Contents

1 BPF Instruction Set Architecture (ISA)

eBPF (which is no longer an acronym for anything), also commonly referred to as BPF, is a technology with origins in the Linux kernel that can run untrusted programs in a privileged context such as an operating system kernel. This document specifies the BPF instruction set architecture (ISA).

1.1 Documentation conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 https://www.rfc-editor.org/info/rfc2119 RFC8174 when, and only when, they appear in all capitals, as shown here.

For brevity and consistency, this document refers to families of types using a shorthand syntax and refers to several expository, mnemonic functions when describing the semantics of instructions. The range of valid values for those types and the semantics of those functions are defined in the following subsections.

1.1.1 Types

This document refers to integer types with the notation *SN* to specify a type's signedness (*S*) and bit width (*N*), respectively.

Meaning of signedness notation

S	Meaning
u	unsigned
s	signed

Meaning of bit-width notation

N	Bit width
8	8 bits
16	16 bits
32	32 bits
64	64 bits
128	128 bits

For example, *u32* is a type whose valid values are all the 32-bit unsigned numbers and *s16* is a type whose valid values are all the 16-bit signed numbers.

1.1.2 Functions

- htobe16: Takes an unsigned 16-bit number in host-endian format and returns the equivalent number as an unsigned 16-bit number in big-endian format.
- htobe32: Takes an unsigned 32-bit number in host-endian format and returns the equivalent number as an unsigned 32-bit number in big-endian format.
- htobe64: Takes an unsigned 64-bit number in host-endian format and returns the equivalent number as an unsigned 64-bit number in big-endian format.
- htole16: Takes an unsigned 16-bit number in host-endian format and returns the equivalent number as an unsigned 16-bit number in little-endian format.
- htole32: Takes an unsigned 32-bit number in host-endian format and returns the equivalent number as an unsigned 32-bit number in little-endian format.
- htole64: Takes an unsigned 64-bit number in host-endian format and returns the equivalent number as an unsigned 64-bit number in little-endian format.
- bswap16: Takes an unsigned 16-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.
- bswap32: Takes an unsigned 32-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.
- bswap64: Takes an unsigned 64-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.

1.1.3 Definitions

Example

Sign extend an 8-bit number `A` to a 16-bit number `B` on a big-endian platform:

```
A:          10000110
B: 11111111 10000110
```

1.1.4 Conformance groups

An implementation does not need to support all instructions specified in this document (e.g., deprecated instructions). Instead, a number of conformance groups are specified. An implementation MUST support the base32 conformance group and MAY support additional conformance groups, where supporting a conformance group means it MUST support all instructions in that conformance group.

The use of named conformance groups enables interoperability between a runtime that executes instructions, and tools such as compilers that generate instructions for the runtime. Thus, capability discovery in terms of conformance groups might be done manually by users or automatically by tools.

Each conformance group has a short ASCII label (e.g., "base32") that corresponds to a set of instructions that are mandatory. That is, each instruction has one or more conformance groups of which it is a member.

This document defines the following conformance groups:

- base32: includes all instructions defined in this specification unless otherwise noted.
- base64: includes base32, plus instructions explicitly noted as being in the base64 conformance group.
- atomic32: includes 32-bit atomic operation instructions (see Atomic operations).
- atomic64: includes atomic32, plus 64-bit atomic operation instructions.
- divmul32: includes 32-bit division, multiplication, and modulo instructions.
- divmul64: includes divmul32, plus 64-bit division, multiplication, and modulo instructions.
- packet: deprecated packet access instructions.

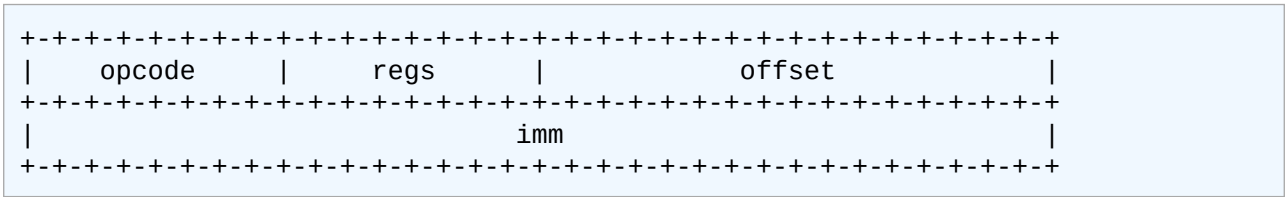
1.2 Instruction encoding

BPF has two instruction encodings:

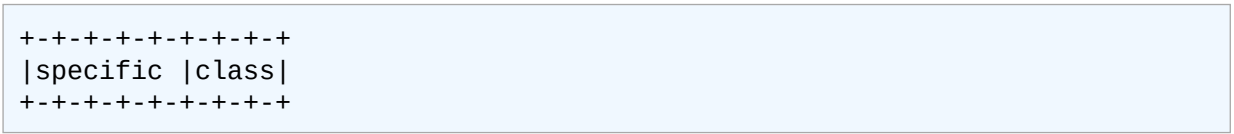
- the basic instruction encoding, which uses 64 bits to encode an instruction
- the wide instruction encoding, which appends a second 64 bits after the basic instruction for a total of 128 bits.

1.2.1 Basic instruction encoding

A basic instruction is encoded as follows:



opcode
: operation to perform, encoded as follows:



specific
: The format of these bits varies by instruction class

class
: The instruction class (see Instruction classes)

regs

The source and destination register numbers, encoded as follows on a little-endian host:

```
+-+-+-+-+-+-+-+-+
|src_reg|dst_reg|
+-+-+-+-+-+-+-+-+
```

and as follows on a big-endian host:

```
+-+-+-+-+-+-+-+-+
|dst_reg|src_reg|
+-+-+-+-+-+-+-+-+
```

src_reg

the source register number (0-10), except where otherwise specified (64-bit immediate instructions reuse this field for other purposes)

dst_reg

destination register number (0-10), unless otherwise specified (future instructions might reuse this field for other purposes)

offset

signed integer offset used with pointer arithmetic, except where otherwise specified (some arithmetic instructions reuse this field for other purposes)

imm

signed integer immediate value

Note that the contents of multi-byte fields ('offset' and 'imm') are stored using big-endian byte ordering on big-endian hosts and little-endian byte ordering on little-endian hosts.

For example:

```
opcode                  offset imm          assembly
       src_reg dst_reg
07     0       1        00 00  44 33 22 11  r1 += 0x11223344 // little
       dst_reg src_reg
07     1       0        00 00  11 22 33 44  r1 += 0x11223344 // big
```

Note that most instructions do not use all of the fields. Unused fields SHALL be cleared to zero.

1.2.2 Wide instruction encoding

Some instructions are defined to use the wide instruction encoding, which uses two 32-bit immediate values. The 64 bits following the basic instruction format contain a pseudo instruction with 'opcode', 'dst_reg', 'src_reg', and 'offset' all set to zero.

This is depicted in the following figure:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    opcode     |     regs      |            offset             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               imm                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           reserved                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           next_imm                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

opcode
: operation to perform, encoded as explained above

regs
: The source and destination register numbers (unless otherwise specified), encoded as explained above

offset
: signed integer offset used with pointer arithmetic, unless otherwise specified

imm
: signed integer immediate value

reserved
: unused, set to zero

next_imm
: second signed integer immediate value

1.2.3 Instruction classes

The three least significant bits of the 'opcode' field store the instruction class:

Instruction class

class	value	description	reference
LD	0x0	non-standard load operations	Load and store instructions
LDX	0x1	load into register operations	Load and store instructions
ST	0x2	store from immediate operations	Load and store instructions
STX	0x3	store from register operations	Load and store instructions
ALU	0x4	32-bit arithmetic operations	Arithmetic and jump instructions
JMP	0x5	64-bit jump operations	Arithmetic and jump instructions
JMP32	0x6	32-bit jump operations	Arithmetic and jump instructions
ALU64	0x7	64-bit arithmetic operations	Arithmetic and jump instructions

1.3 Arithmetic and jump instructions

For arithmetic and jump instructions (`ALU`, `ALU64`, `JMP` and `JMP32`), the 8-bit 'opcode' field is divided into three parts:

```
+-+-+-+-+-+-+-+-+
|  code |s|class|
+-+-+-+-+-+-+-+-+
```

code
: the operation code, whose meaning varies by instruction class

s (source)
: the source operand location, which unless otherwise specified is one of:

Source operand location

source	value	description
K	0	use 32-bit 'imm' value as source operand
X	1	use 'src_reg' register value as source operand

instruction class

the instruction class (see Instruction classes)

1.3.1 Arithmetic instructions

ALU uses 32-bit wide operands while ALU64 uses 64-bit wide operands for otherwise identical operations. ALU64 instructions belong to the base64 conformance group unless noted otherwise. The 'code' field encodes the operation as below, where 'src' refers to the the source operand and 'dst' refers to the value of the destination register.

Arithmetic instructions

name	code	offset	description
ADD	0x0	0	dst += src
SUB	0x1	0	dst -= src
MUL	0x2	0	dst *= src
DIV	0x3	0	dst = (src != 0) ? (dst / src) : 0
SDIV	0x3	1	dst = (src != 0) ? (dst s/ src) : 0
OR	0x4	0	dst |= src
AND	0x5	0	dst &= src
LSH	0x6	0	dst <<= (src & mask)
RSH	0x7	0	dst >>= (src & mask)
NEG	0x8	0	dst = -dst
MOD	0x9	0	dst = (src != 0) ? (dst % src) : dst
SMOD	0x9	1	dst = (src != 0) ? (dst s% src) : dst
XOR	0xa	0	dst ^= src
MOV	0xb	0	dst = src
MOVSX	0xb	8/16/32	dst = (s8,s16,s32)src
ARSH	0xc	0	:term:`sign extending<Sign Extend>` dst >>= (src & mask)
END	0xd	0	byte swap operations (see Byte swap instructions below)

Underflow and overflow are allowed during arithmetic operations, meaning the 64-bit or 32-bit value will wrap. If BPF program execution would result in division by zero, the destination register is instead set to zero. If execution would result in modulo by zero, for ALU64 the value of the destination register is unchanged whereas for ALU the upper 32 bits of the destination register are zeroed.

{ADD, X, ALU}, where 'code' = ADD, 'source' = X, and 'class' = ALU, means:

```
dst = (u32) ((u32) dst + (u32) src)
```

where '(u32)' indicates that the upper 32 bits are zeroed.

{ADD, X, ALU64} means:

```
dst = dst + src
```

{XOR, K, ALU} means:

```
dst = (u32) dst ^ (u32) imm
```

{XOR, K, ALU64} means:

```
dst = dst ^ imm
```

Note that most arithmetic instructions have 'offset' set to 0. Only three instructions (SDIV, SMOD, MOVSX) have a non-zero 'offset'.

Division, multiplication, and modulo operations for ALU are part of the "divmul32" conformance group, and division, multiplication, and modulo operations for ALU64 are part of the "divmul64" conformance group. The division and modulo operations support both unsigned and signed flavors.

For unsigned operations (DIV and MOD), for ALU, 'imm' is interpreted as a 32-bit unsigned value. For ALU64, 'imm' is first :term:`sign extended<Sign Extend>` from 32 to 64 bits, and then interpreted as a 64-bit unsigned value.

For signed operations (SDIV and SMOD), for ALU, 'imm' is interpreted as a 32-bit signed value. For ALU64, 'imm' is first :term:`sign extended<Sign Extend>` from 32 to 64 bits, and then interpreted as a 64-bit signed value.

Note that there are varying definitions of the signed modulo operation when the dividend or divisor are negative, where implementations often vary by language such that Python, Ruby, etc. differ from C, Go, Java, etc. This specification requires that signed modulo MUST use truncated division (where -13 % 3 == -1) as implemented in C, Go, etc.:

```
a % n = a - n * trunc(a / n)
```

The MOVSX instruction does a move operation with sign extension. {MOVSX, X, ALU} :term:`sign extends<Sign Extend>` 8-bit and 16-bit operands into 32-bit operands, and zeroes the remaining upper 32 bits. {MOVSX, X, ALU64} :term:`sign extends<Sign Extend>` 8-bit, 16-bit, and 32-bit operands into 64-bit operands. Unlike other arithmetic instructions, MOVSX is only defined for register source operands (X).

{MOV, K, ALU64} means:

```
dst = (s64)imm
```

{MOV, X, ALU} means:

```
dst = (u32)src
```

{MOVSX, X, ALU} with 'offset' 8 means:

```
dst = (u32)(s32)(s8)src
```

The NEG instruction is only defined when the source bit is clear (K).

Shift operations use a mask of 0x3F (63) for 64-bit operations and 0x1F (31) for 32-bit operations.

1.3.2 Byte swap instructions

The byte swap instructions use instruction classes of ALU and ALU64 and a 4-bit 'code' field of END.

The byte swap instructions operate on the destination register only and do not use a separate source register or immediate value.

For ALU, the 1-bit source operand field in the opcode is used to select what byte order the operation converts from or to. For ALU64, the 1-bit source operand field in the opcode is reserved and MUST be set to 0.

Byte swap instructions

class	source	value	description
ALU	TO_LE	0	convert between host byte order and little endian
ALU	TO_BE	1	convert between host byte order and big endian
ALU64	Reserved	0	do byte swap unconditionally

The 'imm' field encodes the width of the swap operations. The following widths are supported: 16, 32 and 64. Width 64 operations belong to the base64 conformance group and other swap operations belong to the base32 conformance group.

Examples:

{END, TO_LE, ALU} with 'imm' = 16/32/64 means:

```
dst = htole16(dst)
dst = htole32(dst)
dst = htole64(dst)
```

{END, TO_BE, ALU} with 'imm' = 16/32/64 means:

```
dst = htobe16(dst)
dst = htobe32(dst)
dst = htobe64(dst)
```

{END, TO_LE, ALU64} with 'imm' = 16/32/64 means:

```
dst = bswap16(dst)
dst = bswap32(dst)
dst = bswap64(dst)
```

1.3.3 Jump instructions

JMP32 uses 32-bit wide operands and indicates the base32 conformance group, while JMP uses 64-bit wide operands for otherwise identical operations, and indicates the base64 conformance group unless otherwise specified. The 'code' field encodes the operation as below:

Jump instructions

code	value	src_reg	description	notes
JA	0x0	0x0	PC += offset	{JA, K, JMP} only
JA	0x0	0x0	PC += imm	{JA, K, JMP32} only
JEQ	0x1	any	PC += offset if dst == src	
JGT	0x2	any	PC += offset if dst > src	unsigned
JGE	0x3	any	PC += offset if dst >= src	unsigned
JSET	0x4	any	PC += offset if dst & src	
JNE	0x5	any	PC += offset if dst != src	
JSGT	0x6	any	PC += offset if dst > src	signed
JSGE	0x7	any	PC += offset if dst >= src	signed
CALL	0x8	0x0	call helper function by static ID	{CALL, K, JMP} only, see Helper functions
CALL	0x8	0x1	call PC += imm	{CALL, K, JMP} only, see Program-local functions
CALL	0x8	0x2	call helper function by BTF ID	{CALL, K, JMP} only, see Helper functions
EXIT	0x9	0x0	return	{CALL, K, JMP} only
JLT	0xa	any	PC += offset if dst < src	unsigned
JLE	0xb	any	PC += offset if dst <= src	unsigned
JSLT	0xc	any	PC += offset if dst < src	signed
JSLE	0xd	any	PC += offset if dst <= src	signed

where 'PC' denotes the program counter, and the offset to increment by is in units of 64-bit instructions relative to the instruction following the jump instruction. Thus 'PC += 1' skips execution of the next instruction if it's a basic instruction or results in undefined behavior if the next instruction is a 128-bit wide instruction.

Example:

`{JSGE, X, JMP32}` means:

```
if (s32)dst s>= (s32)src goto +offset
```

where 's>=' indicates a signed '>=' comparison.

{JLE, K, JMP} means:

```
if dst <= (u64)(s64)imm goto +offset
```

{JA, K, JMP32} means:

```
gotol +imm
```

where 'imm' means the branch offset comes from the 'imm' field.

Note that there are two flavors of JA instructions. The JMP class permits a 16-bit jump offset specified by the 'offset' field, whereas the JMP32 class permits a 32-bit jump offset specified by the 'imm' field. A > 16-bit conditional jump may be converted to a < 16-bit conditional jump plus a 32-bit unconditional jump.

All CALL and JA instructions belong to the base32 conformance group.

1.3.3.1 Helper functions

Helper functions are a concept whereby BPF programs can call into a set of function calls exposed by the underlying platform.

Historically, each helper function was identified by a static ID encoded in the 'imm' field. The available helper functions may differ for each program type, but static IDs are unique across all program types.

Platforms that support the BPF Type Format (BTF) support identifying a helper function by a BTF ID encoded in the 'imm' field, where the BTF ID identifies the helper name and type. Further documentation of BTF is outside the scope of this document and is left for future work.

1.3.3.2 Program-local functions

Program-local functions are functions exposed by the same BPF program as the caller, and are referenced by offset from the instruction following the call instruction, similar to JA. The offset is encoded in the 'imm' field of the call instruction. An EXIT within the program-local function will return to the caller.

1.4 Load and store instructions

For load and store instructions (LD, LDX, ST, and STX), the 8-bit 'opcode' field is divided as follows:

```
+-+-+-+-+-+-+-+-+
|mode |sz |class|
+-+-+-+-+-+-+-+-+
```

mode

The mode modifier is one of:

Mode modifier

mode modifier	value	description	reference
IMM	0	64-bit immediate instructions	64-bit immediate instructions
ABS	1	legacy BPF packet access (absolute)	Legacy BPF Packet access instructions
IND	2	legacy BPF packet access (indirect)	Legacy BPF Packet access instructions
MEM	3	regular load and store operations	Regular load and store operations
MEMSX	4	sign-extension load operations	Sign-extension load operations
ATOMIC	6	atomic operations	Atomic operations

sz (size)

The size modifier is one of:

Size modifier

size	value	description
W	0	word (4 bytes)
H	1	half word (2 bytes)
B	2	byte
DW	3	double word (8 bytes)

Instructions using `DW` belong to the base64 conformance group.

class

The instruction class (see Instruction classes)

1.4.1 Regular load and store operations

The `MEM` mode modifier is used to encode regular load and store instructions that transfer data between a register and memory.

`{MEM, <size>, STX}` means:

```
*(size *) (dst + offset) = src
```

`{MEM, <size>, ST}` means:

```
*(size *) (dst + offset) = imm
```

`{MEM, <size>, LDX}` means:

```
dst = *(unsigned size *) (src + offset)
```

Where '<size>' is one of: `B`, `H`, `W`, or `DW`, and 'unsigned size' is one of: u8, u16, u32, or u64.

1.4.2 Sign-extension load operations

The `MEMSX` mode modifier is used to encode :term:`sign-extension<Sign Extend>` load instructions that transfer data between a register and memory.

`{MEMSX, <size>, LDX}` means:

```
dst = *(signed size *) (src + offset)
```

Where '<size>' is one of: `B`, `H`, or `W`, and 'signed size' is one of: s8, s16, or s32.

1.4.3 Atomic operations

Atomic operations are operations that operate on memory and can not be interrupted or corrupted by other access to the same memory region by other BPF programs or means outside of this specification.

All atomic operations supported by BPF are encoded as store operations that use the `ATOMIC` mode modifier as follows:

- `{ATOMIC, W, STX}` for 32-bit operations, which are part of the "atomic32" conformance group.
- `{ATOMIC, DW, STX}` for 64-bit operations, which are part of the "atomic64" conformance group.

- 8-bit and 16-bit wide atomic operations are not supported.

The 'imm' field is used to encode the actual atomic operation. Simple atomic operation use a subset of the values defined to encode arithmetic operations in the 'imm' field to encode the atomic operation:

Simple atomic operations

imm	value	description
ADD	0x00	atomic add
OR	0x40	atomic or
AND	0x50	atomic and
XOR	0xa0	atomic xor

`{ATOMIC, W, STX}` with 'imm' = ADD means:

```
*(u32 *)(dst + offset) += src
```

`{ATOMIC, DW, STX}` with 'imm' = ADD means:

```
*(u64 *)(dst + offset) += src
```

In addition to the simple atomic operations, there also is a modifier and two complex atomic operations:

Complex atomic operations

imm	value	description
FETCH	0x01	modifier: return old value
XCHG	0xe0 | FETCH	atomic exchange
CMPXCHG	0xf0 | FETCH	atomic compare and exchange

The `FETCH` modifier is optional for simple atomic operations, and always set for the complex atomic operations. If the `FETCH` flag is set, then the operation also overwrites `src` with the value that was in memory before it was modified.

The `XCHG` operation atomically exchanges `src` with the value addressed by `dst + offset`.

The `CMPXCHG` operation atomically compares the value addressed by `dst + offset` with `R0`. If they match, the value addressed by `dst + offset` is replaced with `src`. In either case, the value that was at `dst + offset` before the operation is zero-extended and loaded back to `R0`.

1.4.4 64-bit immediate instructions

Instructions with the `IMM` 'mode' modifier use the wide instruction encoding defined in Instruction encoding, and use the 'src_reg' field of the basic instruction to hold an opcode subtype.

The following table defines a set of `{IMM, DW, LD}` instructions with opcode subtypes in the 'src_reg' field, using new terms such as "map" defined further below:

64-bit immediate instructions

src_reg	pseudocode	imm type	dst type
0x0	dst = (next_imm << 32) | imm	integer	integer
0x1	dst = map_by_fd(imm)	map fd	map

0x2	dst = map_val(map_by_fd(imm)) + next_imm	map fd	data address
0x3	dst = var_addr(imm)	variable id	data address
0x4	dst = code_addr(imm)	integer	code address
0x5	dst = map_by_idx(imm)	map index	map
0x6	dst = map_val(map_by_idx(imm)) + next_imm	map index	data address

where

- map_by_fd(imm) means to convert a 32-bit file descriptor into an address of a map (see Maps)
- map_by_idx(imm) means to convert a 32-bit index into an address of a map
- map_val(map) gets the address of the first value in a given map
- var_addr(imm) gets the address of a platform variable (see Platform Variables) with a given id
- code_addr(imm) gets the address of the instruction at a specified relative offset in number of (64-bit) instructions
- the 'imm type' can be used by disassemblers for display
- the 'dst type' can be used for verification and JIT compilation purposes

1.4.4.1 Maps

Maps are shared memory regions accessible by BPF programs on some platforms. A map can have various semantics as defined in a separate document, and may or may not have a single contiguous memory region, but the 'map_val(map)' is currently only defined for maps that do have a single contiguous memory region.

Each map can have a file descriptor (fd) if supported by the platform, where 'map_by_fd(imm)' means to get the map with the specified file descriptor. Each BPF program can also be defined to use a set of maps associated with the program at load time, and 'map_by_idx(imm)' means to get the map with the given index in the set associated with the BPF program containing the instruction.

1.4.4.2 Platform Variables

Platform variables are memory regions, identified by integer ids, exposed by the runtime and accessible by BPF programs on some platforms. The 'var_addr(imm)' operation means to get the address of the memory region identified by the given id.

1.4.5 Legacy BPF Packet access instructions

BPF previously introduced special instructions for access to packet data that were carried over from classic BPF. These instructions used an instruction class of `LD`, a size modifier of `W`, `H`, or `B`, and a mode modifier of `ABS` or `IND`. The 'dst_reg' and 'offset' fields were set to zero, and 'src_reg' was set to zero for `ABS`. However, these instructions are deprecated and SHOULD no longer be used. All legacy packet access instructions belong to the "packet" conformance group.