CSci 5980/8980
Manual and Automated Binary Reverse Engineering
Day 2: x86 Overview and Arithmetic

Stephen McCamant
University of Minnesota

Outline

Brief x86 history (1)

- 4-bit Intel 4004 and 8-bit 8008 were mostly for calculators
- 8-bit 8080 powered early hobbyist micro computers
- 16-bit 8086 was binary incompatible (but partially assembly-level compatible) with the 8080
- Cheaper-package 8088 edition of 8086 selected by IBM for the original IBM PC

Brief x86 history (2)

- 80286 added memory protection not easily usable by MS-DOS
- 80386 introduced 32-bit mode and paging
  - Supported modern OSes like Unix and Windows NT
- 80486 was almost the same ISA, but faster
  - Cache, pipelining
- What would have been the 80586 was sold as the "Pentium"

Brief x86 history (3)

- Intel had a history of designing clean-sheet processors that were undercut by cheaper x86es
  - Its first 32-bit, RISC, VLIW, and 64-bit processors were never popular on the desktop
- A backwards-compatible 64-bit extension was designed by AMD undercutting Intel/HP Itanium
- Later adopted by Intel making it a de-facto standard
  - Various called x86-64, AMD64, EMT64T, Intel 64, x64

The x86 ISA: CISC vs. RISC

- Called CISC because it predates the 80s/90s RISC revolution
  - Pre-RISC ISAs were for human assembly programmers
  - RISC CPUs had simpler instructions, moving complexity to compilers
  - (Note, ISAs grew more complex over time anyway)
- ISA is the only aspect of x86 that did not change
  - x86-64 compilers mostly use the RISC-like instructions
  - The internals of modern x86 CPUs are RISC-like

x86 ISA attributes

- Variable-length byte-granularity instructions
- Most instructions overwrite one operand
  - "Two-address" instead of "three-address" style
- Most instructions allow one operand in memory
  - Versus load-store style of RISC
- Rich addressing modes
- Branching using condition codes

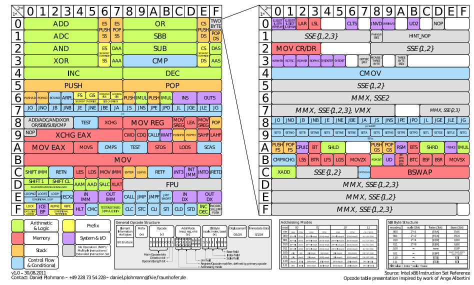
x86 instruction encoding

- Variable length instructions are a "prefix code":
  - Values of bytes tell how many more to read
- Short encodings (some 1 byte) for simple/common instructions
- Long encodings for rare/newer instructions and complex operands
- Overall limit of 15 bytes for any instruction

x86 instruction format parts

- Optional prefix bytes
- One, two, or three-byte opcode
- Extra bytes specifying operands
  - Many insns have a "mod/reg/RM" byte
  - Some addressing modes have an "SIB" byte
  - Some addressing modes have a constant displacement
- Sometimes a immediate (constant) operand

x86 opcode map



Prefix bytes

- 0x26, 0x2e, 0x36, 0x3e, 0x64, 0x65: segment overrides
  - A mostly-obsolete memory management feature
- 0x66 operand size override
  - In 32-bit mode, operand is 16-bit (and vice-versa)
- 0x67 address size override (rarely used)
- 0xf0 lock: block concurrent access
- 0xf2, 0xf3: repne and rep/repe, repeat string operation

x86 condition codes

- Six one-bit flags set based on math or comparison results:
  - CF: (unsigned) carry out
  - OF: (signed) overflow
  - ZF: result is zero
  - SF: result is negative ("sign")
  - PF: parity of result (mostly historical)
  - AF: adjustment needed for BCD (mostly historical)
- More about these when we cover branches

x86-32 operand sizes

- Many general-purpose/integer arithmetic insns can operate on 8, 16, or 32-bit values
- Sometimes the byte insn has an even opcode and the 32-bit opcode is one higher
- For a 16-bit version, use the 32-bit opcode with a 0x66 prefix byte
  - Pre-386, these opcodes were 16-bit

x86-32 general-purpose registers

- 8, 32-bit registers for integers or pointers
- In encoding order: `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`
- Without the "e", refers to the low 16-bits of the 32-bit register

Every register is special

- `esp`: used as stack pointer by stack accesses
- `ebp`: used as frame pointer by `enter`/`leave`
- `eax`: some instructions can only apply to `eax`
- `edx`: more-significant half associated with `eax`
- `ecx`: used a count in loops and shifts
- `esi`, `edi`: source and destination for string ops

The Mod + Reg/Opcode + R/M byte

- Most insns with variable operands have an extra byte to specify them
- 3 fields: 2-bit Mod, 3-bit Reg/Opcode, 3-bit R/M
- The Mod and R/M fields specify an operand that could be in memory:
  - If Mod=11 (byte $\geq$ 0xc0), R/M specifies a register
  - Else if R/M = 100, see next slide
  - Else, register addr maybe with 8 or 32-bit displacement
- The Reg field is the other operand, or a sub-opcode

Example ModR/M addressing modes

- Opcode 0xff/000 means 32-bit increment
- `ff 00 (00 000 000)    : incl (%eax)`
- `ff 01 (00 000 001)    : incl (%ecx)`
- `ff 40 (01 000 000) 05: incl 5(%eax)`
- `ff c0 (11 000 000)    : incl %eax`
- `ff c1 (11 000 001)    : incl %ecx`

The SIB byte

- More complex addressing modes use another byte
- "SIB": 2-bit scale, 3-bit index register, 3-bit base register
- Base and index are added together, with the index multiplied by 1, 2, 4, or 8
  - Think: array indexing
- Base or index can also be omitted

Outline

x86-32 Overview

x86-32 Arithmetic Basics

x86-32 to x86-64

In Compiler Explorer

LEA for arithmetic

- The computations used for addressing modes are also available as a separate instruction `lea`
- No memory access, just stores computed value in another register
- Why?
  - Addition of registers with constants and small multiples
  - Three-address, unlike regular arithmetic
  - Does not set condition codes

8 core binary operators

- Opcodes `0x[0123][01234589abcd]` and `0x8[0123]`
- In encoding order: `add`, `or`, `adc` ("add with carry"), `sbb` ("subtract with borrow"), `and`, `sub`, `xor`, `cmp`
- `cmp` is like `sub`, but the result is discarded, useful only for flags

Shift-family operations

- Opcodes 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3
- In encoding order: `rol`, `ror`, `rcl`, `rcr`, `shl/sal`, `shr`, `(sal)`, `sar`
- The amount operand can be:
  - An 8-bit immediate (0xc0 and 0xc1)
  - One position (0xd0 and 0xd1)
  - The low bits of `ecx` (0xd2 and 0xd3)

Shift-family operations (cont'd)

- `rol`, `ror` are circular bit rotation
- `rcl`, `rcr` are $(n+1)$-bit rotations that also incorporate CF
- `shr` is logical (unsigned) right shift, while `sar` is arithmetic (signed) right shift
- There is no logical/arithmetic distinction for left shift, and only the 100 position is documented
  - 110, which would be `sal`, is an undoc. synonym of 100

Unary-family operations

- Opcodes 0xf6, 0xf7, 0xfe, and 0xff encode several arithmetic operators with only a ModR/M operand
- `inc` and `dec` are increment and decrement
- `not` is bitwise not and `neg` is unary negation

Multiplication

- Widening multiply has unsigned (`mul`) and signed (`imul`) versions, and a unary encoding:
  - One factor is always in `eax`
  - The other factor is a register or memory location
  - The product is in `edx:eax`
- Same-size `imul` also has more flexible binary encodings

Division and remainder

- Division and remainder are always computed together
- There are unsigned (`div`) and signed (`idiv`) versions, with a unary encoding:
  - The dividend is in `edx:eax`
  - The divisor is a register or memory location
  - The quotient is in `eax`
  - The remainder is in `edx`

x87-style floating point

- In the 8086-80386 era, hardware floating point required a separate chip
  - The 8087 was more transistors and more expensive than the 8086
- Pioneering but now-unusual design
  - 80-bit extended register size
  - Stack-structured register file
- Opcodes 0xd8-0xdf, mnemonics starting with "f"

SIMD extensions

- Since the Pentium era, repeated extensions have added SIMD support
  - Single Instruction Multiple Data: wide registers treated like small arrays
  - MMX, SSE, AVX
- Mostly separate register file and instructions
- Two and three-byte opcodes with 0x66, 0xf2, and 0xf3 reused to specify operand size

Outline

x86-32 Overview

x86-32 Arithmetic Basics

x86-32 to x86-64

In Compiler Explorer

x86-64 extension overview

- Extended registers to 64 bits
- 64-bit versions of most operations
- Main use case was 64 bit pointers, but still 32-bit ints
- Doubled number of GPRs from 8 to 16
  - Most RISC ISAs have 32
- Mostly backwards-compatible

REX encoding

- How to signal 64-bit ops, and name new registers?
- Switch opcodes 0x40-0x4f into a new kind of prefix byte with four extra bits
- Bit 3 is set to 1 (e.g. 0x48) to indicate a 64-bit operation
- Other bits become the 4th bit of register numbers, i.e. set means new registers

x86-64 registers

- All the new register names start with "r"
- x86-32 "e" registers extend to 64-bit by changing "e" to "r"
- The new registers are r8 through r15
- Low 32-, 16-, and 8-bit parts are available with more systematic names
  - `d`, `w`, or `l` suffix

Implicit zero extension

- Operations on 8- and 16-bit subregisters leave the rest unchanged
  - Convenient for storing other data in high half
- 32-bit operations in x86-64 are different: they always set the high half to zero
  - Convenient for mixing 32-bit and 64-bit computations
- Exception: 0x90 ("xchg eax, eax") is still a no-op

Outline

x86-32 Overview

x86-32 Arithmetic Basics

x86-32 to x86-64

In Compiler Explorer