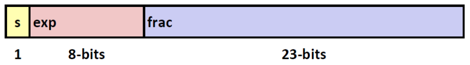
`a | b = ~(~a & ~b)`

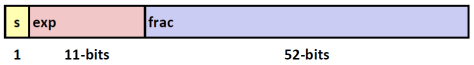
`a ^ b = (a & ~b) | (~a & b)`

b = 1 byte, w = 2 bytes, l = 4 bytes, q = 8 bytes

≈ 7 decimal digits, $10^{\pm 38}$



≈ 16 decimal digits, $10^{\pm 308}$



Normalized value (exp ≠ 000…0 and exp ≠ 111…1)

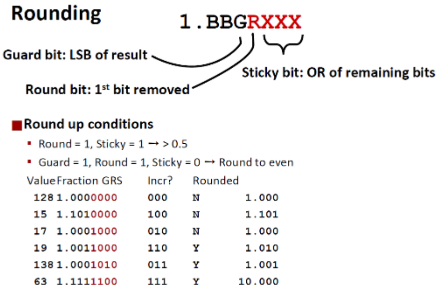
E = Exp – Bias. Bias = $2^{k-1} - 1$

Denormalized Value (exp = 000…0)

Exponent value: E = 1 – Bias (**equispaced**)

Infinity: exp = 111…1, frac = 000…0

NaN: exp = 111…1, frac ≠ 000…0



Round up conditions

- Round = 1, Sticky = 1 → > 0.5
- Guard = 1, Round = 1, Sticky = 0 → Round to even

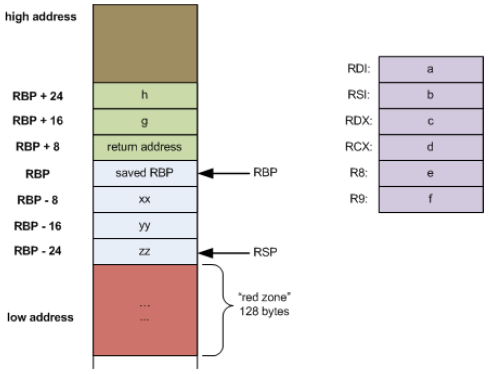
Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

x86-64 linux calling convention:

Integer parameters:

`%rdi, %rsi, %rdx, %rcx, %r8 and %r9`

Others are stored in stack, pushed in reversed (right-to-left) order



CF Carry Flag (for unsigned) **SF** Sign Flag (for signed)

ZF Zero Flag **OF** Overflow Flag (for signed)

Implicitly set by arithmetic operations (but **not set by `leaq` instruction**):

`addq Src DestDest (t = a + b)`

CF set if carry out from most significant bit (unsigned overflow)

ZF set if t == 0

SF set if t < 0 (as signed)

OF set if two's complement (signed) overflow

Rules for turning on the carry flag

1. The carry flag is set if the addition of two numbers causes a carry out of the most significant bits added.
   1111 + 0001 = 0000 (carry flag is turned on)
2. The carry (borrow) flag is also set if the subtraction of two numbers requires a borrow into the most significant (leftmost) bits subtracted
   0000 - 0001 = 1111 (carry flag is turned on)

Rules for turning on the overflow flag

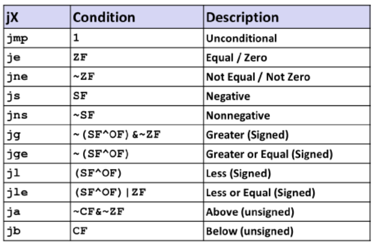
1. If the sum of two numbers with the sign bits off yields a result number with the sign bit on
   0100 + 0100 = 1000 (overflow flag is turned on)
2. If the sum of two numbers with the sign bits on yields a result number with the sign bit off
   1000 + 1000 = 0000 (overflow flag is turned on)

Note that different from above (1111 + 0001 = 0000), the result is correct even though CF is set

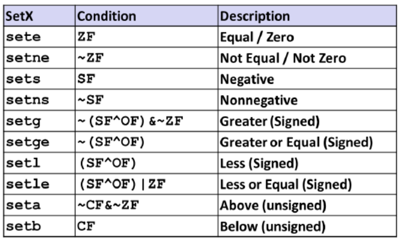
unsigned arithmetic -> CF | signed arithmetic -> OF

`cmp b, a` Computes *b* - *a* (just like `sub`). Sets condition codes based on result, but **does not change *b***

`test a, b` Computes *b* ∧ *a* just like `and`. Sets condition codes (only SF and ZF) based on result, but **does not change *b***

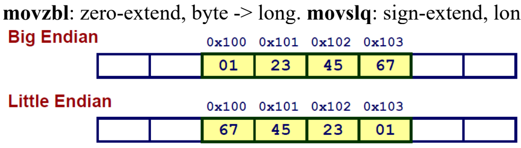


jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)



SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

movzbl: zero-extend, byte -> long. **movslq**: sign-extend, long -> quad. Etc.



Buffer overflow attacks

Stack Smashing Attacks: overwrite normal return address. Code Injection Attacks: overwrite normal return address and jump to exploit code

Measures

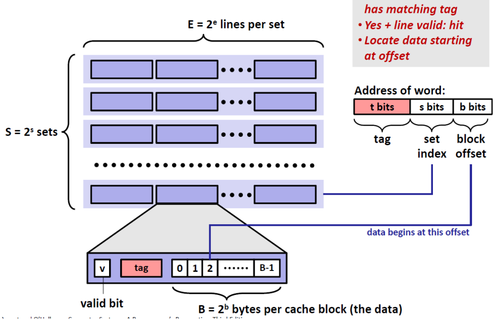
Avoid overflow vulnerabilities: strcpy -> strncpy. Employ system-level protections: randomized stack offsets, nonexecutable code segments. Have compiler use stack canaries

Return-Oriented Programming Attacks

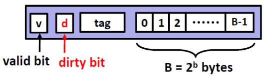
Work around stack randomization and marking stack nonexecutable. Does not overcome stack canaries

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



What about writes?



- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Each cache line needs a dirty bit (set if data has been written to)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location will follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**