

## Integers

- Example of storing 0x1234567 at 0x100

	0x100	0x101	0x102	0x103
Big endian	01	23	45	67
Little endian	67	45	23	01

- $\lceil x/2^k \rceil$  is given by  $(x + (1 \ll k) - 1) \gg k$
- $x/2^k$  is given by  $(x < 0 ? x + (1 \ll k) - 1 : x) \gg k$

## Bitwise Operations

- Logical shift** – Fills left end with zeros (unsigned)
- Arithmetic shift** – Sign-extends left end (signed)

## Floating Point

- Contains 3 parts **sign**, **exp**, **frac**
- 3 parts of **float**:  $1+8+23=32$ ; **double**:  $1+11+52=64$
- Lacks common  $+$  rules  $a*b*c \neq a*(b*c)$ ,  $a+b+c \neq a+(b+c)$ ,  $a*(b+c) \neq a*b + a*c$
- $V = (-1)^s \times M \times 2^E$
- Sign bit  $s$  1-bit, 0 for  $V \geq 0$ , 1 for  $V < 0$
- Exponent  $E$ ,  $k$ -bit **exp** field
- Significand (mantissa)  $M$  –  $0 \leq M < 1$  or  $1 \leq M < 2$ , represented by  $n$ -bit **frac** field ( $f_{n-1} \dots f_1 f_0$ )
- Normalized values

- **exp** is neither all zeros nor all ones
- $E = e - \text{Bias}$  where  $e$  is the unsigned number in **exp** and  $\text{Bias} = 2^{k-1} - 1$
- **frac** represents  $0 \leq f < 1$  with  $0.f_{n-1} \dots f_1 f_0$  and  $M = 1 + f$

1	10000000	1010101010101010101010101
$(-1) \times 2^1 \times 1.1010101010101010101010101$		

- Denormalized values
- **exp** all zeros
- Exponent value is  $E = 1 - \text{Bias}$ ,  $M = f$  (no leading 1)

0	00000000	1010101010101010101010101
$1 \times 2^{1-01111111} \times 0.1010101010101010101010101$		

- Special values
- Exponent field is all ones
- Fraction field all zeros can represent  $\pm\infty$ , depending on sign bit
- Nonzero fraction field is NaN
- $+\infty = 0, 11111111, 000000000000000000000000$
- every conditional expression with NaN is 0
- **NaN == NaN** is 0
- every arithmetic expression with NaN is NaN
- **NaN \* 0** is NaN
- forcibly trans **inf** or NaN to **int** gets  $\text{int}_{\min}$

- Rounding
- Rounds to the nearest even
- **BBGRXXXX** can round to **BBG** or **BBG+1**
- **G** – Guard bit; least significant bit of result
- **R** – Round bit; first bit removed
- **XXXX** – Sticky bit; OR of remaining bits
- Round up conditions:
  - \* Round = 1, Sticky = 1  $\rightarrow$  **BBG+1**
  - \* Round = 0  $\rightarrow$  **BBG**
  - \* Guard = 1, Round = 1, Sticky = 0  $\rightarrow$  **BBG+1**
  - \* Guard = 0, Round = 1, Sticky = 0  $\rightarrow$  **BBG**

- Multiplication
- $(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$   

$$= (-1)^{s_1 \oplus s_2} (M_1 \times M_2) 2^{E_1 + E_2}$$
- If  $M_1 \times M_2 \geq 2$ ,  $M = M_1 \times M_2 / 2$ ,  $E = E_1 + E_2 + 1$
- If  $E$  out of range, overflow to **inf**
- Round  $M$  to fit **frac** precision

## Assembly Basics

- “word” refers to 16-bit data type, “double word” refers to 32-bit (int) and “quad words” refers to 64-bit
- On 64-bit machines pointers are 8-byte quad words
- In operands, scaling factor  $s$  must be either 1, 2, 4, or 8
- mov S, D** has the effect of  $S \rightarrow D$

movq Src, Dest	C Analog
movq \$0x4, %rax	tmp = 0x4
movq \$-147, (%rax)	*p = -147
movq %rax, %rdx	tmp2 = tmp1
movq %rax, (%rdx)	*p = tmp
movq (%rax), %rdx	tmp = *p

- movzbq** moves from byte to quad with zero-extended whereas **movsbq** does the same but sign-extended
- leaq S, D** has the effect of  $\&S \rightarrow D$
- subq S, D** has the effect of  $D - S \rightarrow D$
- salq S, D** has the effect of  $D \cdot 2^S \rightarrow D$

Opcodes of arithmetic operations

addq	+	xorq	$\oplus$	salq	also shlq <<
subq	-	andq	$\&$	sarq	Arithmetic >>
imulq	$\times$	orq		shrq	Logical >>
incq	++	decq	--	negq	-
				notq	$\sim$

Operands (3 types)

Type	Form	Operand value
Immediate	$\$Imm$	$Imm$
Register	$r_a$	$R[r_a]$
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$

16 general purpose registers storing 64-bit values

Type	64-bits	32-bits	16-bits	8-bits
Registers below are <b>caller saved</b>				
Return val	%rax	%eax	%ax	%al
1st arg	%rdi	%edi	%di	%dil
2nd arg	%rsi	%esi	%si	%sil
3rd arg	%rdx	%edx	%dx	%dl
4th arg	%rcx	%ecx	%cx	%cl
5th arg	%r8	%r8d	%r8w	%r8b
6th arg	%r9	%r9d	%r9w	%r9b
Caller	%r10	%r10d	%r10w	%r10b
Caller	%r11	%r11d	%r11w	%r11b
Registers below are <b>callee saved</b>				
Callee	%rbx	%ebx	%bx	%bl
Callee	%r12	%r12d	%r12w	%r12b
Callee	%r13	%r13d	%r13w	%r13b
Callee	%r14	%r14d	%r14w	%r14b
Callee	%r15	%r15d	%r15w	%r15b
Callee	%rbp	%ebp	%bp	%bpl
Stack ptr	%rsp	%esp	%sp	%spl

**%rsp** is the **top of the stack** and **%rbp** is the **bottom**

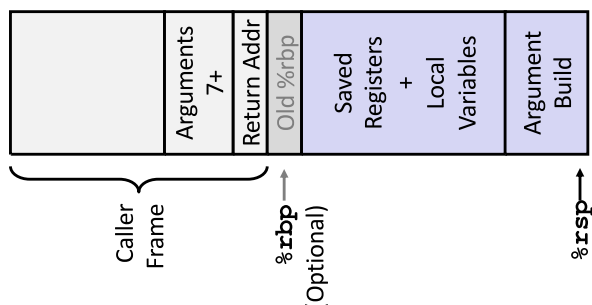
## Conditional Control

- Carry flag (CF) – most recent op generated carry of most significant bit, detects overflow for unsigned
- Zero flag (ZF) – most recent op yielded zero
- Sign flag (SF) – most recent op yielded negative value
- Overflow flag (OF) – most recent op caused two’s complement overflow
- test** instruction behaves like **and** instructions but sets condition codes without altering source or destination often see **testq %rax, %rax** to check if return val is neg, zero, or pos
- cmp** instruction behaves like **sub** but sets condition codes without altering source or destination.
- Switch statement**: jump table stores the address of the target label in an array. **jump[i]  $\rightarrow$  codeblock (x==i)**

set D and jmp suffixes

Instruction	Alias.	Cond.	Desc.
-e	-z	ZF	= /0
-ne	-nz	~ZF	!= /not zero
-s		SF	Neg
-ns		~SF	Nonneg
-g	-nle	~(SF^OF)&~ZF	signed >
-ge	-nl	~(SF^OF)	signed ==>
-l	-nge	SF^OF	signed <
-le	-ng	(SF^OF)   ZF	signed <=
-a	-nbe	~CF&~ZF	unsigned >
-ae	-nb	~CF	unsigned >=
-b	-nae	CF	unsigned <
-be	-na	CF   ZF	unsigned <=

cmpq B, A, jg L1 equals to if (A > B) goto L1;  
setne %al sets %al to 1 if not equal, 0 otherwise



## Machine Data

**Data Alignment:** K-byte object at addr multiple of K

Size (bytes)	Types
1	char
2	short
4	int, float
8	long, double, char *

**Arrays:** Element i at address &A[0] + i \* sizeof(type)

- 1D array access: movl (%rdx,%rcx,4),%eax gets A[%rcx] for int A[]
- 2D row-major: A[i][j] at &A + i\*C\*s + j\*s for A[R][C] with size s
- Multi-level arrays: Array of pointers e.g., int \*A[R] for jagged arrays

**Structs:** Fields stored with alignment padding

- K-byte field must be at addr multiple of K
- Field alignment may require padding between fields
- Struct alignment = max alignment of any field
- Struct size = multiple of largest alignment requirement
- Example: struct {char a; int b; char c;} →

12 bytes	0	1	2	3	4	5	6	7	8	9	10	11
	a		pad			b			c		pad	

- Nested structs: inner struct follows its own alignment rules
- Access field at offset d: movq d(%rdi),%rax

**Unions:** All fields share same memory location

- Size = largest member size
- All fields have offset 0 from base address
- Example: union {int i; float f; char c[4];}

occupies 4 bytes total	0	1	2	3
				i (4 bytes)
				f (4 bytes)
	c[0]	c[1]	c[2]	c[3]

- Access any member using base address: movl (%rdi),%eax
- Used for type punning (e.g., u.f = 1.0f; bits = u.i) or memory optimization

## GNU Debugger

### Running

- run: Run the program
- kill: Kill the program
- step: Go to next line (Source C code)
- next: Go to next line (Source C code) but step over function calls
- stepi: Go to next instruction (Assembly code)
- continue: Continue running
- finish: Continue until current function returns

### Breakpoints

- break <where>:
  - break <function name>
  - break <file>:<line number>
  - break \*<address>
- delete <num>: eg: delete 2 to delete breakpoint 2
- clear: Delete all breakpoints
- enable / disable <num>: Enable or disable
- info breakpoints / registers / functions / locals / args / displays

### Variables and memory

- print/format <what>:
  - print \$register – print register
  - print \$variable – print variable
  - print \*(0x<address>) – print memory
- x/nfu <address>:
  - n: Number of items to print (default 1)
  - f: Format (see below)
  - u: Size of each item:
    - \* b: Byte
    - \* h: Halfword (2 bytes)
    - \* w: Word (4 bytes)
    - \* g: Giant (8 bytes)

### Format for printing

- a: Pointer
- c: Read as integer, print as character
- d: Integer, signed decimal (10)
- f: Floating point number
- o: Integer, Octal (8)
- s: Try to treat as C string
- t: Integer, binary (2)
- u: Integer, unsigned decimal (10)
- x: Integer, hexadecimal (16)

### Initialize for Bomb Lab

# Compiling into asm (sum.s)  
gcc -Og -S sum.c

# Disassembling Object Code  
objdump -d sum > sum.asm

```
# .gdbinit
break phase_1
break initialize_bomb (send_msg / explode_bomb)
command
jump *(0x4014b9)
end
run ans.txt
```

Created at Wuhan University in Spring 2025  
by TonyYin & Pittow2

<https://github.com/TonyYin0418/csapp-cheat-sheet>  
Based on <https://git.io/JcZ29>