Chapter 3
Instruction Set Summary

Chapter 3 provides an overview of the LR333x0 instruction set. The instruction set is divided into categories of instructions and summarized in tabular form. This chapter contains the following sections:

- Instruction Notational Conventions
- Instruction Formats
- Load and Store Instructions
- Computational Instructions
- Jump and Branch Instructions
- Special Instructions
- Coprocessor Instructions
- System Control Coprocessor (CP0) Instructions
- The Instruction Pipeline
- The Delayed Instruction Slot

Compatibility Note

The LR333x0 does not support the R3000 instructions that relate to the Translation Lookaside Buffer (TLB), and it does not support many of the R3000 coprocessor instructions. Execution of these instructions causes the LR333x0 to trap. For details, refer to the instruction summaries in this chapter and to the LSI Logic document entitled *LR33000 Family Instruction Set.*

3.1 Instruction Notational Conventions

In this manual, all variable subfields in an instruction format (such as *rs, rt, immediate*, etc.) are shown in lower case.

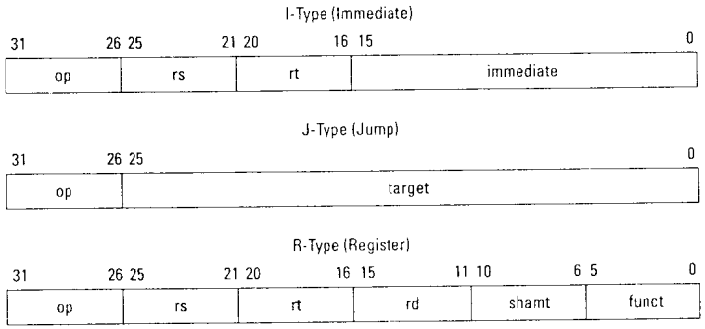
For the sake of clarity, an alias is sometimes used for a variable subfield for specific instruction formats. For example, *rs = base* is the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

The actual bit encoding for all the mnemonics is specified at the end of the *LR33000 Family Instruction Set*.

3.2 Instruction Formats

Every LR333x0 instruction consists of a single word (32 bits) aligned on a word boundary. As shown in Figure 3.1, there are three instruction formats: I-type (immediate), J-type (jump), and R-type (register). This restricted format approach simplifies instruction decoding. More complicated (and less frequently used) operations and addressing modes can be synthesized by the compiler and assembler.

Figure 3.1 LR333x0 Instruction Formats



Notes:

op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination register)
immediate	16-bit immediate, branch displacement, or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
shamt	5-bit shift amount
funct	6-bit function field

3.3 Load and Store Instructions

Load and store instructions move data between memory and general registers. They are all I-type instructions. The only addressing mode directly supported is base register plus 16-bit signed immediate offset.

All load operations have a delay, or latency, of one instruction. That is, the data being loaded from memory into a register is not available to the instruction that immediately follows the load instruction; the data is available to the *second* instruction after the load instruction. An exception is the target register for the "load word left" and "load word right" instructions, which may be specified as the same register as the destination of the load instruction immediately preceding it. Refer to Section 3.9, "The Instruction Pipeline," at the end of this chapter for a detailed discussion of load instruction latency.

The functions listed in Table 3.1 summarize the handling of addresses for load and store operations.

Table 3.1 Load/Store Common Functions

Function	*Description*
BIU	The Bus Interface Unit (BIU) maps accesses in *kseg0* and *kseg1* to 0x0000.0000 - 0x1FFF.FFFF and tests for cacheability.
Load Memory	The LR333x0 loads the addressed word from cache or main memory. If the cache is enabled for this access, the word is returned and loaded into the cache.
Store Memory	The LR333x0 stores the data into the cache and/or main memory at the specified address.

The load/store instruction operation code (opcode) determines the access type, which in turn indicates the size of the data item to be loaded or stored. Regardless of access type or byte-numbering order (endianness), the address specifies the byte that has the smallest byte address of all the bytes in the addressed field. For a big-endian machine, this is the leftmost byte; for a little-endian machine, this is the rightmost byte.

The bytes that are used within the addressed word can be determined directly from the access type and the two low-order bits of the address, as shown in Figure 3.2. Note that certain combinations of access type and low-order address bits can never occur; only the combinations shown in Figure 3.2 are permissible.

Figure 3.2
Byte Specifications for Loads/Stores

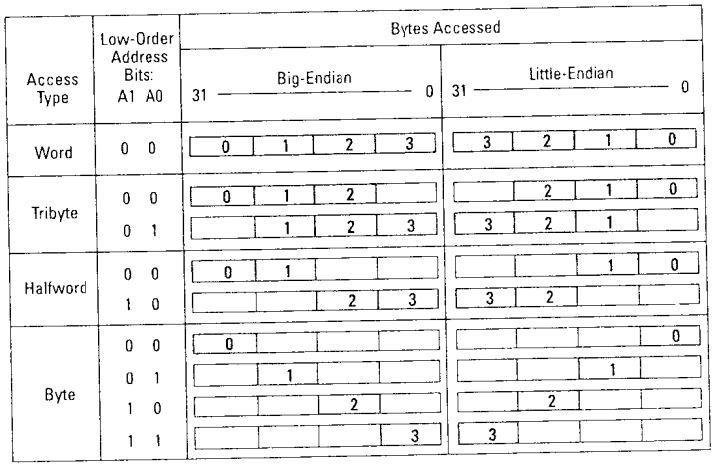


Table 3.2 summarizes the LR333x0 load and store instructions.

Table 3.2
Load and Store
Instruction Summary

Instruction	*Format and Description*
Load Byte	*LB rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Sign-extend contents of addressed byte and load into *rt*.
Load Byte Unsigned	*LBU rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Zero-extend contents of addressed byte and load into *rt*.
Load Halfword	*LH rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Sign-extend contents of addressed halfword and load into *rt*.
Load Halfword Unsigned	*LHU rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Zero-extend contents of addressed halfword and load into *rt*.
Load Word	*LW rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address, and load the addressed word into *rt*.
Load Word Left	*LWL rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift addressed word left so that addressed byte is leftmost byte of a word. Merge bytes from memory with contents of register *rt* and load result into register *rt*.

(Sheet 1 of 2)

Table 3.2 (Cont.)
Load and Store
Instruction Summary

Instruction	*Format and Description*
Load Word Right	*LWR rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift addressed word right so that addressed byte is rightmost byte of a word. Merge bytes from memory with contents of register *rt* and load result into register *rt*.
Store Byte	*SB rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store least-significant byte of register *rt* at addressed location.
Store Halfword	*SH rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store least-significant halfword of register *rt* at addressed location.
Store Word	*SW rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store contents of register *rt* at addressed location.
Store Word Left	*SWL rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift contents of register *rt* right so that the leftmost byte of the word is in the position of the addressed byte. Store word containing shifted bytes into word at addressed byte.
Store Word Right	*SWR rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift contents of register *rt* left so that the rightmost byte of the word is in the position of the addressed byte. Store word containing shifted bytes into word at addressed byte.

(Sheet 2 of 2)

3.4 Computational Instructions

Computational instructions perform arithmetic, logical, and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats. There are four categories of computational instructions:

- ALU Immediate instructions are summarized in Table 3.3.
- 3-Operand, Register-Type instructions are summarized in Table 3.4.
- Shift instructions are summarized in Table 3.5.
- Multiply/Divide instructions are summarized in Table 3.6.

Table 3.3
ALU Immediate
Instruction Summary

Instruction	*Format and Description*
Add Immediate	*ADDI rt, rs, immediate* Add 16-bit, sign-extended *immediate* to register *rs* and place 32-bit result in register *rt*. Trap on two's complement overflow.
Add Immediate Unsigned	*ADDIU rt, rs, immediate* Add 16-bit, sign-extended *immediate* to register *rs* and place 32-bit result in register *rt*. Do not trap on overflow.
Set on Less Than Immediate	*SLTI rt, rs, immediate* Compare 16-bit, sign-extended *immediate* with register *rs* as signed 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*.
Set on Less Than Immediate Unsigned	*SLTIU rt, rs, immediate* Compare 16-bit, sign-extended *immediate* with register *rs* as unsigned 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*.
AND Immediate	*ANDI rt, rs, immediate* Zero-extend 16-bit *immediate*, AND with contents of register *rs*, and place result in register *rt*.
OR Immediate	*ORI rt, rs, immediate* Zero-extend 16-bit *immediate*, OR with contents of register *rs*, and place result in register *rt*.
Exclusive OR Immediate	*XORI rt, rs, immediate* Zero-extend 16-bit *immediate*, exclusive OR with contents of register *rs*, and place result in register *rt*.
Load Upper Immediate	*LUI rt, immediate* Shift 16-bit *immediate* left 16 bits. Set least-significant 16 bits of word to zeros. Store result in register *rt*.

Table 3.4
3-Operand, Register-Type
Instruction Summary

Instruction	*Format and Description*
Add	*ADD rd, rs, rt* Add contents of registers *rs* and *rt* and place 32-bit result in register *rd*. Trap on two's complement overflow.
Add Unsigned	*ADDU rd, rs, rt* Add contents of registers *rs* and *rt* and place 32-bit result in register *rd*. Do not trap on overflow.
Subtract	*SUB rd, rs, rt* Subtract contents of registers *rt* from *rs* and place 32-bit result in register *rd*. Trap on two's complement overflow.
Subtract Unsigned	*SUBU rd, rs, rt* Subtract contents of registers *rt* from *rs* and place 32-bit result in register *rd*. Do not trap on overflow.
Set on Less Than	*SLT rd, rs, rt* Compare contents of register *rt* to register *rs* (as signed, 32-bit integers). If register *rs* is less than *rt*, *rd* = 1; otherwise, *rd* = 0.
Set on Less Than Unsigned	*SLTU rd, rs, rt* Compare contents of register *rt* to register *rs* (as unsigned, 32-bit integers.). If register *rs* is less than *rt*, *rd* = 1; otherwise, *rd* = 0.
AND	AND *rd, rs, rt* Bitwise AND contents of registers *rs* and *rt* and place result in register *rd*.
OR	*OR rd, rs, rt* Bitwise OR contents of registers *rs* and *rt* and place result in register *rd*.
Exclusive OR	*XOR rd, rs, rt* Bitwise exclusive OR contents of registers *rs* and *rt* and place result in register *rd*.
NOR	*NOR rd, rs, rt* Bitwise NOR contents of registers *rs* and *rt* and place result in register *rd*.

Table 3.5
Shift Instruction Summary

Instruction	*Format and Description*
Shift Left Logical	*SLL rd, rt, shamt* Shift contents of register *rt* left by *shamt* bits, inserting zeros into low-order bits. Place 32-bit result in register *rd*.
Shift Right Logical	*SRL rd, rt, shamt* Shift contents of register *rt* right by *shamt* bits, inserting zeros into high-order bits. Place 32-bit result in register *rd*.
Shift Right Arithmetic	*SRA, rd, rt, shamt* Shift contents of register *rt* right by *shamt* bits, sign-extending the high-order bits. Place 32-bit result in register *rd*.
Shift Left Logical Variable	*SLLV rd, rt, rs* Shift contents of register *rt* left. Low-order 5 bits of register *rs* specify the number of bits to shift. Insert zeros into low-order bits of *rt* and place 32-bit result in register *rd*.
Shift Right Logical Variable	*SRLV rd, rt, rs* Shift contents of register *rt* right. Low-order 5 bits of register *rs* specify the number of bits to shift. Insert zeros into high-order bits of *rt* and place 32-bit result in register *rd*.
Shift Right Arithmetic Variable	*SRAV rd, rt, rs* Shift contents of register *rt* right. Low-order 5 bits of register *rs* specify the number of bits to shift. Sign-extend the high-order bits of *rt* and place 32-bit result in register *rd*.

Table 3.6
Multiply/Divide Instruction Summary

Instruction	*Format and Description*
Multiply	*MULT rs, rt* Multiply contents of registers *rs* and *rt* as two's complement values. Place 64-bit results in special registers HI and LO.
Multiply Unsigned	*MULTU rs, rt* Multiply contents of registers *rs* and *rt* as unsigned values. Place 64-bit results in special registers HI and LO.
Divide	*DIV rs, rt* Divide contents of registers *rs* and *rt* as two's complement values. Place 32-bit quotient in special register LO and 32-bit remainder in HI.
Divide Unsigned	*DIVU rs, rt* Divide contents of registers *rs* and *rt* as unsigned values. Place 32-bit quotient in special register LO and 32-bit remainder in HI.
Move From HI	*MFHI rd* Move contents of special register HI to register *rd*.
Move From LO	*MFLO rd* Move contents of special register LO to register *rd*.
Move To HI	*MTHI rd* Move contents of register *rd* to special register HI.
Move To LO	*MTLO rd* Move contents of register *rd* to special register LO.

The execution time of the multiply instructions is as follows:

Table 3.7
Execution Time of Multiply Instructions

Multiply Operands	*Number of Cycles*
any operand < 512	4
512 ≤ any operand < 256K	7
256K ≤ both operands	12

3.5 Jump and Branch Instructions

Jump and Branch instructions change the control flow of a program. All jump and branch instructions occur with a one-instruction delay. That is, the instruction immediately following the jump or branch is always executed while the target instruction is being fetched from storage. Refer to Section 3.10, "The Delayed Instruction Slot," for a detailed discussion of the delayed jump and branch instructions.

The J-type instruction format is used for both jump and jump-and-link instructions for subroutine calls. In this format, the 26-bit target address is shifted left two bits and combined with the four high-order bits of the current program counter to form a 32-bit absolute address.

The R-type instruction format, which takes a 32-bit byte address contained in a register, is used for returns, dispatches, and cross-page jumps.

Branches have 16-bit signed offsets relative to the program counter (I-type). Jump-and-link and branch-and-link instructions save a return address in register 31.

Table 3.8 summarizes the LR333x0 jump instructions, and Table 3.9 summarizes the branch instructions.

Table 3.8
Jump Instruction Summary

Instruction	*Format and Description*
Jump	*J target* Shift 26-bit *target* address left two bits, combine with four high-order bits of PC, and jump to address with a one-instruction delay.
Jump and Link	*JAL target* Shift 26-bit *target* address left two bits, combine with four high-order bits of PC, and jump to address with a one-instruction delay. Place address of instruction following delay slot in *r31* (link register).
Jump Register	*JR rs* Jump to address contained in register *rs* with a one-instruction delay.
Jump and Link Register	*JALR rs, rd* Jump to address contained in register *rs* with a one-instruction delay. Place address of instruction following delay slot in *rd*.

Table 3.9
Branch Instruction Summary

Instruction	*Format and Description*
Branch on Equal	*BEQ rs, rt, offset* Branch to target address[1] if register *rs* is equal to register *rt*.
Branch on Not Equal	*BNE rs, rt, offset* Branch to target address if register *rs* does not equal register *rt*.
Branch on Less than or Equal to Zero	*BLEZ rs, offset* Branch to target address if register *rs* is less than or equal to 0.
Branch on Greater Than Zero	*BGTZ rs, offset* Branch to target address if register *rs* is greater than 0.
Branch on Less Than Zero	*BLTZ rs, offset* Branch to target address if register *rs* is less than 0.
Branch on Greater than or Equal to Zero	*BGEZ rs, offset* Branch to target address if register *rs* is greater than or equal to 0.
Branch on Less Than Zero And Link	*BLTZAL rs, offset* Place address of instruction following delay slot in register *r31* (link register). Branch to target address if register *rs* is less than 0.
Branch on Greater than or Equal to Zero and Link	*BGEZAL rs, offset* Place address of instruction following delay slot in register *r31* (link register). Branch to target address if register *rs* is greater than or equal to 0.

1. All branch-instruction target addresses are computed as follows: add address of instruction in delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

3.6 Special Instructions

Special instructions cause an unconditional branch to the general exception-handling vector. Special instructions are always R-type. Table 3.10 summarizes these instructions.

Table 3.10
Special Instruction Summary

Instruction	*Format and Description*
System Call	*SYSCALL* Initiates system call trap, immediately transferring control to exception handler.
Breakpoint	*BREAK* Initiates breakpoint trap, immediately transferring control to exception handler.

3.7 Coprocessor Instructions

The LR333x0 does not support external coprocessors and—with two exceptions—does not implement the Coprocessor instruction set. The exceptions are the Branch on Coprocessor z True (BCzT) and Branch on Coprocessor z False (BCzF) instructions, which allow software to test the condition of the CPC[3:0] signals on the LR333x0's external interface. The remainder of the coprocessor instructions produce undefined results if executed.

Because most of the coprocessor instructions produce undefined results, software should disable the coprocessors by setting the Cu[3:1] bits in the Status Register to zero. When the coprocessors are disabled in this fashion, coprocessor instructions in the instruction stream will cause the LR333x0 to take a Coprocessor Unusable (CpU) Exception. To use the BCzT and BCzF instructions, software must enable the coprocessor associated with the signal to be tested.

Compatibility Note

When in kernel mode with Cu0 set to zero, the LR333x0 takes a CpU Exception when it decodes a BC0T or BC0F instruction. In like conditions, however, the R3000 *does not* take an exception. In addition, SWC0 and LWC0 are undefined and must not be used. *Executing the SWC0 and LWC0 instructions may cause problems.*

Coprocessor branch instructions are J-type. Table 3.11 summarizes the different coprocessor instructions.

Table 3.11 Coprocessor Instruction Summary

Instruction	*Format and Description*
Load Word to Coprocessor	*LWCz rt, offset(base)* The LR333x0 does not implement this instruction.
Store Word from Coprocessor	*SWCz rt, offset(base)* The LR333x0 does not implement this instruction.
Move To Coprocessor	*MTCz rt, rd* The LR333x0 does not implement this instruction for z = 1-3.
Move From Coprocessor	*MFCz rt, rd* The LR333x0 does not implement this instruction for z = 1-3.
Move Control to Coprocessor	*CTCz rt, rd* The LR333x0 does not implement this instruction.
Move Control From Coprocessor	*CFCz rt, rd* The LR333x0 does not implement this instruction.
Coprocessor Operation	*COPz cofun* The LR333x0 does not implement this instruction.
Branch on Coprocessor z True	*BCzT offset* Compute a branch target address by adding address of instruction to the 16-bit *offset* (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor z's condition line is true.
Branch on Coprocessor z False	*BCzF offset* Compute a branch target address by adding address of instruction to the 16-bit *offset* (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor z's condition line is false.

3.8 System Control Coprocessor (CP0) Instructions

Coprocessor 0 instructions perform operations on the system control coprocessor (CP0) registers to manipulate the memory management and exception-handling facilities of the processor. Table 3.12 summarizes the CP0 instructions.

Because the LR333x0 has no Translation Lookaside Buffer (TLB), it implements none of the TLB read, write, and probe entry instructions. The LR333x0 takes a Reserved Instruction (RI) Exception if it decodes a TLB access instruction.

Compatibility Note

When in user mode with Cu0 set to zero, the LR333x0 takes a Reserved Instruction (RI) Exception if it decodes an RFE, MTC0, or MFC0 instruction. In like conditions, the R3000 takes a Coprocessor Unusable Exception (CpU).

Table 3.12
CP0 Instruction Summary

Instruction	*Format and Description*
Move To CP0	*MTC0 rt, rd* Load contents of CPU register *rt* into CP0 register *rd*.
Move From CP0	*MFC0 rt, rd* Load contents of CP0 register *rd* into CPU register *rt*.
Read Indexed TLB Entry	*TLBR* The LR333x0 does not implement this instruction.
Write Indexed TLB Entry	*TLBWI* The LR333x0 does not implement this instruction.
Write Random TLB Entry	*TLBWR* The LR333x0 does not implement this instruction.
Probe TLB for Matching Entry	*TLBP* The LR333x0 does not implement this instruction.
Restore From Exception	*RFE* Restore previous interrupt mask and mode bits of Status register into current status bits. Restore old status bits into previous status bits.

3.9 The Instruction Pipeline

The execution of a single instruction consists of five primary steps or pipestages:

Step 1. **IF** – Instruction Fetch. Generate the instruction address required to read an instruction from the I-Cache. The instruction is not actually read into the processor until the beginning (phase 1) of the RD pipestage.

Step 2. **RD** – Read any required operands from CPU registers while decoding the instruction (ID = instruction decode).

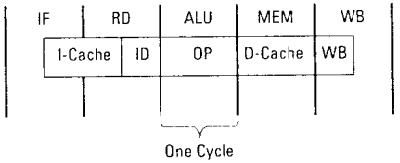
Step 3. **ALU** – Perform the required operation on instruction operands (OP).

Step 4. **MEM** – Access memory (D-Cache) if required (for a load or store instruction).

Step 5. **WB** – Write back ALU results to register file.

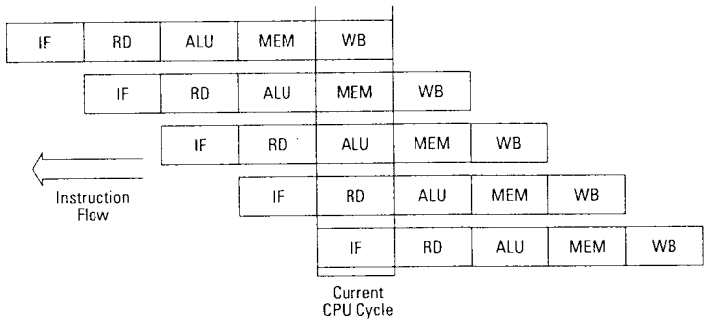
Each of these steps requires approximately one CPU cycle as shown in Figure 3.3 (parts of some operations lap over into another cycle, whereas other operations require only 1/2 cycle).

Figure 3.3
Instruction Execution Sequence



To achieve an instruction execution rate approaching one instruction per CPU cycle, a five-instruction pipeline is used. Thus after the pipeline is filled with the initial five instructions, five instructions are executed in a single cycle, as shown in Figure 3.4.

Figure 3.4
LR333x0 Instruction Pipeline



3.10 The Delayed Instruction Slot

The LR333x0 uses a number of techniques internally to enable execution of all instructions in a single cycle; however, two categories of instructions have special requirements which could disturb the smooth flow of instructions through the pipeline:

- Load instructions have a latency of one cycle before the data being loaded is available to another instruction.
- Jump and branch instructions also have a delay of one cycle while they fetch the instruction and the target address if the branch is taken.

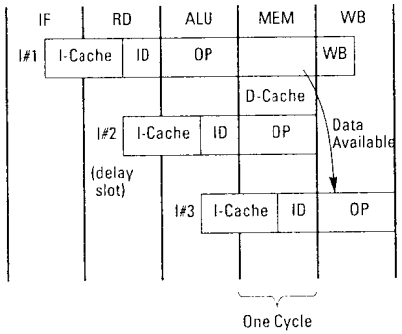
One technique for dealing with the delay inherent with these instructions would be to stall the flow of instructions through the pipeline whenever a load, jump, or branch is executed. However, in addition to the negative impact that this technique would have on instruction throughput, it would also complicate the pipeline logic, exception processing, and system synchronization.

The technique used in the LR333x0 is to continue execution despite the delay. Loads, jumps, and branches do not interrupt the normal flow of instructions through the pipeline; *the processor always executes the instruction immediately following one of these "delayed" instructions.* Instead of dealing with pipeline delays, the LR333x0 relies on software to place useful instructions in the delay slot. Thus an assembler can insert an appropriate instruction immediately following a delayed instruction and must make sure that the inserted instruction is not affected by the delay.

Delayed Loads

Figure 3.5 shows three instructions in the LR333x0 pipeline. Instruction 1 (I#1) is a load instruction. The data from the load is not available until the end of the I#1 MEM cycle—too late to be used by I#2 during its ALU cycle, but available to I#3 for its ALU cycle. Therefore software must ensure that I#2 does not depend on data loaded by I#1. Usually, a compiler can reorganize instructions so that something useful is executed during the delay slot or, if no other instruction is available, can insert a NOP (no operation) instruction into the slot.

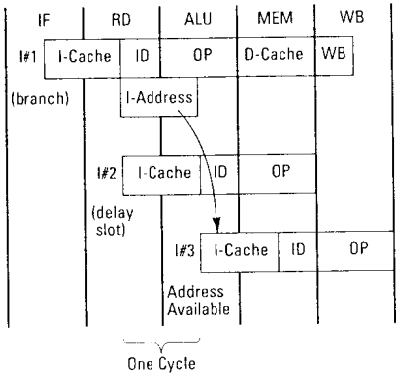
Figure 3.5 The Load Instruction Delay Slot



Delayed Jumps and Branches

Figure 3.6 shows three instructions in the LR333x0 pipeline. In this case, Instruction 1 (I#1) is a branch instruction. I#1 must calculate a branch target address. That address is not available until the beginning of the ALU cycle of I#1—too late for the I-Cache access of I#2, but available to I#3 for its I-Cache access. The instruction in the delay slot (I#2) is always executed before the branch or jump actually occurs.

Figure 3.6 The Jump/Branch Instruction Delay Slot



An assembler has several possibilities for using the branch delay slot productively:

- It can insert an instruction that logically precedes the branch instruction in the delay slot, because the instruction immediately following the jump/branch effectively belongs to the block preceding the transfer instruction.
- It can replicate the instruction that is the target of the branch/jump into the delay slot, provided that no side-effects occur if the branch falls through.
- It can move an instruction up from below the branch into the delay slot, provided that no side-effects occur if the branch is taken.
- If no other instruction is available, it can insert a NOP instruction in the delay slot.

Branch Interpretation Support

The Branch Delay (BD) and Branch Taken (BT) bits in the Cause Register provide support for the exception handler. When the cause of an exception is in the branch delay slot (the LR333x0 sets the BD bit to one), execution resumes either at the target of the branch or at the Exception Program Counter (EPC) + 8. If the branch was taken, the LR333x0 sets the BT bit to one. The branch target address is located in the Target Address Register. The exception handler needs only to load this address into a register and jump to that location.