Stream: Internet Engineering Task Force (IETF)
RFC: 9669
Category: Standards Track
Published: October 2024
ISSN: 2070-1721
Author: D. Thaler, Ed.

RFC 9669
BPF Instruction Set Architecture (ISA)

Abstract

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

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9669.

Copyright Notice

Copyright (c) 2024 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction 3
2. Documentation Conventions 3
   2.1. Types 4
   2.2. Functions 4
   2.3. Definitions 5
   2.4. Conformance Groups 5
3. Instruction Encoding 6
   3.1. Basic Instruction Encoding 6
   3.2. Wide Instruction Encoding 7
   3.3. Instruction Classes 8
4. Arithmetic and Jump Instructions 9
   4.1. Arithmetic Instructions 9
   4.2. Byte Swap Instructions 12
   4.3. Jump Instructions 13
      4.3.1. Helper Functions 15
      4.3.2. Program-Local Functions 15
5. Load and Store Instructions 15
   5.1. Regular Load and Store Operations 16
   5.2. Sign-Extension Load Operations 17
   5.3. Atomic Operations 17
   5.4. 64-bit Immediate Instructions 18
      5.4.1. Maps 19
      5.4.2. Platform Variables 19
   5.5. Legacy BPF Packet Access Instructions 20
6. Security Considerations 20

7. IANA Considerations 20
7.1. BPF Instruction Conformance Group Registry 20
7.1.1. BPF Instruction Conformance Group Registration Template 21
7.2. BPF Instruction Set Registry 22
7.2.1. BPF Instruction Registration Template 22
7.3. Adding Instructions 23
7.4. Deprecating Instructions 24
7.5. Change Control 24
7.6. Expert Review Instructions 25
8. References 25
8.1. Normative References 25
8.2. Informative References 25
Appendix A. Initial BPF Instruction Set Values 26
Acknowledgements 37
Author's Address 38

1. Introduction

eBPF, also commonly referred to as BPF, is a technology with origins in the Linux kernel that can run untrusted programs in a privileged context such as an operating system kernel. This document specifies the BPF instruction set architecture (ISA).

As a historical note, BPF originally stood for Berkeley Packet Filter, but now that it can do so much more than packet filtering, the acronym no longer makes sense. BPF is now considered a standalone term that does not stand for anything. The original BPF is sometimes referred to as cBPF (classic BPF) to distinguish it from the now widely deployed eBPF (extended BPF).

2. Documentation Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

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

2.1. Types

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

S	Meaning
u	unsigned
s	signed

Table 1: Meaning of Signedness Notation

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

Table 2: Meaning of Bit-Width Notation

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

2.2. Functions

The following byteswap functions are direction-agnostic. That is, the same function is used for conversion in either direction discussed below.

- be16: Takes an unsigned 16-bit number and converts it between host byte order and big-endian (IEN137 [IEN137]) byte order.
- be32: Takes an unsigned 32-bit number and converts it between host byte order and big-endian byte order.
- be64: Takes an unsigned 64-bit number and converts it between host byte order and big-endian byte order.
- 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.
- le16: Takes an unsigned 16-bit number and converts it between host byte order and little-endian byte order.
- le32: Takes an unsigned 32-bit number and converts it between host byte order and little-endian byte order.
- le64: Takes an unsigned 64-bit number and converts it between host byte order and little-endian byte order.

2.3. Definitions

Sign Extend: To *sign extend an* `X` *-bit number, A, to a* `Y` *-bit number, B,* means to

To *sign extend* `X`-bit number, *A*, to a `Y`-bit number, *B*, means to

1. Copy all `X` bits from *A* to the lower `X` bits of *B*.
2. Set the value of the remaining `Y` - `X` bits of *B* to the value of the most-significant bit of *A*.

> **Example**
>
> Sign extend an 8-bit number `A` to a 16-bit number `B` on a big-endian platform:
>
> ```
> A:          10000110
> B: 11111111 10000110
> ```

2.4. Conformance Groups

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

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

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

This document defines the following conformance groups:

base32: includes all instructions defined in this specification unless otherwise noted.

base64: includes base32, plus instructions explicitly noted as being in the base64 conformance group.

atomic32: includes 32-bit atomic operation instructions (see Atomic operations (Section 5.3)).

atomic64: includes atomic32, plus 64-bit atomic operation instructions.

divmul32: includes 32-bit division, multiplication, and modulo instructions.

divmul64: includes divmul32, plus 64-bit division, multiplication, and modulo instructions.

packet: deprecated packet access instructions.

3. Instruction Encoding

BPF has two instruction encodings:

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

3.1. Basic Instruction Encoding

A basic instruction is encoded as follows:

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

opcode: operation to perform, encoded as follows:

```
+-+-+-+-+-+-+-+-+
|specific |class|
+-+-+-+-+-+-+-+-+
```

specific: The format of these bits varies by instruction class

class: The instruction class (see Instruction classes (Section 3.3))

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

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

and as follows on a big-endian host:

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

src_reg: the source register number (0-10), except where otherwise specified (64-bit immediate instructions (Section 5.4) reuse this field for other purposes)

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

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

imm: signed integer immediate value

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

For example:

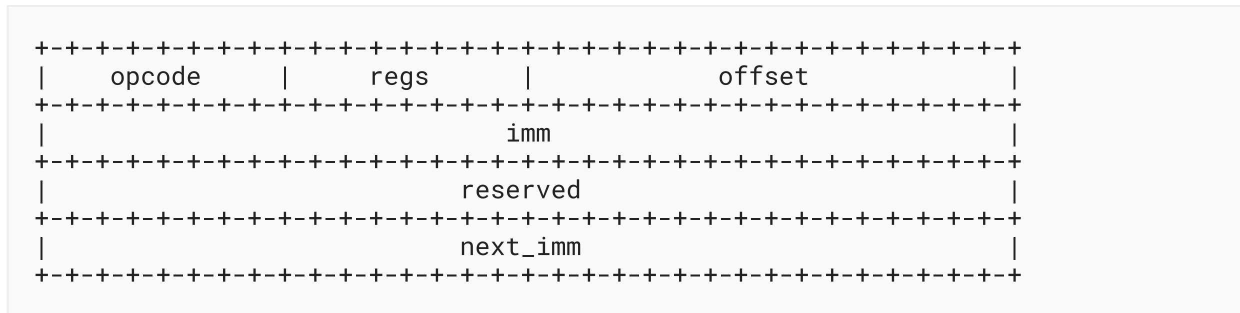
```
opcode                  offset imm          assembly
       src_reg dst_reg
07     0       1        00 00  44 33 22 11  r1 += 0x11223344 // little
       dst_reg src_reg
07     1       0        00 00  11 22 33 44  r1 += 0x11223344 // big
```

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

3.2. Wide Instruction Encoding

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

This is depicted in the following figure:



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

opcode: operation to perform, encoded as explained above

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

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

imm: signed integer immediate value

reserved: unused, set to zero

next_imm: second signed integer immediate value

3.3. Instruction Classes

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

class	value	description	reference
LD	0x0	non-standard load operations	Load and store instructions (Section 5)
LDX	0x1	load into register operations	Load and store instructions (Section 5)
ST	0x2	store from immediate operations	Load and store instructions (Section 5)
STX	0x3	store from register operations	Load and store instructions (Section 5)
ALU	0x4	32-bit arithmetic operations	Arithmetic and jump instructions (Section 4)
JMP	0x5	64-bit jump operations	Arithmetic and jump instructions (Section 4)
JMP32	0x6	32-bit jump operations	Arithmetic and jump instructions (Section 4)

class	value	description	reference
ALU64	0x7	64-bit arithmetic operations	Arithmetic and jump instructions (Section 4)

Table 3: Instruction Class

4. Arithmetic and Jump Instructions

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

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

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

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

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

Table 4: Source Operand Location

instruction class: the instruction class (see Instruction classes (Section 3.3))

4.1. Arithmetic Instructions

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

name	code	offset	description
ADD	0x0	0	dst += src
SUB	0x1	0	dst -= src
MUL	0x2	0	dst *= src
DIV	0x3	0	dst = (src != 0) ? (dst / src) : 0

name	code	offset	description
SDIV	0x3	1	dst = (src != 0) ? (dst s/ src) : 0
OR	0x4	0	dst |= src
AND	0x5	0	dst &= src
LSH	0x6	0	dst <<= (src & mask)
RSH	0x7	0	dst >>= (src & mask)
NEG	0x8	0	dst = -dst
MOD	0x9	0	dst = (src != 0) ? (dst % src) : dst
SMOD	0x9	1	dst = (src != 0) ? (dst s% src) : dst
XOR	0xa	0	dst ^= src
MOV	0xb	0	dst = src
MOVSX	0xb	8/16/32	dst = (s8,s16,s32)src
ARSH	0xc	0	sign extending (Section 2.3) dst >>= (src & mask)
END	0xd	0	byte swap operations (see Byte swap instructions (Section 4.2) below)

Table 5: Arithmetic Instructions

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

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

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

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

`{ADD, X, ALU64}` means:

```
dst = dst + src
```

`{XOR, K, ALU}` means:

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

`{XOR, K, ALU64}` means:

```
dst = dst ^ imm
```

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

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

For unsigned operations (`DIV` and `MOD`), for `ALU`, 'imm' is interpreted as a 32-bit unsigned value. For `ALU64`, 'imm' is first sign extended (Section 2.3) from 32 to 64 bits, and then interpreted as a 64-bit unsigned value.

For signed operations (`SDIV` and `SMOD`), for `ALU`, 'imm' is interpreted as a 32-bit signed value. For `ALU64`, 'imm' is first sign extended (Section 2.3) from 32 to 64 bits, and then interpreted as a 64-bit signed value.

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

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

The `MOVSX` instruction does a move operation with sign extension. `{MOVSX, X, ALU}` sign extends (Section 2.3) 8-bit and 16-bit operands into 32-bit operands, and zeroes the remaining upper 32 bits. `{MOVSX, X, ALU64}` sign extends (Section 2.3) 8-bit, 16-bit, and 32-bit operands into 64-bit operands. Unlike other arithmetic instructions, `MOVSX` is only defined for register source operands (`X`).

`{MOV, K, ALU64}` means:

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

`{MOV, X, ALU}` means:

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

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

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

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

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

4.2. Byte Swap Instructions

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

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

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

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

Table 6: Byte Swap Instructions

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

Examples:

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

```
dst = le16(dst)
dst = le32(dst)
dst = le64(dst)
```

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

```
dst = be16(dst)
dst = be32(dst)
dst = be64(dst)
```

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

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

4.3. Jump Instructions

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

code	value	src_reg	description	notes
JA	0x0	0x0	PC += offset	{JA, K, JMP} only
JA	0x0	0x0	PC += imm	{JA, K, JMP32} only
JEQ	0x1	any	PC += offset if dst == src	
JGT	0x2	any	PC += offset if dst > src	unsigned
JGE	0x3	any	PC += offset if dst >= src	unsigned
JSET	0x4	any	PC += offset if dst & src	
JNE	0x5	any	PC += offset if dst != src	
JSGT	0x6	any	PC += offset if dst > src	signed
JSGE	0x7	any	PC += offset if dst >= src	signed
CALL	0x8	0x0	call helper function by static ID	{CALL, K, JMP} only, see Helper functions (Section 4.3.1)
CALL	0x8	0x1	call PC += imm	{CALL, K, JMP} only, see Program-local functions (Section 4.3.2)
CALL	0x8	0x2	call helper function by BTF ID	{CALL, K, JMP} only, see Helper functions (Section 4.3.1)
EXIT	0x9	0x0	return	{CALL, K, JMP} only

code	value	src_reg	description	notes
JLT	0xa	any	PC += offset if dst < src	unsigned
JLE	0xb	any	PC += offset if dst <= src	unsigned
JSLT	0xc	any	PC += offset if dst < src	signed
JSLE	0xd	any	PC += offset if dst <= src	signed

Table 7: Jump Instructions

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

Example:

`{JSGE, X, JMP32}` means:

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

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

`{JLE, K, JMP}` means:

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

`{JA, K, JMP32}` means:

```
  gotol +imm
```

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

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

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

4.3.1. Helper Functions

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

Historically, each helper function was identified by a static ID encoded in the 'imm' field. Further documentation of helper functions is outside the scope of this document and standardization is left for future work, but use is widely deployed and more information can be found in platform-specific documentation (e.g., Linux kernel documentation).

Platforms that support the BPF Type Format (BTF) support identifying a helper function by a BTF ID encoded in the 'imm' field, where the BTF ID identifies the helper name and type. Further documentation of BTF is outside the scope of this document and standardization is left for future work, but use is widely deployed and more information can be found in platform-specific documentation (e.g., Linux kernel documentation).

4.3.2. Program-Local Functions

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

5. Load and Store Instructions

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

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

mode The mode modifier is one of:

mode modifier	value	description	reference
IMM	0	64-bit immediate instructions	64-bit immediate instructions (Section 5.4)
ABS	1	legacy BPF packet access (absolute)	Legacy BPF Packet access instructions (Section 5.5)
IND	2	legacy BPF packet access (indirect)	Legacy BPF Packet access instructions (Section 5.5)

mode modifier	value	description	reference
MEM	3	regular load and store operations	Regular load and store operations (Section 5.1)
MEMSX	4	sign-extension load operations	Sign-extension load operations (Section 5.2)
ATOMIC	6	atomic operations	Atomic operations (Section 5.3)

Table 8: Mode Modifier

sz (size) The size modifier is one of:

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

Table 9: Size Modifier

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

class The instruction class (see Instruction classes (Section 3.3))

5.1. Regular Load and Store Operations

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

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

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

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

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

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

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

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

5.2. Sign-Extension Load Operations

The `MEMSX` mode modifier is used to encode sign-extension (Section 2.3) load instructions that transfer data between a register and memory.

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

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

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

5.3. Atomic Operations

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

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

- `{ATOMIC, W, STX}` for 32-bit operations, which are part of the "atomic32" conformance group.
- `{ATOMIC, DW, STX}` for 64-bit operations, which are part of the "atomic64" conformance group.
- 8-bit and 16-bit wide atomic operations are not supported.

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

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

Table 10: Simple Atomic Operations

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

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

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

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

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

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

Table 11: Complex Atomic Operations

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

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

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

5.4. 64-bit Immediate Instructions

Instructions with the `IMM` 'mode' modifier use the wide instruction encoding defined in Instruction encoding (Section 3), and use the 'src_reg' field of the basic instruction to hold an opcode subtype.

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

src_reg	pseudocode	imm type	dst type
0x0	dst = (next_imm << 32) | imm	integer	integer
0x1	dst = map_by_fd(imm)	map fd	map
0x2	dst = map_val(map_by_fd(imm)) + next_imm	map fd	data address

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

Table 12: 64-bit Immediate Instructions

where

- map_by_fd(imm) means to convert a 32-bit file descriptor into an address of a map (see Maps (Section 5.4.1))
- 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 (Section 5.4.2)) 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

5.4.1. 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.

5.4.2. 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.

5.5. Legacy BPF Packet Access Instructions

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

6. Security Considerations

BPF programs could use BPF instructions to do malicious things with memory, CPU, networking, or other system resources. This is not fundamentally different from any other type of software that may run on a device. Execution environments should be carefully designed to only run BPF programs that are trusted and verified, and sandboxing and privilege level separation are key strategies for limiting security and abuse impact. For example, BPF verifiers are well-known and widely deployed and are responsible for ensuring that BPF programs will terminate within a reasonable time, only interact with memory in safe ways, adhere to platform-specified API contracts, and don't use instructions with undefined behavior. This level of verification can often provide a stronger level of security assurance than for other software and operating system code. While the details are out of scope of this document, Linux [LINUX] and PREVAIL [PREVAIL] do provide many details. Future IETF work will document verifier expectations and building blocks for allowing safe execution of untrusted BPF programs.

Executing programs using the BPF instruction set also requires either an interpreter or a compiler to translate them to hardware processor native instructions. In general, interpreters are considered a source of insecurity (e.g., gadgets susceptible to side-channel attacks due to speculative execution) whenever one is used in the same memory address space as data with confidentiality concerns. As such, use of a compiler is recommended instead. Compilers should be audited carefully for vulnerabilities to ensure that compilation of a trusted and verified BPF program to native processor instructions does not introduce vulnerabilities.

Exposing functionality via BPF extends the interface between the component executing the BPF program and the component submitting it. Careful consideration of what functionality is exposed and how that impacts the security properties desired is required.

7. IANA Considerations

This document defines two registries.

7.1. BPF Instruction Conformance Group Registry

This document defines an IANA registry for BPF instruction conformance groups, as follows:

- Name of the registry: BPF Instruction Conformance Groups
- Name of the registry group: BPF Instructions

- Required information for registrations: See BPF Instruction Conformance Group Registration Template (Section 7.1.1)
- Syntax of registry entries: Each entry has the following fields: name, description, includes, excludes, status, and reference. See BPF Instruction Conformance Group Registration Template (Section 7.1.1) for more details.
- Registration policy (see Section 4 of [RFC8126] for details):
  - Permanent: Standards action or IESG Approval
  - Provisional: Specification required
  - Historical: Specification required

Initial entries in this registry are as follows:

Name	Description	Includes	Excludes	Status	Reference
atomic32	32-bit atomic instructions	-	-	Permanent	RFC 9669, Section 5.3
atomic64	64-bit atomic instructions	atomic32	-	Permanent	RFC 9669, Section 5.3
base32	32-bit base instructions	-	-	Permanent	RFC 9669
base64	64-bit base instructions	base32	-	Permanent	RFC 9669
divmul32	32-bit division and modulo	-	-	Permanent	RFC 9669, Section 4.1
divmul64	64-bit division and modulo	divmul32	-	Permanent	RFC 9669, Section 4.1
packet	Legacy packet instructions	-	-	Historical	RFC 9669, Section 5.5

Table 13: Initial Conformance Groups

7.1.1. BPF Instruction Conformance Group Registration Template

This template describes the fields that must be supplied in a registration request:

Name: Alphanumeric label indicating the name of the conformance group.

Description: Brief description of the conformance group.

Includes: Any other conformance groups that are included by this group.

Excludes: Any other conformance groups that are excluded by this group.

Status: This reflects the status requested and must be one of 'Permanent', 'Provisional', or 'Historical'.

Contact: Person (including contact information) to contact for further information.

Change controller: Organization or person (often the author), including contact information, authorized to change this.

Reference: A reference to the defining specification. Include full citations for all referenced documents. Registration requests for 'Provisional' registration can be included in an Internet-Draft; when the documents are approved for publication as an RFC, the registration will be updated.

7.2. BPF Instruction Set Registry

This document proposes a new IANA registry for BPF instructions, as follows:

- Name of the registry: BPF Instruction Set
- Name of the registry group: BPF Instructions
- Required information for registrations: See BPF Instruction Registration Template (Section 7.2.1)
- Syntax of registry entries: Each entry has the following fields: opcode, src, imm, offset, description, groups, and reference. See BPF Instruction Registration Template (Section 7.2.1) for more details.
- Registration policy: New instructions require a new entry in the conformance group registry and the same registration policies apply.
- Initial registrations: See the Appendix. Instructions other than those listed as deprecated are Permanent. Any listed as deprecated are Historical.

7.2.1. BPF Instruction Registration Template

This template describes the fields that must be supplied in a registration request:

Opcode: A 1-byte value in hex format indicating the value of the opcode field

Src: Either a numeric value indicating the value of the src field, or "any"

Imm: Either a value indicating the value of the imm field, or "any"

Offset: Either a numeric value indicating the value of the offset field, or "any"

Description: Description of what the instruction does, typically in pseudocode

Groups: A list of one or more comma-separated conformance groups to which the instruction belongs

Contact: Person (including contact information) to contact for further information.

Change controller: Organization or person (often the author), including contact information, authorized to change this.

Reference: A reference to the defining specification. Include full citations for all referenced documents. Registration requests for 'Provisional' registration can be included in an Internet-Draft; when the documents are approved for publication as an RFC, the registration will be updated.

7.3. Adding Instructions

A specification may add additional instructions to the BPF Instruction Set registry. Once a conformance group is registered with a set of instructions, no further instructions can be added to that conformance group. A specification should instead create a new conformance group that includes the original conformance group, plus any newly added instructions. Inclusion of the original conformance group is done via the "includes" column of the BPF Instruction Conformance Group Registry, and inclusion of newly added instructions is done via the "groups" column of the BPF Instruction Set Registry.

For example, consider an existing hypothetical group called "example" with two instructions in it. One might add two more instructions by first adding an "examplev2" group to the BPF Instruction Conformance Group Registry as follows:

name	description	includes	excludes	status
example	Original example instructions	-	-	Permanent
examplev2	Newer set of example instructions	example	-	Permanent

Table 14: Conformance Group Example for Addition

And then adding the new instructions into the BPF Instruction Set Registry as follows:

opcode	...	description	groups
aaa	...	Original example instruction 1	example
bbb	...	Original example instruction 2	example
ccc	...	Added example instruction 3	examplev2
ddd	...	Added example instruction 4	examplev2

Table 15: Instruction Addition Example

Supporting the "examplev2" group thus requires supporting all four example instructions.

7.4. Deprecating Instructions

Deprecating instructions that are part of an existing conformance group can be done by defining a new conformance group for the newly deprecated instructions, and defining a new conformance group that supersedes the existing conformance group containing the instructions, where the new conformance group includes the existing one and excludes the deprecated instruction group.

For example, if deprecating an instruction in an existing hypothetical group called "example", two new groups ("legacyexample" and "examplev2") might be registered in the BPF Instruction Conformance Group Registry as follows:

name	description	includes	excludes	status
example	Original example instructions	-	-	Permanent
legacyexample	Legacy example instructions	-	-	Historical
examplev2	Example instructions	example	legacyexample	Permanent

Table 16: Conformance Group Example for Deprecation

The BPF Instruction Set registry entries for the deprecated instructions would then be updated to add "legacyexample" to the set of groups for those instructions, as follows:

opcode	...	description	groups
aaa	...	Good original instruction 1	example
bbb	...	Good original instruction 2	example
ccc	...	Bad original instruction 3	example, legacyexample
ddd	...	Bad original instruction 4	example, legacyexample

Table 17: Instruction Deprecation Example

Finally, updated implementations that dropped support for the deprecated instructions would then be able to claim conformance to "examplev2" rather than "example".

7.5. Change Control

Registrations can be updated in a registry by the same mechanism as required for an initial registration. In cases where the original definition of an entry is contained in an IESG-approved document, update of the specification also requires IESG approval.

'Provisional' registrations can be updated by the change controller designated in the existing registration. In addition, the IESG can reassign responsibility for a 'Provisional' registration or can request specific changes to an entry. This will enable changes to be made to entries where the original registrant is out of contact or unwilling or unable to make changes.

Transition from 'Provisional' to 'Permanent' status can be requested and approved in the same manner as a new 'Permanent' registration. Transition from 'Permanent' to 'Historical' status requires IESG approval. Transition from 'Provisional' to 'Historical' can be requested by anyone authorized to update the 'Provisional' registration.

7.6. Expert Review Instructions

The IANA registries established by this document are informed by written specifications, which themselves are facilitated and approved by an Expert Review Section 5.3 of [RFC8126] process.

Designated Experts are expected to consult with the active BPF working group (e.g., via email to the working group's mailing list) if it exists, as well as other interested parties (e.g., via email to one or more active mailing list(s) for relevant BPF communities and platforms). The Designed Expert is expected to verify that the encoding and semantics for any new instructions are properly documented in a public-facing specification. In the event of future RFC documents for ISA extensions, experts may permit early assignment before the RFC document is available, as long as a specification exists which satisfies the above requirements.

8. References

8.1. Normative References

[IEN137] Cohen, D., "On Holy Wars and a Plea for Peace", IEN 137, April 1980.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[LINUX] "eBPF verifier", <https://www.kernel.org/doc/html/latest/bpf/verifier.html>.

[PREVAIL] Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J., Rinetzky, N., Ryzhyk, L., and M. Sagiv, "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions", DOI 10.1145/3314221.3314590, June 2019, <https://doi.org/10.1145/3314221.3314590>.

Appendix A. Initial BPF Instruction Set Values

Initial values for the BPF Instruction Set registry are given below. The descriptions in this table are informative. In case of any discrepancy, the reference is authoritative.

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0x00	0x0	0	any	(additional immediate value)	base64	RFC 9669, Section 5.4
0x04	0x0	0	any	dst = (u32)((u32)dst + (u32)imm)	base32	RFC 9669, Section 4.1
0x05	0x0	any	0x00	goto +offset	base32	RFC 9669, Section 4.3
0x06	0x0	0	any	goto +imm	base32	RFC 9669, Section 4.3
0x07	0x0	0	any	dst += imm	base64	RFC 9669, Section 4.1
0x0c	any	0	0x00	dst = (u32)((u32)dst + (u32)src)	base32	RFC 9669, Section 4.1
0x0f	any	0	0x00	dst += src	base64	RFC 9669, Section 4.1
0x14	0x0	0	any	dst = (u32)((u32)dst - (u32)imm)	base32	RFC 9669, Section 4.1
0x15	0x0	any	any	if dst == imm goto +offset	base64	RFC 9669, Section 4.3
0x16	0x0	any	any	if (u32)dst == imm goto +offset	base32	RFC 9669, Section 4.3
0x17	0x0	0	any	dst -= imm	base64	RFC 9669, Section 4.1

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0x18	0x0	0	any	dst = (next_imm << 32) | imm	base64	RFC 9669, Section 5.4
0x18	0x1	0	any	dst = map_by_fd(imm)	base64	RFC 9669, Section 5.4
0x18	0x2	0	any	dst = map_val(map_by_fd(imm)) + next_imm	base64	RFC 9669, Section 5.4
0x18	0x3	0	any	dst = var_addr(imm)	base64	RFC 9669, Section 5.4
0x18	0x4	0	any	dst = code_addr(imm)	base64	RFC 9669, Section 5.4
0x18	0x5	0	any	dst = map_by_idx(imm)	base64	RFC 9669, Section 5.4
0x18	0x6	0	any	dst = map_val(map_by_idx(imm)) + next_imm	base64	RFC 9669, Section 5.4
0x1c	any	0	0x00	dst = (u32)((u32)dst - (u32)src)	base32	RFC 9669, Section 4.1
0x1d	any	any	0x00	if dst == src goto +offset	base64	RFC 9669, Section 4.3
0x1e	any	any	0x00	if (u32)dst == (u32)src goto +offset	base32	RFC 9669, Section 4.3
0x1f	any	0	0x00	dst -= src	base64	RFC 9669, Section 4.1
0x20	0x0	0	any	(deprecated, implementation-specific)	packet	RFC 9669, Section 5.5
0x24	0x0	0	any	dst = (u32)(dst * imm)	divmul32	RFC 9669, Section 4.1
0x25	0x0	any	any	if dst > imm goto +offset	base64	RFC 9669, Section 4.3

Opcode	src_reg	Off-set	imm	Description	Groups	Ref
0x26	0x0	any	any	if (u32)dst > imm goto +offset	base32	RFC 9669, Section 4.3
0x27	0x0	0	any	dst *= imm	divmul64	RFC 9669, Section 4.1
0x28	0x0	0	any	(deprecated, implementation-specific)	packet	RFC 9669, Section 5.5
0x2c	any	0	0x00	dst = (u32)(dst * src)	divmul32	RFC 9669, Section 4.1
0x2d	any	any	0x00	if dst > src goto +offset	base64	RFC 9669, Section 4.3
0x2e	any	any	0x00	if (u32)dst > (u32)src goto +offset	base32	RFC 9669, Section 4.3
0x2f	any	0	0x00	dst *= src	divmul64	RFC 9669, Section 4.1
0x30	0x0	0	any	(deprecated, implementation-specific)	packet	RFC 9669, Section 5.5
0x34	0x0	0	any	dst = (u32)((imm != 0) ? ((u32)dst / (u32)imm) : 0)	divmul32	RFC 9669, Section 4.1
0x34	0x0	1	any	dst = (u32)((imm != 0) ? ((s32)dst s/ imm) : 0)	divmul32	RFC 9669, Section 4.1
0x35	0x0	any	any	if dst >= imm goto +offset	base64	RFC 9669, Section 4.3
0x36	0x0	any	any	if (u32)dst >= imm goto +offset	base32	RFC 9669, Section 4.3
0x37	0x0	0	any	dst = (imm != 0) ? (dst / (u32)imm) : 0	divmul64	RFC 9669, Section 4.1
0x37	0x0	1	any	dst = (imm != 0) ? (dst s/ imm) : 0	divmul64	RFC 9669, Section 4.1
0x3c	any	0	0x00	dst = (u32)((src != 0) ? ((u32)dst / (u32)src) : 0)	divmul32	RFC 9669, Section 4.1

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0x3c	any	1	0x00	dst = (u32)((src != 0) ? ((s32)dst s/(s32)src) : 0)	divmul32	RFC 9669, Section 4.1
0x3d	any	any	0x00	if dst >= src goto +offset	base64	RFC 9669, Section 4.3
0x3e	any	any	0x00	if (u32)dst >= (u32)src goto +offset	base32	RFC 9669, Section 4.3
0x3f	any	0	0x00	dst = (src != 0) ? (dst / src) : 0	divmul64	RFC 9669, Section 4.1
0x3f	any	1	0x00	dst = (src != 0) ? (dst s/ src) : 0	divmul64	RFC 9669, Section 4.1
0x40	any	0	any	(deprecated, implementation-specific)	packet	RFC 9669, Section 5.5
0x44	0x0	0	any	dst = (u32)(dst | imm)	base32	RFC 9669, Section 4.1
0x45	0x0	any	any	if dst & imm goto +offset	base64	RFC 9669, Section 4.3
0x46	0x0	any	any	if (u32)dst & imm goto +offset	base32	RFC 9669, Section 4.3
0x47	0x0	0	any	dst |= imm	base64	RFC 9669, Section 4.1
0x48	any	0	any	(deprecated, implementation-specific)	packet	RFC 9669, Section 5.5
0x4c	any	0	0x00	dst = (u32)(dst | src)	base32	RFC 9669, Section 4.1
0x4d	any	any	0x00	if dst & src goto +offset	base64	RFC 9669, Section 4.3
0x4e	any	any	0x00	if (u32)dst & (u32)src goto +offset	base32	RFC 9669, Section 4.3
0x4f	any	0	0x00	dst |= src	base64	RFC 9669, Section 4.1

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0x50	any	0	any	(deprecated, implementation-specific)	packet	RFC 9669, Section 5.5
0x54	0x0	0	any	dst = (u32)(dst & imm)	base32	RFC 9669, Section 4.1
0x55	0x0	any	any	if dst != imm goto +offset	base64	RFC 9669, Section 4.3
0x56	0x0	any	any	if (u32)dst != imm goto +offset	base32	RFC 9669, Section 4.3
0x57	0x0	0	any	dst &= imm	base64	RFC 9669, Section 4.1
0x5c	any	0	0x00	dst = (u32)(dst & src)	base32	RFC 9669, Section 4.1
0x5d	any	any	0x00	if dst != src goto +offset	base64	RFC 9669, Section 4.3
0x5e	any	any	0x00	if (u32)dst != (u32)src goto +offset	base32	RFC 9669, Section 4.3
0x5f	any	0	0x00	dst &= src	base64	RFC 9669, Section 4.1
0x61	any	any	0x00	dst = *(u32 *)(src + offset)	base32	RFC 9669, Section 5
0x62	0x0	any	any	*(u32 *)(dst + offset) = imm	base32	RFC 9669, Section 5
0x63	any	any	0x00	*(u32 *)(dst + offset) = src	base32	RFC 9669, Section 5
0x64	0x0	0	any	dst = (u32)(dst << imm)	base32	RFC 9669, Section 4.1
0x65	0x0	any	any	if dst s> imm goto +offset	base64	RFC 9669, Section 4.3
0x66	0x0	any	any	if (s32)dst s> (s32)imm goto +offset	base32	RFC 9669, Section 4.3

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0x67	0x0	0	any	dst <<= imm	base64	RFC 9669, Section 4.1
0x69	any	any	0x00	dst = *(u16 *)(src + offset)	base32	RFC 9669, Section 5
0x6a	0x0	any	any	*(u16 *)(dst + offset) = imm	base32	RFC 9669, Section 5
0x6b	any	any	0x00	*(u16 *)(dst + offset) = src	base32	RFC 9669, Section 5
0x6c	any	0	0x00	dst = (u32)(dst << src)	base32	RFC 9669, Section 4.1
0x6d	any	any	0x00	if dst s> src goto +offset	base64	RFC 9669, Section 4.3
0x6e	any	any	0x00	if (s32)dst s> (s32)src goto +offset	base32	RFC 9669, Section 4.3
0x6f	any	0	0x00	dst <<= src	base64	RFC 9669, Section 4.1
0x71	any	any	0x00	dst = *(u8 *)(src + offset)	base32	RFC 9669, Section 5
0x72	0x0	any	any	*(u8 *)(dst + offset) = imm	base32	RFC 9669, Section 5
0x73	any	any	0x00	*(u8 *)(dst + offset) = src	base32	RFC 9669, Section 5
0x74	0x0	0	any	dst = (u32)(dst >> imm)	base32	RFC 9669, Section 4.1
0x75	0x0	any	any	if dst s>= imm goto +offset	base64	RFC 9669, Section 4.3
0x76	0x0	any	any	if (s32)dst s>= (s32)imm goto +offset	base32	RFC 9669, Section 4.3
0x77	0x0	0	any	dst >>= imm	base64	RFC 9669, Section 4.1

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0x79	any	any	0x00	dst = *(u64 *)(src + offset)	base64	RFC 9669, Section 5
0x7a	0x0	any	any	*(u64 *)(dst + offset) = imm	base64	RFC 9669, Section 5
0x7b	any	any	0x00	*(u64 *)(dst + offset) = src	base64	RFC 9669, Section 5
0x7c	any	0	0x00	dst = (u32)(dst >> src)	base32	RFC 9669, Section 4.1
0x7d	any	any	0x00	if dst s>= src goto +offset	base64	RFC 9669, Section 4.3
0x7e	any	any	0x00	if (s32)dst s>= (s32)src goto +offset	base32	RFC 9669, Section 4.3
0x7f	any	0	0x00	dst >>= src	base64	RFC 9669, Section 4.1
0x84	0x0	0	0x00	dst = (u32)-dst	base32	RFC 9669, Section 4.1
0x85	0x0	0	any	call helper function by static ID	base32	RFC 9669, Section 4.3.1
0x85	0x1	0	any	call PC += imm	base32	RFC 9669, Section 4.3.2
0x85	0x2	0	any	call helper function by BTF ID	base32	RFC 9669, Section 4.3.1
0x87	0x0	0	0x00	dst = -dst	base64	RFC 9669, Section 4.1
0x94	0x0	0	any	dst = (u32)((imm != 0)? ((u32)dst % (u32)imm) : dst)	divmul32	RFC 9669, Section 4.1
0x94	0x0	1	any	dst = (u32)((imm != 0) ? ((s32)dst s% imm) : dst)	divmul32	RFC 9669, Section 4.1
0x95	0x0	0	0x00	return	base32	RFC 9669, Section 4.3

Opcode	src_reg	Off-set	imm	Description	Groups	Ref
0x97	0x0	0	any	dst = (imm != 0) ? (dst % (u32)imm) : dst	divmul64	RFC 9669, Section 4.1
0x97	0x0	1	any	dst = (imm != 0) ? (dst s% imm) : dst	divmul64	RFC 9669, Section 4.1
0x9c	any	0	0x00	dst = (u32)((src != 0)? ((u32)dst % (u32)src) : dst)	divmul32	RFC 9669, Section 4.1
0x9c	any	1	0x00	dst = (u32)((src != 0)? ((s32)dst s% (s32)src) :dst)	divmul32	RFC 9669, Section 4.1
0x9f	any	0	0x00	dst = (src != 0) ? (dst % src) : dst	divmul64	RFC 9669, Section 4.1
0x9f	any	1	0x00	dst = (src != 0) ? (dst s% src) : dst	divmul64	RFC 9669, Section 4.1
0xa4	0x0	0	any	dst = (u32)(dst ^ imm)	base32	RFC 9669, Section 4.1
0xa5	0x0	any	any	if dst < imm goto +offset	base64	RFC 9669, Section 4.3
0xa6	0x0	any	any	if (u32)dst < imm goto +offset	base32	RFC 9669, Section 4.3
0xa7	0x0	0	any	dst ^= imm	base64	RFC 9669, Section 4.1
0xac	any	0	0x00	dst = (u32)(dst ^ src)	base32	RFC 9669, Section 4.1
0xad	any	any	0x00	if dst < src goto +offset	base64	RFC 9669, Section 4.3
0xae	any	any	0x00	if (u32)dst < (u32)src goto +offset	base32	RFC 9669, Section 4.3
0xaf	any	0	0x00	dst ^= src	base64	RFC 9669, Section 4.1
0xb4	0x0	0	any	dst = (u32) imm	base32	RFC 9669, Section 4.1

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0xb5	0x0	any	any	if dst <= imm goto +offset	base64	RFC 9669, Section 4.3
0xb6	0x0	any	any	if (u32)dst <= imm goto +offset	base32	RFC 9669, Section 4.3
0xb7	0x0	0	any	dst = imm	base64	RFC 9669, Section 4.1
0xbc	any	0	0x00	dst = (u32) src	base32	RFC 9669, Section 4.1
0xbc	any	8	0x00	dst = (u32) (s32) (s8) src	base32	RFC 9669, Section 4.1
0xbc	any	16	0x00	dst = (u32) (s32) (s16) src	base32	RFC 9669, Section 4.1
0xbd	any	any	0x00	if dst <= src goto +offset	base64	RFC 9669, Section 4.3
0xbe	any	any	0x00	if (u32)dst <= (u32)src goto +offset	base32	RFC 9669, Section 4.3
0xbf	any	0	0x00	dst = src	base64	RFC 9669, Section 4.1
0xbf	any	8	0x00	dst = (s64) (s8) src	base64	RFC 9669, Section 4.1
0xbf	any	16	0x00	dst = (s64) (s16) src	base64	RFC 9669, Section 4.1
0xbf	any	32	0x00	dst = (s64) (s32) src	base64	RFC 9669, Section 4.1
0xc3	any	any	0x00	lock *(u32 *)(dst + offset) += src	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0x01	src = atomic_fetch_add_32((u32 *)(dst + offset), src)	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0x40	lock *(u32 *)(dst + offset) |= src	atomic32	RFC 9669, Section 5.3

Opcode	src_reg	Off-set	imm	Description	Groups	Ref
0xc3	any	any	0x41	src = atomic_fetch_or_32((u32 *)(dst + offset), src)	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0x50	lock *(u32 *)(dst + offset) &= src	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0x51	src = atomic_fetch_and_32((u32 *)(dst + offset), src)	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0xa0	lock *(u32 *)(dst + offset) ^= src	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0xa1	src = atomic_fetch_xor_32((u32 *)(dst + offset), src)	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0xe1	src = xchg_32((u32 *)(dst + offset), src)	atomic32	RFC 9669, Section 5.3
0xc3	any	any	0xf1	r0 = cmpxchg_32((u32 *)(dst + offset), r0, src)	atomic32	RFC 9669, Section 5.3
0xc4	0x0	0	any	dst = (u32)(dst s>> imm)	base32	RFC 9669, Section 4.1
0xc5	0x0	any	any	if dst s< imm goto +offset	base64	RFC 9669, Section 4.3
0xc6	0x0	any	any	if (s32)dst s< (s32)imm goto +offset	base32	RFC 9669, Section 4.3
0xc7	0x0	0	any	dst s>>= imm	base64	RFC 9669, Section 4.1
0xcc	any	0	0x00	dst = (u32)(dst s>> src)	base32	RFC 9669, Section 4.1
0xcd	any	any	0x00	if dst s< src goto +offset	base64	RFC 9669, Section 4.3
0xce	any	any	0x00	if (s32)dst s< (s32)src goto +offset	base32	RFC 9669, Section 4.3

Opcode	src_reg	Off-set	imm	Description	Groups	Ref
0xcf	any	0	0x00	dst s>>= src	base64	RFC 9669, Section 4.1
0xd4	0x0	0	0x10	dst = htole16(dst)	base32	RFC 9669, Section 4.2
0xd4	0x0	0	0x20	dst = htole32(dst)	base32	RFC 9669, Section 4.2
0xd4	0x0	0	0x40	dst = htole64(dst)	base64	RFC 9669, Section 4.2
0xd5	0x0	any	any	if dst s<= imm goto +offset	base64	RFC 9669, Section 4.3
0xd6	0x0	any	any	if (s32)dst s<= (s32)imm goto +offset	base32	RFC 9669, Section 4.3
0xd7	0x0	0	0x10	dst = bswap16(dst)	base32	RFC 9669, Section 4.2
0xd7	0x0	0	0x20	dst = bswap32(dst)	base32	RFC 9669, Section 4.2
0xd7	0x0	0	0x40	dst = bswap64(dst)	base64	RFC 9669, Section 4.2
0xdb	any	any	0x00	lock *(u64 *)(dst + offset) += src	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0x01	src = atomic_fetch_add_64((u64 *)(dst + offset), src)	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0x40	lock *(u64 *)(dst + offset) |= src	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0x41	src = atomic_fetch_or_64((u64 *)(dst + offset), src)	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0x50	lock *(u64 *)(dst + offset) &= src	atomic64	RFC 9669, Section 5.3

Opcode	src_reg	Offset	imm	Description	Groups	Ref
0xdb	any	any	0x51	src = atomic_fetch_and_64((u64 *)(dst + offset), src)	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0xa0	lock *(u64 *)(dst + offset) ^= src	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0xa1	src = atomic_fetch_xor_64((u64 *)(dst + offset), src)	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0xe1	src = xchg_64((u64 *)(dst + offset), src)	atomic64	RFC 9669, Section 5.3
0xdb	any	any	0xf1	r0 = cmpxchg_64((u64 *)(dst + offset), r0, src)	atomic64	RFC 9669, Section 5.3
0xdc	0x0	0	0x10	dst = htobe16(dst)	base32	RFC 9669, Section 4.2
0xdc	0x0	0	0x20	dst = htobe32(dst)	base32	RFC 9669, Section 4.2
0xdc	0x0	0	0x40	dst = htobe64(dst)	base64	RFC 9669, Section 4.2
0xdd	any	any	0x00	if dst s<= src goto +offset	base64	RFC 9669, Section 4.3
0xde	any	any	0x00	if (s32)dst s<= (s32)src goto +offset	base32	RFC 9669, Section 4.3

Table 18: Initial BPF Instruction Set Values

Acknowledgements

This draft was generated from instruction-set.rst in the Linux kernel repository, to which a number of other individuals have authored contributions over time, including Akhil Raj, Alexei Starovoitov, Brendan Jackman, Christoph Hellwig, Daniel Borkmann, Ilya Leoshkevich, Jiong Wang, Jose E. Marchesi, Kosuke Fujimoto, Shahab Vahedi, Tiezhu Yang, Will Hawkins, and Zheng Yejian, with review and suggestions by many others including Alan Jowett, Andrii Nakryiko, David Vernet, Jim Harris, Quentin Monnet, Song Liu, Shung-Hsi Yu, Stanislav Fomichev, Watson Ladd, and Yonghong Song.

Author's Address

Dave Thaler (EDITOR)
Redmond, WA 98052
United States of America
Email: dave.thaler.ietf@gmail.com