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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
29 June 2017 (29.06.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/112315 A1

(51) International Patent Classification:
G06F 21/54 (2013.01) *G06F 21/50* (2013.01)

(21) International Application Number:
PCT/US2016/063665

(22) International Filing Date:
23 November 2016 (23.11.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
14/757,964 23 December 2015 (23.12.2015) US

(71) Applicant: **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95054 (US).

(72) Inventors: **SHANBHOGUE, Vedvyas**; 4912 Pyreness Pass, Austin, TX 78738 (US). **SAHITA, Ravi, L.**; 5314 NW 131st Ave, Portland, OR 97229 (US). **GUPTA, Deepak, K.**; 3063 NW Overlook Drive Apt # 622, Hillsboro, OR 97124 (US).

(74) Agents: **PORTNOVA, Marina** et al.; Lowenstein Sandler LLP, 65 Livingston Avenue, Roseland, NJ 07068 (US).

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

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

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

— *with international search report (Art. 21(3))*

(54) Title: NON-TRACKED CONTROL TRANSFERS WITHIN CONTROL TRANSFER ENFORCEMENT

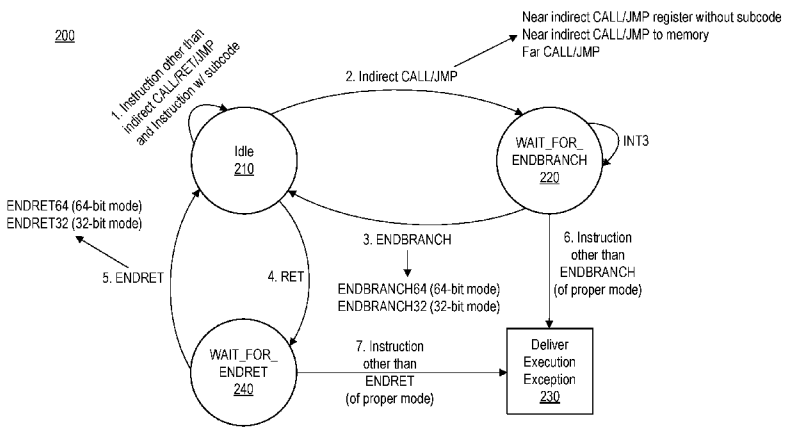

FIG. 2

(57) Abstract: A processor includes an execution unit and a processing logic operatively coupled to the execution unit, the processing logic to: enter a first execution state; responsive to executing a control transfer instruction having a pre-defined subcode, remain in the first execution state; responsive to executing a control transfer instruction not having the pre-defined subcode, transition into a second execution state; and responsive to determining, in the second execution state, that a next instruction to be executed differs from an ENDBRANCH instruction, raise an execution exception.

WO 2017/112315 A1

NON-TRACKED CONTROL TRANSFERS WITHIN CONTROL TRANSFER ENFORCEMENT

Background

[0001] Return-oriented programming (ROP) is a computer security exploit technique in which an attacker uses software control of a stack to execute an attacker-chosen sequence of machine instructions. These clusters of instructions typically end with a programmer-intended or unintended return (RET) instruction within existing program code. The intended or unintended RET instruction transfers execution to the attacker-chosen return address on the stack and allows the attacker to retain execution control through the program code, and direct execution to the next set of chosen sequence of instructions to achieve the attacker's intent. The clusters of attacker-chosen instruction sequences are referred to as gadgets.

[0002] Often the executed gadget includes only several assembler instructions followed by a RET instruction that can already perform a well-defined attack operation. By chaining together a set of these gadgets such that the RET instructions from one gadget lands into the next gadget and so on, the malware writer is able to execute a complex algorithm without injecting any code into the program. Some of these instruction sequences ending in a RET can be found in functions compiled into the program or libraries.

[0003] Thus the ROP technique involves delivering a payload having a set of chained list of pointers to gadgets and parameters to a data memory of a program using vulnerabilities like stack buffer overflows. The exploit also overwrites the return address of the vulnerable function that was used to perform the stack buffer overflow to point to the first gadget in the sequence. When this vulnerable function executes a RET instruction, control transfers to the first gadget instead of the function caller. This gadget can then consume one or more data elements from the stack payload. Using this exploit type, the malware writer is able to change the control flow of the program by causing a control transfer to a non-programmer intended location in the program (e.g., to the middle of an instruction).

[0004] A ROP attack technique can use various characteristics of an x86 instruction set architecture (ISA): variable length and unaligned instruction encoding; large and dense ISA encoding; a stack holding control and data information; and a single byte opcode RET instruction. Current techniques to defend against such attacks can be ineffective and have various shortcomings.

Brief Description of the Drawings

[0005] Figure 1 is a block diagram of a portion of a processor in accordance with an embodiment of the present disclosure.

[0006] Figure 2 is a block diagram of a state machine in accordance with an embodiment of the present disclosure.

[0007] Figure 3A is a flow chart of a method of a compiler deciding whether to add a pre-defined subcode to control transfer instructions of a high-level programming language.

[0008] Figure 3B is a flow chart of a method for non-tracked control transfers and mode-specific control flow termination according to an embodiment of the present disclosure.

[0009] Figure 4 is a flow chart of another method for mode-specific control flow termination according to another embodiment of the present disclosure.

[0010] Figure 5A is a flow diagram of a method of control transfer management in accordance with an embodiment of the present disclosure.

[0011] Figure 5B is a flow diagram of another method of control transfer management in accordance with an embodiment of the present disclosure.

[0012] Figure 6 is a block diagram of a configuration register in accordance with an embodiment of the present disclosure.

[0013] Figure 7A is a block diagram of a call stack frame for code execution in accordance with an embodiment of the present disclosure.

[0014] Figure 7B is a block diagram of control transfer termination logic with legacy interworking in accordance with an embodiment of the present disclosure.

[0015] Figure 8 is a block diagram of a processor core in accordance with one embodiment of the present disclosure.

[0016] Figure 9A is a block diagram illustrating a micro-architecture for a processor core that can be incorporated into the processor of Figure 1.

[0017] Figure 9B is a block diagram illustrating an in-order pipeline and a register renaming stage, out-of-order issue/execution pipeline implemented by the processor core of Figure 9A according to some embodiments of the disclosure.

[0018] Figure 10 illustrates a block diagram of the micro-architecture for a processor that, in one embodiment, can represent portions of the processor of Figure 1.

[0019] Figure 11 is a block diagram of a multi-processor system according to one implementation.

[0020] Figure 12 is a block diagram of a multi-processor system according to another implementation.

[0021] Figure 13 is a block diagram of a system-on-a-chip according to one implementation.

[0022] Figure 14 illustrates another implementation of a block diagram for a computing system.

[0023] Figure 15 illustrates another implementation of a block diagram for a computing system.

Description of Embodiments

[0024] Embodiments provide a set of instruction set architecture (ISA) instructions that enable a processor to determine whether a control transfer instruction is directed to an appropriate target, and if not to take action to prevent instructions beyond the control transfer instruction from being committed to the architectural state of the processor. In this way, at least certain unintended control transfers within a program can be eliminated, constraining the number of gadgets that a malware writer can use.

[0025] More specifically, embodiments provide a set of control transfer termination (CTT) instructions to allow software to specify valid control transfer terminating points in a program such that hardware can enforce control transfers to occur to only programmer-intended locations. These CTT instructions can be opcodes of fixed length that are handled differently than ISA instructions in that the CTT instructions perform no operation other than to signal to the processor that a proper control transfer termination is occurring. In one embodiment, the CTT instructions are mode-specific (e.g., 32-bit or 64-bit), multi-byte opcodes executable as a no operation. Accordingly, CTT instructions perform this enforcement with minimal performance and energy impacts to the program. The CTT ISA extensions can thus mitigate the execution of unintended gadgets in programs.

[0026] An indirect call or jump refers to an indirect operand. In one type of indirect call or jump, this indirect operand can be an indirect destination address that provides an address of the instruction to which the call or jump is to be performed. In another type of indirect call or jump, this indirect operand can be a processor register that provides an address of the instruction to which the call or jump is to be performed. In some cases, a compiler need not add (or emit) CTT instructions at control transfer terminating points when the compiler generates indirect jumps and calls (and returns from same) and already has code generated to ensure that the targets of that indirect call or jump (or return) are restricted to a small subset of targets known at compile time. For example, for a lookup or a switch-case statement, the compiler knows the number of possible targets and computes the address of the code for the possible targets. This optimization by the compiler to not emit a CTT instruction at the control transfer terminating points is generally safe when the indirect jump or call is of the type where the indirect operand is a register. If the indirect operand is a memory location, then the contents of those memory locations can be modified between when the compiler checks the contents and when the indirect call or jump instruction uses the content of that memory location to perform the indirect call or jump.

[0027] In either of these cases, not requiring a CTT instruction following a control transfer instruction reduces the size of the binary generated by a program (due to eliminated CTT instructions); and, jump or call targets become not-valid targets for indirect jumps or calls from other locations due to lacking the CTT instruction following the control transfer instruction, thus increasing security of the program. In one embodiment, a subcode such as a prefix or suffix can be added to a control transfer instruction (such as an indirect call or jump) indicating that the indirect call or jump need not be tracked by a tracker state machine, which can remain idle for such instructions. As will be further explained, the tracker state machine tracks states for control transfer instructions such that a CTT instruction is required after a control transfer instruction, except where the subcode is detected, to avoid issuing an execution exception.

[0028] In an alternative embodiment, the subcode can be added to all control transfer instructions except for those that need not be tracked. In other words, the lack of a subcode can signal to CTT processing logic that those control transfer instructions are not tracked by the tracker state machine, in the alternative embodiment.

[0029] In another embodiment, a new indirect control transfer instruction may be defined that need not be tracked.

[0030] As more computer systems are used in Internet, text, and multimedia applications, additional processor support has been introduced over time to execute these applications. In one embodiment, an instruction set can be associated with one or more computer architectures, including data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). In one embodiment, the ISA can be implemented by one or more micro-architectures, which include processor logic and circuits used to implement one or more instruction sets. Accordingly, processors with different micro-architectures can share at least a portion of a common instruction set.

[0031] In one embodiment, an instruction can include one or more instruction formats. Such instruction format can indicate various fields (number of bits, location of bits, etc.) to specify, among other things, the operation to be performed and the operand(s) on which that operation is to be performed. Some instruction formats can be further broken defined by instruction templates (or sub formats). For example, the instruction templates of a given instruction format can be defined to have different subsets of the instruction format's fields and/or defined to have a given field interpreted differently. In one embodiment, an instruction is expressed using an instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and specifies or indicates the operation and the operands upon which the operation will operate.

[0032] A first CTT instruction, referred to herein as an ENDBRANCH instruction, is used to identify valid locations in a program to which a control transfer can be validly performed using

an indirect CALL or an indirect jump (JMP) instruction. A second CTT instruction, referred to herein as an ENDRET instruction, is used to identify valid locations in a program where a control transfer can be validly performed using a RET instruction.

[0033] Within instruction set architecture, a program compiled with 64-bit mode instructions when decoded in 32-bit mode could decode to a different set of instructions than when decoded in 64-bit mode. This attribute has been exploited by malware designed to jump into 32-bit binaries from 64-bit mode and vice versa. For example, consider the following byte string in the program: 4c 8b d1 b8 50 00 00 00 0f 05 c3 0f 1f 44 00 00. This byte string, when decoded in 32-bit mode, decodes to following instruction sequence:

```
4c          dec esp
8b d1       mov edx, ecx
B850000000  mov eax, 50h
0f 05       syscall
C3          ret
```

[0034] However, this same byte string when decoded in 64-bit mode decodes to following instruction sequence:

```
4c 8b d1    mov r10, rcx
B850000000  mov eax, 50h
0f 05       syscall
C3          ret
```

[0035] Thus, if the above byte string were in a 32-bit binary and then jumps to a 64-bit binary without switching to compatibility mode, the program would lead to execution of an unintended set of instructions. Likewise, if the above byte string were in a 64-bit binary and then jumps to a 32-bit binary without switching to 64-bit mode, the program would lead to execution of an unintended set of instructions. According to an embodiment, therefore, the disclosed processor employs mode-specific opcodes as CTT instructions to block malware attempts to jump to 32-bit programs without switching to compatibility mode or to jump to 64-bit programs before switching to 64-bit mode.

[0036] Accordingly, the ENDBRANCH instructions can be further broken into two mode-specific CTT instructions, e.g., ENDBRANCH32 and ENDBRANCH64 instructions, to avoid situations where malware can attempt to jump between processing modes to generate gadgets. Similarly, the ENDRET instruction can be further broken into two mode-specific CTT instructions, namely ENDRET32 and ENDRET64 to prevent similar malware behavior when a control transfer instruction accesses memory to determine an address of a target ENDRET

instruction. This extension to different modes can be applied to different modes outside of 32-bit and 64-bit modes as would be apparent to one of ordinary skill.

[0037] In an embodiment, these CTT instructions can be 4-byte opcodes chosen such that they are handled differently than ISA instructions when decoded, e.g., the CTT instructions can be defined as no operation (NOP) in the x86 ISA to allow programs compiled with ENDBRANCH/ENDRET instructions to execute on earlier generation processors.

[0038] Although the scope of the present disclosure is not limited in this regard in an embodiment, these CTT instructions can have a general form that includes a multi-byte opcode. In one such implementation, these CTT instructions can be represented by a four-byte opcode that corresponds to an opcode value not presently existing in the current x86 ISA. Beyond this opcode, there can be no additional encoding for the instruction, since the instruction executes as a no operation within execution logic. As such, there may be no identification of a source operand, destination operand or immediate value to be associated with the instruction.

[0039] Figure 1 is a block diagram of a portion 100 of a processor in accordance with an embodiment of the present disclosure. As shown in Figure 1, the portion 100 of the processor includes various portions of a pipelined processor such as an in-order or out-of-order processor. A compiler 108 can compile instructions into machine-readable code also referred to herein as macro-instructions of one or more programs of a given ISA. As seen, instructions from the compiler 108 are provided to a decode unit 110 that decodes the instructions, e.g., into one or more smaller instruction such as micro-operations (μops).

[0040] The decode unit 110 can include a CTT logic 115 (also simply referred to herein as processing logic) in accordance with an embodiment of the present disclosure. The CTT logic 115 can analyze incoming instructions and determine whether any given instruction is associated with a control transfer. If so, the CTT logic 115 can determine whether the control transfer instruction includes a specific subcode (such as a pre-defined prefix or suffix opcode) indicating that there is no threat of gadget creation. The CTT logic can associate certain state information with one or more μops. This state information indicates a state of a state machine 116 (implemented by the CTT logic 115) that is modified by decoding of at least certain control transfer and control transfer termination instructions. If instead the instruction is not associated with a control transfer, a different state can be associated with the one or more μops.

[0041] When the incoming control transfer instruction lacks the subcode, the CTT logic 115 can transition the state machine 116 from the IDLE state to a given WAIT state. Furthermore, to reflect this WAIT state, a given encoding can be associated with the one or more μops decoded from the incoming control transfer instruction. When a next incoming instruction is a control transfer termination (CTT) instruction that immediately follows the control transfer termination,

then state machine 116 can return to the IDLE state and associate a given encoding with the decoded one or more μops. As will be discussed, when a control transfer instruction is not immediately followed by a CTT instruction of the proper mode, the CTT logic 115 can raise an execution exception or insert a fault μop into the processor pipeline (and the state machine can remain in a wait state).

[0042] When the incoming control transfer instruction contains the subcode, the CTT logic 115 can place or retain the state machine 116 in an IDLE state. Furthermore, due to the lack of an ENDBRANCH instruction (because of the subcode), there is no CTT instruction to be tracked. Furthermore, when the state machine 116 is in an IDLE state and an incoming instruction does not relate to a control transfer (or termination), an encoding of IDLE state information can be associated with the one or more μops to indicate that state machine 116 remains in the IDLE state.

[0043] Accordingly, the decode unit 110 outputs a stream of μops and associated state information to indicate a state of the state machine 116 within the CTT logic 115. These μops and state information can be provided to an execution unit 120, which can include various types of units including arithmetic logic units (ALUs), floating point units and so forth that execute operations indicated by the stream of μops. In an embodiment, the CTT instructions can only control the state transitions in the state machine 200, and in execution logic of the processor, these instructions execute as NOP and do not cause a change in the program semantics.

[0044] In turn, results of executing the μops are provided to a retirement unit 130 configured to determine whether given operations were successfully performed and to retire the μops when successfully performed. If a μop is not successfully performed, the retirement unit 130 can raise a fault or exception if an undesired condition occurs as a result of the execution. In an out-of-order processor, the retirement unit 130 can further operate to reorder instructions which can be executed in any order, back into program order. When μops properly retire, the μops can be provided to further portions of a processor such as a memory subsystem.

[0045] With further reference to Figure 1, the retirement unit 130 includes a CTT exception logic 135 that can be configured to determine whether appropriate behavior occurs with regard to control transfer instructions. More specifically, the CTT exception logic 135 can operate to raise an execution exception when a given μop decoded from a control transfer instruction to be retired is not directly followed by an appropriate CTT μop, as described herein. In an embodiment, this determination can be based at least in part on an inserted fault μop and the state information communicated with the μops exiting from decode unit 110. If a CTT fault μop is detected, an execution exception is raised and is communicated to a fault handler 140, which can take various actions in accordance with a given handler to resolve the faulting behavior. Thus, in an embodiment, when a next μop presented to retire after a control transfer μop is not an

appropriate CTT μop, the retirement unit 130 can deliver an fault-class execution exception (e.g., a general protection fault) responsive to this CTT fault μop such that the μop does not retire.

[0046] Still referring to Figure 1, in the case where a misprediction occurs and μops are to be re-executed according to a correct branch, the retirement unit 130 can communicate via a feedback path 138 with decode unit 110 to thus provide an indication of a proper branch or other code flow to be taken. Still further, a state machine recovery signal can be communicated via this feedback path 138 such that the state machine 116 of CTT logic 115 can be placed into an appropriate state to reflect this change in program flow. Stated another way, when a fault μop is present in a mispredicted path, an execution exception is not raised due to this misprediction, and accordingly, the state machine recovery signal can cause the state machine 116 to pass from a WAIT state back to an IDLE state (or to remain in a WAIT state), and to also indicate the last successful instruction to retire, so that the decode unit 110 can decode instructions of the correct branch.

[0047] Figure 2 is a block diagram of a state machine 200 in accordance with an embodiment of the present disclosure. The state machine 200 can correspond to the CTT state machine 116 of Figure 1 or to other processing logic of the portion 100 of the processor. The state machine 200 begins operation in an IDLE state 210 into which the state machine is placed by the CTT logic 115 after reset of a processor.

[0048] When some other instructions or an indirect CALL or JMP instruction that includes a certain subcode (such as 3EH or other opcode, for example) is decoded, the state machine can remain in the IDLE state 210. The subcode can be included as a prefix, a suffix or somewhere within the CALL or JMP instruction (e.g., in a header) that signals to the decode unit 110 that the state machine 200 is not to track the CALL or JMP instruction and carries no risk of aiding to the generation of a gadget. Table 1, below, is an example of the subcode functioning in this way so that the CALL or JMP instruction is not tracked.

TABLE 1

```
int main(int argc, char **argv)
{
        int i;
        i = foo();
        switch(i)
        {
                case 1:
                        printf("Case 1\n");
                        break;
```

```
            case 2:
                printf("Case 2\n");
                break;
            case 3:
                printf("Case 3\n");
                break;
            case 4:
                printf("Case 4\n");
                break;
            case 5:
                printf("Case 5\n");
                break;
            default:
                printf("Nothing\n");
                break;
        }
        return 0;
}
```

```
0000000000400a50 <main>:
400a50:     f3 0f 1e fa              endbr64
400a54:     55                       push   %rbp
400a55:     48 89 e5                 mov    %rsp,%rbp
400a58:     48 83 e4 80              and    $0xffffffffffffff80,%rsp
400a5c:     48 81 ec 80 00 00 00     sub    $0x80,%rsp
400a63:     33 f6                    xor    %esi,%esi
400a65:     bf 03 00 00 00           mov    $0x3,%edi
400a6a:     e8 81 00 00 00           callq  400af0 <new_feature_proc_init>
400a6f:     0f ae 1c 24              stmxcsr (%rsp)
400a73:     81 0c 24 40 80 00 00     orl    $0x8040,(%rsp)
400a7a:     0f ae 14 24              ldmxcsr (%rsp)
400a7e:     0f 31                    rdtsc
400a80:     ff c8                    dec    %eax
400a82:     83 f8 04                 cmp    $0x4,%eax
400a85:     77 47                    ja     400ace <main+0x7e>
```

```
400a87:   48 8b 04 c5 08 1d 40   mov    0x401d08(,%rax,8),%rax
400a8e:   00
400a8f:   3e ff eo               no-trk jmpq   *%rax
400a92:   bf 30 1d 40 00         mov    $0x401d30,%edi
400a97:   e8 84 fe ff ff         callq  400920 <puts@plt>
400a9c:   eb 3a                  jmp    400ad8 <main+0x88>
400a9e:   bf 38 1d 40 00         mov    $0x401d38,%edi
400aa3:   e8 78 fe ff ff         callq  400920 <puts@plt>
400aa8:   eb 2e                  jmp    400ad8 <main+0x88>
400aaa:   bf 40 1d 40 00         mov    $0x401d40,%edi
400aaf:   e8 6c fe ff ff         callq  400920 <puts@plt>
400ab4:   eb 22                  jmp    400ad8 <main+0x88>
400ab6:   bf 48 1d 40 00         mov    $0x401d48,%edi
400abb:   e8 60 fe ff ff         callq  400920 <puts@plt>
400ac0:   eb 16                  jmp    400ad8 <main+0x88>
400ac2:   bf 50 1d 40 00         mov    $0x401d50,%edi
400ac7:   e8 54 fe ff ff         callq  400920 <puts@plt>
400acc:   eb 0a                  jmp    400ad8 <main+0x88>
400ace:   bf 58 1d 40 00         mov    $0x401d58,%edi
400ad3:   e8 48 fe ff ff         callq  400920 <puts@plt>
400ad8:   33 c0                  xor    %eax,%eax
400ada:   48 89 ec               mov    %rbp,%rsp
400add:   5d                     pop    %rbp
400ade:   c3                     retq
```

[0049] In this example, the C code at the top of Table 1 is translated by a compiler 108 into the sequence of instructions that is shown below the C program. Here the code at address 400a8f is the no-track (subcode) jump emitted by the compiler. The compiler 108 can safely emit the no-track indirect jump here because the compiler has sanitized the jump locations at address 400a82 and 400a85 to ensure that the jump target is within bounds and the offset used for the targets of the jump are known at compile time. In this way, the compiler 108 can avoid emitting the ENDBRANCH64 (endbr64) instruction at the targets of, for example, addresses 400aaa and 400ab6 that follow this CALL or JMP instruction.

[0050] When an indirect CALL or JMP instruction lacking the subcode is decoded (or any far CALL or JMP or a CALL or JMP that accesses memory is decoded), the state machine enters a WAIT_FOR_ENDBRANCH state 220. When the next instruction that is decoded is not an

ENDBRANCH instruction of the proper operation mode (e.g., ENDBRANCH32 for 32-bit or compatibility modes and ENDBRANCH64 for 64-bit mode), then the state machine 200 performs a DELIVER_EXECUTION_EXCEPTION operation 230 which can also cause generation of a fault μop (and the state machine 200 can remain in the WAIT_FOR_ENDBRANCH state 220). When, instead, the next instruction to be decoded following a control transfer instruction is an ENDBRANCH instruction of the proper mode, the state machine 200 transitions back to the IDLE state 210.

[0051] The INT3, shown as circling within the WAIT_FOR_ENDBRANCH state 220, is an instruction that is used to invoke a debugger for the state machine 200. When an INT3 instruction is encountered at the target of an indirect CALL or JMP, the state machine 200 is maintained in the WAIT_FOR_ENDBRANCH state 220. This is to support breakpoint generation by debuggers. The way a debugger sets up breakpoints in a program is by replacing the code byte at the address where the debugger wants to break with an INT3 instruction. This gives control to the debugger. The debugger can then examine the program state and then return the original code bytes that were replaced by the INT3 and restart program execution at that address. By retaining the state of the program as WAIT-FOR-ENDBRANCH when this program is restarted by the debugger, the hardware verifies whether the debugger has put back the code bytes for ENDBRANCH (that it had originally replaced with INT3).

[0052] When a RET instruction is decoded, the state machine 200 can determine whether the RET instruction includes the subcode indicating that a target of the RET instruction is valid and safe. (Note that in some instruction architectures, the check for the subcode can be unnecessary if the RET instructions do not access memory to determine a target address, e.g., there is no threat of gadget generation.) When the RET instruction includes the subcode, the state machine 200 can remain in the IDLE state. Otherwise, when the RET instruction lacks the subcode, the state machine can enter the WAIT_FOR_ENDRET state 240.

[0053] When the next instruction that is decoded is not an ENDRET instruction of the proper mode (e.g., ENDRET32 for 32-bit or compatibility modes and ENDRET64 for 64-bit mode), the state machine 200 performs the DELIVER_ EXECUTION_EXCEPTION operation 230. When the next instruction that is decoded is an ENDRET instruction of the proper mode, the state machine 200 transitions back to the IDLE state 210.

[0054] The state machine 200 thus enforces the following rules: the instruction at the target of a RET instruction must be an ENDRET instruction of the proper mode and the instruction at the target of an indirect CALL or indirect JMP instruction must be an ENDBRANCH instruction of the proper mode. When these rules are violated, then the violating instruction (the instruction at

the target of a RET or CALL/JMP instruction) faults and is prevented from retiring by the retirement unit 130.

[0055] Thus by placing ENDBRANCH32 or ENBRANCH64 and ENDRET32 or ENRET64 instructions in a program at valid control transfer locations, a programmer or compiler can prevent unintended control transfers from happening. This placement of ENDBRANCH and ENDRET instructions is as illustrated below in Table 2, as an example:

TABLE 2

```
main( ) {
   int (*f)( );
   f = foo;
   f( );
}
int foo( ) {
   return
}
0000000000400513
<main>:
endbranch64
push %rbp
mov %rsp,%rbp
sub $0x10, %rsp
movq $0x4004fb, –8(%rbp)
mov –8(%rbp), %rdx
mov $0x0, %eax
call *%rdx
endret64
Leaveq
Retq
00000000004004fb <foo>:
endbranch64
```

TABLE 2

```
push %rbp
mov %rsp,%rbp
leaveq
retq
```

[0056] In this example, the C code at the top of Table 2 is translated by a compiler into the sequence of instructions that is shown below the C program. The compiler 108 places an ENDBRANCH64 instruction as the first instruction in the subroutine foo and in the main program, and an ENDRET64 instruction is placed after the CALL instruction to subroutine foo. Thus there are now three ("3") valid control transfer points in this program. Specifically, in execution of the main program, a call instruction (call *%rdx) is executed, causing a control transfer to the subroutine foo., the first instruction in this subroutine is an ENDBRANCH64 instruction, such that a valid control transfer occurs and the CTT state machine thus proceeds from an IDLE state, to a WAIT_FOR_ENDBRANCH state and back to the IDLE state, without raising a fault.

[0057] Similarly, at the conclusion of the subroutine foo, a return instruction (RETQ) is executed, thus causing control to transfer to the first instruction after the calling instruction in the main program. Here, this instruction is an ENDRET64 instruction and as such, a valid control transfer occurs. In this case, the CTT state machine proceeds from the IDLE state, to the WAIT_FOR_ENDRET state, and thereafter back to the IDLE state, without raising a fault.

[0058] Thus, using CTT processing logic in accordance with an embodiment of the present disclosure, a constraint is introduced that a ROP gadget be preceded with an ENDRET32 (or ENDRET64) to be usable, effectively eliminating the possibility for a sequence of instructions to be used as a ROP gadget. As such, a significant reduction in the number of gadgets that can be harvested from a library is realized, and the quality of such gadgets is significantly lower in terms of functionality that these remaining gadgets expose, making ROP attacks harder to execute.

[0059] Figure 3A is a flow chart of a method 300 of a compiler deciding whether to add a pre-defined subcode to control transfer instructions of a high-level programming language. The method can begin by receiving, by a processing device executing a high-level programming language compiler, a list of keywords identifying a programming language statement to be translated into control transfer commands for execution in a non-tracked mode (301). The keywords can include a switch-case statement or a table lookup, for example, where the destination address is known at compile time.

[0060] The method 300 can continue by determining whether the programming language statement includes a keyword from the list of keywords (302). Responsive to determining that the programming language statement includes a keyword from the list of keywords, the method can continue by translating the programming language statement into a control transfer instruction having a pre-defined subcode, the pre-defined subcode to instruct a processor not to track the control transfer instruction (304). Responsive to determining that the programming language statement does not include a keyword from the list of keywords, the method can continue by translating the programming language statement into a trackable control transfer instruction (306) and emitting an ENDBRANCH instruction to follow the trackable control transfer instruction (308).

[0061] Figure 3B is a flow chart of a method 310 for non-tracked control transfers and mode-specific control flow termination according to an embodiment of the present disclosure. The method 310 can be performed by a compiler and by a processor that includes a control transfer termination (CTT) state machine (e.g., processing logic of a decode unit). The method 300 can determine whether a control transfer instruction needs to access memory to determine an address of a target instruction of an indirect call, jump or return (control transfer) instruction (312). The method 310 can also determine whether a target instruction of the control transfer instruction is computed at compile time by the compiler (330). When the answer is no in block 312 and/or yes in block 330, the compiler can add a subcode (such as a prefix or suffix) to the control transfer instruction (315). The state machine can then be placed or retained in an IDLE state (320) for that control transfer instruction, after decoded, in response to detecting the subcode in the control transfer instruction.

[0062] When the answer is yes to block 312 and/or no to block 330, the compiler can compile program instructions as before, adding CTT instructions following control transfer instructions as directed (335). The method 310 can then continue, during decoding, by determining whether the control transfer instruction is followed by a CTT instruction (340). When no, then the processing logic raises an execution exception or generates a fault micro-operation as in step 230 of Figure 2 (360). If the answer is yes to block 340, then the method continues with determining whether the CTT instruction is of the same mode as the processing logic following the execution of the control transfer instruction (whether 32-bit mode or 64-bit mode, for example) (345). When the answer is yes to block 345, then the processing logic can place or retain the state machine in an IDLE state (350), corresponding to state 210 in Figure 2. When the answer is no to block 345, then the processing logic raises an execution exception or generates a fault micro-operation as discussed with respect to as in step 230 of Figure 2 (360). The processor can then take appropriate action in response to the execution exception or the fault. In one embodiment,

the retirement unit 130 can send an execution exception to the fault handler 140 upon receipt of the fault micro-operation through the pipeline of the portion 100 of the processor.

[0063] Figure 4 is a flow chart of another method 400 for mode-specific control flow termination according to another embodiment of the present disclosure. The method 400 can be performed by logic of a retirement unit to handle CTT-based retirement operations. The method 400 can begin by determining whether a control transfer micro-instruction (μop) associated with a WAIT state (from a CTT state machine) is followed by a CTT μop (410). When the answer is no, then the method generates a fault and sends the fault to a fault handler (450). When the answer is yes, then the method 400 can further determine whether the CTT μop is of the same mode as the current mode of the machine (420), whether of a 32-bit or a 64-bit mode, for example. When the answer to block 420 is yes, the logic can retire the CTT μop (430), signal to the state machine to move the state to IDLE (435), and perform further retirement operations (440). When the answer is no at block 420, the logic can generate and send an execution exception to the fault handler (450). The logic can also retain the target μop in active execution, e.g., not retire the target μop (which could belong to a gadget) contingent on the fault handler resolving the execution exception (460).

[0064] Figure 5A is a flow diagram of a method 500 of control transfer management in accordance with an embodiment of the present disclosure. As shown, the method 500 can be performed by a processor including a control transfer termination (CTT) state machine as described herein. Note that the operations shown in Figure 5A relate to state machine operations for control transfer-related instructions. For other instructions, if the state machine is currently in the IDLE state, the state machine remains in the IDLE state. The method 500 can begin by determining whether a feedback signal is received to update the CTT state machine (510). In an embodiment, this feedback signal can be received from a retirement unit or fault handler to cause the state of the state machine to transition to a given state, e.g., due to a misprediction (as from a retirement unit) or responsive to resolving a fault (as from a fault handler). If such feedback signal was received, control passes to block 515 where the state machine is updated with the state communicated through this feedback signal.

[0065] From both of these cases, control passes next to block 520 where an indication that an indirect control transfer instruction such as a call, jump or return has occurred (assuming that the decode unit has decoded such an instruction). And as such, control passes to block 525 where a transition into a WAIT sate of the state machine can occur.

[0066] Still referring to Figure 5A, method 500 can further determine whether an indication of a control transfer termination instruction of the correct mode is received (530). If so, control

passes to block 535 where the IDLE state of the state machine is entered, as pursuant to this proper CTT instruction following the control transfer instruction, a valid control transfer occurs.

[0067] When instead the method 500 determines that the next decoded instruction is not a CTT instruction or is a CTT instruction that does not match the current mode of the machine, control passes to block 540 where a control transfer termination fault instruction can be inserted into the processor pipeline. Note here that the state of the state machine does not change and thus remains in the selected WAIT state. In an embodiment, this fault instruction is a μop that travels through the processor pipeline and if it is selected for retirement, the retirement unit causes a fault to enable an OS-based fault handler to execute to determine the cause of the fault and take appropriate action.

[0068] Figure 5B is a flow diagram of another method 550 of control transfer management in accordance with an embodiment of the present disclosure. The method 550 can be performed at least in part by logic of a retirement unit to handle CTT-based retirement operations. As seen, the method 550 begins by retiring a given micro-instruction (a μop) and storing a CTT state associated with the μop (555). In an embodiment, this information can be stored in a given storage of the retirement unit such as reorder buffer entry. As will be discussed further below, this state can be used in case a misprediction occurs. Next, the method 500 determines whether a misprediction has occurred. If so, control passes to block 570 where information regarding the last validly retired μop present in an entry of the reorder buffer can be obtained and sent back to CTT logic (of the decode unit) to enable updating the state of the state machine into the appropriate state. There further typical retirement operations can continue (575).

[0069] Referring still to Figure 5B, when a fault μop is received (580), control passes to block 585 where a call can be issued to a fault handler. As an example, an OS-based fault handler can be executed. As part of this fault handling, when the fault is due to a CTT fault μop, a supervisor-based CTT state machine can be enabled and used to access the state of the user mode CTT state machine to determine the reason for the fault and to act accordingly. As an example, a target μop (namely a non-CTT target μop) can be prevented from retiring and an appropriate correction mechanism can be performed. Or, the fault handler can take any other action. As part of such operations, the fault handler can cause the user mode CTT state machine to be set to the appropriate state. After completion of the fault handler, retirement operations can be resumed responsive to control of the fault handler (590).

[0070] With CTT instructions enforcing valid control transfer locations, software checks can be placed after these instructions to further check for valid control transfers using techniques like stack canaries. Stack canaries are used to detect stack buffer overflow. This is done by placing a random number on the stack before the return address on entering a function and checking this

random number before return from the function. The CTT logic 115 provides strong positions in the program to add additional checks for detecting buffer overflows. Without CTT logic, these checks could be bypassed. CTT logic, therefore, ensures that these checks cannot be bypassed since the function must be called at the Endbranch and not past the Endbranch to skip the checks. For the example discussed above, reference is made to Table 3:

TABLE 3

```
main( ) {
   foo( );
   endret32;
   <detour/hook to anti-malware code to perform branch
sanity check>
   ...
{
int foo( ) {
   return
}
```

[0071] In the example above, there is thus one place in the program (after the ENDRET32) where such a check is to be placed. Without the CTT processing logic, software cannot effectively check all places that can be used as gadgets as these gadgets can be crafted out of byte sequences in the middle of valid instructions.

[0072] The execution of a program using CTT instructions can be performed by a compiler. In an embodiment, a just-in-time (JIT) compiler or a regular (cc) compiler can perform the instrumentation of the CTT instructions. Alternately such instrumentation can be performed by rewriting the program binary to insert the CTT instructions using a binary rewriting tool that reconstructs the control flow graph from the application binary. The binary rewriting technique can be used in cases where the source of the binary is not available for recompilation. Such binary rewriting can also be done by anti-malware software using such tools.

[0073] In some cases, applications and libraries compiled with CTT instrumentation can be merged with libraries that are not compiled with CTT instrumentation, such as non-CTT instrumented libraries referred to herein as "legacy libraries."

[0074] To support continued operation with these legacy libraries, embodiments can provide additional instructions. In one such embodiment, a suppression instruction, referred to herein as a DISCTT instruction, can be used to suppress the CTT state machine such that it stays in the

IDLE state instead of transitioning to the WAIT_FOR_ENDBRANCH or WAIT_FOR_ENDRET states on an indirect CALL/JMP or RET, respectively. Additionally this instruction returns, in a general purpose register, the state of the CTT suppression at the time the instruction was issued. An enable instruction, referred to herein as an ENCTT instruction, is used to remove the suppression of the CTT state machine put in place by the DISCTT instruction such that the state machine enforces the CTT rules. Additionally, this instruction returns the state of the CTT suppression at the time the instruction was issued.

[0075] The use of DISCTT and ENCTT instructions can be enabled for a process by an operating system. If the operating system does not allow a program to disable CTT, then the DISCTT instruction executes as a NOP and does not suppress CTT.

[0076] The use of the DISCTT and ENCTT instructions in a program to perform legacy interworking is illustrated below in Table 4.

TABLE 4

```
// Issue a DISCTT before invoking a legacy library function foo( )
temp_variable = DISCTT;
foo( );
// If CTT was suppressed by DISCTT prior to this legacy library call then
un-suppress it
IF (temp_variable == NOT_SUPPRESSED)
   ENCTT;
ENDIF
```

[0077] Returning the previous state of CTT as a result of the DISCTT instruction allows for supporting call chains like below:

CTT_function1->legacy_function1->CTT_function2->legacy_function2.

[0078] Here the CTT_function1 issues a DISCTT instruction before calling the legacy_function1. The DISCTT instruction returns the current state of CTT functionality as NOT_SUPPRESSED and then suppresses the CTT functionality. The legacy_function1 calls the CTT_function2. Now when the CTT_function2 calls legacy_function2, it again issues a DISCTT instruction. The DISCTT instruction now returns the current state of the CTT functionality as SUPPRESSED since it has been suppressed by CTT_function1. When the control returns from legacy_function2 to CTT_function2, it does not un-suppress the CTT functionality since it was already suppressed when it was invoked. When the control returns to CTT_function1, it un-suppresses the CTT functionality using the ENCTT instruction since it was suppressed by that function.

[0079] Returning the previous state of CTT responsive to the ENCTT instruction allows for a CTT-enabled library function to be called by a non-CTT enabled library/application to un-suppress CTT before it starts executing and suppress CTT functionality before returning to the caller, if it was suppressed when the function was called.

[0080] This is as illustrated below in Table 5:

TABLE 5

```
Legacy_function1( )
{
   CTT_function1( );
}
CTT_function1( )
{
   //ENDBRANCH is a NOP if this function was called with CTT
   suppressed/disabled ENDBRANCH;
   // Un-suppress CTT. If already unsuppressed this is gratuitous
   temp_variable = ENCTT;
   ....
   ....
   ....
   // If CTT was suppressed when this function was called the suppress
   // it before returning
   IF ( temp_variable == SUPPRESSED )
      DISCTT;
   ENDIF
   RET;
}
```

[0081] Figure 6 is a block diagram of a configuration register in accordance with an embodiment of the present disclosure. As shown in Figure 6, configuration register 600 can include various fields to store state values used in performing CTT functionality. In an embodiment, two such configuration registers can be provided, with each register associated with a particular mode of operation. Stated another way, one configuration register can be used

to control CTT operation in a user mode (e.g., ring 3) while a second configuration register can be used to control CTT functionality in a supervisor mode (e.g., rings less than 3).

[0082] In the embodiment shown, configuration register 600 includes an enable field 605 to store an enable indicator to indicate whether CTT is enabled for the current privilege level. A legacy enable field 610 can be used to store an indicator to indicate whether legacy interworking is enabled. A suppression field 615 can be used to store a suppression indicator to indicate whether CTT faults and tracking are to be suppressed. A tracker field 620 can be used to store a value of the CTT state machine. In an embodiment, this tracker field can be three bits where a value of zero ("0") can indicate the IDLE state, a value of one ("1") can indicate a WAIT_FOR_ENDRET32 state, a value of two ("2") can indicate a WAIT_FOR_ENDRET64 state, a value of three ("3") can indicate a WAIT_ FOR_ENDBRANCH32 state, and a value of four ("4") can indicate a WAIT_FOR_ ENDBRANCH64 state. A reserved field 425 can be used for various extensions. Other fields can be present in other embodiments than those specifically shown in Figure 4.

[0083] Referring now to Figure 7A, shown is a block diagram of a call stack frame for code execution that interlaces CTT-enabled code and legacy code without CTT-enabled functionality. As shown in Figure 7, a code segment 750 includes a first CTT call stack frame 760 and a second CTT call stack frame 762 that in turn calls a legacy call stack frame 765. Thus at the point of calling this legacy call stack frame, the CTT functionality is disabled responsive to a DISCTT instruction. Thus at this point execution begins with CTT functionality disabled for a first legacy call stack frame 765 and a second legacy call stack frame 766. Note that as the called functions return back, at the point of returning to call stack frame 762, execution with CTT functionality is re-enabled by an ENCTT instruction.

[0084] As such, Figure 7A shows an example where a first transfer to legacy code suppresses CTT, which is done using indirect CALL/JMP instructions (not RET) for security reasons. Once CTT is suppressed by a DISCTT instruction, subsequent CALL/JMP/RET instructions can land on instructions other than ENDBRANCH/ENDRET without causing faults. CTT operation is unsuppressed when control returns to the point where suppression was done, via an ENCTT instruction.

[0085] Figure 7B is a block diagram of control transfer termination logic with legacy interworking in accordance with an embodiment of the present disclosure. As shown in Figure 7B, an implementation is present with a CTT-enabled application image 770 that issues a call to a CTT enabled library 775 (Call_1) that in turn initiates a call to a legacy library 785 (Call_2). In turn, legacy library 785 issues a call to a second CTT-enabled library 790 (Call_3). Also present is a heap/stack 780. After execution in second CTT-enabled library 790, control passes back to

legacy library 785 (RET_1), and from there control returns back to first CTT-enabled library 775 (RET_2), and finally control returns back to application image 770 (RET_3).

[0086] Note that upon Call_2, a legacy transfer occurs and thus CTT is suppressed via a DISCTT instruction. Accordingly, for Call_3, CTT remains suppressed, as it does for RET_1. Finally, RET_2 causes a return to the point of suppression and, as such, CTT is unsuppressed via an ENCTT instruction. Note that this legacy interworking can be entered when a legacy interworking enable indicator of a CTT control logic is set and an indirect control transfer (namely a jump or call) occurs to a non-ENDBRANCH instruction.

[0087] The DISCTT and ENCTT instructions can be placed in the program by the programmer if she is aware of the interworking, and/or these DISCTT and ENCTT instructions can be placed in the program by the compiler/linker when it is linking statically to legacy libraries.

[0088] When linking dynamically to libraries, a loader or anti-malware software can insert trampoline functions between the application and the library, where the trampoline functions use DISCTT and ENCTT instructions. For example, calls to functions in a legacy library that are dynamically linked to a CTT enabled application go through a trampoline function, which suppresses CTT and then calls the legacy library function. The legacy library function returns to the trampoline function that un-suppresses CTT and returns to the CTT-enabled application.

[0089] Embodiments can be used by anti-malware software to wrap non-CTT binaries such that they can be used with CTT-enabled binaries. In addition, anti-malware software can restrict the use of the gadgets that can be found in the program even with CTT in use. Embodiments can be particularly applicable to mobile and other portable low power systems, in that software only techniques to mitigate against ROP (like rewriting binaries to remove all instances of RET by use of functionally equivalent, but larger, more complex sequences), generally lead to much larger binaries and increase the execution time of the program and thereby are not suited for mobile applications where power efficiency is a primary concern.

[0090] Figure 8 is a block diagram of a processor core 800 in accordance with one embodiment of the present disclosure. As shown in Figure 8, the processor core 800 can be a multi-stage pipelined out-of-order processor. The core 800 can 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 of MIPS Technologies of Sunnyvale, Calif.; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, Calif.). It should be understood that the core 800 can support multithreading (executing two or more parallel sets of operations or threads), and can do so in a variety of ways including time-sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously

multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[0091] A processor including the core 800 can be a general-purpose processor, such as a Core™ i3, i5, i7, 2 Duo and Quad, Xeon™, Itanium™, XScale™ or StrongARM™ processor, which are available from Intel Corporation. Alternatively, the processor can be from another company, such as a design from ARM Holdings, Ltd, MIPS, etc. The processor can be a special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, co-processor, embedded processor, or the like. The processor can be implemented on one or more chips, and can be a part of and/or can be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

[0092] As shown in Figure 8, core 800 can operate at various voltages and frequencies as a result of integrated voltage regulator 809. As seen in Figure 8, core 800 includes front end units 810, which can be used to fetch instructions to be executed and prepare them for use later in the processor. For example, the front end units 810 can include a fetch unit 801, an instruction cache 803, and an instruction decoder 805. Instruction decoder 805 can include CTT logic 806 in accordance with an embodiment of the present disclosure, with an associated CTT state machine to perform CTT operations as described herein. In some implementations, the front end units 810 can further include a trace cache, along with microcode storage as well as a micro-operation storage. Fetch unit 801 can fetch macro-instructions, e.g., from memory or instruction cache 803, and feed them to instruction decoder 805 to decode the macro-instructions into primitives, e.g., micro-operations for execution by the processor.

[0093] Coupled between front end units 810 and execution units 820 is an out-of-order (OOO) engine 815 that can be used to receive the micro-instructions and prepare them for execution. More specifically, the OOO engine 815 can include various buffers to re-order flow of micro-instructions and allocate various resources needed for execution, as well as to provide renaming of logical registers onto storage locations within various register files such as a register file 830 and an extended register file 835. The register file 830 can include separate register files for integer and floating point operations. The extended register file 835 can provide storage for vector-sized units, e.g., 256 or 512 bits per register.

[0094] Various resources can be present in execution units 820, including, for example, various integer, floating point, and single instruction multiple data (SIMD) logic units, among other specialized hardware. For example, such execution units can include one or more arithmetic logic units (ALUs) 822, among other such execution units.

[0095] Results from the execution units can be provided to a retirement unit 840 including a reorder buffer (ROB). This ROB can include various arrays and logic to receive information associated with instructions that are executed. This information is then examined by retirement unit 840 to determine whether the instructions can be validly retired and result data committed to the architectural state of the processor, or whether one or more exceptions occurred that prevent a proper retirement of the instructions. Of course, retirement unit 840 can handle other operations associated with retirement. For retirement operations here, CTT logic 845 of the retirement unit 840 can store CTT state machine state received with incoming instructions, and feed this information back to the CTT state machine responsive to a misprediction.

[0096] Figure 9A is a block diagram illustrating a micro-architecture for a processor core 900 that can be incorporated into the processor of Figure 1 or execute the state machine 200 (or CTT processing logic) of Figure 2. Specifically, processor core 900 depicts an in-order architecture core and a register renaming logic, out-of-order issue/execution logic to be included in a processor according to at least one embodiment of the disclosure. The embodiments of the error correcting code that carry additional bits can be implemented by processor core 900.

[0097] The processor core 900 includes a front end unit 930 coupled to an execution engine unit 950, and both are coupled to a memory unit 970. The processor core 900 can include 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, processor core 900 can include a special-purpose core, such as, for example, a network or communication core, compression engine, graphics core, or the like. In one embodiment, processor core 900 can be a multi-core processor or can be part of a multi-processor system.

[0098] The front end unit 930 includes a branch prediction unit 932 coupled to an instruction cache unit 934, which is coupled to an instruction translation lookaside buffer (TLB) 936, which is coupled to an instruction fetch unit 938, which is coupled to a decode unit 940. The decode unit 940 (also known as a decoder) can 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 primary instructions. The decoder 940 can 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. The instruction cache unit 934 is further coupled to the memory unit 970. The decode unit 940 is coupled to a rename/allocator unit 952 in the execution engine unit 950.

[0099] The execution engine unit 950 includes the rename/allocator unit 952 coupled to a retirement unit 954 and a set of one or more scheduler unit(s) 956. The scheduler unit(s) 956 represents any number of different schedulers, including reservations stations (RS), central instruction window, etc. The scheduler unit(s) 956 can be coupled to the physical register file unit(s) 958. Each of the physical register file unit(s) 958 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, etc., status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. The physical register file(s) unit(s) 958 can be overlapped by the retirement unit 954 to illustrate various ways in which register renaming and out-of-order execution can be implemented (e.g., using a reorder buffer(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.).

[00100] Generally, the architectural registers are visible from the outside of the processor or from a programmer's perspective. The registers are not limited to any known particular type of circuit. Various different types of registers are suitable as long as they are capable of storing and providing data as described herein. Examples of suitable registers include, but are not limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. The retirement unit 954 and the physical register file(s) unit(s) 958 are coupled to the execution cluster(s) 960. The execution cluster(s) 960 includes a set of one or more execution units 962 and a set of one or more memory access units 964. The execution units 962 can perform various operations (e.g., shifts, addition, subtraction, multiplication) and operate on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point).

[00101] While some embodiments can include a number of execution units dedicated to specific functions or sets of functions, other embodiments can include only one execution unit or multiple execution units that all perform all functions. The scheduler unit(s) 956, physical register file(s) unit(s) 958, and execution cluster(s) 960 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 unit, physical register file(s) unit, 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) 964). It should also be understood that where separate pipelines are used, one or more of these pipelines can be out-of-order issue/execution and the rest in-order.

[00102] The set of memory access units 964 can be coupled to the memory unit 970, which can include a data prefetcher 980, a data TLB unit 972, a data cache unit (DCU) 974, and a level 2 (L2) cache unit 976, to name a few examples. In some embodiments DCU 974 is also known as a first level data cache (L1 cache). The DCU 974 can handle multiple outstanding cache misses and continue to service incoming stores and loads. It also supports maintaining cache coherency. The data TLB unit 972 is a cache used to improve virtual address translation speed by mapping virtual and physical address spaces. In one exemplary embodiment, the memory access units 964 can include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 972 in the memory unit 970. The L2 cache unit 976 can be coupled to one or more other levels of cache and eventually to a main memory.

[00103] In one embodiment, the data prefetcher 980 speculatively loads/prefetches data to the DCU 974 by automatically predicting which data a program is about to consume. Prefetching can refer to transferring data stored in one memory location (e.g., position) of a memory hierarchy (e.g., lower level caches or memory) to a higher-level memory location that is closer (e.g., yields lower access latency) to the processor before the data is actually demanded by the processor. More specifically, prefetching can refer to the early retrieval of data from one of the lower level caches/memory to a data cache and/or to prefetch buffer before the processor issues a demand for the specific data being returned.

[00104] The processor core 900 can 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 of Imagination Technologies of Kings Langley, Hertfordshire, UK; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA).

[00105] It should be understood that the core can support multithreading (executing two or more parallel sets of operations or threads), and can do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[00106] While register renaming is described in the context of out-of-order execution, it should be understood that register renaming can be used in an in-order architecture. While the illustrated embodiment of the processor also includes a separate instruction and data cache units and a shared L2 cache unit, alternative embodiments can have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system can include a combination of an internal cache

and an external cache that is external to the core and/or the processor. Alternatively, all of the cache can be external to the core and/or the processor.

[00107] Figure 9B is a block diagram illustrating an in-order pipeline and a register renaming stage, out-of-order issue/execution pipeline implemented by processor core 900 of Figure 9A according to some embodiments of the disclosure. The solid lined boxes in Figure 9B illustrate an in-order pipeline, while the dashed lined boxes illustrates a register renaming, out-of-order issue/execution pipeline. In Figure 9B, a processor pipeline 990 includes a fetch stage 902, a length decode stage 904, a decode stage 906, an allocation stage 908, a renaming stage 910, a scheduling (also known as a dispatch or issue) stage 912, a register read/memory read stage 914, an execute stage 916, a write back/memory write stage 918, an exception handling stage 922, and a commit stage 924. In some embodiments, the ordering of stages 902-924 can be different than illustrated and are not limited to the specific ordering shown in Figure 9B.

[00108] Figure 10 illustrates a block diagram of the micro-architecture for a processor 1000 (which can represent the processor of Figure 1 in one embodiment) that includes logic circuits. In some embodiments, an instruction in accordance with one embodiment can be implemented to operate on data elements having sizes of byte, word, doubleword, quadword, etc., as well as datatypes, such as single and double precision integer and floating point datatypes. In one embodiment, the in-order front end 1001 is the part of the processor 1000 that fetches instructions to be executed and prepares them to be used later in the processor pipeline.

[00109] The front end 1001 can include several units. In one embodiment, the instruction prefetcher 1016 fetches instructions from memory and feeds them to an instruction decoder 1018 which in turn decodes or interprets them. For example, in one embodiment, the decoder decodes a received instruction into one or more operations called "micro-instructions" or "micro-operations" (also called micro-op or μops) that the machine can execute. In other embodiments, the decoder parses the instruction into an opcode and corresponding data and control fields that are used by the micro-architecture to perform operations in accordance with one embodiment. In one embodiment, the trace cache 1030 takes decoded μops and assembles them into program ordered sequences or traces in the μop queue 1034 for execution. When the trace cache 1030 encounters a complex instruction, the microcode ROM (or RAM) 1032 can provide the μops needed to complete the operation.

[00110] Some instructions are converted into a single micro-op, whereas others need several micro-ops to complete the full operation. In one embodiment, if more than four micro-ops are needed to complete an instruction, the decoder 1018 accesses the microcode ROM 1032 to do the instruction. For one embodiment, an instruction can be decoded into a small number of micro ops for processing at the instruction decoder 1018. In another embodiment, an instruction can be

stored within the microcode ROM 1032 should a number of micro-ops be needed to accomplish the operation. The trace cache 1030 refers to an entry point programmable logic array (PLA) to determine a correct micro-instruction pointer for reading the micro-code sequences to complete one or more instructions in accordance with one embodiment from the micro-code ROM 1032. After the microcode ROM 1032 finishes sequencing micro-ops for an instruction, the front end 1001 of the machine resumes fetching micro-ops from the trace cache 1030.

[00111] The out-of-order execution engine 1003 is where the instructions are prepared for execution. The out-of-order execution logic has a number of buffers to smooth out and re-order the flow of instructions to optimize performance as they go down the pipeline and get scheduled for execution. The allocator logic allocates the machine buffers and resources that each uop needs in order to execute. The register renaming logic renames logic registers onto entries in a register file. The allocator also allocates an entry for each uop in one of the two uop queues, one for memory operations and one for non-memory operations, in front of the instruction schedulers: memory scheduler, fast scheduler 1002, slow/general floating point scheduler 1004, and simple floating point scheduler 1006. The μop schedulers 1002, 1004, 1006, determine when a uop is ready to execute based on the readiness of their dependent input register operand sources and the availability of the execution resources the μops need to complete their operation. The fast scheduler 1002 of one embodiment can schedule on each half of the main clock cycle while the other schedulers can only schedule once per main processor clock cycle. The schedulers arbitrate for the dispatch ports to schedule μops for execution.

[00112] Register files 1008, 1010, sit between the schedulers 1002, 1004, 1006, and the execution units 1012, 1014, 1016, 1018, 1020, 1022, 1024 in the execution block 1011. There is a separate register file 1008, 1010, for integer and floating point operations, respectively. Each register file 1008, 1010, of one embodiment also includes a bypass network that can bypass or forward just completed results that have not yet been written into the register file to new dependent μops. The integer register file 1008 and the floating point register file 1010 are also capable of communicating data with the other. For one embodiment, the integer register file 1008 is split into two separate register files, one register file for the low order 32 bits of data and a second register file for the high order 32 bits of data. The floating point register file 1010 of one embodiment has 128 bit wide entries because floating point instructions typically have operands from 64 to 128 bits in width.

[00113] The execution block 1011 contains the execution units 1012, 1014, 1016, 1018, 1020, 1022, 1024, where the instructions are actually executed. This section includes the register files 1008, 1010, that store the integer and floating point data operand values that the micro-instructions need to execute. The processor 1000 of one embodiment is comprised of a number

of execution units: address generation unit (AGU) 1012, AGU 1014, fast ALU 1016, fast ALU 1018, slow ALU 1020, floating point ALU 1022, floating point move unit 1014. For one embodiment, the floating point execution blocks 1022, 1024, execute floating point, MMX, SIMD, and SSE, or other operations. The floating point ALU 1022 of one embodiment includes a 64-bit-by-64-bit floating point divider to execute divide, square root, and remainder micro-ops. For embodiments of the present disclosure, instructions involving a floating point value can be handled with the floating point hardware.

[00114] In one embodiment, the ALU operations go to the high-speed ALU execution units 1016, 1018. The fast ALUs 1016, 1018, of one embodiment can execute fast operations with an effective latency of half a clock cycle. For one embodiment, most complex integer operations go to the slow ALU 1020 as the slow ALU 1020 includes integer execution hardware for long latency type of operations, such as a multiplier, shifts, flag logic, and branch processing. Memory load/store operations are executed by the AGUs 1012, 1014. For one embodiment, the integer ALUs 1016, 1018, 1020, are described in the context of performing integer operations on 64 bit data operands. In alternative embodiments, the ALUs 1016, 1018, 1020, can be implemented to support a variety of data bits including 16, 32, 128, 256, etc. Similarly, the floating point units 1022, 1024, can be implemented to support a range of operands having bits of various widths. For one embodiment, the floating point units 1022, 1024, can operate on 128 bits wide packed data operands in conjunction with SIMD and multimedia instructions.

[00115] In one embodiment, the μops schedulers 1002, 1004, 1006, dispatch dependent operations before the parent load has finished executing. As μops are speculatively scheduled and executed in processor 1000, the processor 1000 also includes logic to handle memory misses. If a data load misses in the data cache, there can be dependent operations in flight in the pipeline that have left the scheduler with temporarily incorrect data. A replay mechanism tracks and re-executes instructions that use incorrect data. Only the dependent operations need to be replayed and the independent ones are allowed to complete. The schedulers and replay mechanism of one embodiment of a processor are also designed to catch instruction sequences for text string comparison operations.

[00116] The processor 1000 also includes logic to implement compression/decompression optimization in solid-state memory devices according to one embodiment. In one embodiment, the execution block 1011 of processor 1000 can include MCU 115, to perform compression/decompression optimization in solid-state memory devices according to the description herein.

[00117] The term “registers” can refer to the on-board processor storage locations that are used as part of instructions to identify operands. In other words, registers can be those that are usable

from the outside of the processor (from a programmer's perspective). However, the registers of an embodiment should not be limited in meaning to a particular type of circuit. Rather, a register of an embodiment is capable of storing and providing data, and performing the functions described herein. The registers described herein can be implemented by circuitry within a processor using any number of different techniques, such as dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. In one embodiment, integer registers store thirty-two bit integer data. A register file of one embodiment also contains eight multimedia SIMD registers for packed data.

[00118] For the discussions herein, the registers are understood to be data registers designed to hold packed data, such as 64 bits wide MMX™ registers (also referred to as 'mm' registers in some instances) in microprocessors enabled with MMX technology from Intel Corporation of Santa Clara, California. These MMX registers, available in both integer and floating point forms, can operate with packed data elements that accompany SIMD and SSE instructions. Similarly, 128-bit wide XMM registers relating to SSE2, SSE3, SSE4, or beyond (referred to generically as "SSEx") technology can also be used to hold such packed data operands. In one embodiment, in storing packed data and integer data, the registers do not need to differentiate between the two data types. In one embodiment, integer and floating point are either contained in the same register file or different register files. Furthermore, in one embodiment, floating point and integer data can be stored in different registers or the same registers.

[00119] Embodiments can be implemented in many different system types. Referring now to Figure 11, shown is a block diagram of a multiprocessor system 1100 in accordance with an implementation. As shown in Figure 11, multiprocessor system 1100 is a point-to-point interconnect system, and includes a first processor 1170 and a second processor 1180 coupled via a point-to-point interconnect 1150. As shown in Figure 11, each of processors 1170 and 1180 can be multicore processors, including first and second processor cores (i.e., processor cores 1174a and 1174b and processor cores 1184a and 1184b), although potentially many more cores can be present in the processors. The processors each can include hybrid write mode logics in accordance with an embodiment of the present. The embodiments of the page additions and content copying can be implemented in the processor 1170, processor 1180, or both.

[00120] While shown with two processors 1170, 1180, it is to be understood that the scope of the present disclosure is not so limited. In other implementations, one or more additional processors can be present in a given processor.

[00121] Processors 1170 and 1180 are shown including integrated memory controller units 1172 and 1182, respectively. Processor 1170 also includes as part of its bus controller units

point-to-point (P-P) interfaces 1176 and 1188; similarly, second processor 1180 includes P-P interfaces 1186 and 1188. Processors 1170, 1180 can exchange information via a point-to-point (P-P) interface 1150 using P-P interface circuits 1178, 1188. As shown in Figure 11, IMCs 1172 and 1182 couple the processors to respective memories, namely a memory 1132 and a memory 1134, which can be portions of main memory locally attached to the respective processors.

[00122] Processors 1170, 1180 can each exchange information with a chipset 1190 via individual P-P interfaces 1152, 1154 using point to point interface circuits 1176, 1194, 1186, 1198. Chipset 1190 can also exchange information with a high-performance graphics circuit 1138 via a high-performance graphics interface 1139.

[00123] A shared cache (not shown) can be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information can be stored in the shared cache if a processor is placed into a low power mode.

[00124] Chipset 1190 can be coupled to a first bus 1116 via an interface 1196. In one embodiment, first bus 1116 can be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present disclosure is not so limited.

[00125] As shown in Figure 11, various I/O devices 1114 can be coupled to first bus 1116, along with a bus bridge 1118 which couples first bus 1116 to a second bus 1120. In one embodiment, second bus 1120 can be a low pin count (LPC) bus. Various devices can be coupled to second bus 1120 including, for example, a keyboard and/or mouse 1122, communication devices 1127 and a storage unit 1128 such as a disk drive or other mass storage device which can include instructions/code and data 1130, in one embodiment. Further, an audio I/O 1124 can be coupled to second bus 1120. Note that other architectures are possible. For example, instead of the point-to-point architecture of Figure 11, a system can implement a multi-drop bus or other such architecture.

[00126] Referring now to Figure 12, shown is a block diagram of a third system 1200 in accordance with an embodiment of the present disclosure. Like elements in Figures 11 and 12 bear like reference numerals, and certain aspects of Figure 11 have been omitted from Figure 11 in order to avoid obscuring other aspects of Figure 12.

[00127] Figure 12 illustrates that the processors 1270, 1280 can include integrated memory and I/O control logic ("CL") 1272 and 1282, respectively. For at least one embodiment, the CL 1272, 1282 can include integrated memory controller units such as described herein. In addition. CL 1272, 1282 can also include I/O control logic. Figure 12 illustrates that the memories 1232, 1234 are coupled to the CL 1272, 1282, and that I/O devices 1214 are also coupled to the control logic

1272, 1282. Legacy I/O devices 1215 are coupled to the chipset 1290. The embodiments of the page additions and content copying can be implemented in processor 1270, processor 1280, or both.

[00128] Figure 13 is an exemplary system on a chip (SoC) 1300 that can include one or more of the cores 1302. 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.

[00129] With further reference to the SOC 1300, dashed lined boxes are features on more advanced SoCs. In Figure 13 an interconnect unit(s) 1302 is coupled to: an application processor 1317 which includes a set of one or more cores 1302A-N and shared cache unit(s) 1306; a system agent unit 1310; a bus controller unit(s) 1316; an integrated memory controller unit(s) 1314; a set of one or more media processors 1320 which can include integrated graphics logic 1308, an image processor 1324 for providing still and/or video camera functionality, an audio processor 1326 for providing hardware audio acceleration, and a video processor 1328 for providing video encode/decode acceleration; a static random access memory (SRAM) unit 1330; a direct memory access (DMA) unit 1332; and a display unit 1340 for coupling to one or more external displays. The embodiments of the pages additions and content copying can be implemented in SoC 1300.

[00130] Turning next to Figure 14, an embodiment of a system on-chip (SoC) design in accordance with embodiments of the disclosure is depicted. As an illustrative example, SoC 1400 is included in user equipment (UE). In one embodiment, UE refers to any device to be used by an end-user to communicate, such as a hand-held phone, smartphone, tablet, ultra-thin notebook, notebook with broadband adapter, or any other similar communication device. A UE can connect to a base station or node, which can correspond in nature to a mobile station (MS) in a GSM network. The embodiments of the page additions and content copying can be implemented in SoC 1400.

[00131] Here, SoC 1400 includes 2 cores—1406 and 1407. Similar to the discussion above, cores 1406 and 1407 can conform to an Instruction Set Architecture, such as a processor having the Intel® Architecture Core™, an Advanced Micro Devices, Inc. (AMD) processor, a MIPS-based processor, an ARM-based processor design, or a customer thereof, as well as their licensees or adopters. Cores 1406 and 1407 are coupled to cache control 1408 that is associated

with bus interface unit 1409 and L2 cache 1410 to communicate with other parts of SOC 1400. Interconnect 1411 includes an on-chip interconnect, such as an IOSF, AMBA, or other interconnects discussed above, which can implement one or more aspects of the described disclosure.

[00132] Interconnect 1411 provides communication channels to the other components, such as a Subscriber Identity Module (SIM) 1430 to interface with a SIM card, a boot ROM 1435 to hold boot code for execution by cores 1406 and 1407 to initialize and boot SoC 1400, a SDRAM controller 1440 to interface with external memory (e.g., DRAM 1460), a flash controller 1445 to interface with non-volatile memory (e.g., Flash 1465), a peripheral control 1450 (e.g. Serial Peripheral Interface) to interface with peripherals, video codecs 1420 and Video interface 1425 to display and receive input (e.g. touch enabled input), GPU 1415 to perform graphics related computations, etc. Any of these interfaces can incorporate aspects of the embodiments described herein.

[00133] In addition, the system illustrates peripherals for communication, such as a Bluetooth module 1470, 3G modem 1475, GPS 1480, and Wi-Fi 1485. Note as stated above, a UE includes a radio for communication. As a result, these peripheral communication modules can not all be included. However, in a UE some form of a radio for external communication should be included.

[00134] Figure 15 illustrates a diagrammatic representation of a machine in the example form of a computing system 1500 within which a set of instructions, for causing the machine to perform any one or more of the methodologies discussed herein, can be executed. In alternative embodiments, the machine can be connected (e.g., networked) to other machines in a LAN, an intranet, an extranet, or the Internet. The machine can operate in the capacity of a server or a client device in a client-server network environment, or as a peer machine in a peer-to-peer (or distributed) network environment. The machine can be a personal computer (PC), a tablet PC, a set-top box (STB), a Personal Digital Assistant (PDA), a cellular telephone, a web appliance, a server, a network router, switch or bridge, or any machine capable of executing a set of instructions (sequential or otherwise) that specify actions to be taken by that machine. Further, while only a single machine is illustrated, the term “machine” shall also be taken to include any collection of machines that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methodologies discussed herein. The embodiments that execute the processor of Figure 1 can be implemented in or as a part of the computing system 1500.

[00135] The computing system 1500 includes a processing device 1502, main memory 1504 (e.g., flash memory, dynamic random access memory (DRAM) (such as synchronous DRAM

(SDRAM) or DRAM (RDRAM), etc.), a static memory 1506 (e.g., flash memory, static random access memory (SRAM), etc.), and a data storage device 1516, which communicate with each other via a bus 1508.

[00136] Processing device 1502 represents one or more general-purpose processing devices such as a microprocessor, central processing unit, or the like. More particularly, the processing device can be complex instruction set computing (CISC) microprocessor, reduced instruction set computer (RISC) microprocessor, very long instruction word (VLIW) microprocessor, or processor implementing other instruction sets, or processors implementing a combination of instruction sets. Processing device 1502 can also be one or more special-purpose processing devices such as an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), a digital signal processor (DSP), network processor, or the like. In one embodiment, processing device 1502 can include one or processor cores. The processing device 1502 is configured to execute the processing logic or instructions 1526 for performing the operations discussed herein.

[00137] In one embodiment, processing device 1502 can be the processor of Figure 1. Alternatively, the computing system 1500 can include other components as described herein. It should be understood that the core can support multithreading (executing two or more parallel sets of operations or threads), and can do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[00138] The computing system 1500 can further include a network interface device 1518 communicably coupled to a network 1519. The computing system 1500 also can include a video display device 1510 (e.g., a liquid crystal display (LCD) or a cathode ray tube (CRT)), an alphanumeric input device 1512 (e.g., a keyboard), a cursor control device 1514 (e.g., a mouse), a signal generation device 1520 (e.g., a speaker), or other peripheral devices. Furthermore, computing system 1500 can include a graphics processing unit 1522, a video processing unit 1528 and an audio processing unit 1532. In another embodiment, the computing system 1500 can include a chipset (not illustrated), which refers to a group of integrated circuits, or chips, that are designed to work with the processing device 1502 and controls communications between the processing device 1502 and external devices. For example, the chipset can be a set of chips on a motherboard that links the processing device 1502 to very high-speed devices, such as main memory 1504 and graphic controllers, as well as linking the processing device 1502 to lower-speed peripheral buses of peripherals, such as USB, PCI or ISA buses.

[00139] The data storage device 1516 can include a computer-readable storage medium 1524 on which is stored software 1526 embodying any one or more of the methodologies of functions described herein. The software 1526 can also reside, completely or at least partially, within the main memory 1504 as instructions 1526 and/or within the processing device 1502 as processing logic during execution thereof by the computing system 1500; the main memory 1504 and the processing device 1502 also constituting computer-readable storage media.

[00140] The computer-readable storage medium 1524 can also be used to store instructions 1526 utilizing the processing device 1502, such as described with respect to Figures 1-4, and/or a software library containing methods that call the above applications. While the computer-readable storage medium 1524 is shown in an example embodiment to be a single medium, the term "computer-readable storage medium" should be taken to include a single medium or multiple media (e.g., a centralized or distributed database, and/or associated caches and servers) that store the one or more sets of instructions. The term "computer-readable storage medium" shall also be taken to include any medium that is capable of storing, encoding or carrying a set of instruction for execution by the machine and that cause the machine to perform any one or more of the methodologies of the present embodiments. The term "computer-readable storage medium" shall accordingly be taken to include, but not be limited to, solid-state memories, and optical and magnetic media.

[00141] The following examples pertain to further embodiments.

[00142] Example 1 is a processor comprising: 1) an execution unit; and 2) a processing logic operatively coupled to the execution unit, the processing logic to: a) enter a first execution state; b) remain in the first execution state responsive to executing a control transfer instruction having a pre-defined subcode; c) transition into a second execution state responsive to executing a control transfer instruction not having the pre-defined subcode; and d) raise an execution exception responsive to determining, while in the second execution state, that a next instruction to be executed differs from an ENDBRANCH instruction.

[00143] In Example 2, the processor of Example 1, wherein the first execution state comprises an IDLE state and wherein the second execution state comprises a WAIT state.

[00144] In Example 3, the processor of Example 1, wherein the pre-defined subcode comprises a pre-defined opcode prefix.

[00145] In Example 4, the processor of Example 1, wherein the processing logic is further to, responsive to executing a control transfer instruction that is a far call or jump instruction, transition into the second execution state.

[00146] In Example 5, the processor of Example 1, wherein the ENDBRANCH instruction is a multi-byte opcode executable as a no operation.

[00147] In Example 6, the processor of Example 1, wherein the processing logic is further to, responsive to determining that the next instruction to be executed is the ENDBRANCH instruction, transition to the first execution state.

[00148] In Example 7, the processor of Example 1, wherein the processing logic includes a state machine to track an execution state of control transfer instructions.

[00149] Various embodiments can have different combinations of the structural features described above. For instance, all optional features of the computing system described above can also be implemented with respect to the method or process described herein and specifics in the examples can be used anywhere in one or more embodiments.

[00150] Example 8 is a processor comprising: 1) a decode unit to decode a first control transfer instruction, which generates a first micro-operation, wherein the first control transfer instruction includes a pre-defined subcode indicating to a processing logic of the decode unit to not track execution of the first micro-operation; and 2) wherein the processing logic is to raise an execution exception responsive to determining that a target instruction of a second control transfer instruction is not an ENDBRANCH instruction and the second control transfer instruction does not include the pre-defined subcode, wherein the ENDBRANCH instruction is a multi-byte opcode executable as a no operation.

[00151] In Example 9, the processor of Example 8, wherein the processing logic is further to associate a first state with the first micro-operation to indicate that the processing logic is in an IDLE state.

[00152] In Example 10, the processor of Example 8, wherein the processing logic is further to associate a second state with a second micro-operation decoded from the second control transfer instruction to indicate that the processing logic is in a WAIT state.

[00153] In Example 11, the processor of Example 8, further comprising: 1) an execution unit operatively coupled to the decode unit, the execution unit to execute micro-operations; and 2) a retirement unit operatively coupled to the execution unit, the retirement unit to raise the execution exception after processing a target micro-operation decoded from the target instruction.

[00154] In Example 12, the processor of Example 8, wherein the decode unit is further to decode the second control transfer instruction into a second micro-operation and to decode the target instruction into a target micro-operation, further comprising an execution pipeline, wherein the decode unit is further to: a) generate a fault micro-operation in response to determining that the target instruction of the second control transfer instruction is not an ENDBRANCH instruction and the second control transfer instruction does not include the

subcode; and b) feed the second micro-operation, the target micro-operation and the fault micro-operation to the execution pipeline.

[00155] In Example 13, the processor of Example 12, further comprising: 1) an execution unit operatively coupled to the decode unit, the execution unit to execute micro-operations; and 2) a retirement unit operatively coupled to the execution unit, the retirement unit to communicate the execution exception to a fault handler responsive to receipt of the fault micro-operation.

[00156] Various embodiments can have different combinations of the structural features described above. For instance, all optional features of the computing system described above can also be implemented with respect to the method or process described herein and specifics in the examples can be used anywhere in one or more embodiments.

[00157] Example 14 is a method comprising: 1) receiving, by a processing device executing a high-level programming language compiler, a list of keywords identifying a programming language statement to be translated into control transfer commands for execution in a non-tracked mode; 2) determining, by the processing device, whether the programming language statement comprises a keyword from the list of keywords; and 3) responsive to determining that the programming language statement comprises a keyword from the list of keywords, translating the programming language statement into a control transfer instruction having a pre-defined subcode, the pre-defined subcode to instruct a processor not to track the control transfer instruction.

[00158] In Example 15, the method of claim 14, wherein the determining comprises determining whether the keyword identifies a switch-case statement or a table lookup.

[00159] In Example 16, the method of claim 14, wherein the determining further comprises determining whether a control transfer command corresponding to the programming language statement, when executed, needs to access memory to retrieve a destination address, wherein the method further comprises, responsive to determining that the control transfer command does not need to access memory, translating the programming language statement into a control transfer instruction having the pre-defined subcode.

[00160] In Example 17, the method of claim 14, wherein translating comprises appending the subcode to the control transfer instruction as a pre-defined prefix opcode.

[00161] In Example 18, the method of claim 14, further comprising, responsive to determining that the programming language statement does not comprise a keyword from the list of keywords: a) translating the programming language statement into a trackable control transfer instruction; and b) emitting an ENDBRANCH instruction to follow the trackable control transfer instruction.

[00162] Various embodiments can have different combinations of the structural features described above. For instance, all optional features of the computing system described above can also be implemented with respect to the method or process described herein and specifics in the examples can be used anywhere in one or more embodiments.

[00163] Example 19 is a processor comprising: 1) a fetch unit to fetch instructions, wherein the instructions include a first control transfer instruction having a pre-defined subcode and a second control transfer instruction that is missing the pre-defined subcode; 2) a decode unit to decode the instructions and including a processing logic to: a) decode the first control transfer instruction into a first micro-operation; b) remain in a first execution state responsive to detecting the pre-defined subcode of the first control transfer instruction; c) decode the second control transfer instruction into a second micro-operation; and d) enter into a second execution state responsive to detecting an absence of the pre-defined subcode of the second control transfer instruction; 3) an execution unit operatively coupled to the decode unit and to execute the first micro-operation and the second micro-operation; and 4) a retirement unit to retire the first micro-operation and to raise an execution exception responsive to a next micro-operation to be retired after the second micro-operation being other than an ENDBRANCH micro-operation, wherein the ENDBRANCH micro-operation is generated from a multi-byte opcode executable as a no operation.

[00164] In Example 20, the processor of claim 19, wherein the retirement unit is further to: a) retain the next micro-operation in active execution; and b) send the execution exception to a fault handler for corrective action before deciding whether to retire the next micro-operation.

[00165] In Example 21, the processor of claim 19, wherein the first execution state is an IDLE state and the processing logic is further to associate the IDLE state with the first micro-operation.

[00166] In Example 22, the processor of claim 19, wherein the second execution state is a WAIT state and the processing logic is further to associate the WAIT state with the second micro-operation.

[00167] While the present disclosure has been described with respect to a limited number of embodiments, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this present disclosure.

[00168] In the description herein, numerous specific details are set forth, such as examples of specific types of processors and system configurations, specific hardware structures, specific architectural and micro architectural details, specific register configurations, specific instruction types, specific system components, specific measurements/heights, specific processor pipeline stages and operation etc. in order to provide a thorough understanding of the present disclosure.

It will be apparent, however, to one skilled in the art that these specific details need not be employed to practice the present disclosure. In other instances, well known components or methods, such as specific and alternative processor architectures, specific logic circuits/code for described algorithms, specific firmware code, specific interconnect operation, specific logic configurations, specific manufacturing techniques and materials, specific compiler implementations, specific expression of algorithms in code, specific power down and gating techniques/logic and other specific operational details of computer system have not been described in detail in order to avoid unnecessarily obscuring the present disclosure.

[00169] The embodiments are described with reference to compression/decompression optimization in solid-state memory devices in specific integrated circuits, such as in computing platforms or microprocessors. The embodiments can also be applicable to other types of integrated circuits and programmable logic devices. For example, the disclosed embodiments are not limited to desktop computer systems or portable computers, such as the Intel® Ultrabooks™ computers, and can be also used in other devices, such as handheld devices, tablets, other thin notebooks, systems on a chip (SoC) devices, and embedded applications. Some examples of handheld devices include cellular phones, Internet protocol devices, digital cameras, personal digital assistants (PDAs), and handheld PCs. Embedded applications typically include a microcontroller, a digital signal processor (DSP), a system on a chip, network computers (NetPC), set-top boxes, network hubs, wide area network (WAN) switches, or any other system that can perform the functions and operations taught below. It is described that the system can be any kind of computer or embedded system. The disclosed embodiments can especially be used for low-end devices, like wearable devices (e.g., watches), electronic implants, sensory and control infrastructure devices, controllers, supervisory control and data acquisition (SCADA) systems, or the like. Moreover, the apparatuses, methods, and systems described herein are not limited to physical computing devices, but can also relate to software optimizations for energy conservation and efficiency. As will become readily apparent in the description below, the embodiments of methods, apparatuses, and systems described herein (whether in reference to hardware, firmware, software, or a combination thereof) are vital to a 'green technology' future balanced with performance considerations.

[00170] Although the embodiments herein are described with reference to a processor, other embodiments are applicable to other types of integrated circuits and logic devices. Similar techniques and teachings of embodiments of the present disclosure can be applied to other types of circuits or semiconductor devices that can benefit from higher pipeline throughput and improved performance. The teachings of embodiments of the present disclosure are applicable to any processor or machine that performs data manipulations. However, the present disclosure is

not limited to processors or machines that perform 512 bit, 256 bit, 128 bit, 64 bit, 32 bit, or 16 bit data operations and can be applied to any processor and machine in which manipulation or management of data is performed. In addition, the description herein provides examples, and the accompanying drawings show various examples for the purposes of illustration. However, these examples should not be construed in a limiting sense as they are merely intended to provide examples of embodiments of the present disclosure rather than to provide an exhaustive list of all possible implementations of embodiments of the present disclosure.

[00171] Although the below examples describe instruction handling and distribution in the context of execution units and logic circuits, other embodiments of the present disclosure can be accomplished by way of a data or instructions stored on a machine-readable, tangible medium, which when performed by a machine cause the machine to perform functions consistent with at least one embodiment of the disclosure. In one embodiment, functions associated with embodiments of the present disclosure are embodied in machine-executable instructions. The instructions can be used to cause a general-purpose or special-purpose processor that is programmed with the instructions to perform the steps of the present disclosure. Embodiments of the present disclosure can be provided as a computer program product or software which can include a machine or computer-readable medium having stored thereon instructions which can be used to program a computer (or other electronic devices) to perform one or more operations according to embodiments of the present disclosure. Alternatively, operations of embodiments of the present disclosure might be performed by specific hardware components that contain fixed-function logic for performing the operations, or by any combination of programmed computer components and fixed-function hardware components.

[00172] Instructions used to program logic to perform embodiments of the disclosure can be stored within a memory in the system, such as DRAM, cache, flash memory, or other storage. Furthermore, the instructions can be distributed via a network or by way of other computer readable media. Thus a machine-readable medium can include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the computer-readable medium includes any type of tangible machine-readable

medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer).

[00173] A design can go through various stages, from creation to simulation to fabrication. Data representing a design can represent the design in a number of manners. First, as is useful in simulations, the hardware can be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates can be produced at some stages of the design process. Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, the data representing the hardware model can be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data can be stored in any form of a machine readable medium. A memory or a magnetic or optical storage such as a disc can be the machine readable medium to store information transmitted via optical or electrical wave modulated or otherwise generated to transmit such information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or re-transmission of the electrical signal is performed, a new copy is made. Thus, a communication provider or a network provider can store on a tangible, machine-readable medium, at least temporarily, an article, such as information encoded into a carrier wave, embodying techniques of embodiments of the present disclosure.

[00174] A module as used herein refers to any combination of hardware, software, and/or firmware. As an example, a module includes hardware, such as a micro-controller, associated with a non-transitory medium to store code adapted to be executed by the micro-controller. Therefore, reference to a module, in one embodiment, refers to the hardware, which is specifically configured to recognize and/or execute the code to be held on a non-transitory medium. Furthermore, in another embodiment, use of a module refers to the non-transitory medium including the code, which is specifically adapted to be executed by the microcontroller to perform predetermined operations. And as can be inferred, in yet another embodiment, the term module (in this example) can refer to the combination of the microcontroller and the non-transitory medium. Often module boundaries that are illustrated as separate commonly vary and potentially overlap. For example, a first and a second module can share hardware, software, firmware, or a combination thereof, while potentially retaining some independent hardware, software, or firmware. In one embodiment, use of the term logic includes hardware, such as transistors, registers, or other hardware, such as programmable logic devices.

[00175] Use of the phrase 'configured to,' in one embodiment, refers to arranging, putting together, manufacturing, offering to sell, importing and/or designing an apparatus, hardware, logic, or element to perform a designated or determined task. In this example, an apparatus or element thereof that is not operating is still 'configured to' perform a designated task if it is designed, coupled, and/or interconnected to perform said designated task. As a purely illustrative example, a logic gate can provide a 0 or a 1 during operation. But a logic gate 'configured to' provide an enable signal to a clock does not include every potential logic gate that can provide a 1 or 0. Instead, the logic gate is one coupled in some manner that during operation the 1 or 0 output is to enable the clock. Note once again that use of the term 'configured to' does not require operation, but instead focus on the latent state of an apparatus, hardware, and/or element, where in the latent state the apparatus, hardware, and/or element is designed to perform a particular task when the apparatus, hardware, and/or element is operating.

[00176] Furthermore, use of the phrases 'to,' 'capable of/to,' and or 'operable to,' in one embodiment, refers to some apparatus, logic, hardware, and/or element designed in such a way to enable use of the apparatus, logic, hardware, and/or element in a specified manner. Note as above that use of to, capable to, or operable to, in one embodiment, refers to the latent state of an apparatus, logic, hardware, and/or element, where the apparatus, logic, hardware, and/or element is not operating but is designed in such a manner to enable use of an apparatus in a specified manner.

[00177] A value, as used herein, includes any known representation of a number, a state, a logical state, or a binary logical state. Often, the use of logic levels, logic values, or logical values is also referred to as 1's and 0's, which simply represents binary logic states. For example, a 1 refers to a high logic level and 0 refers to a low logic level. In one embodiment, a storage cell, such as a transistor or flash cell, can be capable of holding a single logical value or multiple logical values. However, other representations of values in computer systems have been used. For example the decimal number ten can also be represented as a binary value of 1010 and a hexadecimal letter A. Therefore, a value includes any representation of information capable of being held in a computer system.

[00178] Moreover, states can be represented by values or portions of values. As an example, a first value, such as a logical one, can represent a default or initial state, while a second value, such as a logical zero, can represent a non-default state. In addition, the terms reset and set, in one embodiment, refer to a default and an updated value or state, respectively. For example, a default value potentially includes a high logical value, i.e. reset, while an updated value potentially includes a low logical value, i.e. set. Note that any combination of values can be utilized to represent any number of states.

[00179] The embodiments of methods, hardware, software, firmware or code set forth above can be implemented via instructions or code stored on a machine-accessible, machine readable, computer accessible, or computer readable medium which are executable by a processing element. A non-transitory machine-accessible/readable medium includes any mechanism that provides (i.e., stores and/or transmits) information in a form readable by a machine, such as a computer or electronic system. For example, a non-transitory machine-accessible medium includes random-access memory (RAM), such as static RAM (SRAM) or dynamic RAM (DRAM); ROM; magnetic or optical storage medium; flash memory devices; electrical storage devices; optical storage devices; acoustical storage devices; other form of storage devices for holding information received from transitory (propagated) signals (e.g., carrier waves, infrared signals, digital signals); etc., which are to be distinguished from the non-transitory mediums that can receive information there from.

[00180] Instructions used to program logic to perform embodiments of the disclosure can be stored within a memory in the system, such as DRAM, cache, flash memory, or other storage. Furthermore, the instructions can be distributed via a network or by way of other computer readable media. Thus a machine-readable medium can include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the computer-readable medium includes any type of tangible machine-readable medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer)

[00181] Reference throughout this specification to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present disclosure. Thus, the appearances of the phrases “in one embodiment” or “in an embodiment” in various places throughout this specification are not necessarily all referring to the same embodiment. Furthermore, the particular features, structures, or characteristics can be combined in any suitable manner in one or more embodiments.

[00182] In the foregoing specification, a detailed description has been given with reference to specific exemplary embodiments. It will, however, be evident that various modifications and

changes can be made thereto without departing from the broader spirit and scope of the disclosure as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense. Furthermore, the foregoing use of embodiment and other exemplarily language does not necessarily refer to the same embodiment or the same example, but can refer to different and distinct embodiments, as well as potentially the same embodiment.

[00183] Some portions of the detailed description are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers or the like. The blocks described herein can be hardware, software, firmware or a combination thereof.

[00184] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the above discussion, it is appreciated that throughout the description, discussions utilizing terms such as “defining,” “receiving,” “determining,” “issuing,” “linking,” “associating,” “obtaining,” “authenticating,” “prohibiting,” “executing,” “requesting,” “communicating,” or the like, refer to the actions and processes of a computing system, or similar electronic computing device, that manipulates and transforms data represented as physical (e.g., electronic) quantities within the computing system's registers and memories into other data similarly represented as physical quantities within the computing system memories or registers or other such information storage, transmission or display devices.

[00185] The words “example” or “exemplary” are used herein to mean serving as an example, instance or illustration. Any aspect or design described herein as “example’ or “exemplary” is not necessarily to be construed as preferred or advantageous over other aspects or designs. Rather, use of the words “example” or “exemplary” is intended to present concepts in a concrete fashion. As used in this application, the term “or” is intended to mean an inclusive “or” rather than an exclusive “or.” That is, unless specified otherwise, or clear from context, “X includes A or B” is intended to mean any of the natural inclusive permutations. That is, if X includes A; X

includes B; or X includes both A and B, then “X includes A or B” is satisfied under any of the foregoing instances. In addition, the articles “a” and “an” as used in this application and the appended claims should generally be construed to mean “one or more” unless specified otherwise or clear from context to be directed to a singular form. Moreover, use of the term “an embodiment” or “one embodiment” or “an implementation” or “one implementation” throughout is not intended to mean the same embodiment or implementation unless described as such. Also, the terms "first," "second," "third," "fourth," etc. as used herein are meant as labels to distinguish among different elements and can not necessarily have an ordinal meaning according to their numerical designation.

Claims

What is claimed is:

1. A processor comprising:

an execution unit; and

a processing logic operatively coupled to the execution unit, the processing logic to:

enter a first execution state;

remain in the first execution state responsive to executing a control transfer instruction having a pre-defined subcode;

transition into a second execution state responsive to executing a control transfer instruction not having the pre-defined subcode; and

raise an execution exception responsive to determining, while in the second execution state, that a next instruction to be executed differs from an ENDBRANCH instruction.

2. The processor of claim 1, wherein the first execution state comprises an IDLE state and wherein the second execution state comprises a WAIT state.

3. The processor of claim 1, wherein the pre-defined subcode comprises a pre-defined opcode prefix.

4. The processor of claim 1, wherein the processing logic is further to, responsive to executing a control transfer instruction that is a far call or jump instruction, transition into the second execution state.

5. The processor of claim 1, wherein the ENDBRANCH instruction is a multi-byte opcode executable as a no operation.

6. The processor of claim 1, wherein the processing logic is further to, responsive to determining that the next instruction to be executed is the ENDBRANCH instruction, transition to the first execution state.

7. The processor of claim 1, wherein the processing logic includes a state machine to track an execution state of control transfer instructions.

8. A processor comprising:

a decode unit to decode a first control transfer instruction, which generates a first micro-operation, wherein the first control transfer instruction includes a pre-defined subcode indicating to a processing logic of the decode unit to not track execution of the first micro-operation; and

wherein the processing logic is to raise an execution exception responsive to determining that a target instruction of a second control transfer instruction is not an ENDBRANCH instruction and the second control transfer instruction does not include the pre-defined subcode, wherein the ENDBRANCH instruction is a multi-byte opcode executable as a no operation.

9. The processor of claim 8, wherein the processing logic is further to associate a first state with the first micro-operation to indicate that the processing logic is in an IDLE state.

10. The processor of claim 8, wherein the processing logic is further to associate a second state with a second micro-operation decoded from the second control transfer instruction to indicate that the processing logic is in a WAIT state.

11. The processor of claim 8, further comprising:

an execution unit operatively coupled to the decode unit, the execution unit to execute micro-operations; and

a retirement unit operatively coupled to the execution unit, the retirement unit to raise the execution exception after processing a target micro-operation decoded from the target instruction.

12. The processor of claim 8, wherein the decode unit is further to decode the second control transfer instruction into a second micro-operation and to decode the target instruction into a target micro-operation, further comprising an execution pipeline, wherein the decode unit is further to:

generate a fault micro-operation in response to determining that the target instruction of the second control transfer instruction is not an ENDBRANCH instruction and the second control transfer instruction does not include the subcode; and

feed the second micro-operation, the target micro-operation and the fault micro-operation to the execution pipeline.

13. The processor of claim 12, further comprising:

an execution unit operatively coupled to the decode unit, the execution unit to execute micro-operations; and

a retirement unit operatively coupled to the execution unit, the retirement unit to communicate the execution exception to a fault handler responsive to receipt of the fault micro-operation.

14. A method comprising:

receiving, by a processing device executing a high-level programming language compiler, a list of keywords identifying a programming language statement to be translated into control transfer commands for execution in a non-tracked mode;

determining, by the processing device, whether the programming language statement comprises a keyword from the list of keywords; and

responsive to determining that the programming language statement comprises a keyword from the list of keywords, translating the programming language statement into a control transfer instruction having a pre-defined subcode, the pre-defined subcode to instruct a processor not to track the control transfer instruction.

15. The method of claim 14, wherein the determining comprises determining whether the keyword identifies a switch-case statement or a table lookup.

16. The method of claim 14, wherein the determining further comprises determining whether a control transfer command corresponding to the programming language statement, when executed, needs to access memory to retrieve a destination address, wherein the method further comprises, responsive to

determining that the control transfer command does not need to access memory, translating the programming language statement into a control transfer instruction having the pre-defined subcode.

17. The method of claim 14, wherein translating comprises appending the subcode to the control transfer instruction as a pre-defined prefix opcode.

18. The method of claim 14, further comprising, responsive to determining that the programming language statement does not comprise a keyword from the list of keywords:

translating the programming language statement into a trackable control transfer instruction; and

emitting an ENDBRANCH instruction to follow the trackable control transfer instruction.

19. A processor comprising:

a fetch unit to fetch instructions, wherein the instructions include a first control transfer instruction having a pre-defined subcode and a second control transfer instruction that is missing the pre-defined subcode;

a decode unit to decode the instructions and including a processing logic to:

decode the first control transfer instruction into a first micro-operation;

remain in a first execution state responsive to detecting the pre-defined subcode of the first control transfer instruction;

decode the second control transfer instruction into a second micro-operation; and

enter into a second execution state responsive to detecting an absence of the pre-defined subcode of the second control transfer instruction;

an execution unit operatively coupled to the decode unit and to execute the first micro-operation and the second micro-operation; and

a retirement unit to retire the first micro-operation and to raise an execution exception responsive to a next micro-operation to be retired after the second micro-operation being other than an ENDBRANCH micro-operation, wherein the ENDBRANCH micro-operation is generated from a multi-byte opcode executable as a no operation.

20. The processor of claim 19, wherein the retirement unit is further to:

retain the next micro-operation in active execution; and

send the execution exception to a fault handler for corrective action before deciding whether to retire the next micro-operation.

21. The processor of claim 19, wherein the first execution state is an IDLE state and the processing logic is further to associate the IDLE state with the first micro-operation.

22. The processor of claim 19, wherein the second execution state is a WAIT state and the processing logic is further to associate the WAIT state with the second micro-operation.

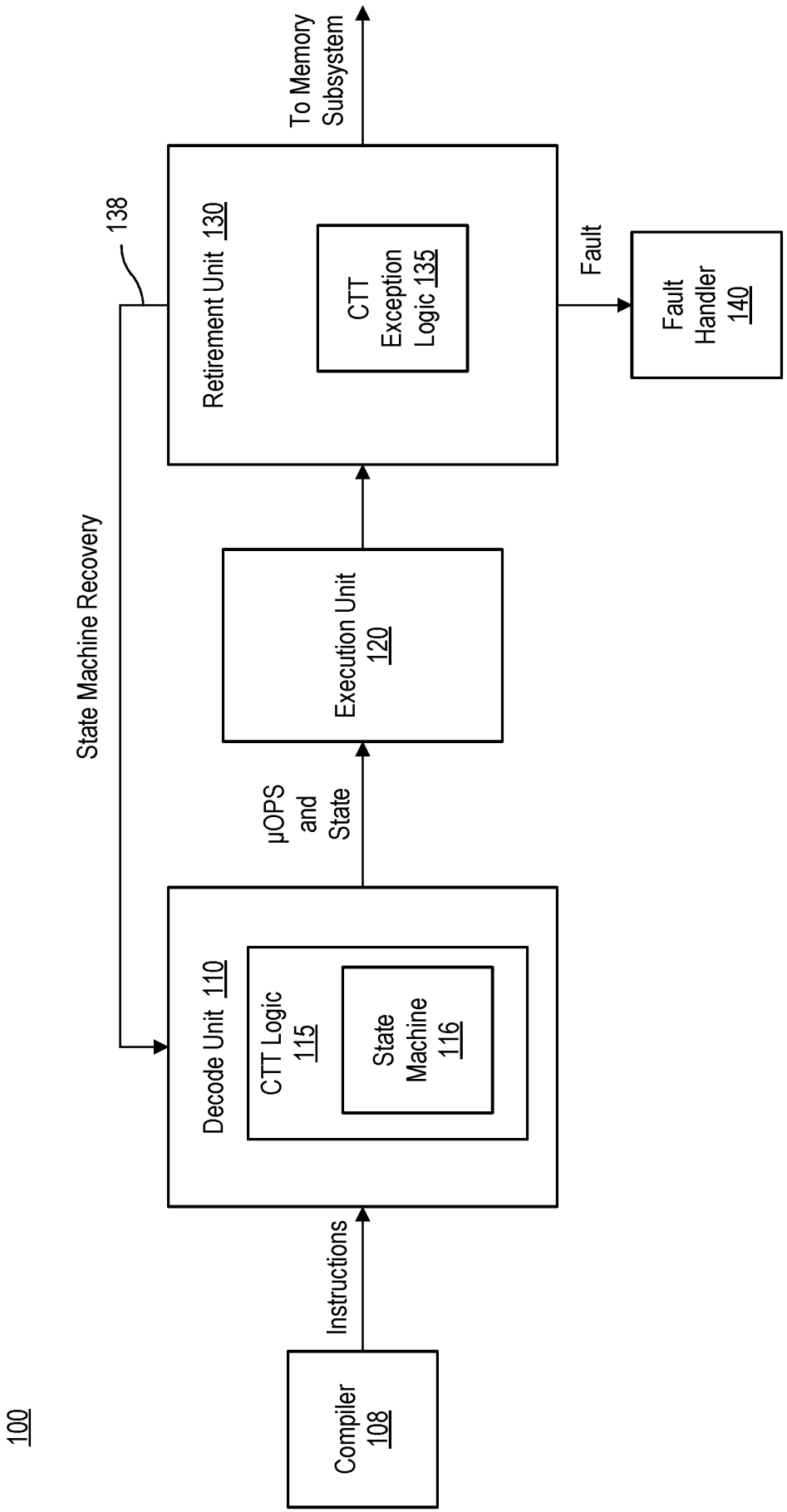
100
Compiler 108
Instructions
Decode Unit 110
CTT Logic 115
State Machine 116
μOPS and State
Execution Unit 120
Retirement Unit 130
CTT Exception Logic 135
138
State Machine Recovery
To Memory Subsystem
Fault
Fault Handler 140

FIG. 1

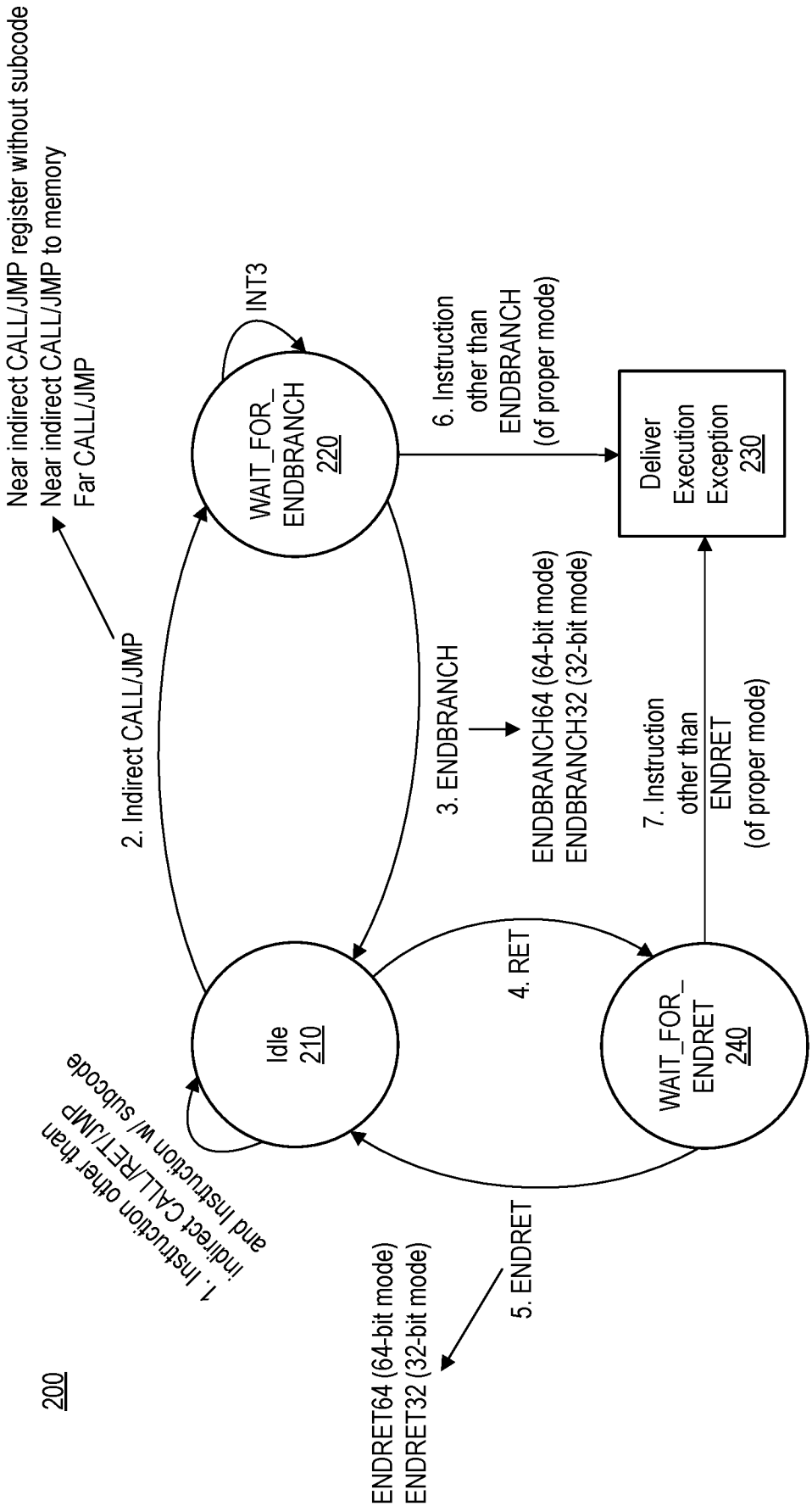
200
Near indirect CALL/JMP register without subcode
Near indirect CALL/JMP to memory
Far CALL/JMP
2. Indirect CALL/JMP
INT3
WAIT_FOR_ ENDBRANCH 220
6. Instruction other than ENDBRANCH (of proper mode)
Deliver Execution Exception 230
3. ENDBRANCH
ENDBRANCH64 (64-bit mode)
ENDBRANCH32 (32-bit mode)
7. Instruction other than ENDRET (of proper mode)
4. RET
Idle 210
WAIT_FOR_ ENDRET 240
1. Instruction other than indirect CALL/RET/JMP and instruction w/ subcode
5. ENDRET
ENDRET64 (64-bit mode)
ENDRET32 (32-bit mode)

FIG. 2

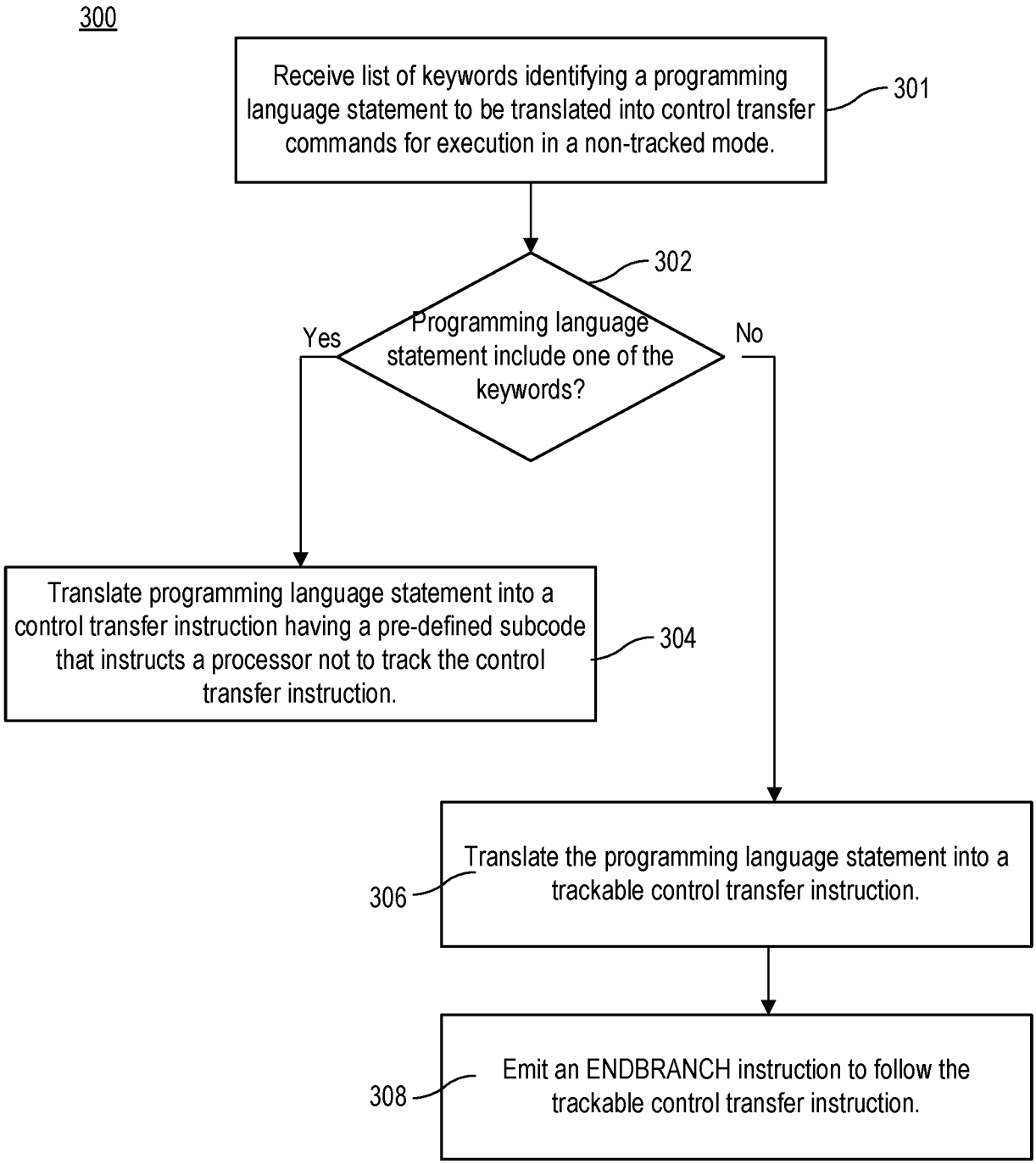
300
Receive list of keywords identifying a programming language statement to be translated into control transfer commands for execution in a non-tracked mode.
301
302
Programming language statement include one of the keywords?
Yes
No
Translate programming language statement into a control transfer instruction having a pre-defined subcode that instructs a processor not to track the control transfer instruction.
304
Translate the programming language statement into a trackable control transfer instruction.
306
Emit an ENDBRANCH instruction to follow the trackable control transfer instruction.
308

FIG. 3A

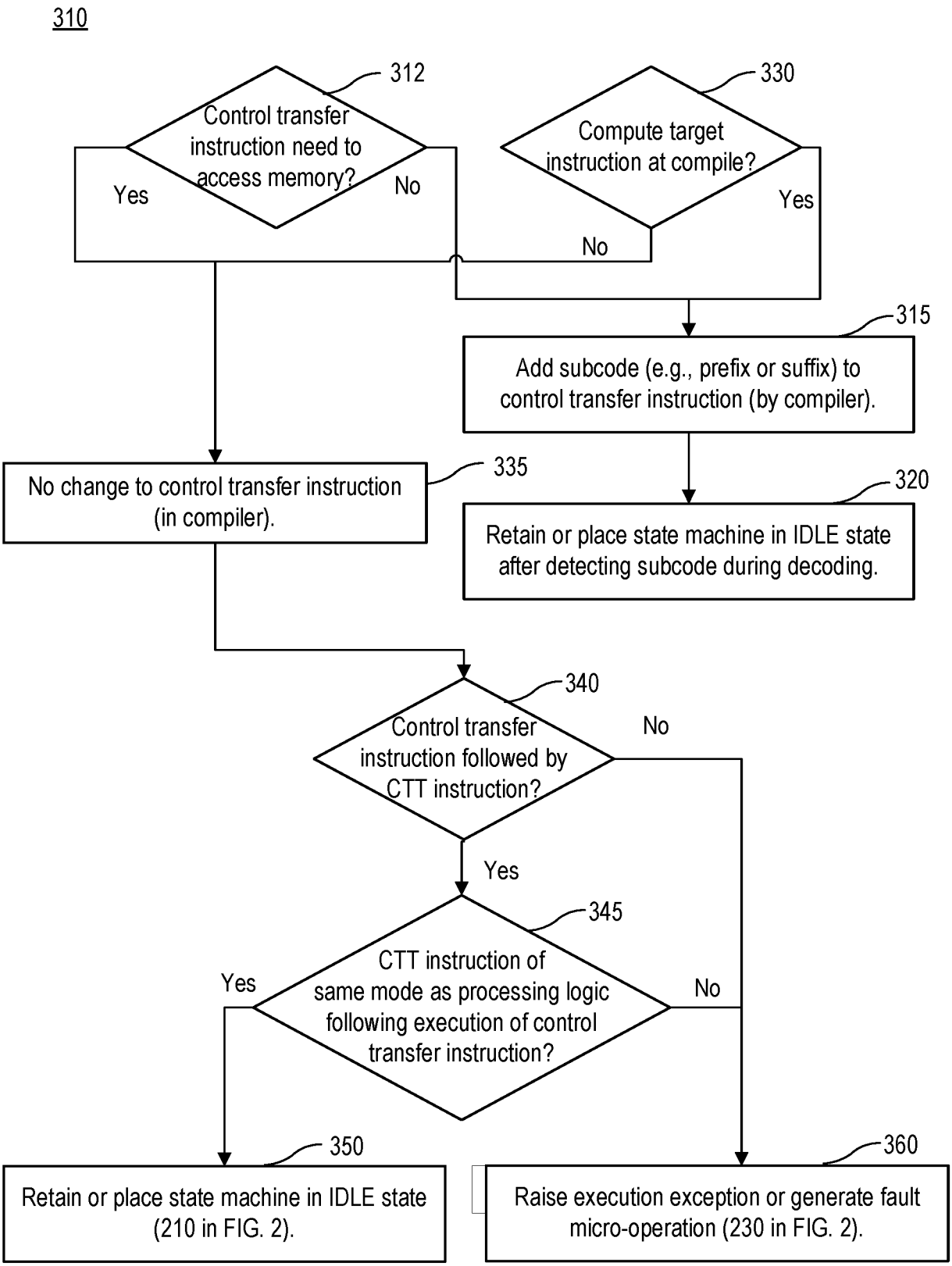
310
312
Control transfer instruction need to access memory?
Yes
No
330
Compute target instruction at compile?
Yes
No
315
Add subcode (e.g., prefix or suffix) to control transfer instruction (by compiler).
335
No change to control transfer instruction (in compiler).
320
Retain or place state machine in IDLE state after detecting subcode during decoding.
340
Control transfer instruction followed by CTT instruction?
No
Yes
345
CTT instruction of same mode as processing logic following execution of control transfer instruction?
Yes
No
350
Retain or place state machine in IDLE state (210 in FIG. 2).
360
Raise execution exception or generate fault micro-operation (230 in FIG. 2).

FIG. 3B

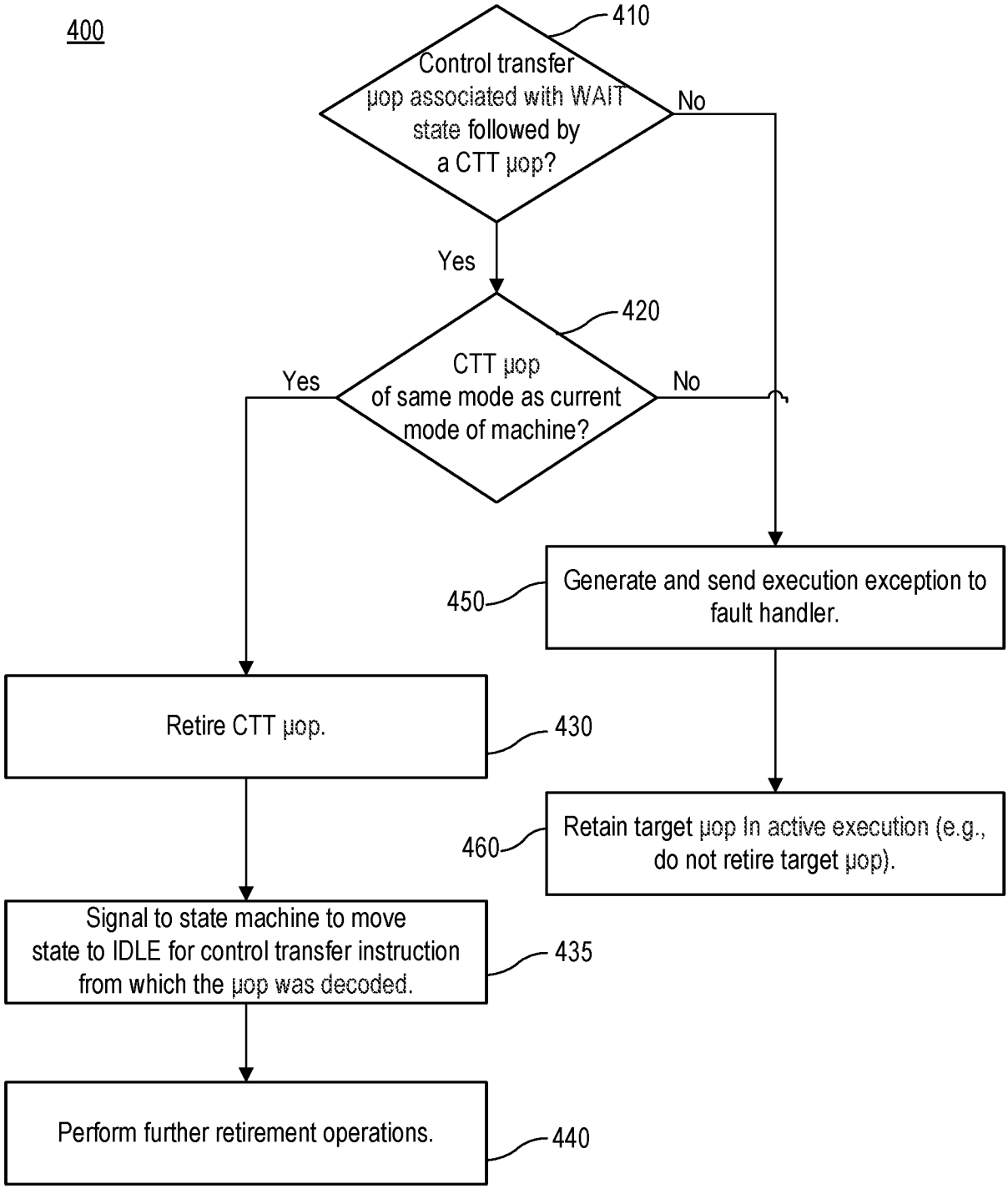
400
410
Control transfer µop associated with WAIT state followed by a CTT µop?
No
Yes
420
CTT µop of same mode as current mode of machine?
Yes
No
450
Generate and send execution exception to fault handler.
Retire CTT µop.
430
460
Retain target µop in active execution (e.g., do not retire target µop).
Signal to state machine to move state to IDLE for control transfer instruction from which the µop was decoded.
435
Perform further retirement operations.
440

FIG. 4

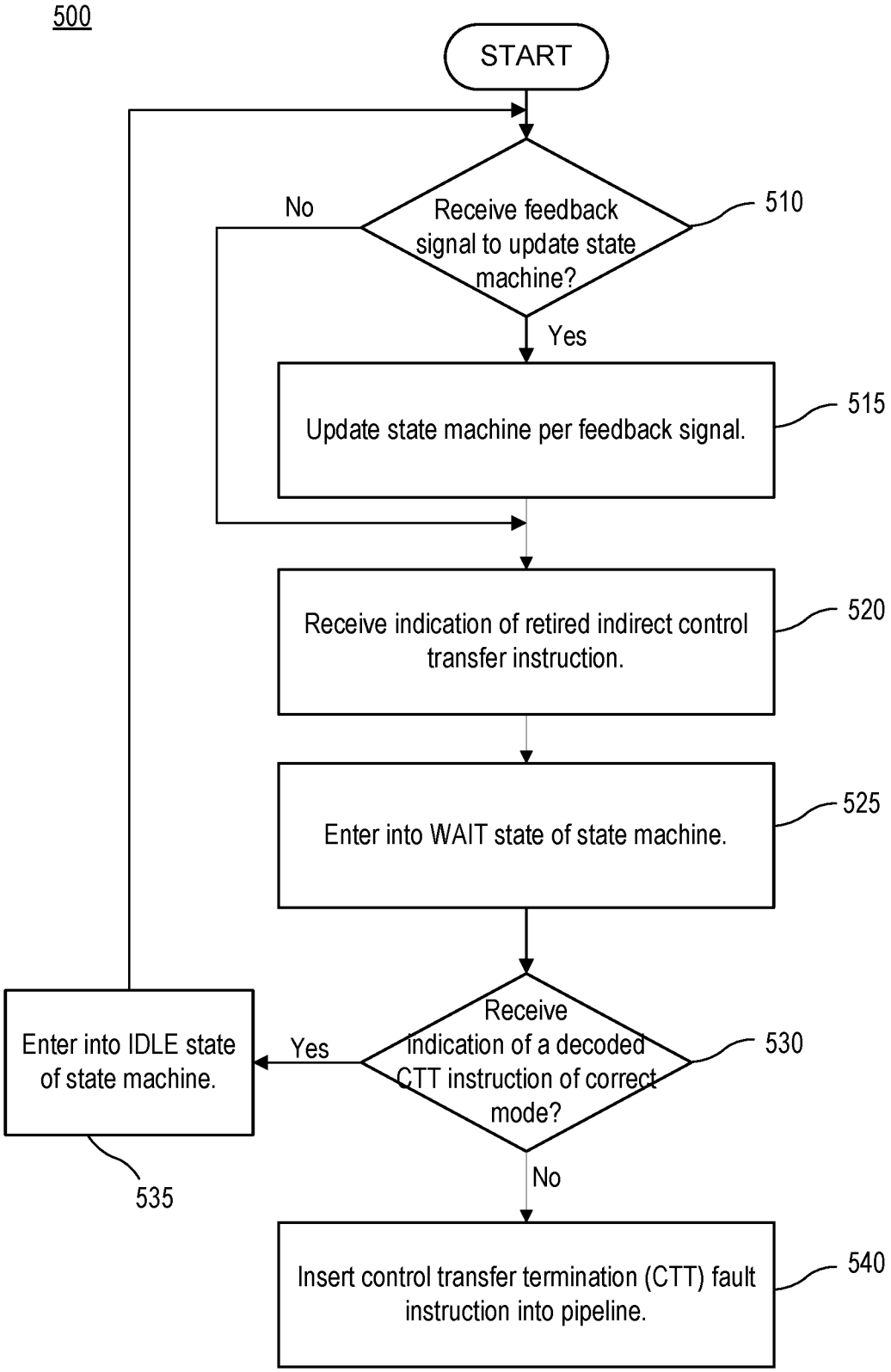
500
START
Receive feedback signal to update state machine?
510
No
Yes
Update state machine per feedback signal.
515
Receive indication of retired indirect control transfer instruction.
520
Enter into WAIT state of state machine.
525
Receive indication of a decoded CTT instruction of correct mode?
530
Yes
No
Enter into IDLE state of state machine.
535
Insert control transfer termination (CTT) fault instruction into pipeline.
540

FIG. 5A

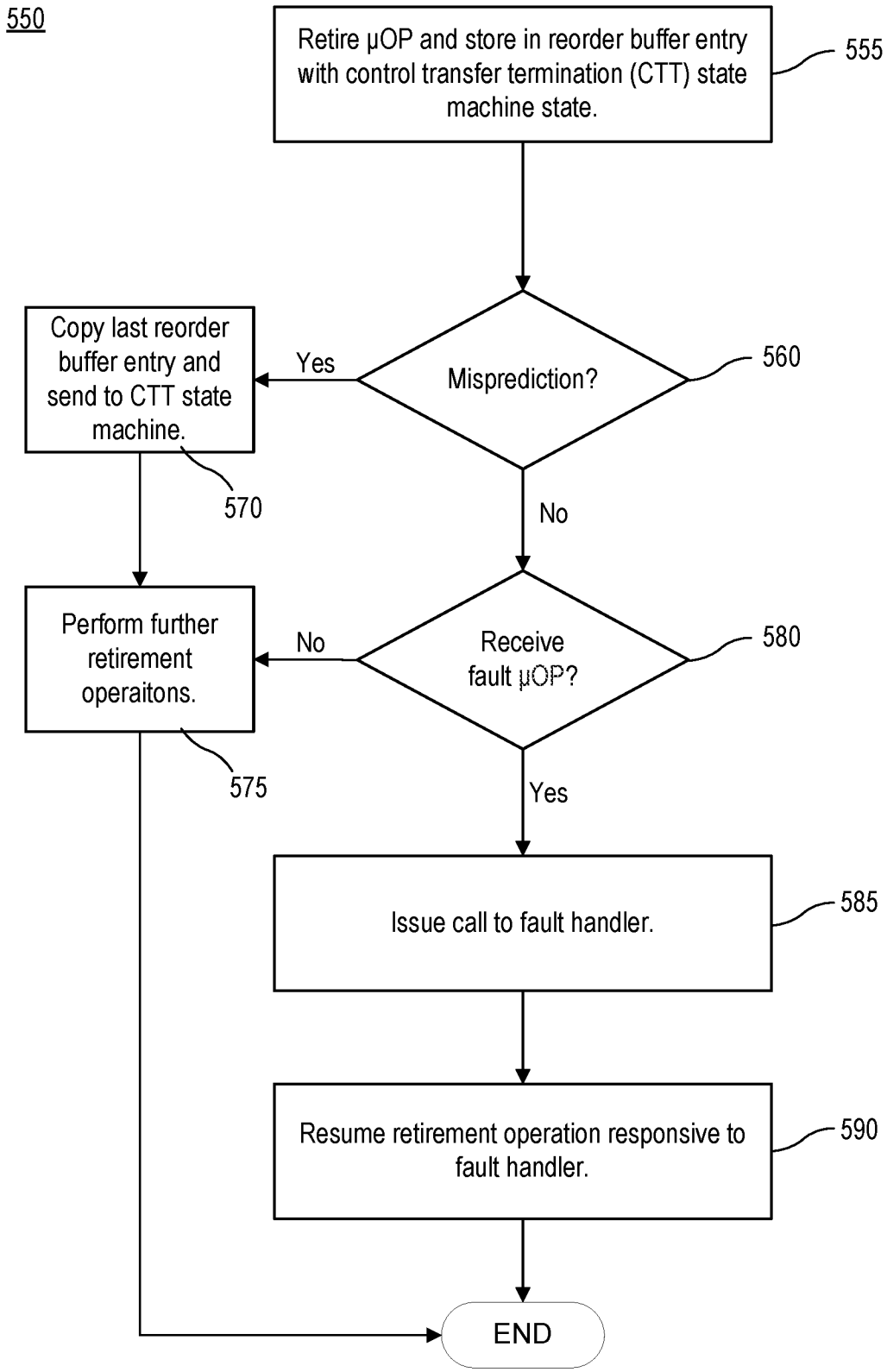
550
Retire μOP and store in reorder buffer entry with control transfer termination (CTT) state machine state.
555
Mispredict?
560
Yes
Copy last reorder buffer entry and send to CTT state machine.
570
No
Receive fault μOP?
580
No
Perform further retirement operaitons.
575
Yes
Issue call to fault handler.
585
Resume retirement operation responsive to fault handler.
590
END

FIG. 5B

Rsvd 625	Tracker 620	Supp 615	IW En 610	En 605

FIG. 6

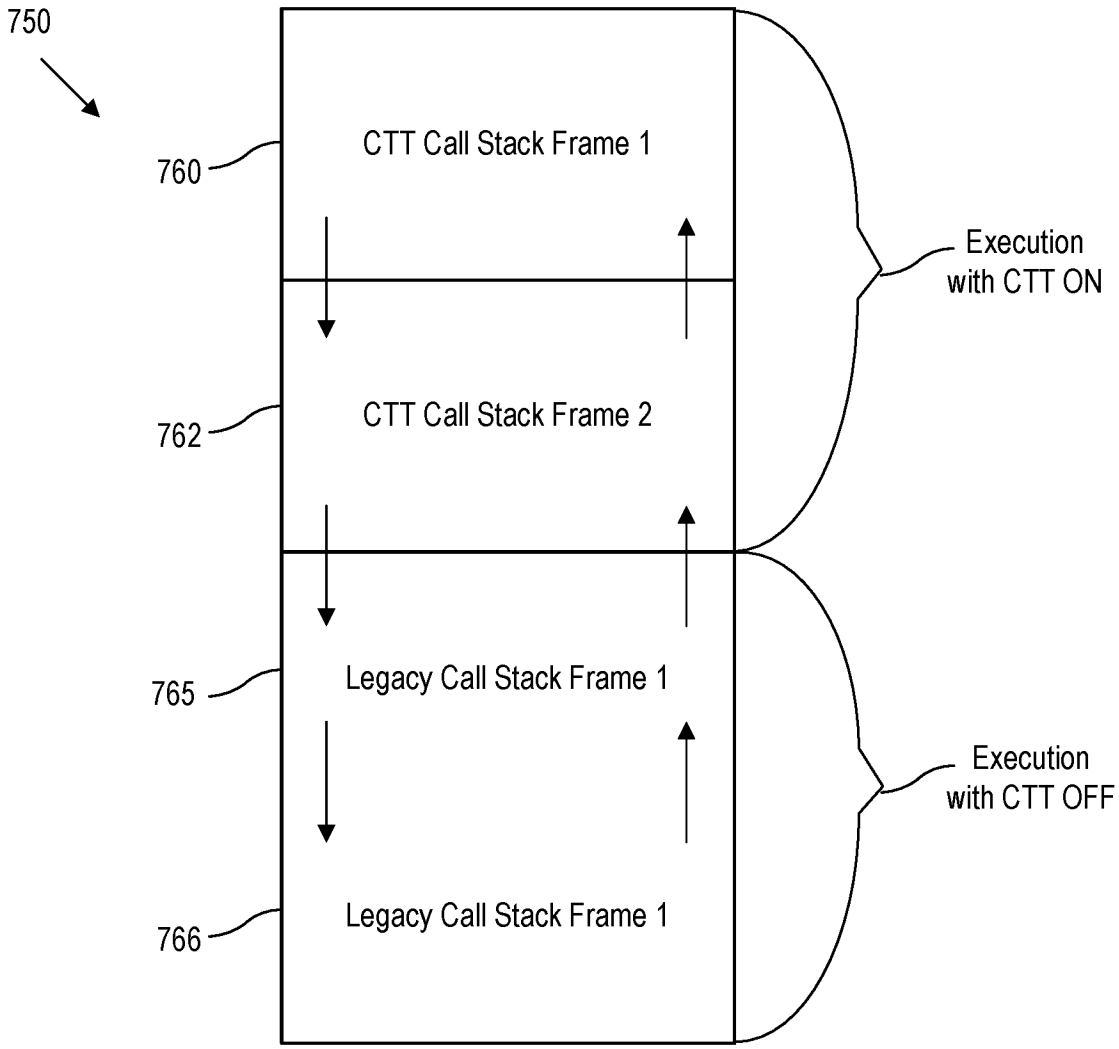

FIG. 7A

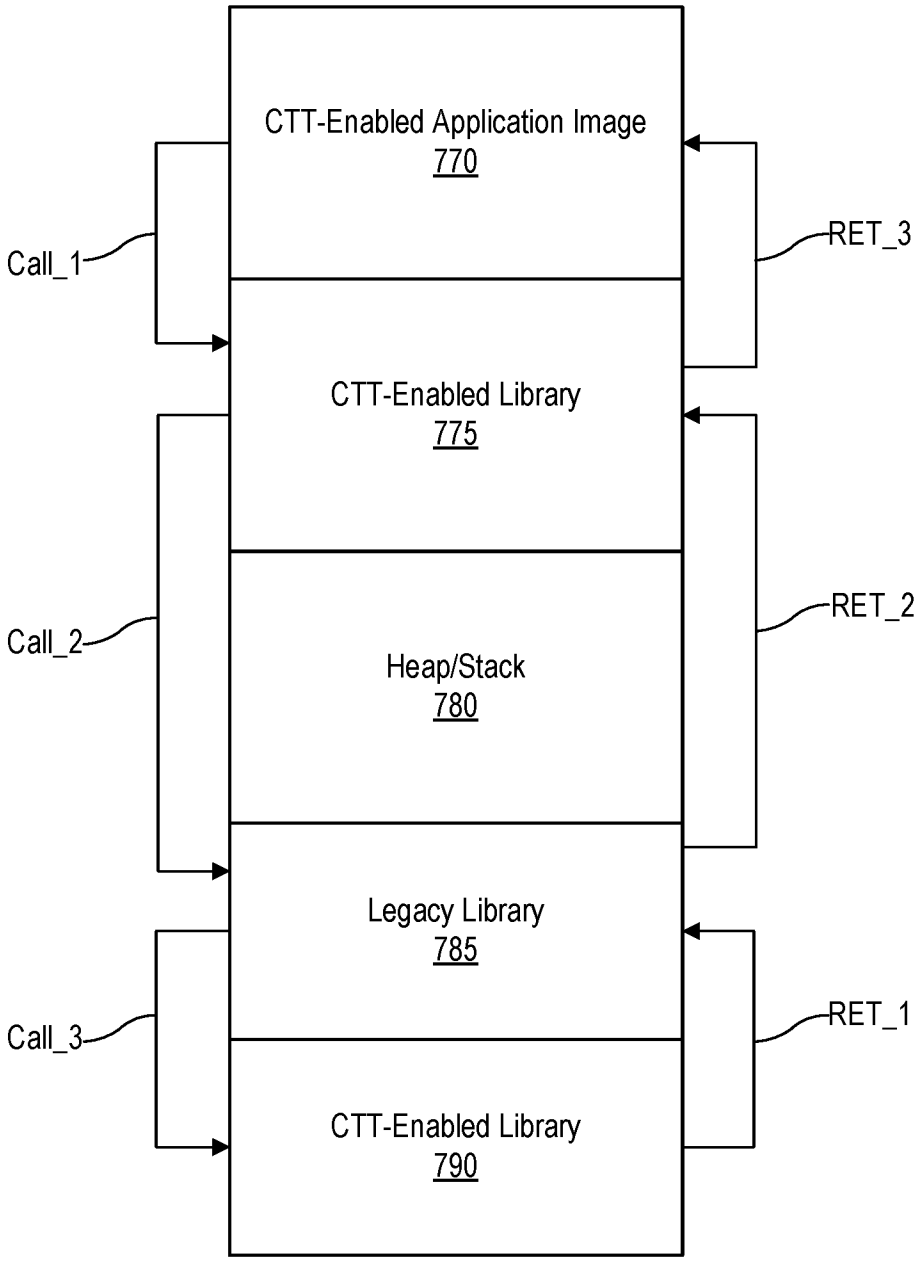
CTT-Enabled Application Image
770
CTT-Enabled Library
775
Heap/Stack
780
Legacy Library
785
CTT-Enabled Library
790
Call_1
Call_2
Call_3
RET_3
RET_2
RET_1

FIG. 7B

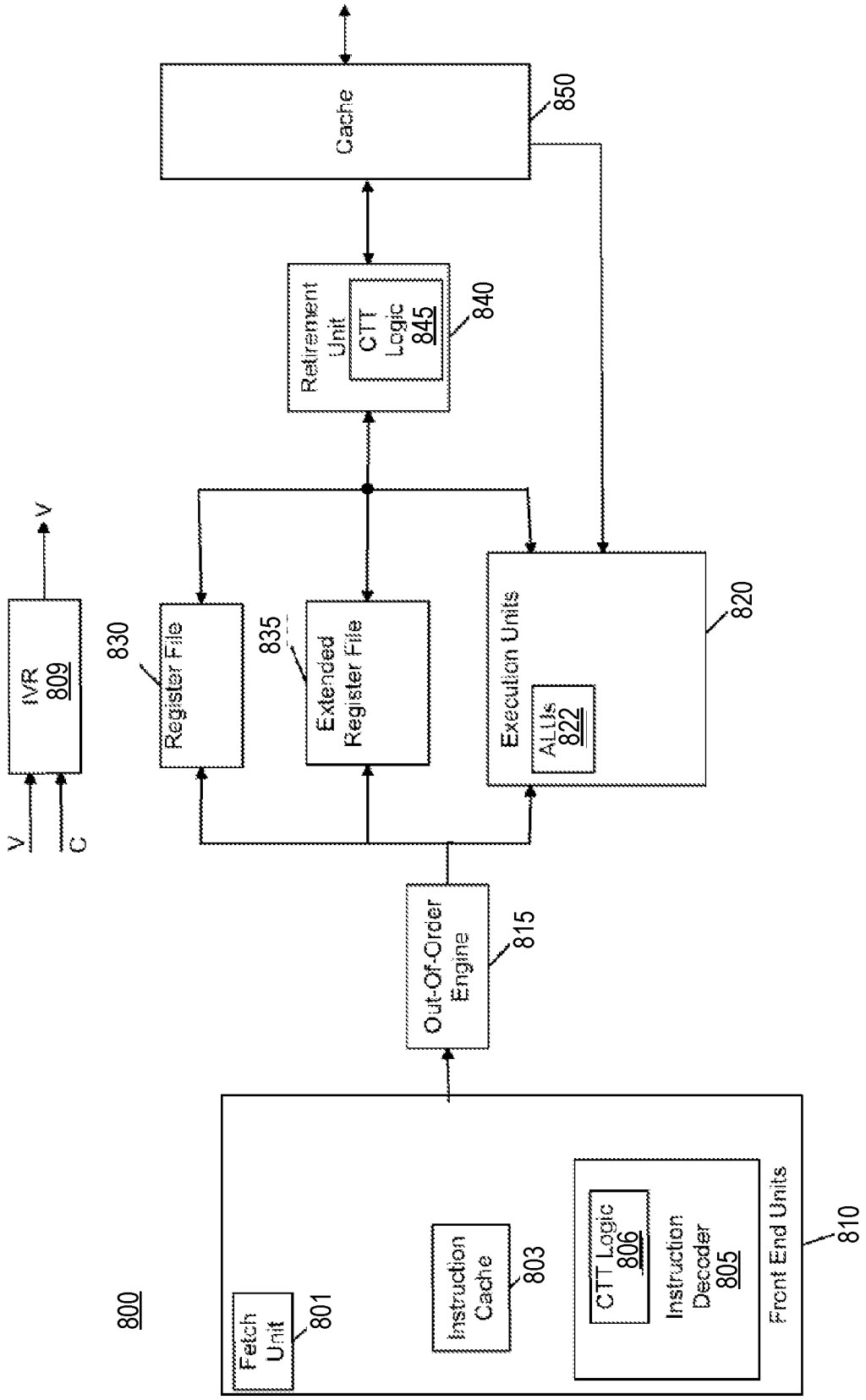
800
Fetch Unit
801
Instruction Cache
803
CTT Logic
806
Instruction Decoder
805
Front End Units
810
Out-Of-Order Engine
815
V
C
IVR
809
V
830
Register File
835
Extended Register File
Execution Units
ALUs
822
820
Retirement Unit
CTT Logic
845
840
Cache
850

FIG. 8

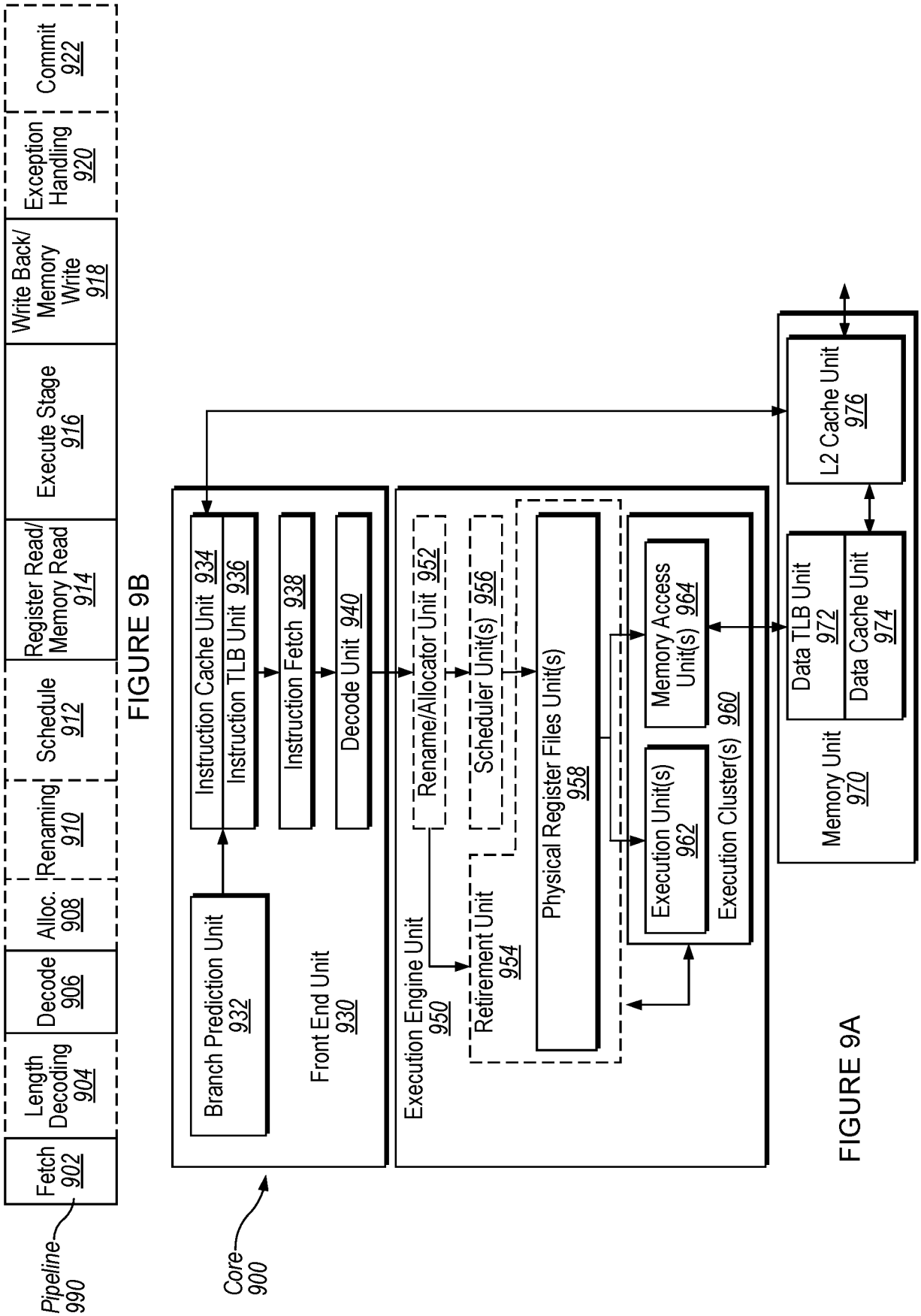

FIGURE 9B

FIGURE 9A

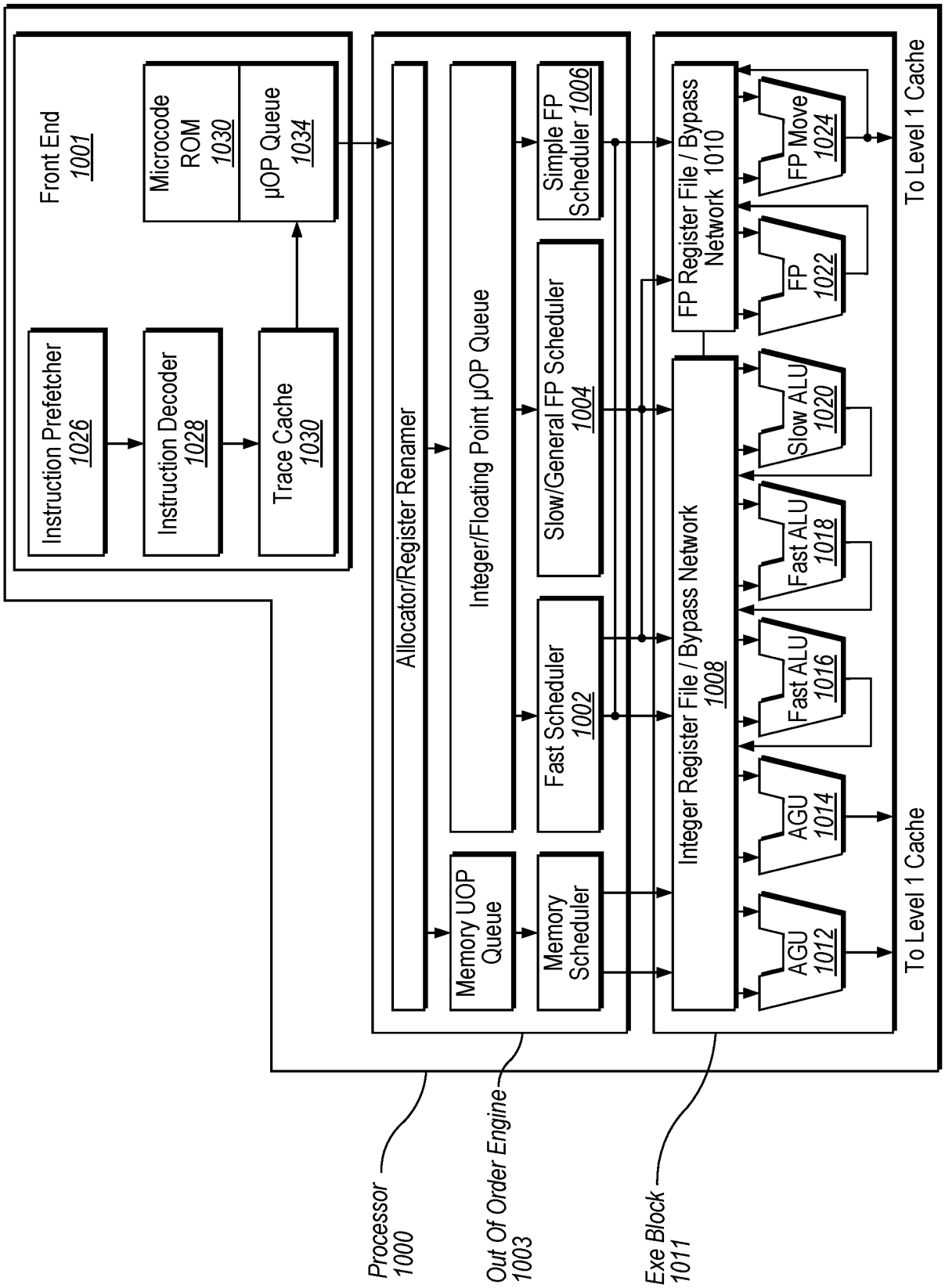
Front End
1001
Instruction Prefetcher
1026
Instruction Decoder
1028
Trace Cache
1030
Microcode
ROM
1030
µOP Queue
1034
Allocator/Register Renamer
Memory UOP
Queue
Memory
Scheduler
Integer/Floating Point µOP Queue
Fast Scheduler
1002
Slow/General FP Scheduler
1004
Simple FP
Scheduler 1006
Integer Register File / Bypass Network
1008
FP Register File / Bypass
Network 1010
AGU
1012
AGU
1014
Fast ALU
1016
Fast ALU
1018
Slow ALU
1020
FP
1022
FP Move
1024
To Level 1 Cache
To Level 1 Cache
Processor
1000
Out Of Order Engine
1003
Exe Block
1011

FIG. 10

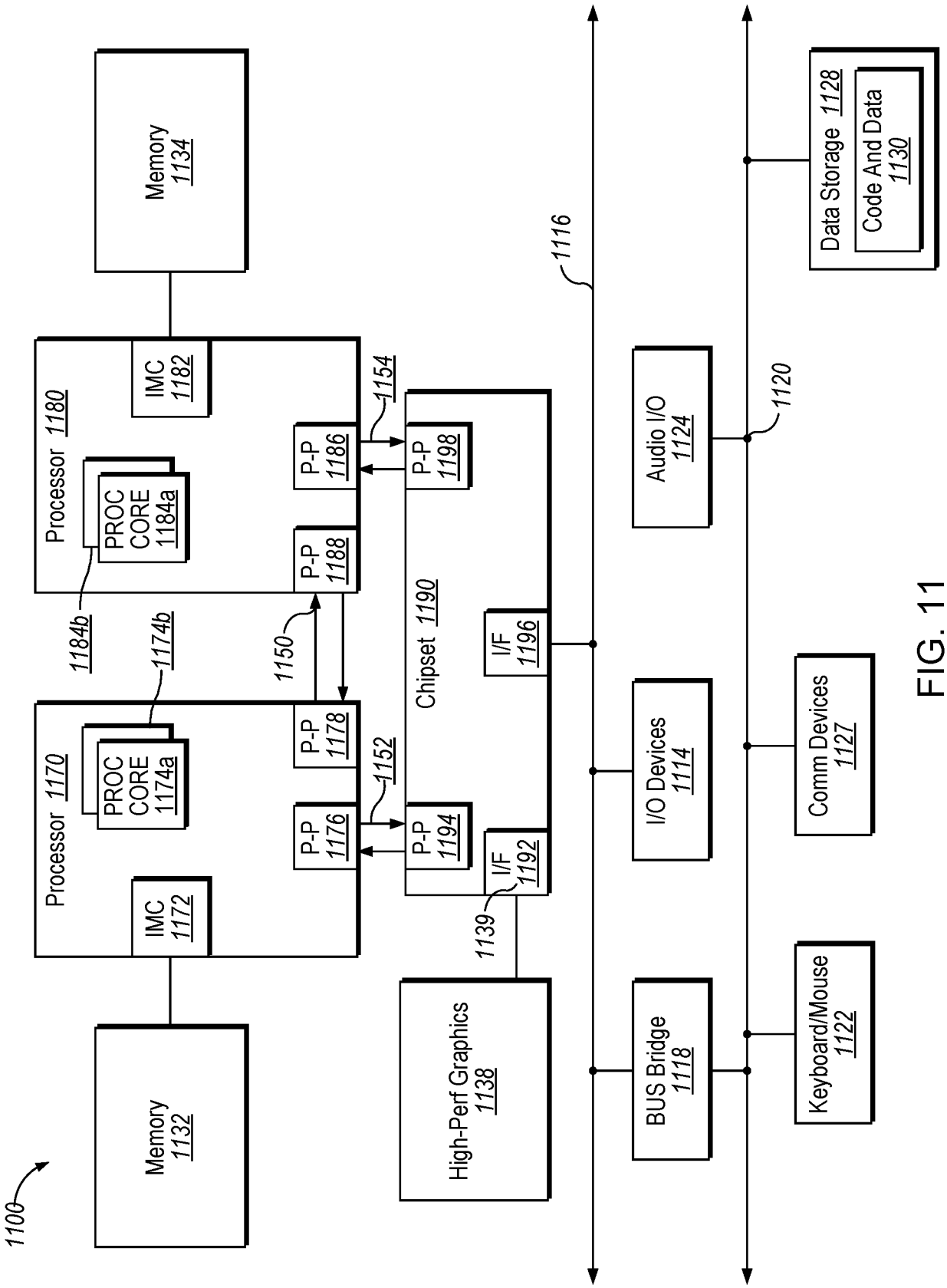
1100
Memory 1132
Memory 1134
Processor 1170
IMC 1172
PROC CORE 1174a
1174b
Processor 1180
IMC 1182
PROC CORE 1184a
1184b
P-P 1176
P-P 1178
P-P 1186
P-P 1188
1150
1152
1154
Chipset 1190
P-P 1194
P-P 1198
I/F 1192
I/F 1196
1139
High-Perf Graphics 1138
1116
BUS Bridge 1118
I/O Devices 1114
Audio I/O 1124
1120
Keyboard/Mouse 1122
Comm Devices 1127
Data Storage 1128
Code And Data 1130

FIG. 11

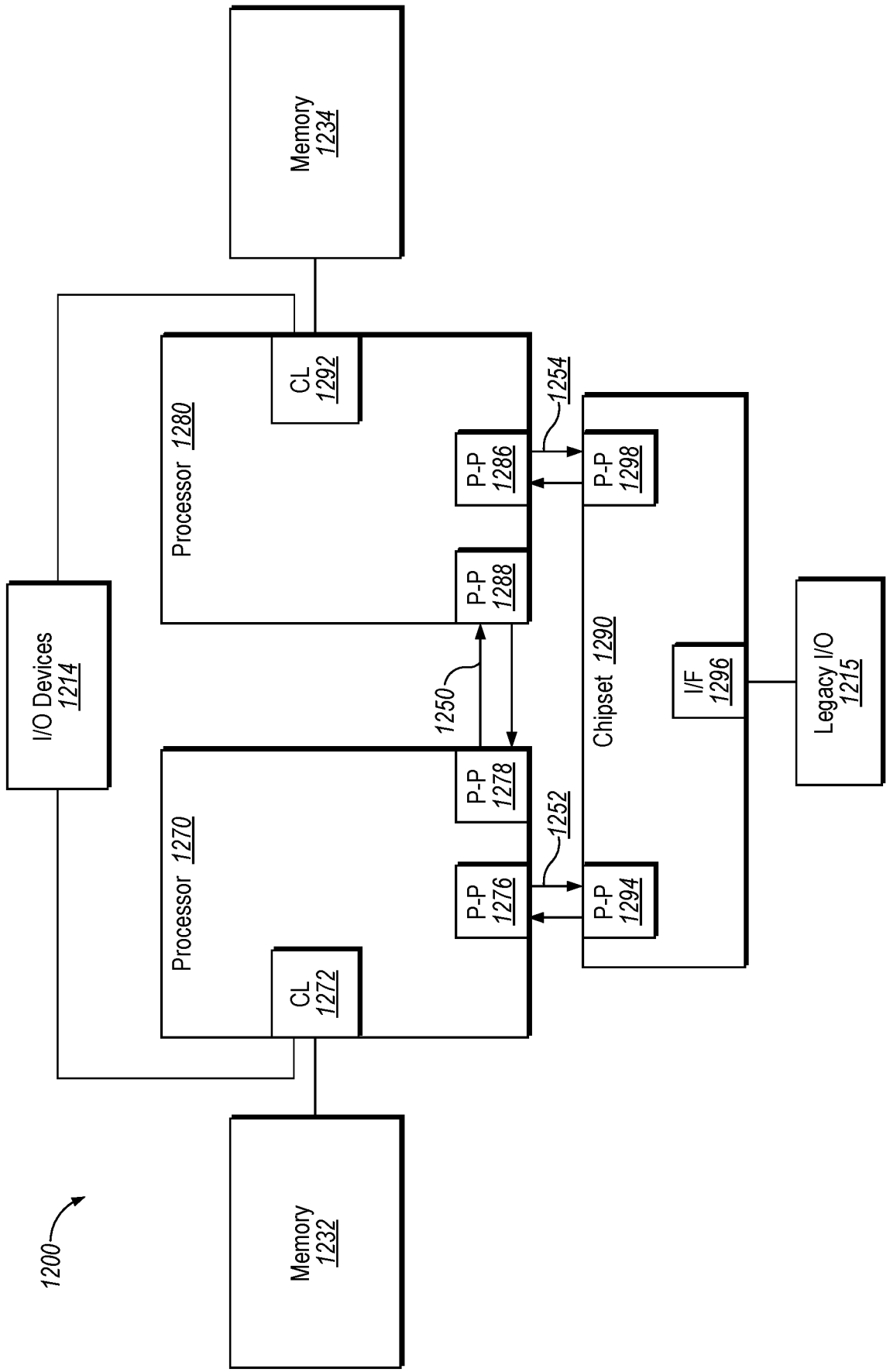
1200
Memory 1232
Processor 1270
CL 1272
P-P 1276
P-P 1278
1252
1250
I/O Devices 1214
Processor 1280
CL 1292
P-P 1286
P-P 1288
1254
Chipset 1290
P-P 1294
P-P 1298
I/F 1296
Legacy I/O 1215
Memory 1234

FIG. 12

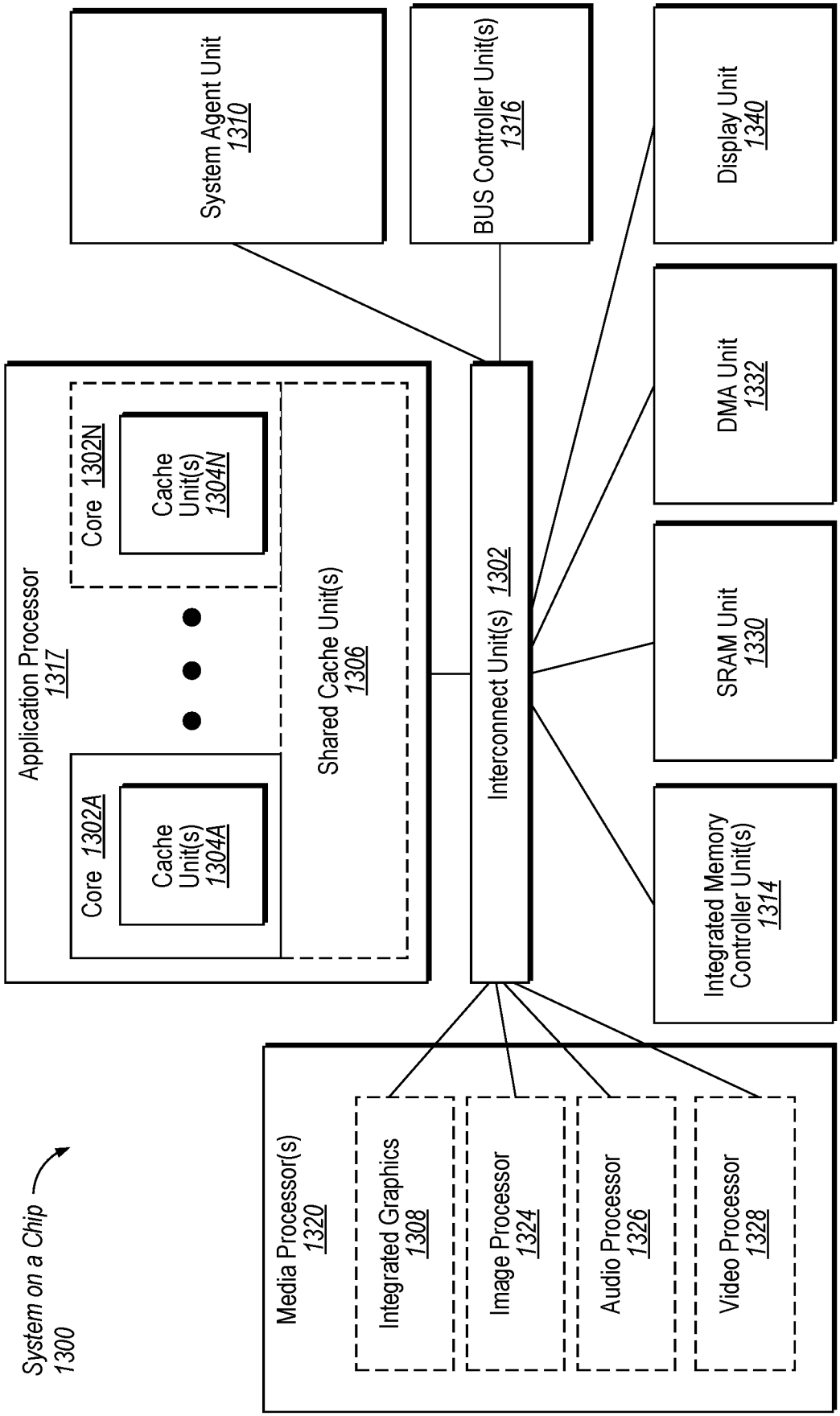
System on a Chip
1300
Application Processor
1317
Core 1302A
Cache Unit(s) 1304A
Core 1302N
Cache Unit(s) 1304N
Shared Cache Unit(s)
1306
System Agent Unit
1310
BUS Controller Unit(s)
1316
Interconnect Unit(s) 1302
Display Unit
1340
DMA Unit
1332
SRAM Unit
1330
Integrated Memory Controller Unit(s)
1314
Media Processor(s)
1320
Integrated Graphics
1308
Image Processor
1324
Audio Processor
1326
Video Processor
1328

FIG. 13

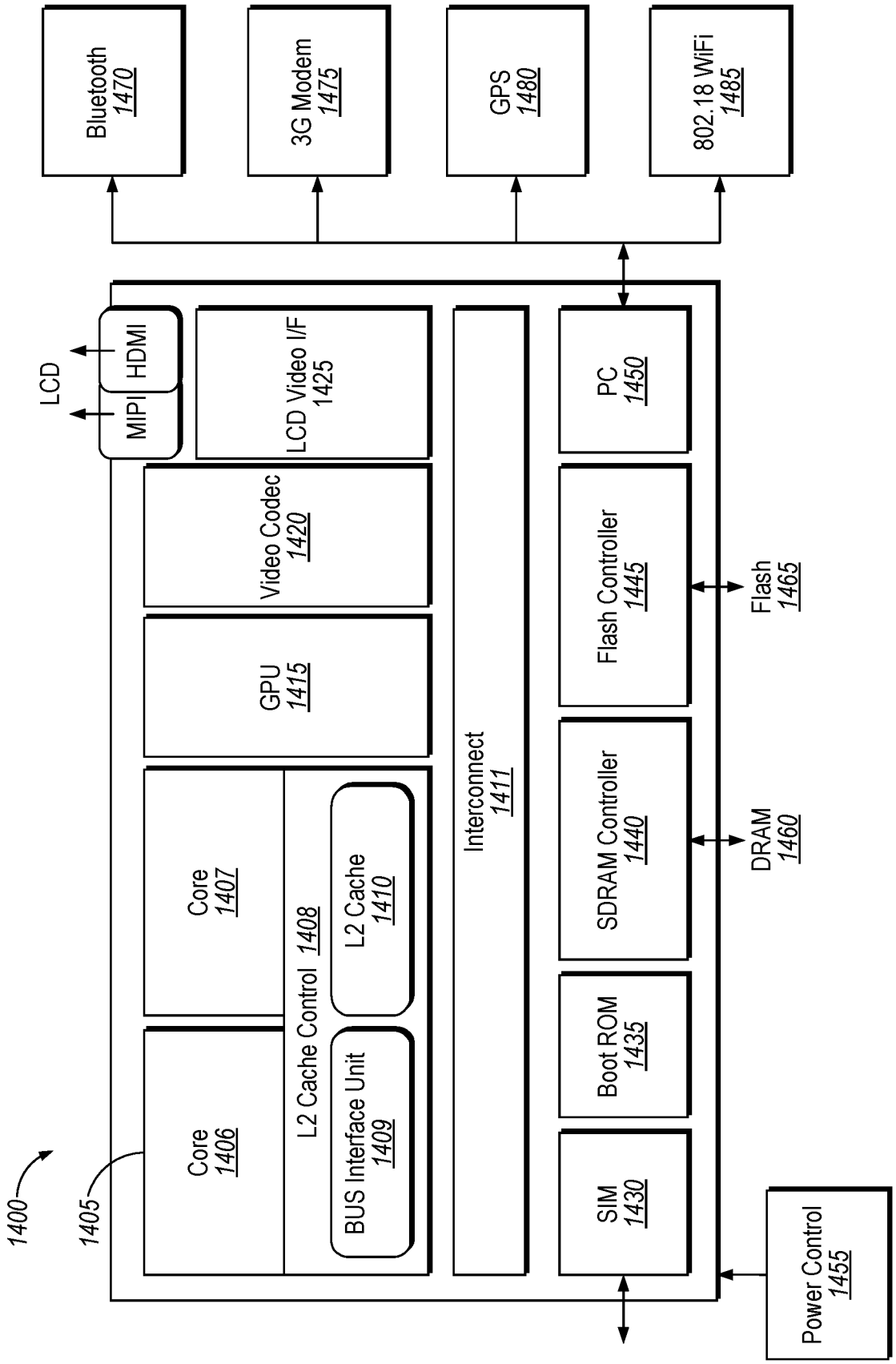
1400
1405
Core 1406
Core 1407
L2 Cache Control 1408
BUS Interface Unit 1409
L2 Cache 1410
GPU 1415
Video Codec 1420
LCD Video I/F 1425
LCD
MIPI
HDMI
Interconnect 1411
SIM 1430
Boot ROM 1435
SDRAM Controller 1440
Flash Controller 1445
PC 1450
DRAM 1460
Flash 1465
Power Control 1455
Bluetooth 1470
3G Modem 1475
GPS 1480
802.18 WiFi 1485

FIG. 14

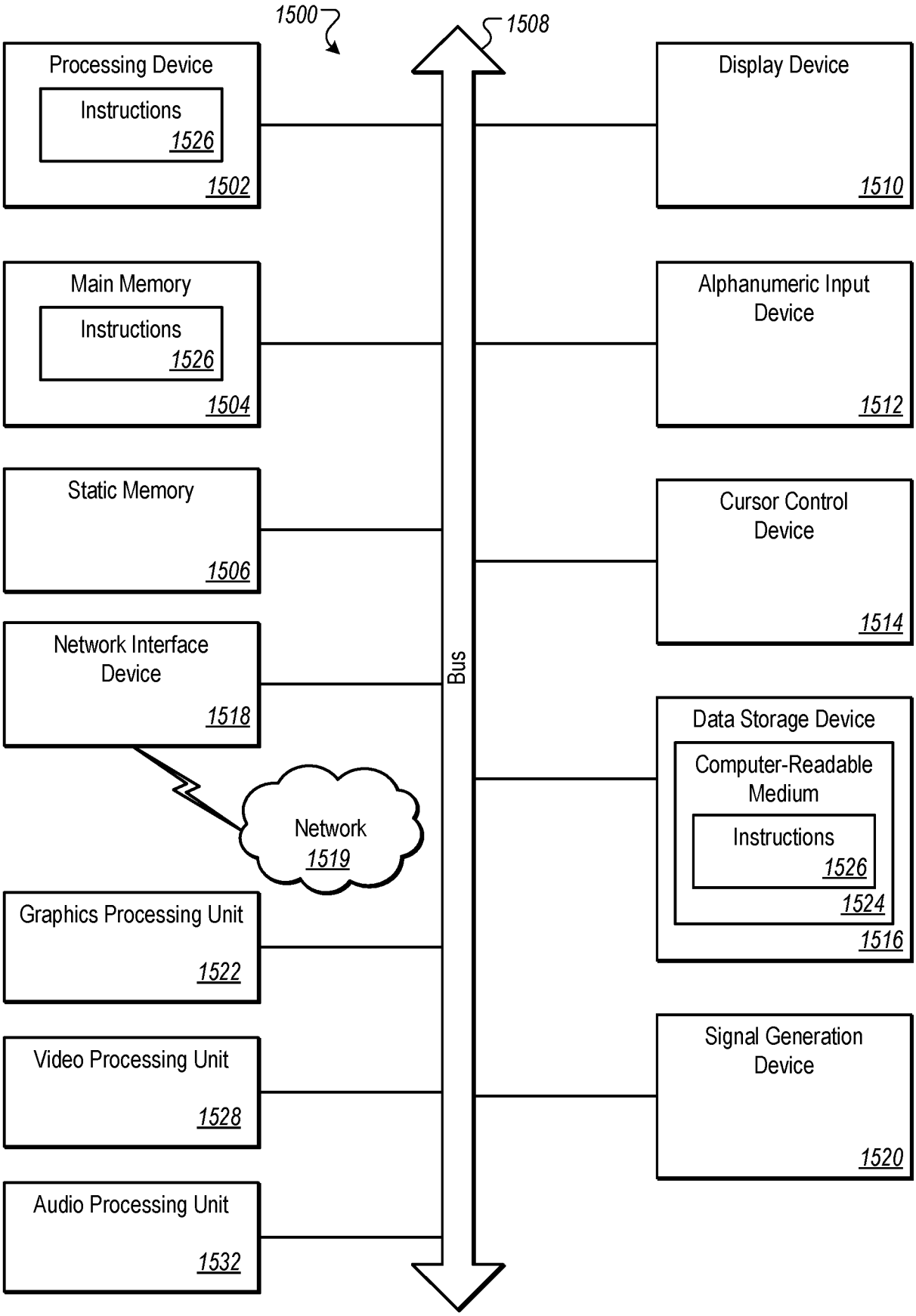
1500
1508
Processing Device
Instructions
1526
1502
Main Memory
Instructions
1526
1504
Static Memory
1506
Network Interface Device
1518
Network
1519
Graphics Processing Unit
1522
Video Processing Unit
1528
Audio Processing Unit
1532
Bus
Display Device
1510
Alphanumeric Input Device
1512
Cursor Control Device
1514
Data Storage Device
Computer-Readable Medium
Instructions
1526
1524
1516
Signal Generation Device
1520

FIG. 15

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/063665

A. CLASSIFICATION OF SUBJECT MATTER

G06F 21/54(2013.01)i, G06F 21/50(2013.01)i

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 21/54; G06F 9/30; G06F 9/44; G06F 9/38; G06F 9/455; G06F 21/50

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: execution state, pre-defined subcode, no operation, idle, wait, exception, fault, retire

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2014-0156972 A1 (VEDYVAS SHANBHOGUE et al.) 05 June 2014 See paragraphs [0007], [0026], [0029]-[0030], [0032]-[0033], [0040], [0042]-[0043]; claims 1-3, 11; and figures 1-2, 3A.	1-22
Y	US 2014-0095833 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 03 April 2014 See paragraphs [0132], [0301]; claim 1; and figure 5.	1-22
Y	US 2013-0111190 A1 (ADAM J. MUFF et al.) 02 May 2013 See paragraph [0080]; and figure 5.	15-16
A	US 7340589 B2 (MAKOTO KUDO) 04 March 2008 See column 7, line 54 - column 8, line 58; and figure 3.	1-22
A	US 8086438 B2 (ACHIM NOHL et al.) 27 December 2011 See column 7, line 27 - column 8, line 22; and figure 4.	1-22

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
20 March 2017 (20.03.2017)	**20 March 2017 (20.03.2017)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea
Facsimile No. +82-42-481-8578

Authorized officer
CHIN, Sang Bum
Telephone No. +82-42-481-8398

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2016/063665

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2014-0156972 A1	05/06/2014	None	
US 2014-0095833 A1	03/04/2014	DE 102012216592 A1	04/04/2013
		GB 2495363 A	10/04/2013
		GB 2495363 A8	24/04/2013
		GB 2495363 B	04/12/2013
		US 2013-0086362 A1	04/04/2013
		US 9311093 B2	12/04/2016
		US 9329869 B2	03/05/2016
US 2013-0111190 A1	02/05/2013	US 8892851 B2	18/11/2014
US 7340589 B2	04/03/2008	CN 1270232 C	16/08/2006
		CN 1490726 A	21/04/2004
		CN 1490726 C	16/08/2006
		JP 2004-030015 A	29/01/2004
		JP 3627725 B2	09/03/2005
		US 2004-0039897 A1	26/02/2004
US 8086438 B2	27/12/2011	EP 1349063 A2	01/10/2003
		EP 1349063 A3	08/12/2004
		US 2003-0217248 A1	20/11/2003
		US 2012-0158397 A1	21/06/2012
		US 8554535 B2	08/10/2013