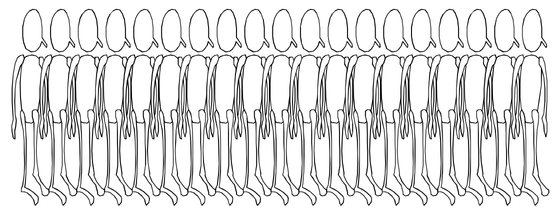
15-251

Great Theoretical Ideas in Computer Science

Grade School Again: A Parallel Perspective



a question

If a man can plough a field in 25 days, how long does it take for 5 men to plough the same field?

5 days

a similar question

If a processor can add two n-bit numbers in n microseconds,
how long does it take for n processors to add together two n-bit numbers?

hmm…

Warming up

thinking about parallelism

Dot products

a = (4 5 -2 1)
b = (1 -3 3 7)

$a = (a_1, a_2 \ldots a_n)$
$b = (b_1\ b_2 \ldots b_n)$

Dot product of a and b
a•b = 4.1 + 5.(-3) + (-2).3 + 1.7 = 10
Also called "inner product".

In general, a•b = $\sum_{i=1}^{n}(a_i b_i)$

Dot products

If we can add/multiply two numbers in time C, how long does it take to compute dot products for n-length vectors?

simplifying assumption for now

n multiplications

n-1 additions

hence, C*(2n-1) time.

Dot products

What if n people decided to compute dot products and they worked in parallel?

Modeling decision:
what are people allowed to do in parallel?

Assume they have shared memory

- Can read same location in memory in parallel
- Each location in memory can be written to by only one person at a time.
- Can write to different locations in memory simultaneously

Parallel dot products

What if n people decided to compute dot products and they worked in parallel? $a \cdot b$

All the pairwise products can be computed in parallel! (1 unit of time) $a_1b_1 \quad a_2b_2 \quad \cdot \quad a_nb_n$

How to add these n products up fast?

Binary tree

Parallel dot products

Assume C=1

What if n people decided to compute dot products and they worked in parallel?

All the pairwise products can be computed in parallel! (1 unit of time)

How to add these n products up fast?

Can add these numbers up in $\lceil \log_2 n \rceil$ rounds
Hence dot products take $\lceil \log_2 n \rceil$ +1 time in parallel.

Not enough people?

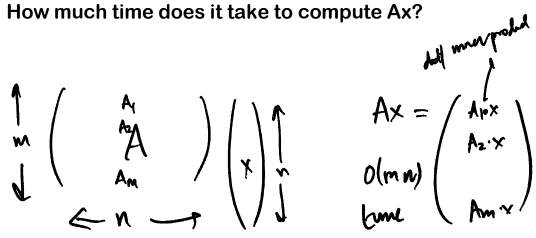
What if there were fewer than n people? $m < n$

(Q) What can you do? Exercise

(Q) Can you do better than $O(\log n)$ rounds if n people computing in parallel?

Another example: Matrix-vector multiplications

Suppose we were given a m*n matrix A
and a n-length vector x
How much time does it take to compute Ax?



How much time in parallel?

Since just m dot product computations
and all of them can be done in parallel

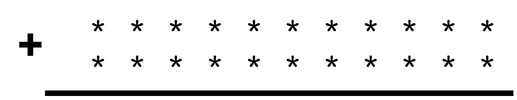
dot products computed in O(log n) time in parallel.
(using n people)

So if we had m*n people, we could compute the solution to Ax in $O(\log_2 n)$ time

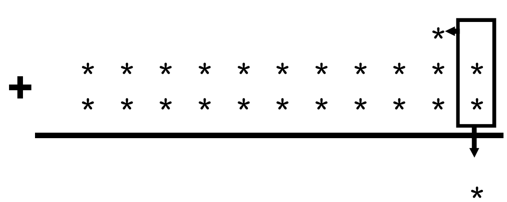
Back to our question…

If a single processor can
add two n-bit numbers
in n microseconds,
how long does it take
n processors
to add together
two n-bit numbers?

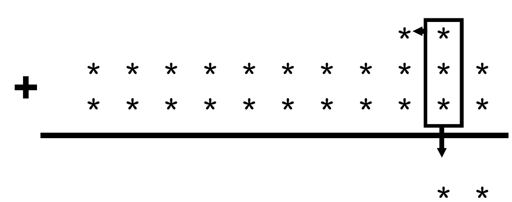
How to add 2 n-bit numbers.



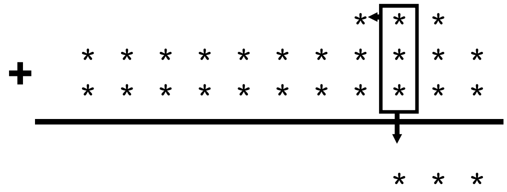
How to add 2 n-bit numbers.



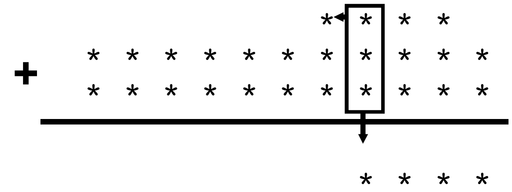
How to add 2 n-bit numbers.



How to add 2 n-bit numbers.

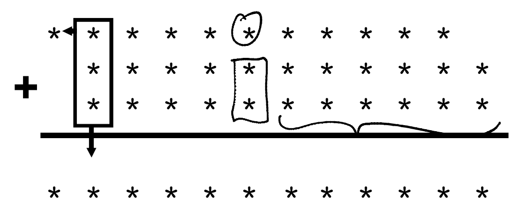


How to add 2 n-bit numbers.



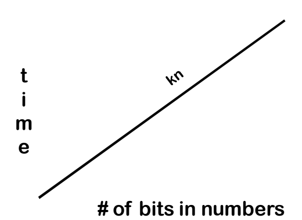
How to add 2 n-bit numbers.

How to add 2 n-bit numbers.

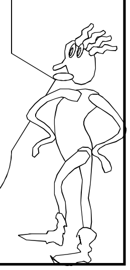


Let k be the maximum time that it takes you to do 

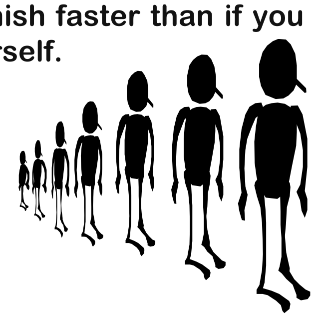
Time = kn
is proportional to n

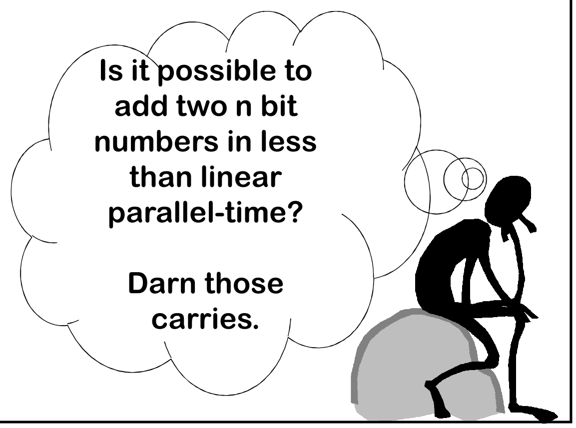


The time grow linearly with input size.



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.





Plus/Minus Binary (Extended Binary)

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

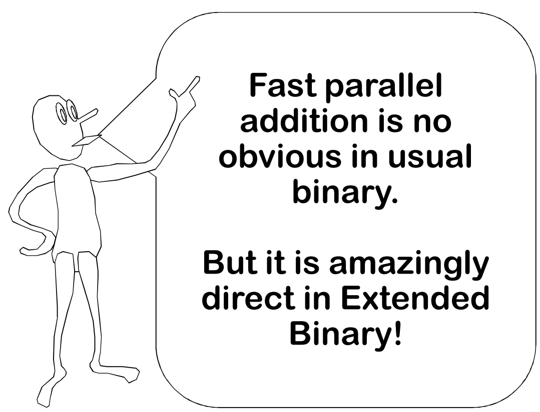
Example:

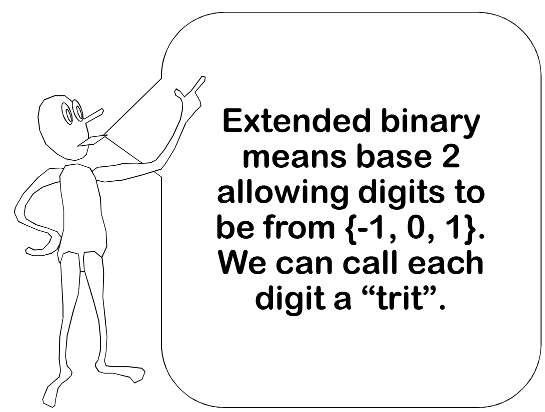
(1 -1 -1)= 4 -2 -1 = 1 = (0, 0, 1)

Not a unique representation system

1 1 1 -1 1 = 11

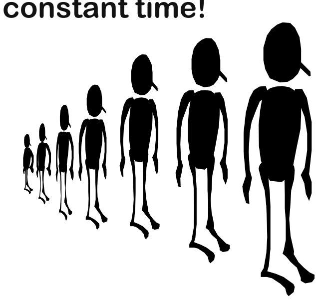
16 8 4 2 1



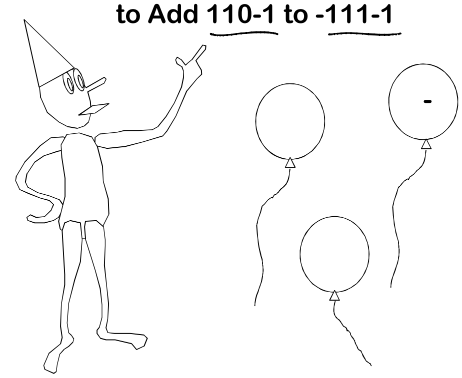


n people can add two n-trit plus/minus binary numbers in constant time!

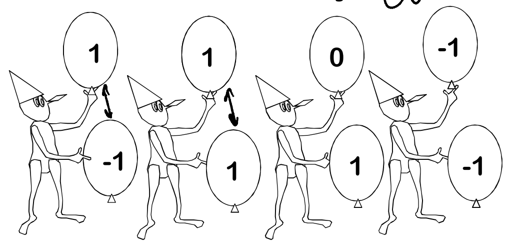
EB



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

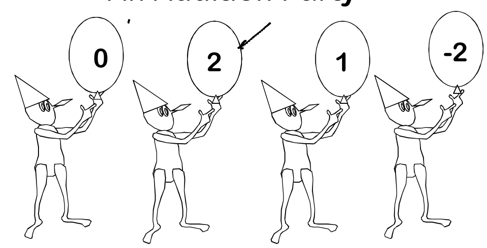


An Addition Party



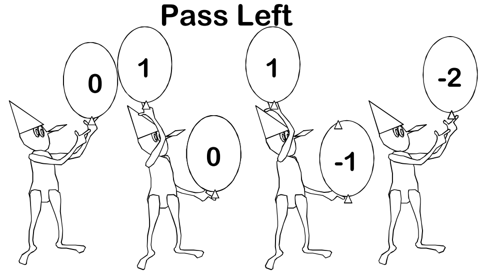
Invite n people to add two n-trit numbers
Assign one person to each trit position

An Addition Party



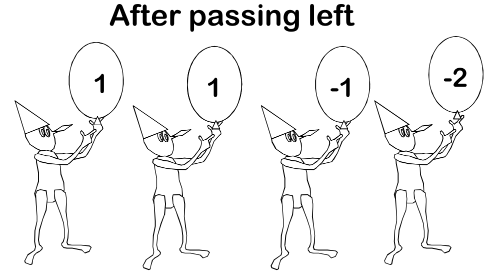
Each person should add the two input trits in their possession.
Problem: 2 and -2 are not allowed in the final answer.

Pass Left



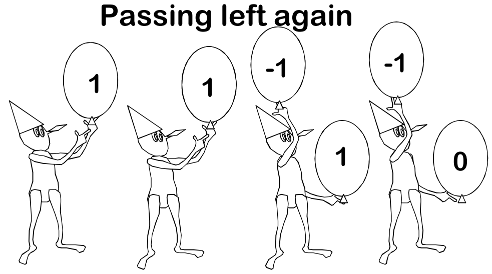
If you have a 1 or a 2 subtract 2 from yourself and pass a 1 to the left. (Nobody keeps more than 0)
Add in anything that is given to you from the right. (Nobody has more than a 1)

After passing left



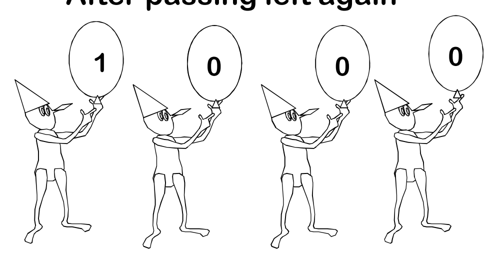
There will never again be any 2s
as everyone had at most 0
and received at most 1 more

Passing left again

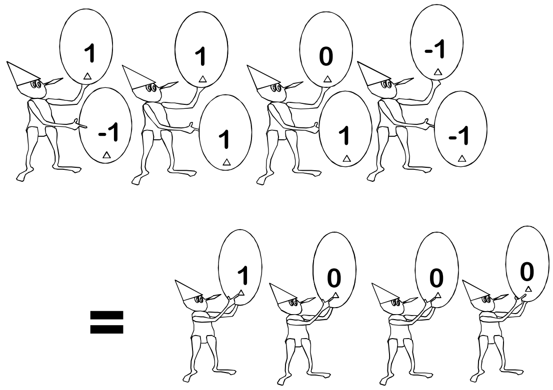


If you have a -1 or -2 add 2 to yourself
and pass a -1 to the left
(Nobody keeps less than 0)

After passing left again

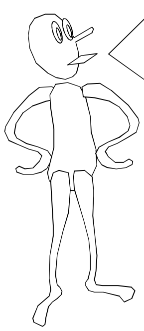


No -2s anymore either.
Everyone kept at least 0 and received at most -1.

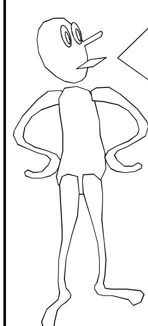


Strategy

To add two n-bit binary numbers
- Consider them to be in extended binary (EB)
  - no work required!
- Sum them up to get an answer in EB.
  - constant parallel time!
- Then convert them back to answer in binary
  - how do we do this fast in parallel???



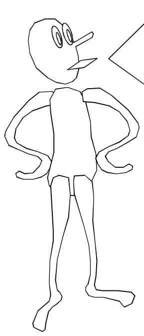
Is there a fast parallel way to convert an Extended Binary number into a standard binary number?



Both problems not quite obvious:

Sub-linear time addition in standard Binary.

Sub-linear time EB to Binary



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

Grade School Addition

$$
\begin{array}{r}
10111111100 \\
+ \quad 1011111101 \\
1000000110 \\
\hline
10100000011
\end{array}
$$

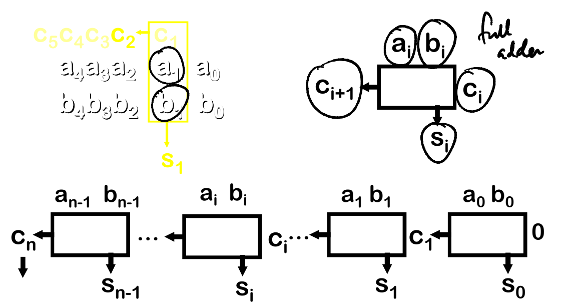
Grade School Addition

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

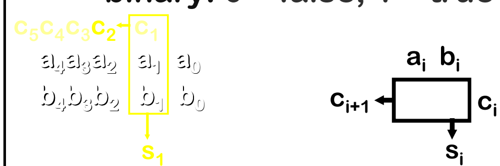
Grade School Addition

$$
\begin{array}{rl}
 & c_5 c_4 c_3 c_2 \leftarrow \boxed{c_1} \\
 & \quad a_4 a_3 a_2 \; a_1 \; a_0 \\
+ & \quad b_4 b_3 b_2 \; b_1 \; b_0 \\
\hline
 & \qquad\qquad s_1
\end{array}
$$

Ripple-carry adder



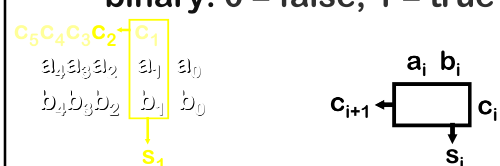
Logical representation of binary: 0 = false, 1 = true

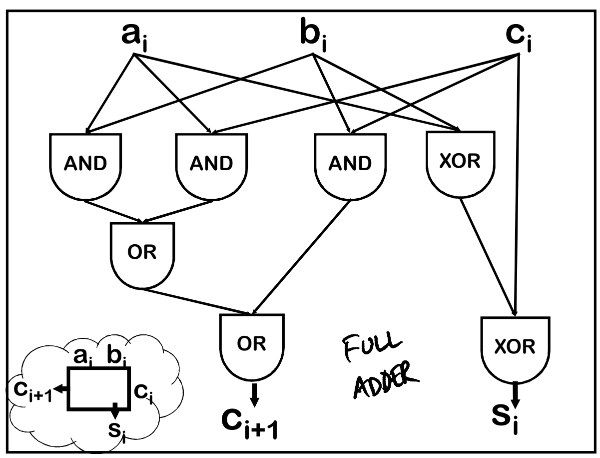


s_1 = (a_1 XOR b_1) XOR c_1

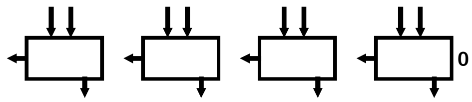
c_2 = (a_1 AND b_1)
OR (a_1 AND c_1)
OR (b_1 AND c_1)

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



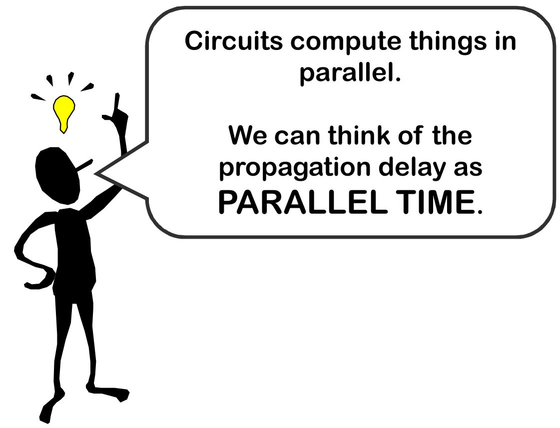


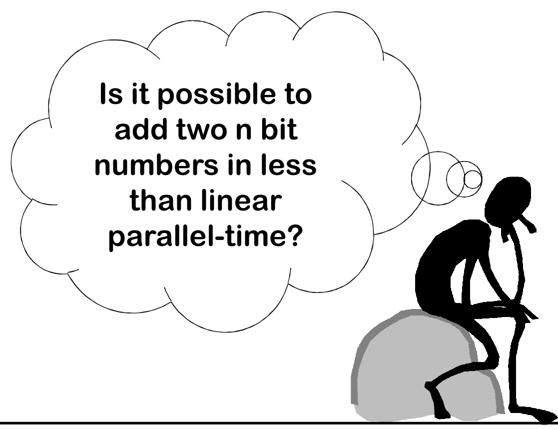
Ripple-carry adder

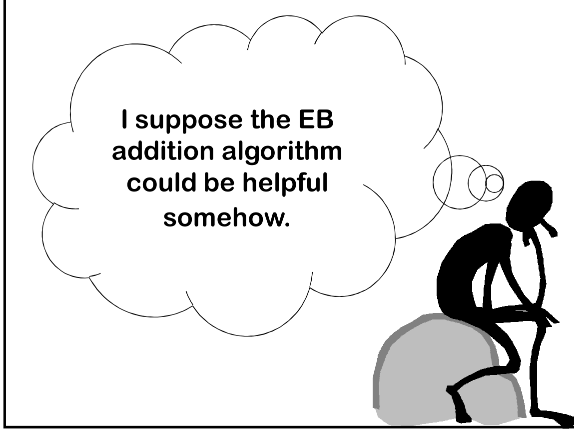


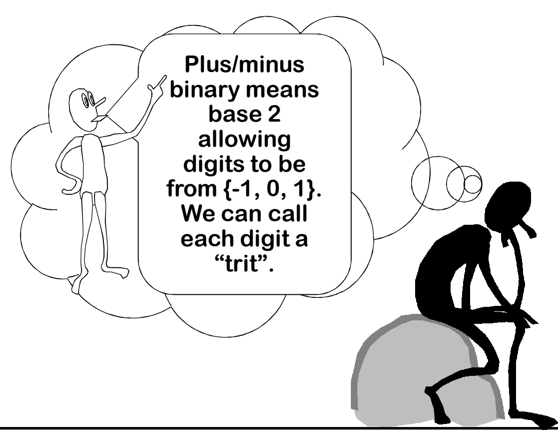
How long to add two n bit numbers?

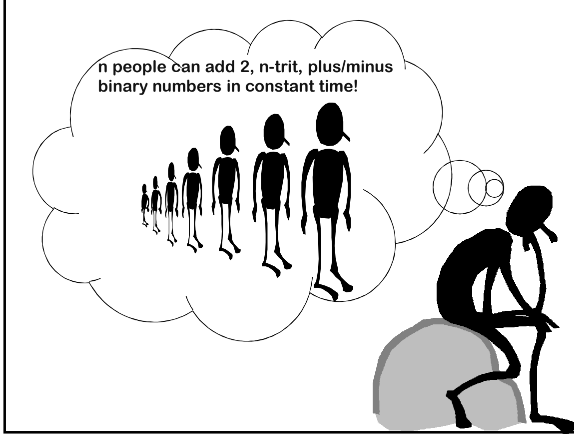
Propagation time through the circuit will be $\theta(n)$

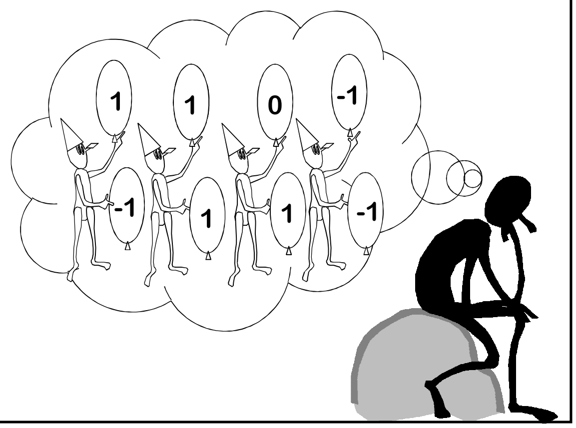


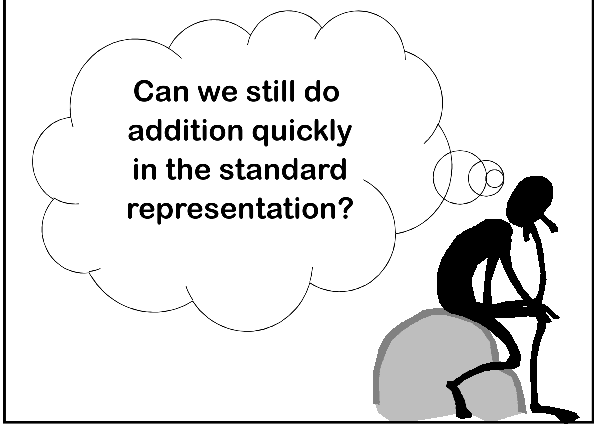












Yes, but first a neat idea…

Instead of adding
two numbers together
to make one number,

let's think about
adding 3 numbers
to make 2 numbers.

Carry-Save Addition

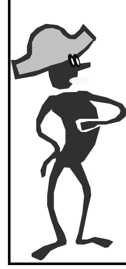
The sum of three numbers can be converted into the sum of 2 numbers in constant parallel time!

```
  1100111011
+ 1011111101
+ 1000000110
------------
```

Carry-Save Addition

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

```
  1100111011
+ 1011111101
+ 1000000110
------------
   1111000000   XOR
+ 10001111110   Carries
```

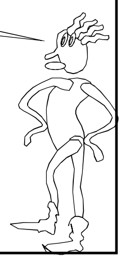


Cool!

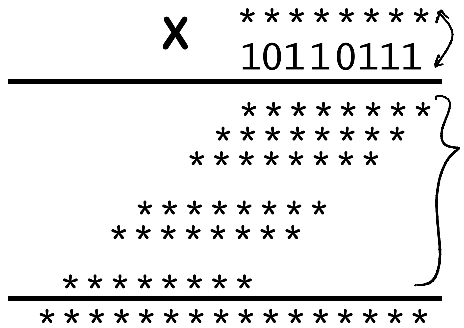
So if we if represent x as a+b, and y as c+d, then can add x,y using two of these (this is basically the same as that extended binary thing).

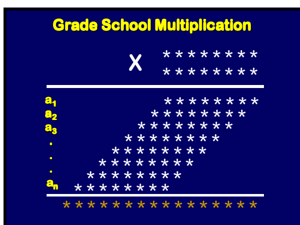
(a+b+c)+d=(e+f)+d=g+h

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



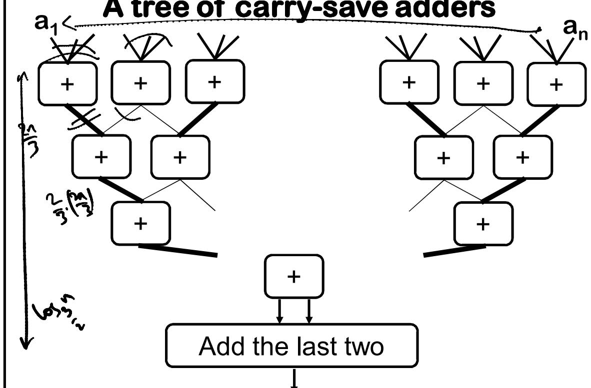
Grade School Multiplication



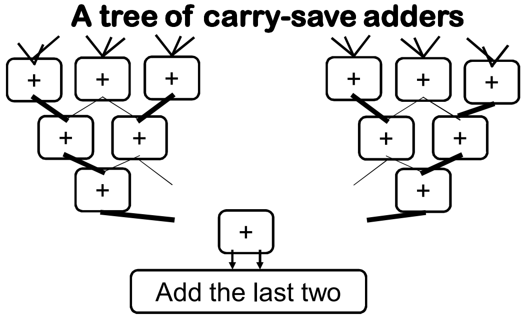


We need to add n 2n-bit numbers:
$a_1, a_2, a_3, \ldots, a_n$

A tree of carry-save adders



A tree of carry-save adders

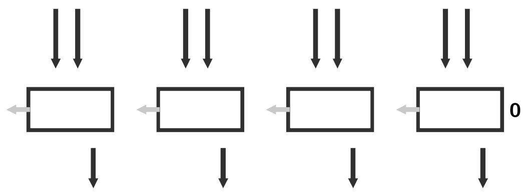


$T(n) \approx \log_{3/2}(n)$ + [last step]

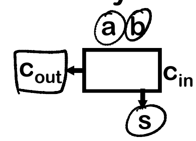
So let's go back to the problem of adding two numbers.

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

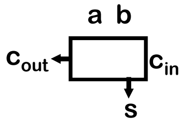


What do we know about the carry-out before we know the carry-in?



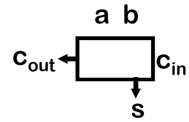
a	b	C_{out}
0	0	0
0	1	Carry in
1	0	Carry in
1	1	1

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



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

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

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 actually have the carries, it will only takes one step more:

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

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

a_i XOR b_i XOR c_i

But we only have the carries in this peculiar notation!

How fast can we convert from this notation back to the standard notation?

$$1011111100$$

Called the "parallel prefix problem"

Idea #2:

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

0

$$\boxed{(1 ☺ (0 ☺(\leftarrow ☺(\leftarrow ☺ (\leftarrow ☺ (\leftarrow ☺ (\leftarrow ☺ (1 ☺ (\leftarrow ☺ 0)))))))))}$$

1 0 1 1 1 1 1

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.

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

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

$$=$$

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

$$=$$

$$\leftarrow ☺ 1 ☺ 0 \;=\; 1$$

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 of binary associative operators

- 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: $\boxed{X_{n-1}, X_{n-2}, \ldots, X_1, X_0}$
Output: $Y_{n-1}, Y_{n-2}, \ldots, Y_1, Y_0$
where
$Y_0 = X_0$ ✓
$Y_1 = X_0 + X_1$ ✓
$Y_2 = X_0 + X_1 + X_2$ ✓
$Y_3 = X_0 + X_1 + X_2 + X_3$
⋮
$Y_{n-1} = X_0 + X_1 + X_2 + X_3 + \ldots + X_{n-1}$

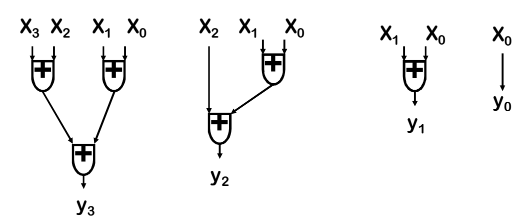
1 ← ←1 ←3
1 1 1 1 0 0

Prefix Sum example when + = addition

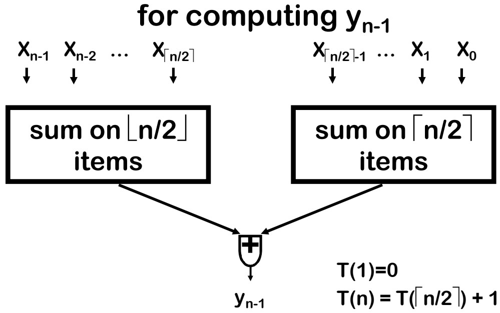
Input: 6, 9, 2, 3, 4, 7
Output: 31, 25, ~~16, 14, 11~~, 7
where
$Y_0 = X_0$
$Y_1 = X_0 + X_1$
$Y_2 = X_0 + X_1 + X_2$
$Y_3 = X_0 + X_1 + X_2 + X_3$
⋮
$Y_{n-1} = X_0 + X_1 + X_2 + X_3 + \ldots + X_{n-1}$

7
11
14
16
25
31

Example circuitry (n = 4)



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

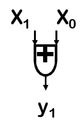


$T(1)=0$
$T(n) = T(\lceil n/2 \rceil) + 1$
$T(n) = \lceil \log_2 n \rceil$

Slightly more fancy construction coming up…

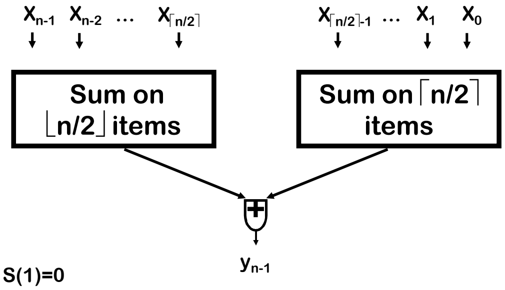
The above construction had small parallel run-time

But it used a lot of addition gates



Let's calculate how many we used…

Size of Circuit (number of gates)

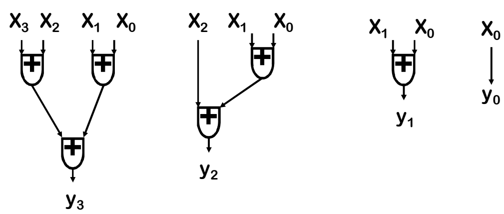


S(1)=0

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

S(n) = n-1

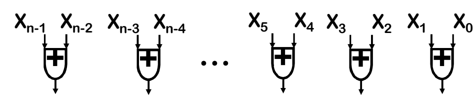
Sum of Sizes



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

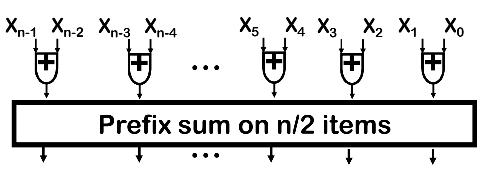
Recursive Algorithm n items (n = power of 2)

If n = 1, $Y_0 = X_0$;



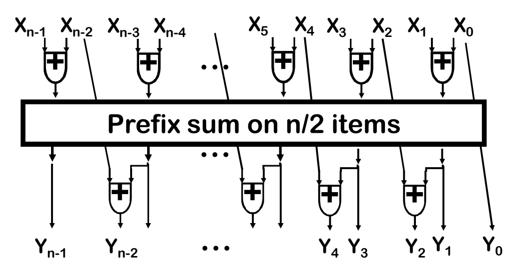
Recursive Algorithm n items (n = power of 2)

If n = 1, $Y_0 = X_0$;



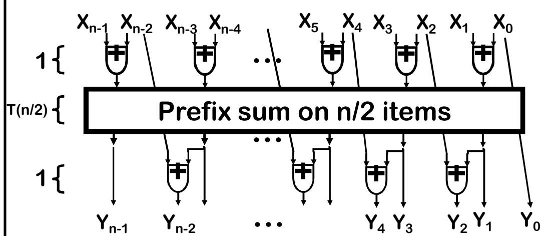
Recursive Algorithm n items (n = power of 2)

If n = 1, $Y_0 = X_0$;



Parallel time complexity

If n = 1, $Y_0 = X_0$;

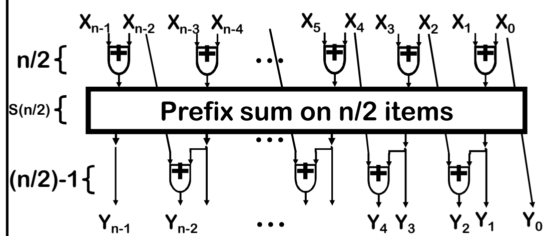


T(1)=0; T(2) = 1; T(n) = T(n/2) + 2

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

Size

If n = 1, $Y_0 = X_0$;



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

$S(n) = 2n – \log_2 n -2$

End of fancier construction

Putting it all together: Carry Look-Ahead Addition

To add two n-bit numbers: a and b

Compute carry in the peculiar notation $_{\leftarrow}01$

Convert it to carry in the standard notation

The sum is carry XOR (a XOR b)

Putting it all together: Carry Look-Ahead Addition

To add two n-bit numbers: a and b

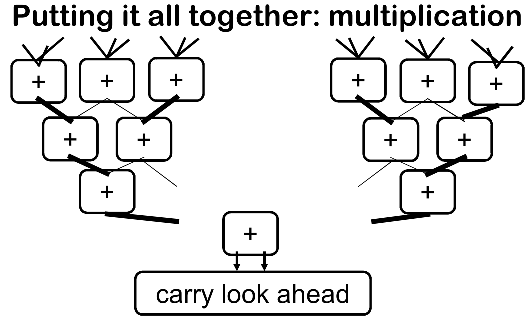
1 step to compute x values ($_{\leftarrow}01$)

$2 \log_2 n - 1$ steps to compute carries c

1 step to compute c XOR (a XOR b)

$2 \log_2 n + 1$ parallel steps total

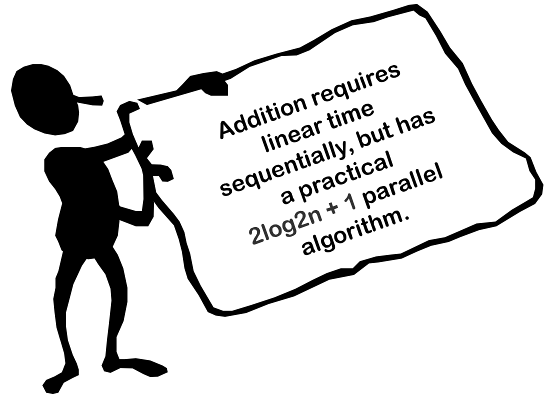
Putting it all together: multiplication

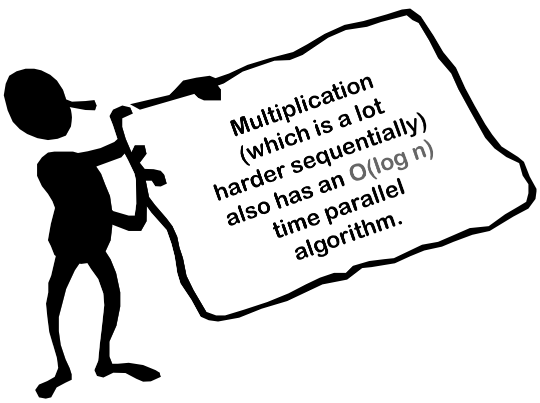


$T(n) \approx \log_{3/2}(n) + 2\log_2 2n + 1$

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

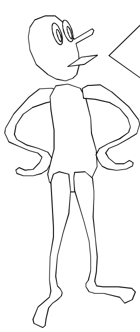
Addition & Multiplication
Both $O(\log n)$





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

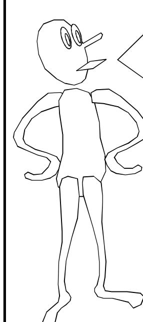
Processor	n	$2\log_2 n + 1$
80186	16	9
Pentium	32	11
Alpha	64	13



In order to handle integer addition/subtraction we use 2's compliment representation, e.g., -44=

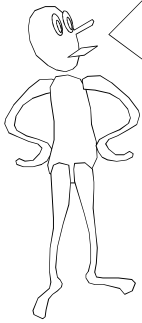
-64	32	16	8	4	2	1
1	0	1	0	1	0	0

44= 0 1 0 1 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

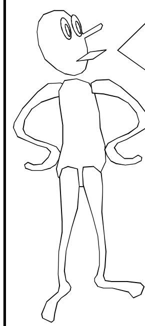


To negate a number, flip each of its bits and add 1.

-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

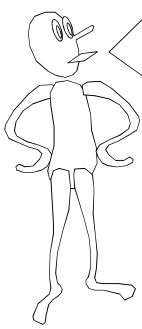


To negate a number, flip each of its bits and add 1.

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

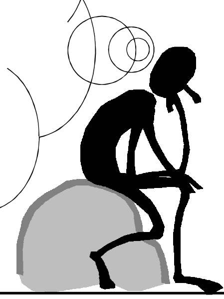
x + flip(x) = -1.

So, -x = flip(x)+1.



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

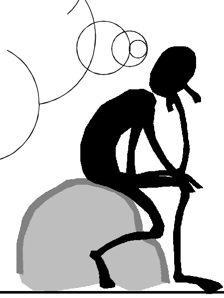
If millions of processors, how much of a speed-up might I get over a single processor?



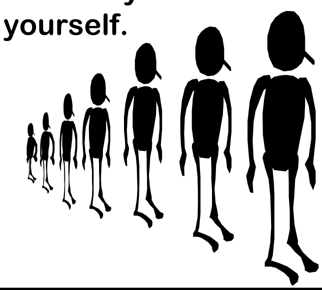
Brent's Law

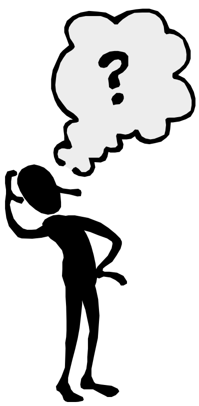
At best, p processors will give you a factor of p speedup over the time it takes on a single processor.

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

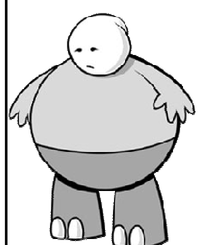


If n^2 people agree to help you compute the GCD of two n bit numbers, it is not obvious that they can finish faster than if you had done it yourself.





No one knows.



Here's What You Need to Know…

Parallel computation

addition in extended binary

one-bit adder

ripple-carry adders

computing carries using parallel prefix sum

addition in parallel O(log n) time

mult. in parallel O(log n) time

one's complement