

Contents

| | |
|--|----------|
| 1 BPF Instruction Set Specification, v1.0 | 1 |
| 1.1 Documentation conventions | 1 |
| 1.1.1 Types | 1 |
| 1.1.2 Functions | 2 |
| 1.1.3 Definitions | 2 |
| 1.1.4 Conformance groups | 2 |
| 1.2 Instruction encoding | 3 |
| 1.2.1 Instruction classes | 4 |
| 1.3 Arithmetic and jump instructions | 4 |
| 1.3.1 Arithmetic instructions | 5 |
| 1.3.2 Byte swap instructions | 6 |
| 1.3.3 Jump instructions | 7 |
| 1.3.3.1 Helper functions | 9 |
| 1.3.3.2 Program-local functions | 9 |
| 1.4 Load and store instructions | 9 |
| 1.4.1 Regular load and store operations | 10 |
| 1.4.2 Sign-extension load operations | 10 |
| 1.4.3 Atomic operations | 10 |
| 1.4.4 64-bit immediate instructions | 11 |
| 1.4.4.1 Maps | 12 |
| 1.4.4.2 Platform Variables | 12 |
| 1.4.5 Legacy BPF Packet access instructions | 12 |

1 BPF Instruction Set Specification, v1.0

This document specifies version 1.0 of the BPF instruction set.

1.1 Documentation conventions

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 |
|----------|----------|
| <i>u</i> | unsigned |
| <i>s</i> | signed |

Meaning of bit-width notation.

| <i>N</i> | 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 types 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 "basic" 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 as such 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., "basic") 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.

The "basic" conformance group includes all instructions defined in this specification unless otherwise noted.

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-bit immediate (i.e., constant) value after the basic instruction for a total of 128 bits.

The fields conforming an encoded basic instruction are stored in the following order:

```
opcode:8 src_reg:4 dst_reg:4 offset:16 imm:32 // In little-endian BPF.
opcode:8 dst_reg:4 src_reg:4 offset:16 imm:32 // In big-endian BPF.
```

imm

signed integer immediate value

offset

signed integer offset used with pointer arithmetic

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)

opcode

operation to perform

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

For example:

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

As discussed below in [64-bit immediate instructions](#), a 64-bit immediate instruction uses a 64-bit immediate value that is constructed as follows. The 64 bits following the basic instruction contain a pseudo instruction using the same format but with opcode, dst_reg, src_reg, and offset all set to zero, and imm containing the high 32 bits of the immediate value.

This is depicted in the following figure:

basic_instruction

```
|-----|
```

```

opcode:8 regs:8 offset:16 imm:32 unused:32 imm:32
                        |-----|
                        pseudo instruction

```

Thus the 64-bit immediate value is constructed as follows:

$$\text{imm64} = (\text{next_imm} \ll 32) \mid \text{imm}$$

where 'next_imm' refers to the imm value of the pseudo instruction following the basic instruction. The unused bytes in the pseudo instruction are reserved and shall be cleared to zero.

1.2.1 Instruction classes

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

| class | value | description | reference |
|-----------|-------|---------------------------------|--|
| BPF_LD | 0x00 | non-standard load operations | Load and store instructions |
| BPF_LD X | 0x01 | load into register operations | Load and store instructions |
| BPF_ST | 0x02 | store from immediate operations | Load and store instructions |
| BPF_ST X | 0x03 | store from register operations | Load and store instructions |
| BPF_ALU | 0x04 | 32-bit arithmetic operations | Arithmetic and jump instructions |
| BPF_JMP | 0x05 | 64-bit jump operations | Arithmetic and jump instructions |
| BPF_JMP32 | 0x06 | 32-bit jump operations | Arithmetic and jump instructions |
| BPF_ALU64 | 0x07 | 64-bit arithmetic operations | Arithmetic and jump instructions |

1.3 Arithmetic and jump instructions

For arithmetic and jump instructions (BPF_ALU, BPF_ALU64, BPF_JMP and BPF_JMP32), the 8-bit 'opcode' field is divided into three parts:

| 4 bits (MSB) | 1 bit | 3 bits (LSB) |
|--------------|--------|-------------------|
| code | source | instruction class |

code

the operation code, whose meaning varies by instruction class

source

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

| source | value | description |
|--------|-------|--|
| BPF_K | 0x00 | use 32-bit 'imm' value as source operand |
| BPF_X | 0x08 | use 'src_reg' register value as source operand |

instruction class

the instruction class (see [Instruction classes](#))

1.3.1 Arithmetic instructions

BPF_ALU uses 32-bit wide operands while BPF_ALU64 uses 64-bit wide operands for otherwise identical operations. The 'code' field encodes the operation as below, where 'src' and 'dst' refer to the values of the source and destination registers, respectively.

| code | value | offset | description |
|-----------|-------|---------|---|
| BPF_ADD | 0x00 | 0 | dst += src |
| BPF_SUB | 0x10 | 0 | dst -= src |
| BPF_MUL | 0x20 | 0 | dst *= src |
| BPF_DIV | 0x30 | 0 | dst = (src != 0) ? (dst / src) : 0 |
| BPF_SDIV | 0x30 | 1 | dst = (src != 0) ? (dst s/ src) : 0 |
| BPF_OR | 0x40 | 0 | dst = src |
| BPF_AND | 0x50 | 0 | dst &= src |
| BPF_LSH | 0x60 | 0 | dst <<= (src & mask) |
| BPF_RSH | 0x70 | 0 | dst >>= (src & mask) |
| BPF_NEG | 0x80 | 0 | dst = -dst |
| BPF_MOD | 0x90 | 0 | dst = (src != 0) ? (dst % src) : dst |
| BPF_SMOD | 0x90 | 1 | dst = (src != 0) ? (dst s% src) : dst |
| BPF_XOR | 0xa0 | 0 | dst ^= src |
| BPF_MOV | 0xb0 | 0 | dst = src |
| BPF_MOVSX | 0xb0 | 8/16/32 | dst = (s8,s16,s32)src |
| BPF_ARSH | 0xc0 | 0 | :term:`sign extending<Sign Extend>` dst >>= (src & mask) |
| BPF_END | 0xd0 | 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 BPF_ALU64 the value of the destination register is unchanged whereas for BPF_ALU the upper 32 bits of the destination register are zeroed.

BPF_ADD | BPF_X | BPF_ALU means:

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

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

BPF_ADD | BPF_X | BPF_ALU64 means:

```
dst = dst + src
```

BPF_XOR | BPF_K | BPF_ALU means:

```
dst = (u32) dst ^ (u32) imm32
```

BPF_XOR | BPF_K | BPF_ALU64 means:

```
dst = dst ^ imm32
```

Note that most instructions have instruction offset of 0. Only three instructions (BPF_SDIV, BPF_SMOD, BPF_MOVSX) have a non-zero offset.

The division and modulo operations support both unsigned and signed flavors.

For unsigned operations (BPF_DIV and BPF_MOD), for BPF_ALU, 'imm' is interpreted as a 32-bit unsigned value. For BPF_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 (BPF_SDIV and BPF_SMOD), for BPF_ALU, 'imm' is interpreted as a 32-bit signed value. For BPF_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 use truncated division (where $-13 \% 3 == -1$) as implemented in C, Go, etc.:

$$a \% n = a - n * \text{trunc}(a / n)$$

The BPF_MOVSX instruction does a move operation with sign extension. BPF_ALU | BPF_MOVSX **:term:sign extends<Sign Extend>** 8-bit and 16-bit operands into 32 bit operands, and zeroes the remaining upper 32 bits. BPF_ALU64 | BPF_MOVSX **:term:sign extends<Sign Extend>** 8-bit, 16-bit, and 32-bit operands into 64 bit operands. Unlike other arithmetic instructions, BPF_MOVSX is only defined for register source operands (BPF_X).

The BPF_NEG instruction is only defined when the source bit is clear (BPF_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 BPF_ALU and BPF_ALU64 and a 4-bit 'code' field of BPF_END.

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

For BPF_ALU, the 1-bit source operand field in the opcode is used to select what byte order the operation converts from or to. For BPF_ALU64, the 1-bit source operand field in the opcode is reserved and must be set to 0.

| class | source | value | description |
|-------|--------|-------|-------------|
|-------|--------|-------|-------------|

| | | | |
|-----------|-----------|------|---|
| BPF_ALU | BPF_TO_LE | 0x00 | convert between host byte order and little endian |
| BPF_ALU | BPF_TO_BE | 0x08 | convert between host byte order and big endian |
| BPF_ALU64 | Reserved | 0x00 | do byte swap unconditionally |

The 'imm' field encodes the width of the swap operations. The following widths are supported: 16, 32 and 64.

Examples:

BPF_ALU | BPF_TO_LE | BPF_END with imm = 16/32/64 means:

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

BPF_ALU | BPF_TO_BE | BPF_END with imm = 16/32/64 means:

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

BPF_ALU64 | BPF_TO_LE | BPF_END with imm = 16/32/64 means:

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

1.3.3 Jump instructions

BPF_JMP32 uses 32-bit wide operands while BPF_JMP uses 64-bit wide operands for otherwise identical operations. The 'code' field encodes the operation as below:

| code | value | src | description | notes |
|---------|-------|------|----------------------------|------------------------|
| BPF_JA | 0x00 | 0x00 | PC += offset | BPF_JMP BPF_K only |
| BPF_JA | 0x00 | 0x00 | PC += imm | BPF_JMP32 BPF_K only |
| BPF_JEQ | 0x01 | any | PC += offset if dst == src | |
| BPF_JGT | 0x02 | any | PC += offset if dst > src | unsigned |
| BPF_JGE | 0x03 | any | PC += offset if dst >= src | unsigned |

| | | | | |
|----------|-----|-----|---------------------------------|---|
| BPF_JSET | 0x4 | any | PC += offset if dst & src | |
| BPF_JNE | 0x5 | any | PC += offset if dst != src | |
| BPF_JSGT | 0x6 | any | PC += offset if dst > src | signed |
| BPF_JSGE | 0x7 | any | PC += offset if dst >= src | signed |
| BPF_CALL | 0x8 | 0x0 | call helper function by address | BPF_JMP BPF_K only, see Helper functions |
| BPF_CALL | 0x8 | 0x1 | call PC += imm | BPF_JMP BPF_K only, see Program-local functions |
| BPF_CALL | 0x8 | 0x2 | call helper function by BTF ID | BPF_JMP BPF_K only, see Helper functions |
| BPF_EXIT | 0x9 | 0x0 | return | BPF_JMP BPF_K only |
| BPF_JLT | 0xa | any | PC += offset if dst < src | unsigned |
| BPF_JLE | 0xb | any | PC += offset if dst <= src | unsigned |
| BPF_JSLT | 0xc | any | PC += offset if dst < src | signed |
| BPF_JSLE | 0xd | any | PC += offset if dst <= src | signed |

The BPF program needs to store the return value into register R0 before doing a BPF_EXIT.

Example:

BPF_JSGE | BPF_X | BPF_JMP32 (0x7e) means:

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

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

BPF_JA | BPF_K | BPF_JMP32 (0x06) means:

```
goto1 +imm
```

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

Note that there are two flavors of `BPF_JA` instructions. The `BPF_JMP` class permits a 16-bit jump offset specified by the 'offset' field, whereas the `BPF_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.

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 an address encoded in the imm field. The available helper functions may differ for each program type, but address values 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.

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 call instruction, similar to `BPF_JA`. The offset is encoded in the imm field of the call instruction. A `BPF_EXIT` within the program-local function will return to the caller.

1.4 Load and store instructions

For load and store instructions (`BPF_LD`, `BPF_LDX`, `BPF_ST`, and `BPF_STX`), the 8-bit 'opcode' field is divided as:

| 3 bits (MSB) | 2 bits | 3 bits (LSB) |
|--------------|--------|-------------------|
| mode | size | instruction class |

The mode modifier is one of:

| mode modifier | value | description | reference |
|-------------------------|-------|-------------------------------------|---|
| <code>BPF_IMM</code> | 0x00 | 64-bit immediate instructions | 64-bit immediate instructions |
| <code>BPF_ABS</code> | 0x20 | legacy BPF packet access (absolute) | Legacy BPF Packet access instructions |
| <code>BPF_IND</code> | 0x40 | legacy BPF packet access (indirect) | Legacy BPF Packet access instructions |
| <code>BPF_MEM</code> | 0x60 | regular load and store operations | Regular load and store operations |
| <code>BPF_MEMSX</code> | 0x80 | sign-extension load operations | Sign-extension load operations |
| <code>BPF_ATOMIC</code> | 0xc0 | atomic operations | Atomic operations |

The size modifier is one of:

| size modifier | value | description |
|---------------------|-------|-----------------------|
| <code>BPF_W</code> | 0x00 | word (4 bytes) |
| <code>BPF_H</code> | 0x08 | half word (2 bytes) |
| <code>BPF_B</code> | 0x10 | byte |
| <code>BPF_DW</code> | 0x18 | double word (8 bytes) |

1.4.1 Regular load and store operations

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

`BPF_MEM` | `<size>` | `BPF_STX` means:

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

`BPF_MEM` | `<size>` | `BPF_ST` means:

```
*(size *) (dst + offset) = imm32
```

`BPF_MEM` | `<size>` | `BPF_LDX` means:

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

Where size is one of: `BPF_B`, `BPF_H`, `BPF_W`, or `BPF_DW` and 'unsigned size' is one of `u8`, `u16`, `u32` or `u64`.

1.4.2 Sign-extension load operations

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

`BPF_MEMSX` | `<size>` | `BPF_LDX` means:

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

Where size is one of: `BPF_B`, `BPF_H` or `BPF_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 `BPF_ATOMIC` mode modifier as follows:

- `BPF_ATOMIC` | `BPF_W` | `BPF_STX` for 32-bit operations
- `BPF_ATOMIC` | `BPF_DW` | `BPF_STX` for 64-bit operations
- 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:

| imm | value | description |
|----------------------|-------------------|-------------|
| <code>BPF_ADD</code> | <code>0x00</code> | atomic add |
| <code>BPF_OR</code> | <code>0x40</code> | atomic or |
| <code>BPF_AND</code> | <code>0x50</code> | atomic and |
| <code>BPF_XOR</code> | <code>0xa0</code> | atomic xor |

`BPF_ATOMIC` | `BPF_W` | `BPF_STX` with 'imm' = `BPF_ADD` means:

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

`BPF_ATOMIC` | `BPF_DW` | `BPF_STX` with 'imm' = `BPF_ADD` means:

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

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

| imm | value | description |
|-------------|------------------|-----------------------------|
| BPF_FETCH | 0x01 | modifier: return old value |
| BPF_XCHG | 0xe0 BPF_FETCH | atomic exchange |
| BPF_CMPXCHG | 0xf0 BPF_FETCH | atomic compare and exchange |

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

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

The `BPF_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 `BPF_IMM` 'mode' modifier use the wide instruction encoding defined in [Instruction encoding](#), and use the 'src' field of the basic instruction to hold an opcode subtype.

The following table defines a set of `BPF_IMM` | `BPF_DW` | `BPF_LD` instructions with opcode subtypes in the 'src' field, using new terms such as "map" defined further below:

| opcode construction | opcode | src | pseudocode | imm type | dst type |
|---------------------------|--------|-----|---|-------------|--------------|
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x0 | dst = imm64 | integer | integer |
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x1 | dst = map_by_fd(imm) | map fd | map |
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x2 | dst = map_val(map_by_fd(imm)) + next_imm | map fd | data pointer |
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x3 | dst = var_addr(imm) | variable id | data pointer |
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x4 | dst = code_addr(imm) | integer | code pointer |
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x5 | dst = map_by_idx(imm) | map index | map |
| BPF_IMM BPF_DW BPF_LD | 0x18 | 0x6 | dst = map_val(map_by_idx(imm)) + next_imm | map index | data pointer |

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. However, these instructions are deprecated and should no longer be used. All legacy packet access instructions belong to the "legacy" conformance group instead of the "basic" conformance group.