# Processor-Specific ELF Supplement for PA-RISC

## Including HP and HP-UX Extensions

*Version 1.5*
*August 20, 1998*

This document is a supplement to the processor-independent definitions of ELF-32 and ELF-64. It presents the machine-specific definitions for the PA-RISC architecture. It also contains the HP vendor-specific and HP-UX environment-specific extensions to ELF that are specific to the PA-RISC architecture.

*Note* *Although this document applies to ELF-32 and ELF-64, the HP-UX operating system does not currently support ELF object files except for wide-mode programs running on PA-RISC 2.0 machines. All programs written for PA-RISC 1.x machines, or for narrow mode on PA-RISC 2.0 must use the SOM format.*

### 1. Overview of an ELF file

The standard ELF file organization is used for the PA-RISC architecture.

It is important to note that the choice of ELF-32 vs. ELF-64 may be made independently of the programming model used by the program being compiled or linked. A 64-bit application may be represented in an ELF-32 file as long as its sections and program segments are small enough to be representable in the smaller format. For 64-bit applications in an ELF-32 file, 64-bit virtual addresses are represented as an `Elf32_Addr` object by concatenating bits 63–62 with bits 29–0 of the 64-bit address, omitting bits 61–30, which must be all zeroes. If any of the omitted bits are ones, the program is too large, and it must be placed in an Elf-64 object file.

### 2. Data representation

The standard ELF-32 and ELF-64 data representations are used for the PA-RISC architecture.

# 3. File Header

The fields of the file header that have interpretations specific to PA-RISC are explained below.

- e_ident identifies the object file as an ELF file, and provides machine-dependent data. Table 1 lists the values of the EI_CLASS field, and Table 2 lists the values of the EI_DATA field that are used for PA-RISC.
- e_type identifies the object file type. There are no processor-specific file types currently defined for PA-RISC.
- e_machine identifies the target machine on which the object file is designed to run. The value 15 has been assigned by SCO for the PA-RISC architecture, as shown in Table 3.

  New definitions will also be added for this field whenever an incompatible change is made to the architecture or software conventions that would prevent successful interoperability between old and new object files. For backward-compatible extensions to the architecture or conventions, see the EF_PARISC_ARCH bits in the e_flags field, defined below.
- e_flags contains flag bits describing the object file. The valid flags for PA-RISC object files are listed in Table 4.

  If the EF_PARISC_EXT bit is set, a .PARISC.archext section must be present in the object file. This section will contain at least one 32-bit word that identifies specific product-specific extensions that are required by the object module. These bits are allocated by HP, and are intended for use only by HP software. The bits currently defined are listed in Table 5.

  The EF_PARISC_WIDE bit must be set for HP-UX, since the narrow-mode loader does not support ELF object files. Narrow-mode programs must use the SOM format on HP-UX.

  The valid values for the architecture version, encoded in the lower 16 bits of the flags, are listed in Table 6.

  The architecture version must be EFA_PARISC_2_0 for HP-UX, since ELF object files are supported only for wide-mode programs running on PA-RISC 2.0.

**Table 1. Object File Classes, e_ident[EI_CLASS]**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| ELFCLASS32 | 1 | ELF-32 object file |
| ELFCLASS64 | 2 | ELF-64 object file |

**Table 2. Data Encodings, e_ident[EI_DATA]**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| ELFDATA2MSB | 2 | Object file data structures are big-endian |

**Table 3. Target Architecture, e_machine**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| EM_PARISC | 15 | PA-RISC 1.0, 1.1, 2.0 |

**Table 4. Header flags, e_flags**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| EF_PARISC_TRAPNIL | 0x00010000 | Trap nil pointer dereferences |
| EF_PARISC_EXT | 0x00020000 | Program uses arch. extensions |
| EF_PARISC_LSB | 0x00040000 | Program expects little-endian mode |
| EF_PARISC_WIDE | 0x00080000 | Program expects wide mode |
| EF_PARISC_NO_KABP | 0x00100000 | Do not allow kernel-assisted branch prediction |
| EF_PARISC_LAZYSWAP | 0x00400000 | Allow lazy swap for dynamically-allocated program segments |
| EF_PARISC_ARCH | 0x0000FFFF | Architecture version |

**Table 5. Architecture Extension Bits**

| *Value* | *Processor(s)* | *Meaning* |
|---|---|---|
| 0000 0001 | PA 7000 | Quadword store instruction |
| 0000 0002 | PA 7100 | Floating-point loads and stores to I/O space |
| 0000 0004 | PA 7000 | Reciprocal square root instruction |
| 0000 0008 | PA 7000, 7100 | FDC instruction includes graphics flushes |
| 0000 0010 | PA 7100LC, 8000 | Halfword parallel add/subtract/ average instructions |
| 0000 0020 | PA 7100LC, 8000 | Halfword parallel shift-and-add instructions |
| 0000 0040 | PA 7100LC | Byte-swapping stores |
| 0000 0080 | PA 7200, 8000 | Data prefetch via load to GR 0 |

**Table 6. Architecture Version, e_flags & EF_PARISC_ARCH**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| EFA_PARISC_1_0 | 0x020B | PA-RISC 1.0 |
| EFA_PARISC_1_1 | 0x0210 | PA-RISC 1.1 |
| EFA_PARISC_2_0 | 0x0214 | PA-RISC 2.0 |

# 4. Sections

## *Section Indices*

Table 7 lists the special section indices that are specific to the PA-RISC architecture and the HP-UX operating system:

**Table 7. Special Section Indices**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| SHN_PARISC_ANSI_COMMON | 0xFF00 | Indicates a symbol that has been declared as a a tentative definition in an ANSI C compilation |
| SHN_PARISC_HUGE_COMMON | 0xFF01 | Indicates a symbol that has been declared as a common block using the huge memory model |

### *Section Header Entries*

The fields of the section header that have interpretations specific to PA-RISC are explained below.

- sh_type identifies the type of section. Section types specific to PA-RISC are listed in Table 8.
- sh_flags contains bits describing the section attributes. Attributes specific to PA-RISC are listed in Table 9.
- sh_info contains extra information about the section. The use of this field for section types specific to PA-RISC is shown in Table 10.

**Table 8. Section Types, sh_type**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| SHT_PARISC_EXT | 0x70000000 | Section contains product-specific extension bits |
| SHT_PARISC_UNWIND | 0x70000001 | Section contains unwind table entries |
| SHT_PARISC_DOC | 0x70000002 | Section contains debug information for optimized code |

**Table 9. Section Attributes, sh_flags**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| SHF_PARISC_SBP | 0x80000000 | Section contains code compiled for static branch prediction |
| SHF_PARISC_HUGE | 0x40000000 | Section should be allocated far from gp |
| SHF_PARISC_SHORT | 0x20000000 | Section should be near gp |

**Table 10. Use of the sh_info Field**

| *Section Type* | *sh_info* |
|---|---|
| SHT_PARISC_UNWIND | Section index of code section to which the unwind table entries apply |

In addition to the standard section names described in the processor-independent document, the sections listed in Table 11 are defined for use on PA-RISC (the "S" flag stands for SHF_PARISC_SHORT).

**Table 11. Standard Sections Used on PA-RISC**

| *Section Name* | *Section Type* | *Flags* | *Use* |
|---|---|---|---|
| .PARISC.archext | SHT_PARISC_EXT | none | Product-specific extension bits |
| .PARISC.milli | SHT_PROGBITS | A, X | Millicode |
| .PARISC.unwind | SHT_PARISC_UNWIND | A | Unwind table |
| .PARISC.unwind_info | SHT_PROGBITS | A | Stack unwind and exception handling information |
| .sdata | SHT_PROGBITS | A, W, S | Short initialized data |
| .sbss | SHT_NOBITS | A, W, S | Short uninitialized data |

Code sections will be named ".text" unless overridden by a pragma in the source code.

The names of the symbolic debug sections will be determined by each particular language system.

## 5. String tables

There are no processor-specific extensions to the string table format for the PA-RISC architecture.

## 6. Symbol Table

One processor-specific symbol type is defined for PA-RISC, and is listed in Table 12.

**Table 12. Symbol Types**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| STT_PARISC_MILLI | 13 | Identifies the entry point of a millicode routine. |

Millicode routines are identified separately from standard functions so that the linker can build export stubs for standard functions but not for millicode routines.

## 7. Relocations

Because many PA-RISC instructions have small immediate fields, the longer form of relocation ("Rela") is always used for relocatable object files on PA-RISC.

- r_offset contains the virtual address of the reference.
- ELF64_R_TYPE(r_info) contains the type of relocation. The relocation types specific to PA-RISC are described below.
- ELF64_R_SYM(r_info) contains the symbol table index of the symbol whose value should be used in the relocation.

- r_addend contains the constant to be added to symbol value during relocation. Note that the byte order of the addend field is controlled by the e_ident[EI_DATA] field in the ELF header, not by the application.

The relocation types defined for 32-bit (narrow mode) applications on PA-RISC are shown in Table 13. *Note that HP-UX does not support the ELF object file format for narrow-mode applications.* All mnemonics have the prefix "R_PARISC_". Instruction formats are shown with a reference to the format number as defined in the *PA-RISC 2.0 Architecture and Instruction Set Reference Manual*.

**Table 13. Relocations for 32-bit Applications**

| *Name* | *Value* | *Instruction Formats* | *Expression* |
|---|---|---|---|
| NONE | 0 | *none* | *none* |
| DIR32 | 1 | 32-bit word | symbol + addend |
| DIR21L | 2 | Long Immediate (7) | LR(symbol, addend) |
| DIR17R | 3 | Branch External (19) | RR(symbol, addend) |
| DIR17F | 4 | Branch External (19) | symbol + addend |
| DIR14R | 6 | Load/Store (1) | RR(symbol, addend) |
| PCREL21L | 10 | Long Immediate (7) | L(symbol – PC – 8 + addend) |
| PCREL17R | 11 | Branch External (19) | R(symbol – PC – 8 + addend) |
| PCREL17F | 12 | Branch (20) | symbol – PC – 8 + addend |
| PCREL17C | 13 | Branch (20) | symbol – PC – 8 + addend |
| PCREL14R | 14 | Load/Store (1) | R(symbol – PC – 8 + addend) |
| DPREL21L | 18 | Long Immediate (7) | LR(symbol – GP, addend) |
| DPREL14WR | 19 | Load/Store Mod. Comp. (2) | RR(symbol – GP, addend) |
| DPREL14DR | 20 | Load/Store Doubleword (3) | RR(symbol – GP, addend) |
| DPREL14R | 22 | Load/Store (1) | RR(symbol – GP, addend) |
| DLTREL21L | 26 | Long Immediate (7) | LR(symbol – GP, addend) |
| DLTREL14R | 30 | Load/Store (1) | RR(symbol – GP, addend) |
| DLTIND21L | 34 | Long Immediate (7) | L(ltoff(symbol + addend)) |
| DLTIND14R | 38 | Load/Store (1) | R(ltoff(symbol + addend)) |
| DLTIND14F | 39 | Load/Store (1) | ltoff(symbol + addend) |
| SETBASE | 40 | *none* | no relocation; base := symbol |
| SECREL32 | 41 | 32-bit word | symbol – SECT + addend |
| BASEREL21L | 42 | Long Immediate (7) | LR(symbol – base, addend) |
| BASEREL17R | 43 | Branch External (19) | RR(symbol – base, addend) |
| BASEREL14R | 46 | Load/Store (1) | RR(symbol – base, addend) |
| SEGBASE | 48 | *none* | no relocation; SB := symbol |
| SEGREL32 | 49 | 32-bit word | symbol – SB + addend |
| PLTOFF21L | 50 | Long Immediate (7) | LR(pltoff(symbol), addend) |
| PLTOFF14R | 54 | Load/Store (1) | RR(pltoff(symbol), addend) |
| PLTOFF14F | 55 | Load/Store (1) | pltoff(symbol) + addend |

**Table 13. Relocations for 32-bit Applications**

| *Name* | *Value* | *Instruction Formats* | *Expression* |
|---|---|---|---|
| PLABEL32 | 65 | 32-bit word | fptr(symbol) |
| PCREL22C | 73 | Branch & Link (21) | symbol – PC – 8 + addend |
| PCREL22F | 74 | Branch & Link (21) | symbol – PC – 8 + addend |
| PCREL14WR | 75 | Load/Store Mod. Comp. (2) | R(symbol – PC – 8 + addend) |
| PCREL14DR | 76 | Load/Store Doubleword (3) | R(symbol – PC – 8 + addend) |
| DIR14WR | 83 | Load/Store Mod. Comp. (2) | RR(symbol, addend) |
| DIR14DR | 84 | Load/Store Doubleword (3) | RR(symbol, addend) |
| DLTREL14WR | 91 | Load/Store Mod. Comp. (2) | RR(symbol – GP, addend) |
| DLTREL14DR | 92 | Load/Store Doubleword (3) | RR(symbol – GP, addend) |
| DLTIND14WR | 99 | Load/Store Mod. Comp. (2) | R(ltoff(symbol + addend)) |
| DLTIND14DR | 100 | Load/Store Doubleword (3) | R(ltoff(symbol + addend)) |
| BASEREL14WR | 107 | Load/Store Mod. Comp. (2) | RR(symbol – base, addend) |
| BASEREL14DR | 108 | Load/Store Doubleword (3) | RR(symbol – base, addend) |
| PLTOFF14WR | 115 | Load/Store Mod. Comp. (2) | RR(pltoff(symbol), addend) |
| PLTOFF14DR | 116 | Load/Store Doubleword (3) | RR(pltoff(symbol), addend) |
| LORESERVE | 128 | | |
| HIRESERVE | 255 | | |

The range of relocation types from 128 through 255 is reserved for environment-specific use.

The relocation types defined for 64-bit (wide mode) applications on PA-RISC 2.0 are shown in Table 14. Many relocations are defined for both sets of applications; these share the same number (some mnemonics differ, but the definitions are identical when two mnemonics share the same value). All mnemonics have the prefix "R_PARISC_". Instruction formats are shown with a reference to the format number as defined in the *PA-RISC 2.0 Architecture and Instruction Set Reference Manual*.

**Table 14. Relocations for 64-bit Applications**

| *Name* | *Value* | *Instruction Formats* | *Expression* |
|---|---|---|---|
| NONE | 0 | *none* | *none* |
| DIR32 | 1 | 32-bit word | symbol + addend |
| DIR21L | 2 | Long Immediate (7) | LR(symbol, addend) |
| DIR17R | 3 | Branch External (19) | RR(symbol, addend) |
| DIR17F | 4 | Branch External (19) | symbol + addend |
| DIR14R | 6 | Load/Store (1) | RR(symbol, addend) |
| PCREL32 | 9 | 32-bit word | symbol – PC – 8 + addend |
| PCREL21L | 10 | Long Immediate (7) | L(symbol – PC – 8 + addend) |
| PCREL17R | 11 | Branch External (19) | R(symbol – PC – 8 + addend) |
| PCREL17F | 12 | Branch (20) | symbol – PC – 8 + addend |

**Table 14. Relocations for 64-bit Applications**

| *Name* | *Value* | *Instruction Formats* | *Expression* |
|---|---|---|---|
| PCREL14R | 14 | Load/Store (1) | R(symbol – PC – 8 + addend) |
| GPREL21L | 26 | Long Immediate (7) | LR(symbol – GP, addend) |
| GPREL14R | 30 | Load/Store (1) | RR(symbol – GP, addend) |
| LTOFF21L | 34 | Long Immediate (7) | L(ltoff(symbol + addend)) |
| LTOFF14R | 38 | Load/Store (1) | R(ltoff(symbol + addend)) |
| SECREL32 | 41 | 32-bit word | symbol – SECT + addend |
| SEGBASE | 48 | *none* | no relocation; SB := symbol |
| SEGREL32 | 49 | 32-bit word | symbol – SB + addend |
| PLTOFF21L | 50 | Long Immediate (7) | LR(pltoff(symbol), addend) |
| PLTOFF14R | 54 | Load/Store (1) | RR(pltoff(symbol), addend) |
| LTOFF_FPTR32 | 57 | 32-bit word | ltoff(fptr(symbol+addend)) |
| LTOFF_FPTR21L | 58 | Long Immediate (7) | L(ltoff(fptr(symbol+addend))) |
| LTOFF_FPTR14R | 62 | Load/Store (1) | R(ltoff(fptr(symbol+addend))) |
| FPTR64 | 64 | 64-bit doubleword | fptr(symbol+addend) |
| PCREL64 | 72 | 64-bit doubleword | symbol – PC – 8 + addend |
| PCREL22F | 74 | Branch & Link (21) | symbol – PC – 8 + addend |
| PCREL14WR | 75 | Load/Store Mod. Comp. (2) | R(symbol – PC – 8 + addend) |
| PCREL14DR | 76 | Load/Store Doubleword (3) | R(symbol – PC – 8 + addend) |
| PCREL16F | 77 | Load/Store (1) | symbol – PC – 8 + addend |
| PCREL16WF | 78 | Load/Store Mod. Comp. (2) | symbol – PC – 8 + addend |
| PCREL16DF | 79 | Load/Store Doubleword (3) | symbol – PC – 8 + addend |
| DIR64 | 80 | 64-bit doubleword | symbol + addend |
| DIR14WR | 83 | Load/Store Mod. Comp. (2) | RR(symbol, addend) |
| DIR14DR | 84 | Load/Store Doubleword (3) | RR(symbol, addend) |
| DIR16F | 85 | Load/Store (1) | symbol + addend |
| DIR16WF | 86 | Load/Store Mod. Comp. (2) | symbol + addend |
| DIR16DF | 87 | Load/Store Doubleword (3) | symbol + addend |
| GPREL64 | 88 | 64-bit doubleword | symbol – GP + addend |
| GPREL14WR | 91 | Load/Store Mod. Comp. (2) | RR(symbol – GP, addend) |
| GPREL14DR | 92 | Load/Store Doubleword (3) | RR(symbol – GP, addend) |
| GPREL16F | 93 | Load/Store (1) | symbol – GP + addend |
| GPREL16WF | 94 | Load/Store Mod. Comp. (2) | symbol – GP + addend |
| GPREL16DF | 95 | Load/Store Doubleword (3) | symbol – GP + addend |
| LTOFF64 | 96 | 64-bit doubleword | ltoff(symbol + addend) |
| LTOFF14WR | 99 | Load/Store Mod. Comp. (2) | R(ltoff(symbol + addend)) |
| LTOFF14DR | 100 | Load/Store Doubleword (3) | R(ltoff(symbol + addend)) |
| LTOFF16F | 101 | Load/Store (1) | ltoff(symbol + addend) |

**Table 14. Relocations for 64-bit Applications**

| *Name* | *Value* | *Instruction Formats* | *Expression* |
|---|---|---|---|
| LTOFF16WF | 102 | Load/Store Mod. Comp. (2) | ltoff(symbol + addend) |
| LTOFF16DF | 103 | Load/Store Doubleword (3) | ltoff(symbol + addend) |
| SECREL64 | 104 | 64-bit doubleword | symbol – SECT + addend |
| SEGREL64 | 112 | 64-bit doubleword | symbol – SB + addend |
| PLTOFF14WR | 115 | Load/Store Mod. Comp. (2) | RR(pltoff(symbol), addend) |
| PLTOFF14DR | 116 | Load/Store Doubleword (3) | RR(pltoff(symbol), addend) |
| PLTOFF16F | 117 | Load/Store (1) | pltoff(symbol) + addend |
| PLTOFF16WF | 118 | Load/Store Mod. Comp. (2) | pltoff(symbol) + addend |
| PLTOFF16DF | 119 | Load/Store Doubleword (3) | pltoff(symbol) + addend |
| LTOFF_FPTR64 | 120 | 64-bit doubleword | ltoff(fptr(symbol+addend)) |
| LTOFF_FPTR14WR | 123 | Load/Store Mod. Comp. (2) | R(ltoff(fptr(symbol+addend))) |
| LTOFF_FPTR14DR | 124 | Load/Store Doubleword (3) | R(ltoff(fptr(symbol+addend))) |
| LTOFF_FPTR16F | 125 | Load/Store (1) | ltoff(fptr(symbol+addend)) |
| LTOFF_FPTR16WF | 126 | Load/Store Mod. Comp. (2) | ltoff(fptr(symbol+addend)) |
| LTOFF_FPTR16DF | 127 | Load/Store Doubleword (3) | ltoff(fptr(symbol+addend)) |
| LORESERVE | 128 | | Environment-specific use |
| COPY | 128 | data | Dynamic relocations only; see text |
| IPLT | 129 | PLT | |
| EPLT | 130 | PLT | |
| TPREL32 | 153 | 32-bit word | symbol – TP + addend |
| TPREL21L | 154 | Long Immediate (7) | LR(symbol – TP, addend) |
| TPREL14R | 158 | Load/Store (1) | RR(symbol – TP, addend) |
| LTOFF_TP21L | 162 | Long Immediate (7) | L(ltoff(symbol – TP + addend)) |
| LTOFF_TP14R | 166 | Load/Store (1) | R(ltoff(symbol – TP + addend)) |
| LTOFF_TP14F | 167 | Load/Store (1) | ltoff(symbol – TP + addend) |
| TPREL64 | 216 | 64-bit word | symbol – TP + addend |
| TPREL14WR | 219 | Load/Store Mod. Comp. (2) | RR(symbol – TP, addend) |
| TPREL14DR | 220 | Load/Store Doubleword (3) | RR(symbol – TP, addend) |
| TPREL16F | 221 | Load/Store (1) | symbol – TP + addend |
| TPREL16WF | 222 | Load/Store Mod. Comp. (2) | symbol – TP + addend |
| TPREL16DF | 223 | Load/Store Doubleword (3) | symbol – TP + addend |
| LTOFF_TP64 | 224 | 64-bit doubleword | ltoff(symbol – TP + addend) |
| LTOFF_TP14WR | 227 | Load/Store Mod. Comp. (2) | R(ltoff(symbol – TP + addend)) |
| LTOFF_TP14DR | 228 | Load/Store Doubleword (3) | R(ltoff(symbol – TP + addend)) |
| LTOFF_TP16F | 229 | Load/Store (1) | ltoff(symbol – TP + addend) |
| LTOFF_TP16WF | 230 | Load/Store Mod. Comp. (2) | ltoff(symbol – TP + addend) |

**Table 14. Relocations for 64-bit Applications**

| *Name* | *Value* | *Instruction Formats* | *Expression* |
|---|---|---|---|
| LTOFF_TP16DF | 231 | Load/Store Doubleword (3) | ltoff(symbol – TP + addend) |
| HIRESERVE | 255 | | |

The range of relocation types from 128 through 255 is reserved for environment-specific use.

The functions and special symbols used in the expression column are defined below:

- GP stands for the gp value for the current load module. This is used for computing a gp-relative displacement. (In 32-bit programs, this is the value of the symbol __dp or __dlt, depending on whether the module is PIC or not. In 64-bit programs, the gp value is assigned by the linker.)
- PC stands for the address of the current instruction. This is used to compute a pc-relative displacement.
- SB stands for the segment base address used for segment-relative relocations. Every segment-relative relocation uses an implicit segment base, which is designated by the nearest preceding SEGBASE relocation within the same relocation section.
- SECT stands for the address of the beginning of the (output) section containing the data item of instruction being relocated.
- L(), R(), LR(), RR() are field selector functions. These are described in Figure 1.
- ltoff() requests the creation of a linkage table entry that will hold the full value of the effective address, and computes the gp-relative displacement of that linkage table entry.
- fptr() evaluates to the address of the “official” plabel descriptor for the given symbol.
- pltoff() requests the creation of a local procedure linkage table (PLT) entry for the given symbol, and computes the gp-relative displacement of that PLT entry. The dynamic loader will copy the “official” plabel descriptor into the PLT entry.
- TP stands for the link-time “value” of the thread pointer, for computing tp-relative offsets. The link-time value of the thread pointer or of a thread-local symbol has no meaning except in computing tp-relative offsets.

The IPLT, EPLT, and COPY relocations are used for dynamic relocations only, and are described in Section 10.

All direct, dp/gp-relative, dlt-relative, and base-relative effective address calculations use the LR and RR rounding modes. In these rounding modes, the left part is computed based on a rounded addend instead of the actual addend. The addend is rounded to the nearest multiple of 8K ($2^{13}$) prior to computing the effective address. The right part is computed as the difference between the full value of the expression and the value used in the corresponding left-part relocation. Because the difference between the original addend and the rounded addend can be no larger than ±4K, this result will always fit in a

signed 14- or 16-bit field. This permits several load and store instructions to reuse the result of a single ADDIL or LDIL instruction, as long as the symbol index and the rounded value of the addend are identical.

For pc-relative relocations, the standard L and R rounding modes are used. The expression is computed based on the actual effective address.

The C language functions in Figure 1 define the operation of the LR, RR, L, and R functions.

```
unsigned long LR(unsigned long x, unsigned long addend)
{
        return L(x + RND(addend));
}

unsigned long RR(unsigned long x, unsigned long addend)
{
        return R(x + RND(addend)) + (addend – RND(addend));
}

unsigned long L(unsigned long x)
{
        return (x & 0xfffff800);
}

unsigned long R(unsigned long x)
{
        return (x & 0x000007ff);
}

unsigned long RND(unsigned long x)
{
        return ((x + 0x1000) & ~0x1fff);
}
```

**Figure 1. Rounding Operations**

The relocations are numbered in a pattern for convenience in decoding them into expression types and instruction formats. The following two tables show how these properties can be derived from the relocation number. Table 15 identifies the instruction format, based on the low-order three bits of the relocation type, and the bit in the $2^6$ position. Table 16 identifies the expression type, based on the result of dividing the relocation type by 8 (discarding the remainder). The basic expression type may be modified by L, R, LR, and RR field selector functions, based on the instruction format.

**Table 15. Deriving the Instruction Format from the Relocation Type**

| | *Instruction Format* | |
|---|---|---|
| | *x0xx xxxx* | *x1xx xxxx* |
| xxxx x000 | None | 64-bit doubleword |
| xxxx x001 | 32-bit word | 32-bit word or Branch & Link (21) |
| xxxx x010 | Long Immediate (7) | Branch & Link (21) |
| xxxx x011 | Branch External (19) | Load/Store Mod. Comp. (2), 14-bit disp. |
| xxxx x100 | Branch External (19) or Branch (20) | Load/Store Doubleword (3), 14-bit disp. |
| xxxx x101 | Branch (20) | Load/Store (1), 16-bit disp. |
| xxxx x110 | Load/Store (1), 14-bit disp. | Load/Store Mod. Comp. (2), 16-bit disp. |
| xxxx x111 | Load/Store (1), 14-bit disp. | Load/Store Doubleword (3), 16-bit disp. |

**Table 16. Deriving the Expression Type from the Relocation Type**

| *Type/8* | *Basic Expression Type* |
|---|---|
| 0 or 10 | symbol + addend |
| 1 or 9 | symbol – PC + addend |
| 2 | symbol – GP + addend (32-bit non-PIC) |
| 3 or 11 | symbol – GP + addend |
| 4 or 12 | ltoff(symbol + addend) |
| 5 or 13 | symbol – SECT + addend (SECREL32, SECREL64)<br>symbol – base + addend (all others) |
| 6 or 14 | symbol – SB + addend (SEGREL32, SEGREL64)<br>pltoff(symbol) + addend (all others) |
| 7 or 15 | ltoff(fptr(symbol)) |
| 8 | fptr(symbol) |

Each instruction format implies how the bits are copied from the expression value to the instruction being relocated. The formats and dispersal of these bits are illustrated in Figure 2. In the illustrations, only the low-order 32 bits of the expression value are shown.

## 8. Program header table

Table 17 lists the segment types that are specific to PA-RISC. Table 18 lists the segment attributes that are specific to PA-RISC.

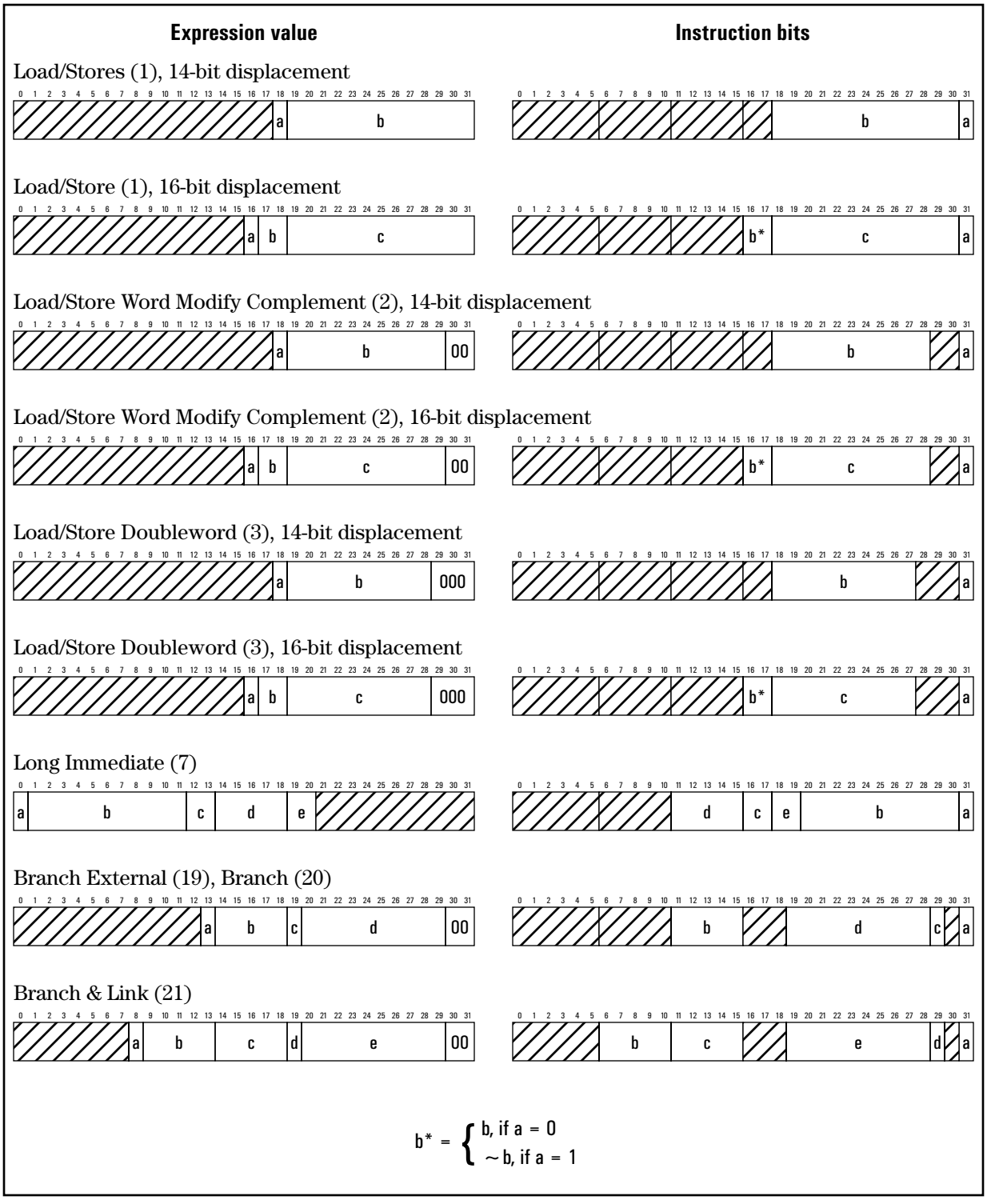


**Figure 2. Instruction Formats**

**Table 17. Segment Types, p_type**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| PT_PARISC_ARCHEXT | 0x70000000 | Segment contains .PARISC.archext section, as described in Section 3. If this program header table entry is present, it must precede all entries of type PT_LOAD or PT_INTERP. |
| PT_PARISC_UNWIND | 0x70000001 | Segment contains the .unwind section. |

**Table 18. Segment Attributes, p_flags**

| *Name* | *Value* | *Meaning* |
|---|---|---|
| PF_PARISC_SBP | 0x08000000 | Segment contains code compiled for static branch prediction |

## 9. Note sections

There are no note section types defined specifically for PA-RISC.

## 10. Dynamic table

Initialization and termination functions are specified using the .init and .fini sections, with the DT_INITARRAY, DT_FINIARRAY, DT_INITARRAYSZ, and DT_FINIARRAYSZ dynamic table entries. The dynamic table entries define the base address and length of the .init and .fini sections, which each contain an array of function pointers. When a module is first loaded (at startup for the main program), each function in the .init list is called in sequence. These initializers are invoked for DLLs before those for the main program. Similarly, the functions in the .fini list are called when a module is unloaded. HP-UX does not support the DT_INIT and DT_FINI dynamic table entries.

The DT_PLTGOT dynamic table entry is used to specify the global pointer (gp) value for the load module. The d_ptr field contains a link-time virtual address that must be relocated by the dynamic loader. The relocated value is used as the gp value for all functions in the load module.

Dynamic relocations use the "Rel" relocation format, where the addend is obtained from the word being relocated. The following relocations may be used as dynamic relocations: NONE, DIR32, DIR64, and FPTR64.

The following HP-specific and HP-UX-specific relocations may also be used as dynamic relocations: TPREL64, IPLT, EPLT, and COPY.

The IPLT relocation refers to a procedure linkage table (PLT) entry that must be initialized with a copy of the function descriptor for the given symbol. The linker creates these dynamic relocations for the PLT entry that is created on behalf of a PLTOFF relocation.

The EPLT relocation refers to an function descriptor for a function entry point. The linker creates a function descriptor for each exported functions and for

each non-exported function that has been referenced by an fptr() operation. A function descriptor is 16 bytes in length. The dynamic loader must relocate the first 8-byte doubleword with the address of the function entry point, and the second doubleword with the gp value for the current load module.

The COPY relocation is used for data defined in a DLL that must be copied into this load module's data segment. The symbol referenced by this relocation specifies a symbol that should be defined both in this load module and in a DLL. At execution time, the dynamic loader copies the data associated with the symbol from the DLL to the location specified by the offset field of the relocation.

## 11. Hash table

There are no processor-specific extensions to the hash table for the PA-RISC architecture.