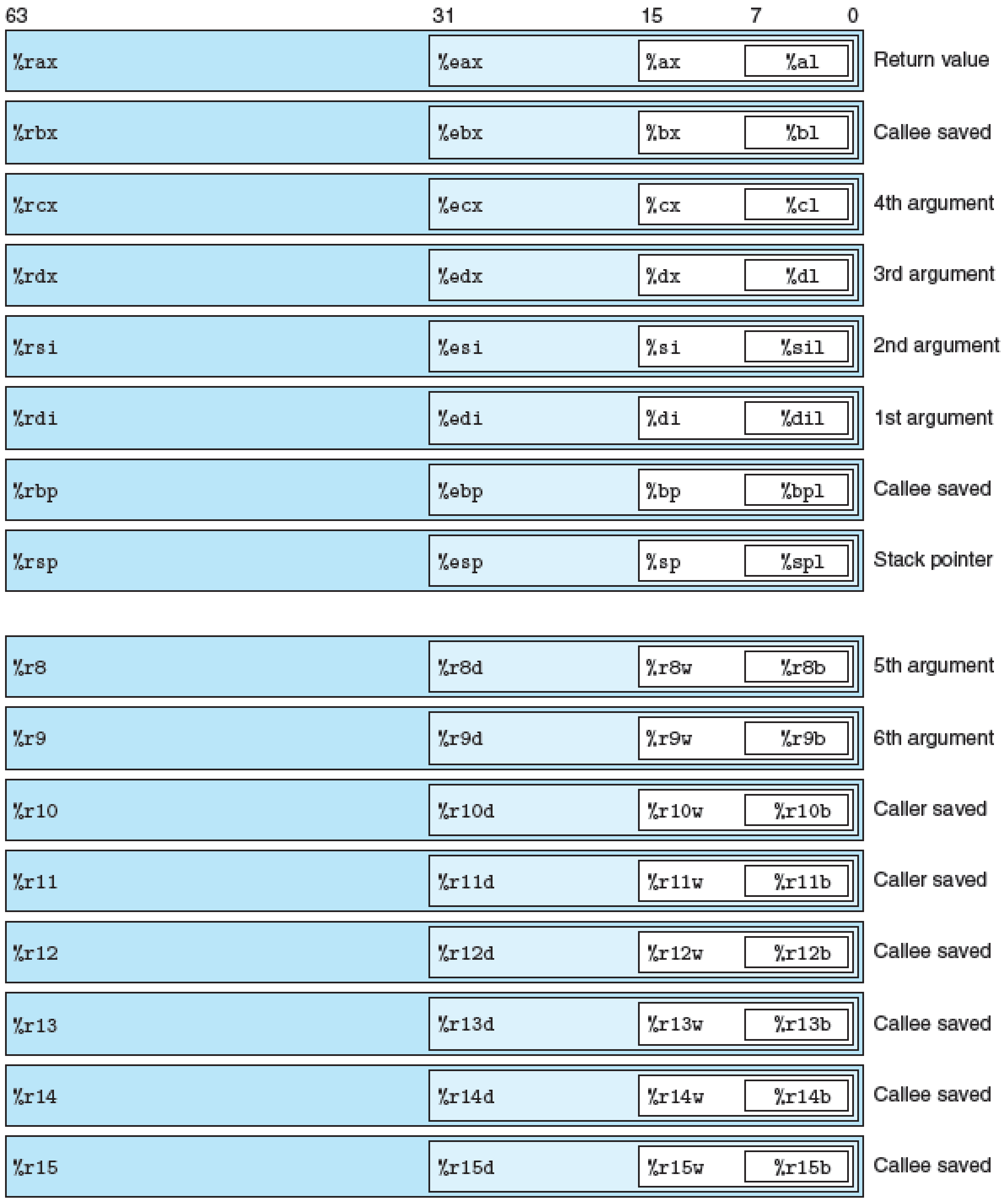
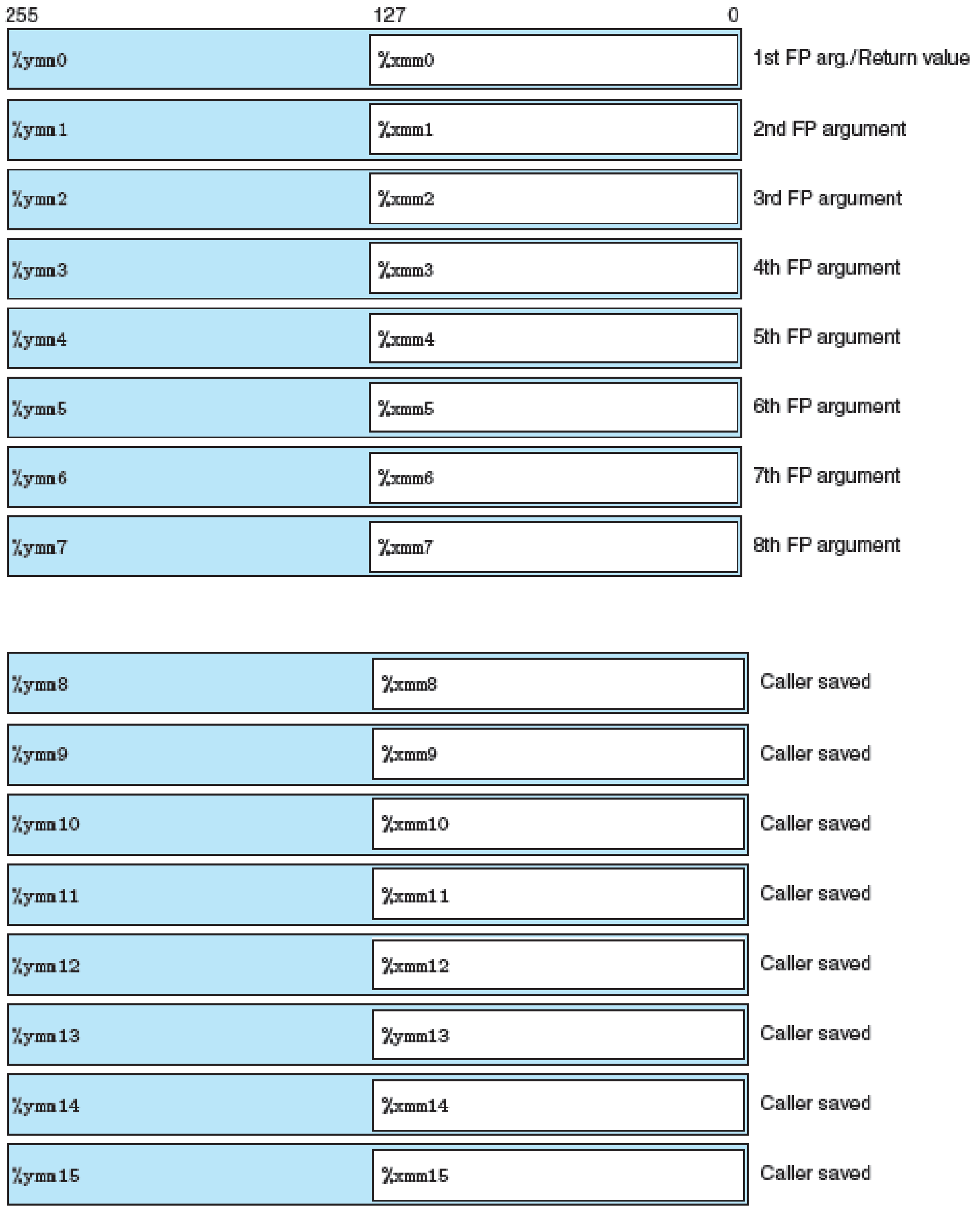
ASM instruction sum x86-64 GAS

x86-64 GPR Register

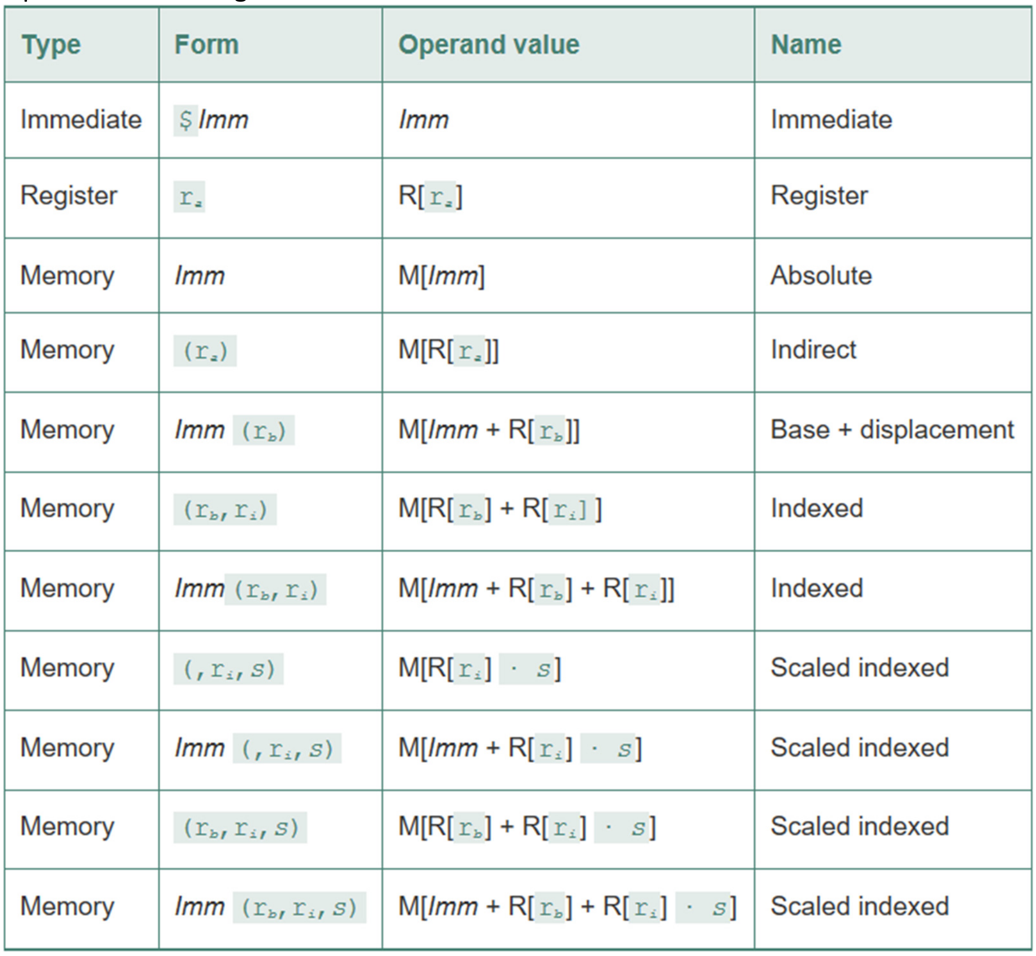

63
31
15
7
0
%rax %eax %ax %al Return value
%rbx %ebx %bx %bl Callee saved
%rcx %ecx %cx %cl 4th argument
%rdx %edx %dx %dl 3rd argument
%rsi %esi %si %sil 2nd argument
%rdi %edi %di %dil 1st argument
%rbp %ebp %bp %bpl Callee saved
%rsp %esp %sp %spl Stack pointer
%r8 %r8d %r8w %r8b 5th argument
%r9 %r9d %r9w %r9b 6th argument
%r10 %r10d %r10w %r10b Caller saved
%r11 %r11d %r11w %r11b Caller saved
%r12 %r12d %r12w %r12b Callee saved
%r13 %r13d %r13w %r13b Callee saved
%r14 %r14d %r14w %r14b Callee saved
%r15 %r15d %r15w %r15b Callee saved

x86-64 floating point register



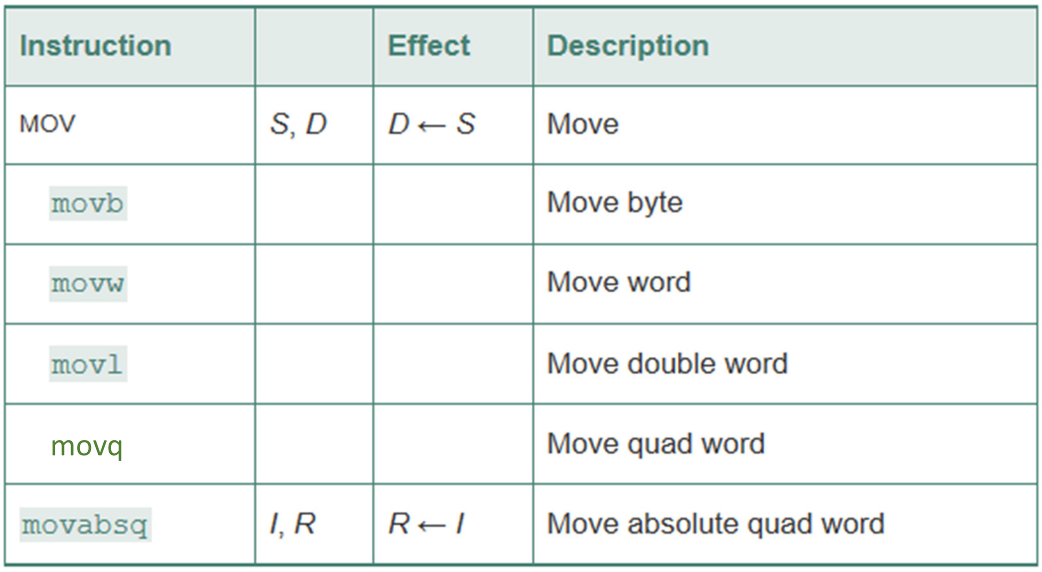
Accessing Data / Data move instructions

Operator addressing



Type	Form	Operand value	Name
Immediate	\$*Imm*	*Imm*	Immediate
Register	r_a	R[r_a]	Register
Memory	*Imm*	M[*Imm*]	Absolute
Memory	(r_a)	M[R[r_a]]	Indirect
Memory	*Imm* (r_b)	M[*Imm* + R[r_b]]	Base + displacement
Memory	(r_b, r_i)	M[R[r_b] + R[r_i]]	Indexed
Memory	*Imm* (r_b, r_i)	M[*Imm* + R[r_b] + R[r_i]]	Indexed
Memory	(, r_i, *s*)	M[R[r_i] · *s*]	Scaled indexed
Memory	*Imm* (, r_i, *s*)	M[*Imm* + R[r_i] · *s*]	Scaled indexed
Memory	(r_b, r_i, *s*)	M[R[r_b] + R[r_i] · *s*]	Scaled indexed
Memory	*Imm* (r_b, r_i, *s*)	M[*Imm* + R[r_b] + R[r_i] · *s*]	Scaled indexed

Mov



Instruction		Effect	Description
MOV	*S*, *D*	*D* ← *S*	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	*I*, *R*	*R* ← *I*	Move absolute quad word

Mov zero extend

Instruction	Effect	Description
`MOVZ` *S*,*R*	*R* ← ZeroExtend(*S*)	Move with zero extension
`movzbw`		Move zero-extended byte to word
`movzbl`		Move zero-extended byte to double word
`movzwl`		Move zero-extended word to double word
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

Mov sign extend

Instruction	Effect	Description
MOVS *S*,*R*	*R* ← SignExtend(*S*)	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	%rax ← SignExtend(%eax)	Sign-extend %eax to %rax

Stack

Instruction	Effect	Description
`pushq` *S*	`R[%rsp] ← R[%rsp] -8;` `M[R[%rsp]] ← S`	Push quad word
`popq` *D*	*D* ← `M[R[%rsp]];` `R[%rsp] ← R[%rsp] + 8`	Pop quad word

Arithmetic and logical ops

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D+S$	Add
SUB	S, D	$D \leftarrow D-S$	Subtract
IMUL	S, D	$D \leftarrow D*S$	Multiply
XOR	S, D	$D \leftarrow D \wedge S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D\&S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
`SHR`	k, D	$D \leftarrow D >>_L k$	Logical right shift

Mul div

Instruction	Effect	Description
`imulq` S	R[`%rdx`]:R[`%rax`] $\leftarrow S \times$ R[`%rax`]	Signed full multiply
`mulq` S	R[`%rdx`]:R[`%rax`] $\leftarrow S \times$ R[`%rax`]	Unsigned full multiply
`cqto`	R[`%rdx`]:R[`%rax`] $\leftarrow$ SignExtend(R[`%rax`])	Convert to oct word
`idivq` S	R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div S$	Signed divide
`divq` S	R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div S$	Unsigned divide

Control

Compare

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
cmpb			Compare byte
cmpw			Compare word
cmpl			Compare double word
cmpq			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
testb			Test byte
testw			Test word
testl			Test double word
testq			Test quad word

Set

Instruction		Synonym	Effect	Set condition
sete	D	setz	D ← ZF	Equal / zero
setne	D	setnz	D ← ~ ZF	Not equal / not zero
sets	D		D ← SF	Negative
setns	D		D ← ← SF	Nonnegative
setg	D	setnle	D ← ~ (SF ^ OF) & ~ ZF	Greater (signed >)
setge	D	setnl	D ← ~ (SF ^ OF)	Greater or equal (signed >=)
setl	D	setnge	D ← SF ^ OF	Less (signed <)
setle	D	setng	D ← (SF ^ OF) | ZF	Less or equal (signed <=)
seta	D	setnbe	D ← ~ CF & ~ ZF	Above (unsigned >)
setae	D	setnb	D ← ~ CF	Above or equal (unsigned >=)
setb	D	setnae	D ← CF	Below (unsigned <)
setbe	D	setna	D ← CF | ZF	Below or equal (unsigned <=)

Jump

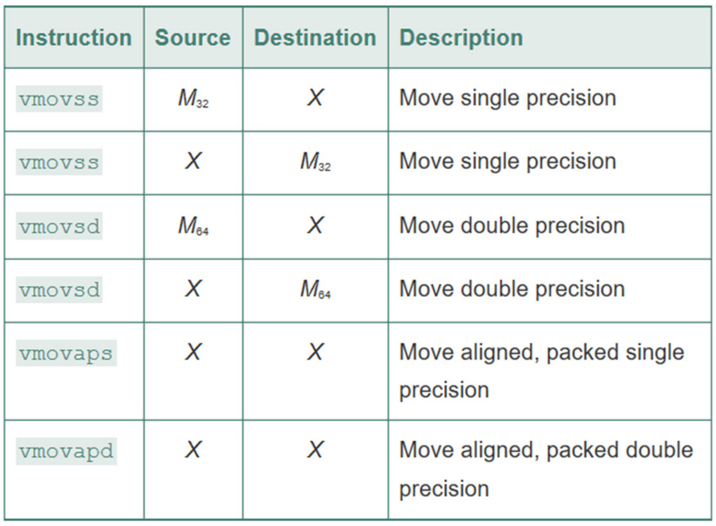
Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	ZF	Equal / zero
`jne`	*Label*	`jnz`	~ZF	Not equal / not zero
`js`	*Label*		SF	Negative
`jns`	*Label*		~SF	Nonnegative
`jg`	*Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`jge`	*Label*	`jnl`	~(SF ^ OF)	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	SF ^ OF	Less (signed <)
`jle`	*Label*	`jng`	(SF ^ OF) | ZF	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jae`	*Label*	`jnb`	~CF	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	CF	Below (unsigned <)
`jbe`	*Label*	`jna`	CF | ZF	Below or equal (unsigned <=)

Conditional mov

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

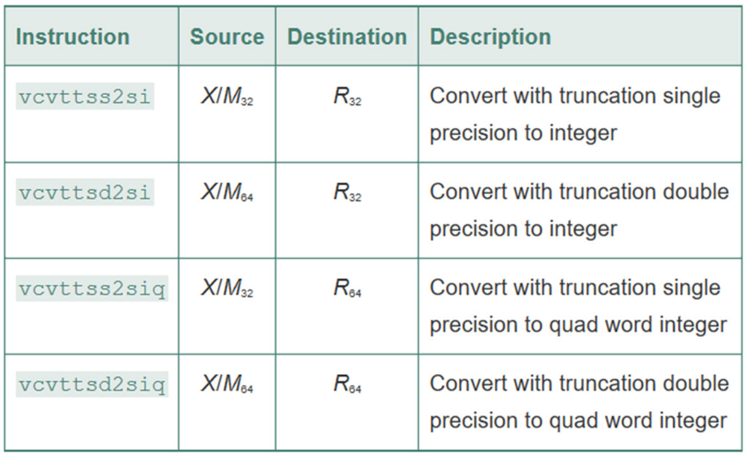
Floating point (AVX scalar)

Move



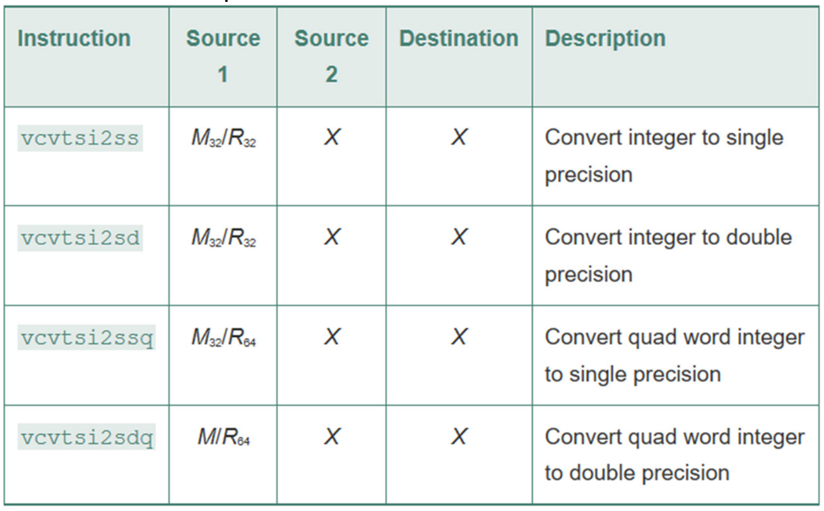
Instruction	Source	Destination	Description
`vmovss`	M_{32}	X	Move single precision
`vmovss`	X	M_{32}	Move single precision
`vmovsd`	M_{64}	X	Move double precision
`vmovsd`	X	M_{64}	Move double precision
`vmovaps`	X	X	Move aligned, packed single precision
`vmovapd`	X	X	Move aligned, packed double precision

Conversion fp to int



Instruction	Source	Destination	Description
`vcvttss2si`	X/M_{32}	R_{32}	Convert with truncation single precision to integer
`vcvttsd2si`	X/M_{64}	R_{32}	Convert with truncation double precision to integer
`vcvttss2siq`	X/M_{32}	R_{64}	Convert with truncation single precision to quad word integer
`vcvttsd2siq`	X/M_{64}	R_{64}	Convert with truncation double precision to quad word integer

Conversion int to fp



Instruction	Source 1	Source 2	Destination	Description
`vcvtsi2ss`	M_{32}/R_{32}	X	X	Convert integer to single precision
`vcvtsi2sd`	M_{32}/R_{32}	X	X	Convert integer to double precision
`vcvtsi2ssq`	M_{32}/R_{64}	X	X	Convert quad word integer to single precision
`vcvtsi2sdq`	M/R_{64}	X	X	Convert quad word integer to double precision

Arithmetic

Single	Double	Effect	Description
`vaddss`	`vaddsd`	$D \leftarrow S_2 + S_1$	Floating-point add
`vsubss`	`vsubsd`	$D \leftarrow S_2 - S_1$	Floating-point subtract
`vmulss`	`vmulsd`	$D \leftarrow S_2 \times S_1$	Floating-point multiply
`vdivss`	`vdivsd`	$D \leftarrow S_2/S_1$	Floating-point divide
`vmaxss`	`vmaxsd`	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
`vminss`	`vminsd`	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
`sqrtss`	`sqrtsd`	$D \leftarrow \sqrt{S_1}$	Floating-point square root

Logic

Single	Double	Effect	Description
`vxorps`	`xorpd`	$D \leftarrow S_2 \wedge S_1$	Bitwise EXCLUSIVE-OR
`vandps`	`andpd`	$D \leftarrow S_2 \& S_1$	Bitwise AND

Compare

Instruction		Based on	Description
`ucomiss`	S_1, S_2	S_2-S_1	Compare single precision
`ucomisd`	S_1, S_2	S_2-S_1	Compare double precision

Ordering S_2:S_1	`CF`	`ZF`	`PF`
Unordered	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

Floating point (AVX vector)

- General notes:
  - instruction: instruction suffix: s=single precision, d= double precision
  - operands: X means xmm-register or ymm-register operand; M means 128bit memory or 256bit memory operand; max one operand can be a memory operand

vector - fp - mov

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vmovaps		X/M	X/M	Move aligned packed single-precision floating-point //AVX (SSE)
vmovapd		X/M	X/M	Move aligned packed double-precision floating-point values //AVX (SSE2)
vmovups		X/M	X/M	Move unaligned packed single-precision floating-point //AVX (SSE)
vmovupd		X/M	X/M	Move unaligned packed double-precision floating-point //AVX (SSE2)

vector - fp - mov - parts

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vmovlps		X/M	X	Move two packed single-precision floating-point (lower part) to destination //AVX (SSE)
vmovlpd		X/M	X	Move double-precision floating-point value (lower part) to destination //AVX (SSE2)
vmovhps		X/M	X	Move two packed single-precision floating-point (higher part) to destination //AVX (SSE)
vmovhpd		X/M	X	Move double-precision floating-point value (higher part) to destination //AVX (SSE2)

vector - fp - shuffle

Instruction	Src3	Src2	Src1	Dest	Description //<availability>
vshufps	I8	X/M	X	X	Select from quadruplet of single-precision floating-point values in src1 and src2 using I8, interleaved result pairs are stored in dest //AVX (SSE)
vshufpd	I8	X/M	X	X	Shuffle two pairs of double-precision floating-point values from src1 and src2 using I8 to select from each pair, interleaved result is stored in dest //AVX (SSE2)

- Note:
  - operands:
    - I8: I8 consists of multiple 2-bit indexes (read backwards (from right to left!); first half of the indexes point to src1, second half point to src2)

vector - fp - extract and insert

Instruction	Src3	Src2	Src1	Dest	Description //<availability>
vextractf128		I8	X256	X128/M128	Extract 128 bits of packed floating-point values from src1 and store results in dest -I8: i8[0]: 0 ->extract lower 128bit //AVX2
vinsertf128	I8	X128 /M128	X256	X256	Insert 128 bits of packed floating-point values from src2 and the remaining values from src1 into dest -I8: i8[0]: 0 ->insert lower 128bit //AVX2

- Note:
  - operands:
    - XYYY means xmm-register or ymm register operand with a bit width of yyy
    - MYYY means memory operand with a bit width of yyy; max one operand can be a memory operand

Instruction	Src3	Src2	Src1	Dest	Description //<availability>
vextractps		I8	X128	X32/M32	Extract one single-precision floating-point value from src1 at the offset specified by I8 and store the result in dest. Zero extend the results in 64-bit register if applicable -I8: i8[1:0] defines the nth float value to extract //AVX (SSE4.1)
vinsertps	I8	X32/M32	X128	X128	-Insert a single-precision floating-point value selected by I8 from src2 and merge with values in src1 at the specified destination element specified by imm8 and write out the result and zero out destination elements in dest as indicated in I8 -I8: --i8[7:6] defines the nth float value in the src2 --i8[5:4] defines the nth float value in the dest --i8[4:0]: A 1 zeros the nth float value in the dest //AVX (SSE4.1)

Note:
-operands:
--XYYY means xmm-register or ymm register operand with a bit width of yyy
--MYYY means memory operand with a bit width of yyy; max one operand can be a memory operand

vector - fp - arith – add

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vaddps	X/M	X	X	Add packed single-precision floating-point values from src2 to src1 and store result in dest.//AVX (SSE)
vaddpd	X/M	X	X	Add packed double-precision floating-point values from src2 to src1 and store result in dest.//AVX (SSE2)

vector - fp - arith - sub

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vsubps	X/M	X	X	Subtract packed single-precision floating-point values in src2 from src1 and stores result in dest. //AVX (SSE)
vsubpd	X/M	X	X	Subtract packed double-precision floating-point values in src2 from src1 and stores result in dest. //AVX (SSE2)

vector - fp - arith - mul

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vmulps	X/M	X	X	Multiply packed single-precision floating-point values in src2 with src1 and store result in dest. //AVX (SSE)
vmulpd	X/M	X	X	Multiply packed double-precision floating-point values in src2 with src1 and store result in dest. //AVX (SSE2)

vector - fp - arith - div

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vdivps	X/M	X	X	Divide packed single-precision floating-point values in src1 by packed single-precision floating-point values in src2 //AVX (SSE)
vdivpd	X/M	X	X	Divide packed double-precision floating-point values in src1 by packed double-precision floating-point values in src2 //AVX (SSE2)

vector - fp - arith - special - horizontal add

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vhaddps	X/M	X	X	Horizontal add packed single-precision floating-point values from src1 and src2 in dest.//AVX (SSE3)
vhaddpd	X/M	X	X	Horizontal add packed double-precision floating-point values from src1 and src2 in dest.//AVX (SSE3)

vector - fp - arith - special - horizontal sub

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vhsubps	X/M	X	X	Horizontal subtract packed single-precision floating-point values from src1 and src2 //AVX (SSE3)
vhsubpd	X/M	X	X	Horizontal subtract packed double-precision floating-point values from src1 and src2 //AVX (SSE3)

vector - fp - arith - special - min/max

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vminps	X/M	X	X	Return the minimum single-precision floating-point values between src1 and src2 in dest. //AVX (SSE)
vminpd	X/M	X	X	Return the minimum double-precision floating-point values between src1 and src2 in dest. //AVX (SSE2)
vmaxps	X/M	X	X	Return the maximum single-precision floating-point values between src1 and src2 in dest. //AVX (SSE)
vmaxpd	X/M	X	X	Return the maximum double-precision floating-point values between src1 and src2 in dest. //AVX (SSE2)

vector - fp - arith - special - sqrt

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vsqrtps		X/M	X	Computes Square Roots of the packed single-precision floating-point values in src1 and stores the result in dest.//AVX (SSE)
vsqrtpd		X/M	X	Computes Square Roots of the packed double-precision floating-point values in src1 and stores the result in dest.//AVX (SSE2)

vector - fp - logical operations

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vandps	X/M	X	X	Return the bitwise logical AND of packed values in src1 and src2 //AVX (SSE)
vandpd	X/M	X	X	Return the bitwise logical AND of packed values in src1 and src2 //AVX (SSE2)
vandnps	X/M	X	X	Return the bitwise logical NAND of packed values in src1 and src2 //AVX (SSE)
vandnpd	X/M	X	X	Return the bitwise logical NAND of packed values in src1 and src2 //AVX (SSE2)
vorps	X/M	X	X	Return the bitwise logical OR of packed values in src1 and src2 //AVX (SSE)
vorpd	X/M	X	X	Return the bitwise logical OR of packed values in src1 and src2 //AVX (SSE2)
vxorps	X/M	X	X	Return the bitwise logical XOR of packed values in src1 and src2 //AVX (SSE)
vxorpd	X/M	X	X	Return the bitwise logical XOR of packed values in src1 and src2 //AVX (SSE2)

vector - fp - conv - floating point from/to floating point

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vcvtps2pd		X/M	X	Convert two packed single-precision floating-point values in src1 to two packed double-precision floating-point values in dest //AVX (SSE2)
vcvtpd2ps		X/M	X	Convert two packed double-precision floating-point values in src1 to two single-precision floating-point values in dest //AVX (SSE2)

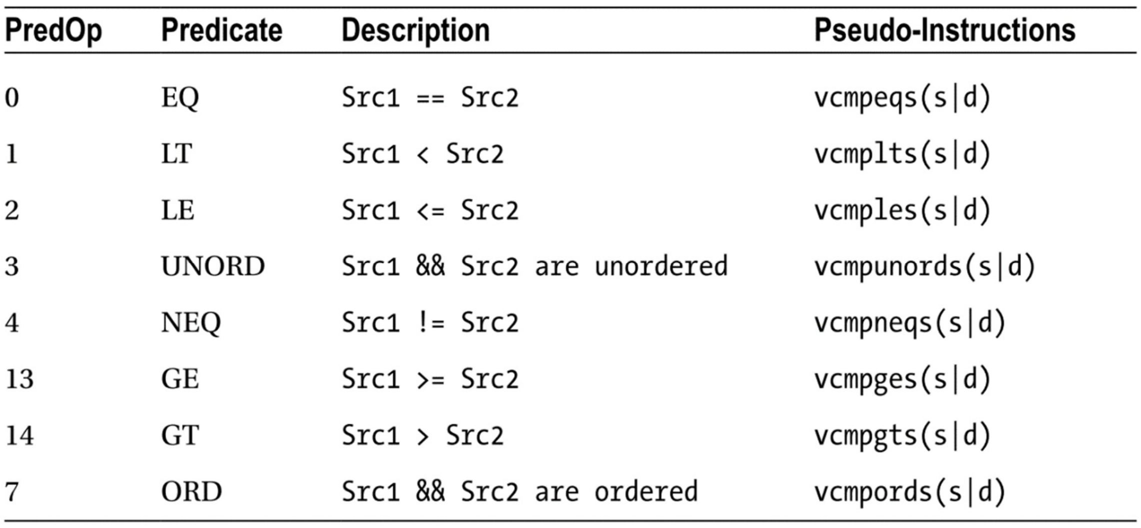
vector - fp - conv - floating point to/from integer

Instruction	Source 2	Source 1	Desti-nation	Description //<availability>
vcvtps2dq		X/M	X	Convert packed single-precision floating-point values from src1 to packed signed doubleword values in dest. //AVX (SSE2)
vcvtpd2dq		X/M	X	Convert packed double-precision floating-point values from src1 to packed signed doubleword values in dest. //AVX (SSE2)
vcvtdq2ps		X/M	X	Convert packed signed doubleword integers from src1 to packed single-precision floating-point values in dest. //AVX (SSE2)
vcvtdq2pd		X/M	X	Convert packed signed doubleword integers from src1 to packed double-precision floating-point values in dest. //AVX (SSE2)

vector - fp - comp(are)

Instruction	Src3	Src2	Src1	Dest	Description //<availability>
vcmpps	I8	X/M	X	X	Compare packed single-precision floating-point values in src2 and src1 using bits 4:0 of src3/I8 as a comparison predicate; if true, dest part contains all ‘1’, if false -> all ‘0’ //AVX (SSE)
vcmppd	I8	X/M	X	X	Compare packed double-precision floating-point values in src2 and src1 using bits 4:0 of src3/I8 as a comparison predicate; if true, dest part contains all ‘1’, if false -> all ‘0’ //AVX (SSE2)

- Note:
  - I8:
    - comparison predicate I8[4:0] (see next table)



PredOp	Predicate	Description	Pseudo-Instructions
0	EQ	Src1 == Src2	vcmpeqs(s|d)
1	LT	Src1 < Src2	vcmplts(s|d)
2	LE	Src1 <= Src2	vcmples(s|d)
3	UNORD	Src1 && Src2 are unordered	vcmpunords(s|d)
4	NEQ	Src1 != Src2	vcmpneqs(s|d)
13	GE	Src1 >= Src2	vcmpges(s|d)
14	GT	Src1 > Src2	vcmpgts(s|d)
7	ORD	Src1 && Src2 are ordered	vcmpords(s|d)

- Note: There are also pseudo-instructions with 3 operands instead of vcmpss/vcmpsd with 4 operands inclusive the imp operand (see 4th column)