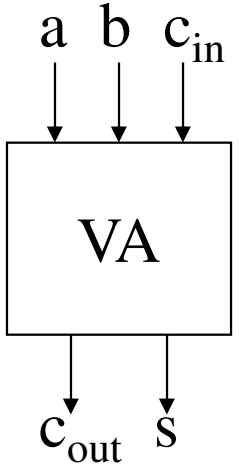
Computer Arithmetic

In this section, we want to know some fundamental techniques of how computers process arithmetic operations. What we will learn here will be needed for the later chapters about sequential circuits, ALU structure and computer hierarchy.

Addition

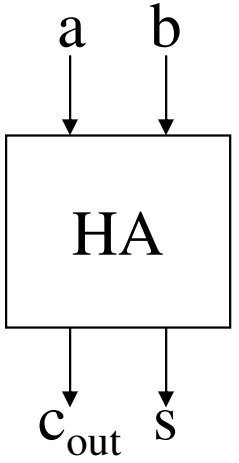
We have learnt about the full adder already. It is a combinatorial circuit with three inputs, a, b, c_{in} and two outputs s and c_{out}. The full adder is capable to add up three bits and give a result as a 2-bit number. The result is between zero to three and hence it can be coded in 2-bits. Here we see the circuit diagram of a full adder and its truth table:



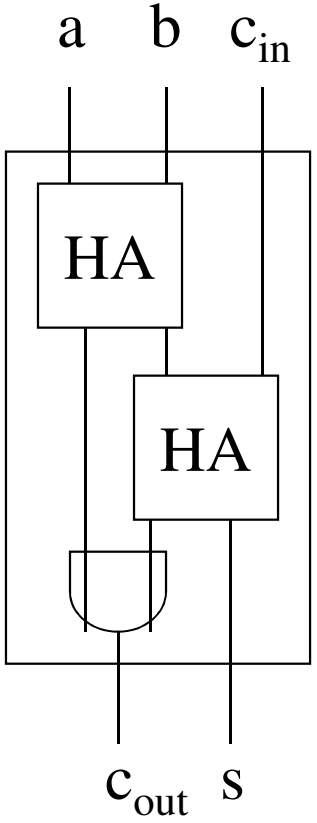
a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Very often full adders are not realized in DMF but in the form of several steps or computations whereby several so-called “half adders” are used. Half adders are combinatorial circuits, capable of adding 2 bits (and hence produce a result between 0 to 2). By connecting 2 half adders and an OR gate, one can produce the functionality of a full adder. Next, we will see the circuit diagrams of a half adder and its truth table and the structure of a full adder consisting of half adders.

a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



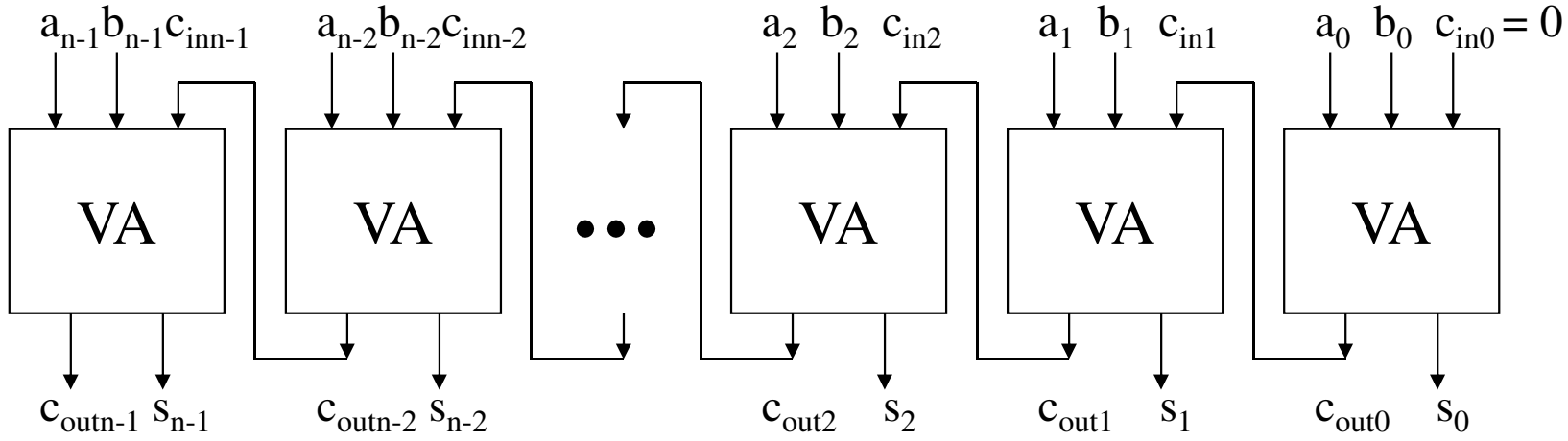
Full adder using two half adders and an OR gate



Usually, we will want to make additions of longer operands (larger numbers). E.g. The binary numbers a= $a_{n-1}a_{n-2}...a_1a_0$ and b= $b_{n-1}b_{n-2}...b_1b_0$. Of course, one could build the required adder in DNF or DMF. This leads to a row of problems:

- Each n leads to a complete different realization
- The fan-in and fan-out at the gates increase in a polynomial rate with n

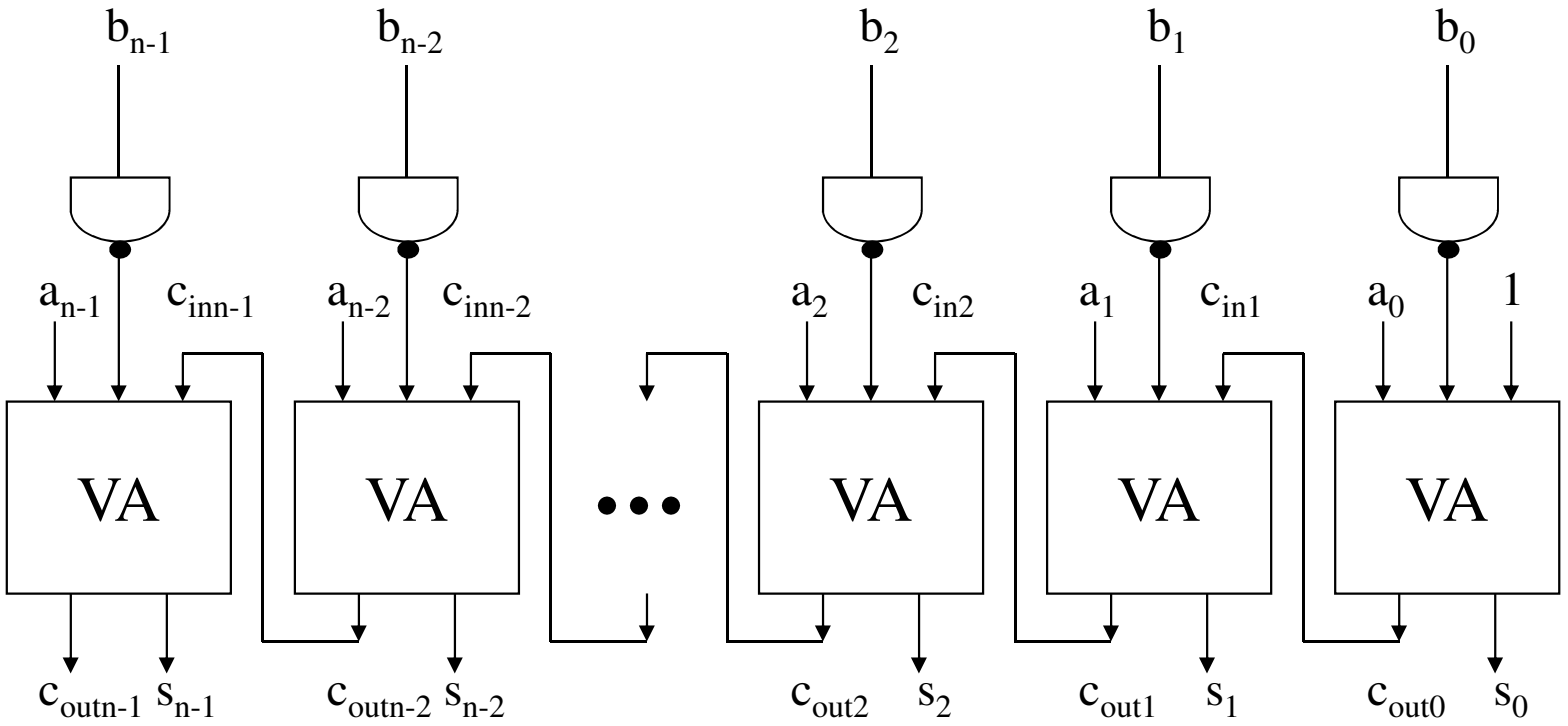
Due to these characteristics, the two-levelled structure for example in DMF is not senseful. Instead, one uses a chain of full adders in the simplest case which basically execute exactly as to what we have learnt about addition in school. One starts with the LSBs (least significant bits) and add these up, produce an overflow which is needed to incorporate with the next bit and so on. A corresponding combinatorial circuit is as follows:



The result of the addition from two n-bit numbers is a n+1-bit number. This is represented through the outputs $c_{n-1}s_{n-1}s_{n-2....}s_2s_1s_0$.

Such an adder is called a **ripple-carry-adder**. Its advantage is its simple and modular construction. Its main disadvantage, is as stated by its name: If the operand has an awkward bit combination, the carry information (overflows) has to go through all full adders, beginning from the level of the least significance up to the level of the highest significance (rippling). That leads to a switching time of a ripple carry adder which is proportional to the number, n bits. This is especially a problem if our computer has to process a number format with many bits (e.g. 64 bits). Of course, we don't want to make the machine clock so slow that 64 full adders can switch one after another in 1 clock cycle. We will soon seen how one can handle this problem.

Firstly, we want to learn how an adder can be used for subtraction. We know already: The 2's complement of a number can be calculated as 1's complement plus 1. Furthermore, the 1's complement is a bitwise negation of a number. If we now perform the addition of 1 using the carry input c_{in0}, we can use the combinatorial circuit of the following sheet to calculate a-b by adding the 2's complement of b to a.

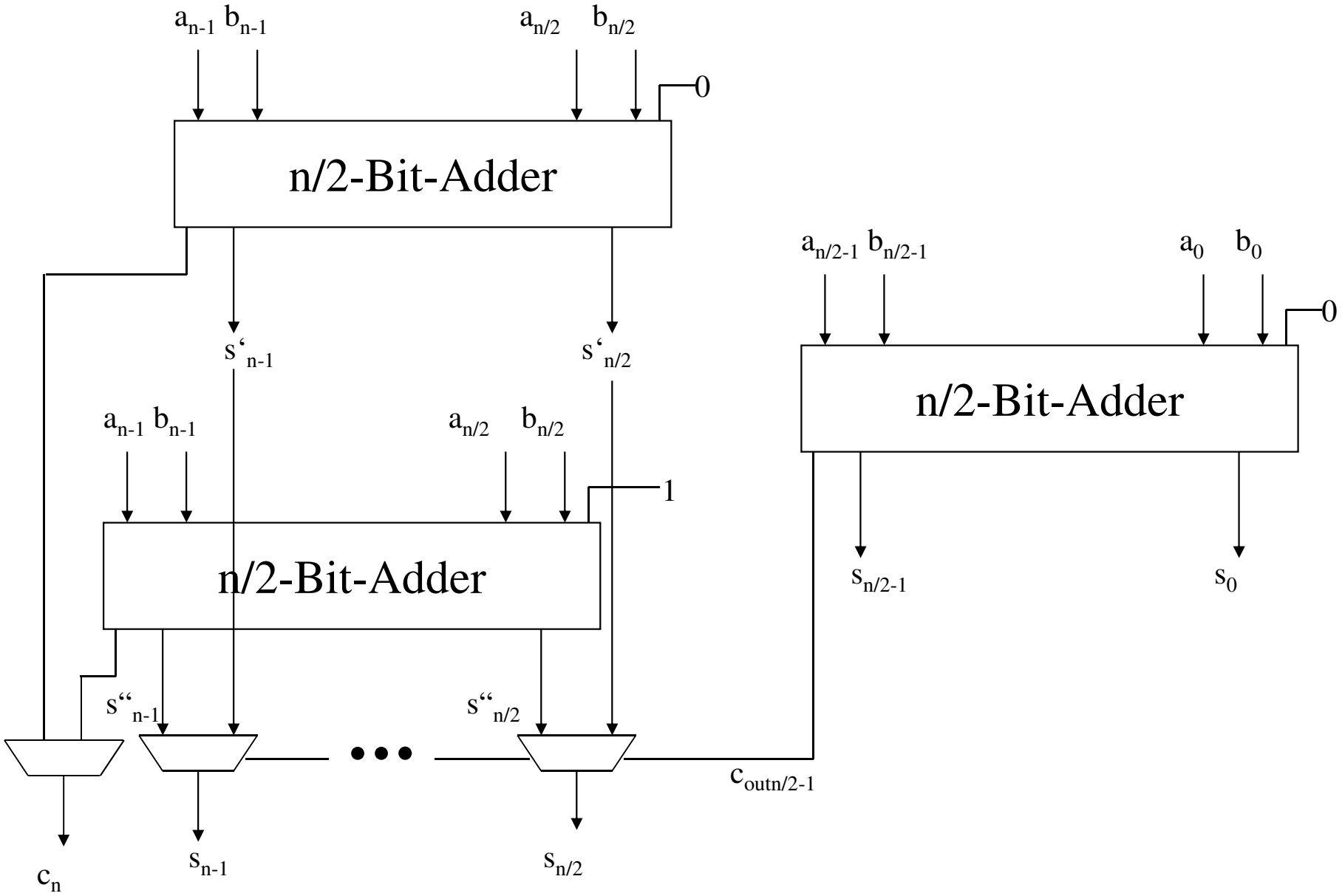

b_{n-1}
b_{n-2}
b_2
b_1
b_0
a_{n-1}
c_{inn-1}
a_{n-2}
c_{inn-2}
a_2
c_{in2}
a_1
c_{in1}
a_0
1
VA
VA
VA
VA
VA
c_{outn-1}
s_{n-1}
c_{outn-2}
s_{n-2}
c_{out2}
s_2
c_{out1}
s_1
c_{out0}
s_0

Faster Addition and Subtraction

What defines the switching time for an addition or subtraction? By the number of full adders where a carry has to propagate one after another. E.g. If we add +1 to -1, the inputs of the adders are a = 00000001 and b = 11111111. The addition causes an overflow at the last digit and this causes an overflow at the digit before the last digit and so on, up to the first digit where we will have the final overflow. Hence, the processing time depends on the number of digits the overflow has to propagate. If every full adder needs the time t_{VA}, the total time needed will be n * t_{VA}. How to reduce this time? A nice solution that is also used often in the computer architecture is the **carry select adder**.

The idea is as follows: The adder will be divided into equally long halves. Both halves start at the same time with the addition. For the left half of the adder (higher significant) we don't know if the carry input of the right full adder is a 1 or a 0. That is why we calculate the addition of the left half of the adder twice. Once, with a 0 as carry input and once with a 1. Once the right half of the adder is finished with the addition, we know the actual carry input of the left half. Hence, we know which result is correct. This is what we select then. The other (wrong) result will be discarded.

The result selection is done using an amount of multiplexers controlled by the incoming carry. The principle can be seen from the following sheets.


a_{n-1} b_{n-1}
$a_{n/2}$ $b_{n/2}$
0
n/2-Bit-Adder
s'_{n-1}
$s'_{n/2}$
a_{n-1} b_{n-1}
$a_{n/2}$ $b_{n/2}$
1
n/2-Bit-Adder
s''_{n-1}
$s''_{n/2}$
c_n
s_{n-1}
$s_{n/2}$
$a_{n/2-1}$ $b_{n/2-1}$
a_0 b_0
0
n/2-Bit-Adder
$s_{n/2-1}$
s_0
$c_{outn/2-1}$

We can see immediately: The amount of gates used is slightly more than 1.5 times the amount of gates we need for the ripple carry adder. Now what is the switching time for such an adder? Because all n-2-Bit-Adder are working at the same time, we only need half of the overall time, namely $n/2 * t_{VA}$ for the addition. Additionally, we need the small time constant for the multiplexers. Hence the total time needed is $n/2 * t_{VA} + t_{MUX}$

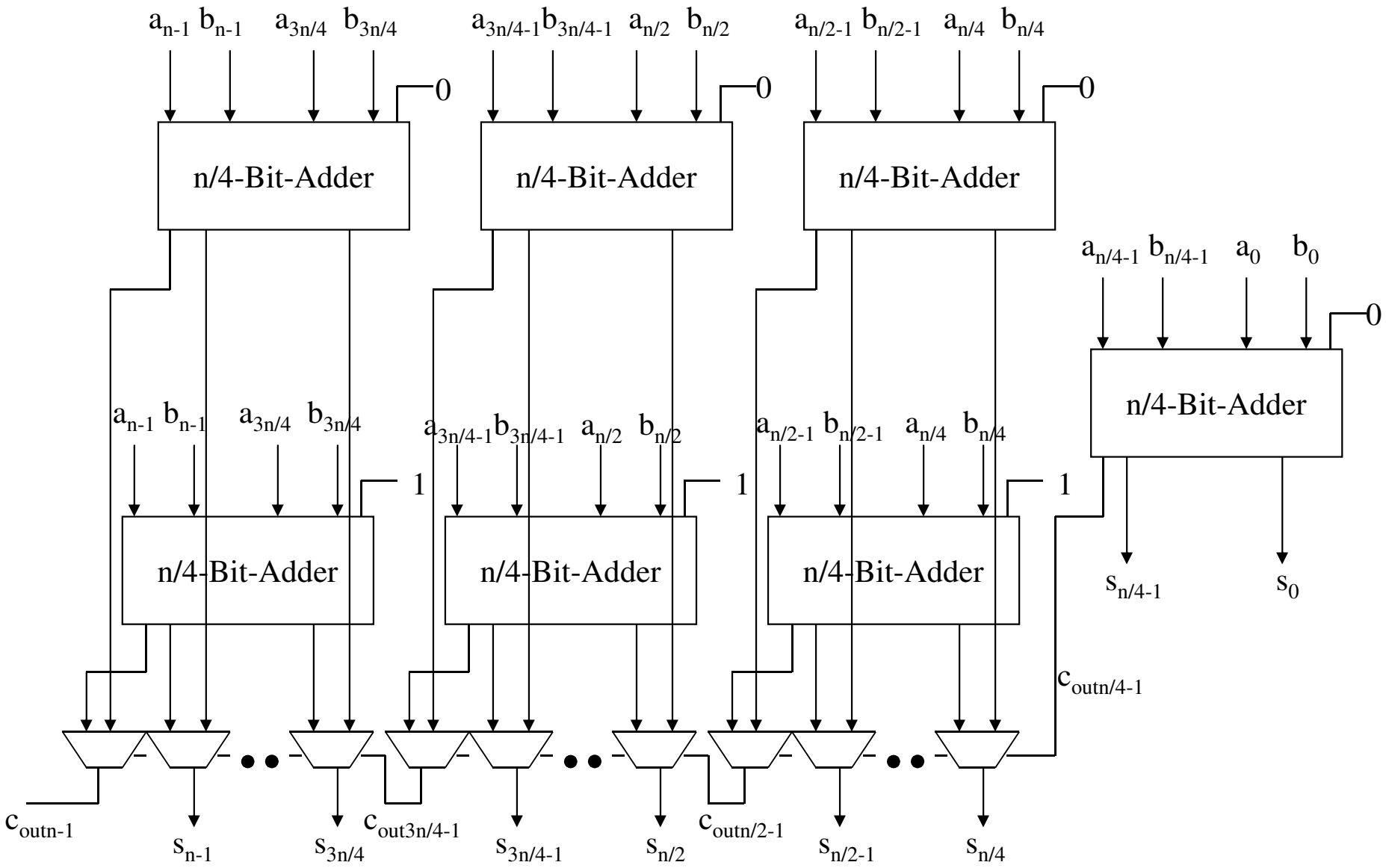
That's why we have improved on the time with a factor of 2. Now this principle can of course, be applied repetitively: Instead of adders of the length n/2, we can also take a length of n/4, n/8 and so on. All such adders (except of the least significant one) have to be implemented twice, where one always uses the carry input 0 and the other carry input 1. Which result will finally be taken is decided by the carry of every previous step.

The following sheets shows the results of this technique for a division in four parts. The run-time is reduced to $n/4 * t_{VA} + 3t_{MUX}$. Generally applicable for a division in m parts:

$$t_{Total} = n/m * t_{VA} + (m-1) * t_{MUX}$$

If we want to calculate the total minimum time, we simplify as follow $t_{VA}=t_{MUX}$. Then, we have to differentiate t_{Total} with respect to m and set the derivative to 0.

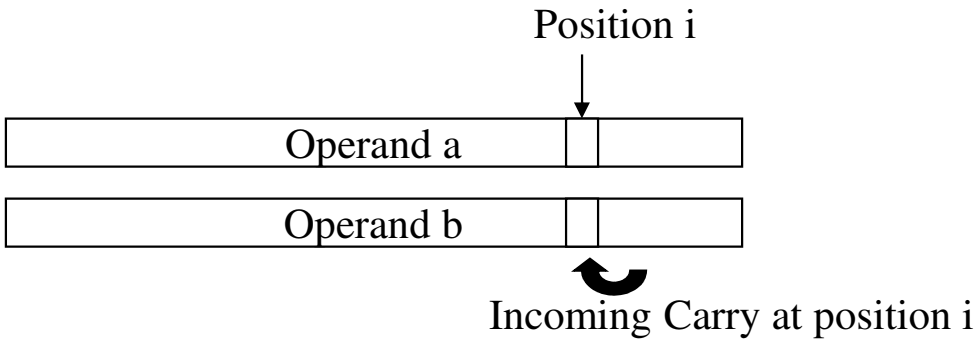
This total time has a minimum at $m = n^{1/2}$. Hence, the addition that uses a carry select adder works in $O(n^{1/2})$, whereas the ripple carry adder works in O(n).


a_{n-1} b_{n-1} $a_{3n/4}$ $b_{3n/4}$
$a_{3n/4-1}$ $b_{3n/4-1}$ $a_{n/2}$ $b_{n/2}$
$a_{n/2-1}$ $b_{n/2-1}$ $a_{n/4}$ $b_{n/4}$
$a_{n/4-1}$ $b_{n/4-1}$ a_0 b_0
0
1
n/4-Bit-Adder
$s_{n/4-1}$
s_0
$c_{outn/4-1}$
c_{outn-1}
s_{n-1}
$s_{3n/4}$
$c_{out3n/4-1}$
$s_{3n/4-1}$
$s_{n/2}$
$c_{outn/2-1}$
$s_{n/2-1}$
$s_{n/4}$

Carry-Lookahead-Addition

A type of adder which is used often in today's microprocessors due to its small constant factor which is asymptotically optimal in time, is the Carry-Lookahead-Adder.

The idea is to determine as early as possible for every position i, if a Carry is entered in this position or not.



This is of course simple for the most right (LSB) position and it gets more and more difficult when we go further to the left. The trick is now to predict (lookahead) the carry situation also at the positions of the highest significant bits from the knowledge of all operand bits that are involved.

Definition:

Let $M = \{0, 1, ..., n-1\}$, $a = a_{n-1}\, a_{n-2} \ldots a_1\, a_0$, , $b = b_{n-1}\, b_{n-2} \ldots b_1\, b_0$

We define $p : MxM \longrightarrow \{0, 1\}$ (partially defined)
$g : MxM \longrightarrow \{0, 1\}$ (partially defined)

for $i \in M$ $p(i,i) := a_i$ XOR b_i
$g(i,i) := a_i \bullet b_i$

for $i,j \in M$, $i<j$ $p(i,j) := p(j,j) \bullet p(i,j-1)$
$g(i,j) := g(j,j) + g(i,j-1) \bullet p(j,j)$

for $i,j \in M$, $i>j$ $p(i,j) :=$ undefined
$g(i,j) :=$ undefined

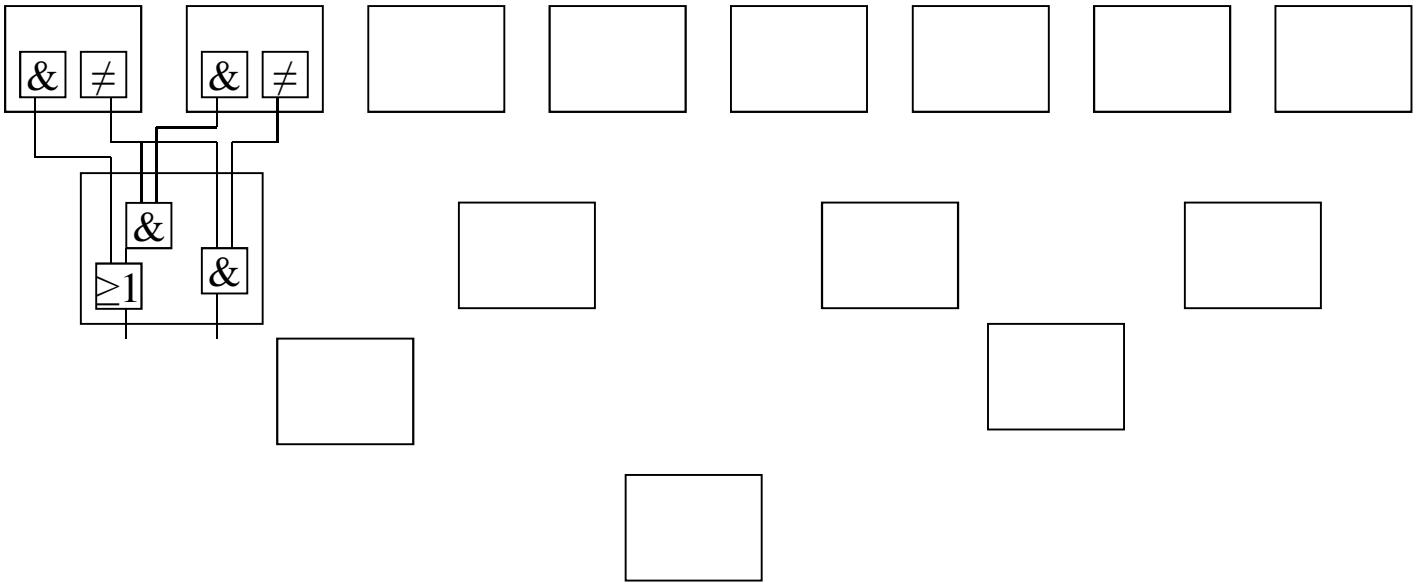
Clause:

(i) for $i < j$: $p(i,j) = p(j,j) \bullet p(j-1,j-1) \bullet p(j-2,j-2) \bullet \ldots \bullet p(i+1,i+1) \bullet p(i,i)$

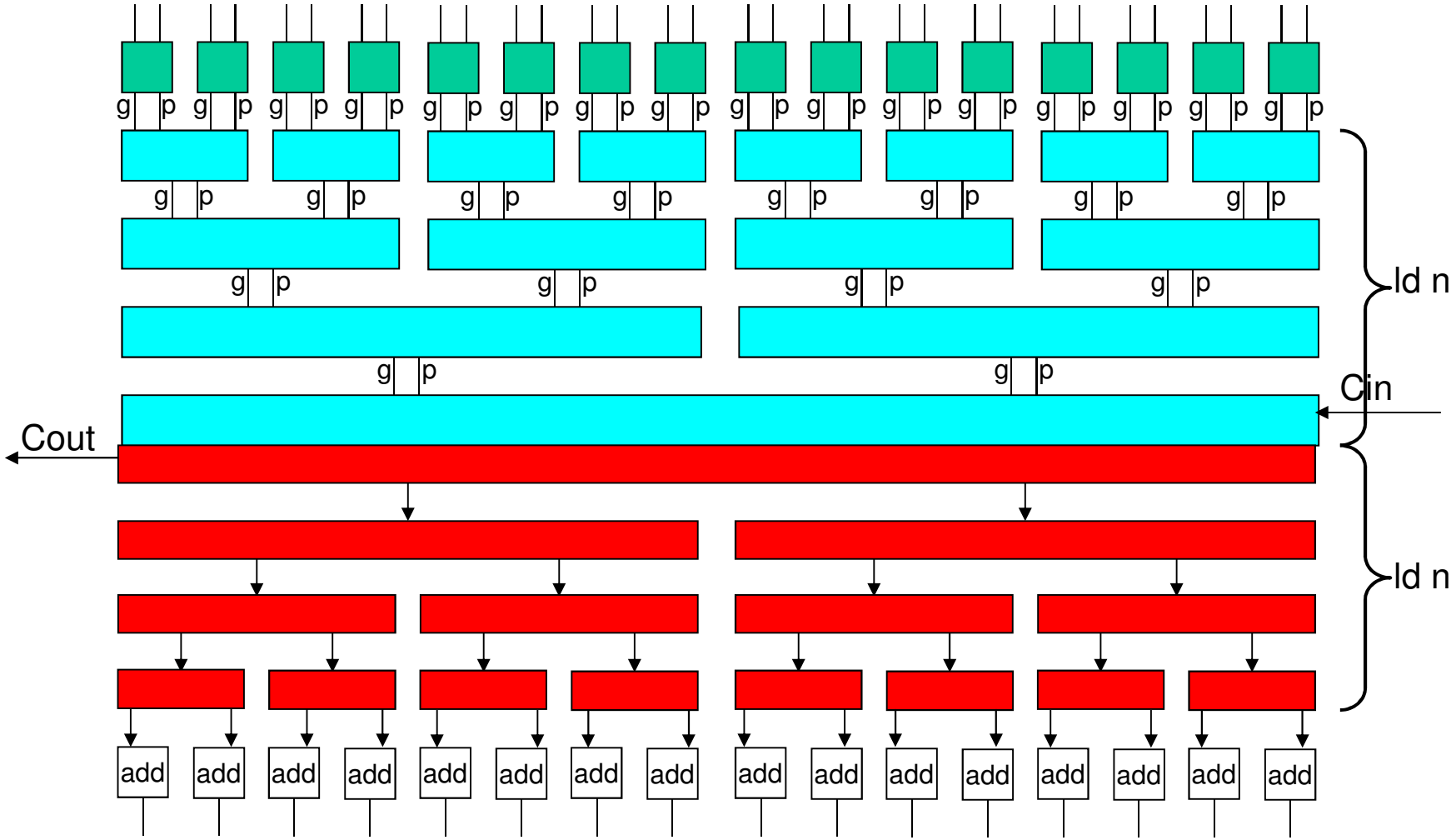
(ii) for $i \leq k < j$: $p(i,j) = p(k+1,j) \bullet p(i,k)$

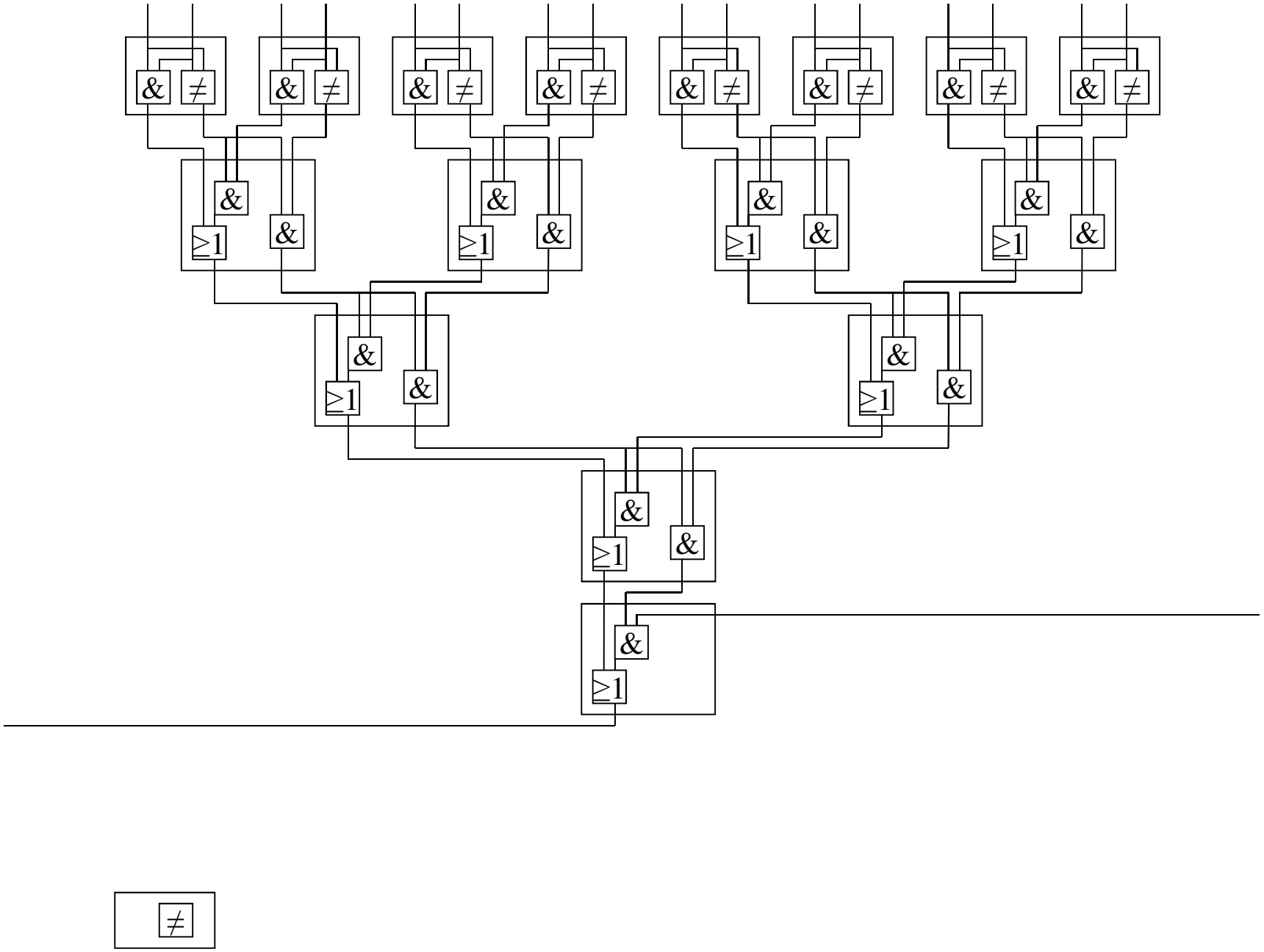
(iii) for $i \leq k < j$: $g(i,j) = g(k+1,j) + g(i,k) \bullet p(k+1,j)$

```
{Step 1: Calculate g(i,i), p(i,i) }
for i:=0 to n-1 pardo
   p(i,i):=a(i) XOR b(i);
   g(i,i):=a(i)·b(i)
endpardo;

{Step 2: Calculate g(i,j), p(i,j) }
for m:=0 to (log n)-1 do
   for k:=0 step 2^(m+1) to n-2^(m+1) pardo
      p(k,k+2^(m+1)-1):= p(k+2^m,k+2^(m+1)-1)·p(k,k+2^m-1);
      g(k,k+2^(m+1)-1):= g(k+2^m,k+2^(m+1)-1)+g(k,k+2^m-1)·p(k+2^m,k+2^(m+1)-1)
   endpardo;

{Step 3: Calculate c_i }
c(0):=0;
for m:=(log n)-1 downto 0 do
   for k:=0 step 2^(m+1) to n-2^(m+1) pardo
      c(k+2^m) := g(k,k+2^m-1)+ c(k)·p(k,k+2^m-1)
   endpardo;

{Step 4: Calculate s_i }
for i:=0 to n-1 pardo
   s(i) := a(i) XOR b(i) XOR c(i)
endpardo;
```


&
≠
&
≠
&
≥1
&

The Carry-Lookahead-Adder




&
≠
≥1

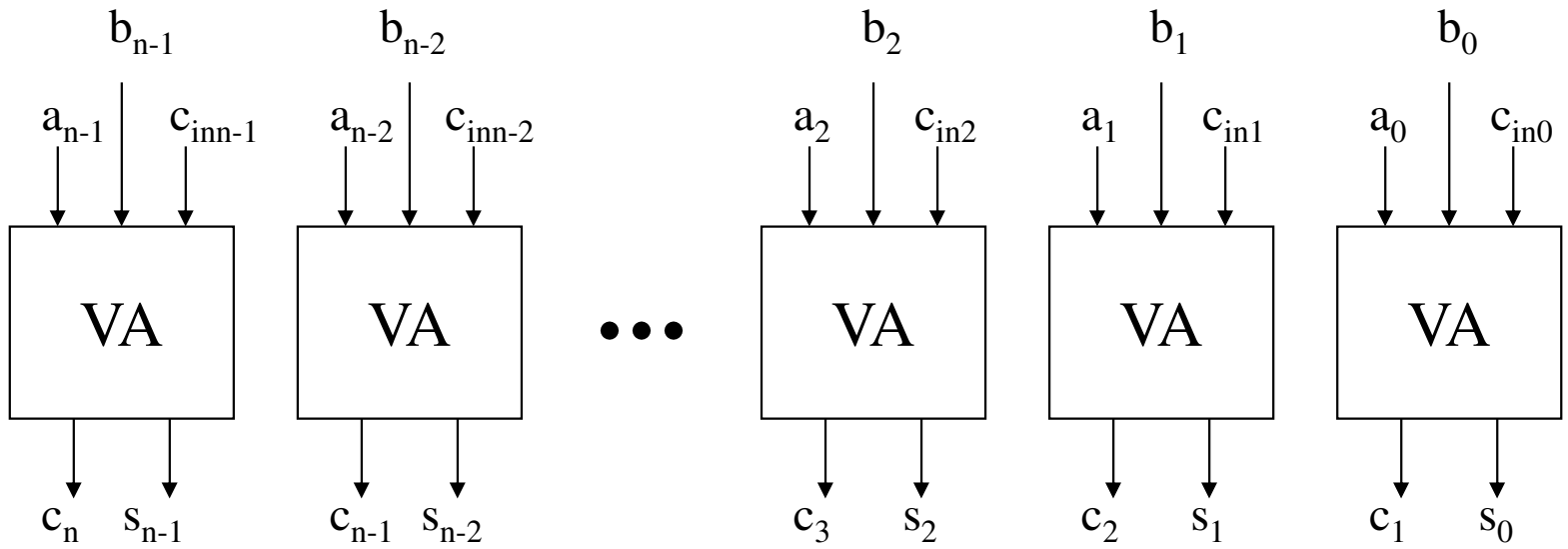
Carry Save Adder

A different solution is the usage of a redundant number representation for temporary results. This technique ensures that an eventually occurring carry only influences a smaller area but not causing a chain reaction like in the ripple carry adder. The hereby introduced adder is named as **carry-save-adder**.

The idea is not to add two operands to one result but ***three*** operands to ***two*** results. For our understanding, this sounds unnatural in the first place but it has the advantage of a very simple and fast realization. The application of such a carry save adder always makes sense if one wants to do many additions one after another, and this is important in multiplication calculations. Hence, we will see now how an extremely fast multiplicator is done by using carry save adders.

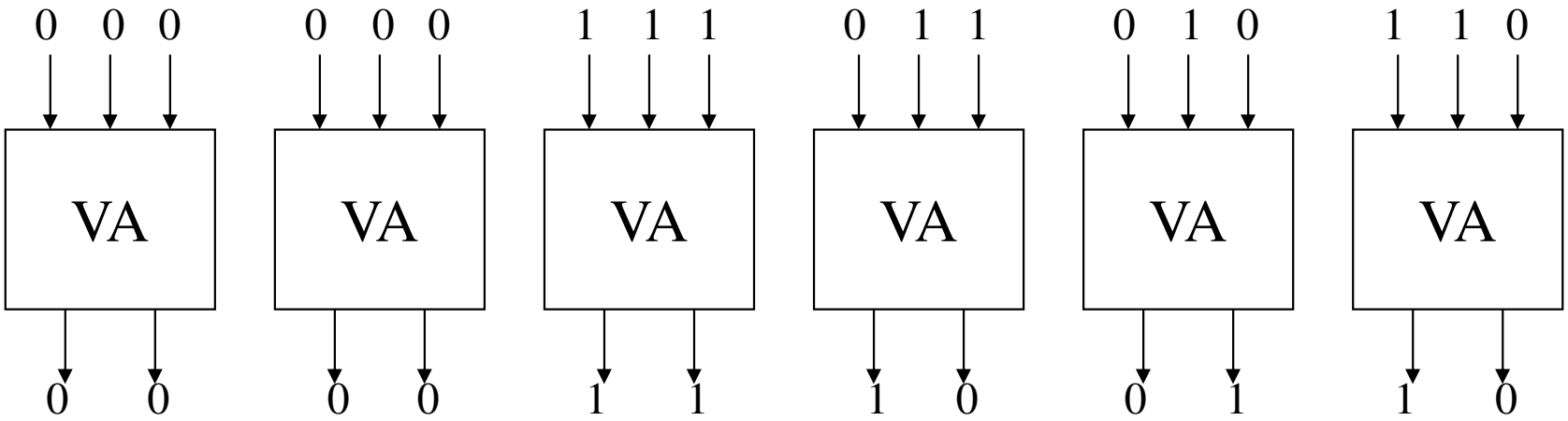
The structure of a carry save adder is only the parallel wiring of n full adders. These are not connected in a row but instead, each delivers two outputs, a s-bit and a c-bit. Out of these, s-word and a c-word are formed. These two words represent the result word. A zero is added to the least significant digit of the c-word and a zero is added to the higher significant digit of the s-word.

Hence, the sum of s and c-word is equal to the sum of the three operand words a, b, c_{in}.


b_{n-1}
b_{n-2}
b_2
b_1
b_0
a_{n-1}
c_{inn-1}
a_{n-2}
c_{inn-2}
a_2
c_{in2}
a_1
c_{in1}
a_0
c_{in0}
VA
VA
VA
VA
VA
c_n
s_{n-1}
c_{n-1}
s_{n-2}
c_3
s_2
c_2
s_1
c_1
s_0

Example:

We add the words a = 001001, b = 001111, c_{in} = 001100



Hence, the result words are c = 011010 and s = 001010. If we add these three operands in the decimal system, the result is 36. We will receive the same result if we add up c and s. One can observe that for this kind of addition there is no carry to ripple. Hence, the time is t_{VA} and hence O(1).

Where does it makes sense to use this kind of addition?

Imagine that we have to add a bunch of 64 numbers. We can add them up using 62 carry save additions and finally calculate 2 result words. We have to add these up with the “real” adder (e.g. carry select adder). The 62 additions require 62 * t_{VA}. The last addition require 8 * t_{VA} + 7 t_{MUX}. Hence, the total time adds up to $70t_{VA}$ + 7 t_{MUX}. If we would have done the entire addition using a carry select adder, we would need 63 * (8 t_{VA} + 7 t_{MUX}) = 504 t_{VA} + 441 t_{MUX} .

In this example, one can see how much more efficient the carry save addition is. Generally: If we want to do an addition of the length n-bit, we need the time O(n*m), using a ripple carry adder and with a Carry-Select-Adder, we need the time $O(n^{1/2}*m)$ and with a Carry-Save-Adder (plus a subsequent Carry-Select-Adder for the last step), we need the time $O(n^{1/2}+m)$.

The optimal time for an addition of two numbers can be achieved using the so-called Carry-Lookahead-Adder. This only needs the time O(log n) for an addition. Due to a lack of time, this type of adder will not be further explained in this part.

Multiplication

Using the school method for each digit of the operand, the multiplication with the respective digit of the other operand is calculated. Next, these products are added. This is where we can use the carry-save adder because now we have the case of a huge amount of operands needed to be added up.

Example:

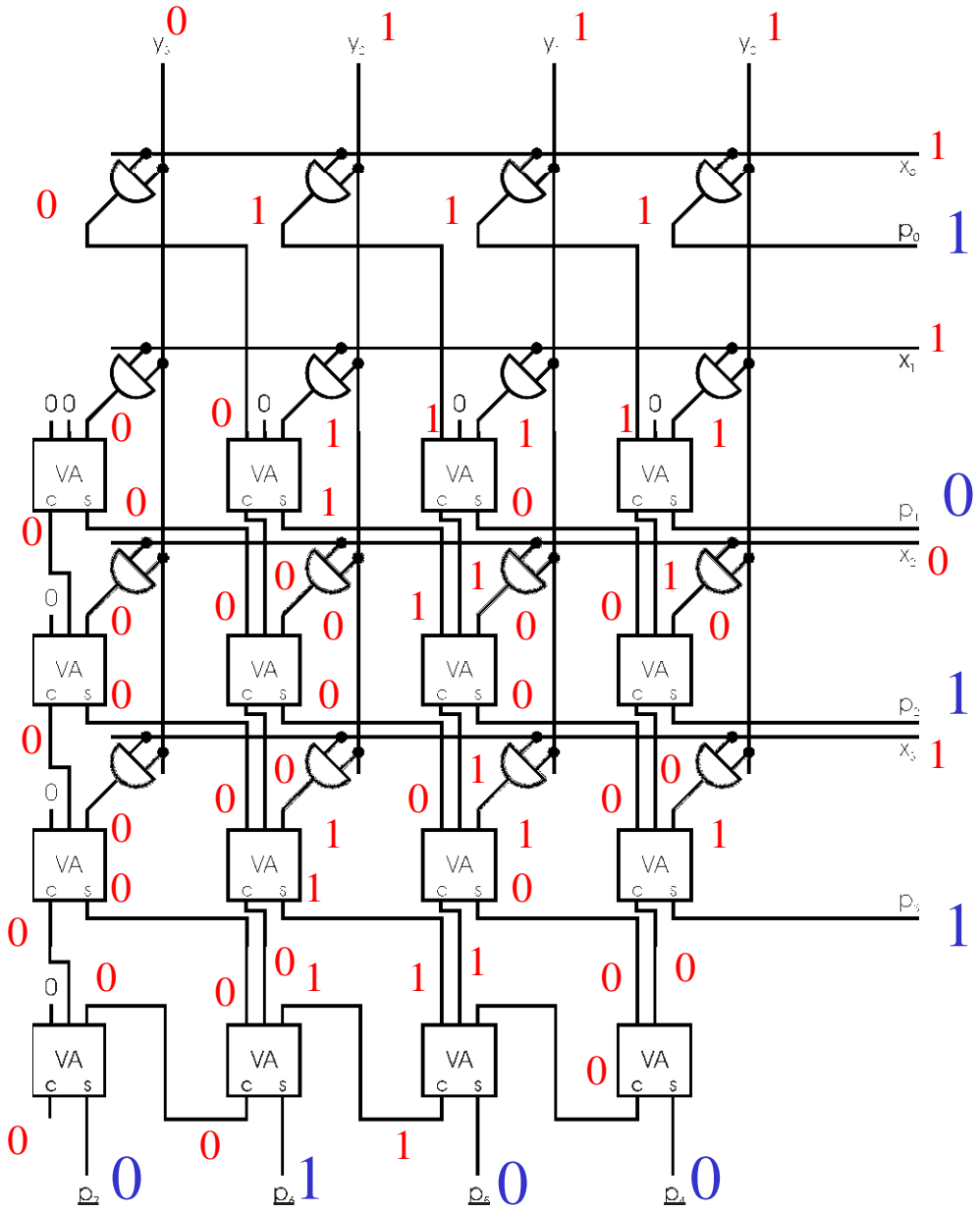
```
10110011 * 10010111
    10110011
     00000000
      00000000
       10110011
        00000000
         10110011
          10110011
           10110011
-----------------------
0110100110010101
```

A Carry-save multiplier for two operands of the length n consists of n^2 AND-Gates. These AND gates perform the required 1-bit multiplications at the same time (the binary multiplications of 1-bit numbers is the logic AND). Everything else that is left adds up all the product parts (like in the school method). This happens in the known 3 out of 2 operands manner, which we have learnt just now for the carry save adder. Finally, for the highest significant n-bits (a classical) 2 out of 1 operand addition is required. This can be done using a conventional adder.

How long is the computation time for such a multiplication? We have to search this combinatorial circuits for the “critical path”, the path where a signal has to go through the maximum amount of switching elements before the final result is calculated. This path consists of n-2 steps. At each step, a new operand is added, plus the n-1 full adders where a carry has to go through at the last addition (in case of a ripple carry adder).

An example for such a 4-bit multiplier can be seen in the next slide.

4-Bit-carry-save-Multiplizierer



Division

Some processors have their own division unit. Normally, two words are used to represent the temporary results which is similar to the carry save adder.

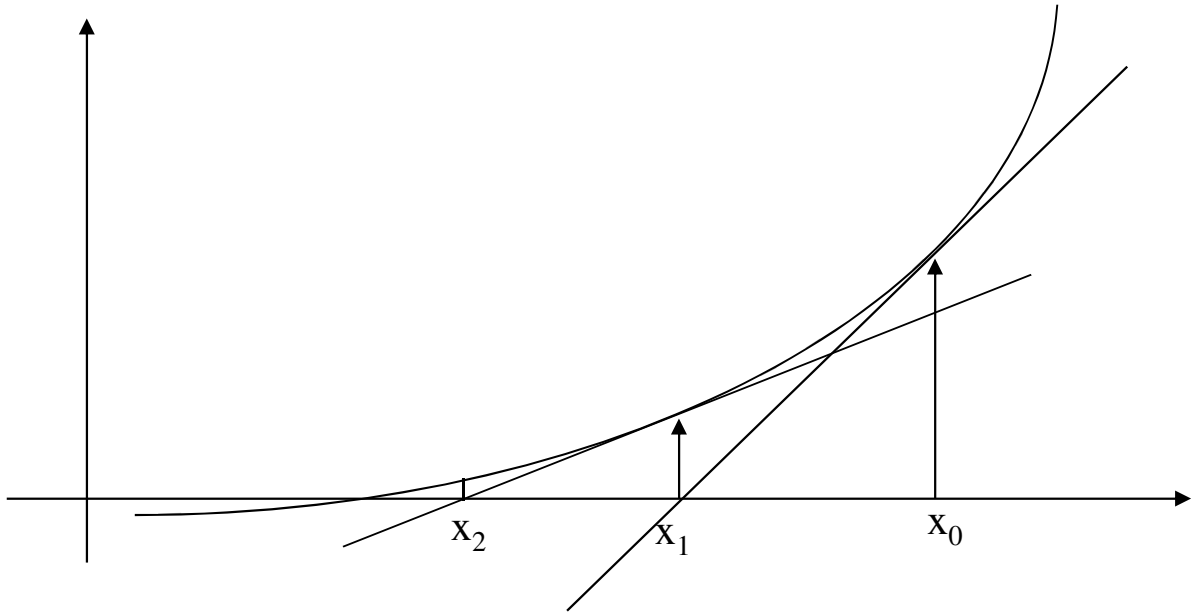
On the other hand, a division is a seldom-used operation. Hence, (make the common case fast) many processors do not use their own hardware for division but instead implement it in software. A common method for that is the Newton-Raphson method which we want to learn here.

Let me mention beforehand that early computers (< 1980) computed the division (usually also in software) using the school method, i.e. the result bits are calculated one after another by comparing the divisors with the remaining higher significant digits of the dividend (number to be divided). If the divisor is greater, the resulting bit is 0, else it is 1. In case it is 1, the divisor is subtracted from the highest significant digits of the dividend and the next digit of the dividend will be used for the next resulting digit.

This method is of course in O(n), if the dividend has n digits. More exactly, one needs n steps and in every step, a comparison and subtraction have to be done. That was proven to be too slow in the case of increasing computation power of computers. Hence, one searches for methods leading to the same results in fewer steps.

The idea of the Newton-method is the approximation of the zeros in a function by constructing a series of converging values that quickly approach zero. One starts by drawing a tangent to the function with an estimated starting value, where the x-coordinate of the intersection point is used for the subsequent calculation of the following function value. And next, draw a tangent to the previous calculated function value and so on. Once the function fulfills to certain requirements and if the starting value is adequately chosen, these series converge towards the zeros of the function.

Example:



For two words x_i and x_{i+1} in the following yields:

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

Setting the above equation in terms of x_{i+1} yields

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

Now, let's have a look at f(x) = 1/x - B . This function has a zero in 1/B. If we use this on the above formula, it results in:

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - B}{-\frac{1}{x_i^2}} = 2x_i - Bx_i^2$$

With that, we have a very simple iterative formula which only needs two multiplications and one subtraction for one iteration.

But how many steps do we need or how fast are these series converging?

Let's us have a look at the error ε, the difference of the series value x_i to the zero in 1/B that we are searching for. It yields:

$$x_{i+1} = 2x_i - Bx_i^2 = 2\left(\frac{1}{B} - \varepsilon\right) - B \cdot \left(\frac{1}{B} - \varepsilon\right)^2 = \frac{1}{B} - B \cdot \varepsilon^2$$

Hence, the error is only $B\varepsilon^2$ after the next iteration. . Now if B is between 0 and 1, that means that we have a quadratic convergence, more exactly, if the result has reached a tolerance of m bits after the i-th iteration, then after the i+1-th iteration, it has the tolerance of 2m bits.

That is why we have to make B between 0 and 1 and that x_0 has a tolerance of 1 bit. Then, x_1 will have a tolerance of 2 bits and x_2 of four bits etc., x_i has then the tolerance of 2^i bits.

Assuming that we have to calculate a/b, then we can do this with a multiplication as a * 1/b whereby we have to be able to calculate the inverse of b (1/b).). If b lies between 0 and 1 and the first digit after the decimal point, is a 1 (hence normalized) then the above iteration is applicable. If not, we have to calculate $B = 2^k * b$ using an adequate k so that B is normalized. Then, we calculate 1/B and multiply that with 2^{-k} (bit shifting).

Conclusion: By using the iteration, we reduce the complexity of 2n operations to 3log n operations, more exactly, log n iterations and in each iteration, three operations are needed. For a 64-bit division, that means a reduction by 128 to 18 operations.