Microprocessor and Assembly Language

Chapter-Four Instructions

Outline

- **Data Movement Instructions**
  - MOV, PUSH/POP
- **Arithmetic and Logic instructions**
- **Arithmetic Instructions**
  - Addition,
  - Subtraction,
  - Multiplication,
  - Division
- **Basic Logic Instructions**
  - AND, OR,
  - Exclusive-OR, NOT, NEG

Introduction

- The microprocessor requires an assembler program, which generates machine language, because machine language instructions are too complex to efficiently generate by hand.
- This chapter describes the more on assembly language syntax(for **arithmetic and logical instructions**) and some of its directives.
- In our previous chapter we have seen some of the **data movement instructions** such as MOV and it's difference addressing modes.
- We have also seen how to operate with the **stack memory managements**. I.e PUSH and POP instructions.
- The latest data transfer instruction implemented on the Pentium Pro and above is the CMOV (conditional move) instruction.
- With this chapter we will see the **various arithmetic and logical instructions**.

Arithmetic Instruction

Addition

- **Addition (ADD)** appears in many forms in the microprocessor. This section details the use of the **ADD** instruction for **8-**, **16-**, and **32-bit binary addition**.
- A second form of addition, called **add-with-carry**, is introduced with the **ADC** instruction.
- Finally, the **increment instruction (INC)** is presented. Increment is a special type of addition that adds 1 to a number. Other forms of addition such as BCD and ASCII are also found.
- There are more than 32,000 variations of the ADD instruction in the instruction set, it is impossible to list them all in this table.
- The only types of addition **not allowed** are memory-to-memory and segment register.

Arithmetic Instruction(Addition)

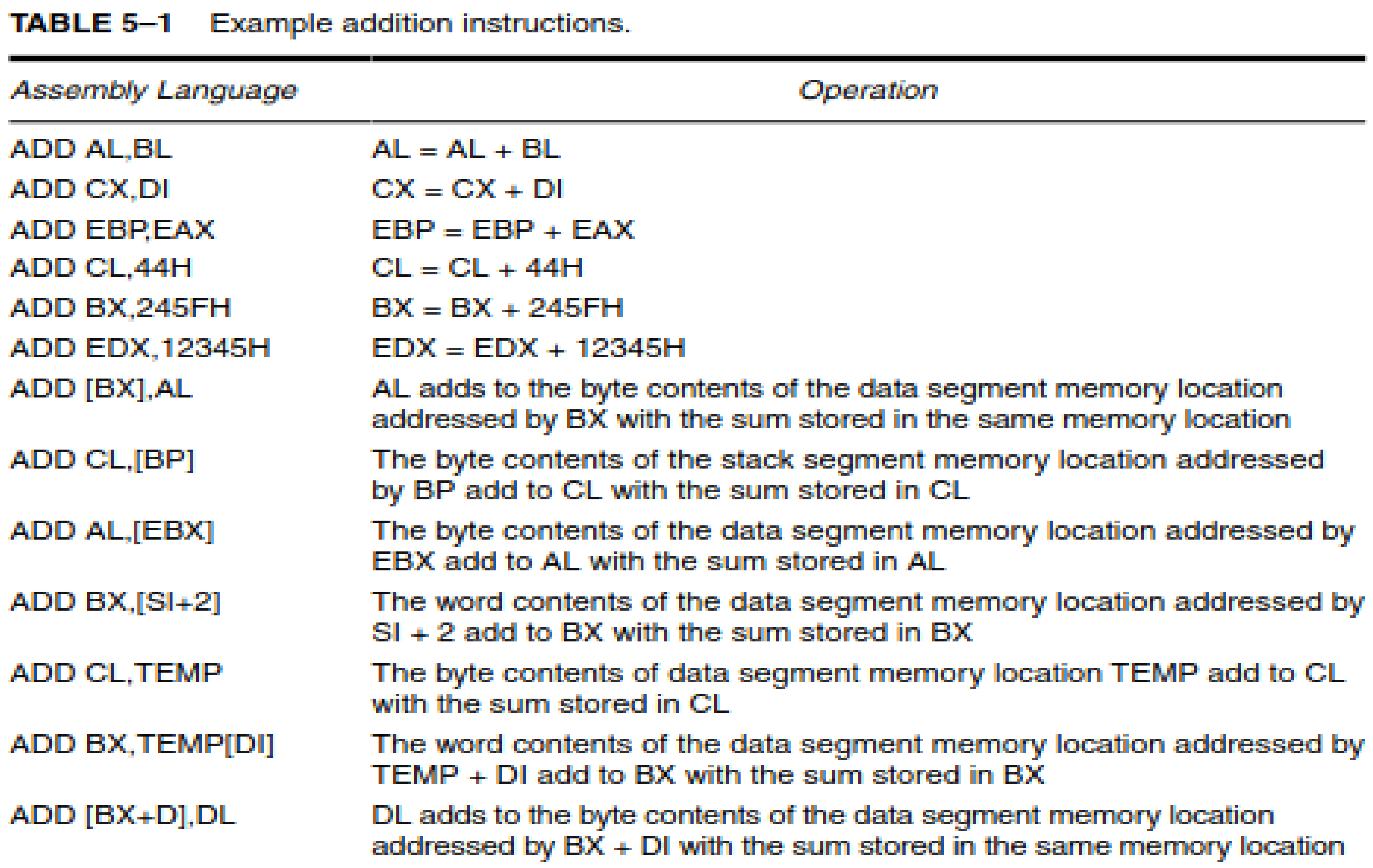


TABLE 5–1 Example addition instructions.

Assembly Language	*Operation*
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL
ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location

Arithmetic Instruction(Addition Cont...)

- The **segment registers** can only be moved, pushed, or popped.
- Note that, as with all other instructions, the **32-bit** registers are available only with the **80386 and above** mp.
- In the **64-bit** mode of the **Pentium 4** and **Core2**, the **64-bit registers** are also used for addition.
- On the next few slides we will see all the different types of addition instructions used in microprocessor.

Arithmetic Instruction(Addition Cont...)

- **Register Addition:** as shown below a simple sequence of instructions that uses register addition **to add the contents of several registers**.

✓ In this example, the contents of AX, BX, CX, and DX are added to form a 16-bit result stored in the AX register.

Example:

```
ADD AX, BX
ADD AX, CX
ADD AX, DX
```

- Whenever arithmetic(such as ADD) and logic instructions execute, the contents of the **flag register may** change.
- Any ADD instruction can modify the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags.

Arithmetic Instruction(Addition Cont...)

- **Immediate Addition:** is an arithmetic addition in which **constant** or **known data** is added to the content of the destination.
- **Example:** Look at the following 8-bit immediate addition.

```
MOV  DL,12H
ADD  DL, 33H
```

- ✓ That is DL is first loaded with 12H by using an immediate move instruction.
- ✓ Next, 33H is added to the 12H in DL by an immediate addition instruction.
- ✓ After the addition, the sum (45H) moves into register DL.

Arithmetic Instruction(Addition Cont...)

- **Memory - to - Register Addition:** the content of memory is added to a register.
- Suppose that an application requires **memory data** to be **added to** the AL **register**. Below is an example that adds two consecutive bytes of data, stored at the data segment offset locations NUMB and NUMB+1, to the AL register.
- **Example:** Look at the following 8-bit immediate addition.

```
MOV  DI, OFFSET NUMB      ;address NUMB
MOV  AL,0                 ;clear sum
ADD  AL,[DI]              ;add NUMB
ADD  AL,[DI+1]            ;add NUMB+1
```

Arithmetic Instruction(Addition Cont...)

- The first instruction loads the destination index register (DI) with offset address NUMB.
- The **DI** register, used in this example, addresses data in the data segment beginning at memory location **NUMB**. After clearing the sum to zero, the **ADD AL, [DI]** instruction adds the contents of memory location NUMB to AL.
- Finally, the **ADD AL, [DI+1]** instruction adds the contents of memory location NUMB plus 1 byte to the AL register.
- After both ADD instructions execute, the result appears in the AL register as the sum of the contents of NUMB plus the contents of NUMB+1.

Arithmetic Instruction(Addition Cont...)

- **Increment Addition:** this instruction **adds 1** to a register or a memory location(except a segment register).
- Below are some of the possible forms of the increment instructions available to the 8086–Core2 processors.

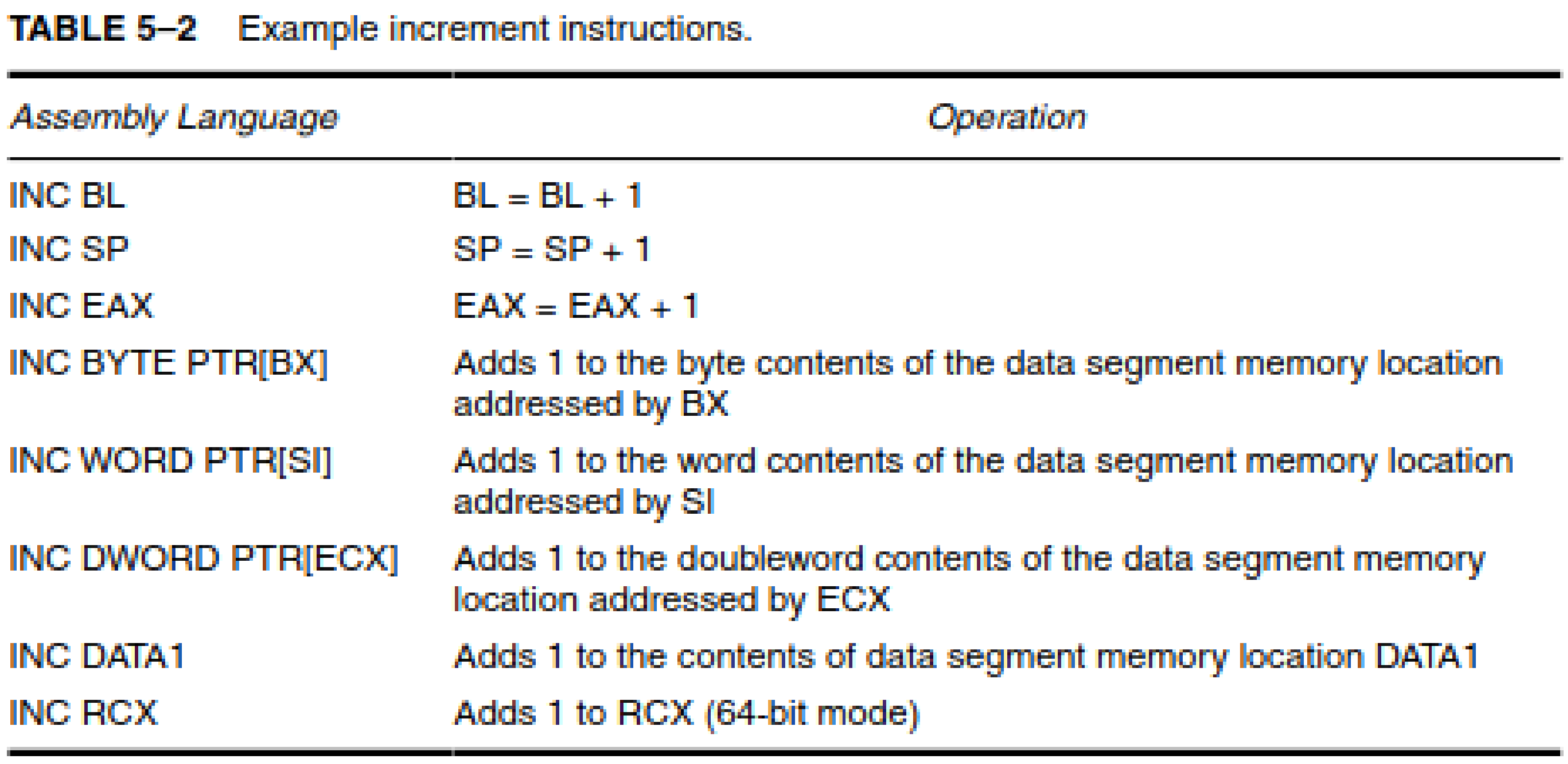


TABLE 5–2 Example increment instructions.

Assembly Language	*Operation*
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD PTR[ECX]	Adds 1 to the doubleword contents of the data segment memory location addressed by ECX
INC DATA1	Adds 1 to the contents of data segment memory location DATA1
INC RCX	Adds 1 to RCX (64-bit mode)

Arithmetic Instruction(Addition Cont...)

- As with other instructions presented thus far, it is impossible to show all variations of the **INC** instruction because of the large number available.
- Segment registers are not used with INC as the other operations.

  Exampe: INC DS ; this is illegal!
- With indirect memory increments, the size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives.
- The reason is that the assembler program cannot determine if, for example, the INC [DI] instruction can be a byte-, word-, or double word-sized increment.
- Thus, we have specify the size using the above directives.

  Therefore INC [DI] is illegal! Rather we have to write as follow:
- The INC BYTE PTR [DI] instruction clearly indicates **byte-sized** memory data;

Arithmetic Instruction(Addition Cont...)

- The INC WORD PTR [DI] instruction unquestionably indicates a **word-sized** memory data;
- The INC DWORD PTR [DI] instruction indicates **doubleword-sized** data.
- In 64-bit mode operation of the Pentium 4 and Core2, the INC QWORD PTR [RSI] instruction indicates **quadword-sized** data.

Note

- ✓ If the sum of two operands exceeds the size of destination operand, then it would set the carry flag to 1.
- ✓ The ADD instruction can affect CF, PF, SF, ZF, OF flags depending upon the result. If the result is zero, the ZF=1. Negative result sets SF to 1.

Arithmetic Instruction(Addition Cont...)

Addition-with-Carry(ADC**):** this ADC and ADD instruction perform the same operation of addition.

- The **only difference** is that ADC instruction also adds the carry flag bit to the sum of two operands. It adds the bit in the carry flag (C) to the operand data.
- This instruction mainly appears in software that adds numbers that are wider than 16 bits in the 8086–80286 or wider than 32 bits in the 80386–Core2.
- When we perform addition, the instruction may or may not generate a carry.
- A carry appears in the carry flag if the **sum is greater than 16bit**(while we're operating with 16bit register).

Arithmetic Instruction(Addition Cont...)

- The ADC instruction adds the 1 or the 0 in the carry flag to the most significant bits of the result.

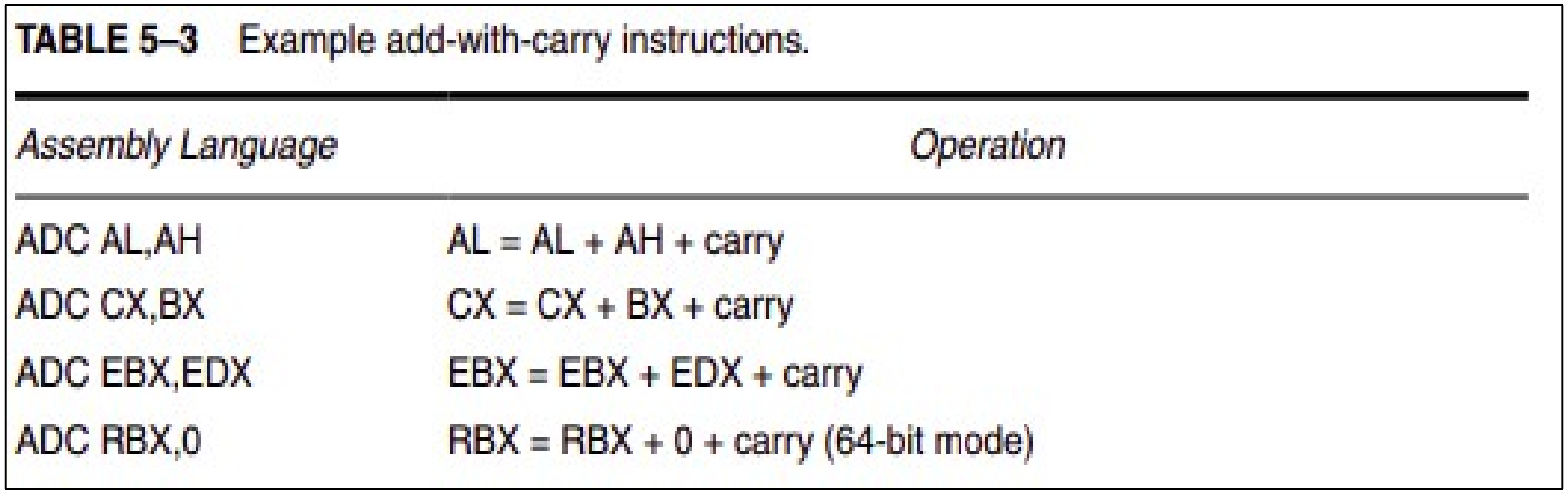


TABLE 5–3 Example add-with-carry instructions.

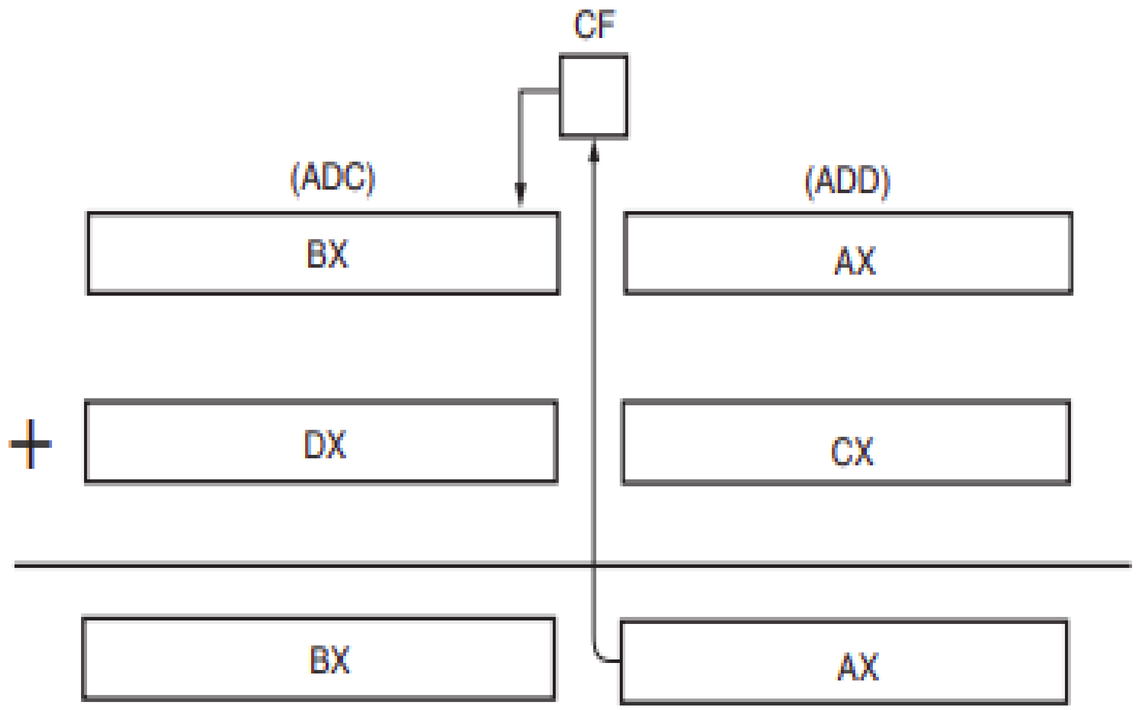
Assembly Language	*Operation*
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)

- Example:

```
ADD AX, CX
ADC BX, DX
```

Arithmetic Instruction(Addition Cont...)

- The above example can be better explained using the following diagram.



- With the first instruction: **ADD AX, CX;** if their is a carry that carry will be stored in CF and it will be added with the next instruction if its ADC!

Arithmetic Instruction(Addition Cont...)

- Let AX: 53ACH, BX: 1133H, CX: DB13H, DX: 253AH

```
AX:   5  3  A  C  H
CX:   D  B  1  3  H
-------------------
      2  E  B  F  H
```

Now we have **CF:1**

ADD AX, CX;

- If following the above instruction if we execute **ADC BX, DX**

```
BX:   1  1  3  3  H
DX:   2  5  3  A  H
-------------------
      3  6  6  E  H
```

Value of CF added here:

- And that is how the ADC works!

Arithmetic Instruction(Addition Cont...)

Exchange and Add (XADD)for the 80486–Core2 Processor: new type of addition that appears in the 80486-Core2.

- The XADD instruction adds the source to the destination and stores the sum in the destination, as with any addition.
- The **difference** is that after the addition takes place, the original value of the destination is copied into the source operand.
- This is **one of the few instructions that change the source**.

Example: **XADD BL, DL**

- If BL = 12H and DL = 02H, after this instruction BL contains the sum 14H and DL becomes 12H.
- Note: the original destination of 12H replaces the source. This instruction functions with any register size and any memory operand.

Arithmetic Instruction(Addition Cont...)

- Note: the difference between the **XADD** and **XCHG** instruction.

✓ **XCHG exchanges the values of two operands.**

XCHG reg, reg
XCHG reg, mem
XCHG mem, reg

Rules

- Operands must be of the same size
- At least one operand must be a register
- No immediate operands are permitted!

```
XCHG  AH, AL         ; exchange 8-bit  regs
XCHG  AX, BX         ; exchange 16-bit regs
XCHG  EAX, EBX       ; exchange 32-bit regs
XCHG  Var1, EBX      ; exchange 32-bit memory op(var1) with EBX
XCHG  Var1, Var2     ; error: two memory operands!
          ; Note on the above Varx is used as a memory. (where x: 1, 2, 3...)
```

Arithmetic Instruction(Subtraction)

Subtraction (SUB)

- Many forms of subtraction (SUB) appear in the instruction set.

These forms use any addressing mode with 8-, 16-, 32-bit,...data.

- A special form of subtraction (**decrement**, or **DEC**) subtracts 1 from any register or memory location.

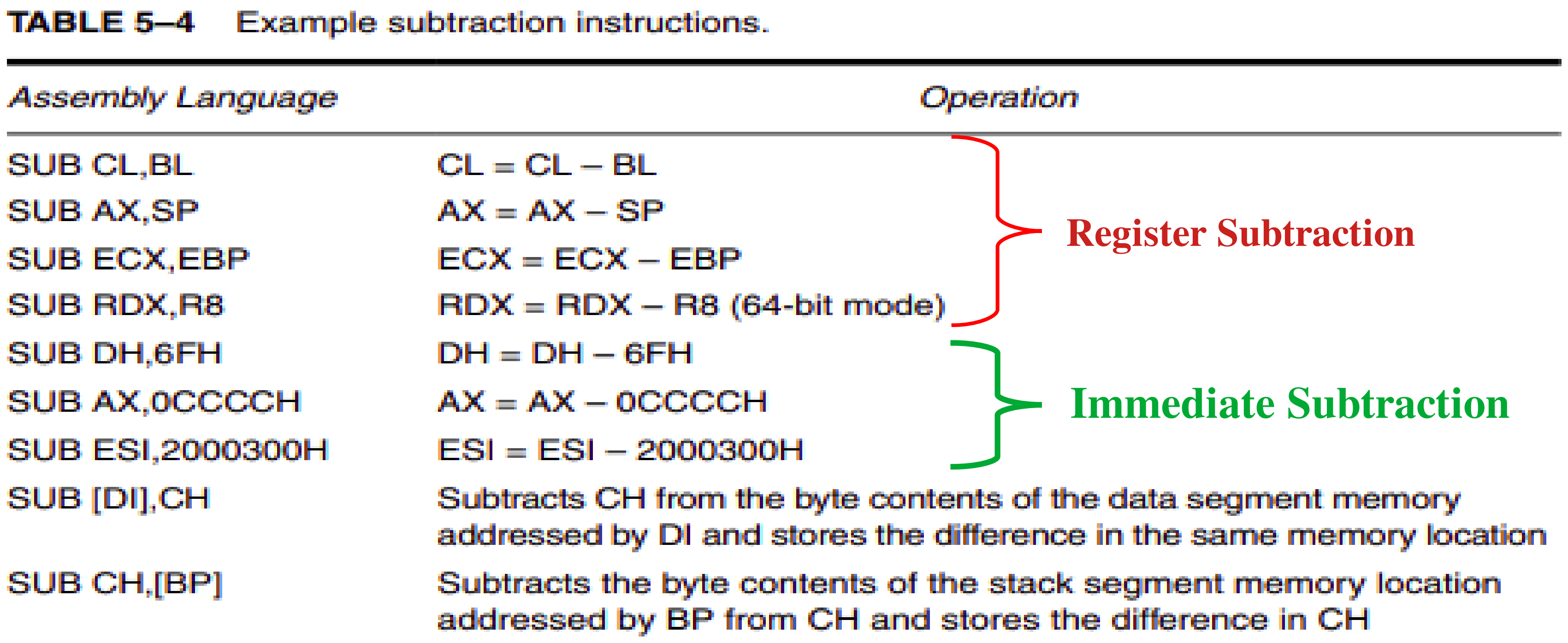


TABLE 5–4 Example subtraction instructions.

Assembly Language	*Operation*
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
SUB RDX,R8	RDX = RDX – R8 (64-bit mode)
SUB DH,6FH	DH = DH – 6FH
SUB AX,0CCCCH	AX = AX – 0CCCCH
SUB ESI,2000300H	ESI = ESI – 2000300H
SUB [DI],CH	Subtracts CH from the byte contents of the data segment memory addressed by DI and stores the difference in the same memory location
SUB CH,[BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH

Register Subtraction

Immediate Subtraction

Arithmetic Instruction(Subtraction)

- The only types of subtraction that are not allowed:
  - memory-to-memory and,
  - segment register subtractions.
- Like other arithmetic instructions, the subtract instruction affects the flag bits.
- Example:

  MOV AX, 1H

  DEC AX ;decrement instruction is typical subtraction that subtracts one from the operand(AX in this case).
- The above instruction affects the Zero flag, setting the value of ZF = 1.

Arithmetic Instruction(Subtraction)

- **Register Subtraction:** Look example below that subtracts the 16-bit contents of registers CX and DX from the contents of register BX. After each subtraction, the microprocessor modifies the contents of the flag register.
- The flags change for most arithmetic and logic operations.

```
Example: SUB BX, CX
         SUB BX, DX
```

- **Immediate Subtraction:** As with addition, the microprocessor also allows immediate operands for the subtraction of constant data.

```
Example: MOV  CH, 22H
         SUB  CH, 44H
```

- The above instruction subtracts 44H from 22H. Here, we first load the 22H into CH using an immediate move.

Arithmetic Instruction(Subtraction)

- **Decrement Subtraction:** Decrement subtraction (DEC) subtracts 1 from a register or the contents of a memory location.

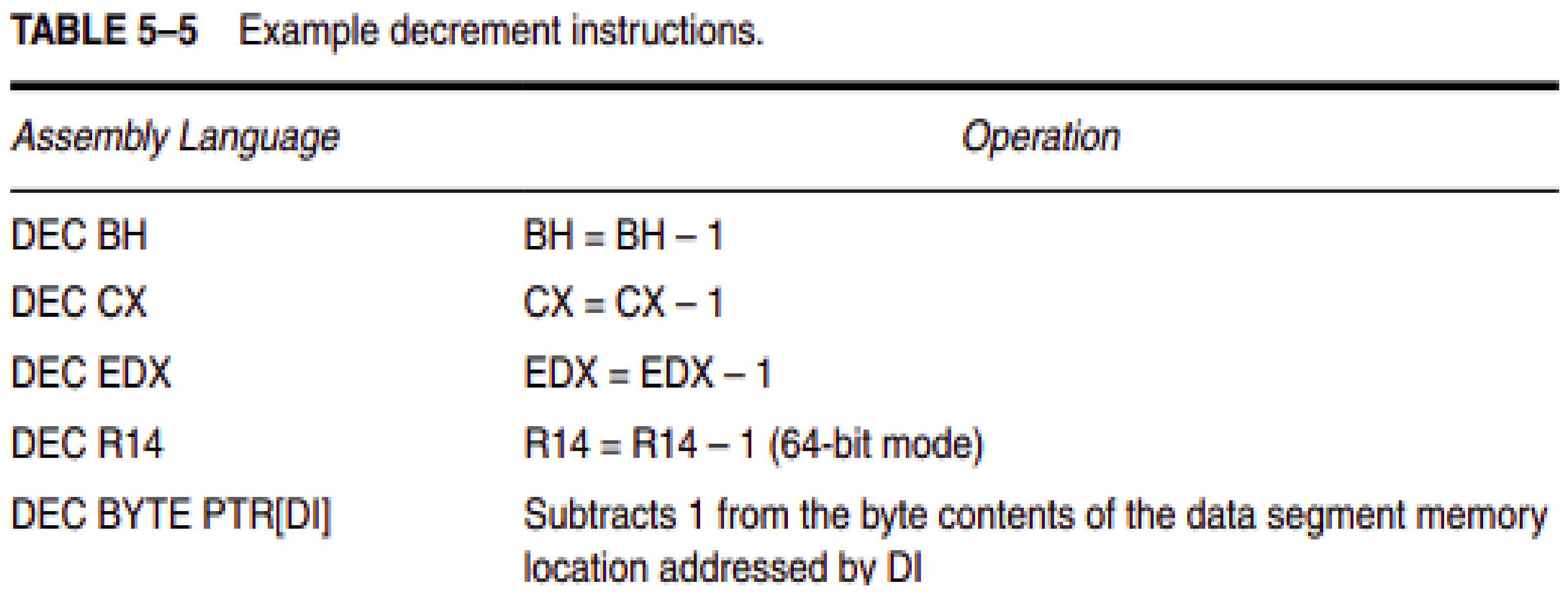


TABLE 5–5 Example decrement instructions.

Assembly Language	*Operation*
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
DEC R14	R14 = R14 – 1 (64-bit mode)
DEC BYTE PTR[DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI

- The decrement indirect memory data instructions require BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR because the assembler cannot distinguish a byte from a word or doubleword when an index register addresses memory.

Arithmetic Instruction(Subtraction) Cont..

- For example, **DEC [SI]** is vague because the assembler cannot determine whether the location addressed by SI is a byte, word, or doubleword.

✓ Using DEC BYTE PTR[SI],

DEC WORD PTR[DI], or

DEC DWORD PTR[SI] reveals size of data to assembler.

Arithmetic Instruction(Subtraction) Cont..

- **Comparison**: the comparison instruction (**CMP**) is a subtraction that changes only the flag bits; the destination operand never changes.
- A comparison is useful for checking the entire contents of a register or a memory location against another value.
- A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.
- The only disallowed forms of compare are memory-to-memory and segment register compares.

Example:

```
CMP  AL, 10H     ;compare AL against 10H
JAE  SUBER       ;if AL is 10H or above
```

In this example, the contents of AL are compared with 10H.

Arithmetic Instruction(Subtraction) Cont..

Conditional jump instructions that often follow the comparison are:

1. JA (jump above)
2. JB (jump below).
3. JAE (jump if above or equal)
4. JBE (jump if below or equal)

- Applying on the previous CMP example;
- If the JA follows the comparison, the jump occurs if the value in AL is above 10H.
- If the JB follows the comparison, the jump occurs if the value in AL is below 10H.
- In the above example, the JAE(jump above or equal) instruction follows the comparison. This instruction causes the program to continue at memory location SUBER if the value in AL is $\geq$10H.
- There is also a JBE (jump below or equal) instruction that could follow the comparison to jump if the outcome is $\leq$ 10H.

Arithmetic Instruction(Subtraction) Cont..

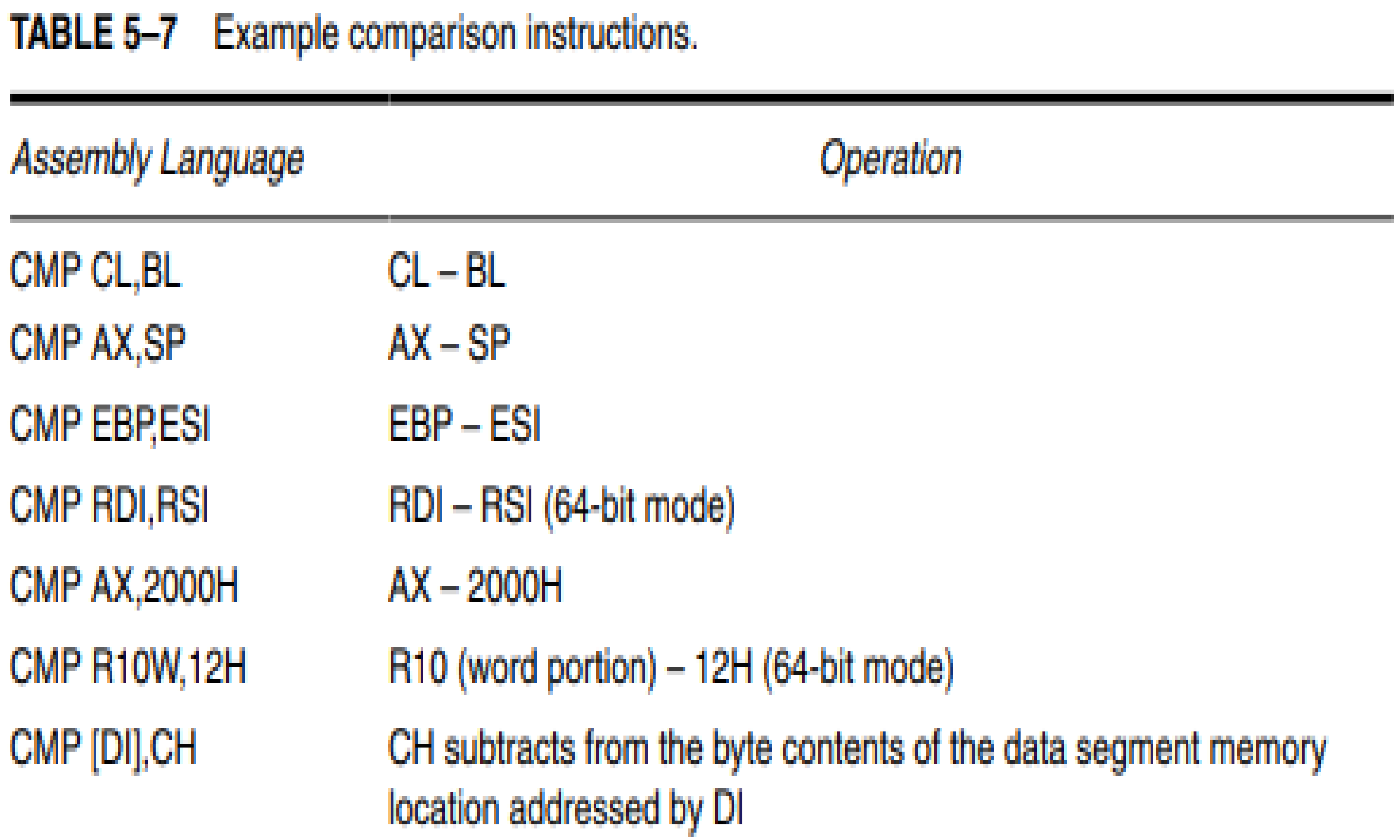


TABLE 5–7 Example comparison instructions.

Assembly Language	*Operation*
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP RDI,RSI	RDI – RSI (64-bit mode)
CMP AX,2000H	AX – 2000H
CMP R10W,12H	R10 (word portion) – 12H (64-bit mode)
CMP [DI],CH	CH subtracts from the byte contents of the data segment memory location addressed by DI

Arithmetic Instruction(Cont...)

- **Compare and Exchange (80486–Core2 Processors Only)**: abbreviated as **CMPXCHG** and is found only in the 80486 through the Core2 instruction sets.
- It compares the **destination** operand with the **accumulator**.

✓ If they are **equal**, the source operand is copied into the Accumulator;

✓ If they are **not equal**, the destination operand is copied into the accumulator.

- This instruction functions with 8-, 16-, 32- or 64-bit data.

CMPXCHG Dest, Src

if Des = Accumulator then Accumulator = Src

if Des ≠ Accumulator then Accumulator = Dest

Arithmetic Instruction(Cont...)

- **Example:**

  CMPXCHG CX, DX ;it first compares the contents of CX with AX.

  ✓ If CX equals AX, $\longrightarrow$ DX is copied into AX;

  ✓ If CX is not equal to AX, $\longrightarrow$ CX is copied into AX.

- This instruction also compares AL with 8-bit data and EAX with 32-bit data if the operands are either 8- or 32-bit.

- Note: The accumulator is a register that is part of arithmetic/logic unit (ALU). It can be of size 8bit, 16bit, 32bit and 64bit.

Arithmetic Instruction(multiplication)

- Multiplication is performed on bytes, words, or doublewords, and can be signed integer (IMUL) or unsigned integer (MUL).
- Note that, as usual only the 80386 through the Core2 processors multiply 32-bit doublewords.
- The product after a multiplication is always a double-width product.
  - ✓ If two 8-bit numbers are multiplied, they generate a 16-bit product;
  - ✓ If two 16-bit numbers are multiplied, they generate a 32-bit product; and
  - ✓ If two 32-bit numbers are multiplied, a 64-bit product is generated.
  - ✓ In the 64-bit mode of the Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product.

Arithmetic Instruction(multiplication)

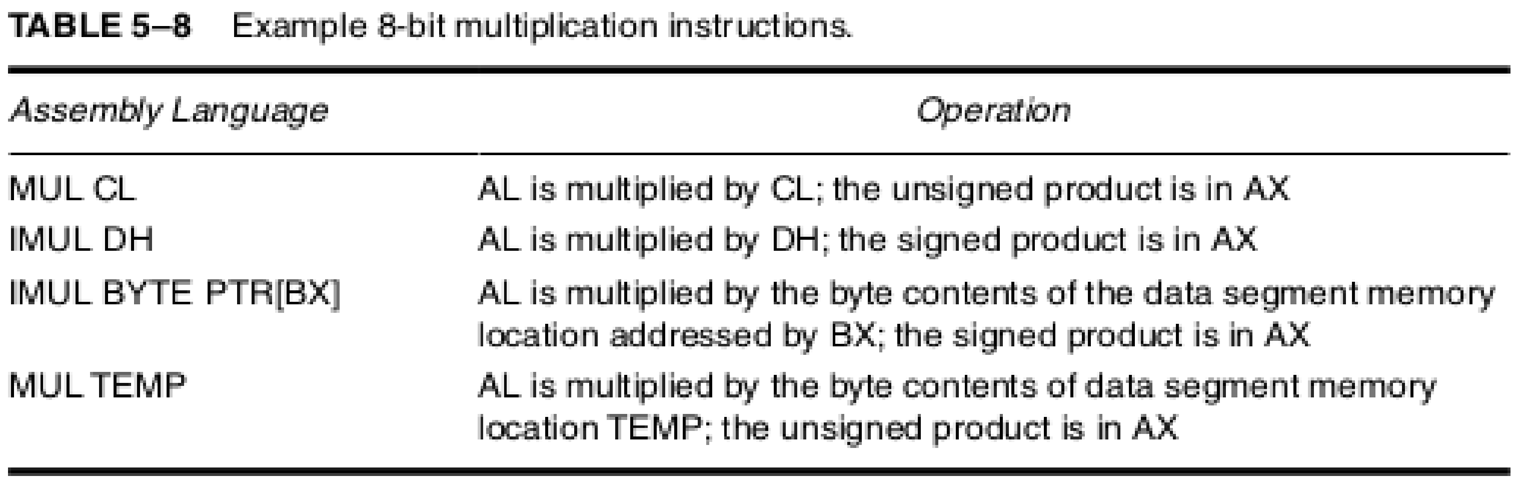


TABLE 5–8 Example 8-bit multiplication instructions.

Assembly Language	*Operation*
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

- Some flag bits (overflow and carry) change when the multiply instruction executes and produce predictable outcomes.
- The other flags also change, but their results are unpredictable and therefore are unused.

Arithmetic Instruction(multiplication)

8-Bit Multiplication: with 8-bit multiplication, the multiplicand is always in the AL register, whether signed or unsigned.

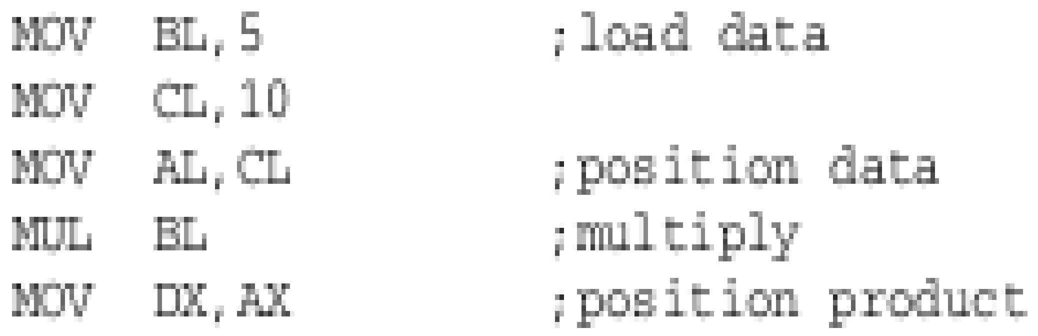
- The multiplier can be any 8-bit register or any memory location.
- Immediate multiplication is not allowed unless the special signed immediate multiplication instruction, discussed later in this section, appears in a program.
- The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL.

Example: MUL BL ;multiplies the unsigned contents of AL by the unsigned contents of BL.

► After the multiplication, the unsigned product is placed in AX - a double-width product.

Arithmetic Instruction(multiplication)

- Suppose that BL and CL each contain two 8-bit unsigned numbers, and these numbers must be multiplied to form a 16-bit product stored in DX.
- This procedure cannot be accomplished by a single instruction because we can only multiply a number times the AL register for an 8-bit multiplication.
- Below shows a short program that generates DX = BL * CL . This example loads register BL and CL with example data 5 and 10. The product, a 50, moves into DX from AX after the multiplication by using the MOV DX,AX instruction.



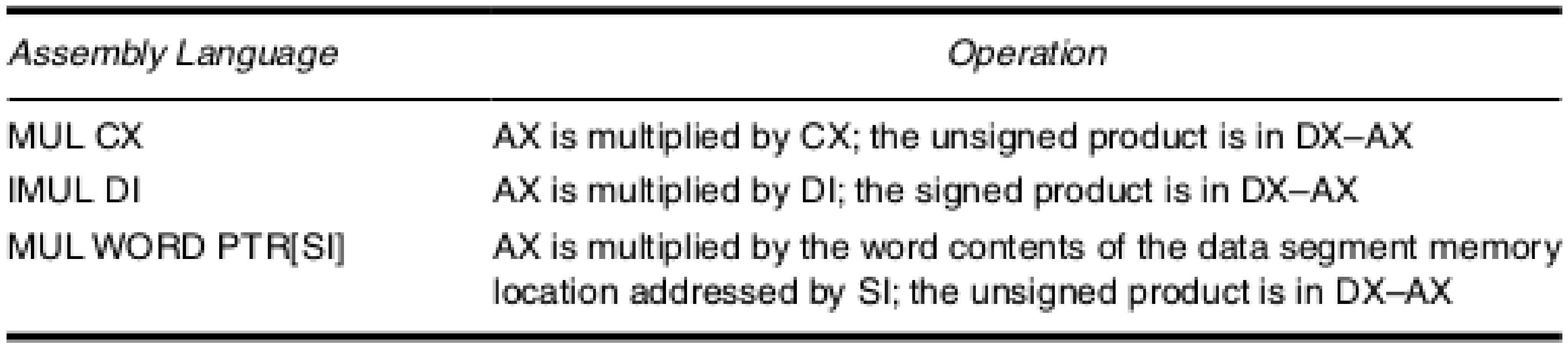
```
MOV   BL,5          ;load data
MOV   CL,10
MOV   AL,CL         ;position data
MUL   BL            ;multiply
MOV   DX,AX         ;position product
```

Arithmetic Instruction(multiplication)

16-Bit Multiplication: this multiplication is very similar to byte multiplication.

- The difference is that AX contains the multiplicand instead of AL, and the 32-bit product appears in **DX–AX** instead of AX.
- The **DX** register always contains the most significant 16 bits of the product, and **AX** contains the least significant 16 bits.
- As with 8-bit multiplication, the choice of the multiplier is up to the programmer.

TABLE 5–9 Example 16-bit multiplication instructions.



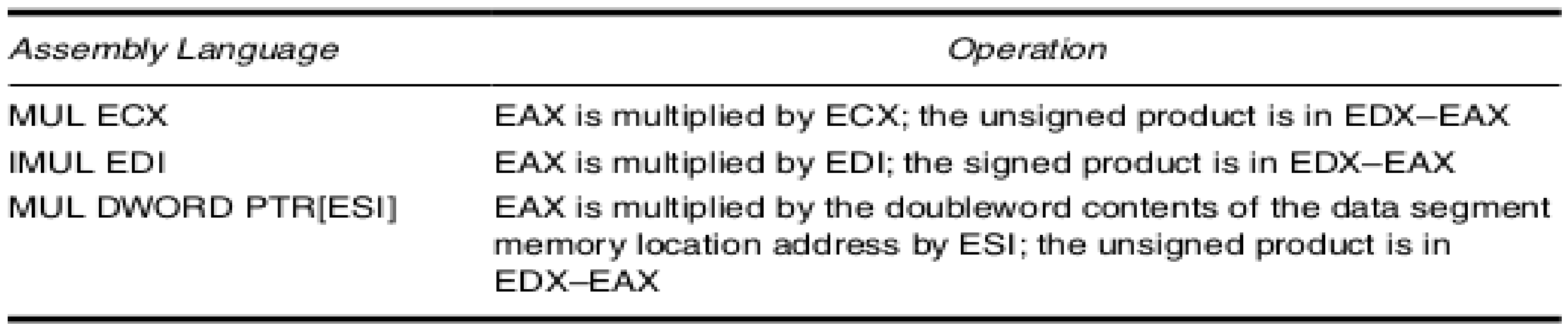
Assembly Language	*Operation*
MUL CX	AX is multiplied by CX; the unsigned product is in DX–AX
IMUL DI	AX is multiplied by DI; the signed product is in DX–AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX–AX

Arithmetic Instruction(multiplication)

32-Bit Multiplication: In the 80386 and above, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers.

- As with 8- and 16-bit multiplication, 32-bit multiplication can be signed or unsigned by using the IMUL and MUL instructions.
- With 32-bit multiplication, the contents of EAX are multiplied by the operand specified with the instruction.
- The product (64 bits wide) is found in **EDX–EAX**, where **EAX** contains the least significant 32 bits and **EDX** contains the most significant 32 bit of the product.

TABLE 5–10 Example 32-bit multiplication instructions.



Assembly Language	*Operation*
MUL ECX	EAX is multiplied by ECX; the unsigned product is in EDX–EAX
IMUL EDI	EAX is multiplied by EDI; the signed product is in EDX–EAX
MUL DWORD PTR[ESI]	EAX is multiplied by the doubleword contents of the data segment memory location address by ESI; the unsigned product is in EDX–EAX

Arithmetic Instruction(multiplication)

64-Bit Multiplication: The result of a 64-bit multiplication in the Pentium 4 appears in the **RDX - RAX** register pair as a 128-bit product.

- **RAX** contains the least significant 64 bits and **RDX** contains the most significant 64 bit of the product.
- Although multiplication of this size is relatively rare, the Pentium 4 and Core2 can perform it on both signed and unsigned numbers.

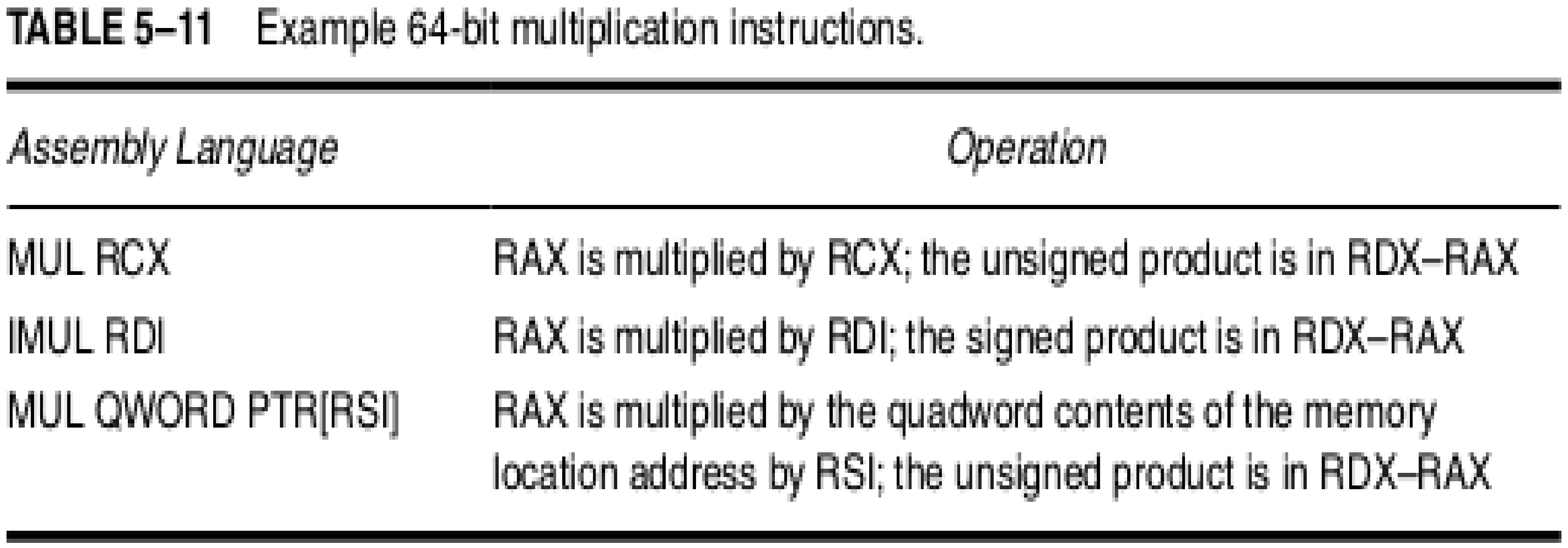
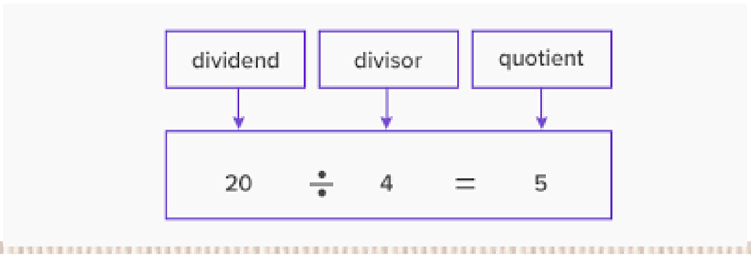


TABLE 5–11 Example 64-bit multiplication instructions.

Assembly Language	*Operation*
MUL RCX	RAX is multiplied by RCX; the unsigned product is in RDX–RAX
IMUL RDI	RAX is multiplied by RDI; the signed product is in RDX–RAX
MUL QWORD PTR[RSI]	RAX is multiplied by the quadword contents of the memory location address by RSI; the unsigned product is in RDX–RAX

Arithmetic Instruction(Division)

- As with multiplication, division occurs on 8- or 16-bit numbers in the 8086–80286 microprocessors, and on 32-bit numbers in the 80386 and above microprocessor.
- These numbers are signed(IDIV) or unsigned (DIV) integers.
- The dividend is always double-width of the divisor. This means that;
  - ✓ an 8-bit division divides a 16-bit number by an 8-bit number;
  - ✓ a 16-bit division divides a 32-bit number by a 16-bit number; and
  - ✓ a 32-bit division divides a 64-bit number by a 32-bit number.
- There is no immediate division instruction available to any microprocessor.
- In the 64-bit mode of the Pentium 4 and Core2, a 64-bit division divides a 128-bit number by a 64-bit number.



Arithmetic Instruction(Division)

- A division can result in two different types of errors; one is an attempt to divide by zero and the other is a divide overflow.
- A **divide overflow** occurs when a small number divides a large number.
- For example, suppose that AX = 3000 and that it is divided by 2. Because the quotient for an 8-bit division appears in **AL**, the result of 1500 causes a divide overflow because the 1500 does not fit into AL.
- In either case, the microprocessor generates an interrupt if a divide error occurs.
- In most systems, a divide error interrupt displays an error message on the video screen.

Arithmetic Instruction(Division)

8-Bit Division: An 8-bit division uses the AX register to store the dividend that is divided by the contents of any 8-bit register or memory location.

- The quotient moves into AL after the division with AH containing a whole number remainder.
- For a signed division, the quotient is positive or negative; the remainder always assumes the sign of the dividend and is always an integer.

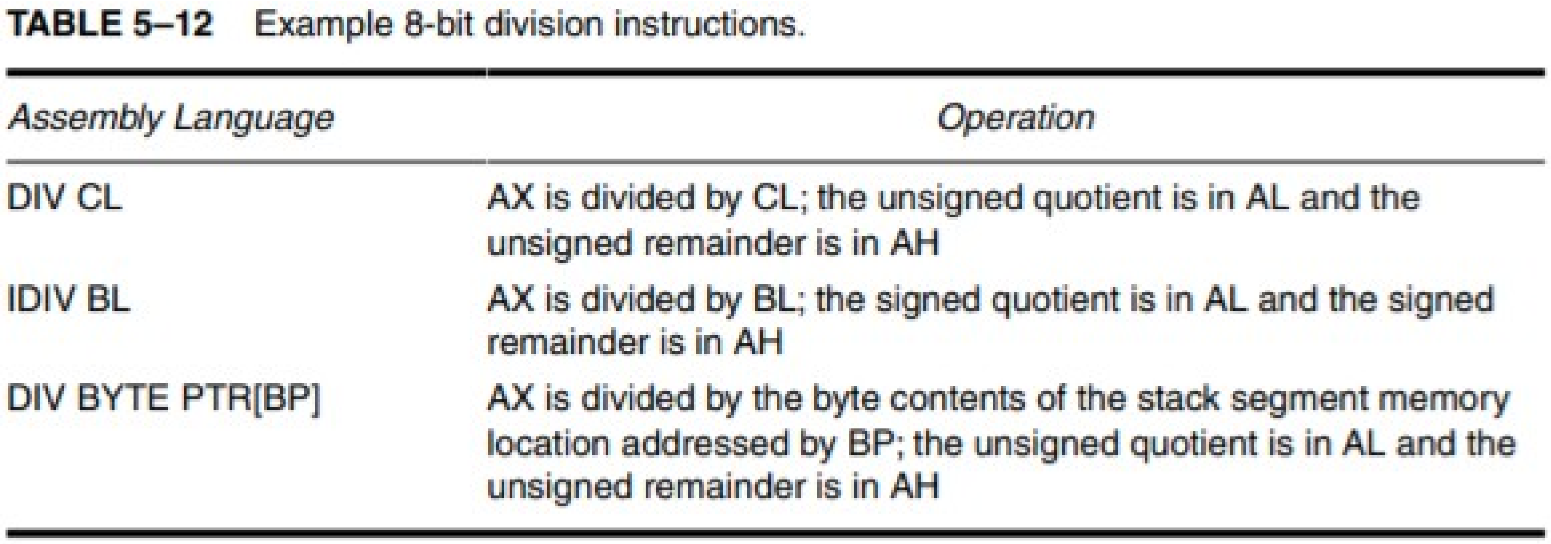


TABLE 5–12 Example 8-bit division instructions.

Assembly Language	*Operation*
DIV CL	AX is divided by CL; the unsigned quotient is in AL and the unsigned remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the signed remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP; the unsigned quotient is in AL and the unsigned remainder is in AH

Arithmetic Instruction(Division)

- What is given along with the DIV or IDIV opcode is the divisor, and by default the dividend is the content of AX(in 8bit division).
- Example:

DIV BH

- The above instruction divides a 16bit content of AX register by content of BH. The quotient will be stored in AL and if there is any remainder it will be stored in AH register.

Example: below is how to perform division of 24 by 2 (24 ÷ 2).

```
MOV AX, 24    ;load 24
MOV BL, 2     ;load 2
DIV BL
```

Arithmetic Instruction(Division)

16-Bit Division: Sixteen-bit division is similar to 8-bit division, except instead of AX for dividend, it uses a 32bit that is stored in **DX - AX** register. And the divisor can be any 16-bit register or memory location.

- In this type of division the quotient appears in AX and the remainder appears in DX.

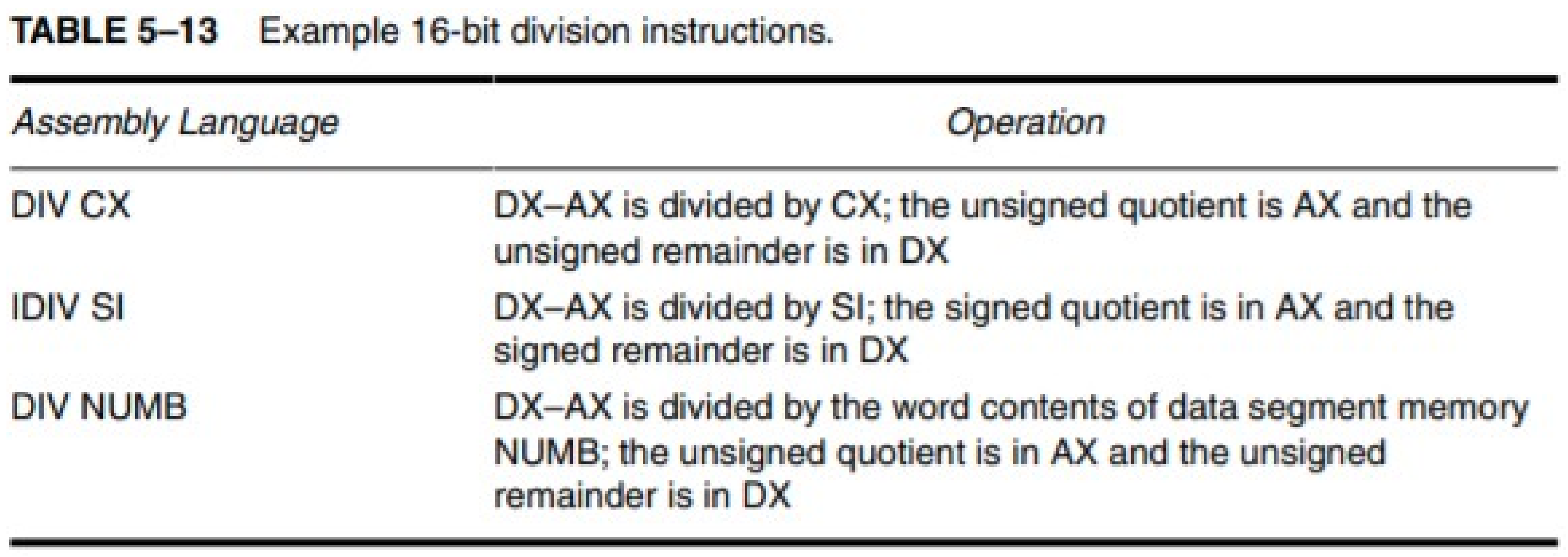


TABLE 5–13 Example 16-bit division instructions.

Assembly Language	*Operation*
DIV CX	DX–AX is divided by CX; the unsigned quotient is AX and the unsigned remainder is in DX
IDIV SI	DX–AX is divided by SI; the signed quotient is in AX and the signed remainder is in DX
DIV NUMB	DX–AX is divided by the word contents of data segment memory NUMB; the unsigned quotient is in AX and the unsigned remainder is in DX

Arithmetic Instruction(Division)

32-Bit Division: The 80386 through the Pentium 4 processors perform 32-bit division on signed or unsigned numbers.

- The 64-bit contents of **EDX - EAX** are divided by the operand specified by the instruction, leaving a 32-bit quotient in EAX and a 32-bit remainder in EDX.
- Other than the size of the registers, this instruction functions in the same manner as the 8- and 16-bit divisions.

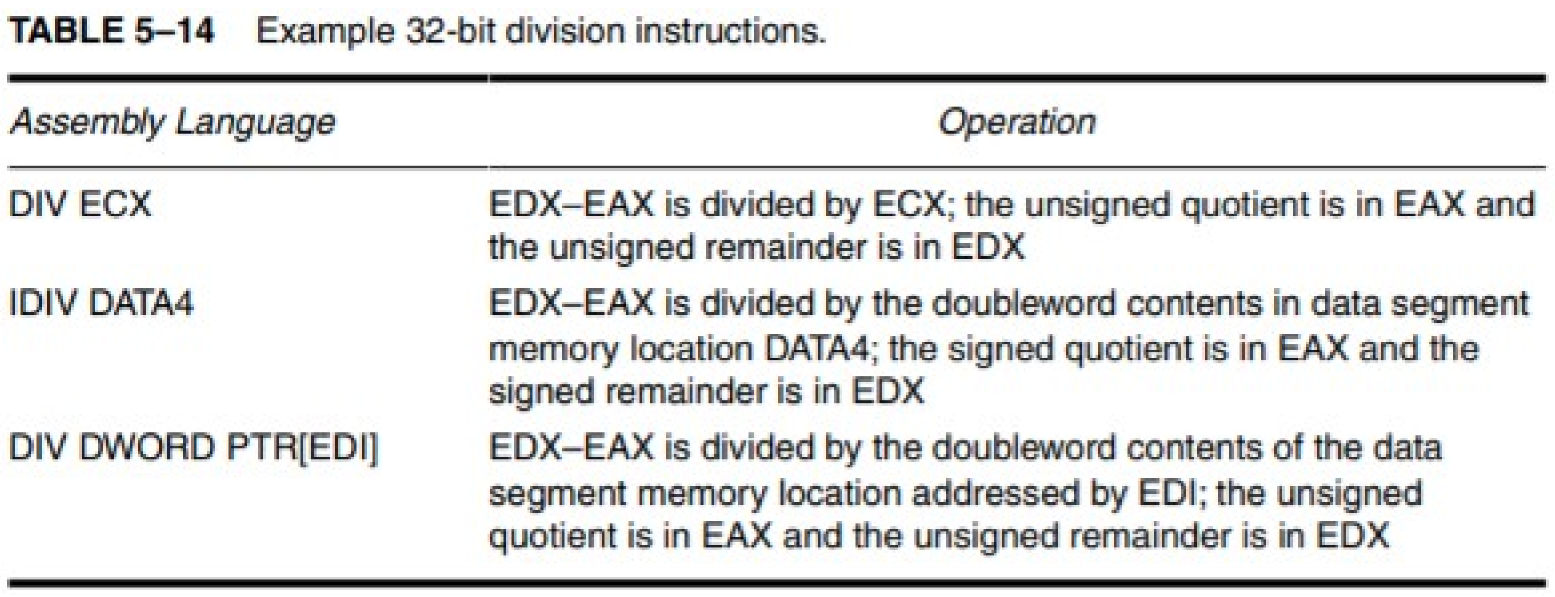


TABLE 5–14 Example 32-bit division instructions.

Assembly Language	*Operation*
DIV ECX	EDX–EAX is divided by ECX; the unsigned quotient is in EAX and the unsigned remainder is in EDX
IDIV DATA4	EDX–EAX is divided by the doubleword contents in data segment memory location DATA4; the signed quotient is in EAX and the signed remainder is in EDX
DIV DWORD PTR[EDI]	EDX–EAX is divided by the doubleword contents of the data segment memory location addressed by EDI; the unsigned quotient is in EAX and the unsigned remainder is in EDX

Arithmetic Instruction(Division)

64-Bit Division: The Pentium 4 processor operated in 64-bit mode performs 64-bit division on signed or unsigned numbers.

- The 64-bit division uses the **RDX - RAX** register pair to hold the dividend and the quotient is found in RAX and the remainder is in RDX after the division.

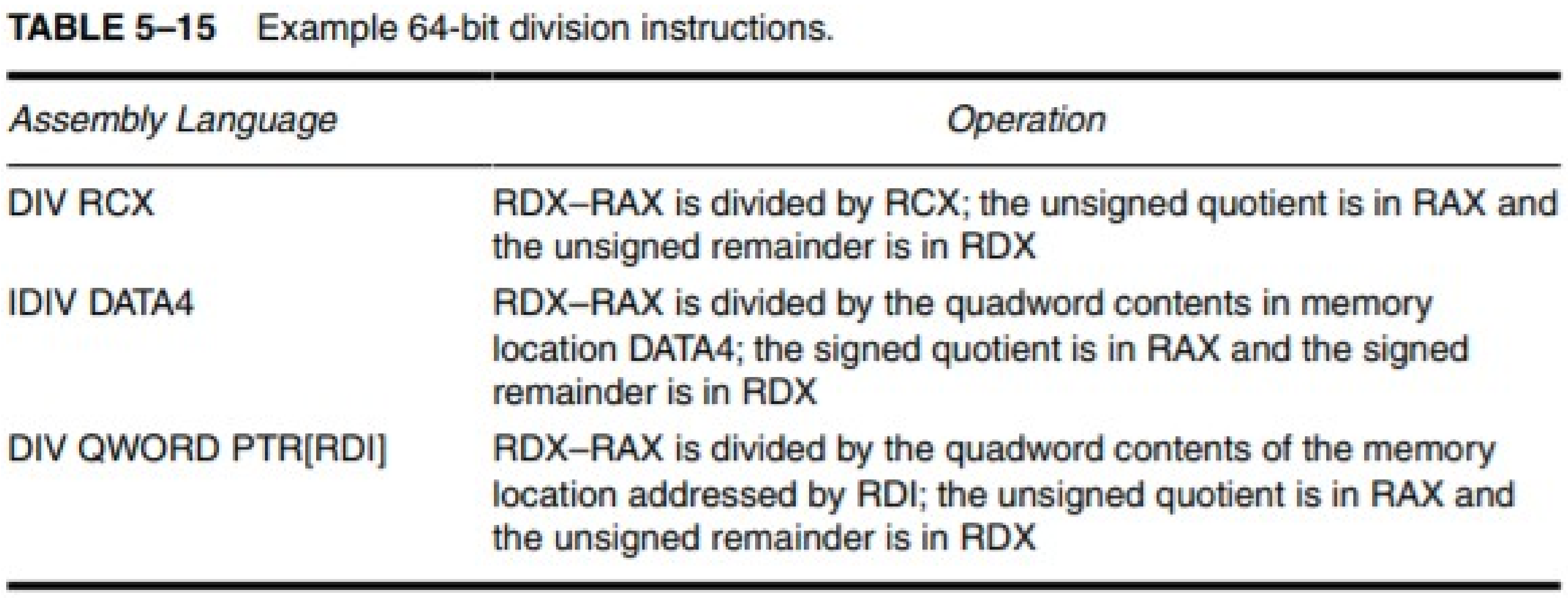


TABLE 5–15 Example 64-bit division instructions.

Assembly Language	*Operation*
DIV RCX	RDX–RAX is divided by RCX; the unsigned quotient is in RAX and the unsigned remainder is in RDX
IDIV DATA4	RDX–RAX is divided by the quadword contents in memory location DATA4; the signed quotient is in RAX and the signed remainder is in RDX
DIV QWORD PTR[RDI]	RDX–RAX is divided by the quadword contents of the memory location addressed by RDI; the unsigned quotient is in RAX and the unsigned remainder is in RDX

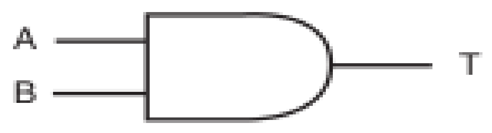
Basic Logic Instructions- AND

- The basic logic instructions include AND, OR, Exclusive-OR, and NOT.
- The AND operation performs logical multiplication, as illustrated below. Here, bits A and B, are ANDed to produce the result T.
- As indicated by the truth table, T is a logic 1 only when both A and B are logic 1s.
- For all other input combinations of A and B, T is a logic 0.
- It is important to remember that 0 AND anything is always 0, and 1 AND 1 is always 1.

FIGURE 5–3 (a) The truth table for the AND operation and (b) the logic symbol of an AND gate.

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

(a)



(b)

Basic Logic Instructions- AND

- Look the following example.

  Example:

```
MOV  AX, 32H
MOV  BX, 11H
AND  AX, BX
```

Here is how it works…

```
          0 0 1 1 0 0 1 0        (32H)
AND       0 0 0 1 0 0 0 1        (11H)
          ---------------        -----
          0 0 0 1 0 0 0 0        (10H)
          ---------------        -----
```

Note that, the ANDed result will be stored in AX register finally.

Basic Logic Instructions- AND

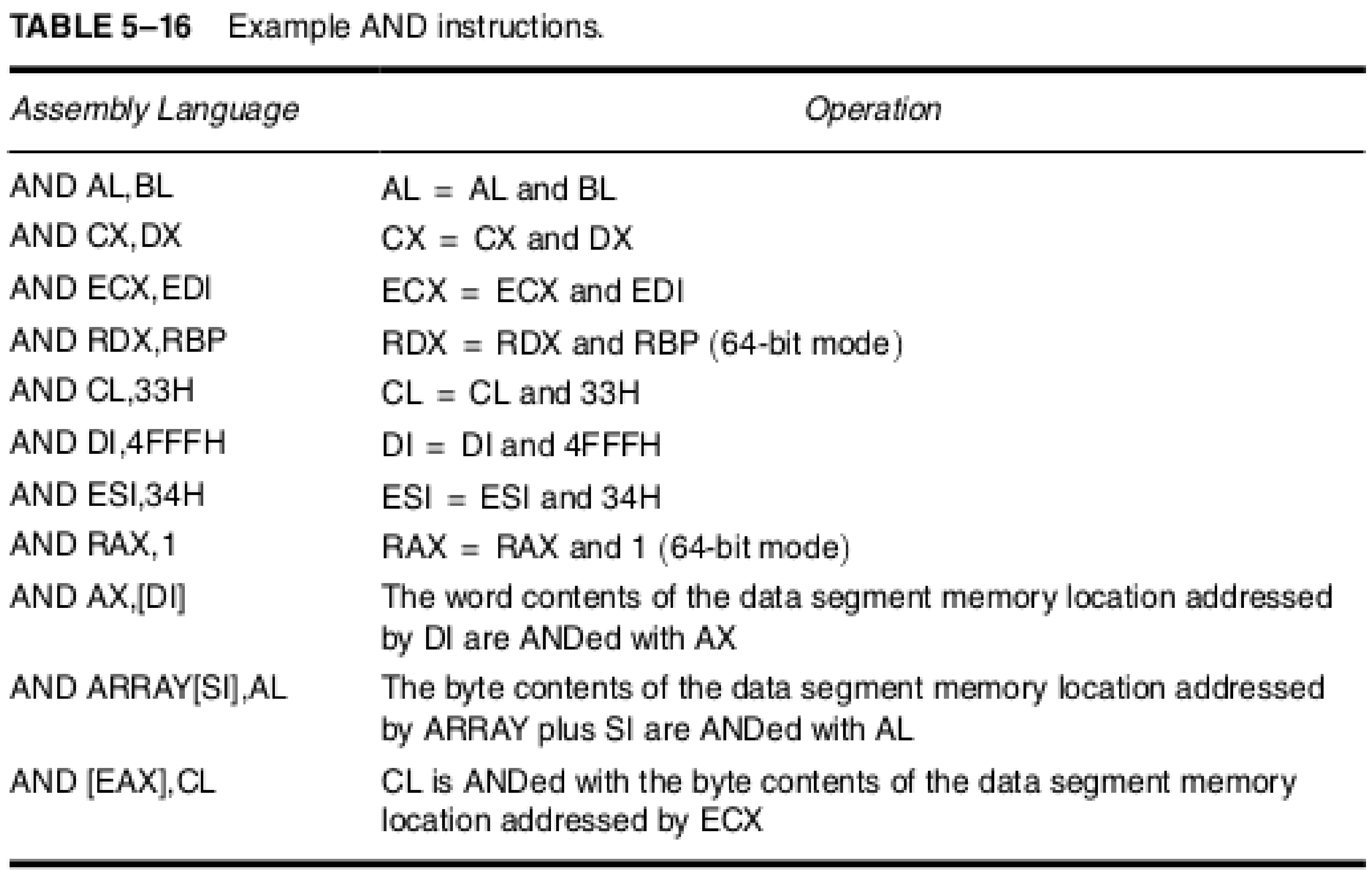


TABLE 5–16 Example AND instructions.

Assembly Language	*Operation*
AND AL,BL	AL = AL and BL
AND CX,DX	CX = CX and DX
AND ECX,EDI	ECX = ECX and EDI
AND RDX,RBP	RDX = RDX and RBP (64-bit mode)
AND CL,33H	CL = CL and 33H
AND DI,4FFFH	DI = DI and 4FFFH
AND ESI,34H	ESI = ESI and 34H
AND RAX,1	RAX = RAX and 1 (64-bit mode)
AND AX,[DI]	The word contents of the data segment memory location addressed by DI are ANDed with AX
AND ARRAY[SI],AL	The byte contents of the data segment memory location addressed by ARRAY plus SI are ANDed with AL
AND [EAX],CL	CL is ANDed with the byte contents of the data segment memory location addressed by ECX

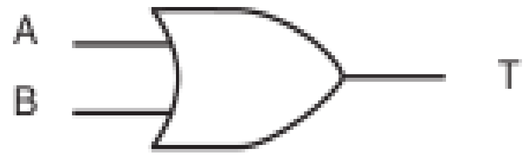
Basic Logic Instructions- OR

- The OR operation performs logical addition and is often called the **Inclusive-OR** function.
- The OR function generates a logic 1 output if any of the inputs are 1, or both of the input are 1.
- A 0 appears at the output only when all inputs are 0.

FIGURE 5–5 (a) The truth table for the OR operation and (b) the logic symbol of an OR gate.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

(a)



(b)

Basic Logic Instructions- OR

- Look the following example.

  Example:

```
MOV  AX, 32H
MOV  BX, 11H
OR   AX, BX
```

Here is how it works…

```
        0 0 1 1 0 0 1 0      (32H)
OR      0 0 0 1 0 0 0 1      (11H)
        ---------------      -----
        0 0 1 1 0 0 1 1      (33H)
        ---------------      -----
```

Note that, the ORed result will be stored in AX register finally.

Basic Logic Instructions- OR

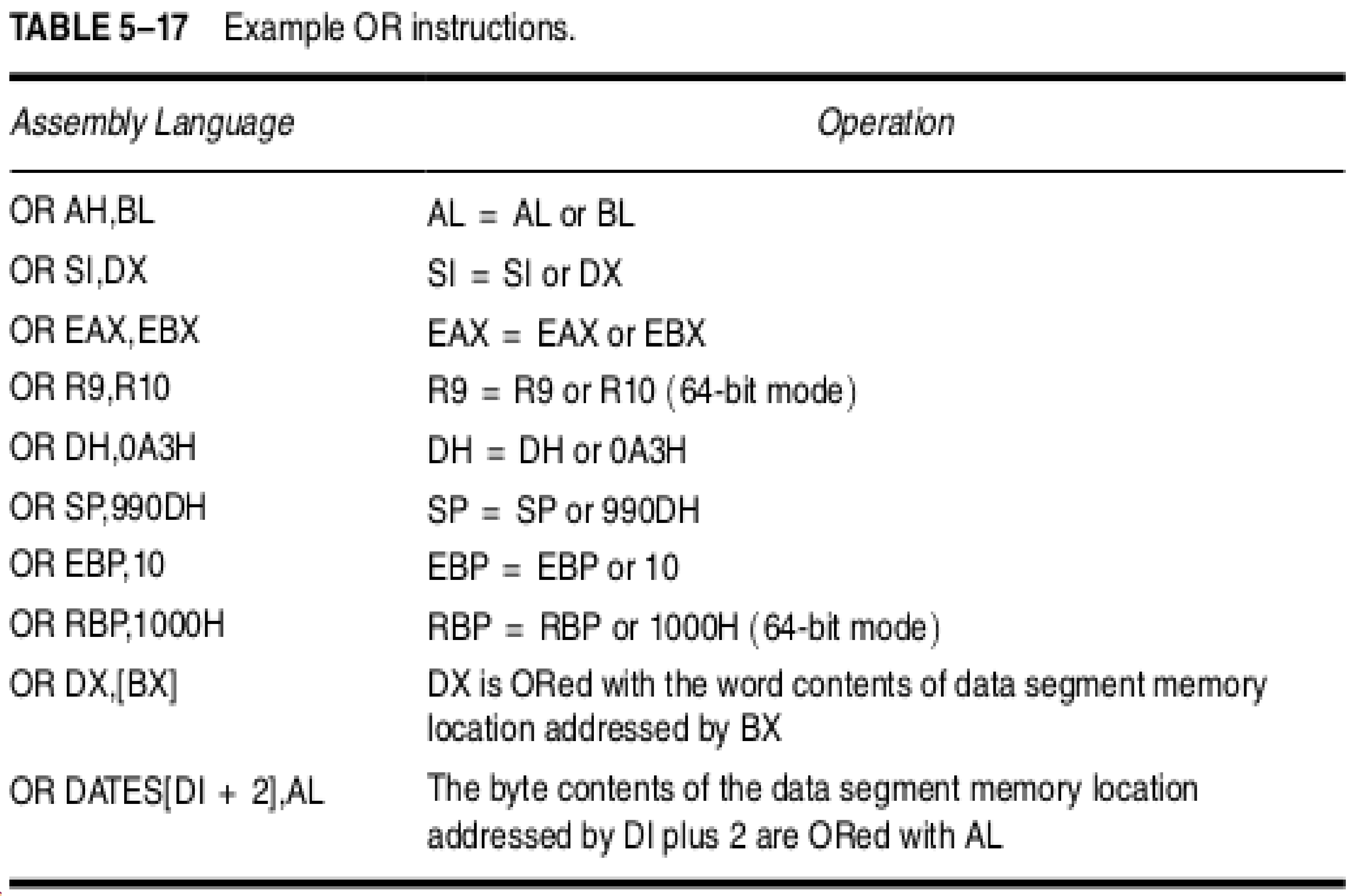


TABLE 5–17 Example OR instructions.

Assembly Language	*Operation*
OR AH,BL	AL = AL or BL
OR SI,DX	SI = SI or DX
OR EAX,EBX	EAX = EAX or EBX
OR R9,R10	R9 = R9 or R10 (64-bit mode)
OR DH,0A3H	DH = DH or 0A3H
OR SP,990DH	SP = SP or 990DH
OR EBP,10	EBP = EBP or 10
OR RBP,1000H	RBP = RBP or 1000H (64-bit mode)
OR DX,[BX]	DX is ORed with the word contents of data segment memory location addressed by BX
OR DATES[DI + 2],AL	The byte contents of the data segment memory location addressed by DI plus 2 are ORed with AL

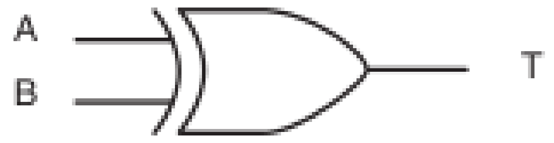
Basic Logic Instructions- Exclusive-OR

- The **Exclusive-OR** instruction (**XOR**) differs from **Inclusive-OR** (**OR**).
- The difference is that a 1,1 condition of the OR function produces 1; but, the 1,1 condition of Exclusive-OR operation produces a 0.
- Exclusive-OR function are both 0 or both 1, the output is 0. If the inputs are different, the output is 1.

FIGURE 5–7 (a) The truth table for the Exclusive-OR operation and (b) the logic symbol of an Exclusive-OR gate.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)

Basic Logic Instructions- Exclusive-OR

- Look the following example.

Example:

```
MOV  AX, 32H
MOV  BX, 11H
XOR     AX, BX
```

Implies...

	0 0 1 1 0 0 1 0	**(32H)**
XOR	**<u>0 0 0 1 0 0 0 1</u>**	**<u>(11H)</u>**
	<u>0 0 1 0 0 0 1 1</u>	**<u>(23H)</u>**

Note that, the XORed result will be stored in AX register finally.

The XOR instruction can be used to clear the content of register by XORing it with itself.

Example, MOV CX, 16H

XOR CX, CX

Implies...

	0 0 0 1 0 1 1 0	**(16H)**
XOR	**<u>0 0 0 1 0 1 1 0</u>**	**<u>(16H)</u>**
	<u>0 0 0 0 0 0 0 0</u>	**<u>(00H)</u>**

Now the content of CX is cleared.

Basic Logic Instructions- Exclusive-OR

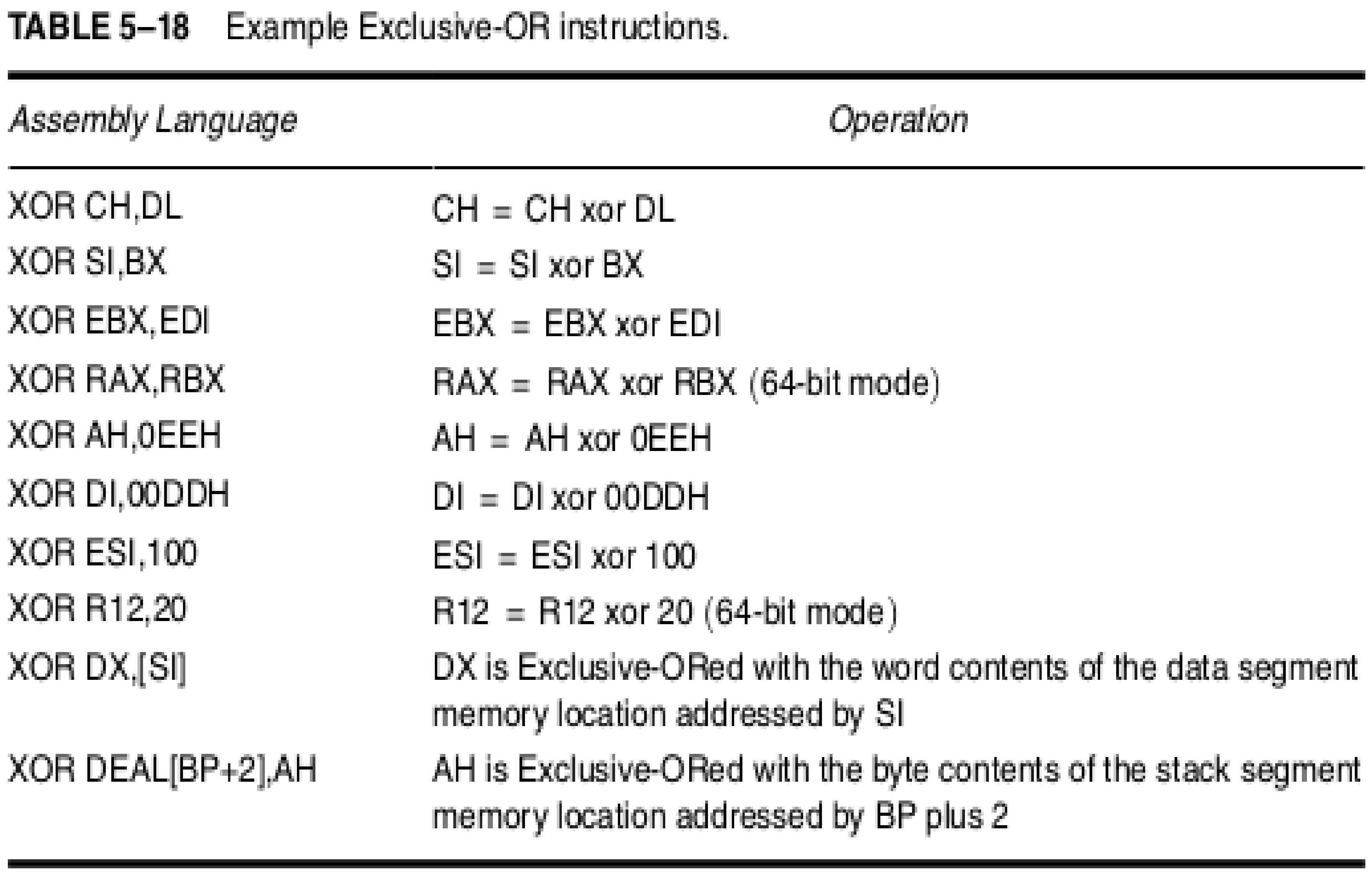


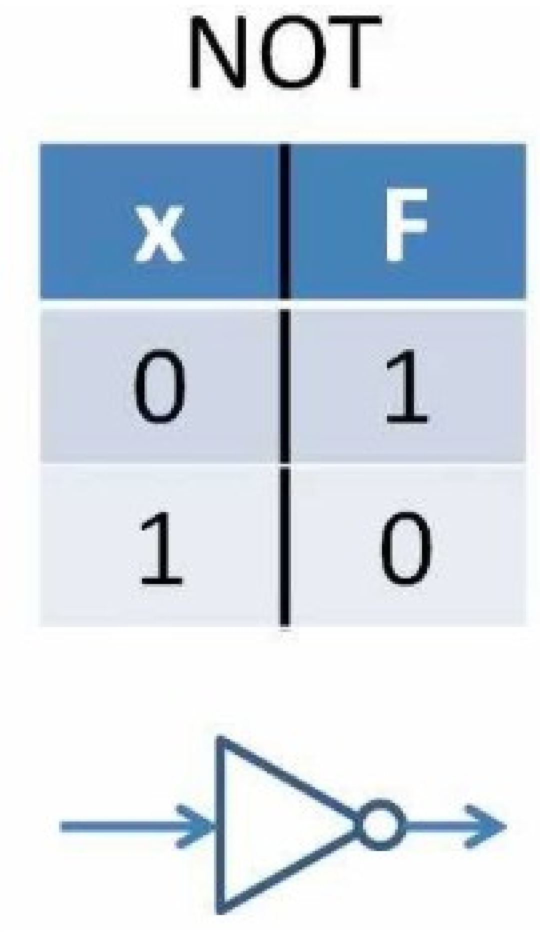
TABLE 5–18 Example Exclusive-OR instructions.

Assembly Language	*Operation*
XOR CH,DL	CH = CH xor DL
XOR SI,BX	SI = SI xor BX
XOR EBX,EDI	EBX = EBX xor EDI
XOR RAX,RBX	RAX = RAX xor RBX (64-bit mode)
XOR AH,0EEH	AH = AH xor 0EEH
XOR DI,00DDH	DI = DI xor 00DDH
XOR ESI,100	ESI = ESI xor 100
XOR R12,20	R12 = R12 xor 20 (64-bit mode)
XOR DX,[SI]	DX is Exclusive-ORed with the word contents of the data segment memory location addressed by SI
XOR DEAL[BP+2],AH	AH is Exclusive-ORed with the byte contents of the stack segment memory location addressed by BP plus 2

Basic Logic Instructions- NOT

- It performs boolean NOT operation on a single destination operand.
- It produces a 1's complement of the operands value.
- Syntax:

  NOT operand ; where the operand can be either memory or register.
- Accordingly, if the input is 1 the output is 0. and if the input is 0 the output is 1.



NOT

x	F
0	1
1	0

Basic Logic Instructions- NOT

- Look at below example;

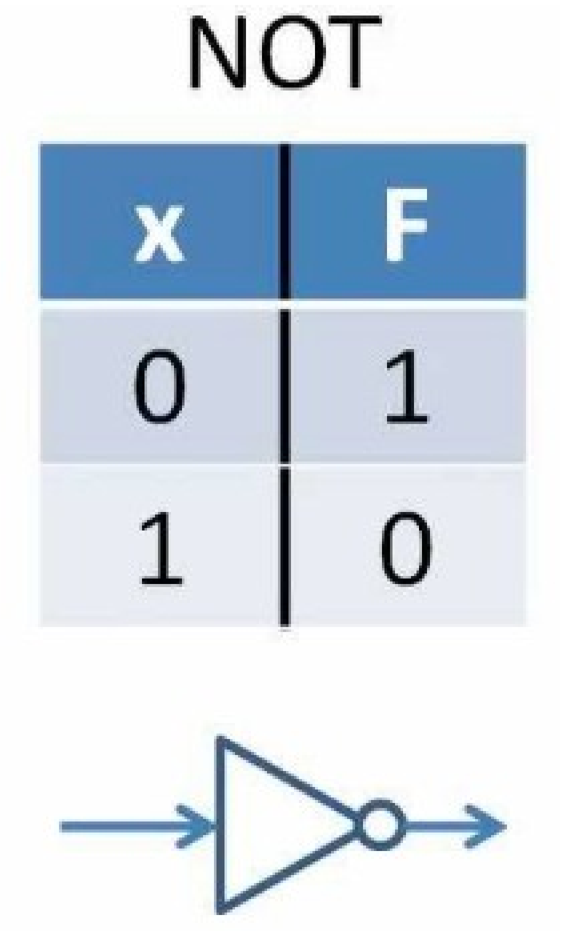
  MOV AX, 32H

  NOT AL

Implies...

NOT 0 0 1 1 0 0 1 0 (32H)
1 1 0 0 1 1 0 1 (CDH)

Now the content of AL is complemented(1's).


NOT

x	F
0	1
1	0

Basic Logic Instructions- NOT

- Note the difference when;

MOV AX, 32H

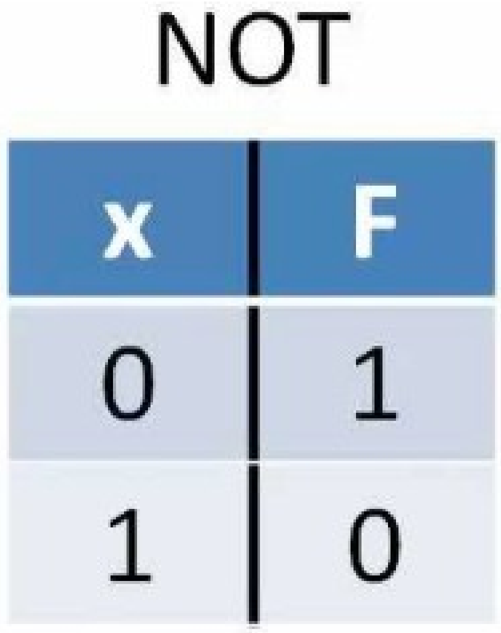
NOT AX

Implies…

NOT   0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 (0032H)

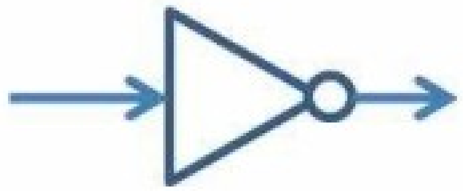
1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1   (FFCDH)

Now the content of AX is complemented(1’s).



NOT

x	F
0	1
1	0



Basic Logic Instructions- NEG

- The **NEG** instruction changes operands content into 2’s complement(negates its value).
  - The negated value will be stored in the operand itself.
- Syntax:

  NEG operand ; where the operand can be either memory or register.
- Note: the difference between NOT and NEG!
- NOT = produces 1’s complement of the operand.
- NEG = produces 2’s complement of the operand.

Basic Logic Instructions- NEG

- Example: Lets suppose the values of AX = 134H and BX = 12H. Then, what would be:

i. NEG AX

Solution: AX = 134H = $(0000\ 000100110100)_2$ Now we have to find it's 2's complement. But we have to first convert it into 1's complement. That is,

(0000 000100110100)$^{|}$ =**1111 111011001011**, then it's 2's complement would be the 1's complement value + 1.

AX = 1111 111011001011 + 1

= 1111 111011001100 = FECCH

i. NEG AH

Solution: we are given with AX = 134H = 0134H

AL=34H

AH=01H

Basic Logic Instructions- NEG

Since AH = 01H = $(00000001)_2$

NEG AH is then,

$(AH)^{|}$ = 11111110

$(AH)^{||}$ = $(AH)^{|}$ + 1 = 11111110 + 1 = 11111111 = FFH

iii. NEG BX (Exercise)

- There is a number whose 2'th complement is itself. Which number do you think it is?

Chapter 5 Continues…