Parallel Multiplication of Two n-bit Integers and The Addition of n n-bit Integers in $O(log_2(n))$ Time

Matthew Zupan

22/12/2022

Contents

1 Introduction

The advent of multiplication algorithms faster than the typical n^2 method written on paper, such as the methods developed by Karatsuba, Toom - Cook, Schönhage - Strassen [1] and recently Harvey - van Der Hoeven [2] have all provided theoretical improvements on the time complexity of multiplication. However, presenting with a larger constant $k * O(.)$, such algorithms may not be faster than their more standard counterparts until numbers presenting with a larger proportion of digits is reached. The case of the Harvey - van Der Hoeven algorithm achieving multiplication in $O(nlog(n))$ is an example of this, currently remaining only of theoretical interest with no practical implementation due to the intractably high constant.

Then, aside from algorithmic improvements and a reduction in the multiplying constant, the next most obvious way to reduce the time taken for multiplication of two numbers is to perform the multiplication in parallel. The following work proposes a scheme as well as a method of construction for an electronic parallel multiplier of two positive integers A and B in time proportional to $O(log(n))$, where n is the number of binary digits in A or B, treating A and B as the same size.

As a basis for improvement, we will consider the traditional schoolbook method of multiplication where a table of consecutive multiplication rows are formed and them summed together as the result. This method runs in the order of $O(n^2)$ numerical processing steps and an example is shown below (the 0's in bold are implicit in this method):

Fig 1.1: Schoolbook Multiplication Method.

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Number1:									1	0	1	1	0	1	1	0
Number2:									1	1	0	1	1	0	0	1
Multiplication Rows:		**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	0	1	1	0	1	1	0
		0	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0	0	**0**
		0	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0	0	**0**	**0**
		0	**0**	**0**	**0**	1	0	1	1	0	1	1	0	**0**	**0**	**0**
		0	**0**	**0**	1	0	1	1	0	1	1	0	**0**	**0**	**0**	**0**
		0	**0**	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**
		0	1	0	1	1	0	1	1	0	**0**	**0**	**0**	**0**	**0**	**0**
		1	0	1	1	0	1	1	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Result:	1	0	0	1	1	0	1	0	0	1	0	0	0	1	1	0

Our goal hereafter will be to utilise this basic multiplication scheme along with some observations and tricks, forming a parallel circuit with fan out of at most

2. The method and justification of which will be addressed in section 2. We will also address in Section 3, how to make use of the multiplication method (with some differences) to perform addition of n n-bit positive integers in log time. This may be useful in the situation where we have a list or table of numbers that must all be summed together. Should we have less than n numbers, we may consider summing the m numbers we have with $n - m$ numbers set to 0's, with all other numbers left padded with 0's should they be less than length n.

2 An $O(log(n))$ Multiplication Scheme

The following multiplication scheme will be restricted to multiplication of two integers A and B both of size $n = 2^k$, where $k \in \mathbb{Z}^+$. This choice was made due to the well known fact that the result of multiplication has a length of at most $n' = 2n$ and therefore its' representation remains a power of 2, being $n' = 2^{k+1}$. An important note is that for any A or B with length less than 2^k, we can treat it as being left padded with 0's such that the length + padded length = 2^k. For example: 110100 having 6 digits is expressed as 00110100 having 8 digits, where $k = 3$, that is: $8 = 2^3$.

2.1 Divide and Conquer With Column Reduction

Returning to the example of Fig1.1. We are required to sum each column before calculating the result. Observe that given we are using binary. A column sum only requires we count the number of 1's that are present in each column. From our example, given there are 8 rows, we can have a minimum of 0 1's and a maxumum of 8 1's present in a column. This maximum sum of 1's could be represented with no less than $\lfloor log_2(8) \rfloor + 1 = 4$ bits, that is: 1000. Expressing the result of these column sums as $s_i = d_1 d_2 d_3 d_4$ and using the example of the sums of column 8: $s_8 = 0011$, we take note that the least significant bit d_4 contributes to the result at column 8, d_3 to result at column 7, d_2 at 6 and lastly d_1 at 5.

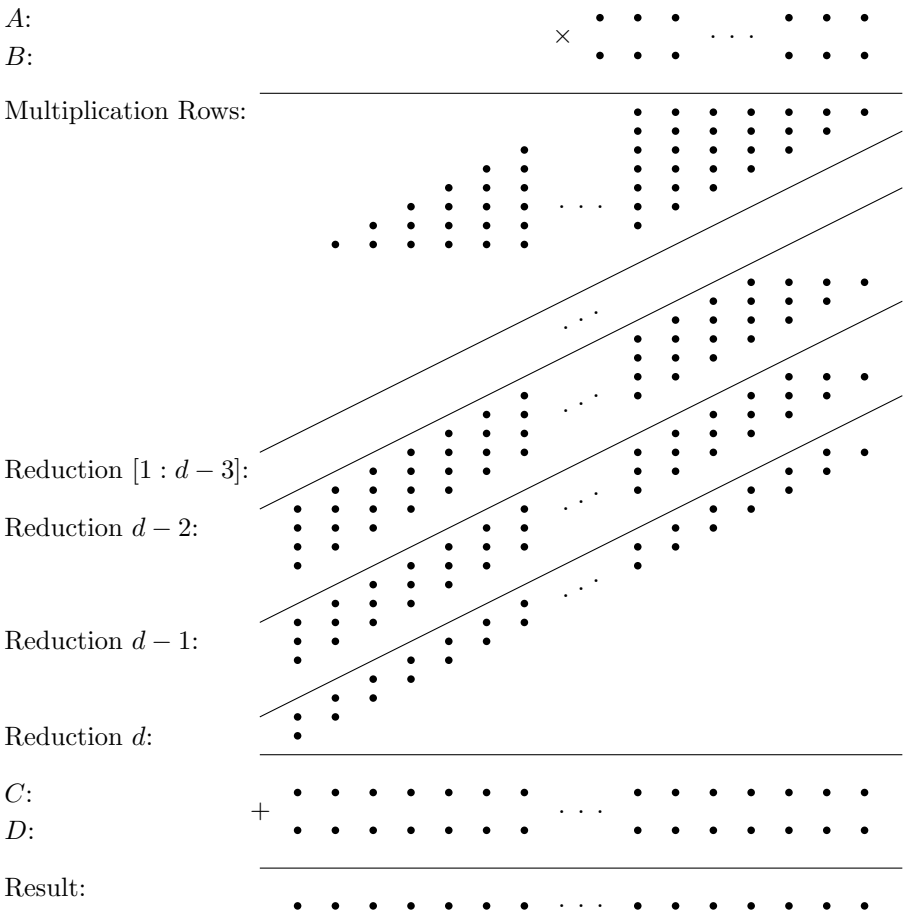
Generalising this idea: Multiplying together A and B of size n will generate n rows, each column therefore containing n bits to be added together will require $|s| = \lfloor log_2(n) \rfloor + 1$ bits, where for each i, the output bit d_j, $1 <= j <= |s|$ is assigned to column $i - (|s| - j)$. This is shown diagrammatically in figure 1.2. Note that whenever $i - (|s| - j) < 0$, the output can be ignored.

Fig 1.2: Summation of Columns Produces Corresponding Digit Outputs.

Index:	0	1	2	3	4	5
Multiplication Rows:		**0**	**0**	1	1	1
		0	0	0	0	**0**
		1	1	1	**0**	**0**
Digit Outputs:					0	1
				0	1	
			1	0		
		0	1			
	0	1				

The multiplication rows from Fig1.2 are said to be *reduced* when summed and represented as the digit outputs. These digit outputs present themselves as a structural copy of the multiplication rows from before and therefore, until the digit outputs are of length 2, they can be recursively reduced in the same way as the original multiplication rows. We stop the reduction at length 2 because the sum of two single digit inputs may lead to a 2 digit output, leaving us back at length 2. Instead, at length 2, we simply treat the diagonal outputs as two numbers C and D (both left padded with a 0 and the top diagonal right padded with a 0 to make both numbers the same length $n' = 2^{k+1}$) and then compute the sum of $C + D$ in $O(log(n))$ time. Ignoring any implementation details for now, we show an example reduction of depth d in Fig1.3 (The author has independently developed this method but has learned the approach is referred to as a Wallace Multiplier or Wallace Tree [3], with some differences).

Fig 1.3: Column Reduction Scheme.

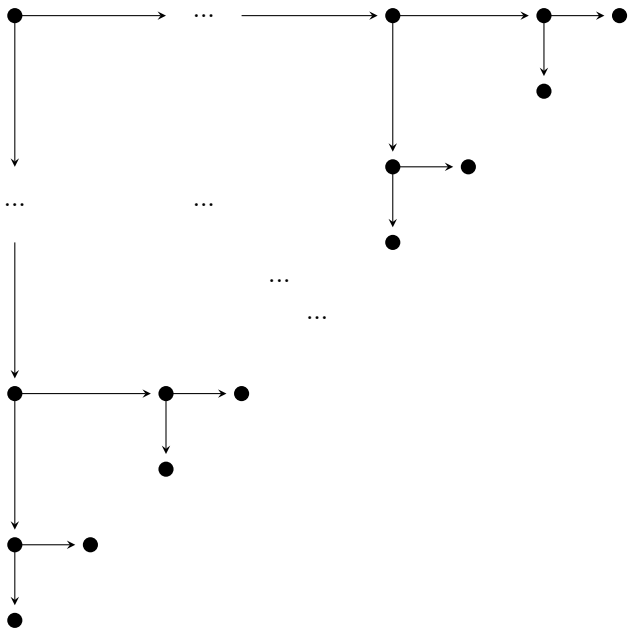


In order to achieve multiplication of A and B in $O(log(n))$ time, we require the formation of multiplication rows, the sum of all column reductions and the final addition of C and D each to take $O(log(n))$ steps. The summation of C and D in $O(log(n))$ is trivial by use of the carry lookahead adder developed in [4]. The formation of multiplication rows can also be achieved within this bound by creating n copies of A and B in $log(n)$ doubling steps, but will require some careful and customised circuit design specific to each n. Now what remains to be shown is that the sum of all column reduction phases is bounded by $O(log(n))$ which will be covered in **subsection 2.3**.

2.2 Formation of Multiplication Rows

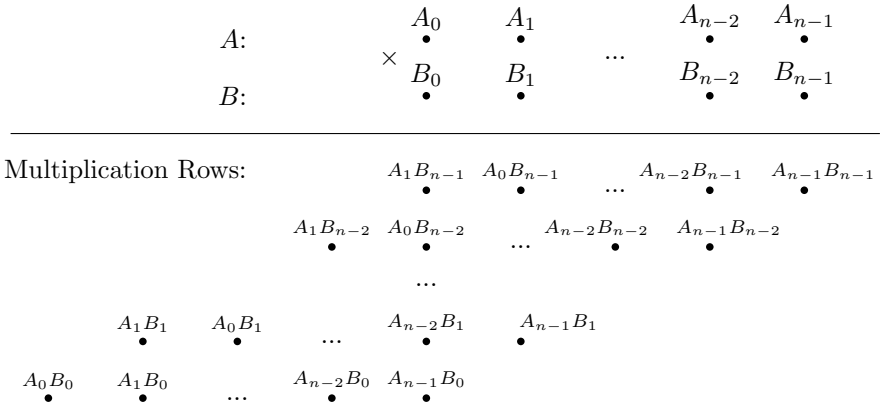
The formation of multiplication rows in parallel firstly requires the numbers A and B be split into n copies such that n^2 processing elements may compute their local value of $A * B$ simultaneously. Copying of A and B can be done in $O(log_2(n))$ time with the structure of a perfect binary tree shown in Fig 2.1:

Fig 2.1: Perfect Binary Tree Splitting Structure.



The splitting tree structure is over simplified and the outputs will need to be reconciled with the regular layout of multiplication rows, where each row i consists of all $A_i * B_j$, $0 \leq j \leq n-1$ as follows:

Fig 2.2: Splitting Tree Output Targets.



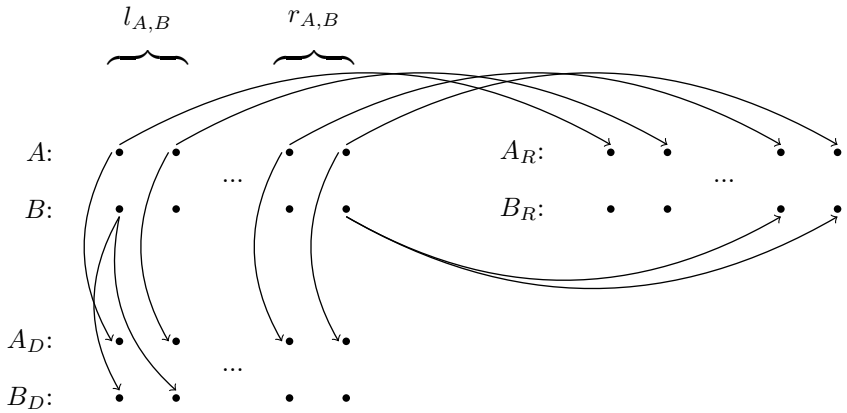
Observe that each row r_i (ordered from the top descending) contains all A_i (ordered from left to right) and only one B_j, where $j = n - i + 1$. In order to satisfy this observed requirement during the copying process, we can copy A and B according to the following rules:

- The size of A and B is n, therefore, define the left of A, l_A and B, l_B as any element at index $i < n/2$ and the right of A, r_A and B, r_B as any element at index $i \geq n/2$.
- Define A_D, B_D as "A down, B down", meaning, the downwards copy of A and B, and A_R, B_R as "A right, B right", meaning the rightwards copy of A and B.

At any stage of the process where a copy A and B is at a height less than $log_2(n)$ from the root of the tree:
- Split each element A_i and send one copy each to the corresponding index i at A_D and A_R.
- Split each element $(l_B)_i$ and send a copy to index $2i$ and $2i+1$ in B_D.
- Split each element $(r_B)_i$ and send a copy to index $2i-n$ and $2i-n+1$ in B_R.

The ruleset is conveyed diagramatically as fig 2.3, with many lines omitted to avoid excessive clutter.

Fig 2.3: Splitting Tree Ruleset.



The ruleset generates the splitting tree with terminal nodes (along the diagonal, bottom left to top right) labelled $N_j, 0 \leq j \leq n-1$ corresponding to the formation of copies of A and B, each containing a single copy of all A_0 to A_{n-1} in left to right order and n copies of B_j. We briefly prove this by induction.

Base case $n = 1$:

We are at height $log_2(n) = log_2(1) = 0$. Since this is distance 0 from the root, we have N_0 containing A_0 and B_0. No further action is required and the base case is true.

Inductive step:

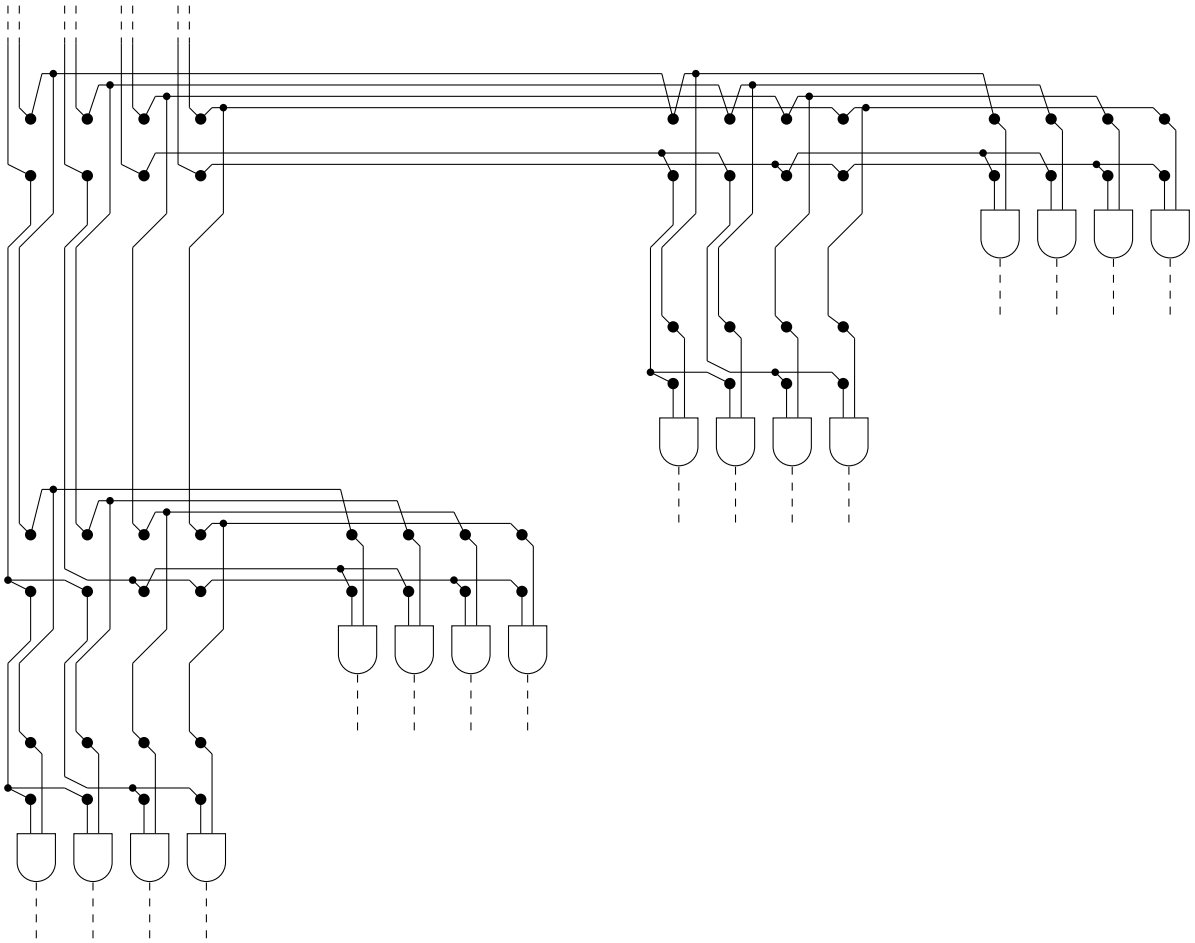
Assume for some value n, we are at height $log_2(n) - 1$ from the root and parent(N_j) contains: all ordered A_0 to A_{n-1}, where $(l_{\text{parent}(N_j)_B})$ contains $n/2$ copies of $[B_j$, if $N_j = \text{parent}(N_j)_D$, or B_{j-1}, if $N_j = \text{parent}(N_j)_R]$ and $r_{\text{parent}(N_j)_B}$ contains $n/2$ copies of $[B_{j+1}$, if $N_j = \text{parent}(N_j)_D$, or B_j, if $N_j = \text{parent}(N_j)_R]$.

Then at height $(log_2(n) - 1) + 1 = log_2(n)$, by the inductive hypothesis: N_j contains all ordered A_0 to A_{n-1} and $2 * n/2 = n$ copies of $[l_{\text{parent}(N_j)_B} = B_j$, if $N_j = \text{parent}(N_j)_D$, or $r_{\text{parent}(N_j)_B} = B_j$, if $N_j = \text{parent}(N_j)_R]$ at all indices $[2(0 \leq i < n/2) \cup 2(0 \leq i < n/2) + 1 = 0 \leq i \leq n$, or $2(n/2 \leq i < n) - n \cup 2(n/2 \leq i \leq n) - n + 1 = 0 \leq i \leq n]$ respectively. Thus the inductive hypothesis is true and this completes the proof.

Outputs to form the multiplication rows are then expressed in the form $A_i \wedge B_j$ for $0 \leq i, j \leq n-1$ with the use of AND gates for the copies of A and B at

height $log_2(n) - 1$ from the root. An example is shown in Fig 2.4.

Fig 2.4: $n = 4$ Splitting Example.



Outputs corresponding to each column of the generated set of multiplication rows are then passed in as inputs to a carry shower circuit (one for each column). Due to the fact that not all columns are the same size, we can still make use of the exact same carry shower circuit for each column by treating any non existent input lines as a "0" input to the circuit. This will be useful in maintaining signal synchronisation among the components.

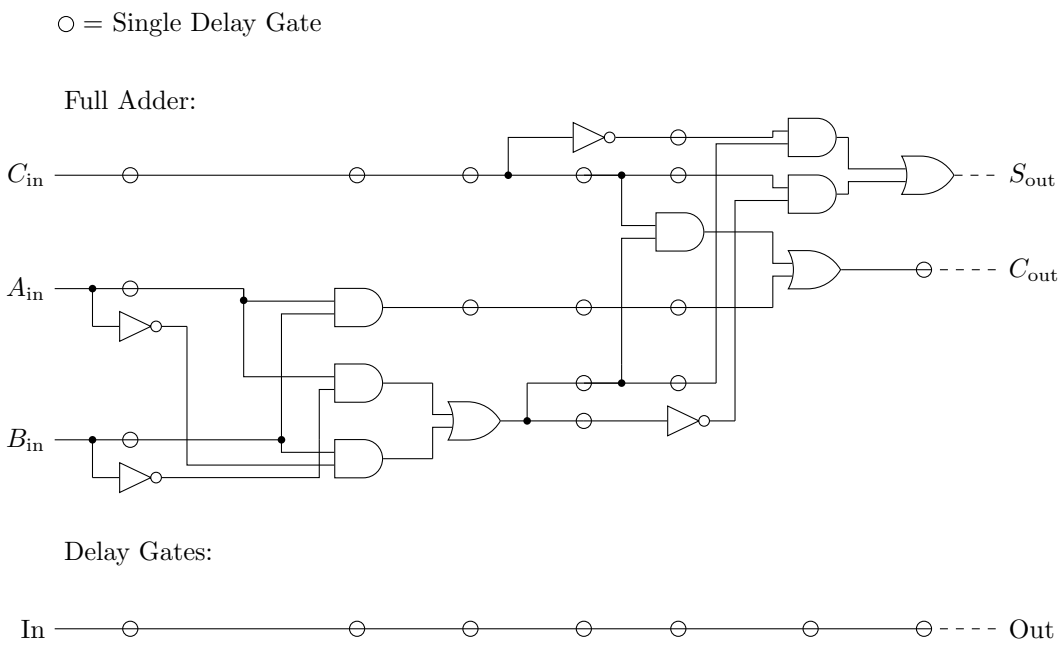
2.3 Column Reduction Via Carry Shower Circuits

Before examining column reduction in more detail, a specific type of circuit will need to be introduced. This is the Carry Shower Circuit as specified in [5] (the time complexity of this circuit is specified as $T = \lceil log_3(n) \rceil + \lceil log_2(n) \rceil - 2$). Here we will provide a detailed method of construction such that they may be used for the purpose of column reduction and as a quick summary of their purpose: They are required to perform the summation of the number of 1's in a binary sequence of length n in log time and output this sum in a binary representation. No proof of correctness for the carry shower will be given as this has been addressed by other authors external to this paper.

Constructing Carry Shower Circuits:

We begin with a description of the carry shower circuit. Take n input signals consisting of 1's and 0's, for our purposes, the carry shower circuit will consist of a number of full adders and delay gates such that signals propagating down the circuit are "synchronised". The full adders used for this will each induce a delay of 7 units of time and therefore each delay gate used in our desciption of the carry shower will be a condensed representation actually consisting of 7 delay gates chained together in a line. This is shown in the following diagram Fig 2.5 (on the next page) using single delay gates, where C_{in}, C_{out} and S_{out} refers to the carry in, carry out and sum out respectively:

Fig 2.5: Full Adder and Delay Gate.



A high level explanation of how to construct the carry shower circuit is provided, first it is necessary to describe rules regarding the most important components: The full adder and the seven delay gates.

- Each full adder or delay gate is asigned a label L from 1 to m, where $m = \lfloor log_2(n) \rfloor + 1$, regarding n as the number of inputs to the carry shower circuit. Each label corresponds to a signal line which represents its' digit output. A delay gate with label L must accept one and only one input from an input line L. A full adder with label L must accept only 3 input lines of label L on any given layer of processing, unless this is the last layer of processing for line L. Then it may accept 2 or 3 input lines only, with the last line set to 0 if there are 2 inputs.

- A delay gate accepting input line L will always produce a single output line L. A full adder accepting input lines L will produce two output lines. The top output being the sum, with label L, and the bottom line being the carry, with label $L + 1$. A completed digit line will be propagated with delay gates separately to the rest of the processors until the entire process is completed.

- A digit input line L is considered completed when one single output line L exists among a current layer of processing and the line $L - 1$ is completed. In the case of $L = 1$, a hypothetical line $L = 0$ can be considered completed for

consistency in the rule set.

To construct the circuit (of which there are many ways), the following guidelines are useful (layer layout ordering - top to bottom):

- Treat the set of inputs as a hypothetical layer 0 with n outputs of label $L = 1$. Then at each layer until the last digit line is completed:

- Count the number of previous layer outputs for each line L (the count of all previously completed digit lines is 0).

- Propagate all completed digit lines with a delay gate for each line.

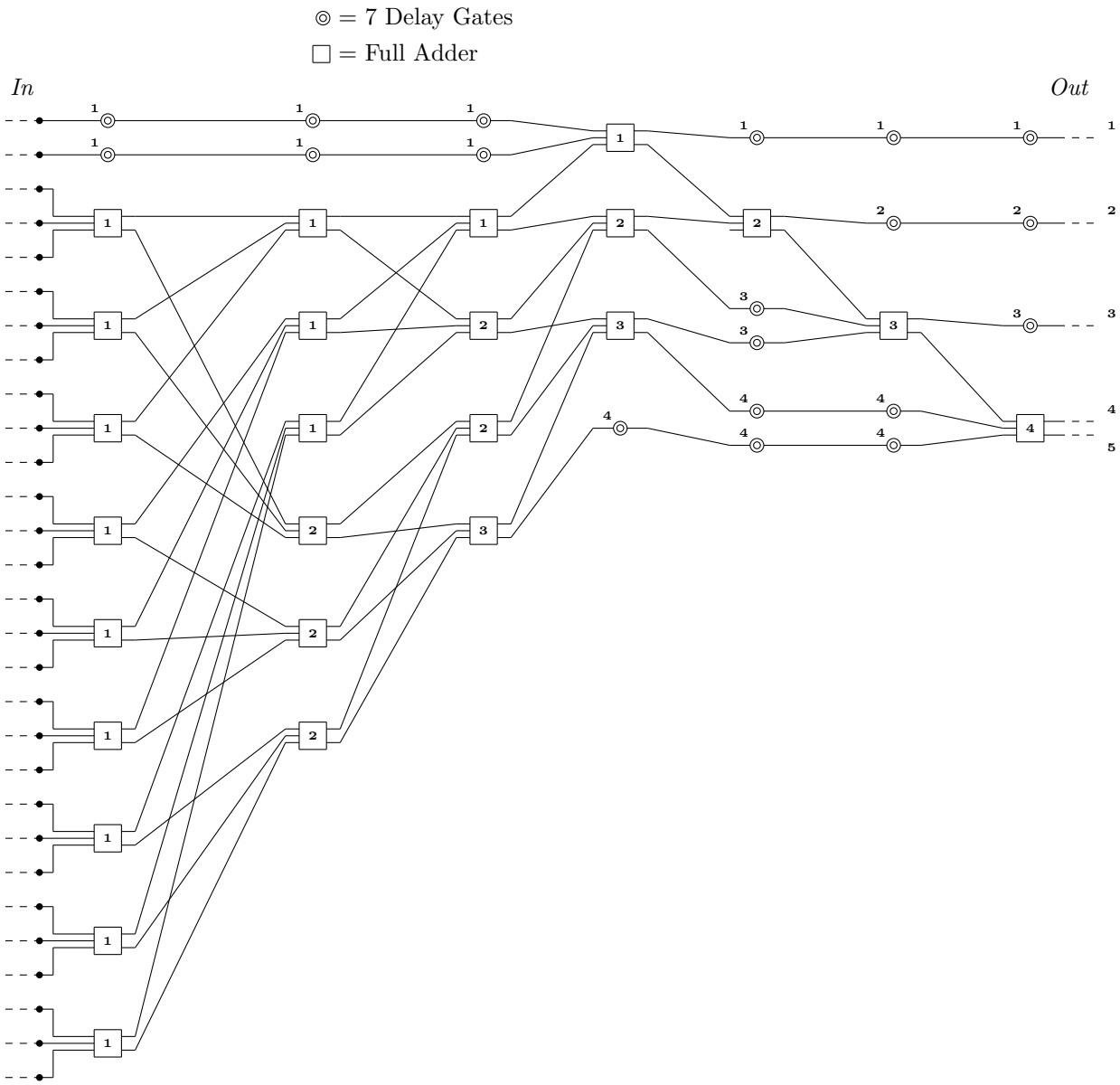
- For each L, take the result of integer division of the count of L by 3 and the remainder $mod3$. The remainder corresponds the number of delay gates, lay these out first. Then lay out the number of full adders according to the integer division result.

- If the previous layer resulted in a completed digit line, connect the top output of the top full adder in the previous layer to the last delay gate in the completed digit section of the current layer. Then connect the outputs of the remaining delay gates in the previous completed digits layer to the current layer.

- For each output in the previous processing layer, connect the output of label L to the next available input (top to bottom) of type L in the current layer.

The following diagram on the next page represents how the previously outlined design rules would construct a carry shower circuit with 29 inputs.

Fig 2.6: Carry Shower Circuit With 29 Inputs.



Proof of $O(log(n))$ Time Complexity:

We apply column reduction to the multiplication rows repeatedly until all columns span a distance of two rows. Each reduction stage is expressed as a term in square brackets as in the following function of space, where $lg(.)$ is shorthand for $log_2(.)$:

$$f(n) = n + [\lfloor lg(n) \rfloor + 1] + [\lfloor lg(\lfloor lg(n) \rfloor + 1) \rfloor + 1] + ... + [\lfloor lg(...\lfloor (lg(n) \rfloor + 1...) \rfloor + 1]$$

The number of reduction stages (terms) is referred to as the *depth*, d of $f(n)$. Each term gives reference to the space taken by reduction, but we will assert that the time complexity is proportional to the space complexity. This will be shown by making use of the carry shower circuit, for which its purpose is to output the summation of n single bit values in time $T = \lceil log_3(n) \rceil + \lceil log_2(n) \rceil - 2$ (while we use our own full adder as shown previously with circuit depth 7 and hence time delay $\Delta t = 7$, the paper [5] considers a full adder to have unit time delay $\Delta t = 1$. This difference is inconsequential in the following arguments).

To simplify our argument for T, assume our function of time $T(n)$ to be $2lg(n) - 2$. Then, for some term $0 \leq j \leq d-1$ in our function of space $f(n)$, we allow time $T(n)_{j+1}$ to operate on reduction space $f(n)_j$. It follows that $T(f(n))_{j+1} = 2lg(f(n)_j - 1) - 2$ when $j > 0$, however, given $f(n)$ is not a reduction step when $j = 0$, $T(n)_{j=0+1}$ operates on the space of size n formed by the presence of n multiplication rows. Note $T(f(n))$ does not operate at depth $d + 1$.

The functions $f(n)_j$ and $T(f(n))_j$ are then paired for all reduction terms in the range $1 \leq j \leq d$. This is due to the fact that:

$$f(n)_{j+1} = lg(f(n)_j) + 1$$
$$2f(n)_{j+1} = 2lg(f(n)_j) + 2$$

Substituting in $T(f(n))_{j+1}$, we have:

$$2f(n)_{j+1} = T(f(n))_{j+1} + 4$$
$$T(f(n))_{j+1} = 2f(n)_{j+1} - 4$$

We may now generalise and state that $T(n) = 2f(n) - 4d$ since we subtract 4 for each term up to depth d, and taking it a step further, that $T(n) = O(f(n))$.

With this previous understanding, we can show the time complexity $T(n) = O(lg(n))$ via analysis of the reduction space complexity. Firstly we will need to establish an upper bound on the rate of growth that is manageable, and by

observing that n may only be some value 2^k, we can transform $f(n)$ to the sum of reduction terms (omitting the floor) with the form:

$$f(2^k) = [lg(2^k) + 1] + [lg(lg(2^k) + 1) + 1)] + ... + [lg(...lg(2^k) + 1...) + 1]$$

Which then simplifies to:

$$f(2^k) = [k + 1] + [lg(k + 1) + 1)] + ... + [lg(...(k + 1)...) + 1]$$

Due to the process of column reduction, $f(2^k)$ terminates when $f(2^k)_d = 2$. Notice that $f(2^k)_d = lg(f(2^k)_{d-1}) + 1$, similarly, $f(2^k)_d = lg(lg(f(2^k)_{d-2}) + 1) + 1$. Beginning at $f(2^k)_d$, we only add a new term at new depth $d' = d + 1$ when $f(2^k)_d = 3$, and as such this new term at d' is equal to 2. $f(2^k)_{d'}$ may then equal 3 when $f(2^k)_d = 4$ and $f(2^k)_{d-1} = 8$ and so on.

We will assume that $f(2^k)_{d-2}$ grows linearly with k to limit the upper bound of growth for some sufficiently large k, such that $d > 3$ (although it grows proportional to the log of the previous term), then we can use as follows from previous observations, that $f(2^k)_{d'} = lg(lg(lg(f(2^k)_{d-2}) + 1) + 1) + 1$, hence $d(2^k)$ grows at the same rate which we will prove is $O(lg(lg(k)))$.

A proof that some $f(x) = O(g(x))$ requires some constant multiple M and starting value n_0 such that $f(n) \leq Mg(n)$ for some $n \geq n_o$.

Let $d(2^k) = lg(lg(lg(2^k) + 1) + 1) + 1$ which simplifies to $lg(lg(k + 1) + 1) + 1$, and let $M = 2$ such that $g(2^k) = 2lg(lg(k))$. Then:

$$\begin{aligned} lg(lg(k + 1) + 1) + 1 &\leq 2lg(lg(k)) \\ 2^{lg(lg(k+1)+1)+1} &\leq 2^{lg(lg^2(k))} \\ 2(lg(k + 1) + 1) &\leq lg^2(k) \\ lg^2(k) - 2lg(k + 1) - 2 &\geq 0 \end{aligned}$$

If we were able to express this inequality in the form of a quadratic equation: $ax^2 + bx + c$, we could then easily solve for k. We will do so by overestimating the term $-2lg(k + 1)$, adjusting it to be $-2lg(2k)$. This is allowable when $2k \geq k + 1$, or more concisely $k \geq 1$ given the adjustment is constant.

Isolating k in the adjusted term, we have $-2lg(2k) = -2lg(k) - 2$ which yields a tractable inequality $lg^2(k) - 2lg(k) - 4 \geq 0$.

Let $x = lg(k)$, then we have obtained our quadratic form: $x^2 - 2x - 4 \geq 0$.

Solving for x and taking only the positive root, we find $x \geq 1 + \sqrt{5}$ and there-

fore $k_0 \geq 2^{1+\sqrt{5}}$, which we can round up to 10 such that k_0 is an integer. Thus we have shown there exists some $M = 2$ and $n_0 = k_0 = 10$ such that $d(2^k) \leq Mg(2^k), k \geq k_0$ and therefore $d(2^k) = O(lg(lg(k)))$.

In this last element of the proof, to show $T(n) = O(lg(n))$, we begin with $f(2^k)$ limited to depth $O(lg(lg(k)))$, given $T(n) = O(f(n))$. It is then sufficient to show $f(2^k) = O(g(2^k)) = O(k)$.

Making use of the geometric series summation: $x + \frac{x}{2} + \frac{x}{4} + ... = 2x$, we can split $Mg(k)$ with $M = 2$ into the sum: $k + \frac{k}{2} + \frac{k}{4} + ... = 2k$. With an infinite set of summands, we can then pair each term in $Mg(k)_j$ with a corresponding term in $f(2^k)_j$ in the range $1 \leq j \leq d$ and show for each term, there is some k_0 such that $2g(2^k)_j \geq f(2^k)_j$.

Generalising $2g(2^k)$, we express each term in the form $\frac{k}{2^j}$, and by playing with values of j, we see for:

$j = 1$:

$$\begin{aligned} \frac{k}{2} &\geq lg(k+1) + 1 \\ \frac{k}{2} - 1 &\geq lg(k+1) \\ 2^{\frac{k}{2}-1} - 1 &\geq k) \end{aligned}$$

$j = 2$:

$$\begin{aligned} \frac{k}{4} &\geq lg(lg(k+1)+1) + 1 \\ \frac{k}{4} - 1 &\geq lg(lg(k+1)+1) \\ 2^{\frac{k}{4}-1} - 1 &\geq lg(k+1) \\ 2^{2^{\frac{k}{4}-1}-1} - 1 &\geq k \end{aligned}$$

$j = 3$:

$$\begin{aligned} \frac{k}{8} &\geq lg(lg(lg(k+1)+1)+1) + 1 \\ \frac{k}{8} - 1 &\geq lg(lg(lg(k+1)+1)+1) \\ 2^{\frac{k}{8}-1} - 1 &\geq lg(lg(k+1)+1) \\ 2^{2^{\frac{k}{8}-1}-1} - 1 &\geq lg(k+1) \\ 2^{2^{2^{\frac{k}{8}-1}-1}-1} - 1 &\geq k \end{aligned}$$

We end up with the power tower $2^{\cdots^{\frac{k}{2^j}-1}\cdots-1} - 1 \geq k$ of height j. By inspection (and this choice is somewhat contrived), we see after choosing $k \geq 2^{j+3}$, we

have $2^{\cdots^{\frac{2^{j+3}}{2^j}-1}\cdots-1} - 1 \geq 2^{j+3}$, which simplifies to $2^{\cdots^{2^7}\cdots-1} - 1 \geq 2^{j+3}$ and when $j \geq 1$, the left hand side must always be greater than the right hand side. This is due to the tower increasing in height, and therefore rapidly outpacing the growth of the right hand side which is limited to 2^{j+3}. Plugging in $j = 1$, we have: $127 \geq 16$, which is exactly what we want to see.

The largest value j can take is $j = d = lg(lg(k))$. Thus we need to solve $k \geq 2^{lg(lg(k))+3}$ for k. We have:

$$k \geq 2^{lg(lg(k))+3}$$
$$k \geq 8lg(k)$$

Solving analytically, we know a function in $O(k)$ grows asymptotically faster than a function in $O(lg(k))$ and we already have our value $M = 2$. So all that needs to be found is a value k_0 such that $k \geq 8lg(k), k \geq k_0$. Choose $k_0 = 64$ and substituting into the equation: $64 \geq 8lg(64)$, which ends with the true result: $64 \geq 48$.

We have now found $M = 2$ and $k_0 = 64$ such that $f(2^k) = Mg(k), k \geq k_0$, which means $f(2^k) = O(g(2^k))$ and finally $T(n) = O(lg(n))$ since $T(n) = O(f(n))$ and $f(n) = f(2^k)$.

This completes the proof that the process of column reduction is bounded by the time complexity $O(log_2(n))$.

2.4 Putting it All Together

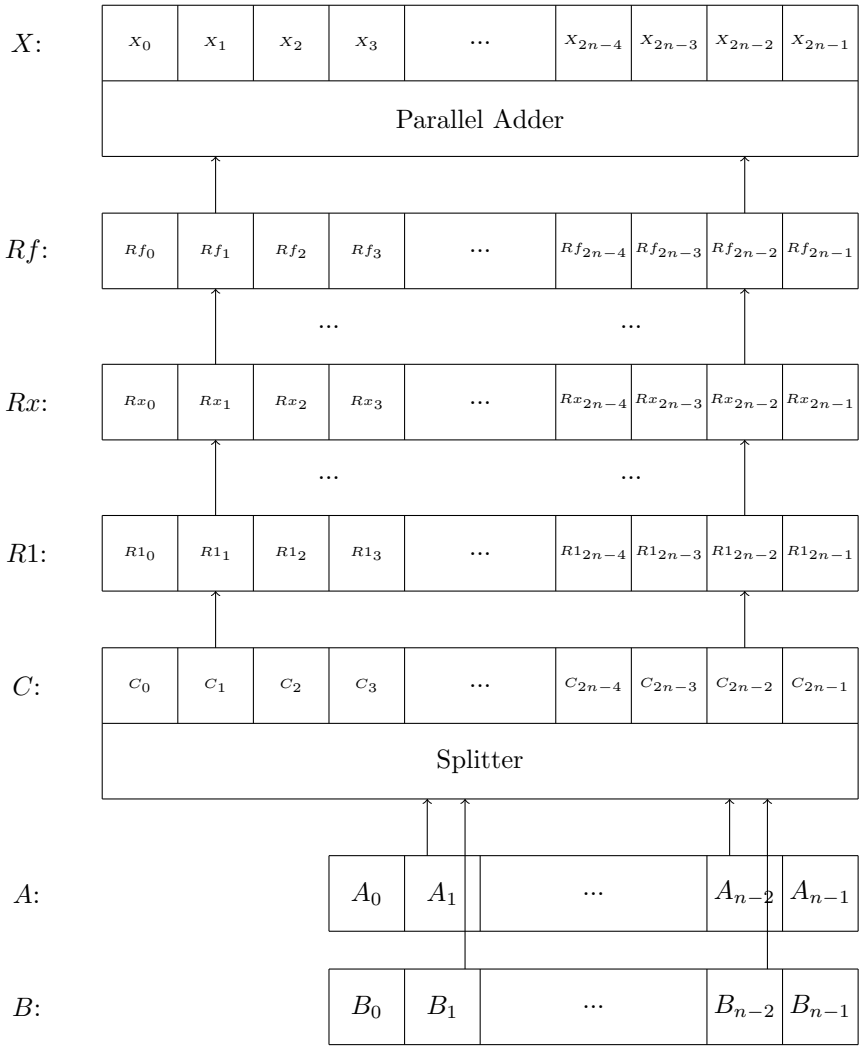
Having described the construction and use of all preceeding components of the multiplication circuit. We present an abstract overview of the inter-relation of these components. Firstly, we had addressed the splitting of A and B into a structure required to form multiplication rows and at the end of this splitting processes, the use of AND gates form the result of multiplication between all A_i and B_i. We may refer to this as a splitter, which we have shown performs its task in $O(log(n))$ time.

Next we had addressed the construction of the carry shower circuit, designed to perform the summation of 1's in columns corresponding to each place value of the multiplication result. The carry shower circuits are used iteratively in reduction layers to further condense the column summations until columns are only of height 2. Once again, we have shown before that this task occurs in $O(log(n))$ time. Only one final stage remains.

The last component is the addition of two $2n$-bit numbers, which can be performed using the parallel adder from [4] and also takes $O(log(n))$ time. We may refer to this component as a parallel adder. Now, having introduced and summarised the functioning of all related components, we diagramatically express an abstract representation of the completed circuit, which as a result of the $O(log(n))$ time complexity for all individual components, takes $O(log(n))$ time as a whole (given that regardless of the input size n, we have $O(log(n)) + O(log(n)) + O(log(n)) \rightarrow O(log(n))$).

The diagram can be explained as such: A and B are fed into the splitter producing multiplication rows partitioned into columns C_i. These are then fed into the first stage of column reduction $R1$ via carry showers and then fed into subsequent column reduction stages ($R1$ to Rx to Rf), where Rx refers to an intermediate stage and Rf the final reduction stage. Lastly, the result of the final reduction is fed into the parallel adder which produces output X.

Fig 2.7: Multiplication Circuit.



This completes Section 2 on the construction of an $O(log(n))$ n-bit multiplier of two numbers.

3 $O(log(n))$ Addition of n n-bit Numbers

This section identifies as a corollary to the problem of multiplication, how to perform the addition of n n-bit numbers together using largely the same process as before. This section will remain shorter, as most of the relevant content has been introduced and described before.

3.1 Similarities to Multiplication

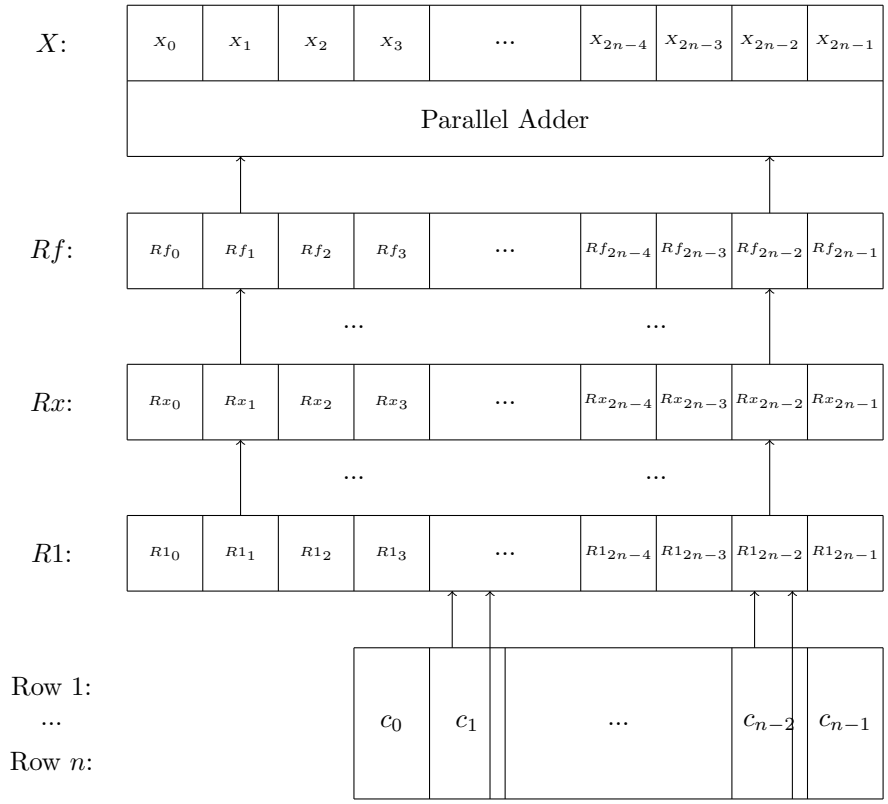
The addition of n n-bit numbers together is similar to the problem of multiplication, in the sense that instead of forming multiplication rows (the columns of which are summed up and repeatedly reduced), we now have addition rows consisting of n numbers that need to be added together. Previously, we had utilised multiplication rows of length $2n$ and of maximal height n in the middle, however the columns could all be considered the same height when summing redundant 0's in the columns. In our case here, we also have n rows of length n, but all columns are exactly height n.

In order to re-use the same circuitry as the multiplication problem (minus the splitting circuit), we can allow the addition rows to span a length $2n$ containing redundant 0's. We may also use the same parallel adder at the end of column reduction to sum the remaining rows of length $2n$. Note that when adding together n n-bit numbers, we expect the result to be at most of size $n + \lfloor log_2(n) \rfloor$, which neatly fits into the $2n$ sized final addition. Given that we are using the same processing elements as described in multiplication, the time complexity of n n-bit addition is $O(log(n))$.

3.2 Putting it All Together

The following is an $O(log(n))$ abstract representation of the entire circuit performing n n-bit addition, making use of the repeated column reduction using carry shower circuits on the n addition rows and the final parallel adder. In the diagram, c refers to "column".

Fig 3.1: n n-bit Addition Circuit.



This completes Section 3 on the construction of an $O(log(n))$ n-bit adder of n numbers.

4 References

[1] Bernstein, Daniel, "Multidigit Multiplication For Mathematicians," 2001.

[2] D. Harvey and J. van der Hoeven, "Integer multiplication in time $O(n\log n)$," Annals of Mathematics, vol. 193, no. 2, Mar. 2021, doi: 10.4007/annals.2021.193.2.4.

[3] C. S. Wallace, "A Suggestion for a Fast Multiplier," IEEE Transactions on Electronic Computers, vol. EC-13, no. 1, pp. 14–17, Feb. 1964, doi: 10.1109/pgec.1964.263830.

[4] Zupan, Matthew, "Parallel Addition in $O(log_2(n))$ Time," 27, Nov. 2022.

[5] C. C. Foster and F. D. Stockton, "Counting Responders in an Associative Memory," IEEE Transactions on Computers, vol. C–20, no. 12, pp. 1580–1583, Dec. 1971, doi: 10.1109/T-C.1971.223175.