

Contents

1 eBPF Instruction Set Specification, v1.0	1
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 512 bytes of stack space.

1.1 Documentation conventions

This specification uses the standard C types (uint32_t, etc.) in documentation.

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: frame pointer to access stack

Registers R0 - R5 are scratch 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	$\text{dst} = (\text{src} \neq 0) ? (\text{dst} \% \text{src}) : \text{src}$
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 Byte swap instructions 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 set to the source value.

Examples:

BPF_ADD | BPF_X | BPF_ALU (0x0c) means:

```
dst = (uint32_t) (dst + src)
```

where '(uint32_t)' 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 = (uint32_t) src ^ (uint32_t) 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 the widely accepted mathematical definition given by Donald Knuth and implemented in C, Go, etc.:

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

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 non-default semantics of the 1-bit 'source' field in the 'opcode' field. 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	0x00	0x0	PC += offset	BPF_JMP only
BPF_JEQ	0x10	any	PC += offset if dst == src	

BPF_JGT	0x20	a n y	PC += offset if dst > src	unsigned
BPF_JGE	0x30	a n y	PC += offset if dst >= src	unsigned
BPF_JSET	0x40	a n y	PC += offset if dst & src	
BPF_JNE	0x50	a n y	PC += offset if dst != src	
BPF_JSGT	0x60	a n y	PC += offset if dst > src	signed
BPF_JSGE	0x70	a n y	PC += offset if dst >= src	signed
BPF_CALL	0x80	0 x 0	call helper function imm	see Helper functions
BPF_CALL	0x80	0 x 1	call PC += offset	see eBPF functions
BPF_CALL	0x80	0 x 2	call runtime function imm	see Runtime functions
BPF_EXIT	0x95	0 x 0	return	BPF_JMP only
BPF_JLT	0xa0	a n y	PC += offset if dst < src	unsigned
BPF_JLE	0xb0	a n y	PC += offset if dst <= src	unsigned
BPF_JSLT	0xc0	a n y	PC += offset if dst < src	signed
BPF_JSLE	0xd0	a n 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:

```
uint64_t function(uint64_t r1, uint64_t r2, uint64_t r3, uint64_t r4, uint64_t 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 not currently 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	64-bit immediate instructions
BPF_ABS	0x20	legacy BPF packet access (absolute)	Legacy BPF Packet access instructions
BPF_IND	0x40	legacy BPF packet access (indirect)	Legacy BPF Packet access instructions
BPF_MEM	0x60	regular load and store operations	Regular load and store operations
BPF_ATOMIC	0xc0	atomic operations	Atomic operations

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.

opcode construction	opcode	pseudocode
<code>BPF_MEM BPF_B BPF_LDX</code>	0x71	<code>dst = *(uint8_t *) (src + offset)</code>
<code>BPF_MEM BPF_H BPF_LDX</code>	0x69	<code>dst = *(uint16_t *) (src + offset)</code>
<code>BPF_MEM BPF_W BPF_LDX</code>	0x61	<code>dst = *(uint32_t *) (src + offset)</code>
<code>BPF_MEM BPF_DW BPF_LDX</code>	0x79	<code>dst = *(uint64_t *) (src + offset)</code>
<code>BPF_MEM BPF_B BPF_ST</code>	0x72	<code>*(uint8_t *) (dst + offset) = imm</code>
<code>BPF_MEM BPF_H BPF_ST</code>	0x6a	<code>*(uint16_t *) (dst + offset) = imm</code>
<code>BPF_MEM BPF_W BPF_ST</code>	0x62	<code>*(uint32_t *) (dst + offset) = imm</code>
<code>BPF_MEM BPF_DW BPF_ST</code>	0x7a	<code>*(uint64_t *) (dst + offset) = imm</code>
<code>BPF_MEM BPF_B BPF_STX</code>	0x73	<code>*(uint8_t *) (dst + offset) = src</code>
<code>BPF_MEM BPF_H BPF_STX</code>	0x6b	<code>*(uint16_t *) (dst + offset) = src</code>
<code>BPF_MEM BPF_W BPF_STX</code>	0x63	<code>*(uint32_t *) (dst + offset) = src</code>
<code>BPF_MEM BPF_DW BPF_STX</code>	0x7b	<code>*(uint64_t *) (dst + offset) = src</code>

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:

```
*(uint32_t *) (dst + offset) += src
```

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

```
*(uint64_t *) (dst + offset) += src
```

In addition to the simple atomic operations above, 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`.

1.5.3 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 co de	s r c	i m m	description	reference
0x 00	0 x 0	a n y	(additional immediate value)	64-bit immediate instructions
0x 04	0 x 0	a n y	dst = (uint32_t)(dst + imm)	Arithmetic instructions
0x 05	0 x 0	0 x 0 0	goto +offset	Jump instructions
0x 07	0 x 0	a n y	dst += imm	Arithmetic instructions
0x 0c	a n y	0 x 0 0	dst = (uint32_t)(dst + src)	Arithmetic instructions
0x 0f	a n y	0 x 0 0	dst += src	Arithmetic instructions

0x14	0 x 0	a n y	dst = (uint32_t)(dst - imm)	Arithmetic instructions
0x15	0 x 0	a n y	if dst == imm goto +offset	Jump instructions
0x16	0 x 0	a n y	if (uint32_t)dst == imm goto +offset	Jump instructions
0x17	0 x 0	a n y	dst -= imm	Arithmetic instructions
0x18	0 x 0 0 0	0 x 0 0 0	dst = imm64	64-bit immediate instructions
0x18	0 x 1	0 x 0 0 0	dst = map_by_fd(imm)	64-bit immediate instructions
0x18	0 x 2	0 x 0 0 0	dst = mva(map_by_fd(imm)) + next_imm	64-bit immediate instructions
0x18	0 x 3	0 x 0 0 0	dst = variable_addr(imm)	64-bit immediate instructions
0x18	0 x 4	0 x 0 0 0	dst = code_addr(imm)	64-bit immediate instructions
0x18	0 x 5	0 x 0 0 0	dst = map_by_idx(imm)	64-bit immediate instructions
0x18	0 x 6	0 x 0 0 0	dst = mva(map_by_idx(imm)) + next_imm	64-bit immediate instructions
0x1c	a n y	0 x 0 0 0	dst = (uint32_t)(dst - src)	Arithmetic instructions
0x1d	a n y	0 x 0 0 0	if dst == src goto +offset	Jump instructions

0x 1e	a n y	0 x 0 0	if (uint32_t)dst == (uint32_t)src goto +offset	Jump instructions
0x 1f	a n y	0 x 0 0	dst -= src	Arithmetic instructions
0x 20	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x 24	0 x 0	a n y	dst = (uint32_t)(dst * imm)	Arithmetic instructions
0x 25	0 x 0	a n y	if dst > imm goto +offset	Jump instructions
0x 26	0 x 0	a n y	if (uint32_t)dst > imm goto +offset	Jump instructions
0x 27	0 x 0	a n y	dst *= imm	Arithmetic instructions
0x 28	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x 2c	a n y	0 x 0 0	dst = (uint32_t)(dst * src)	Arithmetic instructions
0x 2d	a n y	0 x 0 0	if dst > src goto +offset	Jump instructions
0x 2e	a n y	0 x 0 0	if (uint32_t)dst > (uint32_t)src goto +offset	Jump instructions
0x 2f	a n y	0 x 0 0	dst *= src	Arithmetic instructions
0x 30	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x 34	0 x 0	a n y	dst = (uint32_t)((imm != 0) ? (dst / imm) : 0)	Arithmetic instructions
0x 35	0 x 0	a n y	if dst >= imm goto +offset	Jump instructions

0x36	0 x 0	a n y	if (uint32_t)dst >= imm goto +offset	Jump instructions
0x37	0 x 0	a n y	dst = (imm != 0) ? (dst / imm) : 0	Arithmetic instructions
0x38	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x3c	a n y	0 x 0 0	dst = (uint32_t)((imm != 0) ? (dst / src) : 0)	Arithmetic instructions
0x3d	a n y	0 x 0 0	if dst >= src goto +offset	Jump instructions
0x3e	a n y	0 x 0 0	if (uint32_t)dst >= (uint32_t)src goto +offset	Jump instructions
0x3f	a n y	0 x 0 0	dst = (src != 0) ? (dst / src) : 0	Arithmetic instructions
0x40	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x44	0 x 0	a n y	dst = (uint32_t)(dst imm)	Arithmetic instructions
0x45	0 x 0	a n y	if dst & imm goto +offset	Jump instructions
0x46	0 x 0	a n y	if (uint32_t)dst & imm goto +offset	Jump instructions
0x47	0 x 0	a n y	dst = imm	Arithmetic instructions
0x48	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x4c	a n y	0 x 0 0	dst = (uint32_t)(dst src)	Arithmetic instructions
0x4d	a n y	0 x 0 0	if dst & src goto +offset	Jump instructions

0x 4e	a n y	0 x 0 0	if (uint32_t)dst & (uint32_t)src goto +offset	Jump instructions
0x 4f	a n y	0 x 0 0	dst = src	Arithmetic instructions
0x 50	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x 54	0 x 0	a n y	dst = (uint32_t)(dst & imm)	Arithmetic instructions
0x 55	0 x 0	a n y	if dst != imm goto +offset	Jump instructions
0x 56	0 x 0	a n y	if (uint32_t)dst != imm goto +offset	Jump instructions
0x 57	0 x 0	a n y	dst &= imm	Arithmetic instructions
0x 58	a n y	a n y	(deprecated, implementation-specific)	Legacy BPF Packet access instructions
0x 5c	a n y	0 x 0 0	dst = (uint32_t)(dst & src)	Arithmetic instructions
0x 5d	a n y	0 x 0 0	if dst != src goto +offset	Jump instructions
0x 5e	a n y	0 x 0 0	if (uint32_t)dst != (uint32_t)src goto +offset	Jump instructions
0x 5f	a n y	0 x 0 0	dst &= src	Arithmetic instructions
0x 61	a n y	0 x 0 0	dst = *(uint32_t*)(src + offset)	Load and store instructions
0x 62	0 x 0	a n y	*(uint32_t*)(dst + offset) = imm	Load and store instructions

0x63	any	0x00	$*(uint32_t*)(dst + offset) = src$	Load and store instructions
0x64	0x00	any	$dst = (uint32_t)(dst \ll imm)$	Arithmetic instructions
0x65	0x00	any	if $dst \gg imm$ goto +offset	Jump instructions
0x66	0x00	any	if $(int32_t)dst \gg (int32_t)imm$ goto +offset	Jump instructions
0x67	0x00	any	$dst \ll= imm$	Arithmetic instructions
0x69	any	0x00	$dst = *(uint16_t*)(src + offset)$	Load and store instructions
0x6a	0x00	any	$*(uint16_t*)(dst + offset) = imm$	Load and store instructions
0x6b	any	0x00	$*(uint16_t*)(dst + offset) = src$	Load and store instructions
0x6c	any	0x00	$dst = (uint32_t)(dst \ll src)$	Arithmetic instructions
0x6d	any	0x00	if $dst \gg src$ goto +offset	Jump instructions
0x6e	any	0x00	if $(int32_t)dst \gg (int32_t)src$ goto +offset	Jump instructions
0x6f	any	0x00	$dst \ll= src$	Arithmetic instructions
0x71	any	0x00	$dst = *(uint8_t*)(src + offset)$	Load and store instructions
0x72	0x00	any	$*(uint8_t*)(dst + offset) = imm$	Load and store instructions

0x73	any	0x00	$*(uint8_t*)(dst + offset) = src$	Load and store instructions
0x74	0x00	any	$dst = (uint32_t)(dst \gg imm)$	Arithmetic instructions
0x75	0x00	any	if $dst \geq imm$ goto +offset	Jump instructions
0x76	0x00	any	if $(int32_t)dst \geq (int32_t)imm$ goto +offset	Jump instructions
0x77	0x00	any	$dst \gg= imm$	Arithmetic instructions
0x79	any	0x00	$dst = *(uint64_t*)(src + offset)$	Load and store instructions
0x7a	0x00	any	$*(uint64_t*)(dst + offset) = imm$	Load and store instructions
0x7b	any	0x00	$*(uint64_t*)(dst + offset) = src$	Load and store instructions
0x7c	any	0x00	$dst = (uint32_t)(dst \gg src)$	Arithmetic instructions
0x7d	any	0x00	if $dst \geq src$ goto +offset	Jump instructions
0x7e	any	0x00	if $(int32_t)dst \geq (int32_t)src$ goto +offset	Jump instructions
0x7f	any	0x00	$dst \gg= src$	Arithmetic instructions
0x84	0x00	0x00	$dst = (uint32_t)-dst$	Arithmetic instructions
0x85	0x00	any	call helper function imm	Helper functions

0x85	0 x 1	a n y	call PC += offset	eBPF functions
0x85	0 x 2	a n y	call runtime function imm	Runtime functions
0x87	0 x 0	0 x 0 0	dst = -dst	Arithmetic instructions
0x94	0 x 0	a n y	dst = (uint32_t)((imm != 0) ? (dst % imm) : imm)	Arithmetic instructions
0x95	0 x 0	0 x 0 0	return	Jump instructions
0x97	0 x 0	a n y	dst = (imm != 0) ? (dst % imm) : imm	Arithmetic instructions
0x9c	a n y	0 x 0 0	dst = (uint32_t)((src != 0) ? (dst % src) : src)	Arithmetic instructions
0x9f	a n y	0 x 0 0	dst = (src != 0) ? (dst % src) : src	Arithmetic instructions
0xa4	0 x 0	a n y	dst = (uint32_t)(dst ^ imm)	Arithmetic instructions
0xa5	0 x 0	a n y	if dst < imm goto +offset	Jump instructions
0xa6	0 x 0	a n y	if (uint32_t)dst < imm goto +offset	Jump instructions
0xa7	0 x 0	a n y	dst ^= imm	Arithmetic instructions
0xac	a n y	0 x 0 0	dst = (uint32_t)(dst ^ src)	Arithmetic instructions
0xad	a n y	0 x 0 0	if dst < src goto +offset	Jump instructions

0xae	any	0x00	if (uint32_t)dst < (uint32_t)src goto +offset	Jump instructions
0xaf	any	0x00	dst ^= src	Arithmetic instructions
0xb4	0x00	any	dst = (uint32_t) imm	Arithmetic instructions
0xb5	0x00	any	if dst <= imm goto +offset	Jump instructions
0xa6	0x00	any	if (uint32_t)dst <= imm goto +offset	Jump instructions
0xb7	0x00	any	dst = imm	Arithmetic instructions
0xbc	any	0x00	dst = (uint32_t) src	Arithmetic instructions
0xbd	any	0x00	if dst <= src goto +offset	Jump instructions
0xbe	any	0x00	if (uint32_t)dst <= (uint32_t)src goto +offset	Jump instructions
0xbf	any	0x00	dst = src	Arithmetic instructions
0xc3	any	0x00	lock *(uint32_t *) (dst + offset) += src	Atomic operations
0xc3	any	0x01	lock: <div> *(uint32_t *) (dst + offset) += src src = *(uint32_t *) (dst + offset) </div>	Atomic operations
0xc3	any	0x40	*(uint32_t *) (dst + offset) = src	Atomic operations

0xc3	any	0x41	lock: <div>*(uint32_t*)(dst + offset) = src src = *(uint32_t*)(dst + offset)</div>	Atomic operations
0xc3	any	0x50	*(uint32_t*)(dst + offset) &= src	Atomic operations
0xc3	any	0x51	lock: <div>*(uint32_t*)(dst + offset) &= src src = *(uint32_t*)(dst + offset)</div>	Atomic operations
0xc3	any	0xa0	*(uint32_t*)(dst + offset) ^= src	Atomic operations
0xc3	any	0xa1	lock: <div>*(uint32_t*)(dst + offset) ^= src src = *(uint32_t*)(dst + offset)</div>	Atomic operations
0xc3	any	0xe1	lock: <div>temp = *(uint32_t*)(dst + offset) *(uint32_t*)(dst + offset) = src src = temp</div>	Atomic operations
0xc3	any	0xf1	lock: <div>temp = *(uint32_t*)(dst + offset) if *(uint32_t*)(dst + offset) == R0 *(uint32_t*)(dst + offset) = src R0 = temp</div>	Atomic operations
0xc4	0x00	any	dst = (uint32_t)(dst s>> imm)	Arithmetic instructions
0xc5	0x00	any	if dst s< imm goto +offset	Jump instructions
0xc6	0x00	any	if (int32_t)dst s< (int32_t)imm goto +offset	Jump instructions
0xc7	0x00	any	dst s>>= imm	Arithmetic instructions
0xc	any	0x00	dst = (uint32_t)(dst s>> src)	Arithmetic instructions

0xcd	any	0x00	if dst s< src goto +offset	Jump instructions
0xce	any	0x00	if (int32_t)dst s< (int32_t)src goto +offset	Jump instructions
0xcf	any	0x00	dst s>>= src	Arithmetic instructions
0xd4	xx0	0x10	dst = htole16(dst)	Byte swap instructions
0xd4	xx0	0x20	dst = htole32(dst)	Byte swap instructions
0xd4	xx0	0x40	dst = htole64(dst)	Byte swap instructions
0xd5	any	0x0	if dst s<= imm goto +offset	Jump instructions
0xd6	any	0x0	if (int32_t)dst s<= (int32_t)imm goto +offset	Jump instructions
0xdb	any	0x00	lock *(uint64_t*)(dst + offset) += src	Atomic operations
0xdb	any	0x01	lock: <div> *(uint64_t*)(dst + offset) += src src = *(uint64_t*)(dst + offset) </div>	Atomic operations
0xdb	any	0x40	*(uint64_t*)(dst + offset) = src	Atomic operations
0xdb	any	0x41	lock: <div> *(uint64_t*)(dst + offset) = src lock src = *(uint64_t*)(dst + offset) </div>	Atomic operations
0xdb	any	0x50	*(uint64_t*)(dst + offset) &= src	Atomic operations

0x db	a n y	0 x 5 1	lock: <pre>*(uint64_t *) (dst + offset) &= src src = *(uint64_t *) (dst + offset)</pre>	Atomic operations
0x db	a n y	0 x a 0	*(uint64_t *) (dst + offset) ^= src	Atomic operations
0x db	a n y	0 x a 1	lock: <pre>*(uint64_t *) (dst + offset) ^= src src = *(uint64_t *) (dst + offset)</pre>	Atomic operations
0x db	a n y	0 x e 1	lock: <pre>temp = *(uint64_t *) (dst + offset) *(uint64_t *) (dst + offset) = src src = temp</pre>	Atomic operations
0x db	a n y	0 x f 1	lock: <pre>temp = *(uint64_t *) (dst + offset) if *(uint64_t *) (dst + offset) == R0 *(uint64_t *) (dst + offset) = src R0 = temp</pre>	Atomic operations
0x dc	0 x 0	0 x 1 0	dst = htobe16(dst)	Byte swap instructions
0x dc	0 x 0	0 x 2 0	dst = htobe32(dst)	Byte swap instructions
0x dc	0 x 0	0 x 4 0	dst = htobe64(dst)	Byte swap instructions
0x dd	a n y	0 x 0 0	if dst s<= src goto +offset	Jump instructions
0x de	a n y	0 x 0 0	if (int32_t)dst s<= (int32_t)src goto +offset	Jump instructions