

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
10 February 2005 (10.02.2005)

PCT

(10) International Publication Number
WO 2005/013084 A2

(51) International Patent Classification⁷: **G06F**

(21) International Application Number:
PCT/US2004/024667

(22) International Filing Date: 30 July 2004 (30.07.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/492,060 31 July 2003 (31.07.2003) US

(71) Applicant: **CRADLE TECHNOLOGIES, INC.**
[US/US]; 82-103 Pioneer Way, Mountain View, CA
94041-1502 (US).

(72) Inventors: **SIMON, Moshe, B.**; 601 San Tomas Place, San
Ramon, CA 94583 (US). **MACHNICKI, Erik, P.**; 1063
Morse Avenue, Apt. 8-200, Sunnyvale, CA 94089 (US).
HARRISON, David, A.; 1020 November Drive, Cuper-
tino, CA 95014 (US). **SINGH, Rakesh, K.**; E-11, Harsh
Vihar, D.P. Road, Aundh, Pune 411007 (IN). **LONGLEY,
Mark**; 321 Scott Street, Livermore, CA 94550 (US).

(74) Agent: **SCHNECK, Thomas**; Schneck & Schneck, P.O.
Box 2-E, San Jose, CA 95109-0005 (US).

(81) Designated States (unless otherwise indicated, for every
kind of national protection available): AE, AG, AL, AM,

AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN,
CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI,
GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE,
KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD,
MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,
PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM,
TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, 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, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI,
SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,
GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

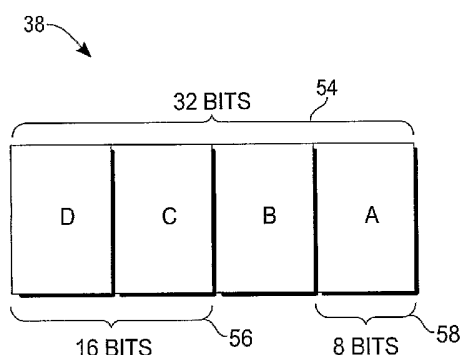
- as to applicant's entitlement to apply for and be granted a
patent (Rule 4.17(ii)) for all designations
- as to the applicant's entitlement to claim the priority of the
earlier application (Rule 4.17(iii)) for all designations

Published:

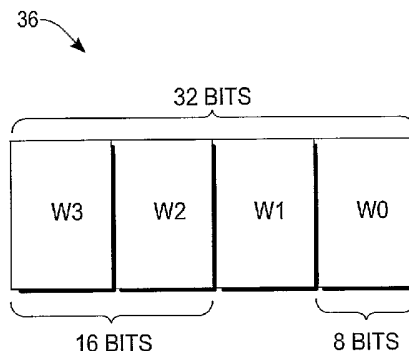
- without international search report and to be republished
upon receipt of that report

For two-letter codes and other abbreviations, refer to the "Guid-
ance Notes on Codes and Abbreviations" appearing at the begin-
ning of each regular issue of the PCT Gazette.

(54) Title: METHOD AND SYSTEM FOR PERFORMING OPERATIONS ON DATA AND TRANSFERRING DATA



WORD A



WORD B

(57) Abstract: Methods and systems for performing operations on data, such as parallel integer multiply accumulate operations on packed data and SIMD operations, and transferring data, are provided. A multiple accumulate unit ("MAC") performs operations on packed integer data and allows results to be individually retrieved from registers and the corresponding accumulator cleared after the read cycle. SIMD instructions, including parallel absolute value and parallel conditional move instructions, as well as a method and circuit for saturating results of operations are provided. Scatter/gather mode for DMA transfer of data provides a table of buffer descriptors which may be used in automatic buffer switching and a table of packet descriptors describing location and size of incoming data, thus decoding packet sizes from buffer sizes.

Description

METHOD AND SYSTEM FOR PERFORMING OPERATIONS
ON DATA AND TRANSFERRING DATA

5

TECHNICAL FIELD

This invention relates to operations performed on data by a processor, particularly parallel integer multiply accumulate operations on packed data and single instruction multiple data ("SIMD") operations on packed data, and direct memory access ("DMA") transfer of data between a processor and an attached device.

10

BACKGROUND ART

15

Processors are capable of performing various operations on data as well as transferring data. Multiple accumulate operations, SIMD operations, and DMA transfer of data are discussed below.

20

Multiply Accumulate Operations

Multiply-accumulate units ("MACs") perform multiplication and accumulation operations in a single instruction cycle in a processor. Usually, the result of a multiplication operation is added, or accumulated to, another result stored in an accumulator, or register. These units are often used to speed up video/graphics applications as well as digital signal processor operations such as convolution and filtering.

25

Single instruction, multiple data ("SIMD") style processing has been used to accelerate multimedia processing. Instruction sets for processors often include SIMD instructions where multiple data elements are packed in a single wide register, with the individual data elements operated on in parallel. One example is Intel's MMX (multimedia extension) TM instruction set.

35

-2-

This parallel operation on data elements accelerates processing.

As noted above, MAC operations are used to accelerate various applications. In addition to speed,
5 it would be desirable to have an architecture that is capable of handling multiply and accumulate operations for different-sized operands as required by the instruction (i.e., 8 x 8 operations, 16 x 16 operations, etc.). It would also be desirable to be able to retrieve
10 individual results of MAC operations and clear the corresponding accumulator. In addition, it would be advantageous to have a MAC unit which could provide the cross-product of operands, pack results into one register, and shift results where desired.

15

SIMD Operations

Single instruction, multiple data ("SIMD") style processing has been used to accelerate multimedia processing, including image processing and data
20 compression. Instruction sets for processors often include SIMD instructions where multiple data elements are packed in a single wide register, with the individual data elements operated on in parallel. Using this approach, multiple operations can be performed with one
25 instruction, thus improving performance. One example is INTEL's MMX (multimedia extension) instruction set.

It would be advantageous to provide new SIMD instructions and supporting circuitry to further enhance multimedia processing, for instance, image segmentation
30 or clipping.

DMA Transfer of Data

Processors whose computer bus architectures have Direct Memory Access ("DMA") capability may transfer
35 data between a device attached to the processor (such as

disk controllers, graphics cards, etc.) and the processor's system memory independent of the main CPU. DMA allows devices of different speeds to communicate without placing a processing burden on the CPU. DMA further speeds up data transfer by eliminating the fetch and execute cycles required when data transfers pass through the CPU.

In a DMA transfer, the CPU stops its bus activity to allow the DMA operation. A DMA controller then reads or writes memory bytes and transfers the data to the desired location; generally, a DMA transfer involves either a device read and memory write or a memory read and a device write. FIFO buffers may be used to store the data to be transferred.

It would be advantageous to enhance DMA transfer of data.

SUMMARY OF THE INVENTION

Methods and systems for performing operations on data, such as parallel integer multiply accumulate operations on packed data and SIMD operations, and transferring data are discussed below.

Multiply Accumulate Operations

A MAC that performs multiply accumulate operations on packed integer data is provided. In one embodiment, the MAC receives 2 32-bit data words which, depending on the specified mode of operation, each contain either four 8-bit operands, two 16-bit operands, or one 32-bit operand. Depending on the mode of operation, the MAC performs either sixteen 8 x 8 operations, four 16 x 16 operations, or one 32 x 32 operation. Results may be individually retrieved from registers and the corresponding accumulator cleared after the read cycle. In addition, the accumulators may be

globally initialized. Two results from the 8 x 8 operations may be packed into a single 32-bit register. The MAC may also shift and saturate the products as required.

5

SIMD Operations

SIMD instructions, including parallel absolute value and parallel conditional move, for parallel processing of packed data are provided as well as a circuit for saturating the result of an operation. Other operations in the instruction set include parallel add, parallel subtract, parallel compare, parallel maximum, and parallel minimum. The operations indicated by the instructions are carried out in the arithmetic logic unit ("ALU") of a processor.

An instruction indicates, among other things, the operation and the data, in the form of a data word containing data elements, on which the operation is performed. Each data word contains several elements; the number of elements is determined by the mode of operation indicated by the instruction. For instance, when an 8-bit mode is specified, a 32-bit data word contains 4 8-bit data elements, or operands, while in 16-bit mode, the same 32-bit data word contains 2 16-bit operands.

A parallel status flags ("PSF") register stores the parallel status flags (PSFs) which monitor the status of data elements in data word. PSFs indicate whether the result of an integer operation is zero, the sign of the result of an integer operation, whether there was a carry out from the ALU operation, and whether there was a 2's complement integer overflow result. The PSF register is updated whenever a SIMD instruction that updates PSF flags is performed.

A parallel conditional test ("PTEST") register contains a code which maps to a test condition. During

parallel conditional move ("PCMOV") instructions, status flags in the PSF register are compared to the test condition in the PTEST register and, if the flags and condition match, the suboperand corresponding to the flags in the PSF register is moved to a specified register.

During parallel absolute value ("PABS") instructions, the processor determines the absolute value of at least two operands and places the absolute value of the operands in specified registers. The absolute value is determined by using one of the following approaches based on the sign bit of each of the operands: 1) where the sign bit of an operands is 1 and at least one of the other bits is 1, the absolute value of the operand is the 2's complement of the operand; 2) where the sign bit of the operand is 1 and each of the other bits is 0, the absolute value of the operand is the 1's complement of the operand; and 3) where the sign bit of the operand is 0, the absolute value of the operand is the value of the operand.

A method and circuit for handling saturation of a result of an operation are also provided. When two m-bit operands are added, as in an addition, average, or subtraction operation, if an average instruction is executed, the m most significant bits are output; otherwise, the m least significant bits are output and the result is saturated if there is overflow and saturation is enabled.

30 DMA Transfer of Data

A method and system for DMA transfer of data in "scatter/gather" mode is provided. In one embodiment, a table of buffer descriptors may be used to determine the next buffer to be used when a current buffer storing data that has been transferred or will be transferred and may

be used in automatic buffer switching, which does not require processor intervention. The method and system also provide for a table of packet descriptors which describes location and size of incoming data and can
5 indicate whether a packet of data straddles two or more buffers, thus decoupling packet sizes from buffer sizes. The method and system also provide for optional end-of-buffer and end-of-packet notifications to the controlling processor as well as optional termination by hardware,
10 without software intervention, on an end-of-buffer condition.

In one embodiment, a method of transferring data between a processor and an attached direct memory access ("DMA") device in scatter/gather mode comprises
15 maintaining a table of buffer descriptors for determining a start address and size of a next buffer to be used when transferring data when a current buffer counter reaches zero, wherein entries in the table are linked programmatically; transferring data between a current
20 buffer and the device via DMA; automatically switching buffers when the selected buffer counter reaches zero, wherein a next buffer descriptor table entry is read from memory and a new buffer pointer and size is updated based on the table entry; and transferring data between the
25 next buffer and the device via DMA.

In another embodiment, a processor for transferring data between the processor and an attached direct memory access ("DMA") device in scatter/gather mode comprises means for connecting the processor to the
30 DMA device; a plurality of buffers for storing data transferred between the DMA device and the processor, each of the plurality of buffers having a buffer counter; a buffer descriptor table for each Data In FIFO and each Data Out FIFO of a DMA channel transferring data between
35 the DMA device and the processor, each entry in the

buffer descriptor table having a buffer address of one of the plurality of buffers, a buffer size of one of the plurality of buffers, and sequencing information for calculating a start address and size of a next buffer to be used for storing data transferred between the DMA device and the processor when a current buffer counter reaches zero, wherein entries in the buffer descriptor table are made programmatically; and a bus interface unit for transferring data between the DMA device and the processor, the bus interface unit including hardware for automatically fetching the calculated start address and size of the next buffer to be used to store data transferred between the DMA device and the processor from the buffer descriptor table so the bus interface unit can automatically switch buffers for storing data transferred between the DMA device and the processor when the current buffer counter reaches zero.

In another embodiment, a method of transferring data between a processor and an attached direct memory access ("DMA") device in scatter/gather mode comprises transferring a packet of data from the DMA device to the processor; detecting an end-of-packet condition; and writing out a packet descriptor to a table in memory, wherein the packet descriptor includes a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary.

In another embodiment, a processor for transferring data between the processor and an attached direct memory access ("DMA") device in scatter/gather mode comprises the means for connecting the processor to the DMA device; a plurality of buffers for storing data transferred between the DMA device and the processor; and a bus interface unit for transferring data between the DMA device and the processor, the bus interface unit including hardware for detecting an end-of-packet event

and writing a packet descriptor to a packet descriptor table for each Data In FIFO of a DMA channel, each packet descriptor having a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary, wherein packet descriptors are written to the packet descriptor table each time an end-of-packet event is detected.

BRIEF DESCRIPTION OF THE DRAWINGS

10 Fig. 1 is a block diagram of the multiply accumulate unit ("MAC") of the invention.

 Fig. 2 is a block diagram of a processor status word used with the apparatus of Fig. 1.

15 Fig. 3 is a chart of modes of operation and resulting operands, number of operations per cycle, and obtained results for the apparatus of Fig. 1.

 Fig. 4 is a block diagram of data words used as input in the apparatus of Fig. 1.

20 Fig. 5 is a block diagram of a 16 x 16 multiplier block in the apparatus of Fig. 1.

 Fig. 6 is a block diagram of a saturation circuit in the apparatus of Fig. 1.

 Fig. 7a is a block diagram of a shift and saturate circuit in the apparatus of Fig. 1.

25 Fig. 7b is a block diagram of a shift and saturate circuit in the apparatus of Fig. 1.

 Fig. 8 is a block diagram of a combined saturation circuit in the apparatus of Fig. 1.

30 Fig. 9 is a block diagram of a processor that may be used to execute the SIMD instructions of the invention.

 Fig. 10 is a block diagram of a processor status word that may be used to indicate an SIMD instruction in accordance with the invention.

Fig. 11 is a block diagram of an SIMD instruction in accordance with the invention.

Fig. 12 is a block diagram of data words used in executing an SIMD instruction in accordance with the invention.

Fig. 13 is a diagram of a saturation circuit used when executing an SIMD instruction in accordance with the invention.

Fig. 14 is a flow chart showing execution of a parallel absolute value instruction in accordance with the invention.

Fig. 15 is a flow chart showing execution of a parallel conditional move instruction in accordance with the invention.

Fig. 16 is a block diagram of a processor's programmable I/O system and devices attached to the processor in one embodiment of the invention.

Fig. 17 is a block diagram of a processor's bus interface unit and DMA logic for a device's pin group in accordance with the invention.

Fig. 18a is a block diagram of the format of DMA registers in accordance with the invention.

Fig. 18b is a block diagram of the format of DMA registers in accordance with the invention.

Fig. 19 is a block diagram of DMA logic in accordance with the invention.

Fig. 20 is a flowchart showing how automatic buffer switching is performed in accordance with the invention.

Fig. 21 is a flowchart showing how boundary alignment is performed in accordance with the invention.

DETAILED DESCRIPTION OF THE INVENTION

Multiply Accumulate Operations

5 In one embodiment of the invention, the MAC is part of a digital signal engine ("DSE") coprocessor. In Fig. 1, a conceptual block diagram of the MAC unit 10 features sixteen 8 x 8 multipliers 12, each with a corresponding adder 14, accumulator 18, and accumulator register 22. In this embodiment, the adder 14 is a 20-bit adder, the accumulator 18 is a 20-bit accumulator, and the register 22 is a 20-bit register. A preclear multiplexer 20 is coupled to the adder 14 and is used to initialize the accumulators 18. A postclear multiplexer 18 is also coupled to the adder 14 and is used to clear any accumulator register 22 that has been accessed in order to retrieve the result of MAC operations. The preclear 20 and postclear 28 multiplexers are set by inputs 28, 30 received by the MAC unit 10. In addition, the unit 10 receives input (for instance, in a processing instruction) indicating whether the accumulator product should be saturated (SA 34) and/or whether the product should be shifted and saturated (SSP 32). The unit 10 is able to send overflow bits 24 to other registers in the processor, for instance hardware registers.

25 A DSE processor status word ("PSW") register controls processor operation in one embodiment of the invention. In Fig. 2, the PSW 122 is 32 bits long and includes the DSE program counter 124, which holds the address of the next DSE instruction to be executed. For purposes of the invention, the other bits of interest include bits 26 and 27, MACM0 128 and MACM1 130, which indicate the mode in which the MAC operates:

-11-

Bit 27	Bit 26	Mode
0	0	Default mode
0	1	8-bit packed mode (8 x 8 mode)
1	0	16-bit packed mode (16 x 16 mode)
1	1	32-bit mode (32 x 32 mode)

Bit 28, the SA bit 132, indicates whether the accumulator value should be saturated (i.e., if the bit is set to "1," the value is saturated). Bit 29, the SSP bit 134, indicates whether the product should be shifted and saturated (i.e., if the bit is set to "1," the product is shifted and saturated). The remaining bits 136 are used to control processor operation. The use of the PSW and the assignment of bits is included here as an example; in other embodiments, the operation of the MAC may be controlled in other ways.

The MAC of the invention receives two z-bit words, each containing a number of m-bit operands, and, depending on the operation mode determined by an instruction, performs a number of m x m multiply accumulate operations. Results of the multiply accumulate operations are placed in accumulator registers, which may be accessed individually in order to retrieve results. Fig. 3 shows that in one embodiment of the invention, the MAC receives two 32-bit words as input which each contain, depending on the mode of operation, four independent 8-bit operands (8 x 8 mode), two independent 16-bit operands (16 x 16 mode), and one 32-bit word (32 x 32 mode). In both 8 x 8 and 16 x 16 modes, each operand is independently configured as signed or unsigned. In 8 x 8 mode, sixteen 8 x 8 MACs may be performed per cycle, resulting in sixteen 16-bit products accumulated into sixteen signed 20-bit accumulator registers. In 16 x 16 mode, four 16 x 16 MACs may be performed per cycle, with four 32-bit products accumulated into 4 signed 40-bit accumulator registers.

-12-

In 32 x 32 mode, one 32 x 32 MAC is performed per cycle and one 64-bit product is accumulated into one of four signed 80-bit accumulator registers. Other embodiments of the invention may perform MAC operations on operands
5 containing a different number of bits than listed above, for instance 64-bit operands.

Referring to Fig. 1, the MAC unit 10 receives two data words, A 38 and B 36, as input as well as an indication (for instance, in the instruction) of whether
10 A 38 and B 36 are signed or unsigned 42, 40. The MAC unit 10 receives an activation signal 26 that also determines what mode it will operate in for the cycle, i.e., 8 x 8 mode 50, 16 x 16 mode 48, 24 x 24 mode 46 (in one embodiment, the MAC unit's 10 default mode is to
15 operate as a 24 x 24 floating point MAC), or 32 x 32 mode 44.

As shown in Fig. 4, the data words A 38 and B 36 in one embodiment consist of 32 bits (or four bytes) apiece (in other embodiments, the words may consist of a
20 larger or fewer number of bits). Depending on the mode of operation, each word may consist of one 32-bit operand 54 (i.e., DCBA and W3W2W1W0), two 16-bit operands 56 (i.e., DC, BA, W3W2, and W1W0, where D and W3 are the most significant bytes and A and W0 are the least
25 significant bytes), or four 8-bit operands 58 (i.e., D, C, B, A, W3, W2, W1, and W0 where D and W3 are the most significant bytes and A and W0 are the least significant bytes).

As noted above, when the MAC unit operates in
30 8x8 mode, the results of sixteen 8 x 8 MAC operations are placed in sixteen 20-bit accumulator registers, or packed byte integer MAC accumulator registers (PBIMs). An example of how operands and the accumulator registers (here labeled 0 through 15) may be mapped follows:

35

-13-

```

PBIM15 += D * W3
PBIM14 += D * W2
PBIM13 += C * W3
PBIM12 += C * W2
5  PBIM11 += D * W1
PBIM10 += D * W0
PBIM9 += C * W1
PBIM8 += C * W0
PBIM7 += B * W3
10 PBIM6 += B * W2
PBIM5 += A * W3
PBIM4 += A * W2
PBIM3 += B * W1
PBIM2 += B * W0
15 PBIM1 += A * W1
PBIM0 += A * W0

```

In the preclear case (when the accumulators are set to "0"), the "+" is replaced by "." The accumulator registers are logical registers and can be implemented in any way so that the registers are shared regardless of the MAC's mode of operation.

In 16 x 16 mode, the results of four 16 x 16 multiply accumulate operations are placed in 40-bit accumulator register, or packed half-word integer MAC ("PHIM") accumulator registers. An example of how operands and PHIM accumulator registers (here labeled 0 through 3) may be mapped follows:

```

30 PHIM3 += DC * W3W2
PHIM2 += DC * W1W0
PHIM1 += BA * W3W2
PHIM0 += BA * W1W0

```

In the preclear case, the "+=" is replaced by "=". The accumulator registers are logical registers and can be implemented in any way so that the registers are shared regardless of the MAC's mode of operation.

5 In 32 x 32 mode, the results of the single 32 x 32 multiply accumulate operation is placed in one of four 80-bit accumulator registers, or unpacked integer MAC (UIM) accumulator registers. Which UIM register is used is determined by instruction type. An example of how the
10 operands and UIM accumulator registers (where n is a number from 0 to 3) may be mapped follows:
UIM(n) += DCBA * W3W2W1W0.

 In the preclear case, the "+=" is replaced by "=". The accumulator registers are logical register and
15 can be implemented in any way so that the registers are shared regardless of the MAC's mode of operation.

 In one embodiment, the PBIM, PHIM, and UIM registers use the same shared 320 bits as indicated in the following table. In other embodiments, other
20 approaches may be employed.

	PBIM0 [19:0]		UIM0 [19:0]
	PBIM1 [19:0]		UIM0 [39:20]
	PBIM2 [19:0]		UIM0 [59:40]
25	PBIM3 [19:0]		UIM0 [79:60]
	PBIM4 [19:0]	PHIM0 [19:0]	UIM1 [19:0]
	PBIM5 [19:0]	PHIM0 [39:20]	UIM1 [39:20]
	PBIM6 [19:0]	PHIM1 [19:0]	UIM1 [59:40]
	PBIM7 [19:0]	PHIM1 [39:20]	UIM1 [79:60]
30	PBIM8 [19:0]	PHIM2 [19:0]	UIM2 [19:0]
	PBIM9 [19:0]	PHIM2 [39:20]	UIM2 [39:20]
	PBIM10 [19:0]	PHIM3 [19:0]	UIM2 [59:40]
	PBIM11 [19:0]	PHIM3 [39:20]	UIM2 [79:60]
	PBIM12 [19:0]		UIM3 [19:0]
35	PBIM13 [19:0]		UIM3 [39:20]
	PBIM14 [19:0]		UIM3 [59:40]
	PBIM15 [19:0]		UIM3 [79:60]

-15-

In Fig. 5, when the MAC is in 16 x 16 mode, the input words A 38 and B 36 are divided into 16-bit segments and sent to 16 x 16 multiplier blocks 62 which are described in greater detail 78 below. When the 16 x 16 multiplier block 62 is to determine the product of BA * W1W0, the individual operands B 86, A 84, W1 82, and W0 80 are input to 8 x 8 multiplier blocks 12. The multiplication operations are carried out and the results are output to an adder 64, which is a 16 x 16 partial product assembler, and each multiplier's 12 20-bit accumulator 18. The results may be sign extended 66 as necessary before being placed in the accumulators 18.

An indication 38, 36 of whether the operands are signed is provided (in one embodiment, as will be discussed in greater detail below, in the instruction). The accumulators 18 may add their contents to the products of the multipliers 12 unless a pre- or postclear operation has been performed 70, in which case the content of the accumulator 18 is forced to "0." The products placed in the accumulator 18 are determined by the MAC's mode of operation 26. For instance, in 16 x 16 mode, the partial product from the adder 64 is passed through a multiplexer 68 and to the accumulator 18. However, in 8 x 8 mode, the product of the 8 x 8 operation is passed through the multiplexer to the accumulator 18. Overflow bits 24 (discussed in greater detail below) are sent to the appropriate register 76. The products of the accumulators 18 are then sent to an order swap multiplexer 74 and then on to the accumulator registers.

Instructions are used to initiate packed integer MAC operations. In one embodiment, the instruction also specifies whether the operands are signed or unsigned. The following instructions, for use with Cradle's DSE coprocessor, are illustrative of the

type of instructions that may be used with the MAC.
 Other instructions may be used in other embodiments. In
 the following table, the index "k" of the accumulator
 depends on the indices "i" and "j" of the packed
 5 operands.

10

15

20

Instruction	Action	Comment
PIMACUU	$A[i] * B[j] + \text{PIM}[k] \rightarrow \text{PIM}[k]$	A, B unsigned; 8 x 8, 16 x 16 mode (PIM is the accumulator value)
PIMACSU	$A[i] * B[j] + \text{PIM}[k] \rightarrow \text{PIM}[k]$	A signed, B unsigned; 8 x 8, 16 x 16 mode
PIMACSS	$A[i] * B[j] + \text{PIM}[k] \rightarrow \text{PIM}[k]$	A, B signed, 8 x 8, 16 x 16 mode
PIMACPUU	$A[i] * B[j] \rightarrow \text{PIM}[k]$	A, B unsigned; 8 x 8, 16 x 16 mode; preclear all accumulators
PIMACPSU	$A[i] * B[j] \rightarrow \text{PIM}[k]$	A signed, B unsigned; 8 x 8, 16 x 16 mode; preclear all accumulators
PIMACPSS	$A[i] * B[j] \rightarrow \text{PIM}[k]$	A, B signed; 8 x 8, 16 x 16 mode; preclear all accumulators
IMAC0	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM0 (M is the accumulator value)
IMAC1	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM1
IMAC2	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM2
IMAC3	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM3
IMACP0	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM0; preclear accumulator
IMACP1	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM1; preclear accumulator
IMACP2	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM2; preclear accumulator
IMACP3	$A * B + M[j] \rightarrow M[j]$	A, B unsigned; 32 x 32 mode; destination register UIM3; preclear accumulator

-17-

In embodiments where the MAC can also operate as a 24 x 24 floating point MAC ("FMAC"), the instructions can have the same opcodes as the FMAC.

The accumulator registers may be accessed using move-like instructions (i.e., the registers are used as source operands in move instructions). In one embodiment, the following logical registers may be accessed for results; other embodiments may employ a different approach.

- 10 1) Registers for getting sign-extended 20-bit results for the 8 x 8 case
 - 15 a) PBIM0-PBIM15 - a 20-bit accumulator register value can be moved sign-extended to a 32-bit dual port data memory ("DPDM") register in the DSE.
 - b) PBIMC0-PBIMC0 - a 20-bit accumulator register value can be moved sign-extended to 32-bit DPDM register in the DSE; the accumulator is cleared at the end of the read cycle.
- 20 2) Registers for getting sign-extended upper 16-bit results for the 8 x 8 case
 - 25 a) UPBIM0-UPBIM15 - the sixteen most significant bits ("msbs") of the 20-bit accumulator register value can be moved sign-extended to a 32-bit DPDM register in the DSE.
 - b) UPBIMC0-UPBIMC15 - the sixteen msbs of the 20-bit accumulator register value can be moved sign-extended to a 32-bit DPDM register in the DSE; the accumulator is cleared at the end of the read cycle.

Note: Extracting the upper 16-bits and sign-extending the value is not integer division by sixteen with rounding towards zero. It is division by sixteen with rounding towards negative infinity.

-18-

- 3) Registers for getting two 16-bit results for the 8 x 8 case packed into a single 32-bit register
- a) PLPBIMC0-PLPBIMC7 - the sixteen least significant bits ("lsbs") of two 20-bit accumulator register values can be packed into one DPDM register; the accumulator is cleared at the end of the read cycle.
 - b) PUPBIMC0-PUPBIMC7 - the sixteen msbs of two 20-bit accumulator register values can be packed into one DPDM register; the accumulator is cleared at the end of the read cycle.
- 4) Registers for getting results for the 16 x 16 case
- a) PHIMT0-PHIMT3 - the 32 msbs of a 40-bit accumulator register value can be moved into a 32-bit DPDM register.
 - b) PHIMTC0-PHIMTC3 - the 32 msbs of a 40-bit accumulator register value can be moved into a 32-bit DPDM register; the accumulator is cleared at the end of the read cycle.
 - c) PHIMU0-PHIMU3 - the 8 msbs of a 40-bit accumulator register value can be moved sign-extended into a 32-bit DPDM register.
- 5) Registers for getting results for the 32 x 32 case
- a) UIML0-UIML3 - the 32 lsbs of an 80-bit accumulator register value can be moved into a 32-bit DPDM register.
 - b) UIMU0-UIMU3 - the 32 msbs of an 80-bit accumulator register value can be moved into a 32-bit DPDM register.
- The MAC unit described herein uses a two-stage pipeline. During the DSE execute stage, operands are clocked into the MAC pipeline. Results are available 2 cycles later. A register holds overflow bits from the MAC. In one embodiment, the overflow register is a read-only hardware register. The following tables show which

overflow bits are visible depending on the MAC mode.
Other embodiments may use a different approach.

MAC Mode	Bit	Function
00	31:0	reserved
01	31:18	16-bit PBIM(n) accumulator overflow bits
01	17	16-bit PBIM1 accumulator overflow bit
01	16	16-bit PBIM0 accumulator overflow bit
01	15:2	20-bit PBIM(n) accumulator overflow bits
01	1	20-bit PBIM1 accumulator overflow bit
01	0	20-bit PBIM0 accumulator overflow bit
10	31:8	reserved
10	7	32-bit PHIM3 accumulator overflow bit
10	6	32-bit PHIM2 accumulator overflow bit
10	5	32-bit PHIM1 accumulator overflow bit
10	4	32-bit PHIM0 accumulator overflow bit
10	3	40-bit PHIM3 accumulator overflow bit
10	2	40-bit PHIM2 accumulator overflow bit
10	1	40-bit PHIM1 accumulator overflow bit
10	0	40-bit PHIM0 accumulator overflow bit
11	31:8	reserved
11	7	64-bit UIM3 accumulator overflow bit
11	6	64-bit UIM2 accumulator overflow bit
11	5	64-bit UIM1 accumulator overflow bit
11	4	64-bit UIM0 accumulator overflow bit
11	3	80-bit UIM3 accumulator overflow bit
11	2	80-bit UIM2 accumulator overflow bit
11	1	80-bit UIM1 accumulator overflow bit
11	0	80-bit UIM0 accumulator overflow bit

Signed overflow occurs when the two inputs to the accumulator adder have the same sign but the output of the adder has the opposite sign. If A and B are the inputs to the adder and Sum is the output, the accumulator overflow bits are defined as follows:

16-bit PBIM(n) overflow = (A[15] XNOR B[15]) AND (A[15] XOR Sum[15])

20-bit PBIM(n) overflow = (A[19] XNOR B[19]) AND (A[19] XOR Sum[19])

32-bit PHIM(n) overflow = (A[31] XNOR B[31]) AND (A[31] XOR Sum[31])

40-bit PHIM(n) overflow = (A[39] XNOR B[39]) AND (A[39] XOR Sum[39])

When both operands are signed, or only operand A is signed, overflow is calculated for MAC operations in one of two ways depending on the embodiment. The calculations are as follows:

- 5 1) Overflow from bit $n-1$ = CarryOut($n-1$) XOR CarryOut($n-2$) of adder, or
- 2) Overflow = $\sim(\text{SignProduct XOR SignAccumlator Operand}) \text{ AND } (\text{SignAdder XOR SignProduct})$

When both operands in an 8 x 8 or 16 x 16 operation are unsigned, the value of the 16- or 32-bit overflow bit is undefined. The accumulator overflow bits for unsigned addition are as follows:

64-bit UIM(n) overflow = "UIM(n) bit 63 carry-out"

80-bit UIM(n) overflow = "UIM(n) bit 80 carry-out"

15 Overflow bits are sticky and remain set unless cleared explicitly, for instance, when the corresponding accumulator is cleared by accessing a postclear register or when a pre-clear instruction is executed.

 In Fig. 6, in 16 x 16 mode, accumulator values
20 90 may be saturated after the accumulator values are provided. In one embodiment, the SA bit 34 in the PSW is set to indicate saturation should occur if there is overflow from bit 31 94. If these conditions are met 112, and the overflow is in the positive direction, then
25 0x7fffffff 102 is sent to the register 110. If the overflow is in the negative direction, 0xff80000000 104 is sent to the register.

 In Fig. 7a, in 16 x 16 mode a bit (in one embodiment, the SSP bit in the PSW) 32 may be set to
30 shift left by one and saturate the product if necessary before it is sent to the register. When the bit 32 is set, the product from the multiplier block 62 is shifted left by 1 120 and saturated 118 where necessary. When 0x8000 is multiplied by 0x8000, the result is 0x40000000.
35 When 0x40000000 is shifted left multiply by 2, the sign

changes. If this occurs, the result must be saturated 18 to the greatest positive number, 0x7FFFFFFF. The results can be sign extended 66, depending on the operands 36, 38.

5 In Fig. 7b, an alternative method 140 of saturation uses two comparators 142, 144 to explicitly check the input operands. Saturation only occurs if both input operands 152, 154 are 0x8000. A check of whether both inputs are 0x8000 146 will determine if saturation 10 118 is required.

Fig. 8 shows the combined saturation circuit 114; the aspects of the combined circuit 114 have been discussed in Figs. 6 and 7a.

15 SIMD Operations

Fig. 9 shows a processor 210, or digital signal engine ("DSE"), that may be used to execute SIMD instructions in one embodiment of the invention. Among the features in the DSE 210 are an instruction memory 20 218, an instruction register 212, a dual port data memory 214, and an integer SIMD ALU 216, where the SIMD instructions are executed. Instructions are stored in a processor-readable medium, which includes any medium that can store or transfer information; examples of a 25 processor-readable medium include an electronic circuit, a semiconductor memory device, a ROM, a flash memory, a floppy diskette, a compact disc, an optical disc, etc. Other processors may be used in other embodiments.

In one embodiment, the DSE is controlled by a 30 processor status word ("PSW") register. In Fig. 10, the PSW 220 is 32 bits long and includes the DSE program counter 222, which holds the address of the next DSE instruction to be executed. For purposes of the invention, other bits of interest include: bit 15, the 35 non-saturation ("NSAT") bit 258, which when set to "1"

-22-

indicates the result should not be saturated, and if set to "0" indicates the result should be saturated if necessary; bit 25, the HSIMD bit 234, which when set to "1" indicates that half-word (for instance, if a word is 32 bits long, half a word is 16 bits) operations should be used (in one embodiment, if the HSIMD bit is not set to "1", 8-bit operations should be used); bit 30, the USIMD bit 236, which when set to "1", indicates the PADD, PSUB, PAVG, PCMP, PMIN, and PMAX operations use unsigned operands; and bit 31, the SIMD bit 238, which when set to "1" indicates SIMD instructions are to be used (this bit may be employed, for instance, when SIMD instructions are aliased with sum of absolute difference ("SAD") instructions in one embodiment). The remaining bits 224 are used to control processor operation. The use of the PSW 220 and the assignment of bits is included here as an example; in other embodiments, the use of SIMD instructions may be controlled in other ways.

With respect to Fig. 11, a DSE instruction 240 used in one embodiment is 20 bits long. Six bits indicate the OpCode 242, and 7 bits are used to indicate register addresses (rb and ra) 244 and 246 of operands. In Fig. 12, the data words, Word A 248 and Word B 250, are shown to be 32 bits long 254 in one embodiment (in other embodiments, data words may consist of some other length) and to contain a number of data elements. Depending on the type of operations specified in the PSW, each 32-bit word 238, 250 may consist of two 16-bit words (for instance, bytes D and C 252) or four 8-bit words (for instance, byte A 256).

A parallel status flags ("PSF") register is part of the DSE. PSFs are used to monitor the status of data elements in data words. The flags are as follows: Zero ("Z") indicates if the result of an integer operation is zero; Sign ("S") indicates the sign of the

result of an integer operation; Carry ("CY") indicates there was a carry out from the ALU operation; and Overflow ("OV") indicates a 2's complement integer overflow result. The register has the following format:

5

10

Bit	Function
31:16	reserved
15:12	PSF3 flags
11:8	PSF2 flags
7:4	PSF1 flags
3	PSF0 OV flag
2	PSF0 CY flag
1	PSF0 S flag
0	PSF0 Z flag

15

The PSF register is updated whenever a SIMD instruction that updates PSF flags is performed. In 8-bit mode, computations on byte 0 (the least significant byte) affect PSF0, computations on byte 1 affect PSF1, etc. In 20 16-bit mode, computations on the lower half-word affect PSF1 while computations on the upper half-word affect PSF3; PSF0 and PSF2 are undefined. Other embodiments of the invention may feature different approaches to handling PSFs.

25

The DSE also features a parallel condition test ("PTEST") register. The PTEST register is used when a parallel conditional move ("PCMOV") instruction is executed. As discussed in greater detail below, a PCMOV operation compares status flags in the PSF register 30 against the test condition specified in the PTEST register; if the flags and the condition match, the suboperand is moved to a specified register. The PTEST register has the following format:

35

Bit	Function
31:4	reserved
3:0	condition code

Each 4-bit condition code in the PTEST register maps to a test condition as follows:

	Code	Mnemonic	Compare Function	Description
5	0	JMP		Move always
	1	JCY		Move if CY=1
	2	JE	Equal	Move if zero (Z=1)
	3	JNE	Not Equal	Move if not zero (Z=0)
	4	JL	Less Than	Move if negative = (sign XOR overflow)
10	5	JGE	Greater or Equal	Move if positive
	6	JG	Greater Than	Move if positive non-zero = not zero AND not (sign XOR overflow)
	7	JLE	Less or Equal	Move if zero = zero OR (sign XOR overflow)
	8	JOV		Move if overflow (OV=1)
	9	JNOV		Move if not overflow (OV=0)
15	10	JS		Move if sign = 1 (S=1)
	11	JNS		Move if sign = 0 (S=0)
	12			reserved
	13	JHI	Unsigned Greater Than	Move if High (CY=0 and Z=0)
	14	JLS	Unsigned Less Than or Equal	Move if Lower or Same (CY=1 OR Z=1)
20	15			reserved

Other embodiments of the invention may feature different approaches to handling condition codes and the PTEST register.

25 SIMD instructions may be executed when the DSE is in SIMD mode (in other words, the SIMD bit discussed above is set to "1"). These instructions take 1 cycle to execute. SIMD instructions which may be executed by the processor described above include the following: a
 30 parallel absolute value ("PABS") instruction, which determines the absolute value of an operand and places that value in a specified register; parallel add/subtract

("PADD/PSUB") instructions that add or subtract operands together and place the results in specified registers; a parallel average ("PAVG") instruction that averages two values and places the result in a specified register;
 5 parallel max/min ("PMAX/PMIN") instructions that compare two values and write the greater or lesser value into a specified register; a parallel integer compare ("PCMP") instruction that compares two operands and modifies condition code flags in the parallel status flag
 10 register; and a parallel conditional move ("PCMOV") instruction that compares status flags in the PSW register with the condition code in the PTEST register and, if the flags and code match, moves the operand to a specified register. The instructions and their actions
 15 may be summarized as follows:

	Instruction	Action
	PADD	$B[i] + A[i] \rightarrow B[i]$
	PAVG	$(A[i] + B[i]) >> 1 \rightarrow B[i]$
20	PSUB	$B[i] - A[i] \rightarrow B[i]$
	PABS	$B[i] = A[i] $
	PMIN	If $B[i] > A[i]$ then $B[i] = A[i]$
	PMAX	If $B[i] < A[i]$ then $B[i] = A[i]$
	PCMP	$B[i] - A[i] \rightarrow PSF[i]$
25	PCMOV	If $PTEST = PSF[i]$ then $A[i] \rightarrow B[i]$

As noted above, when the HSIMD bit in the PSW is set to "1," 16-bit, or half-word, operations are used; otherwise, 8-bit, or byte, operations are employed. (The
 30 remainder of this discussion will address the use of 32-bit data words and 16- or 8-bit operations. This limitation is for explanatory purposes only. Other embodiments may use 64- or 128-bit data words and 32- or 64-bit operations, etc.) When the USIMD bit is set to
 35 "1," PMIN and PMAX use unsigned operands. When the NSAT bit is set to "1," the result should not be saturated. The following table shows which instructions are affected when certain PSW bits are set:

-26-

Instruction	SIMD	USIMD	HSIMD	NSAT
PABS	X		X	
PADD	X	X	X	X
PAVG	X	X	X	
PCMOV			X	
PCMP		X	X	
PMAX		X	X	
PMIN		X	X	
PSUB	X	X	X	X

Sample opcodes for the instruction and updated settings in the PSF register following execution of each instruction are shown below:

Instruction	Opcode	Z	S	CY	OV
PADD	111000	X	X	X	x
PAVG	111001	X	X	X	0
PSUB	111010	X	X	X	x
PMIN	111100	X	X	X	x
PMAX	111101	X	X	X	x
PABS	111011	X	0	X	x
PCMP	111110	X	X	X	x
PCMOV	111111				

The OV flag is set to zero after execution of a PAVG instruction because there is never overflow when this instruction is executed. The S flag is cleared to 0 after execution of a PABS instruction. Execution of a PCMOV instruction does not affect PSFs. Other embodiments may, of course, use different opcodes to identify each instruction.

The PAVG instruction may be executed in 8- or 16-bit mode and may operate on signed or unsigned data. The USIMD PSW bit determines whether sign-extension is done before adding the operands. If the USIMD bit is set, the operands are zero-padded by one bit. If USIMD is not set, the operands are sign-extended by one bit. In 16-bit mode, the PAVG operation is as follows:

$$rb[31:16] = (\{(USIMD?0:rb[31]), rb[31:16]\} + \{(USIMD?0:ra[31]), ra[31:16]\})[16:1]$$

-27-

(Here, if the USIMD bit is set, the operand is zero-padded by one bit; otherwise the operand is sign-extended (i.e., bit 31 is repeated).)

rb[15:0] = ({(USIMD?0:rb[15]), rb[15:0]} +

5 {(USIMD?0:ra[15], ra[15:0]))[16:1]

PSFs following execution of a PAVG instruction in 16-bit mode are as follows:

	PSF3	PSF2	PSF1	PSF0
10 Z	(rb[31:16] == 0) ? 1:0	undefined	(rb[15:0] == 0) ? 1:0	undefined
S	rb[31]	undefined	rb[15]	undefined
CY	cout[31]	undefined	Cout[15]	undefined
OV	0	undefined	0	undefined

15 In 8-bit mode, the PAVG operation is as follows:

rb[31:24] = ({(USIMD?0:rb[31]), rb[31:24]} +

{(USIMD?0:ra[31]), ra[31:24]))[8:1]

rb[23:16] = ({(USIMD?0:rb[23]), rb[23:16]} +

{(USIMD?0:ra[23]), ra[23:16]))[8:1]

20 rb[15:8] = ({(USIMD?0:rb[15]), rb[15:8]} +

{(USIMD?0:ra[15]), ra[15:8]))[8:1]

rb[7:0] = ({(USIMD?0:rb[7], rb[7:0]} + {(USIMD?0:ra[7], ra[7:0]))[8:1]

Following execution of the PAVG operation in 8-bit mode,

25 PSFs are as follows:

	PSF3	PSF2	PSF1	PSF0
30 Z	(rb[31:24] == 0) ? 1:0	(rb[23:16] == 0) ? 1:0	(rb[15:8] == 0) ? 1:0	(rb[7:0] == 0) ? 1:0
S	rb[31]	rb[23]	rb[15]	rb[7]
CY	cout[31]	cout[23]	cout[15]	cout[7]
OV	0	0	0	0

“rb” in the tables above refers to the final result of the instruction, not the input operand. The PAVG instruction always rounds down, not towards 0; negative

numbers are rounded down towards negative infinity.
 Execution of the PAVG instruction provides the 8/16 most
 significant bits ("msbs") of the result of a 9/17 bits
 PADD or PSUB operation. Each 8- or 16-bit operation
 5 updates the corresponding status flags in the PSF
 register.

PADD instructions may be executed in either 16-
 or 8-bit mode on signed and unsigned numbers and will
 provide saturation if the NSAT bit is clear. (When the
 10 USIMD bit is "1," the instructions treat the operands as
 unsigned operands. When the USIMD bit is "0," the
 instructions treat the operands as signed operands.) In
 16-bit mode, a PADD instruction operates as follows:
 $rb[31:16] = \text{SATURATE}(rb[31:16] + ra[31:16])$ (rb and ra
 15 are the register addresses)
 $rb[15:0] = \text{SATURATE}(rb[15:0] + ra[15:0])$
 PSFs following execution of a PADD instruction in 16-bit
 mode are as follows:

	PSF3	PSF2	PSF1	PSF0
20 Z	$(rb[31:16] == 0) ? 1:0$	undefined	$(rb[15:0] == 0) ? 1:0$	undefined
S	rb[31]	undefined	rb[15]	undefined
CY	Cout[31]	undefined	cout[15]	undefined
OV	cout[31] XOR cout[30]	undefined	cout[15] XOR cout[14]	undefined

25

The PADD instruction operates in 8-bit mode as
 follows:

$rb[31:24] = \text{SATURATE}(rb[31:24] + ra[31:24])$
 $rb[23:16] = \text{SATURATE}(rb[23:16] + ra[23:16])$
 30 $rb[15:8] = \text{SATURATE}(rb[15:8] + ra[15:8])$
 $rb[7:0] = \text{SATURATE}(rb[7:0] + ra[7:0])$
 PSFs following an 8-bit operation are as follows:

-29-

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:24] == 0) ? 1:0	(rb[23:16] == 0) ? 1:0	(rb[15:8] == 0) ? 1:0	(rb[7:0] == 0) ? 1:0
S	rb[31]	rb[23]	rb[15]	rb[7]
CY	cout[31]	cout[23]	cout[15]	cout[7]
OV	cout[31] XOR cout[30]	cout[23] XOR cout[22]	cout[15] XOR cout[14]	cout[7] XOR cout[6]

The "rb" in the above tables refers to the final result of the instruction, not the input operand. Each 8- or 16-bit operation updates the corresponding status flags in the PSF register.

PSUB instructions may also be executed in 8-bit or 16-bit mode on signed and unsigned numbers and will provide saturation if the NSAT bit is clear. In 16-bit mode, the PSUB instruction operates as follows:

```
rb[31:16] = SATURATE(rb[31:16] - ra[31:16])
rb[15:0] = SATURATE(rb[15:0] - ra[15:0])
```

PSFs after execution of a PSUB instruction in 8-bit mode are as follows:

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:16] == 0) ? 1:0	undefined	(rb[31:16] == 0) ? 1:0	undefined
S	rb[31]	undefined	rb[15]	undefined
CY	cout[31]	undefined	cout[15]	undefined
OV	cout[31] XOR cout[30]	undefined	cout[15] XOR cout[14]	undefined

The PSUB instruction operates in 8-bit mode as follows:

```
rb[31:24] = SATURATE(rb[31:24] - ra[31:24])
rb[23:16] = SATURATE(rb[23:16] - ra[23:16])
rb[15:8] = SATURATE(rb[15:8] - ra[15:8])
rb[7:0] = SATURATE(rb[7:0] - ra[7:0])
```

Following execution of the instruction in 8-bit operation, PSFs are as follows:

-30-

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:24] == 0) ? 1:0	(rb[23:16] == 0) ? 1:0	(rb[15:8] == 0) ? 1:0	(rb[7:0] == 0) ? 1:0
S	rb[31]	rb[23]	rb[15]	rb[7]
CY	cout[31]	cout[23]	cout[15]	cout[7]
OV	cout[31] XOR cout[30]	cout[23] XOR cout[22]	cout[15] XOR cout[14]	cout[7] XOR cout[6]

The "rb" in the above tables refers to the final result of the instruction, not the input operand. Each 8- or 16-bit operation updates the corresponding status flags in the PSF register.

Results may be saturated in both 8- and 16-bit mode PADD and PSUB operations (in both signed and unsigned mode). No saturation occurs for PAVG operations, since the average can never overflow, and consequently OV is always 0.

In 16-bit unsigned mode, saturation for the PADD instruction occurs as follows:

If ((C == 1) && (NSAT == 0)) rb[31:16] = 0xFFFF

(Here, C represents the current carry value that will be written in to the PSF register at the end of the instruction.)

If ((C == 1) && (NSAT == 0)) rb[15:0] = 0xFFFF

In 8-bit unsigned mode, saturation for the PADD instruction occurs as follows:

If ((C == 1) && (NSAT == 0)) rb[31:24] = 0xFF

If ((C == 1) && (NSAT == 0)) rb[23:16] = 0xFF

If ((C == 1) && (NSAT == 0)) rb[15:8] = 0xFF

If ((C == 1) && (NSAT == 0)) rb[7:0] = 0xFF

In 16-bit unsigned mode, saturation for the PSUB instruction occurs as follows:

If ((C == 0) && (NSAT == 0)) rb[31:16] = 0x0000

If ((C == 0) && (NSAT == 0)) rb[15:0] = 0x0000

If 8-bit unsigned mode, saturation for the PSUB instruction occurs as follows:

-31-

```
If ((C == 0) && (NSAT == 0) rb [31:24] = 0x00
```

```
If ((C == 0) && (NSAT == 0) rb [23:16] = 0x00
```

```
If ((C == 0) && (NSAT == 0) rb[15:8] = 0x00
```

```
If ((C == 0) && (NSAT == 0) rb[7:0] = 0x00
```

5 In 16-bit signed mode, saturation occurs as follows:

```
If ((OV == 1) && (NSAT == 0) && (sum[31] == 1)) rb[31:16]
= 0x7FFF
```

```
10 If ((OV == 1) && (NSAT == 0) && (sum[31] == 0)) rb[31:16]
= 0x8000
```

```
If ((OV == 1) && (NSAT == 0) && (sum[15] == 1)) rb[15:0]
= 0x7FFF
```

```
If ((OV == 1) && (NSAT == 0) && (sum[15] == 0)) rb[15:0]
= 0x8000
```

15 In 8-bit signed mode, saturation occurs as follows:

```
If ((OV == 1) && (NSAT == 0) && (sum[31] == 1)) rb[31:24]
= 0x7F
```

```
20 If ((OV == 1) && (NSAT == 0) && (sum[31] == 0)) rb[31:24]
= 0x80
```

```
If ((OV == 1) && (NSAT == 0) && (sum[23] == 1)) rb[23:16]
= 0x7F
```

```
If ((OV == 1) && (NSAT == 0) && (sum[23] == 0)) rb[23:16]
= 0x80
```

```
25 If ((OV == 1) && (NSAT == 0) && (sum[15] == 1)) rb[15:8]
= 0x7F
```

```
If ((OV == 1) && (NSAT == 0) && (sum[15] == 0)) rb[15:8]
= 0x80
```

```
30 If ((OV == 1) && (NSAT == 0) && (sum[7] == 1)) rb[7:0] =
0x7F
```

```
If ((OV == 1) && (NSAT == 0) && (sum[7] == 0)) rb[7:0] =
0x80
```

35 If OV is 1, sum[7] is the inverse of cout[7] because OV = cout[6] XOR cout[7]. Also, if OV = 1, then sum[7] = cout[6]. Therefore, if OV = 1, sum[7] is the

-32-

inverse of cout[7]. As used here, OV represents the current value that will be written into the PSF register at the end of the current cycle.

In Fig. 13, the circuit 318 for 8-bit PADD, PSUB, and PAVG operations can handle both signed and unsigned operands. When the USIMD bit 360 is set to 1, the operands 364, 372 are treated as unsigned operands; when the USIMD bit 360 is set to 0, the operands 364, 372 are treated as signed operands. Bit 7 366, 374 of each input operand 366, 372 is input into a multiplexer 402, 400. If the USIMD bit 360 is set to 0, bit 7 366, 374 is output from each multiplexer 402, 400 as bit 8 386, 378, which is added to the input operand 364, 374 to produce a 9-bit operand 380, 384 which is input into a 9-bit adder 274. If the SIMD bit 360 is set to 1, a 0 362, 370 is output from each multiplexer 402, 400 as bit 8 386, 378, which is added to the input operand 364, 374 to produce the 9-bit operand 380, 384 which is input into the 9-bit adder 274.

Bits 6 (cout[6] 278) and 7 (cout[7] 276) of the result in the 9-bit adder 274 are input to an XOR gate 280 and the result is sent to a first AND gate 286. The other input to AND gate 286 indicates whether a PAVG instruction 282 is being executed. This input 282 is inverted 284 before it is input to the first AND gate 286. If a PAVG instruction 282 is being executed, the input to the AND gate 286 is 0. If both the inputs to the first AND gate 286 from the inverter 284 and the XOR gate 280 are 1, then the PSV OV flag 310 will be set to 1, indicating an overflow result. When a PAVG instruction 282 is executed, the PSF OV flag is always set to 0.

Cout[7] 276 is also input 408, 276 to two multiplexers 412, 404 (the bit is inverted 406 before being input to one of the multiplexers 412) along with

-33-

260 is 1, the Cout[7] value 408, 276 is output 416, 414 to a three-way multiplexer 418. The output 308 from the three-way multiplexer 418 depends on the operation performed by the circuit - PSUB 416, PADD 414, or PAVG 400 (0 is always output if PAVG is performed). This output 308 represents the current overflow of the operation (and will be discussed further below).

The output 320 (sum[8:0]) from the adder 274 is divided into sum[7:0] 290 and sum[8:1] 288 (the average of the two operands) and sent to a multiplexer 292. If a PAVG instruction 282 is being executed, the multiplexer 292 will output 114 the average, or sum[8:1] 288, to a second multiplexer 300; otherwise, sum[7:0] 290 will be output 314 to the second multiplexer 300.

The other input 398 to the second multiplexer 300 represents saturation values. Cout[7] 276 is input to a third and fourth multiplexer 294, 392. If the value of Cout[7] 276 is 0, 0x7F 298 is output 312 from the third multiplexer 294 to a fifth multiplexer 396, while 0x00 is output 394 from the fourth multiplexer 392 to the fifth multiplexer 396. If Cout[7] 276 is 1, 0x80 295 is output 312 from the third multiplexer 294 to the fifth multiplexer 396 while 0xFF is output 394 from the fourth multiplexer 392 to the fifth multiplexer 396. If the USIMD bit 360 is 0, the output from the third multiplexer 294 is output to the second multiplexer 300; if the USIMD bit 360 is 1, the output from the fourth multiplexer 392 is sent to the second multiplexer 300.

A second AND gate 302 is connected to the second multiplexer 300. The inputs to the AND gate 302 are the output 308 from the three-way multiplexer 418 which indicates whether there is overflow in the current operation and a line 324 indicating whether the result should be saturated (if the NSAT bit 306 is set to 1, the result should not be saturated; if it is set to 0, the

-34-

result should be saturated. The NSAT bit 306 is inverted 304 and input 324 to the second AND gate 302.). If there is overflow 308 and if the result should be saturated 324, the output 316 (i.e., the result of the operation) from the second multiplexer 300 is the saturated value 398. Otherwise the result 316 is either sum[7:0] 290 for a PADD or PSUB operation or sum[8:1] 288 for a PAVG operation. A circuit for handling operands of different sizes, for instance 16 bits, works on similar principles.

PMIN and PMAX can operate in 8-bit or 16-bit mode with signed or unsigned data depending on the USIMD bit. In 16-bit mode, PMIN and PMAX instructions are executed as follows:

rb[31:16] = MIN(rb[31:16],ra[31:16])

rb[15:0] = MIN(rb[15:0],ra[15:0])

rb[31:16] = MAX(rb[31:16],ra[31:16])

rb[15:0] = MAX(rb[15:0],ra[15:0])

PSFs are updated as follows following execution of a PMIN or PMAX instruction in 16-bit mode:

20

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:16] == 0) ? 1:0	undefined	(rb[15:0] == 0) ? 1:0	undefined
S	rb[31]	undefined	rb[15]	undefined
CY	cout[31]	undefined	cout[15]	undefined
OV	cout[31] XOR cout[30]	undefined	cout[15] XOR cout[14]	undefined

25

In 8-bit mode, PMIN and PMAX instructions are executed as follows:

rb[31:24] = MIN(rb[31:24],ra[31:24])

rb[23:16] = MIN(rb[23:16],ra[23:16])

rb[15:8] = MIN(rb[15:8],ra[15:8])

rb[7:0] = MIN(rb[7:0],ra[7:0])

rb[31:24] = MAX(rb[31:24],ra[31:24])

rb[23:16] = MAX(rb[23:16],ra[23:16])

rb[15:8] = MAX(rb[15:8],ra[15:8])

35

-35-

rb[7:0] = MAX(rb[7:0],ra[7:0])

Following execution of the PMIN or PMAX instruction in 8-bit mode, the PSFs are as follows:

5

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:24] == 0) ? 1:0	(rb[23:16] == 0) ? 1:0	(rb[15:8] == 0) ? 1:0	(rb[7:0] == 0) ? 1:0
S	rb[31]	rb[23]	rb[15]	rb[7]
CY	cout[31]	cout[23]	cout[15]	cout[7]
OV	cout[31] XOR cout[30]	cout[23] XOR cout[22]	cout[15] XOR cout[14]	cout[7] XOR cout[6]

10

In the above tables, "rb" refers to the final result of the instruction, not the input operand. Each 8- or 16-bit operation updates the corresponding status flags in the PSF register.

15

The PABS instruction may be executed in either 8- or 16-bit mode depending on the HSIMD PSW bit. The NSAT bit in the PSW does not affect the behavior of the PABS instruction. In 16-bit mode, the PABS instruction is executed as follows:

20

rb[31:16] = ABS(ra[31:16])

rb[15:0] = ABS(ra[15:0])

After execution of the PABS instruction in 16-bit mode, the PSFs are updated as follows:

25

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:16] == 0) ? 1:0	undefined	(rb[15:0] == 0) ? 1:0	undefined
S	0	undefined	0	undefined
CY	cout[31]	undefined	cout[15]	undefined
OV	cout[31] XOR cout[30]	undefined	cout[15] XOR cout[14]	undefined

30

In 8-bit mode, the PABS instruction is executed as follows:

rb[31:24] = ABS(ra[31:24])

rb[23:16] = ABS(ra[23:16])

35

-36-

rb[23:16] = ABS(ra[23:16])

rb[15:8] = ABS(ra[15:8])

rb[7:0] = ABS(ra[7:0])

After execution of the PABS instruction in 8-bit mode,

5 the PSFs are updated as follows:

	PSF3	PSF2	PSF1	PSF0
Z	(rb[31:24] == 0) ? 1:0	(rb[23:16] == 0) ? 1:0	(rb[15: 8] == 0) ? 1:0	(rb[7:0] == 0) ? 1:0
S	0	0	0	0
10 CY	Cout[31]	cout[23]	cout[15]	cout[7]
OV	cout[31] XOR cout[30]	cout[23] XOR cout[22]	cout[15] XOR cout[14]	cout[7] XOR cout[6]

In the above tables, "rb" refers to the final result of the instruction, not the input operand. Each 8- or 16-bit operation updates the corresponding status flags in the PSF register.

The flags tables assume the PABS operation results in 0-ra in the adder. Therefore, overflow will only be set in one case, when the input is 0x80. This is the only instance where the true result of the PABS operation cannot be represented in the required number of bits.

The PABS function behaves as follows as shown in Fig. 14. After the PABS instruction is received (block 260), if the sign bit of the input is 1 (block 262), and all the other bits are 0 (block 264), the result is the 1's complement of the input (block 266). If the sign of the input is 1 (block 262), and the other bits are not all 0 (block 264), the result is the 2's complement of the input (block 270). If the sign bit of the input is 0 (block 262), the result is the input (block 268). For example, in 8-bit mode, ABS(0xFF) = 0x01, ABS(0x80) = 0x7F, and ABS(0x01) = 0x01. In 16-bit

Following execution of the instruction, the PSF register is updated (block 272).

The PCMP instruction may be executed in 8- or 16-bit mode on signed or unsigned operands. In executing this instruction, a subtraction is performed without updating the destination register. Instead, the condition code flags in the PSF register are modified. In 16-bit mode, the PCMP operation is as follows:

PSF3 = CMP(rb[31:16],ra[[31:16])

10 PSF1 = CMP(rb[15:0],ra[15:0])

Following execution of a PCMP instruction in 16-bit mode, PSFs are updated as follows:

15		PSF3	PSF2	PSF1	PSF0
	Z	(sum[31:16] == 0) ? 1:0	undefined	(sum[15:0] == 0) ? 1:0	undefined
	S	sum[31]	undefined	sum[15]	undefined
	CY	cout[31]	undefined	cout[15]	undefined
	OV	cout[31] XOR cout[30]	undefined	cout[15] XOR cout[14]	undefined

20 In 8-bit mode, the PCMP operation is as follows:

PSF3 = CMP(rb[31:24],ra[31:24])

PSF2 = CMP(rb[23:16],ra[23:16])

PSF1 = CMP(rb[15:8],ra[15:8])

PSF0 = CMP(rb[7:0],ra[7:0])

25 The PSF register is updated as follows:

30		PSF3	PSF2	PSF1	PSF0
	Z	(sum[31:24] == 0) ? 1:0	(sum[23:16] == 0) ? 1:0	(sum[15:08] == 0) ? 1:0	(sum[07:00] == 0) ? 1:0
	S	sum[31]	sum[23]	sum[15]	sum
	CY	cout[31]	cout[23]	cout[15]	cout[7]
	OV	cout[31] XOR cout[30]	cout[23] XOR cout[22]	cout[15] XOR cout[14]	cout[7] XOR cout[6]

Each 8- or 16-bit operation updates the corresponding status flags in the PSF register.

PCMOV instructions may be executed in either 16- or 8-bit mode. The instructions test the condition code in the PTEST register (discussed above) against the 4 sets of flags in the PSF register. If the specified condition is true, the corresponding 8 or 16 bits is moved. The PCMOV instruction operates in 16-bit mode as follows:

If (PSF3 == cnd(PTEST[3:0])) rb[31:16] = ra[31:16]

If (PSF1 == cnd(PTEST[3:0])) rb[15:0] = ra[15:0]

10 The PCMOV instruction operates in 8-bit mode as follows:

If (PSF3 == cnd(PTEST[3:0])) rb[31:24] = ra[31:24]

If (PSF2 == cnd(PTEST[3:0])) rb[23:16] = ra[23:16]

If (PSF1 == cnd(PTEST[3:0])) rb[15:8] = ra[15:8]

If (PSF0 == cnd(PTEST[3:0])) rb[7:0] = ra[7:0]

15 To illustrate execution of a PCMOV instruction, in Fig. 15, when a PCMOV instruction is received (block 324), 8- or 16-bit mode is specified. If 16-bit mode is indicated (block 326), the PSF3 and PSF1 flags are tested against the condition code in the PTEST register (blocks 20 328, 332). If the specified condition is true, the operand ("ra") associated with the tested PSF is moved to a destination register ("rb"), i.e., ra[31:16] is moved to rb[31:16] (block 330) and ra[15:0] is moved to rb[15:0] (block 334). If a specified condition is not true (blocks 328, 332) or an operand is moved (blocks 25 330, 334), execution of the instruction is finished (block 352).

If 8-bit mode is specified (block 326), the PSF3, PSF2, PSF1, and PSF0 flags are tested against the condition code in the PTEST register (blocks 30 336, 340, 344, 348). If the specified condition is true, the operand associated with the tested PSF is moved to a destination register, i.e., ra[31:24] is moved to rb[31:24] (block 338), ra[23:16] is moved to rb[23:16] (block 342), ra[15:8] is moved to rb[15:8] (block 346), 35

-39-

(block 342), ra[15:8] is moved to rb[15:8] (block 346), and ra[7:0] is moved to rb[7:0] (block 350). If a specified condition is not true (blocks 336, 340, 344, 348) or an operand is moved (blocks 338, 342, 346, 350),
 5 execution of the instruction is finished (block 354).

The PCMOV instruction allows decisions on multiple data streams to be made in one cycle, for example, clipping in image processing. Suppose 8 x 8 mode is specified and the following transformation of
 10 each of the 4 8-bit results in register ("R") 0 is desired:

If $x < -30$ then $0 \rightarrow x$

If $-30 \leq x \leq +30$ then $c \rightarrow x$, where c is some constant

If $30 < x$ then $255 \rightarrow x$

15 The above may be achieved in 4 cycles, with the result in R1, as shown below. Suppose

PTEST = JG

R1 = c, c, c, c

R2 = 0, 0, 0, 0

20 R3 = -30, -30, -30, -30

R4 = 30, 30, 30, 30

R5 = 255, 255, 255, 255

The following instructions are issued:

PCMP R0, R3

25 PCMOV R1, R1

PCMP R4, R0

PCMOV R5, R1

Note that PCMP x,y does $y-x$ and JG jumps if $y>x$.

30

DMA Transfer

In Fig. 16, a processor (such as Cradle's CT3400, a multiprocessor digital signal processor) contains a programmable I/O system 410. The programmable
 35 I/O system features an I/O "quad" 434 (a processing core within the processor) connected to the processor's global bus 424. Data can be transferred to and from the quad

434 to the global bus 424 via a global bus interface ("GBI") 432 and a quad interface logic unit 30. The I/O quad 434 has boot ROM 478, an instruction cache 476, a counter timer engine and semaphores 484, data memory 412, a data bus 414, an instruction bus 490, a processing element ("PE") 428, and a memory transfer engine ("MTE," used to control I/O transfers in non-scatter/gather and non-DMA configurations) 426, and a bus interface unit ("BIU") 416, which, along with an I/O bus 418, provides communication between the subsystem's 410 data bus 414 and pin groups 420 within the subsystem. Each pin group 420 contains an output driver 482, logic for high speed direct memory access ("DMA") 436 block transfer, and programmed I/O ("PIO") 438 transfer of data. Each of the pins 422 in each pin group 420 (in this embodiment, there are 8 pins per pin group and 16 pin groups; however, in other embodiments, different numbers of pins per pin group and different numbers of pin groups may be employed) may be configured as either a low-speed PIO pin or a high speed DMA I/O pin depending on the attached device (i.e., a DMA pin is for a DMA device). The processor may also have an SDRAM device attached. In this embodiment, 13 pin groups 480 (each group 480 has 8 pins 84; however, in other embodiments different numbers of pins per pin group and different numbers of pin groups may be employed), each with an output driver 494 logic for programmed I/O transfer of data 492 are implemented for SDRAM. Data is transferred to and from the SDRAM device and the global bus 424 via the I/O bus 418, an SDRAM bus 488, the SDRAM control 474, and a GBI 432. (The details of this processor and the programmable I/O system are included here for exemplary purposes only and other embodiments of the invention may have processors using different elements and/or arrangement of elements than described here.)

-41-

In one embodiment, the BIU can support up to 4 DMA channels, or DMA devices. (Other embodiments of the invention may support a different number of DMA channels.) Each DMA channel interfaces to one or more pin groups which functionally implements one I/O device. The BIU provides FIFO data buffering for up to 4 DMA devices, each of which has four FIFOs (Data In, Data Out, Command, and Status) in the BIU. The BIU assembles commands and outgoing data from the local data bus and places the data in the BIU's Data Out FIFO for transport to the DMA device's pin group FIFOs and assembles commands and incoming data from the device's pin group FIFOs and places the incoming data in the BIU's Data In FIFO for transport to the local data bus to be stored in memory. Transport of commands, status, and data between the BIU FIFOs and memory/processors is done either programmatically by the MTE (in non-scatter/gather and non-DMA configurations) or automatically by the BIU DMA engine. The BIU provides FIFO data assembly and buffering on demand, as requested by the MTE and/or the pin groups of the DMA device, and optionally also FIFO data transportation to and from chip memory.

With reference to Fig. 17, the BIU 416 receives and sends data to the DMA logic 436 in each pin group via the I/O bus 418 (in one embodiment, a 32-bit pipelined bus, though this may vary in other embodiments). The BIU 416 initiates all I/O bus transfers and addresses I/O devices on the bus. Data from pins 422 of an I/O device is transferred to a Data In FIFO 452 or from a Data Out FIFO 454 in the DMA logic 436. As noted above, the BIU 416 provides FIFO data buffering for up to 4 DMA devices, each of which has four FIFOs - Data In 444, Data Out 446, Command 440, and Status 442 - in the BIU 416. There are 32-bit lines 472, 468, 464, and 460 between the FIFOs 440, 442, 444, and 446 and the I/O bus 418 (in other

-42-

embodiments, the lines may handle different numbers of bits). The DMA logic 436 also has four FIFOs - Command 448, Status 450, Data In 452, and Data Out 454. There are 8-bit lines between these FIFOs 448, 450, 452, and 454 and the I/O bus 418 (in other embodiments, the lines may handle different numbers of bits). A pin state machine ("PSM") 456 is used to control movement of data between pins 422 of the DMA device and data FIFOs; the PSM 456 controls the other hardware in the pin group, including FIFOs 448, 450, 452, 454, and pins 422.

Each DMA device has 32 addressable registers which are mapped to the I/O quad address space. (In other embodiments, another number of registers may be employed). The format of the DMA register addresses for DMA registers 0-31 496 and DMA registers 32-63 498 are shown in Figs. 18a and 18b, respectively. PEs and MTEs read data from and supply data to their DMA device FIFOs in the BIU through these addresses. A sample table describing the DMA registers as seen from the local bus is included below. The R/W column indicates the read/write capability of the register: RO means read only; RW means read/write capable; WO means write only; and RC means read to clear. The function column indicates whether the register is in the BIU ("BIU") or in the pin groups ("DMA"). In one embodiment, registers in the BIU are 32 bits, though registers may be other sizes in other embodiments.

Table 1

Reg	R/W	Function	Name	Notes
0	RW	BIU	DMA Data FIFO	Read = Data in, Write = Data out (1 byte)
1				Reserved
2	RW	BIU	DMA Data FIFO	Read = Data in, Write = Data out (2 bytes)
3				Reserved
4	RW	BIU	DMA Data FIFO	Read = Data in, Write = Data out (4 bytes)
5				Reserved
6	RW	BIU	DMA Data FIFO	Read = Data in, Write = Data out (8 bytes)
7				Reserved
8	RW	BIU	DMA Command/Status	Read = Status, Write = Command (1,2,4, or 8 bytes)
9	RW	BIU	Data In FIFO Packet Descriptor Table Pointer	For scatter/gather; updated in place
10	RW	BIU	Data Out FIFO Buffer Descriptor Table Pointer	For scatter/gather; updated in place
11	RW	BIU	Data In FIFO Buffer Descriptor Table Pointer	For scatter/gather; updated in place
12	RW	BIU	Data Out FIFO Parameters	
13	RW	BIU	Data in FIFO Parameters	
14	RW	BIU	Command FIFO Parameters	
15	RW	BIU	Status FIFO Parameters	
16	WO	BIU	BIU FIFO Control	Enable, reset for Data In, Data Out, Command, Status FIFOs

	17	RO	BIU	DMA Data In Byte Enables	
5	18:19	RW	DMA	DMA Address Register	18 = Address register, 19 = 2 nd word for 64 bit address
	20:21	RW	DMA	DMA DC 0	Data Constant 0
	22:23	RW	DMA	DMA DC 1	Data Constant 1
	24	RW	BIU	Data Out FIFO DMA Address	For interrupt, DMA, scatter/gather; updated in place
	25	RW	BIU	Data Out FIFO Buffer; End Interrupt Address	For scatter/gather;
10	26	RW	BIU	Data In FIFO DMA Address	For interrupt, DMA, scatter/gather; updated in place
	27	RW	BIU	Data In FIFO Buffer-End Interrupt Address	For scatter/gather;
	28:29	RW	BIU	Command FIFO Address	Command FIFO Addresses for Interrupt of DMA, error
	30:31	RW	BIU	Status FIFO Address	Status FIFO Addresses for Interrupt or DMA, error
	32	RW	BIU	Data In FIFO Packet Descriptor Table Tail	For scatter/gather;
15	33	RC	BIU	Data In FIFO Packet Descriptor Status	For scatter/gather;
	34	RW	BIU	Data In FIFO Packet-End interrupt address	For scatter/gather;

The BIU translates I/O register addresses between the local data bus and the I/O bus. This mapping of DMA register addresses as seen by the pin group is shown in Table 2 below. An "X" in the BIU column indicates these registers are in the BIU, not the pin groups. When the control, status, and data FIFOs are accessed from the local data bus, the FIFOs in the BIU are communicated with. The BIU communicates with the pin group FIFOs using some of these same addresses on the I/O bus; specifically, it uses DMA register addresses 0, 2, 4, and 6 as the address of the data FIFOs and register address 8 for the control/status FIFOs.

Table 2

Type	Reg	R/W	BIU	Function	Notes
DMA	0	RW	X	DMA Data FIFO	Read = Data in, Write = Data out (1 byte)
DMA	1			Reserved	unused
DMA	2	RW	X	DMA Data FIFO	Read = Data in, Write = Data out (2 bytes)
DMA	3			Reserved	
DMA	4	RW	X	DMA Data FIFO	Read = Data in, Write = Data out (4 bytes)
DMA	5			Reserved	Unused
DMA	6	RW	X	DMA Data FIFO	Read = Data in, Write = Data out (8 bytes)
DMA	7			reserved	unused
DMA	8	RW		DMA Control/Status FIFO	Read = Status, Write = Cmd (1, 2, or 4 bytes)
DMA	9:11		X	Scatter/Gather Registers	
DMA	12:15		X	FIFO Parameters	In the BIU only
DMA	16			BIU/Pin Group FIFO Control	Controls both BIU and pin group FIFOs
DMA	17			reserved	

DMA	18:19	RW		Address Register	18 = Addrs reg. 19 = 2 nd word for 64 bit
DMA	20:21	RW		DMA DC 0	20 = DC reg. 21 = 2 nd word for 64 bit
DMA	22:23	RW		DMA DC 1	22 = DC reg. 23 = 2 nd word for 64 bit
DMA	24:34		X	FIFO and scatter/gather	In the BIU only

5

In Fig. 19, the DMA logic 436 is shown in greater detail. The DMA logic 436 consists of a PSM 456, four FIFOs 452, 448, 450, and 454 (each of which has an associated FIFO request counter), an address counter 518, byte parity generate and detect units 510, 508, FIFO update logic 506, DMA request logic 504, and a DMA address decoder 502. As noted above, the PSM 456 controls movement of data between the pins and the data FIFOs 452, 454 (by providing sequence, handshake, and timing control for DMA transfers, decoding commands from the MTE, and generating transfer commands to its associated I/O FIFO). The PSM 456 communicates with a PE or MTE in the quad through the command FIFO 448 (from which the PSM 456 receives commands) and status FIFO 450 (to which the PSM 456 sends status updates). The FIFO update logic 506 fills the command FIFO 448 and data out FIFO 454 and empties the status FIFO 459 and data in FIFO 452. The FIFOs 452, 448, 450, and 454 work with the larger FIFOs in the BIU. When a DMA FIFO has data for the BIU or needs data from the BIU, the FIFO update logic 506 submits a request to the DMA request bus 500, which is associated with the I/O bus 18 (in some embodiments, the DMA request bus 500 and the I/O bus 418 are collocated). The DMA request bus 500 goes to the BIU, where the service request is translated into a data transfer between the DMA logic FIFO and the corresponding Data In FIFO in the BIU. When the DMA request logic 504

10

15

20

25

30

receives a PIO interrupt signal 516 or if one of the FIFO request counters is non-zero, a DMA service request is generated. The DMA logic 436 also has data registers 514, an address counter 518, and an output multiplexer 512.

5 The data registers 514 may be used to supply or receive data constants and the address counter 518 may be used to supply and increment addresses to the pins. The data in and data out FIFOs 452, 454 have serial to parallel and parallel to serial logic, respectively, allowing the DMA
10 logic 436 to accept and supply data in widths of 1, 2, 4, or 8 bits wide under the control of the PSM 456. An output multiplexer 512 delivers data to pins.

To transfer data, FIFOs in each DMA device submit a request, in the form of a DMA service code, via
15 the DMA request logic to the DMA request line in the BIU. (The Data In and Status FIFOs generate requests when they have data for the BIU; the Data Out and Command FIFOs generate requests when they have space available for data from the BIU.) The DMA service code includes the DMA
20 device code and two bits indicating which FIFO is requesting service; the code is decoded by the BIU and selects the appropriate DMA device and read/write transfer type. The requesting device receives service in the form of a bus transfer at its DMA address as long as
25 no quad bus access is pending. Depending on the DMA service code, the bus transfer supplies or accepts data from the DMA device FIFO.

As noted above, the BIU contains four merge FIFOs for each DMA device. Input data and status
30 information sent from the pin groups is aggregated and placed in the BIU FIFOs while output data and commands from the BIU FIFOs is split up and sent to the respective pin groups. In one embodiment, there are sixteen FIFOs implemented in a single dual port RAM. Each FIFO is 8
35 octets of 8 bytes per octet deep, a total of 128 octets

(1 KB) of dual port RAM. (In other embodiments, different numbers of FIFOs and different amounts of RAM may be employed.)

The BIU can operate in one of three different event-driven modes: interrupt mode, DMA mode, and scatter/gather mode, which is an enhanced DMA mode. (In other embodiments of the invention, the BIU may operate in fewer or more modes than described here.) In interrupt mode the BIU issues an interrupt by writing a word to the interrupt address of the FIFO register when a block is ready for transfer to or from the data bus. The command and status FIFOs generally operate in interrupt mode. In automatic DMA mode, the BIU automatically transfers data to or from memory to or from the location defined by the BIU FIFO transfer address when a block is ready for transfer. Scatter/gather mode employs automatic buffer switching during DMA transfer of data. The mode in which the BIU operates is determined by a DMA FIFO parameters register. An example of the BIU FIFO parameters register format used in one embodiment is shown below in Table 3 (in other embodiments, other formats may be used).

Table 3

Bits	No	Type	Function	Comments
31	1	RW	Error Interrupt Enable	
30	1	RW	Interrupt/DMA Enable	
29:28	2	RW	DMA device width:1/2/4 bytes	Bytes per transfer to/from I/O bus
27:25	3	RW	Big Endian (BE) code	For endian swap on transfer
24:23	2	RW	Byte position=LSB position on I/O bus	Byte rotate by 0-3 bytes
22:21	2		reserved	
20	1	RW	Count Mode: 0=absolute, 1=differential	
19	1	RW	Mode: 1=DMA, 0=Interrupt	DMA=automatic DMA to memory

5

18	1	RO	Read End	Read End byte has left BIU FIFO
17	1	RW	Increment Address per transfer	Increment address on DMA xfer
16	1	RW	64 Bit	64-bit DMA device - data FIFOs only
15:14	2	RW	Block size/ Watermark	1/2/4 or 0/2/4/6 octets
13:7	7	RO	I/O bus Counter / reserved	Function depends on bit 20
6:0	7	RO	D Bus Counter / FIFO count	Function depends on bit 20

Scatter/gather is an advanced method of
 10 operating a BIU channel in DMA mode. It is available for Data In and Data Out FIFOs, not command and status FIFOs.

The BIU uses a table of buffer descriptors in scatter/gather mode for each Data In and Data Out FIFO. A table of packet descriptors for each Data In FIFO
 15 (i.e., each incoming scatter/gather DMA channel) is optionally created, for instance, for use with packet-based I/O interfaces. These tables may be located anywhere; in one embodiment, they are located in the I/O Quad data memory. Each table contains a sequence of
 20 entries; in one embodiment, the table of buffer descriptors is up to 128 entries long and the table of packet descriptors can have anywhere from 32 to 256 entries. Each entry in the buffer descriptor table consists of a buffer address, a buffer size, and
 25 sequencing information for calculating the next table entry's location. The entries in the buffer descriptor table are linked in a programmable (i.e., not necessarily sequential or circular) order. The non-linear link sequence of entries in the table can be changed
 30 dynamically, allowing reordering of IO data with minimal memory reads/writes, since changing address bits changes

the order in which buffers are used. Each entry in the packet descriptor table consists of a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary, decoupling
5 packet sizes from buffer sizes.

In scatter/gather mode, the BIU hardware performs DMA to/from the current buffer and performs automatic buffer switching. Buffer switching consists of reading the next buffer descriptor table entry from
10 memory, where the descriptor (indicated by the sequencing information in each entry in the buffer descriptor table) is automatically fetched by hardware, and updating the buffer pointer and size from the table entry; thus, the next buffer address is obtained automatically from a
15 table in memory by hardware. A buffer-end interrupt is optionally generated.

Data transfer is most efficient for Data Out FIFOs if the buffer sizes are a multiple of the DMA device width size and the start address is aligned on a
20 width size boundary. For example, if the device is a 16-bit device with 2 data pin groups, the buffer size should be a multiple of 2 bytes and the start address should be on a half-word (2-byte) boundary. If the buffer is not also aligned on a DMA block size boundary, transfers will
25 be inefficient until the address aligns on a DMA block size boundary. For Data In FIFOs, data transfer is most efficient if the programmer ensures the buffer sizes are a multiple of the DMA transfer block size and the start address is aligned on a DMA transfer block size boundary.

30 The following registers (indicated above in Table 1) are employed in scatter/gather mode:

- 9 - Data In FIFO Packet Descriptor Table Pointer
(updated in place)
 - 10 - Data Out FIFO Buffer Descriptor Table Pointer
(updated in place)
 - 5 11 - Data In FIFO Buffer Descriptor Table Pointer
(updated in place)
 - 24 - Data Out FIFO DMA Address (updated in place)
 - 25 - Data Out FIFO Buffer-End Interrupt Address
 - 26 - Data In FIFO DMA Address (updated in place)
 - 10 27 - Data In FIFO Buffer-End Interrupt Address
 - 32 - Data In FIFO Packet Descriptor Table Tail Pointer
 - 33 - Data In FIFO Packet Descriptor Status
 - 34 - Data In FIFO Packet End Interrupt Address
- 15 All of these registers are read/write except for 33,
which is readable and write-to-clear. DMA registers 24
and 26 contain the current transfer address in the buffer
and may be read at any time to monitor the progress of a
transfer. DMA registers 25 and 27 contain interrupt
20 addresses. Registers 10 and 11 contain buffer descriptor
table entry addresses, which point to the next buffer
descriptor to be used. When scatter/gather is enabled,
bit 30 in the FIFO parameters register is undefined. The
layouts of scatter/gather registers 9, 10, 11, 32, and 33
25 as well as buffer descriptors, packet descriptors, and
the packet end interrupt value written upon packet-end
interrupt are shown below in Table 4.

Table 4

5	Reg. No	Type	Reg Name	Bit	Bit Name	Function
	9	BIU	Packet De-scriptor Table Pointer	31:3	Address	Address bit 31:3 of the table start; bits 2:0 are always 0
				2:0	Size	# of Table entries: 0=> Disable packet descriptor generation 1 => # entries is 32 2 => # entries is 64 3 => # entries is 128 4 => #entries is 256 5-7 => reserved
	10	BIU	Data Out FIFO Buffer De-scriptor Table Pointer	31:3	Address	Address bits 31:3 of start of table; bits 2:0 are always 0
				2:1		Reserved
				0	SGE	Scatter/Gather Enable
10	11	BIU	Data In FIFO Buffer De-scriptor Table Pointer	31:3	Address	Address bits 31:3 of start of table; bits 2:0 are always 0
				2	PEI	Interrupt at Packet End; 1 => interrupt
				1		Reserved
				0	SGE	Scatter/Gather Enable
	32	BIU	Packet De-scriptor Table Tail Pointer	31:11		Read/write bits ignored by BIU

-53-

5				10:3		Address bits of last packet descriptor processed
				2:0		Reserved
	33	BIU	Packet De- scriptor Status Register	31:13		Reserved
				12:3	count	count of non- updates of table because it was full; saturates at 0x3ff
				2:0		Reserved
	LSW	MEM	Buffer De- scriptor Table Entry - Word 0	31:0	Address	Buffer Address
	MSW	MEM	Buffer De- scriptor Table Entry - Word 1	31	EOBI	Interrupt at end- of-buffer; 1 => interrupt
				30	EOBT	Terminate of end- of-buffer; 1 => clear SGE bit of reg 10 or 11
				29:28		Reserved
	10				27:21	NEXT
				20		Reserved
				19:0	BSIZE	Buffer Size
LSW		MEM	Packet De- scriptor Table Entry - Word 0	31:0		Packet address
MSW		MEM	Packet De- scriptor Table Entry -	31	PINC	Packet incomplete; 1=> incomplete

-54-

15		Word 1			
			30	PLAST	Last part of incomplete packet; 1 => last part
			29:20		reserved
			19:0	PSIZE	Packet size
		Packet End Interrupt Write Value	31:4		reserved
			3:2	Channel	DMA channel of FIFO causing interrupt
20			1	Direction	0 => Data in FIFO, 1 => Data Out FIFO
			0	Type	0 => End of buffer, 1 => End of packet

With reference to Fig. 20, when scatter/gather is enabled, the scatter/gather registers and buffer descriptor tables are initialized (block 600). (As shown in Table 4, above, scatter/gather is enabled if the SGE bit (bit 0) in register(s) 10 and/or 11 is set to 1; writing 0 to bit 0 in these registers will terminate any scatter/gather operations in progress, forcing scatter/gather systems to idle.) The packet size counter is reset (block 602) and the next (here, the first) buffer descriptor (the address in the buffer descriptor table register) is read (block 604). The DMA address register and buffer sized are updated based on the just-read buffer descriptor (block 606). The address in the buffer descriptor table pointer register is updated based on the just-read buffer descriptor (block 608). The transfer block is then written to memory in a DMA operation (block 610).

Automatic buffer switching takes place when a buffer end event occurs (block 612). A buffer end event occurs when a transfer block is written to/from memory and the current buffer becomes full/empty. If no buffer

end event occurs (block 612), the DMA operation continues (block 610). If a buffer end event occurs (block 612), and data transfer terminates (block 614) (as will be discussed in greater detail below, an end-of-buffer event may cause a scatter/gather operation to terminate), the process stops (block 616). However, if data transfer is not terminated (block 614), automatic buffer switching takes places to allow the DMA operation to continue. A packet descriptor is optionally assembled (block 622).

10 The packet size counter is reset (block 624). A packet descriptor, a packet end interrupt, and a buffer end interrupt (discussed in greater detail below) are optionally written out (blocks 626, 628, and 630); the packet end interrupt and buffer end interrupt allow the

15 controlling processor to initiate packet and buffer processing when a packet is complete or when the buffer is ready without constant status polling. In automatic buffer switching, the BIU reads the next buffer descriptor using the address in the buffer descriptor

20 table pointer register (block 604). The DMA address register and the buffer size are updated based on the just-read buffer descriptor (block 606). The address in the buffer descriptor table pointer register is updated based on the just-read buffer descriptor (block 608) and

25 the transfer block is written to memory (block 610). DMA transfers should have priority over descriptor and interrupt generation. In one embodiment, reading the next buffer descriptor (block 604) may be done ahead of time and updating the DMA address register and buffer

30 size and optionally assembling a packet descriptor may be done in parallel in the cycle following the buffer-end event. In this embodiment, buffer switching will take only one cycle. The time to complete buffer switching is very short since buffer switching, which is done by

35 hardware and is decoupled from software, requires one

local memory access only; in one embodiment, it requires 20 local bus cycles or less to complete. If reading the next buffer descriptor from memory is done ahead of time, buffer switching time can effectively be reduced to 5
5 nsecs.

If a protocol is used where the packet size is not always a multiple of the DMA block transfer size, the BIU needs to be told when an incoming packet has ended in order to force the last few bytes to be written to
10 memory. This is particularly important for incoming packets since packets generally will not be a multiple of DMA block transfer size and therefore the BIU needs to be told when the packet has ended in order to force the last few bytes to be written to memory.

15 In order to get the BIU to boundary align input data, the PSM issues a Clock Data In and Round Up command at the end of a packet. (In some embodiments, the Clock Data In and Round Up Command also gets the BIU to generate packet descriptors and end-of-packet
20 interrupts.) Boundary alignment means that the Data In FIFO DMA address is incremented by the DMA block size regardless of the actual amount of data written to the FIFO when the Round Up command, which is associated with the last byte of a packet, was received. As a result, the
25 next packet that is written out will start on a DMA block size boundary.

A buffer end interrupt causes a 32-bit write to the address specified in DMA register 25 or 27 (depending on whether the FIFO is Data In or Data Out) indicating
30 the interrupt type and DMA channel number. (In this embodiment, the DMA block transfer size is 32 bits; in other embodiments with different block transfer sizes, the bit length of the write is equal to the DMA block transfer size.) When the PE receives a buffer end
35 interrupt it will possibly write new value(s) to the

-57-

table entry whose buffer has just been emptied or filled and possibly process the data in a just-filled buffer.

As noted above, the BIU can optionally write out a packet descriptor to a packet descriptor table upon
5 detecting an end-of-packet event, which may include an end-of-buffer event. An end-of-packet event occurs one of two ways: 1) when the last byte of a packet, indicated by a Round Up command, has been written out to memory; or 2) when the current buffer fills up but the
10 current packet is not finished (i.e., an end-of-buffer event where no Round Up command is associated with the DMA transfer block that fills up the current buffer). The packet descriptors are written to the current location in the packet descriptor table, in one
15 embodiment, an octet-aligned circular buffer in I/O Quad data memory. In one embodiment, the packet descriptor table can be as little as 32 packet descriptors (256 bytes) or as large as 256 (2048 bytes) packet descriptors. The packet descriptor table pointer
20 register (DMA register 9), which points to the location that will be written to when the next packet descriptor is written out, is updated in place and can be polled to determine whether there are any new packet descriptors to be processed. The start address and size of the packet
25 descriptor table are derived from the packet descriptor table pointer register.

The packet descriptor includes a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary. In one
30 embodiment, packet descriptors are octet-aligned and 8 bytes long. An exemplary format is shown in Table 4. PSIZE contains the packet size (if bit 31 is clear) or the size of the current packet fragment (if bit 31 is set, indicating a packet end event was caused by a buffer
35 switch where the last transfer block to fill up the

buffer did not have a Round Up command associated with it and allowing the PE/MTE to detect when a packet is split across buffers). Bit 30 is set if bit 31 was set for the last packet descriptor and this packet descriptor was generated due to a Round Up command. Bits 31 and 30 allow the PE/MTE to determine if the packet is a complete packet (bits 31:30 have the logic values 0,0), the first part of a packet (bits 31:30 have the logic values 1,0), or the last part of a packet that was split across buffers (bits 31:30 have the logic values 0,1). This is useful if one of the packet descriptors for a packet split across two buffers is lost, since it allows the software to quickly determine if parts of a split packet were lost due to packet descriptor table being full (discussed in greater detail below).

A packet end interrupt is optionally issued (depending on the setting in the buffer descriptor table pointer register) after a packet end event due to Round Up occurs. The packet end interrupt causes a 32-bit write to the address specified in DMA register 34, the packet end interrupt address. The value written indicates the interrupt type and the DMA channel number. If packet descriptor generation is enabled (bits 2:0 of the packet descriptor table pointer register are not 0,0,0), and packet end interrupts are enabled, then the packet end interrupt is generated after the packet descriptor has been written out to memory.

In Fig. 21, when the BIU receives a packet with a Round Up command (block 632) and writes the block out to memory, the DMA address is incremented by the DMA block size (block 634), forcing the next packet to start on the DMA block size boundary. When the block associated with the Round Up command is written out to memory, a packet descriptor is optionally written out to memory by hardware (block 636). The packet size counter

is reset (block 638). An optional end-of-packet interrupt is generated (block 640). When the PE receives an end-of-packet interrupt (block 642), the packet descriptor is read from the packet descriptor table
5 (block 644). The just-received packet is then processed (block 646).

The packet descriptor table never overflows due to hardware throttling of packet descriptors. When the table is full, packet descriptors are not written out and
10 the packet is dropped. The table is almost full when the head pointer (the packet descriptor table pointer register) is equal to the packet descriptor table tail pointer register minus 8; indicating there are only two available slots in the table. These slots are never
15 filled and the packet descriptors are dropped to prevent overflow. As noted above, when scatter/gather is enabled, the packet descriptor table tail pointer is initialized to point to the packet descriptor table start address, signaling the table is empty. DMA register 33,
20 the Data In FIFO packet descriptor status register, counts the number of times a packet descriptor is dropped. This register saturates at 0x3FF. Writing to this register, which is a write to clear register, causes the COUNT field to be reset to 0.

25 A buffer end condition (where the current buffer fills up with incoming data or is emptied of outgoing data) may terminate scatter/gather operation. If the EOBT bit (as shown in Table 4) is set, an end-of-buffer event will clear the scatter/gather enable ("SGE")
30 bit, causing hardware to terminate scatter/gather operation without software intervention. Termination of scatter/gather operation may occur when the data stream is finished. (Other operations may continue in scatter/gather mode.)

In some embodiments where wider interfaces are used, such as 16-bit or 32-bit devices, the packet size may not be a multiple of device width. When this occurs, the packet descriptor PSIZE field will include the unused
5 bytes in the last I/O bus transfer because the BIU has no way of knowing how many of the last bytes are valid when it receives a Round Up command.

In one embodiment, a status byte is appended to the end of a packet when the PSM issues one of two
10 commands - Input Status or Input Status and Round Up. The status byte notifies the PE how many of the last "device width" bytes are valid and the PE can then update the packet descriptor or internally use the corrected value. When either of these commands is issued, the 8-
15 bit status from the PSM and pins will be clocked into the pin group's Data In FIFO. If the Input Status and Round Up command is issued, the Round Up bit will be set for that byte of data. When either of these commands is issued, the status generated in the Data In FIFO's pin
20 group is pushed to the pin group's Data In FIFO.

Claims

1. A method for performing parallel integer multiply accumulate operations on packed data comprising:
- 5 a) inputting to a MAC two z-bit words containing a first number of m-bit operands;
- b) multiplying the m-bit operands in a second number of m x m operations to obtain a third number of n-bit products;
- 10 c) adding the content of at least one of the j-bit accumulators to the n-bit product of the corresponding m x m operation to obtain a fourth number of n-bit results;
- d) outputting the fourth number of n-bit results into a fifth number of j-bit accumulators, wherein the fourth number of n-bit results are output in the same cycle; and
- 15 e) accessing at least one accumulator register to retrieve results, wherein the results may be individually retrieved and the specific accumulator cleared after retrieval.
- 20
2. The method of claim 1 further comprising specifying an m x m mode for each cycle.
- 25
3. The method of claim 2 wherein, depending on the mode specified, the input and output are one of the following:
- a) each of the two z-bit words is 32 bits and has four independent 8-bit operands and sixteen 16-bit products are accumulated into sixteen signed 20-bit accumulator registers, wherein each operand is
- 30 independently configured as signed or unsigned;

-62-

b) each of the two z-bit words is 32 bits and has two independent 16-bit operands and four 32-bit products are accumulated into four signed 40-bit accumulator registers, wherein each operand is
5 independently configured as signed or unsigned; or

c) each of the two z-bit words is 32 long and has one 32-bit operand and one 64-bit product is accumulated into a signed 80-bit accumulator register.

10 4. The method of claim 1 further comprising adding together results of at least two of the second number of $m \times m$ operations to assemble a partial product.

15 5. The method of claim 1 further comprising packing two results into a single accumulator register.

6. The method of claim 1 further comprising storing overflow bits in a register.

20 7. The method of claim 1 further comprising globally initializing all accumulators.

25 8. The method of claim 1 further comprising specifying an $m \times m$ mode for each cycle.

9. The method of claim 1 further comprising shifting results.

30 10. The method of claim 1 further comprising saturating results.

11. The method of claim 1 further comprising truncating results.

12. An apparatus for performing parallel integer multiply accumulate operations on packed data comprising:

- a) a first number of multipliers for multiplying two z-bit words inputted into the apparatus,
5 the two z-bit words having a second number of m-bit operands;
- b) a third number of j-bit accumulators for storing results of operations performed by the apparatus, each of the third number of j-bit accumulators coupled to
10 at least one of the first number of multipliers;
- c) a fourth number of adders for combining the result of at least one multiplier and a value stored in one of the third number of j-bit accumulators, each of the fourth number of adders coupled to the at least one
15 of the first number of multipliers; and
- d) a fifth number of n-bit accumulator registers for accessing results of operations performed by the apparatus, each of the fifth number of n-bit accumulator registers coupled to at least one of the
20 third number of j-bit accumulators, wherein the results stored in the n-bit accumulator registers may be individually retrieved and the accumulator corresponding to the accessed register cleared after retrieval.

25 13. The apparatus of claim 12 further comprising means for receiving an input command specifying a mode of operation for the apparatus, wherein the receiving means is coupled to the first number of multipliers.

30 14. The apparatus of claim 12 further comprising means for receiving two o-bit words wherein the receiving means is coupled to the first number of multipliers.

-64-

15. The apparatus of claim 13 wherein, depending on the mode specified, the input and output are one of the following:

- 5 a) each of the two z-bit words is 32 bits and has four independent 8-bit operands and sixteen 16-bit products are accumulated into sixteen signed 20-bit accumulator registers, wherein each operand is independently configured as signed or unsigned;
- 10 b) each of the two z-bit words is 32 bits and has two independent 16-bit operands and four 32-bit products are accumulated into four signed 40-bit accumulator registers, wherein each operand is independently configured as signed or unsigned; or
- 15 c) each of the two z-bit words is 32 bits has one 32-bit operand and one 64-bit product is accumulated into a signed 80-bit accumulator register.

20 16. The apparatus of claim 12 further comprising adding means for adding together results of at least two of the first number of multipliers to assemble a partial product, wherein the adding means is coupled to the first number of multipliers.

25 17. The apparatus of claim 12 further comprising means for globally initializing each of the third number of accumulators.

18. The apparatus of claim 12 further comprising means for shifting results of operations.

30 19. The apparatus of claim 12 further comprising means for saturating results of the operations.

-65-

20. The apparatus of claim 12 further comprising means for storing overflow bits in a register.

21. The apparatus of claim 12 further comprising means
5 for truncating results of operations.

22. The apparatus of claim 12 wherein, depending on the input command, the apparatus operates in one of the following modes:

- 10 a) 8 x 8 mode;
 b) 16 x 16 mode; or
 c) 32 x 32 mode.

15 23. In a processor, a method for performing a parallel conditional move operation comprising:

- a) comparing at least two sets of status indicators which correspond to at least two operands to a corresponding condition code specified in a register to
20 determine whether the condition indicated by the condition code is true for any of the status indicators;
 and

- b) if the condition indicated by the condition code is true for any of the status indicators,
25 moving the corresponding operand to a specified register.

24. The method of claim 23 wherein each of the operands is obtained from a data word having at least two data elements corresponding to operands.

30 25. The method of claim 24 wherein the number of data elements in each data word is identified in an instruction.

26. A processor-readable storage medium storing an instruction that, when executed by a processor, causes the processor to perform a method for performing a parallel condition move operation, the method comprising:

5 a) comparing at least two sets of status indicators which correspond to at least two operands to a corresponding condition code specified in a register to determine whether the condition indicated by the condition code is true for any of the status indicators;
10 and

 b) if the condition indicated by the condition code is true for any of the status indicators, moving the corresponding operand to a specified register.

15 27. The processor-readable storage medium of claim 26 wherein each of the operands is obtained from a data word having at least two data elements corresponding to operands.

20 28. The processor-readable storage medium of claim 27 wherein a number of operands in each data word identified in the instruction is specified in the instruction.

25 29. In a processor, a method for performing a parallel absolute operation comprising:

 a) determining the absolute value of at least two operands by employing one of the following approaches based on the sign bit of each of the at least two operands:

30 i) where the sign bit of an operand is 1 and at least one of the other bits is 1, the absolute value of the operand is the 2's complement of the operand;

-67-

ii) where the sign bit of the operand is 1 and each of the other bits is 0, the absolute value of the operand is the 1's complement of the operand; and

5 iii) where the sign bit of the operand is 0, the absolute value of the operand is the value of the operand, wherein overflow is set only when the operand is 0x80; and

10 b) placing the absolute value of each of the at least two operands in at least two registers specified to receive the absolute value of the at least two operands.

30. The method of claim 29 further comprising updating parallel status flags in each parallel status flags
15 register corresponding to each register specified to receive the absolute value of one of the at least two operands.

31. The method of claim 29 wherein each of the operands
20 is obtained from a data word having at least two data elements corresponding to operands.

32. The method of claim 31 wherein the number of data
25 elements in each data word is identified in an instruction.

33. A processor-readable storage medium storing an instruction that, when executed by a processor, causes the processor to perform a method for performing a
30 parallel absolute operation, the method comprising:

a) determining the absolute value of at least two operands by employing one of the following approaches based on the sign bit of each of the at least two operands:

i) where the sign bit of an operand is 1 and at least one of the other bits is 1, the absolute value of the operand is the 2's complement of the operand;

5 ii) where the sign bit of the operand is 1 and each of the other bits is 0, the absolute value of the operand is the 1's complement of the operand; and

10 iii) where the sign bit of the operand is 0, the absolute value of the operand is the value of the operand, wherein overflow is set only when the operand is 0x80; and

15 b) placing the absolute value of each of the at least two operands in at least two registers specified to receive the absolute value of the at least two operands.

34. The processor-readable storage medium of claim 33 further comprising a parallel absolute instruction that causes a processor to update parallel status flags in
20 each parallel status flags register corresponding to the specified register.

35. The processor-readable storage medium of claim 33 wherein each of the operands is obtained from a data word
25 having at least two data elements corresponding to operands.

36. The processor-readable storage medium of claim 35 wherein a number of operands in each data word identified
30 in the instruction is specified in the instruction.

37. In a processor, a method for saturating a result of a first operation comprising:

- a) adding together two m-bit operands;
- b) outputting the m most significant bits as the result when an average operation is performed, otherwise outputting the m least significant bits as the result, wherein the result of the m least significant bits is saturated if there is overflow and if saturation is enabled; and
- 10 c) placing the result in a specified register.

38. The method of claim 37 wherein the operands are signed.

15

39. The method of claim 37 wherein the operands are unsigned.

40. The method of claim 37 further comprising setting an overflow flag when adding the two operands together results in an overflow.

20

41. The method of claim 37 wherein each of the m-bit operands is obtained from an n-bit data word having at least two m-bit data elements corresponding to m-bit operands.

25

42. The method of claim 41 wherein the number of m-bit data elements in each data word is identified in an instruction.

30

43. The method of claim 37 wherein at least one additional operation is performed in parallel with the first operation.

-70-

44. The method of claim 37 further comprising updating parallel status flags in each parallel status flags register corresponding to each specified register.

5 45. A circuit configured to saturate a result of a first operation comprising:

a) an $m+1$ -bit adder for adding together two m -bit operands and outputting an $m+1$ bit result;

b) coupled to the adder, a first multiplexer
10 for outputting one of the following values:

i) the m least significant bits output by the adder; or

ii) the m most significant bits output by the adder, wherein the m most significant bits is output
15 by the first multiplexer when an average instruction is executed;

c) a second multiplexer coupled to a third multiplexer for outputting a selected saturation value; and

d) coupled to the adder, the third
20 multiplexer for outputting one of the following values:

i) the output from the first multiplexer; or

ii) the output from the second
25 multiplexer, wherein the output from the second multiplexer is output by the third multiplexer when there is overflow and saturation is enabled.

46. The circuit of claim 45 further comprising means for
30 setting an overflow flag, the setting means coupled to the third multiplexer.

47. The circuit of claim 45 further comprising fifth and sixth multiplexer for outputting saturated values.

-71-

48. The circuit of claim 45 wherein the output of the selected saturation value is determined by whether operands are signed or unsigned.

5 49. The circuit of claim 45 wherein the operation is an addition operation.

50. The circuit of claim 45 wherein the operation is a subtraction operation.

10

51. The circuit of claim 45 wherein the operation is an average operation.

52. A method of transferring data between a processor
15 and an attached direct memory access ("DMA") device in scatter/gather mode comprising:

a) maintaining a table of buffer descriptors for determining a start address and size of a next buffer to be used when transferring data when a current buffer
20 counter reaches zero, wherein entries in the table are linked programmatically;

b) transferring data between a current buffer and the device via DMA;

c) automatically switching buffers when the
25 selected buffer counter reaches zero, wherein a next buffer descriptor table entry is read from memory and a new buffer pointer and size is updated based on the table entry; and

d) transferring data between the next buffer
30 and the device via DMA.

53. The method of claim 52 further comprising creating a table of packet descriptors for each incoming DMA channel.

-72-

54. The method of claim 52 wherein buffer switching further includes updating a buffer pointer and size of the next buffer based on the table entry.

5 55. The method of claim 52 further comprising boundary aligning input data.

56. The method of claim 52 further comprising generating an end-of-packet interrupt.

10

57. The method of claim 56 further comprising causing an m-bit write to an address specified in a register, wherein m is a number of bits representing a DMA block transfer size.

15

58. The method of claim 53 further comprising generating a packet descriptor for the table.

59. The method of claim 57 further comprising throttling packet descriptors when the table is full.

20

60. The method of claim 52 further comprising generating a buffer end interrupt.

25 61. The method of claim 60 further comprising causing an m-bit write to an address specified in a register, wherein m is a number of bits representing a DMA block transfer size.

30 62. The method of claim 52 further comprising appending a status byte to the end of a packet.

63. The method of claim 52 further comprising terminating scatter/gather operation after detecting an end-of-buffer condition when a data stream is finished.

35

-73-

64. A processor configured to transfer data between the processor and an attached direct memory access ("DMA") device in scatter/gather mode comprising:

- a) means for connecting the processor to a DMA device;
- b) a plurality of buffers for storing data transferred between the DMA device and the processor, each of the plurality of buffers having a buffer counter;
- c) a buffer descriptor table for each Data In FIFO and each Data Out FIFO of a DMA channel transferring data between the DMA device and the processor, each entry in the buffer descriptor table having a buffer address of one of the plurality of buffers, a buffer size of one of the plurality of buffers, and sequencing information for calculating a start address and size of a next buffer to be used for storing data transferred between the DMA device and the processor when a current buffer counter reaches zero, wherein entries in the buffer descriptor table are made programmatically; and
- d) a bus interface unit for transferring data between the DMA device and the processor, the bus interface unit including hardware for automatically fetching the calculated start address and size of the next buffer to be used to store data transferred between the DMA device and the processor from the buffer descriptor table so the bus interface unit can automatically switch buffers for storing data transferred between the DMA device and the processor when the current buffer counter reaches zero.

30

65. The processor of claim 64 further comprising the bus interface unit including hardware for detecting an end-of-packet condition.

-74-

66. The processor of claim 65 further comprising the bus interface unit including hardware for writing a packet descriptor to a packet descriptor table stored in memory of the processor for each Data In FIFO of the DMA channel transferring data between the DMA device and the processor, each entry in the packet descriptor table having a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary, wherein the packet is descriptor is written after detection of an end-of-packet condition.

67. The processor of claim 65 further comprising the bus interface unit including hardware for boundary aligning input data.

68. The processor of claim 64 further comprising the bus interface unit including hardware for detecting an end-of-buffer condition.

69. The processor of claim 68 further comprising the bus interface unit including hardware for terminating scatter/gather operation after detecting an end-of-buffer condition when a data stream is finished.

70. A method of transferring data between a processor and an attached direct memory access ("DMA") device in scatter/gather mode comprising:

- a) transferring a packet of data from the DMA device to the processor;
- b) detecting an end-of-packet condition; and
- c) writing out a packet descriptor to a table in memory, wherein the packet descriptor includes a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary.

-75-

71. The method of claim 70 further comprising resetting a packet size counter.

5 72. The method of claim 70 further comprising generating an end-of-packet interrupt.

73. The method of claim 72 further comprising initiating packet processing.

10 74. The method of claim 70 further comprising boundary-aligning input data.

75. The method of claim 70 further comprising throttling packet descriptors when the table is full.

15 76. The method of claim 70 further comprising maintaining a table of buffer descriptors for determining a start address and size of a next buffer to be used when transferring data when a current buffer counter reaches
20 zero, wherein entries in the table are linked programmatically.

77. The method of claim 76 further comprising automatically switching buffers when the selected buffer
25 counter reaches zero, wherein a next buffer descriptor table entry is read from memory and a new buffer pointer and size is updated based on the table entry.

78. The method of claim 70 further comprising
30 terminating scatter/gather operation after detecting a end-of-buffer condition when a data stream is finished.

-76-

79. A processor configured to transfer data between the processor and an attached direct memory access ("DMA") device in scatter/gather mode comprising:

- a) means for connecting the processor to a DMA device;
- b) a plurality of buffers for storing data transferred between the DMA device and the processor; and
- c) a bus interface unit for transferring data between the DMA device and the processor, the bus interface unit including hardware for detecting an end-of-packet event and writing a packet descriptor to a packet descriptor table for each Data In FIFO of a DMA channel, each packet descriptor having a packet start address, a packet size, and a bit indicating whether the packet is split across a buffer boundary, wherein packet descriptors are written to the packet descriptor table each time an end-of-packet event is detected.

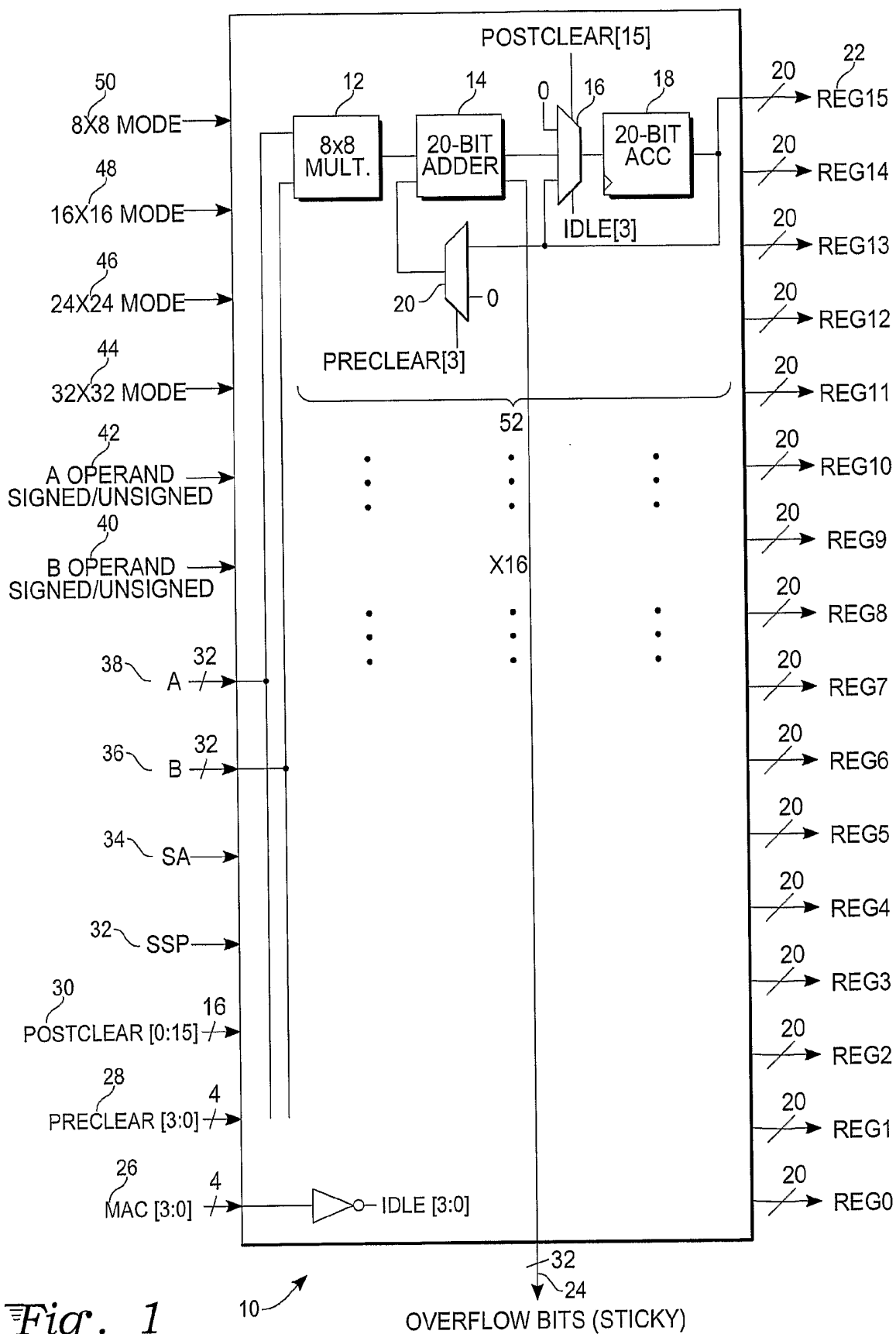
80. The processor of claim 79 further comprising the bus interface unit including hardware for boundary aligning input data.

81. The processor of claim 79 further comprising a buffer descriptor table for each Data In FIFO and each Data Out FIFO of a DMA channel transferring data between the DMA device and the processor, each entry in the buffer descriptor table having a buffer address of one of the plurality of buffers, a buffer size of one of the plurality of buffers, and sequencing information for calculating a start address and size of a next buffer to be used for storing data transferred between the DMA device and the processor when a current buffer counter reaches zero, wherein entries in the buffer descriptor table are made programmatically.

82. The processor of claim 79 further comprising the bus interface unit including hardware for detecting an end-of-buffer condition.

- 5 83. The processor of claim 82 further comprising the bus interface unit including hardware for terminating scatter/gather operation after detecting an end-of-buffer condition when a data stream is finished.

1/21



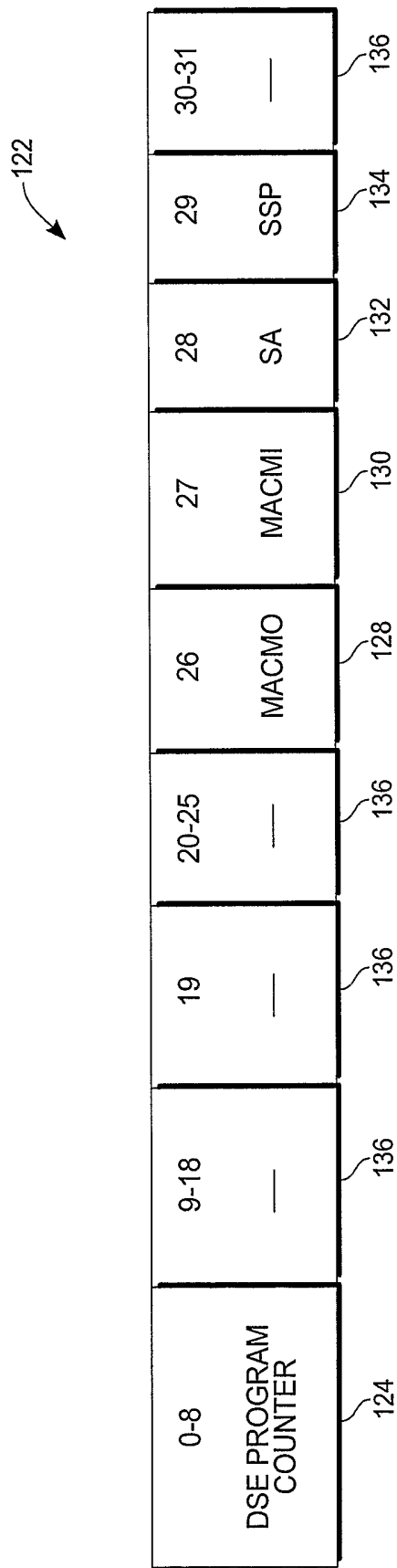


Fig. 2

	OPERANDS	NO. OPERATIONS/CYCLE	RESULTS
8X8 MODE	4 INDEPENDENT 8-BIT OPERANDS/WORD (EACH INDEPENDENTLY CONFIGURED ASSIGNED OR UNSIGNED)	16 8X8 MACS/CYCLE	16 16-BIT PRODUCTS ACCUMULATED INTO 16 SIGNED 20 BIT ACCUMULATOR REGISTERS
16X16 MODE	2 INDEPENDENT 16-BIT OPERANDS/WORD (EACH INDEPENDENTLY CONFIGURED ASSIGNED OR UNSIGNED)	4 16X16 MACS/CYCLE	4 32-BIT PRODUCTS ACCUMULATED INTO 4 SIGNED 40 BIT ACCUMULATOR REGISTERS
32X32 MODE	2 32-BIT OPERANDS/WORD	1 32X32 MAC/CYCLE	1 64-BIT PRODUCT ACCUMULATED INTO 1 OF 4 SIGNED 80 BIT ACCUMULATOR REGISTERS

Fig. 3

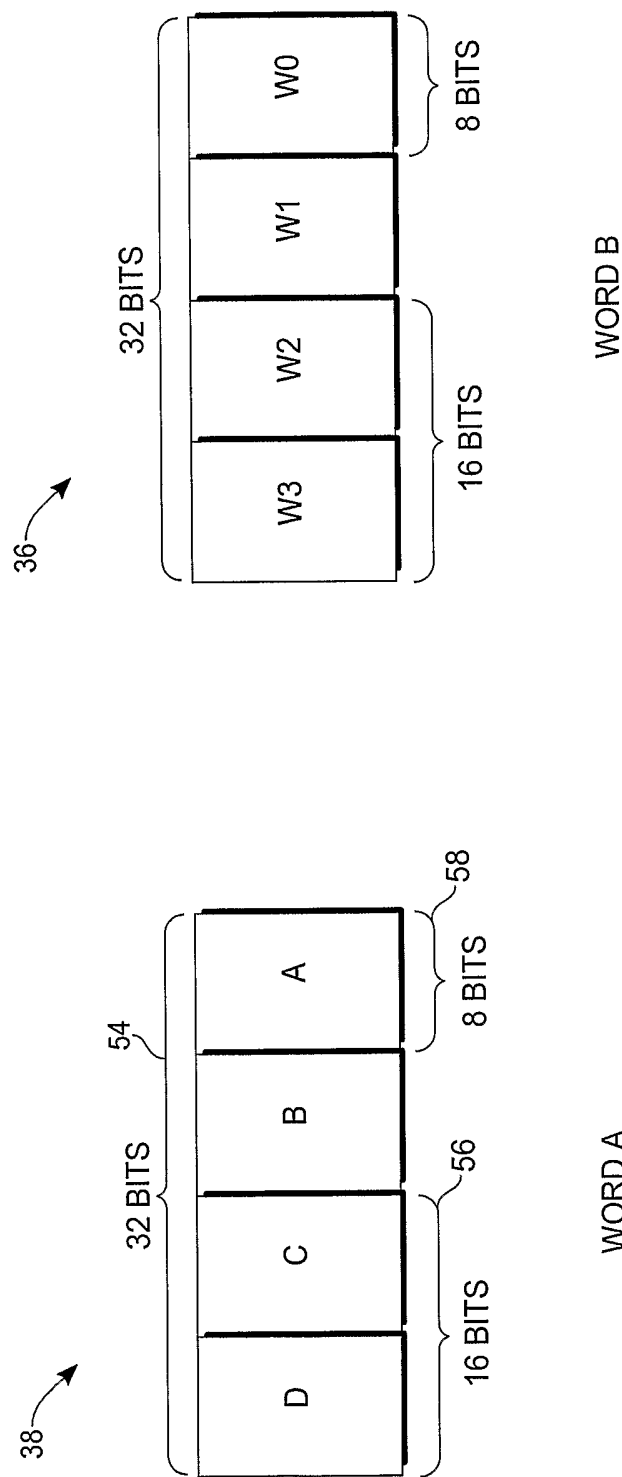


Fig. 4

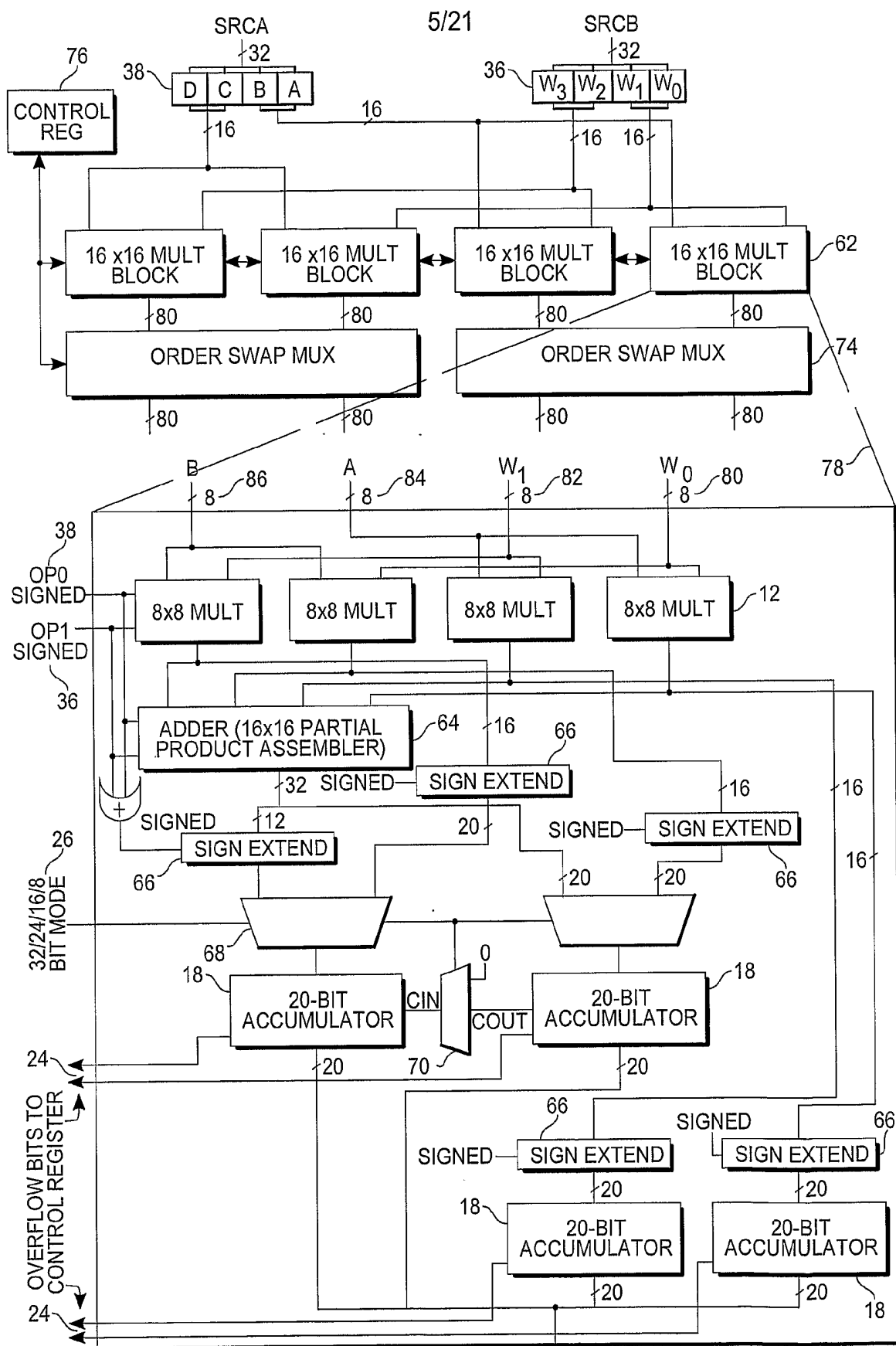


Fig. 5

6/21

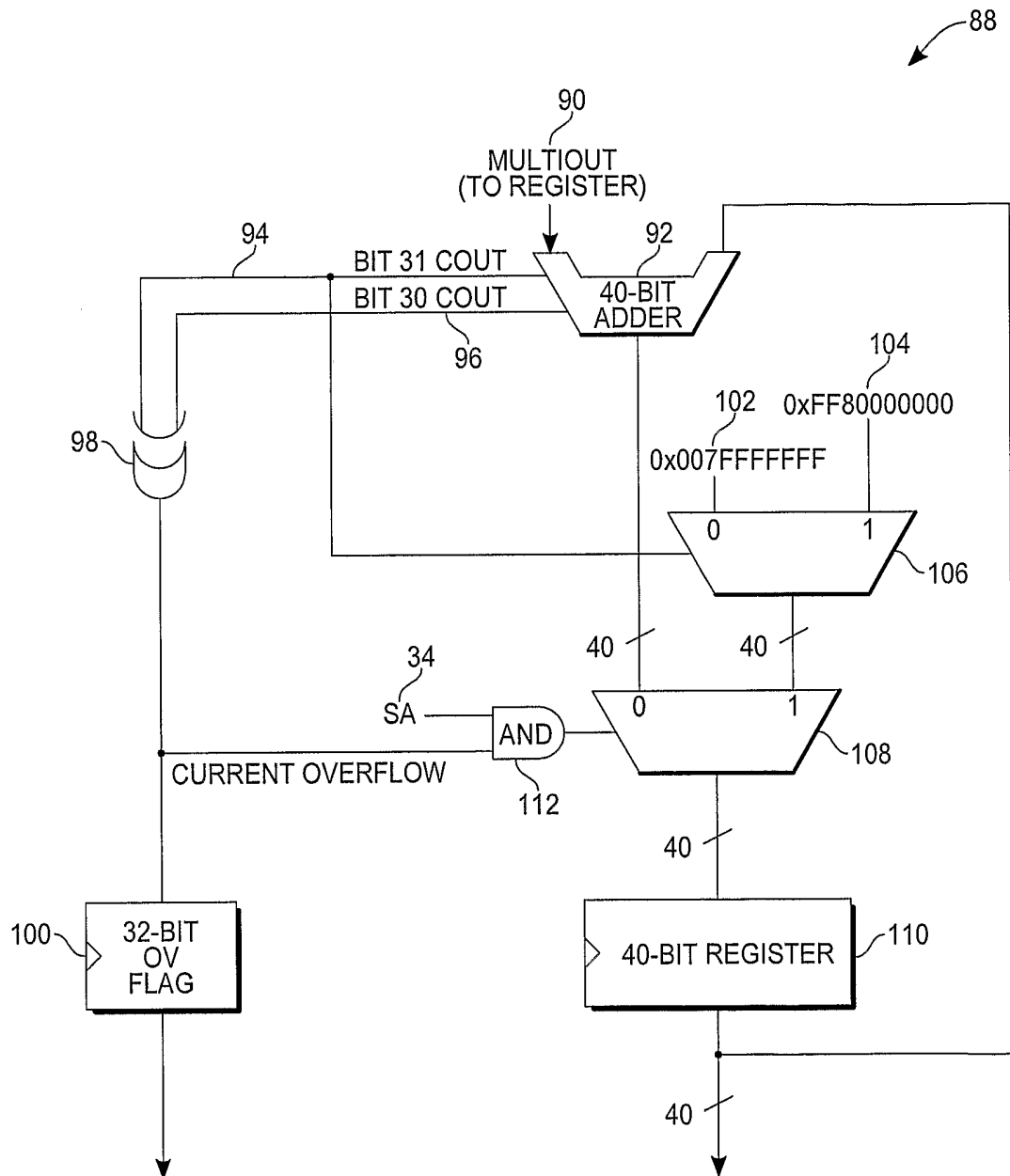


Fig. 6

7/21

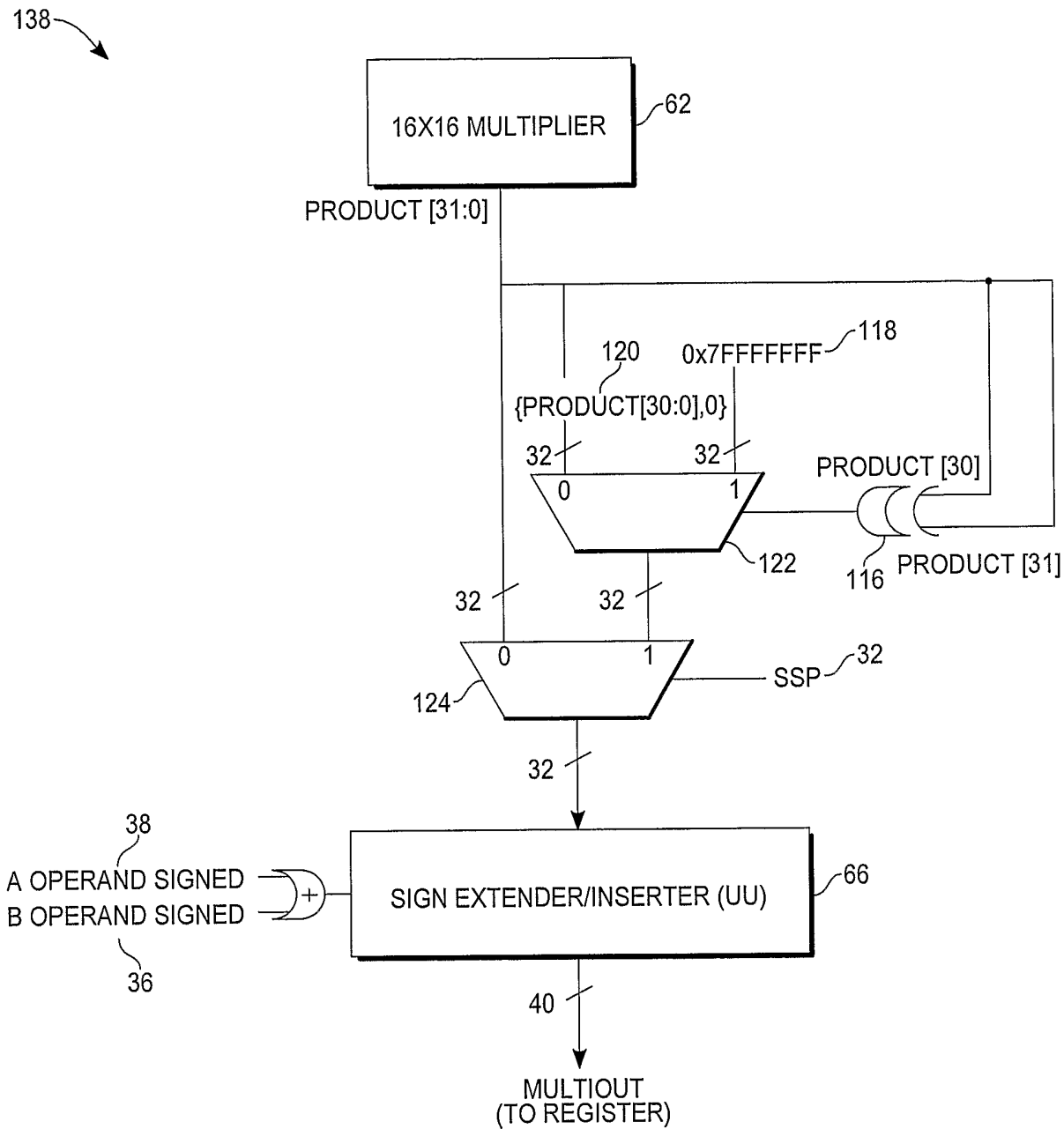


Fig. 7A

8/21

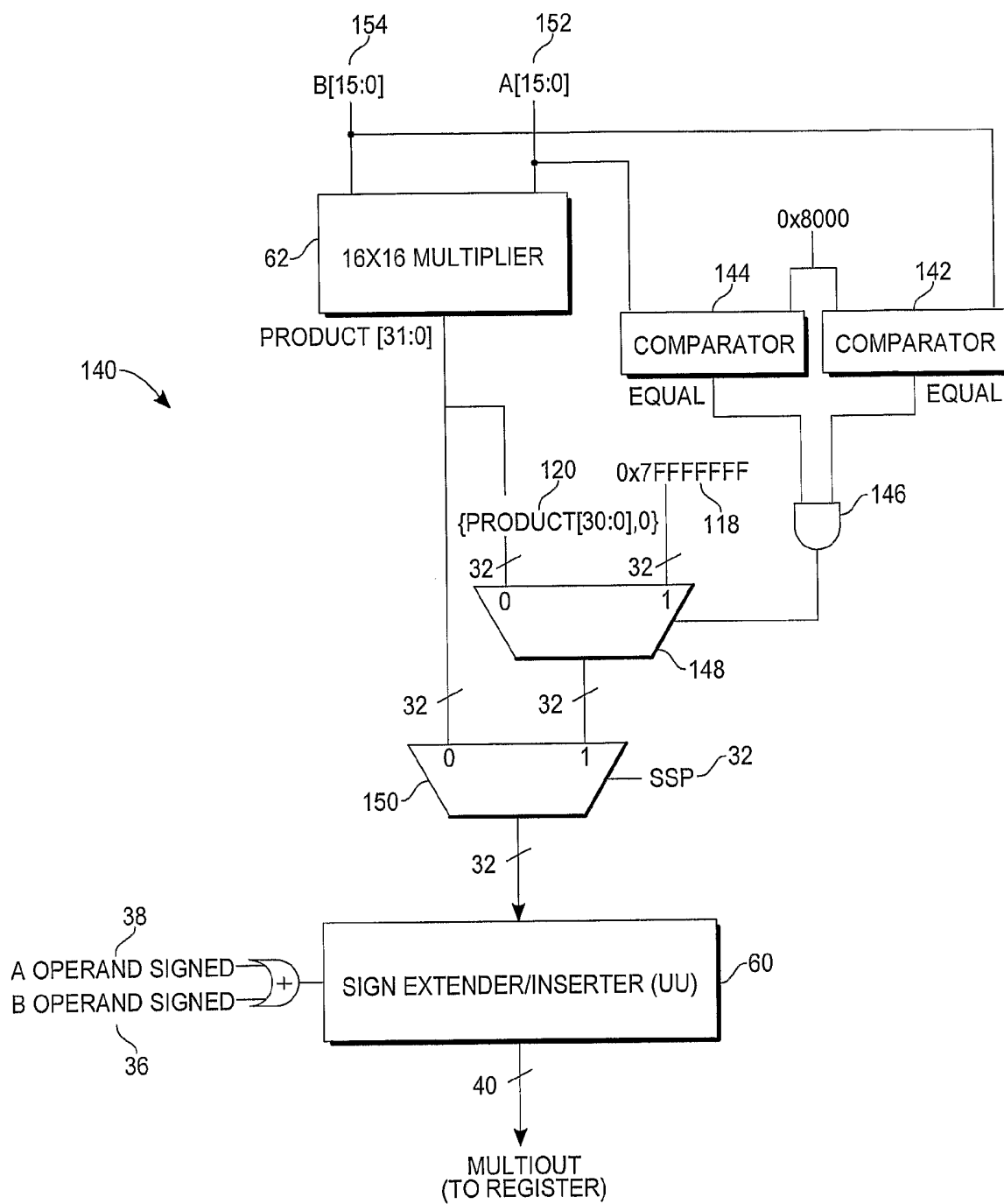


Fig. 7B

9/21

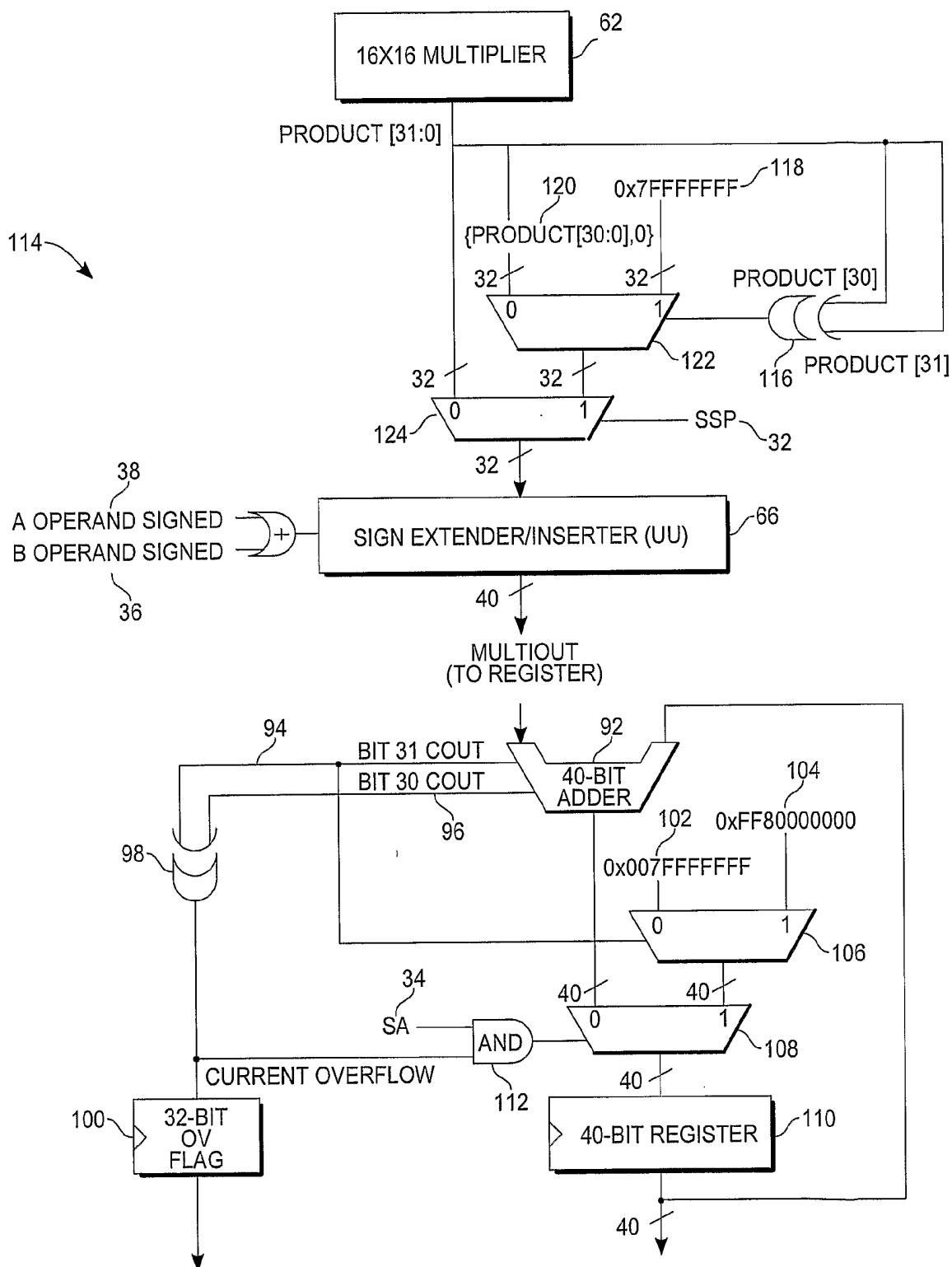


Fig. 8

10/21

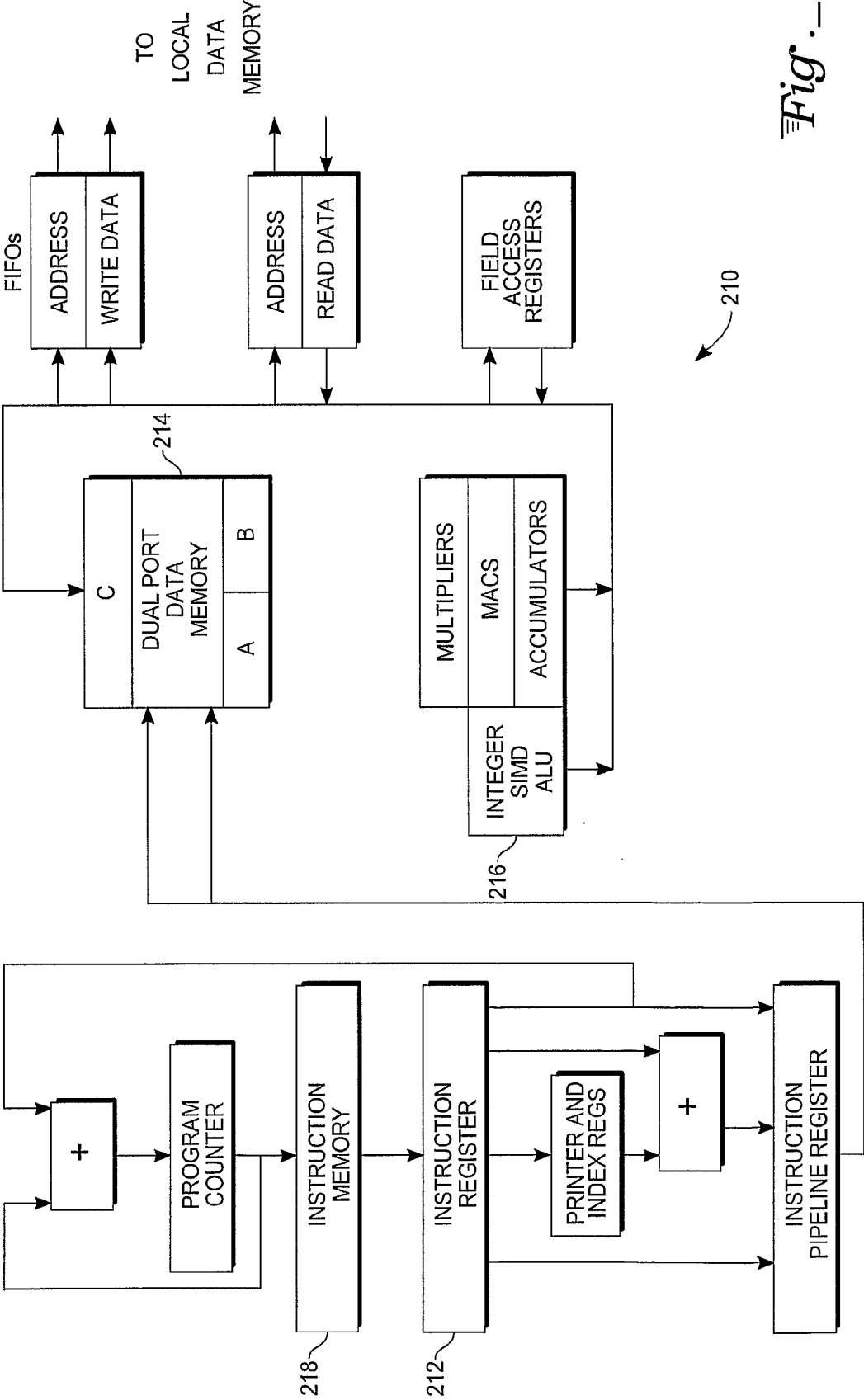


Fig. 9

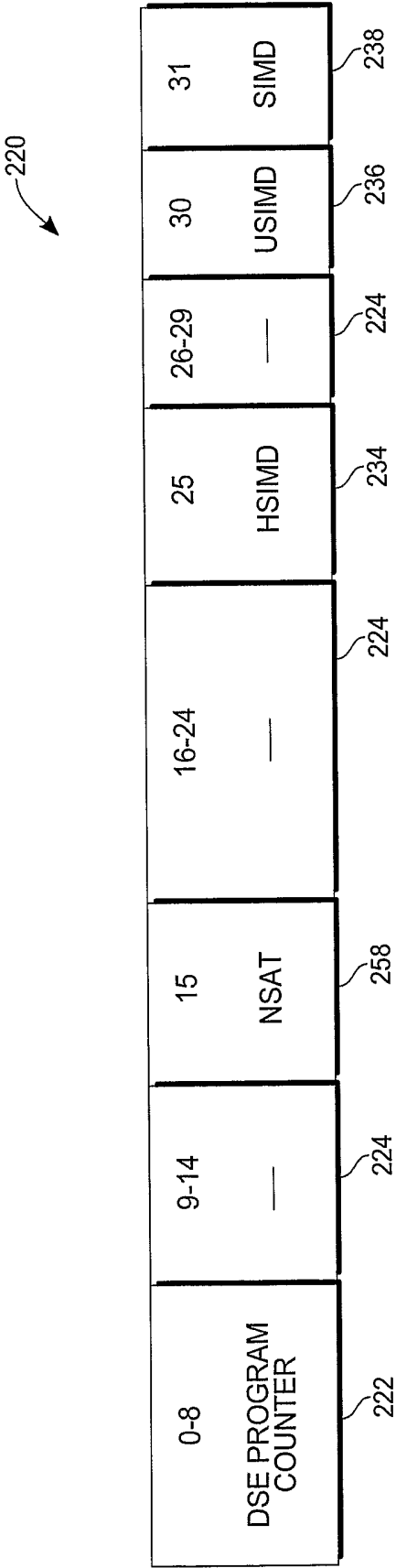


Fig. 10

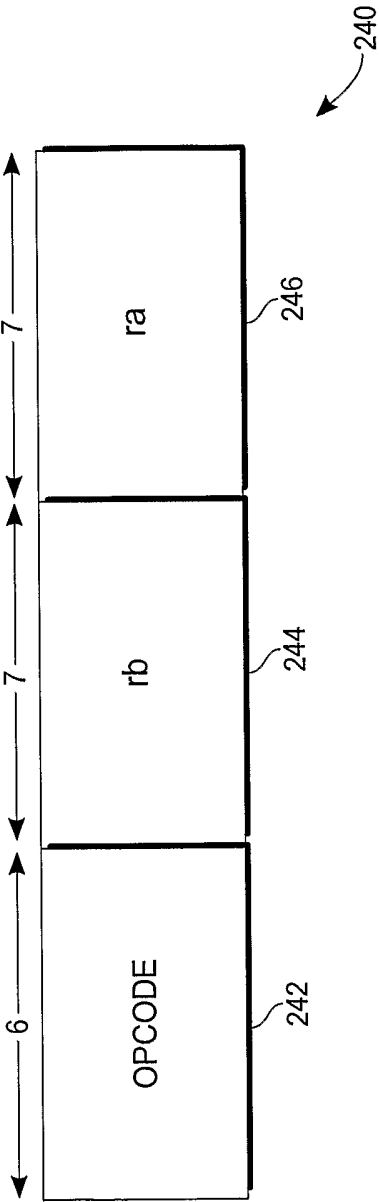


Fig. 11

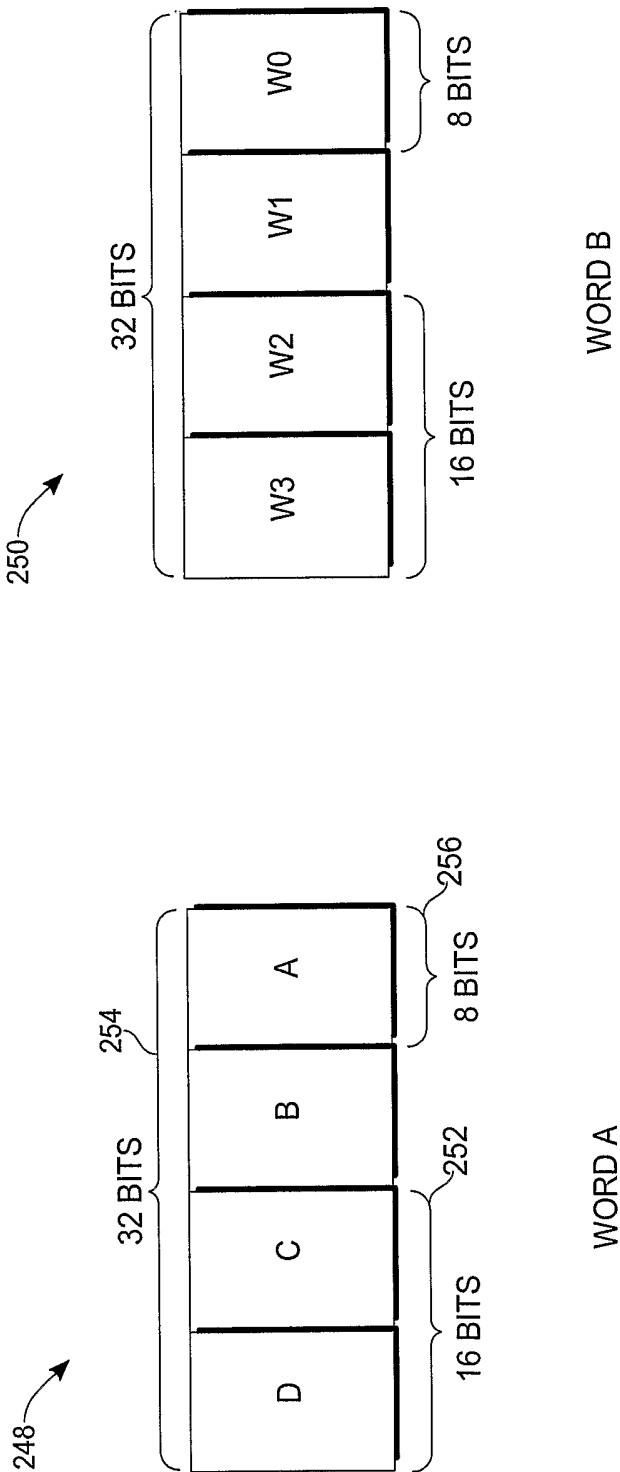


Fig. 12

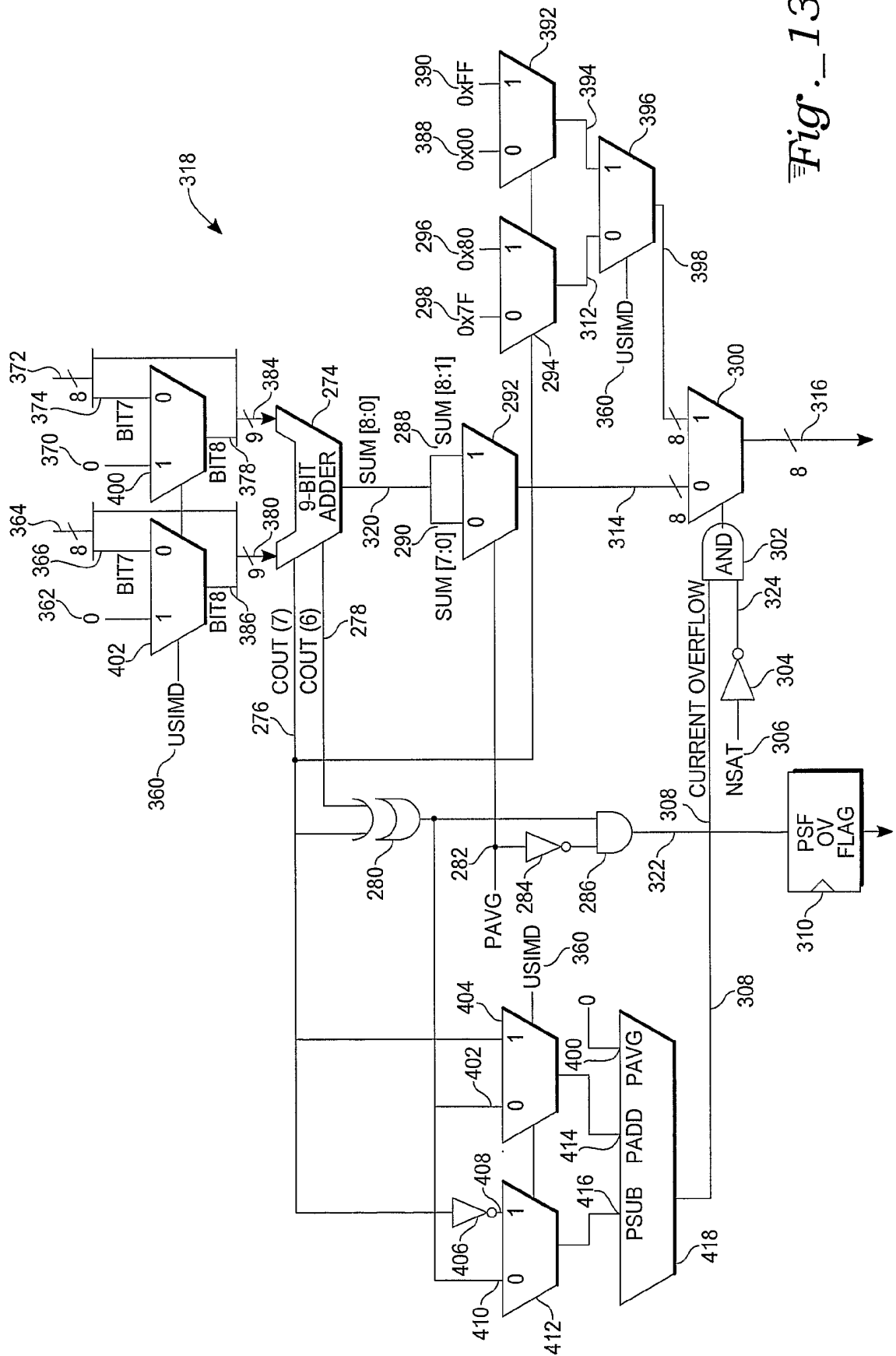
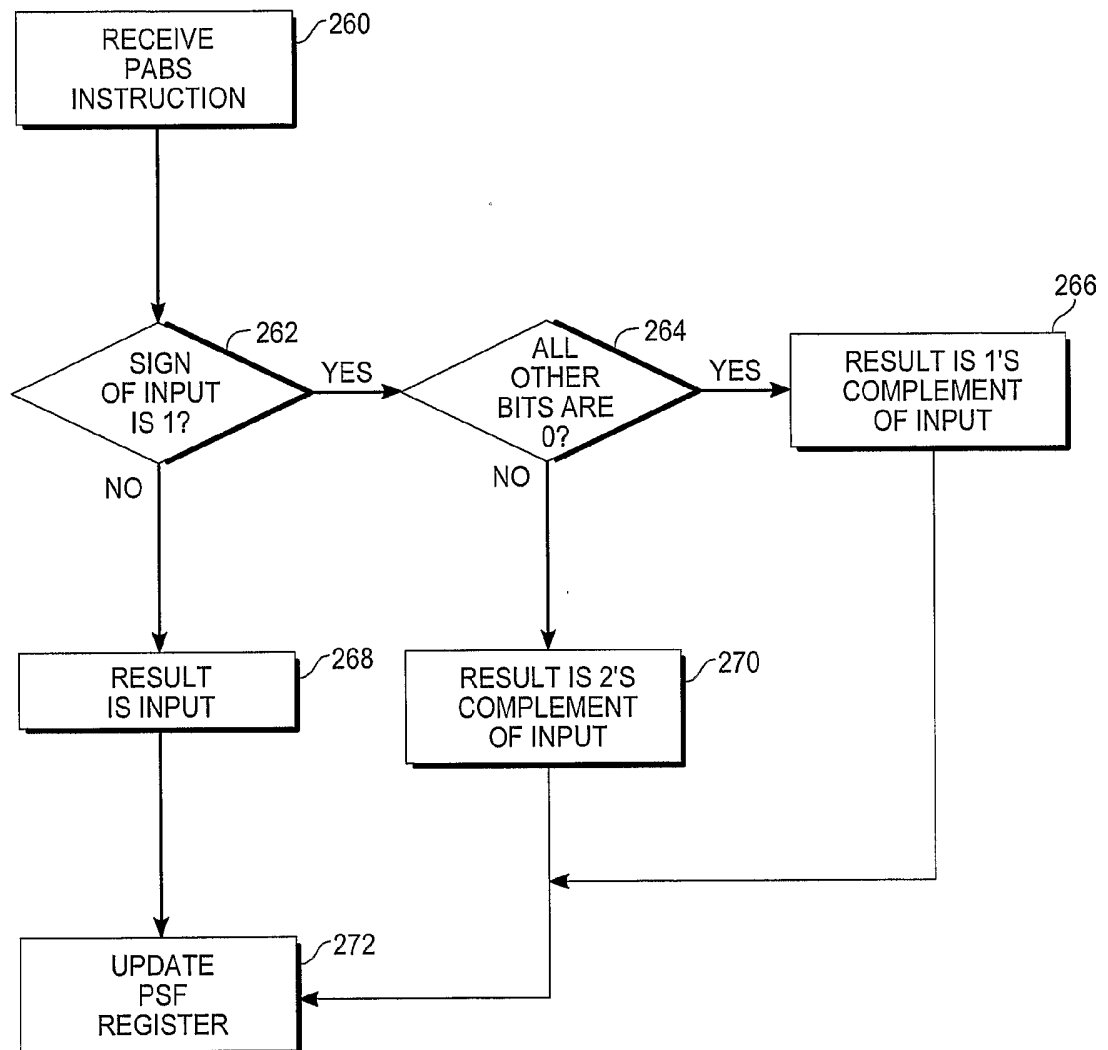


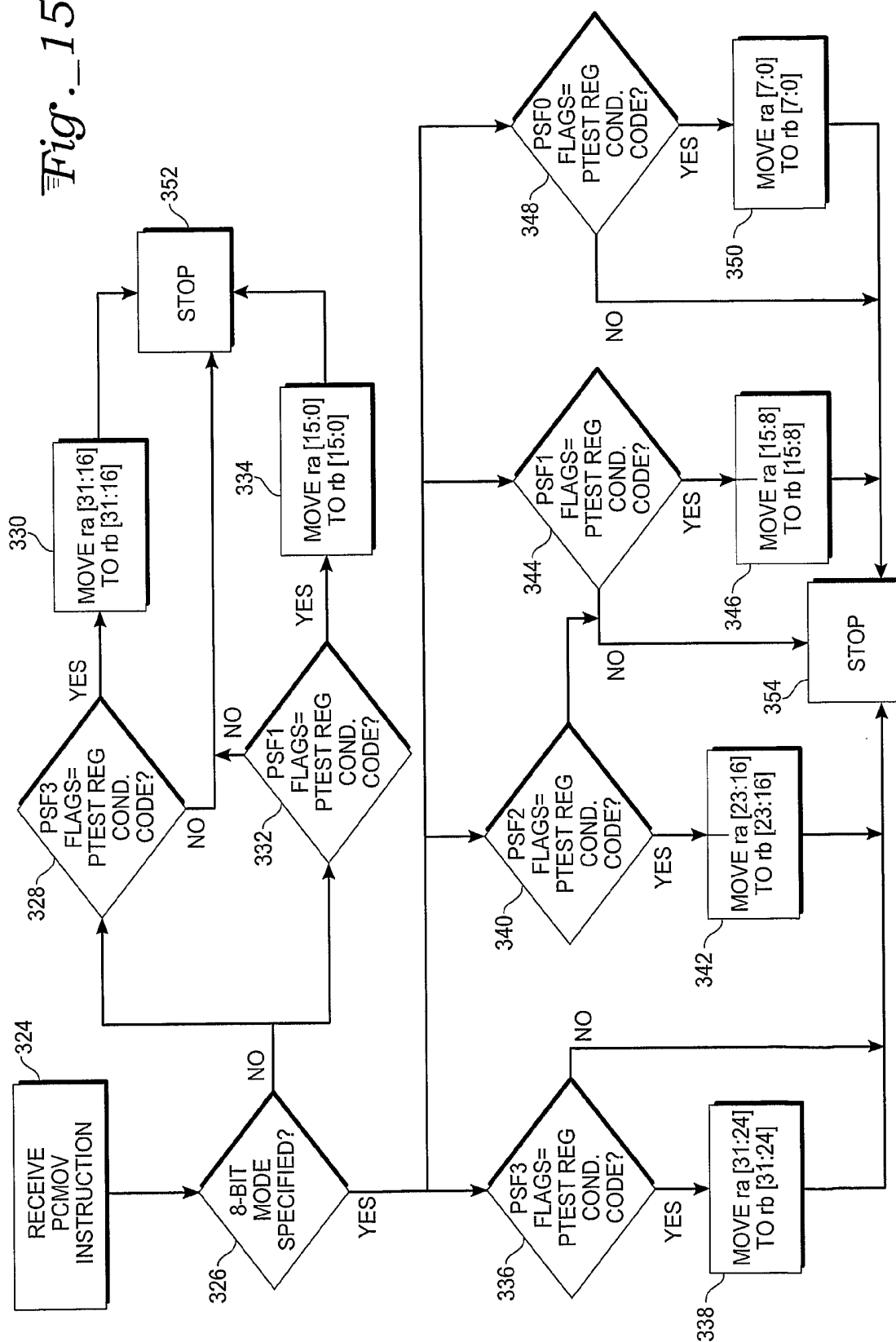
Fig. 13

14/21

*Fig. 14*

15/21

Fig. 15



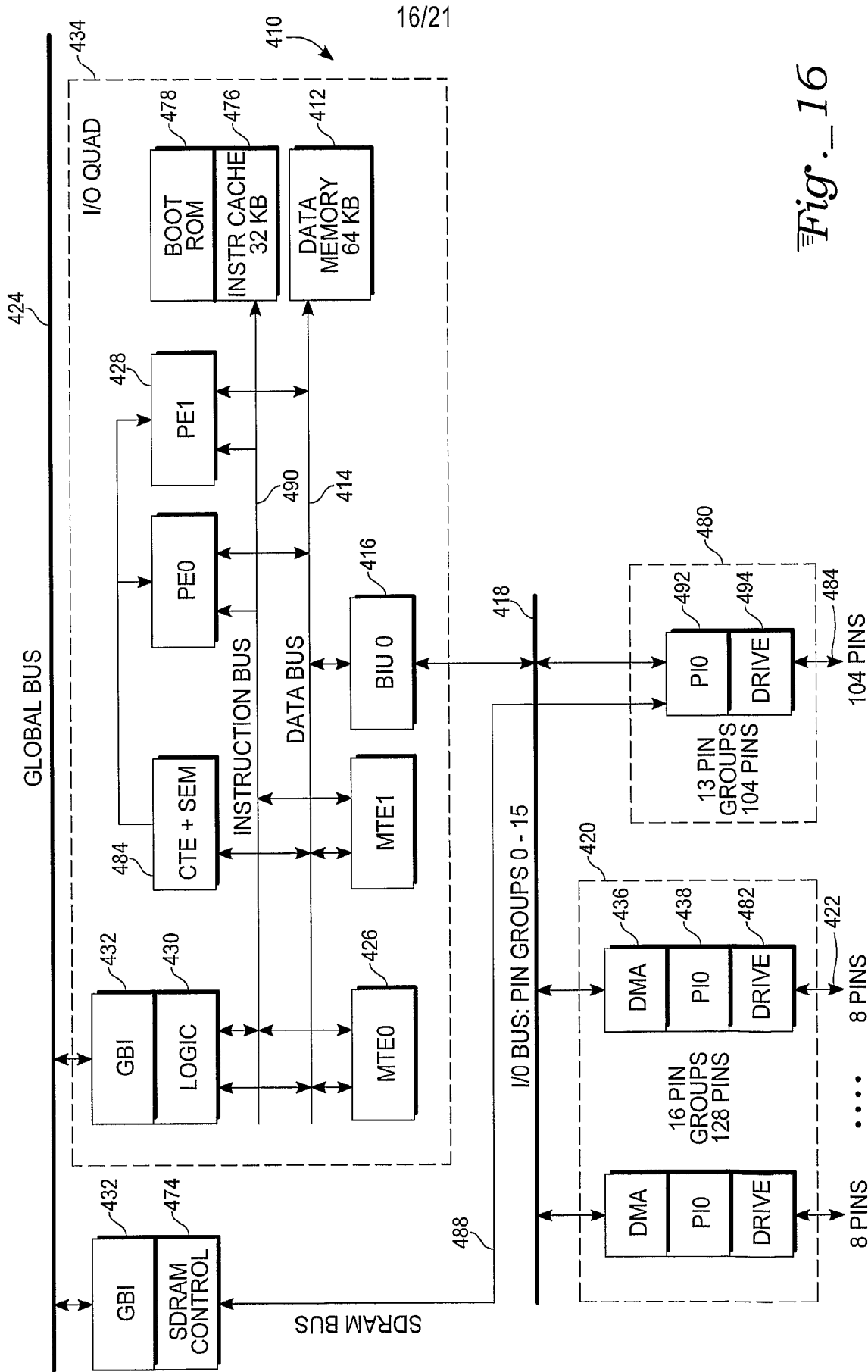
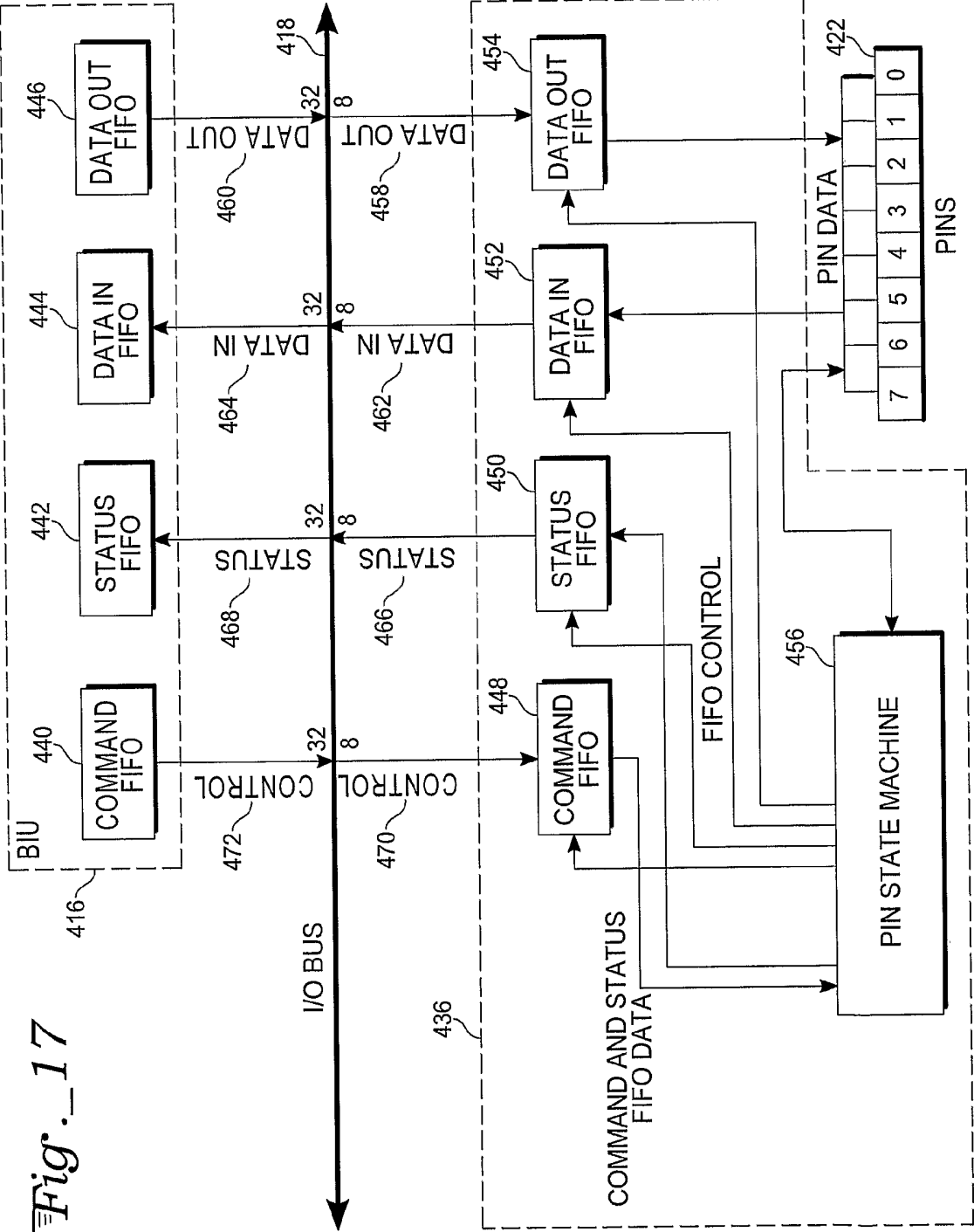


Fig. 16



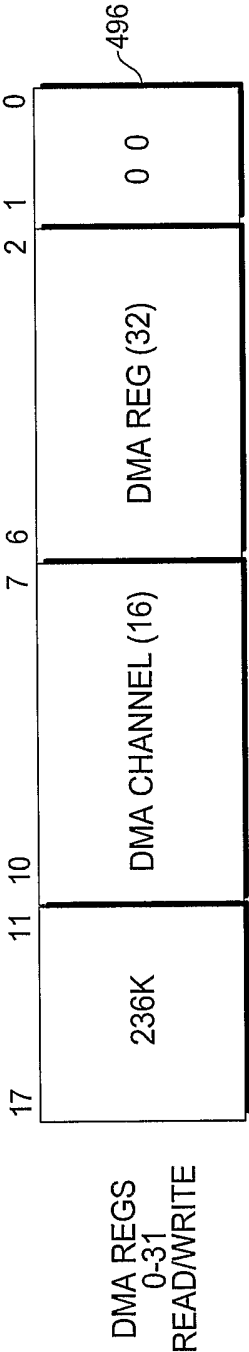


Fig. 18a

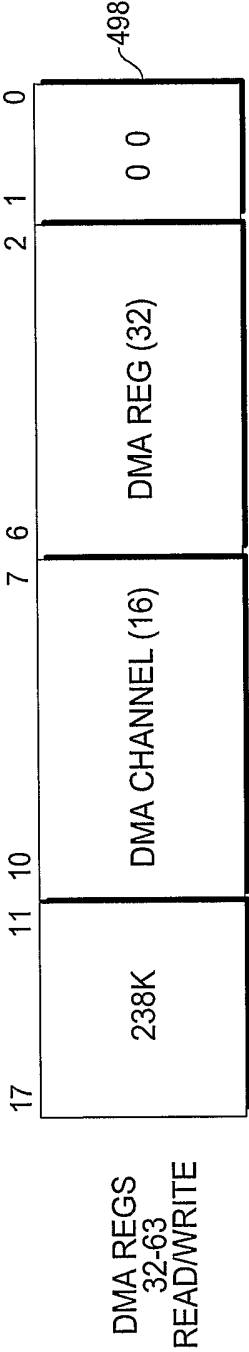


Fig. 18b

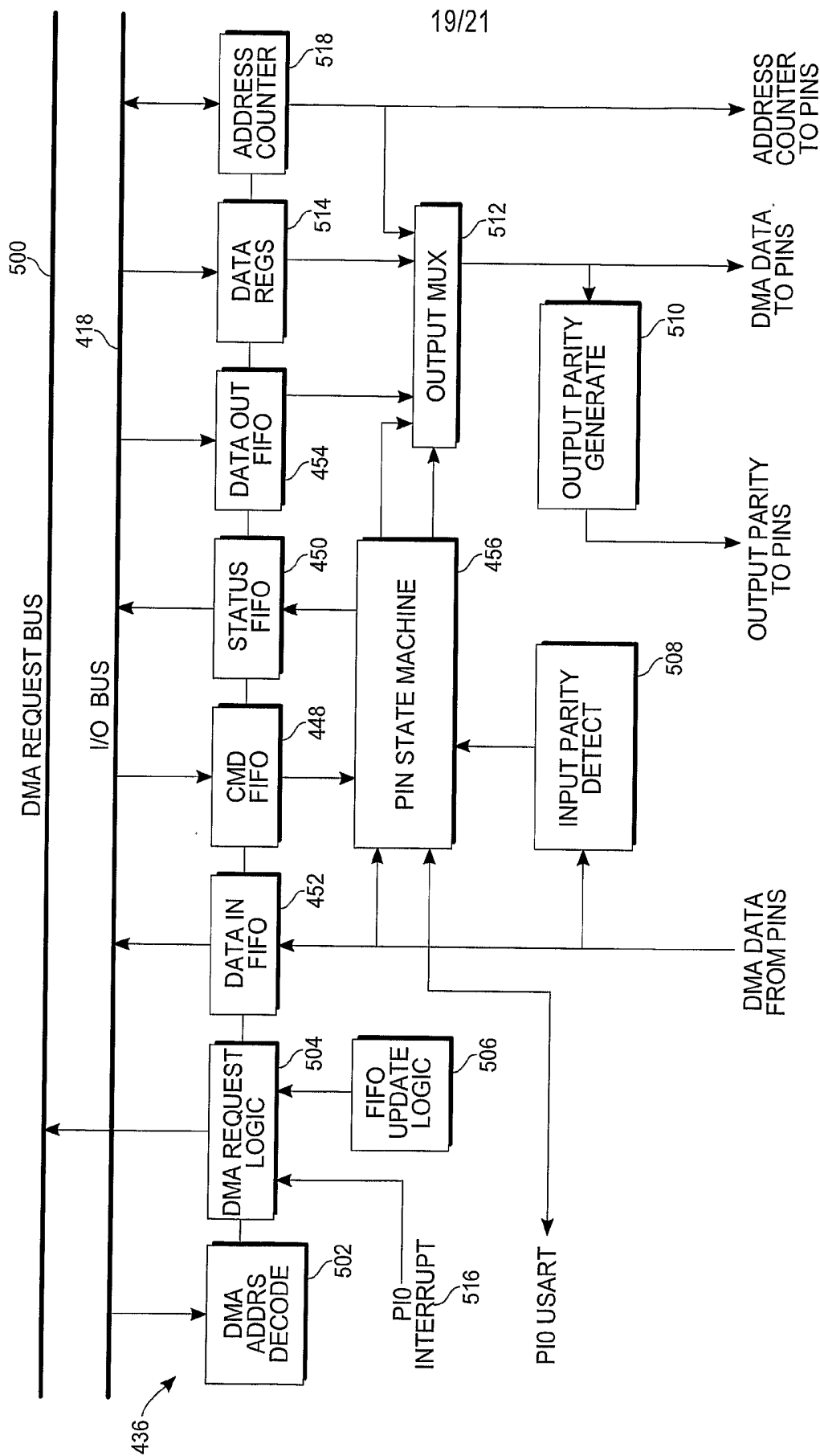


Fig. 19

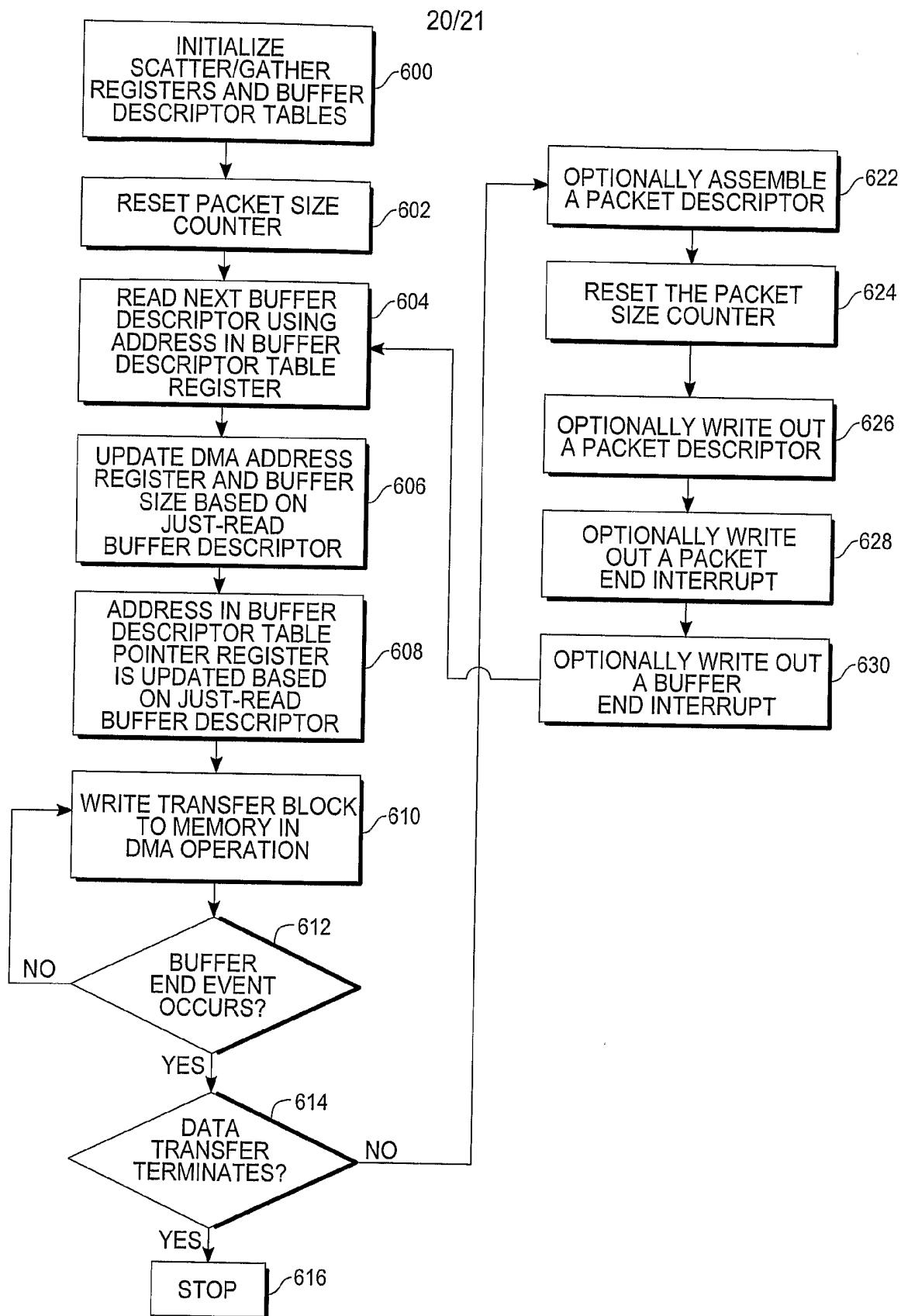
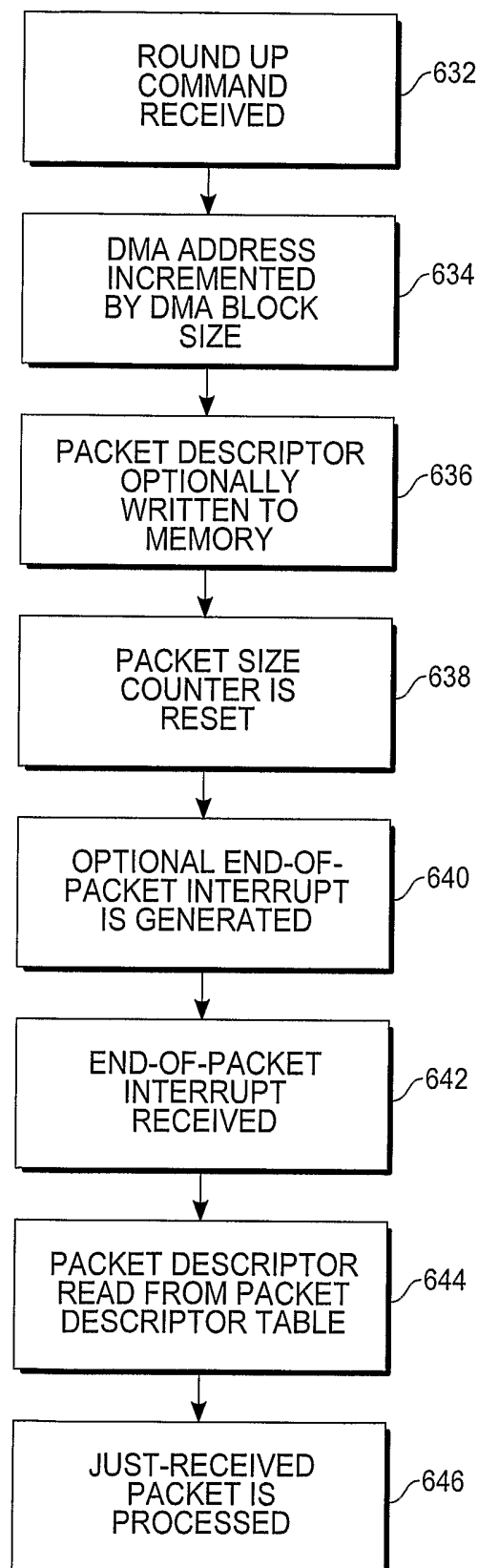


Fig. 20

21/21

*Fig. 21*