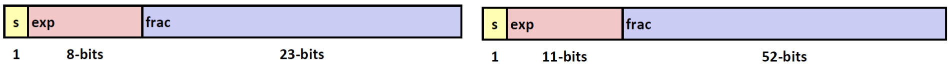
```
a | b = ~(~a & ~b)
a ^ b = (a & ~b) | (~a & b)
```

float and double:

≈ 7 decimal digits, $10^{\pm 38}$ ≈ 16 decimal digits, $10^{\pm 308}$



Normalized value (exp ≠ 000…0 and exp ≠ 111…1)
E = Exp – Bias
Exp: unsigned value of exp field
Bias = $2^{k-1} - 1$, k = # of exponent bits. Single precision: 127 (Exp: 1…254, E: -126…127); Double precision: 1023 (Exp: 1…2046, E: -1022…1023)
Frac = 1.xxx…x

Denormalized Value (exp = 000…0)
Exponent value: E = 1 – Bias (instead of E = 0 – Bias because smallest normalized value has Exp = 1, **equispaced**)
Frac = 0.xxx…x2
exp = 000…0, frac = 000…0 represents zero. There are also –0

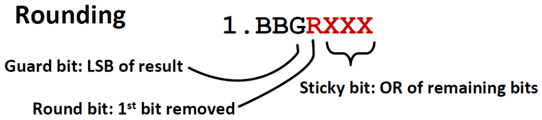
Infinity: exp = 111…1, frac = 000…0
NaN: exp = 111…1, frac ≠ 000…0

Round To Even

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Rounding



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

movb, movw, movl, movq
b = 1 byte, w = 2 bytes, l = 4 bytes, q = 8 bytes

CF Carry Flag (for unsigned) **SF** Sign Flag (for signed)
ZF Zero Flag **OF** Overflow Flag (for signed)

Implicitly set (as side effect) of 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

In unsigned arithmetic, use the carry flag
In signed arithmetic, use the overflow flag

cmp Instruction

`cmp b, a`

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

test instruction

`test a, b`

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

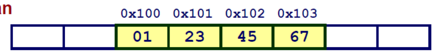
Most common use: `test x, x`

to compare `x` to zero

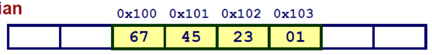
jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below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

Big Endian



Little Endian



Buffer overflow attacks

Stack Smashing Attacks: overwrite normal return address A with address of some other code S. When Q executes ret, will jump to other code

Code Injection Attacks: input string contains byte representation of executable code, overwrite return address A with address of buffer B, when Q executes ret, will jump to exploit code

Measures

Avoid overflow vulnerabilities: strcpy -> strncpy

Employ system-level protections: Randomized stack offsets, Nonexecutable code segments

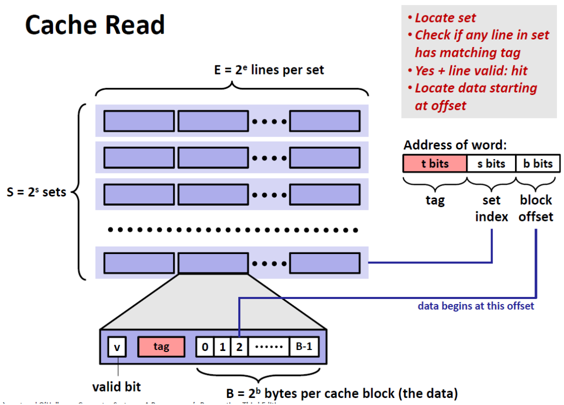
Have compiler use stack canaries

Return-Oriented Programming Attacks

Work around stack randomization and marking stack nonexecutable

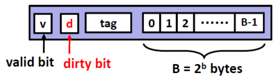
Does not overcome stack canaries

Cache Read



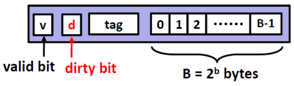
What about writes?

- **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**

Practical Write-back Write-allocate



- **A write to address X is issued**
- **If it is a hit**
  - Update the contents of block
  - Set dirty bit to 1 (bit is sticky and only cleared on eviction)

- **If it is a miss**
  - Fetch block from memory (per a read miss)
  - The perform the write operations (per a write hit)

- **If a line is evicted and dirty bit is set to 1**
  - The entire block of 2^b bytes are written back to memory
  - Dirty bit is cleared (set to 0)
  - Line is replaced by new contents