Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$\mathsf{R}[\mathtt{r}_a]$	Register
Memory	Imm	$\mathsf{M}[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
`movb`			Move byte
`movw`			Move word
`movl`			Move double word
`movq`			Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow \mathsf{ZeroExtend}(S)$	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

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow$ 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` $\leftarrow$ SignExtend(`%eax`)	Sign-extend `%eax` to `%rax`

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers `%eax` and `%rax`.

Note the absence of an explicit instruction to zero-extend a 4-byte source value to an 8-byte destination in Figure 3.5. Such an instruction would logically be named `movzlq`, but this instruction does not exist. Instead, this type of data movement can be implemented using a `movl` instruction having a register as the destination. This technique takes advantage of the property that an instruction generating a 4-byte value with a register as the destination will fill the upper 4 bytes with zeros. Otherwise, for 64-bit destinations, moving with sign extension is supported for all three source types, and moving with zero extension is supported for the two smaller source types.

Figure 3.6 also documents the `cltq` instruction. This instruction has no operands—it always uses register `%eax` as its source and `%rax` as the destination for the sign-extended result. It therefore has the exact same effect as the instruction `movslq %eax, %rax`, but it has a more compact encoding.

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

Figure 3.8 Push and pop instructions.

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	$D \leftarrow D \hat{} S$	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	$D \leftarrow D >>_{\mathrm{A}} k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_{\mathrm{L}} k$	Logical right shift

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

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%rdx` and `%rax` are viewed as forming a single 128-bit oct word.

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

Figure 3.13 **Comparison and test instructions.** These instructions set the condition codes without updating any other registers.

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 <=)

Figure 3.14 **The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

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 <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Aside What do the instructions `rep` and `repz` do?

Line 8 of the assembly code shown on page 243 contains the instruction combination `rep; ret`. These are rendered in the disassembled code (line 6) as `repz retq`. One can infer that `repz` is a synonym for `rep`, just as `retq` is a synonym for `ret`. Looking at the Intel and AMD documentation for the `rep` instruction, we find that it is normally used to implement a repeating string operation [3, 51]. It seems completely inappropriate here. The answer to this puzzle can be seen in AMD's guidelines to compiler writers [1]. They recommend using the combination of `rep` followed by `ret` to avoid making the `ret` instruction the destination of a conditional jump instruction. Without the `rep` instruction, the `jg` instruction (line 7 of the assembly code) would proceed to the `ret` instruction when the branch is not taken. According to AMD, their processors cannot properly predict the destination of a `ret` instruction when it is reached from a jump instruction. The `rep` instruction serves as a form of no-operation here, and so inserting it as the jump destination does not change behavior of the code, except to make it faster on AMD processors. We can safely ignore any `rep` or `repz` instruction we see in the rest of the code presented in this book.

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 <=)

Figure 3.18 The conditional move instructions. These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Instruction		Description
`call`	*Label*	Procedure call
`call`	**Operand*	Procedure call
`ret`		Return from call

Command	Effect
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command-line arguments here)
`kill`	Stop your program
Breakpoints	
`break multstore`	Set breakpoint at entry to function `multstore`
`break *0x400540`	Set breakpoint at address `0x400540`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas multstore`	Disassemble function `multstore`
`disas 0x400544`	Disassemble function around address `0x400544`
`disas 0x400540, 0x40054d`	Disassemble code within specified address range
`print /x $rip`	Print program counter in hex
Examining data	
`print $rax`	Print contents of `%rax` in decimal
`print /x $rax`	Print contents of `%rax` in hex
`print /t $rax`	Print contents of `%rax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($rsp+8)`	Print contents of `%rsp` plus 8 in hex
`print *(long *) 0x7fffffffe818`	Print long integer at address `0x7fffffffe818`
`print *(long *) ($rsp+8)`	Print long integer at address `%rsp` + 8
`x/2g 0x7fffffffe818`	Examine two (8-byte) words starting at address `0x7fffffffe818`
`x/20b multstore`	Examine first 20 bytes of function `multstore`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.39 **Example GDB commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

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

Figure 3.46 Floating-point movement instructions. These operations transfer values between memory and registers, as well as between pairs of registers. (X: XMM register (e.g., `%xmm3`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

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

Figure 3.47 Two-operand floating-point conversion operations. These convert floating-point data to integers. (X: XMM register (e.g., `%xmm3`); R_{32}: 32-bit general-purpose register (e.g., `%eax`); R_{64}: 64-bit general-purpose register (e.g., `%rax`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

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_{64}/R_{64}	X	X	Convert quad word integer to single precision
`vcvtsi2sdq`	M_{64}/R_{64}	X	X	Convert quad word integer to double precision

Figure 3.48 Three-operand floating-point conversion operations. These instructions convert from the data type of the first source to the data type of the destination. The second source value has no effect on the low-order bytes of the result. (X: XMM register (e.g., `%xmm3`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

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

Figure 3.49 Scalar floating-point arithmetic operations. These have either one or two source operands and a destination operand.