BITS | BYTES | INTEGERS

C Data Type	size (bytes)
char	1
short	2
int	4
long (long)	8 (8)
float	4
double	8
pointer	8

(XOR)

^	0	1
0	0	1
1	1	0

lowest rank
highest rank

LEFT SHIFT (x<<y)

b_6	b_5	b_4	b_3	b_2	b_1	b_0

<<1

b_5	b_4	b_3	b_2	b_1	b_0	0

→ equiv. to $x * 2^y$

RIGHT SHIFT (x>>y)

b_6	b_5	b_4	b_3	b_2	b_1	b_0

>>1

b_6	b_6	b_5	b_4	b_3	b_2	b_1

→ equiv. to $\lfloor x/2^y \rfloor$

a) Logical Right shift
→ fill left w/ 0's
→ unsigned values
→ rounds toward 0

Note: $(-11/4) > (-11>>2)$

b) Arithmetic Right shift
→ fills left w/ MSB
→ signed values
→ rounds toward $-\infty$
↳ compute $\lfloor (x+2^y-1)/2^y \rfloor$

	value (hex)	values (wrt w)
unsigned max	0xFFFFFFFF	2^w-1
2's compl. max	0x7FFFFFFF	$2^{w-1}-1$
2's compl. min	0x80000000	-2^{w-1}

↳ $-Tmin = Tmin$

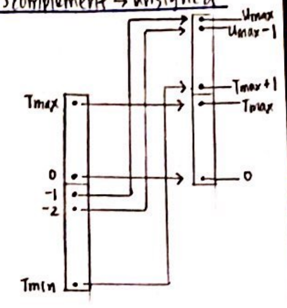
(Negation) Identity: $\sim x + 1 == -x$
↳ except for $x = Tmin$

LITTLE ENDIAN: store object in memory least significant byte first.
eg. int x = 0x01234567; (stored at address 0x100)

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

↑ lowest order byte has lowest address

2's complement → unsigned



General Rules:

1. constants are signed values
   ↳ explicit cast: (int) x;
   ↳ implicit cast: 8u
2. expressions w/ signed and unsigned: (arithmetic / comparisons)
   → signed values implicitly casted to unsigned
3. converting small data type to larger (same sign):
   ⇒ sign extension: make copies of sign bit
   ↳ unsigned values pad w/ 0's
   eg. char c = 0xF5;
   ↳ real value of c = 0xFFFFFFF5;
4. converting large data type to smaller (same sign):
   ⇒ truncation: drop top (higher order) bits
5. 2's complement wraps around both ends (+/-)
   ↳ if $x \geq 2^{w-1}$: becomes negative
   ↳ if $x < -2^{w-1}$: becomes positive
   } overflow occurs when both ⊕ or both ⊖
6. casting (int ↔ float/double)
   i) int → double (53 mantissa)
      · exact conversion
      · no bits lost
   ii) int → float (23 mantissa)
      · will round (bits lost)
      · correct magnitude
      · incorrect precision
   iii) double/float → int
      · truncates fractional part
      · like rounding towards 0
      · not defined when out of range, or NaN
      ⇒ usually sets to Tmin

IEEE floating point values

s	exp	frac

; bias = $2^{k-1}-1$ (# of exp bits)

denormalized
· when exp = 00...000
↳ E = 1 - bias
↳ frac has implied leading 0.
① if frac = 00...000
⇒ represents +/- 0
② if frac ≠ 00...000
⇒ represents small values

normalized
exp ≠ 00...00 and exp ≠ 11...111
↳ E = exp - bias
↳ frac has implied leading 1

special values
when exp = 11...111
① if frac = 00...000
→ represents +/- ∞
② if frac ≠ 00...000
→ represents NaN

I. Single precision (32 bits) (float)

s	exp	frac
1	8 bits	23 bits

↳ bias = 127

II. Double Precision (64 bits) (double)

s	exp	frac
1	11 bits	52 bits

↳ bias = 1023
→ exp: 1...2046
→ E: -1022...1023

7. a) unsigned int x = 0xFF000000 ⇒ exp: 1...254
   unsigned short y = 0xFFFF ⇒ E: -126...127
   ↳ x > (signed short) y
   b) unsigned short x = 0xffff;
   unsigned short y = 1;
   unsigned int z = x + y;
   ↳ z = 0
   ① y is casted to signed short
   ② b/c y is now signed when sign extension occurs, FF.. gets copied
8. modular addition
   ↳ u+v (mod 2^w)
   ↳ signed (drops MSB)
   ‖ modular mult (unsigned)
   ↳ u·v (mod 2^w)
9. strings don't follow little endian
   ↳ bytes ordered normally

Rounding (to even)

x = BBG | RXXXXX

guard: LSB of result
round: 1st bit removed
sticky: OR of the remaining bits

if ((guard && round) || (round && sticky))
round up (+1 to number)

ASSEMBLY

First 6 arguments stored in registers:
%rdi, %rsi, %rdx, %rcx, %r8, %r9

Return value stored in %rax

set X	j X	condition	Description
sete	je	ZF	equal/zero
setne	jne	~ZF	not equal/not zero
sets	js	SF	negative
setns	jns	~SF	nonnegative
setg	jg	~(SF^OF)&~ZF	greater (signed)
setge	jge	~(SF^OF)	greater/equal (signed)
setl	jl	SF^OF	less (signed)
setle	jle	(SF^OF) | ZF	less/equal (signed)
seta	ja	~CF&~ZF	above (unsigned)
setb	jb	CF	below (unsigned)
	jmp	1	unconditional

① CF: carry flag (unsigned)
↳ if add had carry or subtract had borrow
② SF: sign flag (signed)
↳ if answer is negative
③ ZF: zero flag
↳ if answer is 0
④ OF: overflow flag
↳ if +/- overflow in 2's complement

Instruction	Computation
movq (R), %rax	Mem[Reg[R]]
movq D(R), %rax	Mem[Reg[R]+D]
addq source, dest	dest = dest + source
shlq source, dest	dest = dest << source
sarq source, dest	dest = dest >> source
shrq source, dest	dest = dest >> source
incq dest	dest = dest + 1
decq dest	dest = dest - 1
negq dest	dest = -dest
notq dest	dest = ~dest
cmpq src1, src2	src2 - src1 (w/o dest)
testq src1, src2	src2 & src1 (w/o dest)

register R specifies start of memory region (address)

arith. shift

Memory Addressing Mode

D(Rb, Ri, S) ⇒ Mem[Reg[Rb] + S*Reg[Ri] + D]

constant displacement (1,2,4 bytes)
base register
index register (no %rsp)
scale ↳ 1,2,4,8

eg. cmpq b,a
↳ ① CF set: if carry/borrow from MSB
② ZF set: if a == b
③ SF set: if (a-b) < 0 ↳ as signed
④ OF set: if overflow

eg. testq b,a
↳ ① ZF set: if a&b == 0
② SF set: if a&b < 0

Other Instructions:

1. setX dest
   ↳ sets lower order byte of dest to 0 or 1 based on combinations
   ⇒ sets certain flags (can't overwrite register by name)
   ⇒ read flags from condition codes and saves in register for use

→ after setX: need to zero-extend the higher 7 bytes of register
↳ movzbl %al, %eax

%rax	%eax	%al
0000	0000..	

lower 32 bits of register
lower 8 bits of register

Important:
① %rsp decrements by 8 to leave space for return address right before all fxn calls
↳ return addr. is placed after decrement at current location
② push src
i) obtain value of src
ii) decrement %rsp by 8
iii) put val. at curr. %rsp location
③ popq dest
i) obtain value at addr of %rsp
ii) increment %rsp by 8
iii) store value at dest
(%rsp moves, data stays)
④ stack grows down in memory
↳ bottom of stack has highest memory address
↳ %rsp contains lowest memory address (addr. of top element)

⑤ call <label>
1. push return addr. on stack
2. jump to label

⑥ ret
1. pop addr. from stack
2. jump to address

Note: mov dereferences address
leaq doesn't
↳ %rax = 3 * %rax (addr of %rax)
leaq (%rax, %rax, 2)
⇒ computes addr. arithmetic

computes address w/o memory reference

vs. movq (%rax, %rax, 2)
⇒ computes arithmetic w/ contents inside %rax

%rax
%rdi
%rsi
%rdx
%rcx
%r8
%r9
%r10
%r11

%rax: return value
%rdi, %rsi, %rdx, %rcx, %r8, %r9: arguments
%r10, %r11: caller saved temporaries

→ value of registers not guaranteed to be maintained over a call

%rbx
%r12
%r13
%r14
%rbp
%rsp

%rbx, %r12, %r13, %r14: callee saved temporaries
%rbp, %rsp: special

→ if callee (function) wants to use, must save before using, restore when returns } register values are preserved over a call

Arrays

int val[5];

1	5	2	1	3

x, x+4, x+8, x+12, x+16, x+20

	type	value	
val	int*	x	
val[2]	int	2	accessing methods
*(val+2)	int	2	
&val[2]	int*	x+8	addressing methods
val+2	int*	x+8	
val+i	int*	x+(4*i)	