Reference

Hexadecimal

Hex	Binary	Decimal
0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6
0x7	0111	7
0x8	1000	8
0x9	1001	9
0xa	1010	10
0xb	1011	11
0xc	1100	12
0xd	1101	13
0xe	1110	14
0xf	1111	15

Powers of Two

Power	Hex	Decimal
2^0	0x0001	1
2^1	0x0002	2
2^2	0x0004	4
2^3	0x0008	8
2^4	0x0010	16
2^5	0x0020	32
2^6	0x0040	64
2^7	0x0080	128
2^8	0x0100	256
2^9	0x0200	512
2^{10}	0x0400	1024
2^{11}	0x0800	2048
2^{12}	0x1000	4096
2^{13}	0x2000	8192
2^{14}	0x4000	16384
2^{15}	0x8000	32768
2^{16}	0x10000	65536

Arithmetic

$$2^a \times 2^b = 2^{a+b}$$
$$2^a * 2^b = 2^{a+b}$$
$$2^a \div 2^b = 2^{a-b}$$
$$2^a \;/\; 2^b = 2^{a-b}$$

Array index for element at row i and column j: $C * i + j$

x86-64 Data Sizes

C type	Suffix	Size (bytes)
`char`	`b`	`1`
`short`	`w`	`2`
`int`	`l`	`4`
`unsigned`	`l`	`4`
`long`	`q`	`8`

x86-64 Registers

Main Registers			Virtual Subregisters		
Name	**Special Usage**	**Convention**	low 4 bytes	low 2 bytes	low byte
%rax	return value	caller-save	%eax	%ax	%al
%rbx		callee-save	%ebx	%bx	%bl
%rcx	4th argument	caller-save	%ecx	%cx	%cl
%rdx	3rd argument	caller-save	%edx	%dx	%dl
%rsi	2nd argument	caller-save	%esi	%si	%sil
%rdi	1st argument	caller-save	%edi	%di	%dil
%rsp	stack pointer	stack pointer	%esp	%sp	%spl
%rbp		callee-save	%ebp	%bp	%bpl
%r8	5th argument	caller-save	%r8d	%r8w	%r8b
%r9	6th argument	caller-save	%r9d	%r9w	%r9b
%r10		caller-save	%r10d	%r10w	%r10b
%r11		caller-save	%r11d	%r11w	%r11b
%r12		callee-save	%r12d	%r12w	%r12b
%r13		callee-save	%r13d	%r13w	%r13b
%r14		callee-save	%r14d	%r14w	%r14b
%r15		callee-save	%r15d	%r15w	%r15b

x86-64 Addressing Mode

`O(B, I, S)`	$= O + R[B] + R[I] * S$
	`O`: offset, must be a constant in decimal or hex
	`B`: base, must be a register name
	`I`: index, must be a register name
	`S`: scale, must be 1, 2, 4, or 8

Common x86-64 Instructions

`push a`	push `a` onto the stack
`pop b`	pop a value from the stack into `b`
`call target`	push return address onto the stack and jump to `target` label/address
`ret`	pop return address from stack and jump there
`lea a, b`	store address of memory addressing expression `a` in `b` Note: scales may be one of 1, 2, 4, 8.
`mov a, b`	copy `a` into `b`
`movs a, b`	store sign-extended `a` into `b`
`movz a, b`	store zero-extended `a` into `b`
`inc a`	increment `a` by 1
`dec a`	decrement `a` by 1
`add a, b`	store sum `b+a` into `b`
`sub a, b`	store difference `b-a` into `b`
`imul a, b`	store signed product `b*a` into `b`
`and a, b`	store bitwise AND `b&a` into `b`
`or a, b`	store bitwise OR `b|a` into `b`
`shl/sal a, b`	store left shift `b<<a` into `b`
`shr a, b`	store logical right shift `b>>a` into `b`
`sar a, b`	store arithmetic right shift `b>>a` into `b`
`cmp a, b`	set condition codes based on difference `b-a`
`test a, b`	set condition codes based on bitwise AND `b&a`
`jg`	conditional jump to `target` if last comparison was greater than (zero)
`je`	conditional jump to `target` if result of last comparison was equal to (zero)
`jne`	conditional jump to `target` if result of last comparison was not equal to (zero)
`jle`	conditional jump to `target` if result of last comparison was less than or equal to (zero)
`ja`	conditional jump to `target` if result of last comparison was unsigned greater than (zero)
`jb`	conditional jump to `target` if result of last comparison was unsigned less than (zero)
`jmp target`	jump to `target` label/address