SECTION 7
INSTRUCTION TIMING

This section describes instruction flow and the basic instruction pipeline in the RCPU, provides details of execution timing for each execution unit, defines the concepts of serialization and synchronization, provides timing information for each RCPU instruction, and provides timing examples for different types of instructions.

7.1 Instruction Flow

The instruction sequencer provides centralized control over data flow between execution units and register files. The sequencer implements the basic instruction pipeline, fetches instructions from the memory system, issues them to available execution units, and maintains a state history so it can back the machine up in the event of an exception.

The instruction sequencer fetches the instructions from the instruction cache into the instruction pre-fetch queue. The processor uses branch folding (a technique of removing the branch instructions from the pre-fetch queue) in order to execute branches in parallel with execution of sequential instructions. Sequential (non-branch) instructions reaching the top of the instruction queue are issued to the execution units. Instructions may be flushed from the instruction queue when an external interrupt is detected, a previous instruction causes an exception, or a branch prediction turns out to be incorrect.

All instructions, including branches, enter the history buffer along with processor state information that may be affected by the instruction's execution. This information is used to enable out of order completion of instructions together with precise exceptions handling. Instructions may be flushed from the machine when an exception is taken. The instruction queue is always flushed when recovery of the history buffer takes place. Refer to **6.3 Precise Exception Model Implementation** for additional information.

An instruction retires from the machine after it finishes execution without exception and all preceding instructions have already retired from the machine.

Figure 7-1 illustrates the instruction flow in the RCPU.

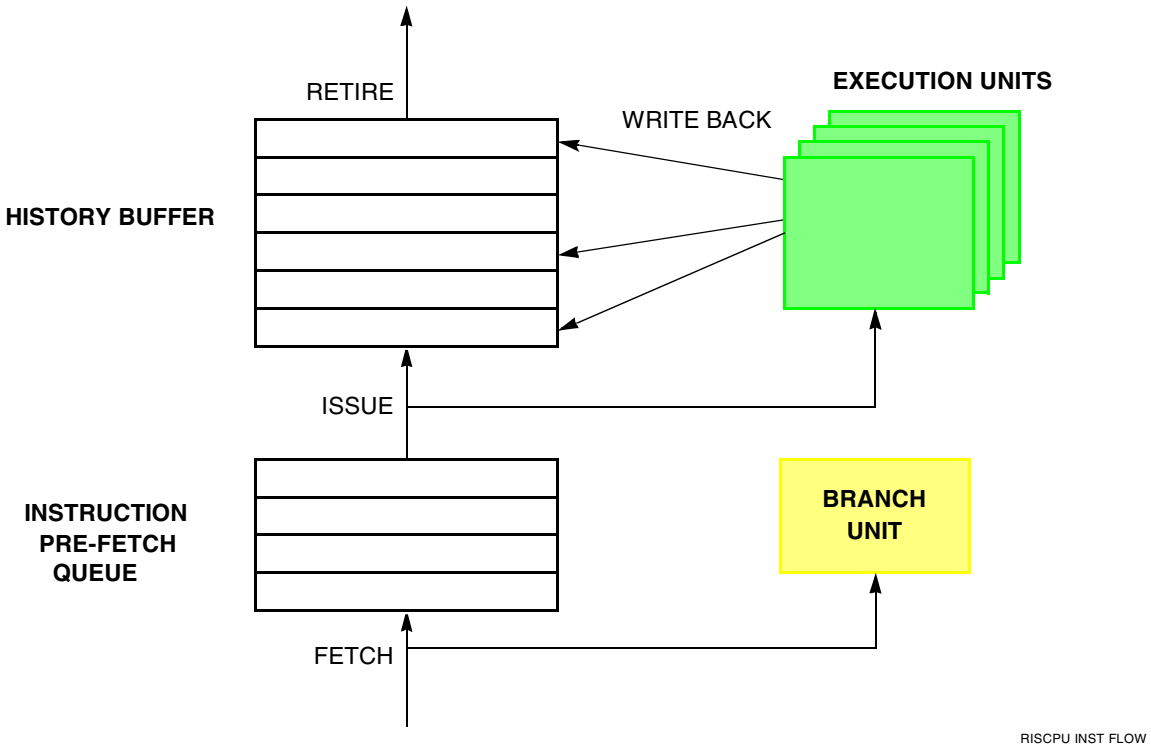

Figure 7-1 Instruction Flow

7.1.1 Instruction Sequencer Data Path

Figure 7-2 illustrates the instruction sequencer data path.

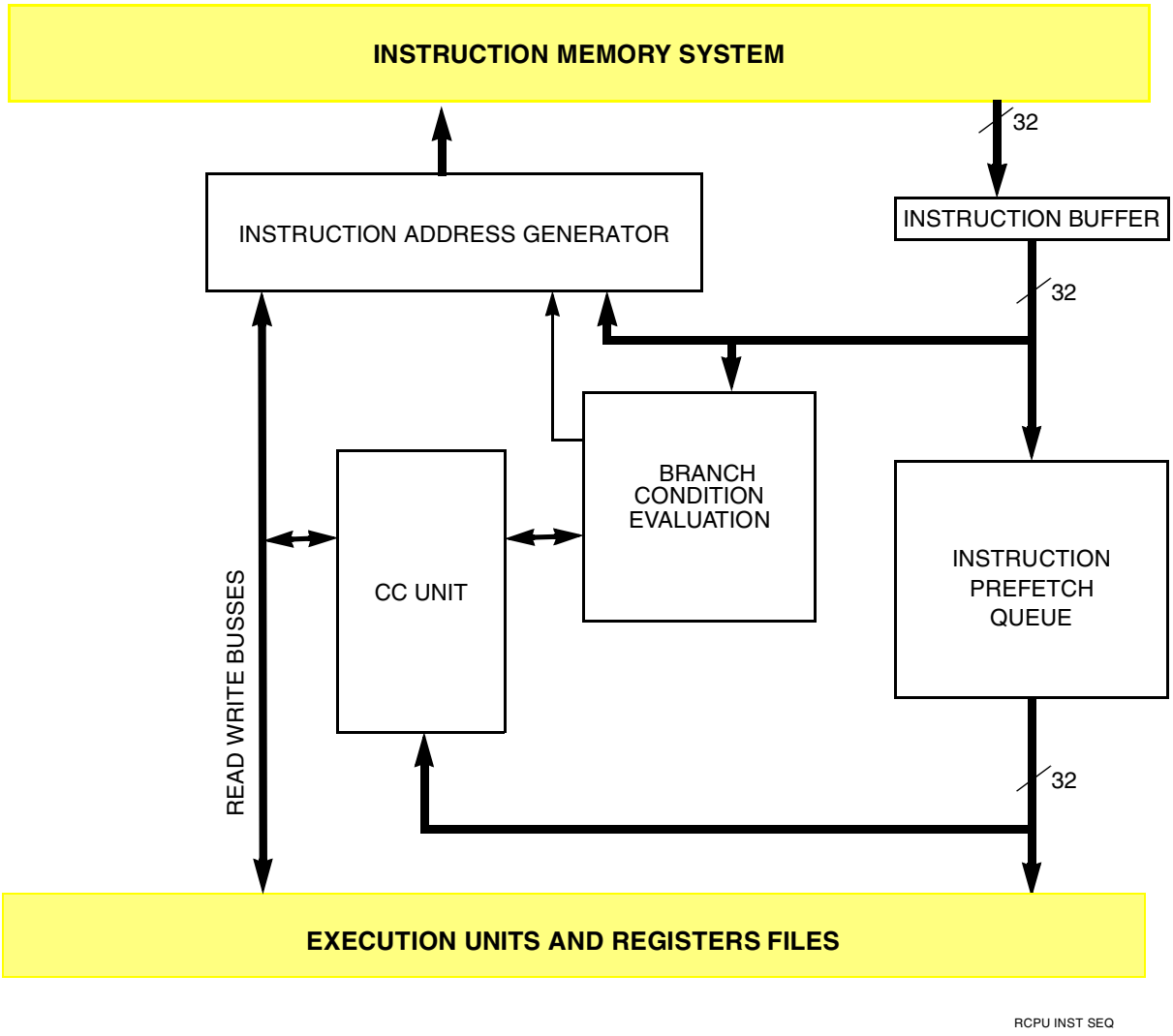

Figure 7-2 Instruction Sequencer Data Path

7.1.2 Instruction Issue

The sequencer attempts to issue a sequential (non-branch) instruction on each clock, if possible. In order for an instruction to be issued, the execution unit must be available and it must determine that the required source data is available and that no other instruction still in execution targets the same destination register. The sequencer broadcasts the presence of the instruction on the instruction bus, and each execution unit decodes the instruction. The execution unit responsible for executing the instruction determines whether the operands and target registers are free and informs the sequencer that it accepts the instruction for execution.

7.1.3 Basic Instruction Pipeline

The RCPU instruction pipeline has four stages:

1. The dispatch stage is implemented using a distributed mechanism. The central dispatch unit broadcasts the instruction to all units. In addition, scoreboard information (regarding data dependencies) is broadcast to each execution unit. Each execution unit decodes the instruction. If the instruction is not implemented, a program exception is taken. If the instruction is legal and no data dependency is found, the instruction is accepted by the appropriate execution unit, and the data found in the destination register is copied to the history buffer. If a data dependency exists, the machine is stalled until the dependency is resolved.
2. In the execute stage, each execution unit that has an executable instruction executes the instruction (perhaps over multiple cycles).
3. In the writeback stage, the execution unit writes the result to the destination register and reports to the history buffer that the instruction is completed.
4. In the retirement stage, the history buffer retires instructions in architectural order. An instruction retires from the machine if it completes execution with no exceptions and if all instructions preceding it in the instruction stream have finished execution with no exceptions. As many as six instructions can be retired in one clock.

The history buffer maintains the correct architectural machine state. An exception is taken only when the instruction is ready to be retired from the machine (i.e., after all previously-issued instructions have already been retired from the machine). When an exception is taken, all instructions following the excepting instruction are canceled, (i.e., the values of the affected destination registers are restored using the values saved in the history buffer during the dispatch stage).

Figure 7-3 illustrates the basic instruction pipeline timing.

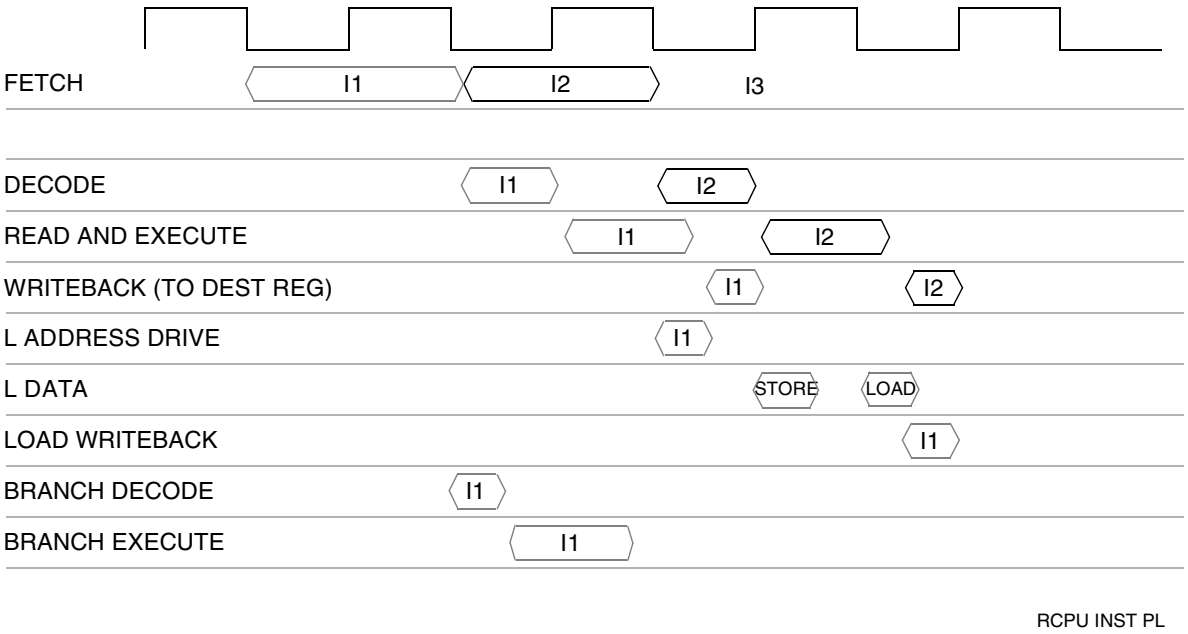

Figure 7-3 Basic Instruction Pipeline

7.2 Execution Unit Timing Details

The following sections describe instruction timing considerations within each RCPU execution unit.

7.2.1 Integer Unit (IU)

The integer unit executes all integer processor instructions, except the integer storage access instructions, which are implemented by the load/store unit. The IU consists of two execution units:

- The IMUL-IDIV executes the integer multiply and divide instructions.
- The ALU-BFU unit executes all integer logic, add, and subtract instructions, and bit-field instructions.

All instructions executed by the ALU-BFU, except for integer trap instructions, have a latency of one clock cycle. Instructions executed by the IMUL-IDIV unit have latencies of more than one clock cycle. The IMUL-IDIV unit is pipelined for multiply instructions, but not for divide instructions. Therefore, the instruction sequencer can issue one instruction to the IU each clock cycle, except when an integer divide instruction is preceded or followed by an integer divide or multiply instruction.

7.2.1.1 Update of the XER During Divide Instructions

Integer divide instructions have a relatively long latency. However, these instructions can update XER[OV], the overflow bit in the integer exception register, after one cycle. Data dependency on the XER is therefore limited to one cycle although the latency of an integer divide instruction can be up to eleven clock cycles.

7.2.2 Floating Point Unit (FPU)

The floating-point unit contains a double-precision multiply array, the floating-point status and control register (FPSCR), and the FPRs. The multiply-add array allows the processor to efficiently implement floating-point operations such as multiply, multiply-add, and divide.

The RCPU depends on a software envelope to fully implement the IEEE floating-point specification. Overflows, underflows, NaNs, and denormalized numbers cause floating-point assist exceptions that invoke a software routine to deliver (with hardware assistance) the correct IEEE result. Refer to **6.11.10 Floating-Point Assist Exception (0x00E00)** for additional information.

To accelerate time-critical operations and make them more deterministic, the RCPU provides a mode of operation that avoids invoking the software envelope and attempts to deliver results in hardware that are adequate for most applications, if not in strict conformance with IEEE standards. In this mode, denormalized numbers, NaNs, and IEEE invalid operations are treated as legitimate, returning default results rather than causing floating-point assist exceptions.

7.2.3 Load/Store Unit (LSU)

The load-store unit handles all data transfer between the integer and floating-point register files and the chip-internal load/store bus (L-bus). The load/store unit is implemented as an independent execution unit so that stalls in the memory pipeline do not cause the master instruction pipeline to stall (unless there is a data dependency). The unit is fully pipelined so that memory instructions of any size may be issued on back-to-back cycles.

There is a 32-bit wide data path between the load/store unit and the integer register file and a 64-bit wide data path between the load/store unit and the floating-point register file.

Single-word accesses to on-chip data RAM require one clock cycle, resulting in two clock cycles latency. Double-word accesses require two clock cycles, resulting in three clock cycles latency. Since the L-bus is 32 bits wide, double-word transfers require two bus accesses.

The LSU interfaces with the external bus interface for all instructions that access memory. Addresses are formed by adding the source one register operand specified by the instruction (or zero) to either a source two register operand or to a 16-bit, immediate value embedded in the instruction.

7.2.3.1 Load/Store Instruction Issue

When a load or store instruction is encountered, the LSU checks the scoreboard to determine if all the operands are available. These operands include:

- Address registers operands
- Source data register operands (for store instructions)
- Destination data register operands (for load instructions)
- Destination address register operands (for load/store with update instructions)

If all operands are available, the LSU takes the instruction and enables the sequencer to issue a new instruction. Using a dedicated interface, the LSU notifies the IU to calculate the effective address.

All load and store instructions are executed and terminated in order. If there are no prior instructions waiting in the address queue, the load or store instruction is issued to the L-bus as soon as the instruction is taken. Otherwise, if there are still prior instructions whose address are still to be issued to the L-bus, the instruction is inserted into the address queue, and data (for store instructions) is inserted into the respective store data queue. Note that for load/store with update instructions, the destination address register is written back on the following clock cycle, regardless of the state of the address queue.

A new store instruction is not issued to the L-bus until all prior instructions have terminated without an exception. This is done in order to implement the PowerPC precise exception model. In case of a load instruction followed by a store instruction, a delay of one clock cycle is inserted between the termination of the load bus cycle and the issuing of the store cycle.

7.2.3.2 Load/Store Synchronizing Instructions

For certain LSU instructions, the instruction is not taken (as defined in the glossary) until all previous instructions have terminated. These instructions are:

- Load/Store Multiple instructions — **lmw**, **stmw**
- Storage Synchronization instructions — **lwarx**, **stwcx**, **sync**
- String instructions — **lswi**, **lswx**, **stswi**, **stswx**
- Move to internal special registers and move to external-to-processor special purpose registers

Issuing of further instructions is stalled until the following load/store instructions terminate:

- Load/Store Multiple insturctions — **lmw**, **stmw**
- Storage Synchronization instructions — **lwarx**, **stwcx**, **sync**
- String instructions — **lswi**, **lswx**, **stswi**, **stswx**

7.2.3.3 Load/Store Instruction Timing Summary

Table 7-1 summarizes the timing of load/store instructions, assuming a parked bus and zero wait state memory references. The parameter “N” denotes the number of registers transferred.

Table 7-1 Load/Store Instructions Timing

Instruction Type	Latency		Cleared from Load/Store Unit	
	Internal Memory	External Memory	Internal Memory	External Memory
Fixed-Point Single Target Register Load Floating-Point Single-Precision Load	2 clocks	4 clocks	2 clocks	4 clocks
Fixed-Point Single Target Register Store Floating-Point Single-Precision Store	1 clock	1 clock	2clock	4 clock
Floating-Point Double-Precision Load[1]	3 clocks	5 clocks	3 clocks	5 clocks
Floating-Point Double-Precision Store[1]	1 clock	1 clock	3 clocks	5 clocks
Load Multiple	1 + N	3 + N + [(N + 1)/3]	1 + N	3 + N + [(N + 1)/3]

NOTES:
1. Double-precision load and store instructions are pipelined on the bus.

7.2.3.4 Bus Cycles for String Instructions

String instructions are broken into a series of aligned bus accesses. **Figure 7-4** illustrates the maximum number of bus cycles needed for string instruction execution. This is the case where the beginning and end of the string are unaligned.

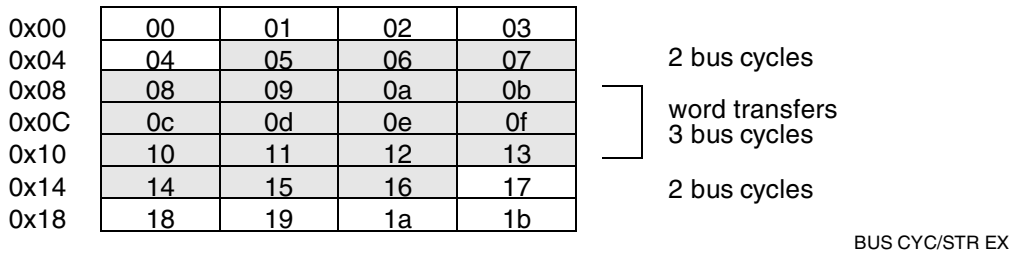

Figure 7-4 Number of Bus Cycles Needed for String Instruction Execution

7.2.3.5 Stalls During Floating-Point Store Instructions

In the following sequence there is a delay of one clock cycle before the second floating-point store instruction is executed:

1. Load instruction
2. First floating-point store instruction
3. Second floating-point store instruction

If the accesses are to zero-wait-state L-bus memory and the instructions are issued on consecutive clock cycles, the second floating-point store instruction is stalled for one clock cycle.

7.2.4 Branch Processing Unit (BPU)

The sequencer maintains a prefetch queue that can hold up to four instructions. This prefetch queue enables branches to be issued in parallel with sequential instructions. In the ideal case, a sequential instruction is issued every clock cycle, even when branches are present in the code. This feature is possible because of branch folding, the removal of branch instructions from the pre-fetch queue.

All instructions are fetched into the instruction prefetch queue, but only sequential instructions are issued to the execution units upon reaching the head of the queue. (Branches are placed into the instruction prefetch queue to enable watchpoint marking — refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.) Since branches do not prevent the issue of sequential instructions unless they come in pairs, the performance impact of entering branches in the instruction prefetch queue is negligible.

In addition to branch folding, the RCPU implements a branch reservation station and static branch prediction to allow branches to issue as early as possible. The reservation station allows a branch instruction to be issued even before its condition is ready. With the branch issued and out of the way, instruction pre-fetch can continue while the branch operand is being computed and the condition is being evaluated. Static branch prediction is used to determine which instruction stream is pre-fetched while the branch is being resolved. When the branch operand becomes available, it is forwarded to the branch unit and the condition is evaluated.

Refer to **4.6.2 Conditional Branch Control** for more information on static branch prediction.

7.3 Serialization

The RCPU has multiple execution units, each of which may be executing different instructions at the same time. This concurrence is normally transparent to the user program. In certain circumstances, however (e.g., debugging, I/O control, and multi-processor synchronization), it may be necessary to force the machine to *serialize*.

Two types of serialization are defined for the RCPU: *execution serialization* and *fetch serialization*.

7.3.1 Execution Serialization

Execution serialization (also referred to as *serialization* or *execution synchronization*) causes the issue of subsequent instructions to be halted until all instructions currently in progress have completed execution, (i.e., all internal pipeline stages and instruction buffers have emptied and all outstanding memory transactions are completed).

An attempt to issue a serializing instruction causes the machine to serialize before the instruction issues. Notice that only the **sync** instruction guarantees serialization across PowerPC implementations.

7.3.2 Fetch Serialization

Fetch serialization (also referred to as “fetch synchronization”) causes instruction fetch to be halted until all instructions currently in the processor (i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued) have completed execution.

Fetch of an **isync** instruction causes fetch serialization. This means that no instructions following **isync** in the instruction stream are pre-fetched until **isync** and all previous instructions have completed execution. In addition, when the SER (serialize mode) bit in the ICTRL is asserted, or when the processor is in debug mode, all instructions cause fetch serialization.

7.4 Context Synchronization

The system call (**sc**) and return from interrupt (**rfi**) instructions are context-synchronizing. Execution of one of these instructions ensures the following:

- No higher priority exception exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an exception.
- Previous instructions complete execution in the context (privilege and protection) under which they were issued.
- The instructions following the context-synchronizing instruction execute in the context established by the instruction.

7.5 Implementation of Special-Purpose Registers

Most special-purpose registers supported by the RCPU are physically implemented within the processor. The following SPRs, however, are physically implemented outside of the processor (i.e., in another module, such as the system interface unit, of the microcontroller):

- Instruction cache control registers (ICCST, ICADR, and IDDAT)
- Time base (TB) and decrementer (DEC)
- Development port data register (DPDR)

These registers are read or written with the **mtspr** and **mfspr** instructions. The registers are physically accessed, however, via the internal L-bus or I-bus as appropriate.

The following encodings are reserved in the RCPU for SPRs not located within the processor:

Table 7-2 Encodings of External-to-the-Processor SPRs

SPR Instruction Field Encoding		Reserved for
SPR[5:9]	SPR[0:4]	
100xx	xxxxx	External-to-the-processor SPRs
100xx	x0xxx	System interface unit (SIU) route from L-bus to I-bus/internal SIU registers
100xx	x1xxx	Peripherals control unit registers
10011	x0xxx	SIU internal registers
0xxxx	xxxxx	DEC or TB, if this encoding appears on the L-bus
10000	x0xxx	Reserved for IBAT
10000	x1xxx	Reserved for DBAT
10001	x00xx	I-cache registers
10001	x1xxx	Reserved for D-cache

Many of the encodings in **Table 7-2** are not used in the RCPU. If the processor attempts to access to an unimplemented external-to-the-processor SPR, or if an error occurs during an access of an external-to-the-processor SPR, an implementation-dependent software emulation exception is taken (rather than a program exception).

An **mtspr** instruction to an external-to-the-processor register is not taken until all preceding instructions have terminated. Refer to **7.6 Instruction Execution Timing** for more information.

7.6 Instruction Execution Timing

Table 7-3 lists the instruction execution timing in terms of latency and blockage of the appropriate execution unit. Latency refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction. Blockage refers to the interval from the time an instruction begins execution until its execution unit is available for a subsequent instruction. Note that a serializing instruction has the effect of blocking all execution units.

Table 7-3 Instruction Execution Timing

Instructions	Latency	Blockage	Execution Unit	Serializing Instruction
Branch Instructions: **b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**, **bclr**, **bclrl**, **bcctr**, **bcctl**	Taken 2	2	BPU	No
	Not taken 1	1		
sc, **rfi**	Serialize + 2	Serialize + 2	BPU	Yes
CR logical instructions: **crand**, **crxor**, **cror**, **crnand**, **crnor**, **crandc**, **creqv**, **crorc**, **mcrf**	1	1	BPU	No
Fixed-point trap instructions: **twi**, **tw**	Taken Serialize + 3	Serialize + 3	ALU/BFU	After
	Not taken 1	1		No
mtspr to LR, CTR	1	1	BPU	No
mtspr to XER, external-to-the-processor SPRs[1]	Serialize + 1	Serialize + 1	LSU	Refer to **Table 7-4**
mtspr (to other registers)	Serialize + 1	Serialize + 1	BPU	Refer to **Table 7-4**
mfspr from external-to-the-processor SPRs[1]	Serialize + load latency	Serialize + 1	LSU	No
mfspr (from other registers)	1	1	BPU	Refer to **Table 7-4**
mftb, **mftbu**	Serialize + load latency	Serialize + 1	LSU	No
mtcrf, **mtmsr**	Serialize + 1	Serialize + 1	BPU	Yes
mfcr, **mfmsr**	Serialize + 1	Serialize + 1	BPU	No
mffs[.]	1	1	FPU	No
mcrxr	Serialize + 1	Serialize + 1	LSU, BPU	Yes
mcrfs	Serialize + 1	Serialize + 1	FPU, BPU	Yes
Other move FPSCR:**mtfsfi[.]**, **mtfsf[.]**, **mtfsb0[.]**, **mtfsb1[.]**	Serialize + 1	Serialize + 1	FPU	Yes
mcrxr	Serialize + 1	Serialize + 1	LSU	Yes (Before)
Integer arithmetic: **addi**, **add[o][.]**, **addis**, **subf[o][.]**, **addic**, **subfic**, **addic.**, **addc[o][.]**, **adde[o][.]**, **subfc[o][.]**, **subfe[o][.]**, **addme[o][.]**, **addze[o][.]**, **subfme[o][.]**, **subfze[o][.]**, **neg[o][.]**	1	1	ALU/BFU	No
Integer arithmetic (divide instructions): **divw[o][.]**, **divwu[o][.]**	Min 2 Max 11[2]	Min 2 Max 11[3]	IMUL/IDIV	No
Integer arithmetic (multiply instructions): **mulli**, **mull[o][.]**, **mulhw[.]**, **mulhwu[.]**	2	1-2[4]	IMUL/IDIV	No

Table 7-3 Instruction Execution Timing (Continued)

Instructions	Latency	Blockage	Execution Unit	Serializing Instruction
Integer compare: **cmpi**, **cmp**, **cmpli**, **cmpl**	1	1	ALU/BFU	No
Integer logical: **andi.**, **andis.**, **ori**, **oris**, **xori**, **xoris**, **and[.]**, **or[.]**, **xor[.]**, **nand[.]**, **nor[.]**, **eqv[.]**, **andc[.]**, **orc[.]**, **extsb[.]**, **extsh[.]**, **cntlzw[.]**	1	1	ALU/BFU	No
Integer rotate and shift: **rlwinm[.]**, **rlwnm[.]**, **rlwimi[.]**, **slw[.]**, **srw[.]**, **srawi[.]**, **sraw[.]**	1	1	ALU/BFU	No
Floating point move: **fmr[.]**, **fneg[.]**, **fabs[.]**, **fnabs[.]**	1	1	FPU	No
Floating point add/subtract: **fadd[.]**, **fadds[.]**, **fsub[.]**, **fsubs[.]**	4	4	FPU	No
Floating point multiply single: **fmuls[.]**	4	4	FPU	No
Floating point multiply double: **fmul[.]**	5	5	FPU	No
Floating point divide single: **fdivs[.]**	10	10	FPU	No
Floating point divide double: **fdiv[.]**	17	17	FPU	No
Floating point multiply-add single: **fmadds[.]**, **fmsubs[.]**, **fnmadds[.]**, **fnmsubs[.]**	6	6	FPU	No
Floating point multiply-add double: **fmadd[.]**, **fmsub[.]**, **fnmadd[.]**, **fnmsub[.]**	7	7	FPU	No
Floating round to single-precision: **frsp[.]**	2	2	FPU	No
Floating convert to integer: **fctiw[.]**, **fctiwz[.]**	3	3	FPU	No
Floating point compare: **fcmpu**, **fcmpo**	1	1	FPU	No
Integer load instructions: **lbz**, **lbzu**, **lbzx**, **lbzux**, **lhz**, **lhzu**, **lhzx**, **lhzux**, **lha**, **lhau**, **lhax**, **lhaux**, **lwz**, **lwzu**, **lwzx**, **lwzux**, **lhbrx**, **lwbrx**, **lhbrx**	2^5	1	LSU	No
Integer store instructions: **stb**, **stbu**, **stbx**, **stbux**, **sth**, **sthu**, **sthx**, **sthux**, **stw**, **stwu**, **stwbrx**	1^6	1	LSU	No
Integer load and store multiple instructions: **lmw**, **smw**	Serialize + 1 + Number of registers	Serialize + 1 + Number of registers	LSU	Yes
Synchronize: **sync**	Serialize + 1	Serialize + 1	LSU	Yes
Order storage access: **eieio**	Load/Store Serialize + 1	1	LSU	No

Table 7-3 Instruction Execution Timing (Continued)

Instructions	Latency	Blockage	Execution Unit	Serializing Instruction
Storage synchronization instructions: **lwarx**, **stwcx**.	Serialize + 2	Serialize + 2	LSU	Yes
Floating-point load single instructions: **lfs**, **lfsu**, **lfsx**, **lfsux**	2	1	LSU	No
Floating-point load double instructions: **lfd**, **lfdu**, **lfdx**, **lfdux**	3	1	LSU	No
Floating-point store single instructions: **stfs**, **stfsu**, **stfsx**, **stfsux**, **stfiwx**	1	1	LSU	No
Floating-point store double instructions: **stfd**, **stfdu**, **stfdx**, **stfdux**	1	1	LSU	No
String instructions: **lswi**, **lswx**, **stswi**, **stswx**	Serialize + 1 + Number of words accessed	Serialize + 1 + Number of words accessed	LSU	Yes
Storage control instructions: **isync**	serialize	serialize	BPU	Yes
eieio	1	1	LSU	Next load or store is serialized relative to all prior load or store
Cache control: **icbi**	1	1	LSU, I-cache	No

NOTES:

1. SPRs that are physically implemented outside of the RCPU are the time base, decrementer, ICCST, ICADR, IC-DAT, AND DPDR.

2. $$\text{DivisionLatency} = \begin{bmatrix} \text{NoOverflow} \Rightarrow 3 + \left\lfloor \frac{34 - \text{divisorLength}}{4} \right\rfloor \\ \text{Overflow} \Rightarrow 2 \end{bmatrix}$$

Where: $$\text{Overflow} = \left(\frac{x}{0}\right) \text{or} \left(\frac{\text{MaxNegativeNumber}}{-1}\right)$$

3. DivisionBlockage = DivisionLatency
4. Blockage of the multiply instruction is dependent on the subsequent instruction
   for subsequent multiply instruction the blockage is one clock.
   for subsequent divide instruction the blockage is two clocks.
5. Assuming non-speculative aligned access, on chip memory and available bus.
6. Although stores issued to the LSU buffers free the CPU pipeline, next load or store will not actually be performed on the bus until the bus is free.

Table 7-4 Control Registers and Serialized Access

SPR Number (Decimal)	Name	Serialize Access
1	XER	Write: full serialization Read: serialization relative to load/store operations
8	LR	No
9	CTR	No
18	DSISR	Write: full serialization Read: serialization relative to load/store operations
19	DAR	Write: full serialization Read: serialization relative to load/store operations
22	DEC	Write
26	SRR0	Write
27	SRR1	Write
80	EIE	Write
81	EID	Write
82	NRI	Write
144 – 147	CMPA – CMPD	Fetch serialized on write
148	ECR	Fetch serialized on write
149	DER	Fetch serialized on write
150	COUNTA	Fetch serialized on write
151	COUNTB	Fetch serialized on write
152 – 155	CMPE – CMPH	Write: fetch serialized Read: serialized relative to load/store operations
156	LCTRL1	Write: fetch serialized Read: serialized relative to load/store operations
157	LCTRL2	Write: fetch serialized Read: serialized relative to load/store operations
158	ICTRL	Fetch serialized on write
159	BAR	Write: fetch serialized Read: serialized relative to load/store operations
268	TBL read[1]	Write — as a store
269	TBU read[1]	Write — as a store
272 – 275	SPRG0 – SPRG3	Write
284	TBL write[2]	Write — as a store
285	TBU write[2]	Write — as a store
287	PVR	No (read only register)

Table 7-4 Control Registers and Serialized Access (Continued)

SPR Number (Decimal)	Name	Serialize Access
560	ICCST	Write — as a store
561	ICADR	Write — as a store
562	ICDAT	Write — as a store
630	DPDR	Read and write
1022	FPECR	Write
—	MSR	Fetch serialized on write
—	CR	Serialized for **mtcrf** only

NOTES:
1. Any write (**mtspr**) to this address results in an implementation-dependent software emulation exception.
2. Any read (**mftb**) of this address results in an implementation-dependent software emulation exception.

7.7 Instruction Execution Timing Examples

This section contains a number of examples illustrating the operation of the instruction pipeline. All examples assume an instruction cache hit.

7.7.1 Load from Internal Memory Example

This is an example of a load from an internal memory module with zero wait states.The **subf** instruction is dependent on the value loaded by the **lwz** to r12. This causes one bubble to occur in the instruction stream. See **7.7.3 Load with Private Write-Back Bus** for an example in which no such dependency exists.

```
lwz             r12,64(r0)
subf            r3,r12,r3
addic           r4,r14,1
mulli           r5,r3,3
addi            r4,r0,3
```

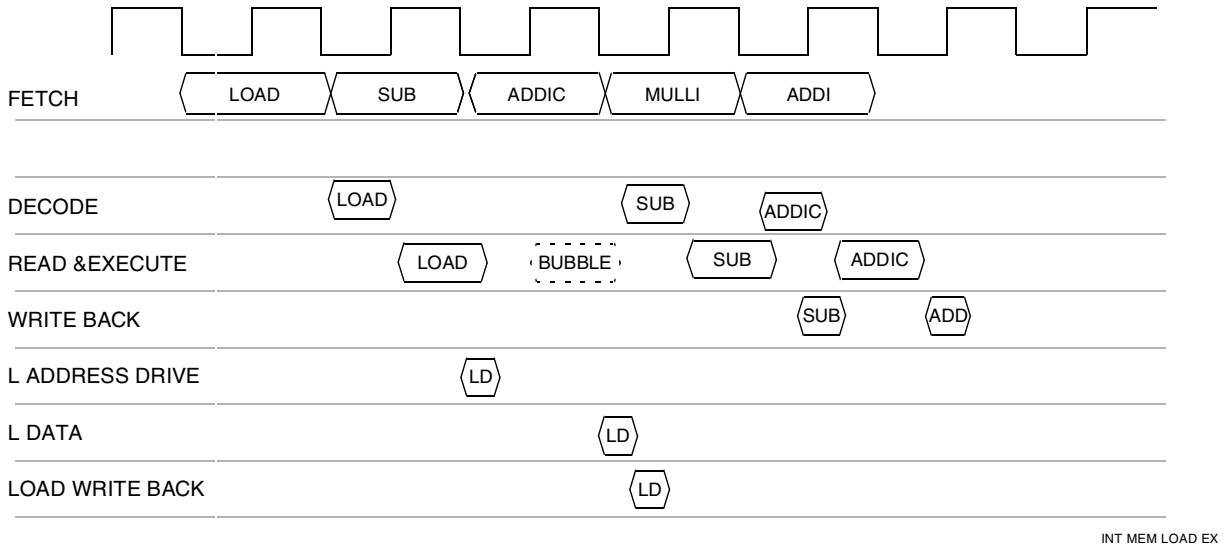

Figure 7-5 Load from Internal Memory Example

7.7.2 Write-Back Arbitration Examples

In the first example, the **addic** is dependent on the **mulli** result. Since the single cycle instruction **subf** has priority on the writeback bus over the **mulli**, the **mulli** write back is delayed one clock and causes a bubble in the execution stream.

```
mulli           r12,r4,3
subf            r3,r15,r3
addic           r4,r12,1
```

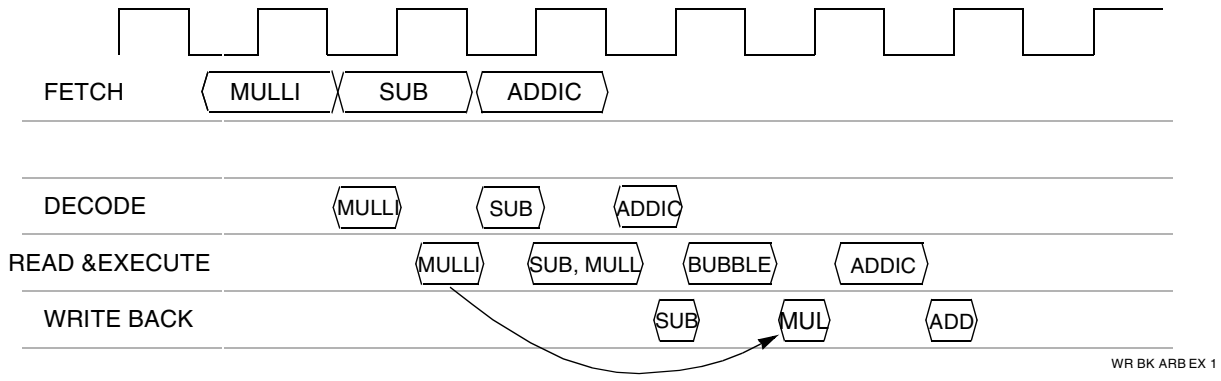

Figure 7-6 Write-Back Arbitration Example I

In the following example, the **addic** is dependent on the **subf** rather then on the **mulli**. Although the write back of the **mulli** is delayed two clocks, there is no bubble in the execution stream.

```
mulli           r12,r4,3
subf            r3,r15,r3
addic           r4,r3,1
```

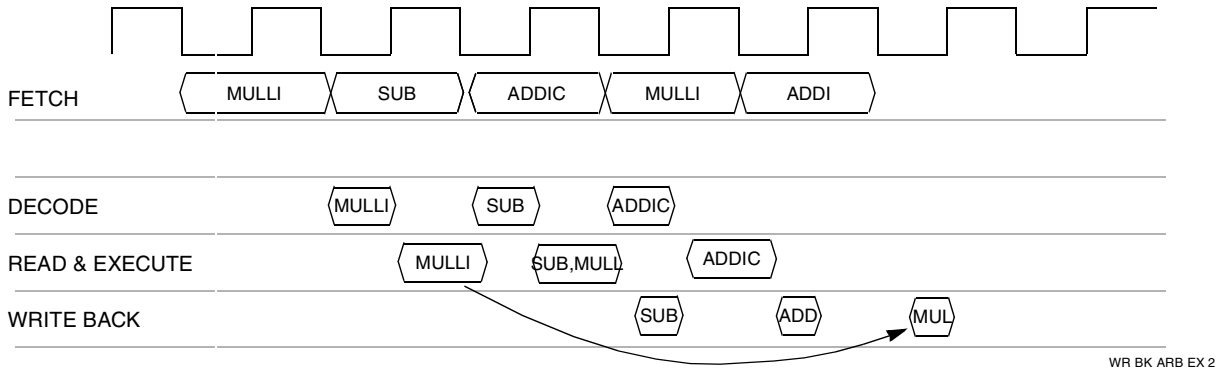

Figure 7-7 Write-Back Arbitration Example II

7.7.3 Load with Private Write-Back Bus

In this example, the **load** and the **and** write back in the same clock cycle, since they use the writeback bus during separate ticks.

```
lwz             r12,64(r0)
subf            r5,r3,r5
addic           r4,r14,1
and             r3,r4,r5
or              r6,r12,r3
```

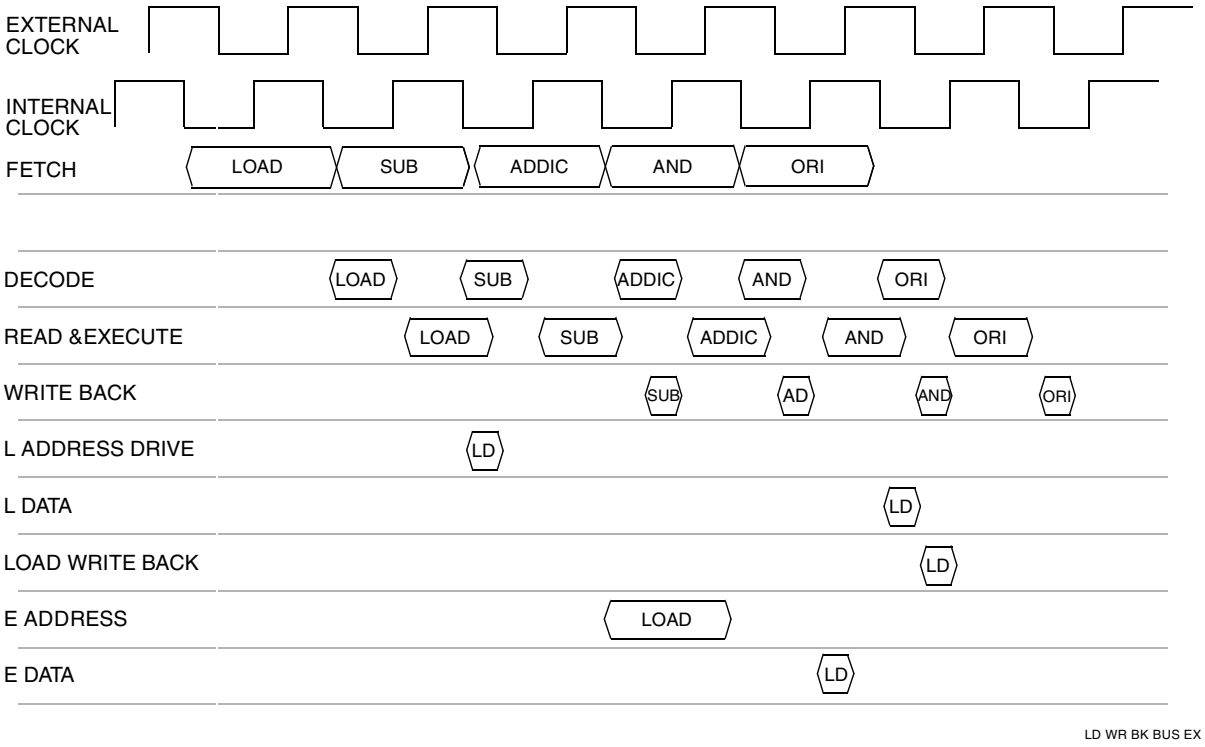

Figure 7-8 Load with Private Write-Back Bus Example

7.7.4 Fastest External Load Example

In this example, the **subf** is dependent on the value read by the **load**. It causes three bubbles in the instruction execution stream.

NOTE

The external clock is shifted 90° relative to the internal clock.

```
lwz             r12,64(r0)
subf            r3,r12,r3
addic           r4,r14,1
```

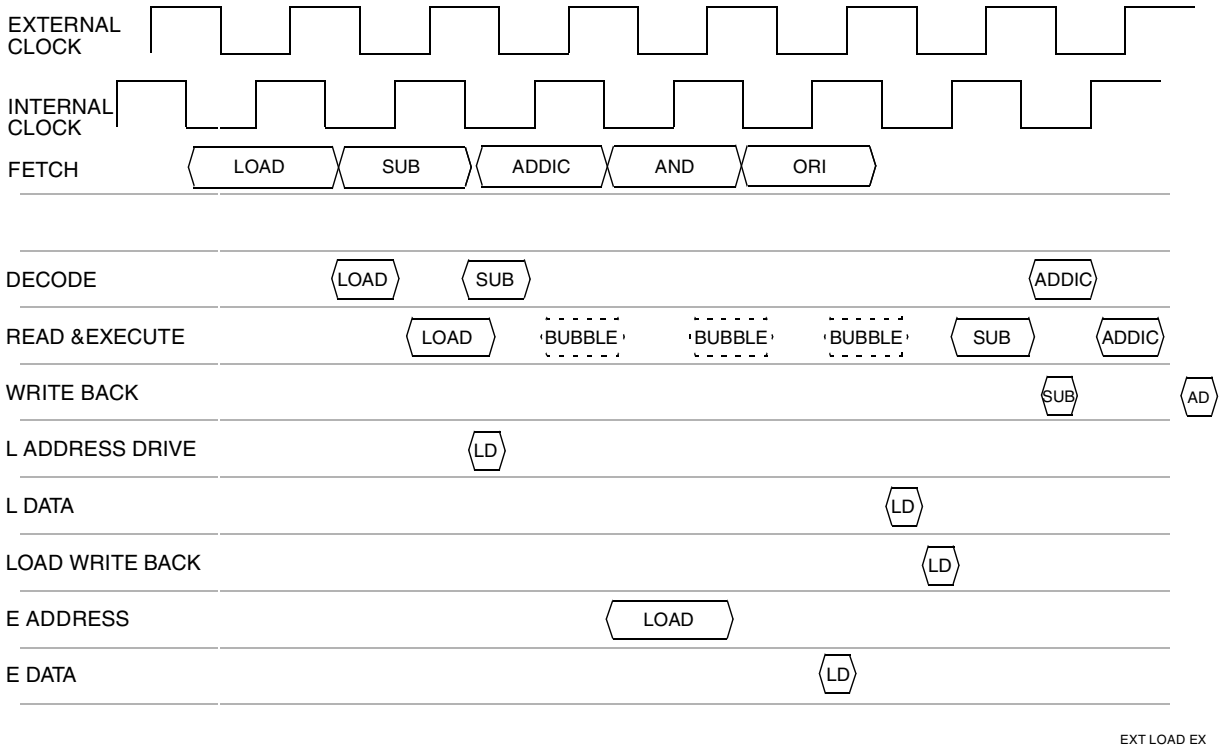

Figure 7-9 External Load Example

7.7.5 History Buffer Full Example

This example demonstrates the condition of a full history buffer. The floating-point history buffer is full by the **fadd** and two of the three **lfd**s.

NOTE

Following writeback of the **fadd** instruction, one additional bubble is required before instruction issue resumes. During this bubble, the history buffer retires the **fadd** instruction (as well as the two **lfd** instructions).

```
fadd        fr5,fr6,fr7
lfd         fr12,0(r2)
lfd         fr13,8(r2)
lfd         fr14,16(r2)
subf        r5,r3,r5
```

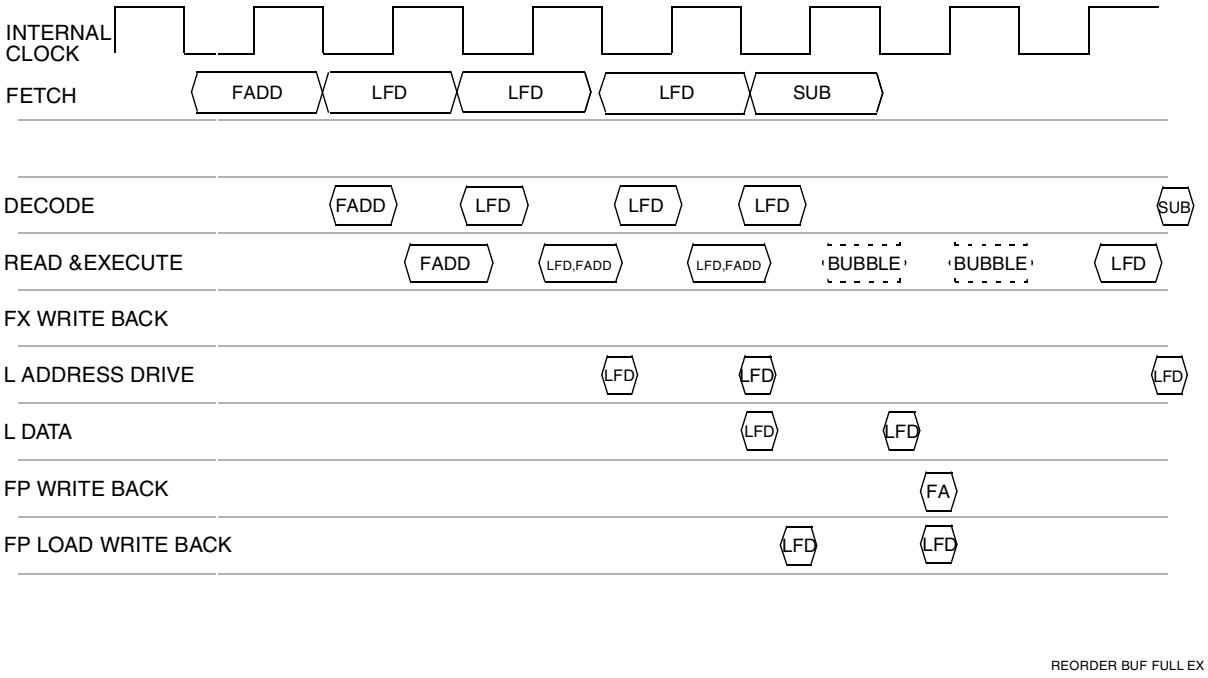

Figure 7-10 History Buffer Full Example

7.7.6 Store and Floating-Point Example

In this example the **stw** access on the L-bus is delayed until the **fadd** instruction is written back.

NOTE

In contrast to full serialization cases, the issue and execution of following instructions continue unaffected.

```
fadd        fr5,fr6,fr7
stw         r12,64(SP)
subf        r5,r5,r3
addic       r4,r14,1
fmul        fr3,fr4,fr5
or          r6,r12,r3
```

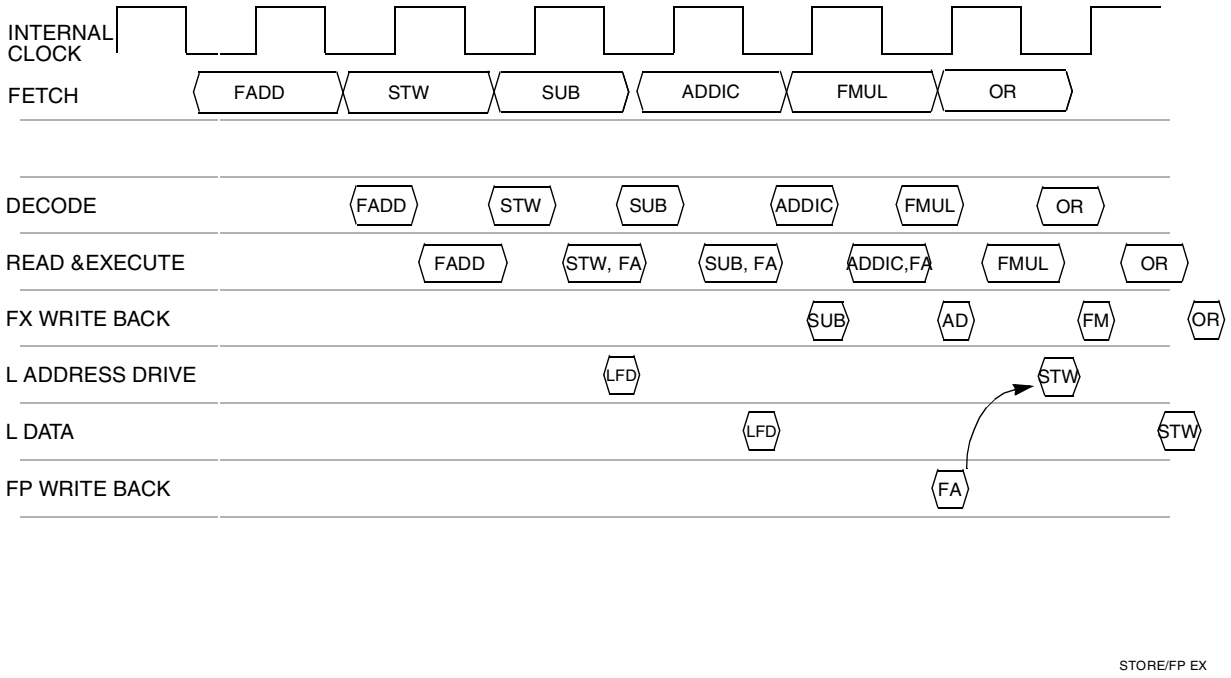

Figure 7-11 Store and Floating-Point Example

7.7.7 Branch Folding Example

In this example, the **lwz** instruction accesses internal storage with one wait state. The instruction prefetch queue and the parallel operation of the branch unit allow the two bubbles caused by the **bl** issue and execution to overlap the two bubbles caused by the **lwz** instruction.

```
            lwz         r12,64(SP)
            subf        r3,r12,r3
            addic       r4,r14,1
            bl          func
            ...
func:       mulli       r5,r3,3
            addi        r4,r0,3
```

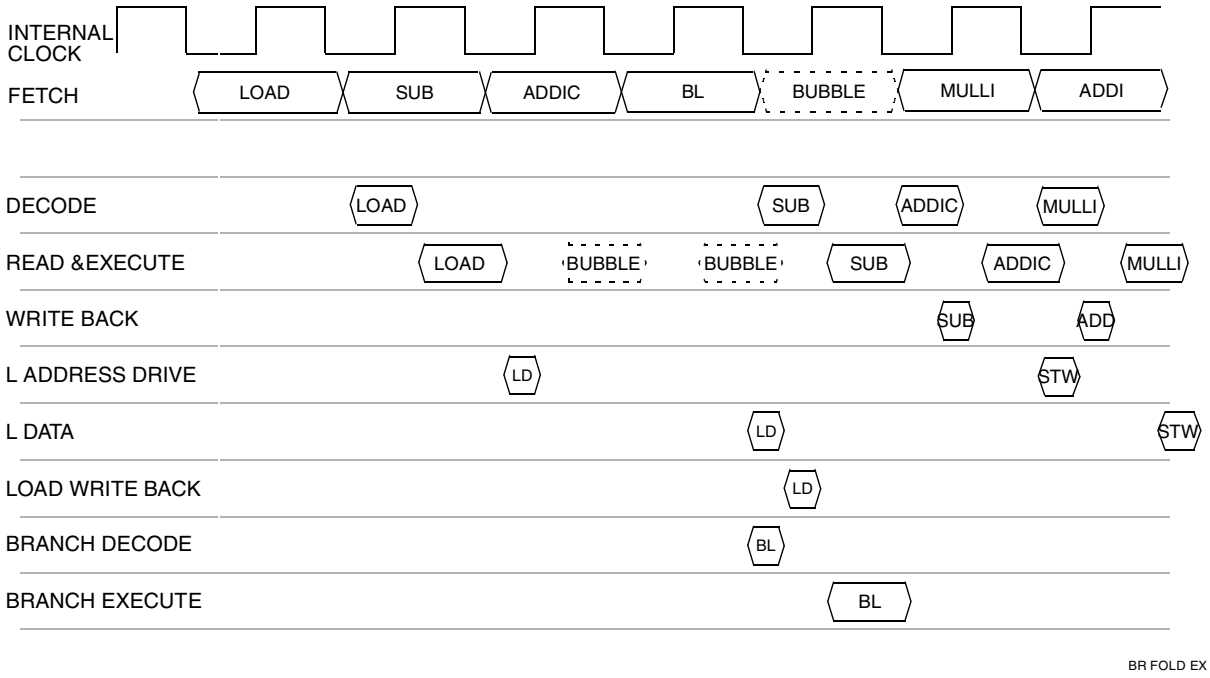

Figure 7-12 Branch Folding Example

7.7.8 Branch Prediction Example

In this example the **blt** instruction is dependent on the **cmpi**. The branch unit still predicts the correct path and allows the bubbles caused by the **blt** instruction to overlap the bubbles caused by the **ld** instruction, as in the previous example.

When the **cmpi** instruction is written back, the branch unit re-evaluates the decision. If the branch was correctly predicted, execution continues without interruption. The fetched instructions on the predicted path are not allowed to execute before the condition is finally resolved. Instead, they are stacked in the instruction prefetch queue.

```
while:      mulli           r3,r12,4
            addi            r4,r0,3
            ...
            lwz             r12,64(r2)
            cmpi            r12,3
            addic           r6,r5,1
            blt             while
            ...
```

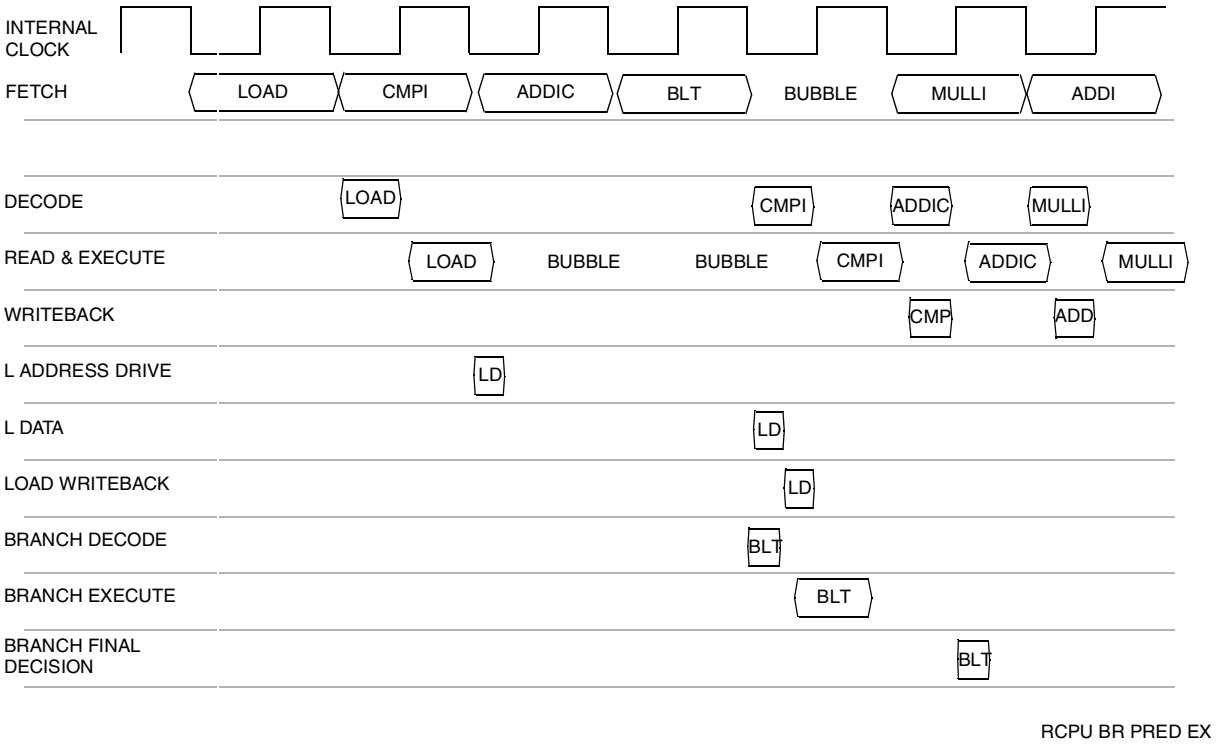

Figure 7-13 Branch Prediction Example