Contents

BPF I	nstruction Set Architecture (ISA)	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 RPF Packet access instructions	12

1 BPF Instruction Set Architecture (ISA)

This document specifies the BPF instruction set architecture (ISA).

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
и	unsigned
S	signed

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

For example, *u32* is a type whose valid values are all the 32-bit unsigned numbers and *s16* is a 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: 1111111 10000110
```

1.1.4 Conformance groups

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

The use of named conformance groups enables interoperability between a runtime that executes instructions, and tools 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., "base32") that corresponds to a set of instructions that are mandatory. That is, each instruction has one or more conformance groups of which it is a member.

This document defines the following conformance groups:

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

1.2 Instruction encoding

BPF has two instruction encodings:

- the basic instruction encoding, which uses 64 bits to encode an instruction
- the wide instruction encoding, which appends a second 64-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:

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 two 32-bit immediate values that are 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:

Here, the imm value of the pseudo instruction is called 'next_imm'. 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	val ue	description	reference
BPF_LD	0x0 0	non-standard load operations	Load and store instructions
BPF_LD X	0x0 1	load into register operations	Load and store instructions
BPF_ST	0x0 2	store from immediate operations	Load and store instructions
BPF_ST X	0x0 3	store from register operations	Load and store instructions
BPF_AL U	0x0 4	32-bit arithmetic operations	Arithmetic and jump instructions
BPF_JM P	0x0 5	64-bit jump operations	Arithmetic and jump instructions
BPF_JM P32	0x0 6	32-bit jump operations	Arithmetic and jump instructions
BPF_AL U64	0x0 7	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. BPF_ALU64 instructions belong to the base64 conformance group unless noted otherwise. The 'code' field encodes the operation as below, where 'src' and 'dst' refer to the values of the source and destination registers, respectively.

	val			
code	ue	offset	description	
BPF_AD D	0x0 0	0	dst += src	
BPF_SU B	0x1 0	0	dst -= src	
BPF_MU L	0x2 0	0	dst *= src	
BPF_DIV	0x3 0	0	dst = (src != 0) ? (dst / src) : 0	
BPF_SDI V	0x3 0	1	dst = (src != 0) ? (dst s/ src) : 0	
BPF_OR	0x4 0	0	dst = src	
BPF_AN D	0x5 0	0	dst &= src	
BPF_LS H	0x6 0	0	dst <<= (src & mask)	
BPF_RS H	0x7 0	0	dst >>= (src & mask)	
BPF_NE G	0x8 0	0	dst = -dst	
BPF_MO D	0x9 0	0	dst = (src != 0) ? (dst % src) : dst	
BPF_SM OD	0x9 0	1	dst = (src != 0) ? (dst s% src) : dst	
BPF_XO R	0xa 0	0	dst ^= src	
BPF_MO V	0xb 0	0	dst = src	
BPF_MO VSX	0xb 0	8/16/3 2	dst = (s8,s16,s32)src	
BPF_AR SH	0xc 0	0	:term:`sign extending <sign extend="">` dst >>= (src & mask)</sign>	

BPF_END	0xd	0	byte swap operations (see Byte swap instructions below)
	0		

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) imm
```

BPF_XOR | BPF_K | BPF_ALU64 means:

```
dst = dst ^ imm
```

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

Division, multiplication, and modulo operations for BPF_ALU are part of the "divmul32" conformance group, and division, multiplication, and modulo operations for BPF_ALU64 are part of the "divmul64" conformance group. 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 * 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	valu e	description
BPF_ALU	BPF_TO_ LE	0x0 0	convert between host byte order and little endian
BPF_ALU	BPF_TO_ BE	0x0 8	convert between host byte order and big endian
BPF_ALU 64	Reserved	0x0 0	do byte swap unconditionally

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

Examples:

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

```
dst = htole16(dst)
dst = htole32(dst)
dst = htole64(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 and indicates the base32 conformance group, while BPF_JMP uses 64-bit wide operands for otherwise identical operations, and indicates the base64 conformance group unless otherwise specified. The 'code' field encodes the operation as below:

code	va Iu e	src_ reg	description	notes
BPF_ JA	0x 0	0x0	PC += offset	BPF_JMP BPF_K only
BPF_ JA	0x 0	0x0	PC += imm	BPF_JMP32 BPF_K only
BPF_ JEQ	0x 1	any	PC += offset if dst == src	

BPF_ JGT	0x 2	any	PC += offset if dst > src	unsigned
BPF_ JGE	0x 3	any	PC += offset if dst >= src	unsigned
BPF_ JSET	0x 4	any	PC += offset if dst & src	
BPF_ JNE	0x 5	any	PC += offset if dst != src	
BPF_ JSG T	0x 6	any	PC += offset if dst > src	signed
BPF_ JSG E	0x 7	any	PC += offset if dst >= src	signed
BPF_ CALL	0x 8	0x0	call helper function by address	BPF_JMP BPF_K only, see Helper functions
BPF_ CALL	0x 8	0x1	call PC += imm	BPF_JMP BPF_K only, see Program-local functions
BPF_ CALL	0x 8	0x2	call helper function by BTF ID	BPF_JMP BPF_K only, see Helper functions
BPF_ EXIT	0x 9	0x0	return	BPF_JMP BPF_K only
BPF_ JLT	0x a	any	PC += offset if dst < src	unsigned
BPF_ JLE	0x b	any	PC += offset if dst <= src	unsigned
BPF_ JSLT	0x c	any	PC += offset if dst < src	signed
BPF_ JSLE	0x d	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:
```

```
gotol +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.

All BPF_CALL and BPF_JA instructions belong to the base32 conformance group.

1.3.3.1 Helper functions

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

Historically, each helper function was identified by 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 <code>BPF_JA</code>. The offset is encoded in the imm field of the call instruction. A <code>BPF_EXIT</code> 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	va lu e	description	reference
BPF_IMM	0x 00	64-bit immediate instructions	64-bit immediate instructions
BPF_ABS	0x 20	legacy BPF packet access (absolute)	Legacy BPF Packet access instructions
BPF_IND	0x 40	legacy BPF packet access (indirect)	Legacy BPF Packet access instructions
BPF_MEM	0x 60	regular load and store operations	Regular load and store operations
BPF_MEM SX	0x 80	sign-extension load operations	Sign-extension load operations
BPF_ATO MIC	0x c0	atomic operations	Atomic operations

The size modifier is 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)	

Instructions using ${\tt BPF_DW}$ belong to the base64 conformance group.

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) = imm
```

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, which are part of the "atomic32" conformance group.
- BPF_ATOMIC | BPF_DW | BPF_STX for 64-bit operations, which are part of the "atomic64" conformance group.
- 8-bit and 16-bit wide atomic operations are not supported.

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

imm	value	description
BPF_ADD	0x00	atomic add
BPF_OR	0x40	atomic or
BPF_AND	0x50	atomic and
BPF_XOR	0xa0	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 reg' field of the basic instruction to hold an opcode subtype.

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

opcode construction	op co de	src _re g	pseudocode	imm type	dst type
BPF_IMM BPF_DW BPF_LD	0x 18	0x0	dst = (next_imm << 32) imm	integer	integer
BPF_IMM BPF_DW BPF_LD	0x 18	0x1	dst = map_by_fd(imm)	map fd	map
BPF_IMM BPF_DW BPF_LD	0x 18	0x2	dst = map_val(map_by_fd(imm)) + next_imm	map fd	data pointer
BPF_IMM BPF_DW BPF_LD	0x 18	0x3	dst = var_addr(imm)	variable id	data pointer
BPF_IMM BPF_DW BPF_LD	0x 18	0x4	dst = code_addr(imm)	integer	code pointer
BPF_IMM BPF_DW BPF_LD	0x 18	0x5	dst = map_by_idx(imm)	map index	map
BPF_IMM BPF_DW BPF_LD	0x 18	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. These instructions used an instruction class of BPF_LD, a size modifier of BPF_W, BPF_H, or BPF_B, and a mode modifier of BPF_ABS or BPF_IND. The 'dst_reg' and 'offset' fields were set to zero, and 'src_reg' was set to zero for BPF_ABS. However, these instructions are deprecated and should no longer be used. All legacy packet access instructions belong to the "legacy" conformance group.