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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
18 October 2012 (18.10.2012)

WIPO | PCT

(10) International Publication Number
WO 2012/141871 A1

(51) International Patent Classification:
H03M 7/30 (2006.01) *G06F 9/445* (2006.01)

(21) International Application Number:
PCT/US2012/030345

(22) International Filing Date:
23 March 2012 (23.03.2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/474,183 11 April 2011 (11.04.2011) US

(71) Applicant *(for all designated States except US)*: **MARVELL WORLD TRADE LTD.** [BB/BB]; L'Horizon, Gunsite Road, Brittons Hill, St.Michael, 14027 (BB).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **MITCHEM, Jeff** [US/US]; 15021 Lantana Drive, Broomfield, CO 80023 (US). **SCHONKEREN, Wim** [BE/BE]; Wijerken 112, B-3920 Lommel (BE). **DE BEAUREGARD, Arnaud, Gouder** [NL/NL]; Gregoriuslaan 12, NL-6114 Susteren (NL).

(74) Agent: **KRAGULJAC, Petar**; KRAGULJAC LAW GROUP, LLC, 4700 Rockside Road, Summit One - Suite 510, Independence, OH 44131 (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, MD, RU,

[Continued on next page]

(54) Title: METHOD FOR COMPRESSION AND REAL-TIME DECOMPRESSION OF EXECUTABLE CODE

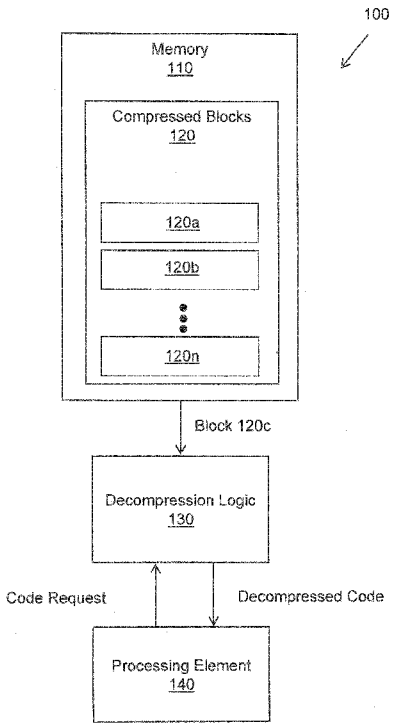

Figure 1

(57) Abstract: Systems, methods, and other embodiments associated with compression and real-time decompression of executable code are described. According to one embodiment, an apparatus includes a memory that stores compressed blocks of data. The data is executable code for a processing element. The apparatus also includes a decompression logic. The decompression logic receives a request from the processing element for data and determines a compressed block that stores the data. The compressed block is decompressed to produce an uncompressed block. The decompression logic then provides the requested data to the processing element. In one embodiment an uncompressed block has a predetermined fixed block size. The predetermined fixed block size is selected based on at least one of an amount of uncompressed data, a desired compression ratio, and a desired access time.

WO 2012/141871 A1

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).

Published:

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

METHOD FOR COMPRESSION AND REAL-TIME DECOMPRESSION OF EXECUTABLE CODE

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This patent disclosure claims the benefit of U.S. Provisional Application No. 61/474,183 filed on April 11, 2011, which is hereby wholly incorporated by reference.

BACKGROUND

[0002] The background description provided herein is for the purpose of generally presenting the context of the disclosure. Work of the presently named inventor(s), to the extent the work is described in this background section, as well as aspects of the description that may not otherwise qualify as prior art at the time of filing, are neither expressly nor impliedly admitted as prior art against the present disclosure.

[0003] Microprocessors and other processing elements access executable code. Executable code is thus stored in memory that is readily accessible to the microprocessor. In system-on-chip microprocessor products, the executable code is typically stored in Dynamic Random Access Memory (DRAM) or Synchronous Random Access Memory (SRAM). DRAM is well suited for storing executable code because it is relatively low cost and has a high density of memory cells, so that it takes up less space. One drawback to using DRAM to store executable code is that the DRAM is typically a separate component from the system-on-chip, necessitating external connections and increasing the overall footprint of the system-on-chip. SRAM or DRAM can also be included within the system-on-chip, eliminating the need for a separate component. This greatly increases the size, and hence cost of the system-on-chip.

[0004] In information theory and computer science, data compression involves encoding data using fewer bits than the data's original representation would use. Compression reduces the storage space needed to store the data. However, compressing data may also result in the necessity of data decompression, requiring additional time and processing. During decompression, the data is decoded to return the data to the original representation. Conventionally, compression and

decompression are performed as quickly as possible to limit the consumption of resources and interference with a user. To this end, popular compression algorithms (e.g., LZ77) attempt to balance speed of compression with the effectiveness of the compression. Thus, while being capable of high speed performance, these compression algorithms may not provide maximal compression.

SUMMARY

[0005] In one embodiment, an apparatus includes a memory that stores compressed blocks of data. The data is executable code for a processing element. The apparatus also includes a decompression logic. The decompression logic receives a request from the processing element for data and determines a compressed block that stores the data. The compressed block is decompressed to produce an uncompressed block. The decompression logic then provides the requested data to the processing element.

[0006] In one embodiment an uncompressed block has a predetermined fixed block size. The predetermined fixed block size is selected based on at least one of an amount of uncompressed data, a desired compression ratio, and a desired access time.

[0007] In another embodiment, a method includes storing data in compressed blocks. The data is executable code for a processing element. A request is received from the processing element for data. It is then determined which compressed block stores the data. The compressed block is decompressed to produce an uncompressed block. The requested data is provided to the processing element.

[0008] In one embodiment uncompressed blocks are of a predetermined fixed block size. To determine which compressed block stores the data, the address of the requested uncompressed block is divided by the fixed block size to determine a block number that identifies a compressed block that stores the data.

[0009] In another embodiment a pair of bits of uncompressed executable code is selected. It is determined whether the pair of bits has been previously copied to a copy bin. If the pair of bits has not been previously copied to the copy bin, the pair of bits is copied to the copy bin. If the pair of bits has been previously copied to the copy bin, an additional bit is added to form a segment. It is then determined if the segment has been previously copied to the copy bin. If the segment has not been previously copied to the

copy bin, a pointer pointing to a location of the pair of bits in the copy bin is stored in a compressed block. If the segment has been previously copied, an additional bit is added to the segment to form a longer segment, until the longer segment is determined as not having been copied previously. A pointer is stored in the compressed block. The pointer points to the longest segment determined as having been previously copied to the copy bin.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] The accompanying drawings, which are incorporated in and constitute a part of the specification, illustrate various systems, methods, and other embodiments of the disclosure. It will be appreciated that the illustrated element boundaries (e.g., boxes, groups of boxes, or other shapes) in the figures represent one example of the boundaries. One of ordinary skill in the art will appreciate that in some examples one element may be designed as multiple elements or that multiple elements may be designed as one element. In some examples, an element shown as an internal component of another element may be implemented as an external component and vice versa. Furthermore, elements may not be drawn to scale.

[0011] Figure 1 illustrates one embodiment of an apparatus associated with compression and real-time decompression of executable code.

[0012] Figure 2A illustrates one embodiment of an apparatus associated with compression and real-time decompression of executable code.

[0013] Figure 2B illustrates one embodiment of executable code being compressed into a compressed block.

[0014] Figure 2C illustrates one embodiment of an apparatus associated with compression and real-time decompression of executable code.

[0015] Figure 3 illustrates one embodiment of a method associated with compression and real-time decompression of executable code.

[0016] Figure 4A illustrates one embodiment of a memory associated with compression and real-time decompression of executable code

[0017] Figure 4B illustrates one embodiment of a memory associated with compression and real-time decompression of executable code.

[0018] Figure 5 illustrates one embodiment of a method associated with compression of executable code that facilitates real-time decompression of the executable code.

[0019] Figure 6 illustrates one embodiment of an apparatus for real-time decompression of executable code.

DETAILED DESCRIPTION

[0020] Described herein are examples of systems, methods, and other embodiments associated with compression and real-time decompression of executable code. In some circumstances, it is desirable to store executable code in on-chip memory (e.g., static random-access memory (SRAM), flash memory). For example, some system-on-chip products enable execution of code that is stored in on-chip flash memory. However, the amount of executable code may exceed the capacity of the on-chip flash memory. Flash memory is relatively costly and also requires more space per bit of storage than other types of memory. Rather than incurring the cost and space of providing additional on-chip memory, according to the systems and methods described herein, the executable code is compressed to fit in the available flash memory. When the executable code is requested by the system-in-chip processing element, the compressed executable code is decompressed so that is available virtually immediately (e.g., in "real time") to the processing element. Other system-on-chip products store code in flash memory in compressed form, then download and decompress the code before storing it in on-chip SRAM or DRAM. The code is then executed from the on-chip SRAM or DRAM. This requires a large amount of on-chip SRAM or DRAM which greatly increases the size and cost of the system-on-chip.

[0021] When compressed data corresponds to executable code for a processing element, the compression is performed once while the decompression is performed repetitively and "on-the-fly". Accordingly, while the decoding should be performed quickly, the compression may not be subject to such stringent speed constraints. The compression operation may be allowed additional time to most effectively compress the executable code. The compression ratio is increased using compression techniques described herein, that provide layered procedures to tightly compress the executable

code. Increasing the compression ratio allows the compressed executable code to fit within a limited amount of memory and facilitates storing the compressed executable code in on–chip memory.

[0022] When the executable code is requested for processing, a compressed block storing the executable code is decompressed and the executable code is processed in real time. The compression and decompression techniques described herein facilitate decompressing executable code and providing it to the processing element in real time. In this manner, a processing element is able to process executable code in real time without requiring additional on-chip memory or sacrificing speed. The systems and methods herein are described in the context of a system-on-chip that stores executable code in compressed form in on-chip flash memory. However, compression and real time decompression of executable code, or any other type of data, as described herein may be performed in any number of processing environments and with any number of memory types.

[0023] With reference to Figure 1, one embodiment of an apparatus 100 is shown that is associated with real-time decompression of executable code. The executable code is stored in a memory 110 as compressed blocks 120a-120n. The executable code may include processing instructions, user data, or system code. When the executable code is requested by a processing element 140, the a decompression logic 130 selects a compressed block (e.g., compressed block 120c) storing the executable code. The executable code is compressed into the compressed blocks 120 individually. Therefore, the selected compressed block 120c can be decompressed independently by the decompression logic 130 to retrieve the requested executable code. The compressed block 120c is sent to a decompression logic 130 to be decompressed. Once the compressed block has been decompressed, the decompressed block is returned to processing element 140 by the decompression logic 130.

[0024] Figure 2A illustrates one embodiment of an apparatus 200 associated with compression and real-time decompression of executable code. Figure 2 includes the memory 110 having the compressed blocks 120 shown in Figure 1. The memory 110 operates in a similar manner as described with respect to Figure 1. The apparatus 200 also includes a compression logic 150 to compress executable code into the

compressed blocks 120. The compression logic 150 is configured to copy bits from uncompressed blocks and store them in corresponding compression blocks. In one embodiment, the compression logic performs a compression method outlined in Figure 3. Unlike many compression algorithms, the compression method performed by the compression logic 150 is not limited by time constraints. While this makes the compression method more time consuming than other methods, it provides a tight compression of the executable code. One embodiment of a configuration of compressed code as stored in memory is illustrated in Figure 4.

[0025] Figure 2B illustrates one embodiment of executable code 210 being compressed into a compressed block 120a. The executable code 210 include bits or words 211-218. A bit contains a value. The bit 211 has a value 1, the bit 212 has a value 1, and the bit 213 has a value 1. The bits 211-213 for a segment 221. A segment represents a maximal length of bits that has already been stored in memory. Thus, a segment of bits having values 1, 1, and 1, like the segment 221, has been previously stored but the bits 211-214 have not.

[0026] A bit 214 has a value 1 and a bit 215 has a value 0. A segment having the values 1 followed by 0 has been previously stored. Thus, the bit 214 and the bit 215 form a segment 222. A bit 216 has a value 0, a bit 217 has a value 0, and a bit 218 has a value 1. A segment having values 0, followed by 0, followed by 1 has been previously stored in the memory. Therefore, bits 216-218 form a segment 223. Segments 221, 222, and 223 are stored as compressed block 120a.

[0027] Figure 2C illustrates one embodiment of a memory 110 associated with compression and real-time decompression of executable code. Segments are stored in an area of the memory 110 allocated as a copy bin 115. To determine whether a segment has been previously saved to the memory 110, the segment of bits is searched for in the copy bin 115. Once a segment is identified as having the maximum number of bits previously stored in the copy bin 115, the segment is used for block compression.

[0028] Figure 3 illustrates one embodiment of a method associated with compression of executable code. To compress the executable code into compressed blocks, the method includes at 310 isolating a pair of words including a first word and a second word of executable code. A word includes at least one bit. At 320, it is

determined whether the pair of words has been previously copied. If the pair has not been previously copied, the pair is considered a literal item. Literal items are copied to a copy bin for storage. Accordingly, the pair is copied to the copy bin at 330 and the method returns to 310. If the pair of words has been previously copied, the pair of words is considered a copy item. To compress the data, a pointer is stored that points to the longest string of copy items available. Accordingly, to determine whether a pair represents the longest string of copy items that can be constructed, at 340 the next word in the executable code is added to the pair to form a three word segment.

[0029] At 350, it is determined whether the segment has been previously copied. In the present example, it is determined whether the three word segment is a copy item. If three word segment has not been previously copied to the copy bin, the pair represents the longest possible segment of previously copied words. Therefore, at 360, a pointer to the copy bin pointing back in the copy bin to the last occurrence of the pair is stored. The pointer includes an offset and a count. The offset indicates the location of the previously copied pair in the copy bin. The count indicates the length, in words, of the copy item. In this example, the copy item includes the first word and the second word of executable code. Accordingly, the count for the copy item is two words.

[0030] Conversely, if at 350, if it is determined that the segment is present in the copy bin as a copy item, the method 300 returns to 340 to add an additional word to the segment. In the example discussed, if the three word segment has been previously copied to the copy bin, the next word in the executable code is added to the pair to form a four word segment at 340. Thus, as it is determined that a segment has been previously copied to the copy bin additional words are added until the segment is not found in the copy bin. Once a segment is not found in the copy bin, it is determined that the segment absent the last word added is the longest possible segment. In this manner, the compression method creates copy items of maximal length to enhance the amount of compression achieved. The maximum length of a copy item may have a predetermined limit. In one embodiment, the maximum length of a copy item is 23 words.

[0031] Figure 4A illustrates one embodiment of a memory that stores compressed executable code. The memory 110 may be a flash memory, quad-serial flash memory,

electrically erasable programmable read-only memory (EEPROM), SRAM, or other storage device. The memory 110 stores compressed executable code in compressed blocks. Each compressed block is comprised of at least one group. The number of groups in a block varies with the compressability of the block. For clarity, compressed block 120a is illustrated with a single group. The group contains a control byte 410, having 8 bits a-h, and up to eight (8) items 420a – 420h. Each bit in the control byte 410 corresponds to one of the eight items 420a – 420h. For example, bit (a) in the control byte 410 corresponds to item 420a, bit (b) in the control byte corresponds item 420b, and so on.

[0032] Each bit in the control byte 410 indicates whether the corresponding item is a literal item that has not been copied previously or a copy item that has been copied previously. For example, a "0" value for bit (a) indicates that item 420a is a literal item, while a "1" indicates the item 420a is a copy item. The compressed blocks 120 may be additionally compressed by Huffman encoding of nibbles of the control byte 410. A nibble is a four bit portion of a byte. A nibble of a control byte may be Huffman encoded into a field that contains 2 to 5 bits, such that the encoded control byte has a length of 4 to 10 bits. Likewise, literal items may be Huffman encoded. Huffman encoding identifies frequently encoded nibbles. The frequently encoded nibbles may be stored in a Huffman table based on the presence of the frequently encoded nibbles or may be preprogrammed into a Huffman table. Huffman encoding is one example of a technique used to further compress the compressed blocks. Alternative techniques may be used.

[0033] Groups are not aligned on byte or nibble boundaries. Thus, after encoding the control byte 410, literal items, and copy items into a group, the group will have a length that is not necessarily a multiple of 4 or 8. The control byte 410, literal items, and copy items are packed together as one long string of bits. However, blocks are aligned on byte boundaries. Thus, bits with a predetermined value, such as "0" are inserted at the end of a block to complete a block, so that the following block will start on a byte boundary.

[0034] Figure 4B illustrates one embodiment of a memory that stores compressed executable code. In addition to storing compressed blocks 120, the memory 110 stores data about the compressed blocks 120. The memory 110 stores a validation code 430,

a configuration field 440, a compression block size 450, and block lookup table 460. The validation code 430 is an eight bit code that indicates whether the memory 110 contains valid data.

[0035] The configuration field 440 is a five bit field that provides information about the compression procedure. In one embodiment, the first two bits of the five bit configuration field 440 indicate the amount of data and executable code stored in the memory 110. The third bit of the five bit configuration field 440 indicates whether checksum bytes are being used. A checksum byte is computed to detect an error introduced during a block's transmission or storage. For example, a "1" in this field indicates that a checksum byte is included at the end of a compressed block. The fourth bit of the five bit configuration field 440 indicates whether the executable code is **s**tored in an uncompressed form or a compressed form. The fifth bit of the five bit configuration field 440 indicates whether nibble encoding has been employed. Nibble encoding is a fixed four bit length encoding that may, based on the executable code, improve compression.

[0036] The compression block size 450 specifies the fixed block size of uncompressed blocks. Uncompressed blocks may have a fixed size so that the compressed blocks can be readily located in the memory 110. The size of the uncompressed block affects the compression ratio and the access time of a compressed block. A smaller uncompressed block size may have better access time, but may provide poor compression as compared to a bigger uncompressed block size. Bigger uncompressed blocks typically have longer access time but better compression. In one example, the memory 110 is configured to determine the fixed block size by computing the smallest block size that achieves a predetermined level of compression. Alternatively, the memory 110 may be preprogrammed with a fixed block size.

[0037] In one embodiment, block size may be indicated in the compression block size 450 with a three bit code. In one example, a three bit binary code "000" indicates that compression will not be used, "001" indicates a fixed block size of 128 bytes, "010" indicates a fixed block size of 256 bytes, "011" indicates a fixed block size of 512 bytes, "100" indicates a fixed block size of 1024 bytes, and "101" indicates a fixed block size of 2048 bytes. Alternatively, the compression block size 450 of memory 110 may also

indicate that a variable uncompressed block size was used. In the event that a variable uncompressed block size was used, additional data may be included in a lookup table to locate the compressed blocks.

[0038] To locate blocks in the memory 110, the memory includes a block lookup table 460. The block lookup table 460 includes a block offset 470, a compression flag 480, and a parity entry 490. Uncompressed executable code is divided into N uncompressed blocks. Uncompressed blocks of executable code are assigned a block number (e.g., 1, 2, 3... N) that will remain the block number once the uncompressed blocks have been compressed. For example, compressed block (A) 120a corresponds to uncompressed block number 1 and compressed block (N) 120n corresponds to uncompressed block number N. In the described embodiment, the uncompressed blocks are a fixed size. Therefore, a starting address for a block can be determined by dividing the total amount of uncompressed data by that uncompressed block's number. The starting address of an uncompressed block is stored as the offset address 470.

[0039] Some uncompressed blocks expand when compressed rather than being reduced in size. Rather than storing the larger compressed blocks, blocks that grow in response to being compressed are stored in uncompressed form. Compression flag 480 indicates whether a block is stored in compressed form or uncompressed form. The compression flag 480 is appended to the offset address 470.

[0040] Parity entry 490 provides error detection for the block lookup table 460. The parity entry 490 is comprised of two bits that provide odd parity based on the offset address 470. While for purposes of clarity a specific number of bits have been given for different entries such as the eight bit validation code, five bit configuration field, and two bit parity entry, more or fewer bits may be used. The numbers of bits are given to illustrate the possible form and function of memory 110, but are not intended to be limiting.

[0041] Figure 5 illustrates one embodiment of a method associated with compression of executable code that facilitates real-time decompression of the executable code. At 510, a request is for executable code is received from a processing element. The processing element may be a system-on-chip configured to access a flash memory for executable code.

[0042] At 520, the method includes determining which compressed block stores the requested executable code. To determine which compressed block stores the executable code, the uncompressed block that stored the executable code is identified. The uncompressed block is identified by the block number of the uncompressed block. The compressed block storing the executable code is identified in the block lookup table using the block number and block offset. Once the correct compressed block is identified, at 530 the compressed block is fetched from memory. At 540, the decompression logic decompresses the compressed block. The decompressed block containing the executable code is returned to the processing element at 550.

[0043] Figure 6 illustrates one embodiment of an apparatus for real-time decompression of executable code. Figure 6 includes the memory 110 having the compressed blocks 120 and the processing element 140 shown in Figure 1. The memory 110 and the processing element 140 operate in a similar manner as described with respect to Figure 1. The apparatus 600 also includes the decompression logic 130. The decompression logic includes a read controller 610, a flash controller 620, a block buffer controller 630, and a group decompressor 640.

[0044] The read controller 610 accepts requests from the processing element 140 for executable code. The executable code is stored in compressed blocks 120 in memory 110. However, the request from the processing element is not in terms of compressed blocks 120. Instead, the request is in terms of bytes of executable code. For example, the processing element 140 may request a 32-byte uncompressed block. The read controller 610 initiates a request to the block buffer controller 630 and/or the memory 110 for the compressed block storing the requested executable code.

[0045] The block buffer controller 630 stores decompressed blocks to provide a level of caching and reduce the frequency of requests to the memory 110. Although the processing element 140 may only request a portion of a compressed block, the entire compressed block, which may have a size ranging from 128-2048 bytes, is decompressed and stored in the block buffer controller 630. Thus, in the event that the processing element subsequently requests the next portion of the executable code from the decompressed compressed block, the executable code is already stored in the block buffer controller 630.

[0046] The block buffer controller 630 may be bifurcated into two buffers each able to store a decompressed block. Thus, the block buffer controller 630 may be configured to store two decompressed blocks. When a third block is decompressed, the decompressed block stored in the block buffer controller 630 last accessed for executable code is deleted, allowing the third decompressed block to be stored in its place in the block buffer controller 630. Any number of cache replacement schemes may be employed by the bock buffer controller 630 in selecting which decompressed blocks should be maintained in the buffers. Alternatively, if the block size is less than the size of the buffers, the block buffer controller 630 may be further divided to store a plurality of decompressed blocks in each buffer.

[0047] If a decompressed block storing the executable code is not in the block buffer controller 630, flash controller 620 proceeds by initiating a block lookup table request from the memory 110 to determine which of the compressed blocks 120 the executable is stored in (see, e.g., block lookup table 460 in Figure 4). Once determining the location of the compressed block, the compressed block is retrieved from the memory 110. The flash controller 620 sends the retrieved compressed block to the decompressor 640 to be decompressed. The decompressed block storing the requested executable code is then stored in the block buffer controller 630. The flash controller 620 initiates a request for the requested executable code from the block buffer controller 630.

[0048] The decompressor 640 decompresses a compressed block that contains executable code that has been requested by the read controller 610 and is not already stored in the block buffer controller 630. The decompressor 640 holds a data stream of bits from a selected compressed block 120 from the memory 110 in a shift register. In one example, the shift register is a 24 bit shift register. The decompressor 640 decodes a data stream that has been subject to nibble encoding in the shift register starting at bit 11 and returns a 4 bit nibble of decoded data and a shift-left value. The shift-value indicates that the shift register is to be shifted to the left by a predetermined number of bits, such as 2 to 5. The decompressor 640 decodes data in the data stream that has not been subjected to nibble encoding in the shift register starting at bit 11 and returns a 5 bit length value and a shift-left value. The shift-value indicates that the shift register is

to be shifted to the left by a predetermined number of bits. Once the data stream has been decoded, it is stored in the block buffer controller 630.

[0049] The following includes definitions of selected terms employed herein. The definitions include various examples and/or forms of components that fall within the scope of a term and that may be used for implementation. The examples are not intended to be limiting. Both singular and plural forms of terms may be within the definitions.

[0050] References to "one embodiment", "an embodiment", "one example", "an example", and so on, indicate that the embodiment(s) or example(s) so described may include a particular feature, structure, characteristic, property, element, or limitation, but that not every embodiment or example necessarily includes that particular feature, structure, characteristic, property, element or limitation. Furthermore, repeated use of the phrase "in one embodiment" does not necessarily refer to the same embodiment, though it may.

[0051] "Logic", as used herein, includes but is not limited to hardware, firmware, instructions stored on a non-transitory medium or in execution on a machine, and/or combinations of each to perform a function(s) or an action(s), and/or to cause a function or action from another logic, method, and/or system. Logic may include a software controlled microprocessor, a discrete logic (e.g., ASIC), an analog circuit, a digital circuit, a programmed logic device, a memory device containing instructions, and so on. Logic may include one or more gates, combinations of gates, or other circuit components. Where multiple logics are described, it may be possible to incorporate the multiple logics into one physical logic. Similarly, where a single logic is described, it may be possible to distribute that single logic between multiple physical logics. One or more of the components and functions described herein may be implemented using one or more of the logic elements.

[0052] While for purposes of simplicity of explanation, illustrated methodologies are shown and described as a series of blocks. The methodologies are not limited by the order of the blocks as some blocks can occur in different orders and/or concurrently with other blocks from that shown and described. Moreover, less than all the illustrated blocks may be used to implement an example methodology. Blocks may be combined

or separated into multiple components. Furthermore, additional and/or alternative methodologies can employ additional, not illustrated blocks.

[0053] To the extent that the term "includes" or "including" is employed in the detailed description or the claims, it is intended to be inclusive in a manner similar to the term "comprising" as that term is interpreted when employed as a transitional word in a claim.

[0054] While example systems, methods, and so on have been illustrated by describing examples, and while the examples have been described in considerable detail, it is not the intention of the applicants to restrict or in any way limit the scope of the appended claims to such detail. It is, of course, not possible to describe every conceivable combination of components or methodologies for purposes of describing the systems, methods, and so on described herein. Therefore, the disclosure is not limited to the specific details, the representative apparatus, and illustrative examples shown and described. Thus, this application is intended to embrace alterations, modifications, and variations that fall within the scope of the appended claims.

Claims

What is claimed is:

1. An apparatus, comprising

a memory configured to store compressed blocks of data, wherein the data is executable code for a processing element; and

a decompression logic configured to:

receive a request from the processing element for data;

determine a compressed block that stores the data;

decompress the compressed block to produce an uncompressed block; and

provide the requested data to the processing element.

2. The apparatus of claim 1, wherein the decompression logic further comprises a block buffer controller configured to store a plurality of decompressed blocks and further wherein when the decompression logic determines that a decompressed block corresponding to the compressed block is stored in the block buffer controller, the decompression logic provides the decompressed block from the block buffer controller to the processing element.

3. The apparatus of claim 1, further comprising a compression logic configured to:

receive uncompressed data;

parse the uncompressed data into at least one uncompressed block;

compress the at least one uncompressed block into a compressed block; and

store the compressed block in the memory.

4. The apparatus of claim 3, wherein the at least one uncompressed block has a predetermined fixed block size, wherein the predetermined fixed block size is selected based on at least one of an amount of uncompressed data, a desired compression ratio, and a desired access time.

5. The apparatus of claim 3, wherein the compression logic further includes a copy bin, and wherein the compression logic is further configured to:

store a plurality of segments in the copy bin, wherein a segment comprises a sequence of bits of executable code;

select a given segment from the plurality of segments;

determine if the given segment is present in the copy bin, and when the given segment is present in the copy bin, iteratively augment the given segment with additional bits of executable code and determine if the augmented segment is present in the copy bin to determine a segment of maximum length that is present in the copy bin; and

store, in the compressed block, a pointer to a location of the segment of maximum length in the copy bin.

6. The apparatus of claim 3, wherein compression logic is configured to:

perform an initial compression on the at least one uncompressed block to form a partially compressed block; and

perform a second compression of the partially compressed block using a second compression technique to form a given compressed block.

7. The apparatus of claim 3, further comprising a block look up table logic that stores, for each uncompressed block of executable code:

a block number derived from an address of the at least one uncompressed block; and

an offset value indicative of the location of the corresponding compressed block in the memory.

8. The apparatus of claim 1, wherein the memory is a flash memory.

9. A method, comprising

storing data in compressed blocks, wherein the data is executable code for a processing element;

receiving a request from the processing element for data; and
determining a compressed block that stores the data;
decompressing the compressed block to produce an uncompressed block; and
providing the requested data to the processing element.

10. The method of claim 9, further comprising determining if the uncompressed block is already stored in a buffer and when the uncompressed block is stored in the buffer, providing the requested data from the buffer.

11. The method of claim 9, wherein uncompressed blocks are of a predetermined fixed block size, wherein the determining comprises dividing an address of the requested uncompressed block by the fixed block size to determine a block number that identifies a compressed block that stores the data.

12. The method of claim 11, wherein the determining comprises accessing a lookup table that stores, for each compressed block, a block number and an offset value indicative of the location of the corresponding compressed block in memory.

13. A method, comprising:
selecting a pair of bits of uncompressed executable code;
determining if the pair of bits has been previously copied to a copy bin; and
when the pair of bits has not been previously copied to the copy bin, copying the pair of bits to the copy bin; and
when the pair of bits has been previously copied to the copy bin, adding an additional bit to form a segment;
determining if the segment has been previously copied to the copy bin; and
when the segment has not been previously copied to the copy bin, storing, in a compressed block, a pointer to a location of the pair of bits in the copy bin; and

when the segment has been previously copied, adding an additional bit to form a longer segment, until the longer segment is determined as not having been copied previously, and

storing, in the compressed block, a pointer to the longest segment determined as having been previously copied to the copy bin.

14. The method of claim 13, wherein the pointer identifies where a last occurrence of the previously stored pair or longest segment is stored in the copy bin.

15. The method of claim 13, further comprising performing a compression of the compressed block using Huffman encoding.

16. The method of claim 15, wherein the Huffman encoding is performed using a preprogrammed Huffman table.

17. The method of claim 15, further comprising:

performing nibble encoding on the compressed block;

identifying frequently encoded nibbles; and

generating a Huffman table based, at least in part on, the frequently encoded nibbles.

18. The method of claim 13, further comprising:

receiving the executable code; and

parsing the executable code into uncompressed blocks.

19. The method of claim 18, wherein the uncompressed blocks have a predetermined fixed block size, wherein the predetermined fixed block size is selected based on at least one of an amount of executable code, a desired compression ratio, and a desired access time.

20. The apparatus of claim 18, wherein the uncompressed blocks have a variable block size.

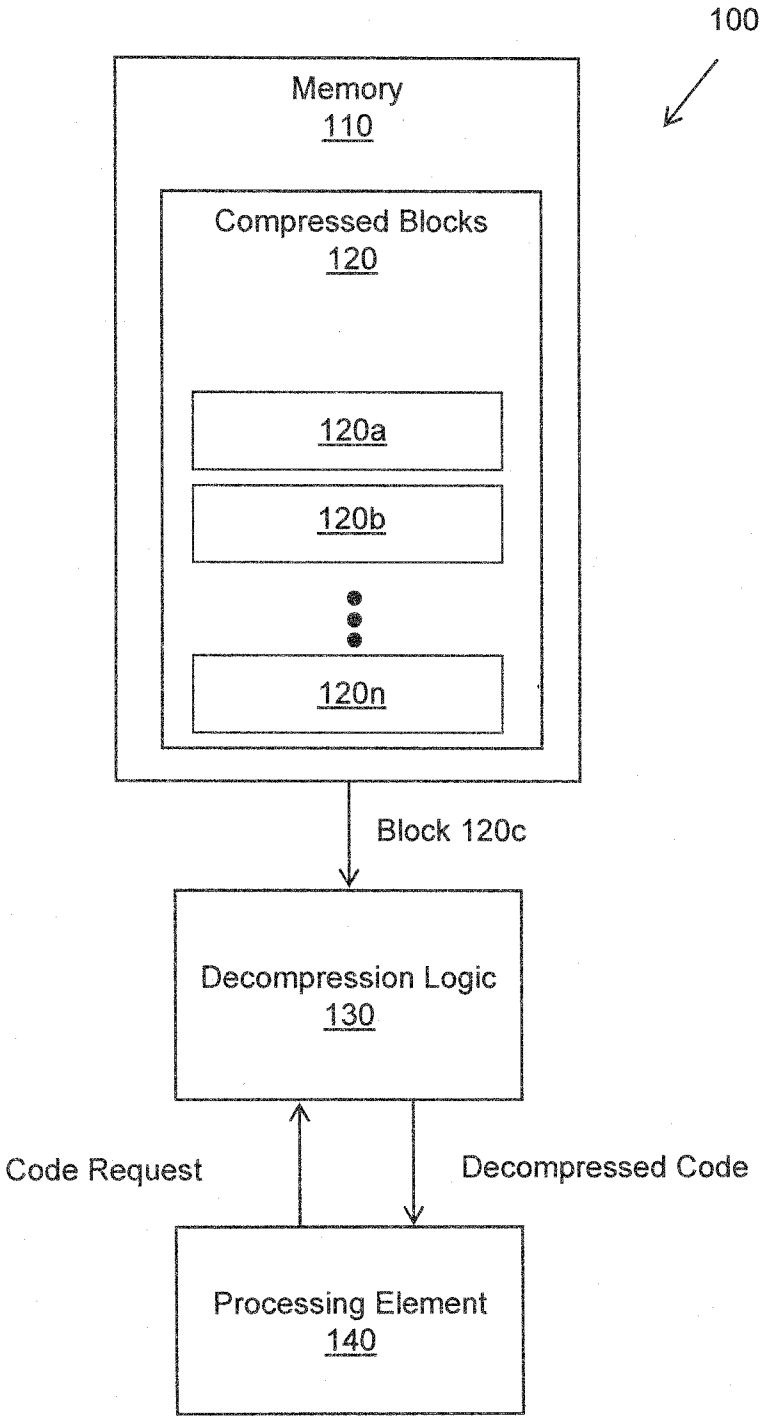
100
Memory
110
Compressed Blocks
120
120a
120b
120n
Block 120c
Decompression Logic
130
Code Request
Decompressed Code
Processing Element
140

Figure 1

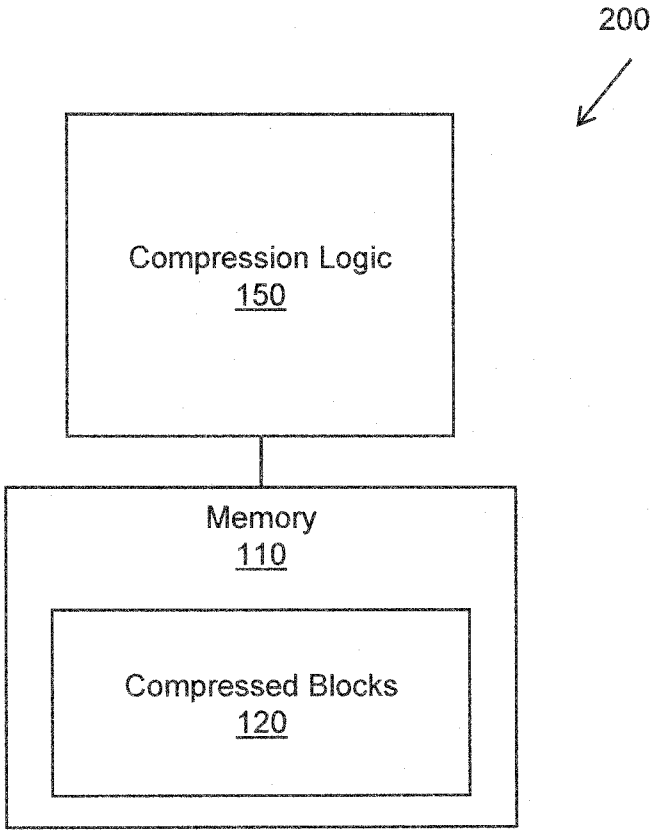
200
Compression Logic
150
Memory
110
Compressed Blocks
120

Figure 2A

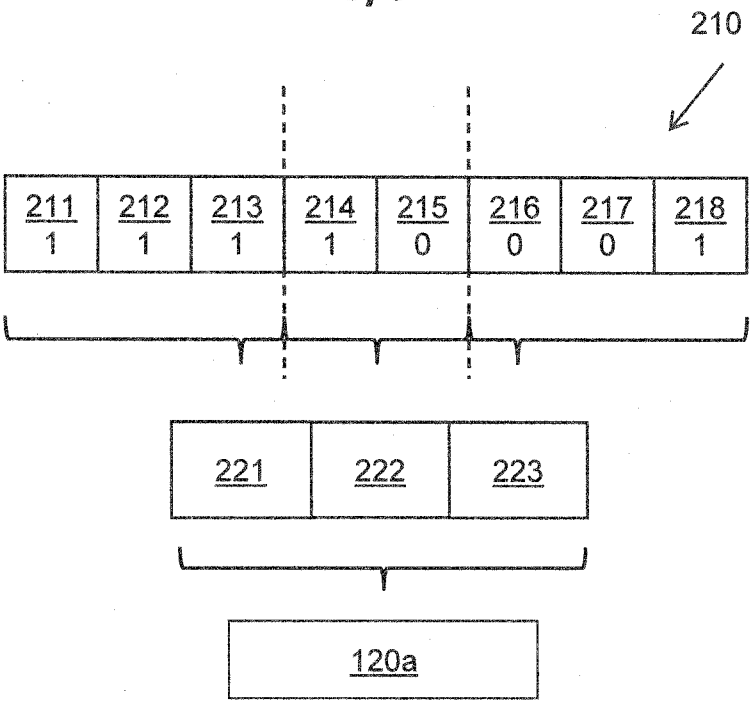

Figure 2B

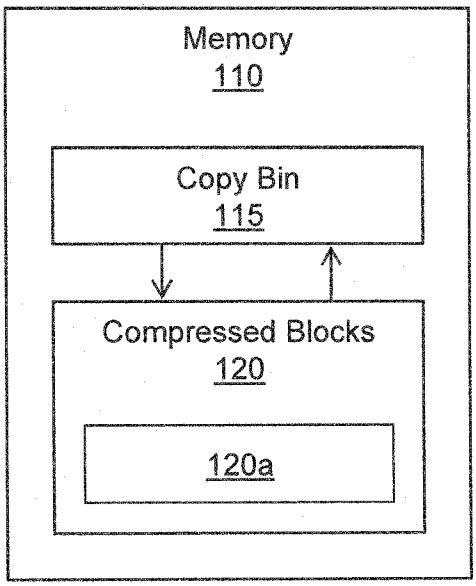

Figure 2C

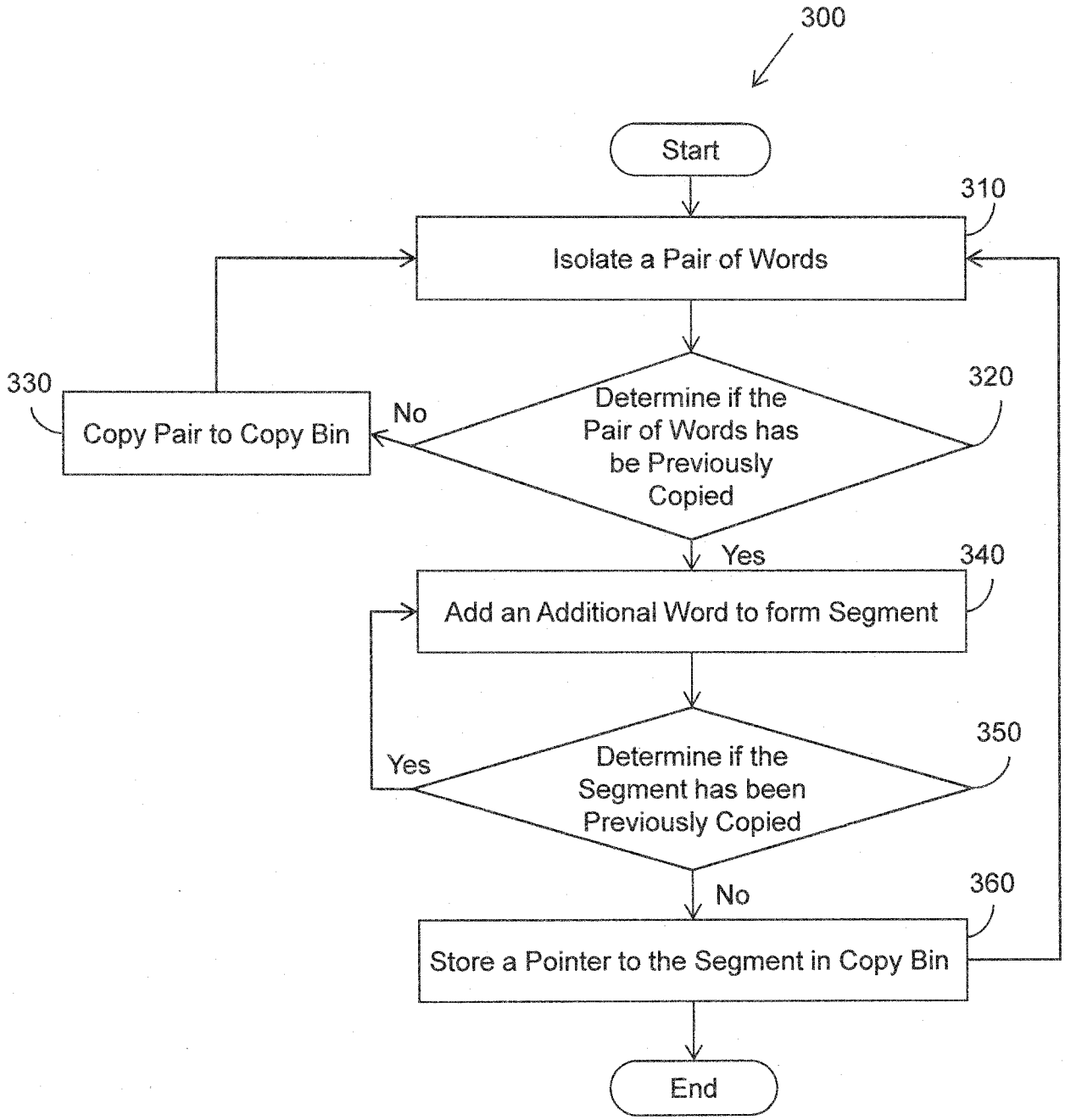
300
Start
310
Isolate a Pair of Words
330
Copy Pair to Copy Bin
No
Determine if the Pair of Words has be Previously Copied
320
Yes
340
Add an Additional Word to form Segment
Yes
Determine if the Segment has been Previously Copied
350
No
360
Store a Pointer to the Segment in Copy Bin
End

Figure 3

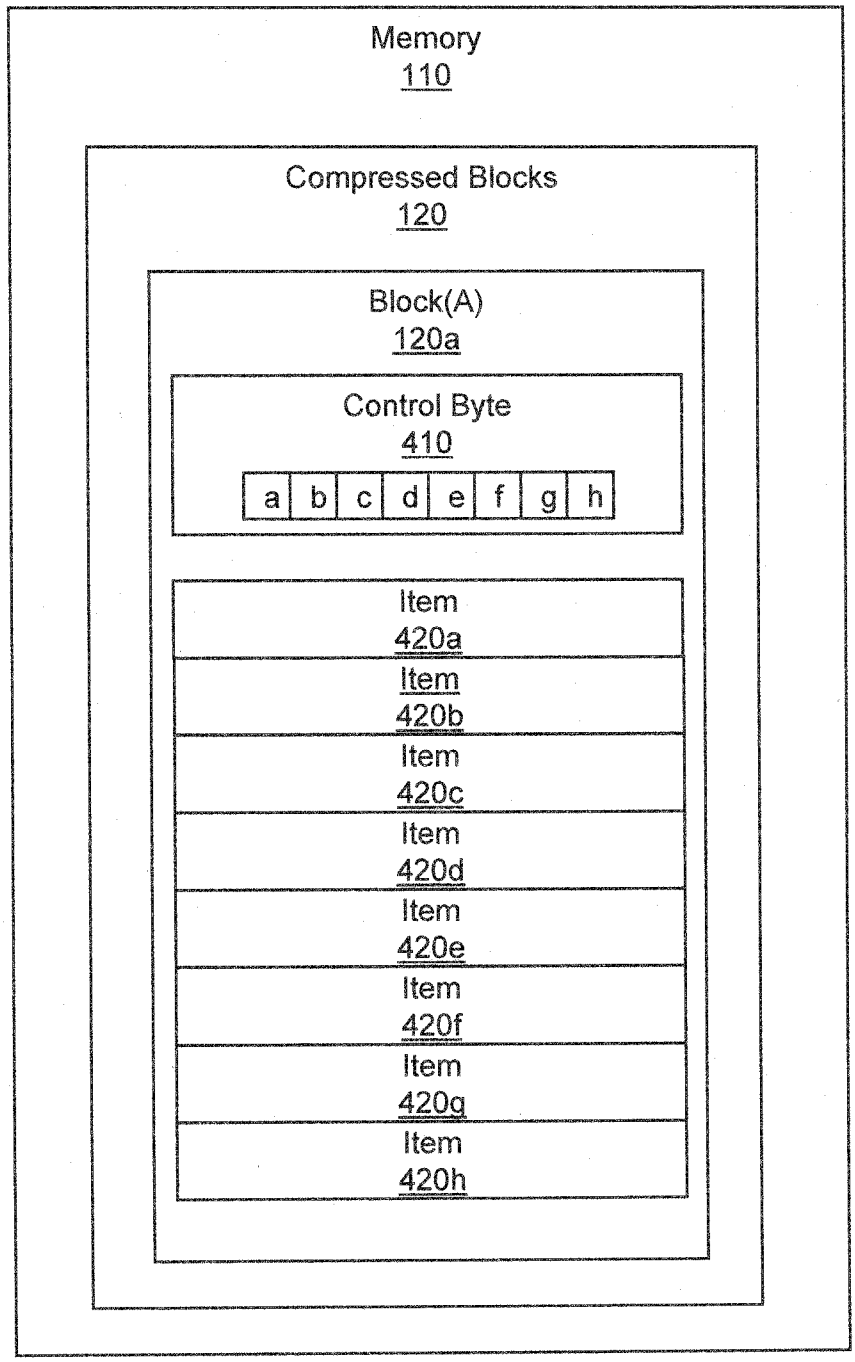
Memory
110
Compressed Blocks
120
Block(A)
120a
Control Byte
410
a b c d e f g h
Item
420a
Item
420b
Item
420c
Item
420d
Item
420e
Item
420f
Item
420g
Item
420h

Figure 4A

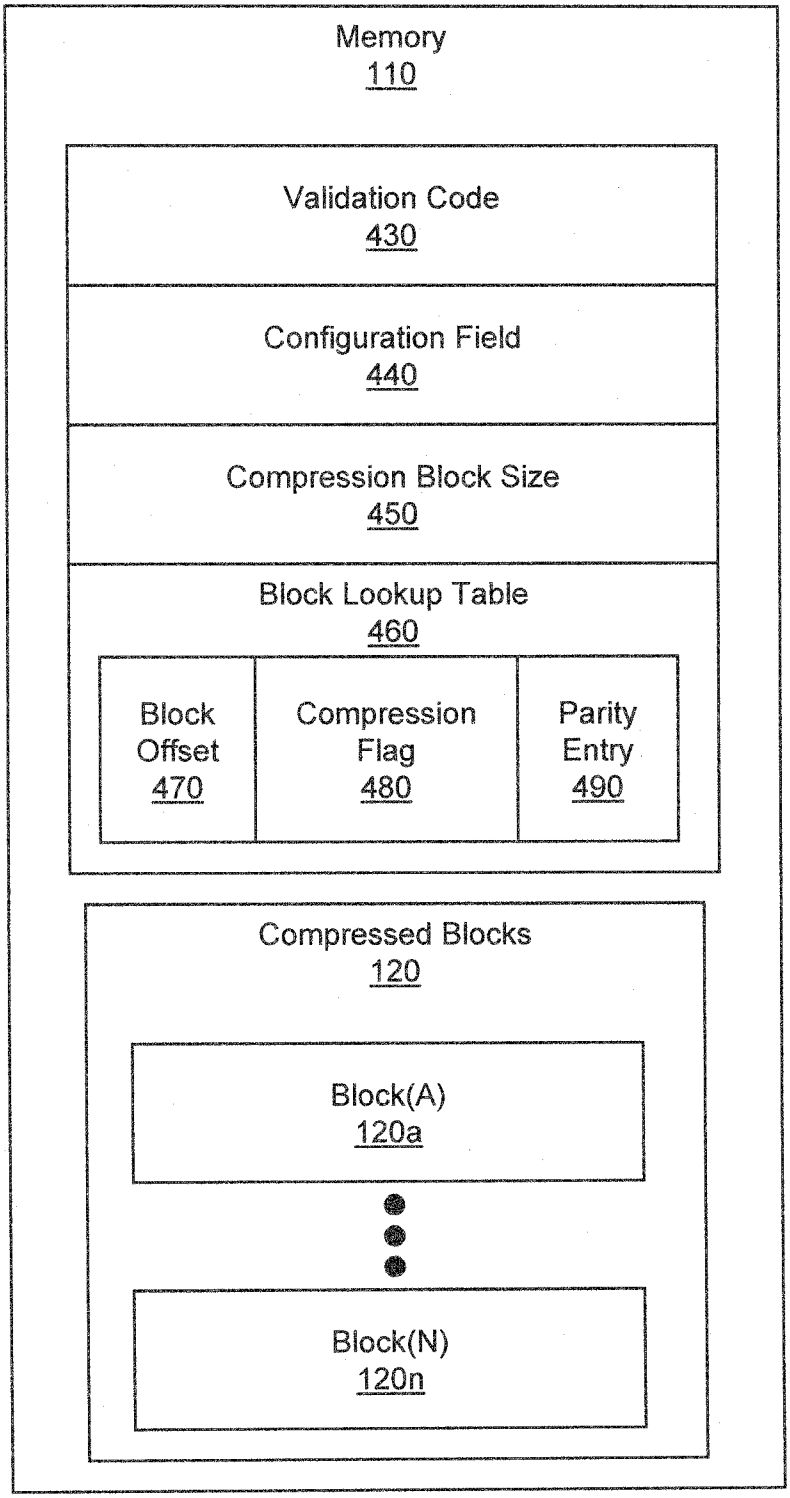
Memory
110
Validation Code
430
Configuration Field
440
Compression Block Size
450
Block Lookup Table
460
Block Offset
470
Compression Flag
480
Parity Entry
490
Compressed Blocks
120
Block(A)
120a
Block(N)
120n

Figure 4B

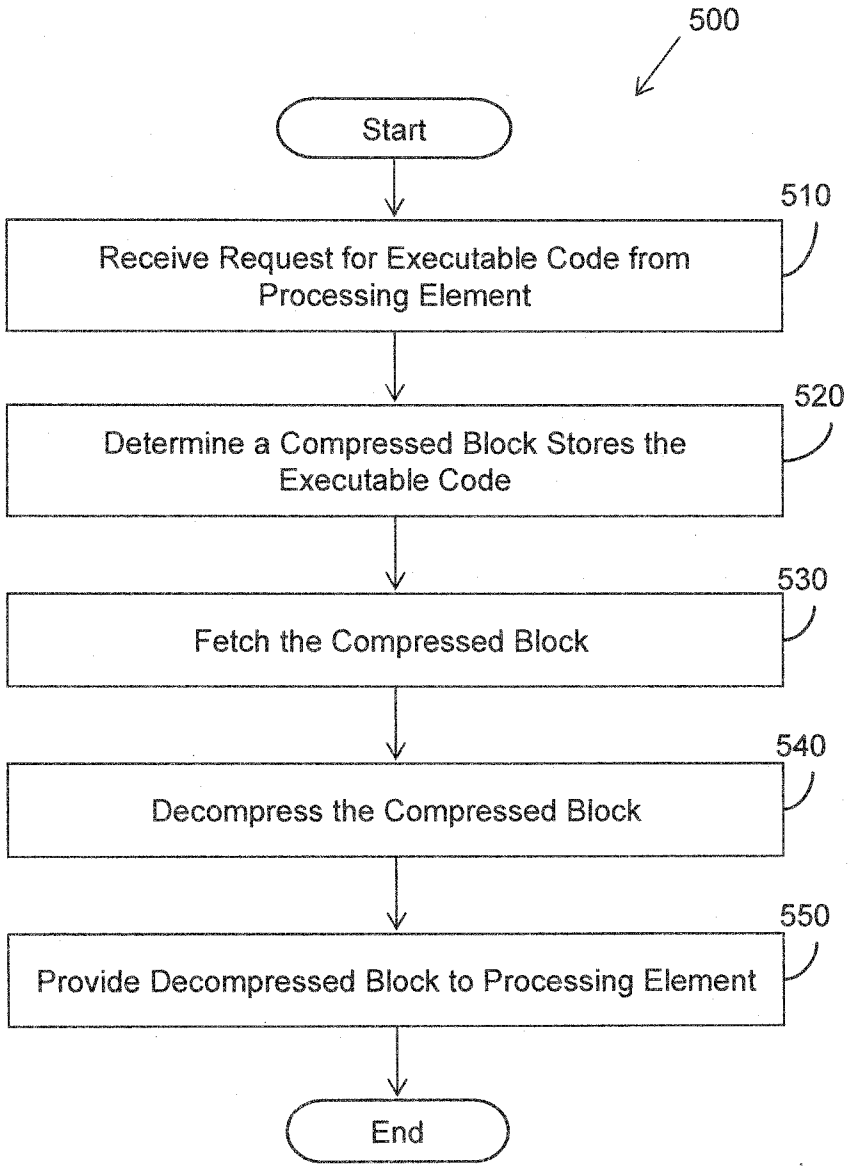
500
Start
510
Receive Request for Executable Code from Processing Element
520
Determine a Compressed Block Stores the Executable Code
530
Fetch the Compressed Block
540
Decompress the Compressed Block
550
Provide Decompressed Block to Processing Element
End

Figure 5

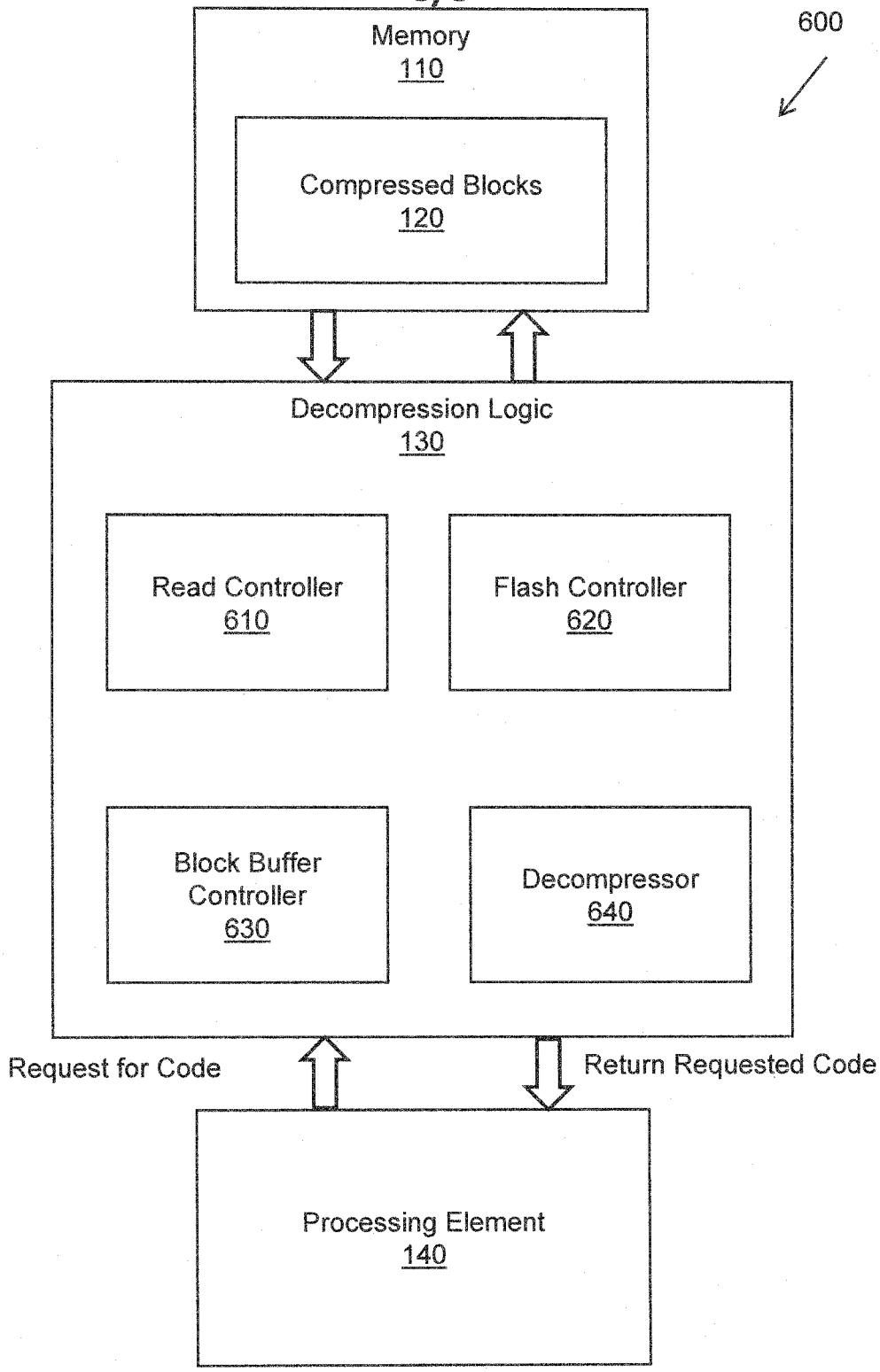
600
Memory
110
Compressed Blocks
120
Decompression Logic
130
Read Controller
610
Flash Controller
620
Block Buffer
Controller
630
Decompressor
640
Request for Code
Return Requested Code
Processing Element
140

Figure 6

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/030345

A. CLASSIFICATION OF SUBJECT MATTER
INV. H03M7/30 G06F9/445
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)
H03M 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

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y A	US 2007/083571 A1 (MELLER EVYATAR [IL] ET AL) 12 April 2007 (2007-04-12) paragraph [0262] - paragraph [0315]; figures 13-15	1-4,6-12 5 13-20

X	SALOMON D ED - SALOMON D: "Data Compression. The Complete Reference passage", 1 November 2009 (2009-11-01), DATA COMPRESSION. THE COMPLETE REFERENCE, SPRINGER VERLAG, DE, PAGE(S) 189 - 192, XP007920658, ISBN: 978-1-84882-902-2	13-20
Y	the whole document	5

X A	EP 2 017 726 A2 (SONY ERICSSON MOBILE COMM JP [JP]) 21 January 2009 (2009-01-21) the whole document	1,9 13

	-/--	

[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
12 June 2012	18/06/2012

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	Bijn, Koen

4

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

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/030345

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 7 669 195 B1 (QUMEI IYAD [US]) 23 February 2010 (2010-02-23)	1,9
A	column 12, line 51 - column 13, line 52; figure 3	13

X	DE 10 2007 045987 A1 (CONTINENTAL AUTOMOTIVE GMBH [DE]) 2 April 2009 (2009-04-02)	1,9
A	the whole document	13

X	US 2009/237698 A1 (HUANG YAO-HUI [TW]) 24 September 2009 (2009-09-24)	1,9
A	paragraph [0010] paragraph [0024] - paragraph [0027]	13

X	WO 2007/005143 A1 (ADVANCED MICRO DEVICES INC [US]; GOODRICH STEVEN [US]) 11 January 2007 (2007-01-11)	1,9
A	paragraph [0008] - paragraph [0010]	13

4

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2012/030345

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2007083571 A1	12-04-2007	CN 101326492 A	17-12-2008
		EP 1934729 A2	25-06-2008
		KR 20080067639 A	21-07-2008
		US 2007083571 A1	12-04-2007
		WO 2007039907 A2	12-04-2007
EP 2017726 A2	21-01-2009	CN 101334736 A	31-12-2008
		EP 2017726 A2	21-01-2009
		JP 2009009392 A	15-01-2009
		US 2009007090 A1	01-01-2009
US 7669195 B1	23-02-2010	US 7669195 B1	23-02-2010
		US 7886093 B1	08-02-2011
DE 102007045987 A1	02-04-2009	NONE	
US 2009237698 A1	24-09-2009	TW 200941221 A	01-10-2009
		US 2009237698 A1	24-09-2009
WO 2007005143 A1	11-01-2007	CN 101213517 A	02-07-2008
		DE 112006001743 T5	08-05-2008
		GB 2441729 A	12-03-2008
		JP 2009510544 A	12-03-2009
		KR 20080040685 A	08-05-2008
		US 2007016693 A1	18-01-2007
		WO 2007005143 A1	11-01-2007