In the 1960's, the term "Op Art" was coined to describe the work of a growing group of abstract painters. This movement was led by Vasarely.

Preliminary version
Do not distribute

Range:
From logic gate to combinatorial arithmetic operators.

Pedagogy:
Experiment, understand, improve.
Complement to the lecture notes.

Method:
Animation of the lecture note figures
Synthesis, simulation, diagrams, algorithms

The entire course is printable.

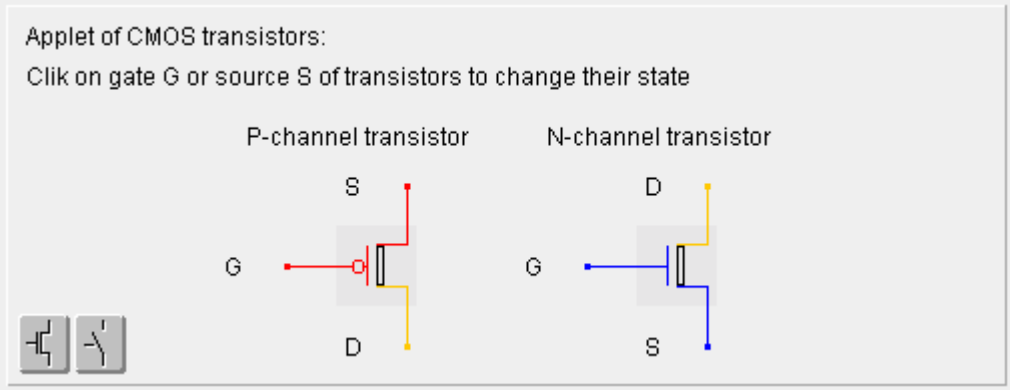
"FA" function in CMOS

CMOS technology

CMOS technology (Complementary Metal Oxide Semiconductor) offers two types of transistors called "N-channel" and "P-channel". CMOS is currently the dominant technology, at least for digital circuits. Its main advantage with respect to other technologies is remarkable low power consumption. Indeed the CMOS circuits exhibit a static current (or quiescent current) practically negligible.

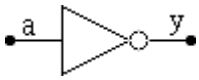
In the figures below :

- A logic '1' is represented by the supply voltage Vdd (current values for Vdd are +5V or +3,3V or +2,8V) and is colored in **red**.
- A logic '0' is connected to the ground voltage, or GND, is colored in **blue**.
- A connection neither to Vdd nor to GND is in **yellow**.



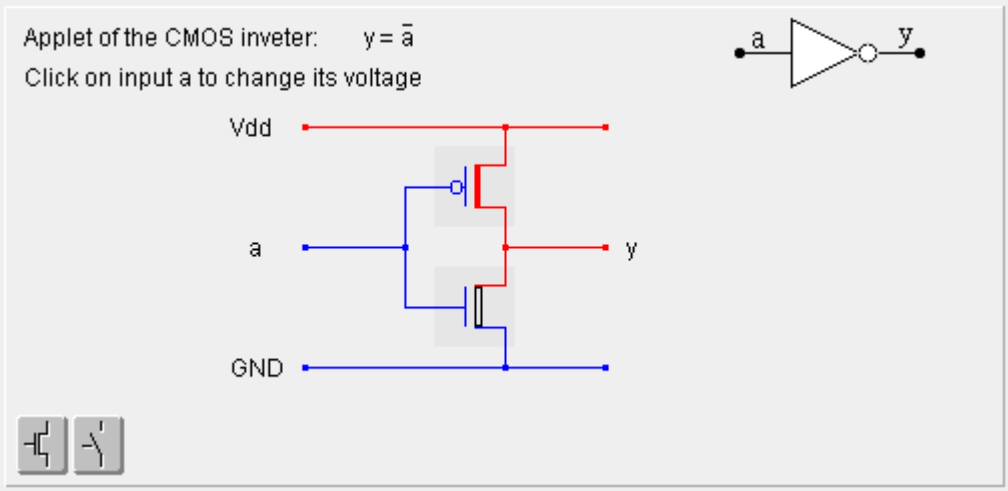
The N-channel transistor conducts when it gate is '1' and the P-channel transistor conducts when its gate is '0' The keys  change the transistor's outline.

CMOS inverter



The CMOS inverter is the most popular gate. It is composed of a N-channel and a P-channel transistors connected through their drain. The figure below illustrates its behavior.

The colors conventions are still red for logic '1' and blue for logic'0' An input voltage in between causes a (mild) short-circuit by maintaining both transistors in conduction. Such a voltage is colored in **green**. Click on input "a" to pass from '0' to short (**green**), then to '1', then to short (**green**) again then back to '0' and so on.

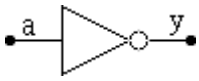


Please notice that when the input is '0' or '1', only one transistor conducts.

Delay and

We just have seen that the inverter dissipates no energy except during commutation.

dissipation of the CMOS inverter



Indeed if the input is '0' or '1' there is no conduction path between the power supply Vdd and the ground GND. In normal conditions, the short circuit current, (unavoidable during input commutation) lasts a very short time, typically a few picosecondes.

The contribution of the parasitic capacitances charge or discharge is much more significant. The transistor gate G forms a capacitance. Anyway this capacitance is necessary to the field effect transistor working. Typically an input capacitance *Cg* may be around 10 fF. If at time t_1, this capacitance is connected to Vdd it is charged (charge $Q = Cg * Vdd$). If later on, at time t_2, the input is connected to GND the capacitance is discharged. This discharge causes a current in the gate $I = dQ/dt = (Cg * Vdd)/(t_2 - t_1)$.

Although the gate charge/discharge current is

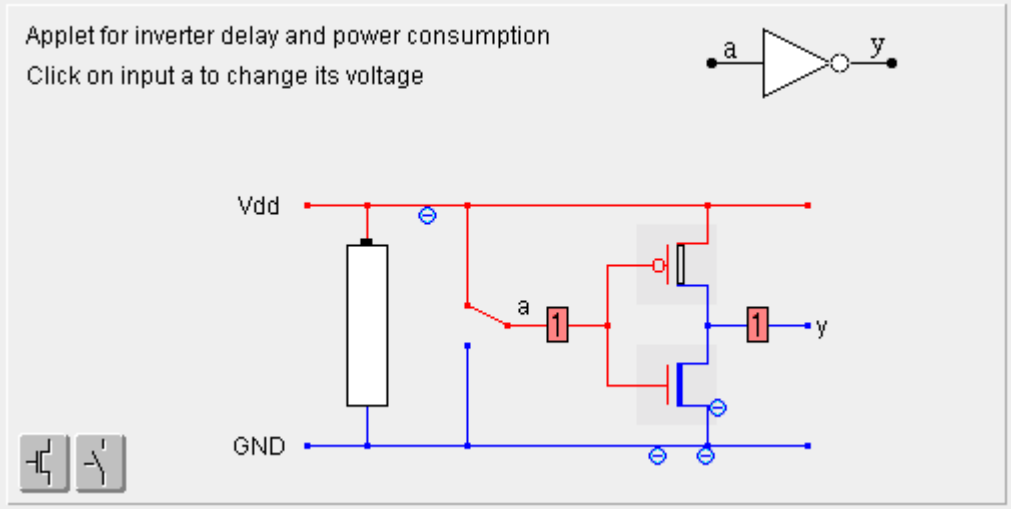
Let us take an example :

- A modern microprocessor may contain 50 million transistors, meaning about 10 million gates. For each clock cycle, about 1% of those gate commutates.
- Clock frequency may reach 1 000 MHz (cycle time 1 ns) with a power supply Vdd = 3.3V.
- The wires connecting the gates most of the time exhibit a parasitic capacitance *Cw* much larger than the gate input capacitance *Cg*. Each a wire commutates, all the attached capacitances must be either charged or discharged. : $Ctotal=Cg+Cw$.
- An average wire capacitance may be around *1 pF*

It is rather difficult to estimate the current due to the short-circuits, it is generally small. On the contrary the current due to the commutation activity is important : $I = (active\ gates) * (Ctotal*Vdd) / dt = (1\% * 1{,}000{,}000) * (1pF * 3.3V) /2ns = 16A$

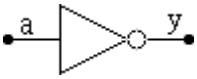
Finally the quiescent current due to the transistors leaks is quite small (for a conventional circuitry). A static memory SRAM of 2K*8 bits in CMOS let leak *1 µA* when not active.

- The figure below shows the current, or electrons ⊖ flow in the CMOS inverter. Whenever the input stays to '1' or to '0', either the N-channel of the P-channel transistor is blocked and there is no current
- When the input changes, the grid of the two transistors must be charged or discharged. This is illustrated by the flow of an electron ⊖ (with a negative charge) coming from GND or going to Vss.
- During the input change, the voltage passes through values that let both transistors conduct, usually during a very short time. This short-circuit current is illustrated by an electron flowing directly from GND to Vss.
- Finally the output is charged or discharged through the transistors. The output capacitance stores two electrons ⊖.

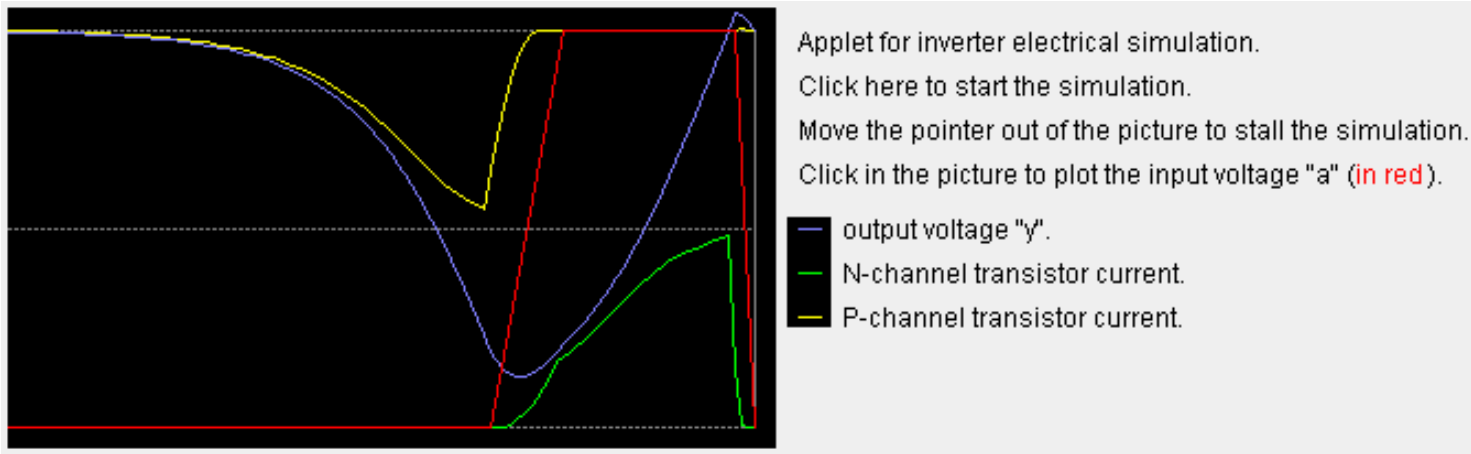


The power dissipated by a conventional CMOS circuit is consequently directly proportional to the clock frequency.

Electrical simulation of the CMOS inverter



By clicking or dragging the mouse inside the chronogram below, you control the input "a" voltage (plotted in **red** on the chronogram). The output voltage "y" is then computed (plotted in **blue**). The current flowing through the N-channel transistor if drawn in **green** and the current through the P-channel transistor is in **yellow**. To suspend the applet and freeze the plot, just get the pointer out of the picture.



Basic NOR and NAND gates

We are now studying some basic CMOS logic gate: a 2-input NOR, a 2-input NAND and finally a full adder cell.

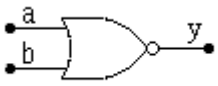
Colors conventions: They are the same as the inverter's one. Connections to Vdd (logic '1') are drawn in **red**, connections to GND (logic '0') are in **blue**, and connections to both Vdd **and** GND are in **green**. Finally connections neither to Vdd nor to GND (floating) are in **yellow**. The two last colors have no logic image.

- Click close to an input changes its value and consequently the state of the transistors attached to this input.
- The line in the truth table that corresponds to the input combination is highlighted.
- Clicking in the truth table changes the input according to the highlighted line.

- Clicking the top of the table highlight the next line.

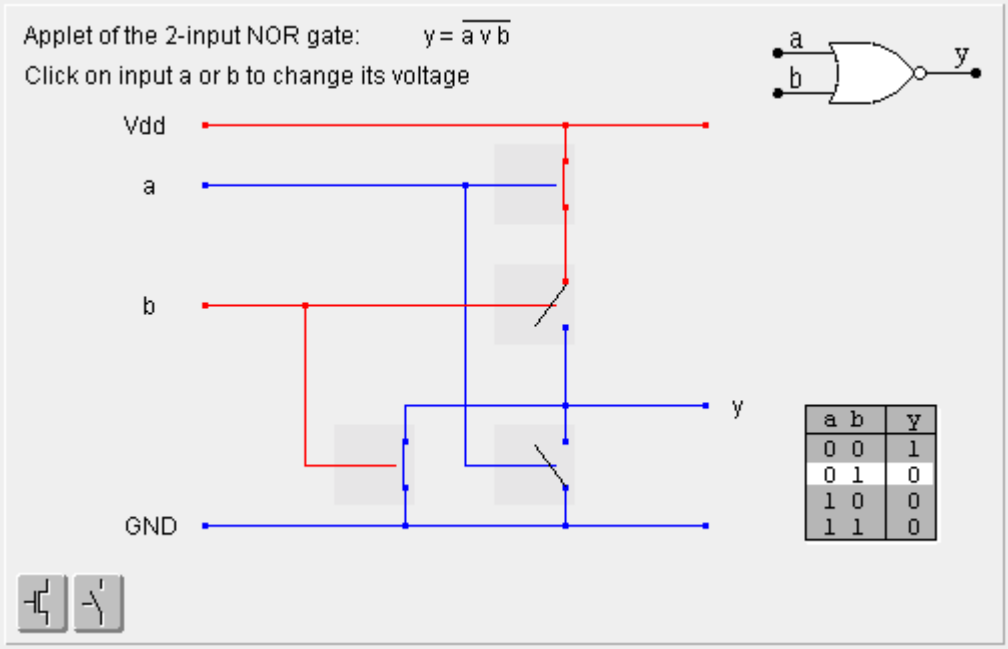
To simplify the applets, only logical '1' and '0' are allowed for the inputs. Therefore it is not possible to input a value causing a short-circuit between Vdd and GND

Two-input NOR gate

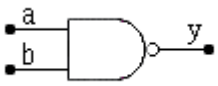


The two-input "NOR" gate is one of the simplest gates to illustrate the term *complementary* : the P-channel transistors are connected in serial while the N-channel transistors are in parallel. The P-channel and N-channel network are complementary.

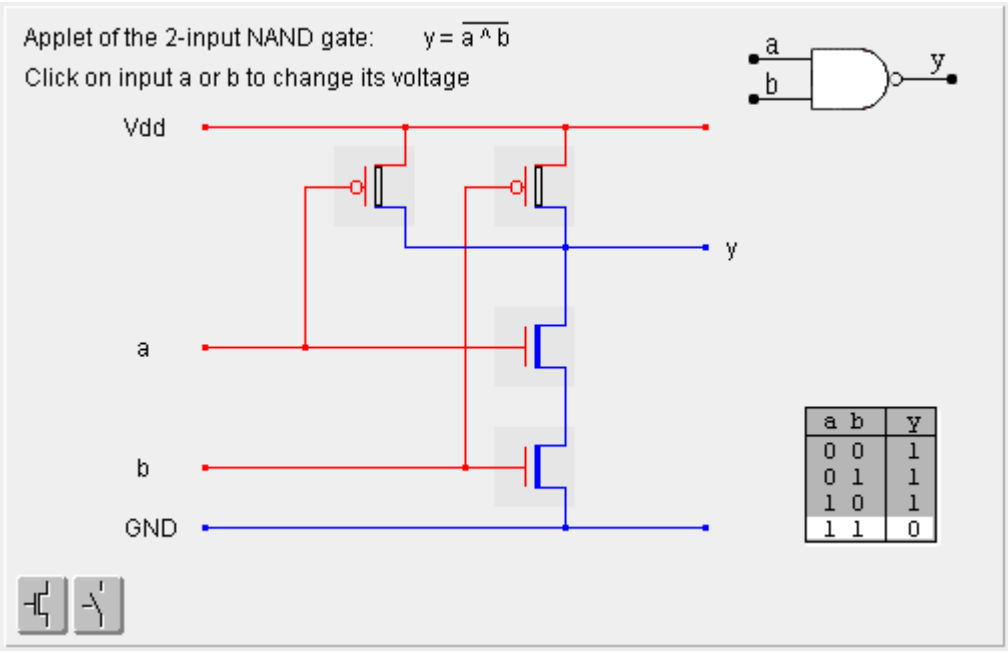
Notice that when none of the two P-channel transistors conduct, their common connection is floating (**yellow**). This is a "non logic" value, however, it does not cause trouble since it is not connected to any transistor gate.



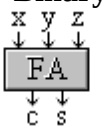
The 2-input NAND gate



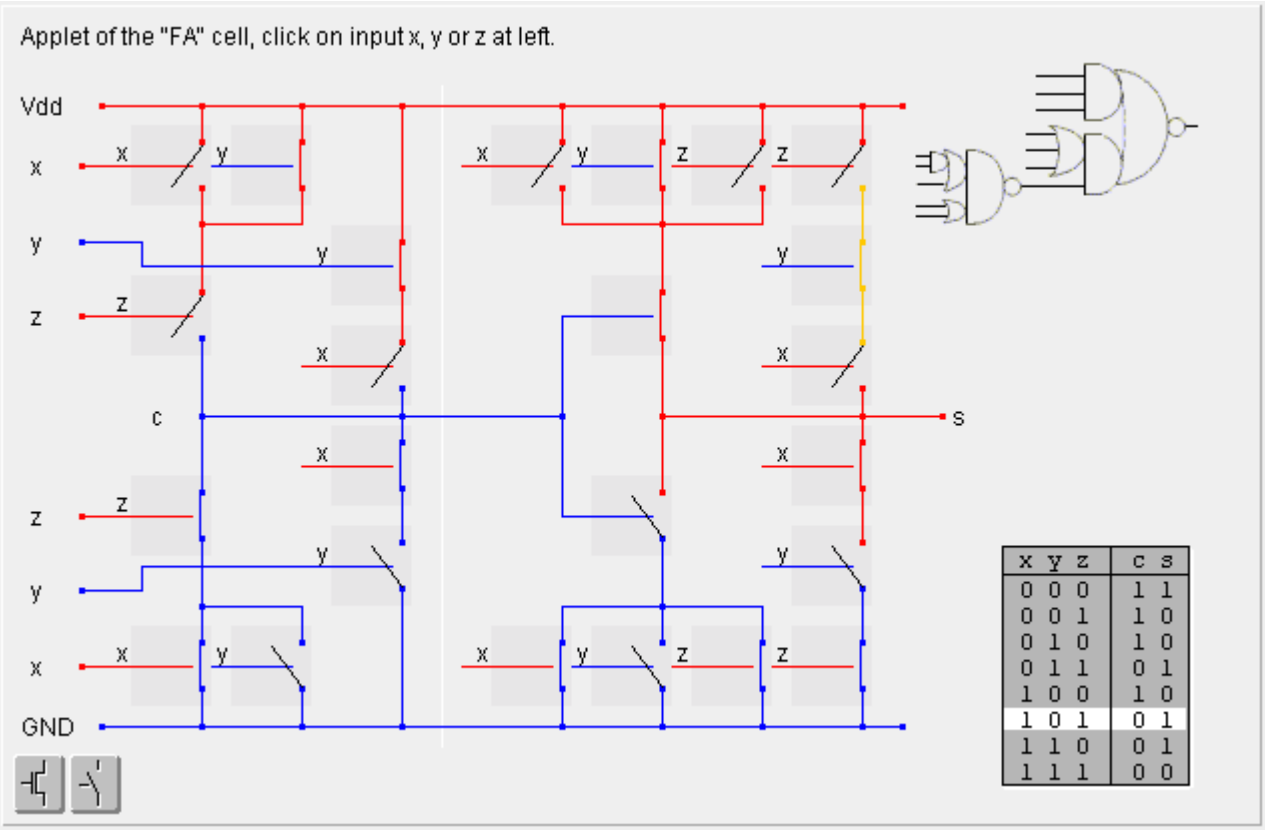
In the two-input NAND gate, the P-channel transistors are connected in parallel while the two N-channel transistors are in serial.



Binary adder



The "Full Adder" cell (FA) is made of two connected complex gates. It realizes an arithmetic equality: the weighted sum of the three inputs "x", "y" et "z" is always equal to the weighted sum of the two outputs "c" et "s", in other words " x + y + z = 2*c + s ". This property can easily be checked thanks to the cell truth table.

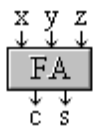


x	y	z	c	s
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0

The P-channel transistor network is symmetrical to the N-channel network. A circuit with this property is called "mirror". All the adders exhibit the property that follows from an arithmetic link between the logic and arithmetic complements. Finally the two circuit outputs are inverted. This follows from an electric property of the CMOS technology, which allows easily non-increasing logical functions only.

Adders

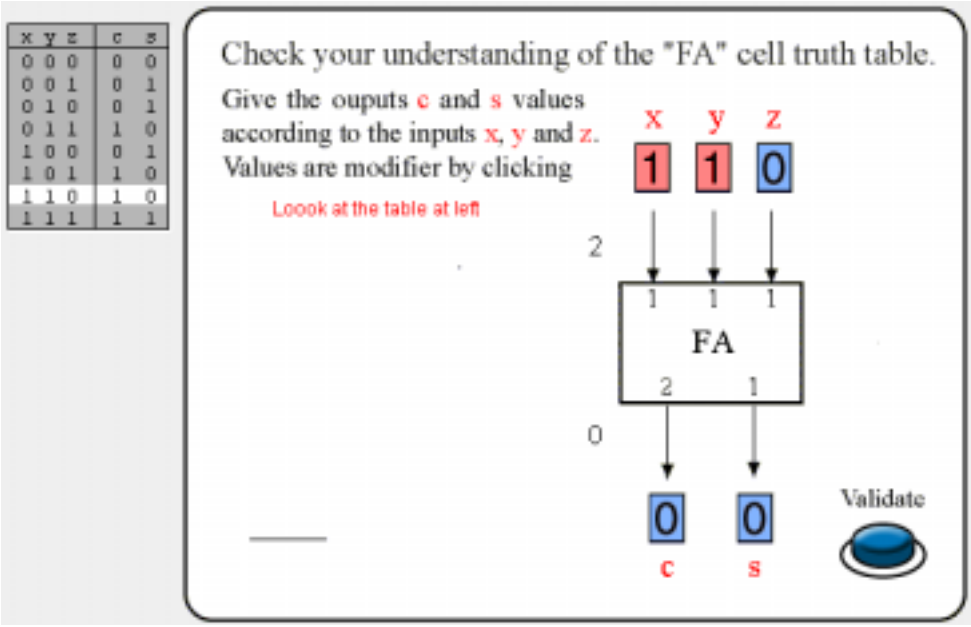
"FA" cell



In the "FA" cell, the weighted sum of the output bits equals the weighted sum of the output bits, i.e. " x + y + z = 2*c + s ". The three input bits share the same weight. Let it be "1". The output bit "s" has also the same weight, while the output bit "c" weight is double (2).

The "FA" cell conserves the sum just like the node conserves the electric current in the "Kirchoff's current law".

The "FA" cell is also called "3 ⇒2 compressor" since it reduces the bit number from 3 to 2 while preserving the *numerical value*.



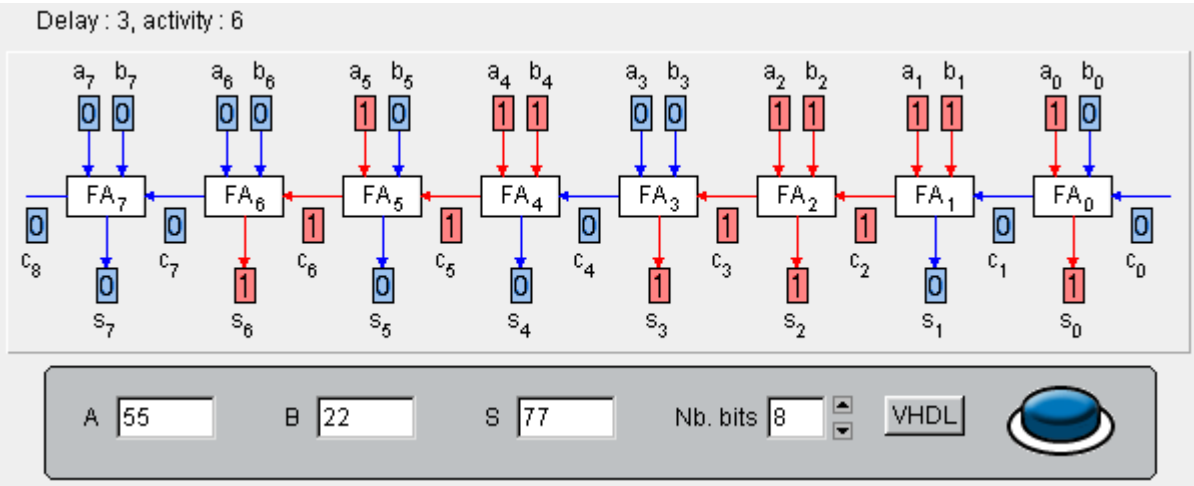
Carry propagate adder

The addition is by far the most common arithmetic operation in digital processors. Addition is itself very frequent and is also the basis of most other arithmetic operations like multiplication, division, square root extraction and elementary functions.

All "consistent" "FA" cells assembling preserves the property: *the weighed sum of the output bits equals the weighted sum of the input bits*.

To construct the adder S = A + B, the input bits come from the two numbers A and B and the output bits form the number S.

The number of "FA" cells is the same as the number of bits of A and B.



Performance of the carry ripple adder

Let us assume that all the possible values for A and B are equiprobable and independent:

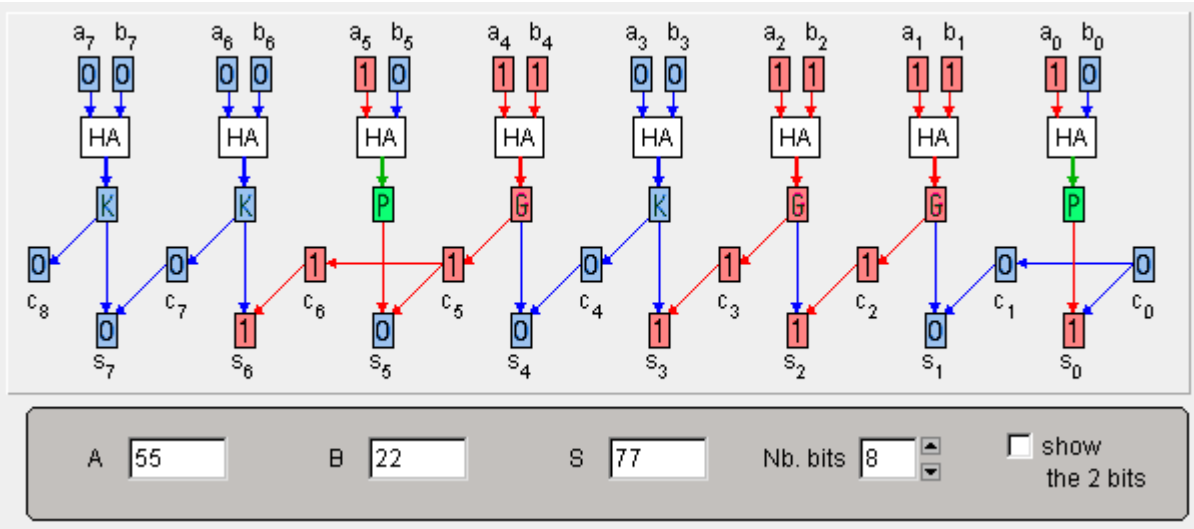
	minimum	average	maximum
delay	0	$\log_2(n)$	n
activity	0	3n / 4	$n^2 / 2$

The maximum delay (worst case) is usually not acceptable. Let us examine the carry propagation path that causes this delay.

Carry propagation path

For each "FA" cell, one of the three following case occurs:

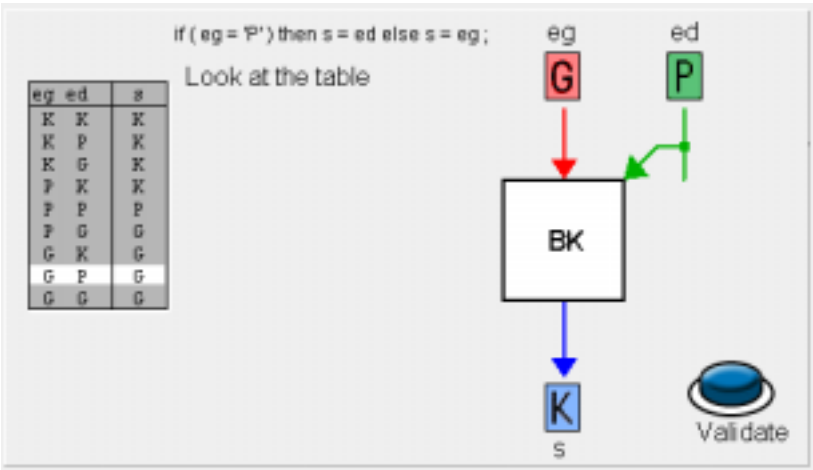
- the carry c_{i+1} is set to '0', noted 'K', if $a_i = 0$ and $b_i = 0$
- the carry c_{i+1} is set to '1', noted 'G', if $a_i = 1$ and $b_i = 1$
- the carry c_{i+1} is propagated, noted 'P', if ($a_i = 0$ and $b_i = 1$) or ($a_i = 1$ and $b_i = 0$). In this last case $c_{i+1} = c_i$. This is the unfavorable case, materialized by an horizontal arrow in the next applet.



The three case 'K', 'G' and 'P' are encoded onto two 2 bits.

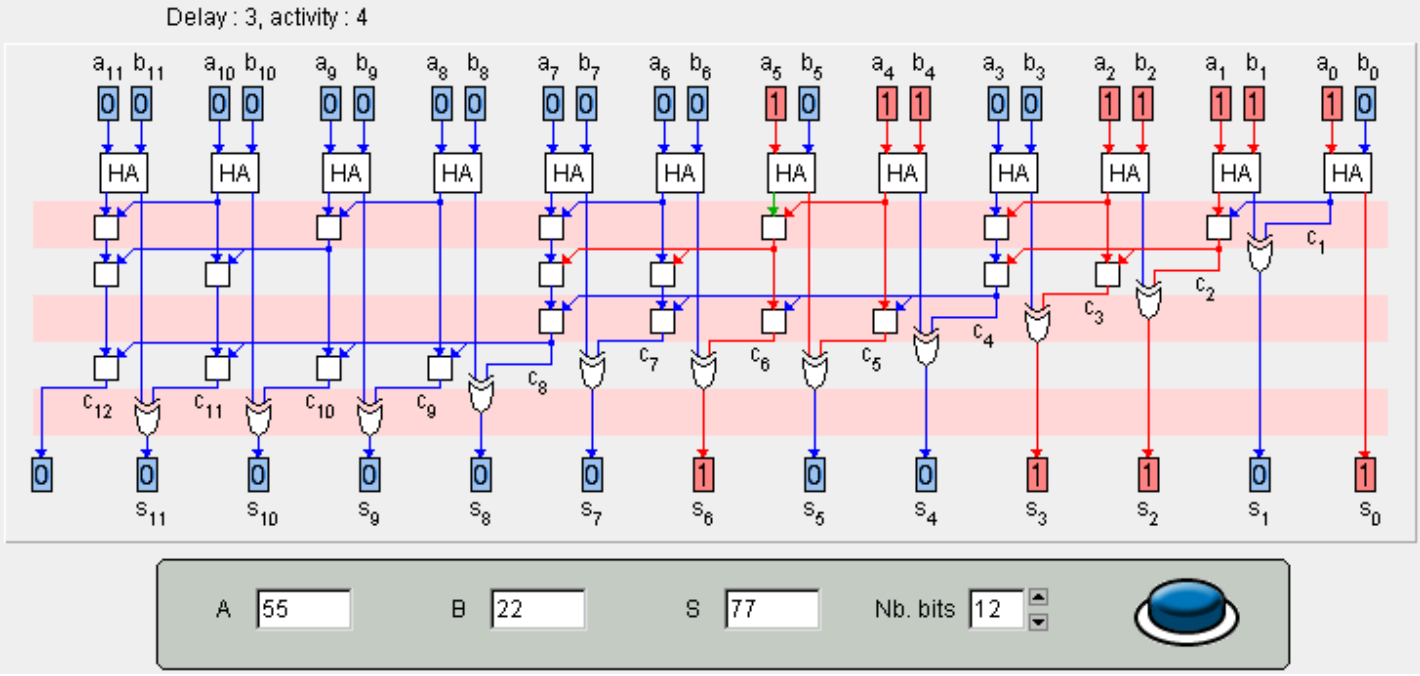
"BK" cell (Brent & Kung

The "BK" cell computes the carry for two binary positions (two "FA" cells) or more generally two blocks of "FA" cells.



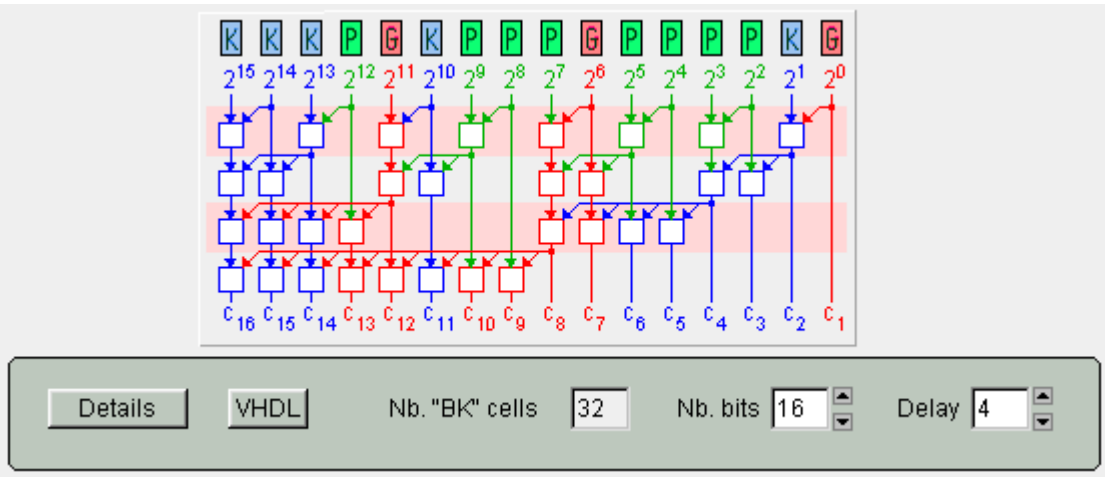
Sklansky's adder

To design fast adders, binary trees of "BK" cells will first generate simultaneously all the carries c_i. The "Sklanski's adder" builds recursively 2-bit adders, 4-bit adders, 8-bit adders, 16-bit adder and so on by abutting each time two smaller adders. The architecture is simple and regular, but may suffer from fan-out problems. Besides in most of the cases it is possible to use less "BK" cells for the same delay.



Fast adders (Brent & Kung)

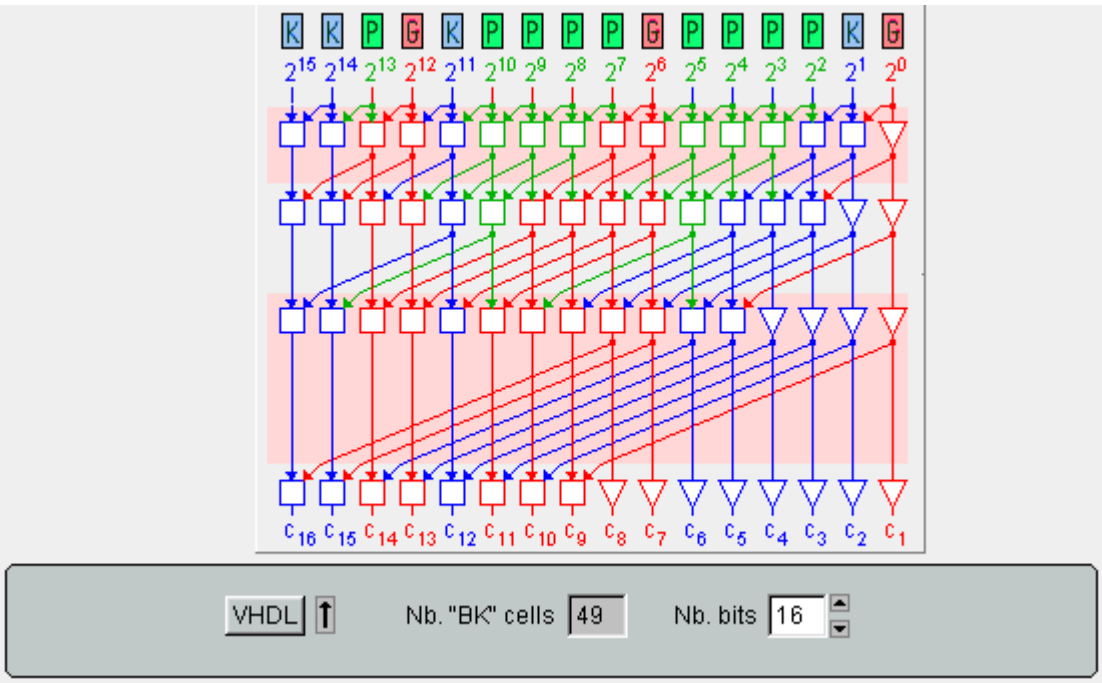
In a fast adder, <u>all the carries</u> c_i are computed <u>simultaneously</u> through a binary tree of "BK" cells. To save on complexity, sharable intermediate results are computed once. There is only <u>one rule</u> to construct the trees: every output with position i must be connected to <u>all inputs</u> of position less than or equal to i by a tree of "BK" cells. The rule simplicity usually allows for many correct constructions.



In the "BK" cell tree, one may trade cells for delay, usually for the same addition the less the delay, the more the "BK" cells.

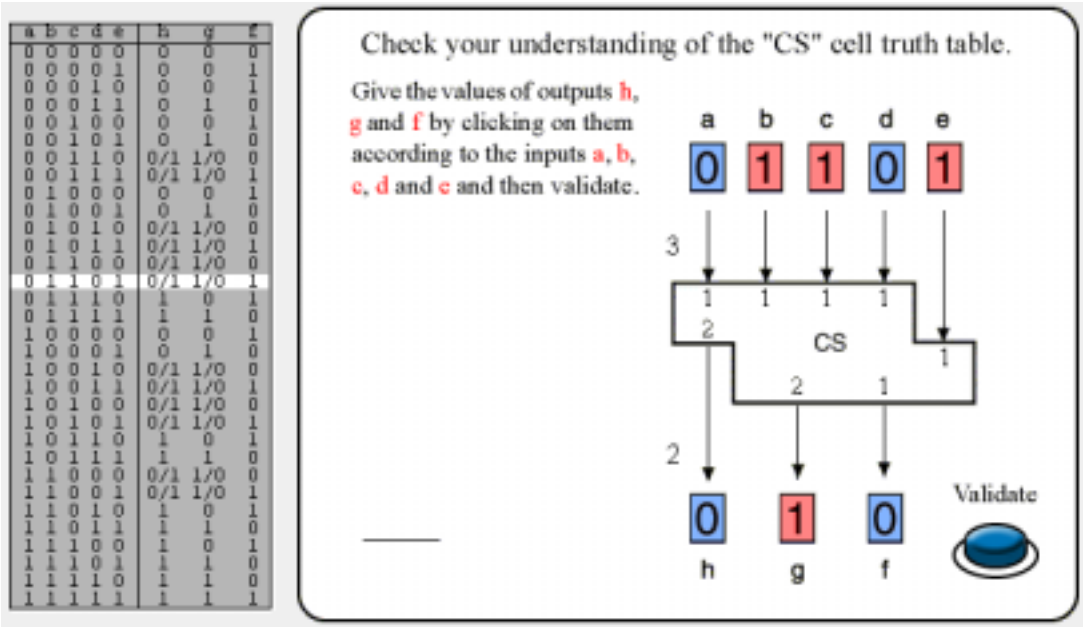
- change the number of bits and/or the delay
- check that the trees follow the construction rule by clicking on a signal (a line), notice that each signal is named by a pair of integers.
- simulate the carries computation by clicking on the keys K.
- display the tree construction process by clicking the key "Details"
- display the adder VHDL description by clicking the key "VHDL". To save the VHDL description, select it, then copy and paste it into a text editor.

Kogge & Stone adders The binary trees of "BK" cells in the Kogge et Stone adders are not sharing. Consequently the signal fan out is reduced to the minimum at the expense of more "BK" cells. Since the delay increases with the fan-out, it is here a bit shorter.



Ling adder In the Ling's adder, the "BK" trees give a primitive called "pseudo carry". It avoids the computation of p_i and g_i, but on the other hand the carry has to be deduced from the "pseudo carry". The trick is that this late computation is overlapped by the "BK" cells delays. Consequently this adder is faster (a little bit) than the corresponding "BK" adder. The VHDL synthesis from the applet takes advantage of that.

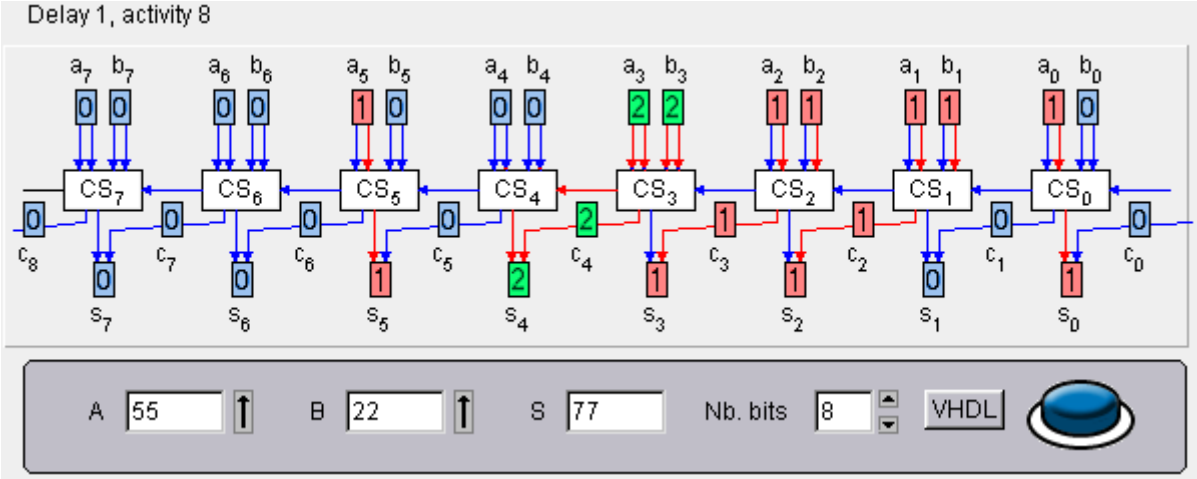
"CS" Cell In the "CS" cell, the weighted sum of the outputs equals the weighted sum of the inputs. In other words "a + b + c + d + e = 2*h + 2*g + f ". The "CS" cell is not only a "5⇒3 counter", but moreover the output "h" is never dependent on the input "e".



a	b	c	d	e	h	g	f
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	0/1	1/0	0
0	0	1	1	1	0/1	1/0	1
0	1	0	0	0	0	0	1
0	1	0	0	1	0	1	0
0	1	0	1	0	0/1	1/0	0
0	1	0	1	1	0/1	1/0	1
0	1	1	0	0	0/1	1/0	0
0	1	1	0	1	0/1	1/0	1
0	1	1	1	0	1	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	0/1	1/0	0
1	0	0	1	1	0/1	1/0	1
1	0	1	0	0	0/1	1/0	0
1	0	1	0	1	0/1	1/0	1
1	0	1	1	0	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	0	0/1	1/0	0
1	1	0	0	1	0/1	1/0	1
1	1	0	1	0	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	0	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1

Carry propagation The "CS" cell does not propagate the input carry "e" to the output "h". It makes "carry propagation free" adders possible. The number of necessary "CS" cells is precisely

free adder given by the number of digits to be added.
On the other hand each digit is coded onto 2 bits and the digit value is the sum of those 2 bits. Therefore the possible digit values are '0', '1' and '2'.

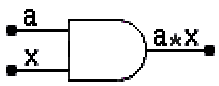


This notation system for integer numbers allows addition with a delay both short and independent of the digits number. Yet this system demands about twice as many bits as the standard binary notation for a comparable range. Consequently the same value may have several representations. The vertical arrow ↑ next to the numbers value changes the representation without changing the value. Among the representations, the one with only '0' or '1' is always unique.

Multipliers

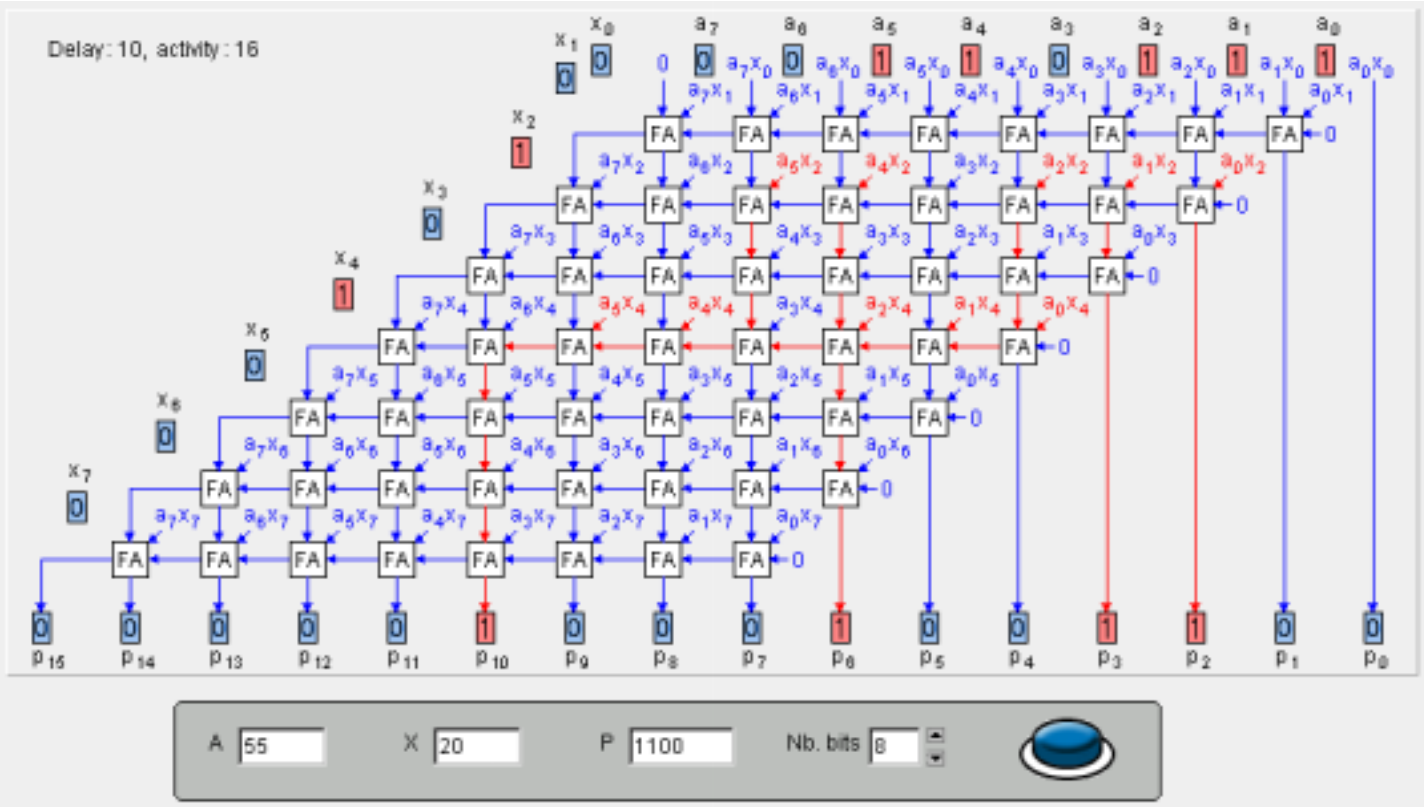
Multiplier The multiplication comes second for frequency of use.

AND gate An "AND" gate multiplies two bits. To multiply two n-bit numbers A and X, n^2 "AND" gates are required. The weighted sum of the n^2 gate outputs has indeed the same value as P = A * X. However this set of bit is not a number, although its value is computed as if it was a number.



Since $A < 2^n$ et $X < 2^n$, the product $P < 2^{2n}$ and therefore P is written with 2 n bits.

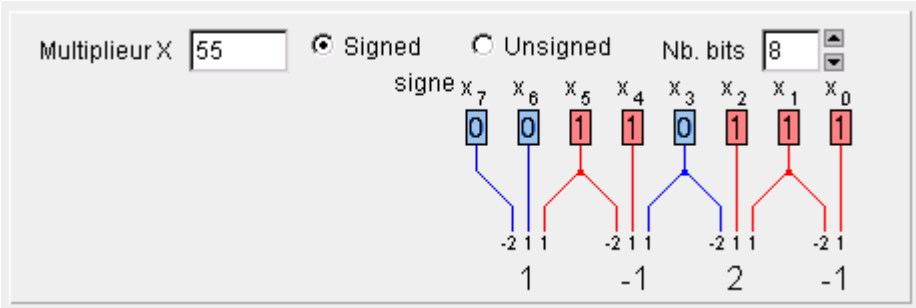
Unsigned Multiplication A regular structure of "AND" gates and "FA" cells with a "consistent" assembling first produces the partial products and then reduces them to a number P. Since each "FA" cell reduces the number of bits by exactly one (while preserving the sum), the necessary number of "FA" cells is $n^2 - 2n$ (number of input bits – number of output bits). Yet in the following applet there are more "FA" than necessary since some '0' must also be reduced, to be precise just as many '0' as bits of X.



Fast Multipliers Many approaches lead to a speed improvement:

- Divide the number of partial product bits using a higher radix.
- Use a tree structure for the "FA" cell reduction net.
- Use "CS" cell, with a reduction power two times the one of "FA" cell. Besides this cell allow balanced binary trees (with some difficulties).

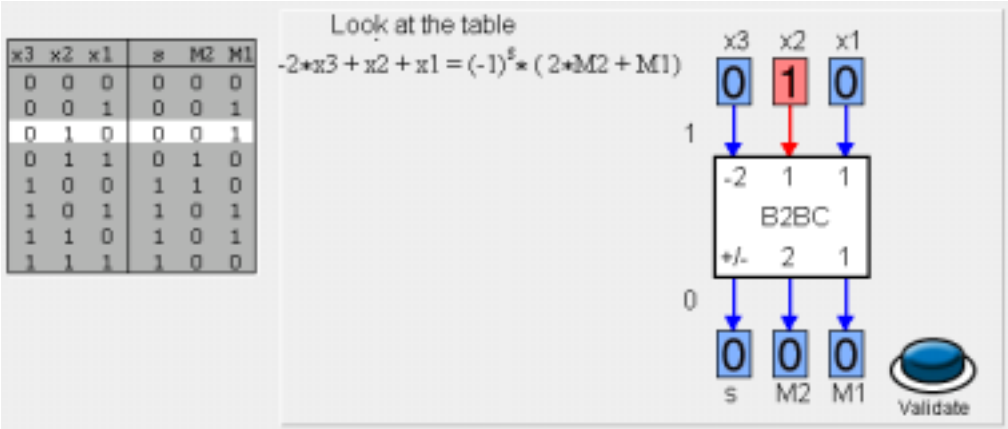
Booth recoding Using a larger radix automatically reduces the multiplier X digits number. Let have a look on radix 4, using two times as less digit as radix 2 for the same range. The "Booth Code" ("BC" for short) is the minimally redundant symmetric radix-4 code. Digits values ∈ {-2, -1, -0, 0, 1, 2}. The 3-bit code picked below, known as "sign/absolute-value", has 2 notations for zero.



However the partial products are computed by a cell more complex than a simple "AND" gate.

Cell of the binary to "BC" converter

Check whether you are acquainted with the logic of the "B2BC" cell, which convert a "BC" digit into "sign/absolute-value" for the generation of partial products. The sum is preserved, i.e. : $-2*x3 + x2 + x1 = (-1)^s * (2*M2 + M1)$:

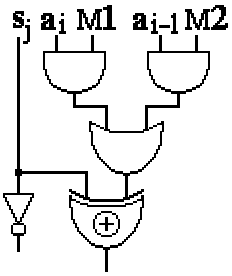


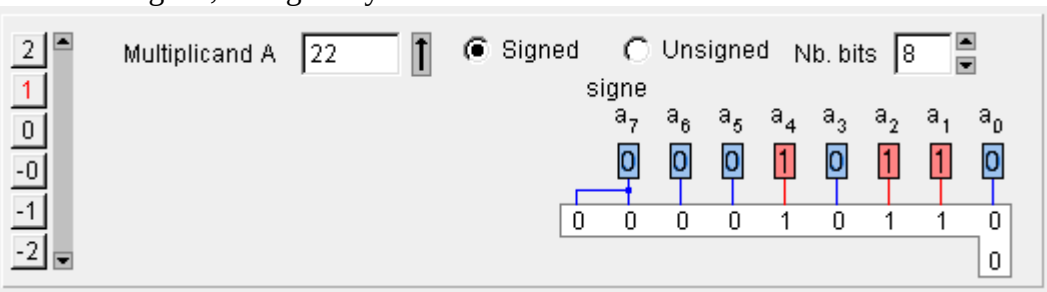
x3	x2	x1	s	M2	M1
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	1	0	0

The conversion of X requires half as many "B2BC" as bits in X.

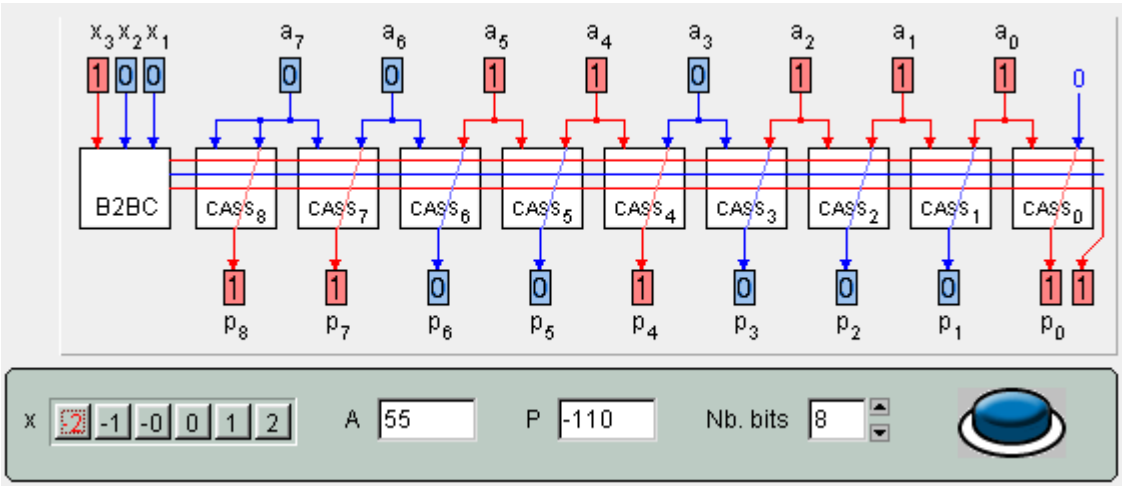
Multiplication of A bits by one "BC" digit

The multiplication by one "BC" digit ∈ {-2, -1, -0, 0, 1, 2} adds 2 bits on top of the bits of A: one at left to get either A or 2A, another for the input carry in case of subtraction. Since A is signed, its sign may have to be extended to the bit added at the left.



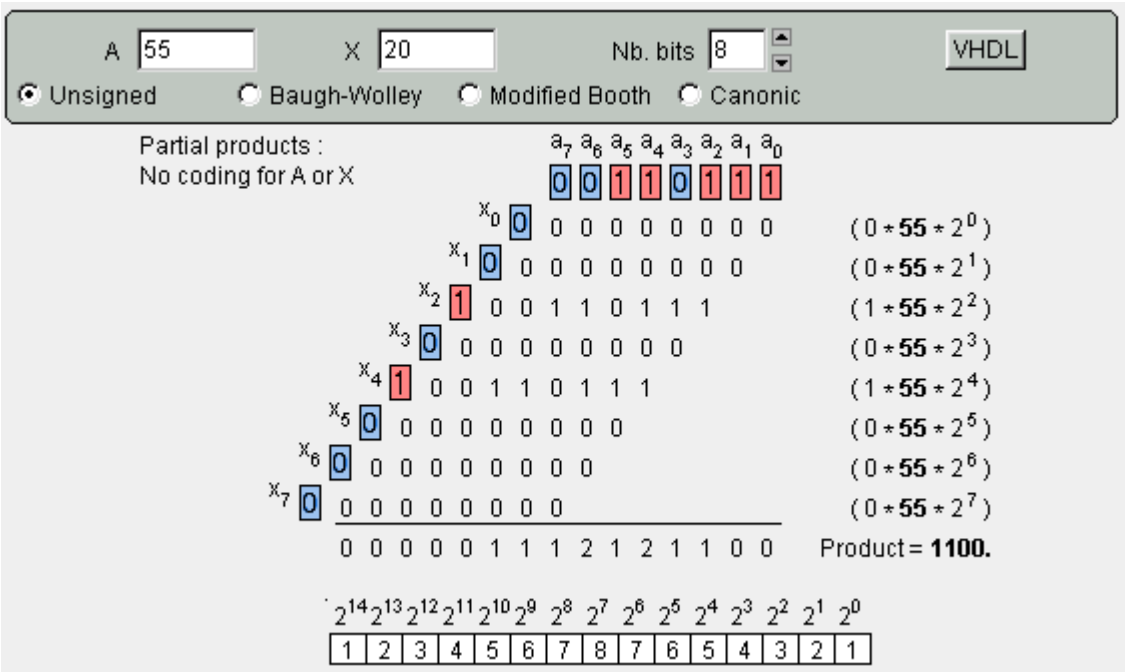


The multiplication requires as many "CASS" cells as bits in A plus 1.



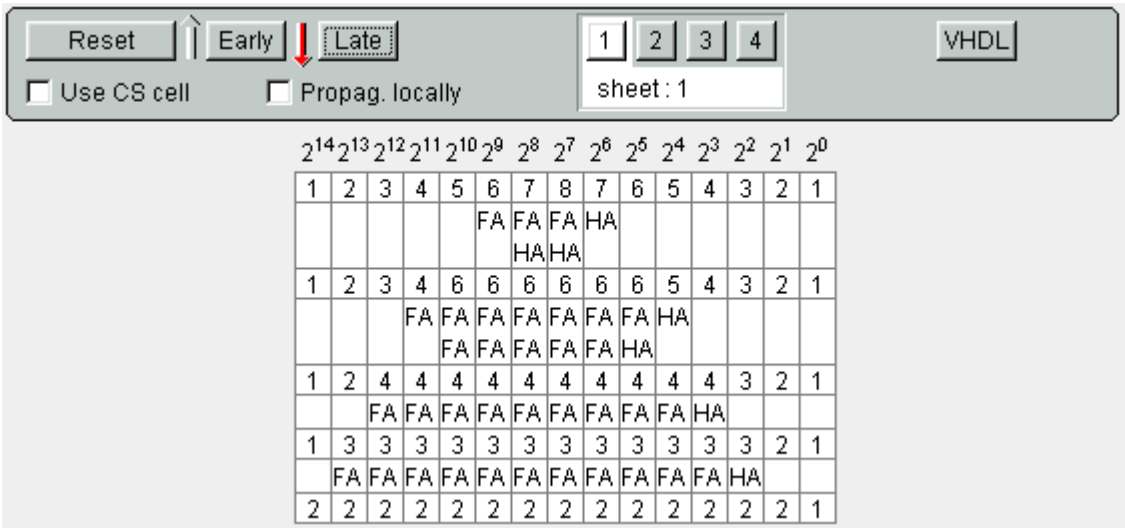
Partial products generation

The multiplication first step generates from A and X a set of bits whose weighted sum is the product P. For unsigned multiplication, P most significant bit is positive, while in 2's complement it is negative.



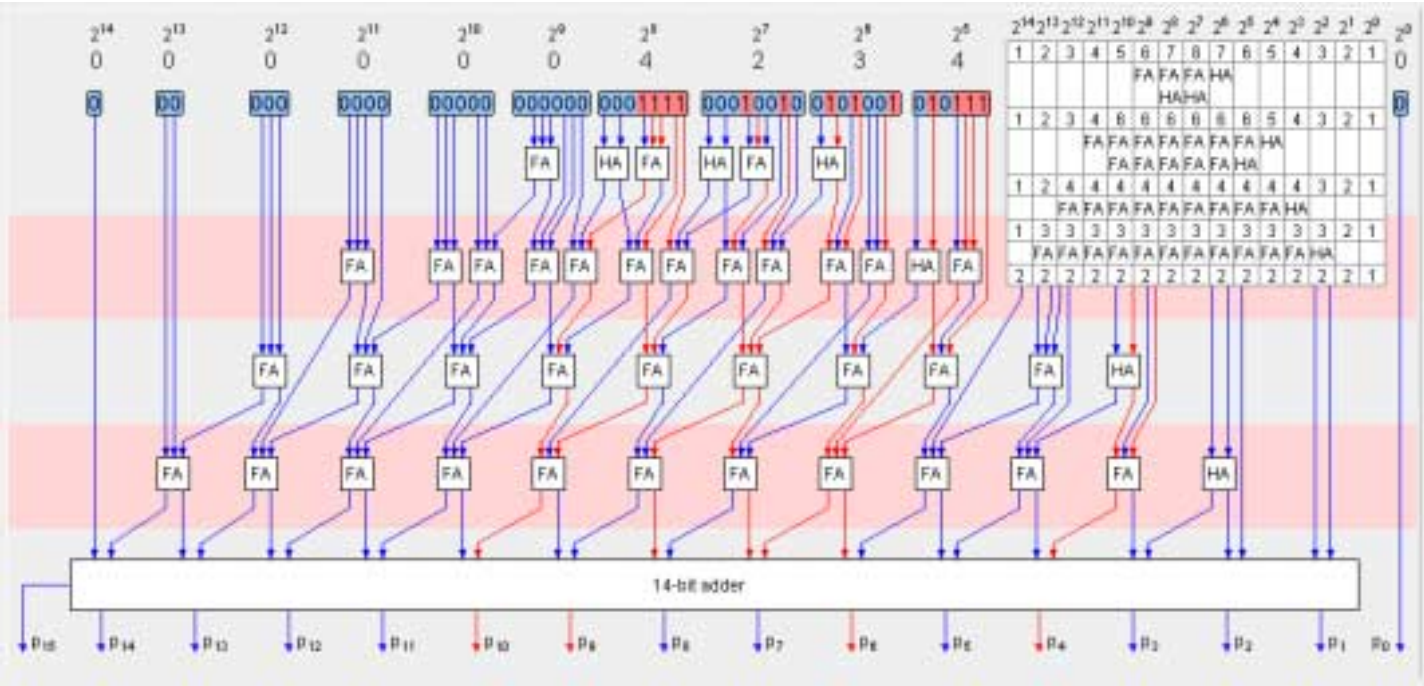
Partial products reduction

The multiplication second step reduces the partial products from the preceding step into two numbers while preserving the weighted sum. The sough after product is the sum of those two numbers. The two numbers will be added during the third step. The reduction trees synthesis follows the Dadda's algorithm, which assures the minimum counter number. If on top of that we impose to reduce as late as (or as soon as) possible then the solution is unique. The two binary number that have to be added during the third step may also be seen a one number in "CS" notation (2 bits per digit).



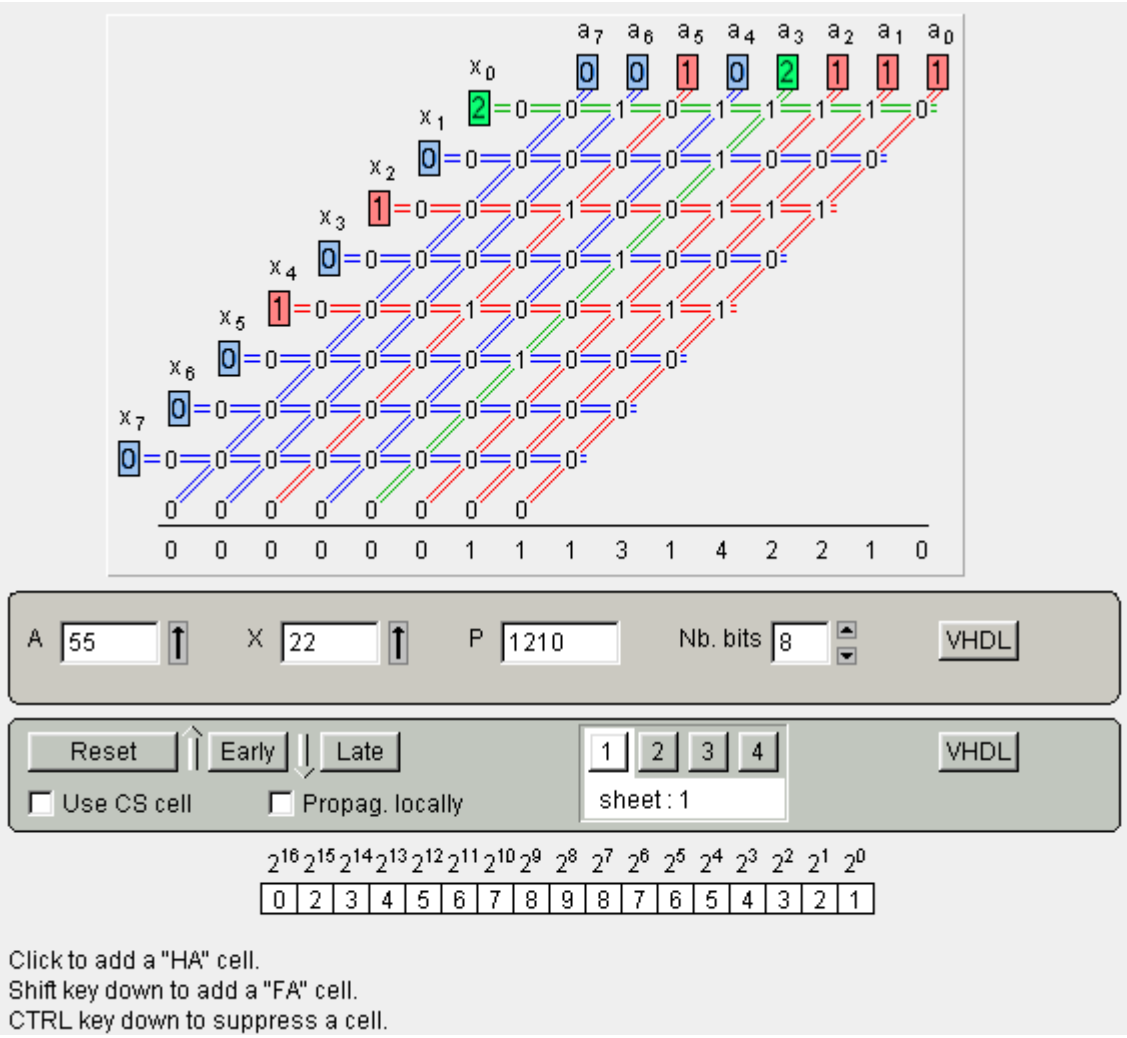
Example of Wallace tree

The following trees reduce 8^2 partial products (for example the product of two 8-bit unsigned integers). The "Wallace trees" reduce "as late as possible" (key "late" on the preceding applet). The weighted sum of the 16 output bits equals the weighted sum of the 64 input bits.



Partial product of "CS" operands

Multiplier X and multiplicand A are now both in "CS" notation, i.e. with digits values ∈ { 0, 1, 2}. We want to generate a set of bits whose weighted sum equals A * X. To make sure that we get bits (easy to add), it is necessary that either in A or in X every digit '2' is preceded by a '0' at its left.

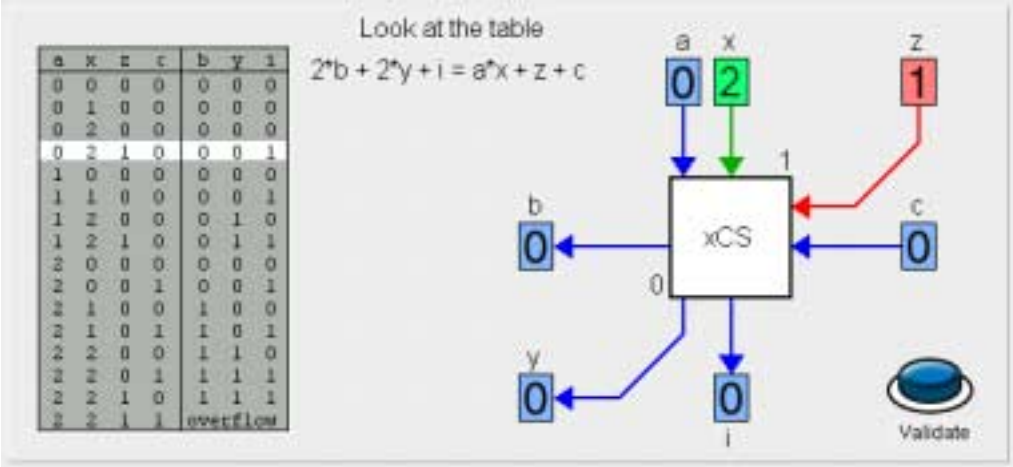


Partial products reduction

The partial product of two "CS" is a simple bit, reduced in the very same way as for conventional fast multiplication.

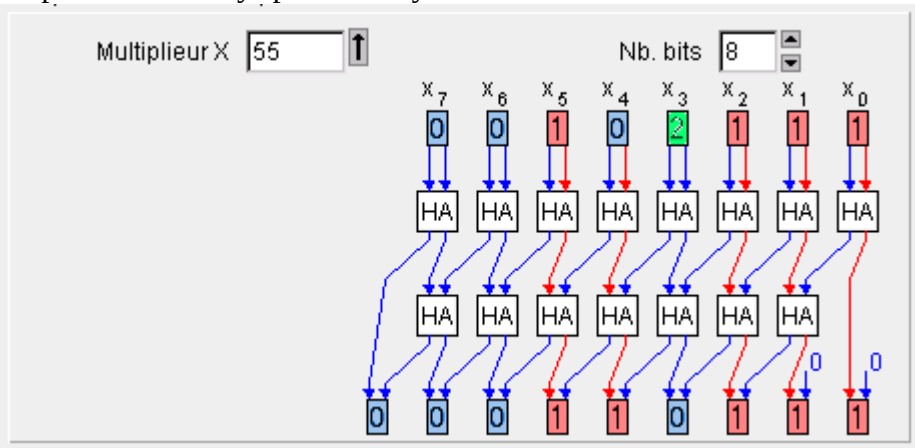
"xCS" cell

The "xCS" cell computes the product of two digits a and x in "CS" notation. Its arithmetic equation is "2 × b + 2 × y + i = a × x + z + c". Furthermore the outputs "b" and "y" does not depend on "c" or "z" (no propagation).



Coding circuit "CS2CS"

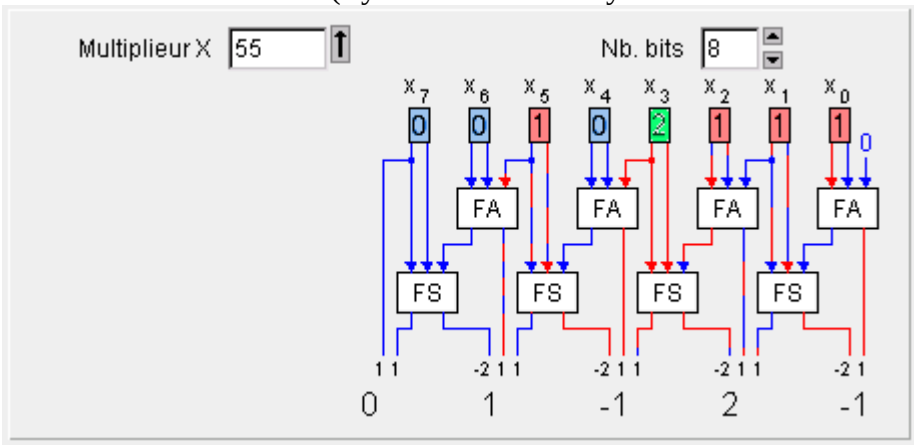
The transcoder "CS2CS" passes from "CS" to "CS" while making sure that in the output a '2' is always preceded by a '0'.



This permits to generate the partial product of a multiplicand A by a multiplier X both in "CS", with no overflow.

Coding circuit "CS2BC"

The transcoder "CS2BC" passes from "CS" to "BC" , that is from the "carry-save" code to the "Booth Code" (symmetric minimally redundant radix-4 code).



Integer constants multiplications

Multiplication of a variable by integer constants

The discrete Fourier transform, the discrete cosine transform or inverse, digital filters, and so on, all contain the multiplication of a variable X by several constants C1, C2, .. Cn. The factorization of those constants permits a dramatic reduction of the number of additions/subtractions demanded by those multiplications. The following applet computes

Y1 = X*C1, Y2 = X*C2, .. Yn = X*Cn.

Nb. const. 3 | 2717 | 2726 | 2723 | VHDL

$Y1 = X*C1 = X*\mathbf{2717} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{4} + X*2^{3} + X*2^{2} + X*2^{0}$

$Y2 = X*C2 = X*\mathbf{2726} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{5} + X*2^{2} + X*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{5} + X*2^{1} + X*2^{0}$

Step 0 : cost = 16 additions

$Y1 = X*C1 = X*\mathbf{2717} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{5} - X*2^{2} + X*2^{0}$

$Y2 = X*C2 = X*\mathbf{2726} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{5} + X*2^{3} - X*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{5} + X*2^{2} - X*2^{0}$

Step 1 : cost = 12 additions and 3 subtractions

$Y1 = X*C1 = X*\mathbf{2717} = -X*2^{2} + X*2^{0} + Y4*2^{5}$

$Y2 = X*C2 = X*\mathbf{2726} = X*2^{11} + X*2^{9} + X*2^{7} + X*2^{5} + X*2^{3} - X*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = X*2^{2} - X*2^{0} + Y4*2^{5}$

$Y4 = X*C4 = X*\mathbf{85} = X*2^{6} + X*2^{4} + X*2^{2} + X*2^{0}$

Step 2 : cost = 9 additions and 3 subtractions

$Y1 = X*C1 = X*\mathbf{2717} = -X*2^{2} + X*2^{0} + Y4*2^{5}$

$Y2 = X*C2 = X*\mathbf{2726} = Y4*2^{3} + Y5*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = X*2^{2} - X*2^{0} + Y4*2^{5}$

$Y4 = X*C4 = X*\mathbf{85} = X*2^{6} + X*2^{4} + X*2^{2} + X*2^{0}$

$Y5 = X*C5 = X*\mathbf{1023} = X*2^{10} - X*2^{0}$

Step 3 : cost = 6 additions and 3 subtractions

$Y1 = X*C1 = X*\mathbf{2717} = -X*2^{2} + X*2^{0} + Y4*2^{5}$

$Y2 = X*C2 = X*\mathbf{2726} = Y4*2^{3} + Y5*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = X*2^{2} - X*2^{0} + Y4*2^{5}$

$Y4 = X*C4 = X*\mathbf{85} = Y6*2^{2} + Y6*2^{0}$

$Y5 = X*C5 = X*\mathbf{1023} = X*2^{10} - X*2^{0}$

$Y6 = X*C6 = X*\mathbf{17} = X*2^{4} + X*2^{0}$

Step 4 : cost = 5 additions and 3 subtractions

$Y1 = X*C1 = X*\mathbf{2717} = Y4*2^{5} - Y7*2^{0}$

$Y2 = X*C2 = X*\mathbf{2726} = Y4*2^{3} + Y5*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = Y4*2^{5} + Y7*2^{0}$

$Y4 = X*C4 = X*\mathbf{85} = Y6*2^{2} + Y6*2^{0}$

$Y5 = X*C5 = X*\mathbf{1023} = X*2^{10} - X*2^{0}$

$Y6 = X*C6 = X*\mathbf{17} = X*2^{4} + X*2^{0}$

$Y7 = X*C7 = X*\mathbf{3} = X*2^{2} - X*2^{0}$

Step 5 : cost = 4 additions and 3 subtractions

$Y1 = X*C1 = X*\mathbf{2717} = Y4*2^{5} - Y7*2^{0}$

$Y2 = X*C2 = X*\mathbf{2726} = Y4*2^{3} + Y5*2^{1}$

$Y3 = X*C3 = X*\mathbf{2723} = Y4*2^{5} + Y7*2^{0}$

$Y4 = X*C4 = X*\mathbf{85} = Y6*2^{2} + Y6*2^{0}$

$Y5 = X*C5 = X*\mathbf{1023} = X*2^{10} - X*2^{0}$

$Y6 = X*C6 = X*\mathbf{17} = X*2^{4} + X*2^{0}$

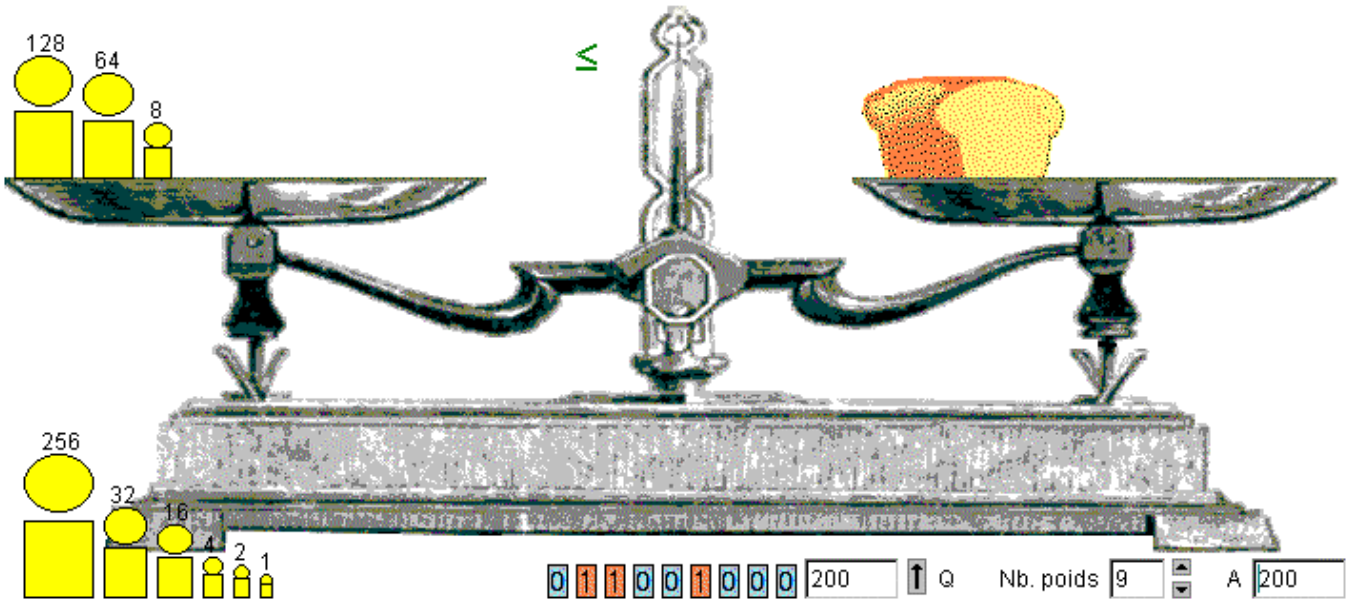
$Y7 = X*C7 = X*\mathbf{3} = X*2^{1} + X*2^{0}$

Step 6 : cost = 5 additions and 2 subtractions

Dividers

Weighting a bread loaf with restoration and without restoration

We want to computer Q = A ÷ D. By a stroke of luck, we have available a scale, a white bread whose weight is actually just A and a set of weights with values D, 2D, 4D, 8D, ... 2^i*D respectively marked with 1, 2, 4, 8, ... 2^i.
In fact D is a binary number, and 2^i*D is simply obtained by shifting D. The scale compares the sum of the weights on each of the two plates (≤ or >).



Digit recurrence division

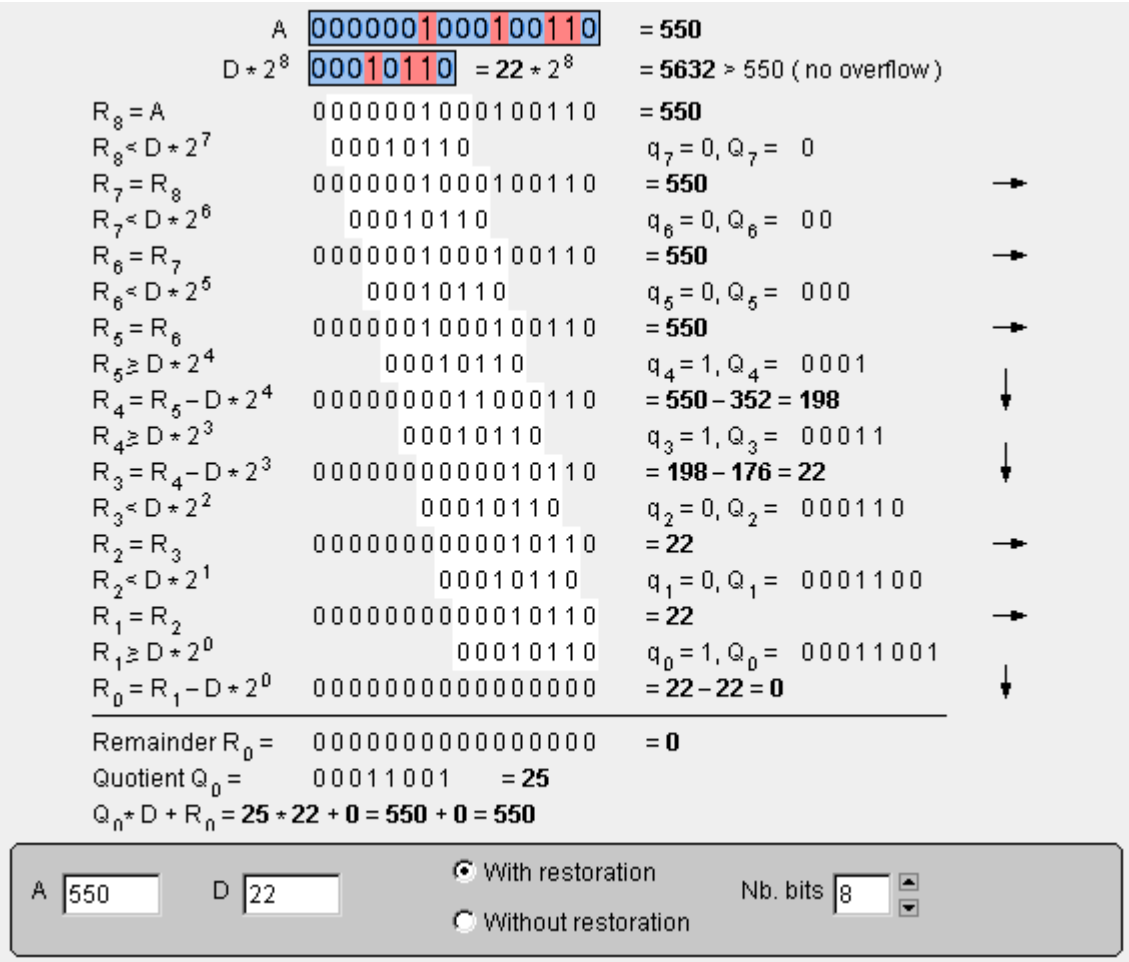
Division is not frequent. Nevertheless since its execution delay is far larger than the addition or multiplication's one, its contribution to the total execution time is substantial, thus it is advisable to design dividers carefully.
Let say that we want Q = A ÷ D. This is equivalent to Q ∗ D = A. Therefore if Q and D are both written onto n bits, A is written onto 2n bits.
Let us build a series Q_n, Q_{n-1}, ... Q_2, Q_1, Q_0 and a series R_n, R_{n-1}, ... R_2, R_1, R_0 such that the invariant $A = Q_j * D + R_j$ holds for all j.
The recurrence is :

- $Q_{j-1} = Q_j + q_{j-1} * 2^{j-1}$
- $R_{j-1} = R_j - q_{j-1} * D * 2^{j-1}$

with initial conditions:

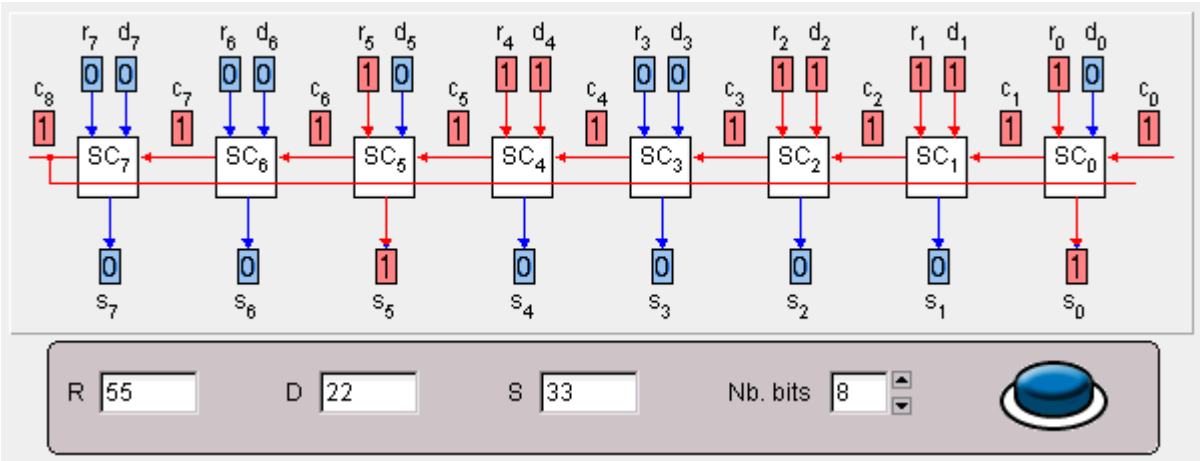
- $Q_n = 0$
- $R_n = A$.

When the recurrence stops, we have $Q = Q_0 = \sum_{i=0}^{n} q_i * 2^i$. $R = R_0$ is the division remainder.

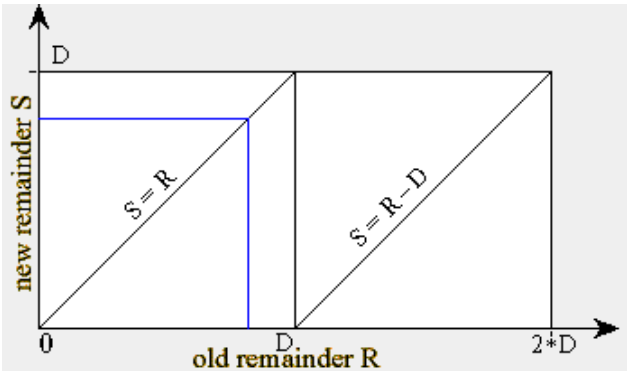


Conditional subtractor

A "conditional subtractor" gives the following result S:
if R < D **then** S = R **else** S = R – D ;
Each "SC" cell computes both the result and the carry (borrow) of the subtraction R – D. If the output carry value (leftmost) is '1' then S is assigned the result of the subtraction else S is assigned the value of R. This last case, that seems to "restore" R to its previous value before the subtraction is sometimes called "restoration", from which the divider's name derives,



The "conditional subtractor" function: **if** R < D **then** S = R **else** S = R – D, is abstracted by its transfer function called "Robertson's diagram". To converge the division imposes moreover that 0 ≤ R ≤ 2*D.

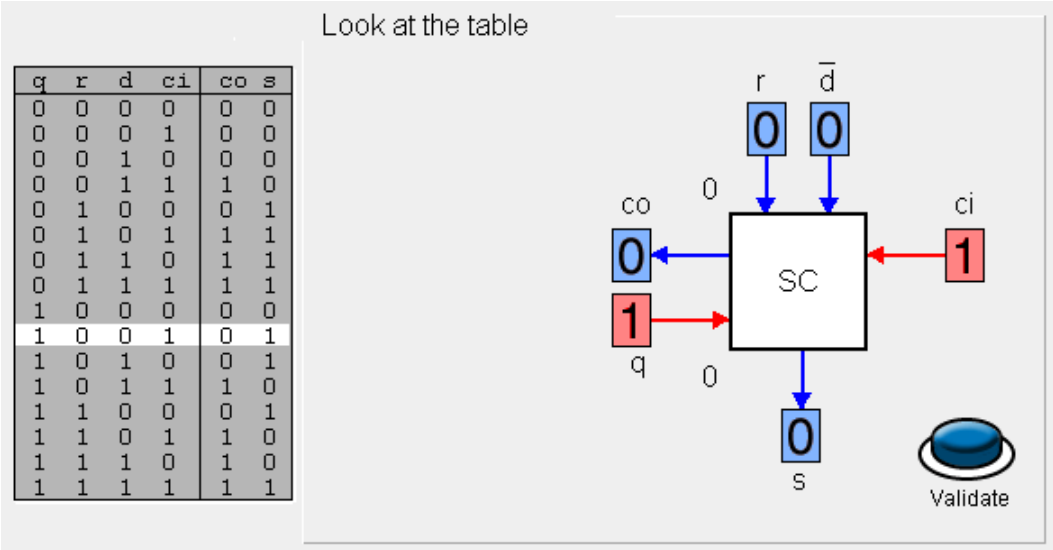


"SC" cell of the conditional subtractor

Check whether you are acquainted with the logic of the "SC" cell :

if q = 0 **then** { co = majority (r, $\overline{d}$, ci) ; s = r ; } // identity

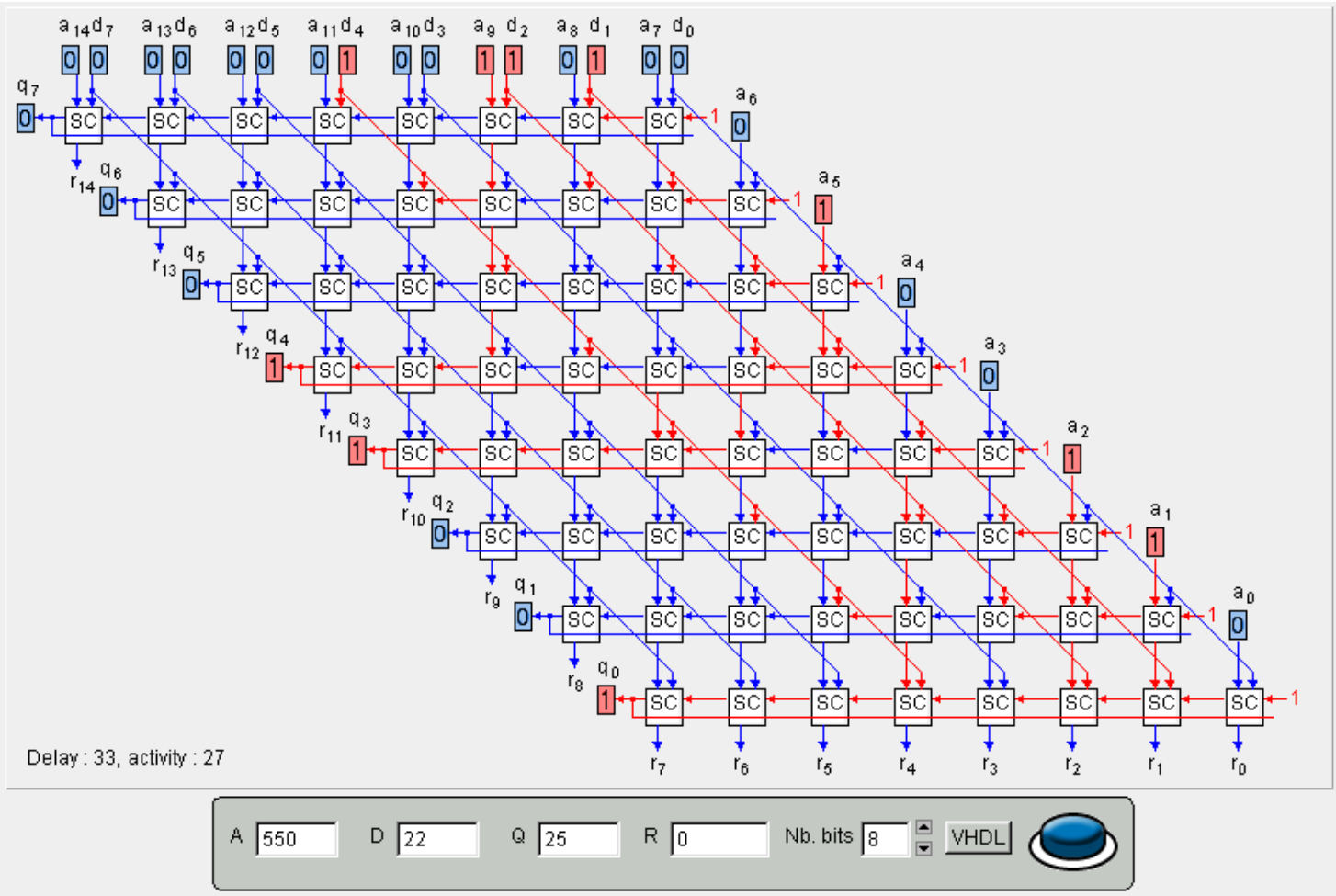
else { co = majority (r, $\overline{d}$, ci) ; s = r ⊕ $\overline{d}$ ⊕ ci ; } // subtraction



q	r	d	ci	co	s
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

Restoring divider

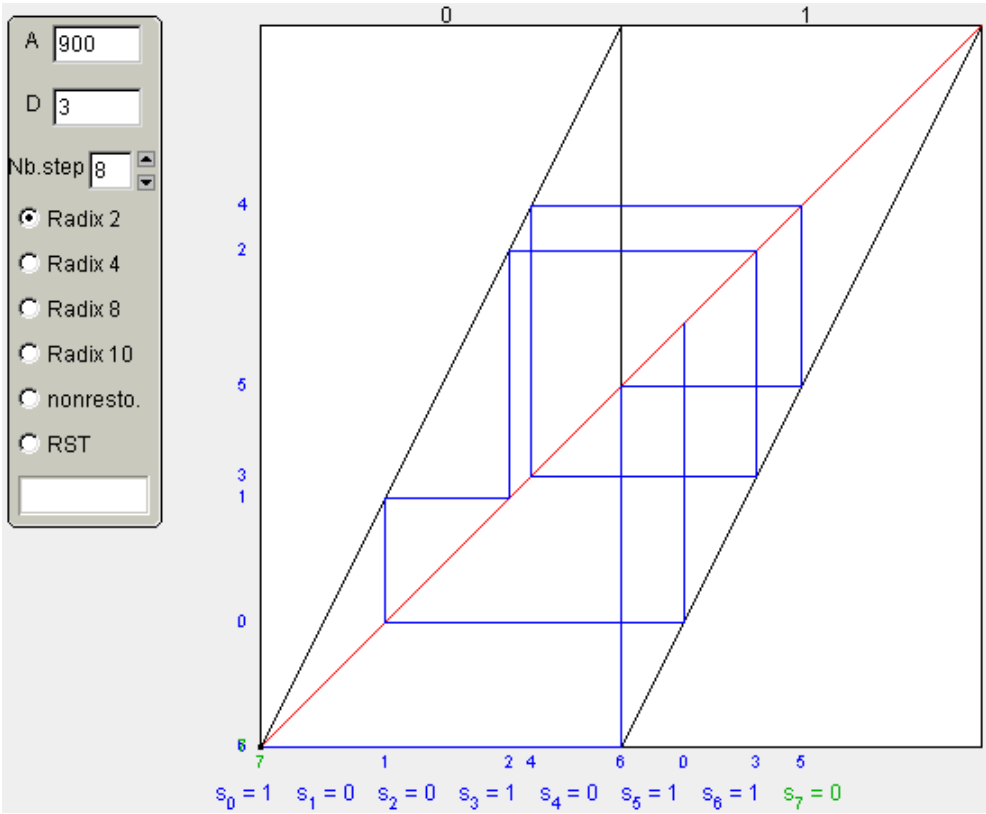
A "restoring" divider consists in a series of shifts and attempted subtraction. It is made of a regular net of conditional subtraction cell "SC" (subtraction or nothing according to a carry out bit).



Fast dividers Three approaches may be combined to realize fast dividers:

- Utilization of carry-propagation-free addition/subtraction.
- Preconditioning of the dividend and the divisor in order to simplify the division.
- Use of higher radixes to reduce the number of steps.

Robertson's Diagram To obtain a square Robertson's diagram, the successive partial remainders are normalized: ($R_j * b^{-j}$) where b is the numeration radix.
The black slopes represent the transfer function $R_j \Rightarrow R_{j-1}$, the red line is the identity function, that passes to the following step. Pulling the mouse out of the pictures suspends the animation. Clicking ends or restarts the animation. Clicking inside the square sets another starting point.

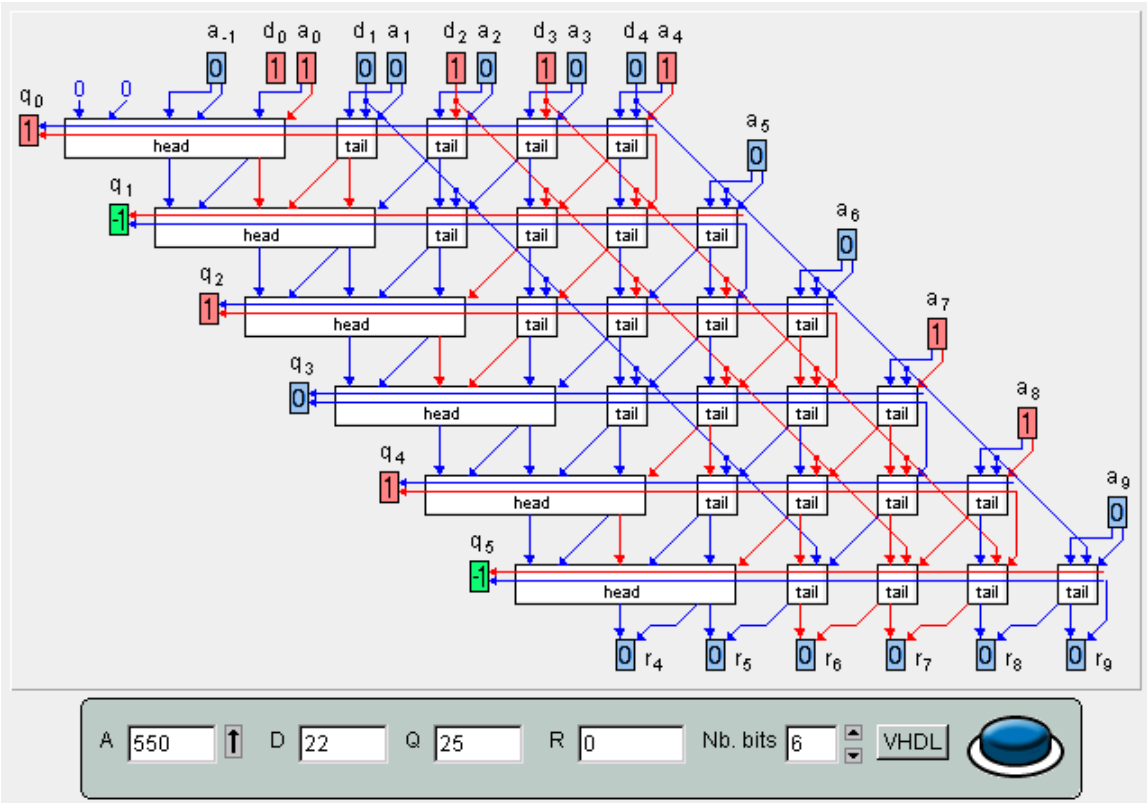


Radix 10 sounds familiar to us; it is given here just for illustration because it would not be very efficient in binary.

"SRT" division or carry propagation free division

To avoid the delay of the carry propagation, the following applet uses a stack of borrow-save "BS" adders/subtractors. The "tail" cell, variant of the "SC" cell, is controlled by two bits and executes one of the three following operations:

- an addition : $R_{j-1} = R_j + 2^{j-1} * D$
- a subtraction: $R_{j-1} = R_j - 2^{j-1} * D$
- an identity: $R_{j-1} = R_j$



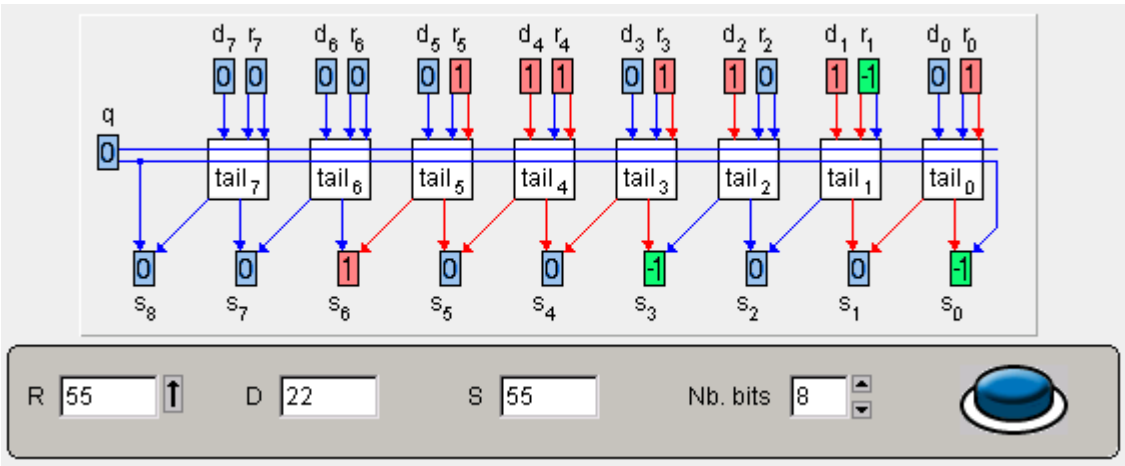
This operation is selected according to the sign of the partial remainders R_j. To always know precisely this sign would require the examination of all the remainder's digits. It is sufficient to check only three. Moreover, the position of the three digits is known: the rightmost one is aligned with the most significant non-zero bits of D. To nail down this digit position, D is "normalized", that is the position of its first "1" bit is fixed. For an n-bit divider, $2^{n-2} - 1 < D < 2^{n-1}$.

Conditional carry-propagation-free adder/subtractor

A "conditional adder/subtractor" yields one among the three following outputs:

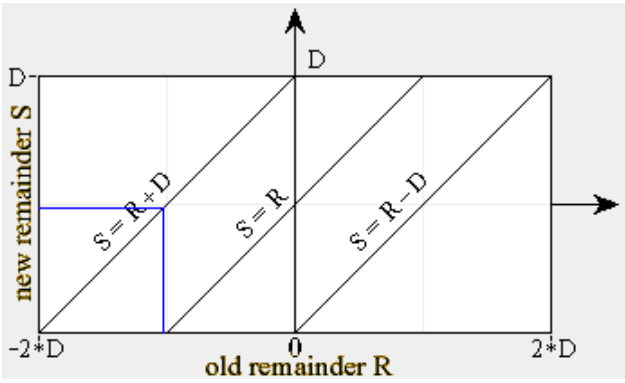
- **if** q = '-1' **then** S = R + D ;
- **if** q = '0' **then** S = R ;
- **if** q = '1' **then** S = R – D ;

Each "tail" cell executes a one-bit addition/subtraction. The carry is not propagated to the "tail" cell at left but fed directly to the "tail" cell below (next line).



The "conditional adder/subtractor" function sis abstracted by its transfer function called "Robertson's diagram". To converge the division imposes moreover that

$-2*D \le R \le 2*D$. If $-D \le R \le 2$ then S has two possible values.

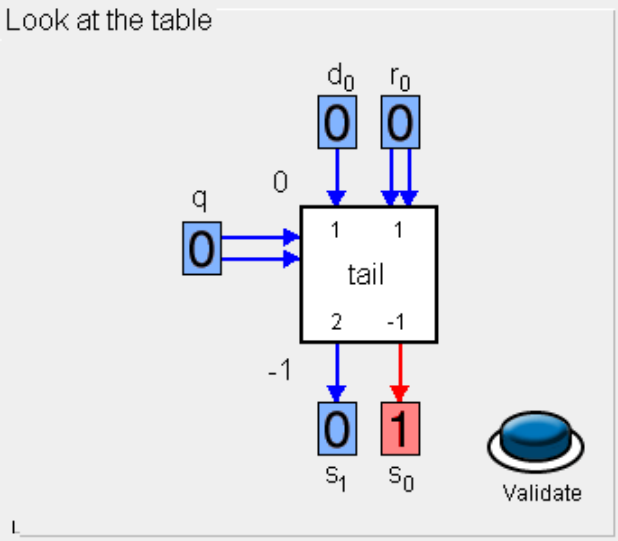


"tail" cell of the "SRT" divider

Check whether you are acquainted with the logic of the "tail" cell.

- **if** q = '-1' **then** $2*s_1 - s_0 = d_0 + r_0$; // addition
- **if** q = '0' **then** $2*s_1 - s_0 = r_0$; // identity
- **if** q = '1' **then** $2*s_1 - s_0 = \overline{d_0} + r_0$; // subtraction

q	d0	r0	s1	s0
-1	0	-1	0	1
-1	1	-1	0	0
-1	0	0	0	0
-1	1	0	1	1
-1	0	+1	1	1
-1	1	+1	1	0
0	0	-1	0	1
0	1	-1	0	1
0	0	0	0	0
0	1	0	0	0
0	0	+1	1	1
0	1	+1	1	1
+1	0	-1	0	0
+1	1	-1	0	1
+1	0	0	1	1
+1	1	0	0	0
+1	0	+1	1	0
+1	1	+1	1	1

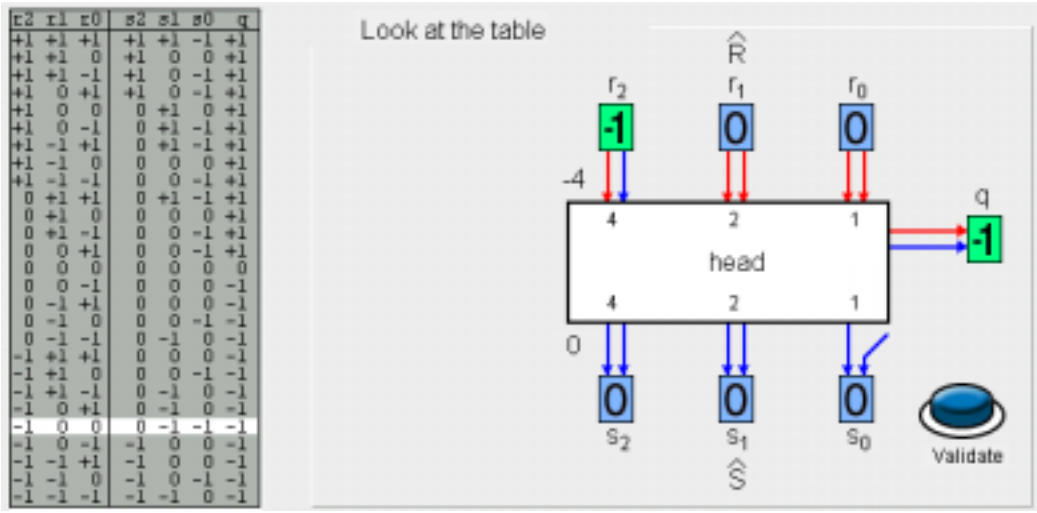


"head" cell of the "SRT" divider

Let $\hat{R} = r_2 * 4 + r_1 * 2 + r_0$ and $\hat{S} = s_2 * 4 + s_1 * 2 + s_0$ be the input and output values of the "head" cell. Another output is one quotient digit q .

- **if** $\hat{R} > 0$ **then** { $\hat{S} = \hat{R} - 2$; q = '1'; }
- **if** $\hat{R} = 0$ **then** { $\hat{S} = \hat{R}$; q = '0' ; }
- **if** $\hat{R} < 0$ **then** { $\hat{S} = \hat{R} + 1$; q = '-1'; }

In a real division (without overflow), output s_2 will <u>always</u> be 0. .



r2	r1	r0	s2	s1	s0	q
+1	+1	+1	+1	+1	-1	+1
+1	+1	0	+1	0	0	+1
+1	+1	-1	+1	0	-1	+1
+1	0	+1	+1	0	-1	+1
+1	0	0	0	+1	0	+1
+1	0	-1	0	+1	-1	+1
+1	-1	+1	0	+1	-1	+1
+1	-1	0	0	0	0	+1
+1	-1	-1	0	0	-1	+1
0	+1	+1	0	+1	-1	+1
0	+1	0	0	0	0	+1
0	+1	-1	0	0	-1	+1
0	0	+1	0	0	-1	+1
0	0	0	0	0	0	0
0	0	-1	0	0	0	-1
0	-1	+1	0	0	0	-1
0	-1	0	0	0	-1	-1
0	-1	-1	0	-1	0	-1
-1	+1	+1	0	0	0	-1
-1	+1	0	0	0	-1	-1
-1	+1	-1	0	-1	0	-1
-1	0	+1	0	-1	0	-1
-1	0	0	0	-1	-1	-1
-1	0	-1	-1	0	0	-1
-1	-1	+1	-1	0	0	-1
-1	-1	0	-1	0	-1	-1
-1	-1	-1	-1	-1	0	-1

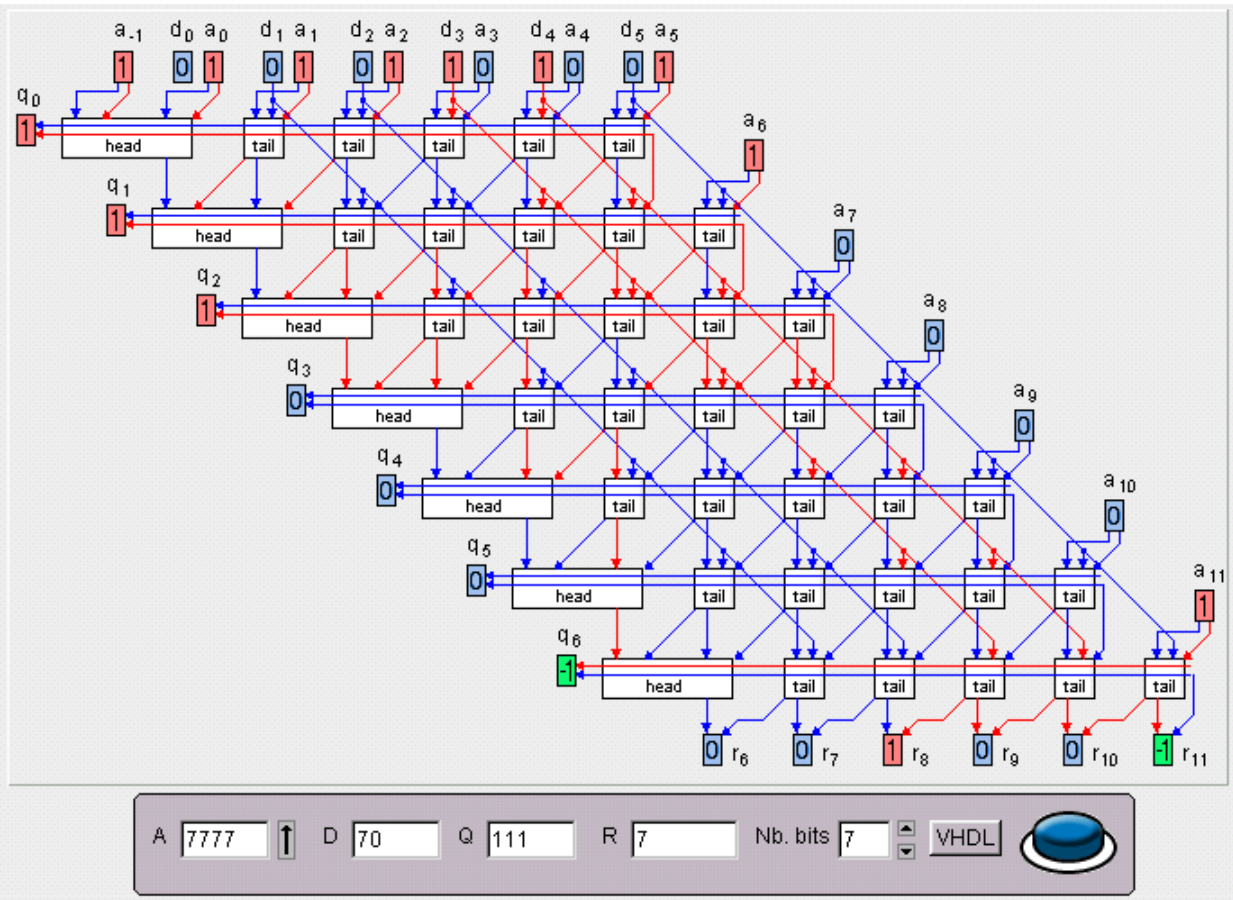
"SRT" division with divider range reduction

The previous division is simple because the fist bit of the divider D is always "1". It may be even further simplified if the <u>two first bits</u> d_0 and d_1 of the divider D are reduced to "1 0" thanks to the following operation:

if d_1 **then** { D = D ∗ 3/4 ; A = A ∗ 3/4 ; } .

This multiplication of A and D by the same constant does not alter the quotient Q, but on the other hand the final remainder R is also multiplied.

For an n-bit divider, $2^{n-1} - 1 < D < 2^{n-1} + 2^{n-2}$.



Head cell of the "SRT" divider with range reduction

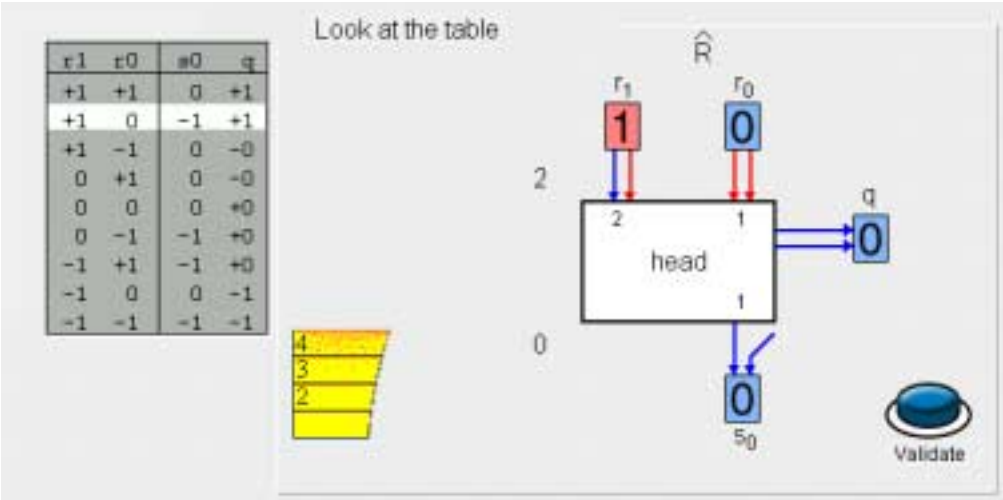
Let $\hat{R} = r_1 * 2 + r_0$ be the "head" cell input value.

- **if** $\hat{R} > 1$ **then** { $s_0 = \hat{R} - 3$; q = +1; }

reduction

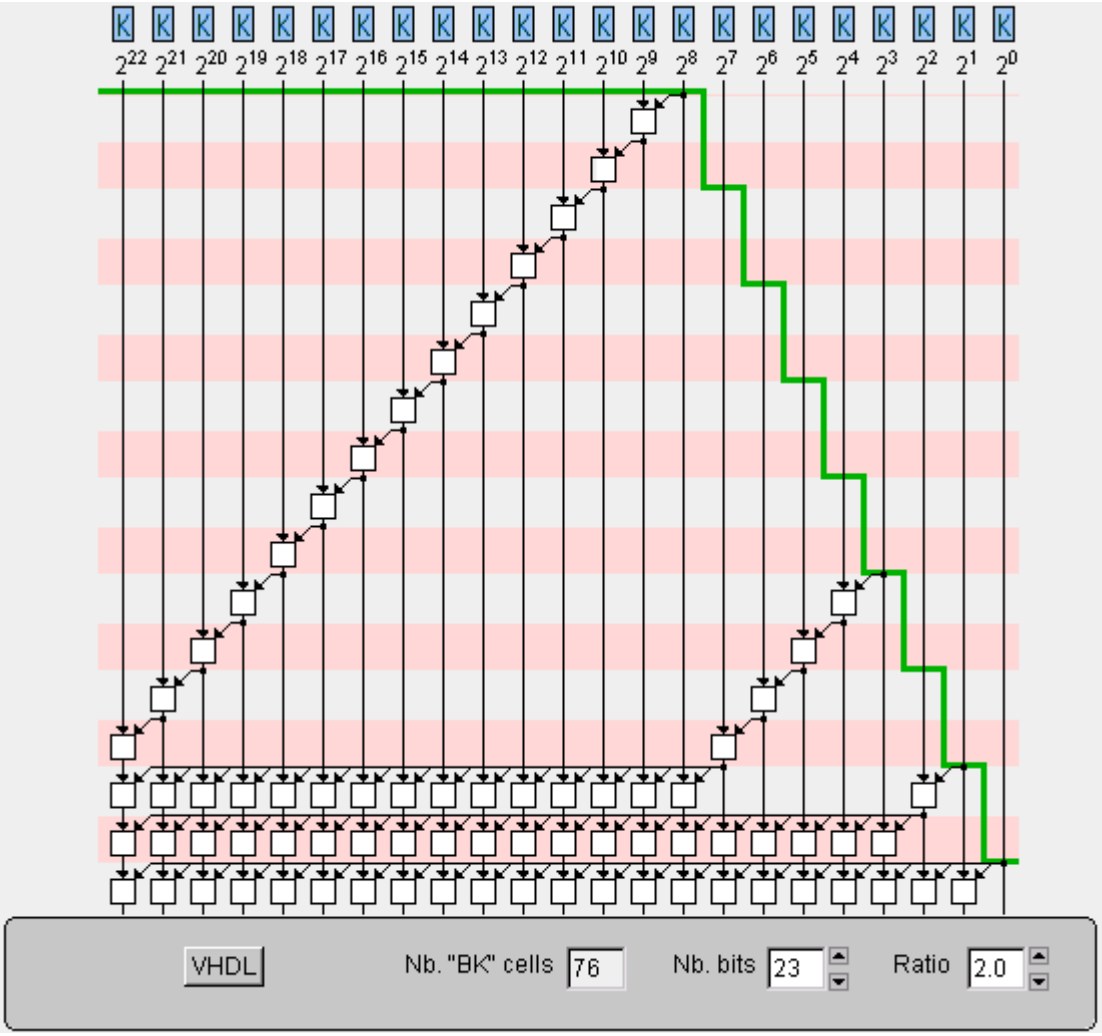
- **if** $\hat{R}$= 1 **then** { s_0 = 0 ; q = - 0 ; }
- **if** $\hat{R}$= 0 **then** { s_0 = 0 ; q = + 0 ; } *or* { s_0 = -1 ; q = - 0 ; }
- **if** $\hat{R}$= -1 **then** { s_0 = -1 ; q = + 0 ; }
- **if** $\hat{R}$< -1 **then** { s_0 = $\hat{R}$ + 2 ; q = -1 ; }

Here the difference between the two 0 representations for q : "- 0" and "+ 0" matters.

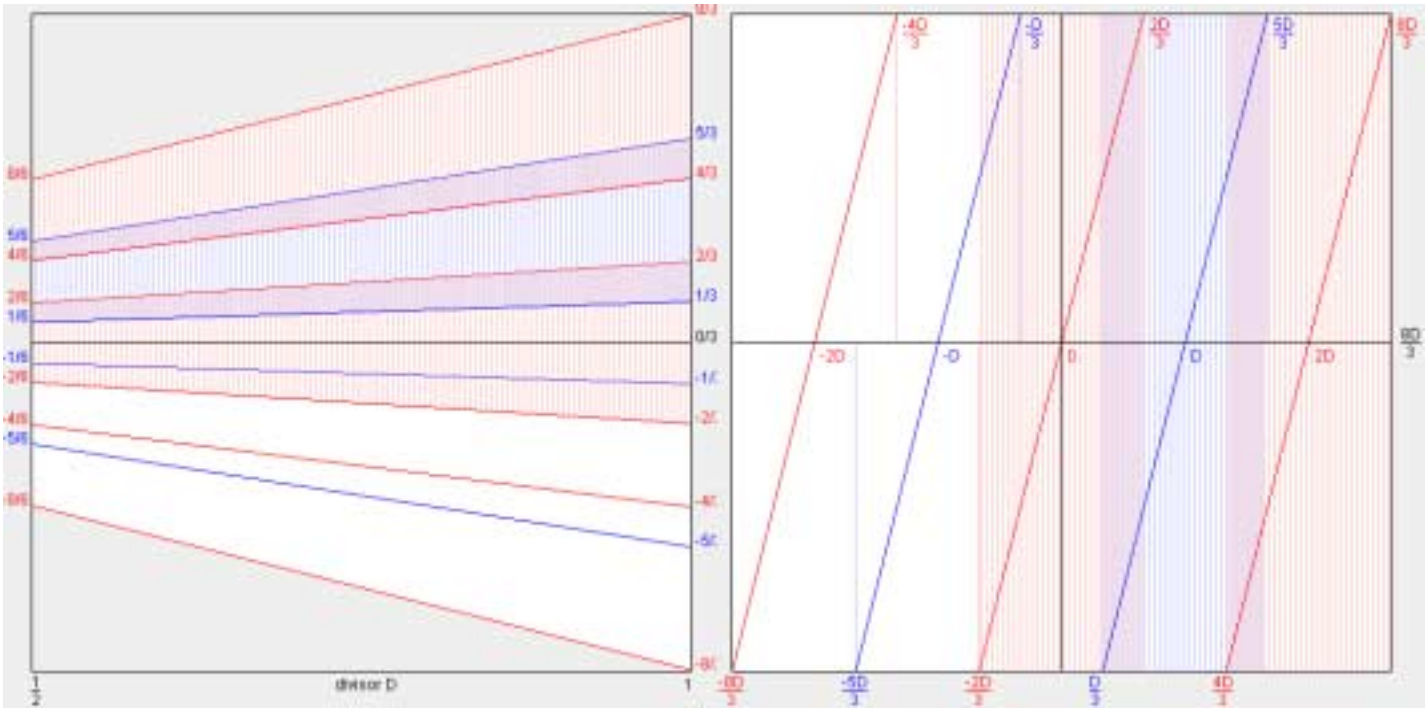


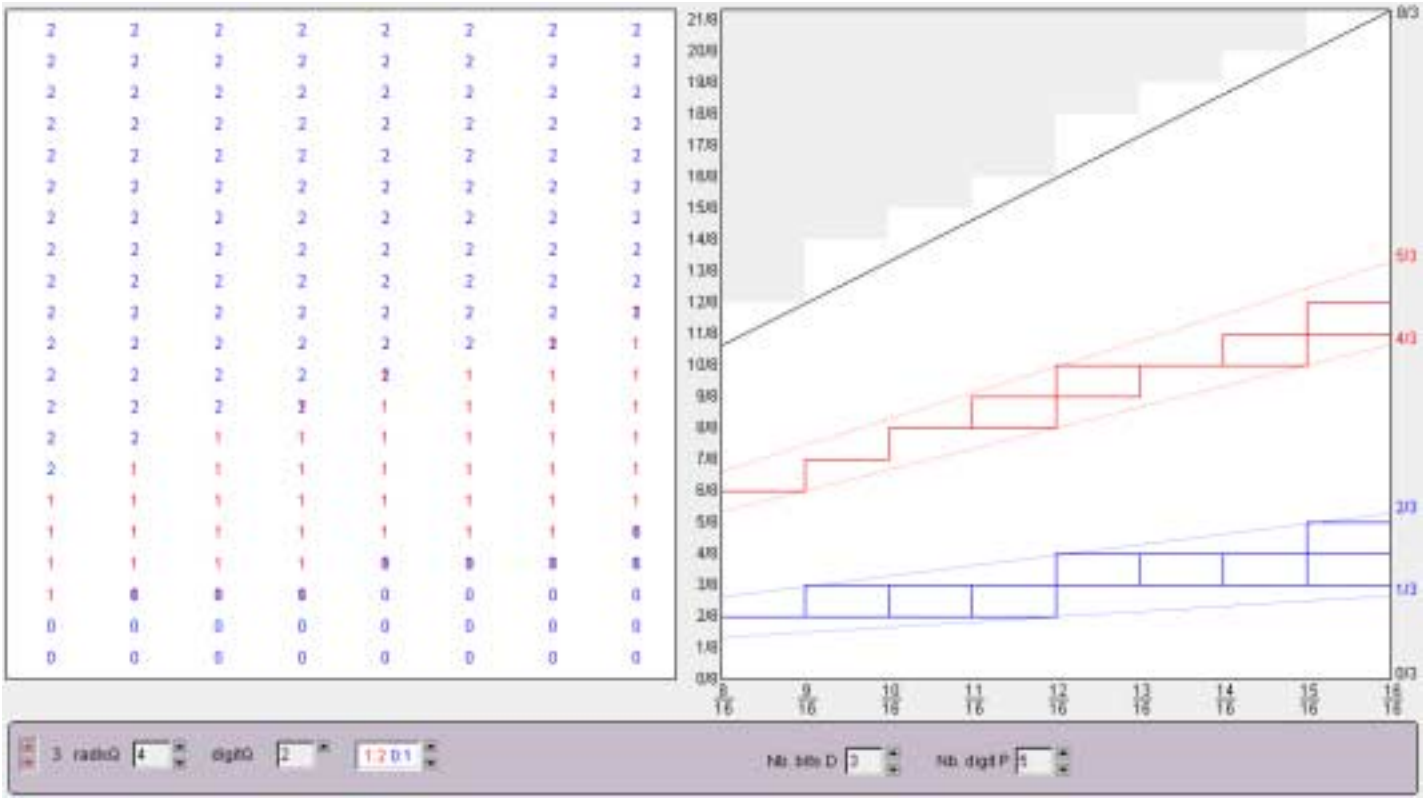
Quotient converter

The quotient Q is in redundant notation. The conversion into a conventional binary representation is obtained thank to an adder (in fact a subtractor). Since the digits q are obtained sequentially, most significant digit first, the conversion can be carried out in parallel with the quotient digits obtaining. Let "ratio" be the "head" cell and "BK" cell delays ratio. The higher the ratio, the simpler the converter.



Divider design The quotient Q digits are redundant and symmetrical. Therefore they are completely defined by the radix and the maximum digit value. This applet let you choose the quotient digit values and then the necessary number of bits from divider D and digit from partial remainder R that must be taken into account for the selection of the quotient digit value.





The leftmost button moves to the next or to the previous step.

- **1-** Robertson's diagram, plotting the next partial remainder according to the current partial remainder. The boundaries take divisor D into account.
- **2-** Symmetrical PD-plot for D in the range [1/2 , 1 [
- **3-** Half PD-plot, upper part of the preceding one. The lower part is obtained by changing the sign.
- **4-** Discretised half PD-plot. The number of D bits taken into account gives the abscissa discretisation, while the number of P digits gives the ordinate discretization. Fixing this number of bits/digits will determine whether a continuous frontier can separate the different values of q.
- **5-** Half truth table for the half PD-plot.

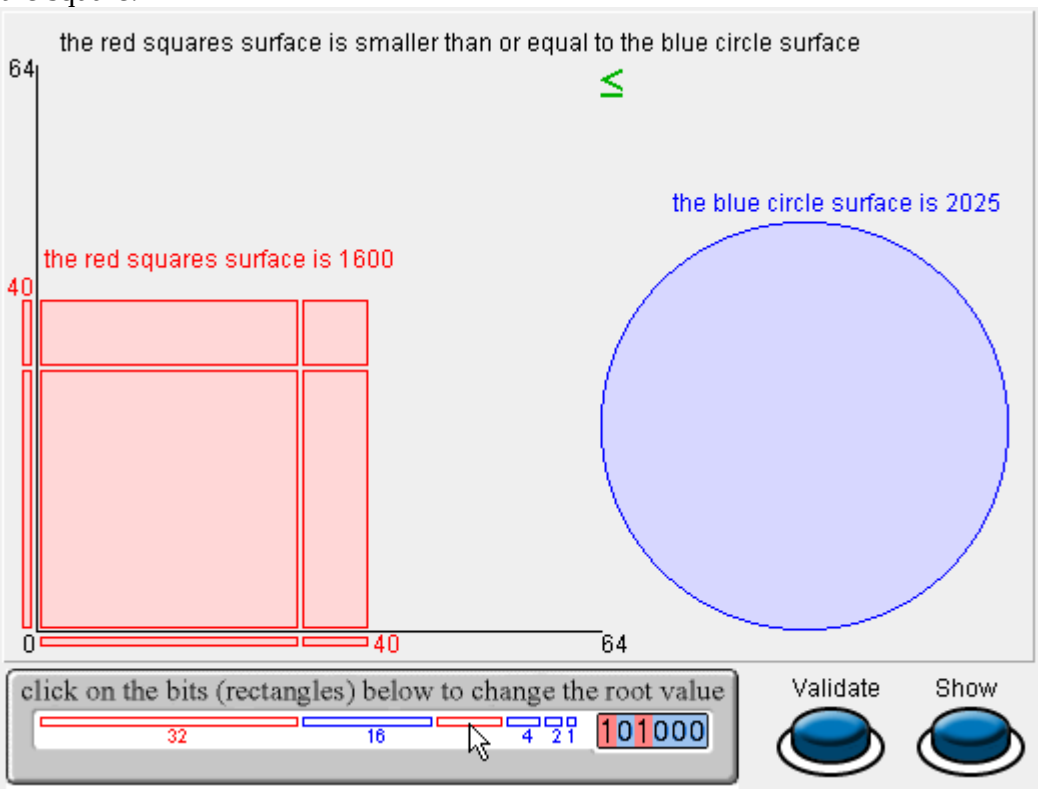
Square root extractor

Square root extraction

The square-root extraction is relatively rare. Nevertheless, it is used among other things for Euclidean distance and least square and is included in the floating-point standard. The square-root operator is similar to the one for division, therefore most of what we already know about division may apply as well to square root. Often the same operator is used either for division or for extraction, collision of the two operations being too rare to justify two operators, moreover each very costly.

Square root extraction algorithm

In the drawing below, the area of each red rectangle represents one bit weight. Only the '1' are drawn. Therefore the total area is the weighted sum of all the bits.
The goal of the game is to find a square with an area equal to a given argument, represented by the area of a blue circle, just by observing one test bit (≤ or >) and by clicking the square side bits. After convergence the sough after number is the side of the square.



Square root extractor

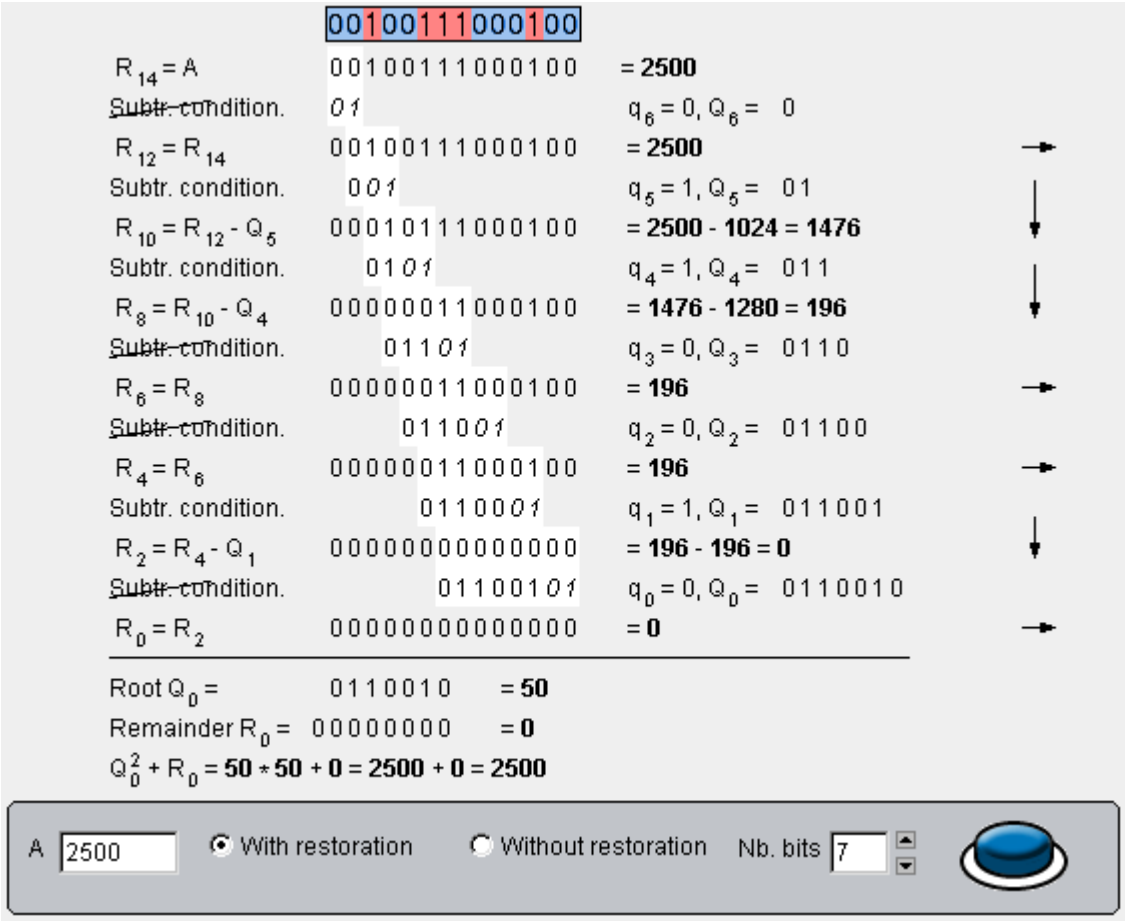
We want to get $Q = \sqrt{A}$. This is equivalent to $Q = A \div Q$. Therefore if Q is written on n bits, A is written on 2n bits.
Let us build a series Q_n, Q_{n-1}, ... Q_2, Q_1, Q_0 and a series R_{2n}, R_{2n-2}, ... R_4, R_2, R_0 such that the invariant $A = Q_j * Q_j + R_{2j}$ holds for all j.
The recurrence is:

- $Q_{j-1} = Q_j + q_{j-1} * 2^{j-1}$
- $R_{2j-2} = R_{2j} - q_{j-1} * 2^{j-1} * (2 * Q_j + 2^{j-1})$

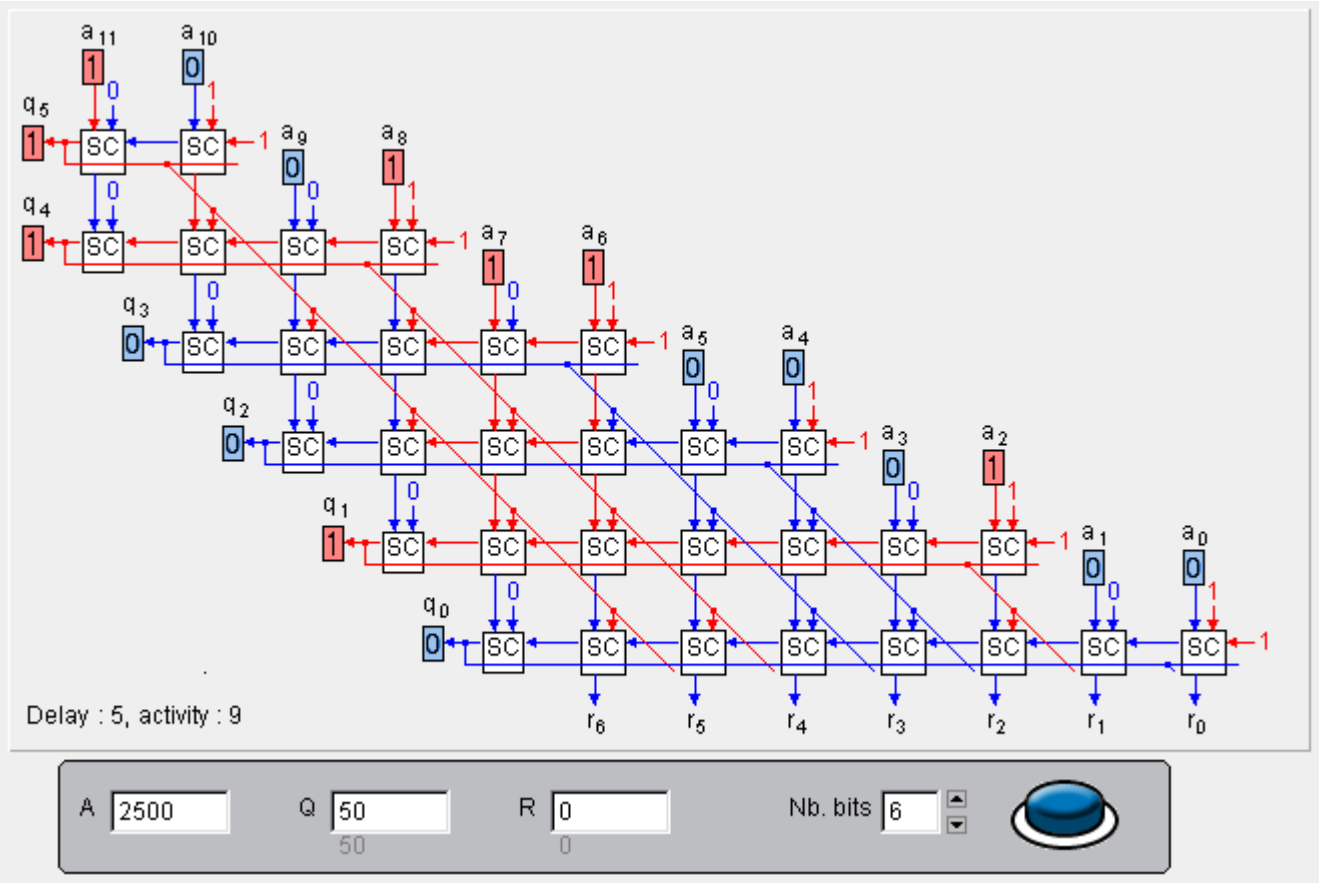
with the initial conditions:

- $Q_n = 0$
- $R_{2n} = A$.

When the recurrence ends, we have $Q = Q_0 = \sum_{i=0}^{n} q_i * 2^i$.
$R = R_0$ is the final remainder of the square root extraction.



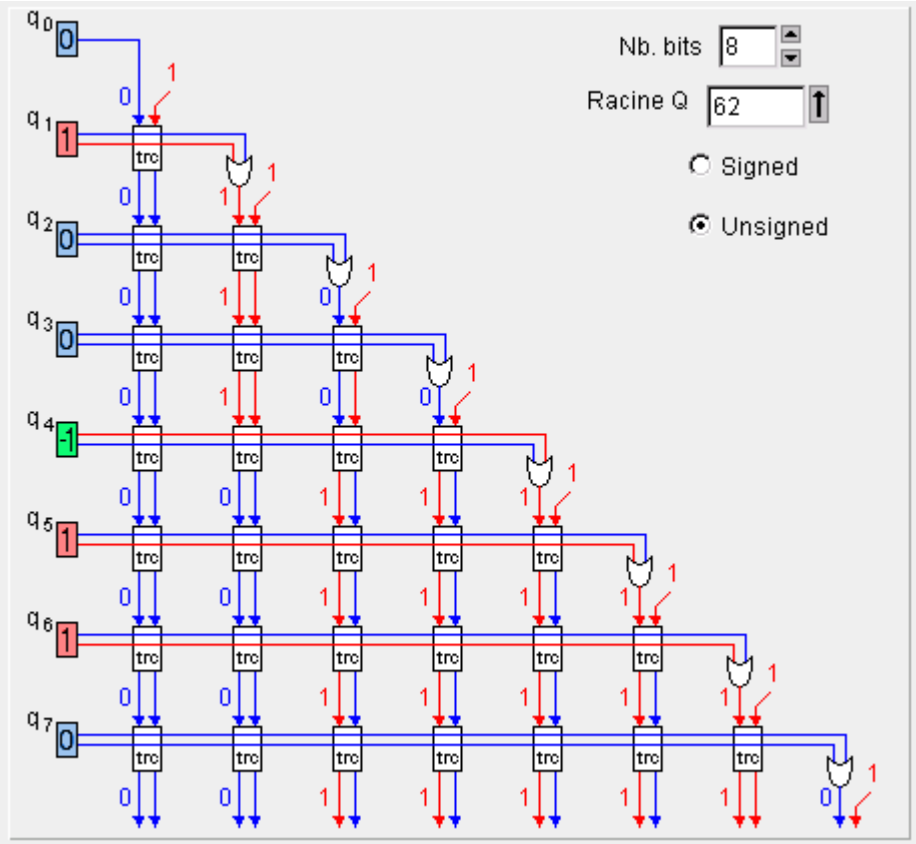
Realization The restoring square-root extractor utilizes the same conditional subtractor "SC" cells as the non-restoring divider.



Fast square-root We want to get rid of the carry propagation by the use of the "BS" notation, the same

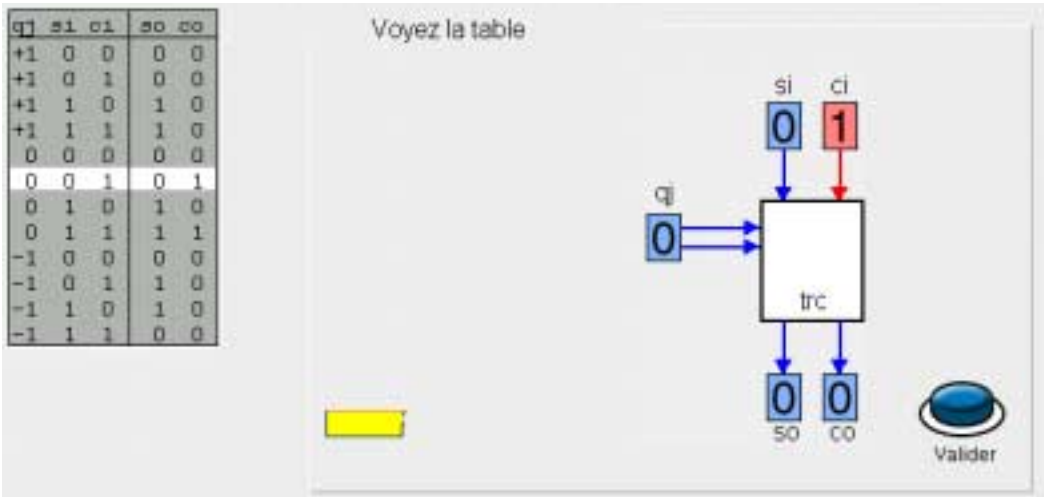
extractor "head" and "tail" cell and architecture similar to the fast division. We bump into three difficulties when trying to use the fast divider for extraction of square roots.

Square root converter The first difficulty is the root feedback. In a similar way as the division, the extractor supplies a partial root Q_j in "BS" notation. On the other hand, the "head" and "tail" cell of the divider accepts a partial root in conventional binary representation. A subtractor could be used to convert each Q_j from "BS" to conventional, but that would be both slow and expansive. The converter below use a 4-input, 2-output "trc" cell, derived from the "BK" cell.



Square root conversion cell Check whether you are acquainted with the logic of "trc" cell, which convert from "BS" notation into standard binary notation.
Input "si" is a bit from Q_j, input "ci" indicates whether the carry propagates in this cell's position. The carry is used in case of subtraction of 1. This signal corresponds to the value 'P' of the "BK" cell.

- **if** q_j = -1 **then** { so = si ⊕ ci ; co = 0 }//subtraction (sum - carry), carry killed
- **if** q_j = 0 **then** { so = si ; co = ci } //sum unchanged, carry propagated
- **if** q_j = 1 **then** { so = si ; co = 0 } //sum unchanged, carry killed



qj	si	ci	so	co
+1	0	0	0	0
+1	0	1	0	0
+1	1	0	1	0
+1	1	1	1	0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
-1	0	0	0	0
-1	0	1	1	0
-1	1	0	1	0
-1	1	1	0	0

Carry-propagation free square root

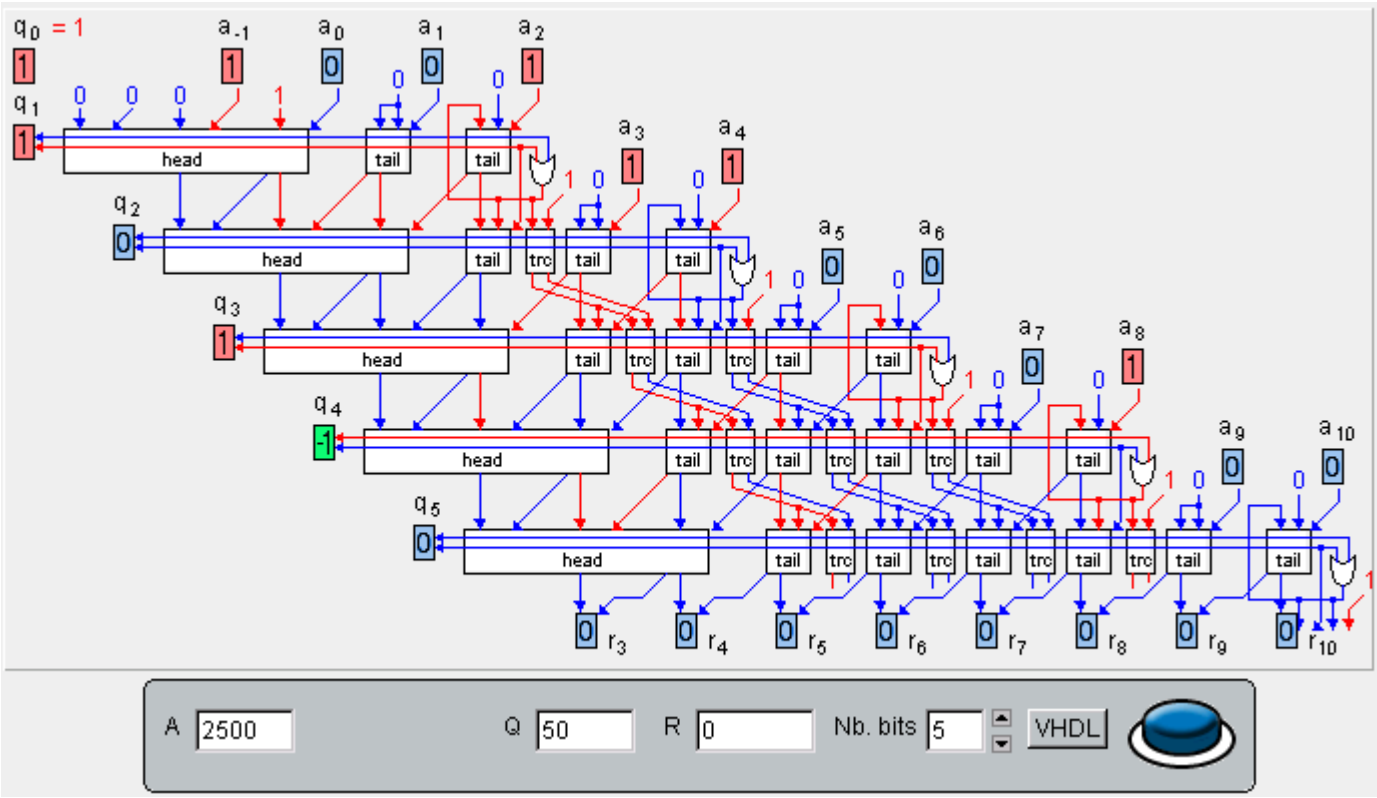
The fast square-root extractor utilizes the same cell as the fast divider to execute at each step one of the three following arithmetic operations:

- **if** q_j = '-1' **then** $R_{2j-2} = R_{2j} + 2^j * Q_j - 2^{2j-1}$ // addition
- **if** q_j = '0' **then** $R_{2j-2} = R_{2j}$ // identity
- **if** q_j = '1' **then** $R_{2j-2} = R_{2j} - 2^j * Q_j - 2^{2j-1}$ //subtraction

Each "head" cell selects the value of one q_j thanks to the sign of an approximate $\hat{R}_{2j}$ of the current remainder R_{2j}.

The second difficulty with respect to division lies in the subtraction of 2^{2j-1} whenever q_j = -1 or q_j = 1. For the bit subtraction, a negative input of the least significant tail cell of each line is used.

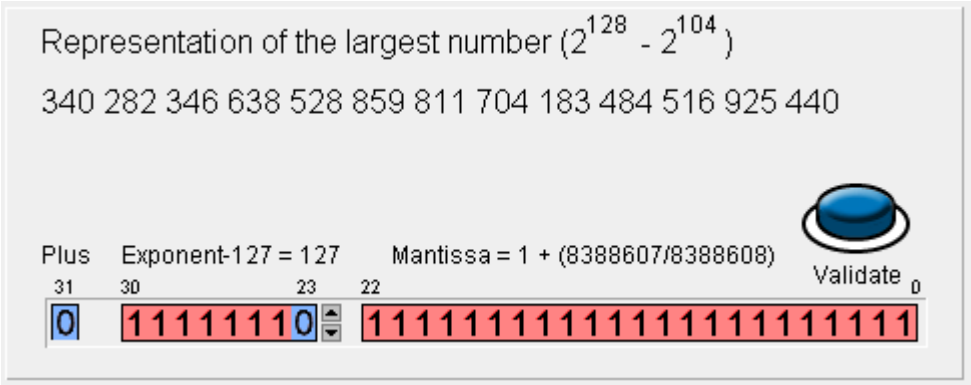
The third difficulty lies in the range of Q. Indeed each Q_j must start with a "1" in the most significant position (implicit). This condition is fulfilled if the two most significant bits of the radicand A are not both zero. This "1" is subtracted from A in the first line thanks to a negative "head" input



Floating-point addition

Floating-point numbers format

The binary code of floating point real numbers is composed of three fields. The sign S (1 bit), the exponent E (8 bits) and the mantissa M, or significand (23 bits). The number value is $(-1)^S * 2^{(E - 127)} * (1 + M / 8388608)$. However if E = 0, the number value is $(-1)^S * 2^{(-126)} * (M / 8388608)$ and if E = 255, the value is infinite.
Check your understanding of this format by entering the code (32 bits) of the proposed numbers.

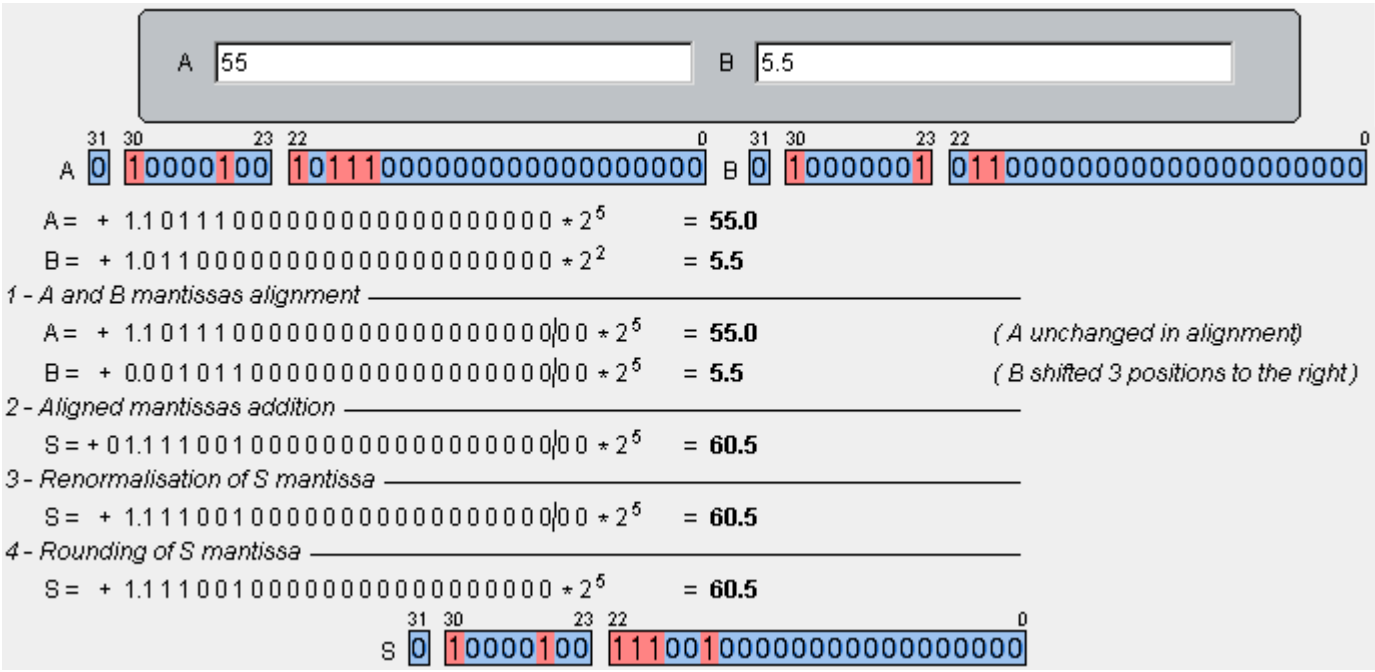


Addition and subtraction

Since real numbers are coded as "sign/absolute-value", toggling the sign-bit inverses the sign. Consequently the same operator performs as well either addition or subtraction according to the operand's sign.
Addition/subtraction of two real numbers S = A + B is more complex than multiplication or division or real numbers.
Floating-point addition progresses in 4 steps:

- Mantissa alignment if A and B exponents are different,
- Addition/subtraction of the aligned mantissas,
- Postnormalization of the mantissas sum S if not already normalized,
- Rounding of sum S.

The alignment step yields a guard bit and a sticky bit for the rounding step .



Adder/ subtractor

A floating-point adder is made of the following blocks:

Bloc 1: outputs the larger of the two exponents (8 bits), outputs the exponent distance (5 bits), outputs the implicit bit of both operands.

Bloc 2: outputs at left the smaller operand mantissa (23 bits), outputs at right the larger operand mantissa (23 bits).

Shifter 1: shifts to the right the smaller operand mantissa, adds the guard bit and the sticky bit, totaling 26 bits.

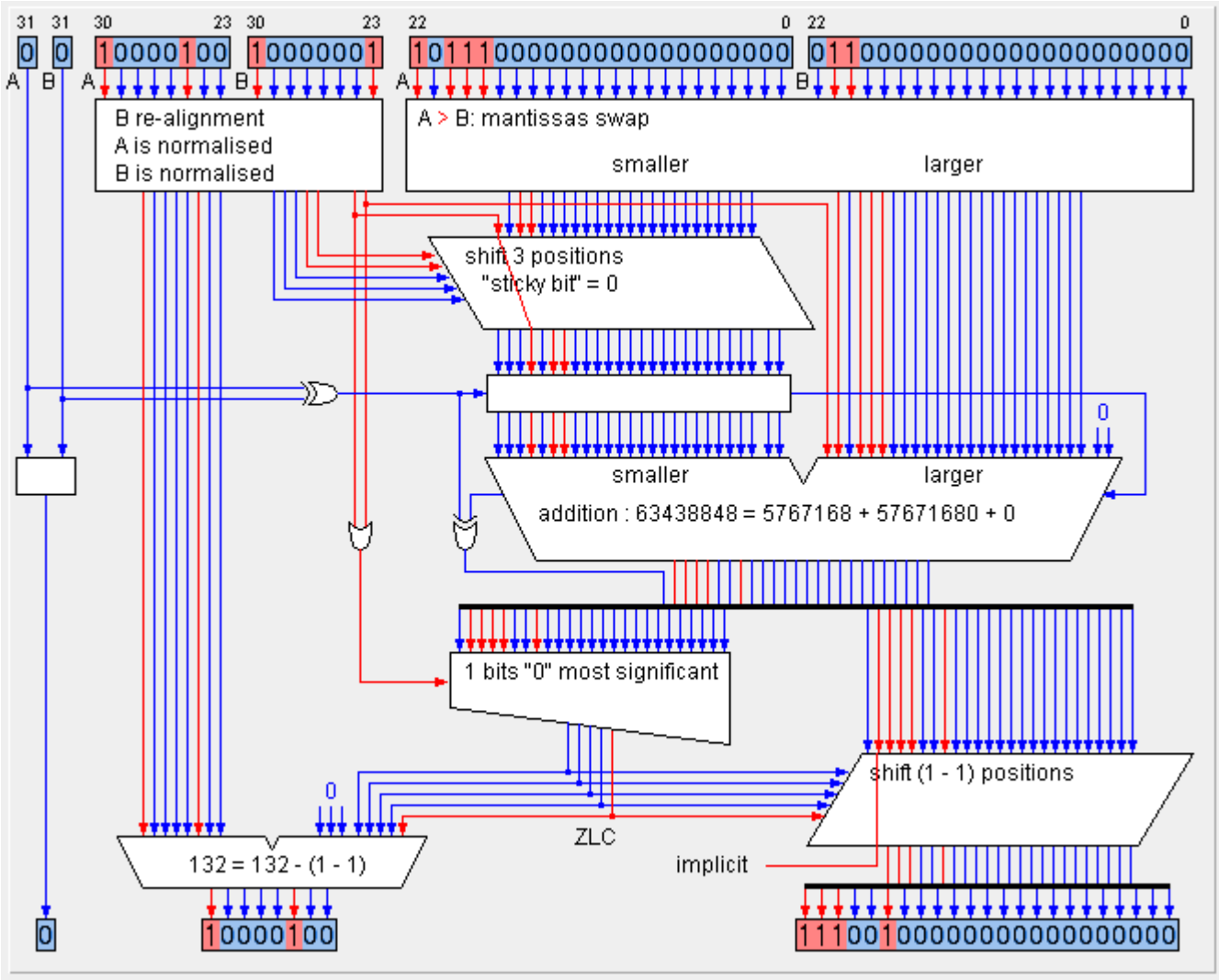
Complementer: on request, does the logic complement for a subtraction.

Adder 1: adds the two inputs and the carry in. outputs the rounded sum and a carry out.

Zero-leading-counter: the ZLC output gives the number of leading '0' if the result is not normalized, and "1" otherwise.

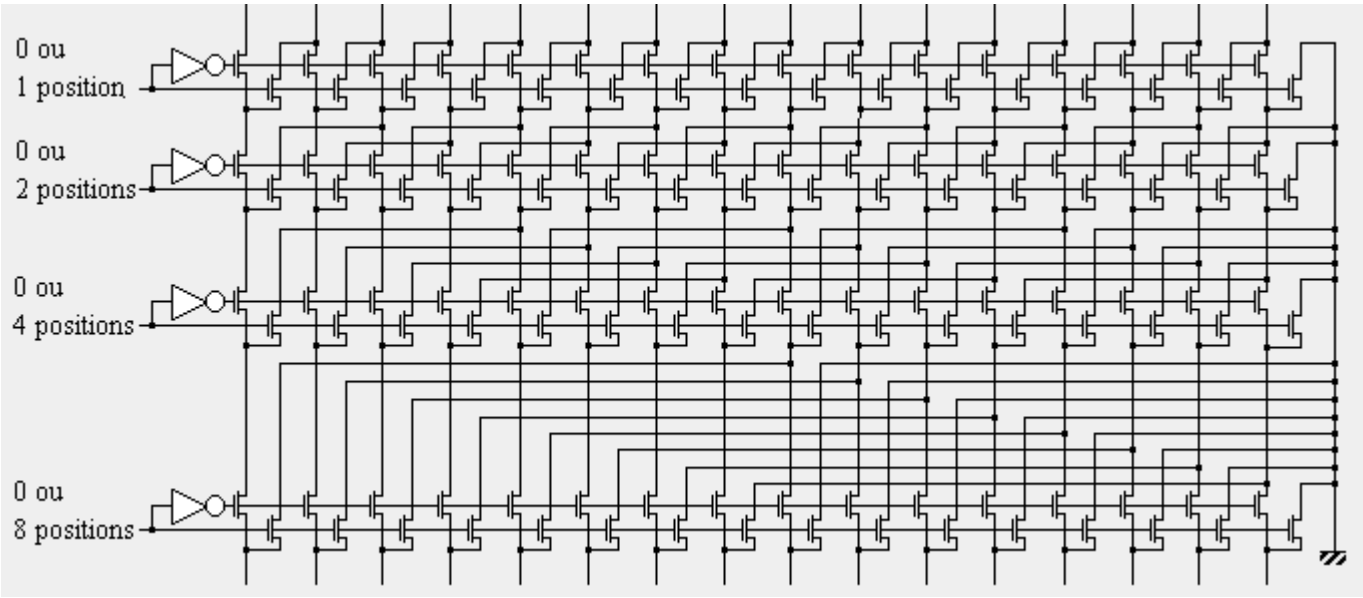
Shifter 2: shifts to the left (ZLC – 1) positions. The fist bit is lost (implicit '1').

Adder 2: subtracts (ZLC – 1) from the greater of the two exponents.



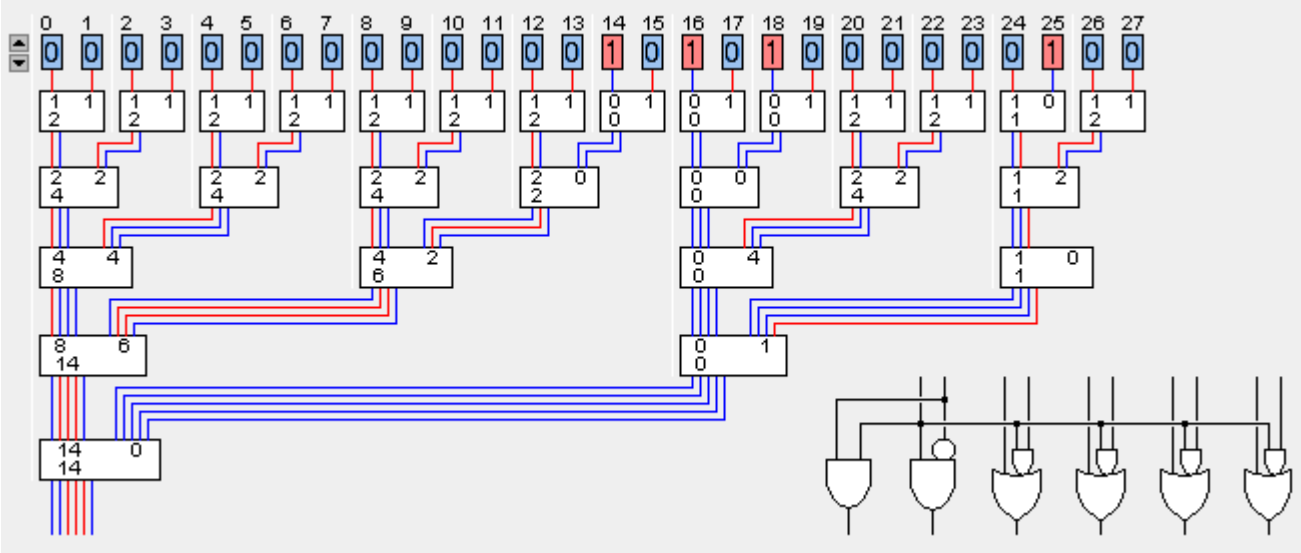
Real numbers fast addition

A floating-point numbers addition requires some integer additions, parametrised shifts (to the right for alignment, to the left for renormalization) and a counting of the leading zeroes of the result. Addition can be completed with delay $\log_2(n)$. Parametrised shift delay is $\log_2(n)$ as well.



Zero leading counter (ZLC)

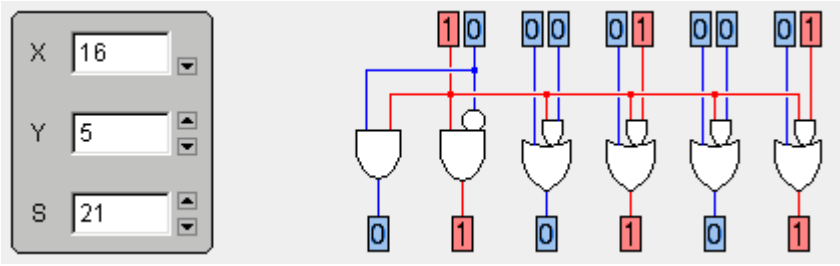
A binary tree counts up the number of '0' in the most significant positions by dichotomy. If the size of the sub-strings is a power of two, then there is no need for adders but multiplexors can be used instead. Indeed only the size of the left subsrting has to be a power of two. The substring at right must simply be shorter than or of the same size as the left subsrting.



Zero leading counter cell

This cell combines the number of leading '0' of two 16-bit strings to obtain the number of leading '0' of the concatenation of the two strings.

- **if** X < 16 **then** S = X **else** S = 16 + Y

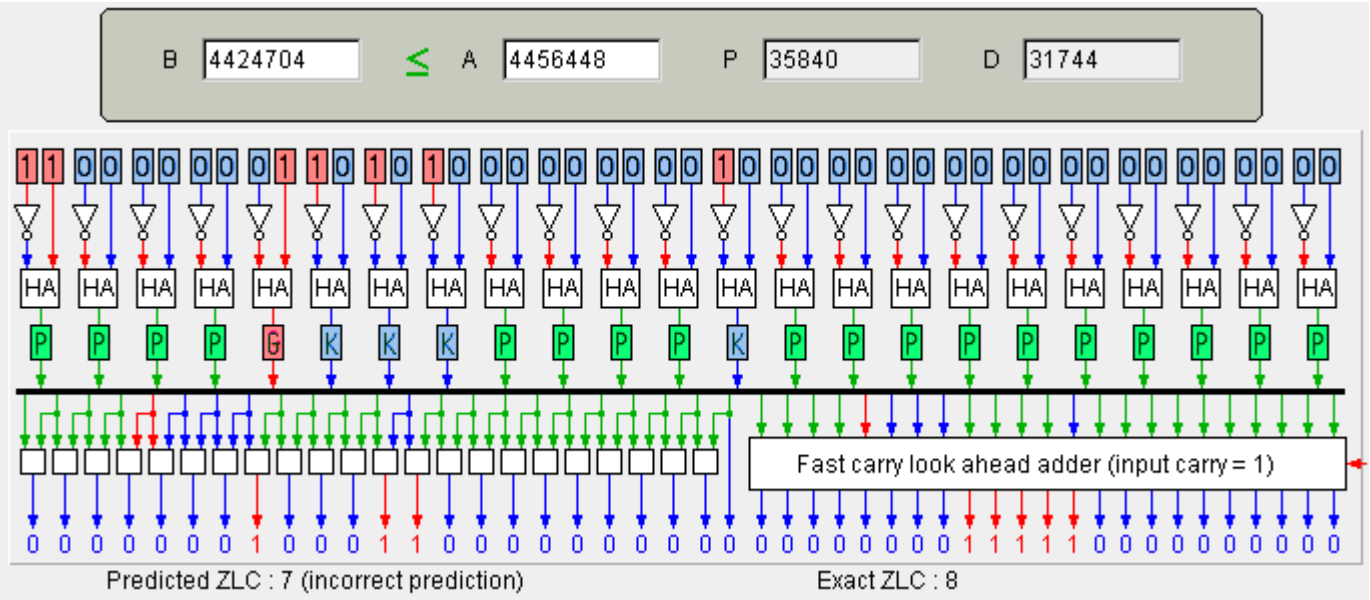


Zero leading prediction

From the mantissas A and B, one can construct in constant time a string P with the same number of leading zeroes, but for at most one, as the result of the difference D = A – B with no need to wait for the subtraction completion. When fed to a ZLC, this string predicts the number of positions required by the shifter. If the result of the shift still exhibits a leading zero, then a shift of one more position is necessary to normalize the result. Otherwise the shifted value is normalized.

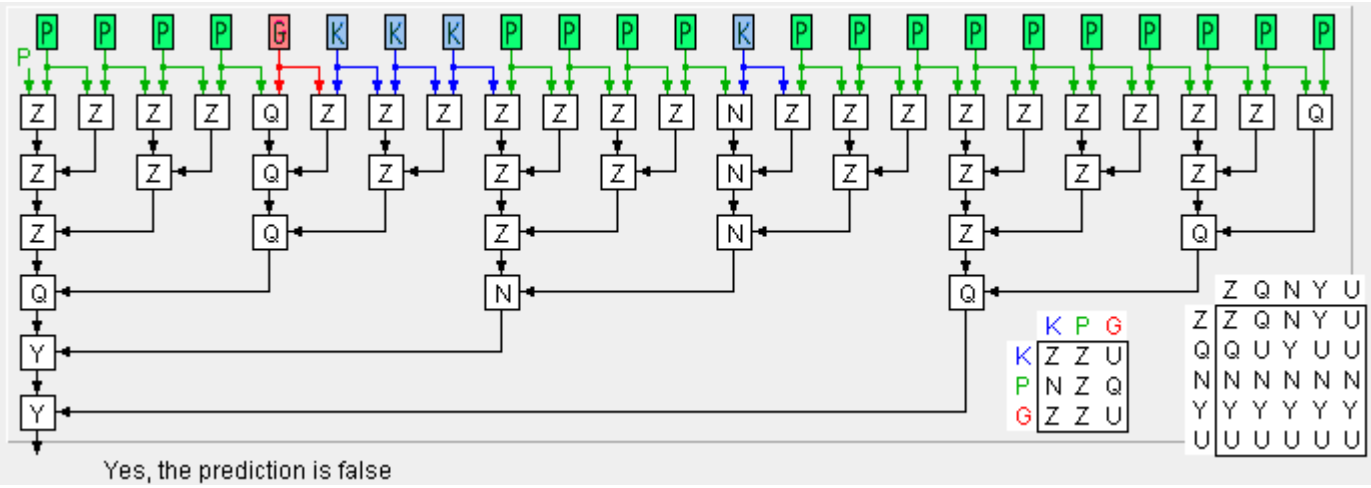
The prediction is valid if A is normalized and B less than or equal to A. This is the always the case in a significand subtraction. The leading zero(es) result from a carry string 'P'* 'G' 'K'* , made up with a number (possibly null) of 'P' followed by an unique 'G' followed by a number (possibly null) of 'K' . The predictor cell outputs a '0' for every pair of symbols in: 'P' 'P' ; 'P' 'G' ; 'G' 'K' et 'K' 'K' and outputs a '1' for every other pair.

This predictor does not take into account the carry propagation that may lead to an error of one position in the predicted bitstring. Since only one bit in 'P'* 'G' 'K'* might be incorrectly predicted, the error is tolerable.



Zero leading prediction adjustment

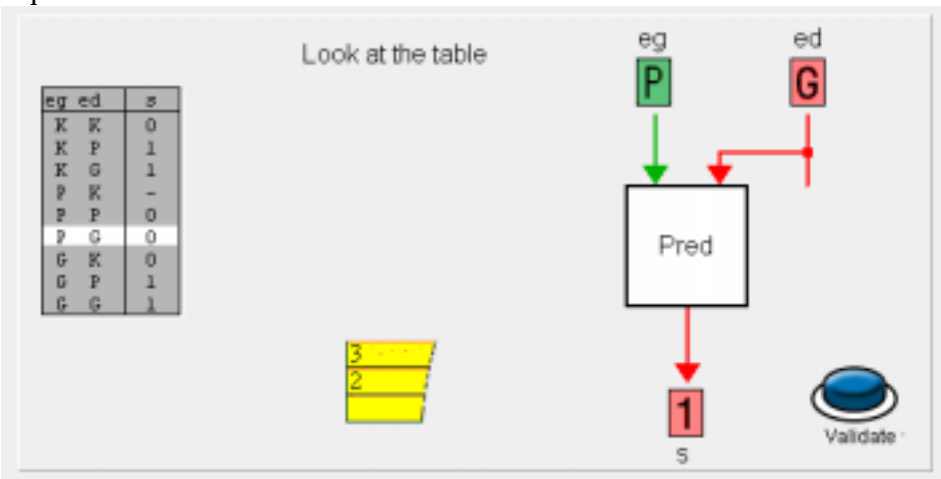
The prediction is incorrect only if the carry string starts with 'P'* 'G' 'K'* 'P'* 'K'. The following circuit output 'Y' whenever the prediction is incorrect, in other words too small by one.



Z indicates a string 'K'* 'P'*

Q indicates a string 'P'* 'G' 'K'* 'P'* (containing only one 'G')
N indicates a string starting with 'P'* 'K'
Y indicates a string starting with 'P'* 'G' 'K'* 'P'* 'K', that is Q followed by N.
U indicates any other string.

Prediction cell The leading '0' prediction cell output a '1' at the end of the string 'P'* 'G' 'K'* and '0' inside the string (and don't care neither inside nor at the end). Check whether you are acquainted with the truth table of this cell.



Elementary Functions

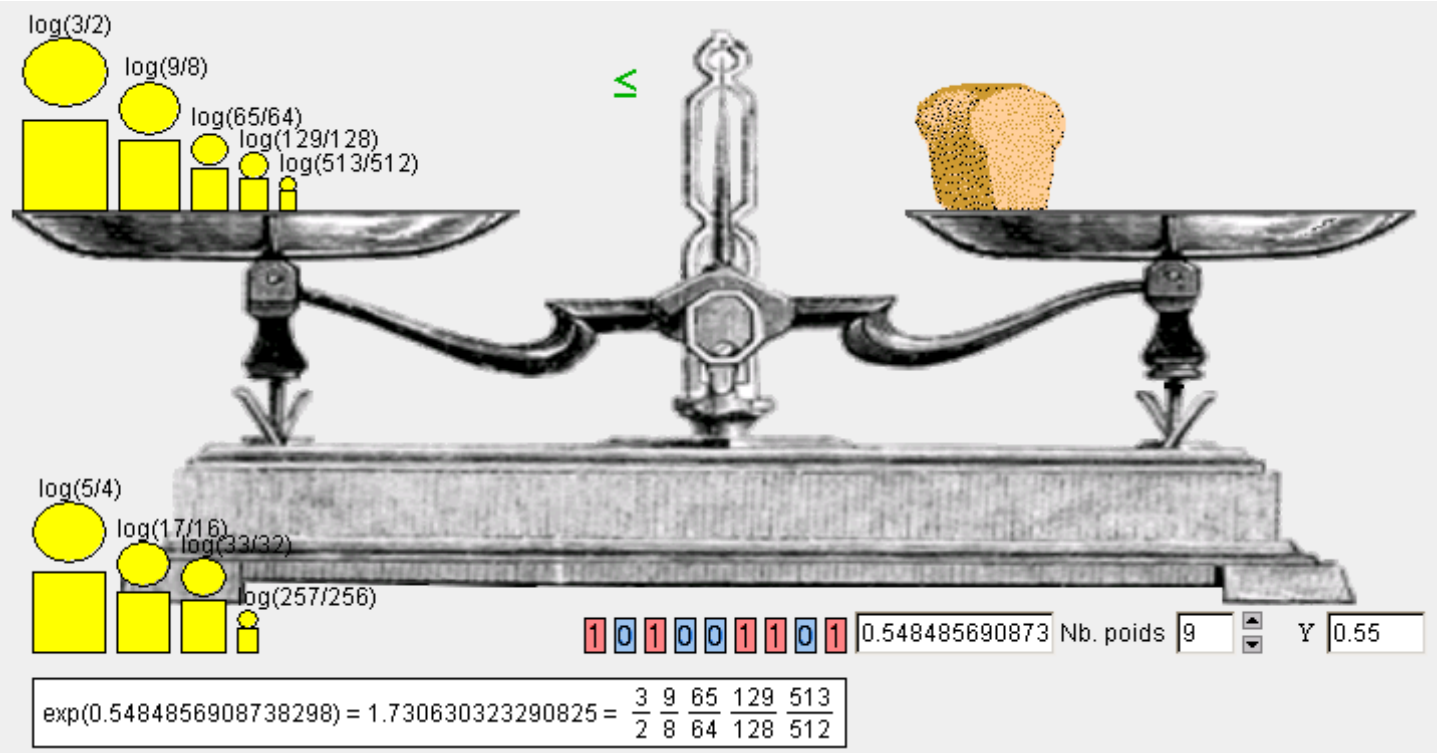
Elementary functions

Realization of operators for Exponential, Logarithm, Sine, Cosine, arcTangent relying on addition/subtraction and fixed shift. The cost and delay of fixed shits are negligible when they are wired.

From a bread loaf weighting to the exponential computation

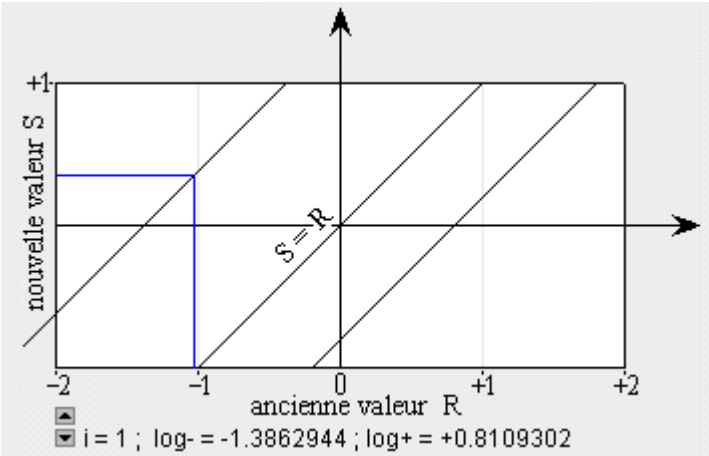
We want to compute exp (Y), we have available a scale, a white bread whose weight is actually just Y and a set of weights with values $\log(1 + 2^{-i})$. The weighting gives the sought-after result in the form of a product of rationals $(2^i + 1) / 2^i$. The multiplication by each rational amount to a mere addition and shift.

A weight put down on the right plate (the bread's one) has it value changed into $-\log(1 - 2^{-i})$. Thank to this trick, the weighting can be restoring, non-restoring or "SRT".



Carry propagation free division for exponential

The scale is replaced by a "SRT" divider whose "Robertson's diagram" is drawn below

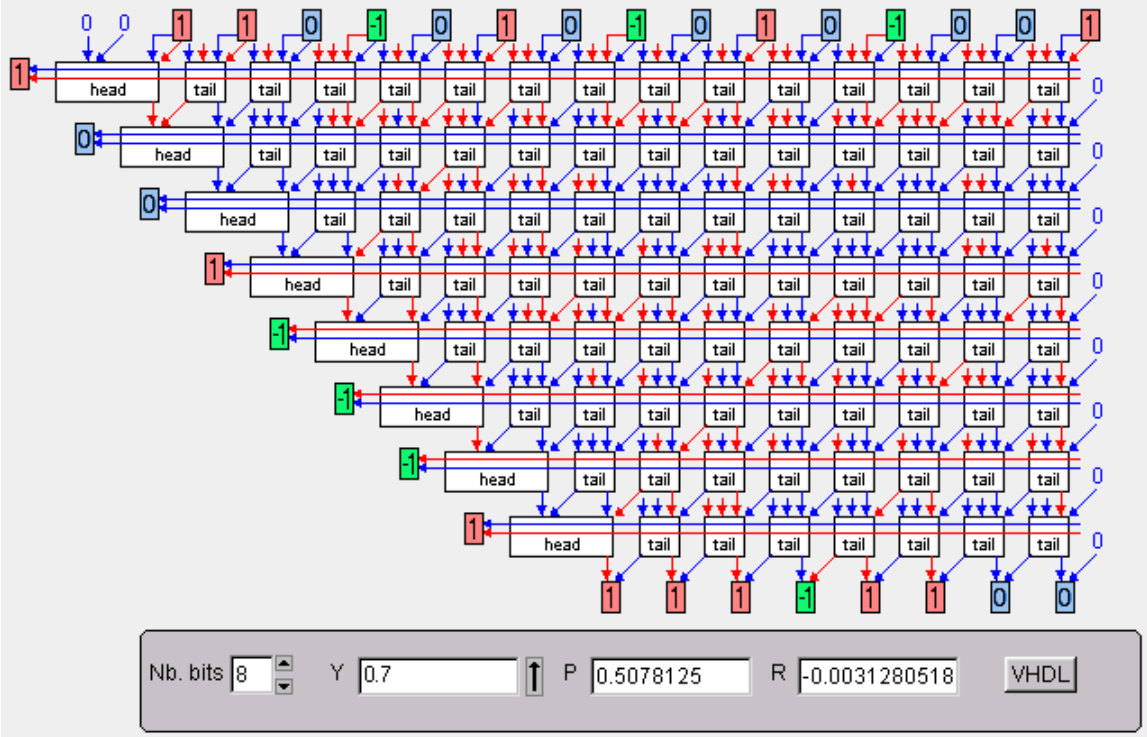


The applet gives all the successive partial remainders.

The dividend Y (top) must be within] -1 , +1 [.

"SRT" divider for exponential

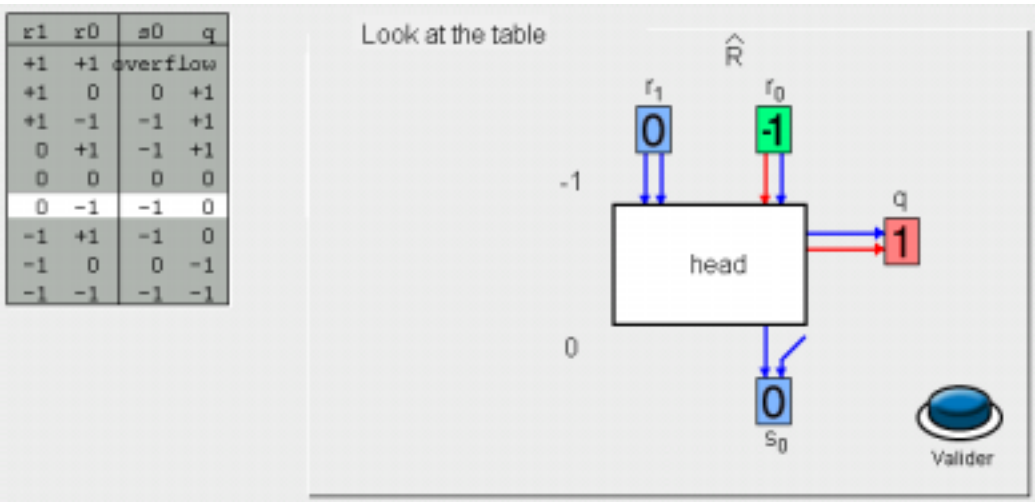
The constants $\log(1 + 2^{-i})$ and $-\log(1 - 2^{-i})$ fed into the "tail" cells are wired . Thus there are 4 variants for the cell according to the values of the two bits.



Operations of a slice of "SRT" divider for exponential

The value of each q_j is selected by a "head" cell according to $\hat{R}_j$, the weighted sum of the two most significant digits r_1 and r_0 of the representation of R_j.

- **if** $\hat{R}_j > 0$ **then** { q_j = '1' ; $s_0 = \hat{R}_j - 2$; $R_{j+1} = R_j + \log(1 - 2^{-j})$ } // subtraction
- **if** $\hat{R}_j = 0$ **or** $\hat{R}_j = -1$ **then** { q_j = '0' ; $s_0 = \hat{R}_j$; $R_{j+1} = R_j + 0$ } // identity
- **if** $\hat{R}_j < -1$ **then** { q_j = '-1' ; $s_0 = \hat{R}_j + 2$; $R_{j+1} = R_j + \log(1 + 2^{-j})$ } //addition

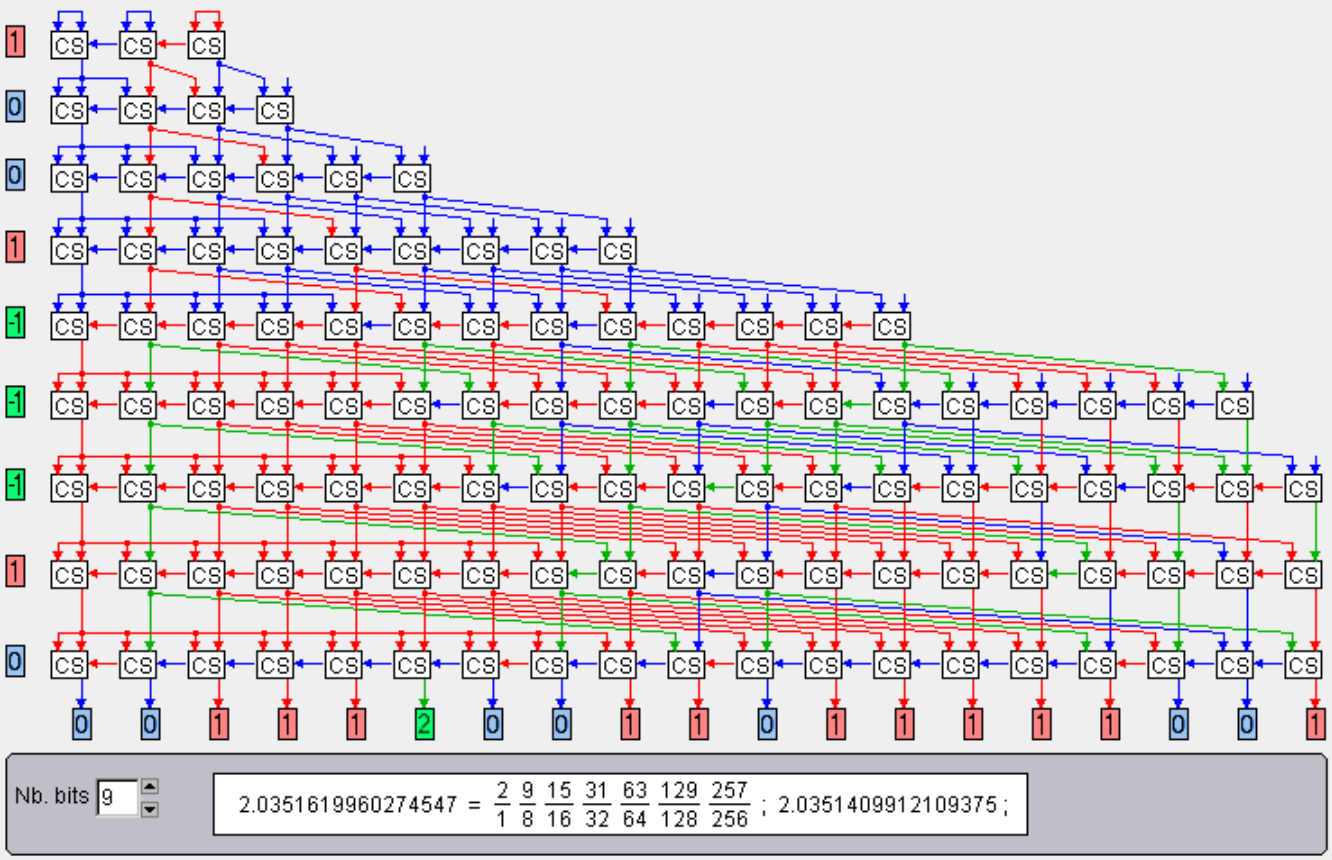


r1	r0	s0	q
+1	+1	overflow	
+1	0	0	+1
+1	-1	-1	+1
0	+1	-1	+1
0	0	0	0
0	-1	-1	0
-1	+1	-1	0
-1	0	0	-1
-1	-1	-1	-1

Suite de multiplications

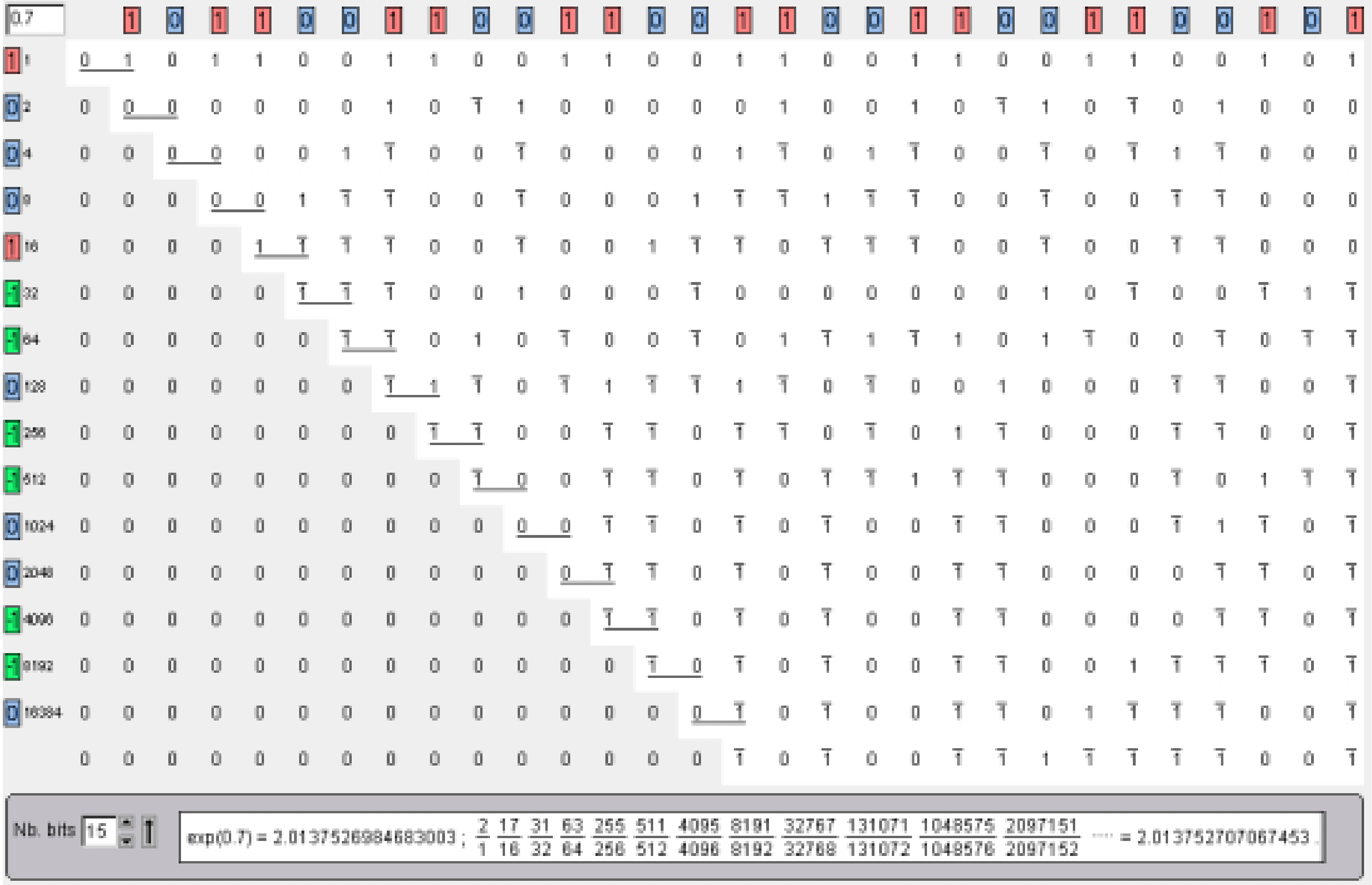
The stack of conditional multipliers by 1 or by $(1 + 2^{-i})$ or by $(1 - 2^{-i})$ needs only one final carry propagation thanks to "CS" adders and wired shifts.
Additions are truncated to 2 n digits, of which two before the point. The third most significant digit (fully left) is the sign. Despite the fact that all partial results are positive, the execution of subtraction in "CS" sometimes brings about an unresolved sign. The final result (bottom line) must be converted from "CS" to binary by one

addition (with carry propagation).
La fenêtre du bas permet de comparer le produit "vraie" des multiplications (sans troncature) au produit avec troncature.



Numerical example of division

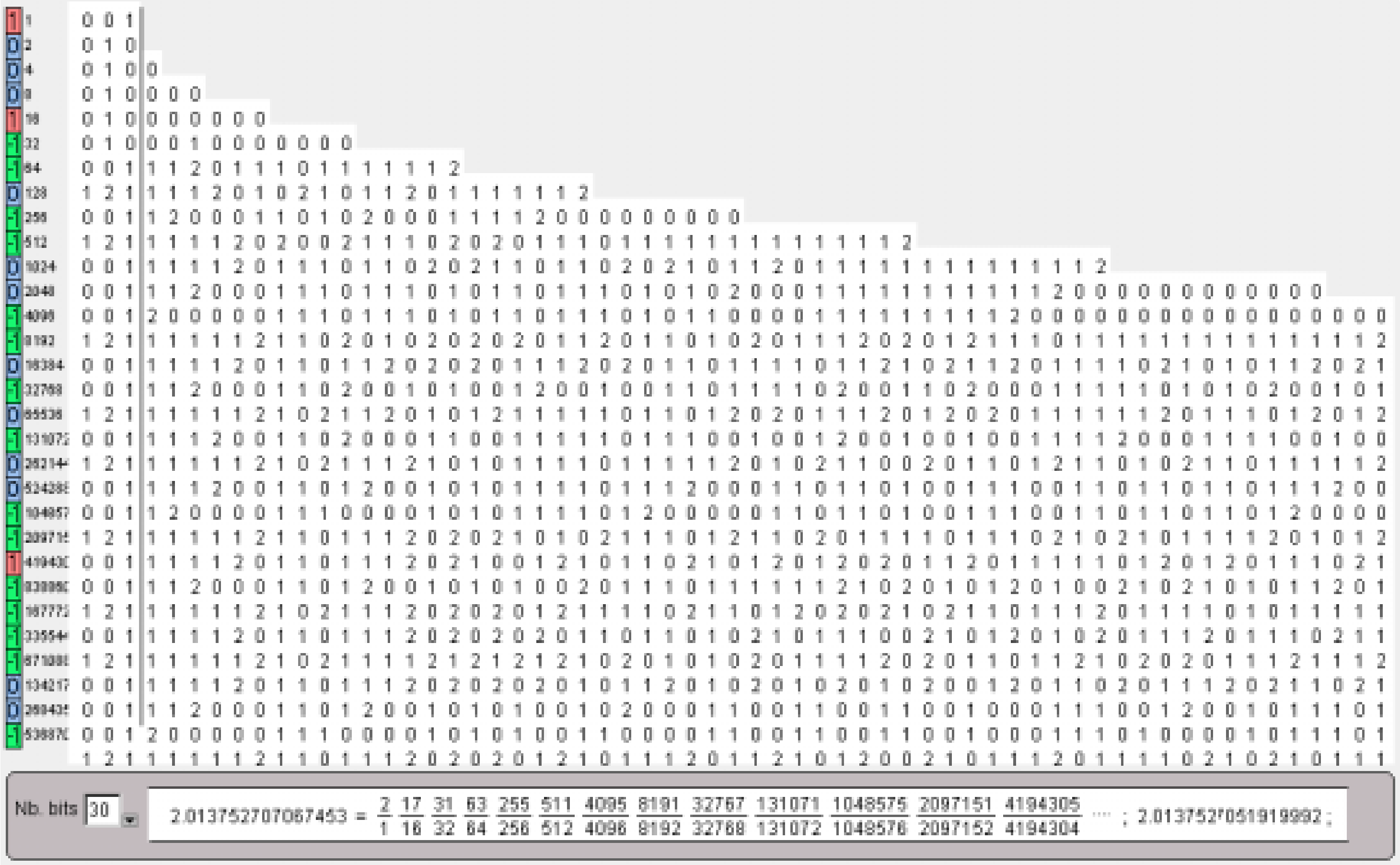
The tables below shows the partial remainders ("BS") and the partial products ("CS"). The windows at the tables bottom gives the actual value of the function, the product of rationals $(1 + 2^{-i})$ or $(1 - 2^{-i})$.and finally the truncated product of rationals to exhibit the errors introduced by the method



0.7		1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	1
1 1	0	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	1
0 2	0	0	0	0	0	0	0	1	0	$\bar{1}$	1	0	0	0	0	0	1	0	0	1	0	$\bar{1}$	1	0	$\bar{1}$	0	1	0	0	0
0 4	0	0	0	0	0	0	1	$\bar{1}$	0	0	$\bar{1}$	0	0	0	0	1	$\bar{1}$	0	1	$\bar{1}$	0	0	$\bar{1}$	0	$\bar{1}$	1	$\bar{1}$	0	0	0
0 8	0	0	0	0	0	1	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	0	0	0	1	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0	0
1 16	0	0	0	0	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	0	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0	0
-1 32	0	0	0	0	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	1	0	0	0	$\bar{1}$	0	0	0	0	0	0	0	1	0	$\bar{1}$	0	0	$\bar{1}$	1	$\bar{1}$
-1 64	0	0	0	0	0	0	$\bar{1}$	$\bar{1}$	0	1	0	$\bar{1}$	0	0	$\bar{1}$	0	1	$\bar{1}$	1	$\bar{1}$	1	0	1	$\bar{1}$	0	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
0 128	0	0	0	0	0	0	0	$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$	0	0	1	0	0	0	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$
-1 256	0	0	0	0	0	0	0	0	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	1	$\bar{1}$	0	0	0	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$
-1 512	0	0	0	0	0	0	0	0	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	0	0	0	$\bar{1}$	0	1	$\bar{1}$	$\bar{1}$
0 1024	0	0	0	0	0	0	0	0	0	0	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0	0	$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$
0 2048	0	0	0	0	0	0	0	0	0	0	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$
-1 4096	0	0	0	0	0	0	0	0	0	0	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$
-1 8192	0	0	0	0	0	0	0	0	0	0	0	0	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$
0 16384	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$

Nb. bits 15

exp(0.7) = 2.01375269846830003 ; $\frac{2}{1}\ \frac{17}{16}\ \frac{31}{32}\ \frac{63}{64}\ \frac{255}{256}\ \frac{511}{512}\ \frac{4095}{4096}\ \frac{8191}{8192}\ \frac{32767}{32768}\ \frac{131071}{131072}\ \frac{1048575}{1048576}\ \frac{2097151}{2097152}$ ···· = 2.013752707067453



Nb. bits 30

2.013752707067453 = $\frac{2}{1}\ \frac{17}{16}\ \frac{31}{32}\ \frac{63}{64}\ \frac{255}{256}\ \frac{511}{512}\ \frac{4095}{4096}\ \frac{8191}{8192}\ \frac{32767}{32768}\ \frac{131071}{131072}\ \frac{1048575}{1048576}\ \frac{2097151}{2097152}\ \frac{4194303}{4194304}$ ···· ; 2.0137527051919992 ;

The series of rational products is given by the concatenation of the quotient Q (left) and the final remainder R (bottom). Actually for high values of i, $2^i * \log(1 + 2^{-i})$ becomes very close to 1. If the divider is close 1, then the remainder becomes an acceptable approximation for the quotient.

Range extension

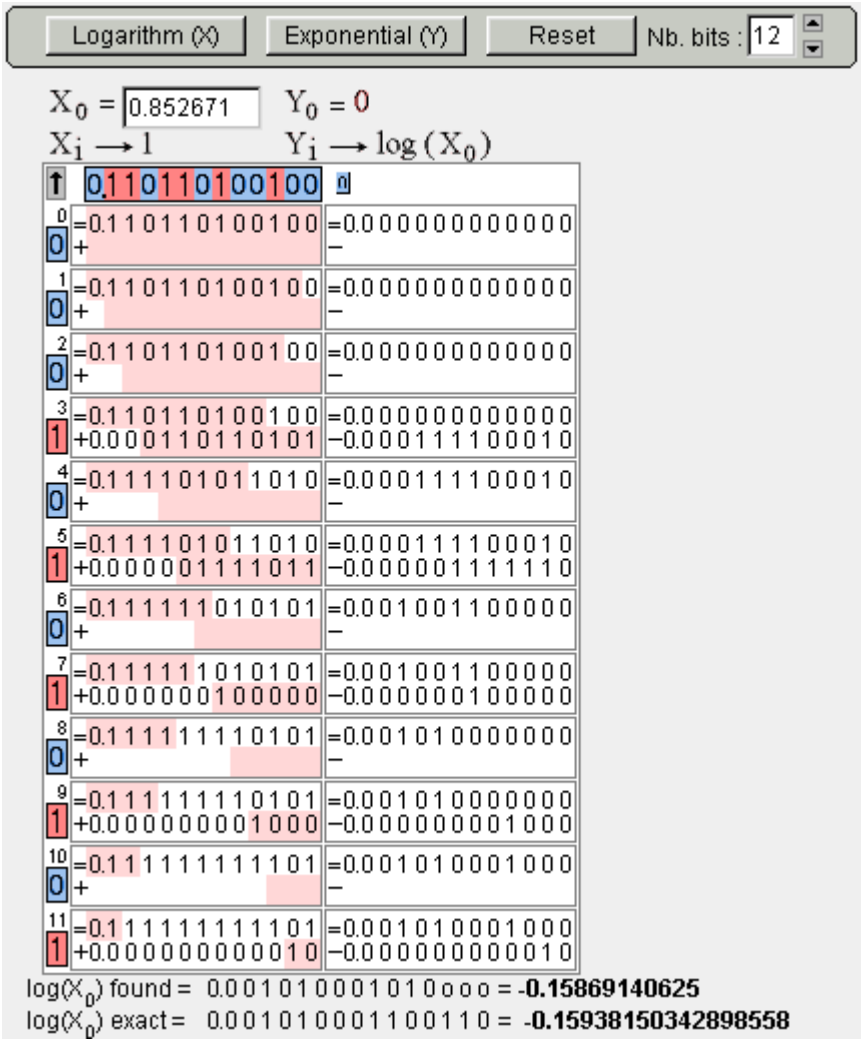
The previous circuit works with Y in the range] -1 , +1 [. For the exponential of any number Y, Y is written $Y = Q*\log(8) + R$, where Q is the integer quotient of the division of Y by log(8) and $R < \log(8) < 1$. Then $\exp(Y) = 8^Q * \exp(R) = 2^{3Q} * \exp(R)$. Since $\exp(R) < 1$, it is acceptable by the above circuit.

Logarithm and exponential

$$X = \prod_{j=1}^{n} (1 + 2^{-j})^{p_j} \iff \log(X) = \sum_{j=1}^{n} p_j \log(1 + 2^{-j})$$

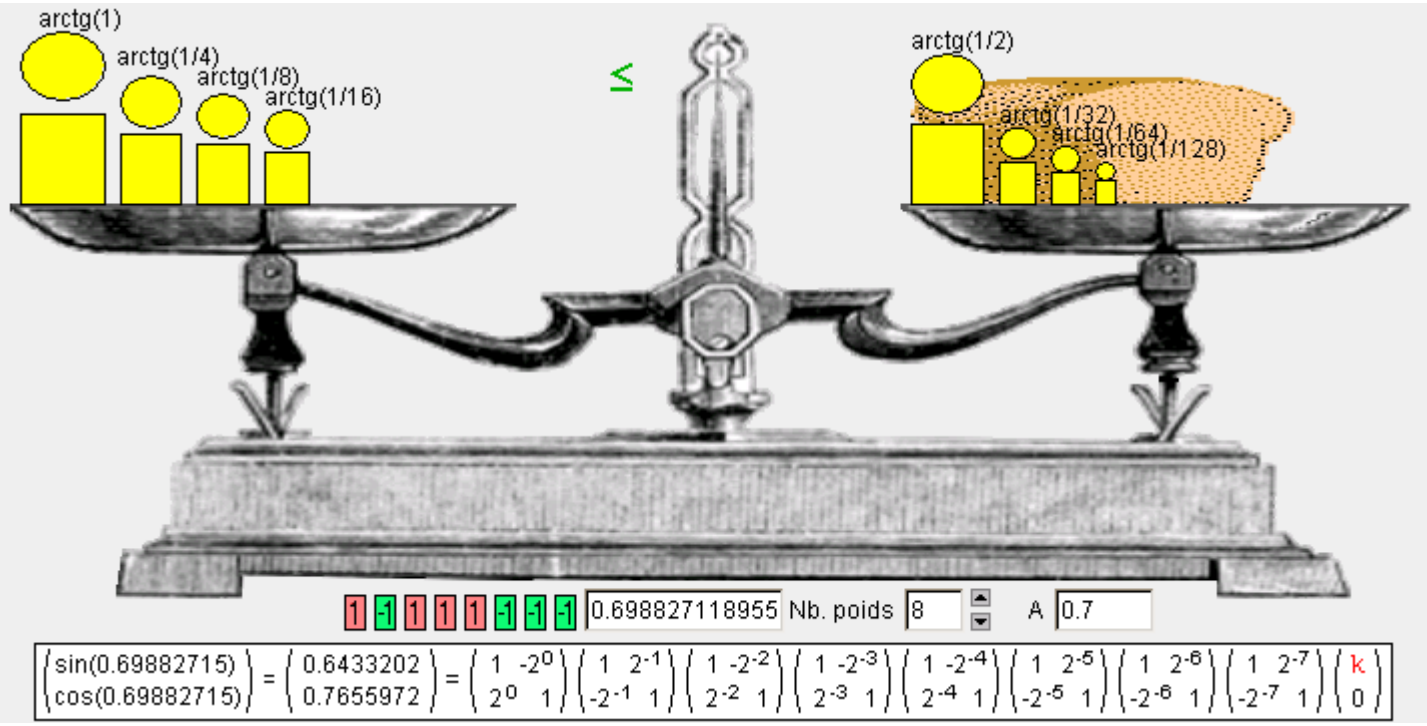
$$\begin{tikzcd}
\sum_{i=0}^{m-1} x_i 2^{-i} \arrow[d, shift right, "1"'] & & \sum_{i=0}^{m-1} y_i 2^{-i} \arrow[d, shift right, "3"'] \\
\prod_{j=1}^{n} (1 + 2^{-j})^{p_j} \arrow[u, shift right, "2"'] \arrow[rr, shift left, "\text{exponentielle}"] \arrow[rr, shift right, "\text{logarithme}"'] \arrow[rr, phantom, "5"] & & \sum_{j=1}^{n} p_j \log(1 + 2^{-j}) \arrow[u, shift right, "4"']
\end{tikzcd}$$

The same operator computes either the Logarithm or the Exponential with additions/subtractions (it is the same operation), shifts and constants. The constants are $\log(1 + 2^{-i})$ and $-\log(1 - 2^{-i})$ and the digits $\in$ { '-1' , '0' , '1' }. The slack selection of the digit value, which unfortunately will be lacking later on for Sine and Cosine, allows to avoid all but one carry propagation



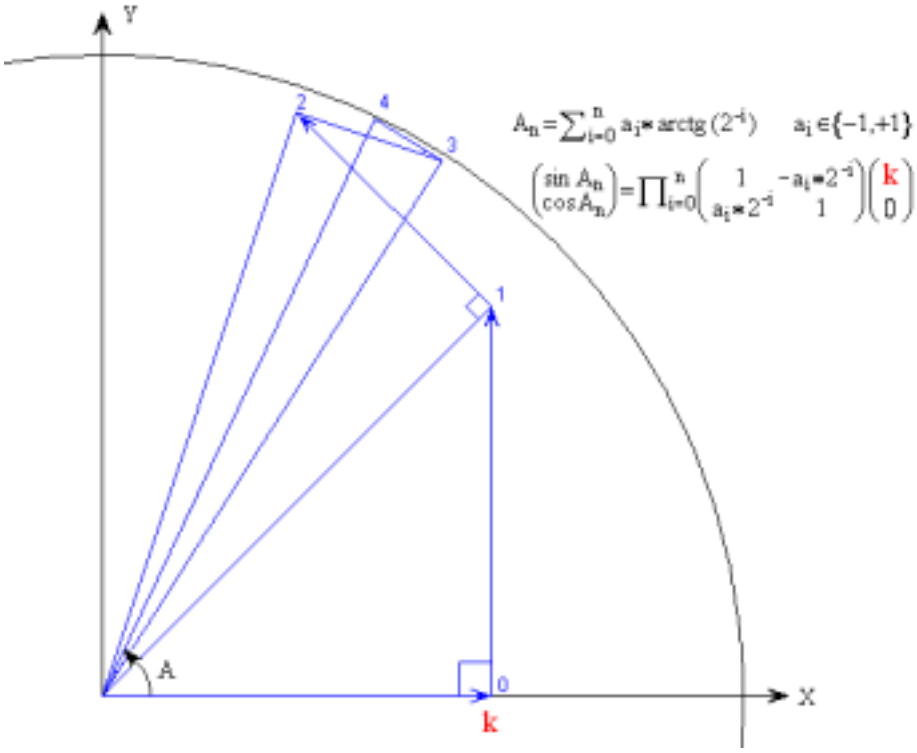
From a bread loaf weighting to sine and cosine

We want to compute sine(A) and/or cosine (A); we have available a scale, a white bread whose weight is actually just A and a set of weights with values arctg (2^{-i}). All the weights must go on the scale plates, either side.



Sine and Cosine computation

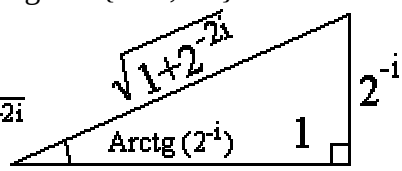
Let V_i be a vector, with extremity (x_i, y_i). A "pseudoRotation" of an angle arctg (2^{-i}) applied to V_i gives V_{i+1} : $x_{i+1} = x_i - y_i * 2^{-i}$ and $y_{i+1} = y_i + x_i * 2^{-i}$. After the angle A is broken down into a weighted sum of arctg (2^{-i}), a series of "pseudoRotations" yields the coordinates of the vector of angle A, those coordinate are the values sin(A) and cos(A) searched for. All the "pseudoRotations" require only addition/subtraction and shift.



The constant k

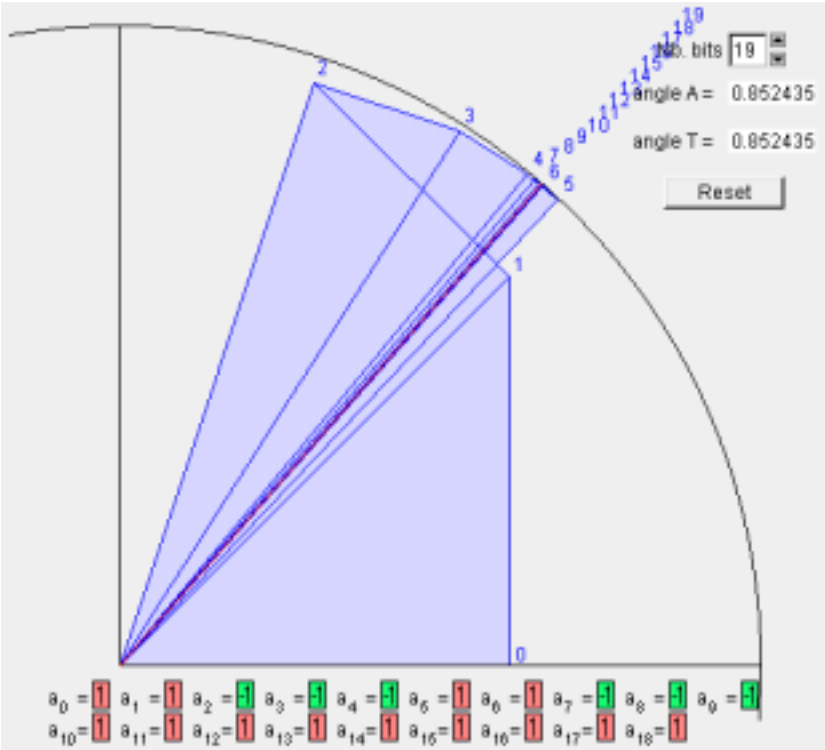
Each "pseudoRotation" of the angle arctg(2^{-i}) brings about a vector lengthening of $\sqrt{1+2^{-2i}}$, for it is not exactly a rotation but rather a displacement of the vector extremity on a perpendicular vector. In order to compensate in advance the product of all the lengthening of a series of "pseudoRotations", the starting vector is ($x_0 = k$, $y_0 = 0$) . For n large enough, k is approximately equal to 0,60725. In order for k to be a constant, the representation with arctg(2^{-i}) use digits ∈ { '-1' , '1' }.

$$k = \frac{1}{\prod_{i=0}^{n} \sqrt{1+2^{-2i}}}$$



Angle decomposition

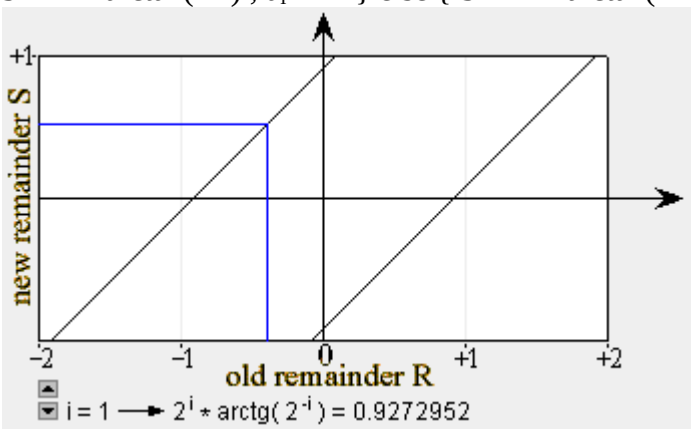
What is the domain of the angles A = $\sum_{i=0}^{n} a_i * \mathrm{arctg}\ (2^{-i})$ and what precision can be expected from this notation ? A is the angle value to reach, and T is the value attained by the series of pseudoRotations. To change the value of A, click in the figure. All values are expressed in radiant. The key "Mise à zéro" allow to 'manually' control the convergence into the notation arctg (2^{-i}) by clicking the digit values.



"Robertson's diagram" for CORDIC

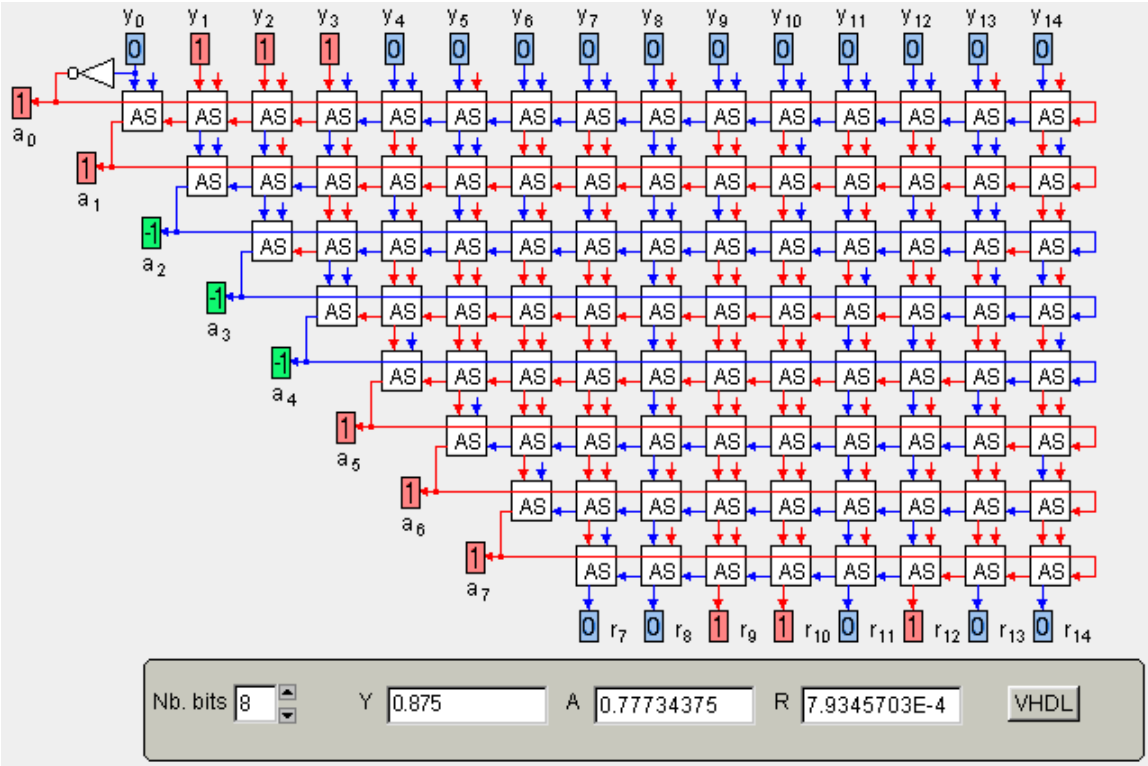
The "Robertson's diagram" shows that the iteration to convert into basis $\arctan(2^{-i})$ may be as follows:

if $R \geq 0$ **then** { $S = R - \arctan(2^{-i})$; a_i = '1' } **else** { $S = R + \arctan(2^{-i})$; a_i = '-1' }.



"non restoring" divider for CORDIC

The angle Y (divider's top) is in the interval [-1.743.. +1.743...]. The constant bits at "AS" cells inputs are wired. The operations are selected according to the previous partial remainder R or by y_0 for the first iteration ..

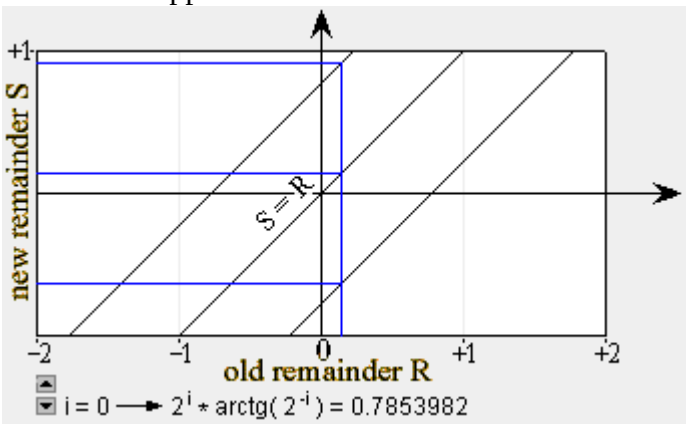


"double rotation" CORDIC " Robertson's Diagramme"

It uses an approximation $\hat{R}$ of the partial remainder R to determine the rotation:

if $\hat{R} > 0$ **then** { a_i = '1' }; **if** $\hat{R} = 0$ **then** { a_i = '0' }; **if** $\hat{R} < 0$ **then** { a_i = '-1' }

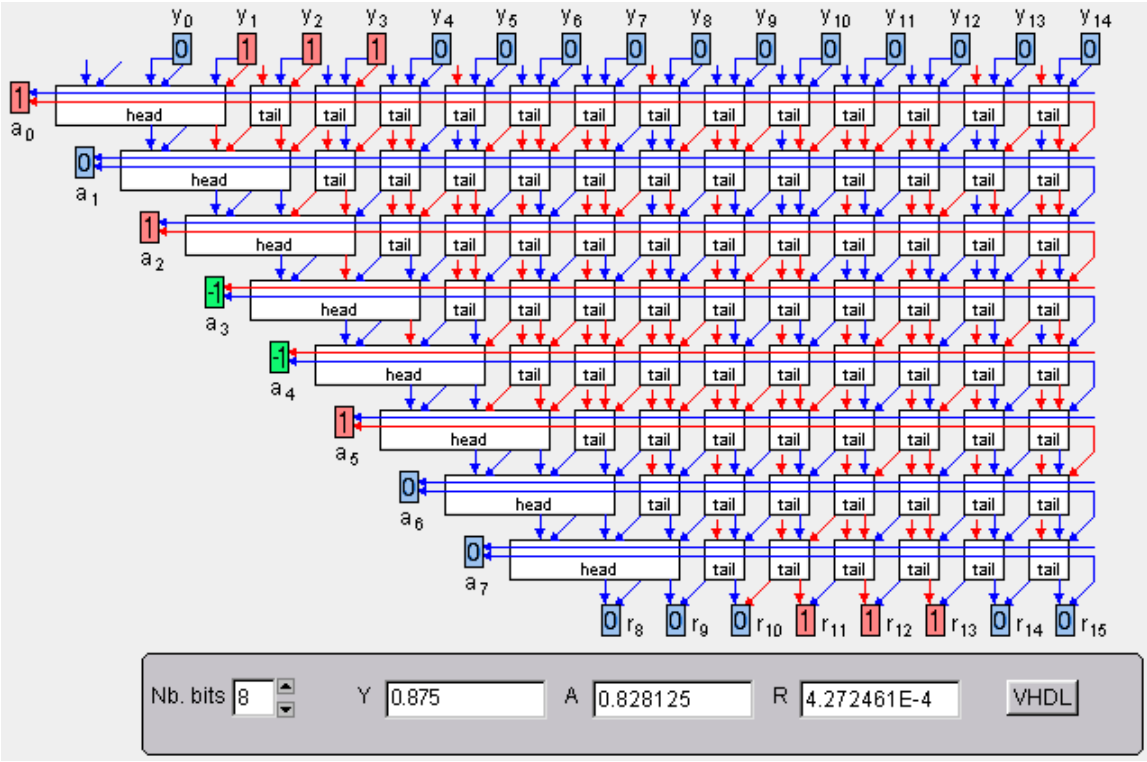
The diagram shows that the approximation can be coarse.



"double rotation" CORDIC

This divider uses the same "head" and "tail" cells as the "SRT" divider. To put up with the '0', the angle is first halved then the "pseudoRotation" are doubled. Thanks to that, the lengthening stays the same ($2*\sqrt{1+4^{-i}}$) whatever the value of a_i .

- **if** a_i= '-1' **then** { rotation of (arctan(2^{-i})) then rotation of (arctan(2^{-i})) };
- **if** a_i = '0' **then** { rotation of (arctan(2^{-i})) then rotation of (-arctan(2^{-i})) } ;
- **if** a_i = '1' **then** { rotation of (-arctan(2^{-i})) then rotation of (-arctan(2^{-i})) } ;

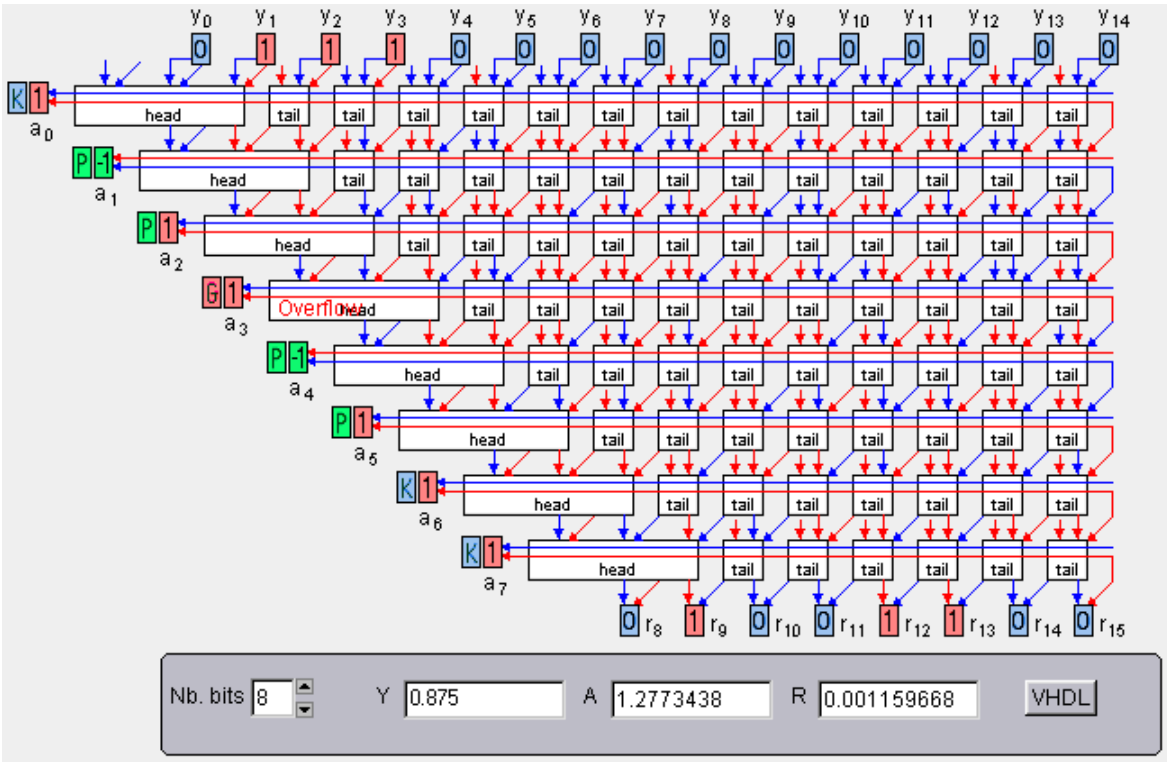


"double division" CORDIC

The "double rotation" has a cost: it doubles the rotation hardware and probably the delay as well. The "double division" is more ingenious. It makes use of two dividers running simultaneously with slightly different "head" cells..

"head" of divider1	"head" of divider2
if $\hat{R} > 0$ **then** { $\hat{S} = \hat{R} - 2$; a_i = '1' ; }	**if** $\hat{R} \geq 0$ **then** { $\hat{S} = \hat{R} - 2$; a_i = '1' ; }
if $\hat{R} \leq 0$ **then** { $\hat{S} = \hat{R} + 1$; a_i = '-1' ; }	**if** $\hat{R} < 0$ **then** { $\hat{S} = \hat{R} + 1$; a_i = '-1' ; }

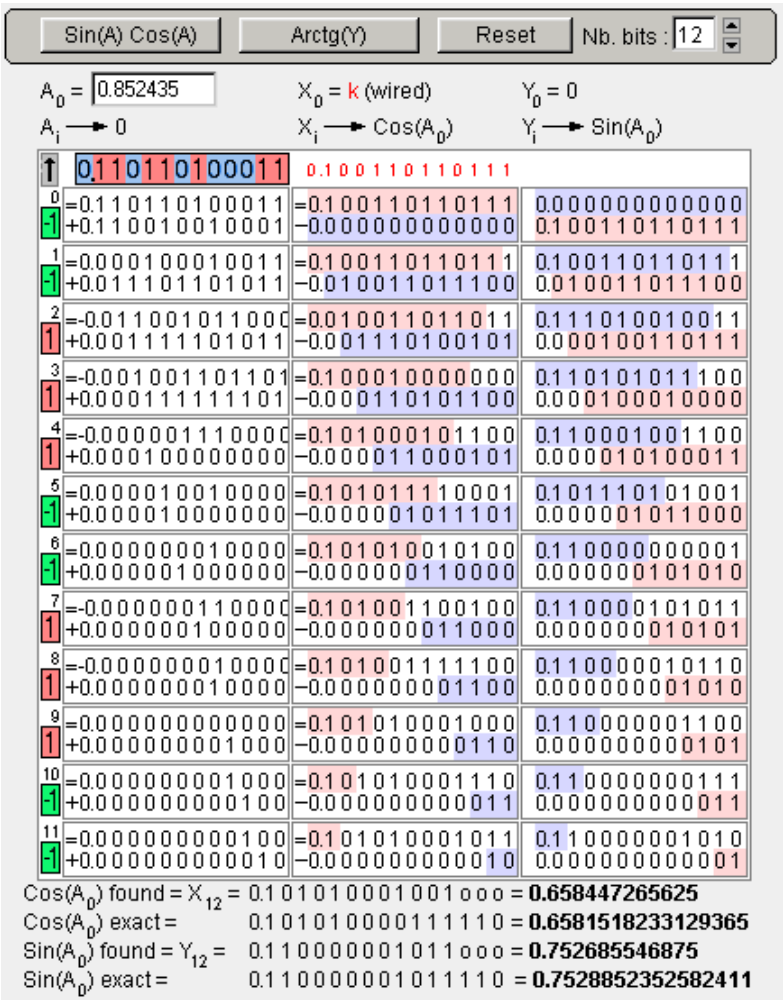
It is clear that whenever $\hat{R} = 0$, divider1 speculates that R < 0 and divider2 that R > 0. At most one of them will eventually overflow, before the occurrence of the next $\hat{R} = 0$. An overflow indicates that the correct output is the other divider's one.
Only divider1 is shown by the following applet.



The two heads detect the overflow to produce together a 3-valued indicator :
'K' the output digit is correct (either divider1 and divider2 give the same value, or divider2 overflows), 'G' the output digit is incorrect and must be complemented (divider1 overflows), 'P' propagate the next indicator's value (values differ, no overflow). Whenever a divider overflows, it carries on with the other divider's partial remainder R.
The propagation is similar to the carry propagation of addition..

Numerical application

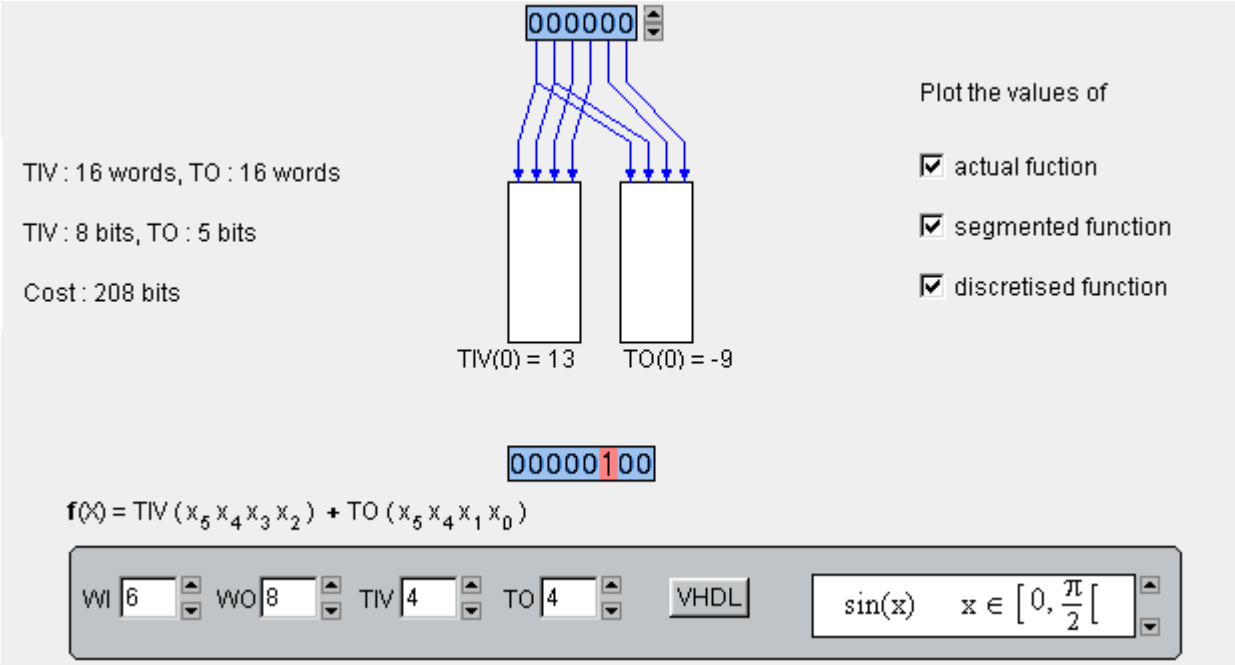
The "Nb. bits" selects simultaneously the number of bits of the calculations and the number of steps. Clicking the vertical arrow ↑ changes the representation. Again, the key "Mise à zéro" allows to control 'manually' the convergence.

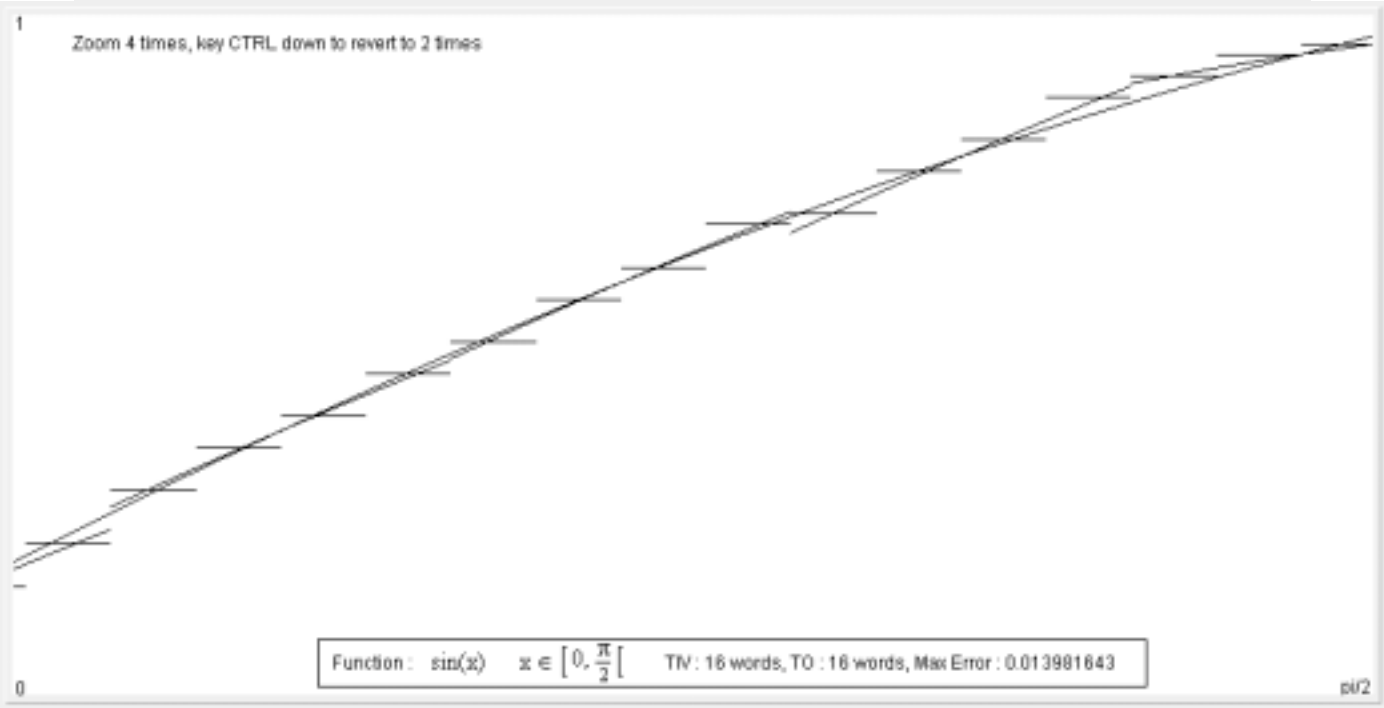


Bipartite Table

$\sin(x)$	$x \in [0, \frac{\pi}{2}[$
$\sin(x)$	$x \in [0, \frac{\pi}{4}[$
$2^x - 1$	$x \in [0, 1[$
$\log_2(x)$	$x \in [1, 2[$
$\frac{1}{x} - \frac{1}{2}$	$x \in [1, 2[$
$\sqrt{x} - 1$	$x \in [1, 4[$
$\frac{1}{\sqrt{x}} - \frac{1}{2}$	$x \in [1, 4[$
$\sqrt[3]{x} - 1$	$x \in [1, 8[$

The values of a function can be precomputed and stored into a table (a ROM). Nevertheless, the table size grows very quickly with the precision. This practically limits this approach. For continuous functions, one may store only a few values in a table named "TIV", and the function slope, in order to interpolate within the stored points, in another table named "TO".
In the applet below, start by selecting a function, then fix WI and then explore solutions varying the values of TIV and TO around 2/3 of WI.
Adding a function to the list implies the modification of the source.





Modular Arithmetic

Modular representation

Let be the set { m_1, m_2, m_3, ... m_n} of n integer constant pairwise prime called moduli and let M be the product of this constants, $M = m_1 * m_2 * m_3 * \ldots * m_n$.

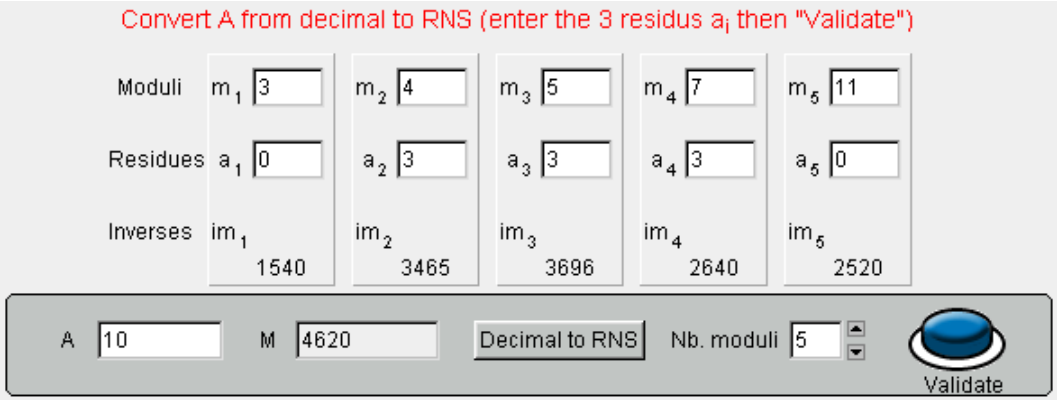
Let A be an integer smaller than M.

A can be written $(a_1 | a_2 | a_3 | \ldots | a_n)_{RNS}$ where a_i = A modulo m_i (residue).

This definition tells how to get the a_i from A. On the other hand it is possible to get back A from the a_i using another set of precomputed constants { im_1, im_2, im_3, ... im_n} called inverse modulo M of the former.

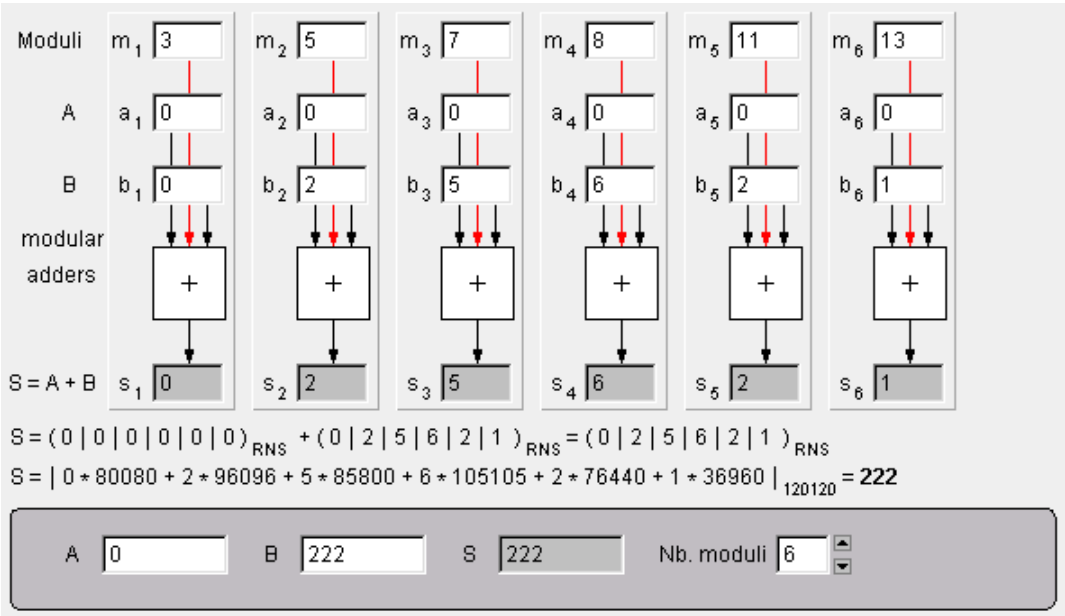
$A = | a_1 * im_1 + a_2 * im_2 + a_3 * im_3 + \ldots\ldots a_n * im_n |_{\text{modulo } M}$

This result is proved in the "Chinese remainder theorem". Check if you are acquainted with this representation by converting A from "decimal to RNS" or from "RNS to decimal".



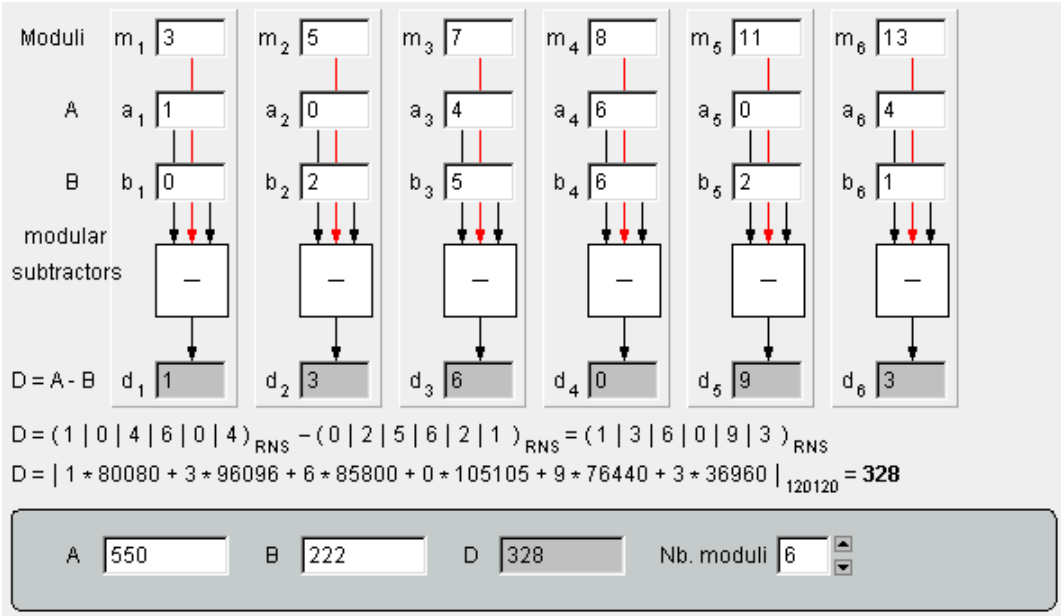
Modular addition

Modular addition uses n <u>small</u> adders computing <u>simultaneously</u> all the sums $s_i = | a_i + b_i |_{\text{modulo}} m_i$.



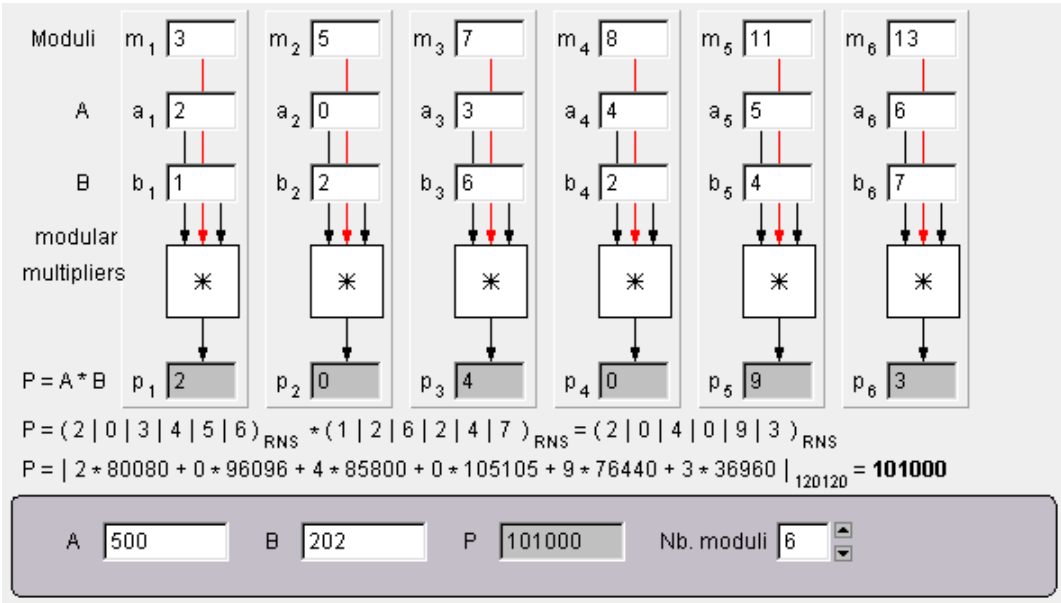
Modular Subtraction

Modular subtraction uses n <u>small</u> subtractors computing <u>simultaneously</u> all the differences $d_i = | a_i + m_i - b_i |_{\text{modulo}} m_i$.



Modular Multiplication

Modular multiplication uses n <u>small</u> multipliers computing <u>simultaneously</u> all the products $p_i = | a_i * b_i |_{modulo\ m_i}$.



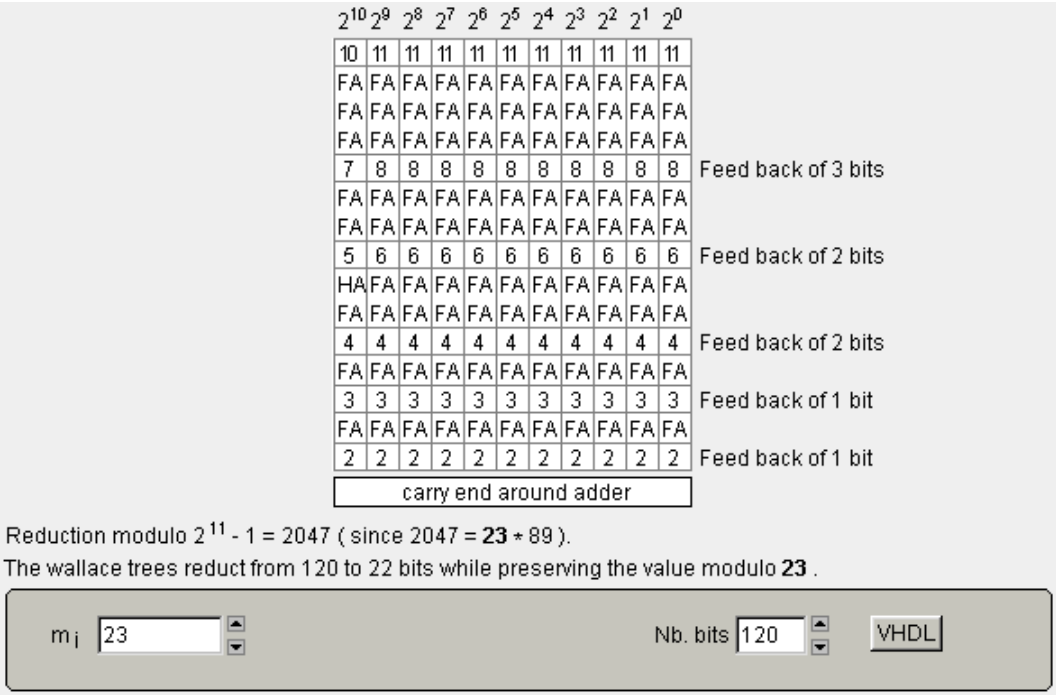
Conversion into RNS

The conversion of a binary variable A into RNS consists in finding all a_i = A modulo m_i i.e. the remainders of the division of A by m_i. But the division is not the best approach.

- the rest modulo 2^n is immediate,
- the rest modulo $2^n - 1$ requires only additions,
- the rest modulo $2^n + 1$ requires some additions and some subtractions.

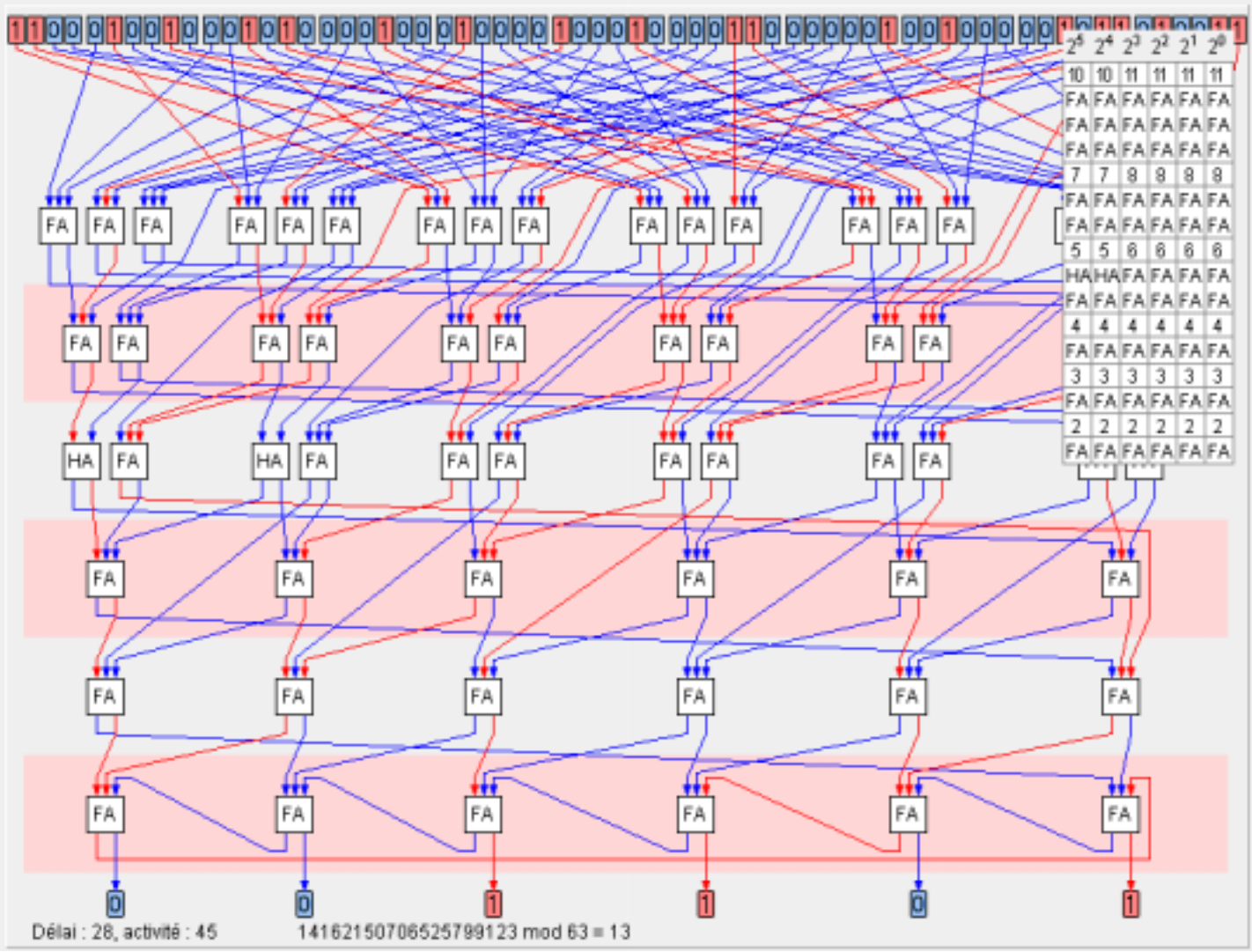
In other cases we resort to the one of the two last expressions with the smallest n. Trees of adders (Wallace trees) reduce A to the sum of two n-bit numbers while respecting the rest modulo m_i.

The graphical conventions are the same as for partial products reduction .



Example of modulo reduction

The following applet reduces 64 bits into 6 bits whereas preserving the value modulo 63 ($63 = 2^6 - 1$) . At the output, zero has two notations: either "000 000" or "111 111"



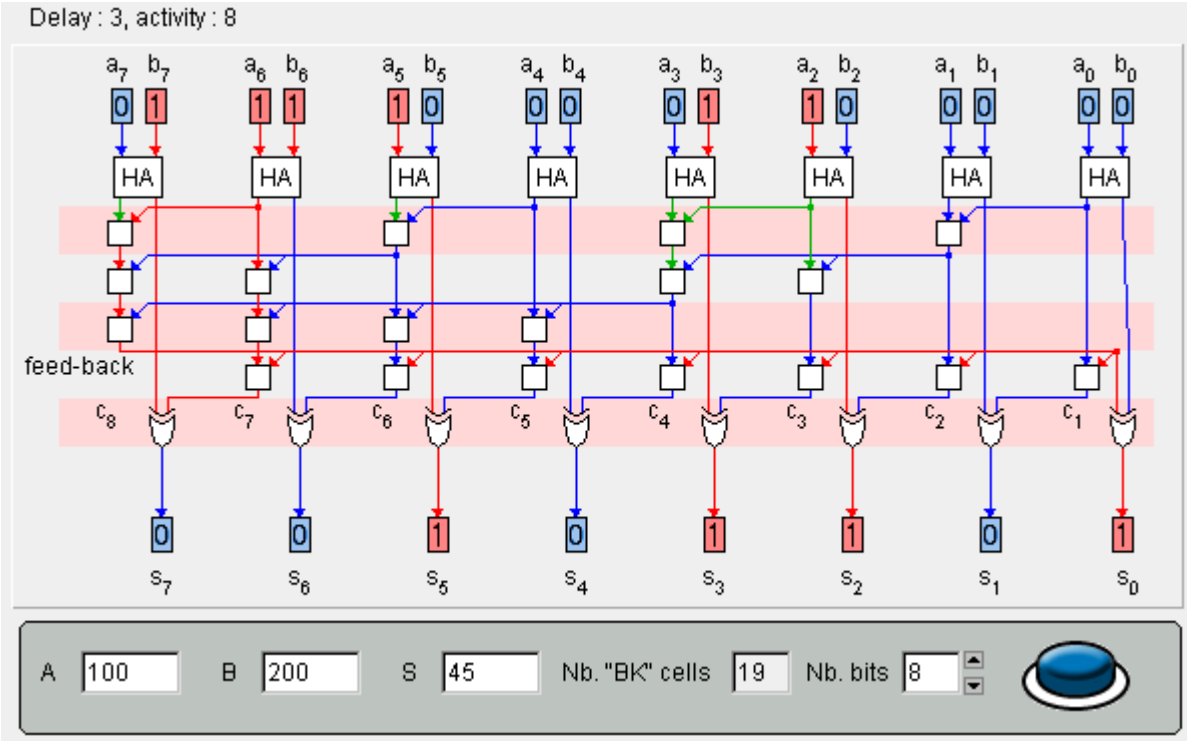
modulo 2^n–1 adder

The "end-around-carry" adder offers two advantages : it works fine modulo $2^n - 1$ and is simple, and two disadvantages as well : it is slow and difficult to test, both for the same raison i.e. for the value zero are two stable cases.

An adder delivers spontaneously a modulo 2^n sum. With a slight modification, the Sklanski's adder delivers a modulo $2^n - 1$ sum S.

- **if** $A + B < 2^n - 1$ **then** $S = A + B$;
- **if** $A + B \geq 2^n - 1$ **then** $S = | A + B + 1 |_{\text{modulo } 2^n}$

The condition is given by the carry out c_n: if c_n = 'K' then $A + B < 2^n - 1$, if c_n = 'P' then $A + B = 2^n - 1$, if c_n = 'G' then $A + B > 2^n - 1$. The "feed-back" signal that controls the "+1" is 'K' if c_n = 'K' and 'G' otherwise.



modulo 2^n +1 adder

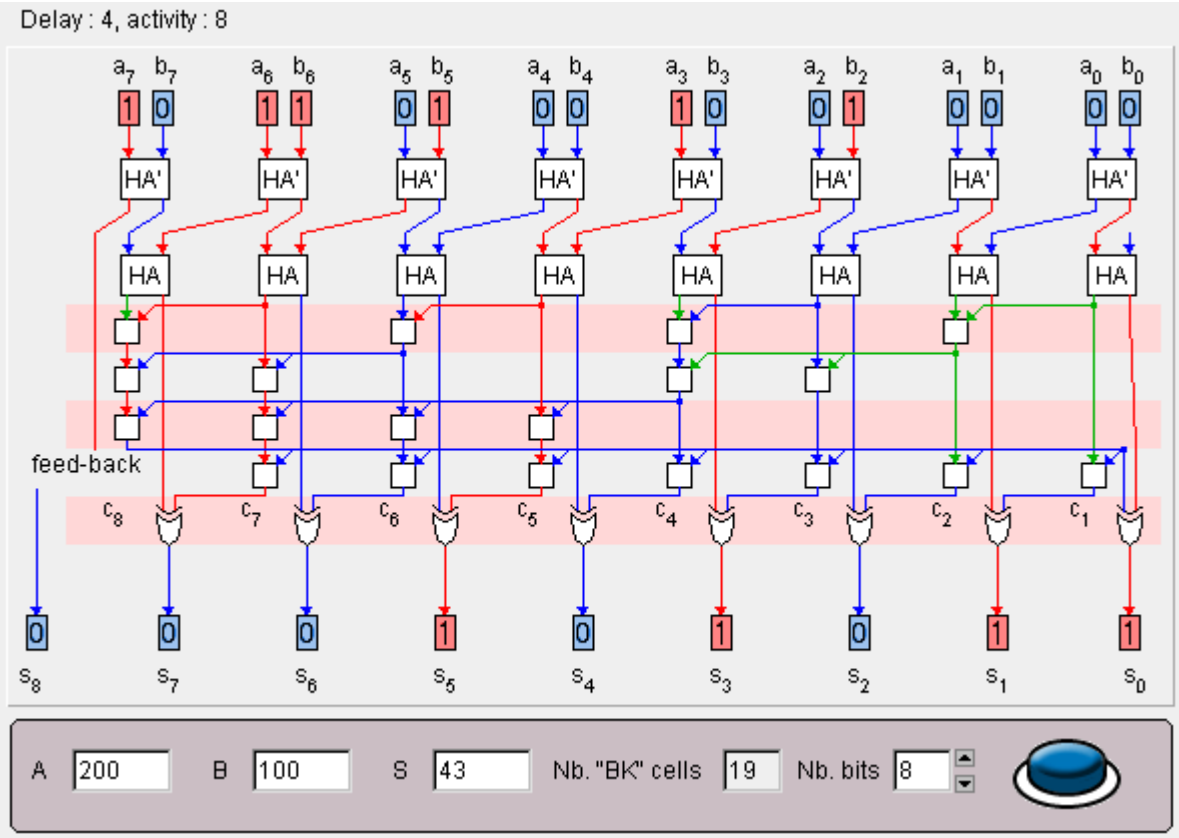
We now want an adder modulo $2^n + 1$.

- **if** $A + B < 2^n + 1$ **then** $S = A + B$;
- **if** $A + B \geq 2^n + 1$ **then** $S = | A + B - 1 |_{\text{modulo } 2^n}$

The previous adder is used with two numbers X and Y such that $X + Y = A + B + 2^n - 1$. A row of HA' cells carries on this addition propagation-free. HA' is the dual of HA.

- **if** $X + Y < 2^{n+1}$ **then** $S = | X + Y + 1 |_{\text{modulo } 2^n}$;
- **if** $X + Y \geq 2^{n+1}$ **then** $S = | X + Y |_{\text{modulo } 2^n}$;

The "feed-back" signal that controls the "+1" is the "nand" of x_n and (c_n = 'K'). The result bit s_n is the "and" of x_n and (c_n = 'P').



Conversion from "RNS" into mixed-radix system "MRS"

The "Mixed Radix System" is a positional number system with weights (1) (m_1) (m_1m_2) ($m_1m_2m_3$) ($m_1m_2m_3.....m_{n-1}$) .
In this system X is written ($z_1 | z_2 | z_3 | .... | z_n)_{MRS}$ with $0 \le z_i < m_i$. Note that the digit set have the same range as the RNS digits, but the digits themselves are different.
The value of $X = z_1 + m_1 * (z_2 + m_2 * (z_3 + m_3 * (.....)))$.