

(51) International Patent Classification:
G06F 9/45 (2006.01)(21) International Application Number:
PCT/CN2012/081728(22) International Filing Date:
21 September 2012 (21.09.2012)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant (for all designated States except US): **INTEL CORPORATION** [US/US]; 2200 Mission College Blvd. Santa Clara, California 95052 (US).

(72) Inventors; and

(71) Applicants (for US only): **ZHONG, Xueliang** [CN/CN]; No.880 Zi Xing Road, Shanghai Zizhu Science Park, Shanghai 200241 (CN). **LI, Jianhui** [CN/CN]; No.880 Zi Xing Road, Shanghai Zizhu Science Park, Shanghai 200241 (CN). **CHEN, JianPing, Jane** [CN/CN]; No.880 Zi Xing Road, Shanghai Zizhu Science Park, Shanghai 200241 (CN). **WANG, Gang** [CN/CN]; Room 3A, No.125 Lane 2518 Longhua Rd., Shengda Garden, Shanghai 200232 (CN). **QIAN, Yi** [CN/CN]; Room 4032, No.67, 800# Dongchuan Rd., Minhang District, Shanghai 200240 (CN). **GU, Huifeng** [CN/CN]; Room 603, No.9,

199# Mincheng Rd., Minhang district, Shanghai 201100 (CN).

(74) Agent: **CHINA PATENT AGENT (HK) LTD**; 22/F., Great Eagle Center, 23 Harbour Road, Wanchai, Hong Kong (CN).

(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, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, 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, 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,

[Continued on next page]

(54) Title: METHODS AND SYSTEMS FOR PERFORMING A BINARY TRANSLATION

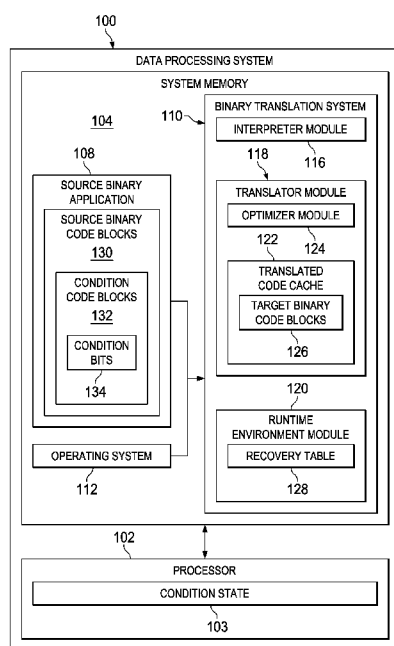


FIG. 1

(57) Abstract: Systems and methods are provided in example embodiments for performing binary translation. A binary translation system converts, by a translator module, source instructions to target instructions. The binary translation system identifies a condition code block in the source instructions, where the condition code block includes a plurality of condition bits. In response to identifying the condition code block, the binary translation system provides an optimizer module to convert the condition code block. Then, the binary translation system performs a pre-execution on the condition code block to resolve the plurality of condition bits in the condition code block.

WO 2014/043886 A1



TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, **Published:**
ML, MR, NE, SN, TD, TG).

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

METHODS AND SYSTEMS FOR PERFORMING A BINARY TRANSLATION

TECHNICAL FIELD

[0001] This disclosure relates in general to the field of binary translation and, more particularly, to executing instructions involving binary translation.

BACKGROUND

[0002] Binary translation is a high performance emulator that allows an application, compiled for one instruction set architecture (referred to herein as a "source ISA") to run on a different ISA (referred to herein as a "target ISA"). Essentially, binary translation (BT) is a general technique to translate binaries built for one source ISA to another target ISA. Accordingly, by using binary translation, it is possible to execute application binaries built for one processor ISA on a processor with a different architecture, without recompiling high-level source code or rewriting assembly code. Since many legacy computer applications are available in binary formats, binary translation is very attractive due to its potential to allow a processor to execute applications that are not built and available for it.

[0003] Generally, binary translation systems follow the same basic execution flow. First, a binary file created for the source architecture is loaded into memory. Sections of this source binary are then translated into target binary code. This translation is typically done "on-demand". In other words, the source-code instructions are translated as the flow of control reaches them. Typically, the translation is performed at the granularity of basic blocks, which are sequences of instructions with a single entry and potentially multiple exits. Once a basic block is translated, the translation is kept in a translation cache (also referred to herein as "code cache") in memory for future reuse.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] To provide a more complete understanding of the present disclosure and features and advantages thereof, reference is made to the following description, taken in conjunction with the accompanying figures, wherein like reference numerals represent like parts, in which:

[0005] FIGURE 1 is an example illustration of a data processing system in accordance with an embodiment;

[0006] FIGURE 2 is an example illustration of user equipment in accordance with an embodiment;

[0007] FIGURE 3 is an example illustration of a processor according to one embodiment;

[0008] FIGURE 4 is an illustration of an execution order of instructions of a source instruction set architecture in accordance with an embodiment;

[0009] FIGURE 5 is an illustration of a binary translation system with an optimizer module in accordance with an embodiment;

[0010] FIGURE 6 is a simplified flowchart illustrating a method for binary translation with an optimizer module in accordance with an embodiment;

[0011] FIGURE 7 is an illustration of a binary translation system with a recovery table in accordance with an embodiment;

[0012] FIGURE 8 is a simplified flowchart illustrating a method for binary translation by a binary translation system with a condition state recovery table in accordance with an embodiment;

[0013] FIGURE 9 is a simplified flowchart illustrating a method for advancing the condition state using the recovery table in accordance with an embodiment;

[0014] FIGURE 10A is a simplified flowchart illustrating a method for identifying a parallel-multiply-addition instruction in accordance with an embodiment;

[0015] FIGURE 10B is a simplified flowchart illustrating a method for binary translation with a parallel-multiply-addition instruction in accordance with an embodiment;

[0016] FIGURE 11A is a simplified flowchart illustrating a method for identifying a parallel-multiply-subtraction instruction in accordance with an embodiment;

[0017] FIGURE 11B is a simplified flowchart illustrating a method for binary translation with a parallel-multiply-subtraction instruction in accordance with an embodiment;

[0018] FIGURE 12A is a simplified flowchart illustrating a method for identifying a partial-long-multiply instruction in accordance with an embodiment; and

[0019] FIGURE 12B is a simplified flowchart illustrating a method for binary translation with a partial-long-multiply instruction in accordance with an embodiment.

DETAILED DESCRIPTION OF EXAMPLE EMBODIMENTS

EXAMPLE EMBODIMENTS

[0020] FIGURE 1 is an example illustration of a data processing system in accordance with an embodiment. Data processing system 100 includes a processor 102 coupled to a system memory 104. Processor 102 may include a condition state 103. System memory 104 may include a source binary application 108, binary translation system 110, and an operating system 114. Binary translation system 110 may include an interpreter module 116, a translator module 118, and a runtime environment module 120, which can be implemented in software, hardware, firmware, or any suitable combination thereof. Translator module 118 may include a translated code cache 122 and an optimizer module 124. Additionally, translated code cache 122 may include target binary code blocks 126. Runtime environment module 120 may include a recovery table 128. Source binary application 108 may include source binary code blocks 130. Source binary code blocks 130 may include condition code blocks 132. Further, condition code blocks 132 may include condition bits 134. For ease of reference, source binary code blocks 130, target binary code blocks 126, and condition code blocks 132 may be referred to herein in singular form, when appropriate for explanation.

[0021] Processor 102 serves to execute instructions for software that may be loaded into system memory 104. Processor 102 may be a number of processors, a multi-processor core, or some other type of processor, depending on the particular implementation. A number, as used herein with reference to an item, means one or more items. Further, processor 102 may be implemented using a number of heterogeneous processor systems in which a main processor is present with secondary processors on a single chip. As another illustrative example, processor 102 may be a symmetric multi-processor system containing multiple processors of the same type. The translator module 118 may be executed by one or more cores of processor 102 to transform source binary code blocks 130 into target binary code blocks 126.

[0022] System memory 104 is an example of a storage device. A storage device is any piece of hardware that is capable of storing information such as, for example and without limitation, data, program code in functional form, and/or other suitable information either on a temporary basis and/or a permanent basis. Storage devices may also be referred to as computer readable storage devices in these examples. System memory 104 could include, for example, a random access memory or any other suitable volatile or non-volatile storage device. System memory 104 could also include persistent storage, which may take various forms, depending on the particular implementation.

[0023] Source binary application 108 may include source binary code blocks 130 corresponding to a source ISA. Likewise, target binary code blocks 126 correspond to a target ISA. Source binary code blocks 130 may be a sequence of one or more instructions that may, for example, end with a branch instruction. Target binary code blocks 126 are configured to include functionality of a corresponding source binary code block.

[0024] Optimizer module 124 may be configured to optimize portions of source binary code blocks 130, for example, condition code blocks 132. The optimized portions may be referred to as target binary code blocks 126. Target binary code blocks 126 may include functionality of source binary code blocks 130 in a form of target binary code blocks 126. The optimized target binary code block may include additional functionality, as described herein.

[0025] Target binary code blocks 126 may be stored in an area of system memory designated as a translated code cache 122. More particularly, one or more target binary code blocks 126 that have been translated from one or more corresponding source binary code blocks 130 may be stored in translated code cache 122.

[0026] Source binary code block 130 may be translated into target binary code block 126. In some embodiments, a single source binary code block 130 is

translated in a single target binary code block 126. However, in other embodiments, there may be any number of source binary code blocks 130 translated into any number of target binary code blocks 126. In other words, the source binary code block may be translated into the target binary code block in response to a call to execute the source binary code block rather than statically, i.e., prior to run time. Target binary code block 126 may then be stored in system memory 104 in translated code cache 122 and provided to processor 102 for execution.

[0027] A condition code block allows a developer to encode multiple condition bits 134 of multiple instructions into one conditions-specifying instruction, and condition bits 134 of following instructions can be dynamically resolved. Thus, condition code blocks 132 enable a compiler to generate more compact code. Condition code blocks 132 provide a technique for embedded system ISAs and are implemented as a compact form of condition bits 134.

[0028] Condition bits (CB) 134 are used in ISA to allow software developers to set certain bits of an instruction as condition state 103. At runtime, a processor may compare a current condition flag register with condition bits 134 of a current instruction. Then, the processor may dynamically decide either to execute the current instruction or to treat it as a no operation instruction (NOP).

[0029] Condition state (CS) 103 is a word data maintained in processor 102 for dynamically resolving condition bits 134 of each instruction in a condition code block (e.g., of condition code blocks 132) and for detecting the end of the condition code block. Condition state 103 can first be initialized by decoding a conditions-specifying instruction of the condition code block, and can be advanced, for example, by bit-mask operations, after executing each instruction in the condition code block. Conditions-specifying instructions may include both advancing and resolving instructions.

[0030] For example, in a THUMB2 instruction set of ARM architecture (for example, a 32-bit reduced instruction set computer), condition code blocks 132 can be used to implement a high level if-then-else semantic. There may be an IT instruction (i.e.,

If-Then instruction) that groups multiple following instructions into one condition code block. An ARM processor maintains a word, referred to as "IT-state," for dynamically resolving condition bits 134 and for conditionally executing each instruction in the condition code block.

[0031] The embodiments of this disclosure enhance binary translation by introducing condition state recovery table 128 and optimizer module 124. With this enhancement, condition code blocks 132 emulated in binary translation system 110 can gain several benefits. First, it may be unnecessary to advance condition state 103 and resolve condition bits 134 in translated code. It may also be unnecessary to translate the conditions-specifying instruction of condition code blocks 132 into target binary code blocks 126 (e.g., IT instruction of THUMB2 ISA). Optimizer module 124 may decode multiple condition states on conditions-specifying instructions, remove the conditions-specifying instructions, and transform condition code blocks 132 into series of code, where each instruction in the code has encoded its own condition bit.

[0032] Furthermore, binary translation system 110 may have the ability to recover condition state 103 at any point of condition code blocks 132 being translated into target binary code blocks 126. Condition state recovery table 128 enables binary translation system 110 to recover condition states and to support precise exception. Condition state recovery table 128 can identify which range of target binary code blocks 126 can be mapped back to a certain instructions in condition code blocks 132.

[0033] FIGURE 2 is an example illustration of user equipment in accordance with an embodiment. In an aspect of this disclosure, user equipment 200 operates as a transmitter and a receiver of wireless communications signals. Specifically, in one example, user equipment 200 may be capable of both transmitting and receiving cellular network voice and data mobile services. Mobile services include such functionality as full Internet access, downloadable and streaming video content, as well as voice telephone communications.

[0034] User equipment 200 may correspond to a conventional wireless or cellular portable telephone, for example such a handset that is capable of receiving "3G", or "third generation" cellular services. In another example, user equipment 200 may be capable of transmitting and receiving "4G" mobile services as well, or any other mobile service.

[0035] Examples of devices that can correspond to user equipment 200 include cellular telephone handsets and smartphones, such as those capable of Internet access, email, and instant messaging communications, and portable video receiving and display devices, along with the capability of supporting telephone services. It is contemplated that those skilled in the art having reference to this specification will readily comprehend the nature of modern smartphone and telephone handset devices and systems suitable for implementation of the different aspects of this disclosure as described herein. As such, the architecture of user equipment 200 illustrated in FIGURE 2 is presented at a relatively high level; even so, it is contemplated that modifications and alternatives to this architecture may be made and will be apparent to the reader, such modifications and alternatives contemplated to be within the scope of this description.

[0036] In an aspect of this disclosure, user equipment 200 includes transceiver 202, which is connected to and in communication with an antenna. Transceiver 202 may be a radio frequency transceiver. Also, wireless signals may be transmitted and received via transceiver 202. Transceiver 202 may be constructed, for example, including analog and digital radio frequency (RF) "front end" functionality, circuitry for converting RF signals to a baseband frequency, via an intermediate frequency (IF) if desired, analog and digital filtering, and other conventional circuitry useful for carrying out wireless communications over modern cellular frequencies, for example, those suited for 3G or 4G communications. Transceiver 202 is connected to processor 204, which may perform the bulk of the digital signal processing of signals to be communicated and signals received, at the baseband frequency. One such function

of processor 204 is to provide a graphics interface to display 206, for the display of text, graphics, and video to the user.

[0037] In an aspect of this disclosure, processor 204 may be a processor, multi-core processor, single core process, micro controller, controller circuit, or any other type of processing device. Processor 204 can execute any type of instructions associated with the data to achieve the operations detailed herein. In one embodiment of this disclosure, the processors could transform an element or an article (e.g., data) from one state or thing to another state or thing. In another example, the activities outlined herein may be implemented with fixed logic or programmable logic (e.g., software/computer instructions executed by a processor) and the elements identified herein could be some type of a programmable processor, programmable digital logic (e.g., a field programmable gate array (FPGA), an erasable programmable read only memory (EPROM), an electrically erasable programmable ROM (EEPROM)), or an application-specific integrated circuit (ASIC) that includes digital logic, software, code, electronic instructions, flash memory, optical disks, compact disc read-only memory (CD-ROMs), digital versatile disc (DVD ROMs), magnetic or optical cards, other types of machine-readable mediums suitable for storing electronic instructions, or any suitable combination thereof.

[0038] In an aspect of this disclosure, processor 204 may also be coupled to a system memory 208 for storing information to be used in achieving operations associated with binary translation, as outlined herein. These devices may further keep information in any suitable system memory (e.g., random access memory (RAM), read only memory (ROM), FPGA, EPROM, EEPROM, etc.), software, hardware, or in any other suitable component, device, element, or object where appropriate and based on particular needs. Any of the memory or storage items discussed herein should be construed as being encompassed within the broad term "system memory" as used herein in this Specification.

[0039] In an aspect of this disclosure, system memory 208 of user equipment 200 may also include binary translation system 210. Binary translation system 210 may be configured to allow an application, compiled for one instruction set architecture to run on a different ISA. Essentially, a binary translation system can translate binaries built for a source ISA to binaries that can run on a target ISA. Binary translation system 210 may be one example of binary translation system 110.

[0040] FIGURE 3 is an example illustration of a processor according to one embodiment. Processor 300 may be the core for any type of processor, such as a microprocessor, an embedded processor, a digital signal processor (DSP), a network processor, or other device to execute code. Although only one processor 300 is illustrated in FIGURE 3, a processing element may alternatively include more than one of processor 300 illustrated in FIGURE 3. Processor 300 may be a single-threaded core or, for at least one embodiment, the processor 300 may be multithreaded in that it may include more than one hardware thread context (or “logical processor”) per core.

[0041] FIGURE 3 also illustrates a memory 302 coupled to processor 300. Memory 302 may be any of a wide variety of memories (including various layers of memory hierarchy) as are known or otherwise available to those of skill in the art. The memory 302 may include code 304, which may be one or more instructions, to be executed by processor 300. Processor 300 follows a program sequence of instructions indicated by code 304. Each instruction enters a front end logic 306 and is processed by one or more decoders 308. The decoder may generate as its output a micro operation such as a fixed width micro operation in a predefined format, or may generate other instructions, microinstructions, or control signals that reflect the original code instruction. Front end logic 306 also includes register renaming logic 310 and scheduling logic 312, which generally allocate resources and queue the operation corresponding to the convert instruction for execution.

[0042] Processor 300 is shown including execution logic 314 having a set of execution units 316-1 through 316-N. Some embodiments may include a number of

execution units dedicated to specific functions or sets of functions. Other embodiments may include only one execution unit or one execution unit that can perform a particular function. Execution logic 314 performs the operations specified by code instructions.

[0043] After completion of execution of the operations specified by the code instructions, back end logic 318 retires the instructions of code 304. In one embodiment, processor 300 allows out of order execution but requires in order retirement of instructions. Retirement logic 320 may take a variety of forms as known to those of skill in the art (e.g., re-order buffers or the like). In this manner, processor 300 is transformed during execution of code 304, at least in terms of the output generated by the decoder, hardware registers and tables utilized by register renaming logic 310, and any registers (not shown) modified by execution logic 314.

[0044] Although not illustrated in FIGURE 3, a processing element may include other elements on a chip with processor 300. For example, a processing element may include memory control logic along with processor 300. The processing element may include I/O control logic and/or may include I/O control logic integrated with memory control logic. The processing element may also include one or more caches.

[0045] FIGURE 4 is an illustration of an execution order of instructions of a source instruction set architecture in accordance with an embodiment. The source instruction set architecture (ISA) 402 includes source condition code block 404 and execution logic 406. Source condition code block 404 includes instructions 410, 412, 414, 416. The illustration in FIGURE 4 depicts the execution of instructions 410-416 over time 408.

[0046] By way of illustration, instruction 410 includes address 0x102 and the operation of IT 0x06. In this example, IT 0x06 may be the operation for a start 418 of source condition code block 404.

[0047] An initial condition state 420 may be 0x06. From condition state 420, execution logic 406 may resolve the first condition of the instruction 412. Execution

logic 406 may identify a condition bit 422. Condition bit 422 may include a ZFLAG. If the ZFLAG is a "1" then execution logic 406 executes instruction 412. After executing instruction 412, execution logic 406 may advance to condition state 424, and resolve condition bit 426. Then, execution logic 406 may execute instruction 414 because the ZFLAG of condition bit 426 is a "1".

[0048] This process is repeated using condition bit 430, condition state 428, and instruction 416 until the end of source condition code block 404. In some systems, a condition state with an "8" at the end indicates the end of a condition code block. For example, condition state 428 is 0x18, and in some systems may indicate the end of a condition code block.

[0049] In some embodiments, an example process of execution logic 406 may start with resolving the following: 1) the initial condition state 0x06 (hexadecimal number) is "0000 0110" in binary form, 2) the high 4-bits are "0000" which means the condition bit ZFLAG=1, and 3) the low 4-bits are "0110," which is not equal to "1000" (1000 is the binary equivalent of 8) which means the condition code block has not ended and includes more instructions to be executed.

[0050] FIGURE 5 is an illustration of a binary translation system 502 with an optimizer module 514 in accordance with an embodiment. Binary translation system 502 translates code from a source binary code 503 to a code on a target ISA 504. In an example embodiment, binary translation system 502 can accelerate, compared to interpreter module 506, the binary translation for emulating condition code block 505 located in source binary code 503. Binary translation system 502 does not need to emulate condition state advancing and condition bit resolving, which saves time in transformed code 516.

[0051] In an example embodiment, condition code block 505 of source binary code 503 includes the example instructions as shown in FIGURE 4. Binary translation system 502 may include an interpreter module 506, a translator module 508, and a runtime environment module 510. Translator module 508 may be an example of

translator module 118 from FIGURE 1. Binary translation system 502 may be one example of binary translation system 110 as shown in FIGURE 1. Source binary code 503 may be an example of source binary code blocks 130 from FIGURE 1.

[0052] Interpreter module 506 can be configured as an interpretation engine that executes source binary code 503. Interpreter module 506 can also trigger utilization of translator module 508 when it identifies a hot code trace. A hot code trace may also be referred to as a hot basic block. Binary translation system 502 may implement probes (counters) to detect "hot" (i.e., frequently executed) basic blocks. An example of a hot code trace may be, for example, condition code block 505.

[0053] Translator module 508 performs source binary code 503 to target ISA 504 binary translation on each hot code trace. Translator module 508 has a translated code cache 512 that stores translated code blocks. Translated code cache 512 may be one example of translated code cache 122 from FIGURE 1.

[0054] Translator module 508 also has optimizer module 514 that optimizes source binary code 503 during translation. Optimizer module 514 may optimize source binary code 503 by transforming it into a transformed code 516. Transformed code 516 may be produced from a pre-execution of the source binary code 503 with the advancing and resolving operations (or instructions) eliminated. Optimizer module 514 may be able to identify some or all of the conditions during the pre-execution of the source binary code 503.

[0055] Optimizer module 514 can perform a source ISA to source ISA transformation on condition code block 505. Optimizer module 514 also decodes the multiple conditions on a conditions-specifying instruction, removes the conditions-specifying instruction, and transforms the condition code block into a series of code as if each instruction has encoded its own condition bits. In this example, IT 0x06 may be the instruction for the start of condition code block 505

[0056] Runtime environment module 510 provides supporting environment and helper libraries to run translated code on target ISA 504.

[0057] In operational terms, and in terms of one particular embodiment the translator module working flow may perform as follows: When a source binary application runs on binary translation system 502, interpreter module 506 starts to execute source binary code 503. When condition code block 505 is detected (e.g., as a hot code trace), interpreter module 506 triggers translator module 508 to perform source ISA to target ISA translation. During translation, when translator module 508 meets condition code block 505, it invokes optimizer module 514 to perform source binary code 503 transformations.

[0058] Then, optimizer module 514 decodes condition bits on a conditions-specifying instruction of condition code block 505, statically advances the condition state and resolves condition bits on each following instruction, and finally removes the conditions-specifying instruction of condition code block 505. In this way, optimizer module 514 transforms condition code block 505 into a series of code (e.g., transformed code 516), as if each instruction has encoded its own condition bits.

[0059] Also, translator module 508 does source ISA to target ISA translation. For source binary code 503 in condition code block 505, translator module 508 guarantees that the translation of each condition code block instruction is strictly in program order, i.e., instruction reordering may not be performed. The address mapping of condition code block 505 may also be recorded in condition state recovery table 518 of the system.

[0060] FIGURE 6 is a simplified flowchart illustrating a method for binary translation with an optimizer module in accordance with an embodiment. The flow 600 may begin at 610, when the interpreter module interprets source instructions. By interpreting the instructions, the interpreter module takes each instruction, analyzes, the instruction, and executes the instruction on the source ISA. During the interpretation, the interpreter module may identify a hot trace (or hot basic block) at 620.

[0061] Once a hot trace is identified, the binary translation system may provide a translator module to perform a translation at 630. The translator module then converts (also referred to as "translates") the source instructions to transformed instructions. In one or more embodiments, transformed instructions may include target instructions, which may also be referred to as target binary code or translated code. At 640, the translator module may identify a condition code block within the hot trace.

[0062] At 650, the translator module may handover the conversion (also referred to as the "translation") to the optimizer module in response to identifying the condition code block. The optimizer module may be a part of the translator module. Next, the optimizer module may perform a pre-execution of the condition code block at 660. The pre-execution may resolve the condition bits in the block to form a pre-resolved condition set, as well as eliminate the need for emulation of condition state advancing. The reduction of these commands makes the translated code more efficient.

[0063] After the pre-execution, at 670, the optimizer module may encode the source instructions with the pre-resolved condition set when converting to an intermediate instruction set to run on the target ISA. Each of the new instructions may be a combined instruction set that is encoded with the condition information. In this example, the translator module does not have to spend resources to identify and advance the condition state. Additionally, the initial condition-specifying instruction is no longer necessary. Thereafter, the process terminates.

[0064] FIGURE 7 is an illustration of a binary translation system 702 with a recovery table 718 in accordance with an embodiment. Binary translation system 702 in FIGURE 7 is similar to binary translation system 502 in FIGURE 5. For instance, binary translation system 702 translates code from a source binary code 703 to code for a target ISA 704. Binary translation system 702 also includes an interpreter module 706, a translator module 708, and a runtime environment module 710. Translator module 708 includes an optimizer module 714 and a translated code cache 712. Runtime environment module 710 of binary translation system 702 includes a recovery table 718.

Also, the various components in binary translation systems 502 and 702 may be combined in different ways to form other binary translations systems according to the various embodiments.

[0065] During execution of the instructions in translated code cache 712, sometimes an interrupt or exception 711 occurs. During exception 711, translator module 708 may lose the condition state information that was pre-resolved by optimizer module 714. Accordingly, translator module 708 may consult recovery table 718 to help recover the condition state information.

[0066] In operational terms, and in terms of one particular embodiment, when exception 711 occurs in translated code of a translation of condition code block 705, a condition state recovery process can include runtime environment module 710 catching exception 711 and identifying an address of the exception instruction in recovery table 718. If the address is not in recovery table 718, runtime environment module 710 does a normal state recovery process. If the address can be found in recovery table 718, runtime environment module 710 fetches the original condition code block 705 address in source binary code 703, and starts a condition state recovery process, for example, the process as shown in FIGURES 8 and 9.

[0067] From the address, runtime environment module 710 can re-decode the condition bits in the conditions-specifying instruction, re-advance condition state to the faulting point, and re-resolve the condition bits of the faulting source ISA instruction. Hence, the condition state of the faulting point can be correctly recovered. Runtime environment module 710 can recover other source ISA states. Once all source ISA states are recovered, binary translation system 702 can safely invoke interpreter module 706.

[0068] A condition state recovery process can include, as input, an exception/interrupt ip in target ISA code. The process can include, as output, a recovered condition_state. Pseudo code could be configured accordingly:

[0069] let eip = exception/interrupt ip in target ISA code

```
[0070]    {offset,ccb_entry_ip}=
BT.runtime.condition_state_recovery_table[eip]
[0071]    condition_state = BT.runtime.decode(ccb_entry_ip)
[0072]    for i = 1 to offset:
[0073]        condition_state=
BT.runtime.condition_state_advance(condition_state)
```

[0074] FIGURE 8 is a simplified flowchart 800 illustrating a method for binary translation by a binary translation system with a condition state recovery table (e.g., binary translation system 702 as shown in FIGURE 7) in accordance with an embodiment. The flow may begin at 810, when the binary translation system (BTS) identifies an exception/interrupt during execution of an instruction of translated code. Then, at 820, the binary translation system retrieves an address of the interrupted instruction.

[0075] Next, using the target address, the binary translation system can use a recovery table to identify a source address at 830. The source address may include an entry address along with an offset. The entry address may be the beginning address of the condition code block in the source binary code. Additionally, the offset may be multiplied by the instruction size. For example, as shown in FIGURE 7, the recovered source address would be 0x102 (entry) plus 2 (offset) times 2 (instruction size). For example, the instruction size may be 2 for a 2-byte instruction.

[0076] Once a source address is identified, the binary translation system can identify, at 840, the corresponding initial condition state at the source address. The initial condition state may be the initial state of the condition code block in the source ISA, containing the source instruction that was translated to the interrupted target ISA code instruction.

[0077] At 850, the binary translation system sets the current condition state in the translator module to the initial condition state that was identified at the source address. However, the initial condition state is not the desired state. To find the desired state, at 860, the binary translation system may need to advance the condition state a

number of times based on the value of the offset. The process for advancing the condition state is shown in FIGURE 7. Once the condition state has been advanced the proper number of times, at 870, the binary translation system may recover the current condition state of the source binary code at the exception (i.e., at the faulting address).

[0078] FIGURE 9 is a simplified flowchart 900 illustrating a method for advancing the condition state using the recovery table in accordance with an embodiment. The flow in FIGURE 9 may complement parts of flow 800 of FIGURE 8. The binary translation system goes through the conditions until it locates the condition state of the source binary code instruction that was interrupted. To find the correct condition, the binary translation system initiates a loop.

[0079] The binary translation system sets a variable to "1" at 910. Then, at 920, the binary translation system determines whether the variable meets the offset that was identified in the recovery table

[0080] If the variable meets the offset, then at 960, the binary translation system sets the condition at the exception in the target binary code to the current condition state in the source binary code.

[0081] If the variable does not match the offset, the binary translation system advances the condition state at 930. The binary translation system then sets the current condition state to the advanced condition state at 940. Also, the binary translation system advances the variable at 950. The process is then repeated beginning at 920. Once the variable meets the offset that was identified in the recovery table, the current condition state of the source ISA at the exception is recovered at 960, and thereafter the process terminates.

[0082] The embodiments of this disclosure recognize that during binary translation, when an interrupt or exception occurs, current systems may recover the state of the source binary code by falling back to the interpreter to finish execution slowly and safely. However, the embodiments of this disclosure provide a process for the translator to recover the condition state.

[0083] FIGURE 10A is a simplified flowchart illustrating a method for identifying a parallel-multiply-addition instruction in accordance with an embodiment. A flow 1000 may begin at 1010, when the translator module receives an instruction.

[0084] Then, at 1020, the translator module determines whether the instruction is a parallel-multiply-addition instruction. If the instruction is not a parallel-multiply-addition instruction, the process terminates. If the instruction is a parallel-multiply-addition instruction, at 1025, the translator module triggers a translation on the parallel-multiply-addition instruction. The translator module may identify the parallel-multiply-addition instruction as including:

[0085] Vector 1 = $X1X0$

[0086] Vector 2 = $Y1Y0$

[0087] $Dest_tmp = X1*Y1$ and $X0*Y0$

[0088] $Dest = X1*Y1 + X0*Y0$,

[0089] with the dest address including the result of the parallel-multiply-addition instruction.

[0090] FIGURE 10B is a simplified flowchart illustrating a method for binary translation with a parallel-multiply-addition instruction in accordance with an embodiment. The flow 1001 may begin at 1030, when the translator module generates translated code to perform a parallel-multiply-addition function on the source vectors indicated in the instruction. For example, if there are two source vectors, they are multiplied and the resulting vector may be referred to as the destination vector.

[0091] At 1040, in the translated code, once the function has been executed, the translated code determines whether there was an overflow. To identify an overflow, the translated code compares the destination vector to a constant. For example, in an embodiment, if the destination vector is equal to the value of $[0x1 \ll (S*N-1)]$ then an overflow is detected. In this formula, "S" is the size of vector elements and "N" is the number of vector elements. Thus, $(S*N)$ is the vector size. If an overflow was not

detected, at 1050, an overflow flag is set to "0". If an overflow is detected, at 1060, the overflow flag is set to "1".

[0092] The embodiments of this disclosure improve the efficiency in binary translation for parallel multiply instruction and its variants (addition, subtraction, extension) when target ISA does not offer these instructions with the same semantics.

[0093] In operational terms, and in terms of one particular embodiment, parallel multiply addition multiplies the individual signed elements of the first source vector by the corresponding signed elements of the second source vector, producing temporary signed, double element results. The adjacent double element results are then summed and stored back to the destination vector.

[0094] In other implementations, the variants of parallel multiply instruction may have slightly different semantics. Overflow may occur during the addition. Addition may become subtraction, eliciting Parallel Multiply Subtraction, which has no target ISA counterpart. Moreover, a third source vector may be added to the final result.

[0095] The embodiments of this disclosure provide processes that show how to efficiently translate the parallel multiply instruction and its variants to the basic operation when only a portion of instructions of the target ISAs is mapped. A vector may contain several elements. Herein, (S) is denoted as the element size, (N) is denoted as the number of elements of the vector, and (N*S) is the vector size.

[0096] In order to implement the closest target ISA instruction into parallel multiply instruction and its variants, overflow and negation of the Parallel Multiply Addition function should be efficiently emulated during the translation.

[0097] Overflow only occurs in Parallel Multiply Addition if the summed result reaches a specific overflow boundary value. In this situation, the overflow flag may need to be set.

[0098] Pseudo code of overflow:

[0099] dest = parallel_multiply_add (src1, src2)

[0100] If (dest == 0x1<<(S*N-1)) generate O_flag

[0101] The embodiments of this disclosure may detect the overflow condition by comparing the result value to a constant value. There may only be one condition that the Parallel Multiple Addition can overflow. For example, when all elements of multiply operation are of the biggest absolute value negative $0x1 \ll (S-1)$, and the multiply operation generates positive $0x1 \ll (2S-2)$. When these two positive numbers are summed, it produces $0x1 \ll (2S-1)$. The vector size cannot hold this large result, which produces an overflow boundary value.

[0102] For example, assume an element size is 8-bit, and the resulting size is 16-bit. Also assume $\text{vector1} = 0x8080$, and $\text{vector2} = 0x8080$. In this case, $\text{parallel_multiply_add}(\text{vector1}, \text{vector2}) = 0x80 * 0x80 + 0x80 * 0x80 = 0x4000 + 0x4000 = 0x8000$. The result is a negative number and, therefore, overflow occurs (two positive numbers are summed to a negative number). Consequently, $0x8000$ is determined as an overflow boundary.

[0103] When a variant of Parallel Multiply Addition requires setting an overflow flag (O_flag), detecting overflow information helps to generate the correct O_flag efficiently. When a variant of Parallel Multiply Addition adds a third source vector to the result of Parallel Multiply Addition, the overflow that occurred during Parallel Multiply Addition affects the calculation of the last addition. Therefore, the overflow conditions that occurred during the first addition and the last addition are combined. Detecting overflow in the Parallel Multiple Addition helps to generate efficient code to calculate the combined overflow flag.

[0104] FIGURE 11A is a simplified flowchart illustrating a method for identifying a parallel-multiply-subtraction instruction in accordance with an embodiment. A flow 1100 may begin at 1110, when the translator module receives an instruction.

[0105] Then, at 1120, the translator module determines whether the instruction is a parallel-multiply-subtraction instruction. In an embodiment, there may be two vectors, a first source vector and a second source vector. Each vector may have

an upper half and a lower half. For example if the first source vector is X1X0 then the lower half is X0. Likewise, a second source vector may have the values Y1Y0. In this example, the value "Y0" may be negated. The vector to be subtracted may be the second source vector. For example, the equation may be the first source vector minus the second source vector.

[0106] If the instruction is not a parallel-multiply-subtraction instruction, as determined at 1120, the process terminates. If the instruction is a parallel-multiply-subtraction instruction, then at 1125, the translator module triggers a translation on the parallel-multiply-subtraction instruction.

[0107] FIGURE 11B is a simplified flowchart illustrating a method for binary translation with a parallel-multiply-subtraction instruction in accordance with an embodiment. The flow 1101 may begin at 1130, when the translator module generates translated code configured to perform a bitwise-not on a low element of a second source vector. At 1135, after the element is changed by the bitwise-not operation, the translated code is then configured to perform a parallel-multiply-addition function on the source vectors indicated in the instruction. For example, if there are two source vectors, they are multiplied and produce a resulting vector. Then, at 1140, the low element of the first source vector is sign extended and added to the resulting vector. This final vector may be referred to as the destination vector.

[0108] Next, the flow moves to identifying an overflow. At 1150, the translated code may be configured to determine whether there is an overflow at the destination vector. The translated code may be configured to compare the destination vector to a constant to identify an overflow. For example, the translated code may be configured to identify the vector to the constant $[0x1 \ll (S*N-1) \text{ plus sign extended } (0x1 \ll (S-1))]$.

[0109] If there is an overflow, at 1160, the translated code may be configured to set the resulting destination vector to a constant value. Thereafter, the process may terminate. If there is not an overflow, at 1170, the translated code may be

configured to then determine whether the high elements were subtracted from the low elements, or vice versa. For example, in the above operations assume that the low elements are subtracted from the high elements. However, if the reverse occurs, the destination vector may need to have a bitwise-not operation applied. Accordingly, if it is determined at 1170 that the low elements are subtracted from the high elements, in other words, the high elements are not subtracted from the low elements, the process may terminate. On the other hand, if it is determined at 1170 that the high elements are subtracted from the low elements, then at 1180, the translated code is configured to perform a bitwise-not operation on the destination vector. Thereafter, the process may terminate.

[0110] In operational terms, and in terms of one particular embodiment, when translating Parallel Multiply Subtraction (addition becomes subtraction), the second source vector's low element needs to be negated before the operation, so that Parallel Multiply Addition can be used to perform Parallel Multiply Subtraction.

[0111] Pseudo code of Parallel Multiply Subtraction 1:

[0112] $\text{src2.low_Sbit} = \text{not}(\text{src2.low_Sbit})$

[0113] $\text{dest}' = \text{parallel_multiply_add}(\text{src1}, \text{src2})$

[0114] $\text{dest} = \text{dest}' + \text{Sign-extend}(\text{src1.low_Sbit})$

[0115] In a second portion of the method, overflow may occur in Parallel Multiply Addition. This occurs when the inputs hold specific values, and therefore, the final result can be determined directly.

[0116] Pseudo code of Parallel Multiply Subtraction 2:

[0117] If (overflow detected)

[0118] {

[0119] $\text{dest} = (0x1 \ll (S*N-1)) + \text{sign_extend}(0x1 \ll (S-1))$

[0120] }

[0121] In a third portion of the method, Parallel Multiply Subtraction has two different subtraction orders, high elements multiply result subtracts low elements

multiply result, or low elements multiply result subtracts high elements multiply result. The above paragraphs address high elements multiply result subtracting low elements multiply result. In the alternative scenario, when low elements multiply result subtracts high elements multiply result, a negation of the final result is performed.

[0122] Pseudo code of Parallel Multiply Subtraction 3:

[0123] dest = ~dest

[0124] The different embodiments provide translations of source binary code into target binary code when the target binary code does not provide equivalent counterparts to overflow and negation problems occurring in the translations of parallel multiply instructions.

[0125] With respect to FIGURES 10 and 11, current translations for parallel multiply instructions use target ISA's general multiply instructions to emulate packed elements multiply. To implement a parallel multiply of two vectors that contain "N" number of elements, "N" times of a general multiply are performed. In addition, extracting elements from the vector needs an extra shift operation. An overflow flag is generated during the addition. However, it should be used to perform the addition to a third vector if this instruction's semantics requires a third addition. This translation may be performed by more than ten instructions.

[0126] The embodiments of this disclosure combine the "N" number of multiplies together as one operation to translate the Parallel Multiply Addition efficiently. In other words, one parallel multiply instruction can be implemented to emulate several general multiply instructions. The embodiments disclosed herein detect the overflow condition by comparing the resulting value to a constant value. Also, embodiments disclosed herein negate one element of the source vector to handle the Parallel Multiply Subtraction. A new translation may be imported into the binary translation system proposed by the embodiments of this disclosure.

[0127] An example overflow handling algorithm may be configured as follows:

[0128] Input: A sequence of source ISA instruction, denoted as SIA_INST

[0129] Output: An overflow flag information, denoted as O_flag

[0130] Algorithm:

[0131] Begin

[0132] dest = parallel_multiply_add (src1, src2)

[0133] If (dest == 0x1<<(S*N-1))

[0134] generate o_flag

[0135] Endif

[0136] End

[0137] An example Parallel Multiply Subtraction algorithm may be configured as follows:

[0138] Input: A sequence of source ISA instruction, denoted as SIA_INST

[0139] Output: A sequence of target ISA instruction, denoted as TIA_INST

[0140] Algorithm:

[0141] Begin

[0142] Src2.low_Sbit = not (src2.low_Sbit)

[0143] dest' = Parallel_multiply_add (src1, src2)

[0144] dest = dest' + Sign-extend(src1.low_Sbit)

[0145] If (overflow detected)

[0146] dest = const_val

[0147] Endif

[0148] If (low elements subtract high elements) dest = ~dest

[0149] Endif

[0150] End

[0151] FIGURE 12A is a simplified flowchart illustrating a method for identifying a partial-long-multiply instruction in accordance with an embodiment. A flow 1200 may begin at 1210, when the translator module receives an instruction.

[0152] Then, at 1220, the translator module determines whether the instruction is a partial-long-multiply instruction. If the instruction is not a partial-long-multiply instruction, the process terminates. If the instruction is a partial-long-multiply instruction, then at 1225, the translator module triggers a translation on the partial-long-multiply instruction.

[0153] FIGURE 12B is a simplified flowchart illustrating a method for binary translation with a partial-long-multiply instruction in accordance with an embodiment. The flow 1201 may begin at 1230, when the translator module generates translated code to extract the lower and upper halves of the original operand of the instruction to form two operands. Also, at 1240, the translated code may be configured to zero-extend the lower half of the original operand, from half size to full size. Next, at 1250, the translated code may be configured to do an arithmetic-shift-right on the upper half of the original operand, from half size to full size.

[0154] Next, at 1260, the translated code may be configured to multiply each half of the original operand with a second operand to form two products. Finally, at 1270, the translated code may be configured to add the two products together. In order to add the products together, the translated code may be configured to adjust the lower-half-multiply-product from full size back to half size, by using arithmetic-shift-right. Thereafter, the process terminates.

[0155] The embodiments of this disclosure improve the efficiency in binary translation for Partial Long Multiply instructions (e.g., signed Word by Halfword Multiply instruction in ARM) when source ISA and target ISA do not offer these instructions with the same semantics.

[0156] In operational terms, and in one embodiment, a Partial Long Multiply instruction multiplies a signed long-bit quantity by a signed short-bit quantity, and produces a partial long-bit result. For example, a 16-bit quantity is multiplied by a 32-bit quantity that produces a 48-bit result; the top 32 bits of the 48-bit result are stored

while the bottom 16 bits of the 48-bit result are ignored. An extra accumulate function can be performed depending on the instruction's semantic.

[0157] Different ISAs may implement different variants of multiply instructions. Typically, the source vectors have the same data size. When translating a multiply instruction of source vectors with different data sizes, however, it is inefficient to auto-extend the shorter data size into the longer data size, use long multiply instruction to perform the calculation, and extract the needed bits to a destination vector.

[0158] Long multiply can be more expensive than short multiply. In comparison to short multiply, long multiply has several disadvantages for translation. In long multiply, efficiency is low. Long multiply has a longer latency and takes both ports, restricting the pipeline. There are also register constraints. A target ISA instruction may occupy a specific register as the operand register that sabotages other components' optimization.

[0159] The embodiments of this disclosure provide how to efficiently translate the Partial Long Multiply instruction by using two short multiply instructions to emulate long multiply instruction. The embodiments use the following denotation in this disclosure. A signed multiply Word by Halfword instruction is denoted as SMULWH. Short-bit Size of the instruction vector is denoted as SB, while Long-bit Size ($LB=2*SB$) is denoted as LB.

[0160] If only a portion of the multiply result is needed, two short multiplies can be used to emulate one long multiply. It equally divides the long-bit source vector into two parts (low-bit half and high-bit half), multiplies these two parts by the second short-bit vector separately, and adds these two intermediate results together to yield the final result. A shift is performed before the addition to align the two intermediate results to the corresponding position.

[0161] The translation for a Partial Long Multiply instruction generally contains two functions. The first function prepares two parts of the long-bit source

vector (src_long). A low half extension (extend (src_long.low_half)) and a high half shift (src_long >> SB) are executed. The low half extension can be a zero extension, and the high half shift can be an arithmetic shift.

[0162] The second function multiplies these two parts with the short-bit vector and assembles the intermediate results together. It shifts one multiplying result (low_multiply_result << SB), then adds two multiplying results together to generate the final result. A third source vector may be added to the final result according to the source ISA instruction's semantics.

[0163] Typically, when encountering a multiply with two vectors with different data sizes, a long-bit data size is selected as the operation size, a short-bit data size vector is signed or unsigned extended to the long-bit data size vector, and a long multiply instruction is used to perform the operation. One partial multiply instruction can be translated into around six target instructions and two extra specific register save and restore instructions. However, one long multiply can be much slower than a short multiply. In addition, a long multiply has register constraints as a specific register usage can cause extra save and restore work. These two aspects can lower the efficiency of translation.

[0164] The embodiments of this disclosure divide one long multiply into two short multiplies. It chooses more efficient, eliminates extra specific register save and restore work, and simplifies the register allocation. One source ISA instruction can be translated into six target ISA instructions.

[0165] An example Instruction Translation Algorithm for a Partial Long Multiply instruction may be configured as follows:

- [0166] Input: A sequence of source ISA instruction, denoted as SIA_INST
- [0167] Output: A sequence of target ISA instruction, denoted as TIA_INST
- [0168] Algorithm:
- [0169] Begin
- [0170] src_long_lh' = zero_extend (src_long_lh, SB_size)

[0171] src_long_hh' = arithmetic_shift_right (src_long_hh, SB_size)

[0172] lh_product = imul (src_long_lh', src_short)

[0173] hh_product = imul (src_long_hh', src_short)

[0174] lh_product' = arithmetic_shift_right (lh_product, SB_size)

[0175] dest = add (lh_product', hh_product)

[0176] If the current instruction is with extension

[0177] dest = add (dest, src_ext)

[0178] Endif

[0179] End

[0180] Note that in certain example implementations, the binary translation functions outlined herein may be implemented by logic encoded in one or more tangible media (e.g., embedded logic provided in an application specific integrated circuit (ASIC), digital signal processor (DSP) instructions, software (potentially inclusive of object code and source code) to be executed by a processor, or other similar machine, etc.). In some of these instances, a memory element can store data used for the operations described herein. This includes the memory element being able to store software, logic, code, or processor instructions that are executed to carry out the activities described in this Specification. A processor can execute any type of instructions associated with the data to achieve the operations detailed herein in this Specification. In one example, the processor could transform an element or an article (e.g., data) from one state or thing to another state or thing. In another example, the activities outlined herein may be implemented with fixed logic or programmable logic (e.g., software/computer instructions executed by a processor) and the elements identified herein could be some type of a programmable processor, programmable digital logic (e.g., FPGA, EPROM, EEPROM) or an ASIC that includes digital logic, software, code, electronic instructions, or any suitable combination thereof.

[0181] In one example implementation, the binary translation systems may include software in order to achieve the binary translation activities outlined herein. The

binary translation systems can include memory elements for storing information to be used in achieving the binary translation activities, as discussed herein. Additionally, the binary translation systems may include a processor that can execute software or an algorithm to perform the binary translation activities, as disclosed in this Specification. These devices may further keep information in any suitable memory element (random access memory (RAM), ROM, EPROM, EEPROM, ASIC, etc.), software, hardware, or in any other suitable component, device, element, or object where appropriate and based on particular needs. Any of the memory items discussed herein (e.g., databases, tables, trees, caches, etc.) should be construed as being encompassed within the broad term 'memory element.' Similarly, any of the potential processing elements, modules, and machines described in this Specification should be construed as being encompassed within the broad term 'processor.'

[0182] Note that with the example provided above, as well as numerous other examples provided herein, interaction might be described in terms of two, three, or four elements. However, this has been done for purposes of clarity and example only. In certain cases, it may be easier to describe one or more of the functionalities of a given set of flows by only referencing a limited number of elements. It should be appreciated that the binary translation systems (and their teachings) are readily scalable and can accommodate a large number of components, as well as more complicated/sophisticated arrangements and configurations. Accordingly, the examples provided should not limit the scope or inhibit the broad teachings of the binary translation systems as potentially applied to a myriad of other architectures.

[0183] It is also important to note that the operations in the preceding flow diagrams illustrate only some of the possible scenarios and patterns that may be executed by, or within, a binary translation system. Some of these operations may be deleted or removed where appropriate, or may be modified or changed considerably without departing from the scope of the present disclosure. In addition, a number of these operations have been described as being executed concurrently with, or in

parallel to, one or more additional operations. However, the timing of these operations may be altered considerably. The preceding operational flows have been offered for purposes of example and discussion. A binary translation system provides substantial flexibility in that any suitable arrangements, chronologies, configurations, and timing mechanisms may be provided without departing from the teachings of the present disclosure.

[0184] Although the present disclosure has been described in detail with reference to particular arrangements and configurations, these example configurations and arrangements may be changed significantly without departing from the scope of the present disclosure.

[0185] The following examples pertain to embodiments in accordance with this Specification. One or more embodiments may provide a method for performing binary translation. The method may include: translating, by a translator module executing on a processor, source instructions to target instructions; identifying a condition code block in the source instructions, wherein the condition code block includes a plurality of condition bits; and transforming the condition code block, wherein the transforming includes performing a pre-execution on the condition code block to resolve the plurality of condition bits in the condition code block to form a pre-resolved condition set.

[0186] In an example of one or more embodiments, the method may further include encoding the source instructions with the pre-resolved condition set.

[0187] One or more embodiments may provide a method for performing binary translation. The method may include: identifying, by a runtime environment module executing on a processor, an interrupt at an instruction in the target instructions; retrieving a target address of the interrupted instruction, wherein the target address is for a target instruction set architecture; and identifying a source address of the interrupted instruction in a recovery table, wherein the target address is paired to the source address in the recovery table.

[0188] In an example of one or more embodiments, the identifying the source address may include: identifying an offset for the source address of the interrupted instruction in the recovery table, wherein the offset is multiplied by an instruction size of the interrupted instruction; identifying an initial condition state at the source address; setting a current condition state to the initial condition state; advancing the current condition state a number of times based on a value of the offset; and recovering the current condition state.

[0189] In another example of one or more embodiments, the advancing the current condition state may include: setting a variable to 1; determining whether the offset is equal to the variable; responsive to the offset equaling the variable, recovering the current condition state; responsive to the offset not equaling the variable, advancing the condition state of the condition code block; responsive to advancing the condition state, setting the current condition state to match the advanced condition state; responsive to setting the current condition state, advancing the variable; and repeating the determining whether the offset is equal to the variable until the offset is equal to the variable.

[0190] One or more embodiments may provide a method for translating a parallel multiply addition instruction. The method may include: receiving, by a translator module executing on a processor, an instruction to be translated, wherein the instruction calls for a parallel multiply addition function for a number of source vectors; and generating translated code to perform: executing the parallel multiply addition function for the number of source vectors; and determining whether there is an overflow.

[0191] In an example of one or more embodiments, the method may further include: responsive to determining an overflow, setting an overflow flag to indicate there is overflow.

[0192] One or more embodiments may provide a method for translating a parallel multiply subtraction instruction. The method may include: receiving, by a

translator module executing on a processor, an instruction to be translated, wherein the instruction calls for a parallel multiply subtraction function for a plurality of source vectors, wherein each vector of the plurality of source vectors has a low element and a high element; and generating translated code to perform: performing a bitwise-not operation on a low element of a second source vector; executing the parallel multiply addition function for the plurality of source vectors to form a result; adding a low element of the first source vector to the result; determining whether there is an overflow; responsive to determining an overflow, setting the result to a constant value.

[0193] In an example of one or more embodiments, the method may further include: responsive to not determining an overflow, determining whether the high elements were subtracted from the low elements; and responsive to the high elements being subtracted from the low elements, performing a bitwise-not operation on the result.

[0194] One or more embodiments may provide a method for translating a partial long multiply instruction. The method may include: receiving, by a translator module executing on a processor, an instruction to be translated, wherein the instruction calls for a partial long multiply function for a plurality of source vectors, and wherein the partial long multiply function has a first operand; and generating translated code to perform: splitting the first operand of the function into an upper half and a lower half; multiplying each half of the first operand with a second operand to form an upper half product and a lower half product; and adding the upper half product to the lower half product.

[0195] In an example of one or more embodiments, the splitting the first operand of the function may include: performing a zero-extend operation on the lower half; and performing an arithmetic shift right operation on the upper half.

[0196] In another example of one or more embodiments, the method may also include: performing an arithmetic shift right operation on the lower half product.

[0197] One or more embodiments may provide at least one computer-readable medium comprising one or more instructions that when executed by a processor: translate source instructions to target instructions; identify a condition code block in the source instructions, wherein the condition code block includes a plurality of condition bits; and transform the condition code block, wherein the transforming includes performing a pre-execution on the condition code block to resolve the plurality of condition bits in the condition code block to form a pre-resolved condition set.

[0198] In an example of one or more embodiments, the at least one computer-readable medium may include further instructions that when executed by the processor: encode the source instructions with the pre-resolved condition set.

[0199] One or more embodiments may provide at least one computer-readable medium comprising one or more instructions that when executed by a processor: identify an interrupt at an instruction in target instructions; retrieve a target address of the interrupted instruction, wherein the target address is for a target instruction set architecture; and identify a source address of the interrupted instruction in a recovery table, wherein the target address is paired to the source address in the recovery table.

[0200] In an example of one or more embodiments of the at least one computer-readable medium, the identifying the source address may include one or more instructions that when executed by the processor: identify an offset for the source address of the interrupted instruction in the recovery table, wherein the offset is multiplied by an instruction size of the interrupted instruction; identify an initial condition state at the source address; set a current condition state to the initial condition state; advance the current condition state a number of times based on a value of the offset; and recover the current condition state.

[0201] In another example of one or more embodiments of the at least one computer-readable medium, the advancing the current condition state may include one or more instructions that when executed by the processor: set a variable to 1;

determine whether the offset is equal to the variable; responsive to the offset equaling the variable, recover the current condition state; responsive to the offset not equaling the variable, advance the condition state of the condition code block; responsive to advancing the condition state, set the current condition state to match the advanced condition state; responsive to setting the current condition state, advance the variable; and repeat determining whether the offset is equal to the variable until the offset is equal to the variable.

[0202] One or more embodiments may provide at least one computer-readable medium comprising one or more instructions that when executed by a processor: receive an instruction to be translated, wherein the instruction calls for a parallel multiply addition function for a number of source vectors; and generate translated code to: execute the parallel multiply addition function for the number of source vectors; and determine whether there is an overflow.

[0203] In an example of one or more embodiments, the at least one computer-readable medium may include further instructions that when executed by the processor: responsive to determining an overflow, set an overflow flag to indicate there is overflow.

[0204] One or more embodiments may provide at least one computer-readable medium comprising one or more instructions that when executed by a processor: receive an instruction to be translated, wherein the instruction calls for a parallel multiply subtraction function for a plurality of source vectors, wherein each vector of the plurality of source vectors has a low element and a high element; and generate translated code to: perform a bitwise-not operation on a low element of a second source vector; execute the parallel multiply addition function for the plurality of source vectors to form a result; add a low element of the first source vector to the result; determine whether there is an overflow; and responsive to determining an overflow, set the result to a constant value.

[0205] In an example of one or more embodiments, the at least one computer-readable medium may include further instructions that when executed by the processor: responsive to not determining an overflow, determine whether the high elements were subtracted from the low elements; and responsive to the high elements being subtracted from the low elements, perform a bitwise-not operation on the result.

[0206] One or more embodiments may provide at least one computer-readable medium comprising one or more instructions that when executed by a processor: receive an instruction to be translated, wherein the instruction calls for a partial long multiply function for a plurality of source vectors, and wherein the partial long multiply function has a first operand; and generate translated code to: split the first operand of the function into an upper half and a lower half; multiply each half of the first operand with a second operand to form an upper half product and a lower half product; and add the upper half product to the lower half product.

[0207] In an example of one or more embodiments of the at least one computer-readable medium, the splitting the first operand of the function may include one or more instructions that when executed by the processor: perform a zero-extend operation on the lower half; and perform an arithmetic shift right operation on the upper half.

[0208] In another example of one or more embodiments, the at least one computer-readable medium may include further instructions that when executed by the processor: perform an arithmetic shift right operation on the lower half product.

[0209] One or more embodiments may provide an apparatus comprising: a translator module configured to: translate source instructions to target instructions; and identify a condition code block in the source instructions, wherein the condition code block includes a plurality of condition bits; and an optimizer module configured to transform the condition code block, wherein the transforming includes performing a pre-execution on the condition code block to resolve the plurality of condition bits in the condition code block to form a pre-resolved condition set.

[0210] In an example of one or more embodiments, the optimizer module may be further configured to: encode the source instructions with the pre-resolved condition set.

[0211] One or more embodiments may provide an apparatus comprising: a recovery table configured to store a number of target addresses and a number of source addresses; and a runtime environment module configured to: identify an interrupt at an instruction in target instructions; retrieve a target address of the interrupted instruction, wherein the target address is for a target instruction set architecture; and identify a source address of the interrupted instruction in the recovery table, wherein the target address of the number of target addresses is paired to the source address of the number of source addresses in the recovery table.

[0212] In an example of one or more embodiments, in the identifying the source address, the runtime environment module may be further configured to: identify an offset for the source address of the interrupted instruction in the recovery table, wherein the offset is multiplied by an instruction size of the interrupted instruction; identify an initial condition state at the source address; set a current condition state to the initial condition state; advance the current condition state a number of times based on a value of the offset; and recover the current condition state.

[0213] In another example of one or more embodiments, in the advancing the current condition state, the runtime environment module may be further configured to: set a variable to 1; determine whether the offset is equal to the variable; responsive to the offset equaling the variable, recover the current condition state; responsive to the offset not equaling the variable, advance the condition state of the condition code block; responsive to advancing the condition state, set the current condition state to match the advanced condition state; responsive to setting the current condition state, advance the variable; and repeat determining whether the offset is equal to the variable until the offset is equal to the variable.

[0214] One or more embodiments may provide an apparatus comprising: a translator module configured to: receive an instruction to be translated, wherein the instruction calls for a parallel multiply addition function for a number of source vectors; and generate translated code to: execute the parallel multiply addition function for the number of source vectors; and determine whether there is an overflow.

[0215] In an example of one or more embodiments, the translator module may be further configured to: responsive to determining an overflow, set an overflow flag to indicate there is overflow.

[0216] One or more embodiments may provide an apparatus comprising: a translator module configured to: receive an instruction to be translated, wherein the instruction calls for a parallel multiply subtraction function for a plurality of source vectors, wherein each vector of the plurality of source vectors has a low element and a high element; and generate translated code to: perform a bitwise-not operation on a low element of a second source vector; execute the parallel multiply addition function for the plurality of source vectors to form a result; add a low element of the first source vector to the result; determine whether there is an overflow; and responsive to determining an overflow, set the result to a constant value.

[0217] In an example of one or more embodiments, the translator module may be further configured to: responsive to not determining an overflow, determine whether the high elements were subtracted from the low elements; and responsive to the high elements being subtracted from the low elements, perform a bitwise-not operation on the result.

[0218] One or more embodiments may provide an apparatus comprising: a translator module configured to: receive an instruction to be translated, wherein the instruction calls for a partial long multiply function for a plurality of source vectors, and wherein the partial long multiply function has a first operand; and generate translated code to: split the first operand of the function into an upper half and a lower half; multiply each half of the first operand with a second operand to form an upper half

product and a lower half product; and add the upper half product to the lower half product.

[0219] In an example of one or more embodiments, in the splitting the first operand of the function, the translator module may be configured to: perform a zero-extend operation on the lower half; and perform an arithmetic shift right operation on the upper half.

[0220] In an example of one or more embodiments, the translator module may be further configured to: perform an arithmetic shift right operation on the lower half product.

WHAT IS CLAIMED IS:

1. A method for performing binary translation, comprising:
translating, by a translator module executing on a processor, source instructions to target instructions;
identifying a condition code block in the source instructions, wherein the condition code block includes a plurality of condition bits; and
transforming the condition code block,
wherein the transforming includes performing a pre-execution on the condition code block to resolve the plurality of condition bits in the condition code block to form a pre-resolved condition set.
2. The method of claim 1, further comprising:
encoding the source instructions with the pre-resolved condition set.
3. A method for performing binary translation, comprising:
identifying, by a runtime environment module executing on a processor, an interrupt at an instruction in the target instructions;
retrieving a target address of the interrupted instruction, wherein the target address is for a target instruction set architecture; and
identifying a source address of the interrupted instruction in a recovery table, wherein the target address is paired to the source address in the recovery table.

4. The method of Claim 3, wherein the identifying the source address comprises:

identifying an offset for the source address of the interrupted instruction in the recovery table, wherein the offset is multiplied by an instruction size of the interrupted instruction;

identifying an initial condition state at the source address;

setting a current condition state to the initial condition state;

advancing the current condition state a number of times based on a value of the offset; and

recovering the current condition state.

5. The method of Claim 4, wherein the advancing the current condition state comprises:

setting a variable to 1;

determining whether the offset is equal to the variable;

responsive to the offset equaling the variable, recovering the current condition state;

responsive to the offset not equaling the variable, advancing the condition state of the condition code block;

responsive to advancing the condition state, setting the current condition state to match the advanced condition state;

responsive to setting the current condition state, advancing the variable; and

repeating the determining whether the offset is equal to the variable until the offset is equal to the variable.

6. A method for translating a parallel multiply addition instruction, comprising:

receiving, by a translator module executing on a processor, an instruction to be translated, wherein the instruction calls for a parallel multiply addition function for a number of source vectors; and

generating translated code to perform:

executing the parallel multiply addition function for the number of source vectors; and

determining whether there is an overflow.

7. The method of claim 6, further comprising:

responsive to determining an overflow, setting an overflow flag to indicate there is overflow.

8. A method for translating a parallel multiply subtraction instruction, comprising:

receiving, by a translator module executing on a processor, an instruction to be translated, wherein the instruction calls for a parallel multiply subtraction function for a plurality of source vectors, wherein each vector of the plurality of source vectors has a low element and a high element; and

generating translated code to perform:

performing a bitwise-not operation on a low element of a second source vector;

executing the parallel multiply addition function for the plurality of source vectors to form a result;

adding a low element of the first source vector to the result;

determining whether there is an overflow; and

responsive to determining an overflow, setting the result to a constant value.

9. The method of claim 8, further comprising:

responsive to not determining an overflow, determining whether the high elements were subtracted from the low elements; and

responsive to the high elements being subtracted from the low elements, performing a bitwise-not operation on the result.

10. A method for translating a partial long multiply instruction, comprising:
receiving, by a translator module executing on a processor, an instruction to be translated, wherein the instruction calls for a partial long multiply function for a plurality of source vectors, and wherein the partial long multiply function has a first operand; and
generating translated code to perform:

splitting the first operand of the function into an upper half and a lower half;

multiplying each half of the first operand with a second operand to form an upper half product and a lower half product; and

adding the upper half product to the lower half product.

11. The method of claim 10, wherein the splitting the first operand of the function comprises:

performing a zero-extend operation on the lower half; and

performing an arithmetic shift right operation on the upper half.

12. The method of claims 10 or 11, further comprising:

performing an arithmetic shift right operation on the lower half product.

13. At least one computer-readable medium comprising one or more instructions that when executed by a processor:

- translate source instructions to target instructions;
- identify a condition code block in the source instructions, wherein the condition code block includes a plurality of condition bits; and
- transform the condition code block,

wherein the transforming includes performing a pre-execution on the condition code block to resolve the plurality of condition bits in the condition code block to form a pre-resolved condition set.

14. The at least one computer-readable medium of claim 13, further comprising one or more instructions that when executed by the processor:

- encode the source instructions with the pre-resolved condition set.

15. At least one computer-readable medium comprising one or more instructions that when executed by a processor:

- identify an interrupt at an instruction in target instructions;
- retrieve a target address of the interrupted instruction, wherein the target address is for a target instruction set architecture; and
- identify a source address of the interrupted instruction in a recovery table, wherein the target address is paired to the source address in the recovery table.

16. The at least one computer-readable medium of claim 15, wherein identifying the source address comprises one or more instructions that when executed by the processor:

- identify an offset for the source address of the interrupted instruction in the recovery table, wherein the offset is multiplied by an instruction size of the interrupted instruction;

- identify an initial condition state at the source address;

- set a current condition state to the initial condition state;

- advance the current condition state a number of times based on a value of the offset; and

- recover the current condition state.

17. The at least one computer-readable medium of claim 16, wherein the advancing the current condition state comprises one or more instructions that when executed by the processor:

- set a variable to 1;

- determine whether the offset is equal to the variable;

- responsive to the offset equaling the variable, recover the current condition state;

- responsive to the offset not equaling the variable, advance the condition state of the condition code block;

- responsive to advancing the condition state, set the current condition state to match the advanced condition state;

- responsive to setting the current condition state, advance the variable; and

- repeat determining whether the offset is equal to the variable until the offset is equal to the variable.

18. At least one computer-readable medium comprising one or more instructions that when executed by a processor:

receive an instruction to be translated, wherein the instruction calls for a parallel multiply addition function for a number of source vectors; and

generate translated code to:

execute the parallel multiply addition function for the number of source vectors; and

determine whether there is an overflow.

19. The at least one computer-readable medium of claim 18, further comprising one or more instructions that when executed by the processor:

responsive to determining an overflow, set an overflow flag to indicate there is overflow.

20. At least one computer-readable medium comprising one or more instructions that when executed by a processor:

receive an instruction to be translated, wherein the instruction calls for a parallel multiply subtraction function for a plurality of source vectors, wherein each vector of the plurality of source vectors has a low element and a high element; and

generate translated code to:

perform a bitwise-not operation on a low element of a second source vector;

execute the parallel multiply addition function for the plurality of source vectors to form a result;

add a low element of the first source vector to the result;

determine whether there is an overflow; and

responsive to determining an overflow, set the result to a constant value.

21. The at least one computer-readable medium of claim 20, further comprising one or more instructions that when executed by the processor:

responsive to not determining an overflow, determine whether the high elements were subtracted from the low elements; and

responsive to the high elements being subtracted from the low elements, perform a bitwise-not operation on the result.

22. At least one computer-readable medium comprising one or more instructions that when executed by a processor:

receive an instruction to be translated, wherein the instruction calls for a partial long multiply function for a plurality of source vectors, and wherein the partial long multiply function has a first operand; and

generate translated code to:

split the first operand of the function into an upper half and a lower half;

multiply each half of the first operand with a second operand to form an upper half product and a lower half product; and

add the upper half product to the lower half product.

23. The at least one computer-readable medium of claim 22, wherein the splitting the first operand of the function comprises one or more instructions that when executed by the processor:

perform a zero-extend operation on the lower half; and

perform an arithmetic shift right operation on the upper half.

24. The at least one computer-readable medium of claims 22 or 23, further comprising one or more instructions that when executed by the processor:

perform an arithmetic shift right operation on the lower half product.

25. An apparatus, comprising:
a translator module configured to:
translate source instructions to target instructions; and
identify a condition code block in the source instructions, wherein the
condition code block includes a plurality of condition bits; and
an optimizer module configured to transform the condition code block,
wherein the transforming includes performing a pre-execution on the condition
code block to resolve the plurality of condition bits in the condition code block to form a
pre-resolved condition set.
26. The apparatus of claim 25, wherein the optimizer module is further
configured to:
encode the source instructions with the pre-resolved condition set.

27. An apparatus, comprising:
a recovery table configured to store a number of target addresses and a number of source addresses; and
a runtime environment module configured to:
identify an interrupt at an instruction in target instructions;
retrieve a target address of the interrupted instruction, wherein the target address is for a target instruction set architecture; and
identify a source address of the interrupted instruction in the recovery table, wherein the target address of the number of target addresses is paired to the source address of the number of source addresses in the recovery table.

28. The apparatus of claim 27, wherein in the identifying the source address, the runtime environment module is configured to:
identify an offset for the source address of the interrupted instruction in the recovery table, wherein the offset is multiplied by an instruction size of the interrupted instruction;
identify an initial condition state at the source address;
set a current condition state to the initial condition state;
advance the current condition state a number of times based on a value of the offset; and
recover the current condition state.

29. The apparatus of claim 28, wherein in the advancing the current condition state, the runtime environment module is configured to:

set a variable to 1;

determine whether the offset is equal to the variable;

responsive to the offset equaling the variable, recover the current condition state;

responsive to the offset not equaling the variable, advance the condition state of the condition code block;

responsive to advancing the condition state, set the current condition state to match the advanced condition state;

responsive to setting the current condition state, advance the variable; and

repeat determining whether the offset is equal to the variable until the offset is equal to the variable.

30. An apparatus, comprising:
a translator module configured to:
 receive an instruction to be translated, wherein the instruction calls for a
 parallel multiply addition function for a number of source vectors; and
 generate translated code to:
 execute the parallel multiply addition function for the number of
 source vectors; and
 determine whether there is an overflow.
31. The apparatus of claim 30, wherein the translator module is further
configured to:
 responsive to determining an overflow, set an overflow flag to indicate
 there is overflow.

32. An apparatus, comprising:

a translator module configured to:

receive an instruction to be translated, wherein the instruction calls for a parallel multiply subtraction function for a plurality of source vectors, wherein each vector of the plurality of source vectors has a low element and a high element; and

generate translated code to:

perform a bitwise-not operation on a low element of a second source vector;

execute the parallel multiply addition function for the plurality of source vectors to form a result;

add a low element of the first source vector to the result;

determine whether there is an overflow; and

responsive to determining an overflow, set the result to a constant value.

33. The apparatus of claim 32, wherein the translator module is further configured to:

responsive to not determining an overflow, determine whether the high elements were subtracted from the low elements; and

responsive to the high elements being subtracted from the low elements, perform a bitwise-not operation on the result.

34. An apparatus, comprising:

a translator module configured to:

receive an instruction to be translated, wherein the instruction calls for a partial long multiply function for a plurality of source vectors, and wherein the partial long multiply function has a first operand; and

generate translated code to:

split the first operand of the function into an upper half and a lower half;

multiply each half of the first operand with a second operand to form an upper half product and a lower half product; and

add the upper half product to the lower half product.

35. The apparatus of claim 34, wherein in the splitting the first operand of the function, the translator module is configured to:

perform a zero-extend operation on the lower half; and

perform an arithmetic shift right operation on the upper half.

36. The apparatus of claims 34 or 35, wherein the translator module is further configured to:

perform an arithmetic shift right operation on the lower half product.

1/9

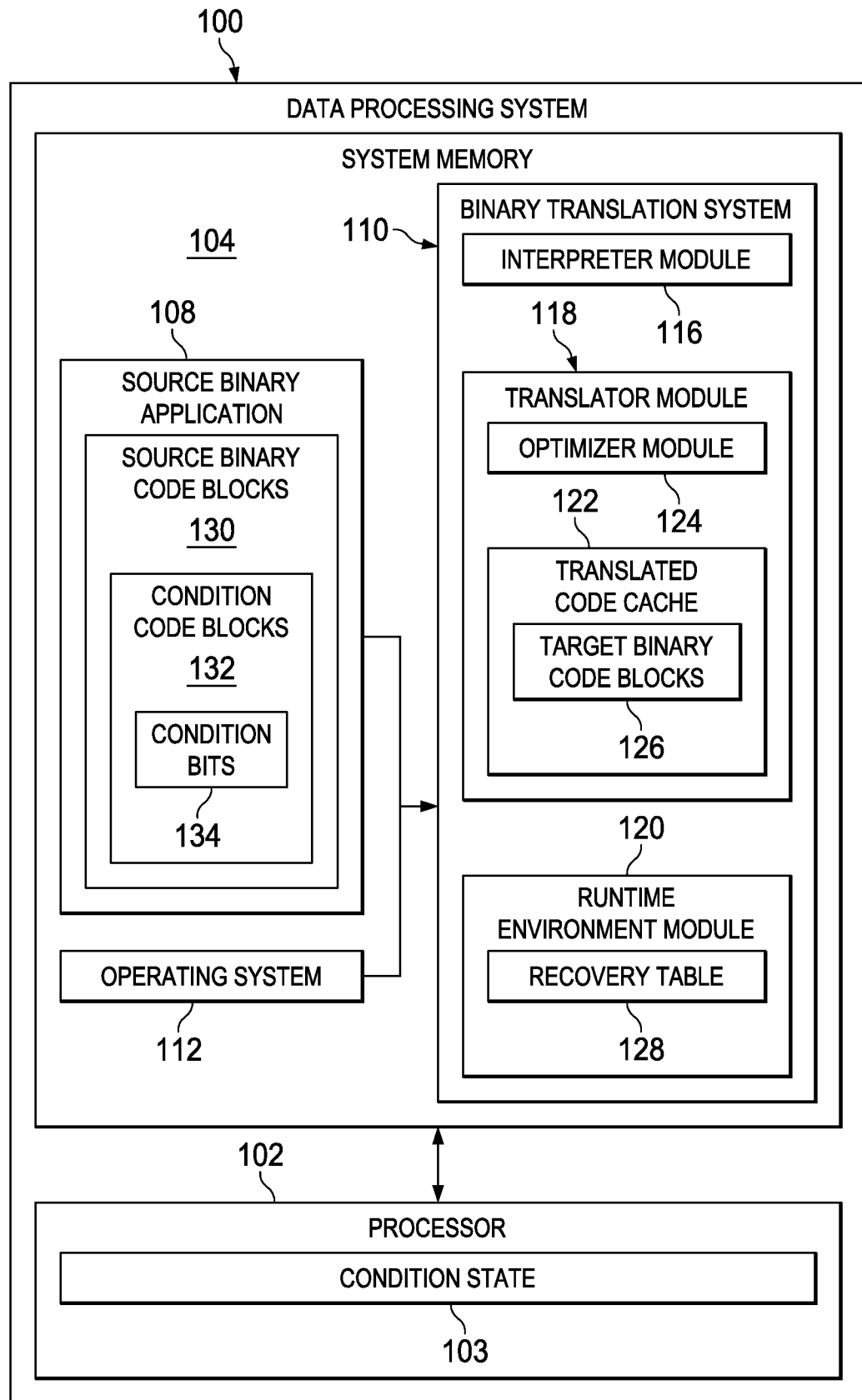


FIG. 1

2/9

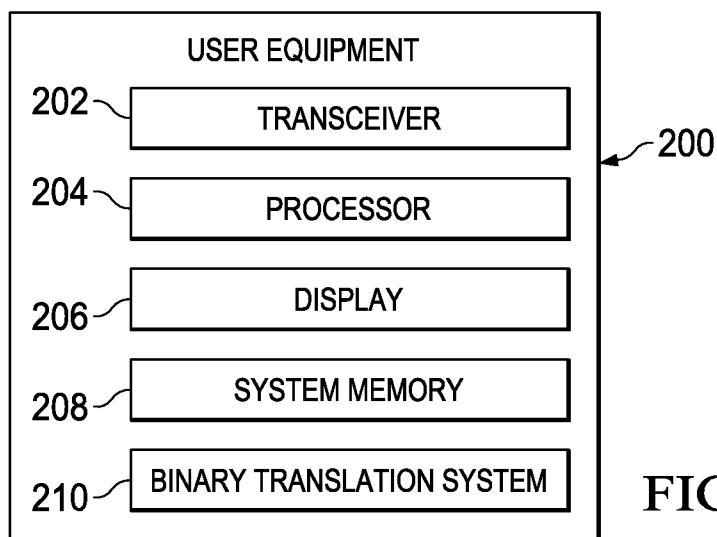


FIG. 2

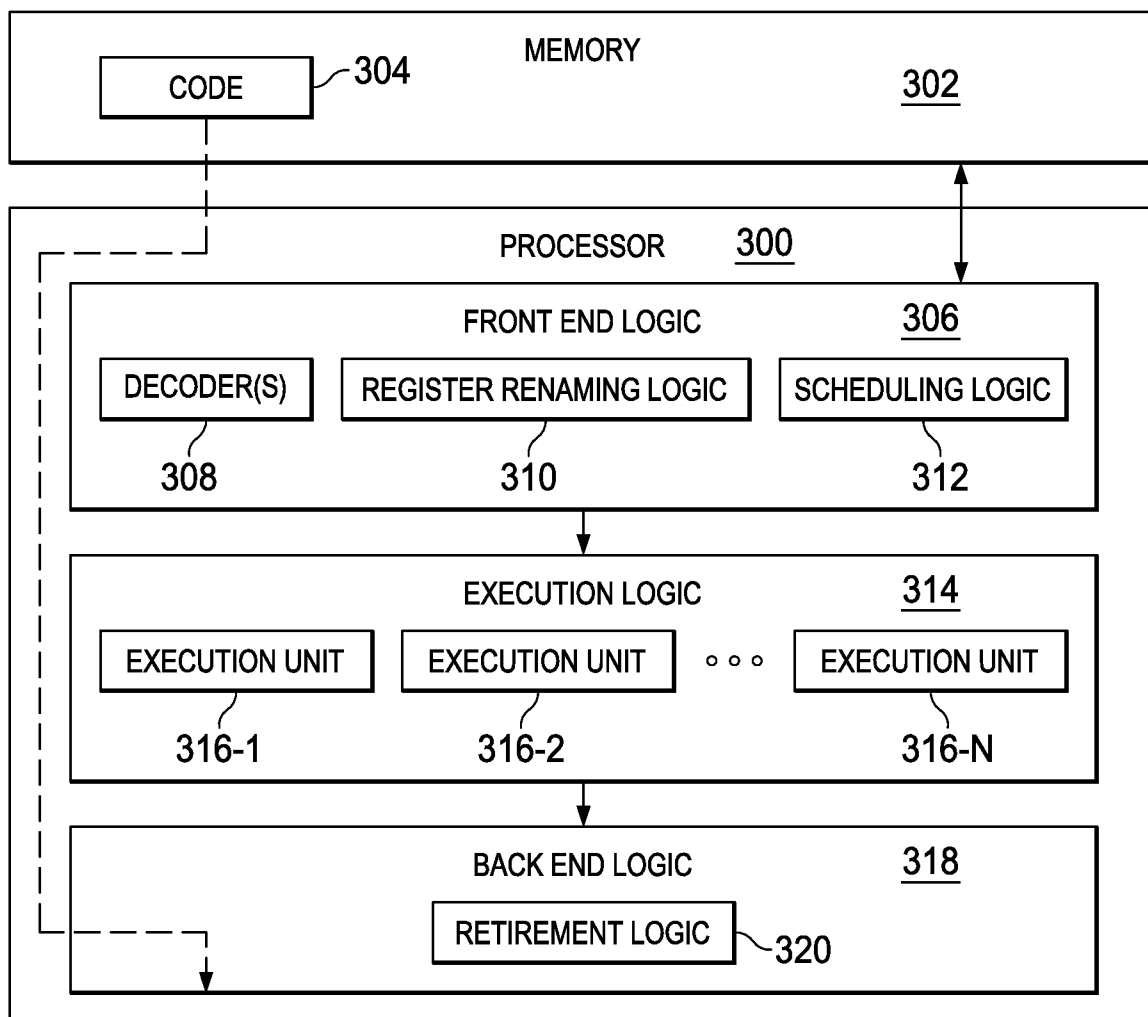


FIG. 3

3/9

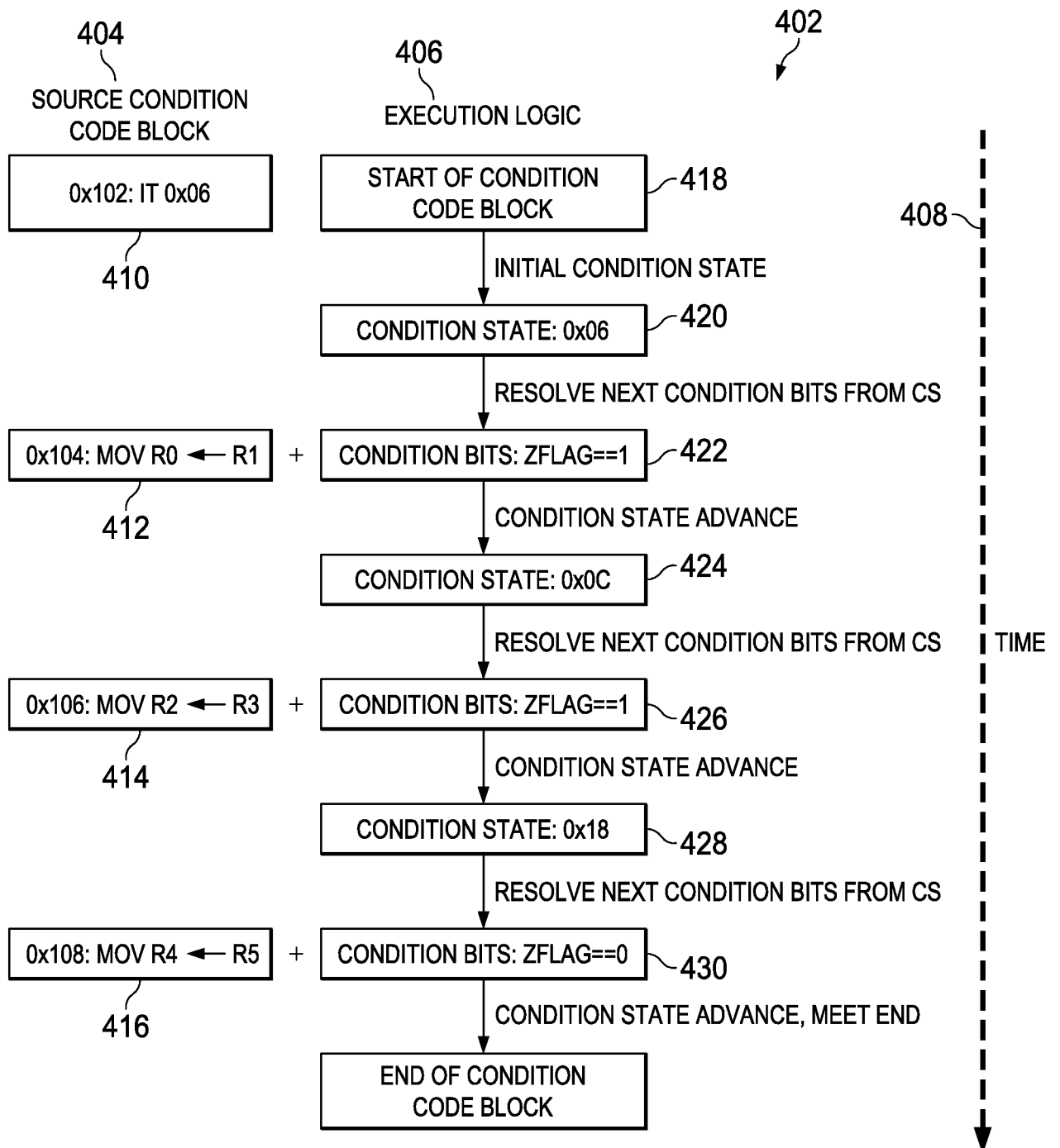


FIG. 4

4/9

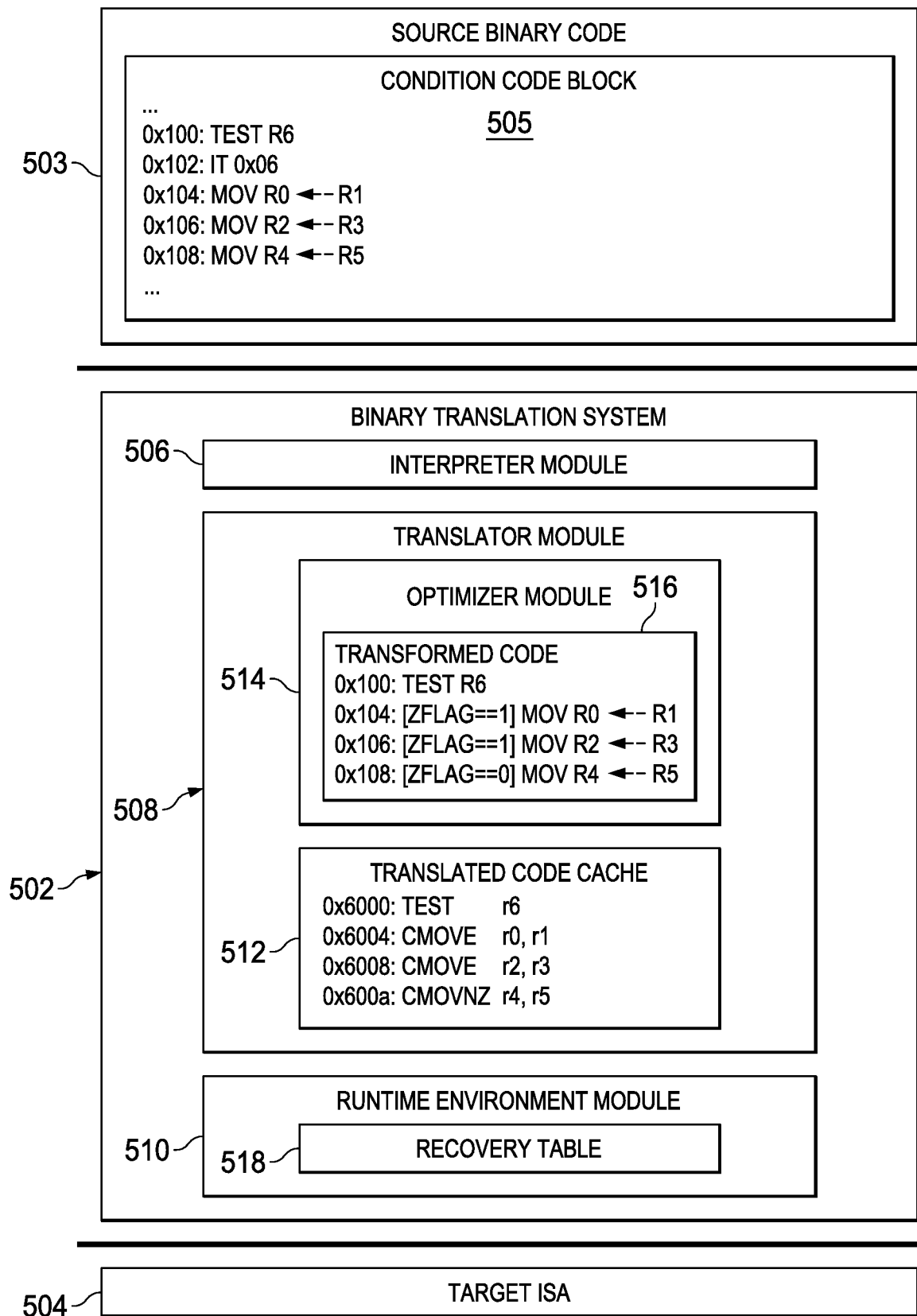


FIG. 5

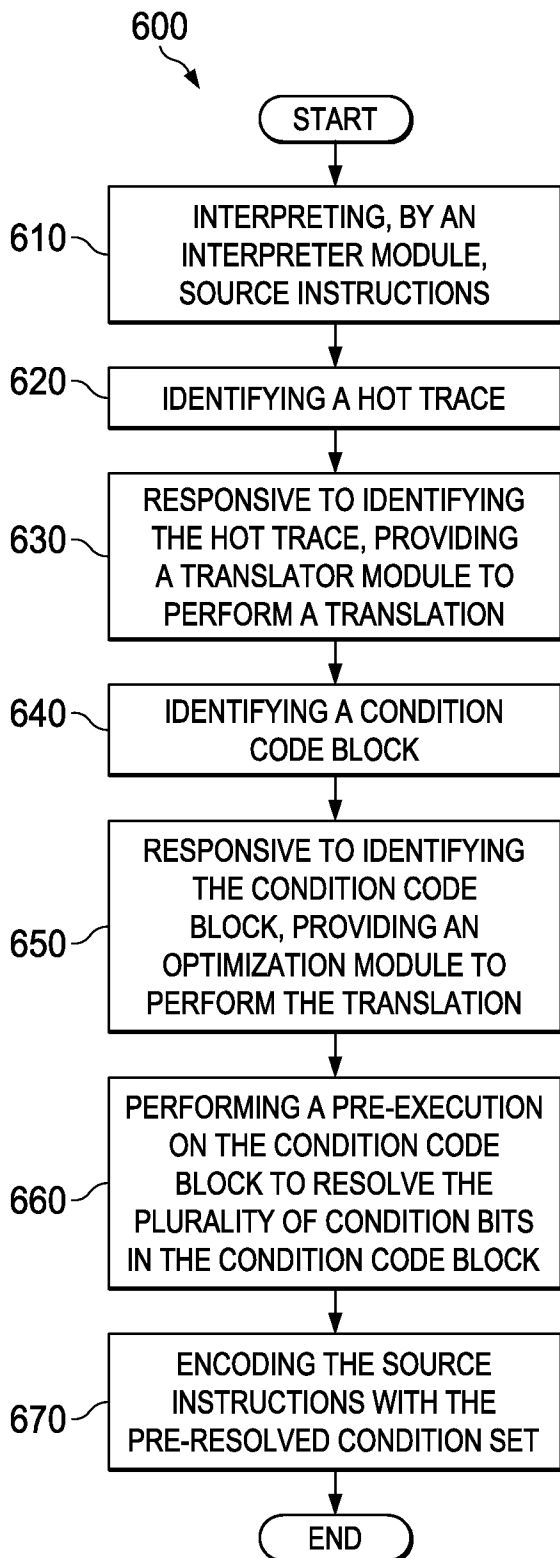


FIG. 6

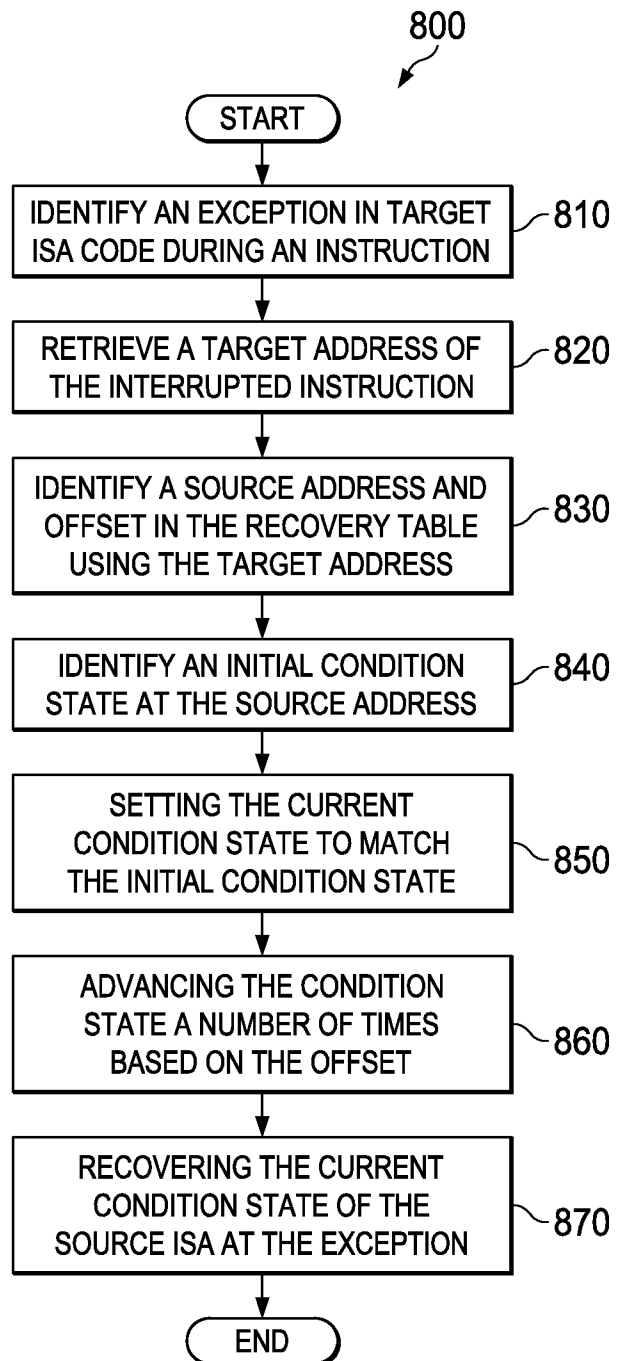


FIG. 8

6/9

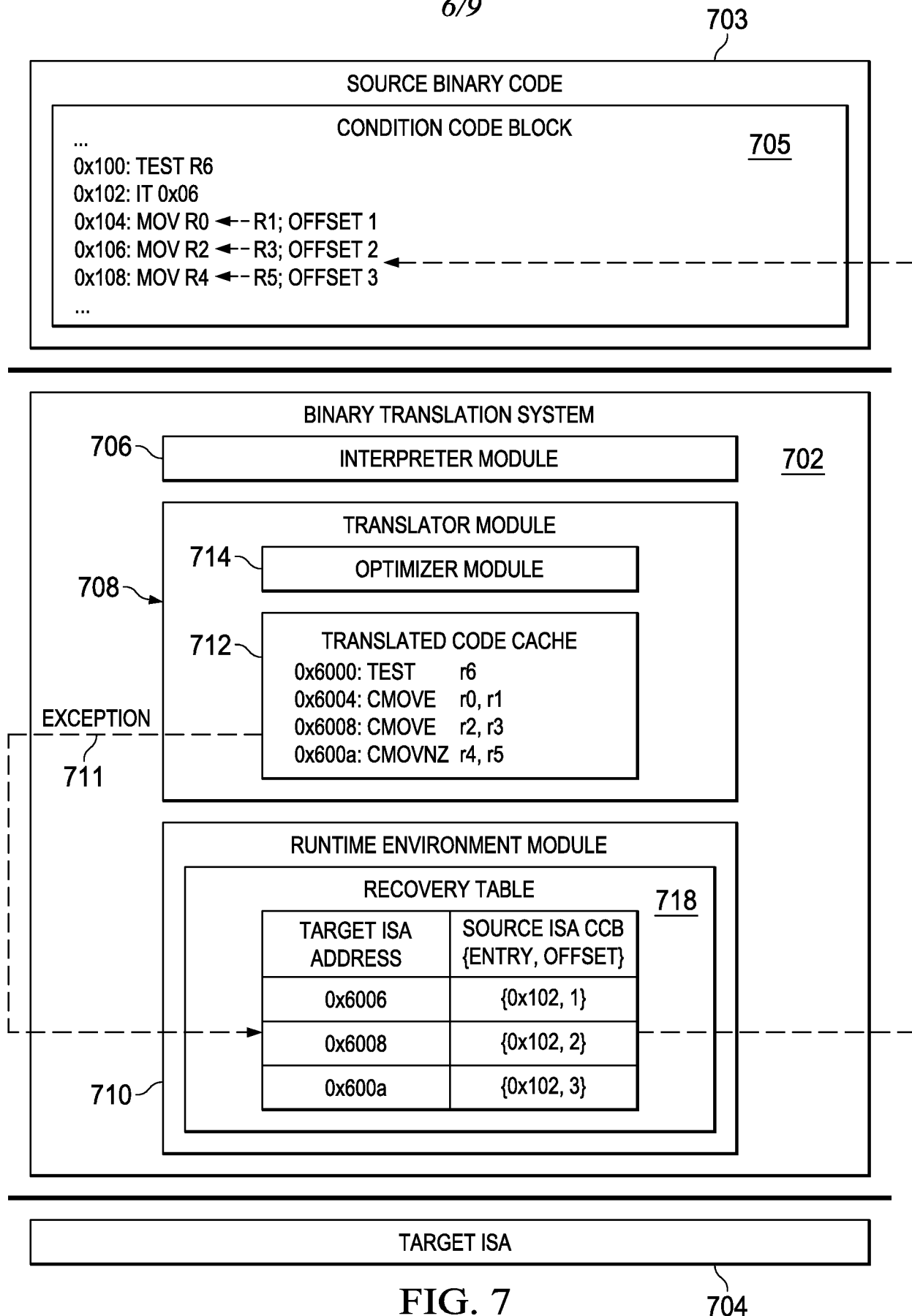


FIG. 7

7/9

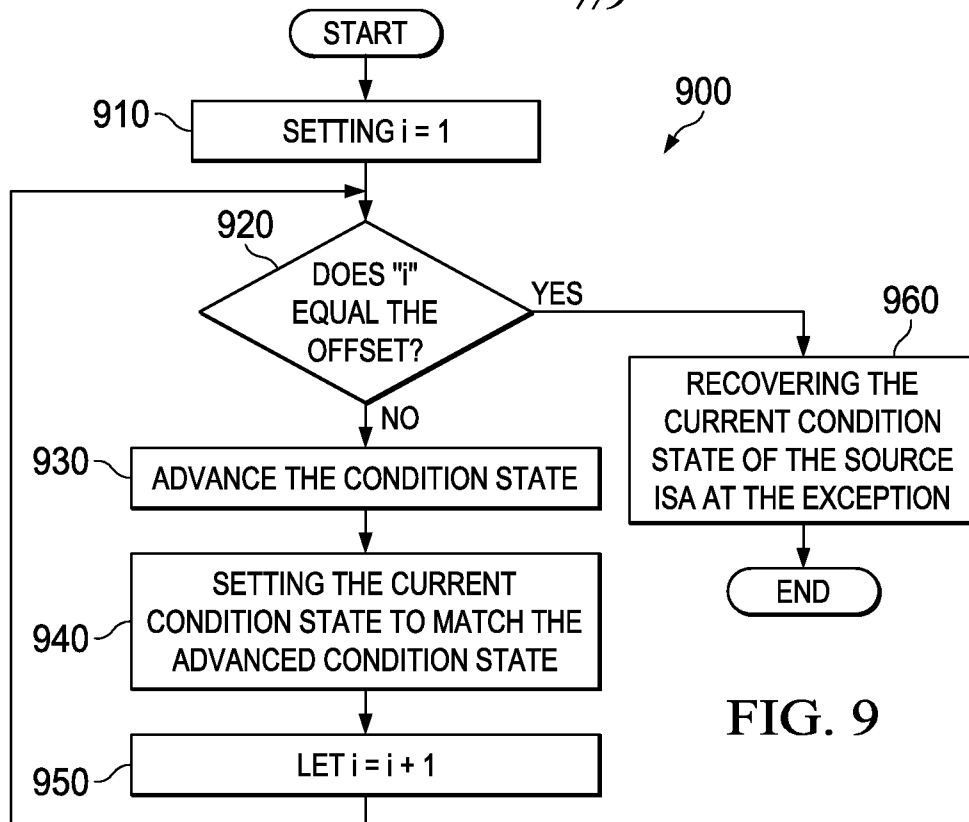


FIG. 9

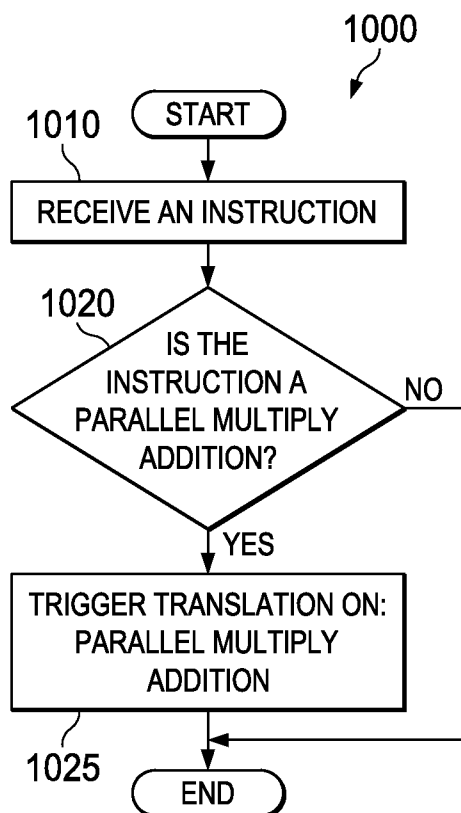


FIG. 10A

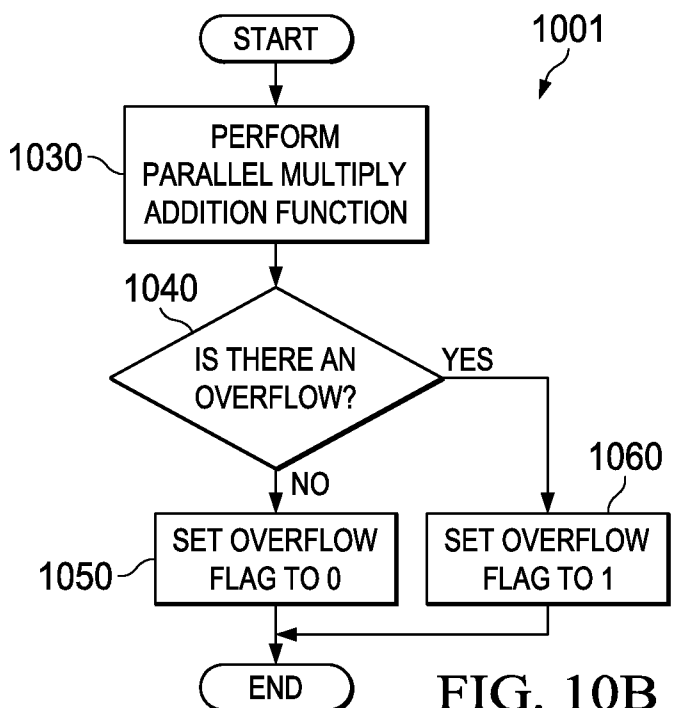


FIG. 10B

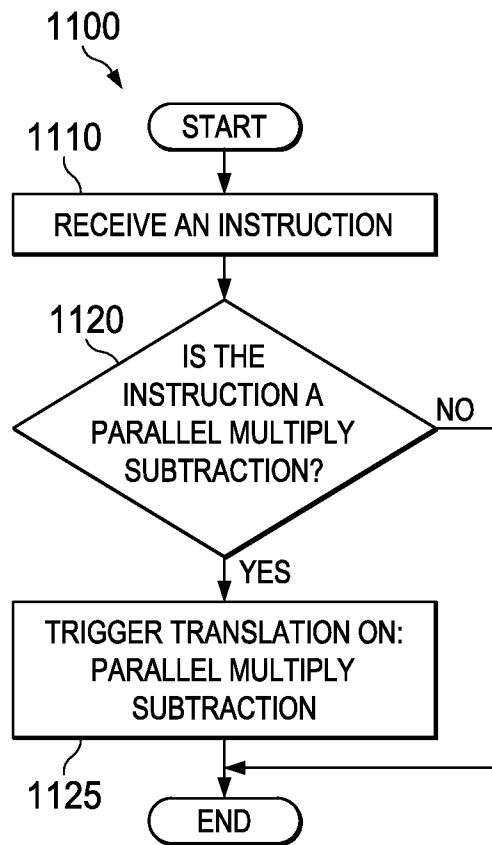


FIG. 11A

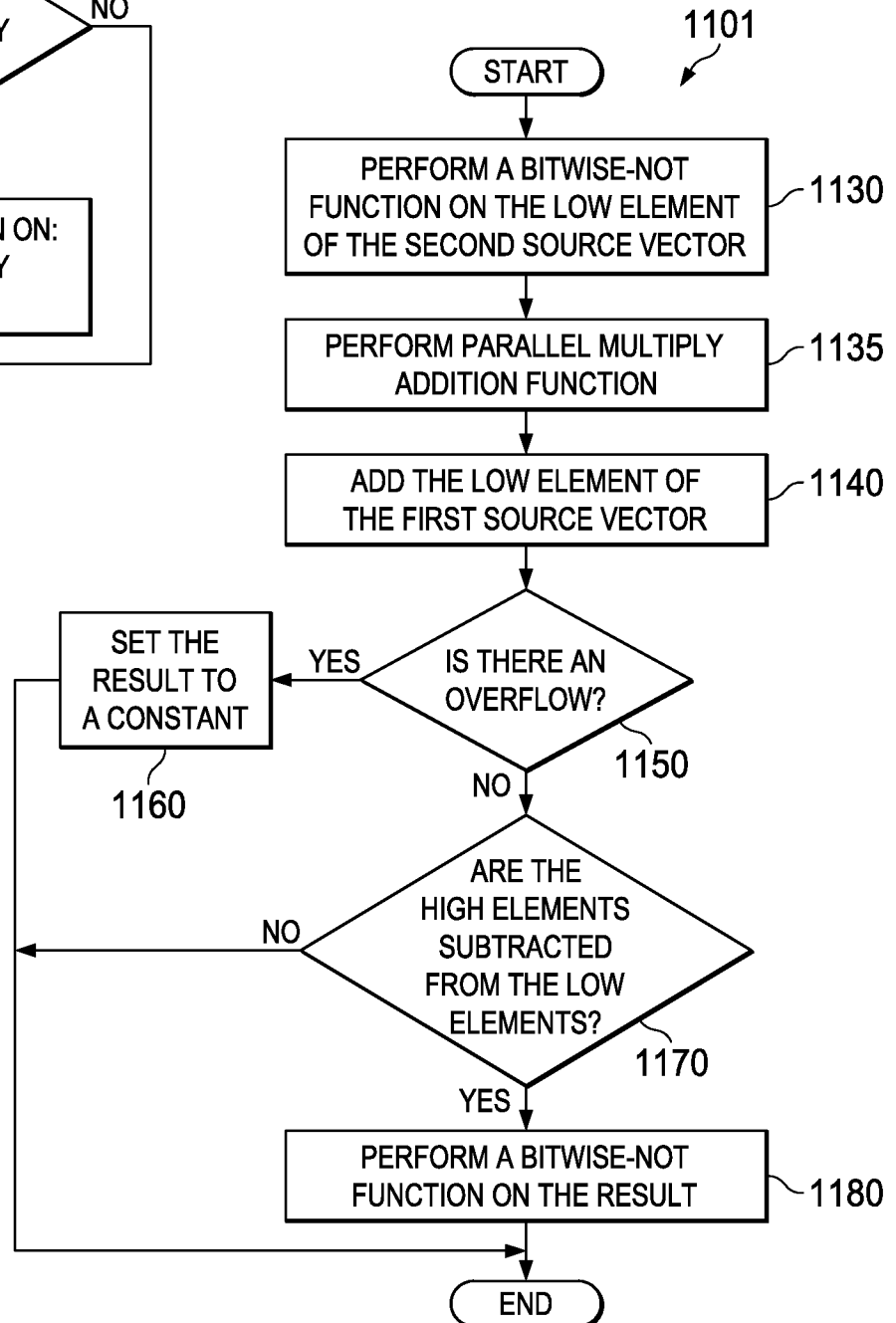


FIG. 11B

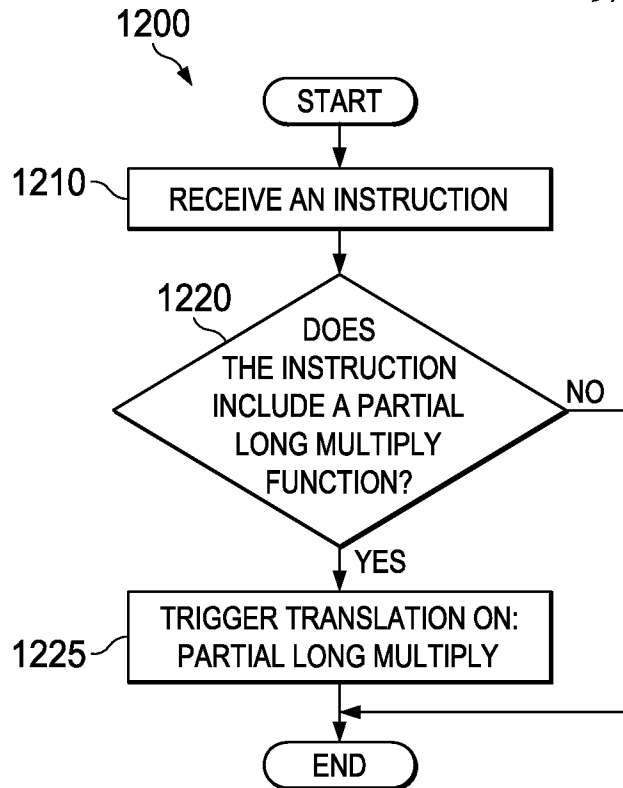


FIG. 12A

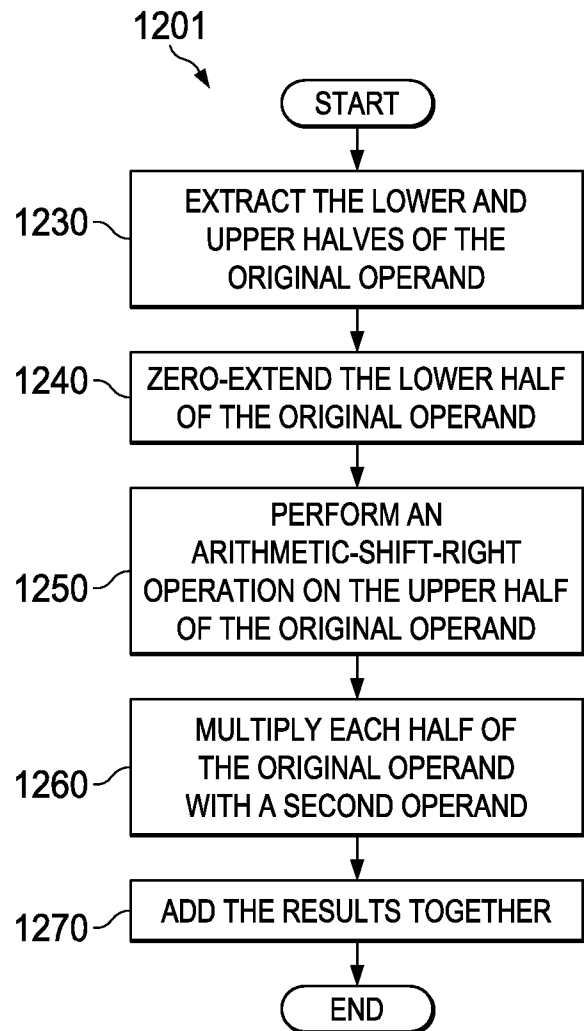


FIG. 12B

INTERNATIONAL SEARCH REPORT

International application No.

PCT/CN2012/081728

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/45 (2006.01) 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)

IPC: G06F, H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

WPI, EPODOC, CNPAT, CNKI: binary, translat+, instruction, source, target, overflow, add+, multiply+, condition, bit?, set+, processor

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US2011/0307876A1 (OTTONI, Guilherme D. et al.) 15 Dec. 2011(15.12.2011) see description, paragraphs 0012-0050 and figures 1-3	1-36
A	US2008/0172657A1 (THE BOARD OF TRUSTEES OF THE LELAND STANFORD JUNIOR UNIVERSITY) 17 Jul. 2008(17.07.2008) see the whole document	1-36
A	US2008/0172661A1 (HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P.) 17 Jul. 2008 (17.07.2008) see the whole document	1-36
A	CN101706734A (INSTITUTE OF COMPUTING TECHNOLOGY CHINESE ACADEMY OF SCIENCES) 12 May 2010(12.05.2010) see the whole document	1-36

☐ Further documents are listed in the continuation of Box C.☒ See patent family annex.

* Special categories of cited documents:	“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
“A” document defining the general state of the art which is not considered to be of particular relevance	“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
“E” earlier application or patent but published on or after the international filing date	“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
“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)	“&” document member of the same patent family
“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	

Date of the actual completion of the international search 08 Jun. 2013(08.06.2013)	Date of mailing of the international search report 27 Jun. 2013 (27.06.2013)
Name and mailing address of the ISA/CN The State Intellectual Property Office, the P.R.China 6 Xitucheng Rd., Jimen Bridge, Haidian District, Beijing, China 100088 Facsimile No. 86-10-62019451	Authorized officer CHEN, Long Telephone No. (86-10)62414059

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/CN2012/081728

Patent Documents referred in the Report	Publication Date	Patent Family	Publication Date
US2011/0307876A1	15.12.2011	EP2400388A1	28.12.2011
		KR20110136749A	21.12.2011
		WO2011159739A2	22.12.2011
		JP2012003761A	05.01.2012
		CN102298514A	28.12.2011
		WO2011159739A3	05.04.2012
		INDELNP201110106E	09.11.2012
		HK1165039A0	28.09.2012
		KR20130018925A	25.02.2013
US2008/0172657A1	17.07.2008	NONE	
US2008/0172661A1	17.07.2008	INCHE200700079A	28.11.2008
CN101706734A	12.05.2010	NONE	