METHOD OF TRANSFORMING AND ADDING BINARY STRINGS, AND CIRCUITS THEREFOR

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority and the benefit of PCT/CN2024/085517, filed April 2, 2024, the entirety of which is incorporated herein by reference

TECHNICAL FIELD OF THE INVENTION

The present invention relates to logic circuits for manipulating the bits in binary strings. In particular, the invention relates to logic circuits for manipulating the bits in binary strings so that the binary strings add up in separate, parallel adding operations for each bit position.

BACKGROUND OF THE INVENTION

A processor comprises logic circuits for performing arithmetic operations on numbers in the form of binary strings, i.e. addition, subtraction, multiplication and division. One of the most basic of these is the digital adder.

Figure 1 shows a generic Half-Adder circuit comprising an AND gate and an Exclusive OR gate. A Half-Adder takes in two single-bit inputs and produces two single-bit outputs, one of which is called the SUM bit, and the other called the Carry-

out (C_{OUT}) bit. If both input bits are 1, a carry is generated because the result is equal to 10.

Figure 2 shows the Full-Adder which is made of two Half-Adders connected together as shown, with the first Half-Adder passing its carry to the second Half-Adder. The Truth Table for the Full-Adder includes an additional input for taking in any carry bit from the summation of the bits in the preceding lower significant place.

The output is the sum and a Carry-out to be connected to the Carry-in of the Full-Adder in the succeeding higher significant position. This allows the series of cascading Full-Adders to add numbers of binary strings in which carry bits are passed on to the more significant bit, in what is called the “ripple effect”.

A series of four Full-Adders can be combined to provide a 4-bit adder, illustrated in Figure 3. The rippling effect from all the carry-overs imposes a delay. Each adder must wait for the carry-over signal before the adder has all the inputs required to produce an output. Each of this waiting time in a series of adders adds to an overall performance delay, called the propagation delay. Propagation delay worsens with greater number of Full-Adders arranged serially to add longer binary strings.

To mitigate the extent of propagation delay, a circuit modification to serially arranged full adders called the Carry Look-ahead Binary Adder has been proposed, as illustrated in Figure 4. Basically, there are some combinations of inputs to the Full-Adder that can predict a carry-over bit, and the inputs to the Full-Adder are provided to an additional logic component called the Carry Look Ahead which decides to send

the carry-over signal to downstream adders while the upstream adders are still calculating. There are two main conditions that can cause a downstream adder to receive a carry-over bit, i.e. when the upstream adder generates a carry-over or passes on a carry-over it received, and these signalled to the downstream adders a logic combination of ‘carry generates’ and ‘carry propagates’, and possibly “group generates” and “group propagates” if there are long binary strings which are divided into multiple groups of bits.

The extent of time saving using carry look-ahead logic is almost exponential in adders of longer binary strings, but so are the accompanying costs and complexity of the circuit design, which spurred the infamous Moore’s Law on the relationship of cost and complexity. The impetus to develop adders of larger capacity wanes against economical and the burden of complexity.

Despite the breakthrough of carry look-ahead logic, the prior art has found no way of escaping the ripple effect, thus making all addition an essentially serial process from the least significant bit to the most significant bit.

Therefore, it is desirable to propose novel methods and designs of logic circuits that could remove or mitigate in a better way any one or more of the discussed limitations.

SUMMARY OF THE INVENTION

In a first aspect, the invention proposes a method of making available a bit position i in one of two binary strings to store a carry-over value, comprising the step(s) of:

replacing a nonzero with a zero in the bit position i, the nonzero in the bit position representing a value; and providing at least one nonzero in a different bit position in the two binary strings representing an equal value.

The method provides the possibility of “moving aside” a nonzero bit from a certain position in a binary string, to make way for a possible carry-over to occupy that vacated position. If every carry-over has a position to occupy, propagation of the carry-over is eliminated. The advantage will be better appreciated if one considers how a processor has to sum millions and millions of binary strings in regular use in a regular computer.

An example of the method in use is shown in Box 25, in relation to a Transformation B.

On one hand, therefore, the step of providing at least one nonzero in a different bit position in the two binary strings is taken when executing either one of case 2) and case 3) according to the transformation truth table:

	Input		Output	
	$(x_i'')_1$	$(x_i'')_2$	$(x_{i+1}''')_2$	$(x_i''')_1$
1)	0	0	0	0
2)	0	1	0	1
3)	1	0	0	1
4)	1	1	1	0

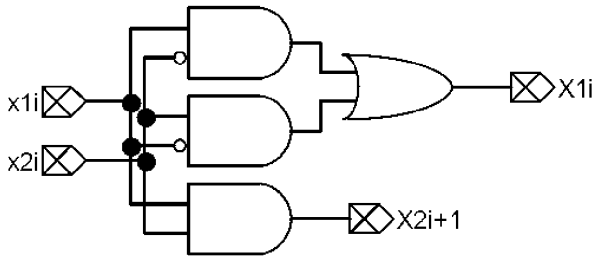
for position i in an augend binary string $(x_i'')_1$ and an addend binary string $(x_i'')_2$ and in transformed augend binary strings $(x_{i+1}''')_2$ and $(x_i''')_1$.

When the method is applied to “generalised binary strings” that have positive and negative binary strings, which will be explained in the following section, it is more complete to say that the step of providing at least one nonzero in a different bit in the two binary strings is taken when executing either one of case 2) and case 3) according to the transformation truth table:

	Positive input		Output		Negative input		Output	
	$(x_i'')_1$	$(x_i'')_2$	$(x_{i+1}''')_2$	$(x_i''')_1$	$(y_i'')_1$	$(y_i'')_2$	$(y_{i-1}''')_2$	$(y_i''')_1$
1)	0	0	0	0	0	0	0	0
2)	0	1	0	1	0	$\bar{1}$	0	$\bar{1}$
3)	1	0	0	1	$\bar{1}$	0	0	$\bar{1}$
4)	1	1	1	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0

the two binary strings each being one of a pair of generalised binary strings $(x_i'')_1$, $(y_i'')_1$and $(x_i'')_2$, $(y_i'')_2$, and of the same sign component.

In a further aspect, the invention proposes a logic circuit for applying a transformation according to the truth tables above shown comprising the following configuration:



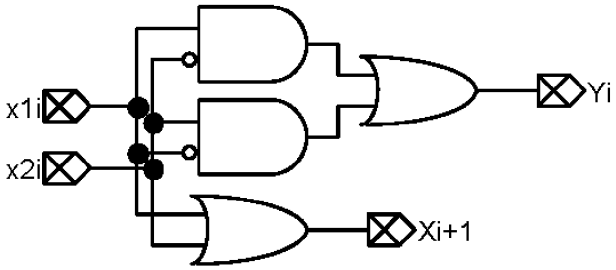
On the other hand, however, the method comprises using at least one nonzero in a further binary string representing a value of a different sign component, said equal value is provided by offsetting the value of the at least one nonzero in a different bit position in the two binary strings with the at least one nonzero in the further binary string. An example of this is Transformation A described as an embodiment.

Preferably, the step of providing at least one nonzero in a different bit in the two binary strings is taken when executing either one of case 2) and case 3) according to the transformation truth table:

	Input n_w^+		Output $(n'_w)_1$		Input n_w^-		Output $(n'_w)_2$	
	$(x_i)_1$	$(x_i)_2$	$(x'_{i+1})_1$ ‡	$(y'_i)_1$ §	$(y_i)_1$	$(y_i)_2$	$(y'_{i+1})_2$	$(x'_i)_2$
1)	0	0	0	0	0	0	0	0
2)	0	1	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	1
3)	1	0	1	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	1
4)	1	1	1	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0

the two binary strings each being one of a pair of generalised binary strings, $(x_i\ )_1$, $(y_i\ )_1$ and $(x_i\ )_2$, $(y_i\ )_2$, and of the same sign component.

Accordingly, in a further aspect, the invention proposes a logic circuit for applying a transformation according to the truth table above, comprising the following configuration:



In yet another aspect, the invention proposes a method of adding at least two numbers in the form of binary strings, the method comprising the steps of:

i. providing a first pair of binary substrings, x_1 and y_1 wherein x_1-y_1 gives the value of a first binary string representing a first number;

ii. providing a second pair of binary substrings, x_2 and y_2 wherein x_2 - y_2 gives the value of a second binary string representing a second number;

iii. performing the following transformations and operation:

Transformation A:

producing a third pair of substrings x'_1 and y'_1 from x_1 and x_2 and producing a fourth pair of substrings x'_2 and y'_2 from y_1 and y_2 according to the truth table where i is bit position:

Input n_w^+		Output $(n'_w)_1$		Input n_w^-		Output $(n'_w)_2$	
$(x_i)_1$	$(x_i)_2$	$(x'_{i+1})_1$ ‡	$(y'_i)_1$ §	$(y_i)_1$	$(y_i)_2$	$(y'_{i+1})_2$	$(x'_i)_2$
0	0	0	0	0	0	0	0
0	1	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	1
1	0	1	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	1
1	1	1	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0

Simplification Transformation:

producing substrings x''_1 and y''_1 from x'_1 and y'_1 by replacing the nonzero values where x'_1 = 1 and y'_1= 1 with zeros for each bit position i; and

producing substrings x''_2 and y''_2 from x'_2 and y'_2 = by replacing the nonzero values where x'_2 = 1 and y'_2 = 1 with zeros in x''_2 and y''_2 for each bit position i;

Transformation B:

consolidating the values of the x''_1, x''_2 to produce a pair of substrings x'''_1 and x'''_2, and further consolidating the values of y''_1, y''_2 to produce a pair of substrings y'''_1 and y'''_2, according to the truth table:

	Positive input		Output		Negative input		Output	
	$(x_i'')_1$	$(x_i'')_2$	$(x_{i+1}''')_2$	$(x_i''')_1$	$(y_i'')_1$	$(y_i'')_2$	$(y_{i-1}''')_2$	$(y_i''')_1$
1)	0	0	0	0	0	0	0	0
2)	0	1	0	1	0	$\bar{1}$	0	$\bar{1}$
3)	1	0	0	1	$\bar{1}$	0	0	$\bar{1}$
4)	1	1	1	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0

Summing Operation:

producing substrings ***X*** and ***Y*** according to the truth table:

Inputs				Outputs	
x'''_1	y'''_1	x'''_2	y'''_2	X	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	0	0	$\bar{1}$	0	$\bar{1}$
1	0	0	0	1	0
1	0	0	$\bar{1}$	0	0
0	$\bar{1}$	0	0	0	$\bar{1}$
0	$\bar{1}$	1	0	0	0

iv. adding up X and Y to produce a single binary string having the value of the sum of the two numbers.

Preferably, before step iv, the method further comprises the steps of:

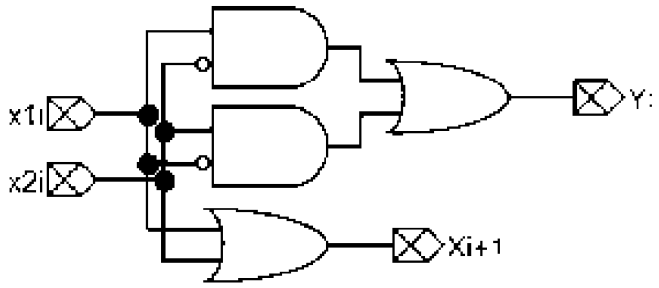
providing a third pair of binary substrings, x_3 and y_3, x_3- y_3 being the value of a third binary string representing a third number;

repeating iii, by taking

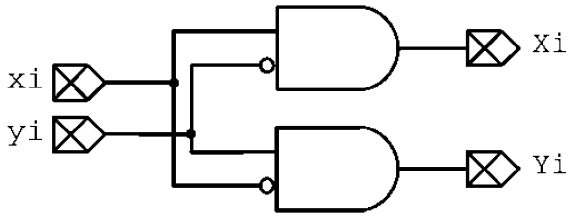
- X substring produced by the Summing Operation as x_1

- Y substring produced by the Summing Operation as y_1;
- x_3 as x_2;
- y_3 as y_2.

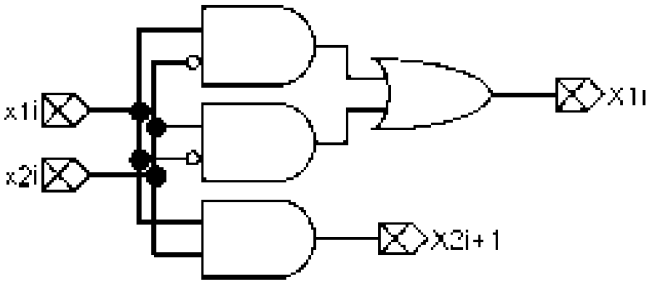
In another aspect, therefore, the invention proposes a logic circuit for applying the Transformation A, comprising the following configuration:



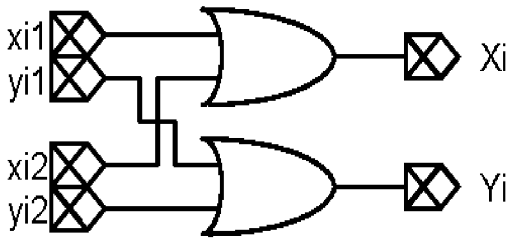
Furthermore, the invention proposes a logic circuit for applying the Simplification Transformation comprising the following configuration:



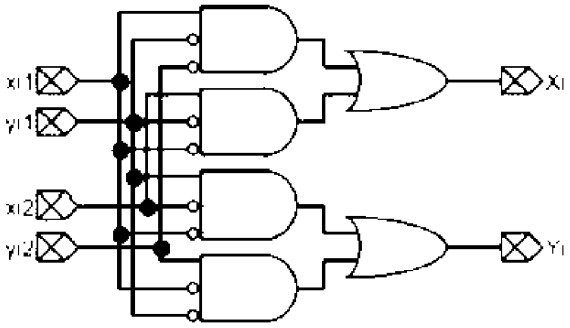
Furthermore, the invention proposes a logic circuit for applying the Transformation B comprising the following configuration:



Furthermore, the invention proposes a logic circuit for applying Summing Operation comprising the following configuration:



Alternatively, the logic circuit for applying Summing Operation is as shown below:



In yet a further aspect, the invention proposes a method of adding two numbers in the form of binary strings, the method comprising the steps of:

providing a first binary string representing the first number and a second binary string representing the second number to each of the first binary string; and the second binary string, assigning a further binary string to form a pair of binary substrings, such that

the sum of a first pair of positive substring x_1 and negative substring y_1 is the value of the first binary string; and

the sum of a second pair of positive substring x_2 and negative substring y_2 is the value of the second binary string;

performing the following transformations and operation:

Sedimentation Transformation:

re-arranging the nonzero bits between x_1 and x_2, and between y_1 and y_2, according to the truth table to output the substrings x_1' , x_2' , y_1' , y_2'

Input		Output		Input		Output	
$(x_i)_1$	$(x_i)_2$	$(x_i')_1$	$(x_i')_2$	$(y_i)_1$	$(y_i)_2$	$(y_i')_1$	$(y_i')_2$
0	0	0	0	0	0	0	0
0	1	1	0	0	$\overline{1}$	$\overline{1}$	0
1	0	1	0	$\overline{1}$	0	$\overline{1}$	0
1	1	1	1	$\overline{1}$	$\overline{1}$	$\overline{1}$	$\overline{1}$

Transformation AB:

taking substrings x_1' , x_2' , y_1' , y_2' as input to produce substrings x_1'' , x_2'' , x_3'' , y_1'' , y_2'', y_3'' as output according to the following truth table:

Input			Output		
$(x_i')_1$	$(x_i')_2$	$(x_{i-1}')_1$	$(x_i'')_1$	$(x_i'')_2$	$(y_i'')_3$
0	0	0	0	0	0
0	0	1	0	1	0
1	0	0	0	0	$\overline{1}$
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	1	0	0
Input			Output		
$(y_i')_1$	$(y_i')_2$	$(y_{i-1}')_1$	$(y_i'')_1$	$(y_i'')_2$	$(x_i'')_3$
0	0	0	0	0	0
0	0	$\overline{1}$	0	$\overline{1}$	0
$\overline{1}$	0	0	0	0	1
$\overline{1}$	0	$\overline{1}$	0	0	0
$\overline{1}$	$\overline{1}$	0	0	0	0
$\overline{1}$	$\overline{1}$	$\overline{1}$	$\overline{1}$	0	0

Summing Operation:

taking substrings x_1'', x_2'', x_3'', y_1'', y_2'', y_3'' as input to produce output substrings x_i''' , y_i''' as output according to the following truth table:

Input					Output
$(x_i'')_1$	$(x_i'')_2$	$(x_i'')_3$	$(x_{i-1}'')_2$	$(x_{i-1}'')_3$	x_i'''
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	0	1
1	0	0	0	1	1

Input					Output
$(y_i'')_1$	$(y_i'')_2$	$(y_i'')_3$	$(y_{i-1}'')_2$	$(y_{i-1}'')_3$	y_i'''
0	0	0	0	0	0
0	0	0	0	$\overline{1}$	0
0	0	0	$\overline{1}$	0	0
0	0	0	$\overline{1}$	$\overline{1}$	$\overline{1}$
0	0	$\overline{1}$	0	0	$\overline{1}$
0	0	$\overline{1}$	$\overline{1}$	0	$\overline{1}$
0	$\overline{1}$	0	0	0	$\overline{1}$
0	$\overline{1}$	0	0	$\overline{1}$	$\overline{1}$
0	$\overline{1}$	$\overline{1}$	0	0	0
$\overline{1}$	0	0	0	0	$\overline{1}$
$\overline{1}$	0	0	0	$\overline{1}$	$\overline{1}$

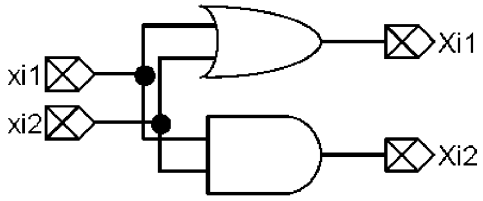
Preferably, the invention comprises a further step after Summing Operation, the step comprising of:

Simplification Transformation:

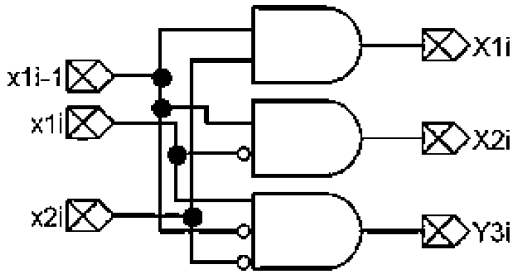
producing output substrings X and Y by replacing the nonzero values where $x_i''' = 1$, $y_i''' = 1$ with zeros for each bit position i.

Preferably, the invention comprises a further step of: adding up x_i''' and y_i''' to produce a single binary string having the value of the sum of the two numbers. This produces a regular binary string of the usual format which may be used in other computation processes, where the generalised binary format is not needed.

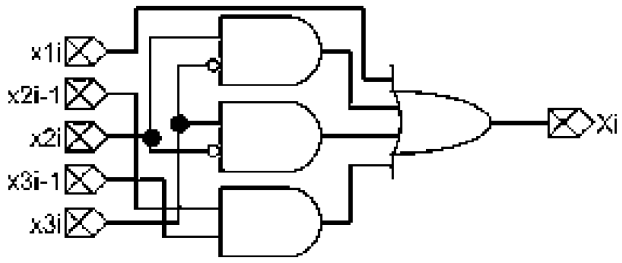
In a further aspect, the invention proposes a logic circuit for applying the Sedimentation Transformation comprising the following configuration:



In a further aspect, the invention proposes a logic circuit for applying the Transformation AB comprising the following configuration:



In a further aspect, the invention proposes a logic circuit for applying the Summing Operation which takes substrings x_1'', x_2'', x_3'', y_1'', y_2'', y_3'' as input, comprising the following configuration:



Among the described mathematical equations, logic circuits and logic truth tables in the following paragraphs of this description, the logic truth tables are the best guides for designing equivalent circuits as the logic truth tables shows the intended input and output, whereas the mathematical equations, and proposed logic may advertently contain an unnoticed error.

BRIEF DESCRIPTION OF THE FIGURES

It will be convenient to further describe the present invention with respect to the accompanying drawings that illustrate possible arrangements of the invention, in which like integers refer to like parts. Other arrangements of the invention are possible, and consequently the particularity of the accompanying drawings is not to be understood as superseding the generality of the preceding description of the invention.

Figure 1 illustrates a half-adder used in the prior art;

Figure 2 illustrates a full-adder used in the prior art;

Figure 3 illustrates a 4-bit ripple carry adder used in the prior art;

Figure 4 illustrates a Carry-Lookahead binary adder used in the prior art;

Figure 5 is a flowchart illustrating the steps in an embodiment of the invention;

Figure 6 is a subcircuit for converting the binary string of a number into the generalised format, which is used in the embodiment of Figure 5;

Figure 7 is a circuit diagram of the content in the subcircuit shown in Figure 6;

Figure 8 illustrates how the circuit diagrams of Figure 6 and Figure 7 generalize a binary string;

Figure 9 is the logic circuit for applying the Specific Equivalence Principle Conversion to x substrings;

Figure 10 is the logic circuit for applying the Specific Equivalence Principle Conversion to y substrings;

Figure 11 is the truth table for Specific Equivalence Principle Conversion in Figure 9 and Figure 10;

Figure 12 shows is a drawing corresponding to Stage 3 and Stage 4 in the flowchart of Figure 5;

Figure 13 is a 2-unit circuit block for performing Transformation A in Stage 3 in the flowchart of Figure 5;

Figure 14 a single unit of the logic circuit in Figure 13;

Figure 15 the truth table underlying the circuit block shown in Figure 13;

Figure 16 is a logic circuit for performing transformation in Simplification Transformation shown in Stage 3 in the flowchart of Figure 5;

Figure 17 is the truth table underlying the logic circuit shown in Figure 16;

Figure 18 is a 2-unit circuit used for processing a pair of units of Figure 19 for Transformation B in Stage 3 in the flowchart of Figure 5;

Figure 19 is a single unit for the circuit in Figure 18;

Figure 20 shows the truth table for Figure 18;

Figure 21 is a bit addition circuit for Summing Operation in Stage 4 in the Flowchart of Figure 5;

Figure 22 is the truth table for Figure 21;

Figure 23 is an equivalent truth table for Figure 22;

Figure 24 is the circuit for adding two 8-bit generalized binary numbers;

Figure 25 shows the subcircuits in Figure 24 in which Subcircuit A4-2XY11 is shown Figure 13, Subcircuit 2SCV-21 is shown in Figure 16, Subcircuit A4-2XY12 is shown in Figure 18 and Subcircuit A4-2XY13 is shown in Figure 21;

Figure 26 shows a subcircuit for determining the propagate, generate, and sum calculated by the adder of Figure 24 from the generalized format into Two's Complement binary string format;

Figure 27 is the logic circuit in the subcircuit of Figure 26;

Figure 28 is the truth table for the logic circuit of Figure 27;

Figure 29 is the truth table of Figure 27;

Figure 30 de-converts a 4-bit generalized binary number into a 5-bit classical binary format;

Figure 31 is a "Carry-Lookahead"-like circuit having modified generate, propagate and output of carry and sum;

Figure 32 is the truth table for bit sedimentation, Stage 3 of the Alternative Method;

Figure 33 is the logic circuit for Figure 32;

Figure 34 is the logic circuit for Figure 32;

Figure 35 is the truth table for Transformation AB of the Alternative Method;

Figure 36 is the logic circuit for Figure 35;

Figure 37 is the logic circuit for Figure 35;

Figure 38 is the truth table for bit addition of the Alternative Method;

Figure 39 is the logic circuit for Figure 38;

Figure 40 is the logic circuit for Figure 38;

Figure 41 is the circuit for adding two 8-bit generalized binary number using the Alternative Method;

Figure 42 shows the subcircuits in Figure 41 in which Subcircuit A4-2XY41 is shown Figure 34, Subcircuit A4-2XY42 is shown in Figure 37, Subcircuit A4-2XY43 is shown in Figure 40; and Subcircuit 2SCV-21 is shown in Figure 16;

Figure 43 is the truth table for an embodiment using only a Transformation A;

Figure 44 is a possible logic circuit for Figure 43;

Figure 45 is also a logic circuit for Figure 43; and

Figure 46 is the logic circuit for an adder that can be used instead of the one in Figure 21.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

The embodiment is founded on a proposed General Equivalence Principle and Generalised Binary Number Format, which have to be explained first before describing any embodiment in detail, a binary string Transformation called the Specific Equivalence Principle.

Equivalence Principle (General)

Any positive whole number with zero excluded (ie. natural number) can be expressed as the sum of two positive whole number components of smaller magnitude, provided that the whole number is not 1. However, if the component numbers may be integers of positive and negative numbers, then any positive whole number can be expressed as multiple different pairs of component integer numbers,

both of which may be positive or negative, or either one be positive and the other negative. The sum of two integers may be expressed as many sets of four component numbers that add up to equal sum.

Accordingly, given a pair of binary strings to be added up to find their sum, it is possible manipulate the zeros and nonzeros between the pair of binary strings so as to provide a different pair of binary strings but which add up to equal sum. The principle of picking different sets (not just a pair) of component numbers the sum of which is a constant number is the proposed General Equivalence Principle.

Generalised Binary Number Format and the process of Generalization

A proposed process of 'generalization' is used to re-express or re-cast a normal binary string representing a certain number into a set or a pair of binary strings that add up to the same number. The paired binary strings are called 'substrings' for distinction, and are specifically given the notations x and y this description. The value of each substring may be read like a typical binary string.

The x substring and y substring may be transformed into another pair of x substring and y substring that have different values but nevertheless add up to equal sum, by using transformations that are designed to observe the Equivalence Principle.

While it is possible in some embodiments that both the x substring and y substring of a pair are positive values, it is preferred that the x substring represents a positive value and the y substring represents a 'negative' value as this configuration provides

a larger range of values for more flexible manipulation. Hence, x substring – y substring = the value of the original binary string. However, it is also preferably that the value of the y substring is an absolute value, and that the value is to be subtracted from the x substring, rather than the y substring being a negative value. This is because a negative number in binary and in a processor environment is expressed in Two's Complement, and tends to have many nonzeros that are potential cause of carry-over values. The embodiment is able to deal with the many nonzeros, but it is not convenient for explaining the embodiment or to check for errors with the many nonzeros to deal with.

Accordingly, it would be a more accurate to refer to the y substring as subtrahend to the x substring as minuend. However, referring to 'minuends' and 'subtrahends' many times makes the description a little convoluted. Therefore, in the following paragraphs, a y substring has an absolute value but may be referred to as a 'negative' binary substring or a 'negative component' substring, the value of which is 'negative', and a nonzero bit in a y substring as a 'negative' bit, which is a little more intuitive to most readers. Conversely, an x substring may be referred to as a positive component substring, the value of which is positive, and a nonzero bit in an x substring as a positive bit.

'String' and 'number' are used interchangeably in this description when describing the binary.

An augend binary number and an addend binary number are provided, which are converted each into a generalised binary number, i.e. $(n_w)_1$ and $(n_w)_2$. Each generalised binary number comprises pair of substrings, x_1, y_1, and x_2, y_2. Generalisation of a n-bit binary string n_w can be presented in mathematical notations as follows:

$$n_w = n_w^+ + n_w^- = a_{n-1}^+ a_{n-2}^+ a_{n-3}^+ \ldots a_2^+ a_1^+ a_0^+ - a_{n-1}^- a_{n-2}^- a_{n-3}^- \ldots a_2^- a_1^- a_0^- = \begin{pmatrix} a_{n-1}^+ a_{n-2}^+ a_{n-3}^+ \ldots a_2^+ a_1^+ a_0^+ \\ a_{n-1}^- a_{n-2}^- a_{n-3}^- \ldots a_2^- a_1^- a_0^- \end{pmatrix} \quad (1)$$

where

$a_i^+ = \langle 0,1 \rangle$ and $a_i^- = \langle 0, \bar{1} \rangle$, in which $i = 0,1,...,n-1$; and

<, > indicates a set of possible values for the variable.

$\bar{1}$ is a shorthand for a nonzero bit in the y substring, consisting of a bar over a digit 1.

The weight of each bit in the binary strings is unity or a positive-integer power of 2. Hence, n_w can be expressed as:

$$n_w = \sum_{i=0}^{n-1} 2^i a_i^+ - \sum_{i=0}^{n-1} 2^i a_i^- = \sum_{i=0}^{n-1} 2^i \left(a_i^+ - a_i^- \right) \quad (2)$$

The domain of n_w covers a range from $-\left(2^n - 1\right)$ to $2^n - 1$.

An algorithm based on the above may be implemented by the circuit block shown as Figure 6, which takes in an 8-bit binary string and outputs a pair of x substring and y

substring, each in 8-bit. Inside the circuit block of Figure 6 is the logic circuit shown as Figure 7. When a number in the form of a binary string is fed into the logic circuit in of Figure 7, a replica of the binary string is made and provided as the x substring, to which is assigned an all-zero binary string of the same length for the y substring. The pairing completes the process of generalization. However, if the sign bit of the binary string is nonzero, the replica of the binary string for the x substring stops short of this sign bit. The nonzero sign bit is placed in the same bit position (he largest) in the otherwise all-zero y substring.

Therefore, a binary string 1000 0011 (-125 in decimal) is generalised into an x substring of 0000 0011 (3 in decimal) and a y substring of 1000 0000 (an absolute number 128 in decimal). As mentioned, the y substring is subtrahend to the x substring as minuend, i.e. x - y = $3 - 128 = -125$.

Upon the pairing, the original binary string has become “generalised” because the string’s value is no longer represented by a binary string but by one of many possible pairs of x and y substring as subtrahend that give and identical sum, in accordance to the General Equivalence Principle.

Preferably, the x substring and y substring are each allocated a string length longer than that of the original binary string by one bit. This extra bit allows the maximum value of the x substring to bigger by one binary order, such that the value of the original binary string falls easily within a large minuend value from which the value of y substring may be subtracting to give the value of the original binary string.

A “Specific” Equivalent Principle Conversion

A particularly useful conversion based on the Equivalence Principle is described in some detail in Box 1. It is easier to appreciate the use of the Generalised Binary Number Format by considering what is called in this description the Specific Equivalence Principle (SEP) conversion.

Box 1

The Specific Equivalence Principle in a generalized binary number system states that:

- Any substring of $m\ (\geq 1)$ 1’s can be replaced by an 1 (more significant) and an $\bar{1}$ (less significant) separated by m-1 zeroes.
- Similarly, any substring of $m(\geq 1)\ \bar{1}$’s can be replaced by an $\bar{1}$ (more significant) and an 1 (less significant) separated by m-1 zeroes.

The effect of the SEP conversion removes all adjacent nonzeros in a substring and can be applied separately to each bit position in parallel processes.

Taking the positive string as $x_i = \langle 0,1 \rangle$ and the negative string as $y_i = \langle 0,\bar{1} \rangle$, the Specific Equivalence Principle for a substring of m (≥ 1) 1’s can be expressed by

Equation 3, while that for a substring of m (≥ 1) $\overline{1}$'s can be expressed by Equation 4.

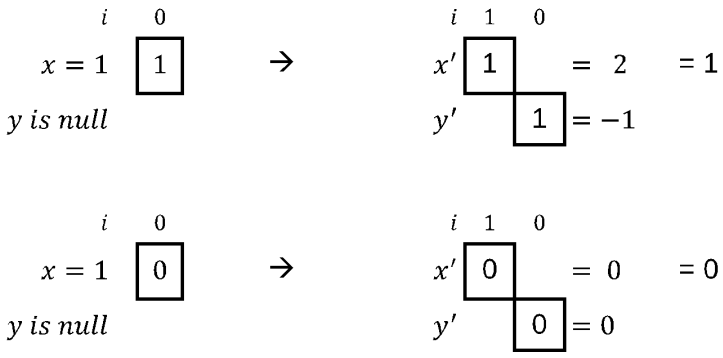
$$X_i = \overline{x}_i x_{i-1} \quad ; \quad Y_i = x_i \overline{x}_{i-1} \tag{3}$$

$$X_i = y_i \overline{y}_{i-1} \quad ; \quad Y_i = \overline{y}_i y_{i-1} \tag{4}$$

The top drawing in Figure 9 is the corresponding circuit block diagram 901 while the bottom drawing in Figure 9 is the underlying logic circuit unit 903 for SEP conversion of an x substring. Similarly, the top drawing in Figure 10 is the corresponding circuit block diagram while the bottom drawing in Figure 10 is the underlying logic circuit unit for SEP conversion of an y substring. The corresponding truth table is presented in Figure 11, where the left half of the table relates to conversion of a nonzero in the positive component x substring, and the right half of the truth table relates to conversion of the negative sign component y substring.

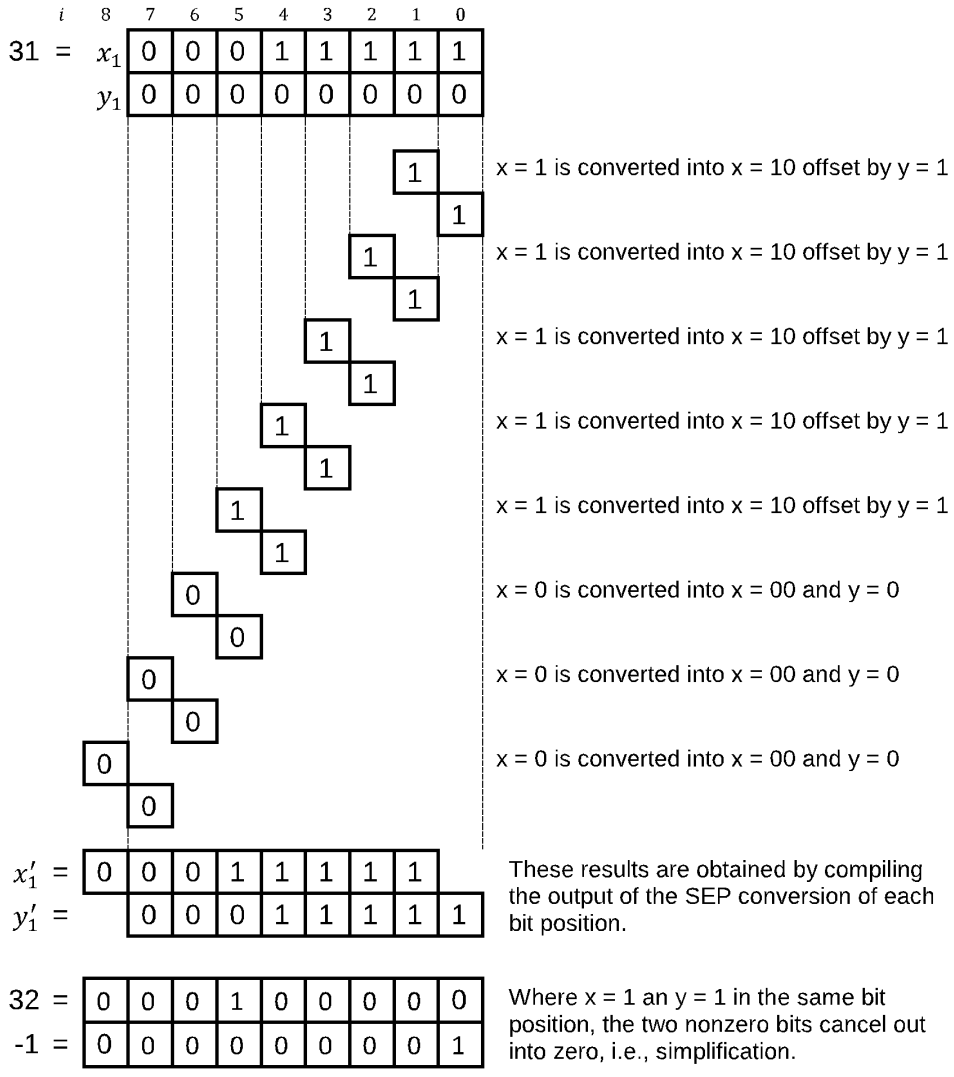
Box 2 below shows the SEP conversion, which is based on the possibility of re-expressing a positive bit having a nonzero value of 1 as a greater number offset with a smaller and negative number, such as $x - y = 2 - 1 = 1$, while observing the Equivalence Principle. For completeness, Box 2 shows that if the bit has zero value the output is zero at a higher order. Actually, this is determined passively from the active re-positioning of nonzero bits in the output.

Box 2



An example of applying the SEP conversion in Box 2 to a binary string is given in the following paragraphs. Taking as example the number 31, which in 8-bit binary format is 0001 1111, generalization re-expresses 0001 1111 as an x substring = 0001 1111, paired with an all-zero y substring = 0000 0000 as there is no sign bit. Box 3 below shows the SEP conversion shown in Box 2 applied to the x substring. The x substring is converted into 0010 0000 (32 in decimal) and the y substring is converted into 0000 0001 (1 in decimal). Therefore, $x - y$ in the decimal system is $32 - 1 = 31$, giving the same sum and observing the General Equivalence Principle. For convenience, these substrings may be written as 0010 000$\bar{1}$ in the recommended shorthand described in Equation 1.

Box 3

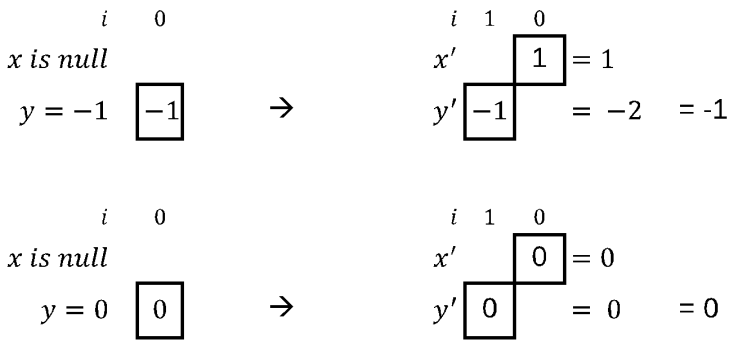


As described already in a preceding paragraph, it can be seen that the overall effect of the SEP conversion lies in the removal of all immediately adjacent nonzero bits in the x substring, but without changing the sum 31, i.e. 32-1. The SEP conversion must be applied consistently to every bit position in the x substring.

Using the shorthand $\bar{1}$ to denote a nonzero bit in the y substring, a pair of substrings x =0001 0000 and y = 0000 0001 may be written in a single line as 0001 000$\bar{1}$. There cannot be a 1 and a $\bar{1}$ in the same bit position as they cancel out and becomes zeros.

The SEP conversion may have to be applied also to every bit position the y substring too, although there is nothing to demonstrate in the example of an all-zero y substring, Box 4 below shows the SEP conversion in Box 2 expressed for the y substring.

Box 4



The example in Box 3 has a final step of simplifying the SEP conversion output, which is not desirable for it adds an additional step to the overall embodiment.

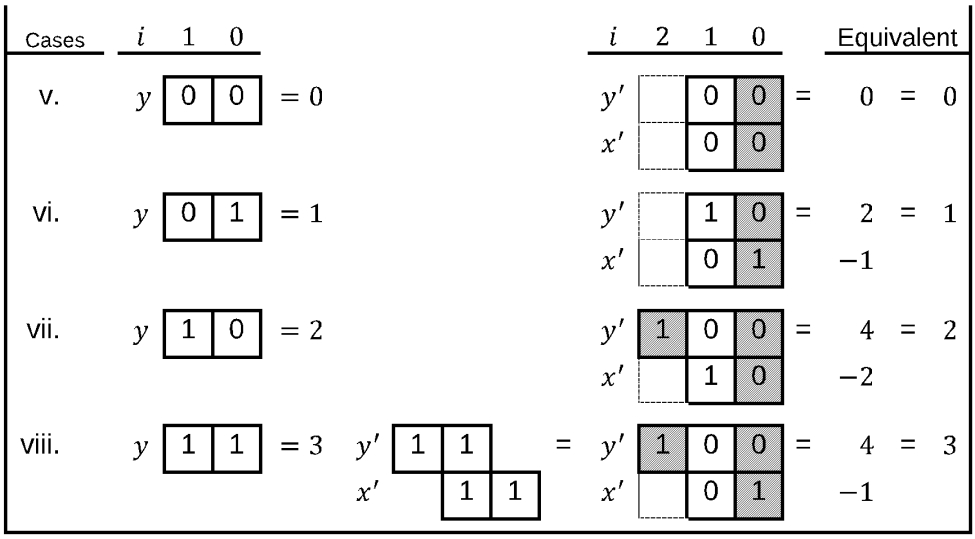
Box 5 below relates to a variation of the SEP conversion, which produces outputs from inputs according to the truth table in Figure 11 but without the need of a

simplification step after SEP conversion. The corresponding logic circuits are shown as Figure 9 and Figure 10.

While the SEP conversion in Box 2 and Box 4 reads a single bit for input and gives two bits for output, the SEP conversion in Box 5 reads two bits for input, one from the current position $i = 1$ and the other from the lower position $i = 0$. If there is no position $i = 0$ because $i = 1$ is already the smallest bit at the start of a binary string, a zero value is prescribed to the non-existent bit in position $i = 0$. Furthermore, although the result of the SEP conversion of Box 5 may include bits in more than one bit position, only the bits in position $i = 1$ are taken compiled into the truth table. To draw the reader's attention to this, SEP conversion output bits in position $i = 1$ are shown unshaded, while adjacent bits not included into the compilation of the truth table are shown shaded.

Box 5

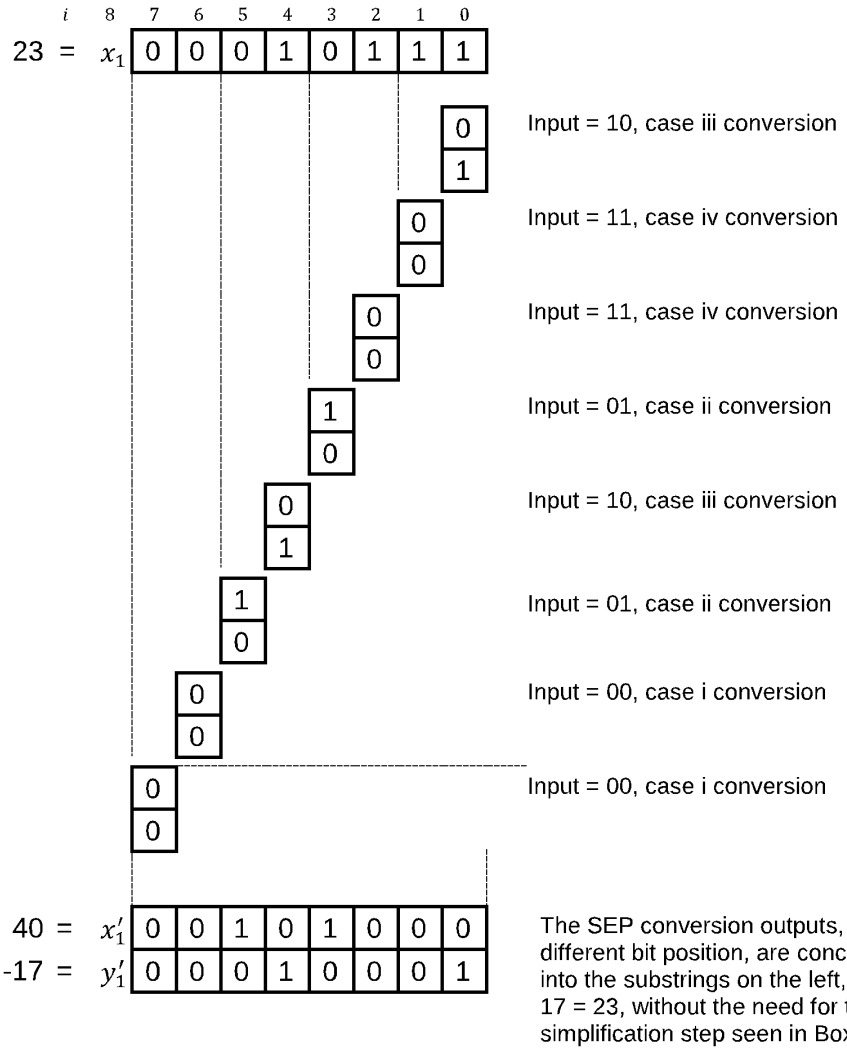
Cases	i 1 0		i 2 1 0		Equivalent
i.	x 0 0	$= 0$	x' 0 0 y' 0 0	$= 0$	$= 0$
ii.	x 0 1	$= 1$	x' 1 0 y' 0 1	$= 2$ $= -1$	$= 1$
iii.	x 1 0	$= 2$	x' 1 0 0 y' 1 0	$= 4$ $= -2$	$= 2$
iv.	x 1 1	$= 3$	x' 1 1 y' 1 1	$= x'$ 1 0 0 y' 0 1	$= 4$ $= -1$ $= 3$



Cases i to iv in Box 5 show SEP conversions of all the four possible combinations of two bits in the positive component x substring, i.e. 00, 01, 10, 11. The two bits are adjacent to each other in the same substring. Conversely, cases v to viii in Box 5 shows the SEP conversion of all the four possible combinations of two bits in the negative component y substring, i.e. 00, 01, 10, 11.

An example of applying the SEP conversion in Box 5 to a binary string is shown in Box 6. Taking as example the number 23, which in an 8-bit binary format is 0001 0111 and which generalizes into an x substring = 0001 0111 and a y substring = 0000 0000.

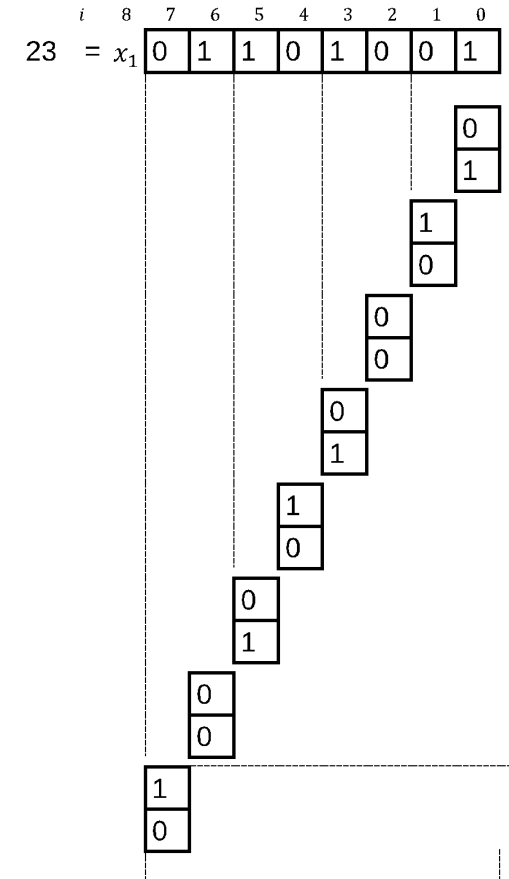
Box 6



Upon SEP conversion of each bit based on the transformation in Box 5, the x substring becomes 0010 1000 (40 in decimal), and the y substring becomes 0001 0001 (17 in decimal). The difference between $x - y$ in the decimal system is 40 – 17 = 23 (General Equivalence Principle observed). These substrings may be written as $001\bar{1}\ 100\bar{1}$ in the recommended shorthand.

As yet another example, consider the negative number, -23, which in Two's Complement 8-bit binary format is 1110 1001. In any processor, the most significant bit is always designated the sign bit, which is an industrial standard. According to Figure 7 and Figure 8, generalizing a negative value 1110 1001 gives an x substring = 0110 1001 (105 in decimal), which is a replica of the original binary string but without the sign bit; and gives a y substring = 1000 0000 (128 in decimal), which is an all zero binary string except for the sign bit.

Box 7



Input = 10, case iii SEP conversion

input = 01, case ii SEP conversion

input = 00, case i SEP conversion

input = 10, case iii SEP conversion

input = 01, case ii SEP conversion

input = 10, case iii SEP conversion

input = 11, case iv SEP conversion

input = 01, case ii SEP conversion

The SEP conversion outputs, each of a different bit position, are concatenated into the substrings on the left, giving 146-169 = 23, without the need for a default simplification step after SEP conversion..

However, simplification is still applied in other stages of the embodiment, so as to change the two nonzeros in position $i = 7$ into zeros and giving 18-41 = -23.

In the recommended shorthand, these substrings may be written as $\bar{1}110\ 1001$. The difference between $x - y$ in the decimal system is $105 - 128 = -23$.

As mentioned in Box 7 and as shown in the following paragraphs, there are often a need to simplify the substrings to remove a nonzero bit in the x substring which is in the same position as a nonzero bit in the y substring. Hence, it is important that the Box 5 SEP conversion is used rather than the Box 2 and Box 4 SEP conversion to reduce as many simplification steps as possible.

Accordingly, the above description explains how a number is converted from being a normal binary number into a generalised binary number, and how the component substrings are treated by SEP conversion into a form ready for subsequent stages.

A First Embodiment

Based on the information provided in the preceding paragraphs, embodiments may now be described which relate to methods and logic circuits that transform binary strings into a format which permits the transformed binary strings to be summed up by a separate adding operation for each bit position (also called bit order).

Embodiments include multi-bit adders each of which is constructed of a plurality of separate adder units, each adder unit being useable to add up bits of the same order independently of adder units operating on the bits in adjacent bit positions. There is no carry-over output lead or carry-over input lead connected the adder units.

Figure 5 is a flowchart of the stages according to one embodiment. Each stage may be executed by one or more logic circuits. The drawings of logic circuits corresponding to each stage are indicated next to the stage. The following is a brief description of the stages before further elaboration is given in subsequent paragraphs.

- In Stage 1, an augend and an addend are provided in the form of binary strings.

- In a Stage 2, the binary strings are each converted into a 'generalized binary number' according to a Generalised Binary Number Format, wherein each generalized binary number is composed of a set or a pair of binary strings. These will be called 'substrings' for distinction from regular binary strings.

- In a Stage 3, each of the two generalised binary numbers, $(n_w)_1$ and $(n_w)_2$, is converted into a different generalised binary number that have the same total value. However, the converted generalised numbers can be added up using parallel adding operations for each bit position in Stage 4. The conversion itself comprises three separate transformations, Transformation A, Simplification Transformation and Transformation B, each of which changes an aspect of the two generalised binary numbers, $(n_w)_1$ and $(n_w)_2$.

  Specifically, Transformation A transforms the generalized binary numbers according to the principle of addition and, in so doing, manifests and

accommodates the carry-over values pre-emptively of actual summation using separate adder units operating in parallel for different bit positions in Stage 4. This prevents the need for carry-over output and input leads between the separate adder units.

Simplification Transformation is then applied to remove redundant bits in the transformed generalized binary numbers produced by Transformation A. Generally but not absolutely, the Simplification Transformation is executed after every transformation.

Subsequently, Transformation B further transforms the generalized binary numbers produced in the preceding Transformation A and Simplification Transformation. Transformation B is based also on the principle of addition. Therefore, similar to Transformation A, Transformation B manifests and accommodates any potential carry-over values pre-emptively of the actual summation in the Stage 4.

Transformation A and Transformation B are different transformations, however, as will be explained in greater detail in following paragraphs.

- In a Stage 4, the transformed generalized numbers produced in the Stage 3 are added up, in a Summing Operation that sums up the generalized binary numbers pre-treated by Transformation A (followed by Simplification Transformation) and Transformation B.

The summation is done in separate, independent adding operations, each addition operation for only the bits in the same bit position. As mentioned, the separate adding operations can be run in parallel because there will be no carry-over from any lower bit position to a higher bit position, since any required carry-overs have been pre-emptively carried out or accommodated by the transformations in the Stage 3. The sum, however, is also a generalised binary number.

- In Stage 5, a decision is made whether to add a further number to the sum produced in Stage 4. If so, it goes directly to Stage 7 where the sum produced in Stage 4 is then simplified using the same process in Simplification Transformation. Subsequently, in a Stage 1a, a new addend in binary form is provided. In Stage 2a, the new addend is generalised and becomes converted into the general binary format. The described Stage 3 is then repeated with the new addend in Generalised Binary Number Format being added to the sum of the previous addition cycle.

- If it is decided no further number is to be added in Stage 5, the sum may be de-converted in Stage 6 from a generalized binary number into a normal binary string. Hence, a basic embodiment may comprise just Stage 1 to Stage 6.

The following paragraphs describe the process of adding up numbers 13 and 14 according to the flowchart in Figure 5. Stage 1 is straightforward and is fulfilled upon choosing 13 as the augend and 14 the addend.

In Stage 2, each of the addend and augend binary strings is converted or "generalized" into a generalised binary number. Any generalised binary number comprises a set or a pair of two binary strings that represent the number or value of the binary string. As neither 13 nor 14 is a negative number, the sign bit is 0. Therefore, the binary string of 13 is generalised by providing a replica of the binary string as x_1 substring, and paired with an all-zero y_1 substring of the same length. Similarly, the binary string of 14 is generalised by providing a replica of the binary string as x_2 substring, and paired with an all-zero y_2 substring of the same length. These are illustrated below in Box 8.

Box 8

									i	8	7	6	5	4	3	2	1
13 =	0	0	0	0	1	1	0	1	= x_1	0	0	0	0	1	1	0	1
									y_1	0	0	0	0	0	0	0	0
14 =	0	0	0	0	1	1	1	0	= x_2	0	0	0	0	1	1	1	0
									y_2	0	0	0	0	0	0	0	0

Stage 3 is next in the flowchart, and relates to transformations of the two generalised binary numbers Transformation A, with subsequent Simplification, and Transformation B.

Transformation A, with subsequent Simplification, and Transformation B pre-emptively process carry-overs that would have triggered carry-overs and cause the

ripple effect in the prior art addition methods. The following paragraphs describe transformations or pre-treatments of the generalised binary strings of 13 and 14.

Transformation A

Transformation A relates to the generalised augend $(n_w)_1$ and generalised addend $(n_w)_2$ according to the following equations.

$$(y'_i)_1 = (x_i)_1(\overline{x}_i)_2 + (\overline{x}_i)_1(x_i)_2 \quad (5)$$

$$(x'_{i+1})_1 = (x_i)_1 + (x_i)_2 \quad (6)$$

for the positive components of $(n_w)_1$ and $(n_w)_2$; and

$$(x'_i)_2 = (y_i)_1(\overline{y}_i)_2 + (\overline{y}_i)_1(y_i)_2 \quad (7)$$

$$(y'_{i+1})_2 = (y_i)_1 + (y_i)_2 \quad (8)$$

for the negative components of $(n_w)_1$ and $(n_w)_2$.

Figure 13 (Subcircuit A4-2XY11) is a drawing of a corresponding sub-circuit which is constructed of the unit logic circuit shown as Figure 14 (Subcircuit A4-2X11). Figure 15 is the corresponding truth table of Transformation A.

The derivation of the truth table in Figure 15 is shown in Box 9. As mentioned in a preceding paragraph, there are only four permutations to two bits, i.e. 00, 10, 01, 11. Box 9 shows the derivation of the cases i, ii, iii, iv which relate to the two positive component substrings x_1 and x_2, which output a new pair of x'_1 and y'_1 substrings.

Conversely, cases v, vi, vii, $viii$ relate to the two negative component substrings y_1 and y_2, and output a new pair of x_2' and y_2' substrings.

Box 9

Derivation of Transformation A (positive components)

Case	(a) Input	(b) Sum	SEP	(c) Output	(d) Transformation
i.	i: x_1 0, x_2 0 (+)	= 0		$i+1$ | i: x_1' 0, y_1' 0	$0 - 0 = 0$
ii.	x_1 0, x_2 1 (+)	= 1	SEP →	x_1' 1, y_1' 1	$2 - 1 = 1$
iii.	x_1 1, x_2 0 (+)	= 1	SEP →	x_1' 1, y_1' 1	$2 - 1 = 1$
iv.	x_1 1, x_2 1 (+)	= 1 0		x_1' 1, y_1' 0	$2 - 0 = 2$

Column (a) shows the values of two input bits, from the substrings of the same sign component. Column (b) shows the sum of the two input bits. Column (c) is output base on the sum in column (b). Column (d) shows that the Equivalence Principle has been observed.

The effect of the Transformation A on every bit ensures that, in every case, the bits are “moved up” by an order while observing the Equivalence Principle. This

movement would make way for any carry-over from the lower position with no change in the original total value.

Only case iv outputs a greater value due to the sum of two nonzero input bits, which requires a carry-over into the higher bit position $i + 1$ in output x_1' substring, i.e. $1 + 1 = 10$. This leaves the original bit position i with a zero value so that any similar carry-over from the lower bit position value may be received and accommodated without need to propagate the carry-over further.

Case i arises when both input bits are zero and give a zero output bit, and the transformation simply moves the output zero bit by one greater bit position in the substring of the same sign component. This leaves the original bit position i with a zero value so that any similar carry-over from the lower bit position value may be received and accommodated without need to propagate the carry-over further.

Cases ii and iii arise when there is only one nonzero bit in position i among the x substrings. The SEP conversion is applied to the sum of the bits in this position, which is actually the value of the sole nonzero bit. The SEP conversion returns a nonzero bit in the higher bit position $i + 1$ but offset by a nonzero bit in the original position i and in the opposite component sign. This is already explained in Box 2, i.e. $1 = 1\,\bar{1}$. In this way, the original bit position i is left with a zero value.

The cases ii, iii, iv will move any occupying nonzero to the next higher bit position, leaving the position free to accommodate a carry-over if there is a case iv in the lower bit position, extinguishing any further propagation of the carry-over.

A carry-over occurs whenever two nonzero bits of the same sign component are added, such as in $(x_1 + x_2) = 1 + 1 = 10$, or in $(y_1 + y_2) = 1 + 1 = 10$.

As a the y substring represents values that is meant to be subtracted from a corresponding x substring and therefore represents a negative value, $(x_1 - y_2)$ or $(x_2 - y_1)$ only give a smaller value and will not trigger any carry-over. Therefore, Transformation A only applies to same sign component substrings in order to manifest potential carry-overs.

Figure 12 is provided as a visual guide for the reader to follow which of the substrings are used in the transformations in this embodiment. Figure 12 shows that Transformation A is applied to substrings x_1 and x_2 to produce x_1' and y_1'. At the same time, if need be or in a real work situation, Transformation A is also applied to substrings y_1 and y_2 to produce x_2' and y_2'.

Box 10 has similar content as Box 9, and is provided for completeness and by showing how Transformation A is applied to the substring of the opposite component sign.

Box 10

Derivation of Transformation A (negative components)

Case	(a) Input	(b) Sum	SEP	(c) Output	(d) Transformation
v.	i: y_1 0, y_2 0	= 0		$i+1$, i: x_2' 0, y_2' 0	$0 - 0 = 0$
vi.	i: y_1 0, y_2 $\bar{1}$	= -1	SEP →	$i+1$, i: x_2' 1, y_2' $\bar{1}$	$1 - 2 = -1$
vii.	i: y_1 $\bar{1}$, y_2 0	= -1	SEP →	$i+1$, i: x_2' 1, y_2' $\bar{1}$	$1 - 2 = -1$
viii.	i: y_1 $\bar{1}$, y_2 $\bar{1}$	= -1 0		$i+1$, i: x_2' 0, y_2' $\bar{1}$	$0 - 2 = -2$

Note: -1 is only illustrative as the y substring has only absolute value of 1, as explained in a preceding paragraph.

The derivation of cases v, vi, vii, $viii$ for the y_1 and y_2 substrings is similar to that of cases i, ii, iii, iv in Box 9 for the x_1 and x_2 substrings, and cases v, vi, vii, $viii$ output a new pair of x_2' and y_2' substrings.

Only case $viii$ outputs a greater value due to the sum of two (negative) nonzero input bits, which requires a carry-over into the higher bit position $i + 1$ in output y_1' substring, i.e. $1 + 1 = 10$. This leaves the original bit position i with zero value and available to accept a carry-over value from the lower bit position, preventing propagation of the carry-over.

Case v arises when both input bits have zero values, Transformation A moves the output bit having zero value by one position in the output substring of the same sign component. This leaves the original bit position i with a zero value so that any carry-over from the lower bit position value may be received and accommodated, stemming the need to propagate the carry-over further.

Cases vii and $viii$ are the opposite sign component corresponding to cases ii and iii, and arise when there is only one nonzero bit in position i among the y substrings, the sum of which should be a single nonzero output in same bit position the output y_1' substring. The SEP conversion transformation for a single nonzero bit is a nonzero bit in the next higher order in position $i + 1$ and of the same sign component, offset by a nonzero bit of the opposite sign component in the same position i, which observes the General Equivalence Principle, i.e. $\bar{1} = \bar{1}1$.

Accordingly, all the cases in Box 9 and Box 10 comprises a method of making available a bit position i in one of two binary strings to store a carry-over value, comprising the step(s) of: replacing a nonzero with a zero in the bit position i, the nonzero in the bit position representing a value; and providing at least one nonzero in a different bit position in the two binary strings representing an equal value.

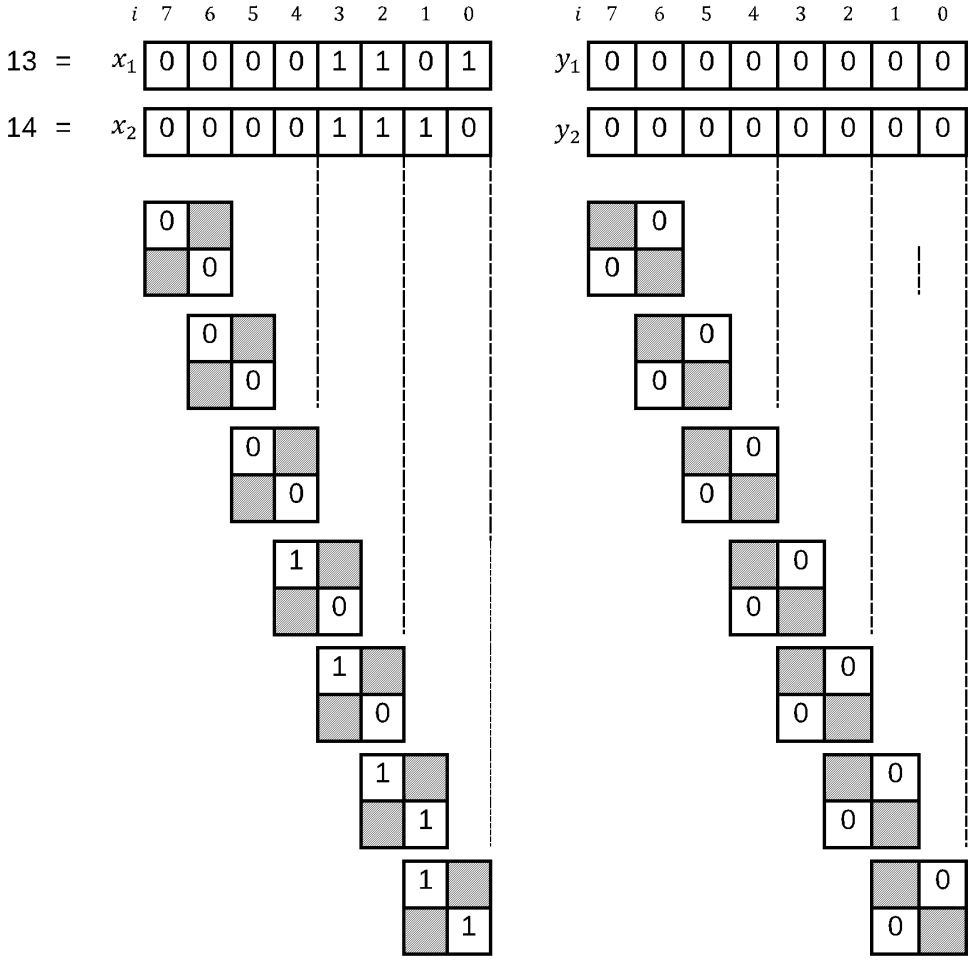
Although Transformation A is based on adding up the values of the relevant bits and even reveals potential carry-over values due to the addition, Transformation A is not really an adding process. This is because two input values becomes two output

values upon Transformation A, no term has been extinguished. In a real adding up process, two input values becomes one output value.

Continuing with the example of adding 14 to 13, Box 11 shows Transformation A applied to the output of Box 8.

As shown in Box 11, for each position i, Transformation A may be applied to bits by the bit position. The outputs of Transformation A may be concatenated to produce the pairs of output substrings.

Box 11



The outputs of Transformation A for the different bit positions are concatenated to produce the following two pairs of substrings, having the combined value of $30 - 3 = 27$.

30 = x'_1 0 0 0 1 1 1 1 0 x'_2 0 0 0 0 0 0 0 0

-3 = y'_1 0 0 0 0 0 0 1 1 y'_2 0 0 0 0 0 0 0 0

Simplification Transformation

Simplification Transformation or just Simplification has been mentioned in preceding paragraphs, and is also applied to the output of Transformation A. In fact, the Simplification Transformation is applied after most transformation if deemed necessary. The effect is to eradicate unnecessary nonzeros bits.

In the Simplification Transformation, nonzeros bits found in identical bit position in a pair of x and y substrings cancel out and may be replaced with zeros. The "simplification" process is only applied to paired x and y substrings, so that the General Equivalence Principle is observed. In other words, where bits in the same position in a pair of substrings are both nonzero, i.e. $x' = 1$ and $y' = 1$, where the interests in value of these bits are replaced by $x'' = 0$ and $y'' = 0$ with no consequential change in the value represented by the substrings.

The following equations describe the process of the Simplification Transformation.

$$\left(x_i''\right)_1 = \left(x_i'\right)_1 \left(\overline{y_i'}\right)_1 \tag{9}$$

$$\left(y_i''\right)_1 = \left(\overline{x_i'}\right)_1 \left(y_i'\right)_1 \tag{10}$$

and

$$\left(x_i''\right)_2 = \left(x_i'\right)_2 \left(\overline{y_i'}\right)_2 \tag{11}$$

$$\left(y_i''\right)_2 = \left(\overline{x_i'}\right)_2 \left(y_i'\right)_2 \tag{12}$$

The corresponding truth table is shown as Figure 17. A digital circuit corresponding to the truth table is shown as Figure 16 (Subcircuit 2SCV-21).

Continuing with the example of adding 14 to 13, Box 12 shows the output of Simplification Transformation applied to each bit position. The two nonzeros in x'_1 and y'_1 in position $i = 1$ is replaced by zeros.

Box 12

			i	7	6	5	4	3	2	1	0		i	7	6	5	4	3	2	1	0	
30	=	x'_1		0	0	0	1	1	1	1	0	x'_2		0	0	0	0	0	0	0	0	=0
-3	=	y'_1		0	0	0	0	0	0	1	1	y'_2		0	0	0	0	0	0	0	0	=0

Simplification of the above substring pairs give the substring pairs below.

28	=	x''_1		0	0	0	1	1	1	0	0	x''_2		0	0	0	0	0	0	0	0	=0
-1	=	y''_1		0	0	0	0	0	0	0	1	y''_2		0	0	0	0	0	0	0	0	=0

Figure 12 shows that a Simplification Transformation is a matter between any paired x and y substrings.

Transformation B

Transformation B is applied to the two pairs of substrings that has been simplified in Box 12, i.e. x''_1, y''_1 and x''_2, y''_2. Transformation B is based on summing up the values

of two bits, just like Transformation A. However, Transformation B gives a different output.

Anyway, the effect of Transformation B includes obtaining the sum of two bits, relegating existing nonzero bits to only one of the two outputs. Transformation B applies the following equations:

$$\left(x_i'''\right)_1 = \left(x_i''\right)_1 \left(\overline{x_i''}\right)_2 + \left(\overline{x_i''}\right)_1 \left(x_i''\right)_2 \quad (13)$$

$$\left(x_{i+1}'''\right)_2 = \left(x_i''\right)_1 \left(x_i''\right)_2 \quad (14)$$

as between the positive component x substrings; and

$$\left(y_i'''\right)_1 = \left(y_i''\right)_1 \left(\overline{y_i''}\right)_2 + \left(\overline{y_i''}\right)_1 \left(y_i''\right)_2 \quad (15)$$

$$\left(y_{i+1}'''\right)_2 = \left(y_i''\right)_1 \left(y_i''\right)_2 \quad (16)$$

as between the negative component y substrings.

Figure 19 (Subcircuit A4-2X12) is an underlying logic circuit unit corresponding to equations 13, 14, 15 and 16. Figure 20 is the truth table of the logic circuit of Figure 19. Figure 18 (Subcircuit A4-2XY12) is the drawing of a subcircuit which contains the logic circuit of Figure 19.

The derivation of the truth table in Figure 20 is shown in Box 13. There are only four permutations possible with any two-bit input, i.e. 00, 10, 01, and 11. Including the “negative” y substrings, therefore, there are just eight transformation cases, i.e.

cases 1, 2, 3, 4 , for inputs x_1'' and x_2'' substrings which output x_1''' and x_2''' substrings; and cases 5, 6, 7, 8 for inputs y_1'' and y_2'' substrings which output y_1''' and y_2''' substrings.

Box 13

Derivation of Transformation B (positive components)

Case	(a) Input	Sum (bin)	(b) Output	(c) Transformation
	i		$i+1$ i	
1)	x_1'' 0 + x_2'' 0	= 0	x_2''' 0 x_1''' 0	$0+0=0$
2)	x_1'' 0 + x_2'' 1	= 1	x_2''' 0 x_1''' 1	$0+1=1$
3)	x_1'' 1 + x_2'' 0	= 1	x_2''' 0 x_1''' 1	$1+0=1$
4)	x_1'' 1 + x_2'' 1	= 10	x_2''' 1 x_1''' 0	$1+1=10$

The effects of Transformation B include adding up the bits in the same bit position in the substrings of the same sign component. Also, Transformation B streams all nonzero values to one of the same sign component substrings, so as to keep the other one of the same sign component substrings has only zero bits and available for accepting any carry-over value from the lower bit position.

In Box 13, case 4 is the only case that give rise to a carry-over, due to adding up two nonzeros in x substrings in position i. The two nonzeros bits are added up and become a single nonzero bit in a position of higher order $i + 1$ in the output substring x_2'''. This leaves the original bit position i with a zero value so that any similar carry-over from the lower bit position value may be received and accommodated without need to propagate the carry-over further.

Cases 2 and 3 arise where there is only one nonzero bit in position i in the x substrings. It does not matter which input x substrings is the nonzero bit, the nonzero bit is relegated into the same output substring x_1'''. The output bit position is the same as the value is unchanged. This leaves the position i in the other output substring x_2''' available for storing a carry-over value from the lower order, $i - 1$, without need to propagate the carry-over further. The output of case 4 directed to the other output substring x_2''' will not meet an occupied bit.

To be precise, however, only case 2 involves an actual transformation in repositioning a nonzero bit into the other x substring. Case 3 just translate the nonzero bit into the same position.

Case 1 arises when there are only zero bits in position i in both the same component substrings. The memory pointer for the zero value bit is just moved by one position in the output substring in x_2'''.

Box 14

Derivation of Transformation B (negative components)

Case	(a) Input	Sum (bin)	(b) Output	(c) Transformation
	i		$i+1$ i	
5)	y_1'' 0 + y_2'' 0	= 0	y_2''' 0 y_1''' 0	$0+0=0$
6)	y_1'' 0 + y_2'' $\bar{1}$	= -1	y_2''' 0 y_1''' $\bar{1}$	$0-1=-1$
7)	y_1'' $\bar{1}$ + y_2'' 0	= -1	y_2''' 0 y_1''' $\bar{1}$	$-1+0=-1$
8)	y_1'' $\bar{1}$ + y_2'' $\bar{1}$	= -10	y_2''' $\bar{1}$ y_1''' 0	$-1-1=-10$

In Box 14 above, the cases are mirrored from the cases in Box 13. Case 8 is the only case that gives a carry-over value into position $i+1$ in substring y_2''', due to the sum of two nonzero input bits, each from one of the two y substrings but both in position i. This vacates the original bit position, leaving with it a zero value, which is capable of accepting any carry-over from the lower bit position $i-1$.

Cases 6 and 7 arise where there is only one nonzero bit in position i between the two input y substrings. The nonzero bit is relegated to the same position i in one of the output substrings y_1''', ensuring that the other output substring y_2''' is unoccupied

by any nonzero value and able to take in any carry-over value without the need to propagate the carry-over further to a higher bit position.

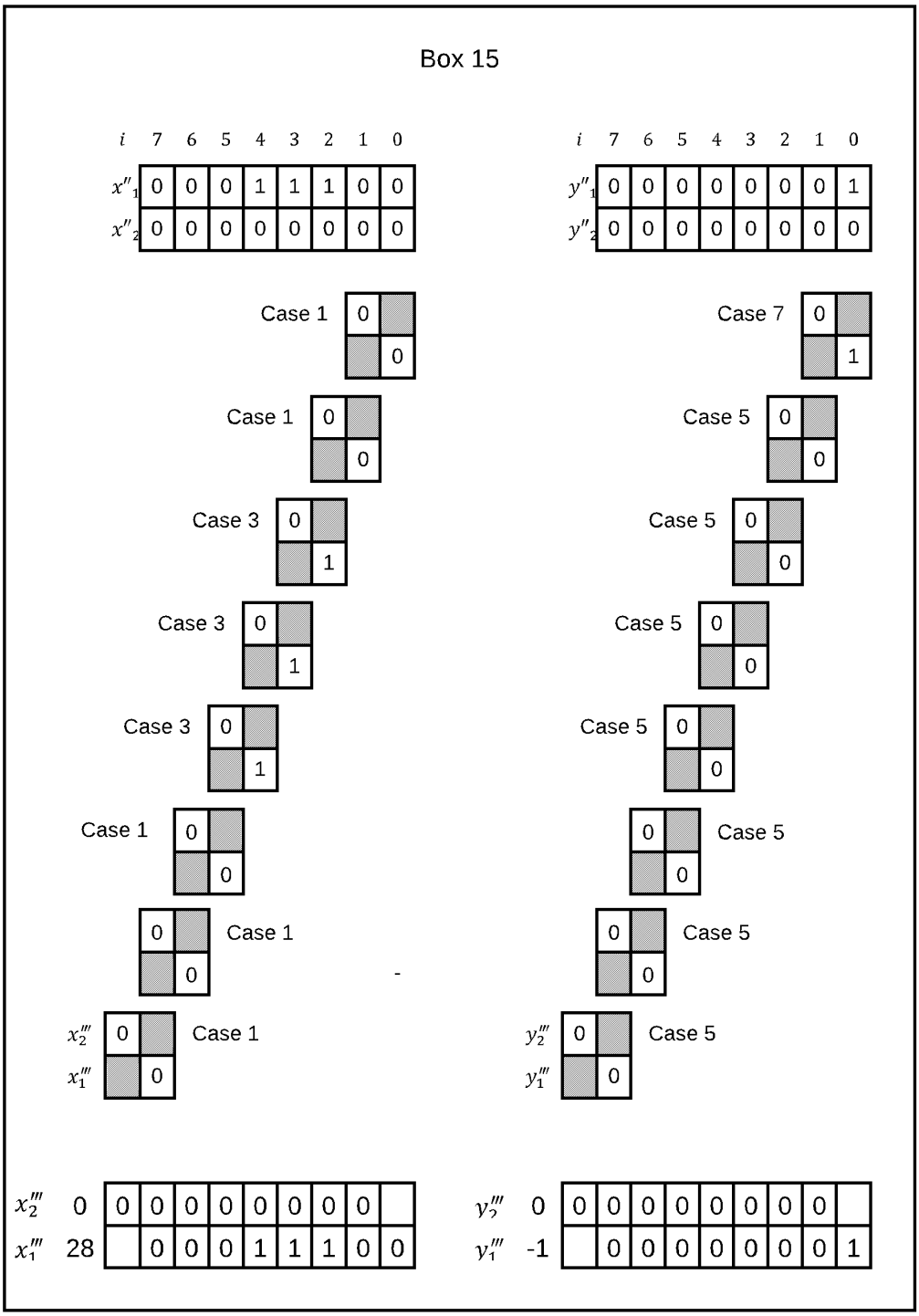
Case 5 arises when there are only zero bits in position i in both the y substrings, and the memory pointer for a zero value bit is moved by one greater bit position in the output substring y_2'''.

Accordingly, Transformation B comprises a method of making available a bit position i in one of two binary strings to store a carry-over value, comprising the step(s) of: replacing a nonzero with a zero in the bit position i, the nonzero in the bit position representing a value; and providing at least one nonzero in a different bit position in the two binary strings representing an equal value.

Figure 12 shows that Transformation B is applied only to the same sign component substrings, i.e. between x substrings only, or between y substrings only, and also returns two same sign component substrings.

With the application of Transformation A, with subsequent Simplification, and Transformation B, all carry-overs will have been manifested and pre-emptively accommodated.

Continuing with the example of adding 13 to 14, the substrings on the top of Box 15 below are the simplified output in Box 12. The Transformation B process applied to bits are shown in Box 15.


Box 15
i 7 6 5 4 3 2 1 0
x''1 0 0 0 1 1 1 0 0
x''2 0 0 0 0 0 0 0 0
i 7 6 5 4 3 2 1 0
y''1 0 0 0 0 0 0 0 1
y''2 0 0 0 0 0 0 0 0
Case 1
Case 1
Case 3
Case 3
Case 3
Case 1
Case 1
x'''2
x'''1
Case 1
Case 7
Case 5
Case 5
Case 5
Case 5
Case 5
Case 5
Case 5
y'''2
y'''1
Case 5
x'''2 0 0 0 0 0 0 0 0 0
x'''1 28 0 0 0 1 1 1 0 0
y'''2 0 0 0 0 0 0 0 0 0
y'''1 -1 0 0 0 0 0 0 0 1

Parallel Summing Operations

Stage 4 relates to the Summing Operation, in which the substrings produced in the preceding Stage 3 are summed up using a multi-bit adder. The sum bits X and Y are produced from the same positive and negative component, respectively, as

$$X_i = \left(x_i'''\right)_1 + \left(x_i'''\right)_2 \tag{17}$$

$$Y_i = \left(y_i'''\right)_1 + \left(y_i'''\right)_2 \tag{18}$$

Stage 4 is not a transformation but an actual addition operation, as there is just one output number (comprising X and Y substrings) produced from two input numbers (comprising substrings x_1''', y_1''' and x_2''', y_2'''). The adder recombines the input bits of x_1, y_1, and x_2, y_2 to produce the value of summed output bits X and Y.

The logic circuit of the adder is shown in Figure 21 (Subcircuit A4-2XY13) and the corresponding truth table in Figure 22 (with an equivalent truth table in Figure 23). Note that Figure 22 uses a shorthand notation in which the combined pair of substrings x and y is expressed as Z. Note that this is not the same shorthand for indicating a nonzero bit in the y substring as "$\overline{1}$".

Figure 21 shows that the adder (Subcircuit A4-2XY13) in this case is just an OR gate for the same sign component input. There is no need to consider a carry-over from the lower bit position or to pass on a carry-over to the higher bit position, as Transformation A with Simplification Transformation and followed by Transformation B has prevented the situation where both bits from the two substrings of the same

sign component are nonzero. For the same sign component substrings, only the following input cases will occur, 0+0, 1+0 or 0+1. As 1+1 does not occur, the summation can be done merely by compiling the nonzero bits in the substrings of the same sign component.

In this way, the embodiment enables independent, parallel summation for each bit position, giving a final pair of X and Y substrings, and there is no need to wait for propagation of carry-over from the lower order bits. The corresponding truth table is shown in Figure 22.

However, if the sum X and Y are to be used in further addition, a Simplification Transformation is applied to remove redundant nonzeros, and this is expressed in equivalent truth table in Figure 23.

Figure 12 shows that all four fully transformed substrings x_1''', y_1''' and x_2''', y_2''' are used to produce the sum substrings X, Y.

Continuing with the example of adding 13 to 14, the substrings at the bottom of Box 15 is reproduce in Box 16 below. The adder in Stage 4 produces the output as shown in Box 16.

Box 16

i	7	6	5	4	3	2	1	0
x_2'''	0	0	0	0	0	0	0	0
x_1'''	0	0	0	1	1	1	0	0
X	0	0	0	1	1	1	0	0

i	7	6	5	4	3	2	1	0
y_2'''	0	0	0	0	0	0	0	0
y_1'''	0	0	0	0	0	0	0	1
Y	0	0	0	0	0	0	0	1

For each bit position i added up the values of the bits; and all the adding in the different positions are done at the same time as there is no carry-over required in any downstream addition to wait for the carry-over input.

However, a Simplification Transformation is preferably applied to X and Y, especially if X and Y are to be recycled for further addition to another number.

Overview of First Embodiment

Figure 24 is a subcircuit block for adding two 8-bit generalized binary numbers, assembled from the logic circuits described in preceding paragraphs. Figure 25 show the underlying arrangement of the adder logic circuits inside the subcircuit block of Figure 24. The different subcircuits in Figure 25 can be seen in larger size in Figure 13 for Subcircuit A4-2XY11, in Figure 16 for Subcircuit 2SCV-21, in Figure 18 for Subcircuit A4-2XY12, and in Figure 21 for Subcircuit A4-2XY13.

Box 17 shows another example which illustrates the embodiment by showing the steps leading to the summation of $\left(\tilde{n}_w\right)_1 = 10\bar{1}1\bar{1}101$ (a generalized binary number

representing decimal 109) and $(\tilde{n}_w)_2 = 0100\bar{1}10\bar{1}$ (a generalized binary number representing decimal 59) using the adder in Figure 24. The sum is $N_w = 1\bar{1}1\bar{1}01000$ (a generalized binary number representing decimal 168).

Box 17		
Generalization of augend $(\tilde{n}_w)_1$ and addend $(\tilde{n}_w)_2$	$(\tilde{n}_w)_1^+ = (x_i)_1$	1 0 0 1 0 1 0 1
	$(\tilde{n}_w)_2^+ = (x_i)_2$	0 1 0 0 0 1 0 0
	$(\tilde{n}_w)_2^- = (y_i)_2$	0 0 0 0 $\bar{1}$ 0 0 $\bar{1}$
	$(\tilde{n}_w)_1^- = (y_i)_1$	0 0 $\bar{1}$ 0 $\bar{1}$ 0 0 0
Transformation A	$(x'_{i+1})_1 = (x_i)_1 + (x_i)_2$	1 1 0 1 0 1 0 1 0
	$(y'_i)_1 = (x_i)_1(\bar{x}_i)_2 + (\bar{x}_i)_1(x_i)_2$	0 $\bar{1}$ $\bar{1}$ 0 $\bar{1}$ 0 0 0 $\bar{1}$
	$(x'_i)_2 = (y_i)_1(\bar{y}_i)_2 + (\bar{y}_i)_1(y_i)_2$	0 0 0 1 0 0 0 0 1
	$(y'_{i+1})_2 = (y_i)_1 + (y_i)_2$	0 0 $\bar{1}$ 0 $\bar{1}$ 0 0 $\bar{1}$ 0
Simplification	$(x''_i)_1 = (x'_i)_1(\bar{y}'_i)_1$	1 0 0 1 0 1 0 1 0
	$(y''_i)_1 = (\bar{x}'_i)_1(y'_i)_1$	0 0 $\bar{1}$ 0 $\bar{1}$ 0 0 0 $\bar{1}$
	$(x''_i)_2 = (x'_i)_2(\bar{y}'_i)_2$	0 0 0 1 0 0 0 0 1
	$(y''_i)_2 = (\bar{x}'_i)_2(y'_i)_2$	0 0 $\bar{1}$ 0 $\bar{1}$ 0 0 $\bar{1}$ 0
Transformation B	$(x'''_i)_1 = (x''_i)_1(\bar{x}''_i)_2 + (\bar{x}''_i)_1(x''_i)_2$	1 0 0 0 0 1 0 1 1
	$(x'''_{i+1})_2 = (x''_i)_1(x''_i)_2$	0 0 0 1 0 0 0 0 0 0
	$(y'''_i)_1 = (y''_i)_1(\bar{y}''_i)_2 + (\bar{y}''_i)_1(y''_i)_2$	0 0 0 0 0 0 0 $\bar{1}$ $\bar{1}$
	$(y'''_{i+1})_2 = (y''_i)_1(y''_i)_2$	0 0 $\bar{1}$ 0 $\bar{1}$ 0 0 0 0 0
Parallel Bit Addition Operations	$X_i = (x'''_i)_1(\bar{y}'''_i)_1(\bar{y}'''_i)_2 + (\bar{x}'''_i)_1(\bar{y}'''_i)_1(x'''_i)_2$	0 1 0 1 0 0 1 0 0 0
	$Y_i = (\bar{x}'''_i)_1(y'''_i)_1(\bar{x}'''_i)_2 + (\bar{x}'''_i)_1(\bar{y}'''_i)_1(y'''_i)_2$	0 0 $\bar{1}$ 0 $\bar{1}$ 0 0 0 0 0
	N_w	0 1 $\bar{1}$ 1 $\bar{1}$ 0 1 0 0 0

More Adding Cycles

If there are more numbers to be added up, Stage 5 directs the sum, which is still in the generalised format, directly to Stage 7 for simplification first as shown in Figure 5, before being re-used in the next addition cycle. The new number to be added is provided in Stage 1a and processed in Stage 2a into a generalised number format and fed into Stage 3.

If there are no more numbers to be added up, Stage 5 directs the sum to Stage 6 as shown in Figure 5 where de-conversion of the final generalized number, with x substring and y substring, is carried out by using a "Carry-Look-Ahead"-like deconversion circuit.

Deconversion

In Stage 6, the sum obtained in Stage 4 is deconverted from a generalised binary number expressed in two substrings into a regular single binary string, as shown in Figure 5.

Before commencement of deconversion, however, the Simplification Transformation is again applied to replace all redundant nonzeros in both the X and Y substrings.

The simplified substrings are combined into a single binary string by using a de-conversion subcircuit. In mathematical expression, it can be said to de-convert a substring of (n-1)-bit Z_w into a n-bit Two's Complement binary string $z_w^{2+}\left(\equiv n_w^{2+}\right)$.

The deconversion can only be done as a serial process of adding up the X and Y substrings, starting from the lowest bit. This is because bits in each bit position are deconvertible only with regard to values carried over from lower order positions. In other words, the embodiments postpone this serial operation to being done only after all the addends have been summed up in the generalized number format, so that there is only one serial addition process for millions of addends.

However, to make the addition of X and Y substrings faster, the addition may be done using any conventional methods for making a serial addition faster. For example, the Carry Lookahead can be modified by replacing the generate (g_i) and the propagate (p_i) signals with the following.

$$p_i = \overline{x}_i \overline{y}_i \quad (19)$$

$$g_i = y_i \quad (20)$$

The "final summation" can be expressed by the following equations:

$$X_i = t_i = \overline{c}_i x_i + c_i p_i + \overline{c}_i y_i \text{ for } i = 0,1,2,\cdots,(n-1) \quad (21)$$

$$Y_n = c_n \quad (22)$$

when a “carry” for each bit has been looked-ahead, where $c_i (i = 0,1,2,\cdots,n)$ are the resultant “carries” of the modified “Carry-Lookahead circuit”.

The above logic is shown in Figure 26 with the underlying logic circuit shown in Figure 27, while the corresponding truth tables are shown in Figure 28 and Figure 29, respectively. Figure 27 is also related to the “Final summation” logic $i = 0,1,2,\cdots,(n-1)$ for de-converting the pair of binary substring which is the sum into a normal Two’s Complement binary format, which works with the predictive input of the truth table in Figure 29. Figures 26 to 29 resemble the classical Carry Lookahead circuit but the generate and propagate follow Equations (19) and (20) instead. Hence, the logic describes a “Carry Lookahead”-like circuit.

Specifically

- in classical Carry-Lookahead, a generate exist (gi=1) if both inputs are “1”, while a propagate exist (pi=1) if either of the inputs is “1”. In the “Carry-Lookahead”-like deconversion circuit, in contract, a generate exist (gi=1) if the y-bit is “1”, while a propagate exist (pi=1) if both inputs are “0” (xi=yi=0).
- in classical Carry-Lookahead, the existence of a generate at a lower-bit and a propagate the next higher-bit (and also the cascaded group generate and group propagate) will ensure a carry to continue to propagate. Similarly and dissimilarly, in the “Carry-Lookahead”-like deconversion circuit, the existence of a generate at a lower-bit and a propagate the next higher-bit (and also the cascaded group generate and group propagate) will, instead of a carry, ensure the nonzero y-bit to continue to be “moved” to the higher bit.

- in classical Carry-Lookahead, the sum bit exists (Si=1) if (i) either one of the three inputs is a “1”; or (ii) all three inputs are “1”. While in the “Carry-Lookahead”-like deconversion circuit, the positive X-bit, which is equivalent to the sum bit, exists (Xi=1) if the positive bit (i) is originally there and remains there without being terminated by an incoming y-bit being propagated from a less significant bit; (ii) is generated by the presence of a y-bit at the specific location; or (iii) is generated by the combination effect of generate (a relatively lower bit) and propagate (relatively higher bits).

Figure 30 shows a subcircuit block for de-converting a 4-bit generalized binary number to a 5-bit classical Two’s Complement number, with the additional bit being the sign bit. Deconversion into a Two’s Complement number will show whether the sum is in fact a negative value or not.

Inside the subcircuit of Figure 30 is a modified carry-lookahead circuit 2101 shown as Figure 31. A pair of x and y substrings of 4-bit length can be supplied to give the modified generate and propagate bits; each bit on the two substrings is fed as one of the inputs to a logic circuit in a corresponding position in an array of subcircuits shown in Figure 28.

The output of the array from Figure 31 will be a 5-bit single binary string Two’s Complement number where the leftmost digit indicates the sign of the binary number. If the value of the sum is positive, the leftmost bit is “0”, otherwise it is “1” if the binary number is negative. This is the concept of the classical Two’s Complement number. The de-conversion of a generalized binary number will

produce a classical Two's Complement number which is one extra bit longer, and the extra bit indicates the sign of the binary number.

Accordingly, the logic circuit in Figure 31 is also able to restore the sign of the value of the sum, to the binary output string in the case that the sum is negative, converting the binary string into true Two's Complement expression.

Continuing with the example of adding 13 to 14, the substrings at the bottom of Box 16 can be de-converted to the classical Two's complement number using the "Carry-lookahead"-like deconversion circuit. The final output sum as shown in Box 18.

Box 18

X = 28	0	0	0	1	1	1	0	0
Y = 1	0	0	0	0	0	0	0	1
X	0	0	0	0	1	1	0	1	1

X = 27

It can be seen the final sum is provided by $X = 27$, and the sum of the original augend and addend is $13 + 14 = 27$.

Subtraction

Subtraction of a subtrahend from a minuend can be done easily by reversing the x substring and y substring of the subtrahend immediately after generalization. Subsequently, the minuend and the "inverted subtrahend" can be regarded as an

augend and an addend and both generalized numbers can be summed up by using the addition method in the embodiment above.

Alternatively, the binary string of the negative subtrahend can be provided in the classical Two's Complement format for the processing system before generalisation.

A Second Embodiment

Another embodiment uses a different set of transformations to allow the addition of bits in every same bit position to be done concurrently as parallel processes. The different set of transformations may offer different advantage such as fewer transformation stages, but the underlying principles are the same as that discussed for the first embodiment. Generally, the transformed substrings do not generate carry-over values when being added up, and therefore may be added up without the ripple effect.

This embodiment has the same flow illustrated in the flowchart as shown as Figure 5, but use different transformations in Stage 3 and different summation adders in Stage 4.

Bit Sedimentation Transformation

In Stage 3 of the second embodiment, the two substrings of the same sign component are consolidated using the Bit Sedimentation Transformation. Where a bit in a first x substring has a zero value, and a bit in the same bit position in the

second x substring has a nonzero value, the nonzero is transferred from the second x substring to the same position in the first x substring. If the first substring has a nonzero value in that bit position, then the nonzero value in the second x substring is not transferred. Simply, the Bit Sedimentation Transformation fills up the 'empty' positions or the zero bits in one of the x substrings as much as possible by transferring all nonzeros in the same bit positions from the second x substring, thereby emptying the second x substring as much as possible. Visually, it looks like sedimentation, which gives this stage its name. The same applies to the opposite sign component y substrings.

This Bit Sedimentation Transformation may be expressed mathematically a follows:

$$\left(x_i'\right)_1 = \left(x_i\right)_1 + \left(x_i\right)_2 \tag{23}$$

$$\left(x_i'\right)_2 = \left(x_i\right)_1 \left(x_i\right)_2 \tag{24}$$

and

$$\left(y_i'\right)_1 = \left(y_i\right)_1 + \left(y_i\right)_2 \tag{25}$$

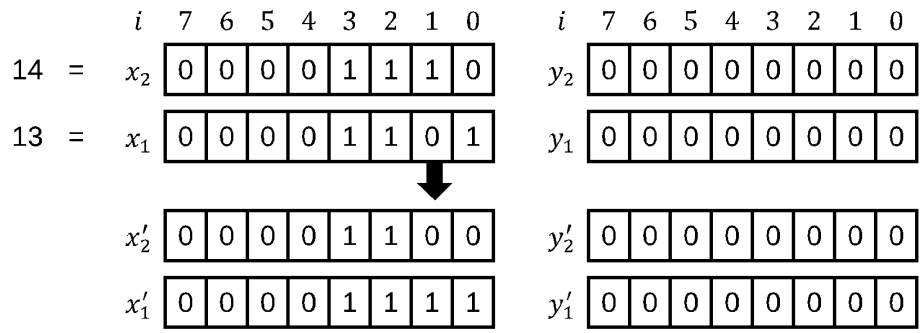
$$\left(y_i'\right)_2 = \left(y_i\right)_1 \left(y_i\right)_2 \tag{26}$$

The logic truth table for Bit Sedimentation Transformation is given as Figure 32. Figure 33 shows a logic circuit unit that corresponds to the truth table of Figure 32. Figure 34 shows two circuit blocks, each circuit block comprising a logic circuit unit shown in Figure 33 but one for the x substrings and the other for the y substrings.

For example, the generalised substrings for numbers 13 and 14 are shown sediment-ing in Box 19 below, where a nonzero is transferred from x_2 to x_1'. Since

both substrings have the same sign, and the bit positon is the same, the transferred nonzero has the same value and the total value of all the four substrings x_1', y_1', x_2', y_2' did not change.

Box 19



Transformation AB

The Transformation AB accommodates potential carry-overs by transforming input bits for the same sign component substrings, i.e. only between x_1 and x_2, and only between y_1 and y_2. This is because only when the value increase could there be carry-over values. The following are mathematical expressions of the transformation.

$$\left(x_i''\right)_1 = \left(x_i'\right)_2 \left(x_{i-1}'\right)_1 \tag{27}$$

$$\left(x_i''\right)_2 = \left(\overline{x_i'}\right)_1 \left(x_{i-1}'\right)_1 \tag{28}$$

$$\left(y_i''\right)_3 = \left(x_i'\right)_1 \left(\overline{x_i'}\right)_2 \left(\overline{x_{i-1}'}\right)_1 \tag{29}$$

and

$$\left(y_i''\right)_1 = \left(y_i'\right)_2 \left(y_{i-1}'\right)_1 \tag{30}$$

$$\left(y_i''\right)_2 = \left(\overline{y_i'}\right)_1 \left(y_{i-1}'\right)_1 \tag{31}$$

$$\left(x_i''\right)_3 = \left(y_i'\right)_1 \left(\overline{y_i'}\right)_2 \left(\overline{y_{i-1}'}\right)_1 \tag{32}$$

Figure 35 is the truth table for Transformation AB. The asterisks in the truth table relate to cases of $\left(x_i'\right)_1 = 0$, $\left(x_i'\right)_2 = 1$, and $\left(y_i'\right)_1 = 0$, $\left(y_i'\right)_2 = 1$, where these cases are excluded in the table as they are prevented due to the Bit Sedimentation Transformation.

Figure 36 shows a logic circuit unit for Transformation AB., marked for the positive x substrings. The same logic circuit unit in Figure 36 can be used for the 'negative' y substrings if the leads are connected to the corresponding input data from the y substrings. As shown in Figure 37, the subcircuit block 4001 at the top being for x substrings and the subcircuit 4003 block at the bottom being for y substrings.

Box 20 shows the derivation of the truth table in Figure 35. With regards to the x substrings, the input to Transformation AB are the bits $(x_i')_1$, $(x_i')_2$, $(x_{i-1}')_1$. Transformation AB is a single step execution of Transformation A and Transformation B

More specifically, $(x_i')_1$, $(x_i')_2$ should be subjected to Transformation B, while $(x_i')_1$, $(x_{i-1}')_1$ should be subjected to Transformation A. However, $(x_i')_1$ should not be transformed twice. Therefore, in Box 20, only one bit $(x_i')_2$ is subjected to Transformation B, while two bits $(x_i')_1$, $(x_{i-1}')_1$ are subjected to Transformation A. It

is noted for Case 7 in Box 20, a further bit simplification should be implemented to the output at $i = 1$ to update $x_1'' = 1$, $y_3'' = 1$ to become $x_1'' = 0$, $y_3'' = 0$.

Box 20

	Input $i = 1$	SEP / Transformation B	Output $i = 1$
1)	x_2' 0 / x_1' 0 0 = 0 & 0 0 =	x_2'' 0 / y_3'' 0 & Transformation B: x_2'' 0 / x_1'' 0	x_2'' 0 / x_1'' 0 ; y_3'' 0
2)	x_2' 0 / x_1' 0 1 = 0 & 0 1 =	x_2'' 1 / y_3'' 0 1 & Transformation B: x_2'' 0 / x_1'' 0	x_2'' 1 / x_1'' 0 ; y_3'' 0 1
3)	x_2' 1 / x_1' 0 0	Not applicable because non-zero in x2 on zero x1 eliminated by sedimentation, reducing cases or permutation to consider.	
4)	x_2' 1 / x_1' 0 1		
5)	x_2' 0 / x_1' 1 0 = 0 & 1 0 =	x_2'' 1 0 / y_3'' 1 0 & Transformation B: x_2'' 0 / x_1'' 0	x_2'' 1 0 / x_1'' 0 ; y_3'' 1
6)	x_2' 0 / x_1' 1 1 = 0 & 1 1 =	x_2'' 1 1 / y_3'' 1 1 & Transformation B: x_2'' 0 / x_1'' 0	x_2'' 1 0 / x_1'' 0 ; y_3'' 0 1
7)	x_2' 1 / x_1' 1 0 = 1 & 1 0 =	x_2'' 1 0 / y_3'' 1 0 & Transformation B: x_2'' 0 / x_1'' 1	x_2'' 1 0 / x_1'' 1 ; y_3'' 1
8)	x_2' 1 / x_1' 1 1 = 1 & 1 1 =	x_2'' 1 1 / y_3'' 1 1 & Transformation B: x_2'' 0 / x_1'' 1	x_2'' 1 0 / x_1'' 1 ; y_3'' 0 1

where at position i = 0, all values of any substring at position -1 have a zero value.

Box 21 is the same as Box 20 except that Box 21 relates to the y substrings. The similar Case 7 in Box 21, as that in Box 20, should be observed where a further bit simplification should be implemented.

Box 21

	Input $i = 1$	SEP / Transformation B	Output $i = 1$
1)	y'_2 0 = 0 = y'_1 0 0 & 0 0	y''_2 0 & y''_1 0 x''_3 0	y''_2 0 x''_3 0 y''_1 0
2)	y'_2 0 = 0 = y'_1 0 1 & 0 1	y''_2 0 & y''_1 0 y''_2 1 x''_3 0 1 y''_1 0	
3)	y'_2 1 y'_1 0 0	Not applicable because non-zero in x2 on zero x1 eliminated by sedimentation, reducing cases or permutation to consider.	
4)	y'_2 1 y'_1 0 1		
5)	y'_2 0 = 0 = y'_1 1 0 & 1 0	y''_2 0 & y''_1 0 y''_2 1 0 x''_3 1 y''_1 0	
6)	y'_2 0 = 0 = y'_1 1 1 & 1 1	y''_2 0 & y''_1 0 y''_2 1 0 x''_3 0 1 y''_1 0	
7)	y'_2 1 = 1 = y'_1 1 0 & 1 0	y''_2 0 & y''_1 1 y''_2 1 0 x''_3 1 y''_1 1	
8)	y'_2 1 = 1 = y'_1 1 1 & 1 1	y''_2 0 & y''_1 1 y''_2 1 0 x''_3 0 1 y''_1 1	

where at position i = 0, all values of any substring at position -1 have a zero value.

Box 22 shows an alternative derivation of the truth table in Figure 35, for the positive components bits $(x_i')_1$, $(x_i')_2$, $(x_{i-1}')_1$. In this case, in order for $(x_i')_1$ to be not transformed twice, the two bits $(x_i')_1$, $(x_i')_2$ are subjected to Transformation B while the single bit $(x_{i-1}')_1$ is subjected to Transformation A.

Box 22

	Input $i=1$	SEP	Transformation B	Output $i=1$
	i 1 0 — 1 — 1 0	i 2 1 0	i 2 1 0	i 2 1 0 — 1 0
1)	x_2' 0 = 0 = x_1' 0 0 0 & 0	x_2'' 0 & y_3'' 0	x_2'' 0 x_1'' 0	x_2'' 0 y_3'' 0 x_1'' 0
2)	x_2' 0 = 0 = x_1' 0 1 0 & 1	x_2'' 1 & y_3'' 0 1	x_2'' 0 x_1'' 0	x_2'' 1 y_3'' 0 1 x_1'' 0
5)	x_2' 0 = 0 = x_1' 1 0 1 & 0	x_2'' 0 & y_3'' 0	x_2'' 0 x_1'' 1	x_2'' 0 y_3'' 0 x_1'' 1
6)	x_2' 0 = 0 = x_1' 1 1 1 & 1	x_2'' 1 & y_3'' 0 1	x_2'' 0 x_1'' 1	x_2'' 1 y_3'' 0 1 x_1'' 1
7)	x_2' 1 = 1 = x_1' 1 0 1 & 0	x_2'' 0 & y_3'' 0	x_2'' 1 x_1'' 0	x_2'' 1 0 y_3'' 0 x_1'' 0
8)	x_2' 1 = 1 = x_1' 1 1 1 & 1	x_2'' 1 & y_3'' 0 1	x_2'' 1 x_1'' 0	x_2'' 1 1 y_3'' 0 1 x_1'' 0

The advantage of the step of sedimentation can be seen in Box 20 and Box 21 where case 3 and case 4 are indicated as impossible cases. This reduces the

number of cases to address. Also, the speed of just sweeping the nonzero values over to one substring is faster and simpler than to operate case 3 and case 4.

Box 23 is the same as Box 22 but for the negative sign components $(y_i')_1$, $(y_i')_2$, $(y_{i-1}')_1$.

Box 23

	Input $i=1$	SEP	Transformation B	Output $i=1$
1)	y_2' 0 = 0 = y_1' 0 0 0 & 0	y_2'' 0 & x_3'' 0	y_2'' 0 y_1'' 0	y_2'' 0 x_3'' 0 y_1'' 0
2)	y_2' 0 = 0 = y_1' 0 1 0 & 1	y_2'' 1 & x_3'' 0 1	y_2'' 0 y_1'' 0	y_2'' 1 x_3'' 0 1 y_1'' 0
5)	y_2' 0 = 0 = y_1' 1 0 1 & 0	y_2'' 0 & x_3'' 0	y_2'' 0 y_1'' 1	y_2'' 0 x_3'' 0 y_1'' 1
6)	y_2' 0 = 0 = y_1' 1 1 1 & 1	y_2'' 1 & x_3'' 0 1	y_2'' 0 y_1'' 1	y_2'' 1 x_3'' 0 1 y_1'' 1
7)	y_2' 1 = 1 = y_1' 1 0 1 & 0	y_2'' 0 & x_3'' 0	y_2'' 1 y_1'' 0	y_2'' 1 0 x_3'' 0 y_1'' 0
8)	y_2' 1 = 1 = y_1' 1 1 1 & 1	y_2'' 1 & x_3'' 0 1	y_2'' 1 y_1'' 0	y_2'' 1 1 x_3'' 0 1 y_1'' 0

Summation of output from Transformation AB

For the three substrings produced by applying Transformation AB shown in Box 20 for the x substrings or Box 21 for the y substrings, each permutation of different three input bit values give three output bit values. One of the three output bits is of the opposite sign component, due to the effect of the SEP conversion.

It has been observed that the specific combinations of the bits of the binary substrings of the same sign component give the sum for the output bit in position i. Some of the cases in the truth table shown as Figure 38 are included due to empirically made observations.

On the other hand, some cases or permutations of input bits that are not shown in the table, as these permutations prevented or eliminated by application of Transformation AB. All useable permutations are already shown in the table.

The following is the corresponding mathematical equations that correspond to the truth table.

$$x_i''' = \left(x_i''\right)_1 + \left(x_i''\right)_2 \left(\overline{x_i''}\right)_3 + \left(\overline{x_i''}\right)_2 \left(x_i''\right)_3 + \left(x_{i-1}''\right)_2 \left(x_{i-1}''\right)_3 \quad (33)$$

$$y''' = \left(y_i''\right)_1 + \left(y_i''\right)_2 \left(\overline{y_i''}\right)_3 + \left(\overline{y_i''}\right)_2 \left(y_i''\right)_3 + \left(y_{i-1}''\right)_2 \left(y_{i-1}''\right)_3 \quad (34)$$

As may be seen from the table or from the equations, the sum value of each bit position can be obtained from bits in the three substrings of the same sign

component, and the bits in the current position i and the lower position i.

Figure 39 is a logic circuit unit corresponding to the truth table shown in Figure 38, clearly showing five input leads and one output lead. The top drawing in Figure 40 is a block diagram of a sub-circuit 4301 for x substrings, which is based on the logic circuit unit in Figure 39. The bottom drawing in Figure 40 is the corresponding block diagram of a sub-circuit 4303 for y substrings, which is the same as the sub-circuit 4301 for x substrings but the input leads of which are connected to take input from y substrings.

Figure 41 is a subcircuit block diagram for adding two 8-bit generalized binary numbers of the present embodiment, Figure 42 shows the details of the subcircuit of Figure 41, which have already been shown separately in Figure 34 for Subcircuit A4-2XY41, Figure 37 for Subcircuit A4-2XY42, Figure 40 for Subcircuit A4-2XY43, and Figure 16 for Subcircuit 2SCV-21.

Box 24 shows a further example that illustrates the four steps above is given below. The sum of $(\tilde{n}_w)_1 = 10\bar{1}1\bar{1}101$ (a generalized binary number representing decimal 109) and $(\tilde{n}_w)_2 = 0100\bar{1}10\bar{1}$ (a generalized binary number representing decimal 59) can be computed using the adder in Figure 41. The sum is $N_w = 1\bar{1}1\bar{1}01000$ (a generalized binary number representing decimal 168). Details of the steps are shown below.

Box 24										
Generalising augend $(\tilde{n}_w)_1$ and addend $(\tilde{n}_w)_2$	$(\tilde{n}_w)_1^+ = (x_i)_1$		1	0	0	1	0	1	0	1
	$(\tilde{n}_w)_2^+ = (x_i)_2$		0	1	0	0	0	1	0	0
	$(\tilde{n}_w)_2^- = (y_i)_2$		0	0	0	0	$\bar{1}$	0	0	$\bar{1}$
	$(\tilde{n}_w)_1^- = (y_i)_1$		0	0	$\bar{1}$	0	$\bar{1}$	0	0	0
Sedimentation	$(x_i')_2 = (x_i)_1 (x_i)_2$		0	0	0	0	0	1	0	0
	$(x_i')_1 = (x_i)_1 + (x_i)_2$		1	1	0	1	0	1	0	1
	$(y_i')_1 = (y_i)_1 + (y_i)_2$		0	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$
	$(y_i')_2 = (y_i)_1 (y_i)_2$		0	0	0	0	$\bar{1}$	0	0	0
Transformation AB	$(x_i'')_1 = (x_i')_2 (x_{i-1}')_1$	0	0	0	0	0	0	0	0	0
	$(x_i'')_2 = (\overline{x_i'})_1 (x_{i-1}')_1$	1	0	0	1	0	1	0	1	0
	$(y_i'')_3 = (x_i')_1 (\overline{x_i'})_2 (\overline{x_{i-1}'})$	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$
	$(y_i'')_1 = (y_i')_2 (y_{i-1}')_1$	0	0	0	0	0	0	0	0	0
	$(y_i'')_2 = (\overline{y_i'})_1 (y_{i-1}')_1$	0	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	0
	$(x_i'')_3 = (y_i')_1 (\overline{y_i'})_2 (\overline{y_{i-1}'})$	0	0	0	1	0	0	0	0	1
Parallel Bit Addition Operations	Eq. (33)	1	0	1	0	0	1	0	1	1
	Eq. (34)	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	$\bar{1}$
Simplification of the sum	N_w	1	$\bar{1}$	1	$\bar{1}$	0	1	0	0	0

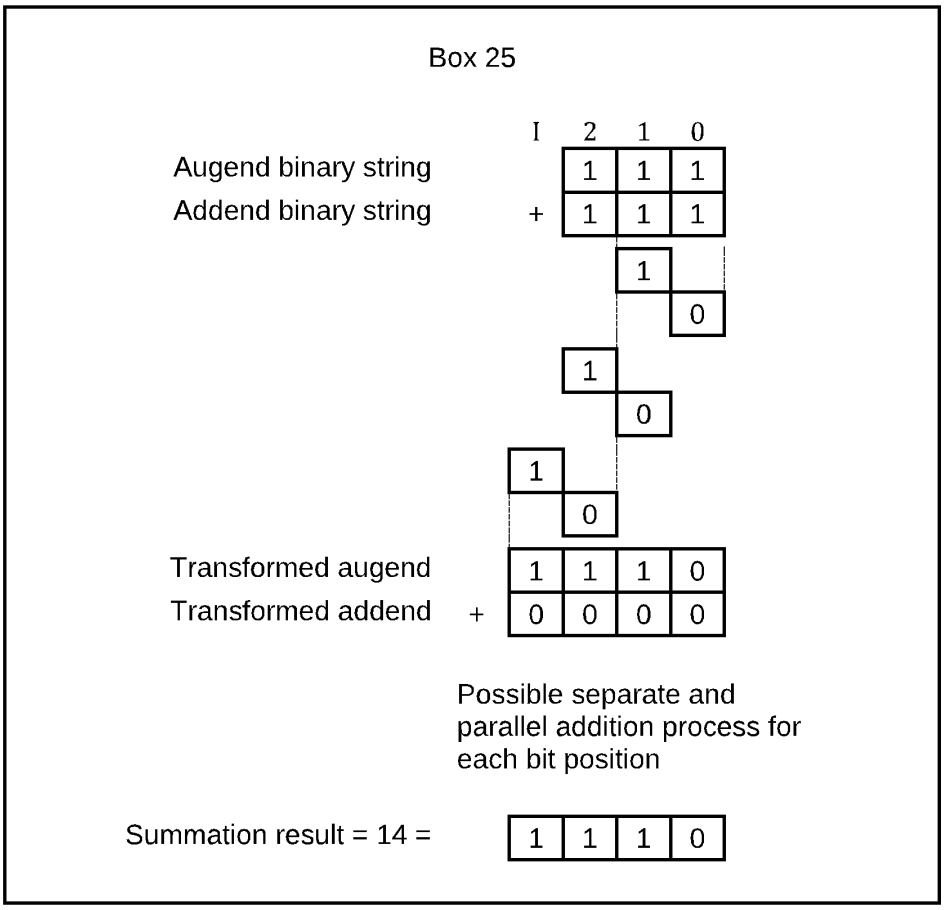
After the sum is obtained, the sum is deconverted into a normal binary string.

Bare Embodiments

As the entire process from generalization of the augend and addend to the summation of transformed generalized binary strings comprises many distinct stages, each of which can be implemented in the form of an independent circuit block, to be connected to each other. Therefore, embodiments include these distinct stages.

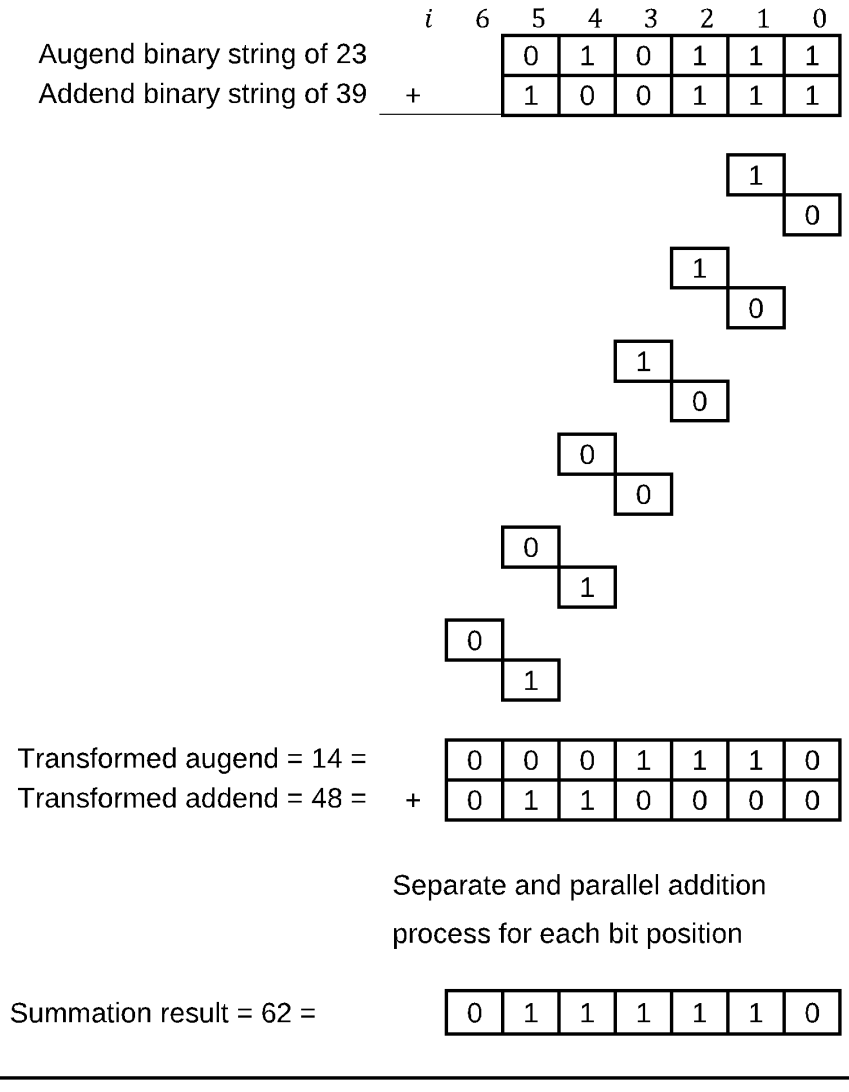
For example, it is possible that the Transformation A may be licensed, sold, used or supplied as algorithm software, in a physical logic circuit or as part of a processor chip, without being bundled to be used with Simplification Transformation And Transformation B. Similarly. Transformation B may be licensed, sold, used or supplied as algorithm software, in a physical logic circuit or as part of a processor chip, without being bundled to be used with Simplification Transformation And Transformation A.

In limited cases, the binary strings transformed by Transformation B without Transformation A can be summed for an adder operating parallel additions, as shown in Box 25, in the addition of 7 and 7, i.e. 111 + 111 = 1110 (14 in decimal), i.e. because the carry-overs have been processed in advance.



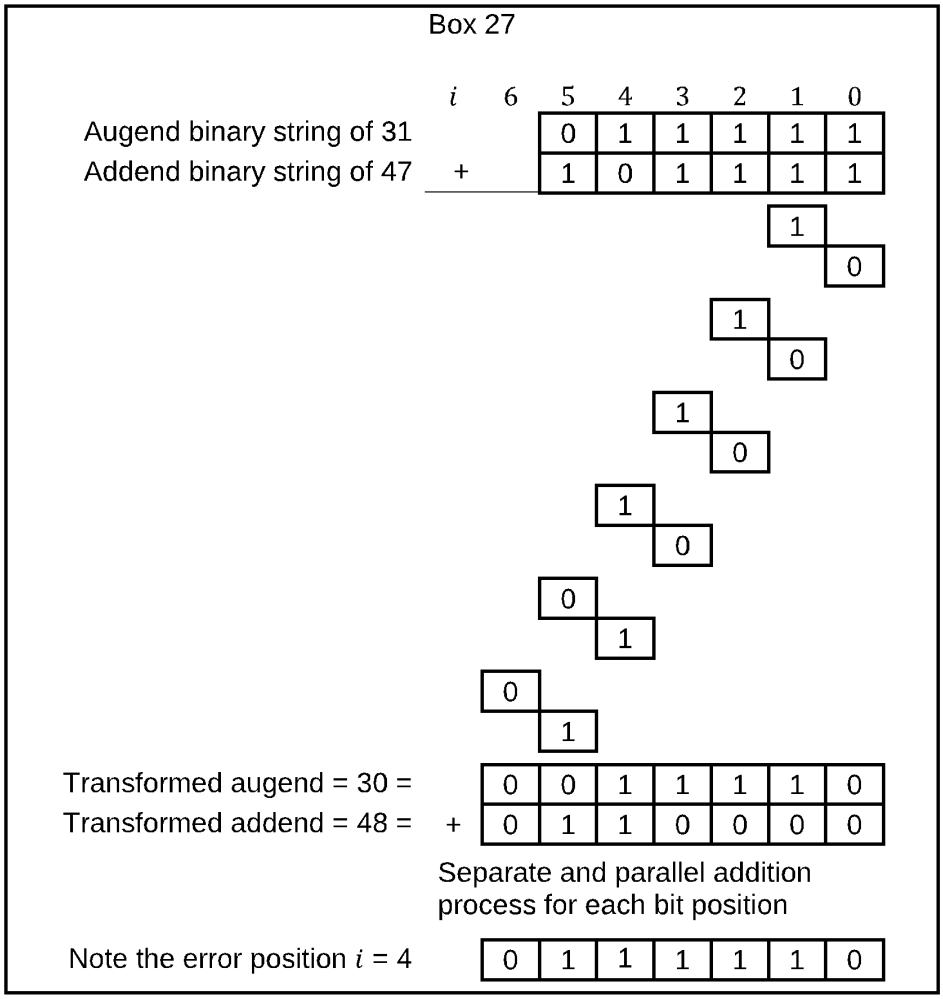
Transformation B may also be used to processed carry-overs in advance for numbers 23 and 39, as shown in Box 26, so that the processed binary strings can be added up in parallel additions.

Box 26

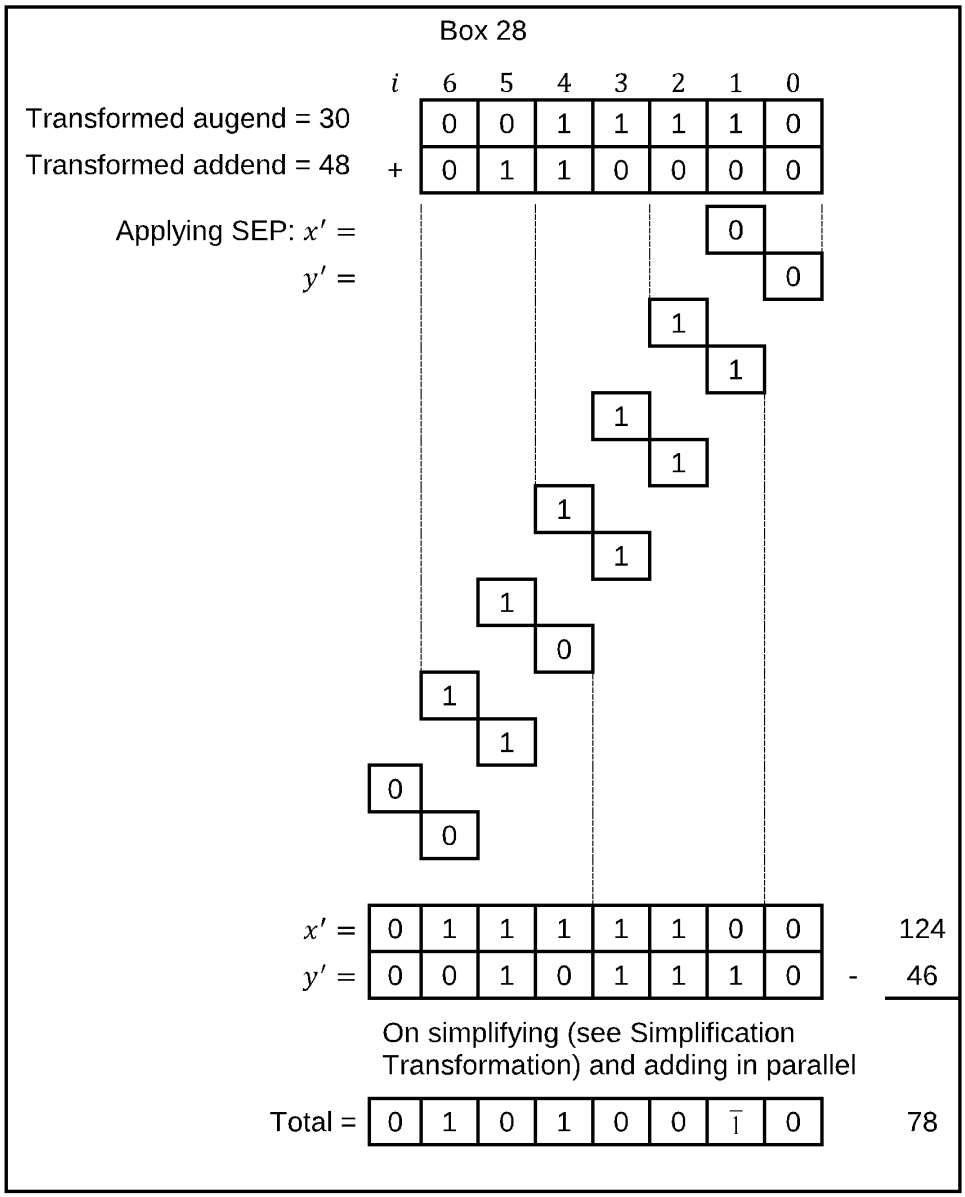


As shown in Box 27, however, an attempt to apply Transformation B to process carry-overs in advance for adding 31 to 47 shows that applying only the Transformation B transformation may be inadequate in some situations to remove all carry-overs in advance. Therefore, further processing using the Transformation A is recommended, shown below. However, there could be a solution or transformation which is not Transformation A and not part of this patent application that may be

used to address this shortcoming of Transformation B. Accordingly, the context-limited use does not prevent Transformation B. from being promoted, sold, and licenced independently and used with other binary string processes.



Box 28 is a continuation of Box 27, showing that an application of the Transformation A produces two binary strings that may be summed using parallel adding operations without the danger of triggering a carry-over.



As shown above, the conversion of Transformation B may be applied independently in some cases without using Simplification Transformation and Transformation A.

The adder for performing the above using parallel adding operations is not provided, as there are many different designs of multi-bit adders that may be conceived once the reader is taught the concepts as discussed.

Embodiment using only Transformation A

In a further embodiment, the same steps executed for the first embodiment are taken except that Transformation B is not applied, i.e. stopping at Simplification Transformation after Transformation A. The following step c can be taken to sum up the two pairs of substrings produced.

(a) Transformation A

(b) Simplification

(c) separately adding the positive and negative strings by applying the following equations;

$$x_i''' = (x_i'')_1 (\overline{x_i''})_2 + (\overline{x_i''})_1 (x_i'')_2 + (x_{i-1}'')_1 (x_{i-1}'')_2 \quad (35)$$

$$y_i''' = (y_i'')_1 (\overline{y_i''})_2 + (\overline{y_i''})_1 (y_i'')_2 + (y_{i-1}'')_1 (y_{i-1}'')_2 \quad (36)$$

where the logic truth is presented in Figure 43, and the logic circuits in Figure 44 and Figure 45.

In this embodiment, the effect of Transformation B in preventing carry-overs is in integrated into the logic circuits as shown in the truth table.

While there has been described in the foregoing description preferred embodiments of the present invention, it will be understood by those skilled in the technology concerned that many variations or modifications in details of design, construction or operation may be made without departing from the scope of the present invention as claimed.

For example, given any combinational logic circuit, there are many other logically equivalent circuits that have the same input/output that comply with the relevant truth table, and these are also embodiments of the inventions and within the scope of the application.

Figure 46 shows an adder that can be used in placed of the one in Figure 21 as a safety measure in case there is an accidental carry-over, to prevent the system from being crashed.

In many written contents relating to the same or similar technical fields, an output is often described as if it is physically different from the input, as this makes it easier for the reader to understand the content. For example, one may say an input substring is transformed into an output which is a changed substring. However, the physical imagery and the corresponding vocabulary should not become a limitation to the scope of the patent, such as in requiring the output to be recorded in a different part of the register or memory from the record of the input, or even requiring the output to be written over the input.

CLAIMS

1. A method of making available a bit position i in one of two binary strings to store a carry-over value, comprising the step(s) of:

replacing a nonzero with a zero in the bit position i, the nonzero in the bit position representing a value; and

providing at least one nonzero in a different bit position in the two binary strings representing an equal value.

2. A method of making available a bit position i in one of two binary strings to store a carry-over value, as claimed in claim 1, further comprising

at least one nonzero in a further binary string representing a value of a different sign component,

said equal value is provided by offsetting the value of the at least one nonzero in a different bit position in the two binary strings with the at least one nonzero in the further binary string .

3. A method of making available a bit position i in one of two binary strings to store a carry-over value, as claimed in claim 1, wherein

the step of providing at least one nonzero in a different bit position in the two binary strings is taken when executing either one of case 2) and case 3) according to the transformation truth table:

	Input		Output	
	$(x_i'')_1$	$(x_i'')_2$	$(x_{i+1}''')_2$	$(x_i''')_1$
1)	0	0	0	0
2)	0	1	0	1
3)	1	0	0	1
4)	1	1	1	0

for position i in an augend binary string $(x_i'')_1$ and an addend binary string $(x_i'')_2$ and in transformed augend binary strings $(x_{i+1}''')_2$ and $(x_i''')_1$.

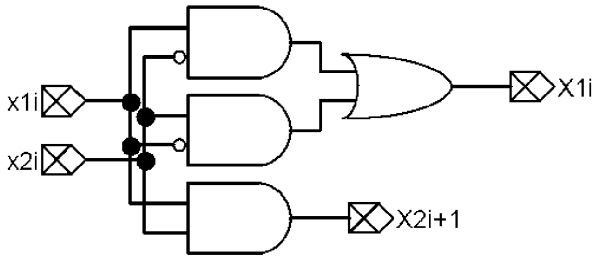
4. A method of making available a bit position i in one of two binary strings to store a carry-over value, as claimed in claim 1,

the step of providing at least one nonzero in a different bit in the two binary strings is taken when executing either one of case 2) and case 3) according to the transformation truth table:

	Positive input		Output		Negative input		Output	
	$(x_i'')_1$	$(x_i'')_2$	$(x_{i+1}''')_2$	$(x_i''')_1$	$(y_i'')_1$	$(y_i'')_2$	$(y_{i-1}''')_2$	$(y_i''')_1$
1)	0	0	0	0	0	0	0	0
2)	0	1	0	1	0	$\overline{1}$	0	$\overline{1}$
3)	1	0	0	1	$\overline{1}$	0	0	$\overline{1}$
4)	1	1	1	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0

the two binary strings each being one of a pair of generalised binary strings $(x_i'')_1$, $(y_i'')_1$and $(x_i'')_2$, $(y_i'')_2$, and of the same sign component.

5. A logic circuit for applying a transformation according to the truth table in claim 3 or claim 4 comprising the following configuration:



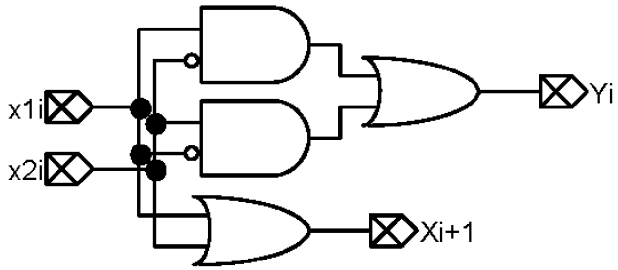
6. A method of making available a bit position i in one of two binary strings to store a carry-over value, as claimed in claim 1, wherein

the step of providing at least one nonzero in a different bit in the two binary strings is taken when executing either one of case 2) and case 3) according to the transformation truth table:

	Input n_w^+		Output $(n'_w)_1$		Input n_w^-		Output $(n'_w)_2$	
	$(x_i)_1$	$(x_i)_2$	$(x'_{i+1})_1$ ‡	$(y'_i)_1$ §	$(y_i)_1$	$(y_i)_2$	$(y'_{i+1})_2$	$(x'_i)_2$
1)	0	0	0	0	0	0	0	0
2)	0	1	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	1
3)	1	0	1	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	1
4)	1	1	1	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0

the two binary strings each being one of a pair of generalised binary strings, $(x_i)_1$, $(y_i)_1$ and $(x_i)_2$, $(y_i)_2$, and of the same sign component.

7. A logic circuit for applying a transformation according to the truth table in claim 6, comprising the following configuration:



8. A method of adding at least two numbers in the form of binary strings, the method comprising the steps of:

i. providing a first pair of binary substrings, x_1 and y_1 wherein x_1-y_1 gives the value of a first binary string representing a first number;

ii. providing a second pair of binary substrings, x_2 and y_2 wherein x_2 - y_2 gives the value of a second binary string representing a second number;

iii. performing the following transformations and operation:

Transformation A: producing a third pair of substrings x'_1 and y'_1 from $x_1$and x_2 and producing a fourth pair of substrings x'_2 and y'_2 from $y_1$and y_2 according to the truth table where i is the bit position:

Input n_w^+		Output $(n'_w)_1$		Input n_w^-		Output $(n'_w)_2$	
$(x_i)_1$	$(x_i)_2$	$(x'_{i+1})_1$ ‡	$(y'_i)_1$ §	$(y_i)_1$	$(y_i)_2$	$(y'_{i+1})_2$	$(x'_i)_2$
0	0	0	0	0	0	0	0
0	1	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	1
1	0	1	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	1
1	1	1	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0

Simplification Transformation: producing substrings x_1'' and y_1'' from x_1' and y_1' by replacing the nonzero values where x_1' = 1 and y_1'= 1 with zeros for each bit position i; and

producing substrings x_2'' and y_2'' from x_2' and y_2' = by replacing the nonzero values where x_2' = 1 and y_2' = 1 with zeros in x_2'' and y_2'' for each bit position i;

Transformation B: consolidating the values of the x_1'', x_2'' to produce a pair of substrings x_1''' and x_2''', and further consolidating the values of y_1'', y_2'' to produce a pair of substrings y_1''' and y_2''', according to the truth table:

	Positive input		Output		Negative input		Output	
	$(x_i'')_1$	$(x_i'')_2$	$(x_{i+1}''')_2$	$(x_i''')_1$	$(y_i'')_1$	$(y_i'')_2$	$(y_{i-1}''')_2$	$(y_i''')_1$
1)	0	0	0	0	0	0	0	0
2)	0	1	0	1	0	$\overline{1}$	0	$\overline{1}$
3)	1	0	0	1	$\overline{1}$	0	0	$\overline{1}$
4)	1	1	1	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0

Summing Operation: performing the addition to produce the sum as two substrings as $\boldsymbol{X}$ and $\boldsymbol{Y}$ according to the truth table:

Inputs				Outputs	
x_1'''	y_1'''	x_2'''	y_2'''	$\boldsymbol{X}'$	$\boldsymbol{Y}'$
0	0	0	0	0	0
0	0	1	0	1	0
0	0	0	$\overline{1}$	0	$\overline{1}$
1	0	0	0	1	0
1	0	1	0	Prevented	
1	0	0	$\overline{1}$	1	$\overline{1}$
0	$\overline{1}$	0	0	0	$\overline{1}$
0	$\overline{1}$	1	0	1	$\overline{1}$
0	$\overline{1}$	0	$\overline{1}$	Prevented	

iv. adding up X and Y to produce a single binary string having the value of the sum of the two numbers.

9. A method of adding at least two numbers in the form of binary strings as claimed in claim 8, wherein

the Summing Operation is followed by a Simplification Transformation to produce the sum as two substrings as □ and $\boldsymbol{Y}$ by replacing the nonzero values where $X' = 1$,$Y' = \overline{1}$ with zeros for each bit position i, according to the truth table:

Inputs				Outputs		Simplified Outputs	
x_1'''	y_1'''	x_2'''	y_2'''	$\boldsymbol{X'}$	$\boldsymbol{Y'}$	$\boldsymbol{X}$	$\boldsymbol{Y}$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	0	0	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$
1	0	0	0	1	0	1	0
1	0	1	0	Prevented			
1	0	0	$\overline{1}$	1	$\overline{1}$	0	0
0	$\overline{1}$	0	0	0	$\overline{1}$	0	$\overline{1}$
0	$\overline{1}$	1	0	1	$\overline{1}$	0	0
0	$\overline{1}$	0	$\overline{1}$	Prevented			

10. A method of adding at least two numbers in the form of binary strings as claimed in claim 9, wherein before step iv in claim 8, the method further comprises the steps of:

providing a third pair of binary substrings, x_3 and y_3, x_3- y_3 being the value of a third binary string representing a third number;

repeating iii, by taking

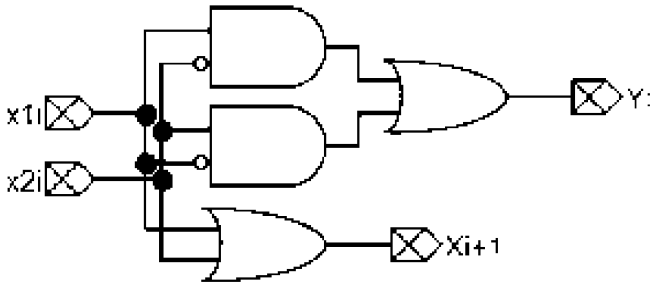
$\boldsymbol{X}$ substring produced by the Summing Operation as x_1

$\boldsymbol{Y}$ substring produced by the Summing Operation as y_1;

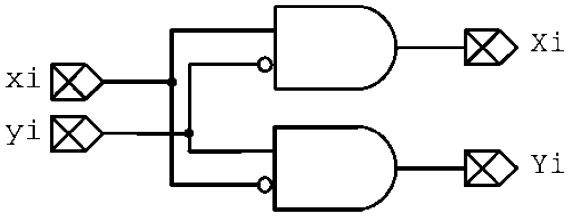
x_3 as x_2;

y_3 as y_2.

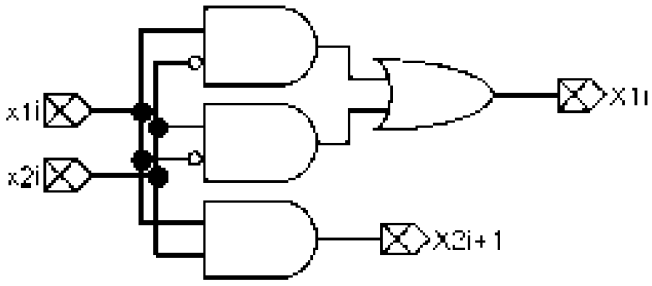
11. A logic circuit for applying the Transformation A of the method of claim 8 comprising the following configuration:



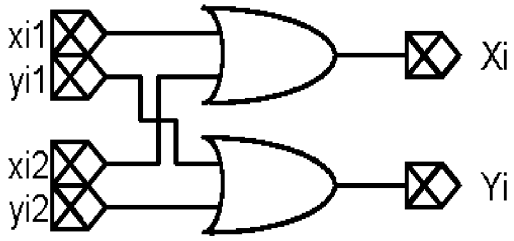
12. A logic circuit for applying Simplification Transformation of the method of claim 8 comprising the following configuration:



13. A logic circuit for applying Transformation B of the method of claim 8 comprising the following configuration:



14. A logic circuit for applying Summing Operation of the method of claim 8 comprising the following configuration:



15. A method of adding two numbers in the form of binary strings, the method comprising the steps of:

providing a first binary string representing the first number and a second binary string representing the second number to each of the first binary string; and the second binary string, assigning a further binary string to form a pair of binary substrings, such that

the sum of a first pair of positive substring x_1 and negative substring y_1 is the value of the first binary string; and

the sum of a second pair of positive substring x_2 and negative substring y_2 is the value of the second binary string;

performing the following transformations and operation:

Sedimentation Transformation: re-arranging the nonzero bits between x_1 and x_2, and between y_1 and y_2, according to the truth table to output the substrings x_1' , x_2' , y_1' , y_2'

Input		Output		Input		Output	
$(x_i)_1$	$(x_i)_2$	$(x_i')_1$	$(x_i')_2$	$(y_i)_1$	$(y_i)_2$	$(y_i')_1$	$(y_i')_2$
0	0	0	0	0	0	0	0
0	1	1	0	0	$\overline{1}$	$\overline{1}$	0
1	0	1	0	$\overline{1}$	0	$\overline{1}$	0
1	1	1	1	$\overline{1}$	$\overline{1}$	$\overline{1}$	$\overline{1}$

Transformation AB: taking substrings x_1' , x_2' , y_1' , y_2' as input to produce substrings x_1'' , x_2'' , x_3'' , y_1'' , y_2'', y_3'' as output according to the following truth table:

Input			Output		
$(x_i')_1$	$(x_i')_2$	$(x_{i-1}')_1$	$(x_i'')_1$	$(x_i'')_2$	$(y_i'')_3$
0	0	0	0	0	0
0	0	1	0	1	0
1	0	0	0	0	$\overline{1}$
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	1	0	0
Input			Output		
$(y_i')_1$	$(y_i')_2$	$(y_{i-1}')_1$	$(y_i'')_1$	$(y_i'')_2$	$(x_i'')_3$
0	0	0	0	0	0
0	0	$\overline{1}$	0	$\overline{1}$	0
$\overline{1}$	0	0	0	0	1
$\overline{1}$	0	$\overline{1}$	0	0	0
$\overline{1}$	$\overline{1}$	0	0	0	0
$\overline{1}$	$\overline{1}$	$\overline{1}$	$\overline{1}$	0	0

Summing Operation: taking substrings x_1'', x_2'', x_3'', y_1'', y_2'', y_3'' as input to produce output substrings x_i''' , y_i''' as output according to the following truth table:

Input					Output
$(x_i'')_1$	$(x_i'')_2$	$(x_i'')_3$	$(x_{i-1}'')_2$	$(x_{i-1}'')_3$	x_i'''
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	0	1
1	0	0	0	1	1
Input					Output
$(y_i'')_1$	$(y_i'')_2$	$(y_i'')_3$	$(y_{i-1}'')_2$	$(y_{i-1}'')_3$	y_i'''
0	0	0	0	0	0
0	0	0	0	$\overline{1}$	0
0	0	0	$\overline{1}$	0	0
0	0	0	$\overline{1}$	$\overline{1}$	$\overline{1}$
0	0	$\overline{1}$	0	0	$\overline{1}$
0	0	$\overline{1}$	$\overline{1}$	0	$\overline{1}$
0	$\overline{1}$	0	0	0	$\overline{1}$
0	$\overline{1}$	0	0	$\overline{1}$	$\overline{1}$
0	$\overline{1}$	$\overline{1}$	0	0	0
$\overline{1}$	0	0	0	0	$\overline{1}$
$\overline{1}$	0	0	0	$\overline{1}$	$\overline{1}$

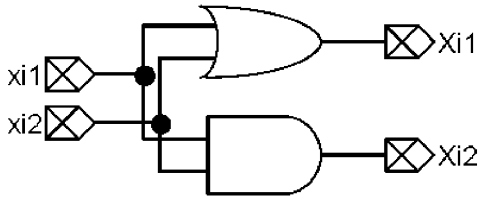
16. A method of adding two numbers in the form of binary strings, as claimed in claim 15, the method comprising a further step after Summing Operation, the further step comprising:

Simplification Transformation: producing output substrings X and Y by replacing the nonzero values where $X' = 1$, $Y' = \overline{1}$ with zeros for each bit position i.

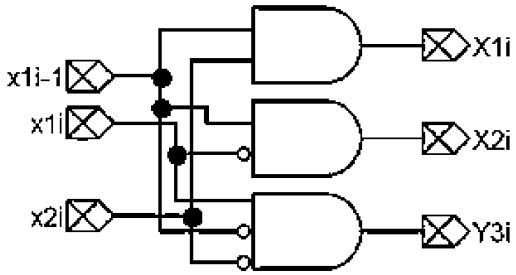
17. A method of adding two numbers in the form of binary strings as claimed in claim 15, the method comprising the further step(s) of:

adding up x_i''' and y_i''' to produce a single binary string having the value of the sum of the two numbers.

18. A logic circuit for applying the Sedimentation Transformation in the method of claim 15 comprising the following configuration:



19. A logic circuit for applying the Transformation AB the method of claim 15 comprising the following configuration:



20. A logic circuit for applying Summing Operation in the method of claim 8 comprising the following configuration:

