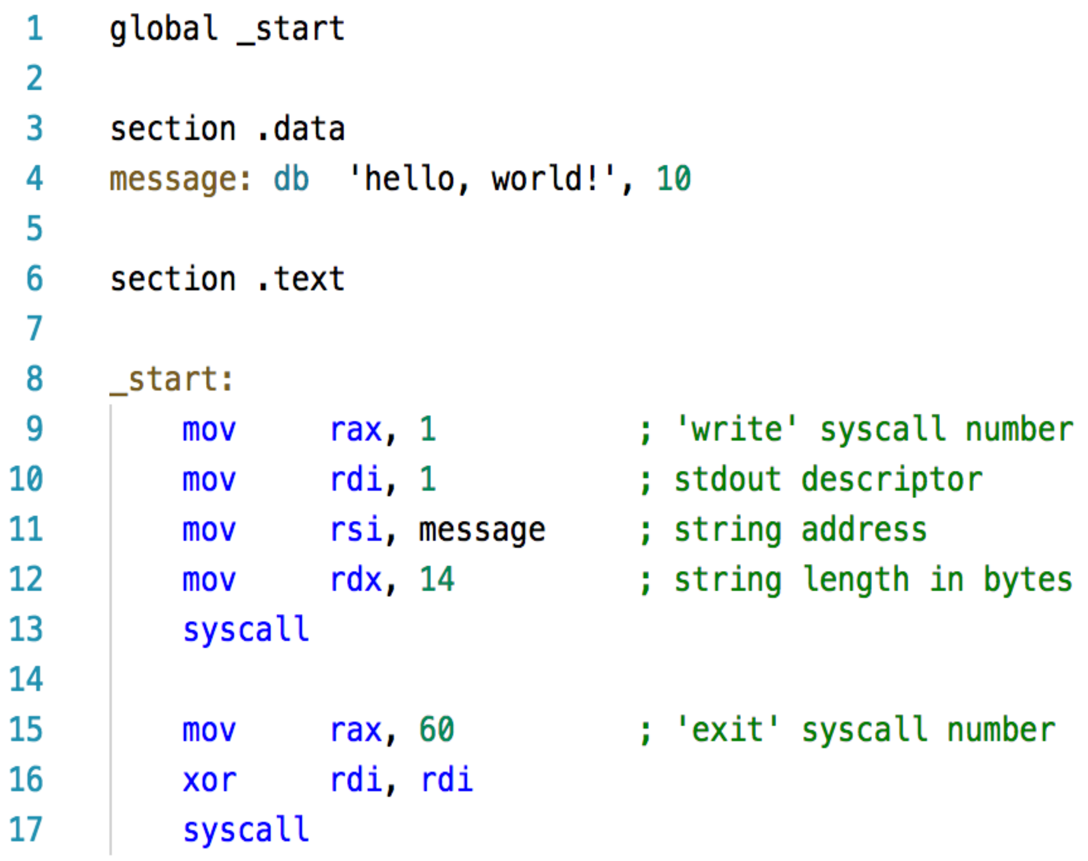
Assembly Programming

Presented by: Han Trung Dinh

Assembly Compilers & Editors

- Assembly Compilers :
  - ✓NASM: The Netwide Assembler, open source
  - ✓YASM: a full rewrite of NASM
  - ✓MASM: Microsoft Macro Assembler, included with MS Visual Studio
  - ✓TASM: Turbo Assembler, developed by Borland
- Editors: (any editor could be fine)
  - Windows: Notepad, Visual Studio, Visual Studio Code, Atom, Eclipse, WinASM
  - Linux: vi/vim, Visual Studio Code, Atom, Eclipse
  - MacOS: vi/vim, Visual Studio Code, Xcode, Atom, Eclipse

NASM Program Structure



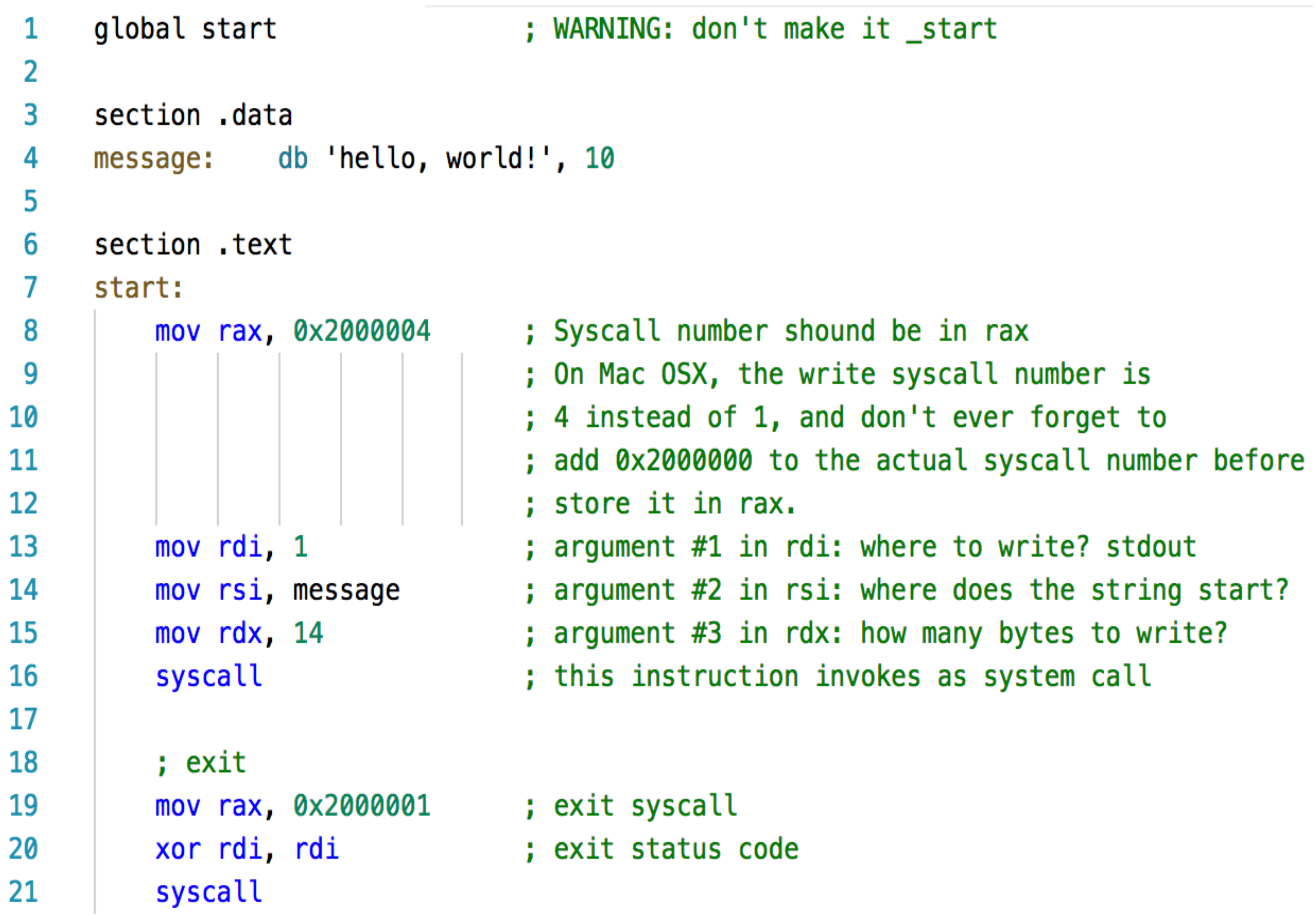
```nasm
global _start

section .data
message: db  'hello, world!', 10

section .text

_start:
    mov     rax, 1              ; 'write' syscall number
    mov     rdi, 1              ; stdout descriptor
    mov     rsi, message        ; string address
    mov     rdx, 14             ; string length in bytes
    syscall

    mov     rax, 60             ; 'exit' syscall number
    xor     rdi, rdi
    syscall
```



```nasm
global start                    ; WARNING: don't make it _start

section .data
message:    db 'hello, world!', 10

section .text
start:
    mov rax, 0x2000004          ; Syscall number shound be in rax
                                ; On Mac OSX, the write syscall number is
                                ; 4 instead of 1, and don't ever forget to
                                ; add 0x2000000 to the actual syscall number before
                                ; store it in rax.
    mov rdi, 1                  ; argument #1 in rdi: where to write? stdout
    mov rsi, message            ; argument #2 in rsi: where does the string start?
    mov rdx, 14                 ; argument #3 in rdx: how many bytes to write?
    syscall                     ; this instruction invokes as system call

    ; exit
    mov rax, 0x2000001          ; exit syscall
    xor rdi, rdi                ; exit status code
    syscall
```

Linux:

```
> nasm -felf64 hello.asm -o hello.o
> ld -o hello hello.o
```

```
> chmod u+x hello
> ./hello
```

MacOS:

```
> nasm -f macho64 hello.asm
> ld -macosx_version_min 10.7.0 -lSystem -o hello hello.o
```

General Purpose Registers (GPRs)

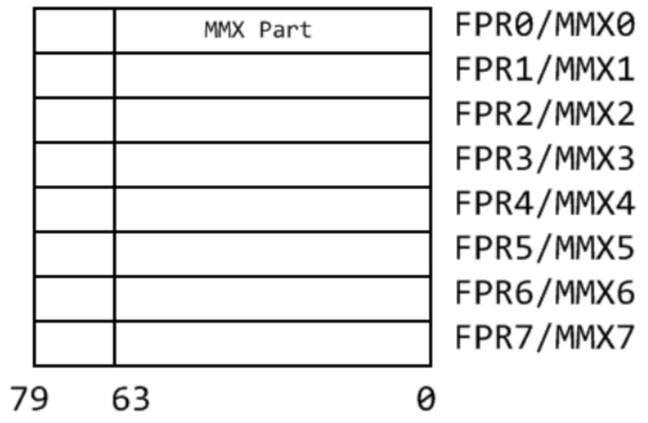
64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

eip -> rip
flags -> rflags

Floating-point registers:

- Eight 80-bit x87 registers.
- Eight 64-bit MMX registers. (These overlap with the x87 registers)
- The original set of eight 128-bit SSE (Streaming SIMD Extensions) registers is increased to sixteen

80-bit floating point
and 64-bit MMX registers
(overlaid)



Conventional Uses for General-Purpose Registers

Register	Conventional Use
EAX	Accumulator
EBX	Memory pointer, base register
ECX	Loop control, counter
EDX	Integer multiplication, integer division
ESI	String instruction source pointer, index register
EDI	String instruction destination pointer, index register
ESP	Stack pointer
EBP	Stack frame base pointer

Common Flags (32 bits register)

Symbol	Bit	Name	Set if....
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has even number of 1‟s, else 0
AF	4	Adjust	Denotes Binary Coded Decimal in-byte carry
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of result is 1
OF	11	Overflow	Overflow on signed operation
DF	10	Direction	Direction string instructions operate (increment or decrement)
ID	21	Identification	Changeability denotes presence of CPUID instruction

Detail EFlags

Bit	Name	Symbol	Use
0	Carry Flag	CF	Status
1	Reserved		1
2	Parity Flag	PF	Status
3	Reserved		0
4	Auxiliary Carry Flag	AF	Status
5	Reserved		0
6	Zero Flag	ZF	Status
7	Sign Flag	SF	Status
8	Trap Flag	TF	System
9	Interrupt Enable Flag	IF	System
10	Direction Flag	DF	Control

Detail EFlags

Bit	Name	Symbol	Use
11	Overflow Flag	OF	Status
12	I/O Privilege Level Bit 0	IOPL	System
13	I/O Privilege Level Bit 1	IOPL	System
14	Nested Task	NT	System
15	Reserved		0
16	Resume Flag	RF	System
17	Virtual 8086 Mode	VM	System
18	Alignment Check	AC	System
19	Virtual Interrupt Flag	VIF	System
20	Virtual Interrupt Pending	VIP	System
21	ID Flag	ID	System
22 - 31	Reserved		0

Floating Point Types

Data type	Length	Precision (bits)	Decimal digits Precision	Decimal Range
Single Precision	32	24	7	1.18*10^-38 to 3.40*10^38
Double Precision	64	53	15	2.23 *10^-308 to 1.79*10^308
Extended Precision	80	64	19	3.37*10^-4932 to 1.18*10^4932

Command sets of MMX and SSE technology

Technology	Register size/type	Item type	Items in Parallel
MMX	64 MMX	Integer	8, 4, 2, 1
SSE	64 MMX	Integer	8,4,2,1
SSE	128 XMM	Float	4
SSE2/SSE3/SSSE3...	64 MMX	Integer	2,1
SSE2/SSE3/SSSE3...	128 XMM	Float	2
SSE2/SSE3/SSSE3...	128 XMM	Integer	16,8,4,2,1

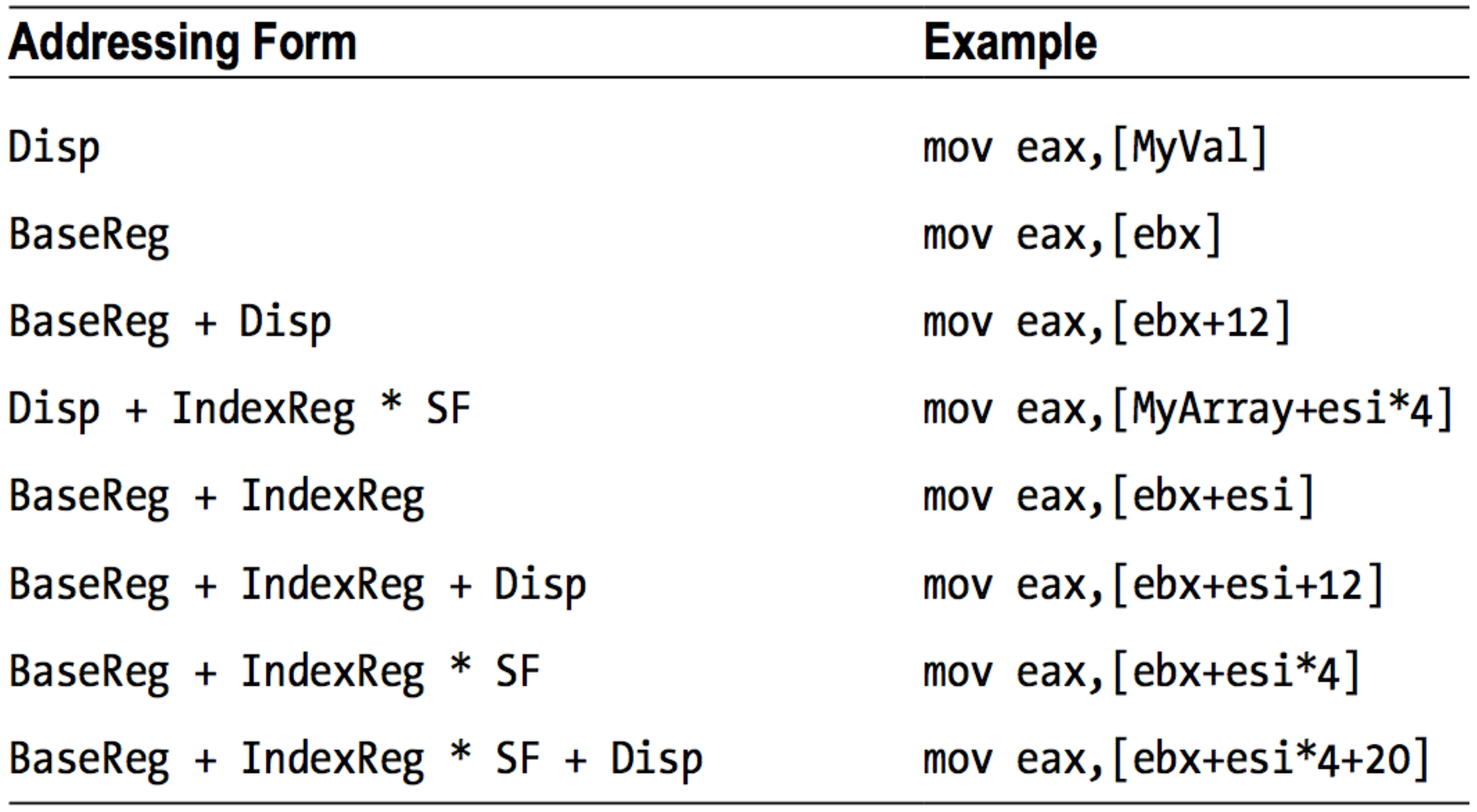
Common Instruction Sets

Opcode	Meaning	Opcode	Meaning
MOV	Move to/from/between memory and registers	AND/OR/XOR/NOT	Bitwise operations
CMOV*	Various conditional moves	SHR/SAR	Shift right logical/arithmetic
XCHG	Exchange	SHL/SAL	Shift left logical/arithmetic
BSWAP	Byte swap	ROR/ROL	Rotate right/left
PUSH/POP	Stack usage	RCR/RCL	Rotate right/left through carry bit
ADD/ADC	Add/with carry	BT/BTS/BTR	Bit test/and set/and reset
SUB/SBC	Subtract/with carry	JMP	Unconditional jump
MUL/IMUL	Multiply/unsigned	JE/JNE/JC/JNC/J*	Jump if equal/not equal/carry/not carry/ many others
DIV/IDIV	Divide/unsigned	LOOP/LOOPE/LOOPNE	Loop with ECX
INC/DEC	Increment/Decrement	CALL/RET	Call subroutine/return
NEG	Negate	NOP	No operation
CMP	Compare	CPUID	CPU information

Examples of Instruction Operands

Type	Example	Equivalent C/C++ Statement
Immediate	`mov eax,42`	`eax = 42`
	`imul ebx,11h`	`ebx *= 0x11`
	`xor dl,55h`	`dl ^= 0x55`
	`add esi,8`	`esi += 8`
Register	`mov eax,ebx`	`eax = ebx`
	`inc ecx`	`ecx += 1`
	`add ebx,esi`	`ebx += esi`
	`mul ebx`	`edx:eax = eax * ebx`
Memory	`mov eax,[ebx]`	`eax = *ebx`
	`add eax,[val1]`	`eax += *val1`
	`or ecx,[ebx+esi]`	`ecx |= *(ebx + esi)`
	`sub word ptr [edi],12`	`*(short*)edi -= 12`

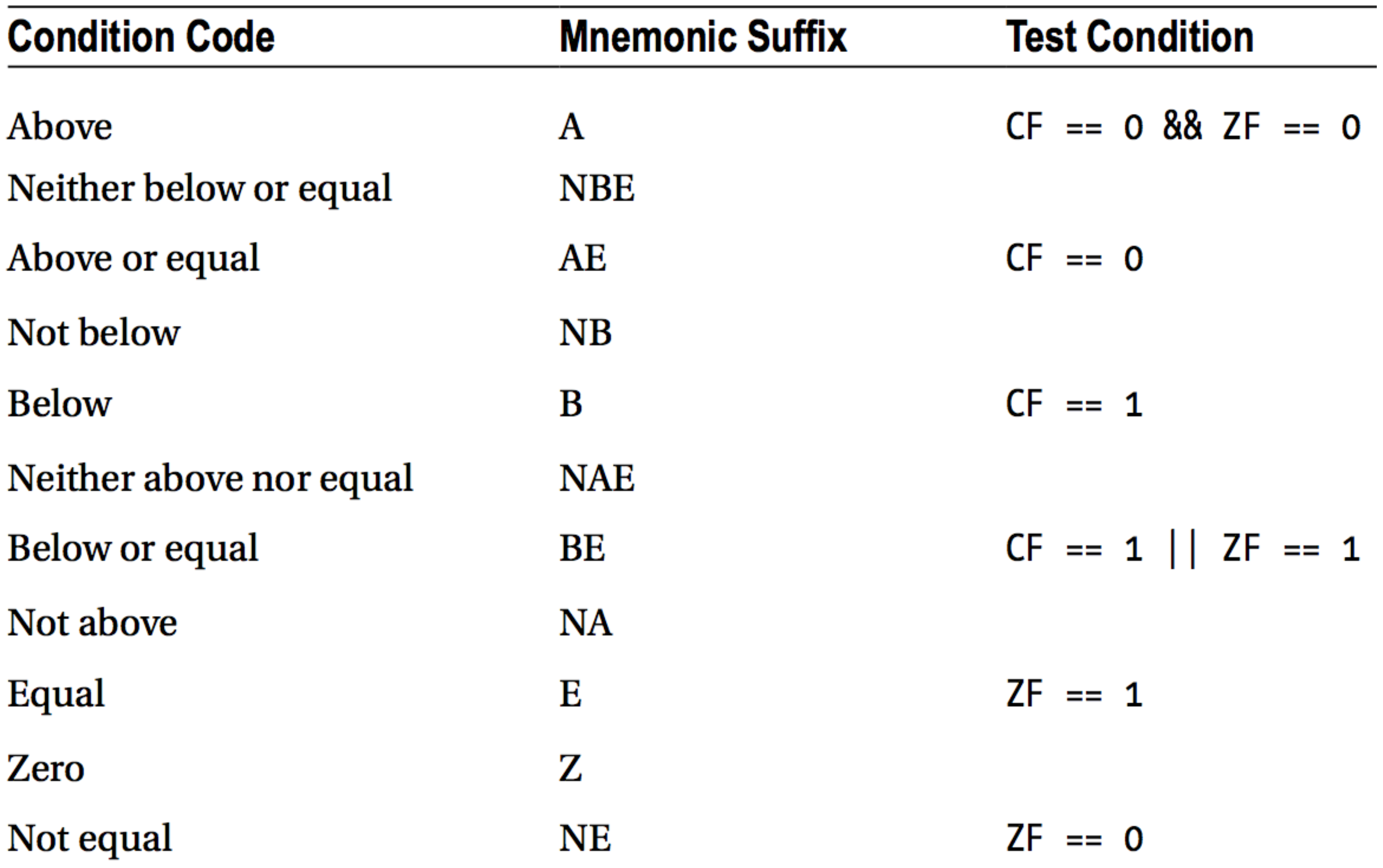
Memory Operand Addressing Forms



Addressing Form	Example
`Disp`	`mov eax,[MyVal]`
`BaseReg`	`mov eax,[ebx]`
`BaseReg + Disp`	`mov eax,[ebx+12]`
`Disp + IndexReg * SF`	`mov eax,[MyArray+esi*4]`
`BaseReg + IndexReg`	`mov eax,[ebx+esi]`
`BaseReg + IndexReg + Disp`	`mov eax,[ebx+esi+12]`
`BaseReg + IndexReg * SF`	`mov eax,[ebx+esi*4]`
`BaseReg + IndexReg * SF + Disp`	`mov eax,[ebx+esi*4+20]`

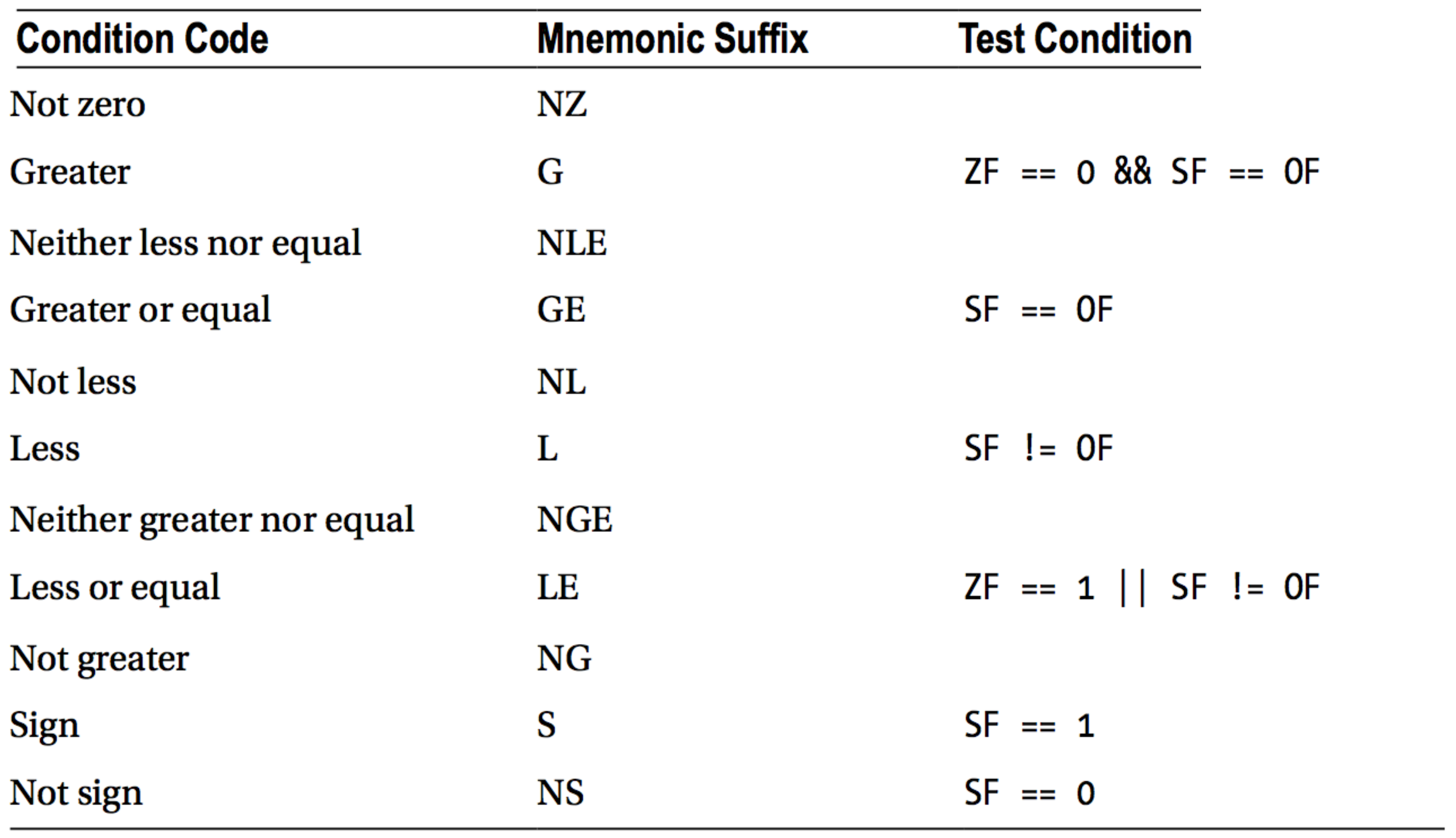
Effective Address = BaseReg + IndexReg * ScaleFactor + Disp
Note that *IndexReg* must not be ESP

Condition Codes, Mnemonic Suffixes, and Test Conditions



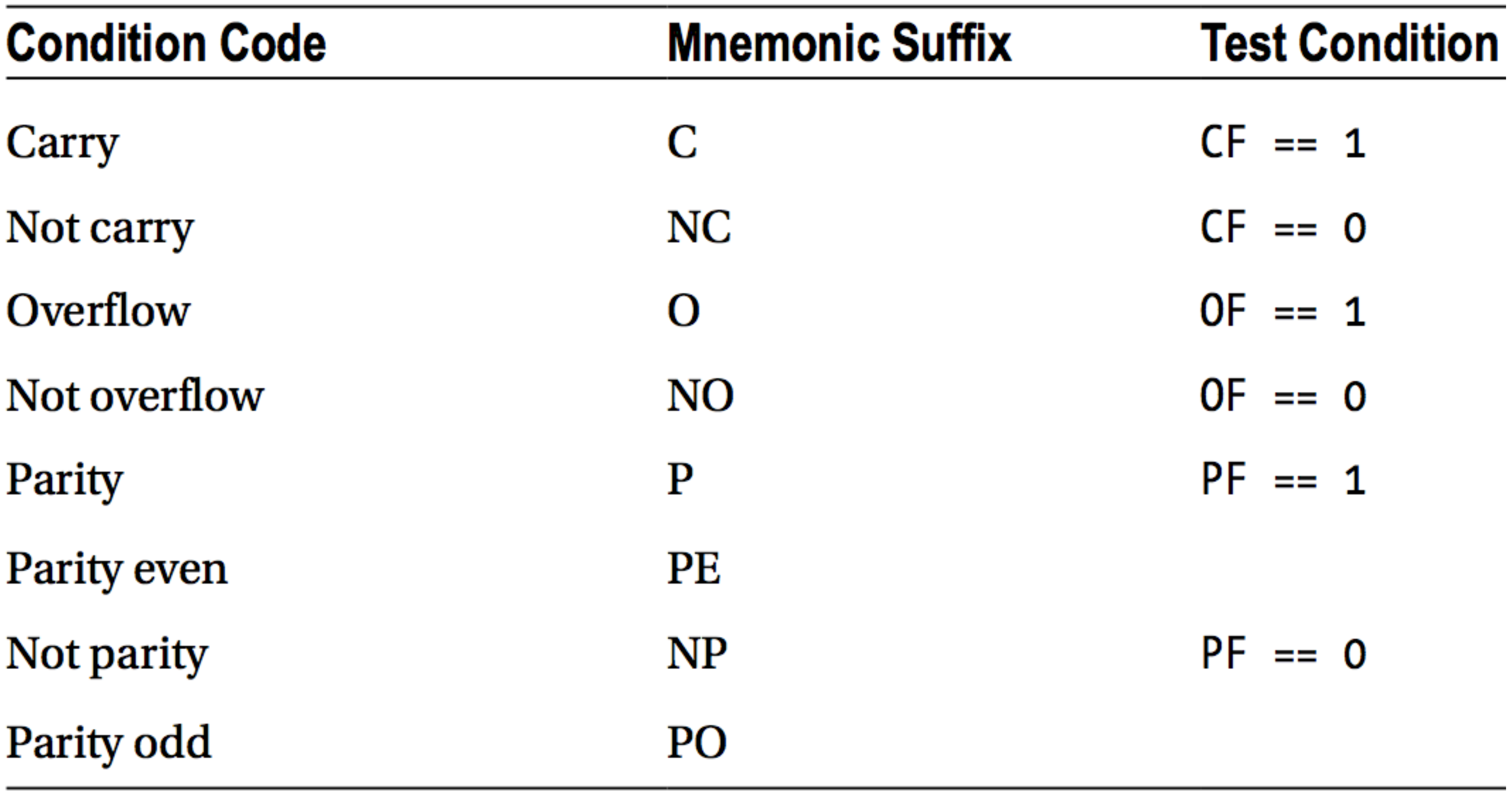
Condition Code	Mnemonic Suffix	Test Condition
Above	A	CF == 0 && ZF == 0
Neither below or equal	NBE	
Above or equal	AE	CF == 0
Not below	NB	
Below	B	CF == 1
Neither above nor equal	NAE	
Below or equal	BE	CF == 1 || ZF == 1
Not above	NA	
Equal	E	ZF == 1
Zero	Z	
Not equal	NE	ZF == 0

Condition Codes, Mnemonic Suffixes, and Test Conditions



Condition Code	Mnemonic Suffix	Test Condition
Not zero	NZ	
Greater	G	`ZF == 0 && SF == OF`
Neither less nor equal	NLE	
Greater or equal	GE	`SF == OF`
Not less	NL	
Less	L	`SF != OF`
Neither greater nor equal	NGE	
Less or equal	LE	`ZF == 1 || SF != OF`
Not greater	NG	
Sign	S	`SF == 1`
Not sign	NS	`SF == 0`

Condition Codes, Mnemonic Suffixes, and Test Conditions

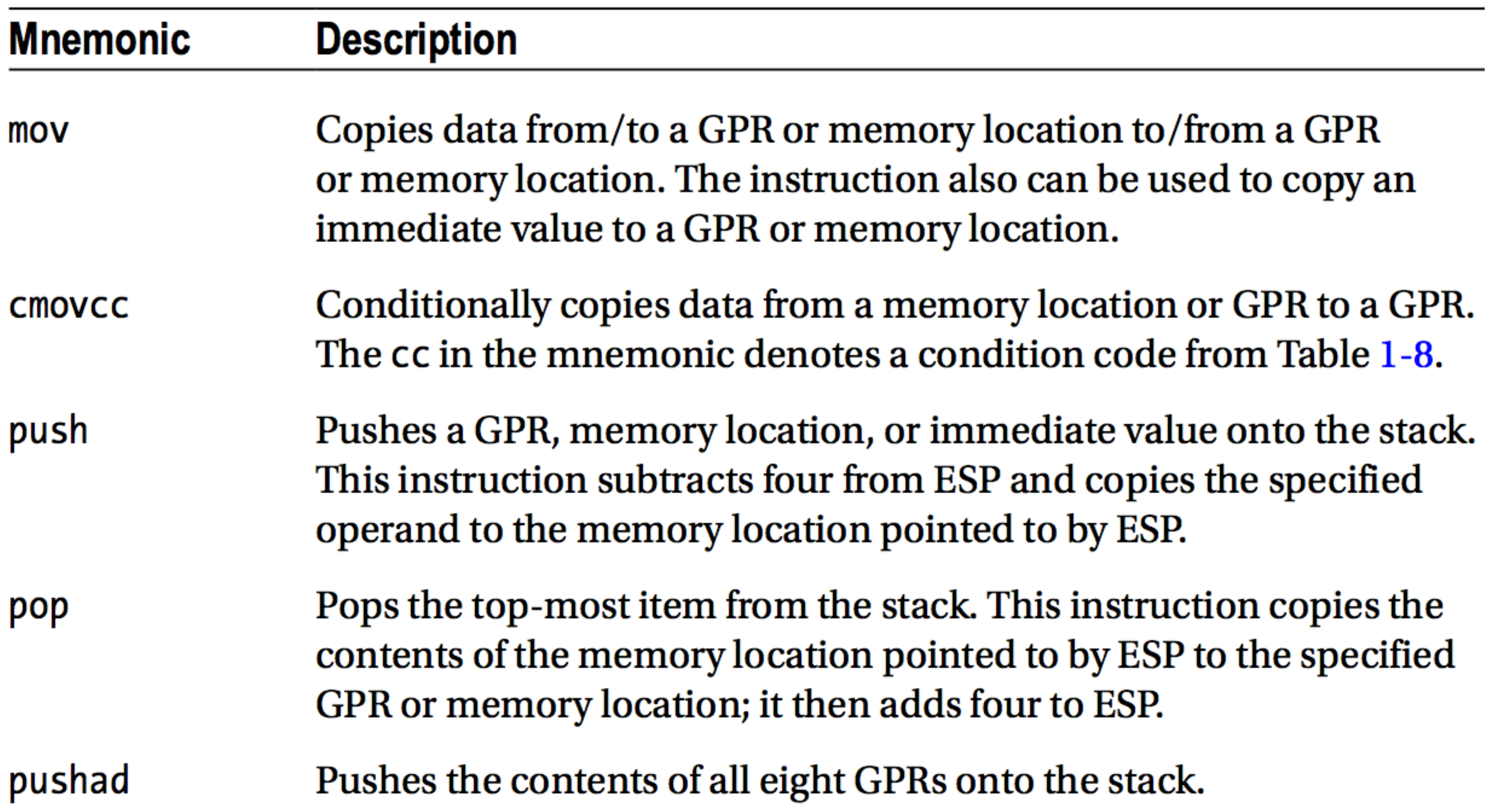


Condition Code	Mnemonic Suffix	Test Condition
Carry	C	CF == 1
Not carry	NC	CF == 0
Overflow	O	OF == 1
Not overflow	NO	OF == 0
Parity	P	PF == 1
Parity even	PE	
Not parity	NP	PF == 0
Parity odd	PO	

Instruction Sets Functional Categories

1. Data transfer
2. Data comparison
3. Data conversion
4. Binary arithmetic
5. Logical
6. Rotate and shift
7. Byte set and bit strings
8. String
9. Flags
10. Control transfer
11. Miscellaneous

Data-Transfer Instructions



Mnemonic	Description
`mov`	Copies data from/to a GPR or memory location to/from a GPR or memory location. The instruction also can be used to copy an immediate value to a GPR or memory location.
`cmovcc`	Conditionally copies data from a memory location or GPR to a GPR. The `cc` in the mnemonic denotes a condition code from Table 1-8.
`push`	Pushes a GPR, memory location, or immediate value onto the stack. This instruction subtracts four from ESP and copies the specified operand to the memory location pointed to by ESP.
`pop`	Pops the top-most item from the stack. This instruction copies the contents of the memory location pointed to by ESP to the specified GPR or memory location; it then adds four to ESP.
`pushad`	Pushes the contents of all eight GPRs onto the stack.

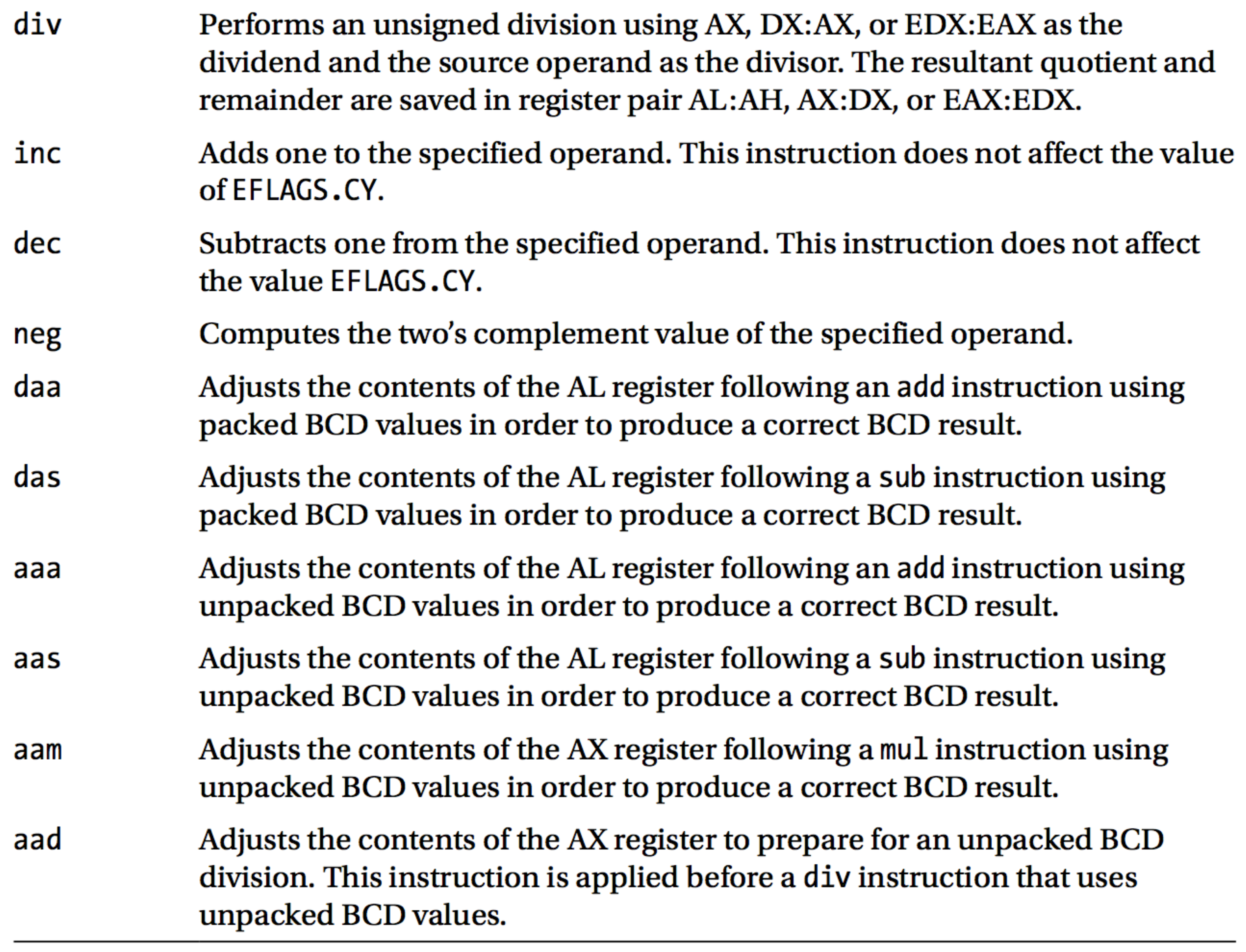
Data-Transfer Instructions (cont.)

Mnemonic	Description
popad	Pops the stack to restore the contents of all GPRs. The stack value for ESP is ignored.
xchg	Exchanges data between two GPRs or a GPR and a memory location. The processor uses a locked bus cycle if the register-memory form of the instruction is used.
xadd	Exchanges data between two GPRs or a GPR and a memory location. The sum of the two operands is then saved to the destination operand.
movsx	Sign-extends the contents of a GPR or memory location and copies the result value to a GPR.
movzx	Zero-extends the contents of a GPR or memory location and copies the result to a GPR.

Binary Arithmetic Instructions

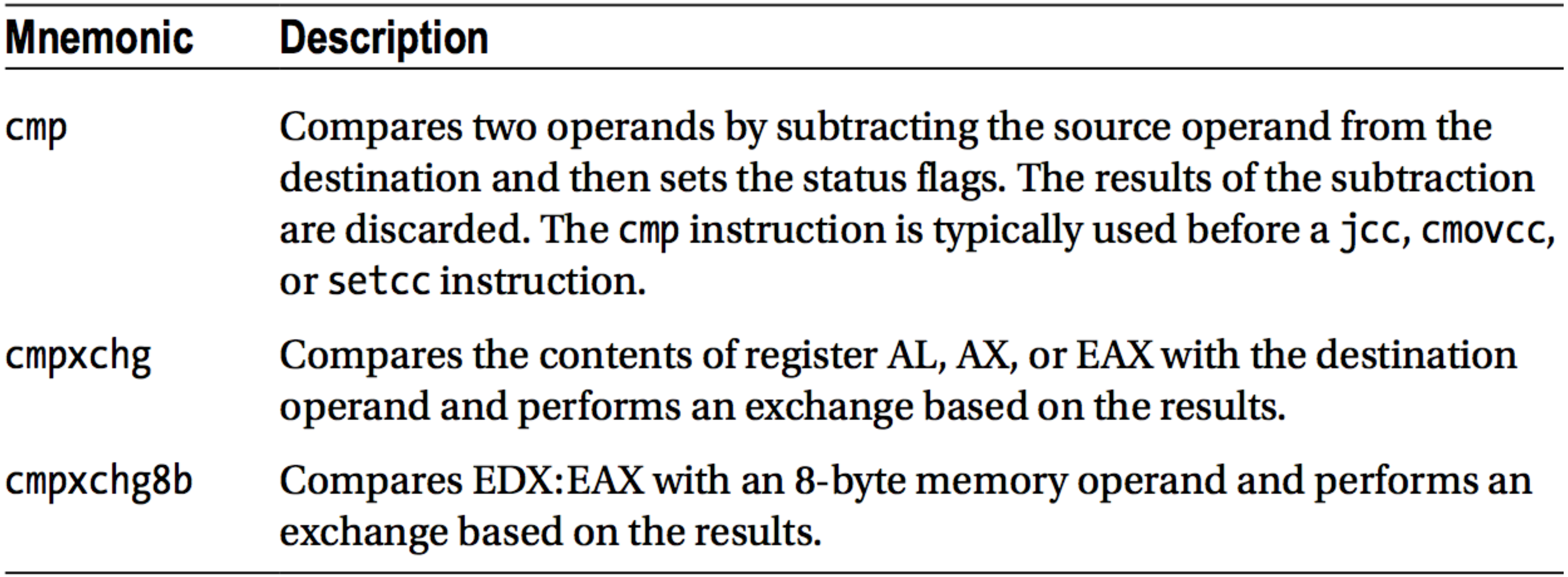
Mnemonic	Description
`add`	Adds the source operand and destination operand. This instruction can be used for both signed and unsigned integers.
`adc`	Adds the source operand, destination operand, and the state of `EFLAGS.CY`. This instruction can be used for both signed and unsigned integers.
`sub`	Subtracts the source operand from the destination operand. This instruction can be used for both signed and unsigned integers.
`sbb`	Subtracts the sum of the source operand and `EFLAGS.CY` from the destination operand. This instruction can be used for both signed and unsigned integers.
`imul`	Performs a signed multiply between two operands. This instruction supports multiple forms, including a single source operand (with AL, AX, or EAX as an implicit operand), an explicit source and destination operand, and a three-operand variant (immediate source, memory/register source, and GPR destination).
`mul`	Performs an unsigned multiply between the source operand and the AL, AX, or EAX register. The results are saved in the AX, DX:AX, or EDX:EAX registers.
`idiv`	Performs a signed division using AX, DX:AX, or EDX:EAX as the dividend and the source operand as the divisor. The resultant quotient and remainder are saved in register pair AL:AH, AX:DX, or EAX:EDX.

Binary Arithmetic Instructions



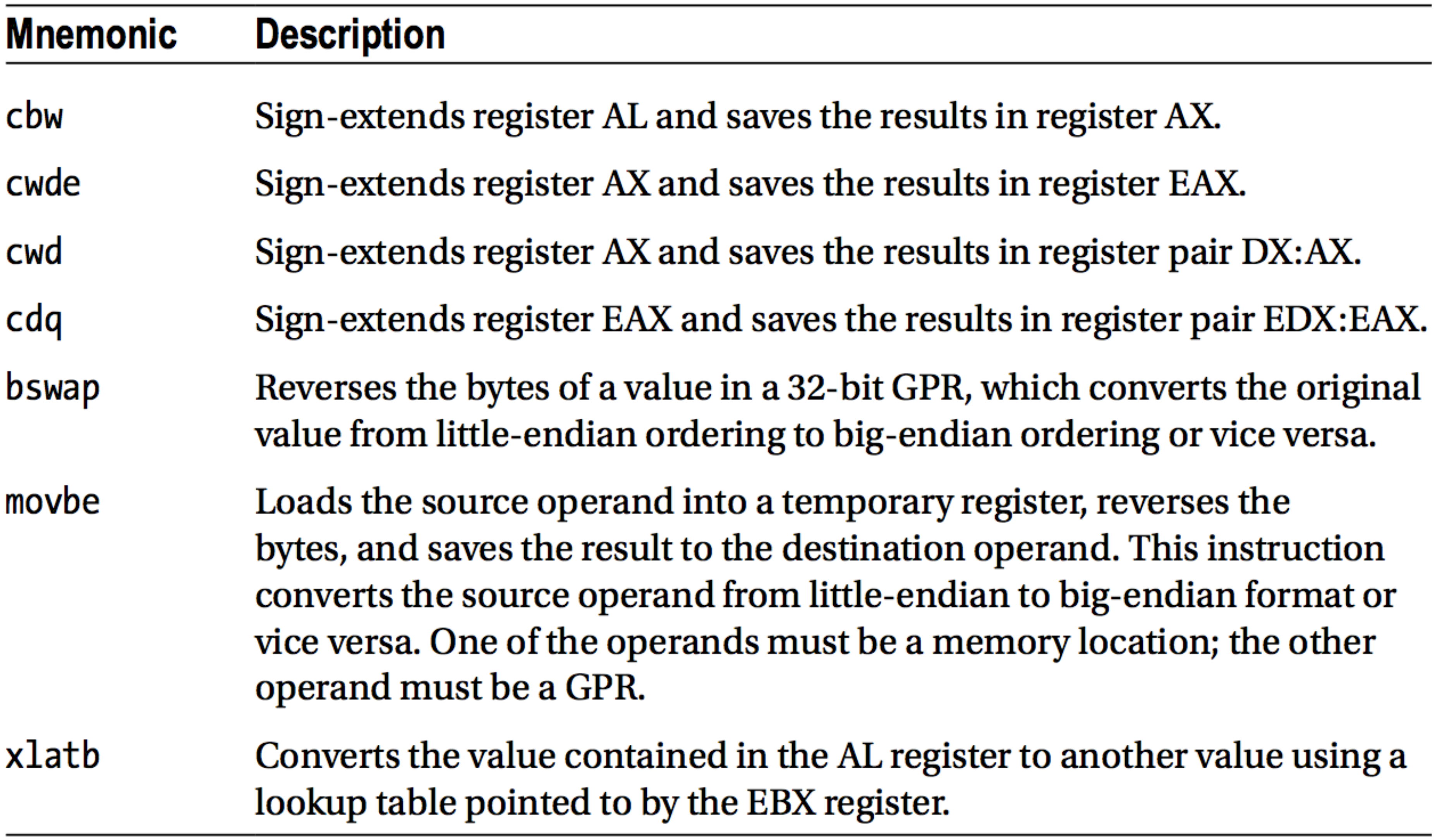
`div`	Performs an unsigned division using AX, DX:AX, or EDX:EAX as the dividend and the source operand as the divisor. The resultant quotient and remainder are saved in register pair AL:AH, AX:DX, or EAX:EDX.
`inc`	Adds one to the specified operand. This instruction does not affect the value of `EFLAGS.CY`.
`dec`	Subtracts one from the specified operand. This instruction does not affect the value `EFLAGS.CY`.
`neg`	Computes the two's complement value of the specified operand.
`daa`	Adjusts the contents of the AL register following an `add` instruction using packed BCD values in order to produce a correct BCD result.
`das`	Adjusts the contents of the AL register following a `sub` instruction using packed BCD values in order to produce a correct BCD result.
`aaa`	Adjusts the contents of the AL register following an `add` instruction using unpacked BCD values in order to produce a correct BCD result.
`aas`	Adjusts the contents of the AL register following a `sub` instruction using unpacked BCD values in order to produce a correct BCD result.
`aam`	Adjusts the contents of the AX register following a `mul` instruction using unpacked BCD values in order to produce a correct BCD result.
`aad`	Adjusts the contents of the AX register to prepare for an unpacked BCD division. This instruction is applied before a `div` instruction that uses unpacked BCD values.

Data Comparison



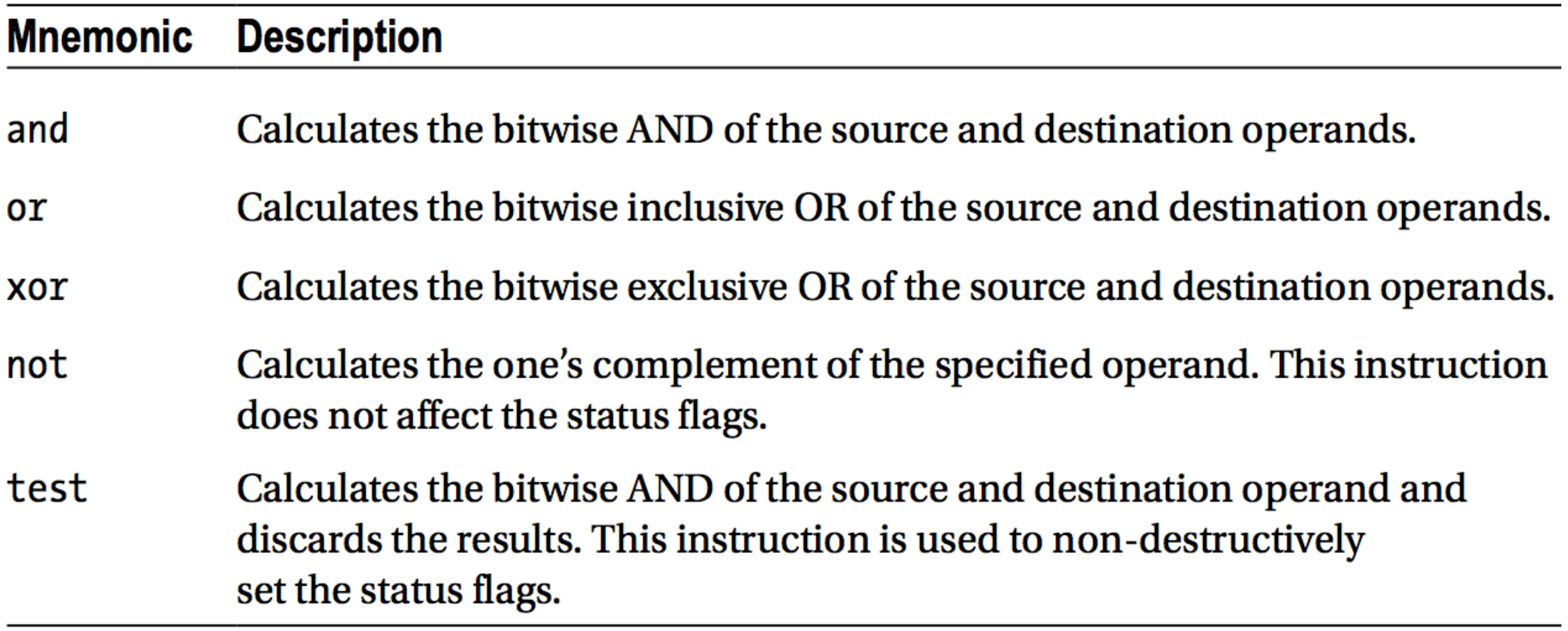
Mnemonic	Description
`cmp`	Compares two operands by subtracting the source operand from the destination and then sets the status flags. The results of the subtraction are discarded. The `cmp` instruction is typically used before a `jcc`, `cmovcc`, or `setcc` instruction.
`cmpxchg`	Compares the contents of register AL, AX, or EAX with the destination operand and performs an exchange based on the results.
`cmpxchg8b`	Compares EDX:EAX with an 8-byte memory operand and performs an exchange based on the results.

Data Conversion



Mnemonic	Description
`cbw`	Sign-extends register AL and saves the results in register AX.
`cwde`	Sign-extends register AX and saves the results in register EAX.
`cwd`	Sign-extends register AX and saves the results in register pair DX:AX.
`cdq`	Sign-extends register EAX and saves the results in register pair EDX:EAX.
`bswap`	Reverses the bytes of a value in a 32-bit GPR, which converts the original value from little-endian ordering to big-endian ordering or vice versa.
`movbe`	Loads the source operand into a temporary register, reverses the bytes, and saves the result to the destination operand. This instruction converts the source operand from little-endian to big-endian format or vice versa. One of the operands must be a memory location; the other operand must be a GPR.
`xlatb`	Converts the value contained in the AL register to another value using a lookup table pointed to by the EBX register.

Logical

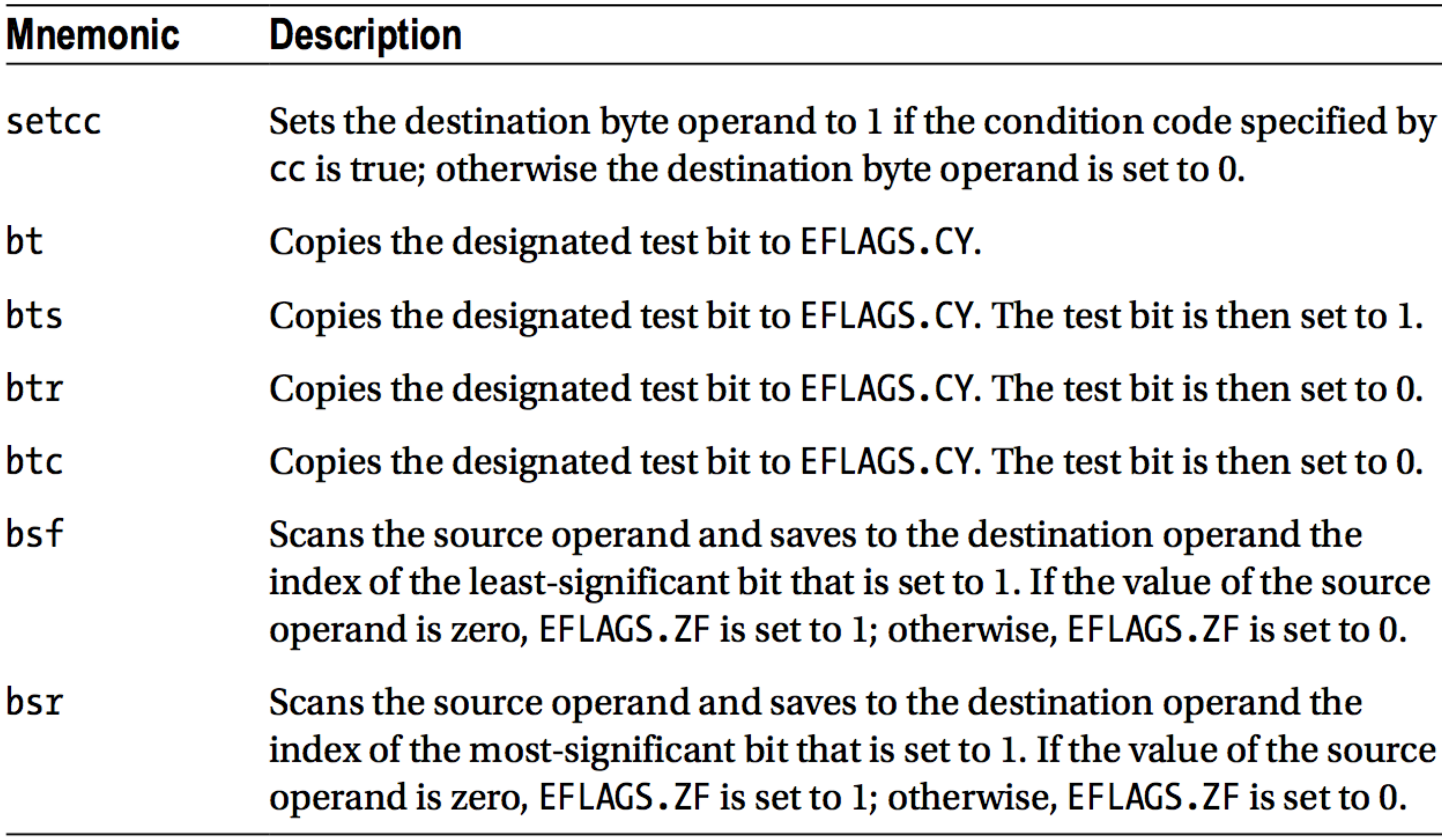


Mnemonic	Description
`and`	Calculates the bitwise AND of the source and destination operands.
`or`	Calculates the bitwise inclusive OR of the source and destination operands.
`xor`	Calculates the bitwise exclusive OR of the source and destination operands.
`not`	Calculates the one's complement of the specified operand. This instruction does not affect the status flags.
`test`	Calculates the bitwise AND of the source and destination operand and discards the results. This instruction is used to non-destructively set the status flags.

Rotate and Shift

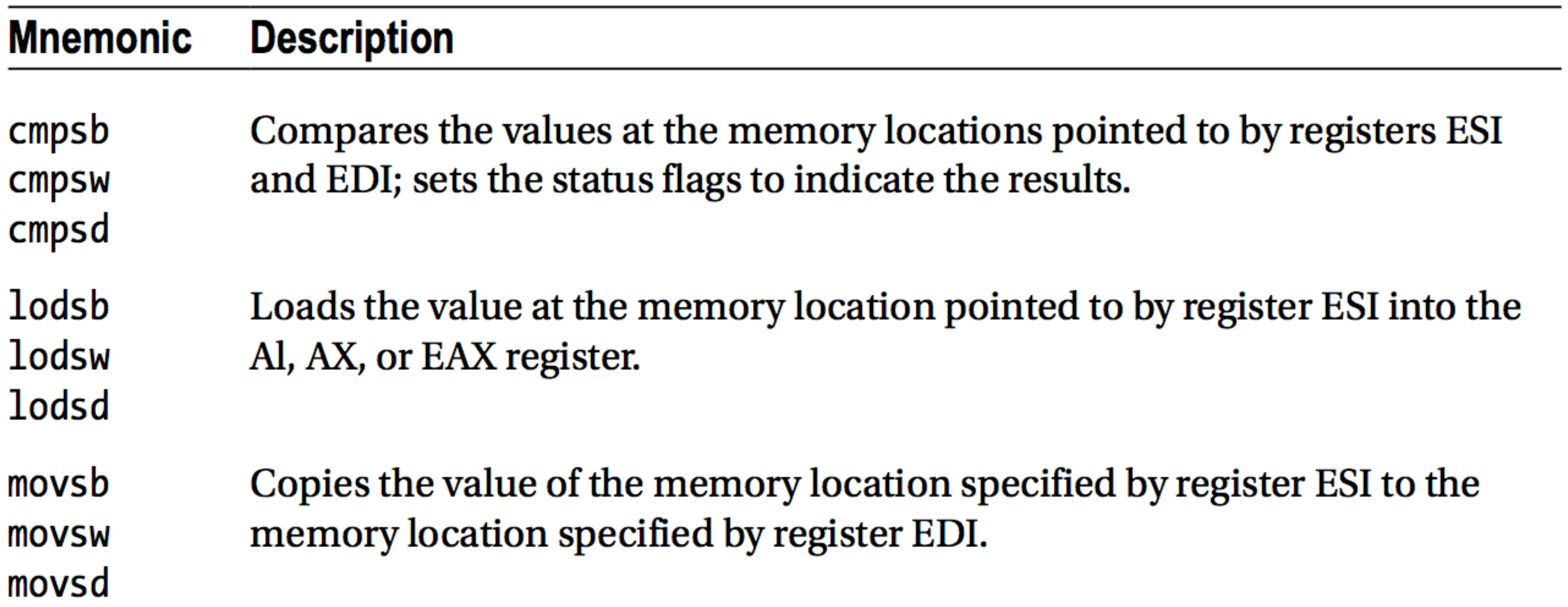
Mnemonic	Description
`rcl`	Rotates the specified operand to the left. `EFLAGS.CY` flag is included as part of the rotation.
`rcr`	Rotates the specified operand to the right. `EFLAGS.CY` flag is included as part of the rotation.
`rol`	Rotates the specified operand to the left.
`ror`	Rotates the specified operand to the right.
`sal/shl`	Performs an arithmetic left shift of the specified operand.
`sar`	Performs an arithmetic right shift of the specified operand.
`shr`	Performs a logical right shift of the specified operand.
`shld`	Performs a double-precision logical left shift using two operands.
`shrd`	Performs a double-precision logical right shift using two operands.

Byte Set and Bit String Instructions



Mnemonic	Description
`setcc`	Sets the destination byte operand to 1 if the condition code specified by cc is true; otherwise the destination byte operand is set to 0.
`bt`	Copies the designated test bit to `EFLAGS.CY`.
`bts`	Copies the designated test bit to `EFLAGS.CY`. The test bit is then set to 1.
`btr`	Copies the designated test bit to `EFLAGS.CY`. The test bit is then set to 0.
`btc`	Copies the designated test bit to `EFLAGS.CY`. The test bit is then set to 0.
`bsf`	Scans the source operand and saves to the destination operand the index of the least-significant bit that is set to 1. If the value of the source operand is zero, `EFLAGS.ZF` is set to 1; otherwise, `EFLAGS.ZF` is set to 0.
`bsr`	Scans the source operand and saves to the destination operand the index of the most-significant bit that is set to 1. If the value of the source operand is zero, `EFLAGS.ZF` is set to 1; otherwise, `EFLAGS.ZF` is set to 0.

String Instructions



Mnemonic	Description
cmpsb cmpsw cmpsd	Compares the values at the memory locations pointed to by registers ESI and EDI; sets the status flags to indicate the results.
lodsb lodsw lodsd	Loads the value at the memory location pointed to by register ESI into the Al, AX, or EAX register.
movsb movsw movsd	Copies the value of the memory location specified by register ESI to the memory location specified by register EDI.

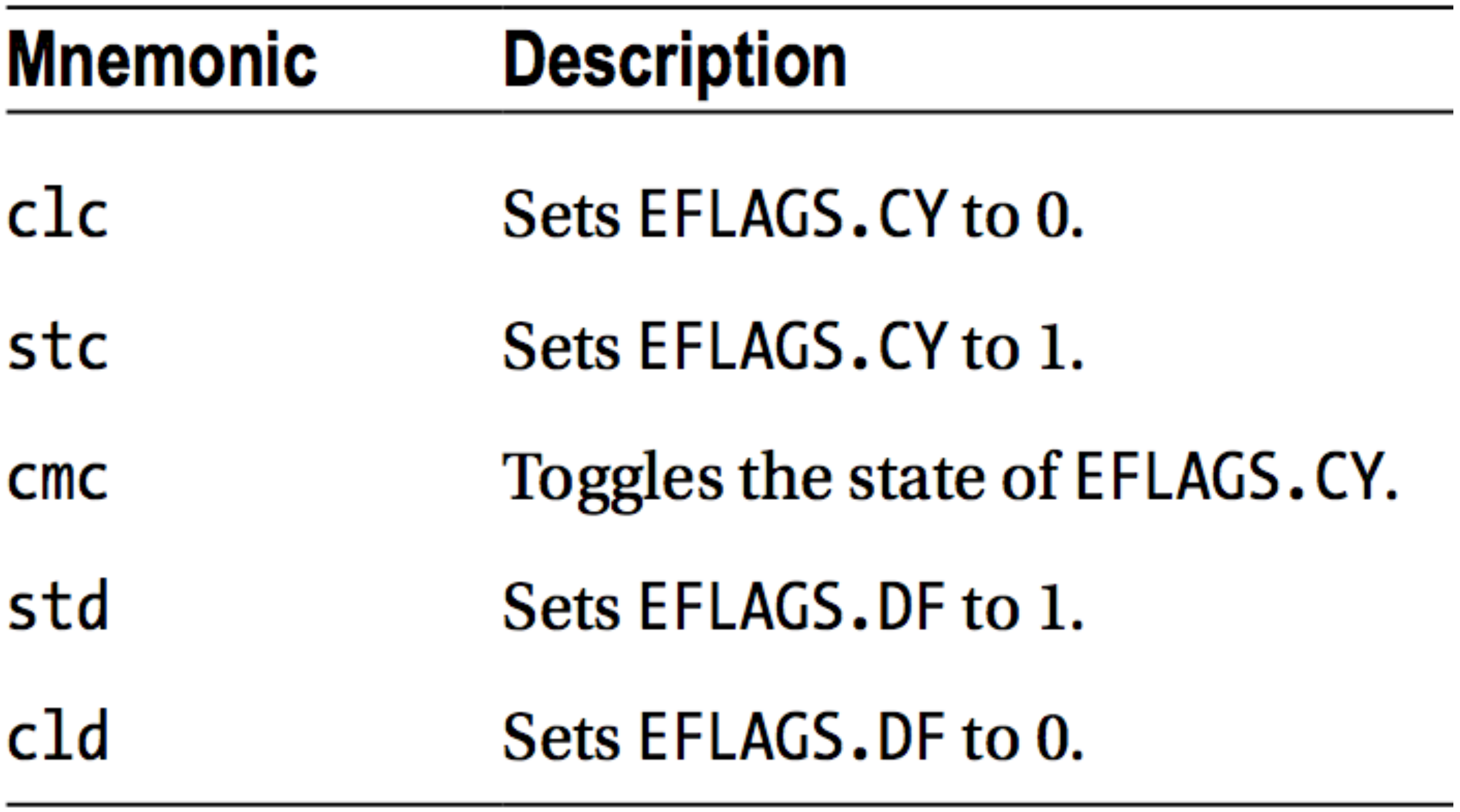
String Instructions (cont.)

`scasb`
`scasw`
`scasd`
	Compares the value of the memory location specified by register EDI with the value contained in register AL, AX, or EAX; sets the status flags based on the comparison results.

`stosb`
`stosw`
`stosd`
	Stores the contents of register AL, AX, or EAX to the memory location specified by register EDI.

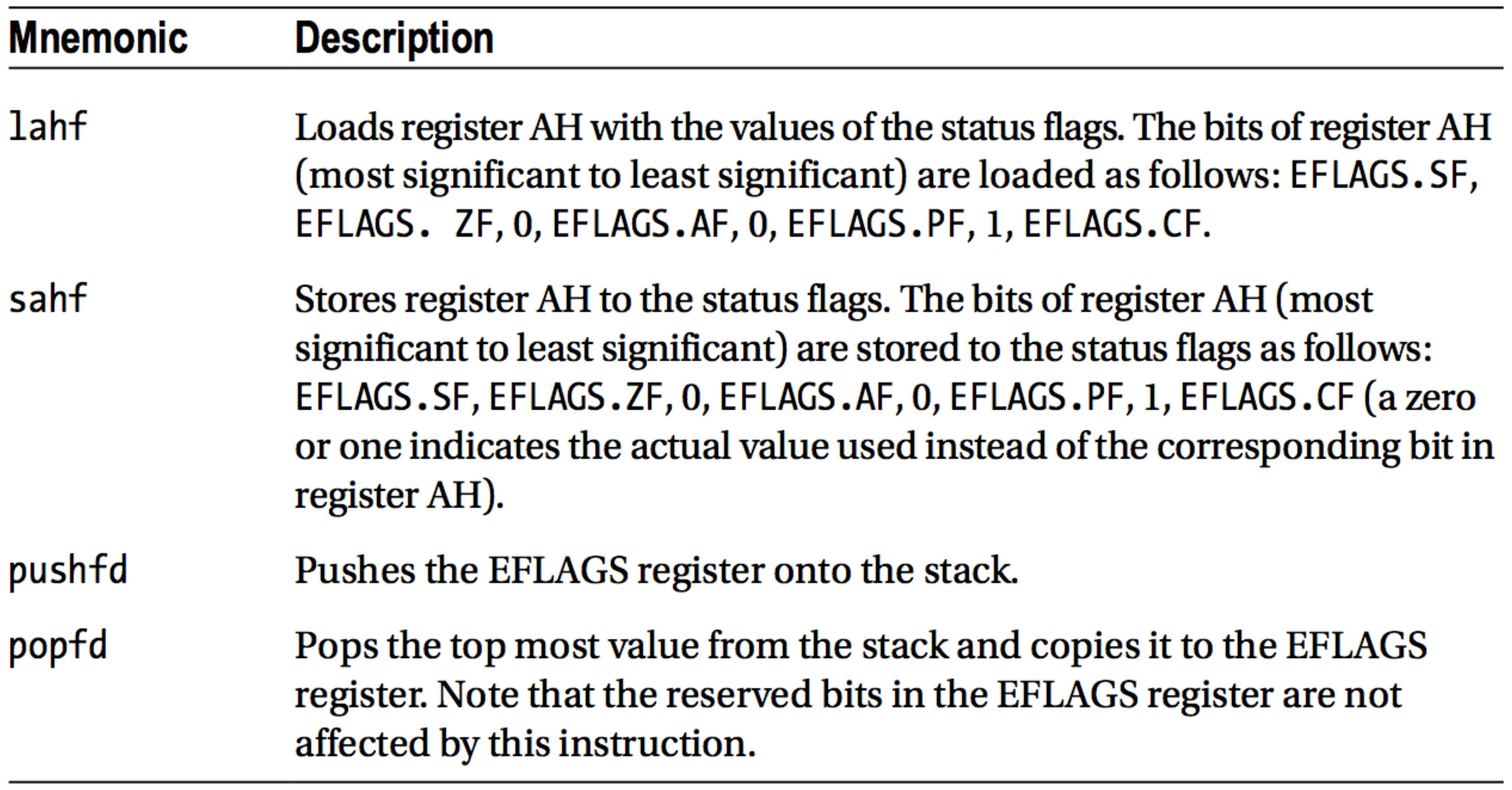
`rep`	Repeats the specified string instruction while the condition `ECX != 0` is true.
`repe` `repz`	Repeats the specified string instruction while the condition `ECX != 0 && ZF == 1` is true.
`repne` `repnz`	Repeats the specified string instruction while the condition `ECX != 0 && ZF == 0` is true.

Flag Manipulation



Mnemonic	Description
`clc`	Sets `EFLAGS.CY` to 0.
`stc`	Sets `EFLAGS.CY` to 1.
`cmc`	Toggles the state of `EFLAGS.CY`.
`std`	Sets `EFLAGS.DF` to 1.
`cld`	Sets `EFLAGS.DF` to 0.

Flag Manipulation (cont.)



Mnemonic	Description
`lahf`	Loads register AH with the values of the status flags. The bits of register AH (most significant to least significant) are loaded as follows: `EFLAGS.SF`, `EFLAGS. ZF`, 0, `EFLAGS.AF`, 0, `EFLAGS.PF`, 1, `EFLAGS.CF`.
`sahf`	Stores register AH to the status flags. The bits of register AH (most significant to least significant) are stored to the status flags as follows: `EFLAGS.SF`, `EFLAGS.ZF`, 0, `EFLAGS.AF`, 0, `EFLAGS.PF`, 1, `EFLAGS.CF` (a zero or one indicates the actual value used instead of the corresponding bit in register AH).
`pushfd`	Pushes the EFLAGS register onto the stack.
`popfd`	Pops the top most value from the stack and copies it to the EFLAGS register. Note that the reserved bits in the EFLAGS register are not affected by this instruction.

Control-Transfer Instructions

Mnemonic	Description
`jmp`	Performs an unconditional jump to the memory location specified by the operand.
`jcc`	Performs a conditional jump to the memory location specified by the operand if the identified condition is true. The `cc` denotes a condition-code mnemonic fromTable 1-8.
`call`	Pushes the contents of register EIP onto the stack and then performs an unconditional jump to the memory location that is specified by the operand.
`ret`	Pops the target address off the stack and then performs an unconditional jump to that address.
`enter`	Creates a stack frame that enables to a function's parameters and local data by initializing the EBP and ESP registers.
`leave`	Removes the stack frame that was created using an `enter` instruction by restoring the caller's EBP and ESP registers.

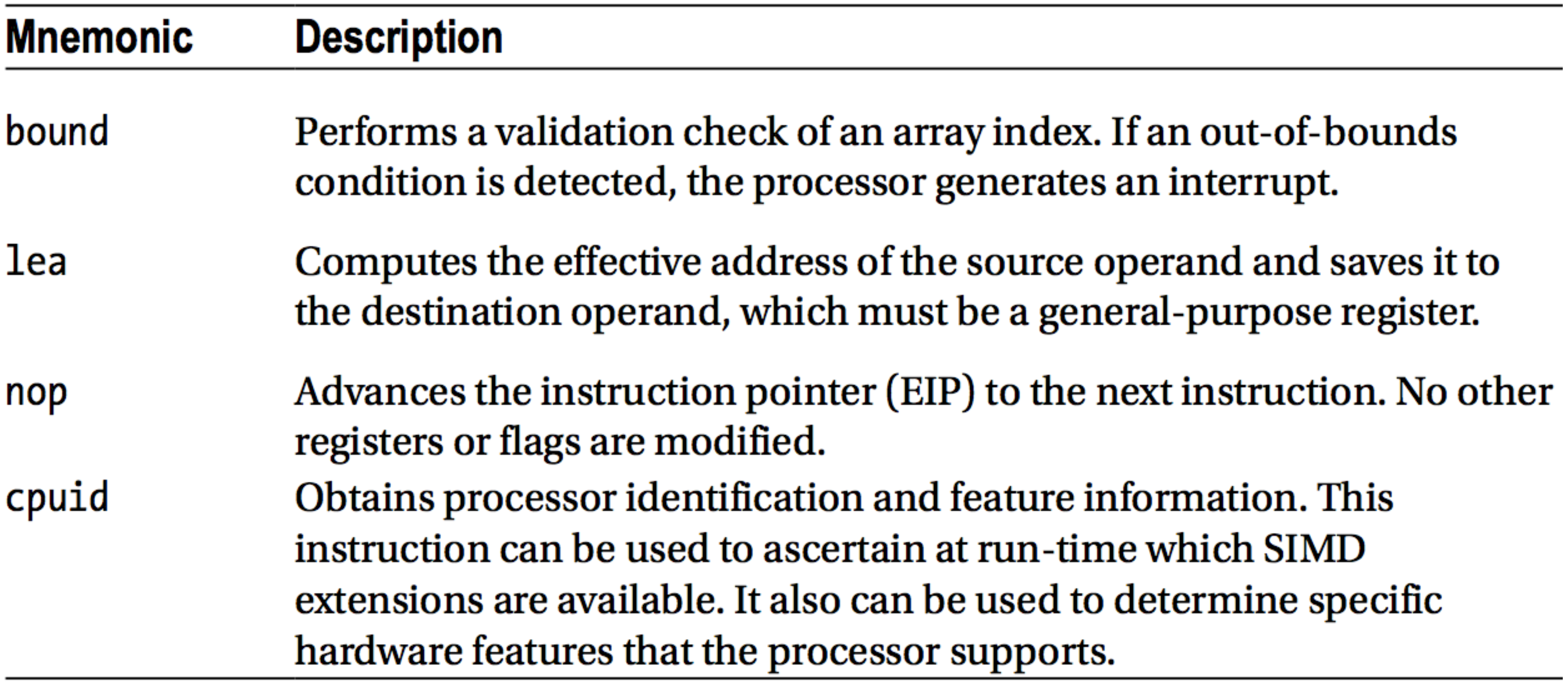
Control-Transfer Instructions (cont.)

Mnemonic	Description
`jecxz`	Performs a jump to the specified memory location if the condition `ECX == 0` is true.
`loop`	Subtracts one from register ECX and jumps to the specified memory location if the condition `ECX == 0` is true.
`loope` `loopz`	Subtracts one from register ECX and jumps to the specified memory location if the condition `ECX != 0 && ZF == 1` is true.
`loopne` `loopnz`	Subtracts one from register ECX and jumps to the specified memory location if the condition `ECX != 0 && ZF == 0` is true.

Jump If Condition is met

Mnemonic	Description
ja	Jump short if above (CF=0 and ZF=0)
jae	Jump short if above or equal (CF=0).
jb	Jump short if below (CF=1).
jbe	Jump short if below or equal (CF=1 or ZF=1).
jc	Jump short if carry (CF=1).
jcxz	Jump short if CX register is 0.
jecxz	Jump short if ECX register is 0.
jrcxz	Jump short if RCX register is 0.
je	Jump short if equal (ZF=1).
jg	Jump short if greater (ZF=0 and SF=OF).
jge	Jump short if greater or equal (SF=OF).
jl	Jump short if less (SF≠OF).
jle	Jump short if less or equal (ZF=1 or SF≠OF).

Miscellaneous



Mnemonic	Description
bound	Performs a validation check of an array index. If an out-of-bounds condition is detected, the processor generates an interrupt.
lea	Computes the effective address of the source operand and saves it to the destination operand, which must be a general-purpose register.
nop	Advances the instruction pointer (EIP) to the next instruction. No other registers or flags are modified.
cpuid	Obtains processor identification and feature information. This instruction can be used to ascertain at run-time which SIMD extensions are available. It also can be used to determine specific hardware features that the processor supports.

What you have learned

- Assembly Compilers & Editors
- NASM Program Structure
- Instructions