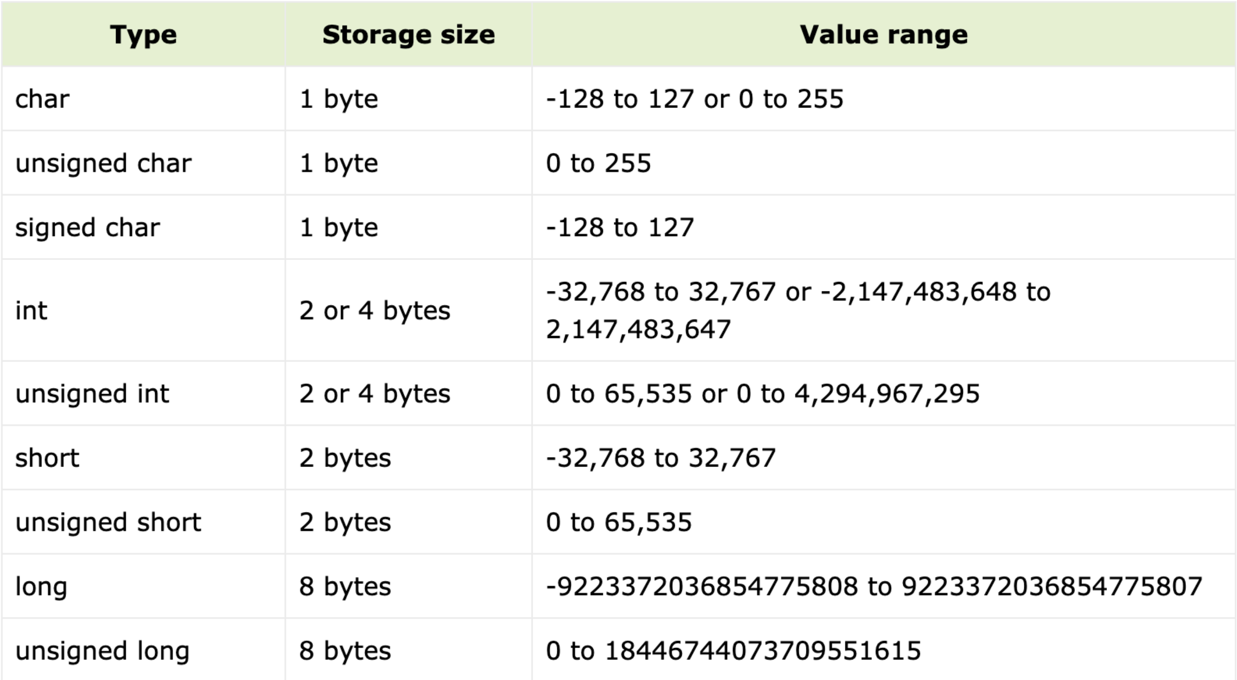
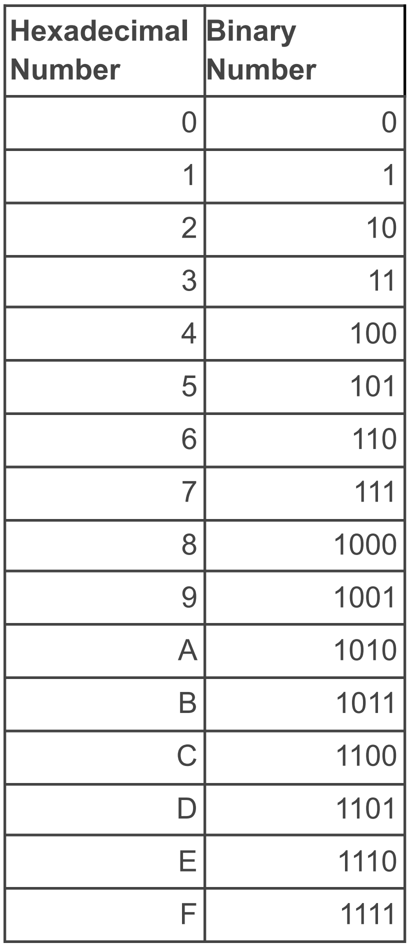
CHEAT SHEET

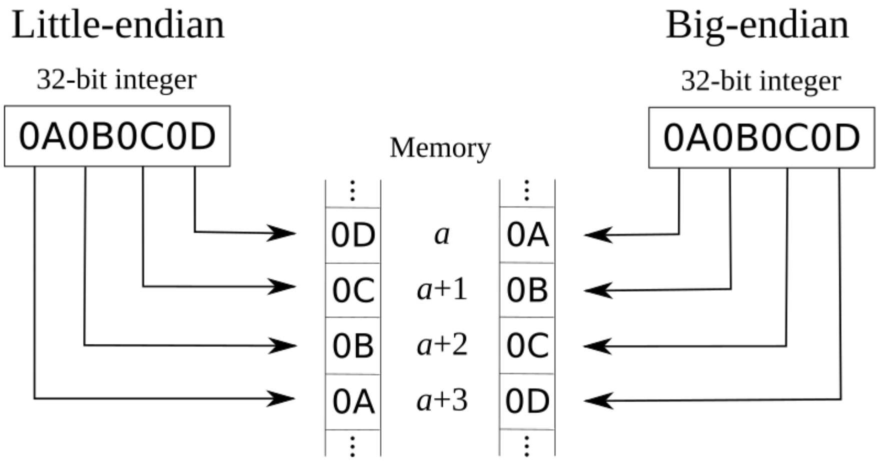
Eric Yoon - CPSC 323 Cheat Sheet



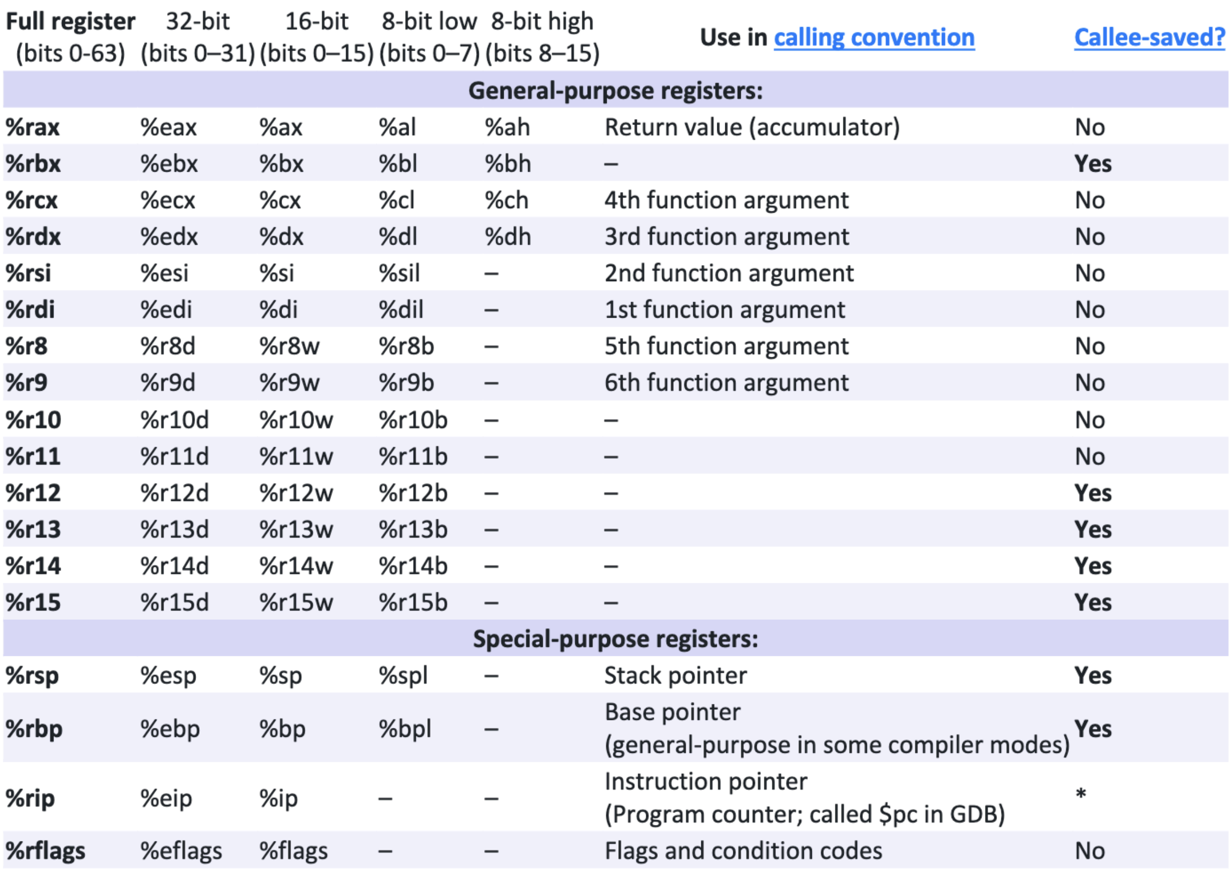
Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615



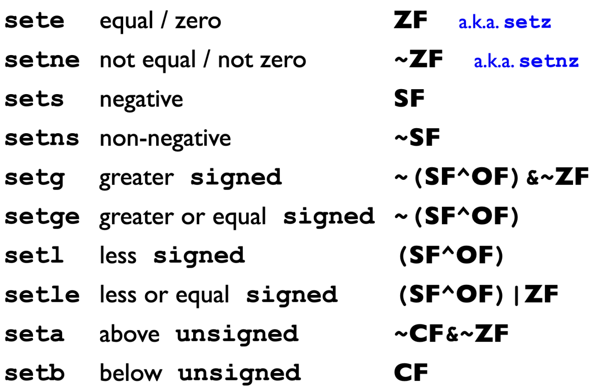
Hexadecimal Number	Binary Number
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111



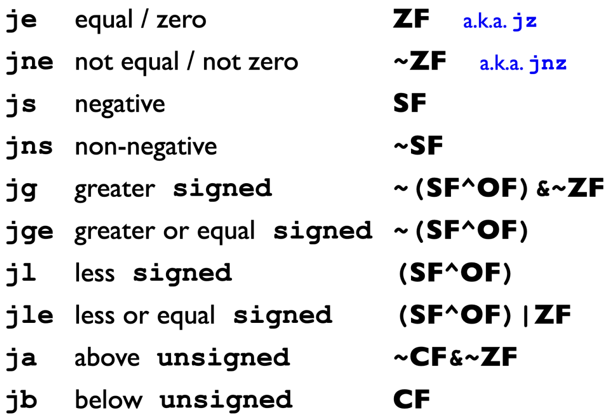
Eric Yoon - CPSC 323 Cheat Sheet



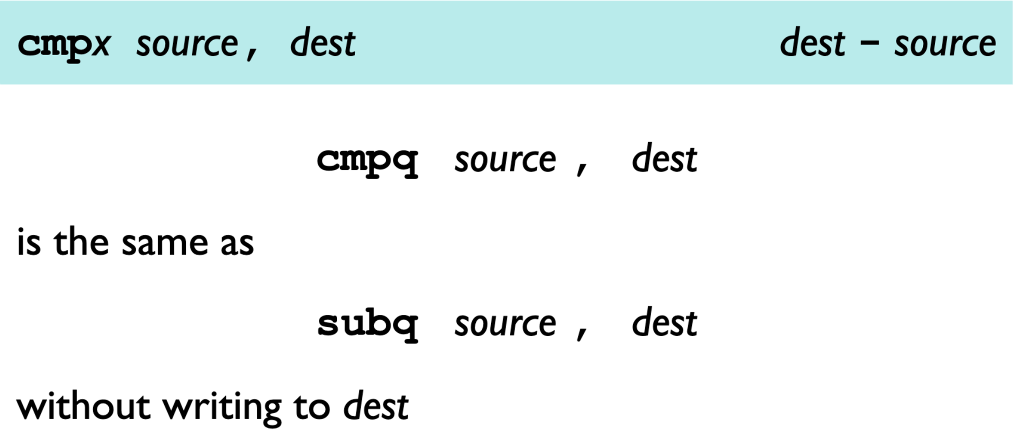
Full register	32-bit	16-bit	8-bit low	8-bit high	**Use in calling convention**	**Callee-saved?**
(bits 0-63)	(bits 0–31)	(bits 0–15)	(bits 0–7)	(bits 8–15)		
					General-purpose registers:	
%rax	%eax	%ax	%al	%ah	Return value (accumulator)	No
%rbx	%ebx	%bx	%bl	%bh	–	**Yes**
%rcx	%ecx	%cx	%cl	%ch	4th function argument	No
%rdx	%edx	%dx	%dl	%dh	3rd function argument	No
%rsi	%esi	%si	%sil	–	2nd function argument	No
%rdi	%edi	%di	%dil	–	1st function argument	No
%r8	%r8d	%r8w	%r8b	–	5th function argument	No
%r9	%r9d	%r9w	%r9b	–	6th function argument	No
%r10	%r10d	%r10w	%r10b	–	–	No
%r11	%r11d	%r11w	%r11b	–	–	No
%r12	%r12d	%r12w	%r12b	–	–	**Yes**
%r13	%r13d	%r13w	%r13b	–	–	**Yes**
%r14	%r14d	%r14w	%r14b	–	–	**Yes**
%r15	%r15d	%r15w	%r15b	–	–	**Yes**
					Special-purpose registers:	
%rsp	%esp	%sp	%spl	–	Stack pointer	**Yes**
%rbp	%ebp	%bp	%bpl	–	Base pointer (general-purpose in some compiler modes)	**Yes**
%rip	%eip	%ip	–	–	Instruction pointer (Program counter; called $pc in GDB)	*
%rflags	%eflags	%flags	–	–	Flags and condition codes	No



sete	equal / zero	**ZF**	a.k.a. **setz**
setne	not equal / not zero	**~ZF**	a.k.a. **setnz**
sets	negative	**SF**	
setns	non-negative	**~SF**	
setg	greater **signed**	**~(SF^OF)&~ZF**	
setge	greater or equal **signed**	**~(SF^OF)**	
setl	less **signed**	**(SF^OF)**	
setle	less or equal **signed**	**(SF^OF)|ZF**	
seta	above **unsigned**	**~CF&~ZF**	
setb	below **unsigned**	**CF**	



je	equal / zero	**ZF**	a.k.a. **jz**
jne	not equal / not zero	**~ZF**	a.k.a. **jnz**
js	negative	**SF**	
jns	non-negative	**~SF**	
jg	greater **signed**	**~(SF^OF)&~ZF**	
jge	greater or equal **signed**	**~(SF^OF)**	
jl	less **signed**	**(SF^OF)**	
jle	less or equal **signed**	**(SF^OF)|ZF**	
ja	above **unsigned**	**~CF&~ZF**	
jb	below **unsigned**	**CF**	



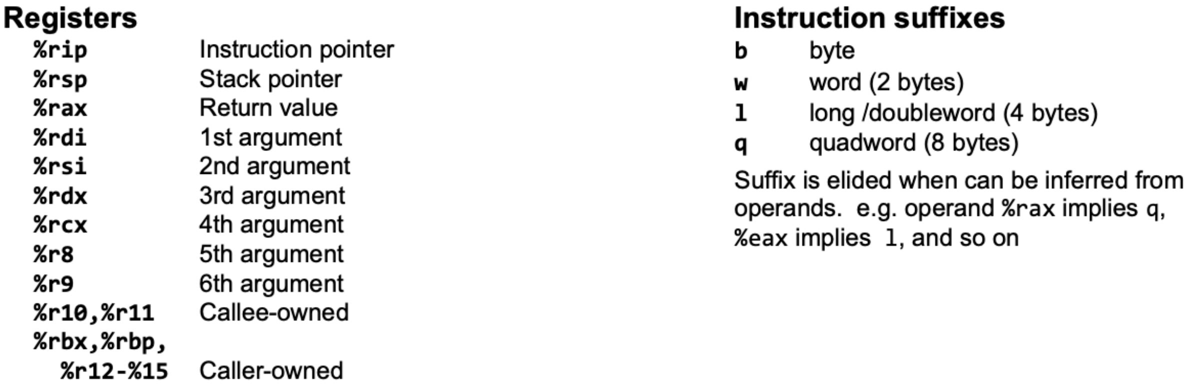
cmp*x* *source* , *dest* *dest* − *source*

cmpq *source* , *dest*

is the same as

subq *source* , *dest*

without writing to *dest*



Registers

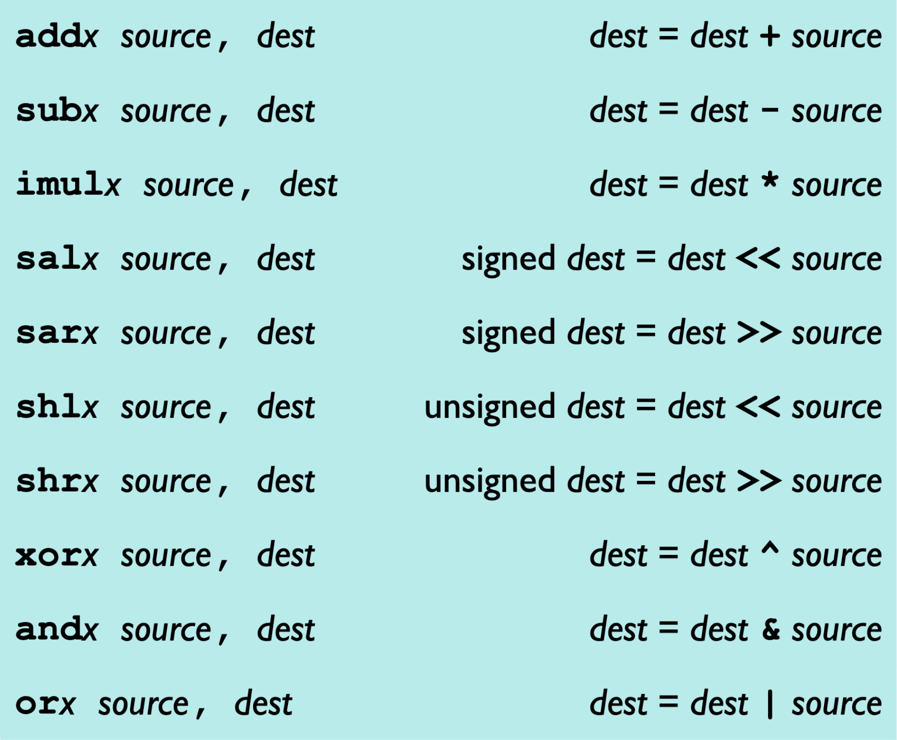
`%rip`	Instruction pointer
`%rsp`	Stack pointer
`%rax`	Return value
`%rdi`	1st argument
`%rsi`	2nd argument
`%rdx`	3rd argument
`%rcx`	4th argument
`%r8`	5th argument
`%r9`	6th argument
`%r10,%r11`	Callee-owned

`%rbx,%rbp,`
`%r12-%15`
	Caller-owned

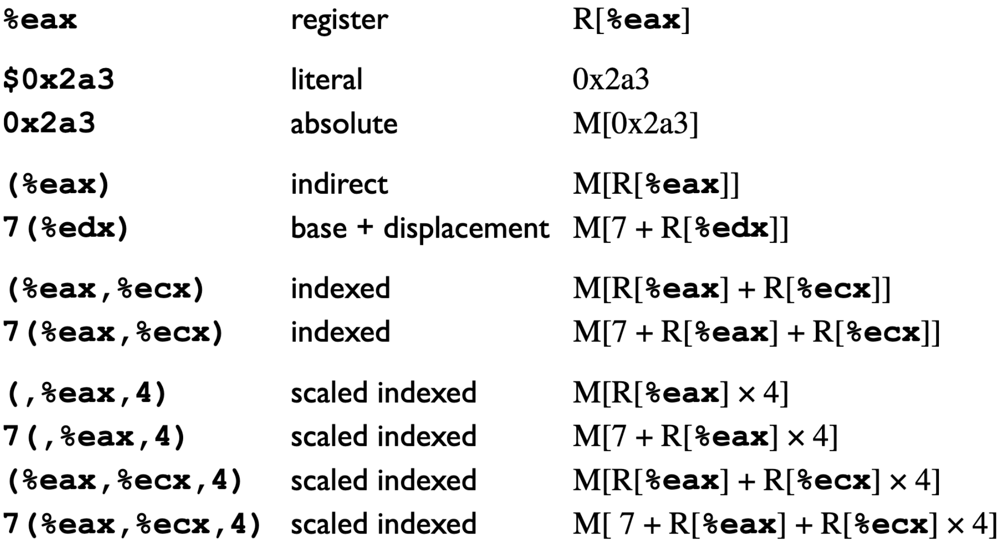
Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long /doubleword (4 bytes)
`q`	quadword (8 bytes)

Suffix is elided when can be inferred from operands. e.g. operand `%rax` implies `q`, `%eax` implies `l`, and so on



add*x* *source*, *dest*	*dest* = *dest* + *source*
sub*x* *source*, *dest*	*dest* = *dest* − *source*
imul*x* *source*, *dest*	*dest* = *dest* * *source*
sal*x* *source*, *dest*	signed *dest* = *dest* << *source*
sar*x* *source*, *dest*	signed *dest* = *dest* >> *source*
shl*x* *source*, *dest*	unsigned *dest* = *dest* << *source*
shr*x* *source*, *dest*	unsigned *dest* = *dest* >> *source*
xor*x* *source*, *dest*	*dest* = *dest* ^ *source*
and*x* *source*, *dest*	*dest* = *dest* & *source*
or*x* *source*, *dest*	*dest* = *dest* | *source*



`%eax`	register	R[`%eax`]
`$0x2a3`	literal	0x2a3
`0x2a3`	absolute	M[0x2a3]
`(%eax)`	indirect	M[R[`%eax`]]
`7(%edx)`	base + displacement	M[7 + R[`%edx`]]
`(%eax,%ecx)`	indexed	M[R[`%eax`] + R[`%ecx`]]
`7(%eax,%ecx)`	indexed	M[7 + R[`%eax`] + R[`%ecx`]]
`(,%eax,4)`	scaled indexed	M[R[`%eax`] × 4]
`7(,%eax,4)`	scaled indexed	M[7 + R[`%eax`] × 4]
`(%eax,%ecx,4)`	scaled indexed	M[R[`%eax`] + R[`%ecx`] × 4]
`7(%eax,%ecx,4)`	scaled indexed	M[7 + R[`%eax`] + R[`%ecx`] × 4]

Eric Yoon - CPSC 323 Cheat Sheet

Common instructions

mov	`src, dst`	dst = src
movsbl	`src, dst`	byte to int, sign-extend
movzbl	`src, dst`	byte to int, zero-fill
cmov	`src, reg`	reg = src when condition holds, using same condition suffixes as jmp
lea	`addr, dst`	dst = addr
add	`src, dst`	dst += src
sub	`src, dst`	dst -= src
imul	`src, dst`	dst *= src
neg	`dst`	dst = -dst (arith inverse)

imulq	`S`	`signed full multiply`
		`R[%rdx]:R[%rax] <- S * R[%rax]`

mulq	`S`	`unsigned full multiply` `same effect as` **imulq**

idivq	`S`	`signed divide`
		`R[%rdx] <- R[%rdx]:R[%rax] mod S`
		`R[%rax] <- R[%rdx]:R[%rax] / S`

divq	`S`	`unsigned divide - same effect as` **idivq**
cqto		`R[%rdx]:R[%rax] <- SignExtend(R[%rax])`
sal	`count, dst`	dst <<= count
sar	`count, dst`	dst >>= count (arith shift)
shr	`count, dst`	dst >>= count (logical shift)
and	`src, dst`	dst &= src
or	`src, dst`	dst |= src
xor	`src, dst`	dst ^= src
not	`dst`	dst = ~dst (bitwise inverse)
cmp	`a, b`	b-a, set flags
test	`a, b`	a&b, set flags
set	`dst`	sets byte at dst to 1 when condition holds, 0 otherwise, using same condition suffixes as jmp
jmp	`label`	jump to label (unconditional)
je	`label`	jump equal ZF=1
jne	`label`	jump not equal ZF=0
js	`label`	jump negative SF=1
jns	`label`	jump not negative SF=0
jg	`label`	jump > (signed) ZF=0 and SF=OF
jge	`label`	jump >= (signed) SF=OF
jl	`label`	jump < (signed) SF!=OF
jle	`label`	jump <= (signed) ZF=1 or SF!=OF
ja	`label`	jump > (unsigned) CF=0 and ZF=0
jae	`label`	jump >= (unsigned) CF=0
jb	`label`	jump < (unsigned) CF=1
jbe	`label`	jump <= (unsigned) CF=1 or ZF=1

push	`src`	add to top of stack Mem[--%rsp] = src
pop	`dst`	remove top from stack dst = Mem[%rsp++]
call	`fn`	push %rip, jmp to fn
ret		pop %rip

Condition codes/flags

ZF	Zero flag
SF	Sign flag
CF	Carry flag
OF	Overflow flag

Addressing modes

Example source operands to **mov**

Immediate

`mov $0x5, dst`

`$val`

source is constant value

Register

`mov %rax, dst`

`%R`

R is register

source in %R register

Direct

`mov 0x4033d0, dst`

`0xaddr`

source read from Mem[0xaddr]

Indirect

`mov (%rax), dst`

`(%R)`

R is register

source read from Mem[%R]

Indirect displacement

`mov 8(%rax), dst`

`D(%R)`

R is register

D is displacement

source read from Mem[%R + D]

Indirect scaled-index

`mov 8(%rsp, %rcx, 4), dst`

`D(%RB,%RI,S)`

RB is register for base

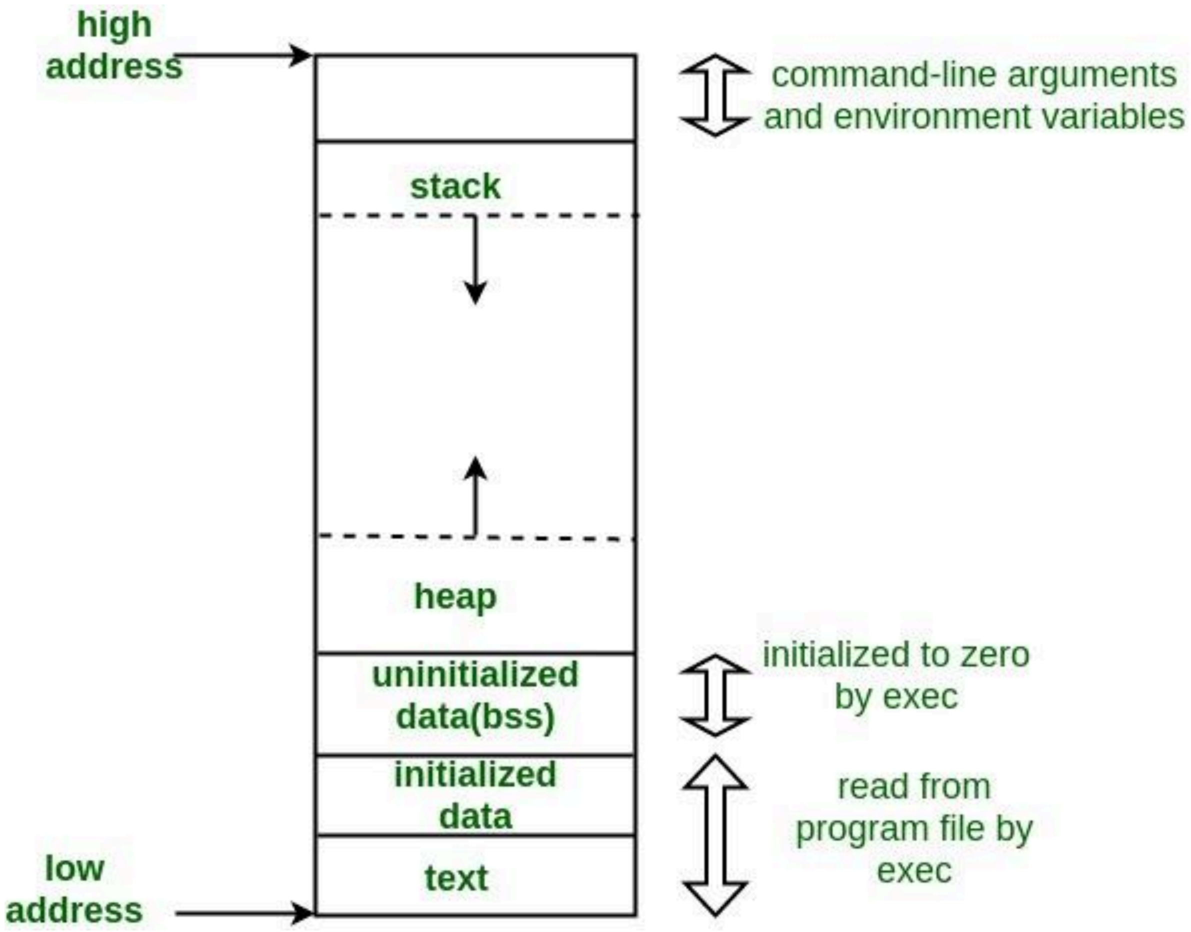
RI is register for index (0 if empty)

D is displacement (0 if empty)

S is scale 1, 2, 4 or 8 (1 if empty)

source read from:

Mem[%RB + D + S*%RI]


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Eric Yoon - CPSC 323 Cheat Sheet

Q1: general question about assembly
Q2: how many memory accesses
Q3: more advanced, come back for later. covered in optional lecture. find dimensions of 2D struct array, and answer questions about struct. know about memory alignment. multiple parts, no partial credit.
Q4: pset2 implementation strategy
Q5: optimizations—what optimizations can you use on what parts of this code? (optimization blockers) multiple parts, no partial credit.
Q6: SA, TF, MCQ. about "philosophy of CS / CS terms." design goals, definitions, pros/cons. abstraction, (micro)architectures, moore's law, switches, flip flops, DRAM/SRAM, recursion. lifetime, etc.
Q7: memory layout. heap vs stack (with C code examples). memory representation of arrays and structs.
Q8: MCQ/TF about rust. design choices of rust—why was rust designed the way it was. some questions about specific syntax and rust dev env. read rust textbook.
Q9: rust code given. determine if it compiles/errors/succeeds. 3-10 lines of code. enums included.
Q10: make sure you are at the lecture location, SAS students exempt.