An Out-of-Order RiSC-16

Tomasulo + Reorder Buffer = Interruptible Out-of-Order

ENEE 446: Digital Computer Design, Fall 2000

Prof. Bruce Jacob, http://www.ece.umd.edu/~blj/

This paper describes an out-of-order implementation of the 16-bit Ridiculously Simple Computer (RiSC-16), a teaching ISA that is based on the Little Computer (LC-896) developed by Peter Chen at the University of Michigan.

1. RiSC-16 Instruction Set

The RiSC-16 is an 8-register, 16-bit computer. All addresses are shortword-addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). Like the MIPS instruction-set architecture, by hardware convention, register 0 will always contain the value 0. The machine enforces this: reads to register 0 always return 0, irrespective of what has been written there. The RiSC-16 is very simple, but it is general enough to solve complex problems. There are three machine-code instruction formats and a total of 8 instructions. The instruction-set is given in the following table.

Assembly-Code Format		Meaning
`add`	`regA, regB, regC`	R[regA] <- R[regB] + R[regC]
`addi`	`regA, regB, immed`	R[regA] <- R[regB] + immed
`nand`	`regA, regB, regC`	R[regA] <- ~(R[regB] & R[regC])
`lui`	`regA, immed`	R[regA] <- immed & 0xffc0
`sw`	`regA, regB, immed`	R[regA] -> Mem[R[regB] + immed]
`lw`	`regA, regB, immed`	R[regA] <- Mem[R[regB] + immed]

`beq`	`regA, regB, immed`	if (R[regA] == R[regB]) {
		PC <- PC + 1 + immed
		(if label, PC <- label)
		}

`jalr`	`regA, regB`	PC <- R[regB], R[regA] <- PC + 1
PSEUDO-INSTRUCTIONS:		
`nop`		do nothing
`halt`		stop machine & print state
`lli`	`regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi`	`regA, immed`	R[regA] <- immed
`.fill`	`immed`	initialized data with value *immed*
`.space`	`immed`	zero-filled data array of size *immed*

The instruction-set is described in more detail (including machine-code formats) in *The RiSC-16 Instruction-Set Architecture*. System calls are described only briefly in the present document; they are special instances of the JALR instruction in which the immediate value is non-zero (and indicates the system-call ID). This addition to the instruction-set architecture is made to support interrupts and interrupt-handling. System calls, interrupts/exceptions, and the handling of interrupts and exceptions are described in more detail in the document *RiSC-16 System Architecture*.

2. Background

To begin with, a little background on today's out-of-order designs: in particular, there are three important papers that helped shape contemporary out-of-order computing:

R. M. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units." *IBM Journal of Research and Development*, 11(1):25–33. January 1967.

J. E. Smith and A. R. Pleszkun. "Implementation of precise interrupts in pipelined processors." In *Proc. 12th Annual International Symposium on Computer Architecture (ISCA-12)*, pp. 36–44. June 1985.

G. S. Sohi and S. Vajapeyam. "Instruction issue logic for high-performance, interruptable pipelined processors." In *Proc. 14th Annual International Symposium on Computer Architecture (ISCA-14)*, pp. 27–34. June 1987.

The first paper gives a concrete hardware architecture for resolving inter-instruction dependencies through the register file, thereby allowing out-of-order issue to the functional units; the second paper describes several mechanisms for handling precise interrupts in pipelines with in-order issue but out-of-order completion, the reorder buffer being one of these mechanisms; finally, the third paper combines the previous two concepts into a mechanism that supports both out-of-order instruction issue and precise interrupts (as well as branch misspeculations). The following sections go into a little more detail on each.

Tomasulo's Algorithm

Tomasulo called his mechanism the Common Data Bus. Because the mechanism is more expansive than a simple bus, it is usually referred to as *Tomasulo's algorithm* instead. The underlying principle is this: *when the data is stale, keep track of where new data will be coming from*. Here is how the principle is used. The register file holds data. For brief windows in time, data words in the register file are stale, in that they are soon to be overwritten by an instruction that has not yet completed. Take, for example, the following code:

```
lw     r1, 16(r2)
addi   r1, r1, 1
```

Ignoring dependences between the instructions, the load instruction would be likely to take longer than the add-immediate, because the load performs both an add-immediate and a memory-access: it requires an add-immediate of register 2 with the value 16 to generate the load address. Only after the address is generated can the memory access begin. Therefore, by the time that the **addi** is ready to read the value of **r1** out of the register file, it is likely that the load is still in mid-execution. If this is the case, then **r1** contains stale data that cannot be used for computation by the **addi** or any other instruction that follows the load.

Previous architectures would either stall in this instance or use forwarding paths in the pipeline. Tomasulo's algorithm uses a different mechanism: instead of keeping track of the data in **r1**, it keeps track of the data's source, i.e. the load instruction which will update **r1** in the near future. When the load is decoded, it is enqueued in a numbered *reservation station* awaiting execution; **r1** is tagged as invalid; and the register file holds the reservation station ID instead of **r1**'s contents.

Therefore, rather than keeping track of the data value, the register keeps track of where the new data will come from. When the **addi** instruction is decoded and enqueued, it reads the ID from the register file and is placed in its own reservation station, knowing that one of its operands is invalid, but also knowing the unique ID of the instruction that will produce the operand. That unique ID is the ID of the reservation station holding the load instruction.

This information is used to forward operands from the functional units to the instructions awaiting data in reservation stations. Whenever a a functional unit produces a result value (either an ALU result or a load-word memory request), the functional unit producing the value broadcasts that value as well as the corresponding instruction's ID on the Common Data Bus. All instructions sitting in reservation stations look to this bus and gate in the data whenever one of its operands is invalid and the corresponding tag matches the ID of the data on the common data bus. As soon as an instruction's operands are all valid, the instruction is ready to execute, whether this is before or after the instructions that come before it in the instruction stream. The register file also monitors this bus, and if the ID on the bus matches the ID in any invalid register, the data is gated in to that register, and the register is marked as valid.

The architecture is very simple but extremely powerful and capable of resolving all dependencies through the register file. It also provides a form of register renaming that allows the simultaneous or out-of-order execution of multiple reads and writes to the same register. The algorithm (as well as numerous variations on it) has become a staple in modern high-performance CPU design.

As an aside, the fixed-point pipeline in the IBM System/360 Model 91 supported out-of-order completion as well, but in a slightly simpler form: all register data communication was through the register file. The pipeline supported no forwarding paths. The mechanism is described briefly in the same IBM Journal of R&D issue as Tomasulo's algorithm, but the description is somewhat incomplete. The document *IBM 360/91's Out-of-Order Fixed-Point Pipe* on the class website describes the issue and commit mechanism (at least, my interpretation of it) in more detail.

Reorder Buffer

The reorder buffer was developed to solve the problem that, in many pipelined computers—even those with in-order instruction issue—execution results are frequently produced out-of-order. For example, this happens when functional units have different latencies. This is what Smith and Pleszkun mean by "pipelined" processors: in-order pipes with functional units that have different latencies, the most common pipeline of the paper's time. In previous machines, results were typically written to the register file as soon as they were produced, and if the results were produced out-of-order, they could therefore update machine state out-of-order. Such an organization causes problems in the case of handling precise interrupts, during which the machine state is required to reflect that of a sequential machine with in-order instruction completion (else the interrupt is not considered "precise"). Smith and Pleszkun solve the problem by providing a mechanism to allow instructions that *generate results* out-of-order to be *completed* in-order. Changes to the state of the machine (register file, memory system) are limited to the time of instruction completion, which is handled in program order, and therefore the state of the machine always reflects that of a sequential implementation.

The fundamental idea is this decoupling of instruction execution and instruction completion. Whereas in previous pipeline organizations execution and completion could be treated as an atomic multi-cycle operation, the reorder buffer separates out the concept of instruction completion as a phase of the instruction life cycle that may or may not happen on the cycle following the generation of results. Thus, the reorder buffer behaves like a holding tank for instructions while they are in the process of being executed (including decode, operand fetch, execution by an ALU, possible memory access, and writeback to the register file). It is easy to envision many possible structures that would perform such a function.

The reorder buffer described in the paper is a circular queue, in which each entry holds all the instruction state necessary to carry one instruction through the various phases of instruction execution (operand fetch, execution, register file writeback, etc.). Instructions are placed into the queue and taken out of the queue in-order. The original implementation enqueues instructions at the time of instruction issue—i.e. once all the operands are available. The paper describes two variations: one in which the operands must be obtained though the register file and another in which operands can be read directly out of the reorder buffer itself if the latest copy of a register value is present in the *result* portion of a reorder buffer entry holding an instruction that has finished executing but has not yet written its result to the register file. Though instructions are enqueued in program order and only when their operands are available, while they are in the reorder buffer program order does not dictate the sequencing of any particular events: in particular, the instructions might finish executing out of program order.

When an instruction is successfully dequeued from the reorder buffer, its results are *committed* to the machine state. At this point, the instruction is said to be *retired*. While an instruction is in the process of execution, i.e. before retiring, it may cause an exceptional condition (invalid opcode, TLB miss, segmentation violation, trap instruction, etc.). To ensure that such exceptions are not handled speculatively or out-of-order, the handling of exceptional conditions is treated like the updating of machine state: exception handling does not occur until instruction commit time. This ensures that no previous instructions have caused as-yet-unhandled exceptions. Therefore, all exceptions are handled in program order, and no exception is handled for an instruction that ends up being discarded (as a result of a branch misprediction, for example).

As with Tomasulo’s algorithm, the concept is very simple but extremely powerful. Consequently, it has appeared in nearly every high-performance architecture in the last half-decade. The paper describes two mechanisms in addition to the reorder buffer that perform the same function in slightly different ways: the future file and history buffer. Typically, these and other mechanisms that perform the function of ensuring in-order commitment of machine state are all called by the microprocessor design community “reorder buffers” whether the description is technically accurate or not.

Register Update Unit

Note that Tomasulo’s algorithm mentions nothing about precise interrupts. In fact, because it stipulates that results are to be sent to the register file as soon as they are produced, it cannot support precise interrupts without modification. Note also that the original reorder buffer paper mentions nothing about out-of-order issue to execution units. Though the mechanism may support such behavior, it is not addressed in the paper.

Why am I mentioning any of this? Though these two mechanisms (Tomasulo’s algorithm and the reorder buffer) solve very important problems, their existence pointed out a gaping hole that would be very important were it filled: the lack of a mechanism that combines both out-of-order issue and precise interrupts, which would provide supercomputer performance to general-purpose computers using modern operating systems facilities.

Out-of-order instruction issue is important because it allows instructions to begin executing as soon as they are ready, thereby freeing up the functional units as early as possible. Otherwise, artificial stall cycles are introduced into the pipeline, lengthening program execution. Tomasulo's algorithm provided a template for building a high performance machine with out-of-order issue and *imprecise* interrupts. However, support for *precise* interrupts became increasingly important roughly in step with the rising importance of operating systems offering multitasking and virtual memory—hardware support for precise interrupts greatly simplifies the implementation of both of these facilities (indeed, it is rather difficult to imagine an implementation of either multitasking or virtual memory *without* a precise-interrupt facility). When the reorder buffer was introduced, out-of-order instruction issue was typically found only in high-end scientific-computing machines; out-of-order completion was the problem more commonly encountered in general-purpose machines. However, the problem remained to implement out-of-order instruction issue in general-purpose machines that also needed precise interrupts.

A few years after the reorder buffer paper appeared, and twenty years after Tomasulo's algorithm appeared, the problem was solved. Sohi and Vajapeyam describe in their paper the marriage of Tomasulo's algorithm and the reorder buffer; they call their mechanism the *register update unit (RUU)*. This mechanism provides both out-of-order issue to functional units and support for precise interrupts.

The RUU is a very intuitive merger of Tomasulo's algorithm and the reorder buffer. It is a circular queue into which instructions are enqueued in-order and out of which instructions are dequeued in-order. Meanwhile, instruction operands are gathered according to Tomasulo's algorithm, and instructions are sent to functional units as soon as all their operands are valid. The mechanism as described in the paper only supports single-instruction enqueue/issue/commit; however, this is not inherent to its design—it is easily extended to wider implementations.

The primary place of deviation with reorder buffer operation is at instruction enqueue: first and foremost, an instruction is enqueued as soon as an RUU slot is available, whether the instruction's operands are available or not. Also, whereas in the reorder buffer scheme an instruction can read its operands out of the reorder buffer if available and out of the register file otherwise, in the RUU scheme an instruction never reads from other RUU entries directly. Rather, operands are gathered from the functional-unit result buses as in Tomasulo's algorithm, and they are also gathered from the commit bus, which carries the result of the currently committing instruction to the register file. Therefore, if at the time of instruction-enqueue the most recent version of one of its operands is found in the result field of another instruction's RUU entry, the operand does not become immediately available to the instruction as it would in the reorder buffer scheme, but it does become available as soon as the value is committed to the register file.

3. RiSC-16 Out-of-Order Implementation: Overview

The RiSC-16 out-of-order design is very much like Sohi & Vajapeyam's register update unit, except that it is twice as wide. The instruction encode/issue/commit logic is Tomasulo's algorithm extended to handle dual-enqueue, dual-commit, and three-way issue to the execute units (two ALU, one memory). The mechanism handles branch mispredictions and interrupts by placing instructions not in a disjoint set of reservation stations but rather in a circular instruction queue that functions similarly to Smith & Pleszkun's reorder buffer. Like the register update unit, the design departs from the reorder buffer in several ways. First, an instruction is enqueued as soon as an instruction-queue slot is available, whether the instruction's operands are available or not.

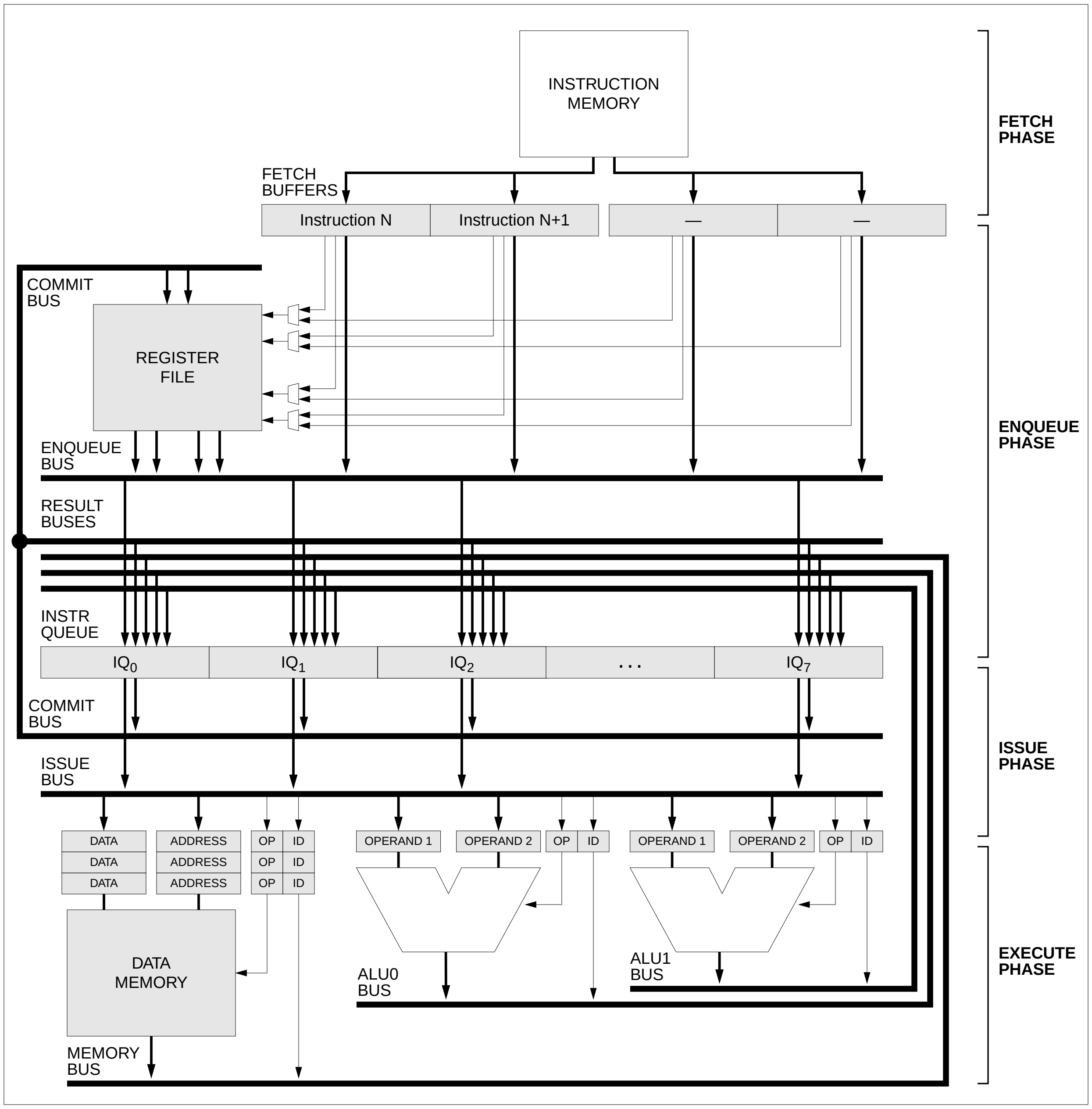


Figure 1: Overview of the pipeline organization

Second, instruction operands are not read from the instruction queue at enqueue time if the register file does not have the latest copy of a data word; rather, the instruction must wait until the data is available on the commit buses, the ALU buses, or the memory bus.

Because the computer-architecture field cannot seem to agree on definitions for issue/dispatch, we do not use the term “dispatch” and instead use the term “enqueue” to mean placing an instruction into the instruction queue and the term “issue” to mean sending a ready instruction to one of the functional units. This is illustrated in Figure 1. The architecture is two-way fetch, two-way enqueue, three-way issue and execute (two ALU instructions and one memory instruction), and two-way commit. Branch prediction is a simple backward-taken/forward-not-taken algorithm. The instruction queue has eight entries and is integrated with the memory queue; therefore, the system effectively has eight miss-status holding registers (MSHRs), enabling the handling of up to eight outstanding cache misses. However, this is artificially limited to three simultaneous requests being handled, to better reflect current DRAM design: there are three entries in the memory queue, which handles the memory interface and thus limits the number of simultaneous requests.

The figure shows four of the phases through which all committed instructions pass: fetch, enqueue, issue, and execute. Each of these takes one cycle. In this document, they are not called “stages” because that term implies more rigid timing and ordering, whereas instructions in an out-of-order core spend variable lengths of time in each stage and can visit some stages out of program order. The following sections describe each of the phases in more detail.

Fetch Phase

During the fetch phase, exactly two instructions are read from the instruction memory and placed into one of the two fetch buffers, each of which is wide enough to hold an instruction-fetch packet of two 16-bit instructions, plus the associated PC values for the instructions fetched. If no fetch buffer is available (e.g., if both of them are full or part full), then fetch stalls. Like the Alpha’s fetch/enqueue mechanism, there are two buffers, each as wide as a fetch packet, and the enqueue mechanism does not move on to the second fetch buffer until the first is completely drained.

Enqueue Phase

During the enqueue phase, up to two instructions are taken from one of the fetch buffers and placed at the tail of the instruction queue. The slots are designated “tail0” and “tail1” in the queue. If the first instruction in the buffer is a BEQ that is predicted taken (in this implementation, it is a simple backward-taken/forward-not-taken algorithm), the second instruction is squashed. If there are instructions in the other fetch buffer (the one that is currently not being considered for enqueue), those instructions are squashed as well. If the second instruction in the fetch buffer is a taken branch, only instructions in the alternate fetch buffer are squashed.

An example of two instructions being enqueued (the first of which is not a taken branch) is shown in Figure 2. The register-file access looks very similar to the decode logic in the single-issue in-order pipeline described in *The Pipelined RiSC-16*. However, the contents of the instruction-queue entry differ slightly from the contents of the pipelined RiSC’s ID/EX register. The main differences are the status bits intended to indicate where in the instruction life cycle this particular instruction is. The fields of the instruction-queue entry are shown in Figure 3.

When an instruction in one of the fetch buffers is enqueued or squashed (by a predict-taken branch), it is marked as invalid in the fetch buffer. Once both instructions are invalid, the enqueue mechanism looks at the alternate fetch buffer.

Data-Incoming Phase

When an instruction is enqueued, it may or may not have all of its operands. If it does, it can be sent immediately to the appropriate functional unit on the following cycle. Otherwise, it must wait in the instruction queue for its operands. During this phase, instructions scan the various result buses for their data. These buses include the ALU0/ALU1 Buses which have values returning from the ALUs, the Memory Bus which has load values returning from data memory, and the Commit Bus which has the values that are currently being written to the register file. If an

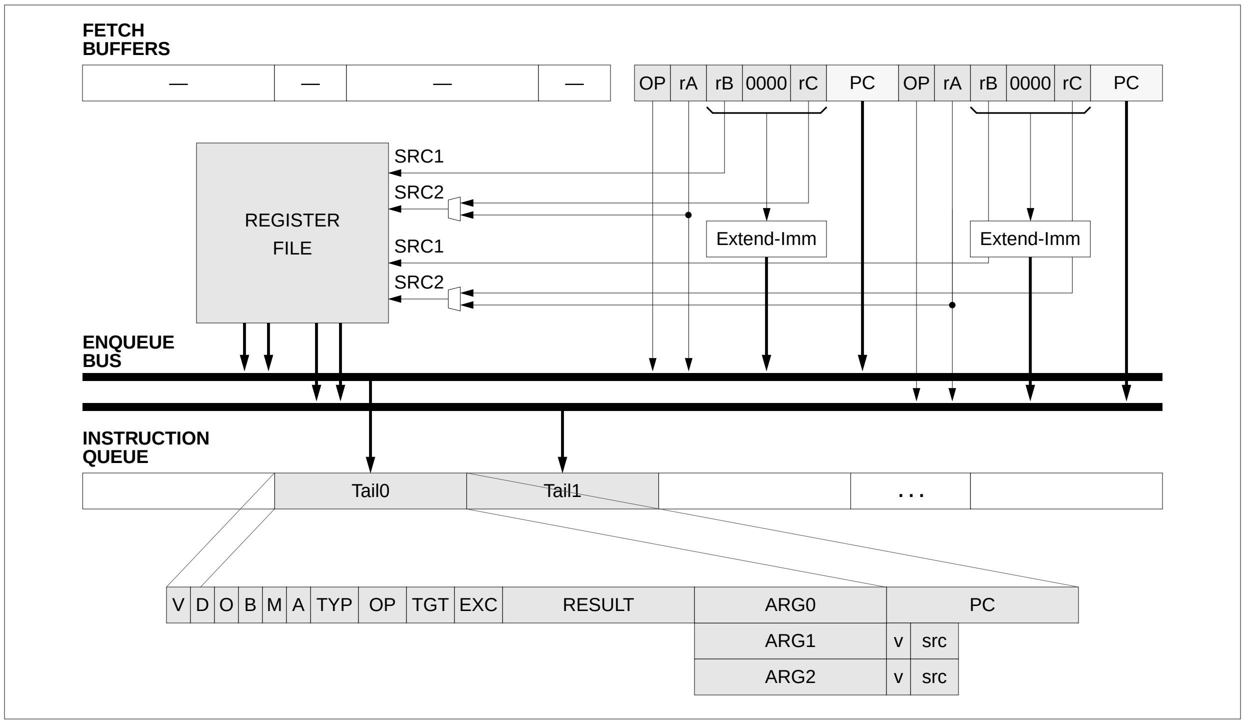


Figure 2: Example enqueue of two instructions

instruction sees that one or more of its operands is invalid and its **src** tag matches the **ID** of the data on one of these buses, it gates the associated data into its local operand storage.

Issue Phase

Once all of an instruction's operands are valid, its operands, opcode, and ID are sent to the input-registers of the appropriate functional unit. The instruction's opcode directs it to the appropriate bus: LW/SW instructions go to the memory queue; all other opcodes go to an ALU. The case when more than two ALU instructions or more than one memory instruction are ready to issue at the same time is covered in a later section on scheduling.

Memory operations are a special case of issue, because they actually issue twice: once to an ALU to generate the target address, and then to the memory queue when the address is known. Therefore, the issue logic considers an instruction ready to issue if all of its operands are valid, or if it is a memory operation and its ARG1 operand is valid. In this instance, the instruction is issued to an ALU as an **addi** instruction, and its results are gated-in to the ARG1 operand upon completion, thereby overwriting the previous valid contents. Upon completion of the **addi** instruction, the **A** bit in the instruction-queue entry is set, indicating that there is no need to re-issue the address-generation portion of the instruction. For load-word operations, the instruction is ready to be issued to the data memory immediately. For store-word operations, the ARG2 operand must also be valid, as it contains the value to be stored. Also, as described below, a store instruction must be issued at the time of commit.

Execute Phase

Issue and execute do not happen in the same cycle: it is a two-cycle process to obtain a result once all of an instruction's operands are valid. As described in the previous section, the instruction's

V	valid bit: signifies whether or not the entry's contents are valid
D	done bit: signifies whether or not the instruction has completed execution
O	out bit: signifies whether or not the instruction has been sent to a functional unit
B	branch bit: signifies that the instruction was predicted taken
M	memory-issue bit: signifies that the instruction is ready to be sent to the data memory
A	address bit: signifies that the instruction's memory address has been generated (only applies to LW/SW operations)
TYP	the instruction's "type" which indicates whether or not it touches memory, and whether or not it can cause a change in control-flow ... note that, since this can be deduced from the opcode, it is not strictly necessary
OP	instruction opcode
TGT	the instruction's register target, or zero if the instruction has no target (BEQ, SW)
EXC	the exception code, or zero if the instruction has caused no exception — note that HALT instructions place an EXC_HALT value directly into this register
RESULT	the register-file value that the instruction produces, if any
ARG0	the instruction's extended/shifted immediate value
ARG1	the instruction's first register operand
ARG1_v	valid bit for ARG1: indicates whether the value in ARG1 is valid
ARG1_src	source bit for ARG1: specifies the data-source for ARG1 (the location in the instruction queue of the instruction that will produce the value)
ARG2	the instruction's second register operand
ARG2_v	valid bit for ARG2: indicates whether the value in ARG2 is valid
ARG2_src	source bit for ARG2: specifies the data-source for ARG2 (the location in the instruction queue of the instruction that will produce the value)
PC	the address of the instruction, to be used if the instruction causes an exception or a branch-mispredict

Figure 3: FIelds of an instruction-queue entry

operands, opcode, and ID are moved to the ALU's input registers during the issue phase. On the following cycle the instruction is executed, and the resulting values are sent out on the various result buses to be latched at the end of the cycle. Memory operations take longer than a single cycle, so load results appear on the memory bus several cycles after they are initiated (this is an arbitrary choice — it is not inherent to the architecture).

In the instruction-queue entries, results are gated into the various operand registers as described above, but they are also gated into the RESULT registers of those instructions whose IDs are on the result buses. When this happens, the instruction is marked "done." The exception to this rule is the ALU result for the effective-address generation component of a memory operation: when this result appears on the bus, the instruction is not yet done.

Commit Phase

When an instruction has completed, it gates the contents of one of the various result buses into its RESULT register and sets the **done** bit. This signifies that the instruction is ready to commit its result to the "permanent" machine state: the register file and/or data memory.

As described in the section above on reorder buffers, this mechanism protects the machine from instructions that would be squashed because of exceptions or mispredicted branches: if an instruction commits its result to the register file and is later found to have immediately followed an instruction that causes an exception or a branch instruction that was mispredicted, it is very difficult to un-do the register-file update. It is even more difficult to un-do changes to main memory. Therefore, changes of this nature to the “permanent” machine state (as opposed to the contents of the instruction queue) are only allowed to occur once it is known that the instruction is non-speculative and definitely causes no exceptions.

On every cycle, the commit logic considers the top two instructions in the instruction queue: those at the head of the queue, labeled “head0” and “head1.” If there is a mispredicted branch in the machine, commit does not proceed, unless the mispredicted branch is in the head1 slot or later. If head0 is ready to commit, it does so. If head0 and head1 are both ready to commit, both do so. Otherwise, nothing happens.

When an instruction commits, its result is sent to the register file and made available to other instructions needing operands. If the instruction is a SW, it is sent to the memory system.

If an instruction that would otherwise be allowed to commit causes an exception, indicated by a non-zero value in the **EXC** field of its instruction-queue entry, the machine reacts just like a branch-mispredict event: the program counter is redirected; the exceptional instruction’s program counter, held in the instruction-queue entry, is saved in a hardware register; the pipeline is flushed from the exceptional instruction to just before tail0; and execution begins with the first instruction in the exception handler. Exception-handling is covered in more detail later.

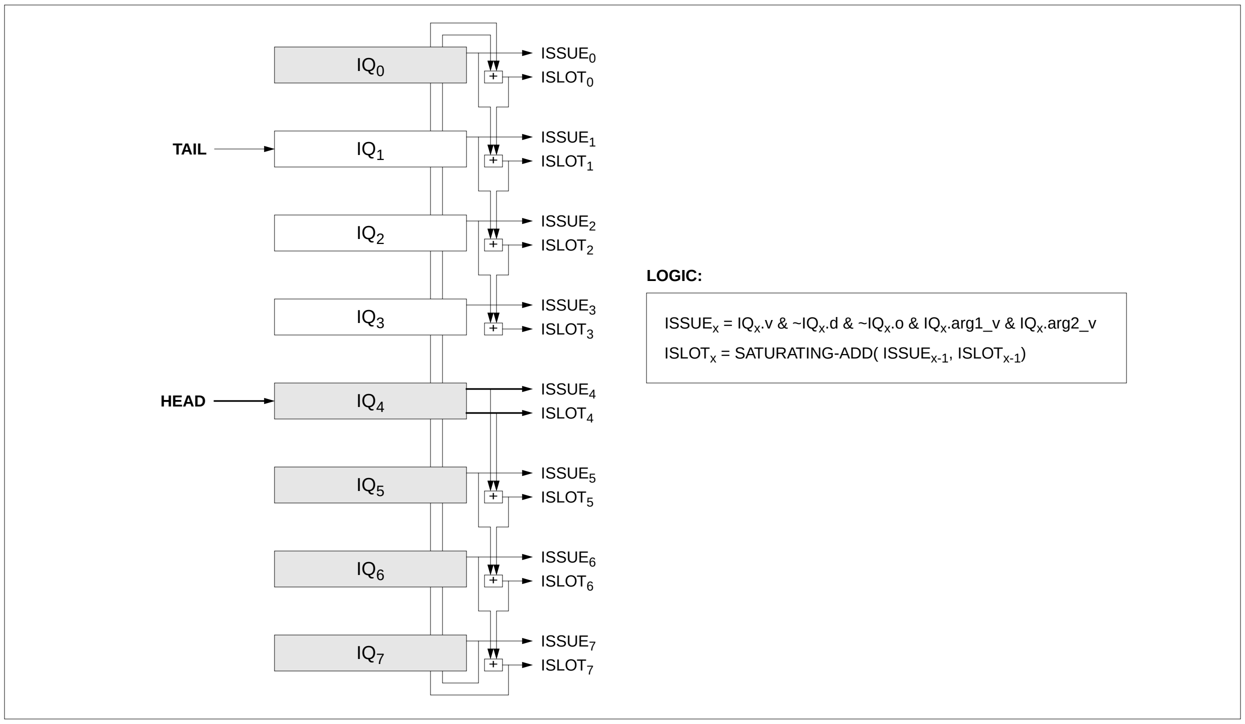
4. Mechanisms that Require a Little More Detail

The previous section gives a high-level overview of what is going on in the pipeline, but there are a few details that are left out for brevity and clarity. This section delves into some of these details.

Instruction Scheduling

Instruction scheduling is the process of assigning ready instructions to functional units. In this implementation, all non-memory functional units (i.e., ALUs) are identical, which simplifies things immensely. However, there are only two ALUs, and on any given cycle it is possible for more than two instructions to be ready for execution. This is where instruction-scheduling comes in. It is the logic that prioritizes instructions. Traditionally, instructions are prioritized by age: the oldest instructions in the pipeline should execute before newer instructions because older instructions are more likely to be holding up other instructions in the pipe, either through dependencies or just by the fact that some instructions are ready to commit but cannot until an older instruction finishes execution.

Figure 4 gives a stylized logic diagram of the instruction-scheduling mechanism. The boxes represent instruction-queue entries, and shaded boxes indicate the presence of valid instructions. Instruction-queue entries put out two signals: *issue* and *islot*. The 1-bit *issue* signal signifies that the instruction is ready to issue. The signal is high if the instruction-queue entry contains a valid instruction, the instruction is not done executing, the instruction is not currently “out” (in the process of being executed), and its operands are both valid. The 2-bit *islot* signal is the ID of the ALU for which the instruction is destined, or an invalid ALU number if there are too many instructions ready to issue. If an instruction is putting out a valid slot number and its issue signal is high, then that instruction is sent to the input-registers of the indicated ALU. Note that the scheme requires logic equivalent to saturating adders—otherwise, if there are enough instructions ready to issue, successive adds could yield a valid ALU ID for a low-priority instruction, even after an invalid ALU number has been produced.



IQ_0
IQ_1
IQ_2
IQ_3
IQ_4
IQ_5
IQ_6
IQ_7
TAIL
HEAD
$ISSUE_0$ … $ISSUE_7$
$ISLOT_0$ … $ISLOT_7$
+
LOGIC:
$ISSUE_x = IQ_x.v \ \& \ \sim IQ_x.d \ \& \ \sim IQ_x.o \ \& \ IQ_x.arg1\_v \ \& \ IQ_x.arg2\_v$
$ISLOT_x = SATURATING\text{-}ADD(ISSUE_{x-1}, ISLOT_{x-1})$

Figure 4: Instruction scheduling logic

Note that the critical path scales as O(*n*) for *n* entries in the instruction queue. This is clearly a problem, as instruction windows are increasing in size—we are currently at the several-dozen mark and pushing rapidly toward the several-hundred mark. A recent ISCA paper by Henry, et al. looked at ways to reduce this to O(log *n*), which is obviously much better.

An optimization: because the issue/execute process is a two-cycle operation from the moment an instruction's operands become valid, a chain of *n* dependent instructions would take 2*n* cycles to execute. This is not particularly efficient. An improvement is to allow data on the ALU result buses to be latched back into the ALU input-registers directly. To accomplish this, an instruction must recognize when a value will be available on an ALU bus on the following cycle and set its *issue* signal high when it sees that the ID of an instruction currently in the execute phase is the same as one of its operand **src** values.

Memory Operations

The decision to allow an instruction to proceed to the memory subsystem is very similar to instruction scheduling. The difference is that only three memory requests can be in transit at once. This corresponds roughly to the design of today's DRAM architectures (e.g. Direct Rambus) that allow pipelined requests to memory but can handle a maximum of three requests in the pipe simultaneously. Thus, while the instruction-scheduling mechanism need only ensure that the instruction issue-width (two) is never exceeded, the memory-scheduling mechanism must ensure that the memory issue-width (one) is never exceeded and must also ensure that the maximum degree of concurrency (three) is never exceeded.

In this implementation, the memory-scheduling is done in two cycles: in the first cycle, once the instruction's operands are ready, the instruction's address is compared to those of all preceding memory instructions. If there are no address conflicts with preceding instructions, and if no preceding instructions are ready to issue to the memory queue, the instruction is marked as ready to issue to the memory queue. The memory request is then sent to the 3-entry memory queue on the following cycle.

An address conflict occurs whenever the reordering of two accesses might cause a problem, i.e. when they access the same memory location and one of them is a store instruction. Loads are allowed to bypass any store instruction whose target address is known to be different and are allowed to bypass any loads whether the target address of the bypassed load instruction is known or not. Loads are not allowed to bypass store instructions whose addresses are not yet known; there is a chance that the store's address might end up being the same. Stores are allowed to bypass one another once their addresses are known to be different. Note that, while this memory architecture works well in a uniprocessor setting, it can cause enormous problems in a multiprocessor setting. Note also that many more aggressive implementations exist that allow speculative bypassing and then, at a later point, repair any damage done if it is determined that an address conflict occurred between bypassed memory operations.

Branch Mispredictions

Though branch mispredictions and precise interrupts require virtually identical support, there are a few differences. Here is one of the most significant examples: when the pipeline is flushed because of an exception, the entire instruction queue is flushed, whereas only a portion of the instruction queue is flushed when a branch mispredict is detected.

Why is this important? It is not particularly difficult to delete only a subset of the instruction queue's entries. The difficulty comes in maintaining the coherence of the register file: if all the instructions in the instruction queue are flushed, the entire contents of the register file can immediately be set to **valid** because all the outstanding instructions have been cancelled. This causes no inconsistencies because the register file is only updated on instruction commit, and therefore its contents are always valid, up to and including the last committed instruction.

However, when a branch mispredict is detected, only a portion of the instructions in the queue are flushed; therefore, the state of the register file is partially valid, partially invalid. The core must determine very quickly which subset of the register file's contents are valid, and, for each register that remains invalid, the processor must determine the ID of the latest non-squashed instruction that will update that register.

Take, for example, the following instruction-queue contents:

```
HEAD:  iq3    add r1, r2, r3
       iq4    nand r3, r4, r5
       iq5    nand r4, r5, r6
       iq6    add r1, r3, r4
       iq7    beq r1, r0, foobar
       iq0    add r1, r4, r5
TAIL:  iq1
```

Assume that the **beq** instruction was mispredicted; therefore upon detection of the misprediction all following instructions are removed from the instruction queue. Thus, the **add** instruction in instruction-queue slot **iq0** is deleted. At the point the misprediction is detected but before it is resolved, the state of the register file indicates that **r1** is invalid and that its source is this very instruction (the **add** instruction in **iq0**). Recovering from the mispredicted branch means that the register file must retain the **invalid** status on **r1**, but its source ID should become **iq6**.

This is non-trivial, but it amounts to solving the same problem as instruction scheduling, except that older instructions are given a lower priority, not a higher priority. Sohi & Vajapeyam's solution was to use counters associated with each register; these counters indicated the number of instructions in the RUU that targeted a given register, and they were used as the ID field instead of using a reservation station number (or, as in the RiSC-16 implementation, the instruction queue entry number). The solutions are functionally equivalent.

Precise Interrupt Handling

As mentioned in the introduction, the goal is to support precise interrupts and provide a simple TLB-miss handling facility. The initial implementation does not provide such support, but this section gives an overview of the proposed interrupt-handling facility.

At instruction-commit time, a check is made to see if an instruction causes an exception. This is a simple check of the instruction's **EXC** field; if this field is non-zero, the instruction caused an exception. Once it is verified that the instruction will commit (i.e. once it is verified that none of the instructions before it cause exceptions or unexpected changes in control flow), the exception handler takes over. For backward compatibility with implementations that do not handle interrupts, interrupt handling is disabled by default. If interrupt handling is enabled, the hardware expects that there is an *exception vector table* in memory at location 0x0000 holding sixteen jump vectors: one for each interrupt type. The following interrupt types are the ones defined so far:

```
#define EXC_NONE        0
#define EXC_HALT        1
#define EXC_TLBMISS     2
#define EXC_SIGSEGV     3
#define EXC_INVALID     4
```

When an instruction causes an exception, that fact is recorded in its instruction-queue entry. At the point when the instruction would normally commit, the exception number is used as an index into the exception vector table to find a jump address. The pipeline is flushed and resumes execution at that location — and, if supervisor mode is implemented, privileges (i.e. supervisor mode) must be enabled. The program counter of the exceptional instruction is held in a hardware register to allow a later return of control to that point, if desired.

When the exception or interrupt handler is finished, control returns to the application, assuming that the exception is non-terminal, as in the case of most TLB misses. The mechanism provided by most architectures is a *return-from-exception* instruction that jumps to the exceptional instruction (or to the one after it, in the case of a *system-call* instruction) and at the same time returns the processor to user mode. The RiSC-16 does exactly this.

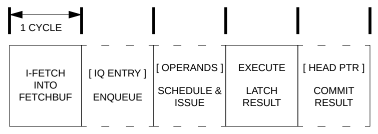
5. Pipeline Timing

As an instruction may spend a variable length of time in many of its phases, different instructions can have different latencies. This is expected, as the architecture is not a rigid N-stage pipeline like the MIPS/DLX. Moreover, different *classes* of instructions will most certainly have different latencies, because they require the use of different functional units that have different latencies, and some instructions require the use of multiple functional units in series. This section describes the timing behavior of the various instruction classes; the next section then illustrates the movement of instructions and their related status information through the pipeline.

In the figures, dashed lines indicate phases that can take one or more cycles.

ALU Instructions

ALU-type instructions (**add**, **addi**, **nand**, **lui**) have the following timing:

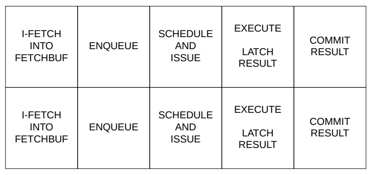


The words in square brackets represent potential reasons for stalling. In the enqueue phase (second cycle), an instruction can wait arbitrarily long for an open instruction-queue entry, and, once an entry is available (tagged as invalid), the enqueue process takes one cycle. In the issue phase (third cycle), an instruction can wait arbitrarily long for its operands, and, once the operands are valid, the issue process takes one cycle. In the commit phase (last cycle), an instruction can wait arbitrarily long for the head pointer to come around, signifying that the instruction is in the next block of instructions to commit, and, once the instruction is marked “done” and all preceding instructions have committed, the commit process takes one cycle.

These are simple instructions, requiring a single cycle of execution in an ALU. They all update the register file. They can have two register operands (**add**, **nand**), one register operand (**addi**), or no register operands (**lui**). As with any other type of instructions, more than one may be executed and committed simultaneously if there are no inter-instruction dependencies. For example, the following code:

```
add    r1, r2, r3
nand   r4, r5, r6
```

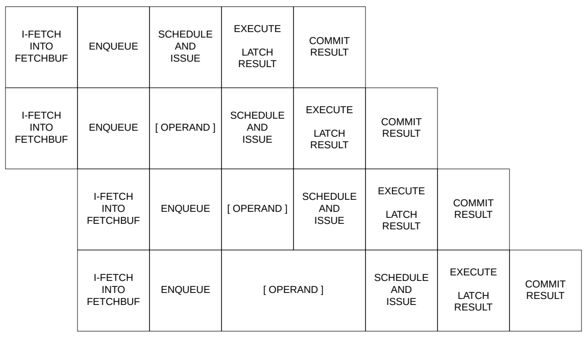
has the following timing, assuming the necessary operands are available in the register file at instruction enqueue:



Because of the optimization described earlier, a chain of dependent instructions is executed by the processor core at a rate of one instruction per cycle. For example, the following chain:

```
lui    r1, 0xabcd
lli    r1, 0xabcd
add    r1, r1, r2
nand   r1, r1, r3
```

has the following timing:

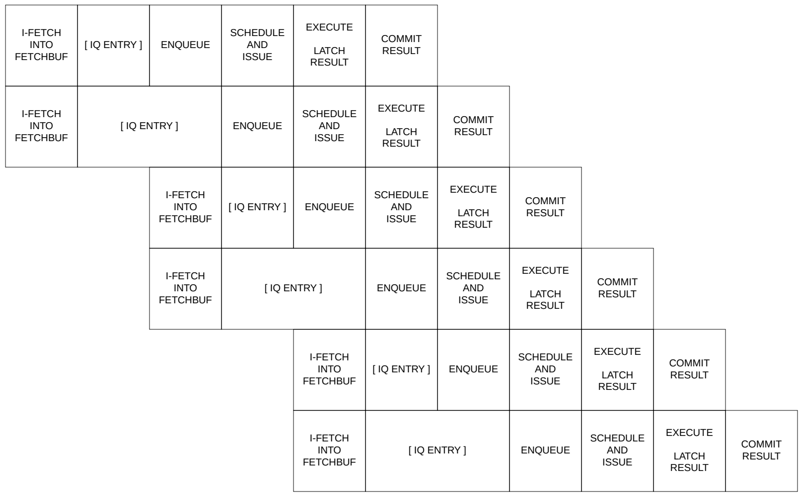


Note that the latency is increasing for each successive instruction. Clearly, this pattern is not sustainable. At some point, in a long line of dependent instructions, the instruction queue fills up, restricting the enqueue mechanism to one instruction per cycle, and the steady-state yields a single-instruction throughput.

For example, the following code:

```
...
addi   r1, r1, 1
add    r1, r1, r2
nand   r1, r1, r3
addi   r1, r1, 1
add    r1, r1, r2
nand   r1, r1, r3
addi   r1, r1, 1
add    r1, r1, r2
nand   r1, r1, r3
...
```

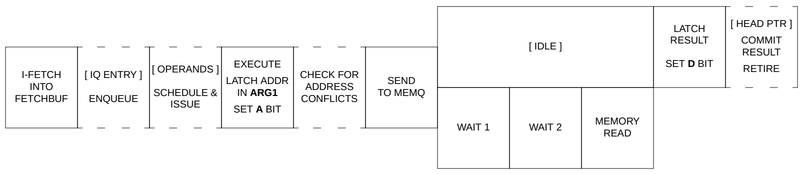
yields the following timing:



The reason that every instruction stalls waiting for an IQ entry is that the alternate fetch buffer is filled as soon as it is emptied—e.g., as soon as the second instruction is enqueued, the third fetch (getting the 5th and 6th instructions) begins.

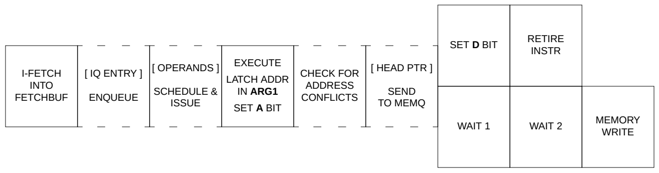
Memory Instructions

Load instructions have the following timing:



After the target address is generated, the memory-scheduling logic compares the target address to that of every instruction earlier in the queue. This phase (“check for address conflicts”) can take an arbitrary amount of time until the instruction queue is free of conflicts. When conflicts have been resolved, the request is sent to the memory queue, illustrated by a separation of the two instruction paths. The memory system returns the requested data three cycles later, to be latched on the fourth cycle. When the data returns, the **done** bit is set and the instruction can commit.

Store instructions have slightly different timing:

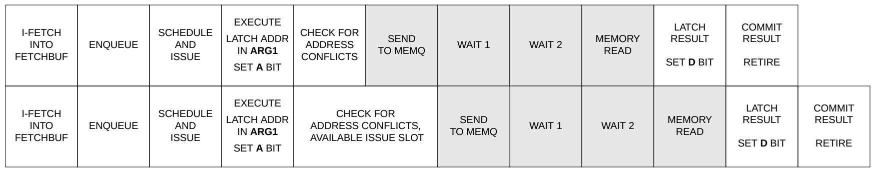


The primary difference is that stores wait to send anything to the memory system until it is known that they will definitely commit. Therefore, the wait for the head pointer to come around happens earlier in the life cycle. Once the request has been handed off to the memory queue, there is no reason for the instruction to remain in the instruction queue, and so it is committed immediately.

Here are the timings for several different combinations of memory instructions. First, two independent memory operations fetched at the same time, whose operands are available in the register file at enqueue time. This shows the stall cycle introduced to the second instruction that is due to the maximum issue width of memory operations (one per cycle). For example, the following code:

```
lw     r1, r0, foo1
lw     r2, r0, foo2
```

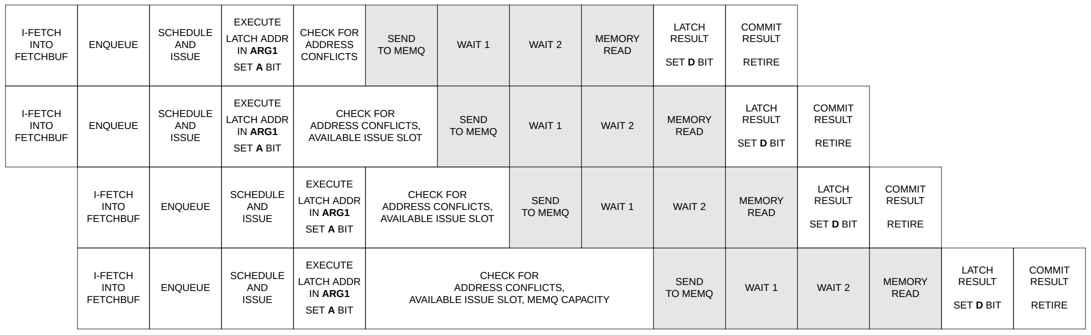
has the following timing, with the shaded boxes indicating the cycles during which the instruction is occupying a slot in the memory queue:



Next, we look at a series of independent instructions that exceeds the memory queue's capacity. Here, memory instructions stall not because of exceeding the memory-issue width (one) but of exceeding the memory queue's capacity of simultaneous instructions (three). The following code:

```
lw     r1, r0, foo1
lw     r2, r0, foo2
lw     r3, r0, foo3
lw     r4, r0, foo4
```

has the following timing, the shaded boxes representing cycles during which instructions occupy slots in the memory queue:



Next, we look at the timing for an instance where a store instruction depends on the data returned from a load instruction. This shows the store stalling during the operand fetch phase. The following code:

```
lw     r1, r0, foo1
sw     r1, r0, foo2
```

has the following timing:

I-FETCH INTO FETCHBUF	ENQUEUE	SCHEDULE AND ISSUE	EXECUTE LATCH ADDR IN **ARG1** SET **A** BIT	CHECK FOR ADDRESS CONFLICTS	SEND TO MEMQ	WAIT 1	WAIT 2	MEMORY READ	LATCH RESULT SET **D** BIT	COMMIT RESULT RETIRE				
I-FETCH INTO FETCHBUF	ENQUEUE	SCHEDULE AND ISSUE	EXECUTE LATCH ADDR IN **ARG1** SET **A** BIT	[WAIT FOR DATA OPERAND]					LATCH DATA OPERAND	CHECK FOR ADDRESS CONFLICTS	SEND TO MEMQ	SET **D** BIT	RETIRE INSTR	MEMORY WRITE

Last, we look at the timing for a pair of memory instructions where the first is a load-word and the second uses the result of that load for its target address. The second instruction could be a load or store instruction. This shows the second instruction stalling during the address-generation phase. The following two code examples produce identical timing (the difference is that the second code example shows the **sw** dependent on the **lw** through **r1** for both address and data):

```
lw     r1, r0, foo1
sw     r0, r1, foo2

lw     r1, r0, foo1
sw     r1, r1, foo2
```

The timing is shown below:

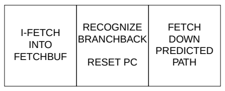
I-FETCH INTO FETCHBUF	ENQUEUE	SCHEDULE AND ISSUE	EXECUTE LATCH ADDR IN **ARG1** SET **A** BIT	CHECK FOR ADDRESS CONFLICTS	SEND TO MEMQ	WAIT 1	WAIT 2	MEMORY READ	LATCH RESULT SET **D** BIT	COMMIT RESULT RETIRE						
I-FETCH INTO FETCHBUF	ENQUEUE	[WAIT FOR ADDRESS OPERAND]							LATCH ADDR OPERAND	SCHEDULE AND ISSUE	EXECUTE LATCH ADDR IN **ARG1** SET **A** BIT	CHECK FOR ADDRESS CONFLICTS	SEND TO MEMQ	SET **D** BIT	RETIRE INSTR	MEMORY WRITE

Note that, compared to ALU instructions, there is an extra cycle between the moment that the necessary operand is produced and the moment that the store instruction is issued to the ALU for its address generation phase. This is the extra cycle between the "latch result" cycle of the load instruction and the "execute" cycle of the store instruction after it—the cycle in which the store instruction does "schedule and issue" operations. If the first instruction were an ALU-type instruction and not a load-word, the store would schedule and issue during the cycle currently marked "latch addr operand." This is not a mistake; the optimization described earlier that allows dependent ALU instructions to issue on successive cycles does not apply to operands appearing on the memory bus. When an instruction obtains an operand from the memory bus, it does not schedule itself until the following cycle.

Branches and Jumps

Conditional branches that are predicted not-taken look just like ALU-type instructions: they collect their operands and are issued to the functional units to verify the prediction. Meanwhile, the program counter simply increments as with a regular instruction. Assuming the prediction is correct, pipeline timing is not affected.

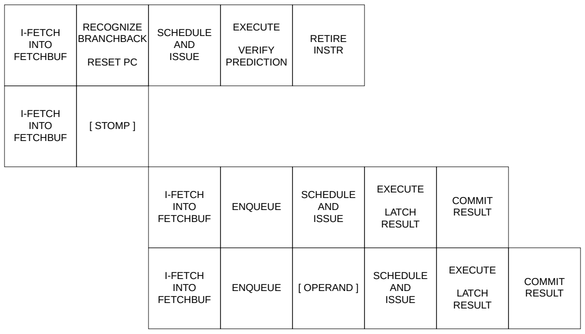
Conditional branches that are predicted taken are resolved in the enqueue phase. While the **beq** instruction is sitting in one of the fetch buffers, the predicted target address is generated and placed into the program counter. Instruction fetch down the predicted path begins on the following cycle. Thus, there is a one-cycle penalty for predicted-taken branches. This is pretty standard for architectures without branch-target buffers. The timing:



For example, assume that the PC currently points to the **beq** instruction. The following code:

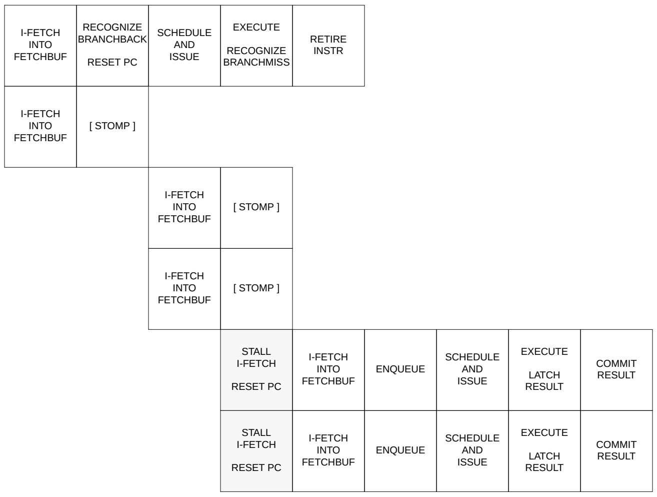
```
back:  add    r2, r3, r4
       nand   r5, r2, r6
       add    r1, r2, r3
       addi   r1, r1, 5
       ...
       beq    r0, r1, back  // PC starts here
       add    r2, r3, r4
```

has the following timing, assuming that the branch is predicted correctly:



Branch mispredictions are resolved in the execute phase, where the functional unit compares the operands and sets the program counter appropriately if it is determined that the branch direction taken was not appropriate. Note that, in this implementation, there is no branch target buffer, so we need not resolve instances where the branch *target* was mispredicted, which also must be accounted for in the case where both the direction and the target are speculative.

The following diagram gives the timing for a mispredicted branch instruction, using the code example above. Assume the PC is pointing at the **beq** instruction.



Jump-and-link instructions are implemented just like branch misspeculations; they are resolved in the execute phase, at which point the target address is known. The timing therefore looks much like the mispredicted branch example, above: during the execute phase, the target address for the JALR is known, and the *branchmiss* signal is set, just as if the instruction were a misspeculated branch. The program counter is redirected during this cycle, and instruction-fetch down the correct path resumes on the following cycle.

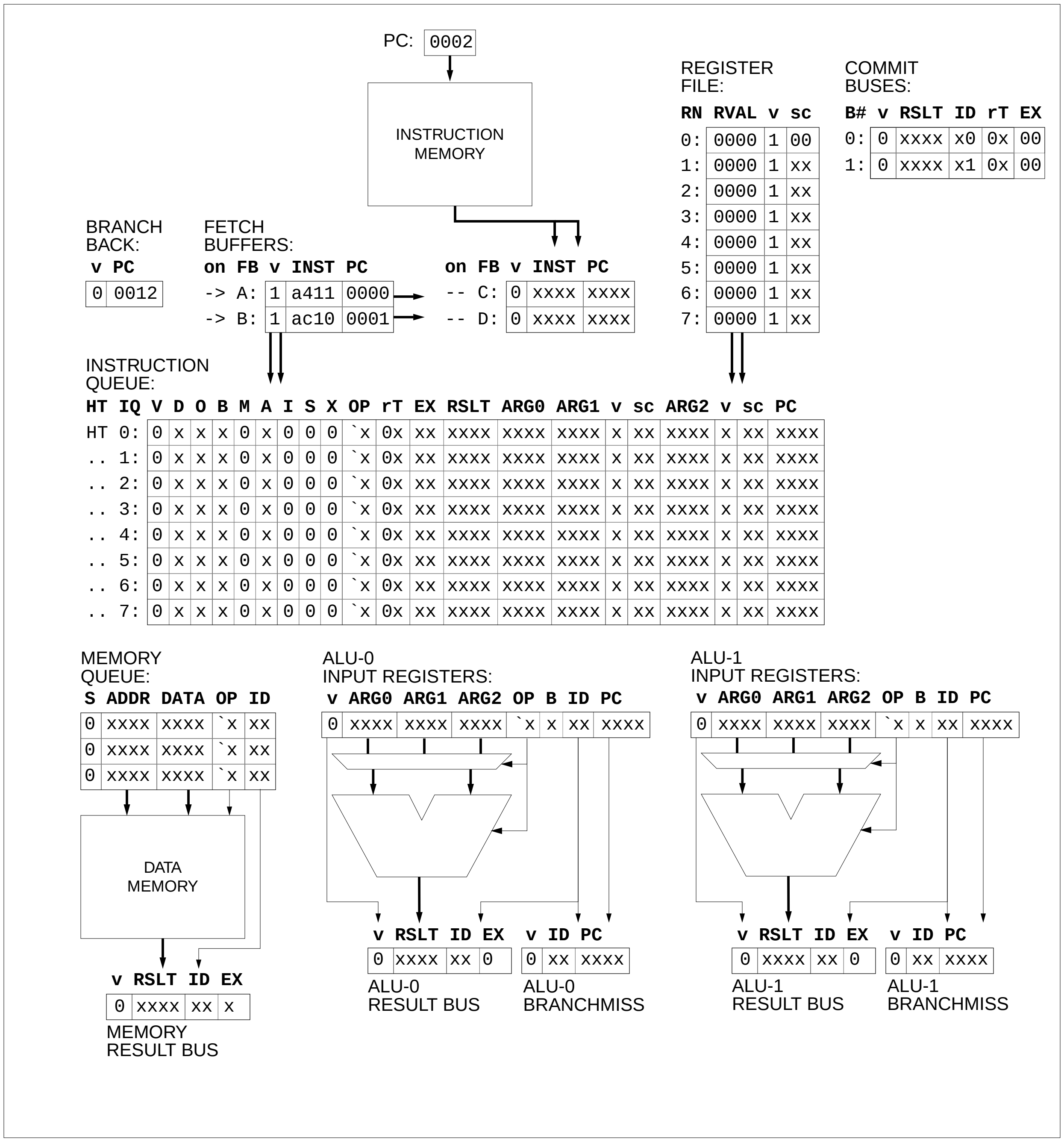
6. Example Operation

The following figures illustrate (in excruciating detail) the movement of instructions, data, and status information through the pipeline during the execution of a relatively simple piece of code. This is done to animate the design, hopefully giving a clear picture of what happens in the machine. The following code example is used (addresses are included for clarity):

```
#
# main loop: loads a number and then a variable number of data items to subtract
# from the first. at end, saves result in “diff” memory location
#
0000                lw     r1, r0, arg1
0001                lw     r3, r0, count
0002         loop:  lw     r2, r4, arg2
0003/4              movi   r7, sub             # resolves to 2 instructions
0005                jalr   r7, r7
0006                addi   r3, r3, -1
0007                beq    r3, 0, exit
0008                addi   r4, r4, 1
0009                beq    r0, r0, loop
000a         exit:  sw     r1, r0, diff
000b                halt
#
# subtract function: operands in r1/r2, return address in r7. result -> r1
#
000c         sub:   nand   r2, r2, r2
000d                addi   r2, r2, 1
000e                add    r1, r1, r2
000f                jalr   r0, r7
#
# data: count is the # of items to subtract from arg1 (in this case, 1: arg2)
# diff is where the result is placed
#
0010         count: .fill  1
0011         arg1:  .fill  9182
0012         arg2:  .fill  737
0013         diff:  .fill  0
```

The execution takes 29 cycles and illustrates many of the possible behaviors: ALU operations, memory operations, BEQ instructions predicted-taken and predicted-not-taken, BEQ instructions predicted correctly and incorrestly, JALR instructions (which use the branch-miss facility and behave like a mispredicted BEQ), instruction-enqueue of 0, 1, and 2 instructions, the filling up of the instruction queue thereby blocking enqueue and fetch, retirement of instructions, etc. Only the first dozen cycles are shown; the remainder are given in the *RiSC-oo.1.v Execution Example*.

The state of the machine at the start of each cycle is shown in Figures 5–16. Dark lines indicate movement of data (which is latched at the end of the cycle and is visible in machine state on next cycle). The top bit of the instruction ID indicates the result bus to watch: memory bus vs. ALU bus. For instance, a LW enqueued in slot 3 will tag the register file with id 13 rather than 03; this notifies other instructions not to latch the results of the LW’s add-immediate operation that simply generates the target address. Opcode values are prefixed with “a” indicating ALU instructions, “b” indicating branch operations, or “m” for memory operations. Non-obvious fields of the IQ entry (not all are fields; some are just signals): *Valid, Done, Out, Branch-taken, Memory-issuable, Address-generated, Issuable* (to ALU), *Slot-number,* and *X = kill the instruction*. The figures start



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	1	xx
2:	0000	1	xx
3:	0000	1	xx
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	1	xx

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	xxxx	x0	0x	00
1:	0	xxxx	x1	0x	00

BRANCH BACK:

v	PC
0	0012

FETCH BUFFERS:

on	FB	v	INST	PC
->	A:	1	a411	0000
->	B:	1	ac10	0001

on	FB	v	INST	PC
--	C:	0	xxxx	xxxx
--	D:	0	xxxx	xxxx

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
HT	0:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	1:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	2:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	3:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	4:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	5:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	6:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	7:	0	x	x	x	0	x	0	0	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	xxxx	xxxx	xxxx	`x	x	xx	xxxx

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	xxxx	xxxx	xxxx	`x	x	xx	xxxx

v	RSLT	ID	EX
0	xxxx	xx	x

MEMORY RESULT BUS

v	RSLT	ID	EX
0	xxxx	xx	0

ALU-0 RESULT BUS

v	ID	PC
0	xx	xxxx

ALU-0 BRANCHMISS

v	RSLT	ID	EX
0	xxxx	xx	0

ALU-1 RESULT BUS

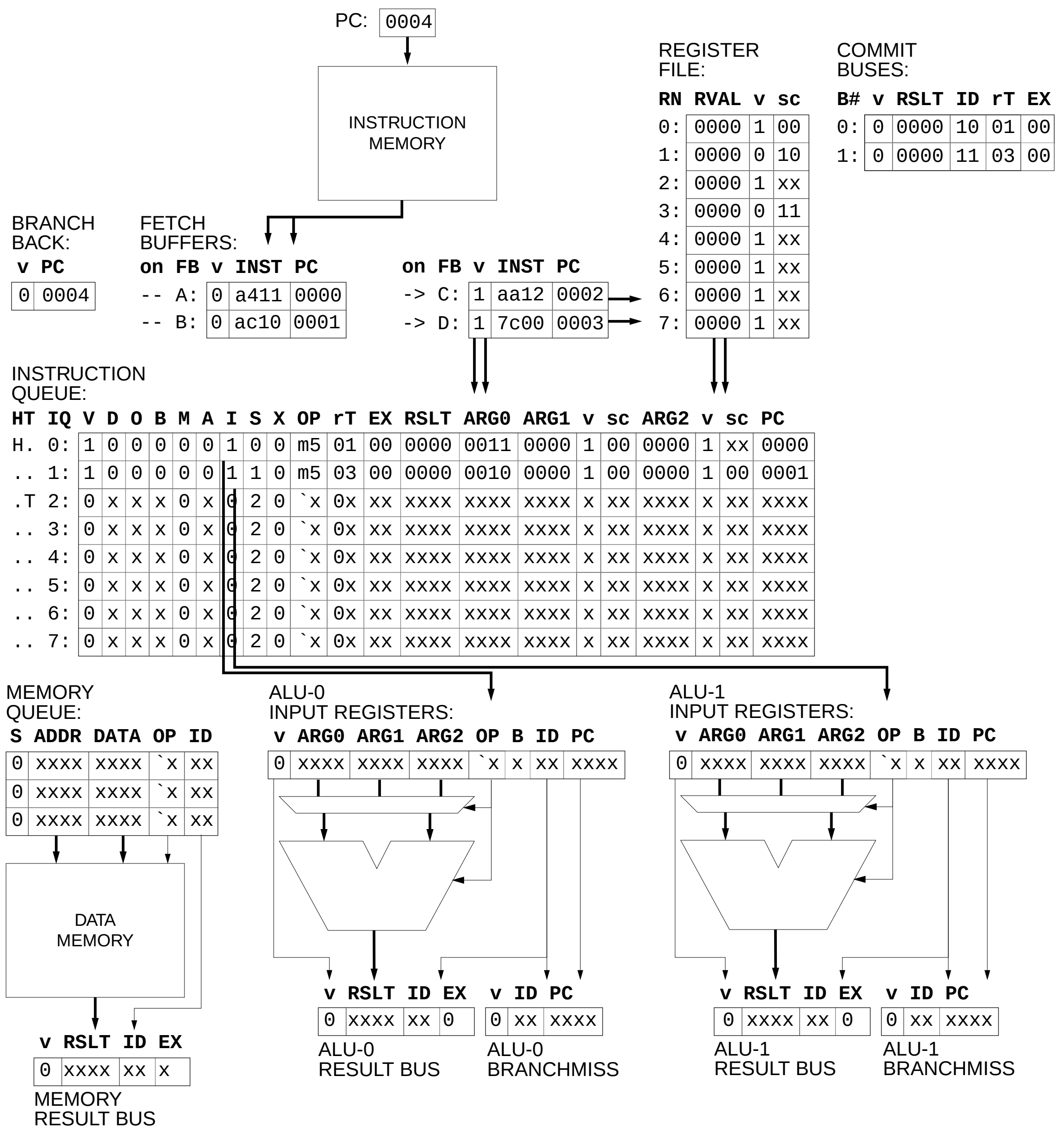
v	ID	PC
0	xx	xxxx

ALU-1 BRANCHMISS

Figure 5: EXECUTION CYCLE 2

on cycle 2; during cycle 1 we fetched instructions at pc=0000 and pc=0001 (two LW instructions) into the two first fetch buffers (A and B) and incremented the program counter by 2.

During execution cycle 2, we enqueue the first two instructions into the slots indicated by the tail pointer (labeled “T” at the side of the instruction queue); this includes reading from the register file: the register value RVAL, its valid bit V, and its source SC. During this phase, the targets of the two LW instructions (r1 and r3) are tagged “invalid” and their SC fields directed to the two LW instructions. Note the top bits of these IDs are “1”, indicating that the final result will come from the memory bus, not an ALU bus. We also fetch the next two instructions (an LW and a LUI—a



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	1	xx
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	1	xx

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0000	10	01	00
1:	0	0000	11	03	00

BRANCH BACK:

v	PC
0	0004

FETCH BUFFERS:

on	FB	v	INST	PC
--	A:	0	a411	0000
--	B:	0	ac10	0001
->	C:	1	aa12	0002
->	D:	1	7c00	0003

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
H.	0:	1	0	0	0	0	0	1	0	0	m5	01	00	0000	0011	0000	1	00	0000	1	xx	0000
..	1:	1	0	0	0	0	0	1	1	0	m5	03	00	0000	0010	0000	1	00	0000	1	00	0001
.T	2:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	3:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	4:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	5:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	6:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	7:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	xxxx	xxxx	xxxx	`x	x	xx	xxxx

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	xxxx	xxxx	xxxx	`x	x	xx	xxxx

ALU-0 RESULT BUS:

v	RSLT	ID	EX
0	xxxx	xx	0

ALU-0 BRANCHMISS:

v	ID	PC
0	xx	xxxx

ALU-1 RESULT BUS:

v	RSLT	ID	EX
0	xxxx	xx	0

ALU-1 BRANCHMISS:

v	ID	PC
0	xx	xxxx

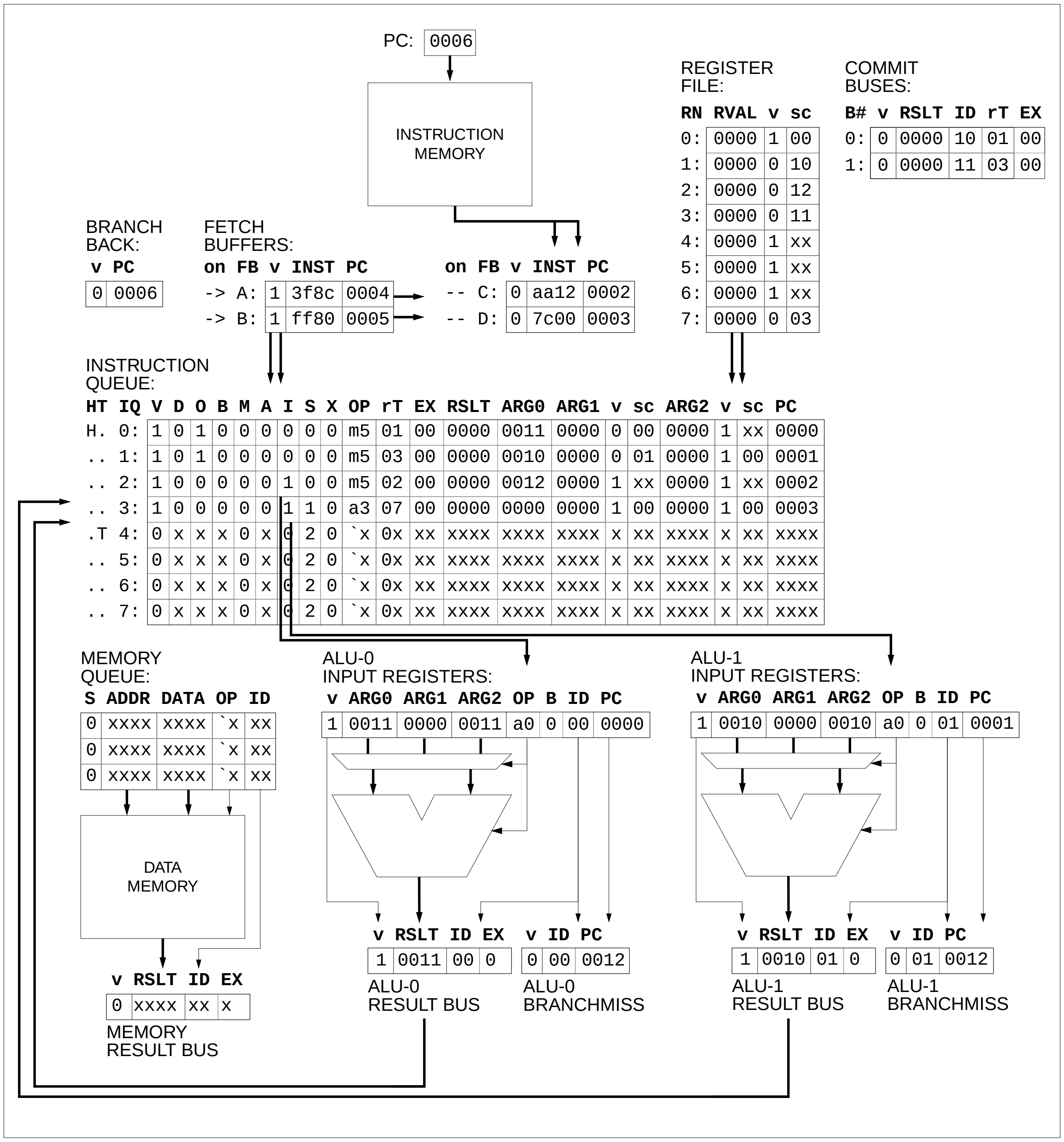
MEMORY RESULT BUS:

v	RSLT	ID	EX
0	xxxx	xx	x

Figure 6: EXECUTION CYCLE 3

MOVI is replaced by the assembler with a LUI+ADDI pair) into the alternate fetch buffers (C and D). The program counter is incremented by two.

During cycle 3, the first two instructions issue address-generate operations to the ALUs; we enqueue the second two instructions; and we fetch the third pair of instructions: an ADDI and a JALR. The program counter is incremented by two. During the enqueue phase, the targets of the two instructions (LW -> r2, LUI -> r7) are set appropriately: the SC field for r2 will become “11” and the SC field for r7 will become “03”, indicating that LUI’s result will be on an ALU bus. Note that the **v/src** fields in each of the two issuing LW instructions will become invalid and refer to the



PC: 0006

REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	12
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	03

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0000	10	01	00
1:	0	0000	11	03	00

BRANCH BACK:

v	PC
0	0006

FETCH BUFFERS:

on	FB	v	INST	PC
->	A:	1	3f8c	0004
->	B:	1	ff80	0005

on	FB	v	INST	PC
--	C:	0	aa12	0002
--	D:	0	7c00	0003

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
H.	0:	1	0	1	0	0	0	0	0	0	m5	01	00	0000	0011	0000	0	00	0000	1	xx	0000
..	1:	1	0	1	0	0	0	0	0	0	m5	03	00	0000	0010	0000	0	01	0000	1	00	0001
..	2:	1	0	0	0	0	0	1	0	0	m5	02	00	0000	0012	0000	1	xx	0000	1	xx	0002
..	3:	1	0	0	0	0	0	1	1	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
.T	4:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	5:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	6:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	7:	0	x	x	x	0	x	0	2	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
1	0011	0000	0011	a0	0	00	0000

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
1	0010	0000	0010	a0	0	01	0001

v	RSLT	ID	EX
1	0011	00	0

ALU-0 RESULT BUS

v	ID	PC
0	00	0012

ALU-0 BRANCHMISS

v	RSLT	ID	EX
1	0010	01	0

ALU-1 RESULT BUS

v	ID	PC
0	01	0012

ALU-1 BRANCHMISS

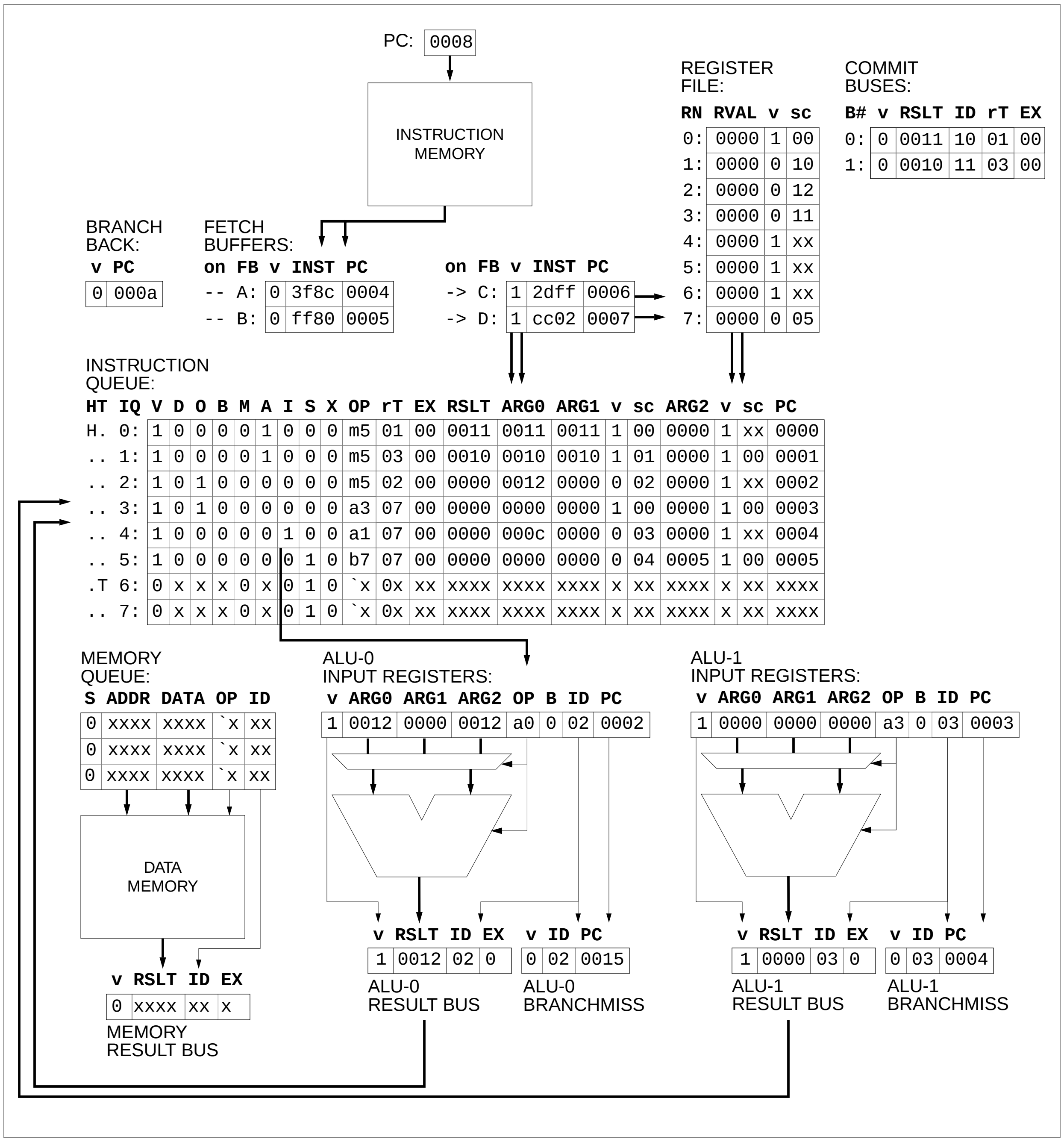
v	RSLT	ID	EX
0	xxxx	xx	x

MEMORY RESULT BUS

Figure 7: EXECUTION CYCLE 4

LW instruction itself. This allows the Tomasulo-style logic to be used to forward the results of an address-generation back into the memory instruction's IQ entry.

During cycle 4, the results of the two address-generate operations are placed on the two ALU-result busses and feed their results to IQ slots 0 and 1. The second pair of instructions issue to ALUs: the LW in slot 2 sends an address-generate operation and the LUI in slot 3 sends a LUI. The third pair of instructions (ADDI+JALR) is enqueued in slots 4 and 5 and sets register-source values appropriately: because both target r7, the SC field for r7 becomes the ID of the latter of the two instructions—that of the JALR, which is enqueued into slot 5.



PC: 0008

REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	12
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0011	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
0	000a

FETCH BUFFERS:

on	FB	v	INST	PC
--	A:	0	3f8c	0004
--	B:	0	ff80	0005
->	C:	1	2dff	0006
->	D:	1	cc02	0007

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
H.	0:	1	0	0	0	0	1	0	0	0	m5	01	00	0011	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	0	0	0	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	1	0	0	0	0	0	0	m5	02	00	0000	0012	0000	0	02	0000	1	xx	0002
..	3:	1	0	1	0	0	0	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	0	0	0	0	0	1	0	0	a1	07	00	0000	000c	0000	0	03	0000	1	xx	0004
..	5:	1	0	0	0	0	0	0	1	0	b7	07	00	0000	0000	0000	0	04	0005	1	00	0005
.T	6:	0	x	x	x	0	x	0	1	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx
..	7:	0	x	x	x	0	x	0	1	0	`x	0x	xx	xxxx	xxxx	xxxx	x	xx	xxxx	x	xx	xxxx

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
1	0012	0000	0012	a0	0	02	0002

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
1	0000	0000	0000	a3	0	03	0003

v	RSLT	ID	EX
0	xxxx	xx	x

MEMORY RESULT BUS

v	RSLT	ID	EX
1	0012	02	0

ALU-0 RESULT BUS

v	ID	PC
0	02	0015

ALU-0 BRANCHMISS

v	RSLT	ID	EX
1	0000	03	0

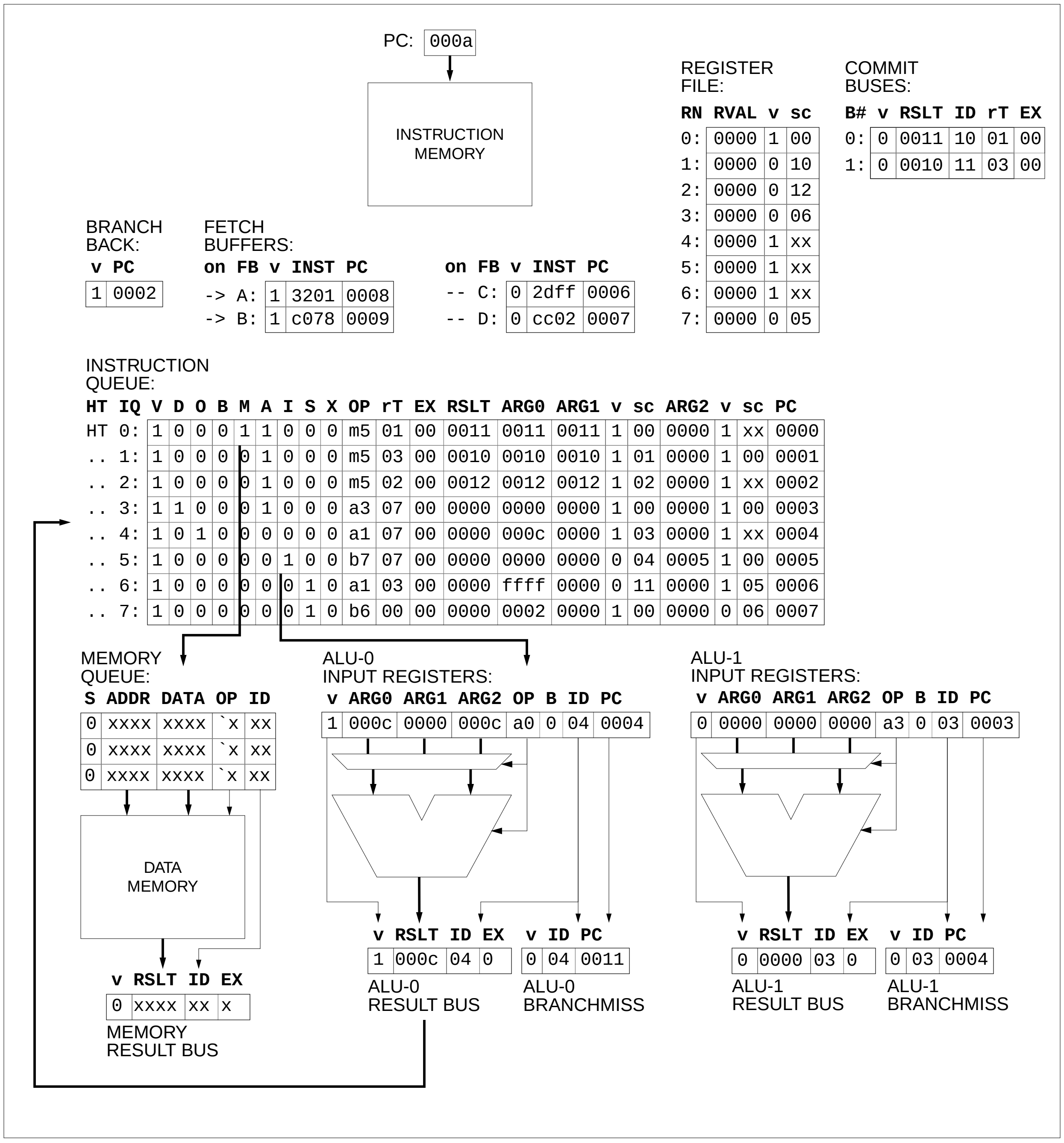
ALU-1 RESULT BUS

v	ID	PC
0	03	0004

ALU-1 BRANCHMISS

Figure 8: EXECUTION CYCLE 5

During cycle 5, the target addresses for the first two LW instructions are compared, and the “M” bits for each are set appropriately, indicating whether the memory operation can be issued to the memory queue. The LW/LUI pair is executed and the results placed on the ALU busses (LUI will be marked “done” in its IQ entry). The ADDI instruction in IQ slot 4 can issue, even though its register operand is tagged invalid in the IQ slot, because its source ID matches that on ALU-1 result bus. There are two IQ slots open; two instructions are enqueued. Two more are fetched



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	12
3:	0000	0	06
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0011	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
1	0002

FETCH BUFFERS:

on	FB	v	INST	PC
->	A:	1	3201	0008
->	B:	1	c078	0009

on	FB	v	INST	PC
--	C:	0	2dff	0006
--	D:	0	cc02	0007

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
HT	0:	1	0	0	0	1	1	0	0	0	m5	01	00	0011	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	0	0	0	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	0	0	0	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	0	1	0	0	0	0	0	0	a1	07	00	0000	000c	0000	1	03	0000	1	xx	0004
..	5:	1	0	0	0	0	0	1	0	0	b7	07	00	0000	0000	0000	0	04	0005	1	00	0005
..	6:	1	0	0	0	0	0	0	1	0	a1	03	00	0000	ffff	0000	0	11	0000	1	05	0006
..	7:	1	0	0	0	0	0	0	1	0	b6	00	00	0000	0002	0000	1	00	0000	0	06	0007

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
1	000c	0000	000c	a0	0	04	0004

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

v	RSLT	ID	EX
0	xxxx	xx	x

MEMORY RESULT BUS

v	RSLT	ID	EX
1	000c	04	0

ALU-0 RESULT BUS

v	ID	PC
0	04	0011

ALU-0 BRANCHMISS

v	RSLT	ID	EX
0	0000	03	0

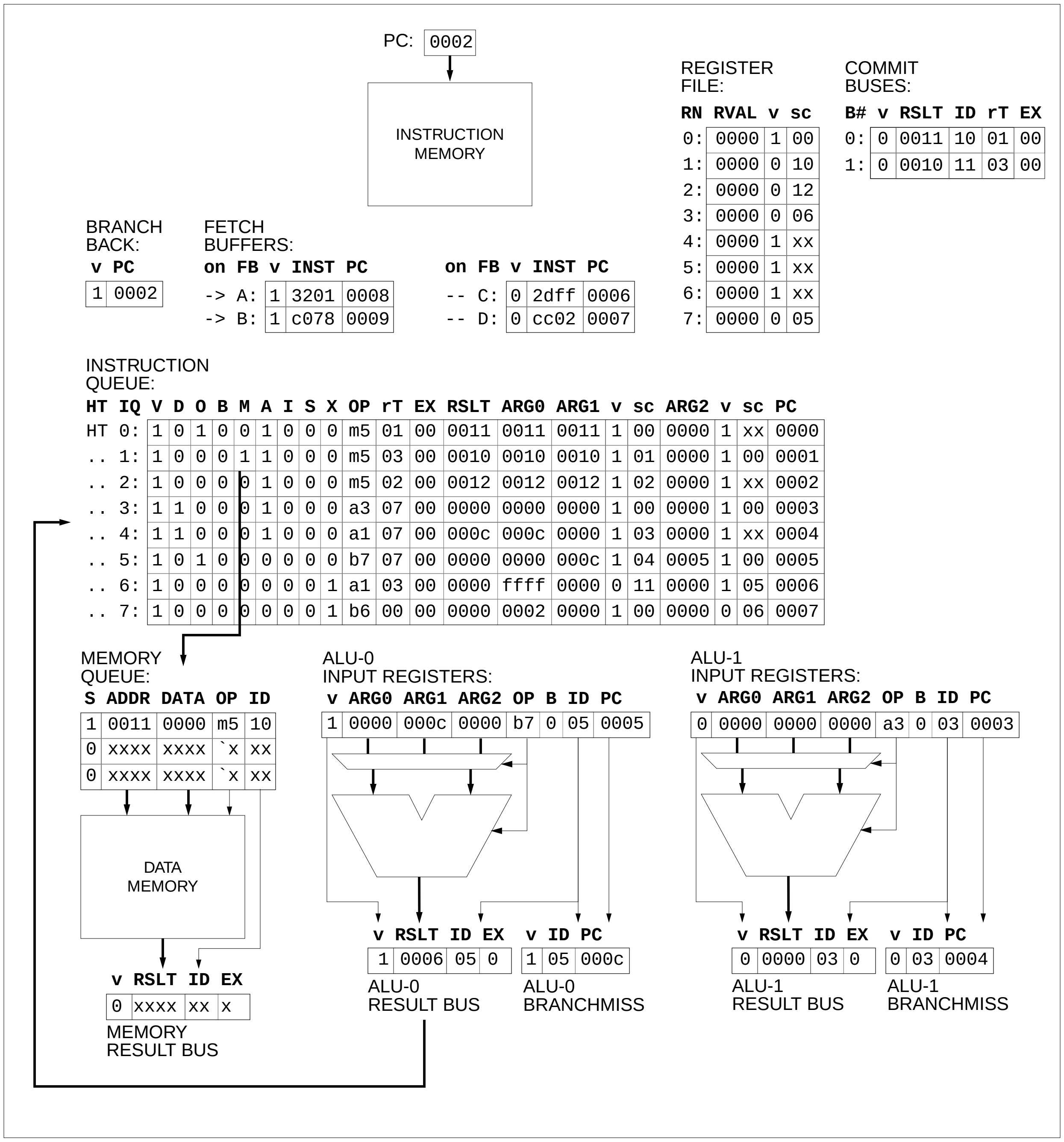
ALU-1 RESULT BUS

v	ID	PC
0	03	0004

ALU-1 BRANCHMISS

Figure 9: EXECUTION CYCLE 6

During cycle 6, the LW instruction in iq0 is issued to the memory queue (its M tag is 1). On the following cycle, we will see this reflected in the memq's entries. the JALR in iq5 is issued to an ALU because its register operand is available on ALU-0 result bus. The instruction queue is full (no entries marked "invalid"), and therefore the enqueue mechanism is stalled. Normally, this would not stall the fetch mechanism (later cycles will illustrate this) ... normally, another two instructions would be fetched into the alternate fetch buffers. However, there is a predicted-taken branch in fetchbuf B (the backwards branch *beq r0,r0, loop*), so the program counter is redirected during this cycle. Fetch will commence down the predicted path on the following cycle.



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	12
3:	0000	0	06
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0011	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
1	0002

FETCH BUFFERS:

on	FB	v	INST	PC
->	A:	1	3201	0008
->	B:	1	c078	0009
--	C:	0	2dff	0006
--	D:	0	cc02	0007

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
HT	0:	1	0	1	0	0	1	0	0	0	m5	01	00	0011	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	0	0	1	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	0	0	0	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	0	1	0	0	0	0	0	0	b7	07	00	0000	0000	000c	1	04	0005	1	00	0005
..	6:	1	0	0	0	0	0	0	0	1	a1	03	00	0000	ffff	0000	0	11	0000	1	05	0006
..	7:	1	0	0	0	0	0	0	0	1	b6	00	00	0000	0002	0000	1	00	0000	0	06	0007

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
1	0011	0000	m5	10
0	xxxx	xxxx	`x	xx
0	xxxx	xxxx	`x	xx

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
1	0000	000c	0000	b7	0	05	0005

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

MEMORY RESULT BUS

v	RSLT	ID	EX
0	xxxx	xx	x

ALU-0 RESULT BUS

v	RSLT	ID	EX
1	0006	05	0

ALU-0 BRANCHMISS

v	ID	PC
1	05	000c

ALU-1 RESULT BUS

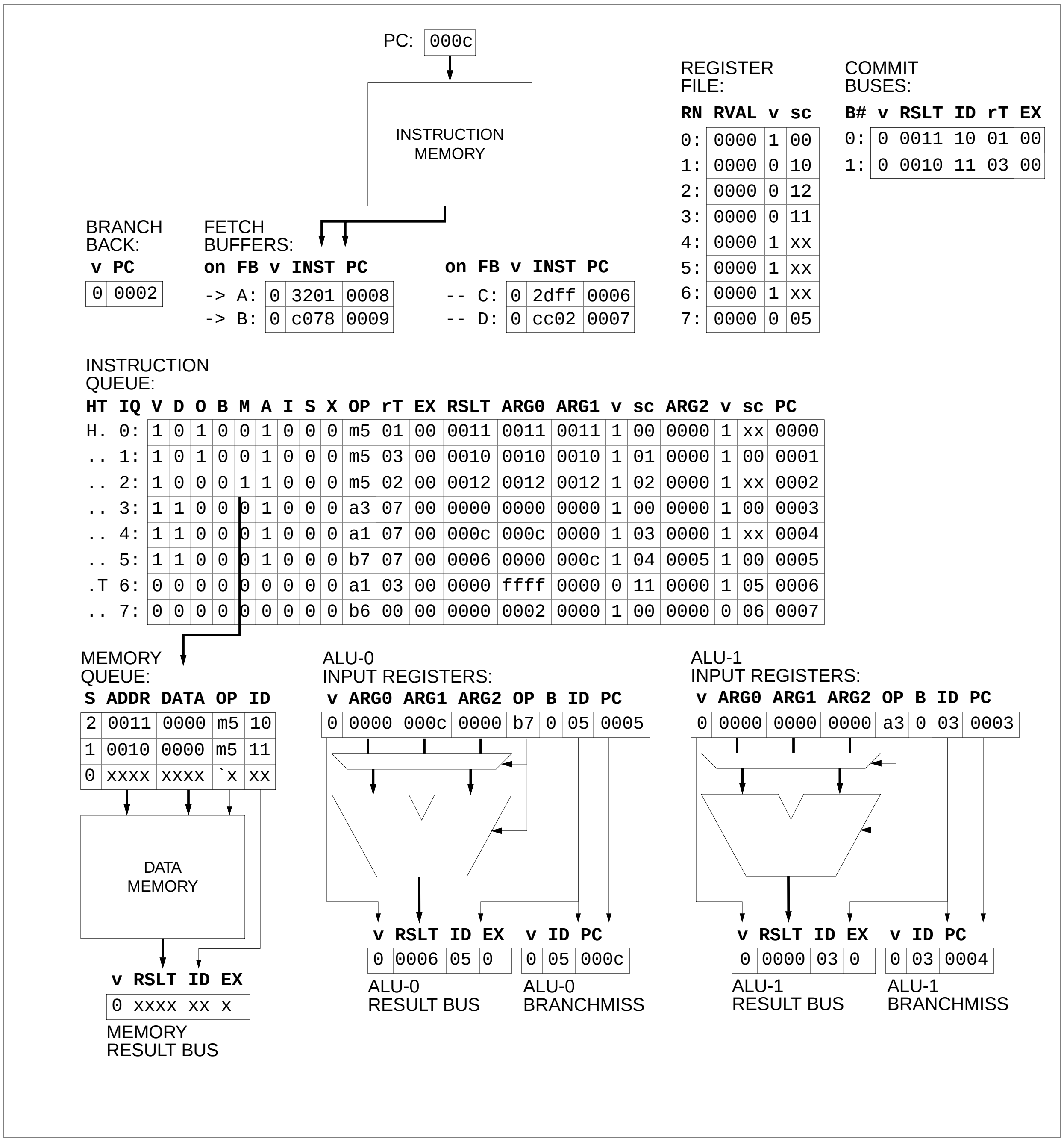
v	RSLT	ID	EX
0	0000	03	0

ALU-1 BRANCHMISS

v	ID	PC
0	03	0004

Figure 10: EXECUTION CYCLE 7

During cycle 7, the second LW instruction is issued to the memory queue (note that both were "ready" on the previous cycle, but we can only issue one per cycle). The "status" of the previous memory operation is "1" which indicates that it is in mid-request (once the status is "3" the operation is complete). The JALR issued on the previous cycle is on the ALU-0 result bus, and it has set the BRANCHMISS valid-bit high, indicating a change in control flow. This stalls both fetch and enqueue and invalidates the fetchbuf entries. The program counter will be redirected, using the valid produced by the JALR instruction. Instructions to be stomped on are tagged "1" in the "X" column: iq6 and iq7—those following the JALR. Those not stomped can still issue (iq1).



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	12
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0011	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
0	0002

FETCH BUFFERS:

on	FB	v	INST	PC
->	A:	0	3201	0008
->	B:	0	c078	0009
--	C:	0	2dff	0006
--	D:	0	cc02	0007

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
H.	0:	1	0	1	0	0	1	0	0	0	m5	01	00	0011	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	1	0	0	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	0	0	1	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	1	0	0	0	1	0	0	0	b7	07	00	0006	0000	000c	1	04	0005	1	00	0005
.T	6:	0	0	0	0	0	0	0	0	0	a1	03	00	0000	ffff	0000	0	11	0000	1	05	0006
..	7:	0	0	0	0	0	0	0	0	0	b6	00	00	0000	0002	0000	1	00	0000	0	06	0007

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
2	0011	0000	m5	10
1	0010	0000	m5	11
0	xxxx	xxxx	`x	xx

MEMORY RESULT BUS:

v	RSLT	ID	EX
0	xxxx	xx	x

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	000c	0000	b7	0	05	0005

ALU-0 RESULT BUS:

v	RSLT	ID	EX
0	0006	05	0

ALU-0 BRANCHMISS:

v	ID	PC
0	05	000c

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

ALU-1 RESULT BUS:

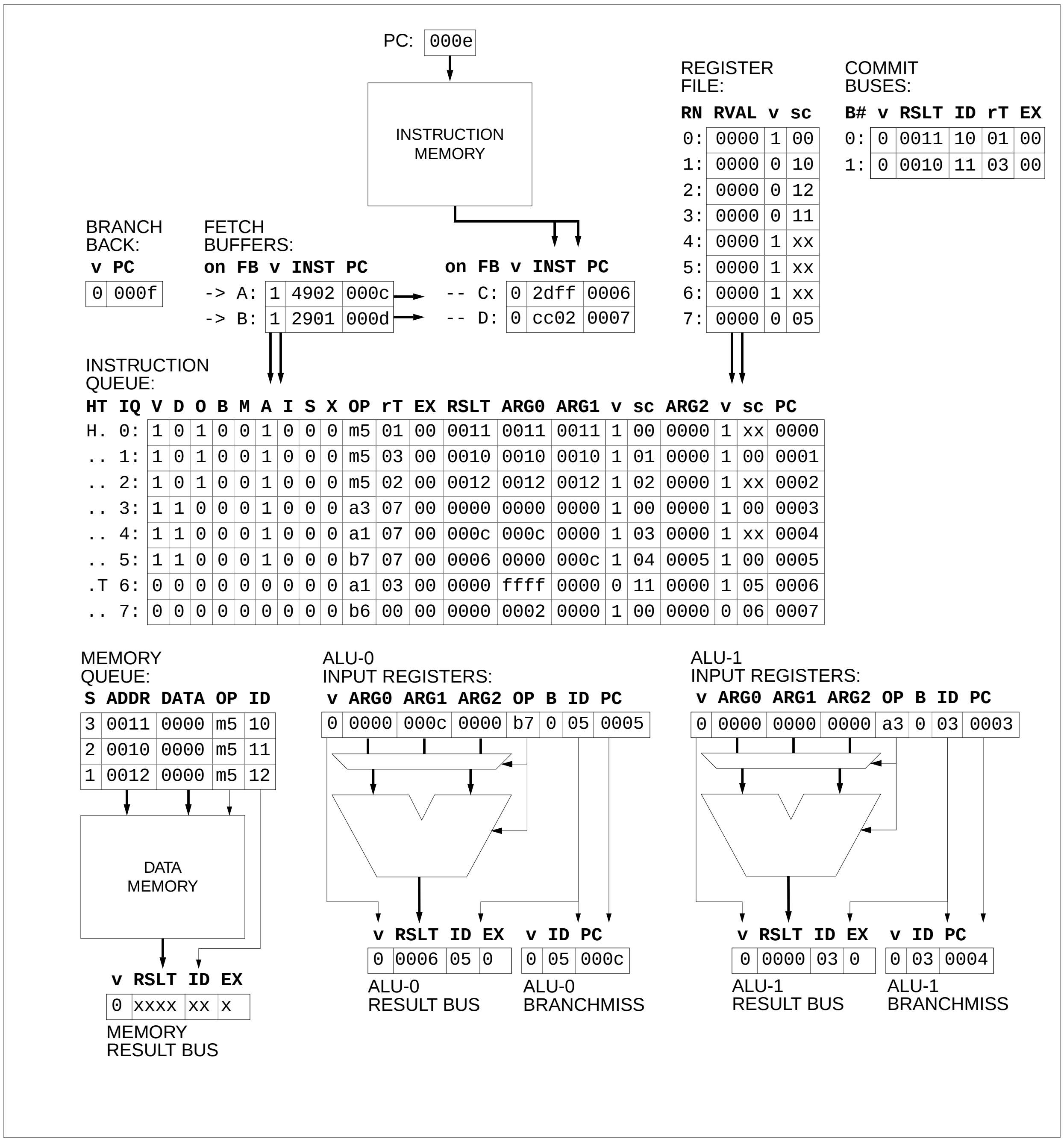
v	RSLT	ID	EX
0	0000	03	0

ALU-1 BRANCHMISS:

v	ID	PC
0	03	0004

Figure 11: EXECUTION CYCLE 8

At the beginning of cycle 8, we see that the stomped instructions are now marked “invalid” in the instruction queue, and the tail pointer has been reset appropriately. The fetch buffers have been marked “invalid.” Several instructions (iq3, iq4, and iq5) are “done” and thus ready to commit, but are held up by the three LW instructions at the head of the queue. The program counter has been rest appropriately (it has the value of the branchmiss status from the previous cycle). Most importantly, the contents of the register file reflect the correct machine state: on the previous cycle the **addi** instruction in iq6 targeted r3, which had “06” as its source. Now, register r3 has the previous source of r3 listed: the LW in iq1. During this cycle, another LW is issued to the memory



PC: 000e

REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	12
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0011	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
0	000f

FETCH BUFFERS:

on	FB	v	INST	PC
->	A:	1	4902	000c
->	B:	1	2901	000d
--	C:	0	2dff	0006
--	D:	0	cc02	0007

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
H.	0:	1	0	1	0	0	1	0	0	0	m5	01	00	0011	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	1	0	0	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	1	0	0	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	1	0	0	0	1	0	0	0	b7	07	00	0006	0000	000c	1	04	0005	1	00	0005
.T	6:	0	0	0	0	0	0	0	0	0	a1	03	00	0000	ffff	0000	0	11	0000	1	05	0006
..	7:	0	0	0	0	0	0	0	0	0	b6	00	00	0000	0002	0000	1	00	0000	0	06	0007

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
3	0011	0000	m5	10
2	0010	0000	m5	11
1	0012	0000	m5	12

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	000c	0000	b7	0	05	0005

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

v	RSLT	ID	EX
0	0006	05	0

ALU-0 RESULT BUS

v	ID	PC
0	05	000c

ALU-0 BRANCHMISS

v	RSLT	ID	EX
0	0000	03	0

ALU-1 RESULT BUS

v	ID	PC
0	03	0004

ALU-1 BRANCHMISS

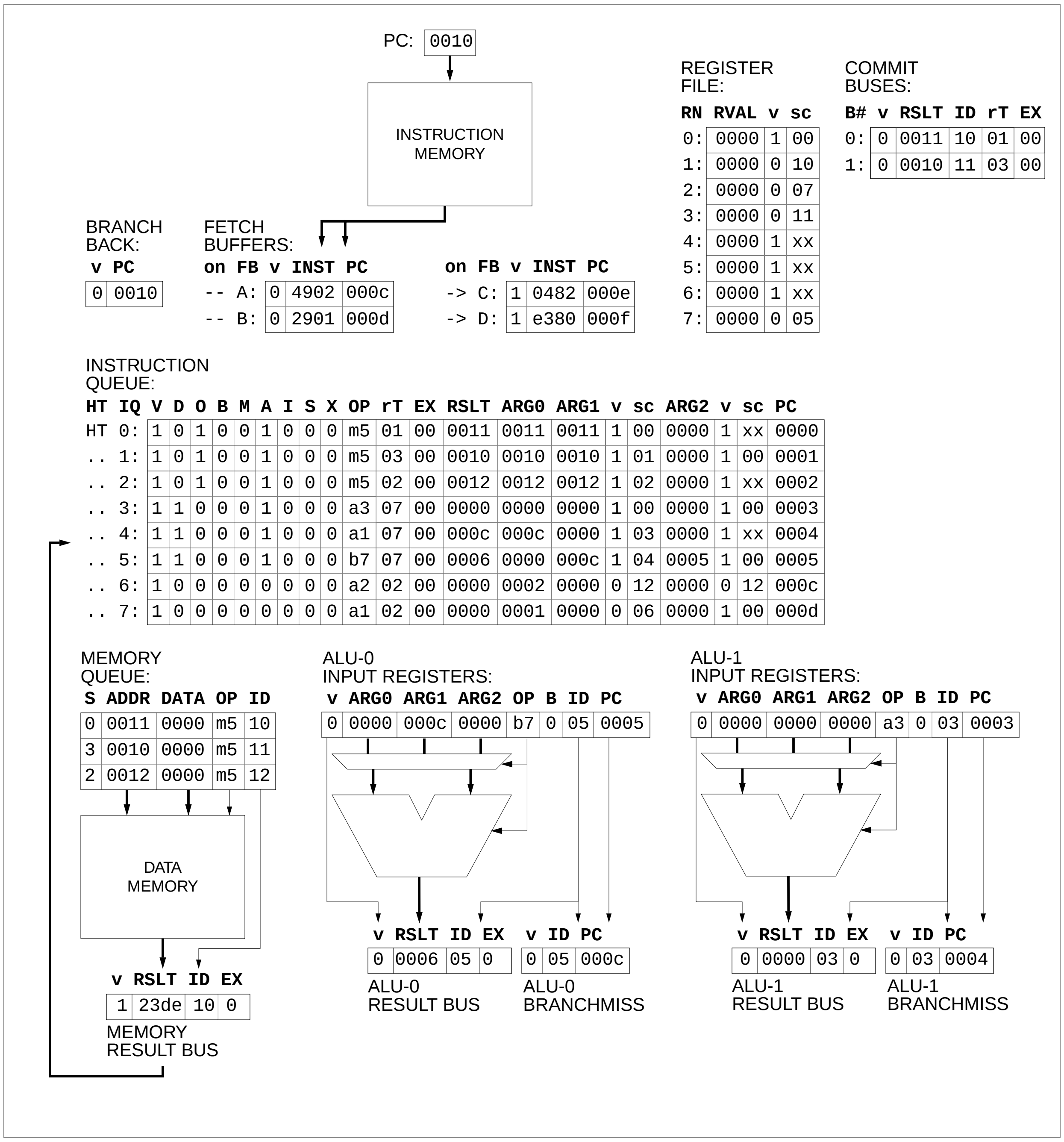
v	RSLT	ID	EX
0	xxxx	xx	x

MEMORY RESULT BUS

Figure 12: EXECUTION CYCLE 9

queue. Two instructions are fetched. Very little else happens because most of the enqueued instructions are done.

During cycle 9, the first of the three LW instructions becomes ready in the memory queue; its result will be sent on the memory result bus on the following cycle. The two instructions fetched on the previous cycle (the NAND and ADDI at the top of the **sub** subroutine) are enqueued and the next two subroutine instructions (ADD and JALR) are fetched. The states of the memory queue entries are incremented by one.



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	07
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	0	0011	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
0	0010

FETCH BUFFERS:

on	FB	v	INST	PC
--	A:	0	4902	000c
--	B:	0	2901	000d

on	FB	v	INST	PC
->	C:	1	0482	000e
->	D:	1	e380	000f

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
HT	0:	1	0	1	0	0	1	0	0	0	m5	01	00	0011	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	1	0	0	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	1	0	0	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	1	0	0	0	1	0	0	0	b7	07	00	0006	0000	000c	1	04	0005	1	00	0005
..	6:	1	0	0	0	0	0	0	0	0	a2	02	00	0000	0002	0000	0	12	0000	0	12	000c
..	7:	1	0	0	0	0	0	0	0	0	a1	02	00	0000	0001	0000	0	06	0000	1	00	000d

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	0011	0000	m5	10
3	0010	0000	m5	11
2	0012	0000	m5	12

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	000c	0000	b7	0	05	0005

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

v	RSLT	ID	EX
1	23de	10	0

MEMORY RESULT BUS

v	RSLT	ID	EX
0	0006	05	0

ALU-0 RESULT BUS

v	ID	PC
0	05	000c

ALU-0 BRANCHMISS

v	RSLT	ID	EX
0	0000	03	0

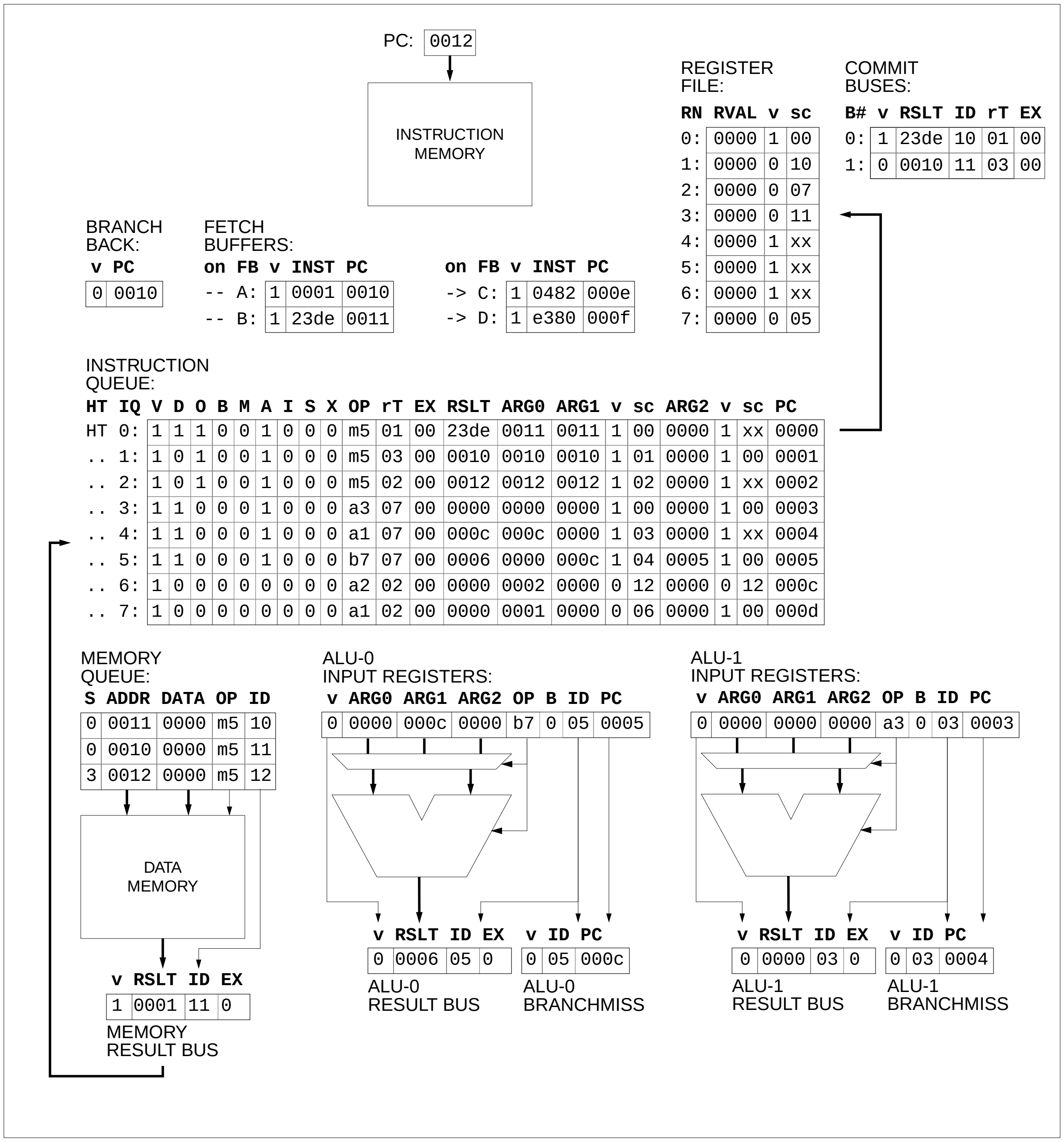
ALU-1 RESULT BUS

v	ID	PC
0	03	0004

ALU-1 BRANCHMISS

Figure 13: EXECUTION CYCLE 10

During cycle 10, the result of the first LW instruction is seen on the memory result bus. THe result is latched at the end of the cycle, when the instruction will be tagged “done”. No instructions are issued to functional units because the two potential instructions are dependent on the LW instruction in iq2 (its result will become available in two cycles). Because the IQ is full, enqueue is stalled. Because there are no predicted-taken branches (i.e. backwards branches) in the fetch buffers, instruction fetch is not stalled; the two instructions following the subroutine are fetched (they are actually data, but they will be discarded when the JALR at the end of the subroutine takes effect).



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	0000	0	10
2:	0000	0	07
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	1	23de	10	01	00
1:	0	0010	11	03	00

BRANCH BACK:

v	PC
0	0010

FETCH BUFFERS:

on	FB	v	INST	PC
--	A:	1	0001	0010
--	B:	1	23de	0011

on	FB	v	INST	PC
->	C:	1	0482	000e
->	D:	1	e380	000f

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
HT	0:	1	1	1	0	0	1	0	0	0	m5	01	00	23de	0011	0011	1	00	0000	1	xx	0000
..	1:	1	0	1	0	0	1	0	0	0	m5	03	00	0010	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	1	0	0	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	1	0	0	0	1	0	0	0	b7	07	00	0006	0000	000c	1	04	0005	1	00	0005
..	6:	1	0	0	0	0	0	0	0	0	a2	02	00	0000	0002	0000	0	12	0000	0	12	000c
..	7:	1	0	0	0	0	0	0	0	0	a1	02	00	0000	0001	0000	0	06	0000	1	00	000d

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	0011	0000	m5	10
0	0010	0000	m5	11
3	0012	0000	m5	12

MEMORY RESULT BUS

v	RSLT	ID	EX
1	0001	11	0

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	000c	0000	b7	0	05	0005

ALU-0 RESULT BUS

v	RSLT	ID	EX
0	0006	05	0

ALU-0 BRANCHMISS

v	ID	PC
0	05	000c

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

ALU-1 RESULT BUS

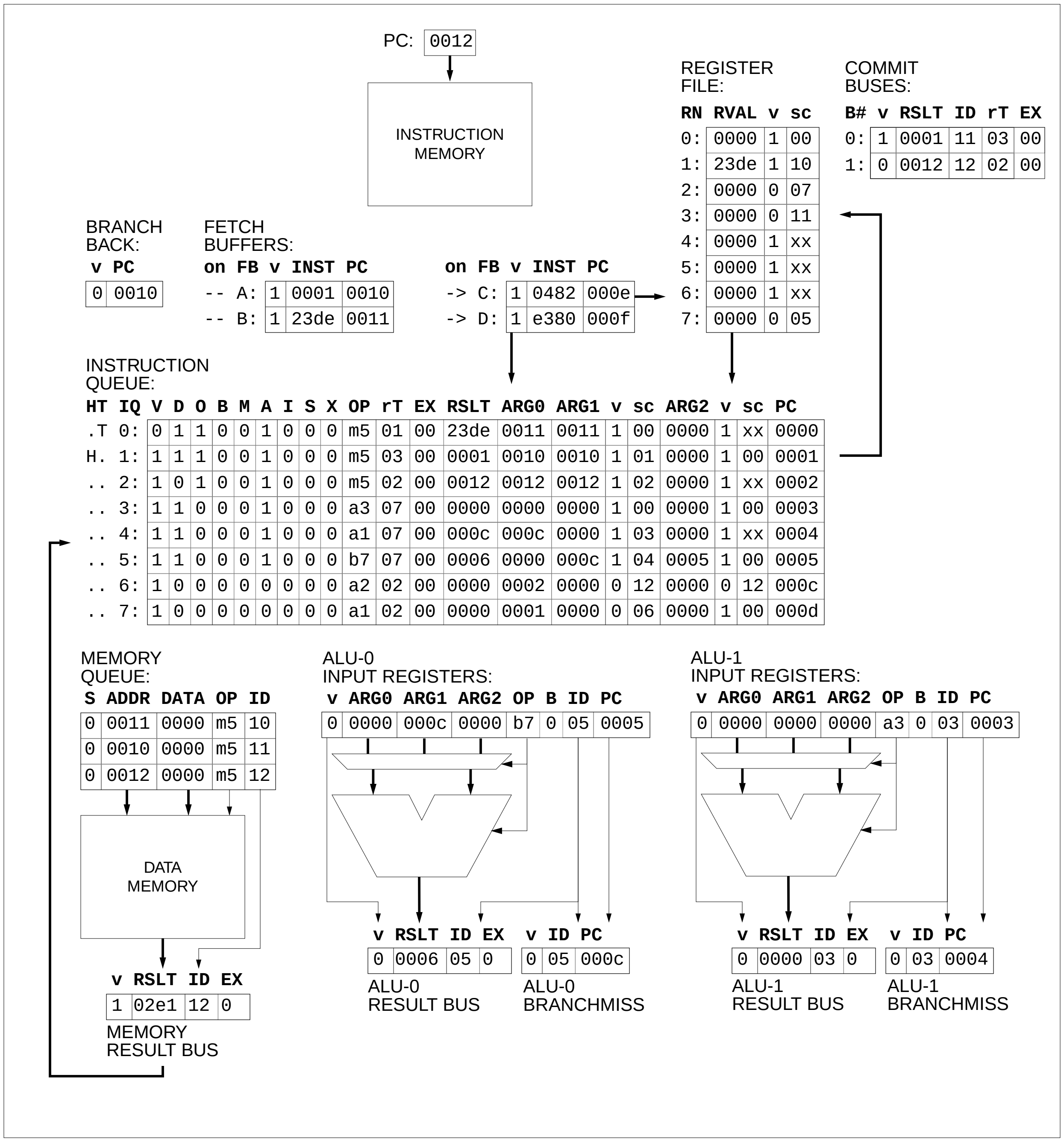
v	RSLT	ID	EX
0	0000	03	0

ALU-1 BRANCHMISS

v	ID	PC
0	03	0004

Figure 14: EXECUTION CYCLE 11

During cycle 11, the first LW instruction commits its result to the register file. On the following cycle, its IQ slot will be tagged as invalid, marking it available for a new instruction. The result for the second LW instruction is seen on the memory result bus. Instructions in iq6 and iq7 are stalled waiting on the third LW instruction. Enqueue is stalled waiting for an available IQ entry. Fetch is stalled waiting for an available fetch buffer.



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	23de	1	10
2:	0000	0	07
3:	0000	0	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	1	0001	11	03	00
1:	0	0012	12	02	00

BRANCH BACK:

v	PC
0	0010

FETCH BUFFERS:

on	FB	v	INST	PC
--	A:	1	0001	0010
--	B:	1	23de	0011
->	C:	1	0482	000e
->	D:	1	e380	000f

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
.T	0:	0	1	1	0	0	1	0	0	0	m5	01	00	23de	0011	0011	1	00	0000	1	xx	0000
H.	1:	1	1	1	0	0	1	0	0	0	m5	03	00	0001	0010	0010	1	01	0000	1	00	0001
..	2:	1	0	1	0	0	1	0	0	0	m5	02	00	0012	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	1	0	0	0	1	0	0	0	b7	07	00	0006	0000	000c	1	04	0005	1	00	0005
..	6:	1	0	0	0	0	0	0	0	0	a2	02	00	0000	0002	0000	0	12	0000	0	12	000c
..	7:	1	0	0	0	0	0	0	0	0	a1	02	00	0000	0001	0000	0	06	0000	1	00	000d

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	0011	0000	m5	10
0	0010	0000	m5	11
0	0012	0000	m5	12

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	000c	0000	b7	0	05	0005

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

ALU-0 RESULT BUS:

v	RSLT	ID	EX
0	0006	05	0

ALU-0 BRANCHMISS:

v	ID	PC
0	05	000c

ALU-1 RESULT BUS:

v	RSLT	ID	EX
0	0000	03	0

ALU-1 BRANCHMISS:

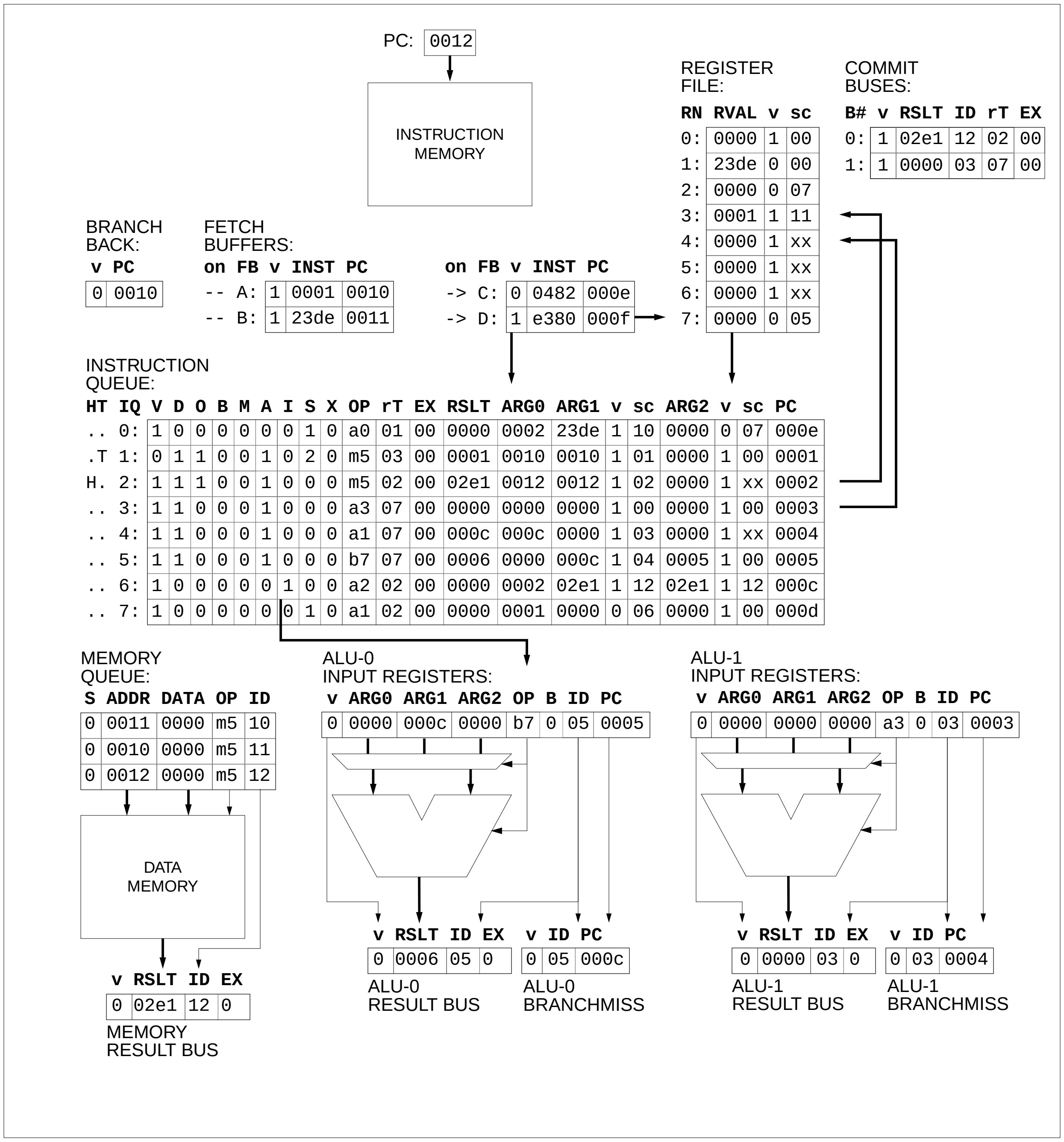
v	ID	PC
0	03	0004

MEMORY RESULT BUS:

v	RSLT	ID	EX
1	02e1	12	0

Figure 15: EXECUTION CYCLE 12

During cycle 12, the second LW instruction commits. The slot opened up by the LW instruction (slot iq0) is the enqueue-target for one of the instructions in the fetchbufs (the ADD instruction in fetchbuf C). The result for the third LW instruction is seen on the memory result bus. This will enable the waiting instructions to issue to functional units on the following cycle. Fetch is still stalled because there are no empty fetch buffers.



REGISTER FILE:

RN	RVAL	v	sc
0:	0000	1	00
1:	23de	0	00
2:	0000	0	07
3:	0001	1	11
4:	0000	1	xx
5:	0000	1	xx
6:	0000	1	xx
7:	0000	0	05

COMMIT BUSES:

B#	v	RSLT	ID	rT	EX
0:	1	02e1	12	02	00
1:	1	0000	03	07	00

BRANCH BACK:

v	PC
0	0010

FETCH BUFFERS:

on	FB	v	INST	PC
--	A:	1	0001	0010
--	B:	1	23de	0011
->	C:	0	0482	000e
->	D:	1	e380	000f

INSTRUCTION QUEUE:

HT	IQ	V	D	O	B	M	A	I	S	X	OP	rT	EX	RSLT	ARG0	ARG1	v	sc	ARG2	v	sc	PC
..	0:	1	0	0	0	0	0	0	1	0	a0	01	00	0000	0002	23de	1	10	0000	0	07	000e
.T	1:	0	1	1	0	0	1	0	2	0	m5	03	00	0001	0010	0010	1	01	0000	1	00	0001
H.	2:	1	1	1	0	0	1	0	0	0	m5	02	00	02e1	0012	0012	1	02	0000	1	xx	0002
..	3:	1	1	0	0	0	1	0	0	0	a3	07	00	0000	0000	0000	1	00	0000	1	00	0003
..	4:	1	1	0	0	0	1	0	0	0	a1	07	00	000c	000c	0000	1	03	0000	1	xx	0004
..	5:	1	1	0	0	0	1	0	0	0	b7	07	00	0006	0000	000c	1	04	0005	1	00	0005
..	6:	1	0	0	0	0	0	1	0	0	a2	02	00	0000	0002	02e1	1	12	02e1	1	12	000c
..	7:	1	0	0	0	0	0	0	1	0	a1	02	00	0000	0001	0000	0	06	0000	1	00	000d

MEMORY QUEUE:

S	ADDR	DATA	OP	ID
0	0011	0000	m5	10
0	0010	0000	m5	11
0	0012	0000	m5	12

ALU-0 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	000c	0000	b7	0	05	0005

ALU-1 INPUT REGISTERS:

v	ARG0	ARG1	ARG2	OP	B	ID	PC
0	0000	0000	0000	a3	0	03	0003

ALU-0 RESULT BUS:

v	RSLT	ID	EX
0	0006	05	0

ALU-0 BRANCHMISS:

v	ID	PC
0	05	000c

ALU-1 RESULT BUS:

v	RSLT	ID	EX
0	0000	03	0

ALU-1 BRANCHMISS:

v	ID	PC
0	03	0004

MEMORY RESULT BUS:

v	RSLT	ID	EX
0	02e1	12	0

Figure 16: EXECUTION CYCLE 13

During cycle 13, two instructions commit, opening up two more slots in the instruction queue. The NAND instruction in iq6 that was waiting on the LW in iq2 issues to a functional unit. The IQ slot opened up by the LW instruction in iq1, which committed on the previous cycle, is filled by the instruction in fetchbuf D: the JALR instruction that marks the end of the subroutine.