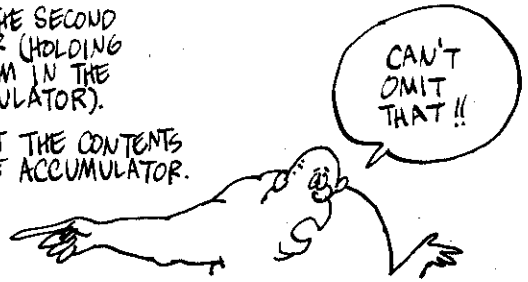


LIKE EVERYTHING ABOUT COMPUTERS, PROGRAMS CAN BE DESCRIBED AT VARIOUS LEVELS. WE BEGIN WITH

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
```

*MNEMONIC = MEMORY-AIDING

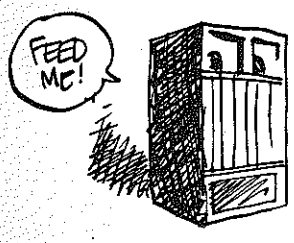
IN GENERAL, ASSEMBLY-LANGUAGE STATEMENTS HAVE TWO PARTS:

THE **OPERATOR**, WHICH DESCRIBES THE STEP TO BE PERFORMED

THE **OPERAND**, WHICH GIVES THE ADDRESS ON WHICH THE OPERATOR ACTS

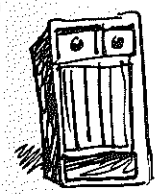
LDA 1E

NOTE HOWEVER! SOME OPERATORS DON'T NEED AN EXPLICIT OPERAND. "OUT", FOR INSTANCE, IS UNDERSTOOD TO APPLY TO THE ACCUMULATOR.



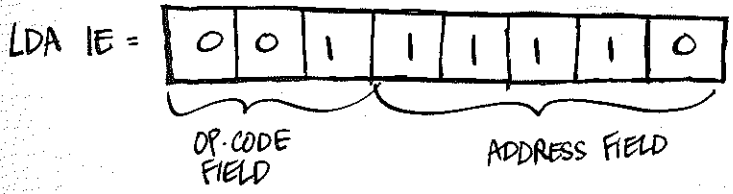
NOW THAT WE HAVE AN ASSEMBLY-LANGUAGE PROGRAM, HOW DO WE FEED IT TO THE MACHINE — WHICH ONLY UNDERSTANDS 0's AND 1's ?

THE ANSWER IS CLEAR: WITHIN THE MACHINE, EACH ***OPERATOR*** IS ***ENCODED*** AS A STRING OF BITS CALLED ITS "OP-CODE." SOME SIMPLE SAMPLES:



OPERATOR	OP-CODE
LDA	001
ADD	010
OUT	110
HALT	111

THEN A MACHINE INSTRUCTION CONSISTS OF AN OP-CODE SEGMENT, OR "FIELD," FOLLOWED BY AN ADDRESS FIELD GIVING THE OPERAND IN BINARY:



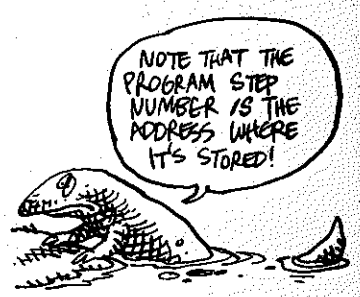
SO HERE'S OUR PROGRAM TRANSLATED INTO MACHINE LANGUAGE:

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

ANY 5 BITS ARE O.K. 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

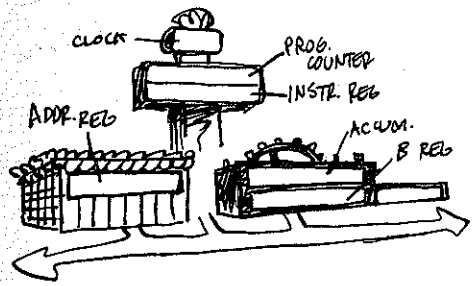
ADDRESS	CONTENTS
0	001 11110
1	010 11111
2	110 00000
3	111 00000



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!

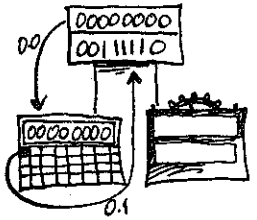


ONCE THE PROGRAM IS STORED, CONTROL CAN BEGIN EXECUTION, IN A SERIES OF EVEN MORE PRIMITIVE STEPS CALLED ***MICROINSTRUCTIONS***, ONE MICROINSTRUCTION OCCURRING WITH EACH CLOCK PULSE. ARE YOU READY FOR THE GORY DETAILS?

CONTROL BEGINS BY ***FETCHING*** THE FIRST INSTRUCTION. IT—

0.0. MOVES CONTENTS OF PROGRAM COUNTER (00000000 TO BEGIN WITH) TO ADDRESS REGISTER

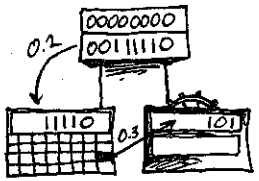
0.1 MOVES CONTENTS OF THAT MEMORY ADDRESS TO INSTRUCTION REGISTER



THE INSTRUCTION REGISTER NOW HOLDS THE FIRST INSTRUCTION. CONTROL "READS" IT AND—

0.2. MOVES THE INSTRUCTION REGISTER'S ADDRESS FIELD TO ADDRESS 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:

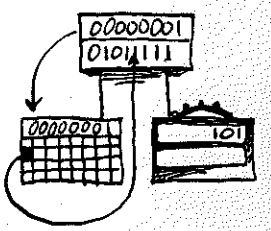
0.4 INCREMENT PROGRAM COUNTER

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 ADDRESS REGISTER

1.1 MOVE CONTENTS OF THAT ADDRESS TO INSTRUCTION REGISTER

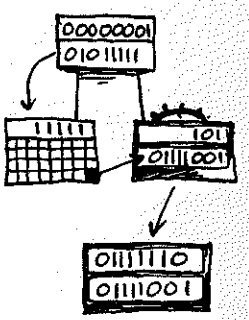


THE INSTRUCTION IN THE INSTRUCTION REGISTER, 01011111, CAUSES CONTROL TO:

1.2 MOVE ADDRESS FIELD FROM INSTRUCTION REGISTER TO ADDRESS REGISTER

1.3 MOVE CONTENTS OF THAT MEMORY ADDRESS TO B REGISTER

1.4 SIGNAL THE ALU TO ***ADD*** AND PUT THE SUM IN ACCUMULATOR



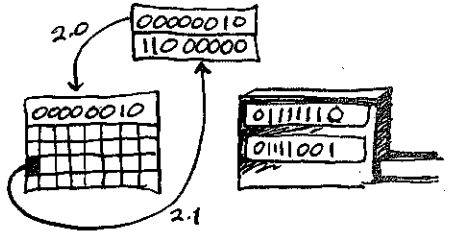
AGAIN, THERE'S ONE MORE STEP:

1.5 INCREMENT PROGRAM COUNTER

AND FINALLY?

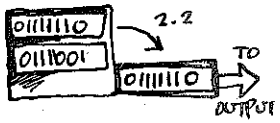
WELL, LUCKILY THE LAST TWO INSTRUCTIONS ARE EASIER:

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—

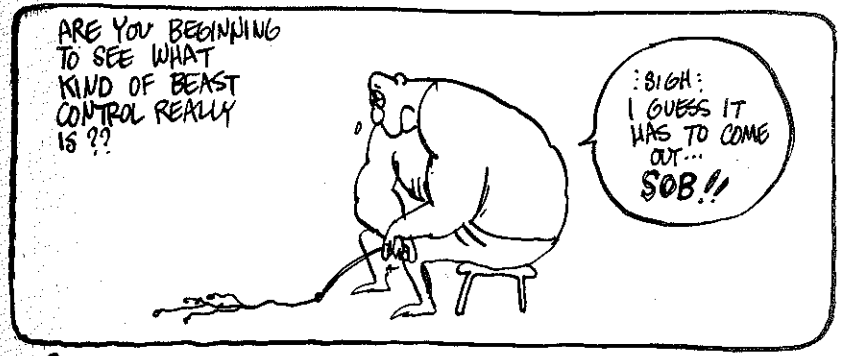
2.2. MOVE CONTENTS OF ACCUMULATOR TO C REGISTER



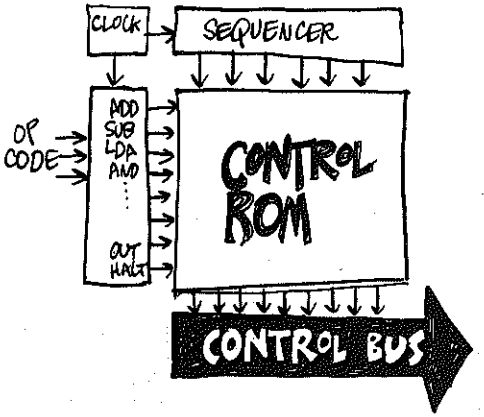
2.3. INCREMENT PROGRAM COUNTER

FINALLY, CONTROL FETCHES THE INSTRUCTION 111 ("HALT"), WHICH CAUSES CONTROL TO—

3.2 DO NOTHING



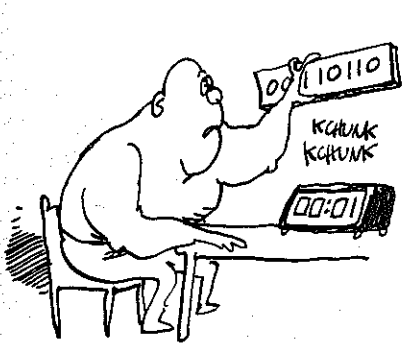
WITHOUT TOO MANY DETAILS, YOU CAN THINK OF CONTROL ROUGHLY LIKE THIS:



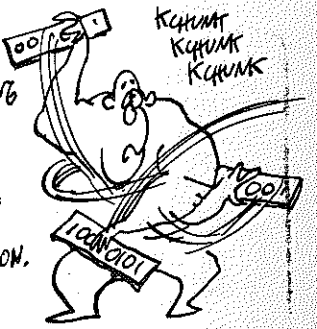
ITS INPUT CONSISTS OF CLOCK PULSES AND OP-CODES. ITS OUTPUT CONSISTS OF A *SEQUENCE* OF SIGNALS TO THE REGISTERS, COUNTERS, ALU, AND MEMORY.

THE "MICROPROGRAM," WHICH CONNECTS THE INPUTS TO THE PROPER OUTPUT COMBINATIONS, IS STORED IN A READ-ONLY MEMORY DEDICATED STRICTLY TO THIS PURPOSE.

THE FIRST COUPLE OF CLOCK PULSES CAUSE CONTROL TO FETCH AN INSTRUCTION...



...AND THE REMAINING PULSES CAUSE IT TO EXECUTE THE INSTRUCTION.



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
- INCREMENT
- DECREMENT
- 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
- BRANCH IF ZERO
- BRANCH IF NOT ZERO
- BRANCH IF EQUAL
- BRANCH IF NOT EQUAL
- BRANCH IF CARRY
- BRANCH IF NO CARRY
- BRANCH IF POSITIVE
- 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 NOT HIGHER
- BRANCH IF LOWER
- BRANCH IF NOT LOWER

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:

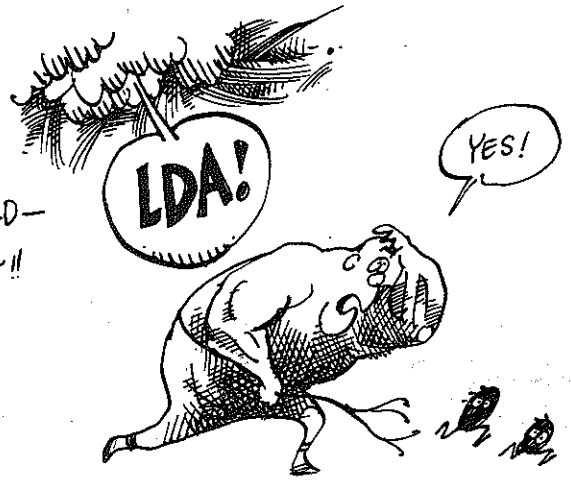
JMP 123

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

EVEN "SMARTER" ARE CONDITIONAL JUMPS. THEY TRANSFER CONTROL **IF** SOME CONDITION IS SATISFIED: FOR INSTANCE, "JUMP IF ZERO" MEANS JUMP IF THE ACCUMULATOR HOLDS 0.

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!

PART III

SOFTWARE