(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

CORRECTED VERSION

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
2 April 2009 (02.04.2009)

PCT

(10) International Publication Number
WO 2009/042106 A9

(51) International Patent Classification:
G06F 7/50 (2006.01)

(21) International Application Number:
PCT/US2008/010999

(22) International Filing Date:
23 September 2008 (23.09.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/974,820 24 September 2007 (24.09.2007) US
12/148,505 18 April 2008 (18.04.2008) US

(71) Applicant *(for all designated States except US)*: **VNS PORTFOLIO LLC** [US/US]; 20400 Stevens Creek Blvd., Fifth Floor, Cupertino, CA 95014 (US).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: **MOORE, Charles, H.** [US/US]; 110 Green Rd., P.O. Box 127, Sierra City, CA 96125 (US).

(74) Agent: **HENNEMAN, Larry, E., Jr.**; Henneman & Associates, PLC, 714 W. Michigan Ave., Three Rivers, MI 49093 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

[Continued on next page]

(54) Title: SHIFT-ADD MECHANISM

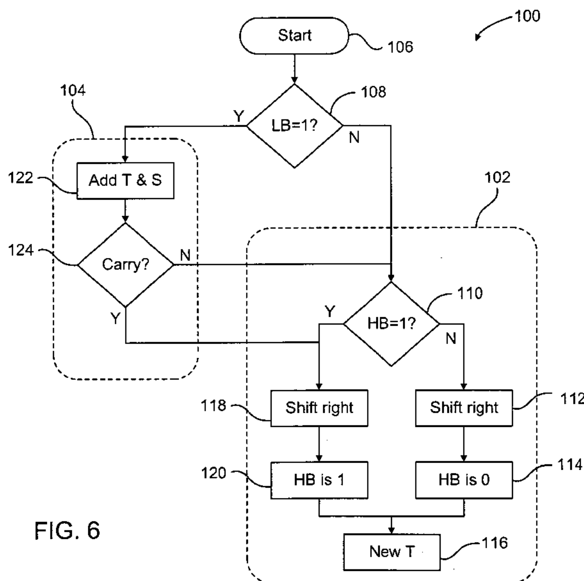

FIG. 6

(57) Abstract: A method to perform a shift-add operation on two values loaded in two memories of a processor where the first memory has a low bit (LB) and a high bit (HB). If the LB is zero, then this is case (1), if the HB is also zero, shifting the first value lower one bit-position and setting the HB to zero, thereby arriving at a new value in the first memory, and alternately if the HB is one, then this is case (2), and proceed shifting the first value lower one bit-position and setting the HB to one, thereby arriving at the new value. However, if the LB is one, then adding the second value to the first value in the first memory and if this does not produce a carry, proceeding as if at case (1) and otherwise proceeding as if at case (2).

WO 2009/042106 A9

WO 2009/042106 A9

(48) Date of publication of this corrected version:
9 July 2009

(15) Information about Correction:
see Notice of 9 July 2009

SHIFT-ADD MECHANISM

5 Inventors: Charles H. Moore

BACKGROUND OF THE INVENTION

10

TECHNICAL FIELD

The present invention relates generally to electrical computers and digital processing systems having processing architectures and performing instruction processing, and more particularly to processes that can be implemented as operational codes in such.

15

BACKGROUND ART

Powerful and efficient operational codes (op-codes) are critical for modern computer processors to perform many tasks. For example, some such tasks are multiplication and producing sequences of pseudorandom numbers.

20

BRIEF SUMMARY OF THE INVENTION

Accordingly, it is an object of the present invention to provide shift-add mechanism that is useful for various operations in a processor.

25 Briefly, one preferred embodiment of the present invention is a method for performing a shift-add operation in a processor having multi-bit registers respectively holding two values. The memories have bit-positions that can all be zero or one and the first memory has low bit (LB) and high bit (HB) bit-positions. If the first memory LB is zero, we have a case (1), and then, if the first memory HB is also zero, the first value is shifted one bit-30 position lower and the first memory HB is set to zero, thus arriving at a new value in the first memory. Alternately, if the first memory HB is one, we have a case (2) and then the first value is shifted one bit-position lower and the first memory HB is set to one, thus alternately

arriving at the new value. However, if the first memory LB was one, then the second value is added to the first value in the first memory. If this does not produce a carry, the method proceeds as if at case (1). Alternately, if this did produce a carry, the method proceeds as if at case (2).

5 These and other objects and advantages of the present invention will become clear to those skilled in the art in view of the description of the best presently known mode of carrying out the invention and the industrial applicability of the preferred embodiment as described herein and as illustrated in the figures of the drawings.

10

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWING(S)

The purposes and advantages of the present invention will be apparent from the following detailed description in conjunction with the appended tables and figures of drawings in which:

15 TBLS. 1-4 represent the values in the T-register and the S-register in a SEAforth™ 24a device in a set of hypothetical +* (shift-add mechanism) examples.

TBLS. 5-10 represent the values in the T-register and the S-register in a SEAforth™ 24a device in a set of hypothetical +* (shift-add mechanism) multiplication examples.

FIG. 1 (background art) is a table of the thirty two operational codes (op-codes) in the

20 Venture Forth™ programming language.

FIG. 2 (background art) is a block diagram showing the general architecture of each of the cores in a SEAforth™ 24a device.

FIGS. 3a-b (background art) are schematic block diagrams depicting how the 18 bit wide registers in the SEAforth™ 24a can be represented, wherein FIG. 3a shows the actual

25 bit arrangement and FIG. 3b shows a conceptual bit arrangement.

FIGS. 4a-b (background art) are schematic block diagrams depicting register content, wherein FIG. 4a shows the slots filled with four • (nop) op-codes and FIG. 4b shows the register filled with the number 236775 (as unsigned binary).

FIGS. 5a-b (background art) are block diagrams respectively and stylistically showing

30 the return and the data stack elements in SEAforth™ 24a cores, wherein FIG. 5a depicts elements in the return stack region and FIG. 5b depicts elements in the data stack region.

FIG. 6 is a flow chart of the inventive shift-add mechanism that shows all of the possible actions associated with a single execution of the +* op-code.

FIG. 7 is a table showing bit relationships in accord with FIG. 6.

FIG. 8 is a flow chart of a shift-add based multiplication process using the present invention.

In the various figures of the drawings, like references are used to denote like or similar elements or steps.

DETAILED DESCRIPTION OF THE INVENTION

A preferred embodiment of the present invention is a shift-add mechanism. As illustrated in the various drawings herein, and particularly in the view of FIG. 6, preferred embodiments of the invention are depicted by the general reference character 100.

THE +* OP-CODE ON THE SEAFORTH™ 24A DEVICE

The inventive shift-add mechanism **100** can be used for a variety of tasks including, without limitation, multiplication and pseudorandom number generation. In the Venture Forth™ programming language, the present inventor employs the shift-add mechanism **100** as a "+*" op-code. Before presenting more detailed examples, it is useful to consider a simple example in the context of a SEAforth™ 24a device by IntellaSys™ Corporation of Cupertino, California, a member of The TPL Group™ of companies.

As general background, the SEAforth™ 24a has 24 stack based microprocessor cores that all use the Venture Forth™ programming language. FIG. 1 (background art) is a table of the thirty two operational codes (op-codes) in this language, in hex, mnemonic, and binary representations. These op-codes are divided into two main categories, memory instructions and arithmetic logic unit (ALU) instructions, with sixteen op-codes in each division. The memory instructions are shown in the left half of the table in FIG. 1, and the ALU instructions are shown in the right half of the table in FIG. 1. It can be appreciated that one clear distinction between the divisions of op-codes is that the memory instructions contain a zero (0) in the left-most bit, whereas the ALU instructions contain a one (1) in the left-most bit. Furthermore, this is the case regardless of whether the op-codes are viewed in their hex or binary representations. The +* op-code of present interest is shown upper-most in the right-hand column.

FIG. 2 (background art) is a block diagram showing the general architecture of each of the cores in the SEAforth™ 24a device. All of the registers in the SEAforth™ 24a are 18

bits wide, except for the B- and PC-registers, which are not relevant here.

There are two distinct approaches that can be taken when a programmer is selecting the bits that will make up the 18 bit wide register space in a SEAforth™ 24a (with limited exceptions for some op-codes that use the A-register). The first of these is to divide this
5 space into four equal slots that can be called: slot 0, slot 1, slot 2, and slot 3. The bit lengths of these slots are not all equal, however, because division of 18 by 4 results in a remainder. The first three slots, slot 0, slot 1, and slot 2, therefore, can each hold 5 bits, while slot 3 holds only three bits.

FIGS. 3a-b (background art) are schematic block diagrams depicting how the 18 bit
10 wide registers in the SEAforth™ 24a device can be represented, wherein FIG. 3a shows the actual arrangement of the bits as bits 0 through 17, and FIG. 3b shows a conceptual arrangement of the bits as bits -2 through 17. In FIG. 3a it can be seen that bits 13-17 inclusive make up slot 0, bits 8-12 inclusive make up slot 1, bits 3-7 inclusive make up slot 2, and bits 0-2 make up slot 3. The designers of the SEAforth™ 24a device often point out the
15 fact that the 18-bit wide registers can each contain three and three/five instructions, and this prompts the question whether slot 3 is significant, since none of the op-codes in FIG. 1 would appear to appear to fit in slot 3. FIG. 3b shows how the designers of the SEAforth™ 24a device have handled this. They allow only certain op-codes to fit into slot 3 by treating the two least significant bits, called bit -1 and bit -2 here, as being hard wired to ground or zero.
20 Of course, since slot 3 effectively has only three bits rather than five bits of space, the number of op-codes that fit into slot 3 is limited to only eight of the 32 possible op-codes. Specifically, these op codes are:

$00	00000b	; (return)
$04	00100b	unext
$08	01000b	@p+
$0C	01100b	!p+
$10	10000b	+*
$14	10100b	+
$18	11000b	dup
$1C	11100b	• (nop).

25 The second approach that a programmer can use when selecting the bits that will

make up the 18-bit wide register space in the SEAforth™ 24a is to simply not divide the 18-bit wide register into slots, and to instead consider the register as containing a single 18-bit binary value. This may appear at first to be a completely different approach than the slot-based approach, but both representations are actually equivalent. FIGS. 4a-b (background
5 art) are schematic block diagrams depicting an example illustrating this. FIG. 4a shows the slots filled with four • *(nop)* op-codes, and FIG. 4b shows the register filled with the number 236775 (as unsigned binary). With reference to FIG. 1, it can be appreciated that the binary bit values in FIGS. 4a-b are the very same. This means that it is been left up to the programmer to differentiate whether a register will contain a number or contain four op-
10 codes.

FIGS. 5a-b (background art) are block diagrams stylistically showing the return and the data stack elements, respectively, that exist in each core of a SEAforth™ 24a device. FIG. 5a depicts how the return stack region includes a top register that is referred to as "R" (or as the R-register) and an eight-register circular buffer. FIG. 5b depicts how the data stack
15 region includes a top register that is referred to as "T" (or as the T-register), a (second) register below T that is referred to as "S" (or as the S-register), and also an eight-register circular buffer. In total, the return stack thus contains nine registers and the data stack contains ten registers. Only the data stack region needs to be considered in the following example.

20 TBLS. 1-4 represent the values in the T-register and the S-register in a set of hypothetical +* examples. For simplicity, only 4-bit field widths are shown. It is important to note in the following that the value in the T-register (T) is changed while the value in the S-register (S) remains unchanged during execution of the +* op-code. [N.b., to avoid confusion between the bits making up values and the locations in memory that may hold
25 such, we herein refer to bits in values and to bit-positions in memory. It then follows that a value has a most significant bit (MSB) and a least significant bit (LSB), and that a location in memory has a high bit (HB) position and a low bit (LB) position.]

TBL. 1 shows the value one (1) initially placed in the T-register and the value three (3) placed in the S-register. Because the low bit (LB) position of T here is a 1, during
30 execution of the +* op-code:

(1) S and T are added together and the result is put in T (TBL. 2 shows the result of this); and

(2) the contents of T are shifted to the right and a 0 is placed in bit 4 (TBL.

3 shows the result of this).

The reason for bit 4 being filled with a 0 is saved for later discussion.

The contents of T and S in TBL. 3 are now used for a second example. Because the LB position of T is now a 0, during another execution of the +* op-code:

5 (1) the contents of T are simply shifted to the right and a 0 is placed in bit 4 (TBL. 4 shows the result of this).

Again, the reason for bit 4 being filled with a 0 is saved for later discussion. Additionally, it should be noted that the shift to the right of all of the bits in T is not associated in any way with the fact that a 1 or 0 filled the LB position of T prior to the

10 execution of the +* op-code. Instead, and more importantly, the shift of all the bits to the right in T is associated with the +* op-code itself.

These two examples demonstrate nearly all of the actions associated with the +* op-code. What was not fully described was why 0 is used to fill bit 4. The following covers this.

15

THE GENERAL CASE OF THE +* OP-CODE

A general explanation of the +* op-code is that it executes a conditional add followed by a bit shift of all bits in T in the direction of the low order bits when either a 1 or a 0 fills the high bit (HB) position of T after the shift.

20 FIG. 6 is a block diagram of the inventive shift-add mechanism **100** that shows all of the possible actions associated with a single execution of the +* op-code. The +* op-code has two major sub-processes, a shift sub-process **102** and a conditional add sub-process **104**. The shift-add mechanism **100** is embodied as a +* op-code that starts in a step **106** and where the content of the LB position of T is examined in a step **108**.

25 Turning first to the shift sub-process **102**, when the LSB of T is 0, in a step **110** the content of the HB position of T is examined. When the HB position of T is 0, in a step **112** the contents of T are shifted right, in a step **114** the HB position of T is filled with a 0, and in a step **116** T contains its new value. Alternately, when the HB position of T is 1, in a step **118** the contents of T are shifted right, in a step **120** the HB position of T is filled with a 1,

30 and step **116** now follows where T now contains its new value.

Turning now to the conditional add sub-process **104**, when the LB position of T is 1, in a step **122** the contents of T and S are added and in a step **124** whether this produces a carry is determined. If there was no carry, the shift sub-process **102** is entered at step **110**, as

shown. Alternately, if there was a carry (the carry bit is 1), the shift sub-process **102** is entered at step **118**, as shown. And then the +* op-code process (the shift-add mechanism **100**) continues with the shift sub-process **102** through step **116**, where T will now contain a new value.

5 While the actions associated with the +* op-code are easy to define, FIG. 6 reveals that the execution of the +* op-code is not conceptually simple. FIG. 7 is a table showing the relationships between the LB position and the HB position of T prior to an execution, here called old T, an intermediate carry when the values in S and T are added (if this action occurs), and finally the HB and the penultimate bit (HB – 1) of T which is produced after
10 execution, here called new T.

A +* PSEUDO-CODE ALGORITHM

The most general case of a +* op-code is now described using a pseudo-code algorithm. For this description, it is assumed that the +* op-code is executed on an n-bit
15 machine wherein an n_t-bit width number t is initially placed in T and an n_s-bit width number s is initially placed in S. Furthermore, it is assumed that only one additional bit is available to represent a carry, even if the +* op-code produces a carry that is theoretically more than one bit can represent. There is no restriction on the lengths of n_t and n_s, only that their individual bit lengths should be less than or equal to the bit width of n. The pseudo-code is as follows:

20 1. If the LB position of T is a 1:

1a. Add the value t in T to the value s in S where the sum of $t+s$, call this t', replaces the present t in T and S are left unchanged.

1a1. If the HB position of T is a 1:

1a1a. If the addition of t and s resulted in a carry:

25 1a1a1. Shift all bits in T to the right one bit. Bit 0 of t' after the shift contains the contents of bit 1 before the shift. Bit 1 of t' after the shift contains the contents of bit 2 before the shift. In the same way, the rest of t' is filled where bit m,
30 $m < n$, {#1} being filled after the shift contains the contents of bit $m + 1$ before the shift. This process leaves bit n devoid while effectively destroying bit 0 of t' before the shift. Bit n {#1}

of t' after the shift will be filled with a 1.

1a1b. If the addition of t and s did not result in a carry:

1a1b1. Shift all bits in T to the right one bit. Bit 0 of t' after the shift contains the contents of bit 1 before the shift. Bit 1 of t' after the shift contains the contents of bit 2 before the shift. In the same way, the rest of t' is filled where bit m, $m < n$, {#1} being filled after the shift contains the contents of bit $m + 1$ before the shift. This process leaves bit n devoid while effectively destroying bit 0 of t' before the shift. Bit n {#1} of t' after the shift will be filled with a 1.

1a2. If the HB position of T is a 0:

1a2a. If the addition of t and s resulted in a carry:

1a2a1. Shift all bits in T to the right one bit. Bit 0 of t' after the shift contains the contents of bit 1 before the shift. Bit 1 of t' after the shift contains the contents of bit 2 before the shift. In the same way, the rest of t' is filled where bit m, $m < n$, {#1} being filled after the shift contains the contents of bit $m + 1$ before the shift. This process leaves bit n devoid while effectively destroying bit 0 of t' before the shift. Bit n {#1} of t' after the shift will be filled with a 1.

1a2b. If the addition of t and s did not result in a carry:

1a2b1. Shift all bits in t to the right one bit. Bit 0 of t' after the shift contains the contents of bit 1 before the shift. Bit 1 of t' after the shift contains the contents of bit 2 before the shift. In the same way, the rest of t' is filled where bit m, $m < n$, {#1} being filled after the shift contains the contents of bit $m + 1$ before the shift. This process leaves bit n devoid while effectively

destroying bit 0 of t' before the shift. Bit n {#1} of t' after the shift will be filled with a 0.

2. If the LB position of T is a 0:

2a. If the HB position of T is a 1:

5 2a1. Shift all bits in T to the right one bit. Bit 0 of t' after the shift contains the contents of bit 1 before the shift. Bit 1 of t' after the shift contains the contents of bit 2 before the shift. In the same way, the rest of t' is filled where bit m, $m < n$, {#1} being filled after the shift contains the contents of bit $m + 1$ before

10 the shift. This process leaves bit n devoid while effectively destroying bit 0 of t' before the shift. Bit n {#1} of t' after the shift will be filled with a 1.

2b. If the HB position of T is a 0:

2b1. Shift all bits in T to the right one bit. Bit 0 of t' after the shift

15 contains the contents of bit 1 before the shift. Bit 1 of t' after the shift contains the contents of bit 2 before the shift. In the same way, the rest of t' is filled where bit m, $m < n$, {#1} being filled after the shift contains the contents of bit $m + 1$ before the shift. This process leaves bit n devoid while effectively

20 destroying bit 0 of t' before the shift. Bit n {#1} of t' after the shift will be filled with a 0.

It is important to note in the preceding that the +* op-code always involves a bit shift to the right (in the direction of the low order bits) of all bits in T. This bit shift is not the result of any event before, during, or after the execution of the +* op-code. +The bit shift is

25 an always executed event associated with the +* op-code.

MULTIPLICATION UTILIZING THE +* OP-CODE

It has been implied herein that the inventive shift-add mechanism **100** can be used for multiplication. An example is now presented followed by an explanation of the general case

30 of utilizing the +* op-code to execute complete and correct multiplication.

Let us suppose that a person would like to multiply the numbers nine (9) and seven (7) and that the letter T is used to represent an 8-bit memory location where the nine is initially placed and S is used to represent an 8-bit memory location where the seven is

initially placed. [Nb., for simplicity we are not using the 18-bit register width of the SEAforth™ 24a device here, although the underlying concept is extendable to that or any bit width.]

TBLS. 5-10 represent the values in the T-register and the S-register in a set of hypothetical +* multiplication examples. TBL. 5 shows the value nine (9) initially placed in the T-register and the value seven (7) placed in the S-register. Next, the value in T is right justified in the 8-bit field width such that the four leading bits are filled with zeros. Conversely, the value in S is left justified in the 8-bit field width so that the four trailing bits are filled with zeroes. TBL. 6 shows the result of these justifications.

Correct multiplication here requires the execution of four +* op-codes in series. The first +* operation has the following effects. The LB position of T is 1 (as shown in TBL. 6), so the values in T and S are added and the result is placed in T (as shown in the left portion of TBL. 7). Next, the value in T is shifted to the right one bit in the same manner described in 1a2b1 (above). The values after this first +* are shown in the right portion of TBL. 7.

The second +* operation is quite simple, because the LB position of T is 0. All of the bits in T are shifted right in the manner described in 2b1 (above). The values after this second +* are shown in TBL. 8.

The third +* operation is similar to the second, because the LB position of T is again 0. All of the bits in T are again shifted right in the manner described in 2b1 (above). The values after this third +* are shown in TBL. 9.

The fourth and final +* operation is similar to the first +* operation. The LB position of T is 1 (as shown in TBL. 9), so the values in T and S are added and the result is placed in T (as shown in the left portion of TBL. 10). Next, the value in T is shifted to the right one bit in the same manner described in 1a2b1 (above). The values after this fourth +* operation are shown in the right portion of TBL. 10.

The resultant T in TBL. 10 is the decimal value 63, which is what one expects when multiplying the numbers nine and seven.

A +* PSEUDO-CODE ALGORITHM FOR MULTIPLICATION

The multiplication of a positive value with a positive value will result in a correct product when the sum of the significant bits in T and S prior to the execution of this pseudo-code is less than or equal to 16 bits. And the multiplication of a positive value with a negative value will result in a correct product when the sum of the significant bits in T and S

prior to the execution of the pseudo-code is less than or equal to 17 bits. Note that S should contain the two's complement of the desired negative value in S prior to the execution of this pseudo code.

1. If the desired multiplication is of a positive value with a positive value:
   1a. Right justify t in the n bit field width of T.
      1a1. Fill all leading bits in T after the MSB of t with zeros. The number of leading bits to fill should be exactly n-n_t.
   1b. Justify s in the n bit field width of S so that the LSB of s is located one bit higher than the MSB of t in T.
      1b1. Fill all leading and trailing bits in S with zeros. The number of bits to fill should be exactly n-n_s.
   1c. Perform the multiplication.
      1c1. Complete a for-loop indexing from 1 to n_t.
         1c1a. Execute the +* pseudo-code as described for the general case (above).
2. If the desired multiplication is of a positive value with a negative value:
   2a. Right justify t in the n bit field width of T.
      2a1. Fill all leading bits in T after the MSB of t with zeros. The number of leading bits to fill should be exactly n-n_t.
   2b. Perform the two's complement of the value s in S.
      2b1. Bit shift the value s in S towards the HB of S by the number of significant bits n_t.
   2c. Perform the multiplication.
      2c1. Complete a for-loop indexing from 1 to n_t.
         2c1a. Execute, the +* pseudo-code as described for the general case, above.
3. If the desired multiplication is of a negative value with a negative value:
   3a. Perform the two's complement of the value t in T.
   3b. Perform the two's complement of the value s in S.
   3b. Execute 1a – 1c.

Of course, the multiplication of a negative value with a positive value is the same as 2 (above) for multiplication, as long as the negative value is in T and the positive value in S.

FIG. 8 is a flow chart of the inventive shift-add based multiplication process **200** in

accord with the present invention. In a step **202** the shift-add based multiplication process **200** starts or is invoked. In a step **204** a first value is arranged in a first memory location, *i.e.*, in the right justified manner described in 1 (above) if T is the first memory location. In a step **206** a second value is arranged in a second memory location, *i.e.*, in the left justified manner described in 2 (above) for multiplication if S is the second memory location. [Those skilled in the programming arts will readily appreciate that alternate programmatic control mechanisms than the following count-compare-work-decrement Approach can be used.] In a step **208** the number of iterations of the +* op-code is determined. Essentially, this number needs to equal the number of significant bits in the first value (in T). In a step **210** whether all needed iterations of the +* op-code have been performed is determined. If not, in a step **212** an iteration of the +* op-code is performed and in a step **214** the count still needed is decremented. Alternately, if step **210** determines that all needed iterations of the +* op-code have been performed, in a step **216** the product of the first and second values is now in the first memory (*i.e.*, in T).

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and that the breadth and scope of the invention should not be limited by any of the above described exemplary embodiments, but should instead be defined only in accordance with the following claims and their equivalents.

-*-

T → 0 0 0 1 → 1

S → 0 0 1 1 → 3

TBL. 1

T → 0 1 0 0 → 4

S → 0 0 1 1 → 3

TBL. 2

T → 0 0 1 0 → 2

S → 0 0 1 1 → 3

TBL. 3

T → 0 0 0 1 → 1

S → 0 0 1 1 → 3

TBL. 4

T → 0 0 0 0 1 0 0 1 → 9

S → 0 0 0 0 0 1 1 1 → 7

TBL. 5

T → 0 0 0 0 1 0 0 1

S → 0 1 1 1 0 0 0 0

TBL. 6

T → 0 1 1 1 1 0 0 1 → T → 0 0 1 1 1 1 0 0

S → 0 1 1 1 0 0 0 0 → S → 0 1 1 1 0 0 0 0

TBL. 7

T → 0 0 0 1 1 1 1 0

S → 0 1 1 1 0 0 0 0

TBL. 8

T → 0 0 0 0 1 1 1 1

S → 0 1 1 1 0 0 0 0

TBL. 9

T → 0 1 1 1 1 1 1 1 → T → 0 0 1 1 1 1 1 1

S → 0 1 1 1 0 0 0 0 → S → 0 1 1 1 0 0 0 0

TBL. 10

CLAIM OR CLAIMS

What is claimed is:

1 1. A method for performing a shift-add operation on multi-bit first and second values
2 respectively loaded in first and second memories of a processor, wherein the first and second
3 memories have bit-positions that can all be zero or one and the first memory has a low bit
4 (LB) and a high bit (HB), the method comprising:
5 (a) if the first memory LB is zero:
6 (1) if the first memory HB is zero:
7 (i) shifting the first value one bit-position lower; and
8 (ii) setting the first memory HB to zero, thereby arriving at a new
9 value in the first memory;
10 (2) if the first memory HB is one:
11 (i) shifting the first value one bit-position lower; and
12 (ii) setting the first memory HB to one, thereby arriving at said new
13 value; and
14 (b) if the first memory LB is one:
15 (1) adding the second value to the first value in the first memory;
16 (2) if said (b)(1) has not produced a carry, proceeding with said (a)(1); and
17 (3) if said (b)(1) has produced a carry, proceeding with said (a)(2)(i).

1 2. The method of claim 1, wherein said processor includes a plurality of registers and
2 said first memory comprises one or more said registers, or said second memory comprises
3 one or more said registers, or both.

1 3. A system for performing a shift-add operation on multi-bit first and second values, the
2 system comprising:
3 a processor having first and second memories to respectively hold the first and second
4 values, wherein said first and second memories each have bit-positions that
5 can all assume a state of zero or one, and wherein said first memory has a low
6 bit (LB) position and a high bit (HB) position;
7 an add logic to add the second value to the first value in said first memory;
8 a zero logic to set said state of said HB of said first memory to said zero state;
9 a one logic to set said state of said HB of said first memory to said one state;
10 a shift logic to shift the first value lower in said first memory by one said bit-position;
11 and
12 a logic to, when said LB of said first memory is in said zero state:
13 (1) when said HB of said first memory is in said zero state, operate
14 said shift logic and said zero logic, thereby arriving at a new value
15 in said first memory;
16 and alternately, when said HB of said first memory is in said one state:
17 (2) operate said shift logic and said one logic, thereby arriving at
18 said new value in said first memory;
19 and yet alternately, when said LB of said first memory is in said one state:
20 operate said add logic and if there is not a carry proceed at said (1) and
21 otherwise proceed at said (2).

1 4. The system of claim 3, wherein said processor includes a plurality of registers and
2 said first memory comprises one or more said registers, or said second memory comprises
3 one or more said registers, or both.

Hex (bit)			Binary (bit)					Hex (bit)			Binary (bit)				
1	0		4	3	2	1	0	1	0		4	3	2	1	0
0	0	; (return)	0	0	0	0	0	1	0	+*	1	0	0	0	0
0	1	;:	0	0	0	0	1	1	1	2*	1	0	0	0	1
0	2	jump	0	0	0	1	0	1	2	2/	1	0	0	1	0
0	3	call	0	0	0	1	1	1	3	not	1	0	0	1	1
0	4	unext	0	0	1	0	0	1	4	+	1	0	1	0	0
0	5	next	0	0	1	0	1	1	5	and	1	0	1	0	1
0	6	if	0	0	1	1	0	1	6	xor	1	0	1	1	0
0	7	-if	0	0	1	1	1	1	7	drop	1	0	1	1	1
0	8	@p+	0	1	0	0	0	1	8	dup	1	1	0	0	0
0	9	@a+	0	1	0	0	1	1	9	pop	1	1	0	0	1
0	a	@b	0	1	0	1	0	1	a	over	1	1	0	1	0
0	b	@a+	0	1	0	1	1	1	b	a@	1	1	0	1	1
0	c	!p+	0	1	1	0	0	1	c	· (nop)	1	1	1	0	0
0	d	!a+	0	1	1	0	1	1	d	push	1	1	1	0	1
0	e	!b	0	1	1	1	0	1	e	b!	1	1	1	1	0
0	f	!a	0	1	1	1	1	1	f	a!	1	1	1	1	1

FIG.1 (background art)

The 32 Venture Forth™ op-codes

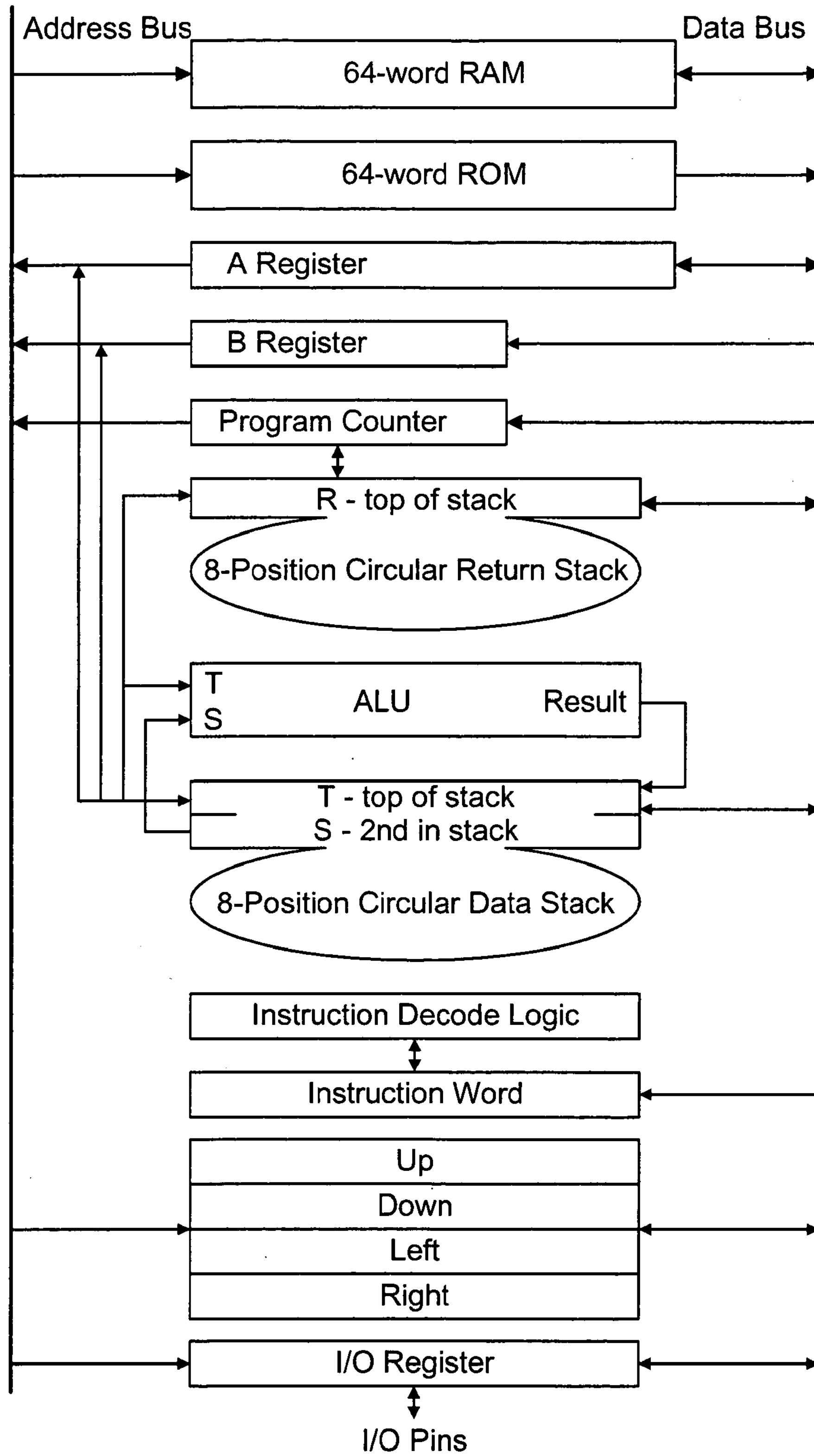

FIG.2 (background art)
The general architecture of the SEAForth 24a

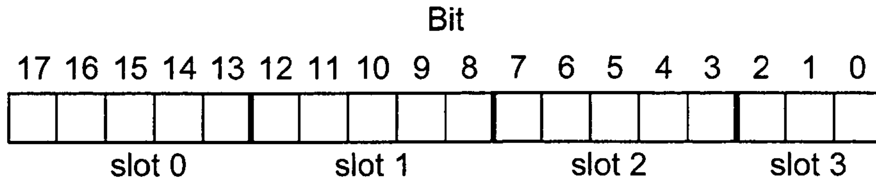

FIG. 3a

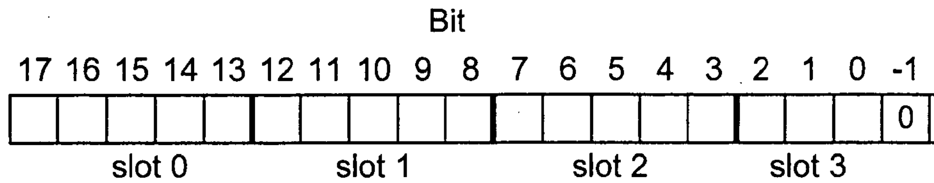

FIG. 3b

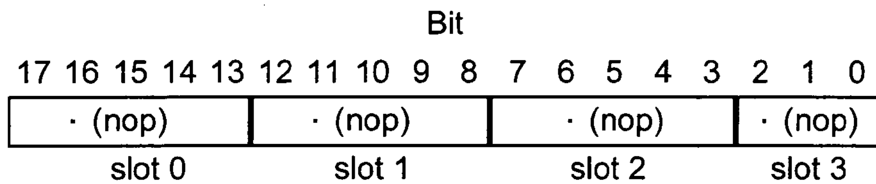

FIG. 4a

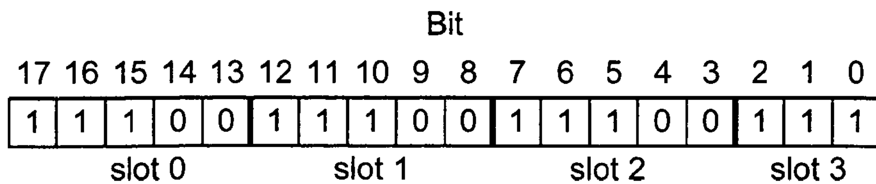

FIG. 4b

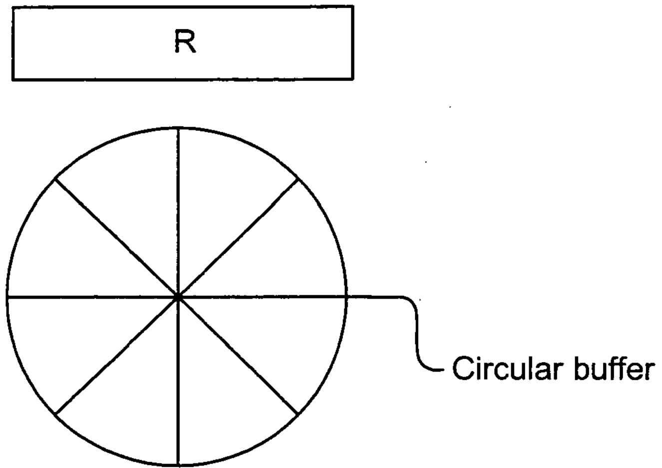

FIG. 5a (background art)
Return stack region

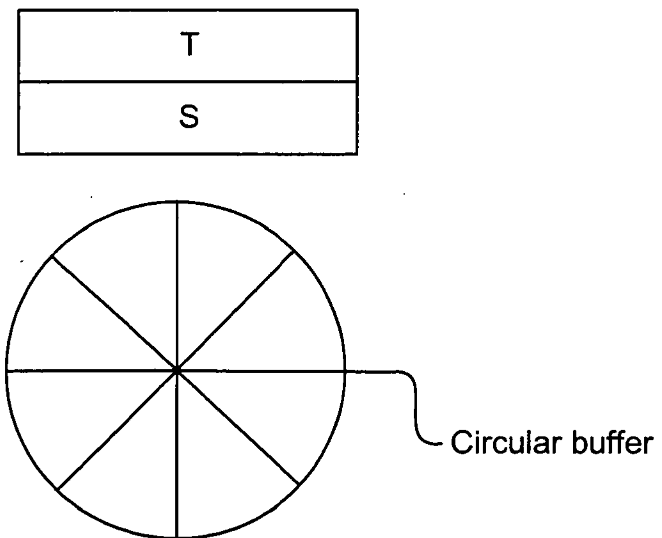

FIG. 5b (background art)
Data stack region

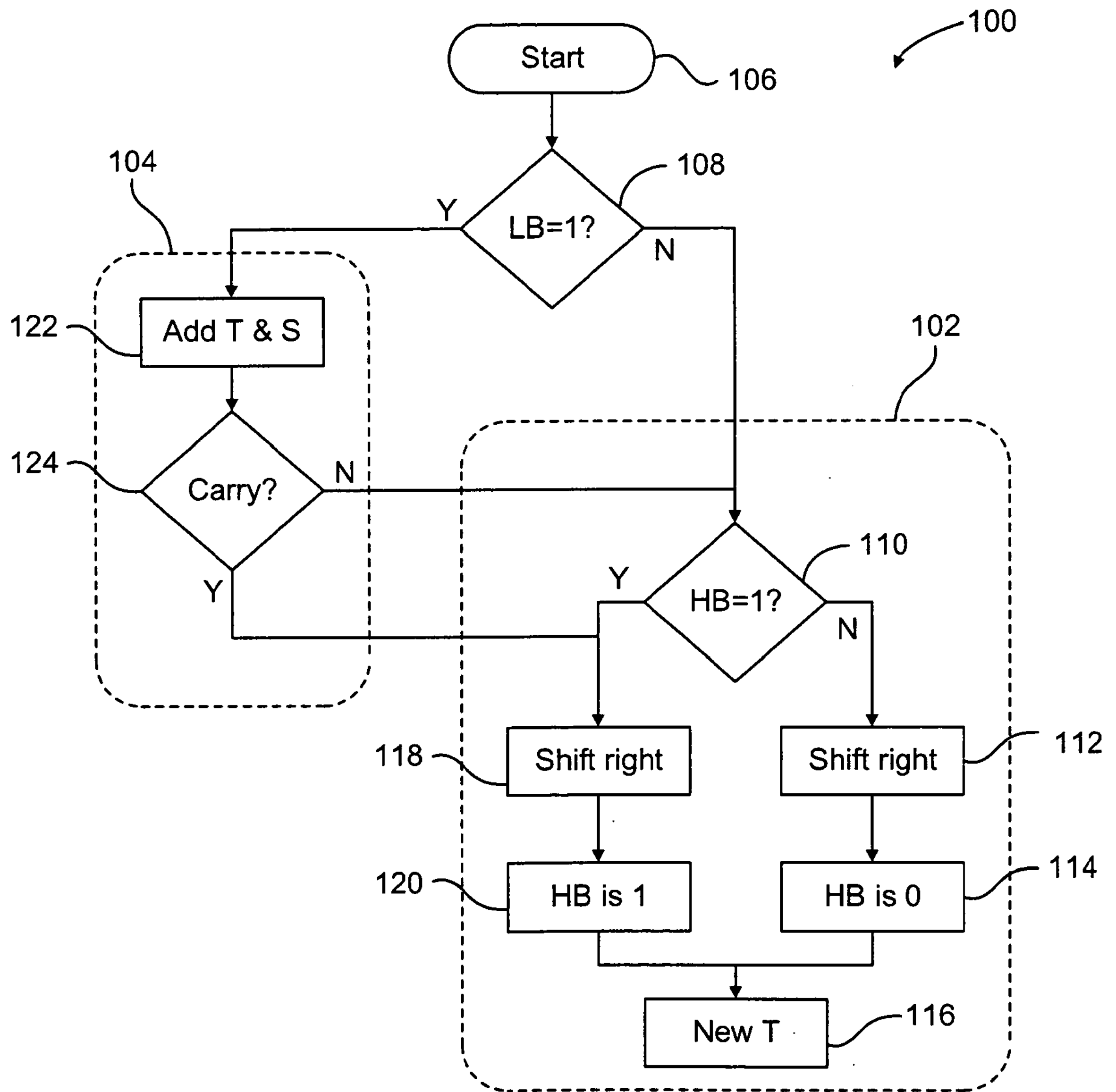

FIG. 6

old T			T + S			new T	
LB	HB		HB	carry bit		HB	HB -1
0	0		-	-		0	0
0	1		-	-		1	1
1	-		0	0		0	0
1	-		0	1		1	0
1	-		1	0		1	1
1	-		1	1		1	1

FIG. 7

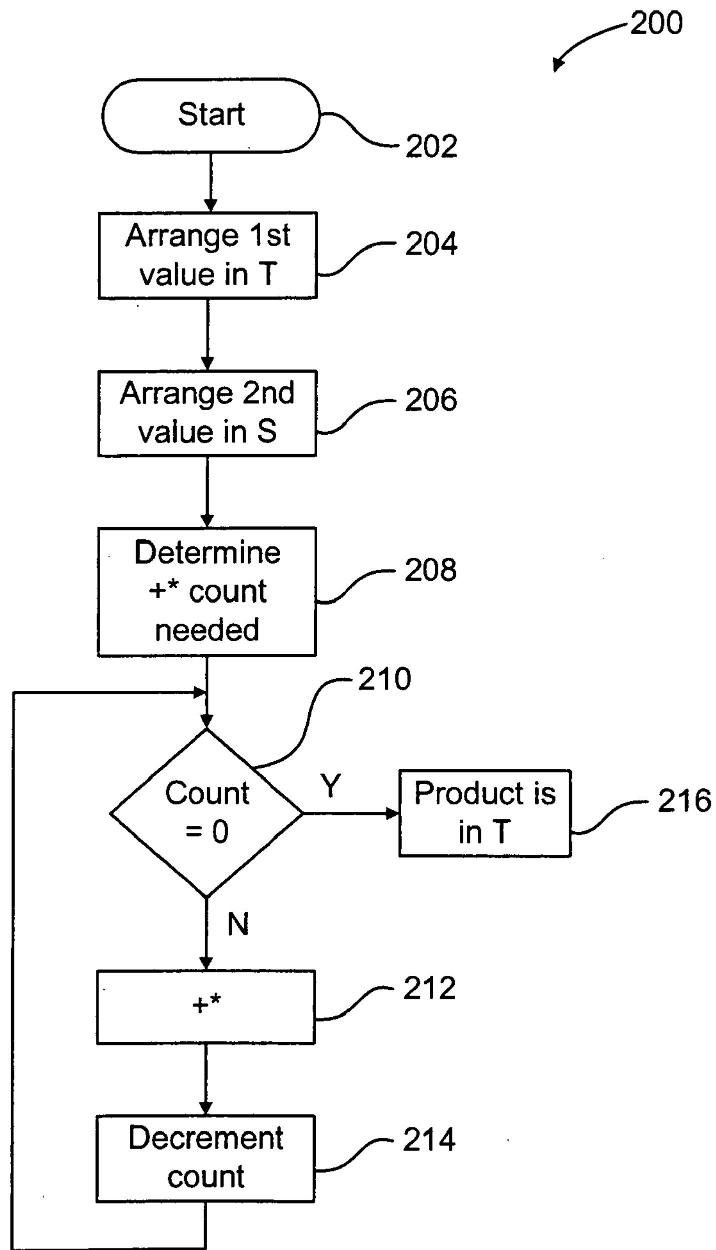

FIG. 8