Parallel Addition and Subtraction in $O(log_2(n))$ Time

Matthew Zupan

27/11/2022

Contents

1 Introduction **3**

2 Conceptual Process of Addition in $O(log^2(n))$ Time **5**
- 2.1 The Main Elements of Computation 5
- 2.2 Propagating Information Efficiently 6
- 2.3 Rules of Carry Propagation 9
- 2.4 Propagating Labels 12
- 2.5 Putting it All Together 13

3 Logical Process of Addition in $O(log(n))$ Time **16**
- 3.1 Labels and Transformations as Opcodes 16
- 3.2 Converting A_i and B_i into Opcodes 17
- 3.3 Converting Opcodes into Carry and Output 19
- 3.4 Carry Propagation 20
- 3.5 Putting it All Together 25

4 The Logical Process of Subtraction in $O(log(n))$ Time **27**

5 Results and Discussion **29**
- 5.1 Method 29
- 5.2 Results 31
- 5.3 Discussion 32

6 Supplementary Materials **34**
- 6.1 Sequential Simulation Code 34
- 6.2 $log^2(n)$ Simulation Code 36
- 6.3 $log(n)$ Simulation Code 38

7 References **44**

1 Introduction

In order to effectively structure the following paper, we first outline the problem motivation, purpose, observations, working assumptions and limitations.

When one considers performing addition of two numbers, typically the first thing that comes to mind is the method taught in school, that is: Start from the right-most place value and add two corresponding digits at the same index from right to left, making sure to add the result to the carry at each step. This process is naturally sequential due to the carry from the previous step influencing the result of the current step and as a result takes $O(n)$ time to compute.

For the purpose of this paper, we can treat the two numbers to be added as having the same number of digits n. If one of the numbers happened to have less than n digits before this treatment, consider padding the shorter length number on the left with 0's, such that the two lengths are the same. For example:
Number 1 has 10 digits and number 2 has 5 digits, so here $n = 10$ and number 2 is padded with five 0's.

Number 1: 4635728102
Number 2: 0000023784

The question that arises is whether addition can be performed faster than $O(n)$ given a parallel addition scheme. The answer is yes, as this paper and others before it and after will continue to address. While not every computational process requires faster addition, there are certain tasks that benefit from this. This paper is purposed towards addition of large numbers due to the fact that while $O(log(n))$ time complexity is achieved, there is a constant factor of overhead that negates the benefit or is worse off from this method if the numbers are small.

When referring to $log(\cdot)$, we are using base 2, $log_2(\cdot)$. Indexing will begin from left to right and starts at 0. We will also be working with numbers in binary and observe that the addition of two n-bit numbers may result in an $n+1$-bit number if the 0^{th} indexed bit of both numbers to be added are both of value 1. For example:

Index:		0	1	2	3	4
Number1:		1	0	1	1	0
Number2:		1	1	0	0	0
Result:	1	0	1	1	1	0

For this reason, we turn these two n-bit numbers into $n+1$-bit numbers by padding once again, but with a single 0, the indexing now starting from this newly added digit. This allows us to safely perform the addition without memory overflow, and for simplicity, we will refer to an n-bit number as n = the length of the number with the extra 0 padding. Then the addition of two n-bit numbers will always result in another n-bit number. The correct implementation is shown below:

Index:	0	1	2	3	4	5
Number1:	0	1	0	1	1	0
Number2:	0	1	1	0	0	0
Result:	1	0	1	1	1	0

The paper will first outline conceptually how parallel addition may occur in $(log(n))^2$ time, treating numbers 1 and 2 as a series of digits stored in a register that is shared memory and operations on each cell of memory, being performed by separate threads or processes with their own local memory. The $log(n)$ version presented afterwards will be an improvement on the first with some modifications and a circuit-like construction. It is important to note that the author does not claim this method to be efficient, many future improvements could be made in both space and time complexity. The author also acknowledges a lack of knowledge of electrical circuits, therefore any such designs may not be directly implementable and are to be considered as a logical representation of how the parallel addition should work.

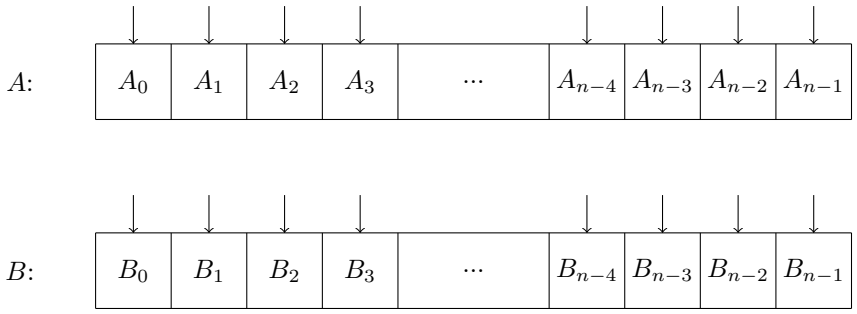
This introduction has extensively focused on addition. However, subtraction in binary will be achieved using almost an identical process to the method of addition. All that is required is taking the complement of one number A, adding this complement to B and taking the complement of the result. The complement is defined as flipping the value of all bits, say from 0 to 1 or 1 to 0.

2 Conceptual Process of Addition in $O(log^2(n))$ Time

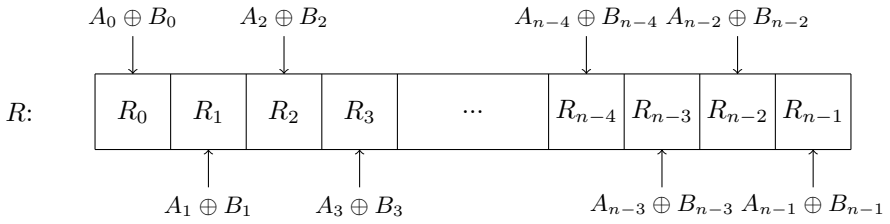
For simplicity, we will restrict the size of numbers to add together to values of $n = 2^k$, where $k \in \mathbb{Z}^+$. Both numbers will be stored in their own separate registers, consisting of a row of n cells in memory. Three additional registers of the same size will be used, the first being the result of addition on numbers A and B via XOR called the result register, the second contains the information of the carry bits called the carry register, and the third will contain the result of XOR between the result and carry registers and is called the output register. This contains the result of the complete computation.

2.1 The Main Elements of Computation

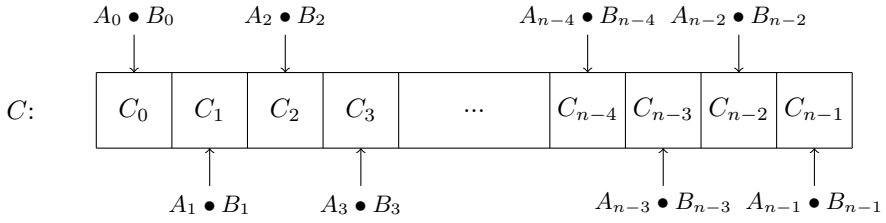
We begin by initialising the two registers of memory for numbers A and B, with each cell in memory independently loading in its corresponding bit:



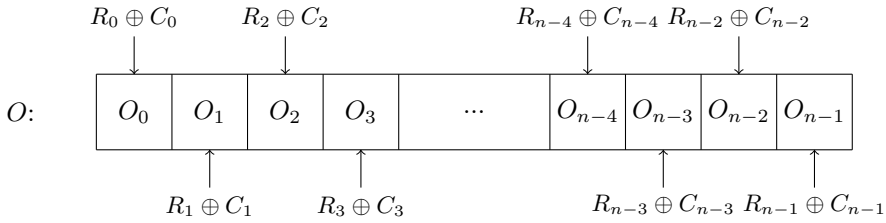
By performing the operation XOR between A and B, with symbol $\oplus$. We are concerning ourselves only with the result of addition modulo 2 seperately for each index i, where $0 \leq i \leq n$. Storing the result in the result register R, we have:



Additionally, by performing some integer multiple of $log(n)$ operations, $A_0 \bullet B_0 \bullet ... \bullet A_{n-1} \bullet B_{n-1}$ for each index i of A and B we will separately populate the carry register C, where the carries in this register represent the exact same carries as if the addition had been computed sequentially:



Finally, once the result and carry registers both contain the correct and required values, the XOR operation is applied again and the result stored in the output register O. The XOR between result and carry is equivalent to the process in sequential addition, but performed on each cell in parallel:



2.2 Propagating Information Efficiently

So far we have assumed that the carry register will contain the same carry data as a sequential addition and in $log(n)$ time. To motivate a path toward a correct solution, consider how information in the form of a bit-value 1 could be propagated from index $n-1$ of a register to all other indices of the same register. That is, all cells in memory (initialised to 0) will eventually contain 1.

The example process is as follows:

Begin by defining a hypothetical register H and its auxiliary register H', both of size n and all initialised to 0 with the exception of $H_{n-1} = 1$. Then, in $m = log(n)$ subsequent steps for the outer loop of the following algorithm, we have transmitted the data to all cells:

BEGIN
For each cell i in H, simultaneously do:
    For m from 0 to $log(n-1)$, do:
        $j \leftarrow 2^m$;
        If $i - j \geq 0$:
            Transmit overriding data from H_i to H'_{i-j};

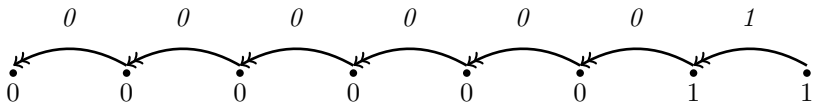
        $H_i \leftarrow H'_i$;
END

To explain the previous algorithm: If possible, each cell in register H will transmit its data into the auxiliary register H' by distance $d = 1$ position to the left, then by a constantly doubling distance $d = 2, 4, 8, 16, ..., \frac{n}{2}$. The data is transferred from the auxiliary register H' into H at each step as to avoid concurrent access to the same memory or memory overwrite. A diagrammatic abstraction for $n = 8$ is as follows:

$m = 0$:

State before transmission:

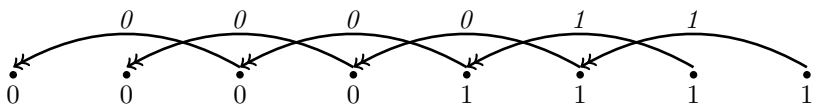
State after transmission:



$m = 1$:

State before transmission:

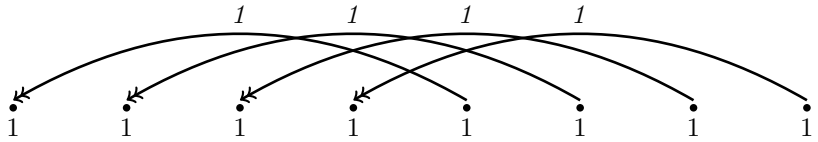
State after transmission:



$m = 2$:

State before transmission:

State after transmission:



The above example assumes that the information of the bit valued 1 overrides the bit valued 0. This is a reasonable assumption as we are seeking to propagate some properties across with greater priority. 0 represents "do nothing" in this case, while 1 represents something of greater importance. This is analogous to propagating a carry of 1 in an addition, although the process of doing so correctly will be more complicated than this.

Firstly, to prove correctness of the above, we do so inductively:

Base Case:
Each cell in memory with index in $0 \leq i \leq n-1$ contains its initialised bit-value $\in \{0, 1\}$.

Inductive Step:
Assume after $k = log(n-1) - 1$ doublings that the information from all cells i has propagated to cells$(k) = [i : i - 2^k]$.
Then at step $k+1$, we have:

$$\begin{aligned}(k+1) &= (log(n-1) - 1) + 1 \\ &= log(n-1)\end{aligned}$$

And for cells containing the information:

$$\text{cells}(k+1) = [i : i - 2^{k+1}]$$

Given that $\forall i(i - 1 - 2^{k+1} < i - 2^{k+1})$, we can choose the greatest value of i and show that the information reaches index 0. Substituting in $i = n - 1$ and $k + 1 = log(n-1)$, we have:

$$\begin{aligned}\text{cells}(k+1) &= [n-1 : (n-1) - 2^{log(n-1)}] \\ &= [n-1 : (n-1) - n - 1] \\ &= [n-1 : 0]\end{aligned}$$

Spanning from cells$[n-1]$ to cells[0] as required.

2.3 Rules of Carry Propagation

We begin with the observation that our two numbers A and B contain specific properties that allow us to identify where a given carry may originate from and where it may not propagate beyond. An obvious barrier is the index $i = 0$ given it is the smallest index in memory allocated to the result of computation. However, we can further observe that if at some i, numbers A and B both contain a 0, then if a carry were to reach this index: The result of addition with the carry is $0 + 1 = 1$ and because $1(mod2) = 1$, there is no carry afterwards (alternatively, a do nothing = 0 carry is propagated afterwards). The index $i = 0$ for numbers A and B were defined to satisfy this property.

Therefore if A and B at some i are both 0, we say this index i is a barrier, the label **Bar** will be given to a barrier.

Similarly, we can create a label **Prop** for propagators of carries, which arise in the circumstance that A and B at index i are both the value 1. A carry always travels left and propagates from a propagator. This is because the result of $1 + 1 = 2$ and $2(mod2) = 0$, requiring we have a carry of 1.

Lastly, of these fundamental labels, we define **Amb** as the ambiguous type label for when A and B at index i is $(0, 1)$ or $(1, 0)$. Because the result of addition is $1 + 0 = 1$ or $0 + 1 = 1$, and $1(mod2) = 1$, the ambiguous type handles the case when we do not know if an index i will be propagating a carry. This depends on the information from propagators and barriers at indices $j > i$ that may or may not reach i at a later time.

Prop, **Amb** and **Bar** itself holds no carry, but is transmitting one. It is not sufficient only to track propagated carry information, we also require that at the end of the process, each index i knows what to do with this information. For example, an index i for which both A and B are valued 1 will always propagate a carry, but may itself recieve a carry from some index $j > i$. Conversely it may not recieve a carry. So we must also somehow encode this information into the label and investigate the hierarchy of precedence between labels and their transformation when transmitting information.

Having the need for additional labels, we introduce the following and then justify their relevance:

PropP: A propagator that has received a signal from a propagator type label. This is a propagator that holds a carry of 1.
PropB: A propagator that has received a signal from a barrier type label. This is a propagator that holds a carry of 0.
BarP: A barrier that has received a signal from a propagator type label. This is a barrier that holds a carry of 1.
BarB: A barrier that has received a signal from a barrier type label. This is a barrier that holds a carry of 0.

We also require making the distinction between fixed and unfixed labels. **Prop** **Amb** and **Bar** are unfixed, meaning that if they receive a signal from another label, they may be subject to transformation into another label. On the other hand, **PropP**, **PropB**, **BarP** and **BarB** are fixed, meaning that regardless of what signal they receive, the label will remain unchanged. We will identify how to treat unfixed labels.

Inspecting the case of **Prop**:

- If **Prop** receives a signal from **Prop**, **PropP** or **PropB**, this means we have received a carry signal and this carry must be stored. However, since we are still propagating carries, **Prop** will then transform into **PropP** such that no future signal can influence its' identity.
- If **Prop** receives a signal from **Amb**, this means we have received no information on the carry and **Prop** shall remain unchanged.
- If **Prop** receives a signal from **Bar**, **BarP** or **BarB**, this means we have received a barrier signal and know that no carry is to be computed at this point. However, since we are still propagating carries, **Prop** will then transform into **PropB** such that no future signal can influence its' identity.

Inspecting the case of **Amb**:

- If **Amb** receives a signal from **Prop**, **PropP** or **PropB**, this means we have received a carry signal and know a carry is being propagated across. **Amb** will then transform into a fixed propagator **PropP** such that no future signal can influence its' identity.
- If **Amb** receives a signal from **Amb**, this means we do not yet know if a carry is to be propagated along or not and so **Amb** will remain unchanged.
- if **Amb** receives a signal from **Bar**, **BarP** or **BarB**, this means we have received a barrier signal and know a carry is not to be propagated across. **Amb** will then transform into a fixed barrier **BarB** such that no future signal can influence its' identity.

Inspecting the case of **Bar**:

- If **Bar** receives a signal from **Prop**, **PropP** or **PropB**, this means we have received a propagator signal and know that a carry is to be computed at this point. Since we are still propagating barriers, **Bar** will transform into **BarP** such that no future signal can influence its' identity.
- If **Bar** receives a signal from **Amb**, this means we have received no information on the carry and **Bar** shall remain unchanged.
- If **Bar** receives a signal from **Bar**, this means we have received a barrier signal and know that a carry is not to be computed at this point. Since we are still propagating barriers, **Bar** will transform into **BarB** such that no future signal can influence its' identity.

Now, we can form a table that concisely specifies how labels are to be transformed, read as row receives column transforms into:

Table 2.3.1: Table of Label Transformations

	Prop	PropP	PropB	Amb	Bar	BarP	BarB
Prop	PropP	PropP	PropP	Prop	PropB	PropB	PropB
PropP	PropP	PropP	PropP	PropP	PropP	PropP	PropP
PropB	PropB	PropB	PropB	PropB	PropB	PropB	PropB
Amb	PropP	PropP	PropP	Amb	BarB	BarB	BarB
Bar	BarP	BarP	BarP	Bar	BarB	BarB	BarB
BarP	BarP	BarP	BarP	BarP	BarP	BarP	BarP
BarB	BarB	BarB	BarB	BarB	BarB	BarB	BarB

A second table will also store the carry that is to be applied at the end of the data transmitting process, given a label. The table is read as label outputs, carry of value:

Table 2.3.1: Table of Label Transformations

Label	Carry
Prop	0
PropP	1
PropB	0
Amb	0
Bar	0
BarP	1
BarB	0

2.4 Propagating Labels

With the assumption that each cell in memory H_i is initialised with its own label, is allocated its own thread, and each thread is capable of performing its own computations. We revisit the algorithm from section 2.2, now with the transformation rules of table 2.3.1.

Algorithm 2.4.1:

BEGIN
For each cell i in H, simultaneously do:
    For m from 0 to $log(n-1)$, do:
        $j \leftarrow 2^m$;
        If $i - j \geq 0$:
            $H'_{i-j} \leftarrow H_i$;

        If $H_i =$ **Prop**:
            If $H'_i =$ **Prop** or **PropP** or **PropB**:
                $H_i \leftarrow$ **PropP**;
            If $H'_i =$ **Bar** or **BarP** or **BarB**:
                $H_i \leftarrow$ **PropB**;

        If $H_i =$ **Amb**:
            If $H'_i =$ **Prop** or **PropP** or **PropB**:
                $H_i \leftarrow$ **PropP**;
            If $H'_i =$ **Bar** or **BarP** or **BarB**:
                $H_i \leftarrow$ **BarB**;

        If $H_i =$ **Bar**:
            If $H'_i =$ **Prop** or **PropP** or **PropB**:
                $H_i \leftarrow$ **BarP**;
            If $H'_i =$ **Bar** or **BarP** or **BarB**:
                $H_i \leftarrow$ **BarB**;
END

Admittedly, this is a rather crude algorithm, having made use of multiple If statements and repeated work. But it is useful as a simple overview of the logic and to prevent early completion by a given thread. Had we used If Else statements, in some cases, threads may perform more or less work than the other threads, finish at different times and therefore cause race condition errors because they aren't synchronised. Note that we do not bother checking if H_i is **PropP**, **PropB**, **BarP** or **BarB** because it will remain unchanged and assignment can be ignored. We do not check if H'_i is **Amb** for the reason that it does not change H_i.
The algorithm takes $O((log(n)^2)$ time to run due to the need to compute ad-

dresses in memory. Storing a number of n bits requires a $log(n)$ size binary encoding of each cells' address. Computing $j \leftarrow 2^m$ only requires a simple left shift and is therefore $log(n)$ work for each iteration of the loop. Calculating $i - j$ similarly requires $log(n)$ work for each pass. The If statements and label transformations requires a constant amount of work and so the algorithm is bounded by $O(log(n) * log(n)) = O((log(n))^2)$ overall.

2.5 Putting it All Together

Before all pieces of the puzzle can fall into place, we must consider converting values A_i and B_i into labels as well as converting labels at the end of carry propagation back into carry values.

Recall labels must initially be:
Prop when A_i and B_i = (1, 1),
Amb when A_i and B_i = (0, 1) or (1, 0),
Bar when A_i and B_i = (0, 0).

So our psuedocode to run on each thread for this conversion process is:

BEGIN
If $A_i = 1$ and $B_i = 1$:
$H_i \leftarrow$ **Prop**;
If ($A_i = 1$ and $B_i = 0$) or ($A_i = 0$ and $B_i = 1$):
$H_i \leftarrow$ **Amb**;
If $A_i = 0$ and $B_i = 0$:
$H_i \leftarrow$ **Bar**;
END

Converting labels into a carry at the end of carry propagation, note that the only labels that hold a carry of 1 are **PropP** and **BarP**, and the others 0, so we have:

BEGIN
If $H_i =$ **PropP** or **BarP**:
$C_i \leftarrow 1$;
If $H_i =$ **Prop** or **PropB** or **Amb** or **Bar** or **BarB**:
$C_i \leftarrow 0$;
END

Now we are ready to outline the entire process and then provide psuedocode composing all processes together.

For all i simultaneously:
- Begin by taking the result of addition via $A_i \oplus B_i$ and store in R_i.
- Convert values A_i and B_i into labels.
- Run algorithm 2.4.1 to propagate carry information.
- Convert labels into carry values.
- Take the addition of result and carry via $R_i \oplus C_i$ and store in O_i.

All together, the composed algorithm is as follows:

Algorithm 2.5.1: Addition of A and B

For all i simultaneously:
BEGIN
// Take the result of addition via $A_i \oplus B_i$ and store in R_i.
$R_i \leftarrow A_i \oplus B_i$;

// Convert values A_i and B_i into labels.
If $A_i = 1$ and $B_i = 1$:
$H_i \leftarrow$ **Prop**;
If ($A_i = 1$ and $B_i = 0$) or ($A_i = 0$ and $B_i = 1$):
$H_i \leftarrow$ **Amb**;
If $A_i = 0$ and $B_i = 0$:
$H_i \leftarrow$ **Bar**;

// Run algorithm 2.4.1 to propagate carry information.
For m from 0 to $log(n-1)$, do:
$j \leftarrow 2^m$;
If $i - j \geq 0$:
$H'_{i-j} \leftarrow H_i$;

If $H_i =$ **Prop**:
If $H'_i =$ **Prop** or **PropP** or **PropB**:
$H_i \leftarrow$ **PropP**;
If $H'_i =$ **Bar** or **BarP** or **BarB**:
$H_i \leftarrow$ **PropB**;

If $H_i =$ **Amb**:
If $H'_i =$ **Prop** or **PropP** or **PropB**:
$H_i \leftarrow$ **PropP**;
If $H'_i =$ **Bar** or **BarP** or **BarB**:
$H_i \leftarrow$ **BarB**;

If H_i = **Bar**:
If H'_i = **Prop** or **PropP** or **PropB**:
$H_i \leftarrow$ **BarP**;
If H'_i = **Bar** or **BarP** or **BarB**:
$H_i \leftarrow$ **BarB**;

// Convert labels into carry values.
If H_i = **PropP** or **BarP**:
$C_i \leftarrow 1$;
If H_i = **Prop** or **PropB** or **Amb** or **Bar** or **BarB**:
$C_i \leftarrow 0$;

// Take the addition of result and carry via $R_i \oplus C_i$ and store in O_i.
$O_i \leftarrow R_i \oplus C_i$;
END

This completes process for the Conceptual Process of Addition in $O(log^2(n))$ Time.

3 Logical Process of Addition in $O(log(n))$ Time

The logical process of addition takes inspiration from the previous section. However, instead of using addressable registers and labels, we now use repeating processing elements in the form of nodes and gates in a circuit, each storing information and passing along single binary values. The proposed method will achieve addition in $O(log(n))$ time but comes at a cost of $O(nlog(n))$ space. Logically, we still require the result R_i as the XOR between each A_i and B_i as well as the output O_i to be the XOR between R_i and C_i, but further processing is required and will be addressed in the following subsections.

3.1 Labels and Transformations as Opcodes

Previously we made use of labels and transformations between labels during the carry propagating process. Now, to convert this into a suitable form for circuit-like processing we will assign an operation code to each label, consisting of three ordered binary values. Labels of type **Prop** will be given their first value as 1, other labels as 0. Fixed type labels will be given their second value 1 and non-fixed labels 0. Lastly, **Prop**, **Bar** and suffix **B** labels will be given last value 1 and the others 0. This results in the following opcodes:

Table 3.1.1: Table of Opcodes

Label	Opcode
Prop	1 0 1
PropP	1 1 0
PropB	1 1 1
Amb	0 0 0
Bar	0 0 1
BarP	0 1 0
BarB	0 1 1

Each opcode value represents an individual signal in parallel to the other opcode values, the combination of which will be reduced at the end of carry propagation to a single binary 1 or 0 as C_i. Therefore we also require the values of A_i and B_i at the beginning to be converted into their respective opcode. Before moving on to the conversions, we will first inspect our current opcode requirements. As per table 2.3.1: Table of Label Transformations, we required labels to transform in the carry propagation process. The same will apply in this section, requiring opcodes to undergo transformation upon receiving a signal.

Table 3.1.2: Table of Opcode Transformations

	1 0 1	1 1 0	1 1 1	0 0 0	0 0 1	0 1 0	0 1 1
1 0 1	1 1 0	1 1 0	1 1 0	1 0 1	1 1 1	1 1 1	1 1 1
1 1 0	1 1 0	1 1 0	1 1 0	1 1 0	1 1 0	1 1 0	1 1 0
1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1
0 0 0	1 1 0	1 1 0	1 1 0	0 0 0	0 1 1	0 1 1	0 1 1
0 0 1	0 1 0	0 1 0	0 1 0	0 0 1	0 1 1	0 1 1	0 1 1
0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0
0 1 1	0 1 1	0 1 1	0 1 1	0 1 1	0 1 1	0 1 1	0 1 1

3.2 Converting A_i and B_i into Opcodes

Given the two input numbers A and B, we require each digit to be converted into its respective opcode based on the Section 2.5 conversion:

Prop = 101 when A_i and B_i = (1, 1),
Amb = 000 when A_i and B_i = (0, 1) or (1, 0),
Bar = 001 when A_i and B_i = (0, 0).

Observe that:

- Digit 1 of the opcode is 1 only for **Prop**, and 0 for the others, so
  digit 1 $\leftarrow A_i \wedge B_i$.

- Digit 2 of **Prop**, **Amb** and **Bar** is always 0, so
  digit 2 $\leftarrow A_i \wedge \neg A_i$.

- Digit 3 of **Prop** and **Bar** is 1, so
  digit 3 $\leftarrow A_i \neg \oplus B_i$.

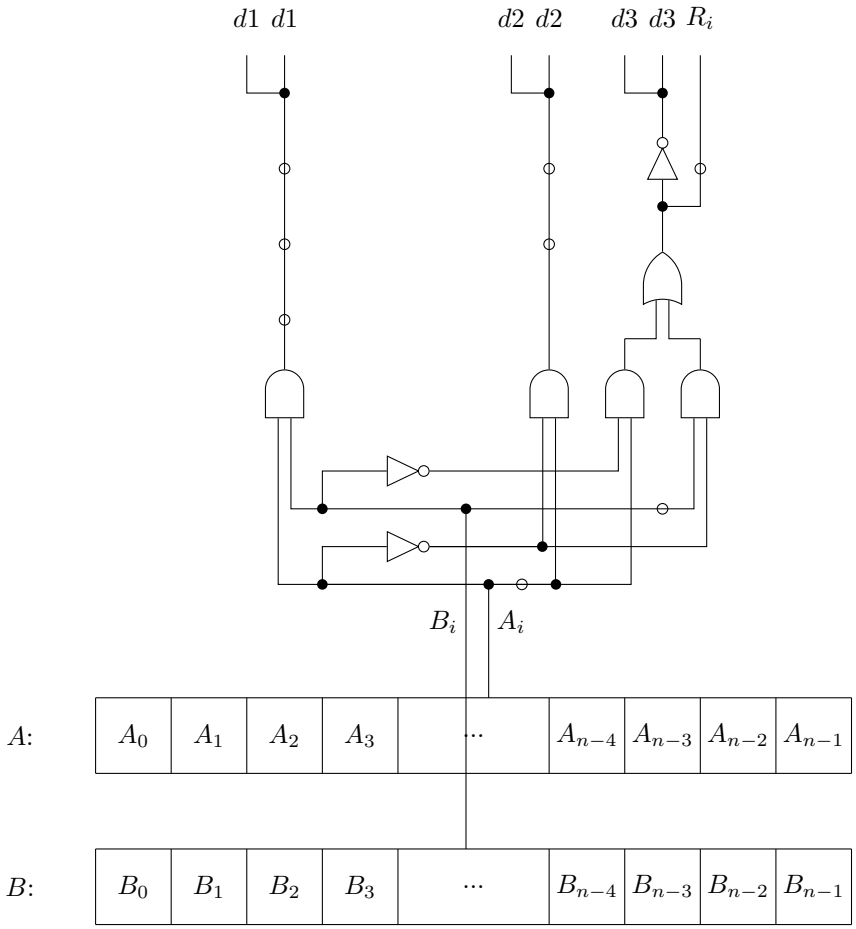
The truth tables to illustrate the above are:

Table 3.2.1: Input to Opcode Conversion Truth Tables

A_i	B_i	digit 1: $A_i \wedge B_i$	digit 2: $A_i \wedge \neg A_i$	digit 3: $\neg A_i \oplus B_i$
1	1	1	0	1
1	0	0	0	0
0	1	0	0	0
0	0	0	0	1

What remains to be done is to design a circuit which executes this scheme. In order to logically synchronise each step in discrete time intervals we will make use of a "repeater gate" $\circ$, which simply delays propagation of a signal by the

same unit of time it would take any other gate (NOT, AND, OR) to complete its operation. We will also decompose gates such as XOR into their respective components, where $\oplus \Leftrightarrow (p \wedge \neg q) \vee (\neg p \wedge q)$. For each A_i and B_i to be converted into an opcode, we will make use of an array of the following mini circuit:



The diagram also contains an output R_i. This is due to the fact that R_i is calculated as $A_i \oplus B_i$. Since we make use of this already in our conversion for digit 3, we use a branch to piggy back from the result of this computation. The result R_i will then be propagated along the line with the opcode digits 1, 2 and 3 ($d1$, $d2$, $d3$) when computing carries. R_i will remain independent from the opcode until the end when we do $R_i \oplus C_i$.

In the diagram, $d1$, $d2$ and $d3$ are split into two as they will later be used to transmit opcodes directly above to another set of processing elements but at different locations. The left $d1$ will be relabelled $L1$, the right $d1$ as $R1$, the

left $d2$ as $L2$ and the right $d2$ as $R2$. Lastly, the left $d3$ as $L3$, the right $d3$ as $R3$ and R_i remains the same. Next, we consider converting opcodes into carries assuming carry propagation has been completed.

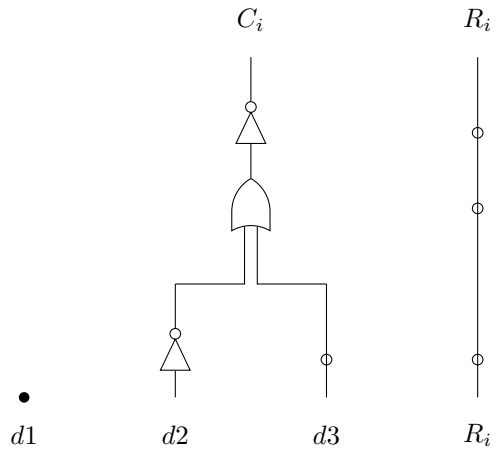
3.3 Converting Opcodes into Carry and Output

The last cycle of carry propagation does not split opcodes into two parallel streams "left" and "right" and so we are left with $d1$, $d2$ and $d3$ which are required to be converted into a single bit value C_i which is either 0 or 1. Recall the only two labels that hold a carry of 1 are **PropP** and **BarP** with opcodes 1 1 0 and 0 1 0 respectively. In fact, they are the only two opcodes that have digits 2 and 3 as 1 0. This means digit 1 can be ignored and converting these labels into a carry of 1, and all other labels into a carry of 0 only requires use of a $\neg \Rightarrow$ operation. This operation has truth table:

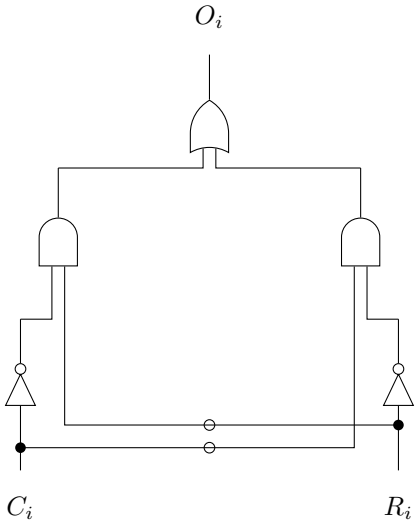
Table 3.3.1: Opcode to Carry Conversion Truth Table

$d2$	$d3$	$\neg(d2 \Rightarrow d3)$
1	1	0
1	0	1
0	1	0
0	0	0

With $\neg(d2 \Rightarrow d3)$ simply being the negation of $\neg d2 \vee d3$, we can construct a carry conversion circuit:



Converting to output O_i simply requires the operation $C_i \oplus R_i$, illustrated below:



Now O_i stores the completed computation of $A_i + B_i$.

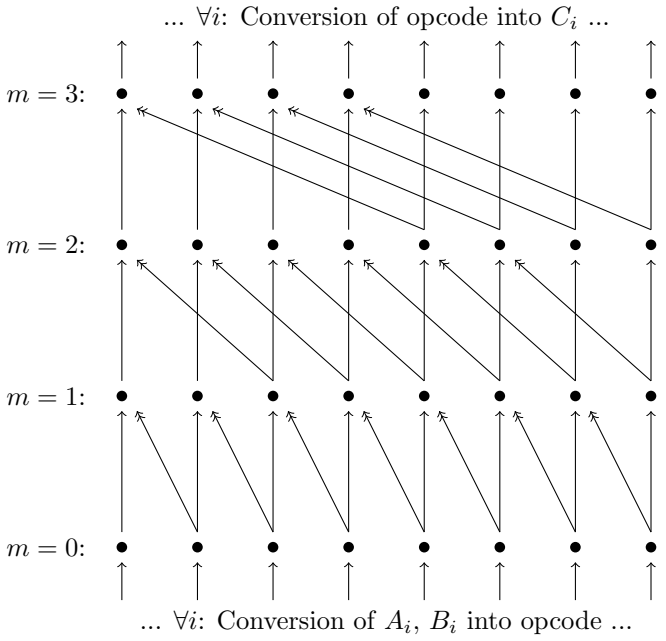
Having addressed the conversions from A_i and B_i to opcodes, conversions from opcodes into carry C_i, computation of R_i and the final output O_i for each i making use of arrays of the same repeated mini circuits, we turn our attention to the process of carry propagation itself, which will take considerably more effort than these previous steps.

3.4 Carry Propagation

Carry information is passed on similarly to the method outlined in Sections 2.2 and 2.4. However, instead of addressing locations in memory, we build a $log(n)$ number of rows that are all interconnected in a special way. The opcode output for each i in a given row m is propogated to the processing element at physical location i in the row $m+1$ above (comes from the "left"), but is also split and connected to the processing element at physical location $i-2^m$ (comes from the "right") provided $i - 2^m \geq 0$.

The 0^{th} row only contains the converted opcodes from A_i and B_i which are then passed on to row 1 where each location i takes the two split opcode inputs "left" and "right". Left has opcode digit inputs $L1$, $L2$ and $L3$. Similarly, right has $R1$, $R2$ and $R3$. The next diagram is an abstraction of this concept for size $n = 8$ with the opcodes treated as a single source.

Overview of the Propogation Setup:



In order to create the propogation mini circuits for each processing element i of rows $m = 1$ to $m = log(n)$, we make use of the Sum of Products method for circuit simplification via a Karnaugh Map tool for 6 variables [1]. We map each $L1$, $L2$, $L3$, $R1$, $R2$ and $R3$ to separate output target states $d1$, $d2$ and $d3$ as given by table 3.1.2: Table of Opcode Transformations, and then combine the three separate circuits into one. For each combination of inputs, we set the ouput target as 0, 1 or x (meaning ignore, because we do not make use of a particular input). The table of this is presented on the next page:

Table 3.4.1: Opcode Input to Output Targets

$L1,2,3$	$R1,2,3$	$d1$	$d2$	$d3$
0 0 0	0 0 0	0	0	0
0 0 0	0 0 1	0	1	1
0 0 0	0 1 0	0	1	1
0 0 0	0 1 1	0	1	1
0 0 0	1 0 0	x	x	x
0 0 0	1 0 1	1	1	0
0 0 0	1 1 0	1	1	1
0 0 0	1 1 1	1	1	1
0 0 1	0 0 0	0	0	1
0 0 1	0 0 1	0	1	1
0 0 1	0 1 0	0	1	1
0 0 1	0 1 1	0	1	1
0 0 1	1 0 0	x	x	x
0 0 1	1 0 1	0	1	0
0 0 1	1 1 0	0	1	0
0 0 1	1 1 1	0	1	0
0 1 0	0 0 0	0	1	0
0 1 0	0 0 1	0	1	0
0 1 0	0 1 0	0	1	0
0 1 0	0 1 1	0	1	0
0 1 0	1 0 0	x	x	x
0 1 0	1 0 1	0	1	0
0 1 0	1 1 0	0	1	0
0 1 0	1 1 1	0	1	0
0 1 1	0 0 0	0	1	1
0 1 1	0 0 1	0	1	1
0 1 1	0 1 0	0	1	1
0 1 1	0 1 1	0	1	1
0 1 1	1 0 0	x	x	x
0 1 1	1 0 1	0	1	1
0 1 1	1 1 0	0	1	1
0 1 1	1 1 1	0	1	1

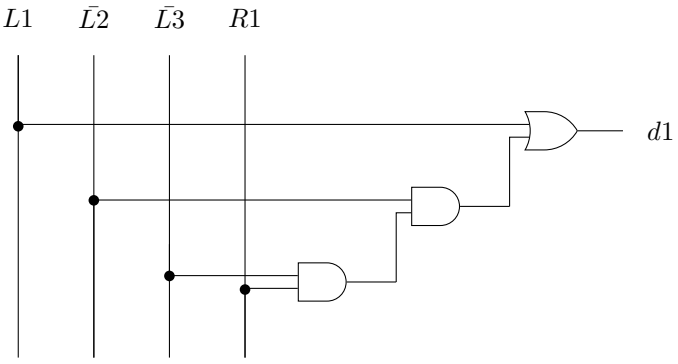
$L1,2,3$	$R1,2,3$	$d1$	$d2$	$d3$
1 0 0	0 0 0	x	x	x
1 0 0	0 0 1	x	x	x
1 0 0	0 1 0	x	x	x
1 0 0	0 1 1	x	x	x
1 0 0	1 0 0	x	x	x
1 0 0	1 0 1	x	x	x
1 0 0	1 1 0	x	x	x
1 0 0	1 1 1	x	x	x
1 0 1	0 0 0	1	0	1
1 0 1	0 0 1	1	1	1
1 0 1	0 1 0	1	1	1
1 0 1	0 1 1	1	1	1
1 0 1	1 0 0	x	x	x
1 0 1	1 0 1	1	1	0
1 0 1	1 1 0	1	1	0
1 0 1	1 1 1	1	1	0
1 1 0	0 0 0	1	1	0
1 1 0	0 0 1	1	1	0
1 1 0	0 1 0	1	1	0
1 1 0	0 1 1	1	1	0
1 1 0	1 0 0	x	x	x
1 1 0	1 0 1	1	1	0
1 1 0	1 1 0	1	1	0
1 1 0	1 1 1	1	1	0
1 1 1	0 0 0	1	1	1
1 1 1	0 0 1	1	1	1
1 1 1	0 1 0	1	1	1
1 1 1	0 1 1	1	1	1
1 1 1	1 0 0	x	x	x
1 1 1	1 0 1	1	1	1
1 1 1	1 1 0	1	1	1
1 1 1	1 1 1	1	1	1

The generated equations for each separate target output $d1$, $d2$ and $d3$ are:

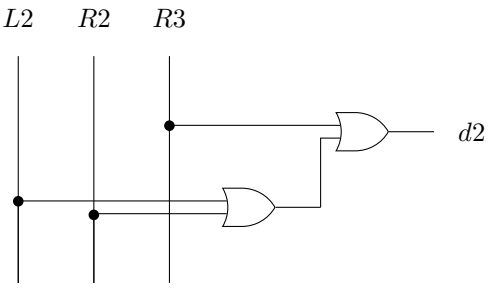
- $d1 = L1 + \bar{L2}.\bar{L3}.R1$
- $d2 = R3 + R2 + L2$
- $d3 = L3.\bar{R1} + L2.L3 + \bar{L2}.\bar{R1}.R3 + \bar{L2}.\bar{R1}.R2$

The respective (asynchronous) circuit diagrams for outputs $d1$, $d2$ and $d3$ are:

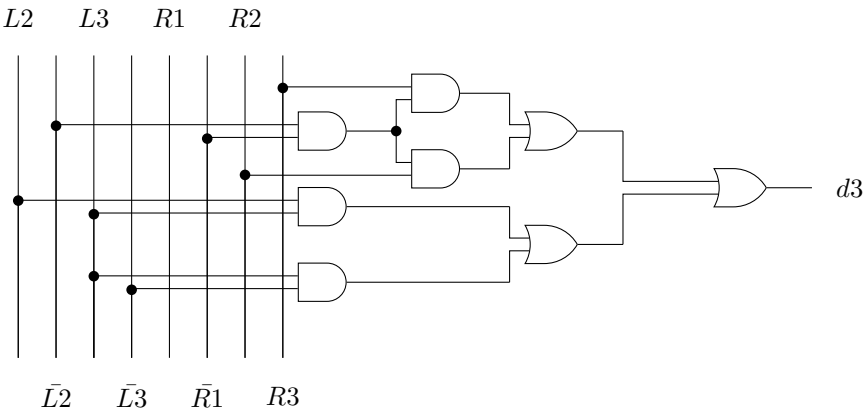
$d1$ Individual Output Circuit:



$d2$ Individual Output Circuit:

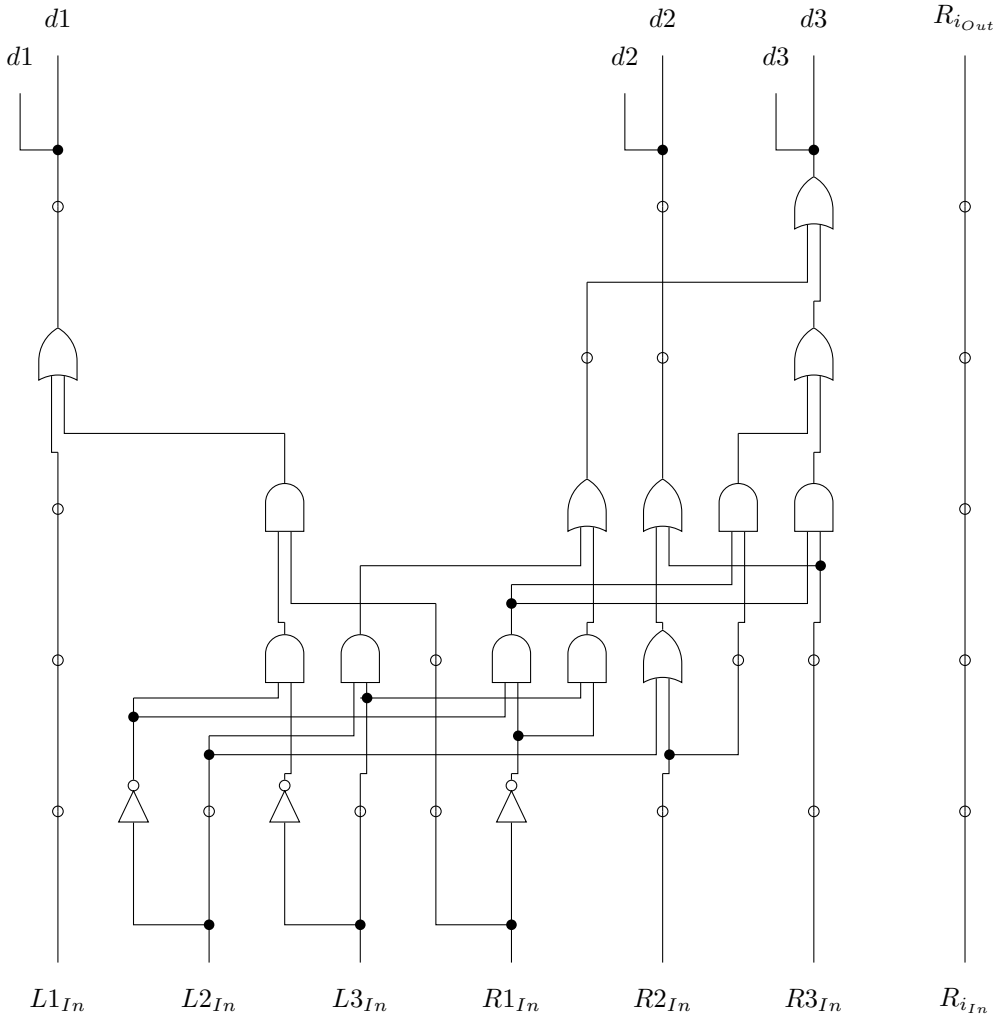


$d3$ Individual Output Circuit:



From the above individual circuits for $d1$, $d2$ and $d3$, we combine them in synchronous fashion to form a processing elements' mini circuit, while also repeating R_i alongside it.

Carry propagation circuit:



Note that this circuit for each i in the carry propagation will differ from the current diagram when $m = log(n)$ only in the output $d1$, $d2$ and $d3$ which are not split into two outputs at the end. This is because we require one opcode output for the conversion to carry bit C_i before taking the XOR between C_i and R_i into O_i.

3.5 Putting it All Together

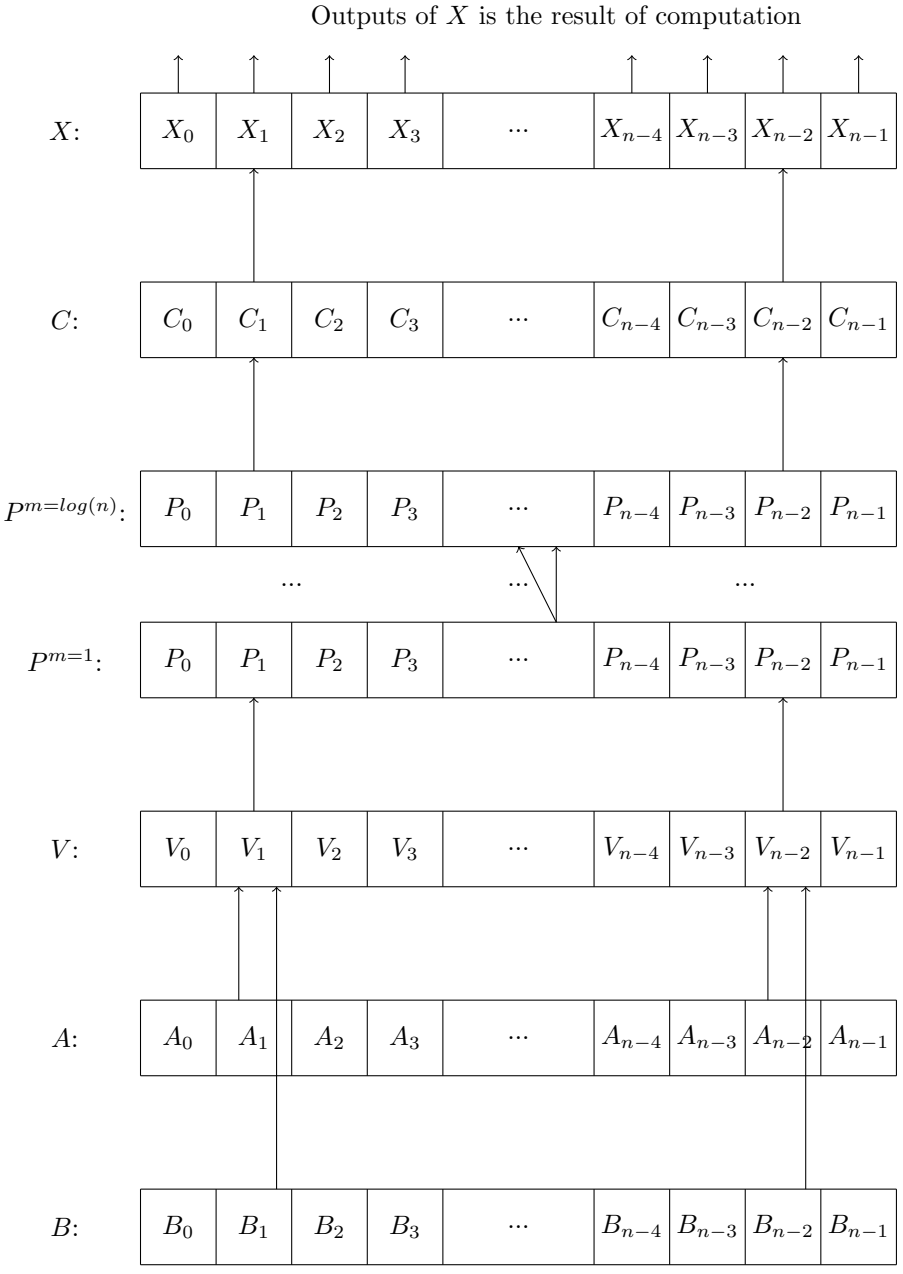
We currently have:
- Initial numbers A and B,
- A conversion to opcode and result circuit, for convenience called V,
- A carry propagation circuit, we will call P,
- A conversion from opcode and result to carry circuit, called C and lastly,
- An XOR carry and result circuit, called X.

Using the above, we outline the general process of computation given A and B of size n:

1. Create a row of circuits V for each i, then V_i will convert A_i and B_i into their respective opcodes and results.
2. Create $log(n) - 1$ rows of circuits P (split output variation), then each P_i in the first row $m = 1$ will propagate the carry and result from V_i to row $m = 2$. Each other row will propagate carry and result information from row $m - 1$ to $m + 1$.
3. The last row $m = log(n)$ will have each P_i propagate carry and result information using the single output variation.
4. Create a row of circuits C for each i which takes input from X_i and converts the opcode and result into carry and result.
5. Create a row of circuits X for each i where X_i takes input from C_i and computes the XOR of carry and result. The collective output of which is the completed computation of $A + B$.

The diagram of this scheme is presented on the following page, note that many up arrows have been omitted but it is implied that one exists for each i.

Schematic of Addition:



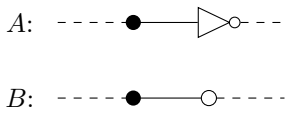
This completes the Logical Process of Addition in $O(log(n))$ Time.

4 The Logical Process of Subtraction in $O(log(n))$ Time

This section will remain brief given that the only difference between addition and subtraction in this scheme is the use of the complement of A before applying the addition operation, and a final complement of the result. Note that since we are taking the complement of A only, this will require the use of a NOT gate to flip the value of all bits in A and therefore incur unit time delay, for this reason, each element of B will need to be fed into a single delay gate such that the signals from A and B are synchronised.

We will a component called the initial complement circuit which addresses the previously outlined requirement, shown in the following diagram Fig 4.1:

Fig 4.1: Initial Complement.

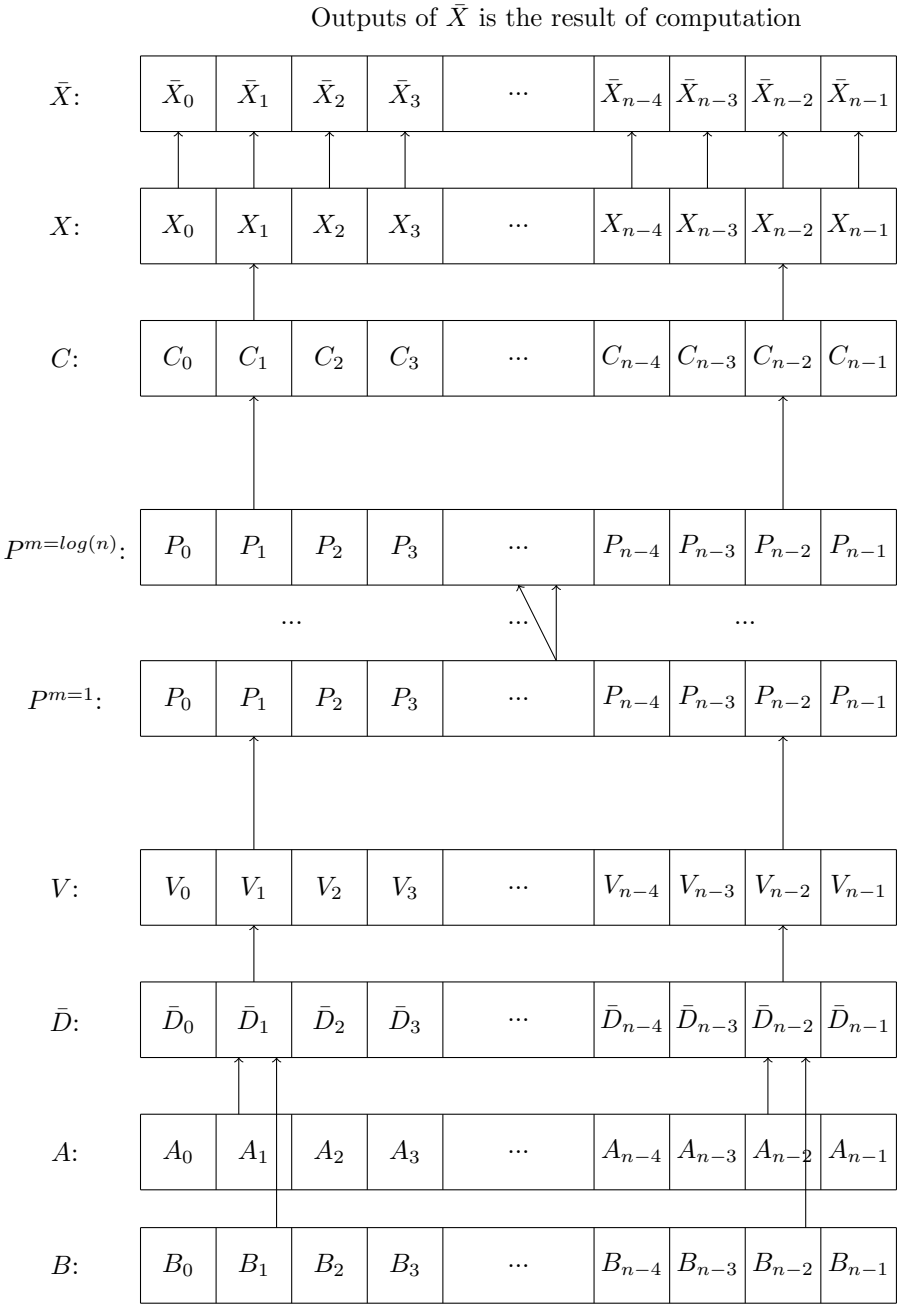


We also define the final complement, being the complement of the result at the end of the addition phase. This is simply a NOT gate for each element of the result X, shown in Fig 4.2:

Fig 4.2: Final Complement.

Finally, we show the entire process of subtraction in Fig 4.2. Taking note that $\bar{X}$ represents the final complement and $\bar{D}$ represents the initial complement.

Fig 4.3: Schematic of Subtraction:



This completes the Logical Process of Subtraction in $O(log(n))$ Time.

5 Results and Discussion

The author is aware of other papers such as that from Brent and Kung which remove the need for opcodes and as such are less expensive in the time needed to perform addition [2]. This method claims to be less efficient, but an $O(log(n))$ solution nonetheless. The simulations are not to be taken too seriously as they are done sequentially in Python3 and the times taken do not reflect actual hardware times. Moreover, the efficiency of algorithms is implementation dependent, with certain timing statistics designed to reflect how the addition should occur. Analysis will only take place for addition.

5.1 Method

Regarding the method of simulation and timing, for each k separately:

1. The sequential addition:

Ripple carry adder with carry C_i is performing xor(A_i, B_i), and(A_i, B_i), then xor(C_i, xor(A_i, B_i)), and(C_i, xor(A_i, B_i)), lastly xor(and(A_i, B_i), and(C_i, xor(A_i, B_i))) [3].

- Begin the timer.
- Generate two randomised n-bit size numbers for each trial.
- Perform Ripple Carry Addition sequentially.
- End the timer.

Then, the total time taken is calculated as endTime - startTime.

Because ripple carry addition uses n full adders, with each full adder containing internal parallelism, we compute the sequential time taken:

- xor requires 7 * 4 + 4 steps and 3 time intervals.
- and requires 4 steps.
- Assume 2 * 10 random number generation.
- Each step of ripple carry requires 3 time intervals.
- Then sequentialTime = (totalTime / stepsPerI) * timeIntervals.
- Since list addresses are calculated on the authors machine with 32-bit addressing:
    - Divide the total time taken by 32 to remove the addressing component of time taken.

2. The conceptual parallel addition:

- Begin the timer.
- Generate two randomised n-bit size numbers for each trial.

- Perform conceptual parallel addition sequentially.
- End the timer.

Then, the total time taken is calculated as endTime - startTime.
- Since we are using n "concurrent" processors, parallel time is the total time taken / n.
- Since list addresses are calculated on the authors machine with 32-bit addressing:
    - Divide the parallel time by 32 to remove the addressing component of time taken.
    - Multiply this by $log(n)$ because the process requires addressing with $log(n)$ bits.

3. The logical parallel addition:
- Begin the timer.
- Generate two randomised n-bit size numbers for each trial.
- Perform conceptual parallel addition sequentially.
- End the timer.

Then, the total time taken is calculated as endTime - startTime.

Calculating parallelTime here is slightly more tricky. By approximating the number of sequential steps taken for each stage of the simulation (number generation, circuits, etc) and the number of time intervals the circuits take to complete their work, we find:
- Convert to opcode and result requires $19 * 4 + 12 = 88$ steps over 1 call and 4 time intervals.
- Propagate requires $40 * 4 + 17 = 177$ steps for the single output variant over 1 call and 5 time intervals.
- Propagate requires $(log(n) - 1) * (40 * 4 + 24) = 184log(n) - 184$ steps for the split output variant over $log(n) - 1$ calls and 5log(n) - 5 time intervals.
- Convert to carry and result requires $7 * 4 + 8 = 36$ steps over 1 call and 3 time intervals.
- XOR carry and result requires $7 * 4 + 6 = 34$ steps over 1 call and 3 time intervals.
- Assume 2 * 10 steps for random number generation over 1 trial and 1 time interval.

Then the steps per intervals is the sum of steps and the number of time intervals is the sum of time intervals.
- parallelTime is calculated as ((totalTime / stepsPerI) / size)*timeIntervals.
- adjustedParallelTime is calculated as parallelTime / 32 bit register size.

5.2 Results

The simulation code for these processes are timed for numbers A and B of size $n = 2^k, 4 \leq k \leq 16$ over 100 trials for each k. Results are tabulated as follows:

Table 4.1: Running Time of Sequential Simulation (Ripple Carry Addition).

size (bits)	totalTime (s)	sequentialTime (s)	adjustedSequentiallTime (s)
16	2.11e-02	1.83e-03	5.71e-05
32	4.21e-02	3.64e-03	1.14e-04
64	7.80e-02	6.75e-03	2.11e-04
128	1.61e-01	1.39e-02	4.36e-04
256	3.27e-01	2.83e-02	8.84e-04
512	6.59e-01	5.70e-02	1.78e-03
1024	1.25e00	1.08e-01	3.38e-03
2048	2.70e00	2.34e-01	7.30e-03
4096	5.35e00	4.63e-01	1.45e-02
8192	1.05e01	9.06e-01	2.83e-02
16384	2.15e01	1.86e00	5.82e-02
32768	4.07e01	3.52e00	1.10e-01
65536	8.03e01	6.95e00	2.17e-01

Table 4.2: Running Time of $log^2(n)$ Simulation (Conceptual Process of Addition).

size (bits)	totalTime (s)	parallelTime (s)	adjustedParallelTime (s)
16	9.00e-03	5.63e-04	7.04e-05
32	2.49e-02	7.79e-04	1.22e-04
64	4.19e-02	6.54e-04	1.23e-04
128	1.00e-01	7.87e-04	1.72e-04
256	2.22e-01	8.71e-04	2.18e-04
512	4.82e-01	9.41e-04	2.65e-04
1024	1.00e00	9.80e-04	3.06e-04
2048	2.17e00	1.06e-03	3.63e-04
4096	4.58e00	1.11e-03	4.19e-04
8192	9.83e00	1.20e-03	4.87e-04
16384	2.06e01	1.26e-03	5.51e-04
32768	4.22e01	1.29e-03	6.03e-04
65536	9.90e01	1.51e-03	7.55e-04

Table 4.3: Running Time of $log(n)$ Simulation (Logical Process of Addition).

size (bits)	totalTime (s)	parallelTime (s)	adjustedParallelTime (s)
16	1.41e-01	3.00e-04	9.39e-06
32	3.17e-01	3.26e-04	1.02-05
64	7.67e-01	3.85e-04	1.20e-05
128	1.69e-00	4.16e-04	1.30e-05
256	3.73e-00	4.52e-04	1.41e-05
512	7.74e-00	4.62e-04	1.45e-05
1024	1.78e01	5.27e-04	1.65e-05
2048	4.06e01	5.96e-04	1.86e-05
4096	8.01e01	5.87e-04	1.85e-05
8192	1.75e02	6.34e-04	1.98e-05
16384	3.73e02	6.72e-04	2.10e-05
32768	8.28e02	7.42e-04	2.32e-05
65536	1.68e03	7.50e-04	2.35e-05

5.3 Discussion

The total time for each simulation appears to be growing linearly to the size of the input. This is due to the fact that the computer is using a fixed size register to perform computations. The time taken to perform these additions over 100 trials may vary due to influences such as background tasks running concurrently and the CPU speed being throttled due to higher temperature after running for a period of time. Also as we approach the larger numbers, running time may increase further due to the necessity of transferring data between cache or even hard drive which can be orders of magnitude slower.

Times between processes may also not be directly comparable (parallelTime and adjustedParallelTime) due to the assumptions and rough estimates made regarding the number of steps in each process when calculating adjustments to account for parallelism. For this reason, the parallelTime and adjustedParallelTime results are rather contrived and are present to model how the method should be performing based on the times and steps we see.

Inspecting the adjustedSequentialTime metrics, a relatively linear increase in running time occurs as n increases. The adjustedParallelTime for the $log^2(n)$ simulation increases relatively to $log^2(n)$, showing a significant improvement on the sequential version by size $n = 65536$. We see with totalTime, that the $log^2(n)$ increases similarly to the sequential versions' totalTime, but this

should not be the case given there is a significantly larger amount of data being transmitted between its processing elements when considering the thread based approach. This has to do with the method of implementation, but as it is supposed to be a conceptual process for which to build the $log(n)$ version, this similarity in timing is unconcerning.

The totalTime for the $log(n)$ version appears to be roughly two orders of magnitude greater than the totalTime of the sequential and $log^2(n)$ versions. This indicates a significant amount of overhead which happens to be present especially within the propagation circuit over $log(n)$ rows (the previously estimated coefficient being $184log(n)$). This is expected and indicates that this method is not practicably useful until large numbers are reached, when the overheads become less relevant compared to size. The adjustedParallelTime for this method appears more promising but once again this is an estimate and not directly comparible. This adjusted time however, does grow relatively to $log(n)$ which would become noticably slower compared to the others.

The simulation sizes began at size $n = 2^4$ to $n = 2^16$ due to the fact that any smaller may cause significant rounding errors in the timer and therefore be unusable, and, that larger sizes will take too long to time and require increasingly slower methods of data transfers between CPU, cache and hard drive. This would artificially increase times by a significant amount and is an unwanted effect to avoid.

Future improvements may consist of better implementation choices, designed to be faster and simpler to estimate the number of steps as well as searching for more optimal configuations of circuits and designs requiring less steps if possible. None of this work has claimed to be near optimal and the largest source of improvement appears to stem from the complexity that arises from the use of opcodes. Therefore eliminating this requirement in future would provide greater benefits (given the propagation circuit requires space proportional to $nlog(n)$ and time to $log(n)$ from the $log(n)$ rows of these processing elements). Analysis has neglected considerations such as the time taken for an electrical signal to propagate through the circuit, rather the assumption that nodes are separated by discrete time intervals has been used to allow for synchronisation and may not reflect the true physical nature of the circuit.

6 Supplementary Materials

This section presents the Python 3 code used in the simulations of addition for the sequential, parallel $log^2(n)$ and parallel $log(n)$ time processes separately to avoid obsstructing the flow of ideas in previous sections. This code will only only simulate addition.

6.1 Sequential Simulation Code

The following code is a simulation of sequential Ripple Carry Addition for use as a benchmark against the other addition processes:

```
# Import statements.
import random
import time

# Timing and control variables.
totalTime, sequentialTime = 0, 0
adjustedSequentialTime = 0
minExponent, maxExponent = 4, 5
numTests = 100
myComputerRegisterSize = 32


# Generate a list of random bits of size n.
def generateRandomNumberDigitList(numDigits):
    digitList = [0] * numDigits
    for i in range(1, numDigits):
        digitList[i] = random.randrange(2)
    return digitList


# And gate.
def andGate(a):
    return int(a[0] and a[1])


# Or gate.
def orGate(a):
    return int(a[0] or a[1])


# Not gate.
```

```
def notGate(a):
    return int(not a[0])


# Repeater gate.
def repeaterGate(a):
    return int(a[0])


# Store gate functions in an array where the index of the function
# is its function code.
gateArray = [andGate, orGate, notGate, repeaterGate]


# The final result of Ai + Bi at location i.
def xor(c, r):
    t1 = [gateArray[2]([c]), gateArray[2]([r]), gateArray[3]([c]),
        gateArray[3]([r])]
    t2 = [gateArray[0]([t1[0], t1[3]]), gateArray[0]([t1[1], t1[2]])]
    t3 = [gateArray[1]([t2[0], t2[1]])]
    out = t3[0]
    return out


exponent, size = 0, 0
startTiming = time.time()
for exponent in range(minExponent, maxExponent):
    size = 2 ** exponent
    for tests in range(numTests):
        list1 = generateRandomNumberDigitList(size)
        list2 = generateRandomNumberDigitList(size)

        carryResult = [0] * (size + 1)
        finalResult = [0] * size

        # n ripple carry additions, store carry and result.
        for iterI in range(size - 1, -1, -1):
            xorAB = xor(list1[iterI], list2[iterI])
            sout = xor(xorAB, carryResult[iterI])
            andAB = andGate([list1[iterI], list2[iterI]])
            andCxorAB = andGate([carryResult[iterI], xorAB])
            cout = xor(andAB, andCxorAB)
            finalResult[iterI] = sout
            carryResult[iterI - 1] = cout
```

```
totalTime = time.time() - startTiming

# ————————————————————
# Approximating steps taken and timing:
# xor requires 7 * 4 + 4 steps and 3 time intervals.
# and requires 4 steps.
# Assume 2 * 10 random number generation.
# Each step of ripple carry requires 3 time intervals.

stepsPerI = (3 * (7 * 4 + 4)) + (2 * 4)

timeIntervals = 3 * 3

sequentialTime = (totalTime / stepsPerI) * timeIntervals

# 32 bit register calculations are being done by the computer,
# divide by this to eliminate register computations.
adjustedSequentialTime = sequentialTime / myComputerRegisterSize

print("Over", numTests, "test/s:")
print("Size: ", size, "bits.")
print("Total Time:", totalTime, "seconds.")
print("Sequential Time:", sequentialTime, "seconds.")
print("Adjusted Sequential Time:", adjustedSequentialTime, "seconds.")
```

6.2 $log^2(n)$ Simulation Code

The following code is a simulation of the $log^2(n)$ addition method:

```
# Import statements.
import random
import time


# Timing and control variables.
totalTime, parallelTime = 0, 0
adjustedParallelTime = 0
minExponent, maxExponent = 16, 17
numTests = 100
myComputerRegisterSize = 32


# Generate a list of random bits of size n.
```

```
def generateRandomNumberDigitList(numDigits):
    digitList = [0] * numDigits
    for i in range(1, numDigits):
        digitList[i] = random.randrange(2)
    return digitList


# Labels: Prop = 0, PropP = 1, PropB = 2, Amb = 3, Bar = 4,
# BarP = 5, BarB = 6.
size, exponent = 0, 0
startTiming = time.time()
for exponent in range(minExponent, maxExponent):
    size = 2 ** exponent
    for tests in range(numTests):
        list1 = generateRandomNumberDigitList(size)
        list2 = generateRandomNumberDigitList(size)

        # H = convert a and b to labels.
        H, Hd = [4 - ((list1[i] + list2[i]) ** 2)
            for i in range(size)], [3 for i in range(size)]

        # Run algorithm 2.4.1 to propagate carry info.
        for m in range(exponent):
            j = 2 ** m
            Hd = [H[i+j] if i + j ¡ size else Hd[i] for i in range(size)]
            for i in range(size):
                if H[i] == 0:
                    if Hd[i] < 3:
                        H[i] = 1
                    elif Hd[i] > 3:
                        H[i] = 2

                elif H[i] == 3:
                    if Hd[i] < 3:
                        H[i] = 1
                    elif Hd[i] > 3:
                        H[i] = 6

                elif H[i] == 4:
                    if Hd[i] < 3:
                        H[i] = 5
                    elif Hd[i] > 3:
                        H[i] = 6
                else:
                    H[i] = H[i]
```

```
        # Take the addition result xor R xor C.
        finalResult = [(list1[i] + list2[i] + (H[i] == 1) + (H[i] == 5))
            % 2 for i in range(size)]

totalTime = time.time() - startTiming

# ------------------------
# Time calculations.
parallelTime = totalTime / size

# 32 bit register calculations are being done by the computer,
# divide by this to eliminate register computations.
adjustedParallelTime = (parallelTime / 32) * exponent

print("Over", numTests, "test/s:")
print("Size: ", size, "bits.")
print("Total Time:", totalTime, "seconds.")
print("Parallel Time:", parallelTime, "seconds.")
print("Adjusted Parallel Time:", adjustedParallelTime, "seconds.")
```

6.3 $log(n)$ Simulation Code

The following code is a simulation of the $log(n)$ addition method:

```
# Import statements.
import random
import time

# Timing and control variables.
totalTime, parallelTime = 0, 0
adjustedParallelTime = 0
minExponent, maxExponent = 15, 16
numTests = 100
myComputerRegisterSize = 32


# Generate a list of random bits of size n.
def generateRandomNumberDigitList(numDigits):
    digitList = [0] * numDigits
    for i in range(1, numDigits):
    digitList[i] = random.randrange(2)
    return digitList
```

```
# And gate.
def andGate(a):
    return int(a[0] and a[1])


# Or gate.
def orGate(a):
    return int(a[0] or a[1])


# Not gate.
def notGate(a):
    return int(not a[0])


# Repeater gate.
def repeaterGate(a):
    return int(a[0])


# Store gate functions in an array where the index of the function
# is its function code.
gateArray = [andGate, orGate, notGate, repeaterGate]


# The final result of Ai + Bi at location i.
def xorCarryAndResult(c, r):
    t1 = [gateArray[2]([c]), gateArray[2]([r]), gateArray[3]([c]),
        gateArray[3]([r])]
    t2 = [gateArray[0]([t1[0], t1[3]]), gateArray[0]([t1[1], t1[2]])]
    t3 = [gateArray[1]([t2[0], t2[1]])]
    out = t3[0]
    return out


# The conversion from opcode to Ci and Ri at location i.
def convertOpcodeToCarryRes(d2, d3, r):
    t1 = [gateArray[2]([d2]), gateArray[3]([d3]), gateArray[3]([r])]
    t2 = [gateArray[1]([t1[0], t1[1]]), gateArray[3]([t1[2]])]
    t3 = [gateArray[2]([t2[0]]), gateArray[3]([t2[1]])]
    out = [t3[0], t3[1]]
    return out


# The conversion of Ai and Bi into (d1, d2, d3)i and Ri at location i.
```

```
def convertIntoOpcodeAndResult(a, b):
    t1 = [gateArray[0]([a, b]), gateArray[2]([a]), gateArray[2]([b]),
        gateArray[3]([a]), gateArray[3]([b])]
    t2 = [gateArray[3]([t1[0]]), gateArray[0]([a, t1[1]]),
        gateArray[0]([t1[3], t1[2]]),
        gateArray[0]([t1[1], t1[4]])]
    t3 = [gateArray[3]([t2[0]]), gateArray[3]([t2[1]]),
        gateArray[1]([t2[2], t2[3]])]
    t4 = [gateArray[3]([t3[0]]), gateArray[3]([t3[1]]),
        gateArray[2]([t3[2]]), gateArray[3]([t3[2]])]
    out = [t4[0], t4[0], t4[1], t4[1], t4[2], t4[2], t4[3]]
    return out


# The propagation of carry information that splits information into
# two output streams.
def propagate(l1, l2, l3, r1, r2, r3, r, numOutputs):
    t1 = [gateArray[3]([l1]), gateArray[2]([l2]), gateArray[3]([l2]),
        gateArray[2]([l3]), gateArray[3]([l3]), gateArray[3]([r1]),
        gateArray[2]([r1]), gateArray[3]([r2]), gateArray[3]([r3]),
        gateArray[3]([r])]
    t2 = [gateArray[3]([t1[0]]), gateArray[0]([t1[1], t1[3]]),
        gateArray[0]([t1[2], t1[4]]), gateArray[3]([t1[5]]),
        gateArray[0]([t1[1], t1[6]]), gateArray[0]([t1[4], t1[6]]),
        gateArray[1]([t1[2], t1[7]]), gateArray[3]([t1[7]]),
        gateArray[3]([t1[8]]), gateArray[3]([t1[9]])]
    t3 = [gateArray[3]([t2[0]]), gateArray[0]([t2[1], t2[3]]),
        gateArray[1]([t2[2], t2[5]]), gateArray[1]([t2[6], t2[8]]),
        gateArray[0]([t2[4], t2[7]]), gateArray[0]([t2[4], t2[8]]),
        gateArray[3]([t2[9]])]
    t4 = [gateArray[1]([t3[0], t3[1]]), gateArray[3]([t3[2]]),
        gateArray[3]([t3[3]]), gateArray[1]([t3[4], t3[5]]),
        gateArray[3]([t3[6]])]
    t5 = [gateArray[3]([t4[0]]), gateArray[3]([t4[2]]),
        gateArray[1]([t4[1], t4[3]]), gateArray[3]([t4[4]])]
    if numOutputs == 1:
        out = [t5[0], t5[1], t5[2], t5[3]]
    else:
        out = [t5[0], t5[0], t5[1], t5[1], t5[2], t5[2], t5[3]]
    return out


exponent, size = 0, 0
startTiming = time.time()
for exponent in range(minExponent, maxExponent):
    size = 2 ** exponent
```

```
for tests in range(numTests):
    list1 = generateRandomNumberDigitList(size)
    list2 = generateRandomNumberDigitList(size)

    # Convert to opcode and result.
    opAndRes = [0] * size
    for iterI in range(size):
        opAndRes[iterI] = convertIntoOpcodeAndResult(list1[iterI],
            list2[iterI])

        # Make logn - 1 carry prop rows with split output.
        numPropSplitOutputRows = exponent - 1
        propRes = [0] * size

        # Do logn - 1 props over all i.
        for m in range(numPropSplitOutputRows):
            j = 2 ** m
            for iterI in range(size):
                # if i + j < size: use 2 inputs l1,2,3 and r1,2,3, else
                 # use l1,2,3 and 0,0,0. Store result in prop res.
                if iterI + j < size:
                    propRes[iterI] = propagate(opAndRes[iterI][0],
                        opAndRes[iterI][2],
                        opAndRes[iterI][4],
                        opAndRes[iterI + j][1],
                        opAndRes[iterI + j][3],
                        opAndRes[iterI + j][5],
                        opAndRes[iterI][6], 2)
                 else:
                    propRes[iterI] = propagate(opAndRes[iterI][0],
                    opAndRes[iterI][2],
                    opAndRes[iterI][4],
                    0, 0, 0,
                    opAndRes[iterI][6], 2)

            # Copy prop res to op and res for input reuse.
            for iterI in range(size):
                opAndRes[iterI] = propRes[iterI]

        # m = log(n)-1 now, last prop with non split output in
        # prop res.
        j = 2 ** numPropSplitOutputRows
        for iterI in range(size):
            if iterI + j < size:
                propRes[iterI] = propagate(opAndRes[iterI][0],
                    opAndRes[iterI][2],
```

```
                        opAndRes[iterI][4],
                        opAndRes[iterI + j][1],
                        opAndRes[iterI + j][3],
                        opAndRes[iterI + j][5],
                        opAndRes[iterI][6], 1)
                else:
                    propRes[iterI] = propagate(opAndRes[iterI][0],
                        opAndRes[iterI][2],
                        opAndRes[iterI][4],
                        0, 0, 0,
                        opAndRes[iterI][6], 1)

            carryAndRes = [0] * size
            # Convert props into carries.
            for iterI in range(size):
                carryAndRes[iterI] = convertOpcodeToCarryRes(propRes[iterI][1],
                    propRes[iterI][2],
                    propRes[iterI][3])

            # xor carry and result.
            finalOutput = [0] * size
            for iterI in range(size):
                finalOutput[iterI] = xorCarryAndResult(carryAndRes[iterI][0],
                    carryAndRes[iterI][1])

totalTime = time.time() - startTiming

# ----------------------
# Approximating steps taken and timing given simulation is sequential:
# Convert to opcode and R requires 19 * 4 + 10 steps in circuit and 4
# time intervals. + 2 list creation/population.

# Propagate requires 40 * 4 + 15 steps if one output; over 1 call: + 2
# list creation/population
# 40 * 4 + 18 steps if two outputs; over log(n) - 1
# calls: + 6 list creation/transfer
# and 5 time intervals.

# Convert to carry requires 7 * 4 + 6 steps and 3 time intervals.
# + 2 list creation/population.

# XOR carry and R requires 7 * 4 + 4 steps and 3 time intervals.
# + 2 list creation/population.

# Assume 2 * 10 random number generation and 1 time interval.
# Assume 1 * log(n) j rows, already factored into propagate.
```

```
stepsPerI = (19 * 4 + 12) + (40 * 4 + 17) + ((40 * 4 + 24)
    * (exponent - 1)) + (7 * 4 + 8) + (7 * 4 + 6) + (2 * 10)

timeIntervals = 4 + 5 + (5 * (exponent - 1)) + 3 + 3 + 1

parallelTime = (totalTime / stepsPerI) / timeIntervals

# 32 bit register calculations are being done by the computer,
# divide by this to eliminate register computations.
adjustedParallelTime = parallelTime / myComputerRegisterSize

print("Over", numTests, "test/s:")
print("Size: ", size, "bits.")
print("Total Time:", totalTime, "seconds.")
print("Parallel Time:", parallelTime, "seconds.")
print("Adjusted Parallel Time:", adjustedParallelTime, "seconds.")
```

7 References

[1] "Online Karnaugh map solver with circuit for up to 6 variables," www.32x8.com. http://www.32x8.com/index.html

[2] Brent and Kung, "A Regular Layout for Parallel Adders," IEEE Transactions on Computers, vol. C–31, no. 3, pp. 260–264, Mar. 1982, doi: 10.1109/tc.1982.1675982.

[3] "Ripple carry adder, 4 bit ripple carry adder circuit , propagation delay," Electronic Circuits and Diagrams-Electronic Projects and Design, Mar. 15, 2012. https://www.circuitstoday.com/ripple-carry-adder