Contents

Bits, Bytes, & Integers

bit level manipulations

- binary: get more precision over n-ary or smth
- and (&), or (|), not (~), xor (^)
- shifts
  - `x << y`
    - throw away extra bits at left
    - fill with `0`s on right
    - $x \cdot 2^y$
  - `x >> y`
    - throw away extra bits at right
    - unsigned shift: uses logical shift: fill with `0`s 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<<y)-1) >> y`
- logical &&, `||`, `!`
  - views 0 as false, nonzero as true
  - returns 0 or 1

integers

- limits
  - `UMax` $= 2^w - 1$
  - `TMin` $= -2^{w-1}$
  - `TMax` $= 2^{w-1} - 1$
- `-x = ~x + 1` in twos complement
  - but if `x = Tmin` (most negative two's complement), you get back `Tmin`

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 $\longleftrightarrow$ unsigned: maintain bit pattern
  - may add/substract 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 evaled in unsigned.

byte order

…	`0x100`	`0x101`	`0x110`	`0x111`	…
…	`01`	`23`	`45`	`67`	…

Table 1: big endian

…	`0x100`	`0x101`	`0x110`	`0x111`	…
…	`67`	`45`	`23`	`01`	…

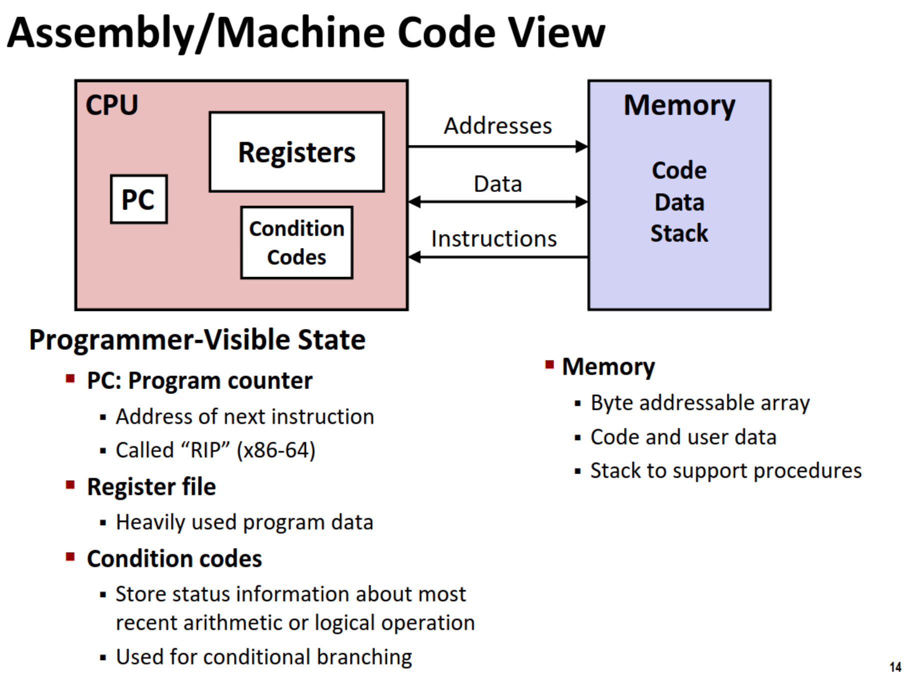
Table 2: little endian

Machine Programming

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



- 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"
- registers
  - eg, `%rax` for full 8 bytes of register, `%eax` right 4 bytes, `%ax` right 2 bytes, splitting further into `%ah` and `%al` for left and right halves of `%ax`.
  - see ref sheet
- memory addressing modes: `D(%Rb, %Ri, S) = Mem[%Rb + (S * %Ri) + D]`
  - `D`: displacement of 1, 2, or 4 bytes. default: 0
  - `Rb`: base register (any of the 16 integer registers)
  - `Ri`: index register (any except %rsp).
  - `S`: scale of 1, 2, 4, or 8. default: 1
- `lea addr dest` instruction
  - sets dest to addr (eg `mov` instead dest to the value at that addr)
  - 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

machine code: control

- 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`) wihtout 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 differnt parts depending on condition codes
    - `jmp, je, jne, jg, jge, etc`
  - `set...` these correspond to `j...` instructions
    - sets only the low-order byte to 0 or 1 based on `...`
    - usually use `movzbl` to alter remaining unaltered bytes.
- `cmov` conditional move
- for `val = Test ? Then_Expr : Else_Expr;`
  - used only when safe; both branches are computed
  - avoid bad performance, side effects, unsafety
- loops exist.
  - do while, while, for loops (beginning conditional is often optimized away)
- switch statements: jump tables
  - notice a jump to a jump table: `jmpq *0x4007f0(,%rdi,8)` and a bunch of suspicious `rets`
  - to inspect a jump table in gdb, `x /8xg 0x4007f0` (8 outputs, hex, giant aka quad word)

machine code: procedures

- need to be able to
  - pass control to procedure and back to return
  - pass data: args, return
  - manage stack memory

- these are implemented via machine instructions, which are defined by Application Binary Interface (ABI)
- the stack
  - grows top to down, bottom address is the "top" of stack: %rsp
- first 6 args in registers, rest on stack
- note: things have alignments and sizes. same for primitives. for complex (structs, arrays), alignments are of their largest component
- note: when we see `sub $0x18, %rsp` for eg, we are "allocating" 18 bytes into the stack. or something. we will pop from stack later by adding 18 back to rsp.

machine code: data

- going over arrays (1d, nd, multilevel), structs (alloc, access, alignment), floats
- arrays
  - contiguous region of `length * sizeof(type)` bytes
  - therefore, nd arrays get row-major ordered
- structs
  - fields ordered to declaration
  - each element must satisfy its own alignment requirement
  - entire struct (inital addr and its size) must be aligned to largest element's size
  - tldr just save space by larger data types first
- alignment restrictions
  - 1 byte: `char, ...`
    - no restrictions on address
  - 2 bytes: `short, ...`
    - lowest 1 bit of address must be 0_2
  - 4 bytes: `int, float, ...`
    - lowest 2 bits of address must be 00_2
  - 8 bytes: `double, long, char *, ...`
    - lowest 3 bits of address must be 000_2
- floating point
  - args passed in %xmm0, %xmm1, ...
  - result in %xmm0
  - all XMM registers call-clobbered