eBPF Instruction Set

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

R0 - R5 are scratch registers and eBPF programs needs to spill/fill them if necessary across calls.

Instruction encoding

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 value (imm64) after the basic instruction for a total of 128 bits.

The basic instruction encoding looks as follows:

32 bits (MSB)	16 bits	4 bits	4 bits	8 bits (LSB)
immediate	offset	source register	destination register	opcode

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

Instruction classes

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

class	value	description
BPF_LD	0x00	non-standard load operations
BPF_LDX	0x01	load into register operations
BPF_ST	0x02	store from immediate operations
BPF_STX	0x03	store from register operations
BPF_ALU	0x04	32-bit arithmetic operations
BPF_JMP	0x05	64-bit jump operations
BPF_JMP32	0x06	32-bit jump operations
BPF_ALU64	0x07	64-bit arithmetic operations

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)
operation code	source	instruction class

The 4th bit encodes the source operand:

source	value	description
BPF_K	0x00	use 32-bit immediate as source operand
BPF_X	0x08	use 'src_reg' register as source operand

The four MSB bits store the operation code.

Arithmetic instructions

BPF_ALU uses 32-bit wide operands while BPF_ALU64 uses 64-bit wide operands for otherwise identical operations. The code field encodes the operation as below:

code	value	description
BPF_ADD	0x00	dst += src
BPF_SUB	0x10	dst -= src
BPF_MUL	0x20	dst *= src
BPF_DIV	0x30	dst /= src
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
BPF_XOR	0xa0	dst ^= src
BPF_MOV	0xb0	dst = src
BPF_ARSH	0xc0	sign extending shift right
BPF_END	0xd0	byte swap operations (see separate section below)

BPF_ADD | BPF_X | BPF_ALU means:

```
dst_reg = (u32) dst_reg + (u32) src_reg;
```

BPF_ADD | BPF_X | BPF_ALU64 means:

```
dst_reg = dst_reg + src_reg
```

BPF_XOR | BPF_K | BPF_ALU means:

```
src_reg = (u32) src_reg ^ (u32) imm32
```

BPF_XOR | BPF_K | BPF_ALU64 means:

```
src_reg = src_reg ^ imm32
```

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.

The 1-bit source operand field in the opcode is used to to select what byte order the operation convert 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.

Examples:

`BPF_ALU | BPF_TO_LE | BPF_END` with imm = 16 means:

```
dst_reg = htole16(dst_reg)
```

`BPF_ALU | BPF_TO_BE | BPF_END` with imm = 64 means:

```
dst_reg = htobe64(dst_reg)
```

`BPF_FROM_LE` and `BPF_FROM_BE` exist as aliases for `BPF_TO_LE` and `BPF_TO_BE` respectively.

Jump instructions

BPF_JMP32 uses 32-bit wide operands while BPF_JMP uses 64-bit wide operands for otherwise identical operations. The code field encodes the operation as below:

code	value	description	notes
BPF_JA	0x00	PC += off	BPF_JMP only
BPF_JEQ	0x10	PC += off if dst == src	
BPF_JGT	0x20	PC += off if dst > src	unsigned
BPF_JGE	0x30	PC += off if dst >= src	unsigned
BPF_JSET	0x40	PC += off if dst & src	
BPF_JNE	0x50	PC += off if dst != src	
BPF_JSGT	0x60	PC += off if dst > src	signed
BPF_JSGE	0x70	PC += off if dst >= src	signed
BPF_CALL	0x80	function call	
BPF_EXIT	0x90	function / program return	BPF_JMP only

BPF_JLT	0xa0	PC += off if dst < src	unsigned
BPF_JLE	0xb0	PC += off if dst <= src	unsigned
BPF_JSLT	0xc0	PC += off if dst < src	signed
BPF_JSLE	0xd0	PC += off if dst <= src	signed

The eBPF program needs to store the return value into register R0 before doing a BPF_EXIT.

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 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)

The mode modifier is one of:

mode modifier	value	description
BPF_IMM	0x00	64-bit immediate instructions
BPF_ABS	0x20	legacy BPF packet access (absolute)
BPF_IND	0x40	legacy BPF packet access (indirect)
BPF_MEM	0x60	regular load and store operations
BPF_ATOMIC	0xc0	atomic operations

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_reg + off) = src_reg
```

`BPF_MEM | <size> | BPF_ST` means:

```
*(size *) (dst_reg + off) = imm32
```

`BPF_MEM | <size> | BPF_LDX` means:

```
dst_reg = *(size *) (src_reg + off)
```

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

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` for 32-bit operations
- `BPF_ATOMIC | BPF_DW | BPF_STX` for 64-bit operations
- 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_reg + off16) += src_reg
```

`BPF_ATOMIC | BPF_DW | BPF_STX` with imm = BPF ADD means:

```
*(u64 *)(dst_reg + off16) += src_reg
```

`BPF_XADD` is a deprecated name for `BPF_ATOMIC | BPF_ADD`.

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_reg` with the value that was in memory before it was modified.

The `BPF_XCHG` operation atomically exchanges `src_reg` with the value addressed by `dst_reg + off`.

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

Clang can generate atomic instructions by default when `-mcpu=v3` is enabled. If a lower version for `-mcpu` is set, the only atomic instruction Clang can generate is `BPF_ADD` *without* `BPF_FETCH`. If you need to enable the atomics features, while keeping a lower `-mcpu` version, you can use `-Xclang -target-feature -Xclang +alu32`.

64-bit immediate instructions

Instructions with the `BPF_IMM` mode modifier use the wide instruction encoding for an extra imm64 value.

There is currently only one such instruction.

`BPF_LD | BPF_DW | BPF_IMM` means:

```
dst_reg = imm64
```

Legacy BPF Packet access instructions

eBPF has special instructions for access to packet data that have been carried over from classic BPF to retain the performance of legacy socket filters running in the eBPF interpreter.

The instructions come in two forms: `BPF_ABS | <size> | BPF_LD` and `BPF_IND | <size> | BPF_LD`.

These instructions are used to access packet data and can only be used when the program context is a pointer to networking packet. `BPF_ABS` accesses packet data at an absolute offset specified by the immediate data and `BPF_IND` access packet data at an offset that includes the value of a register in addition to the immediate data.

These instructions have seven implicit operands:

- Register R6 is an implicit input that must contain pointer to a struct sk_buff.
- Register R0 is an implicit output which contains the data fetched from the packet.
- Registers R1-R5 are scratch registers that are clobbered after a call to `BPF_ABS | BPF_LD` or `BPF_IND | BPF_LD` instructions.

These instructions have an implicit program exit condition as well. When an eBPF program is trying to access the data beyond the packet boundary, the program execution will be aborted.

`BPF_ABS | BPF_W | BPF_LD` means:

```
R0 = ntohl(*(u32 *) (((struct sk_buff *) R6)->data + imm32))
```

`BPF_IND | BPF_W | BPF_LD` means:

```
R0 = ntohl(*(u32 *) (((struct sk_buff *) R6)->data + src_reg + imm32))
```