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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 November 2012 (08.11.2012)

WIPO | PCT

(10) International Publication Number
WO 2012/151334 A1

(51) **International Patent Classification**:
G06F 9/30 (2006.01)

(21) **International Application Number**:
PCT/US2012/036199

(22) **International Filing Date**:
2 May 2012 (02.05.2012)

(25) **Filing Language**: English

(26) **Publication Language**: English

(30) **Priority Data**:
13/099,463 3 May 2011 (03.05.2011) US

(71) **Applicant** *(for all designated States except US)*: **QUALCOMM INCORPORATED** [US/US]; ATTN: INTERNATIONAL IP ADMINISTRATION, 5775 Morehouse Drive, San Diego, California 92121 (US).

(72) **Inventors; and**

(75) **Inventors/Applicants** *(for US only)*: **LARIN, Sergei** [RU/US]; 5775 Morehouse Drive, San Diego, California 92121 (US). **CODRESCU, Lucian** [US/US]; 5775 Morehouse Drive, San Diego, California 92121 (US). **DAS GUPTA, Anshuman** [IN/US]; 5775 Morehouse Drive, San Diego, California 92121 (US).

(74) **Agent**: **TALPALATSKY, Sam**; 5775 Morehouse Drive, San Diego, California 92121 (US).

(81) **Designated States** *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, 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, 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, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

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

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

Published:

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

[Continued on next page]

(54) **Title**: METHODS AND APPARATUS FOR STORAGE AND TRANSLATION OF ENTROPY ENCODED SOFTWARE EMBEDDED WITHIN A MEMORY HIERARCHY

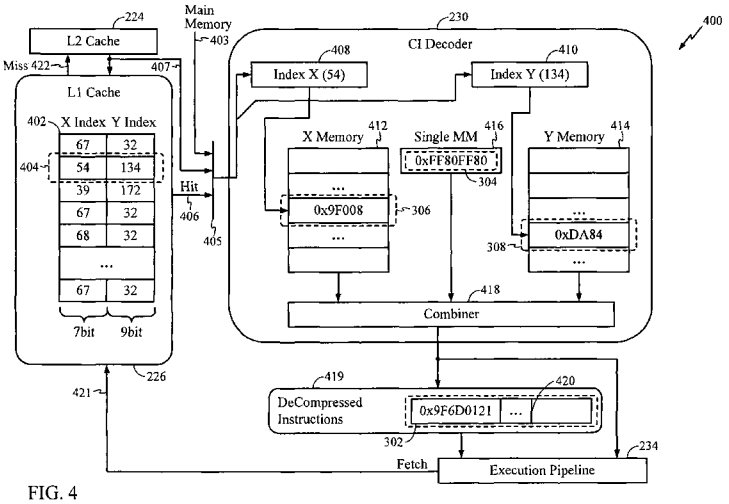

FIG. 4

(57) **Abstract**: A custom entropy bounded encoding in an X-index and Y-index format is generated for a segment of program code, along with a custom decoding dictionary made up of an X pattern memory and a Y pattern memory. In run time decoding, a mix mask is used with an X pattern selected from the X pattern memory according to the X-index and with a Y pattern selected from the Y pattern memory according to the Y-index to determine an executable instruction. The mix mask identifying the order of bits to combine from the X pattern and the Y pattern. Appropriate hardware implementation and placement of the decoding mechanism and address translation is described during execution of an encoded code segment. Methods, including a genetic process, are also described to determine the X-index, the Y-index, the X patterns, the Y patterns, and one or more mix masks.

WO 2012/151334 A1

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

METHODS AND APPARATUS FOR STORAGE AND TRANSLATION OF ENTROPY ENCODED SOFTWARE EMBEDDED WITHIN A MEMORY HIERARCHY

Field of the Invention

{0001} The present invention relates generally to processors having compressed instruction sets for improving code density in embedded software, and more specifically to techniques for generating compressed instructions, storing the compressed instructions, and translating the compressed instructions.

Background of the Invention

{0002} Many portable products, such as cell phones, laptop computers, personal digital assistants (PDAs) or the like, require the use of a processor executing a program supporting communication and multimedia applications. The processing system for such products includes one or more processors, each with storage for instructions, input operands, and results of execution. For example, the instructions, input operands, and results of execution for a processor may be stored in a hierarchical memory subsystem consisting of a general purpose register file, multi-level instruction caches, data caches, and a system memory.

{0003} In order to provide high code density a native instruction set architecture (ISA) may be used having two instruction formats, such as a 16-bit instruction format that is a subset of a 32-bit instruction format. In many cases, a fetched 16-bit instruction is transformed by a processor into a 32-bit instruction prior to or in a decoding process which allows the execution hardware to be designed to only support the 32-bit instruction format. The use of 16-bit instructions that are a subset of 32-bit instructions is a restriction that limits the amount of

information that can be encoded into a 16-bit format. For example, a 16-bit instruction format may limit the number of addressable source operand registers and destination registers that may be specified. A 16-bit instruction format, for example, may use 3-bit or 4-bit register file address fields, while a 32-bit instruction may use 5-bit fields. Processor pipeline complexity may also increase if the two formats are intermixed in a program due in part to instruction addressing restrictions, such as, branching to 16-bit and 32-bit instructions. Also, requirements for code compression vary from program to program making a fixed 16-bit instruction format chosen for one program less advantageous for use by a different program. In this regard, legacy code for existing processors may not be able to effectively utilize the two instruction formats to significantly improve code density and meet, in many cases, real time requirements. These and other restrictions limit the effectiveness of reduced size instructions having fields that are subsets of fields used in the standard size instructions.

SUMMARY OF THE DISCLOSURE

{0004} Among its several aspects, the present invention addresses a need to decouple a program using a native instruction set from a compressed program using a compressed instruction set. The techniques addressed herein allow highly efficient utilization of storage and a transmission conduit for embedded software without affecting the software's execution time and efficiency.

{0005} To such ends, an embodiment of the invention applies a method for decompressing compressed variable length instructions. Compact fixed length instructions are fetched from a system memory configured to store compressed variable length instructions, wherein each compact fixed length instruction comprises an X-index and a Y-index. For each

compact fixed length instruction an X-bit pattern is fetched from an X memory using the X-index and a Y-bit pattern is fetched from a Y memory using the Y-index. The X-bit pattern is combined with the Y-bit pattern based on a mix mask associated with each compact fixed length instruction into a decompressed variable length instruction, wherein a format length of the decompressed variable length instruction is determined by the associated mix mask.

{0006} Another embodiment of the invention addresses an apparatus for translating compressed instructions stored in a virtual memory system. A paged instruction cache is configured to store pages of compressed instructions intermixed with pages of uncompressed instructions. An instruction translation look aside buffer (TLB) is configured to store an address translation entry that identify a page in the paged cache as storing compressed instructions.

{0007} Another embodiment of the invention addresses a system for translating compressed instructions to an executable format. A level 2 cache is configured to store an X-index and a Y-index for each compressed instruction. A translation unit is configured to receive compressed instructions from the level 2 cache, for each received compressed instruction to select an X-bit pattern and a Y-bit pattern from a translation memory using the X-index and the Y-index, and to use a program specified mix mask for combining the selected X-bit pattern and Y-bit pattern into a native instruction format. A level 1 cache is configured to store the native instruction format for each compressed instruction.

{0008} Another embodiment of the invention addresses a method of determining a mix mask for efficiently translating compressed instructions. Pairs of mix masks represented as genes from a seed population of mix masks are bred to produce pairs of offspring mix masks. The offspring mix masks are mutated to produce mutated offspring mix masks that update the seed population. A mix mask is determined from the updated seed population that provides a

high level of compression, wherein patterns of bits are combined according to the determined mix mask to translate compressed instructions of a program to executable form.

{0009} A more complete understanding of the present invention, as well as further features and advantages of the invention, will be apparent from the following Detailed Description and the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

{0010} FIG. 1 is a block diagram of an exemplary wireless communication system in which an embodiment of the invention may be advantageously employed;

{0011} FIG. 2 is a system for code compression designed for run time decompression in accordance with the present invention;

{0012} FIG. 3 illustrates exemplary elements of an instruction partition process that splits an instruction based on a mix mask into an X pattern and a Y pattern with byte overlap pad bits in accordance with the present invention;

{0013} FIG. 4 is a decompressor system having programs stored in compressed form in main memory and a level 2 cache that are decompressed to be stored in a level 1 cache in accordance with the present invention;

{0014} FIG. 5A illustrates exemplary elements of a second instruction partition process that splits a second instruction based on a second mix mask into a second X pattern and a second Y pattern with byte overlap pad bits in accordance with the present invention;

{0015} FIG. 5B illustrates an exemplary storage arrangement for the Ax, Ay patterns of FIG. 3 and the Bx, By patterns of FIG. 5A in an X memory and a Y memory having three different masks Amm ≠ Bmm ≠ Cmm in accordance with the present invention;

{0016} FIG. 5C is a decompressor system having programs stored in compressed form in main memory and a level 2 cache that are decompressed using multiple mix masks and index compression to be stored in a level 1 cache in uncompressed form in accordance with the present invention;

{0017} FIG. 6 illustrates a VLIW packet compression format in accordance with the present invention;

{0018} FIG. 7 illustrates a paged instruction translation look aside buffer (ITLB) and memory organization having compressed pages and uncompressed pages in accordance with the present invention;

{0019} FIG. 8A illustrates a paged decompressor system for decompressing compressed instruction pages and accessing uncompressed instruction pages in accordance with the present invention;

{0020} FIG. 8B illustrates an exemplary decompression state diagram that illustrates the state of L2 cache compressed page instructions and L1 Icache decompressed instructions for execution on processor pipeline;

{0021} FIGs. 9A-9C illustrates a genetic mix mask determination process based on a variation of a genetic algorithm in accordance with the present invention;

{0022} FIG. 9D illustrates implementation of crossover algorithm in accordance with the present invention;

{0023} FIG. 9E illustrates an implementation of mutation algorithm in accordance with the present invention;

{0024} FIG. 10 illustrates an exemplary current cost process in accordance with the present invention;

{0025} FIG. 11 illustrates a weighted Hamming heuristic in accordance with the present invention;

{0026} FIG. 12 illustrates an exhaustive search X/Y table compaction process with pad-sort in accordance with the present invention;

{0027} FIG. 13 illustrates a symbol insertion into compressed storage process in accordance with the present invention; and

{0028} FIG. 14 illustrates the triangular X/Y table compaction with pad-sort process in accordance with the present invention.

DETAILED DESCRIPTION

{0029} The present invention will now be described more fully with reference to the accompanying drawings, in which several embodiments of the invention are shown. This invention may, however, be embodied in various forms and should not be construed as limited to the embodiments set forth herein. Rather, these embodiments are provided so that this disclosure will be thorough and complete, and will fully convey the scope of the invention to those skilled in the art.

{0030} Computer program code or "program code" for being operated upon or for carrying out operations according to the teachings of the invention may be initially written in a high level programming language such as C, C++, JAVA®, Smalltalk, JavaScript®, Visual Basic®, TSQL, Perl, or in various other programming languages. A source program or source code written in one of these languages is compiled to a target processor architecture by converting the high level program code into a native assembler program using instructions encoded in a native instruction format. Programs for the target processor architecture may also

be written directly in a native assembler language. The native assembler program uses instruction mnemonic representations of machine level binary instructions. Program code or computer readable medium produced by a compiler or a human programmer as used herein refers to machine language code such as object code whose format is understandable by a processor.

{0031} FIG. 1 illustrates an exemplary wireless communication system 100 in which an embodiment of the invention may be advantageously employed. For purposes of illustration, FIG. 1 shows three remote units 120, 130, and 150 and two base stations 140. It will be recognized that common wireless communication systems may have many more remote units and base stations. Remote units 120, 130, 150, and base stations 140 which include hardware components, software components, or both as represented by components 125A, 125C, 125B, and 125D, respectively, have been adapted to embody the invention as discussed further below. FIG. 1 shows forward link signals 180 from the base stations 140 to the remote units 120, 130, and 150 and reverse link signals 190 from the remote units 120, 130, and 150 to the base stations 140.

{0032} In FIG. 1, remote unit 120 is shown as a mobile telephone, remote unit 130 is shown as a portable computer, and remote unit 150 is shown as a fixed location remote unit in a wireless local loop system. By way of example, the remote units may alternatively be cell phones, pagers, walkie talkies, handheld personal communication system (PCS) units, portable data units such as personal digital assistants, or fixed location data units such as meter reading equipment. Although FIG. 1 illustrates remote units according to the teachings of the disclosure, the disclosure is not limited to these exemplary illustrated units. Embodiments of the invention may be suitably employed in any processor system.

{0033} FIG. 2 is a compression system 200 for code compression designed for run time decompression in accordance with the present invention. The compression system 200 includes source code as described above and binary library files in uncompressed form in source code and library files 204 which comprise the current program application being compiled. The compression system 200 also includes a compiler and linker 206, optional profile feedback information 208, which is used to generate linked executable code 210 based on native instruction set architecture (ISA) formats and supporting data sections 212. The native ISA is represented by a fixed, uncompressed format and can represent a variety of approaches, including, for example, fixed 64 or 32 or 16-bit encodings and a mixture of such encodings. The native ISA is developed for general utility and not specifically tailored for a current application at hand. By maintaining fixed word boundaries, such as 32-bit instruction word boundaries, an addressing model that supports only fixed word addresses for branches, calls, returns, and the like may be used even though 16-bit and 32-bit instructions may be mixed together in the code.

{0034} Instructions selected from such an ISA may be compressed and tailored to the current application while maintaining addressability of the code and guaranteeing fast, fixed latency decompression time. Such compression may be automated to produce the compressed code in linear time. The original ISA by its very generalized nature has low informational entropy, which is increased in accordance with the present invention by producing a custom entropy bounded encoding for the given source code and library files 204. The linked executable code 210 is provided as input to a translation tool 216 which generates compressed code 218 and decoding tables 220. The compressed code 218 and the supporting data sections 212 are stored in a static storage device 214, such as a hard disk, optical disk, flash memory of an embedded device or other such storage medium from which selected code may be down loaded to a

processor complex 203 for execution. The processor complex 203 includes a main memory 222, a level 2 cache (L2 cache), and a processor core 226. The processor core 226 includes a decoder 228 having translation memory (TM) 230 in accordance with the present invention, a level 1 instruction cache (L1 Icache) 232, and an execution pipeline 234. Compressed code is stored in the static storage device 214, main memory 222, and the L2 cache 224. Decompressed code is stored in the L1 cache and executed by the execution pipeline 234. Various embodiments of the translation tool 216 for generating the compressed code 218 and for decoding compressed instructions in decoder 228 are described in more detail below.

{0035} The processor complex 203 may be suitably employed in hardware components 125A-125D of FIG. 1 for executing program code that is stored in uncompressed form in the L1 Icache 232 and stored in compressed form in the L2 cache 224 and main memory 222. Peripheral devices which may connect to the processor complex are not shown for clarity of discussion. The processor core 226 may be a general purpose processor, a digital signal processor (DSP), an application specific processor (ASP) or the like. The various components of the processing complex 203 may be implemented using application specific integrated circuit (ASIC) technology, field programmable gate array (FPGA) technology, or other programmable logic, discrete gate or transistor logic, or any other available technology suitable for an intended application. Though a single processor core 226 is shown, the processing of compressed instructions of the present invention is applicable to superscalar designs and other architectures implementing parallel pipelines, such as multi-threaded, multi-core, and very long instruction word (VLIW) designs.

{0036} FIG. 3 illustrates exemplary elements 300 of an instruction partition process that splits an original ISA fixed size instruction A 302 based on a binary mix mask (MM) 304 into an

Ax pattern 306 and an Ay pattern 308 with overlap pad bits 310 and 312 in accordance with the present invention. Pad bits are produced due to requirements imposed by modern memory systems to represent instructions and data at least in byte granularity segments. The use of formats having byte granularity segments is utilized to provide a novel compacted representation allowing storage overlap on bit granularity while satisfying byte granularity requirements of the storage system.

{0037} To compress an original ISA code segment, the code segment is partitioned into groups of instructions, with each group contributing a single shared X pattern and a set of unique Y patterns. The Ax pattern 306 represents a bit pattern that is common to a group of instructions to which instruction A belongs. The Ay pattern 308 represents one of a set of unique bit patterns in the same group of instructions. Note, that a code segment can be partitioned into any number of groups between one and N, where N is the number of original instructions in the code segment. The process to determine an optimal or near optimal number of groups and an optimal or near optimal mix mask is a non trivial process described further below. The X patterns for the code segment are stored in an X dictionary comprised of an X memory and the Ay patterns for the code segment are stored in a Y dictionary comprised of a Y memory. An X index is an address of a location in the X memory and a Y index is an address of a location in the Y memory. A combination of these two indexes, patterns from the X and the Y dictionaries and the binary mix mask deterministically represents the original instruction. Byte addressable X/Y dictionary patterns are stored in X/Y memory in compacted form yet are accessible without variable length decompression. Variable length decompression is based on a process of reverse mapping variable length compressed symbols into a native fixed sized alphabet. Fixed size index compression is used for this representation and discussed later. A compress operation 314

uses at least one mix mask for the code segment to select from an instruction 302 an Ax pattern 306 and an Ay pattern 308. In the following examples, a hexadecimal number or Hex number is represented with a '0x' prefix. For example, the instruction 302 [0x9F6D0121] is combined with the mix mask 304 [0xFF80FF80] to produce the Ax pattern 306 [0x9F00(8,9,A,B)] and the Ay pattern 308 [0xDA8(4,5,6,7)]. A decoder 228 decompress operation 316 uses the at least one mix mask for the code segment, an X index fetched X memory pattern and a Y index fetched Y memory pattern to decompress the compressed instruction. For example, the mix mask 304 [0xFF80FF80] is combined with the Ax pattern 306, [0x9F00(8,9,A,B)] fetched from the X memory, and the Ay pattern 308, [0xDA8(4,5,6,7)] fetched from the Y memory, to produce the original instruction 302 [0x9F6D0121].

{0038} As described above, the X patterns and Y patterns are stored in a byte addressable X memory and a byte addressable Y memory. Index compression of X patterns, Y patterns, or both, makes use of a process that eliminates duplicates, reduces double utilization of physical storage and overlaps pad bits, such as the overlap pad bits 310 and 312 and a byte addressable location. A first X pattern may be overlapped with a second X pattern by sharing ending bits of the first X pattern with beginning bits of the second X pattern. In this comparison, the byte having overlapped pad bits allows a further range of options, such as indicted in FIG. 3. For example, the Ax pattern 306 is comprised of three bytes including a first byte 0x9F, a second byte 0x00, and a third byte comprised of a first nibble having a value that is one of a set {8, 9, A, B}, due to the overlap pad bits 310, and a second nibble which may be any number from and including 0x00 to 0xFF. A second X pattern to be overlapped with the first Ax pattern 306 would have one of the set {8, 9, A, B} in a first nibble of beginning bits and any number from and including 0x00 to 0xFF in a second nibble of beginning bits. Another possible overlap

pattern for a third X pattern to be overlapped with the first Ax pattern 306 has 0x00 in a first byte of beginning bits, one of the set {8, 9, A, B} in a next nibble of beginning bits and any number from and including 0x00 to 0xFF in the next consecutive nibble of beginning bits. For Y patterns, the Ay pattern 308 is comprised of two bytes including a first byte 0xDA and a second byte comprised of a first nibble 0x8 and a second nibble which may be one of the set {4, 5, 6, 7} due to the overlap pad bits 312. With such capabilities, it is possible to store four uncompressed instructions in a 128-bit line and for a 32-bit to 16-bit compression with padding and overlap be able to store more than eight compressed instructions in the same space. For example, if four of the compressed instructions could be stored with a byte overlap, ten 16-bit compressed instructions could be stored in the 128-bit line. Thus, the addressing space is compressed and the index addressing would also be compressed as compared to the approach without padding. A second Y pattern to be overlapped with the first Ay pattern 308 would have an '0x8' in a first nibble of beginning bits and a second nibble that is one of the set {4, 5,6, 7}. With a single mix mask, all X patterns are of the same number of bits and all Y patterns are of the same number of bits. With a large number of X and Y patterns, it is possible to have a variety of mappings, with only few mappings taking the least storage, which are considered near optimal or acceptable mappings. The selection of an optimal or near optimal mapping is an NP complete problem, and could not be practically solved for any significant numbers of X and Y patterns. Nevertheless the current invention uses a heuristic that produces an acceptable mapping in a linear time.

{0039} FIG. 4 is a decompressor system 400 having programs stored in compressed form in the main memory 222 and the L2 cache 224 that are decompressed to be stored in the L1 Icache 232 in accordance with the present invention. The L2 cache 224 includes XY index memory 402 that stores an X index and a Y index pair in addressable locations, such as XY entry

404 having a 7-bit X index value of 0x54 and a 9-bit Y index value of 0x734. A multiplexer 405 is used to select an XY entry on an L2 cache hit 406 or an XY value 407 from main memory 222 on a miss in L2 cache. The decompression operation is accomplished in the decoder 228 having index X register 408, index Y register 410, X memory 412, Y memory 414, single MM register 416, and a combiner 418. The L1 Icache 232 includes a plurality of cache lines, such as cache line 420 holding uncompressed instructions.

{0040} At program loading or in an embedded system boot process, main memory 222 is loaded with compressed code, X memory 412 and Y memory 414 are loaded with an associated X and Y dictionary context and the single binary mix mask is set in MM register 416. Note, that the X and Y memory context as well as mix mask can be reloaded during execution if needed. This reload can constitute further granularity of the original code segment in yet smaller sections each with the its custom encoding. For instance, some complex embedded systems, such as smart phones, can invoke multiple independent children applications from a main application, which do not share code space and are self contained. Each such application can have its own custom encoding comprised of an X/Y dictionary and a MM, which is loaded at child process startup.

{0041} A good example of such scenario would be a smart phone operating system (OS) starting an e-mail handling application which for duration of its execution takes over a majority of system resources, and executes code only belonging to the e-mail handling application for a considerable stretch of time. A custom encoding for the e-mail application is loaded at startup, and replaced with a different OS custom encoding only once the e-mail application has finished operation. Furthermore, a possible scenario is when a single dictionary is used for the OS and e-mail handling application, but different sections are utilized via adjusting index X (408) and/or

index Y (410) register contents to offset into the appropriate section of the X/Y dictionary. Loading of the index X (408) and/or index Y (410) register contents may be achieved via a system level request by an OS interrupt, for example, or via a special instruction encoded along with the application code, which is understood by a processor as a request for reloading of the index X (408) and/or index Y (410) register contents. In this case, the number of decoding tables 220 is equal to the number of program segments with different encoding possible for simultaneous residence in L2. Using the above smart phone example, if the OS is dedicated to its own decoder 228 having a translation memory (TM) 230 that always holds OS specific encodings, the system can incorporate another, application specific TM 230 which holds a different encoding, customized for the utilized system applications. Determining which translation memory 230 is appropriate may be made via TLB entries on per code page basis. In contrast, if only one TM 230 is designed in the system, the following procedure may be appropriate depending on the capacity of the installed TM 230. If multiple independent encodings, which exceed the capacity of the OS TM 230, are still desired, as in the above example, once the OS starts the e-mail handling application, a new X/Y dictionary is placed into X/Y Memory (412,414). The MM register 416 may be updated and a L2/TLB flush request is issued. The L2/TLB flush request invalidates the compressed code corresponding to the OS in L2/TLB. For the remainder of the description, a decompressor system 400 is described using a decoder 228 having a single TM 230 and a single encoding to be used for the whole system including any application code.

{0042} Next, the execution pipeline 234 begins fetching instructions from the L1 Icache 232. Initially, each access to the L1 Icache generates a miss indication 422 causing an access to the L2 cache 224. Initially, the access to the L2 cache 224 also generates a miss causing an

access to main memory 222 which responds with a compressed instruction that is loaded in the L2 cache 224 and forwarded to the decoder 228 through multiplexer 405. The decoder 228 decompresses the XY index compressed instruction to an uncompressed format for storage in the L1 Icache 232 and for execution in the execution pipeline 234 as described with regard to the decompress operation 316 of FIG. 3. After a short period of operation, the L1 Icache 232 and L2 cache 224 will have reached steady state.

{0043} From a processor perspective, the execution pipeline 234 attempts a fetch operation with a fetch address and control signals 421 of an instruction to be searched for in the L1 Icache 232. Initially, the L1 Icache 232 determines the instruction is not present and issues a miss indication 422 to the L2 cache 224. The L2 cache fetch operation, for example, is for XY entry 404 which is a hit in the L2 cache 224 causing the XY entry 404 to be passed through multiplexer 405 to the decoder 228. The XY entry 404 is split with the X index value 0x54 received in the index X register 408 and the Y index value 0x734 received in the index Y register 410. The X pattern 306 fetched from the X memory 412 at address 0x54 is applied to the combiner 418. The Y pattern 308 fetched from the Y memory 414 at address 0x734 is also applied to the combiner 418. The single mix mask (MM) 304 [0xFF80FF80] stored in MM register 416 is further applied to the combiner 418. The combiner 418 combines the appropriate bits from the X pattern 306 with the appropriate bits from the Y pattern 308 according to the MM 304 to produce the original instruction 302 that is stored in cache line 420 and passed to the execution pipeline 234.

{0044} Another additional feature of such a system is program content stored in an implied encryption format. Even though no specific encryption type of data scrambling is performed on the instruction stream, program code is stored in the static storage device 214,

main memory 222, and the L2 cache 224 in an application specific and compressed form. Since part of the encoded state of the program code resides inside the processor core 226 in the TM 230, which is not easily accessible in a final product, the static storage 214 and upper memory hierarchy 222 and 224 content is insufficient for restoring the original program, making it difficult to analyze or copy.

{0045} The processor core 226 deals with multiple address spaces. The execution pipeline 234 is operating in virtual address space, which is different from physical address space used throughout a memory hierarchy. The job of translation of one address space into another is generally performed by a translation look aside buffer (TLB) and is of reduced complexity if the physical address space contains instructions of the same size. Therefore, in accordance with the present invention, an execution code segment is represented in fixed size pairs of X and Y indexes that in sum occupy a fixed byte aligned space. This approach, as described in more detail below, allows mixing compressed and uncompressed instructions based on a physical addressing space paging process, where a code page represent an atomic unit of code handled by a single entry of a TLB.

{0046} FIG. 5A illustrates utilization of pad bits in placement of X and Y indexes into X and Y memory. Exemplary elements 500 of another instruction B partition process that splits the instruction B 502 based on a second mix mask 504 into a second X pattern and a second Y pattern with byte overlap pad bits in accordance with the present invention. To compress a second code segment, the second code segment is partitioned into groups of instructions, with each group contributing a Bx pattern and a set of unique By patterns. The Bx pattern 506 represents a bit pattern that is common to a group of instructions from the second code segment. The By pattern 508 represents one of a set of unique bit patterns in the same group of

instructions. The Bx patterns for the code segment are stored in an X dictionary comprised of an X memory and the By patterns for the code segment are stored in a Y dictionary comprised of a Y memory. An X index is an address of a location in the X memory and a Y index is an address of a location in the Y memory. In a compression process, the instruction 502 [0xBAFF0FEB] is combined with the mix mask 504 [0xFFC0FF80] to produce the Bx pattern 506 [0xBAC3(E,F)] and the By pattern 508 [0xFF5(8,9,A,B,C,D,E,F)], for example. In a decompression process, the mix mask 504 [0xFFC0FF80] is combined with the Bx pattern 506, [0xBAC3(E,F)] fetched from the X memory, and the By pattern 508, [0xFF5(8,9,A,B,C,D,E,F)] fetched from the Y memory, to produce the original instruction 502 [0xBAFF0FEB], for example.

{0047} FIG. 5B illustrates an exemplary storage arrangement 520 for the Ax, Ay patterns of FIG. 3, the Bx, By patterns of FIG. 5A and yet another instruction C represented here by Cx and Cy in an X memory 522 and a Y memory 528. The three instructions A, B and C have different masks Amm ≠ Bmm ≠ Cmm in accordance with the present invention. For example, the X memory 522 and the Y memory 528 are byte addressable and utilize 64-bits long access lines. The Ax pattern 306 [0x9F00(8,9,A,B)] is encoded in binary as Ax 523 [1001 1111 0000 0000 10], the By pattern 506 [0xBAC3(E,F)] is encoded in binary as Bx 524 [1011 1010 1100 0011 111] and an exemplary Cx 525 is encoded in binary as Cx 525 [1110 0000 0000 0000 1]. In the exemplary storage arrangement 520 the mix masks Amm 304 [0xFF80FF80], Bmm 504 [0xFFC0FF80], and an exemplary Cmm [FF80FF00] are not equal. Thus, the X patterns associated with each mix mask are a different number of bits and the Y patterns also associated with each mix mask are a different number of bits.

{0048} Index compression of X patterns, Y patterns, or both, makes use of overlap pad bits, such as the overlap pad bits 310, 312, 510, and 512 and a byte addressable location. A first

X pattern may be overlapped with a second X pattern by comparing ending bits of the first X pattern with beginning bits of the second X pattern. In this comparison, the byte having overlapped pad bits allows a further range of options, such as indicted in FIG. 5B. For example, the Ax pattern 523 is comprised of three bytes including a first byte 0x9F, a second byte 0x00, and a third byte comprised of a first nibble having a value that is one of a set {8, 9, A, B}, due to the overlap pad bits 310, and a second nibble which may be any number from and including 0x00 to 0xFF. The Bx pattern 524 is overlapped with the first Ax pattern 523 and has a value of 0xB, which is one of the set {8, 9, A, B}, in a first nibble of the Bx pattern 524. The second nibble of the Bx pattern 524 has the value 0xA which is a number from 0x00 to 0xFF. The beginning bits of the Cx pattern 525 is overlapped with the ending bits of the Bx 524 pattern. For Y patterns, no overlap is shown in the Y memory 528. The bits between patterns, such as bits 534 and 535 may take on any binary value since they are not used in the compression or decompression process. If, due to code changes or additions, a new Dy pattern is generated having a first byte of 0x58, then the new Dy pattern may be overlapped with the By pattern 532 as shown in FIG. 5B. Also, other patterns having a first byte of 0x5 and a second byte as one of the set {8,9,A,B,C,D,E,F} may be overlapped with the ending bits of the By pattern 532.

{0049} FIG. 5C illustrates a decompressor system 540 having programs stored in compressed form in main memory 542 and a level 2 cache 544 that are decompressed in a decoder 546 using multiple mix masks and index compression to be stored in a level 1 instruction cache (L1 Icache) 548 in uncompressed form in accordance with the present invention. The L2 cache 544 includes XYM index memory 552 that stores an X index, a Y index, and a mix mask index in addressable locations, such as XYM entry 554 having a 9-bit X index value of 0x102, an 8-bit Y index value of 0xA9, and a 2-bit M index value of 0x2. A

multiplexer 555 is used to select an XYM entry on an L2 cache hit 556 or an XYM value 557 from main memory 542. The decompression operation is accomplished in the decoder 546 having index X register 558, index Y register 560, a MM index register 561, an X memory 562, Y memory 564, multiple MM memory 566, a double line X register 568, a double line Y register 570, and a combiner 572. The L1 Icache 548 includes a plurality of cache lines, such as cache line 580. After a period of initial operations, the L2 cache 544 and main memory 542 are loaded with compressed code, and the L1 Icache 548 is loaded with decompressed code.

{0050} When the execution pipeline 550 attempts a fetch 581 of an instruction from the L1 Icache 548 and determines that the instruction is not present, a miss indication 582 is issued and the fetch is redirected to the L2 cache 544. For example, the fetch is for XYM entry 554 which is a hit in the L2 cache 544 causing the XY entry 554 to be passed through multiplexer 555 to the decoder 546. The XYM entry 554 is split with the X index value 0x102 received in the index X register 558, the Y index value 0xA9 received in the index Y register 560. And the M index value 0x2 received in the index MM register 561. A line containing the X pattern 506 [0xBAC3E] is fetched from the X memory 562 at address 0x0x100 and loaded into the double line X register 568. A line containing the Y pattern 508 [0xFF58] is fetched from the Y memory 564 at address 0xA8 and loaded into the double line Y register 570. The mix mask 504 [0xFFC0FF80] is fetched from the multiple MM memory 566 at address 0x2 and loaded into the MM register 571. The X pattern 506 [0xBAC3E] is selected from the double line X register 568 based on the X index value of 0x102 and applied to the combiner 572. The Y pattern 508 [0xFF58] is selected from the double line Y register 570 based on the Y index value 0xA9 and applied to the combiner 572. The single mix mask (MM) 504 [0xFFC0FF80] stored in MM register 571 is further applied to the combiner 572. The combiner 572 combines the appropriate

bits from the X pattern 506 with the appropriate bits from the Y pattern 508 according to the MM 504 to produce the original instruction 502 that is stored in cache line 580 and passed to the execution pipeline 550. The other compressed instructions stored along with the XYM entry 554 in the double line X register and double line Y register, especially if using the same mix mask 504, may be decompressed in parallel with the decompression of the XYM entry 554 or in serial fashion depending on an instruction sequence. The format length of a decompressed variable length instruction is determined by the mix mask associated with the compressed instruction.

{0051} A characteristic of the placement of individual patterns within the X and Y memories shown in FIG. 5B and FIG. 5C is that the placement is unrestricted, and governed by a leftmost byte alignment. An acceptable, optimal, or near optimal placement with a high utilization of pad bits can potentially minimize or eliminate byte alignment fragmentation. Any restriction on symbol placement, such as a left adjusted alignment requirement, potentially produces fragmentation. For example, in byte aligned storage of randomly sized objects, a worst case scenario is 7-bits per stored symbol being wasted and carry no useful purpose other than the specific symbol placement guarantee. With the help of index compression in general, and usage of pad bits in particular, these 7-bits could be used for storing useful data. The upper bound of storage savings from wasted bit utilization is 7-bit times the number of symbols in the storage. Total savings from index compression in general is higher, and naturally bounded only by information entropy of the contents. There also could be symbols of different size occupying the same physical bits, and correctly extracted only in presence of appropriate MM which implicitly determines the length of the symbol. If a single MM is used, the X/Y dictionary symbol entries have the same size. If multiple mix masks are utilized, there could be as many variations in symbol sizes as there are MMs. For example, a variable length instruction format may include

32-bit formatted instructions and 64-bit formatted instructions. Nevertheless, the task of optimal placement of symbols is an NP complete problem and could not be expected to be perfectly solved on practice in reasonable linear time. To overcome this limitation, heuristic algorithms are used to produce near optimal placement in linear time, as described in more detail below with regard to FIGs. 12-14.

{0052} Compressed instructions and uncompressed instructions may be mixed in the same executable code segment with no need for a mode switch in operation. In a paged virtual memory system, a page of memory may contain either compressed instructions or uncompressed instructions. FIG. 6 illustrates a paged virtual cache organization 600 having compressed pages and uncompressed pages in accordance with the present invention. A virtual address is generally encoded in two parts. An upper field of address bits usually represent a virtual page number that is encoded based on a selected page size, such as 4K byte pages. A lower field of address bits is a page offset that identifies an address within the addressed page. In a virtual to physical address translation, the virtual page number is translated to a physical page number. The page offset is the same for both the virtual address and the physical address and is not translated.

{0053} A virtual to physical address translation system may include one or more translation look aside buffers (TLBs), such as instruction and data TLBs, to improve performance of a translation process. An instruction TLB (ITLB) is a small cache that stores recent virtual to physical address translations along with attributes of the stored pages, such as entry validation and access permissions. The ITLB generally includes a content addressable memory (CAM) circuit coupled with a random access memory (RAM) circuit and is relatively small, such as having 32 or 64 entries. Each ITLB entry includes a tag in the CAM circuit having the recently used virtual page numbers associated with translated physical page number

in the RAM circuit. For example, the paged virtual cache organization 600 uses an ITLB 602 and a physical memory 604 having uncompressed pages 606 intermixed with compressed pages 608. Each entry of the ITLB 602 has a virtual address tag 610, entry flags 612, such as a valid (V) flag, a read (R) flag, a write (W) flag, a physical page (P-page) address 614, and a compressed (C) field 616. The C field 616 may be a single bit appropriate for identifying a page as compressed or uncompressed for a system having a single mix mask for all compressed pages, for example. Alternatively, the C field 616 may be two or more bits, which for a two bit field may indicate "00" not compressed, "01" compressed with a first mix mask, "10" compressed with a second mix mask, and "11" compressed with a third mix mask. The decision to compress or not to compress an instruction or a block of code is done statically at the compilation and code compression time and might depend on a variety of factors. For instance, if actual implementation of the system is sensitive in any way to the latency of decoding, then performance critical portions of an application might be kept in original uncompressed form, while less frequently executed code may be compressed. Determination of frequently versus infrequently executed code portions is done by the compiler and linker 206 with either optional profile directed feedback information 208 or with compile time heuristics based on the control structure of the code. However, placement of decoder 228 between L2 cache 224 and L1 Cache 232 effectively removes the decoder from performance critical paths of the system. Additional benefits from such placement of the decoder include not requiring changes to the execution pipeline 234 and potential power savings due to increased L2 capacity and thereby minimizing accesses to the main memory. These compression decisions are based on a close interaction of the translation tool 216 with the compiler and linker 206, and can take advantage of profile

feedback information 208, which might identify legacy code to not to be compressed and new function code to be compressed, for example.

{0054} FIG. 7 illustrates a system 700 that mixes compressed and uncompressed code on a per physical memory page basis in a system similar to the paged virtual cache organization 600 of FIG 6. The composition of a compressed page 701 and an uncompressed page 702 is shown for an example main memory physical address space having 4Kilo-byte (4KB) pages with 4 byte (4B) entries. Each page of compressed instructions and each page of uncompressed instructions are of the same capacity having the same fixed number of bytes. However, the number of instructions stored in the compressed page 701 is double the number of instructions stored in the uncompressed page 702, using, for example, a compressed instruction format that is half the number of bits used in the uncompressed instruction format. For example, the compressed page 701 stores compressed instruction d,e,f,g etc. as a pair of X/Y indexes in 16 bits or 2 Bytes. Thus, the compressed page 701 which is a 4K page contains 2048 instructions, while the uncompressed page 702 contains only 1024 instructions. Since the page boundaries are not affected by whether a page holds compressed instructions or holds uncompressed instructions, the address translation for physical location of a page is unchanged, but compressed page holds more individual instructions.

{0055} FIG. 8A illustrates a paged decompressor system 800 for decompressing compressed instruction pages and accessing uncompressed instruction pages in accordance with the present invention. The paged decompressor system 800 is comprised of a processor pipeline 802, an ITLB 804, a physical address buffer 806, an L1 Icache 808, an L2 cache circuit 810, and a compressed instruction decoder circuit 812. A translation process begins by applying a virtual page number 805 selected from a virtual address 803 to the CAM circuit which does a parallel

comparison of the applied virtual page number generally with all of the stored recently used virtual page numbers stored with the entry tags in the CAM tags 819. If there is a match, the CAM circuit accesses a corresponding entry 820 in the RAM circuit which is output as a translated physical page address 815 stored in the physical address buffer 806. A translated physical address 809 comprises the translated physical page address 815 concatenated with the page offset 817 from the virtual address 803.

{0056} For example, in an embedded system with a virtual address space of 4 gigabytes (4GB) and 4K byte pages, a virtual address 803 is comprised of a virtual page number 805 having bits [31:12] and a page offset 807 having bits [11:0]. In the same embedded system, the memory hierarchy of caches and main memory may encompass a physical memory space of 512K bytes and 4K byte pages. On a hit in the ITLB 804 the virtual address 803 is translated to a physical address 809. The physical address 809 is comprised of a physical page number 815 having bits [28:12] and a page offset 817 having bits [11:0]. In such a system, the virtual to physical translation system would translate a virtual page number 805 encoded in bits [31:12] to a physical page number 815 encoded in bits [28:12]. Also, on the hit the compressed bit field 821 is also output to be stored in the physical address buffer 806 as C bit field 822. The placement of the compressed bit field 821 and the C bit field 822 is exemplary.

{0057} The physical address 809 is used to search the L1 Icache 808 for a matching entry. If a matching entry is found, it is a decompressed instruction that is associated with the matching entry and is selected to pass through the L1/L2 multiplexor 824 to the processor pipeline 802. On a miss in the L1 Icache 808, the physical address is directed to the L2 cache 810 to search for a matching entry. On a hit in the L2 cache 810, with the C bit field 822 indicating compressed instructions from a compressed page, a line of compressed instructions

having the associated matching entry is fetched and stored in a L2 read buffer 826. On a hit in the L2 cache 810, with the C bit field 822 indication uncompressed instructions from an uncompressed page, a line of uncompressed instructions having the associated matching entry is fetched and stored in a L2 read buffer 826. Uncompressed instructions bypass the compressed instruction decoder 812 and are made available at the L2 read multiplexer 828 for storage in the L1 Icache 808 and to be selected to pass through the L1/L2 multiplexor 824 to the processor pipeline 802.

{0058} On the hit in the L2 cache 810, the fetched compressed instructions in the L2 read buffer 826 are decompressed in the compressed instruction decoder 812 based on the C bit field 822 indicating compressed instructions to the control circuit 830. The decompression process is described in more detail below with regard to FIG. 8B. The decompressed instructions are stored in decompression buffer 832 which may be selected by the L2 read multiplexor 828 for storage in the L1 Icache 808 and selected to pass through the L1/L2 multiplexor 824 to the processor pipeline 802.

{0059} As was illustrated earlier in FIG. 5C, compression of a native application may specify usage of multiple mix masks. If desired granularity is down to a single instruction, as it is suggested in the FIG 5C, each X/Y index pair must have a way to select appropriate mask. If a mix mask is selectable on an instruction by instruction basis, identification of the mix mask requires additional storage bits, such as the 2-bit mix mask index shown for example in XYN entry 554. Another approach allows a mix mask to be selected per code page, which removes the need for a mix mask marker from an X/Y index pair 554 and places the mix mask marker in a TLB page descriptor as part of the C field, for example.

{0060} FIG. 8B illustrates an exemplary decompression state diagram 850 that illustrates the state of L2 cache compressed page instructions and L1 Icache decompressed instructions for execution on processor pipeline. FIG. 8B shows an L2 cache 852, a L2 read buffer 854, a compressed instruction decoder (CID) 856, a decompression buffer 858, an L1 Icache 860, a physical address buffer 862, and a processor pipeline 864. The L2 cache 852 has an uncompressed line 866 and a compressed line 868 as an initial state. The uncompressed instructions are 32-bit native instructions and each of the compressed instructions are 16-bits made up of an X-index and a Y-index pair, as described above, for example, with respect to FIG. 4. The compressed line 868 holding twice as many instructions as the uncompressed line 866. A fetch for an instruction M is made by the processor pipeline 864 which is translated from a virtual address to a physical address by a ITLB, such as the ITLB 804 of FIG. 8A, and stored in the physical address buffer 862. Initially, the L1 Icache 860 does not contain the instruction M and a miss is generated which causes a fetch to upper levels of the memory hierarchy. The decompression buffer 858 is checked first which in this exemplary scenario, the instruction M is not found. The L2 cache 852 is checked next and the compressed line 868 is found to contain the instruction M, in a compressed form "m" 870. The compressed line 868 is fetched and stored in the L2 read buffer 854. The CID 856 receiving the C bit field 872 and the fetch address allowing the compressed instructions from the L2 read buffer 854 to be fetched sequentially beginning from the compressed instruction "m" 870. Note, a first instruction in a fetch sequence may be accessed, decompressed first and quickly forwarded to the processor execution pipeline to minimize timing requirements. Since the compressed instructions as described herein have fixed length and known storage address locations prior to decompression, a first compressed instruction in a fetched sequence may be identified in a line of compressed instructions. Since

the L1 Icache has a line length of eight uncompressed instructions, the compressed instructions are accessed from the L2 read buffer 854 in a L1 Icache line length, starting from "m, n, o, p, then i, j, k, and l". The second half of the L2 read buffer 854 is accessed next starting from "q, r, s, t, u, v, w, and x". Thus, the compressed instructions are accessed from an arbitrary starting point in the L2 read buffer 854.

{0061} Since the instruction M is fetched first, it is decompressed first in the CID 856 and then may be forwarded to the processor pipeline 864. As the other instructions "n, o, p, then i, j,, k, and l" are decompressed they are combined with the decompressed instruction M and loaded in the decompression buffer 858. Once a full line of instructions have been decompressed they are loaded into the L1 Icache 860. Alternatively, individual decompressed instructions or pairs of decompressed instructions, for example, may be individually updated in the L1 Icache 860 and forwarded as needed to the processor pipeline 864. It is noted that if the decompressed instruction line stored in the decompression buffer 858 is duplicated in the L1 Icache 860, the decompression buffer, holding the decompressed instruction line, may be used as a victim cache temporarily storing the last instruction block evicted from L1.

{0062} The determination of the X-index, Y-index, X pattern, Y pattern, and one or more mix masks depends on the native instruction architecture and the use of instructions in a particular code segment. When analyzing a code segment, such as code segments in a smart phone, a single mix mask may provide comparable compression to that obtained through use of multiple mix masks. The use of a single mix mask could be seen as using a single instruction group and effectively separating the native instructions used in a code segment into two parts, the X patterns and the Y patterns. The original distinction of fixed (X) and mutable (Y) parts in a group becomes less meaningful and can be viewed interchangeably. Also, use of a single mix

mask simplifies the design of a combiner circuit, such as the combiner 418 of FIG. 4. For example, in an experimental code segment having 16,542 native 32-bit instructions, there were only 6,464 unique instructions and overall entropy of 0.76. Using a single mix mask, these 6,464 instructions were split into 1,351 X patterns and 1,345 Y patterns. Using the techniques of the present invention, each of the 16,542 native 32-bit instructions is replaced by a 23-bit X/Y-index pair, providing a 28.1% compression in the storage capacity of the L2 cache and main memory in a system such as shown in FIG. 2. Resulting entropy of the encoded data increased from 0.76 to 0.89. However, the 0.89 entropy measurement out of a theoretically possible 1.0 informational entropy measure, illustrates that a different mix mask may provide considerably better results. The determination of an optimal or near optimal single mix mask to provide the best possible compression over a plurality of code segments is a complex problem.

{0063} FIGs. 9A-9C illustrates an automated mix mask determination process 900 based on a variation of a genetic algorithm in accordance with the present invention. A translation tool, such as the translation tool 216 of FIG. 2A implements the process 900. At block 902, the process 900 is started. At block 904, the code segment P to be compacted is obtained. For example, the code segment P may include boot code, operating system code, and multiple function programs as utilized in a particular product. At block 906, the unique instructions in the code segment P are collected in a P hash table (P_unique). At block 908, the P_unique instructions are sorted in a semantic order, such as an ascending bit pattern order, based on the instruction formats. For example, each instruction in the P hash table is interpreted as an integer number using 32-bits and is sorted in ascending order in a list. This guarantees the smallest Hamming distance between neighboring instructions and enables instruction grouping in later steps. At block 910, a bit switch frequency is collected in sorted order of unique instructions.

For example, if a bit in position p changes from "1" to "0" or "0" to "1", a bit_toggle[p] value is incremented by one. A bit_toggle array size is equal to the width of the largest instruction in the original ISA expressed in bits, such as 32 for 32-bit instructions. At block 912, a set of seed mix masks, the seed MM population, is selected using several known good MMs, several randomly generated MMs, and several MMs generated from the collected bit toggle counts. The known good masks are pre-populated from previously obtained MMs from similar applications and are specific to any given architecture. The collected bit toggle count serves as a good heuristic in estimating likelihood of a bit changing its value in the given position in the actual code. Empirically, selecting a mix mask as a function of threshold for a given position produces good seed mask. At block 912, the seed MM population is an array of individual MMs of size between 10 and 30 individuals and may be selected as an even number of individual MMs. The larger population can produce better results, but a linear increase in the number of MMs chosen requires a corresponding increase in computation time. A particular seed MM population could be selected to fit a specific computational platform. A seed population value of 20 MMs has been shown to be a good tradeoff in an experimental setup. At block 914, a fitness value for each selected MM in the seed MM population is set, for example, to a low but non-zero value. The fitness value is, for example, a double precision floating point number representing the level of benefit a MM provides. The process 900 proceeds to connector A 916 FIG. 9A which is a link to connector A 916 in FIG. 9B.

{0064} In FIG. 9B the repetitive part of the process 900 is contained. A seed population of 20 MMs is provided as input to the process 900. At the decision block 920, a determination is made whether another iteration will take place. A threshold value is compared to a numerical

exit condition and if the exit condition is not reached, as would be the case when the threshold is less than or equal to the exit condition, another iteration is commenced.

{0065} At block 918, an individual having the best fitness value is determined to be the best individual, and its fitness value (cost function) is recorded. Also, the value of total_fitness determined as a sum of cost functions of the individuals in the population is recorded, obtained from process 1000 FIG. 10, as described in more detail below. The total_fitness value is later used as a parameter to a mutation algorithm causing a higher rate of mutation if overall compression effectiveness is low based on an assumption that a faster rate of change is desirable. At decision block 924, a determination is made about progress of completing the process 900. If the last iteration best fitness value was unchanged from the previous iteration, the process 900 proceeds to block 926. At block 926, the threshold value is increased. If the last iteration best fitness value is different from the previous iteration's best fitness value, the process 900 proceeds to block 928. At block 928, the population is sorted by the fitness values. At decision block 930, a determination is made whether the process 900 beginning with the decision block 920 is on a first iteration. If it is determined to be the first iteration, the process 900 proceeds to block 932. Otherwise the process 900 with a current population of MMs proceeds to blocks 934-952 for creating a new population.

{0066} At decision block 934, a determination is made whether an elitism indicator is set. The elitism indicator indicates whether a plurality of best in class MM are to be preserved. If the elitism indicator is set, the process 900 proceeds to block 936. At block 936, two best performer MMs from the previous iteration are directly copied into the new population. At block 938, two offspring MMs are produced from the two best performers as described in more detail below with regard to FIG. 9C. At block 940, the two offspring MMs are randomly mutated to

create two mutated offspring MMs as described in more detail below with regard to FIG. 9D. At block 942, the first four members of the current population are replaced by the two best performer MMs and the two mutated offspring MMs. The process 900 then proceeds to block 946.

{0067} Returning to decision block 934, if the elitism indicator was not set indicating elitism is not elected, the process 900 proceeds to block 946. At blocks 946 and 948, a similar procedure of interbreeding as described at block 938 and mutation as described at block 940 is repeated for unprocessed pairs of remaining individual MMs from the previous iteration population. At block 946, a partner selection algorithm is used to select random individuals, with chance of selection proportional to the fitness of the individual. The better performing individuals are more likely, but not guaranteed, to produce offsprings. At the end of block 948, new iteration population is determined. At block 950, any duplicates in the new iteration population are mutated to insure that there are no duplicates in the new iteration population. At block 952, fitness values for individual MMs in the new iteration population are reset. At this point, the new iteration population is ready for a cost calculation. In order to save computation time, a list of previously computed costs for previously processed MM (individuals) is maintained, and an already determined cost is simply pre-populated, not computed in block 932. The process 900 then proceeds to connector B 922 on FIG. 9C.

{0068} Returning to the breeding operation at block 938 and in reference to FIG. 9B, the two best performer MMs are chosen as parent A 960 and parent B 962, FIG 9D, where the mix mask (MM) is represented by a 32-bit number and for the purpose of this step the MM is denoted a gene. For the breeding process 959 of FIG. 9D implemented at block 946, two individual MMs are selected from the input MM population and may also be represented as parent A 960 and

parent B 962. At blocks 938 and 946, the two parents are breed as shown in FIG. 9D via an exchanging genes process 964 at a random point in the gene at a random frequency producing two offsprings A 966 and B 968. A random selection may be guided by a pseudo-random number generator and may be termed a semi-random selection. One parent, parent A 960, provides a beginning portion 970 of one offspring, offspring A 966, new gene and the other parent, parent B 962, provides a tail portion 971 of that offspring gene. The other offspring B 968 is a complement of the first one. The parent B 962 which produces the tail portion 971 for the offspring A 966 produces a beginning portion for offspring B 968. The parent A 960 which produces the beginning portion 970 for the offspring A 966 produces a tail portion for offspring B 968. The cross point 972 is selected randomly in the range of 0-32. An underlying reason for the quick algorithm convergence could be seen as due to a preservation of successful adaptation traits from one of the parents into one of the offsprings, which is later confirmed by the fitness rate calculation. The breeding process 959 illustrated in FIG. 9D is itself a pseudo-random event and subject to a threshold value identified as a CROSSOVER_RATE threshold. For example, out of 100 attempts to exchange genes between two parents, only CROSSOVER_RATE times the number of exchange attempts will even take place.

{0069} Returning to blocks 940 and 948 of FIG. 9B, the newly generated offspring MM population is mutated by semi-randomly flipping individual bits, changing a "0" to a "1" or a "1" to a "0", in each offspring gene. This mutation process 974 is illustrated in FIG. 9E where an individual MM A 975 is mutated by bit flips associated with bits, 976, 977, and 978. The mutated individual MM A 979 is shows with the results of the bit flips. In the mutation process 974, a pseudo-random mutation acts to introduce previously unavailable traits into the MM population. For example new previously untried MMs are created, but at a controllable rate, so

that undesired mutations may be pruned out quickly. At block 942, four members of the MM population are set to the preserved two best performers from block 936 and the two new mutated offspring from block 940. At block 946, the remaining MM population is breed in pairs, as described with regard to process 955 of FIG. 9C, to obtain pairs of new offsprings. At block 948, the mutation process 974 of FIG. 9D is followed to produce two new offspring. At block 950, any duplicate MMs are mutated. At block 952, the fitness value for each MM individuals is reset. At block 932, previously calculated costs for repeating MMs are reused. The process 900 then proceeds to connector B 922 which is a link to connector B 922 in FIG. 9C.

{0070} In FIG. 9C at block 955, a current individual MM's cost function is calculated, as a dynamic compression function of the selected MM and the sorted P_unique list. The cost function is comprised of two main components: the size of external storage and size of translation dictionary. These two components combined reflect an obtained degree of compression or an achieved level of entropy. The exact relationship of external storage size to internal X/Y memory decompression table size is determined by a parameter indicating a relative importance of the two components. For example, the cost function equals a product of sum x and y index size times X/Y dictionary size in Kbytes: So cost is equal to the X/Y_dictionary_size * (x_index_size + y_index_size). This formula reflects a balanced approach to evaluating the relative importance of internal dictionary size versus external storage size. A preference may be introduced in the cost function by introduction of relative importance weight constants. For example, cost is equal to (weight_constant * X/Y_dictionary_size) * (x_index_size + y_index_size). In a current implementation, the weight_constant is set to 1.00, though may vary from 0.01 to 1.00.

{0071} At decision block 956, a determination is made whether the run time for process 900 has exceeded an allotted time. If the run time has exceeded the allotted time, the process 900 proceeds to block 958 and stops. If the run time has not exceeded the allotted time, the process 900 proceeds to decision block 957. At decision block 957, a determination is made whether a lowest current cost MM has been found. For example, finding only a single MM that generates a current cost less than the threshold may be insufficient reasons to terminate the process 900. On the other hand, it is possible to detect that a theoretical lowest entropy bound representation has already been reached and any further search is unnecessary. In another example, a number of MMs may be found, each generating approximately the same current cost that is less than the threshold which may be sufficient reason to terminate the process 900 at block 958 with selecting the MM with the lowest current cost or selecting one of a plurality of MMs each with the same acceptable lowest current cost. Otherwise the process 900 proceeds to connector A 916 of FIG. 9B.

{0072} FIG. 10 illustrates an exemplary dynamic compression and current cost calculation 980 as a process 1000 in accordance with the present invention. At block 1002, current cost values are input, such as an MM available from block 952 of FIG. 9B that is selected after determining it has not been previously evaluated in decision block 932 and the sorted P_unique instructions are input. At block 1004, the sorted P_unique instructions are grouped according to a current grouping heuristic for the selected MM based on having a single MM or multiple MMs for the code segment. For example, grouping for single MM is a process of separating the unique instructions into X and Y portions, such as the exemplary elements 500 of FIG. 5A, and constructing initial uncompressed X and Y tables. In one example, the whole native instruction program could be viewed as a single group. On the other hand, grouping for

multiple MMs involves partitioning the unique instructions into several groups with a unique MM for each group. In this case, an X pattern represents a fixed pattern for a group, where the whole group shares the single fixed X pattern, and the multiple Y patterns per group represent the mutable part of the group. While several heuristics are possible for near optimal partition into groups, selection of one specific heuristic could be driven by another heuristic which may be based on the size of the input list of unique instructions and its initial entropy. The following heuristics have proven through experimentation to be practical, including Hamming, weighted Hamming, number of ones, and weighted number of ones.

{0073} FIG. 11 illustrates a Hamming heuristic 1100 that is used to create a new group when a specified threshold in hamming distance is exceeded between neighboring entries in the sorted unique instruction list. The weighted Hamming heuristic 1100 assigns certain bit fields positions more weight than others in computing a change score that is compared to a threshold, as shown in FIG 11. In the process 1100 at block 1102, the full list of sorted input raw instruction patterns is obtained (sorted P_unique). It then processes two neighboring entries at a time by analyzing hamming distance between them. At block 1104, a variable hamming distance value (hamm_dist) is calculated as an XOR of the two neighboring entries. For example, if list of raw instructions contains only three entries 0, 1 and 2, hamm_dist is calculated twice – first as XOR between 0^{th} and 1^{st} entry, then as XOR between 1^{st} and 2^{nd} entries. At block 1106 the hamm_dist is parsed one bit at a time, and a sum_weight value is incremented for each non-zero bit. Block 1106 could be seen as counting non zero bits in each hamm_dist. The increment amount may be changed by bit position. An exact increment amount is read from a weight_array which in turn calculated from the Bit_Toggle array computed in block 910 of FIG. 9A or precalculated statically. At decision block 1108, a determination is made whether the

sum_weight value exceeds a weight_threshold. A positive outcome indicates a new instruction group is to be formed. At block 1110, the new group is formed and the process 1100 proceeds to decision block 1112. A negative output of decision block 1108 causes the process 1100 to proceed to the decision block 1112. At decision block 1112, a determination is made whether the end of the sorted P_unique list has been reached. If the end of the list has not been reached, the process 1100 proceeds to block 1114 which increments a loop variable "i" and proceeds to block 1104. If the end of the list has been reached, the process 1100 proceeds to block 1116 and stops.

{0074} Other heuristics may be used, such as the number of ones heuristic and the weighted number of ones heuristics both of which are based on measuring mutability of a bit field in an instruction. For example, a lower number of ones in a MM indicates a larger X pattern and fewer Y patterns per group. A higher number of ones in a MM indicates a smaller X pattern and more Y patterns per group. There exists a non linear dependency with a single point for selection of the number of ones that produces the highest compression. In a current approach, a single MM partitioning strategy is selected at block 1004. Continuing at block 1006, X memory and Y memory tables, such as the X patterns stored in the X memory 412 of FIG. 4 and the Y patterns stored in the Y memory 414, respectively, are determined as described above with regard to FIG. 4. At decision block 1008, a filtering heuristic is used to filter out substandard MMs, which represent unfit individuals, which are possible in the process 900. This filtering is done to conserve computation time and does not affect the effectiveness of the process 1000. A mask is determined to be clearly unfit if the unpacked X/Y table generated from it exceeds in byte size the input raw instruction list size. At decision block 1008, a determination is made whether the X memory size + Y memory size is greater than the number of P_unique

instructions times the native instruction width, which in this exemplary case is 32-bits. At decision block 1008, if the determination is positive, the process 1000 proceeds to block 1010 to reject the mix mask. At block 1010, the fitness value for the MMs rejected at 1008 is set to zero and the process 1000 proceeds to connector B 922 on FIG. 9C. If the determination is negative, the process 1000 proceeds to block 1012.

{0075} At block 1012, the X memory and Y memory tables are compacted. Block 1012 directly affects the size of the X and Y tables which in turn determines the size of indexes needed to address these tables and ultimately determines the size of the external compressed code segment. The X and Y tables are compacted using indexing compression as described with regard to FIG. 5B and FIGs. 12-14. At block 1014, a current cost is generated as equal to a constant C1 times a sum of the X memory size plus Y memory size which is then multiplied by the sum of the X index size plus the Y index size. This process 1000 provides an advantageous heuristic for computing an entropy value which is represented by the current cost that is close to the actual entropy of the compressed code which is now separated into external collection of X/Y indexes and the X/Y compacted tables. Optionally at block 1014, a theoretical minimum entropy measure may be determined and whether further search is needed or not. At block 1016, the generated current cost is output to decision block 956 in FIG. 9C.

{0076} As noted above, near optimal selection of entries in the X/Y table combined with index compression can significantly reduce the size of X/Y tables. Two exemplary placement algorithms may be used based on trade off between degree of compression and execution time. A first heuristic is a near-exhaustive search X/Y table compaction algorithm with pad sorting presented at FIG. 12 in accordance with the present invention. Processing begins at block 1202 by obtaining the list of symbols to be compressed. This is the list of the present set of X/Y table

symbols to be compacted. There are "table_size" symbols in the table. At block 1204, this list is sorted in an order which maximizes likelihood of pad overlapping. Also, at block 1204, an iteration index "i" is reset to 0.

{0077} The pad-sort operation at block 1204 is a specialized way to sort symbols in the X/Y tables to increase likelihood of pad overlapping prior to actual compaction process initiation. The pad sort operation is implemented by a quick sort algorithm, with a comparison function demanding a swap of two symbols if there is a match between one or more left most significant bytes of one symbol and one or more right most significant bytes of the other. Here length of symbol is measured in full bytes, and is left adjusted. Right-most significant byte can have just several significant bits, but for the purpose of sorting it is inconsequential.

{0078} At decision block 1206, a determination is made whether the iteration index "i" is less than the "table_size". If the iteration index "i" is greater than or equal to the "table_size", the process 1200 proceeds to block 1232 and stops. Otherwise, the process 1200 proceeds to decision block 1208, since the i^{th} symbol of the X/Y table has not been processed yet. At decision block 1208, a determination is made whether a parameter seen[i] is equal to a "1" indicating the i^{th} symbol has already been processed. If seen[i]=1, the process 1200 proceeds to block 1210 in which the iteration index "i" is incremented by 1. If the symbol has not already been processed, seen[i]≠1, the process 1200 proceeds to block 1212. At block 1212, the i^{th} symbol is inserted into the compressed table. Details of insertion process are described below with regard to FIG. 13. Also at block 1212, the fact of insertion is noted by setting the parameter seen[i] to 1, setting a best_score variable to 0, and initializing a new iteration index "k" to a value of i+1.

{0079} At decision block 1214, a determination is made whether the new iteration index "k" is less than the number of symbols to be compressed. If the new iteration index "k" is less than the "table_size", the process 1200 proceeds to block decision block 1216, since the k-th symbol has not been processed yet. At decision block 1216, a determination is made whether a parameter seen[k] is equal to a "1" indicating the k^{th} symbol has already been processed. If seen[k]=1, the process 1200 proceeds to block 1218 in which the new iteration index "k" is incremented by 1. If the symbol has not already been processed, seen[k]≠1, the process 1200 proceeds to block 1220. At block 1220, a match score is calculated for the k^{th} symbol.

{0080} At block 1220, a score_match calculation mimics a symbol insertion process, but does not insert any symbols. Rather, the score_match calculation returns a score proportional to an amount of savings expected from placing the current symbol into the compressed table in the table's current form. This calculation process locates among remaining unprocessed symbols in the X/Y uncompressed list, the potentially best matching candidates. Score_match returns a matching indication or a very high score, such as infinity for example, if the symbol could be completely matched within existing contents, or provides a number of overlapping bytes if the symbol is matched at the end of compressed memory. If no overlap is found, a zero score is returned.

{0081} The computed score reflects a degree of savings to be expected from inserting the k-th symbol into the compressed table in its current state. At decision block 1224, a determination is made whether the score is equivalent to infinity. As noted above, a matching indication, such as a score of infinity, means that k-th symbol completely matches to the current state of compressed table, inserting it would not increase the size of the compressed table, and the process 1200 directly proceeds to insertion step in block 1228. Note, that the process of

insertion in this case is a recording of current symbol offset from the beginning of compressed X/Y table which later becomes its representation in external memory. Also note, that in general, when multiple Mix Masks are used, multiple symbols that share a common prefix could have the same offset, but different length. Otherwise, at block 1226, comparison is made between the computed score and best score seen so far. If the newly computed score exceeds the best score so far, the process 1200 proceeds to block 1230. At block 1230, the present best_score is set equal to the newly computed score and a best-index is set to "k". At block 1218, the new iteration index "k" is incremented by one, and the process 1200 proceeds to block 1214. Once "k" reaches the end of the list (table_size), as determined at block 1214, the process 1200 proceeds to block 1222. At block 1222, a determination is made whether the best_score was determined. If a best score has not been determined, the iteration index "i" is incremented at block 1210, and the process 1200 continues from block 1206. At block 1222, a non-zero best_score indicates that a best fitting symbol was found, and the process 1200 proceeds to block 1228. At block 1228, the best fitting symbol is inserted into the compressed table and the fact of insertion is noted in by setting the parameter seen[best_index] array equal to "1". Once the iteration index "i" reaches table_size as determined at block 1206, the process is terminated at block 1232. At this point, the evaluated symbols have been inserted into the compressed table.

{0082} FIG. 13 illustrates a symbol insertion process 1300 in accordance with the present invention. At block 1302, the current state of compressed memory and the new symbol to be inserted are obtained. At decision block 1304, a determination is made whether the new symbol is zero. If the new symbol is zero, the symbol is mapped to a specialized position in the compressed memory at block 1320 and only one byte is stored regardless of symbol size. At decision block 1304, if the new symbol is determined to be non-zero, the process 1300 proceeds

to block 1306. At block 1306, a current pad_mask is computed. The pad_mask is a binary variable used to mask out insignificant portion of the last significant byte in symbol. For instance, if the symbol being inserted has size of 15bits, the last bit of the last byte is insignificant and needed to be ignored during comparison for placement. The pad_mask in this case is 0xfe (pad_mask always applied to the last significant byte only). At decision block 1308, a determination is made whether the iteration index "i" is less than the present compressed table size minus the new symbol size plus 1. If the determination is positive, the process 1300 proceeds to block 1310. At block 1310, the new symbol is repeatedly attempted to be placed in compressed memory checking each symbol location for matching only significant bits to the current contents with byte step. Such an operation is achieved by repeatedly comparing the first significant byte of the new symbol with each byte already in the memory, and if match found for the first significant byte, other remaining bits in the new symbol are matched. The match only takes into account significant bits in the symbol, via application of the pad_mask as described above. If such a location is found at decision block 1312, the process 1300 proceeds to block 1318. At block 1318, the symbol is placed at the determined location, and the iteration index "i" becomes the symbol's offset from the beginning of the compressed table. This index will be eventually stored in external memory and utilized to locate original X/Y pattern in X/Y compressed table. If block 1316 is reached, it means the current symbol could not have been completely matched to the existing content of the compressed memory, and it needs to be placed at the end of list. In that case a maximum overlap check is performed on each byte of the symbol. If no overlap was found, the new symbol is simply placed at the end of the compressed table.

{0083} FIG. 14 illustrates the triangular X/Y table compaction with pad-sort process 1400 in accordance with the present invention. The process 1400 is an alternative to the exhaustive search X/Y table compaction process 1200. The heuristic process 1400 is O(n) faster than the process 1200, but may not produce as good of a compression. However, the compression results of the process 1400 may be acceptable for a number of practical applications. Moreover, a combination of the two processes might be viable in actual implementation – the faster process shown in FIG. 14 is used for determining best mask candidates, and the slower one shown in Fig. 12 is used for generating final encoding. The process 1400 begins with steps 1402, 1404, 1406, 1408, and 1412 that are the same as steps 1202, 1204, 1206, 1208, and 1212 of FIG. 12. At block 1412, a similar insertion of i-th element is performed as is done at block 1212, but beginning with block 1414 there is no best_score that is used as is done with blocks 1214 and 1222 of FIG. 12. Instead of using a best score evaluation, at the remaining blocks of FIG. 14, the symbols between i+1 and table_size are traversed once and every element found with a non-zero score as determined at decision block 1424, is inserted into the compressed table at block 1426.

{0084} The methods described in connection with the embodiments disclosed herein may be embodied in a combination of hardware and in a software module storing non-transitory signals executed by a processor. The software module may reside in random access memory (RAM), flash memory, read only memory (ROM), electrically programmable read only memory (EPROM), hard disk, a removable disk, tape, compact disk read only memory (CD-ROM), or any other form of storage medium known in the art or to be advised in the future. A storage medium may be coupled to the processor such that the processor can read information from, and in some cases write information to, the storage medium. The storage medium coupling to the

processor may be a direct coupling integral to a circuit implementation or may utilize one or more interfaces, supporting direct accesses or data streaming using downloading techniques.

{0085} While the invention is disclosed in the context of illustrative embodiments for use in processors it will be recognized that a wide variety of implementations may be employed by persons of ordinary skill in the art consistent with the above discussion and the claims which follow below.

What is claimed is:

1. A method for decompressing compressed variable length instructions, the method comprising:

fetching compact fixed length instructions from a system memory configured to store compressed variable length instructions, wherein each compact fixed length instruction comprises an X-index and a Y-index;

fetching for each compact fixed length instruction an X-bit pattern from an X memory using the X-index and a Y-bit pattern from a Y memory using the Y-index; and

combining the X-bit pattern with the Y-bit pattern based on a mix mask associated with each compact fixed length instruction into a decompressed variable length instruction, wherein a format length of the decompressed variable length instruction is determined by the associated mix mask.

2. The method of claim 1, wherein the format length of the decompressed variable length instruction includes a 32-bit format length and a 64-bit format length.

3. The method of claim 1, wherein the system memory includes a main memory and a level 2 instruction cache.

4. The method of claim 1, further comprises:

fetching each compact instruction from a level 2 instruction cache in response to a miss in a level 1 instruction cache.

5. The method of claim 4, further comprises:

storing each decompressed variable length instruction in a level 1 instruction cache.

6. The method of claim 1, wherein each compact fixed length instruction further comprises a mix mask index.

7. The method of claim 6, wherein the mix mask associated with each compact fixed length instruction is selected from a mix mask memory using the mix mask index.

8. An apparatus for translating compressed instructions stored in a virtual memory system, the apparatus comprising:

a paged instruction cache configured to store pages of compressed instructions intermixed with pages of uncompressed instructions; and

an instruction translation look aside buffer (TLB) configured to store an address translation entry that identifies a page in the paged cache as storing compressed instructions.

9. The apparatus of claim 8, wherein a bit field in the address translation entry indicates the page in the paged cache as storing compressed instructions.

10. The apparatus of claim 9, wherein the bit field further indicates a mix mask appropriate to use in translating the compressed instructions at an identified page.

11. The apparatus of claim 8, further comprising:

a translation decoder configured to decompress compressed instructions, the translation decoder placed between the paged instruction cache and a lower level instruction cache configured to store uncompressed instructions and decompressed instructions.

12, The apparatus of claim 11, wherein the translation decoder comprises:

a compressed instruction decoder configured to decompress compressed instructions; and

a decompression buffer configured to store the decompressed instructions in preparation for storage in the lower level instruction cache.

13. The apparatus of claim 11, further comprising:

a bypass circuit configured to pass decompressed instructions from the translation decoder directly to a processor pipeline.

14. A system for translating compressed instructions to an executable format, the system comprising:

a level 2 cache configured to store an X-index and a Y-index for each compressed instruction of program code;

a translation unit configured to receive compressed instructions from the level 2 cache, for each received compressed instruction to select an X-bit pattern and a Y-bit pattern from a translation memory using the X-index and the Y-index, and to use a program specified mix mask for combining the selected X-bit pattern and Y-bit pattern into a native instruction format; and

a level 1 cache configured to store the native instruction format for each compressed instruction.

15. The system of claim 14, wherein the level 2 cache is partitioned into a first set of pages that store instructions in a native instruction format and a second set of pages that store the compressed instructions.

16. The system of claim 14, further comprising:

an instruction look aside buffer comprising entries that identify a page in the Level 2 cache contains compressed instructions.

17. The system of claim 14, wherein the program code is stored in the L2 cache in an implied encryption format.

18. A method of determining a mix mask for efficiently translating compressed instructions, the method comprising:

breeding pairs of mix masks represented as genes from a seed population of mix masks to produce pairs of offspring mix masks;

mutating the offspring mix masks to produce mutated offspring mix masks that update the seed population; and

determining a mix mask from the updated seed population that provides a high level of compression, wherein patterns of bits are combined according to the determined mix mask to translate compressed instructions of a program to executable form.

19. The method of claim 18, wherein breeding further comprises:

randomly selecting a crossover point in a pair of mix masks identified as parent A and parent B; and

exchanging a rightmost set of bits starting from the crossover point from parent B with parent A to produce a pair of offspring mix masks.

20. The method of claim 18, further comprising:

randomly selecting breeding based on a random frequency of selection.

21. The method of claim 18, wherein mutating further comprises:

randomly flipping an individual bit in one of the pair of offspring mix masks to determine a mutated offspring mix mask; and

replacing a mix mask in the seed population with the mutated offspring mix mask.

22. The method of claim 18, wherein the program includes boot code, operating system code, and multiple application programs.

23. The method of claim 18, further comprises:

determining a fitness level for each mix mask in a seed population using a current cost of translation hardware associated with each mix mask, wherein a fitness level of a mix mask represents a level of benefit for providing a high level of compression.

24. The method of claim 23, wherein the current cost of translation hardware is based on an X-index and Y-index memory, an X pattern memory, and a Y pattern memory.

25. The method of claim 23, further comprises:

selecting an initial seed population of mix masks to include randomly selected mix masks and previously used mix masks which had a high fitness level for a previous program having instructions that are similar to the instructions used in the program.

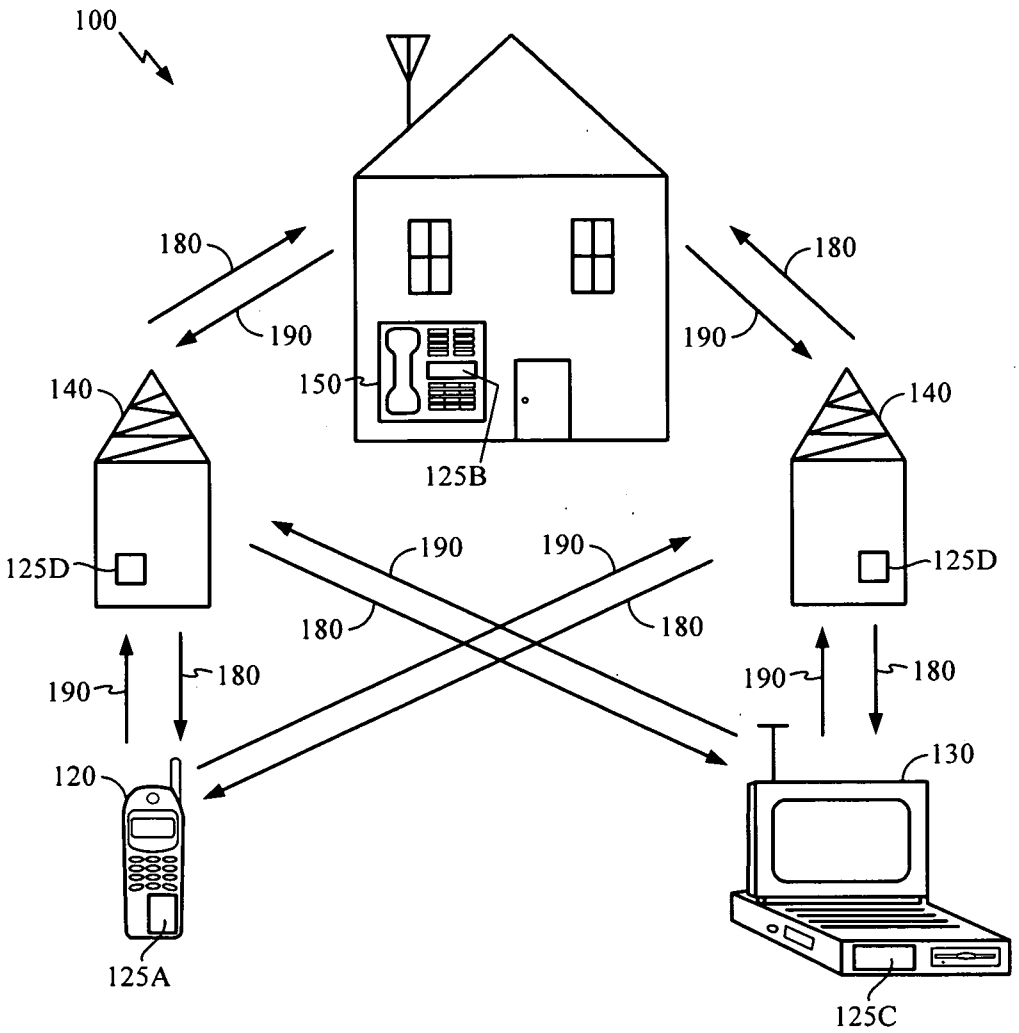
100
180
190
180
190
140
150
140
125B
125D
125D
190
190
180
180
190
180
190
180
120
130
125A
125C

FIG. 1

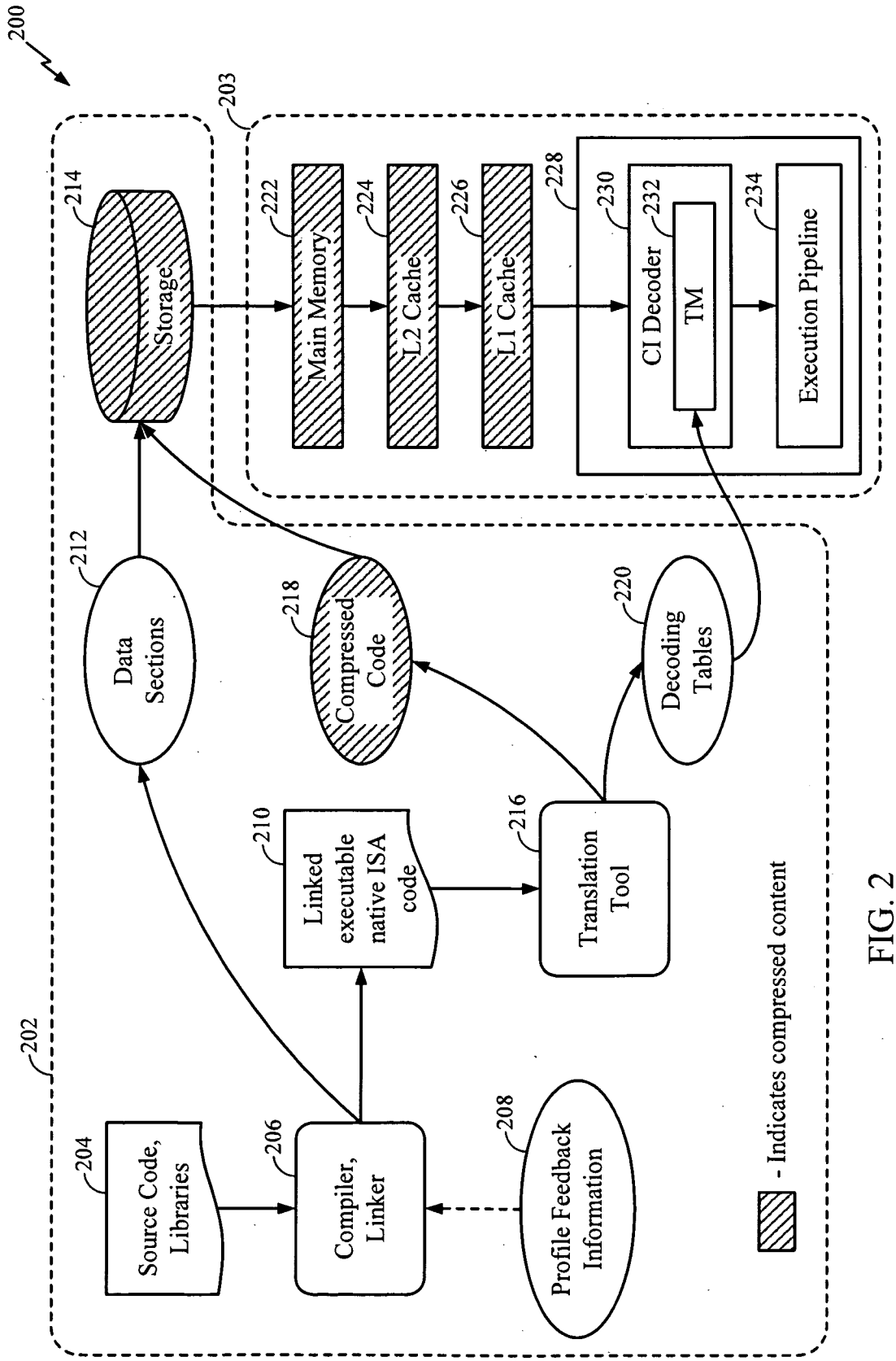
200
202
203
214
Storage
222
Main Memory
224
L2 Cache
226
L1 Cache
228
230
CI Decoder
232
TM
234
Execution Pipeline
212
Data Sections
218
Compressed Code
220
Decoding Tables
210
Linked executable native ISA code
216
Translation Tool
204
Source Code, Libraries
206
Compiler, Linker
208
Profile Feedback Information
- Indicates compressed content

FIG. 2

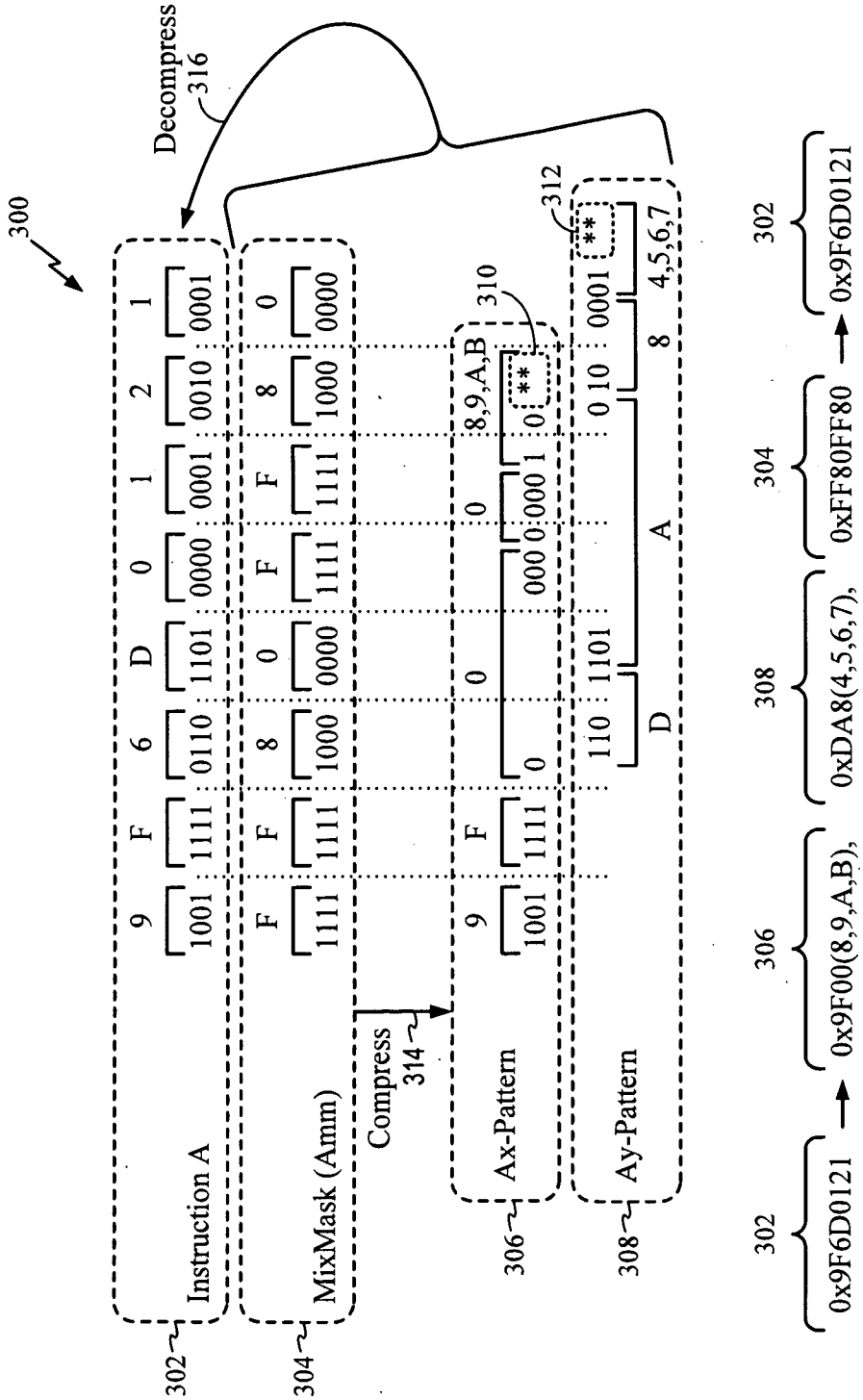
300
Decompress
316
Instruction A
MixMask (Amm)
Compress
314
Ax-Pattern
Ay-Pattern
302
304
306
308
310
312
9 F 6 D 0 1 2 1
1001 1111 0110 1101 0000 0001 0010 0001
F F 8 0 F F 8 0
1111 1111 1000 0000 1111 1111 1000 0000
9 F 0 8,9,A,B 0
1001 1111 0 ** 0 000 0 000 1 0 **
D A 8 4,5,6,7
110 1101 0 10 0001
0x9F6D0121
0x9F00(8,9,A,B),
0xDA8(4,5,6,7),
0xFF80FF80
0x9F6D0121
Hex notation identified by 0x prefix
** overlap pad bits

FIG. 3

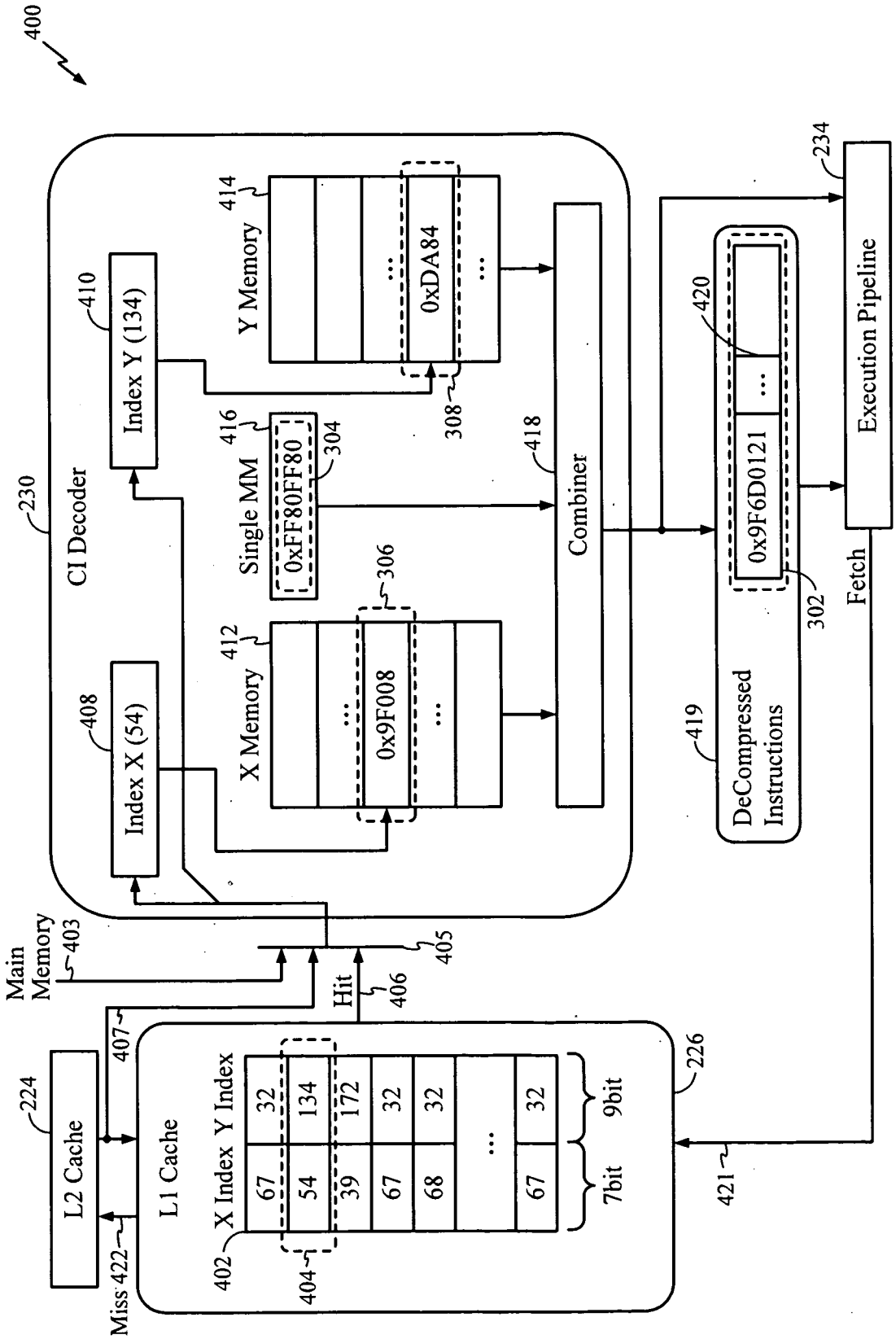
400
230
CI Decoder
Index Y (134)
410
Y Memory
414
0xDA84
308
Single MM
416
0xFF80FF80
304
418
Combiner
420
0x9F6D0121
DeCompressed Instructions
419
302
234
Execution Pipeline
Fetch
Index X (54)
408
X Memory
412
0x9F008
306
Main Memory
403
405
Hit
406
407
224
L2 Cache
L1 Cache
X Index Y Index
67 32
54 134
39 172
67 32
68 32
67 32
7bit
9bit
226
421
Miss 422
402
404

FIG. 4

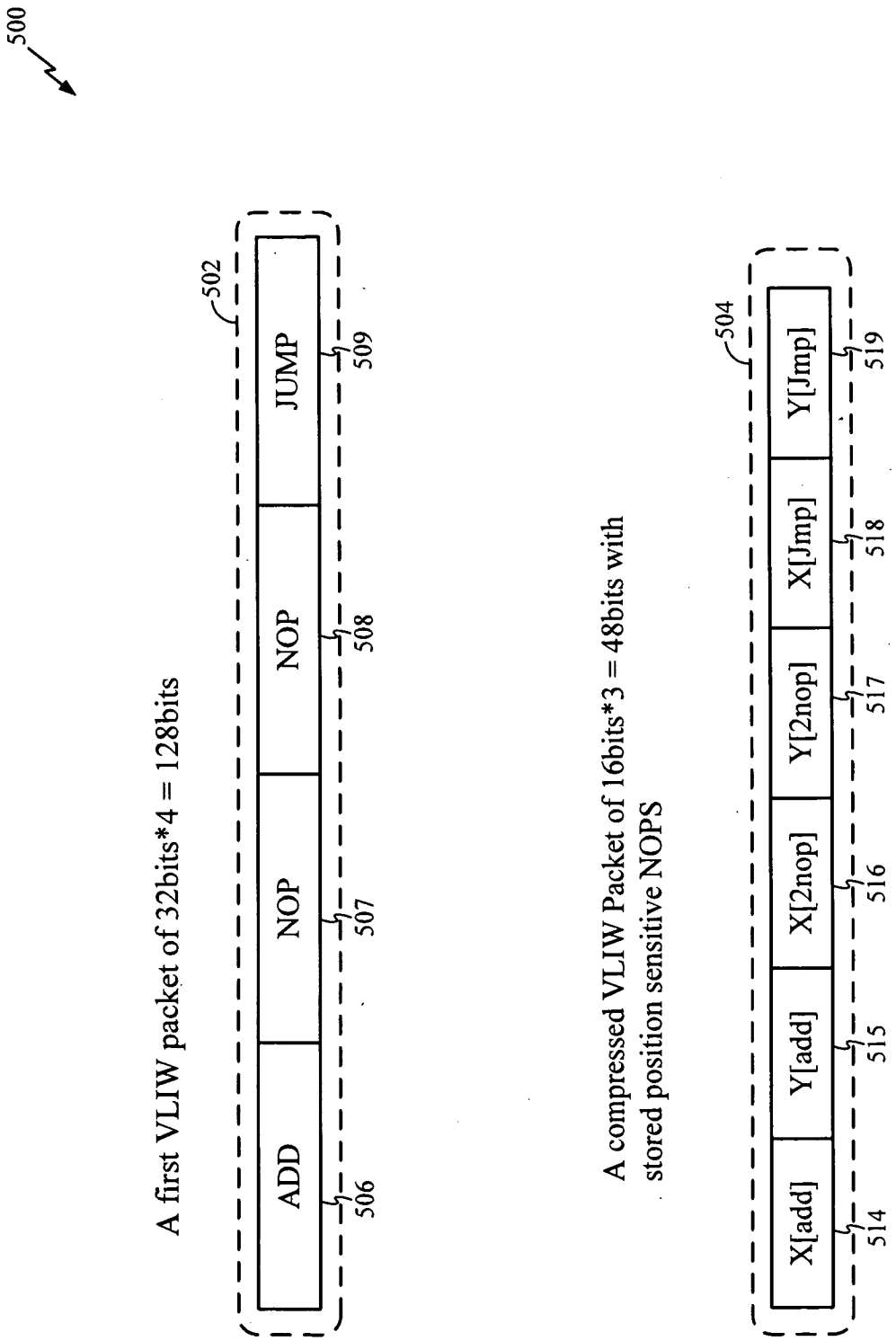
500
A first VLIW packet of 32bits*4 = 128bits
502
ADD
NOP
NOP
JUMP
506
507
508
509
A compressed VLIW Packet of 16bits*3 = 48bits with stored position sensitive NOPS
504
X[add]
Y[add]
X[2nop]
Y[2nop]
X[Jmp]
Y[Jmp]
514
515
516
517
518
519

FIG. 5

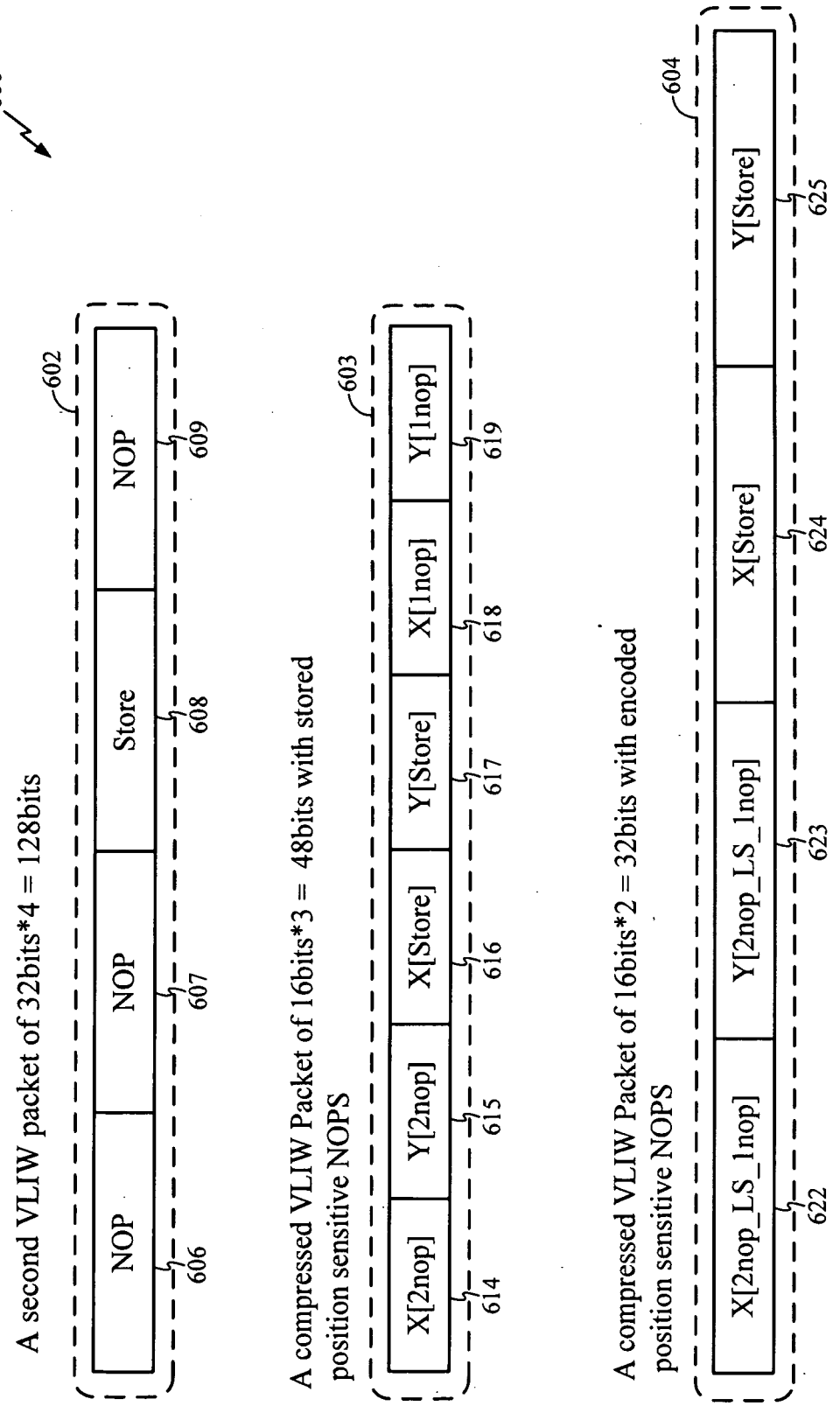
600
A second VLIW packet of 32bits*4 = 128bits
602
NOP
NOP
Store
NOP
606
607
608
609
A compressed VLIW Packet of 16bits*3 = 48bits with stored position sensitive NOPS
603
X[2nop]
Y[2nop]
X[Store]
Y[Store]
X[1nop]
Y[1nop]
614
615
616
617
618
619
A compressed VLIW Packet of 16bits*2 = 32bits with encoded position sensitive NOPS
604
X[2nop_LS_1nop]
Y[2nop_LS_1nop]
X[Store]
Y[Store]
622
623
624
625

FIG. 6

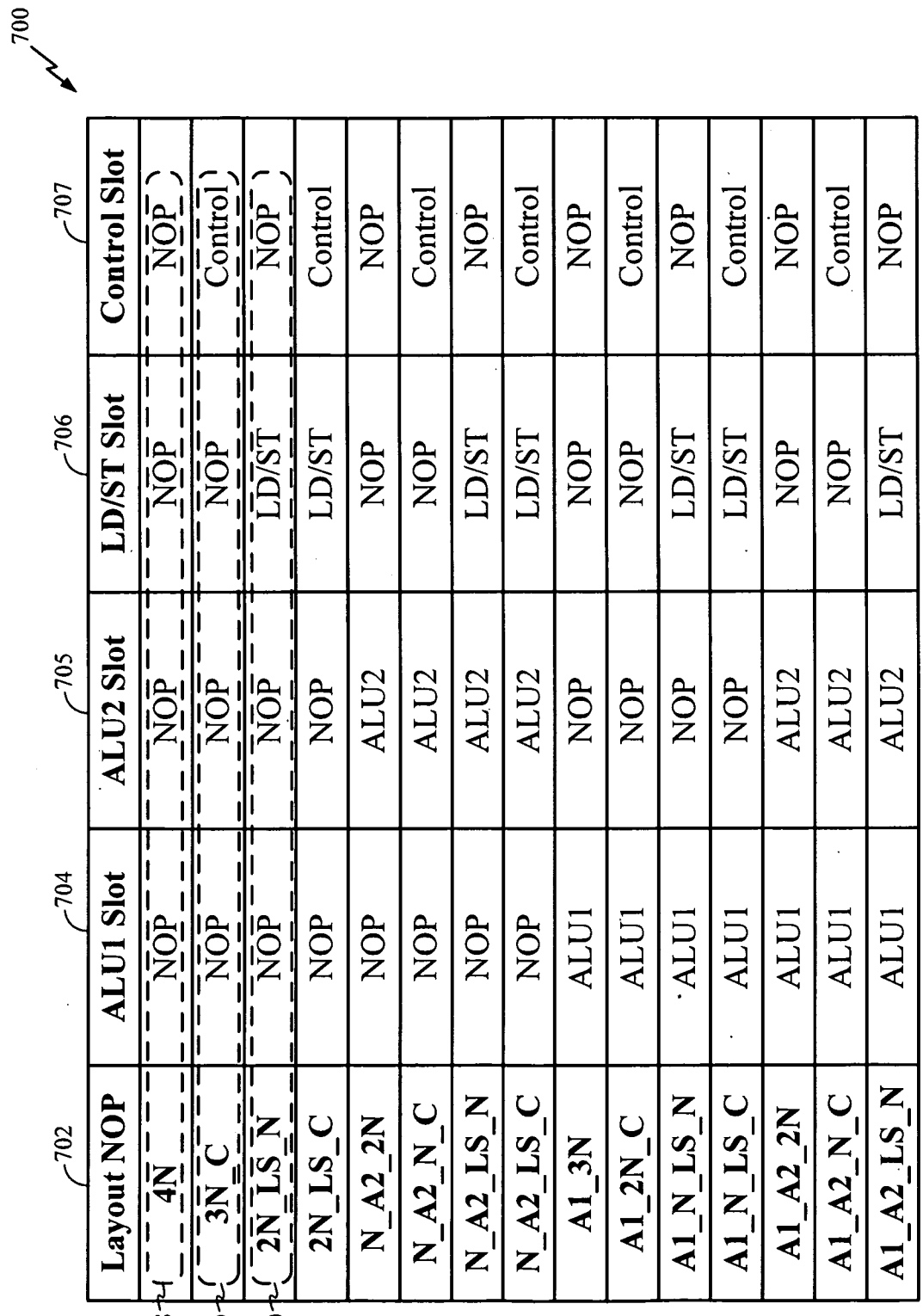
700
702
704
705
706
707
708
709
710
Layout NOP
ALU1 Slot
ALU2 Slot
LD/ST Slot
Control Slot
4N
NOP
NOP
NOP
NOP
3N_C
NOP
NOP
NOP
Control
2N_LS_N
NOP
NOP
LD/ST
NOP
2N_LS_C
NOP
NOP
LD/ST
Control
N_A2_2N
NOP
ALU2
NOP
NOP
N_A2_N_C
NOP
ALU2
NOP
Control
N_A2_LS_N
NOP
ALU2
LD/ST
NOP
N_A2_LS_C
NOP
ALU2
LD/ST
Control
A1_3N
ALU1
NOP
NOP
NOP
A1_2N_C
ALU1
NOP
NOP
Control
A1_N_LS_N
ALU1
NOP
LD/ST
NOP
A1_N_LS_C
ALU1
NOP
LD/ST
Control
A1_A2_2N
ALU1
ALU2
NOP
NOP
A1_A2_N_C
ALU1
ALU2
NOP
Control
A1_A2_LS_N
ALU1
ALU2
LD/ST
NOP

FIG. 7

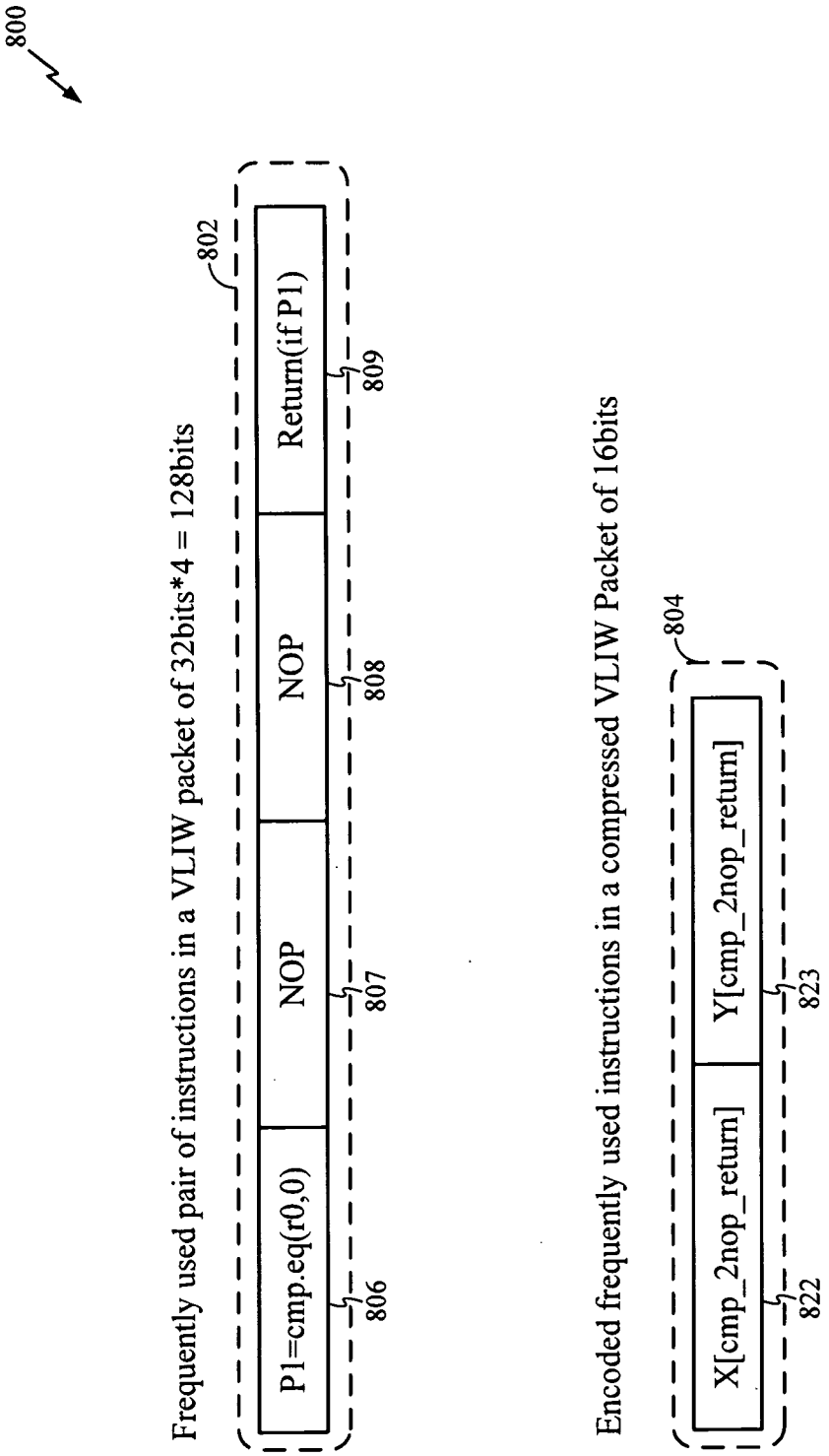
800
Frequently used pair of instructions in a VLIW packet of 32bits*4 = 128bits
802
P1=cmp.eq(r0,0)
NOP
NOP
Return(if P1)
806
807
808
809
Encoded frequently used instructions in a compressed VLIW Packet of 16bits
804
X[cmp_2nop_return]
Y[cmp_2nop_return]
822
823

FIG. 8

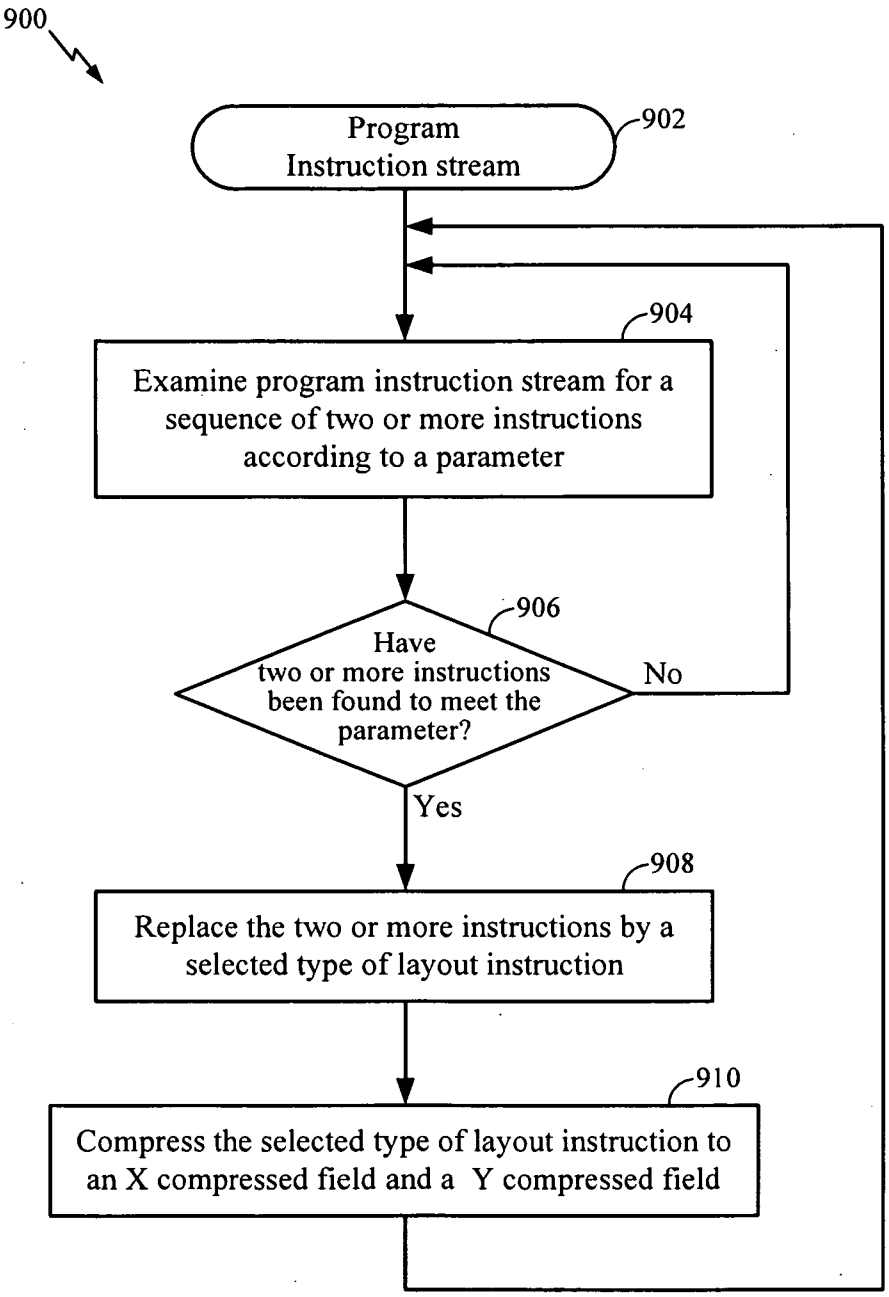
900
Program Instruction stream
902
904
Examine program instruction stream for a sequence of two or more instructions according to a parameter
906
Have two or more instructions been found to meet the parameter?
No
Yes
908
Replace the two or more instructions by a selected type of layout instruction
910
Compress the selected type of layout instruction to an X compressed field and a Y compressed field

FIG. 9A

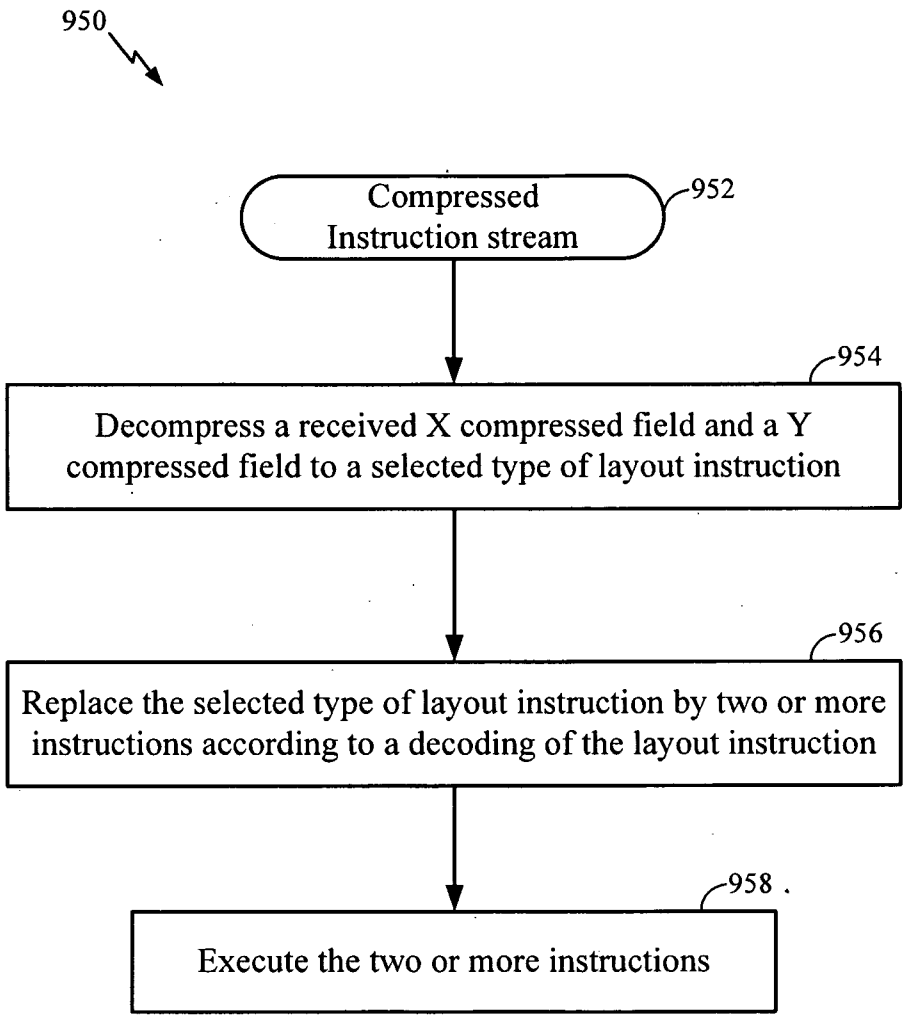
950
Compressed Instruction stream
952
954
Decompress a received X compressed field and a Y compressed field to a selected type of layout instruction
956
Replace the selected type of layout instruction by two or more instructions according to a decoding of the layout instruction
958
Execute the two or more instructions

FIG. 9B

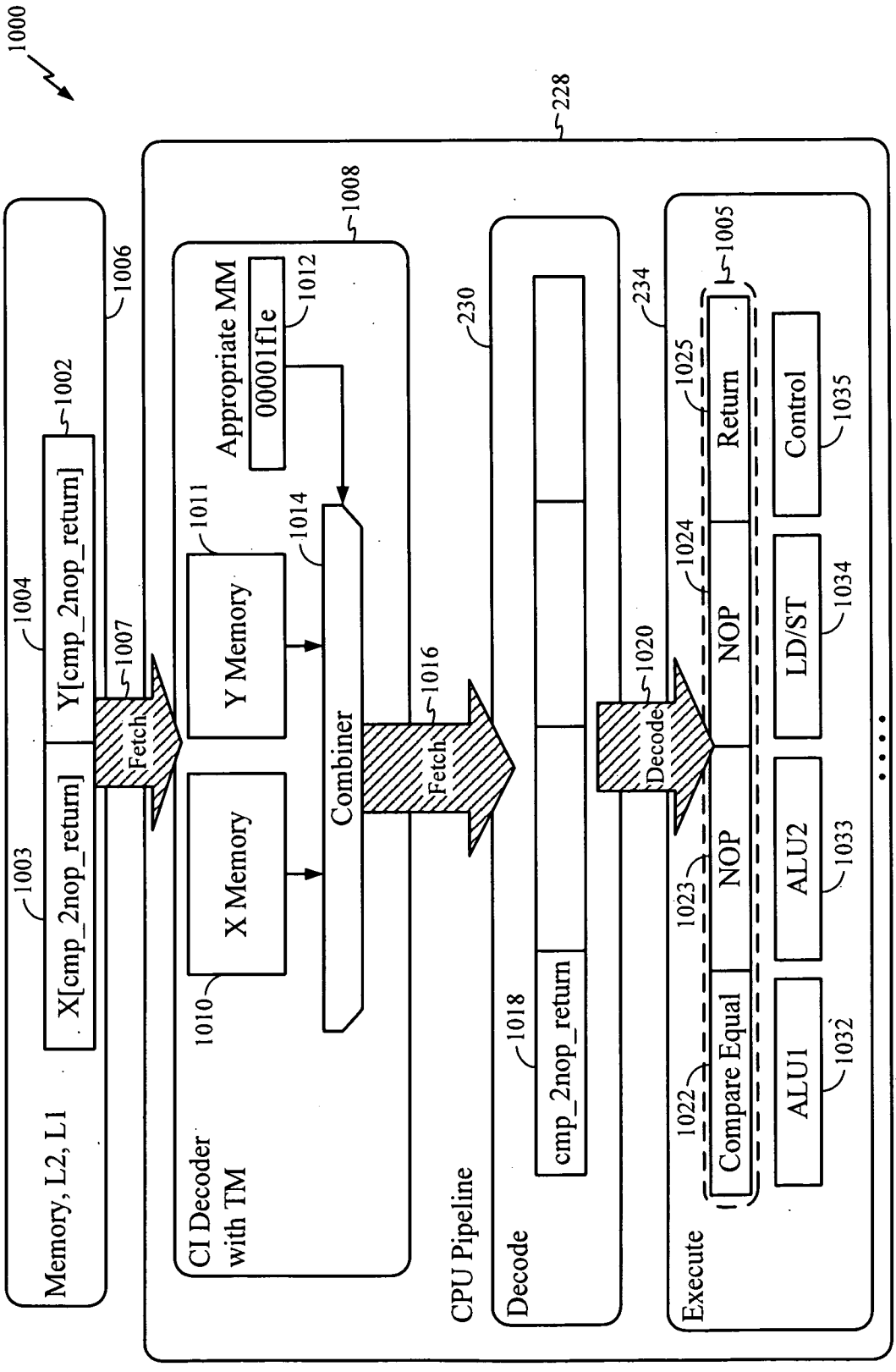
1000
228
Memory, L2, L1
1003
X[cmp_2nop_return]
1004
Y[cmp_2nop_return]
1002
1006
1007
Fetch
CI Decoder with TM
1008
Appropriate MM
00001f1e
1012
1010
X Memory
1011
Y Memory
1014
Combiner
1016
Fetch
CPU Pipeline
230
Decode
1018
cmp_2nop_return
234
1020
Decode
Execute
1005
1022
Compare Equal
1023
NOP
1024
NOP
1025
Return
1032
ALU1
1033
ALU2
1034
LD/ST
1035
Control
....

FIG. 10

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/036199

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/30
ADD.

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

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)
EPO-Internal, INSPEC, IBM-TDB, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 6 199 126 B1 (AUERBACH DANIEL JONATHAN [US] ET AL) 6 March 2001 (2001-03-06)	8,9
A	column 6, line 56 - column 7, line 19	1-7, 10-25

A	WO 2009/061814 A2 (UNIV FLORIDA [US]; MISHRA PRABHAT [US]; SEONG SEOK-WON [US]; BASU KANA) 14 May 2009 (2009-05-14) page 1, line 12 - line 35 page 7, line 37 - page 8, line 34; figures 2-4 page 9, line 22 - page 10, line 27; figures 5-8 page 27, line 36 - page 29, line 16; figures 32,35	1-25

	-/--	

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
4 September 2012	19/09/2012

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Thibaudeau, Jean

2

Form PCT/ISA/210 (second sheet) (April 2005)

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/036199

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	SEOK-WON SEONG ET AL: "Bitmask-Based Code Compression for Embedded Systems", IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, IEEE SERVICE CENTER, PISCATAWAY, NJ, US, vol. 27, no. 4, 1 April 2008 (2008-04-01), pages 673-685, XP011206308, ISSN: 0278-0070 the whole document -----	1-25

2

Form PCT/ISA/210 (continuation of second sheet) (April 2005)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2012/036199

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 6199126 B1	06-03-2001	NONE	
WO 2009061814 A2	14-05-2009	US 2010223237 A1	02-09-2010
		WO 2009061814 A2	14-05-2009