PA-RISC 2.0 Architecture

This document is an excerpt from PA-RISC 2.0 ARCHITECTURE, by Gerry Kane, published by Prentice Hall PTR, isbn 0-13-182734-0. Print copies of the book can be ordered at www.hp.com/hpbooks.

This document contains:
Chapter 6: Instruction Set Overview
This chapter provides an overview of the PA-RISC instruction set. The instructions can be divided into the following functional groups:

- Computation instructions.
- Multimedia instructions.
- Memory Reference instructions.
- Long Immediate instructions.
- Branch instructions.
- System Control instructions.
- Assist instructions.

The instruction set consists of defined, undefined, illegal, and null instructions. This chapter discusses the concepts of undefined and null instructions and includes descriptions of the conditions, their completers, and the notation used in the instruction descriptions. Each instruction is described in detail in Chapter 7, “Instruction Descriptions”. Each description includes the full name of the instruction, the assembly language mnemonic and syntax format, machine instruction format, purpose, a narrative description, an operational description, exceptions, and notes concerning usage. In some cases, programming notes are included for additional guidance to programmers.

Instructions are always 32 bits in width. A 6-bit major opcode is always the first field. Source registers, if specified, are often the next two 5-bit fields and are always in the same place. Target registers, if specified, are not fixed in any particular 5-bit field. Depending on the major opcode, the remainder of the instruction word is divided into fields that specify immediate values, space registers, additional opcode extensions, conditions, and nullification.

**Computation Instructions**

Computation instructions are comprised of the arithmetic, logical, shift, extract, and deposit instructions which operate on the general registers. The two 5-bit fields following the 6-bit opcode field can specify the following combinations:

1. Two source registers.
2. A source register and a target register.
3. A source register and a 5-bit immediate.
4. A target register and a 5-bit immediate.

Table 6-1 summarizes the computation instructions that are provided.
Table 6-1. Computation Instruction Summary

<table>
<thead>
<tr>
<th>3-Register Arithmetic &amp; Logical Instructions</th>
<th>Perform arithmetic and logical operations with two operands in registers and store the result in a third register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SHLADD, SUB, OR, XOR, AND, ANDCM, UADDCM, UXOR, DS, CMPCLR, DCOR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Immediate Arithmetic Instructions</th>
<th>Perform arithmetic operations between a sign-extended immediate and the contents of a register. The result is placed in a register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI, SUBI, CMPICLR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shift Pair, Extract &amp; Deposit Instructions</th>
<th>The shift pair operations allow for a concatenation of two registers followed by a shift of 0 to 63 bit positions. Extract instructions take a field from a source register and insert it right-justified into the target register. Deposits either set the target to zero or leave it unchanged (merge operation). The deposit instructions then take a right-justified field from a source and deposit it into any portion of the target.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHRPD, SHRPW, EXTRD, EXTRW, DEPD, DEPDI, DEPW, DEPWI</td>
<td></td>
</tr>
</tbody>
</table>

The three-register arithmetic and logical instructions take two source arguments from two general registers. These source registers are specified by the two 5-bit fields following the opcode specifier. The rightmost 5-bit field specifies the target register.

Some of the computation instructions have a signed immediate argument which is either five bits or eleven bits in length. The 5-bit immediate is encoded in the second 5-bit field following the opcode field and the target specifier in the first 5-bit field following the opcode field. The 11-bit immediate is encoded in the rightmost 11-bit field, and the target specifier in the second 5-bit field following the opcode specifier.

Many computation instructions may nullify the instruction following, given the correct conditions. The instruction condition completers are used to determine if the instruction following is nullified, based on the contents of the source operands and the operation performed.

Three-Register Arithmetic and Logical Instructions

These instructions perform arithmetic and logical operations between two operands in registers and store the result into a register. Each arithmetic/logical instruction also specifies the conditional occurrence of either a skip or a trap, based on its opcode and the condition field. Not all options are available on every instruction. Only those operations and options considered useful are defined.

Immediate Arithmetic Instructions

The immediate arithmetic instructions operate between a sign-extended 11-bit immediate and the contents of a register. The result is placed in a register. Immediate operations may optionally trap on overflow. In addition, immediate adds may trap on a specific condition.
The 11-bit immediate field has the sign bit in the rightmost position, but the other 10 bits are in the usual order. The 1-bit opcode extension field determines whether overflow causes a trap.

### Shift Pair, Extract, and Deposit Instructions

The shift pair operations allow for a concatenation of two registers followed by a shift of 0 to 63 bit positions. The rightmost 64 bits are placed in a general register. Depending on the choice of the source registers, this operation allows the user to perform right or left shifts, rotates, bit field extractions when the bit field crosses word boundaries, unaligned byte moves, and so on.

Extract instructions take a field from a source register and insert it right-justified into the target register. This field is either zero extended or sign extended. This way, the extract instructions support both logical and arithmetic shift operations.

The deposit instructions either set the target to zero or leave it unchanged (merge operation) and then take a right-justified field from a source and deposit it into any portion of the target. The source can be either a register or a 5-bit signed immediate value. The 5-bit immediate field has the sign bit in the rightmost position, but the other 4 bits are in the usual order. Deposit instructions support left shift operations and simple multiplication by powers of two.

### Multimedia Instructions

PA-RISC provides efficient support for the most frequent multimedia operations because these operations are assuming greater importance in many applications. Instructions in this category perform multiple parallel operations in a single cycle.

In multimedia workloads, a large portion of the arithmetic can be thought of as saturation arithmetic. This means that if the result of a calculation would be too large in magnitude to be represented in a given format, the calculation delivers the largest representable number (as opposed to wrapping to the other end of the representable range, as with modular arithmetic). Commonly, this must be implemented by testing each result (twice, for signed results), and performing conditional branches or skips to force the result to a maximum value. The multimedia instructions in PA-RISC perform multiple parallel computations, with each of the results being tested and forced to the appropriate value if necessary, in a single cycle. The result is a sizeable reduction in pathlength and fewer disruptive breaks in control flow in multimedia algorithms.

Re-arrangement instructions provide efficient support for packed pixel data structures in memory, allow algorithms to make full use of the parallel computation instructions, and enable the use of the full load/store bandwidth of the processor in accessing pixel data.

The multimedia instructions are comprised of halfword arithmetic, halfword shift, and rearrangement instructions. The instructions operate on 16-bit signed or unsigned numbers. The signed representations are two's complement numbers in the range \(-2^{15}\) to \(2^{15} - 1\). Table 6-2 summarizes the multimedia instructions provided.
Parallel Halfword Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HADD, HSUB, HAVG</td>
<td>Parallel halfword add, subtract, and average instructions operate on two 64-bit registers, each containing four 16-bit operands.</td>
</tr>
</tbody>
</table>

Parallel Halfword Shift Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSHLADD, HSHRADD,</td>
<td>Parallel halfword shift instructions allow multiple halfword shift operations with the shift amount encoded in the instruction. Bits are blocked from being shifted across halfword boundaries. The parallel halfword shift-and-add instructions support halfword multiplication and division by constants.</td>
</tr>
<tr>
<td>HSHL, HSHR</td>
<td></td>
</tr>
</tbody>
</table>

Rearrangement Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERMH, MIXH, MIXW</td>
<td>These instructions allow full utilization of halfword parallel instructions by supporting rearrangement of words and halfwords in registers with no memory load/store overhead.</td>
</tr>
</tbody>
</table>

Parallel Halfword Arithmetic Instructions

In multimedia applications, the most common operations on pixels are addition, subtraction, averaging and multiplication (especially multiplication by constants).

The HALFWORD ADD, HALFWORD SUBTRACT and HALFWORD AVERAGE instructions operate on two 64-bit general registers, each containing four 16-bit operands, and produce four 16-bit results, delivered to a general register. Saturation can be optionally performed. For the HALFWORD AVERAGE instruction, unbiased rounding is performed, to reduce the accumulation of rounding errors.

Halfword multiplication and division by constants is supported through parallel halfword shift-and-add instructions. The HALFWORD SHIFT LEFT AND ADD and HALFWORD SHIFT RIGHT AND ADD instructions perform four parallel halfword shift and add operations. These instructions are used as primitive operations in performing halfword integer multiplication and division by a constant.

Saturation

The halfword addition, subtraction, and shift-and-add instructions can be performed with normal modular arithmetic or with signed saturation or unsigned saturation. Saturation arithmetic occurs frequently in multimedia algorithms. When an intermediate result of an operation cannot be represented in the target register, saturation is said to occur and the result is forced to a maximum or minimum value. Thus, when a result is out of range (too large or too small to be represented in the target register) the saturation result is automatically delivered and no extra instructions are required to test for these boundary conditions. Saturation is performed independently on each of the 16-bit results.

Optional saturation is specified via instruction completers. In the instruction descriptions, the term cmplt is used to denote the completer field which encodes the sat field. If no completer is specified, the operands are added or subtracted with modular arithmetic. If signed saturation is specified, both operands are treated as signed numbers and are added or subtracted producing a signed result with signed saturation. If unsigned saturation is specified, the first operand is treated as an unsigned number.
and the second as a signed number. These are added or subtracted producing an unsigned result with unsigned saturation.

The results for maximum saturation and minimum saturation are defined for halfword arithmetic instructions in the following tables. Signed saturation results are defined in Table 6-3 and unsigned saturation results are defined in Table 6-4.

Table 6-3. Signed Saturation Results

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Maximum Saturation</th>
<th>Minimum Saturation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HADD, HSUB, HSHRADD</td>
<td>Intermediate result of an operation is greater than $2^{15} - 1$. The result is forced to $2^{15} - 1$.</td>
<td>Intermediate result of an operation is less than $-2^{15}$. The result is forced to $-2^{15}$.</td>
</tr>
<tr>
<td>HSHLADD</td>
<td>Intermediate result of an operation is greater than $2^{15} - 1$. The result is forced to $2^{15} - 1$. In addition, the result is also forced to $2^{15} - 1$ if the leftmost bit of the first operand is 0, and one or more of the bits shifted out differs from the leftmost bit following the shift.</td>
<td>Intermediate result of an operation is less than $-2^{15}$. The result is forced to $-2^{15}$. In addition, the result is also forced to $-2^{15}$ if the leftmost bit of the first operand is 1, and one or more of the bits shifted out differs from the leftmost bit following the shift.</td>
</tr>
</tbody>
</table>

Table 6-4. Unsigned Saturation Results

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Maximum Saturation</th>
<th>Minimum Saturation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HADD, HSUB</td>
<td>Intermediate result of an operation is greater than $2^{16} - 1$. The result is forced to $2^{16} - 1$.</td>
<td>Intermediate result of an operation is less than 0. The result is forced to 0.</td>
</tr>
</tbody>
</table>

Parallel Halfword Shift Instructions

The halfword shift instructions allow multiple parallel halfword shifts. The shift amount is encoded in the instruction and shifting can be done by any amount, from 0 to 15 bits. These instructions generally use the main shifter, except that they block any bits from being shifted across halfword boundaries. The HALFWORD SHIFT LEFT instruction allows multiplication by $2^n$ in a single instruction. The HALFWORD SHIFT RIGHT instruction allows division by $2^n$ and the shift can be either signed or unsigned. The completer, cmplt, determines which type of shift to perform. The completer is encoded in the se field of the instruction.

Rearrangement Instructions

The PERMUTE HALFWORDS, MIX HALFWORDS, and MIX WORDS instructions allow full utilization of the halfword parallel arithmetic instructions by supporting the rearrangement of words and halfwords in registers without incurring the overhead of memory load and store instructions. These instructions allow arbitrary permutations and combinations of words and halfwords in a register.

The PERMUTE HALFWORDS instruction can generate any arbitrary combination or permutation of the
four halfwords from its source operand.

In the MIX HALFWORDS instruction, two halfwords from the first operand are merged with two halfwords from the second operand to produce the result. The completer, *cmplt*, determines which halfwords are selected. The completer is encoded in the *ea* field of the instruction.

In the MIX WORDS instruction, a word from the first operand is merged with a word from the second operand to produce the result. The completer, *cmplt*, determines which words are selected. The completer is encoded in the *ea* field of the instruction.

**Memory Reference Instructions**

Memory reference instructions load values into and store values from the general registers. Table 6-5 summarizes the memory reference instructions.

<table>
<thead>
<tr>
<th>Memory Reference Instruction Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load/Store</strong></td>
</tr>
<tr>
<td>LDB/STB, LDH/STH, LDW/STW, LDD/STD</td>
</tr>
<tr>
<td><strong>Load/Store Absolute</strong></td>
</tr>
<tr>
<td>LDWA/STWA, LDDA/STDA</td>
</tr>
<tr>
<td><strong>Load and Clear</strong></td>
</tr>
<tr>
<td>LDCW, LDCD</td>
</tr>
<tr>
<td><strong>Store Bytes/DoubleWord Bytes</strong></td>
</tr>
<tr>
<td>STBY, STDBY</td>
</tr>
</tbody>
</table>

Memory reference instructions work directly between the registers and main memory. They also can operate between the registers and the data cache on implementations so equipped. A load instruction loads a general register with data from the data cache. A store instruction stores a data value from a general register into the data cache. Normally this distinction is transparent to the programmer, but provisions are made for cache and TLB operations requiring cognizance of the data cache (see “System Control Instructions” on page 6-17).

The address formation mechanisms supported include: short displacement, long displacement, and indexed. It is possible to modify the base value in a general register by the displacement or index. The rightmost bits of computed addresses are not ignored. Unaligned load and store instructions with data address translation enabled to halfwords, words, or doublewords cause an unaligned data reference trap. Semaphore operations and absolute accesses to unaligned data are undefined operations.

Program synchronization can be done using the LOAD AND CLEAR instructions, which perform indivisible semaphore operations. These instructions are required to use 16-byte aligned addresses.
When using semaphores to synchronize with I/O, care must be taken in placing other information in the same cache line as the semaphore. Data which is writable, can only be placed in the same cache line as a semaphore if access to write the data is controlled by the semaphore.

Depending on the state of the PSW D-bit (data address translation bit), most load and store instructions perform virtual accesses (when the PSW D-bit is 1) or physical accesses (when the PSW D-bit is 0, or when executing LOAD ABSOLUTE, STORE ABSOLUTE instructions).

The state of the PSW E-bit determines whether the data which is loaded or stored is big endian (when the PSW E-bit is 0) or little endian (when the PSW E-bit is 1).

Memory is accessed using the following procedures:

```c
mem_load(space,offset,low,high,hint)
{
    if (PSW[D] == 0)
        return(phys_mem_load(offset,low,high,hint));
    else
        return(virt_mem_load(space,offset,low,high,hint));
}
```

```c
mem_store(space,offset,low,high,hint,data)
{
    if (PSW[D] == 0)
        phys_mem_store(offset,low,high,hint,data);
    else
        virt_mem_store(space,offset,low,high,hint,data);
}
```

There are some restrictions on which instructions can be used for referencing the I/O address space and uncacheable memory. See “Operations Defined for I/O Address Space” on page F-12. For a description of memory reference atomicity, see “Atomicity of Storage Accesses” on page G-1.

LOAD OFFSET, LOAD IMMEDIATE LEFT, LOAD PHYSICAL ADDRESS, LOAD COHERENCE INDEX, and LOAD SPACE IDENTIFIER are not memory reference instructions.

---

**Programming Note**

Execution may be faster if software avoids dependence on register interlocks. Instruction scheduling to avoid the need for interlocking is recommended. A register interlock will occur if an instruction attempts to use a register which is the target of a previous load instruction that has not yet completed. This does not restrict the length of the delay a load instruction may incur in a particular system to a single execution cycle; in fact, the delay may be much longer for a cache miss, a TLB miss, or a page fault.

Debugging is facilitated by the data memory break trap. This trap occurs whenever a store, a semaphore, or a purge data cache operation is performed to a page with the B-bit 1 in its TLB entry and the PSW X-bit is 0.
Address Formation

Addresses are formed by the combination of a Space ID and an address Offset. Address Offsets may be formed as the sum of a base register and any one of the following: a long displacement, a short displacement (which leaves more instruction bits for other functions), or an index register. Figure 6-1 on page 6-9 illustrates typical examples of the various methods of forming addresses for the memory reference instructions. For detailed illustrations of address calculations for each of the available addressing methods refer to “Memory Reference Instruction Address Formation” on page H-1.

Not all address formation methods are available with every memory reference instruction. Table 6-6 summarizes the address formation methods and the instructions where each is available.

Table 6-6. Address Formation Options for Memory Reference Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Indexed</th>
<th>Short Displacement</th>
<th>Long Displacement</th>
<th>Base Register Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDD, LDW, LDH, LDB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>STD, STW, STH, STB</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LDDA, LDWA, LDCD, LDCW</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>STDA, STWA, STBY, STDBY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Base Register Modification

All of the address formation methods provide the option of modifying the contents of the Base Register either before or after the address calculation is performed. The address can be formed by using the contents of Base Register before it is modified (sometimes known as post-increment) or by using the contents of the Base Register after it has been modified by the displacement or index (sometimes known as pre-increment.)
Cache Control

Some memory reference instruction formats contain a 2-bit cache control field, \( cc \), which provides a hint to the processor on how to resolve cache coherence. The processor may disregard the hint without compromising system integrity, but performance may be enhanced by following the hint.

There are three different categories of cache control hints: load instruction cache control hints, store instruction cache control hints, and semaphore instruction cache control hints. The cache control hints are specified by the \( cc \) completer to the instruction and encoded in the \( cc \) field of the instruction.
The cache control hints for load instructions are shown in Table 6-7. Implementation of the hints by a processor is optional, but the processor must treat unimplemented and Reserved hints as if no hint had been specified.

The Spatial Locality cache control hint is a recommendation to the processor to fetch the addressed cache line from memory but not displace any existing cache data because there is good spatial locality but poor temporal locality.

Table 6-7. Load Instruction Cache Control Hints

<table>
<thead>
<tr>
<th>Completer</th>
<th>Description</th>
<th>cc</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>No hint</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>01</td>
</tr>
<tr>
<td>SL</td>
<td>Spatial Locality</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>11</td>
</tr>
</tbody>
</table>

The cache control hints for store instructions are shown in Table 6-8. Implementation of the hints by a processor is optional, but the processor must treat unimplemented and Reserved hints as if no hint had been specified.

Table 6-8. Store Instruction Cache Control Hints

<table>
<thead>
<tr>
<th>Completer</th>
<th>Description</th>
<th>cc</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>No hint</td>
<td>00</td>
</tr>
<tr>
<td>BC</td>
<td>Block Copy</td>
<td>01</td>
</tr>
<tr>
<td>SL</td>
<td>Spatial Locality</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>11</td>
</tr>
</tbody>
</table>

The Block Copy cache control hint is a recommendation to the processor not to fetch the addressed cache line if it is not found in the cache. Instead, the processor may create a cache line for the specified address and perform the store instruction on the created line. If the cache line is not fetched then the processor must zero the rest of the created cache line if the privilege level is 1, 2, or 3. The processor may optionally zero the rest of created the cache line if the privilege level is 0. If the store instruction with the Block Copy hint does not store into at least the first byte of the cache line, the processor must perform the store as if the cache control hint had not been specified.

The Block Copy cache control hint is a way for software to indicate that it intends to store a full cache line worth of data. Note that this hint should only be used if the rest of the memory in the addressed cache line is no longer needed.

The cache control hints for the LOAD AND CLEAR semaphore instructions are shown in Table 6-9. The implementation of the hints by the processor is optional. If no hints are implemented, the processor must treat all hints as if no hint had been specified. If the Coherent Operation hint is implemented, the
processor must treat Reserved hints as if the Coherent Operation hint had been specified.

The Coherent Operation cache control hint is a recommendation to the processor that, if the addressed data is already in the cache, it can operate on the addressed data in the cache rather than having to update memory.

All software users of a semaphore must access the semaphore using the same cache control hint. Sharing a semaphore using different cache control hints is undefined.

### Data Prefetch Instructions

Data prefetch instructions are used to initiate a prefetch of the addressed data into the data cache before it is required by later memory reference instructions, thus hiding some or all of the cache-miss latency.

Data prefetch instructions are encoded as normal load instructions with a target register of GR0. All of the normal load addressing modes (long displacement, short displacement, and indexed), base register modification, and cache hints are available. The prefetch address is never unaligned — the low-order address bits are ignored and the cache line containing the address is fetched.

All interruptions normally associated with memory reference instructions (Data TLB miss fault/data page fault, Data memory access rights trap, Data memory protection ID trap, Page reference trap) are suppressed for data prefetch instructions. If one of these exceptions would occur, the prefetch is simply ignored, but any base register modification specified by the instruction still occurs.

There are four data prefetch instructions, corresponding to targeting GR0 for each of the four load instruction data sizes, as shown in Table 6-10. The two reserved encodings do not perform any prefetch, but otherwise operate as described in this section (e.g., interruptions are suppressed, base modification still occurs).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDD</td>
<td>Prefetch cache line for write</td>
</tr>
<tr>
<td>LDW</td>
<td>Prefetch cache line for read</td>
</tr>
<tr>
<td>LDH</td>
<td>Reserved</td>
</tr>
<tr>
<td>LDB</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Prefetch for read indicates that the cache line is likely to be used in a subsequent load operation while prefetch for write indicates that a subsequent store will use the cache line. These are distinguished in
order to allow the line to be brought into the cache in the correct state.

A prefetch for write is allowed to bring the cache line into the cache in the modified state, in which case the instruction must check access rights as if it was a store and check for the Data memory break and TLB dirty bit traps and suppress the prefetch if any of these checks fails.

A LOAD AND CLEAR instruction with a target register of GR0 may be implemented as a normal LOAD AND CLEAR, which clears the data in memory and discards the original contents, or may be aliased to the equivalent-size load instruction (LDCD to LDD, LDCW to LDW), in which case it behaves exactly like that prefetch instruction and does not clear the data in memory.

Store Bytes Instructions

STORE BYTES and STORE DOUBLEWORD BYTES provide the means for doing unaligned byte moves efficiently. These instructions use a short 5-bit displacement to store bytes to unaligned destinations. The short displacement field is in two’s complement notation with the sign bit as its rightmost bit.

The space identifier is computed like any other data memory reference (see Figure H-1 on page H-2). The calculation of the offset portion of the effective address for different completers is shown in Figure H-5. Space and offset are combined like any other data memory reference (see Figure H-3 on page H-3).

The actual offset and modified address involves some alignment and other considerations. Refer to the instruction description pages for an exact definition.

Long Immediate Instructions

The long immediate instructions do not reference memory. They compute values either from a shifted long immediate (21 bits long), from a shifted long immediate and a source register, or from a base register plus a 16-bit displacement. This computed value is then stored in another general register. These instructions are typically used to compute the addresses of data items. The LOAD OFFSET instruction can also be used to simply load a 16-bit immediate into a register.

Table 6-11. Immediate Instruction Summary

<table>
<thead>
<tr>
<th>Immediate Instructions</th>
<th>The three immediate instructions load a computed value into a register or add an immediate value into a register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDO, LDIL, ADDIL</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-2 on page 6-13 illustrates the operation of the immediate instructions.
Branch Instructions

Branch instructions are classified into three major categories: unconditional local branches, unconditional external branches, and conditional local branches. Within these categories there is sub-classification based on how the target address is computed, whether or not a return address is saved, and whether or not privilege changes can occur. Not all of the options are available for each category. The following sections describe the types of branches. The operation of each branch instruction is detailed in the instruction description in Chapter 7, “Instruction Descriptions”. Table 6-12 summarizes the categories of branch instructions.
Table 6-12. Branch Instruction Summary

<table>
<thead>
<tr>
<th>Unconditional Local Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>B, BLR, BV</td>
</tr>
<tr>
<td>Branch, branch and link, or branch vectored unconditionally within the current space using IA- or base-relative addressing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unconditional External Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE, BVE</td>
</tr>
<tr>
<td>Branch or branch vectored unconditionally to another space using base-relative addressing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditional Local Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDB, ADDIB, BB, CMPB, CMPIB, MOVB, MOVIB</td>
</tr>
<tr>
<td>Branch within the current space if the specified condition is satisfied using IA-relative addressing. Categories include: move and branch, compare and branch, add and branch, and branch on bit.</td>
</tr>
</tbody>
</table>

Unconditional Local Branches

The unconditional local branch instructions are used for intraspace control transfers, procedure calls, and procedure returns. Three types of relative addressing are provided:

1. IA-relative branches with static displacement use the IAOQ_Front plus either a 17-bit or 22-bit signed word displacement. This allows a branch target range of up to plus or minus 8 Mbytes within a space.

2. IA-relative branches with dynamic displacement use the IAOQ_Front plus a shifted index register.

3. Base-relative branches with dynamic displacement use the value in a base register plus a shifted index register.

The BRANCH instruction satisfies most of the requirements for unconditional branching. The branch target is IA relative with a 17-bit static displacement.

A BRANCH instruction with the optional .L (for link) completer is used for procedure calls. The branch target is IA relative with a 22-bit displacement if GR 2 is specified as the link register, and with a 17-bit displacement if any other general register is specified. In addition, this variant of the instruction places the offset of the return point (or link) in the specified GR. The return point is the location four bytes beyond the address of the instruction which executes after the BRANCH.

A BRANCH instruction with the optional .GATE (for gateway) completer is used for intraspace branching with a process privilege level promotion. The branch target is IA relative with a 17-bit static displacement.

The BRANCH AND LINK REGISTER instruction is used for intraspace procedure calls in which the branch target is outside the range for a BRANCH instruction with the .L completer, or when a dynamic target displacement is needed. The branch target address is base relative with a dynamic displacement. Link handling is performed the same way as for a BRANCH instruction with the .L completer.

The BRANCH VECTORED instruction is used for intraspace branching through a table and for procedure returns. The branch target address is base relative with a dynamic displacement. The process privilege level may be demoted.
Unconditional External Branches

The unconditional external branch instructions are used for interspace control transfers, procedure calls, and procedure returns. All unconditional external branch instructions use base-relative addressing and may demote the process privilege level based on the rightmost bits of the base register.

Two types of base-relative addressing are provided:

- Base-relative branches with static displacements use a base register plus a 17-bit signed word displacement. This allows a branch target range of up to plus or minus 256 Kbytes across space boundaries. The target space comes from a Space Register which is specified explicitly.

- Base-relative branches with no displacement or index value. The target space comes from an SR which is specified implicitly by the base register.

The BRANCH EXTERNAL instruction is used for interspace branching and procedure returns.

A BRANCH EXTERNAL instruction with the optional ,L completer is used for interspace procedure calls. It places the offset of the return point in GR 31 and copies the space ID into SR 0. The return point is the location four bytes beyond the address of the instruction which executes after the branch.

The BRANCH VECTORED EXTERNAL instruction is used for interspace branching through a table and for procedure returns. The target space is specified implicitly by the base register.

A BRANCH VECTORED EXTERNAL instruction with the optional ,L completer is used for interspace procedure calls. It places the offset of the return point in GR 2. The return point is the location four bytes beyond the address of the instruction which executes after the branch.

Conditional Local Branches

The conditional local branch instructions are used to perform an operation and then branch if the condition specified is satisfied. All conditional local branch instructions use IA-relative addressing with static displacements. The target address is the current IAOQ_Front plus a 12-bit signed word displacement. This allows a branch target range of up to plus or minus 8 Kbytes within a space.

There are four categories of conditional local branch instructions: move and branch, compare and branch, add and branch, and branch on bit. The branch may be taken if the condition specified is true or false. There are two forms of each instruction, the two-register form and the register plus 5-bit immediate form. The 5-bit immediate operand provides data values in the range -16 to +15.

Branch Target Stack

The Branch Target Stack (or BTS) is an optional processing resource which is used to accelerate indirect branches, such as subroutine returns. The BTS is managed by software, and in processors which implement it, can provide the branch target address in place of the general register specified in the branch instruction.

Operations which push an address onto the stack:

- B,L,PUSH - Used for normal function calls.

- BVE,L,PUSH - Used for intra-space calls, such as calls to library functions.
• **PUSHBTS** - Used to push a value from a GR onto the stack, in preparation for a dynamic branch.

• **PUSHNOM** - Pushes the value in BNR onto the stack; used in the called function if the caller did not push the return address on the stack.

Operations which pop an address from the stack:

• **BVE,POP** - An address is popped from the stack, and if valid, it is used as the target address. Otherwise, the BVE branches to an address given by a GR.

• **POPBTS** - Pops a specified number of entries from the stack and discards them; used for stack unwinding.

• **CLEARBTS** - Pops all entries from the stack, discarding them all and leaving the stack invalid; this is used in situations where the sequence of calls and returns is reset, such as with LONGJMP in Unix systems.

All branch-and-link instructions nominate their link value. That is, the link value which is written to a GR is also copied into BNR.

So, for call/return acceleration, one of these two scenarios is used:

• The caller uses a B,L,PUSH or BVE,L,PUSH to call. The callee uses a BVE,POP to return. (This is the normal scenario.)

• The caller does not explicitly push a value onto the stack (it does not specify a ,PUSH completer on the branch used to call). The callee does a PUSHNOM to push the link onto the stack. The callee uses a BVE,POP to return.

For dynamic branches (such as may be generated by C-language switch statements), this scenario is used:

• A **PUSHBTS** is done as soon as the target address has been calculated. Then, at the point the branch is done, a BVE,POP is done.

**Branch Characteristics**

Figure 6-3 categorizes the characteristics of the branch instructions.
System Control Instructions

The system control instructions provide special register moves, system mask control, return from interruption, probe access rights, memory management operations, and implementation-dependent functions. Table 6-13 summarizes the System Control instructions that are provided.

Figure 6-3. Classification of Branch Instructions
Table 6-13. System Control Instruction Summary

<table>
<thead>
<tr>
<th>Classification</th>
<th>Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special Register Move Instructions</td>
<td>LDSID, MTSP, MFSP, MTCTL, MFCTL, MTSARCM, MFIA,</td>
<td>These instructions move values to and from the space registers, control registers, the shift amount, and instruction address register.</td>
</tr>
<tr>
<td>System Mask Control Instructions</td>
<td>SSM, RSM, MTSM</td>
<td>These instructions set, reset, and move values to the system mask portion of the PSW.</td>
</tr>
<tr>
<td>Return From Interrupt &amp; Break</td>
<td>RFI, BREAK</td>
<td>Restore state and restart interrupted instruction stream or cause a break for debugging purposes.</td>
</tr>
<tr>
<td>Memory Management Instructions</td>
<td>SYNC, SYNCDMA, PROBE, PROBEI, LPA, LCI, PDTLB, PITLB, PDTLBE, PITLBE, IDTLBT,</td>
<td>These instructions synchronize memory operations, probe addresses to determine access rights, load a physical address or a coherence index, purge or insert TLB entries or translations, and purge or flush data or instruction caches or cache entries.</td>
</tr>
<tr>
<td></td>
<td>IITLBT, PDC, FDC, FIC, FDCE, FICE</td>
<td></td>
</tr>
<tr>
<td>Implementation-Dependent Instruction</td>
<td>DIAG</td>
<td>Provide implementation-dependent operations for diagnostic purposes.</td>
</tr>
</tbody>
</table>

The memory management instructions generate instruction and data addresses. Address formation is similar to that of the indexed load instructions. The only difference is that the index register is never shifted before adding to the base register.

Memory management instructions select a space identifier either implicitly or explicitly as shown in Figure 3-8 on page 3-8. The calculation of the offset portion of the address is shown in Figure 6-4.
Assist Instructions

The PA-RISC design generally conforms to the concept of a simple instruction set implemented in cost-effective hardware. Certain algorithms can benefit from substantial performance gains by dedicating specialized hardware to execute specialized instructions. Since few algorithms rely solely upon the specialized hardware alone, it is usually advantageous to combine the central processor with additional assist processors closely coupled to it.

In addition to the instructions executed by a central processor, the instruction set contains instructions to invoke the special, optional, hardware functions provided by the two types of assist processors: Special Function Units (SFUs) and coprocessors. Table 6-14 summarizes the assist instructions that are provided for SFUs and coprocessors.

Table 6-14. Assist Instruction Summary

<table>
<thead>
<tr>
<th>Special Function Instructions</th>
<th>Coprocessor Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPOP0, SPOP1, SPOP2, SPOP3</td>
<td>These instructions invoke SFU operations, copy SFU register or result to a general register, and perform a parameterized SFU operation.</td>
</tr>
<tr>
<td>COPR, CLDD, CLDW, CSTD, CSTW</td>
<td>These instructions invoke a coprocessor operation and load or store words or doublewords to or from a coprocessor register.</td>
</tr>
</tbody>
</table>

Figure 6-4. System Operations
Special function units are closely coupled to the central processor and provide extensions to the instruction set. They use the general registers as operands and targets of operations.

Coprocessors provide functions that use either memory locations or coprocessor registers as operands and targets of operations. Coprocessors are less closely coupled to the central processor, and so are more easily provided as configuration options for an implementation than special function units. Coprocessors may also directly pass doubleword quantities to and from the coprocessor and memory. This is suited to the manipulation of quantities that are too large to be directly handled in general registers.

The special function unit and coprocessor instructions are intended to encapsulate all of the optional hardware features used for non-system-level code. An emulation facility is provided that permits PA-RISC family members to execute code using the standard instruction set when optional hardware is not present. The emulation facility is provided by the assist emulation trap, which passes information in control registers, substantially reducing the instruction path length for emulation.

The assist exception trap permits partial implementations of standard “hardware” functions in a combination of hardware and software. This handles functions that are difficult or not cost-effective to implement fully in hardware.

Compatibility Among Implementations

The standard PA-RISC instruction set contains all defined instructions, including those for all defined assist processors. Particular implementations may choose to implement these instructions in hardware, software, or some combination of the two, using assist emulation traps and/or assist exception traps to complete the implementation. Thus, these instructions can be used by compilers and assemblers without sacrificing object-code portability. Software emulation of the extended functions is also used to permit execution of the object code in a degraded mode for high-availability systems.

Special Function Unit (SFU) Instructions

The SFU mechanism is intended for certain architecturally defined instruction extensions, such as hardware fixed-point binary multiply/divide or encryption hardware, as well as for implementation-specific extensions, such as emulation assist processors or direct I/O controller connections.

SFUs are connected to the general register interface and are invoked by special operation instructions. These instructions cause the execution unit to perform any of several operations (determined by the opcode extension), which may use the contents of registers, or may write back a result. Some instructions conditionally nullify the following instruction.

Some special function operations overlap their execution with succeeding instructions. These operations require that the special function unit’s state be saved and restored when a context switch is made. An interlock occurs if a special function result is requested before the operation has completed, or the special function unit is busy.

An SFU is not required to hold its state in addressable registers. Instead, SFU operations are used to save and restore the state, as well as to pass it operands and receive results from it.

Defined special function units will conform to the requirements of the defined SFU instructions, so that they may be implemented either as built-in or interfaced special function units. The assist emulation
trap permits software implementation of any defined special operation instruction.

The processor must also provide the current privilege level to special function units. Privilege levels could be broadcast each time they change or could be transmitted with each SFU operation. Use of the privilege level by the SFU is specific to each of the units. The operation paragraph of each SFU instruction description specifies the necessary information that must be available to the SFU in the sfu_operation function.

There is one SFU instruction, the IDENTIFY SFU (SPOP1) instruction, that is defined for all SFUs. It must be implemented.

SFU Configuration Register

The SCR (SFU Configuration Register) is an 8-bit control register (within CR 10 bits 16..23) that is used to indicate the presence and usability of a hardware implementation of an SFU. For all bits in the SCR, SCR[i] corresponds to an undefined SFU with a unit identifier that is the same as the bit position, that is the SFU with uid i.

When SCR[i] is 1, the SFU with uid i is implied to be present and usable. SFU instructions are passed to the SFU and the defined operation occurs. Exceptions resulting from the operation cause the instruction to be terminated with an assist exception trap. Assist emulation traps are not allowed to occur for the SFU with uid i when SCR[i] is 1. It is an undefined operation to set to 1 the SCR bit corresponding to a nonexistent SFU.

When SCR[i] is 0, it is not implied that the SFU with uid i is absent from the system, but rather that the SFU, if present, is not currently being used. When the SCR bit is 0, the SFU instruction is terminated with an assist emulation trap. Assist exception traps are not allowed to occur for the SFU with uid i when SCR[i] is 0.

Setting the SCR[i] bit to 0 must logically decouple the SFU with uid i. This must ensure that the state of the SFU with uid i is frozen just prior to the transition of SCR[i] from 1 to 0 and that the state does not change as long as SCR[i] is 0. When SCR[i] is 0, the SFU with uid i must not respond to any SFU operations for the SFU with uid i. The frozen state of an SFU, for example, could also be a state in which the SFU is left “armed” to trap any subsequent operations. For example, if the SFU with uid i is in an “armed-to-trap” state and SCR[i] is 0, any operation involving that SFU must not cause an assist exception trap.

The precedence of the interruptions that are applicable to operations for the SFU with uid i depends on the state of SCR[i]. The assist exception trap and assist emulation trap are always taken in the priority order as described in “Interruption Priorities” on page 5-4.

NOTE

Logical decoupling may be accomplished in a variety of ways. Processors may use abort signals or other schemes to notify SFUs that the current instruction is to be ignored.

When the SCR bit is 0, logical decoupling suppresses any exception traps from an SFU and causes the emulation trap to occur (if it is the highest priority).
Coprocessor Instructions

The coprocessor mechanism is intended for special-purpose data manipulations, for example to handle data larger than will fit in a general register. The interconnection method allows for instruction set extensions with minimal effect on the instruction execution rate, while maintaining short data communication paths between the coprocessors and the rest of the system. Coprocessor instructions can be executed by the coprocessor hardware or emulated by software. Combinations of instructions implemented in hardware and emulated by software are possible even when the coprocessor hardware is present in a system.

When caches are implemented, coprocessors are connected to the CPU-cache interface. For systems that do not have a cache, coprocessors are connected to the CPU-memory bus interface. Coprocessors manipulate data in their own register sets, but use the data cache or memory bus and central processor’s address generation logic. Under control of the CPU, coprocessor load instructions pass data from the data cache or memory bus to a coprocessor, and coprocessor store instructions pass data from a coprocessor to the data cache or memory bus. Coprocessor operations use only the coprocessor’s registers. Some coprocessor operations may nullify the following instruction.

Coprocessor operation, load, and store instructions may overlap their execution with succeeding instructions. An interlock occurs if a coprocessor operation is requested before the coprocessor is able to perform it, and for loads and stores involving busy coprocessor registers.

The coprocessor load and store instructions contain a 5-bit field which normally specifies a coprocessor register, but may also be interpreted by coprocessors as a sub-operation field. Coprocessors keep their state in their registers, so that storing the coprocessor registers and reloading them is sufficient to save and restore the state of a coprocessor.

Some coprocessors are capable of supporting doubleword load and store operations. These operations are implemented on all systems that support such coprocessors, even though they may require additional cycles for some machines. Coprocessor load and store operations must be atomic.

The operation section of each coprocessor instruction description specifies the necessary information that must be available to the coprocessor in the coprocessor_op and send_to_copr functions. There is one coprocessor instruction, the IDENTIFY COPROCESSOR (COPR,uid,0) instruction, that is defined for coprocessors with unit identifiers 4 through 7. Coprocessors with unit identifiers 0 and 3 have a mechanism to identify themselves that is individually defined.

NOTE

An unaligned data reference trap is taken if the appropriate number of rightmost bits of the effective virtual address are not zeros for the COPROCESSOR LOAD WORD, COPROCESSOR LOAD DOUBLEWORD, COPROCESSOR STORE WORD, and COPROCESSOR STORE DOUBLEWORD instructions. Absolute accesses to unaligned data are undefined operations.

Coprocessor Configuration Register

The CCR (Coprocessor Configuration Register) is an 8-bit control register (within CR 10 bits 24..31) that is used to indicate the presence and usability of a hardware implementation of a coprocessor. Bits 0 and 1 in the CCR correspond to the floating-point coprocessor and bit 2 in the CCR corresponds to the performance monitor coprocessor. For all other bits in the CCR, CCR{I} corresponds to an undefined
A coprocessor with a unit identifier that is the same as the bit position, that is the coprocessor with uid i.

Execution of any floating-point instruction with CCR{0} and CCR{1} not set to the same value is an undefined operation. Execution of a coprocessor operation instruction (major opcode 0x0C) with CCR{0}, CCR{1}, and the uid field in the instruction all set to 1 is an undefined operation.

When CCR{i} is 1, the coprocessor with uid i is implied to be present and usable. Coprocessor instructions are passed to the coprocessor and the defined operation occurs. Exceptions resulting from the operation cause the instruction to be terminated with an assist exception trap. Assist emulation traps are not allowed to occur for the coprocessor with uid i when CCR{i} is 1. It is an undefined operation to set to 1 the CCR bit corresponding to a nonexistent coprocessor.

When CCR{i} is 0, it is not implied that the coprocessor with uid i is absent from the system, but rather that the coprocessor, if present, is not currently being used. When the CCR bit is 0, the coprocessor instruction is terminated with an assist emulation trap. Assist exception traps are not allowed to occur for the coprocessor with uid i when CCR{i} is 0.

Setting the CCR{i} bit to 0 must logically decouple the coprocessor with uid i. This must ensure that the state of the coprocessor with uid i is frozen just prior to the transition of CCR{i} from 1 to 0 and that the state does not change as long as CCR{i} is 0. When CCR{i} is 0, the coprocessor with uid i must not respond to any coprocessor operations for the coprocessor with uid i. The frozen state of a coprocessor, for example, could also be a state in which the coprocessor is left “armed” to trap any subsequent operations. For example, if the coprocessor with uid i is in an “armed-to-trap” state and CCR{i} is 0, any operation involving that coprocessor must not cause an assist exception trap.

The precedence of the interruptions that are applicable to operations for the coprocessor with uid i depends on the state of CCR{i}. The assist exception trap and assist emulation trap are always taken in the priority order as described in “Interruption Priorities” on page 5-4.

 logical decoupling may be accomplished in a variety of ways. Processors may use abort signals or other schemes to notify coprocessors that the current instruction is to be ignored.

When the CCR bit is 0, logical decoupling suppresses any exception traps from a coprocessor and causes the emulation trap to occur (if it is the highest priority).

Conditions and Control Flow

Many instructions utilize conditions derived from the values of the operators and the operation performed. The architecture defines several sets of conditions that affect control flow:

- Arithmetic/Logical Conditions.
- Unit Conditions.
- Shift/Extract/Deposit Conditions.
- Branch On Bit Conditions.

Every instruction that tests conditions uses one of these sets. Each set contains a maximum of sixteen
separate conditions and their negations. Most instructions that use conditions may also select the negation of a condition.

The condition completer field, cond, in the assembly language form of the instructions specifies a condition or the negation of a condition. This field expands in the machine language form to fill the condition field, c, (normally 3 bits wide), the 1-bit negation field, f, and the 1-bit doubleword field, d, as required. For some instructions, the negation or doubleword attributes of the condition are controlled by the opcode.

The result of an operation and the specified condition can affect control flow in the following ways:

- **Branching** – the result determines whether or not the branch is taken.
- **Nullifying** – the result determines whether or not the next instruction is nullified.
- **Trapping** – the result determines whether a conditional trap is taken or execution proceeds normally.

### Additional Notes on the Instruction Set

This section defines how the architecture and instruction notation handles such details as undefined, illegal, null, and unimplemented instructions.

#### Undefined and Illegal Instructions

Not all of the 64 possible major opcodes of the instruction set are defined as valid instructions. (See Appendix C, “Operation Codes”, for a list of the valid instruction opcodes.) An undefined major opcode is considered an illegal instruction. Execution of an illegal instruction causes an illegal instruction trap.

Within each major opcode, there may be undefined opcode extensions and modifiers (these are undefined instructions). Interpretation of these opcodes is left to the implementor, but system integrity is not compromised. An undefined instruction, or sequence of undefined instructions, executed at a given privilege level has no effect on system state other than what would have been produced by a sequence of defined instructions running at the same privilege level. This limits the possible side-effects that could result from undefined instructions.

Undefined operations are equivalently specified. These result from normally defined instructions but with operands or specifiers that are explicitly disallowed.

Executing an optional special operation or coprocessor instruction may cause an assist exception trap or an action that depends on the definition of the specific special function unit or coprocessor.

#### Reserved Instruction Fields

In the Format section of the instruction description pages in Chapter 7, instruction fields marked rv are Reserved instruction fields. These fields are reserved for future architectural definition. To avoid incompatibility with future revisions of the architecture, software must provide zeros in all Reserved fields. When decoding instructions, processors must ignore Reserved instruction fields.
Reserved Values of an Instruction Field

Certain values of some instruction fields are Reserved for future architectural definition. To avoid incompatibility with future revisions of the architecture, software must not use the Reserved values. When decoding instructions, processors must treat the Reserved values as described for the specific field.

Null Instructions

Null instructions occur when unimplemented features of the architecture are accessed. The effect of a null instruction is identical to a nullified instruction except that the Recovery Counter is decremented. There is no effect on the machine state except that the IA queues are advanced and the PSW B-bit, N-bit, X-bit, Y-bit, and Z-bit are set to 0.