the method of summation into a method of subtraction of the number of which the substrings are interchanged.

In a second aspect, the invention proposes a logical circuit for applying step 1 of the method, comprising a configuration as shown in Figure 11.

Furthermore, the invention proposes a logical circuit for applying step 2 of the method, comprising a configuration as shown in Figure 14.

Furthermore, the invention proposes a logical circuit for applying step 3 of the method, comprising a configuration as shown in Figure 17.

Furthermore, the invention proposes a logical circuit for applying step a) and step b) of the method, comprising a configuration illustrated in Figure 37a.

Furthermore, the invention proposes a logical circuit for applying step i) of the method, comprising a configuration illustrated in Figure 40.

Furthermore, the invention proposes a logical circuit for applying step ii) of the method, comprising a configuration illustrated in Figure 43.

Furthermore, the invention proposes a logical circuit for the step of converting nonzero bits in the same position in the substrings x and y to zero bits, comprising a configuration illustrated in Figure 30b.

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 two embodiments of the invention;

Figure 6 is a subcircuit for converting a 2's complement binary string into a generalised format used in the embodiments of Figure 5;

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

Figure 7b illustrates how the circuit diagram of Figure 7a generalizes a binary string;

Figure 8 is a higher level subcircuit for rearranging, adding or removing the zeros and nonzeros bits in binary strings output by the subcircuit of Figure 6 for semi-optimization, used in an embodiment referred to in Figure 5;

Figure 9 shows three subcircuits and the arrangement thereof which are represented by the subcircuit in Figure 8;

Figure 10 is a subcircuit diagram that is within one of the three subcircuits in Figure 9;

Figure 11 shows the logic circuit that is in each of the nine subcircuits the circuit diagram shown in Figure 10;

Figure 12 is the Truth Table of the logic circuit of Figure 11;

Figure 13 is a subcircuit diagram that is within another one of the three subcircuits in Figure 9;

Figure 14 shows the logic circuit that is in each of the subcircuits shown in the circuit diagram of Figure 13;

Figure 15 is the Truth Table for the logic circuit of Figure 13;

Figure 16 is a subcircuit diagram that is within another one of the three subcircuits in Figure 9;

Figure 17 shows the logic circuit that is in each of the subcircuits shown in the circuit diagram of Figure 16;

Figure 18 is the Truth Table for the logic circuit of Figure 16;

Figure 19 is a higher level subcircuit that provides an adder used in one of the steps in the flowchart of Figure 5;

Figure 20 is a subcircuit diagram that that is in the subcircuit shown in Figure 19;

Figure 21 shows an adder subcircuit that is in each subcircuit units shown in the subcircuit diagram of Figure 20;

Figure 22 shows the circuit logic of a sub-adder that is used to construct the adder subcircuit in Figure 21

Figure 23 is the Truth Table for the logic circuit of Figure 22;

Figure 24a to Figure 24g is a series of illustrations showing how the output of an adder such as that of the subcircuit shown in Figure 19 is calculated for every bit position in the output substrings concurrently;

Figure 25 shows a subcircuit for a de-converting the sum calculated by the adder of Figure 19 from the generalized format into 2's complement binary string format;

Figure 26 is the logic circuit in the subcircuit of Figure 25;

Figure 27 is the Truth Table for the logic circuit of Figure 26;

Figure 28 is the Truth Table of Figure 26;

Figure 29a de-converts a 4-bit generalized binary number to a 5-bit classical 2's complement number;

Figure 29b De-conversion circuit for generate, propagate and sum similar to a carry lookahead circuit;

Figure 30a is a variation of the flowchart in Figure 5;

Figure 30b shows a subcircuit for removing redundant bits from the substrings of a generalized binary number;

Figure 30c is a Truth table for the logic circuit of Figure 30b;

Figure 31 is a higher level subcircuit for rearranging or changing the zeros and nonzeros in the binary strings output from the circuit of Figure 6 for full

optimization, used in the other one of the embodiments referred to in Figure 5;

Figure 32 shows a subcircuit in the block circuit diagram of Figure 31;

Figure 33 is the subcircuit diagram of each circuit unit in the subcircuit of Figure 32;

Figure 34 is the logic circuit in the subcircuit diagrams of Figure 33;

Figure 35 is the Truth Table (Table 6a) for the subcircuit diagram of Figure 33;

Figure 36 is an alternative subcircuit for rearranging, adding or removing the zeros and nonzeros bits in binary strings output by the subcircuit of Figure 6 for semi-optimization, used in an embodiment referred to in Figure 5;

Figure 37a is the logic circuit in the subcircuit diagrams of Figure 36;

Figure 37b is the Truth Table (Table 6b) of step (a) of a variation of the process of Figure 36.

Figure 37c is the Truth Table (Table 6c) of a step (b) which follows the step (a) of the Truth Table 6b in Figure 37b.

Figure 38 is the Truth Table for the subcircuit diagram of Figure 36;

Figure 39 is an alternative subcircuit for rearranging, adding or removing the zeros and nonzeros bits in binary strings output by the subcircuit of Figure 6 for semi-optimization, used in an embodiment referred to in Figure 5;

Figure 40 is the logic circuit in the subcircuit diagrams of Figure 39;

Figure 41 is the Truth Table for the subcircuit diagram of Figure 39;

Figure 42 is a subcircuit for processing the output of the subcircuit of Figure 39;

Figure 43 is the logic circuit in the subcircuit diagrams of Figure 42; and

Figure 44 is the Truth Table for the subcircuit diagram of Figure 42.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

An embodiment of the invention relates to a multi-bit adder for adding up two binary strings, i.e. the augend and the addend, wherein every two bits each in

the same significant place or the same position on the respective one of two binary strings are added up concurrently, in multiple parallel processes that take place all at once. The embodiment relates mainly to addition, although the embodiment may be modified to perform subtraction, as will be explained.

The embodiment also relates to logic circuits that pre-treat the augend and the addend, by converting the augend and the addend into a format that allows the converted augend and the addend to be summed up by parallel processes. The conversion comprises two steps, a generalization step and an optimization step. Conversion of each one of the augend and addend produces what is termed here “a generalized” and “an optimized” pair of binary strings, the characteristics of which will be described in detail in the following paragraphs. The format of this binary string is such that there is no carry-over value to be passed on continuously to the bits in the more significant places, or rather the carry-over will not perpetuate but be terminated within a few bits downstream. In practice, one may consider that the format of this binary string extinguishes or terminates carry-over bits, thereby obviating the need of carry-over bits to be passed on continuously.

Generally, the optimization algorithm removes all instances of neighbouring nonzero bits from the augend and addend, such that there is at least one zero bit between every two nonzero bits. Any carry-over from adding up the augend and addend is caught by the zero bit in the next more significant place, and therefore does not propagate. However, the optimization algorithm must preserve the original values of the augend and addend for the addition to give the correct sum, by applying to what is called herein the *Equivalence Principle*.

Figure 5 is a 5-step flowchart of exemplary processes comprised in the embodiment. The steps are executed by different logic circuits. Drawings of the logic circuits relevant to each step are indicated in adjacent textboxes. Any

functional-equivalents of these logic circuits can be designed and implemented in digital processors.

The examples given of the different embodiments in this description are mostly of 8-bit binary strings. Actual products are likely to be configured for binary strings of much larger lengths, such as 16-bit strings, 32-bit strings or 64-bit strings.

In the First Step 501 of the flowchart, an augend and an addend in the ordinary 2's complement format are provided. It should be noted that augends and addends are also referred to collectively as "addends" where convenient.

In the Second Step 503, however, binary strings which are the addends are converted into a format called the Generalized-Binary-String Format herein. The converted binary string is called a generalized-binary-string.

In the Third Step 505, the generalized-binary-strings are treated by a process called Optimization, which removes all instances of neighbouring nonzeros in order to prevent propagation of carry-over bits. Optimization of generalized-binary-strings produces optimized-and-generalized-binary-strings.

Optimization can be done by any algorithm and logic circuit designed according to basic concepts that are described in the following paragraphs. There are two main varieties, however, one of which is called the Full-optimization, and the other Semi-Optimization.

In the Fourth Step 507, the addends, now formatted into optimized-and-generalized-binary-strings, are fed as input to the adder. The output sum is generalized-binary-string, and may comprise adjacent nonzeros and therefore no longer optimized.

In the Fifth Step 509, the sum is de-converted from being a generalized-binary-string into the ordinary 2's complement binary string.

Numbers to be processed by logic circuits are always in strings of bits, i.e. a binary string, and each bit is either 1 (nonzero) or 0 in value. Generally, a binary string can represent a positive value or a negative value. There is no particular nomenclature for a binary string representing a positive value. However, if the binary string represents a negative value, the binary has to be in a signed format called the 2's complement format. This is because it would not do to take the binary string representing a positive value and, for want of a better illustration, just using one of the bit in the binary as a negative sign.

For example, the number 1 in 16-bit binary format is 0000 0000 0000 0001. However, the number -1 in negatively signed 2's complement format is 1111 1111 1111 1111. Also, the number 27 in 16-bit binary format is 0000 0000 0001 1011, but the number -27 is 1111 1111 1110 0101 in negatively signed 2's complement format.

To be precise, all conventional binary strings are already in the 2's complement format. It is just a matter of convenience that binary strings representing positive values are not called 2's complement binary strings.

Generalized-Binary-Strings

The process of generalising a binary string is implemented by the subcircuit shown in Figure 6. The logic circuit represented by the block is shown in Figure 7a.

Each of the two addends is fed into a circuit block of Figure 7a. If the bit in the largest significant place is nonzero, the addend represents a negative value in 2's complement format. In this case, as illustrated in greater detail in Figure 7b,

the nonzero value bit in the largest significant place is 1 and it is transferred to the largest significant bit in a substring y. The values of other positions in the addend are transferred into an x substring. Therefore, the addend 1000 0011 is split into a first binary string, 0000 0011, and a second binary string, 1000 0000. These split binary strings are called substrings for distinction. The first substring 0000 0011 is denoted by x, while the second substring 1000 0000 is denoted by y. The value of each substring is read in the same way as any binary string. Therefore, the two binary substrings represent two values split from the addend's value and x - y gives the value of the addend. Typically, the substrings have the same string length as the original addend.

Among the many possible pairs of x and y substrings, there are some pairs in which both substrings comprise nonzero bits which are all interleaved by at least one zero bit. The process for finding some of such substrings is called optimization, and is described in detail in a following section of this description. Optimized substrings of both the addends can now be added up according to the bit position and any carry-over from adding up two nonzeros will be caught by the adjacent more significant zero bit, such that there is no rippling of carry through the bits.

Preferably, x represents a positive value, but y represents a positive value that is to be subtracted from x. In this case, it is more succinct to say that x is a minuend and y is the corresponding subtrahend. Subtracting y from x gives the value of the original addend. In other words, by applying the Equivalence Principle, the original addend value is a checksum for acceptability of the re-arrangement of zeros and nonzeros in x and y.

More specifically, the *Equivalence Principle* states that any sub-string of *m* (≥1) 1's can be replaced by a nonzero bit 1 (in a more significant place) and an $\bar{1}$ (less significant) separated by *m*-1 zeroes. Similarly, any sub-string of *m* (≥1) $\bar{1}$'s can be replaced by a bit $\bar{1}$ (in a more significant place) and a 1 (less

significant) separated by *m*-1 zeroes. Here 1 represents a “1” in the x substring while $\bar{1}$ represents a “1” in the y substring, where the y substring is to be subtracted from the x substring.

Care should be taken to avoid referring to y as a negative number. This is because a true negative value has to be represented in signed 2’s complements, which comprises many adjacent nonzeros, which is avoided by the optimization process.

The addend is considered “generalised” because the addend value is no longer represented by one specific, single, binary string. That is, for the same string length, the arrangement of the zeros and nonzeros in the pair substrings can be changed and re-arranged, as long as the sum of the two substrings remains equal to the addend value. This demonstrates the Equivalence Principle that underlies every variation of the generalisation process. However, note that the substrings x and y are allocated string length one bit longer than the nonzero string length of the original value of addend. This allows x to have a value bigger by one binary order, so that x can be a minuend. Subtracting a suitable y value from x gives the original addend value.

The concept and circuits disclosed throughout this invention for addition of two generalized binary numbers can be easily adapted to perform subtraction of one generalized binary number (i.e. subtrahend) from the other generalized binary number (i.e. minuend). After generalisation of the binary numbers, only the values of x and y of the subtrahend need to be interchanged, such that the operation of the adder becomes a subtraction, i.e., to subtract the subtrahend from the minuend becomes to sum up the minuend with the modified subtrahend (having x and y interchanged).

The following paragraphs will be confined to the case of x being a substring having the role of a minuend and y being a substring having the role of a subtrahend. For convenience when describing a nonzero bit in y, the following

shorthand is used: $\bar{1}$, a bar over a digit 1 is used to denote any nonzero bit in the y substring, or -1 as y is the subtrahend (there is a second shorthand used in the truth tables of this specification, which should not be confused with this shorthand).

Therefore, x =0001 0000 and y =0000 0001 may be written in a single line like this, 0001 000$\bar{1}$. Here $\bar{1}$ in the rightmost string position means that specific bit in the y substring has a nonzero value. Similarly, the bit with 1 in the x substring means the particular bit position has a nonzero value.

An example of generalizing the number 31 is given here. The conventional 2's complement format for number 31 in 8-bit binary format is 0001 1111. On generalization, 0001 1111 is re-expressed as two substrings, where x = 0001 1111, and y = 0000 0000. Upon applying the *Equivalence Principle*, it becomes x = 0010 0000, and y = 0000 0001.

In the recommended shorthand, the two substrings may be written as 0010 000$\bar{1}$, where $\bar{1}$ is 1 in the y substring and in the rightmost bit position, and 0010 0000 is in the x substring and 1 in the 6th-bit position.

As 0010 0000 is 32 (decimal system) and 0000 0001 is 1 (decimal system), $x - y$ in the decimal system is 32-1 = 31.

An example of generalizing the number 23 is given here. The conventional 2's complement format for number 23 in 8-bit binary format is 0001 0111. On generalization, 0001 0111 is re-expressed as two substrings, where x = 0001 0111, and y = 0000 0000. Upon applying the *Equivalence Principle*, it becomes x = 0010 1000, and y = 0001 0001. In the recommended shorthand, the substrings may be written as 001$\bar{1}$ 100$\bar{1}$.

As 0010 1000 = 40 and 0001 0001 is 17, $x - y$ = 40 – 17 = 23.

A further example of generalizing the number -23 is given here. The conventional 2's complement format for number -23 in 8-bit binary format is 1110 1001 (note that the "1" in the most significant digit indicates that it is a negative number). On generalization according to Figure 6 and Figures 7, 1110 1001 is re-expressed as two substrings, where x = 0110 1001, and y = 1000 0000. In the recommended shorthand, the substrings may be written as $\bar{1}$110 1001.

As 0110 1001 = 105 and 1000 0000 is 128, $x - y$ = 105 – 128 = -23.

The concept of generalisation with one substring being a minuend and the other substring a subtrahend can be expressed as shown below.

After the two numbers to be added up are each converted into generalized-Binary-String-format, there will be four substrings, x_1, y_1, x_2 and y_2.

In mathematical expression, an n-bit binary string n_w can be represented as

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

where

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

< , > indicates a set of possible values for the variable. Note that $\bar{1}$ is shorthand for a nonzero bit in the negative substring, as described in the above paragraphs.

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:

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

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

A semi-optimized-generalized-binary-string n_w

The substrings produced by generalising any of the addends only split the value of the addend into two values represented by binary substrings. Optimization is a process to remove all instances of neighbouring nonzeros in the substrings, by changing the positions and the number of zeros and nonzeros in substrings while observing the Equivalence Principle such that the sum of the substrings gives the value of the addend.

The preferred optimization strategy is called semi-optimization, as opposed to a full-optimization which is described in a further section in this description.

Semi-optimization of the two substrings of a generalized addend produces two substrings having the following characteristics, which defines semi-optimization:

1. every two nonzero bits in each substring are separated by at least one zero bit;

2. a nonzero bit can be immediately next to another nonzero bit if one nonzero bit is in one substring and the other nonzero bit is in the other substring;

3. There is no limitation to the number of zero bits.

Examples and negative-examples of semi-optimized-generalized-binary-strings:

Two substrings in semi-optimized format:

x	0	0	0	0	1	0	0	0
y	0	0	0	0	0	0	0	0

- No nonzero bit which is immediately next to another bit of exactly the same value, in the same substring.
- No nonzero bit in the same position on the other substring.

5

Two substrings in semi-optimized format:

x	0	0	0	0	1	0	0	0
y	0	0	0	1	0	0	0	0

- No nonzero bit which is immediately next to another bit of exactly the same value, in the same substring.
- No nonzero bit in the same position on the other substring.
- Possible to have a nonzero bit in immediately adjacent positions on the other substring.

Two substrings in semi-optimized format:

x	0	0	1	0	1	0	0	0
y	0	1	0	1	0	1	0	0

- No nonzero bit which is immediately next to another bit of exactly the same value, in the same substring.
- No nonzero bit in the same position on the other substring.
- Possible to have a nonzero bit in immediately adjacent positions on the other substring.

In other words, a semi-optimized pair of binary substrings is such that all nonzero digits in each of the resulting semi-optimized positive, x substring, and negative, y substring, are separated by at least one zero. Alternatively, no two consecutive 1's (i.e. all in x substring) and no two consecutive $\overline{1}$'s (i.e. all in y substring) are present, but a 1 can be preceded and/or succeeded by an $\overline{1}$, and vice versa.

The following is an example of the semi-optimization process. The substrings are not optimized because there are two nonzeros in the least-significant first and second bit positions.

x = 3	0	0	0	0	0	0	1	1
y = 0	0	0	0	0	0	0	0	0

The above substrings x and y were obtained from number 3, which is 00000011 in the conventional or 2's complement binary format.

By applying the Equivalence Principle, the semi-optimization process coverts the substrings into the following. There x and y substrings of each addend are optimized independently, and without referring to the other addend.

x = 4	0	0	0	0	0	1	0	0

y = 1	0	0	0	0	0	0	0	1

Shorthand for the above is 0000 010$\bar{1}$.

Alternatively, the two substrings may be re-expressed in the following way, which provides that x-y = 5 – 2 = 3, also preserving the value of 3.

x = 5	0	0	0	0	0	1	0	1
y = 2	0	0	0	0	0	0	1	0

Shorthand for the above is 0000 01$\bar{1}$1.

It can be a choice in other embodiments to semi-optimize 011 into 1$\bar{1}$1.

The following is a second example of the semi-optimization process.

x = 7	0	0	0	0	0	1	1	1
y = 0	0	0	0	0	0	0	0	0

5

A first possible semi-optimisation is the following substrings of 8 – 1 = 7

x = 8	0	0	0	0	1	0	0	0
y = 1	0	0	0	0	0	0	0	1

Shorthand for the above is 0000 100$\bar{1}$.

A second possible semi-optimisation is the following substrings of 9 – 2 = 7

x = 9	0	0	0	0	1	0	0	1
y = 2	0	0	0	0	0	0	1	0

Shorthand for the above is 0000 $10\bar{1}1$.

In a third example to semi-optimise the following.

x = 15	0	0	0	0	1	1	1	1
y = 0	0	0	0	0	0	0	0	0

The preferred semi-optimisation is the following.

x = 16	0	0	0	1	0	0	0	0
y = 1	0	0	0	0	0	0	0	1

Shorthand for the above is 0001 $000\bar{1}$.

The above three examples of semi-optimized strings illustrates that many possible semi-optimized substrings are possible as outputs for any given input.

There can be many ways of semi-optimizing a pair of strings representing a value in the generalized format. The most straight forward way is to apply an algorithm to manipulate the bits in each pair of substrings. In this embodiment, the bit manipulation is remotely similar to the Booth's Multiplication Algorithm, a well-known algorithm that gives the product of a binary multiplicand and a binary multiplier by moving their bits. However, this is where the similarity ends. Booth's multiplication algorithm is used to perform multiplication on a single binary string. In contrast, the bit manipulation used in the embodiment is not to give a product

or a sum, but merely to remove all instances of neighbouring nonzeros in each of the substrings.

However, the bit manipulation algorithm of the embodiment must adhere to the Equivalence Principle, such that a pair of substrings representing a numerical value may only be transformed into a different pair of substrings (having different number and arrangement of zeros and nonzeros) which still preserve the same numerical value. The desired transformed pair of substrings should have no neighbouring nonzeros in each of both the x substring and the y substring, and have the characteristics defined for a semi-optimized pair of substrings.

Figure 8 is a subcircuit for semi-optimization of 8-bit generalised binary strings into 9-bit semi-optimized generalised binary strings. Figure 9 shows that the circuit block of Figure 8 comprises three blocks of subcircuits, labelled with codenames Vin8o08-31, Vin8o08-32 and Vin8o08-33.

VinSo08-31 is shown in greater detail in Figure 10, which is made up of a plurality the basic subcircuit shown in Figure 11.

VinSo08-32 is shown in greater detail in Figure 13, which is made up of a plurality the basic subcircuit shown in Figure 14.

VinSo08-33 is shown in greater detail in Figure 16, which is made up of a plurality the basic subcircuit shown in Figure 17.

Advantageously, semi-optimization of substrings using the embodiment of the digital block circuit of Figure 8 can be performed in parallel for every bit position in the substrings, and comprises the following three steps:

Semi-optimization step 1. In relation to subcircuit VinSo08-31, this step eliminates any strings of $\overline{1}$ by applying the *Equivalence Principle* according to

following equations, with the logic in Truth Table 1 shown as Figure 12 and the logic circuit shown in Figure 11:

$$x_i' = x_i \bar{y}_{i-1} + y_i \bar{y}_{i-1} \quad (1a)$$

$$y_i' = \bar{x}_i \bar{y}_i y_{i-1} \quad (1b)$$

In other words, applying the equations to the pair of substrings x and y removes any immediately neighbouring nonzero values from the substring y.

As mentioned in a preceding paragraph, the truth tables in this specification use a further shorthand notation Z, which is also to represent the substring pairs of x and y, in the following way. That is, the two substrings of a generalized binary string can be written in a binary number representation, with one string representing a positive binary number and the other string a negative binary number, with bits in both strings take the values of 0 or 1. In general, a n-bit generalized binary number n_w can be represented as

$$\begin{aligned} z_w &= x_w - y_w \\ z_{n-1}z_{n-2}z_{n-3}\ldots z_2 z_1 z_0 &= x_{n-1}x_{n-2}x_{n-3}\ldots x_2 x_1 x_0 - y_{n-1}y_{n-2}y_{n-3}\ldots y_2 y_1 y_0 \\ &= \begin{pmatrix} x_{n-1}x_{n-2}x_{n-3}\ldots x_2 x_1 x_0 \\ y_{n-1}y_{n-2}y_{n-3}\ldots y_2 y_1 y_0 \end{pmatrix} \end{aligned} \quad (2)$$

where $z_i = \langle \bar{1}, 0, 1 \rangle$, $x_i = \langle 0, 1 \rangle$ and $y_i = \langle 0, 1 \rangle$, in which $i = 0, 1, \ldots, n-1$. Here,

$$\begin{aligned} &z_i = \bar{1} \Leftrightarrow x_i = 0\,;\, y_i = 1 \\ &z_i = 0 \Leftrightarrow x_i = 0\,;\, y_i = 0 \quad \text{or} \quad x_i = 1\,;\, y_i = 1 \\ &z_i = 1 \Leftrightarrow x_i = 1\,;\, y_i = 0 \end{aligned} \quad (3)$$

The weight of each bit is unity or a positive-integer power of 2. Hence, z_w can also be expressed as

$$z_w = \sum_{i=0}^{n-1} 2^i z_i = \sum_{i=0}^{n-1} 2^i x_i - \sum_{i=0}^{n-1} 2^i y_i = \sum_{i=0}^{n-1} 2^i \left(x_i - y_i\right) \tag{4}$$

The domain of z_w covers a range from $-\left(2^n - 1\right)$ to $2^n - 1$. This alternate generalized binary number system is also applicable for the 2's complement numbers.

Semi-optimization step 2. In relation to subcircuit VinSo08-32, this step converts any cluster of $1\bar{1}$ to 1 by applying the *Equivalence Principle* again according to the following equations, with the logic in Truth Table 2 shown in Figure 15 and the logic circuit shown in Figure 14;

$$x_i'' = x_i' \bar{y}_{i-1}' + x_{i+1}' y_i' \tag{5a}$$

$$y_i'' = \bar{x}_{i+1}' y_i' \tag{5b}$$

Semi-optimization step 3. In relation to subcircuit VinSo08-33, this step eliminates any strings of 1 by applying the *Equivalence Principle* according to the following equations, with the logic in Truth Table 3 shown in Figure 18 and the logic circuit in Figure 17;

$$\widetilde{X}_i = \bar{x}_i'' \bar{y}_i'' x_{i-1}'' \tag{6a}$$

$$\widetilde{Y}_i = x_i'' \bar{x}_{i-1}'' \bar{y}_{i-1}'' + y_i'' \bar{x}_{i-1}'' \bar{y}_{i-1}'' \text{ or simply } \tilde{Y}_i = x_i'' \bar{x}_{i-1}'' + y_i'' \bar{x}_{i-1}'' \tag{6b}$$

where $\widetilde{X}_i$ and $\widetilde{Y}_i$ denotes the semi-optimized generalised binary strings of the addend.

In other words, applying the equations to the pair of substrings x and y removes any immediately neighbouring nonzero values from the substring x.

The order of executing step 1, step 2, and step 3, is important and should be adhered to strictly.

Both the generalised binary strings of the addends to be added up are each semi-optimized by step 1, step 2 and step 3, to produce two pairs of semi-optimized generalised binary substrings.

Adding up the optimized-generalised-binary-substrings

Subsequently, the two pairs of 9-bit semi-optimized generalised binary substrings x_1, y_1, x_2, y_2 are fed into an adder shown in Figure 19 to be summed up.
Figure 20 shows a circuit diagram which is the subcircuit of Figure 19 in greater detail. The circuit diagram comprises ten interconnected subcircuits which can receive and add up 9-bit substrings.

Figure 21 shows one of the subcircuits in Figure 20 in greater detail. The inputs to this adder are the four substrings of the two semi-optimized and generalised addends, and the output is a 10-bit generalised binary string having an x output substring and a y output substring.

Figure 22 is the logic circuit of the adder, and two such adders have to be used as shown in Figure 21 to produce the sum in the generalised format, comprising of two substrings.

There are, however, a 4-bit input to the logic circuit in of Figure 22 which has only one output bit. Similarly, an 8-bit input to the subcircuit in Figure 21 is shown.

Actually, the summation only needs two corresponding bits of the addends, each from substrings x_1 and x_2, to produce an output bit X. Similarly, two corresponding bits each from substring y_1 and y_2, are summed to produce another output bit Y. The output bits X and Y form the resultant sum for the generalized binary string. The adder takes adjacent bits of the positions i and i - 1 because one of the bits is zero as ensured by the semi-optimization process. These interleaving zero bits are good for absorbing carry over values and stemming propagation thereof.

Also, as the bits immediately adjacent a nonzero bit are always zero bits, as ensured by the semi-optimization process, taking an extra adjacent bit into an addition operation does not affect the output value at all.

In Figure 20, there are ten subcircuits in each of which is the adder shown in Figure 21. The subcircuits of Figure 20 operate at the same time, as all the logic gates in the circuit are simply reactive to high voltage applied to the input leads.

There is no propagation or rippling of any carry-over from one adder to another, each adder just calculates an output for the corresponding bit position by taking the input values in the relevant position, independently of the other adders in the circuit.

As there is no carry over that could propagate or ripple through entire length of the binary input strings, upscaling the circuit diagrams only entails adding more adder subcircuit to the existing configuration.

In contrast, logic circuits of the prior art such as the four-bit ripple adder or the carry look-ahead adder becomes more complex exponentially every time anyone want to double the length of the typical binary strings.

Figure 24a to Figure 24g show how the four optimized general format binary strings of the two addends are added up. The process is quite intuitive at this stage. Each adder subcircuit in Figure 20 takes only the relevant input bits and produce the output bits for a specific position in the output string. Similarly, the processes illustrated in Figure 24a to Figure 24g all take place in parallel.

The substrings that are added in Figure 24a to Figure 24g are only 6-bit in length and it produces a 7-bit output. There is no difference explaining the embodiment with 7-bit string length and with 8-bit string length. The use of the 6-bit is just an arbitrary choice for example.

Every adder in Figure 20 adds up the bit in a pre-determined position in the x_1 substring and the bit in the same position in x_2 substring, and gives the value of the bit in the same position in an X substring for output. The same is done to the y_1 substring and y_2 substring to produce the Y substring output.

Figure 24a relates to position i, which counts from i = 0. The bit x_{10} in the x_1 substring and the bit x_{20} in the x_2 substring are added up. Similarly, the bit y_{10} in the y_1 substring and the bit y_{20} in the y_2 substring are added up. However, the output is just zero in position i = 0 in the X_o substring and Y_o substring.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{10} = 0	nil	x_{20} = 0	nil	X_0 = 0
y_{10} = 1	nil	y_{20} = 1,	nil	Y_0 = 0

This is because the logic circuit in Figure 22 requires two bits from the x_1 substring, two bits from the x_2 substring, two bits from the y_1 substring, two bits from the y_2 substring.

Figure 24b relates to i = 1, bits x_{10}, x_{11} from x_1 substring, bits x_{20}, x_{21} from x_2 substring, bits y_{10}, y_{11} from y_1 substring, bits y_{20}, y_{21} from y_2 substring are used to provide the output according to the truth table of Figure 23.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{11} = 0	x_{10} = 0	x_{21} = 0	x_{20} = 0	X_1= 0
y_{11} = 0	y_{10} = 1	y_{21} = 0,	y_{20} = 1	Y_1 = 1

Figure 24c relates to the bits in position i = 2. The bits x_{11} and x_{12} in the x_1 substring and the bits x_{21} and x_{22} in the x_2 substring are added up, which give the value of X_2 in the output substring. The corresponding operation applies to the y_1 and y_2 substrings to produce the Y_2 in the output substring.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{12}= 0	x_{11} = 0	x_{22} = 0	x_{21} = 0	X_2 = 0
y_{12} =0	y_{11} = 0	y_{22} = 0	y_{21} = 0	Y_2 = 0

Figure 24d relates to the bits in position i = 3. The bits x_{12} and x_{13} in the x_1 substring and the bits x_{22} and x_{23} in the x_2 substring are added up, which give the value of X_3 in the output substring. The corresponding operation applies to the y_1 and y_2 substrings to produce the Y_3 in the output substring.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{13} = 0	x_{12} = 0	x_{23} = 0	x_{22} = 0	X_3 = 0
y_{13} = 0	y_{12} = 0	y_{23} = 1	y_{22} = 0	Y_3 = 1

Figure 24e relates to the bits in position i = 4. The bits x_{13} and x_{14} in the x_1 substring and the bits x_{23} and x_{24} in x_2 substring are added up, which give the value of X_4 in the output substring. The corresponding operation applies to the y_1 and y_2 substrings to produce the Y_4 in the output substring.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{14} = 0	x_{13} = 0	x_{24} = 0	x_{23} = 0	X_4 = 0
y_{14} = 0	y_{13} = 0	y_{24} = 0	y_{23} = 1	Y_4 = 0

Figure 24f relates to the bits in position i = 5. The bits x_{14} and x_{15} in the x_1 substring and the bits x_{24} and x_{25} in x_2 substring are added up, which give the value of X_5 in the output substring. The corresponding operation applies to the y_1 and y_2 substrings to produce the Y_5 in the output substring.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{15} = 1	x_{14} = 0	x_{25} = 1	x_{24} = 0	X_5 = 0
y_{15} = 0	y_{14} = 0	y_{25} = 0	y_{24} = 0	Y_5 = 0

Figure 24g relates to the bits in position i = 6. The bit x_{15} in the x_1 substring and the bit x_{25} in the x_2 substring are added up, which give the value of X_6 in the output substring. The corresponding operation applies to the y_1 and y_2 substrings to produce the Y_6 in the output substring.

Input x_1 and y_1		Input x_2 and y_2		Output
x_{16} = nil	x_{15} = 1	x_{26} = nil	x_{25} = 1	X_6 = 1
y_{16} = nil	y_{15} = 0	y_{26} = nil	y_{25} = 0	Y_6 = 0

Since the input of four bits comes from the substrings of two generalized binary numbers and the sum is one bit and also in the generalised format. The resultant is two output bits that forms two substrings of an output sum in the generalized format.

In this section, the addition algorithm by semi-optimized general binary strings is presented. The sum is denoted by N_w whereas the semi-optimized augend and addend are denoted by $(\tilde{n}_w)_1$ and $(\tilde{n}_w)_2$.

The addition method can be expressed in the following two steps:

(a) separately adding the positive and negative components by using the following equations;

$$x_i' = (\tilde{x}_i)_1(\bar{\tilde{x}}_i)_2 + (\bar{\tilde{x}}_i)_1(\tilde{x}_i)_2 + (\tilde{x}_{i-1})_1(\tilde{x}_{i-1})_2 \quad (7a)$$

$$y_i' = (\tilde{y}_i)_1(\bar{\tilde{y}}_i)_2 + (\bar{\tilde{y}}_i)_1(\tilde{y}_i)_2 + (\tilde{y}_{i-1})_1(\tilde{y}_{i-1})_2 \quad (7b)$$

where the logic truth and logic circuit are presented in Table 12 in Figure 23 and Figure 22.

Deconversion of the two substrings into a single binary string in 2's Complement

The two substrings X and Y given as sum of the addends, denoted as substrings x and y as the inputs for the next-level circuit, has to be deconverted into the conventional, binary format of a signed single string in 2's complement, that can be read and used in other regular processes.

A general deconversion logic circuit resembles the conventional carry-lookahead circuit, but the details are not exactly identical. The conventional propagate bit and generate bit and the final sum bit in a conventional carry-lookahead circuit is replaced with the subcircuit in Figure 25. Figure 26 shows the corresponding logic circuits inside the subcircuit, while Figure 27 and Figure 28 show the corresponding Truth Table 4 and Truth Table 5a.

A pair of substrings having 8-bit string length requires eight subcircuits. This is because each subcircuit is only able to work on two bits that have in the same

position in the two substrings. That is, one bit in the x substring and another bit in the y substring, but in the same bit position.

If the output of the adder is as follows:

| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

x = 84

| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

y = 17

$x - y$ = 84 – 17 = 67

The first step in deconversion is to replace all instances of nonzeros, that are in the same position in both substrings, with zeros. The nonzeros cancel out in value anyway. See the fourth bit which is 1 in both substrings above, and the substrings right below which are removed of the two bits.

| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

x = 68

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

y = 1

x – y = 68 – 1 = 67

The two substrings can just be combined into a single binary string by using the de-conversion circuit.

| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

= 67

In mathematical expressions, to deconvert a substring of (*n*-1)-bit z_w into a *n*-bit 2's complement binary string $z_w^{2+}\left(\equiv n_w^{2+}\right)$, the method includes an initial bit simplification by replacing $x_i = y_i = 0$ if $x_i = y_i = 1$.

Subsequently, the two substrings so treated are added up using regular adder circuits. The well-known Carry Lookahead carry over predictive logic is modified to suit the deconversion requirement here, where that the conventional generates (g_i) and propagates (p_i) are replaced by the following logic.

$$p_i = \bar{x}_i \bar{y}_i \tag{8a}$$

$$g_i = y_i \tag{8b}$$

The truth logic of the above is shown as Truth Table 4 in Figure 27, relating to output of generates and propagates for deconversion to 2's complement numbers. That is, Truth Table 4 relates to combinations of input bits that predict a carry-over bit will be generated, and conditions that means a carry-over bit will be propagated. Note however, this production of generates and propagates, and the due process of group generates and group propagates similar to the classical Carry Lookahead circuit, from the output *X* substring and the *Y* substring is a serial process. It becomes more complicated when the number of bits increases. Therefore, the deconversion procedure for producing the 2's complement number is not a parallel process.

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

$$X_i = t_i = \bar{c}_i x_i + c_i p_i + \bar{c}_i y_i \text{ for } i = 0,1,2,\cdots,(n-1) \tag{9a}$$

$$Y_n = c_n \tag{9b}$$

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 truth logic for the above is presented in Truth Table 5a in Figure 28, relating to the “Final stage summation” logic $i = 0,1,2,\cdots,(n-1)$ for deconverting the pair of binary substring which is the sum into a normal 2’s complement binary format, and the logic circuit is given in Figure 26.

Figure 29a shows the de-conversion of a 4-bit generalized binary number to a 5-bit classical 2’s complement number. As shown in Figure 29b, two substring of the generalised binary string format of 4-bit is 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 logic circuits shown in Figure 26.

The output of the array from Figure 29b will be a 5-bit single string classical 2’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 2’s complement number. It is noted the de-conversion of a generalized binary number will produce a classical 2’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 29b 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 2’s Complement expression.

Deconversion of only the two substrings only necessary at the end of all required addition operations

As mentioned, the last step of this embodiment is deconversion which includes the adding up of the x substring and the y substring using classical Full Addition algorithm, using the described modified carry-lookahead is just one of the many ways. However, as shown in Figure 30a, this step 509 can be postponed, until all processes that require the addends' output sum to be involved in more addition and subtraction operations, in step 511, step 513 and step 515, have been completed (as decided in step 508). At that stage the memory allocated to the generalized binary string, i.e. the final pair of substrings, is dedicated memory.

However, for every next iteration of the addition process, to add a new number (in the generalized binary format) to the resultant output of the last addition iteration (which is also in the generalized binary format), these two numbers have to be optimized. In particular, the output of an addition will have lost the optimized format even though the output is still in the generalized format. To optimize the output of the last iteration of addition, however, the pair of binary strings has to be treated first to remove the redundant bits.

The removal of redundant bits from the generalized binary substrings can be expressed as the following equations.

$$X_i = x_i' \overline{y}_i' \tag{9c}$$

$$Y_i = \overline{x}_i' y_i' \tag{9d}$$

"Redundant bits" come in pairs, and are defined as a nonzero in a bit position on the x substring and, at the same time, a nonzero bit in the same position on the y substring. These nonzero bits have no value intrinsically other than being in the same position on both the x and y substrings which cancel each other.

In an example of a x substring and a y substring below, the first bit of each substring has a nonzero value. These are redundant bits as the nonzero values are in the same position in both substrings.

x	0	0	0	0	0	1	1	1
y	0	0	0	0	0	0	0	1

Therefore, before optimization (whether semi-optimization or full optimization) these redundant bits are removed by converting the nonzero value into zero value, giving the following.

x	0	0	0	0	0	1	1	0
y	0	0	0	0	0	0	0	0

The above pair of substrings may now be optimized.

Therefore, a pair of redundant bits can be removed by simply converting the bits into zeros, without affecting the value represented and keeping to the Equivalent Principle.

Therefore, in the flowchart of Figure 30a, if there are many numbers to sum up, the resultant sum of the last iteration of addition is first treated to remove any redundant bits, at step 517, before being optimized, at step 515. The new addend to be added to the resultant sum of the last iteration of addition is also optimized, at step 515. There is no need for the new addend to be treated to remove redundant bits because the conversion of the addend into generalized

binary number format has already pre-empted and prevented any occurrence of redundant bits.

The truth logic for removal of redundant bits is presented in Truth Table 5b in Figure 30c, and the logic circuit is given in Figure 30b.

A second embodiment using full-optimization of the generalized binary substrings

As shown in the flowchart in Figure 5, there is an alternative to the semi-optimisation process for removing nonzeros immediately neighbouring a nonzero, called a full-optimization process, and described in Figure 31, Figure 32, Figure 33 and Figure 34.

A substring that meets the requirements to be considered fully-optimized will have already met the requirements to be considered semi-optimized.

However, the converse is not true. The full-optimization is a subset of semi-optimization.

Figure 31 is a higher level subcircuit for fully-optimizing generalized binary substrings. Figure 32 is a circuit diagram interconnecting a plurality of subcircuits, the circuit diagram underlying the higher level subcircuit in Figure 31. Figure 33 shows a circuit diagram which makes up each of the interconnected subcircuits in Figure 32. Figure 34 shows logic circuit of Figure 33.

Upon being fully-optimized, each pair of the two substrings produced by generalising binary strings representing the corresponding addend, are

- a pair of substrings that add up to the value of the addend, and from which the initial values of the substrings split off;

- meet the requirements for fully-optimized substrings, which include:
- no two nonzero bits that are in immediately adjacent positions in the same substring, i.e. any two nonzero bits in the same substring must be separated by at least one zero bit;
- no nonzero bits in the same position on both substrings;
- no two nonzero bits in immediately adjacent positions even if the two nonzeros are on different substrings.

A pair of generalised binary substrings that has been fully-optimized is shown below. That is, the following is an example of a fully-optimized pair of substrings.

Two substrings x and y both full-optimized

Substring x	0	**1**	0	0	0	0	**1**	0	1
Substring y	0	0	0	**1**	0	0	0	0	0

- No nonzero bit which is immediately next to another nonzero bit, whether in the same substring or the other substring.
- No nonzero bit in the same position on the other substring.

If the two substrings are placed next to each other and aligned by the bit positions, one could see five zero bits around any nonzero bit. Unless, if the nonzero bit is the first bit or the last bit of a substring, then there would be just three immediately adjacent zero bits.

The following is an incorrect, negative example of a non-fully-optimized pair of substrings. This is because there must not be two nonzero bits in adjacent positions, even though the two nonzero are on different substrings. This is where the full-optimization differs from semi-optimization, semi-optimization allows two nonzeros in adjacent bit positions provided that the two nonzeros are on different substrings.

Two substrings x and y NOT full-optimized

Substring x	0	1	0	0	0	0	0	0	0
Substring y	0	0	1	0	0	0	0	0	0

In other words, in the substrings of a fully optimized-generalized-binary-string any two nonzero bits that are separated by at least one zero. In contrast, the semi-optimizing algorithm ensures that every two nonzero bits (i =, 1, ..., *n*) are separated by at least one zero bit in the same substring. The optimization algorithm must also apply the *Equivalence Principle*.

The logic circuits in Figure 31, Figure 32, Figure 33 and Figure 34 are produced based on an algorithm for full-optimization of a pair of substrings, which can also be expressed in mathematical notation.

The following equations relate to the full-optimization algorithm implemented in Figure 31, Figure 32, Figure 33 and Figure 34.

$$\widehat{X}_{i-1} = \overline{x}_i x_{i-1} \tag{10a}$$

$$\widehat{Y}_{i-1} = \overline{y}_i y_{i-1} \tag{10b}$$

$$\widehat{X}_i = \left(\overline{x}_{i+1}\overline{y}_{i+1}x_i + x_{i+1}y_i + y_{i+1}y_i\right)\overline{x}_{i-1}\overline{y}_{i-1} + x_{i+1}x_i x_{i-1} + x_{i+1}x_i y_{i-1} \tag{10c}$$

$$\widehat{Y}_i = \left(\overline{x}_{i+1}\overline{y}_{i+1}y_i + y_{i+1}x_i + x_{i+1}x_i\right)\overline{x}_{i-1}\overline{y}_{i-1} + y_{i+1}y_i y_{i-1} + y_{i+1}y_i x_{i-1} \tag{10d}$$

$$\widehat{X}_{i+1} = x_{i+1}\overline{y}_i\overline{x}_{i-1}\overline{y}_{i-1} + \left(\overline{y}_{i+1}x_i + x_{i+1}\overline{y}_i\right)x_{i-1} + \left(\overline{y}_{i+1}x_i + x_{i+1}\overline{y}_i\right)y_{i-1} \tag{10e}$$

$$\widehat{Y}_{i+1} = y_{i+1}\overline{x}_i\overline{x}_{i-1}\overline{y}_{i-1} + \left(\overline{x}_{i+1}y_i + y_{i+1}\overline{x}_i\right)y_{i-1} + \left(\overline{x}_{i+1}y_i + y_{i+1}\overline{x}_i\right)x_{i-1} \tag{10f}$$

where an overbar above x_i or y_i indicates the negation of the value, the absolute number of which represented by the binary Boolean variables.

Negative numbers

The truth of the logic circuits in Figure 31, Figure 32, Figure 33 and Figure 34 is given as Truth Table 6a in Figure 35, and is equivalent to the equations. It is easier for a reader to just consider the Truth Table instead of trying to understand the equations and see them as digital citcuits thoroughly.

That is, the logic circuit in Figure 33 can take input data and give an output result according to Table 6a in Figure 35.

The algorithm, which is recursive in nature, starts with $i = 0$ in the first cycle where z_0 and z_1 are the first two digits, and assuming $z_{-1} = 0$.

The sub-string of $z_1 z_0 z_{-1}$ transforms according to the Table 6a and the resulting sub-string $\hat{Z}_1 \hat{Z}_0 \hat{Z}_{-1}$ replaces $z_1 z_0 z_{-1}$. The second cycle $i = 1$ inputs the *updated sub-string* $z_2 z_1 z_0$ and again transforms according to Table 6a. The procedure continues recursively until the most significant digit is reached where $i = n$ in which $z_{n+1} = 0$ is completed.

It might be easier, in other words, to understand full-optimization by considering semi-optimization. In this case, each semi-optimized-generalized-binary-string can be further optimized into a fully-optimized-generalized-binary-string by an algorithm designed for

1) removing all instances of immediately neighbouring nonzero bits, such as two 1 bits in x substring, two $\bar{1}$ bits in y substring, and a 1 bit in x substring and a $\bar{1}$ bit in the y substring in the immediately adjacent position;

(no bit in the x string whether both 1 and $\bar{1}$ are, these instances can be replaced by two immediately neighbouring bits of 0 and 1; or

2) replacing bits in the above described arrangements with arrangements of immediately adjacent zero and nonzero bits 0($\bar{1}$); and

3) letting any nonzero bit in the substrings remain if positions immediately next to the nonzero bit is preceded by a zero and succeeded 0 in the same substring, and in the same position, in the preceding position and in the succeeding position in the other substring are zero bits.

Variations of semi-optimization process

The above described embodiments are sufficient for any reader to understand the operation of the embodiment. To demonstrate that there are different algorithms that can achieve optimization, and that the invention should not be limited to the above described embodiments and algorithm, two variations of the semi-optimization process are described in the following.

The first variation of a process to semi-optimize a generalized binary number n_w (a pair substrings that add up to the same numerical value of an addend), and has two steps:

(a) Any general binary number n_w can be transformed to the first-form of a partially semi-optimized binary number $\widetilde{N}_w^1$ by the following *Semi-optimization Transformation Equation*

$$\widetilde{N}_w^1 = 2n_w - n_w \qquad (11)$$

This step can be performed by finding the difference of n_w shifted a digit more significant (i.e. $2n_w$) and n_w. The first-form of the partial semi-optimization algorithm as presented in the following equations and the Truth Table 6b, as

$$\tilde{X}_i^1 = \bar{x}_i\bar{y}_i x_{i-1}\bar{y}_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1} + y_i\bar{x}_{i-1}x_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1}y_{i-2} + x_i y_{i-1}x_{i-2} \qquad (12a)$$

$$\tilde{Y}_i^1 = \bar{x}_i\bar{y}_i y_{i-1}\bar{x}_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1} + x_i\bar{y}_{i-1}y_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1}x_{i-2} + y_i x_{i-1}y_{i-2} \qquad (12b)$$

where $z_{n+1} = z_{-1} = z_{-2} = 0$ are assumed.

Equation (11) and the corresponding expanded Equation (12a) and Equation (12b) simply means converting each of the pair of substrings by multiplying a substring by 2, and subtracting from the product the substring. It has been found surprisingly that, in the result, there are remarkably few bits in positions and arrangement that does not meet the requirements of a semi-optimized substring. Hence, this method achieves near complete semi-optimization. That is, this step is imperfect and the algorithm does not guarantee full semi-optimization because it is possible to have (i) an $\bar{1}\,\bar{1}$ in $\tilde{N}_w^1$ if there exists a sub-string of $10\bar{1}$ in n_w; and (ii) an 11 in $\tilde{N}_w^1$ if there exists a sub-string of $\bar{1}01$ in n_w; which are in contradiction with the definition of semi-optimization as set out above.

Figure 37b is the Truth Table 6b according to the above equations and step (a) of this variation of the semi-optimization process or algorithm. This semi-optimization of the above equations does not guarantee full semi-optimization because it is possible to have (i) an $\bar{1}\,\bar{1}$ in $\tilde{N}_w^1$ if there exists a sub-string of $10\bar{1}$ in n_w; and (ii) an 11 in $\tilde{N}_w^1$ if there exists a sub-string of

$\bar{1}01$ in n_w; which are in contradiction with the definition of semi-optimization as set out above.

(b) To address any bits that do not meet the requirements of a semi-optimized substring, any bits in the arrangement of 11 or $\bar{1}\,\bar{1}$ in $\widetilde{N}_w^1$ is transformed to provide a completely semi-optimized number $\widetilde{N}_w^2$ as presented in the following equations and Truth Table in Table 6c:

$$\tilde{X}_i^2 = x_i\bar{x}_{i-1}\bar{x}_{i-2}\bar{y}_{i-2} + y_i y_{i-1} + x_i x_{i-2} + \bar{x}_i\bar{y}_i x_{i-1}x_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1}y_{i-2} \quad (13a)$$

$$\tilde{Y}_i^2 = y_i\bar{y}_{i-1}\bar{x}_{i-2}\bar{y}_{i-2} + x_i x_{i-1} + y_i y_{i-2} + \bar{x}_i\bar{y}_i y_{i-1}y_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1}x_{i-2} \quad (13b)$$

Figure 37c is Truth Table 6c according to the above equations. The bit manipulation expressed in Equation (13a) and Equation (13b) are based on consideration of possible combinations of nonzeros that do not meet the requirement of semi-optimization produced by the Equation (11).

Therefore, step (b) is executed to complement step (a) such that any number can be transformed to a fully semi-optimized number $\tilde{N}_w^2$, or simply $\widetilde{N}_w$.

Combining steps (a) and (b) and assuming $z_{n+1} = z_{-1} = z_{-2} = z_{-3} = 0$, the semi-optimization algorithm in Equation (12a), Equation (12b), Equation (13a) and Equation (13b) can also be combined and expressed as follows:

$$\begin{aligned}\widetilde{X}_i = {} & \bar{x}_i\bar{y}_i x_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}\bar{y}_{i-3} + x_i y_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}x_{i-3} + y_i\bar{x}_{i-1}\bar{y}_{i-1}\bar{x}_{i-2} + y_i y_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}x_{i-3} \\ & + \bar{x}_i\bar{y}_i x_{i-1}x_{i-2} + x_i y_{i-1}x_{i-2} + y_i y_{i-1}x_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1}y_{i-2}\end{aligned} \quad (14a)$$

$$\begin{aligned}\widetilde{Y}_i = {} & \bar{x}_i\bar{y}_i y_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}\bar{x}_{i-3} + y_i x_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}y_{i-3} + x_i\bar{x}_{i-1}\bar{y}_{i-1}\bar{y}_{i-2} + x_i x_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}y_{i-3} \\ & + \bar{x}_i\bar{y}_i y_{i-1}y_{i-2} + y_i x_{i-1}y_{i-2} + x_i x_{i-1}y_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1}x_{i-2}\end{aligned} \quad (14b)$$

where the positive and negative digits of n_w can be semi-optimized in a parallel operation instead of recursively as in the optimization process.

The semi-optimization outputs $\widetilde{X}_i$ and $\widetilde{Y}_i$ are dependent only on z_i, z_{i-1}, z_{i-2} and z_{i-3} of n_w.

Figure 38 is Truth Table 7 of combined step a) and step b), i.e. Equation (14a) and Equation (14b). The corresponding logic circuit is presented in Figure 36 and Figure 37a.

The second variation of a process to semi-optimize a generalized binary number n_w(a pair substrings that add up to the same numerical value of an addend), and has the following two steps:

(a) converting any sub-strings of $10\bar{1}$ and $\bar{1}01$ into $1\bar{1}1$ and $\bar{1}1\bar{1}$, respectively, in accordance with the following equations or the Truth Table 8 in Figure 41, and the corresponding logic circuit is presented in Figure 39 and Figure 40,

$$x_i' = \bar{y}_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}x_i + x_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}y_i + x_{i+1}x_i + y_{i+1}\bar{x}_i\bar{y}_i x_{i-1} + y_{i+1}x_i \qquad (15a)$$

$$y_i' = \bar{x}_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}y_i + y_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}x_i + y_{i+1}y_i + x_{i+1}\bar{x}_i\bar{y}_i y_{i-1} + x_{i+1}y_i \qquad (15b)$$

(b) applying the *Semi-optimization Transformation Equation* (11) the following equations can be obtained, with the Truth Table 9 presented in Figure 44, and logic circuit in Figure 42 and Figure 43.

$$\widetilde{X}_i = \bar{x}_i'\bar{y}_i'x_{i-1}'\bar{y}_{i-2}' + x_i'y_{i-1}'x_{i-2}' + y_i'\bar{x}_{i-1}'\bar{y}_{i-1}' + y_i'y_{i-1}'x_{i-2}' \qquad (16a)$$

$$\widetilde{Y}_i = \bar{x}_i'\bar{y}_i'y_{i-1}'\bar{x}_{i-2}' + y_i'x_{i-1}'y_{i-2}' + x_i'\bar{x}_{i-1}'\bar{y}_{i-1}' + x_i'x_{i-1}'y_{i-2}' \qquad (16b)$$

which result in the semi-optimized digits $\widetilde{X}_i$ and $\widetilde{Y}_i$.

In other words, the difference between the first and second variations of the semi-optimization process is basically a reverse of the order of the step to apply bit manipulation algorithms and the step to apply Equation (11).

The bit manipulation according to Equation (15a) and Equation (15b) In the second variation is different from the bit manipulation according to Equation (13a) and Equation (13b) in the first variation. This is because the possible combinations of nonzeros that have to be removed by bit manipulation in order to meet the requirements of a semi-optimized pair of substrings have been observed to be different between the first and second variations.

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.

All methods disclosed in this document may be implemented using logical circuits. The logical circuits described in this document (a term which may be understood here as ‘fixed logical circuits’) may be part of a larger logical circuit. The larger logical circuit may be configured to perform a computational task on an input set of data, to generate a corresponding output set of data. The element(s) (e.g. the gates) of the logic circuits may be implemented as respective region(s) of an integrated circuit, or as respective physical components.

The input data may be, for example, image data. The computational task may then be one or more of the following: image processing; image classification; object classification; pose estimation; scene recognition; image segmentation;

object localisation or detection; depth estimation; image keypoint detection; image similarity determination; or encoding the image data.

The input data may be, for example, video data comprising a sequence of a plurality of images. The computational task may then be one or more of the following: video processing; action recognition; action detection; or encoding the video data.

The input data may be, for example, audio data. The computational task may then be one or more of the following: audio processing; speech recognition; speaker recognition; a score task; a task to identify or translate the language of any spoken words in the audio data; or encoding the audio data.

The input data may be, for example, sensor data, meaning data representing the state of an environment, which may comprise one or more of video data, image data, temperature data, pressure data, distance data, or position data. The computational task may then be to generate data corresponding to an action to be performed by an agent. The agent may be, for example, a mechanical agent moving in and interacting with a real-world environment; a system for controlling the settings of a networked system; or a system for controlling an industrial facility.

CLAIMS

1. A method implemented using fixed logic circuitry of formatting a number in the form of a binary string into a format suitable for summation with another number in the same format and stemming propagation of any carry-over value resulting from the summation, the method comprising the steps of:

providing the number;

providing the binary string representing the absolute value of the number;

splitting the value of the binary string into a pair of binary substrings, wherein the sum of the pair of substrings x and y gives the value of the binary string;

re-arranging the zeros and nonzeros in the pair binary substrings x and y to:

a) produce a formatted pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$, wherein the sum of the formatted pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$, gives the value of the binary string;

b) in each one of the formatted pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$, every nonzero bit is separated from any other nonzero bit by at least one zero bit; and

c) for any same bit position i in the formatted pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$, the values are not both nonzero.

2. A method of formatting a number as claimed in claim 1, wherein the step of re-arranging the zeros and nonzeros in the pair binary substrings x and y is preceded by a step of:

converting nonzero bits in the same position in the substrings x and y to zero bits.

3. A method of formatting a number as claimed in claim 1, wherein one of the binary substring is a minuend;

the other one of the binary substring is a subtrahend; such that the sum of the pair of substrings is the difference between the subtrahend and the minuend.

4. A method of formatting a number as claimed in claim 1, wherein re-arranging the zeros and nonzeros in the binary substrings comprises the following step 1), step 2) and step 3) in the specific order of:

step 1) applying both the equations to eliminate any immediately neighbouring nonzero bits in the binary substring y

$$x_i' = x_i \bar{y}_{i-1} + y_i \bar{y}_{i-1} \quad (1a)$$

and

$$y_i' = \bar{x}_i \bar{y}_i y_{i-1} \quad (1b)$$

step 2) applying the following equations to eliminate any pairs of neighbouring nonzero bits in the specific order that a x=1, y=0 follow by a x=0, y=1 be changed to a x=0, y=0 follow by a x=1, y=0 on binary substrings x and y:

$$x_i'' = x_i' \bar{y}_{i-1}' + x_{i+1}' y_i' \quad (5a)$$

$$y_i'' = \bar{x}_{i+1}' y_i' \quad (5b)$$

step 3) applying the following equations to eliminate any immediately neighbouring nonzero bits in the binary substring x

$$\widetilde{X}_i = \bar{x}_i'' \bar{y}_i'' x_{i-1}'' \quad (6a)$$

$$\widetilde{Y}_i = x_i'' \bar{x}_{i-1}'' \bar{y}_{i-1}'' + y_i'' \bar{x}_{i-1}'' \bar{y}_{i-1}'' \text{ or simply } \tilde{Y}_i = x_i'' \bar{x}_{i-1}'' + y_i'' \bar{x}_{i-1}'' \quad (6b)$$

where $\widetilde{X}_i$ and $\widetilde{Y}_i$ denotes the value of the bits in position i in the formatted pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$.

5. A method of formatting a number as claimed in claim 1, wherein re-arranging the zeros and nonzeros in the binary substrings comprises executing the following step a) before executing the following step b);

a*) applying the following equation to the* pair of substrings x and y

$$\widetilde{N}_w^1 = 2n_w - n_w \tag{11}$$

where n_w represents *the* pair of substrings x and y;

b) applying the following equations to $\widetilde{N_w^1}$ *to obtain* $\widetilde{N_w^2}$

$$\tilde{X}_i^2 = x_i\overline{x}_{i-1}\overline{x}_{i-2}\overline{y}_{i-2} + y_i y_{i-1} + x_i x_{i-2} + \overline{x}_i\overline{y}_i x_{i-1}x_{i-2} + x_i\overline{x}_{i-1}\overline{y}_{i-1}y_{i-2} \tag{13a}$$

$$\tilde{Y}_i^2 = y_i\overline{y}_{i-1}\overline{x}_{i-2}\overline{y}_{i-2} + x_i x_{i-1} + y_i y_{i-2} + \overline{x}_i\overline{y}_i y_{i-1}y_{i-2} + y_i\overline{x}_{i-1}\overline{y}_{i-1}x_{i-2} \tag{13b}$$

where

$\widetilde{N_w^2}$ = $\tilde{X}$ and $\tilde{Y}$;

$\tilde{X}_i^2$ and $\tilde{Y}_i^2$ denote the value of the bits in position i in the different pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$.

6. A method of formatting a number as claimed in claim 1; further comprising a step of:

interchanging the contents of the pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$.

7. A method of formatting a number as claimed in claim 1, wherein re-arranging the zeros and nonzeros in the binary substrings comprises executing the following step i) before executing the following step ii);

i. converting any sub-strings of $10\bar{1}$ and $\bar{1}01$ into $1\bar{1}1$ and $\bar{1}1\bar{1}$, respectively, in accordance with the following equations

$$x_i' = \bar{y}_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}x_i + x_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}y_i + x_{i+1}x_i + y_{i+1}\bar{x}_i\bar{y}_i x_{i-1} + y_{i+1}x_i \quad (15a)$$

$$y_i' = \bar{x}_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}y_i + y_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}x_i + y_{i+1}y_i + x_{i+1}\bar{x}_i\bar{y}_i y_{i-1} + x_{i+1}y_i \quad (15b)$$

ii. applying the following equation to x' and y'

$$\tilde{N}_w^1 = 2n_w - n_w \quad (11)$$

8. A fixed logical circuit for applying step 1 of the method of claim 4 comprising a configuration having:

inputs x_i, y_{i-1}, and y_i;

(i) 2-input AND Gate 1, (ii) 2-input AND Gate 2, (iii) 3-input AND Gate 3 in parallel; and (iv) 2-input OR Gate 1;

wherein:

x_i is an input of AND Gate 1, and a NOT input of AND Gate 3;

y_{i-1} is a NOT input of AND Gate 1, a NOT Input of AND Gate 2, and an input of AND Gate 3;

y_i is an input of AND Gate 2, and a NOT input of AND Gate 3;

the output of AND Gate 1 and the output of AND Gate 2 are inputs of OR Gate 1;

the output of OR Gate 1 is a first output of the circuit, X_i; and,

the output of AND Gate 3 is a second output of the circuit, Y_i.

9. A fixed logical circuit for applying step 2 of the method of claim 4 comprising a configuration having:

inputs x_i, x_{i+1}, y_{i-1}, and y_i;

(i) 2-input AND Gate 1, (ii) 2-input AND Gate 2, (iii) 2-input AND Gate 3 in parallel; and (iv) 2-input OR Gate 1; and,

wherein:

x_i is an input of AND Gate 1;

x_{i+1} is an input of AND Gate 2, and a NOT input of AND Gate 3;

y_{i-1} is a NOT input of AND Gate 1;

y_i is an input of AND Gate 2, and an input of AND Gate 3;

the output of AND Gate 1 and the output of AND Gate 2 are inputs of OR gate 1;

the output of OR Gate 1 is the output of the circuit, X_i; and

the output of AND Gate 3 is the output of the circuit, Y_i.

10. A fixed logical circuit for applying step 3 of the method of claim 4 comprising a configuration having:

inputs x_{i-1}, x_i, y_{i-1}, y_i;

(i) 3-input AND Gate 1, (ii) 3-input AND Gate 2, (iii) 3-input AND Gate 3 in parallel; and (iv) 2-input OR Gate 1;

wherein:

x_{i-1} is an input of AND Gate 1, a NOT input of AND Gate 2, and a NOT input of AND Gate 3;

x_i is a NOT input of AND Gate 1, and an input of AND Gate 2;

y_{i-1} is a NOT input of AND Gate 2, and a NOT input of AND Gate 3;

y_i is a NOT input of AND Gate 1, and an input of AND Gate 3;

the output of AND Gate 2 and the output of AND Gate 3 are inputs of OR Gate 1;

the output of AND Gate 1 is a first circuit output, X_i; and,

the output of OR Gate 1 is a second circuit output, Y_i.

11. A fixed logical circuit for applying step a) and step b) of the method of claim 5 comprising a configuration having:

inputs x_{i-3}, x_{i-2}, x_{i-1}, x_i, y_{i-3}, y_{i-2}, y_{i-1}, y_I;

(i) 6-input AND Gate 1, (ii) 5-input AND Gate 2, (iii) 4-input AND Gate 3, (iv) 5-input AND Gate 4, (v) 4-input AND Gate 5, (vi) 3-input AND Gate 6, (vii) 3-input AND Gate 7, (viii) 4-input AND Gate 8 in parallel; and (ix) 8-input OR Gate 1;

wherein:

x_{i-3} is an input of AND Gate 2, and an input of AND Gate 4;

x_{i-2} is a NOT input of AND Gate 1, a NOT input of AND Gate 2, a NOT input of AND Gate 3, a NOT input of AND Gate 4, an input of AND Gate 5, an input of AND Gate 6, and an input of AND Gate 7;

x_{i-1} is an input of AND Gate 1, a NOT input of AND Gates 3, an input of AND Gates 5, and a NOT input of AND Gate 8;

x_i is a NOT input of AND Gate 1, an input of AND Gate 2, a NOT input of AND Gate 5, an input of AND Gate 6, and an input of AND Gate 8;

y_{i-3} is a NOT input of AND Gate 1;

y_{i-2} is a NOT input of AND Gate 1, a NOT input of AND Gate 2, a NOT input of AND Gate 4, and an input of AND Gate 8;

y_{i-1} is an input of AND Gate 2, a NOT input of AND Gate 3, an input of AND Gate 4, an input of AND Gate 6, an input of AND Gate 7, and a NOT input of AND Gate 8;

y_i is a NOT input of AND Gate 1, an input of AND Gate 3, an input of AND Gate 4, a NOT input of AND Gate 5, and an input of AND Gate 7;

the output of each of AND Gate 1, AND Gate 2, AND Gate 3, AND Gate 4, AND Gate 5, AND Gate 6, AND Gate 7, and AND Gate 8 being inputs of OR Gate 1; and

the output of OR Gate 1 being an output of the circuit, X_i.

12. A fixed logical circuit for applying step a) of the method of claim 7 comprising a configuration having:

inputs x_{i-1}, x_i, x_{i+1}, x_{i+2}, y_i, y_{i+1}, and y_{i+2};

(i) 2-input AND Gate 1, (ii) 2-input AND Gate 2, (iii) 4-input AND Gate 3, (iv) 4-input AND Gate 4, (v) 4-input AND Gate 5 in parallel; and (vi) 5-input OR

Gate 1;

wherein:

x_{i-1} is an input of AND Gate 3;

x_i is an input of AND Gate 1, an input of AND Gate 2, a NOT input of AND Gate 3, and an input of AND Gate 5;

x_{i+1} is input of an input of AND Gate 1, a NOT input of AND Gate 4, and a NOT input of AND Gate 5;

x_{i+2} is an input of AND Gate 4;

y_i is a NOT input of AND Gate 3, and an input of AND Gate 4;

y_{i+1} is an input of AND Gate 2, an input of AND Gate 3, a NOT input of AND Gate 4, and a NOT input of AND Gate 5;

y_{i+2} is a NOT input of AND Gate 5;

the outputs of AND Gate 1, AND Gate 2, AND Gate 3, AND Gate 4, and AND Gate 5 are inputs of OR Gate 1; and,

the output of OR Gate 1 is an output of the circuit, X_i.

13. A fixed logical circuit for applying step b) of the method of claim 7 comprising a configuration having:

inputs x_{i-2}, x_{i-1}, x_i, y_{i-2}, y_{i-1}, and y_i;

(i) 3-input AND Gate 1; (ii) 3-input AND Gate 2; (iii) 3-input AND Gate 3, (iv) 4-input AND Gate 4 in parallel; and (v) 4-input OR Gate 1;

wherein:

x_{i-2} is an input of AND Gate 1, and an input of AND Gate 2;

x_{i-1} is a NOT input of AND Gate 3, and an input of AND Gate 4;

x_i is an input of AND Gate 1, and a NOT input of AND Gate 4;

y_{i-2} is a NOT input of AND Gate 4;

y_{i-1} is an input of AND Gate 1, an input of AND Gate 2, and a NOT input of AND Gate 3;

y_i is an input of AND Gate 2, an input of AND Gate 3, and a NOT input of AND Gate 4;

the outputs of AND Gate 1, AND Gate 2, AND Gate 3, and AND Gate 4 are inputs of OR Gate 1; and,

the output of OR Gate 1 is an output of the circuit, X_i.

14. A fixed logical circuit for applying the method of claim 2, comprising a configuration having:

inputs x_i and y_i;

(i) 2-input AND Gate 1; and (ii) 2-input AND Gate 2 in parallel;

wherein:

x_i is an input of AND Gate 1, and a NOT input of AND Gate 2;

y_i is a NOT input of AND Gate 1, and an input of AND Gate 2; and

the output of AND Gate 1 is a first output of the circuit, X_i; and,

the output of AND Gate 2 is a second output of the circuit, Y_i.

15. A method implemented by fixed logic circuitry for summing a plurality of numbers, the method comprising:

formatting each number into a binary string by a method as claims in any of claims 1-7,

and bitwise adding the formatted binary strings.

16. An apparatus comprising fixed logical circuitry configured to perform a method as claimed in any of claims 1-7 or claim 15.