Assembly Language and Microcomputer Interface

Chapter 5: Arithmetic and Logic Instructions

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology
Zhejiang University

Introduction

- We examine the arithmetic and logic instructions. The arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, increment, and decrement.
- The logic instructions include AND, OR, Exclusive-OR, NOT, shifts, rotates, and the logical compare (TEST).

Chapter Objectives

Upon completion of this chapter, you will be able to:

- Use arithmetic and logic instructions to accomplish simple binary, BCD, and ASCII arithmetic.
- Use AND, OR, and Exclusive-OR to accomplish binary bit manipulation.
- Use the shift and rotate instructions.

Chapter Objectives (*cont.*)

Upon completion of this chapter, you will be able to:

- Explain the operation of the 80386 through the Core2 exchange and add, compare and exchange, double-precision shift, bit test, and bit scan instructions.
- Check the contents of a table for a match with the string instructions.

5-1 ADDITION, SUBTRACTION AND COMPARISON

- The bulk of the arithmetic instructions found in any microprocessor include addition, subtraction, and comparison.
- Addition, subtraction, and comparison instructions are illustrated.
- Also shown are their uses in manipulating register and memory data.

Addition

- Addition (ADD) appears in many forms in the microprocessor.
- A second form of addition, called **add-with-carry**, is introduced with the ADC instruction.
- The only types of addition *not* allowed are memory-to-memory and segment register.
  - segment registers can only be moved, pushed, or popped
- Increment instruction (INC) is a special type of addition that adds 1 to a number.

Register Addition

- When arithmetic and logic instructions execute, contents of the flag register change.
  - interrupt, trap, and other flags do not change
- Any ADD instruction modifies the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags.

Immediate Addition

- Immediate addition is employed whenever constant or known data are added.

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

Memory-to-Register Addition

- Moves memory data to be added to the AL (and other) register.
- Example 5–3 shows an example that adds two consecutive bytes of data, stored at the data segment offset locations NUMB and , to the AL register.

EXAMPLE 5–3

```
0000 BF 0000 R      MOV DI,OFFSET NUMB         ;address NUMB
0003 B0 00          MOV AL,0                   ;clear sum
0005 02 05          ADD AL,[DI]                ;add NUMB
0007 02 45 01       ADD AL,[DI+1]              ;add NUMB+1
```

Array Addition

- Memory arrays are sequential lists of data.

Array Addition

- Suppose that an array of data (ARRAY) contains 10 bytes, numbered from element 0 through element 9.
- The example shows how to add the contents of array elements 3, 5, and 7 together.

```
MOV AL,0               ;clear sum
MOV SI,3               ;address element 3
ADD AL,ARRAY[SI]       ;add element 3
ADD AL,ARRAY[SI+2]     ;add element 5
ADD AL,ARRAY[SI+4]     ;add element 7
```

Array Addition

- Suppose that an array of data contains words of numbers used to form a 16-bit sum in register AX.
- A sequence of instructions shows scaled-index form addressing to add elements 3, 5, and 7 of an area of memory called ARRAY.

```
MOV EBX,OFFSET ARRAY          ;address ARRAY
MOV ECX,3                     ;address element 3
MOV AX,[EBX+2*ECX]            ;get element 3
MOV ECX,5                     ;address element 5
ADD AX,[EBX+2*ECX]            ;add element 5
MOV ECX,7                     ;address element 7
ADD AX,[EBX+2*ECX]            ;add element 7
```

Array Addition

- In this example
  - EBX is loaded with the address ARRAY
  - ECX holds the array element number
  - The scaling factor is used to multiply the contents of the ECX register by 2 to address words of data

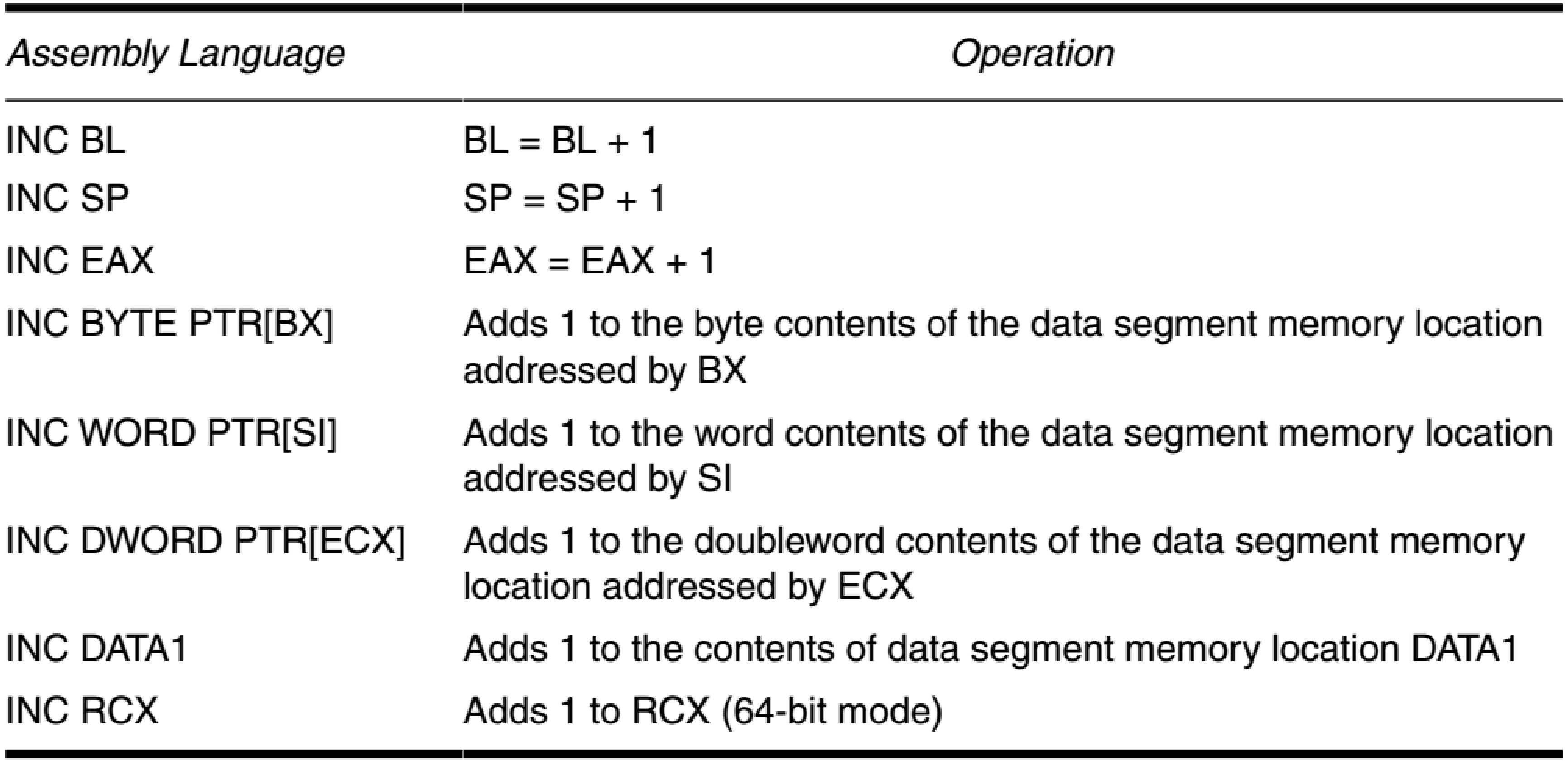
Increment Addition

- The INC instruction adds 1 to any register or memory location (except a segment register), while preserving the state of the CF flag. E.g.,

```
MOV  AX,0FFFFh
INC        ; (opcode) 40
           ; CF = 0
```

```
MOV  AX,0FFFFh
ADD   AX, 1 ; 05 01 00
             ; CF = 1
```

- With indirect memory increments, the size of the data must be described by using the BYTE PTR, WORD PTR or DWORD PTR directives.
- The assembler cannot determine if the INC [DI] is a byte-, word-, or doubleword-sized increment.



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)

examples of increment instructions

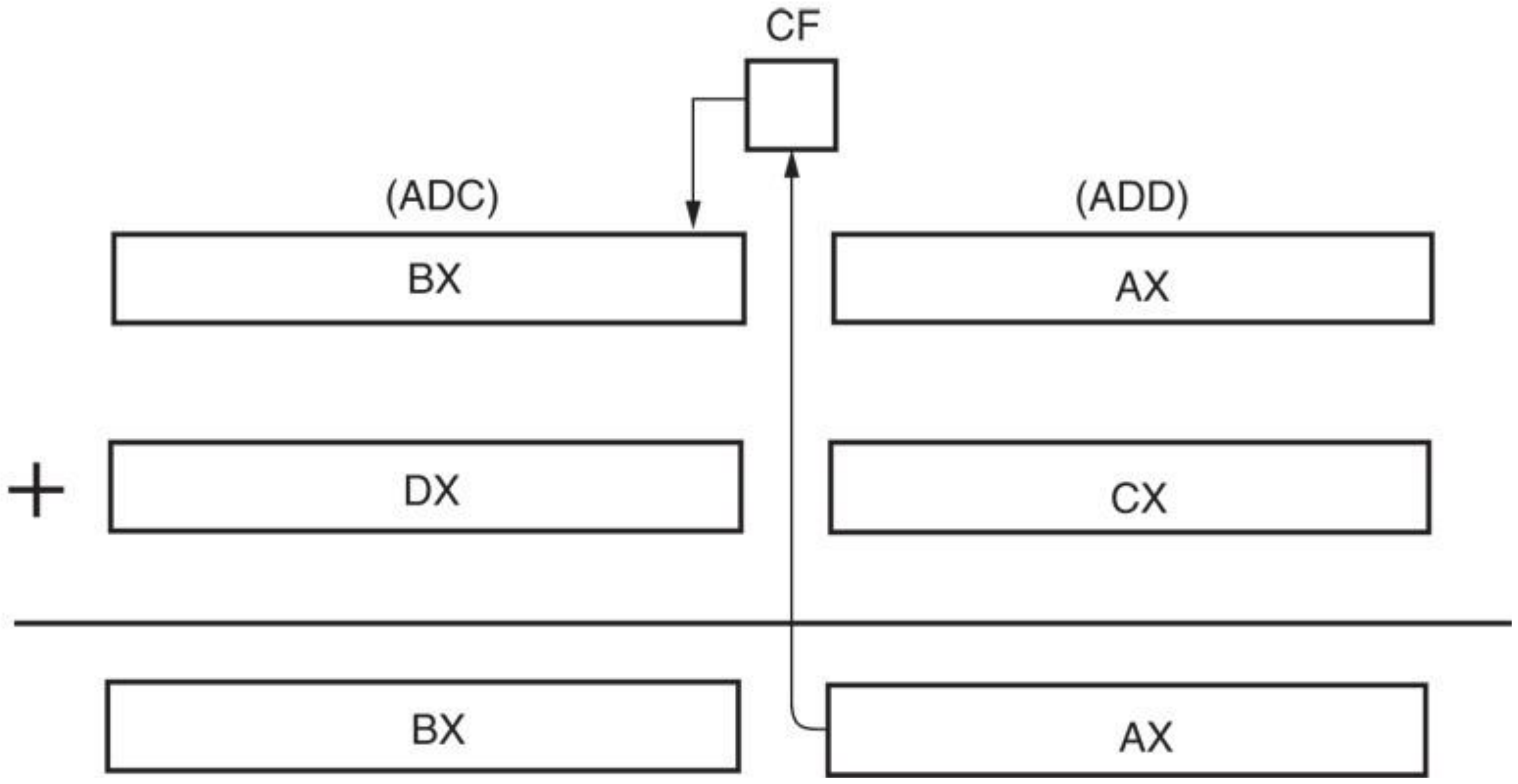
- Why do the INC instruction not affect the Carry Flag (CF)?
- INC is always used in loops where the carry flag is used for arithmetic operation (e.g., allowing to "ripple" from one loop pass to the next)
- INC allows a loop counter to be updated without disturbing the CF flag. For example

```
...
loop:  MOV   EAX, [ESI]
       ADC   [EDI], EAX        ; with carry from previous loop pass
       INC   ECX               ; adjust loop count
       LEA   ESI, [ESI+4]      ; point to next source
       LEA   EDI, [EDI+4]      ; point to next destination
       CMP   ECX,100           ; loop-bound checking
       JLE   loop
...
```

Addition-with-Carry

- ADC (Add with Carry) adds the bit in the carry flag (C) to the operand data.
  - mainly appears in software that adds numbers wider than 16 or 32 bits in the 80386–Core2
  - like ADD, ADC affects the flags after the addition
- Figure 5–1 illustrates this so placement and function of the carry flag can be understood.
  - cannot be easily performed without adding the carry flag bit because the 8086–80286 only adds 8- or 16-bit numbers

Figure 5–1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.



EXAMPLE 5–7

```
0000 03 C1          ADD AX,CX
0002 13 DA          ADC BX,DX
```

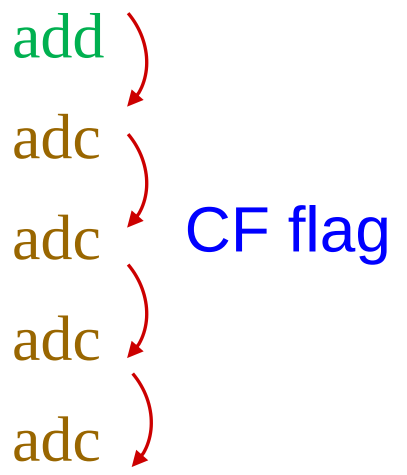
- For example, using ADC to calculate the sum of 2 very long integers that are several dwords in length.

```
.data
int1      DD      1,0,0,0,0,0,0,1
int2      DD      1,0,0,0,0,0,0,1

.code
          MOV   EDI,OFFSET int1
          MOV   ESI,OFFSET int2
          MOV   ECX,0               ; ECX = 0
          CLC                       ; start out with carry flag cleared
loop:     MOV   EAX, [ESI]
          ADC    [EDI], EAX         ; with carry from previous loop pass
          INC   ECX                 ; adjust loop count
          LEA    ESI, [ESI+4]       ; point to next source
          LEA    EDI, [EDI+4]       ; point to next destination
          CMP    ECX,8              ; loop-bound checking
          JLE    loop
```

Two ADC's Variants: adcx and adox

- ADD and ADC instructions are used to speed up large integer arithmetic with a code-sequence like this:



These instructions create a dependency chain, which makes it impossible for the processor to execute arithmetic in parallel.

- To improve upon this, Intel added a second carry chain, which allows for 2 independent carry-chains to happen simultaneously.
- The ADC instruction got two new variants:
  - ADCX and ADOX.

- Two new ADC variants do not influence each other because both have their separate carry flag.
  - ADCX uses the Carry flag as source and destination of overflow and leaves the other flags untouched.
  - ADOX uses the Overflow flag as source and destination of overflow and leaves the other flags untouched.

An example of two parallel addition of numbers

[r9][0:99] = [r8][0:99] + [r9][0:99]

```
     mov r14, 100      ; load counter
     xor   r15, r15    ; clear r15, CF and
                       ; OF flags
lbl:
     mov  rbx, [r8 + r15]  ┐
     adcx rbx, [r9 + r15]  ├ <── CF flag
     mov  [r9 + r15], rbx  ┘     dependency
```

[r11][0:99] = [r10][0:99]+ [r11][0:99]

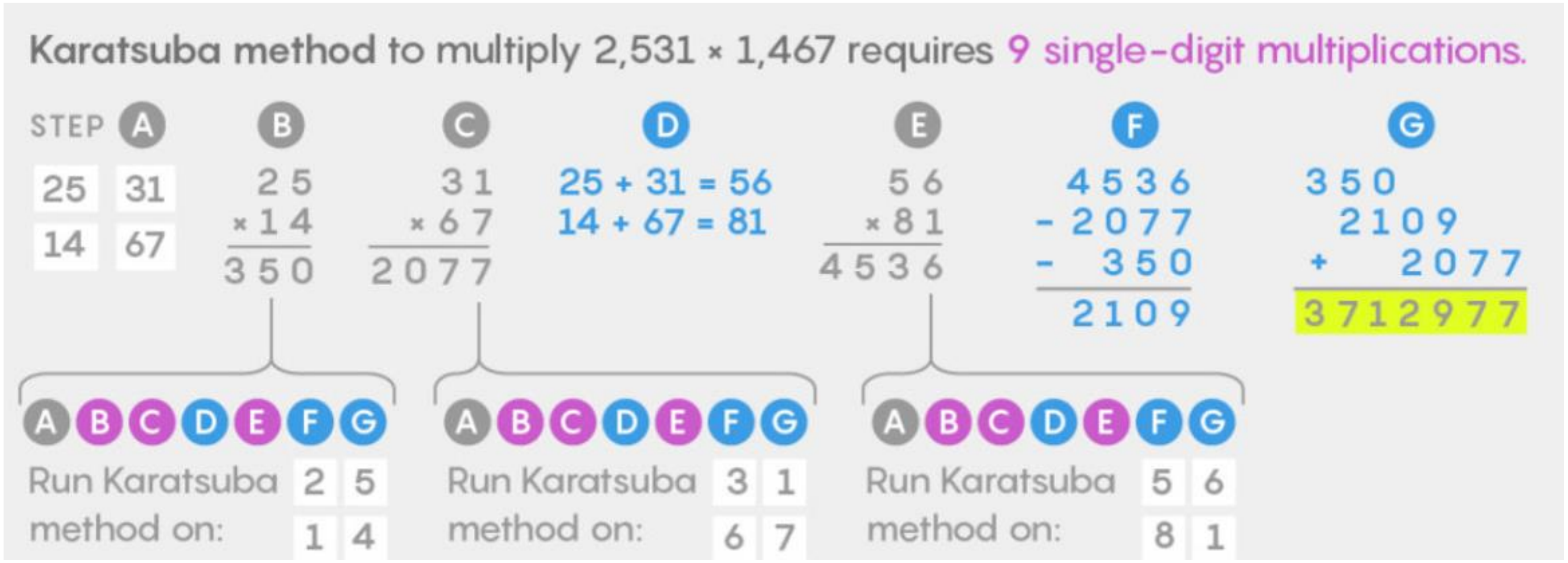
```
mov  rcx, [r10 + r15]  ┐
adox rcx, [r11 + r15]  ├ <── OF flag
mov  [r11 + r15], rcx  ┘     dependency

lea    r15, [r15 + 8]    ; addition without
                         ; effecting flags

dec    r14

jnz    lbl
```

- ADCX and ADOX instructions create a big deal for large integer multiplication.
- Large integer arithmetic has many use cases in cryptography (e.g., RSA public key algorithm) and high performance computing.



Karatsuba method to multiply 2,531 × 1,467 requires 9 single-digit multiplications.
STEP A
B
C
D
E
F
G
25 31
14 67
25 × 14 = 350
31 × 67 = 2077
25 + 31 = 56
14 + 67 = 81
56 × 81 = 4536
4536 − 2077 − 350 = 2109
350
2109
+ 2077
3712977
A B C D E F G
Run Karatsuba method on: 2 5 / 1 4
Run Karatsuba method on: 3 1 / 6 7
Run Karatsuba method on: 5 6 / 8 1

refer to: https://www.intel.cn/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf

Exchange and Add for the 80486–Core2 Processors

- XADD (exchange and add) appears in 80486 and continues through the Core2.
- The operation of XADD des, src is as follows:
  - exchanges the des operand with the src operand

    src ⇔ des
  - loads the sum of the two values into the des operand

    des = des + src
- One of the few instructions that change the source.

Exchange and Add for the 80486–Core2 Processors

- An example of XADD instruction

```
MOV  AX, 1000H
MOV  BX, 2000H   ; AX = 1000H, BX = 2000H
XADD AX, BX      ; AX = 3000H, BX = 1000H
```

- The destination operand can be a register or a memory location; the source operand is a register.

Exchange and Add for the 80486–Core2 Processors

- For multiple processor systems, XADD can be combined with the LOCK prefix in a multiprocessing system to allow multiple processors to execute one DO loop.
- int atomic_xadd (atomic_t *v, int inc)
  - XADD adds given increment "*inc*" to "**v*" and atomically returns the previous value of "*v".
  - XADD performs an atomic exchange and add operation on the atomic value "*v".
  - XADD is locked when multiple CPUs are online.
- XADD can implement shared counters and various data structures.

- XADD might be useful for optimistic locking, which is most applicable to high-volume systems.
- The following example uses optimistic locking to update a shared version by multiple threads safely.

```
.data
version DD      0                    ; shared version number initialized to 0
.           ......
.code
            MOV   ECX, [version]     ; load the current value of version
            ......                   ; working optimistically
            MOV   EAX, 1             ; EAX = 1
            XADD [version], EAX      ; [version]⇔EAX, [version] = [version]+1
            CMP   EAX, ECX           ; check if the value was modified by
                                     ; another thread
            JNE   retry              ;  if version was updated then rollback
            ......
retry:      ......                   ; handle the conflict
```

Subtraction

- Many forms of subtraction (SUB) appear in the instruction set.
  - these use any addressing mode with 8-, 16-, or 32-bit data
  - a special form of subtraction (decrement, or DEC) subtracts 1 from any register or memory location
- Numbers that are wider than 16 bits or 32 bits must occasionally be subtracted.
  - the **subtract-with-borrow instruction** (SBB) performs this type of subtraction

Register Subtraction

- After each subtraction, the microprocessor modifies the contents of the flag register.
  - flags change for most arithmetic/logic operations

Immediate Subtraction

- The microprocessor also allows immediate operands for the subtraction of constant data.

Decrement Subtraction

- The DEC instruction subtracts 1 from a register or memory location.
- The DEC instruction affects all the flag bits except CF.

- Instructions like CMP update all the flag bits as the execution results, but INC and DEC write into flag bits except CF.
- So, if JCC directly uses flag bits from INC/DEC, JCC would possibly have false dependency from unexpected instructions. For example:

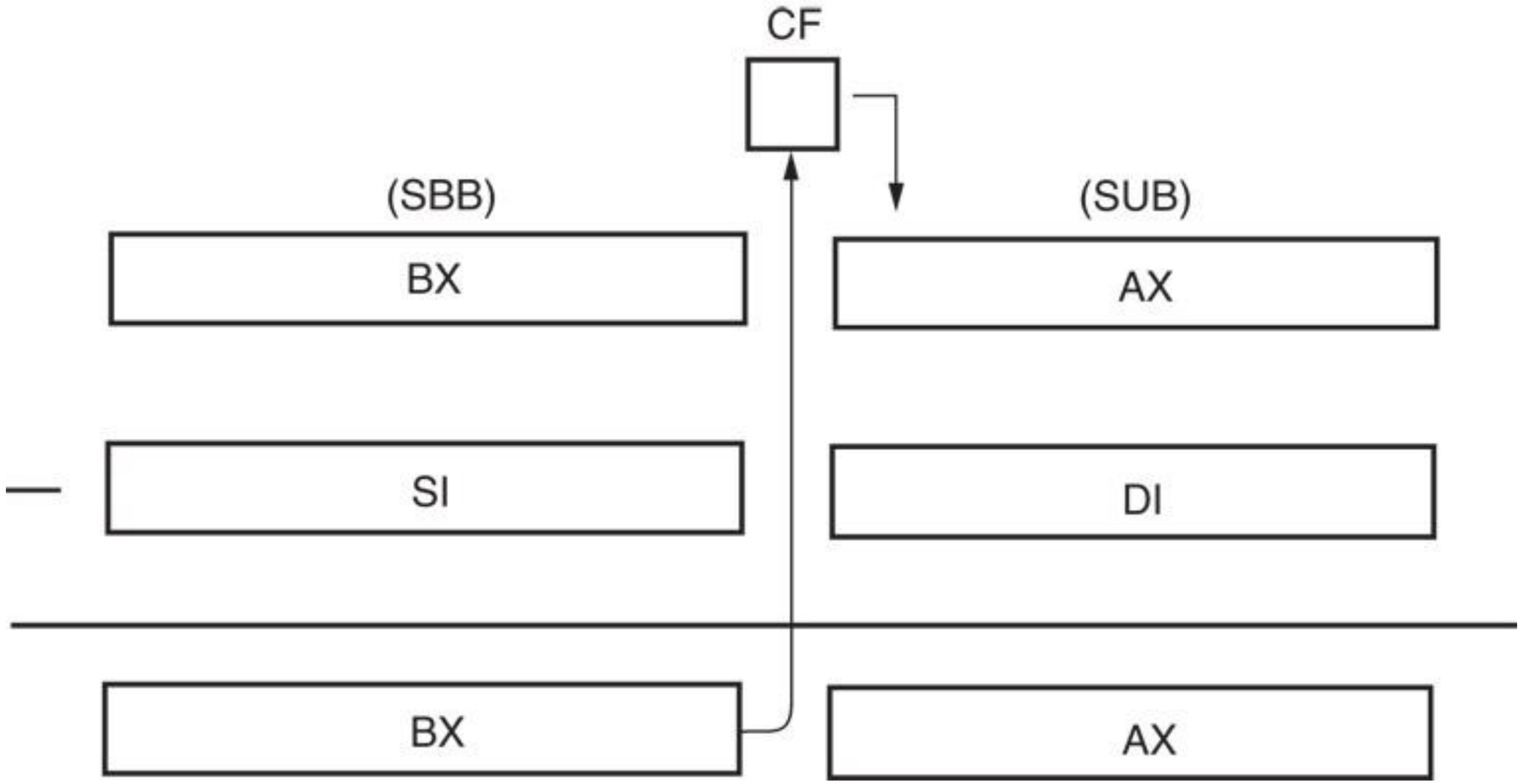
```
     CMP  EAX, EBX
     ...
                      ZF
     DEC   ECX
CF
     JBE   LABEL        ; JBE reads ZF and CF,  so there would be a
                        ; false dependency from CMP
```

- Consequently, compilers usually do not generate INC/DEC for loop count updating or reuse the INC/DEC produced FLAGS on JCC.

Subtraction-with-Borrow

- A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference.
  - most common use is subtractions wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2.
  - wide subtractions require borrows to propagate through the subtraction, just as wide additions propagate the carry

Figure 5–2 Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.



EXAMPLE 5–11

```
0000 2B C7          SUB AX,DI
0002 1B DE          SBB BX,SI
```

Comparison

- The comparison instruction (CMP) is a subtraction that changes only the flag bits.
  - destination operand never changes
- Useful for checking the 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.

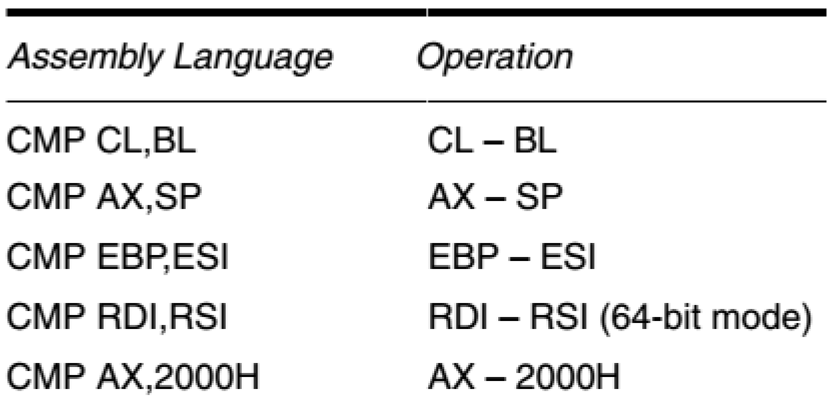


TABLE 5–7 Example comparison instructions.

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

Compare and Exchange (80486–Core2 Processors Only)

- Compare and exchange instruction (CMPXCHG) compares the destination operand with the accumulator (implicit operand), e.g., CMPXCHG CX,DX (AL/AX/EAX)
  - If des == accu, then des = src, ZF =1;
  - if des <> accu, then accu = des, ZF = 0;
  - the ZF-bit in EFLAGS gets assigned accordingly.
- found only in 80486 - Core2 instruction sets
- instruction functions with 8-, 16-, or 32-bit data

- For example, CMPXCHG CX,DX (AX)
  - if CX == AX, CX = DX, ZF =1
  - if CX <> AX, AX = CX, ZF =0
- The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared.

Case 1: before execution:

(CX)=00FFH, (DX)=00EFH, (AX)=00FFH;

after execution:

(CX)=00EFH, (DX)=00EFH, (AX)=00FFH, ZF=1;

Case 2: before execution:

(CX)=00FFH, (DX)=00EFH, (AX)=00EEH ;

after execution:

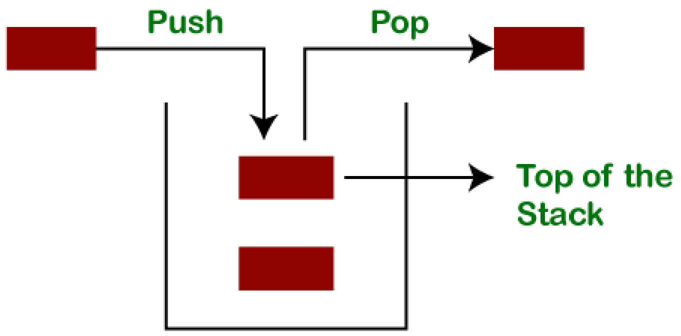
(CX)=00FFH, (DX)=00EFH, (AX)=00FFH, ZF=0;

Semantics and Behavior of CMPXCHG

- int atomic_cmpxchg (atomic_t *v, int new, int old)
  - This performs an atomic compare exchange operation on the atomic value “v”, with the given old and new values.
  - It returns the old value that the atomic variable v had just before the operation.
  - It provides explicit memory barriers around the operation.

- For example, CMPXCHG $\underbrace{\text{CX}}_{\text{atomic value}},\underbrace{\text{DX}}_{\text{new value}}\ \ \underbrace{\text{(AX)}}_{\text{old value}}$
  - if CX equals AX, DX is copied into CX;
  - if CX is not equal to AX, CX is copied into AX;
  - AX holds the value of CX before execution.

Example: a Lock-free Stack (1/3)



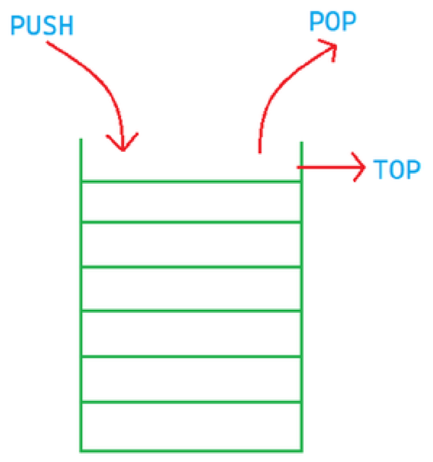
```
struct element {
    int key;
    int value;
    struct element *next;
};
struct element *top;
```

```
void push(struct element *e) {
    e→next = top;
    top = e;
}
struct element *pop(void) {
    struct element *e = top;
    top = e→next;
    return e;
}
```

sequential stack

- Sequential stack is not going to work on a concurrent system with possible race conditions.
- Spinlock and read-write lock can help to transfer lock from one holder to another. However it is expensive.

Example: a Lock-free Stack (2/3)



```
void push(struct element *e)
{
   e → next = top;
   top = e;
}
```

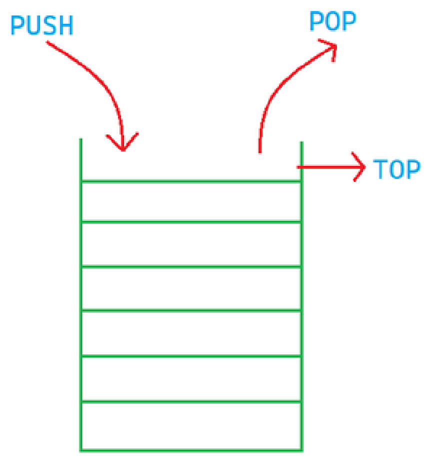
sequential stack

```
int cmpxchg(int *addr, int new, int old,) {
        int was = *addr;
        if (was == old)
                *addr = new;
        return was;
}
```

```
void push(struct element *e) {
again:
        e→next = top;
        if (cmpxchg(&top, e, e→next) !=
                                e→next)
                goto again;
}
```

concurrent stack without locks

Example: a Lock-free Stack (3/3)



```
struct element *pop(void)
{
    struct element *e = top;
    top = e→next;
    return e;
}
```

sequential stack

```
int cmpxchg(int *addr, int new, int old) {
    int was = *addr;
    if (was == old)
        *addr = new;
    return was;
}
```

```
struct element *pop(void) {
again:
        struct element *e = top;
        if (cmpxchg(&top, e→next, e) != e)
            goto again;
        return e;
}
```

concurrent stack without locks

CMPXCHG8B/CMPXCHG16B (Compare and Exchange 8/16 Bytes)

- CMPXCHG8B instruction compares the 64-bit value located in EDX:EAX with a 64-bit number located in memory.
- Syntax: CMPXCHG8B operand
  - If operand = EDX:EAX then
    - operand = ECX:EBX, ZF =1
  - else
    - EDX:EAX = operand, ZF = 0
  - The Z (zero) flag bit indicates that the values are equal after the comparison.

CMPXCHG8B/CMPXCHG16B (Compare and Exchange 8/16 Bytes)

- CMPXCHG16B compares the 128-bit value in RDX:RAX with the 128-bit number (destination operand) located in memory.
- If the values are equal, the 128-bit value in RCX:RBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into RDX:RAX.
- CMPXCHG16B requires that the destination (memory) operand should be 16-byte aligned.

5-2 MULTIPLICATION AND DIVISION

- Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions.
  - manufacturers were aware of this inadequacy, they incorporated multiplication and division into the instruction sets of newer microprocessors.
- Pentium–Core2 contains special circuitry to do multiplication in as few as one clocking period.
  - over 40 clocking periods in earlier processors

Multiplication

- Performed on bytes, words, or doublewords,
  - can be signed (IMUL) or unsigned integer (MUL)
  - for MUL, the multiplicand is always in the AL/AX/EAX register as an implicit operand
  - e.g., MUL CL   ; AX = AL*CL
- Product always a double-width product.
  - two 8-bit numbers multiplied generate a 16-bit product; two 16-bit numbers generate a 32-bit; two 32-bit numbers generate a 64-bit product
  - in 64-bit mode of Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product

8-Bit Multiplication

- With 8-bit multiplication, the multiplicand is always in the AL register (implicit operand), signed or unsigned.
  - multiplier can be any 8-bit register or memory location
  - needs directive to indicate the operand size when using memory operand, e.g. MUL WORD PTR [BX]
- Immediate multiplication is not allowed (e.g., MUL 12H), unless the two or three-operand form of IMUL multiplication.

8-Bit Multiplication

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

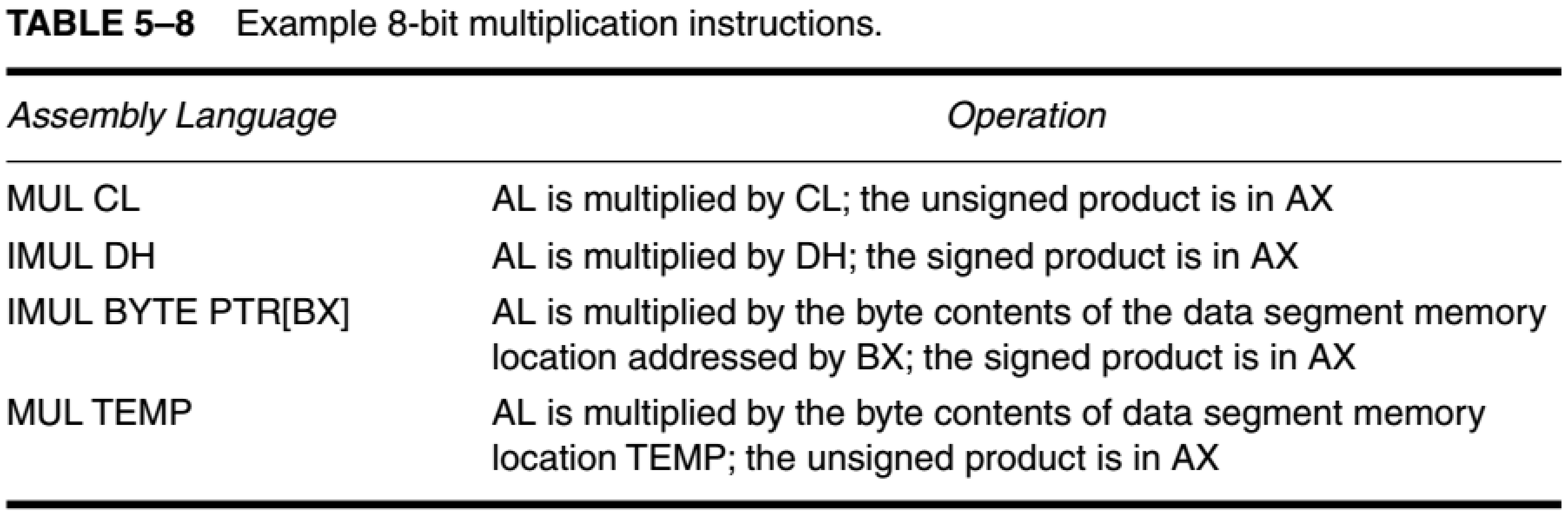


TABLE 5–8 Example 8-bit multiplication instructions.

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

16-Bit Multiplication

- Word multiplication is very similar to byte multiplication.
- AX contains the multiplicand instead of AL.
- 32-bit product appears in DX–AX:
  - DX contains the most significant 16 bits of the product;
  - AX contains the least significant 16 bits.
- As with 8-bit multiplication, the choice of the multiplier is up to the programmer.

32-Bit Multiplication

- In 80386 and above, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers.
  - can be signed or unsigned by using IMUL and MUL instructions
- Contents of EAX are multiplied by the operand specified with the instruction.
- The 64 bit product is found in EDX–EAX, where EAX contains the least significant 32 bits of the product.

64-Bit Multiplication

- In the Pentium 4, the result of a 64-bit multiplication appears in the RDX:RAX register pair as a 128-bit product.
- Although multiplication of this size is relatively rare, the Pentium 4 and Core2 can perform it on both signed and unsigned numbers.

IMUL—Signed Multiply

- IMUL instruction has three forms:
  - one-operand form: this form is identical to that used by the MUL instruction.
  - two-operand form: the destination operand is a register and the source operand is an immediate value, a register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.

    ```
    e.g., IMUL ECX, [EAX+4]     ; ECX = ECX * [EAX+4]
          IMUL ECX, 16          ; ECX = ECX * 16
    ```
  - three-operand form: the first source operand is multiplied by the second source operand. The intermediate product is truncated and stored in the destination operand.

    ```
    e.g., IMUL ECX, [EAX+4], 5  ; ECX = [EAX+4] * 5
    ```

A Special Immediate Multiplication

- In two-operand or three-operand form of IMUL instruction, the source operand can be an immediate value.
- When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.
- For example, the IMUL CX,DX,12H instruction multiplies 12H times DX and leaves a 16-bit signed product in CX.
- Note that a two-operand form is assembled to an three-operand to support immediate multiplication, e.g., IMUL BX,16H → IMUL BX,BX,16H.

Difference between MUL and IMUL

- The MUL instruction fills the upper part with zero-extension, while the IMUL instruction fills the upper part with sign-extension, e.g.,

```
MOV   AL, 48
MOV   BL, 2
MUL   BL                 AH           AL
; AX = 0060h    (00000000 01100000)
                         ⇩
                   zero-extension
```

```
MOV  AL, 48
MOV  BL, -2
IMUL  BL              AH           AL
; AX = FFA0h (11111111 10100000)
                      ⇩
                sign-extension
```

FLAGS Affected by MUL

- When the product fits completely within the lower register of the product, the MUL instruction clears OF and CF flags, otherwise, OF and CF flags are set, e.g.,

```
MOV   AL, 48
MOV   BL, 3
MUL   BL              AH          AL
; AX = 0090h    (00000000 10010000)

                 CF=0,    OF=0
```

```
MOV  AL, 48
MOV  BL, 8
MUL  BL               AH          AL
; AX = 0180h (00000001 10000000)

                 CF=1,   OF=1
```

- The CF and OF flags indicate whether or not the upper half of the product contains significant digits.

FLAGS Affected by IMUL

- When the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, the CF and OF flags are set; otherwise the CF and OF flags are cleared.

```
MOV   AL, 48
MOV   BL, 3
IMUL  BL
; AX = 0090h
```

sign bit

AH AL

(00000000 10010000)

CF=**1**, OF=**1**

```
MOV  AL, 48
MOV  BL, 2
IMUL BL
; AX = 0060h
```

sign bit

AH AL

(00000000 01100000)

CF=**0**, OF=**0**

- With the two and three-operand forms, because of the truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

Division

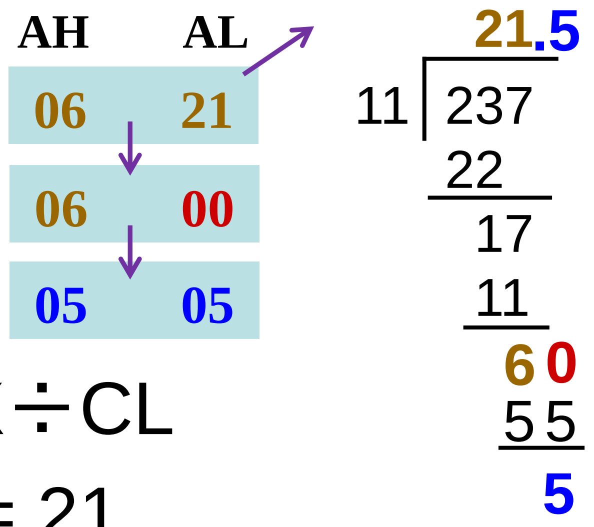
- Occurs on 8- or 16-bit and 32-bit numbers depending on microprocessor.
  - signed (IDIV) or unsigned (DIV) integers
- Dividend is always a double-width dividend, divided by the operand.
- There is no immediate division instruction available to any microprocessor.
- In 64-bit mode Pentium 4 & Core2, divide a 128-bit number by a 64-bit number.

- A division can result in two types of errors:
  - attempt to divide by zero
  - other is a divide overflow, which occurs when a small number divides into a large number
- 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.

8-Bit Division

- Uses AX to store the dividend divided by the contents of any 8-bit register or memory location, namely divide AX by r/m8, with result stored in AL := quotient, AH := remainder
- Quotient moves into AL after the division with AH containing a whole number remainder.
- For example,
  - MOV AX, 237 ; AX = 237
  - MOV CL, 11 ; CL = 11
  - DIV CL ; AH: AL = AX÷CL

  ; AH = 6, AL = 21



- Quotient is positive or negative; remainder always assumes sign of the dividend. This rounding mode is called round-toward-zero.
- For example, IDIV BL $\nearrow$ $16 \div (-3) = -5.\dot{3} \begin{cases} -5 \text{ R } 1 \\ -6 \text{ R } -2 \end{cases}$
  - for AX=10H (+16) and BL=0FDH (-3)
    - result: quotient of -5 (AL), remainder 1 (AH)
  - for AX=0FFF0H (-16) and BL=03H (+3)
    - result: quotient of -5 (AL), remainder -1 (AH)
- Numbers usually 8 bits wide in 8-bit division.
  - the dividend must be converted to a 16-bit wide number in AX ; accomplished differently for signed and unsigned numbers

- The following example illustrates how to divide the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1.
- Note that the contents of location NUMB is zero-extended to form a 16-bit unsigned number for the dividend.

```
MOV   AL,NUMB          ;get NUMB
MOV   AH,0             ;zero-extend
DIV   NUMB1            ;divide by NUMB1
MOV   ANSQ,AL          ;save quotient
MOV   ANSR,AH          ;save remainder
```

16-Bit Division

- Sixteen-bit division is similar to 8-bit division
  - instead of dividing into AX, the 16-bit number is divided into DX–AX, a 32-bit dividend
- As with 8-bit division, numbers must often be converted to the proper form for the dividend.
  - if a 16-bit unsigned number is placed in AX, DX must be cleared to zero
- In the 80386 and above, the number is zero-extended by using the MOVZX instruction.

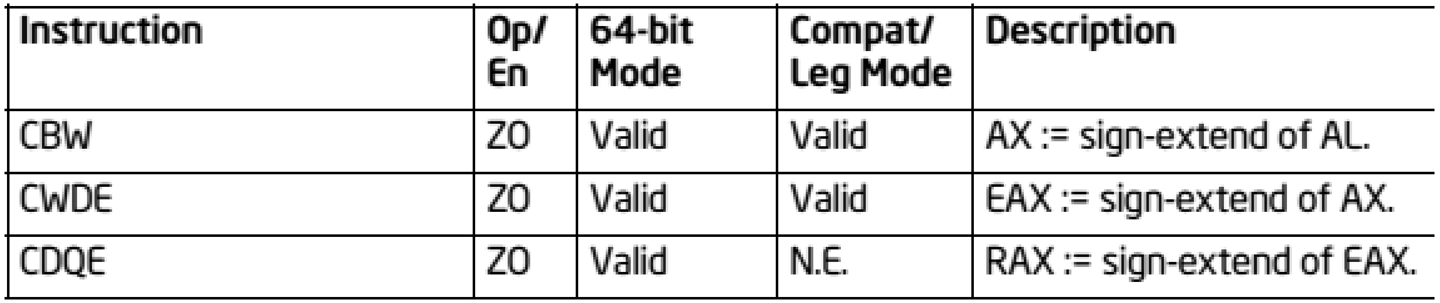
32-Bit Division

- 80386 - Pentium 4 perform 32-bit division on signed or unsigned numbers.
  - 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.

64-Bit Division

- Pentium 4 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.
- The quotient is found in RAX and the remainder is in RDX after the division.

- Three types of instructions to perform signed extension:
  - CBW/CWDE/CDQE is to convert a signed byte, word, or doubleword in the AL, AX or EAX register into a signed word, doubleword, or quadword in the RAX register.



Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
CBW	ZO	Valid	Valid	AX := sign-extend of AL.
CWDE	ZO	Valid	Valid	EAX := sign-extend of AX.
CDQE	ZO	Valid	N.E.	RAX := sign-extend of EAX.

  - CWD/CDQ/CQO copy the sign bit in the rAX register to all bits of the rDX register.

```
MOV   AX,-100         ;load a -100
MOV   CX,9            ;load +9
CWD                   ; convert the signed 16-bit number in AX
IDIV   CX             ; to a 32-bit signed number in DX: AX
```

- The above example shows the division of two 16-bit signed numbers by CWD instruction.

– MOVSX/MOVSXD copy the contents of the source operand (register or memory location) to the destination operand (register) by sign extension.

MOVSX *reg16, reg/mem8*	MOVSX *reg32, reg/mem16*
MOVSX *reg32, reg/mem8*	MOVSX *reg64, reg/mem16*
MOVSX *reg64, reg/mem8*	MOVSXD *reg64, reg/mem32*

- MOVZX copies the value in a register or memory location (second operand) into a register (first operand), zero extending the value to fit in the destination register.

MOVZX *reg16, reg/mem8*	MOVZX *reg32, reg/mem16*
MOVZX *reg32, reg/mem8*	MOVZX *reg64, reg/mem16*
MOVZX *reg64, reg/mem8*	

The Remainder

- There are a few possible choices to do with the remainder after a division:
  - dropped to truncate the quotient
  - round the quotient: If division is unsigned, rounding requires the remainder be compared with half the divisor to decide whether to round up the quotient.
  - fractional remainder : the remainder could also be converted to a fractional remainder.

- Example 5–16 shows a program that divides AX by BL and rounds the unsigned result. This program doubles the remainder before comparing it with BL to decide whether to round the quotient. Here, an INC instruction rounds the contents of AL after the comparison.

EXAMPLE 5–16 round the quotient

```
0000 F6 F3                DIV   BL          ;divide
0002 02 E4                ADD   AH,AH       ;double remainder
0004 3A E3                CMP   AH,BL       ;test for rounding
0006 72 02                JB    NEXT        ;if OK
0008 FE C0                INC   AL          ;round
000A               NEXT:
```

- Example 5–17 shows how 13 is divided by 2. The 8-bit quotient is saved in memory location ANSQ, and then AL is cleared. Next, the contents of AX are again divided by 2 to generate a fractional remainder.

EXAMPLE 5–17 fractional remainder

```
0000 B8 000D            MOV  AX,13         ;load 13
0003 B3 02              MOV  BL,2          ;load 2
0005 F6 F3              DIV  BL            ;13/2
0007 A2 0003 R          MOV  ANSQ,AL       ;save quotient
000A B0 00              MOV  AL,0          ;clear AL
000C F6 F3              DIV  BL            ;generate remainder
000E A2 0004 R          MOV  ANSR,AL       ;save remainder
```

- After second division, the AL register equals 80H. If the binary point (radix) is placed before the leftmost bit of AL, the fractional remainder in AL is 0.5_{10} or 0.10000000_2 . The remainder is saved in memory location ANSR.

5-3 BCD and ASCII Arithmetic

- The microprocessor allows arithmetic manipulation of both BCD (**binary-coded decimal**) and ASCII (**American Standard Code for Information Interchange**) data.
- These instructions are not valid in 64-bit mode. Using them in 64-bit will generates an invalid-opcode (#UD) exception.

BCD Arithmetic

- BCD operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic.
- Two arithmetic techniques operate with BCD data: addition and subtraction.
- DAA (**decimal adjust after addition**) instruction follows BCD addition,
- DAS (**decimal adjust after subtraction**) follows BCD subtraction.
  - both correct the result of addition or subtraction so it is a BCD number
- These instructions use register AX as the source and as the destination.

DAA Instruction

- DAA adjusts the sum of two packed BCD values to create a packed BCD result.
- The DAA instruction is only useful when it follows an ADD or ADC instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register.
- The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result.
- If a decimal carry is detected, the CF and AF flags are set accordingly.
  - Auxiliary carry holds the carry (half-carry) after addition.

DAA Instruction Examples

- AL is the implied source and destination operand.
- Example 1: calculate BCD 35+48

Auxiliary carry (half-carry)

```
MOV AL,  35H
ADD AL,   48H          ;  AL = 7DH, AF = 0
DAA                    ;  AL = 83H,  CF = 0
```

- Example 2: calculate BCD 69+29

```
MOV AL,  69H
ADD AL,   29H          ;  AL = 92H,  AF = 1
DAA                    ;  AL = 98H,  CF = 0
```

- Example 3: calculate BCD 35+65

```
MOV AL,  35H
ADD AL,   65H          ;  AL = 9AH,  AF = 0
DAA                    ;  AL = 00H,  CF = 1
```

Example 5–18 provides a sample program that adds the BCD numbers in DX and BX, and stores the result in CX.

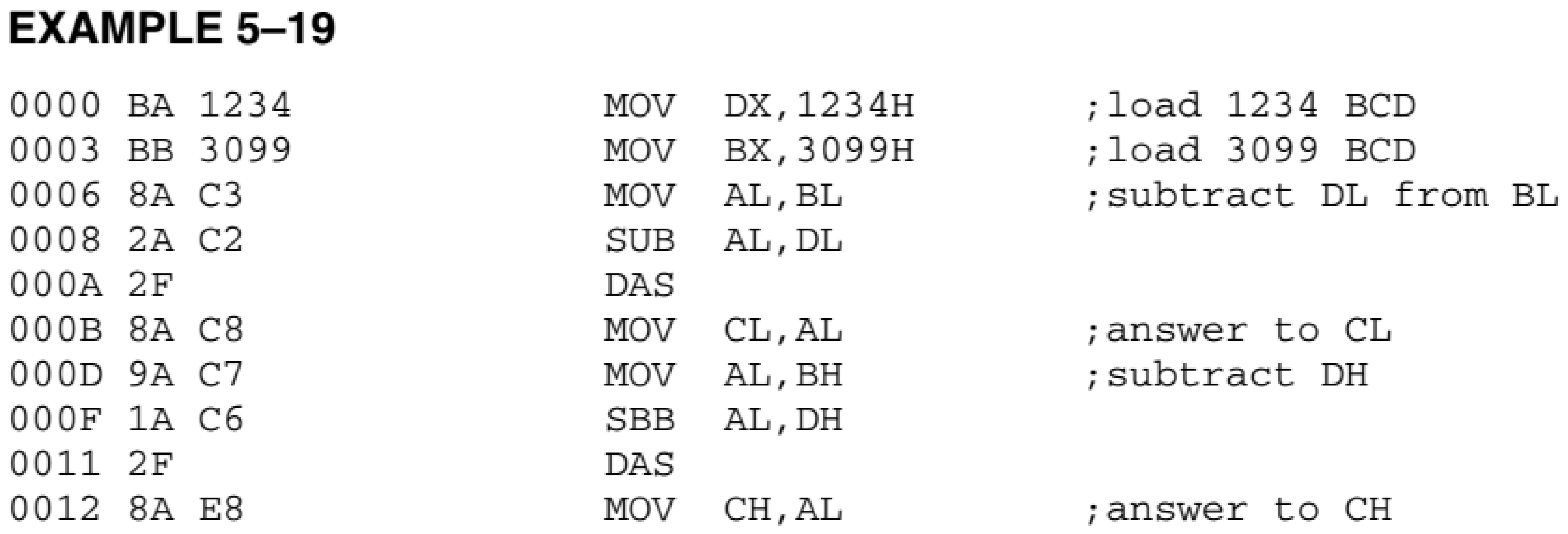
EXAMPLE 5–18

```
0000 BA 1234          MOV  DX,1234H      ;load 1234 BCD
0003 BB 3099          MOV  BX,3099H      ;load 3099 BCD
0006 8A C3            MOV  AL,BL         ;sum BL and DL
0008 02 C2            ADD  AL,DL         ;AL = CDH
000A 27               DAA                ;AL = 33H,  CF=1
000B 8A C8            MOV  CL,AL         ;answer to CL
000D 9A C7            MOV  AL,BH         ;sum BH, DH and carry
000F 12 C6            ADC  AL,DH         ;AL = 43H
0011 27               DAA
0012 8A E8            MOV  CH,AL         ;answer to CH
                                         ;CX = 4333H
```

```
      DH: DL    12 34  H
+     BH: BL    30 99  H
-------------------------
=     CH: CL    43 33  H
```

DAS Instruction

- Functions as does DAA instruction, except it follows a subtraction instead of an addition.
- Example 5-19 is the same as Example 5–18, except that it subtracts instead of adds DX and BX.



EXAMPLE 5–19

```
0000 BA 1234            MOV  DX,1234H      ;load 1234 BCD
0003 BB 3099            MOV  BX,3099H      ;load 3099 BCD
0006 8A C3              MOV  AL,BL         ;subtract DL from BL
0008 2A C2              SUB  AL,DL
000A 2F                 DAS
000B 8A C8              MOV  CL,AL         ;answer to CL
000D 9A C7              MOV  AL,BH         ;subtract DH
000F 1A C6              SBB  AL,DH
0011 2F                 DAS
0012 8A E8              MOV  CH,AL         ;answer to CH
```

ASCII Arithmetic

- ASCII arithmetic instructions function with coded numbers, value 30H to 39H for 0–9.
- Four instructions in ASCII arithmetic operations:
  - AAA (**ASCII adjust after addition**)
  - AAS (**ASCII adjust after subtraction**)
  - AAM (**ASCII adjust after multiplication**)
  - AAD (**ASCII adjust before division**)
- These instructions use register AX as the source and as the destination.

AAA Instruction

- Addition of two one-digit ASCII-coded numbers will not result in any useful data.
- AAA instruction adjusts the value in AL register to an unpacked BCD result.
- The following example shows the way ASCII addition functions.

```
MOV   AX, '1'       ;load ASCII 1
ADD   AL,  '9'      ;add ASCII 9          AX=6A
AAA                 ;adjust sum           AX=0100H
ADD   AX, 3030h     ;answer to ASCII      AX=3130H (10)
```

AAS Instruction

- AAS adjusts the AX register after an ASCII subtraction.

AAM Instruction

- Follows multiplication instruction after multiplying two one-digit unpacked BCD numbers.
- AAM converts from binary to unpacked BCD.
- If a binary number between 0000H and 0063H appears in AX, AAM converts it to BCD.

AAD Instruction

- Appears before a division.
- The AAD instruction requires the AX register contain a two-digit unpacked BCD number (not ASCII) before executing.

5-4 BASIC LOGIC INSTRUCTIONS

- Include AND, OR, Exclusive-OR, and NOT.
  - also TEST, a special form of the AND instruction
  - NEG, similar to the NOT instruction
- Logic operations provide binary bit control in low-level software.
  - allow bits to be set, cleared, or complemented
- Low-level software appears in machine language or assembly language form and often controls the I/O devices in a system.

- All logic instructions affect the flag bits except NOT instruction with flags unchanged.
- Logic operations always:
  - clear the OF and CF flags
  - change the SF, ZF, and PF flags to reflect the result
  - the state of the AF flag is undefined
- When binary data are manipulated in a register or a memory location, the rightmost bit position is always numbered bit 0.

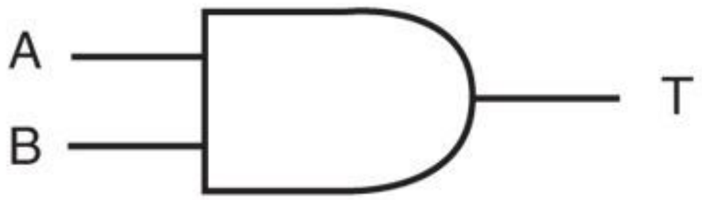
AND

- Performs logical multiplication, illustrated by a truth table.
- AND can replace discrete AND gates if the speed required is not too great
  - normally reserved for embedded control applications
- In 8086, the AND instruction often executes in about a microsecond.
  - with newer versions, the execution speed is greatly increased

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)

- AND clears bits of a binary number.
  - called **masking**
- AND uses any mode except memory-to-memory and segment register addressing.
- An ASCII-coded number can be converted to BCD by using AND to mask off the leftmost four binary bit positions. This converts the ASCII 30H to 39H to 0–9.

Figure 5–4 The operation of the AND function showing how bits of a number are cleared to zero.

```
  x x x x  x x x x   Unknown number
• 0 0 0 0  1 1 1 1   Mask
 ─────────────────
  0 0 0 0  x x x x   Result
```

- Example 5–25 shows a short program that converts the ASCII contents of BX into BCD.

EXAMPLE 5–25

```
MOV BX,3135H          ;load ASCII
AND BX,0F0FH          ;mask BX
```

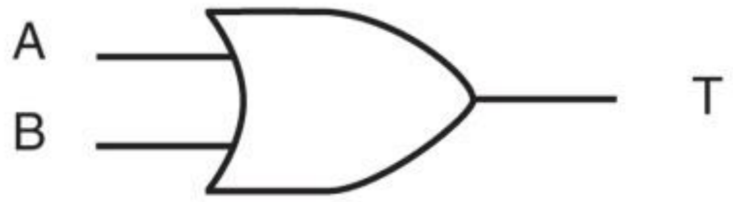
OR

- Performs logical addition
  - often called the *Inclusive-OR* function
- The OR function generates a logic 1 output if any inputs are 1.
  - a 0 appears at output only when all inputs are 0
- Figure 5–6 shows how the OR gate sets (1) any bit of a binary number.
- The OR instruction uses any addressing mode except segment register addressing.

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)

Figure 5–6 The operation of the OR function showing how bits of a number are set to one.

```
   x x x x  x x x x   Unknown number
+  0 0 0 0  1 1 1 1   Mask
--------------------
   x x x x  1 1 1 1   Result
```

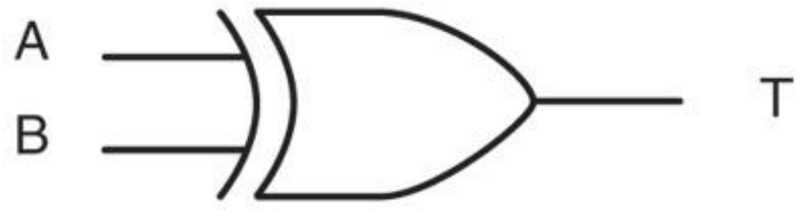
Exclusive-OR

- Differs from Inclusive-OR (OR) in that the 1,1 condition of Exclusive-OR produces a 0.
  - a 1,1 condition of the OR function produces a 1
- The Exclusive-OR operation *excludes* this condition; the Inclusive-OR *includes* it.
- If inputs of the Exclusive-OR function are both 0 or both 1, the output is 0; if the inputs are different, the output is 1.
- Exclusive-OR is sometimes called a comparator.

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)

- XOR uses any addressing mode except segment register addressing.
- Exclusive-OR is useful if some bits of a register or memory location must be inverted.
- Figure 5–8 shows how just part of an unknown quantity can be inverted by XOR.
  - when a 1 Exclusive-ORs with X, the result is $\overline{X}$
  - if a 0 Exclusive-ORs with X, the result is X
- A common use for the Exclusive-OR instruction is to clear a register to zero

Figure 5–8 The operation of the Exclusive-OR function showing how bits of a number are inverted.

$$\begin{array}{rl} x\,x\,x\,x\;\;x\,x\,x\,x & \text{Unknown number} \\ \oplus\,0\,0\,0\,0\;\;1\,1\,1\,1 & \text{Mask} \\ \hline x\,x\,x\,x\;\;\overline{x}\,\overline{x}\,\overline{x}\,\overline{x} & \text{Result} \end{array}$$

Test and Bit Test Instructions

- **TEST** performs the AND operation.
  - only affects the condition of the flag register, which indicates the result of the test
  - functions the same manner as a CMP
- Usually the followed by either the JZ (jump if zero) or JNZ (jump if not zero) instruction.
- The destination operand is normally tested against immediate data.

- Example 5–28 lists a short program that tests the rightmost and leftmost bit positions of the AL register. Here, 1 selects the rightmost bit and 128 selects the leftmost bit.

EXAMPLE 5–28

tests the rightmost and leftmost bit positions of the AL

```
TEST AL,1                  ;test right bit
JNZ  RIGHT                 ;if set
TEST AL,128                ;test left bit
JNZ  LEFT                  ;if set
```

- CMP and TEST are the instructions that are commonly used for comparison, and these instructions are known as conditionals, e.g.,

```
MOV  EAX, 1                                MOV  EAX, 1
CMP  EAX, 1  ; C=0,Z=1,S=0,O=0             TEST  EAX, 1  ; C=0,Z=0,S=0,O=0
JE       LABEL ; jump to LABEL             JE       LABEL ; do not jump
```

- TEST same,same is used to determine if signed numbers are greater than zero, e.g.,

```
TEST  EAX, EAX   ; if EAX = 0 set Z = 1, if EAX < 0 set S = 1
JLE    ERROR       ; if EAX is equal or less than zero then jump
```

- TEST EAX,EAX is almost identical to CMP EAX, 0, except that it is shorter than CMP.

```
TEST  EAX, EAX ; 85 C0                      CMP  EAX, 0 ; 83 F8 00
```

- 80386 - Pentium 4 contain four additional test instructions that test single bit positions.
  – all bit test instructions test the bit position in the destination operand selected by the source operand.

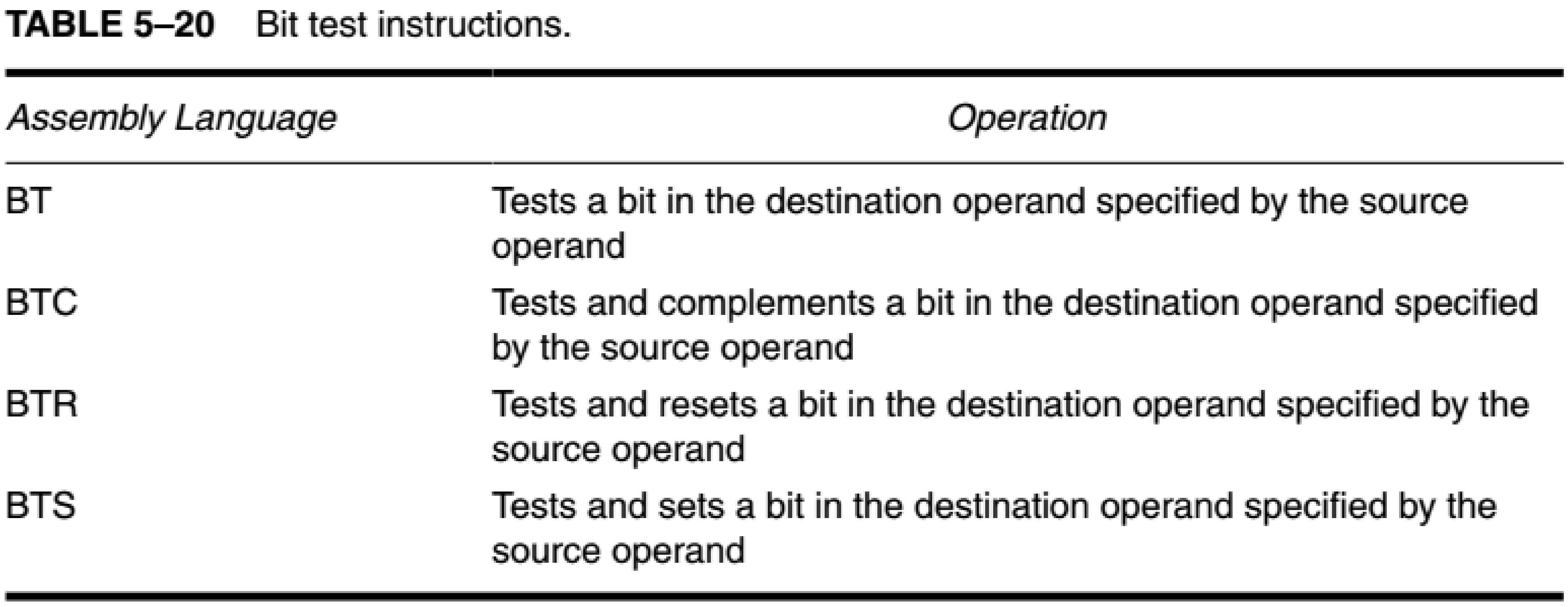


TABLE 5–20 Bit test instructions.

Assembly Language	*Operation*
BT	Tests a bit in the destination operand specified by the source operand
BTC	Tests and complements a bit in the destination operand specified by the source operand
BTR	Tests and resets a bit in the destination operand specified by the source operand
BTS	Tests and sets a bit in the destination operand specified by the source operand

- For example, the BT AX,4 instruction tests bit position 4 in AX.
  - the result of the test is located in the carry flag bit;
  - If bit position 4 is a 1, carry is set;
  - if bit position 4 is a 0, carry is cleared.

```
OR    CX,0600H        ;set bits 9 and 10
AND   CX,0FFFCH       ;clear bits 0 and 1
XOR   CX,1000H        ;invert bit 12
```

```
BTS   CX,9            ;set bit 9
BTS   CX,10           ;set bit 10
BTR   CX,0            ;clear bit 0
BTR   CX,1            ;clear bit 1
BTC   CX,12           ;complement bit 12
```

NOT and NEG

- NOT and NEG can use any addressing mode except segment register addressing.
- The NOT instruction inverts all bits of a byte, word, or doubleword.
- NEG two’s complements a number.
- The NOT function is considered logical, NEG function is considered an arithmetic operation.
- The NOT instruction does not affect any flag bits, while the NEG instruction affects flag bits as follows:
  - if the operand = 0, then CF = 0, otherwise CF = 1.
  - the OF, SF, ZF, AF, and PF flags are set according to the result.

- An interesting example of signum function:

```
int signum (int x)
{
    if (x > 0)
        return 1;
    else if (x < 0)
        return -1;
    else
        return 0;
}
```

```
        test    edi, edi
        jle     LABEL    ; edi <=0
        mov     eax, 1   ; eax = 1
        ret
LABEL:
        sar     edi, 31    ; sign-
                           ; extension
        mov     eax, edi
        ret
```

x86-64 ICC –O3

```
cwd                ; dx : ax ← ax
neg   ax           ; if (ax) CF = 1
adc   dx, dx       ; dx = dx + dx + CF
```

minimum code of signum

```
int signum (int x)
{
   if (x > 0)
         return 1;
   else if (x < 0)
         return -1;
   else
         return 0;
}
```

```
cwd                 ; dx : ax ← ax
neg   ax            ; if (ax) CF = 1
adc   dx, dx        ; dx = dx + dx + CF
```

minimum code of signum

if ax = 0
    dx = dx + dx + CF = 0+0+0 =0
if ax > 0
    dx = dx + dx + CF = 0+0+1 =1
if ax < 0
    dx = dx + dx + CF = -1-1+1 =-1

- The minimum code of signum function is generated by a program called superoptimizer.

Superoptimizer: A look at the smallest program. *ASPLOS II*, 1987

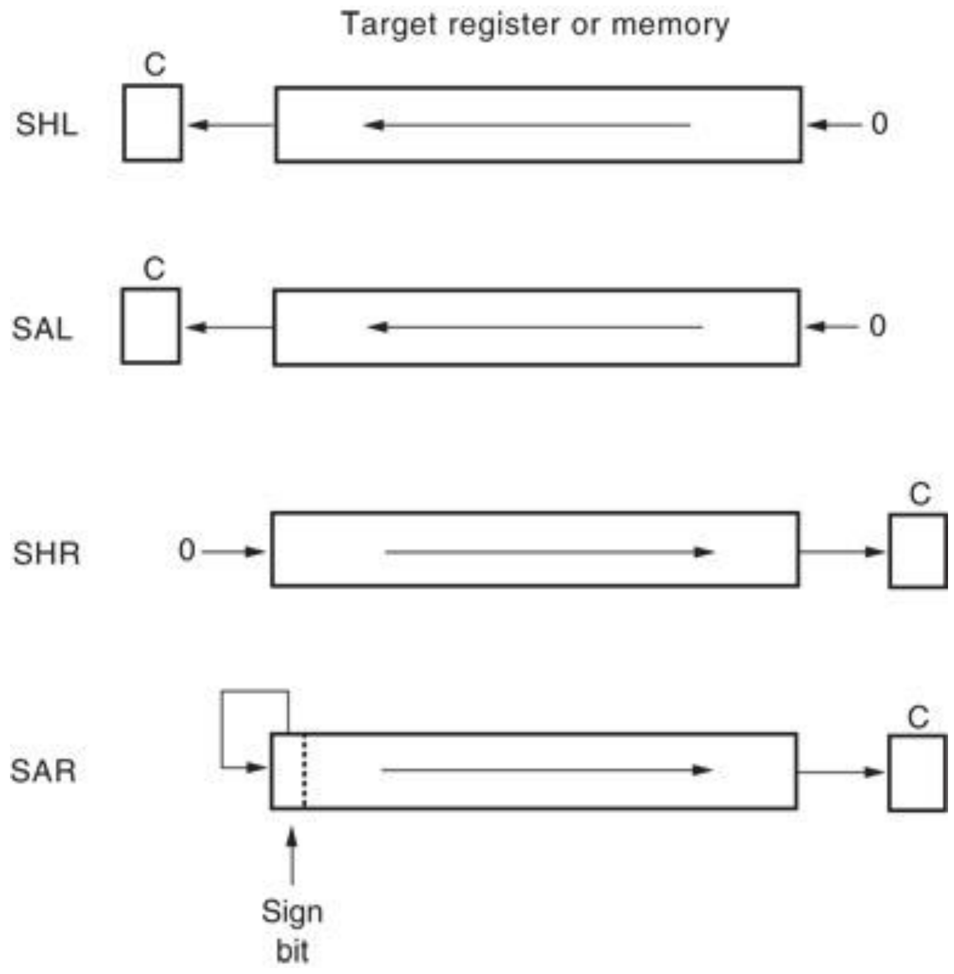
Shift and Rotate

- Shift and rotate instructions manipulate binary numbers at the binary bit level.
  - as did AND, OR, Exclusive-OR, and NOT
- Common applications in low-level software used to control I/O devices.
- The microprocessor contains a complete complement of shift and rotate instructions that are used to shift or rotate any memory data or register.

Shift

- Position or move numbers to the left or right within a register or memory location, for example, SHL AX, 4
  - also perform simple arithmetic as multiplication by powers of 2^{+n} (left shift) and division by powers of 2^{-n} (right shift).
- The microprocessor's instruction set contains four different shift instructions:
  - two are logical; two are arithmetic shifts
  - SHL/SAL/SHR/SAR   REG/MEM, Count
- All four shift operations appear in Figure 5–9.

Figure 5–9 The shift instructions showing the operation and direction of the shift.



- SHL and SAL shift the bits in the destination operand to the left
- SHL and SAL perform the same operation
- SHR copies a 0 through the number
- SAR copies the sign-bit through the number

TABLE 5–22 Example shift instructions.

Assembly Language	*Operation*
SHL AX,1	AX is logically shifted left 1 place
SHR BX,12	BX is logically shifted right 12 places
SHR ECX,10	ECX is logically shifted right 10 places
SHL RAX,50	RAX is logically shifted left 50 places (64-bit mode)
SAL DATA1,CL	The contents of data segment memory location DATA1 are arithmetically shifted left the number of spaces specified by CL
SHR RAX,CL	RAX is logically shifted right the number of spaces specified by CL (64-bit mode)
SAR SI,2	SI is arithmetically shifted right 2 places
SAR EDX,14	EDX is arithmetically shifted right 14 places

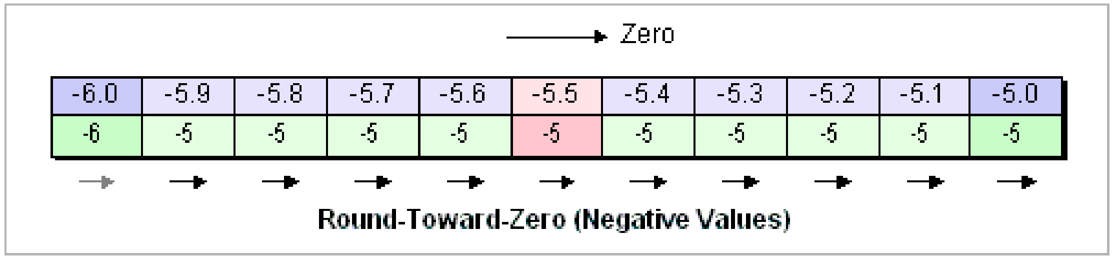
- The count operand can be an immediate value or the CL register. The following example shows how to shift the DX register left 14 places in two different ways.
  - SHL DX,14 ⟺
  - MOV CL, 14
  - SHL DX, CL
- In 16 or 32 bit mode, shift count is a modulo-32 count. The count range is limited to 0 to 31, e.g., SHL AX, 12.
- In 64-bit mode, shift count is a modulo-64 count, The count range is limited to 0 to 63, e.g., SHL RAX, 50.

- Logical shifts multiply or divide unsigned data; arithmetic shifts multiply or divide signed data.
- For example, SHR EAX, CL performs unsigned divide by 2, CL times.
- The SAR and SHR instructions can be used to perform signed or unsigned division of the destination operand by powers of 2.
- Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction.

SAR Rounding for Negative Numbers

- For negative numbers, the quotient from the IDIV is rounded toward zero, while the SAR is rounded toward negative infinity, producing inconsistent result. For example
  - IDIV: divide -9 by 4, the result is -2 with a remainder of -1 (-9÷4 = -2 R -1). ← -9÷4 = -2.25

Round-Toward-Zero

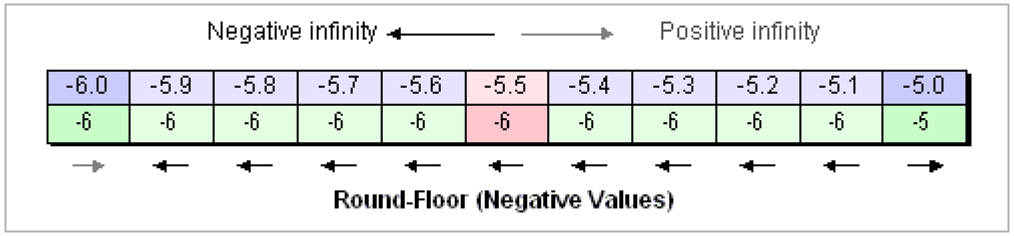


-6.0	-5.9	-5.8	-5.7	-5.6	-5.5	-5.4	-5.3	-5.2	-5.1	-5.0
-6	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5
→	→	→	→	→	→	→	→	→	→	→

Round-Toward-Zero (Negative Values)

  - SAR: shift -9 (0x11110111) right by two, the result is -3 and the “remainder” is +3 (-9÷4 = -3 R +3).

Round-Toward Negative Infinity

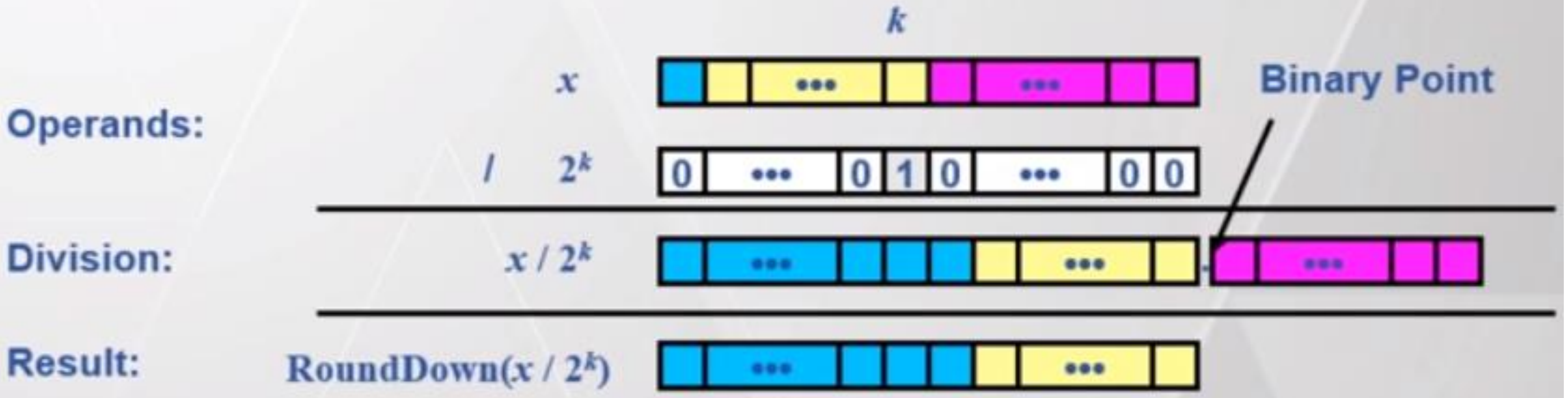

Negative infinity ←
→ Positive infinity

-6.0	-5.9	-5.8	-5.7	-5.6	-5.5	-5.4	-5.3	-5.2	-5.1	-5.0
-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-5
→	←	←	←	←	←	←	←	←	←	→

Round-Floor (Negative Values)

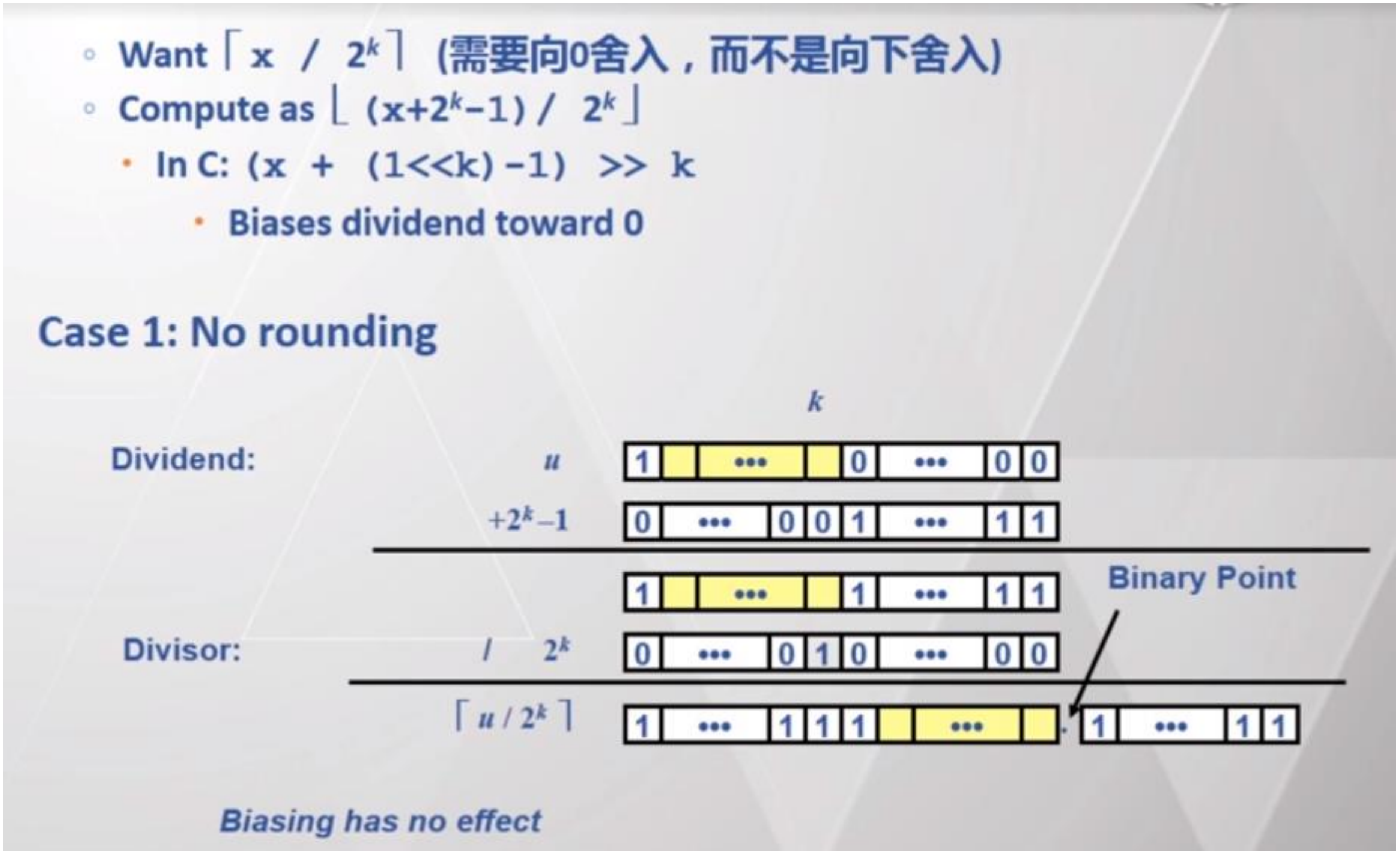
SAR Rounding for Negative Numbers

- x >> k gives $\lfloor x / 2^k \rfloor$
- 采用算术右移
  - 但是x < 0时，舍入错误



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

SAR Rounding for Negative Numbers



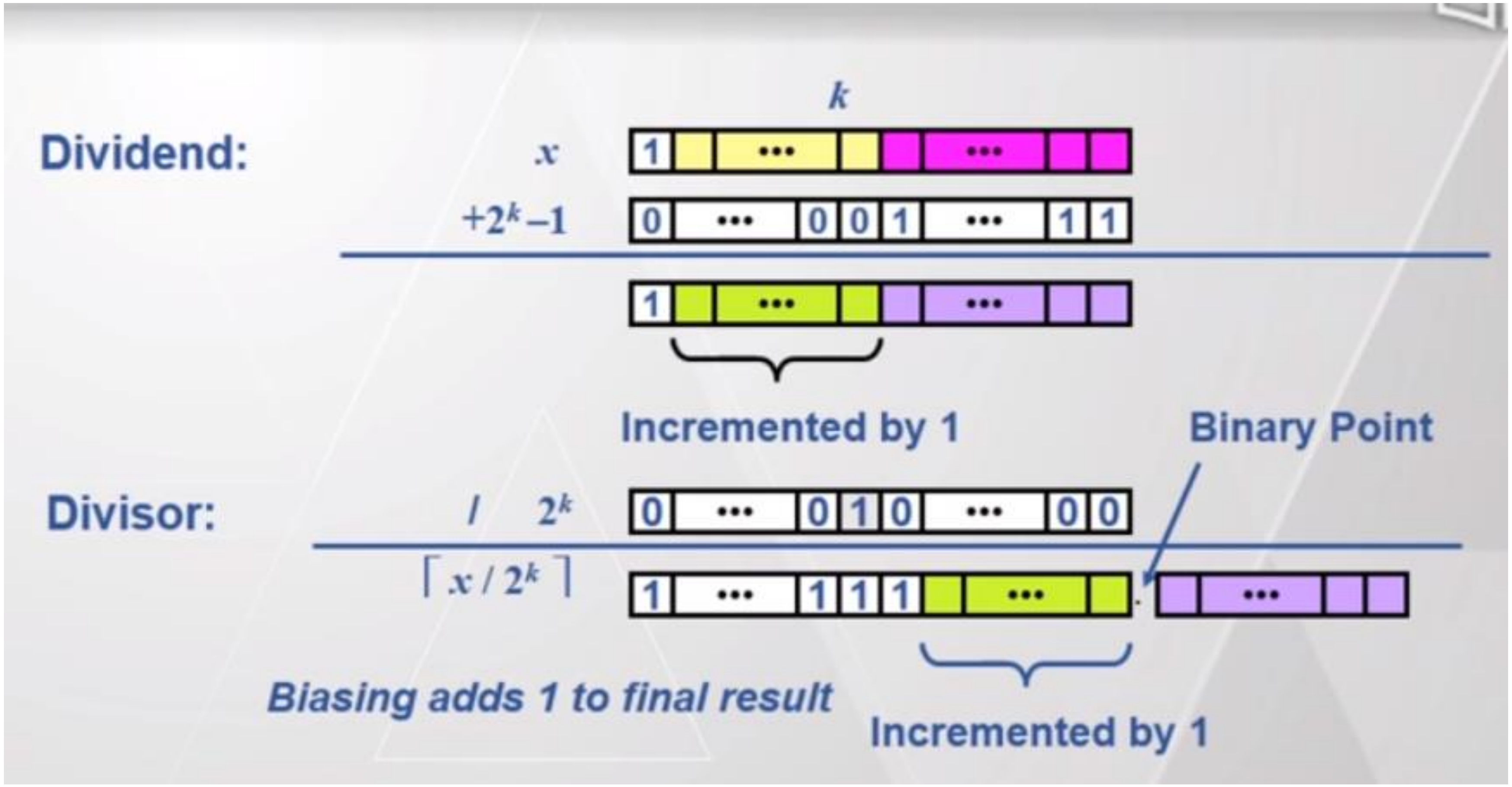
- Want $\lceil x\ /\ 2^k \rceil$ (需要向0舍入，而不是向下舍入)
- Compute as $\lfloor (x+2^k-1)\ /\ 2^k \rfloor$
  - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0

Case 1: No rounding

Biasing has no effect

SAR Rounding for Negative Numbers

Case 2: Rounding



SAR rounding for Negative Numbers

C function

```
int idiv (int x)
{
   return x/8;
}
```

x86-64 clang 13.0.0 -O3

idiv: assembly

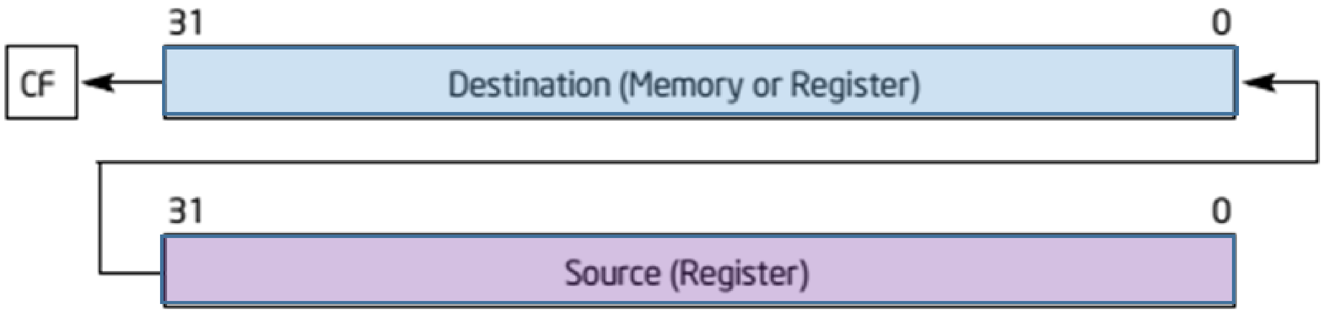
```
1    lea     eax, [rdi + 7]
2    test    edi, edi
3    cmovns  eax, edi
4    sar     eax, 3
```

explanation

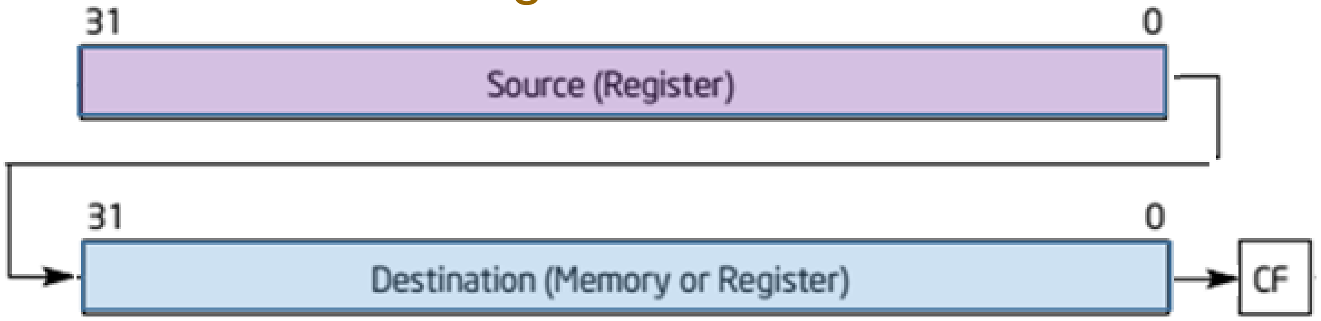
```
1    eax = rdi + 7;
2    if edi >= 0
3       eax = edi;
4    return  eax >> 3;
```

Double-Precision Shifts (80386–Core2 Only)

- 80386 and above contain two double precision shifts: SHLD (shift left) and SHRD (shift right), essentially cross-register shifts.
- Each instruction contains three operands (SHLD/SHRD D, S, Count) instead of 2.
- E.g., the instruction SHLD reg1, reg2, imm8 concatenates the registers reg1 and reg2 and shifts them to the left by the amount specified by imm8.
- Both function with two 16-or 32-bit registers,
  - or with one 16- or 32-bit memory location and a register



SHLD: Logical Shift Left



SHRD: Logical Shift Right

- Example
  - **shld ebx,ecx,16** ; The leftmost 16 bits of ecx fill the
    ; rightmost 16 bits of ebx. The contents
    ; of ecx remain unchanged.
  - **shrd ax, bx, 12** ; Logical right shift of ax by 12
    ; rightmost 12 bits of bx into
    ; leftmost 12 bits of ax. The contents
    ; of bx remain unchanged.

- Example: division of a 128 bit value by eight

```
__uint128_t   u128div (__uint128_t  x)
{
    return x/8;
}
```

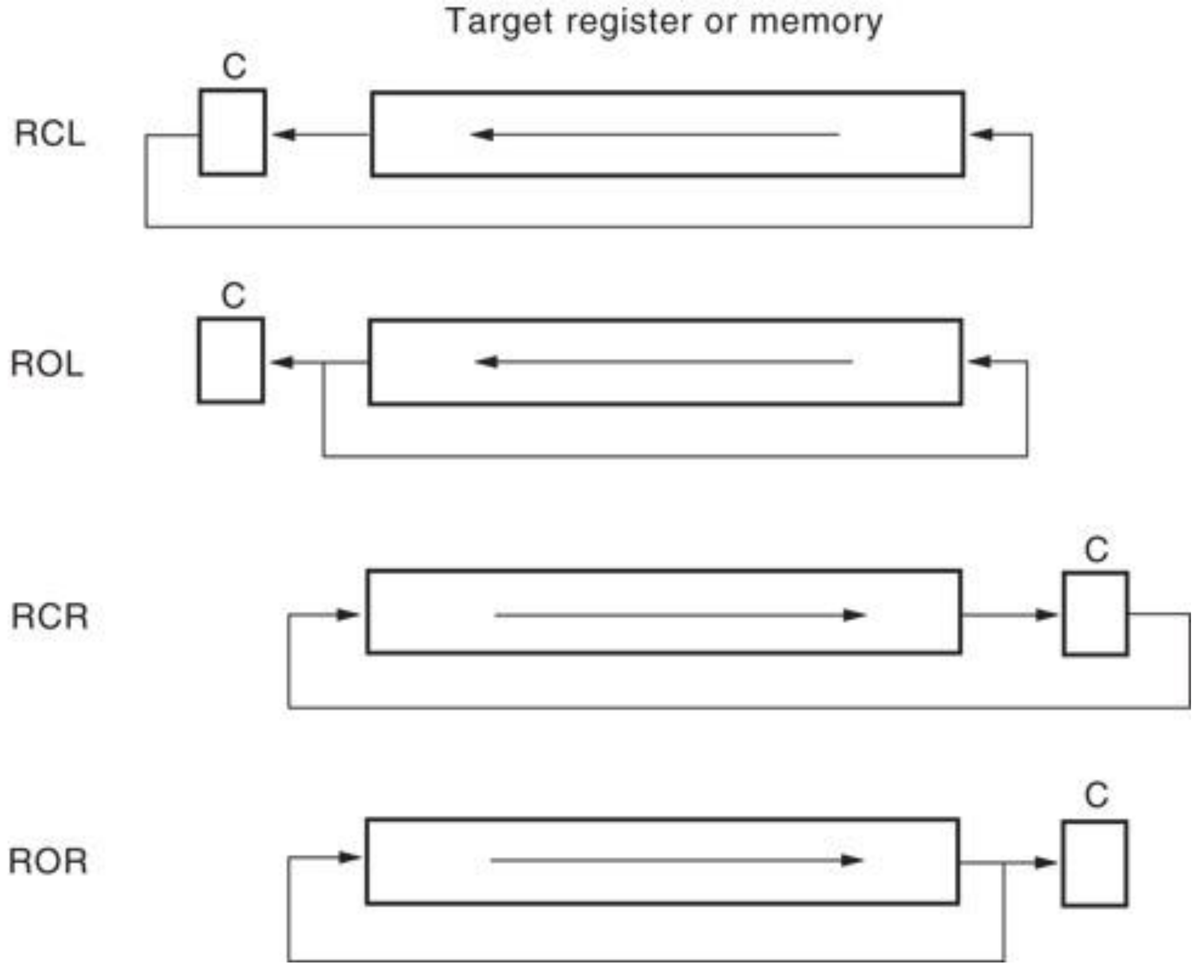
x86-64 gcc 12.2   -O3

```
u128div:
        mov     rax, rdi            ; rax is lower 64 bit
        mov     rdx, rsi            ; rdx is upper 64 bit
        shrd    rax, rsi, 0x3       ; [rsi : rax] >> 3
        shr     rdx, 0x3            ; rdx >> 3
        ret                         ; result =  [rdx : rax] / 8
```

Rotate

- Positions binary data by rotating information in a register or memory location, either from one end to another or through the carry flag.
  - ROL/ROR/RCL/RCR REG/MEM, Count
- With either type of instruction, the programmer can select either a left or a right rotate.
- The rotate left (ROL) and rotate right (ROR) don't include the CF flag in the rotation.
- The rotate through carry left (RCL) and rotate through carry right (RCR) shift the CF flag into the most or least-significant bit.

Figure 5–10 The rotate instructions showing the direction and operation of each rotate.



- A rotate count can be immediate or located in register CL.
  - if CL is used for a rotate count, it does not change
- Rotate instructions are often used to shift wide numbers to the left or right.

- Example: division of a 128 bit value by two

```
__uint128_t  u128div (__uint128_t  x)
{
    return x/2;
}
```

SHRD version

```
u128div:
        mov     rax, rdi
        mov     rdx, rsi
        shrd    rax, rsi, 0x1
        shr     rdx, 0x1
        ret
```

RCR version

```
u128div:
     mov     rax, rdi    ; rax is lower 64 bit
     mov     rdx, rsi    ; rdx is upper 64 bit
     shr     rdx, 0x1    ; rdx >> 1, shift LSB
                         ; of rdx into CF
     rcr     rax, 0x1    ; rotate rax by 1, and
                         ; shift CF into rax
     ret                 ; result =  [rdx : rax] / 2
```

Bit Scan Instructions

- Scan through a number searching for a 1-bit.
  - accomplished by shifting the number
  - available in 80386–Pentium 4
- BSF (bit scan forward) scans the number from the least significant bit toward the left.
- BSR (bit scan reverse) scans the number from the most significant bit toward the right.
  - if no 1-bit is encountered the zero flag is set (ZF = 1)
  - if a 1-bit is encountered, the zero flag is cleared (ZF = 0) and the bit position number of the 1-bit is placed into the destination operand
  - BSF/BSR REG, REG/MEM

- For example, let EAX = 60000000H = 0110 0000 0000 0000 0000 0000 0000 0000B)
- BSF EBX,EAX
  - EBX = 29 (bit 29 is 1)
  - ZF = 0
- BSR EBX,EAX
  - EBX = 30 (bit 30 is 1)
  - ZF = 0
- Extensions of the BSF and BSR instructions:
  - TZCNT (trailing zero count) counts the number of trailing zero bits.
  - LZCNT (leading zero count) returns the number of leading zero bits.

- The key differences between TZCNT/LZCNT and BSF/BSR are that
  - if source operand = 0 (no 1-bit), the content of destination operand in BSF/BSR are undefined, while TZCNT/LZCNT provide operand size as output;
  - if source operand = 0, BSF/BSR only affect the ZF, while TZCNT/LZCNT set both ZF and CF (ZF = 1, CF = 1) and cleared otherwise.
- For example, let EAX = 60000000H = 0110 0000 0000 0000 0000 0000 0000 0000B)
- LZCNT EBX,EAX
  - EBX = 1 (1 leading zero bit)
  - ZF = 0, CF = 0
- BSR EBX,EAX
  - EBX = 30 (bit 30 is 1)
  - ZF = 0

5-6 STRING COMPARISONS

- String instructions are powerful because they allow the programmer to manipulate large blocks of data with relative ease.
- Block data manipulation occurs with MOVS, LODS, STOS, INS, and OUTS.
- Additional string instructions allow a section of memory to be tested against a constant or against another section of memory.
  - SCAS (**string scan**); CMPS (**string compare**)

SCAS

- SCAS instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results.
- The operand to be matched in
  - AL, AX or EAX register (implicit operand).
- The size of operands is selected by
  - SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison)

- The memory operand address is read from ES:(E)DI register depending on the address-size attribute of the instruction and the current operational mode.
- SCAS uses direction flag (D) to select auto-increment or auto-decrement operation for DI/EDI.
- SCAS can be preceded by the conditional repeat prefix REPE (repeat while equal) or REPNE (repeat while not equal) for block comparisons.

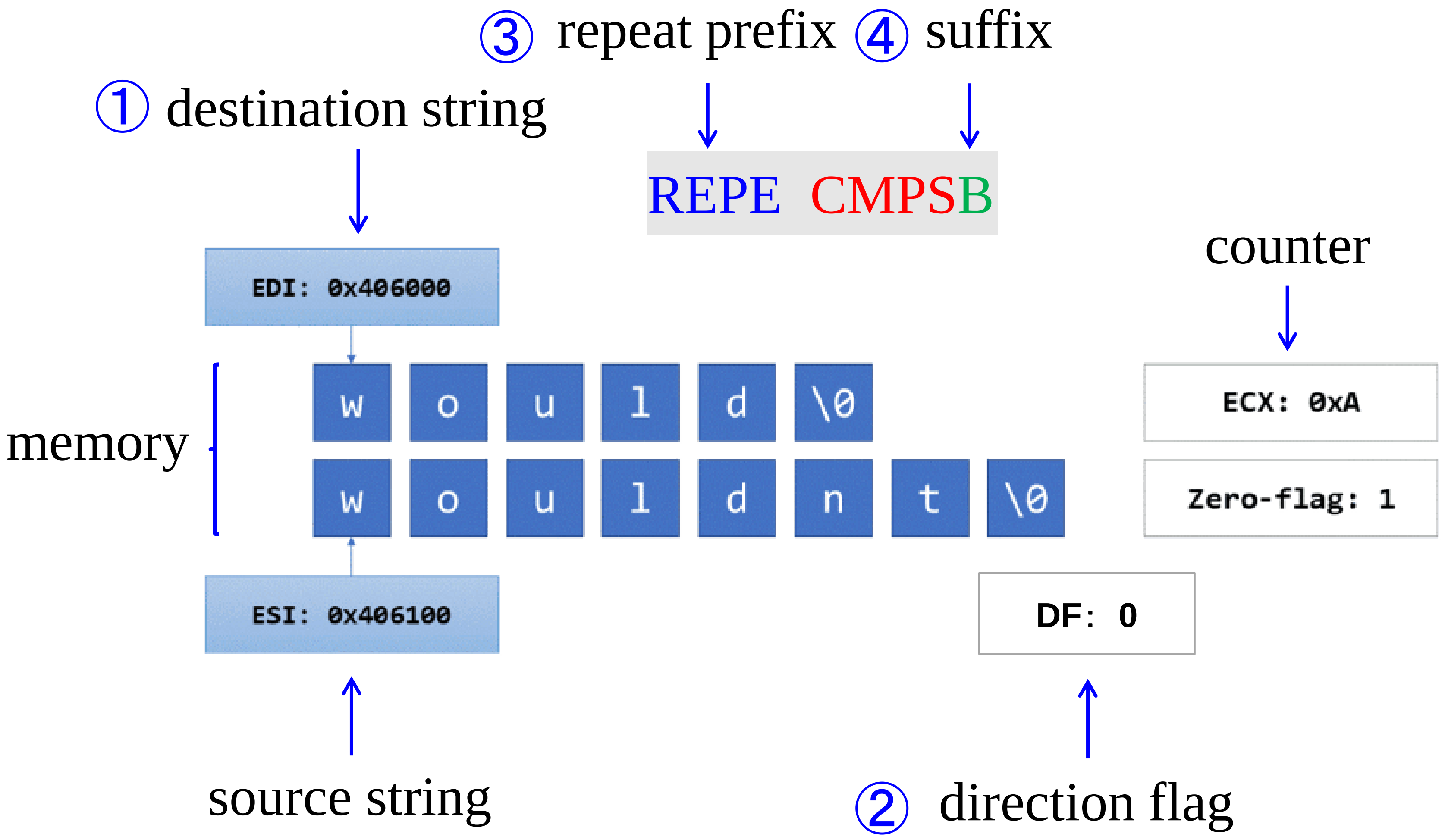
- Suppose that a section of memory is 100 bytes long and begins at location BLOCK.
- This section of memory must be tested to see whether any location contains 00H.
- The following program shows how to search this part of memory for 00H using the SCASB instruction.

```
MOV   DI,OFFSET BLOCK      ;address data
CLD                        ;auto-increment
MOV   CX,100               ;load counter
XOR   AL,AL                ;clear AL
REPNE SCASB
```

CMPS

- Always compares two sections of memory data as bytes (CMPSB), words (CMPSW), or doublewords (CMPSD).
  - contents of the data segment memory location addressed by SI/ESI are compared with contents of extra segment memory addressed by DI/EDI
  - CMPS instruction increments/decrements
- Normally used with REPE or REPNE prefix.
  - alternates are REPZ (repeat while zero) and REPNZ (repeat while not zero)

Example: String Comparison



- The following example compares two sections of memory searching for a match.

```
MOV  SI,OFFSET LINE    ;address LINE
MOV  DI,OFFSET TABLE   ;address TABLE
CLD                    ;auto-increment
MOV  CX,10             ;load counter
REPE CMPSB             ;search
```

  – The CMPSB instruction is prefixed with REPE. This causes the search to continue as long as an equal condition exists.
  – When the CX register becomes 0 or an unequal condition exists, the CMPSB instruction stops execution.
  – After the CMPSB instruction ends
    - if CX = 0 and ZF =1, then two strings match.
    - if CX ≠ 0 or ZF ≠ 1, then the strings do not match.

Assignment

5-2, 3, 9, 11, 13, 16, 17, 21-23, 26, 28, 44-45, 47-48, 51-52, 55

Addition-with-Carry

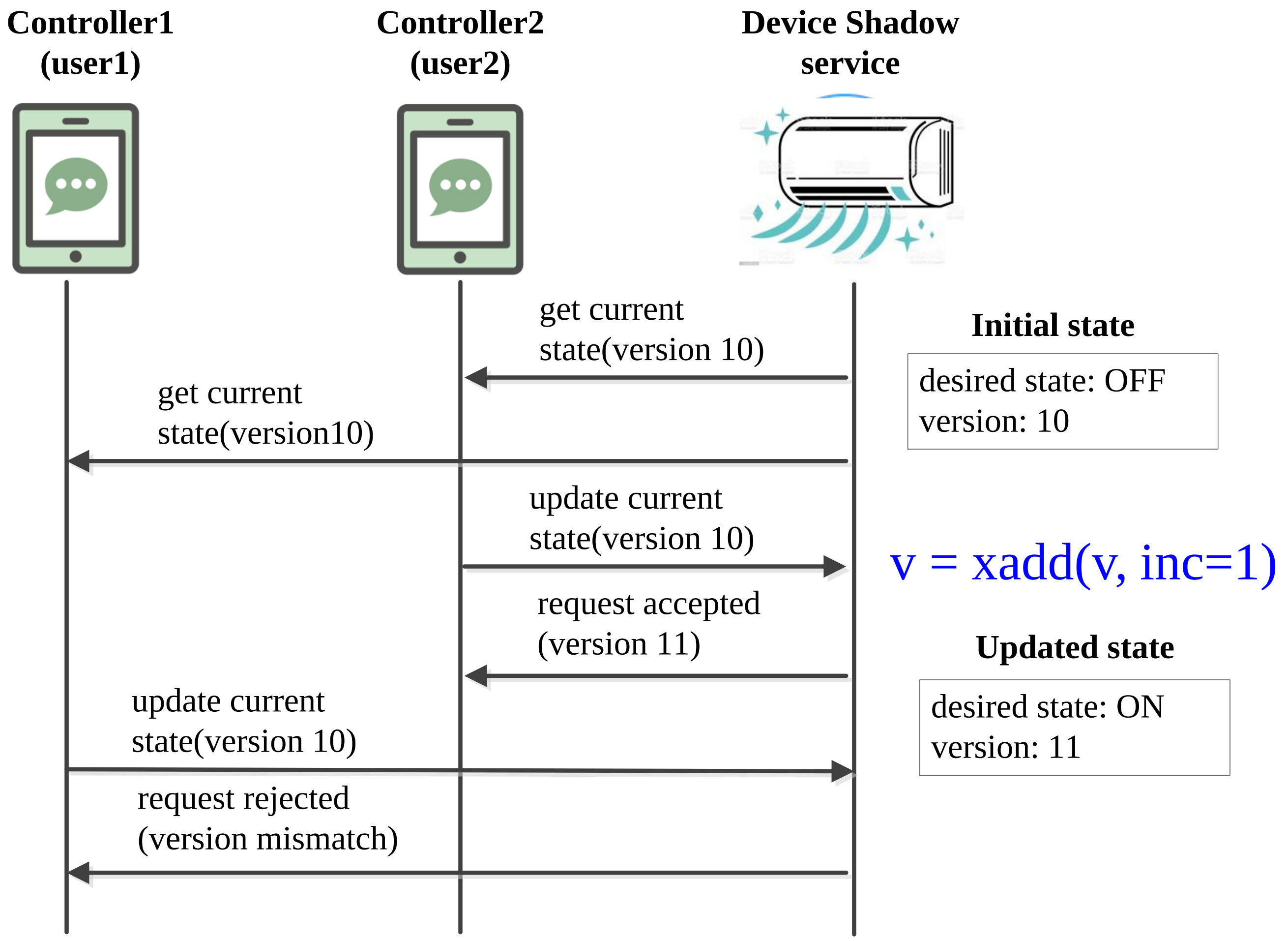
- For example, using ADC to calculate the sum of 12000H and 1F000H.

```
.data
d1    dd 12000h
d2    dd 1f000h
sum dd 0

.code
mov  ax, @DATA
mov  ds, ax
clc                              ; clear carry flag
mov  ax, word ptr d1             ; move 2000h to ax
mov  dx, word ptr d1[2]          ; move 0001h to dx
add   ax, word ptr d2            ; add low part of d2
adc   dx, word ptr d2[2]         ; add high part of d2
mov  word ptr sum, ax            ; move low part of the sum
mov  word ptr sum[2], dx         ; move high part of the sum
```

- XADD might be useful for optimistic locking, which is most applicable to high-volume systems where you do not necessarily maintain a connection for your session.

Version-based optimistic locking for control conflict resolving in AWS IoT platform


Controller1 (user1)
Controller2 (user2)
Device Shadow service
get current state(version 10)
get current state(version10)
update current state(version 10)
request accepted (version 11)
update current state(version 10)
request rejected (version mismatch)
Initial state
desired state: OFF
version: 10
v = xadd(v, inc=1)
Updated state
desired state: ON
version: 11