US 20260087125A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2026/0087125 A1**

Constable et al. (43) **Pub. Date: Mar. 26, 2026**

(54) **APPARATUS AND METHOD TO INJECT NON-CANONICAL ADDRESSES INTO FAULTING INSTRUCTION OUTPUTS TO MITIGATE TRANSIENT EXECUTION VULNERABILITIES**

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

(72) Inventors: **Scott Constable**, PORTLAND, OR (US); **Jason Agron**, San Jose, CA (US); **Jason Brandt**, Austin, TX (US); **Joseph Nuzman**, Haifa (IL); **Carlos Rozas**, Portland, OR (US); **Fangfei Liu**, Hillsboro, OR (US); **Thomas Unterluggauer**, Carinthia (AT); **Xiang Zou**, Portland, OR (US); **Yuan Xiao**, Chicago, IL (US)

(21) Appl. No.: **18/896,759**

(22) Filed: **Sep. 25, 2024**

Publication Classification

(51) **Int. Cl.**

G06F 21/54 (2013.01)

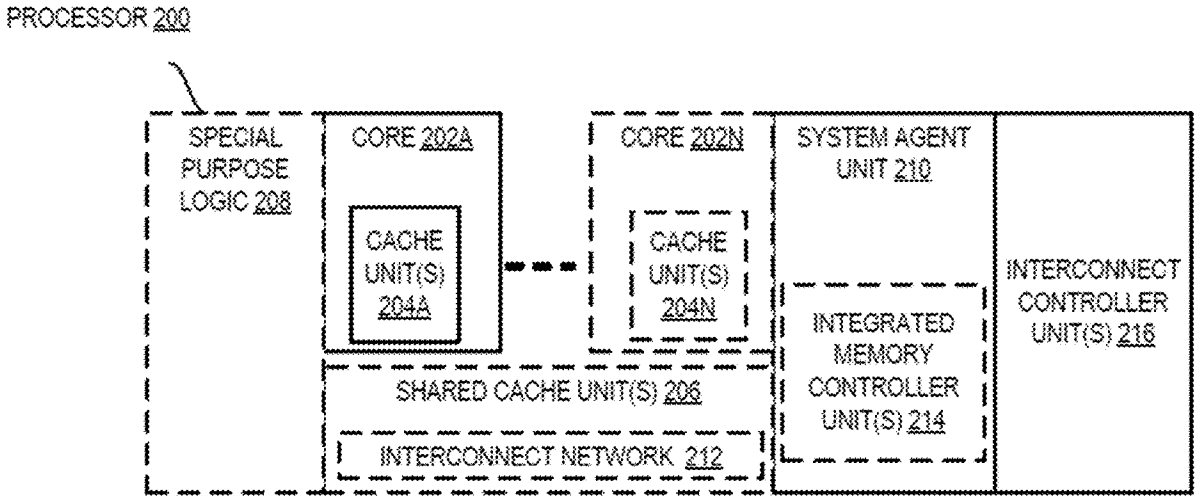
G06F 9/30 (2018.01)

(52) **U.S. Cl.**

CPC ***G06F 21/54*** (2013.01); ***G06F 9/30043*** (2013.01); ***G06F 9/30072*** (2013.01)

(57) **ABSTRACT**

An apparatus and method for injecting non-canonical addresses into instruction outputs to mitigate transient execution vulnerabilities. For example, one embodiment of a method comprises: decoding a sequence of instructions by a decoder of a processor, the sequence of instructions including a conditional instruction; executing the conditional instruction, wherein executing includes: outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be false, and outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.



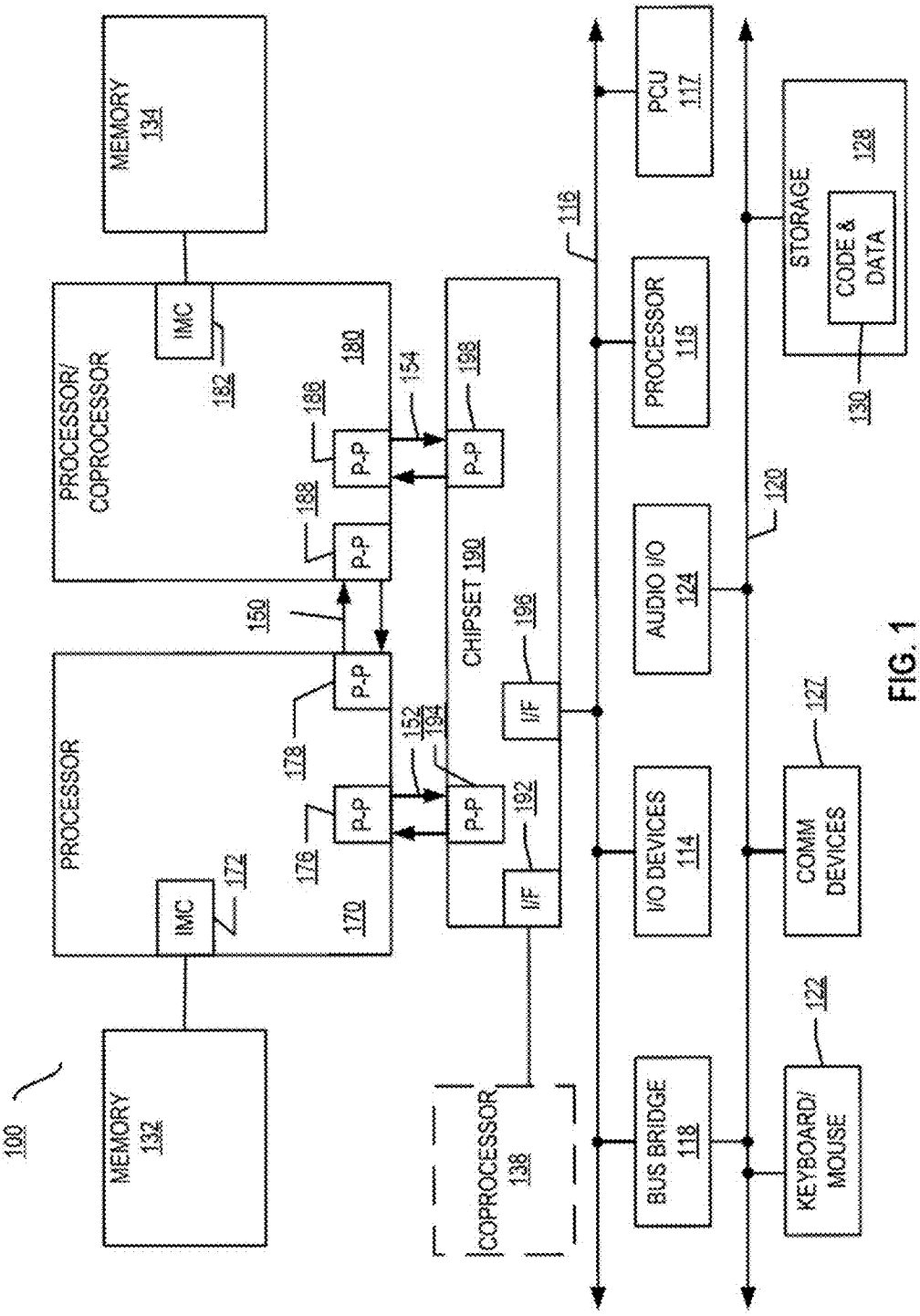

100
MEMORY 132
MEMORY 134
PROCESSOR
IMC 172
170
PROCESSOR/ COPROCESSOR
IMC 182
180
P-P 176
P-P 178
P-P 186
P-P 188
150
152
154
CHIPSET 190
P-P 192
P-P 194
P-P 198
I/F 196
I/F
COPROCESSOR 138
BUS BRIDGE 118
I/O DEVICES 114
AUDIO I/O 124
PROCESSOR 115
PCU 117
116
120
COMM DEVICES 127
KEYBOARD/ MOUSE 122
STORAGE 128
CODE & DATA
130

FIG. 1

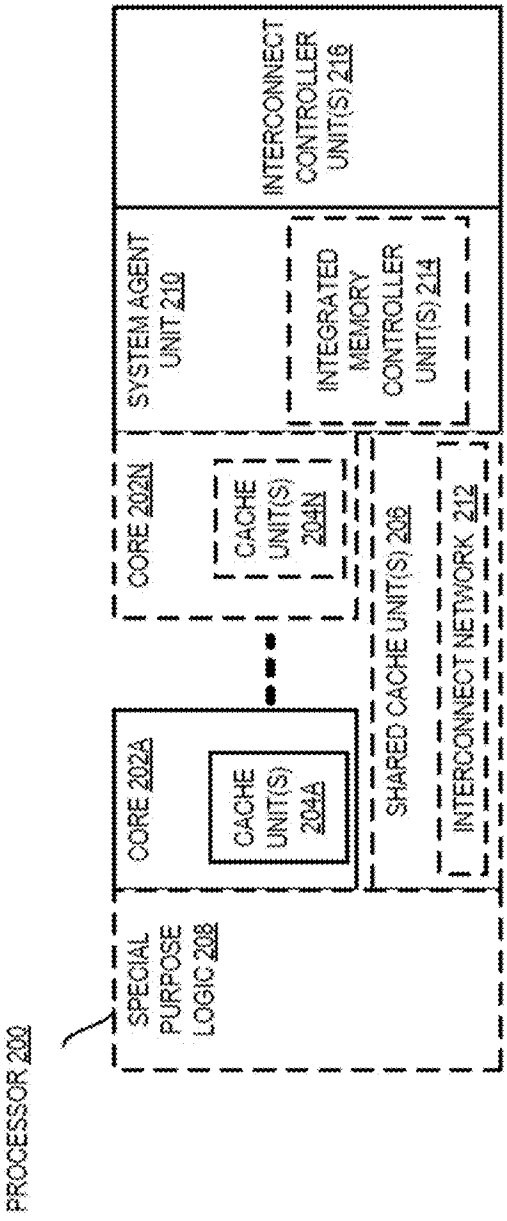

PROCESSOR 200
SPECIAL PURPOSE LOGIC 208
CORE 202A
CACHE UNIT(S) 204A
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) 206
INTERCONNECT NETWORK 212
SYSTEM AGENT UNIT 210
INTEGRATED MEMORY CONTROLLER UNIT(S) 214
INTERCONNECT CONTROLLER UNIT(S) 216

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)

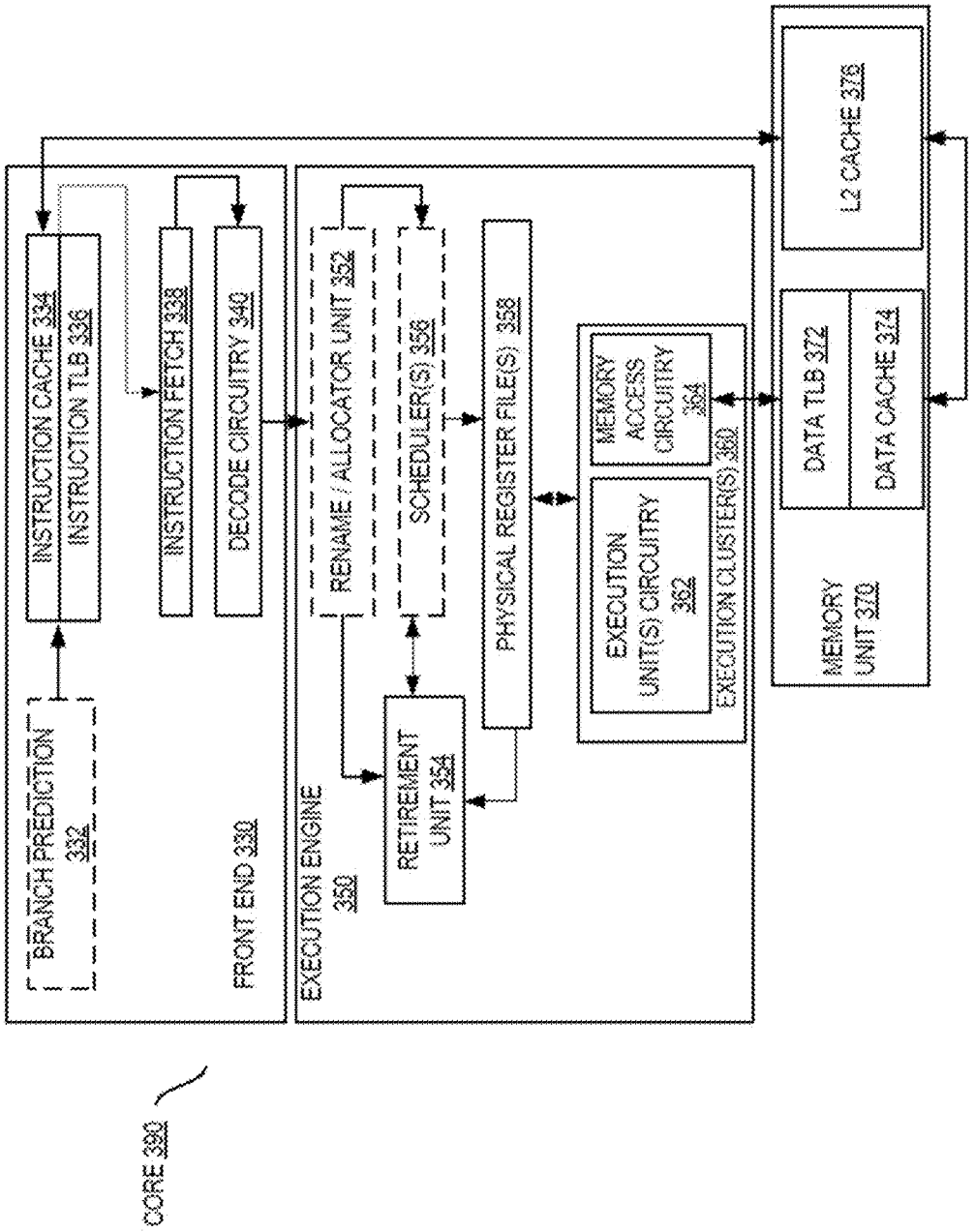

CORE 390
FRONT END 330
BRANCH PREDICTION 332
INSTRUCTION CACHE 334
INSTRUCTION TLB 336
INSTRUCTION FETCH 338
DECODE CIRCUITRY 340
EXECUTION ENGINE 350
RENAME / ALLOCATOR UNIT 352
SCHEDULER(S) 356
RETIREMENT UNIT 354
PHYSICAL REGISTER FILE(S) 358
EXECUTION UNIT(S) CIRCUITRY 362
MEMORY ACCESS CIRCUITRY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

FIG. 3(B)

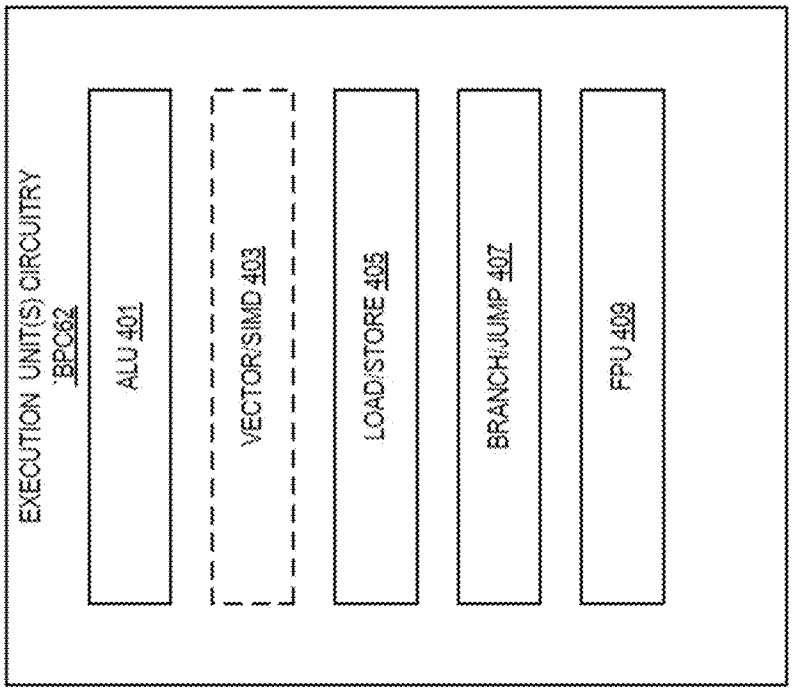

EXECUTION UNIT(S) CIRCUITRY
BPC62
ALU 401
VECTOR/SIMD 403
LOAD/STORE 405
BRANCH/JUMP 407
FPU 409

FIG. 4

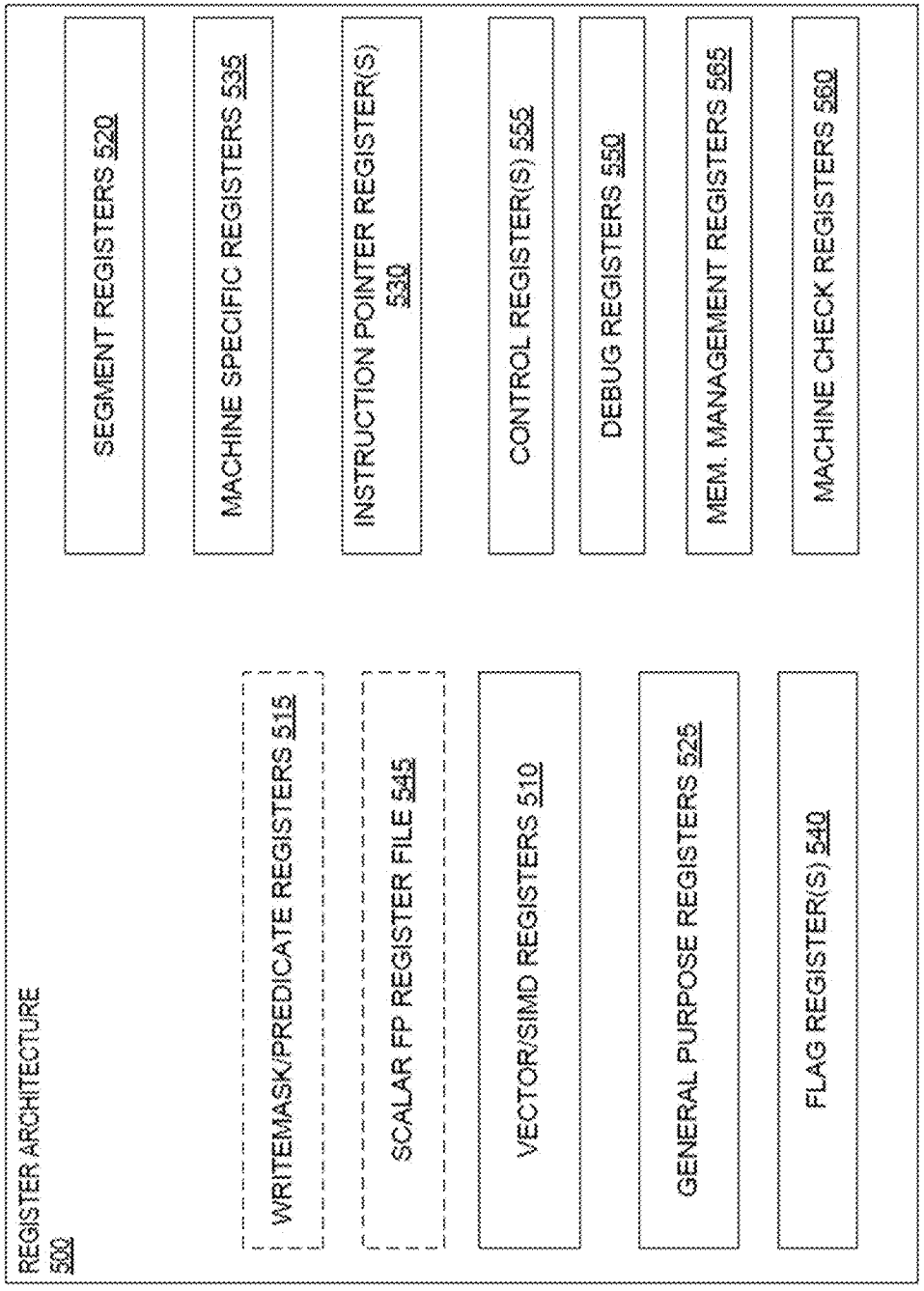

REGISTER ARCHITECTURE
500
WRITEMASK/PREDICATE REGISTERS 515
SCALAR FP REGISTER FILE 545
VECTOR/SIMD REGISTERS 510
GENERAL PURPOSE REGISTERS 525
FLAG REGISTER(S) 540
SEGMENT REGISTERS 520
MACHINE SPECIFIC REGISTERS 535
INSTRUCTION POINTER REGISTER(S) 530
CONTROL REGISTER(S) 555
DEBUG REGISTERS 550
MEM.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

FIG. 5

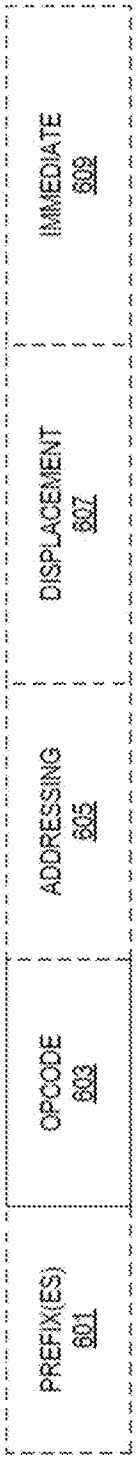

PREFIX(ES)
601
OPCODE
603
ADDRESSING
605
DISPLACEMENT
607
IMMEDIATE
609

FIG. 6

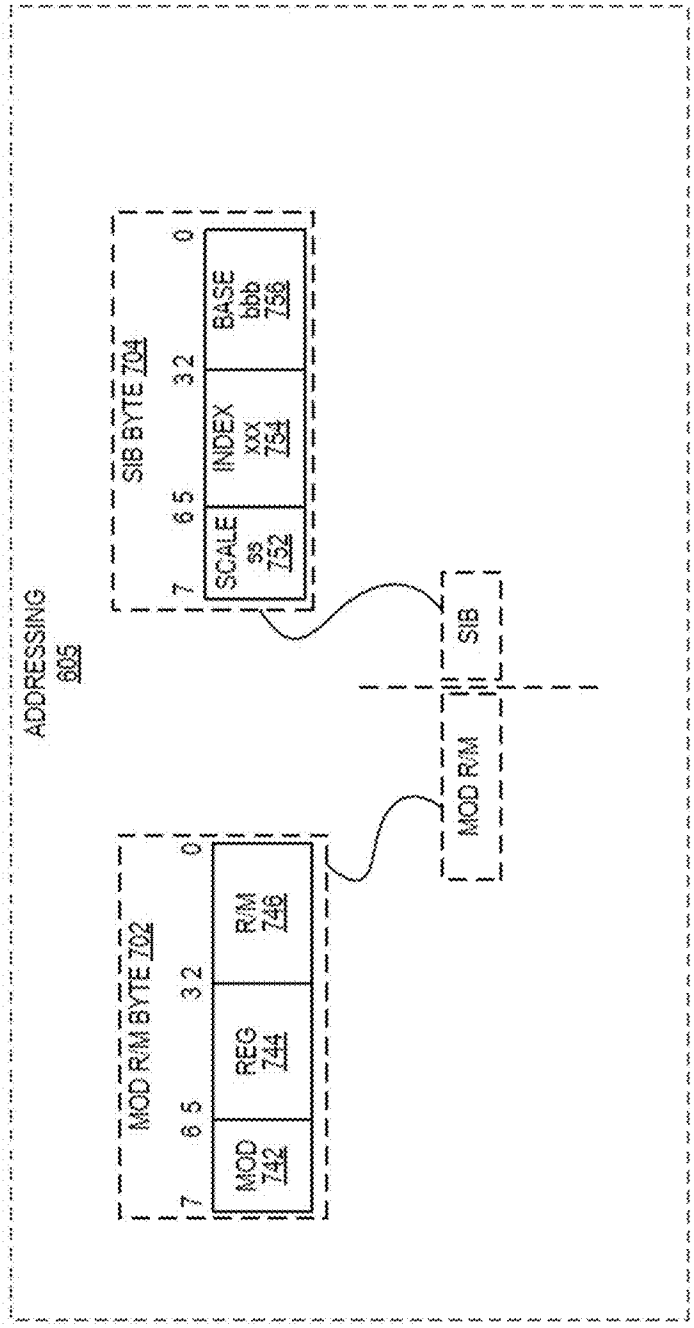

ADDRESSING
605
MOD R/M BYTE 702
7 6 5 3 2 0
MOD 742
REG 744
R/M 746
MOD R/M
SIB
SIB BYTE 704
7 6 5 3 2 0
SCALE ss 752
INDEX xxx 754
BASE bbb 756

FIG. 7

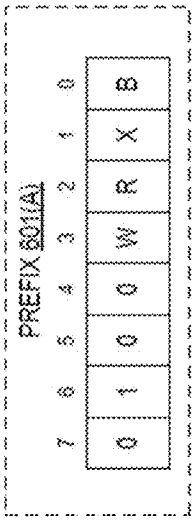


FIG. 8

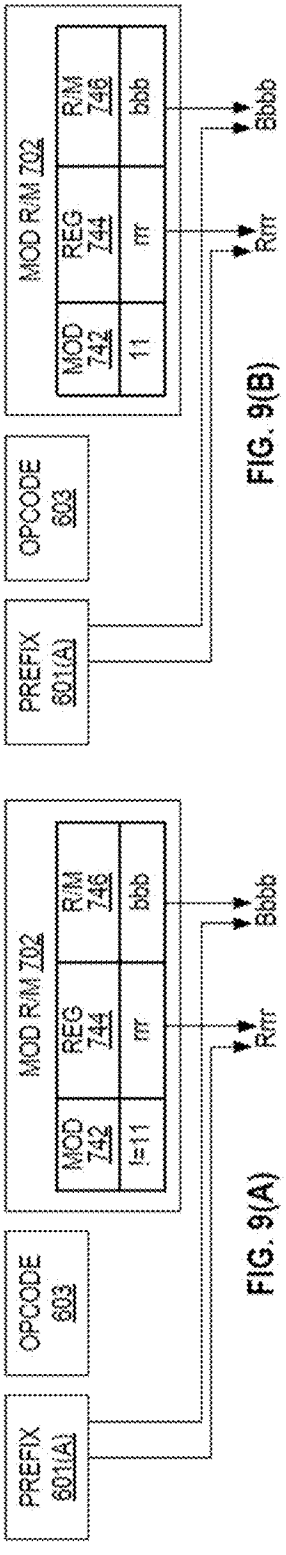


FIG. 9(B)

FIG. 9(A)

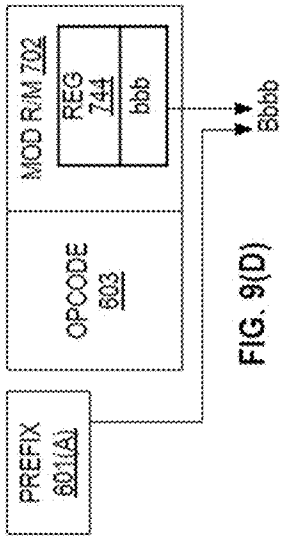


FIG. 9(D)

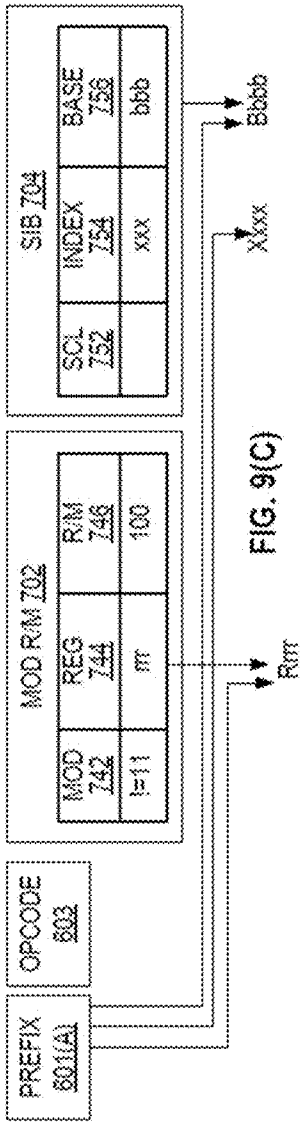


FIG. 9(C)

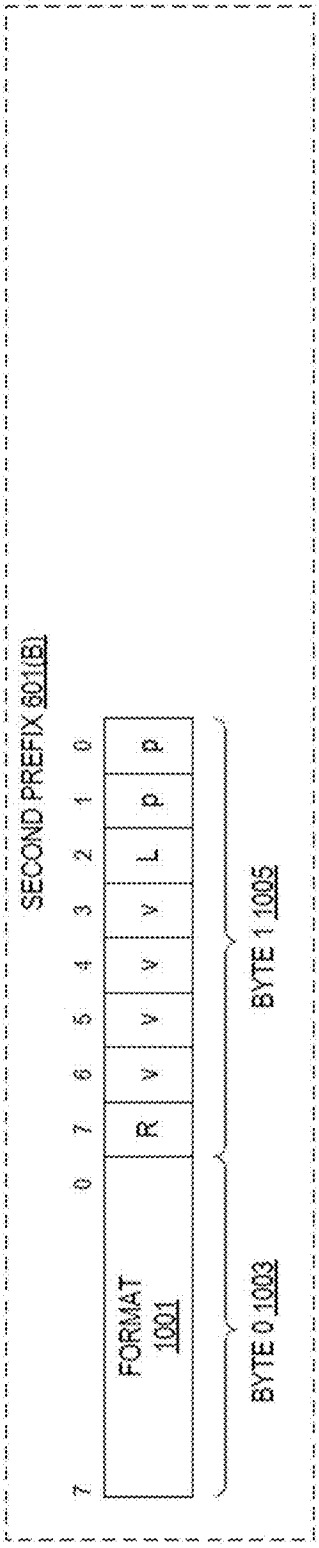


FIG. 10(A)

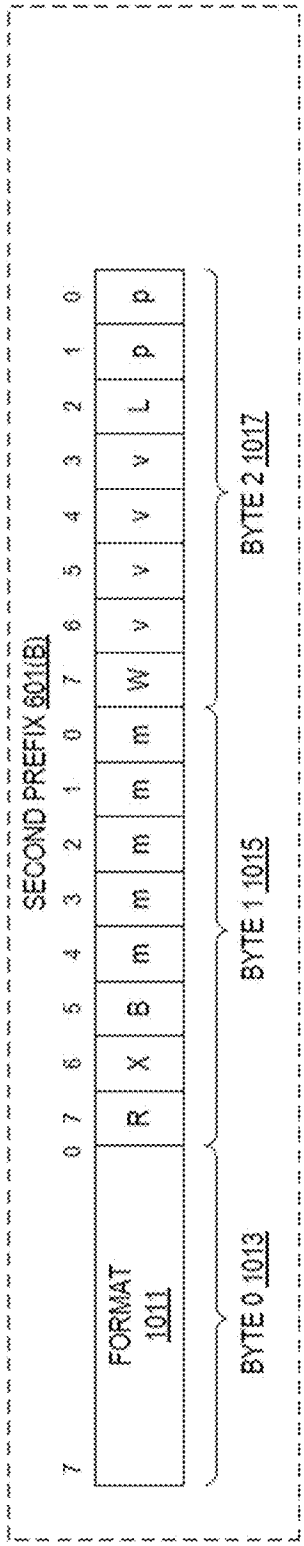


FIG. 10(B)

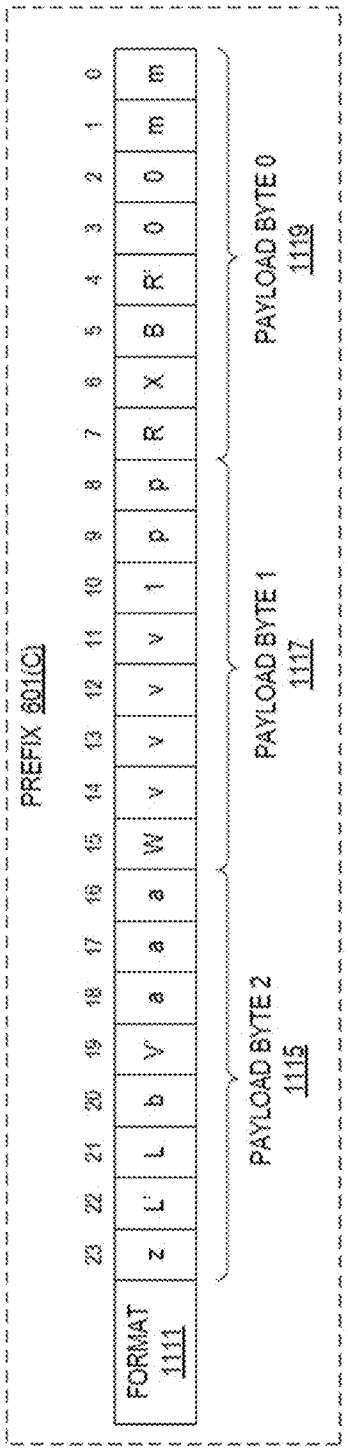

PREFIX 801(C)
23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
FORMAT 1111
z
L'
L
b
V'
a
a
a
W
v
v
v
v
I
p
p
R
X
B
R'
0
0
m
m
PAYLOAD BYTE 2 1115
PAYLOAD BYTE 1 1117
PAYLOAD BYTE 0 1119

FIG. 11

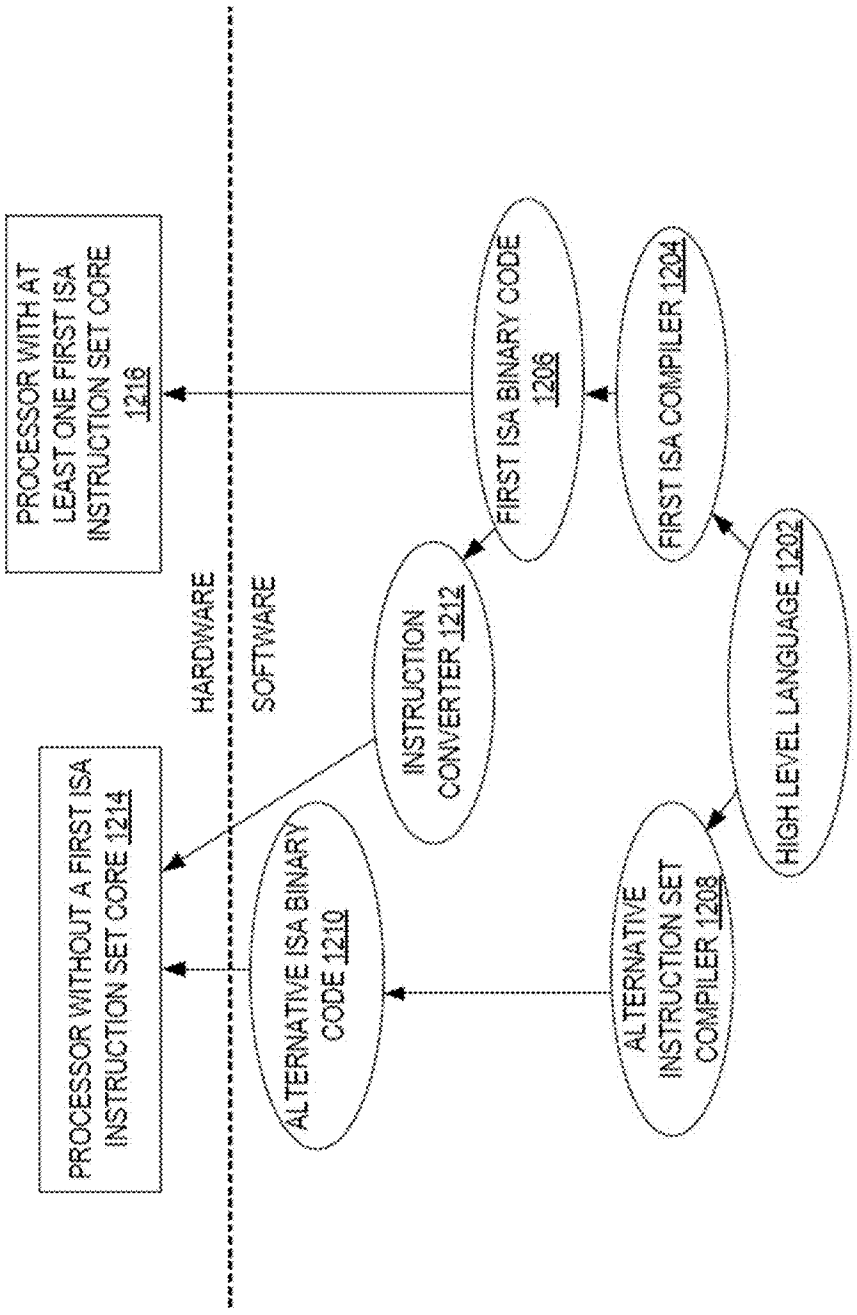

PROCESSOR WITH AT LEAST ONE FIRST ISA INSTRUCTION SET CORE 1216
HARDWARE
SOFTWARE
PROCESSOR WITHOUT A FIRST ISA INSTRUCTION SET CORE 1214
FIRST ISA BINARY CODE 1206
FIRST ISA COMPILER 1204
INSTRUCTION CONVERTER 1212
HIGH LEVEL LANGUAGE 1202
ALTERNATIVE ISA BINARY CODE 1210
ALTERNATIVE INSTRUCTION SET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

FIG. 12

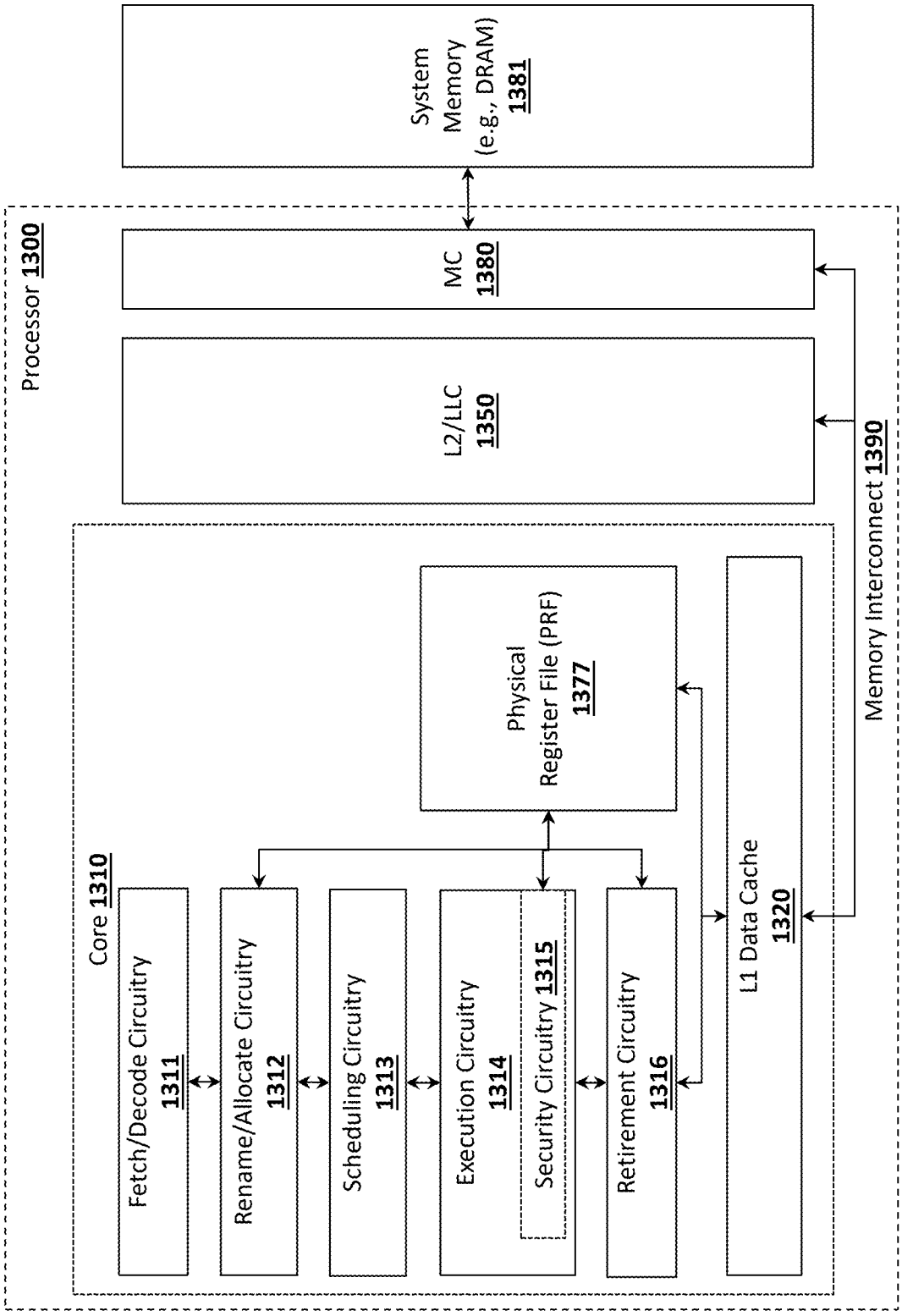

System Memory (e.g., DRAM) 1381
Processor 1300
MC 1380
L2/LLC 1350
Core 1310
Physical Register File (PRF) 1377
Fetch/Decode Circuitry 1311
Rename/Allocate Circuitry 1312
Scheduling Circuitry 1313
Execution Circuitry 1314
Security Circuitry 1315
Retirement Circuitry 1316
L1 Data Cache 1320
Memory Interconnect 1390

FIG. 13

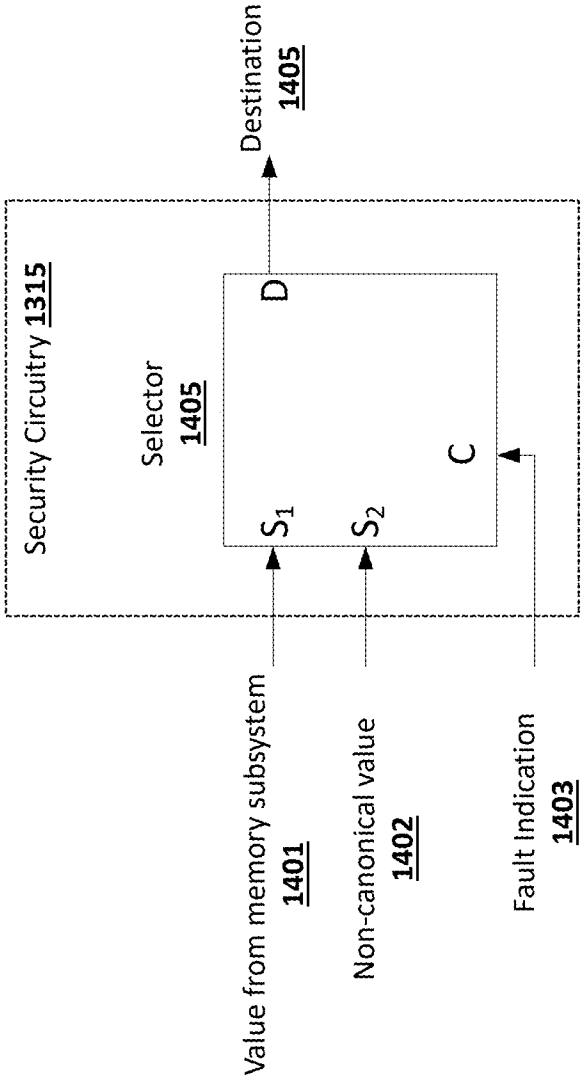

Security Circuitry 1315
Selector
1405
D
C
S1
S2
Destination
1405
Value from memory subsystem
1401
Non-canonical value
1402
Fault Indication
1403

FIG. 14

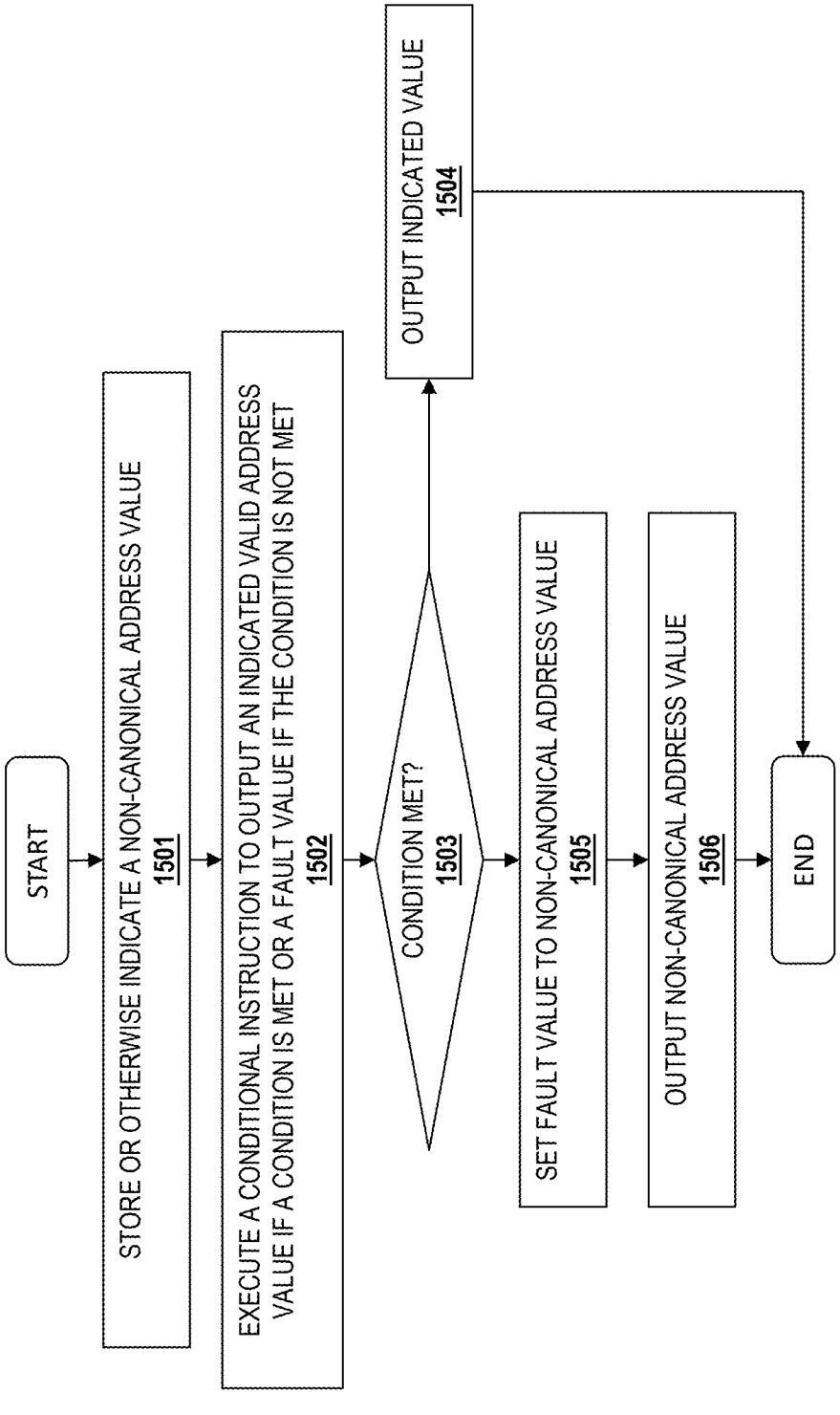

START
STORE OR OTHERWISE INDICATE A NON-CANONICAL ADDRESS VALUE
1501
EXECUTE A CONDITIONAL INSTRUCTION TO OUTPUT AN INDICATED VALID ADDRESS VALUE IF A CONDITION IS MET OR A FAULT VALUE IF THE CONDITION IS NOT MET
1502
CONDITION MET?
1503
OUTPUT INDICATED VALUE
1504
SET FAULT VALUE TO NON-CANONICAL ADDRESS VALUE
1505
OUTPUT NON-CANONICAL ADDRESS VALUE
1506
END

FIG. 15

APPARATUS AND METHOD TO INJECT NON-CANONICAL ADDRESSES INTO FAULTING INSTRUCTION OUTPUTS TO MITIGATE TRANSIENT EXECUTION VULNERABILITIES

BACKGROUND

Field of the Invention

[0001] This invention relates generally to the field of computer processors. More particularly, the invention relates to an apparatus and method to inject non-canonical addresses into faulting instruction outputs to mitigate transient execution vulnerabilities.

Description of the Related Art

[0002] Various forms of vulnerabilities have been detected and exploited on modern microprocessor architectures. Specific types of vulnerabilities that affect modern microprocessors include Meltdown, Foreshadow, Microarchitectural Data Sampling (MDS), and Load Value Injection (LVI).

[0003] Meltdown is a transient execution vulnerability which allows a rogue process to read all memory without authorization. Foreshadow, also known as L1 Terminal Fault, is similar to Meltdown, but is more effective at bypassing security measures. Microarchitectural Data Sampling (MDS), which affects many modern processors, allows a malicious program to read sampling data from counters and other registers. Load Value Injection (LVI) builds on the concepts of Meltdown and MDS by injecting data values into a victim program, and is more difficult to mitigate than previous vulnerabilities.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0005] FIG. **1** illustrates an example computer system architecture.

[0006] FIG. **2** illustrates a processor comprising a plurality of cores.

[0007] FIG. **3**A illustrates a plurality of stages of a processing pipeline.

[0008] FIG. **3**B illustrates details of one embodiment of a core.

[0009] FIG. **4** illustrates execution circuitry in accordance with one embodiment.

[0010] FIG. **5** illustrates one embodiment of a register architecture.

[0011] FIG. **6** illustrates one example of an instruction format.

[0012] FIG. **7** illustrates addressing techniques in accordance with one embodiment.

[0013] FIG. **8** illustrates one embodiment of an instruction prefix.

[0014] FIGS. **9**A-D illustrate embodiments of how the R, X, and B fields of the prefix are used.

[0015] FIGS. **10**A-B illustrate examples of a second instruction prefix.

[0016] FIG. **11** illustrates payload bytes of one embodiment of an instruction prefix.

[0017] FIG. **12** illustrates instruction conversion and binary translation implementations.

[0018] FIG. **13** illustrates a processor architecture in accordance with some embodiments of the invention.

[0019] FIG. **14** illustrates one example of a selector circuit in accordance with some embodiments.

[0020] FIG. **15** illustrates a method in accordance with embodiments of the invention.

DETAILED DESCRIPTION

[0021] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments of the invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

Exemplary Computer Architectures

[0022] 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.

[0023] 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.

[0024] 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.

[0025] 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.

[0026] 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.

[0027] 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).

[0028] 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.

[0029] 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

[0030] 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.

[0031] 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**.

[0032] 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 (in-

cluding 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.

[0033] 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).

[0034] 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.

[0035] 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

[0036] 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.

[0037] 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.

[0038] 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**.

[0039] 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.

[0040] 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**.

[0041] 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.

[0042] 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.

[0043] 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.

[0044] 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

[0045] 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

[0046] 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.

[0047] 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).

[0048] 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.

[0049] 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.

[0050] 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.

[0051] 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.

[0052] 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.

[0053] 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.

[0054] 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.

[0055] 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

[0056] 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

[0057] 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.

[0058] 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.

[0059] 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.

[0060] 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.

[0061] 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**.

[0062] 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.

[0063] 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.

[0064] 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.

[0065] 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).

[0066] 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**.

[0067] 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.

[0068] 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).

[0069] 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.

[0070] 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.

[0071] 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.

[0072] 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.

[0073] Bit position 1 (X) X bit may modify the SIB byte index field **754**.

[0074] 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**).

[0075] 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 **704** 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 **704** 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**.

[0076] 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 as A=B+C.

[0077] 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.

[0078] 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 (1s complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in 1s 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.

[0079] 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.

[0080] 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.

[0081] 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.

[0082] 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.

[0083] 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 (1s complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in 1s 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.

[0084] 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.

[0085] 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.

[0086] 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.

[0087] 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.

[0088] 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).

[0089] 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.).

[0090] 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).

[0091] 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 (1s complement) form and valid for instructions with 2 or more source operands; 2) encode the destination register operand, specified in 1s 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.

[0092] 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.

[0093] 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.

[0094] 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).

[0095] 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

[0096] 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.

[0097] 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.

[0098] 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.

[0099] 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.

[0100] 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.

[0101] 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.

[0102] 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.

[0103] 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 certain implementations. 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**.

[0104] 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**.

Apparatus and Method to Inject Non-Canonical Addresses into Faulting Instruction Outputs to Mitigate Transient Execution Vulnerabilities

[0105] Modern processors mitigate microarchitectural transient execution vulnerabilities such as Meltdown, Foreshadow, Microarchitectural Data Sampling (MDS), and Load Value Injection (LVI) by preventing operations that depend on a faulting load from executing. Failure to prevent execution of these operations can potentially cause sensitive data to become exposed through a microarchitectural covert channel. Operations on some microarchitectures can impose a performance penalty when they prevent dependent operations from waking early. One example is the (Conditional Faulting CMOVcc) CFCMOVCC instruction, a new EVEX map 4 instruction introduced in the Advanced Performance Extensions (APX). When the condition code evaluates to false, a CFCMOVcc instruction suppresses all memory faults and the debug exception (#DB):

EVEX.ND	EVEX.NF	Instruction Forms	Instruction Semantics

0	0	CFCMOVcc reg, r/m	IF (flags satisfies cc):
			reg := r/m
			ELSE:
			// memory faults are suppressed
			reg := 0

[0106] The CFCMOVCC instruction ignores a memory fault if the condition code is not satisfied and instead writes a constant value such as 0 to the destination register. Architecturally, the decision to write a constant value or to write from the source operand is dictated by the condition code. Microarchitecturally, the condition code can be speculatively bypassed. For example, if the processor detects a fault before the condition code is ready, then the processor may speculate that the condition code will be unsatisfied, and therefore that the constant value should be written to the destination. This allows dependent operations to wake sooner. Similarly, if the load succeeds, the processor can speculate that the loaded value should be written to the destination.

[0107] Optimizations of this type can introduce risks, especially in security-critical environments such as a secure enclave, which is a contiguous region of memory within a process protected by the processor (e.g., Intel SGX enclaves). Code that runs inside the enclave does not necessarily trust code that runs outside of the enclave. For example, code running outside of the enclave may store malicious data at address 0 (assuming that address 0 is not mapped into the enclave). Then, for example, the enclave may execute the following sequence:

[0108] CMP 0, QWORD PTR [RSI]

[0109] CFCMOVE R8, QWORD PTR [RDI]

[0110] CFCMOVE R9, QWORD PTR [R8]

[0111] Additionally, suppose that the malicious system software had un-mapped the page referenced by RDI before running the enclave, and hence the second instruction faults. Furthermore, suppose that the memory referenced by RSI contains the value 0 but isn't cached, so the CMP executes slowly. In this scenario, the optimization described above triggers on the first CFCMOVE when the fault is detected and the processor (falsely) speculates that the "equals" condition won't be satisfied. In this scenario, the second CFCMOVE loads from address 0, which succeeds, and therefore (correctly) speculates that the condition code will be satisfied and writes the malicious contents of address 0 to R9. This injection of malicious data into an enclave during speculative execution can become a launching point for a family of exploits known as Zero Value Injection (ZVI).

[0112] Another example is a vulnerability known as Register File Data Sampling (RFDS). On affected processors, a divide instruction may cause a divide error when, for instance, the divisor is 0. When the divide instruction takes a divide error, the processor may cancel the write to the register file. Therefore, operations that depend on the divide instruction may read stale data from the register file, potentially sensitive data belonging to another context, or malicious data injected by an attacker.

[0113] Embodiments of the invention prevent faulting/assisting instructions such as loads from yielding values that could be used by a malicious adversary to infer sensitive data, such as cryptographic keys. These embodiments ensure security for new performance-enhancing features (e.g., Advanced Performance Extensions (APX)) so that the features can be used safely in secure enclaves and other security-critical environments.

[0114] Embodiments of the invention include or inject operations that write a non-canonical address to the destination on a fault condition. As used herein, an address is non-canonical if it invalid in accordance with the microarchitecture. In 48-bit paging mode (used, for example, in x86-64 processors), valid low-canonical (user) addresses range from 0x0000000000000000 to 0x00007FFFFFFFFFFF and valid high-canonical (kernel) addresses range from 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF; all other addresses are considered non-canonical.

[0115] A specific example of a non-canonical address is utilized in the following CFCMOVCC variant, where CFCMOVcc semantics are augmented to write a non-canonical value to the register destination on a fault:

EVEX.ND	EVEX.NF	Instruction Forms	Instruction Semantics

TDB	TBD	CFCMOVcc reg, r/m	IF (flags satisfies cc):
			reg := r/m
			ELSE:
			// memory faults are suppressed
			reg := 0x5555555555555555

[0116] Embodiments of the invention leverage the observation that for zero-value injection (ZVI) and similar types of vulnerabilities, injection of a constant value X can only be used to craft a successful exploit if:

[0117] (i) X is a valid address that contains malicious data, or

[0118] (ii) The constant value can be manipulated using manipulations in the victim program to construct a valid address that contains malicious data.

[0119] In other words, the attacker might be able to utilize addition, subtraction, or multiplication operations in the victim program to set the upper 16 bits of X to either all 0's or all 1's (or to set the upper 7 bits to all 0's or 1's in 57-bit paging mode). The constant value in this CFCMOVCC example, 0x5555555555555555, is non-canonical on x86-64 architectures in both 48-bit and 57-bit paging modes, thus obviating exploit condition (i) above.

[0120] This value also addresses condition (ii) because an attacker would need to find a manipulation in the victim program that takes the output of a faulting operation and then does something that makes 0x5555555555555555 canonical, which would require the upper 16 bits (0b0101010101010101) to become either 0b0000000000000000 or 0b1111111111111111 (and likewise for the upper 7 bits in 57-bit paging mode). For example, consider addition, subtraction, and multiplication operations, which are the operations most commonly used to manipulate memory addresses in computer programs:

[0121] 0x5555555555555555 remains non-canonical when shifted to the left by up to 63 bit positions (i.e., when multiplied by up to 2^63).

[0122] 0x5555555555555555 remains non-canonical when added to (or subtracted from) a canonical address.

[0123] Embodiments of this invention may also inject a truncated non-canonical address into destination registers that are smaller than the width of an address. For example, one embodiment may inject the value 0x55555555 into faulting 32-bit loads. Hence, if the result of a faulting 32-bit load is concatenated (as the high-order bits) to another 32-bit value to form a 64-bit address, the result will be non-canonical.

[0124] Some embodiments of this invention may be implemented in software (at least in part) to make use of

existing instructions to achieve similar security properties. For example, consider the following CFCMOVCC encoding from the APX specification:

EVEX.ND	EVEX.NF	Instruction Forms	Instruction Semantics

1	1	CFCMOVcc ndd, reg, r/m	IF (flags satisfies cc):
			ndd := r/m
			ELSE:
			// memory faults are suppressed
			ndd := reg

[0125] A software developer (or a compiler, runtime, JIT, etc.) may choose to keep a non-canonical value such as 0x5555555555555555 in a register to allow CFCMOVCC to be used safely, as follows:

[0126] MOV R15, 0x5555555555555555

[0127] . . .

[0128] CMP 0, QWORD PTR [RSI]

[0129] CFCMOVE R8, R15, QWORD PTR [RDI]

[0130] CFCMOVE R9, R15, QWORD PTR [R8]

This code ensures that R8 and R9 will receive non-canonical values if either CFCMOVE faults, respectively.

[0131] FIG. **13** illustrates an example processor **1300** on which the embodiments described herein may be implemented. The circuitry of a single core **1310** is shown for simplicity, although the processor **1300** may have a plurality of cores with the same or similar architectures. The illustrated core **1310** includes fetch/decode circuitry **1311** for fetching instructions and decoding the instructions into microoperations. Rename/allocate circuitry **1313** comprises register renaming circuitry for performing register renaming (e.g., within a physical register file (PRF) **1377**) and allocation circuitry for allocating execution resources to execute the instructions. Scheduling circuitry **1313**, such as a reservation station (RS), schedules instructions for execution on specific circuit blocks within the execution circuitry **1314**. Security circuitry **1315** implements one or more of the techniques described herein, such as injecting or otherwise utilizing non-canonical address values to mitigate zero-value injection (ZVI) and similar types of microarchitectural vulnerabilities. Retirement circuitry **1316** retires the executed instructions (assuming no conflicts), committing the results to the visible architectural state and potentially writing back the results to the cache/memory subsystem.

[0132] The cache/memory subsystem of the processor **1300** comprises a level 1 (L1) data cache unit **1320** integral to the core **1310**, a level 2 (L2) and/or last-level cache (LLC) **1350**, and one or more memory controllers **1380** to couple the various cache levels to a system memory **1381** (e.g., a DRAM). Although illustrated as separate components, the various cache levels may operate together, communicating over a memory interconnect **1390** to perform cache management operations such as moving cache lines between cache levels and accessing cache lines from memory **1381** via the memory controller **1380**.

[0133] FIG. **14** illustrates one embodiment of the security circuitry **1315** including a selector **1405** (e.g., a multiplexer) for selecting non-canonical values **1402** as described herein. For example, in response to a conditional move or load instruction, the value from the memory subsystem **1401** is moved/loaded to the destination **1405** (e.g., a specified register in the physical register file **1377**) if the specified condition is met, meaning that the fault indication **1403** indicates no fault. If the condition is not met, the fault condition **1403** is set, thereby causing the multiplexer to select the non-canonical address value **1402** to be moved/loaded to the destination **1405**. As described herein, the non-canonical address cannot itself be used to breach the security circuitry **1315** and cannot be converted to a canonical address via mathematical manipulation (e.g., multiplication or addition operations).

[0134] Embodiments of this invention may also include different microarchitectural techniques to yield a (temporary) non-canonical value from an instruction that faults. For example, an unconditional load from memory (such as memory-to-register MOV) can incorporate the multiplexer **1405**. If the load instruction's fault indication **1403** is high (meaning that the load faulted), then the MUX selects S_2, the non-canonical address value **1402**, to write to the output destination **1405** that is written back to the register file or passed to dependent operations; otherwise, the multiplexer **1405** passes the data S_1 provided by the memory subsystem to the destination **1405**. Note that if the fault indication is high then dependent operations may still execute transiently with the non-canonical value, but they will be invalidated/flushed due to the non-canonical value **1402** when the processor handles the fault.

[0135] The implementation shown in FIG. **14** may be applied to single micro-operations (uops decoded from an instruction), and can also be used for more complex instructions comprising a plurality of uops. Returning to the earlier example, CFCMOVcc is logically:

[0136] (1) A fault-suppressing load that returns a fault indication, and

[0137] (2) A conditional MOV that consumes the live-ins of the CFCMOVcc, as well as the load data and fault indication returned by (1).

[0138] Under normal, non-faulting circumstances, the (1) portion of the CFCMOVcc would return actual load data as gathered via the memory execution units of the CPU/core (load machinery, load buffer, and cache):

[0139] (actual_load_data, fault_indication=false)

[0140] However, in a faulting circumstance where the address cannot be read without faulting, the (1) portion of the CFCMOVcc must drive a value to the (2) portion of the CFCMOVcc operation. Since actual load data cannot be retrieved, the microarchitecture must pass synthetic data with the fault indication:

[0141] (synthetic_load_value, fault_indication=true)

[0142] A synthetic value of 0 has the risks described above, since the (2) portion of CFCMOVcc and any speculatively scheduled/executed dependents may consume the value 0. However, a synthetic value that has the non-canonical properties described above (e.g., 0x5555555555555555), as passed to the (2) portion of CFCMOVcc and any speculatively scheduled/executed dependents, can mitigate ZVI exploits without requiring the non-canonical value injection to be explicitly defined by the processor's ISA.

[0143] A method in accordance with embodiments of the invention is illustrated in FIG. **15**. The method may be implemented on the various architectures described herein, but is not limited to any particular processor or system architecture.

[0144] At **1501**, a non-canonical address value is stored or otherwise indicated (e.g., in a specified register or memory

location). At **1502**, a conditional instruction is executed to output an indicated valid address value if a condition is met or a fault value if the condition is not met. Any type of condition may be specified such as equal to, less than, greater than, greater than or equal to, less than or equal to, etc.

[0145] If the condition is met, determined at **1503**, then the indicated valid address value is output at **1504**. For example, if the instruction is a load instruction, then the valid address value may be read from the cache/memory subsystem and stored in a destination register. If the instruction is a move instruction, then the indicated address value may be moved from a source register to a destination register.

[0146] If the condition is not met, then at **1505**, the fault value is set to the non-canonical address value (indicated at **1501**) and the non-canonical address value is output at **1506**. Thus, instead of storing the valid address value at **1504**, a non-canonical address value, which is invalid based on the processor microarchitecture, is stored to the destination, thereby preventing further operations using the non-canonical address value (such as LVI or other malicious operations).

[0147] Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

EXAMPLES

[0148] The following are example implementations of different embodiments of the invention.

[0149] Example 1. A processor, comprising: decode circuitry to decode a sequence of instructions, including a conditional instruction; execution circuitry to execute the conditional instruction, the execution circuitry comprising security circuitry to mitigate injection-based exploits, the security circuitry to perform operations comprising: outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and when the condition associated with the conditional instruction is determined to be false: setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value; and outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.

[0150] Example 2. The processor of example 1, wherein the non-canonical address value comprises an invalid address value based on a microarchitecture of the processor.

[0151] Example 3. The processor of examples 1 or 2, wherein the non-canonical address value comprises a first non-canonical address value, wherein adding a canonical value to the first non-canonical address value results in a second non-canonical address value and wherein multiplying the first non-canonical address value by a canonical value results in a third non-canonical address value.

[0152] Example 4. The processor of any of examples 1-3, further comprising: a register to store the non-canonical address value prior to execution of the conditional instruction, the register to be used as a source register for the conditional instruction when the condition associated with the conditional instruction is determined to be false.

[0153] Example 5. The processor of any of examples 1-4, wherein the execution circuitry comprises: selector circuitry to select a first value comprising the valid address value or a second value comprising the non-canonical address value or truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be true or false, respectively.

[0154] Example 6. The processor of any of examples 1-5, wherein the conditional instruction comprises a conditional load instruction to load the valid address value from a cache-memory subsystem or a conditional move instruction to move the valid address value from a register when the condition associated with the conditional instruction is determined to be true.

[0155] Example 7. The processor of any of examples 1-6, wherein the condition associated with the conditional instruction comprises a result of a comparison operation between a first source value and a second source value.

[0156] Example 8. The processor of any of examples 1-7, wherein the comparison operation between the first source value and the second source value comprises one of: greater than, less than, equal to, greater than or equal to, and less than or equal to.

[0157] Example 9. A method, comprising: decoding a sequence of instructions by a decoder of a processor, the sequence of instructions including a conditional instruction; executing the conditional instruction, wherein executing includes: outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be false, and outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.

[0158] Example 10. The method of example 9, wherein the non-canonical address value comprises an invalid address value based on a microarchitecture of the processor.

[0159] Example 11. The method of examples 9 or 10, wherein the non-canonical address value comprises a first non-canonical address value, wherein adding a canonical value to the first non-canonical address value results in a second non-canonical address value and wherein multiplying the first non-canonical address value by a canonical value results in a third non-canonical address value.

[0160] Example 12. The method of any of examples 9-11, further comprising: storing the non-canonical address value in a register prior to execution of the conditional instruction, the register to be used as a source register for the conditional instruction when the condition associated with the conditional instruction is determined to be false.

[0161] Example 13. The method of any of examples 9-12, further comprising: selecting a first value comprising the valid address value or a second value comprising the non-canonical address value or truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be true or false, respectively.

[0162] Example 14. The method of any of examples 9-13, wherein the conditional instruction comprises a conditional load instruction to load the valid address value from a cache-memory subsystem or a conditional move instruction to move the valid address value from a register when the condition associated with the conditional instruction is determined to be true.

[0163] Example 15. The method of any of examples 9-14, wherein the condition associated with the conditional instruction comprises a result of a comparison operation between a first source value and a second source value.

[0164] Example 16. The method of any of examples 9-15, wherein the comparison operation between the first source value and the second source value comprises one of: greater than, less than, equal to, greater than or equal to, and less than or equal to.

[0165] Example 17. A machine-readable medium having program code stored thereon which, when executed by a machine, causes the machine to perform additional operations, comprising: decoding a sequence of instructions by a decoder of a processor, the sequence of instructions including a conditional instruction; executing the conditional instruction, wherein executing includes: outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be false, and outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.

[0166] Example 18. The machine-readable medium of example 17, wherein the non-canonical address value comprises an invalid address value based on a microarchitecture of the processor.

[0167] Example 19. The machine-readable medium of examples 17 or 18, wherein the non-canonical address value comprises a first non-canonical address value, wherein adding a canonical value to the first non-canonical address value results in a second non-canonical address value and wherein multiplying the first non-canonical address value by a canonical value results in a third non-canonical address value.

[0168] Example 20. The machine-readable medium of any of examples 17-19, further comprising program code to cause the machine to perform the operations of: storing the non-canonical address value in a register prior to execution of the conditional instruction, the register to be used as a source register for the conditional instruction when the condition associated with the conditional instruction is determined to be false.

[0169] As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals-such as carrier waves, infrared signals, digital signals, etc.).

[0170] In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware.

[0171] 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.

What is claimed is:

1. A processor, comprising:

decode circuitry to decode a sequence of instructions, including a conditional instruction;

execution circuitry to execute the conditional instruction, the execution circuitry comprising security circuitry to perform operations comprising:

outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and

when the condition associated with the conditional instruction is determined to be false:

setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value; and

outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.

2. The processor of claim **1**, wherein the non-canonical address value comprises an invalid address value based on a microarchitecture of the processor.

3. The processor of claim **2**, wherein the non-canonical address value comprises a first non-canonical address value, wherein adding a canonical value to the first non-canonical address value results in a second non-canonical address

value and wherein multiplying the first non-canonical address value by a canonical value results in a third non-canonical address value.

4. The processor of claim **1**, further comprising:

a register to store the non-canonical address value prior to execution of the conditional instruction, the register to be used as a source register for the conditional instruction when the condition associated with the conditional instruction is determined to be false.

5. The processor of claim **4**, wherein the execution circuitry comprises:

selector circuitry to select a first value comprising the valid address value or a second value comprising the non-canonical address value or truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be true or false, respectively.

6. The processor of claim **1**, wherein the conditional instruction comprises a conditional load instruction to load the valid address value from a cache-memory subsystem or a conditional move instruction to move the valid address value from a register when the condition associated with the conditional instruction is determined to be true.

7. The processor of claim **6**, wherein the condition associated with the conditional instruction comprises a result of a comparison operation between a first source value and a second source value.

8. The processor of claim **7**, wherein the comparison operation between the first source value and the second source value comprises one of: greater than, less than, equal to, greater than or equal to, and less than or equal to.

9. A method, comprising:

decoding a sequence of instructions by a decoder of a processor, the sequence of instructions including a conditional instruction;

executing the conditional instruction, wherein executing includes:

outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and

setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be false, and outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.

10. The method of claim **9**, wherein the non-canonical address value comprises an invalid address value based on a microarchitecture of the processor.

11. The method of claim **10**, wherein the non-canonical address value comprises a first non-canonical address value, wherein adding a canonical value to the first non-canonical address value results in a second non-canonical address value and wherein multiplying the first non-canonical address value by a canonical value results in a third non-canonical address value.

12. The method of claim **9**, further comprising:

storing the non-canonical address value in a register prior to execution of the conditional instruction, the register to be used as a source register for the conditional instruction when the condition associated with the conditional instruction is determined to be false.

13. The method of claim **12**, further comprising:

selecting a first value comprising the valid address value or a second value comprising the non-canonical address value or truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be true or false, respectively.

14. The method of claim **9**, wherein the conditional instruction comprises a conditional load instruction to load the valid address value from a cache-memory subsystem or a conditional move instruction to move the valid address value from a register when the condition associated with the conditional instruction is determined to be true.

15. The method of claim **14**, wherein the condition associated with the conditional instruction comprises a result of a comparison operation between a first source value and a second source value.

16. The method of claim **15**, wherein the comparison operation between the first source value and the second source value comprises one of: greater than, less than, equal to, greater than or equal to, and less than or equal to.

17. A machine-readable medium having program code stored thereon which, when executed by a machine, causes the machine to perform additional operations, comprising:

decoding a sequence of instructions by a decoder of a processor, the sequence of instructions including a conditional instruction;

executing the conditional instruction, wherein executing includes:

outputting a valid address value indicated by the conditional instruction to a destination when a condition associated with the conditional instruction is determined to be true; and

setting an output fault value associated with the conditional instruction to a non-canonical address value or a truncated portion of the non-canonical address value when the condition associated with the conditional instruction is determined to be false, and outputting the non-canonical address value or truncated portion of the non-canonical address value to the destination.

18. The machine-readable medium of claim **17**, wherein the non-canonical address value comprises an invalid address value based on a microarchitecture of the processor.

19. The machine-readable medium of claim **18**, wherein the non-canonical address value comprises a first non-canonical address value, wherein adding a canonical value to the first non-canonical address value results in a second non-canonical address value and wherein multiplying the first non-canonical address value by a canonical value results in a third non-canonical address value.

20. The machine-readable medium of claim **17**, further comprising program code to cause the machine to perform the operations of:

storing the non-canonical address value in a register prior to execution of the conditional instruction, the register to be used as a source register for the conditional instruction when the condition associated with the conditional instruction is determined to be false.

* * * * *