

Contents

bit level manipulations	1
integers	1
casting integers	1
byte order	2
history	2
assembly/machine code view	2

bit level manipulations

- binary: get more precision over n-ary or smth
- and (&), or (|), not (~), xor (^)
- shifts
 - $x \ll y$
 - throw away extra bits at left
 - fill with 0s on right
 - $x \cdot 2^y$
 - $x \gg y$
 - throw away extra bits at right
 - unsigned shift: uses logical shift: fill with 0s on left
 - signed shift: uses arithmetic shift: replicate sign bit on left
 - *undefined*: shift amtn < 0 or \geq word size
 - $\lfloor x / 2^y \rfloor$ i.e. rounding to left
 - to round to zero ($\lceil x / 2^y \rceil$): $(x + (1 \ll y) - 1) \gg y$
- logical &&, ||, !
 - views 0 as false, nonzero as true
 - returns 0 or 1

integers

- limits
 - $U_{\max} = 2^w - 1$
 - $T_{\min} = -2^{w-1}$
 - $T_{\max} = 2^{w-1} - 1$
- $-x = \sim x + 1$ in twos complement
 - but if $x = T_{\min}$ (most negative two's complement), you get back T_{\min}

casting integers

- constants are signed ints by default
 - specify 10U for unsigned or 24L for long
 - source of mistakes: make sure to, eg, `1ULL << 36`
- signed \leftrightarrow unsigned: maintain bit pattern
 - may add/subtract 2^w (0b1000 is 8 unsigned, -8 signed.)
 - casting to larger? sign extend.
 - casting to smaller? drop significant bits.
- mix of signed and unsigned in expression (eg ==)? implicitly casted and eval'd in unsigned.

byte order

...	0x100	0x101	0x110	0x111	...
...	01	23	45	67	...

Table 1: big endian

...	0x100	0x101	0x110	0x111	...
...	67	45	23	01	...

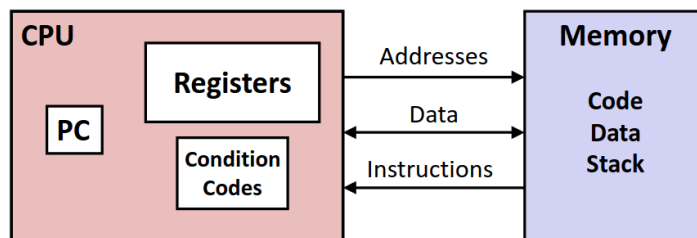
Table 2: little endian

history

- intel x86 processors
 - a Complex Instruction Set Computer (CISC), lots of instructions
 - Reduced: (RISC) can be fastish but esp good for low power
- architecture: processor design spec?? needed to know how to write assembly/machine code??
- microarchitecture: implementation of architecture
- machine code: byte-level programs processors exec.
- assembly code: text readable machine code

assembly/machine code view

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called "RIP" (x86-64)
- **Register file**
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

14

- integer registers: prof: "compiler %rsp 64 bit, %esp 32 bit, compiler will spit out whichever is smaller and fits your data so b careful." also stuff like "%eax vs %ax vs %ah/%al"

movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4,%rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

21

- lea instruction
 - intended to calculate pointer to obj: eg array elem
 - compiler authors end up using it to do arithmetic
 - doesn't touch condition codes
- which registers are pointers?
 - %rsp (top of stack pointer) %rip (current instruction/program counter pointer) always pointers
 - pointers near stack pointer or program counter pointer *probably* also pointers.
 - mov (%rsi), %rsi: register used as pointer? value is probably pointer.
 - (%rsi, %rbx) one of these is a pointer, don't know which
 - (%rsi, %rbx, 2) rsi is a pointer, not rbx (why?)
 - 0x400570(, %rbx, 2) 0x is pointer, not rbx (why?) (assume blank, is 0)
 - lea (anything), %rax idk bro
- control flow
 - lots of GOTOs. c0vm moment
- condition codes (status of recent tests): CF, ZF, SF, OF
 - set as side effect of arithmetic
 - Carry Flag: set if carry from unsigned overflow (or borrowing a 1 to make 0x0 - 0x1 work)
 - Zero Flag: get a 0
 - Sign Flag: t < 0
 - Overflow Flag: signed overflow
 - in GDB as eflags register (a flag isn't showing up? is set to 0.)
 - compare instruction (cmp)

- computes $b-a$ without setting b , unlike `sub`
- used for `if` statements
- ▶ `test` instruction
 - computes $b \& a$ (like `and`) without setting b
 - used to compare `%rX` to 0 (`test %rX %rX`)
 - used to check if 1-bits are same in two registers, like normal `&` usage
- ▶ `j...` instructions: jump to different parts depending on condition codes
 - `jmp`, `je`, `jne`, `jg`, `jge`, etc
- ▶ `set...` these exist if