

(19) World Intellectual Property Organization  
International Bureau(43) International Publication Date  
27 September 2007 (27.09.2007)

PCT

(10) International Publication Number  
WO 2007/109631 A2

## (51) International Patent Classification:

G06F 9/38 (2006.01)

## (21) International Application Number:

PCT/US2007/064331

## (22) International Filing Date: 19 March 2007 (19.03.2007)

## (25) Filing Language: English

## (26) Publication Language: English

## (30) Priority Data:

11/378,712 17 March 2006 (17.03.2006) US

(71) Applicant (for all designated States except US): QUALCOMM INCORPORATED [US/US]; Attn: International IP Administration, 5775 Morehouse Drive, San Diego, California 92121 (US).

## (72) Inventors; and

(75) Inventors/Applicants (for US only): DIEFFENDER-FER, James Norris [US/US]; 4000 Inkberry Court, Apex, North Carolina 27539 (US). RYCHLIK, Bohuslav [US/US]; 1017 Sweet Spot Circle, Morrisville, North Carolina 27560 (US).

(74) Agents: OGROD, Gregory D. et al.; Attn: International IP Administration, 5775 Morehouse Drive, San Diego, California 92121 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM,

AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, 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, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

## Declarations under Rule 4.17:

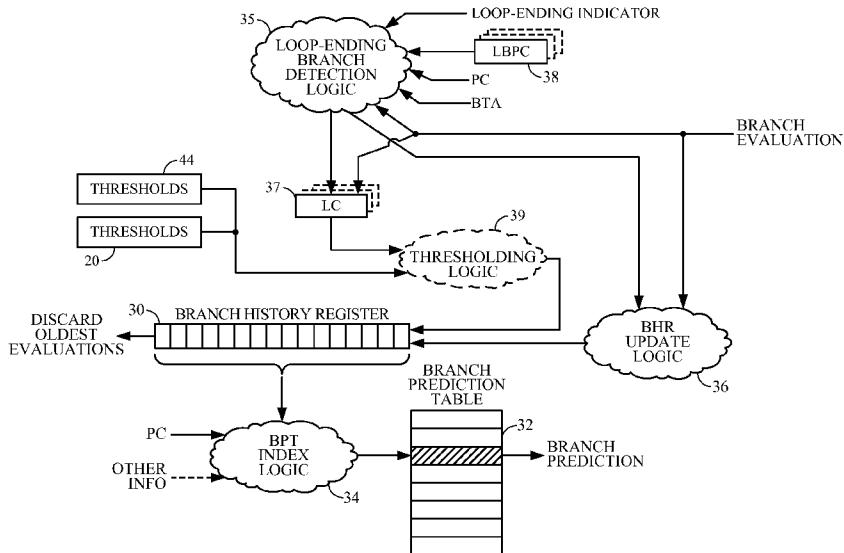
- as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))
- as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))

## Published:

- without international search report and to be republished upon receipt of that report

[Continued on next page]

## (54) Title: REPRESENTING LOOP BRANCHES IN A BRANCH HISTORY REGISTER WITH MULTIPLE BITS



(57) **Abstract:** In response to a property of a conditional branch instruction associated with a loop, such as a property indicating that the branch is a loop-ending branch, a count of the number of iterations of the loop is maintained, and a multi-bit value indicative of the loop iteration count is stored in a Branch History Register (BHR). In one embodiment, the multi-bit value may comprise the actual loop count, in which case the number of bits is variable. In another embodiment, the number of bits is fixed (e.g., two) and loop iteration counts are mapped to one of a fixed number of multi-bit values (e.g., four) by comparison to thresholds. Separate iteration counts may be maintained for nested loops, and a multi-bit value stored in the BHR may indicate a loop iteration count of only an inner loop, only the outer loop, or both.

WO 2007/109631 A2



*For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.*

## REPRESENTING LOOP BRANCHES IN A BRANCH HISTORY REGISTER WITH MULTIPLE BITS

### FIELD

**[0001]** The present disclosure relates generally to the field of processors and in particular to a method of representing loop branches in a branch history register with multiple bits.

### BACKGROUND

**[0002]** Microprocessors perform computational tasks in a wide variety of applications. Improved processor performance is almost always desirable, to allow for faster operation and/or increased functionality through software changes. In common embedded applications, such as portable electronic devices, conserving power is also desirable.

**[0003]** Common modern processors employ a pipelined architecture, where sequential instructions, each having multiple execution steps, are overlapped in execution. For maximum performance, the instructions should flow continuously through the pipeline. Any situation that causes instructions to stall in the pipeline detrimentally affects performance. If instructions must be flushed from the pipeline and subsequently re-fetched, both performance and power consumption may suffer.

**[0004]** Commonly all real-world programs include conditional branch instructions, the actual branching behavior of which is commonly not known until the instruction is evaluated deep in the pipeline. To avoid a stall that may result from waiting for actual evaluation of the branch instruction, common modern processors employ some form of branch prediction, whereby the branching behavior of conditional branch instructions is predicted early in the pipeline. Based on the predicted branch evaluation, the processor speculatively fetches (prefetches) and executes instructions from a predicted address – either the branch target address (if the branch is predicted taken) or the next sequential address after the branch instruction (if the branch is predicted not taken).

When the actual branch behavior is determined, if the branch was mispredicted, the speculatively fetched instructions must be flushed from the pipeline, and new instructions fetched from the correct next address. Prefetching instructions in response to an erroneous branch prediction adversely impacts processor performance and power consumption. Consequently, improving the accuracy of branch prediction is desirable.

**[0005]** Known branch prediction techniques include both static and dynamic predictions. The likely behavior of some branch instructions can be statically predicted by a programmer and/or compiler. One example is an error checking routine. Common code executes properly, and errors are rare. Hence, the branch instruction implementing a “branch on error” function will evaluate “not taken” a very high percentage of the time. Such an instruction may include a static branch prediction bit in the op code, set by a programmer or compiler with knowledge of the common likely outcome of the branch condition.

**[0006]** Dynamic prediction is generally based on the branch evaluation history (and in some cases the branch prediction accuracy history) of the branch instruction being predicted and/or other branch instructions in the same code. Extensive analysis of actual code indicates that recent past branch evaluation patterns may be a good indicator of the evaluation of future branch instructions.

**[0007]** One known form of dynamic branch prediction, depicted in Figure 1, utilizes a Branch History Register (BHR) 100 to store the past  $n$  branch evaluations. In a simple implementation, the BHR 30 comprises a shift register. The common recent branch evaluation result is shifted in (for example, a 1 indicating taken and a 0 indicating not taken), with the oldest past evaluation in the register being displaced. A processor may maintain a local BHR 100 for each branch instruction. Alternatively (or additionally), a BHR 100 may contain the recent past evaluations of all conditional

branch instructions, sometimes known in the art as a global BHR, or GHR. As used herein, BHR refers to both local and global Branch History Registers.

**[0008]** As depicted in Figure 1, the BHR 100 may index a Branch Predictor Table (BPT) 102, which again may be local or global. The BHR 100 may index the BPT 102 directly, or may be combined with other information, such as the Program Counter (PC) of the branch instruction in BPT index logic 104. Other inputs to the BPT index logic 104 may additionally be utilized. The BPT index logic 104 may concatenate the inputs (known in the art as gselect), XOR the inputs (gshare), perform a hash function, or combine or transform the inputs in a variety of ways.

**[0009]** As one example, the BPT 102 may comprise a plurality of saturation counters, the MSBs of which serve as bimodal branch predictors. For example, each table entry may comprise a 2-bit counter that assumes one of four states, each assigned a weighted prediction value, such as:

**[0010]** 11 – Strongly predicted taken

**[0011]** 10 – Weakly predicted taken

**[0012]** 01 – Weakly predicted not taken

**[0013]** 00 – Strongly predicted not taken

**[0014]** The counter increments each time a corresponding branch instruction evaluates “taken” and decrements each time the instruction evaluates “not taken.” The MSB of the counter is a bimodal branch predictor; it will predict a branch to be either taken or not taken, regardless of the strength or weight of the underlying prediction. A saturation counter reduces the prediction error of an infrequent branch evaluation direction. A branch that consistently evaluates one way will saturate the counter. An infrequent evaluation the other way will alter the counter value (and the strength of the prediction), but not the bimodal prediction value. Thus, an infrequent evaluation will only mispredict once, not twice. The table of saturation counters is an illustrative

example only; in general, a BHT may index a table containing a variety of branch prediction mechanisms.

**[0015]** Regardless of the branch prediction mechanism employed in the BPT 102, the BHR 100 – either alone or in combination with other information such as the branch instruction PC – indexes the BPT 102 to obtain branch predictions. By storing prior branch evaluations in the BHR 100 and using the evaluations in branch prediction, the branch instruction being predicted is correlated to past branch behavior – its own past behavior in the case of a local BHR 100 and the behavior of other branch instructions in the case of a global BHR 100. This correlation is the key to accurate branch predictions, at least in the case of highly repetitive code.

**[0016]** Note that Figure 1 depicts branch evaluations being stored in the BHR 100 – that is, the actual evaluation of a conditional branch instruction, which may only be known deep in the pipeline, such as in an execute pipe stage. While this is the ultimate result, in practice, common high performance processors store the predicted branch evaluation from the BPT 102 in the BHR 100, and correct the BHR 100 later as part of a misprediction recovery operation if the prediction turns out to be erroneous. The drawing figures do not reflect this implementation feature, for clarity.

**[0017]** A common code structure that may reduce the efficacy a branch predictor employing a BHR 100 is the loop. A loop ends with a conditional branch instruction that tests a loop-ending condition, such as whether an index variable that is incremented each time through the loop has reached a loop-ending value. If not, execution branches back to the beginning of the loop for another iteration, and another loop-ending conditional branch evaluation.

**[0018]** If the loop executes through a large number of iterations, the “taken” backwards branches of the loop-ending branch instruction partially or fully saturate the BHR 100. Where the number of loop iterations equals or exceeds the BHR 100 width, at the end of the loop an  $n$ -bit BHR will contain precisely  $n-1$  ones (taken) followed by a

single zero (not taken), corresponding to a long series of branch taken evaluations resulting from the loop iterations, and ending with a single not-taken branch evaluation when the loop terminates. This effectively destroys the efficacy of the BHR 100, as all correlations with prior branch evaluations (for either a local or global BHR 100) are lost. In this case, the BHR 100 will likely map to the same BPT 102 entry for a given branch instruction (depending on the other inputs to the BPT index logic 104), rather than to an entry containing a branch prediction that reflects the correlation of the branch instruction to prior branch evaluations.

**[0019]** Additionally, the saturated BHR 100 may increase aliasing in the BPT 102. That is, all branch instructions following loops with common iterations will map to the same BPT 102 entry, if the BHR 100 directly indexes the BPT 102. Even where the BHR 100 is combined with other information, the chance of aliasing is increased. This adversely impacts prediction accuracy not only for the branch instruction following the loop, but also for all of the branch instructions that alias to its entry in the BPT 102.

**[0020]** If the loop executes through fewer iterations than the width of the BHR 100, the BHR 100 is not saturated and some prior branch evaluation history is retained. However, the bits representing the prior branch evaluation history are displaced in the BHR 100 by numerous “taken” results of the loop-ending branch instruction. Particularly where the number of loop iterations varies, this has two deleterious effects on branch prediction. First, the branch instruction will map to a much larger number of entries in the BPT 102 to capture the same correlation with prior branch evaluations, requiring a larger BPT 102 to support the same accuracy for the same number of branch instructions than would be required without the loop-ending branch affecting the BHR 30. Second, the branch predictors in the BPT 102 will take longer to “train,” increasing the amount of code that must execute before the BPT 102 begins to provide accurate branch predictions.

**[0021]** Patent application Serial No. 11/066,508, assigned to the assignee of the present application and incorporated herein by reference, proposes to suppress updating the BHR 100 for loop-ending branch instructions. This resolves common of the deleterious effects of BHR 100 saturation or partial saturation on branch prediction accuracy. However, it fails to capture and exploit correlations that may exist between loop behavior and subsequent branch evaluation.

**[0022]** In common applications, the evaluation of a branch instruction may be correlated to the number of iterations of a preceding loop. For example, a scientific program may capture data points in a loop and, following the loop, branch to a statistical analysis subroutine where the captured data is analyzed. However, if the loop iterates relatively few times, capturing few data points and yielding a small sample, statistical analysis may be unreliable, and may be skipped. In this case, the evaluation of the conditional instruction branching to the statistical analysis subroutine is strongly correlated to the number of iterations of the data acquisition loop.

**[0023]** In another example, an application may use a loop structure to search through a list, transaction log, history file, or similar data structure. If an item matching the search parameters appears frequently in the list, relatively few loop iterations will be required to locate the item. Consequently, an item appearing infrequently may require a large number of iterations through the search loop. The evaluation of a subsequent branch instruction may be correlated to the frequency with which a particular item appears in the list, and hence correlated to the number of loop iterations required to locate the item.

**[0024]** Suppressing the update of the BHR 100 in response to loop-ending branch instruction evaluations fails to capture any correlation between the number of loop iterations and the branch behavior of a subsequent branch instruction. On the other hand, maintaining a full history of the evaluations of the loop-ending branch instruction

fully or partially saturates the BHR 100, losing the correlation to branch instruction evaluations prior to the loop.

## SUMMARY

**[0025]** In response to a property of a conditional branch instruction associated with a loop, such as a property indicating that the branch is a loop-ending branch, a count of the number of iterations of the loop is maintained, and a multi-bit value indicative of the loop iteration count is stored in the BHR. In one embodiment, the multi-bit value may comprise the actual loop count, in which case the number of bits is variable. In another embodiment, the number of bits is fixed (e.g., two) and loop iteration counts are mapped to a multi-bit value by comparison to thresholds. Separate iteration counts may be maintained for nested loops, and a multi-bit value stored in the BHR may indicate a loop iteration count of only an inner loop, only the outer loop, or both.

**[0026]** One embodiment relates to a method of branch prediction. A property of a branch instruction associated with a loop is identified. In response to the property, a multi-bit value is stored in a BHR, the multi-bit value indicative of the number of iterations of a loop associated with the branch instruction.

**[0027]** Another embodiment relates to a processor. The processor includes a branch predictor operative to predict the evaluation of conditional branch instructions, and an instruction execution pipeline operative to speculatively fetch and execute instructions based on a prediction from the branch predictor. The processor also includes a BHR operative to store the evaluation of conditional branch instructions, and a loop counter operative to count the number of iterations of a code loop. The processor further includes a control circuit operative to store in the BHR a multi-bit value indicative of the number of iterations of a loop associated with a conditional branch instruction.

## BRIEF DESCRIPTION OF DRAWINGS

**[0028]** Figure 1 is a functional block diagram of a prior art branch predictor circuit.

**[0029]** Figure 2 is a functional block diagram of a processor.

**[0030]** Figure 3 is a flow diagram of a method of executing a branch instruction.

**[0031]** Figure 4 is a functional block diagram of a branch predictor circuit including one or more Last Branch PC registers and Loop Counters.

## DETAILED DESCRIPTION

**[0032]** Figure 2 depicts a functional block diagram of a processor 10. The processor 10 executes instructions in an instruction execution pipeline 12 according to control logic 14. In some embodiments, the pipeline 12 may be a superscalar design, with multiple pipelines. The pipeline 12 includes various registers or latches 16, organized in pipe stages, and one or more Arithmetic Logic Units (ALU) 18. A General Purpose Register (GPR) file 20 provides registers comprising the top of the memory hierarchy.

**[0033]** The pipeline 12 fetches instructions from an instruction cache (I-cache) 22, with memory address translation and permissions managed by an Instruction-side Translation Lookaside Buffer (ITLB) 24. When conditional branch instructions are decoded early in the pipeline 12, a branch predictor 26 predicts the branch behavior, and provides the prediction to an instruction prefetch unit 28. The instruction prefetch unit 28 speculatively fetches instructions from the I-cache 22, at a branch target address calculated in the pipeline 12 for “taken” branch predictions, or at the next sequential address for branches predicted “not taken.” In either case, the prefetched instructions are loaded into the pipeline 12 for speculative execution.

**[0034]** The branch predictor 26 includes a Branch History Register (BHR) 30, a Branch Predictor Table (BPT) 32, BPT index logic 34, and BHR update logic 36. The

branch predictor 26 may additionally include one or more Last Branch PC registers 38, and one or more loop counters 37 providing inputs to the BHR update logic 36.

**[0035]** Data is accessed from a data cache (D-cache) 40, with memory address translation and permissions managed by a main Translation Lookaside Buffer (TLB) 42. The TLB 42 may be a dedicated data TLB, or may comprise an integrated TLB that manages address translations and permissions for both instructions and data. Additionally, in various embodiments of the processor 10, the I-cache 22 and D-cache 40 may be integrated, or unified. Misses in the I-cache 22 and/or the D-cache 40 cause an access to main (off-chip) memory 44, under the control of a memory interface 46.

**[0036]** The processor 10 may include an Input/Output (I/O) interface 48, controlling access to various peripheral devices 50, 52. Those of skill in the art will recognize that numerous variations of the processor 10 are possible. For example, the processor 10 may include a second-level (L2) cache for either or both the I and D caches 22, 40. In addition, one or more of the functional blocks depicted in the processor 10 may be omitted from a particular embodiment.

**[0037]** Multi-Bit Indications of Loop Iterations

**[0038]** According to one or more embodiments, branch prediction accuracy is improved by correlating branch evaluation to both previous branch evaluations and the iteration count of previous loops. This is accomplished by storing a multi-bit value indicative of loop iteration count(s) in the BHR 30. By mapping potentially large loop iteration counts to relatively few bits, characteristic information about the loop is preserved while preventing the loop-ending branches from fully or partially saturating one or more BHRs 30 in the branch predictor 26.

**[0039]** This process is depicted as a flow diagram in Figure 3. An instruction is fetched and decoded (block 52). If the instruction is a not conditional branch instruction (block 53), it proceeds in the pipeline and the process returns to fetch the

next instruction (block 52). If the instruction is a conditional branch instruction (block 53), a determination is made whether the branch is a loop-ending branch (block 54). If not, when the branch evaluates (such as in an execute stage of the pipeline) the BHR 30 is updated to record the branch evaluation using a single bit (block 56), *i.e.*, whether the branch instruction evaluated as “taken” or “not taken.” Execution then continues (block 64) at the branch target address or the next sequential address, respectively. If the branch is a loop-ending branch (block 54), a loop iteration count is incremented (block 58). When the loop completes (block 60), a multi-bit value indicative of the number of loop iterations is written to the BHR 30 (block 62). The loop iteration counter is then cleared (block 63) and execution continues (block 64) at the next sequential address.

**[0040]** Both the query (block 54) – that is, identifying a branch instruction as a loop-ending branch instruction – and maintaining the loop iteration count (block 58) may be accomplished in a variety of ways. In one embodiment, one or more Loop Counters (LC) 37 are incremented when a conditional branch instruction that is determined to be a loop-ending branch evaluates “taken.” The value of the LC 37, or a value derived therefrom (as discussed more fully herein), is written to the BHR 30 when the loop-ending branch evaluates “not taken,” indicating an exit of the loop. The LC 37 is also cleared at that time.

**[0041]** Identifying Loop-Ending Branch Instructions

**[0042]** Loops iterate by branching backwards from the end of the loop to the beginning of the loop. According to one embodiment, every conditional branch instruction with a branch target address less than the branch instruction address, or PC – that is, a backwards branch – is assumed to be a loop-ending branch instruction. This method of identifying Loop-Ending branches offers simplicity. As depicted in Figure 4, the branch instruction PC is compared to the branch target address (BTA) in Loop-Ending Branch Detection logic 35 when the branch instruction is actually

evaluated in the pipeline. If BTA < PC, a Loop Counter (LC) 37 is incremented. This embodiment requires an address comparison when the branch target address is determined, and may suffer from some inaccuracy as not all backward branches are loop-ending branches.

**[0043]** Another way to detect a loop-ending branch is to recognize repeated execution of the same branch instruction. In one embodiment, as depicted in Figure 4, a Last Branch PC (LBPC) register 38 stores the PC of the last branch instruction executed. In the case of a simple loop, if the PC of a branch instruction matches the LBPC 38 in the Loop-Ending Branch Detection logic 35 – that is, the branch instruction was the last branch instruction evaluated – the branch instruction is assumed to be a loop-ending branch instruction, and a LC 37 is incremented.

**[0044]** Code that includes conditional branch instructions within a loop can complicate loop-ending branch detection. For example, common algorithms for searching an array or linked list create a loop. A branch within the loop checks for a match and exits the loop early if a match is found, while a loop-ending branch checks for the end of the array or linked list. In this case, a single LBPC 38 cannot detect the loop-ending branch, because it will be alternately overwritten by the two branch PCs. Accordingly, in one embodiment, two or more LBPC registers 38 may be provided (as depicted in Fig. 4), with the PCs of successively evaluated branch instructions stored in corresponding LBPC registers (LBPC<sub>0</sub>, LBPC<sub>1</sub>, ... LBPC<sub>M</sub>) 38. This allows for the detection of loop-ending branches even in the presence of other conditional branches within the loop.

**[0045]** Loop-ending branch instructions may also be statically marked by a compiler or assembler. In one embodiment, a compiler generates a particular type of branch instruction that is only used for loop-ending branches, for example, “BRLP”. The BRLP instruction is recognized, and a LC 37 is incremented every time a BRLP instruction evaluates as taken in an execution pipe stage. In another embodiment, a

compiler or assembler may embed a loop-ending branch indication in a branch instruction, such as by setting one or more predefined bits in the instruction op code. The loop-ending branch bits are detected, and a LC 37 is incremented when that branch instruction evaluates as taken in an execute pipe stage. In general, a Loop-Ending Indicator may be provided to the Loop-Ending Branch Detection logic 35. Static identification of loop-ending branches reduces hardware and computational complexity by moving the loop-ending identification function into the compiler or assembler.

**[0046]** Generating and Storing an Indicator of the Number of Loop Iterations

**[0047]** Regardless of the technique used to identify a loop-ending branch instruction (for either an inner or outer loop), according to one or more embodiments, an indication of the number of times the loop iterates is stored in the BHR 30, to exploit any correlation between the number of loop iterations and subsequent conditional branch instruction evaluation.

**[0048]** In one embodiment, the actual count of loop iterations is stored in the BHR 30. That is, the multi-bit count from, e.g., a LC 37 is shifted into the BHR 30, displacing prior branch evaluations in the MSBs of the BHR 30. For very large loops, this count will require  $\log_2(n)$  bits for  $n$  loop iterations, displacing a corresponding number of prior branch evaluations. However, even for large loop iteration counts, the binary representation is a significant compression of the loop-ending branch behavior stored in prior art branch history registers (i.e.,  $n-1$  ones followed by a single zero). In one embodiment, the iteration count is accumulated in a LC 37 and shifted into the BHR 30. In another embodiment (not shown), at least the lower  $k$  bits of the BHR 30 may be implemented as a dual-mode shift-register/counter, and the loop iteration count may be incremented directly in the BHR 30 (with prior BHR 30 contents shifting left as the count grows).

**[0049]** In another embodiment, a fixed, multi-bit indication of the number of loop iterations is stored in the in the BHR 30. For example, two bits may be allocated to record an indication of the number of loop iterations, with the following meanings:

**[0050]** 11 – Very large number of loop iterations

**[0051]** 10 – Large number of loop iterations

**[0052]** 01 – Moderate number of loop iterations

**[0053]** 00 – Small number of loop iterations

**[0054]** In this example, three threshold values – between small and moderate (00 - > 01); between moderate and large (01 -> 10), and between large and very large (10 -> 11) – are used to map the actual loop iteration count to a two-bit representation, as indicated by Thresholding logic 39. In one embodiment, the threshold values may be fixed for a given processor implementation, such as for certain embedded applications where common loop sizes are generally known. In another embodiment, the mapping may be scalable, with the threshold values read from GPRs 20, a table in memory 44, or the like. In common applications, a fixed multi-bit indication of the number of loop iterations may provide sufficient indication of the loop iteration count to allow for meaningful correlation with branch evaluation behavior, while at the same time limiting the number of prior branch evaluations displaced from the BHR 30 by loop-ending branches.

**[0055]** Of course, actual loop iteration counts may be mapped to any fixed number of bits, with the corresponding number of threshold values, e.g., three bits and seven thresholds; four bits and fifteen thresholds; etc., to achieve the desired or required balance between loop iteration count accuracy and branch evaluation displacements in the BHR 30.

**[0056]** Nested Loops

**[0057]** A loop may contain one or more nested, or inner, loops. Accordingly, in one embodiment, a plurality of LC counters ( $LC_0, LC_1, \dots, LC_M$ ) 37 are provided, which may

correspond to the number of LBPC 38 registers. A LC 37 is incremented upon a taken evaluation and a match with the relevant LBPC<sub>M</sub> 38, to maintain the iteration counts of nested loops.

**[0058]** By storing a multi-bit indication of the number of iterations of an inner loop every time the inner loop exits, the BHR 30 may become partially or completely saturated by the time the outer loop iterates relatively few times. Accordingly, in one embodiment, once a loop is determined to be an inner loop, subsequent iteration counts after an initial execution of the inner loop may be suppressed from updating the BHR 30. In this embodiment, by the time the outer loop exits, the BHR 30 will include an indication of the iteration count of the inner loop only for the first iteration of the outer loop, and an indication of the iteration count of the outer loop.

**[0059]** In another embodiment, the inner loop iteration count may be discarded by overwriting its value in the BHR 30 with an indication of the outer loop iteration count. This preserves the greatest number of prior branch evaluations in the BHR 30, while allowing for correlation of subsequent branch evaluations with the iteration count of the outer loop.

**[0060]** In another embodiment, the outer loop iteration count may be discarded by suppressing an update of the BHR 30 when the outer loop exits. This again preserves the greatest number of prior branch evaluations in the BHR 30, while allowing for correlation of subsequent branch evaluations with the iteration count of the inner loop, which may be more predictive of subsequent branch evaluation in some applications.

**[0061]** A conditional branch instruction has common properties, including for example the branch instruction address or PC, the instruction type, and the presence, *vel non*, of indicator bits in the op code. As used herein, properties of the branch operation, and/or properties of the program that relate to the branch, are considered properties of the branch instruction. For example, whether the branch instruction PC matches the contents of one or more LBPC registers 38, and whether the branch target

address is forward or backward relative to the branch instruction PC, are properties of the branch instruction.

**[0062]** Although the present disclosure has been described herein with respect to particular features, aspects and embodiments thereof, it will be apparent that numerous variations, modifications, and other embodiments are possible within the broad scope of the present disclosure, and accordingly, all variations, modifications and embodiments are to be regarded as being within the scope of the disclosure. The present embodiments are therefore to be construed in all aspects as illustrative and not restrictive and all changes coming within the meaning and equivalency range of the appended claims are intended to be embraced therein.

## CLAIMS

What is claimed is:

1. A method of branch prediction, comprising:  
identifying a property of a branch instruction associated with a loop; and  
in response to the property, storing a multi-bit value in a Branch History Register (BHR), the value indicative of the number of iterations of a loop associated with the branch instruction.
2. The method of claim 1 wherein the property is the direction of the branch.
3. The method of claim 2 wherein the branch is backwards.
4. The method of claim 1 wherein the branch instruction is a loop-ending branch instruction.
5. The method of claim 4 wherein the PC of the branch instruction matches the contents of a Last Branch PC (LBPC) register storing the PC of the last branch instruction to update the BHR.
6. The method of claim 4 wherein the PC of the branch instruction matches the contents of any of a plurality of LBPC registers storing PCs of the last plurality of branch instruction to update the BHR.
7. The method of claim 4 wherein the branch instruction is a unique branch instruction generated by a compiler for ending loops.

8. The method of claim 4 wherein the branch instruction op-code includes one or more bits indicating it is a loop-ending branch instruction.
9. The method of claim 1 wherein storing a multi-bit value in the BHR comprises storing a predetermined number of bits in the BHR.
10. The method of claim 9 further comprising determining the value of the predetermined number of bits according to a fixed mapping of the number of loop iterations to the multi-bit value.
11. The method of claim 9 further comprising determining the value of the predetermined number of bits according to a scalable mapping of the number of loop iterations to the multi-bit value.
12. The method of claim 11 further comprising reading a plurality of thresholds to determine the scalable mapping of the number of loop iterations to the multi-bit value.
13. The method of claim 1 wherein storing a multi-bit value in the BHR comprises storing a variable number of bits in the BHR, the number of bits varying in response to the number of iterations of the loop.
14. The method of claim 13 further comprising counting loop iterations in the least significant plurality of bits of the BHR.
15. The method of claim 13 further comprising counting loop iterations in a loop counter, and transferring the value of the loop counter to the BHR when the loop terminates.

16. The method of claim 1 wherein identifying a property of a branch instruction associated with a loop comprises detecting a first loop-ending branch instruction associated with a first loop and a second loop-ending branch instruction associated with a second loop, the first loop nested within the second loop.

17. The method of claim 16 wherein storing a multi-bit value in the BHR comprises storing a multi-bit value indicative of the number of iterations of the first loop in the BHR, and further comprising storing a multi-bit value indicative of the number of iterations of the second loop in the BHR.

18. The method of claim 16 wherein storing a multi-bit value in the BHR comprises storing only a multi-bit value indicative of the number of iterations of the first loop in the BHR.

19. The method of claim 16 wherein storing a multi-bit value in the BHR comprises storing only a multi-bit value indicative of the number of iterations of the second loop in the BHR.

20. A processor, comprising:

a branch predictor operative to predict the evaluation of conditional branch instructions;

an instruction execution pipeline operative to speculatively fetch and execute instructions based on a prediction from the branch predictor;

a Branch History Register (BHR) operative to store the evaluation of conditional branch instructions;

a Loop Counter (LC) operative to count the number of iterations of a code loop; and

a control circuit operative to store in the BHR a multi-bit value indicative of the number of iterations of a loop associated with a conditional branch instruction.

21. The processor of claim 20 further comprising a Last Branch PC (LBPC) register operative to store the PC of a conditional branch instruction, and wherein the control circuit determines a conditional branch instruction is associated with a loop if the PC of the branch instruction matches the contents of the LBPC register.

22. The processor of claim 21 further comprising two or more LBPCs registers and a corresponding two or more LCs, a first LBPCs operative to store the PC of a conditional branch instruction associated with a first loop and a first LC operative to maintain a count of iterations of the first loop, and a second LBPCs operative to store the PC of a conditional branch instruction associated with a second loop and a second LC operative to maintain a count of iterations of the second loop, wherein the first loop is nested within the second loop.

23. The processor of claim 22, wherein the control circuit is operative to store in the BHR a multi-bit value indicative of the number of iterations of one execution of the first loop and a multi-bit value indicative of the number of iterations of the second loop.

24. The processor of claim 22, wherein the control circuit is operative to store in the BHR a multi-bit value indicative of the number of iterations of one execution of the first loop and to not store in the BHR an indication of the number of iterations of the second loop.

25. The processor of claim 22, wherein the control circuit is operative to store in the BHR a multi-bit value indicative of the number of iterations of the second loop and to not store in the BHR an indication of the number of iterations of any execution of the first loop.

26. The processor of claim 20 wherein the BHR is operative to increment a plurality of bits in response to each taken evaluation of the conditional branch instruction associated with the loop so as to maintain a loop iteration count directly in the BHR.

27. The processor of claim 20 further comprising thresholding logic operative to map a loop iteration count to a fixed, a multi-bit value in response to a plurality of threshold values.

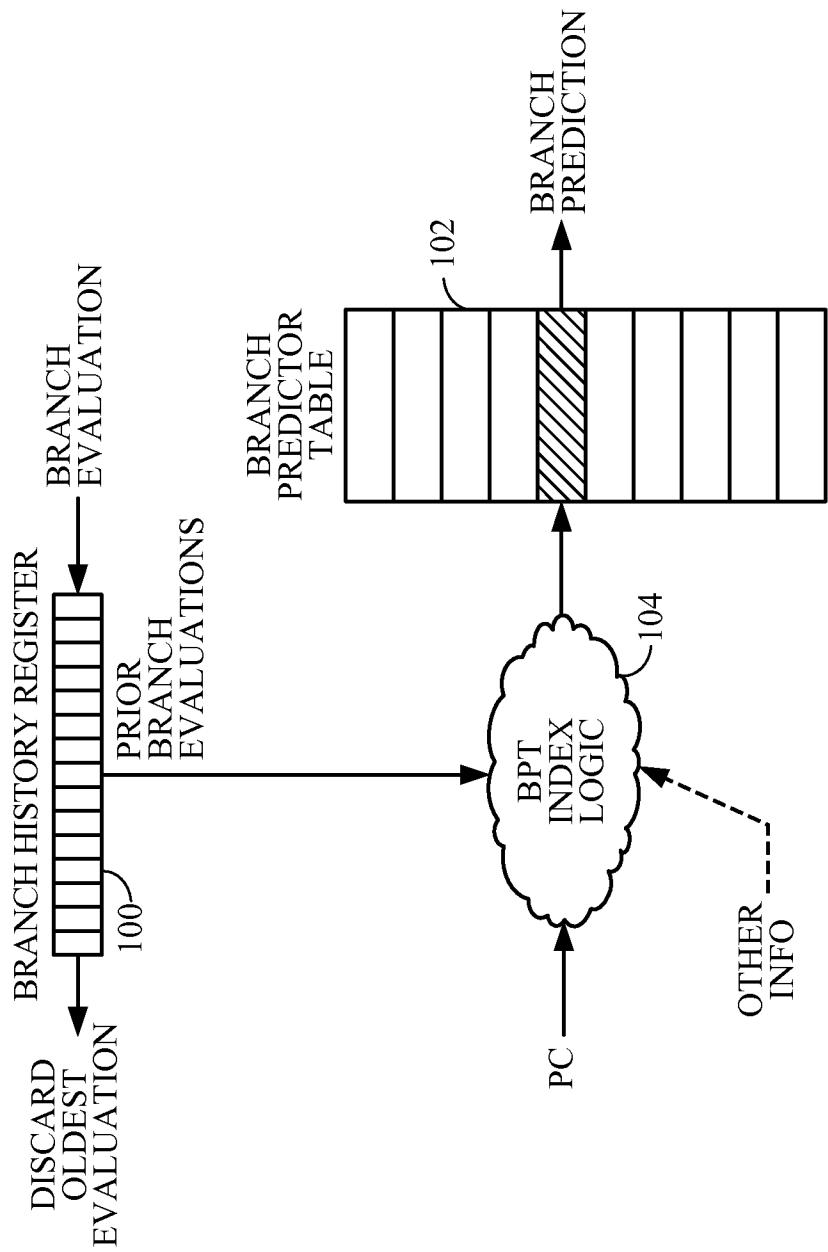


FIG. 1  
(PRIOR ART)

2/4

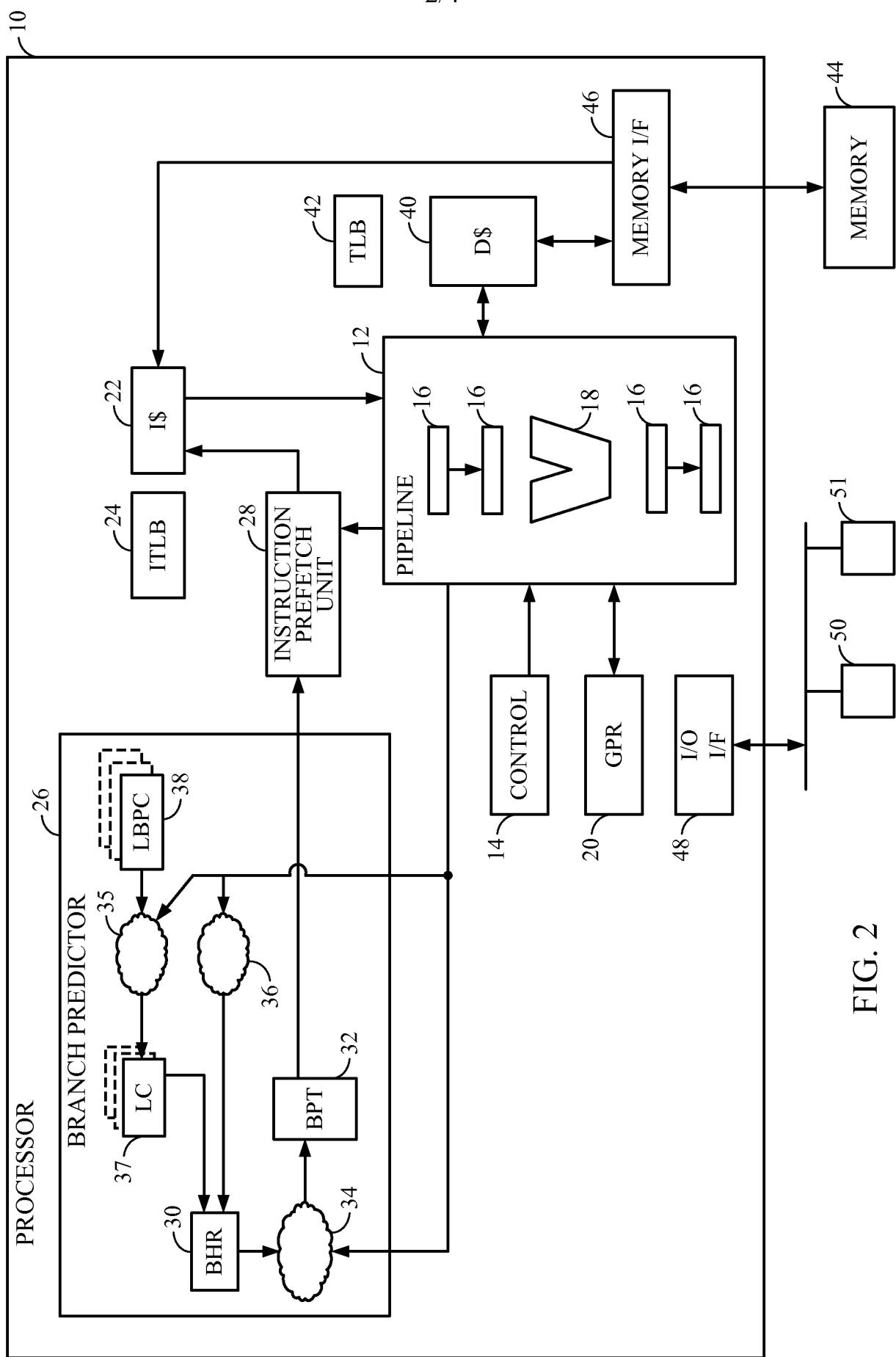


FIG. 2

3/4

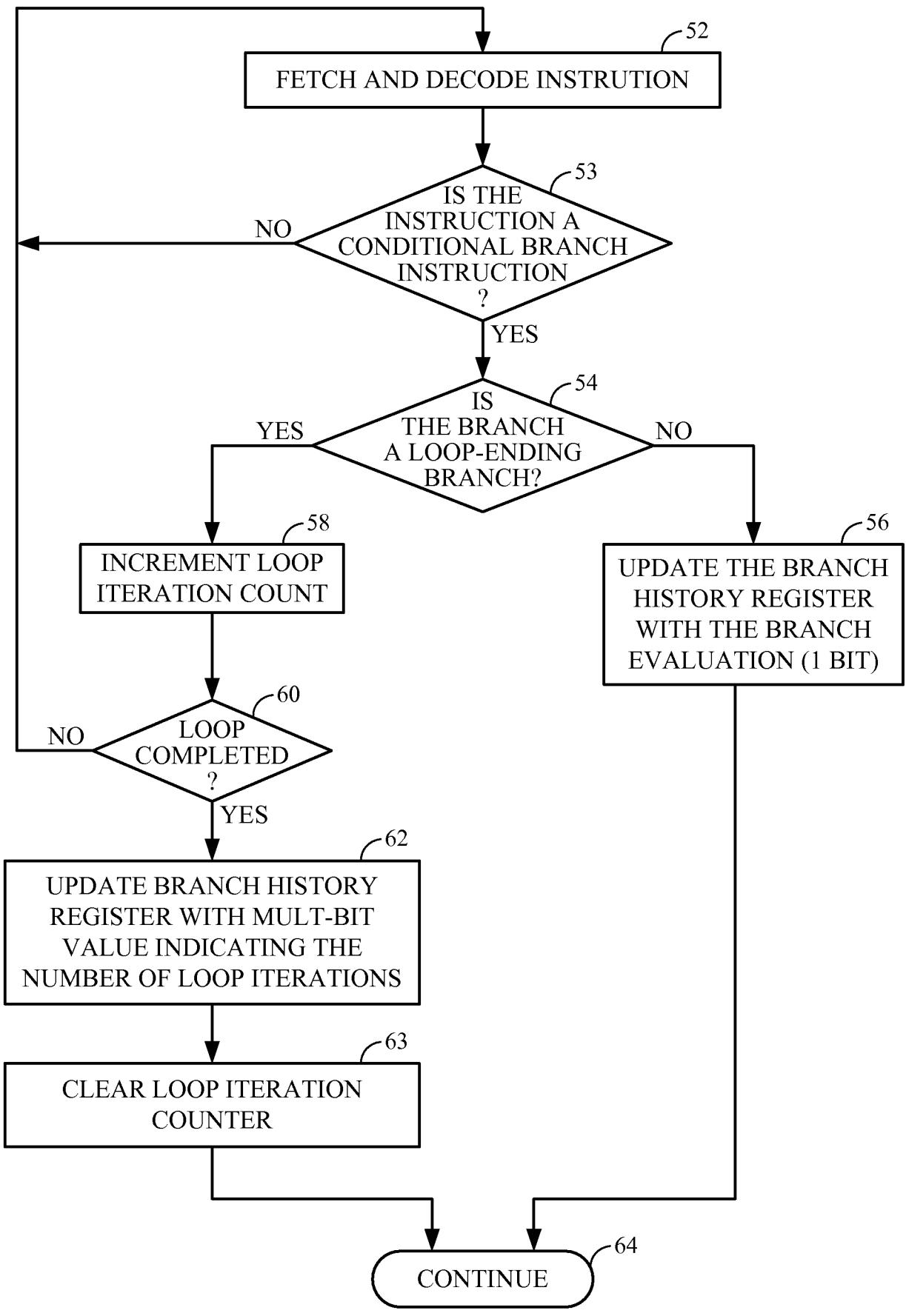


FIG. 3

4/4

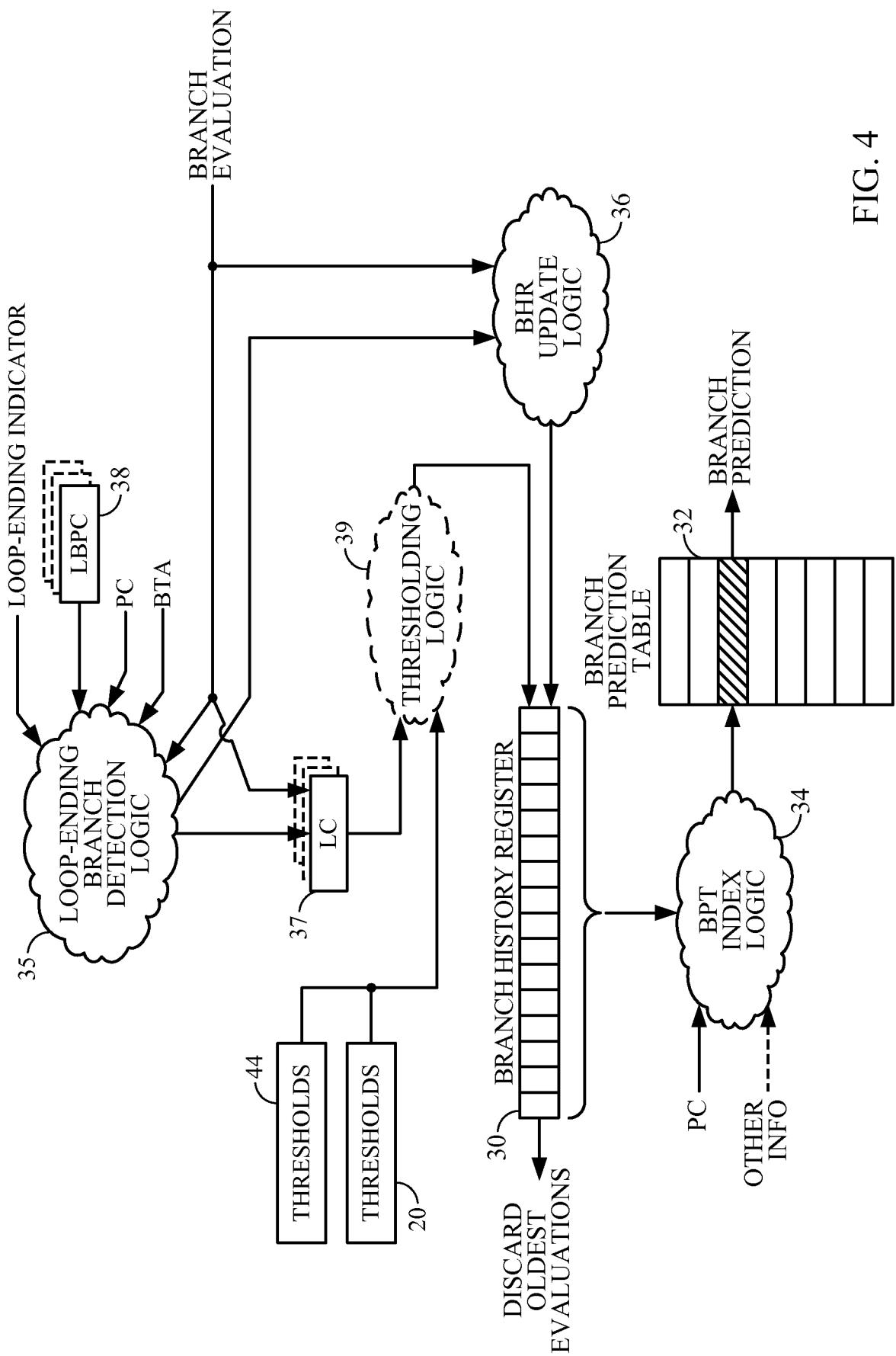


FIG. 4