Memory Operand Forms		NOTES
(ra)	M[ra]	
imm(rb)	M[imm + rb]	
(rb, ri)	M[rb + ri]	
imm(rb, ri)	M[imm + rb + ri]	
(, ri, s)	M[ri * s]	s has to be base 2 (up to 8): (1),2,4,8
imm(, ri, s)	M[imm + ri * s]	s has to be base 2 (up to 8)
(rb, ri, s)	M[rb + ri * s]	s has to be base 2 (up to 8)
imm(rb, ri, s)	M[imm + rb + ri * s]	s has to be base 2 (up to 8)
MOV		
mov[] S, D	D = S	S and D can be addresses or reg. (not at same time).
		movl sets 4 high bytes to 0 if into 64-bit reg.
		source imm can only be 32-bit (2's complement)
movabsq I, R	R = I (I: 64-bit imm)	R has to be register, I can be 64-bit imm.
movz[][] S, R	R = S (0 extension)	R must be larger than S
movs[][] S, R	R = S (sign extension)	" "
c[]t[]	sign extend low to high (rax)	If [] = o, extends rax into rdx
cmov[c] S, R	R (=) S	see "set[c]" for conditions
STACK		
pushq S	|push stack| = S	||: rsp = rsp - 8; M[rsp] = S;
popq D	D = |pop stack|	||: D = M[rsp]; rsp = rsp + 8;
ARITHMETIC		
leaq S, D	D = *M[S]	copies address directly (allows mem ops)
inc[] – dec[] D	D = D + 1 – D - 1	
neg[] – not[] D	D = -D -- ~D	
add[] – sub[] S, D	D = D + S – D - S	
xor[] – or[] -- and[] S, D	D = S ^ D – S | D – S & D	
MULTIPLICATION		
[s]mul[] S, D	D = D * S	
[s]mulq S	rdx:rax = rax * S	
[s]divq S	rax = rdx:rax / S	
	rdx = rdx:rax % S	
SHIFTS		
sal[] -- shl[] K, D	D = D << K	K must be single byte register
sar[] K, D	D = D >>(arithmetic) K	fills high bits with high most bit (signed)
shr[] K, D	D = D >>(logical) K	fills high bits with 0s (unsigned)
CONDITIONALS		
cmp[] S1, S2	rflag (=) S2 - S1	usually used to check ordering
		compares in S2, S1 ordering (e.g. S2 > S1)
test[] S1, S2	rflag (=) S1 & S2	usually used to check zero, pos., or neg.
set[c] D	D = |[c]|	
JUMP		
jmp L – *O -- *(Q)	rip = L – O -- M[Q]	O is reg with loc. Q ptr to mem with loc
j[c] L	rip (=) L	see "set[c]"
		can only be direct (no *O)
PROCEDURES		
call L -- *O	|push stack|; rip = L -- O	||: rsp = rsp - 8; M[rsp] = rip;
ret	rip = |pop stack|	||: rip = M[rsp]; rsp = rsp + 8;
leave	rsp = rbp; rbp = |pop stack|	sets rsp to bottom of frame, then sets rbp to prev. rbp
syscall	calls system call with specific type	type: %rax, args: %rdi, %rsi, %rdx, %r10, %r8, %r9

SIZE []			
b	byte (1-byte)		
w	word (2-bytes)		
l	long (double) (4-bytes)		
q	quad (8-bytes)		
o	oct (16-bytes)	not common	
SIGN [s]			
i	signed		
_	unsigned		
CONDITIONS [c]			**|[c]|**
e -- z	equal -- zero		ZF
ne -- nz	not equal – not zero		~ZF
s	negative		SF
ns	not negative		~SF
g -- nle	greater (>)	signed	~(SF ^ OF) & ~ZF
ge -- nl	greater or equal (>=)	signed	~(SF ^ OF)
l -- nge	less (<)	signed	SF ^ OF
le -- ng	less or equal (<=)	signed	(SF ^ OF) | ZF
a -- nbe	above (>)	unsigned	~CF & ~ZF
ae -- nb	aove or equal (>=)	unsigned	~CF
b -- nae	below (<)	unsigned	CF
be -- na	below or equal (<=)	unsigned	CF | ZF
FLAGS			
CF	Carry Flag	1 if unsigned overflow, 0 if not	
PF	Parity Flag	xor of 8 least sig. bits	
AF	Adjust Flag	used for decimal/hex math	
ZF	Zero Flag	0 if result is 0, 0 if not	
SF	Sign Flag	most significant bit of result	
TF	Trap Flag		
IF	Interupt Flag		
DF	Direction Flag		
OF	Overflow Flag	1 if signed overflow, 0 if not	

REGISTERS						
64-bit	**32-bit**	**16-bit**	**8-bit**	**func. data**	**callee saved (rest caller)**	
rax	eax	ax	ah – al	returned value		
rbx	ebx	bx	bh – bl		callee	
rcx	ecx	cx	ch – cl	4th arg.		
rdx	edx	dx	dh – dl	3rd arg.		
rsi	esi	si	sih – sil	2nd arg.		
rdi	edi	di	dih – dil	1st arg.		
rbp	ebp	bp	bph – bpl	base ptr. (current frame)	callee; Contains the rbp of the prev. frame	
rsp	esp	sp	sph – spl	stack ptr.	callee (because of retq)	
r8	r8d	r8w	r8b	5th arg.		
r9	r9d	r9w	r9b	6th arg.		
r10	r10d	r10w	r10b			
r11	r11d	r11w	r11b			
r12	r12d	r12w	r12b		callee	
r13	r13d	r13w	r13b		callee	
r14	r14d	r14w	r14b		callee	
r15	r15d	r15w	r15b		callee	
REGISTERS	BIG					
256-bit	**128-bit**	**func. data**				
ymm0	xmm0	1st arg – return				
ymm1	xmm1	2nd arg				
ymm2	xmm2	3rd arg				
ymm3	xmm3	4th arg				
ymm4	xmm4	5th arg				
ymm5	xmm5	6th arg				
ymm6	xmm6	7th arg				
ymm7	xmm7	8th arg				
ymm8	xmm8	caller saved				
ymm9	xmm9	caller saved				
ymm10	xmm10	caller saved				
ymm11	xmm11	caller saved				
ymm12	xmm12	caller saved				
ymm13	xmm13	caller saved				
ymm14	xmm14	caller saved				
ymm15	xmm15	caller saved				
xmm for float	ymm for SIMD					

[] - PERCISION		
s	single (float)	
d	double (double)	
MOV		
vmovs[] S, D	D = S	
vmovap[] S, D	D = S	causes issue if not aligned to 16-byte
CONVERSION		
vcvtts[]2si S, D	D = [](S)	with truncation
vcvtts[]2si S, D	D = [](S)	
vcvtts[]2siq S, D	D = [](S)	
vcvtts[]2siq S, D	D = [](S)	
vcvtsi2s[]	D = [](S)	without truncation
vcvtsi2s[]	D = [](S)	
vcvtsi2s[]q	D = [](S)	
vcvtsi2s[]q	D = [](S)	
MATH		
vadds[] S1, S2, D	D = S2 + S1	
vsubs[] S1, S2, D	D = S2 - S1	
vmuls[] S1, S2, D	D = S2 * S1	
vdivs[] S1, S2, D	D = S2/S1	
vmaxs[] S1, S2, D	D = max(S2, S1)	
vmins[] S1, S2, D	D = min(s2, S1)	
sqrts[] S1, D	D = sqrt(S1)	
COMPARISON		
ucomis[] S1, S2	S2 - S1	compare based on this
		If NaN is set, parity flag is set
BITWISE/LOGIC OPS		
Precedence	**Notes**	
< <=		
>= >		
== !=		
&		
^	can be done in any order	
|		
&&		
||		
FUs (HASWELL)		
0: float & int ops, branch	incl. int div	
1: float & int ops	incl. int mult	
2: load & address ops		
3: load & address ops		
4: store		
5: int ops	no int mult/div	
6: int ops, branch	no int mult/div	
7: store address ops	(store op requires 2 functional units)	

POSIX THREADS	
int pthread_create(pthread_t* **tid**, pthread_attr_t* **attr**, func* **f**, void* **arg**)	**RET:** 0 if ok, not 0 if error
	tid: ID of thread
	attr: NULL (in our scope)
	f: func of form: void* f(void* arg)
	arg: void* argument passed to f
pthread_t pthread_self(void)	**RET:** thread ID of caller
void pthread_exit(void* **thread_return**)	**thread_return:** what the thread returns
int pthread_cancel(pthread_t **tid**)	**RET:** 0 if ok, not 0 if error
	tid: ID of thread to cancel
int pthread_join(pthread_t **tid**, void** **thread_return**)	**RET:** 0 if ok, not 0 if error
	tid: thread to wait for
	thread_return: ptr to a void* that will point to return of tid
int pthread_detach(pthread_t **tid**)	**RET:** 0 if ok, not 0 if error
	tid: the thread to detach (can be self)
SEMAPHORES	
int sem_init(sem_t* **sem**, 0, unsigned int **value**)	**RET:** 0 if ok, -1 if error
	sem: pointer to semaphore to be used
	value: total number of threads that can access at same time
int sem_wait(sem_t* **s**)	**RET:** 0 if ok, -1 if error
	s: the semaphore to use
int sem_post(sem_t* **s**)	**RET:** 0 if ok, -1 if error
	s: the semaphore to use
SIGNALS & EXCEPTIONS	
int kill(pid_t **pid**, int **sig**)	**RET:** 0 if ok, -1 if error
	pid: the id of the processes (can be self)
	sig: the signal that it will send
int alarm(unsigned int **secs**)	**RET:** remaining secs of prev. alarm, or 0 if there is none
	secs: the number of seconds before SIGALRM is sent
sighandler_t signal(int **signum**, sighandler_t **handler**)	**RET:** ptr to prev. handler, SIG_ERR if there was an error
	signum: the signal that should evoke this handler
	handler: ptr to function that will handle signal in following form:
	void func(int) <- int is signum, so 1 handler works with others
int setjmp(jmp_buf **env**)	**RET:** 0 if ok, nonzero due to longjmps
	env: the buffer used by jmp_buf later to jump back here
void longjmp(jmp_buf **env**, int **retval**)	**RET:** never, it never returns!
	env: the buffer used to jump to the setjmp
	retval: the return value of setjmp (if 0, auto sets to 1)
MMAP	
void *mmap(void ***addr**, size_t **length**, int **prot**, int **flags**, int **fd**, off_t **offset**);	**Eg:** int* map = mmap(0, FILESIZE,
	PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
	mmapped an array of int
	RET: pointer to mapped area, or MAP_FAILED (-1) on error
	addr: if not NULL, then kernel takes as a hint
	about where to place the mapping;
int munmap(void ***addr**, size_t **length**);	**RET**: 0 on OK, -1 on error
	addr: exactly where to unmap