Floating-Point Arithmetic and Beyond

ENS Lyon – Nov. 2020

Jean-Michel Muller

CNRS - Laboratoire LIP

`http://perso.ens-lyon.fr/jean-michel.muller/`

Floating-Point Arithmetic

- by far the most frequent solution for manipulating real numbers in computers;
- comes from the "scientific notation" used for 3 centuries by the scientific community;
- roughly speaking:

$$x = \pm m_x \times \beta^{e_x},$$

where

  - β is the base or radix (in general 2 or 10 but more exotic things have existed)
  - $m_x \in \{0\} \cup [1, \beta)$ is the significand (often called mantissa);
  - $e_x \in \mathbb{Z}$ is the exponent,

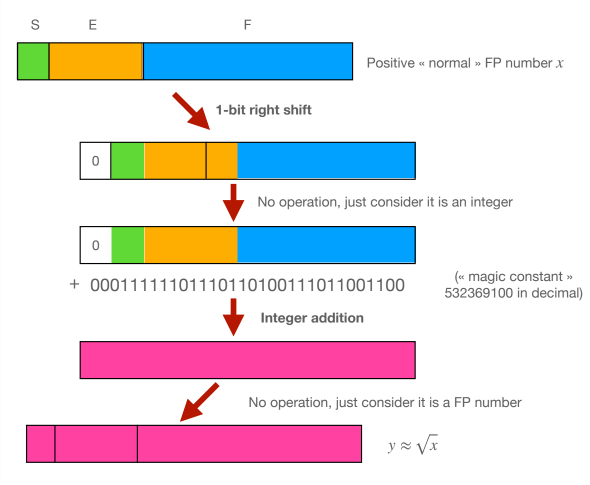
. . . but much more will be said later on.

Floating-Point Arithmetic

Sometimes a bad reputation. . . for bad reasons:

- intrinsically approximate. . .
  - but most physical data is approximate;
  - but most numerical problems we deal with have no closed-form solution;
  - and in a subtle way (correct rounding), FP arithmetic is exact.
- part of the literature comes from times when it was poorly specified;
- too often, viewed as a set of dirty tricks for geeks:. . . but there are such tricks (see next slide).

A very odd trick



A similar trick first appears in The game Quake III Arena

We wish to show that

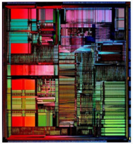
- it is a well specified arithmetic, on which one can build trustable calculations;
- one can formally prove useful properties and build efficient algorithms on FP arithmetic;
- one can prove useful mathematical properties using FP arithmetic.

Desirable properties of an arithmetic system

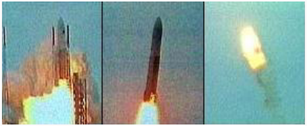
- Speed: tomorrow's weather must be computed in less than 24 hours;
- Accuracy;
- Range: represent big and tiny numbers as well;
- "Size": silicon area for hardware, memory consumption;
- Power consumption;
- Portability: the programs we write on a given system must run on different systems without requiring huge modifications;
- Easiness of implementation and use: If a given arithmetic is too arcane, nobody will use it.

Weird behaviours

- 1994, Pentium 1 division bug: 8391667/12582905 gave $0.666869\cdots$ instead of $0.666910\cdots$;



- 1996, maiden flight ... and flop of the Ariane 5 European rocket: arithmetic overflow



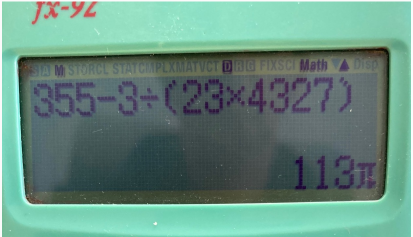
- November 1998, USS Yorktown warship, somebody erroneously entered a "zero" on a keyboard → division by 0 → series of errors → the propulsion system stopped.

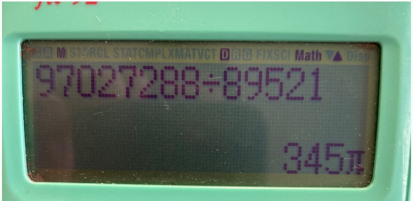
Weird behaviours

- Maple version 6.0 (2000). Enter 214748364810, you get 10. Note that $2147483648 = 2^{31}$;
- Excel'2007 (first releases), compute $65535 - 2^{-37}$, you get 100000;
- if you have a Casio FX 83-GT Plus or a FX-92 pocket calculator, compute $11^6/13$, you will get

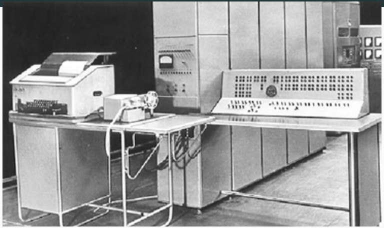
$$\frac{156158413}{3600}\pi$$

You want more ?





Other strange things



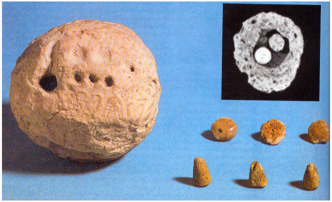
- Setun Computer, Moscow University, 1958. 50 copies;
- Base 3 and digits -1, 0 and 1. Numbers represented using 18 "trits";
- idea: base β, n digits $\rightarrow$ "Cost": $\beta \times n$;
- minimize $\beta \times n$ with $\beta^n \geq M$: as soon as

$$M \geq e^{\frac{5}{(2/\ln(2))-(3/\ln(3))}} \approx 1.09 \times 10^{14}$$

  the optimal β is 3

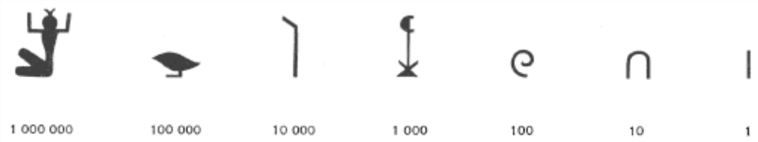
Early representation of the integers

- "one, two, three, many"...
- "unary" representations

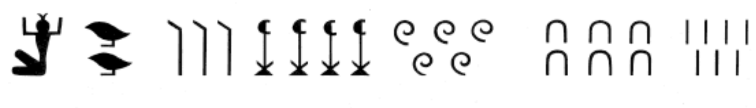


- systems that make it possible to represent the integers, but are not convenient for computing. Egyptian example:

Egyptian example



- number 1234567:



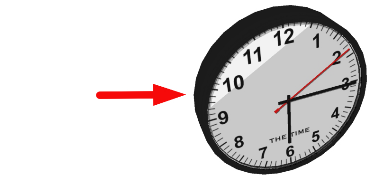
- needs infinitely many symbols (one for each power of 10);
- solution: the position of a symbol in the representation of a number indicates of which power of the base it is a multiple.

→ positional number systems

Positional number systems

Base 60: Babylon (≈ 4000 years ago)

$$1 + \frac{24}{60} + \frac{51}{60^2} + \frac{10}{60^3} = 1.41421296\cdots$$
$$\sqrt{2} = 1.41421356\cdots$$



- Base 20: Mayas;
- Base 10: India;
- Computer science: base 2 ou 10.

Our positional system makes human computing easy. Is it well adapted to automated computing?

Positional number systems

Base (or radix) $\beta \geq 2$, n digits taken in the digit set $\mathcal{D} = \{a, a+1, a+2, \ldots, b\}$, with $a \leq 0$ and $b > 0$. The digit chain

$$m_{n-1} m_{n-2} \cdots m_1 m_0$$

represents the integer

$$m_{n-1}\beta^{n-1} + m_{n-2}\beta^{n-2} + \cdots + m_1\beta + m_0 = \sum_{i=0}^{n-1} m_i \beta^i.$$

Theorem 1

- *if $b - a + 1 \geq \beta$ then all integers between $a \cdot \frac{\beta^n - 1}{\beta - 1}$ and $b \cdot \frac{\beta^n - 1}{\beta - 1}$ can be represented (i.e., by allowing unbounded n, all integers if $a < 0$, and all positive integers if $a = 0$);*
- *if $b - a + 1 = \beta$ the representation is unique;*
- *if $b - a + 1 > \beta$ some numbers have several representations: the system is redundant.*

Positional number systems

Define $I_n = [\![a \cdot \frac{\beta^n - 1}{\beta - 1}, b \cdot \frac{\beta^n - 1}{\beta - 1}]\!]$.

Goal : all elements of I_n are representable.

<u>Proof</u> Induction on n.

- $n = 1$ straightforward ($I_n = [\![a, b]\!]$)
- assume the property holds for n.

Consider $J_k^n = k \cdot \beta^n + I_n$ for $k \in \mathcal{D}$

$J_k^n = \{$ numbers representable with k as leftmost digit $\}$

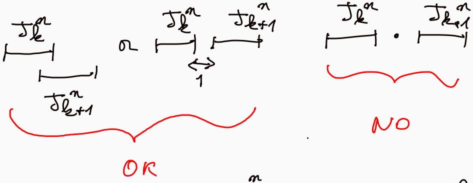
We need to show : $\bigcup_{k \in \mathcal{D}} J_k^n = I_{n+1}$.

$$J_k^n = [\![l_k^n, r_k^n]\!] \text{ with } \begin{cases} l_k^n = k\beta^n + a \cdot \frac{\beta^n - 1}{\beta - 1} \\ r_k^n = k\beta^n + b \cdot \frac{\beta^n - 1}{\beta - 1} \end{cases}$$

$l_a^n = a \cdot \frac{\beta^{n+1} - 1}{\beta - 1}$ and $r_b^n = b \cdot \frac{\beta^{n+1} - 1}{\beta - 1}$.

$\longrightarrow I_{n+1} = [\![l_a^n, r_b^n]\!]$

we need to show: no "holes" between consecutive J_k^n's.



$$r_k^n - l_{k+1}^n = k\cdot\beta^n + b\,\frac{\beta^n-1}{\beta-1} - (k+1)\beta^n - a\,\frac{\beta^n-1}{\beta-1}$$

$$= -\beta^n + (b-a)\,\frac{\beta^n-1}{\beta-1}$$

① $b-a \geq \beta-1 \Rightarrow r_k^n - l_{k+1}^n \geq -1$ OK

② if $b-a = \beta-1$ the sets do not overlap

→ unique choice of the n^{th} digit

Positional number systems: particular cases

- $a = 0$ and $b = \beta - 1$: conventional base-β representation;
- $a = 0$ and $b = \beta$: carry-save representations;
- $a = -r$ and $b = +r$, with $r \geq \lfloor \beta/2 \rfloor$: signed-digit representations.

The redundant representations (e.g., carry-save, or signed-digit with $2r + 1 > \beta$) allow for very fast, parallel additions.

Cauchy (1840): base 10, $\mathcal{D} = [\![-5, +5]\!]$. Goal: limit carry propagations in multiplications.

Avizienis (1961): parallel addition algorithm for redundant signed-digit systems.

Exercise

All this is easily generalizable to fractional representations:

$$x_n x_{n-1} x_{n-2} \dots x_0 . x_{-1} x_{-2} \dots x_{-m} = \sum_{i=-m}^{n} x_i \beta^i.$$

Consider the system $\beta = 3$ and digit-set $\{-1, 0, 1\}$ and the "truncation at position m" function:

$$x_n \dots x_0 . x_{-1} x_{-2} \dots x_{-m} x_{-m-1} x_{-m-2} \dots \rightarrow x_n \dots x_0 . x_{-1} x_{-2} \dots x_{-m}$$

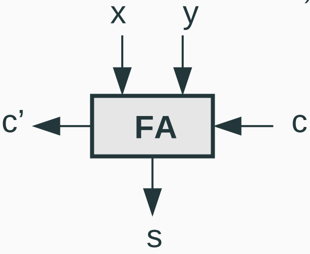
show that in this system, truncating at position m is equivalent to rounding to (the ? a ? discuss) nearest multiple of 3^{-m}. A number system with that property is called an RN-code.

Just a glance on algorithms used in circuits: binary (nonredundant) addition (Base 2, digits 0 and 1)

Elementary addition cell: 3 entries x, y and c, and two outputs s and c', equal to 0 or 1, that satisfy

$$2c' + s = x + y + c.$$

(here "+" is the addition, not the boolean "or").

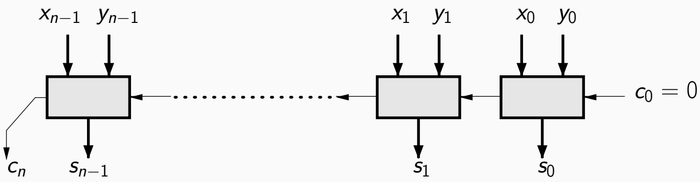


FA means "Full Adder Cell".

The pair (c', s) is the binary representation of $x + y + c$.
Easily implemented with a few logic gates.

Just a glance on algorithms used in circuits: binary addition

- Input: $(x_{n-1}x_{n-2}\cdots x_0)$ and $(y_{n-1}y_{n-2}\cdots y_0)$ represented in binary.
- output: $(s_{n-1}s_{n-2}\cdots s_0)$ in binary too.
- c_0 and c_n: carry-in and carry-out.



The Carry-Ripple adder.

- Intrinsically sequential algorithm. The delay grows linearly with the number of digits;
- can be improved: we give a simple example now.

Conditional sum addition

- Many algorithms/architectures for fast addition: we just give a simple one for illustration;
- more efficient algorithms in Knowles' paper *A family of adders*.

```
  1001011001011011110001011110001101010010110100101
+ 0110100100111001101001011110010101001011110110110
```

Addition of two $2n$-bit numbers.

Conditional sum addition

10010110010110111110001011	110001101010010110100101
011010010011100110100101	1110010101001011110110110

Each 2n-bit operand split into two n-bit operands.

Conditional sum addition

1001011001011011110001011

011010010011100110100101

1100011010100101101000101

1110010101001011110110110

1001011001011011110001011

011010010011100110100101

Conditional sum addition

```
                                0
  1001011001011011110001011
+ 0110100100111001101001001
```

```
  1100011010100101101001 01
+ 111001010100101111011 0110
```

```
                                 1
  1001011001011011110001011
+ 0110100100111001101001001
```

Conditional sum addition

0
1001011001011011110001011

\+ 011010010011100110100101
yyyyyyyyyyyyyyyyyyyyyyyyy

110001101010010110100101

\+ 1110010101001011110110110
1xxxxxxxxxxxxxxxxxxxxxxxxxxx

1
1001011001011011110001011

\+ 011010010011100110100101
zzzzzzzzzzzzzzzzzzzzzzzzz

Conditional sum addition

```
                          0
  10010110010110111110001011        110001101010010110100101

+ 011010010011100110100101        + 1110010101001011110110110
   yyyyyyyyyyyyyyyyyyyyyyyyy        1xxxxxxxxxxxxxxxxxxxxxxxxxxx
                                  /
                                 /
                             <--
                           1

  10010110010110111110001011

+ 011010010011100110100101
   zzzzzzzzzzzzzzzzzzzzzzzzz
```

When done recursively...

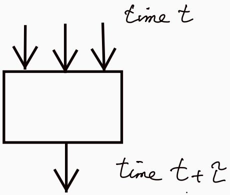
- T_n delay of n-bit addition,

$$T_n = T_{n/2} + C;$$

- recursive implementation: $\log_2(n)$ steps;
- can we do better?

Winograd's theorem

r-circuit, made of *r*-elements. An *r*-element is a "logic gate" with at most r binary inputs, 1 binary output. It generates its output in a delay τ that does not depend on the inputs or the computed function.



A boolean function $f : \{0,1\}^m \to \{0,1\}$ depends on all its entries if $\forall i \in [\![1,n]\!]$ there exists $(x_1, x_2, \ldots x_m)$ s.t.

$$f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m) \neq f(x_1, \ldots, x_{i-1}, \overline{x_i}, x_{i+1}, \ldots, x_m).$$

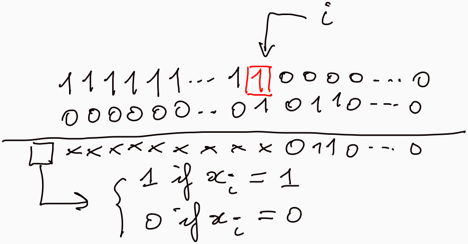
where $\overline{x_i} = 1 - x_i$.

Winograd's theorem

Theorem 2

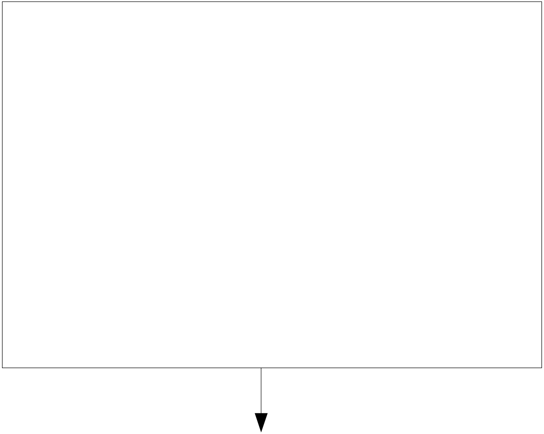
If $f : \{0,1\}^m \to \{0,1\}$ *depends on all its entries then an* r*-circuit requires a delay* $\geq \lceil \log_r(m) \rceil \cdot \tau$ *to compute it.*

Remark: Addition depends on all its entries.



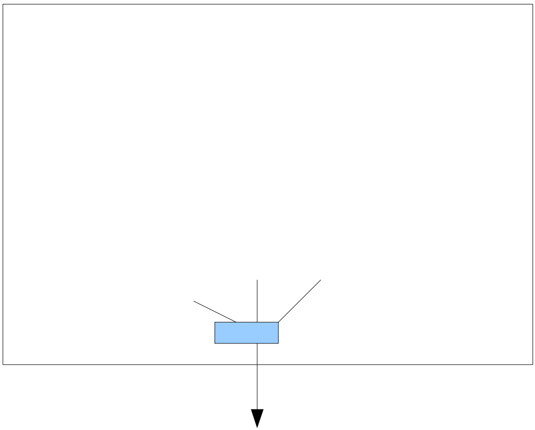
Sketch of the proof

Time 0: output of the result. It necessarily comes from an r-element.



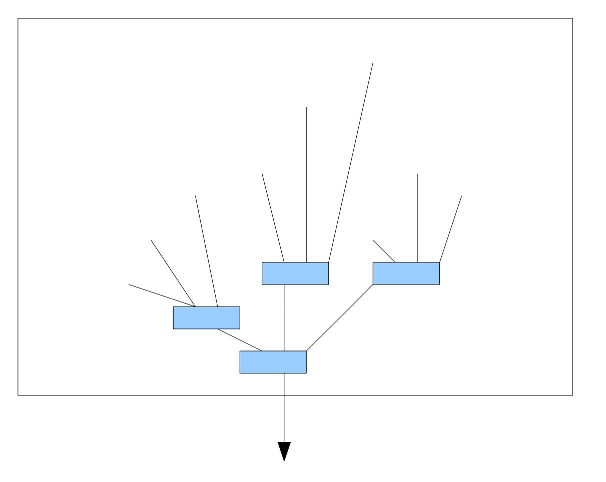
Sketch of the proof

Time $-\tau$: at most r terms can have an influence on the final result



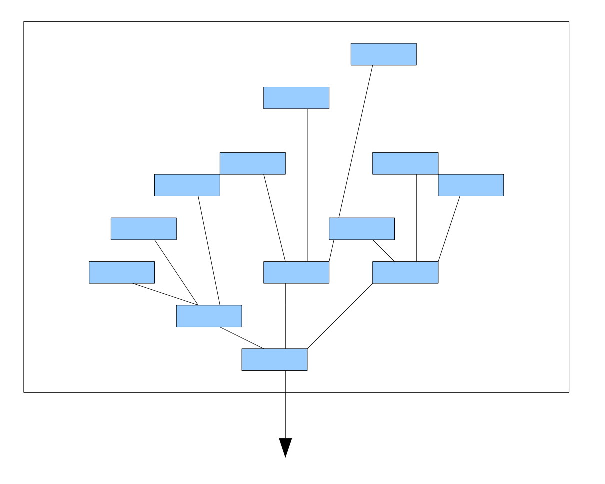
Sketch of the proof

Time -2τ, at most r^2 terms can have an influence on the final result



Sketch of the proof

Time -3τ, at most r^3 terms can have an influence on the final result



Sketch of the proof

- Time $-k\tau$, at most r^k terms can have an influence on the final result. We need $r^k \geq m$.

$$\rightarrow \text{ time } \geq \lceil \log_r(m) \rceil \cdot \tau.$$

- Addition: the $2n$ bits that represent the inputs of the addition of two n-bit numbers can influence the leftmost digit of the result

$\rightarrow$ the delay is therefore at least $t = k\tau$, where $r^k \geq 2n$,

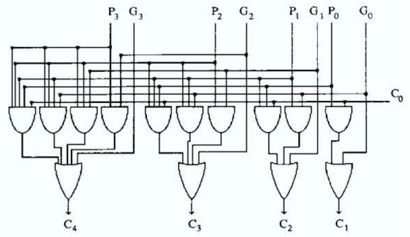
- gives

$$t \geq \tau \times \log_r(2n).$$

Faster than logarithmic-time addition ?

At least one of the conditions of Winograd's theorem must be unsatisfied:

- ~~Computation with r-elements:~~ allow logic gates with unbounded number of inputs → carry-lookahead adder. Unrealistic for large *n*;



- ~~at least one digit of the result does not depend on all the entries:~~ use a redundant number system.

Carry-save arithmetic

- base β and digit-set $\{0, 1, \ldots, \beta\}$;
- redundant: the number β can be written 10 or 0β;
- widely used in base 2 where each digit $d_i \in \{0, 1, 2\}$ is represented by two bits $d_i^{(1)}, d_i^{(2)} \in \{0, 1\}$ s.t. $d_i = d_i^{(1)} + d_i^{(2)}$.

Major interest: very fast addition

$$\text{CarrySave} + \text{ConventionalBinary} \rightarrow \text{CarrySave}.$$

(used for instance for building multipliers)

Carry-save addition

- $a = a_{n-1}a_{n-2}\dots a_0 = \sum_{i=0}^{n-1} a_i$ in carry-save arithmetic ($a_i = a_i^{(1)} + a_i^{(2)} \in \{0, 1, 2\}$);
- $b = b_{n-1}b_{n-2}\dots b_0 = \sum_{i=0}^{n-1} b_i$ in conventional binary ($b_i \in \{0, 1\}$)

We have

$$a_i^{(1)} + a_i^{(2)} + b_i \in \{0, 1, 2, 3\}$$

→ it can be written $2s_{i+1}^{(2)} + s_i^{(1)}$ with $s_{i+1}^{(2)}, s_i^{(1)} = 0$ or 1.

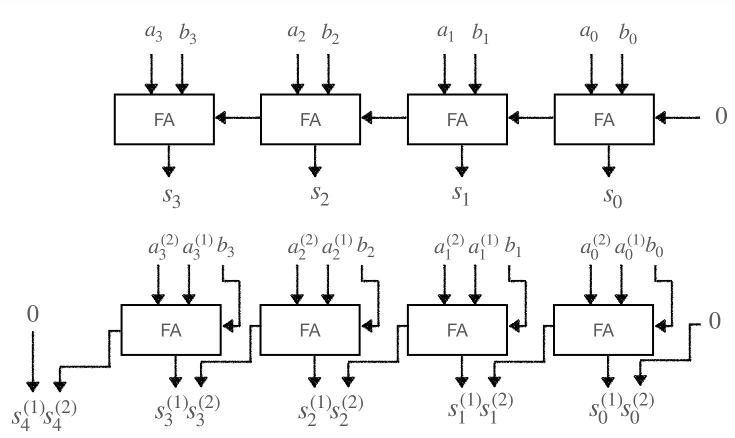
→ with the convention $s_n^{(1)} = s_0^{(2)} = 0$, and denoting $s_i = s_i^{(2)} + s_i^{(1)}$, the carry-save number

$$s_n s_{n-1} \dots s_0 = \sum_{i=0}^{n} (s_i^{(2)} + s_i^{(1)}) \cdot 2^i = \sum_{i=0}^{n-1} (2s_{i+1}^{(2)} + s_i^{(1)}) \cdot 2^i$$

represents $a + b$.

Carry-save addition

The sequential, “ripple-carry” adder, and the carry-save adder:



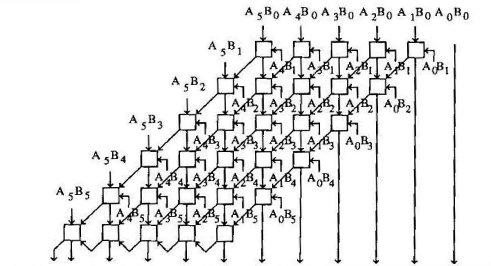
$\rightarrow$ delay of an n-bit CS addition = delay of a 1-bit sequential addition!

Carry-save addition

- conversion carry save → conventional representation: a conventional addition;

→ interesting only if the amount of calculation done in carry-save arithmetic is big in front of an addition;

- typical example: multiplication



Exercise

- how would you add two carry-save numbers ?
- Using carry-save arithmetic and the associativity of addition, show that we can multiply two n-bit numbers in time proportional to $\log(n)$.