# IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Utility Patent Application (Provisional)

**TITLE:** HIVE MULTIPROCESSOR

**INVENTORS:** Suchir Mishra, Pavan Bellam, Arohan Deshpande, Rohan Malla, Pujan Shah

## FIELD OF THE INVENTION

**[0001]** This invention pertains to a novel multi-threaded microprocessor structure and an associated instruction pipeline mechanism named RAPID (Real-time Adaptive Pipeline Instruction Dispatch). The invention focuses on optimizing on-chip instruction dispatch for improved efficiency, introducing significant advancements to conventional microprocessor architectures.

## BACKGROUND

**[0002]** In modern computing, multitasking and execution of multiple programs at once is handled through the use of multiple cores present in the central processing unit of computers to allow the distribution and execution of multiple threads of instructions relating to said programs simultaneously. Each core is its own separate microprocessor capable of executing instructions independently.

**[0003]** There are two major flaws that lie in this existing design which limit the efficiency of instruction execution - and thus the overall performance and power consumption of microprocessors. There exist many types of programs that are unable to be spread effectively across multiple instruction threads, creating a performance bottleneck dependent on the instruction execution rate of a single core.

**[0004]** Additionally, due to the pipeline system present in virtually all modern microprocessors, instruction execution is spread across multiple stages in a single core, with internal static buffers holding the statuses of instructions in between the stages of the fetching of the instruction and the execution. When these buffers are fully occupied, the pipeline is forced to stall at any of the various stages as it waits for existing instructions to be executed and the buffers to be emptied, wasting execution clock cycles. While mechanisms such as out of order execution have been invented to mitigate this, this issue is still present in all microprocessors.

**[0005]** As the demand for computing performance and the consequent energy consumption grows rapidly, the advent of novel microprocessor architecture designs

becomes increasingly important. This invention aims to mitigate the issues of single-core bottlenecks as well as further minimizing the emergence of pipeline stalls.

## BRIEF SUMMARY OF THE INVENTION

**[0006]** The invention relates to a novel multi-thread microprocessor architecture designed to optimize instruction processing, reducing pipeline stall latency while increasing power efficiency. The invention pioneers the RAPID (Real-time Adaptive Pipeline Instruction Dispatch) mechanism, a dynamic instruction dispatch system allowing the microprocessor to dispatch instructions to one backend pipeline unit when the buffer is full. By doing so, the invention mitigates pipeline stalls present in existing microprocessor designs, improving overall system performance and power efficiency, by introducing two major modifications to conventional microprocessor architectures.

**[0007]** First, this invention splits the conventional processor execution pipeline, with the first half including only the components necessary to appear as an individual logical thread to executing software and to fetch instructions independently from memory. The second half consists of the remaining components necessary for the proper decoding and execution of instructions. By splitting these two sections of a conventional pipeline, multiple instances can be made of the instruction fetching “front-ends” as well as the executing “back-ends.” With this, the independent front-ends are able to route instructions to any of the available back-end pipelines, dynamically optimizing routing based on execution resource availability.

**[0008]** Second, this invention adds a new pipeline stage dubbed “Fill Check.” Fill Check is an additional pipeline stage after the instruction has been fetched and serves as the mechanism used to check the buffers of the back ends in any one design to determine

whether they are filled or if they are available to execute the instruction in question. The specific operation of the Fill Check stage may vary depending on the implementation.

**[0009]** Lastly, this invention provides modified versions of pipeline mechanisms commonly found in existing instruction execution pipeline designs in order to make them compatible with the novel split-pipeline and instruction dispatch systems.

**[00010]** Clustering the front end logical threads to the back end execution units would dramatically decrease latency and increase power efficiency. This configuration will henceforth be referred to as a "HIVE," for the purpose of discerning the invention presented from the common application of cores within computer processors. A HIVE-based architecture improving efficiency would lead to lesser power draw within large applications of processing power, such as data centers (which are ever-increasing in demand). Portable electronics would also greatly benefit from the decreased power draw, as longer-lasting batteries grow more and more in desire.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[00011]** Some embodiments of the present invention are illustrated as an example and are not limited by the figures of the accompanying drawings, in which like references may indicate similar elements and in which:

**[00012]** FIG. **1** - Figure **1** depicts a comprehensive overview of a conventional multicore processor pipeline, in the form of a block diagram.

**[00013]** FIG. 2 - Figure **2** illustrates an overview of a HIVE-based multiprocessor pipeline, in the form of a block diagram.

**[00014]** FIG. **3** - Figure **3** is a block diagram that illustrates the Real-time Adaptive Instruction Dispatch mechanism (RAPID).

**[00015]** FIG. **4** - Figure **4** is a diagram that depicts the storage structures that are present in the backend of a HIVE-based multiprocessor.

**[00016]** FIG. **5** - Figure **5** is a flowchart that depicts the process of how RAPID selects the back end to dispatch instructions to.

**[00017]** FIG. **6** - Figure **6A** illustrates the branch predictor pipeline in the front end of a HIVE-based processor, in the form of a block diagram.

**[00018]** FIG. **6** - Figure **6B** is a block diagram that shows the pipeline of the branch predictor in the back end of a HIVE based processor.

**[00019]** FIG. **7** - Figure **7A** is a block diagram that depicts the routing of an instruction in relation to memory in the front end of a HIVE based processor.

**[00020]** FIG. **7** - Figure **7B** is a block diagram that depicts the routing of an instruction in relation to memory in the back end of a HIVE based processor.

**[00021]** FIG. **8** - Figure **8A** is a block diagram that depicts the routing of an instruction in relation to the μOP Cache in the front end of a HIVE based processor

**[00022]** FIG. **8** - Figure **8B** is a block diagram that depicts the routing of an instruction in relation the the μOP Cache in the back end of a HIVE based processor

**[00023]** FIG. **9** - Figure **9** is a block diagram depicting a novel register allocation and renaming system necessary to allow for proper functioning of a HIVE based processor.

## DETAILED DESCRIPTION OF THE INVENTION

**[00024]** FIG. 1 shows a high-level block diagram of the structure of conventional modern multiprocessors in order to provide a comparison with which the differences present in the HIVE-based multiprocessor can be clarified. In the context of this invention, the "front-end" of a processor refers to the collection of mechanisms responsible for fetching processor instructions, holding the instruction set architecture (ISA)-defined architectural state of a logical execution thread as seen by an operating system, and initially dispatching them to the "back-end." Each front-end present in a HIVE-based multiprocessor holds the architectural state for its own separate logical thread, capable of accepting its own instructions independent of other existing front-ends. The "back-end" consists of the remaining elements within a (Turing complete) processor pipeline responsible for the flow and management of internal operations and execution of instructions.

**[00025]** FIG. 2 shows a high-level block diagram of the structure of the HIVE-based multiprocessor. While a traditional multiprocessor as depicted in FIG. 1 consists of groups of joined front-ends and back-ends (commonly referred to as "cores"), a HIVE-based multiprocessor has front-ends that are capable of dispatching instructions freely to any of the available back-ends through an instruction dispatch mechanism named RAPID (Real-time Adaptive Pipeline Instruction Dispatch). Allowing front-ends to dispatch instructions to any of the available back-ends enables each logical thread as visible by an operating system to have all of the execution resources available in a multiprocessor design as opposed to traditional processing cores where the joint front-end

and back-end restricts a logical thread as visible by an operating system to only have the execution resources available within the back-end of a single core. The functionality and behavior of the logical threads in a HIVE-based multiprocessor as viewed by the operating system should be identical to the logical threads of a similar traditional multiprocessor.

**[00026]** In the context of this invention, a singular "HIVE" is defined as a cluster of interconnected front-ends and back-ends. The configuration of front-ends and back-ends within a single HIVE can vary between implementations. Any number of front-ends and back-ends can be clustered into a HIVE through the RAPID mechanism. Additionally, the back-ends within a HIVE are not required to be identical to each other. The number of pipeline stages, execution resources, and execution functionality may vary between individual back-ends found within a HIVE.

**[00027]** One processor may incorporate multiple HIVEs in its design. In such a situation, front-ends may only dispatch instructions to the back-ends located in their own HIVEs. Front-ends are unable to dispatch instructions to back-ends located in HIVEs other than their own in a processor implementation with multiple HIVEs.

**[00028]** FIG. 3 and FIG. 4, are block diagrams depicting the workings of the RAPID mechanism. As shown in FIG. 3, the implementation of RAPID defines two pipeline stages that need to be present in the pipeline of a front-end: the "Classification" stage and the "Fill Check" stage. FIG. 4 depicts the types of elements that must be present in all implementations of a HIVE-based back-end as part of the RAPID mechanism.

**[00029]** The exact implementation of the Classification stage may vary based on implementation, however it must perform two functions. First, it must read the instruction word fetched from memory in the preceding pipeline stages of the front-end and interpret what categories the operations encoded in the same instruction are classified under. These categories may differ according to implementations, however some examples of operation categories would be integer operations, floating-point operations, load/store operations, and SIMD (Single Instruction/Multiple Data) operations, among others. Second, it must communicate the categories of operations that the instruction in question requires to the succeeding Fill Check Stage.

**[00030]** The Fill Check stage is the final stage present in the front-end pipeline of a HIVE multiprocessor. The exact implementation of the Fill Check stage may vary, however it must perform one function. In the Fill Check stage,the front-end must communicate the categories of operations of its current instruction, that it received from the preceding Classification stage, with the Fill Check Bus defined in FIG. 3 as a part of the RAPID mechanism. The Fill Check Bus accesses registers present in every HIVE-defined back-end to determine whether the necessary buffers present in a back-end are capable of accepting operations. Examples of necessary buffers would be the reorder buffer present in many modern processor pipelines that support out-of-order execution functionality as well as the issue queue buffers for the necessary operation types encoded in the current instruction. Once the fill-check bus finds the first suitable back-end to dispatch to, routes all information needed for the execution of the current instruction to

that back-end. The exact contents of all the information routed at this stage may vary between implementations. Each front-end contains its own Fill Check Bus.

**[00031]** FIG. 4 depicts the types of elements in an arbitrary HIVE-defined back-end that must be present in any implementation of a HIVE-based multiprocessor and the RAPID mechanism. Each back-end must hold registers that record the count of empty buffer entries of all present buffers within the same back-end in real-time. The Fill Check Bus reads the registers for the buffers corresponding to what is required for the current instruction to determine whether a certain back-end is capable of receiving the instruction. Examples of these buffers would be the same as the ones described above: the reorder buffer present in many modern processor pipelines that support out-of-order execution functionality as well as the issue queue buffers for the necessary operation types encoded in the current instruction. If any other operation types with their own issue queue buffers are implemented in a back-end, they must also have registers keeping count of empty buffer entries that are readable by the Fill Check Bus.

**[00033]** FIG. 6a and FIG. 6b depict a method for incorporating branch prediction in HIVE-based multiprocessors. Branch prediction is a mechanism found at the fetch stage of many modern processing pipelines for improving performance by speculatively predicting the outcome of a branch operation before it goes through the remainder of the processor pipeline and is executed. Once the branch is executed, the result must be fed back into the branch predictor in order for it to verify its prediction. This is a simple process in traditional joint processing pipelines, however, it becomes slightly more

complicated when incorporated into a HIVE-based processor. Because the branch predictor exists in the front-end, and an instruction finishes execution in a back-end, a mechanism to route the result of a branch instruction back to its respective front-end must be utilized. FIG. 6a and FIG. 6b together outline such a mechanism.

**[00034]** As outlined in FIG. 6a, when a branch instruction is routed to a back-end, information about the index corresponding to the originating front-end is also sent and stored in the instruction entry located in the back-end while the branch instruction is being executed. As outlined in FIG. 6b, once execution has commenced, the result is routed back to the correct branch predictor by utilizing the index corresponding to its originating front-end.

**[00035]** FIG. 7a and FIG. 7b illustrate a mechanism for incorporating virtual memory functionality in HIVE-based multiprocessors. In modern computing, virtual memory addresses are unique for each logical thread. Since the architectural state for a logical thread is held in the front-end, a similar complication as the one found in branch prediction arises when incorporating virtual memory in a HIVE-based processor. Once again, by sending the respective front-end index to the selected back-end, instructions involving memory operations are able to access the correct memory management units and translation lookaside buffers - which are located in the front-end and are key elements of virtual memory mechanisms - during execution.

**[00036]** FIG. 8a and FIG. 8b illustrate a mechanism for incorporating micro-op caches into HIVE-based multiprocessors. Micro-op caches are memory units that hold the decoded micro-operations of the most recent instructions accessed by the logical thread.

Since the most recent instructions accessed are respective to independent logical threads, a micro-op cache, if implemented, must reside in the front-end. In HIVE multiprocessor implementations where the decode stage occurs in the back-end, information about the respective front-end index must once again be sent when an instruction is dispatched to a back-end so that the decoded instructions are able to be routed back to the correct micro-op cache to be stored.

**[00037]** FIG. 9 depicts a novel register renaming mechanism that is necessitated due to the nature of HIVE-based multiprocessors when implementing out-of-order execution. Register renaming is a mechanism found in processor pipelines with out-of-order execution in order to store and manage data dependencies reliably when executing instructions in an order different from the original program. A traditional register renaming scheme involves a single register allocation table (RAT) that holds translations for the ISA-defined architectural register indexes into indexes corresponding to their actual physical location inside the register file. Managing and indexing the necessary register becomes more complicated in a HIVE-based processor where the operand data necessary for executing an instruction in a back-end may originate from a different, separate back-end.

**[00038]** In order to address this complication, a register renaming scheme involving two levels of register files, labeled L1 and L2 has been defined. The L2 register file is accessible by every front-end and back-end present in a HIVE and stores all register data being utilized throughout the HIVE. Each back-end has its own L1 cache, which is not accessible to any other back-end except the one that it resides in.

**[00039]** Each front-end has a RAT that holds information about which ISA-defined architectural register index corresponds to which physical register index in the L2 register file. The operand and result indexes present in an instruction are translated from their original architectural defined ones to their physical L2 locations using the RAT before they are dispatched to a back-end for execution. Each back-end must contain its own sub-RAT that next translates the L2 indexes to the indexes of its own L1 register file. If a valid entry for the corresponding L1 index is not found in the RAT, data is fetched from the L2 register file and an entry is created in the sub-RAT. From here, the instruction is executed normally.

**[00040]** After the instruction is executed, if it has written a result to a register in the L1 register file, the data must be propagated back to the L2 register file so the latest architectural state is available for every back-end. Once the data is successfully propagated, entries corresponding to the L2 index, where the latest register state now resides, in the sub-RATs of other back-ends must be invalidated since they are no longer accurate. When those back-ends need to index that location again, they will fetch the data from L2, copy it into their L1 register files again, and create a new entry in their sub-RATs.

**CLAIMS**

What is claimed is:

1. A multi-threaded microprocessor architecture, comprising:
    a. a split processor execution pipeline comprising:
        i. A first third with components for instruction fetching, preservation of the processor's architectural state, and preliminary dispatch;
        ii. A second third with components for dynamic instruction routing; and
        iii. A last third containing components for instruction decoding and final execution;
    b. a plurality of instruction fetching "front-ends" corresponding to the first third of the split pipeline;
    c. a real-time adaptive pipeline instruction dispatch (RAPID) mechanism corresponding to the second third of the split pipeline
    d. a plurality of execution "back-ends" corresponding to the second half of the split pipeline, with the front-ends dispatching instructions through RAPID, which dynamically routes instructions to any available back-end based on execution resource availability.
2. The microprocessor architecture of claim 1, wherein the dynamic routing of instructions from the front-ends to available back-ends is controlled by a Real-time Adaptive Pipeline Instruction Dispatch (RAPID) mechanism.

3. The microprocessor architecture of claim 1 or 2, further comprising a "Fill Check" pipeline stage added after instruction fetching, the Fill Check stage being configured to determine whether buffers of the back-ends are filled or available for executing an instruction.
4. The microprocessor architecture of any of the preceding claims, wherein any number of instances of each stage of the split pipelines of mentioned in preceding claims can be clustered together to form a “HIVE” and function as a multiprocessor.

**ABSTRACT**

This patent application describes a novel microprocessor architecture incorporating the Real-time Adaptive Pipeline Instruction Dispatch (RAPID) mechanism. The architecture enhances on-chip instruction dispatch by dividing the execution pipeline into stages: instruction fetching and dispatch, dynamic routing, and instruction decoding and execution. The invention aims to maximize the use of on-chip instruction execution resources by enabling flexible instruction routing to available resources. This approach introduces a departure from traditional microprocessor architectures, potentially improving overall performance.

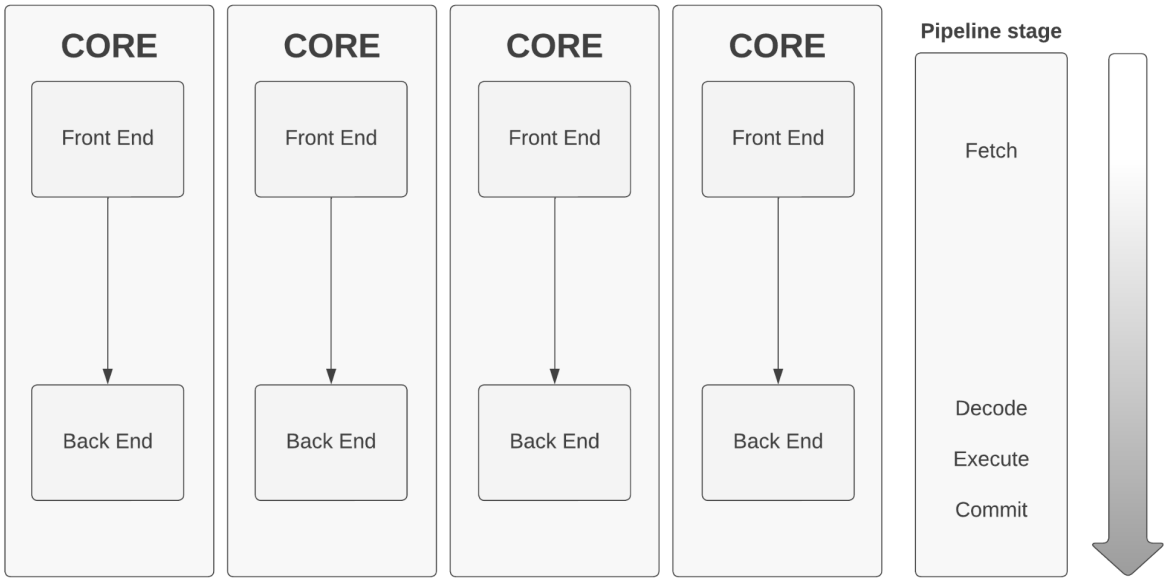
CORE
Front End
Back End
CORE
Front End
Back End
CORE
Front End
Back End
CORE
Front End
Back End
Pipeline stage
Fetch
Decode
Execute
Commit

FIG. 1

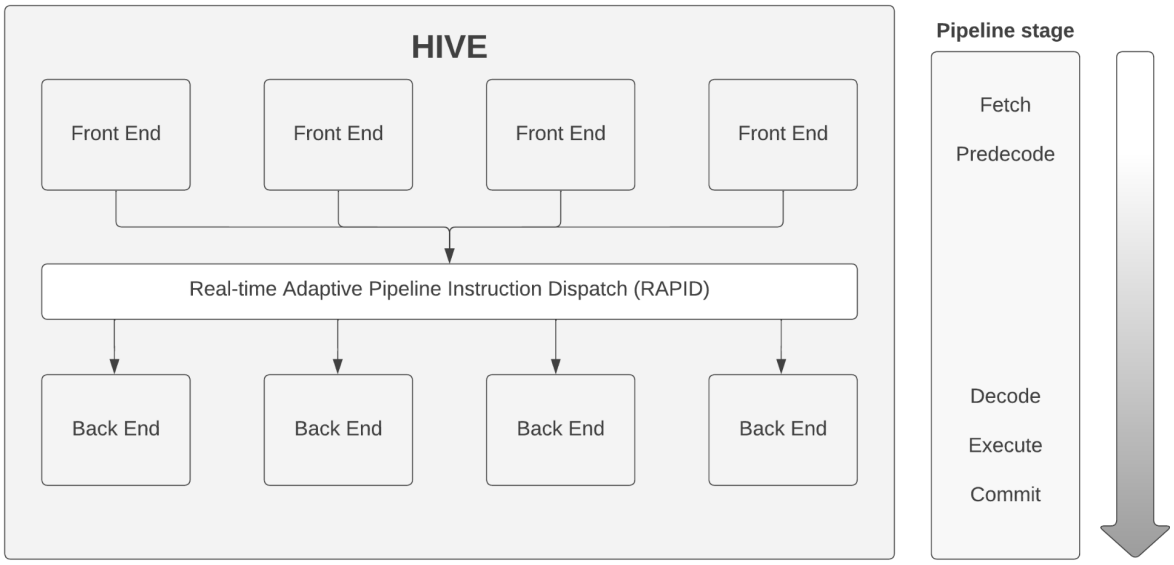
HIVE
Front End
Front End
Front End
Front End
Real-time Adaptive Pipeline Instruction Dispatch (RAPID)
Back End
Back End
Back End
Back End
Pipeline stage
Fetch
Predecode
Decode
Execute
Commit

FIG. 2

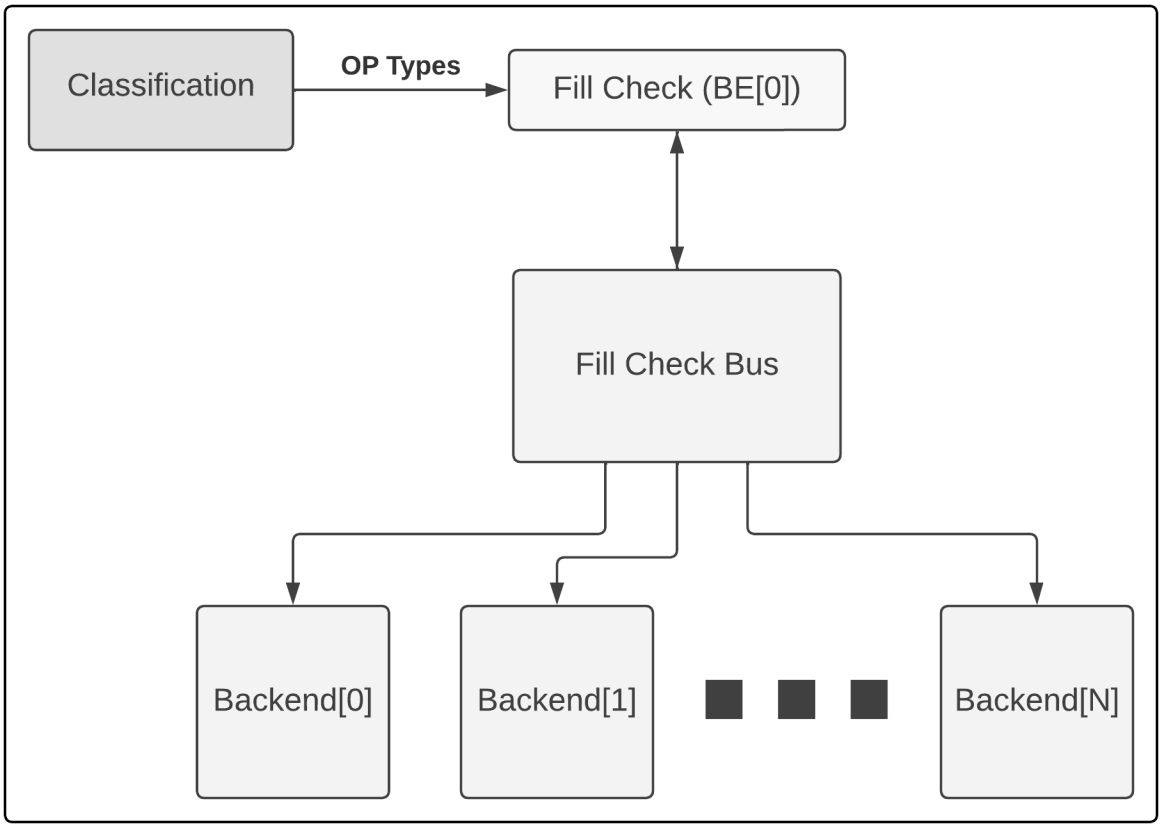
Classification
OP Types
Fill Check (BE[0])
Fill Check Bus
Backend[0]
Backend[1]
Backend[N]

**FIG. 3**

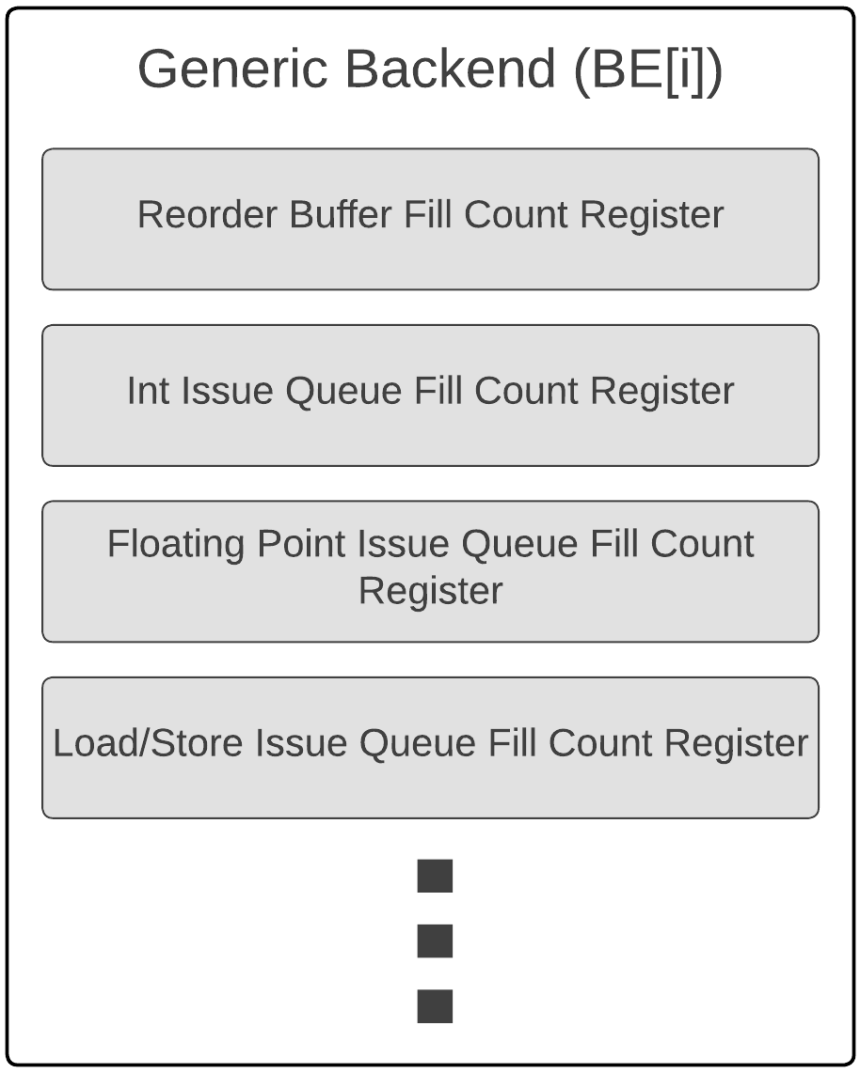
Generic Backend (BE[i])
Reorder Buffer Fill Count Register
Int Issue Queue Fill Count Register
Floating Point Issue Queue Fill Count Register
Load/Store Issue Queue Fill Count Register

**FIG. 4**

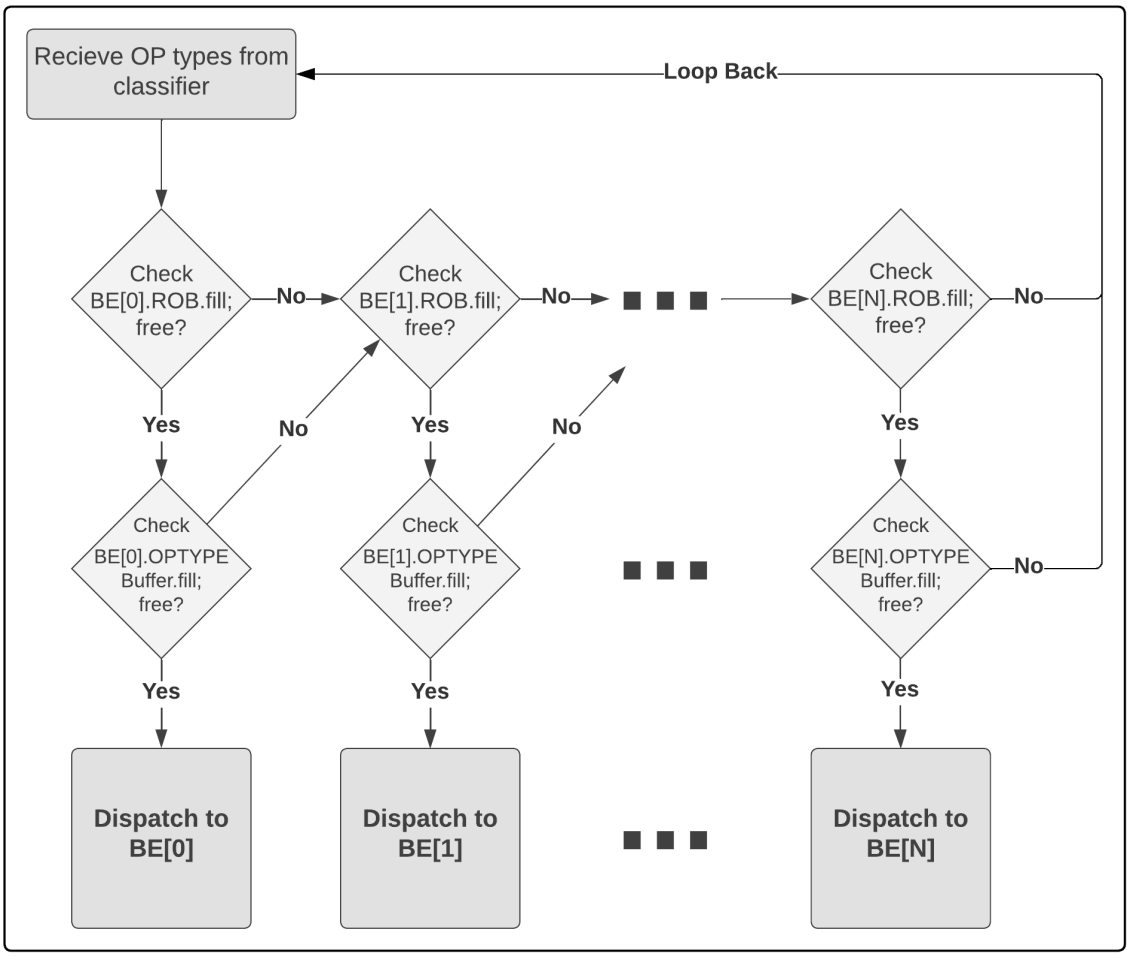
Recieve OP types from classifier
Loop Back
Check BE[0].ROB.fill; free?
No
Check BE[1].ROB.fill; free?
No
Check BE[N].ROB.fill; free?
No
Yes
Yes
Yes
No
No
Check BE[0].OPTYPE Buffer.fill; free?
Check BE[1].OPTYPE Buffer.fill; free?
Check BE[N].OPTYPE Buffer.fill; free?
No
Yes
Yes
Yes
Dispatch to BE[0]
Dispatch to BE[1]
Dispatch to BE[N]

**FIG. 5**

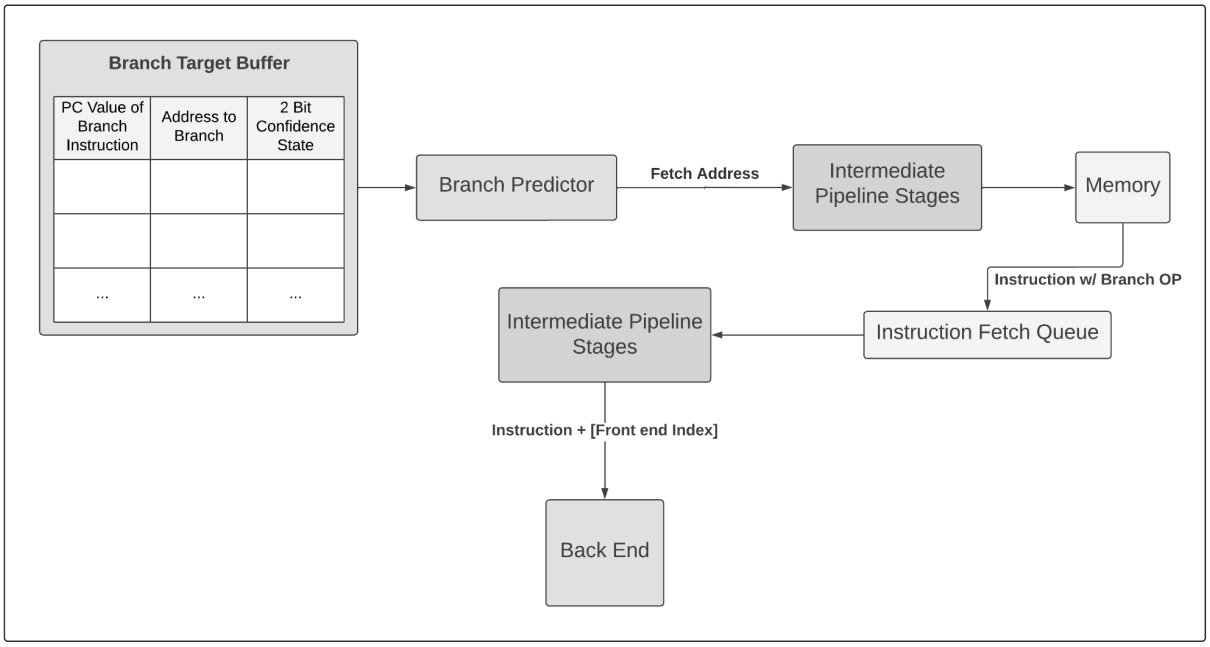


FIG. 6A

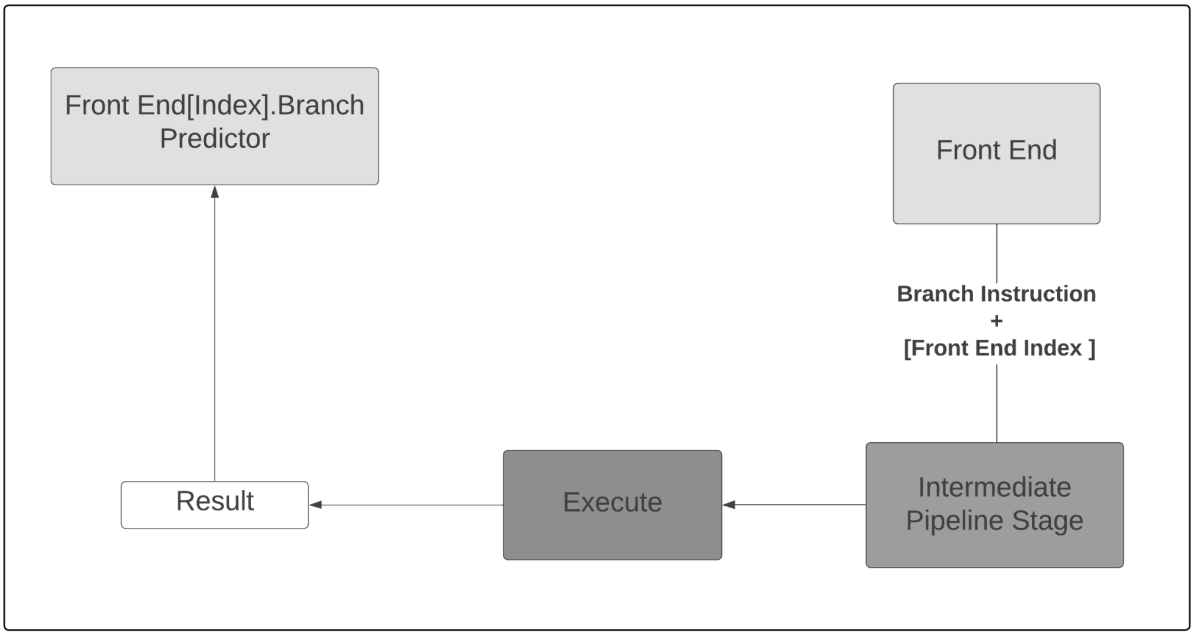


FIG. 6B

FIG. 6

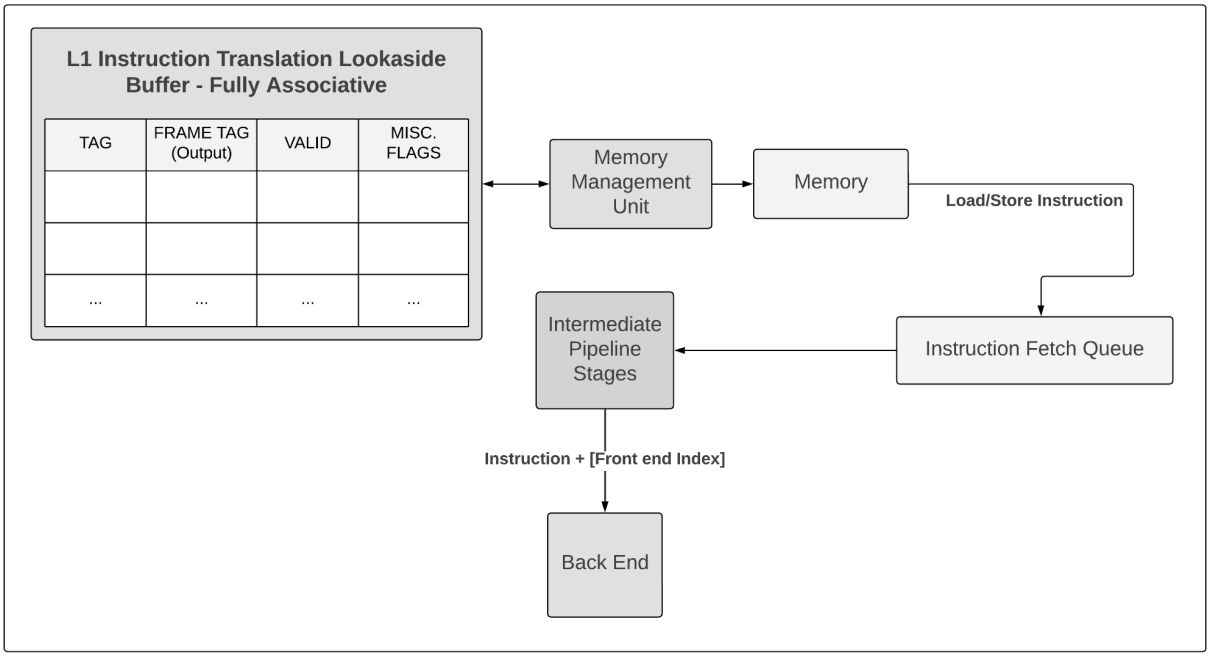


FIG. 7a

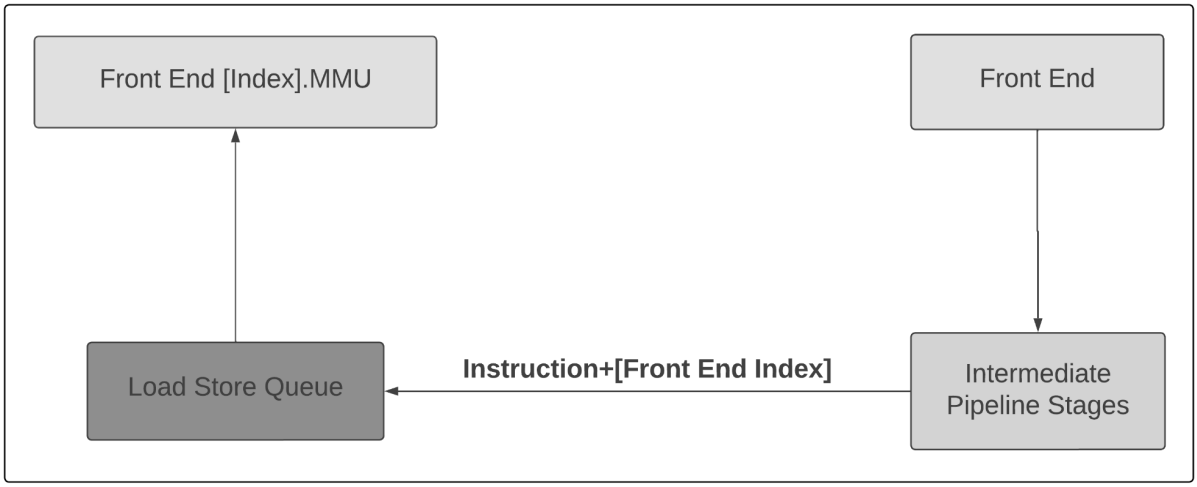


FIG. 7b

FIG. 7

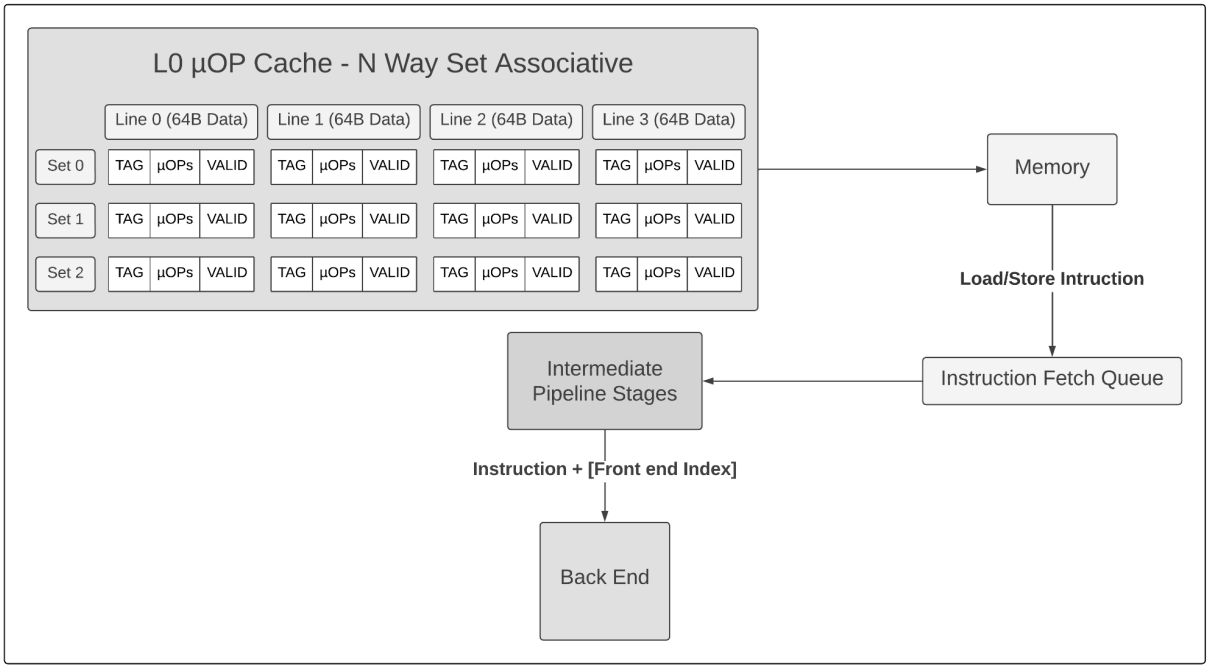


FIG. 8A

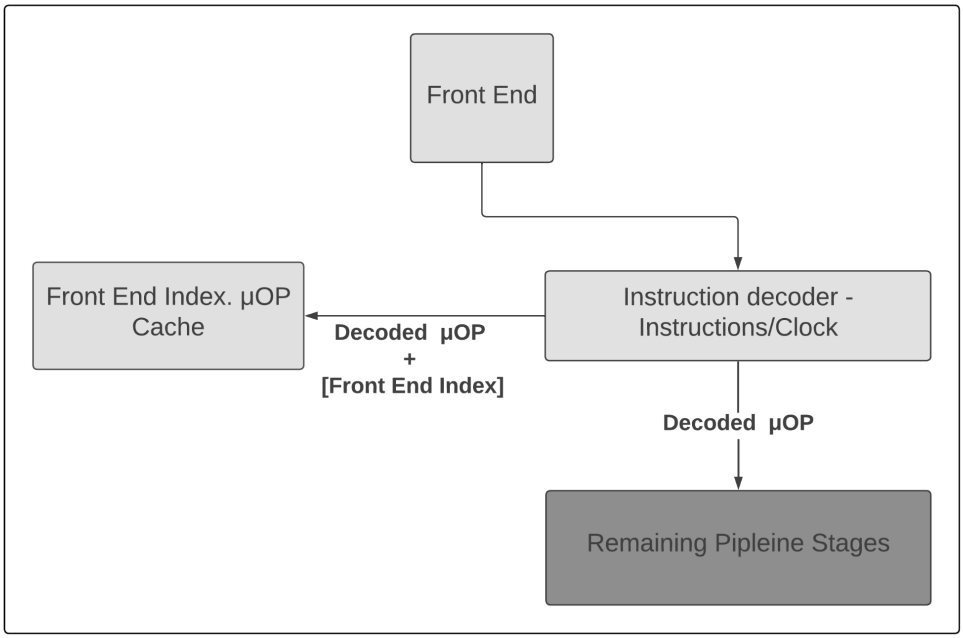


FIG. 8b

FIG. 8

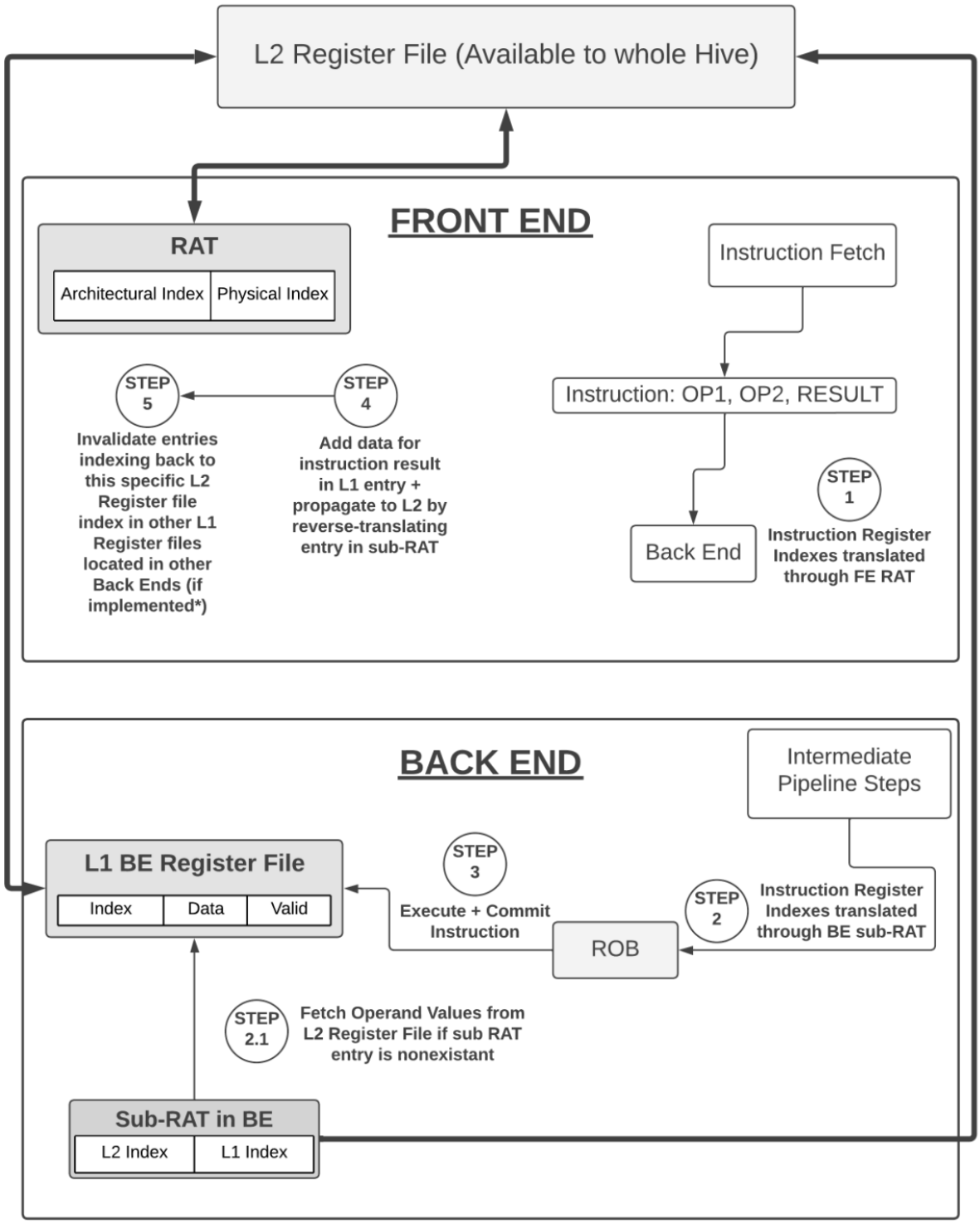
L2 Register File (Available to whole Hive)
FRONT END
RAT
Architectural Index
Physical Index
Instruction Fetch
Instruction: OP1, OP2, RESULT
STEP 5
Invalidate entries indexing back to this specific L2 Register file index in other L1 Register files located in other Back Ends (if implemented*)
STEP 4
Add data for instruction result in L1 entry + propagate to L2 by reverse-translating entry in sub-RAT
Back End
STEP 1
Instruction Register Indexes translated through FE RAT
BACK END
Intermediate Pipeline Steps
L1 BE Register File
Index
Data
Valid
STEP 3
Execute + Commit Instruction
ROB
STEP 2
Instruction Register Indexes translated through BE sub-RAT
STEP 2.1
Fetch Operand Values from L2 Register File if sub RAT entry is nonexistant
Sub-RAT in BE
L2 Index
L1 Index

FIG. 9