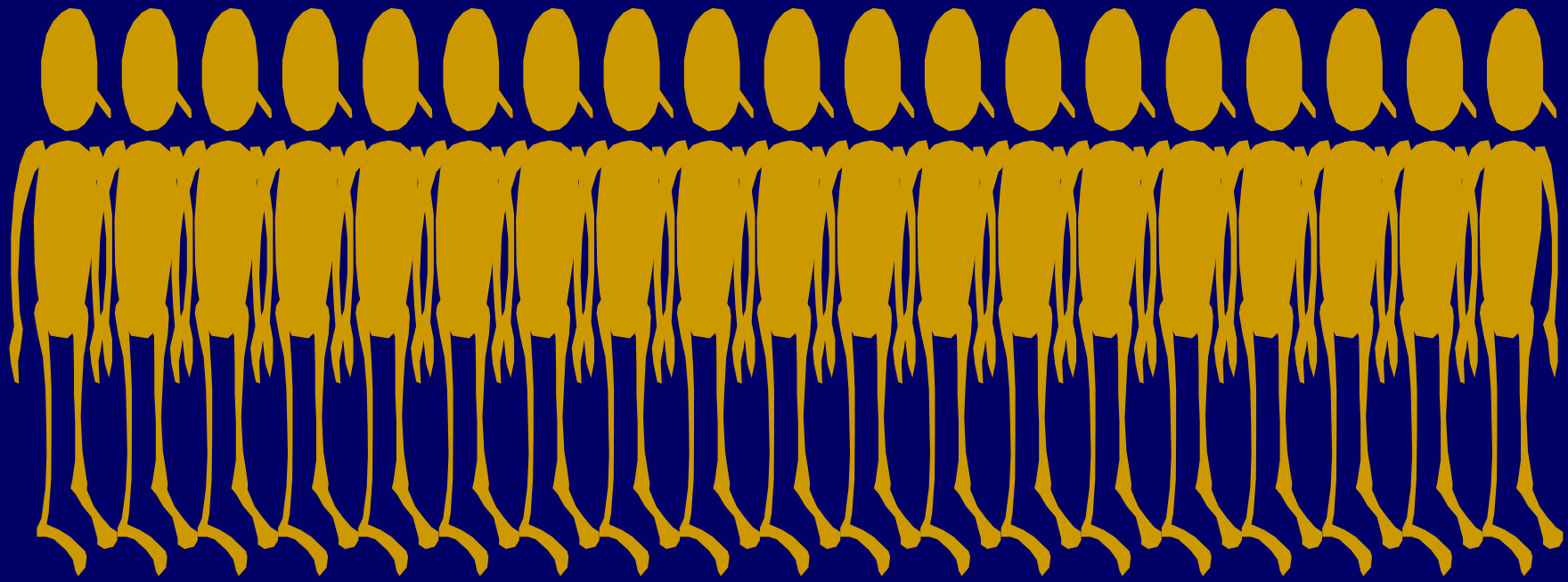
Great Theoretical Ideas In Computer Science

Steven Rudich CS 15-251 Spring 2004

Lecture 17 Mar 16, 2004 Carnegie Mellon University

Grade School Again: A Parallel Perspective

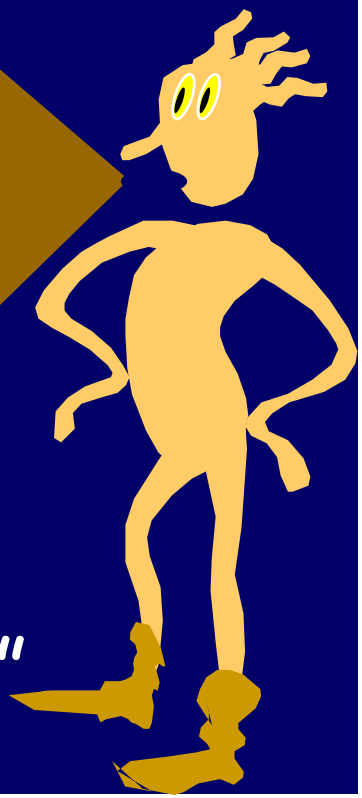


Plus/Minus Binary
(Extended Binary)

Base 2: Each digit can be -1, 0, 1,

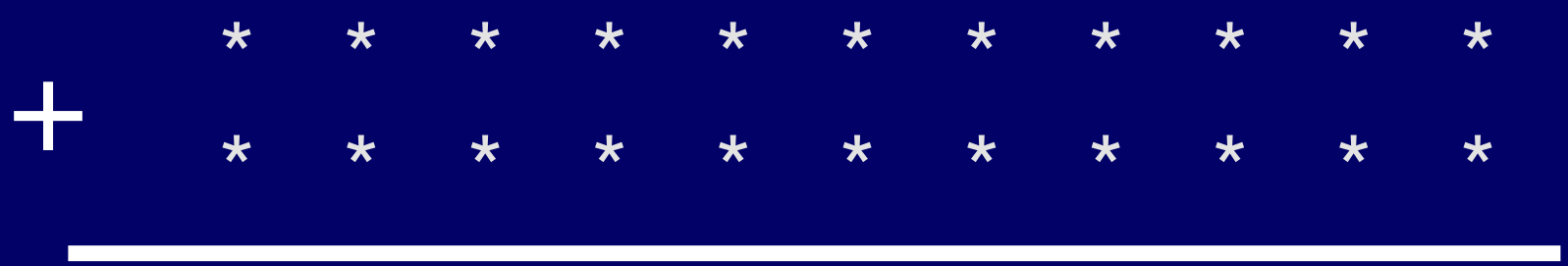
Example:

$$1\ -1\ -1 = 4\ -2\ -1 = 1$$

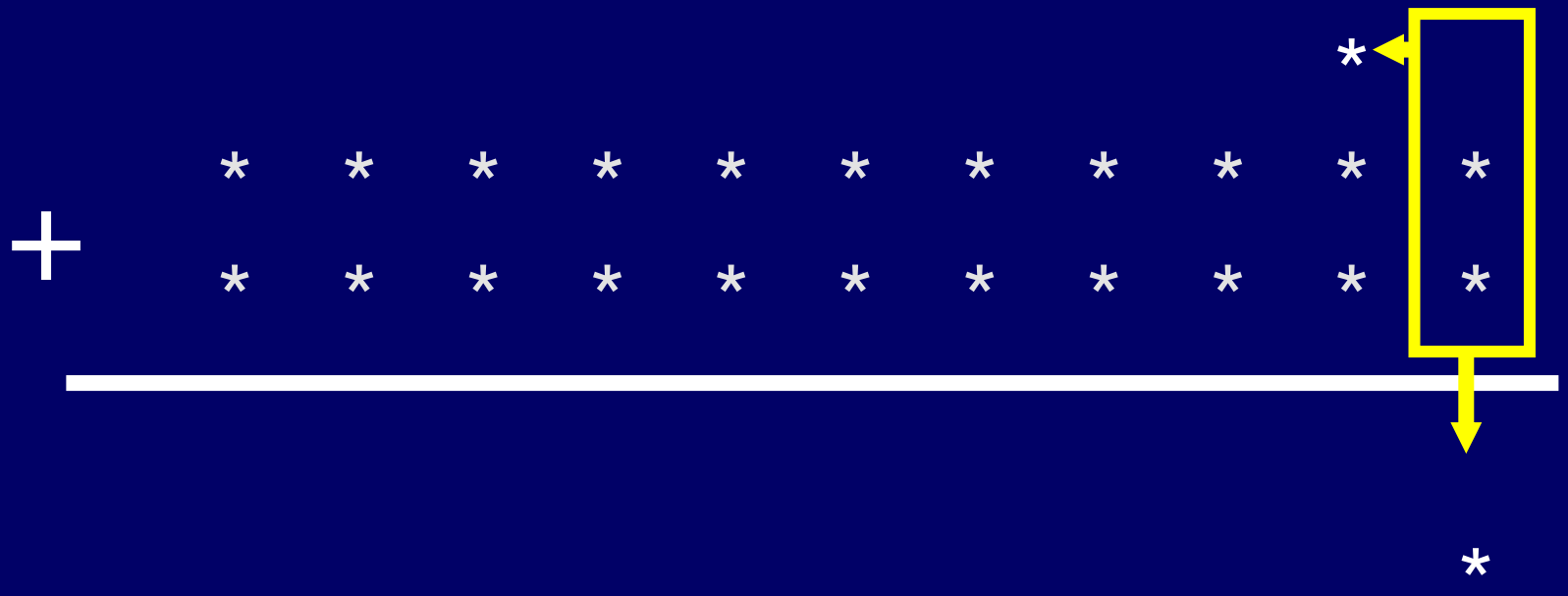


One weight for each power of 3.
Left = "negative". Right = "positive"

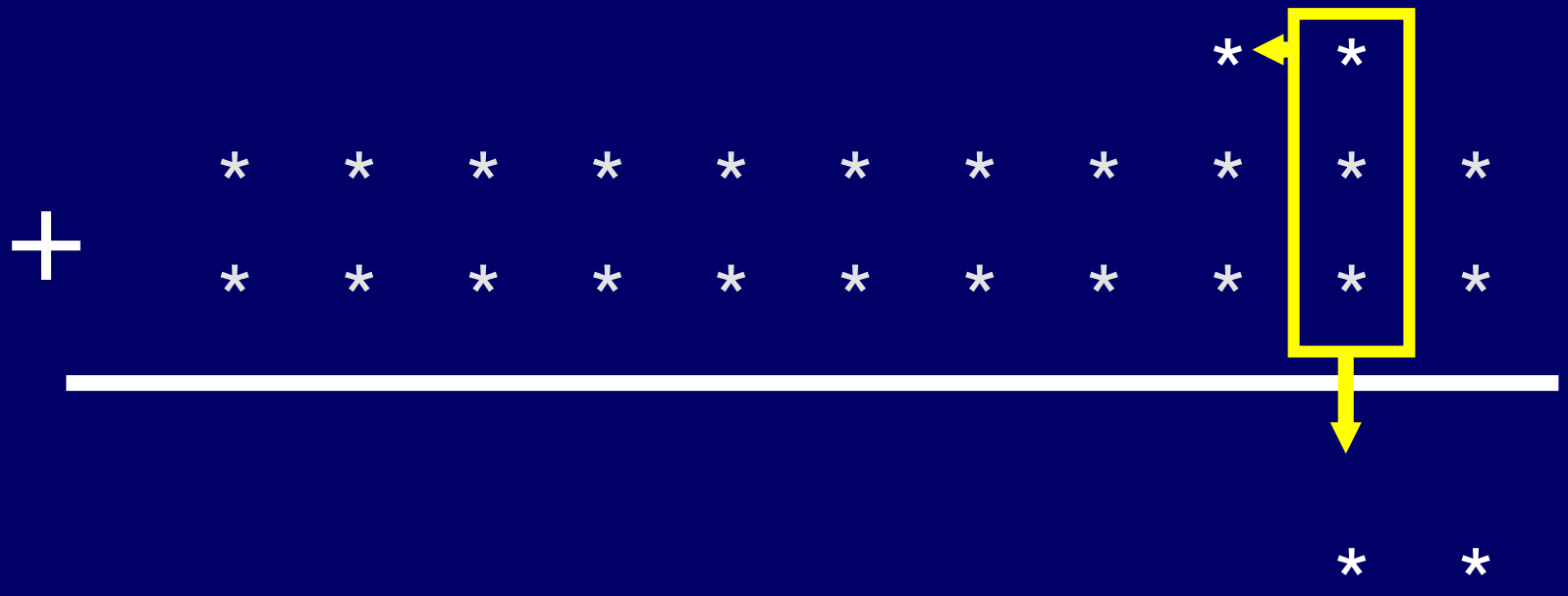
How to add 2 n-bit numbers.



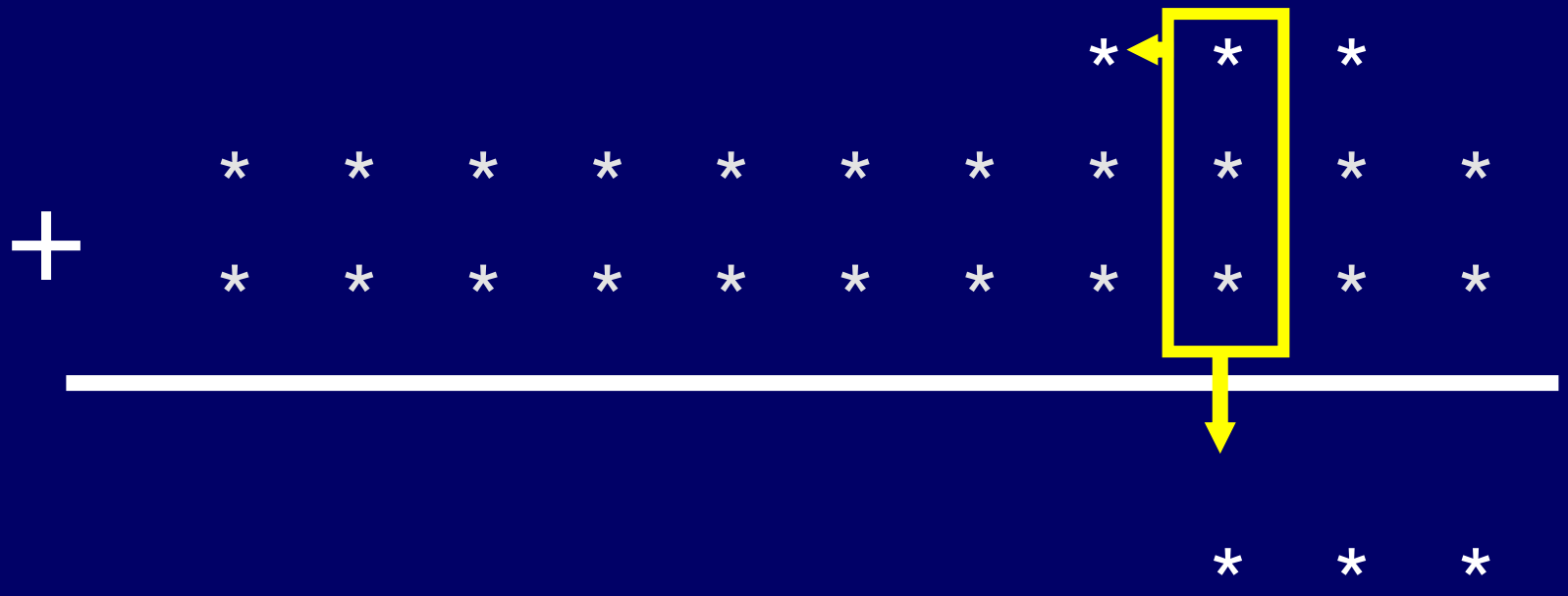
How to add 2 n-bit numbers.



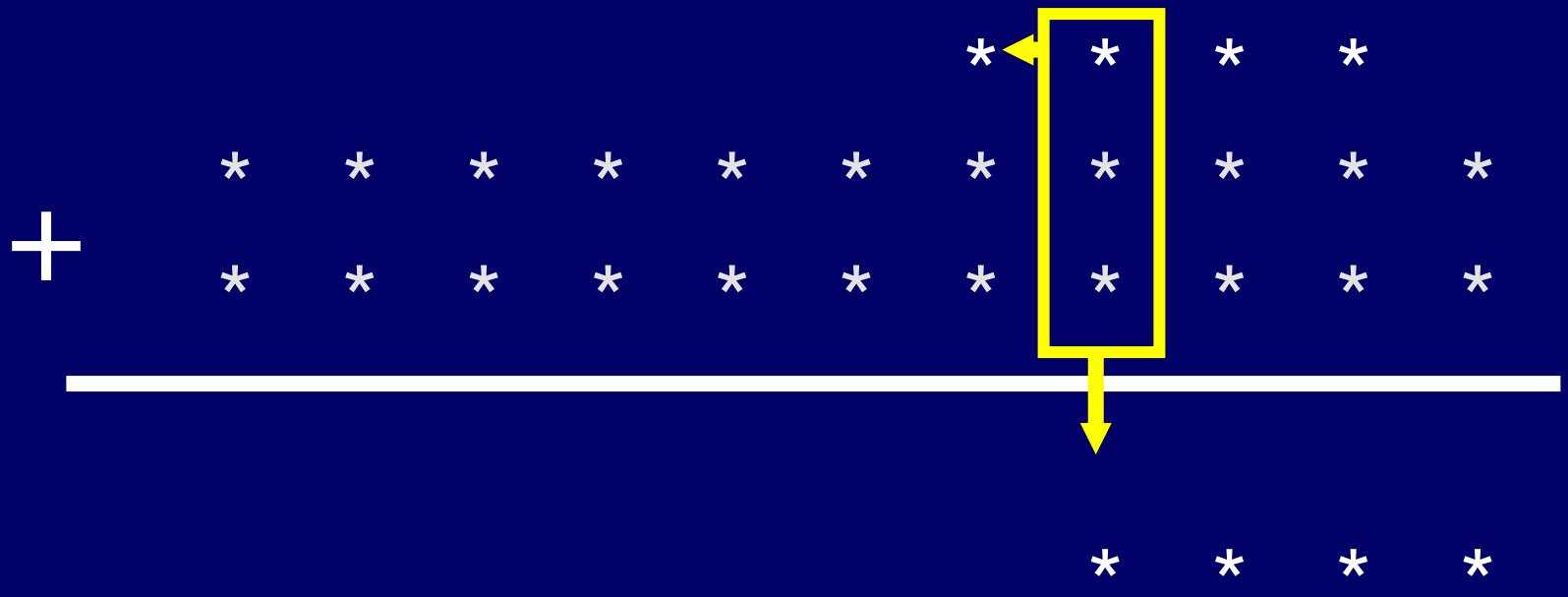
How to add 2 n-bit numbers.



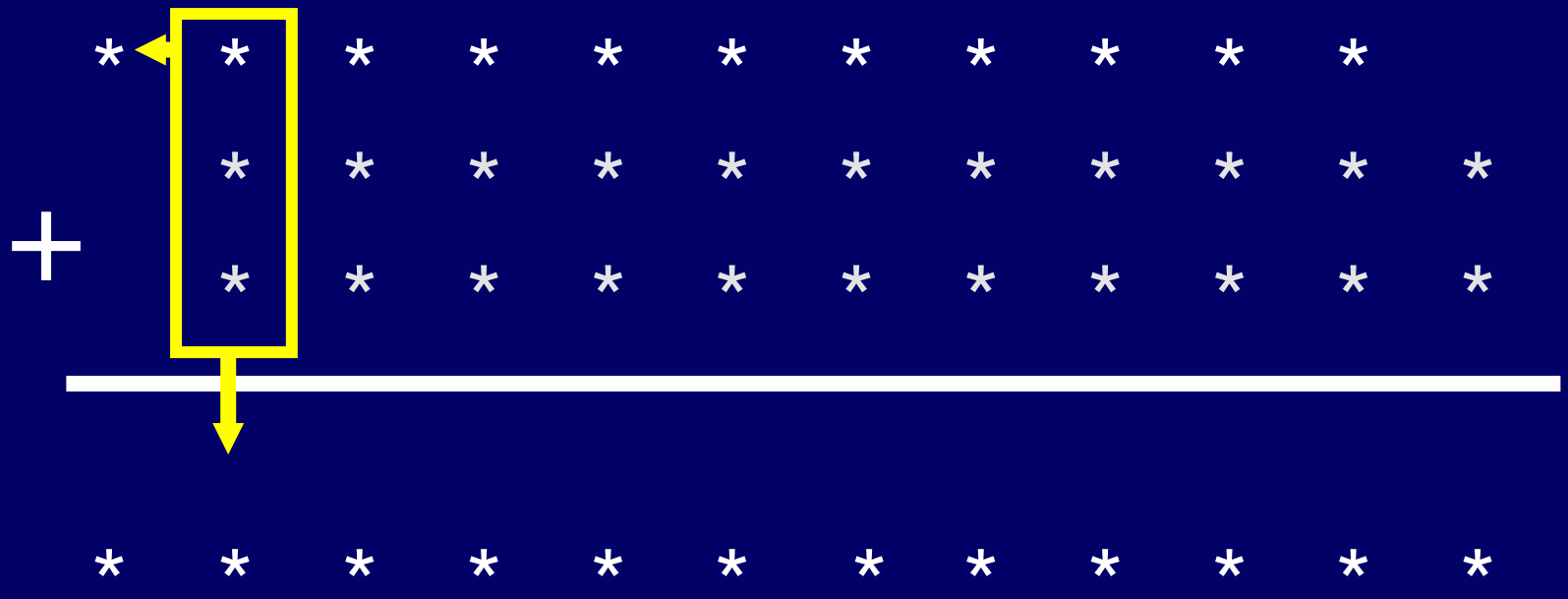
How to add 2 n-bit numbers.



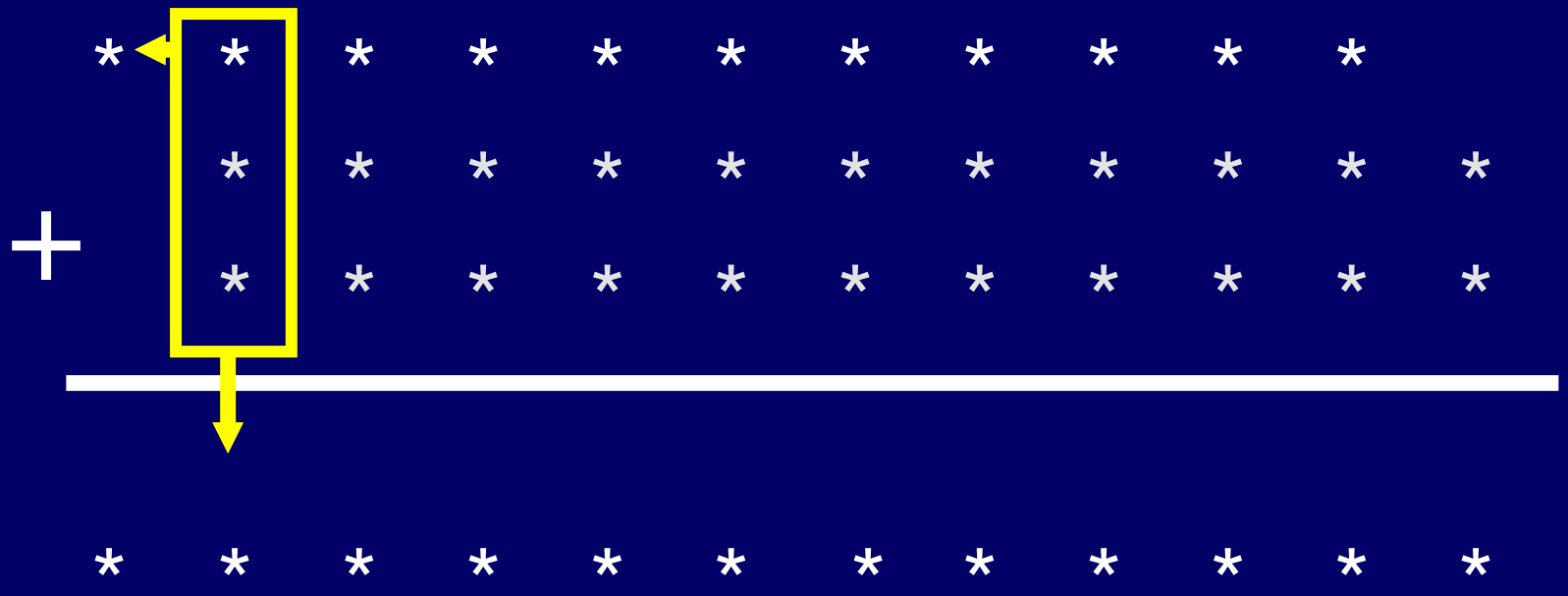
How to add 2 n-bit numbers.



How to add 2 n-bit numbers.

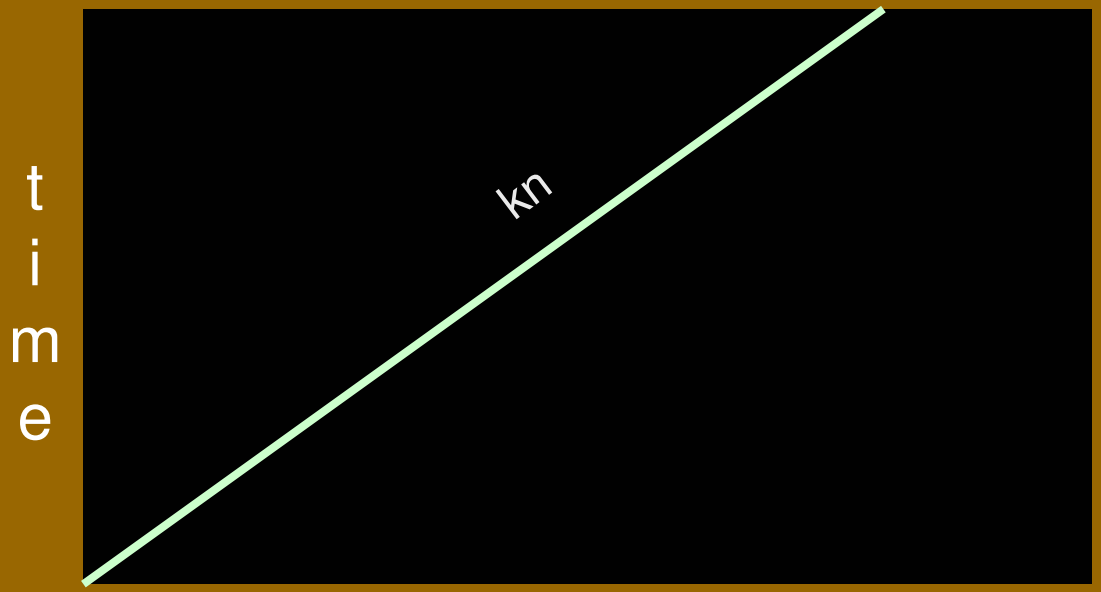


How to add 2 n-bit numbers.

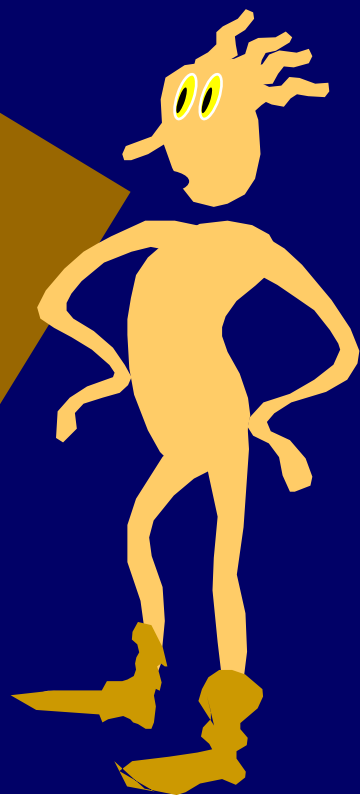


Let k be the maximum time that it takes you to do

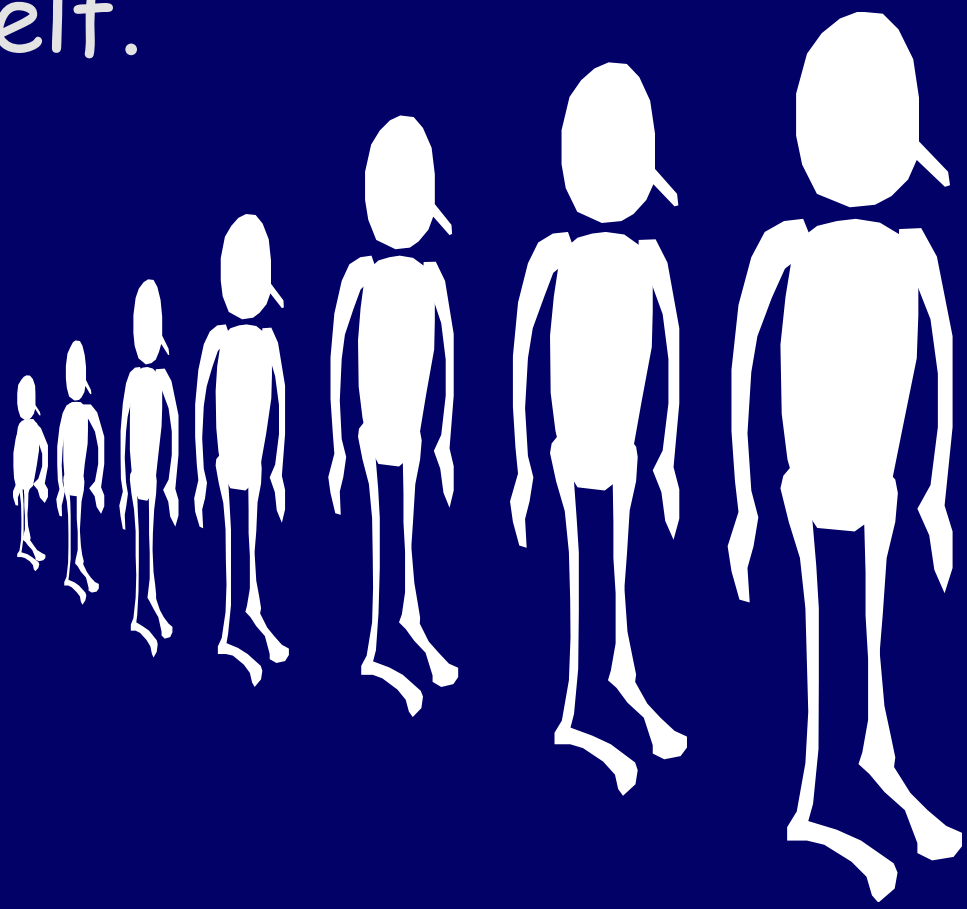
Time < kn is proportional to n



The time grow linearly with input size.

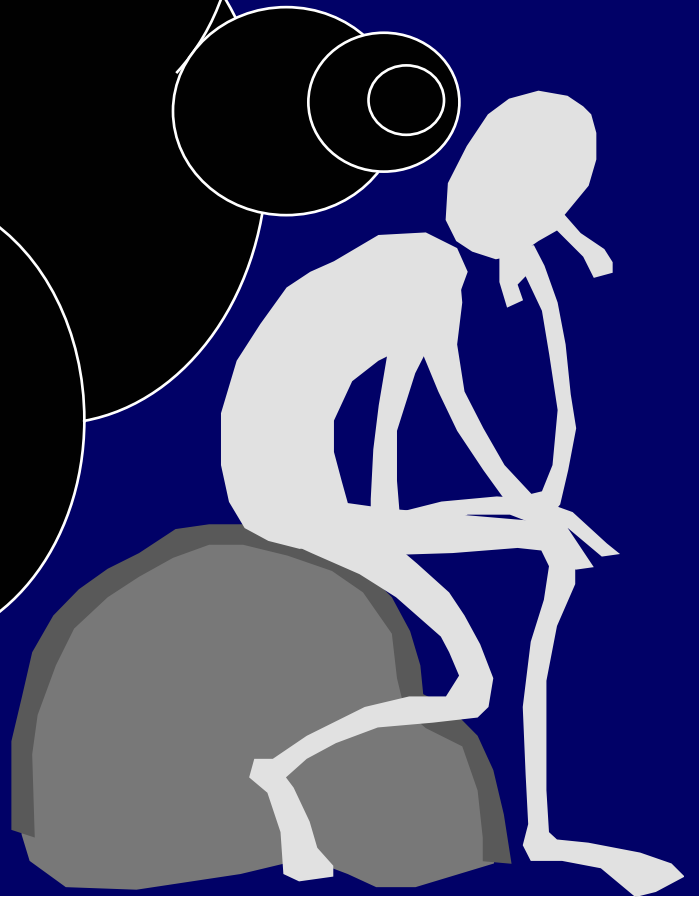


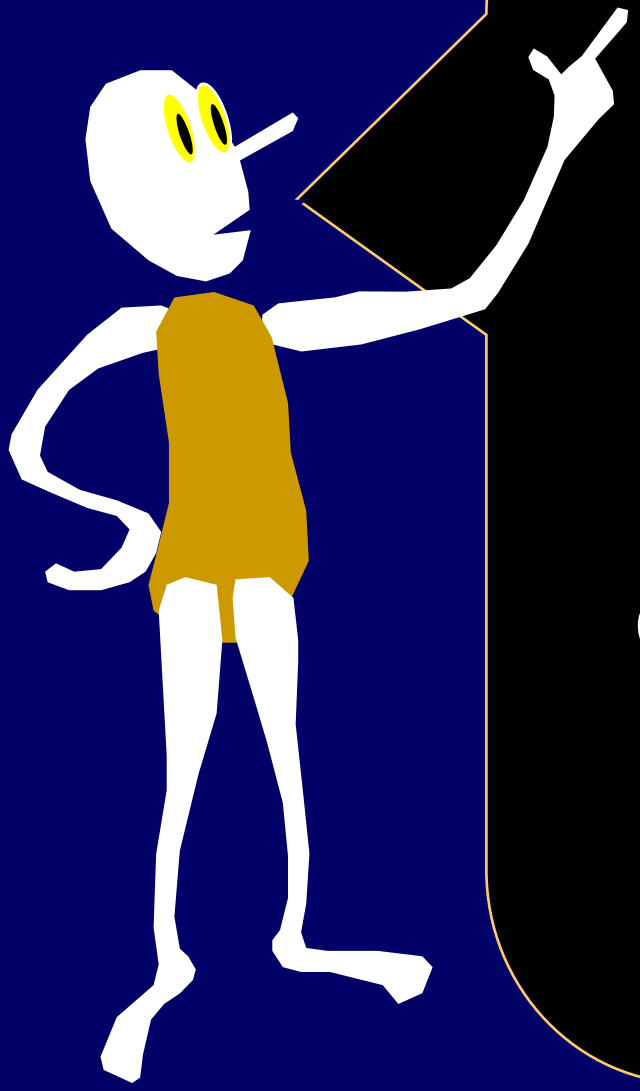
If n people agree to help you add two n bit numbers, it is not obvious that they can finish faster than if you had done it yourself.



Is it possible to add two n bit numbers in less than linear parallel-time?

Darn those carries.

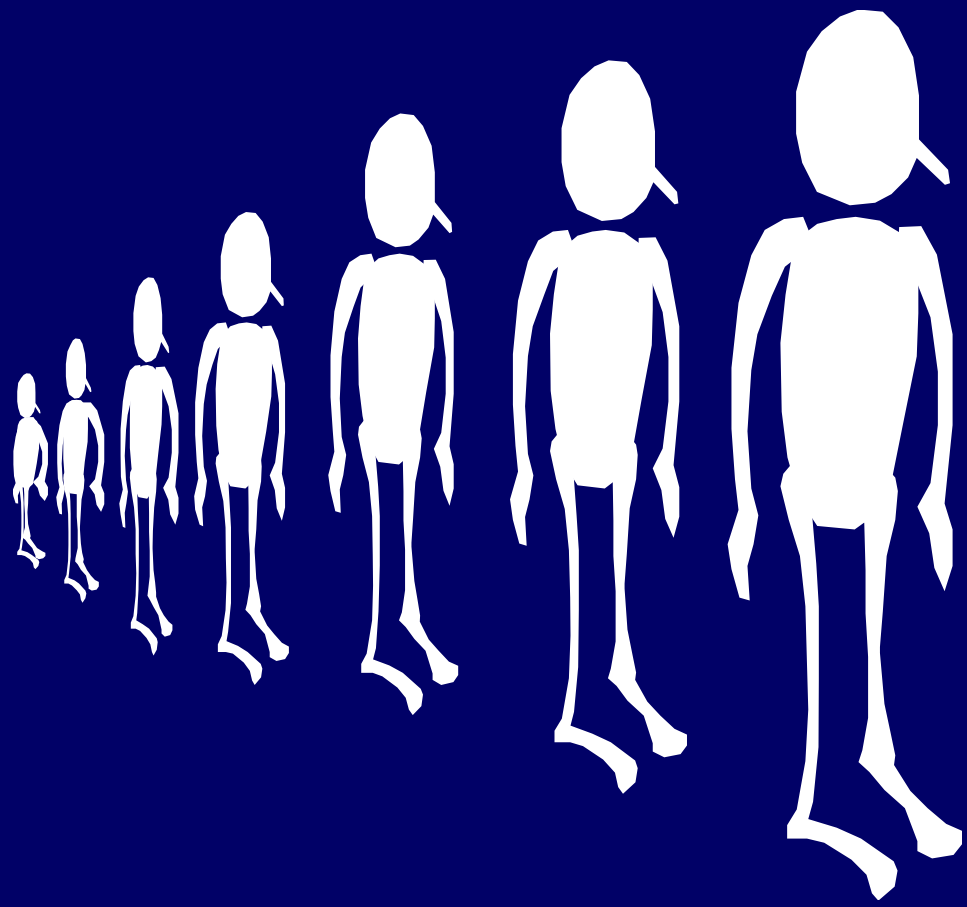




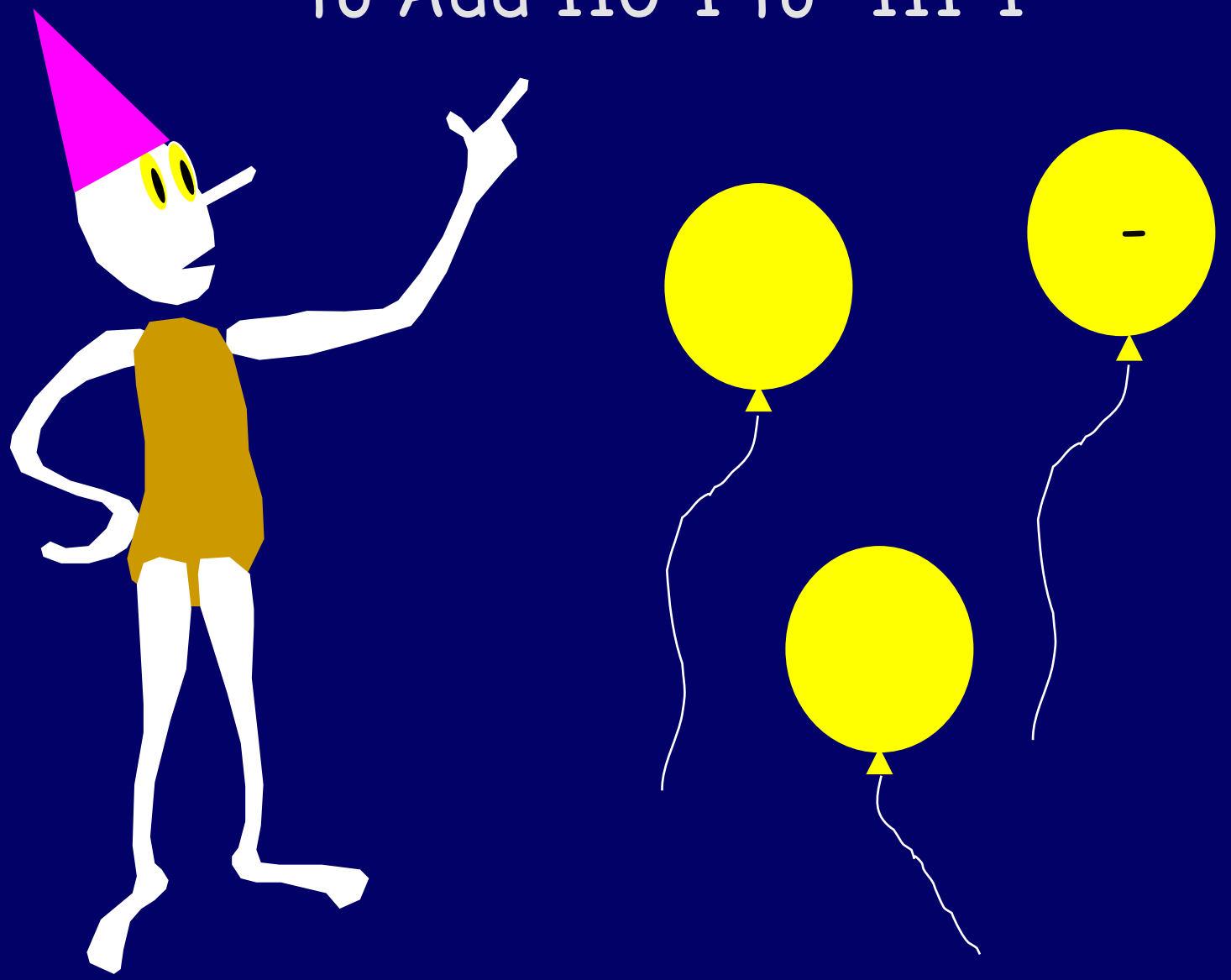
Fast parallel addition is no obvious in usual binary. But it is amazingly direct in Extended Binary!

Extended binary means base 2 allowing digits to be from {-1, 0, 1}. We can call each digit a "trit".

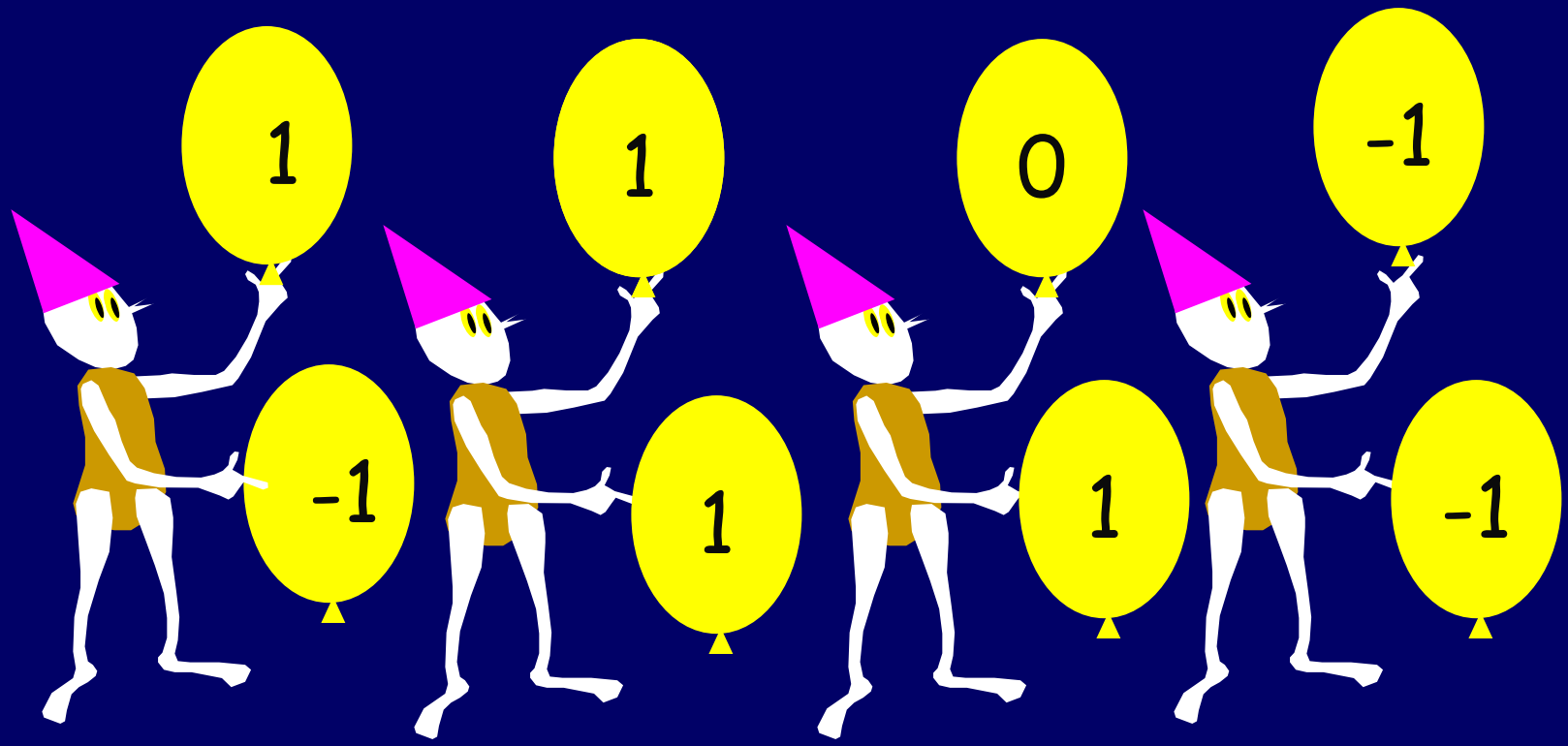
n people can add 2, n-trit, plus/minus binary numbers in constant time!



An Addition Party to Add 110-1 to -111-1



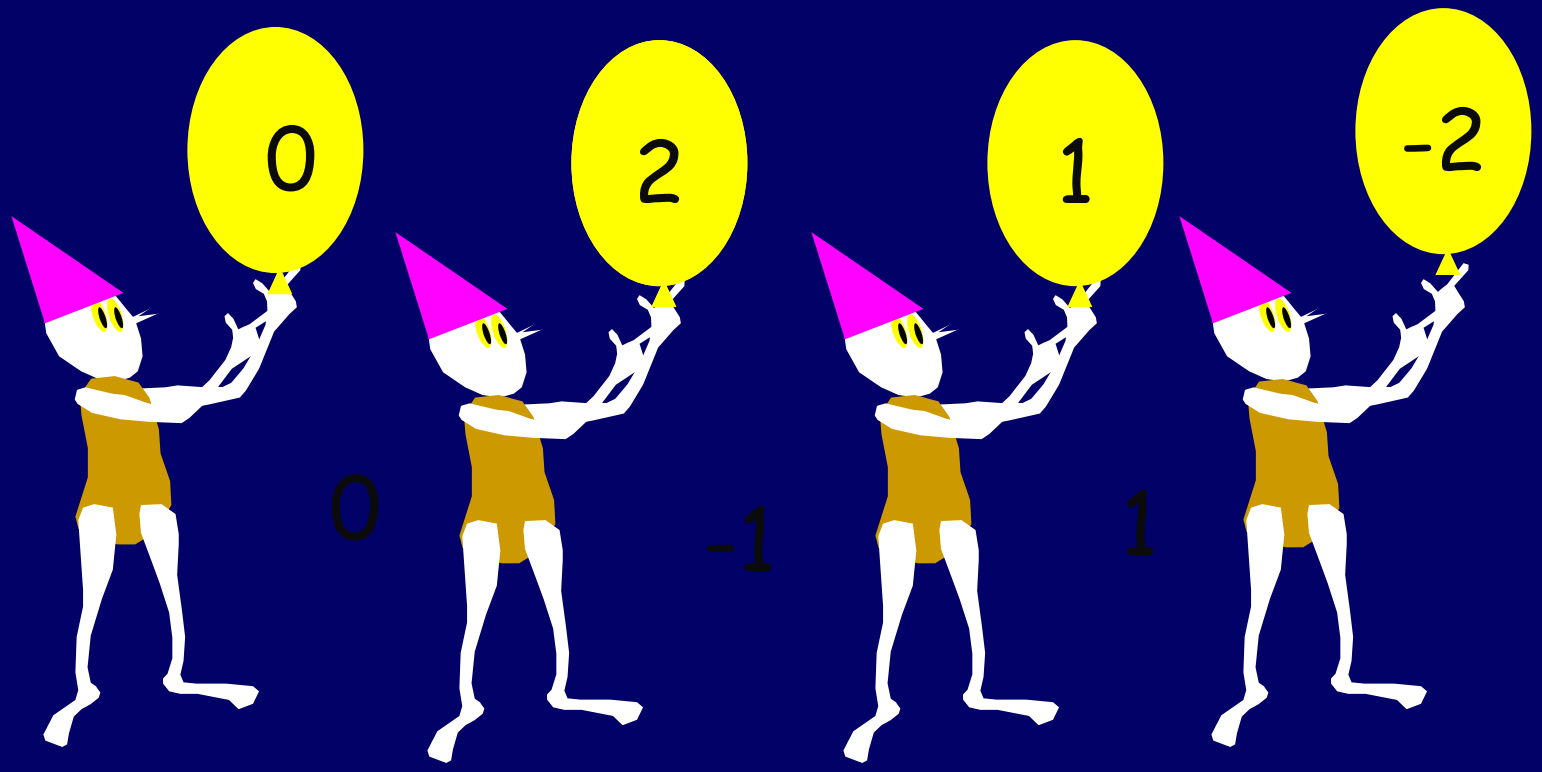
An Addition Party



Invite n people to add two n-trit numbers

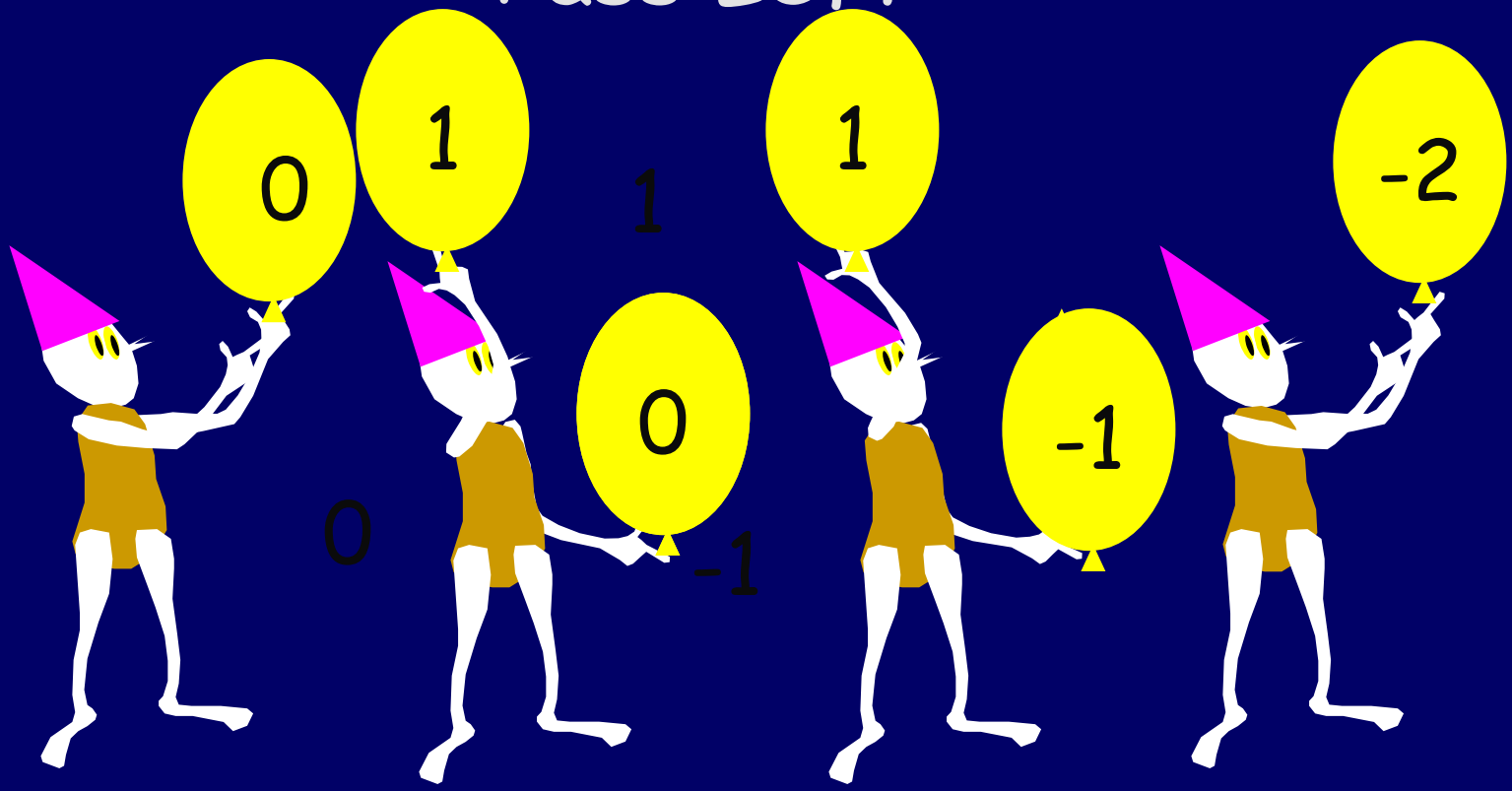
Assign one person to each trit position

An Addition Party



Each person should add the two input trits in their possession. Problem: 2 and -2 are not allowed in the final answer.

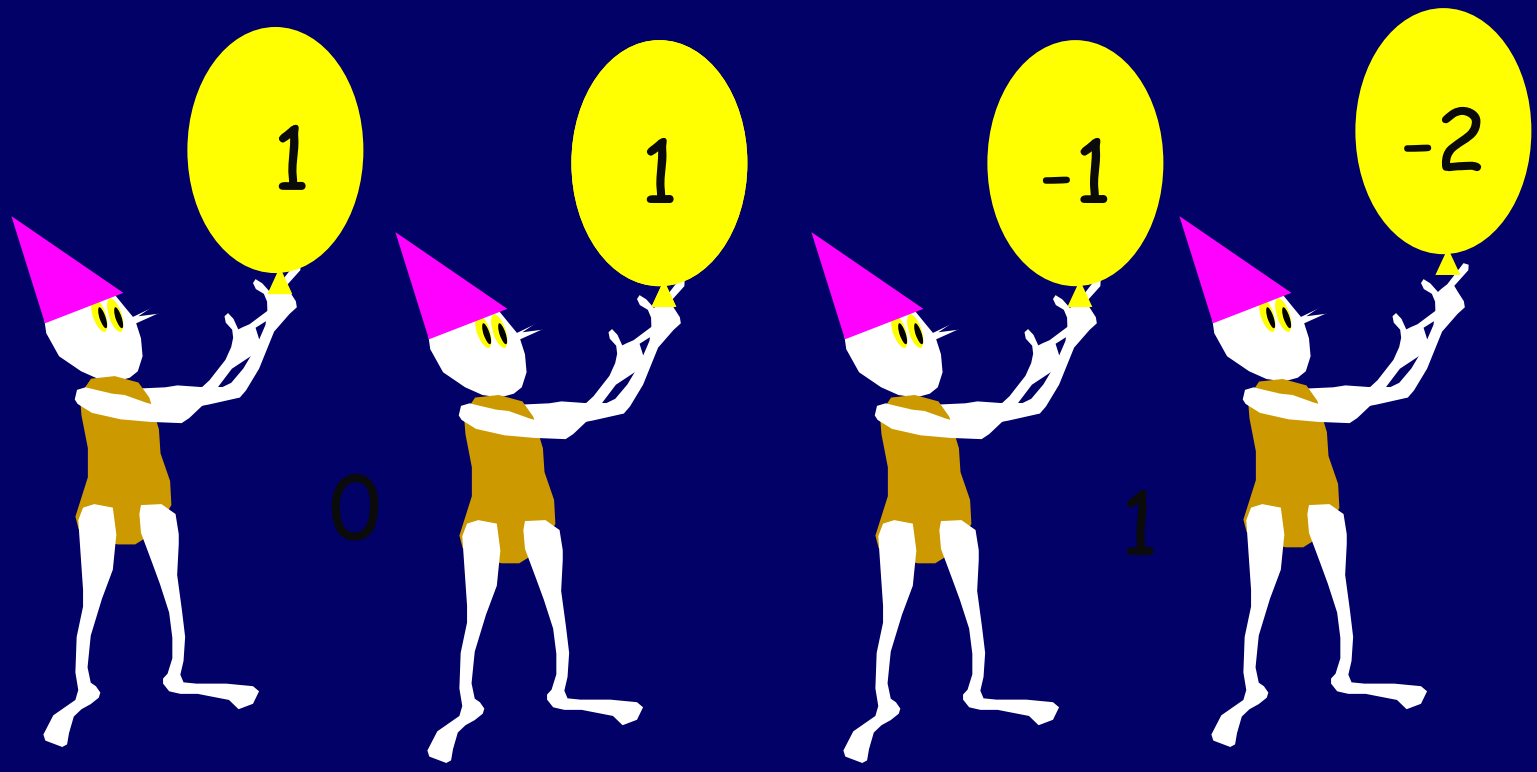
Pass Left



If you have a 1 or a 2 subtract 2 from yourself and pass a 1 to the left. (Nobody keeps more than 0)

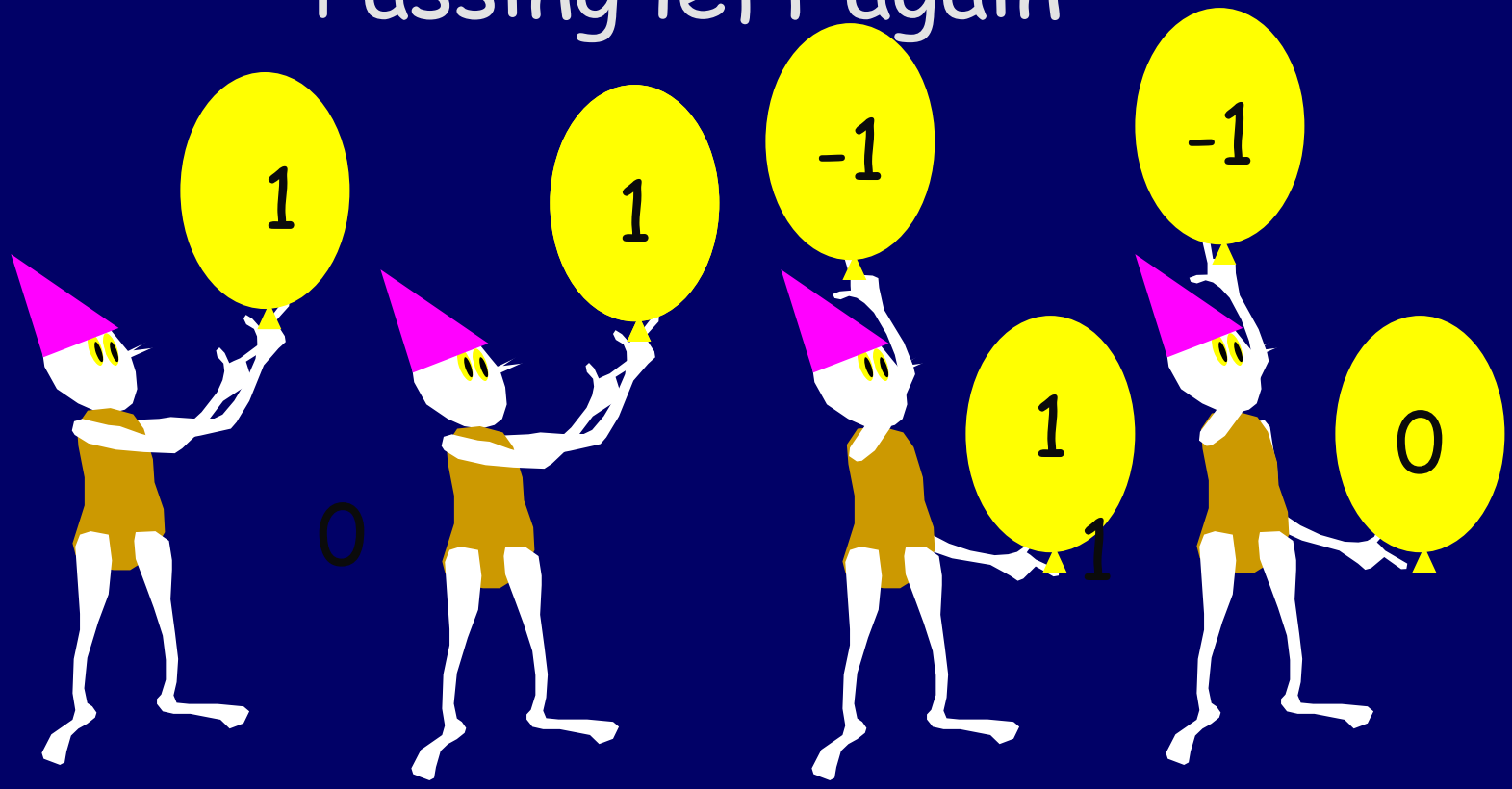
Add in anything that is given to you from the right. (Nobody has more than a 1)

After passing left



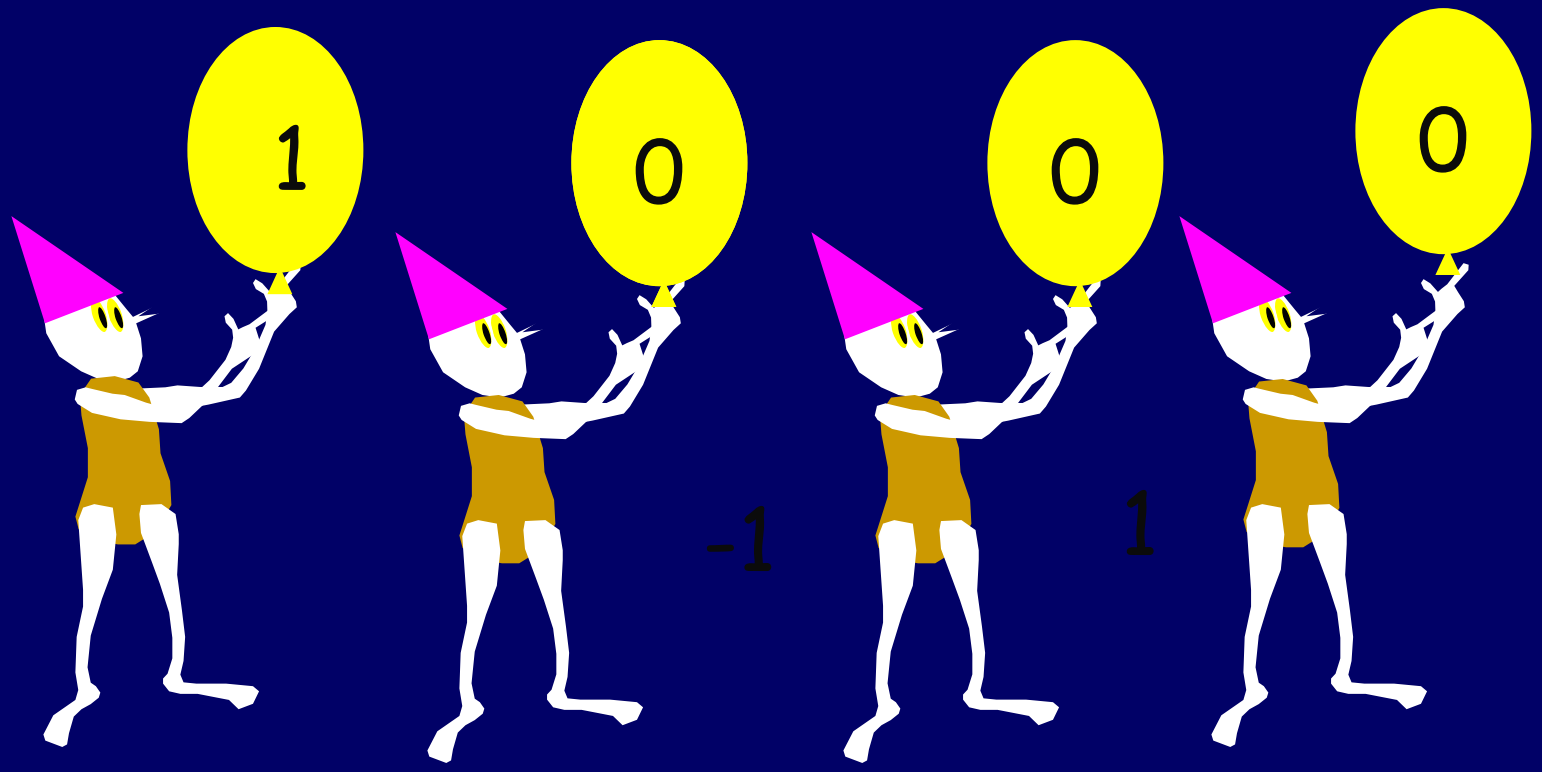
There will never again be any 2s
as everyone had at most 0
and received at most 1 more

Passing left again



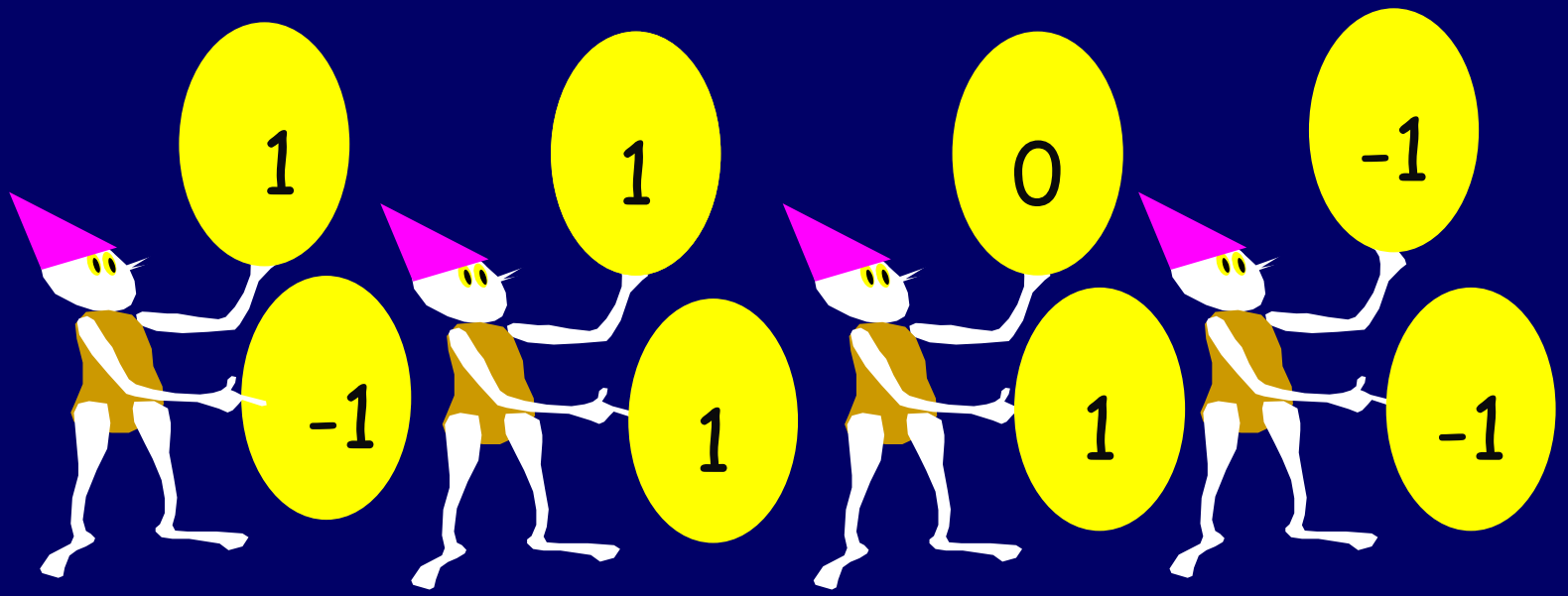
If you have a -1 or -2 add 2 to yourself and pass a -1 to the left
(Nobody keeps less than 0)

After passing left again

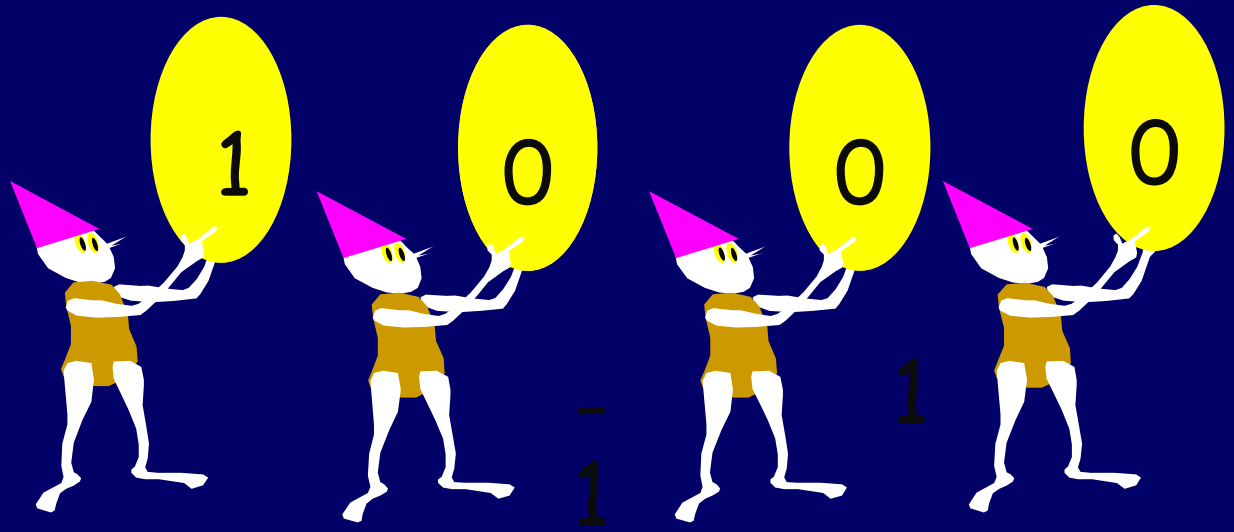


No -2s anymore either.

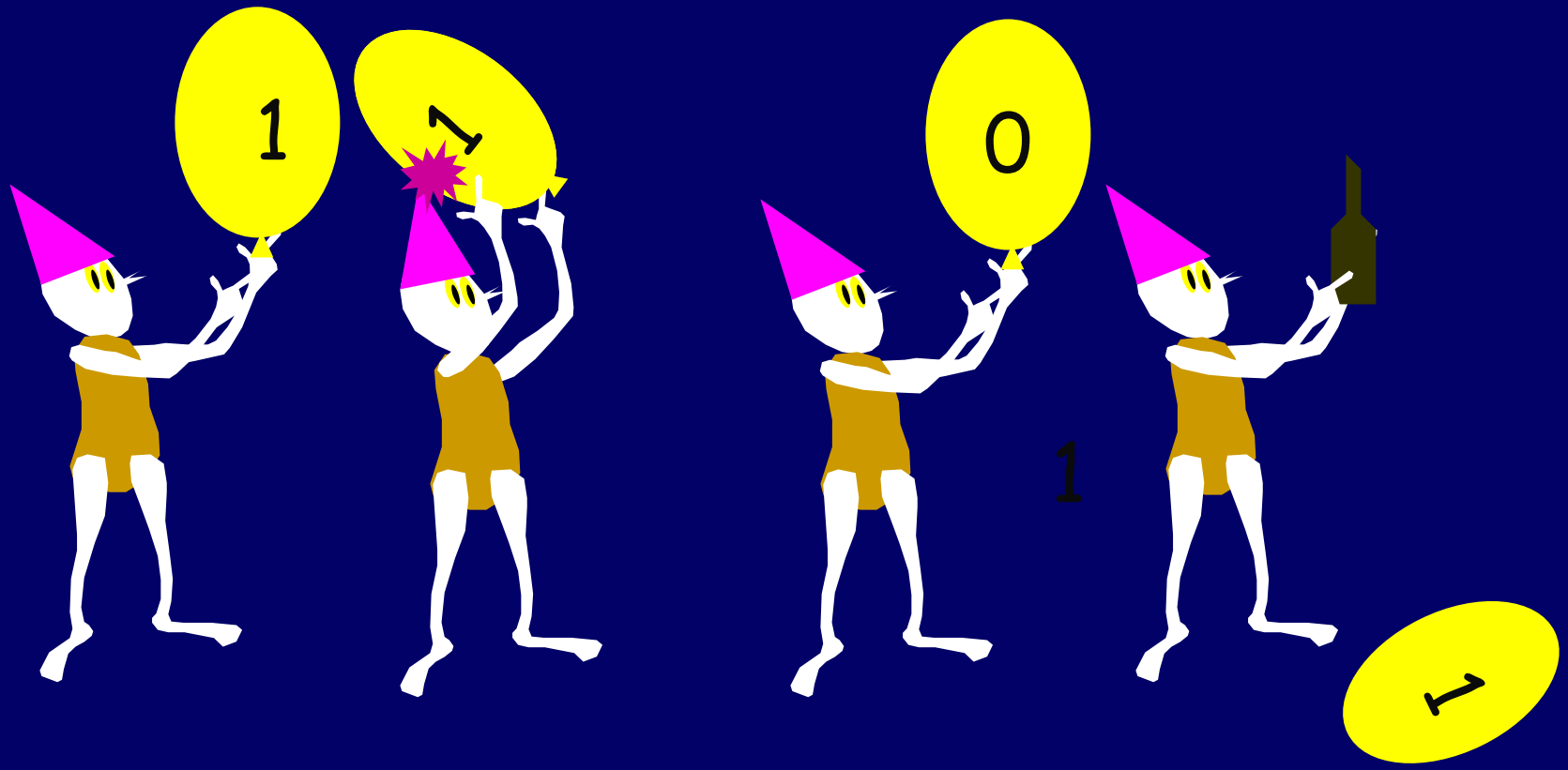
Everyone kept at least 0 and received At most -1.

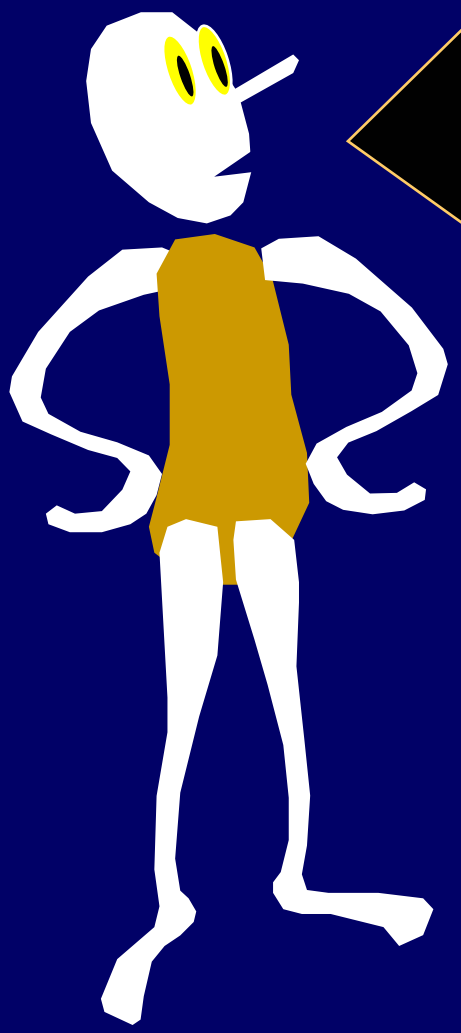

1
-1
1
1
0
1
-1
-1

=

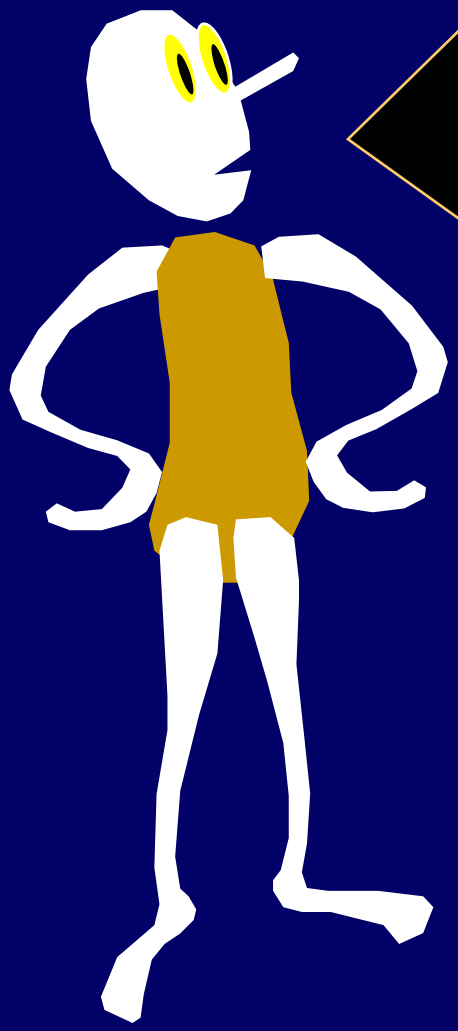

1
0
-
1
0
1
0

Caution:
Parties and Algorithms Do not Mix





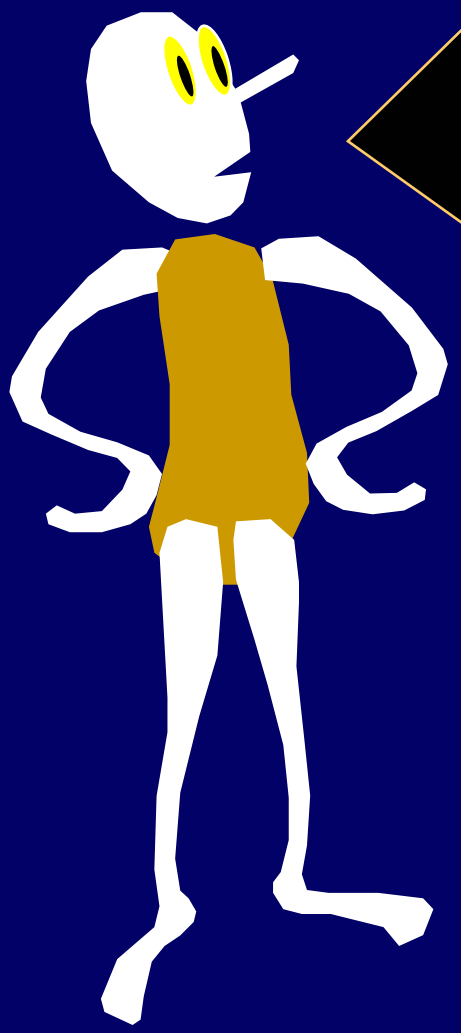
Is there a fast parallel way to convert an Extended Binary number into a standard binary number?



Not obvious:

Sub-linear addition in standard Binary.

Sub-linear EB to Binary



Let's reexamine grade school addition from the view of a computer circuit.

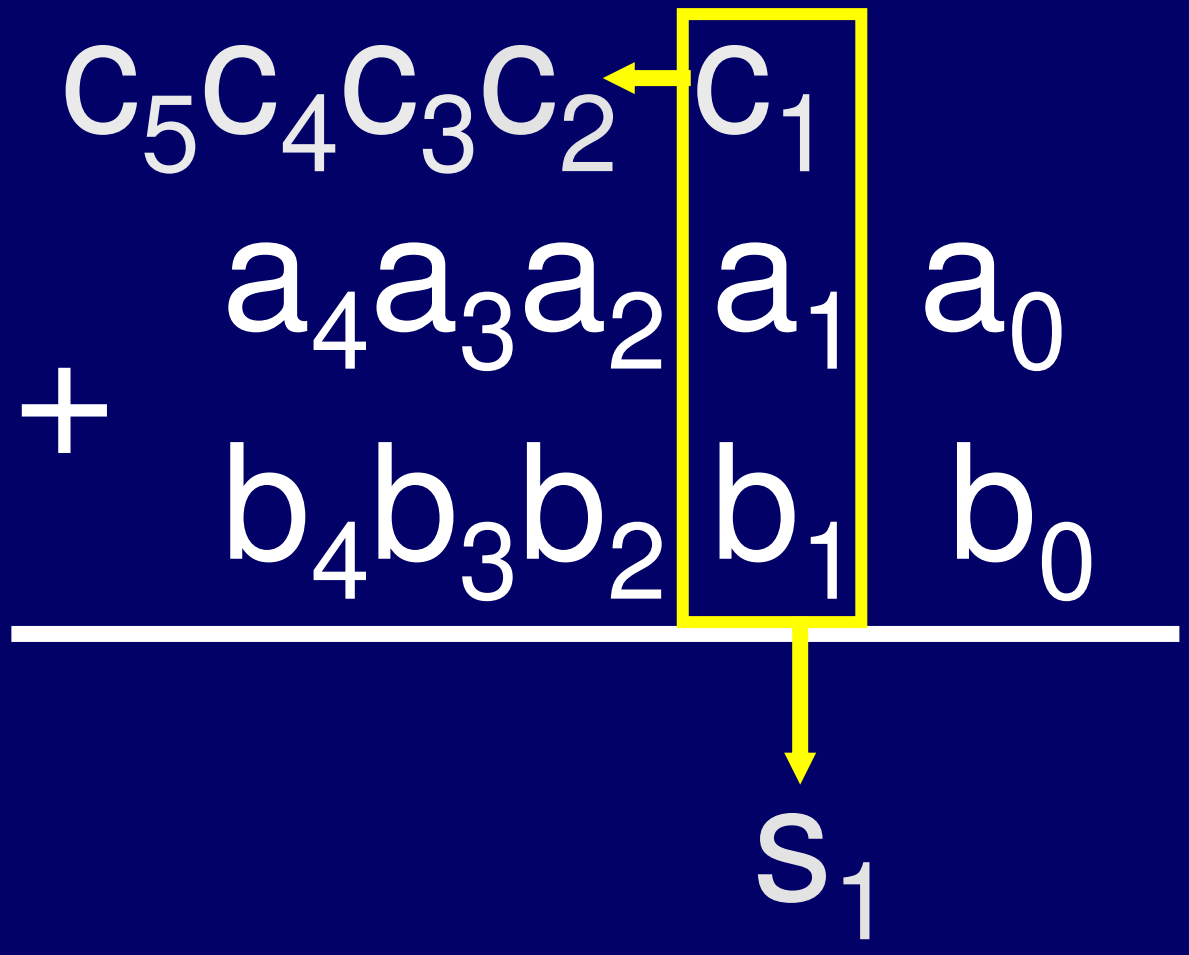
Grade School Addition

$$
\begin{array}{r}
1011111100 \\
1011111101 \\
+\quad 1000000110 \\
\hline
1010000011
\end{array}
$$

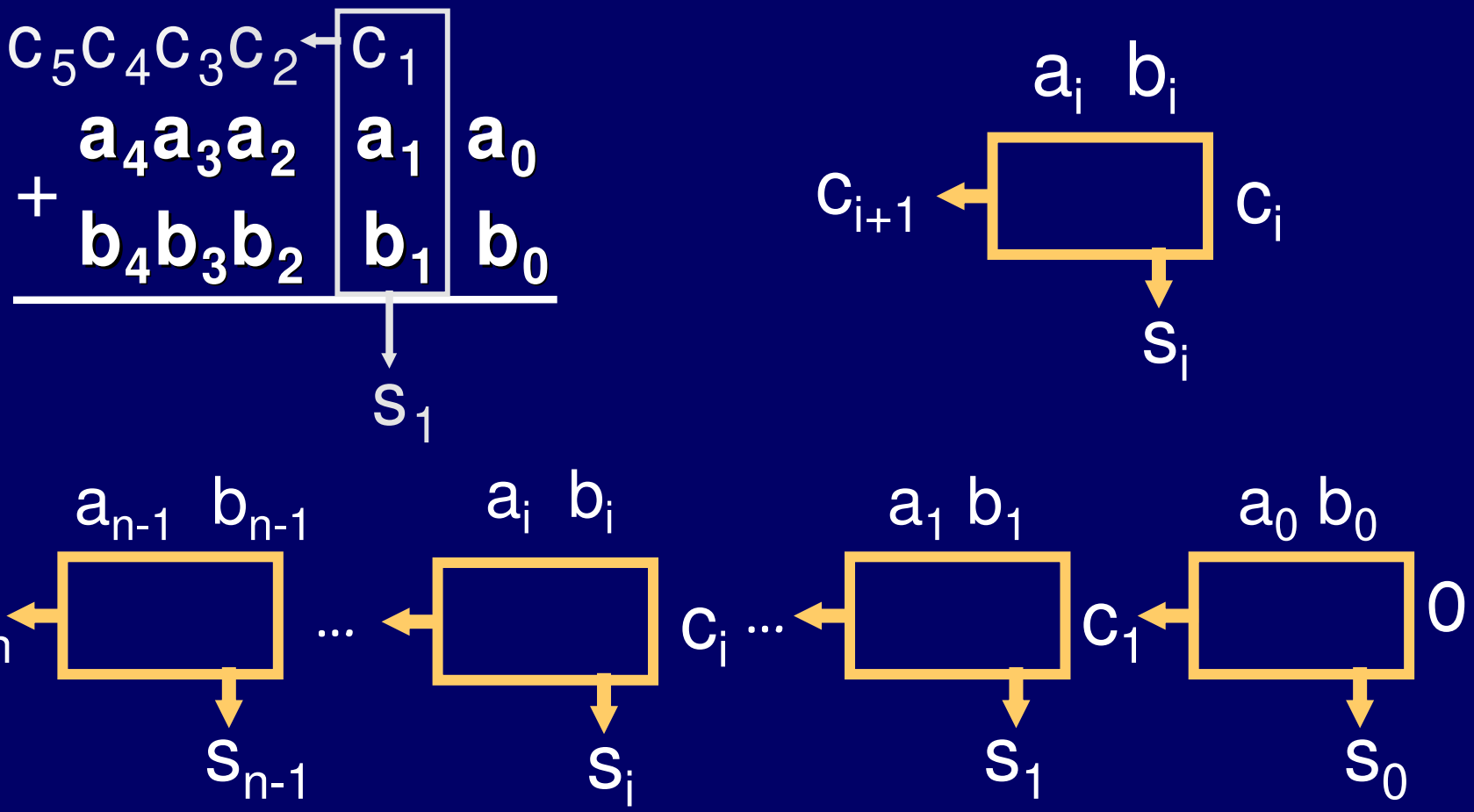
Grade School Addition

$$
\begin{array}{r}
c_5 c_4 c_3 c_2 c_1 \\
\quad a_4 a_3 a_2 a_1 a_0 \\
+ \quad b_4 b_3 b_2 b_1 b_0 \\
\hline
\end{array}
$$

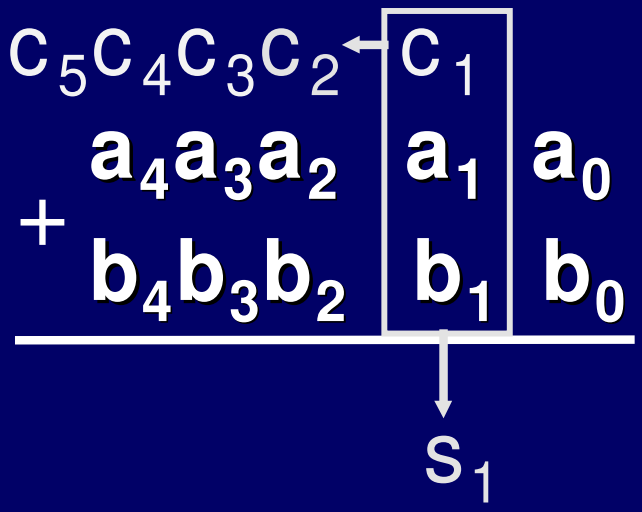
Grade School Addition

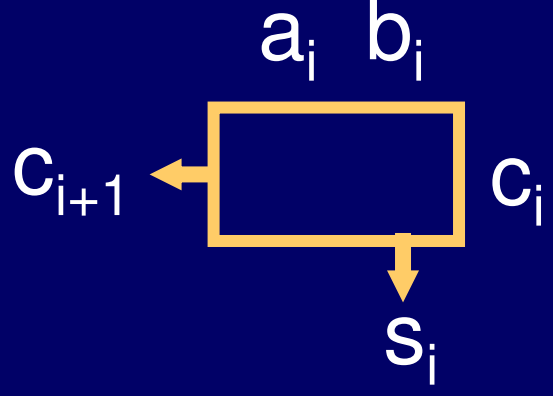


Ripple-carry adder



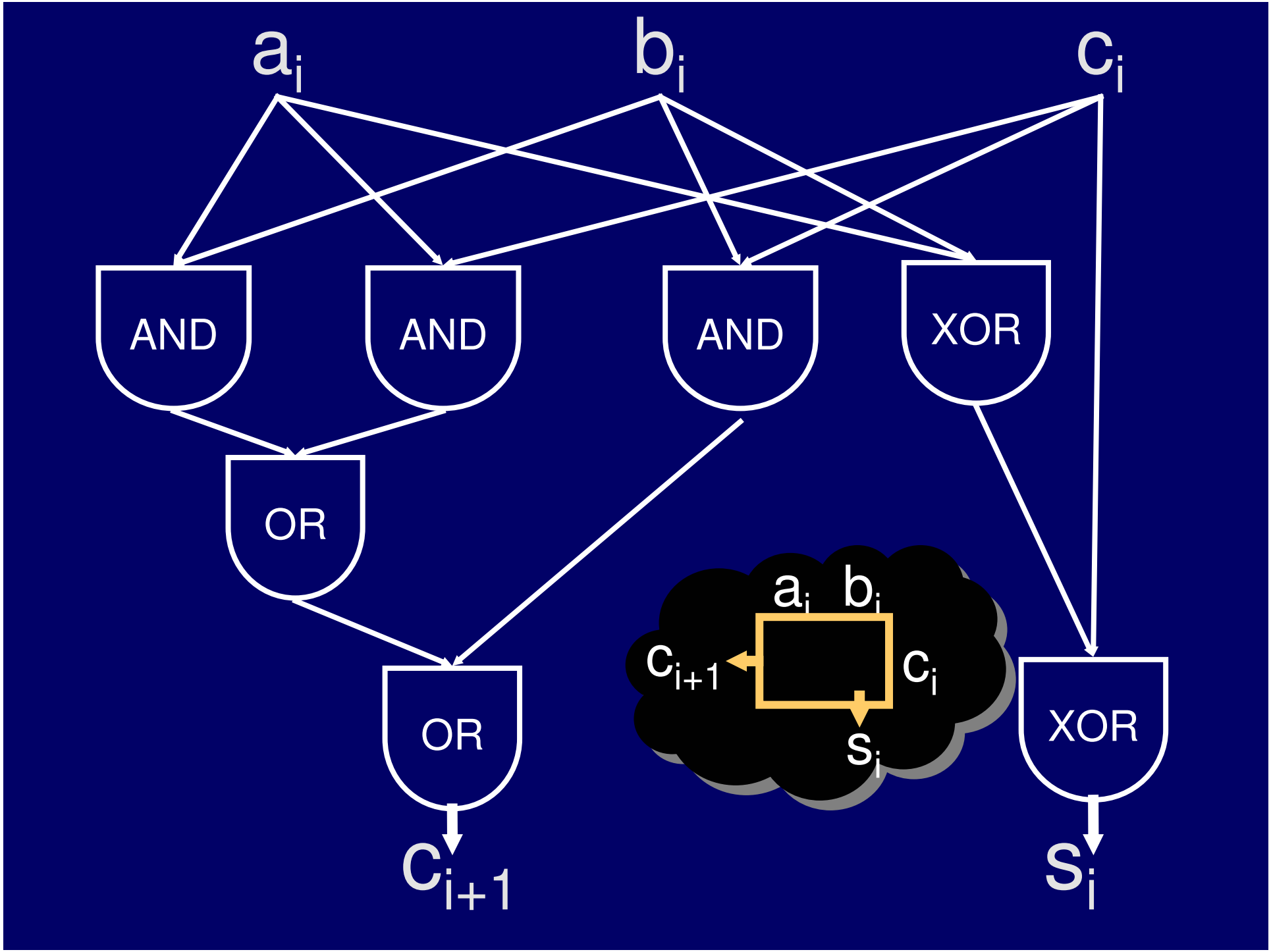
Logical representation of binary: 0 = false, 1 = true



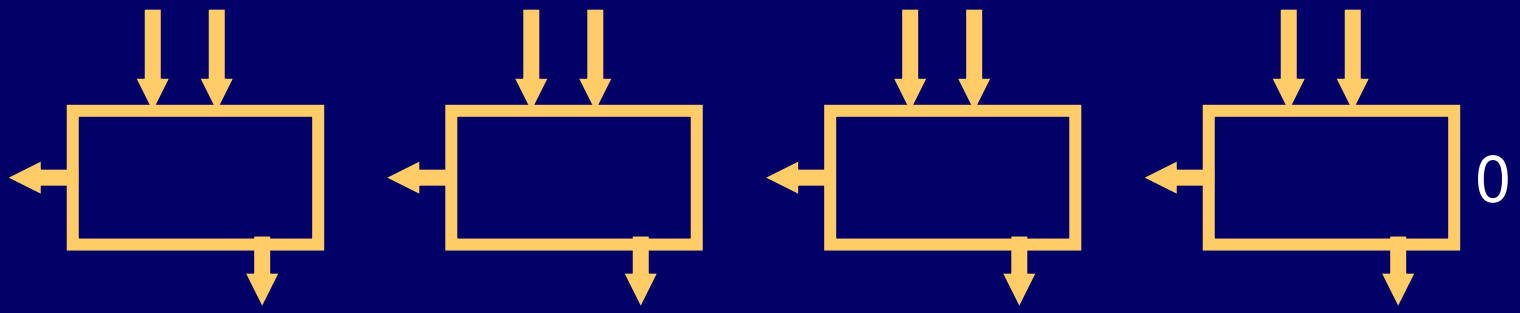


$s_1 = (a_1 \text{ XOR } b_1) \text{ XOR } c_1$

$c_2 = (a_1 \text{ AND } b_1)$
$\text{OR } (a_1 \text{ AND } c_1)$
$\text{OR } (b_1 \text{ AND } c_1)$

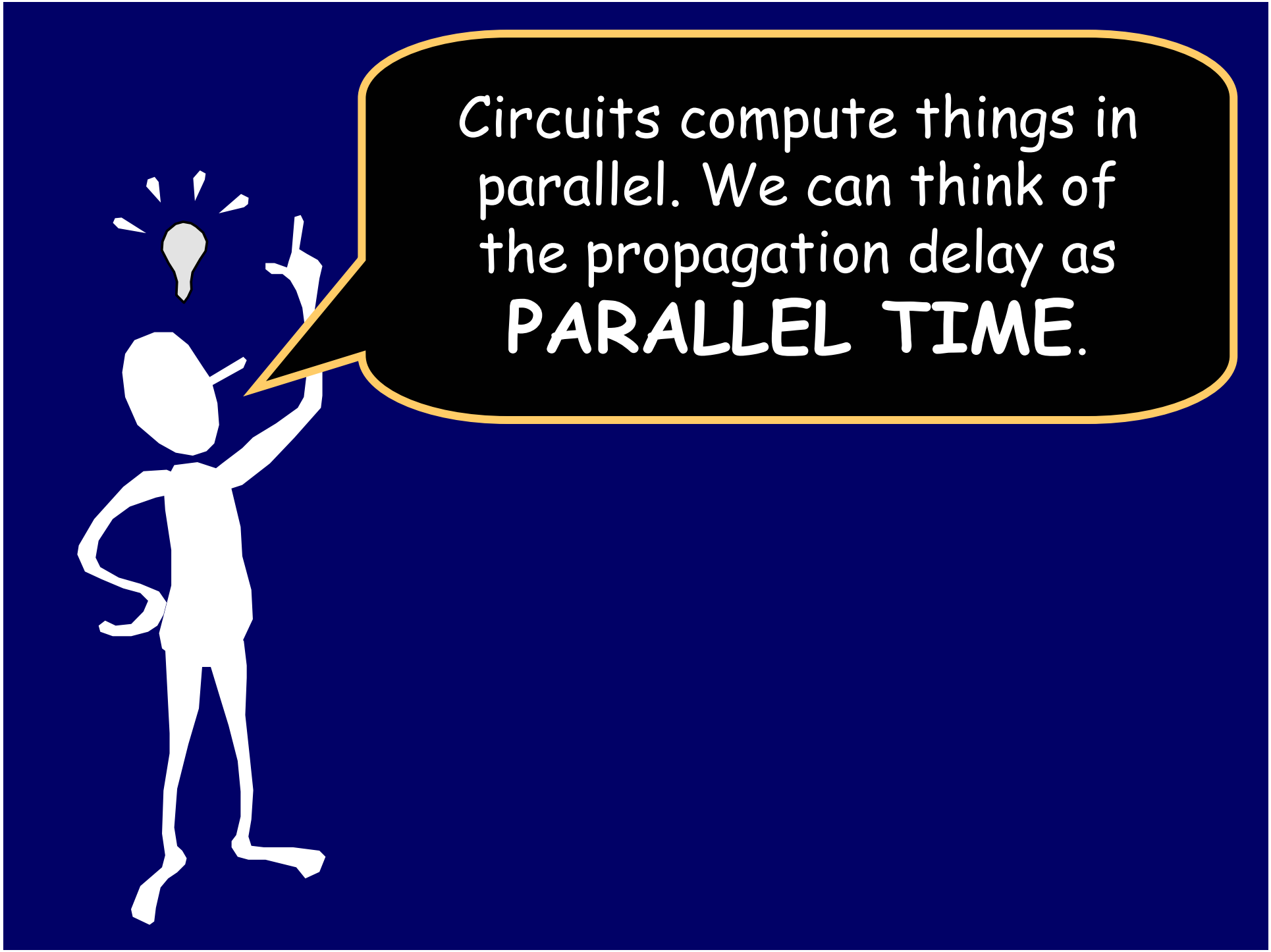

a_i
b_i
c_i
AND
AND
AND
XOR
OR
OR
a_i b_i
c_{i+1}
c_i
s_i
XOR
c_{i+1}
s_i

Ripple-carry adder

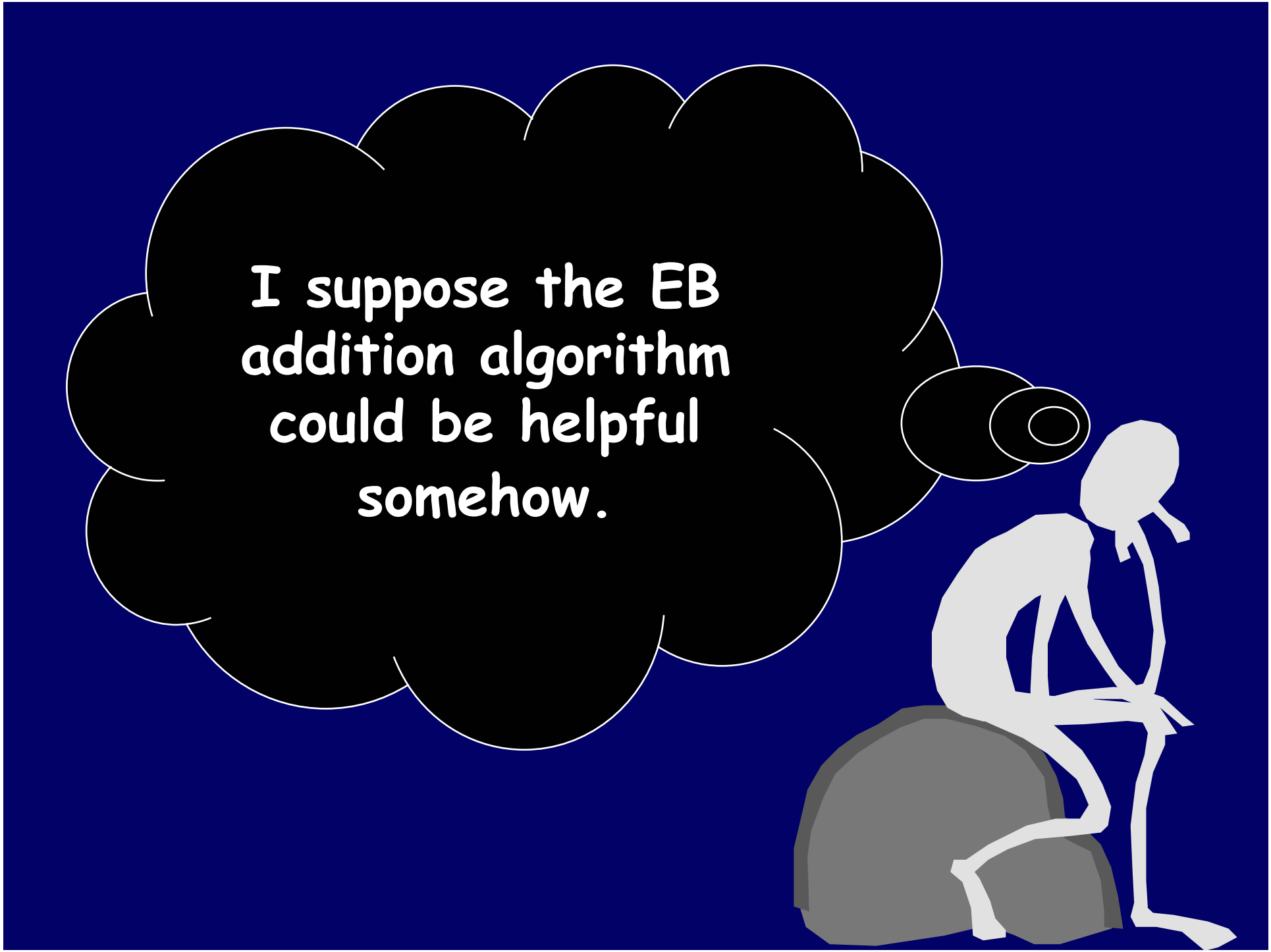


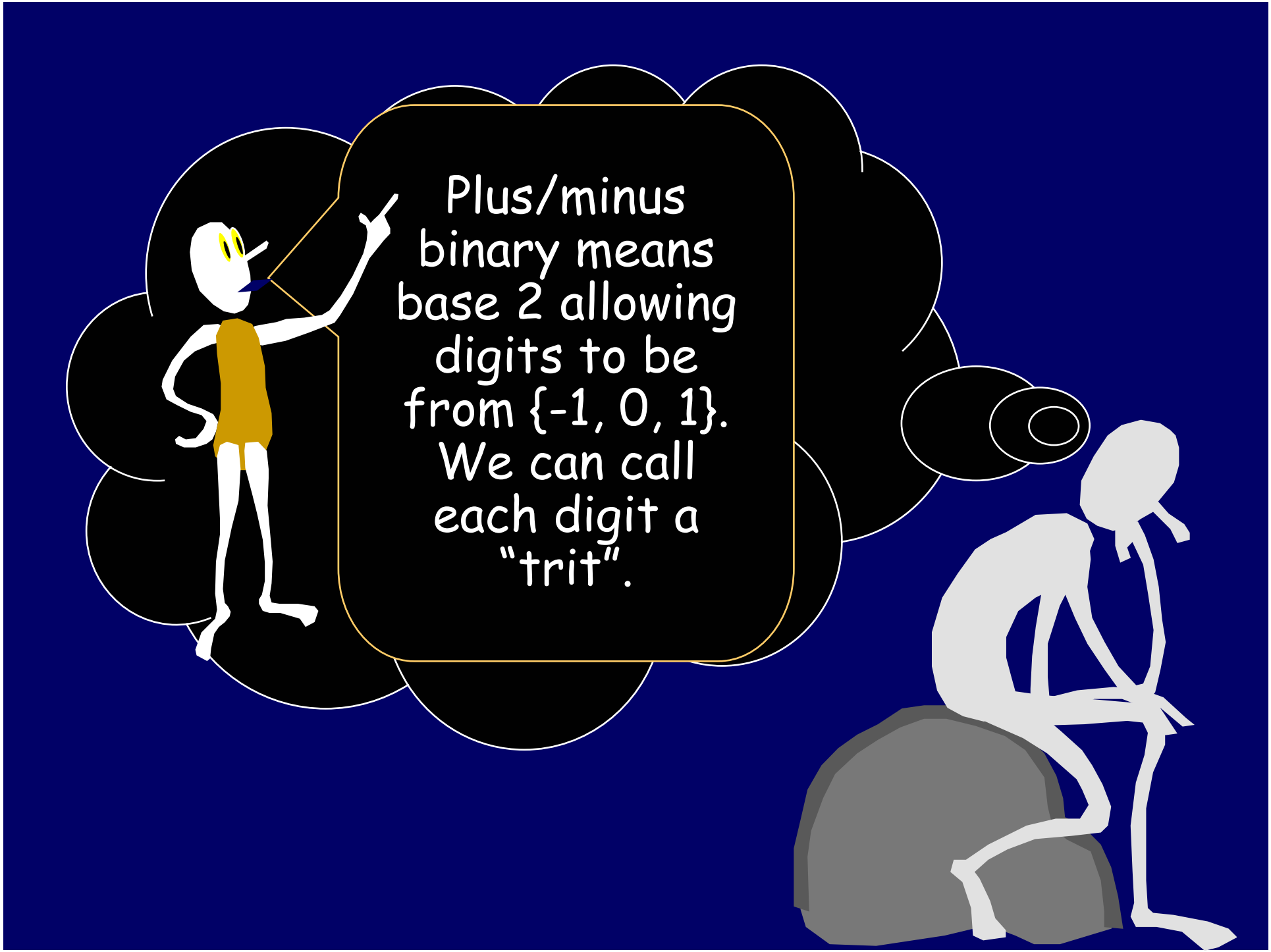
How long to add two n bit numbers?

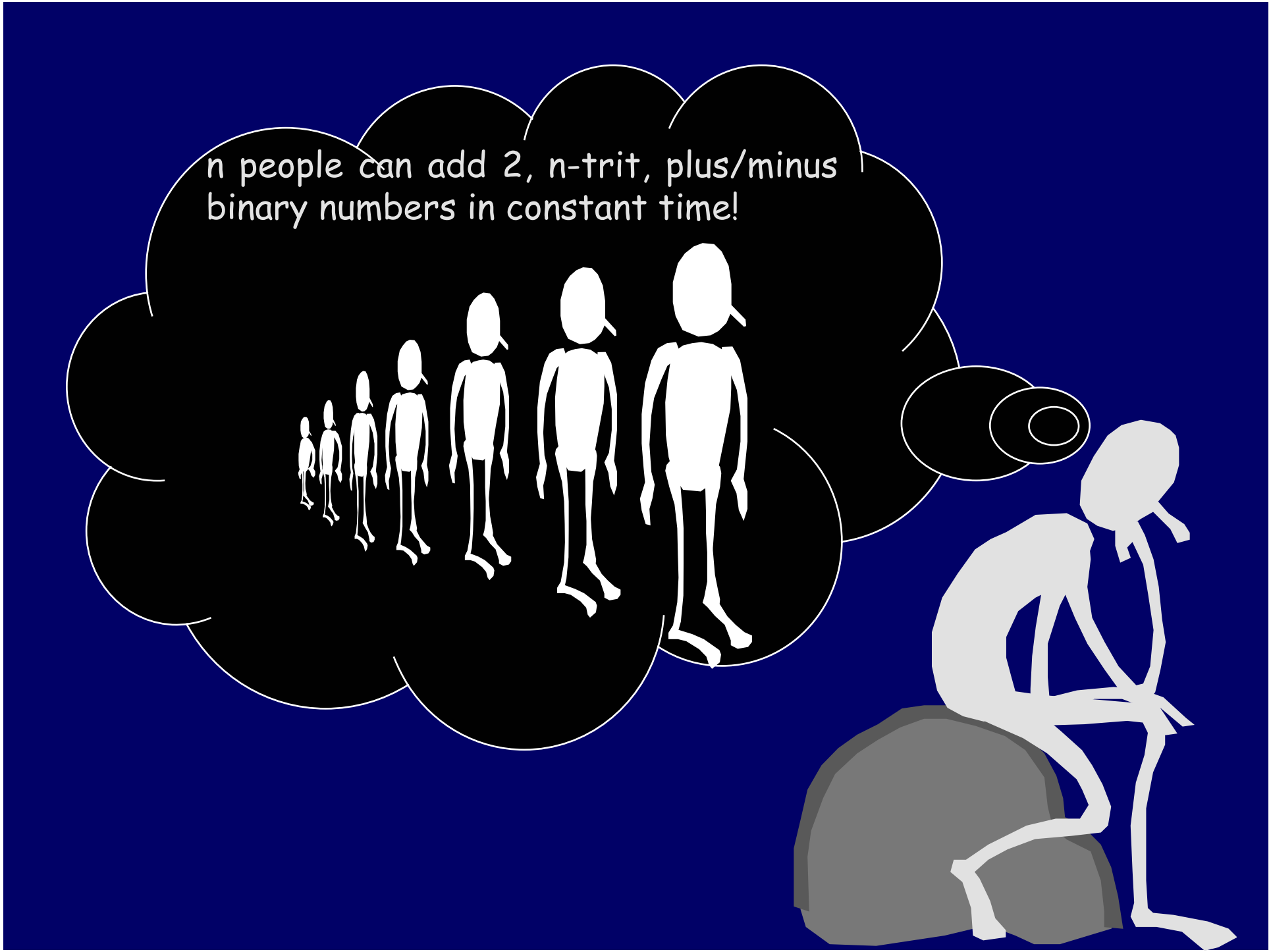
Propagation time through the circuit will be $\theta(n)$

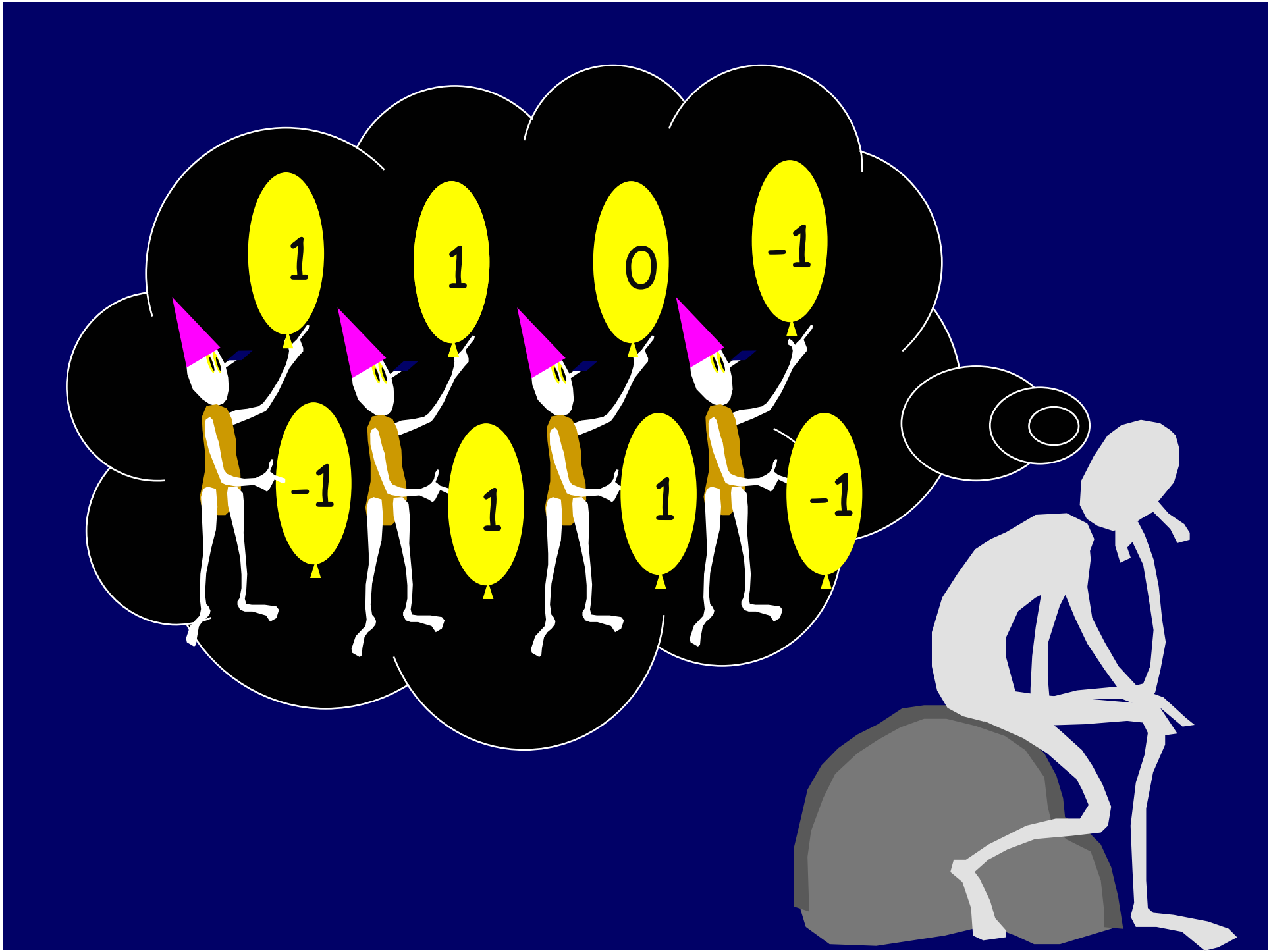

Circuits compute things in parallel. We can think of the propagation delay as PARALLEL TIME.

Is it possible to add two n bit numbers in less than linear parallel-time?


I suppose the EB addition algorithm could be helpful somehow.


Plus/minus binary means base 2 allowing digits to be from {-1, 0, 1}. We can call each digit a "trit".


n people can add 2, n-trit, plus/minus binary numbers in constant time!


1
1
0
-1
-1
1
1
-1

Can we still do addition quickly in the standard representation?

Yes, but first a neat idea…

Instead of adding two numbers to make one number, let's think about adding 3 numbers to make 2 numbers.

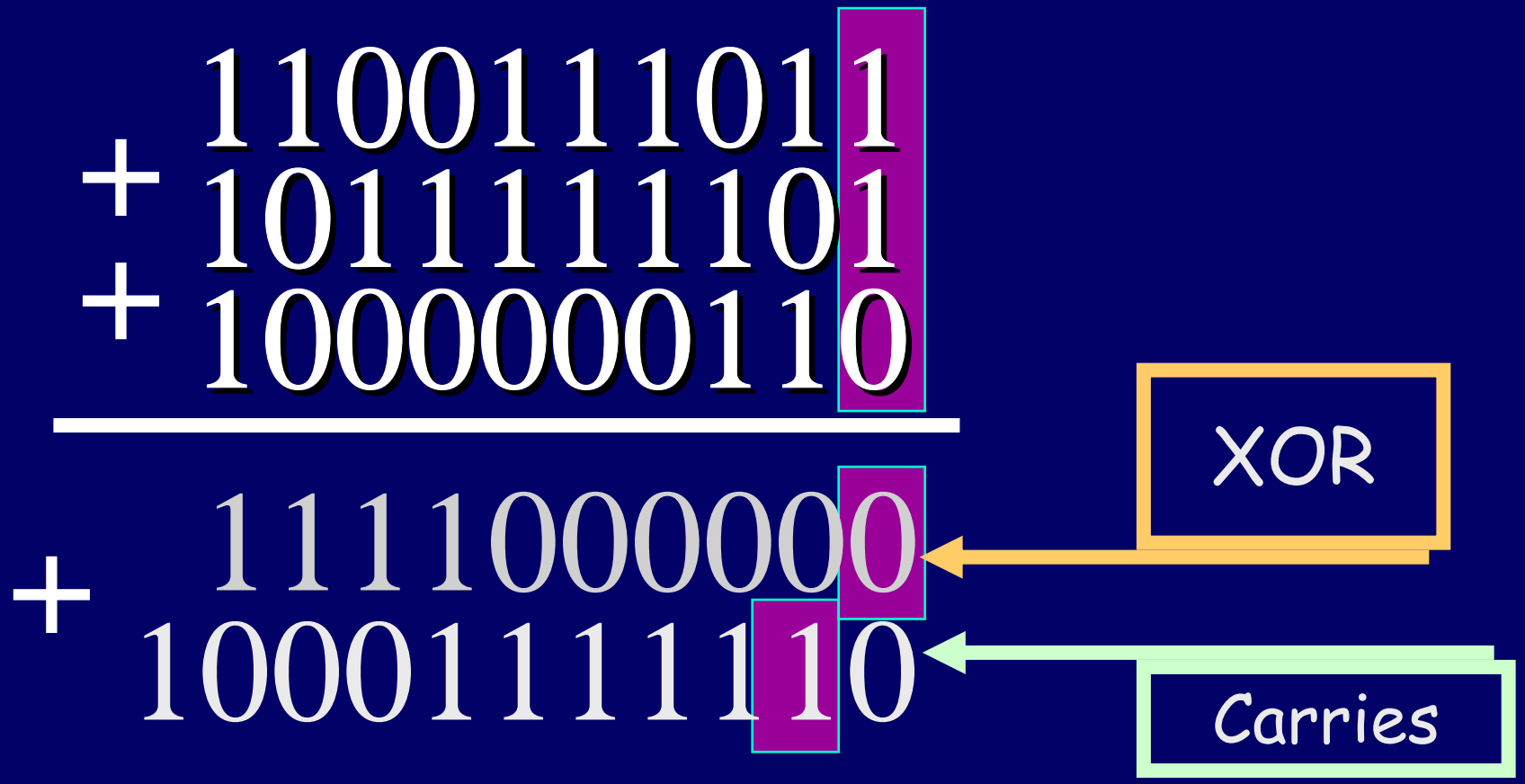
Carry-Save Addition

The sum of three numbers can be converted into the sum of 2 numbers in constant parallel time!

$$
\begin{array}{r}
1100111011 \\
+\ 1011111101 \\
+\ 1000000110 \\
\hline
\end{array}
$$

Carry-Save Addition

The sum of three numbers can be converted into the sum of 2 numbers in constant parallel time!

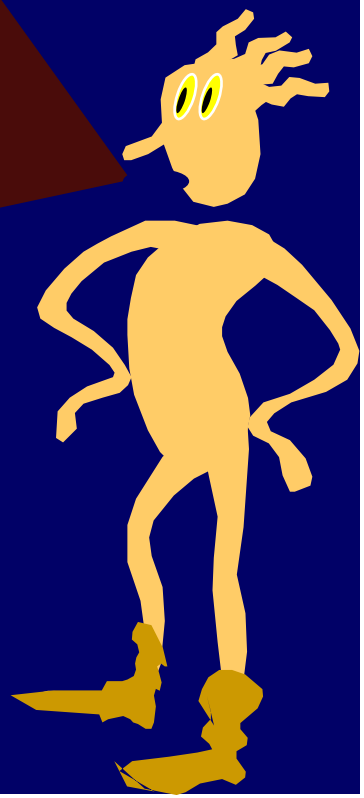


Cool!

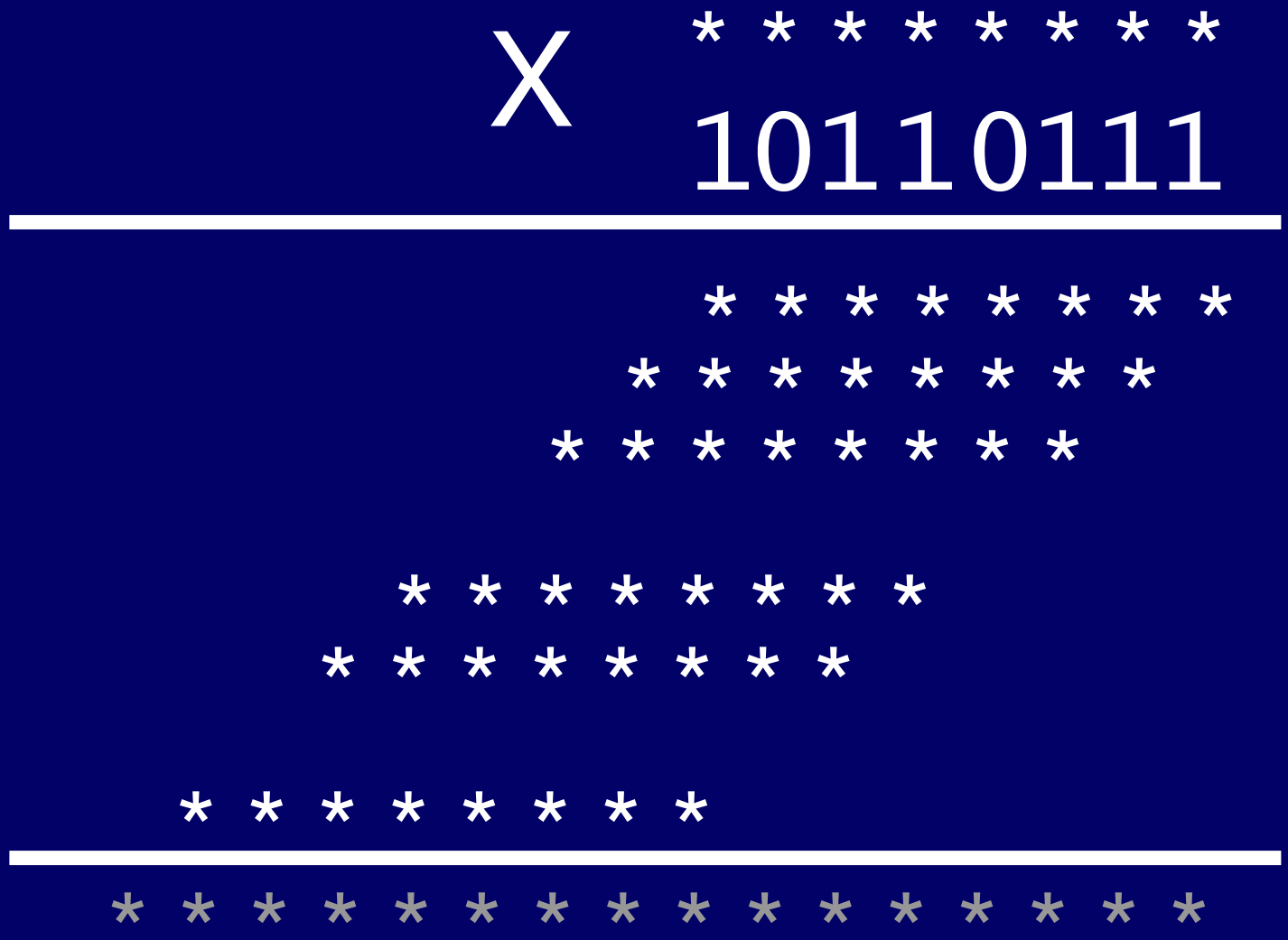
So if we if represent x as a+b, and y as c+d, then can add x,y using two of these (this is basically the same as that plus/minus binary thing).

(a+b+c)+d=(e+f)+d=g+h

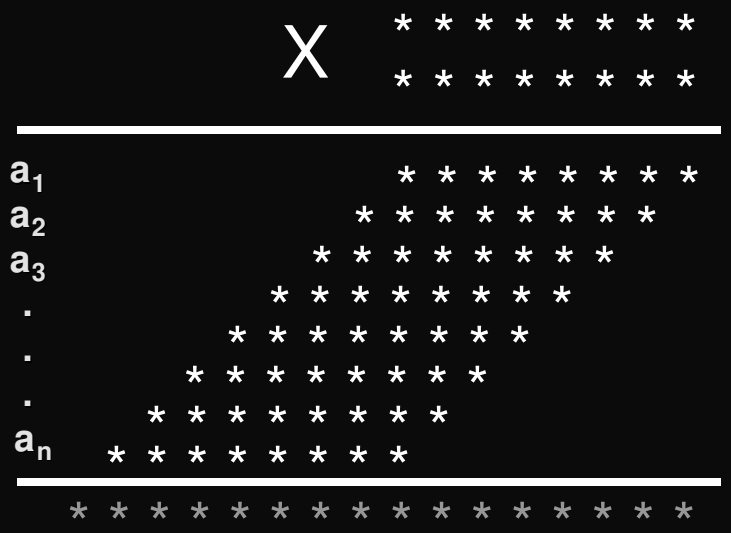
Even in standard representation, this is *really* useful for multiplication.



Grade School Multiplication



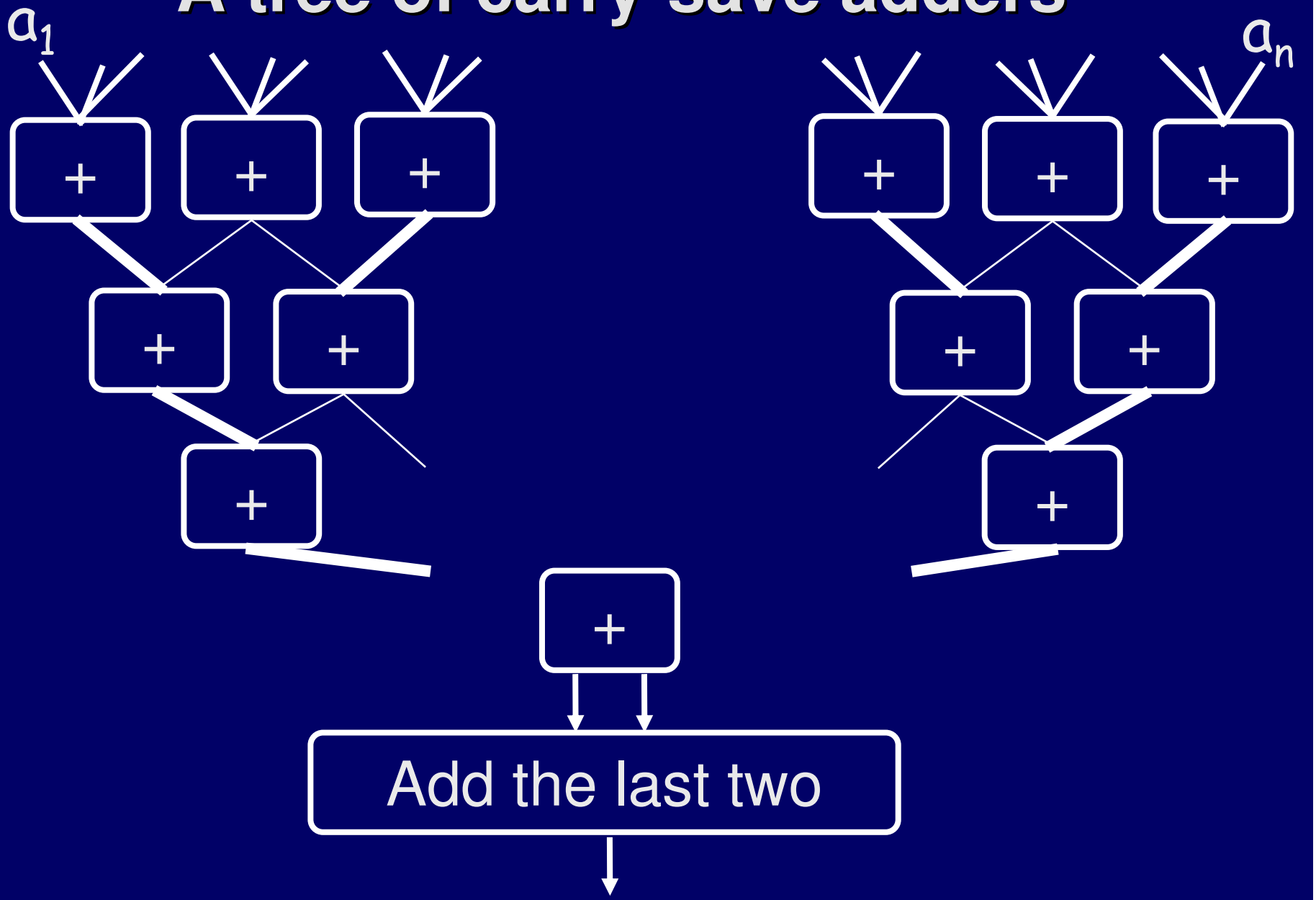
Grade School Multiplication



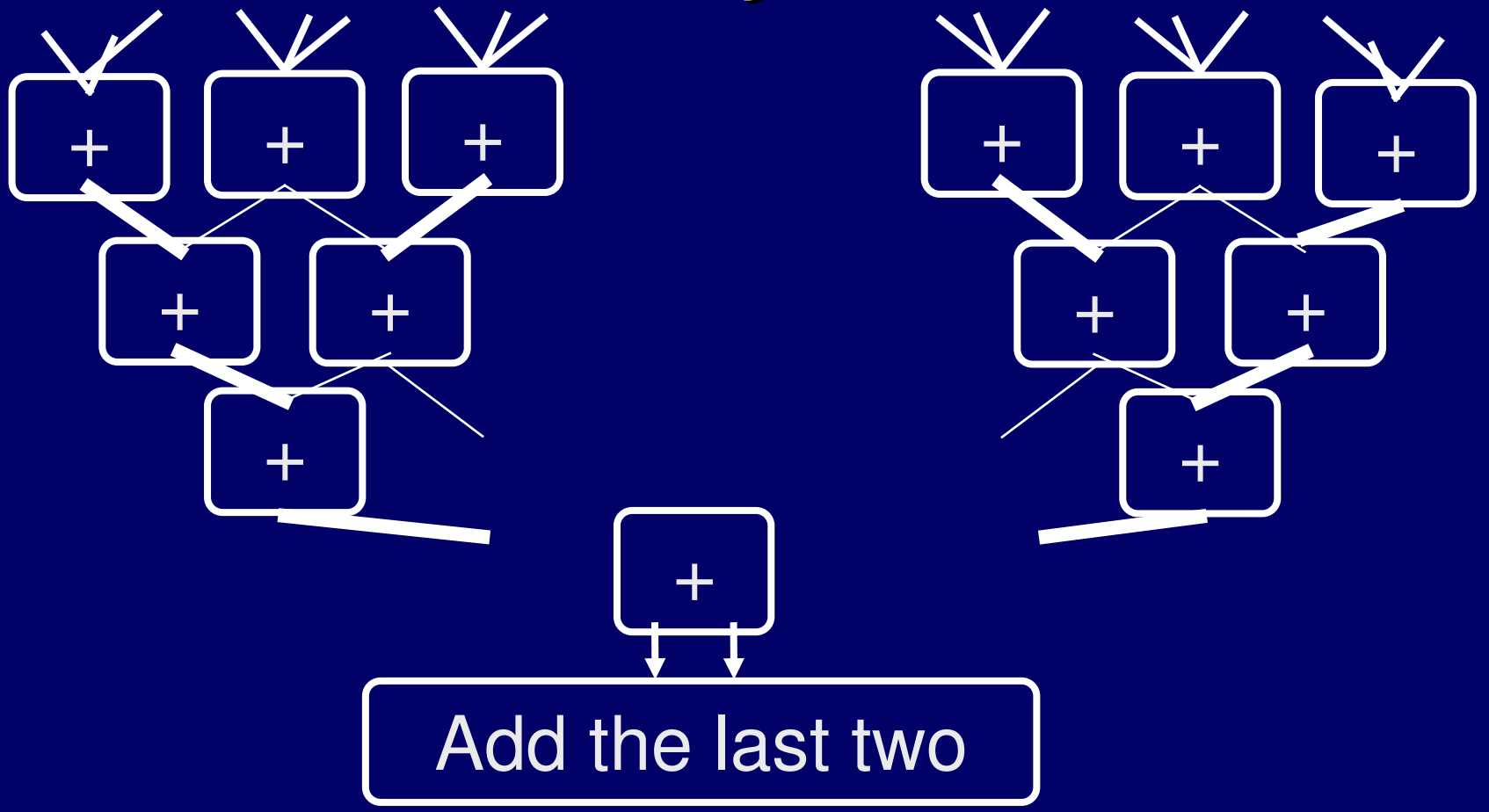
We need to add n 2n-bit numbers:

$a_1, a_2, a_3, \ldots, a_n$

A tree of carry-save adders



A tree of carry-save adders

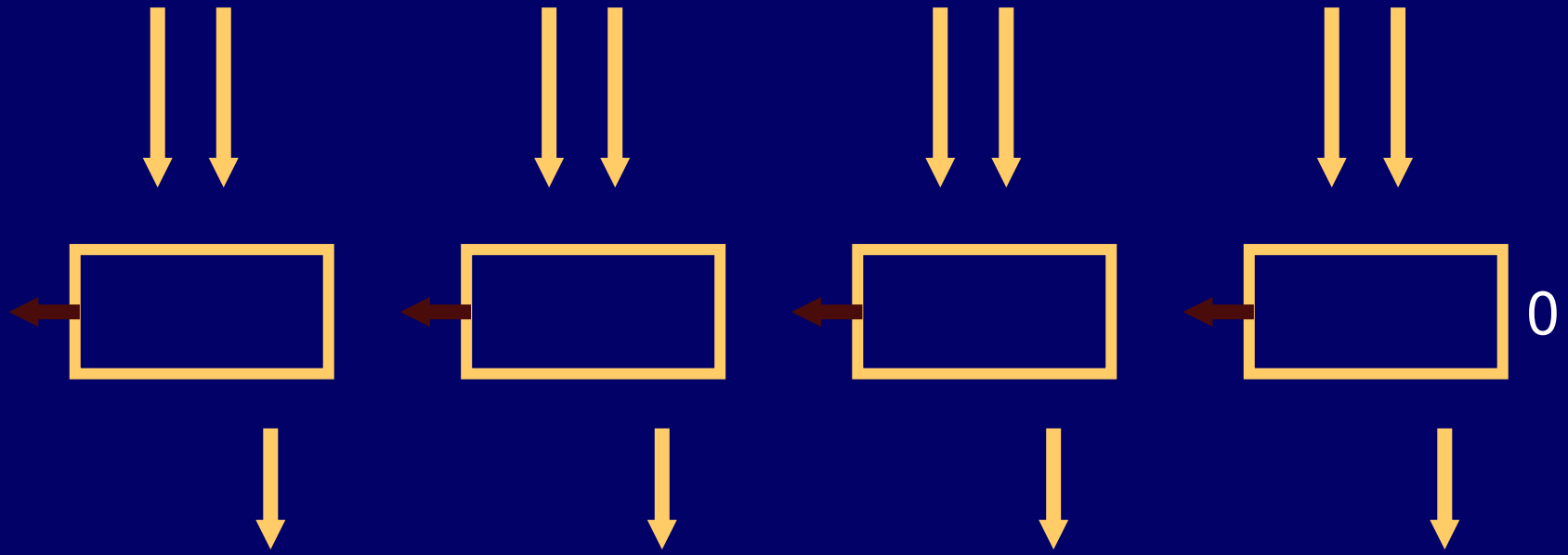


$T(n) \approx \log_{3/2}(n) + [\text{last step}]$

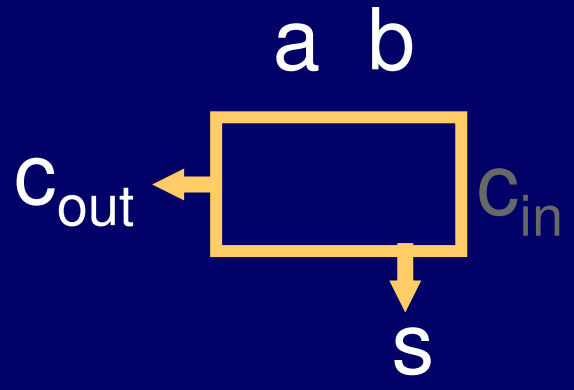
So let's go back to the problem of adding two numbers.

In particular, if we can add two numbers in O(log n) parallel time, then we can multiply in O(log n) parallel time too!

If we knew the carries it would be very easy to do fast parallel addition

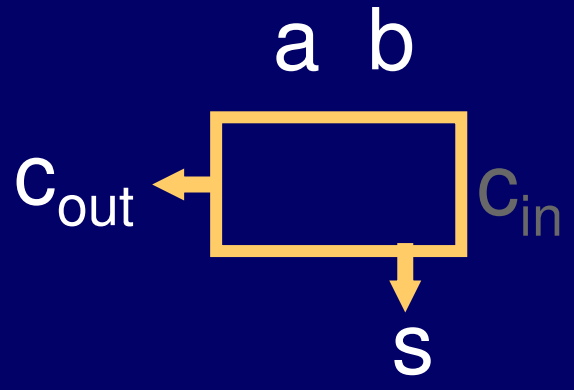


What do we know about the carry-out before we know the carry-in?



a	b	C_{out}
0	0	0
0	1	C_{in}
1	0	C_{in}
1	1	1

What do we know about the carry-out before we know the carry-in?



a	b	C_{out}
0	0	0
0	1	←
1	0	←
1	1	1

Hey, this is just a function of a and b. We can do this in parallel.

a	b	C_{out}
0	0	0
0	1	←
1	0	←
1	1	1

Idea #1: do this calculation first.

$$
\begin{array}{r}
10_{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}1_{\leftarrow}0 \\
1011111101 \\
+\quad 1000000110 \\
\hline
\end{array}
$$

This takes just one step!

Idea #1: do this calculation first.

$$
\begin{array}{r}
10_{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}1_{\leftarrow}0 \\
1011111101 \\
+\quad 1000000110 \\
\hline
\end{array}
$$

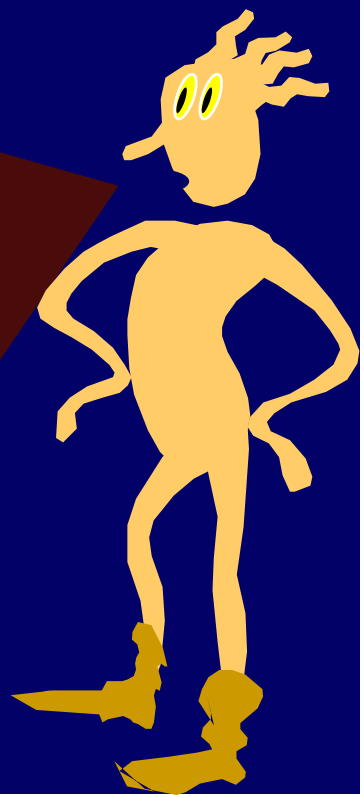
Also, once we have the carries, it only takes one step more:

$$s_i = (a_i \text{ XOR } b_i) \text{ XOR } c_i$$

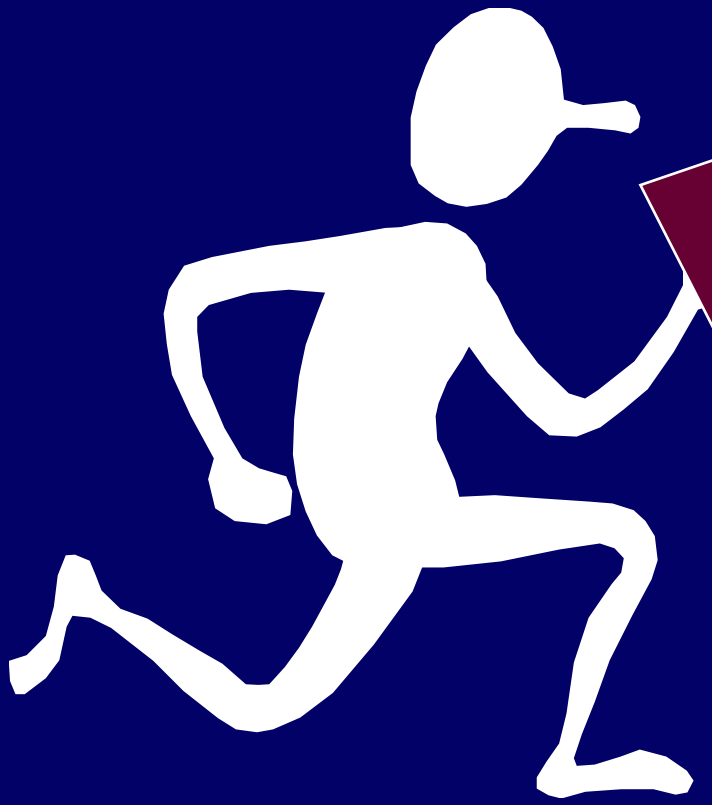
$10_{\leftarrow}\ _{\leftarrow}\ _{\leftarrow}{}_{\leftarrow}{}_{\leftarrow}1_{\leftarrow}0$

So, everything boils down to: can we find a fast parallel way to convert each position to its final 0/1 value?

Called the "parallel prefix problem"



$10_{\leftarrow} \leftarrow \leftarrow \leftarrow \leftarrow 1_{\leftarrow} 0$



So, we need to do this quickly….

Idea #2:

Can think of $10\leftarrow\ \leftarrow\ \leftarrow\leftarrow\leftarrow 1\leftarrow 0$ as all partial results in:

(1 ☺ (0 ☺(← ☺(← ☺ (← ☺ (← ☺ (← ☺ (1 ☺ (← ☺ 0))))))))

for the operator ☺:

←☺ x = x

1 ☺ x = 1

0 ☺ x = 0

☺	0	1	←
0	0	0	0
1	1	1	1
←	0	1	←

Idea #2 (cont):

And, the ☺ operator is associative.

$$10 \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow 1 \leftarrow 0$$

$$(\leftarrow \☺ (\leftarrow \☺ (\leftarrow \☺ (1 \☺ (\leftarrow \☺ 0)))))$$

$$=$$

$$(\leftarrow \☺ \leftarrow) \☺ (\leftarrow \☺ 1) \☺ (\leftarrow \☺ 0)$$

$$=$$

$$\leftarrow \☺ 1 \☺ 0 \;=\; 1$$

Just using the fact that we have an *Associative*, Binary Operator

Binary Operator: an operation that takes two objects and returns a third.

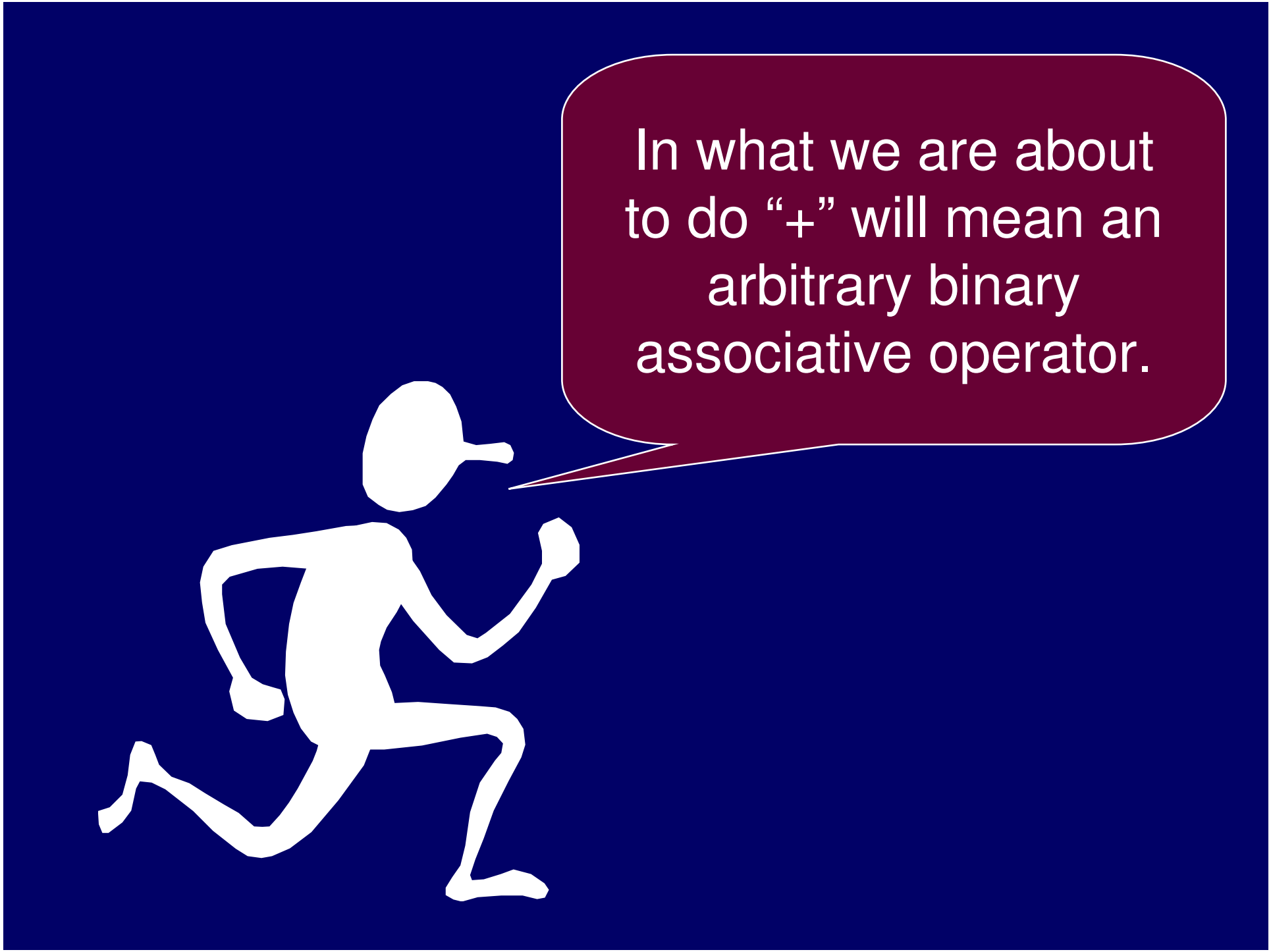
- A ♠ B = C

Associative:

- (A ♠ B) ♠ C = A ♠ (B ♠ C)

Examples

- Addition on the integers
- Min(a,b)
- Max(a,b)
- Left(a,b) = a
- Right(a,b) = b
- Boolean AND
- Boolean OR
- ☺


In what we are about to do “+” will mean an arbitrary binary associative operator.

Prefix Sum Problem

Input: $X_{n-1}, X_{n-2},\ldots,X_1, X_0$

Output: $Y_{n-1}, Y_{n-2},\ldots,Y_1, Y_0$

where

$Y_0 = X_0$

$Y_1 = X_0 + X_1$

$Y_2 = X_0 + X_1 + X_2$

$Y_3 = X_0 + X_1 + X_2 + X_3$

$\vdots$

$Y_{n-1} = X_0 + X_1 + X_2 + X_3 + \ldots + X_{n-1}$

Prefix Sum example

Input: 6, 9, 2, 3, 4, 7

Output: 31, 25, 16, 14, 11, 7

where

$Y_0 = X_0$

$Y_1 = X_0 + X_1$

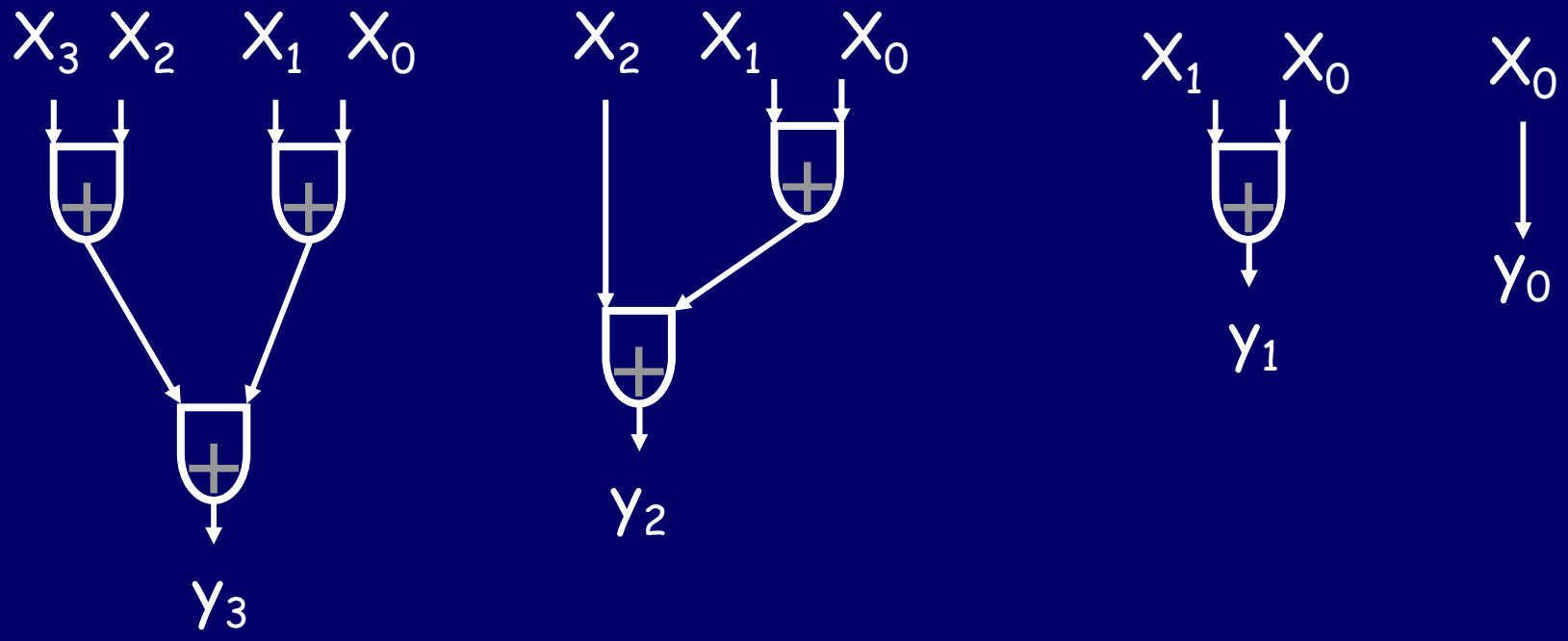
$Y_2 = X_0 + X_1 + X_2$

$Y_3 = X_0 + X_1 + X_2 + X_3$

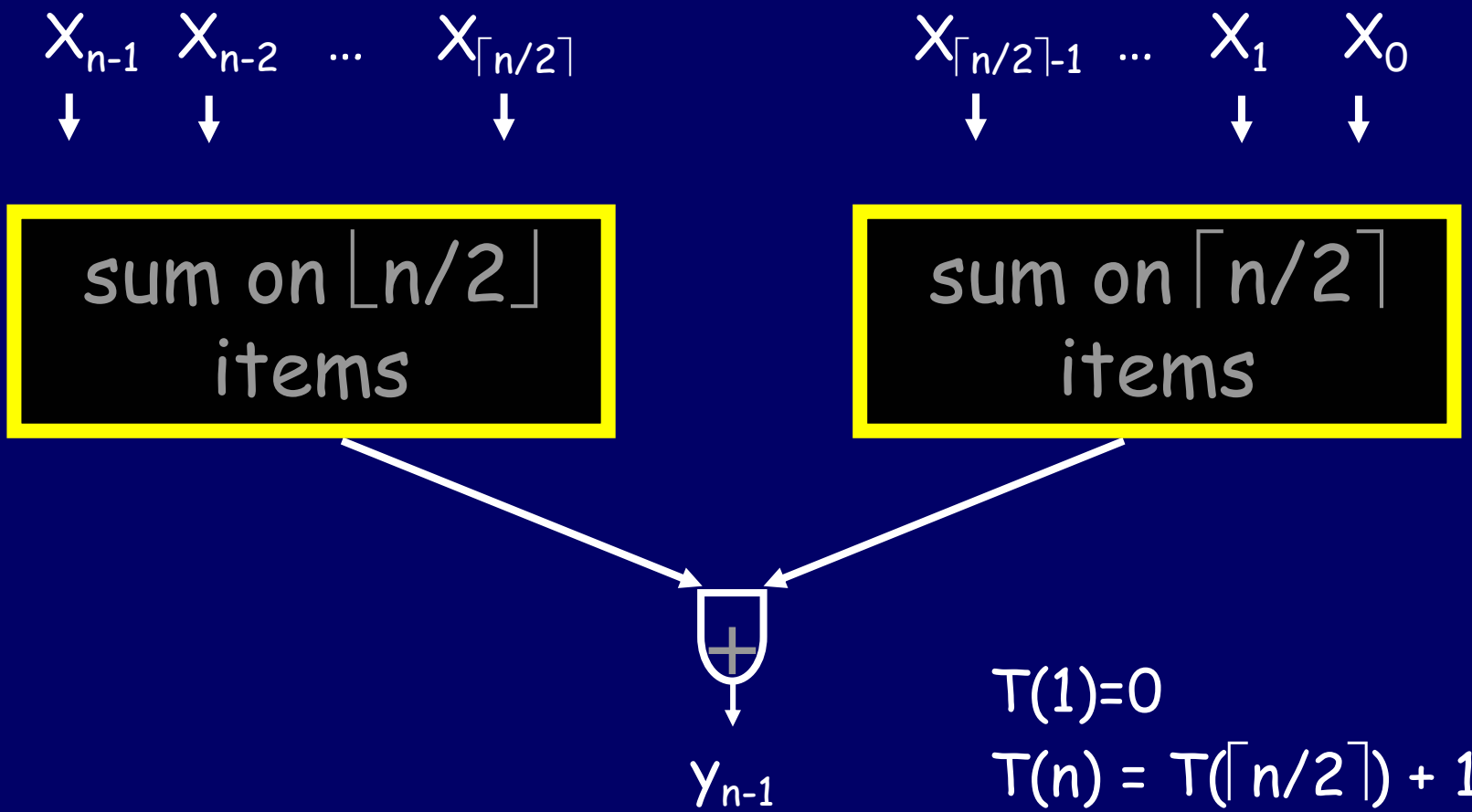
$\vdots$

$Y_{n-1} = X_0 + X_1 + X_2 + X_3 + \ldots + X_{n-1}$

Example circuitry (n = 4)



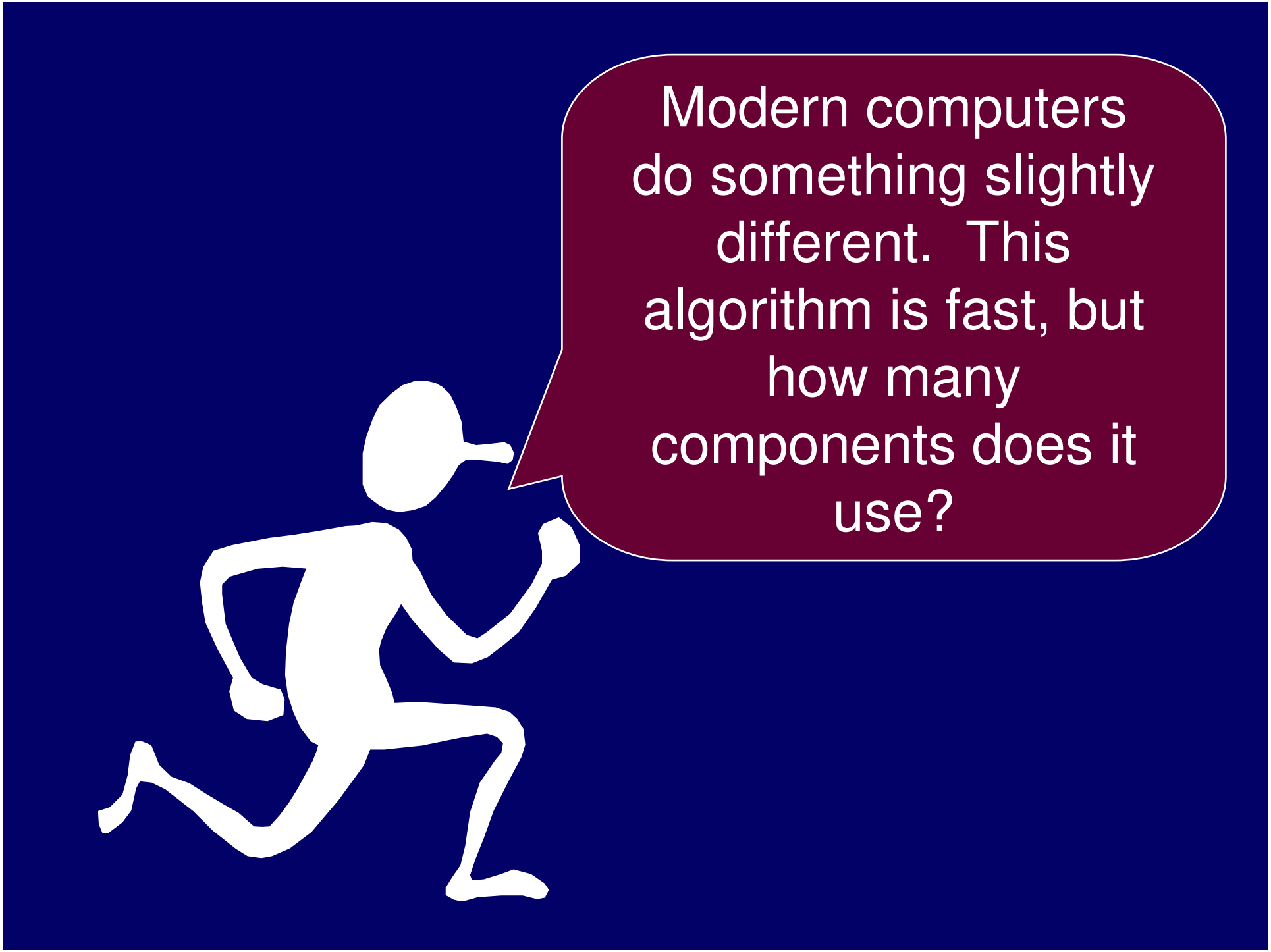
Divide, conquer, and glue for computing y_{n-1}



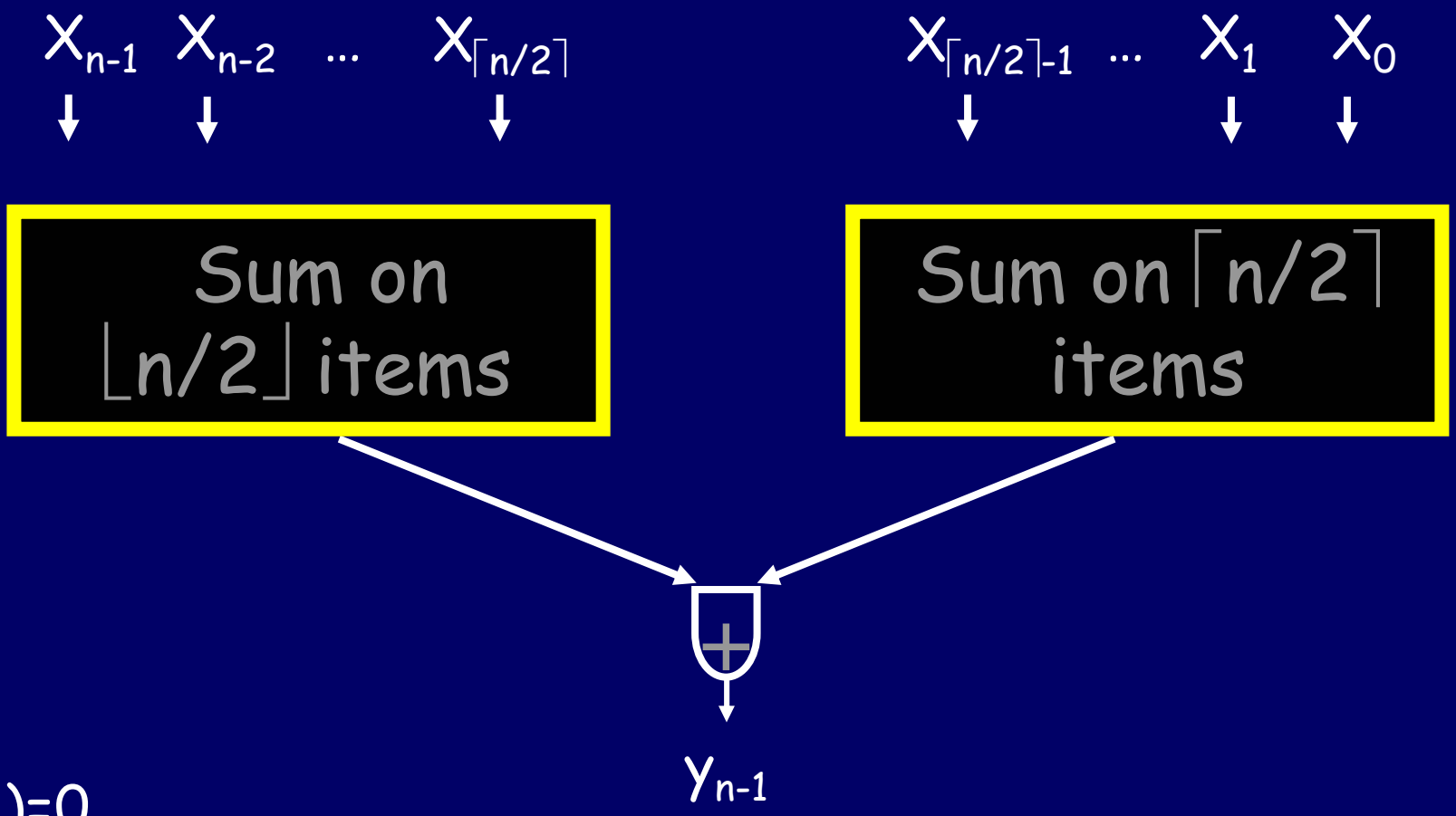
$T(1)=0$

$T(n) = T(\lceil n/2 \rceil) + 1$

$T(n) = \lceil \log_2 n \rceil$


Modern computers do something slightly different. This algorithm is fast, but how many components does it use?

Size of Circuit (number of gates)

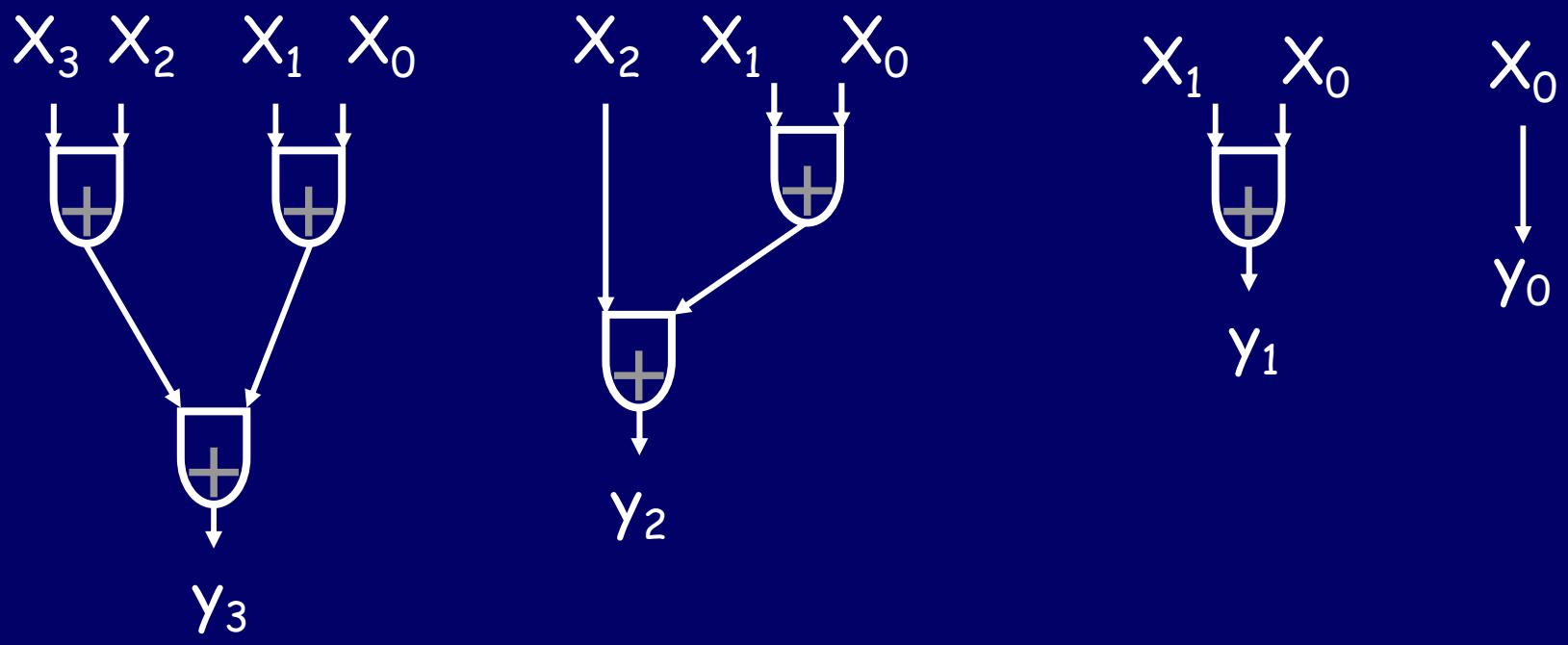


$S(1)=0$

$S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + 1$

$S(n) = n-1$

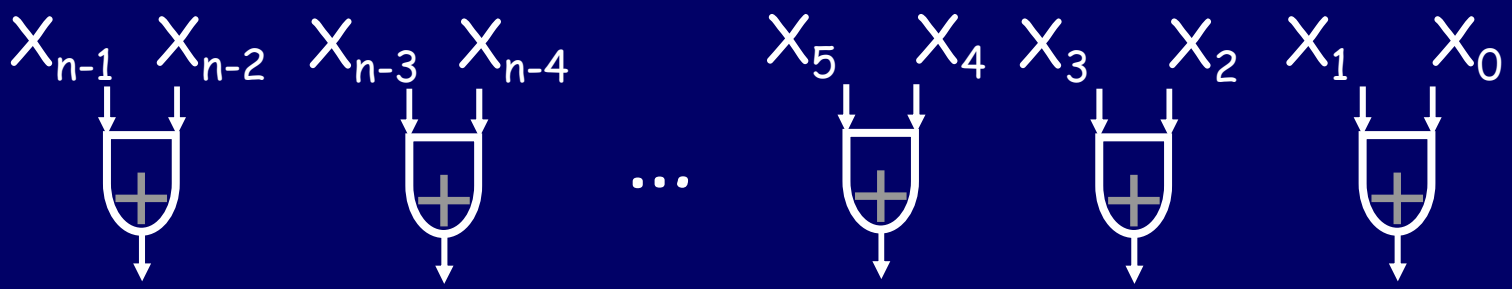
Sum of Sizes



$S(n) = 0 + 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2$

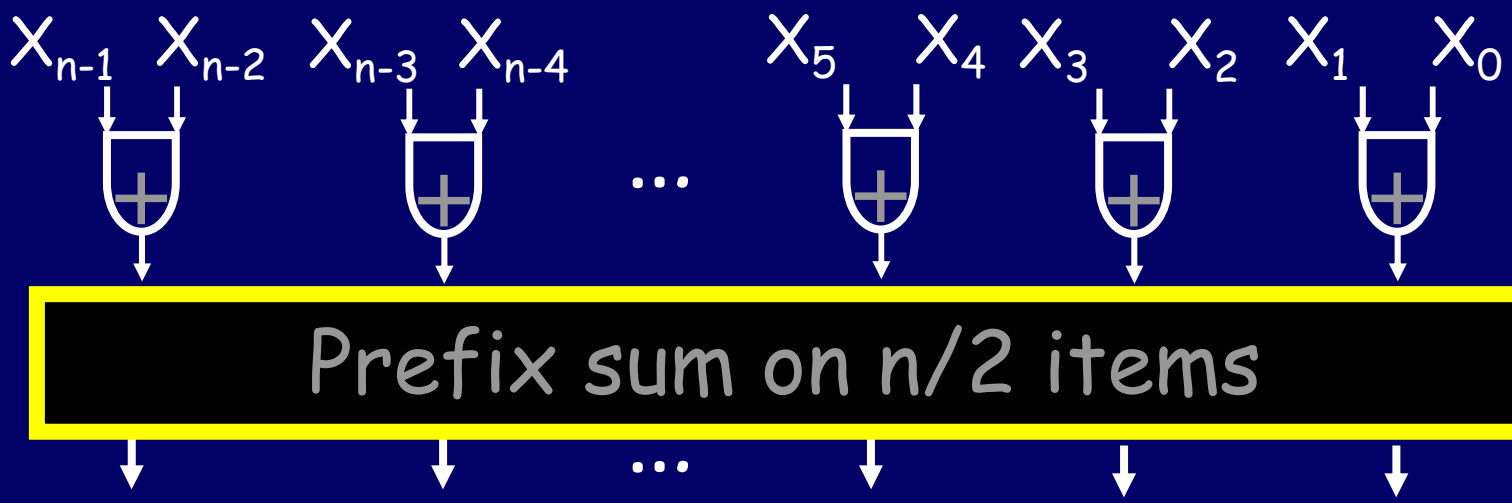
Recursive Algorithm
n items (n = power of 2)

If $n = 1$, $Y_0 = X_0$;



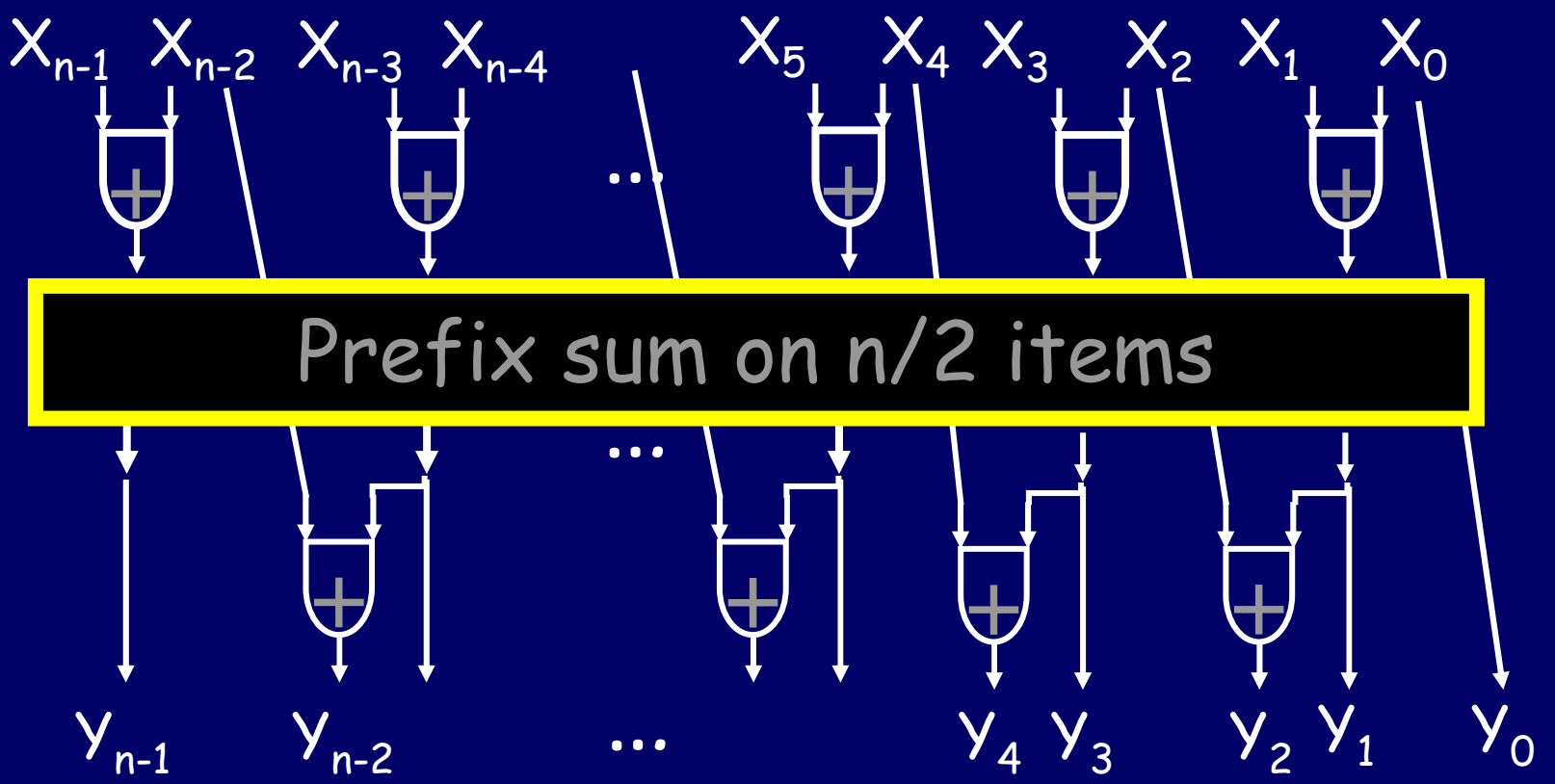
Recursive Algorithm
n items (n = power of 2)

If $n = 1$, $Y_0 = X_0$;



Recursive Algorithm
n items (n = power of 2)

If $n = 1$, $Y_0 = X_0$;



Parallel time complexity

If n = 1, $Y_0 = X_0$;

$T(1)=0;\ T(2) = 1;\ T(n) = T(n/2) + 2$

$T(n) = 2 \log_2(n) - 1$

Size

If $n = 1$, $Y_0 = X_0$;

$S(1)=0$; $S(n) = S(n/2) + n - 1$

$S(n) = 2n - \log_2 n - 2$

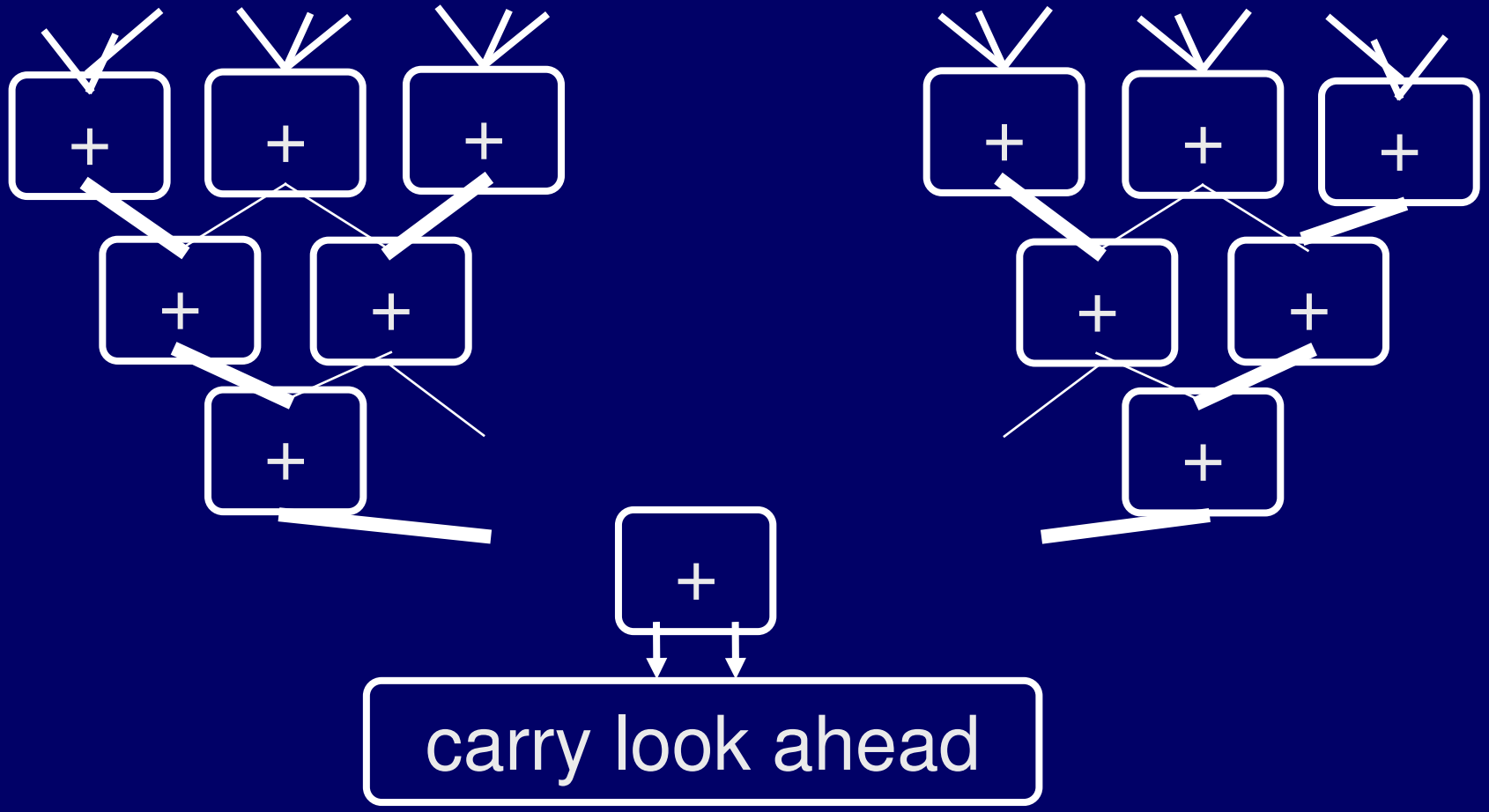
Putting it all together: Carry Look-Ahead Addition

To add two n-bit numbers: a and b

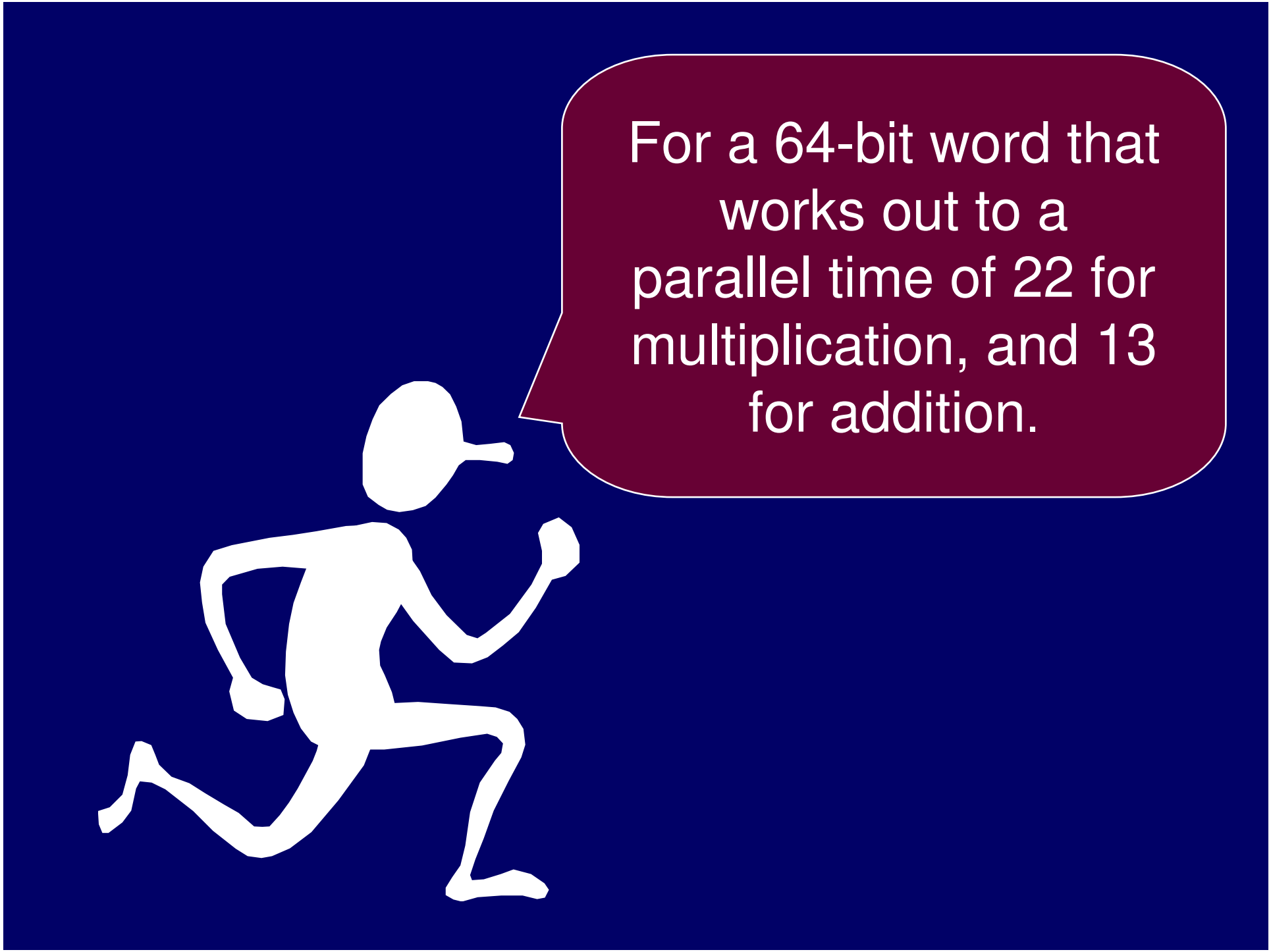
- 1 step to compute x values ($\leftarrow 01$)
- 2 $\log_2 n$ - 1 steps to compute carries c
- 1 step to compute c XOR (a XOR b)

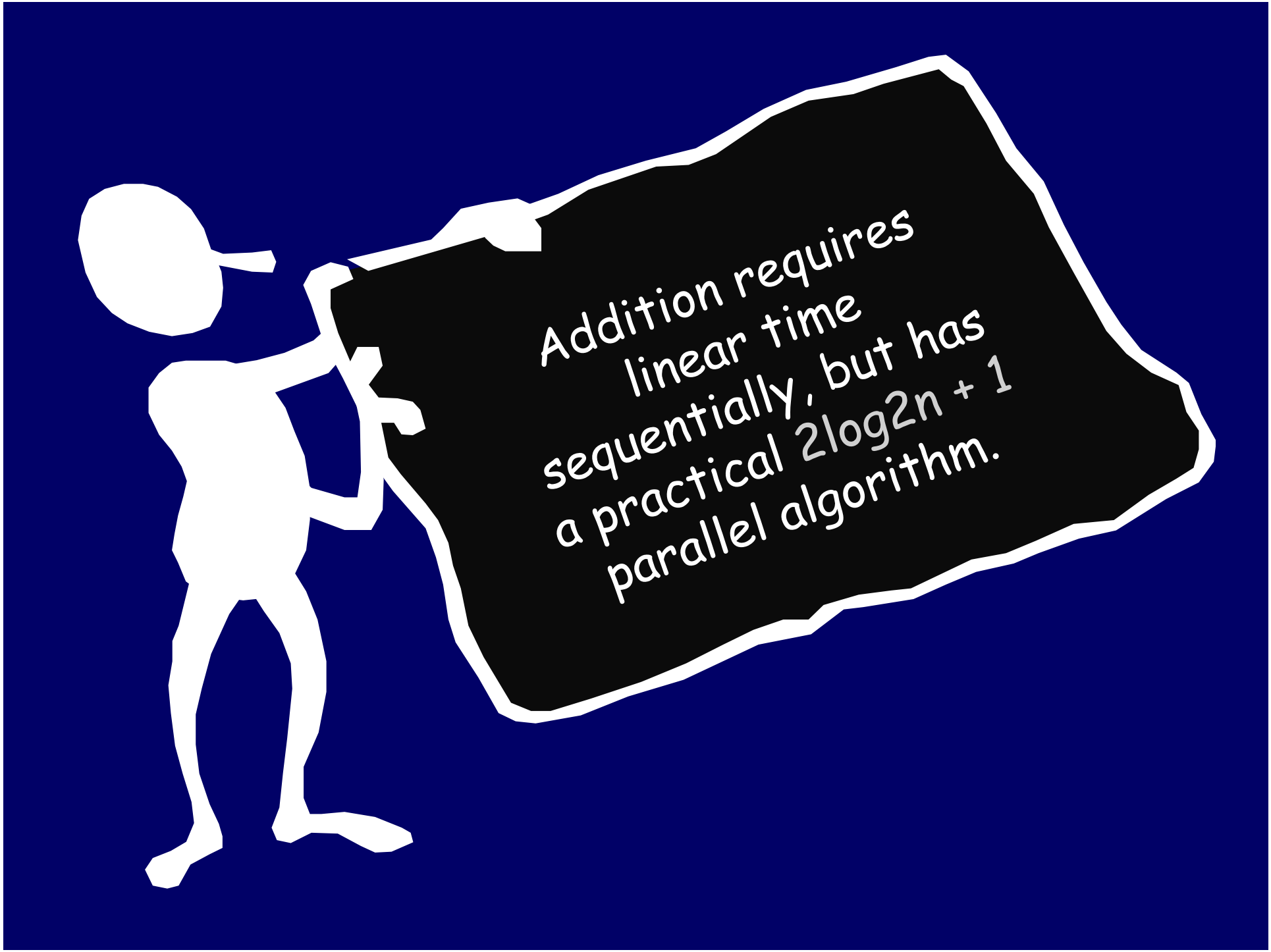
2 $\log_2 n$ + 1 steps total

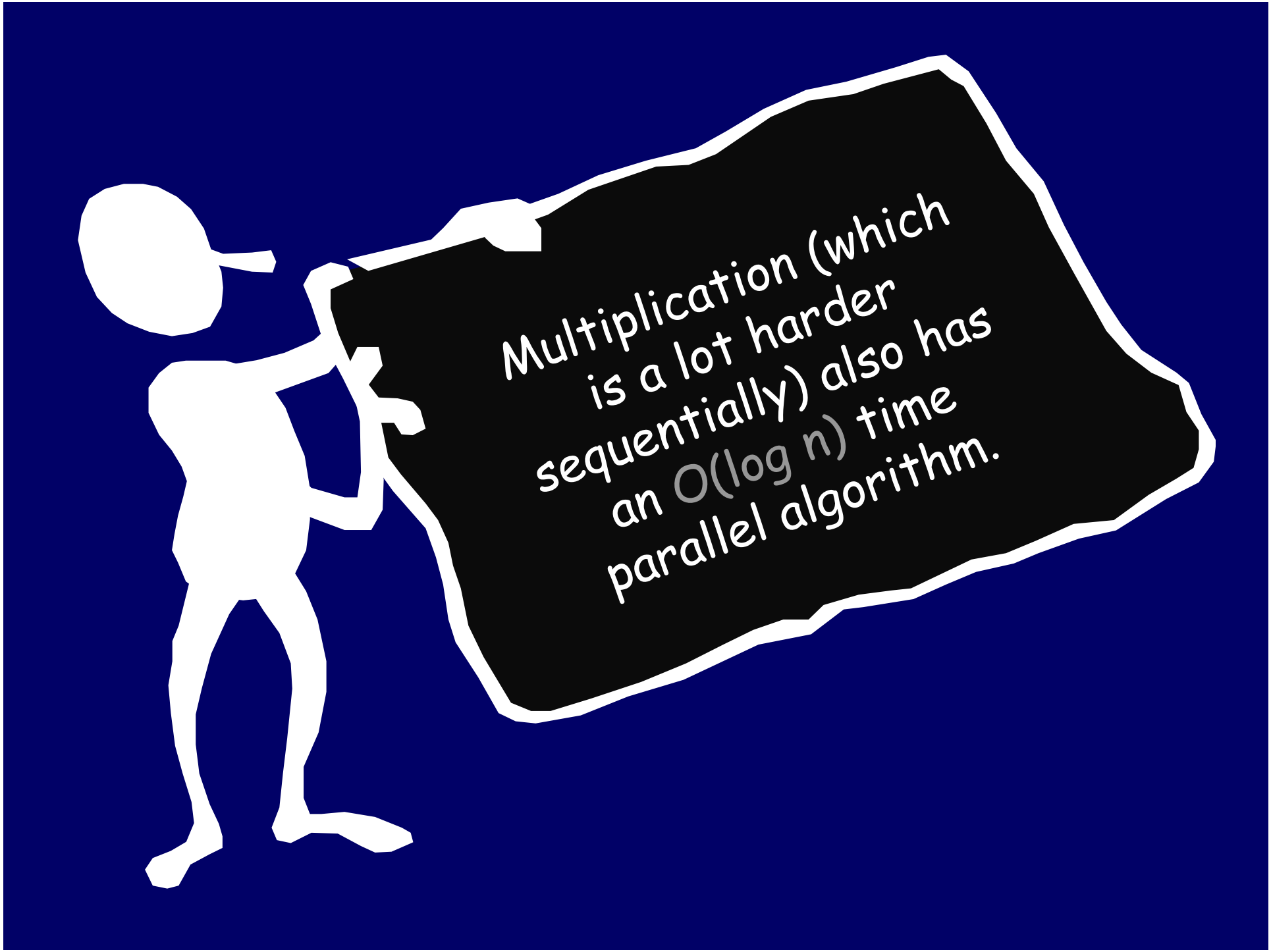
Putting it all together: multiplication



$$T(n) \approx \log_{3/2}(n) + 2\log_2 2n + 1$$

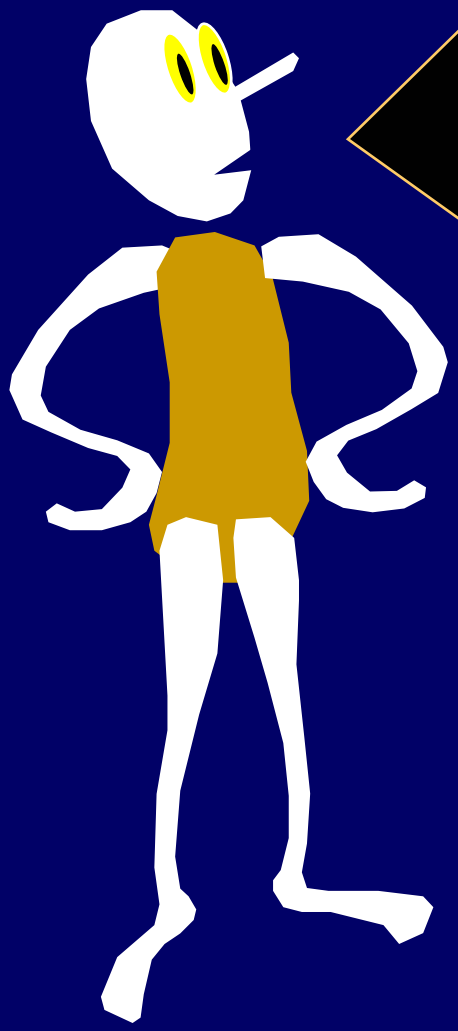

For a 64-bit word that works out to a parallel time of 22 for multiplication, and 13 for addition.


Addition requires
linear time
sequentially, but has
a practical 2log2n + 1
parallel algorithm.


Multiplication (which is a lot harder sequentially) also has an O(log n) time parallel algorithm.

And this is how addition works on commercial chips.....

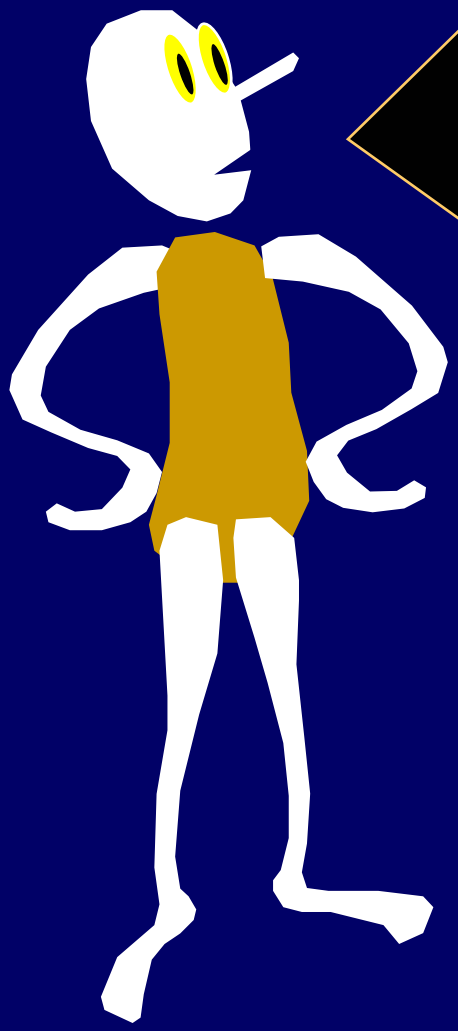
Processor	n	$2\log_2 n + 1$
80186	16	9
Pentium	32	11
Alpha	64	13



In order to handle integer addition/subtraction we use 2's compliment representation, e.g.,

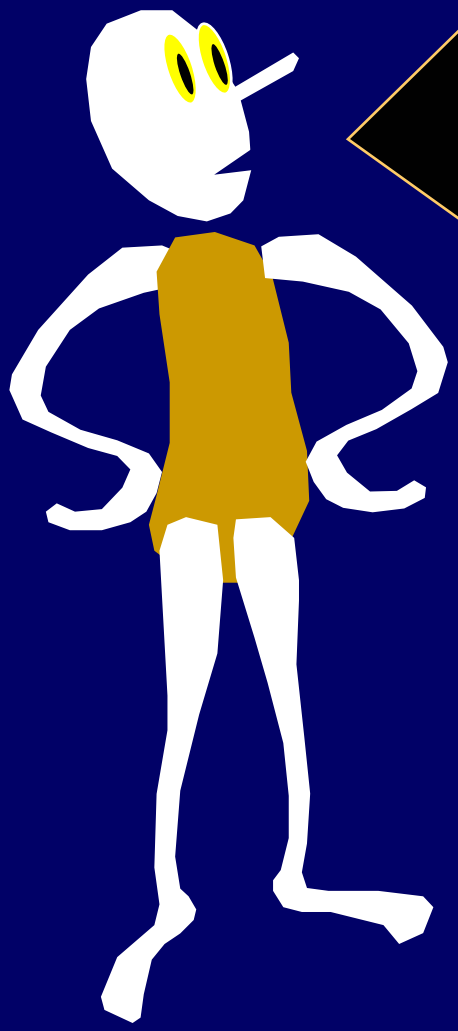
-44=

-64	32	16	8	4	2	1
1	0	1	0	1	0	0



Addition of two numbers works the same way (assuming no overflow).

-64	32	16	8	4	2	1
1	0	1	0	1	0	0

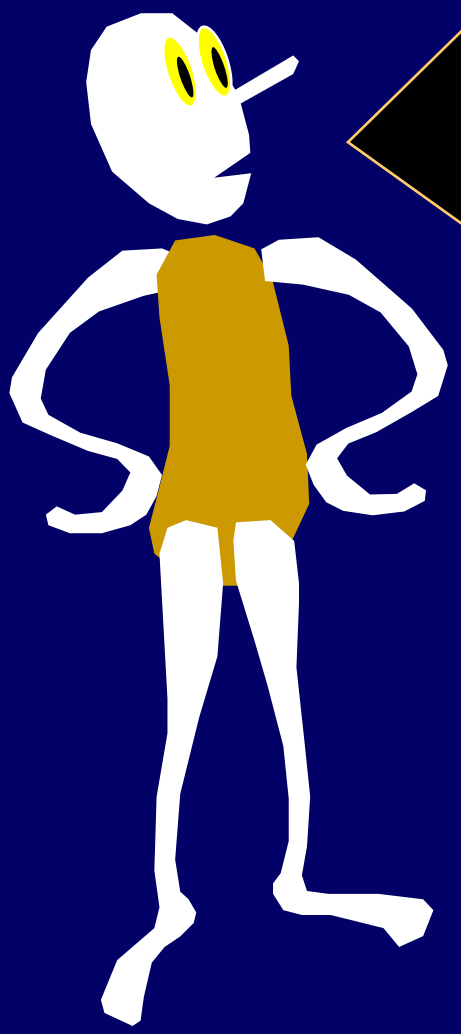


To negate a number, flip each of its bits and add 1.

-64	32	16	8	4	2	1
1	0	1	0	1	0	0

-64	32	16	8	4	2	1
0	1	0	1	0	1	1

-64	32	16	8	4	2	1
0	1	0	1	1	0	0

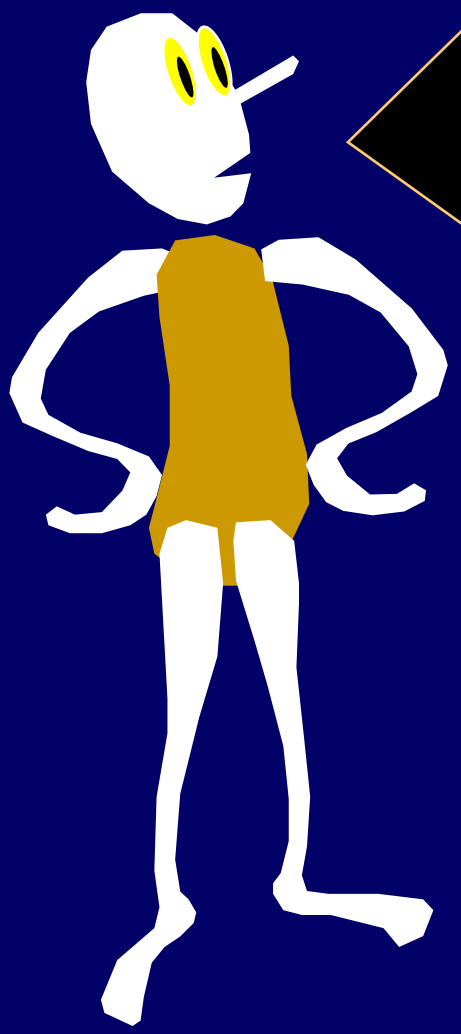


To negate a number, flip each of its bits and add 1.

-64	32	16	8	4	2	1
1	1	1	1	1	1	1

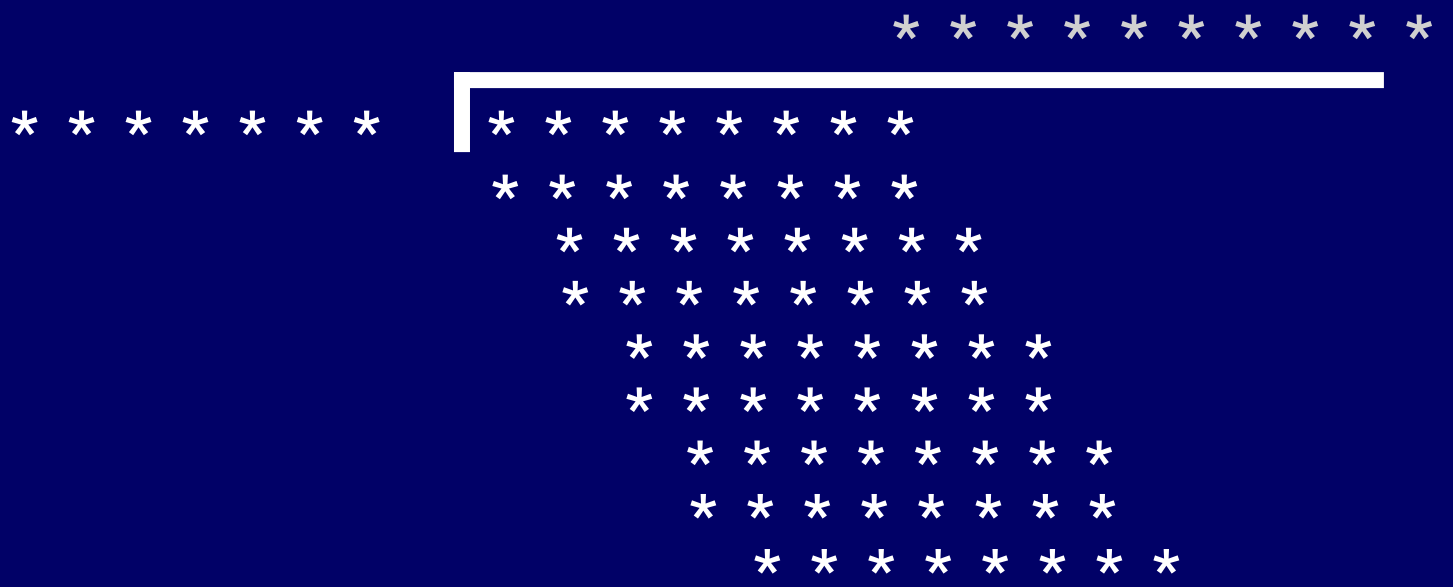
$x + \text{flip}(x) = -1.$

So, $-x = \text{flip}(x)+1.$



Most computers use two's compliment representation to perform integer addition and subtraction.

Grade School Division



n bits of precision:
n subtractions costing $2\log_2 n + 1$ each
$\theta(n \log n)$

Let’s see if we can reduce to O(n) by being clever about it.

Idea: internally, allow ourselves to "go negative" using trits so we can do constant-time subtraction.

Then convert back at the end.

(technically, called "extended binary")

SRT division algorithm

```
                1  1 -1  1  0   r -1 -1  1
1  0  1  1 | 1  1  1  0  1  1  0  1
             -1 0 -1 -1
             ----------
                1  0 -1  1
               -1 0 -1 -1
               ----------
                    -2 0
               = -1  0  0 1
                  1  0 1  1
                  ---------
                      1  2
                = 1  0  0  0
                 -1  0 -1 -1
                 ----------
                    0 -1 -1  1
```

```
      21 r 6          22 r -5
11 | 237          11 | 237
```

Rule: Each bit of quotient is determined by comparing first bit of divisor with first bit of dividend. Easy!

Time for n bits of precision in result:

$$\approx \underbrace{3n}_{\text{1 addition per bit}} + \underbrace{2\log_2(n) + 1}_{\text{Convert to standard representation by subtracting negative bits from positive.}}$$

Intel Pentium division error

- The Pentium uses essentially the same algorithm, but computes more than one bit of the result in each step. Several leading bits of the divisor and quotient are examined at each step, and the difference is looked up in a table.

- The table had several bad entries.

- Ultimately Intel offered to replace any defective chip, estimating their loss at $475 million.

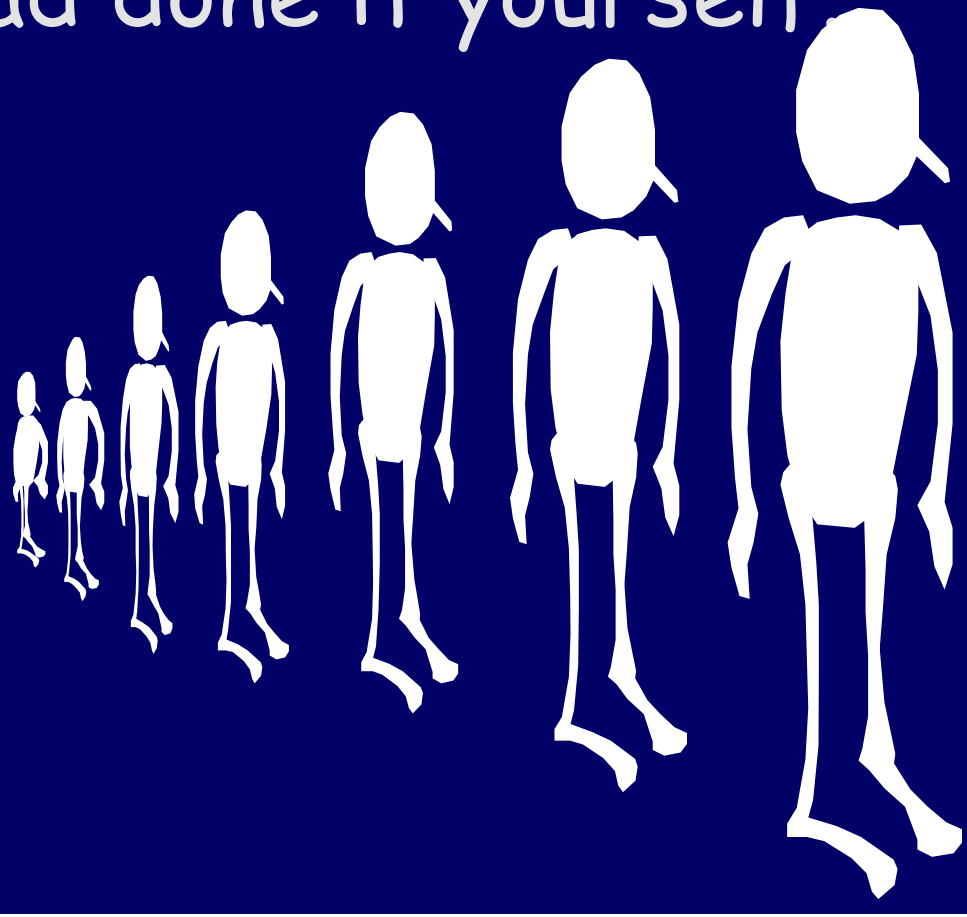
If millions of processors, how much of a speed-up might I get over a single processor?

Brent's Law

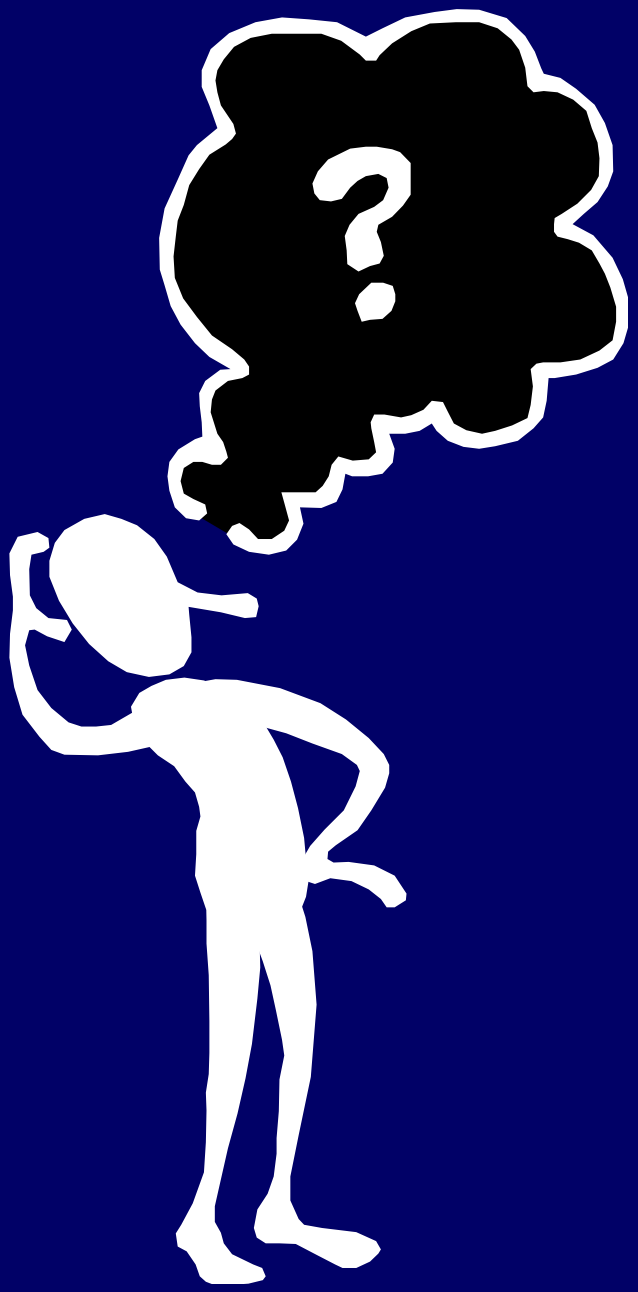
At best, p processors will give you a factor of p speedup over the time it takes on a single processor.

The traditional GCD algorithm will take linear time to operate on two n bit numbers. Can it be done faster in parallel?

If n^2 people agree to help you compute the GCD of two n bit numbers, it is not obvious that they can finish faster than if you had done it yourself



Is GCD
inherently
sequential?



No one knows.

Deep Blue

Many processors help DB look many chess moves ahead. It was not obvious that more processors could really help with game tree search. I remember a heated debate in C.B.'s Ph.D. thesis defense.