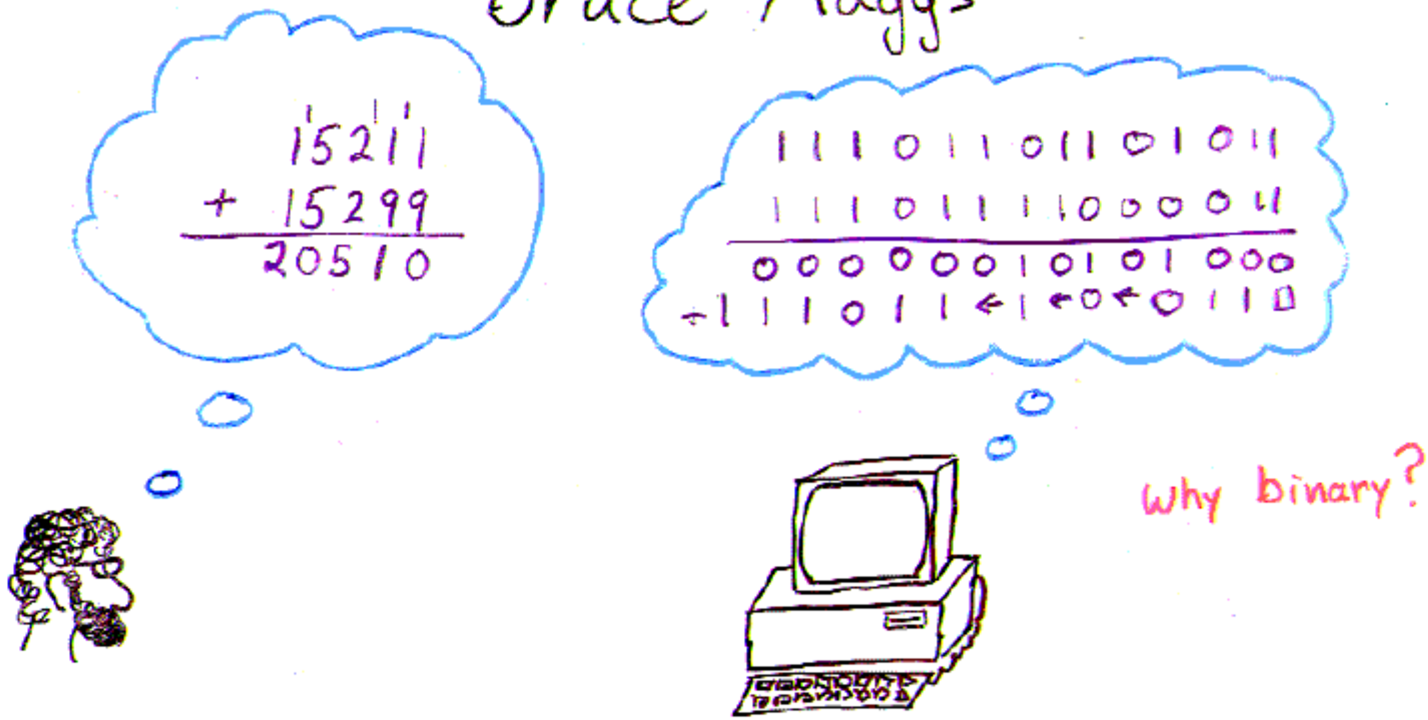
How Computers Really Do Arithmetic

Bruce Maggs



- addition
- multiplication
- division
- square roots

"Schoolboy" Arithmetic

- carry-propagate addition

decimal

$$
\begin{array}{r}
\overbrace{1\,9\,9\,7}^{n \text{ digits}} \\
+\;3\,0\,0\,3 \\
\hline
5\,0\,0\,0
\end{array}
$$

binary

$$
\begin{array}{r}
\overbrace{1\,0\,1\,1\,1\,1\,0\,1}^{n \text{ bits}} \\
+\;1\,0\,0\,0\,0\,0\,1\,1 \\
\hline
1\,0\,1\,0\,0\,0\,0\,0\,0
\end{array}
$$

"Schoolboy" Arithmetic

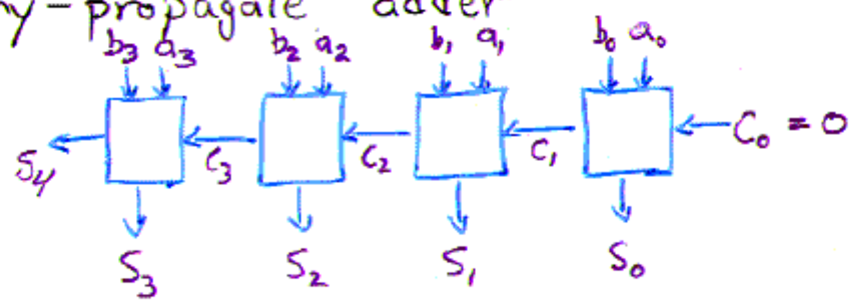
- carry-propagate addition

decimal	binary
$\overbrace{}^{n \text{ digits}}$	$\overbrace{}^{n \text{ bits}}$

$$
\begin{array}{r}
1997 \\
+\,3003 \\
\hline
5000
\end{array}
\qquad\qquad
\begin{array}{r}
10111101 \\
+\,10000011 \\
\hline
101000000
\end{array}
$$

- carry-propagate adder

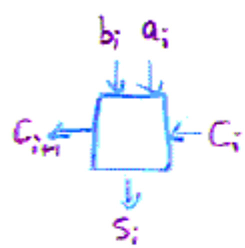
$$
\begin{array}{rrrrr}
 & a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
\hline
s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
$$



where

$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i \cdot b_i \lor a_i c_i \lor b_i c_i$$



Time?
n
why?

Carry-Look-Ahead Addition

To add
$$
\begin{array}{r}
1\,0\,1\,1\,1\,1\,0\,1 \\
+\;1\,0\,0\,0\,0\,0\,1\,1 \\
\hline
\end{array}
$$

1) compute parity p_i at each bit position

$$p_i = a_i \oplus b_i$$

1 time step (compute all p_i values in parallel)

$$
\begin{array}{r}
1\,0\,1\,1\,1\,1\,0\,1 \\
1\,0\,0\,0\,0\,0\,1\,1 \\
\hline
0\,0\,1\,1\,1\,1\,1\,0
\end{array}
$$

Carry-Look-Ahead Addition

To add

$$
\begin{array}{rcccccccc}
 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
\end{array}
$$

1) compute parity p_i at each bit position

$$p_i = a_i \oplus b_i$$

1 time step (compute all p_i values in parallel)

$$
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0
\end{array}
$$

2) compute carry-in C_i at each bit position

$$
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
1 & 0 & \underset{1}{\leftarrow} & \underset{1}{\leftarrow} & \underset{1}{\leftarrow} & \underset{1}{\leftarrow} & \underset{1}{\leftarrow} & 1
\end{array}
$$

How? What does this notation mean?

Carry-Look-Ahead Addition

To add
$$
\begin{array}{r}
1\,0\,1\,1\,1\,1\,0\,1 \\
+\;1\,0\,0\,0\,0\,0\,1\,1 \\
\hline
\end{array}
$$

1) compute parity p_i at each bit position

$$p_i = a_i \oplus b_i$$

1 time step

(compute all p_i values in parallel)

$$
\begin{array}{r}
1\,0\,1\,1\,1\,1\,0\,1 \\
1\,0\,0\,0\,0\,0\,1\,1 \\
\hline
0\,0\,1\,1\,1\,1\,1\,0
\end{array}
$$

2) compute carry-in c_i at each bit position

$$
\begin{array}{r}
1\,0\,1\,1\,1\,1\,0\,1 \\
1\,0\,0\,0\,0\,0\,1\,1 \\
\hline
1\;0\;\underset{1}{\leftarrow}\underset{1}{\leftarrow}\underset{1}{\leftarrow}\underset{1}{\leftarrow}\underset{1}{\leftarrow}\;1
\end{array}
$$

How? What does this notation mean?

3) compute $s_i = p_i \oplus c_i$ at each position

1 time step

$$
\begin{array}{r}
0\,0\,1\,1\,1\,1\,1\,0 \\
\oplus\;1\,0\,1\,1\,1\,1\,1\,1\,0 \\
\hline
1\,0\,1\,0\,0\,0\,0\,0\,0\,0
\end{array}
$$

Prefix Sums Calculation

Input: sequence $X = x_{n-1}, x_{n-2}, \ldots, x_1, x_0$

binary associative operator $+$

(associative : $(x+y)+z = x+(y+z)$)

Output: sequence $Y = y_{n-1}, y_{n-2}, \cdots, y_1, y_0$

where $y_i = \sum_{j=0}^{i} x_j$

Example: $+ \equiv$ integer addition

$X = 3, 2, 1, 4, 8, 2$

$Y = 20 \; 17 \; 15 \; 14 \; 10 \; 2$

Prefix Sums Calculation

Input: sequence $X = x_{n-1}, x_{n-2}, \ldots, x_1, x_0$

binary associative operator $+$

(associative : $(x+y)+z = x+(y+z)$)

Output: sequence $Y = y_{n-1}, y_{n-2}, \cdots, y_1, y_0$

where $y_i = \sum_{j=0}^{i} x_j$

Example: $+ \equiv$ integer addition

$X = 3, 2, 1, 4, 8, 2$

$Y = 20 \; 17 \; 15 \; 14 \; 10 \; 2$

Other Associative Operators:

multiplication, min, max, and, or, left, right

left$(a,b) = a$ right$(a,b) = b$

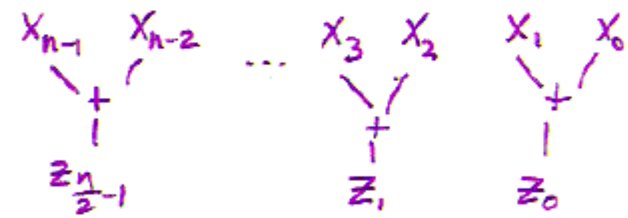
Algorithm for Computing Prefix Sums

(assume n is a power of 2)

Base case: $n=1$, $y_0 = x_0$

Recursive case: 1) let $z_i = x_{2i+1} + x_{2i}$, $0 \le i \le \frac{n}{2}-1$

i.e. add pairs in X:



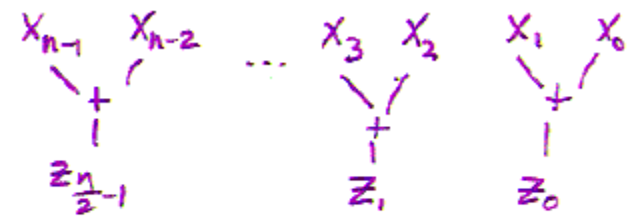
Algorithm for Computing Prefix Sums

(assume n is a power of 2)

Base case: $n = 1$, $y_0 = x_0$

Recursive case: 1) let $z_i = x_{2i+1} + x_{2i}$, $0 \le i \le \frac{n}{2} - 1$

i.e. add pairs in X:



2) recursively compute prefix sums

$W = w_{\frac{n}{2}-1}, \ldots, w_1, w_0$ for sequence Z

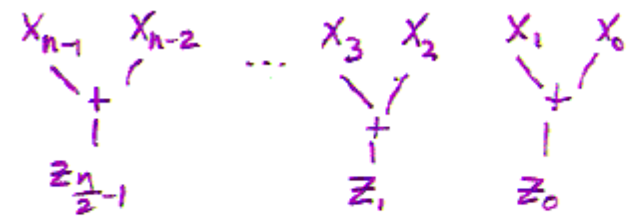
Algorithm for Computing Prefix Sums

(assume n is a power of 2)

Base case: $n=1$, $Y_0 = X_0$

Recursive case: 1) let $Z_i = X_{2i+1} + X_{2i}$, $0 \le i \le \frac{n}{2}-1$

i.e. add pairs in X:



2) recursively compute prefix sums

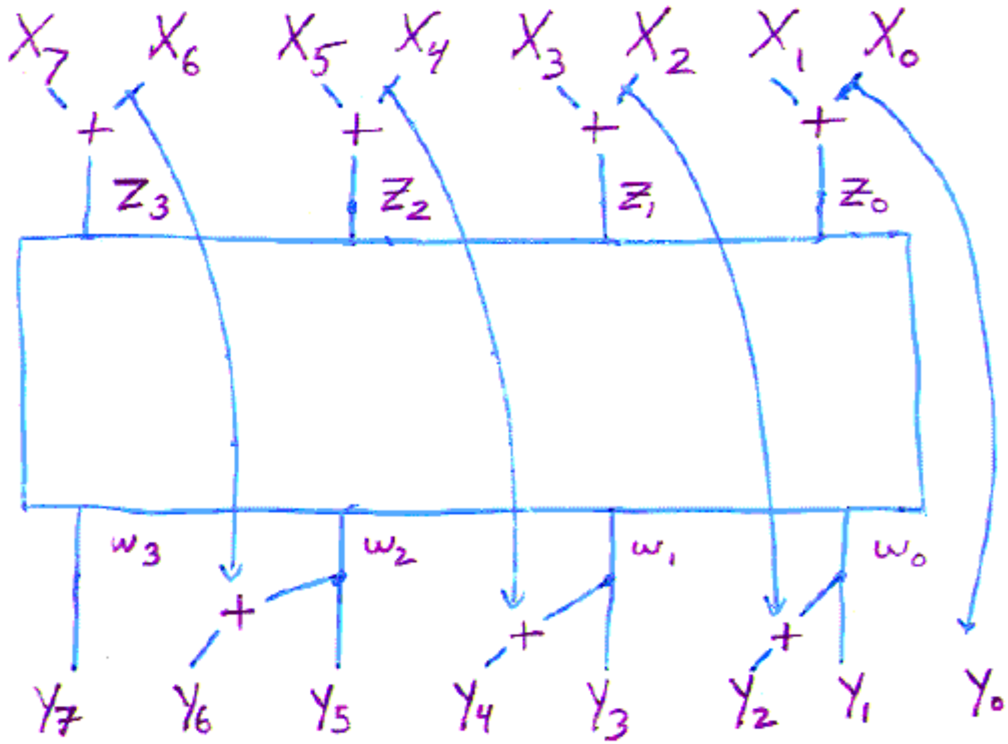
$W = w_{\frac{n}{2}-1}, \ldots, w_1, w_0$ for sequence Z

3) let

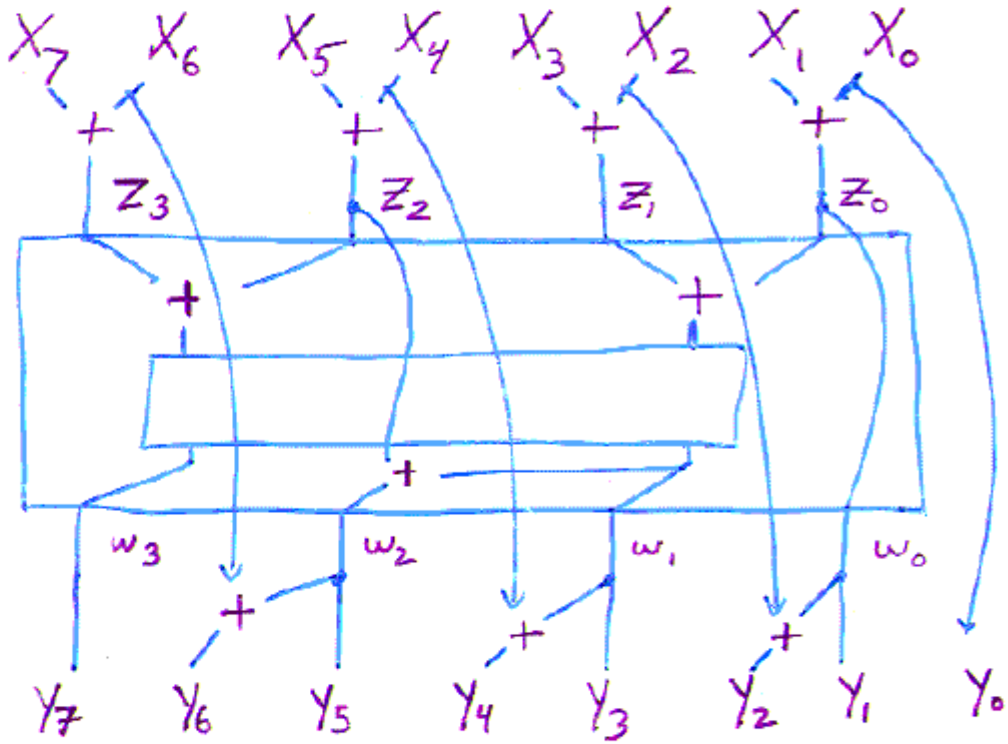
$$Y_i = \begin{cases} W_{\frac{i-1}{2}} & i \text{ odd} \\ X_i + W_{\frac{i}{2}-1} & i \text{ even}, i > 0 \\ X_0 & i = 0 \end{cases}$$

i.e. $Y = w_{\frac{n}{2}-1}, \ldots, x_4 + w_1, w_1, x_2 + w_0, w_0, x_0$

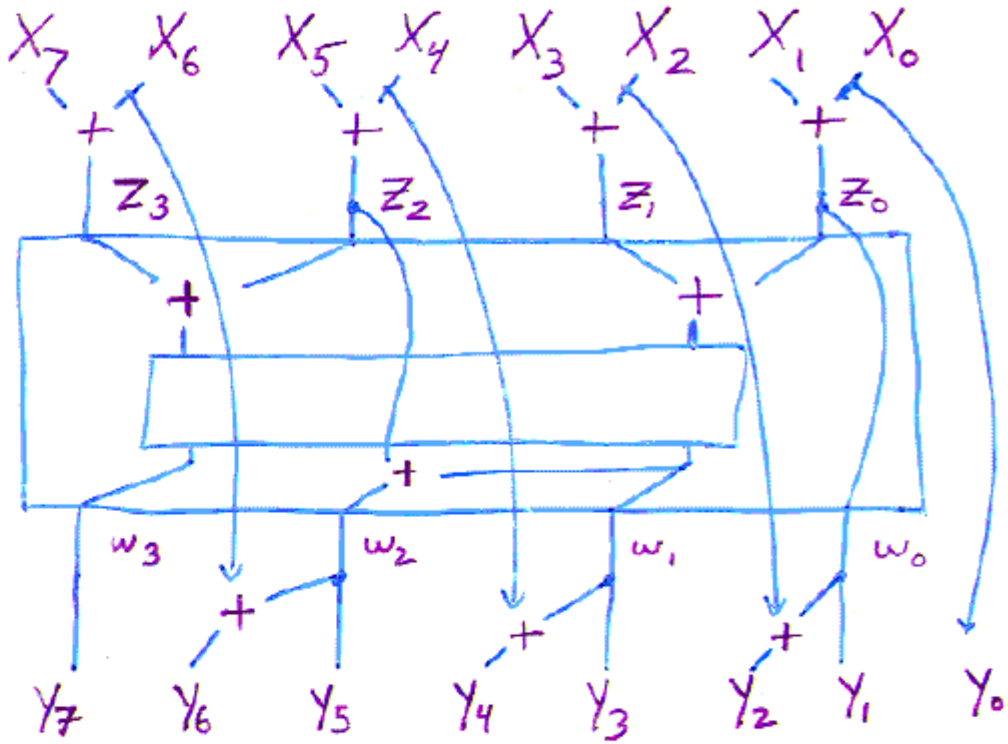
Circuit Diagram



Circuit Diagram



Circuit Diagram



Time: $T(n) = \begin{cases} 1 & n = 1 \\ 1 + T(\frac{n}{2}) + 1 & n > 1 \end{cases}$

$= 2\log_2 n + 1$

Better than n.

Back to carry-lookahead addition

$$1\ 0\ \underbrace{\overset{1}{\leftarrow}\ \overset{1}{\leftarrow}\ \overset{1}{\leftarrow}\ \overset{1}{\leftarrow}\ \overset{1}{\leftarrow}}\ 1$$

we want to replace each ← with a 1, and not sequentially!

What's the connection to prefix sums?

define operator * as follows

*	0	1	←
0	0	0	0
1	1	1	1
←	0	1	←

now apply to

$$X = 1\ 0\ \leftarrow\ \leftarrow\ \leftarrow(\leftarrow(\leftarrow 1))$$

$$Y = 1\ 0\ 1\ 1\ 1\ \underline{1}\ \underline{1}\ 1$$

Back to carry-lookahead addition

$$1\ 0\ \underbrace{\overset{1}{\leftarrow}\ \overset{1}{\leftarrow}\ \overset{1}{\leftarrow}\ \overset{1}{\leftarrow}\ \overset{1}{\leftarrow}}\ 1$$

we want to replace each ← with a 1, and not sequentially!

What's the connection to prefix sums?

define operator * as follows

*	0	1	←
0	0	0	0
1	1	1	1
←	0	1	←

now apply to

$X = 1\ 0\ \leftarrow \leftarrow \leftarrow (\leftarrow (\leftarrow 1))$

$Y = 1\ 0\ 1\ 1\ 1\ \underline{1}\ \underline{1}\ 1$

We should verify that * is associative. Easy if $x = 0, 1$.

x	y	z	x * (y * z)	(x * y) * z
0	0	0	0	0
0	0	1	0	0
0	0	←	0	0
⋮			⋮	⋮

x	y	z	x * (y * z)	(x * y) * z
←	0	0	0	0
←	0	1	0	0
←	0	←	0	0
←	1	0	1	1
←	1	1	1	1
←	1	←	1	1
←	←	0	0	0
←	←	1	1	1
←	←	←	←	←

Bottom Line:

Time to add 2 n-bit numbers =

$$\underbrace{2\log_2 n + 1}_{\substack{\text{compute } c_i\text{'s} \\ \text{(and } p_i\text{'s)}}} \quad \underbrace{+\ 1}_{\substack{\text{compute} \\ s_i = p_i \oplus c_i}} = 2\log_2 n + 2$$

processor	n	$2\log_2 n + 2$
80186	16	10
Pentium	32	12
Alpha	64	14

More "Schoolboy" Arithmetic

MULTIPLICATION

decimal

```
   n
  211
x 299
 1899
1899
422
63089
```

binary

```
                  n
         0 1 1 0 1 0 0 1 1
       x 1 0 0 1 0 1 0 0 1
         0 1 1 0 1 0 0 1 1      a0
     0 1 1 0 1 0 0 1 1          a1
   0 1 1 0 1 0 0 1 1            a2
 0 1 1 0 1 0 0 1 1               ⋮
```

up to n "partial products"

Time? ≈ n^2 using one-column-at-a-time addition

More "Schoolboy" Arithmetic

MULTIPLICATION

decimal

$$\begin{array}{r} \overbrace{211}^{n} \\ \times\ 299 \\ \hline 1899 \\ 1899 \\ 422 \\ \hline 63089 \end{array}$$

binary

$$\begin{array}{r} \overbrace{011010011}^{n} \\ \times\ 100101001 \\ \hline 011010011 \\ 011010011 \\ 011010011 \\ 011010011 \\ \hline \end{array}$$

up to n "partial products"

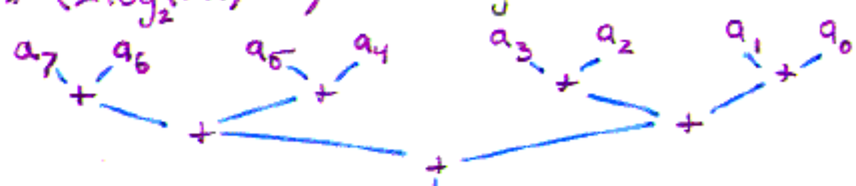
a_0
a_1
a_2
$\vdots$

Time? $\approx n^2$ using one-column-at-a-time addition

$\approx n \cdot 2\log_2(2n)$ computing sequentially $a_0 + a_1$, $a_0 + a_1 + a_2$, ... (with carry look-ahead)

Why $2n$?

$\approx \log_2 n * (2\log_2(2n)+2)$ using tree



We can beat this!

Carry-Save Addition

Idea: Can convert sum of 3 numbers into a sum of 2 numbers in one step.

Example

```
            0 1 1 0 1 0 0 1 1
      0 1 1 0 1 0 0 1 1 0 0 0
  0 1 1 0 1 0 0 1 1 0 0 0 0 0
  ---------------------------
 <. . .   1 1 0 1 0 1 1        ← parity
        1 0 1 0 0 0 0          ← carry in
```

Carry-Save Addition

Idea: Can convert sum of 3 numbers into a sum of 2 numbers in one step.

Example

```
        0 1 1 0 1 0 0 1 1
    0 1 1 0 1 0 0 1 1 0 0 0
0 1 1 0 1 0 0 1 1 0 0 0 0 0
---------------------------
  <· · ·   1 1 0 1 0 1 1      ← parity
         1 0 1 0 0 0 0        ← carry in
```

Wallace Tree

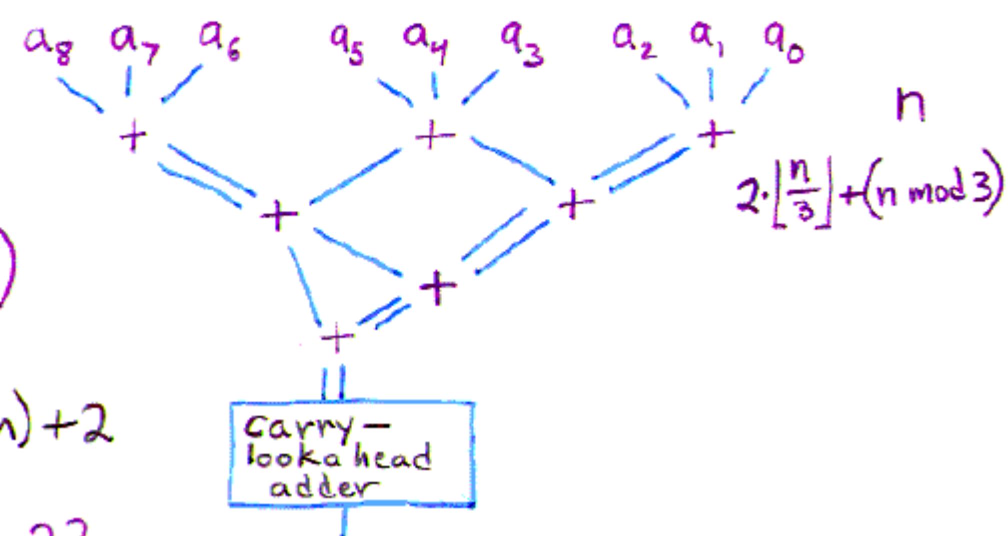
depth of tree $\approx \log_{3/2} n$

(exacty 9 for $n=64$)

total time $\approx \log_{3/2} n + 2\log(2n)+2$

time for $n=64$: 23

(vs. 14 to add!!)



"Schoolboy" Division

$$
\begin{array}{r}
1.0057\ldots \\
15211\,\overline{)\,15299.} \\
-15211 \\
\hline
88000 \\
76055 \\
\hline
119450
\end{array}
$$

$$
\begin{array}{r}
0 \\
15211 \\
30422 \\
45633 \\
60844 \\
76055 \\
\vdots
\end{array}
$$

"Schoolboy" Division

```
           1 . 0 0 5 7 . . .
15211 | 1 5 2 9 9 .
       - 1 5 2 1 1
                9   9
             8 8 7 0 0 0
             7 6 0 5 5
             1 1 9 4 5 0
```

```
  0
15211
30422
45633
60844
76055
  ⋮
```

Time to get n digits of precision:

- build table of multiples of divisor
  $1 * 15211,\ 2 * 15211,\ 3 * 15211, \ldots,\ 9 * 15211$

- n table lookups
- n subtractions   ($\approx 2 \log n + 2$ time each)

Total: $\approx n(2 \log n + 2)$
(We can do better !!!)

A few words about subtraction

- it never costs more than addition

Almost all machines use 2's-complement representation of signed integers.

example:

−128	+64	+32	+16	+8	+4	+2	+1	
1	1	0	0	1	1	0	1	= −51

A few words about subtraction

- it never costs more than addition

Almost all machines use 2's-complement representation of signed integers.

example:

−128	+64	+32	+16	+8	+4	+2	+1
1	1	0	0	1	1	0	1

= −51

Assuming no overflows, addition is unchanged:

$$\begin{array}{rcccccccc l}
 & & & & & 1 & & & & \\
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \quad -51 \\
+ & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & \quad +45 \\
\hline
 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \quad -6
\end{array}$$

A few words about subtraction

- it never costs more than addition

Almost all machines use 2's-complement representation of signed integers.

example:

−128	+64	+32	+16	+8	+4	+2	+1	
1	1	0	0	1	1	0	1	= −51

Assuming no overflows, addition is unchanged:

$$
\begin{array}{cccccccccc}
 & & & & & 1 & & & & \\
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & -51 \\
+ & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & +45 \\
\hline
 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & -6
\end{array}
$$

To negate a number, invert all bits and then add 1.

$$
\begin{array}{cccccccccc}
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & -51 \\
 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & +51 \\
 & & & & & & & & +1 & \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &
\end{array}
$$

Redundant Representation of Integers

Idea: Allow each digit to be 0, 1, −1

examples: 0 1 0 1 = 5

1 0 −1 −1 = 5

New addition algorithm:

1) add corresponding digits, no carries

		0	1	0	1	−1	0	1	1	75
+	1	0	−1	0	−1	0	1	0		90
	1	1	−1	1	−2	0	2	1		165

Redundant Representation of Integers

Idea: Allow each digit to be 0, 1, −1

examples:

0	1	0	1	= 5
1	0	−1	−1	= 5

New addition algorithm:

1) add corresponding digits, no carries

	0	1	0	1	−1	0	1	1	75
+	1	0	−1	0	−1	0	1	0	90
	1	1	−1	1	−2	0	2	1	165

2) remove +2's by changing to 0 passing +1 left
" +1's " " " −1 " +1 "

1 0 −1 0 −1 −2 1 1 −1

Now all digits are −2, −1, 0, 1 why?

Redundant Representation of Integers

Idea: Allow each digit to be 0, 1, −1

examples: 0 1 0 1 = 5

1 0 −1 −1 = 5

New addition algorithm:

1) add corresponding digits, no carries

	0	1	0	1	−1	0	1	1	75
+	1	0	−1	0	−1	0	1	0	90
	1	1	−1	1	−2	0	2	1	165

2) remove +2's by changing to 0 passing +1 left

" +1's " " " −1 " +1 "

1 0 −1 0 −1 −2 1 1 −1

Now all digits are −2, −1, 0, 1 why?

3) remove −2's by changing to 0 passing −1 left

" −1's " " " +1 " −1 "

1 −1 +1 −1 0 0 1 0 +1

We've added 2 n-bit numbers in 3 steps!

SRT Division Algorithm

```
        1 1 -1 1 1 0  r  -1 -1 1
1011 | 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
              0 1 2
               1 0 0 0
              -1 0 -1 -1
                0 -1 -1 1
```

```
   21 r 6
11 | 237
```

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

22 r-5

SRT Division Algorithm

```
          1  1 -1  1  0  r  -1 -1  1
1011 ) 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
                0  1  2
                1  0  0  0
               -1  0 -1 -1
                   0 -1 -1  1
```

```
   21 r 6
11 ) 237
```

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

22 r-5

Time for n bits of precision in result:

$$\approx \underbrace{3 \cdot n}_{\text{1 addition per bit}} + \underbrace{2 \log n + 2}_{\text{convert to standard representation by subtracting negative bits from positive}}$$

Intel Pentium Division Error

- used essentially the same algorithm, but computed more than one bit of result in each step. Examined several leading bits of divisor and remainder and looked in table.

- table had several bad entries
- ultimately Intel offered to replace any defective chip, estimating their loss at

$ 475 million

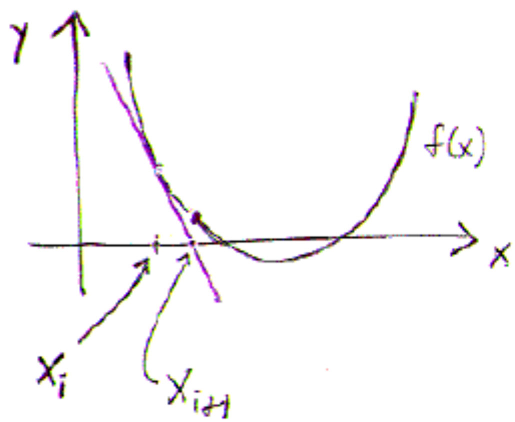
$\sqrt{\text{square roots}}$

Who remembers the "schoolboy" method ?
Fortunately, calculators became cheap before
I managed to learn it.

Computers use another technique:

Newton's Method

(also used in IBM 360 for division)



- finds x s.t. $f(x) = 0$
- improves guess x_i by interpolation $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- e.g. $f(x) = x^2 - a$ (find $\sqrt{a}$)
- e.g. $f(x) = \frac{1}{x} - a$ (find $\frac{1}{a}$)

Newton's Method – Division

deriving the recurrence:

$$f(x) = \frac{1}{x} - a \qquad f'(x) = -\frac{1}{x^2}$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

$$= x_i - \frac{\frac{1}{x_i} - a}{-\left(\frac{1}{x_i}\right)^2}$$

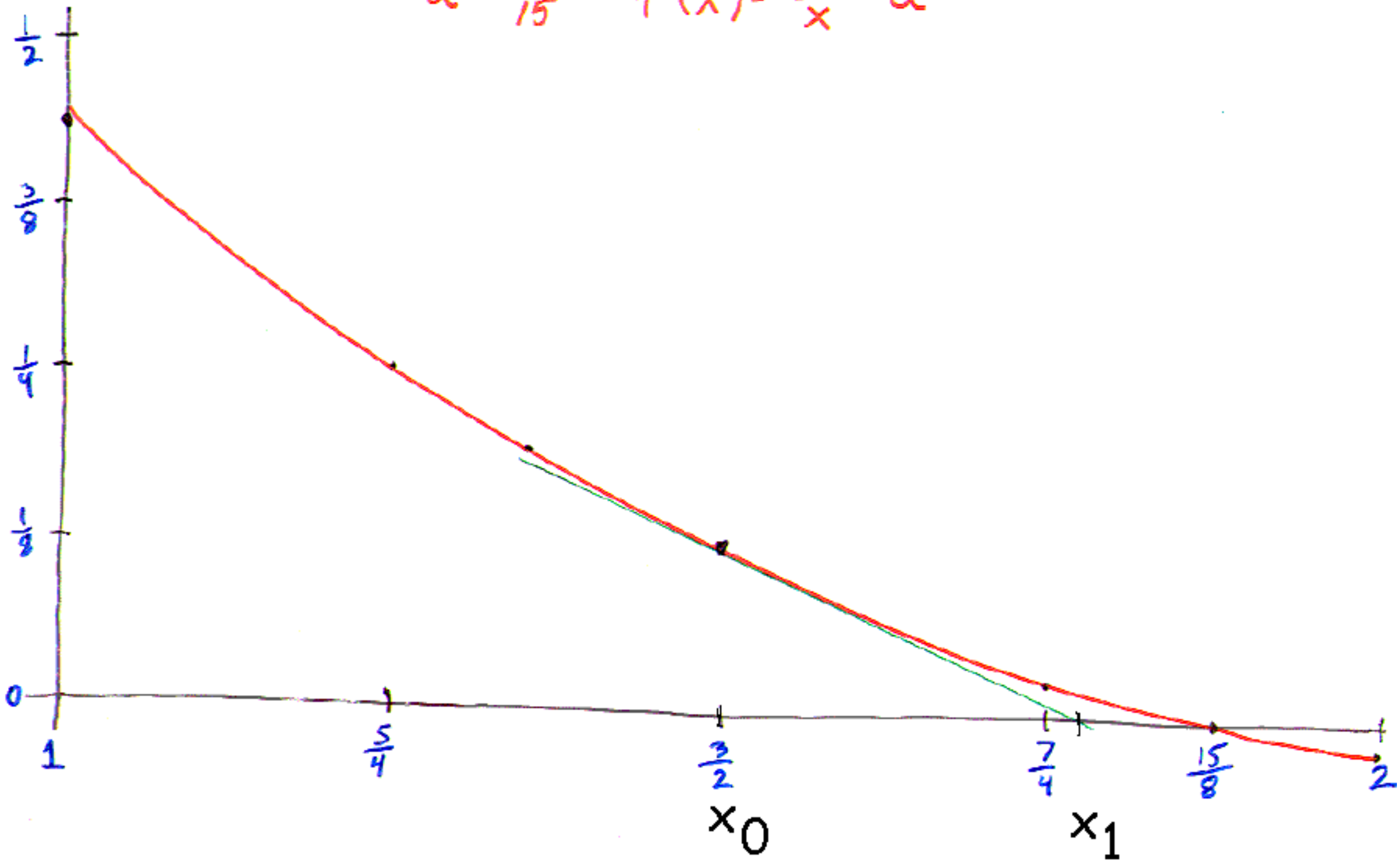
$$= 2x_i - x_i^2 a$$

Note: Some functions f s.t. $f(\frac{1}{a}) = 0$ don't work out. E.g.:

$$f(x) = x - \frac{1}{a} \Rightarrow x_{i+1} = \frac{1}{a}$$

$$f(x) = xa - 1 \Rightarrow x_{i+1} = \frac{1}{a}$$

$a = \frac{8}{15} \quad f(x) = \frac{1}{x} - a$



Error Analysis – Division

Assume $\frac{1}{2} < a < 1$ so that $1 < \frac{1}{a} < 2$.

Let $\varepsilon_i = x_i - \frac{1}{a}$.

$\uparrow$ error after i iterations

Then $x_i = \frac{1}{a} + \varepsilon_i$, and

$$
\begin{aligned}
x_{i+1} &= 2x_i - a \cdot x_i^2 \\
&= 2\cdot\left(\tfrac{1}{a} + \varepsilon_i\right) - a \cdot \left(\tfrac{1}{a} + \varepsilon_i\right)^2 \\
&= \tfrac{1}{a} - a\varepsilon_i^2,
\end{aligned}
$$

so $\varepsilon_{i+1} = -a\varepsilon_i^2$.

Notice that $\varepsilon_{i+1} < 0$, and $|\varepsilon_{i+1}| < |\varepsilon_i|^2$ (for $0 < a < 1$).

Convergence – Division

$\frac{1}{2} < a < 1 \qquad 1 < \frac{1}{a} < 2$

pick $x_0 = \frac{3}{2}$ (initial guess)

$\Rightarrow |\varepsilon_0| < \frac{1}{2}$

$\Rightarrow |\varepsilon_1| < \left(\frac{1}{2}\right)^2, \; |\varepsilon_2| < \left(\left(\frac{1}{2}\right)^2\right)^2$

$|\varepsilon_i| < \frac{1}{2^{2^i}}$

$\therefore$ After i iterations, x_i is correct to 2^i bits

$x_i = 1.\underbrace{01011011}_{2^i}101\cdots$

except for possible error smaller in magnitude than max. contribution of all bits beyond 2^i.

(Adding $|\varepsilon_i|$ to x_i might change some of the leading 2^i bits, though.)

Total time?
$O(\log^2 n)$

Newton's Method – Square Root

$$f(x) = x^2 - a \qquad \tfrac{1}{2} < a < 1, \quad \tfrac{1}{2} < \sqrt{a} < 1$$

derivation: $$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad x_0 = \frac{3}{4}$$

$$= x_i - \frac{x_i^2 - a}{2x_i}$$

$$= \frac{1}{2}x_i + \frac{a}{2x_i}$$

error analysis: Let $x_i = \sqrt{a} + \varepsilon_i$. $|\varepsilon_0| < \frac{1}{4}$

$$x_{i+1} = \frac{1}{2}(\sqrt{a} + \varepsilon_i) + \frac{a}{2(\sqrt{a} + \varepsilon_i)}$$

$$= \frac{1}{2}(\sqrt{a} + \varepsilon_i) + \frac{\sqrt{a}}{2}\left(\frac{1}{1 + \frac{\varepsilon_i}{\sqrt{a}}}\right)$$

$$= \frac{1}{2}(\sqrt{a} + \varepsilon_i) + \frac{\sqrt{a}}{2}\left(1 - \frac{\varepsilon_i}{\sqrt{a}} + \frac{\varepsilon_i^2}{a} - \cdots\right)$$

(provided that $\left|\frac{\varepsilon_i}{\sqrt{a}}\right| < 1$)

$$= \sqrt{a} + \frac{\varepsilon_i^2}{2\sqrt{a}} - \cdots$$

$$\therefore\ |\varepsilon_{i+1}| < |\varepsilon_i|^2$$

Total time? $O(\log^3 n)$