BPF指令集架构简介

> *参考:*
>
> *BPF Standardization — The Linux Kernel documentation*
>
> */tools/include/uapi/linux/bpf.h Linux kernel source tree*
>
> */tools/include/uapi/linux/bpf_common.h Linux kernel source tree*

这篇文章介绍了BPF寄存器和BPF指令。

零、 BPF寄存器

BPF有十个通用寄存器和一个帧指针寄存器，所有寄存器都为64bit。

寄存器调用约定如下：

R0：函数调用的返回值和BPF程序的退出值

R1-R5：函数调用的参数

R6-R9：被调用函数的寄存器的值在发生函数调用时将保存下来

R10：帧指针寄存器

R0-R5为易失寄存器，因此在函数调用时应得到保存

BPF 程序在执行 EXIT 之前需要将返回值存储到寄存器 R0 中

在源码中：

```
58 /* Register numbers */
59 enum {
60         BPF_REG_0 = 0,
61         BPF_REG_1,
62         BPF_REG_2,
63         BPF_REG_3,
64         BPF_REG_4,
65         BPF_REG_5,
66         BPF_REG_6,
67         BPF_REG_7,
68         BPF_REG_8,
69         BPF_REG_9,
70         BPF_REG_10,
71         __MAX_BPF_REG,
72 };
73
74 /* BPF has 10 general purpose 64-bit registers and stack frame. */
```

一、一致性组（Conformance groups）

文档中规定了一些一致性组，OS必须要支持base32一致性组，选择性支持其他一致性组（类似RVA Profile）。支持一个一致性组意味着要支持该组中所有指令。

文档中规定了如下的一致性组：

1. base32：文档中没有额外说明的指令即属于base32
2. base64：包括base32，外加额外说明的指令
3. atomics32：包括32位原子操作指令
4. atomics64：包括atomics32，外加64位原子操作指令
5. divmul32：包括32位的乘法、除法、取余指令
6. divmul64：包括divmul32，外加64位的乘法、除法、模指令
7. packet：弃用的数据包访问指令

二、指令编码格式

BPF有两种指令编码格式：

1. 基本指令编码，一条指令为64位
2. 宽指令编码，一条指令为128位

2.1 基本指令编码

基本指令按如下格式编码：

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

- opcode：要进行的操作，按如下方式编码：

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

  - class：指令的类
  - specific：这些位的格式因指令的类而异
- regs：源寄存器和目的寄存器数

  小段序按如下方式编码：

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

  大端序按如下方式编码：

  ```
  +-+-+-+-+-+-+-+-+
  |dst_reg|src_reg|
  +-+-+-+-+-+-+-+-+
  ```
- offset：与指针运算一起使用的有符号整数偏移量
- imm：有符号整数立即数

需要注意的是，imm和offset中的值跟随主机的字节序

在源码中：

```
77 struct bpf_insn {
78         __u8    code;           /* opcode */
79         __u8    dst_reg:4;      /* dest register */
80         __u8    src_reg:4;      /* source register */
81         __s16   off;            /* signed offset */
82         __s32   imm;            /* signed immediate constant */
83 };
```

2.2 宽指令编码

宽指令按如下格式编码：

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

- opcode：要进行的操作
- regs：源寄存器和目的寄存器
- offset：与指针运算一起使用的有符号整数偏移量
- imm：有符号整数立即数
- reserved：全为0
- next_imm：第二个有符号整数立即数

未在源码中找到，疑似未支持

2.3 指令类

所有的指令类如下：

Instruction class

class	value	description	reference
LD	0x0	non-standard load operations	Load and store instructions
LDX	0x1	load into register operations	Load and store instructions
ST	0x2	store from immediate operations	Load and store instructions
STX	0x3	store from register operations	Load and store instructions
ALU	0x4	32-bit arithmetic operations	Arithmetic and jump instructions
JMP	0x5	64-bit jump operations	Arithmetic and jump instructions
JMP32	0x6	32-bit jump operations	Arithmetic and jump instructions
ALU64	0x7	64-bit arithmetic operations	Arithmetic and jump instructions

在源码中：

```
 5 /* Instruction classes */
 6 #define BPF_CLASS(code) ((code) & 0x07)
 7 #define         BPF_LD          0x00
 8 #define         BPF_LDX         0x01
 9 #define         BPF_ST          0x02
10 #define         BPF_STX         0x03
11 #define         BPF_ALU         0x04
12 #define         BPF_JMP         0x05
13 #define         BPF_RET         0x06
14 #define         BPF_MISC        0x07
```

BPF_RET和BPF_MISC应该是被舍弃掉，但在源码中仍存在，疑惑

```
16 /* instruction classes */
17 #define BPF_JMP32       0x06    /* jmp mode in word width */
18 #define BPF_ALU64       0x07    /* alu mode in double word width */
```

三、算数和跳转指令

对于算术和跳转指令（ALU、ALU64、JMP、JMP32），opcode域划分为如下三个部分：

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

- code：操作码，根据指令类而变化
- s(source)：源操作数域，除非特别说明，为下面两者之一

Source operand location

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

在源码中：

```
49 #define BPF_SRC(code)   ((code) & 0x08)
50 #define         BPF_K           0x00
51 #define         BPF_X           0x08
```

- class：指令类

3.1 算术指令

ALU使用32位操作数，ALU使用64位操作数。ALU64指令属于base64一致性组。code域定义的指令如下所示：

Arithmetic instructions

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
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 dst >>= (src & mask)
END	0xd	0	byte swap operations (see Byte swap instructions below)

在源码中：

```
30 /* alu/jmp fields */
31 #define BPF_OP(code)    ((code) & 0xf0)
32 #define         BPF_ADD         0x00
33 #define         BPF_SUB         0x10
34 #define         BPF_MUL         0x20
35 #define         BPF_DIV         0x30
36 #define         BPF_OR          0x40
37 #define         BPF_AND         0x50
38 #define         BPF_LSH         0x60
39 #define         BPF_RSH         0x70
40 #define         BPF_NEG         0x80
41 #define         BPF_MOD         0x90
42 #define         BPF_XOR         0xa0
```

```
26 /* alu/jmp fields */
27 #define BPF_MOV         0xb0    /* mov reg to reg */
28 #define BPF_ARSH        0xc0    /* sign extending arithmetic shift right */
29
30 /* change endianness of a register */
31 #define BPF_END         0xd0    /* flags for endianness conversion: */
32 #define BPF_TO_LE       0x00    /* convert to little-endian */
33 #define BPF_TO_BE       0x08    /* convert to big-endian */
34 #define BPF_FROM_LE     BPF_TO_LE
35 #define BPF_FROM_BE     BPF_TO_BE
```

可以看到，code域相同的指令在源码中未重复列出，因为它们将在offset域做区分

算术指令中产生的溢出是被允许的，因此此值会发生回绕。如果BPF程序导致被0除的操作发生，目的寄存器会被置0。如果产生了模0操作，对于ALU64，目的寄存器的值不变，对于ALU，目的寄存器的高32位被置0。

对于ALU指令，源寄存器和目的寄存器的高32位被置为0。

需要注意的是，大部分算术指令将offset置为0，只有三个指令（SDIV、SMOD、MOVSX）有非零的的offset。

除法和模指令同时支持有符号和无符号数。

对于无符号操作（DIV和MOD），对于ALU，imm为32位无符号数。对于ALU64，imm先从32位符号扩展到64位，然后解释为64位有符号数。

需要注意的是，被除数或除数为负数时，有符号模操作有着不同的定义，具体实现往往因语言而异。本规范要求有符号的模操作必须使用截断除法。

MOVSX指令执行符号扩展的移动操作。`{MOVSX, X, ALU}`将8bit、16bit操作数符号扩展到32bit，高32位保持为0。`{MOVSX, X, ALU64}`将8bit、16bit、32bit操作数符号扩展到64bit。MOVSX的操作数为寄存器。

NEG指令的操作数为立即数。

对于64位操作，移位指令的掩码为0x3F，对32为操作，则为0x1F。

3.2 字节交换指令

字节交换指令的指令类为ALU和ALU64，code域为END。

字节交换指令只对目标寄存器进行操作，不使用源寄存器或立即数。

对于ALU，1bit的code域用来选择操作数要转换的字节序（如下图）。对于ALU，1bit的code域必须置为0。

Byte swap instructions

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

imm域编码为交换操作的宽度，支持以下宽度：16、32和64bit。64位操作属于base64一致性组，其他宽度指令属于base32一致性组。

3.3 跳转指令

JMP32操作数为32位，属于base32一致性组，JMP操作数为64位，属于base64一致性组。code域编码格式如下：

Jump instructions

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
CALL	0x8	0x1	call PC += imm	{CALL, K, JMP} only, see Program-local functions
CALL	0x8	0x2	call helper function by BTF ID	{CALL, K, JMP} only, see Helper functions
EXIT	0x9	0x0	return	{CALL, K, JMP} only
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

在源码中：

```
44 #define        BPF_JA          0x00
45 #define        BPF_JEQ         0x10
46 #define        BPF_JGT         0x20
47 #define        BPF_JGE         0x30
48 #define        BPF_JSET        0x40
```

```
37 /* jmp encodings */
38 #define BPF_JNE         0x50    /* jump != */
39 #define BPF_JLT         0xa0    /* LT is unsigned, '<' */
40 #define BPF_JLE         0xb0    /* LE is unsigned, '<=' */
41 #define BPF_JSGT        0x60    /* SGT is signed '>', GT in x86 */
42 #define BPF_JSGE        0x70    /* SGE is signed '>=', GE in x86 */
43 #define BPF_JSLT        0xc0    /* SLT is signed, '<' */
44 #define BPF_JSLE        0xd0    /* SLE is signed, '<=' */
45 #define BPF_JCOND       0xe0    /* conditional pseudo jumps: may_goto, goto_or_n
   op */
46 #define BPF_CALL        0x80    /* function call */
47 #define BPF_EXIT        0x90    /* function return */
```

增量的偏移量offset以64bit为单位。

需要注意的是有两种JA指令。JMP类允许使用offset域的16bit偏移量，而JMP32类允许使用imm的32bit偏移量。大于16bit的条件跳转会被转换为小于16bit的条件跳转加32bit的非条件跳转。

CALL和JA指令属于base32一致性组。

四、加载存储指令

对于加载和存储指令（LD、LDX、ST和STX），8bit opcode域编码格式如下：

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

- mode为下列之一：

Mode modifier

mode modifier	value	description	reference
IMM	0	64-bit immediate instructions	64-bit immediate instructions
ABS	1	legacy BPF packet access (absolute)	Legacy BPF Packet access instructions
IND	2	legacy BPF packet access (indirect)	Legacy BPF Packet access instructions
MEM	3	regular load and store operations	Regular load and store operations
MEMSX	4	sign-extension load operations	Sign-extension load operations
ATOMIC	6	atomic operations	Atomic operations

在源码中：

```
22 #define BPF_MODE(code)  ((code) & 0xe0)
23 #define         BPF_IMM         0x00
24 #define         BPF_ABS         0x20
25 #define         BPF_IND         0x40
26 #define         BPF_MEM         0x60
27 #define         BPF_LEN         0x80
28 #define         BPF_MSH         0xa0
```

BPF_LEN和BPF_MSH指令似乎已经遗弃

```
22 #define BPF_MEMSX       0x80    /* load with sign extension */
23 #define BPF_ATOMIC      0xc0    /* atomic memory ops - op type in immediate */
```

- size为下列之一：

Size modifier

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

在源码中：

```
#define         BPF_W           0x00 /* 32-bit */
#define         BPF_H           0x08 /* 16-bit */
#define         BPF_B           0x10 /*  8-bit */
```

```
21 #define BPF_DW          0x18    /* double word (64-bit) */
```

- class：指令类

4.1 常规加载存储指令

mode域为MEM时，为常规加载存储指令，该指令用于在寄存器和内存直接传输数据。STX的源操作数为寄存器中的值。

4.2 符号扩展加载操作

mode域为MEMSX时，为符号扩展常规加载存储指令，该指令用于在寄存器和内存直接传输数据。

4.3 原子操作

原子操作对一块内存进行操作，并且不能被其他对该内存区域的操作所中断。所有原子操作mode域编码为ATOMIC

`{ATOMIC, W, STX}`为32位操作，属于atomic32一致性组

`{ATOMIC, DW, STX}`为64位操作，属于atomic64一致性组

不支持8bit和16bit宽的原子操作

imm域用做编码实际的原子操作：

Simple atomic operations

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

除了上述简单的原子操作，还有一个修饰符和两个复杂原子操作：

Complex atomic operations

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

在源码中：

```
49 /* atomic op type fields (stored in immediate) */
50 #define BPF_FETCH       0x01    /* not an opcode on its own, used to build other
   s */
51 #define BPF_XCHG        (0xe0 | BPF_FETCH)      /* atomic exchange */
52 #define BPF_CMPXCHG     (0xf0 | BPF_FETCH)      /* atomic compare-and-write */
```

对于简单原子操作，FETCH是可选的，对于复杂原子操作，则是一直设置的。

XCHG操作原子交换src寄存器和dst+offset地址中的值

CMPXCHG操作原子比较dst+offset地址中的值和R0寄存器中的值，如果它们相等，将会用src寄存器中的值替换dst+offset地址中的值。

4.4 64bit立即数指令

mode域使用IMM编码使用宽指令编码，并且使用基本指令中的src_reg域来保存操作子类型。

下表定义了一系列使用子类型的`{IMM, DW, LD}`指令

64-bit immediate instructions

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
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

其中

- map_by_fd(imm)意味着将32位文件描述符转换为map地址
- map_by_idx(imm)意味着将32位索引转换为map地址
- map_val(map)获取给定map中第一个值的地址
- var_addr（imm）获取具有给定ID的平台变量的地址
- code_addr(imm) 获取（64 位）指令中指定相对偏移处的指令地址
- 反汇编程序可以使用“imm type”进行显示
- “dst 类型”可用于验证和 JIT 编译

4.4.1 Maps

在一些平台上，maps是BPF程序可以访问的共享内存。

如果平台支持，每个map将会有一个文件描述符，而map_by_fd(imm)意味着通过特定的文件描述符得到map。每个BPF程序在加载的时候也可以定义为能够使用一组跟程序相关的map，map_by_idx(imm)意味着在这个BPF程序相关的一组maps中，通过索引得到map。

4.4.2 平台变量

平台变量是内存区域，由整数id确定，在运行时暴露出来，在某些平台上可以被BPF程序访问到。var_addr(imm)操作的意思是获取给定id标识的内存区域的地址。