APPENDIX B
Partial x64 Instruction Reference

Depending on how you count, the x64 architecture now has more than 1,000 machine instructions. Don't panic: A lot of those machine instructions may be used only by operating systems in protected mode. A large number of them implement floating-point math, which for space reasons I can't cover in this book. A fair number are highly specialized for things such as fast encryption and decryption.

In this appendix, I'll present short summaries of the most common machine instructions, the ones you're most likely to use as a beginner to write userspace programs. If you want a more complete (and completely authoritative) instruction reference, see the Intel instruction set documentation:

https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4

Plan to spend a little time with it: The PDF is 5,060 pages long.

Or if that's a triple handful (it is), a useful web distillation of the Intel docs can be found here:

www.felixcloutier.com/x86/index.html

This site basically allows you to click around and find the instructions you're looking for and saves you from having to download and hunt through one gigantic document.

What's Been Removed from x64

During the evolution of the x86 CPUs, instructions have been added and removed with each generation. Most of these are fairly arcane, but there are a few that you may have learned as a beginner in the 32-bit era that are no longer available. Most of these now-obsolete instructions date back to the earliest x86 implementations. The following are the most common ones:

- The BCD math instructions: `AAA`, `AAS`, `AAD`, `AAM`, `DAA`, and `DAS`. These instructions haven't been necessary for decades, and they take up

valuable space in binary instruction encoding.

- The push-all and pop-all instructions: `PUSHA`, `POPA`, `PUSHAD`, and `POPAD`. There are 16 general-purpose registers now, each 8 bytes in size. That's a lot of data to move at one time. Registers must be pushed and popped individually now.
- The `JCXZ` instruction, which jumps when CX=0. This is an ancient 16-bit instruction. Its function is now shared by `JRCXZ` (jump when RCX=0) and `JECXZ` (jump when ECX=0.)
- The `PUSHFD` instruction, which pushes the `EFLAGS` register onto the stack. Use `PUSHFQ` now, which pushes the `RFLAGS` register onto the stack.
- The `POPFD` instruction, which pops a double word (32 bits) from the top of the stack into the `EFLAGS` register. Its job is now done with `POPFQ`, which pops a quadword (64 bits) from the top of the stack into the `RFLAGS` register.

Flag Results

Each instruction contains a flag summary that looks like this. The asterisks present will vary from instruction to instruction:

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*   ? ? * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

The nine most important flags are all represented here. An asterisk indicates that the instruction on that page affects that flag. A blank space under the flag header means that the instruction does not affect that flag in any way. If a flag is affected in a defined way, it will be affected according to these rules:

- **OF:** Set if the result is too large to fit in the destination operand.
- **DF:** Set by `STD`; cleared by `CLD`.
- **IF:** Set by `STI`; cleared by `CLI`. Not used in userspace programming and can be ignored.

- **TF:** For debuggers; not used in userspace programming and can be ignored.
- **SF:** Set when the sign of the result forces the destination operand to become negative.
- **ZF:** Set if the result of an operation is zero. If the result is nonzero, ZF is cleared.
- **AF:** Auxiliary carry used for 4-bit BCD math. Set when an operation causes a carry out of a 4-bit BCD quantity. Not used in x64 work, as the BCD instructions are no longer available in x64 CPUs.
- **PF:** Set if the number of 1 bits in the low byte of the result is even; cleared if the number of 1 bits in the low byte of the result is odd. Used in data communications applications but little else.
- **CF:** Set if the result of an add or shift operation carries out a bit beyond the destination operand; otherwise cleared. May be manually set by `STC` and manually cleared by `CLC` when CF must be in a known state before an operation begins.

Some instructions force certain flags to become undefined. These are indicated by a question mark under the flag header. “Undefined” means *don't count on it being in any particular state*. Until you know that a flag in an undefined state has gone to a defined state by the action of another instruction, do not test or in any other way use the state of the flag.

For a figure showing all flags in detail, see Figure 7.2 in Chapter 7.

Size Specifiers

There is a problem inherent in accessing memory data from assembly language: How much memory data is being acted upon by the instruction in question? Suppose you want to increment a location in memory:

```
inc [rdi+4]
```

So are you incrementing a byte, a word, a double-word, or a quadword? From the instruction as written, there's no way to tell,

and NASM will call you on it. But there's an easy way out. NASM recognizes a number of *size specifiers*, which when used with memory data tell NASM the size of the operation. These are `BYTE`, `WORD`, `DWORD`, and `QWORD`. They specify a data size of 8, 16, 32, and 64 bytes, respectively. The `BYTE` specifier will treat a memory access as an 8-bit quantity, and so on for the others. The improper INC instruction can be fixed by adding an appropriate size specifier:

```
inc qword [rdi+4]
```

Now we know that we're incrementing a 64-bit value beginning at the effective address RDI+4. This rule applies to all of the x64 instructions taking only a single operand: `INC`, `DEC`, `NOT`, `NEG`, `SHR`, `SHL`, `ROR`, and `ROL`. When you're acting on memory, you need a size specifier.

With instructions that take two operands, the issue is more subtle. In any form of a two-operand instruction where one operand is a memory access and the other is a register, NASM infers the size of the access from the size of the register. For example:

```
mov [rdi+rbx*8],rcx
```

Here, NASM sees 64-bit register RCX and thus moves the 64 bits in RCX from RCX to the 8 bytes beginning at `[RDI+RBX*8]`. Any legal effective address can be used. The same holds true for instruction forms that move data from memory to a register:

```
mov r15,[rbp]
```

Because R15 is a 64-bit register, NASM knows to move the 8 bytes starting at `[RBP]` to R15. No need to drop in a size specifier, though having an unnecessary size specifier is not always an error.

The only other case in which a size specifier is required is where one operand is in memory and the other operand is a literal constant:

```
mov word [rdi],42
```

This form copies a value of 42 to a word (two bytes) of memory starting at the address in RDI. Of course, a literal constant can't be the destination operand, so in such instruction forms the literal constant must *always* be the source operand.

To save space, I haven't tried to explain this on every instruction's page in this reference. What you should do is stare at an instruction's form and ask yourself, *what here tells NASM the size of the memory access?* If there's a register as either operand, that tells NASM the access is the size of the register. If the source operand is a literal constant, or if the sole operand is a memory reference, you need a size specifier. And if you figure wrong and don't add a size specifier where one needs to go, NASM will tell you.

Instruction Index

INSTRUCTION	REFERENCE PAGE	TEXT PAGE	CPU
ADC			
ADD			
AND			
BT			386+
CALL			
CLC		Here only	
CLD			
CMP			
DEC			
INC			
INT			
IRET			
J?			
JECXZ			x64+
JMP			
JRCXZ			x64+
LEA			
LOOP			

INSTRUCTION	REFERENCE PAGE	TEXT PAGE	CPU
LOOPNZ/LOOPNE			
LOOPZ/LOOPE			
MOV			
MUL			
NEG			
NOP		Here only	
NOT			
OR			
POP			
POPF			
POPFQ			x64+
PUSH			
PUSHF			
PUSHFQ			x64+
RET			
ROL			
ROR			
SBB		Here only	
SHL			
SHR			
STC		Here only	
STD			
STOS			
SUB			
SYSCALL			x64+
XCHG			

INSTRUCTION	REFERENCE PAGE	TEXT PAGE	CPU
XLAT			
XOR			

ADC: Arithmetic Addition with Carry

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
ADC r/m8,i8
ADC r/m16,i16
ADC r/m32,i32       386+
ADC r/m64,i32       x64+   NOTE: ADC r/m64X,i64 is NOT valid!
ADC r/m8,r8
ADC r/m16,r16
ADC r/m32,r32       386+
ADC r/m64,r64       x64+
ADC r/m16,i8
ADC r/m32,i8
ADC r/m64,i8        x64+
ADC r8,r/m8
ADC r16,r/m16
ADC r32,r/m32       386+
ADC r64,r/m64       x64+
ADC AL,i8
ADC AX,i16
ADC EAX,i32         386+
ADC RAX,i32         x64+   NOTE: ADC RAX,i64 is NOT valid!
```

Examples

```
ADC BX,DI
ADC EAX,5
```

```
ADC AX,0FFFFH
ADC AL,42H
ADC RBP,17H
ADC QWORD [RBX+RSI+Inset],5
```

Notes

ADC adds the source operand and the Carry flag to the destination operand, and after the operation, the result replaces the destination operand. The add operation is an arithmetic add, and the carry allows multiple-precision additions across several registers or memory locations. (To add without taking the Carry flag into account, use the ADD instruction.) All affected flags are set according to the operation. Most importantly, if the result does not fit into the destination operand, the Carry flag is set to 1.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

ADD: Arithmetic Addition

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
ADD r/m8,i8
ADD r/m16,i16
ADD r/m32,i32        386+
ADD r/m64,i32        x64+  NOTE: ADD r/m64,i64 is NOT valid!
ADD r/m16,i8
ADD r/m32,i8         386+
ADD r/m64,i8         x64+
ADD r/m8,r8
ADD r/m16,r16
ADD r/m32,r32        386+
ADD r/m64,r64        x64+
ADD r8,r/m8
ADD r16,r/m16
ADD r32,r/m32        386+
ADD r64,r/m64        x64+
ADD AL,i8
ADD AX,i16
ADD EAX,i32          386+
ADD RAX,i32          x64+  NOTE: ADD RAX,i64 is NOT valid!
```

Examples

```
ADD BX,DI
ADD AX,0FFFFH
ADD AL,42H
ADD [EDI],EAX
AND QWORD [RAX],7BH
```

Notes

`ADD` adds the source operand to the destination operand, and after the operation, the result replaces the destination operand. The add operation is an arithmetic add and does *not* take the Carry flag into account. (To add using the Carry flag, use the `ADC` Add with Carry instruction.) All affected flags are set according to the operation. Most importantly, if the result does not fit into the destination operand, the Carry flag is set to 1.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
```

```
i32 = 32-bit immediate data        i64 = 64-bit immediate
data
d8 = 8-bit signed displacement     d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH       r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

AND: Logical AND

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * ? * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
AND r/m8,i8
AND r/m16,i16
AND r/m32,i32      386+
AND r/m64,i32      x64+   NOTE: AND r/m64,i64 is NOT valid!
AND r/m16,i8
AND r/m32,i8       386+
AND r/m64,i8       x64+
AND r/m8,r8
AND r/m16,r16
AND r/m32,r32      386+
AND r/m64,r64      x64+
AND r8,r/m8
AND r16,r/m16
AND r32,r/m32      386+
AND r64,r/m64      x64+
AND AL,i8
AND AX,i16
AND EAX,i32        386+
AND RAX,i32        x64+ NOTE: AND RAX,i64 is NOT valid!
```

Examples

```
AND BX,DI
AND EAX,5
AND AX,0FFFFH
AND AL,42H
AND [BP+SI],DX
AND QWORD [RDI],42
AND QWORD [RBX],0B80000H
```

Notes

AND performs the AND logical operation on its two operands. Once the operation is complete, the result replaces the destination operand. AND is performed on a bit-by-bit basis, such that bit 0 of the source is ANDed with bit 0 of the destination, bit 1 of the source is ANDed with bit 1 of the destination, and so on. The AND operation yields a 1 if *both* of the operands are 1; and a 0 only if *either* operand is 0. Note that the operation makes the Auxiliary carry flag undefined. CF and OF are cleared to 0, and the other affected flags are set according to the operation's results.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

BT: Bit Test

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
                *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
BT r/m16,r16       386+
BT r/m32,r32       386+
BT r/m64,r64       x64+
BT r/m16,i8        386+
BT r/m32,i8        386+
BT r/m64,i8        x64+
```

Examples

```
BT AX,CX
BT EAX,EDX
BT RAX,5
BT [RAX+RDX],RCX
```

Notes

BT copies a single specified bit from the left operand to the Carry flag, where it can be tested or fed back into a quantity using one of the shift/rotate instructions. Which bit is copied is specified by the right operand. Neither operand is altered by BT.

When the right operand is an 8-bit immediate value, the value specifies the number of the bit to be copied. In `BT AX,5`, bit 5 of AX is copied into CF. When the immediate value exceeds the size of the left operand, the value is expressed modulo the size of the left operand. That is, because there are not 66 bits in EAX, `BT EAX,66` pulls out as many 32s from the immediate value as can be taken, and what remains is the bit number. (Here, 2.) When the right operand is *not* an immediate value, the right operand not only specifies the bit to be

tested but also an offset from the memory reference in the left operand. This is complicated and not covered completely in this book. See a detailed discussion in a full assembly language reference.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

CALL: Call Procedure

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
     <none>        IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
CALL d16           Not valid in x64
CALL d32           Not valid prior to x64
CALL r/m16         Not valid in x64
CALL r/m32         Not valid in x64
CALL r/m64         Not valid prior to x64
```

Examples

```
CALL DoSomething
CALL RAX
CALL QWORD [EBX+ECX+16]
```

Notes

CALL transfers control to a procedure address. Before transferring control, CALL pushes the address of the instruction immediately after itself onto the stack. This allows a RET instruction (see also) to pop the return address into RIP and thus return control to the instruction immediately after the CALL instruction.

In addition to the obvious CALL to a defined label, CALL can transfer control to an address in a 64-bit general-purpose register (r64) and also to an address located in memory. This is shown in the Legal Forms column as m64. CALL m64 is useful for creating jump tables of procedure addresses. D32 is simply a 32-bit unsigned displacement used in most calls to procedure labels. In 64-bit long mode, a d32 displacement is sign-extended to 64 bits.

There are several more variants of the CALL instruction with provisions for working with the protection mechanisms of operating systems. These are not covered here, and for more information you should see an advanced text or a full assembly language reference.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

CLC: Clear Carry Flag (CF)

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
                *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
CLC
```

Examples

```
CLC
```

Notes

`CLC` simply sets the Carry flag (CF) to the cleared (0) state. Use `CLC` in situations where the Carry flag *must* be in a known cleared state before work begins, as when you are rotating a series of words or bytes using the rotate instructions `RCL` and `RCR`.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

CLD: Clear Direction Flag (DF)

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
  *                IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
CLD
```

Examples

```
CLD
```

Notes

`CLD` simply sets the Direction flag (DF) to the cleared (0) state. This affects the adjustment performed by repeated string instructions such as `STOS`, `SCAS`, and `MOVS`. When DF = 0, the destination pointer is increased, and decreased when DF = 1. DF is set to 1 with the `STD` instruction.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

CMP: Arithmetic Comparison

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
CMP r/m8,i8
CMP r/m16,i16
CMP r/m32,i32       386+
CMP r/m64,i32       x64+   NOTE: CMP RAX,i64 is NOT valid!
CMP r/m16,i8
CMP r/m32,i8        386+
CMP r/m64,i8        x64+
CMP r/m8,r8
CMP r/m16,r16
CMP r/m32,r32       386+
CMP r/m64,r64       x64+
CMP r8,r/m8
CMP r16,r/m16
CMP r32,r/m32       386+
CMP r64,r/m64       x64+
CMP AL,i8
CMP AX,i16
CMP EAX,i32         386+
CMP RAX,i32         x64+   NOTE: CMP RAX,i64 is NOT valid!
```

Examples

```
CMP RAX,5
CMP AL,19H
CMP EAX,ECX
CMP QWORD [RBX+RSI+inset],0B80000H
```

Notes

`CMP` compares its two operations and sets the flags to indicate the results of the comparison. *The destination operand is not affected*.

The operation itself is identical to arithmetic subtraction of the source from the destination without borrow (SUB), save that the result does not replace the destination. Typically, CMP is followed by one of the conditional jump instructions; that is, JE to jump if the operands were equal; JNE if they were unequal; and so forth.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

DEC: Decrement Operand

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * *    IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
DEC r/m8
DEC r/m16
DEC r/m32          386+
DEC r/m64          x64+
```

Examples

```
DEC AL
DEC AX
DEC EAX
DEC QWORD [RBX+RSI]
```

Notes

DEC subtracts 1 from its single operand and does *not* affect the Carry flag CF. Be careful about that; it's a common error to try to use CF after a DEC instruction as though it were SUB instead.

DEC acting on memory data forms *must* be used with a data size specifier such as BYTE, WORD, DWORD, and QWORD. See the examples given earlier.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

DIV: Unsigned Integer Division

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
?       ? ? ? ? ?  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
DIV r/m8    Dividend in AX. Quotient in AL; remainder in AH.
DIV r/m16   Dividend in EAX. Quotient in AX; remainder in DX.
DIV r/m32   386+ Dividend in EAX:EDX. Quotient in EAX;
remainder in EDX.
DIV r/m64   x64+ Dividend in RAX:RDX. Quotient in RAX;
remainder in RDX.
```

Examples

```
DIV AL
DIV AX
DIV EAX
DIV QWORD [RDI+RSI] Don't use A or D regs in an effective
address!
```

Notes

`DIV` divides the implicit dividend by the explicit divisor specified as `DIV`'s single operand. For dividing by 8-bit quantities, the dividend is assumed to be in AX. For dividing by 16-bit, 32-bit, and 64-bit quantities, the dividend is assumed to be in two registers, allowing a much greater range of calculation. The least significant portion of the dividend is placed in the "A" register (AX / EAX / RAX), and the most significant portion of the dividend is placed in the "D" register (DX / EDX / RDX). Note that even when there is no "high" portion of the dividend, the "D" register is cleared to 0 by `DIV` and cannot be used to hold independent values while a `DIV` instruction is executed. For more on `DIV`, see the Chapter 7 discussion on p. [203].

Remember that when the operand is a memory value, you *must* place one of the type specifiers `BYTE`, `WORD`, `DWORD`, or `QWORD` before the operand. Also note from the "Legal Forms" section, there is no legal form using an immediate value of any size.

`DIV` leaves no information in the flags. Note, however, that OF, SF, ZF, AF, PF, and CF become undefined after a `DIV` instruction.

```
m8 = 8-bit memory data          m16 = 16-bit memory data
m32 = 32-bit memory data        m64 = 64-bit memory data
i8  = 8-bit immediate data      i16 = 16-bit immediate
data
```

```
i32 = 32-bit immediate data        i64 = 64-bit immediate
data
d8 = 8-bit signed displacement     d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH       r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

INC: Increment Operand

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * *    IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
INC r/m8
INC r/m16
INC r/m32          386+
INC r/m64          x64+
```

Examples

```
INC AL
INC EAX
INC QWORD [RBP]
INC QWORD [RBX+RSI]
```

Notes

INC adds 1 to its single operand and does *not* affect the Carry flag CF. Be careful about that; it's a common error to try to use CF after an INC instruction as though it were ADD instead.

INC acting on memory data forms *must* be used with a data size specifier such as BYTE, WORD, DWORD, and QWORD. See the two examples given earlier.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

J??: Jump If Condition Is Met

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms	Description	Jump If Flags Are
JA/JNBE d8 d32	Jump If Above / Jump If Not Below Or Equal	CF=0 AND ZF=0
JAE/JNB d8 d32	Jump If Above Or Equal / Jump If Not Below	CF=0
JB/JNAE d8 d32	Jump If Below / Jump If Not Above Or Equal	CF=1
JBE/JNA d8 d32	Jump If Below Or Equal / Jump If Not Above	CF=1 OR ZF=1
JC d8 d32	Jump If Carry; synonym for JNAE &	CF=1

	JB	
JNC d8 d32	Jump If Not Carry	CF=0
JE/JZ d8 d32	Jump If Equal / Jump If Zero	ZF=1
JNE/JNZ d8 d32	Jump If Not Equal / Jump If Not Zero	ZF=0
JG/JNLE d8 d32	Jump If Greater / Jump If Not Less Or Equal	ZF=0 AND SF=OF
JGE/JNL d8 d32	Jump If Greater Or Equal / Jump If Not Less	SF=OF
JL/JNGE d8 d32	Jump If Less / Jump If Not Greater Or Equal	SF≠OF
JLE/JNG d8 d32	Jump If Less Than Or Equal/Jump If Not Greater	ZF=1 OR SF≠OF
JO d8 d32	Jump If Overflow	OF=1
JNO d8 d32	Jump If Not Overflow	OF=0
JP/JPE d8 d32	Jump If Parity Set / Jump If Parity Even	PF=1
JNP/JPO d8 d32	Jump If Parity Cleared / Jump If Parity Odd	PF=0
JS d8 d32	Jump If Sign Flag Set	SF=1
JNS d8 d32	Jump If Sign Flag Cleared	SF=0

Examples

```
JB HalfSplit    ;Jumps if CF=1
JLE TooLow      ;Jumps if ZF=1 AND SF=OF
JG NEAR WayOut  ;Jumps to a 32-bit displacement in 32-bit and
                ; 64-bit protected modes
```

Notes

By default all these instructions make a d8 short jump (127 bytes forward or 128 bytes back) if some condition is true, or fall through if the condition is not true. All legal forms also support a 32-bit displacement (d32) for making a jump to anywhere in the code segment. Most of these jump instructions can also accept a 16-bit displacement, but *not* in 64-bit protected mode. I have left out the d16 tags to make the table simpler. To use a 32-bit displacement, you must follow the conditional jump instruction with the qualifier `NEAR`, which tells the assembler to use a 32-bit displacement when

generating binary code. Remember that there is no 64-bit displacement.

The conditions all involve flags, and the flag conditions in question are given to the right of the mnemonic and its description, under the heading “Jump If Flags Are.”

There are often two synonyms for a single conditional jump. For example, `JE` and `JZ` are the same instruction, meaning Jump if ZF is set. The synonyms are there to help you understand the code: Jump if the previous comparison showed parameters equal, or jump if the zero flag is set.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

JECXZ: Jump if ECX=0

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
JECXZ d8           386+
```

Examples

```
JECXZ AllDone      ; Label AllDone must be within +127 or
-128 bytes.
```

Notes

Several instructions use ECX as a count register, and JECXZ allows you to test and jump to see if ECX has become 0. The jump may be only a short jump (that is, no more than 127 bytes forward or 128 bytes back) and will be taken if ECX = 0 at the time the instruction is executed. If ECX is any value other than 0, execution falls through to the next instruction. See also the "Jump on Condition" instructions.

JECXZ is most often used to bypass the ECX = 0 condition when using the LOOP instruction. Because LOOP decrements ECX before testing for ECX = 0, if you enter a loop governed by LOOP with ECX = 0, you will end up iterating the loop 2,147,483,648 (2^{32}) times, hence the need for JECXZ. If you use LOOP and it seems to lock up, check this first.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

JRCXZ: Jump If RCX=0

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
```

```
F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
JRCXZ d8           x64+
```

Examples

```
JRCXZ AllDone      ; Label AllDone must be within +127 or
-128 bytes.
```

Notes

This instruction operates identically to JECXZ except that the register tested is RCX, not ECX. Because it tests RCX, JRCXZ is available only in x64 processors.

JRCXZ is most often used to bypass the RCX = 0 condition when using the LOOP instruction. Because LOOP decrements RCX before testing for RCX = 0, if you enter a loop governed by LOOP with RCX = 0, you will end up iterating the loop 2^{64} times, hence the need for JRCXZ. (If you use LOOP and it seems to lock up the program, check this first.)

```
m8 = 8-bit memory data               m16 = 16-bit memory data
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

JMP: Unconditional Jump

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
JMP d8             In 64-bit mode, displacement sign-extended
to 64 bits
JMP d16            Not supported in 64-bit long mode
JMP d32            In 64-bit mode, displacement sign-extended
to 64 bits
JMP r/m16          Not supported in 64-bit long mode
JMP r/m32          386+; Not supported in 64-bit long mode
JMP r/m64          x64+
```

Examples

```
JMP RightCloseBy   Label ; RightCloseBy must be +127 or -128
bytes.
JMP EAX            ; Not supported in 64-bit mode
JMP RDX
JMP QWORD [RBX+EDI+17]
```

Notes

JMP transfers control unconditionally to the destination given as the single operand. In 64-bit mode, in addition to defined labels, JMP can transfer control to an 8-bit signed offset from RIP, a 32-bit signed offset from RIP, or a 64-bit absolute address (either as an immediate or indirectly through a register or memory). The m64 form is useful for creating jump tables in memory, where a jump table is an array of addresses. For example, `JMP [RBX+RDI+17]` would transfer control to the 64-bit address found at the based-indexed-displacement address `[RBX+RDI+17]`.

No flags are affected and, unlike CALL, no return address is pushed onto the stack. Note that there are additional JMP forms for other modes and exotic work at higher privilege levels than userspace. See an Intel instruction reference for more details.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

LEA: Load Effective Address

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
LEA r16,m
LEA r32,m          386+
LEA r64,m          x64+
```

Examples

```
LEA RBP,MyVariable   ; Loads address expressed by MyVariable
to RBP
```

```
LEA R15,[RAX+RDX*4+17] ; Loads effective address from
calculation to R15
```

Notes

LEA derives the address of the source operand and loads that offset into the destination operand. The destination operand must be a register and *cannot* be memory. The source operand must be a memory operand, but it can be any size. The address stored in the destination operand is the address of the first byte of the source in memory, and the size of the source in memory is unimportant.

This is a good, clean way to place the address of a variable into a register prior to a procedure call or a system call. See SYSCALL.

LEA can also be used to perform register math, since the address specified in the second operand is *calculated* but not *accessed*. The address can thus be an address for which your program does not have permission to access. Any math that can be expressed as a valid address calculation may be done with LEA.

This is one of the few places where NASM does not require a size specifier before an operand providing a memory address, again because LEA calculates the address but moves no data to or from that address.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

LOOP: Loop Until CX/ECX/RCX=0

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
LOOP d8            386+ if using ECX for the counter; 64+ if
using RCX
```

Examples

```
LOOP AllDone       ; Label AllDone must be within +127 or
-128 bytes.
```

Notes

LOOP is a combination decrement counter, test, and jump instruction. It uses CX as the counter in 16-bit modes, ECX in 32-bit modes, or RCX in 64-bit modes. The operation of LOOP is logistically identical in all three modes, and I use 64-bit coding as an example here.

LOOP simplifies code by acting as a DEC RCX instruction, a CMP RCX,0 instruction, and JZ instruction in one, executed in that order. A loop repeat count must be initially loaded into RCX. When the LOOP instruction is executed, it *first* decrements RCX. Then it tests to see if RCX = 0. If RCX is *not* 0, LOOP transfers control to the 8-bit displacement specified as its operand:

```
    MOV RCX,17    ; Loop 17 times
DoIt: CALL CrunchIt
      CALL StuffIt
      LOOP DoIt
```

Here, the two procedure CALLs will be made 17 times. The first 16 times through, RCX will still be nonzero, and LOOP will transfer

control to DoIt. On the 17th pass, however, LOOP will decrement RCX to 0 and then fall through to the next instruction in sequence when it tests CX.

LOOP does *not* alter any flags, even when RCX is decremented to 0. *Warning:* Watch your initial conditions! If you're in 16-bit mode and CX is initially 0, LOOP will decrement it to 65,535 (0FFFFH) and then perform the loop 65,535 times. Worse, if you're working in 32-bit protected mode and enter a loop with ECX = 0, the loop will be performed more than 2 *billion* times, which might be long enough to look like a system lockup. If you're using RCX, well, the loop will go 2^{64} times, which *will* be a system lockup.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

LOOPNZ/LOOPNE: Loop Until CX/ECX/RCX=0 and ZF=0

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
LOOPNZ d8          386+ if using ECX for the counter; 64+ if
using RCX
LOOPNE d8          386+ if using ECX for the counter; 64+ if
using RCX
```

Examples

```
LOOPNZ AllDone     ; Label AllDone must be within +127 or
-128 bytes.
```

Notes

LOOPNZ and LOOPNE are synonyms and generate identical opcodes. Like LOOP, they use CX, ECX, or RCX depending on the "bit-ness" of the CPU. In 64-bit work, LOOPNZ/LOOPNE decrements RCX and jumps to the location specified in the target operand if RCX is not 0 and the Zero flag ZF is 0. Otherwise, execution falls through to the next instruction.

What this means is that the loop is pretty much controlled by ZF. If ZF remains 0, the loop is looped until the counter register is decremented to 0. But as soon as ZF is set to 1, the loop terminates. Think of it as "Loop While Not Zero Flag."

Keep in mind that LOOPNZ and LOOPNE do not *themselves* affect ZF. Some instruction within the loop (typically one of the string instructions) must do something to affect ZF to terminate the loop before CX/ECX/RCX counts down to 0.

```
m8 = 8-bit memory data             m16 = 16-bit memory data
m32 = 32-bit memory data           m64 = 64-bit memory data
i8  = 8-bit immediate data         i16 = 16-bit immediate
data
i32 = 32-bit immediate data        i64 = 64-bit immediate
data
d8 = 8-bit signed displacement     d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH       r16 = AX BX CX DX BP SP SI
DI
```

```
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

LOOPZ/LOOPE: Loop Until CX/ECX/RCX=0 and ZF=1

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
LOOPZ d8           386+ if using ECX for the counter; 64+ if
using RCX
LOOPE d8           386+ if using ECX for the counter; 64+ if
using RCX
```

Examples

```
LOOPZ AllDone      ; Label AllDone must be within +127 or
-128 bytes.
```

Notes

LOOPZ and LOOPE are synonyms and generate identical opcodes. Like LOOP, they use CX, ECX, or RCX depending on the current “bit-ness” of the CPU. In 64-bit work, LOOPZ/LOOPE first decrements RCX and jumps to the location specified in the target operand if RCX is not 0 and the Zero flag ZF is 1. Otherwise, execution falls through to the next instruction.

This means the loop is pretty much controlled by ZF. If ZF remains 1, the loop is looped until the counter register is decremented to 0. But as soon as ZF is cleared to 0, the loop terminates. Think of it as “Loop While Zero Flag.”

Remember that LOOPZ/LOOPE do not *themselves* affect ZF. Some instruction within the loop (typically one of the string instructions) must do something to zero ZF to terminate the loop before CX/ECX/RCX counts down to 0.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

MOV: Copy Right Operand into Left Operand

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
MOV r/m8,r8
MOV r/m16,r16
MOV r/m32,r32      386+
MOV r/m64,r64      x64+
MOV r8,r/m8
MOV r16,r/m16
MOV r32,r/m32      386+
MOV r64,r/m64      x64+
MOV r/m8,i8
MOV r/m16,i8
```

```
MOV r/m16,i16
MOV r/m32,i8        386+
MOV r/m32,i32       386+
MOV r/m64,i8        x64+
MOV r/m64,i64       x64+
MOV r8,i8
MOV r16,i16
MOV r32,i32         386+
MOV r64,i64         x64+
```

Examples

```
MOV AX,BP
MOV R14,RDX
MOV [EBP],EAX
MOV RAX,[RDX]
MOV RBP,17H
MOV QWORD [RBX+RSI+Inset],5
```

Notes

`MOV` is perhaps the most-used of all instructions. The source (right) operand is copied into the left (destination) operand. The source operand is not changed. The flags are not affected.

Note that there are additional forms of `MOV` that deal with segment : offset addressing, which is not used in x64 userspace and which I am not covering in this book. See the Intel documentation for more information.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

MOVS: Move String

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
MOVSB
MOVSW
MOVSD
MOVSQ
```

Examples

```
MOVSB               ;Copies byte at [RSI] to [RDI]
MOVSW               ;Copies word at [RSI] to [RDI]
MOVSD               ;Copies double word at [RSI] to [RDI]
REP MOVSB           ;Copies memory region starting at [RSI] to
region
                    ;  starting at [RDI], for RCX repeats, one
byte
                    ;  at a time
```

Notes

`MOVS` copies memory in 8-bit (`MOVSB`), 16-bit (`MOVSW`), 32-bit (`MOVSD`), or 64-bit (`MOVSQ`) chunks, from the address stored in RSI to the address stored in RDI. The mnemonic that you use from these four is about the size of the chunks, *not* the mode you're using. For example, you can use `MOVSB` in x64 long mode if you need to move data a byte at a time. If you need to move dwords, use `MOVSD`, etc.

By placing an operation repeat count (*not* a byte, word, dword, or qword count!) in RCX and preceding the mnemonic with the `REP` prefix, `MOVS` can do an automatic “machine-gun” copy of data from a

memory region starting at `[RSI]` to a memory region starting at `[RDI]`.

After each copy operation, RSI and RDI are adjusted (see the next paragraph) by 1 (for 8-bit operations), 2 (for 16-bit operations), 4 (for 32-bit operations), or 8 (for 64-bit operations) and RCX is decremented by 1. Don't forget that RCX counts *operations* (the number of times a data item is copied from source to destination) and *not* bytes!

Adjusting means incrementing RSI and RDI if the Direction flag is cleared (by CLD) or decrementing RSI and RDI if the Direction flag has been set (by STD). The Direction flag DF thus determines whether your copy operation moves up-memory if DF is cleared (0) and down-memory if DF is set (1).

There are additional forms of the `REP` prefix (`REPE`, `REPNE`, `REPZ`, and `REPNZ`) that add the ability to terminate a `MOVS` operation before the count register goes to 0, by checking the state of the Zero flag ZF. Those additional forms are not covered in this book. See the Intel documentation for details.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

MOVSX: Copy with Sign Extension

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
MOVSX r16,r/m8     386+
MOVSX r32,r/m8     386+
MOVSX r64,r/m8     x64+
MOVSX r32,r/m16    386+
MOVSX r64,r/m16    x64+
```

Examples

```
MOVSX AX,AL
MOVSX CX,BYTE [EDI]       ; Acts on the byte at EDI
MOVSX ECX,DL
MOVSX RSI,QWORD [RBX+RDI] ; Acts on the doubleword at EBX+EDI
```

Notes

`MOVSX` operates like `MOV` but copies values from source operand to the destination operand with sign extension. That is, it carries the sign bit of the smaller source operand to the sign bit of the larger destination operand. This way, for example, a 16-bit signed value in AX will still be a signed value when copied into 32-bit register EDX or 64-bit register RDX. Without sign extension, the sign bit of AX would simply become another bit in the binary value copied into RDX, and the value in RDX would bear no resemblance to the supposedly identical value in AX.

The destination operand must be a register. `MOVSX` can copy data *from* a memory location, but not *to* a memory location. Also note that the destination operand must be a wider value than the source

operand; that is, MOVSX will copy from an 8-bit or 16-bit value to a 32-bit value, but not a 16-bit to a 16-bit, nor 32-bit to 32-bit.

MOVSX is present only in 386 and later CPUs. It does not affect any flags.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

MUL: Unsigned Integer Multiplication

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       ? ? ? ? *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
MUL r/m8          Dividend in AX. Quotient in AL; remainder in
AH.
MUL r/m16         Dividend in EAX. Quotient in AX; remainder
in DX.
MUL r/m32  386+   Dividend in EAX:EDX. Quotient in EAX;
remainder in EDX.
MUL r/m64  x64+   Dividend in RAX:RDX. Quotient in RAX;
remainder in RDX.
```

Examples

```
MUL CH               ; AL * CH --> AX
MUL BX               ; AX * BX --> DX:AX
MUL ECX              ; EAX * ECX --> EDX:EAX
MUL DWORD [EBX+EDI] ; EAX * [EBX+EDI] --> EDX:EAX
MUL QWORD [R14]      ; RAX * [R14] --> RDX:RAX
```

Notes

MUL multiplies its single operand by AL, AX, EAX, or RAX, and the result is placed in AX, in DX:AX, in EDX:EAX, or in RDX:RAX. If MUL is given an 8-bit operand (either an 8-bit register or an 8-bit memory operand), the results will be placed in AX. This means that AH will be affected, even if the results will fit entirely in AL.

Similarly, if MUL is given a 16-bit operand, the results will be placed in DX:AX, *even if the entire result will fit in AX!* It's easy to forget that MUL affects DX on 16-bit multiplies, EDX in 32-bit multiplies, and RDX in 64-bit multiplies. Keep that in mind! Also, if you're multiplying a value in memory, you must add the size specifier BYTE, WORD, DWORD, or QWORD.

Note: It's easy to assume that IMUL is identical to MUL save for IMUL's ability to operate on signed values. Not so: IMUL has more legal instruction forms and is considerably more complex than MUL. For more details, see the Intel documentation.

The Carry and Overflow flags are cleared to 0 if the result value is 0; otherwise, both are set to 1. Remember that SF, ZF, AF, and PF become undefined after MUL.

```
m8 = 8-bit memory data                m16 = 16-bit memory data
m32 = 32-bit memory data              m64 = 64-bit memory data
i8  = 8-bit immediate data            i16 = 16-bit immediate
data
i32 = 32-bit immediate data           i64 = 64-bit immediate
data
d8 = 8-bit signed displacement        d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH          r16 = AX BX CX DX BP SP SI
```

```
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

NEG: Negate (Two's Complement; i.e., Multiply by −1)

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
NEG r/m8
NEG r/m16
NEG r/m32          386+
NEG r/m64          x64+
```

Examples

```
NEG CH
NEG BX
NEG ECX
NEG DWORD [EBX]
NEG QWORD [R14+RDI*4]
```

Notes

`NEG` is the assembly language equivalent of multiplying a value by 1. Keep in mind that negation is *not* the same as simply inverting each bit in the operand. (Another instruction, `NOT`, does that.) The process is also known as generating the *two's complement* of a value. The two's complement of a value added to that value yields zero.

−1 = $FF; −2 = $FE; −3 = $FD; and so forth.

If the operand is 0, CF is cleared, and ZF is set; otherwise, CF is set, and ZF is cleared. If the operand contains the maximum negative value for the operand size, the operand does not change, but OF and CF are set. SF is set if the result is negative, or else SF is cleared. PF is set if the low-order 8 bits of the result contain an even number of set (1) bits; otherwise, PF is cleared.

`NEG` acting on memory data forms *must* be used with a data size specifier such as `BYTE`, `WORD`, `DWORD`, and `QWORD`. See the two examples given earlier.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

NOP: No Operation

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
NOP
```

Examples

```
NOP ;AllDone   ; NOP replaces a jump instruction for
debugging purposes,                ; but the
 parameter is commented out, since NOP                ; can't
take a label
 as a parameter.
```

Notes

NOP, the easiest-to-understand of all x86/x64-family machine instructions, simply does nothing. Its job is to take up space in sequences of instructions. The flags are not affected. NOP is used for "NOPing out" machine instructions during debugging, leaving space for future procedure or interrupt calls.

In ancient times, NOP was used for padding timing loops. This makes sense on the surface but can no longer be done. Modern CPUs have the ability to perform various context-sensitive optimizations on executing code inside the CPU. *Precise assembly-time prediction of instruction execution time is no longer possible!*

```
m8 = 8-bit memory data             m16 = 16-bit memory data
m32 = 32-bit memory data           m64 = 64-bit memory data
i8  = 8-bit immediate data         i16 = 16-bit immediate
data
i32 = 32-bit immediate data        i64 = 64-bit immediate
data
d8 = 8-bit signed displacement     d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH       r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

NOT: Logical NOT (One's Complement)

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
NOT r/m8
NOT r/m16
NOT r/m32          386+
NOT r/m64          x64+
```

Examples

```
NOT CL
NOT DX
NOT ECX
NOT DWORD [EDI]
NOT QWORD [RDI+RCX*4]
```

Notes

NOT inverts each individual bit within the operand separately. That is, every bit that was 1 becomes 0, and every bit that was 0 becomes 1. This is the “logical NOT” or “one's complement” operation. See the NEG instruction for the negation, or two's complement, operation.

After execution of NOT, the value FFH would become 0; the value AAH would become 55H. Note that the Zero flag is *not* affected, even when NOT forces its operand to 0.

NOT acting on memory data forms *must* be used with a data size specifier such as BYTE, WORD, DWORD, and QWORD. See the two examples given earlier.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
```

```
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

OR: Logical OR

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * ? * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
OR r/m8,i8
OR r/m16,i16
OR r/m32,i32        386+
OR r/m64,i32        x64+  NOTE: OR r/m64,i64 is NOT valid!
OR r/m16,i8
OR r/m32,i8         386+
OR r/m64,i8         x64+
OR r/m8,r8
OR r/m16,r16
OR r/m32,r32        386+
OR r/m64,r64        x64+
OR r8,r/m8
OR r16,r/m16
OR r32,r/m32        386+
OR r64,r/m64        x64+
OR AL,i8
OR AX,i16
```

```
OR EAX,i32          386+
OR RAX,i32          x64+  NOTE: OR RAX,i64 is NOT valid!
```

Examples

```
OR BX,DI
OR EAX,5
OR AX,0FFFFH
OR AL,42H
OR [BP+SI],DX
OR [RDI],RAX
OR QWORD [RBX],0B80000H
```

Notes

OR performs the OR logical operation between its two operands. Once the operation is complete, the result replaces the destination operand. OR is performed on a bit-by-bit basis, such that bit 0 of the source is ORed with bit 0 of the destination, bit 1 of the source is ORed with bit 1 of the destination, and so on. The OR operation yields a 1 if one of the operands is 1; and a 0 only if both operands are 0. Note that the OR instruction makes the Auxiliary Carry flag undefined. CF and OF are cleared to 0, and the other affected flags are set according to the operation's results.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

POP: Copy Top of Stack into Operand

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
POP m16
POP m32   32-bit CPUs only: not valid in x64 mode
POP m64   x64+
POP r16
POP r32   32-bit CPUs only: not valid in x64 mode
POP r64   x64+
```

Examples

```
POP DX
POP RCX
POP QWORD [RDI]
POP QWORD [RDI+RCX*4]
```

Notes

It is impossible to pop an 8-bit item from the stack. Also remember that the *top of the stack* is defined (in 16-bit modes) as the word at address SS:SP, and there's no way to override that using prefixes. In 32-bit modes, the top of the stack is the `DWORD` at [ESP]. In 64-bit mode, the top of the stack is the `QWORD` at [RSP]. The 32-bit forms of `POP` are invalid in 64-bit mode. There is a separate pair of instructions, `PUSHF/D/Q` and `POPF/D/Q`, for pushing and popping the Flags register.

`POP` acting on memory data forms *must* be used with a data size specifier such as `BYTE`, `WORD`, `DWORD`, and `QWORD`. See the examples given earlier.

There are several forms of `POP` for popping segment registers, but these forms cannot be used in userspace programming. For details, see the Intel documentation.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

POPF/D/Q: Copy Top of Stack into Flags Register

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
* * * * * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
POPF
POPFD               32-bit CPUs only: Invalid in 64-bit mode
POPFQ               x64+
```

Examples

```
POPF                ;Pops 16-bit top of stack into the FLAGS
register
POPFD               ;Pops 32-bit top of stack into the EFLAGS
```

```
register
POPFQ               ;Pops 64-bit top of stack into the RFLAGS
register
```

Notes

These instructions pop data at the top of the stack into the flags register appropriate to the mode. `POPF` pops 16 bits into the FLAGS register. `POPFD` pops 32 bits into EFLAGS. `POPFQ` pops 64 bits into RFLAGS. The stack pointer is incremented by 2 after `POPF`, 4 after `POPFD`, and 8 after `POPFQ`. `POPF` may be used in 64-bit mode, if `PUSHF` was done earlier. Remember that RFLAGS contains EFLAGS, which in turn contains FLAGS. When you pop the top 16 bits off the stack with `POPF`, you're popping those bits into the lowest 16 bits of both EFLAGS and RFLAGS.

`POPFD` is invalid in 64-bit mode.

Pushing and popping the CPU flags is a subtle business and more complex than you might think. It's mostly done by the operating system and isn't often done in userspace programming. For details, see the Intel documentation.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

PUSH: Push Operand onto Top of Stack

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
PUSH r/m16
PUSH r/m32          32-bit CPUs only; not valid in 64-bit mode
PUSH r/m64          x64+
PUSH i16
PUSH i32            32-bit CPUs only; not valid in 64-bit mode
PUSH i64            x64+
```

Examples

```
PUSH DX
PUSH R13
PUSH QWORD 5
PUSH QWORD 034F001h
```

Notes

`PUSH` decrements the stack pointer and then copies its operand onto the stack. The stack pointer then points at the new data. (Before the stack is used, the stack pointer points to empty memory.) It is impossible to push 8-bit data onto the stack. Also remember that the *top of the stack* is defined (in 16-bit modes) as the word at address SS:SP, and there's no way to override that using prefixes. In 32-bit modes the top of the stack is the `DWORD` at `[ESP]`. In 64-bit mode the top of the stack is the `QWORD` at `[RSP]`. The 32-bit forms are not valid in x64. There is a separate set of instructions, `PUSHF/D/Q` and `POPF/D/Q`, for pushing and popping the FLAGS/EFLAGS/RFLAGS registers.

PUSH acting on memory data forms *must* be used with a data size specifier such as BYTE, WORD, DWORD, and QWORD. See the examples given earlier.

There are special forms of PUSH for pushing the segment registers, but those forms are not listed here since they cannot be used in ordinary Linux userspace programming.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

PUSHF/D/Q: Push Flags Onto the Stack

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
PUSHF
PUSHFD             32-bit CPUs only: Invalid in 64-bit mode
PUSHFQ             x64+
```

Examples

```
PUSHF               ;Pushes 16-bit FLAGS register onto the
stack
PUSHFD              ;Pushes 32-bit EFLAGS register onto the
stack
PUSHFQ              ;Pushes 64-bit RFLAGS register onto the
stack
```

Notes

These three instructions push the FLAGS/EFLAGS/RFLAGS register onto the stack. `PUSHF` pushes the 16-bit FLAGS register. `PUSHFD` pushes 32-bit EFLAGS. `PUSHFQ` pushes 64-bit RFLAGS. The stack pointer is decremented *before* the flags values are pushed onto the stack. Remember that RFLAGS contains EFLAGS, which in turn contains FLAGS. When you pop the top 16 bits off the stack with `POPF`, you're popping those bits into the lowest 16 bits of both EFLAGS and RFLAGS.

`PUSHFD` is invalid in 64-bit mode.

Pushing and popping the CPU flags is a subtle business, and more complex than you might think. It's mostly done by the operating system and isn't often done in userspace programming. For details, see the Intel documentation.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

RET: Return from Procedure

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
     <none>        IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
RET
RETN
RET i16
RETN i16
```

Examples

```
RET
RET 16             ; Removes 2 64-bit (8 byte) parameters
from the stack
```

Notes

There are two kinds of returns from procedures: Near and Far, where Near is within the current code segment and Far is to some other code segment. This is not an issue in 32-bit and 64-bit protected mode, for which there is only one code segment in userspace code and all calls and returns are Near. Ordinarily, the `RET` form is used, and the assembler resolves it to a Near or Far return opcode to match the procedure definition's use of the `NEAR` or `FAR` specifier if one is present. Specifying `RETN` may be done for Near returns when necessary.

`RET` may take an operand indicating how many bytes of stack space are to be released on returning from the procedure. This figure is subtracted from the stack pointer to erase data items that had been pushed onto the stack for the procedure's use immediately prior to the procedure call. Make sure you calculate the immediate value

correctly, or the stack will be corrupted and probably trigger a segmentation fault.

There are additional variants of the `RET` instruction with provisions for working with the protection mechanisms of operating systems. These are not covered here, and for more information, you should see an advanced text or a full assembly language reference.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

ROL/ROR: Rotate Left/Rotate Right

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*              *   IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
ROL/ROR r/m8,1
ROL/ROR r/m16,1
ROL/ROR r/m32,1    386+
ROL/ROR r/m64,1    x64+
ROL/ROR r/m8,CL
ROL/ROR r/m16,CL
ROL/ROR r/m32,CL   386+
```

```
ROL/ROR r/m64,CL   x64+
ROL/ROR r/m8,i8    286+
ROL/ROr r/m16,i8   286+
ROL/ROR r/m32,i8   386+
ROL/ROR m/m64,i8   x64+
```

Examples

```
ROL/ROR AX,1
ROL/ROR DWORD {EBX+ESI],9
ROL/ROR R14,17
ROL/ROR QWORD [BPI],CL
```

Notes

ROL and ROR rotate the bits within the destination operand to the left (ROL) and the right (ROR), where left is toward the most significant bit (MSB) and right is toward the least significant bit (LSB). A rotate is a shift (see SHL and SHR) that wraps around: For ROL, the leftmost bit of the operand is shifted into the rightmost bit, and all intermediate bits are shifted one bit to the left. Going the other way, for ROR the rightmost bit of the operand is shifted into the leftmost bit, with all other bits moving one bit to the right. Except for the direction that the shift operation takes, ROL is identical to ROR, which is why I treat both instructions on the same page.

The number of bit positions shifted may be specified either as an 8-bit immediate value or by the value in CL—*not* CX/ECX/RCX. (The 8086 and 8088 may only use the forms shifting by the immediate value 1.) Note that while CL may accept a value up to 255, it is meaningless to shift by any value larger than the native word size. The 286 and later limit the number of shift operations performed to the native word size, except when running in Virtual 86 mode.

With ROL, the leftmost bit is copied into CF on each shift operation. With ROR, the rightmost bit is copied into CF on each shift operation. For both ROL and ROR, OF is modified *only* by the shift-by-one forms. After shift-by-CL forms, *OF becomes undefined*. However, if the number of bits to shift by is 0, none of the flags is affected.

ROL or ROR acting on memory data forms *must* be used with a data size specifier such as BYTE, WORD, DWORD, and QWORD. See the examples

given earlier.

Although I'm not giving them a separate page here, `RCL` and `RCR` work the same way, except that the Carry flag CF is part of the rotation, in essence adding a bit to the rotation.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

SBB: Arithmetic Subtraction with Borrow

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
SBB r/m8,i8
SBB r/m16,i16
SBB r/m32,i32      386+
SBB r/m64,i32      x64+  NOTE: SBB r/m64,i64 is NOT valid!
SBB r/m16,i8
SBB r/m32,i8       386+
SBB r/m64,i8       x64+
SBB r/m8,r8
SBB r/m16,r16
SBB r/m32,r32      386+
```

```
SBB r/m64,r64      x64+
SBB r8,r/m8
SBB r16,r/m16
SBB r32,r/m32      386+
SBB r64,r/m64      x64+
SBB AL,i8
SBB AX,i16
SBB EAX,i32        386+
SBB RAX,i32        x64+  NOTE: SBB RAX,i64 is NOT valid!
```

Examples

```
SBB DX,DI
SBB AX,04B2FH
SBB AL,CBH
SBB BP,19H
SBB DWORD [ESI],EAX
SBB QWORD [RAX],7BH
```

Notes

SBB performs a subtraction with borrow, where the source operand is subtracted from the destination operand, and then the Carry flag is subtracted from the result. The result then replaces the destination operand. If the result is negative, the Carry flag is set. To subtract without taking the Carry flag into account (i.e., without borrowing), use the SUB instruction.

```
m8 = 8-bit memory data             m16 = 16-bit memory data
m32 = 32-bit memory data           m64 = 64-bit memory data
i8  = 8-bit immediate data         i16 = 16-bit immediate
data
i32 = 32-bit immediate data        i64 = 64-bit immediate
data
d8 = 8-bit signed displacement     d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH       r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

SHL/SHR: Shift Left/Shift Right

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * ? * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
SHL/SHR r/m8,1
SHL/SHR r/m16,1
SHL/SHR r/m32,1    386+
SHL/SHR r/m64,1    x64+
SHL/SHR r/m8,CL
SHL/SHR r/m16,CL
SHL/SHR r/m32,CL   386+
SHL/SHR r/m64,CL   x64+
SHL/SHR r/m8,i8    286+
SHL/SHR r/m16,i8   286+
SHL/SHR r/m32,i8   386+
SHL/SHR m/m64,i8   x64+
```

Examples

```
SHL/SHR AX,1
SHL/SHR DWORD {EDX+ESI],4
SHL/SHR R12,15
SHL/SHR QWORD [RDI],CL
```

Notes

SHL and SHR shift the bits in their destination operands by a count given in the source operand. SHL shifts the bits within the destination operand to the left, where left is toward the most significant bit (MSB). SHR shifts the bits within the destination operand to the right, where right is toward the least significant bit (LSB).The number of bit positions shifted may be specified either as an 8-bit immediate value or by the value in CL—*not* CX/ECX/RCX. (The 8086 and 8088

are limited to the immediate value 1.) Note that while CL may accept a value up to 255, it is meaningless to shift by any value larger than the native word size. The 286 and later limit the number of shift operations performed to the native word size except when running in Virtual 86 mode.

With SHL, the leftmost bit of the operand is shifted into CF; the rightmost bit is cleared to 0. With SHR, the rightmost bit is shifted into CF; the leftmost bit is cleared to 0. The Auxiliary Carry flag AF becomes undefined after both SHL and SHR. OF is modified *only* by the shift-by-one forms. After any of the shift-by-CL forms, OF becomes undefined.

SHL or SHR acting on memory data forms *must* be used with a data size specifier such as BYTE, WORD, DWORD, and QWORD. See the examples given earlier.

SHL is a synonym for SAL (Shift Arithmetic Left). SHR is a synonym for SAR (Shift Arithmetic Right.) Except for the direction the shift operation takes, SHL is identical to SHR.

```
m8 = 8-bit memory data                m16 = 16-bit memory data
m8 = 8-bit memory data               m16 = 16-bit memory data
m32 = 32-bit memory data             m64 = 64-bit memory data
i8  = 8-bit immediate data           i16 = 16-bit immediate
data
i32 = 32-bit immediate data          i64 = 64-bit immediate
data
d8 = 8-bit signed displacement       d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH         r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

STC: Set Carry Flag (CF)

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
                *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
STC
```

Examples

```
STC
```

Notes

`STC` changes the Carry flag CF to a known set state (1). Use it prior to some task that needs a bit in the Carry flag. The `CLC` instruction is similar and will clear CF to a known state of 0.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

STD: Set Direction Flag (DF)

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
  *                IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
STD
```

Examples

```
STD
```

Notes

STD simply changes the Direction flag DF to the set (1) state. This affects the adjustment performed by repeated string instructions such as STOS, SCAS, and MOVS. Typically, when DF = 0, the destination pointer is increased and decreased when DF = 1. DF is cleared to 0 with the CLD instruction.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

STOS/B/W/D/Q: Store String

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
STOS m8
STOS m16
STOS m32
STOS m64
STOSB               ; For 8-bit operations
STOSW               ; For 16-bit operations
STOSD               ; For 32-bit operations
STOSQ               ; For 64-bit operations
```

Examples

```
STOSB               ; Stores AL to [EDI/RDI]
REP STOSB           ; Stores AL to [EDI/RDI] and up, for
ECX/RCX repeats
STOSW               ; Stores AX to [EDI/RDI]
STOSD               ; Stores EAX to [EDI/RDI]
REP STOSQ           ; Stores RAX to [EDI/RDI] and up, for
ECX/RCX repeats
```

Notes

`STOS` stores AL (for 8-bit store operations), AX (for 16-bit operations), and EAX (for 32-bit operations) or RAX (for 64-bit operations) to the location at `[EDI]` / `[RDI]`. For 16-bit legacy modes, ES must contain the segment address of the destination and cannot be overridden. For 32-bit and x64 protected modes, all segments are congruent, and thus ES does not need to be specified explicitly. Similarly, DI, EDI, or RDI must always contain the destination offset. The `STOS` form must always have an operand specifying a

memory location and size. The `STOSB`, `STOSW`, `STOSD`, and `STOSQ` forms contain the size of the operation in their mnemonics, and their operands are implicit, with the store operation going to a memory address in EDI or RDI.

By placing an operation repeat count (*not* a byte count!) in CX/ECX/RCX and preceding the mnemonic with the `REP` prefix, `STOS` can do an automatic "machine-gun" store of AL/AX/EAX/RAX into successive memory locations beginning at the initial address `[DI]`, `[EDI]`, or `[RDI]`. After each store, the DI/EDI/RDI register is adjusted (see the next paragraph) by 1 (for 8-bit store operations), 2 (for 16-bit store operations), 4 (for 32-bit store operations), or 8 (for 64-bit store operations, and CX/ECX/RCX is decremented by 1. Don't forget that CX/ECX/RCX counts *operations* (the number of times a data item is stored to memory) and *not* bytes!

Adjusting means incrementing if the Direction flag is cleared (by `CLD`) or decrementing if the Direction flag has been set (by `STD`).

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

SUB: Arithmetic Subtraction

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
```

```
*       * * * * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
SUB r/m8,i8
SUB r/m16,i16
SUB r/m32,i32       386+
SUB r/m64,i32       x64+   NOTE: SUB r/m64,i64 is NOT valid!
SUB r/m16,i8
SUB r/m32,i8        386+
SUB r/m64,i8        x64+
SUB r/m8,r8
SUB r/m16,r16
SUB r/m32,r32       386+
SUB r/m64,r64       x64+
SUB r8,r/m8
SUB r16,r/m16
SUB r32,r/m32       386+
SUB r64,r/m64       x64+
SUB AL,i8
SUB AX,i16
SUB EAX,i32         386+
SUB RAX,i32         x64+   NOTE: SUB RAX,i64 is NOT valid!
```

Examples

```
SUB AX,DX
SUB AL,DL
SUB EBP,17
SUB RAX,0FFFBH  ; The i32 value is sign-extended to 64 bits
                ;   before the operation
SUB DWORD [EDI],EAX
AND QWORD [RAX],7BH ; The i32 value is sign-extended to 64
bits
                    ;   before the operation
```

Notes

`SUB` performs a subtraction without borrow, where the source operand is subtracted from the destination operand, and the result replaces the destination operand. If the result is negative, the Carry flag CF is set.

In 64-bit mode, 32-bit source operands are sign-extended to 64 bits before the subtraction operation happens.

Multiple-precision subtraction can be performed by following `SUB` with `SBB` (Subtract with Borrow), which takes the Carry flag into account as an arithmetic borrow.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

SYSCALL: Fast System Call into Linux

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
      <none>       IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
SYSCALL
```

Examples

```
SYSCALL
```

Notes

SYSCALL makes a fast call to a predefined operating system service routine. (This does *not* include calls into the C library!) It is available only in 64-bit mode. There are currently 335 such service routines. These routines do not have names but are selected by a number. Typically, registers are loaded with the number of the desired service routine and values appropriate to the chosen service routine before SYSCALL is executed.

SYSCALL trashes RCX and R11. All other registers are preserved. For a list of available x64 Linux system calls, see these sites, which were available at this edition's publication date in 2023:

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64

https://hackeradam.com/x86-64-linux-syscalls

SYSCALL replaces the INT 80 calling protocol in 32-bit Linux. Remember that the numbers of the x64 system calls are *not* the same as those from 32-bit x86 Linux!

```
m8 = 8-bit memory data                m16 = 16-bit memory data
m32 = 32-bit memory data              m64 = 64-bit memory data
i8  = 8-bit immediate data            i16 = 16-bit immediate
data
i32 = 32-bit immediate data           i64 = 64-bit immediate
data
d8 = 8-bit signed displacement        d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH          r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

XCHG: Exchange Operands

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
     <none>        IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
XCHG r/m8,r8
XCHG r/m16,r16
XCHG r/m32,r32     386+
XCHG r/m64,r64     x64+
XCHG r8,r/m8
XCHG r16,r/m16
XCHG r32,r/m32     386+
XCHG r64,r/m64     x64+
XCHG AX,r16
XCHG EAX,r32       386+
XCHG RAX,r64       x64+
XCHG r16,AX
XCHG r32,EAX       386+
XCHG r64,RAX       x64+
```

Examples

```
XCHG AL,AH
XCHG EAX,EBX
XCHG R12,[RSI+Offset]
XCHG [RDI],RDX
```

Notes

XCHG exchanges the contents of its two operands. The two operands must be the same size.

```
m8 = 8-bit memory data             m16 = 16-bit memory data
m32 = 32-bit memory data           m64 = 64-bit memory data
i8  = 8-bit immediate data         i16 = 16-bit immediate
data
```

```
i32 = 32-bit immediate data        i64 = 64-bit immediate
data
d8 = 8-bit signed displacement     d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH       r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

XLAT: Translate Byte Via Table

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
     <none>        IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
XLAT
XLATB
```

Examples

```
XLAT               ; 32-bit: Loads AL with byte table entry
at EBX+AL
XLAT               ; 64-bit: Loads AL with byte table entry
at RBX+AL
```

Notes

`XLAT` and its synonym `XLATB` perform a table translation of the 8-bit value in AL. All operands are implicit. The value in AL is treated as the index into a table in memory, located at the address contained in EBX (in 32-bit mode) or RBX (in x64 mode). When `XLAT` is executed, the value at `[EBX+AL]` / `[RBX+AL]` replaces the value previously in AL.

AL is hard-coded as an implicit operand; no other register may be used.

The table located at the 32-bit or 64-bit address in EBX/RBX does not have to be 256 bytes in length, but a value in AL larger than the length of the table will result in an undefined value being placed in AL.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```

XOR: Exclusive OR

Flags Affected

```
O D I T S Z A P C  OF: Overflow flag  TF: Trap flag AF: Aux
carry
F F F F F F F F F  DF: Direction flag SF: Sign flag PF:
Parity flag
*       * * ? * *  IF: Interrupt flag ZF: Zero flag CF: Carry
flag
```

Legal Forms

```
XOR r/m8,i8
XOR r/m16,i16
XOR r/m32,i32      386+
XOR r/m64,i32      x64+  NOTE: XOR r/m64,i64 is NOT valid!
XOR r/m16,i8
XOR r/m32,i8       386+
XOR r/m64,i8       x64+
```

```
XOR r/m8,r8
XOR r/m16,r16
XOR r/m32,r32       386+
XOR r/m64,r64       x64+
XOR r8,r/m8
XOR r16,r/m16
XOR r32,r/m32       386+
XOR r64,r/m64       x64+
XOR AL,i8
XOR AX,i16
XOR EAX,i32         386+
XOR RAX,i32         x64+ NOTE: XOR RAX,i64 is NOT valid!
```

Examples

```
XOR BX,DI
XOR EAX,5
XOR AX,0FFFFH
XOR AL,42H
XOR [BP+SI],DX
XOR [RDI],RAX
XOR QWORD [RBX],0B80000H
```

Notes

XOR performs a bitwise exclusive OR logical operation between its two operands. Once the operation is complete, the result replaces the destination operand. The XOR operation is performed on a bit-by-bit basis, such that bit 0 of the source is XORed with bit 0 of the destination, bit 1 of the source is XORed with bit 1 of the destination, and so on. The XOR operation yields a 1 if the operands are different and a 0 if the operands are the same. Note that the XOR instruction makes the Auxiliary Carry flag AF undefined. CF and OF are cleared to 0, and the other affected flags are set according to the operation's results.

When XOR is used between a 64-bit value and an immediate value, the immediate value cannot be 64-bits in size. The immediate value may be only 32 bits in size.

Performing XOR between a register and itself is a common way of clearing a register to 0. There is no form to use XOR on a memory value against itself, as only one of XOR's two operands may be a

memory value. Therefore, XOR cannot be used to zero a memory location.

```
m8 = 8-bit memory data              m16 = 16-bit memory data
m32 = 32-bit memory data            m64 = 64-bit memory data
i8  = 8-bit immediate data          i16 = 16-bit immediate
data
i32 = 32-bit immediate data         i64 = 64-bit immediate
data
d8 = 8-bit signed displacement      d16 = 16-bit signed
displacement
d32 = 32-bit unsigned displacement NOTE: There is no 64-bit
displacement
r8 = AL AH BL BH CL CH DL DH        r16 = AX BX CX DX BP SP SI
DI
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
r64 = RAX RBX RCX RDX RBP RSP RSI RDI R8 R9 R10 R11 R12 R13
R14 R15
```