x86 Instruction Encoding

...and the nasty hacks we do in the kernel

Borislav Petkov
SUSE Labs
bp@suse.de

TOC

- x86 Instruction Encoding
- Funky kernel stuff
  - Alternatives, i.e. runtime instruction patching
  - Exception tables
  - Jump labels

Some history + timeline

- Rough initial development line
  - 4004: 1971, Busycom calc
  - 8008: 1972, Intel's first 8-bit CPU (insn set by Datapoint, CRT terminals)
  - 8080: 1974, extended insn set, asm src compat with 8008
  - 8085: 1977, depletion load NMOS with single power supply
  - 8086: 1978, 16-bit CPU with 16-bit external data bus
  - 8088: 16-bit, 8-bit ext data bus (16 bit IO split into two 8-bit cycles) → IBM PC, Stephen Morse called it the castrated version of 8086 :-)
  - ...

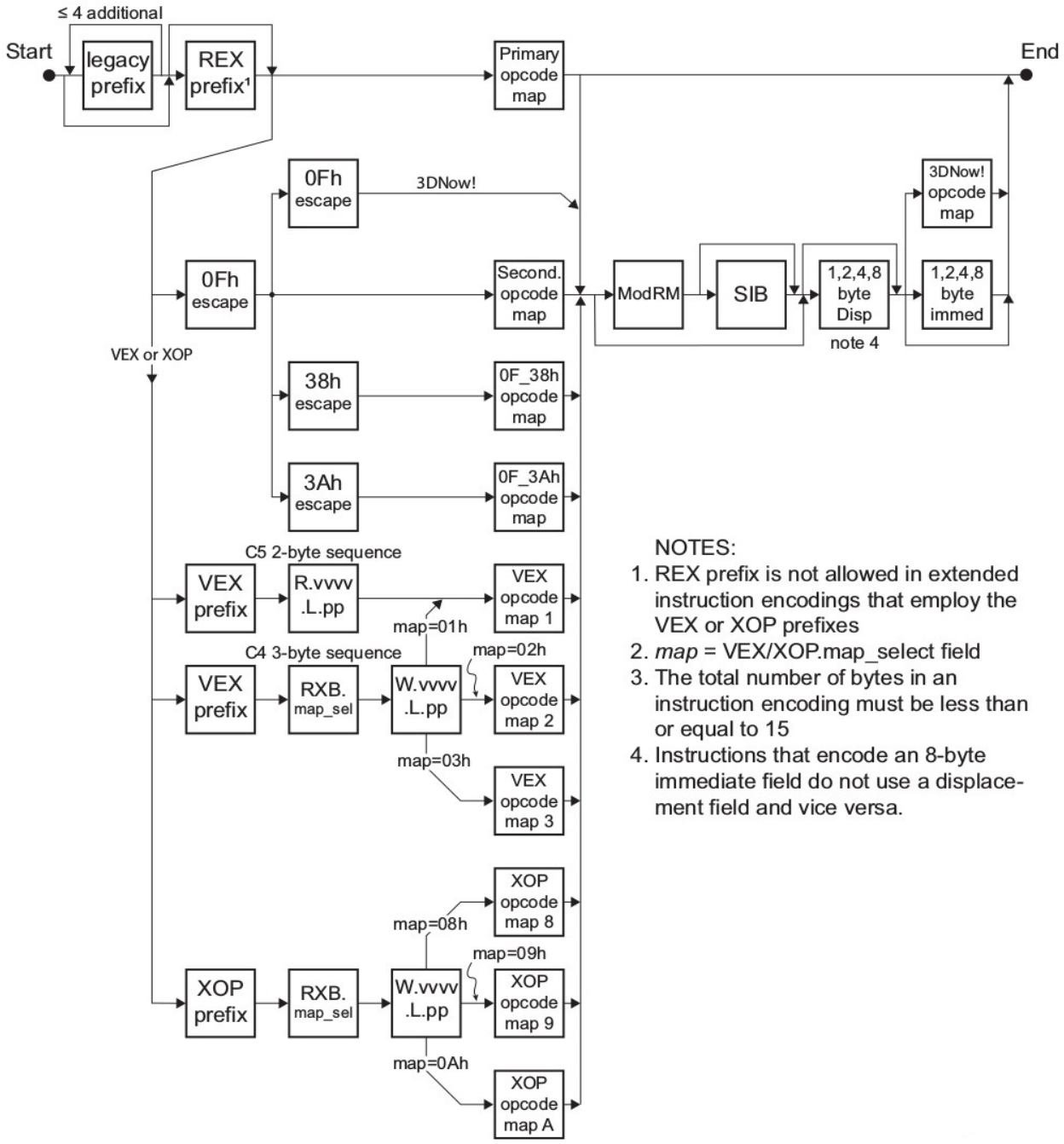
x86 ISA

- Insn set backwards-compatible to Intel 8086
- A hybrid CISC
- Extension of 8-bit 8008 and 8080 arches
- Little endian byte order
- Variable length, max 15 bytes long

```
4004ba:       66 66 66 66 66 66 66    data16 data16 data16 data16 data16 data16 data16 data16 mov $0x0,%rbx
4004c1:       66 48 c7 c3 00 00 00
4004c8:       00
```

That one still executes ok. One more prefix and:

```
traps: a[5157] general protection ip:4004ba sp:7fffafa5aab0 error:0 in
a[400000+1000]
```



NOTES:

1. REX prefix is not allowed in extended instruction encodings that employ the VEX or XOP prefixes
2. *map* = VEX/XOP.map_select field
3. The total number of bytes in an instruction encoding must be less than or equal to 15
4. Instructions that encode an 8-byte immediate field do not use a displacement field and vice versa.

Intel's version's nice too

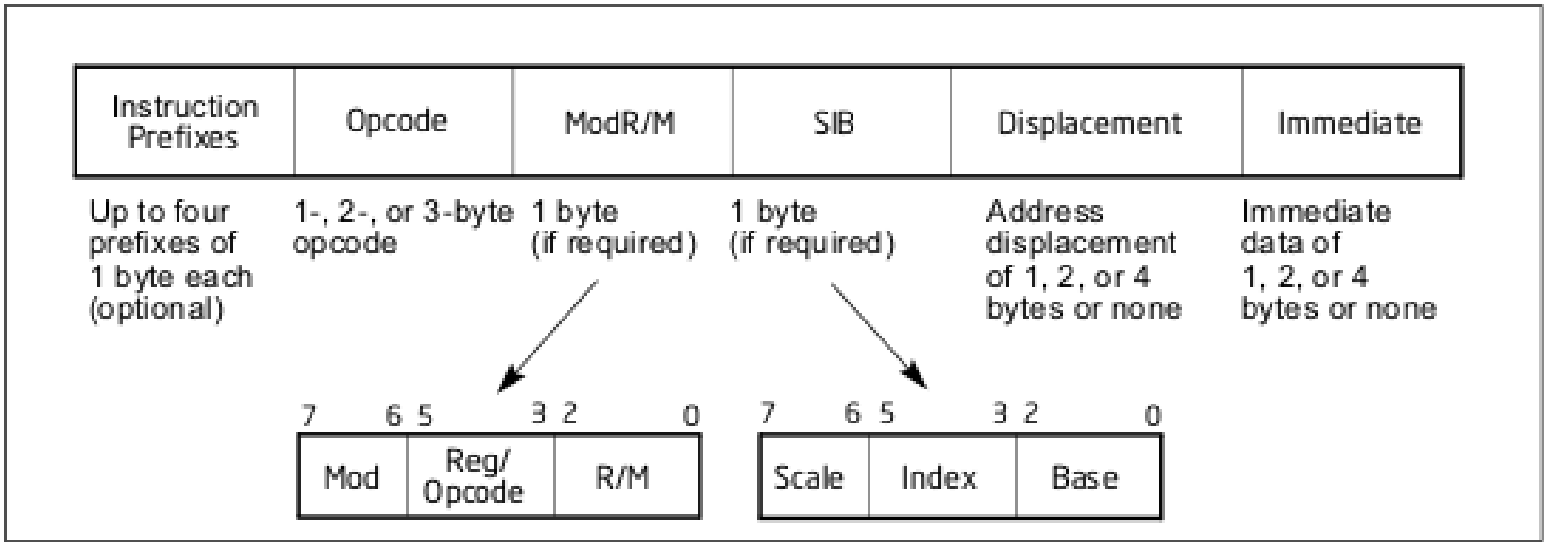


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

(Almost) General Purpose Registers

- rAX: Accumulator
- rCX: Count (LOOP, REP <string insn>, MSR reg)
- rDX: Double precision (extend rAX)
- rBX: Base idx
- rSI: src idx (MOVS,...)
- rDI: dst idx
- rBP: Base Pointer of current frame
- rSP: Stack Pointer
- r8-r15: for lack of imagination

Prefixes

- Instruction modifiers
  - Legacy
    - LOCK: 0F
    - REPNE/REPNZ: F2, REPE/REPZ: F3
    - Operand-size override: 66 (use selects non-default size, doh)
    - Segment-override: 36, 26, 64, 65, 2E, 3E (last two taken/not taken branch hints with Jcc on Intel – ignored on AMD)
    - Address-size override: 67
  - REX (40-4f) precede opcode or legacy pfx
    - 8 additional regs (%r8-%r15), size extensions
- Encoding escapes: different encoding syntax
  - VEX/XOP/EVEX/MVEX...

Opcode

- Single byte denoting basic operation; opcode is mandatory
- A byte => 256 entry primary opcode map; but we have more instructions
- Escape sequences select alternate opcode maps
  - Legacy escapes: 0f [0f, 38, 3a]
    - Thus [0f <opcode>] is a two-byte opcode; for example, vendor extension 3DNow! is 0f 0f
    - 0f 38/3a primarily SSE* → separate opcode maps; additional table rows with repurposed prefixes 66, F2, F3
  - VEX (c4/c5), XOP (8f) prefixes → AVX, AES, FMA, etc maps with pfx byte 2, map_select[4:0]; {M,E}VEX (62)

Opcode, octal

- Most manuals opcode tables in hex, let's look at them in octal :)
- Octal groups encode groups of operation (8080/8085/z80 ISA design decisions)
- "*For some reason absolutely everybody misses all of this, even the Intel people who wrote the reference on the 8086 (and even the 8080).[1]*"
- Bits in opcode itself used for direction of operation, size of displacements, register encoding, condition codes, sign extension – this is in the SDM

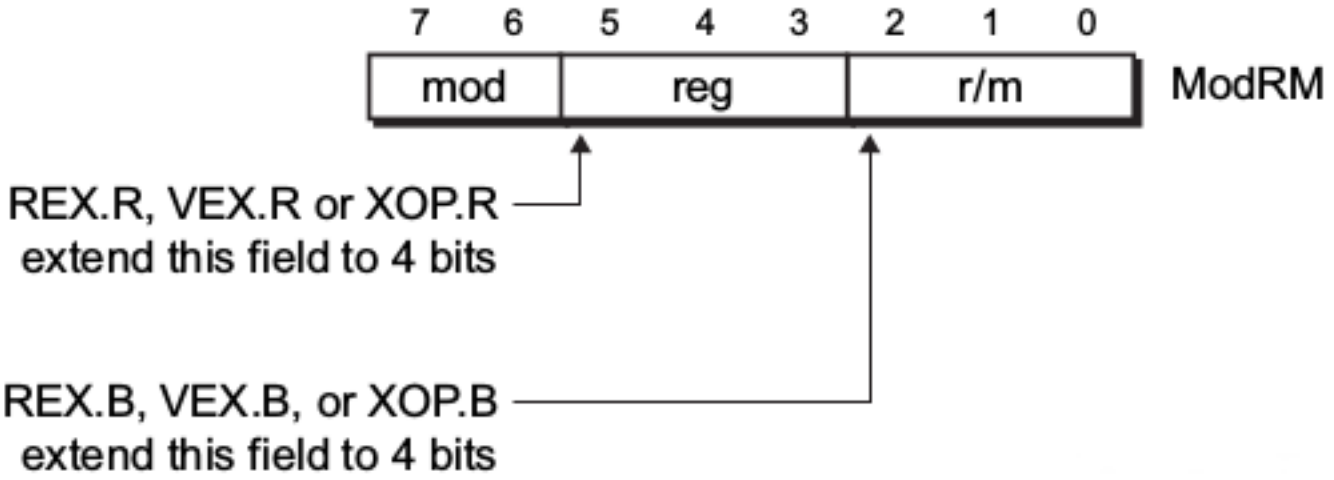
Opcodes in Octal groups/classes

- 000-077: arith-logical operations: ADD, ADC,SUB, SBB,AND...
  - 0P[0-7], where P in {0: add, 1: or, 2: adc, 3: sbb, 4: and, 5: sub, 6: xor, 7: cmp}
- 100-177: INC/PUSH/POP, Jcc,...
- 200-277: data movement: MOV,LODS,STOS,...
- 300-377: misc and escape groups

```
opc  oct  +dir,  +width
================================
0x00 0000 +{d: 0, w: 0}:    ADD Eb,Gb; ADD reg/mem8, reg8; 0x00 /r
0x01 0001 +{d: 0, w: 1}:    ADD Ev,Gv; ADD reg/mem{16,32,64}, reg{16,32,64}; 1 /r
0x02 0002 +{d: 1, w: 0}:    ADD Gb,Eb; ADD reg8, reg/mem8, 0x02 /r
0x03 0003 +{d: 1, w: 1}:    ADD Gv,Ev; ADD reg{16,32,64}, reg/mem{16,32,64};
0x3 /r
0x04 0004 +{d: 0, w: 0}:    ADD AL,Ib; ADD AL, imm8; 0x04 ib
0x05 0005 +{d: 0, w: 1}:    ADD rAX,Iz; ADD {,E,R}AX, imm{16,32}; with REX.W imm32
gets sign-extended to 64-bit
0x06 0006 +{d: 1, w: 0}:    PUSH ES; invalid in 64-bit mode
0x07 0007 +{d: 1, w: 1}:    POP ES; invalid in 64-bit mode
0x08 0010 +{d: 0, w: 0}:    OR Eb,Gb; OR reg/mem8, reg8; 0x08 /r
0x09 0011 +{d: 0, w: 1}:    OR Gv,Ev; OR reg/mem{16,32,64}, reg{16,32,64}; 0x09 /r
0x0a 0012 +{d: 1, w: 0}:    OR Gb,Eb; reg8, reg/mem8; 0x0a /r
0x0b 0013 +{d: 1, w: 1}:    OR Gv,Ev; OR reg{16,32,64}, reg/mem{16,32,64}; 0b /r
0x0c 0014 +{d: 0, w: 0}:    OR AL,Ib; OR AL, imm8; OC ib
0x0d 0015 +{d: 0, w: 1}:    OR rAX,Iz; OR rAX,imm{16,32}; 0d i{w,d}, rAX |
imm{16,32};RAX version sign-extends imm32
0x0e 0016 +{d: 1, w: 0}:    PUSH CS onto the stack
0x0f 0017 +{d: 1, w: 1}:    escape to secondary opcode map
0x10 0020 +{d: 0, w: 0}:    ADC Eb,Gb; ADC reg/mem8, reg8 + CF; 0x10 /r
0x11 0021 +{d: 0, w: 1}:    ADC Gv,Ev; ADC reg/mem{16,32,64}, reg{16,32,64} + CF;
0x11 /r
0x12 0022 +{d: 1, w: 0}:    ADC Gb,Eb; ADC reg8, reg/mem8 + CF; 0x12 /r
0x13 0023 +{d: 1, w: 1}:    ADC Gv,Ev; ADC reg16, reg/mem16; 13 /r; reg16 +=
reg/mem16 + CF
0x14 0024 +{d: 0, w: 0}:    ADC AL,Ib; ADC AL,imm8; AL += imm8 + rFLAGS.CF
0x15 0025 +{d: 0, w: 1}:    ADC rAX,Iz; ADC rAX, imm{16,32}; rAX += (sign-
extended) imm{16,32} + rFLAGS.CF
...
```

ModRM: Mode-Register-Memory

- Optional; describes operation and operands
- If missing, reg field in the opcode, i.e. PUSH/POP

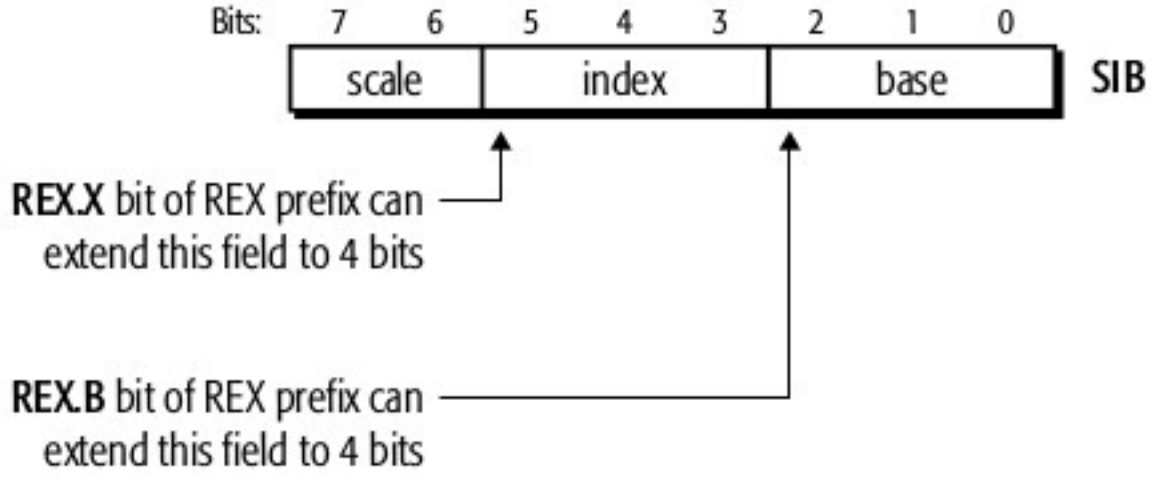


ModRM

- mod[7:6] – 4 addressing modes
  - 11b – register-direct
  - !11b – register-indirect modes, disp. specification follows
- reg[.R, 5:3] – register-based operand or extend operation encoding
- r/m[.B, 2:0] – register or memory operand when combined with **mod** field.
- Addressing mode can include a following SIB byte {mod=00b,r/m=101b}

SIB: Scale-Index-Base

- Indexed register-indirect addressing



SIB

- scale[7:6]: $2^{[6:7]scale}$ = scale factor
- index[.X, 5:3] – reg containing the index portion
- base[.B, 2:0] – reg containing the base portion
- `eff_addr = scale * index + base + offset`

```
Prefixes:
 REX:           4: [w]: 1 [r]: 0 [x]: 0 [b]: 0
Opcode:         0x89
ModRM:          0x94  [mod:10b][.R:0b,reg:010b][.B:0b,r/m:100b]
                register-indirect mode, 4-byte offset in displ. field
SIB:            0xc3 [.B:0b,base:011b][.X:0b,idx:000b][scale: 3]

 MOV Ev,Gv; MOV reg/mem{16,32,64} reg{16,32,64}
               0:       48 89 94 c3 00 10 00    mov %rdx,0x1000(%rbx,%rax,8)
               7:       00
```

Displacement

- signed offset
  - absolute: added to the base of the code segment
  - relative: rIP
- 1, 2 or 4 bytes
- sign-extended in 64-bit mode if operand 64-bit

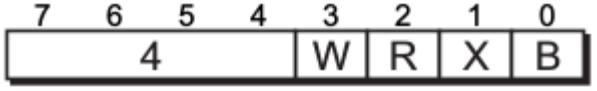
Immediates

- encoded in the instruction, come last
- 1,2,4 or 8 bytes
- with def. operand size in 64-bit mode, sign-extended
- MOV-to-GPR (A0-A3) versions can specify 64-bit immediate absolute address called *moffset.*

```
Prefixes:
 REX:           4: [w]: 1 [r]: 0 [x]: 0 [b]: 0
Opcode:         0xc7
ModRM:          0xc0  [mod:11b][.R:0b,reg:000b][.B:0b,r/m:000b]
                register-direct mode, reg operand specified by r/m.

 MOV Ev,Iz: MOV reg/mem{16,32,64}, imm{16,32} no 64-bit immed: C7 /0 i{w,d}
               0:       48 c7 c0 ef be ad de    mov $0xffffffffdeadbeef,%rax
```

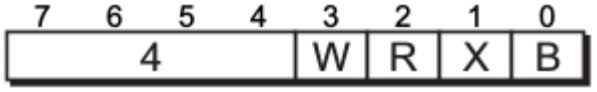
```
Prefixes:
 REX:           4: [w]: 1 [r]: 0 [x]: 0 [b]: 0
Opcode:         0xb8

 MOV r{AX,8},Iv: MOV reg{16,32,64}, imm{16,32,64}: 0xb8 + r{w,d,q} i{w,d,q}
               0:       48 b8 be ba fe ca ef    mov $0xdeadbeefcafebabe,%rax
               7:       be ad de
```

REX: AMD64



- only one allowed
- must come immediately before opcode
- with other mandatory prefixes, it comes after them
- when meaningless and specified → ignored
- Instruction bytes recycling
  - single-byte INC/DECs
  - ModRM versions in 64-bit mode

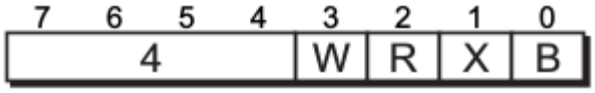
REX: AMD64



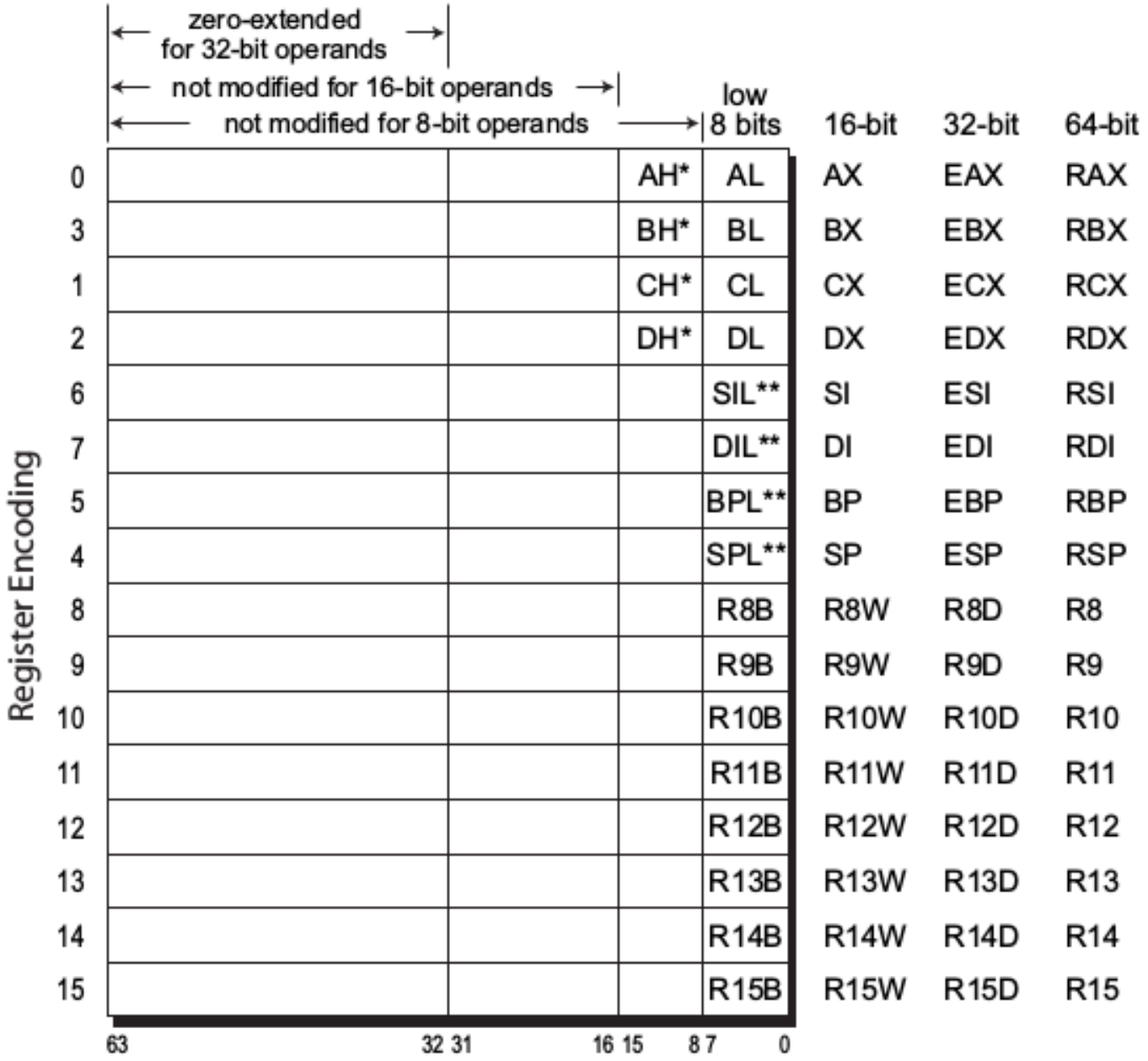
- 64-bit VAs/rIP, 64-bit PAs (actual width impl-specific)
- flat address space, no segmentation (not really)
- Widens GPRs to 64-bit
- Default operand size 32b, sign-extend to 64 if req.
  - (0x66 and REX.W=0b) → 16bit
  - REX.W=0 → CS.D
  - REX.W=1 → 64-bit

```
Prefixes:
 operand-size override: 0x66
 REX:            4: [w]: 0 [r]: 1 [x]: 0 [b]: 1
Opcode:          0x89
ModRM:           0xae  [mod:10b][.R:1b,reg:101b][.B:1b,r/m:110b]
                 register-indirect mode, 4-byte offset in displ. field

 MOV Ev,Gv; MOV reg/mem{16,32,64} reg{16,32,64}
                0:       66 45 89 ae 0a 05 00    mov %r13w,0x50a(%r14)
                7:       00
```

REX: Additional registers



- 8 new GPRs %r8-%r15 through REX[2:0] ([7:4] = 4h)
  - ModRM and SIB each use 3 bytes for register addressing. Add fourth with REX.{r,x,b} bits.
  - REX.R – extend ModRM.reg for reg selection (MSB)
  - REX.X – SIB.index extension (MSB)
  - REX.B – SIB.base or ModRM.r/m
- LSB-reg addressing capability: %spl,%bpl, %sil, %dil
  - REX selects those 4, %[a-d]h only addressable with !REX
  - %r[8-15]b selectable with REX.b=1b
- 8 additional 128-bit SSE* regs %xmm8-%xmm15


zero-extended
for 32-bit operands
not modified for 16-bit operands
not modified for 8-bit operands
low
8 bits
16-bit
32-bit
64-bit
Register Encoding
0 AH* AL AX EAX RAX
3 BH* BL BX EBX RBX
1 CH* CL CX ECX RCX
2 DH* DL DX EDX RDX
6 SIL** SI ESI RSI
7 DIL** DI EDI RDI
5 BPL** BP EBP RBP
4 SPL** SP ESP RSP
8 R8B R8W R8D R8
9 R9B R9W R9D R9
10 R10B R10W R10D R10
11 R11B R11W R11D R11
12 R12B R12W R12D R12
13 R13B R13W R13D R13
14 R14B R14W R14D R14
15 R15B R15W R15D R15
63 32 31 16 15 8 7 0

REX: Examples

```
Prefixes:
 REX:           4: [w]: 1 [r]: 0 [x]: 0 [b]: 0
Opcode:         0x89
ModRM:          0xc3  [mod:11b][.R:0b,reg:000b][.B:0b,r/m:011b]
                register-direct mode, reg operand specified by r/m.

 MOV Ev,Gv; MOV reg/mem{16,32,64} reg{16,32,64}
               0:       48 89 c3                mov %rax,%rbx
```

```
Opcode:         0x89
ModRM:          0xc3  [mod:11b][reg:000b][rm:011b]
                register-direct mode, reg operand specified by r/m.

 MOV Ev,Gv; MOV reg/mem{16,32,64} reg{16,32,64}
               0:       89 c3                   mov %eax,%ebx
```

```
Prefixes:
 REX:           4: [w]: 1 [r]: 1 [x]: 0 [b]: 1
Opcode:         0x89
ModRM:          0xc1  [mod:11b][.R:1b,reg:000b][.B:1b,r/m:001b]
                register-direct mode, reg operand specified by r/m.

 MOV Ev,Gv; MOV reg/mem{16,32,64} reg{16,32,64}
               0:       4d 89 c1                mov %r8,%r9
```

```
Prefixes:
 REX:           4: [w]: 0 [r]: 1 [x]: 0 [b]: 1
Opcode:         0x89
ModRM:          0xc1  [mod:11b][.R:1b,reg:000b][.B:1b,r/m:001b]
                register-direct mode, reg operand specified by r/m.

 MOV Ev,Gv; MOV reg/mem{16,32,64} reg{16,32,64}
               0:       45 89 c1                mov %r8d,%r9d
```

REX: Examples

```
Prefixes:
 REX:            4: [w]: 0 [r]: 0 [x]: 0 [b]: 0
Opcode:          0x80
ModRM:           0xfe  [mod:11b][.R:0b,reg:111b][.B:0b,r/m:110b]
                 register-direct mode, reg operand specified by r/m.

 CMP Eb,Ib; CMP reg/mem8, imm8, 80 /7 ib
               0:       40 80 fe 0c              cmp $0xc,%sil
```

```
Prefixes:
 REX:            4: [w]: 0 [r]: 0 [x]: 0 [b]: 1
Opcode:          0x80
ModRM:           0xfe  [mod:11b][.R:0b,reg:111b][.B:1b,r/m:110b]
                 register-direct mode, reg operand specified by r/m.

 CMP Eb,Ib; CMP reg/mem8, imm8, 80 /7 ib
               0:       41 80 fe 0c              cmp $0xc,%r14b
```

```
Opcode:          0x80
ModRM:           0xfe  [mod:11b][reg:111b][rm:110b]
                 register-direct mode, reg operand specified by r/m.

 CMP Eb,Ib; CMP reg/mem8, imm8, 80 /7 ib
               0:       80 fe 0c                 cmp $0xc,%dh
```

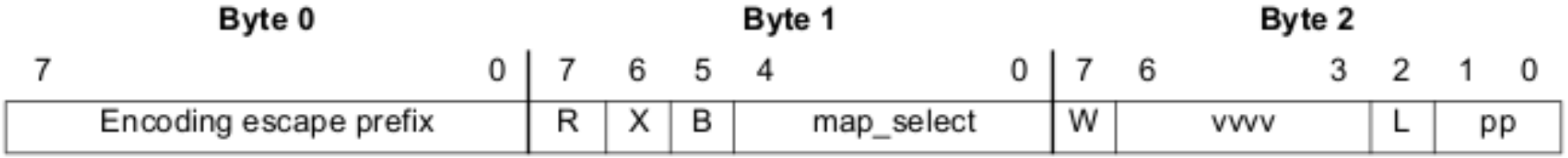
REX: RIP-relative addressing: cool

- only in control transfers in legacy mode
- PIC code + accessing global data much more efficient
- eff_addr = 4 byte signed disp (± 2G) + 64-bit next-rIP
- ModRM.mod=0b, r/m=101b (ModRM disp32 encoding in legacy; 64-bit mode encodes this with a SIB{base=101b,idx=100b,scale=n/a})
- the very first insn in vmlinux:

```
Prefixes:
 REX:           4 [w]: 1 [r]: 0 [x]: 0 [b]: 0
Opcode:         0x8d
ModRM:          0x2d  [mod:00b][.R:0b,reg:101b][.B:0b,r/m:101b]
                register-indirect mode, offset 0

 LEA Gv, M; LEA reg{16,32,64}, mem; 8D /r
               0:       48 8d 2d f9 ff ff ff    lea -0x7(%rip),%rbp
```

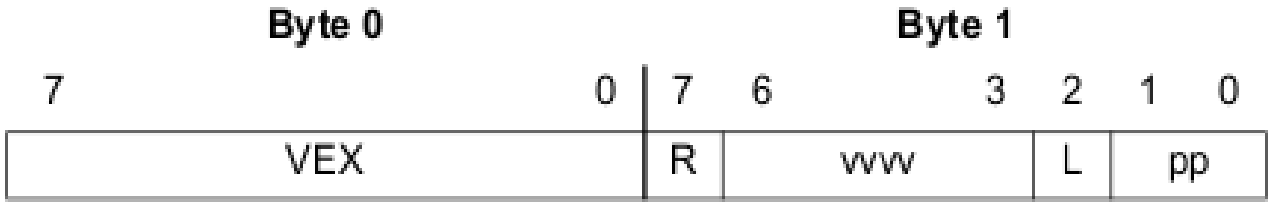
VEX/XOP

- VEX: C4 (LES: load far ptr in seg. reg. in legacy mode)
  - 3rd-byte: additional fields
  - spec. of 2 additional operands with another bit sim. to REX
  - alternate opcode maps
  - more compact/packed representation of an insn
- XOP: 8F; TBM insns on AMD
  - 8f /0, POP reg/mem{16,32,64} if XOP.map_select < 8



VEX, 2-byte

- C5 (LDS: load far ptr in %DS)
  - 128-bit, scalar and most common 256-bit AVX insns
  - has only REX.R equivalent VEX.R

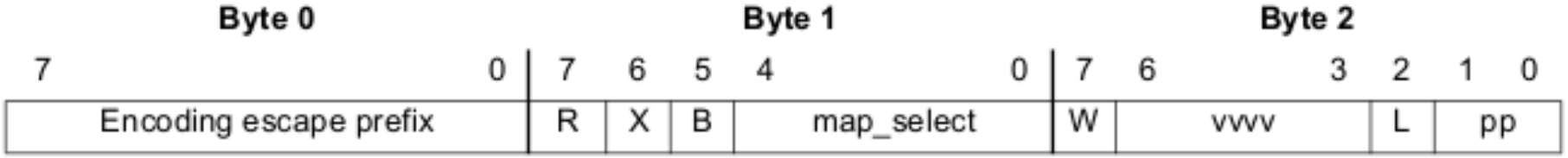


VEX

- must precede first opcode byte
- with SIMD (66/F2/F3), LOCK, REX prefixes → #UD
- regs spec. in 1s complement: 0000b → {X,Y}MM15/... , 1111b → {X,Y}MM0,...

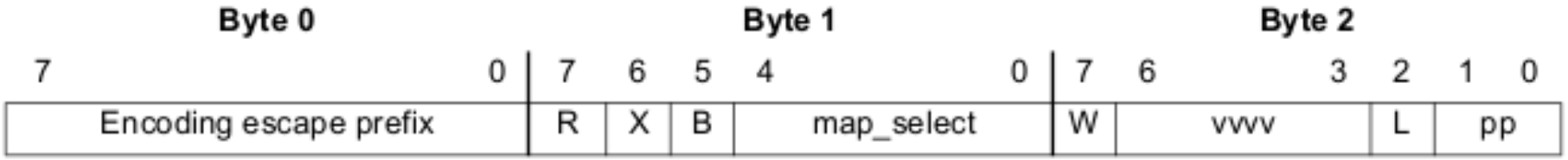
VEX/XOP structure

- byte0 [7:0] – encoding escape prefix
- byte1
  - R[7]: inverted, i.e. !ModRM.reg
  - X[6]: !SIB.idx ext
  - B[5]: !SIB.base or !ModRM.r/m
  - [4:0]: opcode map select
    - 0: reserved
    - 1: opcode map1: secondary opcode map
    - 2: opcode map2: 0f 38 three-byte map
    - 3: opcode map3: 0f 3a three-byte map
    - 8-1f: XOP maps



VEX/XOP structure

- byte 2:
  - W[7]: GPR operand size/op conf for certain X/YMM regs
  - vvvv[6:3]: non-desctructive src/dst reg selector in 1s complement
  - L[2]: vector length: 0b → 128bit, 1b → 256bit
  - pp[1:0]- SIMD eqiuv. to 66, F2 or F3 opcode ext.



AVX512

- EVEX: 62h (BOUND, invalid in 64-bit, MPX defines new insns)
- 4-byte long spec.
- 32 vector registers: zmm0-zmm31
- 8 new opmask registers k0-k7
- along with bits for those...
- Fun :-)

Kernel Hacks

Alternatives

- Replace instructions with “better” ones at runtime
  - When a CPU with a certain feature has been detected
    - we like to run everything in hw, if possible
  - When we online a second CPU, i.e. SMP (btw, we don't patch back to UP anymore, see 816afe4ff98e), we would like to adjust locking
  - Wrap vendor-specific pieces:
    - rdtsc_barrier(), AMD synces with MFENCE, Intel/Centaur with LFENCE
  - Bug workarounds: X86_BUG_11AP
- Thus, optimize generic kernel for hw it is running on → use single kernel image

Alternatives: Example

- Select b/w function call and insn call
- Instruction has equivalent functionality
- POPCNT vs __sw_hweight64

```
unsigned long __sw_hweight64(__u64 w)
{
#if BITS_PER_LONG == 32
        return __sw_hweight32((unsigned int)(w >> 32)) +
                __sw_hweight32((unsigned int)w);
#elif BITS_PER_LONG == 64
#ifdef ARCH_HAS_FAST_MULTIPLIER
        w -= (w >> 1) & 0x5555555555555555ul;
        w =  (w & 0x3333333333333333ul) + ((w >> 2) & 0x3333333333333333ul);
        w =  (w + (w >> 4)) & 0x0f0f0f0f0f0f0f0ful;
        return (w * 0x0101010101010101ul) >> 56;
#else
        __u64 res = w - ((w >> 1) & 0x5555555555555555ul);
        res = (res & 0x3333333333333333ul) + ((res >> 2) & 0x3333333333333333ul);
        res = (res + (res >> 4)) & 0x0F0F0F0F0F0F0F0Ful;
        res = res + (res >> 8);
        res = res + (res >> 16);
        return (res + (res >> 32)) & 0x00000000000000FFul;
#endif
#endif
}
```

Alternatives: Example

```
static inline unsigned long __arch_hweight64(__u64 w)
{
        unsigned long res = 0;

#ifdef CONFIG_X86_32
        return  __arch_hweight32((u32)w) +
                __arch_hweight32((u32)(w >> 32));
#else
        asm (ALTERNATIVE("call __sw_hweight64", POPCNT64, X86_FEATURE_POPCNT)
                        : "="REG_OUT (res)
                        : REG_IN (w));
#endif /* CONFIG_X86_32 */

        return res;
}
```

Alternatives: Example

```
old insn VA: 0xffffffff81b90959, CPU feat: 4*32+23 (151), size: 5
 ffffffff81b90959:      e8 b2 53 76 ff          callq 0xffffffff812f5d10
repl insn: 0xffffffff81c7fcbb, size: 5
 ffffffff81c7fcbb:      f3 48 0f b8 c7          popcnt %rdi,%rax
```

```
ffffffff81b9092a <alternative_instructions>:
ffffffff81b9092a:       55                      push   %rbp
ffffffff81b9092b:       48 89 e5                mov    %rsp,%rbp
ffffffff81b9092e:       e8 2d 74 47 ff          callq  ffffffff81007d60 <stop_nmi>
ffffffff81b90933:       48 c7 c6 04 fb c7 81    mov    $0xffffffff81c7fb04,%rsi
ffffffff81b9093a:       48 c7 c7 f8 8e c7 81    mov    $0xffffffff81c78ef8,%rdi
ffffffff81b90941:       e8 5a a7 47 ff          callq  ffffffff8100b0a0 <apply_alternatives>
ffffffff81b90946:       83 3d 6f 0e 10 00 00    cmpl   $0x0,0x100e6f(%rip)        # ffffffff81c917bc <noreplace_smp>
ffffffff81b9094d:       75 4d                   jne    ffffffff81b9099c <alternative_instructions+0x72>
ffffffff81b9094f:       48 8b 05 8a 25 aa ff    mov    -0x55da76(%rip),%rax        # ffffffff81632ee0 <cpu_present_mask>
ffffffff81b90956:       0f b6 38                movzbl (%rax),%edi
ffffffff81b90959:       e8 b2 53 76 ff          callq  ffffffff812f5d10 <__sw_hweight64>
```

```
ffffffff81c7fb04 <.altinstr_replacement>:
...
ffffffff81c7fcb2:       ff ca                   dec    %edx
ffffffff81c7fcb4:       75 f6                   jne    ffffffff81c7fcac <__alt_instructions_end+0x1a8>
ffffffff81c7fcb6:       f3 48 0f b8 c7          popcnt %rdi,%rax
ffffffff81c7fcbb:       f3 48 0f b8 c7          popcnt %rdi,%rax
ffffffff81c7fcc0:       f3 48 0f b8 c7          popcnt %rdi,%rax
ffffffff81c7fcc5:       f3 48 0f b8 c7          popcnt %rdi,%rax
ffffffff81c7fcca:       f3 48 0f b8 c7          popcnt %rdi,%rax
ffffffff81c7fccf:       f3 48 0f b8 c7          popcnt %rdi,%rax
```

Alternatives: Example 2

- static_cpu_has_safe()
- test CPU features as fast as possible
- initially run safe variant before alternatives have run
- overwrite JMP to out-of-line code when CPU feature present

```
static __always_inline __pure bool use_eager_fpu(void)
{
        return static_cpu_has_safe(X86_FEATURE_EAGER_FPU);
}
```

```
static __always_inline __pure bool _static_cpu_has_safe(u16 bit)
{
#ifdef CC_HAVE_ASM_GOTO
/*
 * We need to spell the jumps to the compiler because, depending on the offset,
 * the replacement jump can be bigger than the original jump, and this we cannot
 * have. Thus, we force the jump to the widest, 4-byte, signed relative
 * offset even though the last would often fit in less bytes.
 */
                asm_volatile_goto("1: .byte 0xe9\n .long %l[t_dynamic] - 2f\n"
                        "2:\n"
                        ".section .altinstructions,\"a\"\n"
                        " .long 1b - .\n"                       /* src offset */
                        " .long 3f - .\n"                       /* repl offset */
                        " .word %P1\n"                          /* always replace */
                        " .byte 2b - 1b\n"                      /* src len */
                        " .byte 4f - 3f\n"                      /* repl len */
                        ".previous\n"
                        ".section .altinstr_replacement,\"ax\"\n"
                        "3: .byte 0xe9\n .long %l[t_no] - 2b\n"
                        "4:\n"
                        ".previous\n"
                        ".section .altinstructions,\"a\"\n"
                        " .long 1b - .\n"                       /* src offset */
                        " .long 0\n"                            /* no replacement */
                        " .word %P0\n"                          /* feature bit */
                        " .byte 2b - 1b\n"                      /* src len */
                        " .byte 0\n"                            /* repl len */
                        ".previous\n"
                        : : "i" (bit), "i" (X86_FEATURE_ALWAYS)
                        : : t_dynamic, t_no);
                return true;
        t_no:
                return false;
        t_dynamic:
                return __static_cpu_has_safe(bit);
```

Alternatives: Example 2

```
old insn VA: 0xffffffff81001440, CPU feat: X86_FEATURE_ALWAYS, size: 5
__switch_to:
 ffffffff81001440:      e9 4b 00 00 00          jmpq 0xffffffff81001490
repl insn: 0xffffffff81e1c028, size: 5
 ffffffff81001440:      e9 59 00 00 00          jmpq 0xffffffff8100149e

old insn VA: 0xffffffff81001440, CPU feat: X86_FEATURE_EAGER_FPU, size: 5
__switch_to:
 ffffffff81001440:      e9 4b 00 00 00          jmpq 0xffffffff81001490
repl insn: 0xffffffff81e15180, size: 0
```

```
ffffffff81001400 <__switch_to>:
ffffffff81001400:       55                      push   %rbp
ffffffff81001401:       48 89 e5                mov    %rsp,%rbp
ffffffff81001404:       41 57                   push   %r15
ffffffff81001440:       e9 4b 00 00 00          jmpq   ffffffff81001490 <__switch_to+0x90>
ffffffff81001445:       41 8b b4 24 9c 05 00    mov    0x59c(%r12),%esi
ffffffff8100144c:       00
ffffffff8100144d:       85 f6                   test   %esi,%esi
ffffffff8100144f:       0f 85 8b 00 00 00       jne    ffffffff810014e0 <__switch_to+0xe0>
ffffffff81001455:       41 c6 84 24 bc 05 00    movb   $0x0,0x5bc(%r12)
ffffffff8100145c:       00 00
ffffffff8100145e:       41 c7 84 24 98 05 00    movl   $0xffffffff,0x598(%r12)
ffffffff81001465:       00 ff ff ff ff
ffffffff8100146a:       80 83 bc 05 00 00 01    addb   $0x1,0x5bc(%rbx)
ffffffff81001471:       e9 ba 02 00 00          jmpq   ffffffff81001730 <__switch_to+0x330>
ffffffff81001476:       48 8b 83 a0 05 00 00    mov    0x5a0(%rbx),%rax
ffffffff8100147d:       0f 18 08                prefetcht0 (%rax)
ffffffff81001480:       be 01 00 00 00          mov    $0x1,%esi
ffffffff81001485:       e9 66 00 00 00          jmpq   ffffffff810014f0 <__switch_to+0xf0>
ffffffff8100148a:       eb 7a                   jmp    ffffffff81001506 <__switch_to+0x106>
ffffffff8100148c:       0f 1f 40 00             nopl   0x0(%rax)
ffffffff81001490:       bf 7d 00 00 00          mov    $0x7d,%edi
ffffffff81001495:       e8 46 40 01 00          callq  ffffffff810154e0 <__static_cpu_has_safe>
ffffffff8100149a:       84 c0                   test   %al,%al
ffffffff8100149c:       75 32                   jne    ffffffff810014d0 <__switch_to+0xd0>
ffffffff8100149e:       80 bb bc 05 00 00 05    cmpb   $0x5,0x5bc(%rbx)
```

Alternatives: how

- `.altinstructions` and `.altinstr_replacement` sections
- `.altinstructions` contains info for runtime replacing:

```
struct alt_instr {
        s32 instr_offset;       /* original instruction */
        s32 repl_offset;        /* offset to replacement instruction */
        u16 cpuid;              /* cpuid bit set for replacement */
        u8  instrlen;           /* length of original instruction */
        u8  replacementlen;     /* length of new instruction, <= instrlen */
};
```

- Rely on linker to compute proper offsets: s32
- Use the position we're loaded at + offsets to get to both old and replacement instructions

ELF editing

```
diff --git a/arch/x86/include/asm/cpufeature.h b/arch/x86/include/asm/cpufeature.h
index bb9b258d60e7..814261c7745c 100644
--- a/arch/x86/include/asm/cpufeature.h
+++ b/arch/x86/include/asm/cpufeature.h
@@ -470,7 +470,7 @@ static __always_inline __pure bool _static_cpu_has_safe(u16 bit)
  * have. Thus, we force the jump to the widest, 4-byte, signed relative
  * offset even though the last would often fit in less bytes.
  */
-               asm_volatile_goto("1: .byte 0xe9\n .long %l[t_dynamic] - 2f\n"
+               asm_volatile_goto("1: jmp %l[t_dynamic]\n"
                                  "2:\n"
                                  ".section .altinstructions,\"a\"\n"
                                  " .long 1b - .\n"                /* src offset */
@@ -480,7 +480,7 @@ static __always_inline __pure bool _static_cpu_has_safe(u16 bit)
                                  " .byte 4f - 3f\n"               /* repl len */
                                  ".previous\n"
                                  ".section .altinstr_replacement,\"ax\"\n"
-                                 "3: .byte 0xe9\n .long %l[t_no] - 2b\n"
+                                 "3: jmp %l[t_no]\n"
                                  "4:\n"
                                  ".previous\n"
                                  ".section .altinstructions,\"a\"\n"
```

ELF editing

- Need to control exact JMP versions
- Current solution is to enforce JMPs with s32 offsets
- Some of them can be 1- or 2-byte JMPs => lower I$/prefetcher load
- Parse vmlinux and binary-edit the JMPs
- Prototype in arch/x86/tools/relocs.*

ELF editing, proto

- original alt_instr contents:

```
old insn VA: 0xffffffff81001440, CPU feat: X86_FEATURE_ALWAYS, size: 2
__switch_to:
 ffffffff81001440:      eb 4e                   jmp 0xffffffff81001490
repl insn: 0xffffffff81e1c068, size: 5
 ffffffff81e1c068:      e9 31 54 1e ff          jmpq 0xffffffff8100149e
```

- converted JMP:

```
old insn VA: 0xffffffff81001440, CPU feat: X86_FEATURE_ALWAYS, size: 2
__switch_to:
 ffffffff81001440:      eb 4e                    jmp 0xffffffff81001490
repl insn: 0xffffffff81e1c068, size: 5
 ffffffff81001440:      eb 5c                    jmp 0xffffffff8100149e
```

ELF editing

```
@@ -34,9 +34,9 @@ ffffffff8100147a:     0f 18 00
 old insn VA: 0xffffffff81001482, CPU feat: X86_FEATURE_ALWAYS, size: 2
 __switch_to:
  ffffffff81001482:     eb 6c                   jmp 0xffffffff810014f0
-repl insn: 0xffffffff81e1c075, size: 5
- ffffffff81e1c075:     e9 8a 54 1e ff          jmpq 0xffffffff81001504
-ffffffff81001482:     e9 8a 54 1e ff          jmpq 0xffffffff801e6911
+repl insn: 0xffffffff81e1c075, size: 4
+ ffffffff81e1c075:     66 e9 7e 00             jmp 0xffffffff81e1c0f7
+ffffffff81001482:     66 e9 7e 00             jmp 0xffffffff81001504

 old insn VA: 0xffffffff81001482, CPU feat: X86_FEATURE_EAGER_FPU, size: 2
 __switch_to:
@@ -46,9 +46,9 @@ repl insn: 0xffffffff81e151fc, size: 0
 __switch_to:
  ffffffff810016b0:     e9 23 01 00 00          jmpq 0xffffffff810017d8
-repl insn: 0xffffffff81e1c07a, size: 5
- ffffffff81e1c07a:     e9 41 56 1e ff          jmpq 0xffffffff810016c0
-ffffffff810016b0:     e9 41 56 1e ff          jmpq 0xffffffff801e6cf6
+repl insn: 0xffffffff81e1c07a, size: 2
+ ffffffff81e1c07a:     eb 0e                   jmp 0xffffffff81e1c08a
+ffffffff810016b0:     eb 0e                   jmp 0xffffffff810016c0
```

ELF editing: Problem

```
at 0x121527c
 old { va: 0xffffffff81001772, off: 0x1772, PA: 0x201772 CPU: 117, len: 2}
  insn bytes: eb 7c
 repl { va: 0xffffffff81e1c096, off: 0x2e, PA: 0x121c096 }
  insn bytes: e9 6d 57 1e ff
  next_rip: 0xffffffff81e1c097, in-insn disp: 0xff1e576d, tgt_rip: 0xffffffff81001804
 actual off: 0x92
 modified repl_insn bytes: 66 e9 92 00
```

Exception tables

- Collect addresses of insns which can cause specific exceptions: #PF, #GP, #MF, #XF
- When hit, jump to fixup code
- What are they good for:
  - accessing process address space
  - accessing maybe unimplemented hw resources (MSRs,...)
  - WP bit test on some old x86 CPUs
  - …
- basically everywhere where we can recover safely from faulting on an insn

Exception tables: how

- `__ex_table` section contains

```
struct exception_table_entry {
        int insn, fixup;
};
```

- both are relative to the entry itself: `&e->insn + e->insn`
- ->insn is the addr where we fault, ->fixup is where we jump to after handling the fault
- fixup_exception(): put new rIP into regs->ip and say that we've fixed up the exception

Exception tables: Examples

```
static inline unsigned long long native_read_msr_safe(unsigned int msr,
                                                      int *err)
{
        DECLARE_ARGS(val, low, high);

        asm volatile("2: rdmsr ; xor %[err],%[err]\n"
                     "1:\n\t"
                     ".section .fixup,\"ax\"\n\t"
                     "3:  mov %[fault],%[err] ; jmp 1b\n\t"
                     ".previous\n\t"
                     _ASM_EXTABLE(2b, 3b)
                     : [err] "=r" (*err), EAX_EDX_RET(val, low, high)
                     : "c" (msr), [fault] "i" (-EIO));
        return EAX_EDX_VAL(val, low, high);
}
```

Exception tables: Examples

```
#NO_APP
.L114:
        movl    $139, %ecx      #, tmp407
#APP
# 73 "./arch/x86/include/asm/msr.h" 1
        2: rdmsr ; xor %esi,%esi        # __err
1:
        .section .fixup,"ax"
        3:  mov $-5,%esi ; jmp 1b       #, __err
        .previous
         .pushsection "__ex_table","a"
 .balign 8
 .long (2b) - .
 .long (3b) - .
 .popsection
```

Exception tables: Examples

```
native_read_msr_safe:
ffffffff81040316:       0f 32                   rdmsr

ffffffff81607f67:       b9 fb ff ff ff          mov $0xfffffffb,%ecx
```

- -EIO

```
__put_user_4:
ffffffff812cd650:       89 01                   mov %eax,(%rcx)

bad_put_user:
ffffffff812cd689:       b8 f2 ff ff ff          mov $0xfffffff2,%eax
```

```
__put_user_8:
ffffffff812cd680:       48 89 01                mov %rax,(%rcx)

bad_put_user:
ffffffff812cd689:       b8 f2 ff ff ff          mov $0xfffffff2,%eax
```

- -EFAULT

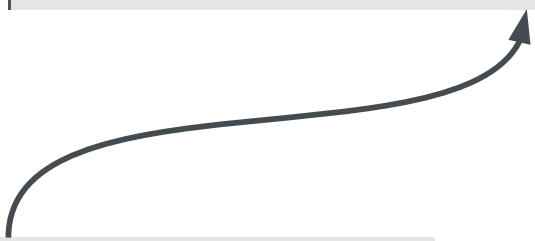
jmp labels/static keys

- Put seldomly executed code in a fast-path optimally
- Main user: tracepoints for zero-overhead tracing
- GCC puts unlikely code out-of-line for default case
- GCC machinery: asm goto

jmp labels: Example

- in C:

```
int sched_clock_stable(void)
{
        return static_key_false(&__sched_clock_stable);
}
```



```
u64 cpu_clock(int cpu)
{
        if (!sched_clock_stable())
                return sched_clock_cpu(cpu);

        return sched_clock();
}
```

jmp labels: Example

```
static __always_inline bool arch_static_branch(struct static_key *key)
{
        asm_volatile_goto("1:"
                ".byte " __stringify(STATIC_KEY_INIT_NOP) "\n\t"
                ".pushsection __jump_table,  \"aw\" \n\t"
                _ASM_ALIGN "\n\t"
                _ASM_PTR "1b, %l[l_yes], %c0 \n\t"
                ".popsection \n\t"
                : :  "i" (key) : : l_yes);
        return false;
l_yes:
        return true;
}
```

jmp labels: Example

```
#APP
# 21 "./arch/x86/include/asm/jump_label.h" 1
        1:.byte 0x0f,0x1f,0x44,0x00,0
        .pushsection __jump_table,  "aw"
         .balign 8
         .quad 1b, .L65, __sched_clock_stable    #,
        .popsection

# 0 "" 2
#NO_APP
        call    sched_clock_cpu #
        popq    %rbp    #
        ret
        .p2align 4,,10
        .p2align 3
.L65:
        call    sched_clock     #
        popq    %rbp    #
        ret
        .size   cpu_clock, .-cpu_clock
        .section        .text.unlikely
```

```
struct jump_entry {
        jump_label_t code;
        jump_label_t target;
        jump_label_t key;
};
```

jmp labels: Example

- and because a picture says a 1000 words...

```
cpu_clock:
target: ffffffff810826a0: key: __sched_clock_stable (0xffffffff81e5af90), replacement: jmp 0x7

ffffffff81082690:       55                      push %rbp
ffffffff81082691:       48 89 e5                mov %rsp,%rbp
ffffffff81082694:       0f 1f 44 00 00          nop 0x0(%rax,%rax,1)                --*
ffffffff81082699:       e8 f2 fe ff ff          callq 0xffffffff81082590
ffffffff8108269e:       5d                      pop %rbp
ffffffff8108269f:       c3                      retq
ffffffff810826a0:       e8 ab a0 f8 ff          callq 0xffffffff8100c750            <--
ffffffff810826a5:       5d                      pop %rbp
ffffffff810826a6:       c3                      retq
```

jmp labels: TODO

- JMP to unlikely code only initially
- avoid unconditional JMP after jmp labels init has run

```
when the jump labels get initialized and all checks done, at runtime we
have this:

          0xffffffff8100ce40 <sched_clock>:    push   %rbp
          0xffffffff8100ce41 <sched_clock+1>:  mov    %rsp,%rbp
          0xffffffff8100ce44 <sched_clock+4>:  and    $0xfffffffffffffff0,%rsp

unconditional JMP!!!

          0xffffffff8100ce48 <sched_clock+8>:  jmpq   0xffffffff8100ce70 <sched_clock+48>

unlikely code using jiffies

          0xffffffff8100ce4d <sched_clock+13>: mov    0x9a71ac(%rip),%r8        # 0xffffffff819b4000 <jiffies_64>
          0xffffffff8100ce54 <sched_clock+20>: movabs $0xffc2f745d964b800,%rax
          0xffffffff8100ce5e <sched_clock+30>: leaveq
          0xffffffff8100ce5f <sched_clock+31>: imul   $0x3d0900,%r8,%rdx
          0xffffffff8100ce66 <sched_clock+38>: add    %rdx,%rax
          0xffffffff8100ce69 <sched_clock+41>: retq
          0xffffffff8100ce6a <sched_clock+42>: nopw   0x0(%rax,%rax,1)

likely code using RDTSC, see JMP target address.

          0xffffffff8100ce70 <sched_clock+48>: rdtsc
```

Backup

Decoding an oops “Code:” section

- last talk was about oopses
- let's connect it to this one

```
general protection fault: 0000 [#1] PREEMPT SMP
Modules linked in:
CPU: 0 PID: 0 Comm: swapper/0 Not tainted 3.11.0-rc3+ #4
Hardware name: Bochs Bochs, BIOS Bochs 01/01/2011
task: ffffffff81a10440 ti: ffffffff81a00000 task.ti: ffffffff81a00000
RIP: 0010:[<ffffffff81015aaa>]  [<ffffffff81015aaa>] init_amd+0x1a/0x640
RSP: 0000:ffffffff81a01ed8  EFLAGS: 00010296
RAX: ffffffff81015a90 RBX: 0000000000726f73 RCX: 00000000deadbeef
RDX: 0000000000000000 RSI: 0000000000000000 RDI: ffffffff81aadf00
RBP: ffffffff81a01f18 R08: 0000000000000000 R09: 0000000000000001
R10: 0000000000000001 R11: 0000000000000000 R12: ffffffff81aadf00
R13: ffffffff81b572e0 R14: ffff88007ffd8400 R15: 0000000000000000
FS:  0000000000000000(0000) GS:ffff88007fc00000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: ffff88000267c000 CR3: 0000000001a0b000 CR4: 00000000000006b0
Stack:
 ffffffff817cda76 0000000000000001 0000001000000001 0000000000000000
 ffffffff81a01f18 0000000000726f73 ffffffff81aadf00 ffffffff81b572e0
 ffffffff81a01f38 ffffffff81014260 ffffffffffffffff ffffffff81b50020
Call Trace:
 [<ffffffff81014260>] identify_cpu+0x2d0/0x4d0
 [<ffffffff81ad53b9>] identify_boot_cpu+0x10/0x3c
 [<ffffffff81ad5409>] check_bugs+0x9/0x2d
 [<ffffffff81acfe31>] start_kernel+0x39d/0x3b9
 [<ffffffff81acf894>] ? repair_env_string+0x5a/0x5a
 [<ffffffff81acf5a6>] x86_64_start_reservations+0x2a/0x2c
 [<ffffffff81acf699>] x86_64_start_kernel+0xf1/0xf8
Code: 00 0f b6 33 eb 8f 66 66 2e 0f 1f 84 00 00 00 00 00 e8 2b 2b 4e 00 55 b9 ef be ad de
48 89 e5 41 55 41 54 49 89 fc 53 48 83 ec 28 <0f> 32 80 3f 0f 0f 84 13 02 00 00 4c 89 e7
e8 03 fd ff ff f0 41
RIP  [<ffffffff81015aaa>] init_amd+0x1a/0x640
 RSP <ffffffff81a01ed8>
---[ end trace 3c9ee0eeb6dd208c ]---
Kernel panic - not syncing: Fatal exception
```

```
$ ./scripts/decodecode < ~/dev/boris/x86d/oops.txt
[ 0.016000] Code: ff ff 66 66 66 66 66 66 2e 0f 1f 84 00 00 00 00 00 e8 1b bb 58 00 55 b9 ef be
ad de 48 89 e5 41 55 41 54 49 89 fc 53 48 83 ec 20 <0f> 32 80 3f 0f 0f 84 0b 02 00 00 4c 89 e7
e8 e3 fe ff ff f0 41
All code
========
   0:   ff                      (bad)
   1:   ff 66 66                jmpq   *0x66(%rsi)
   4:   66 66 66 66 2e 0f 1f    data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
   b:   84 00 00 00 00 00
  11:   e8 1b bb 58 00          callq  0x58bb31
  16:   55                      push   %rbp
  17:   b9 ef be ad de          mov    $0xdeadbeef,%ecx
  1c:   48 89 e5                mov    %rsp,%rbp
  1f:   41 55                   push   %r13
  21:   41 54                   push   %r12
  23:   49 89 fc                mov    %rdi,%r12
  26:   53                      push   %rbx
  27:   48 83 ec 20             sub    $0x20,%rsp
  2b:*  0f 32                   rdmsr          <-- trapping instruction
  2d:   80 3f 0f                cmpb   $0xf,(%rdi)
  30:   0f 84 0b 02 00 00       je     0x241
  36:   4c 89 e7                mov    %r12,%rdi
  39:   e8 e3 fe ff ff          callq  0xffffffffffffff21
  3e:   f0                      lock
  3f:   41                      rex.B

Code starting with the faulting instruction
===========================================
   0:   0f 32                   rdmsr
   2:   80 3f 0f                cmpb   $0xf,(%rdi)
   5:   0f 84 0b 02 00 00       je     0x216
   b:   4c 89 e7                mov    %r12,%rdi
   e:   e8 e3 fe ff ff          callq  0xfffffffffffffef6
  13:   f0                      lock
  14:   41                      rex.B
```

```
$ grep Code oops.txt | sed 's/[<>]//g; s/^\s*\[.*Code: //' | ./x86d -m 2
-
WARNING: Invalid instruction: 0xff
                1:   ff 66 66                jmpq *0x66(%rsi)
                4:   66 66 66 66 2e 0f 1f    data32 data32 data32 nop
%cs:0x0(%rax,%rax,1)
                b:   84 00 00 00 00 00
               11:   e8 1b bb 58 00          callq 0x58bb31
               16:   55                      push %rbp
               17:   b9 ef be ad de          mov $0xdeadbeef,%ecx
               1c:   48 89 e5                mov %rsp,%rbp
               1f:   41 55                   push %r13
               21:   41 54                   push %r12
               23:   49 89 fc                mov %rdi,%r12
               26:   53                      push %rbx
               27:   48 83 ec 20             sub $0x20,%rsp
               2b:   0f 32                   rdmsr
               2d:   80 3f 0f                cmpb $0xf,(%rdi)
               30:   0f 84 0b 02 00 00       jz 0x241
               36:   4c 89 e7                mov %r12,%rdi
               39:   e8 e3 fe ff ff          callq 0xffffffffffffff21
```