Description

Title of Invention: METHODS OF MULTIPLYING BINARY STRINGS AND MULTIPLIER CIRCUITS IMPLEMENTING THE SAME

[0001] CROSS-REFERENCE TO RELATED APPLICATIONS

[0002] This application claims priority and the benefit of PCT/CN2024/085517, filed 2 April 2024, and PCT/CN2024/119240, filed 16 September 2024, the entireties of which are incorporated herein by reference.

TECHNICAL FIELD OF THE INVENTION

[0003] The invention relates to logic circuits for manipulating bits in input binary strings to produce arithmetic output. The invention relates particularly to multiplication circuits.

BACKGROUND OF THE INVENTION

[0004] 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 logic circuits is the binary adder.

[0005] Figure 1 illustrates a generic Half-Adder circuit constructed of an AND gate and an Exclusive OR gate. The Half-Adder has two input leads for single-bits and two output leads, one for giving the sum as a single-bit and the other to issue a carry if the signal for binary 1 is supplied to both inputs, the result is 10_{bin} has to be contained in a higher bit positon.

[0006] A Full-Adder is produced when two Half-Adders are connected as shown in Figure 2. The output of the Full-Adder is the sum and a carry-out to be connected to the Carry-in of a Full-Adder in the next significant position. The Truth Table for the Full-Adder includes an additional input for taking in a carry bit passed from the immediate smaller bit position. A series of Full-Adders connected in this way is able to add up two binary strings, in which carry bits are passed on to more significant positions in what is called the “ripple effect” . Figure 3 is an illustration of four Full-Adders combined in series to provide a 4-bit adder.

[0007] Each Full-adder has to wait for the carry-over signal in order to have all the required inputs to produce an output. The combined waiting time in a series of Full-adders leads to an overall propagation delay. Propagation delay worsens with a greater number of Full-Adders in the series for adding longer binary strings.

[0008] To mitigate the extent of propagation delay, a modified circuit of serially arranged full adders called the Carry Look-ahead Binary Adder has been proposed, which is illustrated in Figure 4. The inputs of the Full-Adder are replicated to a logic component

called the Carry Look-ahead, which is able to detect combinations of inputs that are predictive of a carry-over and sends a carry-over signal to the downstream Full-adder even before the upstream Full-adder have finished calculating. For example, when an upstream Full-adder generates a carry-over or passes on a received carry-over, signals are sent to downstream Full-adders in a logic combination of 'carry generates' and 'carry propagates' .

[0009] The extent of time saved by using the Carry Look-ahead is almost exponential in adders for longer binary strings, but also the accompanying costs and circuit complexity, observing Moore's Law on the relationship between cost and complexity. The impetus to develop adders of larger capacity wanes against costliness and the burden of complexity.

[0010] Presently, there is no way of escaping the ripple effect, as binary additions are essentially a serial process of addition from the least significant bits to the most significant bits.

[0011] The effect of propagation delay is felt more poignantly in multiplication. Multiplication of two whole numbers means repeatedly adding one number, the multiplicand, to itself by as many times as the value of other number, the multiplier. Therefore, a multiplier circuit for computing the product of two binary strings comprises a plurality of adders to perform the additions. The multiplicand and the multiplier may also be called factors.

[0012] To multiply two binary strings, the first step is to obtain the partial products of the two binary strings. Each partial product is the product of the multiplicand and a specific bit in the multiplier, the product bit-shifted to the bit-position of the multiplier bit.

[0013] .

[0014] The partial products can be adding one after another to an accumulating sum. However, this means a renewed propagation delay from every addition of a partial product.

[0015] Present methods for reducing propagation delays in multiplier circuits are mostly efficiency strategies that may add up different parts of the partial products separately such as postponing carry-overs all to be summed up at once in a distinct, final stage in a specific circuit called the Carry-Save Multiplier, identifying patterns that are repetitive in adding up partial products, or doing several steps at the same time. None of these methods removes or reduces the cause of or need to wait for carry-overs.

[0016] It is therefore desirable to propose methods and related circuit that provide a possibility of relieving the need to wait for the signal of a carry-over bit.

SUMMARY OF THE INVENTION

[0017] In a first aspect the invention proposes a method of replacement of a nonzero in a bit-position among a plurality of bit-positions in two of four binary strings comprised in two generalised numbers to be added up; the replacement comprising the steps of replacing one or more nonzeros in bit-positions that includes the bit-position with a zero in the bit-position and one or more nonzeros in the other bit-positions among the plurality of bit-positions in two of four binary strings representing the same value.

[0018] The above aspect provides the possibility of making way for carry-over to be stored a bit-position among a plurality of bit-positions in binary strings, removing the need for carry propagation. This can be seen in the steps of the SEP, the rSEPx, rSEPy, Transformation A and Transformation B.

[0019] In a second aspect, the invention proposes a method of multiplying a multiplicand by a multiplier, comprising the steps of: 2a) expanding the multiplicand and the multiplier to obtain partial products in the form of generalised numbers; 2b) operating a plurality of addition operations to add up the partial products and produce a multiplication product; wherein each addition operation is preceded with the steps of the method of the first aspect.

[0020] Optionally, the multiplicand in step 2a) is a generalized number represented by two binary strings and the multiplier is a number represented by at least one binary string.

[0021] Optionally, the multiplier in step 2a) is a generalised number. Preferably, method comprises the following steps preceding step 2a) : replacing nonzeros in a block of four bits comprising two bits in the same adjacent positions in each binary string and representing a value with a block of four bits in the same positions having at least three zeros and a nonzero representing the same value.

[0022] In avoidance of doubt, this feature relates to execution of two steps described in the embodiments, the first of semi-optimising a multiplier generalised number, which replaces adjacent nonzeros in any one of substrings using SEP and a second step of replacing adjacent nonzeros in opposite sign substrings using rSEP.

[0023] Optionally, step 2a) comprises expanding the multiplicand by each consecutive block in the multiplier.

[0024] In a further aspect, where the multiplicand is a generalized number $(n_w)_1$ having substrings x_1 and y_1; and the multiplier being at least one binary string x_2 providing a number n_2; step 2a) comprises the steps of:

[0025] 7a) expanding the multiplicand $(n_w)_1$ by each bit $(x_j)_2$ in the multiplier n_2 for j=0, 1, 2…, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...; wherein

$(x'_i)_j = (x_i)_1(x_j)_2$ (1a)

$(y'_i)_j = (y_i)_1(x_j)_2$ (1b)

[0026] according to the Truth Table below for i=0, 1, 2, ... and j=0, 1, 2, ...:

Input		Output
$(z_i)_1$	$(x_j)_2$	$(z'_j)_2$
0	0	0
1	0	0
$\overline{1}$	0	0
0	1	0
1	1	1
$\overline{1}$	1	$\overline{1}$

[0027] in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$; $(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of $(n'_w)_1$, and so on;

[0028] 7b) for each iteration including pre-treating and adding up at least two partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, repeating a number of iteration until all the partial products are summed up into a final product N_w; wherein the pre-treating including a step of

[0029] 7c) replacing a nonzero with a zero in the bit-position i, the nonzero in the bit-position representing a value; and

[0030] providing at least one nonzero in a different bit-position in the same or the other one of the at least two partial products; such that

[0031] the bit-position i is able to store a carry-over during the adding up.

[0032] Optionally, the multiplier comprises a further binary string y_2 being a generalised number $(\hat{n}_w)_2$; wherein

[0033] step 2a) comprises:

[0034] expanding the multiplicand $(n_w)_1$ by each pair of bits $(x_j)_2$, $(y_j)_2$ of the multiplier $(\hat{n}_w)_2$ for j =0, 1, 2, …, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...;where

$(x'_i)_j=(x_i)_1(x_j)_2+(y_i)_1(y_j)_2$ (2a)

$(y'_i)_j=(y_i)_1(x_j)_2+(x_i)_1(y_j)_2$ (2b)

[0035] according to the Truth Table below for i=0, 1, 2, ...and j=0, 1, 2, ...:

Input		Output
$(z_i)_1$	$(z_j)_2$	$(z'_i)_j$
0	0	0
1	0	0
$\overline{1}$	0	0
0	1	0
1	1	1
$\overline{1}$	1	$\overline{1}$
0	$\overline{1}$	0
1	$\overline{1}$	$\overline{1}$
$\overline{1}$	$\overline{1}$	1

[0036] in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$;$(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of $(n'_w)_1$, and so on.

[0037] Optionally, step 7c) comprises the steps of:

[0038] 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

[0039] 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

[0040] 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;

[0041] 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

[0042] Summing Operation: performing the addition to produce the sum as two substrings as 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	$\overline{1}$	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	

[0043] and; adding up X and Y to produce a single binary string having the value of the sum of the two numbers.

[0044] Optionally, step 7c) comprises the steps of:

[0045] 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:

[0046] step 1) applying both the following equations to eliminate any immediately neighbouring nonzero bits in the binary substring y

$$x_i' = x_i\overline{y}_{i-1} + y_i\overline{y}_{i-1} \quad (3a)$$

[0047] and

$$y_i' = \overline{x}_i\overline{y}_i y_{i-1} \quad (3b)$$

[0048] 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:

$$x_i'' = x_i'\overline{y}_{i-1}' + x_{i+1}'y_i' \quad (4a)$$

$$y_i'' = \overline{x}_{i+1}'y_i' \quad (4b)$$

[0049] step 3) applying the following equations to eliminate any immediately neighbouring nonzero bits in the binary substring x

$$\widetilde{X}_i = \overline{x}_i''\overline{y}_i''x_{i-1}'' \quad (5a)$$

[0050] $\widetilde{Y}_i = x_i''\overline{x}_{i-1}''\overline{y}_{i-1}'' + y_i''\overline{x}_{i-1}''\overline{y}_{i-1}''$ or simply $\widetilde{Y}_i = x_i''\overline{x}_{i-1}'' + y_i''\overline{x}_{i-1}''$ (5b)

[0051] 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 $\tilde{y}$; and separately adding the positive and negative components by using the following equations;

$$x_i' = (\tilde{x}_i)_1(\overline{\tilde{x}}_i)_2 + (\overline{\tilde{x}}_i)_1(\tilde{x}_i)_2 + (\tilde{x}_{i-1})_1(\tilde{x}_{i-1})_2 \quad (6a)$$

$$y_i' = (\tilde{y}_i)_1(\overline{\tilde{y}}_i)_2 + (\overline{\tilde{y}}_i)_1(\tilde{y}_i)_2 + (\tilde{y}_{i-1})_1(\tilde{y}_{i-1})_2 \quad (6b)$$

.

[0052] Optionally, the method comprises the following steps preceding step 7a):

[0053] treating the multiplier $(n_w)_2$ as follows:

[0054] i)applying the following equations to eliminate any immediately neighbouring nonzero bits in the binary substring y

$$x_i' = x_i\bar{y}_{i-1} + y_i\bar{y}_{i-1} \quad (3a)$$

[0055] and

$$y_i' = \bar{x}_i\bar{y}_i y_{i-1} \quad (3b)$$

[0056] similar to Equations (3a) and (3b) above.

[0057] ii)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:

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

[0058] similar to Equations (4a) and (4b) above.

[0059] iii) applying the following equations to eliminate any immediately neighbouring nonzero bits in the binary substring x

$$\tilde{X}_i = \bar{x}_i''\bar{y}_i''x_{i-1}''$$

[0060] $\tilde{Y}_i = x_i''\bar{x}_{i-1}''\bar{y}_{i-1}'' + y_i''\bar{x}_{i-1}''\bar{y}_{i-1}''$ or simply $\tilde{Y}_i = x_i''\bar{x}_{i-1}'' + y_i''\bar{x}_{i-1}''$

[0061] similar to Equations (5a) and (5b) above, where $\tilde{X}_i$ and $\tilde{Y}_i$ denotes the value of the bits in position i in the formatted pair of binary substrings, $\tilde{X}$ and $\tilde{Y}$;

[0062] iv) reducing the semi-optimized multiplier $(\hat{n}_w)_2$ in blocks of consecutive two-bits according to the equations

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

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

$$(x_{i+1}')_2 = (\tilde{x}_{i+1})_2(\bar{\tilde{x}}_i)_2(\bar{\tilde{y}}_i)_2 \quad (7c)$$

$$(y_{i+1}')_2 = (\tilde{y}_{i+1})_2(\bar{\tilde{x}}_i)_2(\bar{\tilde{y}}_i)_2 \quad (7d)$$

[0063] where i = 0,2,4,…, such that there is at most one nonzero digit $(x_i')_2$, $(y_i')_2$, $(x_{i+1}')_2$, $(y_{i+1}')_2$ and the reduced multiplier $(n_w')_2$ is represented by the resultant substrings $(x_i')_2$ and $(y_i')_2$.

[0064] v) determining the partial products of $(\hat{n}_w)_1$ each consecutive two-digit pairs of the positive and negative strings of the reduced multiplier $(x_{i+1}')_2(x_i')_2$ and $(y_{i+1}')_2(y_i')_2$ for i = 0,2,4…to obtain partial products $(n_w)_{0-1}$, $(n_w)_{2-3}$, $(n_w)_{4-5}$,…;and

[0065] such that step (a) becomes:

[0066] expanding the multiplicand $(n_w)_1$ by each pair of bits $(x_j)_2$, $(y_j)_2$ of the multiplier $(\hat{n}_w)_2$ for j =0, 1, 2, …, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...;where

$(x'_i)_j=(x_i)_1(x_j)_2+(y_i)_1(y_j)_2$ (8a)

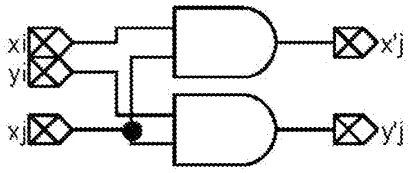
$(y'_i)_j=(y_i)_1(x_j)_2+(x_i)_1(y_j)_2$ (8b)

[0067] according to the Truth Table below for i=0, 1, 2, ...and j=0, 1, 2, ...:

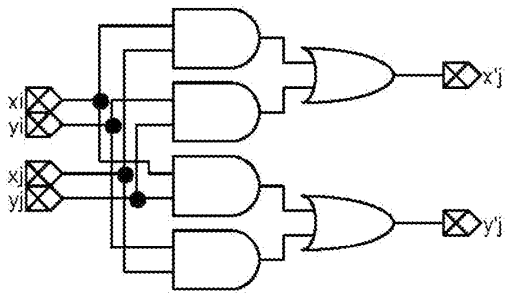
Input		Output
$(z_i)_1$	$(z_j)_2$	$(z'_i)_j$
0	0	0
1	0	0
$\bar{1}$	0	0
0	1	0
1	1	1
$\bar{1}$	1	$\bar{1}$
0	$\bar{1}$	0
1	$\bar{1}$	$\bar{1}$
$\bar{1}$	$\bar{1}$	1

[0068] in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$;$(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of $(n'_w)_1$, and so on.

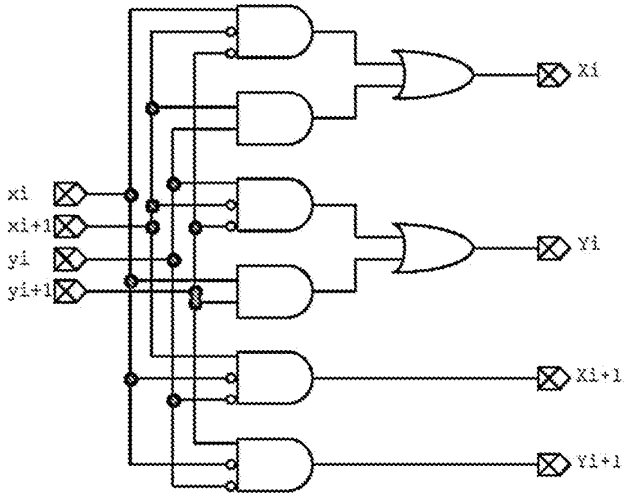
[0069] In a further aspect, the invention proposes a unit multiplier of an expansion circuit for obtaining partial products between a generalised number and a binary string according to step 2a) , comprising:



[0070] In a further aspect, the invention proposes a unit multiplier of an expansion circuit for obtaining partial products between two generalised numbers according to step 2a) of claim 4, comprising:



[0071] In a further aspect, the invention proposes a unit multiplier of an expansion circuit for obtaining partial products between two generalised numbers according to step 2a) of claim 6, comprising:



[0072] BRIEF DESCRIPTION OF THE FIGURES

[0073] 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.

[0074] Figure 1 illustrates a half-adder used in the prior art;

[0075] Figure 2 illustrates a full-adder used in the prior art;

[0076] Figure 3 illustrates a 4-bit ripple carry adder used in the prior art;

[0077] Figure 4 illustrates a Carry-Look-ahead binary adder used in the prior art;

[0078] Figure 5 is a flowchart illustrating the steps in an embodiment of the invention;

[0079] 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;

[0080] Figure 7 is a circuit diagram of the content in the subcircuit shown in Figure 6;

[0081] Figure 8 illustrates how the circuit diagrams of Figure 6 and Figure 7 generalize a binary string;

[0082] Figure 9 is the logic circuit for applying the Specific Equivalence Principle Conversion to x substrings;

[0083] Figure 10 is the logic circuit for applying the Specific Equivalence Principle Conversion to y substrings;

[0084] Figure 11 is the Truth Table for Specific Equivalence Principle Conversion in Figure 9 and Figure 10;

[0085] Figure 12 shows a drawing corresponding to Stage 3 and Stage 4 in the flowchart of Figure 5;

[0086] Figure 13 is a 2-unit circuit block for performing Transformation A in Stage 3 in the flowchart of Figure 5;

[0087] Figure 14 a single unit of the logic circuit in Figure 13;

[0088] Figure 15 the Truth Table underlying the circuit block shown in Figure 13;

[0089] Figure 16 is a logic circuit for performing transformation in Simplification Transformation shown in Stage 3 in the flowchart of Figure 5;

[0090] 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;

[0091] Figure 19 is a single unit for the circuit in Figure 18;

[0092] Figure 20 shows the Truth Table for Figure 18;

[0093] Figure 21 is a bit addition circuit for Summing Operation in Stage 4 in the Flowchart of Figure 5;

[0094] Figure 22 is the Truth Table for Figure 21;

[0095] Figure 23 is an equivalent Truth Table for Figure 22;

[0096] Figure 24 is the circuit for adding two 8-bit generalized binary numbers;

[0097] 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;

[0098] 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;

[0099] Figure 27 is the logic circuit in the subcircuit of Figure 26;

[0100] Figure 28 is the Truth Table for the logic circuit of Figure 27;

[0101] Figure 29 is the Truth Table of Figure 27;

[0102] Figure 30 de-converts a 4-bit generalized binary number into a 5-bit classical binary format;

[0103] Figure 31 is a “Carry-Look-ahead” -like circuit having modified generate, propagate and output of carry and sum;

[0104] Figure 32 is the Truth Table for bit sedimentation, Stage 3 of the Alternative Method;

[0105] Figure 33 is the logic circuit for Figure 32;

[0106] Figure 34 is the logic circuit for Figure 32;

[0107] Figure 35 is the Truth Table for Transformation AB of the Alternative Method;

[0108] Figure 36 is the logic circuit for Figure 35;

[0109] Figure 37 is the logic circuit for Figure 35;

[0110] Figure 38 is the Truth Table for bit addition of the Alternative Method;

[0111] Figure 39 is the logic circuit for Figure 38;

[0112] Figure 40 is the logic circuit for Figure 38;

[0113] Figure 41 is the circuit for adding two 8-bit generalized binary number using the Alternative Method;

[0114] 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;

[0115] Figure 43 is the Truth Table for an embodiment using only a Transformation A;

[0116] Figure 44 is a possible logic circuit for Figure 43;

[0117] Figure 45 is also a logic circuit for Figure 43;

[0118] Figure 46 is the logic circuit for an adder that can be used instead of the one in Figure 21;

[0119] Figure 47 is a higher level subcircuit for adding up two generalised numbers treated by a process called semi-optimisation in a second embodiment of a multiplier circuit;

[0120] Figure 48 shows three subcircuits and the arrangement thereof which are represented by the subcircuit in Figure 47;

[0121] Figure 49 is a subcircuit diagram that is within one of the three subcircuits in Figure 48;

[0122] Figure 50 shows the logic circuit that is in each of the nine subcircuits the circuit diagram shown in Figure 49;

[0123] Figure 51 is the Truth Table of the logic circuit of Figure 50;

[0124] Figure 52 is a subcircuit diagram that is within another one of the three subcircuits in Figure 48;

[0125] Figure 53 shows the logic circuit that is in each of the subcircuits shown in the circuit diagram of Figure 52;

[0126] Figure 54 is the Truth Table for the logic circuit of Figure 52;

[0127] Figure 55 is a subcircuit diagram that is within another one of the three subcircuits in Figure 48;

[0128] Figure 56 shows the logic circuit that is in each of the subcircuits shown in the circuit diagram of Figure 55;

[0129] Figure 57 is the Truth Table for the logic circuit of Figure 55;

[0130] Figure 58 is a higher level subcircuit that provides an adder used in one of the steps in the flowchart of Figure 5;

[0131] Figure 59 is a subcircuit diagram that that is in the subcircuit shown in Figure 58;

[0132] Figure 60 shows an adder subcircuit that is in each subcircuit units shown in the subcircuit diagram of Figure 59;

[0133] Figure 61 shows the circuit logic of a sub-adder that is used to construct the adder subcircuit in Figure 60

[0134] Figure 62 is the Truth Table for the logic circuit of Figure 61;

[0135] Figure 63 is the Truth Table multiplication of generalised number by another generalised number;

[0136] Figure 64 shows a circuit block diagram implementing the Truth table shown in Figure 63;

[0137] Figure 65 shows the subcircuit inside the circuit block of Figure 64;

[0138] Figure 66 is a multiplier circuit for multiplication of generalised number by another generalised implementing the circuits shown in Figure 65 and Figure 64;

[0139] Figure 67 shows the internal sub-circuits in the multiplier circuit of Figure 66

[0140] Figure 68 shows the circuit for obtaining partial products in the multiplier circuit of Figure 66

[0141] Figure 69 shows the multiplication subcircuit for obtaining a partial product in the circuit of Figure 68;

[0142] Figure 70 is the Truth Table for simplification;

[0143] Figure 71 is a block circuit diagram implementing the Truth Table in Figure 70;

[0144] Figure 72 is the logic circuit inside the block circuit shown in Figure 71;

[0145] Figure 73 is the Truth Table for expansion of a generalised number with a regular single binary string;

[0146] Figure 74 is a block circuit diagram implementing Figure 73;

[0147] Figure 75 shows the logic circuit inside the block circuit diagram shown in Figure 74;

[0148] Figure 76 shows a multiplier circuit block diagram of a multiplicand of an 8-bit generalised number and a multiplier of as 4 bit regular binary string;

[0149] Figure 77 shows the sub-circuits in the multiplier circuit black of Figure 76

[0150] Figure 78 shows one of the subcircuits in Figure 77 that is for obtaining partial products;

[0151] Figure 79 is the subcircuit inside the circuit black diagram shown in Figure 78;

[0152] Figure 80 is the Truth Table of step (a) of a variation of the process of Figure 83.

[0153] Figure 81 is the Truth Table of a step (b) which follows step (a) of the Truth Table shown as Figure 80.

[0154] Figure 82 is the Truth Table for the subcircuit diagram of Figure 83;

[0155] Figure 83 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;

[0156] Figure 84 is the logic circuit in the subcircuit diagrams of Figure 83;

[0157] Figure 85 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;

[0158] Figure 86 is the logic circuit in the subcircuit diagrams of Figure 85;

[0159] Figure 87 is the Truth Table for the subcircuit diagram of Figure 85;

[0160] Figure 88 is a subcircuit for processing the output of the subcircuit of Figure 85;

[0161] Figure 89 is the logic circuit in the subcircuit diagrams of Figure 88;

[0162] Figure 90 is the Truth Table for the subcircuit diagram of Figure 88' and

[0163] Figure 91 shows Tables (a) and (b) for a multiplier strategy called the LimTree.

[0164] LISTING OF EQUATIONS

[0165] Equations are as an alternative description of some features of embodiments in the following section. For convenience of one looking up the equations, a list of the equations by their purpose is provided. The list is not exhaustive, as embodiments that do not fit the equations without departing from the inventive concepts are possible. Furthermore, particularity of the equations should not be mis-taken as limiting to or superseding the generality of the invention.

[0166] Equation 1 relates to ggeneralization of a n-bit binary string n_w that comprises pair of substrings, x_1, y_1, and x_2, y_2.

[0167] Equation 2 relates to the expression of n_w where the weight of each bit in the binary strings is unity or a positive-integer power of 2 and the domain covers a range from $-(2^n-1)$ to 2^n-1.

[0168] Equations 3 (a, b) relate to SEP for a substring of m ($\geq$1) 1's .

[0169] Equations 4 (a, b) relate to SEP for a substring of m($\geq$ 1) $\bar{1}$'s.

[0170] Equations 5 (a, b) relate to Transformation A for the positive components of generalized augend $(n_w)_1$ and generalised addend $(n_w)_2$.

[0171] Equations 6 (a, b) relate to Transformation A for the negative components of generalized augend $(n_w)_1$ and generalised addend $(n_w)_2$.

[0172] Equations 7 (a, b) relate to Simplification for $(n'_w)_1$ following Transformation A.

[0173] Equations 8 (a, b) relate to Simplification for $(n'_w)_2$ following Transformation A.

[0174] Equation 9 relates to the shorthand notation z representing a pair of x and y substrings of an n-bit generalized binary number n_w.

[0175] Equation 10 defines the set of values for an n-bit generalized binary number z.

[0176] Equation 11 relates to the definition of z and its range from - (2^n-1) to 2^n-1.

[0177] Equations 12 (a, b) relate to Transformation B for positive component x substrings following Transformation A then Simplification.

[0178] Equations 13 (a, b) relate Transformation B for negative component y substrings following Transformation A then Simplification.

[0179] Equations 14 (a, b) relate to parallel summation of positive and negative substrings for two generalized numbers having treated with Transformation A, Simplification and Transformation B in sequence.

[0180] Equations 15 (a, b) relate to new definitions for generate (g_i) and propagate (p_i) for de-converting a generalized number with positive and negative substrings.

[0181] Equations 16 (a, b) relate to "final summation" logic for de-conversion.

[0182] Equations 17 (a, b) relate to Bit Sedimentation for the positive components of generalised augend $(n_w)_1$ and generalised addend $(n_w)_2$.

[0183] Equations 18 (a, b) relate to Bit Sedimentation for the negative components of generalised augend $(n_w)_1$ and generalised addend $(n_w)_2$.

[0184] Equations 19 (a-c) relate to Transformation AB for positive substrings of $(n'_w)_1$ and $(n'_w)_2$ following Transformation A.

[0185] Equations 20 (a-c) relate. relate to Transformation AB for negative substrings of $(n'_w)_1$ and $(n'_w)_2$ following Transformation A.

[0186] Equations 21 (a, b) relate to parallel summation of positive and negative substrings for two generalized numbers having treated with Bit Sedimentation and Transformation AB in sequence.

[0187] Equations 22 (a, b) relate parallel summation of positive and negative substrings following Transformation A then Simplification, without Transformation B.

[0188] Equations 23 (a, b) relate to expansion of multiplicand $(n_w)_1$ substrings $(x_i)_1$, $(y_i)_1$ for i=0, 1, 2, ...; by each bit pair of multiplier $(\hat{n}_w)_2$ substrings of $(x_j)_2$, $(y_j)_2$ for j =0, 1, 2, …, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...

[0189] Equations 24 (a, b) relate to ySEP to eliminate any immediately neighbouring nonzero bits in binary substring y;

[0190] Equations 25 (a, b) relate to reverse rSEPx to eliminate any pairs of neighbouring nonzero bits in 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;

[0191] Equations 26 (a, b) relate to xSEP to eliminate any immediately neighbouring nonzero bits in binary substring x;

[0192] Equations 27 (a, b) relate to separate and parallel addition of positive and negative components for semi-optimized generalized binary numbers;

[0193] Equations 28 (a-d) relate concurrent application of rSEPx and rSEPy for reducing semi-optimized multiplier $(\hat{n}_w)_2$ in blocks of consecutive two-bits such that there is at most one nonzero digit $(x'_i)_2$, $(y'_i)_2$, $(x'_{i+1})_2$, $(y'_{i+1})_2$, and the reduced multiplier $(n'_w)_2$ is represented by the resultant substrings $(x'_i)_2$, and $(y'_i)_2$;

[0194] Equations 29 (a, b) relate to expansion of multiplicand $(n_w)_1$ substrings $(x_i)_1$, $(y_i)_1$ for i=0, 1, 2, ...; by each multiplier bits $(x_j)_2$ for j=0, 1, 2…, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...

[0195] Equation 30 relates to one of the two steps for semi-optimizing a generalized number n_w, by which n_w, having positive and negative substrings, is subtracted from an instant of the same substrings moved by one significant bit, ie. $2n_w$-n_w.

[0196] Equations 31 (a, b) relate to the parallel logic operation for $2n_w$-n_w, where $zn-n_{+1}=z_{-1}=z_{-2}=z_{-3}=0$ are assumed.

[0197] Equations 32 (a, b) relates to a parallel logic operation that complements Equations 31 (a, b) to obtain semi-optimized generalized numbers.

[0198] Equations 33 (a, b) relate to a combination of Equations 31 (a, b) and Equations 32(a, b) such that a generalized binary number can be semi-optimized in parallel in a single step.

[0199] Equations 34 (a, b) relate to transforming any substrings of $10\bar{1}$ and $\bar{1}01$ into $1\bar{1}1$ and $\bar{1}1\bar{1}$, respectively.

[0200] Equations 35 (a, b) is a parallel logic operation that applies $2n_w$-n_w.

[0201] Equations 36 (a, b) relate to determining the number of bit strings ki (i=0, 1, 2, …) at the i-th level for LimTree and Wallace Tree, respectively.

[0202] Equations 37 (a, b) relate the number of bit strings for two 64-bit binary strings of the regular single-strand format (not generalized numbers each comprising with a pair of x and y substrings) by applying LimTree and Wallace Tree, respectively.

[0203] DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

[0204] Embodiments described in the following paragraphs are 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.

[0205] Equivalence Principle (General)

[0206] Any positive whole number with zero excluded (i.e. 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.

[0207] Accordingly, given a pair of binary strings to be added up to find their sum, it is possible to 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.

[0208] Generalised Binary Number Format and the process of Generalization

[0209] 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.

[0210] 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.

[0211] 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.

[0212] Accordingly, it would be 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's ubstring, 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.

[0213] 'String' and 'number' are used interchangeably in this description when describing the binary.

[0214] 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:

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

[0215] where

[0216] $a_i^+ = \langle 0,1 \rangle$ and $a_i^- = \langle 0, \bar{1} \rangle$, in which i=0, 1, ..., n-1; and

[0217] <, > indicates a set of possible values for the variable.

[0218] $\bar{1}$ is a shorthand for a nonzero bit in the y substring, consisting of a bar over a digit 1.

[0219] The weight of each bit in the binary strings is unity or a positive-integer power of 2.

[0220] Hence, n_w can be expressed as:

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

[0221] The domain of n_w covers a range from - (2^n-1) to 2^n-1.

[0222] 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 (the most significant) in the otherwise all-zero y substring.

[0223] 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.

[0224] 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.

[0225] 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.

[0226] A "Specific" Equivalent Principle Conversion

[0227] 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 (≥ 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.*

[0228] 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.

[0229] Taking the positive string as x_i= <0, 1> 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 Equations (3a) and (3b) , while that for a substring of m (≥ 1) $\bar{1}$'s can be expressed by Equations (4a) and (4b) .

$$X_i = \bar{x}_i x_{i-1} \quad ; \quad Y_i = x_i \bar{x}_{i-1} \qquad (3a,3b)$$

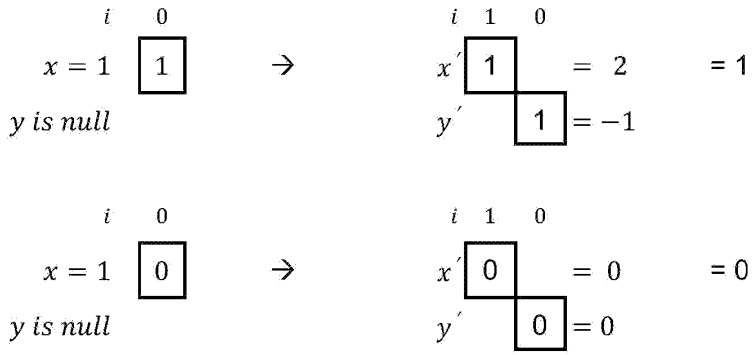
$$X_i = y_i \bar{y}_{i-1} \quad ; \quad Y_i = \bar{y}_i y_{i-1} \qquad (4a,4b)$$

[0230] The left in Figure 9 is the corresponding circuit block diagram 901 while the right drawing in Figure 9 is the underlying logic circuit unit 903 for SEP conversion of an x substring. Similarly, the left drawing in Figure 10 is the corresponding circuit

block diagram while the right 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.

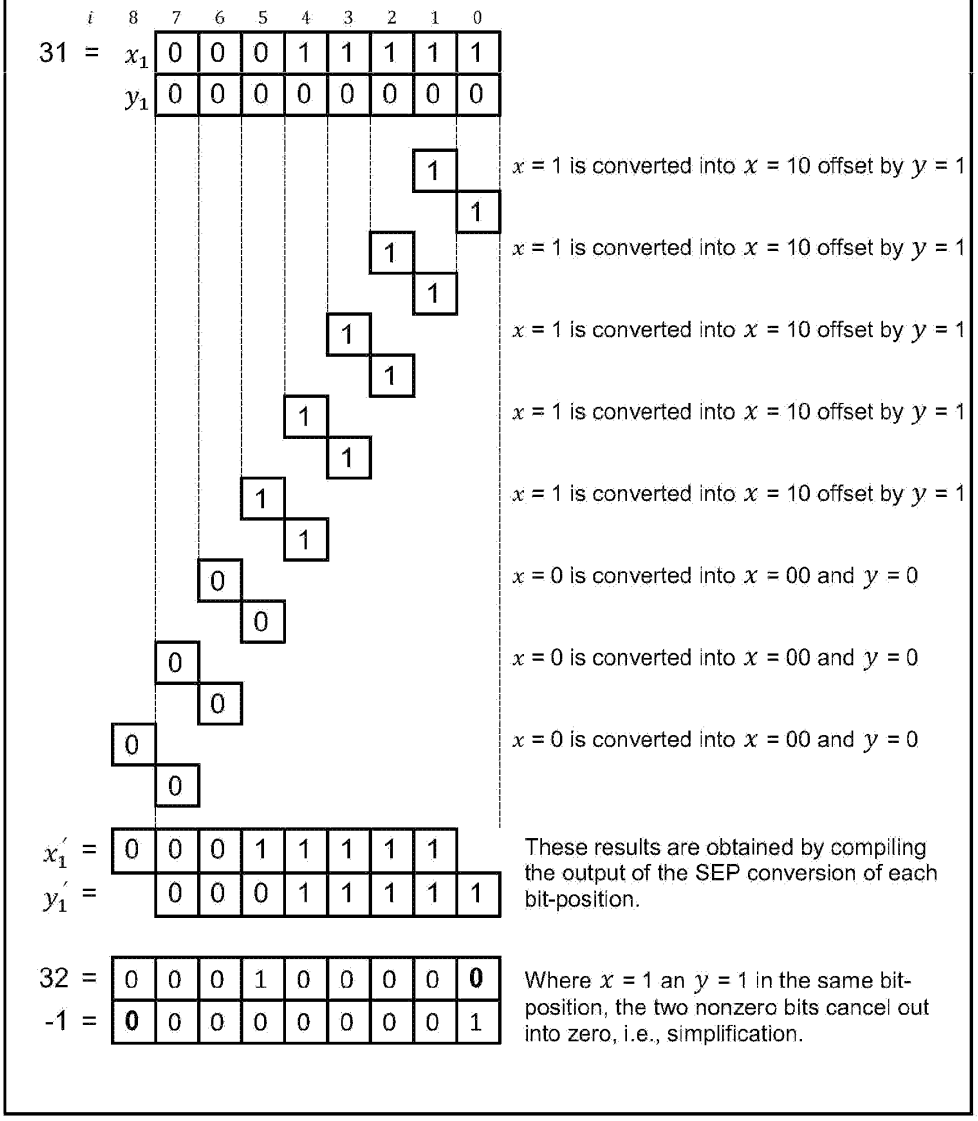
[0231] Box 2 below shows the SEP conversion, which is based on the possibility of re-expressing a positive bit having a nonzero value of 1 as a greater number offset with a smaller and negative number, such as x –y = 2-1= 1, while observing the Equivalence Principle. For completeness, Box 2 shows that 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



[0232] 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

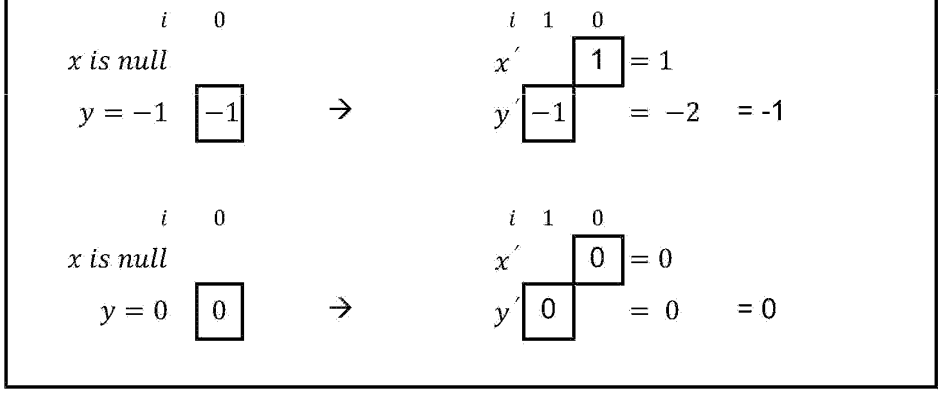


[0233] As described already in a preceding paragraph, it can be seen that the overall effect of the SEP conversion lies in the removal of all immediately adjacent nonzero bits in the x substring, but without changing the sum 31, i.e. 32-1. The SEP conversion must be applied consistently to every bit-position in the x substring.

[0234] 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.

[0235] 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



[0236] The example in Box 4 has a final step of simplifying the SEP conversion output, which is not desirable for it adds an additional step to the overall embodiment.

[0237] 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.

[0238] 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	$0 = 0$
			y' 0 0	
ii.	x 0 1	$= 1$	x' 1 0	$2 = 1$
			y' 0 1	$= -1$
iii.	x 1 0	$= 2$	x' 1 0 0	$4 = 2$
			y' 1 0	$= -2$
iv.	x 1 1	$= 3$	x' 1 1 = x' 1 0 0	$4 = 3$
			y' 1 1 y' 0 1	$= -1$

Cases	i 1 0		i 2 1 0	Equivalent
v.	y 0 0	$= 0$	x' 0 0	$0 = 0$
			y' 0 0	
vi.	y 0 1	$= -1$	x' 0 1	$1 = -1$
			y' 1 0	-2
vii.	y 1 0	$= -2$	x' 1 0	$2 = -2$
			y' 1 0 0	-4
viii.	y 1 1	$= -3$	x' 1 1 = x' 0 1	$1 = -3$
			y' 1 1 y' 1 0 0	-4

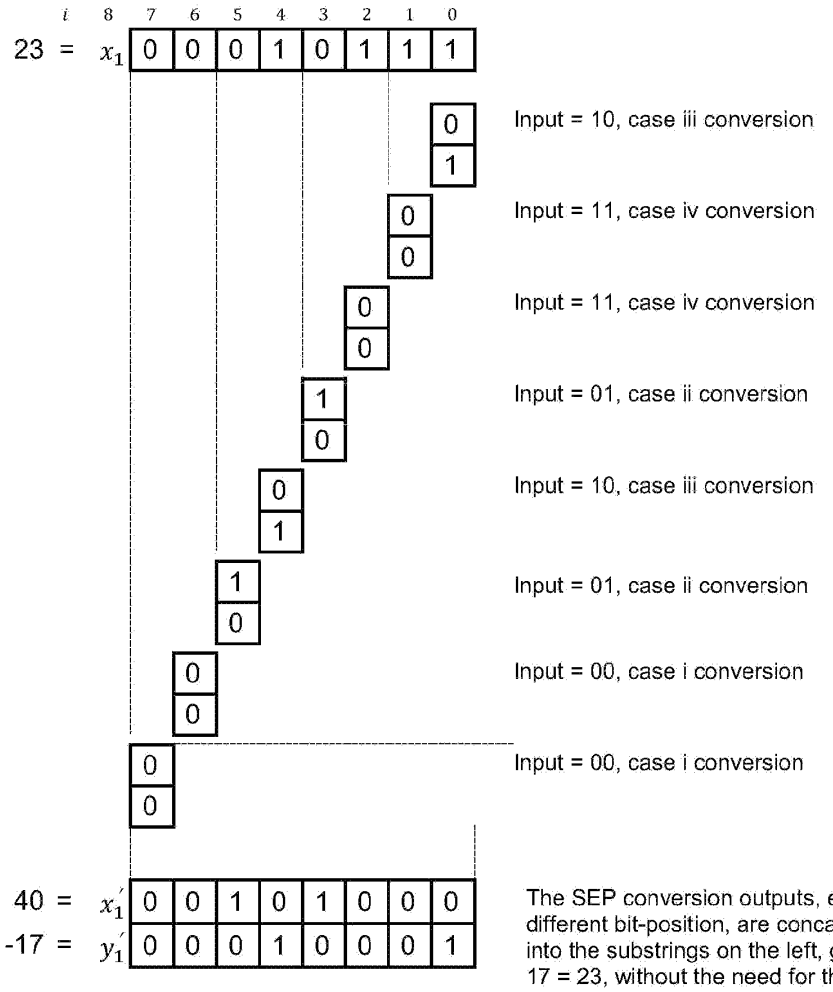
[0239] Cases i to iv in Box 5 show SEP conversions of all the four possible combinations of two bits in the positive component x substring, i.e. 00, 01, 10, 11. The two bits are adjacent to each other in the same substring. Conversely, cases v to viii in Box 5 shows the SEP conversion of all the four possible combinations of two bits in the negative component y substring, i.e. 00, 01, 10, 11.

[0240] 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.

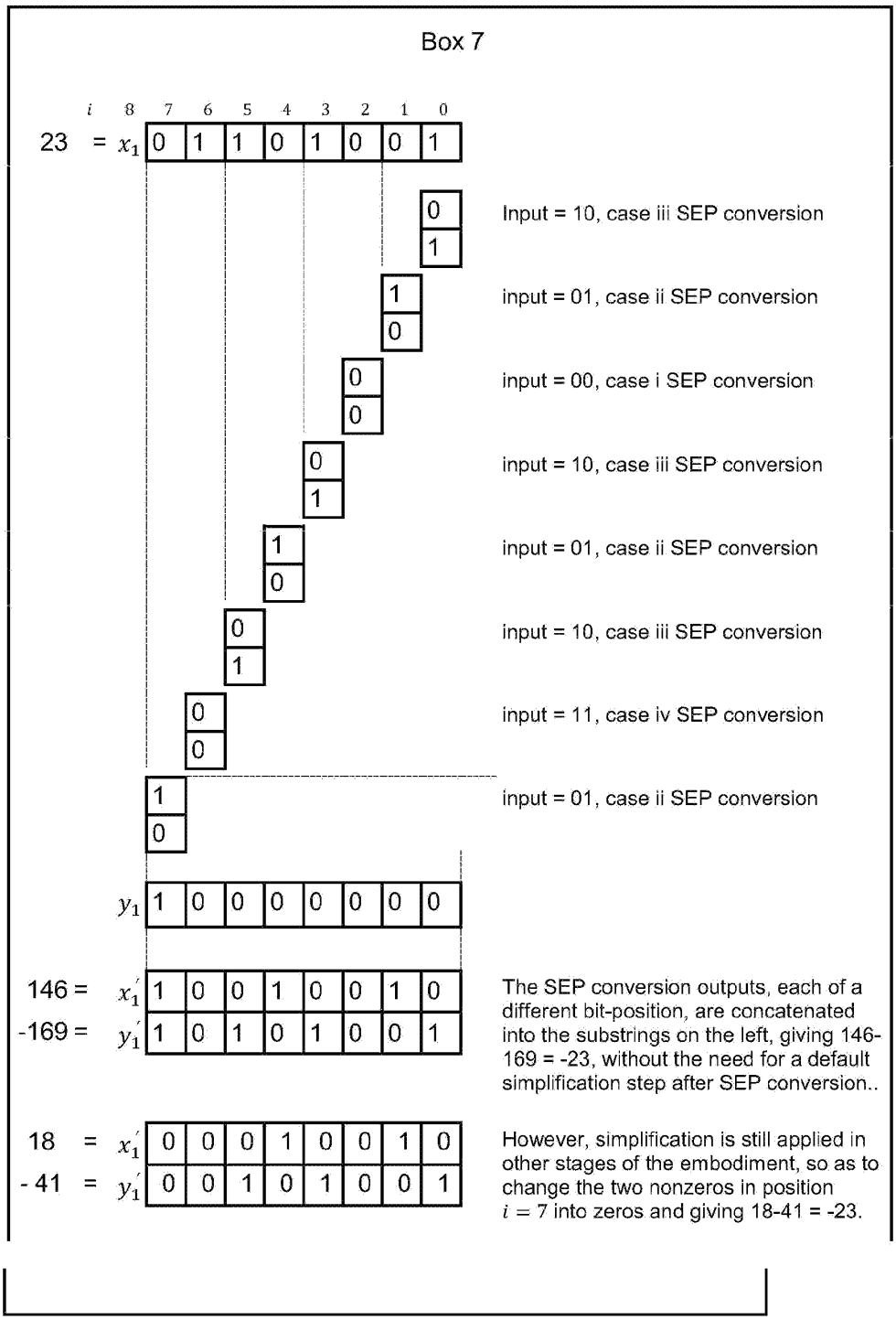
[0241] Upon SEP conversion of each bit based on the transformation in Box 6, the x substring becomes 0010 1000 (40 in decimal) , and the y substring becomes 0001 0001 (17 in decimal) . The difference between x-y in the decimal system is 40 - 17 = 23 (General Equivalence Principle observed) . These substrings may be written as $001\bar{1}\ 100\bar{1}$ in the recommended shorthand.

[0242] 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 6



The SEP conversion outputs, each of a different bit-position, are concatenated into the substrings on the left, giving 40-17 = 23, without the need for the simplification step seen in Box 3.



[0243] 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.

[0244] 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.

[0245] 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.

[0246] A First Adder Embodiment

[0247] 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.

[0248] 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.

[0249] · In Stage 1, an augend and an addend are provided in the form of binary strings.

[0250] · 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.

[0251] · 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 $(n_w)_2$.

[0252] 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.

[0253] 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.

[0254] 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.

[0255] Transformation A and Transformation B are different transformations, however, as will be explained in greater detail in the following paragraphs.

[0256] · 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.

[0257] 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.

[0258] · 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 by being converted into the generalized number 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.

[0259] · 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.

[0260] 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.

[0261] 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	7	6	5	4	3	2	1	0
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

[0262] 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.

[0263] Transformation A, with subsequent Simplification, and Transformation B preemptively 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.

[0264] Transformation A

[0265] Transformation A relates to the generalised augend $(n_w)_1$ and generalised addend $(n_w)_2$ according to the following equations.

$$(y_i')_1 = (x_i)_1(\overline{x_i})_2 + (\overline{x_i})_1(x_i)_2 \qquad (5a)$$

$$(x'_{i+1})_1 = (x_i)_1 + (x_i)_2 \qquad (5b)$$

[0266] for the positive components of $(n_w)_1$ and $(n_w)_2$; and

$$(x_i')_2 = (y_i)_1(\overline{y_i})_2 + (\overline{y_i})_1(y_i)_2 \qquad (6a)$$

$$(y'_{i+1})_2 = (y_i)_1 + (y_i)_2 \qquad (6b)$$

[0267] for the negative components of $(n_w)_1$ and $(n_w)_2$.

[0268] 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.

[0269] 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 positivecomponent 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			$i+1$ i	
i.	x_1 0 + x_2 0	= 0		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	$2 - 0 = 2$

[0270] 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.

[0271] 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.

[0272] Only case iv outputs a greater value due to the sum of two nonzero input bits, whichrequires 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.

[0273] 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.

[0274] 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.

[0275] 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.

[0276] 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 $(y_1+y_2) = 1+1=10$.

[0277] 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.

[0278] 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.

[0279] 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
	i			$i+1$ i	
v.	y_1 0 + y_2 0	= 0		x'_2 ▒ 0 / y'_2 0 ▒	$0 - 0 = 0$
vi.	y_1 0 + y_2 $\bar{1}$	= -1	SEP →	x'_2 ▒ 1 / y'_2 $\bar{1}$ ▒	$1 - 2 = -1$
vii.	y_1 $\bar{1}$ + y_2 0	= -1	SEP →	x'_2 ▒ 1 / y'_2 $\bar{1}$ ▒	$1 - 2 = -1$
viii.	y_1 $\bar{1}$ + y_2 $\bar{1}$	= -1 0		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.

[0280] 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, viiioutput a new pair of x'_2 and y'_2 substrings.

[0281] Only case viii outputs a greater value due to the sum of two (negative) nonzeroinput 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.

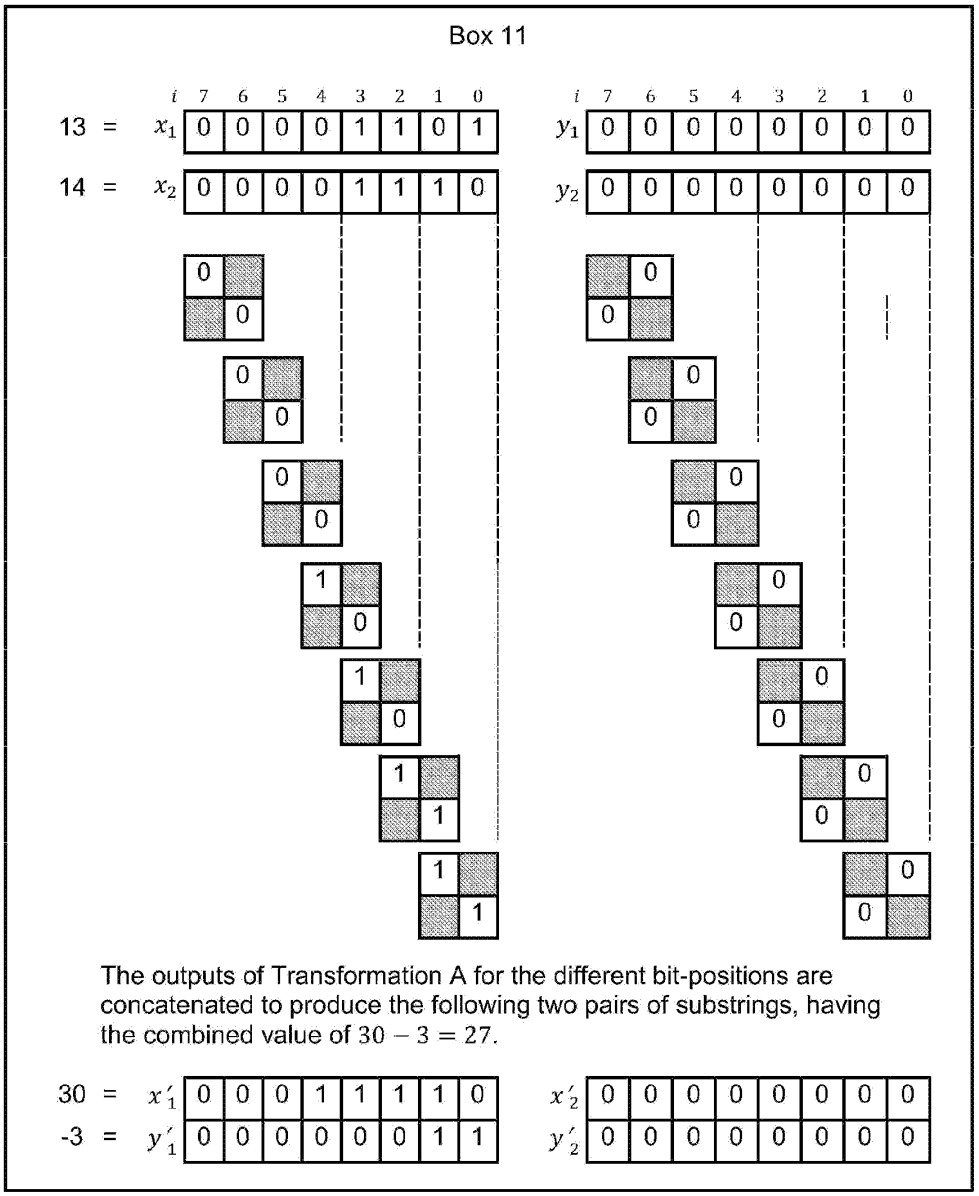
[0282] Case v arises when both input bits have zero values, Transformation A moves the output bit having zero value by one position in the output substring of the same sign component. This leaves the original bit-position i with a zero value so that any carry-over from the lower bit-position value may be received and accommodated, stemming the need to propagate the carry-over further.

[0283] 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. $\bar{1} = \bar{1}1$.

[0284] 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.

[0285] 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.

[0286] Continuing with the example of adding 14 to 13, Box 11 shows Transformation A applied to the output of Box 8.



[0287] 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.

[0288] Simplification Transformation

[0289] 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.

[0290] 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.

[0291] The following equations describe the process of the Simplification Transformation.

$$\left(x_i''\right)_1 = \left(x_i'\right)_1 \left(\overline{y_i'}\right)_1 \tag{7a}$$

$$\left(y_i''\right)_1 = \left(\overline{x_i'}\right)_1 \left(y_i'\right)_1 \tag{7b}$$

[0292] and

$$\left(x_i''\right)_2 = \left(x_i'\right)_2 \left(\overline{y_i'}\right)_2 \tag{8a}$$

$$\left(y_i''\right)_2 = \left(\overline{x_i'}\right)_2 \left(y_i'\right)_2 \tag{8b}$$

[0293] 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) .

[0294] Shorthand Notation Z

[0295] A shorthand notation Z is used to represent a pair of x and y substring of an n-bit generalized binary number n_w as follow:

$$\begin{aligned} z_w &= x_w - y_w \\ z_{n-1}z_{n-2}z_{n-3}\ldots z_2z_1z_0 &= x_{n-1}x_{n-2}x_{n-3}\ldots x_2x_1x_0 - y_{n-1}y_{n-2}y_{n-3}\ldots y_2y_1y_0 \\ &= \begin{pmatrix} x_{n-1}x_{n-2}x_{n-3}\ldots x_2x_1x_0 \\ y_{n-1}y_{n-2}y_{n-3}\ldots y_2y_1y_0 \end{pmatrix} \end{aligned} \tag{9}$$

[0296] where $z_i = \langle \overline{1}, 0, 1 \rangle$, x_i= <0, 1> and y_i= <0, 1> , in which i=0, 1, ..., n-1. Here,

$$\begin{aligned} &z_i = \overline{1} \Leftrightarrow x_i = 0\,;\, y_i = 1 \\ &z_i = 0 \Leftrightarrow x_i = 0\,;\, y_i = 0 \quad \text{or} \quad x_i = 1\,;\, y_i = 1 \\ &z_i = 1 \Leftrightarrow x_i = 1\,;\, y_i = 0 \end{aligned} \tag{10}$$

[0297] The weight of each bit is unity or a positive-integer power of 2. Hence, z_w can also be expressed as

$$z_w = \sum_{i=0}^{n-1} 2^i z_i = \sum_{i=0}^{n-1} 2^i x_i - \sum_{i=0}^{n-1} 2^i y_i = \sum_{i=0}^{n-1} 2^i \left(x_i - y_i\right) \tag{11}$$

[0298] The domain of z_w covers a range from - (2^n-1) to 2^n-1. This generalized binary number shorthand is also applicable to negative values in 2's complement form.

[0299] 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 gives 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

[0300] Box 12 shows that a Simplification Transformation is a matter between any paired x and y substrings.

[0301] Transformation B

[0302] 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.

[0303] 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:

$$\left(x_i'''\right)_1 = \left(x_i''\right)_1 \left(\overline{x_i''}\right)_2 + \left(\overline{x_i''}\right)_1 \left(x_i''\right)_2 \qquad (12a)$$

$(x'''_{i+1})_2 = (x_i'')_1 (x_i'')_2$ (12b)

[0304] as between the positive component x substrings; and

$$\left(y_i'''\right)_1 = \left(y_i''\right)_1 \left(\overline{y_i''}\right)_2 + \left(\overline{y_i''}\right)_1 \left(y_i''\right)_2 \qquad (13a)$$

$(y'''_{i+1})_2 = (y_i'')_1 (y_i'')_2$ (13b)

[0305] as between the negative component y substrings.

[0306] Figure 19 (Subcircuit A4-2X12) is an underlying logic circuit unit corresponding to Equations (12a) , (12b) and Equations (13a) , (13b) . 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.

[0307] 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	$1+1=10$

[0308] 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 having only zero bits and being available for accepting any carry-over value from the lower bit-position.

[0309] 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 the need to propagate the carry-over further.

[0310] 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 in, 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

the 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.

[0311] 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.

[0312] 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 (position i)	Sum (bin)	(b) Output (positions $i+1$, i)	(c) Transformation
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	$-1-1=-10$

[0313] In Box 14, 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.

[0314] 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.

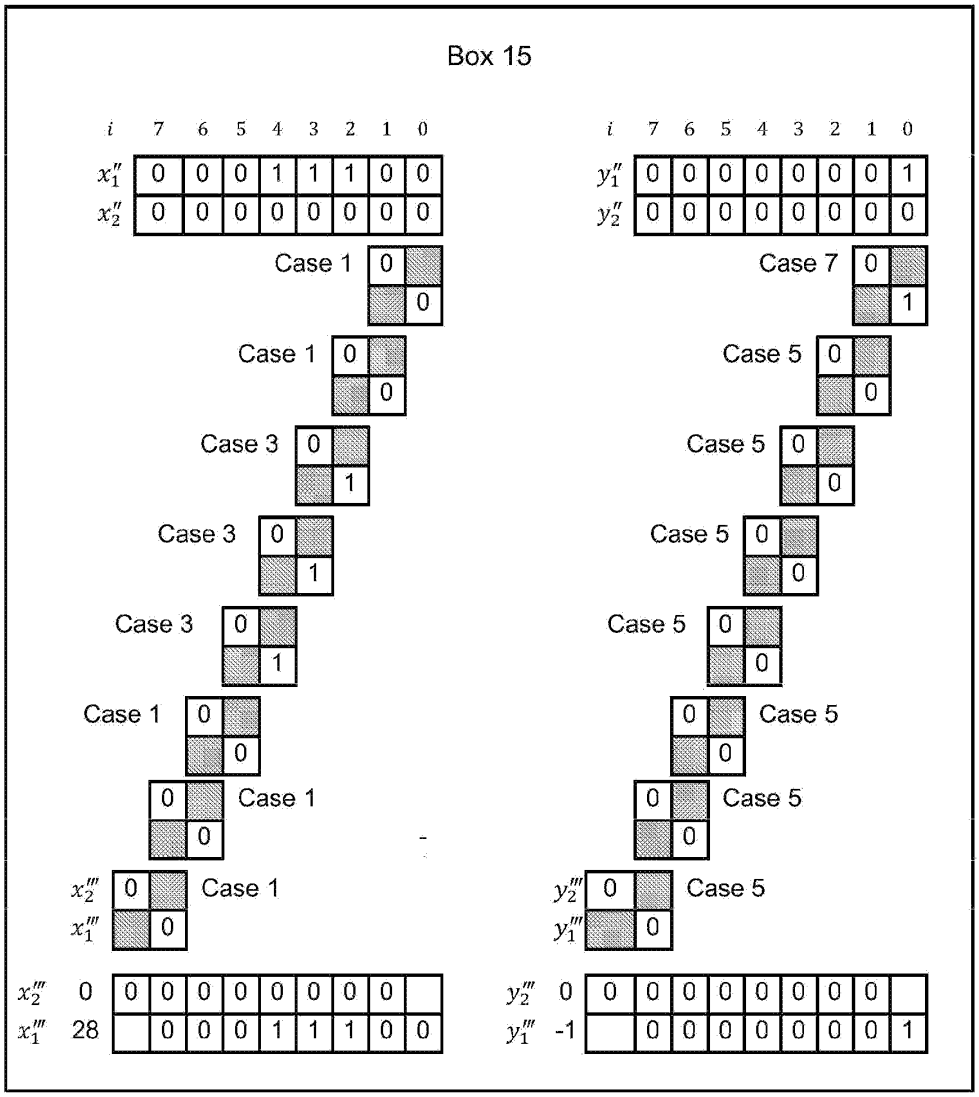
[0315] Case 5 arises when there are only zero bits in position i in both the y substrings, and the memory pointer for a zero value bit is moved by one greater bit-position in the output substring y'''_2.

[0316] 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.

[0317] 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.

[0318] With the application of Transformation A, with subsequent Simplification, and Transformation B, all carry-overs will have been manifested and pre-emptively accommodated.

[0319] 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.



[0320] Parallel Summing Operations

[0321] 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

$X_i = (x'''_i)_1 + (x'''_i)_2$ (14a)

$Y_i = (y'''_i)_1 + (y'''_i)_2$ (14b)

[0322] 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.

[0323] 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 "$\bar{1}$".

[0324] 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.

[0325] 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.

[0326] 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.

[0327] 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.

[0328] 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.

[0329] 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.

[0330] Overview of First Adder Embodiment

[0331] 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.

[0332] Box 17 shows another example which illustrates the embodiment by showing the steps leading to the summation 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) using the adder in Figure 24. The sum is N_w $1\bar{1}1\bar{1}01000$ (ageneralized 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

[0333] More Adding Cycles

[0334] 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.

[0335] Deconversion

[0336] 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+} ($\equiv n_w^{2+}$).

[0337] 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 de-convertible 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.

[0338] 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 Look-ahead can be modified by replacing the generate (g_i) and the propagate (p_i) signals with the following.

$$p_i = \overline{x}_i \overline{y}_i \qquad (15a)$$

$g_i=y_i$ (15b)

[0339] The "final summation" can be expressed by the following equations:

$$X_i = t_i = \overline{c}_i x_i + c_i p_i + \overline{c}_i y_i \text{ for } i = 0,1,2,\cdots,(n-1) \qquad (16a)$$

$Y_n=c_n$ (16b)

[0340] when a "carry" for each bit has been looked-ahead, where c_i (i=0, 1, 2, …, n) are the resultant "carries" of the modified "Carry-Look-ahead circuit" .

[0341] 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, …, (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 Look-ahead circuit but the generate and propagate follow Equations (15a) and (15b) instead. Hence, the logic describes a "Carry Look-ahead" -like circuit.

[0342] Specifically

[0343] · in classical Carry-Look-ahead, 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-Look-ahead" -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) .

[0344] · in classical Carry-Look-ahead, the existence of a generate at a lower-bit and a propagate at 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-Look-ahead" -like deconversion circuit, the existence of a generate at a lower-bit and a propagate at 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.

[0345] · in classical Carry-Look-ahead, 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-Look-ahead" -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 (arelatively lower bit) and propagate (relatively higher bits) .

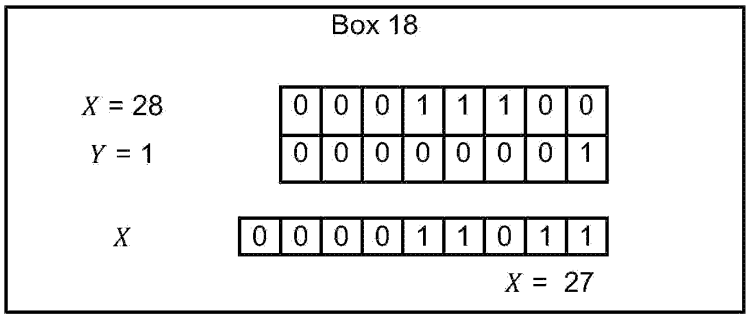
[0346] Figure 30 shows a subcircuit block for de-converting a 4-bit generalized binary number to a 5-bit classical Two's Complement number, with the additional bit being the sign bit. Deconversion into a Two's Complement number will show whether the sum is in fact a negative value or not.

[0347] Inside the subcircuit of Figure 30 is a modified carry-Look-ahead circuit 2101 shown as Figure 31. Subcircuit 2101 is exactly identical to any standard "Look-ahead logic algorithm" available in any undergraduate text in fundamentals of digital logic electronics. 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.

[0348] 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.

[0349] 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.

[0350] 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-Look-ahead" -like deconversion circuit. The final output sum as shown in Box 18.



[0351] 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.

[0352] Subtraction

[0353] 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.

[0354] Alternatively, the binary string of the negative subtrahend can be provided in the classical Two's Complement format for the processing system before generalisation.

[0355] A Second Adder Embodiment

[0356] 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.

[0357] 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.

[0358] Bit Sedimentation Transformation

[0359] 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.

[0360] This Bit Sedimentation Transformation may be expressed mathematically a follows:

$(x'_i)_1 = (x_i)_1 + (x_i)_2$ (17a)

$(x'_i)_2 = (x_i)_1 (x_i)_2$ (17b)

[0361] and

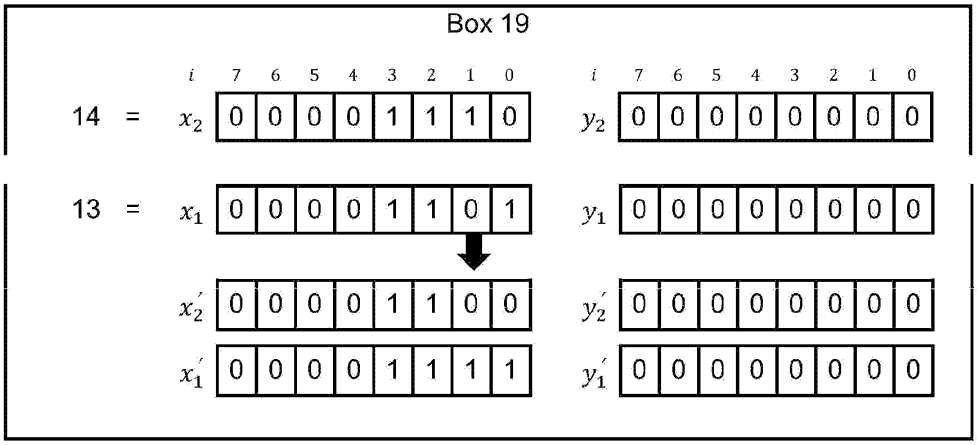
$(y'_i)_1 = (y_i)_1 + (y_i)_2$ (18a)

$(y'_i)_2 = (y_i)_1 (y_i)_2$ (18b)

[0362] 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.

[0363] For example, the generalised substrings for numbers 13 and 14 are shown sedimenting 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 transferrednonzero 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



[0364] Transformation AB

[0365] 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.

$(x_i'')_1 = (x'_i)_2 (x'_{i-1})_1$ (19a)

$$(x_i'')_2 = (\overline{x_i'})_1 (x'_{i-1})_1 \quad (19b)$$

$$(y_i'')_3 = (x_i')_1 (\overline{x_i'})_2 (\overline{x}_{i-1})_1 \quad (19c)$$

[0366] and

$(y_i'')_1 = (y'_i)_2 (y'_{i-1})_1$ (20a)

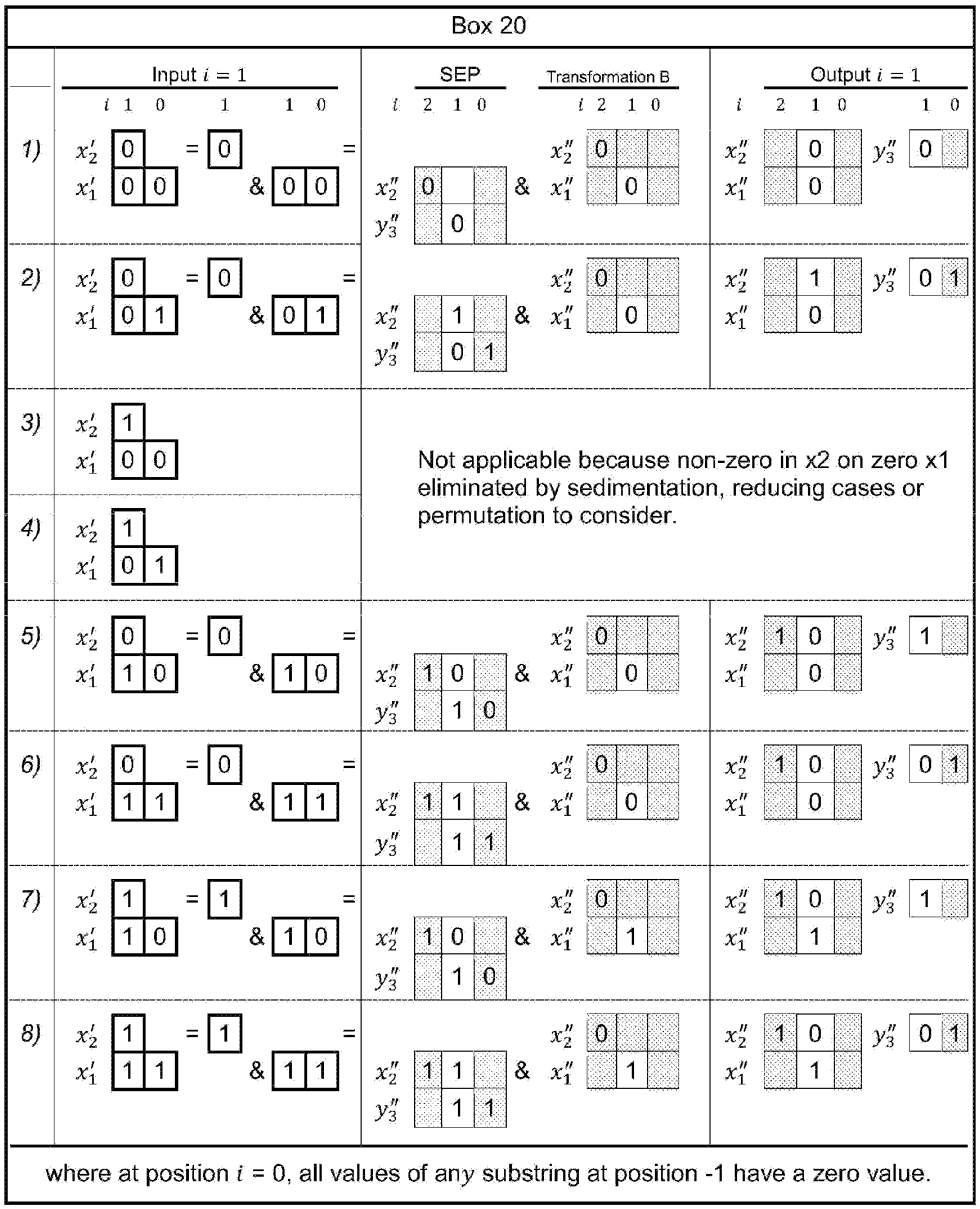
$$(y_i'')_2 = (\overline{y'_i})_1 (y'_{i-1})_1 \quad (20b)$$

$$(x_i'')_3 = (y'_i)_1 (\overline{y'_i})_2 (\overline{y'_{i-1}})_1 \quad (20c)$$

[0367] Figure 35 is the Truth Table for Transformation AB. The asterisks in the Truth Table relate to cases of $(x'_i)_1=0$, $(x'_i)_2=1$, and $(y'_i)_1=0$, $(y'_i)_2=1$, where these cases are excluded in the table as they are prevented due to the Bit Sedimentation Transformation.

[0368] 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

[0369] 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 $x''_1=0$, $y_3''=0$.



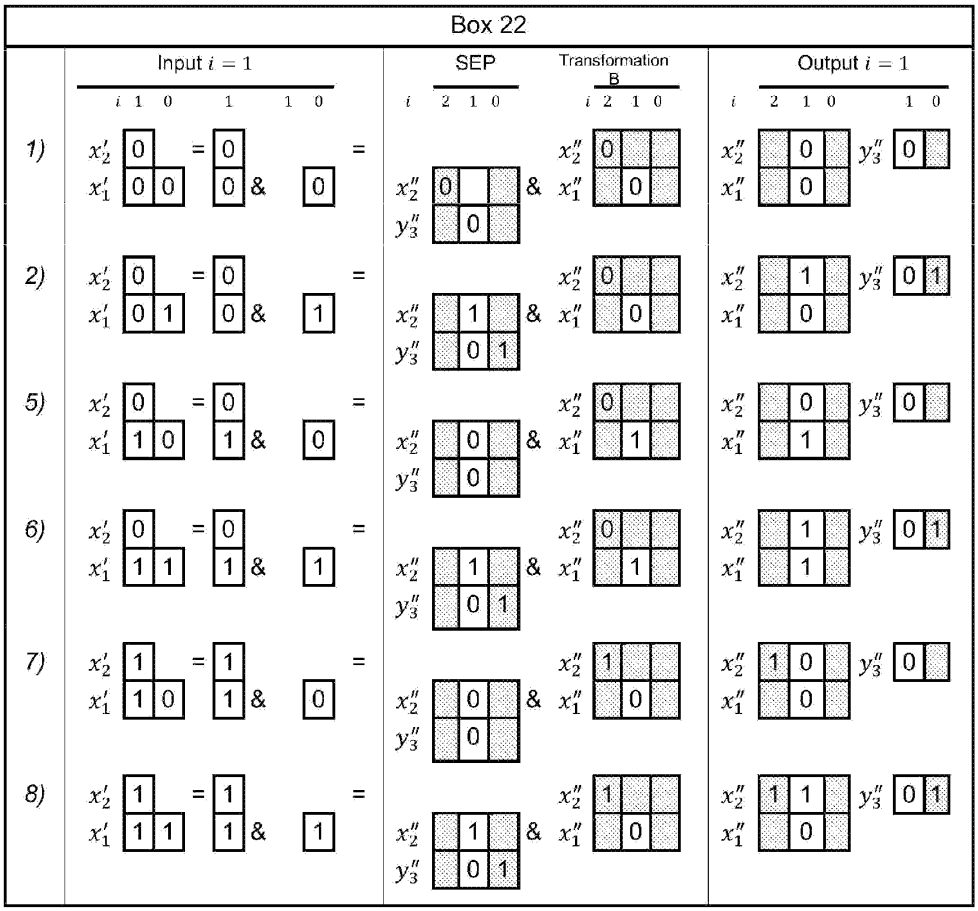
[0370] Box 21 is the same as Box 20 except that Box 21 relates to the y substrings. The similar Case 7 in Box 21, as that in Box 20, should be observed where a further bit simplification should be implemented.

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 1 & y''_1 0 x''_3 0 1	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 1 0 & y''_1 0 x''_3 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 1 1 & y''_1 0 x''_3 1 1	y''_2 1 0 x''_3 0 1 y''_1 0
7)	y'_2 1 = 1 = y'_1 1 0 & 1 0	y''_2 1 0 & y''_1 1 x''_3 1 0	y''_2 1 0 x''_3 1 y''_1 1
8)	y'_2 1 = 1 = y'_1 1 1 & 1 1	y''_2 1 1 & y''_1 1 x''_3 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.

[0371] 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.



[0372] 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 & y''_2 0 y''_1 0 & y''_1 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 & y''_2 0 y''_1 0 1 & y''_1 0 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 & y''_2 0 y''_1 0 & y''_1 1 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 & y''_2 0 y''_1 0 1 & y''_1 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 & y''_2 1 y''_1 0 & y''_1 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 & y''_2 1 y''_1 0 1 & y''_1 0 x''_3 0 1	y''_2 1 y''_1 0	y''_2 1 1 x''_3 0 1 y''_1 0

[0373] Summation of output from Transformation AB

[0374] 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.

[0375] 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.

[0376] 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.

[0377] The following is the corresponding mathematical equations that correspond to the Truth Table.

$$x_i''' = (x_i'')_1 + (x_i'')_2 (\overline{x_i''})_3 + (\overline{x_i''})_2 (x_i'')_3 + (x_{i-1}'')_2 (x_{i-1}'')_3 \quad (21a)$$

$$y''' = (y_i'')_1 + (y_i'')_2 (\overline{y_i''})_3 + (\overline{y_i''})_2 (y_i'')_3 + (y_{i-1}'')_2 (y_{i-1}'')_3 \quad (21b)$$

[0378] 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.

[0379] 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.

[0380] 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.

[0381] 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.

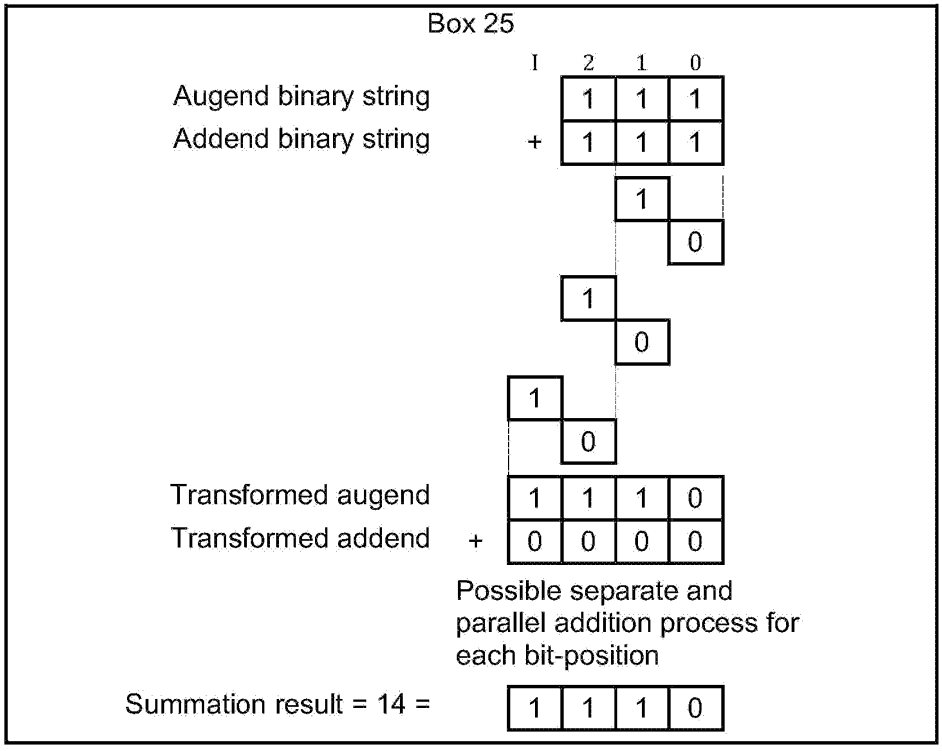
Box 24									
Generalising augend $(\tilde{n}_w)_1$ and addend $(\tilde{n}_w)_2$	$(\tilde{n}_w)_1^+ = (x_i)_1$	1	0	0	1	0	1	0	1
	$(\tilde{n}_w)_2^+ = (x_i)_2$	0	1	0	0	0	1	0	0
	$(\tilde{n}_w)_2^- = (y_i)_2$	0	0	0	0	$\bar{1}$	0	0	$\bar{1}$
	$(\tilde{n}_w)_1^- = (y_i)_1$	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0
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
	$(x_i'')_2 = (\bar{x}_i')_1 (x_{i\,1}')_1$	1	0	0	1	0	1	0	1	0
	$(y_i'')_3 = (x_i')_1 (\bar{x}_i')_2 (\bar{x}_{i-1}')_1$	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
	$(y_i'')_2 = (\bar{y}_i')_1 (y_{i\,1}')_1$	0	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	0
	$(x_i'')_3 = (y_i')_1 (\bar{y}_i')_2 (\bar{y}_{i-1}')_1$	0	0	0	1	0	0	0	0	1
Parallel Bit Addition Operations	Eq. (21a)	1	0	1	0	0	1	0	1	1
	Eq. (21b)	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

[0382] After the sum is obtained, the sum is deconverted into a normal binary string.

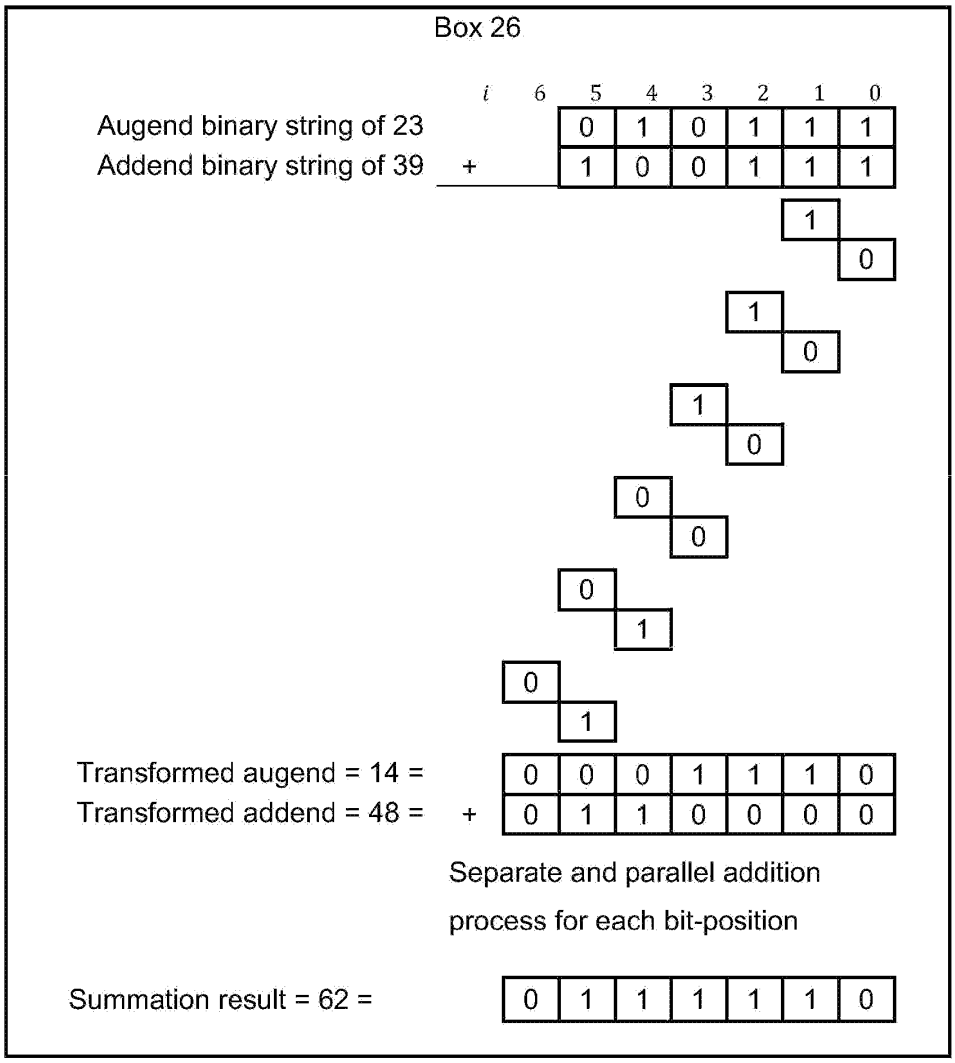
[0383] Embodiments of A Bare Adder

[0384] 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

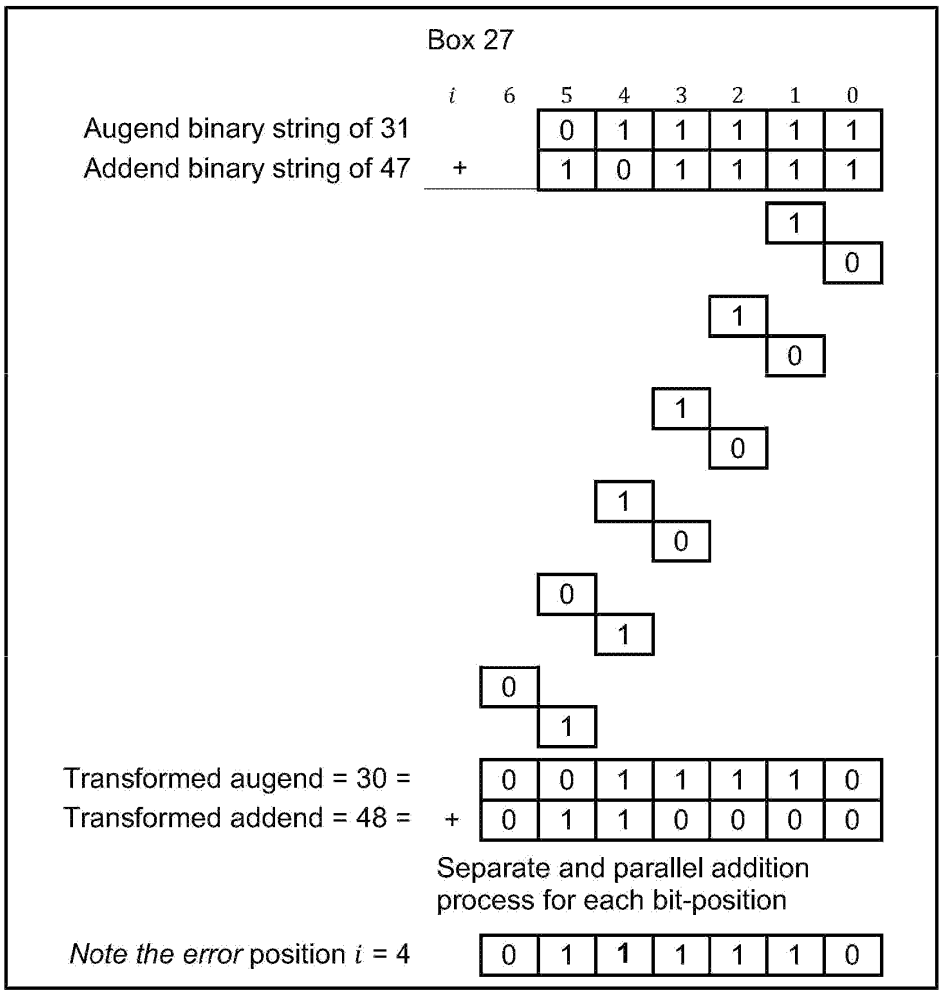
25, in the addition of 7 and 7, i.e. 111 + 111 = 1110 (14 in decimal) , i.e. because the carry-overs have been processed in advance.



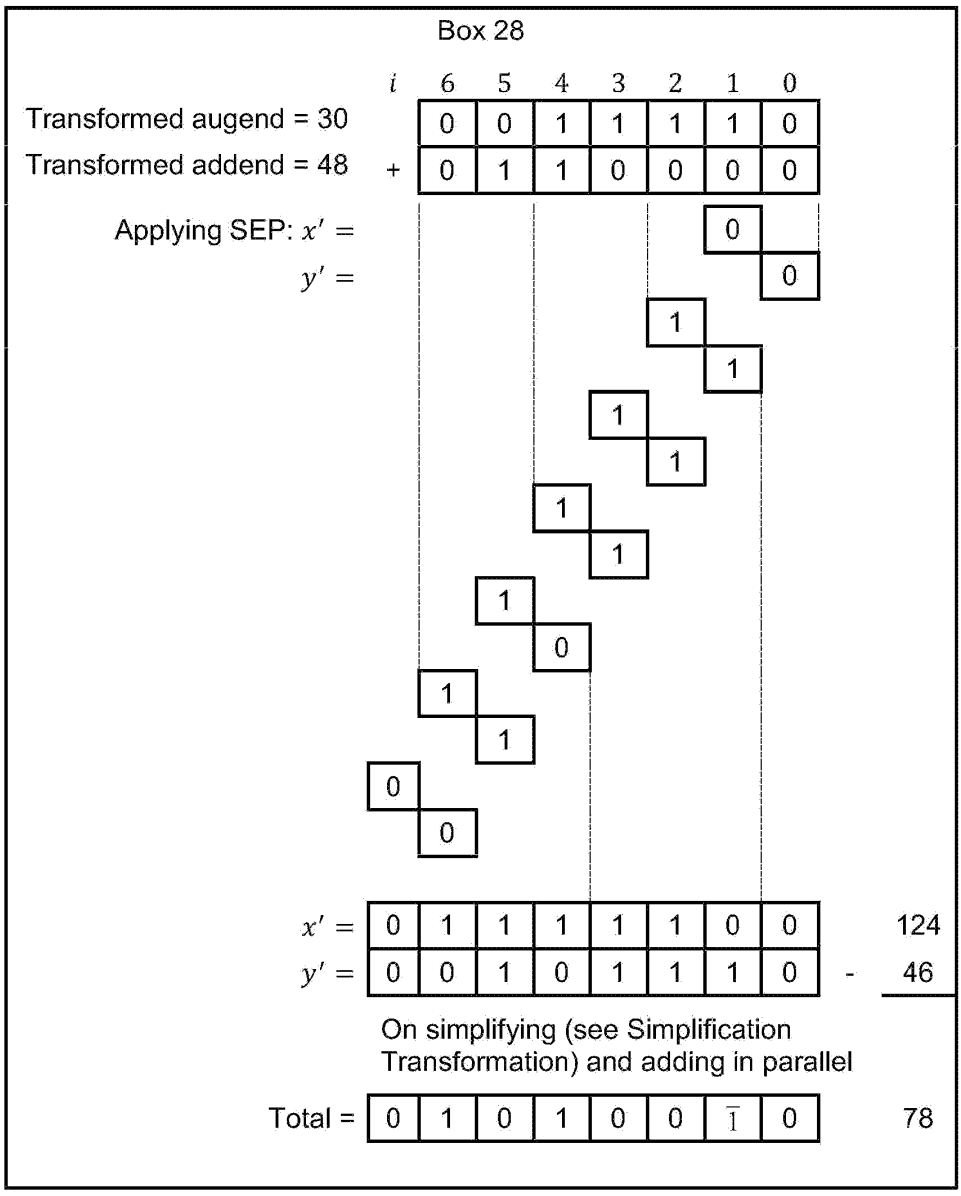
[0385] Transformation B may also be used to processed carry-overs in advance for numbers 23 and 39, as shown in Box 26, so that the processed binary strings can be added up in parallel additions.



[0386] 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.



[0387] 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.



[0388] 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.

[0389] An Adder Embodiment Using Only Transformation A

[0390] 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 Trans-

formation after Transformation A. The following step (c) can be taken to sum up the two pairs of substrings produced.

[0391] (a) Transformation A

[0392] (b) Simplification

[0393] (c) separately adding the positive and negative strings by applying the following equations;

$$x_i'''=\left(x_i''\right)_1\left(\overline{x_i''}\right)_2+\left(\overline{x_i''}\right)_1\left(x_i''\right)_2+\left(x_{i-1}''\right)_1\left(x_{i-1}''\right)_2 \quad (22a)$$

$$y_i'''=\left(y_i''\right)_1\left(\overline{y_i''}\right)_2+\left(\overline{y_i''}\right)_1\left(y_i''\right)_2+\left(y_{i\,1}''\right)_1\left(y_{i\,1}''\right)_2 \quad (22b)$$

[0394] where the logic truth is presented in Figure 43, and the logic circuits in Figure 44 and Figure 45.

[0395] 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.

[0396] 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.

[0397] 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.

[0398] 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.

[0399] 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.

[0400] A First Embodiment Of A Multiplier Circuit

[0401] In multiplication, a multiplicand is added to itself by a number of times indicated by a multiplier. For example, 123 (multiplicand) x 48 (multiplier) means 123 is added up 48 times.

[0402] In the prior art, both a multiplicand and a multiplier are supplied as inputs to a multiplier circuit in the regular form of single-strand binary strings, from which the multiplication product is calculated using a two stage process. In the first stage, the multiplicand is expanded by every bit of the multiplier to provide a number of binary strings called the partial products. Each partial product is replica of the multiplicand if the multiplier bit is 1 or a string of zeros if the multiplier bit is 0, but position-shifted so that the smallest bit of each partial product is in same position as the respective multiplier bit. In a corresponding array type logic circuit, the expansion is hardwired and done concurrently for every permutation of a multiplicand bit multiplied by a multiplier bit. The multiplication operator for two single-bits can be just an AND gate. There are only four possible input combinations to the two single-bit inputs, as shown on the left of the equations in Box 29a, of which the respective product is a single bit output of the value shown on right side.

Box 29A
1 x 0 = 0
0 x 1 = 0
0 x 0 = 0
1 x 1 = 1

[0403] In the second stage, the partial products are added up and their sum is the multiplication product. If the partial products are added up one after another, a set of 8 partial products requires 7 addition operations to sum, each addition being one iteration in the serial order. If done in parallel, 7 addition operations in a total of 3 iterations are required. This is because the first iteration reduces 8 partial products to 4 new partial products, the second iteration reduces the 4 partial products to 2 new partial products and finally a third iteration reduces the 2 partial products to the final product.

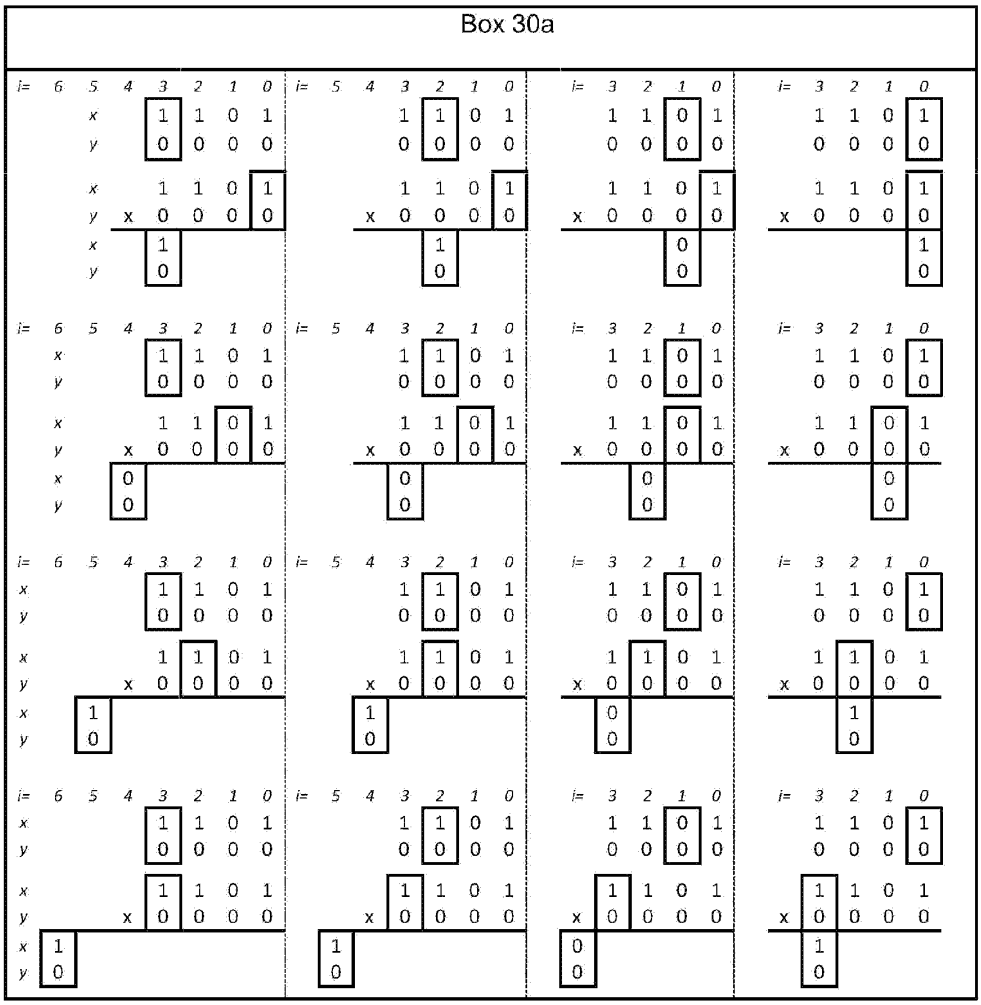
[0404] The present embodiment is a multiplier circuit that applies a similar two-stage operation but the multiplicand and multiplier as well as their partial products are generalised binary numbers, which may be added up using embodiments of the adder described in the preceding sections.

[0405] A generalised binary number will be called by a shortened “generalized number” for convenience in the rest of this description.

[0406] That is, every two such partial products to be added up are pre-treated by Transformation A, followed by a simplification operation, and then by Transformation B. Transformation A is described in paragraphs related Box 9 and box 10, and Trans-

formation B is described in paragraphs related to Box 13 and Box 14. Transformation A and Transformation B relate to making available a bit in every position i in one of two binary strings to receive a carry-over value without having to propagate the carry-over the next higher bit-position, and comprise 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. Variations or combinations of Transformation A and Transformation B may be used instead. For convenience, these methods will be referred to as AB methods and the relevant adders as AB adders in the rest of this specification. The following describes the workings of 13 x 13 x 13 in a multiplier circuit using AB adders for adding up partial products. 13 x 13 is completed first before the product is multiplied by 13.

Box 30a



[0407] The regular binary string for 13 is 1101. The simplest way to generalize a regular binary string is to designated it the x substring, and paired with a newly generated binary string that has zero value for the y substring, i.e. x substring = 1101 is paired with a y substring = 0000. Expanding two such generalised numbers produces partial products which are also generalised numbers, as illustrated in Box 30a. In every bit-position of each partial product is a pair of x and y bits obtained by multiplication of two pairs of x and y bits, one from each of the multiplicand and multiplier.

[0408] Box 30b shows all the four possible combinations of such two 2-bit inputs that are consequential; there is no need to show the combinations involving a pair of x and y bits that are both zeros, as the product is invariably also pair of x and y bits which are both zeros. Also, any pair of x and y bits which are both nonzeros self-cancels into zeros since the y bit represents a value to be taken from the x bit..

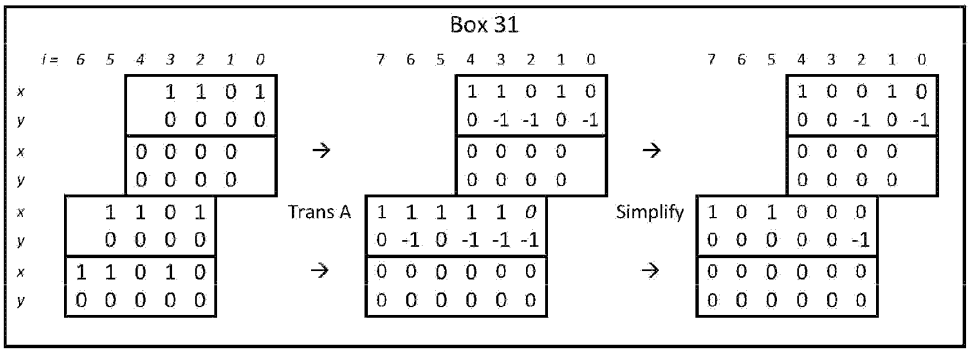
Box 30b

x	0	x	1	= -1 x 1 = -1 =	0
y	-1		0		-1
x	1	x	1	= 1 x 1 = 1 =	1
y	0		0		0
x	0	x	0	= -1 x -1 = 1 =	1
y	-1		-1		0
x	1	x	0	= 1 x -1 = -1 =	0
y	0		-1		-1

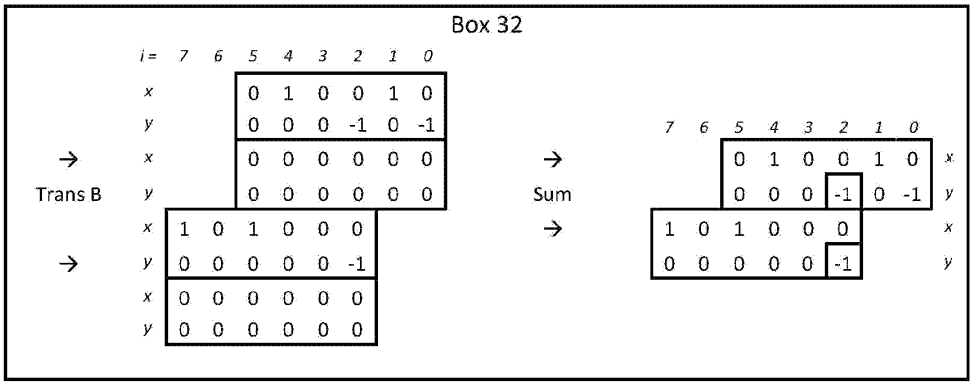
[0409] The partial products in Box 30a are shown compiled into four partial products on the left side in Box 31, the four coming from the four-bit length of the multiplier.

[0410] The right side in Box 31 shows the partial products on the left side after being treated , first by Transformation A and then simplified. Transformation A is applied to two generalised numbers about to be added together. At this point of the example, it would be the top two pairs of x and y substrings for one addition operation, and the bottom two pairs of x and y substrings for another addition operation. Preferably, both addition operations are done concurrently, although these addition operations can be done successively in less efficient embodiments.

[0411] As explained in preceding paragraphs, simplification refers to a transformation by which nonzero bits in the same position in both x and y substrings are replaced with zeros due to mutual cancellation, a transformation not as conveniently applied if y substrings represented a negative values in 2's Complement which requires filling leading bit-positions with nonzeros unlike absolute values.



[0412] The simplified data on the left in Box 31 is then treated with Transformation B to produce the data shown on the left side in Box 32. Subsequently, the treated versions of the generalised numbers are added up to give the data on the right side in Box 32.



[0413] Where one iteration involves adding up of every two partial products concurrently, the number of partial products is reduced by half after every iteration. This strategy is described in as the LimTree method in other parts of this description.

[0414] In an array-type multiplier circuit, as many iterations as needed for a certain number of partial products to be added into one generalised number is hardwired in the circuit, based on pre-determined string lengths of the multiplier. An 8-bit multiplicand and a 4-bit multiplier circuit has to add up four partial products in two iterations. In a 64-bit processor embodiment, a multiplier of 64 bits produces 64 partial products, each a pair of x and y substring pairs, requiring six iterations to sum up completely.

[0415] Accordingly, a second iteration is required to add up the two new partial products shown on the right side in Box 32. Transformation A is applied the new partial products followed by simplification, giving the data shown on the left side in Box 33.

Box 33

	i=	8	7	6	5	4	3	2	1	0			8	7	6	5	4	3	2	1	0
Trans A		1	0	1	0	0	0	1	0	0			1	0	1	0	0	0	1	0	1
→		0	-1	0	0	-1	0	0	-1	0		Trans B	0	-1	0	0	-1	-1	0	0	0
Simplify		0	0	0	0	0	0	0	0	1		→	0	0	0	0	0	0	0	0	0
		0	0	0	0	0	-1	0	-1	0			0	0	0	0	0	0	-1	0	0

[0416] Subsequently, Transformation B is applied to the data on the left side in Box 33, which gives the data on the right side of Box 33. The data on the right side of Box 33 are then added up, which gives the generalised number in Box 34 that is the product of 13 x 13 = 169.

Box 34

	i =	8	7	6	5	4	3	2	1	0	
SUM		1	0	1	0	0	0	0	0	1	= 169
→		0	-1	0	0	-1	-1	0	0	0	

[0417] The product s multiplied again by the same number 13, in which the first stage produces the partial products shown in Box 35. The details of the expansion are omitted this, being similar to those shown in Box 30a.

Box 35

i=	9	8	7	6	5	4	3	2	1	0
		1	0	1	0	0	0	0	0	1
		0	-1	0	0	-1	-1	0	0	0
							1	1	0	1
x							0	0	0	0
	0	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	0	

	8	7	6	5	4	3	2	1	0
	1	0	1	0	0	0	0	0	1
	0	-1	0	0	-1	-1	0	0	0
						1	1	0	1
x						0	0	0	0
	1	0	1	0	0	0	0	0	1
	0	-1	0	0	-1	-1	0	0	0

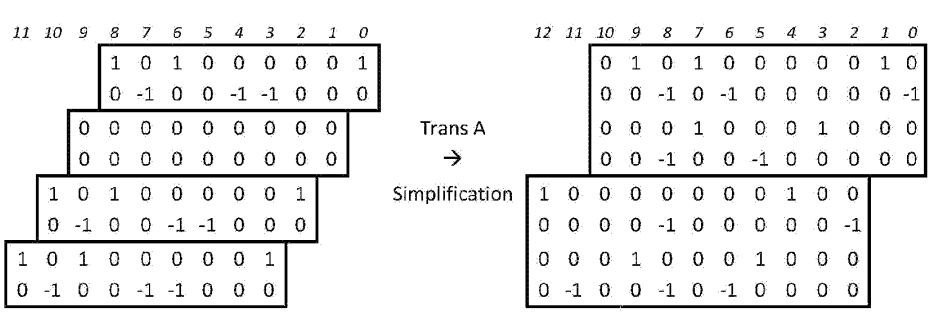
i=	11	10	9	8	7	6	5	4	3	2	1	0
				1	0	1	0	0	0	0	0	1
				0	-1	0	0	-1	-1	0	0	0
									1	1	0	1
		x							0	0	0	0
	1	0	1	0	0	0	0	0	1			
	0	-1	0	0	-1	-1	0	0	0			

	10	9	8	7	6	5	4	3	2	1	0
			1	0	1	0	0	0	0	0	1
			0	-1	0	0	-1	-1	0	0	0
								1	1	0	1
		x						0	0	0	0
	1	0	1	0	0	0	0	0	1		
	0	-1	0	0	-1	-1	0	0	0		

[0418] The partial products in Box 35 are shown consolidated on the left side of in Box 36, comprising four generalised numbers. In a first iteration, every two of the partial

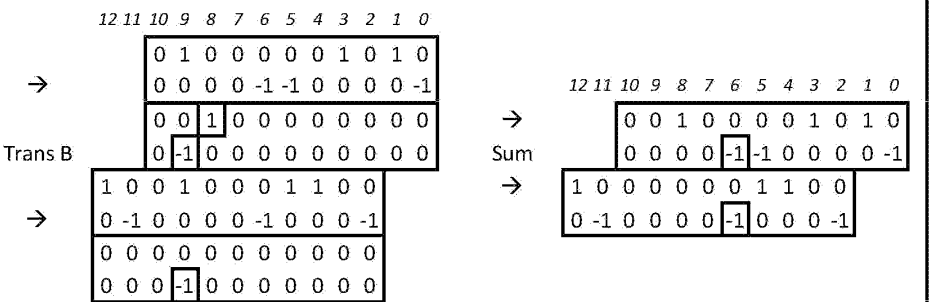
products are treated with Transformation A, followed immediately by simplification, which provides the data on the right side in Box 36.

Box 36



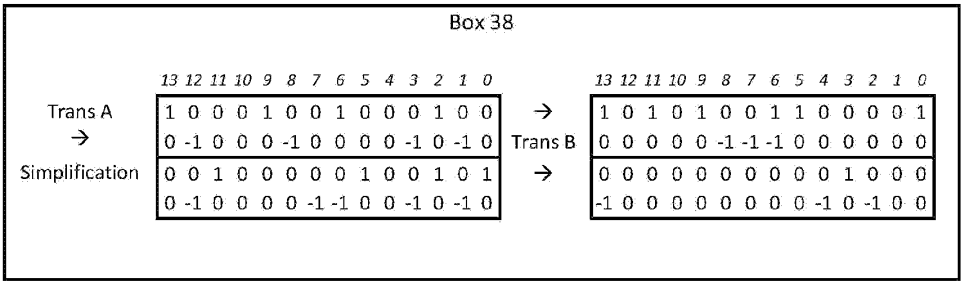
[0419] The data on the right side in Box 36 is then treated with Transformation B, the result of which is shown on the left side in Box 37. Subsequently, every two of the treated partial products are added up to give the new partial products shown on the right side in Box 37.

Box 37



[0420] A second iteration commences with Transformation A applied to the remaining two partial products on the right side in Box 37, followed by simplification, which produces the data on the left side in Box 38. Transformation B is then applied, which produces the data on the right side in Box 38.

Box 38



[0421] Box 39a shows the data on the right side in Box 38 added up, which is a generalised number having the value of 2197 for 13x13x13.

Box 39a

	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Sum	0	0	1	0	1	0	0	0	1	0	1	0	0	1	=	2197
	0	0	0	0	0	-1	-1	0	0	-1	0	-1	0	0		

[0422] The generalised number in Box 39a may then be deconverted into a regular, single-strand, binary string by the process of deconversion described in relation to Figure 30. generalized binary number

[0423] The above steps can be implemented an array-type multiplier circuit 6601 illustrated as a block circuit in Figure 66. The multiplier circuit 6601 has input leads 6603 for an 8-bit multiplicand $(n_w)_1$ and a 4-bit multiplier $(n_w)_2$, and output leads for a product of up to a 12-bit length.

[0424] The multiplicand and multiplier are generalised numbers. Therefore, there are two 8-bit input leads for the multiplicand's x and y substrings. Similarly, there are two 4-bit input leads for the multiplier, and two 12-bit output leads for the product.

[0425] Figure 67 is an illustration of the sub-circuits in the multiplier circuit 6601, which includes an expansion circuit 6701 for producing partial products, treatment subcircuits 6705 for transforming the partial products to be suitable for added up by adder sub-circuits 6707 without need of carry over propagation.

[0426] The expansion circuit 6701 is connected directly to the input leads 6603 on the circuit block in Figure 66., which distributes the input data to four expansion sub-circuits 6701a, the details of which can be seen in Figure 68.

[0427] Each expansion subcircuit 6701a receives the entire multiplicand as input but only a respective pair of multiplier x and y bits 6803, and outputs a partial product obtained by expansion of the multiplicand and the pair of multiplier x and y bits. Although not shown in the illustration, it should be understood that partial product is position-shifted according the multiplier x and y bits, and zero is attributed to all the bits in positions shifted over. The logic circuit of an expansion subcircuit 6701a is shown as Figure 69, where it can be seen as including eight multiplication subcircuits 6901.

[0428] Each of the multiplication subcircuits 6901 has two input leads, each input lead for a pair of x and y bits. One of the two input leads is connected to input leads 6603 on the multiplier circuit 6601 that corresponds to the respective pair of multiplier x and y bits, so that the pair of multiplier x and y bits a common factor to all the multiplication subcircuits 6901. The other input lead of the multiplication subcircuits 6901 is also

connected to input leads 6603 on the multiplier circuit 6601, but to a respective pair of multiplicand x and y bits and according to their bit-positions. The inputs supplied in this way allow each of the multiplication subcircuits 6901 to output a pair of single bits for the respective bit-position in the x and y substrings of the partial product for.

[0429] Figure 64 shows is a block diagram of a multiplication subcircuit 6901 while Figure 65 shows the logic circuit inside, which operates according to the Truth Table in Figure 63 and the following equations, which relates to multiplying a multiplicand $(n_w)_1$ by a pair of single x and y bits of multiplier $(n_w)_2$.

[0430] In mathematical expressions, a method of multiplying a generalized binary number $(n_w)_1$ multiplicand by a generalized binary number $(n_w)_2$ multiplier includes the steps of

[0431] (a) expanding the multiplicand $(n_w)_1$ substrings $(x_i)_1$, $(y_i)_1$ for i=0, 1, 2…, ; by each bit-pair of multiplier $(n_w)_2$ substrings of$(x_j)_2$and $(y_j)_2$for j=0, 1, 2, …, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...; where

$(x'_i)_j = (x_i)_1(x_j)_2 + (y_i)_1(y_j)_2$ (23a)

$(y'_i)_j = (y_i)_1(x_j)_2 + (x_i)_1(y_j)_2$ (23b)

[0432] for i=0, 1, 2…and j=0, 1, 2, …; in which

[0433] $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$;

[0434] (x'_i) i, (y'_i) i are the bit-pairs of $(n'_w)_1$, etc.

[0435] Subsequently, every two of the partial products are fed to a treatment subcircuit 6705 to be treated and then to an adder subcircuit 6707 to be added up in parallel addition operations for every bit position. The adder subcircuit 6707 outputs the sum as a new partial product, which is also a generalised number.

[0436] In turn, every two new partial products are fed into another treatment subcircuit 6705 and adder subcircuit 6707, until the final product of the multiplication is obtained. Each round of addition using treatment subcircuits 6705 and adder subcircuits 6707 represents one iteration. Therefore, the treatment subcircuits 6705 and adder sub-circuits in Figure 67 provide two iterations.

[0437] The treatment subcircuit 6705 and adder subcircuit 6707 in each iteration have a combined logic circuit such as that shown Figure 25, which includes the subcircuits A42XY11, 2CV-21 and A4-XY12 and A4-2XY13 for Transformation A, Simplification Transformation, Transformation B, and addition. These are discussed with reference to Figure 13 to Figure 21 in preceding paragraphs.

[0438] A Second Embodiment Of A Multiplier Circuit (semi-optimisation method)

[0439] Before a second embodiment of a multiplier circuit is described, a different set of pre-treatments of generalised numbers to be added up without propagation of carry-overs are introduced. These pre-treatments are called optimisation and adders for

adding up optimised generalised numbers are called Optimised Adders in the rest of this specification. Optimisation can be sub-divided into semi-optimisation and full-optimisation, but it suffices that the following paragraphs describe only semi-optimisation.

[0440] It should be noted in advance regardless of the type of adders implemented in different multiplier circuit embodiments, logic circuits for operations not related directly to the pre-treatment or adding up of generalised numbers can be implemented in all the embodiments without partiality. For example, pre-addition operations such as generalisation of a regular binary string into the generalised number format, or post-addition operations such as deconversion of a generalised number into a regular binary string are useable in all multiplier circuit embodiments. Accordingly, there is no need to describe again these circuits and their operations.

[0441] A semi-optimized generalized-binary-string$n_w$has the following characteristics:

[0442] 1. every two nonzero bits in each substring are separated by at least one zero bit;

[0443] 2. a nonzero can be immediately next to another nonzero if one nonzero is in one substring and the other nonzero is in the other substring;

[0444] 3. no nonzeros can be in the same position in both substrings, as these cancels out to zero in both substrings.

[0445] 4. there is no limitation to the number of zero bits.

[0446] The following is a generalized number in semi-optimized arrangement of nonzeros.

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

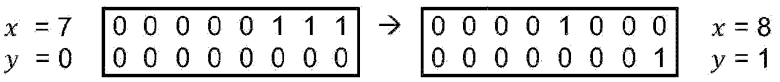
[0447] Semi-optimization can be described by (2n -n) , where n is the entire generalised number comprising both x and y substrings.

[0448] The generalized number shown on the left below, having substrings x=3 and y=0 representing a combined value of 3, is not semi-optimized. This is due to two adjacent nonzeros in the first and second bit-positions in the x substring. Semi-optimization of this generalized number produces the generalized number shown on the right, which has x and y substrings that mean 4 - 1 = 3, and may be written as $0000\,0\,10\,\bar{1}$ in the recommended shorthand. There is no change in their combined value, keeping to the Equivalence Principle. Recall that 1 in y substrings is written interchangeably as -1 in other parts of this description.

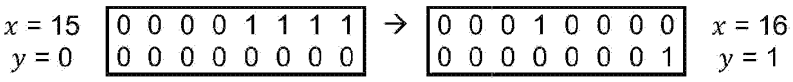
$x = 3$	0 0 0 0 0 0 1 1	→	0 0 0 0 0 1 0 0	$x = 4$
$y = 0$	0 0 0 0 0 0 0 0		0 0 0 0 0 0 0 1	$y = 1$

[0449] The substrings x and y immediately below, on the left, means 7 –0 representing the combined value 7. Thus, the generalized number, n, is not semi-optimized because

of three adjacent nonzeros in the x substring. Semi-optimization gives the x and y substrings on the right, meaning 8 - 1, which $0000100\bar{1}$ in shorthand.



[0450] The generalized number on the left below means 15 –0 = 15 and is semi-optimized into the generalized number on the right below which means 16-1=15, which is $0001000\bar{1}$ in shorthand.

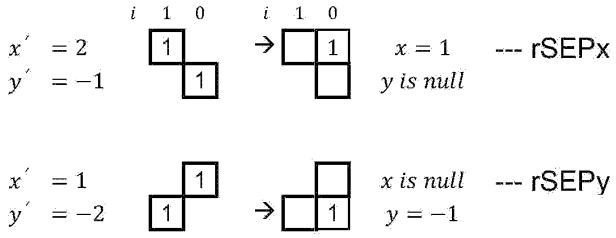


[0451] Reverse SEP

[0452] Before showing how semi-optimisation is implemented in logic circuits, the concept of Reverse SEP is introduced first.

[0453] Reverse SEP (rSEP) reduces two immediately adjacent nonzeros, in immediately neighbouring positions i and i +1, each in either one of a pair of x and y substrings, by replacement of the four bits with a single nonzero in the position of the smaller adjacent nonzero but in the opposite substring, as shown in Box 39b. The is just the value of the x substring bit minus the value of y substring bit in immediately adjacent positions, reducing the number of nonzeros with no change in combined value.

Box 39b



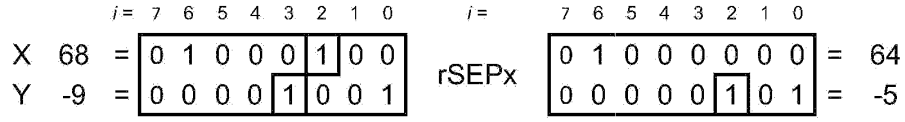
[0454] An rSEP operation can be considered as a combination of two versions of sub-operations. In a version called rSEPx, adjacent nonzeros in which the larger of the adjacent nonzeros in the x substring and the smaller in the y substring are replaced by a nonzero in the x substring in the same bit-position as the replaced smaller nonzero.

[0455] In a complementary version called rSEPy, adjacent nonzeros in which the larger of the adjacent nonzeros in the y substring and the smaller in the x substring are replaced by a nonzero in the y substring in the same bit-position as the replaced smaller nonzero. In some situations, only either one of rSEPx and rSEPy is applied. In other situations, rSEP (meaning both rSEPx and rSEPy) is applied in a single step.

[0456] The rSEP should not be confused with the SEP conversion described in Box 1 and Box 3. The following shows a generalised number having x and y substrings meaning 59 -0 being semi-optimised by applying SEP conversion onto the x substring only (shorthand SEPx) .



[0457] The Truth Table for rSEPx is shown in Figure 15 and the logic circuit thereof is in Figure 14. The logic circuit and Truth Table for rSEPy is just in the reverse sign and need not be provided. The above transformed generalised number is then treated by rSEPx as shown from left to right.



[0458] Both the rSEP and SEP conversions reduce the numbers of nonzeros in a generalised number but in different ways.

[0459] Logic circuits for semi-optimising a generalised number in the preferred embodiments may comprise execution of the following three steps.

[0460] · Step 1. Apply SEP to the y substring but not the x substring of a generalised number. The combined effect is similar to the example in Box 3 but in the opposite sign substring, resulting in replacement of a series of adjacent nonzeros in the y substring with zeros, a nonzero one order higher than the largest nonzero in the series and a nonzero of opposite sign in the same position of the smallest nonzero in the series.

[0461] · Step 2. Apply rSEPx to the output of Step 1.

[0462] · Step 3. Apply SEP to the x substring of the output of Step 2.

[0463] It is just an arbitrary choice that Step 1 operates on y substring. The converse is included in the contemplation of the embodiments, where Step 1 applies SEP on the x substring, Step 2 applies rSEPy, and Step 3 applies SEP on the y substring of the output of Step 2.

[0464] Figure 47 shows a subcircuit for semi-optimization of an 8-bit generalised number into a semi-optimized generalised number of 9 bits. Figure 47 comprises three blocks of subcircuits as shown in Figure 48, labelled with codenames Vin8o08-31, Vin8o08-32 and Vin8o08-33. The digital block circuit of Figure 47 operates on the substrings by each bit-position in parallel.

[0465] VinSo08-31, which is shown in greater detail in Figure 49 as comprising a plurality the basic subcircuits shown in Figure 50, is used to execute Step 1 to eliminate any

string of adjacent $\overline{1}$ by applying SEP conversion to the y substring according to following equations and Truth Table in Figure 51 for eliminating strings of $\overline{1}$.

$$x_i' = x_i\bar{y}_{i-1} + y_i\bar{y}_{i-1} \quad (24a)$$

$$y_i' = \bar{x}_i\bar{y}_i y_{i-1} \quad (24b)$$

[0466] VinSo08-32, which is shown in greater detail in Figure 52 as comprising a plurality of the basic subcircuit shown in Figure 53, is used to execute Step 2 and apply rSEPx on the output of Step 1, according to the following equations and Truth Table shown in Figure 54.

$$x_i'' = x_i'\bar{y}_{i-1}' + x_{i+1}'y_i' \quad (25a)$$

$$y_i'' = \bar{x}_{i+1}'y_i' \quad (25b)$$

[0467] VinSo08-33, which is shown in greater detail in Figure 55 as comprising a plurality of the basic subcircuit shown in Figure 56 is used to execute Step 3 to eliminate any string of adjacent 1 by applying SEP conversion to the x substring in the output of Step 2, according to the following equations and Truth Table shown in Figure 57.

$$\tilde{X}_i = \bar{x}_i''\bar{y}_i''x_{i-1}'' \quad (26a)$$

$$\tilde{Y}_i = x_i''\bar{x}_{i-1}''\bar{y}_{i-1}'' + y_i''\bar{x}_{i-1}''\bar{y}_{i-1}'' \text{ or simply } \tilde{Y}_i = x_i''\bar{x}_{i-1}'' + y_i''\bar{x}_{i-1}'' \quad (26b)$$

[0468] where $\tilde{X}_i$ and $\tilde{Y}_i$ denotes the semi-optimized generalised binary strings of the addend.

[0469] Figure 58 is a circuit block diagram of a "Semi-optimised" adder by which two 9-bit semi-optimized generalised binary substrings x_1, y_1, x_2, y_2 can be summed up. The output is a 10-bit generalised number. Figure 59 shows the circuit block in Figure 58 as comprising ten subcircuits, each of which is an adder as shown in Figure 60. The subcircuits in Figure 59 operate in parallel and concurrently, being simply reactive to the respective inputs. Figure 61 is the logic circuit of an adder unit, and two such adder units are combined to provide the adder shown in Figure 60 in order to provide an output in the generalised number format.

[0470] For each bit-position i, the addition algorithm considers the adjacent bits in positions i and i -1 in substrings x_1 and x_2 for positive values to produce an output bit X in the position i. Similarly, for each bit-position i, the addition algorithm considers the adjacent bits in positions i and i -1 in substrings y_1 and y_2 for values to produce in an output bit Y in position i. The addition algorithm in each adder just calculates the output for the corresponding bit-position independently of the other adders.

[0471] Semi-optimisation ensures that one in every two bits is zero, i.e. an interleaving zero bits, which is able to accepting any carry-over without propagating the carry-over forwards. The concatenated strings of output bits X and Y represent the sum in generalized number format.

[0472] The addition algorithm can be expressed in the following two steps, with the sum denoted by N_w whereas the semi-optimized augend and addend are denoted by $(\tilde{n}_w)_1$ and $(\tilde{n}_w)_2$.

[0473] (a) separately adding the positive and negative components by using the following equations;

$$x_i' = (\tilde{x}_i)_1(\bar{\tilde{x}}_i)_2 + (\bar{\tilde{x}}_i)_1(\tilde{x}_i)_2 + (\tilde{x}_{i-1})_1(\tilde{x}_{i-1})_2 \quad (27a)$$

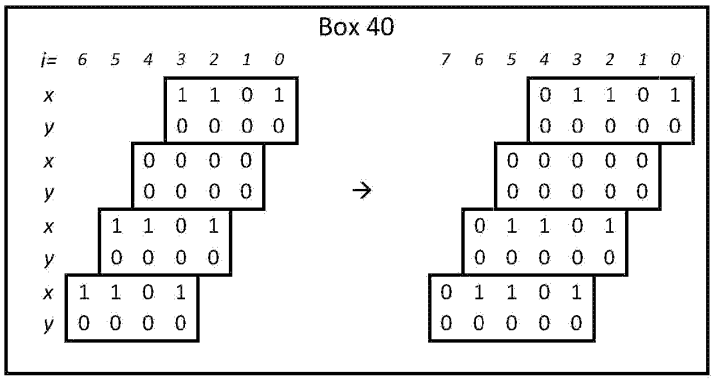
$$y_i' = (\tilde{y}_i)_1(\bar{\tilde{y}}_i)_2 + (\bar{\tilde{y}}_i)_1(\tilde{y}_i)_2 + (\tilde{y}_{i-1})_1(\tilde{y}_{i-1})_2 \quad (27b)$$

[0474] of which the logic truth and logic circuit are presented in Table 12 in Figure 62) and Figure 61.

[0475] The following is an example of the working in a multiplier circuit implemented with semi-optimised adders for 13 x 13 x 13. The partial products are obtained in the same way as describe for Box 30a, which does not need to be repeated. The multiplier circuit applies the semi-optimisation transformation on each partial product by the described 3 Steps.

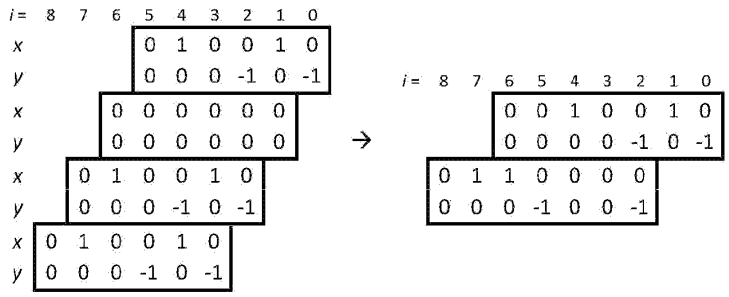
[0476] Partial products obtained in Box 30a shown on the left in Box 40, and semi-optimisation of the y-substrings give the data on the right. Incidentally, as there is no nonzeros in the y substrings, the output is unchanged.

Box 40



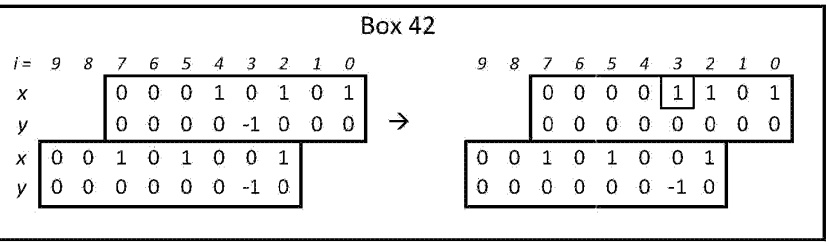
[0477] Subsequently, the data on the right in Box 40 is treated with rSEPx. Again, incidentally, as there is no nonzeros in the y substrings, the output is unchanged by rSEPx. This is shown as the identical data on the right in Box 40.

Box 41



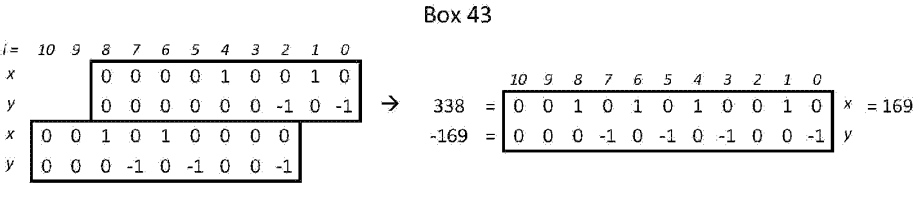
[0478] Subsequently, SEP is applied to the x substrings on the data on the right in Box 40 which produces the data on the left side in Box 41. If the results has a nonzero in the same position in each of the x and y substrings, the SEP logic circuit as shown automatically cancels them to zeros in an operation similar to simplification. Then, every two of the now semi-optimised partial products are added up, wherein x substring is added to x substring, and y substring is added to y substring, and the result is simplified to give the output. The semi-optimized adder follows the logic in Table 12 in Figure 62 and the circuits in Figure 60 and Figure 61. Thus, the output has half the number of partial products as shown on the right side in Box 41. This completes one iteration of adding up every two partial products. A further iteration is required to arrive at the final sum, starting with semi-optimising the y substrings on the data on the right side in Box 41, giving the output on the left side in Box 42, with the result simplified before output.

Box 42



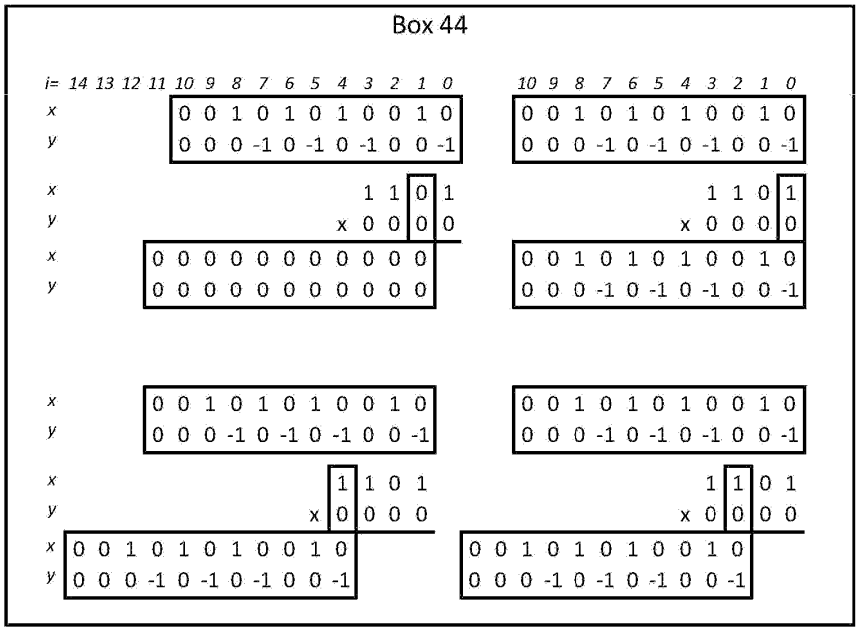
[0479] The right side in Box 42 shows the data on the left side in Box 42 after treatment with rSEPx. Subsequently, applying SEP onto the x substrings on the right in Box 42 gives the data on the left in Box 43.

Box 43



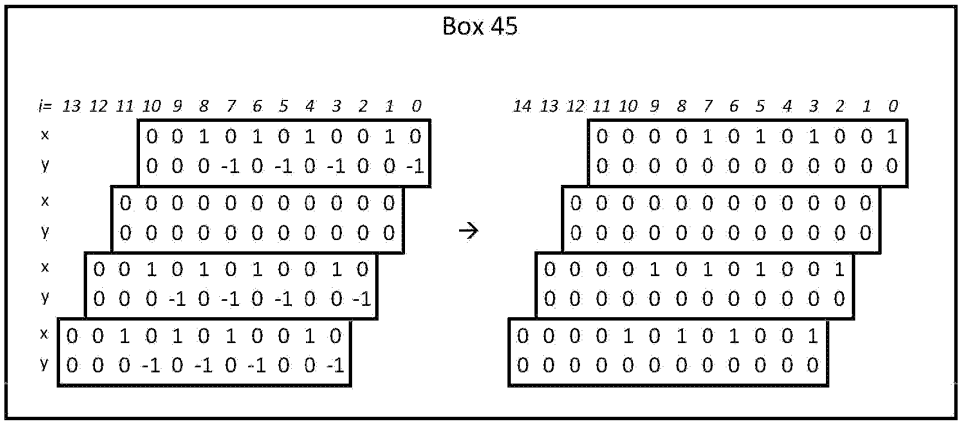
[0480] Adding up the x substrings and the y substrings separately gives the data on the right in Box 43, which is the multiplication product. This completes the second iteration. The product is now used as the multiplicand in a subsequent multiplication, in which case the product is not deconverted into the regular binary format. The multiplier is again the number 13, which gives the partial products with the multiplicand as shown in Box 44.

Box 44



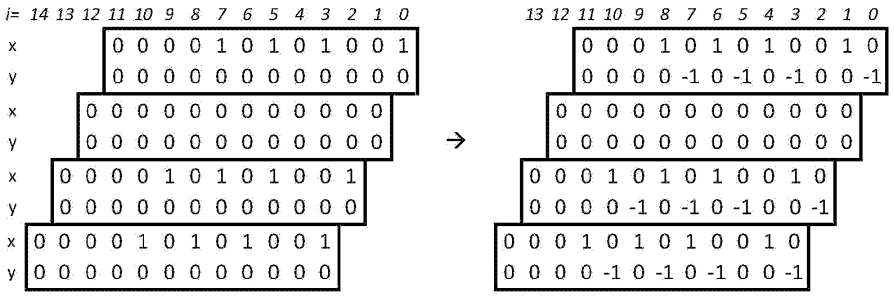
[0481] The partial products are compiled into the left side in Box 45. Semi-optimising the y substrings followed by simplification gives the data on the right in Box 45.

Box 45



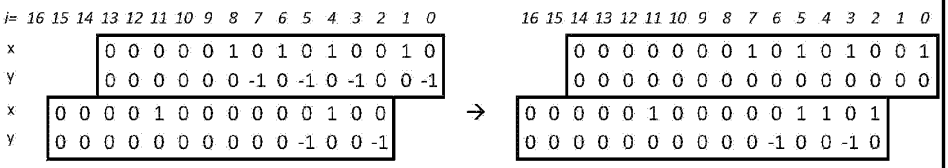
[0482] The data on the right in Box 45 is treated with rSEPx to give the data on the left side in Box 46. There is no change because there is no nonzeros in the y substrings incidentally. Subsequently, the x-substrings are semi-optimised and simplified to give the data on the right in Box 46.

Box 46



[0483] Adding up the x substrings and the y substrings separately gives the data on the left in Box 47.

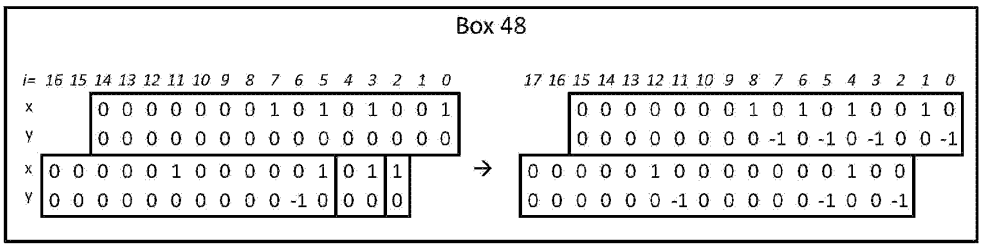
Box 47



[0484] However, a further iteration is required to arrive at the final sum, starting with semi-optimising the y substrings on the left side in Box 47, the se result being simplified before output as the data shown on the right side in Box 47.

[0485] Then, the data on the right side in Box 47 is treated with rSEPx. Incidentally, there is only spot to be converted, which is in the lower pair of x and y substrings. This gives the result on the left in Box 48.

Box 48



[0486] Treating the data on the left in Box 48 by semi-optimisation of the x substrings followed by simplification gives the data on the right.

Box 49

i=	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	1	0	4386	= 2197
	0	0	0	0	0	0	-1	0	0	0	-1	0	0	0	-1	-1	0	-1	-2189	

[0487] Adding up the data on the right in Box 48 above ends the second iteration and gives the product in Box 49.

[0488] The product can be used in any further data processing. However, if arriving at the product is the end of a series of processes, to deconvert the product from the generalised number format into the regular binary format, the deconversion steps as discussed in the previous sections describing embodiments of adders are applied.

[0489] A multiplier circuit for this embodiment has the same structure as that shown in Figure 66. The subcircuits, illustrated and arranged as shown in Figure 67, Figure 68 and Figure 69 apply to this embodiment as well as to the first embodiment. There is no need to repeat the same description for the expansion circuits and expansion subcircuits.

[0490] The main difference distinguishing the multiplier circuits of the first and second embodiments is inside the treatment subcircuits 6705 and the adder subcircuit 6707. Externally, however, the treatment subcircuits 6705 and the adder subcircuit 6707 are arranged and linked to the same expansion circuit 6701 and to the output leads in the same way shown in Figure 67.

[0491] Unlike in the first multiplier circuit embodiment, the treatment subcircuits 6705 each contains two of the subcircuit (SubcircuitVinSo08-3) shown in Figure 48. Each of the subcircuit shown in Figure 48 is for semi-optimising an 8-bit generalised number. Thus, two generalised numbers are fed to each treatment subcircuit 6705, which outputs the two generalised numbers, now semi-optimised, to the adder subcircuit 6707.

[0492] The adder subcircuit 6707 has the same structure as shown as the adder VinSoXY33 shown as Figure 58, and cannot be merely an AND gate as in the first embodiment. This is because there could be a carry-over from nonzeros in the same bit positions in both the substrings of the same sign. There is only no propagation since the nonzeros in both same sign substrings are interleaved with at least one zero, and every adder must consider the neighbouring bits to decide the value to output for the pertinent bit-position. The two partial products are reduced into one partial product on being added up in the adder. For completely, it may be noted that Transformation B in the first multiplier circuit embodiment has expressed any carryovers in advance of the adder

which allows the adder for generalised numbers treated by the AB method to be just an AND gate.

[0493] For two iterations, there are two sets of the treatment subcircuit and adder subcircuit 6707, just as described for the first embodiment.

[0494] A Third Embodiment Of A Multiplier Circuit (semi-optimisation method)

[0495] In a third embodiment, the multiplier is a generalised number that is semi-optimised before expansion with the multiplicand into partial products. This leads to a reduction in the overall number of partial products The generalization can done by the same 3 Steps of SEPy, rSEPx and SEPx, all arrays adjacent nonzeros in the same substring are 'spread out' apart.

[0496] Applying rSEP (both rSEPx and rSEPy) now to every next block of two adjacent bits of both substrings, i.e. a total of 4 bits, produces in the treated multiplier at least 3 zeros in the 4 bits of the two bit-positons. There are either just four zeros or three zeros with one nonzero. This allows the multiplicand to be replicated easily during expansion, each partial product being all zeros if there is no nonzero in the four bit, identical to multiplicand but position shifted if there is one nonzero, or identical and swapped signs and position shifted if the nonzero is in the y substring.

[0497] Therefore, the multiplicand and the treated multiplier are expanded by multiplying the x and y bits of the multiplicand in each bit-position by every consecutive block of two adjacent bit-positions of the multiplier, the output position-shifted accordingly. The number of partial products can be half the number obtained by expanding the multiplicand with every bit-position in the multiplier. Thus, a 4-bit multiplier expands into two partial products instead of four in the examples of the preceding paragraphs. This reduces the complexity of the multiplier circuit by half the number of the partial products.

[0498] The partial products still undergo semi-optimisation or the AB transformation using the steps and logic circuits described in the preceding paragraphs before being added up with the appropriate adder. Therefore, where the multiplier is a four bit generalised number, only one iteration is needed to arrive at the final sum.

[0499] In mathematical expression, this method for multiplying two semi-optimized binary numbers $(\hat{n}_w)_1$ and $(\hat{n}_w)_2$ comprises the steps of:

[0500] (a) providing two generalised numbers for the multiplicand and the multiplier $(n_w)_1$ and $(n_w)_2$.

[0501] (b) apply semi-optimization to $(n_w)_1$ and $(n_w)_2$ produce semi-optimized multiplicand $(\hat{n}_w)_1$ and semi-optimized multiplier $(\hat{n}_w)_2$, using the 3 steps described.

[0502] (c) apply concurrently rSEPx and rSEPy concurrently to reduce the semi-optimized multiplier $(\hat{n}_w)_2$ in blocks of consecutive two-bits.

$$(x_i')_2 = (\bar{\tilde{x}}_{i+1})_2(\bar{\tilde{y}}_{i+1})_2(\tilde{x}_i)_2 + (\tilde{x}_{i+1})_2(\tilde{y}_i)_2 \quad (28a)$$

$$(y_i')_2 = (\bar{\tilde{x}}_{i+1})_2(\bar{\tilde{y}}_{i+1})_2(\tilde{y}_i)_2 + (\tilde{y}_{i+1})_2(\tilde{x}_i)_2 \quad (28b)$$

$$(x_{i+1}')_2 = (\tilde{x}_{i+1})_2(\bar{\tilde{x}}_i)_2(\bar{\tilde{y}}_i)_2 \quad (28c)$$

$$(y_{i+1}')_2 = (\tilde{y}_{i+1})_2(\bar{\tilde{x}}_i)_2(\bar{\tilde{y}}_i)_2 \quad (28d)$$

[0503] where i=0, 2, 4, ..., such that there is at most one nonzero digit among $(x'_i)_2$, $(y'_i)_2$, $(x'_{i+1})_2$ and $(y'_{i+1})_2$, and the reduced multiplier $(n'_w)_2$ is represented by the resultant strings $(x'_i)_2$ and $(y'_i)_2$. It should be highlighted that the reduced multiplier $(n'_w)_2$ is no longer a semi-optimized binary number.

[0504] The Truth Table for rSEP (applying rSEPx and rSEPy simultaneously) is shown as Figure 70. The relevant circuit block diagram is shown in Figure 71 and the underling logic circuit is shown in Figure 72.

[0505] (d) determining the partial products of $(\hat{n}_w)_1$ and each consecutive two-digit pairs of the positive and negative strings of the reduced multiplier $(x'_{i+1})_2(x'_i)_2$ and $(y'_{i+1})_2(y'_i)_2$ for i = 0, 2, 4…to obtain partial products $(n_w)_{0\text{-}1}$, $(n_w)_{2\text{-}3}$, $(n_w)_{4\text{-}5}$, …;

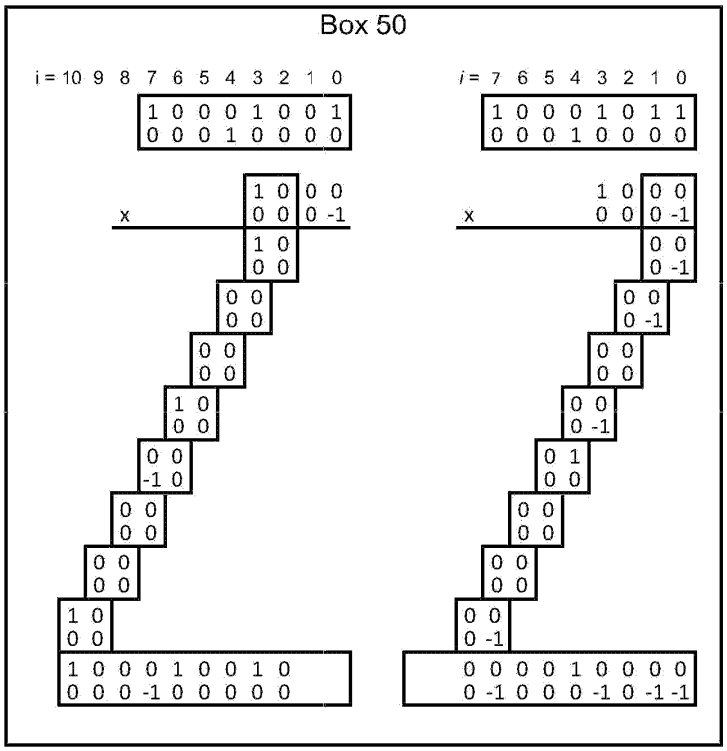
[0506] An example of rSEP on a multiplier $10\bar{1}1$ gives the reduced multiplier $100\bar{1}$ is shown below.

$i=$	3	2	1	0
	1	0	0	1
	0	0	-1	0

→

$i=$	3	2	1	0
	1	0	0	0
	0	0	0	-1

[0507] An example of expanding a multiplicand by every 2-bit blocks of the multiplier is shown in Box 50.



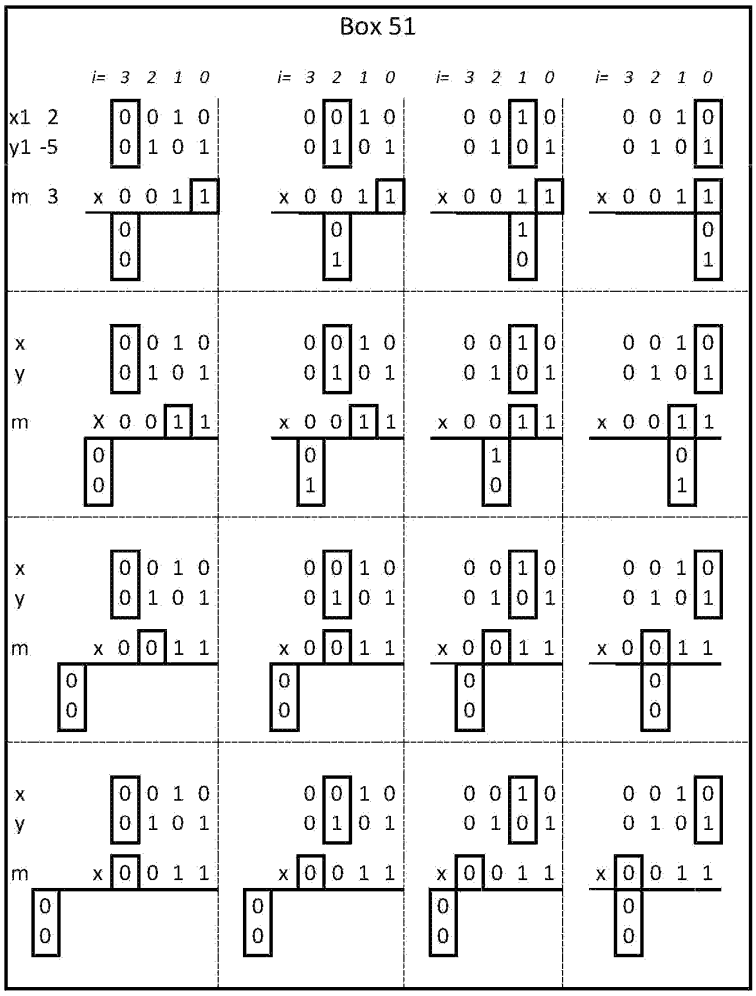
[0508] (e) summing up the above partial products ensues using described semi-optimisation and adder subcircuits with no propagation of carry-overs, or using the AB method and the appropriate adder, also with no propagation of carry-overs.

[0509] The treatment subcircuit 6705 and an adder subcircuit 6707 described for Figure 66 is used to semi-optimised and add up the two partial products.

[0510] A Fourth Embodiment Of A Multiplier Circuit (semi-optimisation method)

[0511] In a fourth embodiment, only one of the multiplicand or multiplier is a generalised number and the other is a regular, single-strand, binary string. Multiplication of a generalised number by a regular binary string produces partial products which are generalised numbers, which are added up using any of the preceding pre-treatment-and-addition methods.

[0512] An example is shown for -3 x 3 = -9. One generalised binary format of -3 has substrings x = 0010 and y = 0101, meaning 2-5. The multiplier, m = 0011 in regular binary format, which is 3 in decimal. Expanding the multiplicand and multiplier gives the partial products as shown in Box 51.



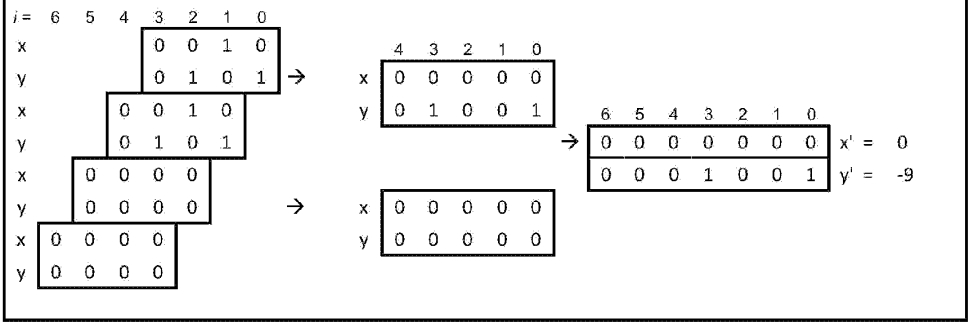
[0513] Box 52 shows the expansion of an x and y bit pair with a single bit as applied in Box 51. Not shown are all cases with m = 0 which gives zeros in both output x and y bits, and all cases with the input x and y bits that are both nonzeros, which mutually-cancel to zeros.

Box 52

x	0	x	m	-1	=	-1	x	-1	=	1
y	-1									0
x	1	x	m	-1	=	1	x	-1	=	-1
y	0									1
x	0	x	m	1	=	-1	x	1	=	-1
y	-1									1
x	1	x	m	1	=	1	x	1	=	1
y	0									0

[0514] The output value in every bit-position is shown concatenated into the respective partial product on the left in Box 53. There are eight substrings in the four partial products. These final sum of x and y substrings will have the combined value equal to the product of 3 x -3 = -9, expressed as x'= 0000 and y'= 1001. The treatment of and subsequent adding up of every two partial products may be done by any of the described methods. The output of the product is a generalised number with can be multiplied again with a binary number of the regular format in another multiplication.

Box 53



[0515] A multiplication circuit M84 for an 8-bit generalized binary number $(n_w)_1$ (multiplicand) and a 4-bit classical single string, binary number n_2 (multiplier) is shown as a circuit block in Figure 76 and corresponding Figure 77 showing sub-circuits for each major step in the multiplier circuit of Figure 76. With the exception of the expansion circuit 7701 and the expansion subcircuits 7701 therein, the multiplier circuit has the same structure as that shown in Figure 66, Figure 67, Figure 68 and Figure 69.

[0516] The expansion circuit 7701 requires only a number of input leads for the multiplier equal to the multiplier string length, shown on the rightmost. The expansion circuit 7701 distributes the input data to four expansion subcircuits 7701a, as illustrated in Figure 78. Each expansion subcircuit 7701a receives the entire multiplicand but

only one multiplier bit 7803 and outputs, at 7801, a pair of x and y bits which is the product of the multiplicand x and y bits in the same bit position and the multiplier bit. Each output is also position-shifted such that zeros are attributed to the positions shifted over. In this way, each of the four expansion subcircuits 7701a outputs a partial product, but position-shifted according to the order of the pertinent multiplier bit,

[0517] Figure 79 shows the internal details of an expansion subcircuit 7701a, which comprises eight units of a multiplication subcircuit MZX for the inputs of a single-bit and a single pair of x and y bits 7901. Figure 74 is the circuit block diagram of one such unit 7901. Figure 75 shows the logic circuit inside the unit, which operates according to the Truth Table in Figure 73 and to the following equations.

[0518] In mathematical expression, this method of multiplying a generalized binary number $(n_w)_1$ as multiplicand by a one-string classical binary number n_2 as multiplier, includes the steps of:

[0519] (a) expanding the multiplicand $(n_w)_1$ substrings $(x_i)_1$, $(y_i)_1$ for i=0, 1, 2, ...; by each multiplier n_2 bits $(x_j)_2$ for j=0, 1, 2, ..., to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ... for i=0, 1, 2, ... and j=0, 1, 2, ...; where

$(x'_i)_j = (x_i)_1(x_j)_2$ (29a)

$(y'_i)_j = (y_i)_1(x_j)_2$ (29b)

[0520] in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of (n'_w) 0; $(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of (n'_w) 1, etc., according to the Truth Table Figure 73, implemented in block circuit shown as Figure 74 . Figure 75 shows the sub-circuit in Figure 74 .

[0521] (b) summing the partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ...; to obtain the final product N_w is the same as in other embodiments.

[0522] A Fifth Embodiment Of A Multiplier Circuit (semi-optimisation method)

[0523] It is only partial products that have to be in the generalised number format in order to be treated by the transformations and added up without carry-over propagation. In less efficient embodiments, the partial products, multiplicand and multiplier are all in the regular format of a single-strand binary string.

[0524] Generalisation and pre-treatment of generalised numbers are performed only on any two such partial products to be added up. It is also possible that their sum in is de-converted into the regular format immediately, only to be re-generalized and treated when involved in another addition to another partial product or new partial product. Therefore, two numbers to be multiplied up can be supplied by a prior art processor to an external multiplication circuit that implements this embodiment, which returns their product as the regular binary format to the prior art processor.

[0525] Alternative Methods of the Semi-Optimization

[0526] The example below shows semi-optimisation of number 7 according to the formula (2n -n) introduced in a preceding paragraph, which in practice would be SEP applied to both x and y substrings concurrently or in immediate succession of either one to the other.

$$n = 7 = \begin{array}{r} 8 \\ -1 \end{array} \begin{array}{c} x \\ y \end{array} \boxed{\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{array}} \rightarrow 2n = 14 = 16-2 \boxed{\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{array}} \rightarrow \boxed{\begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & -1 & 0 \end{array}} \begin{array}{l} = 17 \\ = -10 \end{array} = 7$$

$$-n = -7 = -(8-1) \boxed{\begin{array}{ccccc} 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 0 \end{array}}$$

[0527] However, the example below shows a case that using the formula only does not prevent adjacent nonzeros in the output, which therefore fails to semi-optimise the input.

$$n = 11 \begin{array}{c} x \\ y \end{array} \boxed{\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{array}} \rightarrow 2n = 22 \boxed{\begin{array}{ccccc} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{array}} \rightarrow \boxed{\begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & -1 & 0 \end{array}} \begin{array}{l} = 17 \\ = -6 \end{array}$$

$$-n = -11 \boxed{\begin{array}{cccc} 0 & 0 & 0 & 1 \\ -1 & -1 & 0 & 0 \end{array}}$$

[0528] The problem arises when a generalised number comprises one of the following problematic arrangements of nonzero bits, $10\bar{1}$ or $\bar{1}01$, i.e.

$$\begin{array}{c} x \\ y \end{array} \boxed{\begin{array}{c|c|c} 1 & 0 & 0 \\ \hline 0 & 0 & -1 \end{array}}$$

$$\begin{array}{c} x \\ y \end{array} \boxed{\begin{array}{c|c|c} 0 & 0 & 1 \\ \hline -1 & 0 & 0 \end{array}}$$

[0529] The method of semi-optimisation in the 3 Steps described in the preceding paragraphs deals with these two arrangements by applying rSEPx in between applying SEP to the y substring and applying SEP to the x substring, without applying rSEPy at all. This provides an asymmetrical transformation producing staggered zero bits that can accommodate a carry-over from the immediately smaller bit-position, eliminating the need to propagate the carry-over to immediately higher bit. However, there are other methods of dealing with the two problematic arrangements of nonzero bits.

[0530] A first alternative method for semi-optimising a generalised number comprises the following steps:

[0531] (a) A generalized numbern_w is transformed according to the formula,

$$\tilde{N}_w^1 = 2n_w - n_w \qquad (30)$$

[0532] with the relevant Truth Table in Figure 80 by which a substring is subtracted from an instant of the same substring moved by one significant bit. The logic of Equation (30) can be represented by the following logic equation

$$\tilde{X}_i^1 = \bar{x}_i\bar{y}_i x_{i-1}\bar{y}_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1} + y_i\bar{x}_{i-1}x_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1}y_{i-2} + x_i y_{i-1}x_{i-2} \quad (31a)$$

$$\tilde{Y}_i^1 = \bar{x}_i\bar{y}_i y_{i-1}\bar{x}_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1} + x_i\bar{y}_{i-1}y_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1}x_{i-2} + y_i x_{i-1}y_{i-2} \quad (31b)$$

[0533] where $z_{n+1}=z_{-1}=z_{-2}=0$ are assumed.

[0534] However, there is a risk that the output $\tilde{N}_w^1$ is not semi-optimised if the input n_w contains any of the problematic arrangements $10\bar{1}$ and $\bar{1}01$.

[0535] (b) Certainty of a semi-optimized number $\tilde{N}_w^2$, or simply $\tilde{N}_w$ is then provided by pre-emptively eliminating any adjacent nonzeros 11 or $\bar{1}\,\bar{1}$ in $\tilde{N}_w^1$ according to the Truth Table 6c in Figure 81 and the following equations,

$$\tilde{X}_i^2 = x_i\bar{x}_{i-1}\bar{x}_{i-2}\bar{y}_{i-2} + y_i y_{i-1} + x_i x_{i-2} + \bar{x}_i\bar{y}_i x_{i-1}x_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1}y_{i-2} \quad (32a)$$

$$\tilde{Y}_i^2 = y_i\bar{y}_{i-1}\bar{x}_{i-2}\bar{y}_{i-2} + x_i x_{i-1} + y_i y_{i-2} + \bar{x}_i\bar{y}_i y_{i-1}y_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1}x_{i-2} \quad (32b)$$

[0536] Assuming $z_{n+1}=z_{-1}=z_{-2}=z_{-3}=0$, Equations (31a) , (31b) and Equations (32a) and (32b) can be combined and expressed as follows:

$$\begin{aligned}\tilde{X}_i = {} & \bar{x}_i\bar{y}_i x_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}\bar{y}_{i-3} + x_i y_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}x_{i-3} + y_i\bar{x}_{i-1}\bar{y}_{i-1}\bar{x}_{i-2} + y_i y_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}x_{i-3} \\ & + \bar{x}_i\bar{y}_i x_{i-1}x_{i-2} + x_i y_{i-1}x_{i-2} + y_i y_{i-1}x_{i-2} + x_i\bar{x}_{i-1}\bar{y}_{i-1}y_{i-2}\end{aligned} \quad (33a)$$

$$\begin{aligned}\tilde{Y}_i = {} & \bar{x}_i\bar{y}_i y_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}\bar{x}_{i-3} + y_i x_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}y_{i-3} + x_i\bar{x}_{i-1}\bar{y}_{i-1}\bar{y}_{i-2} + x_i x_{i-1}\bar{x}_{i-2}\bar{y}_{i-2}y_{i-3} \\ & + \bar{x}_i\bar{y}_i y_{i-1}y_{i-2} + y_i x_{i-1}y_{i-2} + x_i x_{i-1}y_{i-2} + y_i\bar{x}_{i-1}\bar{y}_{i-1}x_{i-2}\end{aligned} \quad (33b)$$

[0537] where positive and negative digits of n_w can be semi-optimized in parallel operations.

[0538] The semi-optimization outputs $\tilde{x}_i$ and $\tilde{y}_i$ are dependent only on Z_i, Z_{i-1}, Z_{i-2} and Z_{i-3} of n_w. Figure 82 shows Truth Table of steps a) and b) combined, i.e. Equation (33a) and (33b) . The corresponding logic circuit is presented in Figure 83 and Figure 84.

[0539] In a second alternative method of semi-optimising a generalised number

[0540] (a) any substrings of $10\bar{1}$ and $\bar{1}01$ is converted into $1\bar{1}1$ and $\bar{1}1\bar{1}$, respectively, in accordance to Equations (34a) and (34b) and Truth Table in Figure 87, and logic circuits in Figure 85 and Figure 86,

$$x_i' = \bar{y}_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}x_i + x_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}y_i + x_{i+1}x_i + y_{i+1}\bar{x}_i\bar{y}_i x_{i-1} + y_{i+1}x_i \quad (34a)$$

$$y_i' = \bar{x}_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}y_i + y_{i+2}\bar{x}_{i+1}\bar{y}_{i+1}x_i + y_{i+1}y_i + x_{i+1}\bar{x}_i\bar{y}_i y_{i-1} + x_{i+1}y_i \quad (34b)$$

[0541] (b) applying the formula in Equation (30) the following equations can be derived, with the relevant Truth Table in Figure 90, and the logic circuit in Figure 88 and Figure 89.

$$\tilde{X}_i = \bar{x}_i'\bar{y}_i'x_{i-1}'\bar{y}_{i-2}' + x_i'y_{i-1}'x_{i-2}' + y_i'\bar{x}_{i-1}'\bar{y}_{i-1}' + y_i'y_{i-1}'x_{i-2}' \quad (35a)$$

$$\tilde{Y}_i = \bar{x}_i'\bar{y}_i'y_{i-1}'\bar{x}_{i-2}' + y_i'x_{i-1}'y_{i-2}' + x_i'\bar{x}_{i-1}'\bar{y}_{i-1}' + x_i'x_{i-1}'y_{i-2}' \quad (35b)$$

[0542] which result in the semi-optimized digits $\widetilde{X}_i$ and $\widetilde{Y}_i$.

[0543] The first and second alternative methods differs in that they apply in opposite order bit manipulation algorithms and transformation of Equation (30) . Furthermore, the bit manipulation Equations (34a) and (34b) and bit manipulation Equations (32a) and (32b) remove the different non-semi-optimized arrangements of nonzeros between the alternative methods.

[0544] LimTree: A New Multiplication Algorithm

[0545] In some embodiments, partial products can be treated in a similar way as the Wallace Tree, that is, instead of adding specific rows of classical single string two's complement numbers iteratively, normally using the carry-save adders, to an accumulating sum, the circuit is designed to add up every two partial products, each partial product comprising a generalized binary number of paired strings, concurrently in any iteration. Note that the term partial product also means the sum of any two or more partial products that has to undergo at least a further iteration in process of producing the multiplication product.

[0546] As many iterations as required is designed into the multiplier circuit for adding up all the partial products using the described 4-2 general adder in a circuit, though similar but distinct in principle with that of a Wallace tree. This circuit applies a new logic algorithm termed the LimTree. Comparing to Wallace tree, considerable logic operations and propagation delay can be avoided using the LimTree.

[0547] Defining $\left\lfloor \frac{n}{m} \right\rfloor$ as the integer division function, Mod (n, m) as the modulus function for arguments n and m, k_0 as the original number of bit strings, then the number of bit strings k_i (i=0, 1, 2, …) at the i-th level for LimTree is given by

$$k_{i+1} = \left\lfloor \frac{k_i}{2} \right\rfloor + \mathrm{Mod}(k_i, 2) \qquad (36a)$$

[0548] where $k_i \geq 2$ and it requires $\left\lfloor \frac{k_i}{2} \right\rfloor$ number of 4-2 general adders; while in a Wallace Tree the number of bit strings at the i-th level is

$$k_{i+1} = 2\left\lfloor \frac{k_i}{3} \right\rfloor + \mathrm{Mod}(k_i, 3) \qquad (36b)$$

[0549] where $k_i \geq 3$ and it requires $\left\lfloor \frac{k_i}{3} \right\rfloor$ number of 3-2 adders.

[0550] For both LimTree and Wallace tree, a Carry-Look-ahead adder, or any other serial adder, is required to de-convert a pair of x and y substrings (or a pair of two's complement numbers in the case of Wallace tree) into a classical single binary string representing the sum of the partial products. However, LimTree has a significant edge over Wallace tree in the following aspects.

[0551] · A level of 4-2 adder used in LimTree has considerable fewer gate-level operations than two levels of 3-2 adder (full adder) used in Wallace tree;

[0552] · Significant improvement in propagation delay because of significant fewer gate levels and number of iterations;

[0553] · LimTree logic algorithm is far more structured in terms of logic operations;

[0554] For multiplication of two 64-bit binary strings of the regular single-strand format (not generalized numbers each comprising with a pair of x and y substrings) , applying Eq. (36a) for LimTree yields

$$k_0=32,\ k_1=\lfloor 32/2\rfloor+\mathrm{Mod}(32,2)=16,\ k_2=\lfloor 16/2\rfloor+\mathrm{Mod}(16,2)=8,$$
$$k_3=\lfloor 8/2\rfloor+\mathrm{Mod}(8,2)=4,\ k_4=\lfloor 4/2\rfloor+\mathrm{Mod}(4,2)=2,\ k_5=\lfloor 2/2\rfloor+\mathrm{Mod}(2,2)=1 \quad (37a)$$

[0555] where five levels of 4-2 logic operations plus one level of carry-Look-ahead adder are required.

[0556] Note that the product is the sum of 64 single-strand binary strings of partial products, from expanding the 64 bit multiplicand, weight-shifted and multiplied by each bit in the 64-bit multiplier. It can be transformed into 32 pairs of 64-bit x and y substrings each being a 64-bit generalized number. Hence, k_0=32. For k_3=1 in the final level, it is a generalized number as one pair of x and y substrings. On the other hand, applying Eq. (36b) for the multiplication of two 64-bit classical binary numbers using Wallace tree yields

$$k_0=64,\ k_1=2\lfloor 64/3\rfloor+\mathrm{Mod}(64,3)=43,\ k_2=2\lfloor 43/3\rfloor+\mathrm{Mod}(43,3)=29,$$
$$k_3=2\lfloor 29/3\rfloor+\mathrm{Mod}(29,3)=20,\ k_4=2\lfloor 20/3\rfloor+\mathrm{Mod}(20,3)=14,$$
$$k_5=2\lfloor 14/3\rfloor+\mathrm{Mod}(14,3)=10,\ k_6=2\lfloor 10/3\rfloor+\mathrm{Mod}(10,3)=7,\ k_7=2\lfloor 7/3\rfloor+\mathrm{Mod}(7,3)=5, \quad (37b)$$
$$k_8=2\lfloor 5/3\rfloor+\mathrm{Mod}(5,3)=4,\ k_9=2\lfloor 4/3\rfloor+\mathrm{Mod}(4,3)=3,\ k_{10}=2\lfloor 3/3\rfloor+\mathrm{Mod}(3,3)=2$$

[0557] where ten levels of 3-2 logic operations plus one level of Carry-Look-ahead adder are required.

[0558] An example of using LimTree for multiplication of two generalized binary numbers is presented. An 8-bit 8X8 multiplication algorithm is illustrated as follows where Figure 91 (Table (a) shows the standard LimTree while Table (b) shows the inverted LimTree. An inverted LimTree is just a graphical re-arrangement of the partial products into a pyramidal form, and this re-arrangement was used to let circuit designers follow the progress of their designs easily by marking the bit-position that has the most rows of practical products. It is not necessary to use the inversion if the string length is short.

[0559] All methods disclosed in this document may be implemented using logical circuits.

[0560] The logical circuits described in this document (aterm 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 one or more integrated circuits, or as respective physical components.

[0561] 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.

[0562] 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.

[0563] 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.

[0564] 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.

[0565] 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.

Claims

[Claim 1] A method implemented using fixed logic circuitry of replacement of a nonzero in a bit-position among a plurality of bit-positions in two of four binary strings comprised in two generalised numbers to be added up;
the replacement comprising the steps of
replacing one or more nonzeros in bit-positions that includes the bit-position with
a zero in the bit-position and one or more nonzeros in the other bit-positions among the plurality of bit-positions in two of four binary strings representing the same value.

[Claim 2] A method of multiplying a multiplicand by a multiplier, comprising the steps of:
2a) expanding the multiplicand and the multiplier to obtain partial products in the form of generalised numbers;
2b) operating a plurality of addition operations to add up the partial products and produce a multiplication product; wherein
each addition operation is preceded with the steps as claimed in claim 1.

[Claim 3] A method of multiplying a multiplicand by a multiplier as claimed in claim 2, wherein
the multiplicand in step 2a) is a generalized number represented by two binary strings and the multiplier is a number represented by at least one binary string.

[Claim 4] A method of multiplying a multiplicand by a multiplier as claimed in claim 3, wherein the multiplier in step 2a) is a generalised number.

[Claim 5] A method of multiplying a multiplicand by a multiplier as claimed in claim 4, comprising the following steps preceding step 2a) :
replacing nonzeros in a block of four bits comprising two bits in the same adjacent positions in each binary string and representing a value with a block of four bits in the same positions having at least three zeros and a nonzero representing the same value.

[Claim 6] A method of multiplying a multiplicand by a multiplier as claimed in claim 5, wherein step 2a) comprises expanding the multiplicand by each consecutive block in the multiplier.

[Claim 7] A method of multiplying a multiplicand by a multiplier as claimed in claim 3, the multiplicand being a generalized number $(n_w)_1$ having substrings x_1 and y_1; and

the multiplier being at least one binary string x_2 providing a number n_2; wherein step 2a) comprises the steps of:

7a) expanding the multiplicand $(n_w)_1$ by each bit $(x_j)_2$ in the multiplier n_2 for j=0, 1, 2…, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ... ; wherein

$(x'_i)_j = (x_i)_1(x_j)_2$ (1a)

$(y'_i)_j = (y_i)_1(x_j)_2$ (1b)

according to the Truth Table below for i=0, 1, 2, ... and j=0, 1, 2, ... :

Input		Output
$(z_i)_1$	$(x_j)_2$	$(z'_j)_2$
0	0	0
1	0	0
$\bar{1}$	0	0
0	1	0
1	1	1
$\bar{1}$	1	$\bar{1}$

in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$; $(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of $(n'_w)_1$, and so on;

7b) for each iteration including pre-treating and adding up at least two partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, repeating a number of iteration until all the partial products are summed up into a final product N_w; wherein

the pre-treating including a step of

7c) 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 same or the other one of the at least two partial products; such that

the bit-position i is able to store a carry-over during the adding up.

[Claim 8] A method of multiplying a multiplicand by a multiplier as claimed in claim 4,

the multiplier comprises a further binary string y_2 being a generalised number $(\hat{n}_w)_2$; wherein

step 2a) comprises:

expanding the multiplicand $(n_w)_1$ by each pair of bits $(x_j)_2$, $(y_j)_2$ of the multiplier $(\hat{n}_w)_2$ for j =0, 1, 2, …, to obtain partial products $(n'_w)_0$, $(n'_w)_1$, $(n'_w)_2$, ... ; where

$(x'_i)_j = (x_i)_1(x_j)_2 + (y_i)_1 (y_j)_2$ (2a)

$(y'_i)_j = (y_i)_1 (x_j)_2 + (x_i)_1 (y_j)_2$ (2b)

according to the Truth Table below for i=0, 1, 2, ... and j=0, 1, 2, ... :

Input		Output
$(z_i)_1$	$(z_j)_2$	$(z'_i)_j$
0	0	0
1	0	0
$\overline{1}$	0	0
0	1	0
1	1	1
$\overline{1}$	1	$\overline{1}$
0	$\overline{1}$	0
1	$\overline{1}$	$\overline{1}$
$\overline{1}$	$\overline{1}$	1

in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$; $(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of $(n'_w)_1$, and so on.

[Claim 9] A method of multiplying a multiplicand by a multiplier as claimed in claim 7; wherein

step 7c) comprises the steps of:

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 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	$\overline{1}$	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	

i. adding up x and y to produce a single binary string having the value of the sum of the two numbers.

[Claim 10] A method of multiplying a multiplicand by a multiplier as claimed in claim 7, wherein

step 7c) comprises the steps of:

re-arranging the zeros and nonzeros in the binary substrings comprises the following step 10-1) , step 10-2) and step 10-3) in the specific order of:

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

$$x'_i = x_i \overline{y}_{i-1} + y_i \overline{y}_{i-1} \quad (3a)$$

and

$$y'_i = \overline{x}_i \overline{y}_i y_{i-1} \quad (3b)$$

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:

$$x_i''= x_i'\bar{y}_{i-1}' + x_{i+1}'y_i' \quad (4a)$$

$$y_i''= \bar{x}_{i+1}'y_i' \quad (4b)$$

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

$$\tilde{X}_i = \bar{x}_i''\bar{y}_i''x_{i-1}'' \quad (5a)$$

$$\tilde{Y}_i = x_i''\bar{x}_{i-1}''\bar{y}_{i-1}'' + y_i''\bar{x}_{i-1}''\bar{y}_{i-1}'' \text{ or simply } \tilde{Y}_i = x_i''\bar{x}_{i-1}'' + y_i''\bar{x}_{i-1}'' \quad (5b)$$

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

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

$$x_i' = (\tilde{x}_i)_1(\bar{\tilde{x}}_i)_2 + (\bar{\tilde{x}}_i)_1(\tilde{x}_i)_2 + (\tilde{x}_{i-1})_1(\tilde{x}_{i-1})_2 \quad (6a)$$

$$y_i' = (\tilde{y}_i)_1(\bar{\tilde{y}}_i)_2 + (\bar{\tilde{y}}_i)_1(\tilde{y}_i)_2 + (\tilde{y}_{i-1})_1(\tilde{y}_{i-1})_2 \quad (6b)$$

[Claim 11] A method of multiplying a multiplicand by a multiplier as claimed in claim 7, comprising the following steps preceding step 7a) :

treating the multiplier $(n_w)_2$ as follows:

i) applying the following equations to eliminate any immediately neighbouring nonzero bits in the binary substring y

$$x_i' = x_i\bar{y}_{i-1} + y_i\bar{y}_{i-1}$$

and $y_i' = \bar{x}_i\bar{y}_i y_{i-1}$

similar to Equations (3a) and (3b) above.

ii) 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:

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

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

similar to Equations (4a) and (4b) above.

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

$$\tilde{X}_i = \bar{x}_i'' \bar{y}_i'' x_{i-1}''$$

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

similar to Equations (5a) and (5b) above, where $\tilde{X}_i$ and $\tilde{Y}_i$ denotes the value of the bits in position i in the formatted pair of binary substrings, $\tilde{X}$ and

$\tilde{Y}$;

iv) reducing the semi-optimized multiplier $(\hat{n}_w)_2$ in blocks of consecutive two-bits according to the equations

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

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

$$(x_{i+1}')_2 = (\tilde{x}_{i+1})_2(\bar{\tilde{x}}_i)_2(\bar{\tilde{y}}_i)_2 \quad (7c)$$

$$(y_{i+1}')_2 = (\tilde{y}_{i+1})_2(\bar{\tilde{x}}_i)_2(\bar{\tilde{y}}_i)_2 \quad (7d)$$

where i = 0, 2, 4, …, such that there is at most one nonzero digit $(x'_i)_2$, $(y'_i)_2$, $(x'_{i+1})_2$, $(y'_{i+1})_2$ and the reduced multiplier $(n'_w)_2$ is represented by the resultant substrings $(x'_i)_2$ and $(y'_i)_2$.

v) determining the partial products of $(\hat{n}_w)_1$ each consecutive two-digit pairs of the positive and negative strings of the reduced multiplier $(x'_{i+1})_2$ $(x'_i)_2$ and $(y'_{i+1})_2$ $(y'_i)_2$ for i = 0, 2, 4… to obtain partial products $(n_w)_{0\text{-}1}$, $(n_w)_{2\text{-}3}$, $(n_w)_{4\text{-}5}$, …; and

such that step (a) becomes:

expanding the multiplicand $(n_w)_1$ by each pair of bits $(x_j)_2$, $(y_j)_2$ of the multiplier $(\hat{n}_w)_2$ for j =0, 1, 2, …, to obtain partial products, $(n'_w)_1$, $(n'_w)_2$, ...; where

$(x'_i)_j = (x_i)_1 (x_j)_2 + (y_i)_1 (y_j)_2$ (8a)

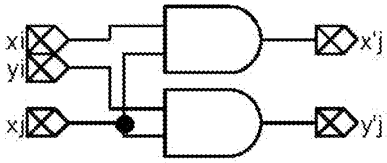
$(y'_i)_j = (y_i)_1 (x_j)_2 + (x_i)_1 (y_j)_2$ (8b)

according to the Truth Table below for i=0, 1, 2, ... and j=0, 1, 2, ...:

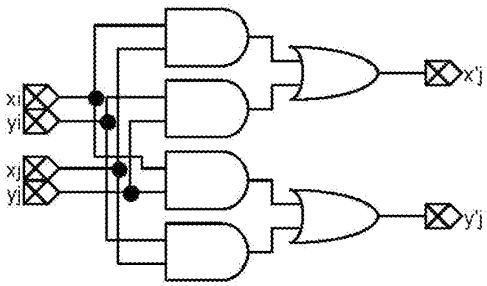
Input		Output
$(z_i)_1$	$(z_j)_2$	$(z'_i)_j$
0	0	0
1	0	0
$\overline{1}$	0	0
0	1	0
1	1	1
$\overline{1}$	1	$\overline{1}$
0	$\overline{1}$	0
1	$\overline{1}$	$\overline{1}$
$\overline{1}$	$\overline{1}$	1

in which $(x'_i)_0$, $(y'_i)_0$ are the bit-pairs of $(n'_w)_0$; $(x'_i)_1$, $(y'_i)_1$ are the bit-pairs of $(n'_w)_1$, and so on.

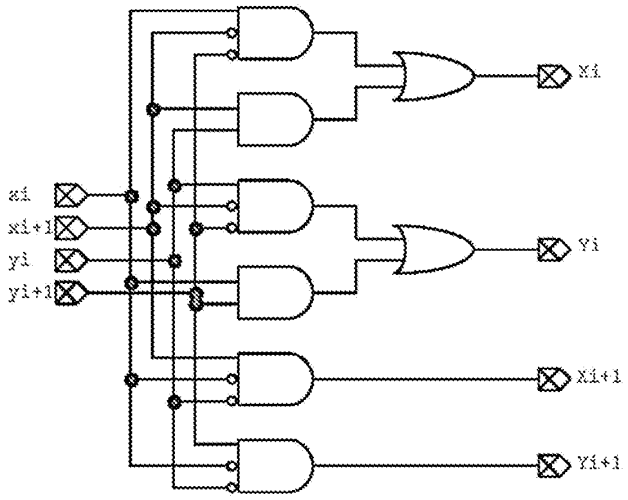
[Claim 12] A unit multiplier, , provided as fixed logic circuitry, of an expansion circuit for obtaining partial products between a generalised number and a binary string according to step 2a) of claim 3, comprising:



[Claim 13] A unit multiplier, provided as fixed logic circuitry, of an expansion circuit for obtaining partial products between two generalised numbers according to step 2a) of claim 4, comprising:



[Claim 14] A unit multiplier, provided as fixed logic circuitry, of an expansion circuit for obtaining partial products between two generalised numbers according to step 2a) of claim 6, comprising:



[Claim 15] An apparatus comprising fixed logical circuitry configured to perform a method as claimed in any of claims 1-11.