

# Contents

|   |          |
|---|----------|
| <b>1 eBPF Instruction Set Specification, v1.0</b> | <b>1</b> |
| 1.1 Documentation conventions                     | 1        |
| 1.2 Registers and calling convention              | 1        |
| 1.3 Instruction encoding                          | 2        |
| 1.3.1 Instruction classes                         | 2        |
| 1.4 Arithmetic and jump instructions              | 3        |
| 1.4.1 Arithmetic instructions                     | 3        |
| 1.4.1.1 Byte swap instructions                    | 5        |
| 1.4.2 Jump instructions                           | 5        |
| 1.4.2.1 Helper functions                          | 6        |
| 1.4.2.2 Runtime functions                         | 7        |
| 1.4.2.3 eBPF functions                            | 7        |
| 1.5 Load and store instructions                   | 7        |
| 1.5.1 Regular load and store operations           | 8        |
| 1.5.2 Atomic operations                           | 8        |
| 1.5.3 64-bit immediate instructions               | 9        |
| 1.5.3.1 Map objects                               | 10       |
| 1.5.3.2 Variables                                 | 10       |
| 1.5.4 Legacy BPF Packet access instructions       | 10       |
| 1.6 Appendix                                      | 10       |

## 1 eBPF Instruction Set Specification, v1.0

This document specifies version 1.0 of the eBPF instruction set.

The eBPF instruction set consists of eleven 64 bit registers, a program counter, and at least 512 bytes of stack space.

### 1.1 Documentation conventions

For brevity, this document uses the type notion "u64", "u32", etc. to mean an unsigned integer whose width is the specified number of bits, and "s32", etc. to mean a signed integer of the specified number of bits.

### 1.2 Registers and calling convention

eBPF has 10 general purpose registers and a read-only frame pointer register, all of which are 64-bits wide.

The eBPF calling convention is defined as:

- R0: return value from function calls, and exit value for eBPF programs
- R1 - R5: arguments for function calls
- R6 - R9: callee saved registers that function calls will preserve
- R10: read-only frame pointer to access stack

Registers R0 - R5 are caller-saved registers, meaning the BPF program needs to either spill them to the BPF stack or move them to callee saved registers if these arguments are to be reused across multiple function calls. Spilling means that the value in the register is moved to the BPF stack. The reverse operation of moving the variable from the BPF stack to the register is called filling. The reason for spilling/filling is due to the limited number of registers.

Upon entering execution of an eBPF program, registers R1 - R5 initially can contain the input arguments for the program (similar to the argc/argv pair for a typical C program). The actual number of registers used, and their meaning, is defined by the program type; for example, a networking program might have an argument that includes network packet data and/or metadata.

## 1.3 Instruction encoding

An eBPF program is a sequence of instructions.

eBPF 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 basic instruction encoding is as follows, where MSB and LSB mean the most significant bits and least significant bits, respectively:

| 32 bits (MSB) | 16 bits | 4 bits | 4 bits | 8 bits (LSB) |
|---------------|---------|--------|--------|--------------|
| imm           | offset  | src    | dst    | opcode       |

### imm

signed integer immediate value

### offset

signed integer offset used with pointer arithmetic

### src

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

### dst

destination register number (0-10)

### opcode

operation to perform

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

As discussed below in [64-bit immediate instructions](#), some instructions use 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, src, and offset all set to zero, and imm containing the high 32 bits of the immediate value.

| 64 bits (MSB)     | 64 bits (LSB)      |
|-------------------|--------------------|
| basic instruction | pseudo instruction |

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

$$\text{imm64} = \text{imm} + (\text{next\_imm} \ll 32)$$

where 'next\_imm' refers to the imm value of the pseudo instruction following the basic instruction.

In the remainder of this document 'src' and 'dst' refer to the values of the source and destination registers, respectively, rather than the register number.

### 1.3.1 Instruction classes

The encoding of the 'opcode' field varies and can be determined from the three least significant bits (LSB) of the 'opcode' field which holds the "instruction class", as follows:

| 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 |

## 1.4 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' register value as source operand |

### instruction class

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

### 1.4.1 Arithmetic instructions

Instruction class BPF\_ALU uses 32-bit wide operands (zeroing the upper 32 bits of the destination register) while BPF\_ALU64 uses 64-bit wide operands for otherwise identical operations.

The 4-bit 'code' field encodes the operation as follows:

| code    | value | description |
|---------|-------|-------------|
| BPF_ADD | 0x00  | dst += src  |
| BPF_SUB | 0x10  | dst -= src  |
| BPF_MUL | 0x20  | dst *= src  |

|          |      |  |
|----------|------|--|
| BPF_DIV  | 0x30 | $\text{dst} = (\text{src} \neq 0) ? (\text{dst} / \text{src}) : 0$           |
| BPF_OR   | 0x40 | $\text{dst}  = \text{src}$   |
| BPF_AND  | 0x50 | $\text{dst} \&= \text{src}$  |
| BPF_LSH  | 0x60 | $\text{dst} \ll= \text{src}$   |
| BPF_RSH  | 0x70 | $\text{dst} \gg= \text{src}$   |
| BPF_NEG  | 0x80 | $\text{dst} = \sim \text{src}$   |
| BPF_MOD  | 0x90 | $\text{dst} = (\text{src} \neq 0) ? (\text{dst} \% \text{src}) : \text{dst}$ |
| BPF_XOR  | 0xa0 | $\text{dst} \wedge= \text{src}$  |
| BPF_MOV  | 0xb0 | $\text{dst} = \text{src}$  |
| BPF_ARSH | 0xc0 | sign extending shift right   |
| BPF_END  | 0xd0 | byte swap operations (see <a href="#">Byte swap instructions</a> below)      |

where 'src' is the source operand value.

Underflow and overflow are allowed during arithmetic operations, meaning the 64-bit or 32-bit value will wrap. If eBPF program execution would result in division by zero, the destination register is instead set to zero. If execution would result in modulo by zero, the destination register is instead left unchanged.

Examples:

BPF\_ADD | BPF\_X | BPF\_ALU (0x0c) means:

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

where '(u32)' indicates truncation to 32 bits.

BPF\_ADD | BPF\_X | BPF\_ALU64 (0x0f) means:

```
dst = dst + src
```

BPF\_XOR | BPF\_K | BPF\_ALU (0xa4) means:

```
src = (u32) src ^ (u32) imm
```

BPF\_XOR | BPF\_K | BPF\_ALU64 (0xa7) means:

```
src = src ^ imm
```

Also note that the modulo operation often varies by language when the dividend or divisor are negative, where Python, Ruby, etc. differ from C, Go, Java, etc. This specification requires that modulo use truncated division (where  $-13 \% 3 == -1$ ) as implemented in C, Go, etc.:

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

#### 1.4.1.1 Byte swap instructions

The byte swap instructions use an instruction class of `BPF_ALU` 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.

Byte swap instructions use the 1-bit 'source' field in the 'opcode' field as follows. Instead of indicating the source operator, it is instead used to select what byte order the operation converts from or to:

| source    | value | description                                       |
|-----------|-------|---|
| BPF_TO_LE | 0x00  | convert between host byte order and little endian |
| BPF_TO_BE | 0x08  | convert between host byte order and big endian    |

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

| opcode construction           | opcode | imm | mnemonic | pseudocode         |
|-------------------------------|--------|-----|----------|--------------------|
| BPF_END   BPF_TO_LE   BPF_ALU | 0xd4   | 16  | le16 dst | dst = htole16(dst) |
| BPF_END   BPF_TO_LE   BPF_ALU | 0xd4   | 32  | le32 dst | dst = htole32(dst) |
| BPF_END   BPF_TO_LE   BPF_ALU | 0xd4   | 64  | le64 dst | dst = htole64(dst) |
| BPF_END   BPF_TO_BE   BPF_ALU | 0xdc   | 16  | be16 dst | dst = htobe16(dst) |
| BPF_END   BPF_TO_BE   BPF_ALU | 0xdc   | 32  | be32 dst | dst = htobe32(dst) |
| BPF_END   BPF_TO_BE   BPF_ALU | 0xdc   | 64  | be64 dst | dst = htobe64(dst) |

where

- mnemonic indicates a short form that might be displayed by some tools such as disassemblers
- 'htoleNN()' indicates converting a NN-bit value from host byte order to little-endian byte order
- 'htobeNN()' indicates converting a NN-bit value from host byte order to big-endian byte order

#### 1.4.2 Jump instructions

Instruction class `BPF_JMP32` uses 32-bit wide operands while `BPF_JMP` uses 64-bit wide operands for otherwise identical operations.

The 4-bit 'code' field encodes the operation as below, where PC is the program counter:

| code    | value | src | description                | notes        |
|---------|-------|-----|----------------------------|--------------|
| BPF_JA  | 0x0   | 0x0 | PC += offset               | BPF_JMP only |
| BPF_JEQ | 0x1   | any | PC += offset if dst == src |              |

|          |     |             |                            |                                       |
|----------|-----|-------------|----------------------------|---------------------------------------|
| BPF_JGT  | 0x2 | a<br>n<br>y | PC += offset if dst > src  | unsigned                              |
| BPF_JGE  | 0x3 | a<br>n<br>y | PC += offset if dst >= src | unsigned                              |
| BPF_JSET | 0x4 | a<br>n<br>y | PC += offset if dst & src  |                                       |
| BPF_JNE  | 0x5 | a<br>n<br>y | PC += offset if dst != src |                                       |
| BPF_JSGT | 0x6 | a<br>n<br>y | PC += offset if dst > src  | signed                                |
| BPF_JSGE | 0x7 | a<br>n<br>y | PC += offset if dst >= src | signed                                |
| BPF_CALL | 0x8 | 0<br>x<br>0 | call helper function imm   | see <a href="#">Helper functions</a>  |
| BPF_CALL | 0x8 | 0<br>x<br>1 | call PC += offset          | see <a href="#">eBPF functions</a>    |
| BPF_CALL | 0x8 | 0<br>x<br>2 | call runtime function imm  | see <a href="#">Runtime functions</a> |
| BPF_EXIT | 0x9 | 0<br>x<br>0 | return                     | BPF_JMP only                          |
| BPF_JLT  | 0xa | a<br>n<br>y | PC += offset if dst < src  | unsigned                              |
| BPF_JLE  | 0xb | a<br>n<br>y | PC += offset if dst <= src | unsigned                              |
| BPF_JSLT | 0xc | a<br>n<br>y | PC += offset if dst < src  | signed                                |
| BPF_JSLE | 0xd | a<br>n<br>y | PC += offset if dst <= src | signed                                |

#### 1.4.2.1 Helper functions

Helper functions are a concept whereby BPF programs can call into a set of function calls exposed by the eBPF runtime. Each helper function is identified by an integer used in a `BPF_CALL` instruction. The available helper functions may differ for each eBPF program type.

Conceptually, each helper function is implemented with a commonly shared function signature defined as:

u64 function(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)

In actuality, each helper function is defined as taking between 0 and 5 arguments, with the remaining registers being ignored. The definition of a helper function is responsible for specifying the type (e.g., integer, pointer, etc.) of the value returned, the number of arguments, and the type of each argument.

Note that `BPF_CALL` | `BPF_X` | `BPF_JMP` (0x8d), where the helper function integer would be read from a specified register, is reserved and currently not permitted.

#### 1.4.2.2 Runtime functions

Runtime functions are like helper functions except that they are not specific to eBPF programs. They use a different numbering space from helper functions, but otherwise the same considerations apply.

#### 1.4.2.3 eBPF functions

eBPF functions are functions exposed by the same eBPF program as the caller, and are referenced by offset from the call instruction, similar to `BPF_JA`. A `BPF_EXIT` within the eBPF function will return to the caller.

## 1.5 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 |

### mode

one of:

| mode modifier | value | description                         | reference   |
|---------------|-------|-------------------------------------|---|
| BPF_IMM       | 0x00  | 64-bit immediate instructions       | <a href="#">64-bit immediate instructions</a>         |
| BPF_ABS       | 0x20  | legacy BPF packet access (absolute) | <a href="#">Legacy BPF Packet access instructions</a> |
| BPF_IND       | 0x40  | legacy BPF packet access (indirect) | <a href="#">Legacy BPF Packet access instructions</a> |
| BPF_MEM       | 0x60  | regular load and store operations   | <a href="#">Regular load and store operations</a>     |
| BPF_ATOMIC    | 0xc0  | atomic operations                   | <a href="#">Atomic operations</a>                     |

### size

one of:

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

### instruction class

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

### 1.5.1 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_reg
```

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

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

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

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

where `size` is one of: `BPF_B`, `BPF_H`, `BPF_W`, or `BPF_DW`.

### 1.5.2 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 eBPF programs or means outside of this specification.

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

- `BPF_ATOMIC` | `BPF_W` | `BPF_STX` (0xc3) for 32-bit operations
- `BPF_ATOMIC` | `BPF_DW` | `BPF_STX` (0xdb) for 64-bit operations

Note that 8-bit (`BPF_B`) and 16-bit (`BPF_H`) wide atomic operations are not supported, nor is `BPF_ATOMIC` | `<size>` | `BPF_ST`.

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> | 0x00  | atomic add  |
| <code>BPF_OR</code>  | 0x40  | atomic or   |
| <code>BPF_AND</code> | 0x50  | atomic and  |
| <code>BPF_XOR</code> | 0xa0  | atomic xor  |

`BPF_ATOMIC` | `BPF_W` | `BPF_STX` (0xc3) with 'imm' = `BPF_ADD` means:

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

`BPF_ATOMIC` | `BPF_DW` | `BPF_STX` (0xdb) with 'imm' = `BPF_ADD` means:

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

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

| imm                    | value                         | description                |
|------------------------|-------------------------------|----------------------------|
| <code>BPF_FETCH</code> | 0x01                          | modifier: return old value |
| <code>BPF_XCHG</code>  | 0xe0   <code>BPF_FETCH</code> | 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`.

### 1.5.3 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 instructions are defined, and use additional concepts defined below:

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

where

- `map_by_fd(fd)` means to convert a 32-bit POSIX file descriptor into an address of a map object (see [Map objects](#))
- `map_by_index(index)` means to convert a 32-bit index into an address of a map object
- `mva(map)` gets the address of the first value in a given map object
- `variable_addr(id)` gets the address of a variable (see [Variables](#)) with a given id
- `code_addr(offset)` gets the address of the instruction at a specified relative offset in units of 64-bit blocks
- the 'imm type' can be used by disassemblers for display
- the 'dst type' can be used for verification and JIT compilation purposes

### 1.5.3.1 Map objects

Maps are shared memory regions accessible by eBPF programs on some platforms, where we use the term "map object" to refer to an object containing the data and metadata (e.g., size) about the memory region. 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 'mva(map)' is currently only defined for maps that do have a single contiguous memory region. Support for maps is optional.

Each map object can have a POSIX file descriptor (fd) if supported by the platform, where 'map\_by\_fd(fd)' means to get the map with the specified file descriptor. Each eBPF program can also be defined to use a set of maps associated with the program at load time, and 'map\_by\_index(index)' means to get the map with the given index in the set associated with the eBPF program containing the instruction.

### 1.5.3.2 Variables

Variables are memory regions, identified by integer ids, accessible by eBPF programs on some platforms. The 'variable\_addr(id)' operation means to get the address of the memory region identified by the given id. Support for such variables is optional.

### 1.5.4 Legacy BPF Packet access instructions

eBPF 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.

## 1.6 Appendix

For reference, the following table lists opcodes in order by value.

| op<br>co<br>de | s<br>r<br>c | i<br>m<br>m      | description                  | reference                                     |
|----------------|-------------|------------------|------------------------------|---|
| 0x<br>00       | 0<br>x<br>0 | a<br>n<br>y      | (additional immediate value) | <a href="#">64-bit immediate instructions</a> |
| 0x<br>04       | 0<br>x<br>0 | a<br>n<br>y      | dst = (u32)(dst + imm)       | <a href="#">Arithmetic instructions</a>       |
| 0x<br>05       | 0<br>x<br>0 | 0<br>x<br>0<br>0 | goto +offset                 | <a href="#">Jump instructions</a>             |
| 0x<br>07       | 0<br>x<br>0 | a<br>n<br>y      | dst += imm                   | <a href="#">Arithmetic instructions</a>       |
| 0x<br>0c       | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = (u32)(dst + src)       | <a href="#">Arithmetic instructions</a>       |
| 0x<br>0f       | a<br>n<br>y | 0<br>x<br>0<br>0 | dst += src                   | <a href="#">Arithmetic instructions</a>       |
| 0x<br>14       | 0<br>x<br>0 | a<br>n<br>y      | dst = (u32)(dst - imm)       | <a href="#">Arithmetic instructions</a>       |

|      |             |                  |                                       |                                       |
|------|-------------|------------------|---------------------------------------|---------------------------------------|
| 0x15 | 0<br>x<br>0 | a<br>n<br>y      | if dst == imm goto +offset            | Jump instructions                     |
| 0x16 | 0<br>x<br>0 | a<br>n<br>y      | if (u32)dst == imm goto +offset       | Jump instructions                     |
| 0x17 | 0<br>x<br>0 | a<br>n<br>y      | dst -= imm                            | Arithmetic instructions               |
| 0x18 | 0<br>x<br>0 | a<br>n<br>y      | dst = imm64                           | 64-bit immediate instructions         |
| 0x18 | 0<br>x<br>1 | a<br>n<br>y      | dst = map_by_fd(imm)                  | 64-bit immediate instructions         |
| 0x18 | 0<br>x<br>2 | a<br>n<br>y      | dst = mva(map_by_fd(imm)) + next_imm  | 64-bit immediate instructions         |
| 0x18 | 0<br>x<br>3 | a<br>n<br>y      | dst = variable_addr(imm)              | 64-bit immediate instructions         |
| 0x18 | 0<br>x<br>4 | a<br>n<br>y      | dst = code_addr(imm)                  | 64-bit immediate instructions         |
| 0x18 | 0<br>x<br>5 | a<br>n<br>y      | dst = map_by_idx(imm)                 | 64-bit immediate instructions         |
| 0x18 | 0<br>x<br>6 | a<br>n<br>y      | dst = mva(map_by_idx(imm)) + next_imm | 64-bit immediate instructions         |
| 0x1c | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = (u32)(dst - src)                | Arithmetic instructions               |
| 0x1d | a<br>n<br>y | 0<br>x<br>0<br>0 | if dst == src goto +offset            | Jump instructions                     |
| 0x1e | a<br>n<br>y | 0<br>x<br>0<br>0 | if (u32)dst == (u32)src goto +offset  | Jump instructions                     |
| 0x1f | a<br>n<br>y | 0<br>x<br>0<br>0 | dst -= src                            | Arithmetic instructions               |
| 0x20 | a<br>n<br>y | a<br>n<br>y      | (deprecated, implementation-specific) | Legacy BPF Packet access instructions |

|          |             |                  |  |                                       |
|----------|-------------|------------------|--|---------------------------------------|
| 0x<br>24 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{u32})(\text{dst} * \text{imm})$                             | Arithmetic instructions               |
| 0x<br>25 | 0<br>x<br>0 | a<br>n<br>y      | if $\text{dst} > \text{imm}$ goto +offset  | Jump instructions                     |
| 0x<br>26 | 0<br>x<br>0 | a<br>n<br>y      | if $(\text{u32})\text{dst} > \text{imm}$ goto +offset                            | Jump instructions                     |
| 0x<br>27 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} *= \text{imm}$   | Arithmetic instructions               |
| 0x<br>28 | a<br>n<br>y | a<br>n<br>y      | (deprecated, implementation-specific)  | Legacy BPF Packet access instructions |
| 0x<br>2c | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{u32})(\text{dst} * \text{src})$                             | Arithmetic instructions               |
| 0x<br>2d | a<br>n<br>y | 0<br>x<br>0<br>0 | if $\text{dst} > \text{src}$ goto +offset  | Jump instructions                     |
| 0x<br>2e | a<br>n<br>y | 0<br>x<br>0<br>0 | if $(\text{u32})\text{dst} > (\text{u32})\text{src}$ goto +offset                | Jump instructions                     |
| 0x<br>2f | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} *= \text{src}$   | Arithmetic instructions               |
| 0x<br>30 | a<br>n<br>y | a<br>n<br>y      | (deprecated, implementation-specific)  | Legacy BPF Packet access instructions |
| 0x<br>34 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{u32})((\text{imm} \neq 0) ? (\text{dst} / \text{imm}) : 0)$ | Arithmetic instructions               |
| 0x<br>35 | 0<br>x<br>0 | a<br>n<br>y      | if $\text{dst} \geq \text{imm}$ goto +offset                                     | Jump instructions                     |
| 0x<br>36 | 0<br>x<br>0 | a<br>n<br>y      | if $(\text{u32})\text{dst} \geq \text{imm}$ goto +offset                         | Jump instructions                     |
| 0x<br>37 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{imm} \neq 0) ? (\text{dst} / \text{imm}) : 0$               | Arithmetic instructions               |
| 0x<br>38 | a<br>n<br>y | a<br>n<br>y      | (deprecated, implementation-specific)  | Legacy BPF Packet access instructions |

|          |             |                  |  |                                       |
|----------|-------------|------------------|--|---------------------------------------|
| 0x<br>3c | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{u32})((\text{imm} \neq 0) ? (\text{dst} / \text{src}) : 0)$ | Arithmetic instructions               |
| 0x<br>3d | a<br>n<br>y | 0<br>x<br>0<br>0 | if $\text{dst} \geq \text{src}$ goto +offset                                     | Jump instructions                     |
| 0x<br>3e | a<br>n<br>y | 0<br>x<br>0<br>0 | if $(\text{u32})\text{dst} \geq (\text{u32})\text{src}$ goto +offset             | Jump instructions                     |
| 0x<br>3f | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{src} \neq 0) ? (\text{dst} / \text{src}) : 0$               | Arithmetic instructions               |
| 0x<br>40 | a<br>n<br>y | a<br>n<br>y      | (deprecated, implementation-specific)  | Legacy BPF Packet access instructions |
| 0x<br>44 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{u32})(\text{dst}   \text{imm})$                             | Arithmetic instructions               |
| 0x<br>45 | 0<br>x<br>0 | a<br>n<br>y      | if $\text{dst} \& \text{imm}$ goto +offset                                       | Jump instructions                     |
| 0x<br>46 | 0<br>x<br>0 | a<br>n<br>y      | if $(\text{u32})\text{dst} \& \text{imm}$ goto +offset                           | Jump instructions                     |
| 0x<br>47 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst}  = \text{imm}$   | Arithmetic instructions               |
| 0x<br>48 | a<br>n<br>y | a<br>n<br>y      | (deprecated, implementation-specific)  | Legacy BPF Packet access instructions |
| 0x<br>4c | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{u32})(\text{dst}   \text{src})$                             | Arithmetic instructions               |
| 0x<br>4d | a<br>n<br>y | 0<br>x<br>0<br>0 | if $\text{dst} \& \text{src}$ goto +offset                                       | Jump instructions                     |
| 0x<br>4e | a<br>n<br>y | 0<br>x<br>0<br>0 | if $(\text{u32})\text{dst} \& (\text{u32})\text{src}$ goto +offset               | Jump instructions                     |
| 0x<br>4f | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst}  = \text{src}$   | Arithmetic instructions               |

|      |     |      |  |                                       |
|------|-----|------|--|---------------------------------------|
| 0x50 | any | any  | (deprecated, implementation-specific)                                | Legacy BPF Packet access instructions |
| 0x54 | 0x0 | any  | $\text{dst} = (\text{u32})(\text{dst} \& \text{imm})$                | Arithmetic instructions               |
| 0x55 | 0x0 | any  | if $\text{dst} \neq \text{imm}$ goto +offset                         | Jump instructions                     |
| 0x56 | 0x0 | any  | if $(\text{u32})\text{dst} \neq \text{imm}$ goto +offset             | Jump instructions                     |
| 0x57 | 0x0 | any  | $\text{dst} \&= \text{imm}$  | Arithmetic instructions               |
| 0x58 | any | any  | (deprecated, implementation-specific)                                | Legacy BPF Packet access instructions |
| 0x5c | any | 0x00 | $\text{dst} = (\text{u32})(\text{dst} \& \text{src})$                | Arithmetic instructions               |
| 0x5d | any | 0x00 | if $\text{dst} \neq \text{src}$ goto +offset                         | Jump instructions                     |
| 0x5e | any | 0x00 | if $(\text{u32})\text{dst} \neq (\text{u32})\text{src}$ goto +offset | Jump instructions                     |
| 0x5f | any | 0x00 | $\text{dst} \&= \text{src}$  | Arithmetic instructions               |
| 0x61 | any | 0x00 | $\text{dst} = *(\text{u32} *) (\text{src} + \text{offset})$          | Load and store instructions           |
| 0x62 | 0x0 | any  | $*(\text{u32} *) (\text{dst} + \text{offset}) = \text{imm}$          | Load and store instructions           |
| 0x63 | any | 0x00 | $*(\text{u32} *) (\text{dst} + \text{offset}) = \text{src}$          | Load and store instructions           |
| 0x64 | 0x0 | any  | $\text{dst} = (\text{u32})(\text{dst} \ll \text{imm})$               | Arithmetic instructions               |
| 0x65 | 0x0 | any  | if $\text{dst} \geq \text{imm}$ goto +offset                         | Jump instructions                     |

|      |             |                  |                                      |                             |
|------|-------------|------------------|--------------------------------------|-----------------------------|
| 0x66 | 0<br>x<br>0 | a<br>n<br>y      | if (s32)dst s> (s32)imm goto +offset | Jump instructions           |
| 0x67 | 0<br>x<br>0 | a<br>n<br>y      | dst <= imm                           | Arithmetic instructions     |
| 0x69 | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = *(u16 *)(src + offset)         | Load and store instructions |
| 0x6a | 0<br>x<br>0 | a<br>n<br>y      | *(u16 *)(dst + offset) = imm         | Load and store instructions |
| 0x6b | a<br>n<br>y | 0<br>x<br>0<br>0 | *(u16 *)(dst + offset) = src         | Load and store instructions |
| 0x6c | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = (u32)(dst << src)              | Arithmetic instructions     |
| 0x6d | a<br>n<br>y | 0<br>x<br>0<br>0 | if dst s> src goto +offset           | Jump instructions           |
| 0x6e | a<br>n<br>y | 0<br>x<br>0<br>0 | if (s32)dst s> (s32)src goto +offset | Jump instructions           |
| 0x6f | a<br>n<br>y | 0<br>x<br>0<br>0 | dst <= src                           | Arithmetic instructions     |
| 0x71 | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = *(u8 *)(src + offset)          | Load and store instructions |
| 0x72 | 0<br>x<br>0 | a<br>n<br>y      | *(u8 *)(dst + offset) = imm          | Load and store instructions |
| 0x73 | a<br>n<br>y | 0<br>x<br>0<br>0 | *(u8 *)(dst + offset) = src          | Load and store instructions |
| 0x74 | 0<br>x<br>0 | a<br>n<br>y      | dst = (u32)(dst >> imm)              | Arithmetic instructions     |
| 0x75 | 0<br>x<br>0 | a<br>n<br>y      | if dst s>= imm goto +offset          | Jump instructions           |

|      |                  |                  |                                       |                             |
|------|------------------|------------------|---------------------------------------|-----------------------------|
| 0x76 | 0<br>x<br>0      | a<br>n<br>y      | if (s32)dst s>= (s32)imm goto +offset | Jump instructions           |
| 0x77 | 0<br>x<br>0      | a<br>n<br>y      | dst >>= imm                           | Arithmetic instructions     |
| 0x79 | a<br>n<br>y      | 0<br>x<br>0<br>0 | dst = *(u64 *)(src + offset)          | Load and store instructions |
| 0x7a | 0<br>x<br>0      | a<br>n<br>y      | *(u64 *)(dst + offset) = imm          | Load and store instructions |
| 0x7b | a<br>n<br>y      | 0<br>x<br>0<br>0 | *(u64 *)(dst + offset) = src          | Load and store instructions |
| 0x7c | a<br>n<br>y      | 0<br>x<br>0<br>0 | dst = (u32)(dst >> src)               | Arithmetic instructions     |
| 0x7d | a<br>n<br>y      | 0<br>x<br>0<br>0 | if dst s>= src goto +offset           | Jump instructions           |
| 0x7e | a<br>n<br>y      | 0<br>x<br>0<br>0 | if (s32)dst s>= (s32)src goto +offset | Jump instructions           |
| 0x7f | a<br>n<br>y      | 0<br>x<br>0<br>0 | dst >>= src                           | Arithmetic instructions     |
| 0x84 | 0<br>x<br>0<br>0 | 0<br>x<br>0<br>0 | dst = (u32)-dst                       | Arithmetic instructions     |
| 0x85 | 0<br>x<br>0      | a<br>n<br>y      | call helper function imm              | Helper functions            |
| 0x85 | 0<br>x<br>1      | a<br>n<br>y      | call PC += offset                     | eBPF functions              |
| 0x85 | 0<br>x<br>2      | a<br>n<br>y      | call runtime function imm             | Runtime functions           |
| 0x87 | 0<br>x<br>0<br>0 | 0<br>x<br>0<br>0 | dst = -dst                            | Arithmetic instructions     |



|      |             |                  |  |                         |
|------|-------------|------------------|--|-------------------------|
| 0x94 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{u32})((\text{imm} \neq 0) ? (\text{dst} \% \text{imm}) : \text{dst})$ | Arithmetic instructions |
| 0x95 | 0<br>x<br>0 | 0<br>x<br>0<br>0 | return   | Jump instructions       |
| 0x97 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{imm} \neq 0) ? (\text{dst} \% \text{imm}) : \text{dst}$               | Arithmetic instructions |
| 0x9c | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{u32})((\text{src} \neq 0) ? (\text{dst} \% \text{src}) : \text{dst})$ | Arithmetic instructions |
| 0x9f | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{src} \neq 0) ? (\text{dst} \% \text{src}) : \text{dst}$               | Arithmetic instructions |
| 0xa4 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{u32})(\text{dst} \wedge \text{imm})$                                  | Arithmetic instructions |
| 0xa5 | 0<br>x<br>0 | a<br>n<br>y      | if $\text{dst} < \text{imm}$ goto +offset  | Jump instructions       |
| 0xa6 | 0<br>x<br>0 | a<br>n<br>y      | if $(\text{u32})\text{dst} < \text{imm}$ goto +offset                                      | Jump instructions       |
| 0xa7 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} \wedge= \text{imm}$  | Arithmetic instructions |
| 0xac | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} = (\text{u32})(\text{dst} \wedge \text{src})$                                  | Arithmetic instructions |
| 0xad | a<br>n<br>y | 0<br>x<br>0<br>0 | if $\text{dst} < \text{src}$ goto +offset  | Jump instructions       |
| 0xae | a<br>n<br>y | 0<br>x<br>0<br>0 | if $(\text{u32})\text{dst} < (\text{u32})\text{src}$ goto +offset                          | Jump instructions       |
| 0xaf | a<br>n<br>y | 0<br>x<br>0<br>0 | $\text{dst} \wedge= \text{src}$  | Arithmetic instructions |
| 0xb4 | 0<br>x<br>0 | a<br>n<br>y      | $\text{dst} = (\text{u32}) \text{imm}$   | Arithmetic instructions |

|      |             |                  |  |                         |
|------|-------------|------------------|--|-------------------------|
| 0xb5 | 0<br>x<br>0 | a<br>n<br>y      | if dst <= imm goto +offset   | Jump instructions       |
| 0xa6 | 0<br>x<br>0 | a<br>n<br>y      | if (u32)dst <= imm goto +offset  | Jump instructions       |
| 0xb7 | 0<br>x<br>0 | a<br>n<br>y      | dst = imm  | Arithmetic instructions |
| 0xbc | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = (u32) src  | Arithmetic instructions |
| 0xbd | a<br>n<br>y | 0<br>x<br>0<br>0 | if dst <= src goto +offset   | Jump instructions       |
| 0xbe | a<br>n<br>y | 0<br>x<br>0<br>0 | if (u32)dst <= (u32)src goto +offset   | Jump instructions       |
| 0xbf | a<br>n<br>y | 0<br>x<br>0<br>0 | dst = src  | Arithmetic instructions |
| 0xc3 | a<br>n<br>y | 0<br>x<br>0<br>0 | lock *(u32 *) (dst + offset) += src  | Atomic operations       |
| 0xc3 | a<br>n<br>y | 0<br>x<br>0<br>1 | lock:<br><div>*(u32 *) (dst + offset) += src<br/>src = *(u32 *) (dst + offset)</div>     | Atomic operations       |
| 0xc3 | a<br>n<br>y | 0<br>x<br>4<br>0 | *(u32 *) (dst + offset)  = src   | Atomic operations       |
| 0xc3 | a<br>n<br>y | 0<br>x<br>4<br>1 | lock:<br><div>*(u32 *) (dst + offset)  = src<br/>src = *(u32 *) (dst + offset)</div>     | Atomic operations       |
| 0xc3 | a<br>n<br>y | 0<br>x<br>5<br>0 | *(u32 *) (dst + offset) &= src   | Atomic operations       |
| 0xc3 | a<br>n<br>y | 0<br>x<br>5<br>1 | lock:<br><div>*(u32 *) (dst + offset) &amp;= src<br/>src = *(u32 *) (dst + offset)</div> | Atomic operations       |

|      |     |      |   |                         |
|------|-----|------|---|-------------------------|
| 0xc3 | any | 0x0  | $*(u32*)(dst + offset) \wedge= src$   | Atomic operations       |
| 0xc3 | any | 0x1  | lock:<br><div><math>*(u32*)(dst + offset) \wedge= src</math><br/><math>src = *(u32*)(dst + offset)</math></div>   | Atomic operations       |
| 0xc3 | any | 0x1e | lock:<br><div><math>temp = *(u32*)(dst + offset)</math><br/><math>*(u32*)(dst + offset) = src</math><br/><math>src = temp</math></div>  | Atomic operations       |
| 0xc3 | any | 0xf1 | lock:<br><div><math>temp = *(u32*)(dst + offset)</math><br/>if <math>*(u32)(dst + offset) == R0</math><br/>    <math>*(u32)(dst + offset) = src</math><br/><math>R0 = temp</math></div> | Atomic operations       |
| 0xc4 | 0x0 | any  | $dst = (u32)(dst \gg imm)$  | Arithmetic instructions |
| 0xc5 | 0x0 | any  | if $dst \leq imm$ goto +offset  | Jump instructions       |
| 0xc6 | 0x0 | any  | if $(s32)dst \leq (s32)imm$ goto +offset  | Jump instructions       |
| 0xc7 | 0x0 | any  | $dst \gg= imm$  | Arithmetic instructions |
| 0xc  | any | 0x00 | $dst = (u32)(dst \gg src)$  | Arithmetic instructions |
| 0xcd | any | 0x00 | if $dst \leq src$ goto +offset  | Jump instructions       |
| 0xce | any | 0x00 | if $(s32)dst \leq (s32)src$ goto +offset  | Jump instructions       |
| 0xcf | any | 0x00 | $dst \gg= src$  | Arithmetic instructions |

|          |             |                  |   |                        |
|----------|-------------|------------------|---|------------------------|
| 0x<br>d4 | 0<br>x<br>0 | 0<br>x<br>1<br>0 | dst = htole16(dst)  | Byte swap instructions |
| 0x<br>d4 | 0<br>x<br>0 | 0<br>x<br>2<br>0 | dst = htole32(dst)  | Byte swap instructions |
| 0x<br>d4 | 0<br>x<br>0 | 0<br>x<br>4<br>0 | dst = htole64(dst)  | Byte swap instructions |
| 0x<br>d5 | 0<br>x<br>0 | a<br>n<br>y      | if dst s<= imm goto +offset   | Jump instructions      |
| 0x<br>d6 | 0<br>x<br>0 | a<br>n<br>y      | if (s32)dst s<= (s32)imm goto +offset   | Jump instructions      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>0<br>0 | lock *(u64 *) (dst + offset) += src   | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>0<br>1 | lock:<br><div>*(u64 *) (dst + offset) += src<br/>src = *(u64 *) (dst + offset)</div>      | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>4<br>0 | *(u64 *) (dst + offset)  = src  | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>4<br>1 | lock:<br><div>*(u64 *) (dst + offset)  = src<br/>lock src = *(u64 *) (dst + offset)</div> | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>5<br>0 | *(u64 *) (dst + offset) &= src  | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>5<br>1 | lock:<br><div>*(u64 *) (dst + offset) &amp;= src<br/>src = *(u64 *) (dst + offset)</div>  | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>a<br>0 | *(u64 *) (dst + offset) ^= src  | Atomic operations      |

|          |             |                  |   |                        |
|----------|-------------|------------------|---|------------------------|
| 0x<br>db | a<br>n<br>y | 0<br>x<br>a<br>1 | lock:<br><br><pre>*(u64*)(dst + offset) ^= src<br/>src = *(u64*)(dst + offset)</pre>  | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>e<br>1 | lock:<br><br><pre>temp = *(u64*)(dst + offset)<br/>*(u64*)(dst + offset) = src<br/>src = temp</pre>                                       | Atomic operations      |
| 0x<br>db | a<br>n<br>y | 0<br>x<br>f<br>1 | lock:<br><br><pre>temp = *(u64*)(dst + offset)<br/>if *(u64*)(dst + offset) == R0<br/>    *(u64*)(dst + offset) = src<br/>R0 = temp</pre> | Atomic operations      |
| 0x<br>dc | 0<br>x<br>0 | 0<br>x<br>1<br>0 | dst = htole16(dst)  | Byte swap instructions |
| 0x<br>dc | 0<br>x<br>0 | 0<br>x<br>2<br>0 | dst = htole32(dst)  | Byte swap instructions |
| 0x<br>dc | 0<br>x<br>0 | 0<br>x<br>4<br>0 | dst = htole64(dst)  | Byte swap instructions |
| 0x<br>dd | a<br>n<br>y | 0<br>x<br>0<br>0 | if dst <= src goto +offset  | Jump instructions      |
| 0x<br>de | a<br>n<br>y | 0<br>x<br>0<br>0 | if (s32)dst <= (s32)src goto +offset  | Jump instructions      |