US012393422B2

(12) **United States Patent**
Madduri et al.

(10) **Patent No.: US 12,393,422 B2**
(45) **Date of Patent: Aug. 19, 2025**

(54) **APPARATUS AND METHOD FOR VECTOR PACKED SIGNED/UNSIGNED SHIFT, ROUND, AND SATURATE**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Venkateswara Rao Madduri**, Austin, TX (US); **Robert Valentine**, Kiryat (IL); **Mark Charney**, Lexington, MA (US); **Cristina Anderson**, Hillsboro, OR (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 805 days.

(21) Appl. No.: **17/359,552**

(22) Filed: **Jun. 26, 2021**

(65) **Prior Publication Data**

US 2023/0004393 A1 Jan. 5, 2023

(51) **Int. Cl.**
G06F 9/30 (2018.01)
G06F 9/38 (2018.01)

(52) **U.S. Cl.**
CPC ***G06F 9/30149*** (2013.01); ***G06F 9/30036*** (2013.01); ***G06F 9/30038*** (2023.08); ***G06F 9/3836*** (2013.01); ***G06F 9/384*** (2013.01)

(58) **Field of Classification Search**
None
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

9,804,823 B2 10/2017 Lundvall et al.
10,324,719 B2 6/2019 Cowlishaw et al.
(Continued)

FOREIGN PATENT DOCUMENTS

EP 3480710 A1 5/2019
EP 3716047 A1 9/2020
WO 2019/066798 A1 4/2019

OTHER PUBLICATIONS

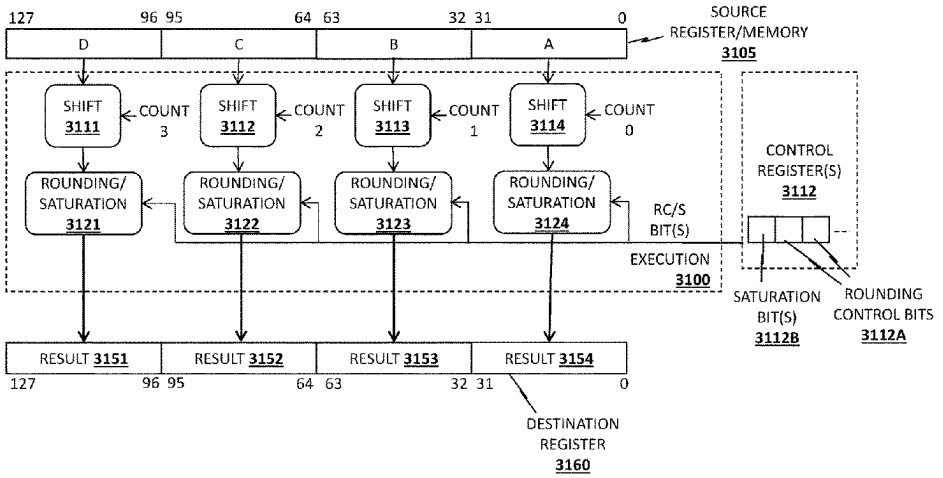
European Search Report and Search Opinion, EP App. No. 22160649.4, Sep. 9, 2022, 7 pages.
(Continued)

Primary Examiner — Eric Coleman
(74) *Attorney, Agent, or Firm* — NICHOLSON DE VOS WEBSTER & ELLIOTT LLP

(57) **ABSTRACT**

Apparatus and method for signed and unsigned shift, round and saturate using different data element values. For example, one embodiment of an apparatus comprises a decoder to decode an instruction having fields for a first packed data source operand to provide a first source data element and a second source data element, a second packed data source operand or immediate to provide a first shift value and a second shift value corresponding to the first source data element and second source data element, respectively, and a packed data destination operand to indicate a first result value and a second result value corresponding to the first source data element and second source data element, and execution circuitry to execute the decoded instruction to: shift the first source data element by an amount based on the first shift value to generate a first shifted data element; shift the second source data element by an amount based on the second shift value to generate a second shifted data element; update a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first and/or second source data elements; round and/or saturate the first and second shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first and second result data elements; and store the first result value and the second result value in a first data element location and a second data element location in a destination register.

21 Claims, 37 Drawing Sheets



(56) **References Cited**

U.S. PATENT DOCUMENTS

10,664,270	B2	5/2020	Ould-Ahmed-Vall et al.	
11,175,891	B2 *	11/2021	Rubanovich	G06F 17/16
11,409,525	B2	8/2022	Heinecke et al.	
11,768,681	B2	9/2023	Heinecke et al.	
2003/0023646	A1 *	1/2003	Lin	G06F 7/762 712/E9.034
2005/0125476	A1	6/2005	Symes et al.	
2006/0271764	A1	11/2006	Nilsson et al.	
2015/0089197	A1 *	3/2015	Gopal	G06F 9/3893 712/207
2016/0098249	A1 *	4/2016	Carlough	G06F 5/012 708/203
2018/0088940	A1 *	3/2018	Rubanovich	G06F 7/483
2018/0095758	A1	4/2018	Dubtsov et al.	
2018/0226970	A1 *	8/2018	Miyadai	H03K 19/017509
2018/0293078	A1 *	10/2018	Gabrielli	G06F 9/30036
2019/0042544	A1	2/2019	Kashyap et al.	
2019/0102168	A1	4/2019	Madduri et al.	
2019/0102174	A1	4/2019	Madduri et al.	
2019/0102191	A1	4/2019	Madduri et al.	
2019/0102193	A1	4/2019	Madduri et al.	
2019/0102195	A1	4/2019	Madduri et al.	
2019/0196828	A1	6/2019	Madduri et al.	
2019/0227797	A1	7/2019	Heinecke et al.	
2020/0192665	A1	6/2020	Ould-Ahmed-Vall et al.	
2021/0124560	A1	4/2021	Liu et al.	

OTHER PUBLICATIONS

Office Action , EP App. No. 22160649.4, Aug. 30, 2023, 4 pages.
European Search Report and Search Opinion, EP App. No. 22166208.3, Sep. 9, 2022, 11 pages.
European Search Report and Search Opinion, EP App. No. 22165820.6, Oct. 4, 2022, 7 pages.
European Search Report and Search Opinion, EP App. No. 22181081.5, Nov. 21, 2022, 9 pages.
Final Office Action, U.S. Appl. No. 17/359,538, Dec. 4, 2024, 11 pages.
Intention to Grant, EP App. No. 22165820.6, Mar. 5, 2024, 6 pages.
Non-Final Office Action, U.S. Appl. No. 17/359,538, Aug. 15, 2024, 11 pages.
Notice of Allowance, U.S. Appl. No. 17/359,522, Sep. 11, 2024, 9 pages.
Notice of Allowance, U.S. Appl. No. 17/359,561, Aug. 22, 2024, 8 pages.

* cited by examiner

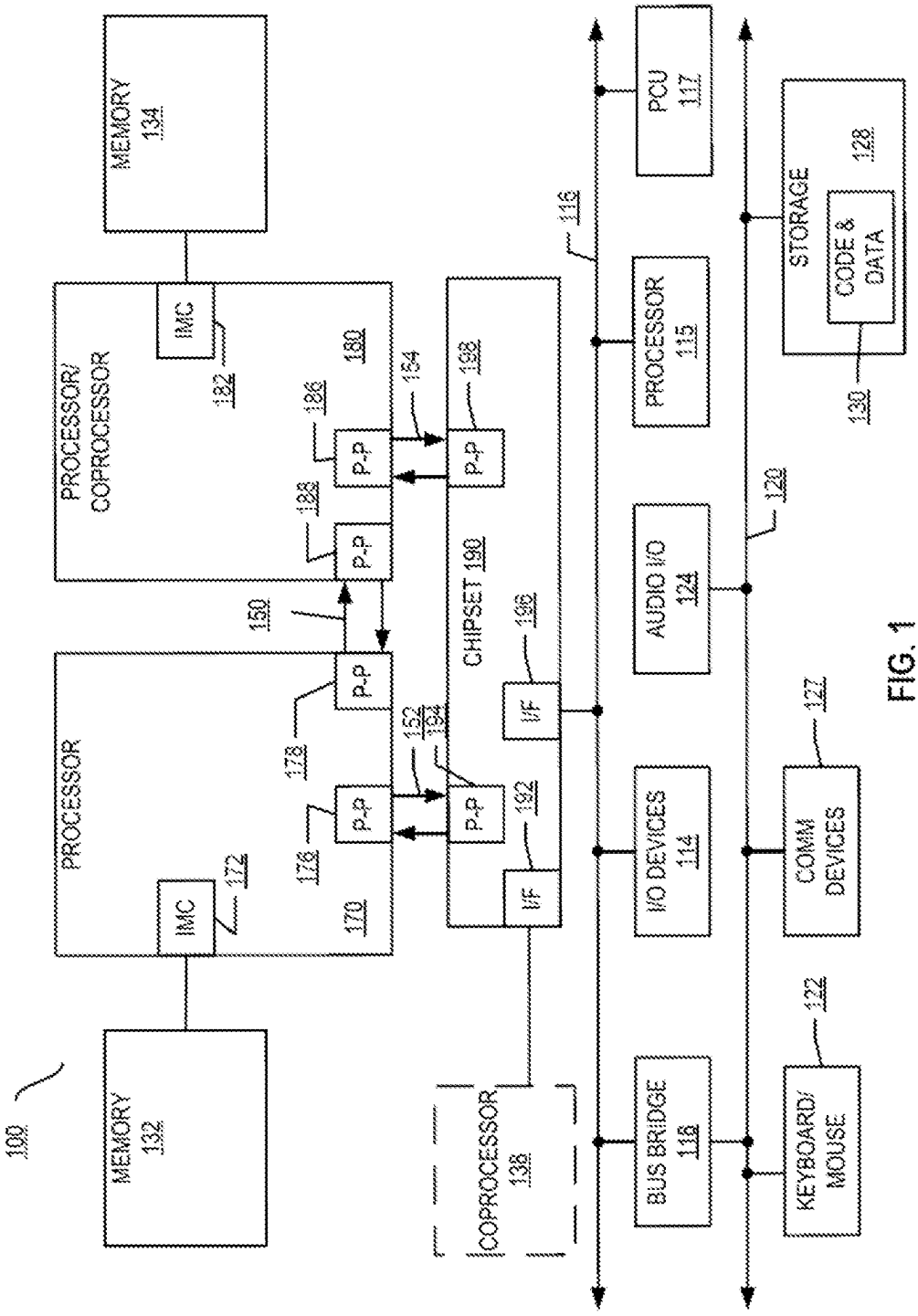

100
MEMORY 132
MEMORY 134
PROCESSOR
IMC 172
170
178
176
PROCESSOR/ COPROCESSOR
IMC 182
180
186
188
150
152
154
P-P
CHIPSET 190
192
194
196
198
I/F
COPROCESSOR 138
BUS BRIDGE 116
I/O DEVICES 114
AUDIO I/O 124
PROCESSOR 115
PCU 117
116
120
127
COMM DEVICES
122
KEYBOARD/ MOUSE
STORAGE
CODE & DATA
128
130

FIG. 1

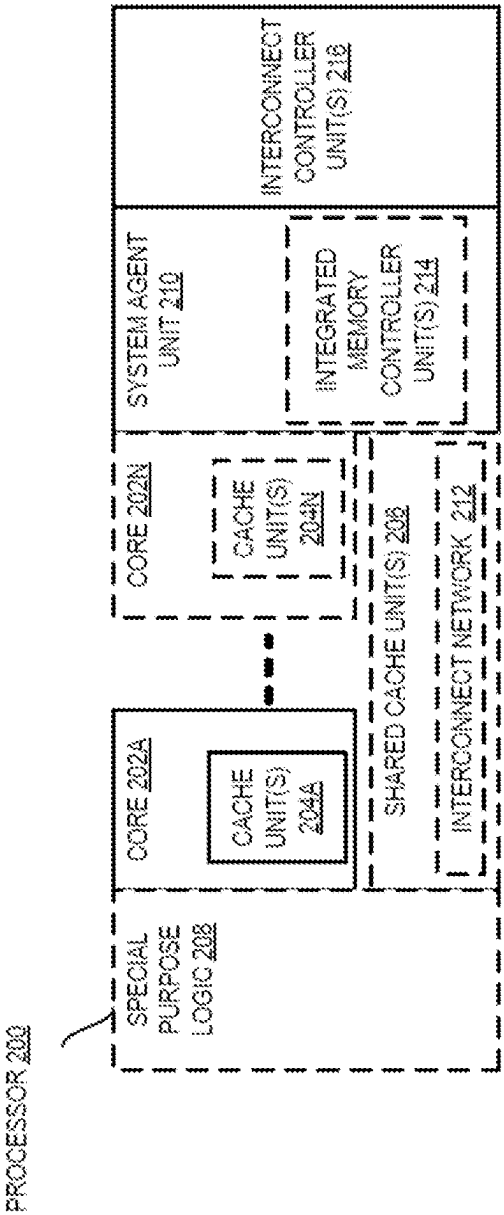

PROCESSOR 200
SPECIAL PURPOSE LOGIC 208
CORE 202A
CACHE UNIT(S) 204A
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) 206
INTERCONNECT NETWORK 212
SYSTEM AGENT UNIT 210
INTEGRATED MEMORY CONTROLLER UNIT(S) 214
INTERCONNECT CONTROLLER UNIT(S) 216

FIG. 2

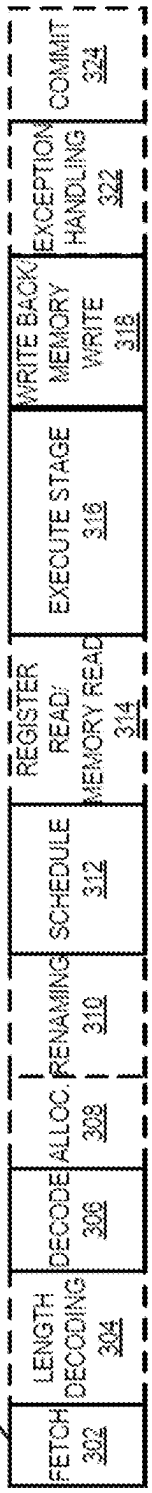

PIPELINE 300
FETCH 302
LENGTH DECODING 304
DECODE 306
ALLOC. 308
RENAMING 310
SCHEDULE 312
REGISTER READ/ MEMORY READ 314
EXECUTE STAGE 316
WRITE BACK/ MEMORY WRITE 318
EXCEPTION HANDLING 322
COMMIT 324

FIG. 3(A)

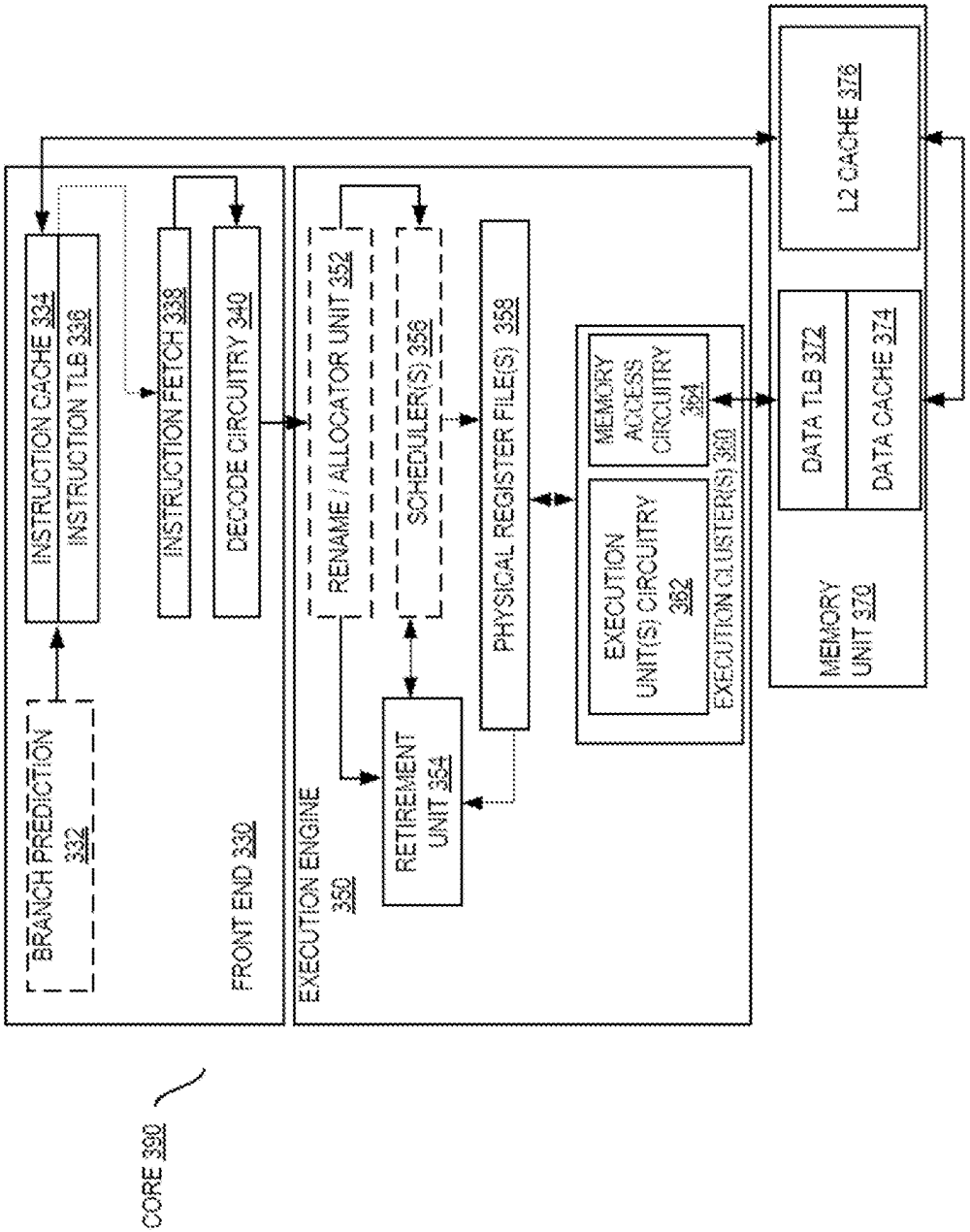

CORE 390
FRONT END 330
BRANCH PREDICTION 332
INSTRUCTION CACHE 334
INSTRUCTION TLB 336
INSTRUCTION FETCH 338
DECODE CIRCUITRY 340
EXECUTION ENGINE 350
RENAME / ALLOCATOR UNIT 352
SCHEDULER(S) 356
PHYSICAL REGISTER FILE(S) 358
RETIREMENT UNIT 354
EXECUTION UNIT(S) CIRCUITRY 362
MEMORY ACCESS CIRCUITRY 364
EXECUTION CLUSTER(S) 360
MEMORY UNIT 370
DATA TLB 372
DATA CACHE 374
L2 CACHE 376

FIG. 3(B)

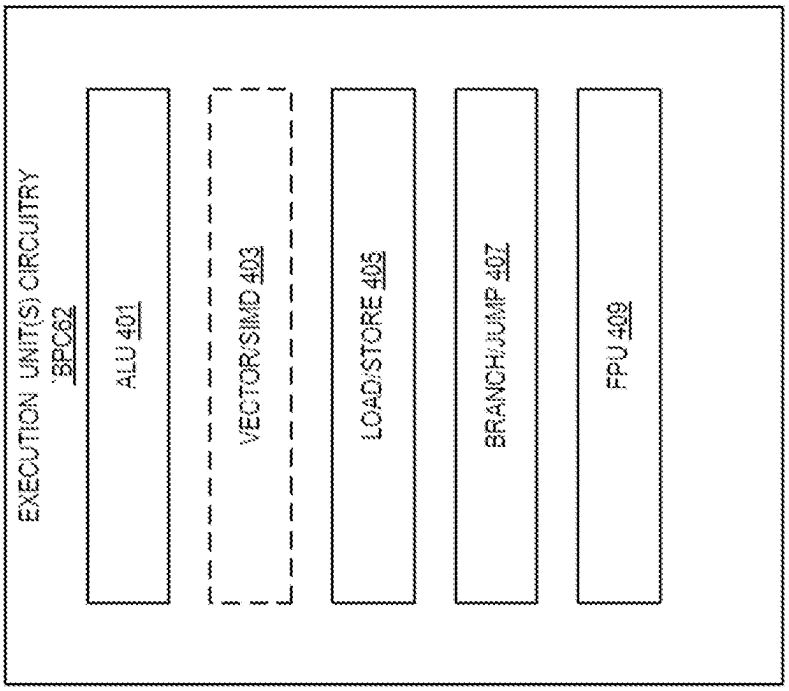

EXECUTION UNIT(S) CIRCUITRY
BPC62
ALU 401
VECTOR/SIMD 403
LOAD/STORE 405
BRANCH/JUMP 407
FPU 409

FIG. 4

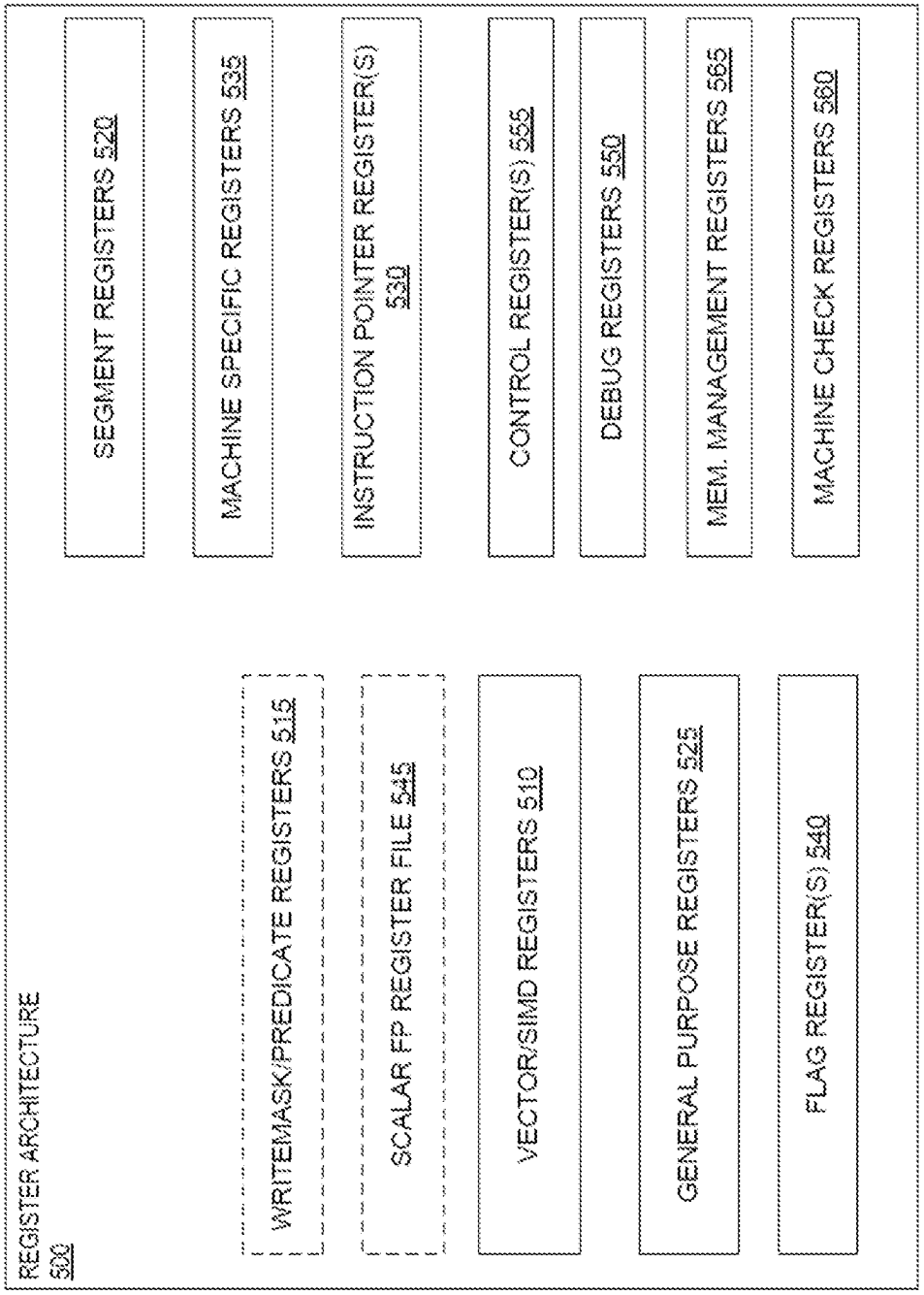

REGISTER ARCHITECTURE
500
SEGMENT REGISTERS 520
MACHINE SPECIFIC REGISTERS 535
INSTRUCTION POINTER REGISTER(S) 530
CONTROL REGISTER(S) 555
DEBUG REGISTERS 550
MEM. MANAGEMENT REGISTERS 565
MACHINE CHECK REGISTERS 560
WRITEMASK/PREDICATE REGISTERS 515
SCALAR FP REGISTER FILE 545
VECTOR/SIMD REGISTERS 510
GENERAL PURPOSE REGISTERS 525
FLAG REGISTER(S) 540

FIG. 5

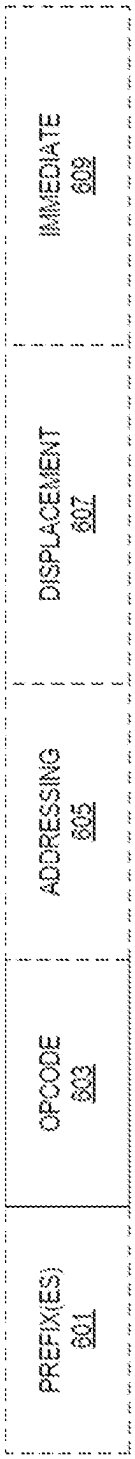

PREFIX(ES)
601
OPCODE
603
ADDRESSING
605
DISPLACEMENT
607
IMMEDIATE
609

FIG. 6

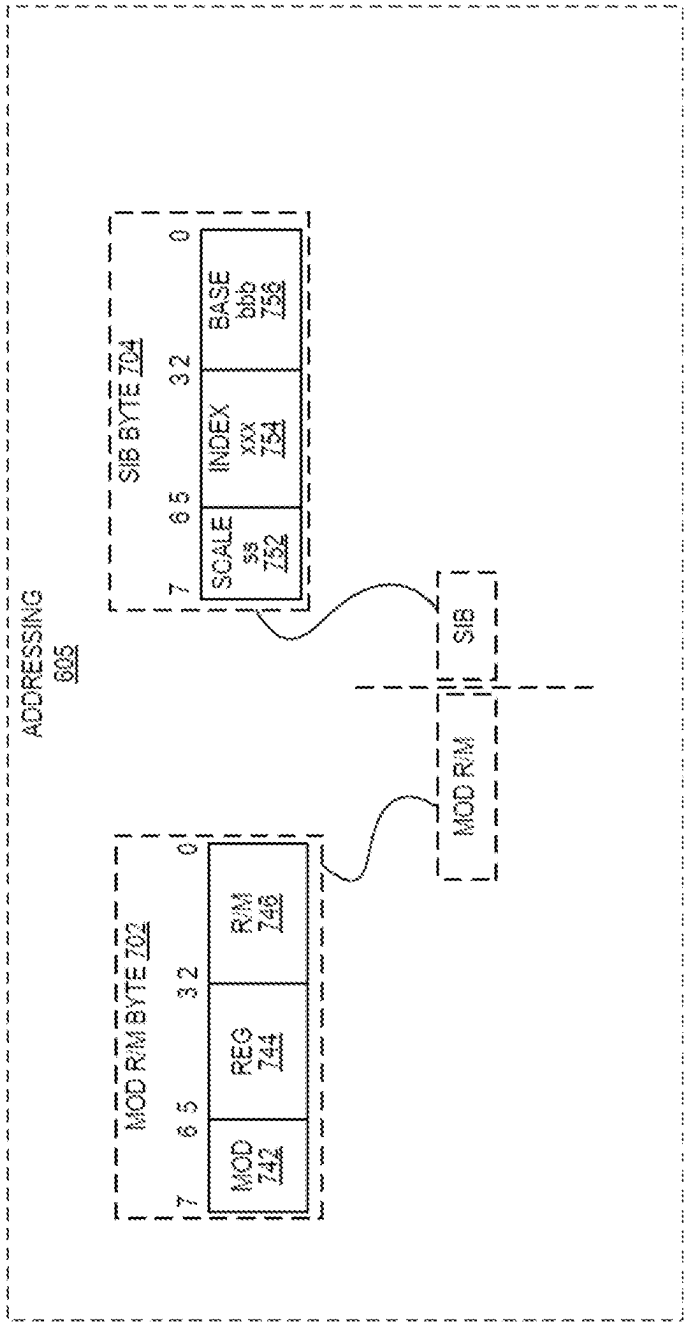

ADDRESSING
605
MOD R/M BYTE 702
7 6 5 3 2 0
MOD 742
REG 744
R/M 746
SIB BYTE 704
7 6 5 3 2 0
SCALE ss 752
INDEX xxx 754
BASE bbb 756
MOD R/M
SIB

FIG. 7

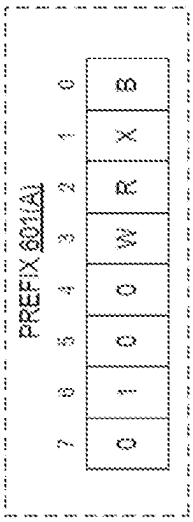


FIG. 8

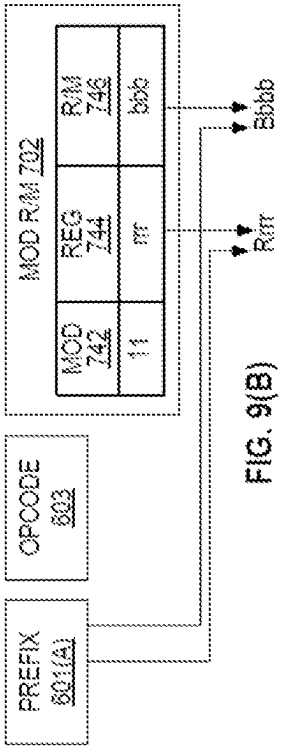


FIG. 9(B)

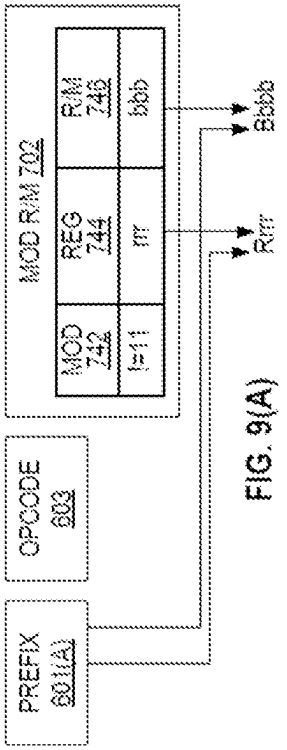


FIG. 9(A)

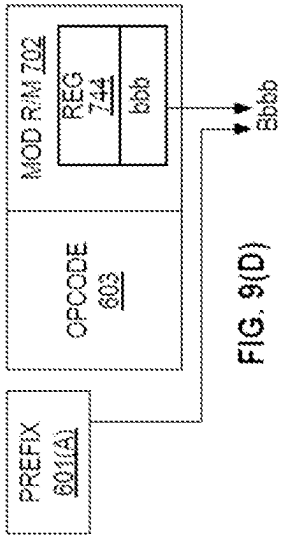


FIG. 9(D)

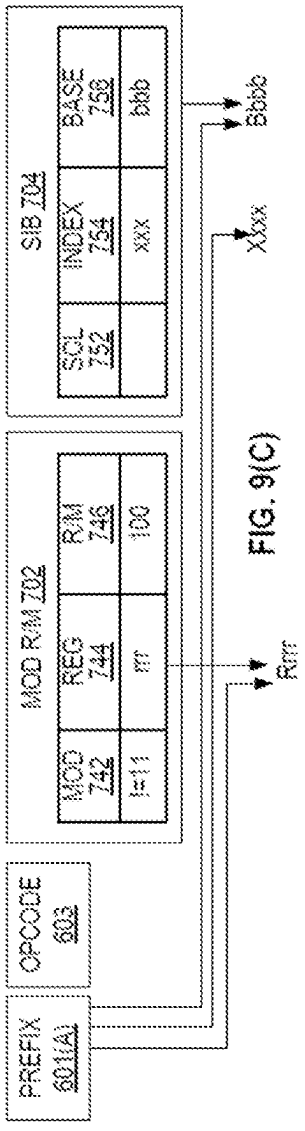


FIG. 9(C)

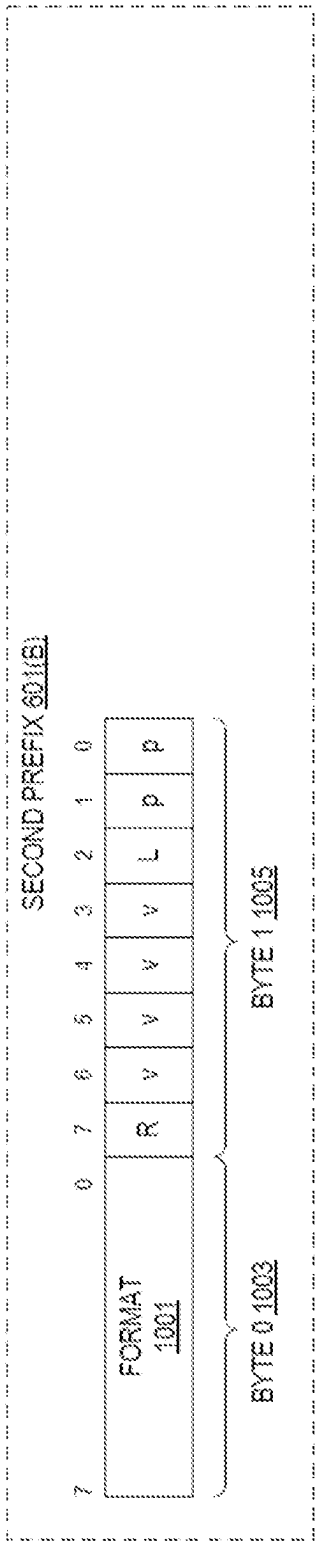


FIG. 10(A)

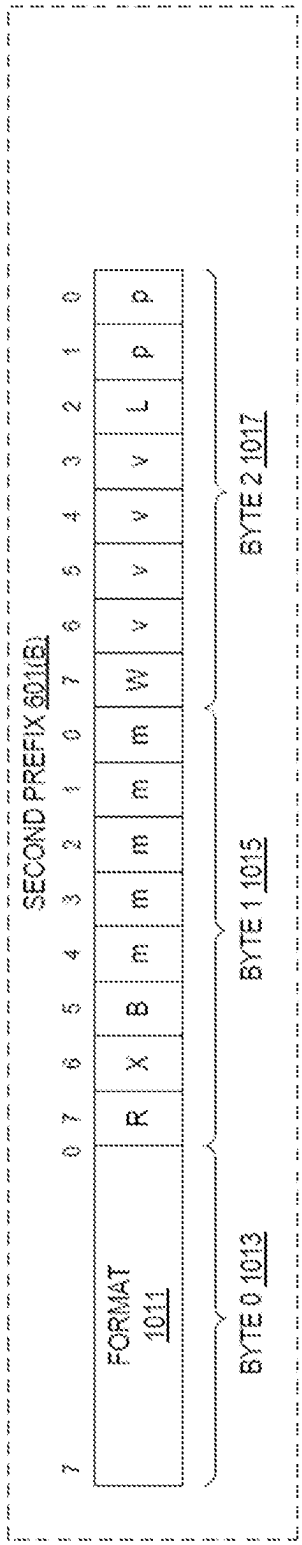


FIG. 10(B)

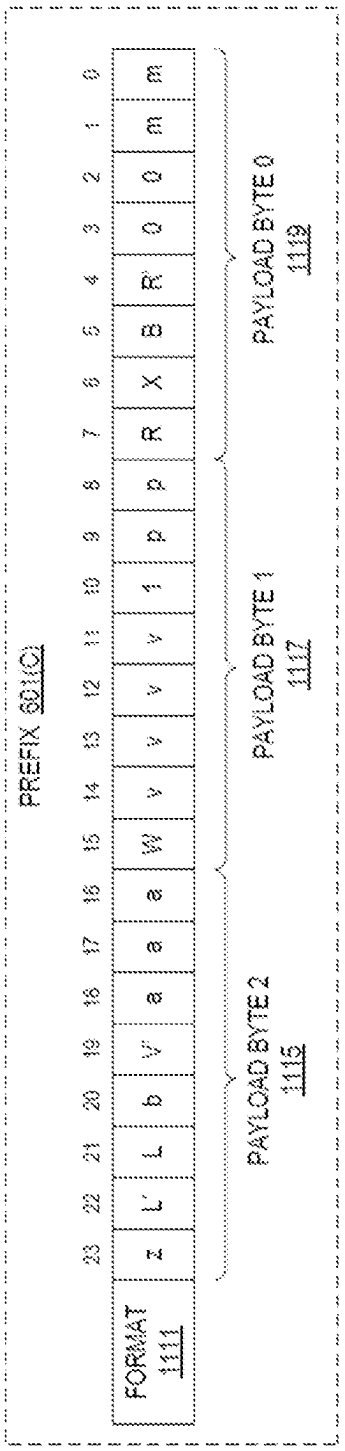

PREFIX 601(C)
FORMAT 1111
PAYLOAD BYTE 2 1115
PAYLOAD BYTE 1 1117
PAYLOAD BYTE 0 1119

FIG. 11

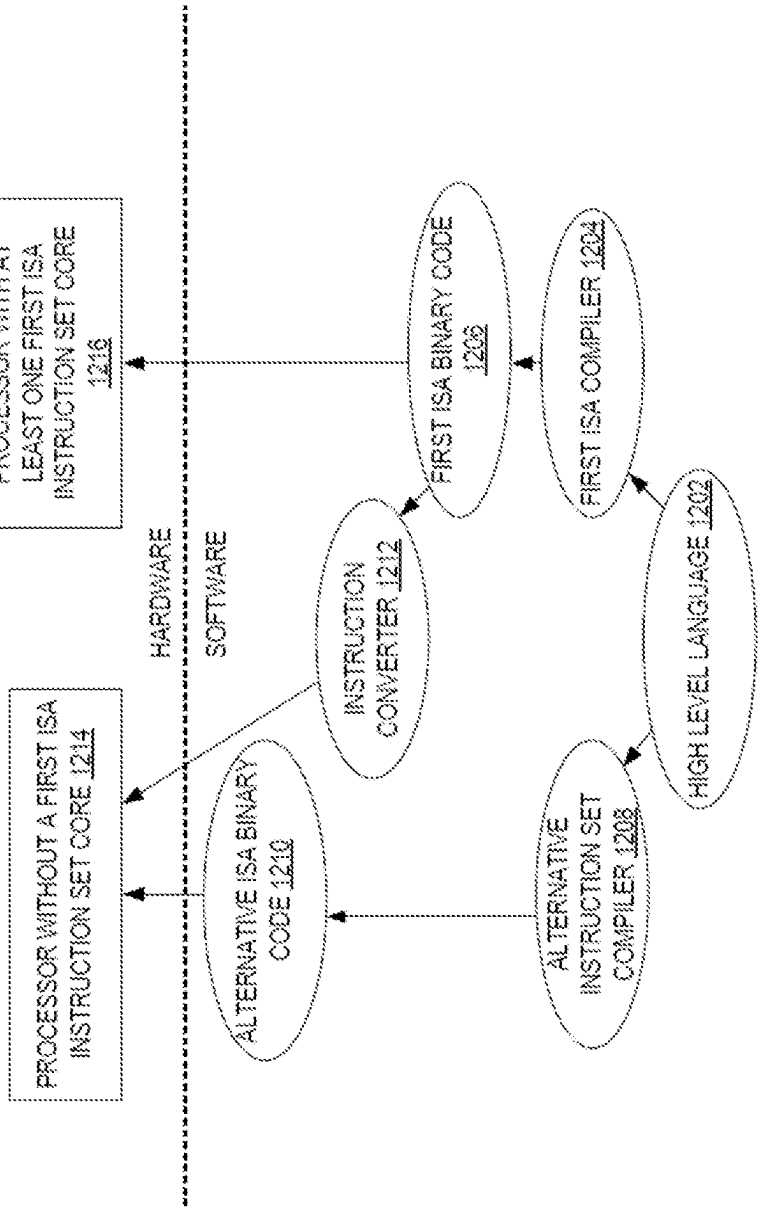

PROCESSOR WITH AT LEAST ONE FIRST ISA INSTRUCTION SET CORE 1216
HARDWARE
SOFTWARE
PROCESSOR WITHOUT A FIRST ISA INSTRUCTION SET CORE 1214
ALTERNATIVE ISA BINARY CODE 1210
INSTRUCTION CONVERTER 1212
FIRST ISA BINARY CODE 1206
ALTERNATIVE INSTRUCTION SET COMPILER 1208
FIRST ISA COMPILER 1204
HIGH LEVEL LANGUAGE 1202

FIG. 12

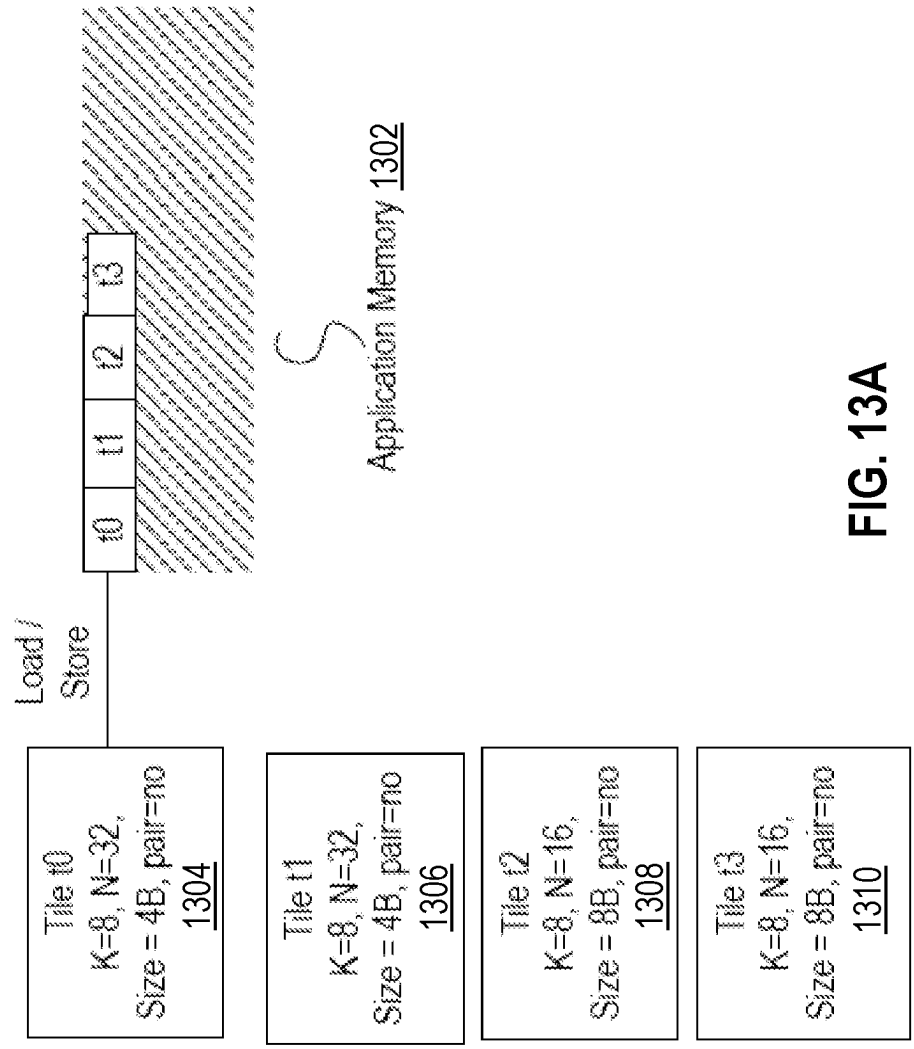

Load / Store
t0
t1
t2
t3
Application Memory 1302
Tile t0
K=8, N=32,
Size = 4B, pair=no
1304
Tile t1
K=8, N=32,
Size = 4B, pair=no
1306
Tile t2
K=8, N=16,
Size = 8B, pair=no
1308
Tile t3
K=8, N=16,
Size = 8B, pair=no
1310

FIG. 13A

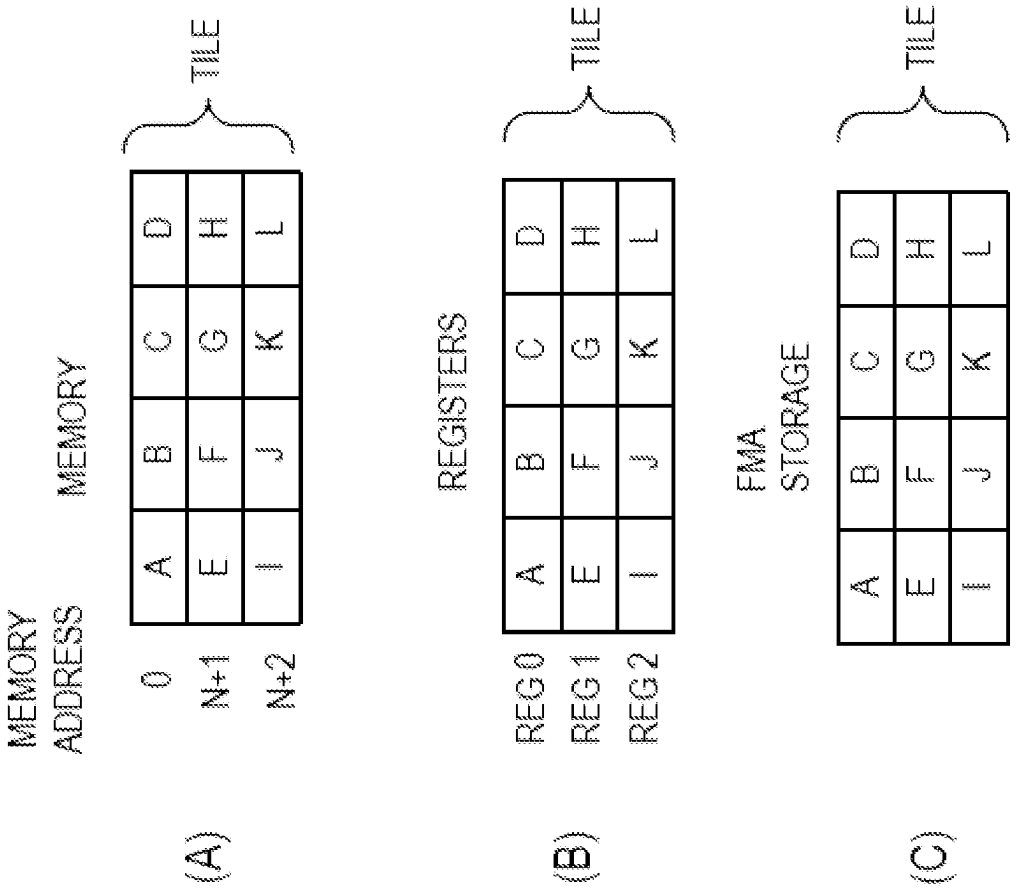

MEMORY ADDRESS
MEMORY
TILE
0
N+1
N+2
A B C D
E F G H
I J K L
(A)
REGISTERS
TILE
REG 0
REG 1
REG 2
A B C D
E F G H
I J K L
(B)
EMA STORAGE
TILE
A B C D
E F G H
I J K L
(C)

FIG. 13B

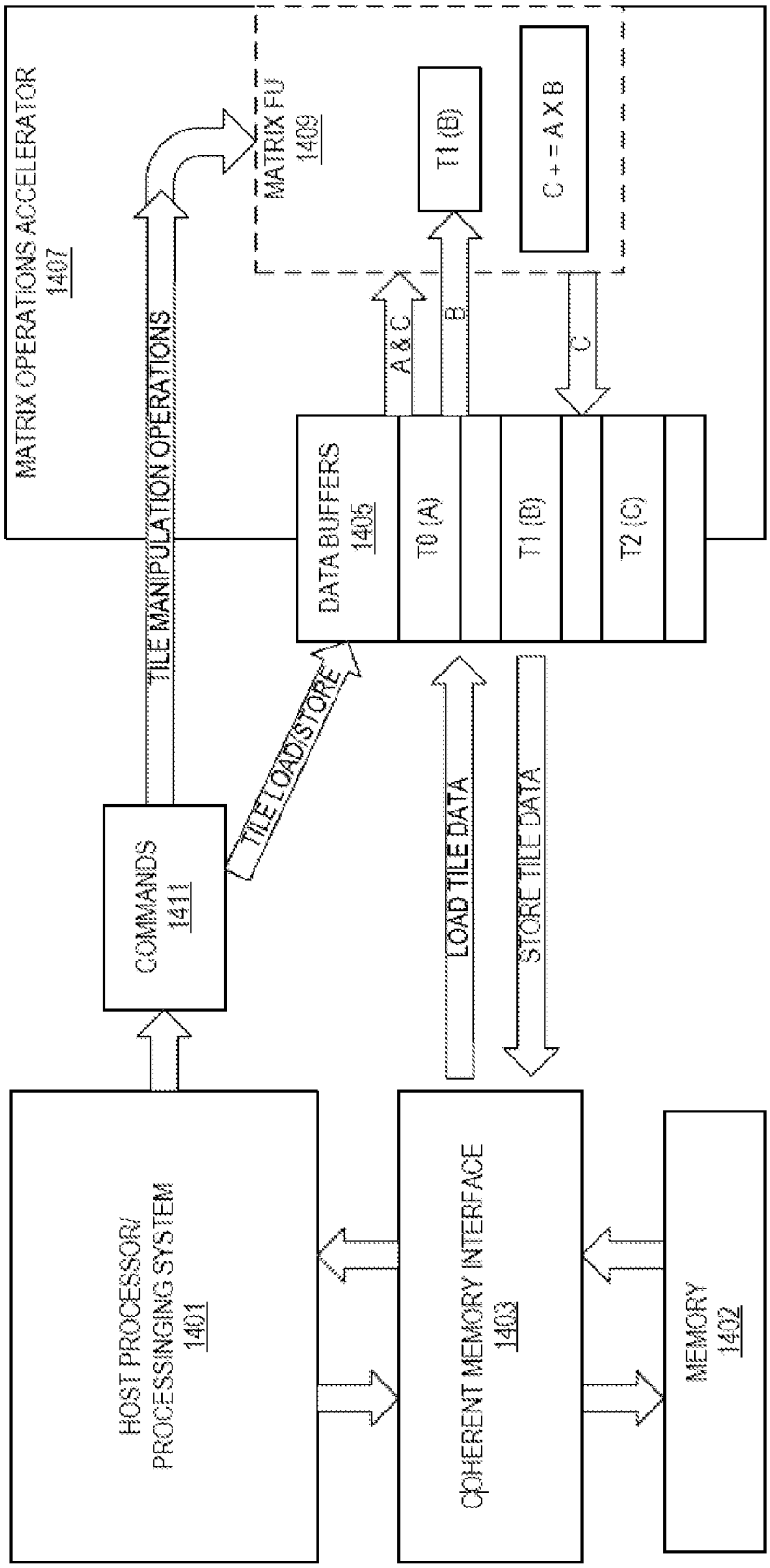

MATRIX OPERATIONS ACCELERATOR
1407
MATRIX FU
1409
T1 (B)
C + = A X B
A & C
B
C
TILE MANIPULATION OPERATIONS
DATA BUFFERS
1405
T0 (A)
T1 (B)
T2 (C)
TILE LOAD/STORE
COMMANDS
1411
LOAD TILE DATA
STORE TILE DATA
HOST PROCESSOR/
PROCESSING SYSTEM
1401
COHERENT MEMORY INTERFACE
1403
MEMORY
1402

FIG. 14

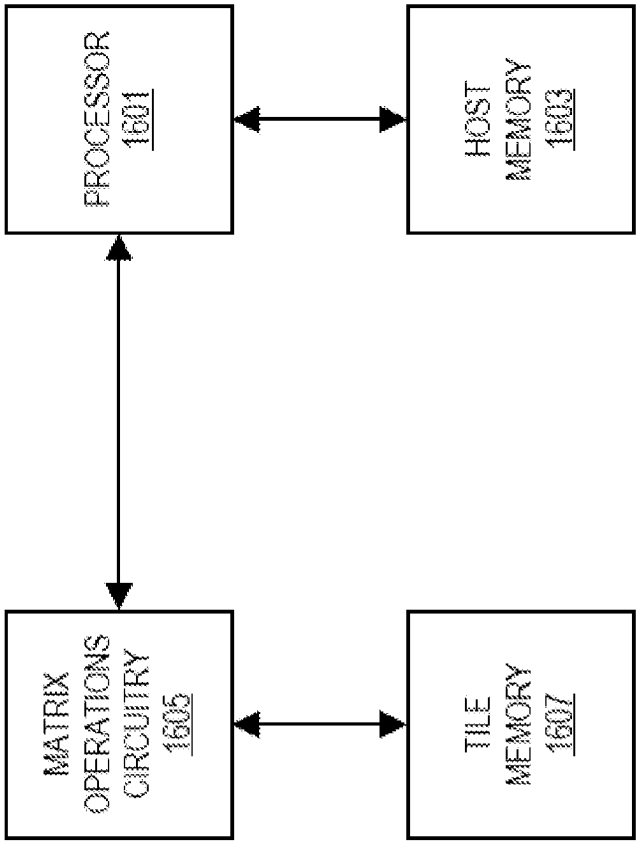

PROCESSOR
1601
HOST
MEMORY
1603
MATRIX
OPERATIONS
CIRCUITRY
1605
TILE
MEMORY
1607

FIG. 16

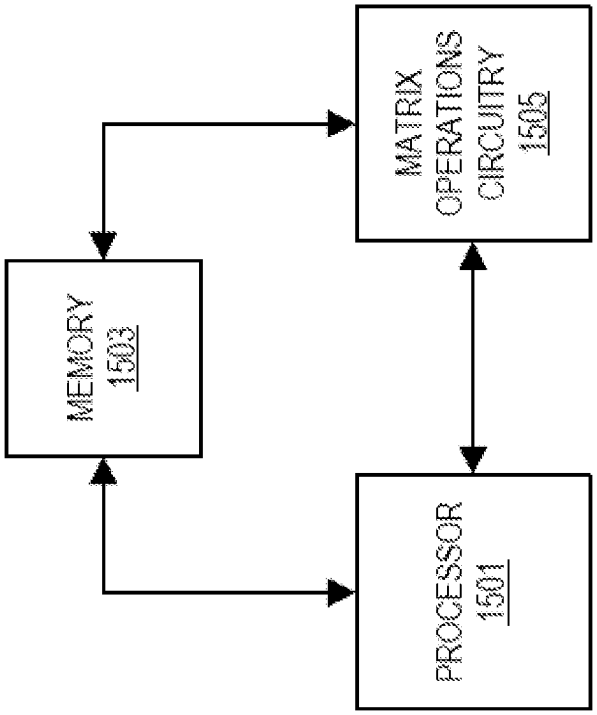

MATRIX
OPERATIONS
CIRCUITRY
1505
MEMORY
1503
PROCESSOR
1501

FIG. 15

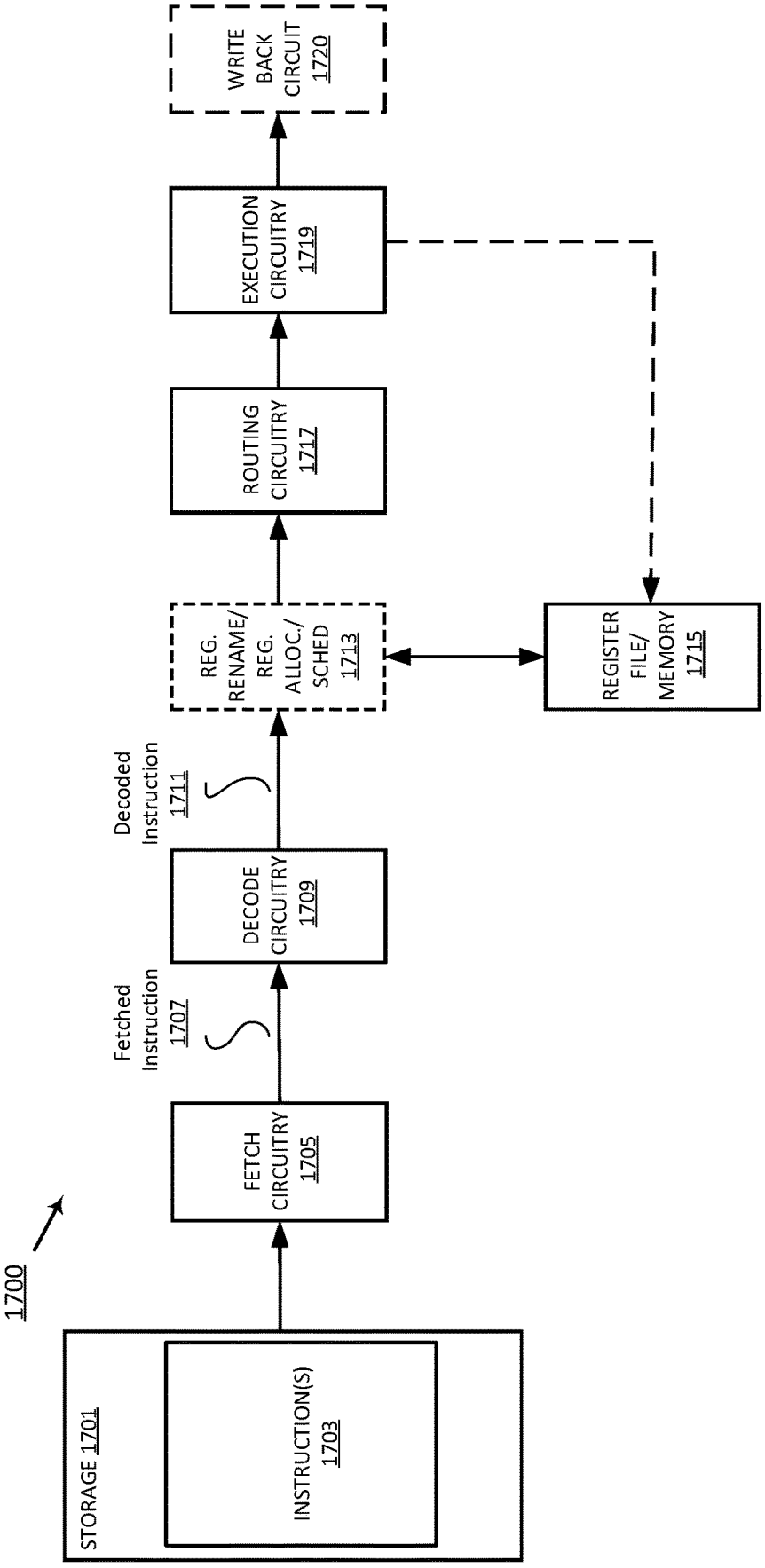

1700
STORAGE 1701
INSTRUCTION(S) 1703
FETCH CIRCUITRY 1705
Fetched Instruction 1707
DECODE CIRCUITRY 1709
Decoded Instruction 1711
REG. RENAME/ REG. ALLOC./ SCHED 1713
REGISTER FILE/ MEMORY 1715
ROUTING CIRCUITRY 1717
EXECUTION CIRCUITRY 1719
WRITE BACK CIRCUIT 1720

FIG. 17

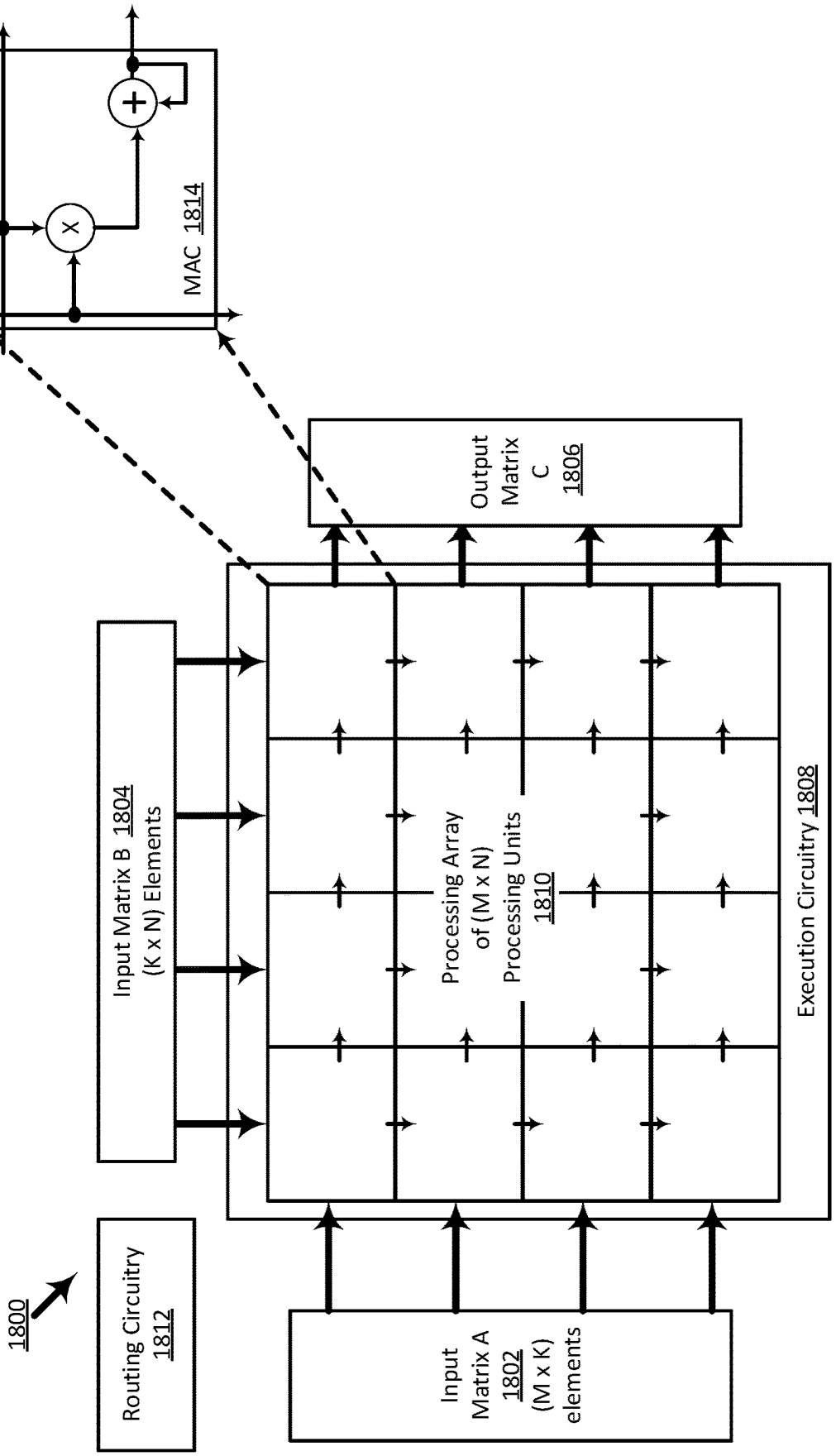

1800
Routing Circuitry 1812
Input Matrix A 1802 (M x K) elements
Input Matrix B 1804 (K x N) Elements
Processing Array of (M x N) Processing Units 1810
Execution Circuitry 1808
Output Matrix C 1806
MAC 1814
x
+

FIG. 18

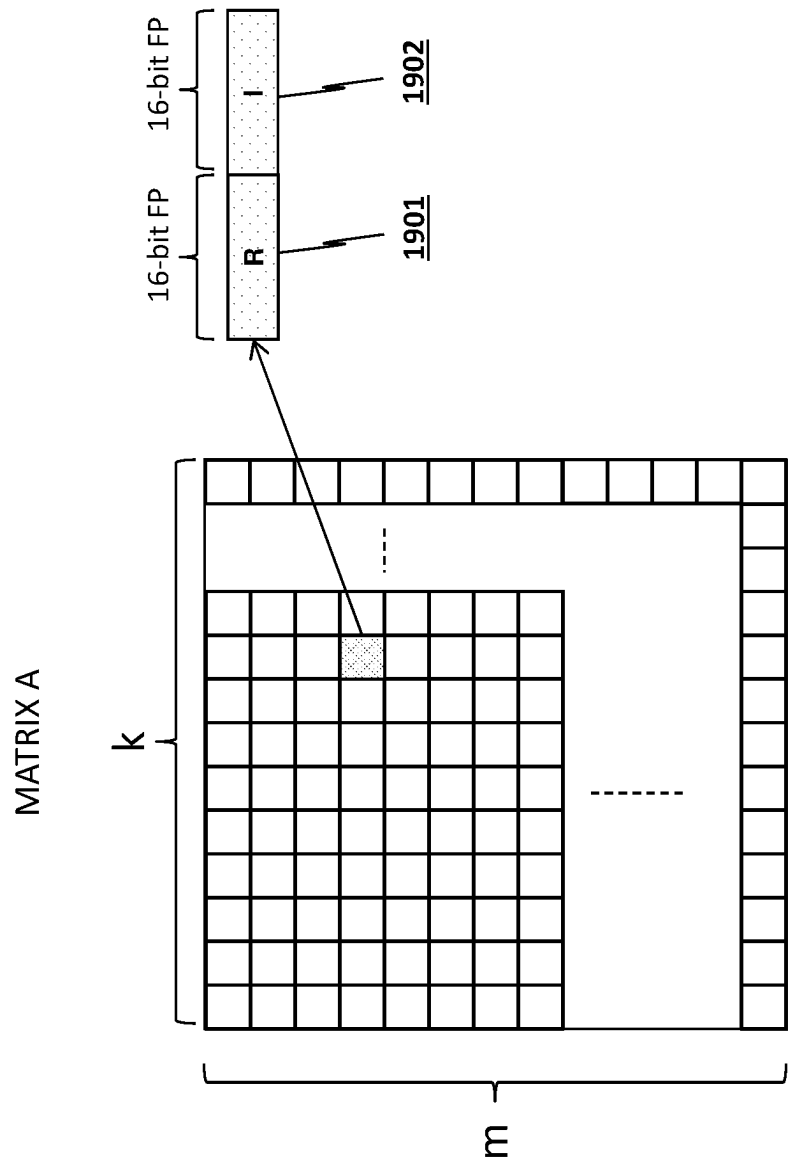


FIG. 19

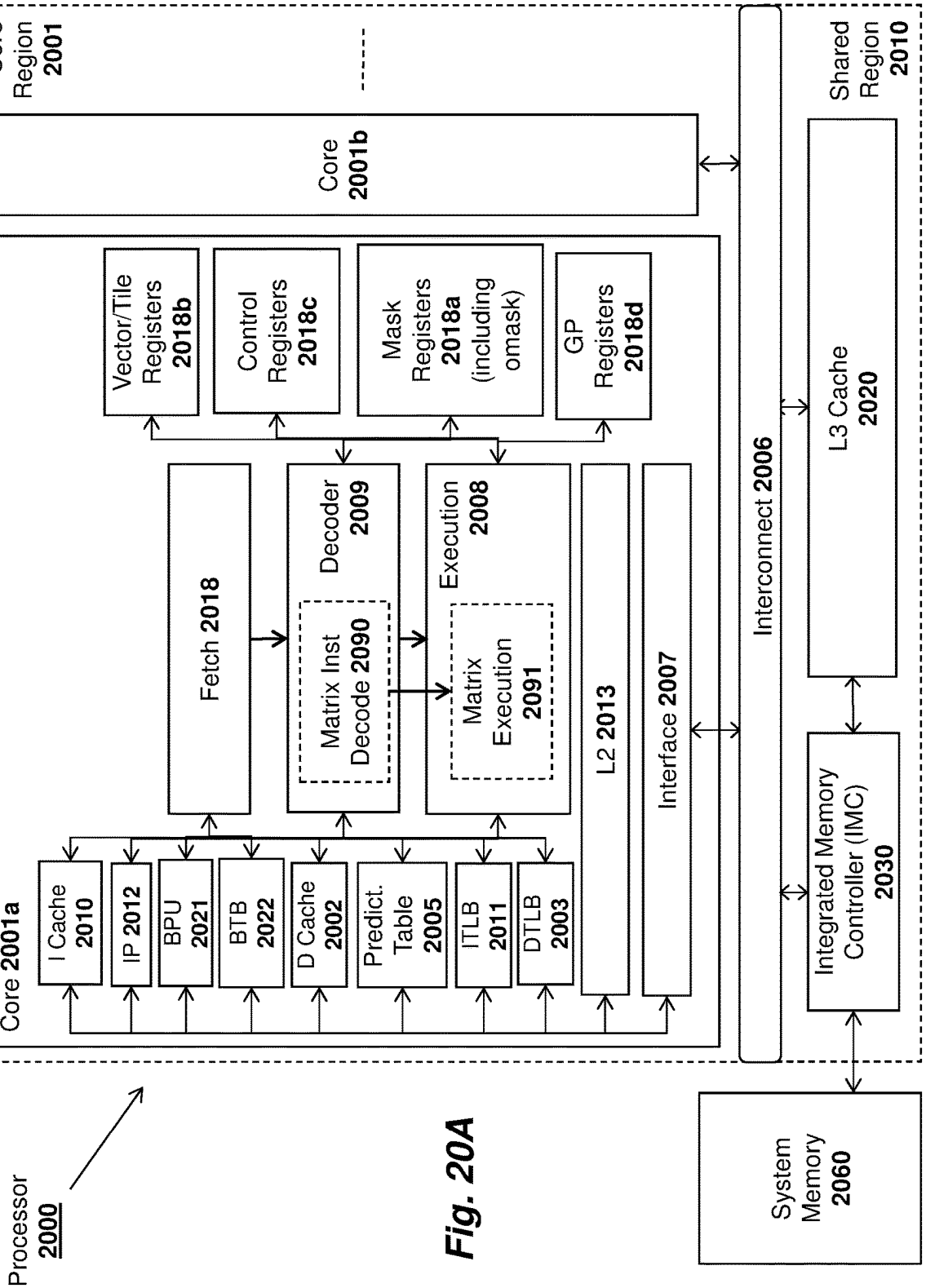

Processor 2000
Core Region 2001
Core 2001b
Core 2001a
Vector/Tile Registers 2018b
Control Registers 2018c
Mask Registers 2018a (including omask)
GP Registers 2018d
Fetch 2018
Decoder 2009
Matrix Inst Decode 2090
Execution 2008
Matrix Execution 2091
L2 2013
Interface 2007
I Cache 2010
IP 2012
BPU 2021
BTB 2022
D Cache 2002
Predict. Table 2005
ITLB 2011
DTLB 2003
Interconnect 2006
Shared Region 2010
L3 Cache 2020
Integrated Memory Controller (IMC) 2030
System Memory 2060

Fig. 20A

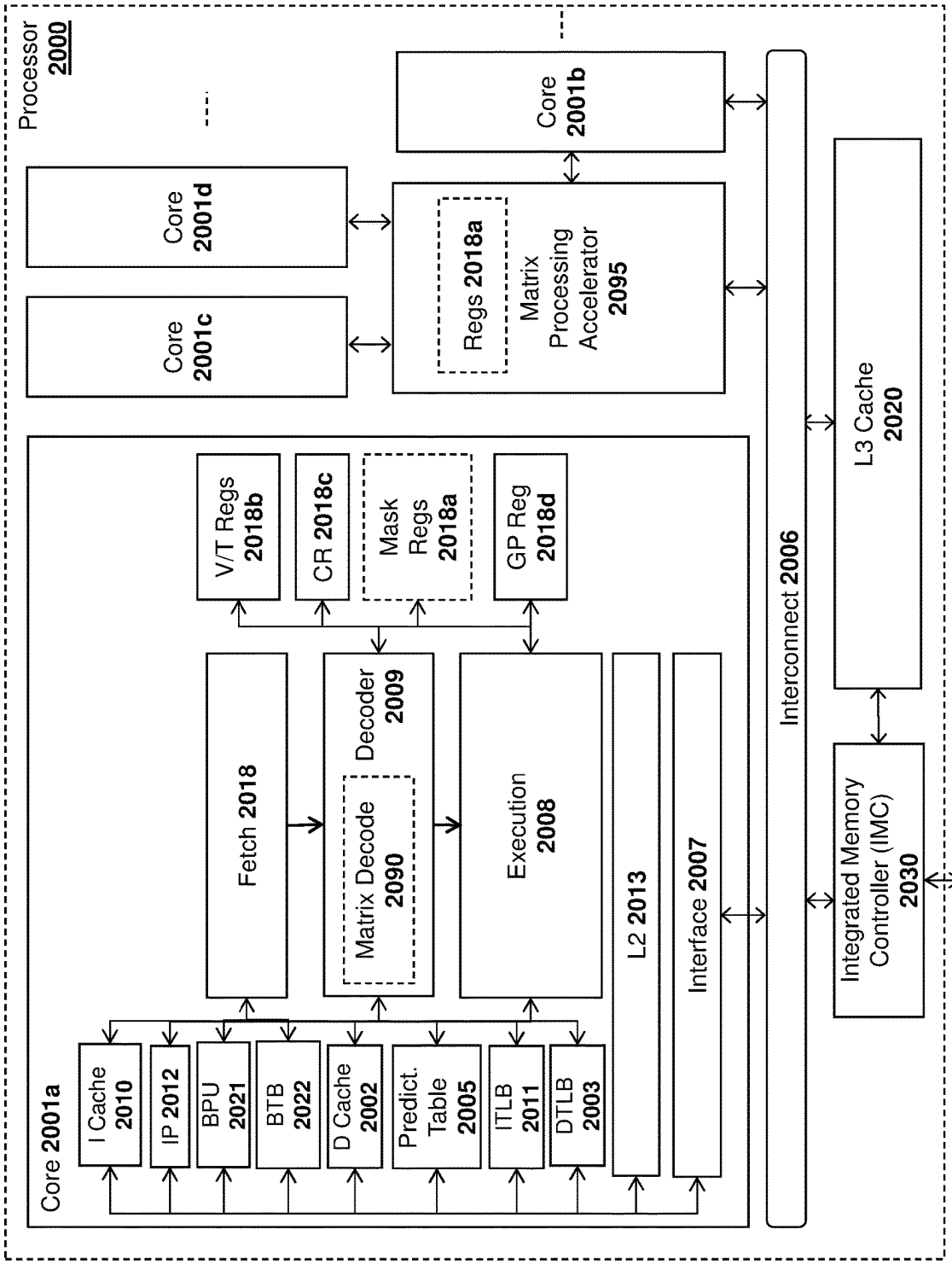

Processor 2000
Core 2001a
I Cache 2010
IP 2012
BPU 2021
BTB 2022
D Cache 2002
Predict. Table 2005
ITLB 2011
DTLB 2003
Fetch 2018
Decoder 2009
Matrix Decode 2090
Execution 2008
L2 2013
Interface 2007
V/T Regs 2018b
CR 2018c
Mask Regs 2018a
GP Reg 2018d
Core 2001c
Core 2001d
Core 2001b
Regs 2018a
Matrix Processing Accelerator 2095
Interconnect 2006
L3 Cache 2020
Integrated Memory Controller (IMC) 2030
System Memory 2060

Fig. 20B

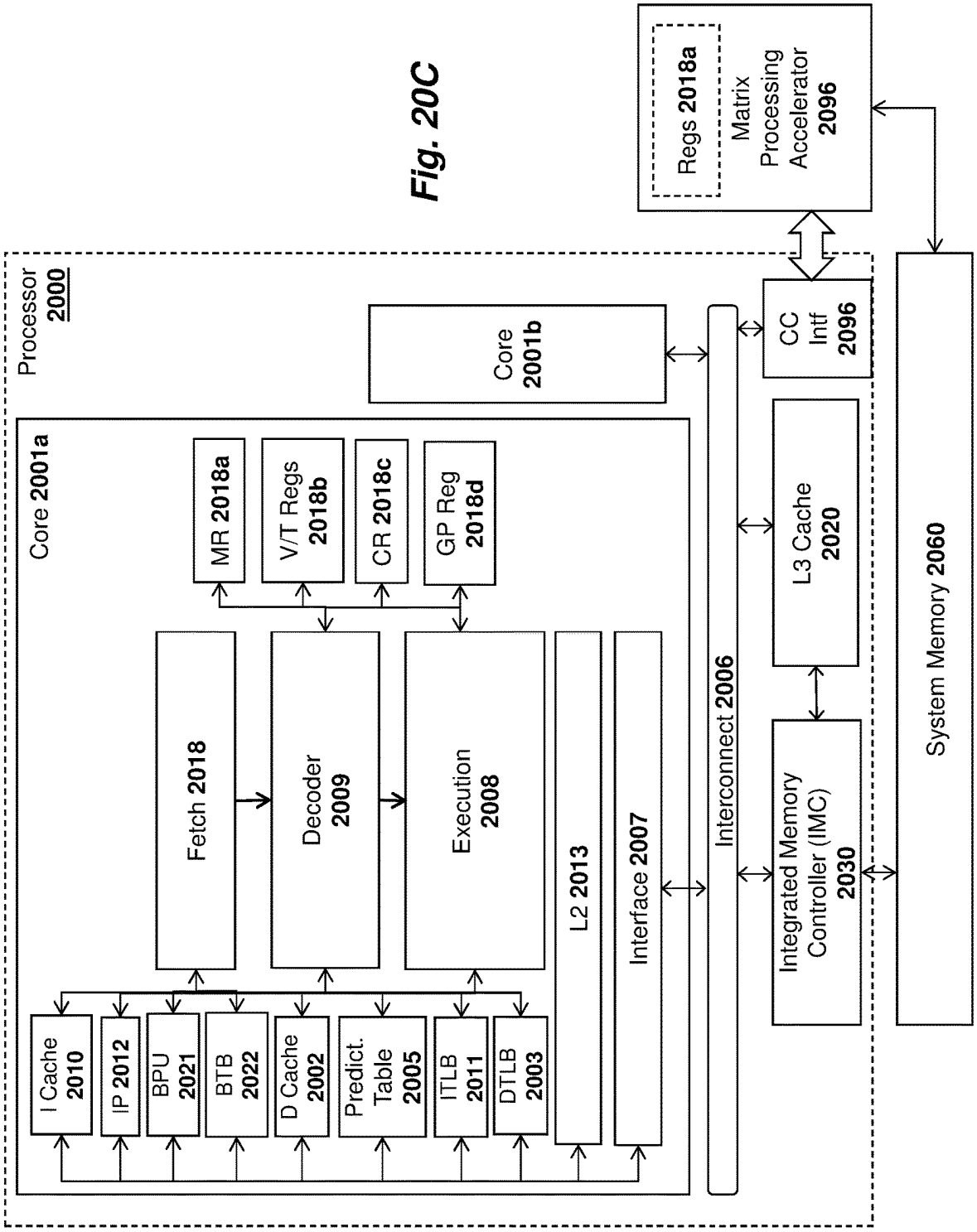

Processor 2000
Core 2001a
I Cache 2010
IP 2012
BPU 2021
BTB 2022
D Cache 2002
Predict. Table 2005
ITLB 2011
DTLB 2003
Fetch 2018
Decoder 2009
Execution 2008
MR 2018a
V/T Regs 2018b
CR 2018c
GP Reg 2018d
L2 2013
Interface 2007
Core 2001b
Interconnect 2006
CC Intf 2096
L3 Cache 2020
Integrated Memory Controller (IMC) 2030
Regs 2018a
Matrix Processing Accelerator 2096
System Memory 2060

Fig. 20C

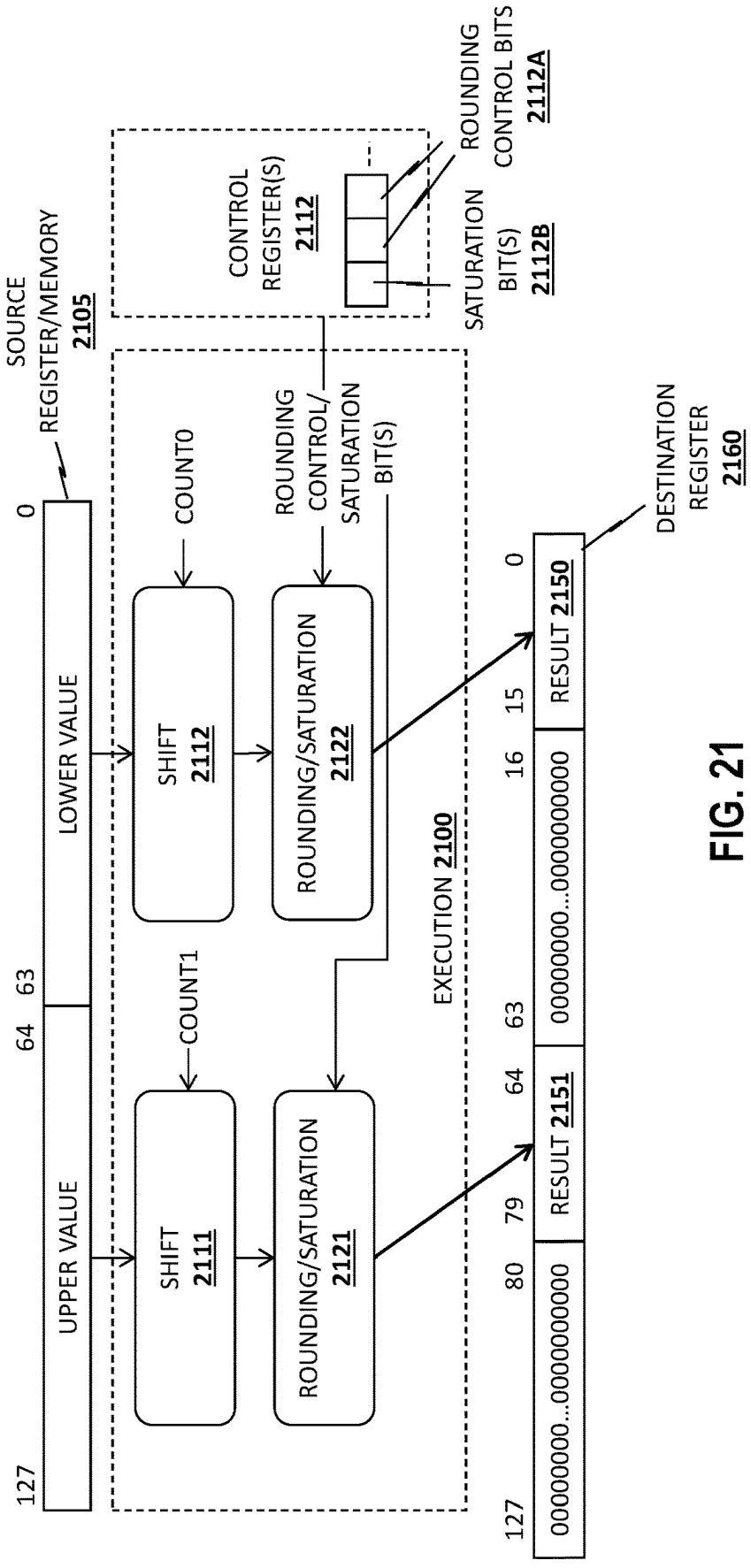

SOURCE REGISTER/MEMORY 2105
0
127
64
63
UPPER VALUE
LOWER VALUE
COUNT1
COUNT0
SHIFT 2111
SHIFT 2112
ROUNDING/SATURATION 2121
ROUNDING/SATURATION 2122
ROUNDING CONTROL/ SATURATION BIT(S)
CONTROL REGISTER(S) 2112
ROUNDING CONTROL BITS 2112A
SATURATION BIT(S) 2112B
EXECUTION 2100
DESTINATION REGISTER 2160
RESULT 2151
RESULT 2150
00000000...0000000000
00000000...0000000000
80
79
64
63
16
15
0
127

FIG. 21

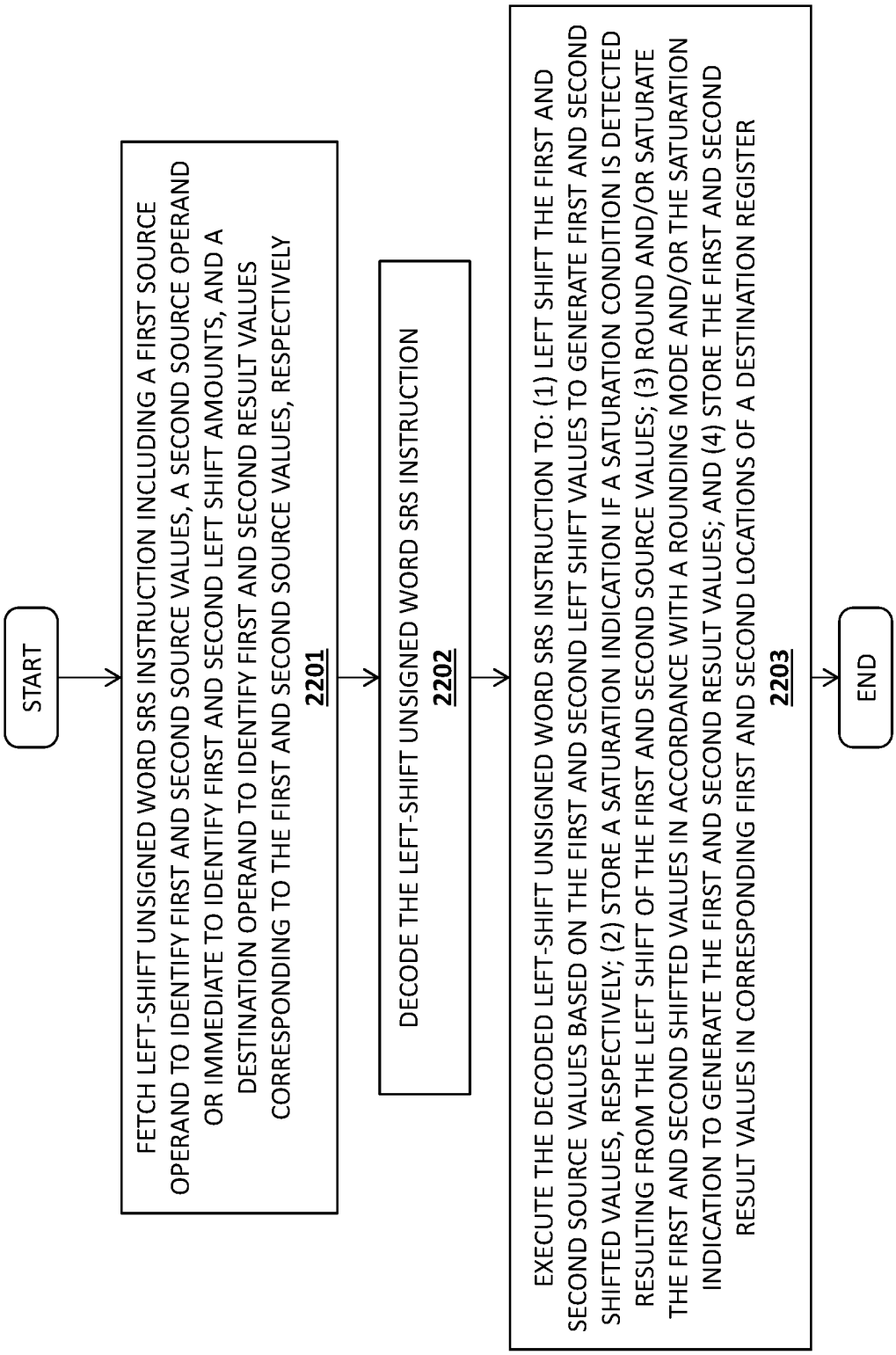

START
FETCH LEFT-SHIFT UNSIGNED WORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY FIRST AND SECOND LEFT SHIFT AMOUNTS, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2201
DECODE THE LEFT-SHIFT UNSIGNED WORD SRS INSTRUCTION
2202
EXECUTE THE DECODED LEFT-SHIFT UNSIGNED WORD SRS INSTRUCTION TO: (1) LEFT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE FIRST AND SECOND LEFT SHIFT VALUES TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) STORE A SATURATION INDICATION IF A SATURATION CONDITION IS DETECTED RESULTING FROM THE LEFT SHIFT OF THE FIRST AND SECOND SOURCE VALUES; (3) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR THE SATURATION INDICATION TO GENERATE THE FIRST AND SECOND RESULT VALUES; AND (4) STORE THE FIRST AND SECOND RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2203
END

FIG. 22

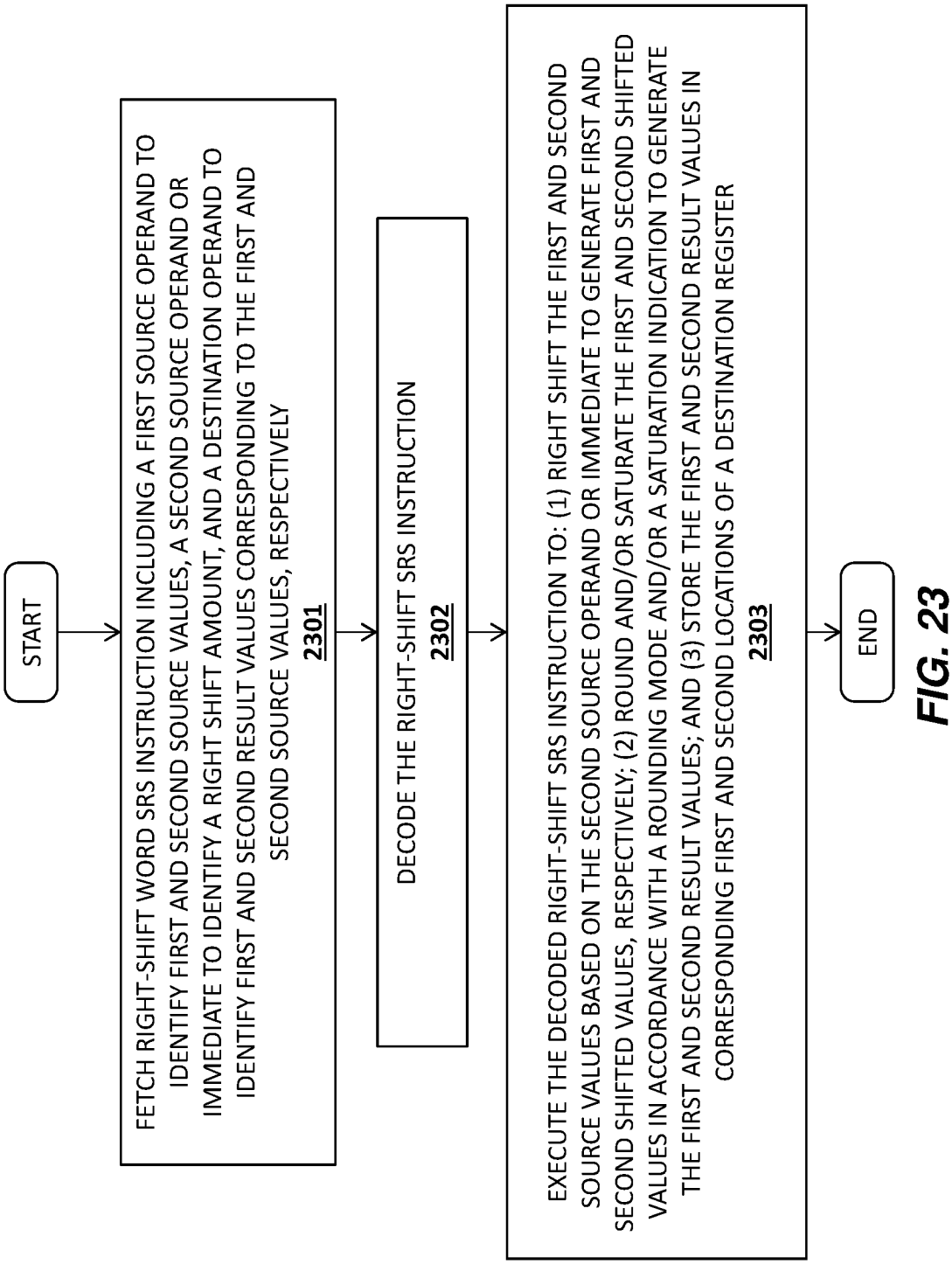

START
FETCH RIGHT-SHIFT WORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY A RIGHT SHIFT AMOUNT, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2301
DECODE THE RIGHT-SHIFT SRS INSTRUCTION
2302
EXECUTE THE DECODED RIGHT-SHIFT SRS INSTRUCTION TO: (1) RIGHT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE SECOND SOURCE OPERAND OR IMMEDIATE TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR A SATURATION INDICATION TO GENERATE THE FIRST AND SECOND RESULT VALUES; AND (3) STORE THE FIRST AND SECOND RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2303
END

FIG. 23

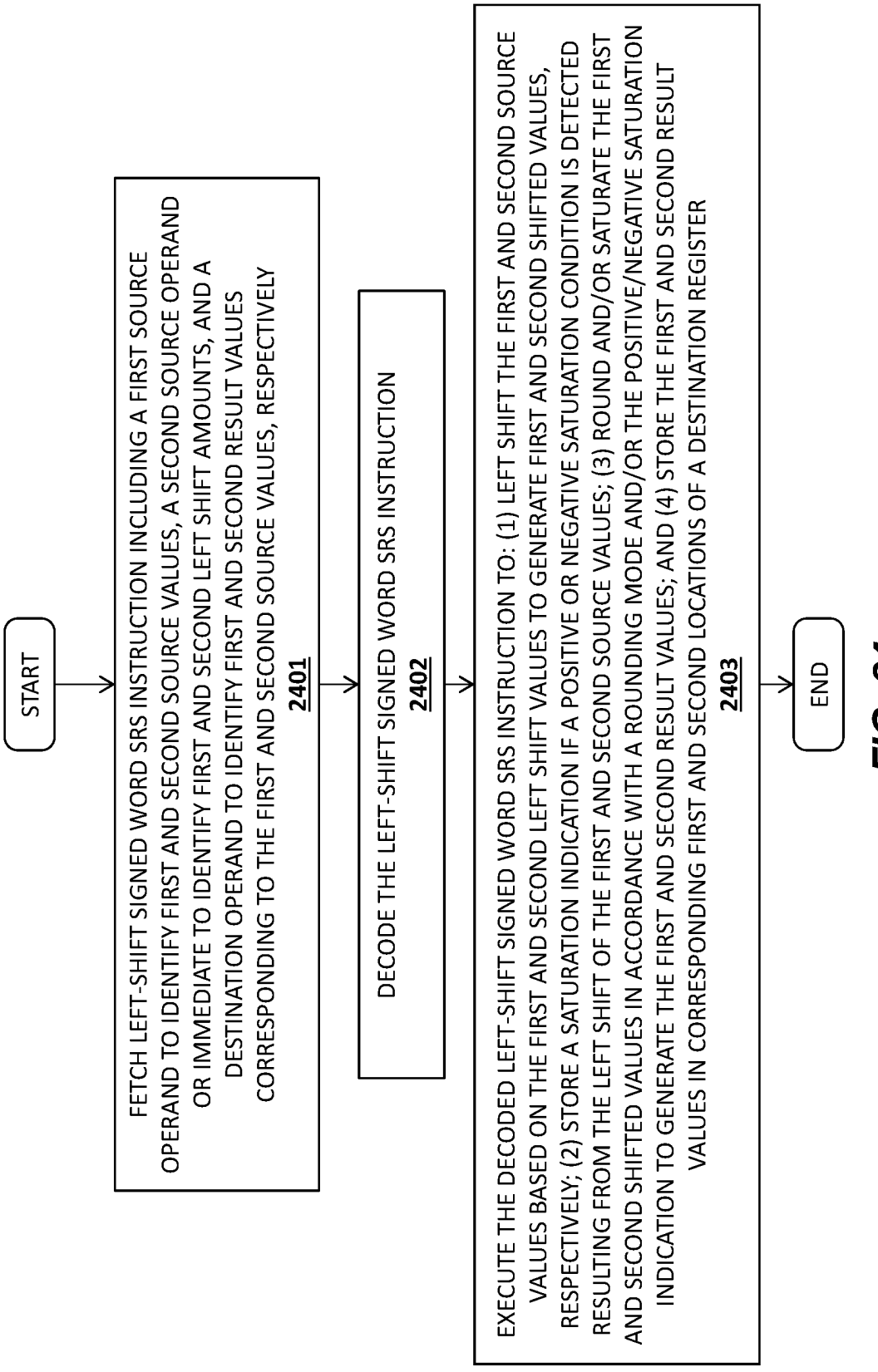

START
FETCH LEFT-SHIFT SIGNED WORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY FIRST AND SECOND LEFT SHIFT AMOUNTS, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2401
DECODE THE LEFT-SHIFT SIGNED WORD SRS INSTRUCTION
2402
EXECUTE THE DECODED LEFT-SHIFT SIGNED WORD SRS INSTRUCTION TO: (1) LEFT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE FIRST AND SECOND LEFT SHIFT VALUES TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) STORE A SATURATION INDICATION IF A POSITIVE OR NEGATIVE SATURATION CONDITION IS DETECTED RESULTING FROM THE LEFT SHIFT OF THE FIRST AND SECOND SOURCE VALUES; (3) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR THE POSITIVE/NEGATIVE SATURATION INDICATION TO GENERATE THE FIRST AND SECOND RESULT VALUES; AND (4) STORE THE FIRST AND SECOND RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2403
END

FIG. 24

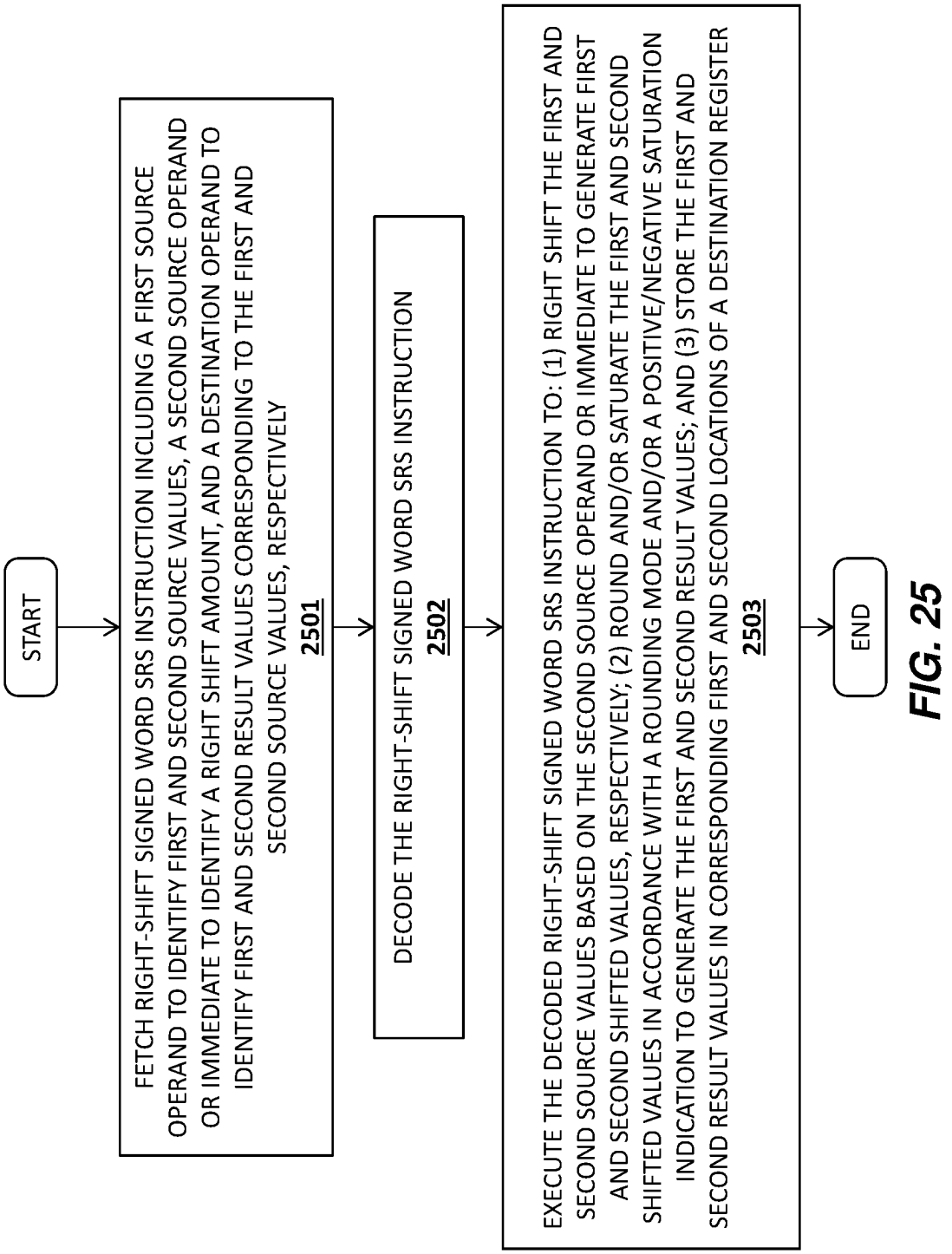

START
FETCH RIGHT-SHIFT SIGNED WORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY A RIGHT SHIFT AMOUNT, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2501
DECODE THE RIGHT-SHIFT SIGNED WORD SRS INSTRUCTION
2502
EXECUTE THE DECODED RIGHT-SHIFT SIGNED WORD SRS INSTRUCTION TO: (1) RIGHT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE SECOND SOURCE OPERAND OR IMMEDIATE TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR A POSITIVE/NEGATIVE SATURATION INDICATION TO GENERATE THE FIRST AND SECOND RESULT VALUES; AND (3) STORE THE FIRST AND SECOND RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2503
END

FIG. 25

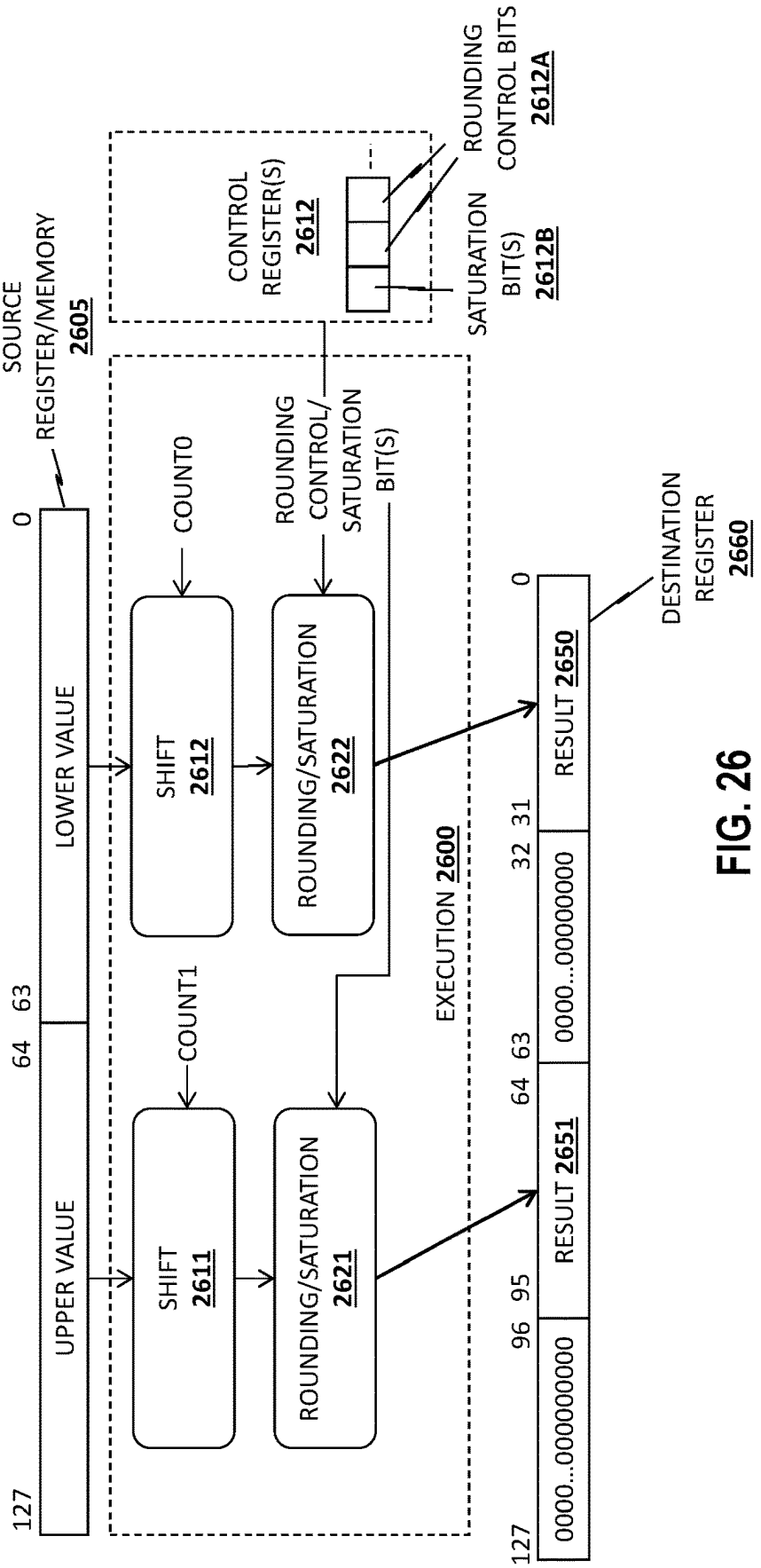

SOURCE REGISTER/MEMORY 2605
0
63
64
127
LOWER VALUE
UPPER VALUE
COUNT0
COUNT1
SHIFT 2612
SHIFT 2611
ROUNDING/SATURATION 2622
ROUNDING/SATURATION 2621
ROUNDING CONTROL/ SATURATION BIT(S)
CONTROL REGISTER(S) 2612
ROUNDING CONTROL BITS 2612A
SATURATION BIT(S) 2612B
EXECUTION 2600
0
31
32
63
64
95
96
127
RESULT 2650
0000...00000000
RESULT 2651
0000...000000000
DESTINATION REGISTER 2660

FIG. 26

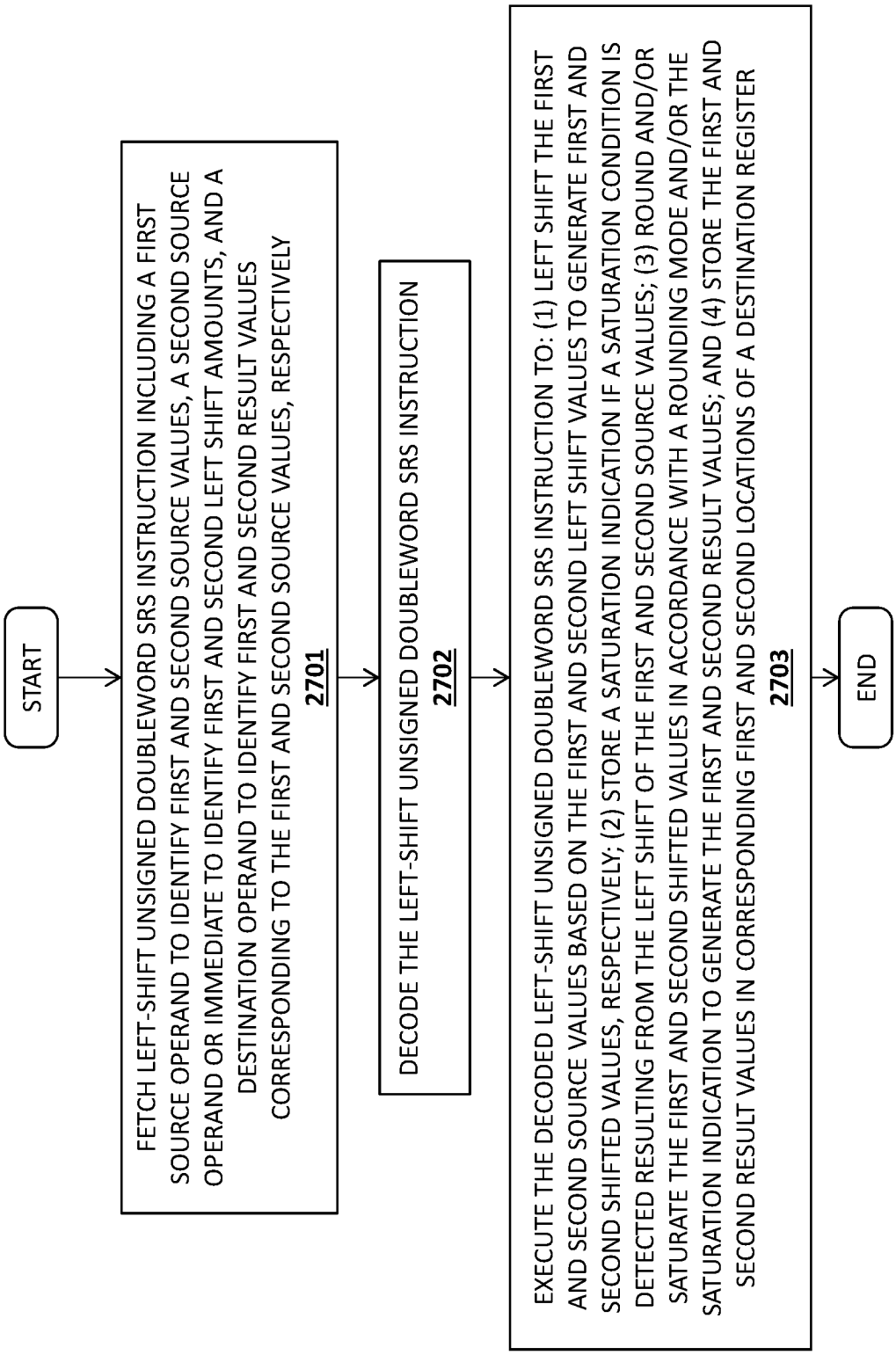

START
FETCH LEFT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY FIRST AND SECOND LEFT SHIFT AMOUNTS, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2701
DECODE THE LEFT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION
2702
EXECUTE THE DECODED LEFT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) LEFT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE FIRST AND SECOND LEFT SHIFT VALUES TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) STORE A SATURATION INDICATION IF A SATURATION CONDITION IS DETECTED RESULTING FROM THE LEFT SHIFT OF THE FIRST AND SECOND SOURCE VALUES; (3) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR THE SATURATION INDICATION TO GENERATE THE FIRST AND SECOND RESULT VALUES; AND (4) STORE THE FIRST AND SECOND RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2703
END

FIG. 27

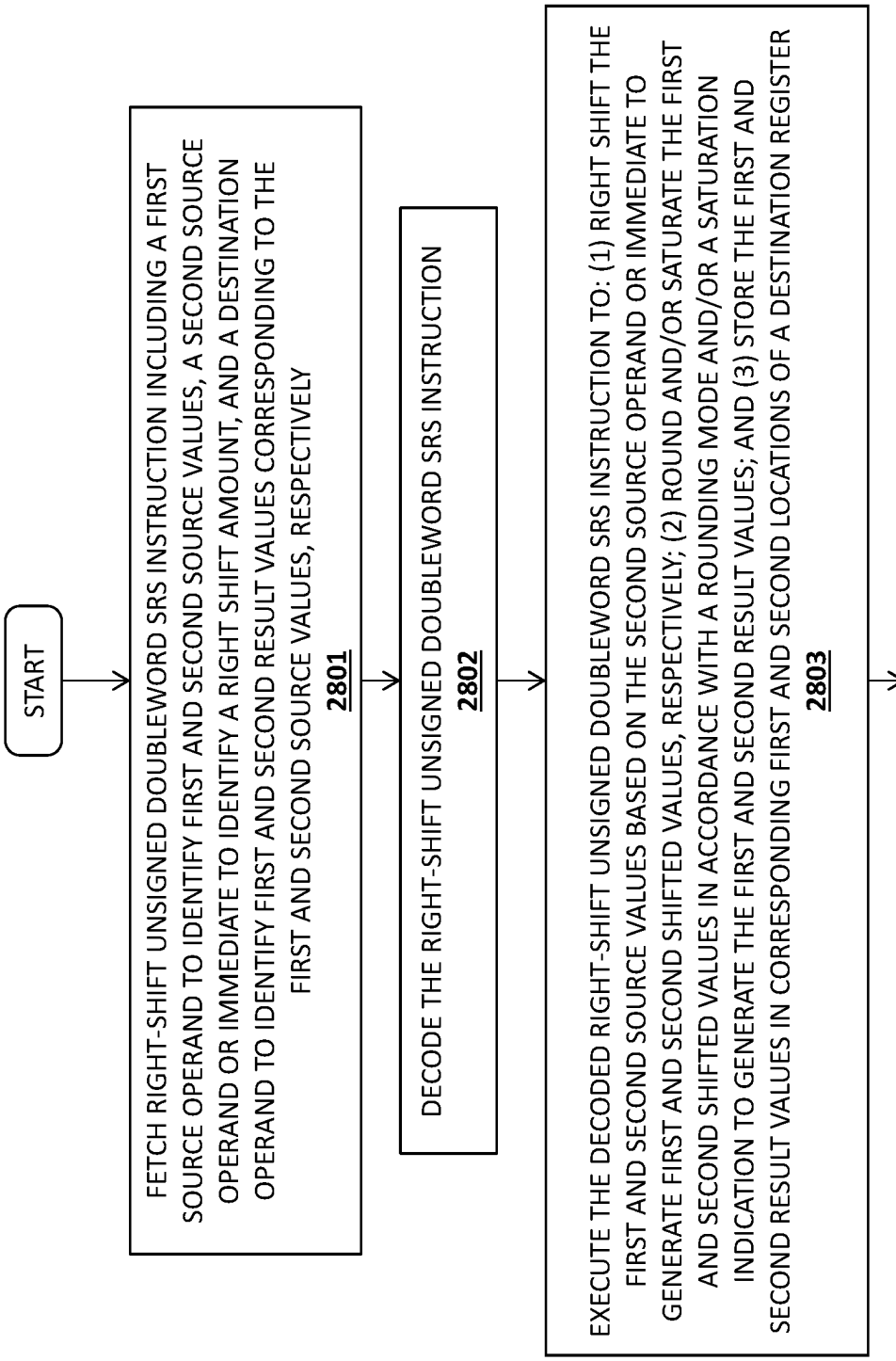

START
FETCH RIGHT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY A RIGHT SHIFT AMOUNT, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2801
DECODE THE RIGHT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION
2802
EXECUTE THE DECODED RIGHT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) RIGHT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE SECOND SOURCE OPERAND OR IMMEDIATE TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR A SATURATION INDICATION TO GENERATE THE FIRST AND SECOND RESULT VALUES; AND (3) STORE THE FIRST AND SECOND RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2803
END

FIG. 28

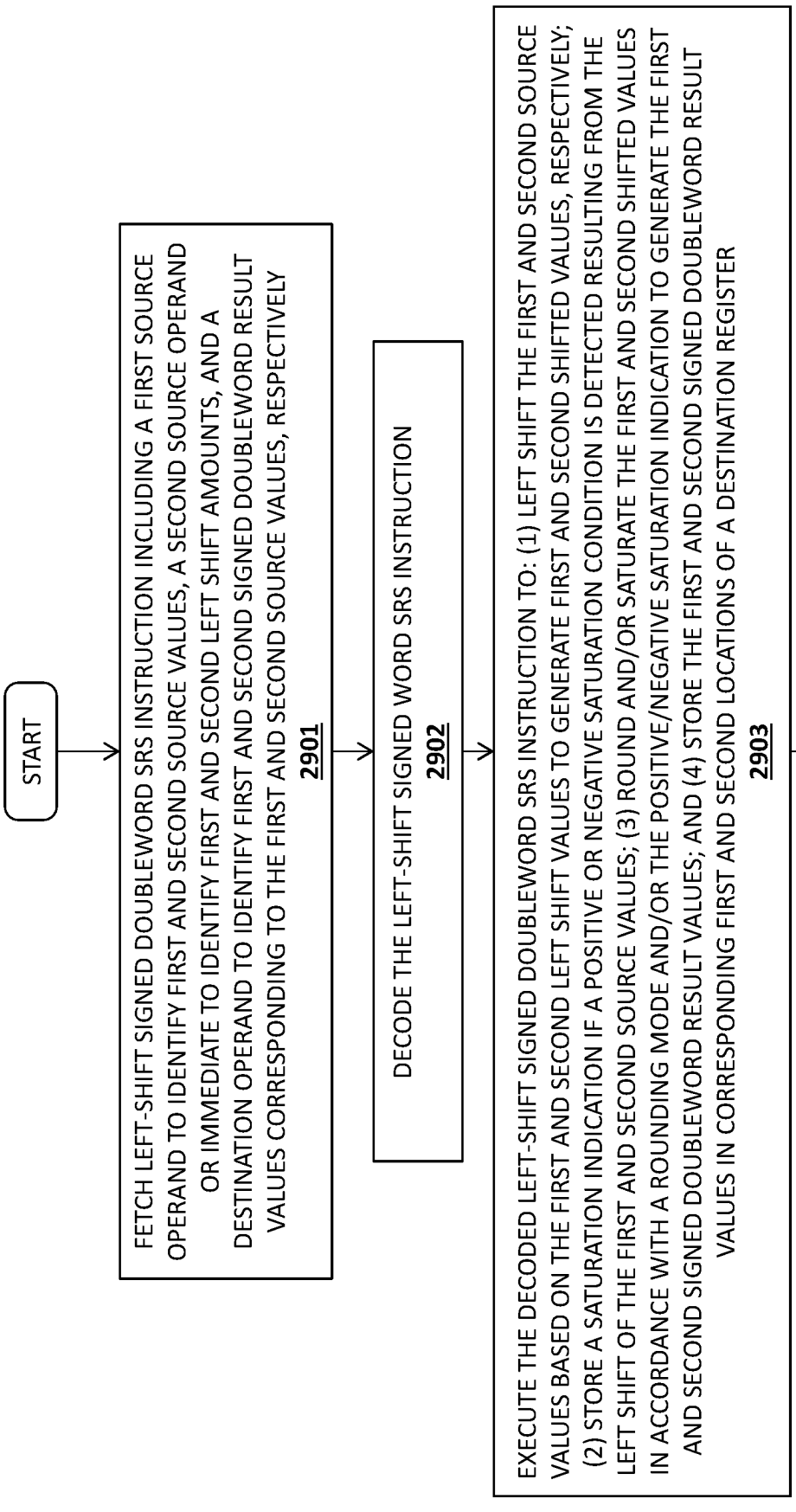

START
FETCH LEFT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY FIRST AND SECOND LEFT SHIFT AMOUNTS, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND SIGNED DOUBLEWORD RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
2901
DECODE THE LEFT-SHIFT SIGNED WORD SRS INSTRUCTION
2902
EXECUTE THE DECODED LEFT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) LEFT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE FIRST AND SECOND LEFT SHIFT VALUES TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) STORE A SATURATION INDICATION IF A POSITIVE OR NEGATIVE SATURATION CONDITION IS DETECTED RESULTING FROM THE LEFT SHIFT OF THE FIRST AND SECOND SOURCE VALUES; (3) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR THE POSITIVE/NEGATIVE SATURATION INDICATION TO GENERATE THE FIRST AND SECOND SIGNED DOUBLEWORD RESULT VALUES; AND (4) STORE THE FIRST AND SECOND SIGNED DOUBLEWORD RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
2903
END

FIG. 29

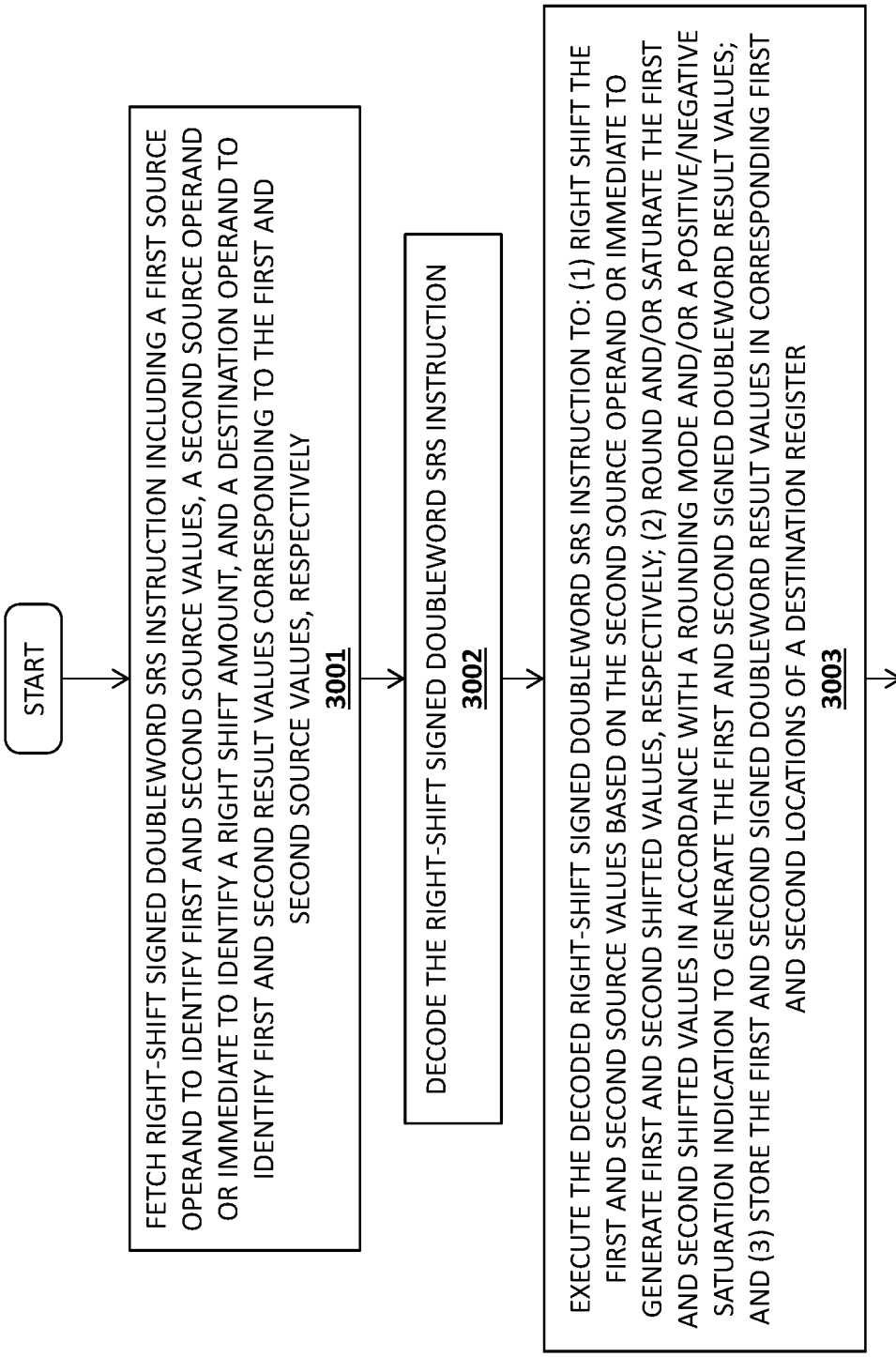

START
FETCH RIGHT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST AND SECOND SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY A RIGHT SHIFT AMOUNT, AND A DESTINATION OPERAND TO IDENTIFY FIRST AND SECOND RESULT VALUES CORRESPONDING TO THE FIRST AND SECOND SOURCE VALUES, RESPECTIVELY
3001
DECODE THE RIGHT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION
3002
EXECUTE THE DECODED RIGHT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) RIGHT SHIFT THE FIRST AND SECOND SOURCE VALUES BASED ON THE SECOND SOURCE OPERAND OR IMMEDIATE TO GENERATE FIRST AND SECOND SHIFTED VALUES, RESPECTIVELY; (2) ROUND AND/OR SATURATE THE FIRST AND SECOND SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR A POSITIVE/NEGATIVE SATURATION INDICATION TO GENERATE THE FIRST AND SECOND SIGNED DOUBLEWORD RESULT VALUES; AND (3) STORE THE FIRST AND SECOND SIGNED DOUBLEWORD RESULT VALUES IN CORRESPONDING FIRST AND SECOND LOCATIONS OF A DESTINATION REGISTER
3003
END

FIG. 30

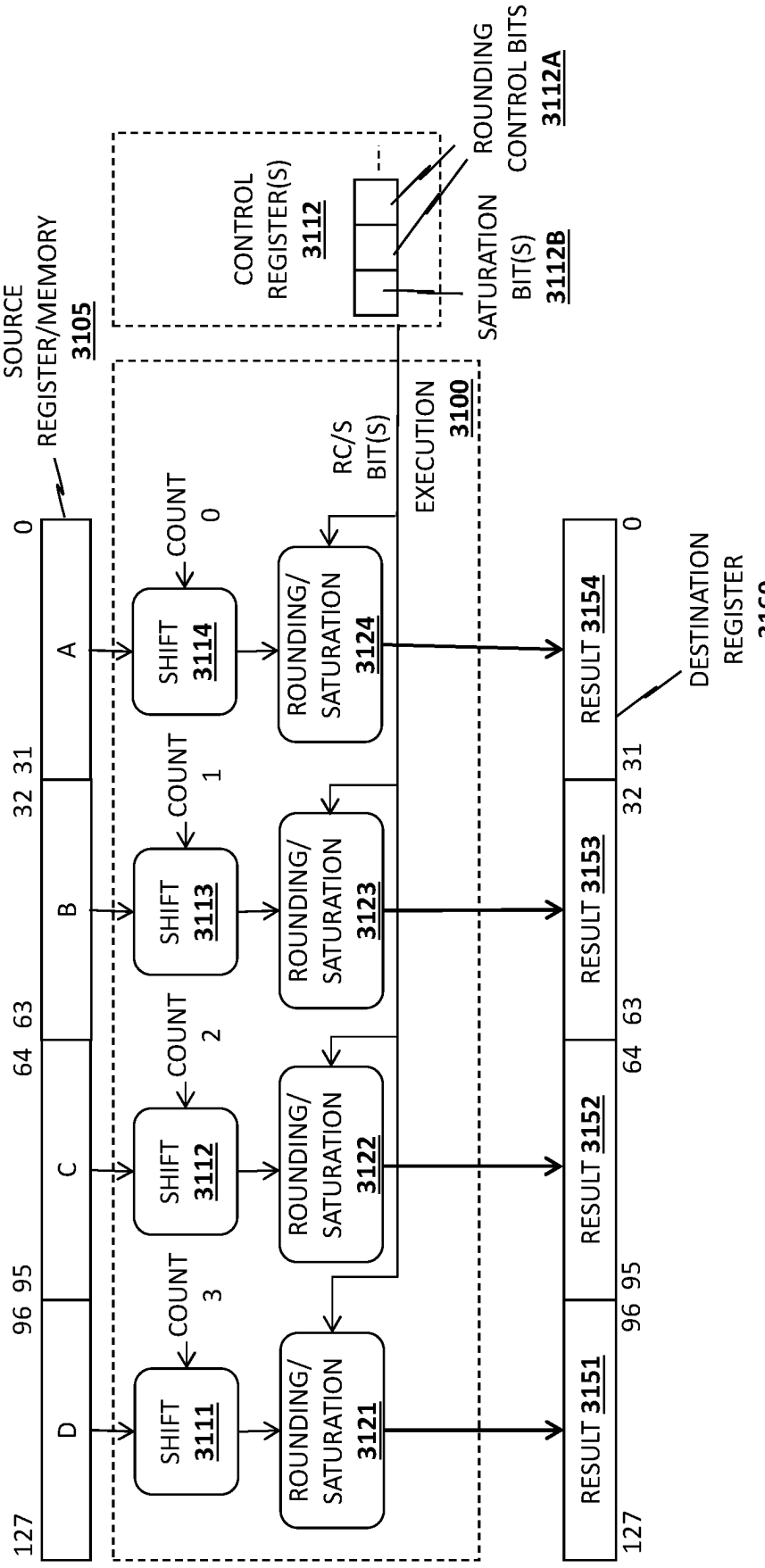

SOURCE REGISTER/MEMORY 3105
CONTROL REGISTER(S) 3112
ROUNDING CONTROL BITS 3112A
SATURATION BIT(S) 3112B
EXECUTION 3100
RC/S BIT(S)
DESTINATION REGISTER 3160
0
31
32
63
64
95
96
127
A
B
C
D
SHIFT 3114
SHIFT 3113
SHIFT 3112
SHIFT 3111
COUNT 0
COUNT 1
COUNT 2
COUNT 3
ROUNDING/ SATURATION 3124
ROUNDING/ SATURATION 3123
ROUNDING/ SATURATION 3122
ROUNDING/ SATURATION 3121
RESULT 3154
RESULT 3153
RESULT 3152
RESULT 3151

FIG. 31

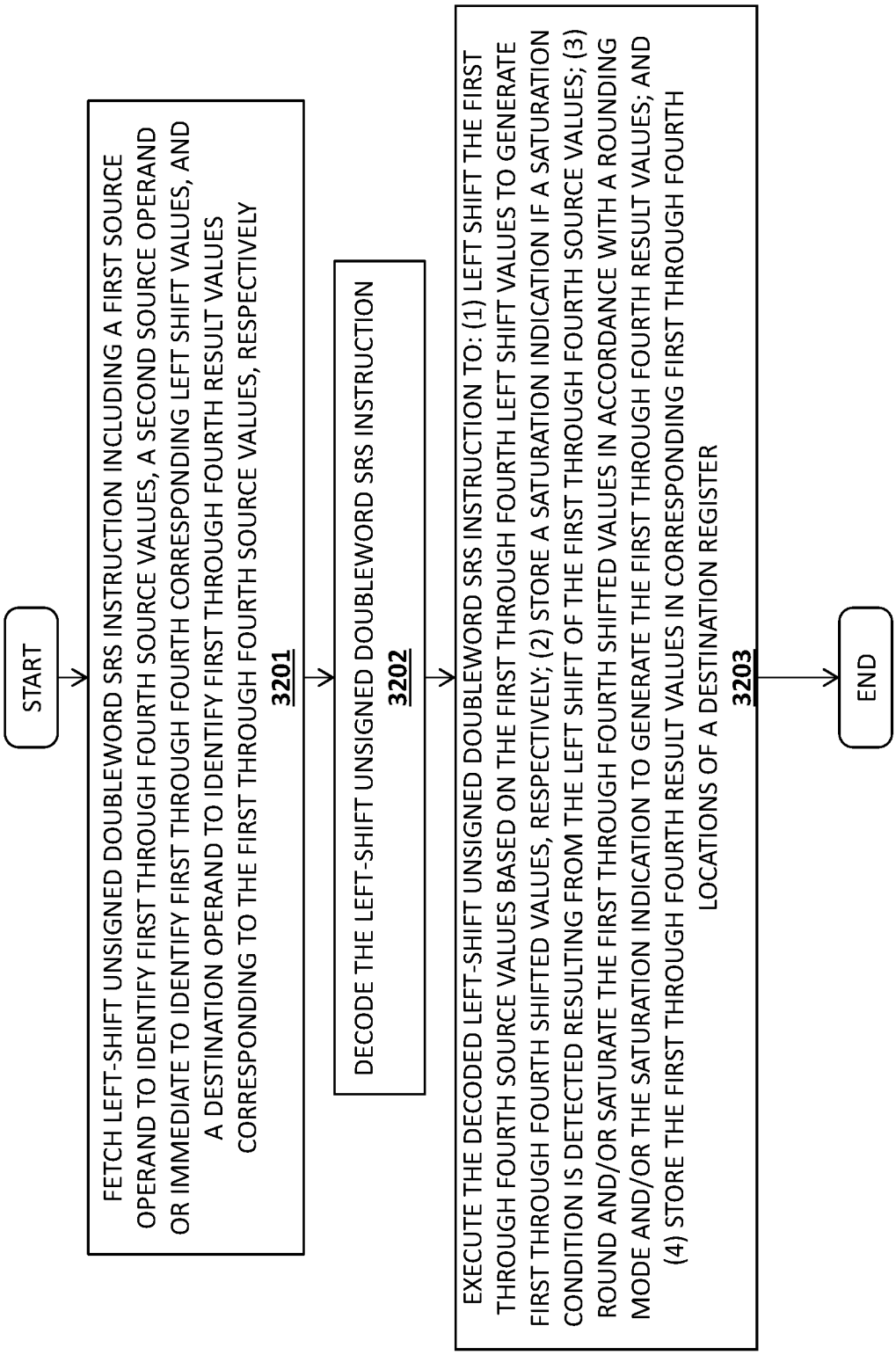

START
FETCH LEFT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST THROUGH FOURTH SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY FIRST THROUGH FOURTH CORRESPONDING LEFT SHIFT VALUES, AND A DESTINATION OPERAND TO IDENTIFY FIRST THROUGH FOURTH RESULT VALUES CORRESPONDING TO THE FIRST THROUGH FOURTH SOURCE VALUES, RESPECTIVELY
3201
DECODE THE LEFT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION
3202
EXECUTE THE DECODED LEFT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) LEFT SHIFT THE FIRST THROUGH FOURTH SOURCE VALUES BASED ON THE FIRST THROUGH FOURTH LEFT SHIFT VALUES TO GENERATE FIRST THROUGH FOURTH SHIFTED VALUES, RESPECTIVELY; (2) STORE A SATURATION INDICATION IF A SATURATION CONDITION IS DETECTED RESULTING FROM THE LEFT SHIFT OF THE FIRST THROUGH FOURTH SOURCE VALUES; (3) ROUND AND/OR SATURATE THE FIRST THROUGH FOURTH SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR THE SATURATION INDICATION TO GENERATE THE FIRST THROUGH FOURTH RESULT VALUES; AND (4) STORE THE FIRST THROUGH FOURTH RESULT VALUES IN CORRESPONDING FIRST THROUGH FOURTH LOCATIONS OF A DESTINATION REGISTER
3203
END

FIG. 32

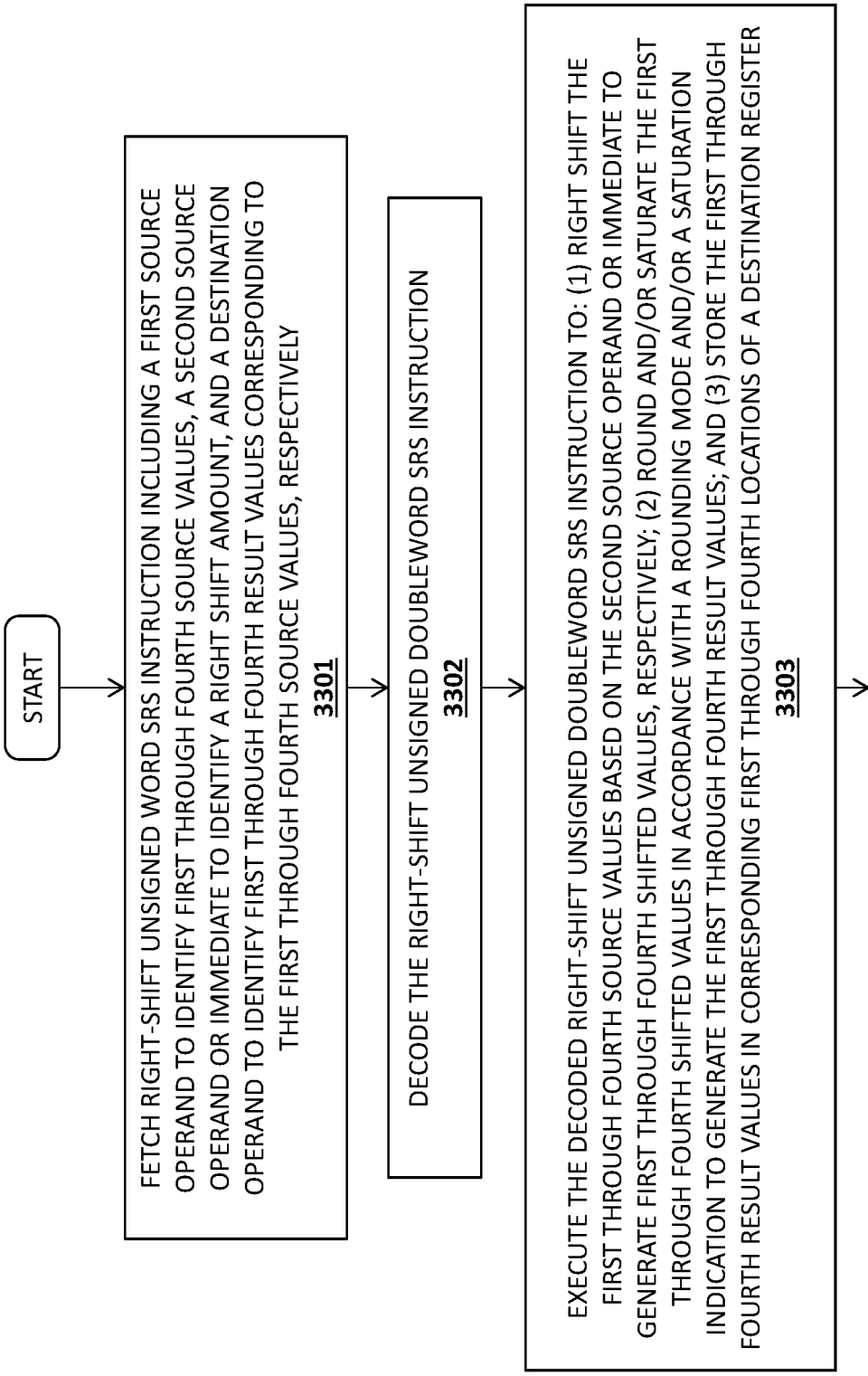

START
FETCH RIGHT-SHIFT UNSIGNED WORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST THROUGH FOURTH SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY A RIGHT SHIFT AMOUNT, AND A DESTINATION OPERAND TO IDENTIFY FIRST THROUGH FOURTH RESULT VALUES CORRESPONDING TO THE FIRST THROUGH FOURTH SOURCE VALUES, RESPECTIVELY
3301
DECODE THE RIGHT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION
3302
EXECUTE THE DECODED RIGHT-SHIFT UNSIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) RIGHT SHIFT THE FIRST THROUGH FOURTH SOURCE VALUES BASED ON THE SECOND SOURCE OPERAND OR IMMEDIATE TO GENERATE FIRST THROUGH FOURTH SHIFTED VALUES, RESPECTIVELY; (2) ROUND AND/OR SATURATE THE FIRST THROUGH FOURTH SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR A SATURATION INDICATION TO GENERATE THE FIRST THROUGH FOURTH RESULT VALUES; AND (3) STORE THE FIRST THROUGH FOURTH RESULT VALUES IN CORRESPONDING FIRST THROUGH FOURTH LOCATIONS OF A DESTINATION REGISTER
3303
END

FIG. 33

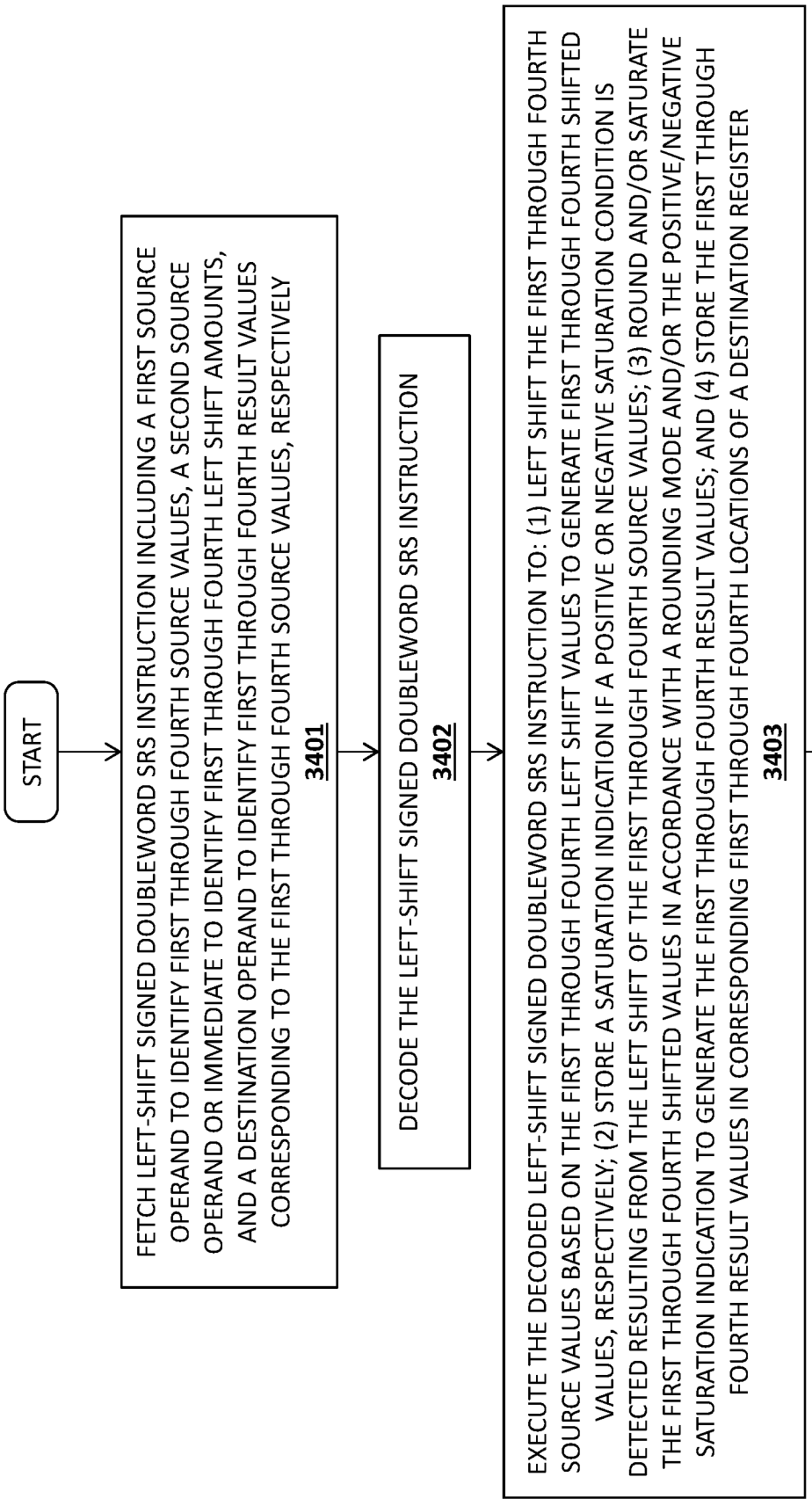

START
FETCH LEFT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST THROUGH FOURTH SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY FIRST THROUGH FOURTH LEFT SHIFT AMOUNTS, AND A DESTINATION OPERAND TO IDENTIFY FIRST THROUGH FOURTH RESULT VALUES CORRESPONDING TO THE FIRST THROUGH FOURTH SOURCE VALUES, RESPECTIVELY
3401
DECODE THE LEFT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION
3402
EXECUTE THE DECODED LEFT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) LEFT SHIFT THE FIRST THROUGH FOURTH SOURCE VALUES BASED ON THE FIRST THROUGH FOURTH LEFT SHIFT VALUES TO GENERATE FIRST THROUGH FOURTH SHIFTED VALUES, RESPECTIVELY; (2) STORE A SATURATION INDICATION IF A POSITIVE OR NEGATIVE SATURATION CONDITION IS DETECTED RESULTING FROM THE LEFT SHIFT OF THE FIRST THROUGH FOURTH SOURCE VALUES; (3) ROUND AND/OR SATURATE THE FIRST THROUGH FOURTH SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR THE POSITIVE/NEGATIVE SATURATION INDICATION TO GENERATE THE FIRST THROUGH FOURTH RESULT VALUES; AND (4) STORE THE FIRST THROUGH FOURTH RESULT VALUES IN CORRESPONDING FIRST THROUGH FOURTH LOCATIONS OF A DESTINATION REGISTER
3403
END

FIG. 34

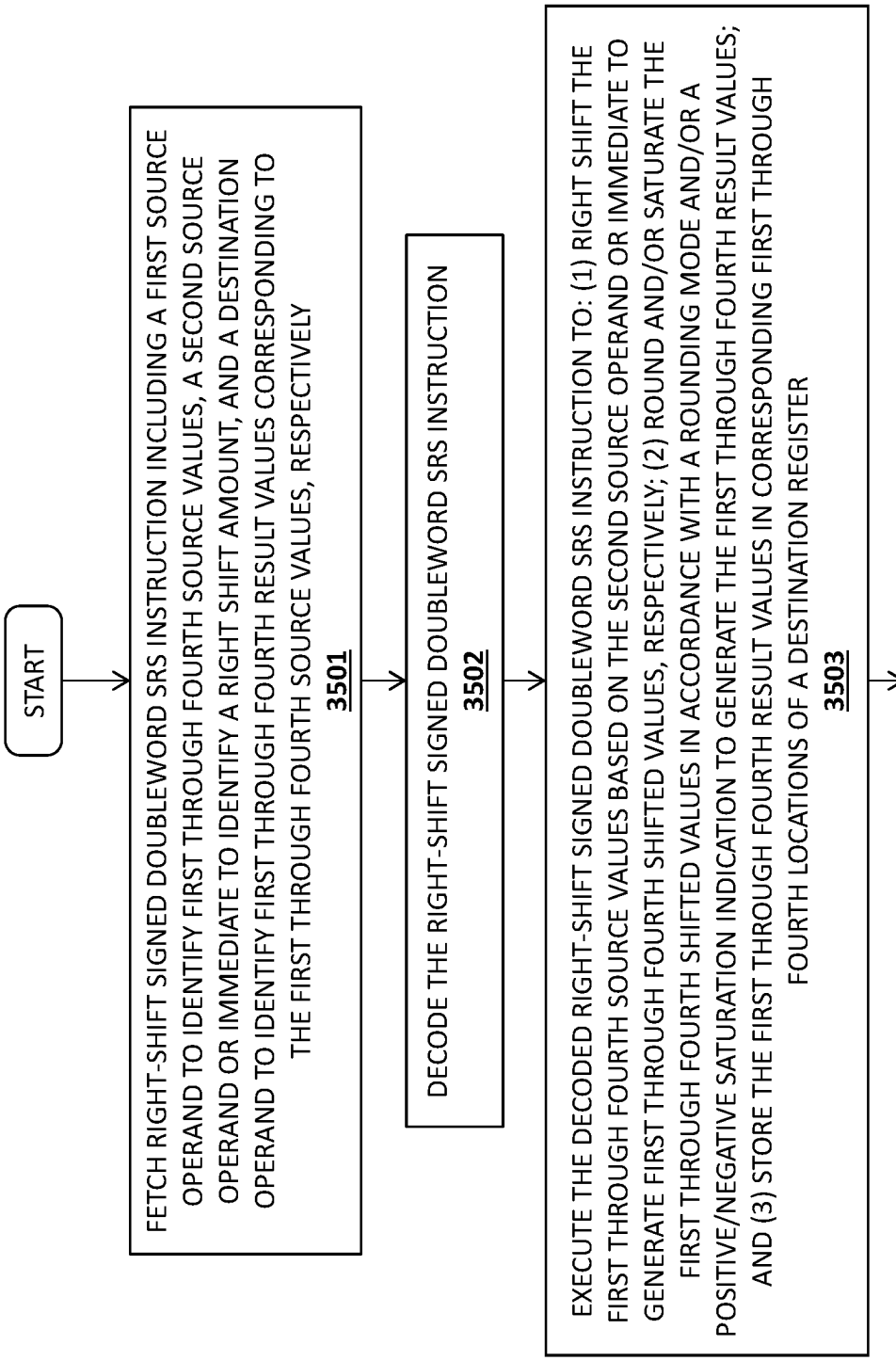

START
FETCH RIGHT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION INCLUDING A FIRST SOURCE OPERAND TO IDENTIFY FIRST THROUGH FOURTH SOURCE VALUES, A SECOND SOURCE OPERAND OR IMMEDIATE TO IDENTIFY A RIGHT SHIFT AMOUNT, AND A DESTINATION OPERAND TO IDENTIFY FIRST THROUGH FOURTH RESULT VALUES CORRESPONDING TO THE FIRST THROUGH FOURTH SOURCE VALUES, RESPECTIVELY
3501
DECODE THE RIGHT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION
3502
EXECUTE THE DECODED RIGHT-SHIFT SIGNED DOUBLEWORD SRS INSTRUCTION TO: (1) RIGHT SHIFT THE FIRST THROUGH FOURTH SOURCE VALUES BASED ON THE SECOND SOURCE OPERAND OR IMMEDIATE TO GENERATE FIRST THROUGH FOURTH SHIFTED VALUES, RESPECTIVELY; (2) ROUND AND/OR SATURATE THE FIRST THROUGH FOURTH SHIFTED VALUES IN ACCORDANCE WITH A ROUNDING MODE AND/OR A POSITIVE/NEGATIVE SATURATION INDICATION TO GENERATE THE FIRST THROUGH FOURTH RESULT VALUES; AND (3) STORE THE FIRST THROUGH FOURTH RESULT VALUES IN CORRESPONDING FIRST THROUGH FOURTH LOCATIONS OF A DESTINATION REGISTER
3503
END

FIG. 35

APPARATUS AND METHOD FOR VECTOR PACKED SIGNED/UNSIGNED SHIFT, ROUND, AND SATURATE

BACKGROUND

Field of the Invention

The embodiments of the invention relate generally to the field of computer processors. More particularly, the embodiments relate to an apparatus and method for vector packed signed/unsigned shift, round, and saturate.

Description of the Related Art

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). It should be noted that the term "instruction" generally refers herein to macro-instructions—that is instructions that are provided to the processor for execution—as opposed to micro-instructions or micro-ops—that is the result of a processor's decoder decoding macro-instructions. The micro-instructions or micro-ops can be configured to instruct an execution unit on the processor to perform operations to implement the logic associated with the macro-instruction.

The ISA is distinguished from the microarchitecture, which is the set of processor design techniques used to implement the instruction set. Processors with different microarchitectures can share a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale Calif. implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. For example, the same register architecture of the ISA may be implemented in different ways in different microarchitectures using well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB) and a retirement register file). Unless otherwise specified, the phrases register architecture, register file, and register are used herein to refer to that which is visible to the software/programmer and the manner in which instructions specify registers. Where a distinction is required, the adjective "logical," "architectural," or "software visible" will be used to indicate registers/files in the register architecture, while different adjectives will be used to designate registers in a given microarchitecture (e.g., physical register, reorder buffer, retirement register, register pool).

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

FIG. **1** illustrates an example computer system architecture;

FIG. **2** illustrates a processor comprising a plurality of cores;

FIG. **3**A illustrates a plurality of stages of a processing pipeline;

FIG. **3**B illustrates details of one embodiment of a core;

FIG. **4** illustrates execution circuitry in accordance with one embodiment;

FIG. **5** illustrates one embodiment of a register architecture;

FIG. **6** illustrates one example of an instruction format;

FIG. **7** illustrates addressing techniques in accordance with one embodiment;

FIG. **8** illustrates one embodiment of an instruction prefix;

FIGS. **9**A-D illustrate embodiments of how the R, X, and B fields of the prefix are used;

FIGS. **10**A-B illustrate examples of a second instruction prefix;

FIG. **11** illustrates payload bytes of one embodiment of an instruction prefix;

FIG. **12** illustrates techniques for executing different instruction set architectures;

FIGS. **13**A-B illustrate embodiments of configured tiles and associated registers/storage;

FIG. **14** illustrates an embodiment of a system utilizing a matrix operations accelerator;

FIGS. **15** and **16** show different embodiments of how memory is shared using a matrix operations accelerator;

FIG. **17** illustrates an example pipeline for executing a matrix multiplication operation;

FIG. **18** illustrates execution circuitry including a processing array;

FIG. **19** illustrates an example of a matrix containing complex values;

FIGS. **20**A-C illustrate different implementations of matrix processing circuitry;

FIG. **21** illustrates an architecture for executing complex matrix multiplication instructions;

FIG. **22** illustrate methods for performing a complex matrix multiplication;

FIG. **23** illustrates an architecture for executing complex matrix transpose and multiplication instructions;

FIG. **24** illustrate methods for performing a complex matrix transpose and multiplication;

FIG. **25** illustrates an architecture for executing a complex matrix conjugation instruction;

FIG. **26** illustrates a method for performing a complex matrix conjugation operation;

FIG. **27** illustrates an architecture for executing complex matrix conjugation and multiplication instructions; and

FIG. **28** illustrates a method for performing a complex matrix conjugation and multiplication.

FIG. **29** illustrates operation of one embodiment of a left-shift signed doubleword SRS instruction;

FIG. **30** illustrates operation of one embodiment of a right-shift signed doubleword SRS instruction;

FIG. **31** illustrates one embodiment for shifting based on count values following by rounding/saturation;

FIG. **32** illustrates operation of one embodiment of a left-shift unsigned doubleword SRS instruction;

FIG. **33** illustrates operation of one embodiment of a right-shift unsigned doubleword SRS instruction;

FIG. **34** illustrates operation of one embodiment of a left-shift signed doubleword SRS instruction;

FIG. **35** illustrates operation of one embodiment of a right-shift signed doubleword SRS instruction.

DETAILED DESCRIPTION

Exemplary Computer Architectures

Detailed below are describes of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs,

personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

FIG. **1** illustrates embodiments of an exemplary system. Multiprocessor system **100** is a point-to-point interconnect system and includes a plurality of processors including a first processor **170** and a second processor **180** coupled via a point-to-point interconnect **150**. In some embodiments, the first processor **170** and the second processor **180** are homogeneous. In some embodiments, first processor **170** and the second processor **180** are heterogenous.

Processors **170** and **180** are shown including integrated memory controller (IMC) units circuitry **172** and **182**, respectively. Processor **170** also includes as part of its interconnect controller units point-to-point (P-P) interfaces **176** and **178**; similarly, second processor **180** includes P-P interfaces **186** and **188**. Processors **170**, **180** may exchange information via the point-to-point (P-P) interconnect **150** using P-P interface circuits **178**, **188**. IMCs **172** and **182** couple the processors **170**, **180** to respective memories, namely a memory **132** and a memory **134**, which may be portions of main memory locally attached to the respective processors.

Processors **170**, **180** may each exchange information with a chipset **190** via individual P-P interconnects **152**, **154** using point to point interface circuits **176**, **194**, **186**, **198**. Chipset **190** may optionally exchange information with a coprocessor **138** via a high-performance interface **192**. In some embodiments, the coprocessor **138** is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

A shared cache (not shown) may be included in either processor **170**, **180** or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

Chipset **190** may be coupled to a first interconnect **116** via an interface **196**. In some embodiments, first interconnect **116** may be a Peripheral Component Interconnect (PCI) interconnect, or an interconnect such as a PCI Express interconnect or another I/O interconnect. In some embodiments, one of the interconnects couples to a power control unit (PCU) **117**, which may include circuitry, software, and/or firmware to perform power management operations with regard to the processors **170**, **180** and/or co-processor **138**. PCU **117** provides control information to a voltage regulator to cause the voltage regulator to generate the appropriate regulated voltage. PCU **117** also provides control information to control the operating voltage generated. In various embodiments, PCU **117** may include a variety of power management logic units (circuitry) to perform hardware-based power management. Such power management may be wholly processor controlled (e.g., by various processor hardware, and which may be triggered by workload and/or power, thermal or other processor constraints) and/or the power management may be performed responsive to external sources (such as a platform or power management source or system software).

PCU **117** is illustrated as being present as logic separate from the processor **170** and/or processor **180**. In other cases, PCU **117** may execute on a given one or more of cores (not shown) of processor **170** or **180**. In some cases, PCU **117** may be implemented as a microcontroller (dedicated or general-purpose) or other control logic configured to execute its own dedicated power management code, sometimes referred to as P-code. In yet other embodiments, power management operations to be performed by PCU **117** may be implemented externally to a processor, such as by way of a separate power management integrated circuit (PMIC) or another component external to the processor. In yet other embodiments, power management operations to be performed by PCU **117** may be implemented within BIOS or other system software.

Various I/O devices **114** may be coupled to first interconnect **116**, along with an interconnect (bus) bridge **118** which couples first interconnect **116** to a second interconnect **120**. In some embodiments, one or more additional processor(s) **115**, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays (FPGAs), or any other processor, are coupled to first interconnect **116**. In some embodiments, second interconnect **120** may be a low pin count (LPC) interconnect. Various devices may be coupled to second interconnect **120** including, for example, a keyboard and/or mouse **122**, communication devices **127** and a storage unit circuitry **128**. Storage unit circuitry **128** may be a disk drive or other mass storage device which may include instructions/code and data **130**, in some embodiments. Further, an audio I/O **124** may be coupled to second interconnect **120**. Note that other architectures than the point-to-point architecture described above are possible. For example, instead of the point-to-point architecture, a system such as multiprocessor system **100** may implement a multi-drop interconnect or other such architecture.

Exemplary Core Architectures, Processors, and Computer Architectures

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die as the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by descriptions of exemplary processors and computer architectures.

FIG. **2** illustrates a block diagram of embodiments of a processor **200** that may have more than one core, may have an integrated memory controller, and may have integrated graphics. The solid lined boxes illustrate a processor **200** with a single core **202**A, a system agent **210**, a set of one or more interconnect controller units circuitry **216**, while the optional addition of the dashed lined boxes illustrates an alternative processor **200** with multiple cores **202**(A)-(N), a set of one or more integrated memory controller unit(s) circuitry **214** in the system agent unit circuitry **210**, and special purpose logic **208**, as well as a set of one or more interconnect controller units circuitry **216**. Note that the processor **200** may be one of the processors **170** or **180**, or co-processor **138** or **115** of FIG. **1**.

Thus, different implementations of the processor **200** may include: 1) a CPU with the special purpose logic **208** being integrated graphics and/or scientific (throughput) logic (which may include one or more cores, not shown), and the cores **202**(A)-(N) being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, or a combination of the two); 2) a coprocessor with the cores **202**(A)-(N) being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores **202**(A)-(N) being a large number of general purpose in-order cores. Thus, the processor **200** may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit circuitry), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor **200** may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

A memory hierarchy includes one or more levels of cache unit(s) circuitry **204**(A)-(N) within the cores **202**(A)-(N), a set of one or more shared cache units circuitry **206**, and external memory (not shown) coupled to the set of integrated memory controller units circuitry **214**. The set of one or more shared cache units circuitry **206** may include one or more mid level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, such as a last level cache (LLC), and/or combinations thereof. While in some embodiments ring-based interconnect network circuitry **212** interconnects the special purpose logic **208** (e.g., integrated graphics logic), the set of shared cache units circuitry **206**, and the system agent unit circuitry **210**, alternative embodiments use any number of well-known techniques for interconnecting such units. In some embodiments, coherency is maintained between one or more of the shared cache units circuitry **206** and cores **202**(A)-(N).

In some embodiments, one or more of the cores **202**(A)-(N) are capable of multi-threading. The system agent unit circuitry **210** includes those components coordinating and operating cores **202**(A)-(N). The system agent unit circuitry **210** may include, for example, power control unit (PCU) circuitry and/or display unit circuitry (not shown). The PCU may be or may include logic and components needed for regulating the power state of the cores **202**(A)-(N) and/or the special purpose logic **208** (e.g., integrated graphics logic). The display unit circuitry is for driving one or more externally connected displays.

The cores **202**(A)-(N) may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores **202**(A)-(N) may be capable of executing the same instruction set, while other cores may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Core Architectures

In-Order and Out-of-Order Core Block Diagram

FIG. **3**(A) is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. FIG. **3**(B) is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in FIGS. **3**(A)-(B) illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

In FIG. **3**(A), a processor pipeline **300** includes a fetch stage **302**, an optional length decode stage **304**, a decode stage **306**, an optional allocation stage **308**, an optional renaming stage **310**, a scheduling (also known as a dispatch or issue) stage **312**, an optional register read/memory read stage **314**, an execute stage **316**, a write back/memory write stage **318**, an optional exception handling stage **322**, and an optional commit stage **324**. One or more operations can be performed in each of these processor pipeline stages. For example, during the fetch stage **302**, one or more instructions are fetched from instruction memory, during the decode stage **306**, the one or more fetched instructions may be decoded, addresses (e.g., load store unit (LSU) addresses) using forwarded register ports may be generated, and branch forwarding (e.g., immediate offset or an link register (LR)) may be performed. In one embodiment, the decode stage **306** and the register read/memory read stage **314** may be combined into one pipeline stage. In one embodiment, during the execute stage **316**, the decoded instructions may be executed, LSU address/data pipelining to an Advanced Microcontroller Bus (AHB) interface may be performed, multiply and add operations may be performed, arithmetic operations with branch results may be performed, etc.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline **300** as follows: 1) the instruction fetch **338** performs the fetch and length decoding stages **302** and **304**; 2) the decode unit circuitry **340** performs the decode stage **306**; 3) the rename/allocator unit circuitry **352** performs the allocation stage **308** and renaming stage **310**; 4) the scheduler unit(s) circuitry **356** performs the schedule stage **312**; 5) the physical register file(s) unit(s) circuitry **358** and the memory unit circuitry **370** perform the register read/memory read stage **314**; the execution cluster **360** perform the execute stage **316**; 6) the memory unit circuitry **370** and the physical register file(s) unit(s) circuitry **358** perform the write back/memory write stage **318**; 7) various units (unit circuitry) may be involved in the exception handling stage **322**; and 8) the retirement unit circuitry **354** and the physical register file(s) unit(s) circuitry **358** perform the commit stage **324**.

FIG. **3**(B) shows processor core **390** including front-end unit circuitry **330** coupled to an execution engine unit circuitry **350**, and both are coupled to a memory unit circuitry **370**. The core **390** may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core,

or a hybrid or alternative core type. As yet another option, the core **390** may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit circuitry **330** may include branch prediction unit circuitry **332** coupled to an instruction cache unit circuitry **334**, which is coupled to an instruction translation lookaside buffer (TLB) **336**, which is coupled to instruction fetch unit circuitry **338**, which is coupled to decode unit circuitry **340**. In one embodiment, the instruction cache unit circuitry **334** is included in the memory unit circuitry **370** rather than the front-end unit circuitry **330**. The decode unit circuitry **340** (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit circuitry **340** may further include an address generation unit circuitry (AGU, not shown). In one embodiment, the AGU generates an LSU address using forwarded register ports, and may further perform branch forwarding (e.g., immediate offset branch forwarding, LR register branch forwarding, etc.). The decode unit circuitry **340** may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core **390** includes a microcode ROM (not shown) or other medium that stores microcode for certain macroinstructions (e.g., in decode unit circuitry **340** or otherwise within the front end unit circuitry **330**). In one embodiment, the decode unit circuitry **340** includes a micro-operation (micro-op) or operation cache (not shown) to hold/cache decoded operations, micro-tags, or micro-operations generated during the decode or other stages of the processor pipeline **300**. The decode unit circuitry **340** may be coupled to rename/allocator unit circuitry **352** in the execution engine unit circuitry **350**.

The execution engine circuitry **350** includes the rename/allocator unit circuitry **352** coupled to a retirement unit circuitry **354** and a set of one or more scheduler(s) circuitry **356**. The scheduler(s) circuitry **356** represents any number of different schedulers, including reservations stations, central instruction window, etc. In some embodiments, the scheduler(s) circuitry **356** can include arithmetic logic unit (ALU) scheduler/scheduling circuitry, ALU queues, arithmetic generation unit (AGU) scheduler/scheduling circuitry, AGU queues, etc. The scheduler(s) circuitry **356** is coupled to the physical register file(s) circuitry **358**. Each of the physical register file(s) circuitry **358** represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit circuitry **358** includes vector registers unit circuitry, writemask registers unit circuitry, and scalar register unit circuitry. These register units may provide architectural vector registers, vector mask registers, general-purpose registers, etc. The physical register file(s) unit(s) circuitry **358** is overlapped by the retirement unit circuitry **354** (also known as a retire queue or a retirement queue) to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) (ROB(s)) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit circuitry **354** and the physical register file(s) circuitry **358** are coupled to the execution cluster(s) **360**. The execution cluster(s) **360** includes a set of one or more execution units circuitry **362** and a set of one or more memory access circuitry **364**. The execution units circuitry **362** may perform various arithmetic, logic, floating-point or other types of operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating-point, packed integer, packed floating-point, vector integer, vector floating-point). While some embodiments may include a number of execution units or execution unit circuitry dedicated to specific functions or sets of functions, other embodiments may include only one execution unit circuitry or multiple execution units/execution unit circuitry that all perform all functions. The scheduler(s) circuitry **356**, physical register file(s) unit(s) circuitry **358**, and execution cluster(s) **360** are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating-point/packed integer/packed floating-point/vector integer/vector floating-point pipeline, and/or a memory access pipeline that each have their own scheduler circuitry, physical register file(s) unit circuitry, and/or execution cluster—and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) circuitry **364**). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

In some embodiments, the execution engine unit circuitry **350** may perform load store unit (LSU) address/data pipelining to an Advanced Microcontroller Bus (AHB) interface (not shown), and address phase and writeback, data phase load, store, and branches.

The set of memory access circuitry **364** is coupled to the memory unit circuitry **370**, which includes data TLB unit circuitry **372** coupled to a data cache circuitry **374** coupled to a level 2 (L2) cache circuitry **376**. In one exemplary embodiment, the memory access units circuitry **364** may include a load unit circuitry, a store address unit circuit, and a store data unit circuitry, each of which is coupled to the data TLB circuitry **372** in the memory unit circuitry **370**. The instruction cache circuitry **334** is further coupled to a level 2 (L2) cache unit circuitry **376** in the memory unit circuitry **370**. In one embodiment, the instruction cache **334** and the data cache **374** are combined into a single instruction and data cache (not shown) in L2 cache unit circuitry **376**, a level 3 (L3) cache unit circuitry (not shown), and/or main memory. The L2 cache unit circuitry **376** is coupled to one or more other levels of cache and eventually to a main memory.

The core **390** may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set; the ARM instruction set (with optional additional extensions such as NEON)), including the instruction(s) described herein. In one embodiment, the core **390** includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

Exemplary Execution Unit(s) Circuitry

FIG. **4** illustrates embodiments of execution unit(s) circuitry, such as execution unit(s) circuitry **362** of FIG. **3**(B).

As illustrated, execution unit(s) circuitry **362** may include one or more ALU circuits **401**, vector/SIMD unit circuits **403**, load/store unit circuits **405**, and/or branch/jump unit circuits **407**. ALU circuits **401** perform integer arithmetic and/or Boolean operations. Vector/SIMD unit circuits **403** perform vector/SIMD operations on packed data (such as SIMD/vector registers). Load/store unit circuits **405** execute load and store instructions to load data from memory into registers or store from registers to memory. Load/store unit circuits **405** may also generate addresses. Branch/jump unit circuits **407** cause a branch or jump to a memory address depending on the instruction. Floating-point unit (FPU) circuits **409** perform floating-point arithmetic. The width of the execution unit(s) circuitry **362** varies depending upon the embodiment and can range from 16-bit to 1,024-bit. In some embodiments, two or more smaller execution units are logically combined to form a larger execution unit (e.g., two 128-bit execution units are logically combined to form a 256-bit execution unit).

Exemplary Register Architecture

FIG. **5** is a block diagram of a register architecture **500** according to some embodiments. As illustrated, there are vector/SIMD registers **510** that vary from 128-bit to 1,024 bits width. In some embodiments, the vector/SIMD registers **510** are physically 512-bits and, depending upon the mapping, only some of the lower bits are used. For example, in some embodiments, the vector/SIMD registers **510** are ZMM registers which are 512 bits: the lower 256 bits are used for YMM registers and the lower 128 bits are used for XMM registers. As such, there is an overlay of registers. In some embodiments, a vector length field selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length. Scalar operations are operations performed on the lowest order data element position in a ZMM/YMM/XMM register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

In some embodiments, the register architecture **500** includes writemask/predicate registers **515**. For example, in some embodiments, there are 8 writemask/predicate registers (sometimes called k0 through k7) that are each 16-bit, 32-bit, 64-bit, or 128-bit in size. Writemask/predicate registers **515** may allow for merging (e.g., allowing any set of elements in the destination to be protected from updates during the execution of any operation) and/or zeroing (e.g., zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation). In some embodiments, each data element position in a given writemask/predicate register **515** corresponds to a data element position of the destination. In other embodiments, the writemask/predicate registers **515** are scalable and consists of a set number of enable bits for a given vector element (e.g., 8 enable bits per 64-bit vector element).

The register architecture **500** includes a plurality of general-purpose registers **525**. These registers may be 16-bit, 32-bit, 64-bit, etc. and can be used for scalar operations. In some embodiments, these registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

In some embodiments, the register architecture **500** includes scalar floating-point register **545** which is used for scalar floating-point operations on 32/64/80-bit floating-point data using the x87 instruction set extension or as MMX registers to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

One or more flag registers **540** (e.g., EFLAGS, RFLAGS, etc.) store status and control information for arithmetic, compare, and system operations. For example, the one or more flag registers **540** may store condition code information such as carry, parity, auxiliary carry, zero, sign, and overflow. In some embodiments, the one or more flag registers **540** are called program status and control registers.

Segment registers **520** contain segment points for use in accessing memory. In some embodiments, these registers are referenced by the names CS, DS, SS, ES, FS, and GS.

Machine specific registers (MSRs) **535** control and report on processor performance. Most MSRs **535** handle system-related functions and are not accessible to an application program. Machine check registers **560** consist of control, status, and error reporting MSRs that are used to detect and report on hardware errors.

One or more instruction pointer register(s) **530** store an instruction pointer value. Control register(s) **555** (e.g., CR0-CR4) determine the operating mode of a processor (e.g., processor **170**, **180**, **138**, **115**, and/or **200**) and the characteristics of a currently executing task. Debug registers **550** control and allow for the monitoring of a processor or core's debugging operations.

Memory management registers **565** specify the locations of data structures used in protected mode memory management. These registers may include a GDTR, IDRT, task register, and a LDTR register.

Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Instruction Sets

An instruction set architecture (ISA) may include one or more instruction formats. A given instruction format may define various fields (e.g., number of bits, location of bits) to specify, among other things, the operation to be performed (e.g., opcode) and the operand(s) on which that operation is to be performed and/or other data field(s) (e.g., mask). Some instruction formats are further broken down though the definition of instruction templates (or sub-formats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands.

Exemplary Instruction Formats

Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

FIG. **6** illustrates embodiments of an instruction format. As illustrated, an instruction may include multiple components including, but not limited to, one or more fields for:

one or more prefixes **601**, an opcode **603**, addressing information **605** (e.g., register identifiers, memory addressing information, etc.), a displacement value **607**, and/or an immediate **609**. Note that some instructions utilize some or all of the fields of the format whereas others may only use the field for the opcode **603**. In some embodiments, the order illustrated is the order in which these fields are to be encoded, however, it should be appreciated that in other embodiments these fields may be encoded in a different order, combined, etc.

The prefix(es) field(s) **601**, when used, modifies an instruction. In some embodiments, one or more prefixes are used to repeat string instructions (e.g., 0xF0, 0xF2, 0xF3, etc.), to provide section overrides (e.g., 0x2E, 0x36, 0x3E, 0x26, 0x64, 0x65, 0x2E, 0x3E, etc.), to perform bus lock operations, and/or to change operand (e.g., 0x66) and address sizes (e.g., 0x67). Certain instructions require a mandatory prefix (e.g., 0x66, 0xF2, 0xF3, etc.). Certain of these prefixes may be considered "legacy" prefixes. Other prefixes, one or more examples of which are detailed herein, indicate, and/or provide further capability, such as specifying particular registers, etc. The other prefixes typically follow the "legacy" prefixes.

The opcode field **603** is used to at least partially define the operation to be performed upon a decoding of the instruction. In some embodiments, a primary opcode encoded in the opcode field **603** is 1, 2, or 3 bytes in length. In other embodiments, a primary opcode can be a different length. An additional 3-bit opcode field is sometimes encoded in another field.

The addressing field **605** is used to address one or more operands of the instruction, such as a location in memory or one or more registers. FIG. **7** illustrates embodiments of the addressing field **605**. In this illustration, an optional ModR/M byte **702** and an optional Scale, Index, Base (SIB) byte **704** are shown. The ModR/M byte **702** and the SIB byte **704** are used to encode up to two operands of an instruction, each of which is a direct register or effective memory address. Note that each of these fields are optional in that not all instructions include one or more of these fields. The MOD R/M byte **702** includes a MOD field **742**, a register field **744**, and R/M field **746**.

The content of the MOD field **742** distinguishes between memory access and non-memory access modes. In some embodiments, when the MOD field **742** has a value of b11, a register-direct addressing mode is utilized, and otherwise register-indirect addressing is used.

The register field **744** may encode either the destination register operand or a source register operand, or may encode an opcode extension and not be used to encode any instruction operand. The content of register index field **744**, directly or through address generation, specifies the locations of a source or destination operand (either in a register or in memory). In some embodiments, the register field **744** is supplemented with an additional bit from a prefix (e.g., prefix **601**) to allow for greater addressing.

The R/M field **746** may be used to encode an instruction operand that references a memory address, or may be used to encode either the destination register operand or a source register operand. Note the R/M field **746** may be combined with the MOD field **742** to dictate an addressing mode in some embodiments.

The SIB byte **704** includes a scale field **752**, an index field **754**, and a base field **756** to be used in the generation of an address. The scale field **752** indicates scaling factor. The index field **754** specifies an index register to use. In some embodiments, the index field **754** is supplemented with an additional bit from a prefix (e.g., prefix **601**) to allow for greater addressing. The base field **756** specifies a base register to use. In some embodiments, the base field **756** is supplemented with an additional bit from a prefix (e.g., prefix **601**) to allow for greater addressing. In practice, the content of the scale field **752** allows for the scaling of the content of the index field **754** for memory address generation (e.g., for address generation that uses 2^{scale}*index+base).

Some addressing forms utilize a displacement value to generate a memory address. For example, a memory address may be generated according to 2^{scale}*index+base+displacement, index*scale+displacement, r/m+displacement, instruction pointer (RIP/EIP)+displacement, register+displacement, etc. The displacement may be a 1-byte, 2-byte, 4-byte, etc. value. In some embodiments, a displacement field **607** provides this value. Additionally, in some embodiments, a displacement factor usage is encoded in the MOD field of the addressing field **605** that indicates a compressed displacement scheme for which a displacement value is calculated by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of a b bit, and the input element size of the instruction. The displacement value is stored in the displacement field **607**.

In some embodiments, an immediate field **609** specifies an immediate for the instruction. An immediate may be encoded as a 1-byte value, a 2-byte value, a 4-byte value, etc.

FIG. **8** illustrates embodiments of a first prefix **601**(A). In some embodiments, the first prefix **601**(A) is an embodiment of a REX prefix. Instructions that use this prefix may specify general purpose registers, 64-bit packed data registers (e.g., single instruction, multiple data (SIMD) registers or vector registers), and/or control registers and debug registers (e.g., CR8-CR15 and DR8-DR15).

Instructions using the first prefix **601**(A) may specify up to three registers using 3-bit fields depending on the format: 1) using the reg field **744** and the R/M field **746** of the Mod R/M byte **702**; 2) using the Mod R/M byte **702** with the SIB byte **704** including using the reg field **744** and the base field **756** and index field **754**; or 3) using the register field of an opcode.

In the first prefix **601**(A), bit positions 7:4 are set as 0100. Bit position 3 (W) can be used to determine the operand size, but may not solely determine operand width. As such, when W=0, the operand size is determined by a code segment descriptor (CS.D) and when W=1, the operand size is 64-bit.

Note that the addition of another bit allows for 16 (2^4) registers to be addressed, whereas the MOD R/M reg field **744** and MOD R/M R/M field **746** alone can each only address 8 registers.

In the first prefix **601**(A), bit position 2 (R) may an extension of the MOD R/M reg field **744** and may be used to modify the ModR/M reg field **744** when that field encodes a general purpose register, a 64-bit packed data register (e.g., a SSE register), or a control or debug register. R is ignored when Mod R/M byte **702** specifies other registers or defines an extended opcode.

Bit position 1 (X) X bit may modify the SIB byte index field **754**.

Bit position B (B) B may modify the base in the Mod R/M R/M field **746** or the SIB byte base field **756**; or it may modify the opcode register field used for accessing general purpose registers (e.g., general purpose registers **525**).

FIGS. **9**(A)-(D) illustrate embodiments of how the R, X, and B fields of the first prefix **601**(A) are used. FIG. **9**(A)

illustrates R and B from the first prefix **601**(A) being used to extend the reg field **744** and R/M field **746** of the MOD R/M byte **702** when the SIB byte **704** is not used for memory addressing. FIG. **9**(B) illustrates R and B from the first prefix **601**(A) being used to extend the reg field **744** and R/M field **746** of the MOD R/M byte **702** when the SIB byte **7 04** is not used (register-register addressing). FIG. **9**(C) illustrates R, X, and B from the first prefix **601**(A) being used to extend the reg field **744** of the MOD R/M byte **702** and the index field **754** and base field **756** when the SIB byte **7 04** being used for memory addressing. FIG. **9**(D) illustrates B from the first prefix **601**(A) being used to extend the reg field **744** of the MOD R/M byte **702** when a register is encoded in the opcode **603**.

FIGS. **10**(A)-(B) illustrate embodiments of a second prefix **601**(B). In some embodiments, the second prefix **601**(B) is an embodiment of a VEX prefix. The second prefix **601**(B) encoding allows instructions to have more than two operands, and allows SIMD vector registers (e.g., vector/SIMD registers **510**) to be longer than 64-bits (e.g., 128-bit and 256-bit). The use of the second prefix **601**(B) provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as A=A+B, which overwrites a source operand. The use of the second prefix **601**(B) enables operands to perform nondestructive operations such asA=B+C.

In some embodiments, the second prefix **601**(B) comes in two forms—a two-byte form and a three-byte form. The two-byte second prefix **601**(B) is used mainly for 128-bit, scalar, and some 256-bit instructions; while the three-byte second prefix **601**(B) provides a compact replacement of the first prefix **601**(A) and 3-byte opcode instructions.

FIG. **10**(A) illustrates embodiments of a two-byte form of the second prefix **601**(B). In one example, a format field **1001** (byte 0 **1003**) contains the value C5H. In one example, byte 1 **1005** includes a "R" value in bit[7]. This value is the complement of the same value of the first prefix **601**(A). Bit[2] is used to dictate the length (L) of the vector (where a value of 0 is a scalar or 128-bit vector and a value of 1 is a 256-bit vector). Bits[1:0] provide opcode extensionality equivalent to some legacy prefixes (e.g., 00=no prefix, 01=66H, 10=F3H, and 11=F2H). Bits[6:3] shown as vvvv may be used to: 1) encode the first source register operand, specified in inverted (is complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in is complement form for certain vector shifts; or 3) not encode any operand, the field is reserved and should contain a certain value, such as 1111b.

Instructions that use this prefix may use the Mod R/M R/M field **746** to encode the instruction operand that references a memory address or encode either the destination register operand or a source register operand.

Instructions that use this prefix may use the Mod R/M reg field **744** to encode either the destination register operand or a source register operand, be treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, vvvv, the Mod R/M R/M field **746** and the Mod R/M reg field **744** encode three of the four operands. Bits[7:4] of the immediate **609** are then used to encode the third source register operand.

FIG. **10**(B) illustrates embodiments of a three-byte form of the second prefix **601**(B). In one example, a format field **1011** (byte 0 **1013**) contains the value C4H. Byte 1 **1015** includes in bits[7:5] "R," "X," and "B" which are the complements of the same values of the first prefix **601**(A). Bits[4:0] of byte 1 **1015** (shown as mmmmm) include content to encode, as need, one or more implied leading opcode bytes. For example, 00001 implies a 0FH leading opcode, 00010 implies a 0F38H leading opcode, 00011 implies a leading 0F3AH opcode, etc.

Bit[7] of byte 2 **1017** is used similar to W of the first prefix **601**(A) including helping to determine promotable operand sizes. Bit[2] is used to dictate the length (L) of the vector (where a value of 0 is a scalar or 128-bit vector and a value of 1 is a 256-bit vector). Bits[1:0] provide opcode extensionality equivalent to some legacy prefixes (e.g., 00=no prefix, 01=66H, 10=F3H, and 11=F2H). Bits[6:3], shown as vvvv, may be used to: 1) encode the first source register operand, specified in inverted (is complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in is complement form for certain vector shifts; or 3) not encode any operand, the field is reserved and should contain a certain value, such as 1111b.

Instructions that use this prefix may use the Mod R/M R/M field **746** to encode the instruction operand that references a memory address or encode either the destination register operand or a source register operand.

Instructions that use this prefix may use the Mod R/M reg field **744** to encode either the destination register operand or a source register operand, be treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, vvvv, the Mod R/M R/M field **746**, and the Mod R/M reg field **744** encode three of the four operands. Bits[7:4] of the immediate **609** are then used to encode the third source register operand.

FIG. **11** illustrates embodiments of a third prefix **601**(C). In some embodiments, the first prefix **601**(A) is an embodiment of an EVEX prefix. The third prefix **601**(C) is a four-byte prefix.

The third prefix **601**(C) can encode 32 vector registers (e.g., 128-bit, 256-bit, and 512-bit registers) in 64-bit mode. In some embodiments, instructions that utilize a writemask/opmask (see discussion of registers in a previous figure, such as FIG. **5**) or predication utilize this prefix. Opmask register allow for conditional processing or selection control. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the second prefix **601**(B).

The third prefix **601**(C) may encode functionality that is specific to instruction classes (e.g., a packed instruction with "load+op" semantic can support embedded broadcast functionality, a floating-point instruction with rounding semantic can support static rounding functionality, a floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality, etc.).

The first byte of the third prefix **601**(C) is a format field **1111** that has a value, in one example, of 62H. Subsequent bytes are referred to as payload bytes **1115-1119** and collectively form a 24-bit value of P[23:0] providing specific capability in the form of one or more fields (detailed herein).

In some embodiments, P[1:0] of payload byte **1119** are identical to the low two mmmmm bits. P[3:2] are reserved in some embodiments. Bit P[4] (R') allows access to the high 16 vector register set when combined with P[7] and the ModR/M reg field **744**. P[6] can also provide access to a high 16 vector register when SIB-type addressing is not needed. P[7:5] consist of an R, X, and B which are operand specifier modifier bits for vector register, general purpose register, memory addressing and allow access to the next set

of 8 registers beyond the low 8 registers when combined with the ModR/M register field **744** and ModR/M R/M field **746**. P[9:8] provide opcode extensionality equivalent to some legacy prefixes (e.g., 00=no prefix, 01=66H, 10=F3H, and 11=F2H). P[10] in some embodiments is a fixed value of 1. P[14:11], shown as vvvv, may be used to: 1) encode the first source register operand, specified in inverted (is complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in is complement form for certain vector shifts; or 3) not encode any operand, the field is reserved and should contain a certain value, such as 1111b.

P[15] is similar to W of the first prefix **601**(A) and second prefix **611**(B) and may serve as an opcode extension bit or operand size promotion.

P[18:16] specify the index of a register in the opmask (writemask) registers (e.g., writemask/predicate registers **515**). In one embodiment of the invention, the specific value aaa=000 has a special behavior implying no opmask is used for the particular instruction (this may be implemented in a variety of ways including the use of a opmask hardwired to all ones or hardware that bypasses the masking hardware). When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the opmask field allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the opmask field's content selects one of a number of opmask registers that contains the opmask to be used (and thus the opmask field's content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's content to directly specify the masking to be performed.

P[19] can be combined with P[14:11] to encode a second source vector register in a non-destructive source syntax which can access an upper 16 vector registers using P[19]. P[20] encodes multiple functionalities, which differs across different classes of instructions and can affect the meaning of the vector length/rounding control specifier field (P[22:21]). P[23] indicates support for merging-writemasking (e.g., when set to 0) or support for zeroing and merging-writemasking (e.g., when set to 1).

Exemplary embodiments of encoding of registers in instructions using the third prefix **601**(C) are detailed in the following tables.

TABLE 1

32-Register Support in 64-bit Mode

	4	3	[2:0]	REG. TYPE	COMMON USAGES
REG	R'	R	ModR/M reg	GPR, Vector	Destination or Source
VVVV	V'		vvvv	GPR, Vector	2nd Source or Destination
RM	X	B	ModR/M R/M	GPR, Vector	1st Source or Destination
BASE	0	B	ModR/M R/M	GPR	Memory addressing
INDEX	0	X	SIB.index	GPR	Memory addressing
VIDX	V'	X	SIB.index	Vector	VSIB memory addressing

TABLE 2

Encoding Register Specifiers in 32-bit Mode

	[2:0]	REG. TYPE	COMMON USAGES
REG	ModR/M reg	GPR, Vector	Destination or Source
VVVV	vvvv	GPR, Vector	2^{nd} Source or Destination
RM	ModR/M R/M	GPR, Vector	1^{st} Source or Destination
BASE	ModR/M R/M	GPR	Memory addressing
INDEX	SIB.index	GPR	Memory addressing
VIDX	SIB.index	Vector	VSIB memory addressing

TABLE 3

Opmask Register Specifier Encoding

	[2:0]	REG. TYPE	COMMON USAGES
REG	ModR/M Reg	k0-k7	Source
VVVV	vvvv	k0-k7	2^{nd} Source
RM	ModR/M R/M	k0-7	1^{st} Source
{k1}	aaa	$k0^1$-k7	Opmask

Program code may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example, a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The program code may be implemented in a high-level procedural or object-oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as "IP cores"

may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

Emulation (Including Binary Translation, Code Morphing, Etc.)

In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

FIG. **12** illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIG. **12** shows a program in a high level language **1202** may be compiled using a first ISA compiler **1204** to generate first ISA binary code **1206** that may be natively executed by a processor with at least one first instruction set core **1216**. The processor with at least one first ISA instruction set core **1216** represents any processor that can perform substantially the same functions as an Intel® processor with at least one first ISA instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the first ISA instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one first ISA instruction set core, in order to achieve substantially the same result as a processor with at least one first ISA instruction set core. The first ISA compiler **1204** represents a compiler that is operable to generate first ISA binary code **1206** (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one first ISA instruction set core **1216**.

Similarly, FIG. **12** shows the program in the high level language **1202** may be compiled using an alternative instruction set compiler **1208** to generate alternative instruction set binary code **1210** that may be natively executed by a processor without a first ISA instruction set core **1214**. The instruction converter **1212** is used to convert the first ISA binary code **1206** into code that may be natively executed by the processor without a first ISA instruction set core **1214**. This converted code is not likely to be the same as the alternative instruction set binary code **1210** because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter **1212** represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have a first ISA instruction set processor or core to execute the first ISA binary code **1206**.

Exemplary Tile/Matrix Operations and Hardware

FIG. **13**A illustrates an embodiment of configured tiles. As shown, there are four tiles **1304**, **1306**, **1308**, and **1310** that are loaded from application memory **1302**. In this example, tiles T0 **1304** and T1 **1306** have K rows and N columns with 4 element bytes (e.g., single precision data). Tiles T2 **1308** and T3 **1310** have M rows and N/2 columns with 8 element bytes (e.g., double precision data). As the double precision operands are twice the width of single precision, this configuration is consistent with a palette, used to provide tile options, supplying at least 4 names with total storage of 16*N*M bytes. Depending upon the instruction encoding scheme used, the number of tiles available varies.

In some embodiments, tile parameters are definable. For example, a "palette" is used to provide tile options. Exemplary options include, but are not limited to: the number of tile names, the number of bytes in a row of storage, the number of rows and columns in a tile, etc. For example, a maximum "height" (number of rows) of a tile may be defined as: Tile Max Rows=Architected Storage/(The Number of Palette Names*The Number of Bytes per row)

As such, an application can be written such that a fixed usage of names will be able to take advantage of different storage sizes across implementations.

Configuration of tiles is done using a tile configuration instruction ("TILECONFIG"), where a particular tile usage is defined in a selected palette. This declaration includes the number of tile names to be used, the requested number of rows and columns per name (tile), and, in some embodiments, the requested datatype of each tile. In some embodiments, consistency checks are performed during the execution of a TILECONFIG instruction to determine that it matches the restrictions of the palette entry.

FIG. **13**B illustrates several examples of matrix storage. In (A), a tile is stored in memory. As shown, each "row" consists of four packed data elements. To get to the next "row," a stride value is used. Note that rows may be consecutively stored in memory. Strided memory accesses allows for access of one row to then next when the tile storage does not map the underlying memory array row width.

Tile loads from memory and stores to memory are typically strided accesses from the application memory to packed rows of data. Exemplary TILELOAD and TILESTORE instructions, or other instruction references to application memory as a TILE operand in load-op instructions, are, in some embodiments, restartable to handle (up to)

2*rows of page faults, unmasked floating point exceptions, and/or interrupts per instruction.

In (B), a matrix is stored in a tile comprised of a plurality of registers such as packed data registers (single instruction, multiple data (SIMD) or vector registers). In this example, the tile is overlaid on three physical registers. Typically, consecutive registers are used, however, this need not be the case.

In (C), a matrix is stored in a tile in non-register storage accessible to a fused multiple accumulate (FMA) circuit used in tile operations. This storage may be inside of a FMA, or adjacent to it. Additionally, in some embodiments, discussed below, the storage may be for a data element and not an entire row or tile.

The supported parameters for the TMMA architecture are reported via CPUID. In some embodiments, the list of information includes a maximum height and a maximum SIMD dimension. Configuring the TMMA architecture requires specifying the dimensions for each tile, the element size for each tile and the palette identifier. This configuration is done by executing the TILECONFIG instruction.

Successful execution of a TILECONFIG instruction enables subsequent TILE operators. A TILERELEASEALL instruction clears the tile configuration and disables the TILE operations (until the next TILECONFIG instructions executes). In some embodiments, XSAVE, XSTORE, etc. are used in context switching using tiles. In some embodiments, 2 XCRO bits are used in XSAVE, one for TILECONFIF metadata and one bit corresponding to actual tile payload data.

TILECONFIG not only configures the tile usage, but also sets a state variable indicating that the program is in a region of code with tiles configured. An implementation may enumerate restrictions on other instructions that can be used with a tile region such as no usage of an existing register set, etc.

Exiting a tile region is typically done with the TILERELEASEALL instruction. It takes no parameters and swiftly invalidates all tiles (indicating that the data no longer needs any saving or restoring) and clears the internal state corresponding to being in a tile region.

In some embodiments, tile operations will zero any rows and any columns beyond the dimensions specified by the tile configuration. For example, tile operations will zero the data beyond the configured number of columns (factoring in the size of the elements) as each row is written. For example, with 64-byte rows and a tile configured with 10 rows and 12 columns, an operation writing FP32 elements would write each of the first 10 rows with 12*4 bytes with output/result data and zero the remaining 4*4 bytes in each row. Tile operations also fully zero any rows after the first 10 configured rows. When using 1K tile with 64-byte rows, there would be 16 rows, so in this example, the last 6 rows would also be zeroed.

In some embodiments, a context restore (e.g., XRSTOR), when loading data, enforces that the data beyond the configured rows for a tile will be maintained as zero. If there is no valid configuration, all rows are zeroed. XRSTOR of tile data can load garbage in the columns beyond those configured. It should not be possible for XRSTOR to clear beyond the number of columns configured because there is not an element width associated with the tile configuration.

Context save (e.g., XSAVE) exposes the entire TILE storage area when writing it to memory. If XRSTOR loaded garbage data in to the rightmost part of a tile, that data will be saved by XSAVE. XSAVE will write zeros for rows beyond the number specified for each tile.

In some embodiments, tile instructions are restartable. The operations that access memory allow restart after page faults. The computational instructions that deal with floating point operations also allow for unmasked floating-point exceptions, with the masking of the exceptions controlled by a control and/or status register.

To support restarting instructions after these events, the instructions store information in the start registers detailed below.

FIG. **14** illustrates an embodiment of a system utilizing a matrix (tile) operations accelerator. In this illustration, a host processor/processing system **1401** communicates commands **1411** (e.g., matrix manipulation operations such as arithmetic or matrix manipulation operations, or load and store operations) to a matrix operations accelerator **1407**. However, this is shown this way for discussion purposes only. As detailed later, this accelerator **1407** may be a part of a processing core. Typically, commands **1411** that are tile manipulation operator instructions will refer to tiles as register-register ("reg-reg") or register-memory ("reg-mem") format. Other commands such as TILESTORE, TILELOAD, TILECONFIG, etc., do not perform data operations on a tile. Commands may be decoded instructions (e.g., micro-ops) or macro-instructions for the accelerator **1407** to handle.

In this example, a coherent memory interface **1403** is coupled to the host processor/processing system **1401** and matrix operations accelerator **1407** such that they can share memory.

FIGS. **15** and **16** show different embodiments of how memory is shared using a matrix operations accelerator. As shown in FIG. **15**, the host processor **1501** and matrix operations accelerator circuitry **1505** share the same memory **1503**. FIG. **16** illustrates an embodiment where the host processor **1601** and matrix operations accelerator **1605** do not share memory, but can access each other's memory. For example, processor **1601** can access tile memory **1607** and utilize its host memory **1603** as normal. Similarly, the matrix operations accelerator **1605** can access host memory **1603**, but more typically uses its own memory **1607**. Note these memories may be of different types.

The matrix operations accelerator **1407** includes a plurality of FMAs **1409** coupled to data buffers **1405** (in some implementations, one or more of these buffers **1405** are stored in the FMAs of the grid as shown). The data buffers **1405** buffer tiles loaded from memory and/or tiles to be stored to memory (e.g., using a tileload or tilestore instruction). Data buffers may be, for example, a plurality of registers. Typically, these FMAs are arranged as a grid of chained FMAs **1409** which are able to read and write tiles. In this example, the matrix operations accelerator **1407** is to perform a matrix multiply operation using tiles T0, T1, and T2. At least one of tiles is housed in the FMA grid **1409**. In some embodiments, all tiles in an operation are stored in the FMA grid **1409**. In other embodiments, only a subset are stored in the FMA grid **1409**. As shown, T1 is housed and T0 and T2 are not. Note that A, B, and C refer to the matrices of these tiles which may or may not take up the entire space of the tile.

FIG. **17** is a block diagram illustrating processing components for executing variable-format, matrix multiplication instruction(s) **1703**, according to some embodiments. As illustrated, storage **1701** stores instruction(s) **1703** to be executed. As described further below, in some embodiments, computing system **1700** is a single instruction, multiple data (SIMD) processor to concurrently process multiple data elements based on a single instruction.

In operation, the instruction **1703** is to be fetched from storage **1701** by fetch circuitry **1705**. The fetched instruction **1707** is to be decoded by decode circuitry **1709**. The instruction format, which is further illustrated and described herein, has fields (not shown here) to specify an opcode, and destination, multiplier, multiplicand, and summand complex vectors. Decode circuitry **1709** decodes the fetched instruction **1707** into one or more operations. In some embodiments, this decoding includes generating a plurality of micro-operations to be performed by execution circuitry (such as execution circuitry **1719**) in conjunction with routing circuitry **1718**. The decode circuitry **1709** also decodes instruction suffixes and prefixes (if used). Execution circuitry **1719**, operating in conjunction with routing circuitry **1717** executes the decoded instruction.

In some embodiments, register renaming, register allocation, and/or scheduling circuit **1713** provides functionality for one or more of: 1) renaming logical operand values to physical operand values (e.g., a register alias table in some embodiments), 2) allocating status bits and flags to the decoded instruction, and 3) scheduling the decoded VFVSMM instruction **1711** for execution on execution circuitry **1719** out of an instruction pool (e.g., using a reservation station in some embodiments).

Registers (register file) and/or memory **1715** store data as operands of decoded VFVSMM instruction **1711** to be operated on by execution circuitry **1719**. Exemplary register types include writemask registers, packed data registers, general purpose registers, and floating point registers, as further described herein. In some embodiments, write back circuit **1720** commits the result of the execution of the decoded instruction **1711**.

FIG. **18** is a block diagram of a processing array **1810** to execute a matrix multiply instruction (or sequence of instructions) to perform parallel multiply accumulate operations to multiply a first matrix **1802** (Matrix A with M×K elements) by a second matrix **1804** (Matrix B with K×N elements) to generate an output matrix (Matrix C). In one embodiment, the matrix multiplication instruction is a variable-sparsity matrix multiplication (VFVSMM) instruction, which performs dense-dense, sparse-dense, and sparse-sparse matrix multiplications. However, the underlying principles of the invention are not limited to any specific matrix types.

In one embodiment, the processing array **1810** includes (M×N) processing units **1814** each of which multiplies one or more data elements from the first matrix **1802** and one or more data elements from the second matrix **1804**, and accumulates the resulting products (e.g., adding the products and an accumulated value). In some embodiments, each of the processing units in the processing array **1810** is a multiply-accumulate circuit, one example of which is shown as MAC **1814**. While the illustrated MAC **1814** shows a single multiplier, each MAC **1814** may include a plurality of parallel multipliers to perform parallel multiplications. In one embodiment, the number of parallel multiplications is based on the size of the input operands. For example, each MAC unit **1814** may be capable of performing one 16-bit integer multiplication, two 8-bit integer multiplications, or four 4-bit integer multiplications, all of which may be accumulated to a 32-bit integer value. Similarly, each MAC unit **1814** may be capable of multiplying one 32-bit floating point, two 16-bit floating point values (e.g., FP16 or Bfloat16), and accumulating the result into 32-bit or 64-bit floating point value.

In some embodiments, for example when processing 8-bit integer data, execution circuitry throughput is quadrupled by configuring each processing unit to perform a 2×2 matrix multiplication.

As described herein, a processing unit is sometimes referred to as a processing element, a processing circuit, or a processing node. Regardless of the wording, the processing unit is intended to comprise circuitry to perform data path computations and provide control logic.

In some implementations, the tile/matrix sources contain complex numbers in the format of 16-bit floating point (FP16) pairs of real and imaginary parts: each element in the source tile is 32-bit wide complex number where the lower 16 bits represent the real part in FP16 format and the higher 16 bits represent the imaginary part in FP16 format. FIG. **19** illustrates an example matrix—Matrix A—with m rows and k columns of complex data elements, where each data element includes a FP16 real component **1901** and a FP16 imaginary component **1902**. Alternatively, two separate matrices may be used for the real and imaginary components.

In one embodiment, the result matrix C may include 32-bit real values (Matrix CR) or 32-bit imaginary values (Matrix CI), depending on the instruction being executed. In other embodiments, the real and imaginary values may be combined into a single result matrix (Matrix C).

While certain specific data element sizes are described herein, it should be noted that the underlying principles of the invention may be implemented using various other data types for the sources and results described herein including, but not limited to, 4-bit integer, 8-bit integer, 16-bit integer, 32-bit integer, Bfloat16, TensorFloat (TF)-32, 32-bit floating-point, and 64-bit floating point (FP), to name a few.

Vector Packed Signed/Unsigned Shift, Round, Saturate

In sensing applications such as convolution neural networks (CNNs) which use 32-bit, 16-bit, 8-bit, 4-bit integer data types, post processing of the data may be required following accumulation. In one embodiment, the packed signed, unsigned shift instructions described below are used to quantize the accumulator data.

FIGS. **20**A-C illustrate different architectures in which the embodiments described herein may be implemented. In FIG. **20**A, matrix decode circuitry **2090** within each core **2001***a* decodes the instructions described herein and matrix execution circuitry **2091** executes the instructions. In this embodiment, the matrix processing circuitry **2090**-**2091** is integral to the pipeline of each core **2001***a*. Alternatively, in FIG. **20**B, a matrix processing accelerator **2095** is on the same chip and shared by a plurality of cores **2001***a-d*. In FIG. **20**C, the matrix processing accelerator **2096** is on a different chip (but potentially in the same package) as the cores **2001***a-b*. In each implementation, the underlying principles of the invention operate as described herein.

While the signed/unsigned shift/round/saturate instructions described herein may be executed within the context of matrix operations (e.g., to perform post-processing of values resulting from matrix multiplications), these instructions may be processed directly by the decoder **2009** and execution unit **2008** (without the need for dedicated matrix circuitry).

Turning first to FIG. **20**A, the illustrated architectures include a core region **2001** and a shared, or "uncore" region **2010**. The shared region **2010** includes data structures and circuitry shared by all or a subset of the cores **2001***a-b*. In

the illustrated embodiment, the plurality of cores **2001***a-b* are simultaneous multithreaded cores capable of concurrently executing multiple instruction streams or threads. Although only two cores **2001***a-b* are illustrated in FIG. **20**A for simplicity, it will be appreciated that the core region **2001** may include any number of cores, each of which may include the same architecture as shown for core **2001***a*. Another embodiment includes heterogeneous cores which may have different instruction set architectures and/or different power and performance characteristics (e.g., low power cores combined with high power/performance cores).

The various components illustrated in FIG. **20**A may be implemented in the same manner as corresponding components described above. For example, the core **2001***a* may execute the signed/unsigned shift/round/saturate instructions using one of the instruction formats and register architectures described herein. In addition, the cores **2001***a* may include the components of core **490** shown in FIG. **3**B, and may include any of the other processor/core components described herein (e.g., FIGS. **2**, **4**, etc).

Each of the cores **2001***a-b* includes instruction pipeline components for performing simultaneous execution of instruction streams including instruction fetch circuitry **2018** which fetches instructions from system memory **2060** or the L1 instruction cache **2010** and decoder **2009** to decode the instructions. Execution circuitry **2008** executes the decoded instructions to perform the underlying operations, as specified by the instruction operands, opcodes, and any immediate values.

In the illustrated embodiment, the decoder **2009** includes matrix decode circuitry **2090** to decode certain instructions into uops for execution by the matrix execution circuitry **2091** (integrated within the execution circuitry **2008** in this embodiment). Although illustrated as separate blocks in FIG. **20**A, the matrix decode circuitry **2090** and matrix execution circuitry **2091** may be distributed as functional circuits spread throughout the decoder **2009** and execution circuitry **2008** (e.g., multipliers, multiplexers, etc).

In the embodiment illustrated in FIG. **20**B the matrix processing accelerator **2095** is tightly coupled to the processor cores **2001***a-b* over a cache coherent interconnect **2006**. The matrix processing accelerator **2095** of this embodiment is configured as a peer of the cores, participating in the same set of cache coherent memory transactions as the cores. As illustrated, the matrix processing accelerator **2095** may include its own set of registers **2018***a* (e.g., tile registers, vector registers, mask registers, etc) to perform the operations described herein. In this embodiment, the decoder **2009** decodes the instructions which are to be executed by the matrix processing accelerator **2095** and the resulting microoperations are passed for execution to the matrix processing accelerator **2095** over the interconnect **2006**. In another embodiment, the matrix processing accelerator **2095** includes its own fetch and decode circuitry to fetch and decode instructions, respectively, from a particular region of system memory **2060**. In either implementation, after executing the instructions, the matrix accelerator **2091** may store the results to the region in system memory **2060** (which may be accessed by the cores **2001***a-b*).

FIG. **20**C illustrates another embodiment in which the matrix processing accelerator **2096** is on a different chip from the cores **2001***a-b* but coupled to the cores over a cache coherent interface **2096**. In one embodiment, the cache coherent interface **2096** uses packet-based transactions to ensure that data accessed/cached by the matrix processing accelerator **2096** is kept coherent with the cache hierarchy of the cores **2001***a-c*.

Also illustrated in FIGS. **20**A-C are general purpose registers (GPRs) **2018***d*, a set of vector/tile registers **2018***b*, a set of mask registers **2018***a* (which may include tile mask registers as described below), and a set of control registers **2018***c*. In one embodiment, multiple vector data elements are packed into each vector register which may have a 512-bit width for storing two 256-bit values, four 128-bit values, eight 64-bit values, sixteen 32-bit values, etc. Groups of vector registers may be combined to form the tile registers described herein. Alternatively, a separate set of 2-D tile/tensor registers may be used. However, the underlying principles of the invention are not limited to any particular size/type of vector/tile data. In one embodiment, the mask registers **2018***a* include eight 64-bit operand mask registers used for performing bit masking operations on the values stored in the vector registers **2006** (e.g., implemented as mask registers k0-k7 described above). However, the underlying principles of the invention are not limited to any particular mask register size/type.

The control registers **2018***c* store various types of control bits or "flags" which are used by executing instructions to determine the current state of the processor core **2001***a*. By way of example, and not limitation, in an x86 architecture, the control registers include the EFLAGS register.

An interconnect **2006** such as an in-die interconnect (IDI) or memory fabric implementing an IDI/coherence protocol communicatively couples the cores **2001***a-b* (and potentially the matrix accelerator **2095**) to one another and to various components. For example, the interconnect **2006** couples core **2001***a* via interface **2007** to a level 3 (L3) cache **2020** and an integrated memory controller **2030**. In addition, in some embodiments, the interconnect **2006** may be used to couple the cores **2001***a-b* to the matrix processing accelerator **2095**.

The integrated memory controller **2030** provides access to a system memory **2060**. One or more input/output (I/O) circuits (not shown) such as PCI express circuitry may also be included in the shared region **2010**.

An instruction pointer register **2012** stores an instruction pointer address identifying the next instruction to be fetched, decoded, and executed. Instructions may be fetched or prefetched from system memory **2060** and/or one or more shared cache levels such as an L2 cache **2013**, the shared L3 cache **2020**, or the L1 instruction cache **2010**. In addition, an L1 data cache **2002** stores data loaded from system memory **2060** and/or retrieved from one of the other cache levels **2013**, **2020** which cache both instructions and data. An instruction TLB (ITLB) **2011** stores virtual address to physical address translations for the instructions fetched by the fetch circuitry **2018** and a data TLB (DTLB) **2003** stores virtual-to-physical address translations for the data processed by the decode circuitry **2009** and execution circuitry **2008**.

A branch prediction unit **2021** speculatively predicts instruction branch addresses and branch target buffers (BTBs) **2022** for storing branch addresses and target addresses. In one embodiment, a branch history table (not shown) or other data structure is maintained and updated for each branch prediction/misprediction and is used by the branch prediction unit **2002** to make subsequent branch predictions.

Note that FIGS. **20**A-C are not intended to provide an exhaustive view of all circuitry and interconnects employed within an example processor. Rather, various components which are not pertinent to the embodiments of the invention are not shown. Conversely, some components are shown merely for the purpose of providing an example architecture

in which embodiments of the invention may be implemented, but are not necessarily required for complying with the underlying principles of the invention.

Signed, Unsigned Shift, Round, Saturate Instructions

For many of the sensing algorithms that perform dot products of 32/16/8/4-bit signed and unsigned integer data types (e.g., using matrix multiplication as described above), the following shift, round, and saturate instructions may be used to improve the performance and enable new sensor use cases. These embodiments also help to reduce the code memory footprint and provide significant performance gains for various sensing algorithms.

These embodiments include signed and unsigned word and doubleword shift/round/saturate (SRS) instructions. Additional variants of the doubleword SRS instructions are provided which are designed for certain doubleword accumulators.

One embodiment includes the following unsigned word SRS instructions:

DVPSRRSUQW xmm1, xmm2/m128, imm8
DVPSRVRSUQW xmm1, xmm2, xmm3/m128
DVPSLRSUQW xmm1, xmm2/m128, imm8
DVPSLVRSUQW xmm1, xmm2, xmm3/m128

All of the instructions listed above identify a 128-bit destination register (e.g., xmm1). The DVPSRRSUQW instruction also specifies a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8). The DVPSRVRSUQW instruction specifies a first 128-bit source register (xmm2) and a second 128-bit source register (xmm3) or memory location (m128). Both of these instructions perform right shift operations as described below.

The DVPSLRSUQW instruction specifies a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8) and the DVPSLVRSUQW instruction specifies a first 128-bit source register (xmm2) and a second 128-bit source register or memory location (xmm3/m128). Both of these instructions perform left shift operations as described below.

Example pseudocode is provided below showing details in accordance with one embodiment:

```
TEMP        ← SRC2[127:0];
COUNT0[5:0] ← (imm8[5:0] OR SRC3[5:0]);
COUNT1[5:0] ← (imm8[5:0] OR SRC3[69:64]);
SAT_POS0    ← 0;
SAT_POS1    ← 0;
IF (COUNT0 > 63)
  COUNT0[63:0]  ← 64;
FI;
IF (COUNT1 > 63)
  COUNT1[63:0]  ← 64;
FI;
DO WHILE (COUNT0 != 0)
    IF (Instruction is DVPSLRSUQW or DVPSLVRSUQW) THEN
      SAT_POS0 ← (TEMP[63] | | SAT_POS0); (* Accum. Shifted 1's
*)
      TEMP[63:0] ← {TEMP[62:0], 1'b0} ;      (* Left Shift *)
    ELSE (* Instruction is DVPSRRSUQW or DVPSRVRSUQW *)
      TEMP[63:0] ← {1'b0,TEMP[63:1]} ;      (* Logical Shift Right *)
COUNTO ← (COUNT0 - 1);
ENDWHILE;
DO WHILE (COUNT1 != 0)
    IF (Instruction is DVPSLRSUQW or DVPSLVRSUQW) THEN
      SAT_POS1 ← (TEMP[127] | | SAT_POS1); (* Accum. Shifted 1's
      *)
        TEMP[127:64] ← {TEMP[126:64], 1b0}; (* Left Shift *)
```

-continued

```
    ELSE (* Instruction is DVPSRRSUQW or DVPSRVRSUQW *)
      TEMP[127:64] ← {1'b0,TEMP[127:65]}; (* Logical Shift Right *)
COUNT1 ← (COUNT1 - 1);
ENDWHILE;
IF (Instruction is DVPSLRSUQW or DVPSLVRSUQW) THEN
    RoundBit0 ← CalcUnSignedWordRoundBit(TEMP[63:0],
    MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ← CalcUnSignedWordRoundBit(TEMP[127:64],
    MXCSR.IRM[1:0], RoundBit1);
ELSE (* Instruction is DVPSRRSUQW or DVPSRVRSUQW *)
    RoundBit0 ← CalcUnSignedWordRoundBit(SRC2[63:0],
    MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ← CalcUnSignedWordRoundBit(SRC2[127:64],
    MXCSR.IRM[1:0], RoundBit1);
IF (Instruction is DVPSLRSUQW or DVPSLVRSUQW) THEN
    DEST[15:0]   ← AddBitSaturateToUnSignedWord(TEMP[63:48],
    RoundBit0, SAT_POS0, DEST[15:0]);
    DEST[79:64]  ← AddBitSaturateToUnSignedWord(TEMP[127: 112],
    RoundBit1, SAT_POS1, DEST[79:64]);
    DEST[63:16]  ← 0;
    DEST[127:80] ← 0;
ELSE (* Instruction is DVPSRRSUQW or DVPSRVRSUQW *)
    DEST[15:0]   ← AddBitSaturateToUnSignedWord(TEMP[63:48],
    RoundBit0, 0, DEST[15:0]);
    DEST[79:64]  ← AddBitSaturateToUnSignedWord(TEMP[127:112],
    RoundBit1, 0, DEST[79:64]);
    DEST[63:16]  ← 0;
    DEST[127:80] ← 0;
```

In accordance with the above code sequence, the two values in the upper and lower 64-bit locations of the first 128-bit source register (SRC2) are shifted based on the count values, COUNT0 and COUNT1. The upper and lower values may be 64-bit values or may be values encoded with fewer than 64 bits (e.g., 32 bits, 16 bits, etc) but packed into 64-bit locations. The two result values are 16-bit words.

In accordance with the above code sequence, the 5-bit count values, COUNT0 and COUNT1, indicate the number of bits to shift right or shift left, where COUNT0 corresponds to the lower value in the first 128-bit source register or memory location and COUNT1 corresponds to the upper value in the first 128-bit source register or memory location. The upper and lower values may be 64-bit values or may be values encoded with fewer than 64 bits (e.g., 32 bits, 16 bits, etc) but packed into 64-bit locations. Depending on the instruction, the count values are provided by an immediate (imm8) or a second source register or memory location (SRC3). The shifted values are then rounded and saturated to generate two 16-bit results, which are stored in bits 15:0 and 79:64 of the destination register.

FIG. **21** illustrates one particular implementation in which the lower value is stored in bits 63:0 of the 128-bit source register or memory location **2105** and the upper value is stored in bits 127:64 of the 128-bit source register or memory location **2105**. Shift units, **2112** and **2111**, of the execution circuitry **2100** shift the lower and upper values, respectively, based on the corresponding count values, COUNT0 and COUNT1 (which, as mentioned, are provided via an immediate or second source register/memory location). The shift direction (i.e., right or left) is determined by the specific instruction being executed.

As mentioned, DVPSRRSUQW and DVPSRVRSUQW perform right-shift operations. In the present embodiment, logical right-shift operations are performed in which zeroes are shifted in from the left as the original bit values are shifted out to the right. In other embodiment described below (e.g., instructions operating on signed values), arithmetic shift right operations are performed in which the value of the most significant bit is shifted in from the left as original bit values are shifted out to the right.

The resulting shifted values are provided to rounding/saturation units **2122** and **2121** which perform the rounding/saturation operations described herein to generate the two 16-bit results **2150-2151** in the 128-bit destination register **2160**. In particular, the first result **2150**, associated with the lower source values, is stored in bits 16:0 and the second result **2151**, associated with the upper source value, is stored in bits 79:64.

In one embodiment, a control/status register **2112** such as an MXCSR register, includes a set of rounding control bits which indicate a rounding mode to be used by the rounding/saturation units **2122** and **2121**. In one particular implementation, the rounding control bits **2112**A comprise two bits in an Integer Rounding Mode Control (IRM) field of the MXCSR register **2112**. For example, in some embodiments, a value of two zeros ('00') may indicate that "even rounding" (or "convergent" rounding) is to be performed, which indicates that the value is to be rounded to the nearest even value. Similarly, bit values one followed by a zero ('10') may indicate that "rounding up" is to be performed, which could be performed (as for 2's complement values) by adding ½ of the least significant bit to the result and then truncating that resultant value. Additionally, a bit value of two ones ('**11**') may indicate that the rounding is to be a truncation (an eliminating or "dropping") of extra bits. It should be noted, however, that the underlying principles of the invention are not limited to any particular rounding mode.

Additionally, in some embodiments, the rounding/saturation units **2122** and **2121** can determine whether saturation is to be performed based upon detecting overflow or underflow. For example, when a saturation condition is detected, each of the final 16-bit results **2150** and **2151** can be saturated to a most positive or most negative value. In the code sequence listed above, the SAT_POS0 and SAT_POS1 values are set to 1 when the left shift operation performed by DVPSLRSUQW and DVPSLVRSUQW shifts out a binary 1. In one embodiment, this causes corresponding saturation bit(s) **2112**B in the control register **2112** to be set to 1. Thus, a saturation can be determined based on SAT_POS0 and/or SAT_POS1 being set to 1 during left shift operations and reflected in the corresponding saturation bit(s) **2112**B for use by subsequent operations. In one embodiment, the SAT_POS0/SAT_POS1 values and/or the corresponding saturation bit(s) **2112**B are maintained at 0 for the DVPSRRSUQW or DVPSRVRSUQW instructions because the most significant is are maintained during right shift operations.

As mentioned, a saturation field **2112**B of the control/status register(s) **2112** can be set to indicate saturation. For example, the MXCSR.SAT bit (e.g, MXCSR[20]) can be set to indicate that the saturation has occurred. Thus, in some embodiments, a value in a saturation field **2112**B of the control/status register(s) **2112** can be updated when the execution circuitry **2100** detects that saturation has occurred responsive to the prior executed operations on the data values. In some embodiments the saturation field **2112**B can be a single bit location that can ordinarily be a zero but set to a value of one when saturation is detected.

Following rounding/saturation, the 16-bit result values **2150** and **2151** are stored in bits 16:0 and 79:64 of the destination register **2160**. In the illustrated embodiment, zeroes are stored in bit ranges 63:16 and 127:80.

A method for performing a left shift SRS instruction is illustrated in FIG. **22**. The method may be performed on the processor or system architectures described herein, but is not limited to any particular processor or system architecture.

At **2201** a left-shift SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second left shift values, and a destination operand to identify first and second result values corresponding to the first and second source values, respectively.

At **2202**, the left-shift SRS instruction is decoded. In a microcoded implementation, for example, decoding of the left-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2203**, the left-shift SRS instruction is executed to left-shift the first and second source values based on the first and second left shift values of the second source operand or immediate to generate first and second shifted values. A saturation indication is stored if a saturation condition is detected resulting from the left shift of the first and second source values (e.g., from shifting out a most-significant 1). The first and second shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or the saturation indication, respectively, to generate the first and second result values in the destination register **2160**. The first and second result values are then stored in corresponding first and second locations of a destination register.

A method for performing a right-shift SRS instruction is illustrated in FIG. **23**. The method may be performed on the processor and system architectures described herein, but is not limited to any particular processor or system architecture.

At **2301** a right-shift SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second right shift values, and a destination operand to identify first and second result values corresponding to the first and second source values, respectively.

At **2302**, the right-shift SRS instruction is decoded. In a microcoded implementation, for example, decoding of the right-shift/round/saturate instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2303**, the right-shift SRS instruction is executed to right-shift the first and second source values based on the first and second right-shift values of the second source operand or immediate to generate first and second shifted values. The first and second shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or a saturation indication, respectively, to generate the first and second result values. The first and second result values are then stored in corresponding first and second locations of a destination register.

One embodiment further includes the following signed word SRS instructions:

DVPSRARSQW xmm1, xmm2/m128, imm8
DVPSRAVRSQW xmm1, xmm2, xmm3/m128
DVPSLRSQW xmm1, xmm2/m128, imm8
DVPSLVRSQW xmm1, xmm2, xmm3/m128

All of the instructions specified above identify a 128-bit destination register (xmm1 in the example). The DVPSRARSQW instruction also specifies a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8). The DVPSRAVRSQW instruction specifies a first 128-bit source register (xmm2) and a second 128-bit source register or memory location (xmm3/m128). Both of these instructions perform right shift operations as described below

The DVPSLRSQW instruction specifies a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8) and the DVPSLVRSQW instruction

specifies a first 128-bit source register (xmm2) and a second 128-bit source register or memory location (xmm3/m128). Both of these instructions perform left shift operations as described below.

Example pseudocode is provided below showing details in accordance with one embodiment:

```
TEMP[127:0] ← SRC2[127:0];
COUNT0[5:0] ← (imm8[5:0] OR SRC3[5:0]);
COUNT1[5:0] ← (imm8[5:0] OR SRC3[69:64]);
SIGN_BIT0 ← SRC2[63];
SIGN_BIT1 ← SRC2[127];
SAT_POS0 ← 0;
SAT_POS1 ← 0;
SAT_NEG0 ← 0;
SAT_NEG1 ← 0;
IF (COUNT0 > 63)
  COUNT0[63:0] ← 64;
FI;
IF (COUNT1 > 63)
  COUNT1[63:0] ← 64;
FI;
DO WHILE (COUNT0 != 0)
  IF (Instruction is DVPSLRSQW or DVPSLVRSQW) THEN
    SAT_POS0 ← ((~SIGN_BIT0 &TEMP[62]) ? 1 : 0) | | SAT_POS0 ;
      (* Check Sign Bit for Positive Saturation *)
    SAT_NEG0 ← ((SIGN_BIT0 & ~TEMP[62]) ? 1 : 0) | | SAT_NEG0
;
      (* Check Sign Bit for Negative Saturation *)
    TEMP[63:0] ← {TEMP[62:0], 1'b0}; (* Left Shift *)
  ELSE (* Instruction is DVPSRARSQW or DVPSRAVRSQW *)
    TEMP[63:0] ←{TEMP[63], TEMP[63:1]} ; (* Arith Right Shift *)
COUNT0 ← (COUNT0 − 1);
ENDWHILE;
DO WHILE (COUNT1 != 0)
  IF (Instruction is DVPSLRSQW or DVPSLVRSQW) THEN
    SAT_POS1 ← ((~SIGN_BIT1 &TEMP[126]) ? 1 : 0) | | SAT_POS1 ;
      (* Check Sign Bit for Positive Saturation *)
    SAT_NEG1 ← ((SIGN_BIT1 & ~TEMP[126]) ? 1 : 0) | | SAT-
_NEG1 ;
      (* Check Sign Bit for Negative Saturation *)
    TEMP[127:64] ← {TEMP[126:64], 1'b0}; (* Left Shift *)
  ELSE (* Instruction is DVPSRARSQW or DVPSRAVRSQW *)
    TEMP[127:64] ← {TEMP[127], TEMP[127:65]} ;
      (* Arithmetic Right Shift *)
COUNT1 ← (COUNT1 − 1);
ENDWHILE;
IF (Instruction is DVPSLRSQW or DVPSLVRSQW) THEN
    RoundBit0 ← CalcSignedWordRoundBit(TEMP[63:0],
    MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ← CalcSignedWordRoundBit(TEMP[127:64],
    MXCSR.IRM[1:0], RoundBit1);
ELSE (* Instruction is DVPSRLRSQW or DVPSRLVRSQW *)
    RoundBit0 ← CalcSignedWordRoundBit(SRC2[63:0],
    MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ← CalcSignedWordRoundBit(SRC2[127:64],
    MXCSR.IRM[1:0], RoundBit1);
IF (Instruction is DVPSLRSQW or DVPSLVRSQW ) THEN
  DEST[15:0] ← AddBitSaturateToSignedWord(TEMP[63:48], Round-
Bit0,
  SAT_POS, SAT_NEG, DEST[15:0]);
  DEST[63:16] ← 0;
  DEST[79:64] ← AddBitSaturateToSignedWord (TEMP[127:112],
  RoundBit1, SAT_POS1, SAT_NEG1, DEST[79:64]);
  DEST[127:80] ← 0;
ELSE (* Instruction is DVPSRARSQW or DVPSRAVRSQW *)
  DEST[15:0] ← AddBitSaturatePosToSignedWord(TEMP[63:48],
  RoundBit0,1'b0, DEST[15:0]);
  DEST[63:16] ← 0;
  DEST[79:64] ← AddBitSaturatePosToSignedWord (TEMP[127:112],
  RoundBit1, 1'b0, DEST[79:64]);
  DEST[127:80] ← 0;
```

Thus, the signed word SRS instructions use positive saturation flags (SAT_POS0 and SAT_POS1) to indicate positive saturation and negative saturation flags (SAT_NEG0 and SAT_NEG1) to indicate positive or negative saturation, respectively. The rounding/saturation units **2121-2122** can then evaluate the positive and negative saturation flags to determine whether the result values are to be saturated negatively or positively. The signed SRS instructions must also process the sign values (SIGN_BIT0 and SIGN_BIT1) for each of the source values.

For the right-shift instructions DVPSRARSQW and DVPSRAVRSQW, arithmetic shift right operations are performed to preserve the sign of the underlying value. In the arithmetic shift right operations, the value of the most significant bit is shifted in from the left as the least significant bit values are shifted out to the right.

A method for performing a left-shift signed word SRS instruction is illustrated in FIG. **24**. The method may be performed on the processor or system architectures described herein, but is not limited to any particular processor or system architecture.

At **2401** a left-shift signed word SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second left shift values, and a destination operand to identify first and second result values corresponding to the first and second source values, respectively.

At **2402**, the left-shift signed word SRS instruction is decoded. In a microcoded implementation, for example, decoding of the left-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2403**, the left-shift signed word SRS instruction is executed to left-shift the first and second signed source values based on the first and second left shift values of the second source operand or immediate to generate first and second signed, shifted values. A positive or negative saturation indication is stored if a positive/negative saturation condition is detected resulting from the left shift of the first and second source values. The first and second signed shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or the positive/negative saturation indication, respectively, to generate the first and second result values in the destination register **2160**. The first and second result values are then stored in corresponding first and second locations of a destination register.

A method for performing a right-shift signed word SRS instruction is illustrated in FIG. **25**. The method may be performed on the processor and system architectures described herein, but is not limited to any particular processor or system architecture.

At **2501** a right-shift signed word SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second right shift values, and a destination operand to identify first and second result values corresponding to the first and second source values, respectively.

At **2502**, the right-shift signed word SRS instruction is decoded. In a microcoded implementation, for example, decoding of the right-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2503**, the right-shift SRS instruction is executed to right-shift the first and second source values based on the first and second right-shift values of the second source operand or immediate to generate first and second shifted values. The first and second shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or a positive/negative saturation indication, respectively, to generate the first and second result values.

The first and second result values are then stored in corresponding first and second locations of a destination register.

In addition to signed/unsigned word SRS instructions, one embodiment of the processor also supports the following signed/unsigned doubleword SRS instructions:

Unsigned DWORD SRS Instructions:

DVPSRRSUQD xmm1, xmm2/m128, imm8
DVPSRVRSUQD xmm1, xmm2, xmm3/m128
DVPSLRSUQD xmm1, xmm2/m128, imm8
DVPSLVRSUQD xmm1, xmm2, xmm3/m128

Signed DWORD SRS Instructions:

DVPSRARSQD xmm1, xmm2/m128, imm8
DVPSRAVRSQD xmm1, xmm2, xmm3/m128
DVPSLRSQD xmm1, xmm2/m128, imm8
DVPSLVRSQD xmm1, xmm2, xmm3/m128

All of the instructions listed above identify a 128-bit destination register (e.g., xmm1). The DVPSRRSUQW and DVPSRRSUQD instructions also specify a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8). The DVPSRVRSUQW and DVPSRVRSUQD instructions specify a first 128-bit source register (xmm2) and a second 128-bit source register (xmm3) or memory location (m128). Each of these instructions perform right-shift operations as described below.

The DVPSLRSUQW and DVPSLRSUQD instructions specify a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8) and the DVPSLVRSUQW and DVPSLVRSUQD instructions specify a first 128-bit source register (xmm2) and a second 128-bit source register or memory location (xmm3/m128). Each of these instructions perform left-shift operations as described below.

The following pseudocode sequence defines the operation for one embodiment of the unsigned doubleword SRS instructions:

```
TEMP ← SRC2[127:0];
COUNT0[5:0] ← (imm8[5:0] OR SRC3[5:0]);
COUNT1[5:0] ← (imm8[5:0] OR SRC3[69:64]);
SAT_POS0 ← 0;
SAT_POS1 ← 0;
IF (COUNT0 > 63)
  COUNT0[63:0] ← 64;
FI;
IF (COUNT1 > 63)
  COUNT1[63:0] ← 64;
FI;
DO WHILE (COUNT0 != 0)
    IF (Instruction DVPSLRSUQD or DVPSLVRSUQD) THEN
      SAT_POS0 ← (TEMP[63] | | SAT_POS0) ;
        (* Check Sign Bit for Positive Saturation *)
      TEMP [63:0] ← ({TEMP[62:0], 1'b0}) ; (* Left Shift *)
    ELSE (* Instruction DVPSRRSUQD or DVPSRVRSUQD *)
      TEMP [63:0] ← {1'b0, TEMP[63:1]} ; (* Logical Right Shift *)
    COUNT0 ← (COUNT0 - 1);
ENDWHILE;
DO WHILE (COUNT1 != 0)
    IF (Instruction DVPSLRSUQD or DVPSLVRSUQD) THEN
      SAT_POS1 ← (TEMP[127] | | SAT_POS1) ;
        (* Check Sign Bit for Positive Saturation *)
      TEMP [127:64] ← {TEMP[126:64], 1'b0} ; (* Left Shift *)
    ELSE (* Instruction DVPSRRSUQD or DVPSRVRSUQD *)
      TEMP [127:64] ← {1'b0, TEMP[127:65]} ; (* Logical Right Shift
*)
    COUNT1 ← (COUNT1 - 1);
ENDWHILE;
IF (Instruction is DVPSLRSUQD or DVPSLVRSUQD) THEN
    RoundBit0 ← CalcUnSignedDwordRoundBit(TEMP[63:0],
    MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ← CalcUnSignedDwordRoundBit(TEMP[127:64],
    MXCSR.IRM[1:0], RoundBit1);
ELSE (* Instruction is DVPSRRSUQD or DVPSRVRSUQD *)
```

-continued

```
    RoundBit0 ← CalcUnSignedDwordRoundBit(SRC2[63:0],
    MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ← CalcUnSignedDwordRoundBit(SRC2[127:64],
MXCSR.IRM[1:0], RoundBit1);
IF Instruction DVPSLRSUQD or DVPSLVRSUQD THEN
    DEST[31:0] ← AddBitSaturateToUnSignedDword(TEMP[63:32],
    RoundBit0, SAT_POS0, DEST[31:0]);
    DEST[63:32] ← 0;
    DEST[95:64] ← AddBitSaturateToUnSignedDword(TEMP[127:96],
    RoundBit1, SAT_POS1, DEST[95:64]);
    DEST[127:96] ← 0;
ELSE (* Instruction DVPSRRSUQD or DVPSRVRSUQD *)
    DEST[31:0] ← AddBitSaturateToUnSignedDword(TEMP[63:32],
    RoundBit0, 1'b0, DEST[31:0]);
    DEST[63:32] ← 0;
    DEST[95:64] ← AddBitSaturateToUnSignedDword(TEMP[127:96],
    RoundBit1, 1'b0, DEST[95:64]);
    DEST[127:96] ← 0;
```

In accordance with the above code sequence, the two values in the upper and lower 64-bit locations of the first 128-bit source register (SRC2) are shifted based on the count values, COUNT0 and COUNT1. The upper and lower values may be 64-bit values or may be values encoded with fewer than 64 bits (e.g., 32 bits, 16 bits, etc) but packed into 64-bit locations. The two result values are 32-bit doublewords.

In accordance with the above code sequence, the 5-bit count values, COUNT0 and COUNT1, indicate the number of bits to shift right or shift left, where COUNT0 corresponds to the lower value in the first 128-bit source register or memory location and COUNT1 corresponds to the upper value in the first 128-bit source register or memory location. Depending on the instruction, the count values are provided by an immediate (imm8) or a second source register or memory location (SRC3). The shifted values are then rounded and saturated to generate two 16-bit results, which are stored in bits 15:0 and 79:64 of the destination register.

FIG. **26** illustrates one particular implementation in which the lower value is stored in bits 63:0 of the 128-bit source register or memory location **2605** and the upper value is stored in bits 127:64 of the 128-bit source register or memory location **2605**. Shift units, **2612** and **2611**, of the execution circuitry **2600** shift the lower and upper values, respectively, based on the corresponding count values, COUNT0 and COUNT1 (which, as mentioned, are provided via an immediate or second source register/memory location).

The shift direction (i.e., right or left) is determined by the specific instruction being executed. As mentioned, DVPSRRSUQD and DVPSRVRSUQD perform right-shift operations while DVPSLRSUQD and DVPSLVRSUQD perform left-shift operations. Because the values are unsigned, the right-shift instructions DVPSRRSUQD and DVPSRVRSUQD use a logical right-shift operation.

The resulting shifted values are provided to rounding/saturation units **2622** and **2621** which perform the rounding/saturation operations described herein to generate the two 32-bit results **2650**-**2651** in the 128-bit destination register **2660**. In particular, the first result **2650**, associated with the lower source values, is stored in bits 31:0 and the second result **2651**, associated with the upper source value, is stored in bits 95:64.

In one embodiment, a control/status register **2612** such as an MXCSR register, includes a set of rounding control bits which indicate a rounding mode to be used by the rounding/saturation units **2622** and **2621**. In one particular implementation, the rounding control bits **2612**A comprise two bits in

an Integer Rounding Mode Control (IRM) field of the MXCSR register **2612**. For example, in some embodiments, a value of two zeros ('00') may indicate that "even rounding" (or "convergent" rounding) is to be performed, which indicates that the value is to be rounded to the nearest even value. Similarly, bit values one followed by a zero ('10') may indicate that "rounding up" is to be performed, which could be performed (as for 2's complement values) by adding ½ of the least significant bit to the result and then truncating that resultant value. Additionally, a bit value of two ones ('11') may indicate that the rounding is to be a truncation (an eliminating or "dropping") of extra bits. It should be noted, however, that the underlying principles of the invention are not limited to any particular rounding mode.

Additionally, in some embodiments, the rounding/saturation units **2622** and **2621** can determine whether saturation is to be performed based upon detecting overflow or underflow. For example, when a saturation condition is detected, each of the final 32-bit results **2650** and **2651** can be saturated to a most positive or most negative value. In the code sequence listed above, the SAT_POS0 and SAT_POS1 values are set to 1 when the left shift operation performed by DVPSLRSUQD or DVPSLVRSUQD shifts out a binary 1. In one embodiment, this causes corresponding saturation bit(s) **2612**B in the control register **2612** to be set to 1. Thus, a saturation can be determined based on SAT_POS0 and/or SAT_POS1 being set to 1 during left shift operations and reflected in the corresponding saturation bit(s) **2612**B for use by subsequent operations.

In one embodiment, the SAT_POS0/SAT_POS1 values and/or the corresponding saturation bit(s) **2612**B are maintained at 0 for the DVPSRRSUQD and DVPSRVRSUQD instructions because the most significant is are maintained during right shift operations.

As mentioned, a saturation field **2612**B of the control/status register(s) **2612** can be set to indicate saturation. For example, the MXCSR.SAT bit (e.g, MXCSR[20]) can be set to indicate that the saturation has occurred. Thus, in some embodiments, a value in a saturation field **2612**B of the control/status register(s) **2612** can be updated when the execution circuitry **2600** detects that saturation has occurred responsive to the prior executed operations on the data values. In some embodiments the saturation field **2612**B can be a single bit location that can ordinarily be a zero but set to a value of one when saturation is detected.

Following rounding/saturation, the 16-bit result values **2650** and **2651** are stored in bits 31:0 and 95:64 of the destination register **2660**. In the illustrated embodiment, zeroes are stored in bit ranges 63:32 and 127:96.

A method for performing a left shift doubleword SRS instruction is illustrated in FIG. **27**. The method may be performed on the processor or system architectures described herein, but is not limited to any particular processor or system architecture.

At **2701** a left-shift doubleword SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second left shift values, and a destination operand to identify first and second result values corresponding to the first and second source values, respectively.

At **2702**, the left-shift unsigned doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the left-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2703**, the left-shift unsigned doubleword SRS instruction is executed to left-shift the first and second source values based on the first and second left shift values of the second source operand or immediate to generate first and second shifted values. A saturation indication is stored if a saturation condition is detected resulting from the left shift of the first and second source values (e.g., from shifting out a most-significant 1). The first and second shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or the saturation indication, respectively, to generate the first and second unsigned doubleword (32-bit) result values in the destination register **2660**. The first and second unsigned doubleword result values are then stored in corresponding first and second locations of a destination register.

A method for performing a right-shift unsigned doubleword SRS instruction is illustrated in FIG. **28**. The method may be performed on the processor and system architectures described herein, but is not limited to any particular processor or system architecture.

At **2801** a right-shift unsigned doubleword SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second right shift values, and a destination operand to identify first and second result values corresponding to the first and second source values, respectively.

At **2802**, the right-shift unsigned doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the right-shift/round/saturate instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2803**, the right-shift unsigned doubleword SRS instruction is executed to right-shift the first and second source values based on the first and second right-shift values of the second source operand or immediate to generate first and second shifted values. The first and second shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or a saturation indication, respectively, to generate the first and second doubleword (32-bit) result values. The first and second doubleword result values are then stored in corresponding first and second locations of a destination register.

The following pseudocode sequence defines the operation for one embodiment of the signed doubleword SRS instructions:

```
TEMP ← SRC2[127:0];
COUNT0[5:0] ← (imm8[5:0] OR SRC3[5:0]);
COUNT1[5:0] ← (imm8[5:0] OR SRC3[69:64]);
SIGN_BIT0 ← SRC2[63];
SIGN_BIT1 ← SRC2[127];
SAT_POS0 ← 0;
SAT_POS1 ← 0;
SAT_NEG0 ← 0;
SAT_NEG1 ← 0;
```

-continued

```
IF (COUNT0 > 63)
  COUNT0[63:0] ← 64;
FI;
IF (COUNT1 > 63)
  COUNT1[63:0] ← 64;
FI;
DO WHILE (COUNT0 != 0)
      IF (Instruction is DVPSLRSQD or DVPSLVRSQD) THEN
        SAT_POS0 ← ((~SIGN_BIT0 &TEMP[62]) ? 1 : 0) | | SAT_POS0;
        (* Check Sign Bit for Positive Saturation *)
        SAT_NEG0 ← ((SIGN_BIT0 & ~TEMP[62]) ? 1 : 0) | | SAT_NEG0 ;
        (* Check Sign Bit for Negative Saturation *)
        TEMP [63:0] ← {TEMP[62:0], 1'b0}) ;(* Left Shift *)
        ELSE (* Instruction is DVPSRARSQD or DVPSRAVRSQD *)
        TEMP [63:0] ← {TEMP[63], TEMP[63:1]}; (* Arith Right Shift *)
      COUNT0 ← (COUNT0 – 1);
ENDWHILE;
DO WHILE (COUNT1 != 0)
    IF (Instruction is DVPSLRSQD or DVPSLVRSQD) THEN
      SAT_POS1 ← ((~SIGN_BIT1 &TEMP[126]) ? 1 : 0) | | SAT_POS1;
          (* Check Sign Bit for Positive Saturation *)
      SAT_NEG1 ← ((SIGN_BIT1 & ~TEMP[126]) ? 1 : 0) | | SAT_NEG1;
        (* Check Sign Bit for Negative Saturation *)
      TEMP [127:64] ← {TEMP[126:64], 1'b0}) ; (* Left Shift *)
      ELSE (* Instruction is DVPSRARSQD or DVPSRAVRSQD *)
      TEMP [127:64] ← {TEMP[127], TEMP[127:65]} ; (* Arith Rt Shift *)
    COUNT1 ← (COUNT1 – 1);
ENDWHILE;
IF (Instruction is DVPSLRSQD or DVPSLVRSQD) THEN
      RoundBit0 ← CalcSignedDwordRoundBit(TEMP[63:0],
      MXCSR.IRM[1:0], RoundBit0);
      RoundBit1 ← CalcUnSignedDwordRoundBit(TEMP[127:64],
      MXCSR.IRM[1:0], RoundBit1);
ELSE (* Instruction is DVPSRLRSQD or DVPSRLVRSQD *)
      RoundBit0 ← CalcSignedDwordRoundBit(SRC2[63:0],
      MXCSR.IRM[1:0], RoundBit0);
      RoundBit1 ← CalcSignedDwordRoundBit(SRC2[127:64],
      MXCSR.IRM[1:0], RoundBit1);
IF (Instruction is DVPSLRSQD or DVPSLVRSQD) THEN
    DEST[31:0] ← AddBitSaturateToSignedDword(TEMP[63:32], RoundBit0,
        SAT_POS0, SAT_NEG0, DEST[31:0]);
    DEST[63:32] ← 0;
    DEST[95:64] ← AddBitSaturateToSignedDword(TEMP[127:96],
        RoundBit1, SAT_POS1, SAT_NEG1, DEST[95:64]);
    DEST[127:96] ← 0;
ELSE (* Instruction is DVPSRARSQD or DVPSRAVRSQD *)
    DEST[31:0] ← AddBitSaturatePosToSignedDword(TEMP[63:32],
    RoundBit0,1'b0, DEST[31:0]);
    DEST[63:32] ← 0;
    DEST[95:64] ← AddBitSaturatePosToSignedDword(TEMP[127:96],
    RoundBit1,1'b0, DEST[95:64]);
    DEST[127:96] ← 0;
```

Thus, the signed doubleword SRS instructions use positive saturation flags (SAT_POS0 and SAT_POS1) to indicate positive saturation and negative saturation flags (SAT_NEG0 and SAT_NEG1) to indicate positive or negative saturation, respectively. The rounding/saturation units **2621-2622** can then evaluate the positive and negative saturation flags to determine whether the result values are to be saturated negatively or positively. The signed SRS instructions must also process the sign values (SIGN_BIT0 and SIGN_BIT1) for each of the source values.

In this embodiment, an arithmetic shift right is performed for DVPSRARSQD or DVPSRAVRSQD to preserve the sign value. In the arithmetic shift right operations, the value of the most significant bit is shifted in from the left as the least significant bit values are shifted out to the right.

A method for performing a left shift signed doubleword SRS instruction is illustrated in FIG. **29**. The method may be performed on the processor or system architectures described herein, but is not limited to any particular processor or system architecture.

At **2901** a left-shift signed doubleword SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second left shift values, and a destination operand to identify first and second signed doubleword result values corresponding to the first and second source values, respectively.

At **2902**, the left-shift signed doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the left-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **2903**, the left-shift signed doubleword SRS instruction is executed to left-shift the first and second signed source values based on the first and second left shift values of the second source operand or immediate to generate first and second signed, shifted values. A positive or negative saturation indication is stored if a positive/negative saturation condition is detected resulting from the left shift of the first and second source values. The first and second signed shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or the positive/negative saturation indication, respectively, to generate the first

and second result values in the destination register **2660**. The first and second signed doubleword result values are then stored in corresponding first and second locations of a destination register.

A method for performing a right-shift signed doubleword SRS instruction is illustrated in FIG. **30**. The method may be performed on the processor and system architectures described herein, but is not limited to any particular processor or system architecture.

At **3001** a right-shift signed doubleword SRS instruction is fetched, the instruction including a first source operand to identify first and second source values, a second source or immediate to identify first and second right shift values, and a destination operand to identify first and second signed doubleword (32-bit) result values corresponding to the first and second source values, respectively.

At **3002**, the right-shift signed doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the right-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **3003**, the right-shift signed doubleword SRS instruction is executed to right-shift the first and second source values based on the first and second right-shift values of the second source operand or immediate to generate first and second shifted values. The first and second shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or a positive/negative saturation indication, respectively, to generate the first and second signed doubleword result values. The first and second signed doubleword result values are then stored in corresponding first and second locations of a destination register.

The following pseudocode sequence defines the operation for additional embodiments of unsigned and signed doubleword SRS instructions which specify four doubleword source and destination values. These embodiments may be particularly useful for implementations with doubleword accumulators.

Unsigned Dword SRS Instructions:

DVPSRRUD xmm1, xmm2/m128, imm8
DVPSRVRUD xmm1, xmm2, xmm3/m128
DVPSLSUD xmm1, xmm2/m128, imm8
DVPSLVSUD xmm1, xmm2, xmm3/m128

Signed Dword SRS Instructions:

DVPSRARD xmm1, xmm2/m128, imm8
DVPSRAVRD xmm1, xmm2, xmm3/m128
DVPSLSD xmm1, xmm2/m128, imm8
DVPSLVSD xmm1, xmm2, xmm3/m128

All of the instructions listed above identify a 128-bit destination register (e.g., xmm1). The DVPSRRUD and DVPSRARD instructions also specify a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8). The DVPSRVRUD and DVPSRAVRD instructions specify a first 128-bit source register (xmm2) and a second 128-bit source register (xmm3) or memory location (m128). Each of these instructions perform right-shift operations as described below.

The DVPSLSUD and DVPSLSD instructions specify a 128-bit source register or 128-bit memory location (xmm2/m128) and an 8-bit immediate (imm8) and the DVPSLVSUD and DVPSLVSD instructions specify a first 128-bit source register (xmm2) and a second 128-bit source register or memory location (xmm3/m128). Each of these instructions perform left-shift operations as described below.

The following pseudocode sequence defines the operation for one embodiment of the unsigned doubleword SRS instructions:

```
TEMP[127:0] ← SRC2[127:0];
COUNT0[4:0] ← (imm8[4:0] OR SRC3[4:0]);
COUNT1[4:0] ← (imm8[4:0] OR SRC3[36:32]);
COUNT2[4:0] ← (imm8[4:0] OR SRC3[68:64]);
COUNT3[4:0] ← (imm8[4:0] OR SRC3[100:96]);
SAT_POS0 ← 0;
SAT_POS1 ← 0;
SAT_POS2 ← 0;
SAT_POS3 ← 0;
SIGN_BIT0 ← SRC2[31];
SIGN_BIT1 ← SRC2[63];
SIGN_BIT2 ← SRC2[95];
SIGN_BIT3 ← SRC2[127];
TEMP0[31:0] ← 0;
TEMP1[31:0] ← 0;
TEMP2[31:0] ← 0;
TEMP3[31:0] ← 0;
IF (COUNT0 > 31)
  COUNT0[31:0] ← 32;
FI;
IF (COUNT1 > 31)
  COUNT1[31:0] ← 32;
FI;
IF (COUNT2 > 31)
  COUNT2[31:0] ← 32;
FI;
IF (COUNT3 > 31)
  COUNT3[31:0] ← 32;
FI;
DO WHILE (COUNT0 != 0)
    IF (Instruction is DVPSLSUD or DVPSLVSUD) THEN
      SAT_POS0 ← (TEMP[31] | | SAT_POS0); (* Accum Shifted 1's *)
      TEMP[31:0] ← {TEMP[30:0], 1'b0} ; (* Left Shift *)
    ELSE (* Instruction is DVPSRRUD or DVPSRVRUD *)
      SAT_POS0 ← 0;
      TEMP0[31:0] ← {TEMP[0], TEMP0[31:1]};
        (* Shifted out bits to be used for Rounding *)
      TEMP[31:0] ← {1'b0,TEMP[31:1]} ; (* Logical Shift Right *)
    COUNT0 ← (COUNT0 - 1);
ENDWHILE;
DO WHILE (COUNT1 != 0)
    IF (Instruction is DVPSLSUD or DVPSLVSUD) THEN
      SAT_POS1 ← (TEMP[63] | | SAT_POS1); (* Accum Shifted 1's *)
      TEMP[63:32] ← {TEMP[62:32], 1'b0} ; (* Left Shift *)
    ELSE (* Instruction is DVPSRRUD or DVPSRVRUD *)
        SAT_POS0 ← 0;
        TEMP1[31:0] ← {TEMP[32], TEMP1[31:1]};
            (* Shifted out bits to be used for Rounding *)
        TEMP[63:32] ← {1'b0,TEMP[63:33]} ; (* Logical Shift Right
*)
    COUNT1 ← (COUNT1 - 1);
ENDWHILE;
DO WHILE (COUNT2 != 0)
    IF (Instruction is DVPSLSUD or DVPSLVSUD) THEN
        SAT_POS2 ← (TEMP[95] | | SAT_POS2); (* Accum Shifted
1's*)
        TEMP[95:64] ← {TEMP[94:64], 1'b0} ; (* Left Shift *)
    ELSE (* Instruction is DVPSRRUD or DVPSRVRUD *)
        SAT_POS2 ← 0;
        TEMP2[31:0] ← {TEMP[64], TEMP2[31:1]};
          (* Shifted out bits to be used for Rounding *)
        TEMP[95:64] ← {1'b0,TEMP[95:65]} ; (* Logical Shift Right *
)
    COUNT2 ← (COUNT2 - 1);
ENDWHILE;
DO WHILE (COUNT3 != 0)
    IF (Instruction is DVPSLSUD or DVPSLVSUD) THEN
      SAT_POS3 ← (TEMP[127] | | SAT_POS3); (* Accum Shifted 1's
*)
      TEMP[127:96] ← {TEMP[126:96], 1'b0} ; (* Left Shift *)
    ELSE (* Instruction is DVPSRRUD or DVPSRVRUD *)
      SAT_POS3 ← 0;
      TEMP3[31:0] ← {TEMP[96], TEMP3[31:1]};
          (* Shifted out bits to be used for Rounding *)
      TEMP[127:96] ← {1'b0,TEMP[127:97]} ; (* Logical Shift Right
*)
    COUNT3 ← (COUNT3 - 1);
ENDWHILE;
IF (Instruction is DVPSLSUD or DVPSLVSUD) THEN
    RoundBit0 ← 0;
    RoundBit1 ← 0;
```

-continued

```
    RoundBit2 ← 0;
    RoundBit3 ← 0;
ELSE (* Instruction is DVPSRRUD or DVPSRVRUD *)
    RoundBit0 ← CalcShiftUnSignedDwordRoundBit({TEMP[0],
    TEMP0[31:0]}, MXCSR.IRM[l:0], RoundBit0);
    RoundBit1 ←
    CalcShiftUnSignedDwordRoundBit({TEMP[32],TEMPl[31:0]},
    MXCSR.IRM[1:0], RoundBit1);
    RoundBit2 ←
    CalcShiftUnSignedDwordRoundBit({TEMP[64],TEMP2[31:0]},
    MXCSR.IRM[1:0], RoundBit2);
    RoundBit3 ←
    CalcShiftUnSignedDwordRoundBit({TEMP[96],TEMP3[31:0]},
    MXCSR.IRM[1:0], RoundBit3);
IF (Instruction is DVPSLSUD or DVPSLVSUD) THEN
  DEST[31:0] ← SaturateToUnSignedDword(SAT_POS0, TEMP[31:0],
  DEST[31:0]);
  DEST[63:32] ← SaturateToUnSignedDword(SAT_POS1, TEMP[63:32],
  DEST[63:32]);
  DEST[95:64] ← SaturateToUnSignedDword(SAT_POS2, TEMP[95:64],
  DEST[95:64]);
  DEST[127:96] ← SaturateToUnSignedDword(SAT_POS3, TEMP[127:
96],
  DEST[127:96]);
ELSE (* Instruction is DVPSRRUD or DVPSRVRUD *)
  DEST[31:0] ← (TEMP[31:0] + {31'b0,RoundBit0});
    (* Add Rounding bit with wrapping *)
  DEST[63:32] ← (TEMP[63:32] + {31'b0,RoundBit1});
    (* Add Rounding bit with wrapping *)
  DEST[95:64] ← (TEMP[95:64] + {31'b0,RoundBit2});
    (* Add Rounding bit with wrapping *)
  DEST[127:96] ← (TEMP[127:96] + {31'b0,RoundBit3});
    (* Add Rounding bit with wrapping *)
```

In accordance with the above code sequence, the four unsigned 32-bit values stored in the first 128-bit source register (SRC2) are shifted based on the four corresponding count values, COUNT0, COUNT1, COUNT 2, and COUNT 3. The four result values are unsigned 32-bit doublewords.

In accordance with the above code sequence, the 5-bit count values, COUNT0, COUNT1, COUNT 2, and COUNT 3, indicate the number of bits to shift right or shift left. Depending on the instruction, the count values are provided by an immediate (imm8) or a second source register or memory location (SRC3). The shifted values are then rounded and saturated to generate four 32-bit results, which are stored in bits 31:0, 63:32, 95:64, and 127:96 of the destination register.

FIG. **31** illustrates one particular implementation in which unsigned 32-bit data elements A-D are stored in a 128-bit source register or memory location **3105**. Shift units, **3112** and **3111**, of the execution circuitry **3100** shift the lower and upper values, respectively, based on the corresponding count values, COUNT0 and COUNT1 (which, as mentioned, are provided via an immediate or second source register/memory location).

The shift direction (i.e., right or left) is determined by the specific instruction being executed. As mentioned, DVPSLSUD and DVPSLVSUD perform left-shift operations while DVPSRRUD and DVPSRVRUD perform right-shift operations. Because the values are unsigned, logical right-shift operations are performed.

The resulting shifted values are provided to rounding/saturation units **3122** and **3121** which perform the rounding/saturation operations described herein to generate the four 32-bit results **3151-3154** in the 128-bit destination register **3160**.

As in prior embodiments, a control/status register **3112** such as an MXCSR register, includes a set of rounding control bits which indicate a rounding mode to be used by the rounding/saturation units **3122** and **3121**. In one particular implementation, the rounding control bits **3112**A comprise two bits in an IRM control field of the MXCSR register **3112**.

Additionally, in some embodiments, the rounding/saturation units **3122** and **3121** can determine whether saturation is to be performed based upon detecting overflow or underflow. For example, when a saturation condition is detected, each of the final 32-bit results **3151-3154** can be saturated to a most positive or most negative value. In the code sequence listed above, the SAT_POS0, SAT_POS1, SAT_POS2, and SAT_POS3 values are set to 1 when the left shift operation performed by DVPSLSUD and DVPSLVSUD shifts out a binary 1. In one embodiment, this causes corresponding saturation bit(s) **3112**B in the control register **3112** to be set to 1. Thus, a saturation can be determined based on the value of SAT_POS0-SAT_POS3 during left shift operations and reflected in the corresponding saturation bit(s) **3112**B for use by subsequent operations. In one embodiment, the SAT_POS0-SAT_POS3 values and/or the corresponding saturation bit(s) **3112**B are maintained at 0 for the DVPSRRUD or DVPSRVRUD instructions because the most significant is are maintained during right shift operations.

As mentioned, a saturation field **3112**B of the control/status register(s) **3112** can be set to indicate saturation. For example, the MXCSR.SAT bit (e.g, MXCSR[20]) can be set to indicate that the saturation has occurred. Thus, in some embodiments, a value in a saturation field **3112**B of the control/status register(s) **3112** can be updated when the execution circuitry **3100** detects that saturation has occurred responsive to the prior executed operations on the data values. In some embodiments the saturation field **3112**B can be a single bit location that can ordinarily be a zero but set to a value of one when saturation is detected.

Following rounding/saturation, the 32-bit result values **3151-3154** are stored in bits 31:0, 63:32, 95:64, and 127:96 of the 128-bit destination register **3160**.

A method for performing a left-shift unsigned doubleword SRS instruction is illustrated in FIG. **32**. The method may be performed on the processor or system architectures described herein, but is not limited to any particular processor or system architecture.

At **3201** a left-shift unsigned doubleword SRS instruction is fetched, the instruction including a first source operand to identify first through fourth source values, a second source or immediate to identify first through fourth left shift values, and a destination operand to identify first through fourth result values corresponding to the first through fourth source values, respectively.

At **3202**, the left-shift unsigned doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the left-shift unsigned doubleword SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **3203**, the left-shift unsigned doubleword SRS instruction is executed to left-shift the first through fourth source values based on the first through fourth left shift values of the second source operand or immediate to generate first through fourth shifted values. A saturation indication is stored if a saturation condition is detected resulting from the left shift of the first through fourth source values (e.g., from shifting out a most-significant 1). The first through fourth shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or the saturation indication, respectively, to generate the first through fourth result values in the destination register **2160**. The first through fourth result values are then stored in corresponding first through fourth locations of a destination register.

A method for performing a right-shift unsigned doubleword SRS instruction is illustrated in FIG. **33**. The method may be performed on the processor and system architectures described herein, but is not limited to any particular processor or system architecture.

At **3301** a right-shift unsigned doubleword SRS instruction is fetched, the instruction including a first source operand to identify first through fourth source values, a second source or immediate to identify first through fourth right shift values, and a destination operand to identify first through fourth result values corresponding to the first through fourth source values, respectively.

At **3302**, the right-shift unsigned doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the right-shift/round/saturate instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **3303**, the right-shift unsigned doubleword SRS instruction is executed to right-shift the first through fourth source values based on the first through fourth right-shift values of the second source operand or immediate to generate first through fourth shifted values. The first through fourth shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or a saturation indication, respectively, to generate the first through fourth result values. The first through fourth result values are then stored in corresponding first through fourth locations of a destination register.

The following pseudocode sequence defines the operation for one embodiment of the signed doubleword SRS instructions DVPSRARD, DVPSRAVRD, DVPSLSD, and DVPSLVSD:

```
TEMP[127:0] ← SRC2[127:0];
COUNT0[4:0] ← (imm8[4:0] OR SRC3[4:0]);
COUNT1[4:0] ← (imm8[5:0] OR SRC3[36:32]);
COUNT2[4:0] ← (imm8[5:0] OR SRC3[68:64]);
COUNT3[4:0] ← (imm8[5:0] OR SRC3[100:96]);
IF (COUNT0 > 31)
  COUNT0[31:0] ← 32;
FI;
IF (COUNT1 > 31)
  COUNT1[31:0] ← 32;
FI;
IF (COUNT2 > 31)
  COUNT2[31:0] ← 32;
FI;
IF (COUNT3 > 31)
  COUNT3[31:0] ← 32;
FI;
SIGN_BIT0 ← SRC2[31];
SIGN_BIT1 ← SRC2[63];
SIGN_BIT2 ← SRC2[95];
SIGN_BIT3 ← SRC2[127];
SAT_POS0 ← 0;
SAT_POS1 ← 0;
SAT_POS2 ← 0;
SAT_POS3 ← 0;
SAT_NEG0 ← 0;
SAT_NEG1 ← 0;
SAT_NEG2 ← 0;
SAT_NEG3 ← 0;
TEMP0[31:0] ← 0;
TEMP1[31:0] ← 0;
TEMP2[31:0] ← 0;
TEMP3[31:0] ← 0;
TEMP4[31:0] ← 0;
TEMP5[31:0] ← 0;
TEMP6[31:0] ← 0;
TEMP7[31:0] ← 0;
(* First Dword Accumulator *)
DO WHILE (COUNT0 != 0)
      IF (Instruction is DVPSLSD OR DVPSLVSD) THEN
          SAT_POS0 ← ((~SIGN_BIT0 &TEMP[30]) ? 1 : 0) | | SAT_POS0 ;
            (* Check Sign Bit for Positive Saturation *)
          SAT_NEG0 ← ((SIGN_BIT0 & ~TEMP[30]) ? 1 : 0) | | SAT_NEG0 ;
            (* Check Sign Bit for Negative Saturation *)
          TEMP [31:0] ← {TEMP[30:0], 1'b0}) ;(* Left Shift *)
      ELSE (* Instruction is DVPSRARD OR DVPSRAVRD *)
          SAT_POS0 ← 0;
          SAT_NEG0 ← 0;
          TEMP0[31:0] ← {TEMP[0], TEMP0[31:1]}; (* Shifted bits *)
          TEMP[31:0] ← {TEMP[31], TEMP[31:1]}; (* Arith Rt Shift *)
          COUNT0 ← (COUNT0 - 1);
ENDWHILE;
(* Second Dword Accumulator *)
DO WHILE (COUNT1 != 0)
  IF (Instruction is DVPSLSD OR DVPSLVSD) THEN
        SAT_POS1 ← ((~SIGN_BIT1 &TEMP[62]) ? 1 : 0) | | SAT_POS1 ;
      (* Check Sign Bit for Positive Saturation *)
        SAT_NEG1 ← ((SIGN_BIT1 & ~TEMP[62]) ? 1 : 0) | | SAT_NEG1 ;
      (* Check Sign Bit for Negative Saturation *)
        TEMP [63:32] ← {TEMP[62:32], 1'b0}) ;(* Left Shift *)
```

-continued

```
    ELSE (* Instruction is DVPSRARD OR DVPSRAVRD *)
        SAT_POS1 ← 0;
        SAT_NEG1 ← 0;
        TEMP1[31:0] ← {TEMP[32], TEMPl[31:1]}; (* Shifted bits *)
        TEMP[63:32] ← {TEMP[63], TEMP[63:33]}; (* Arith Right Shift *)
        COUNT1 ← (COUNT1 − 1);
ENDWHILE;
(* Third Dword Accumulator *)
DO WHILE (COUNT2 != 0)
      IF (Instruction is DVPSLSD OR DVPSLVSD) THEN
          SAT_POS2 ← ((~SIGN_BIT2 &TEMP[94]) ? 1 : 0) | | SAT_POS2 ;
            (* Check Sign Bit for Positive Saturation *)
          SAT_NEG2 ← ((SIGN_BIT2 & ~TEMP[94]) ? 1 : 0) | | SAT_NEG2 ;
            (* Check Sign Bit for Negative Saturation *)
          TEMP[95:64] ← {TEMP[94:64], 1'b0}) ;(* Left Shift *)
  ELSE (* Instruction is DVPSRARD OR DVPSRAVRD *)
            SAT_POS2 ← 0;
            SAT_NEG2 ← 0;
            TEMP2[31:0] ← {TEMP[64], TEMP2[31:1]}; (* Shifted bits *)
            TEMP[95:64] ← {TEMP[95], TEMP[95:65]}; (* Arith Rt Shft *)
            COUNT2 ← (COUNT2 − 1);
ENDWHILE;
(* Fourth Dword Accumulator *)
DO WHILE (COUNT3 != 0)
  IF (Instruction is DVPSLSD OR DVPSLVSD) THEN
        SAT_POS3 ← ((~SIGN_BIT3 &TEMP[126]) ? 1 : 0) | |
        SAT_POS3 ; (* Check Sign Bit for Positive Saturation *)
        SAT_NEG3 ← ((SIGN_BIT3 & ~TEMP[126]) ? 1 : 0) | |
        SAT_NEG3 ; (* Check Sign Bit for Negative Saturation *)
        TEMP [127:96] ← {TEMP[126:64], 1'b0}) ; (* Left Shift *)
  ELSE (* Instruction is DVPSRARD OR DVPSRAVRD *)
        SAT_POS3 ← 0;
        SAT_NEG3 ← 0;
        TEMP3[31:0] ← {TEMP[96], TEMP3[31:1]}; (* Shifted bits *)
        TEMP[127:96] ← {TEMP[127], TEMP[127:97]} ;
          (* Arithmetic Right Shift *)
        COUNT3 ← (COUNT3 − 1);
ENDWHILE;
IF (Instruction is DVPSLSD OR DVPSLVSD) THEN
      RoundBit0 ← 0;
      RoundBit1 ← 0;
      RoundBit2 ← 0;
      RoundBit3 ← 0;
ELSE (* Instruction is DVPSRARD OR DVPSRAVRD *)
      RoundBit0 ← CalcShiftSignedDwordRoundBit({TEMP[0],TEMP0[31:0]},
      SIGN_BIT0, MXCSR.IRM[l:0], RoundBit0);
      RoundBit1 ←
      CalcShiftSignedDwordRoundBit({TEMP[32],TEMPl[31:0]}, SIGN_BIT1,
      MXCSR.IRM[1:0], RoundBit1);
      RoundBit2 ←
      CalcShiftSignedDwordRoundBit({TEMP[64],TEMP2[31:0]}, SIGN_BIT2,
      MXCSR.IRM[1:0], RoundBit2);
      RoundBit3 ←
      CalcShiftSignedDwordRoundBit({TEMP[96],TEMP3[31:0]}, SIGN_BIT3,
      MXCSR.IRM[1:0], RoundBit3);
IF (Instruction is DVPSLSD OR DVPSLVSD) THEN
      DEST[31:0] ← AddBitSaturateToSignedDword(TEMP[31:0], 0,
      SAT_POS0, SAT_NEG0, DEST[31:0]);
      DEST[63:32] ← AddBitSaturateToSignedDword(TEMP[63:32], 0,
      SAT_POS1, SAT_NEG1, DEST[63:32]);
      DEST[95:64] ← AddBitSaturateToSignedDword(TEMP[95:64], 0,
      SAT_POS2, SAT_NEG2, DEST[95:64]);
      DEST[127:96] ← AddBitSaturateToSignedDWord(TEMP[127:96], 0,
      SAT_POS3, SAT_NEG3, DEST[127:96]);
ELSE (* Instruction is DVPSRARD OR DVPSRAVRD *)
      DEST[31:0] ← (TEMP[31:0] + {31'b0,RoundBit0});
          (* Add Rounding bit with wrapping *)
      DEST[63:32] ← (TEMP[63:32] + {31'b0,RoundBit1});
          (* Add Rounding bit with wrapping *)
      DEST[95:64] ← (TEMP[95:64] + {31'b0,RoundBit2});
          (* Add Rounding bit with wrapping *)
      DEST[127:96] ← (TEMP[127:96] + {31'b0,RoundBit3});
          (* Add Rounding bit with wrapping *)
```

Thus, as in prior embodiments, the signed doubleword SRS instructions use positive saturation flags (SAT_POS0, SAT_POS1, SAT_POS2, and SAT_POS3) to indicate positive saturation and negative saturation flags (SAT_NEG0, SAT_NEG1, SAT_NEG2, and SAT_NEG3) to indicate positive or negative saturation, respectively. The rounding/saturation units **3121-3124** can then evaluate the positive and negative saturation flags to determine whether the result

values are to be saturated negatively or positively. The signed SRS instructions must also process the sign values (SIGN_BIT0, SIGN_BIT1, SIGN_BIT2, and SIGN_BIT3) for each of the signed doubleword source values.

DVPSLSD and DVPSLVSD perform left-shift operations and DVPSRARD and DVPSRAVRD perform right-shift operations. Because the values are signed, DVPSRARD and DVPSRAVRD perform arithmetic shift-right operations in which the sign bit is shifted in from the left as the least significant bit values are shifted out to the right.

A method for performing a left-shift signed doubleword SRS instruction is illustrated in FIG. **34**. The method may be performed on the processor or system architectures described herein, but is not limited to any particular processor or system architecture.

At **3401** a left-shift signed doubleword SRS instruction is fetched, the instruction including a first source operand to identify first through fourth source values, a second source or immediate to identify first through fourth left shift values, and a destination operand to identify first through fourth result values corresponding to the first through fourth source values, respectively.

At **3402**, the left-shift signed doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the left-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **3403**, the left-shift signed doubleword SRS instruction is executed to left-shift the first through fourth signed source values based on the first through fourth left shift values of the second source operand or immediate to generate first through fourth signed, shifted values. A positive or negative saturation indication is stored if a positive/negative saturation condition is detected resulting from the left shift of the first through fourth source values. The first through fourth signed shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or the positive/negative saturation indication, respectively, to generate the first through fourth result values in the destination register **2160**. The first through fourth result values are then stored in corresponding first through fourth locations of a destination register.

A method for performing a right-shift signed doubleword SRS instruction is illustrated in FIG. **35**. The method may be performed on the processor and system architectures described herein, but is not limited to any particular processor or system architecture.

At **3501** a right-shift signed doubleword SRS instruction is fetched, the instruction including a first source operand to identify first through fourth source values, a second source or immediate to identify first through fourth right shift values, and a destination operand to identify first through fourth result values corresponding to the first through fourth source values, respectively.

At **3502**, the right-shift signed doubleword SRS instruction is decoded. In a microcoded implementation, for example, decoding of the right-shift SRS instruction generates sequences of microoperations which are then executed by the execution circuitry.

At **3503**, the right-shift SRS instruction is executed to right-shift the first through fourth source values based on the first through fourth right-shift values of the second source operand or immediate to generate first through fourth shifted values. The first through fourth shifted values may then be rounded and/or saturated in accordance with a rounding mode and/or a positive/negative saturation indication, respectively, to generate the first through fourth result values. The first through fourth result values are then stored in corresponding first through fourth locations of a destination register.

One embodiment includes another pair of signed right-shift doubleword SRS instructions which operate using logical right-shift operations (instead of arithmetic right shift operations as in the prior signed embodiments). In order to do so, these instructions maintain the sign bits and all shifted out bits in temporary storage to be used for the rounding operations. The signed right-shift doubleword SRS instructions of this embodiment are defined as follows:

DVPSRLRD xmm1, xmm2/m128, imm8

DVPSRLVRD xmm1, xmm2, xmm3/m128

The following pseudocode sequence defines the operation of the above instructions in accordance with one embodiment:

```
TEMP[127:0] ← SRC2[127:0];
COUNT0[4:0] ← (imm8[4:0] OR SRC3[4:0]);
COUNT1[4:0] ← (imm8[4:0] OR SRC3[36:32]);
COUNT2[4:0] ← (imm8[4:0] OR SRC3[68:64]);
COUNT3[4:0] ← (imm8[4:0] OR SRC3[100:96]);
SIGN_BIT0 ← SRC2[31];
SIGN_BIT1 ← SRC2[63];
SIGN_BIT2 ← SRC2[95];
SIGN_BIT3 ← SRC2[127];
TEMP0[31:0] ← 0;
TEMP1[31:0] ← 0;
TEMP2[31:0] ← 0;
TEMP3[31:0] ← 0;
DO WHILE (COUNT0 != 0)
    IF (* Instruction is DVPSRLRD or DVPSRLVRD *)
      TEMP0[31:0] ← {TEMP[0], TEMP0[31:1]};
        (* Shifted out bits to be used for Rounding *)
      TEMP[31:0] ← {1'b0,TEMP[31:1]} ; (* Logical Shift Right *)
      COUNT0 ← (COUNT0 – 1);
ENDWHILE;
DO WHILE (COUNT1 != 0)
    IF (* Instruction is DVPSRLRD or DVPSRLVRD *)
        TEMP1[31:0] ← {TEMP[32], TEMP1[31:1]};
        (* Shifted out bits to be used for Rounding *)
        TEMP[63:32] ← {1'b0,TEMP[63:33]} ; (* Logical Shift Right *)
    COUNT1 ← (COUNT1 – 1);
ENDWHILE;
DO WHILE (COUNT2 != 0)
```

-continued

```
    IF (* Instruction is DVPSRLRD or DVPSRLVRD *)
    TEMP2[31:0] ← {TEMP[64], TEMP2[31:1]};
    (* Shifted out bits to be used for Rounding *)
    TEMP[95:64] ← {1'b0,TEMP[95:65]} ; (* Logical Shift Right *)
    COUNT2 ← (COUNT2 – 1);
ENDWHILE;
DO WHILE (COUNT3 != 0)
    IF (* Instruction is DVPSRLRD or DVPSRLVRD *)
    TEMP3[31:0] ← {TEMP[96], TEMP3[31:1]};
    (* Shifted out bits to be used for Rounding *)
    TEMP[127:96] ← {1'b0,TEMP[127:97]} ; (* Logical Shift Right *)
    COUNT3 ← (COUNT3 – 1);
ENDWHILE;
IF (* Instruction is DVPSRLRD or DVPSRLVRD *)
    RoundBit0 ← CalcShiftSignedDwordRoundBit((TEMP[0],
    TEMP0[31:0]}, SIGN_BIT0, MXCSR.IRM[1:0], RoundBit0);
    RoundBit1 ←
    CalcShiftSignedDwordRoundBit({TEMP[32],TEMPl[31:0]}, SIGN_BIT1,
    MXCSR.IRM[1:0], RoundBit1);
    RoundBit2 ←
    CalcShiftSignedDwordRoundBit({TEMP[64],TEMP2[31:0]}, SIGN_BIT2,
    MXCSR.IRM[1:0], RoundBit2);
    RoundBit3 ←
    CalcShiftSignedDwordRoundBit({TEMP[96],TEMP3[31:0]}, SIGN_BIT3,
    MXCSR.IRM[1:0], RoundBit3);
IF (* Instruction is DVPSRLRD or DVPSRLVRD *)
  DEST[31:0] ← (TEMP[31:0] + {31'b0,RoundBit0});
    (* Add Rounding bit with wrapping *)
  DEST[63:32] ← (TEMP[63:32] + {31'b0,RoundBit1});
    (* Add Rounding bit with wrapping *)
  DEST[95:64] ← (TEMP[95:64] + {31'b0, RoundBit2});
    (* Add Rounding bit with wrapping *)
  DEST[127:96] ← (TEMP[127:96] + {31'b0,RoundBit3});
    (* Add Rounding bit with wrapping *)
```

References to "one embodiment," "an embodiment," "an example embodiment," etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment may not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

EXAMPLES

The following are example implementations of different embodiments of the invention.

Example 1. An apparatus comprising: a decoder to decode an instruction having fields for a first packed data source operand to provide a first source data element and a second source data element, a second packed data source operand or immediate to provide a first shift value and a second shift value corresponding to the first source data element and second source data element, respectively, and a packed data destination operand to indicate a first result value and a second result value corresponding to the first source data element and second source data element, and execution circuitry to execute the decoded instruction to: shift the first source data element by an amount based on the first shift value to generate a first shifted data element; shift the second source data element by an amount based on the second shift value to generate a second shifted data element; update a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first and/or second source data elements; round and/or saturate the first and second shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first and second result data elements; and store the first result value and the second result value in a first data element location and a second data element location in a destination register.

Example 2. The apparatus of example 1 wherein the first packed data source operand is to further provide a third source data element and a fourth source data element, the second packed data source operand or immediate is to further provide a third shift value and a fourth shift value corresponding to the third source data element and fourth source data element, and the packed data destination operand is to indicate a third result value and a fourth result value corresponding to the third source data element and the fourth source data element, and the execution circuitry to execute the decoded instruction to: shift the third source data element by an amount based on the third shift value to generate a third shifted data element; shift the fourth source data element by an amount based on the fourth shift value to generate a fourth shifted data element; update a saturation indicator responsive to detecting a saturation condition resulting from the shift of the third and/or fourth source data elements; round and/or saturate the third and fourth shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the third and fourth result data elements; and store the third result value and the fourth result value in a third data element location and a fourth data element location in a destination register.

Example 3. The apparatus of example 1 wherein the execution circuitry is to right-shift the first and second source data elements by the amount based on the first and second shift values, respectively.

Example 4. The apparatus of example 3 wherein the first and second source data elements and the first and second result values comprise signed word or signed doubleword values.

Example 5. The apparatus of example 4 wherein the right-shift comprises an arithmetic right shift in which a sign value of the first and second source data elements is preserved.

Example 6. The apparatus of example 1 wherein the execution circuitry is to left-shift the first and second source data elements by the amount based on the first and second shift values, respectively.

Example 7. The apparatus of example 5 wherein the execution circuitry is to set a positive saturation flag upon detecting a saturation condition associated with the left-shift, the positive saturation flag to be used during the round and/or saturate operations to generate positively saturated values for the first result value or the second result value.

Example 8. The apparatus of example 4 wherein the execution circuitry is to set a positive saturation flag upon detecting a first saturation condition associated with the left-shift and a negative saturation flag associated with a second saturation condition associated with the left-shift, one of the positive and negative saturation flags to be used during the round and/or saturate operations to generate positively or negatively saturated values for the first result value or the second result value.

Example 9. A method comprising: decoding an instruction having fields for a first packed data source operand to provide a first source data element and a second source data element, a second packed data source operand or immediate to provide a first shift value and a second shift value corresponding to the first source data element and second source data element, respectively, and a packed data destination operand to indicate a first result value and a second result value corresponding to the first source data element and second source data element, and executing the decoded instruction to perform the operations of: shifting the first source data element by an amount based on the first shift value to generate a first shifted data element; shifting the second source data element by an amount based on the second shift value to generate a second shifted data element; updating a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first and/or second source data elements; rounding and/or saturating the first and second shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first and second result data elements; and store the first result value and the second result value in a first data element location and a second data element location in a destination register.

Example 10. The method of example 9 wherein the first packed data source operand is to further provide a third source data element and a fourth source data element, the second packed data source operand or immediate is to further provide a third shift value and a fourth shift value corresponding to the third source data element and fourth source data element, and the packed data destination operand is to indicate a third result value and a fourth result value corresponding to the third source data element and the fourth source data element, the method further comprising: shifting the third source data element by an amount based on the third shift value to generate a third shifted data element; shifting the fourth source data element by an amount based on the fourth shift value to generate a fourth shifted data element; updating a saturation indicator responsive to detecting a saturation condition resulting from the shifting of the third and/or fourth source data elements; rounding and/or saturating the third and fourth shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the third and fourth result data elements; and storing the third result value and the fourth result value in a third data element location and a fourth data element location in a destination register.

Example 11. The method of example 9 wherein shifting comprises right-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

Example 12. The method of example 9 wherein the first and second source data elements and the first and second result values comprise signed word or signed doubleword values.

Example 13. The method of example 12 wherein the right-shifting comprises an arithmetic right shift in which a sign value of the first and second source data elements is preserved.

Example 14. The method of example 9 wherein shifting comprises left-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

Example 15. The method of example 13 further comprising: setting a positive saturation flag upon detecting a saturation condition associated with the left-shift, the positive saturation flag to be used during the round and/or saturate operations to generate positively saturated values for the first result value or the second result value.

Example 16. The method of example 12 further comprising: setting a positive saturation flag upon detecting a first saturation condition associated with the left-shift and a negative saturation flag associated with a second saturation condition associated with the left-shift, one of the positive and negative saturation flags to be used during the round and/or saturate operations to generate positively or negatively saturated values for the first result value or the second result value.

Example 17. A machine-readable medium having program code stored thereon which, when executed by a machine, causes the machine to perform the operations of: decoding an instruction having fields for a first packed data source operand to provide a first source data element and a second source data element, a second packed data source operand or immediate to provide a first shift value and a second shift value corresponding to the first source data element and second source data element, respectively, and a packed data destination operand to indicate a first result value and a second result value corresponding to the first source data element and second source data element, and executing the decoded instruction to perform the operations of: shifting the first source data element by an amount based on the first shift value to generate a first shifted data element; shifting the second source data element by an amount based on the second shift value to generate a second shifted data element; updating a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first and/or second source data elements; rounding and/or saturating the first and second shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first and second result data elements; and store the first result value and the second result value in a first data element location and a second data element location in a destination register.

Example 18. The machine-readable medium of example 17 wherein the first packed data source operand is to further provide a third source data element and a fourth source data

element, the second packed data source operand or immediate is to further provide a third shift value and a fourth shift value corresponding to the third source data element and fourth source data element, and the packed data destination operand is to indicate a third result value and a fourth result value corresponding to the third source data element and the fourth source data element, the machine-readable medium further comprising program code to cause the machine to perform the operations of: shifting the third source data element by an amount based on the third shift value to generate a third shifted data element; shifting the fourth source data element by an amount based on the fourth shift value to generate a fourth shifted data element; updating a saturation indicator responsive to detecting a saturation condition resulting from the shifting of the third and/or fourth source data elements; rounding and/or saturating the third and fourth shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the third and fourth result data elements; and storing the third result value and the fourth result value in a third data element location and a fourth data element location in a destination register.

Example 19. The machine-readable medium of example 17 wherein shifting comprises right-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

Example 20. The machine-readable medium of example 17 wherein the first and second source data elements and the first and second result values comprise signed word or signed doubleword values.

Example 21. The machine-readable medium of example 20 wherein the right-shifting comprises an arithmetic right shift in which a sign value of the first and second source data elements is preserved.

Example 22. The machine-readable medium of example 17 wherein shifting comprises left-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

Example 23. The machine-readable medium of example 21 program code to cause the machine to perform the operations of: setting a positive saturation flag upon detecting a saturation condition associated with the left-shift, the positive saturation flag to be used during the round and/or saturate operations to generate positively saturated values for the first result value or the second result value.

Example 24. The machine-readable medium of example 20 further comprising program code to cause the machine to perform the operations of: setting a positive saturation flag upon detecting a first saturation condition associated with the left-shift and a negative saturation flag associated with a second saturation condition associated with the left-shift, one of the positive and negative saturation flags to be used during the round and/or saturate operations to generate positively or negatively saturated values for the first result value or the second result value.

Moreover, in the various embodiments described above, unless specifically noted otherwise, disjunctive language such as the phrase "at least one of A, B, or C" is intended to be understood to mean either A, B, or C, or any combination thereof (e.g., A, B, and/or C). As such, disjunctive language is not intended to, nor should it be understood to, imply that a given embodiment requires at least one of A, at least one of B, or at least one of C to each be present.

The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense. It will, however, be evident that various modifications and changes may be made thereunto without departing from the broader spirit and scope of the disclosure as set forth in the claims.

Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

We claim:

1. An apparatus comprising:

a decoder to decode an instruction having fields for a first packed data source operand to provide a first source data element, a second source data element, a third source data element, and a fourth source data element, a second packed data source operand or immediate to provide a first shift value, a second shift value, a third shift value, and a fourth shift value corresponding to the first source data element, the second source data element, the third source data element, and the fourth source data element, respectively, and a packed data destination operand to indicate a first result value, a second result value, a third result value, and a fourth result value corresponding to the first source data element, second source data element, the third source data element, and the fourth source data element, and

execution circuitry to execute the decoded instruction to:

shift the first source data element by an amount based on the first shift value to generate a first shifted data element;

shift the second source data element by an amount based on the second shift value to generate a second shifted data element;

shift the third source data element by an amount based on the third shift value to generate a third shifted data element;

shift the fourth source data element by an amount based on the fourth shift value to generate a fourth shifted data element;

update a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first, the second, the third, and/or the fourth source data elements;

round and/or saturate the first, the second, the third, and the fourth shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first, the second, the third, and the fourth result data elements; and

store the first result value, the second result value, the third result value, and the fourth result value, in a first data element location, a second data element location, a third data element location, and a fourth data element location in a destination register.

2. The apparatus of claim **1** wherein the execution circuitry is to right-shift the first and second source data elements by the amount based on the first and second shift values, respectively.

3. The apparatus of claim **2** wherein the first and second source data elements and the first and second result values comprise signed word or signed doubleword values.

4. The apparatus of claim **3** wherein the right-shift comprises an arithmetic right shift in which a sign value of the first and second source data elements is preserved.

5. The apparatus of claim 1 wherein the execution circuitry is to left-shift the first and second source data elements by the amount based on the first and second shift values, respectively.

6. The apparatus of claim 5 wherein the execution circuitry is to set a positive saturation flag upon detecting a saturation condition associated with the left-shift, the positive saturation flag to be used during the round and/or saturate operations to generate positively saturated values for the first result value or the second result value.

7. The apparatus of claim 5 wherein the execution circuitry is to set a positive saturation flag upon detecting a first saturation condition associated with the left-shift and a negative saturation flag associated with a second saturation condition associated with the left-shift, one of the positive and negative saturation flags to be used during the round and/or saturate operations to generate positively or negatively saturated values for the first result value or the second result value.

8. A method comprising:

decoding an instruction having fields for a first packed data source operand to provide a first source data element, a second source data element, a third source data element, and a fourth source data element, a second packed data source operand or immediate to provide a first shift value, a second shift value, a third shift value, and a fourth shift value corresponding to the first source data element, the second source data element, the third source data element, and the fourth source data element, respectively, and a packed data destination operand to indicate a first result value, a second result value, a third result value, and a fourth result value corresponding to the first source data element, second source data element, the third source data element, and the fourth source data element, and

executing the decoded instruction to perform:

shifting the first source data element by an amount based on the first shift value to generate a first shifted data element;

shifting the second source data element by an amount based on the second shift value to generate a second shifted data element;

shift the third source data element by an amount based on the third shift value to generate a third shifted data element;

shift the fourth source data element by an amount based on the fourth shift value to generate a fourth shifted data element;

updating a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first, the second, the third, and/or the fourth source data elements;

rounding and/or saturating the first, the second, the third, and the fourth shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first, the second, the third, and the fourth result data elements; and

store the first result value, the second result value, the third result value, and the fourth result value, in a first data element location, a second data element location, a third data element location, and a fourth data element location in a destination register.

9. The method of claim 8 wherein shifting comprises right-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

10. The method of claim 8 wherein the first and second source data elements and the first and second result values comprise signed word or signed doubleword values.

11. The method of claim 10 wherein the shifting comprises an arithmetic right shift in which a sign value of the first and second source data elements is preserved.

12. The method of claim 8 wherein shifting comprises left-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

13. The method of claim 12 further comprising:

setting a positive saturation flag upon detecting a saturation condition associated with the left-shift, the positive saturation flag to be used during the round and/or saturate operations to generate positively saturated values for the first result value or the second result value.

14. The method of claim 12 further comprising:

setting a positive saturation flag upon detecting a first saturation condition associated with the left-shift and a negative saturation flag associated with a second saturation condition associated with the left-shift, one of the positive and negative saturation flags to be used during the round and/or saturate operations to generate positively or negatively saturated values for the first result value or the second result value.

15. A non-transitory machine-readable medium having program code stored thereon which, when executed by a machine, causes the machine to perform:

decoding an instruction having fields for a first packed data source operand to provide a first source data element, a second source data element, a third source data element, and a fourth source data element, a second packed data source operand or immediate to provide a first shift value, a second shift value, a third shift value, and a fourth shift value corresponding to the first source data element, the second source data element, the third source data element, and the fourth source data element, respectively, and a packed data destination operand to indicate a first result value, a second result value, a third result value, and a fourth result value corresponding to the first source data element, second source data element, the third source data element, and the fourth source data element, and

executing the decoded instruction to perform:

shifting the first source data element by an amount based on the first shift value to generate a first shifted data element;

shifting the second source data element by an amount based on the second shift value to generate a second shifted data element;

shift the third source data element by an amount based on the third shift value to generate a third shifted data element;

shift the fourth source data element by an amount based on the fourth shift value to generate a fourth shifted data element;

updating a saturation indicator responsive to detecting a saturation condition resulting from the shift of the first, the second, the third, and/or the fourth source data elements;

rounding and/or saturating the first, the second, the third, and the fourth shifted data elements in accordance with a specified rounding mode and the saturation indicator, respectively, to generate the first, the second, the third, and the fourth result data elements; and

store the first result value, the second result value, the third result value, and the fourth result value, in a first data element location, a second data element location, a third data element location, and a fourth data element location in a destination register.

16. The non-transitory machine-readable medium of claim **15** wherein shifting comprises right-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

17. The non-transitory machine-readable medium of claim **15** wherein the first and second source data elements and the first and second result values comprise signed word or signed doubleword values.

18. The non-transitory machine-readable medium of claim **17** wherein the shifting comprises an arithmetic right shift in which a sign value of the first and second source data elements is preserved.

19. The non-transitory machine-readable medium of claim **15** wherein shifting comprises left-shifting the first and second source data elements by the amount based on the first and second shift values, respectively.

20. The non-transitory machine-readable medium of claim **19** program code to cause the machine to perform:

setting a positive saturation flag upon detecting a saturation condition associated with the left-shift, the positive saturation flag to be used during the round and/or saturate operations to generate positively saturated values for the first result value or the second result value.

21. The non-transitory machine-readable medium of claim **19** further comprising program code to cause the machine to perform the operations of:

setting a positive saturation flag upon detecting a first saturation condition associated with the left-shift and a negative saturation flag associated with a second saturation condition associated with the left-shift, one of the positive and negative saturation flags to be used during the round and/or saturate operations to generate positively or negatively saturated values for the first result value or the second result value.

* * * * *