Contents

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 an implementation-specific amount (e.g., 512 bytes) of stack space.

1.1 Documentation conventions

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

1.2 Registers and calling convention

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

The eBPF calling convention is defined as:

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

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

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

1.3 Instruction encoding

An eBPF program is a sequence of instructions.

eBPF has two instruction encodings:

- the basic instruction encoding, which uses 64 bits to encode an instruction
- the wide instruction encoding, which appends a second 64-bit immediate (i.e., constant) value after the basic instruction for a total of 128 bits.

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

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

imm
: signed integer immediate value

offset
: signed integer offset used with pointer arithmetic

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

dst_reg
: destination register number (0-10)

opcode
: operation to perform

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

For example:

```
opcode                  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 must be set to zero.

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

This is depicted in the following figure:

```
                                '--------------'
                               pseudo instruction
```

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

imm64 = (next_imm << 32) | imm

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

1.3.1 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_LDX	0x01	load into register operations	Load and store instructions
BPF_ST	0x02	store from immediate operations	Load and store instructions
BPF_STX	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_reg' 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 'code' field encodes the operation as below, where 'src' and 'dst' refer to the values of the source and destination registers, respectively.

code	value	description
BPF_ADD	0x00	dst += src
BPF_SUB	0x10	dst -= src
BPF_MUL	0x20	dst *= src
BPF_DIV	0x30	dst = (src != 0) ? (dst / src) : 0
BPF_OR	0x40	dst |= src
BPF_AND	0x50	dst &= src
BPF_LSH	0x60	dst <<= src
BPF_RSH	0x70	dst >>= src
BPF_NEG	0x80	dst = ~src
BPF_MOD	0x90	dst = (src != 0) ? (dst % src) : dst
BPF_XOR	0xa0	dst ^= src
BPF_MOV	0xb0	dst = src
BPF_ARSH	0xc0	sign extending shift right
BPF_END	0xd0	byte swap operations (see Byte swap instructions below)

Underflow and overflow are allowed during arithmetic operations, meaning the 64-bit or 32-bit value will wrap. If 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, 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.

Examples:

BPF_ADD | BPF_X | BPF_ALU (0x0c) means:

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

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

BPF_ADD | BPF_X | BPF_ALU64 (0x0f) means:

```
dst = dst + src
```

BPF_XOR | BPF_K | BPF_ALU (0xa4) means:

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

BPF_XOR | BPF_K | BPF_ALU64 (0xa7) means:

```
dst = dst ^ imm32
```

Also note that the division and modulo operations are unsigned. Thus, for BPF_ALU, 'imm' is first interpreted as an unsigned 32-bit value, whereas for BPF_ALU64, 'imm' is first sign extended to 64 bits and the result interpreted as an unsigned 64-bit value. There are no instructions for signed division or modulo.

1.4.1.1 Byte swap instructions

The byte swap instructions use an instruction class of BPF_ALU and a 4-bit 'code' field of BPF_END.

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

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

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

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

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

where

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

1.4.2 Jump instructions

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

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

code	value	src	description	notes
BPF_JA	0x0	0x0	PC += offset	BPF_JMP only
BPF_JEQ	0x1	any	PC += offset if dst == src	
BPF_JGT	0x2	any	PC += offset if dst > src	unsigned
BPF_JGE	0x3	any	PC += offset if dst >= src	unsigned
BPF_JSET	0x4	any	PC += offset if dst & src	
BPF_JNE	0x5	any	PC += offset if dst != src	
BPF_JSGT	0x6	any	PC += offset if dst > src	signed
BPF_JSGE	0x7	any	PC += offset if dst >= src	signed
BPF_CALL	0x8	0x0	call helper function imm	see Helper functions
BPF_CALL	0x8	0x1	call PC += offset	see eBPF functions
BPF_CALL	0x8	0x2	call runtime function imm	see Runtime functions
BPF_EXIT	0x9	0x0	return	BPF_JMP only
BPF_JLT	0xa	any	PC += offset if dst < src	unsigned
BPF_JLE	0xb	any	PC += offset if dst <= src	unsigned

BPF_JSLT	0xc	any	PC += offset if dst < src	signed
BPF_JSLE	0xd	any	PC += offset if dst <= src	signed

Example:

`BPF_JSGE | BPF_X | BPF_JMP32` (0x7e) means:

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

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

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.

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

1.4.2.2 Runtime functions

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

1.4.2.3 eBPF functions

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

1.5 Load and store instructions

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

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

mode

one of:

mode modifier	value	description	reference
BPF_IMM	0x00	64-bit immediate instructions	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.

`BPF_MEM | <size> | BPF_STX` means:

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

`BPF_MEM | <size> | BPF_ST` means:

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

`BPF_MEM | <size> | BPF_LDX` means:

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

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

1.5.2 Atomic operations

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

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

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

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

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

imm	value	description
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` (0xc3) with 'imm' = BPF_ADD means:

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

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

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

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

imm	value	description
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	dst = imm64	integer	integer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x1	dst = map_by_fd(imm)	map fd	map
BPF_IMM | BPF_DW | BPF_LD	0x18	0x2	dst = mva(map_by_fd(imm)) + next_imm	map fd	data pointer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x3	dst = variable_addr(imm)	variable id	data pointer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x4	dst = code_addr(imm)	integer	code pointer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x5	dst = map_by_idx(imm)	map index	map

BPF_IMM | BPF_DW | BPF_LD	0x18	0x6	dst = mva(map_by_idx(imm)) + next_imm	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.