



- (51) **International Patent Classification:**
G06F 9/38 (2006.01) *G06F 9/34* (2006.01)
- (21) **International Application Number:**
PCT/IB2016/053999
- (22) **International Filing Date:**
4 July 2016 (04.07.2016)
- (25) **Filing Language:** English
- (26) **Publication Language:** English
- (30) **Priority Data:**
- | | | |
|------------|--------------------------|----|
| 14/794,835 | 9 July 2015 (09.07.2015) | US |
| 14/794,837 | 9 July 2015 (09.07.2015) | US |
| 14/794,841 | 9 July 2015 (09.07.2015) | US |
| 14/794,853 | 9 July 2015 (09.07.2015) | US |
- (71) **Applicant:** CENTIPEDE SEMI LTD. [IL/IL]; 4 Haomanut Street, P.O. Box 8195, 4250438 Netanya (IL).
- (72) **Inventors:** MIZRAHI, Noam; 13A Havered Street, 45100 Hod Hasharon (IL). FRIEDMANN, Jonathan; 5 Ha'etrog Street, P. O. Box 5068, 4053260 Even Yehuda (IL).
- (74) **Agent:** D. KLIGLER I.P. SERVICES LTD.; P. O. Box 57651, 61576 Tel Aviv (IL).

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

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

Published:

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

(54) **Title:** PROCESSOR WITH EFFICIENT MEMORY ACCESS

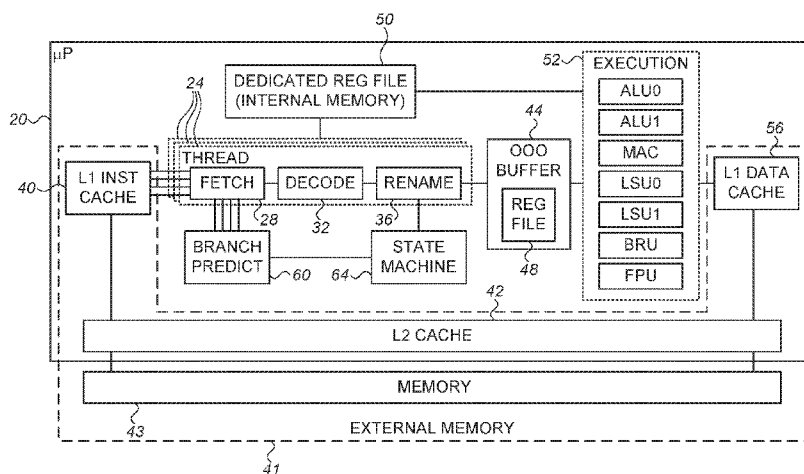


FIG. 1

(57) **Abstract:** A method includes, in a processor (20), processing program code that includes memory-access instructions, wherein at least some of the memory-access instructions include symbolic expressions that specify memory addresses in an external memory (41) in terms of one or more register names. A relationship between the memory addresses accessed by two or more of the memory-access instructions is identified, based on respective formats of the memory addresses specified in the symbolic expressions. An outcome of at least one of the memory-access instructions is assigned to be served from an internal memory (50) in the processor, based on the identified relationship.

PROCESSOR WITH EFFICIENT MEMORY ACCESS

FIELD OF THE INVENTION

The present invention relates generally to microprocessor design, and particularly to methods and systems for efficient memory access in microprocessors.

BACKGROUND OF THE INVENTION

One of the major bottlenecks that limit parallelization of code in microprocessors is dependency between memory-access instructions. Various techniques have been proposed to improve parallelization performance of code that includes memory access. For example, Tyson and Austin propose a technique referred to as “memory renaming,” in “Memory Renaming: Fast, Early and Accurate Processing of Memory Communication,” International Journal of Parallel Programming, Volume 27, No. 5, 1999, which is incorporated herein by reference. Memory renaming is a modification of the processor pipeline that applies register access techniques to load and store instructions to speed the processing of memory traffic. The approach works by predicting memory communication early in the pipeline, and then re-mapping the communication to fast physical registers.

SUMMARY OF THE INVENTION

An embodiment of the present invention that is described herein provides a method including, in a processor, processing program code that includes memory-access instructions, wherein at least some of the memory-access instructions include symbolic expressions that specify memory addresses in an external memory in terms of one or more register names. A relationship between the memory addresses accessed by two or more of the memory-access instructions is identified, based on respective formats of the memory addresses specified in the symbolic expressions. An outcome of at least one of the memory-access instructions is assigned to be served from an internal memory in the processor, based on the identified relationship.

In some embodiments, identifying the relationship between the memory addresses is independent of actual numerical values of the memory addresses. In an embodiment, identifying the relationship between the memory addresses is performed at a point in time at which the actual numerical values of the memory addresses are undefined. In a disclosed embodiment, identifying the relationship is performed by a given pipeline stage in a pipeline of the processor, and the actual numerical values of the memory addresses are calculated in another pipeline stage that is located later in the pipeline than the given pipeline stage.

In some embodiments, identifying the relationship includes searching the program code for memory-access instructions that specify the memory addresses using identical symbolic expressions. In an embodiment, identifying the relationship includes searching the program code for memory-access instructions that specify the memory addresses using different symbolic
5 expressions that refer to the same memory address. In another embodiment, assigning the outcome of at least one of the memory-access instructions is performed by a decoding unit or a renaming unit in a pipeline of the processor.

In some embodiments, assigning the outcome to be served from the internal memory further includes executing a memory-access instruction in the external memory, and verifying
10 that the outcome of the memory-access instruction executed in the external memory matches the outcome assigned to the memory-access instruction from the internal memory. In an embodiment, verifying the outcome includes comparing the outcome of the memory-access instruction executed in the external memory with the outcome assigned to the memory-access instruction from the internal memory. In another embodiment, verifying the outcome includes
15 verifying that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

In yet another embodiment, verifying the outcome includes adding to the program code one or more instructions or micro-ops that verify the outcome, or modifying one or more existing instructions or micro-ops to the instructions or micro-ops that verify the outcome. In another
20 embodiment, the method further includes flushing subsequent code upon finding that the outcome executed in the external memory does not match the outcome served from the internal memory.

In some embodiments, the method further includes inhibiting the at least one of the memory-access instructions from being executed in the external memory. In other embodiments,
25 the method further includes parallelizing execution of the program code, including assignment of the outcome from the internal memory, over multiple hardware threads. In another embodiment, processing the program code includes executing the program code, including assignment of the outcome from the internal memory, in a single hardware thread.

In a disclosed embodiment, identifying the relationship includes identifying the
30 memory-access instructions in a loop or a function. In another embodiment, identifying the relationship is performed at runtime. In an embodiment, identifying the relationship is performed, at least partly, based on indications embedded in the program code.

There is additionally provided, in accordance with an embodiment of the present invention, a processor including an internal memory and processing circuitry. The processing

circuitry is configured to process program code that includes memory-access instructions, wherein at least some of the memory-access instructions include symbolic expressions that specify memory addresses in an external memory in terms of one or more register names, to identify a relationship between the memory addresses accessed by two or more of the memory-access instructions, based on respective formats of the memory addresses specified in the symbolic expressions, and to assign an outcome of at least one of the memory-access instructions to be served from the internal memory, based on the identified relationship.

The present invention will be more fully understood from the following detailed description of the embodiments thereof, taken together with the drawings in which:

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram that schematically illustrates a processor, in accordance with an embodiment of the present invention;

Fig. 2 is a flow chart that schematically illustrates a method for processing code that contains memory-access instructions, in accordance with an embodiment of the present invention;

Fig. 3 is a flow chart that schematically illustrates a method for processing code that contains recurring load instructions, in accordance with an embodiment of the present invention;

Fig. 4 is a flow chart that schematically illustrates a method for processing code that contains load-store instruction pairs, in accordance with an embodiment of the present invention;

Fig. 5 is a flow chart that schematically illustrates a method for processing code that contains repetitive load-store instruction pairs with intervening data manipulation, in accordance with an embodiment of the present invention; and

Fig. 6 is a flow chart that schematically illustrates a method for processing code that contains recurring load instructions from nearby memory addresses, in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION OF EMBODIMENTS

OVERVIEW

Embodiments of the present invention that are described herein provide improved methods and systems for processing software code that includes memory-access instructions. In the disclosed techniques, a processor monitors the code instructions, and finds relationships between memory-access instructions. Relationships may comprise, for example, multiple load

instructions that access the same memory address, load and store instruction pairs that access the same memory address, or multiple load instructions that access a predictable pattern of memory addresses.

Based on the identified relationships, the processor is able to serve the outcomes of some memory-access instructions, to subsequent code that depends on the outcomes, from internal memory (e.g., internal registers, local buffer) instead of from external memory. In the present context, reading from the external memory via a cache, which is possibly internal to the processor, is also regarded as serving an instruction from the external memory.

In an example embodiment, when multiple load instructions read from the same memory address, the processor reads a value from this memory address on the first load instruction, and saves the value to an internal register. When processing the next load instructions, the processor serves the value to subsequent code from the internal register, without waiting for the load instruction to retrieve the value from the memory address. As a result, subsequent code that depends on the outcomes of the load instructions can be executed sooner, dependencies between instructions can be relaxed, and parallelization can be improved.

Typically, the next load instructions are still carried out in the external memory, e.g., in order to verify that the value served from the internal memory is still valid, but execution progress does not have to wait for them to complete. This feature improves performance since the dependencies of subsequent code on the load instructions are broken, and instruction parallelization can be improved.

In order to identify the relationships, it is possible in principle to wait until the numerical values of the memory addresses accessed by the memory-access instructions have been decoded, and then identify relationships between numerical values of decoded memory addresses. This solution, however, is costly in terms of latency because the actual numerical addresses accessed by the memory-access instructions are known only late in the pipeline.

Instead, in the embodiments described herein, the processor identifies the relationships between memory-access instructions based on the formats of the symbolic expressions that specify the memory addresses in the instructions, and not based on the actual numerical values of the addresses. The symbolic expressions are available early in the pipeline, as soon as the instructions are decoded. As a result, the disclosed techniques identify and act upon interrelated memory-access instructions with small latency, thus enabling fast operation and a high degree of parallelization.

Several examples of relationships between memory-access instructions, which can be identified and exploited, are described herein. Several schemes for handling the additional

internal registers are also described, e.g., schemes that add micro-ops to the code and schemes that modify the conventional renaming of registers.

The disclosed techniques provide considerable performance improvements and are suitable for implementation in a wide variety of processor architectures, including both multi-thread and single-thread architectures.

SYSTEM DESCRIPTION

Fig. 1 is a block diagram that schematically illustrates a processor 20, in accordance with an embodiment of the present invention. Processor 20 runs pre-compiled software code, while parallelizing the code execution. Instruction parallelization is performed by the processor at run-time, by analyzing the program instructions as they are fetched from memory and processed.

In the present example, processor 20 comprises multiple hardware threads 24 that are configured to operate in parallel. Each thread 24 is configured to process a respective segment of the code. Certain aspects of thread parallelization, including definitions and examples of partially repetitive segments, are addressed, for example, in U.S. Patent Applications 14/578,516, 14/578,518, 14/583,119, 14/637,418, 14/673,884, 14/673,889 and 14/690,424, which are all assigned to the assignee of the present patent application and whose disclosures are incorporated herein by reference.

In the present embodiment, each thread 24 comprises a fetching unit 28, a decoding unit 32 and a renaming unit 36. Although some of the examples given below refer to instruction parallelization and to multi-thread architectures, the disclosed techniques are applicable and provide considerable performance improvements in single-thread processors, as well.

Fetching unit 24 fetch the program instructions of their respective code segments from a memory, e.g., from a multi-level instruction cache. In the present example, the multi-level instruction cache comprises a Level-1 (L1) instruction cache 40 and a Level-2 (L2) cache 42 that cache instructions stored in a memory 43. Decoding units 32 decode the fetched instructions (and possibly transform them into micro-ops), and renaming units 36 carry out register renaming.

The decoded instructions following renaming are buffered in an Out-of-Order (OOO) buffer 44 for out-of-order execution by multiple execution units 52, i.e., not in the order in which they have been compiled and stored in memory. The renaming units assign names (physical registers) to the operands and destination registers such that the OOO buffer issues (send for execution) instructions correctly based on availability of their operands. Alternatively, the buffered instructions may be executed in-order.

OOO buffer 44 comprises a register file 48. In some embodiments the processor further comprises a dedicated register file 50, also referred to herein as an internal memory. Register file 50 comprises one or more dedicated registers that are used for expediting memory-access instructions, as will be explained in detail below.

The instructions buffered in OOO buffer 44 are scheduled for execution by the various execution units 52. Instruction parallelization is typically achieved by issuing multiple (possibly out of order) instructions/micro-ops to the various execution units at the same time. In the present example, execution units 52 comprise two Arithmetic Logic Units (ALU) denoted ALU0 and ALU1, a Multiply-Accumulate (MAC) unit, two Load-Store Units (LSU) denoted LSU0 and LSU1, a Branch execution Unit (BRU) and a Floating-Point Unit (FPU). In alternative embodiments, execution units 52 may comprise any other suitable types of execution units, and/or any other suitable number of execution units of each type. The cascaded structure of threads 24, OOO buffer 44 and execution units 52 is referred to herein as the pipeline of processor 20.

The results produced by execution units 52 are saved in register file 48 and/or register file 50, and/or stored in memory 43. In some embodiments a multi-level data cache mediates between execution units 52 and memory 43. In the present example, the multi-level data cache comprises a Level-1 (L1) data cache 56 and L2 cache 42.

In some embodiments, the Load-Store Units (LSU) of processor 20 store data in memory 43 when executing store instructions, and retrieve data from memory 43 when executing load instructions. The data storage and/or retrieval operations may use the data cache (e.g., L1 cache 56 and L2 cache 42) for reducing memory access latency. In some embodiments, high-level cache (e.g., L2 cache) may be implemented, for example, as separate memory areas in the same physical memory, or simply share the same memory without fixed pre-allocation.

In the present context, memory 43, L1 cache 40 and 56, and L2 cache 42 are referred to collectively as an external memory 41. Any access to memory 43, cache 40, cache 56 or cache 42 is regarded as an access to the external memory. References to “addresses in the external memory” or “addresses in external memory 41” refer to the addresses of data in memory 43, even though the data may be physically retrieved by reading cached copies of the data in cache 56 or 42. By contrast, access to register file 50, for example, is regarded as access to internal memory.

A branch prediction unit 60 predicts branches or flow-control traces (multiple branches in a single prediction), referred to herein as “traces” for brevity, that are expected to be traversed by the program code during execution. The code may be executed in a single-thread processor

or a single thread within a multi-thread processor, or by the various threads 24 as described in U.S. Patent Applications 14/578,516, 14/578,518, 14/583,119, 14/637,418, 14/673,884, 14/673,889 and 14/690,424, cited above.

Based on the predictions, branch prediction unit 60 instructs fetching units 28 which new instructions are to be fetched from memory. Branch prediction in this context may predict entire traces for segments or for portions of segments, or predict the outcome of individual branch instructions. When parallelizing the code, e.g., as described in the above-cited patent applications, a state machine unit 64 manages the states of the various threads 24, and invokes threads to execute segments of code as appropriate.

In some embodiments, processor 20 parallelizes the processing of program code among threads 24. Among the various parallelization tasks, processor 20 performs efficient processing of memory-access instructions using methods that are described in detail below. Parallelization tasks are typically performed by various units of the processor. For example, branch prediction unit 60 typically predicts the control-flow traces for the various threads, state machine unit 64 invokes threads to execute appropriate segments at least partially in parallel, and renaming units 36 handle memory-access parallelization. In alternative embodiments, memory parallelization unit may be performed by decoding units 32, and/or jointly by decoding units 32 and renaming units 36.

Thus, in the context of the present disclosure and in the claims, units 60, 64, 32 and 36 are referred to collectively as thread parallelization circuitry (or simply parallelization circuitry for brevity). In alternative embodiments, the parallelization circuitry may comprise any other suitable subset of the units in processor 20. In some embodiments, some or even all of the functionality of the parallelization circuitry may be carried out using run-time software. Such run-time software is typically separate from the software code that is executed by the processor and may run, for example, on a separate processing core.

In the present context, register file 50 is referred to as internal memory, and the terms “internal memory” and “internal register” are sometimes used interchangeably. The remaining processor elements are referred to herein collectively as processing circuitry that carries out the disclosed techniques using the internal memory. Generally, other suitable types of internal memory can also be used for carrying out the disclosed techniques.

As noted already, although some of the examples described herein refer to multiple hardware threads and thread parallelization, many of the disclosed techniques can be implemented in a similar manner with a single hardware thread. The processor pipeline may comprise, for example, a single fetching unit 28, a single decoding unit 32, a single renaming

unit 36, and no state machine 64. In such embodiments, the disclosed techniques accelerate memory access in single-thread processing. As such, although the examples below refer to memory-access acceleration functions being performed by the parallelization circuitry, these functions may generally be carried out by the processing circuitry of the processor.

5 The configuration of processor 20 shown in Fig. 1 is an example configuration that is chosen purely for the sake of conceptual clarity. In alternative embodiments, any other suitable processor configuration can be used. For example, in the configuration of Fig. 1, multi-threading is implemented using multiple fetching, decoding and renaming units. Additionally or alternatively, multi-threading may be implemented in many other ways, such as using multiple
10 OOO buffers, separate execution units per thread and/or separate register files per thread. In another embodiment, different threads may comprise different respective processing cores.

As yet another example, the processor may be implemented without cache or with a different cache structure, without branch prediction or with a separate branch prediction per thread. The processor may comprise additional elements not shown in the figure. Further
15 alternatively, the disclosed techniques can be carried out with processors having any other suitable micro-architecture.

Moreover, although the embodiments described herein refer mainly to parallelization of repetitive code, the disclosed techniques can be used to improve the processor performance, e.g., replace (and reduce) memory access time with register access time, reduce the number of
20 external memory access operations, regardless of thread parallelization. Such techniques can be applied in single-thread configurations or other configurations that do not necessarily involve thread parallelization.

Processor 20 can be implemented using any suitable hardware, such as using one or more Application-Specific Integrated Circuits (ASICs), Field-Programmable Gate Arrays (FPGAs)
25 or other device types. Additionally or alternatively, certain elements of processor 20 can be implemented using software, or using a combination of hardware and software elements. The instruction and data cache memories can be implemented using any suitable type of memory, such as Random Access Memory (RAM).

Processor 20 may be programmed in software to carry out the functions described herein.
30 The software may be downloaded to the processor in electronic form, over a network, for example, or it may, alternatively or additionally, be provided and/or stored on non-transitory tangible media, such as magnetic, optical, or electronic memory.

In some embodiments, the parallelization circuitry of processor 20 monitors the code processed by one or more threads 24, identifies code segments that are at least partially

repetitive, and parallelizes execution of these code segments. Certain aspects of parallelization functions performed by the parallelization circuitry, including definitions and examples of partially repetitive segments, are addressed, for example, in U.S. Patent Applications 14/578,516, 14/578,518, 14/583,119, 14/637,418, 14/673,884, 14/673,889 and 14/690,424, cited above.

EARLY DETECTION OF RELATIONSHIPS BETWEEN MEMORY-ACCESS INSTRUCTIONS BASED ON INSTRUCTION FORMAT

Typically, the program code that is processed by processor 20 contains memory-access instructions such as load and store instructions. In many cases, different memory-access instructions in the code are inter-related, and these relationships can be exploited for improving performance. For example, different memory-access instructions may access the same memory address, or a predictable pattern of memory addresses. As another example, one memory-access instruction may read or write a certain value, subsequent instructions may manipulate that value in a predictable way, and a later memory-access instruction may then write the manipulated value to memory.

In some embodiments, the parallelization circuitry in processor 20 identifies such relationships between memory-access instructions, and uses the relationships to improve parallelization performance. In particular, the parallelization circuitry identifies the relationships by analyzing the formats of the symbolic expressions that specify the addresses accessed by the memory-access instructions (as opposed to the numerical values of the addresses).

Typically, the operand of a memory-access instruction (e.g., load or store instruction) comprises a symbolic expression, i.e., an expression defined in terms of one or more register names, specifying the memory-access operation to be performed. The symbolic expression of a memory-access instruction may specify, for example, the memory address to be accessed, a register whose value is to be written, or a register into which a value is to be read.

Depending on the instruction set defined in processor 20, the symbolic expressions may have a wide variety of formats. Different symbolic formats may relate to different addressing modes (e.g., direct vs. indirect addressing), or to pre-incrementing or post-incrementing of indices, to name just a few examples.

In a typical flow, decoding units 32 decode the instructions, including the symbolic expressions. At this stage, however, the actual numerical values of the expressions (e.g., numerical memory addresses to be accessed and/or numerical values to be written) are not yet known and possibly undefined. The symbolic expressions are typically evaluated later, by

renaming units 36, just before the instructions are written to OOO buffer 44. Only at the execution stage, the LSUs and/or ALUs evaluate the symbolic expressions and assign the memory-access instructions actual numerical values.

In one example embodiment, the numerical memory addresses to be accessed is evaluated in the LSU and the numerical values to be written are evaluated in the ALU. In another example embodiment, both the numerical memory addresses to be accessed, and the numerical values to be written, are evaluated in the LSU.

It should be noted that the time delay between decoding an instruction (making the symbolic expression available) and evaluating the numerical values in the symbolic expression is not only due to the pipeline delay. In many practical scenarios, a symbolic expression of a given memory-access instruction cannot be evaluated (assigned numerical values) until the outcome of a previous instruction is available. Because of such dependencies, the symbolic expression may be available, in symbolic form, long before (possibly several tens of cycles before) it can be evaluated.

In some embodiments, the parallelization circuitry identifies and exploits the relationships between memory-access instructions by analyzing the formats of the symbolic expressions. As explained above, the relationships may be identified and exploited at a point in time at which the actual numerical values are still undefined and cannot be evaluated (e.g., because they depend on other instructions that were not yet executed). Since this process does not wait for the actual numerical values to be assigned, it can be performed early in the pipeline. As a result, subsequent code that depends on the outcomes of the memory-access instructions can be executed sooner, dependencies between instructions can be relaxed, and parallelization can thus be improved.

In some embodiments, the disclosed techniques are applied in regions of the code containing one or more code segments that are at least partially repetitive, e.g., loops or functions. Generally, however, the disclosed techniques can be applied in any other suitable region of the code, e.g., sections of loop iterations, sequential code and/or any other suitable instruction sequence, with a single or multi-threaded processor.

Fig. 2 is a flow chart that schematically illustrates a method for processing code that contains memory-access instructions, in accordance with an embodiment of the present invention. The method begins with the parallelization circuitry in processor 20 monitoring code instructions, at a monitoring step 70. The parallelization circuitry analyzes the formats of the symbolic expressions of the monitored memory-access instructions, at a symbolic analysis step

74. In particular, the parallelization circuitry analyzes the parts of the symbolic expressions that specify the addresses to be accessed.

Based on the analyzed symbolic expressions, the parallelization circuitry identifies relationships between different memory-access instructions, at a relationship identification step

78. Based on the identified relationships, at a serving step 82, the parallelization circuitry serves the outcomes of at least some of the memory-access instructions from internal memory (e.g., internal registers of processor 20) instead of from external memory 41.

As noted above, the term “serving a memory-access instruction from external memory 41” covers the cases of serving a value that is stored in memory 43, or cached in cache 56 or 42.

The term “serving a memory-access instruction from internal memory” refers to serving the value either directly or indirectly. One example of serving the value indirectly is copying the value to an internal register, and then serving the value from that internal register. Serving from the internal memory may be assigned, for example, by decoding unit 32 or renaming unit 36 of the relevant thread 24 and later performed by one of execution units 52.

The description that follows depicts several example relationships between memory-access instructions, and demonstrates how processor 20 accelerates memory access by identifying and exploiting these relationships. The code examples below are given using the ARM® instructions set, purely by way of example. In alternative embodiments, the disclosed techniques can be carried out using any other suitable instruction set.

EXAMPLE RELATIONSHIP: LOAD INSTRUCTIONS ACCESSING THE SAME MEMORY ADDRESS

In some embodiments, the parallelization circuitry identifies multiple load instructions (e.g., *ldr* instructions) that read from the same memory address in the external memory. The identification typically also includes verifying that no store instruction writes to this same memory address between the load instructions.

One example of such a scenario is a load instruction of the form

ldr r1,[r6]

that is found inside a loop, wherein r6 is a global register. In the present context, the term “global register” refers to a register that is not written to between the various loads within the loop iterations (i.e., the register value does not change between loop iterations). The instruction above loads from memory the value which resides in the address which is held in r6 and puts it in r1.

In this embodiment, the parallelization circuitry analyzes the format of the symbolic expression of the address “[r6]”, identifies that r6 is global, recognizes that the symbolic expression is defined in terms of one or more global registers, and concludes that the load instructions in the various loop iterations all read from the same address in the external memory.

5 The multiple load instructions that read from the same memory address need not necessarily occur within a loop. Consider, for example, the following code:

ldr r1,[r5,r2]

inst

inst

10 *inst*

ldr r3,[r5,r2]

inst

inst

ldr r3,[r5,r2]

15 In the example above, all three load instructions access the same memory address, assuming registers r5 and r2 are not written to between the load instructions. Note that, as in the above example, the destination registers of the various load instructions are not necessarily the same.

In the examples above, all the identified load instructions specify the address using the same symbolic expression. In alternative embodiments, the parallelization circuitry identifies load instructions that read from the same memory address, even though different load instructions may specify the memory address using different symbolic expressions. For example, the load instructions

ldr r1,[r6,#4]!

25 *ldr r1,[r6]*

ldr r4,[r6]

all access the same memory address (in the first load the register r6 is first updated by adding 4 to its value). Another example for accessing the same memory address is repetitive load instructions such as:

ldr r1,[r6,#4]

or

ldr r1,[r6,r4] (where r4 is also unchanged)

or

ldr r1,[r6,r4 lsl #2]

The parallelization circuitry may recognize that these symbolic expressions all refer to the same address in various ways, e.g., by holding a predefined list of equivalent formats of symbolic expressions that specify the same address.

5 Upon identifying such a relationship, the parallelization circuitry saves the value read from the external memory by one of the load instructions in an internal register, e.g., in one of the dedicated registers in register file 50. For example, the processor parallelization circuitry may save the value read by the load instruction in the first loop iteration. When executing a subsequent load instruction, the parallelization circuitry may serve the outcome of the load
10 instruction from the internal memory, without waiting for the value to be retrieved from the external memory. The value may be served from the internal memory to any subsequent code instructions that depend on this value.

15 In alternative embodiments, the parallelization circuitry may identify recurring load instructions not only in loops, but also in functions, in sections of loop iterations, in sequential code, and/or in any other suitable instruction sequence.

In various embodiments, processor 20 may implement the above mechanism in various ways. In one embodiment, the parallelization circuitry (typically decoding unit 32 or renaming unit 36 of the relevant thread) implements this mechanism by adding instructions or micro-ops to the code.

20 Consider, for example, a loop that contains (among other instructions) the three instructions

ldr r1,[r6]
add r7,r6,r1
mov r1,r8

25 wherein r6 is a global register in this loop. The first instruction in this example loads a value from memory into r1, and the second instruction sums the value of r6 and r1 and puts it into r7. Note that the second instruction depends on the first. Further note that the value which was loaded from memory is “lost” in the third instruction which assigns the value of r8 to r1, and thus, there is a need to reload it from memory in each iteration. In an embodiment, upon
30 identifying the relationship between the recurring *ldr* instructions, the parallelization circuitry adds an instruction of the form

mov MSG,r1

after the *ldr* instruction in the first loop iteration, wherein MSG denotes a dedicated internal register. This instruction assigns the value which was loaded from memory in an additional register. The first loop iteration thus becomes

```

5      ldr r1,[r6]
      mov MSG,r1
      add r7,r6,r1
      mov r1,r8

```

As a result, when executing the first loop iteration, the address specified by “[r6]” will be read from external memory and the read value will be saved in register MSG.

10 In the subsequent loop iterations, the parallelization circuitry adds an instruction of the form

```

      mov r1,MSG

```

which assigns the value that was saved in the additional register to r1 after the *ldr* instruction. The subsequent loop iterations thus become

```

15      ldr r1,[r6]
      mov r1,MSG
      add r7,r6,r1
      mov r8,r1

```

As a result, when executing the subsequent loop iterations, value of register MSG will be loaded into register r1 without having to wait for the *ldr* instruction to retrieve the value from external memory 41.

Since the *mov* instruction is an ALU instruction and does not involve accessing the external memory, it is considerably faster than the *ldr* instruction (typically a single cycle instead of four cycles). Furthermore, the *add* instruction no longer depends on the *ldr* instruction but only on the *mov* instruction and thus, the subsequent code benefits from the reduction in processing time.

In an alternative embodiment, the parallelization circuitry implements the above mechanism without adding instructions or micro-ops to the code, but rather by configuring the way registers are renamed in renaming units 36. Consider the example above, or a loop containing (among other instructions) the three instructions

```

      ldr r1,[r6]
      add r7,r6,r1

```

mov r1,r8

When processing the *ldr* instruction in the first loop iteration, renaming unit 36 performs conventional renaming, i.e., renames destination register r1 to some physical register (denoted p8 in this example), and serves the operand r1 in the *add* instruction from p8. When processing the *mov* instruction, r1 is renamed to a new physical register (e.g., p9). Unlike conventional renaming, p8 is not released when p9 is committed. The processor thus maintains the value of register p8 that holds the value loaded from memory.

When executing the subsequent loop iterations, on the other hand, renaming unit 36 applies a different renaming scheme. The operands r1 in the *add* instructions of all subsequent loop iterations all read the value from the same physical register p8, eliminating the need to wait for the result of the load instruction. Register p8 is released only after the last loop iteration.

Further alternatively, the parallelization circuitry may serve the read value from the internal register in any other suitable way. Typically, the internal register is dedicated for this purpose only. For example, the internal register may comprise one of the processor's architectural registers in register file 48 which is not exposed to the user. Alternatively, the internal register may comprise a register in register file 50, which is not one of the processor's architectural registers in register file 48 (like r6) or physical registers (like p8). Alternatively to saving the value in an internal register of the processor, any other suitable internal memory of the processor can be used for this purpose.

Serving the outcome of a *ldr* instruction from an internal register (e.g., MSG or p8), instead of from the actual content of the external memory address, involves a small but non-negligible probability of error. For example, if a different value were to be written to the memory address in question at any time after the first load instruction, then the actual read value will be different from the value saved in the internal register. As another example, if the value of register r6 were to be changed (even though it is assumed to be global), then the next load instruction will read from a different memory address. In this case, too, the actual read value will be different from the value saved in the internal register.

Thus, in some embodiments the parallelization circuitry verifies, after serving an outcome of a load instruction from an internal register, that the served value indeed matches the actual value retrieved by the load instruction from external memory 41. If a mismatch is found, the parallelization circuitry may flush subsequent instructions and results. Flushing typically comprises discarding all subsequent instructions from the pipeline such that all processing that was performed with a wrong operand value is discarded. In other words, the processor executes

the subsequent load instructions in the external memory and retrieves the value from the memory address in question, for the purpose of verification, even though the value is served from the internal register.

The above verification may be performed, for example, by verifying that no store (e.g., *str*) instruction writes to the memory address between the recurring load instructions. Additionally or alternatively, the verification may ascertain that no fence instructions limit the possibility of serving subsequent code from the internal memory.

In some cases, however, the memory address in question may be written to by another entity, e.g., by another processor or processor core, or by a debugger. In such cases it may not be sufficient to verify that the monitored program code does not contain an intervening store instruction that writes to the memory address. In an embodiment, the verification may use an indication from a memory management subsystem, indicative of whether the content of the memory address was modified.

In the present context, intervening store instructions, intervening fence instructions, and/or indications from a memory management subsystems, are all regarded as intervening events that create a mismatch between the value in the external memory and the value served from the internal memory. The verification process may consider any of these events, and/or any other suitable intervening event.

In yet other embodiments, the parallelization circuitry may initially assume that no intervening event affects the memory address in question. If, during execution, some verification mechanism fails, the parallelization circuitry may deduce that an intervening event possibly exists, and refrain from serving the outcome from the internal memory.

As another example, the parallelization circuitry (typically decoding unit 32 or renaming unit 36) may add to the code an instruction or micro-op that retrieves the correct value from the external memory and compares it with the value of the internal register. The actual comparison may be performed, for example, by one of the ALUs or LSUs in execution units 52. Note that no instruction depends on the added micro-op, as it does not exist in the original code and is used only for verification. Further alternatively, the parallelization circuitry may perform the verification in any other suitable way. Note that this verification does not affect the performance benefit gained by the fast loading to register *r1* when it is correct, but rather flushes this fast loading in cases where it was wrong.

Fig. 3 is a flow chart that schematically illustrates a method for processing code that contains recurring load instructions, in accordance with an embodiment of the present invention. The method begins with the parallelization circuitry of processor 20 identifying a recurring

plurality of load instructions that access the same memory address (with no intervening event), at a recurring load identification step 90.

As explained above, this identification is made based on the formats of the symbolic expressions of the load instructions, and not based on the numerical values of the memory addresses. The identification may also consider and make use of factors such as the Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the load instructions in the program code.

At a load execution step 94, processor 20 dispatches the next load instruction for execution in external memory 41. The parallelization circuitry checks whether the load instruction just executed is the first occurrence in the recurring load instructions, at a first occurrence checking step 98.

On the first occurrence, the parallelization circuitry saves the value read from the external memory in an internal register, at a saving step 102. The parallelization circuitry serves this value to subsequent code, at a serving step 106. The parallelization circuitry then proceeds to the next occurrence in the recurring load instructions, at an iteration incrementing step 110. The method then loops back to step 94, for executing the next load instruction. (Other instructions in the code are omitted from this flow for the sake of clarity.)

On subsequent occurrences of load instruction from the same address, the parallelization circuitry serves the outcome of the load instruction (or rather assigns the outcome to be served) from the internal register, at an internal serving step 114. Note that although step 114 appears after step 94 in the flow chart, the actual execution which relates to step 114 ends before the execution which is related to step 94.

At a verification step 118, the parallelization circuitry verifies whether the served value (the value saved in the internal register at step 102) is equal to the value retrieved from the external memory (retrieved at step 94 of the present iteration). If so, the method proceeds to step 110. If a mismatch is found, the parallelization circuitry flushes the subsequent instructions and/or results, at a flushing step 122.

In some embodiments, the recurring load instructions all recur in respective code segments having the same flow-control. For example, if a loop does not contain any conditional branch instructions, then all loop iterations, including load instructions, will traverse the same flow-control trace. If, on the other hand, the loop does contain one or more conditional branch instructions, then different loop iterations may traverse different flow-control traces. In such a case, a recurring load instruction may not necessarily recur in all possible traces.

In some embodiments, the parallelization circuitry serves the outcome of a recurring load instruction from the internal register only to subsequent code that is associated with the same flow-control trace as the initial load instruction (whose outcome was saved in the internal register). In this context, the traces considered by the parallelization circuitry may be actual traces traversed by the code, or predicted traces that are expected to be traversed. In the latter case, if the prediction fails, the subsequent code may be flushed. In alternative embodiments, the parallelization circuitry serves the outcome of a recurring load instruction from the internal register to subsequent code regardless of whether it is associated with the same trace or not.

For the sake of clarity, the above description referred to a single group of read instructions that read from the same memory address. In some embodiments, the parallelization circuitry may handle two or more groups of recurring read instructions, each reading from a respective common address. Such groups may be identified and handled in the same region of the code containing segments that are at least partially repetitive. For example, the parallelization circuitry may handle multiple dedicated registers (like the MSG register described above) for this purpose.

In some cases, the recurring load instruction is located at or near the end of a loop iteration, and the subsequent code that depends on the read value is located at or near the beginning of a loop iteration. In such a case, the parallelization circuitry may serve a value obtained in one loop iteration to a subsequent loop iteration. The iteration in which the value was initially read and the iteration to which the value is served may be processed by different threads 24 or by the same thread.

In some embodiments, the parallelization circuitry is able to recognize that multiple load instructions read from the same address even when the address is specified indirectly using a pointer value stored in memory. Consider, for example, the code

```
ldr r3,[r4]
ldr r1,[r3,#4]
add r8,r1,r4
mov r3,r7
mov r1,r9
```

wherein r4 is global. In this example, the address [r4] holds a pointer. Nevertheless, the value of all loads to r1 (and r3) is the same in all iterations.

In some embodiments, the parallelization circuitry saves the information relating to the recurring load instructions as part of a data structure (referred to as a “scoreboard”) produced by monitoring the relevant region of the code. Certain aspects of monitoring and scoreboard

construction and usage are addressed, for example, in U.S. Patent Applications 14/578,516, 14/578,518, 14/583,119, 14/637,418, 14/673,884, 14/673,889 and 14/690,424, cited above. In such a scoreboard, the parallelization circuitry may save, for example, the address format or PC value. Whenever reaching this code region, the parallelization circuitry (e.g., the renaming unit) may retrieve the information from the scoreboard and add micro-ops or change the renaming scheme accordingly.

EXAMPLE RELATIONSHIP: LOAD-STORE INSTRUCTION PAIRS ACCESSING THE SAME MEMORY ADDRESS

In some embodiments, the parallelization circuitry identifies, based on the formats of the symbolic expressions, a store instruction and a subsequent load instruction that both access the same memory address in the external memory. Such a pair is referred to herein as a “load-store pair.” The parallelization circuitry saves the value stored by the store instruction in an internal register, and serves (or at least assigns for serving) the outcome of the load instruction from the internal register, without waiting for the value to be retrieved from external memory 41. The value may be served from the internal register to any subsequent code instructions that depend on the outcome of the load instruction in the pair. The internal register may comprise, for example, one of the dedicated registers in register file 50.

The identification of load-store pairs and the decision whether to serve the outcome from an internal register may be performed, for example, by the relevant decoding unit 32 or renaming unit 36.

In some embodiments, both the load instruction and the store instruction specify the address using the same symbolic format, such as in the code

str r1,[r2]

inst

inst

inst

ldr r8,[r2]

In other embodiments, the load instruction and the store instruction specify the address using different symbolic formats that nevertheless refer to the same memory address. Such load-store pairs may comprise, for example

str r1,[r2,#4]! and *ldr r8,[r2],*

or

str r1,[r2],#4 and *ldr r8,[r2,#-4]*

In the first example (*str r1,[r2,#4]!*), the value of r2 is updated to increase by 4 before the store address is calculated. Thus, the store and load refer to the same address. In the second example (*str r1,[r2],#4*), the value of r2 is updated to increase by 4 after the store address is calculated, while the load address is then calculated from the new value of r2 subtracted by 4. Thus, in this example too, the store and load refer to the same address.

In some embodiments, the store and load instructions of a given load-store pair are processed by the same hardware thread 24. In alternative embodiments, the store and load instructions of a given load-store pair may be processed by different hardware threads.

As explained above with regard to recurring load instructions, in the case of load-store pairs too, the parallelization circuitry may serve the outcome of the load instruction from an internal register by adding an instruction or micro-op to the code. This instruction or micro-op may be added at any suitable location in the code in which the data for the store instruction is ready (not necessarily after the store instruction – possibly before the store instruction). Adding the instruction or micro-op may be performed, for example, by the relevant decoding unit 32 or renaming unit 36.

Consider, for example, the following code:

```
str r8,[r6]
inst
inst
inst
ldr r1,[r6],#1
```

The parallelization circuitry may add the micro-op

```
mov MSGL,r8
```

that assigns the value of r8 into another register (which is referred to as MSGL) at a suitable location in which the value of r8 is available. Following the *ldr* instruction the parallelization circuitry may add the micro-op

```
mov r1,MSGL
```

that assigns the value of MSGL into register r1.

Alternatively, the parallelization circuitry may serve the outcome of the load instruction from an internal register by configuring the renaming scheme so that the outcome is served from the same physical register mapped by the store instruction. This operation, too, may be

performed at any suitable time in which the data for the store instruction is already assigned to the final physical register, e.g., once the micro-op that assigns the value to r8 has passed the renaming unit. For example, renaming unit 36 may assign the value stored by the store instruction to a certain physical register, and rename the instructions that depend on the outcome of the corresponding load instruction to receive the outcome from this physical register.

In an embodiment, the parallelization circuitry verifies that the registers participating in the symbolic expression of the address in the store instruction are not updated between the store instruction and the load instruction of the pair.

In an embodiment, the store instruction stores a word of a certain width (e.g., a 32-bit word), and the corresponding load instruction loads a word of a different width (e.g., an 8-bit byte) that is contained within the stored word. For example, the store instruction may store a 32-bit word in a certain address, and the load instruction in the pair may load some 8-bit byte within the 32-bit word. This scenario is also regarded as a load-store pair that accesses the same memory address.

To qualify as a load-store pair, the symbolic expressions of the addresses in the store and load instructions need not necessarily use the same registers. The parallelization circuitry may pair a store instruction and a load instruction together, for example, even if their symbolic expressions use different registers but are known to have the same values.

In some embodiments, the registers in the symbolic expressions of the addresses in the store and load instructions are indices, i.e., their values increment with a certain stride or other fixed calculation so as to address an array in the external memory. For example, the load instruction and corresponding store instruction may be located inside a loop, such that each pair accesses an incrementally-increasing memory address.

In some embodiments, the parallelization circuitry verifies, when serving the outcome of the load instruction in a load-store pair from an internal register, that the served value indeed matches the actual value retrieved by the load instruction from external memory 41. If a mismatch is found, the parallelization circuitry may flush subsequent instructions and results.

Any suitable verification scheme can be used for this purpose. For example, as explained above with regard to recurring load instructions, the parallelization circuitry (e.g., the renaming unit) may add an instruction or micro-op that performs the verification. The actual comparison may be performed by the ALU or alternatively in the LSU. Alternatively, the parallelization circuitry may verify that the registers appearing in the symbolic expression of the address in the store instruction are not written to between the store instruction and the corresponding load instruction. Further alternatively, the parallelization circuitry may check for various other

intervening events (e.g., fence instructions, or memory access by other entities) as explained above.

In some embodiments, the parallelization unit may inhibit the load instruction from being executed in the external memory. In an embodiment, instead of inhibiting the load instruction, the parallelization circuitry (e.g., the renaming unit) modifies the load instruction to an instruction or micro-op that performs the above-described verification.

In some embodiments, the parallelization circuitry serves the outcome of the load instruction in a load-store pair from the internal register only to subsequent code that is associated with a specific flow-control trace or traces in which the load-store pair was identified. For other traces, which may not comprise the load-store pair in question, the parallelization circuitry may execute the load instructions conventionally in the external memory.

In this context, the traces considered by the parallelization circuitry may be actual traces traversed by the code, or predicted traces that are expected to be traversed. In the latter case, if the prediction fails, the subsequent code may be flushed. In alternative embodiments, the parallelization circuitry serves the outcome of a load instruction from the internal register to subsequent code associated with any flow-control trace.

In some embodiments, the identification of the store or load instruction in the pair and the location for inserting micro-ops may also be based on factors such as the Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the load and store instructions in the program code. For example, when the load-store pair is identified in a loop, the parallelization circuitry may save the PC value of the load instruction. This information indicates to the parallelization circuitry exactly where to insert the additional micro-op whenever the processor traverses this PC.

Fig. 4 is a flow chart that schematically illustrates a method for processing code that contains load-store instruction pairs, in accordance with an embodiment of the present invention. The method begins with the parallelization circuitry identifying one or more load-store pairs that, based on the address format, access the same memory address, at a pair identification step 130.

For a given pair, the parallelization circuitry saves the value that is stored (or to be stored) by the store instruction in an internal register, at an internal saving step 134. At an internal serving step 138, the parallelization circuitry does not wait for the load instruction in the pair to retrieve the value from external memory. Instead, the parallelization circuitry serves the outcome of the load instruction, to any subsequent instructions that depend on this value, from the internal register.

The examples above refer to a single load-store pair in a given repetitive region of the code (e.g., loop). Generally, however, the parallelization circuitry may identify and handle two or more different load-store pairs in the same code region. Furthermore, multiple load instructions may be paired to the same store instruction. The parallelization circuitry may regard this scenario as multiple load store pairs, but assign the stored value to an internal register only once.

As explained above with regard to recurring load instructions, the parallelization circuitry may store the information on identification of load-store pairs in the scoreboard relating to the code region in question. In an alternative embodiment, the renaming unit may use the physical name of the register being stored as the operand of the registers to be loaded when the *mov* micro-op is added.

EXAMPLE RELATIONSHIP: LOAD-STORE INSTRUCTION PAIRS WITH PREDICTABLE MANIPULATION OF THE STORED VALUE

As explained above, in some embodiments the parallelization circuitry identifies a region of the code containing one or more code segments that are at least partially repetitive, wherein the code in this region comprises repetitive load-store pairs. In some embodiments, the parallelization circuitry further identifies that the value loaded from external memory is manipulated using some predictable calculation between the load instructions of successive iterations (or, similarly, between the load instruction and the following store instruction in a given iteration).

These identifications are performed, e.g., by the relevant decoding unit 32 or renaming unit 36, based on the formats of the symbolic expressions of the instructions. As will be explained below, the repetitive load-store pairs need not necessarily access the same memory address.

In some embodiments, the parallelization circuitry saves the loaded value in an internal register or other internal memory, and manipulates the value using the same predictable calculation. The manipulated value is then assigned to be served to subsequent code that depends on the outcome of the next load instruction, without having to wait for the actual load instruction to retrieve the value from the external memory.

Consider, for example, a loop that contains the code

```
A    ldr r1,[r6]
B    add r7,r6,r1
C    inst
```

D *inst*
 E *ldr r8,[r6]*
 F *add r8,r8,#1*
 G *str r8,[r6]*

5 in which r6 is a global register. Instructions E-G increment a counter value that is stored in memory address “[r6]”. Instructions A and B make use of the counter value that was set in the previous loop iteration. Between the load instruction and the store instruction, the program code manipulates the read value by some predictable manipulation (in the present example, incrementing by 1 in instruction F).

10 In the present example, instruction A depends on the value stored into “[r6]” by instruction G in the previous iteration. In some embodiments, the parallelization circuitry assigns the outcome of the load instruction (instruction A) to be served to subsequent code from an internal register (or other internal memory), without waiting for the value to be retrieved from external memory. The parallelization circuitry performs the same predictable manipulation on
 15 the internal register, so that the served value will be correct. When using this technique, instruction A still depends on instruction G in the previous iteration, but instructions that depend on the value read by instruction A can be processed earlier.

 In one embodiment, in the first loop iteration the parallelization circuitry adds the micro-op

20 *mov MSI,r1*

after instruction A or

mov MSI,r8

after instruction E and before instruction F, wherein MSI denotes an internal register, such as one of the dedicated registers in register file 50. In the subsequent loop iterations, the
 25 parallelization circuitry adds the micro-op

MSI,MSI,#1

at the beginning of the iteration, or at any other suitable location in the loop iteration before it is desired to make use of MSI. This micro-op increments the internal register MSI by 1, i.e., performs the same predictable manipulation of instruction F in the previous iteration. In

30 addition, the parallelization circuitry adds the micro-op

mov r1,MSI

(after the first increment micro-op was inserted) after each load instruction that accesses “[r6]” (after instructions A and E in the present example – note that after instruction E the micro-op *mov r8,MSI* would be added). As a result, any instruction that depends on these load instructions

will be served from the internal register MSI instead of from the external memory. Adding the instructions or micro-ops above may be performed, for example, by the relevant decoding unit 32 or renaming unit 36.

In the above example, the parallelization circuitry performs the predictable manipulation once in each iteration, so as to serve the correct value to the code of the next iteration. In alternative embodiments, the parallelization circuitry may perform the predictable manipulation multiple times in a given iteration, and serve different predicted values to code of different subsequent iterations. In the counter incrementing example above, in the first iteration the parallelization circuitry may calculate the next n values of the counter, and provide the code of each iteration with the correct counter value. Any of these operations may be performed without waiting for the load instruction to retrieve the counter value from external memory. This advance calculation may be repeated every n iterations.

In an alternative embodiment, in the first iteration, the parallelization circuitry renames the destination register $r1$ (in instruction A) to a physical register denoted $p8$. The parallelization circuitry then adds one or more micro-ops or instructions (or modifies an existing micro-op, e.g., instruction A) to calculate a vector of n $r8, r8, \#1$ values. The vector is saved in a set of dedicated registers $m_1 \dots m_n$, e.g., in register file 50. In the subsequent iterations, the parallelization circuitry renames the operands of the add instructions (instruction D) to read from respective registers $m_1 \dots m_n$ (according to the iteration number). The parallelization circuitry may comprise suitable vector-processing hardware for performing these vectors in a small number of cycles.

Fig. 5 is a flow chart that schematically illustrates a method for processing code that contains repetitive load-store instruction pairs with intervening data manipulation, in accordance with an embodiment of the present invention. The method begins with the parallelization circuitry identifying a code region containing repetitive load-store pairs having intervening data manipulation, at an identification step 140. The parallelization circuitry analyzes the code so as to identify both the load-store pairs and the data manipulation. The data manipulation typically comprises an operation performed by the ALU, or by another execution units such as an FPU or MAC unit. Typically although not necessarily, the manipulation is performed by a single instruction.

When the code region in question is a program loop, for example, each load-store pair typically comprises a store instruction in a given loop iteration and a load instruction in the next iteration that reads from the same memory address.

For a given load-store pair, the parallelization circuitry assigns the value that was loaded by a first load instruction in an internal register, at an internal saving step 144. At a manipulation step 148, the parallelization circuitry applies the same data manipulation (identified at step 140) to the internal register. The manipulation may be applied, for example, using the ALU, FPU or MAC unit.

At an internal serving step 152, the parallelization circuitry does not wait for the next load instruction to retrieve the manipulated value from external memory. Instead, the parallelization circuitry assigns the manipulated value (calculated at step 148) to any subsequent instructions that depend on the next load instruction, from the internal register.

In the examples above, the counter value is always stored in (and retrieved from) the same memory address (“ $[r6]$ ”, wherein $r6$ is a global register). This condition, however, is not mandatory. For example, each iteration may store the counter value in a different (e.g., incrementally increasing) address in external memory 41. In other words, within a given iteration the value may be loaded from a given address, manipulated and then stored in a different address. A relationship still exists between the memory addresses accessed by the load and store instructions of different iterations: The load instruction in a given iteration accesses the same address as the store instruction of the previous iteration.

In an embodiment, the store instruction stores a word of a certain width (e.g., a 32-bit word), and the corresponding load instruction loads a word of a different width (e.g., an 8-bit byte) that is contained within the stored word. For example, the store instruction may store a 32-bit word in a certain address, and the load instruction in the pair may load some 8-bit byte within the 32-bit word. This scenario is also regarded as a load-store pair that accesses the same memory address. In such embodiments, the predictable manipulation should be applied to the smaller-size word loaded by the load instruction.

As in the previous examples, the parallelization circuitry typically verifies, when serving the manipulated value from the internal register, that the served value indeed matches the actual value after retrieval by the load instruction and manipulation. If a mismatch is found, the parallelization circuitry may flush subsequent instructions and results. Any suitable verification scheme can be used for this purpose, such as by adding one or more instructions or micro-ops, or by verifying that the address in the store instruction is not written to between the store instruction and the corresponding load instruction.

Further alternatively, the parallelization circuitry may check for various other intervening events (e.g., fence instructions, or memory access by other entities) as explained above.

Addition of instructions or micro-ops can be performed, for example, by the renaming unit. The actual comparison between the served value and the actual value may be performed by the ALU or LSU.

In some embodiments, the parallelization unit may inhibit the load instruction from being executed in the external memory. In an embodiment, instead of inhibiting the load instruction, the parallelization circuitry (e.g., the renaming unit) modifies the load instruction to an instruction or micro-op that performs the above-described verification.

In some embodiments, the parallelization circuitry serves the manipulated value from the internal register only to subsequent code that is associated with a specific flow-control trace or group of traces, e.g., only if the subsequent load-store pair is associated with the same flow-control trace as the current pair. In this context, the traces considered by the parallelization circuitry may be actual traces traversed by the code, or predicted traces that are expected to be traversed. In the latter case, if the prediction fails, the subsequent code may be flushed. In alternative embodiments, the parallelization circuitry serves the manipulated value from the internal register to subsequent code associated with any flow-control trace.

In some embodiments, the decision to serve the manipulated value from an internal register, and/or the identification of the location in the code for adding or manipulate micro-ops, may also consider factors such as the Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the load and store instructions in the program code. The decision to serve the manipulated value from an internal register, and/or the identification of the code to which the manipulated value should be served, may be carried out, for example, by the relevant renaming or decoding unit.

The examples above refer to a single predictable manipulation and a single sequence of repetitive load-store pairs in a given region of the code (e.g., loop). Generally, however, the parallelization circuitry may identify and handle two or more different predictable manipulations, and/or two or more sequences of repetitive load-store pairs, in the same code region. Furthermore, as described above, multiple load instructions may be paired to the same store instruction. This scenario may be considered by the parallelization circuitry as multiple load-store pairs, wherein the stored value is assigned to an internal register only once.

As explained above, the parallelization circuitry may store the information on identification of load-store pairs and predictable manipulations in the scoreboard relating to the code region in question.

EXAMPLE RELATIONSHIP: RECURRING LOAD INSTRUCTIONS THAT ACCESS A PATTERN OF NEARBY MEMORY ADDRESSES

In some embodiments, the parallelization circuitry identifies a region of the program code, which comprises a repetitive sequence of load instructions that access different but nearby
5 memory addresses in external memory 41. Such a scenario occurs, for example, in a program loop that reads values from a vector or other array stored in the external memory, in accessing the stack, or in image processing or filtering applications.

In one embodiment, the load instructions in the sequence access incrementing adjacent memory addresses, e.g., in a loop that reads respective elements of a vector stored in the external
10 memory. In another embodiment, the load instructions in the sequence access addresses that are not adjacent but differ from one another by a constant offset (sometimes referred to as “stride”). Such a case occurs, for example, in a loop that reads a particular column of an array.

Further alternatively, the load instructions in the sequence may access addresses that increment or decrement in accordance with any other suitable predictable pattern. Typically
15 although not necessarily, the pattern is periodic. Another example of a periodic pattern, more complex than a stride, occurs when reading two or more columns of an array (e.g., matrix) stored in memory.

The above examples refer to program loops. Generally, however, the parallelization circuitry may identify any other region of code that comprises such repetitive load instructions,
20 e.g., in sections of loop iterations, sequential code and/or any other suitable instruction sequence.

The parallelization circuitry identifies the sequence of repetitive load instructions, and the predictable pattern of the addresses being read from, based on the formats of the symbolic expressions that specify the addresses in the load instructions. The identification is thus performed early in the pipeline, e.g., by the relevant decoding unit or renaming unit.

Having identified the predictable pattern of addresses accessed by the load instruction
25 sequence, the parallelization circuitry may access a plurality of the addresses in response to a given read instruction in the sequence, before the subsequent read instructions are processed. In some embodiments, in response to a given read instruction, the parallelization circuitry uses the identified pattern to read a plurality of future addresses in the sequence into internal registers
30 (or other internal memory). The parallelization circuitry may then assign any of the read values from the internal memory to one or more future instructions that depend on the corresponding read instruction, without waiting for that read instruction to read the value from the external memory.

In some embodiments, the basic read operation performed by the LSUs reads a plurality of data values from a contiguous block of addresses in memory 43 (possibly via cache 56 or 42). This plurality of data values is sometimes referred to as a “cache line.” A cache line may comprise, for example, sixty-four bytes, and a single data value may comprise, for example four or eight bytes, although any other suitable cache-line size can be used. Typically, the LSU or cache reads an entire cache line regardless of the actual number of values that were requested, even when requested to read a single data value from a single address.

In some embodiments, the LSU or cache reads a cache line in response to a given read instruction in the above-described sequence. Depending on the pattern of addresses, the cache line may also contain one or more data values that will be accessed by one or more subsequent read instructions in the sequence (in addition to the data value requested by the given read instruction). In an embodiment, the parallelization circuitry extracts the multiple data values from the cache line based on the pattern of addresses, saves them in internal registers, and serves them to the appropriate future instructions.

Thus, in the present context, the term “nearby addresses” means addresses that are close to one another relative to the cache-line size. If, for example, each cache line comprises n data values, the parallelization circuitry may repeat the above process every n read instructions in the sequence.

Furthermore, if the parallelization circuitry, LSU or cache identifies that in order to load n data values from memory there is a need to get another cache line, it may initiate a read from memory of the relevant cache line. Alternatively, instead of reading the next cache line into the LSU, it is possible to set a prefetch trigger based on the identification and the pattern, for reading the data to L1 cache 56.

This technique is especially effective when a single cache line comprises many data values that will be requested by future read instructions in the sequence (e.g., when a single cache line comprises many periods of the pattern). The performance benefit is also considerable when the read instructions in the sequence arrive in execution units 52 at large intervals, e.g., when they are separated by many other instructions.

Fig. 6 is a flow chart that schematically illustrates a method for processing code that contains recurring load instructions from nearby memory addresses, in accordance with an embodiment of the present invention. The method begins at a sequence identification step 160, with the parallelization circuitry identifying a repetitive sequence of read instructions that access respective memory addresses in memory 43 in accordance with a predictable pattern.

In response to a given read instruction in the sequence, an LSU in execution units 52 (or the cache) reads one or several cache lines from memory 43 (possibly via cache 56 or 42), at a cache-line readout step 164. At an extraction step 168, the parallelization circuitry extracts the data value requested by the given read instruction from the cache line. In addition, the parallelization circuitry uses the identified pattern of addresses to extract from the cache lines one or more data values that will be requested by one or more subsequent read instructions in the sequence. For example, if the pattern indicates that the read instructions access every fourth address starting from some base address, the parallelization circuitry may extract every fourth data value from the cache lines.

As an internal storage step 168, the parallelization circuitry saves the extracted data values in internal memory. The extracted data values may be saved, for example, in a set of internal registers in register file 50. The other data in the cache lines may be discarded. In other embodiments, the parallelization circuitry may copy the entire cache lines to the internal memory, and later assign the appropriate values from the internal memory in accordance with the pattern.

At a serving step 172, the parallelization circuitry serves the data values from the internal registers to the subsequent code instructions that depend on them. For example, the k^{th} extracted data value may be served to any instruction that depends on the outcome of the k^{th} read instruction following the given read instruction. The k^{th} extracted data value may be served from the internal memory without waiting for the k^{th} read instruction to retrieve the data value from external memory.

Consider, for example, a loop that contains the following code:

```
ldr r1,[r6],#4
add r7,r6,r1
```

wherein r6 is a global register. This loop reads data values from every fourth address, starting from some base address that is initialized at the beginning of the loop. As explained above, the parallelization circuitry may identify the code region containing this loop, identify the predictable pattern of addresses, and then extract and serve multiple data values from a retrieved cache line.

In some embodiments, this mechanism is implemented by adding one or more instructions or micro-ops to the code, or modifying existing one or more instructions or micro-ops, e.g., by the relevant renaming unit 36.

Referring to the example above, in an embodiment, in the first loop iteration the parallelization circuitry modifies the load (*ldr*) instruction to

vec_ldr MA,r1

wherein MA denotes a set of internal registers, e.g., in register file 50.

5 In subsequent loop iterations, the parallelization circuitry adds the following instruction after the *ldr* instruction:

mov r1,MA(iteration_number)

The *vec_ldr* instruction in the first loop iteration saves multiple retrieved values to the MA registers, and the *mov* instruction in the subsequent iterations assigns the values from the MA registers to register r1 with no direct relationship to the *ldr* instruction. This allows the subsequent *add* instruction to be issued/executed without waiting for the *ldr* instruction to complete.

10 In an alternative embodiment, the parallelization circuitry (e.g., renaming unit 36) implements the above mechanism by proper setting of the renaming scheme. Referring to the example above, in an embodiment, in the first loop iteration the parallelization circuitry modifies the load (*ldr*) instruction to

vec_ldr MA,r1

In the subsequent loop iterations, the parallelization circuitry renames the operands of the *add* instructions to read from *MA(iteration_num)* even though the new *ldr* destination is renamed to a different physical register. In addition, the parallelization circuitry does not release the mapping of the MA registers in a conventional manner, i.e., on the next time the write to r1 is committed. Instead, the mapping is retained until all data values extracted from the current cache line have been served.

20 In the two examples above, the parallelization circuitry may use a series of *ldr* micro-ops instead of the *ldr_vec* instruction.

For a given pattern of addresses, each cache line contains a given number of data values. If the number of loop iterations is larger than the number of data values per cache line, or if one of the loads crosses the cache-line boundary (e.g., because since the loads are not necessarily aligned with the beginning of a cache line), then a new cache line should be read when the current cache line is exhausted. In some embodiments, the parallelization circuitry automatically instructs the LSU to read a next cache line.

Other non-limiting examples of repetitive load instructions that access predictable nearby address patterns may comprise:

ldr r2,[r5,r1] wherein r1 is an index

or

ldr r2,[r1,#4]!

or

ldr r2, [r1],#4

5 or

ldr r3,[r8,sl,lsl #2] wherein sl is an index

or an example of an unrolled loop:

ldr r1,[r5,#4]

ldr r1,[r5,#8]

10 *ldr r1,[r5,#12]*

...

In some embodiments, all the load instructions in the sequence are processed by the same hardware thread 24 (e.g., when processing an unrolled loop, or when the processor is a single-thread processor). In alternative embodiments, the load instructions in the sequence may be processed by at least two different hardware threads.

In some embodiments, the parallelization circuitry verifies, when serving the outcome of a load instruction in the sequence from the internal memory, that the served value indeed matches the actual value retrieved by the load instruction from external memory. If a mismatch is found, the parallelization circuitry may flush subsequent instructions and results. Any suitable verification scheme can be used for this purpose. For example, as explained above, the parallelization circuitry (e.g., the renaming unit) may add an instruction or micro-op that performs the verification. The actual comparison may be performed by the ALU or alternatively in the LSU.

As explained above, the parallelization circuitry may also verify, e.g., based on the formats of the symbolic expressions of the instructions, that no intervening event causes a mismatch between the served values and the actual values in the external memory.

In yet other embodiments, the parallelization circuitry may initially assume that no intervening event affects the memory address in question. If, during execution, some verification mechanism fails, the parallelization circuitry may deduce that an intervening event possibly exists, and refrain from serving the outcome from the internal memory.

In some embodiments, the parallelization unit may inhibit the load instruction from being executed in the external memory. In an embodiment, instead of inhibiting the load instruction, the parallelization circuitry (e.g., the renaming unit) modifies the load instruction to an instruction or micro-op that performs the above-described verification.

In some embodiments, the parallelization circuitry serves the outcome of a load instruction from the internal memory only to subsequent code that is associated with one or more specific flow-control traces (e.g., traces that contain the load instruction). In this context, the traces considered by the parallelization circuitry may be actual traces traversed by the code, or predicted traces that are expected to be traversed. In the latter case, if the prediction fails, the subsequent code may be flushed. In alternative embodiments, the parallelization circuitry serves the outcome of a load instruction from the internal register to subsequent code associated with any flow-control trace.

In some embodiments, the decision to assign the outcome from an internal register, and/or the identification of the locations in the code for adding or modifying instructions or micro-ops, may also consider factors such as the Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the load instructions in the program code.

In some embodiments, the MA registers may reside in a register file having characteristics and requirements that differ from other registers of the processor. For example, this register file may have a dedicated write port buffer from the LSU, and only read ports from the other execution units 52.

The examples above refer to a single sequence of load instructions that access a single predictable pattern of memory addresses in a region of the code. Generally, however, the parallelization circuitry may identify and handle in the same code region two or more different sequences of load instructions, which access two or more respective patterns of memory addresses.

As explained above, the parallelization circuitry may store the information on identification of the sequence of load instructions, and on the predictable pattern of memory addresses, in the scoreboard relating to the code region in question.

In the examples given in Figs. 2-6 above, the relationships between memory-access instructions and the resulting actions, e.g., adding or modifying instructions or micro-ops, are performed at runtime. In alternative embodiments, however, at least some of these functions may be performed by a compiler that compiles the program code for execution by processor 20. Thus, in some embodiments, processor 20 identifies and acts upon the relationships between memory-access instructions, at partially based on hints or other indications embedded in the program code by the compiler.

It will thus be appreciated that the embodiments described above are cited by way of example, and that the present invention is not limited to what has been particularly shown and described hereinabove. Rather, the scope of the present invention includes both combinations

and sub-combinations of the various features described hereinabove, as well as variations and modifications thereof which would occur to persons skilled in the art upon reading the foregoing description and which are not disclosed in the prior art. Documents incorporated by reference in the present patent application are to be considered an integral part of the application except
5 that to the extent any terms are defined in these incorporated documents in a manner that conflicts with the definitions made explicitly or implicitly in the present specification, only the definitions in the present specification should be considered.

CLAIMS

1. A method, comprising:

in a processor, processing program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names;

identifying a relationship between the memory addresses accessed by two or more of the memory-access instructions, based on respective formats of the memory addresses specified in the symbolic expressions; and

assigning an outcome of at least one of the memory-access instructions to be served from an internal memory in the processor, based on the identified relationship.

2. The method according to claim 1, wherein identifying the relationship between the memory addresses is independent of actual numerical values of the memory addresses.

3. The method according to claim 2, wherein identifying the relationship between the memory addresses is performed at a point in time at which the actual numerical values of the memory addresses are undefined.

4. The method according to claim 2, wherein identifying the relationship is performed by a given pipeline stage in a pipeline of the processor, and wherein the actual numerical values of the memory addresses are calculated in another pipeline stage that is located later in the pipeline than the given pipeline stage.

5. The method according to claim 1, wherein identifying the relationship comprises searching the program code for memory-access instructions that specify the memory addresses using identical symbolic expressions.

6. The method according to claim 1, wherein identifying the relationship comprises searching the program code for memory-access instructions that specify the memory addresses using different symbolic expressions that refer to the same memory address.

7. The method according to claim 1, wherein assigning the outcome of at least one of the memory-access instructions is performed by a decoding unit or a renaming unit in a pipeline of the processor.

8. The method according to any of claims 1-7, wherein assigning the outcome to be served from the internal memory further comprises:

executing a memory-access instruction in the external memory; and

verifying that the outcome of the memory-access instruction executed in the external memory matches the outcome assigned to the memory-access instruction from the internal memory.

9. The method according to claim 8, wherein verifying the outcome comprises comparing the outcome of the memory-access instruction executed in the external memory with the outcome assigned to the memory-access instruction from the internal memory.

10. The method according to claim 8, wherein verifying the outcome comprises verifying that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

11. The method according to claim 8, wherein verifying the outcome comprises adding to the program code one or more instructions or micro-ops that verify the outcome, or modifying one or more existing instructions or micro-ops to the instructions or micro-ops that verify the outcome.

12. The method according to claim 8, further comprising flushing subsequent code upon finding that the outcome executed in the external memory does not match the outcome served from the internal memory.

13. The method according to any of claims 1-7, further comprising inhibiting the at least one of the memory-access instructions from being executed in the external memory.

14. The method according to any of claims 1-7, further comprising parallelizing execution of the program code, including assignment of the outcome from the internal memory, over multiple hardware threads.

15. The method according to any of claims 1-7, wherein processing the program code comprises executing the program code, including assignment of the outcome from the internal memory, in a single hardware thread.

16. The method according to any of claims 1-7, wherein identifying the relationship comprises identifying the memory-access instructions in a loop or a function.

17. The method according to any of claims 1-7, wherein identifying the relationship is performed at runtime.

18. The method according to any of claims 1-7, wherein identifying the relationship is performed, at least partly, based on indications embedded in the program code.

19. A processor, comprising:
an internal memory; and

processing circuitry, which is configured to process program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names, to identify a relationship between the memory addresses accessed by two or more of the memory-access instructions, based on respective formats of the memory addresses specified in the symbolic expressions, and to assign an outcome of at least one of the memory-access instructions to be served from the internal memory, based on the identified relationship.

20. The processor according to claim 19, wherein the processing circuitry is configured to identify the relationship between the memory addresses independently of actual numerical values of the memory addresses.

21. The processor according to claim 20, wherein the processing circuitry is configured to identify the relationship between the memory addresses at a point in time at which the actual numerical values of the memory addresses are undefined.

22. The processor according to claim 20, wherein the processing circuitry is configured to identify the relationship in a given pipeline stage in a pipeline of the processor, and to calculate the actual numerical values of the memory addresses in another pipeline stage that is located later in the pipeline than the given pipeline stage.

23. The processor according to claim 19, wherein the processing circuitry is configured to identify the relationship by searching the program code for memory-access instructions that specify the memory addresses using identical symbolic expressions.

24. The processor according to claim 19, wherein the processing circuitry is configured to identify the relationship by searching the program code for memory-access instructions that specify the memory addresses using different symbolic expressions that refer to the same memory address.

25. The processor according to claim 19, wherein the processing circuitry is configured to assign the outcome of at least one of the memory-access instructions by a decoding unit or a renaming unit in a pipeline of the processor.

26. The processor according to any of claims 19-25, wherein the processing circuitry is configured to assign the outcome to be served from the internal memory by:
executing a memory-access instruction in the external memory; and

verifying that the outcome of the memory-access instruction executed in the external memory matches the outcome assigned to the memory-access instruction from the internal memory.

27. The processor according to claim 26, wherein the processing circuitry is configured to verify the outcome by comparing the outcome of the memory-access instruction executed in the external memory with the outcome assigned to the memory-access instruction from the internal memory.

28. The processor according to claim 26, wherein the processing circuitry is configured to verify the outcome by verifying that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

29. The processor according to claim 26, wherein the processing circuitry is configured to add to the program code an instruction or micro-op that verifies the outcome, or to modify an existing instruction or micro-op to the instruction or micro-op that verifies the outcome.

30. The processor according to any of claims 19-25, wherein the processing circuitry is configured to inhibit the at least one of the memory-access instructions from being executed in the external memory.

31. A method, comprising:

in a processor, processing program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names;

identifying, based on respective formats of the memory addresses specified in the symbolic expressions, a sequence of load instructions that access a predictable pattern of memory addresses in the external memory;

retrieving from the external memory at least one cache line that comprises a plurality of data values;

saving from the cache line to the internal memory, based on the predictable pattern, two or more of the data values that are requested by respective load instructions in the sequence; and

assigning the saved data values to be served from the internal memory to one or more instructions that depend on the respective load instructions.

32. The method according to claim 31, wherein identifying the load instructions is independent of actual numerical values of the memory addresses.

33. The method according to claim 31 or 32, wherein saving the data values comprises determining locations of the data values in the cache lines based on the pattern of the memory addresses in the sequence of load instructions, and copying the data values from the determined locations in the cache lines to the internal memory.

34. The method according to claim 31 or 32, further comprising fetching at least one additional cache line from the external memory in response to identifying that the predictable pattern extends beyond the retrieved cache line.

35. The method according to claim 34, wherein fetching the additional cache line comprises prefetching the additional cache line to a cache memory.

36. The method according to claim 31 or 32, wherein assigning the data values comprises adding to the program code one or more instructions or micro-ops that serve the data values, or modifying one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the data values.

37. The method according to claim 36, wherein one or more of the added or modified instructions or micro-ops save the two or more data values from the cache line to the internal memory.

38. The method according to claim 36, wherein adding or modifying the instructions or micro-ops is performed by a decoding unit or a renaming unit in a pipeline of the processor.

39. The method according to claim 31 or 32, wherein assigning the data values comprises: saving the two or more data values in respective physical registers of the processor; and renaming the instructions that depend on the respective load instructions to receive the data values from the physical registers.

40. The method according to claim 31 or 32, wherein the load instructions in the sequence are processed by the same hardware thread.

41. The method according to claim 31 or 32, wherein the load instructions in the sequence are processed by at least two different hardware threads.

42. The method according to claim 31 or 32, wherein assigning the outcome to be served from the internal memory comprises inhibiting one or more of the load instructions from being executed in the external memory.

43. The method according to claim 31 or 32, wherein assigning the data values comprises providing the data values from the internal memory only to instructions that are associated with one or more specific flow-control traces.

44. The method according to claim 31 or 32, wherein assigning the data values comprises providing the data values from the internal memory to instructions that are associated with any flow-control trace.

45. The method according to claim 31 or 32, wherein assigning the data values comprises marking a location in the program code, to be modified for assigning the data values, based on at least one parameter selected from a group of parameters consisting of Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the load instructions in the program code.

46. The method according to claim 31 or 32, wherein assigning the data values further comprises:

executing the load instructions in the external memory; and

verifying that outcomes of the load instructions executed in the external memory matches the data values assigned from the internal memory.

47. The method according to claim 46, wherein verifying the data values comprises comparing the outcomes of the load instructions executed in the external memory to the data values assigned from the internal memory.

48. The method according to claim 46, wherein verifying the data values comprises verifying that no intervening event causes a mismatch between the outcomes in the external memory and the data values assigned from the internal memory.

49. The method according to claim 46, wherein verifying the data values comprises adding to the program code one or more instructions or micro-ops that verify the data values, or modifying one or more existing instructions or micro-ops to the instructions or micro-ops that verify the data values.

50. The method according to claim 46, further comprising flushing subsequent code upon finding that the outcomes executed in the external memory do not match the data values served from the internal memory.

51. The method according to claim 31 or 32, further comprising parallelizing execution of the program code, including assignment of the data values from the internal memory, over multiple hardware threads.

52. The method according to claim 31 or 32, wherein processing the program code comprises executing the program code, including assignment of the data values from the internal memory, in a single hardware thread.

53. The method according to claim 31 or 32, wherein identifying the sequence is performed, at least partly, based on indications embedded in the program code.

54. A processor, comprising:

an internal memory; and

processing circuitry, which is configured to process program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names, to identify, based on respective formats of the memory addresses specified in the symbolic expressions, a sequence of load instructions that access a predictable pattern of memory addresses in the external memory, to retrieve from the external memory at least one cache line that comprises a plurality of data values, to save from the cache line to the internal memory, based the predictable pattern, two or more of the data values that are requested by respective load instructions in the sequence, and to assign the saved data values to be served from the internal memory to one or more instructions that depend on the respective load instructions.

55. The processor according to claim 54, wherein the processing circuitry is configured to determine locations of the data values in the cache lines based on the pattern of the memory addresses in the sequence of load instructions, and to copy the data values from the determined locations in the cache lines to the internal memory.

56. The processor according to claim 54 or 55, wherein the processing circuitry is configured to fetch at least one additional cache line from the external memory in response to identifying that the predictable pattern extends beyond the retrieved cache line.

57. The processor according to claim 54 or 55, wherein the processing circuitry is configured to add to the program code one or more instructions or micro-ops that serve the data values, or to modify one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the data values.

58. The processor according to claim 54 or 55, wherein the processing circuitry is configured to assign the data values by:

saving the two or more data values in respective physical registers of the processor; and
renaming the instructions that depend on the respective load instructions to receive the

5 data values from the physical registers.

59. The processor according to claim 54 or 55, wherein the processing circuitry is configured to inhibit one or more of the load instructions from being executed in the external memory.

60. The processor according to claim 54 or 55, wherein the processing circuitry is configured to assign the data values by:

10 executing the load instructions in the external memory; and
verifying that outcomes of the load instructions executed in the external memory
matches the data values assigned from the internal memory.

61. A method, comprising:

in a processor, processing program code that includes memory-access instructions,
15 wherein at least some of the memory-access instructions comprise symbolic expressions that
specify memory addresses in an external memory in terms of one or more register names;

identifying in the program code at least first and second load instructions that access a
same memory address in the external memory, based on respective formats of the memory
addresses specified in the symbolic expressions of the load instructions; and

20 assigning an outcome of at least one of the load instructions to be served from an internal
memory in the processor.

62. The method according to claim 61, wherein identifying the first and second load
instructions further comprises identifying that no store instruction accesses the same memory
address between the first and second load instructions.

25 63. The method according to claim 61, wherein assigning the outcome comprises reading a
value from the same memory address in response to the first load instruction, saving the value
in the internal memory, and assigning the value in response to the second load instruction from
the internal memory.

64. The method according to claim 61, wherein identifying the first and second load
30 instructions comprises identifying that the symbolic expressions in the first and second load
instructions are defined in terms of one or more registers that are not written to between the first
and second load instructions.

65. The method according to claim 61, wherein assigning the outcome comprises providing the outcome from the internal memory only if the second load instruction is associated with the same flow-control trace as the first load instruction.

66. The method according to claim 61, wherein assigning the outcome comprises providing the outcome from the internal memory regardless of whether the second load instruction is associated with the same flow-control trace as the first load instruction.

67. The method according to claim 61, wherein assigning the outcome comprises marking a location in the program code, to be modified for assigning the outcome, based on at least one parameter selected from a group of parameters consisting of Program-Counter (PC) values, program addresses, destination registers, instruction-indices and address-operands of the load instructions in the program code.

68. The method according to any of claims 61-67, wherein assigning the outcome comprises adding to the program code one or more instructions or micro-ops that serve the outcome, or modifying one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the outcome.

69. The method according to claim 68, wherein one of the added or modified instructions or micro-ops saves the outcome of the first load instruction to the internal memory.

70. The method according to claim 69, wherein one of the added or modified instructions or micro-ops copies the outcome from the internal memory to a destination register of the second load instruction.

71. The method according to claim 68, wherein adding or modifying the instructions or micro-ops is performed by a decoding unit or a renaming unit in a pipeline of the processor.

72. The method according to any of claims 61-67, wherein assigning the outcome to be served from the internal memory further comprises:

executing the second load instruction in the external memory; and
verifying that the outcome of the second load instruction executed in the external memory matches the outcome assigned to the second load instruction from the internal memory.

73. The method according to claim 72, wherein verifying the outcome comprises comparing the outcome of the second load instruction executed in the external memory to the outcome assigned to the second load instruction from the internal memory.

74. The method according to claim 72, wherein verifying the outcome comprises verifying that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

75. The method according to claim 72, wherein verifying the outcome comprises adding to the program code one or more instructions or micro-ops that verify the outcome, or modifying one or more existing instructions or micro-ops to the instructions or micro-ops that verify the outcome.

76. The method according to claim 72, further comprising flushing subsequent code upon finding that the outcome executed in the external memory does not match the outcome served from the internal memory.

77. The method according to any of claims 61-67, further comprising inhibiting the at least one of the load instructions from being executed in the external memory.

78. The method according to any of claims 61-67, further comprising parallelizing execution of the program code, including assignment of the outcome from the internal memory, over multiple hardware threads.

79. The method according to any of claims 61-67, wherein processing the program code comprises executing the program code, including assignment of the outcome from the internal memory, in a single hardware thread.

80. The method according to any of claims 61-67, wherein assigning the outcome comprises: saving the outcome of the first load instruction in a physical register of the processor; and

renaming both the first load instruction and the second load instruction to receive the outcome from the physical register.

81. The method according to any of claims 61-67, wherein identifying the load instructions is performed, at least partly, based on indications embedded in the program code.

82. A processor, comprising:

an internal memory; and

processing circuitry, which is configured to process program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names, to identify in the program code at least first and second load instructions that

access a same memory address in the external memory, based on respective formats of the memory addresses specified in the symbolic expressions of the load instructions, and to assign an outcome of at least one of the load instructions to be served from the internal memory.

83. The processor according to claim 82, wherein the processing circuitry is further configured to identify that no store instruction accesses the same memory address between the first and second load instructions.

84. The processor according to claim 82, wherein the processing circuitry is configured to assign the outcome by reading a value from the same memory address in response to the first load instruction, saving the value in the internal memory, and assigning the value in response to the second load instruction from the internal memory.

85. The processor according to claim 82, wherein the processing circuitry is configured to identify that the symbolic expressions in the first and second load instructions are defined in terms of one or more registers that are not written to between the first and second load instructions.

86. The processor according to claim 82, wherein the processing circuitry is configured to assign the outcome from the internal memory only if the second load instruction is associated with the same flow-control trace as the first load instruction.

87. The processor according to claim 82, wherein the processing circuitry is configured to assign the outcome from the internal memory regardless of whether the second load instruction is associated with the same flow-control trace as the first load instruction.

88. The processor according to claim 82, wherein the processing circuitry is configured to mark a location in the program code, to be modified for assigning the outcome, based on at least one parameter selected from a group of parameters consisting of Program-Counter (PC) values, program addresses, destination registers, instruction-indices and address-operands of the load instructions in the program code.

89. The processor according to any of claims 82-88, wherein the processing circuitry is configured to add to the program code one or more instructions or micro-ops that serve the outcome, or to modify an existing instruction or micro-op to the one or more instructions or micro-ops that serve the outcome.

90. The processor according to claim 89, wherein one of the added or modified instructions or micro-ops saves the outcome of the first load instruction to the internal memory.

91. The processor according to claim 90, wherein one of the added or modified instructions or micro-ops copies the outcome from the internal memory to a destination register of the second load instruction.

92. The processor according to claim 89, wherein the processing circuitry is configured to add or modify the instructions or micro-ops by a decoding unit or a renaming unit in a pipeline of the processor.

93. The processor according to any of claims 82-88, wherein the processing circuitry is configured to assign the outcome to be served from the internal memory by:

executing the second load instruction in the external memory; and

verifying that the outcome of the second load instruction executed in the external memory matches the outcome assigned to the second load instruction from the internal memory.

94. The processor according to claim 93, wherein the processing circuitry is configured to verify the outcome by comparing the outcome of the second load instruction executed in the external memory to the outcome assigned to the second load instruction from the internal memory.

95. The processor according to claim 93, wherein the processing circuitry is configured to verify the outcome by verifying that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

96. The processor according to claim 93, wherein the processing circuitry is configured to add to the program code an instruction or micro-op that verifies the outcome, or to modify an existing instruction or micro-op to the instruction or micro-op that verifies the outcome.

97. The processor according to claim 93, wherein the processing circuitry is configured to flush subsequent code upon finding that the outcome executed in the external memory does not match the outcome served from the internal memory.

98. The processor according to any of claims 82-88, wherein the processing circuitry is configured to inhibit the at least one of the load instructions from being executed in the external memory.

99. The processor according to any of claims 82-88, wherein the processing circuitry is configured to parallelize execution of the program code, including assignment of the outcome from the internal memory, over multiple hardware threads.

100. The processor according to any of claims 82-88, wherein the processing circuitry is configured to execute the program code, including assignment of the outcome from the internal memory, in a single hardware thread.

101. The processor according to any of claims 82-88, wherein the processing circuitry is configured to assign the outcome by:

saving the outcome of the first load instruction in a physical register of the processor;
and

renaming both the first load instruction and the second load instruction to receive the outcome from the physical register.

102. The processor according to any of claims 82-88, wherein the processing circuitry is configured to identify the load instructions, at least partly based on indications embedded in the program code.

103. A method, comprising:

in a processor, processing program code that includes memory-access instructions,
wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names;

identifying at least a store instruction and a subsequent load instruction that access the same memory address in the external memory, based on respective formats of the memory addresses specified in the symbolic expressions; and

assigning an outcome of at least one of the memory-access instructions, to be served to one or more instructions that depend on the load instruction, from an internal memory in the processor.

104. The method according to claim 103, wherein both the store instruction and the load instruction specify the memory address using the same symbolic expression.

105. The method according to claim 103, wherein the store instruction and the load instruction specify the memory address using different symbolic expressions.

106. The method according to claim 103, wherein both the store instruction and the load instruction are processed by the same hardware thread.

107. The method according to claim 103, wherein the store instruction and the load instruction are processed by different hardware threads.

108. The method according to claim 103, wherein identifying the store instruction and the load instruction comprises identifying that the symbolic expressions in the store instruction and in the load instruction are defined in terms of one or more registers that are not written to between the store instruction and the load instruction.

5 109. The method according to claim 103, wherein a register that specifies the memory address in the store instruction and the load instruction comprises an incrementing index or a fixed calculation, such that multiple iterations of the store instruction and the load instruction access an array in the external memory.

10 110. The method according to claim 103, wherein assigning the outcome to be served from the internal memory comprises inhibiting the load instruction from being executed in the external memory.

111. The method according to claim 103, wherein assigning the outcome comprises providing the outcome from the internal memory only if the store instruction and the load instruction are associated with one or more specific flow-control traces.

15 112. The method according to claim 103, wherein assigning the outcome comprises providing the outcome from the internal memory regardless of a flow-control trace with which the store instruction and the load instruction are associated.

20 113. The method according to claim 103, wherein assigning the outcome comprises marking a location in the program code, to be modified for assigning the outcome, based on at least one parameter selected from a group of parameters consisting of Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the store instruction and the load instruction in the program code.

25 114. The method according to any of claims 103-113, wherein assigning the outcome comprises adding to the program code one or more instructions or micro-ops that serve the outcome, or modifying one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the outcome.

115. The method according to claim 114, wherein one of the added or modified instructions or micro-ops saves a value stored, or to be stored, by the store instruction to the internal memory.

30 116. The method according to claim 114, wherein adding or modifying the instructions or micro-ops is performed by a decoding unit or a renaming unit in a pipeline of the processor.

117. The method according to any of claims 103-113, wherein assigning the outcome to be served from the internal memory further comprises:

executing the load instruction in the external memory; and

5 verifying that the outcome of the load instruction executed in the external memory matches the outcome assigned to the load instruction from the internal memory.

118. The method according to claim 117, wherein verifying the outcome comprises comparing the outcome of the load instruction executed in the external memory to the outcome assigned to the load instruction from the internal memory.

119. The method according to claim 117, wherein verifying the outcome comprises verifying
10 that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

120. The method according to claim 117, wherein verifying the outcome comprises adding to the program code one or more instructions or micro-ops that verify the outcome, or modifying one or more existing instructions or micro-ops to the instructions or micro-ops that verify the
15 outcome.

121. The method according to claim 117, further comprising flushing subsequent code upon finding that the outcome executed in the external memory does not match the outcome served from the internal memory.

122. The method according to any of claims 103-113, further comprising inhibiting the load
20 instruction from being executed in the external memory.

123. The method according to any of claims 103-113, further comprising parallelizing execution of the program code, including assignment of the outcome from the internal memory, over multiple hardware threads.

124. The method according to any of claims 103-113, wherein processing the program code
25 comprises executing the program code, including assignment of the outcome from the internal memory, in a single hardware thread.

125. The method according to any of claims 103-113, wherein identifying at least the store instruction and the subsequent load instruction comprises identifying multiple subsequent load instructions that access the same memory address as the store instruction, and assigning the
30 outcome to be served to one or more instructions that depend on the multiple load instructions from the internal memory.

126. The method according to any of claims 103-113, wherein assigning the outcome comprises:

saving a value stored, or to be stored, by the store instruction in a physical register of the processor; and

5 renaming one or more instructions that depend on the outcome of the load instruction to receive the outcome from the physical register.

127. The method according to any of claims 103-113, wherein identifying the load instruction and the store instruction is performed, at least partly, based on indications embedded in the program code.

10 128. A processor, comprising:

an internal memory; and

processing circuitry, which is configured to process program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names, to identify at least a store instruction and a subsequent load instruction that
15 access the same memory address in the external memory, based on respective formats of the memory addresses specified in the symbolic expressions, and to assign an outcome of at least one of the memory-access instructions, to be served to one or more instructions that depend on the load instruction, from the internal memory.

20 129. The processor according to claim 128, wherein both the store instruction and the load instruction specify the memory address using the same symbolic expression.

130. The processor according to claim 128, wherein the store instruction and the load instruction specify the memory address using different symbolic expressions.

25 131. The processor according to claim 128, wherein both the store instruction and the load instruction are processed by the same hardware thread.

132. The processor according to claim 128, wherein the store instruction and the load instruction are processed by different hardware threads.

30 133. The processor according to claim 128, wherein the processing circuitry is configured to identify the store instruction and the load instruction by identifying that the symbolic expressions in the store instruction and in the load instruction are defined in terms of one or more registers that are not written to between the store instruction and the load instruction.

134. The processor according to claim 128, wherein a register that specifies the memory address in the store instruction and the load instruction comprises an incrementing index or a fixed calculation, such that multiple iterations of the store instruction and the load instruction access an array in the external memory.

135. The processor according to claim 128, wherein the processing circuitry is configured to inhibit the load instruction from being executed in the external memory.

136. The processor according to claim 128, wherein the processing circuitry is configured to assign the outcome from the internal memory only if the store instruction and the load instruction are associated with one or more specific flow-control traces.

137. The processor according to claim 128, wherein the processing circuitry is configured to assign the outcome from the internal memory regardless of a flow-control trace with which the store instruction and the load instruction are associated.

138. The processor according to claim 128, wherein the processing circuitry is configured to mark a location in the program code, to be modified for assigning the outcome, based on at least one parameter selected from a group of parameters consisting of Program-Counter (PC) values, program addresses, instruction-indices and address-operands of the store instruction and the load instruction in the program code.

139. The processor according to any of claims 128-138, wherein the processing circuitry is configured to add to the program code one or more instructions or micro-ops that serve the outcome, or to modify one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the outcome.

140. The processor according to claim 139, wherein one of the added or modified instructions or micro-ops saves a value stored, or to be stored, by the store instruction to the internal memory.

141. The processor according to claim 139, wherein the processing circuitry is configured to add or modify the instructions or micro-ops by a decoding unit or a renaming unit in a pipeline of the processor.

142. The processor according to any of claims 128-138, wherein the processing circuitry is configured to assign the outcome to be served from the internal memory by:

executing the load instruction in the external memory; and

verifying that the outcome of the load instruction executed in the external memory matches the outcome assigned to the load instruction from the internal memory.

143. The processor according to claim 142, wherein the processing circuitry is configured to verify the outcome by comparing the outcome of the load instruction executed in the external memory to the outcome assigned to the load instruction from the internal memory.

144. The processor according to claim 142, wherein the processing circuitry is configured to verify the outcome by verifying that no intervening event causes a mismatch between the outcome in the external memory and the outcome assigned from the internal memory.

145. The processor according to claim 142, wherein the processing circuitry is configured to add to the program code an instruction or micro-op that verifies the outcome, or to modify an existing instruction or micro-op to the instruction or micro-op that verifies the outcome.

146. The processor according to claim 142, wherein the processing circuitry is configured to flush subsequent code upon finding that the outcome executed in the external memory does not match the outcome served from the internal memory.

147. The processor according to any of claims 128-138, wherein the processing circuitry is configured to inhibit the load instruction from being executed in the external memory.

148. The processor according to any of claims 128-138, wherein the processing circuitry is configured to parallelize execution of the program code, including assignment of the outcome from the internal memory, over multiple hardware threads.

149. The processor according to any of claims 128-138, wherein the processing circuitry is configured to process the program code, including assignment of the outcome from the internal memory, in a single hardware thread.

150. The processor according to any of claims 128-138, wherein the processing circuitry is configured to identify multiple subsequent load instructions that access the same memory address as the store instruction, and to assign the outcome to be served to one or more instructions that depend on the multiple load instructions from the internal memory.

151. The processor according to any of claims 128-138, wherein the processing circuitry is configured to assign the outcome by:

saving a value stored, or to be stored, by the store instruction in a physical register of the processor; and

renaming one or more instructions that depend on the outcome of the load instruction to receive the outcome from the physical register.

152. The processor according to any of claims 128-138, wherein the processing circuitry is configured to identify the load instruction and the store instruction, at least partly based on indications embedded in the program code.

153. A method, comprising:

5 in a processor, processing program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names;

based on respective formats of the memory addresses specified in the symbolic expressions, identifying a repetitive sequence of instruction pairs, each pair comprising a store instruction and a subsequent load instruction that access the same respective memory address in
10 the external memory, wherein a value read by the load instruction of a first pair undergoes a predictable manipulation before the store instruction of a second pair that follows the first pair in the sequence;

saving the value read by the load instruction of the first pair in the internal memory;

15 applying the predictable manipulation to the value stored in the internal memory; and

assigning the manipulated value from the internal memory, to be served to one or more subsequent instructions that depend on the load instruction of the second pair.

154. The method according to claim 153, wherein identifying the repetitive sequence comprises identifying that the store instruction and the load instruction of a given pair access
20 the same memory address, by identifying that the symbolic expressions in the store instruction and in the load instruction of the given pair are defined in terms of one or more registers that are not written to between the store instruction and the load instruction of the given pair.

155. The method according to claim 153, wherein assigning the manipulated value comprises inhibiting the load instruction of the first pair from being executed in the external memory.

25 156. The method according to claim 153, wherein assigning the manipulated value comprises providing the manipulated value from the internal memory only if the first and second pairs are associated with one or more specific flow-control traces.

157. The method according to claim 153, wherein assigning the manipulated value comprises providing the manipulated value from the internal memory regardless of a flow-control trace
30 with which the first and second pairs are associated.

158. The method according to any of claims 153-157, wherein assigning the manipulated value comprises adding to the program code one or more instructions or micro-ops that serve

the manipulated value, or modifying one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the manipulated value.

159. The method according to claim 158, wherein one of the added instructions or micro-ops saves the value read by the load instruction of the first pair to the internal memory.

5 160. The method according to claim 158, wherein one of the added or modified instructions or micro-ops applies the predictable manipulation.

161. The method according to claim 158, wherein adding or modifying the instructions or micro-ops is performed by a decoding unit or a renaming unit in a pipeline of the processor.

10 162. The method according to any of claims 153-157, wherein assigning the manipulated value further comprises:

executing the load instruction of the first pair in the external memory; and
verifying that the outcome of the load instruction of the first pair executed in the external memory matches the manipulated value assigned from the internal memory.

15 163. The method according to claim 162, wherein verifying the outcome comprises comparing the outcome of the load instruction of the first pair executed in the external memory to the manipulated value assigned from the internal memory.

164. The method according to claim 162, wherein verifying the outcome comprises verifying that no intervening event causes a mismatch between the outcome in the external memory and the manipulated value assigned from the internal memory.

20 165. The method according to claim 162, wherein verifying the outcome comprises adding to the program code one or more instructions or micro-ops that verify the outcome, or modifying one or more existing instructions or micro-ops to the instructions or micro-ops that verify the outcome.

25 166. The method according to any of claims 153-157, wherein assigning the manipulated value comprises:

saving the value read by the load instruction of the first pair in a physical register of the processor; and

renaming one or more instructions that depend on the load instruction of the second pair to receive the outcome from the physical register.

30 167. The method according to any of claims 153-157, wherein assigning the manipulated value comprises applying the predictable manipulation multiple times, so as to save in the

internal memory multiple different manipulated values corresponding to multiple future pairs in the sequence, and providing each of the multiple manipulated values from the internal memory to the one or more instructions that depend on the load instruction of a corresponding future pair.

5 168. The method according to any of claims 153-157, wherein identifying the repetitive sequence is performed, at least partly, based on indications embedded in the program code.

169. A processor, comprising:

an internal memory; and

10 processing circuitry, which is configured to process program code that includes memory-access instructions, wherein at least some of the memory-access instructions comprise symbolic expressions that specify memory addresses in an external memory in terms of one or more register names, to identify, based on respective formats of the memory addresses specified in the symbolic expressions, a repetitive sequence of instruction pairs, each pair comprising a store instruction and a subsequent load instruction that access the same respective memory address in
15 the external memory, wherein a value read by the load instruction of a first pair undergoes a predictable manipulation before the store instruction of a second pair that follows the first pair in the sequence, to save the value read by the load instruction of the first pair in the internal memory, to apply the predictable manipulation to the value stored in the internal memory, and to assign the manipulated value from the internal memory, to be served to one or more
20 subsequent instructions that depend on the load instruction of the second pair.

170. The processor according to claim 169, wherein the processing circuitry is configured to identify that the store instruction and the load instruction of a given pair access the same memory address, by identifying that the symbolic expressions in the store instruction and in the load instruction of the given pair are defined in terms of one or more registers that are not written to
25 between the store instruction and the load instruction of the given pair.

171. The processor according to claim 169, wherein the processing circuitry is configured to inhibit the load instruction of the first pair from being executed in the external memory.

172. The processor according to claim 169, wherein the processing circuitry is configured to assign the outcome from the internal memory only if the first and second pairs are associated
30 with one or more specific flow-control traces.

173. The processor according to claim 169, wherein the processing circuitry is configured to assign the outcome from the internal memory regardless of a flow-control trace with which the first and second pairs are associated.

174. The processor according to any of claims 169-173, wherein the processing circuitry is configured to add to the program code one or more instructions or micro-ops that serve the outcome, or to modify one or more existing instructions or micro-ops to the one or more instructions or micro-ops that serve the outcome.

175. The processor according to claim 174, wherein one of the added instructions or micro-ops saves the value read by the load instruction of the first pair to the internal memory.

176. The processor according to claim 174, wherein one of the added or modified instructions or micro-ops applies the predictable manipulation.

177. The processor according to claim 174, wherein the processing circuitry is configured to add or modify the instructions or micro-ops by a decoding unit or a renaming unit in a pipeline of the processor.

178. The processor according to any of claims 169-173, wherein the processing circuitry is configured to assign the outcome to be served from the internal memory by:

executing the load instruction of the first pair in the external memory; and
verifying that the outcome of the load instruction of the first pair executed in the external memory matches the manipulated value assigned from the internal memory.

179. The processor according to claim 178, wherein the processing circuitry is configured to verify the outcome by comparing the outcome of the load instruction of the first pair executed in the external memory to the manipulated value assigned from the internal memory.

180. The processor according to claim 178, wherein the processing circuitry is configured to verify the outcome by verifying that no intervening event causes a mismatch between the outcome in the external memory and the manipulated value assigned from the internal memory.

181. The processor according to claim 178, wherein the processing circuitry is configured to add to the program code an instruction or micro-op that verifies the outcome, or to modify an existing instruction or micro-op to the instruction or micro-op that verifies the outcome.

182. The processor according to any of claims 169-173, wherein the processing circuitry is configured to assign the outcome by:

saving the value read by the load instruction of the first pair in a physical register of the processor; and

renaming one or more instructions that depend on the load instruction of the second pair to receive the outcome from the physical register.

- 5 183. The processor according to any of claims 169-173, wherein the processing circuitry is configured to assign the outcome by applying the predictable manipulation multiple times, so as to save in the internal memory multiple different manipulated values corresponding to multiple future pairs in the sequence, and providing each of the multiple manipulated values from the internal memory to the one or more instructions that depend on the load instruction of
10 a corresponding future pair.

184. The processor according to any of claims 169-173, wherein the processing circuitry is configured to identify the repetitive sequence, at least partly based on indications embedded in the program code.

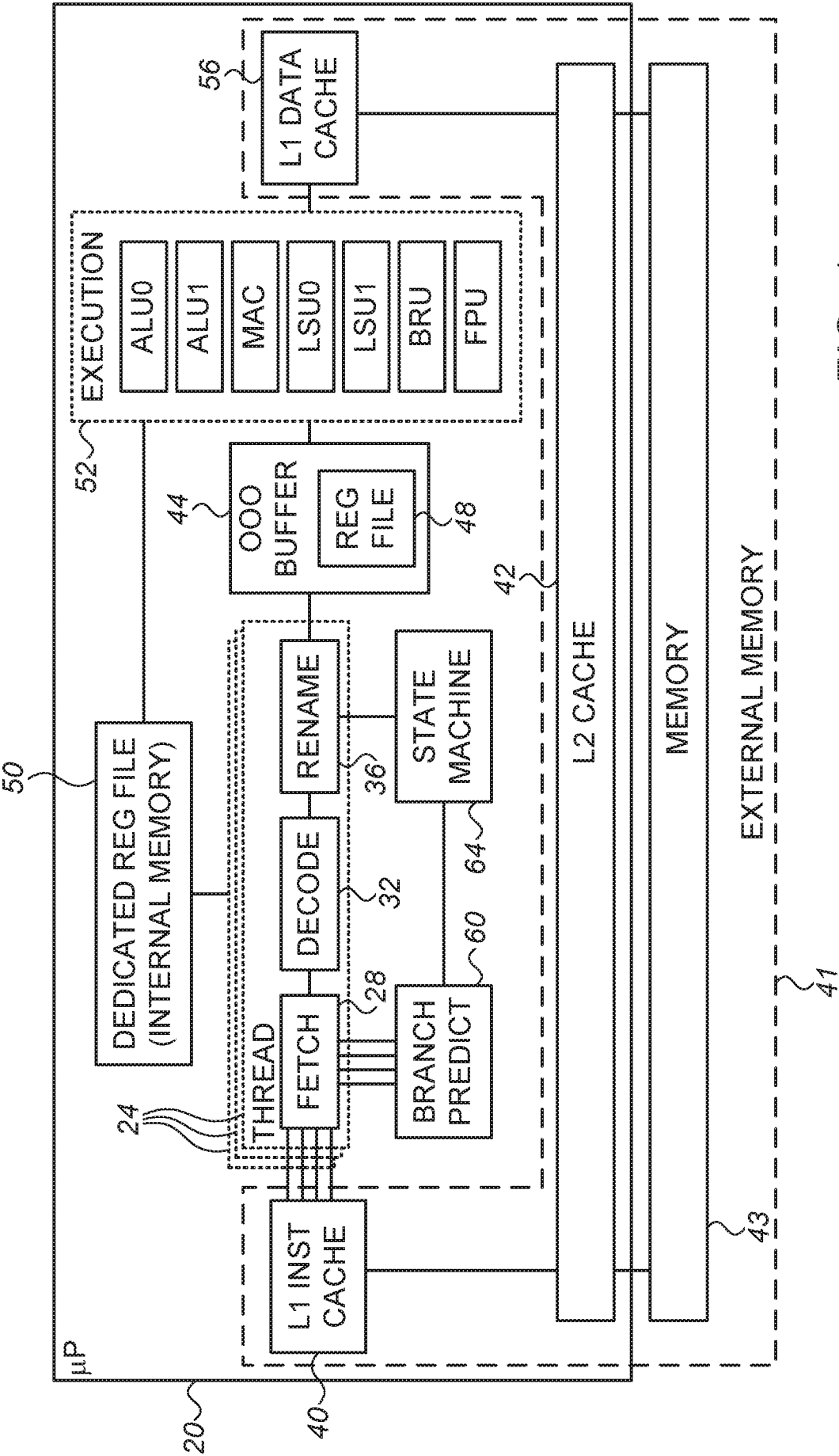


FIG. 1

2/5

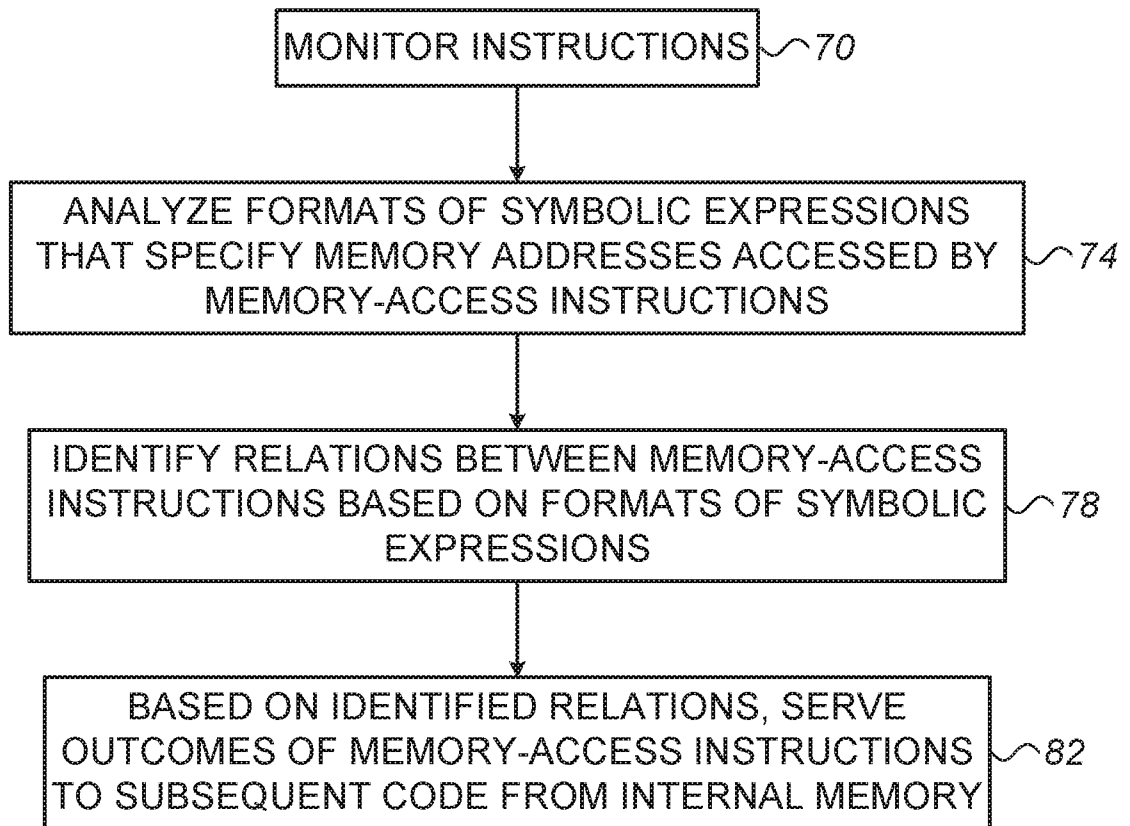


FIG. 2

3/5

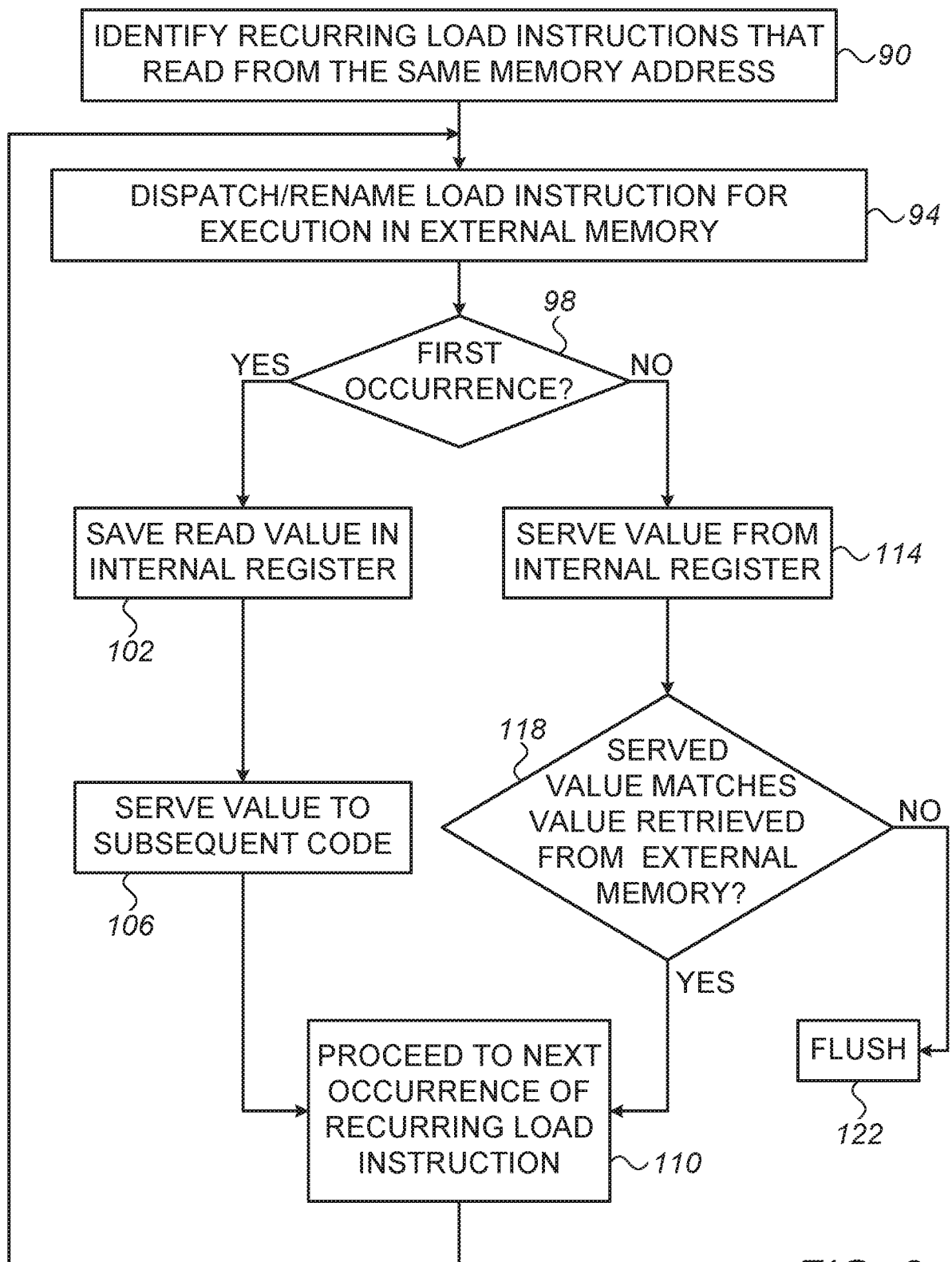
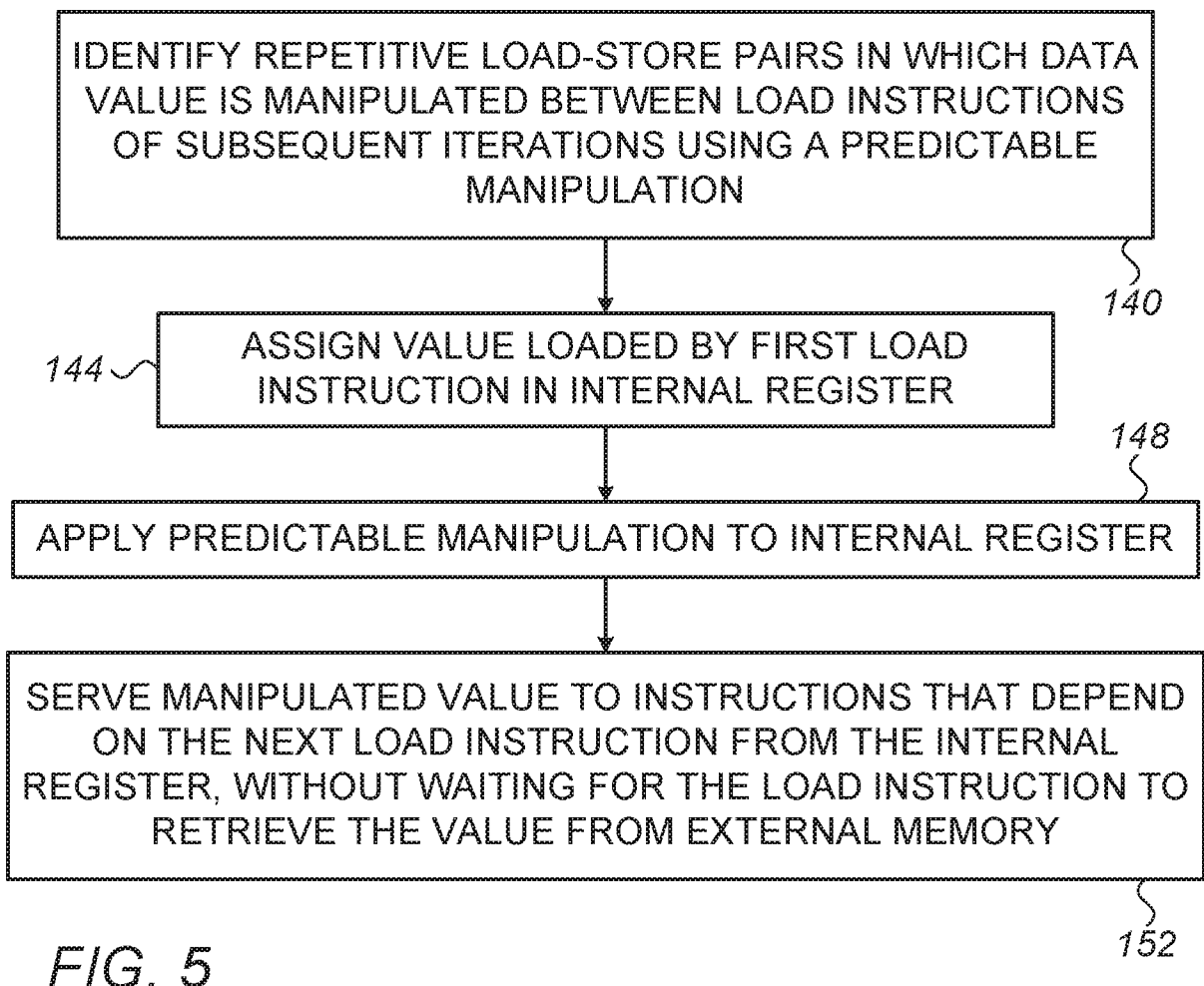
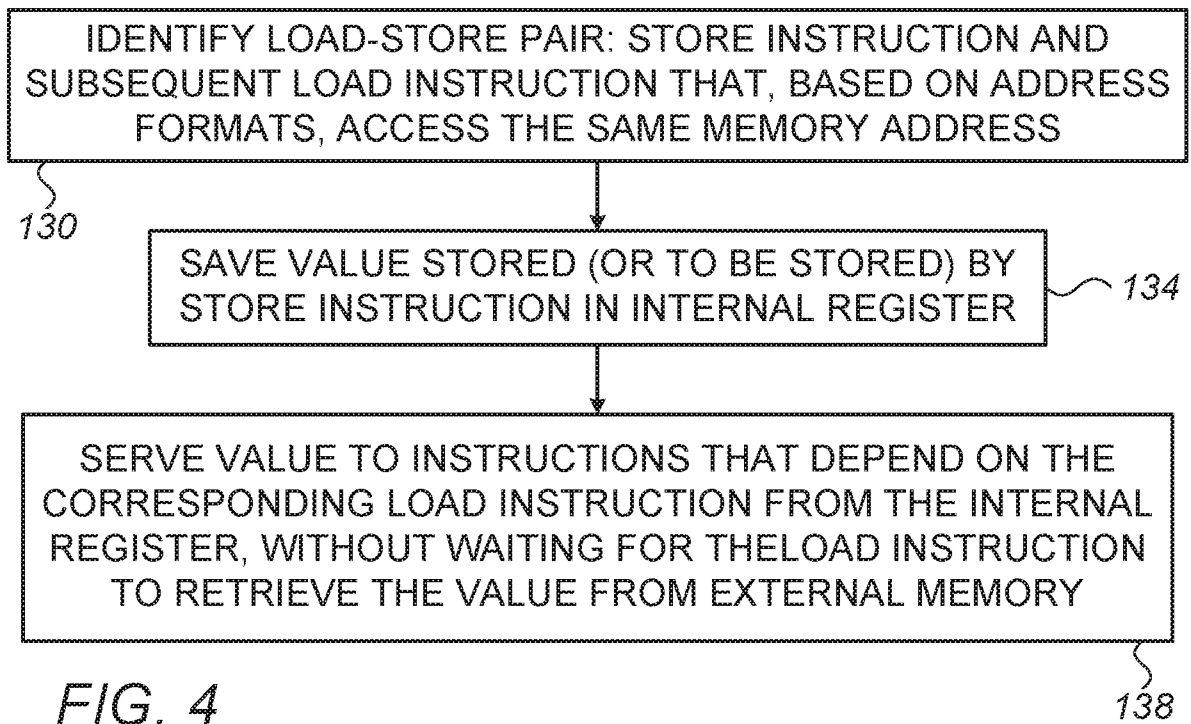


FIG. 3

4/5



5/5

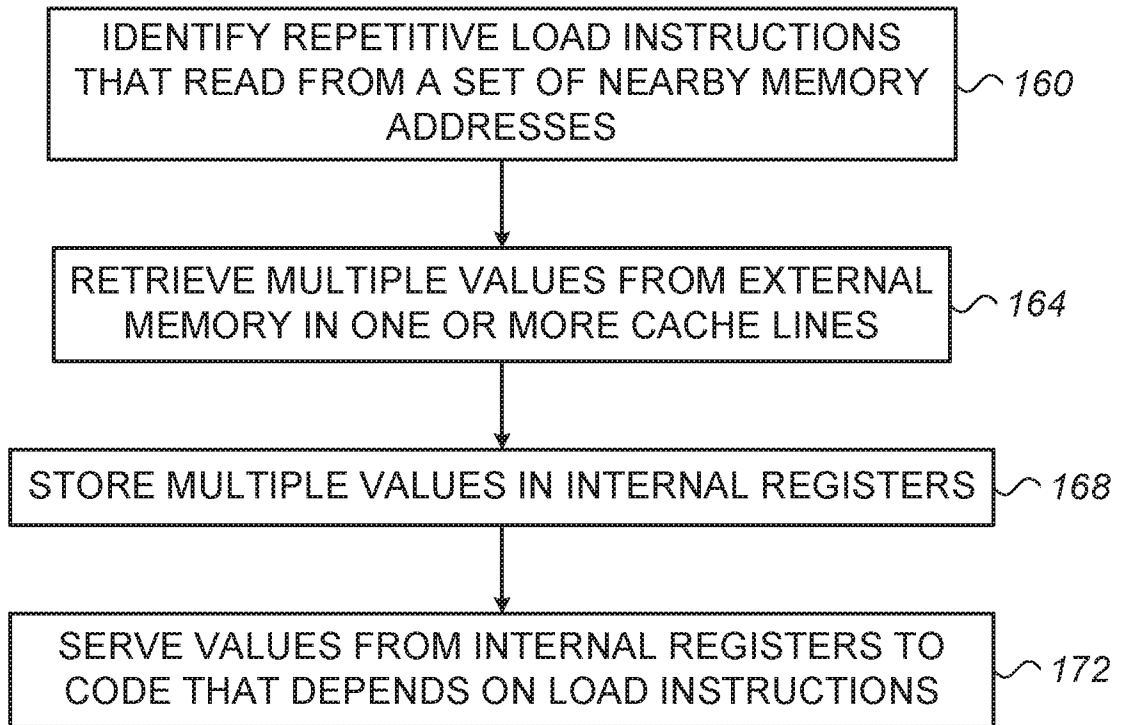


FIG. 6

INTERNATIONAL SEARCH REPORT

International application No.

PCT/IB2016/053999

A. CLASSIFICATION OF SUBJECT MATTER

IPC (2016.01) G06F 9/38, G06F 9/34

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC (2016.01) G06F 9/38, G06F 9/34

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)

Databases consulted: Google Scholar, PatBase

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EP 1008050 B1 28 Feb 2007 (2007/02/28) par. 0115, 0092	1-30



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

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

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

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

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

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

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

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

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

“&” document member of the same patent family

Date of the actual completion of the international search

13 Oct 2016

Date of mailing of the international search report

13 Oct 2016

Name and mailing address of the ISA:

Israel Patent Office

Technology Park, Bldg.5, Malcha, Jerusalem, 9695101, Israel

Facsimile No. 972-2-5651616

Authorized officer

RUSS Eran

Telephone No. 972-2-5651701

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/IB2016/053999

Patent document cited search report			Publication date	Patent family member(s)		Publication Date
EP 1008050 B1			28 Feb 2007	EP	1008050 A2	14 Jun 2000
				EP	1008050 A4	16 Aug 2001
				EP	1008050 B1	28 Feb 2007
				AT	355558 T	15 Mar 2006
				CA	2262928 C	30 Jan 2001
				CN	1241272 A	12 Jan 2000
				CN	1141647 C	10 Mar 2004
				DE	69737423 D1	12 Apr 2007
				DE	69737423 T2	08 Nov 2007
				JP	2001504957 A	10 Apr 2001
				JP	3753743 B2	08 Mar 2006
				KR	20000048630 A	25 Jul 2000
				KR	100385426 B1	27 May 2003
				US	5926832 A	20 Jul 1999
				WO	9813740 A2	02 Apr 1998
				WO	9813740 A3	02 Jul 1998