RC 20733 (91858) 2/17/97
Computer Sciences/Mathematics

IBM Research Report

ForestaPC (Scalable-VLIW) User Instruction Set Architecture

Jaime H. Moreno
jmoreno@watson.ibm.com

Kemal Ebcioglu
kemal@watson.ibm.com

Mayan Moudgill
mayan@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Dave Luick
luick@rchvmx.vnet.ibm.com

IBM AS/400 Division
3605 Highway 52 N
Rochester, MN 55901

Research Division
Almaden ● Austin ● China ● Haifa ● Tokyo ● T.J. Watson ● Zurich

ForestaPC (Scalable-VLIW)
User Instruction Set Architecture

Book I
Version 1.0

December 27, 1996

Preface

This document defines the ForestaPC User Instruction Set Architecture. It covers the base instruction set and related facilities available to the application programmer.

Other related documents are:

- *Book II, ForestaPC Virtual Environment Architecture*, which defines the storage model and related instructions and facilities available to the application programmer;
- *Book III, ForestaPC Operating Environment Architecture*, which defines the system (privileged) instructions and related facilities; and
- *Book IV, ForestaPC Implementation Features*, which defines the implementation-dependent aspects of a particular implementation.

As used in this document, the term “ForestaPC Architecture” refers to the instructions and facilities described in *Books I, II,* and *III*. The description of an instance of the ForestaPC Architecture in a given implementation also includes the material in *Book IV* for that implementation.

Table of Contents

Chapter 1. Introduction and Formats

This chapter gives an overview of the *ForestaPC architecture*, discusses the compatibility among the ForestaPC architecture and the PowerPC architecture, describes the format of the ForestaPC instructions, the classes and format of primitive instructions, the exceptions, and the storage addressing.

1.1 Processor Overview

1.1.1 Basic Description

The ForestaPC architecture defines the register set, the instruction set, the storage model, and other facilities described in this document. This architecture is tailored for extensive exploitation of *instruction-level parallelism (ILP)* in programs, that is, for executing many basic (primitive) instructions at a time. ForestaPC is a *scalable-VLIW* architecture, which also allows implementations exploiting limited instruction-level parallelism (*superscalar* sequential processors).

Implementations of the ForestaPC architecture contain many functional units which are used simultaneously for the execution of multiple primitive instructions.

The ForestaPC architecture allows the following types of implementations:

- 64-bit implementations, in which all registers excepting some Special Purpose Registers are 64 bits long, and effective addresses are 64-bits long. All 64-bit implementations have two modes of computation: 64-bit mode and 32-bit mode. This mode controls how the effective address is interpreted and how status bits are set. All instructions provided for 64-bit implementations are available in both modes.
- 32-bit implementations, in which all registers except the Condition Register and the Floating-Point Registers are 32-bits long, and effective addresses are 32-bits long.

The instructions defined in this document are provided in 64-bit and 32-bit implementations unless stated otherwise. Instructions provided only for 64-bit implementations are illegal in 32-bit implementations, and vice-versa.

The ForestaPC architecture has two distinct modes of operation, each with a different user instruction set architecture:

- **VLIW Native mode,** in which programs in storage have an explicit representation of ILP in the form of *tree-instructions*, as defined in this document.
- **PowerPC mode,** in which programs comply with the definitions in the *PowerPC Architecture*. In this mode, a program in storage contains PowerPC primitive instructions.

See *Book I, PowerPC User Instruction Set Architecture,* for additional information regarding the user instruction set architecture in PowerPC mode.

Unless explicitly stated otherwise, the description given in this document refers to VLIW Native mode.

The base mode (VLIW Native mode or PowerPC mode) in use for an instruction is determined by a bit in the Page Table Entry for the page that contains the instruction. Thus, a base mode change is accomplished by simply branching to a page which contains instructions in the mode other than the one in use at a given time. No synchronization instructions are required. Normally, this branch should occur at a function call boundary, so that programs in both

modes comply with standard call conventions. A base mode change also occurs when sequential execution flows to a page with instructions in a different mode, but this would not usually be done.

64-bit Implementations

In 64-bit mode and 32-bit mode of a 64-bit implementation, instructions that set a 64-bit register affect all 64-bits, and the value placed in the register is independent of mode. In both modes, effective address computations use all 64-bits of the relevant registers, and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored when accessing data, and are set to 0 when fetching instructions.

32-bit Implementations

For a 32-bit implementation, all reference to 64-bit mode in this document should be disregarded. The semantics of instructions are as shown in this document for 32-bit mode in a 64-bit implementation, except that in a 32-bit implementation all registers other than the Condition Register and the Floating-Point Registers are 32-bits long. Bit numbers for registers are shown in braces ({}) when they differ from the corresponding numbers for a 64-bit implementation, as described in Section 1.4.1, “Definitions and Notation,” on page 9.

VLIW Native Mode

A program executed by a ForestaPC processor in VLIW Native mode consists of a sequence of *tree-instructions* (or simply *trees*), each of which corresponds to an *unlimited multiway-branch* and an *unlimited set of operations (primitive instructions)*. The multiway-branch is associated with the internal nodes of a tree, whereas the operations are associated with the arcs (see Figure 1). The multiway-branch is the result of a set of *binary tests on conditions codes*; the left outgoing arc from a tree node corresponds to the false outcome of the test, whereas the right outgoing arc corresponds to the true outcome of the test.

Primitive instructions in a tree are subject to *sequential semantics for each path of the tree,* as if each primitive instruction were executed in the order in which it appears in the tree-path (a tree-path starts from the root of the tree and ends in a destination target). As a result, a primitive instruction cannot use a processor resource which is the target of a previous instruction in the same tree-path. This requirement may not be checked nor enforced by the hardware. If this requirement is not fulfilled within a tree-instruction, the results from primitive instructions having an operand set by a previous instruction in the same tree-path are undefined.

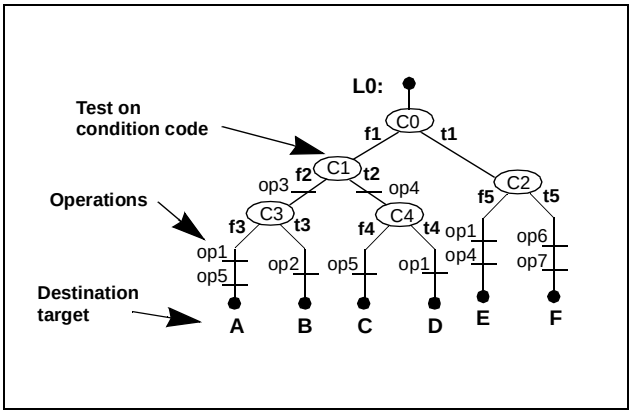


Figure 1: Tree instruction

Architecture Note: Sequential semantics in each tree-path is required to guarantee binary compatibility among different implementations of the ForestaPC architecture, with varying degrees of parallel execution capabilities, so that large trees can be decomposed into subtrees which are executed in different cycles.

A tree-instruction is represented in main storage as a *contiguous sequence of primitive instructions*, wherein each primitive instruction is encoded in one memory word. The sequence of primitive instructions is obtained from the *depth-first traversal* of the tree-instruction (see Figure 2).

```
L0: skip C0,t1          t4: op1
f1: skip C1,t2              b   D
f2: op3                 t1: skip C2,t5
    skip C3,t3          f5: op1
f3: op1                     op4
    op5                     b   E
    b   A               t5: op6
t3: op2                     op7
    b   B                   b   F
t2: op4
    skip C4,t4
f4: op5
    b   C
```

Figure 2: Sequential representation of tree-instruction in Figure 1

A binary test on a condition code is performed with a *skip* instruction, which corresponds to a flow-control operation within the tree, and which indicates where the tree-path corresponding to the true outcome of the test continues in storage; as a result, a skip instruction is a branch with a (short) positive displacement. All destination targets of the tree are represented as *unconditional branch* instructions, which specify the next tree to be executed when that path of the current tree is selected. Consequently, the end of a tree-instruction is delimited by *an instruction that follows an unconditional branch which is not reachable by any skip instruction within the tree.*

Note that any primitive instruction within a tree-instruction can also correspond to the starting point of another tree. As a result, branching into a tree-instruction leads to the execution of a tree which is a subset of a larger tree.

A ForestaPC processor fetches tree-instructions from main storage for execution. If the size of a tree-instruction exceeds the resources in the processor (such as number of branches, number of fixed-point or floating-point operations, and so on), then the tree-instruction is dynamically decomposed (*pruned*) to fit the resources available in the processor (see Figure 3). The resulting subtrees are executed in successive cycles, unless the taken path is completely contained within the first subtree.

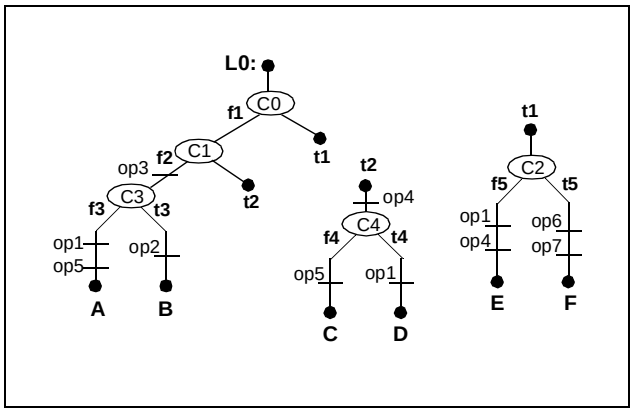


Figure 3: Pruning a tree-instruction

Programming Note: Better performance after pruning might be obtained by allocating the most frequently taken path to the left-most path in a tree-instruction.

The pruning process transforms arbitrary-size tree-instructions into subtree-instructions which fit the resources available in a processor implementation. These subtrees have the same general structure as the original trees (that is, a multiway-branch tree with operations in the tree-paths), but their size is limited. These subtrees correspond to *Very Long Instruction Words (VLIWs)* which are directly executed by the processor.

The primitive instructions are classified as follows:

- skip and branch instructions;
- storage access instructions;
- fixed-point instructions; and
- floating-point instructions.

Fixed-point instructions operate on byte, half-word, word, and double-word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. Storage access instructions provide byte, half-word, word, and double-word operand fetches and stores between storage and a set of 64 General Purpose Registers (GPRs). Storage access instructions also provide word and double-word operand fetches and stores between storage and a set of 64 Floating-Point Registers (FPRs).

Signed integers are represented in two's complement form.

No primitive instructions other than store instructions modify storage. To use a storage operand in a computation and then modify the same or another storage location, the contents of storage must be loaded into a register, modified, and then stored back to the target location.

Figure 4 is a logical representation of VLIW Native instruction processing. Tree instructions are fetched from storage and fed into a Pruning Unit, which converts them into VLIWs whose requirements fit the specific implementation. The output from the Pruning Unit is placed in a VLIW Register, which feeds a multiway Branch Processor and multiple Fixed-Point and Floating-Point Processors. The Branch Processor generates the storage address for the next tree-instruction, whereas the Fixed-Point and Floating-Point Processors perform the operations and interact with storage to transfer data.

Figure 5 shows the registers available in VLIW Native mode.

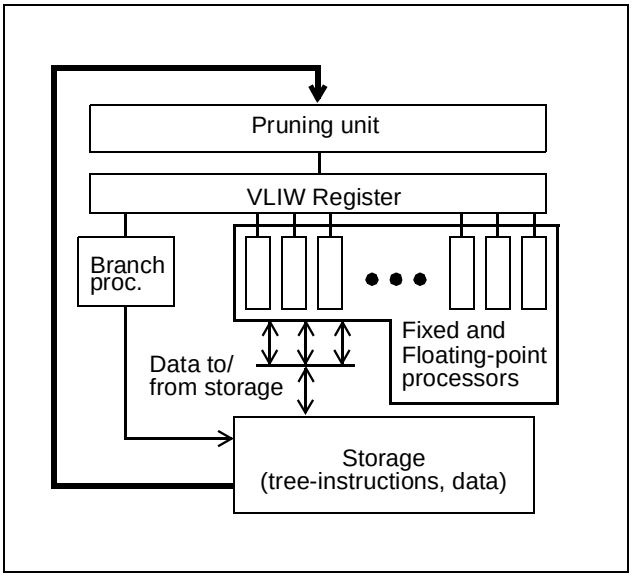


Figure 4: Logical processing model in *Native* mode

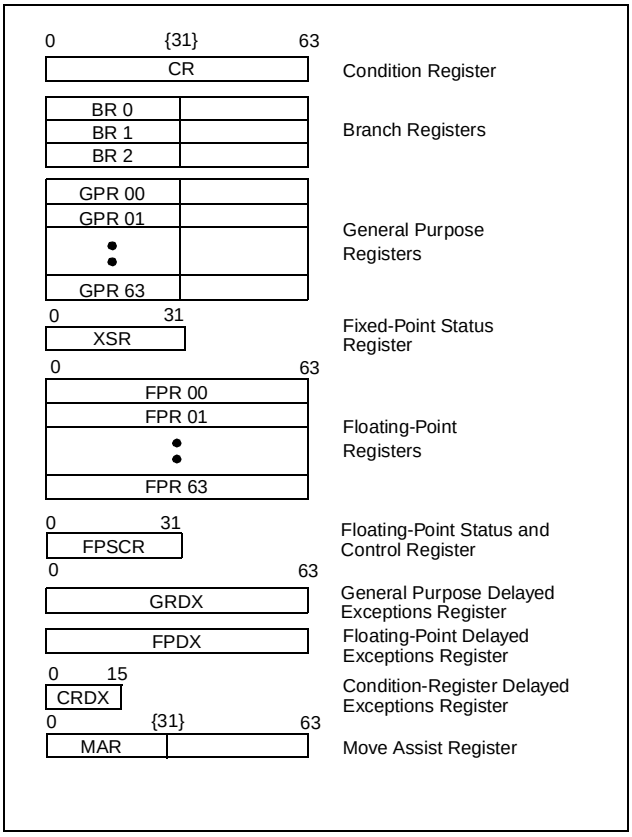


Figure 5: Native mode user register set

PowerPC Mode

Figure 6 is a logical representation of instruction processing in PowerPC mode. PowerPC primitive instructions are fetched from storage as blocks and fed into the Translation Unit, which converts them into groups of VLIW Native primitive instructions executable in parallel (based on dependencies among the PowerPC instructions and the resources in the processor). The resulting groups can be regarded as single-path tree-instructions. The output from the Translation Unit is placed in the VLIW Register, which feeds the resources in the processor. The Branch Processor generates the storage address for the next PowerPC instruction to be executed after the group, whereas the Fixed-Point and Floating-Point Processors perform the operations and interact with storage to transfer data.

Figure 7 shows the registers available in PowerPC mode (the same ones as in the PowerPC architecture).

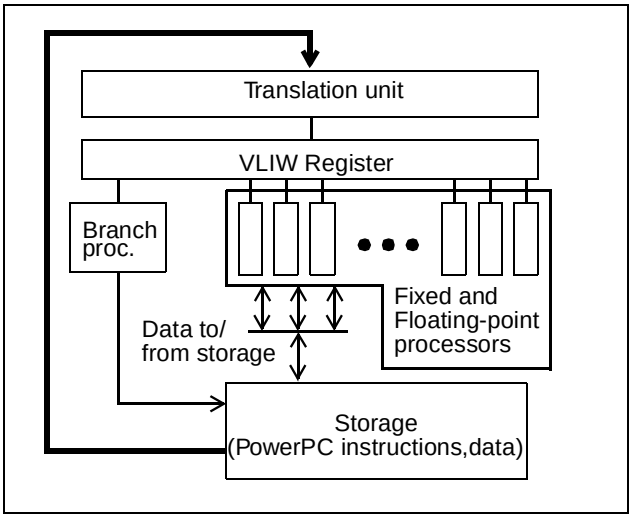


Figure 6: Logical processing model in PowerPC mode

The description of the primitive instructions in PowerPC mode is not given in this document; they are defined in *Book I, PowerPC User Instruction Set Architecture*.

1.1.2 Basic Processor Organization

The basic components of a ForestaPC processor are as follows (see Figure 8):

- a Pruning/Translation Unit
- a Very Long Instruction Word (VLIW) Register;
- a (multiported) General Purpose Register (GPR) file;

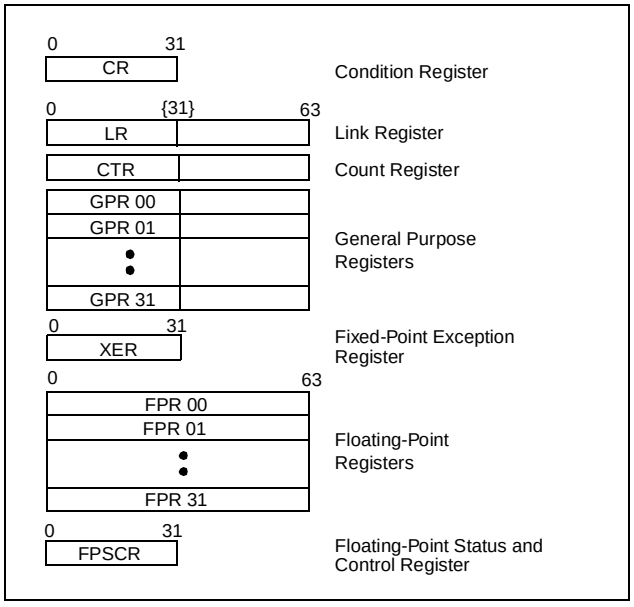


Figure 7: PowerPC mode user register set

- a (multiported) Floating-Point Register (FPR) file;
- a (multiported) set of Special Purpose Registers (SPRs).
- a Multiway-Branch Processor;
- an implementation-dependent number of Fixed-Point Processors;
- an implementation-dependent number of Floating-Point Processors;
- a Storage Subsystem;
- an Input/Output Subsystem.

The Very Long Instruction Word register (see Figure 11) is divided into *slots* or *parcels* of one word (32 bits each), which are numbered from left to right (starting from 0). A set of primitive instructions are allocated to the slots in the VLIW register.

Fixed-Point and Floating-Point processors are associated with slots within a VLIW, and are interconnected among themselves in nearest-neighbor fashion; for example, there is a path between the processor in slot *k* and the processor in slot *k-1*, between the processor in slot *k-1* and the processor in slot *k-2*, and so on. However, there is no path between the processor in slot 0 (the leftmost slot) and the processor in the rightmost slot. The existing paths are used by some special *Extender instructions* which allow creating multiparcel primitive instructions.

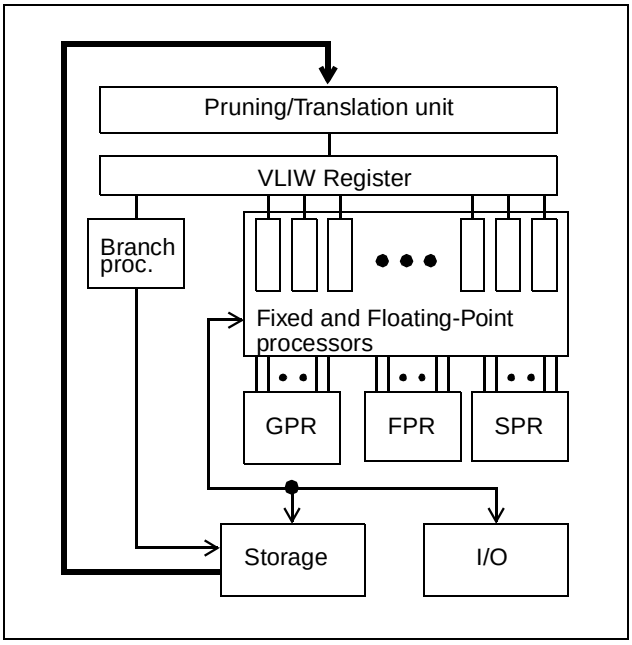


Figure 8: ForestaPC processor

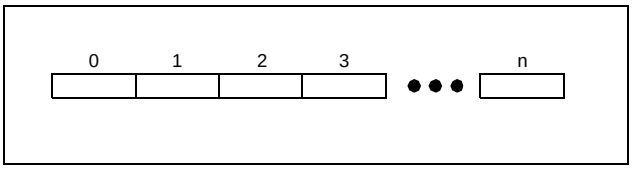


Figure 9: Format of Very-Long Instruction Word register

1.1.3 Semantics of a VLIW

Very Long Instruction Words specify a multiway-branch (Skip and Branch instructions) and a number of Fixed-Point, Storage Access, and Floating-Point instructions, all executable concurrently. The semantics of this group of instructions is as follows:

- Instructions that are on the taken path of the multiway-branch (as determined by the outcome from the conditions in the *skip* instructions) are executed to completion and their results placed in the corresponding target registers or storage locations.

  In contrast, instructions that are not on the taken path of the multiway-branch are inhibited from committing their results to storage or registers. Such instructions do not produce any effect on the state of the processor, nor are they observed by other processors.

- The following rules apply to instructions that are on the taken path of the multiway-branch:
  - All instructions are executed concurrently.
  - The results from all instructions are subject to sequential semantics. The results from an operation that uses a processor resource set by a previous operation in the path are undefined.
  - If two or more instructions target the same memory byte, register, field or bit of certain special registers, the value placed in that target corresponds to the instruction appearing later in the tree-instruction (in sequential storage order).

1.1.4 Speculative Execution

Speculative execution is a technique usable by the compiler (programmer) for improving performance in VLIW Native mode.

A *speculative operation* is one that has been placed above a branch with respect to a sequential execution stream, on the *speculation* that the result will be needed. If subsequent events indicate that the speculative instruction would not have been executed, or the results of the speculative instruction are not valid, any result produced by the instruction is not used. Typically, instructions are placed speculatively by the compiler/programmer when there are resources that would otherwise be idle so that the operation is done without cost, or when it might lead to reducing delays in the program.

Most fixed-point instructions (*Arithmetic, Logic, Load* instructions including *Floating-Point Loads*) can be executed speculatively. *Store* instructions should not be executed speculatively, nor should other instructions that produce unrecoverable effects.

An operand which has been loaded or computed speculatively, and any value derived from it, must be *committed* before it can be used non-speculatively (usually, at the original place in the sequential instruction stream). Special instructions are available to commit speculative operands.

No error of any kind other than Machine Check is reported due to the execution of a speculative instruction, until the result from its execution (or any other result derived from it) is committed. If there were errors, the instruction should be re-executed at that point, as well as any other instructions already executed that depend on the speculative operation.

Speculative execution is supported by the following resources and procedures:

- Each GPR, FPR and CR Field has an associated *Delayed Exception* bit, which is used to report (in delayed manner) if an exception occurred during execution of a speculative instruction which targets the corresponding register or field.
- Reading a register whose Delayed Exception bit is 1 either raises an exception or propagates the Delayed Exception bit to the target register of the operation, as follows:
  - if the operation is a *commit operation*, then a *delayed exception* is raised to the processor;
  - if the register is used to generate the address of a memory location accessed by a *store operation*, then an *invalid operation exception* is raised to the processor;
  - otherwise, the Delayed Exception bit associated with each target register of the operation is set to 1; the register contents become undefined.
- Placing in storage a register whose Delayed Exception bit may be set to 1 requires storing the Delayed Exception bit explicitly. Similarly, reading from storage a value which may have associated a Delayed Exception bit set to 1 requires reading the Delayed exception bit explicitly.
- *Speculative load operations* are identified as such through a *Speculative Flag* bit SF=1 in the instruction. No other speculative operations are explicitly identified as such.
- *Speculative load operations* that succeed (i.e., that do not raise an exception) are observed by other processors, as described in *Book II, ForestaPC Virtual Environment Architecture*. *Speculative load operations* that do not succeed set the Delayed Exception bit in the target register, and are not observed by other processors.
- An operand which has been generated speculatively is committed by executing a *Commit* instruction. The architecture includes *Commit* instructions for General-Purpose Registers, Floating-Point Registers, and Condition Register Fields. These instructions copy a speculative register contents into another register (of the same type), checking the *Delayed Exception* bit in the process. If the Delayed Exception bit is not set, the move register operation proceeds; otherwise, a *delayed exception* is generated.

- When a delayed exception is raised by a *Commit* instruction, the exception handler activates recovery code which re-executes the speculative instruction which generated the exception as well as those instructions that depend on it and which were executed before the exception was raised. For these purposes, the instructions executed between the speculative instruction generating the exception and the commit operation must not destroy the operands of the instructions that are re-executed in the recovery code.
- Speculation of other operations is managed by the compiler (programmer), without explicit indication.

The Delayed Exception bits of General Purpose Registers, Floating-Point Registers, and 4-bit Condition Register Fields are kept in special purpose registers GRDX, FPDX, and CRDX, respectively. GRDX contains the Delayed Exception bits of General Purpose Registers 0 to 63, in left to right order. FPDX contains the Delayed Exception bits of Floating-Point Registers 0 to 63, also in left to right order. CRDX contains the Delayed Exception bits of CR fields 0 to 15, also in left to right order.

1.1.5 Out-of-order Load Instructions

An *out-of-order Load* instruction is one that has been placed above a *Store* instruction with respect to sequential execution. *Load* instructions frequently start a sequence of dependent operations that depend on the datum loaded, so it is advantageous to initiate the loads as early as possible. However, an out-of-order *Load* may conflict with a *Store* operation over which it has been moved if the addresses of the *Load* and *Store* cannot be disambiguated by the compiler (programmer).

A software-based *coherence test* allows reordering load instructions relative to store instructions, in spite of the possibility of conflicts due to memory references which cannot be disambiguated. Whenever a *Load instruction* is moved earlier than a sequentially preceding ambiguous *Store instruction* by the compiler(programmer), a coherence test is inserted at the original position of the *Load instruction* in the sequential instruction stream. The coherence test consists of two instructions: a *Load* instruction from the same memory location, followed by a *Trap if not equal* instruction which compares the value just loaded with the value loaded out-of-order. If the values are identical, then the value loaded out-of-order and all other values derived from it are correct, and execution can proceed normally. On the other hand, if the value just loaded is different from the value loaded out-of-order (which implies that the corresponding memory location has been modified after been read), then the value loaded out-of-order as well as all other values derived from it are incorrect and must be recomputed.

As in the case of speculative instructions that raise exceptions, when a trap is generated by the *Trap if not equal* instruction that is part of the coherence test, the trap handler activates recovery code which re-executes the out-of-order load instruction as well as those instructions that depend on it and which were executed before the trap was generated. For these purposes, the instructions executed between the out-of-order load instruction and the coherence test must not destroy the operands of the instructions that are re-executed in the recovery code.

1.2 Compatibility with the PowerPC Architecture

In PowerPC mode, the ForestaPC architecture provides binary compatibility with the PowerPC Architecture; the User Instruction Set Architecture is the same.

In VLIW Native mode, the ForestaPC architecture does not provide binary compatibility for PowerPC programs. Instead, the ForestaPC architecture relies on object-code translation into ForestaPC code; some primitive instructions in the architecture are intended to facilitate object-code translation.

A summary of the incompatibilities among the PowerPC Architecture and the ForestaPC Architecture in VLIW Native mode is described in this section.

Many of the primitive instructions have the same functionality as PowerPC instructions, though they have different instruction format and opcode encoding. In most of these cases, the ForestaPC instruction name and mnemonics are the same as those in PowerPC. Due to the differences in architecture, some PowerPC instructions do not exist in the ForestaPC architecture; their functionality is achieved by several ForestaPC primitive instructions executed sequentially or in parallel. In addition, some new instructions have been incorporated.

The register set is larger than the one available in the PowerPC architecture; in particular, there are 64 General Purpose Registers, 64 Floating-Point Registers and 16 Condition Register fields, in addition to several new Special Purpose Registers. Some PowerPC Special Purpose Registers are not available, or are set differently.

Storage access instructions have a 11-bit signed displacement field; this is in contrast to the PowerPC architecture, wherein most storage access instructions have a 16-bit signed displacement. Moreover, there is a single address mode (register plus displacement); there are no *indexed mode* nor *update form* of storage access instructions, as in the PowerPC architecture.

Primitive instructions do not have the equivalent of the Rc bit available in the PowerPC architecture to set CR0 or CR1. In contrast, primitive instructions that set the Condition Register can directly set any of the sixteen Condition Register fields. Fixed-point instructions that do not have a CR field can be augmented with a special *Extender* instruction specifying a CR field.

XSR is the register corresponding to XER in the PowerPC architecture. However, XSR is set only by special *Move to Special-Purpose Register* instructions. All other fixed-point instructions do not set XSR directly. Instead, all other fixed-point instructions generate a value called *Fixed-Point Status Image (XSR-Image)*. Fixed-point instructions can be augmented with a special *Extender* instruction specifying a General Purpose Register; the *Extender* is used to place the XSR-Image generated by the instruction being augmented into the specified General Purpose Register. This approach is used to enhance instruction-level parallelism by allowing the simultaneous execution of multiple instructions that set fields of XSR (CA, OV).

FPSCR is set only by special *Move to FPSCR* instructions. All other floating-point instructions do not set FPSCR directly. Instead, all other floating-point instructions generate a value called *Floating-Point Status Image (FSR-Image)*. Floating-point instructions can be augmented with a special *Extender* instruction specifying a General Purpose Register; the *Extender* is used to place the FSR-Image generated by the instruction being augmented into the specified General Purpose Register. This approach is used to enhance instruction-level parallelism by allowing the simultaneous execution of multiple instructions that set fields of FPSCR.

Floating-point instructions can be augmented with a special *Extender* instruction which specifies the immediate generation of exceptions arising from the execution of the floating-point operations.

String, load/store multiple, and other complex PowerPC primitives (such as *rlwimi and rldimi)* have been excluded from the architecture; their functionality is implemented by a series of simpler primitive instructions.

1.3 Instruction Mnemonics and Operands

In PowerPC mode, each instruction has the same representation and features defined in *Book I, PowerPC User Instruction Set Architecture*. See that document for additional details; further information is not provided here.

For VLIW Native mode, the description of each primitive instruction includes the mnemonics and a formatted list of operands. Some examples are

- stw RS,D(RA)
- addi RT,RA,SI
- ldbz? RT,D(RA)

In most cases, the mnemonics are the same ones as in the PowerPC architecture. A load instruction mnemonic ending with the symbol “?” indicates that the instruction is *speculative*.

The description of every tree-instruction starts with a label, and includes the specification of *skips, branches*, and other primitive instructions. *Skip* and *branch* instructions have an associated label. An example of tree-instruction is depicted in Figure 10.

```
L0: skip        cr0.ne,t1
f1: skip        cr1.gt,t2
f2: add         r10,cr8,r14,r56
    skip        cr3.eq,t3
f3: subf        r12,cr9,r14,r44
    andi        r22,r16,0x34
    b           A
t3: or          r16,cr10,r16,r17
    b           B
t2: addi        r21,r16,0x1234
    skip        cr4.lt,t4
f4: andi        r22,r16,0x34
    b           C
t4: subf        r12,cr9,r14,r44
    b           D
t1: skip        cr2.eq,t5
f5: subf        r12,cr9,r14,r44
    addi        r21,r16,0x1234
    b           E
t5: lbz         r23,64(r2)
    stw         r24,32(r2)
    b           F
```

Figure 10: Example of a tree-instruction

ForestaPC-compliant assemblers will support the mnemonics and operand lists exactly as shown, and will also provide certain extended mnemonics. They may also provide high-level representations for multiway-branches, such as nested if-then-else and goto constructs.

1.4 Document Conventions

1.4.1 Definitions and Notation

The following definitions and notation are used throughout the ForestaPC Architecture documents.

- A VLIW Native program is a sequence of related tree-instructions.
- A PowerPC program is a sequence of related PowerPC instructions.
- A tree-instruction is a variable-length sequence of primitive instructions; each primitive instruction is one word long (32 bits per word).
- A tree-path is a path within a tree-instruction starting at the first operation in the tree and ending in an unconditional branch instruction.
- A tree-branch is a subtree starting at the target of a skip instruction.
- The binary tests in a tree-instruction comprise a multiway-branch which, at run time, selects one out of several tree-paths; the selected path is also called the taken path. Only those operations on the selected tree-path are actually executed.
- VLIW refers to a Very Long Instruction Word whose length is implementation-dependent. A VLIW corresponds to a tree-instruction not exceeding the resources of the implementation.
- Primitive instruction (or just instruction) refers to a 32-bit *native* or *primitive* instruction word.
- Slot or parcel refers to a 32-bit word within a VLIW, which contains a primitive instruction. Slots are numbered from left to right, starting from slot 0.
- Quadwords are 128 bits, doublewords are 64 bits, words are 32 bits, halfwords are 16 bits, and bytes are 8 bits.
- All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.
  - Underscores may be used between digits.
- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- RT, RA, RB, ... refer to General Purpose Registers.
- FRT, FRA, FRB, ... refer to Floating-Point Registers.
- BRT, BRS refer to Branch Registers.
- (x) means the contents of register x, wherein x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, wherein RA and FRA are instruction fields. Names such as BR0 and XSR denote registers, not fields, so parentheses are not used with them. In addition, when register x is assigned a value, parentheses are omitted.
- (RA|0) means the contents of register RA if the RA field has the value 1-63, or the value 0 if the RA field is 0.
- Bits in registers, instructions, storage and fields are specified as follows.
  - Bits are numbered left to right, starting with bit 0.
  - Ranges of bits are specified by two numbers separated by a colon (:). For example, the range 3:8 consists of bits 3 through 8.
  - For registers that are 64-bits long in 64-bit implementations and 32-bits long in 32-bit implementations, bit numbers and ranges are specified with the values for 32-bit implementations enclosed in braces ({}). {} means a bit that does not exist in 32-bit implementations. {:} means a range that does not exist in 32-bit implementations.
- X_p means bit p of register/field X.
  $X_{p\{r\}}$ means bit p of register/field X in a 64-bit implementation, and bit r of register/field X in a 32-bit implementation.
- $X_{p:q}$ means bits p through q of register/field X.
  $X_{p:q\{r:s\}}$ means bits p through q of register/field X in a 64-bit implementation, and bits r through s of register/field X in a 32-bit implementation.
- $X_{p\ q\ ...}$ means bits p, q, ... of register/field X.
  $X_{p\ q\ ...\ \{r\ s\ ...\}}$ means bits p, q, ... of register/field X in a 64-bit implementation, and bits r, s, ... of register/field X in a 32-bit implementation.
- $\neg$(RA) means the one's complement of the contents of register RA.
- 2^n means 2 raised to the n^{th} power.

- ^{n}x means the replication of x, n times (i.e., x concatenated to itself n-1 times). $^{n}0$ and $^{n}1$ are particular cases:
  - $^{n}0$ means a field of n bits with each bit equal to 0. Thus, $^{5}0$ is equivalent to 0b00000.
  - $^{n}1$ means a field of n bits with each bit equal to 1. Thus, $^{5}1$ is equivalent to 0b11111.
- Positive means greater than zero.
- Negative means less than zero.
- A speculative instruction is an instruction that has been moved above a sequentially preceding conditional branch.
- An out-of-order *Load* instruction is an instruction that has been moved above a sequentially preceding *Store* instruction.
- A *Load* instruction mnemonics followed by the symbol "?" indicates a speculative load instruction.
- A system library program is a component of the system software that can be invoked by an application program using a *Branch* instruction.
- A system service program is a component of the system software that can be invoked by an application program using a *System Call* instruction.
- The system trap handler is a component of the system software that receives control when the conditions specified in a *Trap* instruction are satisfied.
- The system error handler is a component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of errors. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.
- Each bit and field in instructions, in status and control registers (XSR and FPSCR), and in Special Purpose Registers, is either defined or reserved.
- /, //, ///, ... denotes a reserved field in an instruction or in an architected storage table.
- 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.
- Unavailable refers to a resource that cannot be used by the program. Data or instruction storage is unavailable if an instruction is denied access to it. See *Book III, ForestaPC Operating Environment Architecture*.
- The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary sequence of instructions, starting in the state the machine was in before executing the given instruction. Boundedly undefined results for a given instruction may vary among implementations, and among different executions on the same implementation, and are not further defined in this document.
- The sequential execution model in VLIW Native mode is the model of program execution described in Section 3.1, "Fetching Tree-Instructions," on page 31.

1.4.2 Reserved Fields

All reserved fields in primitive instructions should be zero. If they are not, the instruction form is invalid (see Section 1.9, "Invalid Instruction Forms," on page 17).

The handling of reserved bits in status and control registers, and in Special Purpose Registers, is implementation-dependent. For each such reserved bit, an implementation shall either:

- ignore the source value for the bit on write, and return zero for it on read; or
- set the bit from the source on write, and return the value last set for it on read.

Programming Note: It is the responsibility of software to preserve bits that are now reserved in status and control registers and in Special Purpose Registers, as they may be assigned a meaning in some future version of the architecture. In order to accomplish this preservation in implementation independent manner, software should do the following:

- Initialize each such register supplying zeroes for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

XSR and FPSCR are partial exceptions to this recommendation. Software can alter the status bits in these registers, preserving the reserved bits, by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined bit in the FPSCR by executing a *Floating-Point Status and Control Register*

instruction. Using such instructions is likely to yield better performance than using the method described in the second item above.

When a currently reserved bit is subsequently assigned a meaning, every effort will be made to have the value to which the system initializes the bit correspond to the "old behavior".

1.4.3 Description of Instruction Operation

The operation of all primitive instructions is described textually. In addition, the operation of most primitive instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation summarized below, in addition to the definitions and notation described in Section 1.4.1, "Definitions and Notation," on page 9. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instructions, except that *standard* setting of the Condition Register is not shown. (*Non-standard* setting of this registers, such as the setting of Condition Register Field 8 by the *stwc* instruction, is shown.) Fields of the XSR-Image or FSR-Image generated by an instruction are indicated. The RTL descriptions do not cover cases in which the system error handler is invoked, or for which the results are boundedly undefined.

The RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

The following elements are used in the RTL descriptions:

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_{iea}$	Assignment of an instruction effective address. In 32-bit mode of a 64-bit implementation, the high-order 32-bits of the 64-bit target are set to 0.
$\neg$	NOT logical operator
+	Two's complement addition
-	Two's complement subtraction, unary minus
$\times$	Multiplication
$\div$	Division (yielding quotient)
$\sqrt{}$	Square root
$=, \neq$	Equals and Not Equals relations
$<, \leq, >, \geq$	Signed comparison relations
$<^u, >^u$	Unsigned comparison relations
?	When used as a relation, unordered comparison relation; when used as a value, an implementation-dependent 0/1 (false/true) 1-bit value with implementation-dependent variability; when used in an instruction mnemonic, speculative operation
&, |	AND, OR logical operators
$\oplus, \equiv$	Exclusive-OR, Equivalence logical operators ($(a \equiv b) = (a \oplus \neg b)$)
ABS(x)	Absolute value of x
BR0, BR1, BR2	Branch Registers
CEIL(x)	Least integer $\geq$ x
CRB	Condition Register viewed as 64 independently-addressable bits
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model given in page 42
EXTS(s)	Result of extending x on the left with sign bits
FLOOR(x)	Greatest integer $\leq$ x
FPR(x)	Floating-Point Register x
GPR(x)	General Purpose Register x
MASK(x,y)	Mask having 1's in positions x through y (wrapping if x > y) and 0's elsewhere
MEM(x,y)	Contents of y bytes of memory starting at address x. In 32-bit mode of a 64-bit implementation, the high-order 32-bits of the 64-bit value are ignored.
$ROTL_{64}(x,y)$	Result of rotating the 64-bit value x left by y positions.
$ROTL_{32}(x,y)$	Result of rotating the 64-bit value x||x left by y positions, where x is 32 bits long

Notation	Meaning
SINGLE(X)	Result of converting x from floating-point double format to floating-point single format, using the model shown on page 42
SPREG(x)	Special Purpose Register x
TRAP	Invoke the system trap handler
characterization	Reference to the setting of status bits, in a standard way that is explained in the text
undefined	An undefined value. The value may vary among implementations, and among different executions on the same implementation
CIA	Current Instruction Address, which is the 64{32}-bit address of the primitive instruction being described by a sequence of RTL. In 32-bit mode of a 64-bit implementation, the high-order 32-bits of CIA are always set to 0. Does not correspond to any architected register.
NIA	Next Instruction Address, which is the 64{32}-bit address of the next primitive instruction to be executed. In 32-bit mode of a 64-bit implementation, the high-order 32-bits of NIA are always set to 0. Does not correspond to any architected register.
XSR-Image	XSR-Image generated by an instruction. The bit-definitions of this image are the same as those in the XSR register, excepting the Summary Overflow bit which is not defined. Does not correspond to any architected register.
FSR-Image	FSR-Image generated by a floating-point instruction. The bit-definitions of this image are the same as the status bits in the FPSCR register, excepting the Summary bits which are not defined. Does not correspond to any architected register.
if ... then ... else	Conditional execution, indenting shows range, else is optional
do	Do loop, indenting shows range. “To” and/or “by” clauses specify incrementing an iteration variable, and a “while” clause gives termination conditions
leave	Leave innermost do loop, or do loop described in leave statement

1.4.3.1 Precedence Rules

The precedence rules for RTL operators are summarized in Table 1. Operators at higher rows in the table are applied before those at lower rows. Operators at the same row in the table associate from left to right, from right to left, or not at all, as indicated in each case. For example, - associates from left to right, so a-b-c=(a-b)-c. Parentheses are used to override the evaluation order implied by the table, or to increase clarity; parenthesized expressions are evaluated before serving as operands.

TABLE 1. Operator Precedence

Operators	Associativity
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	right to left
unary -, ¬	right to left
×, ÷	left to right
+, -	left to right
||	left to right
=, ≠, <, ≤, >, ≥, $<^u$, $>^u$	left to right
&, ⊕, ≡	left to right
|	left to right
: (range)	none
← (assignment)	none

1.5 Format of Tree-Instructions

All tree-instructions are aligned on a word (4-byte) boundary. Whenever tree-instruction addresses are presented to the processor, the two least-significant bits are ignored. Similarly, whenever the processor produces a tree-instruction address, the two least-significant bits are zero.

The format of a tree-instruction consists of a sequence of contiguous words (four bytes), as illustrated in Figure 11.

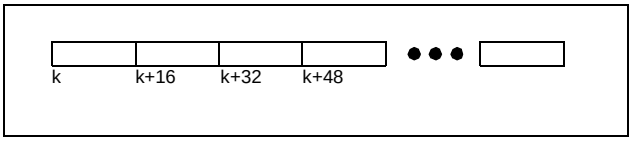


Figure 11: Format of a tree-instruction

1.6 Formats of Primitive Instructions

All primitive instructions are one word (four bytes) long and word aligned.

Bits 0:3 of a primitive instruction always specify the primitive opcode (OP). Most primitive instructions also have an extended opcode (XO). The remaining bits of the primitive instruction contain one or more fields, as shown below for the different instruction formats. In all cases, the value of field OP determines the length of field XO.

Editor's Note: The assignment of opcodes to instructions (enumeration of the instructions assigned to each primary and extended opcode) is tentative. The assignment might be revised in a future version of the architecture.

The format diagrams given below show horizontally all valid combinations of instruction fields. The diagrams include instruction fields that are used only by instructions defined in *Book II, ForestaPC Virtual Environment Architecture*, or *Book III, ForestaPC Operating Environment Architecture*. See those Books for the definition of such fields. The name of a format ends with a number which specifies the length of the extended opcode field.

In some cases an instruction field is reserved, or must contain a particular value. If a reserved field does not have all bits set to 0, or if a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described in Section 1.9, "Invalid Instruction Forms," on page 17.

Split Field Notation.

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits which are used in permuted order. Such a field is called a *split field*. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in lowercase characters, once for each of the contiguous sequences, followed by an identification digit. In the RTL description of an instruction having a split field, the name of the split field in uppercase characters represents the concatenation of the sequences from left to right in increasing order of identification digit. In all other cases, and in certain places where individual bits of a split field are identified, the name of the field in lowercase characters represents the concatenation of the sequences in some order, which need not be left to right, as described for each relevant instruction

1.6.1 I0-Form

0	4	10	16
OP	RT	RA	SI
OP	RT	RA	UI

1.6.2 M0-Form

0	4	10	16	22	27
OP	RT	RA	RB	MB	ME

1.6.3 M1-Form

0	4	10	16	17	22	27	31
OP	RT	RA	me_1	SH	MB	me_0	XO

1.6.4 I1-Form

0	4	8	9	10	16	31
OP	CRT	si_1	L	RA	si_0	XO
OP	CRT	ui_1	L	RA	ui_0	XO

1.6.5 B2-Form

0	4	6	30
OP	ADDR		XO
OP	BRT	ADDR	XO

1.6.6 D4-Form

0	4	10	16	27	28
OP	RT	RA	D	SF	XO
OP	RT	dl_1	dl_0	SF	XO
OP	FRT	RA	D	SF	XO

1.6.7 X4-Form

0	4	10	16	22	28
OP	FRT	FRA	FRB	FRC	XO
OP	RT	RA	RB	MB	XO
OP	RT	RA	RB	ME	XO
OP	RT	RA	SH	MB	XO
OP	RT	RA	SH	ME	XO
OP	RT	IA	IB	CB	XO
OP	RT	IA	RB	CB	XO
OP	RT	RA	IB	CB	XO
OP	RT	RA	RB	CB	XO

1.6.8 D5-Form

0	4	10	16	22	27
OP	d_0	RA	RB	d_1	XO
OP	d_0	RA	FRB	d_1	XO
OP	RT	RA	D		XO

1.6.9 X6-Form

0	4	10	16	22	26
OP	RT	RA	RB	CRT	XO
OP	RT	RA	SH	CRT	XO
OP	RT	RA	RB	/ SH	XO

1.6.10 D8-Form

0	4	7	10	16	24
OP	d_1	SCL	RA	d_0	XO
OP	d_1	//	RA	d_0	XO

1.6.11 I8-Form

0	4	8	16	24
OP	CRT	si_1	si_0	XO

1.6.12 X8-Form

0	4	10	16	20	24
OP	RT	RA	CRT	CRS	XO

1.6.13 B10-Form

0	4	8	11	22
OP	CRS	BC	ADDR	XO
OP	ADDR			XO

1.6.14 I10-Form

0	4	6	16	22
OP	//	si_1	si_0	XO

1.6.15 D10-Form

0	4	10	16		22
OP	d_1	RA	d_0	/	XO

1.6.16 X10-Form

0	4	6	8	10	12	14	16	18	20	22
OP	BT			BA			BB			XO
OP	RT			RA			RB			XO
OP	FRT			FRA			FRB			XO
OP	CRT		//	FRA			FRB			XO
OP	CRT		/ L	RA			RB			XO
OP	RT			RA			CRT		//	XO
OP	TO		/	RA			RB			XO
OP	FRT			FRA			///			XO
OP	FRT			RA			///			XO
OP	RT			FRA			///			XO
OP	RT			RA			//		XM	XO
OP	RT			///			FM			XO
OP	RT			RA			CRT		XM	XO
OP	RT			///			//		XM	XO
OP	//	spt_1		RA			spt_0			XO
OP	RT			//	sps_1		sps_0			XO
OP	///			///			///			XO
OP	CRT		//	RA			FM			XO
OP	CRT		//	//	BFI		BFT		//	XO
OP	RT			///			CRT		//	XO
OP	/	FBT		///			CRT		//	XO
OP	RT			RA			///			XO
OP	TO		/	RA			///			XO
OP	RT			///			CRS		//	XO
OP	CRT		//	RA			///			XO
OP	CRT		//	///			CRS		//	XO
OP	CRT		//	///			CRI		//	XO
OP	CRT		//	///			BFS		//	XO
OP	RT			///		L	///			XO
OP	///			///		L	RB			XO
OP	RT			///			///			XO
OP	///			RA			///			XO
OP	BRT		//	BRS		//	///			XO
OP	CRT		//	///			///			XO

1.7 Instruction Fields

ADDR(4:29 or 6:29)
Field used to specify the target address of a *Branch* instruction.

ADDR(4:21)
Field used to specify the block address of an *Instruction Cache* instruction.

ADDR(11:21)
Field used to specify the target address of a *Skip* instruction.

BA(10:15) and BB(16:21)
Field used to specify a bit in CR to be used as a source.

BC(8:10)
Field used to specify the condition tested in a *Skip* instruction.

BFI(12:15)
Field used to specify a 4-bit constant in a *Move to FPSCR* instruction.

BFS(16:18) and BFT(16,18)
Fields used to specify, respectively, a source and destination field in FPSCR.

BRS(10:11)
Field used to specify a branch register to be used as the source of an operation.

BRT(4:5)
Field used to specify a branch register to be used as the target of an operation.

BS(24:26)
Field used to specify an 8-bit (byte) portion of a register to be used as the target of byte immediate operations.

BT(4:9)
Field used to specify a bit in CR to be used as the target of the result of an instruction.

CB(22:27)
Field used to specify a bit in CR to be used as a source in the *Select* instructions.

CRI(16:19)
Field used to specify a 4-bit constant in a *Move to CR* instruction.

CRS(4:7, 16:19 or 20:23)
Field used to specify a field in CR used as a source operand.

CRT(4:7, 16:19, or 22:25)
Field used to specify a field in CR used as a target.

D(16:20||4:9, 16:23||4:6, 4:9||22:26 or 16:26)
Immediate field specifying an 11-bit unsigned integer which is used as the displacement for *storage access* instructions.

DL(16:26||10:15)
Immediate field specifying an 17-bit unsigned integer which is used as the displacement for *Load Table of Contents* instructions.

FBT(5:9)
Field used to specify a bit in FPSCR to be used as the target of the result of an instruction.

FM(16:21)
Field mask used to specify fields of FPSCR.

FRA(10:15), FRB(16:21) and FRC(22:27)
Fields used to specify a FPR as a source of an operation.

FRT(4:9)
Field used to specify a FPR as a target of an operation.

IA(10:15) and IB(16:21)
6-bit signed immediate value used in the *Select* instructions.

L(9 or 15)
Field used to specify whether certain instructions use 64-bit or 32-bit numbers, and to specify whether certain instructions use the most- or least-significant 32-bits of a register.

MB(22:26, or 22:27)
Field used to specify the first 1-bit of a 64-bit mask.

ME(27:31, 27:30||16, or 22:27)
Field used to specify the last 1-bit of a 64-bit mask.

OP(0:3)
Primary opcode field

RA(10:15) and RB(16:21)
Fields used to specify a GPR as a source of an operation.

RT(4:9)
Field used to specify a GPR as a target of an operation.

SBI(16:23)
Immediate field used to specify an 8-bit signed integer.

SCL(7:9)
Field used to specify the level of storage for *Touch* instructions.

SF(27)
Single-bit field used to specify a speculative *Load* operation.

SH(16:21, 17:21, or 23:25)
Field used to specify a shift amount.

SI(16:31, 16:30||8, 16:23||8:15, or 16:21||6:15)
Immediate field used to specify a 16-bit signed integer.

SPS(16:21||12:15)
Field used to specify a Special Purpose Register as a source of an operation.

SPT(16:21||6:9)
Field used to specify a Special Purpose Register as a target of an operation.

TO(4:8)
Field used to specify conditions on which to trap.

UBI(16:23)
Immediate field used to specify an 8-bit unsigned integer.

UI(16:31 or 16:30||8)
Immediate field used to specify a 16-bit unsigned integer.

XO(22:31, 24:31, 26:31, 27:31, 28:31, 30:31 or 31)
Extended opcode field.

XM(20:21)
Field mask used to specify fields of XSR.

1.8 Classes of Instructions

Any primitive instruction falls into exactly one of the following three classes:

- Defined
- Reserved
- Illegal

The class is determined by examining the opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction nor of a reserved instruction, the instruction is illegal.

Some instructions are defined only for 64-bit implementations and a few are defined only for 32-bit implementations (see Section 1.8.2, “Illegal Instruction Class,” on page 17). With the exception of these, a given instruction is in the same class for all implementations of the ForestaPC Architecture. In future versions of this architecture, instructions that are now illegal may become defined (by being added to the architecture) or reserved. Similarly, instructions that are now reserved may become defined.

1.8.1 Defined Instruction Class

This class of instructions contains all the instructions defined in the *ForestaPC User Instruction Set Architecture*, *ForestaPC Virtual Environment Architecture*, and *ForestaPC Operating Environment Architecture*.

Defined instructions are guaranteed to be supported in all implementations, except as stated in the instruction descriptions. (The exceptions are instructions that are supported only in 64-bit implementations or only in 32-bit implementations.)

A defined instruction can have invalid forms, as described in Section 1.9, “Invalid Instruction Forms,” on page 17.

1.8.2 Illegal Instruction Class

For 64-bit implementations, this class includes all instructions that are defined only for 32-bit implementations. For 32-bit implementations, it includes all instructions that are defined only for 64-bit implementations.

Excluding instructions that are defined for one type of implementation but not the other, illegal instructions are available for future extensions of the ForestaPC architecture; that is, some future version of the ForestaPC architecture may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0's is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or non-initialized storage will result in the invocation of the system illegal instruction error handler.

1.8.3 Reserved Instruction Class

Reserved instructions are allocated to specific purposes that are outside the scope of the ForestaPC architecture.

Any attempt to execute a reserved instruction will

- perform the actions described in *Book IV, ForestaPC Implementation Features* for the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.

1.9 Invalid Instruction Forms

Some of the defined instructions have invalid forms. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by just examining its instruction encoding.

Any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked, or will yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some invalid forms can be deduced from the primitive instruction layout. In particular:

- Field shown as / but coded as non-zero.

These invalid forms are not discussed further.

Instructions having invalid forms that cannot be so deduced are listed below. These kinds of invalid forms are identified in the instruction descriptions.

- *Move To/From Special Purpose Register* instructions

- *Extender* instructions which are placed in the right-adjacent slot to an instruction that cannot be extended.

Assembler Note: To the extent possible, the Assembler should report uses of invalid instruction forms as errors.

Engineering Note: Causing the system illegal instruction error handler to be invoked if attempt is made to execute an invalid form of an instruction facilitates the debugging of software

1.10 Optional Instructions

Some of the defined instructions are optional. The optional instructions are defined in the section entitled “Look-aside Buffer Management Instructions (Optional)” and the appendices entitled “Optional Facilities and Instructions” in Book II and Book III.

Any attempt to execute an optional instruction that is not provided by the implementation will cause the system illegal instruction error handler to be invoked. Exceptions to this rule are stated in the instruction descriptions.

1.11 Exceptions

There are two kinds of exception: those caused directly by the execution of an instruction, and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a *privileged* instruction (see *Book III, ForestaPC Operating Environment Architecture*) (system illegal instruction error handler or system privileged instruction error handler);
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler);
- the execution of an optional instruction that is not provided by the implementation (system illegal instruction error handler);
- an attempt to access a storage location that is unavailable (system error handler);
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler);
- an attempt to access storage with an effective address computed using a register whose Delayed Exception bit is set to 1 (system illegal instruction error handler);
- the execution of a *System Call* instruction (system service program);
- the execution of a *Trap* instruction that traps (system trap handler);
- the execution of a floating-point instruction when floating-point instructions are unavailable (system floating-point unavailable error handler);
- the execution of a floating-point instruction that requires system software assistance (system floating-point assist error handler; the conditions under which such software assistance is required are implementation-dependent);
- the execution of a *commit* instruction using a register whose *delayed exception bit* is set to 1 (system delayed exception handler);

The exceptions that can be caused by an asynchronous event are described in *Book III, ForestaPC Operating Environment Architecture*.

The invocation of the system error handler is precise, except when one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect, in which case the invocation of the system floating-point enabled exception error handler may be imprecise. When the invocation is precise, all VLIWs prior to the invocation of the system error handler have completed, all operations in the taken path of the tree prior to the one invoking the handler have completed, the operation invoking the handler and all operations that follow it in the taken path have not been executed, and no VLIWs subsequent to the invocation have been executed. When the system error handler is invoked imprecisely, the excepting VLIW does not appear to complete before the next VLIW starts (because one of the effects of the excepting VLIW, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in *Book III, ForestaPC Operating Environment Architecture*.

1.12 Delayed Exceptions

If a speculative operation causes an exception, the exception must not be raised until the result of that operation, or any value derived from it, is used in a *commit* instruction.

The architecture defines the following mechanisms for handling exceptions arising from speculative instructions:

- *Load* instructions which may produce an exception while executed speculatively have a special bit to indicate it, called the *Speculative Flag*.
- When the Speculative Flag is disabled (SF=0), the corresponding *Load* instruction is non-speculative; consequently, an exception occurring during execution should be raised to the processor and handled normally.
- When the Speculative Flag is enabled (SF=1), the corresponding *Load* instruction is a speculative operation. If the operation incurs an exception, then the Delayed Exception bit associated with the target register is set to 1, but the exception is not raised to the processor.

Reading a register whose Delayed Exception bit is 1 either raises an exception or propagates the Delayed Exception bit to the target register of the operation, as follows:

- if the operation is a *commit operation*, then a *delayed exception* is raised to the processor;
- if the register is used to generate the address of a memory location accessed by a *store operation*, then an *invalid operation exception* is raised to the processor;
- otherwise, the Delayed Exception bit associated with each destination register of the operation is set to 1; the register contents become undefined.

Placing in storage a register whose Delayed Exception bit may be set to 1 requires storing the Delayed Exception bit explicitly. Similarly, reading from storage a value which may have associated a Delayed Exception bit set to 1 requires reading the Delayed exception bit explicitly.

1.13 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Storage Access* instruction (or certain other instructions described in *Book II, ForestaPC Virtual Environment Architecture*, and *Book III, ForestaPC Operating Environment Architecture*) or when it fetches the next instruction (tree-instruction in VLIW Native mode, PowerPC instruction in PowerPC mode).

1.13.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Storage operands may be bytes, halfwords, words, or doublewords. The address of a storage operand is the address of its first byte (i.e., of its lowest numbered byte). Byte ordering is Big-Endian by default, but the processor can be operated in a mode in which byte ordering is Little-Endian.

Operand length is implicit for each instruction.

The operand of a *Storage Access* instruction has a *natural* alignment boundary equal to the operand length. In other words, the *natural address* of an operand is an integral multiple of the operand length. A storage operand is said to be *aligned* if it is aligned at its natural boundary: otherwise it is said to be *unaligned*.

Storage operands have the following characteristics. (Although not permitted as storage operands, quadwords are shown because quadword alignment is desirable for certain storage operands).

Operand	**Length**	**Addr$_{60:63}$ if aligned**
Byte	8 bits	xxxx
Half-word	2 bytes	xxx0
Word	4 bytes	xx00
Double-word	8 bytes	x000
Quad-word	16 bytes	0000
Note: An “x” in an address bit position indicates that the bit can be 0 or 1, independent of the state of other bits in the address.		

The concept of alignment is also applied more generally, to any datum in storage. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. The best performance is obtained when storage operands are aligned. Additional effects of data placement

on performance are described in *Book II, ForestaPC Virtual Environment Architecture*.

Tree-instructions have varying length and are word-aligned. Primitive instructions are always four bytes long and word-aligned.

1.13.2 Effective Address Calculation

The 64- or 32-bit address computed by the processor when executing a *Storage Access* instruction (or certain other instructions described in *Book II, ForestaPC Virtual Environment Architecture*, and *Book III, ForestaPC Operating Environment Architecture*) or when fetching the next tree-instruction, is called the *effective address*, and specifies a byte in storage. For a *Storage Access* instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the storage operand is considered to wrap around from the maximum effective address to effective address 0, as described below.

Effective address computations, for both data and instruction accesses, use 64{32}-bit unsigned binary arithmetic. A carry from bit 0 is ignored. In a 64-bit implementation, the 64-bit current instruction address and next instruction address are not affected by a change from 32-bit mode to 64-bit mode, but they are affected by a change from 64-bit mode to 32-bit mode (the high-order 32 bits are set to 0).

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, 2^{64}-1, to address 0.

In 32-bit mode, the low order 32 bits of the 64-bit result comprise the effective address for the purpose of addressing storage. The high-order 32 bits of the 64-bit effective address are ignored for the purpose of accessing data. The high-order 32 bits of the 64-bit effective address are set to 0 for the purpose of fetching instructions, and whenever a 64-bit effective address is placed in a Branch Register. The high-order 32 bits of the 64-bits effective address are set to 0 in Special-Purpose Registers when the system error handler is invoked. As used to address storage, the effective address arithmetic appears to wrap around from the maximum address, 2^{32}-1, to address 0.

A zero in the RA field indicates the absence of the corresponding address component. For the absent component, a value of zero is used for the address. This is shown in the instruction descriptions as (RA|0).

In both 64-bit and 32-bit modes, the calculated Effective Address may be modified in its three low-order bits before accessing storage if the system is operating in Little-Endian mode.

Effective addresses are computed as follows. In the descriptions below, it should be understood that “the contents of a GPR” refers to the entire 64-bits, independent of mode, but that in 32-bit mode, only bits 32:63 of the 64-bit result of the computation are used to address storage.

- With D-form instructions (D4, D5, D8, D10), the displacement field is zero-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With the *Branch* instruction (B2-form), the 26-bit ADDR field is concatenated in the right with 0b00; the resulting 28-bit value is concatenated to the right of $CIA_{0:35}$.
- With B10-form instructions, the 11-bit ADDR field is concatenated in the right with 0b00; the resulting 13-bit value is concatenated to the right of $CIA_{0:50}$.
- With the *Branch Register* instruction (X10-form), bits 0:61 of Branch Register 0 are concatenated to the right with 0b00 to form the effective address of the next tree-instruction.

For instructions that refer to more than one byte of storage, the effective address for each byte after the first is computed by adding 1 to the effective address of the preceding byte.

Chapter 2. Registers in the ForestaPC Architecture

This chapter describes the registers that exist in the ForestaPC Architecture. Section 2.1 describes the General Purpose Registers, Section 2.2 describes the Floating-Point Registers, and Section 2.3 describes the Special Purpose Registers.

The VLIW Native mode and the PowerPC mode define different sets of registers. Some registers exist in both modes, whereas some registers exist only in VLIW Native mode.

2.1 General Purpose Registers

The principal storage accessed by the Fixed-Point instructions is a set of 64 General Purpose Registers (GPRs), each with 64{32} bits of data (see Figure 12).

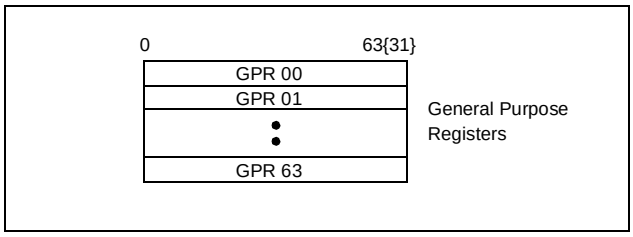


Figure 12: General Purpose Registers

In VLIW Native mode, all 64 GPRs are defined. In PowerPC mode, only General Purpose Registers 0 through 31 are defined.

2.2 Floating-Point Registers

The principal storage accessed by the Floating-Point instructions is a set of 64 Floating-Point Registers (FPRs). Each FPR contains 64 bits which support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

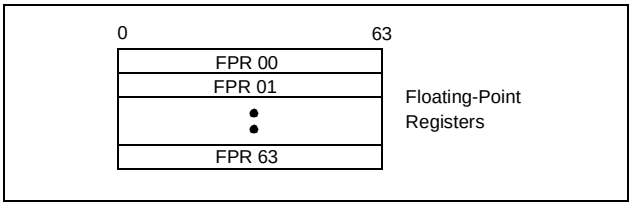


Figure 13: Floating-Point Registers

In VLIW Native mode, all 64 FPRs are defined. In PowerPC mode, only Floating-Point Registers 0 through 31 are defined.

2.3 Special Purpose Registers

The ForestaPC architecture has many Special Purpose Registers (SPRs). Some of these register exist only in VLIW Native mode, whereas others exist in both VLIW Native and PowerPC mode*s* but may be used differently.

2.3.1 Branch Registers

The Branch Registers (BRs) are three 64{32}-bit registers. In VLIW Native mode, the Branch Registers are used by the Branch instructions as follows:

- to hold the return address for procedure calls and *System Call* instructions; and
- to hold branch target addresses for *Branch Registers* instructions.

In addition to this dedicated use, Branch Registers can be used as source and destination of Fixed-Point instructions which manipulate branch addresses.

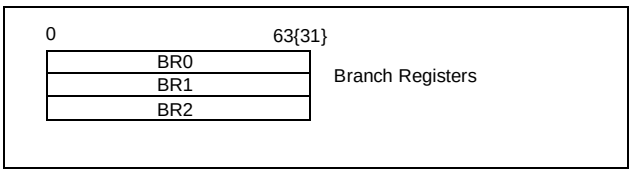


Figure 14: Branch Registers

In PowerPC mode, Branch Registers 1 and 2 do not exist, whereas Branch Register 0 is known as the Link Register (LR).

2.3.2 Count Register

The Count Register (CTR) is a 64{32}-bit register.

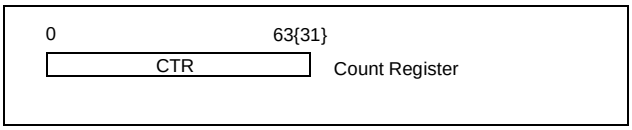


Figure 15: Count Register

The Count Register exists only in PowerPC mode; it does not exist in VLIW Native mode.

2.3.3 Condition Register

The Condition Register (CR) is a 64-bit register which reflects the results of certain operations and provides a mechanism for testing (and branching).

The bits in the Condition Register are grouped into 4-bit fields, named CR field 0 (CR0), CR field 1 (CR1), and so on. Sixteen CR fields (CR0 through CR15) are defined in VLIW Native mode, whereas eight CR fields are defined in PowerPC mode.

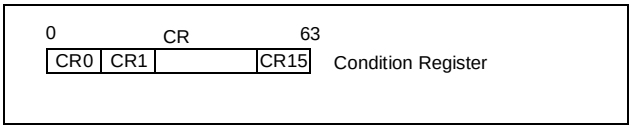


Figure 16: Condition Register

Architecture Note: In PowerPC mode, CR fields are named CR0 through CR7 but they physically correspond to CR8 through CR15, respectively.

The rest of this section describes the setting of CR in VLIW Native mode. See *Book I, PowerPC User Instruction Set Architecture* for the definition of the setting of CR in PowerPC mode.

CR fields are set in one of the following ways:

- Several specified fields of CR can be set by a move to the CR from a GPR (*mtcr*).
- A specified field of CR can be set by a move to the CR from:
  - another CR field (*mcrf*);
  - an immediate field (*mcrfi*);
  - a GPR (*mtcrf*);
  - a field from the FPSCR (*mcrfs*).
- A specified field of CR can be set by fixed-point instructions that have a CR destination field (CRT), or by fixed-point instructions that have been extended with an *Extend Immediate and Condition Register* (*xicr)* instruction.
- A specified field of CR can be set as the result of either a fixed-point or a floating-point *Compare* instruction.
- CR field 8 is set by a *Store Conditional* instruction.

Instructions are also provided to perform logical operations on individual CR bits, and to test individual CR bits. In the description of these instructions, the Condition Register is referred to as CRB, which denotes the Condition Register viewed as 64 single bits, rather than as 16 4-bit fields.

For all fixed-point instructions which have a CR destination field, or which have been extended with an *xicr* instruction, the first three bits of the specified CR field CRT are set by signed comparison of the result to zero, and the fourth bit of CR field CRT is copied from the OV field of the XSR-Image generated by the instruction (set to 0 if no image is generated). “Result” here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode.

```
if (64-bit implementation) & (64-bit mode)
   then M ← 0
   else M ← 32
if (target_register)M:63 < 0
               then          c ← 0b100
else if (target_register)M:63 > 0
               then          c ← 0b010
else                         c ← 0b001
CRCRT ← c || XSR-ImageOV
```

If any portion of the result is undefined, the value placed into the CR field CRT is undefined.

The bits of CR field CRT are interpreted as follows:

Bit	Description
0	**Negative (LT)** The result is negative.
1	**Positive (GT)** The result is positive
2	**Zero (EQ)** The result is zero.
3	**Overflow (OV)** This is a copy of bit XSR-Image$_{OV}$.

Programming Note: CR field CRT may not reflect the "true" (infinitely precise) result if overflow occurs; see Section 5.8, "Fixed-Point Arithmetic Instructions," on page 72.

For *Compare* instructions, a specified CR field is set to reflect the result of the comparison. The bits of the specified field are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 5.10, "Fixed-Point Compare Instructions," on page 83, and Section 6.6.4, "Floating-Point Compare Instructions," on page 143.

Bit	Description
0	**Less Than, Floating-Point Less Than (LT, FL)** For fixed-point *Compare* instructions, (RA) < SI or (RB) (signed comparison), or (RA) $<^{u}$ UI or (RB) (unsigned comparison). For floating-point *Compare* instructions, (FRA) < (FRB).
1	**Greater Than, Floating-Point Greater Than (GT, FG)** For fixed-point *Compare* instructions, (RA) > SI or (RB) (signed comparison), or (RA) $>^{u}$ SI or (RB) (unsigned comparison). For floating-point *Compare* instructions, (FRA) > (FRB).
2	**Equal, Floating-Point Equal (EQ, FE)** For fixed-point *Compare* instructions, (RA) = SI, UI, or (RB). For floating-point *Compare* instructions, (FRA) = (FRB).
3	**Zero, Floating-Point Unordered (ZE, FU)** For fixed-point *Compare* instructions, this bit is set to 0. For floating-point *Compare* instructions, one or both of (FRA) and (FRB) is a NaN.

2.3.4 Fixed-Point Status Register

The Fixed-Point Status Register (XSR) is a 32-bit register. This register is defined both in VLIW Native mode as well as in PowerPC mode. In PowerPC mode, this register is called Fixed-Point Exception Register (XER).

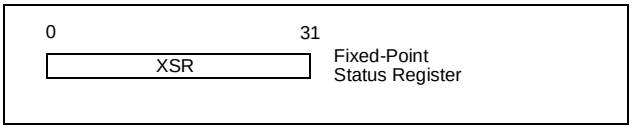


Figure 17: Fixed-Point Status Register

The rest of this section describes the setting of XSR in VLIW Native mode. See *Book I, PowerPC User Instruction Set Architecture* for the definition of the setting of XER in PowerPC mode.

The Fixed-Point Status Register is set only by instructions *Update XSR (uxsr)* and *Move to Special-Purpose Register (mtspr).*

The bit definitions for the Fixed-Point Status Register are as shown next.

Bit(s)	Description
0	**Summary Overflow** (SO) The Summary Overflow bit is set to 1 whenever an *Update XSR (uxsr)* instruction sets the Overflow bit. Once set, SO remains set until it is cleared by a *mtspr* instruction (specifying XSR). Executing a *mtspr* instruction to XSR, supplying the values 0 for SO and 1 for OV, causes SO to be set to zero and OV to be set to one.

Bit(s)	Description
1	**Overflow** (OV) The Overflow bit is used to indicate that an overflow has occurred during execution of an instruction.
2	**Carry** (CA) The Carry bit is used to indicate that a carry out has occurred during execution of an instruction.
3:31	**Reserved**

Fixed-Point Status Image

Fixed-point instructions generate a Fixed-Point Status Image (XSR-Image*),* which can be saved in a General-Purpose Register by executing a special *Extender* instruction in the right-adjacent slot. The *Extender* instruction may also specify a GPR containing a previous XSR-Image whose CA bit is used as an operand for the instruction being extended. The *Extender* instructions are described in Section 5.7, “Extender Instructions,” on page 67.

An XSR-Image is saved in a GPR only if there is an *Extender* instruction in the right-adjacent slot; otherwise, it is discarded. The XSR-Image does not correspond to any architected register. The XSR-Image in a GPR can be used to update XSR with an *Update XSR (uxsr)* instruction.

The bits of the XSR-Image are set based on the operation of an instruction considered as whole, not on intermediate results (e.g., in a *Subtract from Carrying* operation, the result of which is specified as the sum of three values, the XSR-Image bits are set based on the entire operation, not on an intermediate sum).

The bit definitions for the Fixed-Point Status Image are as shown next, wherein *M=0* in 64-bit mode, and *M=32* in 32-bit mode.

Bit(s)	Description
0	**Reserved**
1	**Overflow** (OV) The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction that is being extended with an *Extend XSR* instruction which specifies the OV field. *Add* and *Subtract From* instructions set OV to 1 if the carry out of bit *M* is not equal to the carry out of bit *M*+1, and set it to zero otherwise. *Multiply Low* and *Divide* instructions set the OV field to 1 if the result cannot be represented in 64 bits (*mulld, divd, divdu*), or in 32 bits (*mullw, divw, dviwu*), and set it to 0 otherwise. The OV bit is not altered by other instructions.
2	**Carry** (CA) The Carry bit is set to indicate that a carry out has been generated during execution of an instruction that is being extended with an *Extend XSR* instruction which specifies the CA field. *Add* and *Subtract from* instructions set CA to 1 if there is a carry out of bit *M*, and set it to 0 otherwise. *Shift Right Algebraic* instructions set CA to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by other instructions.
3:31	**Reserved**

2.3.5 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) is a 32-bits register which controls the handling of floating-point exceptions and records the status resulting from floating-point operations. Bits 0:23 are status bits, bits 24:31 are control bits. This register is defined both in VLIW Native mode as well as in PowerPC mode.

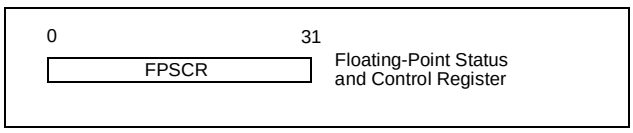


Figure 18: Floating-Point Status and Control Register

The exception bits (bits 0:12 and 21:23) in FPSCR are *sticky,* with the exception of Floating-Point Enabled Exception Summary (FEX) and Floating-Point Invalid Operation

Exception Summary (VX). That is, once set these bits remain set until they are cleared by an *mcrfs, mtfsfi, mtfsf,* or *mtfsb0* instruction. Bits FEX and VX (bits 1 and 2) are simply the ORs of other FPSCR bits.

The rest of this section describes the setting of FPSCR in VLIW Native mode. See *Book I, PowerPC User Instruction Set Architecture* for the definition of the setting of FPSCR in PowerPC mode.

The Floating-Point Status and Control Register is set only by instructions *Update FPSCR (ufsr), Move to Special-Purpose Register (mtspr), Move to FPSCR Field Immediate (mtfsfi), Move to FPSCR Bit (mtfsb0, mtfsb1),* and *Move Condition Register to FPSCR (mcrfsr).*

The field definitions for the *Floating-Point Status and Control Register* are as shown below.

Bit(s)	Description
0	**Floating-Point Exception Summary** (FX) The FX bit is used to indicate whether any of the exception bits in the FPSCR is set to 1. *mcrfs, mtfsfi, mtfsf, mtfsb0* and *mtfsb1* can alter FX explicitly.
1	**Floating-Point Enabled Exception Summary** (FEX) The FEX bit is used to indicate whether any of the enabled exception bits in the FPSCR is set to 1. *mcrfs, mtfsfi, mtfsf, mtfsb0* and *mtfsb1* cannot alter FEX explicitly.
2	**Floating-Point Invalid Operation Exception Summary** (VX) The VX bit is used to indicate whether any invalid operation exception bits are set to 1. *mcrfs, mtfsfi, mtfsf, mtfsb0* and *mtfsb1* cannot alter VX explicitly.
3	**Floating-Point Overflow Exception** (OX) See Section 6.3.3, “Overflow Exception,” on page 128.
4	**Floating-Point Underflow Exception** (UX) See Section 6.3.3, “Overflow Exception,” on page 128.
5	**Floating-Point Zero Divide Exception** (ZX**)** See Section 6.3.3, “Overflow Exception,” on page 128.
6	**Floating-Point Inexact Exception** (XX) See Section 6.3.5, “Inexact Exception,” on page 129.
7	**Floating-Point Invalid Operation Exception (SNaN)** (VXSNAN) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
8	**Floating-Point Invalid Operation Exception** (∞-∞) (VXISI) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
9	**Floating-Point Invalid Operation Exception** (∞÷∞). (VXIDI) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
10	**Floating-Point Invalid Operation Exception** (0÷0) (VXZDZ) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
11	**Floating-Point Invalid Operation Exception** (∞×0) (VXIMZ) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
12	**Floating-Point Invalid Operation Exception** (Invalid Compare) (VXVC) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
13	**Floating-Point Fraction Rounded** (FR) The FR bit is used to indicate whether an *Arithmetic* or *Rounding and Conversion* instruction that rounded the intermediate result increased the fraction (see Section 6.2.6, “Rounding,” on page 122).

Bit(s)	Description
14	**Floating-Point Fraction Inexact** (FI) The FI bit is used to indicate whether an *Arithmetic* or *Rounding and Conversion* instruction either rounded the intermediate result (producing an inexact fraction) or caused a disabled Overflow Exception (see Section 6.2.6, “Rounding,” on page 122). See the definition of XX above, regarding the relationship among FI and XX.
15:19	**Floating-Point Result Flags** (FPRF) For *Arithmetic*, *Rounding*, and *Conversion* instructions, the FPRF field is used to reflect the result placed in a floating-point register, except that if any portion of the result is undefined then the value placed into FPRF is undefined.
15	**Floating-Point Result Class Descriptor** (C) For *Arithmetic*, *Rounding*, and *Conversion* instructions, the C bit is used with the FPCC field to indicate the class of the result placed in a floating-point register, as shown in Figure 19 on page 27.
16:19	**Floating-Point Condition Code** (FPCC) The FPCC field is used with the C bit to indicate the class of the result placed in a floating-point register.
16	**Floating-Point Less Than or Negative** (FL or <)
17	**Floating-Point Greater Than or Positive** (FG or >)
18	**Floating-Point Equal, Zero** (FE or =)
19	**Floating-Point Unordered or NaN** (FU or ?)
20	Reserved
21	**Floating-Point Invalid Operation Exception (Software Request** (VXSOFT)) The VXSOFT bit can be altered only by *mcrfs, mtfsfi, mtfsfm, mtfsb0* or *mtfsb1*. See Section 6.3.1, “Invalid Operation Exception,” on page 126.

Bit(s)	Description
22	**Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT) See Section 6.3.1, “Invalid Operation Exception,” on page 126. **Architecture Note:** This bit is defined even for implementations that do not support either of the two optional instructions, namely *Floating Square Root* and *Floating Reciprocal Square Root Estimate*. Defining it for all implementations gives software a standard interface for handling square root exceptions **Programming Note:** If the implementation does not support the *Floating Square Root* instruction or the *Floating Reciprocal Square Root Estimate* instruction, software can simulate the instruction and set this bit to reflect the exception.
23	**Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
24	**Floating-Point Invalid Operation Exception Enable** (VE) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
25	**Floating-Point Overflow Exception Enable** (OE) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
26	**Floating-Point Underflow Exception Enable** (UE) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
27	**Floating-Point Zero Divide Exception Enable** (ZE) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
28	**Floating-Point Inexact Exception Enable** (XE) See Section 6.3.1, “Invalid Operation Exception,” on page 126.

Bit(s)	Description
29	**Floating-Point Non-IEEE Mode** (NI) If this bit is set to 1, the remaining FPSCR bits may have meanings other than those given in this document, and the results of floating-point operations need not conform to the IEEE standard. If the IEEE-conforming result of a floating-point operation would be a denormalized number, the result of that operation is 0 (with the same sign as the denormalized number) whenever NI=1 and other requirements for the implementation are met (these requirements are specified in *Book IV, ForestaPC Implementation Features* for the implementation). The other effects of setting this bit to 1 are described in Book IV, and may differ among implementations.
30:31	**Floating-Point Rounding Control (RN)** See Section 6.2.6, “Rounding,” on page 122**.** 00 Round to Nearest 01 Round toward Zero 10 Round toward +infinity 11 Round toward -infinity

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	-Infinity
0	1	0	0	0	-Normalized Number
1	1	0	0	0	-Denormalized Number
1	0	0	1	0	-Zero
0	0	0	1	0	+Zero
1	0	1	0	0	+Denormalized Number
0	0	1	0	0	+Normalized Number
0	0	1	0	1	+Infinity

Figure 19: Floating-Point Result Flags

Architecture Note: Setting Floating-Point Non-IEEE Mode (NI) to 1 is intended to permit results to be approximate, and to cause performance to be more predictable and less data-dependent than when NI=0. For example, in Non-IEEE Mode an implementation returns 0 instead of a denormalized number, and may return a large number instead of an infinity. In Non-IEEE mode an implementation should provide the means for ensuring that all results are produced without software assistance (i.e., without causing a Floating-Point Enabled Exception type Program interrupt or a Floating-Point Assist interrupt, and without invoking an “emulation assist,” see *Book III, ForestaPC Operating Environment Architecture*). The means may be controlled by one or more FPSCR bits (recall that the other FPSCR bits have implementation-dependent meanings when NI=1).

Floating-Point Status Image

In VLIW Native mode, most floating-point instructions generate a Floating-Point Status Image (FSR-Image*),* which can be saved in a General-Purpose Register by executing a special *Extender* instruction in the right-adjacent slot. The *Extender* instructions are described in Section 5.7, “Extender Instructions,” on page 67

A FSR-Image is saved only if there is a suitable *Extender* instruction in the right-adjacent slot; otherwise, it is discarded. The FSR-Image does not correspond to any archi-

tected register. The FSR-Image in a GPR can be used to update FPSCR with an *Update FPSCR (ufsr)* instruction.

The definition of fields in the FSR-Image is the same as the status fields in the FPSCR. Control bits in FPSCR do not exist in the FSR-Image. Bits in the FSR-Image are not sticky, that is, they represent the status only of the instruction that generated the image.

Bit(s)	Description
0	**Floating-Point Exception Summary** (FX) The FEX bit is used to indicate whether any of the exception bits in the FSR-Image is set to 1.
1	**Floating-Point Enabled Exception Summary** (FEX) The FEX bit is used to indicate whether any of the enabled exception bits in the FSR-Image is set to 1.
2	**Floating-Point Invalid Operation Exception Summary** (VX) The VX bit is used to indicate whether any invalid operation exception bits in the FSR-Image are set to 1.
3	**Floating-Point Overflow Exception** (OX) See Section 6.3.3, “Overflow Exception,” on page 128.
4	**Floating-Point Underflow Exception (UX)** See Section 6.3.3, “Overflow Exception,” on page 128.
5	**Floating-Point Zero Divide Exception (ZX)** See Section 6.3.3, “Overflow Exception,” on page 128.
6	**Floating-Point Inexact Exception** (XX) See Section 6.3.5, “Inexact Exception,” on page 129.
7	**Floating-Point Invalid Operation Exception (SNaN)** (VXSNAN) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
8	**Floating-Point Invalid Operation Exception** (∞-∞) (VXISI) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
9	**Floating-Point Invalid Operation Exception** (∞÷∞). (VXIDI) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
10	**Floating-Point Invalid Operation Exception** (0÷0) (VXZDZ) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
11	**Floating-Point Invalid Operation Exception** (∞×0) (VXIMZ) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
12	**Floating-Point Invalid Operation Exception** (Invalid Compare) (VXVC) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
13	**Floating-Point Fraction Rounded** (FR) The FR bit is used to indicate whether an *Arithmetic* or *Rounding and Conversion* instruction that rounded the intermediate result increased the fraction (see Section 6.2.6, “Rounding,” on page 122).
14	**Floating-Point Fraction Inexact** (FI) The FI bit is used to indicate whether an *Arithmetic* or *Rounding and Conversion* instruction either rounded the intermediate result (producing an inexact fraction) or caused a disabled Overflow Exception (see Section 6.2.6, “Rounding,” on page 122).
15:19	**Floating-Point Result Flags** (FPRF) For *Arithmetic, Rounding,* and *Conversion* instructions, the field is set based on the result placed in the destination register, except that if any portion of the result is undefined then the value placed into FPRF is undefined.
15	**Floating-Point Result Class Descriptor** (C) *Arithmetic, Rounding,* and *Conversion* instructions set this bit with the FPCC bits, to indicate the class of the result, as shown in Figure 19 on page 27.

Bit(s)	Description
16:19	**Floating-Point Condition Code** (FPCC) Floating-Point *Compare* instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. *Arithmetic, Rounding,* and *Conversion* instructions set the FPCC bits with the C bit to indicate the class of the result, as shown in Figure 19 on page 27. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.
16	**Floating-Point Less Than or Negative** (FL or <)
17	**Floating-Point Greater Than or Positive** (FG or >)
18	**Floating-Point Equal, Zero** (FE or =)
19	**Floating-Point Unordered or NaN** (FU or ?)
20:21	Reserved
22	**Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
23	**Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI) See Section 6.3.1, “Invalid Operation Exception,” on page 126.
24:31	Reserved

Architecture Note:
This bit is defined even for implementations that do not support either of the two optional instructions that set it, namely *Floating Square Root* and *Floating Reciprocal Square Root Estimate*. Defining it for all implementations gives software a standard interface for handling square root exceptions

Programming Note
If the implementation does not support the *Floating Square Root* instruction or the *Floating Reciprocal Square Root Estimate* instruction, software can simulate the instruction and set this bit to reflect the exception.

2.3.6 GPR Delayed Exceptions Register

The GPR Delayed Exceptions Register (GRDX) is a 64-bit register. This register is defined only in VLIW Native mode; it is not defined in PowerPC mode.

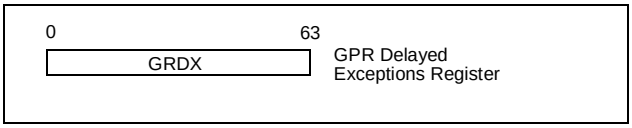


Figure 20: GPR Delayed Exceptions Register

Each bit of GRDX is associated to a General Purpose Register, in left to right order: bit 0 is associated with General Purpose Register 0, bit 1 is associated with GPR 1, etc.

A GRDX bit is set to 1 by any *speculative Load* instruction that stores a result in the corresponding GPR and that incurs an exception. Such speculative load operations cause only the associated Delayed Exception bit to be set but do not raise the exception. A GRDX bit is also set to 1 by any operation that places a result in the corresponding GPR, if it has an operand whose associated Delayed Exception bit is set to 1.

A *Delayed Exception* is raised by a *Commit* instruction which attempts to utilize a GPR whose associated Delayed Exception bit is set to 1.

2.3.7 FPR Delayed Exceptions Register

The FPR Delayed Exceptions Register (FPDX) is a 64-bit register. This register is defined only in VLIW Native mode; it is not defined in PowerPC mode.

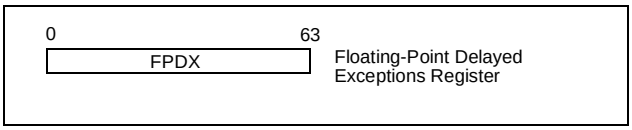


Figure 21: Floating-Point Delayed Exceptions Register

Each bit of FPDX is associated to a Floating-Point Register, in left to right order: bit 0 is associated with Floating-Point Register 0, bit 1 is associated with FPR 1, etc.

A FPDX bit is set to 1 by any *speculative Load* instruction that stores a result in the corresponding FPR and that incurs an exception. Such speculative Load operations cause only the associated Delayed Exception bit to be set but do not raise the exception. An FPDX bit is also set to 1 by any operation that places a result in the corresponding FPR, if it has an operand whose associated Delayed Exception bit is set to 1.

A *Delayed Exception* is raised by a *Commit* instruction which attempts to utilize an FPR whose associated Delayed Exception bit is set to 1.

2.3.8 CR Delayed Exceptions Register

The CR Delayed Exceptions Register (CRDX) is a 16-bit register. This register is defined only in VLIW Native mode; it is not defined in PowerPC mode.

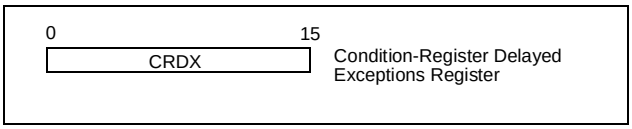


Figure 22: Floating-Point Delayed Exceptions Register

Each bit of CRDX is associated to a Condition Register field, in left to right order: bit 0 is associated with Condition Register Field 0, bit 1 is associated with Condition Register Field 1, etc.

A CRDX bit is set to 1 by any operation that sets the corresponding CR field, if it has an operand whose associated Delayed Exception bit is set to 1.

A *Delayed Exception* is raised by a *commit* instruction which attempts to utilize a Condition Register Field whose associated Delayed Exception bit is set to 1.

2.3.9 Move Assist Register

The Move Assist Register (MAR) is a 64{32}-bit register used by the *Move Assist* instructions. This register is defined only in VLIW Native mode; it is not defined in PowerPC mode.

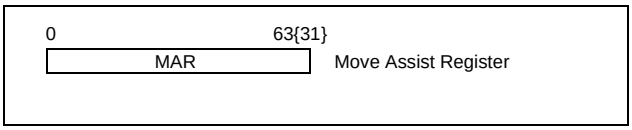


Figure 23: Move Assist Register

MAR is used to specify the ending byte address of the operand (string, block) accessed by the *Move Assist* instructions. When used, it contains the address of the last byte of the string, plus 1.

Chapter 3. Branch Instructions

This chapter describes the Branch instructions in VLIW Native mode. Section 3.1 describes how tree-instruction addresses are specified, Section 3.2 summarizes the registers available to the Branch Processor, Section 3.3 indicates the facilities used for multiway-branching, Section 3.4 describes the procedure call features, and Section 3.5 details the branch primitive instructions.

The features of the Branch instructions in PowerPC mode are described in *Book I, PowerPC User Instruction Set Architecture.*

3.1 Fetching Tree-Instructions

The ForestaPC architecture in VLIW Native mode has no concept of sequential execution of tree-instructions in the order in which tree-instructions appear in storage. Instead, tree-instructions are executed in an order determined at execution time. Each tree-instruction is converted into one or more VLIWs before execution; the resulting VLIWs also corresponds to tree-instructions, perhaps smaller. Each VLIW explicitly indicates the next tree-instruction to be executed; *branch* primitives are used to specify the storage address of the target tree-instructions (one target per *branch* primitive).

Exceptions to the execution order above are:

- *Trap* instructions for which the trap conditions are satisfied, and *System Call* instructions, cause the appropriate system handler to be invoked.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.11, “Exceptions,” on page 18.
- Returning from a system services program, system trap handler, or system error handler causes execution to continue at a specified address.

The model of program execution in which each VLIW appears to complete before the next VLIW starts, and each primitive instruction appears to complete before the next primitive instruction in the taken path of a VLIW starts, is called the “sequential execution model.” In general, from the view of the processor executing the VLIWs and primitive instructions, the sequential model is obeyed. For the instructions and facilities defined in this Book, the only exceptions to this rule are the following:

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 6.3, “Floating-Point Exceptions,” on page 123). The instruction that causes the exception does not complete before the next instruction starts, with respect to setting exception bits and (if exception is enabled) invoking the system error handler.
- A *Store* instruction modifies a storage location that contains an instruction. Software synchronization is required to ensure that subsequent instruction fetches from that location obtain the modified version of the instruction; see *Book III, ForestaPC Operating Environment Architecture*.
- A primitive instruction uses a resource set by an earlier primitive instruction in the taken path of a VLIW. The result of the later primitive instruction is undefined.

Programming Note:

If a program modifies the tree-instructions it intends to execute, it should call the appropriate system library program before attempting to execute the modified instructions, to ensure that the modifications have taken effect with respect to instruction fetching.

3.2 Branch Instructions Registers

The registers accessible to the Branch instructions are the following (see Chapter 2., “Registers in the ForestaPC Architecture,” on page 21 for a description of these registers):

- Branch Registers (BR0, BR1, BR2)
- Condition Register (CR)

In general the bits in the Condition Register fields are named as follows (alternative names are used to represent the setting of a CR field by some specific instructions; see Section 2.3.3, “Condition Register,” on page 22):

Bit	Name
0	**LT : Negative**
1	**GT : Positive**
2	**EQ : Zero**
3	**OV : Overflow**

3.3 Multiway Branch Facilities

The ForestaPC architecture has multiway-branching capabilities with the following features:

- multiple branch conditions in a tree-instruction (multiway decision-tree);
- conditional execution of operations, depending on which path of the multiway tree is taken.

The format of a tree-instruction in storage is depicted in Figure 24.

Three types of branch-related primitives are defined in the architecture:

- *skip* primitives, which appear inside the tree-paths and target a tree-branch *within* the tree-instruction, so the corresponding displacement is a (small) positive value;

```
   L0: skip   cr0.ne,t1
   f1: skip   cr1.gt,t2
   f2: add    r10,cr8,r14,r56
       skip   cr3.eq,t3
   f3: subf   r12,cr9,r14,r44
       andi   r22,r16,0x34
       b      A
   t3: or     r16,cr10,r16,r17
       b      B
   t2: addi   r21,r16,0x1234
       skip   cr4.lt,t4
   f4: andi   r22,r16,0x34
       b      C
   t4: subf   r12,cr9,r14,r44
       b      D
   t1: skip   cr2.eq,t5
   f5: subf   r12,cr9,r14,r44
       addi   r21,r16,0x1234
       b      E
   t5: lbz    r23,64(r2)
       stw    r24,32(r2)
       b      F
```

Figure 24: Example of a tree-instruction

- *direct branches*, which appear at the end of a tree-path and target a tree-instruction within a 2^{28} (256M) bytes *segment;* and
- *register branches,* which appear at the end of a tree-path and target a tree-instruction anywhere in storage, with a Branch Register providing the destination address. This type includes *Branch Register* as well as *System Call* instructions.

Programming Note: Better performance may be obtained when all the targets of a tree-instruction are stored within a 1024-byte block of storage.

Several branch conditions can be specified in a single tree, through a set of *skip* primitive instructions. Each *skip* instruction consists of a test on a Condition Register Field and has an associated target address.

All *skip conditions* in a VLIW are evaluated simultaneously, and a single path through the VLIW is selected as the *taken path*. Operations on the taken path are executed to completion, whereas operations in the other paths are not completed (such operations appear as if they have not been executed at all).

3.4 Procedure calls

The branch-and-link-address features of the PowerPC architecture have been decomposed into separate primitives.

In the case of procedure calls, two primitive instructions are required within a tree-instruction, as follows:

```
Lk: cbri BR0,ret_addr   # save return addr.
                        # in BR0
    b    proc
```

The first one of these primitive instructions saves the return address in a Branch Register, whereas the second one (which is also the last primitive in the tree-path) specifies the branch to the target procedure.

In the case of multiway-branching, each path of a tree-instruction could either call a different procedure or just perform a regular branch (that is, no return address is saved).

The procedure return process is executed as follows:

```
Lj: br   BR0          # branch to the
                      # contents of BR0
```

3.5 Branch Primitive Instructions

The sequence of tree-instructions executed is determined by the *branch primitives*. The set of operations executed within a tree-instruction is determined by the *skip* instructions.

The *Branch* instructions specify the effective address (EA) of the target in one of the following ways:

1. Concatenating a 28-bit offset to the most-significant bits of the address of the current tree-instruction *(Unconditional Branch*).
2. Using the address contained in a Branch Register (*Branch Register).*

The *Skip* instructions compute the effective address of the target tree-branch by concatenating a 13-bit offset to the most-significant bits of the address of the current tree-instruction.

Architecture Note: A tree-instruction may not straddle a 2^{20} word memory segment.

Branching to the next tree-instruction is always unconditional, without providing a return address. If a return address is required, it is computed explicitly with the instruction *Compute Branch Register Immediate.*

In *Skip* instructions, field CRS specifies the Condition Register field tested, and field BC specifies the condition under which the taken tree-branch is selected at the point of the *Skip* instruction. The encoding for field BC uses the first two bits to indicate the bit tested in the CR field, whereas the third bit indicates the value tested, as follows:

Code	Condition	Symbol	Bit
000	Greater Than or Equal	ge	LT = 0
001	Less Than	lt	LT = 1
010	Less Than or Equal	le	GT = 0
011	Greater Than	gt	GT = 1
100	Not Equal	ne	EQ =0
101	Equal	eq	EQ = 1
110	No Overflow	no	OV = 0
111	Overflow	ov	OV =1

Extended mnemonics for skip instructions

Many extended mnemonics are provided so that *Skip* instructions can be coded with the condition as part of the instruction mnemonic rather than as an operand. Some of these are shown with the *Skip Conditional* instruction.

3.5.1 Skip Instruction

This instruction provides the means by which a program specifies the different tree-branches within a tree-instruction and the conditions under which each tree-branch is selected for execution.

Skip Conditional B10-form

skip CRS,BC,target_addr

0	4	8	11	22
0	CRS	BC	ADDR	223

if ($CRS_{BC(0:1)}$ = BC_2) then
    NIA $\leftarrow_{iea}$ $CIA_{0:50}$ || ADDR || 0b00
else
    NIA $\leftarrow_{iea}$ CIA + 4

target_addr specifies the address of the target tree-branch.

The tree-instruction splits into two tree-branches at the location of the *skip* instruction; only one of these two tree-branches is executed, depending on the condition. If the condition is true, the tree-branch starting at address $CIA_{0:50}$ concatenated with (ADDR || 0b00) is executed; otherwise, the tree-branch starting at address CIA+4 is executed. The high-order 32 bits of this address are set to 0 in 32-bit mode of 64-bit implementations.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Skip Conditional*:

Extended:		*Equivalent to:*	
skeq	CRS,target	skip	CRS,5,target
skne	CRS,target	skip	CRS,4,target

3.5.2 Branch Instructions

These instructions provide the means by which a program specifies the next tree-instruction to be executed. These instructions indicate the end of a tree-path.

Branch Unconditional B2-form

b target_addr

0	4	30
10	ADDR	0

NIA $\leftarrow_{iea}$ $CIA_{0:35}$ || ADDR || 0b00

target_addr specifies the address of the target tree-instruction.

The target tree-instruction address is the value $CIA_{0:35}$ concatenated with (ADDR || 0b00). The high-order 32 bits of this address are set to 0 in 32-bit mode of 64-bit implementations.

Special Registers Altered:

None

Branch Register X10-form

br BRS

0	4	10	12	16	22
0	///	BRS	///	///	818

NIA $\leftarrow_{iea}$ $BRS_{0:61}$ || 0b00

The target tree-instruction address is the value $BRS_{0:61}$ concatenated with 0b00. The high-order 32 bits of this address are set to 0 in 32-bit mode of 64-bit implementations.

Special Registers Altered:

None

3.5.3 System Call Instruction

This instruction provides the means by which a program can call upon the system to perform a service. This instruction indicates the end of a tree-path.

System Call X10-form

sc

0	4	10	16	22
0	///	///	///	823

This instruction calls the system to perform a service. A complete description of this instruction can be found in *Book III, ForestaPC Operating Environment Architecture*.

A *System Call* instruction has the same basic functionality as a *Branch Register* instruction, and is placed as the last operation in a tree-path (in the same way as *Branch Register* primitives). See *Book III, ForestaPC Operating Environment Architecture* for additional functions performed by the *System Call* instruction.

When a *System Call* instruction is executed, Branch Register BR2 must contain the value 0xC00; if this is not observed, the system illegal instruction error handler is invoked.

The address of the next tree-instruction to be executed after returning from the system call is usually computed with a separate *Compute Branch Register Immediate* instruction, and is stored in a Branch Register.

When control is returned to the program that executed a *System Call* instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

This instruction is context synchronizing (see *Book III, ForestaPC Operating Environment Architecture*).

Special Registers Altered:

Dependent on the system service

Chapter 4. Storage Access Instructions

This chapter describes the Storage Access instructions in VLIW Native mode. Section 4.1 lists the registers accessible to the storage access instructions, Section 4.2 describes general features of the Storage Access Instruction Set Architecture, whereas the remaining sections describe the corresponding instructions.

The features of the Storage Access instructions in PowerPC mode are described in *Book I, PowerPC User Instruction Set Architecture.*

4.1 Storage Access Registers

The set of registers accessible by the Storage Access instructions consists of

- sixty-four General Purpose Registers (GPRs)
- sixty-four Floating-Point Register (FPRs)
- the 64-bit Condition Register (CR)
- three Branch Registers (BRs)
- the 64-bit GPR Delayed Exceptions Register (GRDX) and the 64-bit FPR Delayed Exceptions Register (FPDX).
- the 16-bit CR Delayed Exceptions Register (CRDX)
- other Special-Purpose Registers

See Chapter 2., “Registers in the ForestaPC Architecture,” on page 21 for a complete description of these registers and their fields.

4.2 General Features

Storage Access instructions operate on the General Purpose Registers and Floating-Point Registers. These registers may be the source or destination of *Storage Access* instructions. Some *Storage Access* instructions specify a Condition Register field which is set depending on the result of the operation.

Each GPR has an associated bit in GRDX (the bit whose number is the same as the General Purpose Register number). Each FPR has an associated bit in FPDX (the bit whose number is the same as the Floating-Point Register number). Each CR field has an associated bit in CRDX (the bit whose number is the same as the Condition Register Field number).

The description of *Storage Access* instructions in this chapter does not include the setting of GRDX, FPDX or CRDX; it is assumed that instructions follow the rules described above regarding these entities.

4.2.1 Effective Address

Storage Access instructions compute the Effective Address (EA) of the storage to be accessed, as described in Section 1.13.2, “Effective Address Calculation,” on page 20.

The order of bytes accessed by halfword, word, and doubleword loads and stores is Big-Endian, unless Little-Endian storage is selected.

Storage Access instructions have a single mode for computing the effective address: register plus an 11-bits signed displacement.

4.2.2 Floating-Point Storage Accesses

Load and *Store Floating-Point Double* instructions transfer 64 bits of data between storage and the Floating-Point Registers, with no conversion.

Load Floating-Point Single instructions transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the Floating-Point Registers.

Store Floating-Point Single instructions transfer and convert floating-point values in floating-point double format from the Floating-Point Registers to the same value in floating-point single format in storage.

See Chapter 6., “Floating-Point Instructions,” on page 117 for additional details on floating-point data formats.

4.2.3 Storage Access Exceptions

Storage Access instructions will cause the system error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

4.2.4 Speculative Load Instructions

Load instructions have a single-bit field (SF) which is used to specify when the instruction is *speculative* (it has been moved by the compiler/programmer backward in the instruction stream, across a conditional branch).

Speculative Load instructions (SF=1) have the following additional functionality:

- if an exception occurs when executing a speculative *Load* instruction, the only effect of the instruction is to set the bit in the Delayed Exceptions Register associated with the target register of the instruction; the exception is not raised to the processor.

4.3 Fixed-Point Load Instructions

The byte, halfword, word or doubleword in storage addressed by EA is loaded into a General Purpose Register.

Programming Note: In some implementations, the *Load Algebraic* instructions may have greater latency than other types of *Load* instructions.

Load Byte and Zero D4-form

lbz[?] RT,D(RA)

0	4	10	16	27	28
11	RT	RA	D	SF	6

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT  ← 56 0 ‖ MEM(EA,1)
```

Let the effective address (EA) be the sum (RA|0) + D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero D4-form

lhz[?] RT,D(RA)

0	4	10	16	27	28
11	RT	RA	D	SF	1

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT  ← 48 0 ‖ MEM(EA,2)
```

Let the effective address (EA) be the sum (RA|0) + D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword Algebraic D4-form

lha[?] RT,D(RA)

0	4	10	16	27	28
11	RT	RA	D	SF	5

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT  ← EXTS(MEM(EA,2))
```

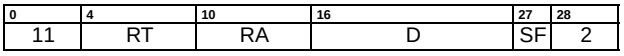
Let the effective address (EA) be the sum (RA|0) + D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword

Special Registers Altered:

None

Load Word and Zero D4-form

lwz[?] RT,D(RA)



0	4	10	16	27	28
11	RT	RA	D	SF	2

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT  ← 320 || MEM(EA,4)
```

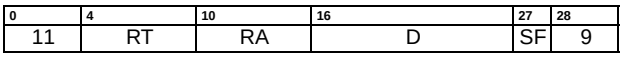
Let the effective address (EA) be the sum (RA|0) + D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word Algebraic D4-form

lwa[?] RT,D(RA)



0	4	10	16	27	28
11	RT	RA	D	SF	9

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT  ← EXTS(MEM(EA,4))
```

Let the effective address (EA) be the sum (RA|0) + D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

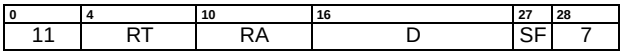
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

Load Doubleword D4-form

ld[?] RT,D(RA)



0	4	10	16	27	28
11	RT	RA	D	SF	7

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT  ← MEM(EA,8)
```

Let the effective address (EA) be the sum (RA|0) + D. The doubleword in storage addressed by EA is loaded into RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

4.4 Fixed-Point Store Instructions

The contents of a General Purpose Register are stored into the byte, halfword, word or doubleword in storage addressed by EA.

Store Byte D5-form

stb RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	3

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,1)   ← (RB)56:63
```

Let the effective address (EA) be the sum (RA|0) + D. $(RB)_{56:63}$ is stored into the byte in storage addressed by EA.

Special Registers Altered:

None

Store Halfword D5-form

sth RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	1

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,2)   ← (RB)48:63
```

Let the effective address (EA) be the sum (RA|0) + D. $(RB)_{48:63}$ is stored into the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Word D5-form

stw RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	6

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,4)   ← (RB)32:63
```

Let the effective address (EA) be the sum (RA|0) + D. $(RB)_{32:63}$ is stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Doubleword D5-form

std RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	8

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,8)   ← (RB)
```

Let the effective address (EA) be the sum (RA|0) + D. (RB) is stored into the doubleword in storage addressed by EA.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

4.5 Floating-Point Load Instructions

There are two basic forms of *Floating-Point Load* instructions: single-precision and double-precision. As floating-point registers support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operands into the target floating-point register.

The conversion and loading steps are as follows. Let $WORD_{0:31}$ be the floating-point single-precision data accessed from storage.

Normalized Operand

```
If (WORD1:8>0) and (WORD1:8<255) then
  FRT0:1      ←  WORD0:1
  FRT2        ← ¬WORD1
  FRT3        ← ¬WORD1
  FRT4        ← ¬WORD1
  FRT5:63     ←  WORD2:31 || 29 0
```

Denormalized Operand

```
If (WORD1:8=0) and (WORD9:31≠0) then
  sign        ←  WORD0
  exp         ←  -126
  frac0:52    ←  0b0 || WORD9:31 || 29 0
  normalize the operand
    Do while frac0 = 0
      frac    ←  frac1:52 || 0b0
      exp     ←  exp - 1
    End
  FRT0        ←  sign
  FRT1:11     ←  exp + 1023
  FRT12:63    ←  frac1:52
```

Zero / Infinity / NaN

```
If (WORD1:8=255) or (WORD1:31=0) then
  FRT0:1      ← WORD0:1
  FRT2        ← WORD1
  FRT3        ← WORD1
  FRT4        ← WORD1
  FRT5:63     ← WORD2:31 || 29 0
```

For double-precision *Load Floating-Point* instructions, no conversion is required because the data from storage is copied directly into the floating-point registers.

Engineering Note: The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

Load Floating-Point Single D4-form

lfs[?] FRT,D(RA)

0	4	10	16	27	28
11	FRT	RA	D	SF	12

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
FRT ← DOUBLE(MEM(EA,4))
```

Let the effective address (EA) be the sum (RA|0) + D. The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 42) and loaded into register FRT.

Special Registers Altered:

None

Load Floating-Point Double D4-form

lfd[?] FRT,D(RA)

0	4	10	16	27	28
11	FRT	RA	D	SF	13

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
FRT  ← MEM(EA,8)
```

Let the effective address (EA) be the sum (RA|0) + D. The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:

None

4.6 Floating-Point Store Instructions

There are two basic forms of *Floating-Point Store* instruction: single-precision and double-precision. As floating-point registers support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format prior to storing the operands into the storage.

The conversion steps are as follows: Let $WORD_{0:31}$ be the word in storage written to.

No Denormalization Required (includes Zero / Infinity / NaN)

```
If (FRB1:11>896) or (FRB1:63=0) then
  WORD0:1    ← (FRB)0:1
  WORD2:31   ← (FRB)5:34
```

Denormalization Required

```
If (874 ≤ FRB1:11 ≤896) then
  sign ← (FRA)0
  exp  ← (FRA)1:11 - 1023
  frac ← 0b1 || (FRA)12:63
  Denormalize the operand
    Do while exp < -126
      frac ← 0b0 || frac0:62
      exp ← exp + 1
    End
  WORD0    ← sign
  WORD1:8  ← 0x00
  WORD9:31 ← frac1:23
else
  WORD ← undefined
```

Notice that, if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well defined value, but is not numerically equal to the value in the source register (i.e., the result of a *Load Floating-Point Single* from WORD will not compare equal to the contents of the original source register).

Engineering Note: The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Store Floating-Point* instructions, no conversion is required because the data from the FPR is copied directly into storage.

Store Floating-Point Single D5-form

stfs FRB,D(RA)

0	4	10	16	22	27
13	d_0	RA	FRB	d_1	10

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,4)   ← SINGLE(FRB)
```

Let the effective address (EA) be the sum (RA|0) + D. The contents of register FRB are converted to single format (see page 42) and stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Double D5-form

stfd FRB,D(RA)

0	4	10	16	22	27
13	d_0	RA	FRB	d_1	11

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,8)   ← (FRB)
```

Let the effective address (EA) be the sum (RA|0) + D. The contents of register FRB are stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

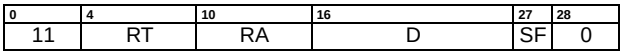
4.7 Fixed-Point Load and Store with Byte Reversal Instructions

When used in a system operating with Big-Endian byte order (the default), these instructions have the effect of loading and storing data in Little-Endian order. Likewise, when used in a system operating with Little-Endian byte order, these instructions have the effect of loading and storing data in Big-Endian order.

Programming Note: In some implementations, the *Load Byte-Reverse* instructions may have greater latency than other *Load* instructions.

Load Halfword Byte-Reversed D4-form

lhbr[?] RT,D(RA)



0	4	10	16	27	28
11	RT	RA	D	SF	0

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT ← 48 0 || MEM(EA+1,1) || MEM(EA,1)
```

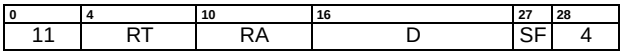
Let the effective address (EA) be the sum (RA|0) + D. Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the halfword in storage addressed by EA are loaded into $RT_{48:55}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Word Byte-Reversed D4-form

lwbr[?] RT,D(RA)



0	4	10	16	27	28
11	RT	RA	D	SF	4

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RT ← 32 0 || MEM(EA+3,1) || MEM(EA+2,1)
         || MEM(EA+1,1) || MEM(EA,1)
```

Let the effective address (EA) be the sum (RA|0) + D. Bits 0:7 of the word in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the word in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the word in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the word in storage addressed by EA are loaded into $RT_{32:39}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Store Halfword Byte-Reversed D5-form

sthbr RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	0

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,2)   ← (RB)56:63 || (RB)48:55
```

Let the effective address (EA) be the sum (RA|0) + D. $(RB)_{56:63}$ are stored into bits 0:7 of the halfword in storage addressed by EA. $(RB)_{48:55}$ are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Word Byte-Reversed D5-form

stwbr RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	4

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
MEM(EA,4)   ← (RB)56:63 || (RB)48:55
              || (RB)40:47 || (RB)32:39
```

Let the effective address (EA) be the sum (RA|0) + D. $(RB)_{56:63}$ are stored into bits 0:7 of the word in storage addressed by EA. $(RB)_{48:55}$ are stored into bits 8:15 of the word in storage addressed by EA. $(RB)_{40:47}$ are stored into bits 16:23 of the word in storage addressed by EA. $(RB)_{32:39}$ are stored into bits 24:31 of the word in storage addressed by EA.

Special Registers Altered:

None

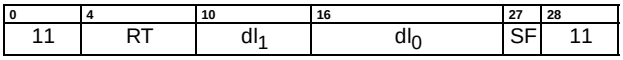
4.8 Load Table of Contents Instructions

The *Load Table of Contents* (TOC) instructions are used for accessing tables of externally referenced variables. These instructions assume that General Purpose Register 2 contains a pointer to the TOC area, thus allowing the instructions to specify a 19-bit displacement field.

Programming Note: Better performance may be obtained with *Load Table of Contents* instructions if the pointer in GPR(2) is aligned on a 2^{19} (512k byte) boundary.

Load TOC Word and Zero D4-form

ltocwz[?] RT,DL



0	4	10	16	27	28
11	RT	dl_1	dl_0	SF	11

```
DL  ← dl0 || dl1 || 0b00
EA  ← (R2) + DL
RT  ← 32 0 || MEM(EA, 4)
```

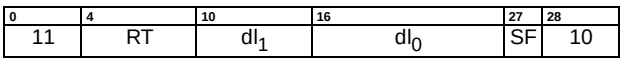
Let the effective address (EA) be the sum (R2) + DL || 0b00. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ is set to 0.

Special Registers Altered:

None

Load TOC Doubleword D4-form

ltocd[?] RT,DL



0	4	10	16	27	28
11	RT	dl_1	dl_0	SF	10

```
DL  ← dl0 || dl1 || 0b00
EA  ← (R2) + DL
RT  ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (R2) + DL || 0b00. The doubleword in storage addressed by EA is loaded into RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

4.9 Load and Store String Instructions

The *Load/Store String* instructions allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

A set of *Load/Store String* primitives together with a set of *Shift Left/Right String* primitives allow arbitrarily aligned strings to be moved quickly.

Loading a string starting from an arbitrary alignment is implemented as a two-step process:

- load several aligned storage locations into GPRs; and
- simultaneously left-shift several GPRs.

Similarly, storing a string at an arbitrary alignment is implemented as a two-step process:

- simultaneously right-shift several GPRs; and
- store several GPRs into aligned storage locations.

The *Load/Store String* instructions use two registers to specify the string, as follows:

- RA: a General Purpose Register containing the starting storage address (byte address) of the string;
- MAR: a Special Purpose Register containing the ending byte address of the string, plus 1.

Load/Store String instructions of length zero have no effect, except that *Load String* instructions of length zero may set the destination register to an undefined value.

On systems operating with Little-Endian byte order, the execution of a *Load/Store String* instruction causes the system alignment error handler to be invoked.

Programming Note: The PowerPC string instructions have been decomposed into simpler primitives in the ForestaPC architecture; these primitive instructions are executed concurrently in different parcels (composing a multiparcel primitive).

Programming Note: In contrast to the PowerPC architecture, these instructions use the starting and ending byte address of the string instead of the starting address and the byte count.

Load String Word and Zero D4-form

lswz[?] RT,D(RA)

0	4	10	16	27	28
11	RT	RA	D	SF	8

```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + D) & (^62 1 || 0b00)

if EA0:61 < MAR0:61 then
   nb ← 4
else if EA0:61 = MAR0:61 then
   nb ← MAR62:63
else
   nb ← 0

if nb > 0 then
   RT ← ^32 0 || MEM(EA,nb) || ^8×(4-nb) 0
else
   RT ← 0
```

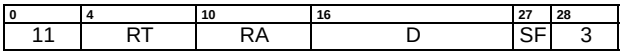
General Purpose Register RA and Special Purpose Register MAR, respectively, contain the starting byte address and ending byte address plus 1 of a string. Let the effective address (EA) be the sum ((RA|0) + D) ANDed with ($^{62}1$ || 0b00). EA is the address of the aligned word in storage which contains the first byte to be loaded. Let *nb* be the number of bytes to be loaded, which is determined from the difference between the address of the aligned word containing the first byte to be loaded and the ending byte address plus 1 of the string. Based on the value of *nb*, 0 to 4 bytes in storage addressed by EA are loaded left aligned into $RT_{32:63}$, padding the data to the right with zeros when fewer than four bytes are loaded. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load String Doubleword D4-form

lsd[?] RT,D(RA)



0	4	10	16	27	28
11	RT	RA	D	SF	3

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + D) & (61 1 || 0b000)

if EA0:60 < MAR0:60 then
   nb ← 8
else if EA0:60 = MAR0:60 then
   nb ← MAR61:63
else
   nb ← 0

if nb > 0 then
   RT ← MEM(EA,nb) || 8×(8-nb)0
else
   RT ← 0
```

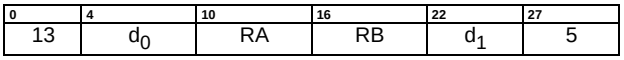
General Purpose Register RA and Special Purpose Register MAR, respectively, contain the starting byte address and ending byte address plus 1 of a string. Let the effective address (EA) be the sum ((RA|0) + D) ANDed with ($^{61}1$ || 0b000). EA is the address of the aligned doubleword in storage which contains the first byte to be loaded. Let *nb* be the number of bytes to be loaded, which is determined from the difference between the address of the aligned doubleword containing the first byte to be loaded and the ending byte address plus 1 of the string. Based on the value of *nb*, 0 to 8 bytes in storage addressed by EA are loaded left aligned into RT, padding the data to the right with zeros when fewer than eight bytes are loaded.

Special Registers Altered:

None

Store String Word D5-form

stsw RB,D(RA)



0	4	10	16	22	27
13	d_0	RA	RB	d_1	5

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + D
if D > 3 then EA ← EA & (62 1 || 0b00)
fb ← EA62:63

if EA0:61 < MAR0:61 then
   lb ← 4
else if EA0:61 = MAR0:61 then
   lb ← MAR62:63
else
   lb ← 0

if lb > fb then
  MEM(EA,lb-fb) ← RB8×fb+32:8×lb+31
else
  null
```

General Purpose Register RA and Special Purpose Register MAR, respectively, contain the starting byte address and ending byte address plus 1 of a string. Let the effective address (EA) be the sum ((RA|0) + D) ANDed with ($^{62}1$ || 0b00) if D is greater than 3. EA is the address of the byte in storage where the first byte must be stored. Let *fb* and *lb* be the byte number of the first byte and last byte plus 1, respectively, to be stored within the aligned word in storage. Based on the difference between *fb* and *lb*, 0 to 4 bytes from the low-order 32 bits of register RB are stored in storage starting at address EA.

Special Registers Altered:

None

Store String Doubleword D5-form

stsd RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	7

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + D
if D > 7 then EA ← EA & (61)1 || 0b000)
fb ← EA61:63

if EA0:60 < MAR0:60 then
   lb ← 8
else if EA0:60 = MAR0:60 then
   lb ← MAR61:63
else
   lb ← 0

if lb > fb then
  MEM(EA,lb-fb) ← RB8×fb:8×lb-1
else
  null
```

General Purpose Register RA and Special Purpose Register MAR, respectively, contain the starting byte address and ending byte address plus 1 of a string. Let the effective address (EA) be the sum ((RA|0) + D) ANDed with ($^{61}1$ || 0b000) if D is greater than 7. EA is the address of the byte in storage where the first byte must be stored. Let *fb* and *lb* be the byte number of the first byte and last byte plus 1, respectively, to be stored within the aligned word in storage. Based on the difference between *fb* and *lb*, 0 to 8 bytes from register RB are stored in storage starting at address EA.

Special Registers Altered:

None

4.10 Storage Synchronization Instructions

The *Storage Synchronization* instructions can be used to control the order in which storage operations are completed with respect to asynchronous events, and the order in which storage operations are seen by other processors and by other mechanisms that access storage. Additional information about these instructions, and about related aspects of storage management, can be found in *Book II, ForestaPC Virtual Environment Architecture*, and *Book III, ForestaPC Operating Environment Architecture*.

The *Load and Reserve* and *Store Conditional Reserve* instructions permit the programmer to write a sequence of instructions that appear to perform an atomic update operation on a storage location. This operation depends upon a single reservation resource in each processor. At most one reservation exists on any given processor; there are not separate reservations for words and for doublewords.

On a system operating with Little-Endian byte order, the three low-order bits of the Effective Address computed by instructions *Load and Reserve* and *Store Conditional Reserve* are modified before accessing storage.

Load and Reserve instructions cannot be executed speculatively, so these instructions do not have a *SF* bit.

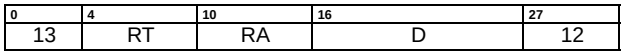
Programming Note: Because the *Storage Synchronization* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, etc.) needed by application programs. Application programs should use these library programs, rather than use the *Storage Synchronization* instructions directly.

Architecture Note: The *Load and Reserve* and *Store Conditional Reserve* instructions require the EA to be aligned. Software should not attempt to emulate an unaligned *Load and Reserve* or *Store Conditional Reserve* instruction, because there is no correct way to define the address associated with the reservation.

Engineering Note: Causing the system alignment error handler to be invoked if attempt is made to execute a *Load and Reserve* or *Store Conditional Reserve* instruction having an incorrectly aligned Effective Address facilitates the debugging of software.

Load Word and Reserve D5-form

lwar RT,D(RA)



0	4	10	16	27
13	RT	RA	D	12

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 32 0 || MEM(EA,4)
```

Let the effective address (EA) be the sum (RA|0) + D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

This instruction creates a reservation for use by a *Store Word Conditional Reserve* instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. The manner in which the address to be associated with the reservation is computed from the EA is described in *Book II, ForestaPC Virtual Environment Architecture*.

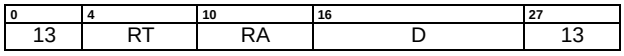
EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Load Doubleword and Reserve D5-form

ldar RT,D(RA)



0	4	10	16	27
13	RT	RA	D	13

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← MEM(EA,8)
```

Let the effective address (EA) be the sum (RA|0) + D. The doubleword in storage addressed by EA is loaded into RT.

This instruction creates a reservation for use by a *Store Doubleword Conditional Reserve* instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. The manner in which the address to be associated with the reservation is computed from the EA is described in *Book II, ForestaPC Virtual Environment Architecture*.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

Store Word Conditional Reserve D5-form

stwcr RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	2

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + D
if RESERVE then
   if RESERVE_ADDR=real_addr(EA) then
      MEM(EA,4) ← (RB)32:63
      CR8 ← 0b0010
   else
      u ← undefined 1-bit value
      if u then MEM(EA,4) ← (RB)32:63
      CR8 ← 0b00 || u || 0b0
   RESERVE ← 0
else
   CR8 ← 0b0000
```

Let the effective address (EA) be the sum (RA|0)+D.

If a reservation exists and the storage address specified by the *stwcr* instruction is the same as that specified by the *Load and Reserve* instruction that established the reservation, $(RB)_{32:63}$ is stored into the word in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage address specified by the *stwcr* instruction is not the same as that specified by the *Load and Reserve* instruction that established the reservation, the reservation is cleared, and it is undefined whether $(RB)_{32:63}$ is stored into the word in storage addressed by EA.

If the reservation does not exist, the instruction completes without altering storage.

CR Field 8 is set to reflect whether the store operation was performed, as follows:

CR8 = 0b00 || store_performed || 0b0

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR Field 8

Programming Note: The granularity with which reservations are managed is implementation-dependent. Therefore, the storage to be accessed by the *Load and Reserve* and *Store Conditional Reserve* instructions should be allocated by a system library program. Additional information can be found in *Book II, ForestaPC Virtual Environment Architecture*.

Store Doubleword Conditional Reserve D5-form

stdcr RB,D(RA)

0	4	10	16	22	27
13	d_0	RA	RB	d_1	9

```
D ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + D
if RESERVE then
   if RESERVE_ADDR=real_addr(EA) then
      MEM(EA,8) ← (RB)
      CR8 ← 0b0010
   else
      u ← undefined 1-bit value
      if u then MEM(EA,4) ← (RB)
      CR8 ← 0b00 || u || 0b0
   RESERVE ← 0
else
   CR8 ← 0b0000
```

Let the effective address (EA) be the sum (RA|0) + D.

If a reservation exists and the storage address specified by the *stdcr* instruction is the same as that specified by the *Load and Reserve* instruction that established the reservation, (RB) is stored into the word in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage address specified by the *stdcr* instruction is not the same as that specified by the *Load and Reserve* instruction that established the reservation, the reservation is cleared, and it is undefined whether (RB) is stored into the word in storage addressed by EA.

If the reservation does not exist, the instruction completes without altering storage.

CR Field 8 is set to reflect whether the store operation was performed, as follows:

CR8 = 0b00 || store_performed || 0b0

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

CR Field 8

Programming Note: When correctly used, the *Load and Reserve* and *Store Conditional Reserve* instructions can provide an atomic update function for a single aligned word (*Load Word And Reserve* and *Store Word Conditional Reserve*) or doubleword (*Load Doubleword And Reserve* and *Store Doubleword Conditional Reserve*) of storage.

In general, correct use requires that *Load Word And Reserve* be paired with *Store Word Conditional Reserve*, and *Load Doubleword And Reserve* with *Store Doubleword Conditional Reserve*, with the same storage address specified by both instructions of the pair. The only exception is that a non-paired *Store Word Conditional Reserve or Store Doubleword Conditional Reserve* instruction to any (scratch) storage address can be used to clear any reservation held by the processor.

A reservation is cleared if any of the following events occurs:

- The processor holding the reservation executes another *Load and Reserve* instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a *Store Conditional Reserve* instruction to *any* address.
- Another processor executes any *Store* instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

See *Book II, ForestaPC Virtual Environment Architecture*, for additional information.

Synchronize X10-form

sync

0	4	10	16	22
0	///	///	///	825

The *sync* instruction provides an ordering function for the effects of all instructions executed by a given processor. Executing a *sync* instruction ensures that all instructions previously initiated by the given processor, as well as all other instructions in the same VLIW as the *sync* instruction, appear to have completed before the *sync* instruction completes, and that no subsequent instructions are initiated by the given processor until after the *sync* instruction completes. When the *sync* instruction completes, all storage accesses initiated by the given processor prior to the *sync* will have been performed with respect to all other mechanisms that access storage. (See *Book II, ForestaPC Virtual Environment Architecture*, for a more complete description. See also the section entitled “Table Update Synchronization Requirements” in *Book III, ForestaPC Operating Environment Architecture*, for an exception involving TLB invalidates.)

The *sync* instruction must be the last instruction in a tree-path, immediately preceding the branch primitive that ends the path.

This instruction is execution synchronizing (see *Book III, ForestaPC Operating Environment Architecture*).

Special Registers Altered:

None

Programming Note: The *sync* instruction can be used to ensure that the results of all stores into a data structure, performed in a “critical section” of a program, are seen by other processors before the data structure is seen as unlocked. Examples of use of the *sync* instruction can be found in *Book II, ForestaPC Virtual Environment Architecture* and *Book III, ForestaPC Operating Environment Architecture*.

The functions performed by the *sync* instruction will normally take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. In addition, the time required to execute *sync* may vary from one execution to another.

The *Enforce In-order Execution of I/O* (*eieio*) instruction, described in *Book II, ForestaPC Virtual Environment Architecture*, may be more appropriate than *sync* for many cases.

Engineering Note: The guarantee that *sync* ensures that all prior stores have been performed with respect to all other mechanisms that access storage applies to coherent accesses. See Book II.

Engineering Note: Unlike a context synchronizing operation, *sync* need not discard prefetched instructions.

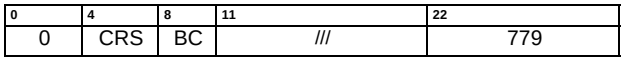
4.11 Conditional Store Extender Instructions

These instructions transform a store instruction into a two-parcel primitive which executes in two adjacent slots in a VLIW. The right-most slot used by the multiparcel primitive contains a *Conditional Store Extender* instruction. The *Conditional Store Extender* instructions are used to condition the execution of the store instruction in the left-adjacent parcel.

Conditional Store Extended instructions (two-parcel instructions) are regarded as a single indivisible operation for the purposes of VLIW semantics. That is, the results from the operation consist of the results generated by the two-parcel instruction.

Extend Conditional Store B10-form

xcst CRS,BC



0	4	8	11	22
0	CRS	BC	///	779

```
if (left-adj-inst is store) then
  if (CRS_BC(0:1) = BC_2) then
    perform store operation specified by
    left-adj-inst
```

If the instruction executing in the left-adjacent slot is a store operation, and if the condition specified by the instruction is true, then the store operation in the left-adjacent slot is performed. If the condition specified by the instruction is false, then the the store operation in the left-adjacent slot is not performed.

If the left-adjacent parcel does not specify a store operation, the instruction form is invalid.

Special Registers Altered:

None

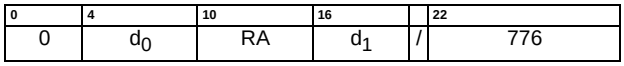
4.12 Store Extender Instructions

These instructions transform a primitive instruction into a two-parcel primitive which executes in two adjacent slots in a VLIW. The right-most slot used by the multiparcel primitive contains a *Store Extender* instruction. The *Store Extender* instructions are used to store the result computed in the left-adjacent parcel that is placed in a GPR. *Store Extender* instructions cannot be used to extend a *Load* instruction.

Store Extended instructions (two-parcel instructions) are regarded as a single indivisible operation for the purposes of VLIW semantics. That is, the results from the operation consist of the results generated by the two-parcel instruction.

Extend Store Doubleword D10-form

xstd D(RA)



0	4	10	16		22
0	d_0	RA	d_1	/	776

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + D
if left-adj-inst specifies RT then
   MEM(EA,8) ← (RT) computed
                by left-adj-op
```

Let the effective address (EA) be the sum (RA|0) + D. If the instruction executing in the left-adjacent slot targets a General Purpose Register, the result from the operation in the left-adjacent slot is stored into the doubleword in storage addressed by EA.

If the left-adjacent parcel does not specify a target General Purpose Register, or is a *Load* instruction, the instruction form is invalid.

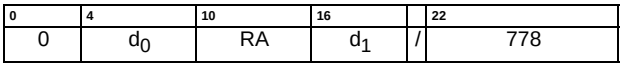
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

Extend Store Word D10-form

xstw D(RA)



0	4	10	16		22
0	d_0	RA	d_1	/	778

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
if left-adj-inst specifies RT then
   MEM(EA,4) ← (RT)32:63 computed
                by left-adj-op
```

Let the effective address (EA) be the sum (RA|0) + D. If the instruction executing in the left-adjacent slot targets a General Purpose Register, bits 32:63 of the result from the operation in the left-adjacent slot are stored into the word in storage addressed by EA.

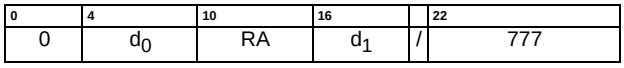
If the left-adjacent parcel does not specify a target General Purpose Register, or is a *Load* instruction, the instruction form is invalid.

Special Registers Altered:

None

Extend Store Halfword D10-form

xsth D(RA)



0	4	10	16		22
0	d_0	RA	d_1	/	777

```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
if left-adj-inst specifies RT then
   MEM(EA,2) ← (RT)48:63 computed
                by left-adj-op
```

Let the effective address (EA) be the sum (RA|0) + D. If the instruction executing in the left-adjacent slot targets a General Purpose Register, bits 48:63 of the result from the operation in the left-adjacent slot are stored into the halfword in storage addressed by EA.

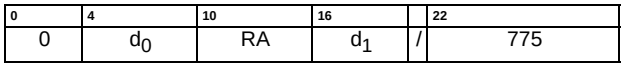
If the left-adjacent parcel does not specify a target General Purpose Register, or is a *Load* instruction, the instruction form is invalid.

Special Registers Altered:

None

Extend Store Byte D10-form

xstb D(RA)




```
D  ← d0 || d1
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + D
if left-adj-inst specifies RT then
   MEM(EA,1) ← (RT)56:63 computed
                by left-adj-op
```

Let the effective address (EA) be the sum (RA|0) + D. If the instruction executing in the left-adjacent slot targets a General Purpose Register, bits 56:63 of the result from the operation in the left-adjacent slot are stored into the half-word in storage addressed by EA.

If the left-adjacent parcel does not specify a target General Purpose Register, or is a *Load* instruction, the instruction form is invalid.

Special Registers Altered:

None

Chapter 5. Fixed-Point Instructions

This chapter describes the Fixed-Point Instructions. Section 5.1 describes the registers accessible by fixed-point instructions, Section 5.2 describes general features associated with these instructions, whereas the remaining sections describe the corresponding instructions.

5.1 Registers

The set of registers accessible by Fixed-Point instructions consists of

- sixty-four General Purpose Registers (GPRs)
- sixty-four Floating-Point Registers (FPRs) (only for *Commit* instructions)
- the 64-bit Condition Register (CR)
- three Branch Registers (BRs)
- the 32-bit Fixed-Point Status Register (XSR)
- the 32-bit Floating-Point Status and Control Register
- the 64-bit GPR Delayed Exceptions Register (GRDX) and the 64-bit FPR Delayed Exceptions Register (FPDX)
- the 16-bit CR Delayed Exceptions Register (CRDX)

See Chapter 2., “Registers in the ForestaPC Architecture,” on page 21 for a complete description of these registers and their fields.

5.2 General Features

Most Fixed-Point instructions operate on data stored in General Purpose Registers and/or the Condition Register, and place the result(s) in these same registers. In addition, Fixed-Point instructions may set one bit in the GPR Delayed Exceptions Register, and one bit in the CR Delayed Exceptions Register.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

Floating-Point Registers may be the source or destination of *Commit* instructions. Special Purpose Registers may be the source or destination of *Move Special-Purpose Register* instructions, and other specific instructions.

Some Fixed-Point instructions specify a Condition Register Field field which is set depending on the result of the operation. Instructions which do not specify such a field may be augmented with an *Extend Immediate and Condition Register (xicr)* instruction in the right-adjacent slot (composing a two-word primitive); the *Extender* primitive specifies the Condition Register Field to be set by the instruction.

If the primitive instruction specifies a Condition Register Field, or if the instruction is augmented with a *xicr* primitive, the first three bits of the specified CR field are set to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to zero. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

Fixed-Point instructions generate a XSR-Image; when augmented with an *Extend XSR (xsrx)* primitive in the right-adjacent slot (composing a two-word primitive), the XSR-Image is placed in the GPR specified by the *xsrx* primitive.

Unless otherwise noted and when appropriate, when a CR field and the XRS-Image are set, they reflect the value placed into the target register.

Fixed-Point instructions are speculated without explicit indication; the programmer/compiler is in charge of keeping track of speculative results.

Any Fixed-Point instruction other than a *Commit* instruction using an operand whose associated bit in GRDX or CRDX is set to 1, sets to 1 the bit in GRDX and/or CRDX associated with the target register of the instruction; the contents of the target register (or register field) are undefined.

Any *Commit* instruction using an operand whose associated bit in GRDX, FPDX, or CRDX is set to 1, generates a delayed exception.

The description of instructions in this chapter does not include the setting of GRDX or CRDX; it is assumed that instructions follow the rules described above regarding these entities.

5.3 Branch Register Instructions

These instructions are used to place instruction addresses into the Branch Registers (for example, computing the return address before performing a procedure call or system call), or to copy the Branch Registers. The Branch Registers are also accessed with instructions *mtspr, mfspr.*

Compute Branch Register Immediate B2-form

cbri BRT,ADDR

0	4	6	30
10	BRT	ADDR	1

$BRT \leftarrow (CIA)_{0:37} \parallel ADDR \parallel 0b00$

The value $CIA_{0:37} \parallel$ ADDR $\parallel$ 0b00 is placed into Branch Register BRT.

Special Registers Altered:
BRT

XSR-Image Fields Generated:
None

Move Branch Register X10-form

mbr BRT,BRS

0	4	6	10	12	16	22
0	BRT	//	BRS	//	///	816

$BRT \leftarrow (BRS)$

The contents of Branch Register BRS are placed into Branch Register BRT.

Special Registers Altered:
Branch register BRT

XSR-Image Fields Generated:
None

5.4 Condition Register Logical Instructions

These instructions are used to perform logical operations on individual bits of the Condition Register. These instructions refer to the Condition Register as a register that contains 64 single bits, rather than as a register that contains 4-bit fields. This alternate view of the CR is denoted by CRB.

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics allows additional Condition Register logical operations, beyond those provided by the basic *Condition Register Logical* instructions, to be easily coded. Some of these are shown as examples with the *CR Logical* instructions.

Condition Register AND X10-form

crand BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	128

$CRB_{BT} \leftarrow CRB_{BA}$ & CRB_{BB}

The bit in the Condition Register specified by BA is ANDed with the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Condition Register OR X10-form

cror BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	133

$CRB_{BT} \leftarrow CRB_{BA} \mid CRB_{BB}$

The bit in the Condition Register specified by BA is ORed with the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:	*Equivalent to:*
crmove Bx,By	cror Bx,By,By

Condition Register XOR X10-form

crxor BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	135

$CRB_{BT} \leftarrow CRB_{BA} \oplus CRB_{BB}$

The bit in the Condition Register specified by BA is XORed with the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:	*Equivalent to:*
crclr Bx	crxor Bx,Bx,Bx

Condition Register NOR X10-form

crnor BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	132

$CRB_{BT} \leftarrow \neg(CRB_{BA} \mid CRB_{BB})$

The bit in the Condition Register specified by BA is ORed with the bit in the Condition Register specified by BB and the complemented result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:	*Equivalent to:*
crnot Bx,By	cror Bx,By,By

Condition Register NAND X10-form

crnand BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	131

$CRB_{BT} \leftarrow \neg(CRB_{BA} \ \& \ CRB_{BB})$

The bit in the Condition Register specified by BA is ANDed with the bit in the Condition Register specified by BB and the complemented result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Condition Register Equivalent X10-form

creqv BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	130

$CRB_{BT} \leftarrow CRB_{BA} \equiv CRB_{BB}$

The bit in the Condition Register specified by BA is XORed with the bit in the Condition Register specified by BB and the complemented result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:	*Equivalent to:*
crset Bx	creqv Bx,Bx,Bx

Condition Register AND with Complement X10-form

crandc BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	129

$CRB_{BT} \leftarrow CRB_{BA} \ \& \ \neg CRB_{BB}$

The bit in the Condition Register specified by BA is ANDed with the complement of the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:

CR

XSR-Image Fields Generated:

None

Condition Register OR with Complement X10-form

crorc BT,BA,BB

0	4	10	16	22
0	BT	BA	BB	134

$CRB_{BT} \leftarrow CRB_{BA} \ | \ \neg CRB_{BB}$

The bit in the Condition Register specified by BA is ORed with the complement of the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:

CR

XSR-Image Fields Generated:

None

5.5 Condition Register Field Instructions

These instructions are used to move data to/from fields of the Condition Register.

Move Condition Register Field X10-form

mcrf CRT,CRS

0	4	8	10	16	20	22
0	CRT	/	///	CRS	/	802

$CR_{CRT} \leftarrow CR_{CRS}$

The contents of Condition Register field CRS are copied into Condition Register field CRT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Move Condition Register Field Immediate X10-form

mcrfi CRT,CRI

0	4	8	10	16	20	22
0	CRT	/	///	CRI	/	803

$CR_{CRT} \leftarrow CRI$

The contents of the immediate field CRI are placed into Condition Register field CRT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Move to Condition Register Field X10-form

mtcrf CRT,RA

0	4	8	10	16	22
0	CRT	//	RA	///	801

$CR_{CRT} \leftarrow RA_{60:63}$

The contents of bits 60:63 from register RA are placed into Condition Register field CRT.

Special Registers Altered:
CR

XSR-Image Fields Generated:
None

Move From Condition Register Field X10-form

mfcrf RT,CRS

0	4	10	16	20	22
0	RT	///	CRS	/	800

$RT \leftarrow {}^{60}0 \parallel CR_{CRS}$

The contents of Condition Register field CRS are placed into bits 60:63 of register RT. $RT_{0:59}$ are set to 0.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

5.6 Condition Register Instructions

These instructions are used to move data to/from the Condition Register.

Move From Condition Register X10-form

mfcr RT

0	4	10	16	22
0	RT	//	///	835

RT ← CR

The contents of the Condition Register are placed into General Purpose Register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Move From Condition Register Word X10-form

mfcrw RT,L

0	4	10	15	16	22
0	RT	//	L	///	805

```
if L = 0 then
   RT ← 32 0 || CR8:15
else
   RT ← 32 0 || CR0:7
```

If L = 0, the contents of Condition Register Fields 8 through 15 are placed into the low-order 32 bits of register RT. If L=1, the contents of Condition Register Fields 0 through 7 are placed into the low-order 32 bits of register RT. The high-order 32 bits of register RT are set to zero.

Special Registers Altered:

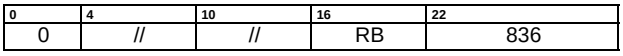
None

XSR-Image Fields Generated:

None

Move to Condition Register X10-form

mtcr RB



0	4	10	16	22
0	//	//	RB	836

CR ← (RB)

The contents of General-Purpose Register RB are placed into the Condition Register.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

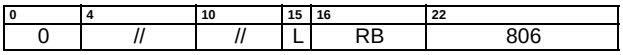
CR

XSR-Image Fields Generated:

None

Move to Condition Register Word X10-form

mtcrw L,RB



0	4	10	15	16	22
0	//	//	L	RB	806

```
if L = 0 then
  CR8:15 ← (RB)32:63
else
  CR0:7 ← (RB)32:63
```

If L = 0, the contents of bits 32:63 of register RB are placed into Condition Register Fields 8 through 15. If L = 1, the contents of bits 32:63 of register RB are placed into the Condition Register Fields 0 through 7.

Special Registers Altered:

CR

XSR-Image Fields Generated:

None

5.7 Extender Instructions

The *Extender* instructions are used to extend the capabilities of other primitive instructions. In particular, *Extender* instructions are used to

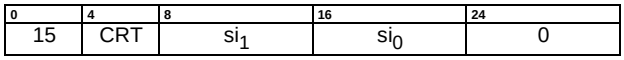
- generate 32-bits immediate fields;
- provide the ability to set a Condition Register field in instructions which do not specify a CR field;
- provide the ability to save a XSR-Image in a GPR;
- provide the ability to save a FSR-Image in a GPR;
- provide the ability to generate an exception based on the results from another fixed-point or floating-point operation;
- provide an additional operand to some fixed-point instructions.

Extender instructions transform a primitive instruction into a two-parcel primitive which executes in two adjacent slots in a VLIW. The right-most slot used by the multiparcel primitive contains the *Extender* instruction, whereas the slot to its left contains the instruction being extended.

Extended instructions (two-parcel instructions) are regarded as a single indivisible operation for the purposes of VLIW semantics and pruning. That is, the results from the operation consist of the results generated by the two-parcel instruction.

Extend Immediate and Condition Register I8-form

xicr CRT,SI



0	4	8	16	24
15	CRT	si_1	si_0	0

```
SI  ← si0 || si1
to_left_slot ← SI || CRT
```

This instruction provides additional fields to the left-adjacent execution slot.

If the instruction in the left-adjacent slot has a 16-bit immediate field, the 16-bit immediate value SI is appended to the 16-bit immediate value in the left-adjacent slot as the high-order bits, to produce a 32-bit immediate value which is used by the operation specified in the left-adjacent slot.

If the instruction in the left-adjacent slot does not specify a target CR field, the 4-bit value CRT is used to specify a CR field which is set according to the results of the operation in the left-adjacent slot.

This instruction can be paired only with instructions that have a 16-bit immediate field, with instructions that do not have a target CR field, or with *trap immediate* instructions; otherwise, the instruction form is invalid.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Programming Note: The *xicr* instruction is a mechanism for allowing an arithmetic instruction without a CRT field to target a Condition Register Field. When the *xicr* instruction is paired with an instruction which does not have an immediate field, the only purpose of the *xicr* instruction is to provide the target CR field (the SI field is ignored).

Extend XSR X10-form

xsrx RT,CRT,XM

0	4	10	16	20	22
0	RT	//	CRT	XM	771

```
if left_adj_inst is arith_fixed_point then
   to_left_slot ← CRT
   RT ← XSR-image from left_adj_inst
```

If the instruction in the left-adjacent slot is a *Fixed-Point Arithmetic* instruction, a *Fixed-Point Multiply/Divide* instruction, or a *Shift Right Algebraic* instruction, the Fixed-Point Status Image (XSR-Image) generated by the instruction in the left-adjacent slot is placed in register RT. Only the XSR bits specified by the XM mask are saved, as follows:

OV	if $XM_0 = 1$
CA	if $XM_1 = 1$

If the instruction in the left-adjacent slot does not specify a target CR field, the 4-bit value CRT is used to specify a CR field which is set according to the results of the operation in the left-adjacent slot.

If the instruction in left-adjacent slot is not a *Fixed-Point Arithmetic* instruction, a *Fixed-Point Multiply/Divide* instruction, or a *Shift Right Algebraic* instruction, the instruction form is invalid.

Special Registers Altered:

CR field CRT

XSR-Image Fields Generated:

None

Extended Extend XSR X10-form

xsrxe RT,CRT,RA,XM

0	4	10	16	20	22
0	RT	RA	CRT	XM	772

```
if left_adj_inst is arith_fixed_point then
   left_adj_RT ← left_adj_op + RA_CA
   to_left_slot ← CRT
   RT ← XSR-image from left_adj_inst
```

If the instruction in the left-adjacent slot is a *Fixed-Point Arithmetic* instruction, the CA bit from the XSR-Image in register RA is added to the result of that instruction. The final result is placed into the target register specified in the left-adjacent slot.

The Fixed-Point Status Image (XSR-Image) generated by the instruction in the left-adjacent slot is placed in register RT. Only the XSR bits specified by the XM mask are saved, as follows:

OV	if $XM_0 = 1$
CA	if $XM_1 = 1$

If the instruction in the left-adjacent slot does not specify a target CR field, the 4-bit value CRT is used to specify a CR field which is set according to the results of the operation in the left-adjacent slot.

If the instruction in left-adjacent slot is not a *Fixed-Point Arithmetic* instruction, the instruction form is invalid.

Special Registers Altered:

CR field CRT

XSR-Image Fields Generated:

None

Extend FSR X10-form

xfps RT,FM

0	4	10	16	22
0	RT	///	FM	770

```
if left_adj_inst is float_point then
   RT ← FSR-image from left_adj_inst
```

If the instruction in the left-adjacent slot is a *Floating-Point* instruction other than a *Floating-Point Move* instruction or a *Floating-Point Select* instruction, the Floating-Point Status Image (FSR-Image) generated by the instruction in the left-adjacent slot is placed in register RT. Only the FSR fields specified by the FM mask are saved, as follows:

FX OX	if $FM_0 = 1$
UX ZX XX VXSNAN	if $FM_1 = 1$
VXISI VXIDI VXZDZ VXIMZ	if $FM_2 = 1$
VXVC	if $FM_3 = 1$
VXSOFT VXSQRT VXCVI	if $FM_4 = 1$
FPRF FR FI	if $FM_5 = 1$

If the instruction in the left-adjacent slot is not a *Floating-Point* instruction other than a *Floating-Point Move* instruction or a *Floating-Point Select* instruction, the instruction form is invalid.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Extend XSR and Trap X10-form

xtx RT,CRT,XM

0	4	10	16	20	22
0	RT	///	CRT	XM	774

```
if left_adj_inst is arith_fixed_point then
   to_left_slot ← CRT
   RT ← XSR-image from left_adj_inst
   if XM0 & (XSR-imageOV from left_adj_op) |
      XM1 & (XSR-imageCA from left_adj_op)
   then TRAP
```

If the instruction in the left-adjacent slot is a *Fixed-Point Arithmetic* instruction, a *Fixed-Point Multiply/Divide* instruction, or a *Shift Right Algebraic* instruction, the Fixed-Point Status Image (XSR-Image) generated by the instruction in the left-adjacent slot is placed in register RT. Only the XSR bits specified by the XM mask are saved, as follows:

OV	if $XM_0 = 1$
CA	if $XM_1 = 1$

If the instruction in the left-adjacent slot does not specify a target CR field, the 4-bit value CRT is used to specify a CR field which is set according to the results of the operation in the left-adjacent slot.

If any bit in the XSR-Image generated by the instruction in the left-adjacent slot is set to 1, and the corresponding bit in the XM mask is 1, then the system trap handler is invoked.

If the instruction in the left-adjacent slot is not a *Fixed-Point Arithmetic* instruction, a *Fixed-Point Multiply/Divide* instruction, or a *Shift Right Algebraic* instruction, the instruction form is invalid.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Extend FSR and Trap X10-form

xtf RT,FM

0	4	10	16	22
0	RT	fm_0	fm_1	773

```
if left-adj-inst is arith-float-point
   RT ← FSR-image from left_adj_inst
   if FM0 & (FSR-imagexx from left-adj-op) |
      FM1 & (FSR-imagexx from left-adj-op) |

   then TRAP
```

If the instruction in the left-adjacent slot is a *Floating-Point Arithmetic* instruction, the Floating-Point Status Image (FSR-Image) generated by the instruction in the left-adjacent slot is placed in register RT. Only the FSR fields specified by the FM mask are saved, as follows:

FX OX	if $FM_0 = 1$
UX ZX XX VXSNAN	if $FM_1 = 1$
VXISI VXIDI VXZDZ VXIMZ	if $FM_2 = 1$
VXVC	if $FM_3 = 1$
VXSOFT VXSQRT VXCVI	if $FM_4 = 1$
FPRF FR FI	if $FM_5 = 1$

If any bit in the FSR-Image is set to 1, and the corresponding bit in the FM mask is 1, then the system trap handler is invoked.

If the instruction in the left-adjacent slot is not a *Floating-Point Arithmetic* instruction, the instruction form is invalid.

Special Registers Altered:

None

FSR-Image Fields Generated:

None

Extend Add X10-form

xadd RT,RA,XM

0	4	10	16	20	22
0	RT	RA	///	XM	768

```
if left_adj_inst is (arith_fixed_point or
                     logic_fixed_point) then
   left_adj_RT = left_adj_op + RA
   RT ← XSR-image from (left_adj_op + RA)
```

If the instruction in the left-adjacent slot is a *Fixed-Point Arithmetic* or *Fixed-Point Logical* instruction, the contents of register RA are added to the result of that instruction. The final result is placed into the target register specified in the left-adjacent slot.

The Fixed-Point Status Image (XSR-Image) generated by the operation in the left-adjacent slot is placed in register RT. Only the XSR bits specified by the XM mask are saved, as follows:

OV	if $XM_0 = 1$
CA	if $XM_1 = 1$

If the left-adjacent slot is not executing a *Fixed-Point Arithmetic* or *Fixed-Point Logical* instruction, the instruction form is invalid.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Extend Subtract X10-form

xsub RT,RA,XM

0	4	10	16	20	22
0	RT	RA	///	XM	769

```
if left_adj_inst is (arith_fixed_point or
                     logic_fixed_point) then
   left_adj_RT = left_adj_op - RA
   RT ← XSR-image from (left_adj_op - RA)
```

If the instruction in the left-adjacent slot is a *Fixed-Point Arithmetic* or *Fixed-Point Logical* instruction, the contents of register RA are subtracted from the result of that instruction. The final result is placed into the target register specified in the left-adjacent slot.

The Fixed-Point Status Image (XSR-Image) generated by the operation in the left-adjacent slot is placed in register RT. Only the XSR bits specified by the XM mask are saved, as follows:

OV	if $XM_0 = 1$
CA	if $XM_1 = 1$

If the left-adjacent slot is not executing a *Fixed-Point Arithmetic* or *Fixed-Point Logical* instruction, the instruction form is invalid.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

5.8 Fixed-Point Arithmetic Instructions

These instructions are used to perform addition and subtraction operations on data in the General Purpose Registers, placing the result in a GPR. In addition, these instructions may set a Condition Register Field.

Some *Fixed-Point Arithmetic* instructions do not specify a Condition Register Field to be set according to the result of the instruction. These instructions can be augmented with an *Extender* instruction, composing a two-parcel primitive; the *Extender* primitive specifies a Condition Register Field.

Fixed-Point Arithmetic instructions can be augmented with an *Extend XSR* instruction, composing a two-parcel primitive; the *Extender* primitive is used to place a Fixed-Point Status Image (XSR-Image) in a General Purpose Register. A mask field in the *Extend XSR* instruction indicates which bits of the Fixed-Point Status Image are saved.

If the *Extend XSR* instruction specifies the CA bit, that bit is set to reflect the carry out of bit 0 in 64-bit mode, and out of bit 32 in 32-bit mode.

If the *Extend XSR* instruction specifies the OV bit, that bit is set to reflect overflow of the result. The setting of this bit is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode, and overflow of the low-order 32-bit result in 32-bit mode.

If bits CA and OV are set differently, their setting is indicated with the specific instructions.

Programming Note: Notice that the CR field may not reflect the “true” (infinitely precise) result if overflow occurs.

Extended mnemonics for addition and subtraction

Extended mnemonics are provided that use the *Add Immediate* and *Add Byte Immediate* instructions to load an immediate value into a target register. Some of these are shown as examples with the corresponding primitive instructions.

Extended mnemonics are provided that use the *Extend XSR* instruction to implement the PowerPC architecture *Carrying*, *Overflow* and *Extended* form of *Add* and *Subtract* instructions. Some of these are shown as examples with the corresponding primitive instructions.

The ForestaPC architecture supplies *Subtract From* instructions, which subtract the second operand from the third. A set of extended mnemonics that uses the more “normal” order is provided, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate *Add* and *Subtract From* instructions.

Add X6-form

add RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	16

```
RT ← (RA) + (RB)
```

The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

CA, OV

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*				
addc	Rx,Rw,Ry,Rz	add	Rx,cr0,Ry,Rz	||	xsrx	Rw,cr0,1
addo	Rx,Rw,Ry,Rz	add	Rx,cr0,Ry,Rz	||	xsrx	Rw,cr0,2
addco	Rx,Rw,Ry,Rz	add	Rx,cr0,Ry,Rz	||	xsrx	Rw,cr0,3
adde	Rx,Rw,Ry,Rz,Rv	add	Rx,cr0,Ry,Rz	||	xsrxe	Rw,cr0,Rv,1
addeo	Rx,Rw,Ry,Rz,Rv	add	Rx,cr0,Ry,Rz	||	xsrxe	Rw,cr0,Rv,3

Add Immediate I0-form

addi RT,RA,SI

0	4	10	16
1	RT	RA	SI

```
if RA = 0 then RT ← EXTS(SI)
else           RT ← (RA) + EXTS(SI)
```

The sum (RA|0) + SI is placed into register RT.

Special Registers Altered:

None

XSR-Image Fields Generated:

CA, OV

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*				
li	Rx,value	addi	Rx,0,value			
la	Rx,displ(Ry)	addi	Rx,Ry,disp			
addis	Rx,Ry,value	addi	Rx,Ry,0	||	xicr	cr0,value
addic	Rx,Rw,Ry,value	addi	Rx,Ry,value	||	xrsx	Rw,cr0,1
subi	Rx,Ry,value	addi	Rx,Ry,-value			
addze	Rx,Rw,Ry,Rv	addi	Rx,Ry,0	||	xrsxe	Rw,cr0,Rv,1
addzeo	Rx,Rw,Ry,Rv	addi	Rx,Ry,0	||	xrsxe	Rw,cr0,Rv,3

Programming Note: *addi* uses the value 0, not the contents of GPR(0), if RA = 0.

Subtract From X6-form

subf RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	17

RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) + 1 is placed into register RT.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

CA,OV

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
sub	Rx,Ry,Rz	subf	Rx,cr0,Rz,Ry
subfc	Rx,Rw,Ry,Rz	subf	Rx,cr0,Ry,Rz|| xsrx Rw,cr0,1
subfo	Rx,Rw,Ry,Rz	subf	Rx,cr0,Ry,Rz|| xsrx Rw,cr0,2
subfco	Rx,Rw,Ry,Rz	subf	Rx,cr0,Ry,Rz|| xsrx Rw,cr0,3
subc	Rx,Rw,Ry,Rz	subf	Rx,cr0,Rz,Ry|| xsrx Rw,cr0,1

Subtract From Immediate I0-form

subfi RT,RA,SI

0	4	10	16
2	RT	RA	SI

RT ← ¬(RA) + EXTS(SI) + 1

The sum ¬(RA) + SI + 1 is placed into register RT.

Special Registers Altered:

None

XSR-Image Fields Generated:

CA,OV

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*			
subfic	Rx,Rw,Ry,value	subfi	Rx,Ry,value ||	xsrx	Rw,cr0,1
neg	Rx,Rw,Ry	subfi	Rx,Ry,0		
nego	Rx,Rw,Ry	subfi	Rx,Ry,0 ||	xsrx	Rw,cr0,2

Programming Note: In 64-bit mode, if register RA contains the most negative 64-bit number (0x8000_0000_0000_0000) and SI = 0, the result is the most negative number, and XSR-Image bit OV is set to 1. Similarly, in 32-bit mode if $(RA)_{32:63}$ contains the most negative number (0x8000_0000) and SI = 0, the low order 32 bits of the result contain the most negative 32-bit number, and XSR-Image bit OV is set to 1.

Programming Note:

The setting of XSR-Image bit CA by the *Add* and *Subtract* instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.

5.9 Fixed-Point Multiply and Divide Instructions

These instructions are used to perform multiplication and division operations on data in the General Purpose Registers, placing the result in a GPR.

Fixed-Point Multiply and Divide instructions do not specify a Condition Register Field to be set according to the result of the instruction. These instructions can be augmented with an *Extender* instruction, composing a two-parcel primitive; the *Extender* primitive specifies a Condition Register Field.

Fixed-Point Multiply and Divide instructions can be augmented with an *Extend XSR* instruction, composing a two-parcel primitive; the *Extender* primitive is used to place a Fixed-Point Status Image (XSR-Image) in a General Purpose Register. A mask field in the *Extend XSR* instruction indicates which bits of the Fixed-Point Status Image are saved.

If the *Extend XSR* instruction specifies the OV bit, that bit is set to reflect overflow of the result. The setting of this bit is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode, and overflow of the low-order 32-bit result in 32-bit mode.

If bit OV is set differently, its setting is indicated with the specific instructions.

.

Multiply Low Immediate I0-form

mulli RT,RA,SI

0	4	10	16
3	RT	RA	SI

$prod_{0:127} \leftarrow (RA) \times EXTS(SI)$
$RT \leftarrow prod_{64:127}$

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the SI field. The low-order 64-bits of the 128-bit product of the operands are placed into register RT.

Both the operands and the product are interpreted as signed integers.

Special Registers Altered:

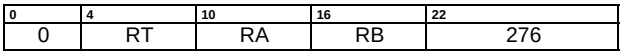
None

XSR-Image Fields Generated:

OV

Multiply Low Doubleword X10-form

mulld RT,RA,RB



0	4	10	16	22
0	RT	RA	RB	276

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{64:127}$

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

XSR-Image field OV is set to 1 if the product cannot be represented in 64 bits.

Both the operands and the product are interpreted as signed integers.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

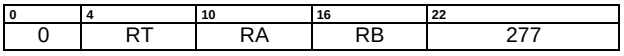
XSR-Image Fields Generated:
OV

Examples of Extended Mnemonics:

Extended:	*Equivalent to:*
mulldo Rx,Ry,Rz	mulld Rx,Ry,Rz || xsrx Rw,cr0,2

Multiply Low Word X10-form

mullw RT,RA,RB



0	4	10	16	22
0	RT	RA	RB	277

$RT \leftarrow (RA)_{32:63} \times (RB)_{32:63}$

The 32-bit operands are the low order 32-bits of (RA) and (RB). The 64-bit product of the operands is placed into register RT.

XSR-Image field OV is set to 1 if the product cannot be represented in 32 bits.

Both the operands and the product are interpreted as signed integers.

Special Registers Altered:
None

XSR-Image Fields Generated:
OV

Examples of Extended Mnemonics:

Extended:	*Equivalent to:*
mullwo Rx,Ry,Rz	mullw Rx,Ry,Rz || xsrxRw,cr0,2

Programming Note: For *mulli* and *mullw*, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For *mulli* and *mulld*, the low order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. For *mulli* and *mullw*, the low order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

Multiply High Doubleword X10-form

mulhd RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	272

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both the operands and the product are interpreted as signed integers.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

OV

Multiply High Doubleword Unsigned X10-form

mulhdu RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	273

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both the operands and the product are interpreted as unsigned integers.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

OV

Programming Note: If this instruction is extended with an *Extender* instruction specifying a CRT field, the first three bits of the CR field specified by the *Extender* are set by signed comparison of the result to zero.

Multiply High Word X10-form

mulhw RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	274

$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow undefined$

The 32-bit operands are the low order 32 bits of (RA) and (RB). The high-order 32 bits of the 64-bit product of the operands are placed into register $RT_{32:63}$. $RT_{0:31}$ are undefined.

Both the operands and the product are interpreted as signed integers.

Special Registers Altered:
None

XSR-Image Fields Generated:
OV

Multiply High Word Unsigned X10-form

mulhwu RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	275

$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow undefined$

The 32-bit operands are the low order 32 bits of (RA) and (RB). The high-order 32 bits of the 64-bit product of the operands are placed into register $RT_{32:63}$. $RT_{0:31}$ are undefined.

Both the operands and the product are interpreted as unsigned integers.

Special Registers Altered:
None

XSR-Image Fields Generated:
OV

Programming Note: If this instruction is extended with a *xicr* instruction, the first three bits of the CR field specified by the *Extender* are set by signed comparison of the result to zero.

Divide Doubleword X10-form

divd RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	192

$dividend_{0:63} \leftarrow$ (RA)
$divisor_{0:63} \leftarrow$ (RB)
RT $\leftarrow$ dividend ÷ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor is placed into register RT. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r \le |divisor|$ if the dividend is non-negative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

OV

Programming Note: If this instruction is extended with an *xicr* instruction, and the value placed in register RT by this instruction is undefined, the contents of bits LT, GT, and EQ in the CR field specified by the *Extender* are also undefined. If OV is specified, it is set to 1.

Programming Note: The 64-bit signed remainder of dividing (RA) by (RB) can be computed as follows, except in the case that (RA) = -2^{63} and (RB) = -1.

```
divd  RT,RA,RB   # RT = quotient
mulld RT,RT,RB   # RT = quotient*divisor
subf  RT,RT,RA    # RT = remainder
```

Divide Word X10-form

divw RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	194

$dividend_{0:63} \leftarrow EXTS((RA)_{32:63})$
$divisor_{0:63} \leftarrow EXTS((RB)_{32:63})$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow undefined$

The 64-bit dividend is the sign-extended value of $(RA)_{32:63}$. The 64-bit divisor is sign-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into register $RT_{32:63}$. $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r \le |divisor|$ if the dividend is non-negative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined.

Special Registers Altered:

None

XSR-Image Fields Generated:

OV

Programming Note: If this instruction is extended with an *xicr* instruction, and the value placed in register RT by this instruction is undefined, the contents of bits LT, GT, and EQ in the CR field specified by the *Extender* are also undefined. If OV is specified, it is set to 1.

Programming Note: The 32-bit signed remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows, except in the case that $(RA)_{32:63} = -2^{31}$ and $(RB)_{32:63} = -1$.

```
divw  RT,RA,RB   # RT = quotient
mullw RT,RT,RB   # RT = quotient*divisor
subf  RT,RT,RA    # RT = remainder
```

Divide Doubleword Unsigned X10-form

divdu RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	193

$dividend_{0:63} \leftarrow$ (RA)
$divisor_{0:63} \leftarrow$ (RB)
RT $\leftarrow$ dividend ÷ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor is placed into register RT. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as unsigned integers, except that if the instruction is extended with a *xicr* instruction, the first three bits of the specified CR field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r \le divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

OV

Programming Note: If this instruction is extended with an *xicr* instruction, and the value placed in register RT by this instruction is undefined, the contents of bits LT, GT, and EQ in the CR field specified by the *Extender* are also undefined. If OV is specified, it is set to 1.

Programming Note: The 64-bit unsigned remainder of dividing (RA) by (RB) can be computed as follows:

```
divdu RT,RA,RB   # RT = quotient
mulld RT,RT,RB   # RT = quotient*divisor
subf  RT,RT,RA    # RT = remainder
```

Divide Word Unsigned X10-form

divwu RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	195

$\texttt{dividend}_{0:63} \leftarrow {}^{32}\texttt{0} \parallel (\texttt{RA})_{32:63}$
$\texttt{divisor}_{0:63} \leftarrow {}^{32}\texttt{0} \parallel (\texttt{RB})_{32:63}$
$\texttt{RT}_{32:63} \leftarrow \texttt{dividend} \div \texttt{divisor}$
$\texttt{RT}_{0:31} \leftarrow \texttt{undefined}$

The 64-bit dividend is the zero-extended value of $(RA)_{32:63}$. The 64-bit divisor is zero-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into register $RT_{32:63}$. $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as unsigned integers, except that if the instruction is extended with a *xicr* instruction, the first three bits of the specified CR field are set by signed comparison of the result to zero. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r \le divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined.

Special Registers Altered:

None

XSR-Image Fields Generated:

OV

Programming Note: If this instruction is extended with an *xicr* instruction, and the value placed in register RT by this instruction is undefined, the contents of bits LT, GT, and EQ in the CR field specified by the *Extender* are also undefined. If OV is specified, it is set to 1.

Programming Note: The 32-bit unsigned remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows.

```
divwu RT,RA,RB   # RT = quotient
mullw RT,RT,RB   # RT = quotient*divisor
subf  RT,RT,RA   # RT = remainder
```

5.10 Fixed-Point Compare Instructions

The *Fixed-Point Compare* instructions compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of register RB. The comparison is signed for *cmpi* and *cmp*, and unsigned for *cmpli* and *cmpl*.

For 64-bit implementations, the L field controls whether the operands are treated as 64- or 32-bit quantities, as follows:

L	Operand length
0	32-bit operands
1	64-bit operands

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

For 32-bit implementations, the L field must be zero.

The *Compare* instructions set one bit in the left-most three bits of the designated CR field to one, and the other two to zero. Bit 3 of the designated CR field is set to 0.

The CR field is set as follows:

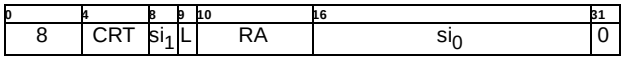
Bit	Name	Description
0	LT	(RA) < SI or (RB) (signed comparison) (RA) $<^{u}$ UI or (RB) (unsigned comparison)
1	GT	(RA) > SI or (RB) (signed comparison) (RA) $>^{u}$ UI or (RB) (unsigned comparison)
2	EQ	(RA) = SI, UI or (RB)
3		Set to 0.

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the instruction mnemonics rather than as a numeric operand. Some of these are shown as examples with the *Compare* instructions. The extended mnemonics for doubleword comparisons are available only in 64-bit implementations.

Compare Immediate I1-form

cmpi CRT,L,RA,SI




```
SI  ← si0 || si1
if L = 0 then a ← EXTS((RA)32:63)
         else a ← (RA)
if      a < EXTS(SI) then c ← 0b1000
else if a > EXTS(SI) then c ← 0b0100
else                      c ← 0b0010
CRCRT    ← c
```

The contents of register RA ($(RA)_{32:63}$ sign-extended to 64 bits if L=0) are compared with the sign-extended value of the SI field, treating the operands as signed integers.The result of the comparison is placed into CR field CRT.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

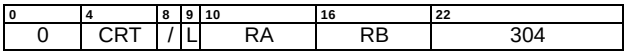
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
cmpdi	Rx,value	cmpi	cr0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	cr3,0,Rx,value

Compare X10-form

cmp CRT,L,RA,RB




```
if L = 0 then a ← EXTS((RA)32:63)
              b ← EXTS((RB)32:63)
         else a ← (RA)
              b ← (RB)
if      a < b then c ← 0b1000
else if a > b then c ← 0b0100
else               c ← 0b0010
CRCRT     ← c
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as signed integers. The result of the comparison is placed into CR field CRT.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

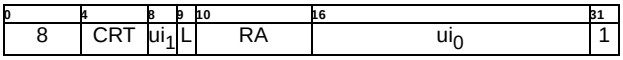
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

Compare Logical Immediate I1-form

cmpli CRT,L,RA,UI




```
UI  ← ui0 ‖ ui1
if L = 0 then a ← 320 ‖ (RA)32:63
         else a ← (RA)
if      a <u (480 ‖ UI) then c ← 0b1000
else if a >u (480 ‖ UI) then c ← 0b0100
else                         c ← 0b0010
CRCRT     ← c
```

The contents of register RA ($(RA)_{32:63}$ zero-extended to 64 bits if L=0) are compared with $^{48}0 \,\|\, UI$, treating the operands as unsigned integers. The result of the comparison is placed into CR field CRT.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
cmpldi	Rx,value	cmpli	0,1,Rx,value
cmplwi	cr3,Rx,value	cmpli	3,0,Rx,value

Compare Logical X10-form

cmpl CRT,L,RA,RB

0	4	8	9	10	16	22
0	CRT	/	L	RA	RB	305

```
if L = 0 then a ← 32 0 || (RA)32:63
               b ← 32 0 || (RB)32:63
         else a ← (RA)
              b ← (RB)
if      a <u b then c ← 0b1000
else if a >u b then c ← 0b0100
else                  c ← 0b0010
CRCRT     ← c
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as unsigned integers. The result of the comparison is placed into CR field CRT.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
cmpld	Rx,Ry	cmpl	0,1,Rx,Ry
cmplw	cr3,Rx,Ry	cmpl	3,0,Rx,Ry

5.11 Fixed-Point Trap Instructions

The *Trap* instructions are provided to test for a specified set of conditions. If any of the conditions tested by a *Trap* instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally.

The instructions *tdi* and *twi* must be used together with instruction *xicr* as a pair, executing in adjacent slots. The instruction *xicr* in the slot to the right specifies a 16-bit immediate value which is used by the *tdi* or *twi* instruction in the slot to the left.

The contents of register RA are compared, depending on the *Trap* instruction, either with the contents of register RB or the sign-extended value of the SI field specified by a right-adjacent *xicr* instruction. For *tdi* and *td*, the entire contents of RA (and RB) participate in the comparison. For *twi* and *tw*, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

The comparison results in five conditions which are ANDed with TO. If the result is not 0, the system trap handler is invoked. The comparison functions consist of one or more of the following conditions:

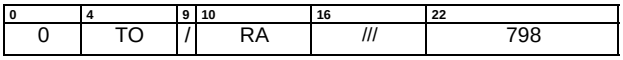
TO bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the instruction mnemonics rather than as a numeric operand. Some of these are shown as examples with the *Trap* instructions.

Trap Doubleword Immediate X10-form

tdi TO,RA



0	4	9	10	16	22
0	TO	/	RA	///	798

```
SI  ← from_right_parcel
a ← (RA)
b ← EXTS(SI)
if (a < b)  & TO0 then TRAP
if (a > b)  & TO1 then TRAP
if (a = b)  & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

tdi and *xicr* are used always as a parcel-pair in adjacent slots.

The contents of register RA are compared with the sign-extended value received from the *xicr* instruction executing in the right-adjacent parcel.

If any bit in the TO field is set to 1 and the corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

If the instruction in the right-adjacent parcel is not *xicr*, the instruction form is invalid.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

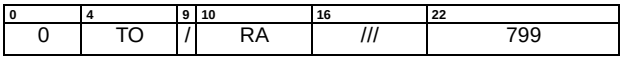
None

XSR-Image Fields Generated:

None

Trap Word Immediate X10-form

twi TO,RA



0	4	9	10	16	22
0	TO	/	RA	///	799

```
SI  ← from_right_parcel
a ← EXTS((RA)32:63)
b ← EXTS(SI)
if (a < b)  & TO0 then TRAP
if (a > b)  & TO1 then TRAP
if (a = b)  & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

twi and *xicr* are used always as a parcel-pair in adjacent slots.

The contents of $RA_{32:63}$ are compared with the sign-extended value received from the *xicr* instruction executing in the right-adjacent parcel.

If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

If the instruction in the right-adjacent parcel is not *xicr*, the instruction form is invalid.

Special Registers Altered:

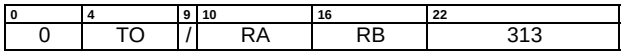
None

XSR-Image Fields Generated:

None

Trap Doubleword X10-form

td TO,RA,RB



0	4	9	10	16	22
0	TO	/	RA	RB	313

```
a ← (RA)
b ← (RB)
if (a < b)  & TO0 then TRAP
if (a > b)  & TO1 then TRAP
if (a = b)  & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and the corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

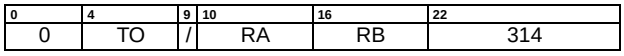
None

XSR-Image Fields Generated:

None

Trap Word X10-form

tw TO,RA,RB



0	4	9	10	16	22
0	TO	/	RA	RB	314

```
a ← (RA)32:63
b ← (RB)32:63
if (a < b)  & TO0 then TRAP
if (a > b)  & TO1 then TRAP
if (a = b)  & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of $RA_{32:63}$ are compared with the contents of $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

5.12 Fixed-Point Select Instructions

The *Fixed-Point Select* instructions set a target register to one of two values, according to the value of a specified bit in the Condition Register. Any bit in the 64-bit CR may be tested. These instructions treat the Condition Register as a register that contains 64 independently addressable bits, denoted by CRB.

Programming Note: The *Select* instructions are intended to be used to improve program execution speed by reducing branching. For example, they can be used, often after a *Compare* instruction, to implement the fixed-point minimum, maximum, and absolute value functions, to obtain 0/1 or 0/-1 values for relational expressions, and to implement certain simple forms of C conditional expressions and if-then-else constructs.

Extended mnemonics for selects

A set of extended mnemonics is provided so that selects can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Select* instructions.

Select Immediate-Immediate X4-form

selii RT,IA,IB,CB

0	4	10	16	22	28
12	RT	IA	IB	CB	8

```
if CRB_CB then   RT ← EXTS(IA)
else             RT ← EXTS(IB)
```

The Condition Register bit at position CB is tested. If it is 1, register RT is set to the sign-extended value of IA. Otherwise, register RT is set to the sign-extended value of IB.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
seleqii	Rx,valy,valz	selii	Rx,valy,valz,2

Select Immediate-Register X4-form

selir RT,IA,RB,CB

0	4	10	16	22	28
12	RT	IA	RB	CB	9

```
if CRB_CB then   RT ← EXTS(IA)
else             RT ← (RB)
```

The Condition Register bit at position CB is tested. If it is 1, register RT is set to the sign-extended value of IA. Otherwise, register RT is set to (RB).

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
selltir	Rx,valy,Rz	selir	Rx,valy,Rz,0

Select Register-Immediate X4-form

selri RT,RA,IB,CB

0	4	10	16	22	28
12	RT	RA	IB	CB	10

```
if CRB_CB then   RT ← (RA)
else             RT ← EXTS(IB)
```

The Condition Register bit at position CB is tested. If it is 1, register RT is set to (RA). Otherwise, register RT is set to the sign-extended value of IB.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
selgtri	Rx,Ry,valz	selri	Rx,Ry,valz,1

Select Register-Register X4-form

selrr RT,RA,RB,CB

0	4	10	16	22	28
12	RT	RA	RB	CB	11

```
if CRB_CB then   RT ← (RA)
else             RT ← (RB)
```

The Condition Register bit at position CB is tested. If it is 1, register RT is set to (RA). Otherwise, register RT is set to (RB).

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
selovrr	Rx,Ry,Rz	selrr	Rx,Ry,Rz,3

5.13 Fixed-Point Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The *Logical Immediate* instructions do not specify a Condition Register field to be set as part of the instruction. The *Logical Immediate* instructions can be augmented with an *Extender* instruction, in the right adjacent parcel, specifying a CR field.

The first three bits of the specified CR field are set to characterize the result of the logical operation. The CR field is set as if the result of the operation was algebraically compared to zero.

Extended mnemonics for logical operations

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

AND Immediate I0-form

andi RT,RA,UI

0	4	10	16
4	RT	RA	UI

```
RT ← (RA) & (48_0 || UI)
```

The contents of register RA are ANDed with $^{48}0$ || UI, and the result is placed into register RT.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

OR Immediate I0-form

ori RT,RA,UI

0	4	10	16
5	RT	RA	UI

RT ← (RA) | (^{48}0 || UI)

The contents of register RA are ORed with ^{48}0 || UI, and the result is placed into register RT.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

XOR Immediate I0-form

xori RT,RA,UI

0	4	10	16
6	RT	RA	UI

RT ← (RA) xor (^{48}0 || UI)

The contents of register RA are XORed with ^{48}0 || UI, and the result is placed into register RT.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

AND X6-form

and RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	8

RT ← (RA) & (RB)

The contents of register RA are ANDed with the contents of register RB, and the result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

OR X6-form

or RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	12

RT ← (RA) | (RB)

The contents of register RA are ORed with the contents of register RB, and the result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:
Example of extended mnemonics for *OR*:

Extended:		*Equivalent to:*	
mr	Rx,Ry	or	Rx,Ry,Ry

XOR X6-form

xor RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	9

RT ← (RA) ⊕ (RB)

The contents of register RA are XORed with the contents of register RB, and the result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

NAND X6-form

nand RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	10

RT ← ¬((RA) & (RB))

The contents of register RA are ANDed with the contents of register RB, and the complemented result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Programming Note: *nand* or *nor* with RA=RB can be used to obtain one's complement.

NOR X6-form

nor RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	11

RT ← ¬((RA) | (RB))

The contents of register RA are ORed with the contents of register RB, and the complemented result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
not	Rx,Ry	nor	Rx,Ry,Ry.

Equivalent X6-form

eqv RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	15

RT ← (RA) ≡ (RB)

The contents of register RA are XORed with the contents of register RB, and the complemented result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

AND with Complement X6-form

andc RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	13

RT ← (RA) & ¬(RB)

The contents of register RA are ANDed with the complement of the contents of register RB, and the result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

OR with Complement X6-form

orc RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	14

RT ← (RA) | ¬(RB)

The contents of register RA are ORed with the complement of the contents of register RB, and the result is placed into register RT.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Extend Sign Byte X10-form

extsb RT,CRT,RA

0	4	10	16	20	22
0	RT	RA	CRT	//	308

$s \leftarrow (RA)_{56}$
$RT_{56:63} \leftarrow (RA)_{56:63}$
$RT_{0:55} \leftarrow {}^{56}s$

Bits $(RA)_{56:63}$ are placed into $RT_{56:63}$. Bit 56 of register RA is placed into $RT_{0:55}$.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Extend Sign Halfword X10-form

extsh RT,CRT,RA

0	4	10	16	20	22
0	RT	RA	CRT	//	309

$s \leftarrow (RA)_{48}$
$RT_{48:63} \leftarrow (RA)_{48:63}$
$RT_{0:47} \leftarrow {}^{48}s$

Bits $(RA)_{48:63}$ are placed into $RT_{48:63}$. Bit 48 of register RA is placed into RT0:47.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Extend Sign Word X10-form

extsw RT,CRT,RA

0	4	10	16	20	22
0	RT	RA	CRT	//	310

$s \leftarrow (RS)_{32}$
$RT_{32:63} \leftarrow (RA)_{32:63}$
$RT_{0:31} \leftarrow {}^{32}s$

Bits $(RA)_{32:63}$ are placed into $RT_{32:63}$. Bit 32 of register RA is placed into $RT_{0:31}$.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

No-operation X10-form

nop

0	4	10	16	22
0	///	///	///	1

This instruction does not modify any registers or affect any facilities. It is intended to fill unused words in a tree-instruction, if any, or to fill unused storage locations within a program.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Programming Note: Instruction *Nop* is used to fill gaps in tree-instructions that can arise from implementation constraints. Such constraints are described in *Book IV, ForestaPC Implementation Features* for a specific implementation.

Count Leading Zeros Doubleword X10-form

cntlzd RT,CRT,RA

0	4	10	16	20	22
0	RT	RA	CRT	//	306

```
n ← 0
do while n < 64
   if (RA)n = 1 then leave
   n ← n + 1
end
RT ← n
```

A count of the number of consecutive zero bits starting at bit 0 of register RA is placed into RT. This number ranges from 0 to 64, inclusive.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Count Leading Zeros Word X10-form

cntlzw RT,CRT,RA

0	4	10	16	20	22
0	RT	RA	CRT	//	307

```
n ← 32
do while n < 64
   if (RS)n = 1 then leave
   n ← n + 1
end
RT ← n - 32
```

A count of the number of consecutive zero bits starting at bit 32 of register RA is placed into RT. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
CR field CRT

XSR-Image Fields Generated:
None

Programming Note: For both *Count Leading Zeros* instructions, LT is set to zero in CR field CRT.

5.14 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Rotate instructions perform rotation operations on data from a GPR and return the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported:

- $rotate_{64}$ or $ROTL_{64}$, wherein the value rotated is the given 64-bit value. The $rotate_{64}$ operation is used to rotate a given 64-bit quantity.
- $rotate_{32}$ or $ROTL_{32}$, wherein the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The $rotate_{32}$ operation is used to rotate a given 32-bit quantity.

The *Rotate* and *Shift* instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, *mstart*, through and including a stop bit, *mstop*, and 0-bits elsewhere. The values of *mstart* and *mstop* range from zero to 63. If *mstart* > *mstop*, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
        maskm_start:mstop = ones
        mask_all other bits = zeros
else
        mask_mstart:63 = ones
        mask_0:mstop = ones
        mask_all other bits = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the $rotate_{32}$ operation, the mask start and stop positions are always in the low-order 32-bits of the register.

The use of the mask is described in the following sections.

The *Rotate* instructions do not specify a Condition Register field to be set as part of the instruction. The *Rotate* instructions can be augmented with an *Extend Immediate and Condition Register* instruction, in the right adjacent parcel, specifying a CR field. On the other hand, *Shift* instructions specify a Condition Register field to be set as part of the instruction. In all cases, the CR field is set as described in Section 2.3.3, “Condition Register,” on page 22. *Rotate* and *Shift* instructions do not generate bit OV in the XSR-Image. Moreover, *Rotate* and *Shift* instructions, excepting algebraic right shifts, do not generate bit CA in the XSR-Image.

Extended mnemonics for rotates and shifts

The *Rotate* and *Shift* instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allows simpler coding of often-used functions, such as clearing the left-most or right-most bits of a register, left justifying or right-justifying an arbitrary field, and simple rotates and shifts. Some of these are shown as examples with the *Rotate* instructions.

5.14.1 Fixed-Point Rotate Instructions

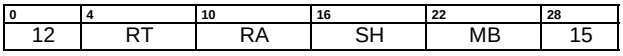
These instructions rotate the contents of a register. The result of the rotation is ANDed with a mask before being placed into the target register.

The *Rotate Left* instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64-N, where N is the number of bits by which to rotate right. These instructions allow performing right-rotation of the contents of the low-order half of a register (in concept) by a left-rotation of 32-N, where N is the number of bits by which to rotate right.

Programming Note: The PowerPC *rldimi* and *rlwimi* instructions have been dropped from the ForestaPC Architecture; their functionality is obtained from a sequence of primitive instructions.

Rotate Left Doubleword Immediate then Clear Left X4-form

rldicl RT,RA,SH,MB



0	4	10	16	22	28
12	RT	RA	SH	MB	15

```
n  ← SH
r  ← ROTL64((RA),n)
b  ← MB
m  ← MASK(b, 63)
RT ← r & m
```

The contents of register RA are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

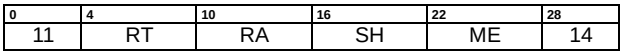
Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
extrdi	Rx,Ry,n,b	rldicl	Rx,Ry,b+n,64-n
srdi	Rx,Ry,n	rldicl	Rx,Ry,64-n,n
clrldi	Rx,Ry,n	rldicl	Rx,Ry,0,n

Programming Note: *rldicl* can be used to extract an *n*-bit field, which starts at bit position *b* in register RA, right-justified into register RT (clearing the remaining 64-*n* bits of RT), by setting SH=*b+n* and MB=64-*n*. It can be used to rotate the contents of a register left (right) by *n* bits, by setting SH=*n* (64-*n*) and MB=0. It can be used to shift the contents of a register right by *n* bits, by setting SH=64-*n* and MB=*n*. It can be used to clear the high-order *n* bits of a register, by setting SH=0 and MB=*n*. Extended mnemonics are provided for all of these uses.

Rotate Left Doubleword Immediate then Clear Right X4-form

rldicr RT,RA,SH,ME



0	4	10	16	22	28
11	RT	RA	SH	ME	14

```
n  ← SH
r  ← ROTL64((RA), n)
e  ← ME
m  ← MASK(0, e)
RT ← r & m
```

The contents of register RA are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

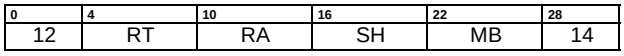
Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
extldi	Rx,Ry,n,b	rldicr	Rx,Ry,b,n-1
sldi	Rx,Ry,n	rldicr	Rx,Ry,n,63-n
clrrdi	Rx,Ry,n	rldicr	Rx,Ry,0,63-n

Programming Note: *rldicr* can be used to extract an *n*-bit field, which starts at bit position *b* in register RA, left-justified into register RT (clearing the remaining 64-*n* bits of RT), by setting SH=*b* and ME=*n*-1. It can be used to rotate the contents of a register left (right) by *n* bits, by setting SH=*n* (64-*n*) and ME=63. It can be used to shift the contents of a register left by *n* bits, by setting SH=*n* and ME=63-*n*. It can be used to clear the low-order *n* bits of a register, by setting SH=0 and ME=63-*n*. Extended mnemonics are provided for all of these uses.

Rotate Left Doubleword Immediate then Clear X4-form

rldic RT,RA,SH,MB




```
n  ← SH
r  ← ROTL64((RA),n)
b  ← MB
m  ← MASK(b,¬n)
RT ← r & m
```

The contents of register RA are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH, and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

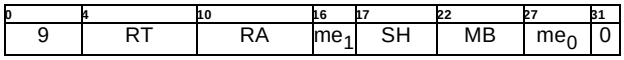
Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
clrlsldi	Rx,Ry,b,n	rldic	Rx,Ry,n,b-n

Programming Note: *rldic* can be used to clear the high-order *b* bits of the contents of a register, and then shift the result left by *n* bits by setting SH=*n* and MB=*b-n*. It can be used to clear the high-order *n* bits of a register, by setting SH=0 and MB=*n*. Extended mnemonics are provided for all of these uses.

Rotate Left Word Immediate then AND with Mask M1-form

rlwinm RT,RA,SH,MB,ME




```
ME ← me0 ‖ me1
n  ← SH
r  ← ROTL32((RA)32:63,n)
m  ← MASK(MB+32,ME+32)
RT ← r & m
```

The contents of register RA are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
extlwi	Rx,Ry,n,b	rlwinm	Rx,Ry,b,0,n-1
srwi	Rx,Ry,n	rlwinm	Rx,Ry,32-n,n,31
slwi	Rx,Ry,n	rlwinm	Rx,Ry,n,0,31-n
clrrwi	Rx,Ry,n	rlwinm	Rx,Ry,0,0,31-n

Programming Note: Let RAL represent the low-order half of register RA, with the bits numbered from 0 through 31.

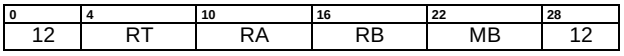
rlwinm can be used to extract an *n*-bit field, which starts at bit position *b* in RAL, right-justified into the low-order half of register RT (clearing the remaining 32-*n* bits of the low-order half of RT), by setting SH=*b*+*n*, MB=32-*n*., and ME=31. It can be used to extract an *n*-bit field, that starts at bit position *b* in RAL, left-justified into the low-order half of register RT (clearing the remaining 32-*n* bits of the low-order half of RT), by setting SH=*b*, MB = 0, and ME=*n*-1. It can be used to rotate the contents of the low-order half of a register left (right) by *n* bits, by setting SH=*n* (32-*n*), MB=0, and ME=31. It can be used to shift the contents of the low-order half of a register right by *n* bits, by setting SH=32-*n*, MB=*n*, and ME=31. It can be used to clear the high-order *b* bits of the low-order half

of a register, and then shift the result left by *n* bits, by setting SH=*n*, MB=*b*-*n*, and ME=31-*n*. It can be used to clear the low-order *n* bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31-*n*.

For all the uses given above, the high-order 32 bits of register RT are cleared.

Rotate Left Doubleword then Clear Left X4-form

rldcl RT,RA,RB,MB




```
n  ← (RB)58:63
r  ← ROTL64((RA),n)
b  ← MB
m  ← MASK(b,63)
RT ← r & m
```

The contents of register RA are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

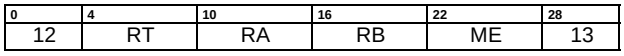
Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
rotld	Rx,Ry,Rz	rldcl	Rx,Ry,Rz,0

Programming Note: *rldcl* can be used to extract an *n*-bit field, which starts at variable bit position *b* in register RA, right-justified into register RT (clearing the remaining 64-*n* bits of RT), by setting RB$_{58:63}$=*b*+*n* and MB=64-*n*. It can be used to rotate the contents of a register left (right) by variable *n* bits by setting RB$_{58:63}$=*n* (64-*n*) and MB=0. Extended mnemonics are provided for all of these uses.

Rotate Left Doubleword then Clear Right X4-form

rldcr RT,RA,RB,ME



0	4	10	16	22	28
12	RT	RA	RB	ME	13

```
n  ← (RB)58:63
r  ← ROTL64((RA),n)
e  ← ME
m  ← MASK(0, e)
RT ← r & m
```

The contents of register RA are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

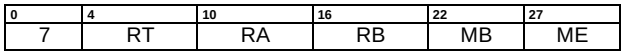
XSR-Image Fields Generated:

None

Programming Note: *rldcr* can be used to extract an *n* field, which starts at variable bit position *b* in register RA, left-justified into register RT (clearing the remaining 64-*n* bits of RT), by setting RB$_{58:63}$=*b* and ME=*n*-1. It can be used to rotate the contents of a register left (right) by variable *n* bits by setting RB$_{58:63}$=*n* (64-*n*) and ME=63. Extended mnemonics are provided for all of these uses.

Rotate Left Word then AND with Mask M0-form

rlwnm RT,RA,RB,MB,ME



0	4	10	16	22	27
7	RT	RA	RB	MB	ME

```
n  ← (RB)59:63
r  ← ROTL32((RA)32:63,n)
m  ← MASK(MB+32,ME+32)
RT ← r & m
```

The contents of register RA are rotated$_{32}$ left the number of bits specified by (RB)$_{59:63}$. A mask is generated having 1-bits from bit MB through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RT.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
rotlw	Rx,Ry,Rz	rlwnm	Rx,Ry,Rz,0,31

Programming Note: Let RAL represent the low-order half of register RA, with the bits numbered from 0 through 31.

rlwnm can be used to extract an *n*-bit field, which starts at variable bit position *b* in RAL, right-justified into the low-order half of register RT (clearing the remaining 32-*n* bits of the low-order 32 bits of RT), by setting RB$_{59:63}$=*b+n*, MB=32-*n*, and ME=31. It can be used to extract an *n*-bit field, which starts at variable bit position *b* in RAL, left-justified into the low-order half of register RT (clearing the remaining 32-*n* bits of the low-order half of RT), by setting RB$_{59:63}$=*b*, MB = 0, and ME=*n*-1. It can be used to rotate the contents of the low-order half of a register left (right) by variable *n* bits, by setting RB$_{59:63}$=*n* (32-*n*), MB=0, and ME=31.

For all the uses given above, the high-order half of register RT is cleared.

Extended mnemonics are provided for all of these uses.

5.14.2 Fixed-Point Shift Instructions

These instructions perform left and right shift of the contents of a register.

Extended mnemonics for shifts

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain *Rotate* instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand, as well as simple rotates and shifts. Some of these are shown as examples with the instructions.

Programming Note: Any *Shift Right Algebraic* instruction, followed by *addze*, can be used to divide quickly by 2^N. The setting of the CA bit by the *Shift Right Algebraic* instructions is independent of mode.

Engineering Note: The instructions intended for use with 32-bit data are shown as doing a rotate_{32} operation. This is strictly necessary only for setting the CA bit for *srawi* and *sraw*. *slw* and *srw* could do a rotate_{64} operation if that is easier.

Shift Left Doubleword X6-form

sld RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	6

```
n ← (RB)58:63
r ← ROTL64((RA),n)
if (RB)57 = 0 then
   m ← MASK(0,63-n)
else
   m ← 64 0
RT ← r & m
```

The contents of register RA are shifted left the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RT. Shift amounts from 64 to 127 give a zero result.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

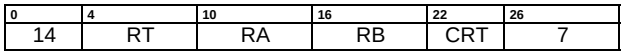
CR Field CRT

XSR-Image Fields Generated:

None

Shift Left Word X6-form

slw RT,CRT,RA,RB



0	4	10	16	22	26
14	RT	RA	RB	CRT	7

```
n ← (RB)59:63
r ← ROTL32((RA)32:63,n)
if (RB)58 = 0 then
   m ← MASK(32,63-n)
else
   m ← 640
RT ← r & m
```

The contents of the low-order 32 bits of register RA are shifted left the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RT_{32:63}$. $RT_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

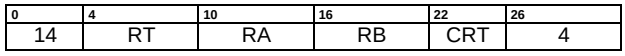
CR Field CRT

XSR-Image Fields Generated:

None

Shift Right Doubleword X6-form

srd RT,CRT,RA,RB



0	4	10	16	22	26
14	RT	RA	RB	CRT	4

```
n ← (RB)58:63
r ← ROTL64((RA),64-n)
if (RB)57 = 0 then
   m ← MASK(n,63)
else
   m ← 640
RT ← r & m
```

The contents of register RA are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into register RT. Shift amounts from 64 to 127 give a zero result.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

None

Shift Right Word X6-form

srw RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	5

```
n ← (RB)59:63
r ← ROTL32((RA)32:63,64-n)
if (RB)58 = 0 then
   m ← MASK(n+32,63)
else
   m ← 64 0
RT ← r & m
```

The contents of the low-order 32 bits of register RA are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RT_{32:63}$. $RT_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

None

Shift Right Algebraic Doubleword Immediate X6-form

sradi RT,CRT,RA,SH

0	4	10	16	22	26
14	RT	RA	SH	CRT	0

```
n  ← SH
r  ← ROTL64((RA),64-n)
m  ← MASK(n,63)
s  ← (RA)0
RT ← r&m | 64s&¬m
```

The contents of register RA are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RA is replicated to fill the vacated positions on the left. The result is placed into register RT. A shift amount of zero causes RT to be set equal to (RA).

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

CA

Programming Note: XSR-Image field CA is set to 1 if (RA) is negative and any 1-bits are shifted out of position 63; otherwise $XSR\text{-}Image_{CA}$ is set to 0.

A shift amount of zero causes $XSR\text{-}Image_{CA}$ to be set to 0.

Shift Right Algebraic Word Immediate X6-form

srawi RT,CRT,RA,SH

0	4	10	16	22	26
14	RT	RA	SH	CRT	1

```
n  ← SH
r  ← ROTL32((RA)32:63,64-n)
m  ← MASK(n+32,63)
s  ← (RA)32
RT ← r&m | 64s&¬m
```

The contents of the low-order 32 bits of register RA are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 32 of RA is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RT_{32:63}$. Bit 32 of RA is replicated to fill $RT_{0:31}$. A shift amount of zero causes $RT_{32:63}$. to receive EXTS($(RA)_{32:63}$).

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

CA

Programming Note: XSR-Image field CA is set to 1 if (RA) is negative and any 1-bits are shifted out of position 63; otherwise XSR-Image$_{CA}$ is set to 0.

A shift amount of zero causes XSR-Image$_{CA}$ to be set to 0.

Shift Right Algebraic Doubleword X6-form

srad RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	2

```
n  ← (RB)58:63
r  ← ROTL64((RA),64-n)
if (RB)57 = 0 then
   m ← MASK(n,63)
else
   m ← 640
s  ← (RA)0
RT ← r&m | 64s&¬m
```

The contents of register RA are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Bit 0 of RA is replicated to fill the vacated positions on the left. The result is placed into register RT. A shift amount of zero causes RT to be set equal to (RA). Shift amounts from 64 to 127 give a result of 64 sign bits in RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

CA

Programming Note: XSR-Image field CA is set to 1 if (RA) is negative and any 1-bits are shifted out of position 63; otherwise XSR-Image$_{CA}$ is set to 0.

A shift amount of zero causes XSR-Image$_{CA}$ to be set to 0.

Shift Right Algebraic Word X6-form

sraw RT,CRT,RA,RB

0	4	10	16	22	26
14	RT	RA	RB	CRT	3

```
n ← (RB)59:63
r ← ROTL32((RA)32:63,64-n)
if (RB)58 = 0 then
   m ← MASK(n+32,63)
else
   m ← 64 0
s  ← (RA)32
RT ← r&m | 64 s&¬m
```

The contents of the low-order 32 bits of register RA are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Bit 32 of RA is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RT_{32:63}$. Bit 32 of RA is replicated to fill $RT_{0:31}$. A shift amount of zero causes RT to receive $EXTS((RA)_{32:63})$. Shift amounts from 32 to 63 give a result of 64 sign bits.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

CA

Programming Note: XSR-Image field CA is set to 1 if (RA) is negative and any 1-bits are shifted out of position 63; otherwise $XSR\text{-}Image_{CA}$ is set to 0.

A shift amount of zero causes $XSR\text{-}Image_{CA}$ to be set to 0.

5.15 Fixed-Point Move Assist Instructions

The *Move Assist* instructions are used to assist the movement of data in storage without concern for alignment.

A set of *Move Assist* primitives, when executed in adjacent parcels within a VLIW, allows for arbitrarily alignment of strings in General Purpose Registers; these strings are used by *Load/Store String* instructions.

Loading an arbitrarily aligned string is implemented as a two-step process:

- load several aligned storage locations into GPRs; and
- simultaneously left-shift several GPRs.

Similarly, storing an arbitrarily aligned string is implemented as a two-step process:

- simultaneously right-shift several GPRs; and
- store several GPRs into aligned storage locations.

The *Move Assist* instructions use two registers to specify the string, as follows:

- RA: a General Purpose Register containing the starting storage address (byte address) of the string;
- MAR: a Special Purpose Register containing the ending byte address of the string, plus 1.

Programming Note: The PowerPC string instructions have been factored into simpler primitives in the ForestaPC architecture; these primitive instructions are executed concurrently in different parcels (composing a multiparcel primitive).

Programming Note: In contrast to a PowerPC processor, these instructions use the starting and ending byte address of the string instead of the starting address and the byte count.

Shift Left String Word X10-form

slsw RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	289

$dw \leftarrow (RB)_{32:63} \parallel from\_right\_parcel_{32:63}$
$bs \leftarrow 8 \times (RA)_{62:63}$
$RT \leftarrow {}^{32}0 \parallel dw_{bs:bs+31}$
$to\_left\_parcel \leftarrow undefined \parallel (RB)_{32:63}$

This instruction is a multiparcel primitive. Register RA contains the starting storage byte address of a string; $(RB)_{32:63}$ is a word of the string. Let *dw* be a doubleword composed of $(RB)_{32:63}$ concatenated with the low-order 32 bits of the data received from the right-adjacent parcel. If the input from the right-adjacent parcel is not active (not executing an *slsw*), or the parcel executing this instruction is the right-most parcel in a VLIW, the corresponding data is set to zero. Let *bs* be the number of bits that the data must be shifted to the left so that the string becomes left-aligned in the registers; this number is determined from the starting byte address of the string. The contents of $dw_{bs:(bs+31)}$ are stored into the low-order 32 bits of register RT. $RT_{0:31}$ are set to 0.

$(RB)_{32:63}$ is also passed to the parcel on the left, to contribute to a possible *slsw* primitive executing there, unless the parcel executing this instruction is the left-most parcel in a VLIW.

The right end of the string is filled with zeros.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Shift Left String Doubleword X10-form

slsd RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	288

$qw \leftarrow (RB) \parallel from\_right\_parcel$
$bs \leftarrow 8 \times (RA)_{61:63}$
$RT \leftarrow qw_{bs:bs+63}$
$to\_left\_parcel \leftarrow (RB)$

This instruction is a multiparcel primitive. Register RA contains the starting byte address of a string; (RB) is a doubleword of the string. Let *qw* be a quadword composed of (RB) concatenated with 64-bits of data received from the right-adjacent parcel. If the input from the right-adjacent parcel is not active (not executing an *slsd*), or the parcel executing this instruction is the right-most parcel in a VLIW, the corresponding data is set to zero. Let *bs* be the number of bits that the data must be shifted to the left so that the string becomes left-aligned in the registers; this number is determined from the starting byte address of the string. The contents of $qw_{bs:(bs+63)}$ are stored into register RT.

(RB) is also passed to the left parcel, to contribute to a possible *slsd* primitive executing there, unless the parcel executing this instruction is the left-most parcel in a VLIW.

The right end of the string is filled with zeros.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Shift Right String Word X10-form

srsw RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	291

$dw \leftarrow (RB)_{32:63} \parallel from\_right\_parcel_{32:63}$
$bs \leftarrow 8\times(4-(RA)_{62:63})$
$RT \leftarrow {}^{32}0 \parallel dw_{bs:bs+31}$
$to\_left\_parcel \leftarrow undefined \parallel (RB)_{32:63}$

This instruction is a multiparcel primitive. Register RA contains the starting byte address of a string; $(RB)_{32:63}$ is a word of the string. Let *dw* be a doubleword composed of $(RB)_{32:63}$ concatenated with the low-order 32 bits of the data received from the right-adjacent parcel. If the input from the right-adjacent parcel is not active (not executing an *srsw*), or the parcel executing this instruction is the rightmost parcel in a VLIW, the corresponding data is set to zero. Let *bs* be the number of bits that the data must be shifted to the right so that the string becomes unaligned in the registers; this number is determined from the starting byte address of the string. The right-shift is actually implemented as a left-shift. The contents of $dw_{bs:(bs+31)}$ are stored into the low-order 32-bits of register RT. $RT_{0:31}$ are set to 0.

$(RB)_{32:63}$ is also passed to the parcel to the left, to contribute to a possible *srsw* primitive executing there, unless the parcel executing this instruction is the leftmost parcel in a VLIW.

The ends of the string are filled with zeros.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Shift Right String Doubleword X10-form

srsd RT,RA,RB

0	4	10	16	22
0	RT	RA	RB	290

$qw \leftarrow (RB) \parallel from\_right\_parcel$
$bs \leftarrow 8\times(8-(RA)_{61:63})$
$RT \leftarrow dw_{bs:bs+63}$
$to\_left\_parcel \leftarrow (RB)$

This instruction is a multiparcel primitive. Register RA contains the starting byte address of a string; (RB) is a doubleword of the string. Let *qw* be a quadword composed of (RB) concatenated with the 64-bits of data received from the right-adjacent parcel. If the input from the right-adjacent parcel is not active (not executing an *srsd*), or the parcel executing this instruction is the rightmost parcel in a VLIW, the corresponding data is set to zero. Let *bs* be the number of bits that the data must be shifted to the right so that the string becomes unaligned in the registers; this number is determined from the starting byte address of the string. The right-shift is actually implemented as a left-shift. The contents of $qw_{bs:(bs+63)}$ are stored into register RT.

(RB) is also passed to the parcel to the left, to contribute to a possible *srsd* primitive executing there, unless the parcel executing this instruction is the leftmost parcel in a VLIW.

The ends of the string are filled with zeros.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

5.16 Fixed-Point Shift and Add Instructions

These instructions combine an add operation with a left shift by a specified number of bit positions less than or equal to 8.

Shift Left Doubleword Immediate then Add X6-form

sldia RT,RA,RB,SH

0	4	10	16		23	26
14	RT	RA	RB	/	SH	32

```
n  ← SH + 1
r  ← ROTL64((RA),n)
m  ← MASK(0,63-n)
RT ← (r & m) + RB
```

The contents of register RA are shifted left SH+1 bits. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The shifted value is added to the contents of register RB. The result is placed into register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Shift Left Word Immediate then Add X6-form

slwia RT,RA,RB,SH

0	4	10	16		23	26
14	RT	RA	RB	/	SH	33

```
n  ← SH + 1
r  ← ROTL32((RA)32:63,n)
m  ← MASK(32,63-n)
RT ← (r & m) + RB
```

The contents of the low-order 32 bits of register RA are shifted left SH+1 bits. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The shifted value is zero-extended to the left to 32 bits. The shifted/extended value is added to the contents of register RB. The result is placed into register RT.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

5.17 Move To/From Special Purpose Registers Instructions

Extended mnemonics

A set of extended mnemonics is provided for the *mtspr* and *mfspr* instructions so that they can be coded with the name of the Special Purpose Register as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the relevant instructions.

Move To Special Purpose Register X10-form

mtspr SPT,RA

0	4	6	10	16	22
0	/	spt_1	RA	spt_0	784

```
n  ← spt0 || spt1
if length(SPREG(n))=64 then
  SPREG(n) ← (RA)
else
  SPREG(n) ← (RA)32:63{0:31}
```

The SPT field denotes a Special Purpose Register, encoded as shown in the table below. The contents of register RA are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RA are placed into the SPR.

decimal	SPT	Register name
1	00000 00001	XSR
4	00000 00100	FPSCR
8	00000 01000	BR0
16	00000 10000	MAR

If the SPT field contains any value other than one of the values shown above, then one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system privileged instruction error handler is invoked.
- The results are boundedly undefined.

A complete description of this instruction is given in *Book III, ForestaPC Operating Environment Architecture*.

Special Registers Altered:.
See above

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
mtxer	Rx	mtspr	1,Rx
mtbr0	Rx	mtspr	8,Rx
mtmar	Rx	mtspr	16,Rx

Move From Special Purpose Register X10-form

mfspr RT,SPS

0	4	10	12	16	22
0	RT	/	sps_1	sps_0	785

```
n  ← sps0 || sps1
if length(SPREG(n))=64 then
  RT ← SPREG(n)
else
  RT ← 320 || SPREG(n)
```

The SPS field denotes a Special Purpose Register, encoded as shown in the table below. The contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the SPR, and the high-order 32-bits of RT are set to 0.

decimal	SPS	Register name
1	00000 00001	XSR
4	00000 00100	FPSCR
8	00000 01000	BR0
16	00000 10000	MAR

If the SPS field contains any value other than one of the values shown above, then one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system privileged instruction error handler is invoked.
- The results are boundedly undefined.

A complete description of this instruction is given in *Book III, ForestaPC Operating Environment Architecture*.

Special Registers Altered:.
None

XSR-Image Fields Generated:
None

Examples of Extended Mnemonics:

Extended:		*Equivalent to:*	
mfxer	Rx	mfspr	Rx,1
mfbr0	Rx	mfspr	Rx,8
mfmar	Rx	mfspr	Rx,16

Move to Condition Register from XSR X10-form

mcrxr CRT

0	4	8	10	16	22
0	CRT	//	///	///	817

$CR_{CRT} \leftarrow XSR_{0:3}$
$XSR_{0:3} \leftarrow$ 0b0000

The contents of $XSR_{0:3}$ are copied into the Condition Register field designated by CRT. $XSR_{0:3}$ are set to zero.

Special Registers Altered:
XSR bits 0:3
CR Field CRT

XSR-Image Fields Generated:
None

Update XSR From Image X10-form

uxsr RA,XM

0	4	10	16	20	22
0	///	RA	//	XM	782

$XSR \leftarrow RA_{XM}$

The contents of the XSR-Image in register RA are placed into XSR. Only the XSR fields specified by the XM mask are copied, as follows:

OV	if $XM_0 = 1$
CA	if $XM_1 = 1$

Special Registers Altered:
XSR

XSR-Image Fields Generated:
None

5.18 Move To/From FPSCR Instructions

Every *Move To/From FPSCR* instruction appears to synchronize the effects of all instructions executed by a processor. Executing a *Move To/From FPSCR* instruction ensures that all *Update FPSCR* instructions previously initiated by the processor appear to have completed before the *Move To/From FPSCR* instruction is initiated, and that no subsequent *Update FPSCR* instructions appear to be initiated by the processor until the *Move To/From FPSCR* instruction has completed. In particular:

- all exceptions that will be caused by the previously initiated *Update FPSCR* instructions are recorded in the FPSCR before the *Move To/From FPSCR* instruction is initiated;
- all invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated *Update FPSCR* instructions have occurred before the *Move To/From FPSCR* instruction is initiated; and
- no subsequent floating-point instruction that depends on or alters the setting of any FPSCR bits appears to be initiated until the *Move To/From FPSCR* instruction has completed.

(Floating-point *Storage Access* instructions are not affected.)

Move From FPSCR X10-form

mffs RT,CRT

0	4	10	16	20	22
0	RT	///	CRT	/	789

RT ← FPSCR

The contents of the FPSCR are placed into bits 32:63 of register RT. Bits 0:31 of register RT are undefined.

CR field CRT is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register.

Special Registers Altered:

CR Field CRT

XSR-Image Fields Generated:

None

Move to Condition Register From FPSCR X10-form

mcrfs CRT,BFS

0	4	8	10	16	19	22
0	CRT	//	//	BFS	//	804

The contents of FPSCR field BFS are copied to CR field CRT. All exception bits copied (except FEX and VX) are set to 0 in the FPSCR.

CR field CRT is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register.

Special Registers Altered:

CR Field CRT

FPSCR Fields	
FX OX	(if BFS = 0)
UX ZX XX VXSNAN	(if BFS = 1)
VXISI VXIDI VXZDZ VXIMZ	(if BFS = 2)
VXVC	(if BFS = 3)
VXSOFT VXSQRT VXCVI	(if BFS = 5)

XSR-Image Fields Generated:

None

Update FPSCR From Image X10-form

ufsr RA,FM

0	4	10	16	22
0	///	RA	FM	783

$FPSCR \leftarrow RA_{FM}$

The contents of the FSR-Image in register RA are placed into FPSCR. Only the FSR fields specified by the FM mask are copied, as follows:

FX OX	if $FM_0 = 1$
UX ZX XX VXSNAN	if $FM_1 = 1$
VXISI VXIDI VXZDZ VXIMZ	if $FM_2 = 1$
VXVC	if $FM_3 = 1$
VXSOFT VXSQRT VXCVI	if $FM_4 = 1$
FPRF FR FI	if $FM_5 = 1$

Special Registers Altered:

FPSCR

XSR-Image Fields Generated:

None

Move To FPSCR Field Immediate X10-form

mtfsfi BFT,CRT,BFI

0	4	8	10	12	16	19	22
0	CRT	//	//	BFI	BFT	//	788

$FPSCR_{4*BFT:4*BFT+3} \leftarrow BFI$

$FPSCR_2 \leftarrow FPSCR_7 \mid FPSCR_8 \mid FPSCR_9 \mid FPSCR_{10} \mid FPSCR_{11} \mid FPSCR_{12} \mid FPSCR_{21} \mid FPSCR_{22} \mid FPSCR_{23}$

$FPSCR_1 \leftarrow FPSCR_0 \& FPSCR_{25} \mid FPSCR_4 \& FPSCR_{26} \mid FPSCR_5 \& FPSCR_{27} \mid FPSCR_6 \& FPSCR_{28} \mid FPSCR_2 \& FPSCR_{24}$

The value of field BFI is placed into FPSCR field BFT.

$FPSCR_0$ (FX) is altered only if BFT = 0.

CR field CRT is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register.

Special Registers Altered:

FPSCR field BFT
CR Field CRT

XSR-Image Fields Generated:

None

Programming Note: When $FPSCR_{0:3}$ is specified, bits 0 (FX) and 3 (OX) are set to the values of BFI_0 and BFI_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from BFI_0 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given on Section 2.3.5, “Floating-Point Status and Control Register,” on page 24, and not from $BFI_{1:2}$

Move To FPSCR Fields X10-form

mtfsf CRT,RA,FM

0	4	8	10	16	22
0	CRT	fm_1	RA	fm_0	787

$FM \leftarrow fm_1 \parallel fm_0$

The contents of bits 32:63 of register RA are placed into the FPSCR, under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0 to 7. If FM_i=1 then FPSCR field *i* (FPSCR bits 4*i* through 4*i*+3) is set to the contents of the corresponding field of the low-order 32 bits of register RA.

$FPSCR_0$ (FX) is altered only if $FM_0 = 0$.

CR field CRT is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register.

Special Registers Altered:

FPSCR fields selected by mask
CR Field CRT

XSR-Image Fields Generated:

None

Programming Note: Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

Programming Note: When $FPSCR_{0:3}$ is specified, bits 0 (FX) and 3 (OX) are set to the values of $(RA)_{32}$ and $(RA)_{35}$ (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from $(RA)_{32}$ and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given on page XX, and not from $(RA)_{33:34}$.

Move To FPSCR Bit 0 X10-form

mtfsb0 FBT,CRT

0	4	5	10	16	20	22
0	/	FBT	///	CRT	//	790

Bit FBT of the FPSCR is set to 0.

CR field CRT is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register.

Special Registers Altered:

FPSCR bit FBT
CR Field CRT

XSR-Image Fields Generated:

None

Programming Note: Bits 1 and 2 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X10-form

mtfsb1 FBT,CRT

0	4	5	10	16	20	22
0	/	FBT	///	CRT	//	791

Bit FBT of the FPSCR is set to 1.

CR field CRT is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register.

Special Registers Altered:

FPSCR bit FBT
CR Field CRT

XSR-Image Fields Generated:

None

Programming Note: Bits 1 and 2 (FEX and VX) cannot be explicitly set.

5.19 Move Register Instructions

These *Move Register* instructions allow the movement of data between registers.

Move from Floating-Point Register X10-form

mffpr RT,FRA

0	4	10	16	22
0	RT	FRA	///	796

RT ← (FRA)

The contents of Floating-Point Register FRA are placed into General Purpose Register RT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

Move to Floating-Point Register X10-form

mtfpr FRT,RA

0	4	10	16	22
0	FRT	RA	///	797

FRT ← (RA)

The contents of General Purpose Register RA are placed into Floating-Point Register FRT.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

XSR-Image Fields Generated:
None

5.20 Commit Instructions

The *Commit* instructions are used to commit results generated speculatively. In particular, *Commit* instructions are used to commit the contents of one register into another register.

Commit Speculative Register X10-form

csr RT,RA

0	4	10	16	22
0	RT	RA	///	792

The Delayed Exception Bit associated with RA is checked. If the bit is not set, the contents of register RA are placed into register RT; otherwise, a Delayed Exception is raised to the processor.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Commit Speculative FPR X10-form

csfr FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	793

The Delayed Exception Bit associated with FRA is checked. If the bit is not set, the contents of register FRA are placed into register FRT; otherwise, a Delayed Exception is raised to the processor.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Commit Speculative Register and Condition Register Field X8-form

csrcr RT,CRT,RA,CRS

0	4	10	16	20	24
15	RT	RA	CRT	CRS	1

RT ← (RA)
CRT ← (CRS)

General Purpose Register RT and Condition Register Field CRT are respectively updated with the contents of General Purpose Register RA and Condition Register Field CRS. The Delayed Exception Bit associated with RA and CRS are checked; if neither one of these bits is set to 1, the update operations take place, otherwise a Delayed Exception is raised to the processor.

Special Registers Altered:

None

XSR-Image Fields Generated:

None

Chapter 6. Floating-Point Instructions

This chapter describes the Floating-Point instructions and their associated features. Section 6.1 provides an overview of the Floating-Point Instruction Set Architecture, Section 6.2 describes the floating-point data formats, Section 6.3 describes the exceptions arising from floating-point operations, Section 6.4 describes the floating-point execution models, Section 6.5 describes the speculative execution of floating-point instructions, and Section 6.6 describes the instructions.

Storage access instructions for floating-point operands are described in Section 4.2.2, “Floating-Point Storage Accesses,” on page 37.

6.1 Floating-Point Overview

The Floating-Point Instruction Set Architecture provides instructions for:

- performing arithmetic, conversion, comparison and other floating-point operations on data in Floating-Point Registers, storing the result in a Floating-Point Register;
- moving floating-point data between Floating-Point Registers; and
- performing conversion of data in floating-point format in a Floating-Point Register into integer format in a General Purpose Register, and for performing conversion of data in integer format in a General Purpose Register into floating-point format in a Floating-Point Register.

The architecture provides for the processor to implement a floating-point system as defined in ANSI/IEEE Standard 754-1985, “IEEE Standard for Binary Floating-Point Arithmetic” (hereafter referred to as “the IEEE standard”), but requires software support in order to conform fully with that standard. That standard defines certain required “operations” (addition, subtraction, etc.); the term “floating-point operation” is used in this chapter to refer to one of these required operations, or to the operation performed by one of the *Multiply-Add* or *Reciprocal Estimate* instructions. All floating-point operations conform to that standard, except if software sets the Floating-Point Non-IEEE Mode (NI) bit in the Floating-Point Status and Control Register to 1 (see Section 2.3.5, “Floating-Point Status and Control Register,” on page 24), in which case floating-point operations do not necessarily conform to that standard.

The floating-point instructions are divided into two categories:

- floating-point computational instructions

  These instructions perform addition, subtraction, multiplication, division, extracting the square-root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They generate status information in a Floating-Point Status Image (FSR-Image) These instructions are described in Section 6.6.2 through Section 6.6.4.
- floating-point non-computational instructions

  These instructions move the contents of a floating-point register to another floating-point register possibly altering the sign, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations, and they do not generate status information in a Floating-Point Status Image. These instructions are described in Section 6.6.1 and Section 6.6.5.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this num-

ber is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values which are "Not a Number" (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation; their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables, and can be produced by certain invalid operations.

Floating-Point Exceptions

There is one class of exceptional events which occur during execution of floating-point instructions:

- Floating-Point Exceptions

Floating-point exceptions are signalled with bits set in the Floating-Point Status Image (FSR-Image). Floating-Point exceptions can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set in an *Extend FSR* instruction in the right-adjacent slot.

The following floating-point exceptions are detected by the processor:

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - Infinity-Infinity (VXISI)
  - Infinity÷Infinity (VXIDI)
  - Zero÷Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software Request (VXSOFT)
  - Invalid Square Root (VXSQRT)
  - Invalid Integer Convert (VXCVI)
- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FSR-Image. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 2.3.5, "Floating-Point Status and Control Register," on page 24 and Section , "Floating-Point Status Image," on page 27 for a description of these exception and enable bits, and Section 6.3, "Floating-Point Exceptions," on page 123, for a detailed discussion of the floating-point exceptions, including the effects of the enable bits.

Floating-Point Registers

This architecture provides 64 floating-point registers (FPRs), numbered 0-63. Each Floating-Point Register (FPR) contains 64-bits which support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The floating-point computational instructions, and the *Move* and *Select* instructions, operate on data located in Floating-Point Registers (FPRs) and, with the exception of the *Floating-Point Compare* instructions, place the result value into a Floating-Point Register. *Compare* instructions place the result into the Condition Register.

Load and *Store Double* instructions (which correspond to *Storage Access* instructions) are provided to transfer 64 bits of data between storage and the FPRs with no conversion. *Load Single* instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. *Store Single* instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage. These instructions are described in Chapter 4., "Storage Access Instructions," on page 37.

Instructions are provided for manipulating the Floating-Point Status and Control Register; these instructions are described in Section 5.18, "Move To/From FPSCR Instructions," on page 110. Some of these instructions copy data from a GPR to the Floating-Point Status and Control Register, or vice versa.

The floating-point computational instructions and the *Floating-Point Select* instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FSR-Image, are undefined.

The floating-point arithmetic, rounding and conversion instructions produce intermediate results which may be regarded as being infinitely precise. After normalization or denormalization, if the infinitely precise intermediate result is not representable in the destination format (either 32-bit or 64-bit) then it is rounded. The final result is then placed into the target floating-point register in the double format.

6.2 Floating-Point Data

6.2.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The length of the exponent and the fraction fields differ among these two formats. The structure of the single and double formats is shown below:

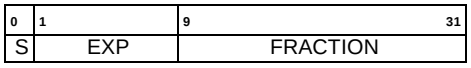


Figure 25: Floating-Point Single Format

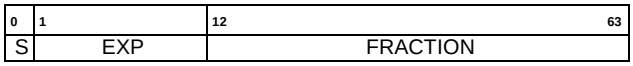


Figure 26: Floating-Point Double Format

Values in floating-point formats are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

If only a portion of a floating-point data item in storage is accessed, such as with a *Load* or *Store* instruction for a byte or halfword (or word in the case of a floating-point double format), the value affected will depend on whether the system is operation with Big-Endian byte order (the default), or Little-Endian byte order.

Representation of numerical values in the floating-point formats consists of a sign bit S, a biased exponent EXP, and the fraction portion FRACTION of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers, and is located in the unit bit position (i.e. the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 27.

The architecture requires that the FPRs only support the floating-point double format.

	Format	
	Single	**Double**
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	-126	-1022
Width (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

Figure 27: IEEE Floating-Point Fields

6.2.2 Value Representation

This architecture defines numerical and non-numerical values representable within each of the two supported formats. The numerical values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numerical values representable are the Infinities and the Not-a-Numbers (NaNs). The infinities are adjoined to the real numbers but are not numbers themselves, and the standard rules of arithmetic do not hold when they appear in an operation. They are related to the real numbers by order alone. It is possible, however, to define restricted operations among numbers and infinities as described below. The relative location on the real number line for each of the defined entities is shown in Figure 28.

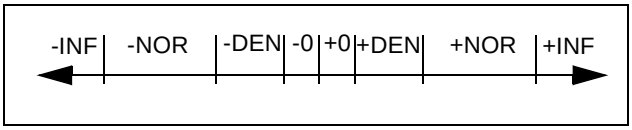


Figure 28: Approximation to Real Numbers

The NaNs are not related to the numbers or infinities by order or value; instead, they are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers:

Machine representable values used as approximations to real numbers. Three categories of numbers

are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers(±NOR)**:**

These are values which have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is one. Normalized numbers are interpreted as follows:

$$NOR = (-1)^s \times 2^E \times (1.fraction)$$

where (s) is the sign, (E) is the unbiased exponent and (1.fraction) is the significand which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

$1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$

Double Format:

$2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$

Zero values(±0)**:**

These are values which have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to -0).

Denormalized numbers(±DEN)**:**

These are values which have a biased exponent value of zero and a non-zero fraction value. They are non-zero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is zero. Denormalized numbers are interpreted as follows:

$$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$$

where Emin is the minimum representable exponent value (-126 for single-precision, -1022 for double-precision).

Infinities(±∞)**:**

These are values which have the maximum biased exponent value:

255 in the single format
2047 in the double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$$-\infty < \text{every finite number} < +\infty$$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 6.3.1, “Invalid Operation Exception,” on page 126.

Not a Numbers (NaNs)**:**

These are values which have the maximum biased exponent value and a non-zero fraction value. The sign bit is ignored (i.e. NaNs are neither positive nor negative). If the high-order bit of the fraction field is a zero then the Nan is a *Signalling NaN*, otherwise it is a *Quiet NaN*.

*Signalling NaN*s are used to signal exceptions when they appear as arithmetic operands.

Quiet NaNs are used to represent the result of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled ($FPSCR_{VE}$=0). *Quiet NaNs* propagate through all operations except ordered comparison, *Floating Round to Single Precision*, and conversion to integer. *Quiet NaNs* do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings, in QNaNs, can thus be preserved through a sequence of operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of an operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to one that is to be stored as the result:

```
if (FRA) is a NaN
    then (FRT) ← (FRA)
    else if (FRB) is a NaN
        then if instruction is frsp
            then (FRT) ← (FRB)0:34 || 29 0
            else (FRT) ← (FRB)
        else if (FRC) is a NaN
            then (FRT) ← (FRC)
```

else if generated QNaN
then (FRT) ← generated QNaN

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is *frsp*. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of zero, an exponent field of all ones, and a high-order fraction bit of one with all other fraction bits zero. Any instruction that generates a QNaN as the result of a disabled Invalid Operation must generate this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaNs fraction are zero.

6.2.3 Sign of Result

The following rules govern the sign of the result of a floating-point arithmetic operation, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation x-y is the same as the sign of the result of the add operation x+(-y).

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round towards -Infinity, in which mode the sign is negative.
- The sign of the result of a multiplication or division operation is the Exclusive-OR of the signs of the operands.
- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.
- The sign of the result of a *Round to Single-Precision* or *Convert to/from Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the addition or subtraction operation is the result of the multiply operation).

6.2.4 Normalization and Denormalization

The intermediate result of a floating-point arithmetic or *frsp* instruction may require normalization and/or denormalization, as described below. Normalization and denormalization do not affect the sign of the result.

When a floating-point arithmetic or *frsp* instruction produces an intermediate result, consisting of a sign bit, an exponent, and a nonzero significand with a zero leading bit, it is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decreasing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 6.4.1, “Execution Model for IEEE Operations,” on page 130) participate in the shift, with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a non-zero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 6.3.4, “Underflow Exception,” on page 128. These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by one for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process then “Loss of Accuracy” has occurred (see Section 6.3.4, “Underflow Exception,” on page 128) and Underflow Exception is signalled.

Engineering Note: When denormalized numbers are operands of floating-point multiply, divide, and square root operations, some implementations

may pre-normalize the operands internally before performing the operations.

6.2.5 Data Handling and Precision

The Floating-Point Instruction Set Architecture includes instructions to move floating-point data between the FPRs and storage. For double format, the data is not altered during the move. For single format, a format conversion from single to double is performed when loading from storage into an FPR, and a format conversion from double to single is performed when storing an FPR into storage. No floating-point exceptions are caused by these instructions.

All computational, *Move* and *Select* instructions use the floating-point double format.

Floating-point single-precision values are obtained with the following types of instructions:

1. *Load Floating-Point Single*

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double-precision, and loads it into an FPR. No floating-point exceptions are caused by these instructions.

2. Round to Floating-Point Single-Precision

   The *Floating Round to Single Precision* instruction rounds a double-precision operand to single-precision if the operand is not already in single-precision range, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the *Floating Round to Single Precision* instruction, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

   This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result correct to infinite precision and with unbounded range, and then coerces this intermediate result to fit in single format. Status bits in the FSR-Image are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

   All input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FSR-Image are undefined.

4. *Store Floating-Point Single*

   This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions (the value being stored is effectively assumed to be the result of an instruction of one of the preceding three types).

When the result of a *Load Floating-Point Single, Floating Round to Single-Precision,* or single-precision arithmetic instructions is stored in an FPR, the low-order 29 FRACTION bits are zero.

Programming Note: The *Floating Round to Single Precision* instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision *Load* and arithmetic instructions and by *fcfid*) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision *Load* and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the *Store* or the arithmetic instruction by a *Floating Round to Single Precision* instruction.

Programming Note: A single-precision value can be used in double-precision arithmetic operations. The reverse is not necessarily true (it is true only if the double-precision value is representable in single format).

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

6.2.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the infinitely precise intermediate result

of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 6.2.2, “Value Representation,” on page 119 and Section 6.3, “Floating-Point Exceptions,” on page 123 for the cases not covered here.

With the exception of the two *Estimate* instructions, *Floating Reciprocal Estimate Single* and *Floating Reciprocal Square Root Estimate*, all arithmetic, rounding and conversion instructions defined by this architecture produce an intermediate result that can be regarded as being infinitely precise. This result must then be written with a precision of finite length into a FPR. After normalization or denormalization, if the infinitely precise intermediate result is not representable in the precision required by the instruction then it is rounded before being placed into the target FPR.

The instructions that may round their result are the *Arithmetic* and *Rounding and Conversion* instructions. For a given instance of one of these instructions, whether rounding actually occurs depends on the values of the inputs. Each of these instructions sets FSR-Image bits FR and FI, according to whether rounding occurred (FI) and whether the fraction was increased (FR). If rounding occurred, FI is set to 1, and FR may be set to either 0 or 1. If rounding did not occur, both FR and FI are set to 0.

The two *Estimate* instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

Four user-selectable modes of rounding are provided through the Floating-Point Round Control field in the FPSCR. See Section 2.3.5, “Floating-Point Status and Control Register,” on page 24. These are encoded as follows:

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward -Infinity

Let Z be the infinitely precise intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then no rounding occurs, and the result in all rounding modes is equivalent to truncation of Z. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then, Z1 or Z2 can be used to approximate the result in the target format.

Figure 29 shows the relationship among Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. “LSB” means “least significant bit”.

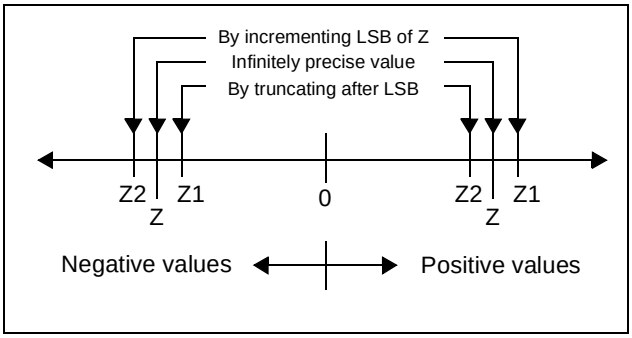


Figure 29: Selection of Z1 and Z2

Round to Nearest:
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one which is even (least significant bit 0).

Round toward Zero:
Choose the smaller in magnitude (Z1 or Z2).

Round Toward +Infinity:
Choose Z1.

Round Toward -Infinity:
Choose Z2.

See Section 6.4.1, “Execution Model for IEEE Operations,” on page 130 for a detailed explanation of rounding.

6.3 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity-Infinity
  - Infinity÷Infinity
  - Zero÷Zero
  - Infinity×Zero
  - Invalid Compare
  - Software Request
  - Invalid Square Root

- Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of floating-point computational instructions. In addition, an Invalid Operation Exception occurs when a *Status and Control Register* instruction sets $FPSCR_{VXSOFT}$ to 1 (Software Request).

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FSR-Image. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 125) and with the FM bits in a *Update XSR* instruction, whether and how the system floating-point enabled exception error handler is invoked. (In general, the functionality specified by the enable bit corresponds to enabling the invocation of the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

The floating-point Exception Summary bit (FX) in the FPSCR is set to 1 by any *Move to FPSCR* instruction, except *mtfsfi* and *mtfsf*, that causes any of the floating-point exception bits in the FPSCR to change from 0 to 1, or by a *mtfsfi, mtfsf*, or *mtfsb1* instruction that explicitly sets the bit to 1. The floating-point Enabled Exception Summary bit (FEX) in the FPSCR is set when any of the exceptions is set and the exception is enabled (enable bit is one).

Unless state otherwise, this section describes the events that take place when a floating-point instruction extended with a *Extend FSR* instruction are executed; the actual reporting of the exception takes place when the *FSR-Image* generated by the floating-point instruction and placed by the *Extend FSR* instruction in a GPR is used to update the FPSCR.

A single instruction may set more than one exception bit in the *FSR-Image* only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception ($\infty \times 0$) for *Multiply-Add* instructions for which the values being multiplied are infinity and zero, and the value being added is an SNaN.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for *Convert to Integer* instructions.

When an exception occurs, the instruction execution may be suppressed or a result may be delivered, depending on the exception.

Instruction execution is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE stan-

dard for the “trap enabled” case; the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case; the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case, the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur; software can inspect the FSR-Image exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case, the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs.

Whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs is controlled by the FE0 and FE1 bits. The location of these bits and the requirements for altering them are described in *Book III, ForestaPC Operating Environment Architecture*. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception). The effects of the four possible settings of these bits are as follows:

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction nor the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.
1	0	**Imprecise Recoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler so that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

Architecture Note: The FE0 and FE1 bits must be defined in Book III in a manner such that they can be changed dynamically and can be easily treated as part of a process' state.

In all cases the question of whether or not a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all VLIWs and instructions within the current VLIW prior to the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no VLIW or instructions within the current VLIW after the instruction at which the system floating-point enabled exception error handler is invoked has been executed. (Recall that, for the two imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed, unless it is the excepting instruction, in which case it has been executed unless the kind of exception is among those listed above as suppressed.

Programming Note: In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register*

instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

A *sync* instruction, or any other execution synchronizing instruction or event (e.g., *isync*: see *Book II, ForestaPC Virtual Environment Architecture*), also has the effects described above. However, in order to obtain the best performance across the widest range of implementations, a *Floating-Point Status and Control Register* instruction should be used to obtain these effects.

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines:

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used, with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

Engineering Note: It is permissible for the implementation to be precise in any of the three modes that permit exceptions, or to be recoverable in Nonrecoverable Mode.

6.3.1 Invalid Operation Exception

6.3.1.1 Definition

An Invalid Operation exception is detected whenever an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a signalling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities (∞-∞)
- Division of infinity by infinity (∞÷∞)
- Division of zero by zero (0÷0)
- Multiplication of infinity by zero (∞×0)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and non-zero) number (Invalid Square Root)
- Integer convert involving a large number, an infinity, or a NaN (Invalid Integer Convert)

In addition, an Invalid Operation Exception occurs if software explicitly request this by executing a *mtfsfi, mtfsf*, or *mtfsb1* instruction that sets $FPSCR_{VXSOFT}$ to 1 (Software Request).

Programming Note: The purpose of $FPSCR_{VXSOFT}$ is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

6.3.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled, $FPSCR_{VE}$=1, and Invalid Operation occurs or software explicitly requests the exception, then the following actions are taken:

1. One or two Invalid Operation Exceptions is(are) set:

$FSR\text{-}Image_{VXSNAN}$ (if SNaN)
$FSR\text{-}Image_{VXISI}$ (if ∞-∞)
$FSR\text{-}Image_{VXIDI}$ (if ∞÷∞)
$FSR\text{-}Image_{VXZDZ}$ (if 0÷0)
$FSR\text{-}Image_{VXIMZ}$ (if ∞×0)
$FSR\text{-}Image_{VXVC}$ (if invalid comp)
$FSR\text{-}Image_{VXSOFT}$ (if software req)
$FSR\text{-}Image_{VXSQRT}$ (if invalid sqrt)
$FSR\text{-}Image_{VXCVI}$ (if invalid int cvrt)

2. If the operation is an arithmetic, *Floating Round to Single-Precision*, or convert to integer operation:

   the target FPR is unchanged;
   $FSR\text{-}Image_{FR\ FI}$ are set to zero;
   $FSR\text{-}Image_{FPRF}$ is unchanged.

3. If the operation is a compare:

   $FSR\text{-}Image_{FR\ FI\ C}$ are unchanged;
   $FSR\text{-}Image_{FPCC}$ is set to reflect unordered.

4. If software explicitly requests the exception:

   $FPSCR_{FR\ FI\ FPRF}$ are as set by the *mtfsfi, mtfsf*, or *mtfsb1* instruction.

When Invalid Operation Exception is disabled ($FPSCR_{VE}$=0) and Invalid Operation occurs or software explicitly request the exception, then the following actions are taken:

1. One or two Invalid Operation Exceptions is set:

   $FSR\text{-}Image_{VXSNA}$ (if SNaN)
   $FSR\text{-}Image_{VXISI}$ (if ∞-∞)
   $FSR\text{-}Image_{VXIDI}$ (if ∞÷∞)
   $FSR\text{-}Image_{VXZDZ}$ (if 0÷0)
   $FSR\text{-}Image_{VXIMZ}$ (if ∞×0)
   $FSR\text{-}Image_{VXVC}$ (if NaN comp)
   $FSR\text{-}Image_{VXSOFT}$ (if software req)
   $FSR\text{-}Image_{VXSQRT}$ (if invalid sqrt)
   $FSR\text{-}Image_{VXCVI}$ (if invalid int cvrt)

2. If the operation is an arithmetic or *Floating Round to Single Precision* operation:

   the target FPR is set to a Quiet NaN;
   $FSR\text{-}Image_{FR\ FI}$ are set to zero;
   $FSR\text{-}Image_{FPRF}$ is set to indicate the class of the result (Quiet NaN).

3. If the operation is a convert to 64-bit integer operation:

   the target FPR is set as follows:

   FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or +∞, and to the most negative 64-bit integer if the operand in FRB is a negative number, -∞, or NaN;

   $FSR\text{-}Image_{FR\ FI}$ are set to zero;
   $FSR\text{-}Image_{FPRF}$ is undefined.

4. If the operation is a convert to 32-bit integer operation:

   the target FPR is set as follows:

   $FRT_{0:31}$ ← undefined;
   $FRT_{32:63}$ are set to the most positive 32-bit integer if the operand in FRB is a positive number, or +∞, and to the most negative 32-bit integer if the operand in FRB is a negative number, -∞, or NaN;

   $FSR\text{-}Image_{FR\ FI}$ are set to zero;
   $FSR\text{-}Image_{FPRF}$ is undefined.

5. If the operation is a compare:

   $FSR\text{-}Image_{FR\ FI\ C}$ are unchanged;
   $FSR\text{-}Image_{FPCC}$ is set to reflect unordered.

6. If software explicitly requests the exception:

   $FPSCR_{FR\ FI\ FPRF}$ are set by the *mtfsfi, mtfsf*, or *mtfsb1* instruction

6.3.2 Zero Divide Exception

6.3.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite non-zero dividend value. It also occurs when a *Reciprocal Estimate* instruction (*fres* or *frsqrte*) is executed with an operand value of zero.

Architecture Note: The name is a misnomer used for historical reasons. The proper name for this exception should be “Exact Infinite Result from Finite Operands” corresponding to what mathematicians call a “pole”.

6.3.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}$=1) and Zero Divide occurs then the following actions are taken:

1. Zero Divide Exception is set:
   $FSR\text{-}Image_{ZX} \leftarrow 1$
2. The target FPR is unchanged.
3. $FSR\text{-}Image_{FR\ FI}$ are set to zero.
4. $FSR\text{-}Image_{FPRF}$ is unchanged.

When Zero Divide Exception is disabled ($FPSCR_{ZE}$=0) and Zero Divide occurs then the following actions are taken:

1. Zero Divide Exception is set
   $FSR\text{-}Image_{ZX} \leftarrow 1$
2. The target FPR is set to a ±∞, where the sign is determined by the XOR of the signs of the operands.
3. $FSR\text{-}Image_{FR\ FI}$ are set to zero
4. $FSR\text{-}Image_{FPRF}$ is set to indicate the class and sign of the result (±Infinity)

6.3.3 Overflow Exception

6.3.3.1 Definition

Overflow occurs when the magnitude of what would have been the rounded result if the exponent range was unbounded exceeds that of the largest finite number of the specified result precision.

6.3.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled ($FPSCR_{OE}$=1) and exponent overflow occurs then the following actions are taken:

1. Overflow Exception is set
   $FSR\text{-}Image_{OX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536.
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192.
4. The adjusted rounded result is placed into the target FPR.
5. $FSR\text{-}Image_{FPRF}$ is set to indicate the class and sign of the result (±Normal Number).

When Overflow Exception is disabled ($FPSCR_{OE}$=0) and exponent overflow occurs then the following actions are taken:

1. Overflow Exception is set
   $FSR\text{-}Image_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FSR\text{-}Image_{XX} \leftarrow 1$
3. The result is determined by the rounding mode ($FPSCR_{RN}$) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store ±Infinity, where the sign is the sign of the intermediate result.
   - Round towards Zero
     Store the format's largest finite number with the sign of the intermediate result.
   - Round towards +Infinity
     For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity.
   - Round towards -Infinity
     For negative overflow, store -Infinity; for positive overflow, store the format's largest finite number.
4. The result is placed into the target FPR.
5. $FSR\text{-}Image_{FR}$ is undefined
6. $FSR\text{-}Image_{FI}$ is set to 1
7. $FSR\text{-}Image_{FPRF}$ is set to indicate the class and sign of the result (±Infinity or ±Normal Number).

6.3.4 Underflow Exception

6.3.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is “Tiny”.
- Disabled:
  Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A “Tiny” result is detected before rounding, when a non-zero result value computed as though the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is “Tiny” and the Underflow Exception Enable is off ($FPSCR_{UE}$=0) then the intermediate result is denormalized (Section 6.2.4, “Normalization and Denormalization,” on page 121) and rounded (Section 6.2.6, “Rounding,” on page 122) before being placed into the target FPR.

“Loss of Accuracy” is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded.

6.3.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and exponent underflow occurs then the following actions are taken:

1. Underflow Exception is set

   $FSR\text{-}Image_{UX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536.
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192.
4. The adjusted rounded result is placed into the target FPR
5. $FSR\text{-}Image_{FPRF}$ is set to indicate the class and sign of the result (±Normalized Number).

Programming Note: The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and underflow occurs then the following actions are taken:

1. Underflow Exception is set

   $FSR\text{-}Image_{UX} \leftarrow 1$
2. The rounded result is placed into the target FPR.
3. $FSR\text{-}Image_{FPRF}$ is set to indicate the class and sign of the result (±Denormalized Number or ±Zero).

6.3.5 Inexact Exception

6.3.5.1 Definition

Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming the intermediate result exponent range and precision to be unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

6.3.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs then the following actions are taken:

1. Inexact Exception is set

   $FSR\text{-}Image_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR.
3. $FSR\text{-}Image_{FPRF}$ is set to indicate the class and sign of the result

---

Programming Note: In some implementations, enabling Inexact Exceptions may degrade performance more than enabling other types of floating-point exceptions.

6.4 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to insure that identical results are obtained.

Special rules are provided in the definition of the arithmetic instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 6.2.2, “Value Representation,” on page 119, and Section 6.3, “Floating-Point Exceptions,” on page 123 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bit positions to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision and double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. The ForestaPC architecture follows these guidelines: double-precision arithmetic instructions can have operands of either or both precisions, whereas single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and fcfid produce double-precision values, whereas single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, whereas conversions from single-precision to double-precision are done implicitly.

6.4.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming 64-bit significand arithmetic is considered to be performed with a floating-point accumulator having the following format:

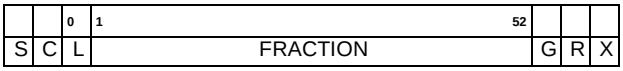


Figure 30: IEEE 64-bit Execution Model

The S bit is the sign bit.

The C bit is the carry bit that captures the carry out of the significand.

The L bit is the leading unit bit of the significand which receives the implicit bit from the operands.

The FRACTION is a 52-bit field which accepts the fraction of the operands.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low order bits of the accumulator. The G and R bits are required for post normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits which may appear to the low-order side of the R bit, either due to shifting the accumulator right or other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Figure 27 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

Figure 31: Interpretation of G, R and X bits

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G,R and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

Before the result is stored into an FPR, the significand of the infinitely precise intermediate result described above is rounded if necessary, using the rounding mode specified by $FPSCR_{RN}$. If rounding results in a carry into C, the significand is shifted right one position and the exponent increased by one. This yields an inexact result and possibly also exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR and low-order bit positions, if any, are set to zero.

Four user-selectable rounding modes are provided through $FPSCR_{RN}$, as described in Section 6.2.6, “Rounding,” on page 122. For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 32 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26:52, G, R, X

Figure 32: Location of the Guard, Round and Sticky Bits

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the Guard, Round, or Sticky bits are nonzero, then the result is inexact.

Z1 and Z2, as defined on page 123, can be used to approximate the result in the target format when one of the following rules is used.

- **Round to Nearest**

  Guard bit = 0

  The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011)).

  Guard bit = 1

  Depends on Round and Sticky bits:

  Case a:

  If the Round or Sticky bit is 1 (inclusive), the result is increased by 1. (Result closest to next higher value in magnitude (GRX = 101, 110, or 111)).

  Case b:

  If the Round and Sticky bits are zero (result midway between closest representable values) then if the low-order bit of the result is one the result is incremented. Otherwise (the low-order bit of the result is zero) the result is truncated (this is the case of a tie rounded to even).

- **Round towards Zero**

  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is non-zero, the result is inexact.

- **Round towards +Infinity**

  Choose Z1.

- **Round towards -Infinity**

  Choose Z2.

Where the result is to have fewer than 53 bits of precision because the instruction is a *Floating Round to Single-Precision* or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before any rounding is done.

6.4.2 Execution Model for Multiply-Add Type Instructions

The ForestaPC Architecture makes use of a special form of instruction which performs up to three operations in one instruction (a multiplication, an addition and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format:

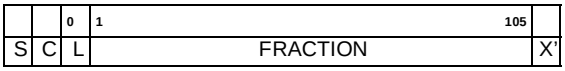


Figure 33: Multiply-Add 64-bit Execution Model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the fraction, and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the fraction) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount which is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the infinitely precise intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round and Sticky bits are defined in terms of accumulator bits. Figure 32 shows the positions of the Guard, Round and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X’
Single	24	25	OR of 26:105, X’

Figure 34: Location of the Guard, Round and Sticky bits in the multiply-add execution model

The rules of rounding the intermediate result are the same as those given in Section Section 6.4.1, “Execution Model for IEEE Operations,” on page 130.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract* the final result is negated.

6.5 Speculative Execution of Floating-Point Instructions

In the ForestaPC architecture, speculative execution is a technique usable by the compiler/programmer for improving performance. A speculative operation is one that has been placed out-of-order with respect to a sequential execution stream, on the speculation that the result will be needed. If subsequent events indicate that the speculative instruction would not have been executed, or the results of the speculative instruction are not valid, any result produced by the speculative instruction are not used. Typically, instructions are placed speculatively by the compiler/programmer when there are resources that would otherwise be idle so that the operation is done without cost, or to reduce delays in the program.

No error of any kind other than Machine Check should be reported due to the execution of a speculative instruction, until the result from its execution is used non-speculatively. If there were errors, the instruction should be re-executed at that point, as well as any other speculative instructions already executed that depend on the faulting instruction.

A floating-point value that has been loaded speculatively must be *committed* before it can be used non-speculatively (usually at the original place in the sequential execution stream). Floating-point values are committed using the instruction *Commit Speculative Floating-Point Register (csfr).*

Floating-point instructions other than Load can be speculated without explicit indication.

Floating-point instructions can be paired with an *Extend FSR (xfps)* instruction in the right-adjacent slot to the one containing the instruction. The *Extend* primitive specifies a GPR where an FSR-Image is placed, which can later be used to update FPSCR.

Programming Note: Floating-point instructions implicitly use the control fields from FPSCR (exception enable fields, rounding mode field, and so on). A speculative floating-point operation uses the values in the FPSCR control fields when the instruction is executed. Consequently, the speculation of a floating-point instruction across an instruction that may change the contents of the FPSCR control fields is a programming error.

6.6 Floating-Point Instructions

6.6.1 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for *fneg, fabs,* and *fnabs*. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of the NaN may be altered by *fneg, fabs,* and *fnabs*). These instructions do not generate a FSR-Image.

Floating Move Register X10-form

fmr FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	518

FRT ← (FRA)

The contents of register FRA are placed into register FRT.

Special Registers Altered:
None

FSR-Image Fields Generated:
None

Floating Absolute Value X10-form

fabs FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	517

FRT ← 0 || $(FRA)_{1:63}$

The contents of register FRA with bit 0 set to zero are placed into register FRT.

Special Registers Altered:
None

FSR-Image Fields Generated:
None

Floating Negate X10-form

fneg FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	520

FRT ← ¬$(FRA)_0$ || $(FRA)_{1:63}$

The contents of register FRA with bit 0 inverted are placed into register FRT.

Special Registers Altered:
None

FSR-Image Fields Generated:
None

Floating Negative Absolute Value X10-form

fnabs FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	519

FRT ← 1 || $(FRA)_{1:63}$

The contents of register FRA with bit 0 set to 1 are placed into register FRT.

Special Registers Altered:
None

FSR-Image Fields Generated:
None

6.6.2 Floating-Point Arithmetic Instructions

6.6.2.1 Floating-Point Elementary Arithmetic Instructions

Floating Add [Single] X10-form

fadd FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	256

fadds FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	257

The floating-point operand in register FRA is added to the floating-point operand in register FRB.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the sign of the operands, to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI

Floating Subtract [Single] X10-form

fsub FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	262

fsubs FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	263

The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

The execution of the *Floating Subtract* instruction is identical to that of *Floating Add*, except that the contents of FRB participates in the operation with its sign bit (bit 0) inverted.

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI

Floating Multiply [Single] X10-form

fmul FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	260

fmuls FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	261

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRB.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FSR-Image$_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR$_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ

Floating Divide [Single] X10-form

fdiv FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	258

fdivs FRT,FRA,FRB

0	4	10	16	22
0	FRT	FRA	FRB	259

The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

FSR-Image$_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR$_{VE}$=1 and Zero Divide Exceptions when FPSCR$_{ZE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

Floating Square-Root [Single] X10-form

fsqrt FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	524

fsqrts FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	525

The square-root of the floating-point operand in register FRA is placed into register FRT.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN [a]	VXSQRT
<0	QNaN [a]	VXSQRT
-0	-0	None
+∞	+∞	None
SNaN	QNaN[a]	VXSNAN
QNaN	QNaN	None

a. No result if FPSCRVE=1

FSR-Image$_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR$_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX XX
VXSNAN VXSQRT

Floating Reciprocal Estimate Single X10-form

fres FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	521

A single-precision estimate of the reciprocal of the floating-point operand in register FRA is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 256 of the reciprocal of (FRB), i.e.,

$$\mathrm{ABS}\left(\frac{\text{estimate} - 1/x}{1/x}\right) \leq \frac{1}{256}$$

where *x* is the initial value in FRB. Note that the value placed into register FRT may vary among implementations, and among different executions on the same implementation.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	-0	None
-0	-∞ [a]	ZX
+0	+∞ [a]	ZX
+∞	+0	None
SNaN	QNaN [b]	VXSNAN
QNaN	QNaN	None

a. No result if FPSCR$_{ZE}$=1
b. No result if FPSCR$_{VE}$=1

FSR-Image$_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR$_{VE}$=1 and Zero Divide Exceptions when FPSCR$_{ZE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR (undefined) FI (undefined)
FX OX UX ZX
VXSNAN

Architecture Note: No double-precision version of this instruction is provided because graphics applications are expected to need only the single-precision version, and no other important performance-critical applications are expected to need a double-precision version.

Floating Reciprocal Square-Root Estimate X10-form

frsqrte FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	523

A double-precision estimate of the reciprocal of the square-root of the floating-point operand in register FRA is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 32 of the reciprocal of the square-root of (FRB), i.e.,

$$\mathrm{ABS}\left(\frac{\text{estimate} - 1/(\sqrt{x})}{1/(\sqrt{x})}\right) \leq \frac{1}{32}$$

where *x* is the initial value in FRB. Note that the value placed into register FRT may vary among implementations, and among different executions on the same implementation.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN [a]	VXSQRT
<0	QNaN [a]	VXSQRT
-0	-∞ [b]	ZX
+0	+∞ [b]	ZX
+∞	+0	None
SNaN	QNaN [a]	VXSNAN
QNaN	QNaN	None

a. No result if $FPSCR_{VE}$=1

b. No result if $FPSCR_{ZE}$=1

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR (undefined) FI (undefined)
FX ZX
VXSNAN VXSQRT

Architecture Note: No single-precision version of this instruction is provided because it would be superfluous: if (FRB) is representable in single format, then so is (FRT).

6.6.2.2 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

Status bits in the FSR-Image are set as follows:

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (*fmul[s],* followed by *fadd[s]* or *fsub[s]*). That is, multiplication of infinity by 0 or multiplication of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Floating Multiply-Add [Single] X4-form

fmadd FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	4

fmadds FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	5

The operation

$$FRT \leftarrow [(FRA) \times (FRC)] + (FRB)$$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ

Floating Multiply-Subtract [Single] X4-form

fmsub FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	6

fmsubs FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	1

The operation

$$FRT \leftarrow [(FRA) \times (FRC)] - (FRB)$$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR and placed into register FRT.

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ

Floating Negative Multiply-Add [Single] X4-form

fnmadd FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	2

fnmadds FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	3

The operation

$$FRT \leftarrow -([(FRA) \times (FRC)] + (FRB))$$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FSR-Image$_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR$_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ

Floating Negative Multiply-Subtract [Single] X4-form

fnmsub FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	7

fnmsubs FRT,FRA,FRC,FRB

0	4	10	16	22	28
12	FRT	FRA	FRB	FRC	0

The operation

$$FRT \leftarrow -([(FRA) \times (FRC)] - (FRB))$$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

The result is normalized if the most significant bit of the resultant significand is not 1. The result is rounded to the target precision under control of the Floating-Point Round Control field RN of FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FSR-Image$_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR$_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ

6.6.3 Floating-Point Rounding and Conversion Instructions

Floating Convert to Integer Doubleword X10-form

fctid RT,FRA

0	4	10	16	22
0	RT	FRA	///	513

The floating-point operand in register FRA is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into fixed-point register RT.

If the operand in FRA is greater than $2^{63} - 1$, then RT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRA is less than -2^{63}, then RT is set to 0x8000_0000_0000_0000.

Except for enabled Invalid Operation Exceptions, FSR-$Image_{FPRF}$ is undefined. FSR-$Image_{FR}$ is set if the result is incremented when rounded. FSR-$Image_{FI}$ is set if the result is inexact.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

FSR-Image Fields Generated:
FPRF(undefined) FR FI
FX XX
VXSNAN VXCVI

Floating Convert to Integer Doubleword with round toward Zero X10-form

fctidz RT,FRA

0	4	10	16	22
0	RT	FRA	///	514

The floating-point operand in register FRA is converted to a 64-bit signed fixed-point integer, using the rounding mode *Round toward Zero*, and placed into fixed-point register RT.

If the operand in FRA is greater than $2^{63} - 1$, then RT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRA is less than -2^{63}, then RT is set to 0x8000_0000_0000_0000.

Except for enabled Invalid Operation Exceptions, FSR-$Image_{FPRF}$ is undefined. FSR-$Image_{FR}$ is set if the result is incremented when rounded. FSR-$Image_{FI}$ is set if the result is inexact.

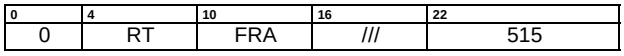
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
None

FSR-Image Fields Generated:
FPRF(undefined) FR FI
FX XX
VXSNAN VXCVI

Floating Convert to Integer Word X10-form

fctiw RT,FRA



0	4	10	16	22
0	RT	FRA	///	515

The floating-point operand in register FRA is converted to a 32-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into bits 32:63 of the fixed-point register RT. Bits 0:31 of register RT are undefined.

If the operand in FRA is greater than $2^{31} - 1$, then bits 32:63 of RT are set to 0x7FFF_FFFF. If the operand in FRA is less than -2^{31}, then bits 32:63 of RT are set to 0x8000_0000.

Except for enabled Invalid Operation Exceptions, FSR-$Image_{FPRF}$ is undefined. FSR-$Image_{FR}$ is set if the result is incremented when rounded. FSR-$Image_{FI}$ is set if the result is inexact.

Special Registers Altered:

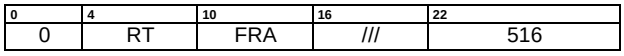
None

FSR-Image Fields Generated:

FPRF(undefined) FR FI
FX XX
VXSNAN VXCVI

Floating Convert to Integer Word with round toward Zero X10-form

fctiwz RT,FRA



0	4	10	16	22
0	RT	FRA	///	516

The floating-point operand in register FRA is converted to a 32-bit signed fixed-point integer, using the rounding mode *Round toward Zero*, and placed into bits 32:63 of fixed-point register RT. Bits 0:31 of register RT are undefined.

If the operand in FRA is greater than $2^{31} - 1$, then bits 32:63 of RT are set to 0x7FFF_FFFF. If the operand in FRA is less than -2^{31}, then bits 32:63 of RT are set to 0x8000_0000.

Except for enabled Invalid Operation Exceptions, FSR-$Image_{FPRF}$ is undefined. FSR-$Image_{FR}$ is set if the result is incremented when rounded. FSR-$Image_{FI}$ is set if the result is inexact.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF(undefined) FR FI
FX XX
VXSNAN VXCVI

Floating Convert From Integer Doubleword X10-form

fcfid FRT,RA

0	4	10	16	22
0	FRT	RA	///	512

The 64-bit signed fixed-point operand in fixed-point register RA is converted to an infinitely precise floating-point operand. If the result of the conversion is already in double-precision range, it is placed into floating-point register FRT. Otherwise, the result of the conversion is rounded to double-precision using the rounding mode specified by $FPSCR_{RN}$, and placed into floating-point register FRT.

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result. $FSR\text{-}Image_{FR}$ is set if the result is incremented when rounded. $FSR\text{-}Image_{FI}$ is set if the result is inexact.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX XX

Floating Round to Single-Precision X10-form

frsp FRT,FRA

0	4	10	16	22
0	FRT	FRA	///	522

If it is already in single precision range, the floating-point operand in register FRA is placed into register FRTs. Otherwise, the floating-point operand in register FRA is rounded to single-precision using the rounding mode specified by $FPSCR_{RN}$ and placed into register FRT.

$FSR\text{-}Image_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

None

FSR-Image Fields Generated:

FPRF FR FI
FX OX UX XX
VXSNAN

6.6.4 Floating-Point Compare Instructions

The floating-point compare instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to -0). The comparison can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1, and the other three to 0.

The CR field specified by the instruction is interpreted as follows:

Bit	Name	Description
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Floating Compare Unordered X10-form

fcmpu CRT,FRA,FRB

0	4	8	10	16	22
0	CRT	//	FRA	FRB	265

```
if (FRA) is a NaN or
   (FRB) is a NaN then     c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                       c ← 0b0010
CR_CRT ← c

if (FRA) is a SNaN or
   (FRB) is a SNaN then
     FSR-Image_VXSNAN ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field CRT.

If either of the operands is a NaN, either quiet or signaling, then CR field CRT is set to reflect unordered. If either of the operands is a Signalling NaN, then FSR-Image$_{VXSNAN}$ is set.

Special Registers Altered:
CR Field CRT

FSR-Image Fields Generated:
FX
VXSNAN

Floating Compare Ordered X10-form

fcmpo CRT,FRA,FRB

0	4	8	10	16	22
0	CRT	//	FRA	FRB	264

```
if (FRA) is a NaN or
   (FRB) is a NaN then     c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                       c ← 0b0010
CR_CRT ← c

if (FRA) is a SNaN or
   (FRB) is a SNaN then
     FSR-Image_VXSNAN ← 1
     if VE=0 then FSR-Image_VXVC ← 1
else if (FRA) is a QNaN or
     (FRB) is a QNaN then FSR-Image_VXVC ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field CRT.

If either of the operands is a NaN, either quiet or signaling, then CR field CRT is set to reflect unordered. If either of the operands is a Signalling NaN, then FSR-Image$_{VXSNAN}$ is set and, if Invalid Operation is disabled (VE=0), FSR-Image$_{VXVC}$ is set. If neither operand is a Signalling NaN but at least one operand is a Quiet NaN, then FSR-Image$_{VXVC}$ is set.

Special Registers Altered:

CR Field CRT

FSR-Image Fields Generated:

FX
VXSNAN VXVC

6.6.5 Floating-Point Select Instruction

Floating Select X4-form

fsel FRT,FRA,FRC,FRB

0	4	10	16	22	28
11	FRT	FRA	FRB	FRC	15

```
if (FRA) ≥ 0.0  then FRT ← (FRC)
else                 FRT ← (FRB)
```

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e. regards +0 as equal to -0).

Special Registers Altered:

None

FSR-Image Fields Generated:

None

Architecture Note: The select instruction is similar to a *Move* Instruction, and therefore does not alter FPRF

Warning Note: Care must be taken in using *fsel* if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

Appendix A. Book II and Book III Instructions

The following instructions are described in *Book II, Foresta Virtual Environment Architecture* and *Book III, Foresta Operating Environment Architecture*:

dcbt	Data Cache Block Touch
dcbtst	Data Cache Block Touch for Store
dcbi	Data Cache Block Invalidate
dcbf	Data Cache Block Flush
dcbst	Data Cache Block Store
dcbz	Data Cache Block Set to Zero
eciw	External Control In Word
ecow	External Control Out Word
eieio	Enforce In-order Execution of I/O
isync	Instruction Synchronize
icbi	Instruction Cache Block Invalidate
icbt	Instruction Cache Block Touch
mftb	Move From Time-Base Register
mfmsr	Move from Machine State Register
mtmsr	Move to Machine State Register
rfi	Return From Interrupt
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
slbiex	SLB Invalidate Entry by Index
tlbia	TLB Invalidate All
tlbie	TLB Invalidate Entry
tlbiex	TLB Invalidate Entry by Index
tlbsync	TLB Synchronize

Appendix B. ForestaPC User Instruction Set Sorted by Opcode

This appendix lists all the instructions in the ForestaPC Architecture. A page number is shown for instructions that are defined in this Book (*Book I, ForestaPC User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (*Book II, ForestaPC Virtual Environment Architecture*, *Book III, ForestaPC Operating Environment Architecture*). If an instruction is defined in more than one Book, the lowest-numbered Book is used.

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
0	1	X10	nop	No-operation	93
0	128	X10	crand	Condition Register AND	61
0	129	X10	crandc	Condition Register AND with Complement	63
0	130	X10	creqv	Condition Register Equivalent	62
0	131	X10	crnand	Condition Register NAND	62
0	132	X10	crnor	Condition Register NOR	62
0	133	X10	cror	Condition Register OR	61
0	134	X10	crorc	Condition Register OR with Complement	63
0	135	X10	crxor	Condition Register XOR	62
0	192	X10	divd	Divide Doubleword	79
0	193	X10	divdu	Divide Doubleword Unsigned	81
0	194	X10	divw	Divide Word	80
0	195	X10	divwu	Divide Word Unsigned	82
0	223	B10	skip	Skip Conditional	34
0	224	B10	icbi	Instruction Cache Block Invalidate	145
0	225	B10	icbt	Instruction Cache Block Touch	145
0	256	X10	fadd	Floating Add	134
0	257	X10	fadds	Floating Add Single	134
0	258	X10	fdiv	Floating Divide	135
0	259	X10	fdivs	Floating Divide Single	135
0	260	X10	fmul	Floating Multiply	135
0	261	X10	fmuls	Floating Multiply	135
0	262	X10	fsub	Floating Subtract	134
0	263	X10	fsubs	Floating Subtract Single	134
0	264	X10	fcmpo	Floating Compare Ordered	144
0	265	X10	fcmpu	Floating Compare Unordered	143
0	272	X10	mulhd	Multiply High Doubleword	77
0	273	X10	mulhdu	Multiply High Doubleword Unsigned	77
0	274	X10	mulhw	Multiply High Word	78
0	275	X10	mulhwu	Multiply High Word Unsigned	78
0	276	X10	mulld	Multiply Low Doubleword	76
0	277	X10	mullw	Multiply Low Word	76
0	288	X10	slsd	Shift Left String Doubleword	105
0	289	X10	slsw	Shift Left String Word	105
0	290	X10	srsd	Shift Right String Doubleword	106
0	291	X10	srsw	Shift Right String Word	106
0	304	X10	cmp	Compare	84
0	305	X10	cmpl	Compare Logical	85
0	306	X10	cntlzd	Count Leading Zeros Doubleword	94
0	307	X10	cntlzw	Count Leading Zeros Word	94
0	308	X10	extsb	Extend Sign Byte	92
0	309	X10	extsh	Extend Sign Halfword	92

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
0	310	X10	extsw	Extend Sign Word	93
0	313	X10	td	Trap Doubleword	87
0	314	X10	tw	Trap Word	87
0	512	X10	fcfid	Floating Convert From Integer Doubleword	142
0	513	X10	fctid	Floating Convert to Integer Doubleword	140
0	514	X10	fctidz	Floating Convert to Integer Doubleword with round toward Zero	140
0	515	X10	fctiw	Floating Convert to Integer Word	141
0	516	X10	fctiwz	Floating Convert to Integer Word with round toward Zero	141
0	517	X10	fabs	Floating Absolute Value	133
0	518	X10	fmr	Floating Move Register	133
0	519	X10	fnabs	Floating Negative Absolute Value	133
0	520	X10	fneg	Floating Negate	133
0	521	X10	fres	Floating Reciprocal Estimate Single	136
0	522	X10	frsp	Floating Round to Single-Precision	142
0	523	X10	frsqrte	Floating Reciprocal Square-Root Estimate	137
0	524	X10	fsqrt	Floating Square-Root	136
0	525	X10	fsqrts	Floating Square-Root Single	136
0	768	X10	xadd	Extend Add	70
0	769	X10	xsub	Extend Subtract	71
0	770	X10	xfps	Extend FSR	69
0	771	X10	xsrx	Extend XSR	68
0	772	X10	xsrxe	Extended Extend XSR	68
0	773	X10	xtf	Extend FSR and Trap	70
0	774	X10	xtx	Extend XSR and Trap	69
0	775	D10	xstb	Extend Store Byte	57
0	776	D10	xstd	Extend Store Doubleword	55
0	777	D10	xsth	Extend Store Halfword	56
0	778	D10	xstw	Extend Store Word	56
0	779	B10	xcst	Extend Conditional Store	55
0	782	X10	uxsr	Update XSR From Image	109
0	783	X10	ufsr	Update FPSCR From Image	111
0	784	X10	mtspr	Move To Special-Purpose Register	108
0	785	X10	mfspr	Move From Special-Purpose Register	109
0	786	X10	mftb	Move From Time-Base Register	145
0	787	X10	mtfsf	Move To FPSCR Fields	112
0	788	X10	mtfsfi	Move To FPSCR Field Immediate	111
0	789	X10	mffs	Move From FPSCR	110
0	790	X10	mtfsb0	Move To FPSCR Bit 0	112
0	791	X10	mtfsb1	Move To FPSCR Bit 1	113
0	792	X10	csr	Commit Speculative Register	115
0	793	X10	csfr	Commit Speculative FPR	115
0	794	X10	eciw	External Control In Word	145

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
0	795	X10	ecow	External Control Out Word	145
0	796	X10	mffpr	Move from Floating-Point Register	114
0	797	X10	mtfpr	Move to Floating-Point Register	114
0	798	X10	tdi	Trap Doubleword Immediate	86
0	799	X10	twi	Trap Word Immediate	86
0	800	X10	mfcrf	Move from Condition Register Field	64
0	801	X10	mtcrf	Move to Condition Register Field	64
0	802	X10	mcrf	Move Condition Register Field	64
0	803	X10	mcrfi	Move Condition Register Field Immediate	64
0	804	X10	mcrfs	Move to Condition Register From FPSCR	110
0	805	X10	mfcrw	Move From Condition Register Word	65
0	806	X10	mtcrw	Move to Condition Register Word	66
0	808	X10	mfmsr	Move from Machine State Register	145
0	809	X10	mtmsr	Move to Machine State Register	145
0	810	X10	slbie	SLB Invalidate Entry	145
0	811	X10	slbiex	SLB Invalidate Entry by Index	145
0	812	X10	slbia	SLB Invalidate All	145
0	813	X10	tlbie	TLB Invalidate Entry	145
0	814	X10	tlbiex	TLB Invalidate Entry by Index	145
0	815	X10	tlbia	TLB Invalidate All	145
0	816	X10	mbr	Move Branch Register	60
0	817	X10	mcrxr	Move to Condition Register from XSR	109
0	818	X10	br	Branch Register	34
0	819	X10	eieio	Enforce In-order Execution of I/O	145
0	820	X10	isync	Instruction Synchronize	145
0	822	X10	rfi	Return From Interrupt	145
0	823	X10	sc	System Call	35
0	825	X10	sync	Synchronize	54
0	826	X10	tlbsync	TLB Synchronize	145
0	835	X10	mfcr	Move From Condition Register	65
0	836	X10	mtcr	Move to Condition Register	66
1		I0	addi	Add Immediate	73
2		I0	subfi	Subtract from Immediate	74
3		I0	mulli	Multiply Low Immediate	75
4		I0	andi	AND Immediate	89
5		I0	ori	OR Immediate	90
6		I0	xori	XOR Immediate	90
7		M0	rlwnm	Rotate Left Word then AND with Mask	99
8	0	I1	cmpi	Compare Immediate	83
8	1	I1	cmpli	Compare Logical Immediate	84
9	0	M1	rlwinm	Rotate Left Word Immediate then AND with Mask	97
10	0	B2	b	Branch Unconditional	34

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
10	1	B2	cbri	Compute Branch Register Immediate	60
11	0	D4	lhbr	Load Halfword Byte-Reversed	44
11	1	D4	lhz	Load Halfword and Zero	39
11	2	D4	lwz	Load Word and Zero	40
11	3	D4	lsd	Load String Doubleword	48
11	4	D4	lwbr	Load Word Byte-Reversed	44
11	5	D4	lha	Load Halfword Algebraic	39
11	6	D4	lbz	Load Byte and Zero	39
11	7	D4	ld	Load Doubleword	40
11	8	D4	lswz	Load String Word and Zero	47
11	9	D4	lwa	Load Word Algebraic	40
11	10	D4	ltocd	Load TOC Doubleword	46
11	11	D4	ltocwz	Load TOC Word and Zero	46
11	12	D4	lfs	Load Floating-Point Single	42
11	13	D4	lfd	Load Floating-Point Double	42
11	14	X4	rldicr	Rotate Left Doubleword Immediate then Clear Right	96
11	15	X4	fsel	Floating Select	144
12	0	X4	fnmsubs	Floating Negative Multiply-Subtract Single	139
12	1	X4	fmsubs	Floating Multiply-Subtract Single	138
12	2	X4	fnmadd	Floating Negative Multiply-Add	139
12	3	X4	fnmadds	Floating Negative Multiply-Add Single	139
12	4	X4	fmadd	Floating Multiply-Add	138
12	5	X4	fmadds	Floating Multiply-Add Single	138
12	6	X4	fmsub	Floating Multiply-Subtract	138
12	7	X4	fnmsub	Floating Negative Multiply-Subtract	139
12	8	X4	selii	Select Immediate-Immediate	88
12	9	X4	selir	Select Immediate-Register	88
12	10	X4	selri	Select Register-Immediate	89
12	11	X4	selrr	Select Register-Register	89
12	12	X4	rldcl	Rotate Left Doubleword then Clear Left	98
12	13	X4	rldcr	Rotate Left Doubleword then Clear Right	99
12	14	X4	rldic	Rotate Left Doubleword Immediate then Clear	97
12	15	X4	rldicl	Rotate Left Doubleword Immediate then Clear Left	96
13	0	D5	sthbr	Store Halfword Byte-Reversed	45
13	1	D5	sth	Store Halfword	41
13	2	D5	stwcr	Store Word Conditional Reserve	52
13	3	D5	stb	Store Byte	41
13	4	D5	stwbr	Store Word Byte-Reversed	45
13	5	D5	stsw	Store String Word	48
13	6	D5	stw	Store Word	41
13	7	D5	stsd	Store String Doubleword	49
13	8	D5	std	Store Doubleword	41

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
13	9	D5	stdcr	Store Doubleword Conditional Reserve	53
13	10	D5	stfs	Store Floating-Point Single	43
13	11	D5	stfd	Store Floating-Point Double	43
13	12	D5	lwar	Load Word and Reserve	51
13	13	D5	ldar	Load Doubleword and Reserve	51
14	0	X6	sradi	Shift Right Algebraic Doubleword Immediate	102
14	1	X6	srawi	Shift Right Algebraic Word Immediate	103
14	2	X6	srad	Shift Right Algebraic Doubleword	103
14	3	X6	sraw	Shift Right Algebraic Word	104
14	4	X6	srd	Shift Right Doubleword	101
14	5	X6	srw	Shift Right Word	102
14	6	X6	sld	Shift Left Doubleword	100
14	7	X6	slw	Shift Left Word	101
14	8	X6	and	AND	90
14	9	X6	xor	XOR	91
14	10	X6	nand	NAND	91
14	11	X6	nor	NOR	91
14	12	X6	or	OR	90
14	13	X6	andc	AND with Complement	92
14	14	X6	orc	OR with Complement	92
14	15	X6	eqv	Equivalent	91
14	16	X6	add	Add	73
14	17	X6	subf	Subtract From	74
14	32	X6	sldia	Shift Left Doubleword Immediate then Add	107
14	33	X6	slwia	Shift Left Word Immediate then Add	107
15	0	I8	xicr	Extend Immediate and Condition Register	67
15	1	X8	csrcr	Commit Speculative and Condition Register Field	115
15	16	D8	dcbtst	Data Cache Block Touch for Store	145
15	17	D8	dcbt	Data Cache Block Touch	145
15	18	D8	dcbi	Data Cache Block Invalidate	145
15	19	D8	dcbf	Data Cache Block Flush	145
15	20	D8	dcbst	Data Cache Block Store	145
15	21	D8	dcbz	Data Cache Block Set to Zero	145

Appendix C. ForestaPC User Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the ForestaPC Architecture. A page number is shown for instructions that are defined in this Book (*Book I, ForestaPC User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (*Book II, ForestaPC Virtual Environment Architecture*, *Book III, ForestaPC Operating Environment Architecture*). If an instruction is defined in more than one Book, the lowest-numbered Book is used.

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
14	16	X6	add	Add	73
1		I0	addi	Add Immediate	73
14	8	X6	and	AND	90
14	13	X6	andc	AND with Complement	92
4		I0	andi	AND Immediate	89
10	0	B2	b	Branch Unconditional	34
0	818	X10	br	Branch Register	34
10	1	B2	cbri	Compute Branch Register Immediate	60
0	304	X10	cmp	Compare	84
8	0	I1	cmpi	Compare Immediate	83
0	305	X10	cmpl	Compare Logical	85
8	1	I1	cmpli	Compare Logical Immediate	84
0	306	X10	cntlzd	Count Leading Zeros Doubleword	94
0	307	X10	cntlzw	Count Leading Zeros Word	94
0	128	X10	crand	Condition Register AND	61
0	129	X10	crandc	Condition Register AND with Complement	63
0	130	X10	creqv	Condition Register Equivalent	62
0	131	X10	crnand	Condition Register NAND	62
0	132	X10	crnor	Condition Register NOR	62
0	133	X10	cror	Condition Register OR	61
0	134	X10	crorc	Condition Register OR with Complement	63
0	135	X10	crxor	Condition Register XOR	62
0	793	X10	csfr	Commit Speculative FPR	115
0	792	X10	csr	Commit Speculative Register	115
15	1	X8	csrcr	Commit Speculative Register and Condition Register Field	115
15	19	D8	dcbf	Data Cache Block Flush	145
15	18	D8	dcbi	Data Cache Block Invalidate	145
15	20	D8	dcbst	Data Cache Block Store	145
15	17	D8	dcbt	Data Cache Block Touch	145
15	16	D8	dcbtst	Data Cache Block Touch for Store	145
15	21	D8	dcbz	Data Cache Block Set to Zero	145
0	192	X10	divd	Divide Doubleword	79
0	193	X10	divdu	Divide Doubleword Unsigned	81
0	194	X10	divw	Divide Word	80
0	195	X10	divwu	Divide Word Unsigned	82
0	794	X10	eciw	External Control In Word	145
0	795	X10	ecow	External Control Out Word	145
0	819	X10	eieio	Enforce In-order Execution of I/O	145
14	15	X6	eqv	Equivalent	91
0	308	X10	extsb	Extend Sign Byte	92
0	309	X10	extsh	Extend Sign Halfword	92
0	310	X10	extsw	Extend Sign Word	93

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
0	517	X10	fabs	Floating Absolute Value	133
0	256	X10	fadd	Floating Add	134
0	257	X10	fadds	Floating Add Single	134
0	512	X10	fcfid	Floating Convert From Integer Doubleword	142
0	264	X10	fcmpo	Floating Compare Ordered	144
0	265	X10	fcmpu	Floating Compare Unordered	143
0	513	X10	fctid	Floating Convert to Integer Doubleword	140
0	514	X10	fctidz	Floating Convert to Integer Doubleword with round toward Zero	140
0	515	X10	fctiw	Floating Convert to Integer Word	141
0	516	X10	fctiwz	Floating Convert to Integer Word with round toward Zero	141
0	258	X10	fdiv	Floating Divide	135
0	259	X10	fdivs	Floating Divide Single	135
12	4	X4	fmadd	Floating Multiply-Add	138
12	5	X4	fmadds	Floating Multiply-Add Single	138
0	518	X10	fmr	Floating Move Register	133
12	6	X4	fmsub	Floating Multiply-Subtract	138
12	1	X4	fmsubs	Floating Multiply-Subtract Single	138
0	260	X10	fmul	Floating Multiply	135
0	261	X10	fmuls	Floating Multiply Single	135
0	519	X10	fnabs	Floating Negative Absolute Value	133
0	520	X10	fneg	Floating Negate	133
12	2	X4	fnmadd	Floating Negative Multiply-Add	139
12	3	X4	fnmadds	Floating Negative Multiply-Add Single	139
12	7	X4	fnmsub	Floating Negative Multiply-Subtract	139
12	0	X4	fnmsubs	Floating Negative Multiply-Subtract Single	139
0	521	X10	fres	Floating Reciprocal Estimate Single	136
0	522	X10	frsp	Floating Round to Single-Precision	142
0	523	X10	frsqrte	Floating Reciprocal Square-Root Estimate	137
11	15	X4	fsel	Floating Select	144
0	524	X10	fsqrt	Floating Square-Root	136
0	525	X10	fsqrts	Floating Square-Root Single	136
0	262	X10	fsub	Floating Subtract	134
0	263	X10	fsubs	Floating Subtract Single	134
0	224	B10	icbi	Instruction Cache Block Invalidate	145
0	225	B10	icbt	Instruction Cache Block Touch	145
0	820	X10	isync	Instruction Synchronize	145
11	6	D4	lbz	Load Byte and Zero	39
11	7	D4	ld	Load Doubleword	40
13	13	D5	ldar	Load Doubleword and Reserve	51
11	13	D4	lfd	Load Floating-Point Double	42
11	12	D4	lfs	Load Floating-Point Single	42
11	5	D4	lha	Load Halfword Algebraic	39

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
11	0	D4	lhbr	Load Halfword Byte-Reversed	44
11	1	D4	lhz	Load Halfword and Zero	39
11	3	D4	lsd	Load String Doubleword	48
11	8	D4	lswz	Load String Word and Zero	47
11	10	D4	ltocd	Load TOC Doubleword	46
11	11	D4	ltocwz	Load TOC Word and Zero	46
11	9	D4	lwa	Load Word Algebraic	40
13	12	D5	lwar	Load Word and Reserve	51
11	4	D4	lwbr	Load Word Byte-Reversed	44
11	2	D4	lwz	Load Word and Zero	40
0	816	X10	mbr	Move Branch Register	60
0	802	X10	mcrf	Move Condition Register Field	64
0	803	X10	mcrfi	Move Condition Register Field Immediate	64
0	804	X10	mcrfs	Move to Condition Register From FPSCR	110
0	817	X10	mcrxr	Move to Condition Register from XSR	109
0	835	X10	mfcr	Move From Condition Register	65
0	800	X10	mfcrf	Move from Condition Register Field	64
0	805	X10	mfcrw	Move From Condition Register Word	65
0	796	X10	mffpr	Move from Floating-Point Register	114
0	789	X10	mffs	Move From FPSCR	110
0	808	X10	mfmsr	Move from Machine State Register	145
0	785	X10	mfspr	Move From Special-Purpose Register	109
0	786	X10	mftb	Move From Time-Base Register	145
0	836	X10	mtcr	Move to Condition Register	66
0	801	X10	mtcrf	Move to Condition Register Field	64
0	806	X10	mtcrw	Move to Condition Register Word	66
0	797	X10	mtfpr	Move to Floating-Point Register	114
0	790	X10	mtfsb0	Move To FPSCR Bit 0	112
0	791	X10	mtfsb1	Move To FPSCR Bit 1	113
0	787	X10	mtfsf	Move To FPSCR Fields	112
0	788	X10	mtfsfi	Move To FPSCR Field Immediate	111
0	809	X10	mtmsr	Move to Machine State Register	145
0	784	X10	mtspr	Move To Special-Purpose Register	108
0	272	X10	mulhd	Multiply High Doubleword	77
0	273	X10	mulhdu	Multiply High Doubleword Unsigned	77
0	274	X10	mulhw	Multiply High Word	78
0	275	X10	mulhwu	Multiply High Word Unsigned	78
0	276	X10	mulld	Multiply Low Doubleword	76
3		I0	mulli	Multiply Low Immediate	75
0	277	X10	mullw	Multiply Low Word	76
14	10	X6	nand	NAND	91
0	1	X10	nop	No-operation	93

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
14	11	X6	nor	NOR	91
14	12	X6	or	OR	90
14	14	X6	orc	OR with Complement	92
5		I0	ori	OR Immediate	90
0	822	X10	rfi	Return From Interrupt	145
12	12	X4	rldcl	Rotate Left Doubleword then Clear Left	98
12	13	X4	rldcr	Rotate Left Doubleword then Clear Right	99
12	14	X4	rldic	Rotate Left Doubleword Immediate then Clear	97
12	15	X4	rldicl	Rotate Left Doubleword Immediate then Clear Left	96
11	14	X4	rldicr	Rotate Left Doubleword Immediate then Clear Right	96
9	0	M1	rlwinm	Rotate Left Word Immediate then AND with Mask	97
7		M0	rlwnm	Rotate Left Word then AND with Mask	99
0	823	X10	sc	System Call	35
12	8	X4	selii	Select Immediate-Immediate	88
12	9	X4	selir	Select Immediate-Register	88
12	10	X4	selri	Select Register-Immediate	89
12	11	X4	selrr	Select Register-Register	89
0	223	B10	skip	Skip Conditional	34
0	812	X10	slbia	SLB Invalidate All	145
0	810	X10	slbie	SLB Invalidate Entry	145
0	811	X10	slbiex	SLB Invalidate Entry by Index	145
14	6	X6	sld	Shift Left Doubleword	100
14	32	X6	sldia	Shift Left Doubleword Immediate then Add	107
0	288	X10	slsd	Shift Left String Doubleword	105
0	289	X10	slsw	Shift Left String Word	105
14	7	X6	slw	Shift Left Word	101
14	33	X6	slwia	Shift Left Word Immediate then Add	107
14	2	X6	srad	Shift Right Algebraic Doubleword	103
14	0	X6	sradi	Shift Right Algebraic Doubleword Immediate	102
14	3	X6	sraw	Shift Right Algebraic Word	104
14	1	X6	srawi	Shift Right Algebraic Word Immediate	103
14	4	X6	srd	Shift Right Doubleword	101
0	290	X10	srsd	Shift Right String Doubleword	106
0	291	X10	srsw	Shift Right String Word	106
14	5	X6	srw	Shift Right Word	102
13	3	D5	stb	Store Byte	41
13	8	D5	std	Store Doubleword	41
13	9	D5	stdc	Store Doubleword Conditional Reserve	53
13	11	D5	stfd	Store Floating-Point Double	43
13	10	D5	stfs	Store Floating-Point Single	43
13	1	D5	sth	Store Halfword	41
13	0	D5	sthbr	Store Halfword Byte-Reversed	45

Opcode Prim.	Ext.	Form	Mnemonic	Instruction	Page
13	7	D5	stsd	Store String Doubleword	49
13	5	D5	stsw	Store String Word	48
13	6	D5	stw	Store Word	41
13	4	D5	stwbr	Store Word Byte-Reversed	45
13	2	D5	stwc	Store Word Conditional Reserve	52
14	17	X6	subf	Subtract From	74
2		I0	subfi	Subtract from Immediate	74
0	825	X10	sync	Synchronize	54
0	313	X10	td	Trap Doubleword	87
0	798	X10	tdi	Trap Doubleword Immediate	86
0	815	X10	tlbia	TLB Invalidate All	145
0	813	X10	tlbie	TLB Invalidate Entry	145
0	814	X10	tlbiex	TLB Invalidate Entry by Index	145
0	826	X10	tlbsync	TLB Synchronize	145
0	314	X10	tw	Trap Word	87
0	799	X10	twi	Trap Word Immediate	86
0	783	X10	ufsr	Update FPSCR From Image	111
0	782	X10	uxsr	Update XSR From Image	109
0	768	X10	xadd	Extend Add	70
0	779	B10	xcst	Extend Conditional Store	55
0	770	X10	xfps	Extend FSR	69
15	0	I8	xicr	Extend Immediate and Condition Register	67
14	9	X6	xor	XOR	91
6		I0	xori	XOR Immediate	90
0	771	X10	xsrx	Extend XSR	68
0	772	X10	xsrxe	Extended Extend XSR	68
0	775	D10	xstb	Extend Store Byte	57
0	776	D10	xstd	Extend Store Doubleword	55
0	777	D10	xsth	Extend Store Halfword	56
0	778	D10	xstw	Extend Store Word	56
0	769	X10	xsub	Extend Subtract	71
0	773	X10	xtf	Extend FSR and Trap	70
0	774	X10	xtx	Extend XSR anf Trap	69