CPSC-240 Computer Organization and Assembly Language

Chapter 7

Instruction Set Overview

Instructor: Yitsen Ku, Ph.D.
Department of Computer Science,
California State University, Fullerton, USA

CALIFORNIA STATE UNIVERSITY
FULLERTON™

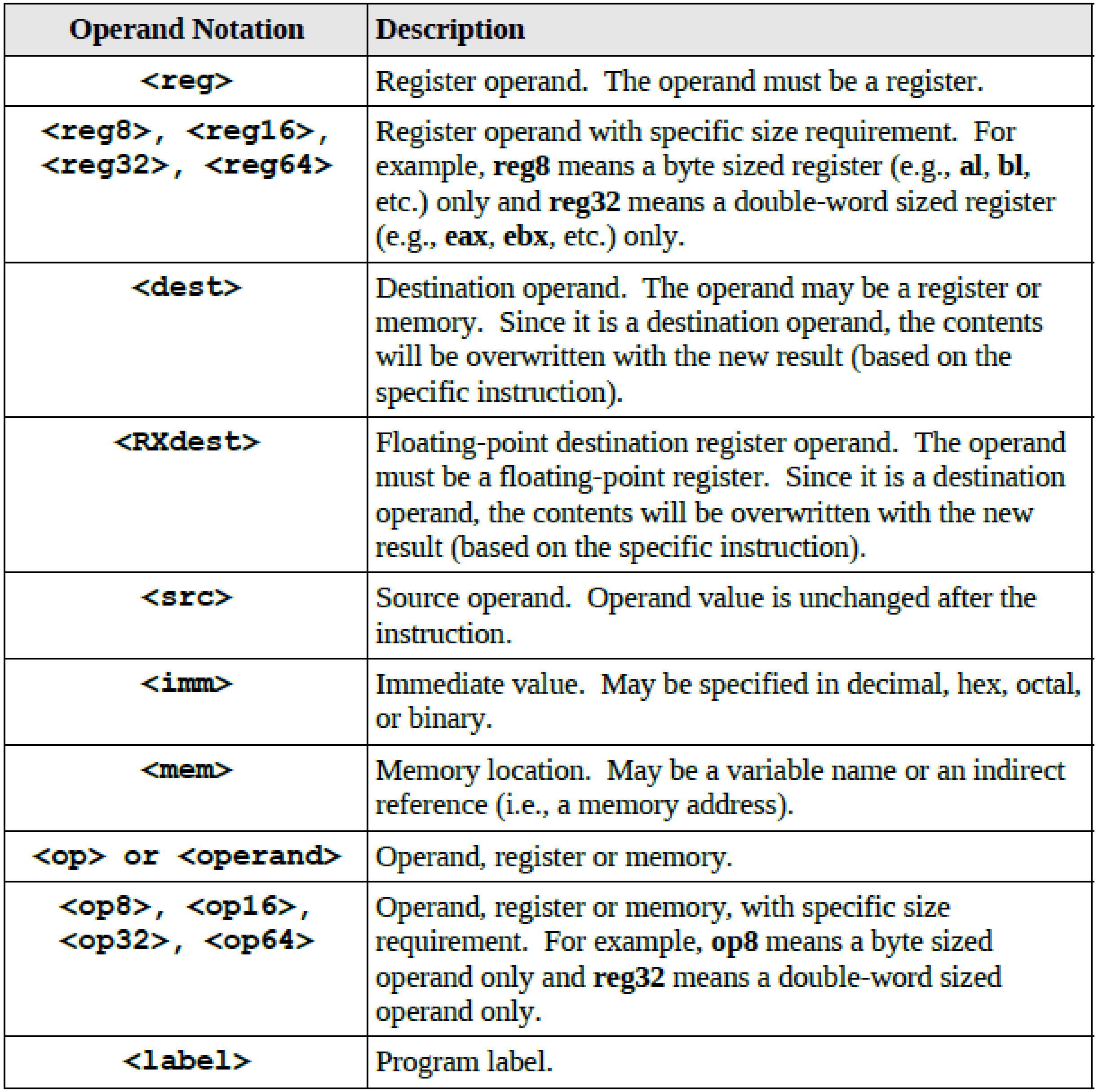
Outline

- Notational Conventions
- Data Movement
- Addresses and Values
- Conversion Instructions
- Integer Arithmetic Instructions
- Logical Instructions
- Control Instructions
- Example Program, Sum of Squares

Notational Conventions

Notational Conventions

- An instruction will consist of the instruction or operation itself (i.e., add, sub, mul, etc.) and the ***operands***.
- The operands refer to where the data (to be operated on) is coming from and/or where the result is to be placed.

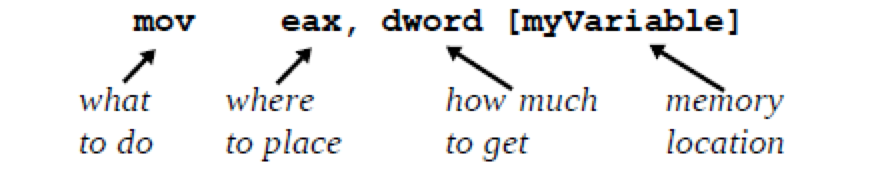


Operand Notation	Description
`<reg>`	Register operand. The operand must be a register.
`<reg8>, <reg16>, <reg32>, <reg64>`	Register operand with specific size requirement. For example, **reg8** means a byte sized register (e.g., **al**, **bl**, etc.) only and **reg32** means a double-word sized register (e.g., **eax**, **ebx**, etc.) only.
`<dest>`	Destination operand. The operand may be a register or memory. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
`<RXdest>`	Floating-point destination register operand. The operand must be a floating-point register. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
`<src>`	Source operand. Operand value is unchanged after the instruction.
`<imm>`	Immediate value. May be specified in decimal, hex, octal, or binary.
`<mem>`	Memory location. May be a variable name or an indirect reference (i.e., a memory address).
`<op> or <operand>`	Operand, register or memory.
`<op8>, <op16>, <op32>, <op64>`	Operand, register or memory, with specific size requirement. For example, **op8** means a byte sized operand only and **reg32** means a double-word sized operand only.
`<label>`	Program label.

Data Movement

Data Movement

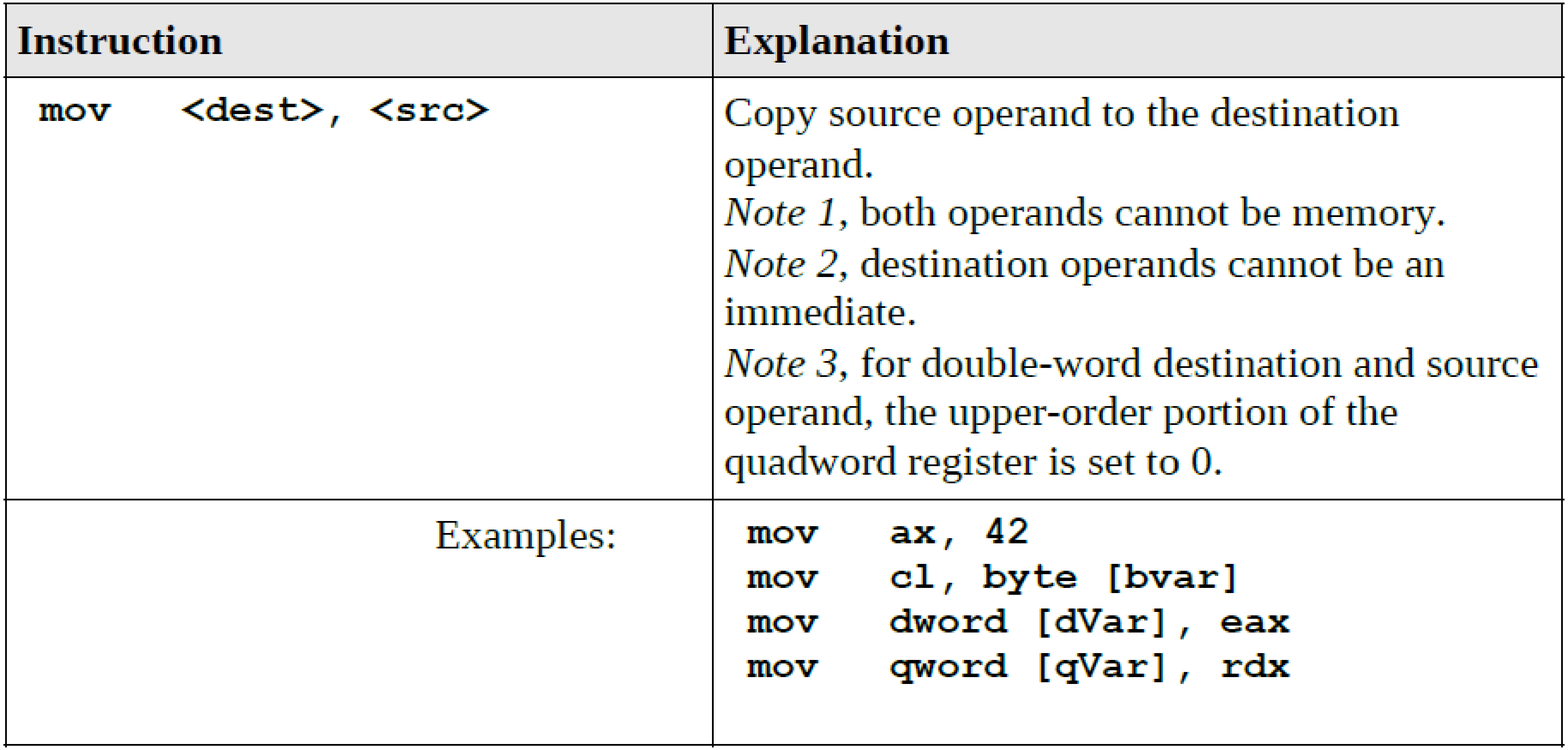
- The general form of the move instruction is:
  mov <dest>, <src>
- The source operand is copied from the source operand into the destination operand. Thevalue of the source operand is unchanged. The destination and source operand must beof the same size (both bytes, both words, etc.). The destination operand cannot be an immediate. Both operands cannot be memory.



```
mov     eax, dword [myVariable]
```

Illustration 14: MOV Instruction Overview

Summary of Move Instructions



Instruction	Explanation
`mov    <dest>, <src>`	Copy source operand to the destination operand. *Note 1*, both operands cannot be memory. *Note 2*, destination operands cannot be an immediate. *Note 3*, for double-word destination and source operand, the upper-order portion of the quadword register is set to 0.

Examples:	`mov    ax, 42`
	`mov    cl, byte [bvar]`
	`mov    dword [dVar], eax`
	`mov    qword [qVar], rdx`

Example

Ex. Assuming the following data declarations:

```
dValue  dd    0
bNum    db    42
wNum    dw    5000
dNum    dd    73000
qNum    dq    73000000
bAns    db    0
wAns    dw    0
dAns    dd    0
qAns    dq    0
```

To perform, the basic operations of:

```
dValue = 27
bAns = bNum
wAns = wNum
dAns = dNum
qAns = qNum
```

Example

- The following instructions could be used:

```
mov     dword [dValue], 27          ; dValue = 27
mov     al, byte [bNum]
mov     byte [bAns], al             ; bAns = bNum
mov     ax, word [wNum]
mov     word [wAns], ax             ; wAns = wNum
mov     eax, dword [dNum]
mov     dword [dAns], eax           ; dAns = dNum
mov     rax, qword [qNum]
mov     qword [qAns], rax           ; qAns = qNum
```

Addresses and Values

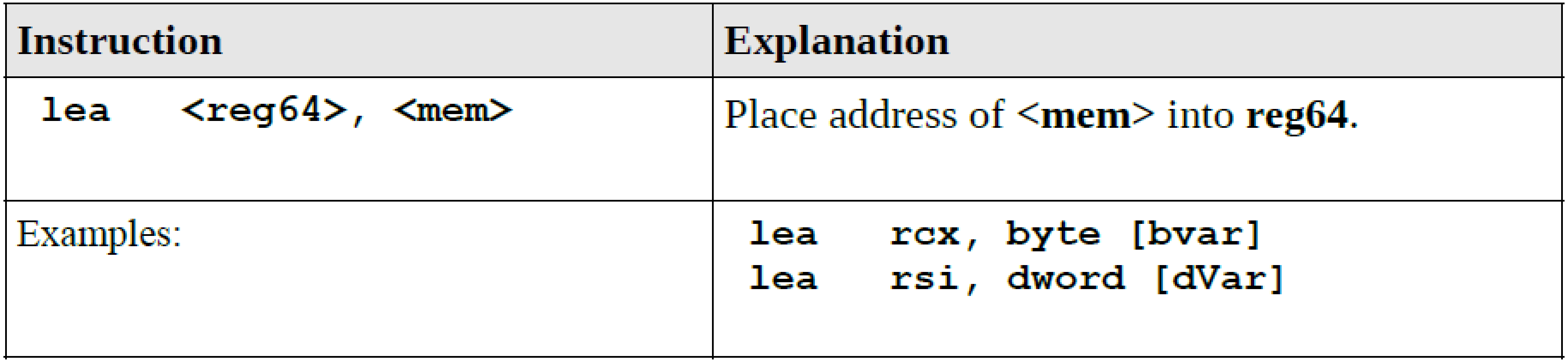
Addresses and Values

- The only way to access memory is with the brackets ([]'s). Omitting the brackets will not access memory and instead obtain the address of the item. For example:

```
mov rax, qword [var1]     ; value of var1 in rax
mov rax, var1             ; address of var1 in rax
```

Addresses and Values

- In addition, the address of a variable can be obtained with the load effective address, or **lea**, instruction. The load effective address instruction is summarized as follows:



Instruction	Explanation
`lea <reg64>, <mem>`	Place address of <**mem**> into **reg64**.

Examples:	`lea rcx, byte [bvar]`
	`lea rsi, dword [dVar]`

- Additional information and extensive examples are presented in Chapter 8, Addressing Modes.

Conversion Instructions

Conversion Instructions

- It is sometimes necessary to convert from one size to another size. For example, a byte might need to be converted to a double-word for some calculations in a formula.
- The process used for conversions depends on the size and type of the operand. The following sections summarize how conversions are performed.

Narrowing Conversions

- Narrowing conversions are converting from a larger type to a smaller type (i.e., word to byte or double-word to word).
- Ex1. if the value of 50 (0x32) is placed in the **rax** register, the **al** register may be accessed directly to obtain the value as follows:

```
mov rax, 50
mov byte [bVal], al
```

Narrowing Conversions

- Ex2. if the value of 500 (0x1f4) is placed in the **rax** register, the **al** register can still be accessed.

```
mov  rax, 500
mov  byte [bVal], al
```

- In this example, the ***bVal*** variable will contain 0xf4 which may lead to incorrect results.

Widening Conversions

- Widening conversions are from a smaller type to a larger type (e.g., byte to word or word to double-word).
- Since the size is being expanded, the upper-order bits must be set based on the sign of the original value.
- As such, the data type, signed or unsigned, must be known and the appropriate process or instructions must be used.

Unsigned Conversions

- For unsigned widening conversions, the upper part of the memory location or register must be set to zero. Since an unsigned value can only be positive, the upper-order bits can only be zero.
- Ex3. to convert the byte value of 50 in the **al** register, to a quadword value in **rbx**, the following operations can be performed.

```
mov   al, 50
mov   rbx, 0
mov   bl, al
```

Unsigned Conversions

- An unsigned conversion from a smaller size to a larger size can also be performed with a special move instruction, as follows:

```
movzx        <dest>, <src>
```

- Which will fill the upper-order bits with zero.
- The **movzx** instruction does not allow a quadword destination operand with a double-word source operand.
- As previously noted, a **mov** instruction with a double-word register destination operand with a double-word source operand will zero the upper-order double-word of the quadword destination register.

Summary of movzx Instruction

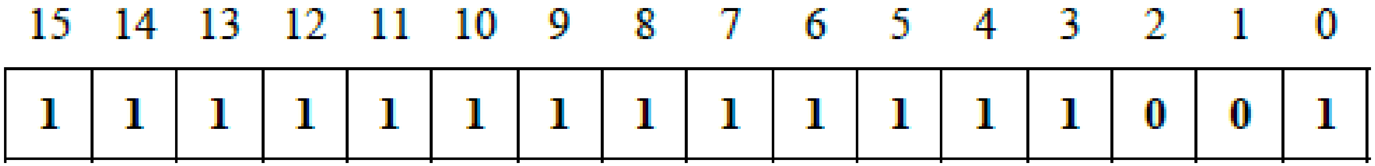
Instruction	Explanation

`movzx <dest>, <src>`
 	
`movzx <reg16>, <op8>`
`movzx <reg32>, <op8>`
`movzx <reg32>, <op16>`
`movzx <reg64>, <op8>`
`movzx <reg64>, <op16>`
	Unsigned widening conversion.
	Note 1, both operands cannot be memory.
	Note 2, destination operands cannot be an immediate.
	Note 3, immediate values not allowed.

Examples:	`movzx cx, byte [bVar]`
	`movzx dx, al`
	`movzx ebx, word [wVar]`
	`movzx ebx, cx`
	`movzx rbx, cl`
	`movzx rbx, cx`

Signed Conversions

- For signed widening conversions, the upper-order bits must be set to either 0's or 1's depending on if the original value was positive or negative.
- Ex. given that the **ax** register is set to -7 (0xfff9), the bits would be set as follows:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1

Signed Conversions

- Since the value is negative, the upper-order bit (bit 15) is a 1. To convert the word value in the **ax** register into a double-word value in the **eax** register, the upper-order bit (1 in this example) is extended or copied into the entire upper-order word (bits 31-16) resulting in the following:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1

Signed Conversions

- A more generalized signed conversion from a smaller size to a larger size can also be performed with some special move instructions, as follows:

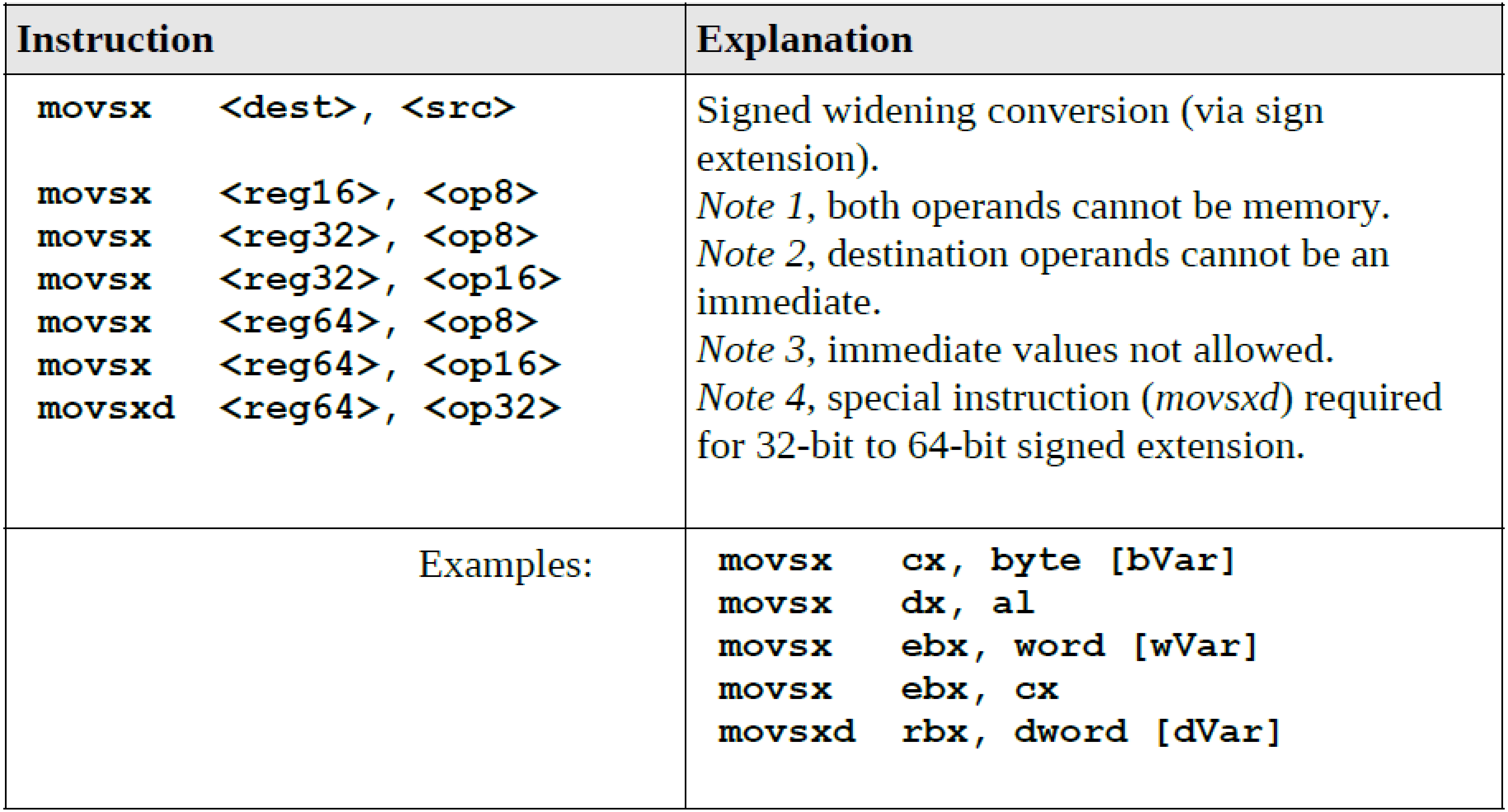
```
movsx     <dest>, <src>
movsxd    <dest>, <src>
```

- The **movsx** instruction is the general form and the **movsxd** instruction is used to allow a quadword destination operand with a double-word source operand.

Summary of Signed Widening Conversion

Instruction	Explanation	Examples
cbw	Convert byte in **al** into word in **ax**. *Note*, only works for **al** to **ax** register.	cbw
cwd	Convert word in **ax** into double-word in **dx:ax**. *Note*, only works for **ax** to **dx:ax** registers.	cwd
cwde	Convert word in **ax** into double-word in **eax**. *Note*, only works for **ax** to **eax** register.	cwde
cdq	Convert double-word in **eax** into quadword in **edx:eax**. *Note*, only works for **eax** to **edx:eax** registers.	cdq
cdqe	Convert double-word in **eax** into quadword in **rax**. *Note*, only works for **rax** register.	cdqe
cqo	Convert quadword in **rax** into word in doublequadword in **rdx:rax**. *Note*, only works for **rax** to **rdx:rax** registers.	cqo

Summary of Signed Widening Conversion



Instruction	Explanation
movsx <dest>, <src> movsx <reg16>, <op8> movsx <reg32>, <op8> movsx <reg32>, <op16> movsx <reg64>, <op8> movsx <reg64>, <op16> movsxd <reg64>, <op32>	Signed widening conversion (via sign extension). *Note 1*, both operands cannot be memory. *Note 2*, destination operands cannot be an immediate. *Note 3*, immediate values not allowed. *Note 4*, special instruction (*movsxd*) required for 32-bit to 64-bit signed extension.
Examples:	movsx cx, byte [bVar] movsx dx, al movsx ebx, word [wVar] movsx ebx, cx movsxd rbx, dword [dVar]

Integer Arithmetic Instructions

Addition

Addition

- The general form of the integer addition instruction is as follows:

```
        add    <dest>, <src>      ; <dest> = <dest> + <src>
```

- Specifically, the source and destination operands are added and the result is placed in the destination operand (over-writing the previous contents).
- The value of the source operand is unchanged.
- The destination and source operand must be of the same size (both bytes, both words, etc.).
- The destination operand cannot be an immediate.

Addition

Ex. assuming the following data declarations:

```
bNum1   db    42
bNum2   db    73
bAns    db    0
wNum1   dw    4321
wNum2   dw    1234
wAns    dw    0
dNum1   dd    42000
dNum2   dd    73000
dAns    dd    0
qNum1   dq    42000000
qNum2   dq    73000000
qAns    dq    0
```

To perform the basic operations of:

```
bAns = bNum1 + bNum2
wAns = wNum1 + wNum2
dAns = dNum1 + dNum2
qAns = qNum1 + qNum2
```

Addition

The following instructions could be used:

```
; bAns = bNum1 + bNum2
mov     al, byte [bNum1]
add     al, byte [bNum2]
mov     byte [bAns], al

; wAns = wNum1 + wNum2
mov     ax, word [wNum1]
add     ax, word [wNum2]
mov     word [wAns], ax

; dAns = dNum1 + dNum2
mov     eax, dword [dNum1]
add     eax, dword [dNum2]
mov     dword [dAns], eax

; qAns = qNum1 + qNum2
mov     rax, qword [qNum1]
add     rax, qword [qNum2]
mov     qword [qAns], rax
```

Increment

- In addition to the basic add instruction, there is an increment instruction that will add one to the specified operand. The general form of the increment instruction is as follows:

```
inc <operand>        ; <operand> = <operand> + 1
```

- The result is exactly the same as using the add instruction (and adding one). When using a memory operand, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) is required to clearly define the size.

Increment

- For example, assuming the following data declarations:

```
bNum        db      42
wNum        dw      4321
dNum        dd      42000
qNum        dq      42000000
```

- To perform, the basic operations of:

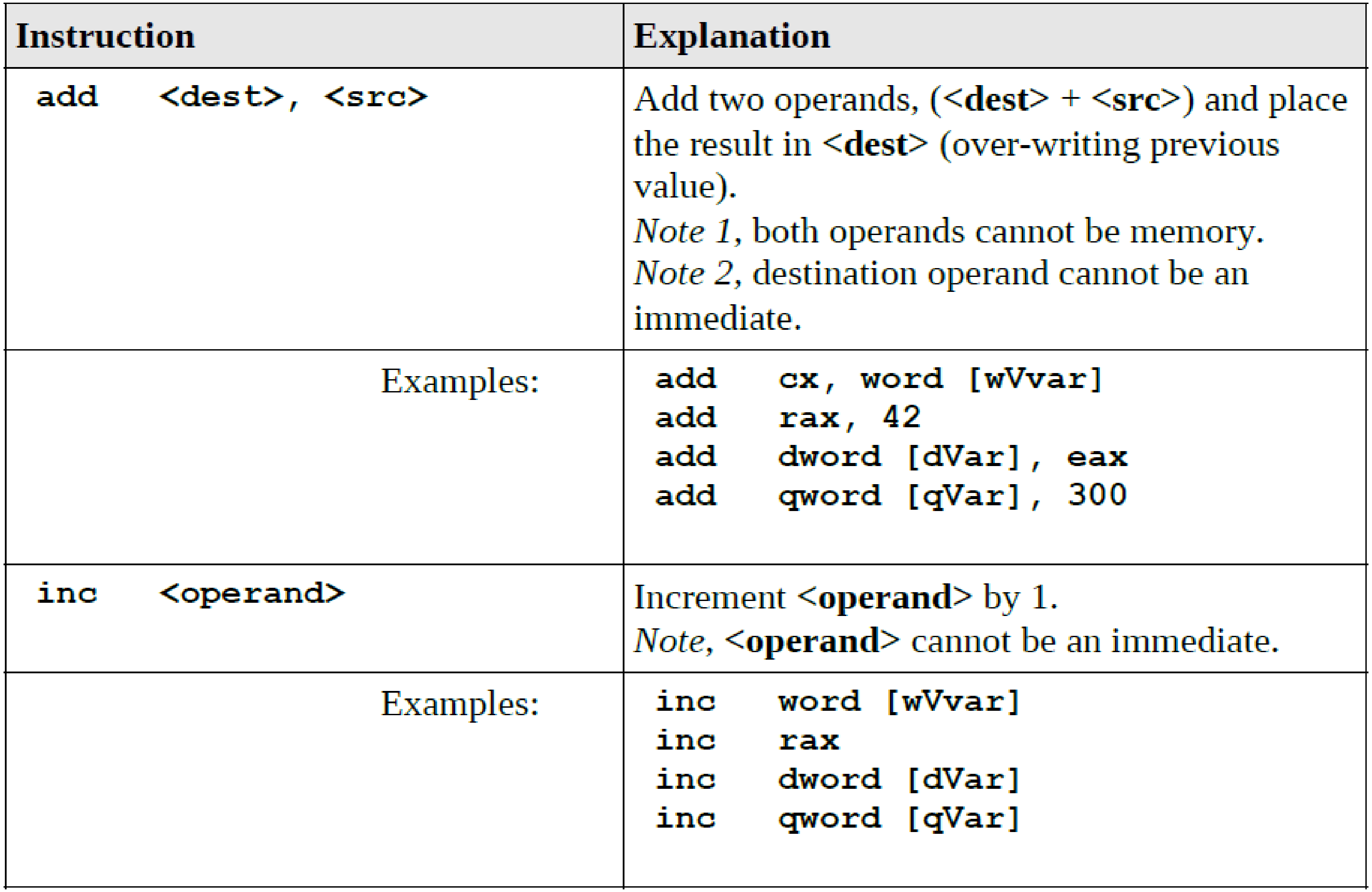
```
rax = rax + 1
bNum = bNum + 1
wNum = wNum + 1
dNum = dNum + 1
qNum = qNum + 1
```

Increment

- The following instructions could be used:

```
inc rax                     ; rax = rax + 1
inc byte [bNum]             ; bNum = bNum + 1
inc word [wNum]             ; wNum = wNum + 1
inc dword [dNum]            ; dNum = dNum + 1
inc qword [qNum]            ; qNum = qNum + 1
```

Summary of add and inc Instruction



Instruction	Explanation
`add    <dest>, <src>`	Add two operands, (<**dest**> + <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.

Examples:	`add    cx, word [wVvar]`
	`add    rax, 42`
	`add    dword [dVar], eax`
	`add    qword [qVar], 300`

`inc    <operand>`	Increment <**operand**> by 1. *Note*, <**operand**> cannot be an immediate.

Examples:	`inc    word [wVvar]`
	`inc    rax`
	`inc    dword [dVar]`
	`inc    qword [qVar]`

Addition with Carry

- For assembly language programs the Least Significant Quadword (LSQ) is added with the **add** instruction and then immediately the Most Significant Quadword (MSQ) is added with the **adc** which will add the quadwords and include a carry from the previous addition operation.
- The general form of the integer add with carry instruction is as follows:

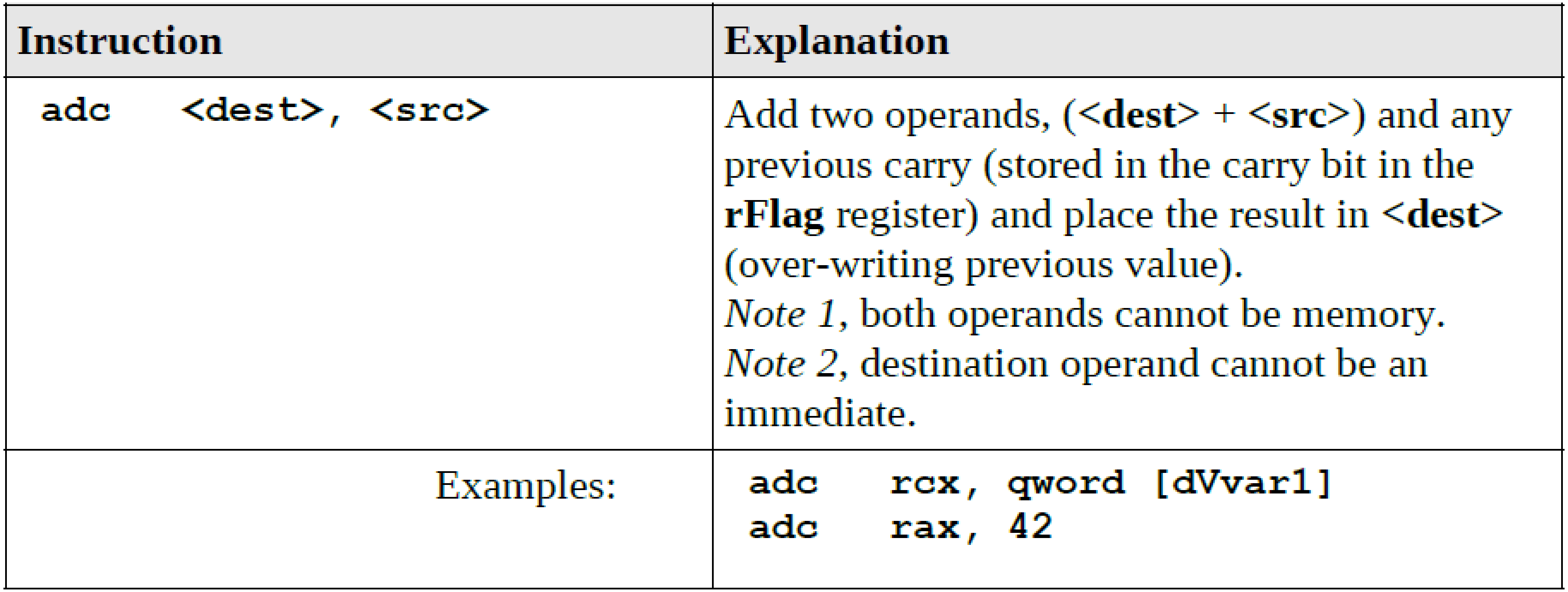
adc <dest>, <src>   ; <dest> = <dest> + <src> + <carryBit>

Addition with Carry

Ex. given the following declarations

```
dquad1  ddq 0x1A000000000000000    ; 128 bits
dquad2  ddq 0x2C000000000000000    ; 128 bits
dqSum   ddq  0                     ; 128 bits
mov     rax, qword [dquad1]
mov     rdx, qword [dquad1+8]
add     rax, qword [dquad2]        ; add low 64 bits
adc     rdx, qword [dquad2+8]      ; add high 64 bits
mov     qword [dqSum], rax
mov     qword [dqSum+8], rdx
```

Summary of ADC Instruction



Instruction	**Explanation**
`adc <dest>, <src>`	Add two operands, (<**dest**> + <**src**>) and any previous carry (stored in the carry bit in the **rFlag** register) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.

Examples:	`adc rcx, qword [dVvar1]`
	`adc rax, 42`

Integer Arithmetic Instructions

Subtraction

Subtraction

- The general form of the integer subtraction instruction is as follows:

```
sub <dest>, <src>        ; <dest> = <dest> - <src>
```

- The source operand is subtracted from the destination operand and the result is placed in the destination operand (over-writing the previous value).
- The value of the source operand is unchanged.
- The destination and source operand must be of the same size (both bytes, both words, etc.).
- The destination operand cannot be an immediate.

Subtraction

Ex. Assuming the following data declarations:

```
bNum1   db   73
bNum2   db   42
bAns    db   0
wNum1   dw   1234
wNum2   dw   4321
wAns    dw   0
dNum1   dd   73000
dNum2   dd   42000
dAns    dd   0
qNum1   dq   73000000
qNum2   dq   42000000
qAns    dq   0
```

Question: To perform the basic operations of:

```
bAns = bNum1 - bNum2
wAns = wNum1 - wNum2
dAns = dNum1 - dNum2
qAns = qNum1 - qNum2
```

Subtraction

The following instructions could be used:

```
; bAns = bNum1 - bNum2
mov     al, byte [bNum1]
sub     al, byte [bNum2]
mov     byte [bAns], al

; wAns = wNum1 - wNum2
mov     ax, word [wNum1]
sub     ax, word [wNum2]
mov     word [wAns], ax

; dAns = dNum1 - dNum2
mov     eax, dword [dNum1]
sub     eax, dword [dNum2]
mov     dword [dAns], eax

; qAns = qNum1 - qNum2
mov     rax, qword [qNum1]
sub     rax, qword [qNum2]
mov     qword [qAns], rax
```

Decrement

- In addition to the basic sub instruction, there is an increment instruction that will subtract one from the specified operand. The general form of the decrement instruction is as follows:

```
dec <operand>      ; <operand> = <operand> - 1
```

- The result is exactly the same as using the sub instruction (and subtracting one). When using a memory operand, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) is required to clearly define the size.

Decrement

- Ex. Assuming the following data declarations:

```
bNum  db    42
wNum dw   4321
dNum  dd    42000
qNum  dq    42000000
```

- Question: To perform, the basic operations of:

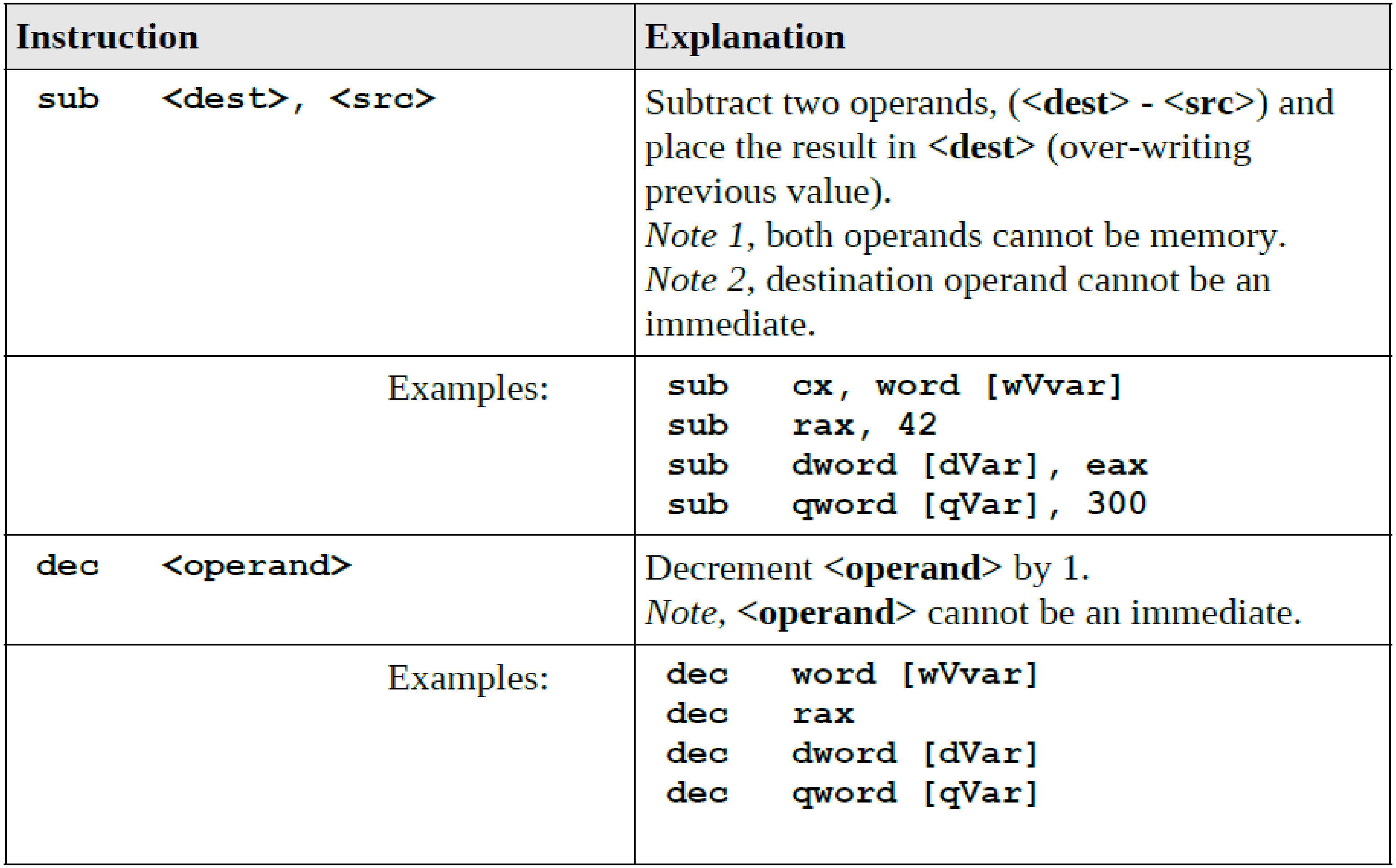
```
rax = rax - 1
bNum = bNum - 1
wNum = wNum - 1
dNum = dNum - 1
qNum = qNum - 1
```

Decrement

- The following instructions could be used:

```
dec   rax                   ; rax = rax - 1
dec   byte [bNum]           ; bNum = bNum - 1
dec   word [wNum]           ; wNum = wNum - 1
dec   dword [dNum]          ; dNum = dNum - 1
dec   qword [qNum]          ; qNum = qNum - 1
```

Summary of sub and dec Instruction



Instruction	Explanation
sub <dest>, <src>	Subtract two operands, (<**dest**> - <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.
Examples:	sub cx, word [wVvar] sub rax, 42 sub dword [dVar], eax sub qword [qVar], 300
dec <operand>	Decrement <**operand**> by 1. *Note*, <**operand**> cannot be an immediate.
Examples:	dec word [wVvar] dec rax dec dword [dVar] dec qword [qVar]

Integer Arithmetic Instructions

Multiplication

Multiplication

- Mul instruction is used for unsigned multiplication. Imul instruction is used for signed multiplication.
- Multiplication typically produces double sized results. That is, multiplying two ***n***-bit values produces a 2***n***-bit result.

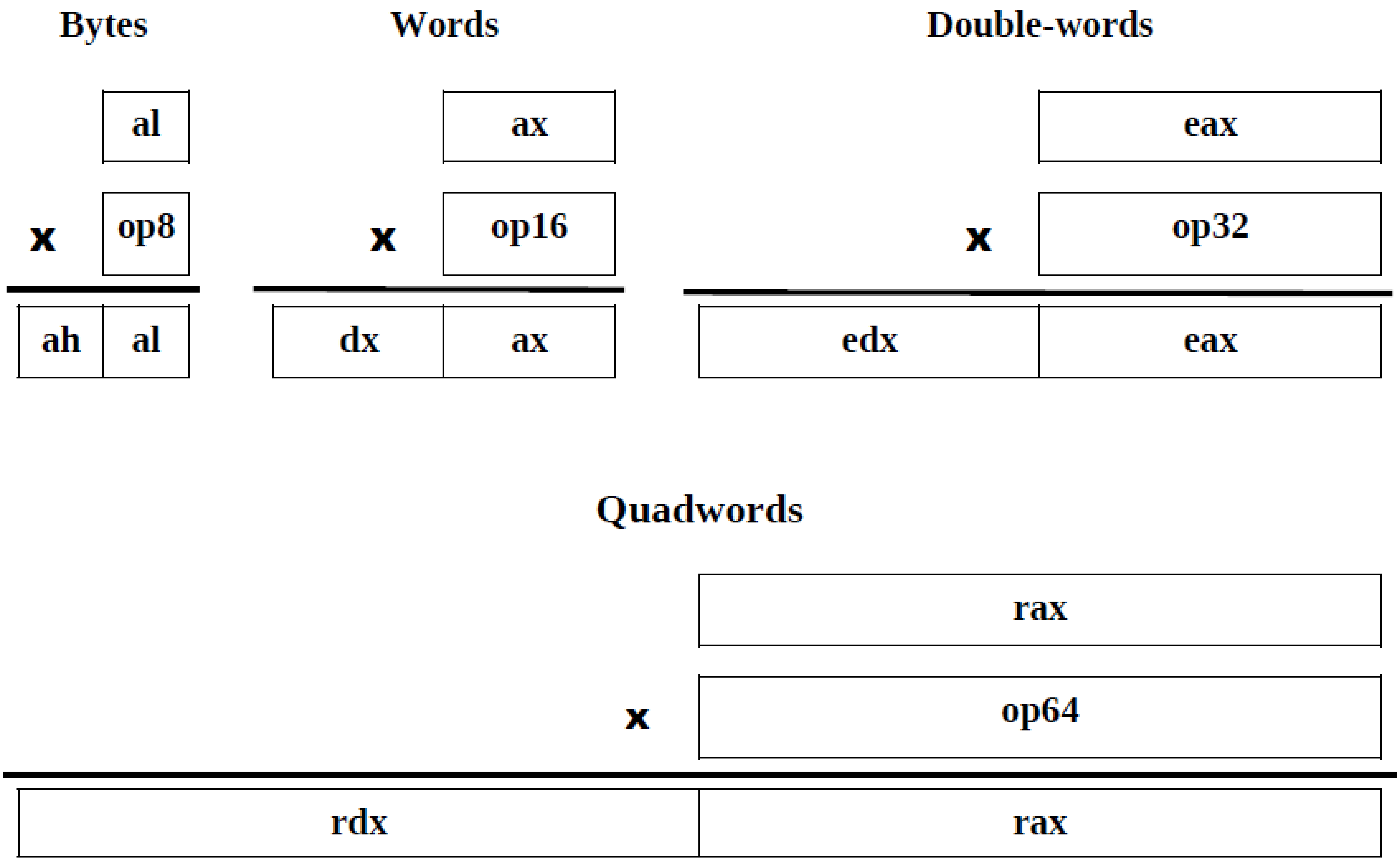
Unsigned Multiplication

- The general form of the integer multiplication instruction is as follows:

```
    mul <src>                         ; <A> = <A> * <src>
```

- Where the source operand must be a register or memory location. An immediate operand is not allowed.
- For the single operand multiply instruction, the **A** register (**al/ax/eax/rax**) must be used for one of the operands.
- The other operand can be a memory location or register, but not an immediate.
- The result will be placed in the **A** and possibly **D** registers.

Unsigned Multiplication



Bytes

al

x op8

ah al

Words

ax

x op16

dx ax

Double-words

eax

x op32

edx eax

Quadwords

rax

x op64

rdx rax

Unsigned Multiplication

Ex. assuming the following data declarations:

```
bNum1       db      42
bNum2       db      73
wAns        db      0
wAns1       dw      0

wNum1       dw      4321
wNum2       dw      1234
dAns2       dw      0

dNum1       dd      42000
dNum2       dd      73000
qAns3       dd      0

qNum1       dq      42000000
qNum2       dq      73000000
dqAns4      dq      0
```

Unsigned Multiplication

Question: To perform the basic operations of:

```
wAns = bNumA^2                          ; bNumA squared
bAns1 = bNumA * bNumB
wAns1 = bNumA * bNumB
wAns2 = wNumA * wNumB
dAns2 = wNumA * wNumB
dAns3 = dNumA * dNumB
qAns3 = dNumA * dNumB
qAns4 = qNumA * qNumB
dqAns4 = qNumA * qNumB
```

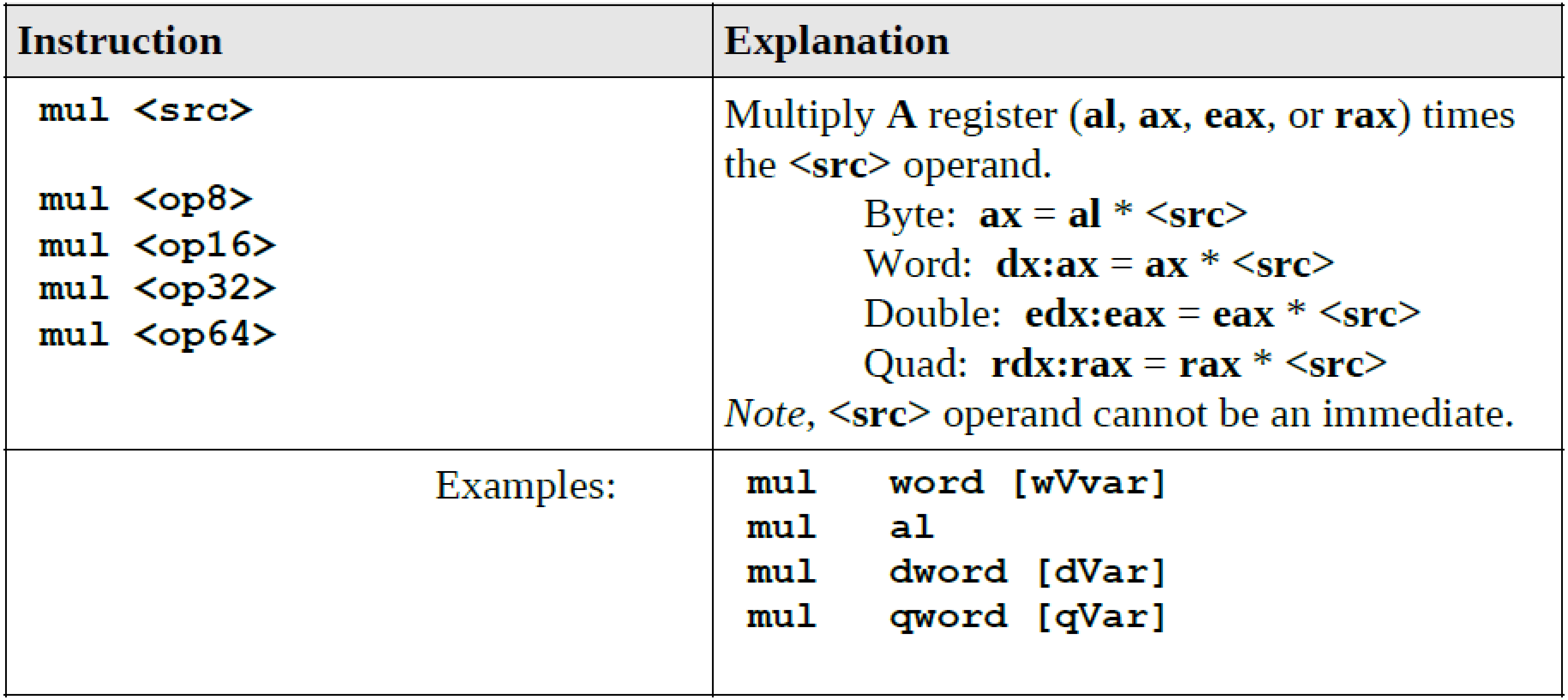
The following instructions could be used:

```
; wAns = bNumA^2 or bNumA squared
mov     al, byte [bNumA]
mul     al                              ; result in ax
mov     word [wAns], ax
; wAns1 = bNumA * bNumB
mov     al, byte [bNumA]
mul     byte [bNumB]                    ; result in ax
mov     word [wAns1], ax
```

Unsigned Multiplication

```
; dAns2 = wNumA * wNumB
mov   ax, word [wNumA]
mul   word [wNumB]                  ; result in dx:ax
mov   word [dAns2], ax
mov   word [dAns2+2], dx
; qAns3 = dNumA * dNumB
mov   eax, dword [dNumA]
mul   dword [dNumB]                 ; result in edx:eax
mov   dword [qAns3], eax
mov   dword [qAns3+4], edx
; dqAns4 = qNumA * qNumB
mov   rax, qword [qNumA]
mul   qword [qNumB]                 ; result in rdx:rax
mov   qword [dqAns4], rax
mov   qword [dqAns4+8], rdx
```

Summary of mul Instruction



Instruction	Explanation

`mul <src>`
 	
`mul <op8>`
`mul <op16>`
`mul <op32>`
`mul <op64>`
	Multiply **A** register (**al**, **ax**, **eax**, or **rax**) times the <**src**> operand.
	Byte: **ax** = **al** * <**src**>
	Word: **dx:ax** = **ax** * <**src**>
	Double: **edx:eax** = **eax** * <**src**>
	Quad: **rdx:rax** = **rax** * <**src**>
	Note, <**src**> operand cannot be an immediate.

Examples:	`mul word [wVvar]`
	`mul al`
	`mul dword [dVar]`
	`mul qword [qVar]`

Signed Multiplication

- The signed multiplication allows a wider range of operands and operand sizes. The general forms of the signed multiplication are as follows:

```
imul <source>                 ; <A> = <A> * <source>
imul <dest>, <src/imm>        ; <dest> = <dest>*<src/imm>
imul <dest>, <src>, <imm>     ; <dest> = <src> * <imm>
```

- The size of the immediate value is limited to the size of the source operand, up to a double-word size (32-bit), even for quadword multiplications.
- The final result is truncated to the size of the destination operand. A byte sized destination operand is not supported.

Signed Multiplication

Ex. assuming the following data declarations:

```
wNumA   dw    1200
wNumB   dw    -2000
wAns1   dw    0
wAns2   dw    0
dNumA   dd    42000
dNumB   dd    -13000
dAns1   dd    0
dAns2   dd    0
qNumA   dq    120000
qNumB   dq    -230000
qAns1   dq    0
qAns2   dq    0
```

Question: To perform the basic operations of:

```
wAns1 = wNumA * -13
wAns2 = wNumA * wNumB
dAns1 = dNumA * 113
dAns2 = dNumA * dNumB
qAns1 = qNumA * 7096
qAns2 = qNumA * qNumB
```

Signed Multiplication

The following instructions could be used:

```
; wAns1 = wNumA * -13
mov     ax, word [wNumA]
imul    ax, -13                                         ; result in ax
mov     word [wAns1], ax
; wAns2 = wNumA * wNumB
mov     ax, word [wNumA]
imul    ax, word [wNumB]                                ; result in ax
mov     word [wAns2], ax
; dAns1 = dNumA * 113
mov     eax, dword [dNumA]
imul    eax, 113                                        ; result in eax
mov     dword [dAns1], eax
; dAns2 = dNumA * dNumB
mov     eax, dword [dNumA]
imul    eax, dword [dNumB]                              ; result in eax
mov     dword [dAns2], eax
```

Signed Multiplication

```
        ; qAns1 = qNumA * 7096
        mov     rax, qword [qNumA]
        imul    rax, 7096                       ; result in rax
        mov     qword [qAns1], rax
        ; qAns2 = qNumA * qNumB
        mov     rax, qword [qNumA]
        imul    rax, qword [qNumB]              ; result in rax
        mov     qword [qAns2], rax
```

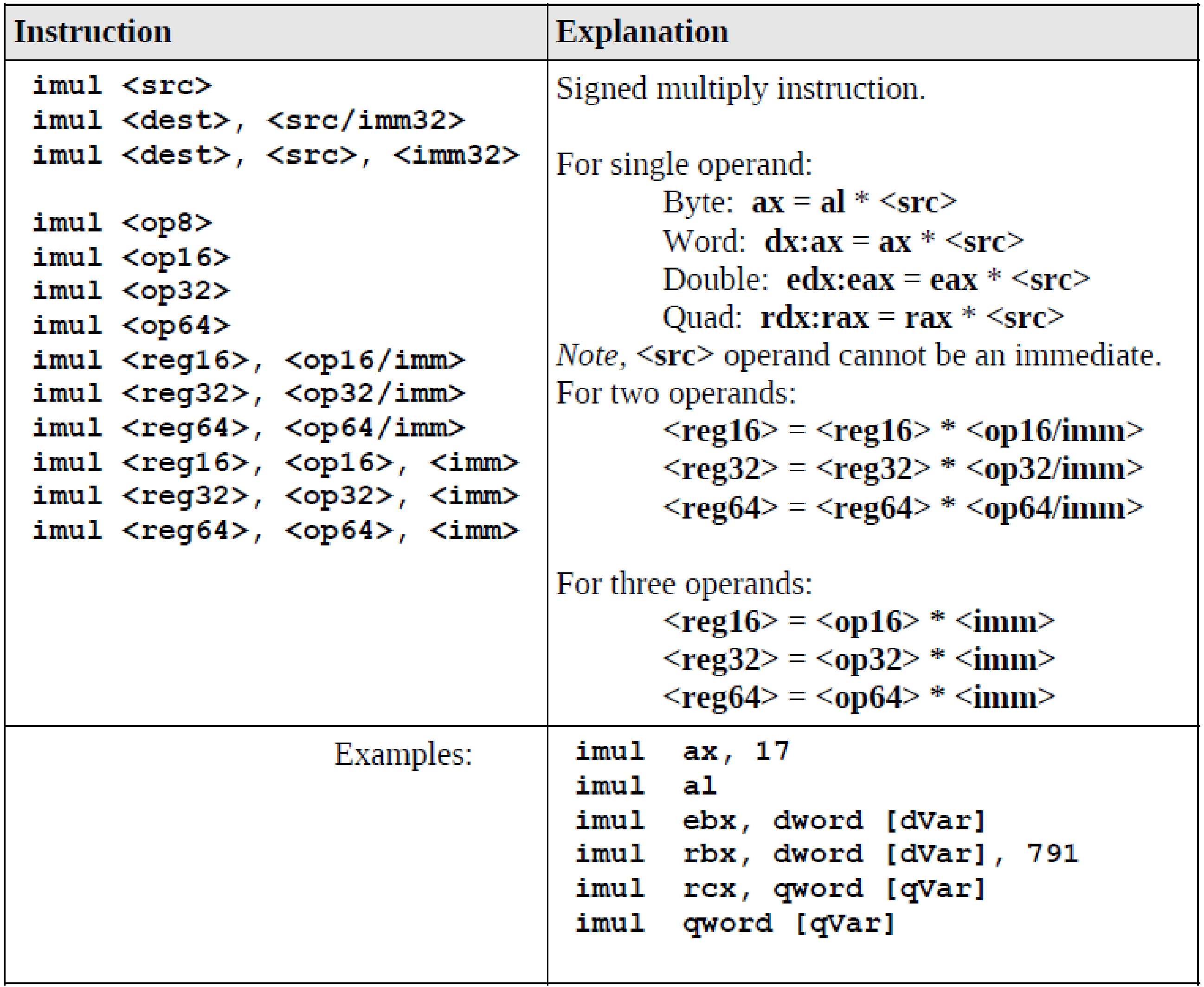
Another way to perform the multiplication of

qAns1 = qNumA * 7096

Would be as follows:

```
        ; qAns1 = qNumA * 7096
        mov     rcx, qword [qNumA]
        imul    rbx, rcx, 7096                  ; result in rbx
        mov     qword [qAns1], rbx
```

Summary of imul Instruction



Instruction	Explanation

`imul <src>`
`imul <dest>, <src/imm32>`
`imul <dest>, <src>, <imm32>`
 	
`imul <op8>`
`imul <op16>`
`imul <op32>`
`imul <op64>`
`imul <reg16>, <op16/imm>`
`imul <reg32>, <op32/imm>`
`imul <reg64>, <op64/imm>`
`imul <reg16>, <op16>, <imm>`
`imul <reg32>, <op32>, <imm>`
`imul <reg64>, <op64>, <imm>`
	Signed multiply instruction.
	
	For single operand:
	Byte: **ax** = **al** * <src>
	Word: **dx:ax** = **ax** * <src>
	Double: **edx:eax** = **eax** * <src>
	Quad: **rdx:rax** = **rax** * <src>
	Note, <src> operand cannot be an immediate.
	For two operands:
	<reg16> = **<reg16>** * **<op16/imm>**
	<reg32> = **<reg32>** * **<op32/imm>**
	<reg64> = **<reg64>** * **<op64/imm>**
	
	For three operands:
	<reg16> = **<op16>** * **<imm>**
	<reg32> = **<op32>** * **<imm>**
	<reg64> = **<op64>** * **<imm>**

Examples:	`imul   ax, 17`
	`imul   al`
	`imul   ebx, dword [dVar]`
	`imul   rbx, dword [dVar], 791`
	`imul   rcx, qword [qVar]`
	`imul   qword [qVar]`

Integer Arithmetic Instructions

Division

Integer Division

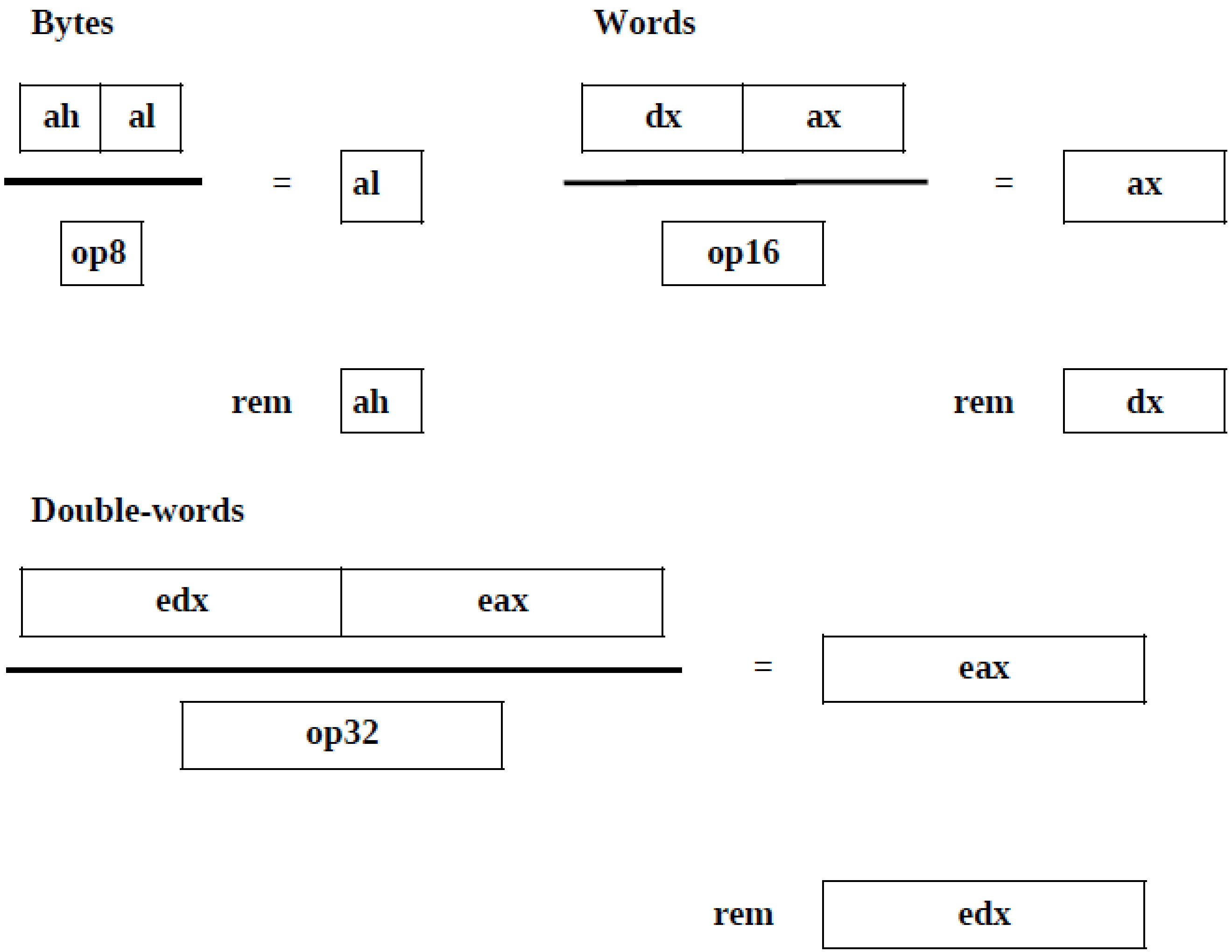
- Mathematically, there are special rules for handling division of signed values. As such, different instructions are used for unsigned division (**div**) and signed division (**idiv**).

Recall that $\dfrac{dividend}{divisor} = quotient$

Integer Division

- The **A**, and possibly the **D** register, must be used in combination for the dividend.
  - Byte Divide: **ax** for 16-bits
  - Word Divide: **dx:ax** for 32-bits
  - Double-word divide: **edx:eax** for 64-bits
  - Quadword Divide: **rdx:rax** for 128-bits
- The divisor can be a memory location or register, but not an immediate. Additionally, the result will be placed in the **A** register (**al**/**ax**/**eax/rax**) and the remainder in either the **ah**, **dx**, **edx**, or **rdx** register.
- division by zero will crash the program and damage the space-time continuum. So, try not to divide by zero.

Integer Division



Bytes

ah al

op8

= al

rem ah

Words

dx ax

op16

= ax

rem dx

Double-words

edx eax

op32

= eax

rem edx

Integer Division

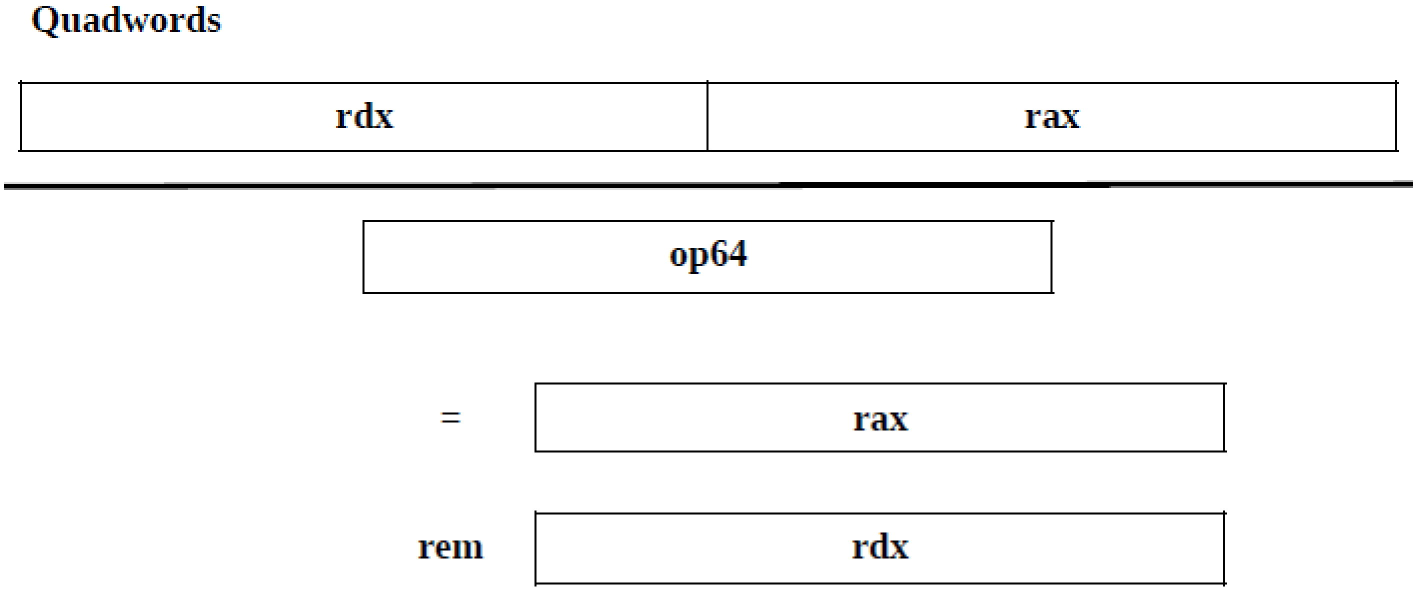


Illustration 16: Integer Division Overview

Integer Division

- The general forms of the unsigned and signed division are as follows:

```
div <src>        ; unsigned division
idiv <src>       ; signed division
```

- The source operand and destination operands (A and D registers) are described in the preceding table.

Example of Integer Division

Ex. Assuming the following data declarations:

```
bNumA   db   63
bNumB   db   17
bNumC   db   5
bAns1   db   0
bAns2   db   0
bRem2   db   0
bAns3   db   0

wNumA   dw   4321
wNumB   dw   1234
wNumC   dw   167
wAns1   dw   0
wAns2   dw   0
wRem2   dw   0
wAns3   dw   0

dNumA   dd   42000
dNumB   dd   -3157
dNumC   dd   -293
dAns1   dd   0
dAns2   dd   0
dRem2   dd   0
dAns3   dd   0

qNumA   dq   730000
qNumB   dq   -13456
qNumC   dq   -1279
qAns1   dq   0
qAns2   dq   0
qRem2   dq   0
qAns3   dq   0
```

Example of Integer Division

Question: To perform, the basic operations of:

```
bAns1 = bNumA / 3                        ; unsigned
bAns2 = bNumA / bNumB                    ; unsigned
bRem2 = bNumA % bNumB                    ; % is modulus
bAns3 = (bNumA * bNumC) / bNumB          ; unsigned
wAns1 = wNumA / 5                        ; unsigned
wAns2 = wNumA / wNumB                    ; unsigned
wRem2 = wNumA % wNumB                    ; % is modulus
wAns3 = (wNumA * wNumC) / wNumB          ; unsigned
dAns = dNumA / 7                         ; signed
dAns3 = dNumA * dNumB                    ; signed
dRem1 = dNumA % dNumB                    ; % is modulus
dAns3 = (dNumA * dNumC) / dNumB          ; signed
qAns = qNumA / 9                         ; signed
qAns4 = qNumA * qNumB                    ; signed
qRem1 = qNumA % qNumB                    ; % is modulus
qAns3 = (qNumA * qNumC) / qNumB          ; signed
```

Example of Integer Division

The following instructions could be used:

```
; -----
; example byte operations, unsigned

; bAns1 = bNumA / 3 (unsigned)
mov     al, byte [bNumA]
mov     ah, 0
mov     bl, 3
div     bl                          ; al = ax / 3
mov     byte [bAns1], al

; bAns2 = bNumA / bNumB (unsigned)
mov     ax, 0
mov     al, byte [bNumA]
div     byte [bNumB]                ; al = ax / bNumB
mov     byte [bAns2], al
mov     byte [bRem2], ah            ; ah = ax % bNumB

; bAns3 = (bNumA * bNumC) / bNumB (unsigned)
mov     al, byte [bNumA]
mul     byte [bNumC]                ; result in ax
div     byte [bNumB]                ; al = ax / bNumB
mov     byte [bAns3], al
```

Example of Integer Division

```
; -----
; example word operations, unsigned
; wAns1 = wNumA / 5 (unsigned)
mov     ax, word [wNumA]
mov     dx, 0
mov     bx, 5
div     bx ; ax = dx:ax / 5
mov     word [wAns1], ax

; wAns2 = wNumA / wNumB (unsigned)
mov     dx, 0
mov     ax, word [wNumA]
div     word [wNumB]                    ; ax = dx:ax / wNumB
mov     word [wAns2], ax
mov     word [wRem2], dx

; wAns3 = (wNumA * wNumC) / wNumB (unsigned)
mov     ax, word [wNumA]
mul     word [wNumC]                    ; result in dx:ax
div     word [wNumB]                    ; ax = dx:ax / wNumB
mov     word [wAns3], ax
```

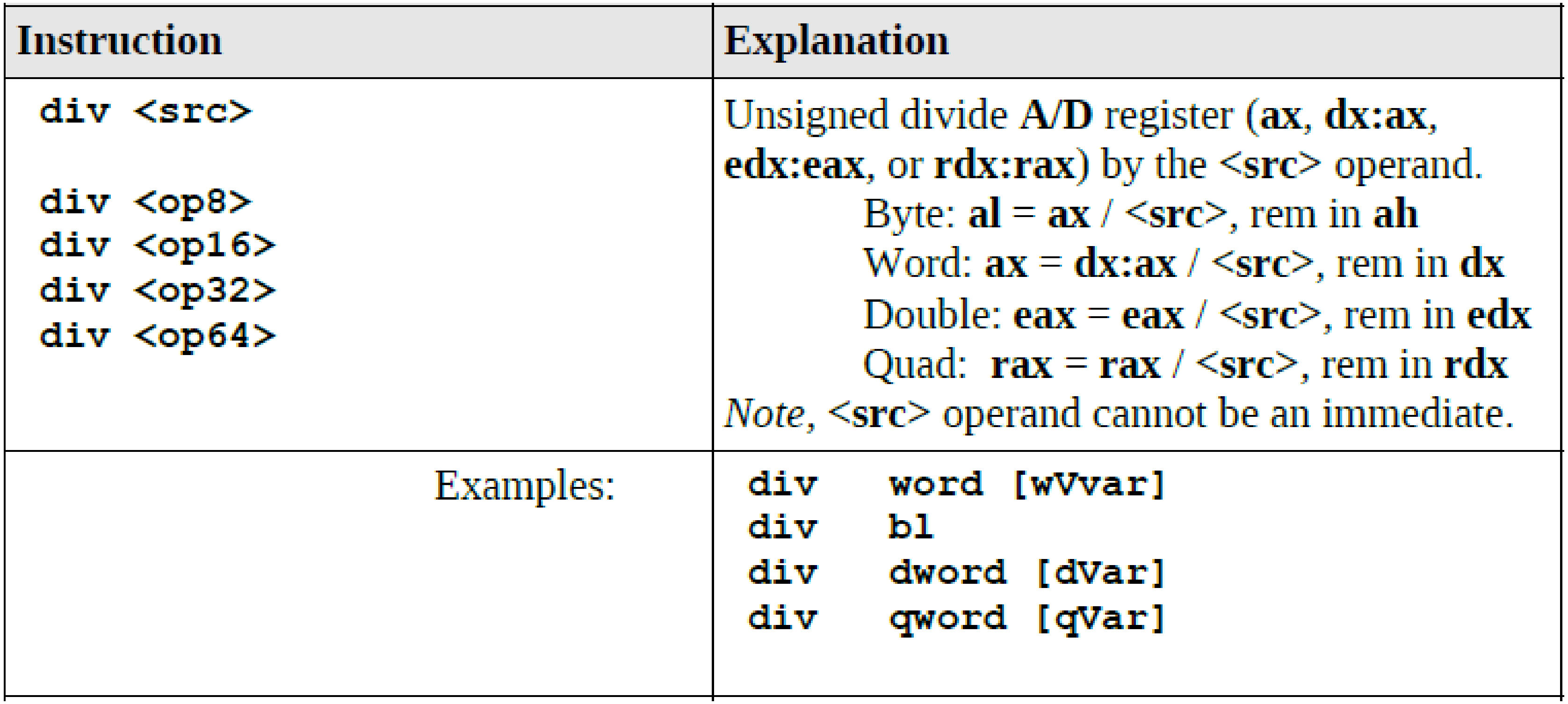
Example of Integer Division

```
; -----
; example double-word operations, signed
; dAns1 = dNumA / 7 (signed)
mov     eax, dword [dNumA]
cdq                                     ; eax → edx:eax
mov     ebx, 7
idiv    ebx                             ; eax = edx:eax / 7
mov     dword [dAns1], eax

; dAns2 = dNumA / dNumB (signed)
mov     eax, dword [dNumA]
cdq ; eax → edx:eax
idiv    dword [dNumB]                   ; eax = edx:eax/dNumB
mov     dword [dAns2], eax
mov     dword [dRem2], edx              ; edx = edx:eax%dNumB

; dAns3 = (dNumA * dNumC) / dNumB (signed)
mov     eax, dword [dNumA]
imul    dword [dNumC]                   ; result in edx:eax
idiv    dword [dNumB]                   ; eax = edx:eax/dNumB
mov     dword [dAns3], eax
```

Example of Integer Division

```
; -----
; example quadword operations, signed
; qAns1 = qNumA / 9 (signed)
mov     rax, qword [qNumA]
cqo                                     ; rax → rdx:rax
mov     rbx, 9
idiv    rbx                             ; eax = edx:eax / 9
mov     qword [qAns1], rax
; qAns2 = qNumA / qNumB (signed)
mov     rax, qword [qNumA]
cqo                                     ; rax → rdx:rax
idiv    qword [qNumB]                   ; rax = rdx:rax/qNumB
mov     qword [qAns2], rax
mov     qword [qRem2], rdx              ; rdx = rdx:rax%qNumB
; qAns3 = (qNumA * qNumC) / qNumB (signed)
mov     rax, qword [qNumA]
imul    qword [qNumC]                   ; result in rdx:rax
idiv    qword [qNumB]                   ; rax = rdx:rax/qNumB
mov     qword [qAns3], rax
```

Summary of div Instruction

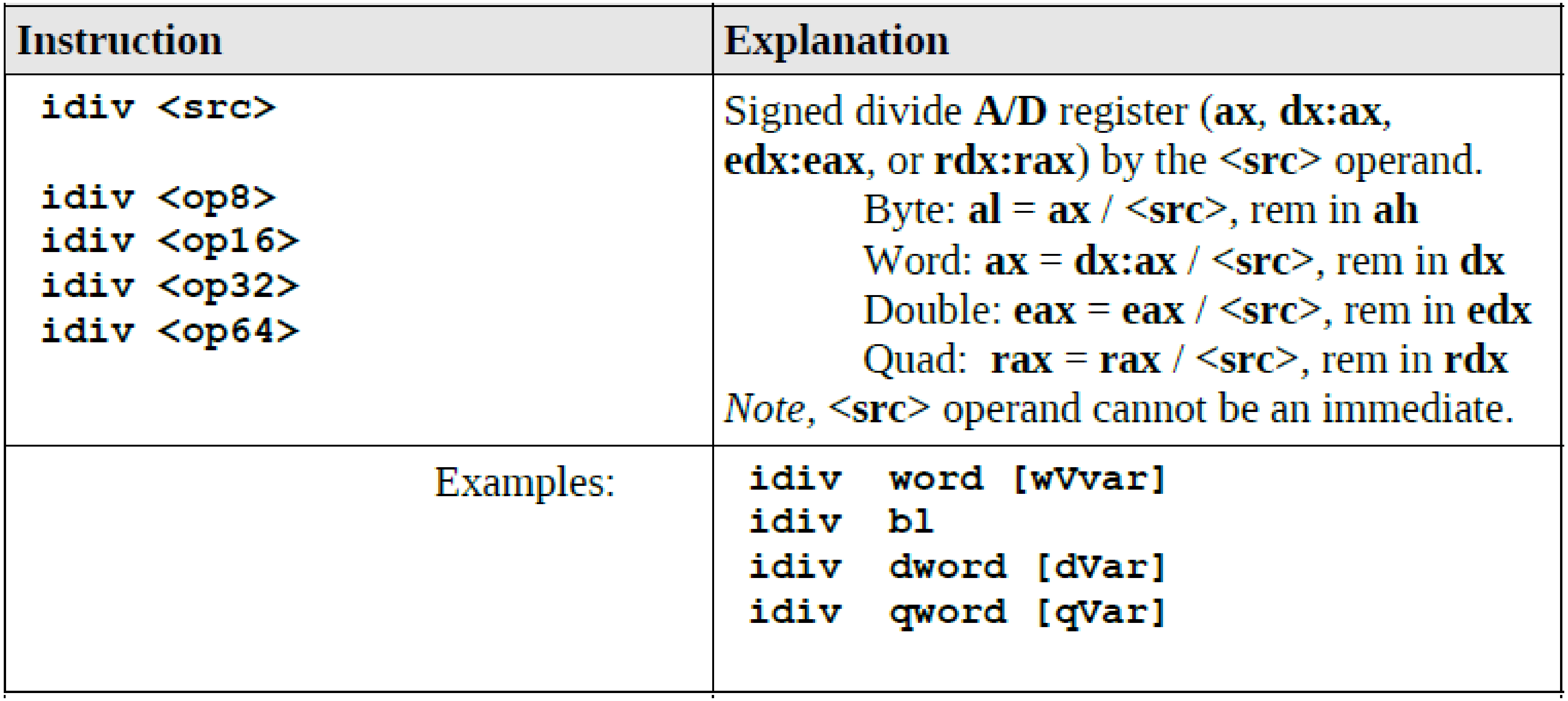


Instruction	Explanation

`div <src>`
 	
`div <op8>`
`div <op16>`
`div <op32>`
`div <op64>`
	Unsigned divide **A/D** register (**ax**, **dx:ax**, **edx:eax**, or **rdx:rax**) by the <**src**> operand.
	Byte: **al** = **ax** / <**src**>, rem in **ah**
	Word: **ax** = **dx:ax** / <**src**>, rem in **dx**
	Double: **eax** = **eax** / <**src**>, rem in **edx**
	Quad: **rax** = **rax** / <**src**>, rem in **rdx**
	Note, <**src**> operand cannot be an immediate.

Examples:	`div    word [wVvar]`
	`div    bl`
	`div    dword [dVar]`
	`div    qword [qVar]`

Summary of idiv Instruction



Instruction	Explanation

`idiv <src>`
 	
`idiv <op8>`
`idiv <op16>`
`idiv <op32>`
`idiv <op64>`
	Signed divide **A/D** register (**ax**, **dx:ax**, **edx:eax**, or **rdx:rax**) by the <**src**> operand.
	Byte: **al** = **ax** / <**src**>, rem in **ah**
	Word: **ax** = **dx:ax** / <**src**>, rem in **dx**
	Double: **eax** = **eax** / <**src**>, rem in **edx**
	Quad: **rax** = **rax** / <**src**>, rem in **rdx**
	Note, <**src**> operand cannot be an immediate.

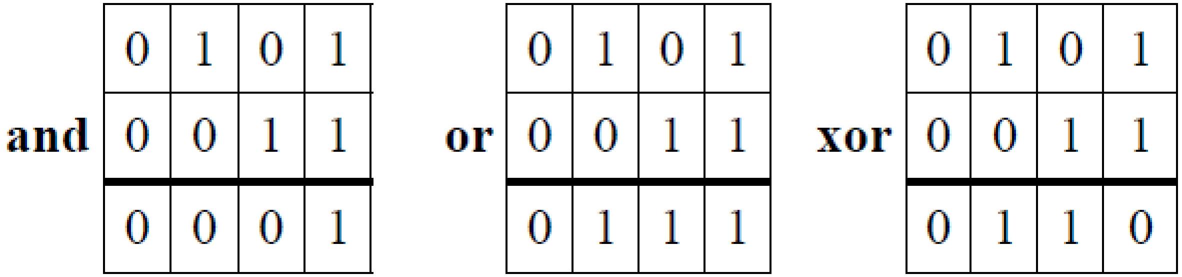
Examples:	`idiv word [wVvar]`
	`idiv bl`
	`idiv dword [dVar]`
	`idiv qword [qVar]`

Logical Instructions

Logical Operations

Logical Operations

- As you should recall, below are the truth tables for the basic logical operations;



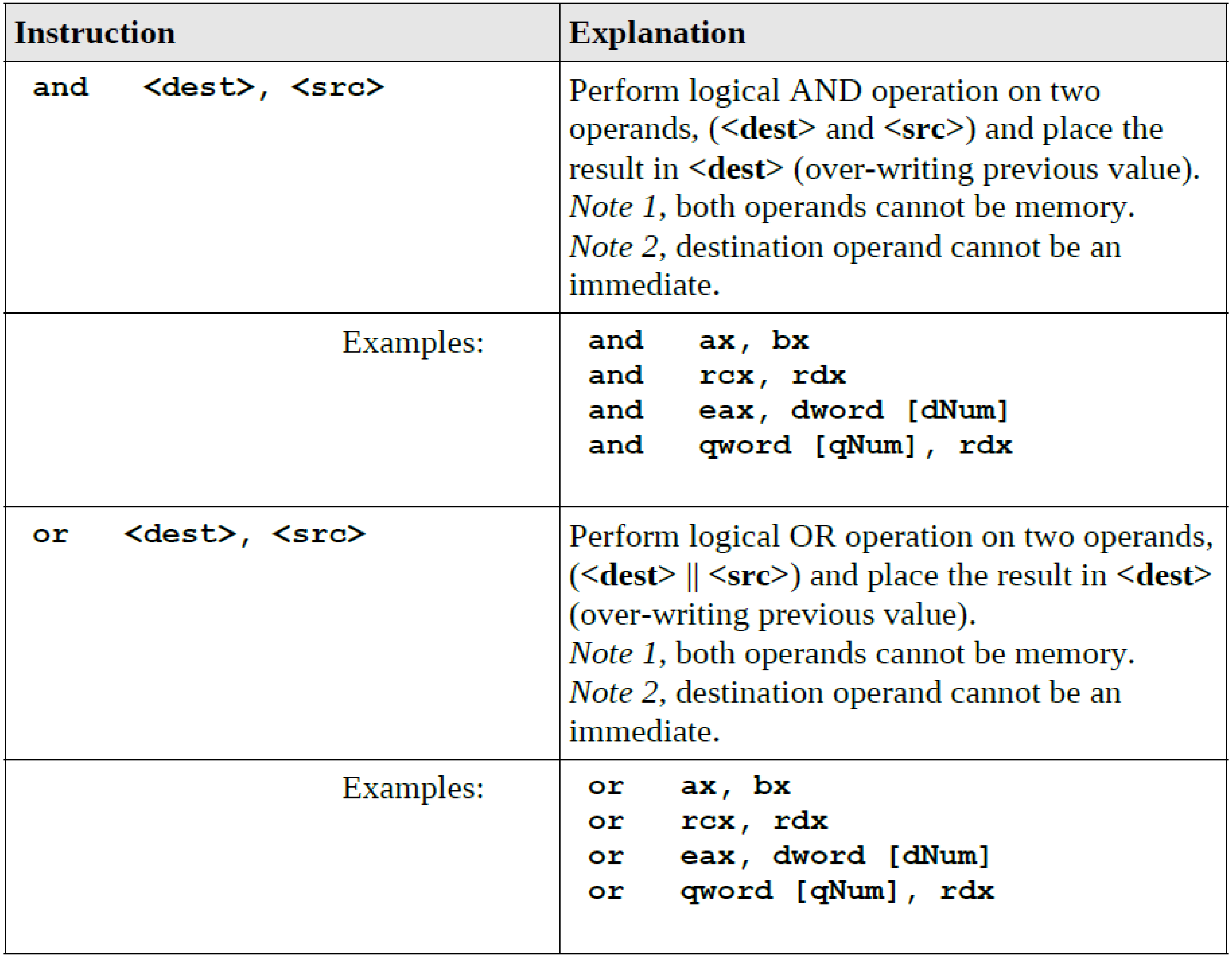
	0	1	0	1
and	0	0	1	1
	0	0	0	1

	0	1	0	1
or	0	0	1	1
	0	1	1	1

	0	1	0	1
xor	0	0	1	1
	0	1	1	0

Illustration 17: Logical Operations

Summary of Logical Instructions (1)



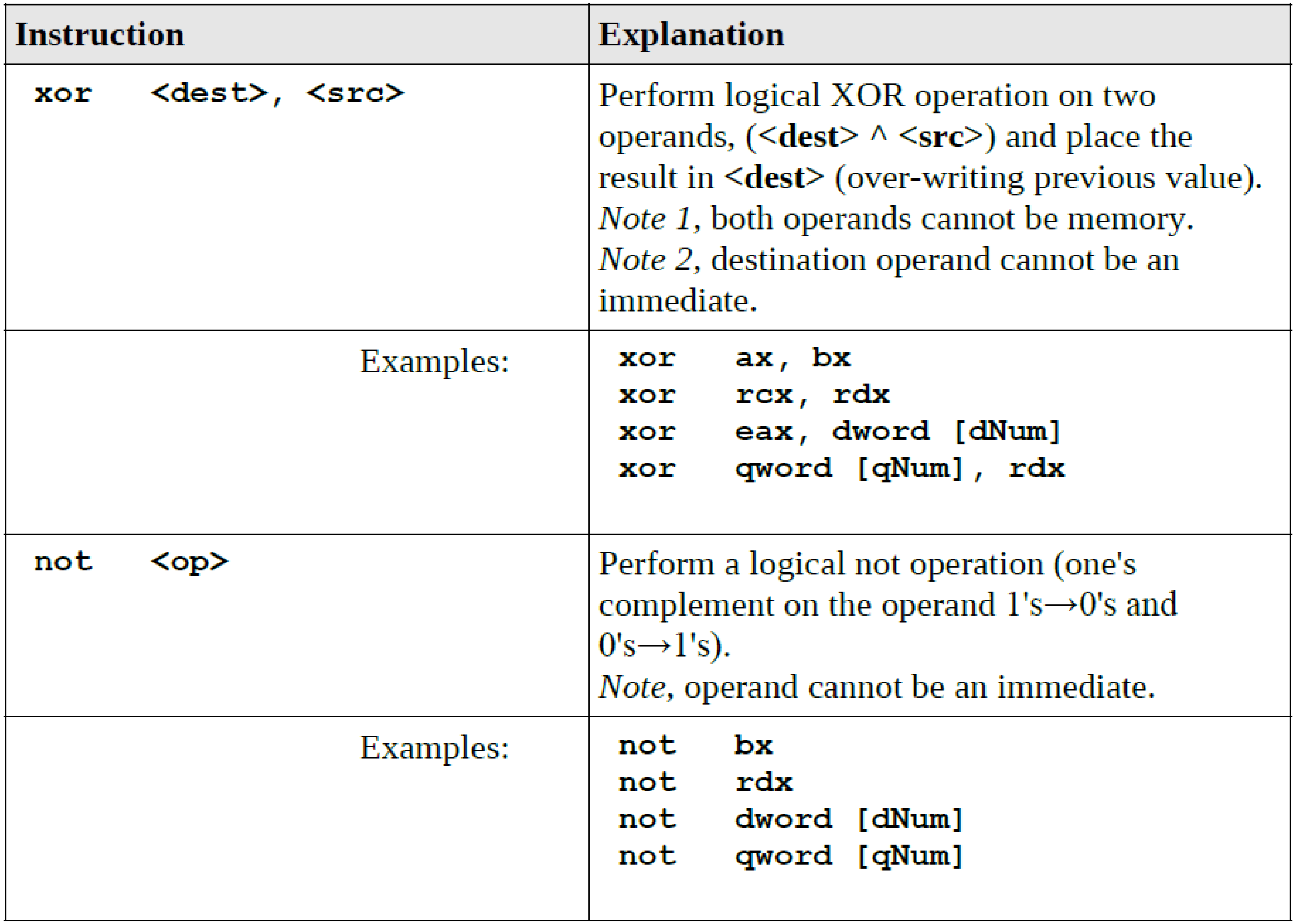
Instruction	Explanation
`and <dest>, <src>`	Perform logical AND operation on two operands, (<**dest**> and <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.

Examples:	`and ax, bx`
	`and rcx, rdx`
	`and eax, dword [dNum]`
	`and qword [qNum], rdx`

`or <dest>, <src>`	Perform logical OR operation on two operands, (<**dest**> || <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.

Examples:	`or ax, bx`
	`or rcx, rdx`
	`or eax, dword [dNum]`
	`or qword [qNum], rdx`

Summary of Logical Instructions (2)



Instruction	Explanation
`xor <dest>, <src>`	Perform logical XOR operation on two operands, (<**dest**> ^ <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.

Examples:	`xor ax, bx`
	`xor rcx, rdx`
	`xor eax, dword [dNum]`
	`xor qword [qNum], rdx`

`not <op>`	Perform a logical not operation (one's complement on the operand 1's→0's and 0's→1's). *Note*, operand cannot be an immediate.

Examples:	`not bx`
	`not rdx`
	`not dword [dNum]`
	`not qword [qNum]`

Logical Instructions

Shift Operations

Logical Shift

- The following diagram shows how the right and left shift operations work for byte sized operands.

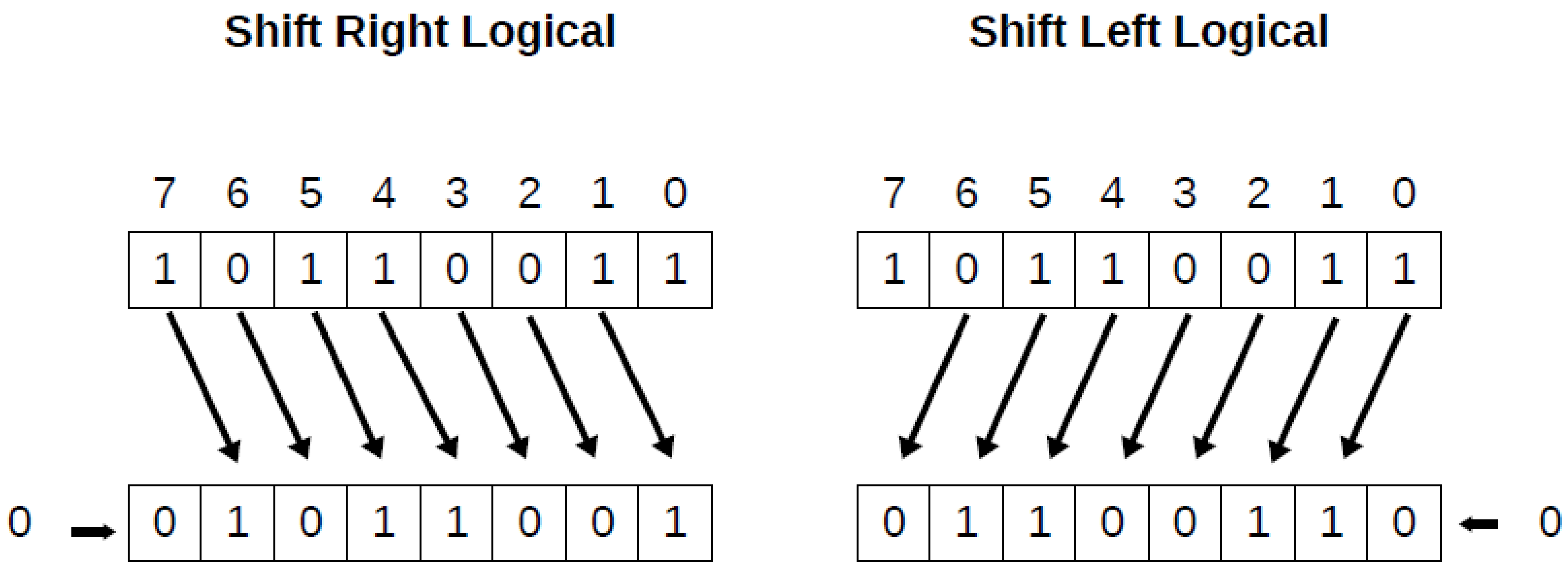
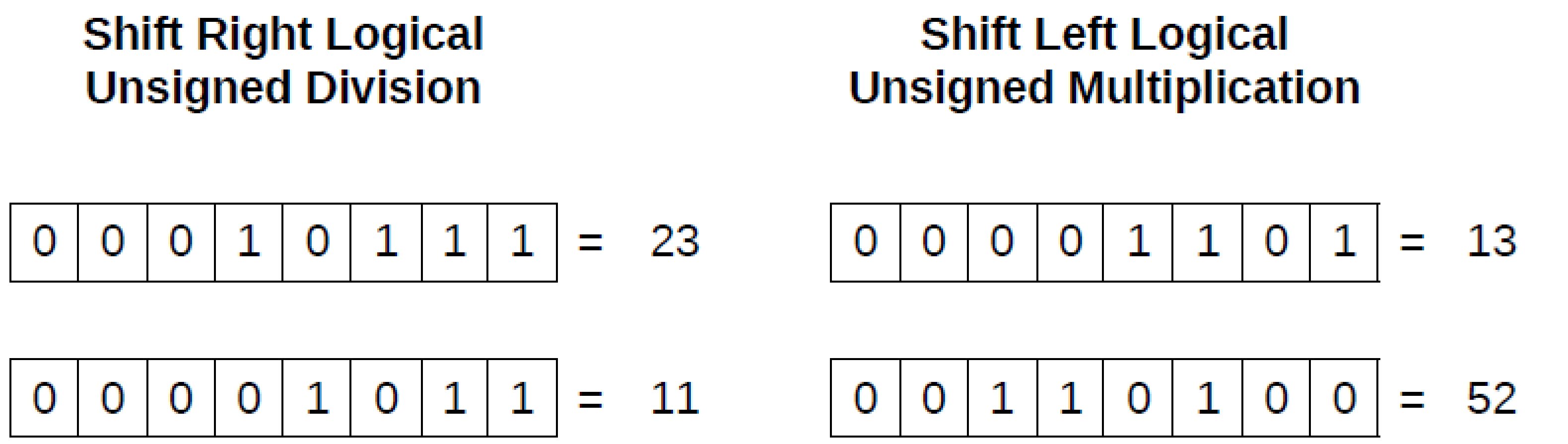


Illustration 18: Logical Shift Overview

Logical Shift

- In the examples below, 23 is divided by 2 by performing a shift right logical one bit. The resulting 11 is shown in binary.
- Next, 13 is multiplied by 4 by performing a shift left logical two bits. The resulting 52 is shown in binary.



Shift Right Logical
Unsigned Division

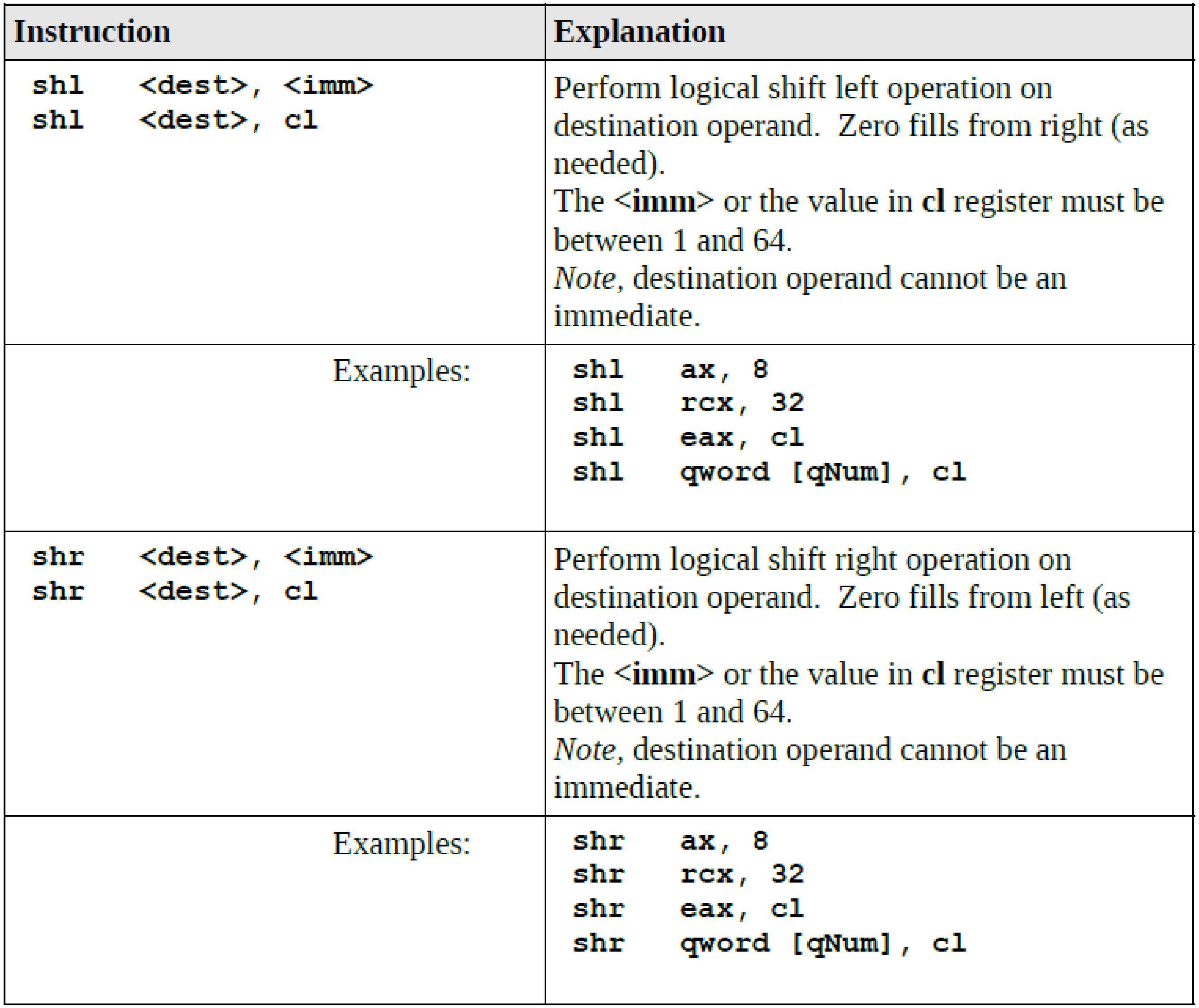
0	0	0	1	0	1	1	1	=	23
0	0	0	0	1	0	1	1	=	11

Shift Left Logical
Unsigned Multiplication

0	0	0	0	1	1	0	1	=	13
0	0	1	1	0	1	0	0	=	52

Illustration 19: Logical Shift Operations

Summary of Logical Shift



Instruction	Explanation

`shl <dest>, <imm>`
`shl <dest>, cl`
	Perform logical shift left operation on destination operand. Zero fills from right (as needed).
	The <**imm**> or the value in **cl** register must be between 1 and 64.
	Note, destination operand cannot be an immediate.

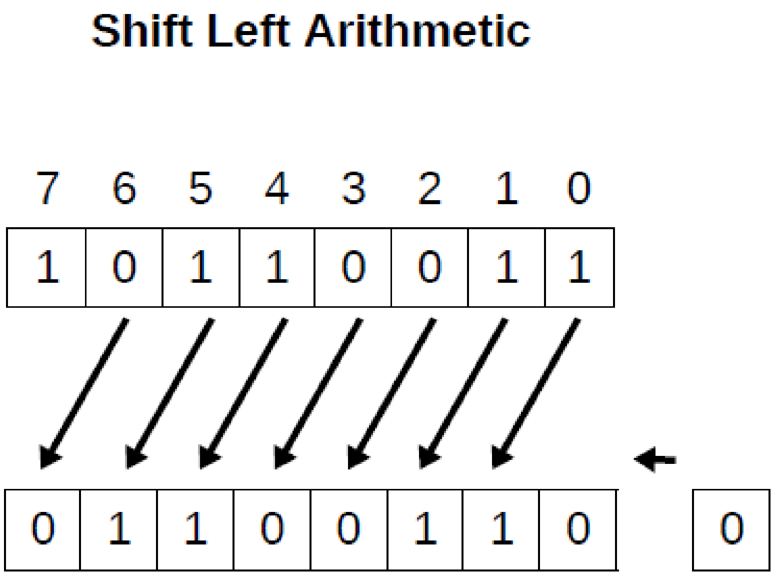
Examples:	`shl ax, 8`
	`shl rcx, 32`
	`shl eax, cl`
	`shl qword [qNum], cl`

`shr <dest>, <imm>`
`shr <dest>, cl`
	Perform logical shift right operation on destination operand. Zero fills from left (as needed).
	The <**imm**> or the value in **cl** register must be between 1 and 64.
	Note, destination operand cannot be an immediate.

Examples:	`shr ax, 8`
	`shr rcx, 32`
	`shr eax, cl`
	`shr qword [qNum], cl`

Arithmetic Shift

- The following diagrams show how the shift left and shift right arithmetic operations works for a byte sized operand.



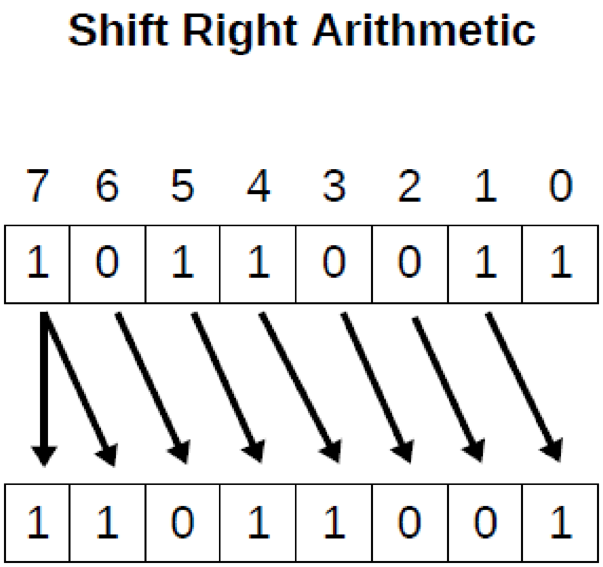
Shift Left Arithmetic

7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0

Illustration 20: Arithmetic Left Shift

Arithmetic Shift

- The arithmetic left shift moves bits the number of specified places to the left and zero fills the least significant bit.
- The arithmetic right shift moves bits the number of specified places to the right and treats the operand as a signed number which extends the sign (negative in this example).

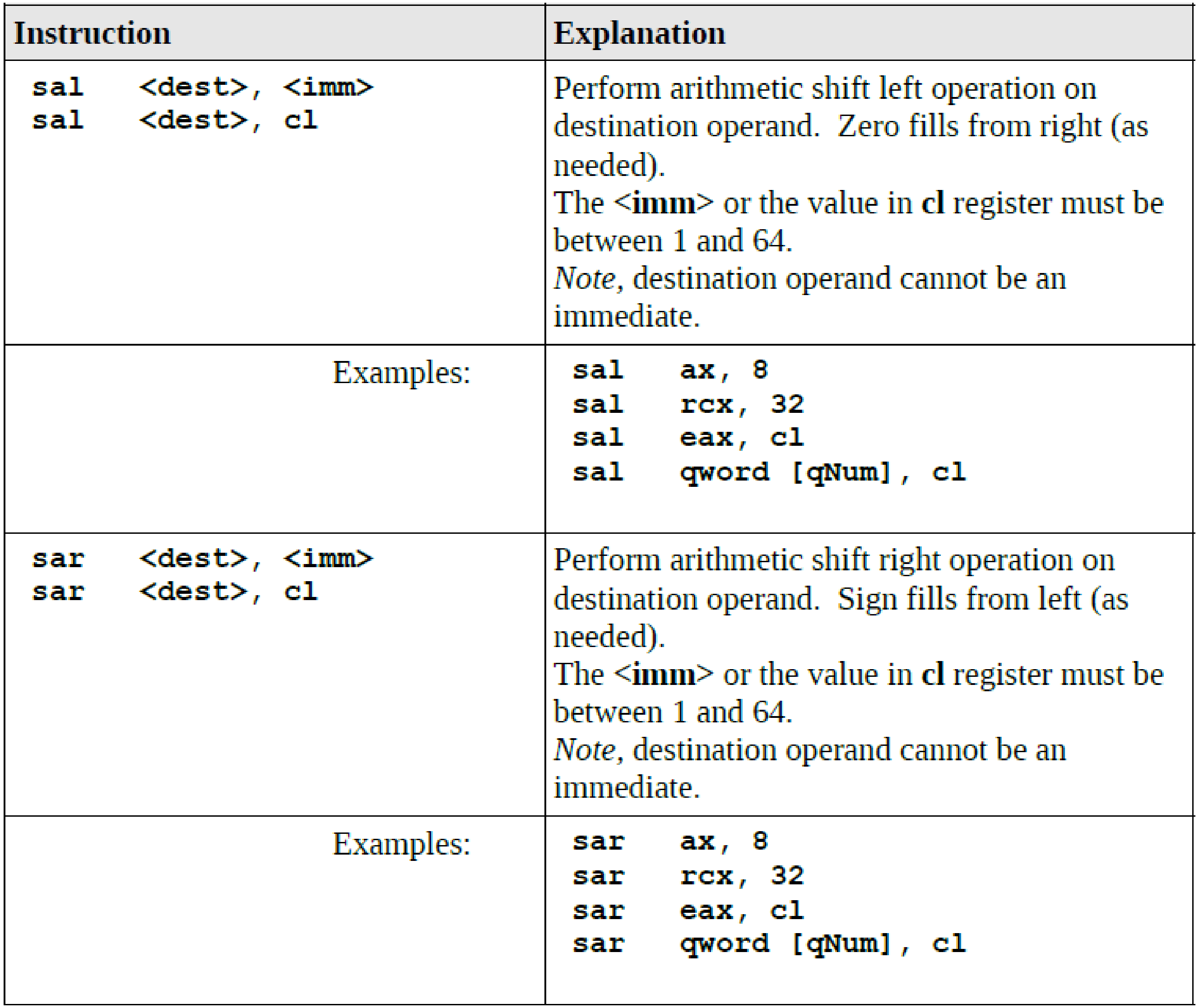


Shift Right Arithmetic

7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	1
1	1	0	1	1	0	0	1

Illustration 21: Arithmetic Right Shift

Summary of Arithmetic Shift



Instruction	Explanation

`sal    <dest>, <imm>`
`sal    <dest>, cl`
	Perform arithmetic shift left operation on destination operand. Zero fills from right (as needed).
	The <**imm**> or the value in **cl** register must be between 1 and 64.
	Note, destination operand cannot be an immediate.

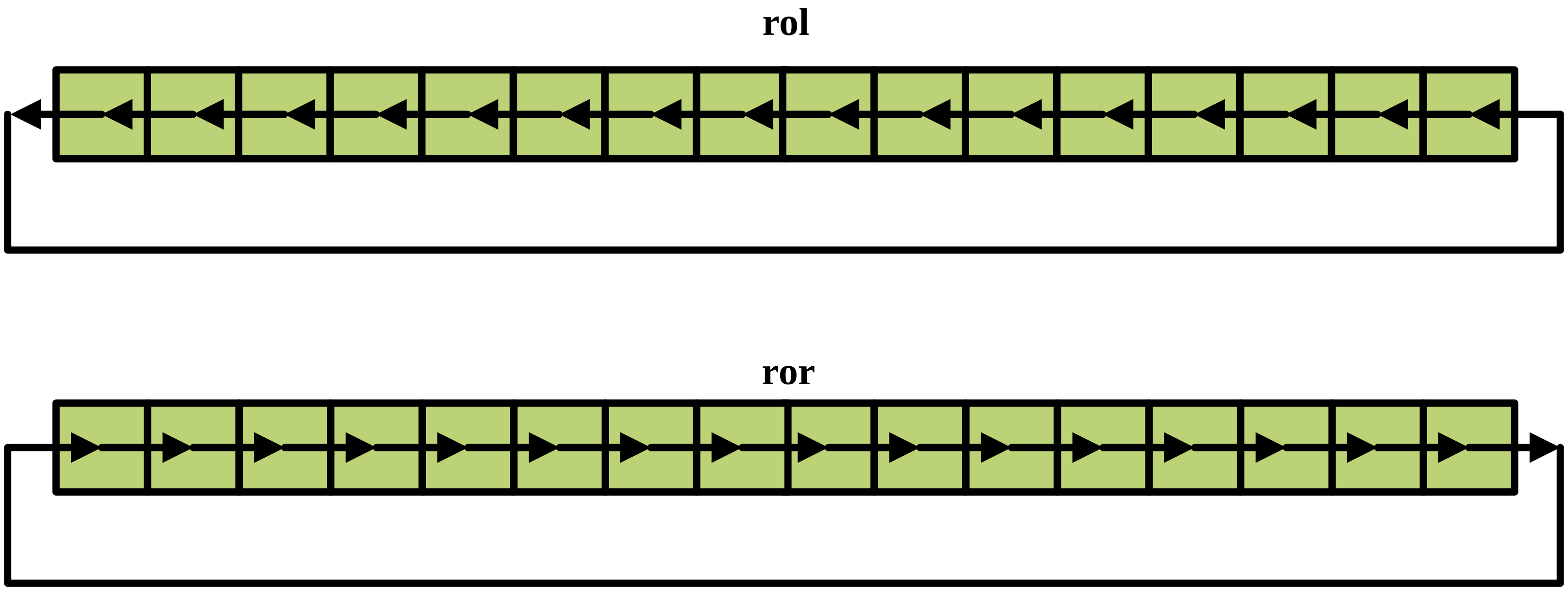
Examples:	`sal    ax, 8`
	`sal    rcx, 32`
	`sal    eax, cl`
	`sal    qword [qNum], cl`

`sar    <dest>, <imm>`
`sar    <dest>, cl`
	Perform arithmetic shift right operation on destination operand. Sign fills from left (as needed).
	The <**imm**> or the value in **cl** register must be between 1 and 64.
	Note, destination operand cannot be an immediate.

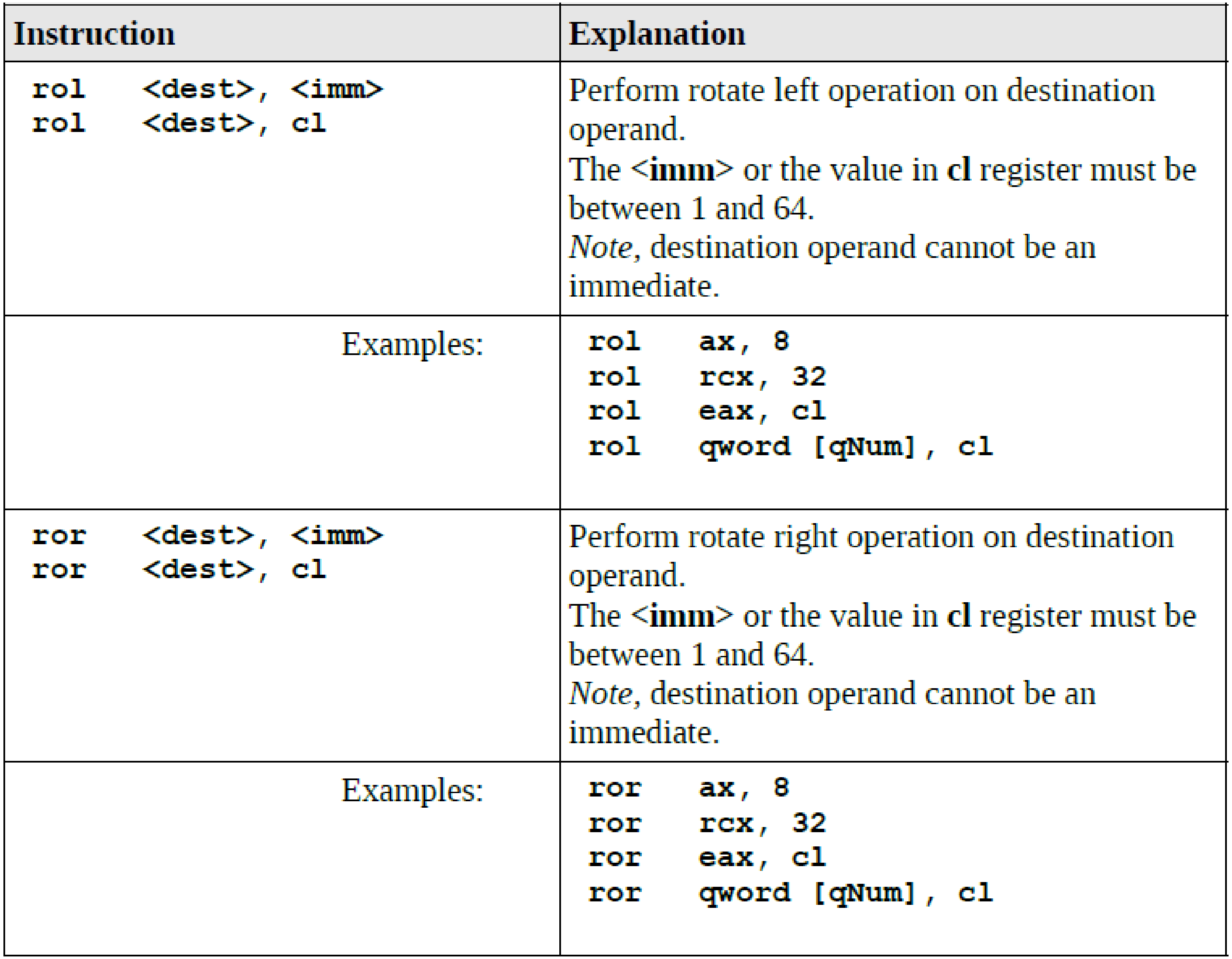
Examples:	`sar    ax, 8`
	`sar    rcx, 32`
	`sar    eax, cl`
	`sar    qword [qNum], cl`

Rotate Operations

- The rotate operation shifts bits within an operand, either left or right, with the bit that is shifted outside the operand is rotated around and placed at the other end.



Summary of Rotate Operations



Instruction	Explanation

`rol    <dest>, <imm>`
`rol    <dest>, cl`
	Perform rotate left operation on destination operand.
	The <**imm**> or the value in **cl** register must be between 1 and 64.
	Note, destination operand cannot be an immediate.

Examples:	`rol    ax, 8`
	`rol    rcx, 32`
	`rol    eax, cl`
	`rol    qword [qNum], cl`

`ror    <dest>, <imm>`
`ror    <dest>, cl`
	Perform rotate right operation on destination operand.
	The <**imm**> or the value in **cl** register must be between 1 and 64.
	Note, destination operand cannot be an immediate.

Examples:	`ror    ax, 8`
	`ror    rcx, 32`
	`ror    eax, cl`
	`ror    qword [qNum], cl`

Control Instructions

Control Instructions

- Program control refers to basic programming structures such as IF statements and looping.
- Assembly language provides an unconditional branch (or jump) and a conditional branch or an IF statement that will jump to a target label or not jump.

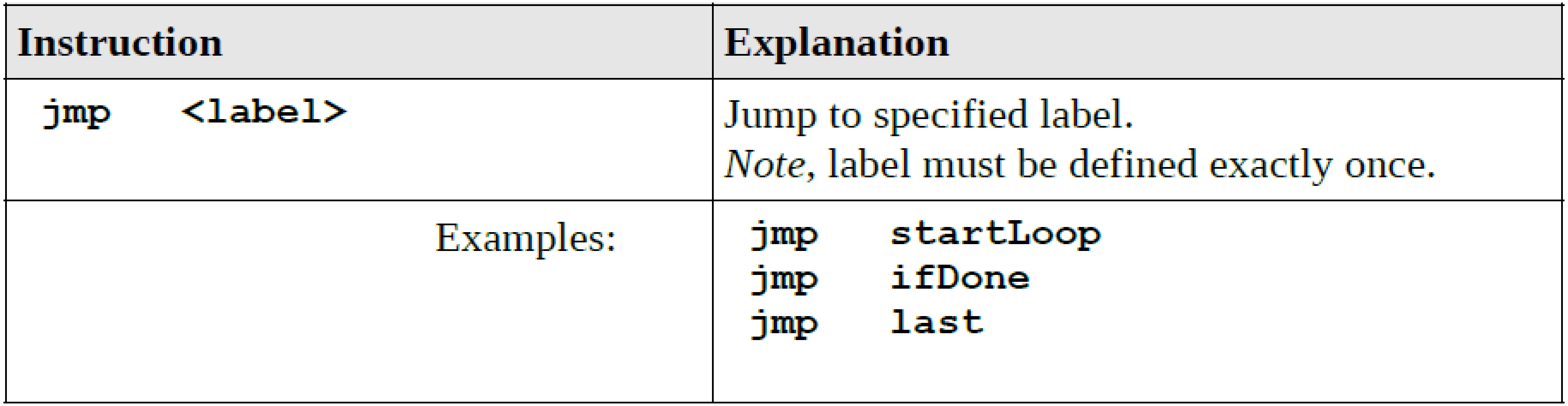
Labels

- A program label is the target, or a location to jump to, for control statements.
- Generally, a label starts with a letter, followed by letters, numbers, or symbols (limited to "_"), terminated with a colon (":").
- Labels in **yasm** are case sensitive.
- For example,

```
loopStart:
last:
```

Unconditional Control Instructions

- The unconditional instruction provides an unconditional jump to a specific location in the program denoted with a program label. The target label must be defined exactly once and accessible and within scope from the originating jump instruction.



Instruction	Explanation
`jmp    <label>`	Jump to specified label. *Note*, label must be defined exactly once.

Examples:	`jmp    startLoop`
	`jmp    ifDone`
	`jmp    last`

Conditional Control Instructions

- The conditional jump instruction will act (jump or not jump) based on the contents of the **rFlag** register.
- The general form of the compare instruction is:

```
        cmp <op1>, <op2>
```

- Where **<op1>** and **<op2>** are not changed and must be of the same size.
- Either, but not both, may be a memory operand.
- The **<op1>** operand cannot be an immediate, but the **<op2>** operand may be an immediate value.

Conditional Control Instructions

- The general form of the signed conditional instructions along with an explanatory comment are as follows:

```
je      <label>         ; if <op1> == <op2>
jne     <label>         ; if <op1> != <op2>
jl      <label>         ; signed, if <op1> < <op2>
jle     <label>         ; signed, if <op1> <= <op2>
jg      <label>         ; signed, if <op1> > <op2>
jge     <label>         ; signed; if <op1> >= <op2>
jb      <label>         ; unsigned, if <op1> < <op2>
jbe     <label>         ; unsigned, if <op1> <= <op2>
ja      <label>         ; unsigned, if <op1> > <op2>
jae     <label>         ; unsigned, if <op1> >= <op2>
```

Conditional Control Instructions

- Ex1. given the following pseudo-code for signed data:

```
    if (currNum > myMax)
            myMax = currNum;
```

- Assuming the following data declarations:

```
    currNum        dq        0
    myMax          dq        0
```

- The following instructions could be used:

```
    mov   rax, qword [currNum]
    cmp   rax, qword [myMax]        ; if currNum <= myMax
    jle   notNewMax                 ; skip set new max
    mov   qword [myMax], rax
notNewMax:
```

Conditional Control Instructions

- Ex2. A more complex example might be as follows:

```
if (x != 0) {
        ans = x / y;
        errFlg = FALSE;
} else {
        ans = 0;
        errFlg = TRUE;
}
```

- Assuming the following data declarations:

```
TRUE      equu   1
FALSE     equ    0
x         dd     0
y         dd     0
ans       dd     0
errFlg    db     FALSE
```

Conditional Control Instructions

- The following code could be used to implement the above IF-ELSE statement.

```
    cmp   dword [x], 0            ; if statement
    je    doElse
    mov   eax, dword [x]          ; {
    cdq                           ; convert eax to rax
    idiv  dword [y]
    mov   dword [ans], eax
    mov   byte [errFlg], FALSE
    jmp   skpElse                 ; }
doElse:                           ; else
    mov   dword [ans], 0          ; {
    mov   byte [errFlg], TRUE
skpElse:                          ; }
```

Jump Out of Range

- The target label must be within ±128 bytes from the conditional jump instruction.
- While this limit is not typically a problem, for very large loops, the assembler may generate an error referring to “jump out-of-range”.
- The unconditional jump (**jmp**) is not limited in range.

Jump Out of Range

- If a “jump out-of-range” is generated, it can be eliminated by reversing the logic and using an unconditional jump for the long jump. For example, the following code:

```
        cmp    rcx, 0
        jne    startOfLoop
```

- might generate a “jump out-of-range” assembler error if the label, ***startOfLoop***, is a long distance away. The error can be eliminated with the following code:

```
        cmp    rcx, 0
        je     endOfLoop
        jmp    startOfLoop
endOfLoop:
```

Summary of Jump Instructions (1)

Instruction	Explanation	Examples
cmp <op1>, <op2>	Compare **<op1>** with **<op2>**. Results are stored in the **rFlag** register. *Note 1*, operands are not changed. *Note 2*, both operands cannot be memory. *Note 3*, **<op1>** operand cannot be an immediate.	**cmp rax, 5** **cmp ecx, edx** **cmp ax, word [wNum]**
je <label>	Based on preceding comparison instruction, jump to **<label>** if **<op1> == <op2>**. Label must be defined exactly once.	**cmp rax, 5** **je wasEqual**
jne <label>	Based on preceding comparison instruction, jump to **<label>** if **<op1> != <op2>**. Label must be defined exactly once.	**cmp rax, 5** **jne wasNotEqual**
jl <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1> < <op2>**. Label must be defined exactly once.	**cmp rax, 5** **jl wasLess**

Summary of Jump Instructions (2)

Instruction	Explanation	Examples
jle <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1>** <= **<op2>**. Label must be defined exactly once.	**cmp rax, 5** **jle wasLessOrEqual**
jg <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1> > <op2>**. Label must be defined exactly once.	**cmp rax, 5** **jg wasGreater**
jge <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1>** >= **<op2>**. Label must be defined exactly once.	**cmp rax, 5** **Jge wasGreaterOrEqual**
jb <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1> < <op2>**. Label must be defined exactly once.	**cmp rax, 5** **jl wasLess**

Summary of Jump Instructions (3)

Instruction	Explanation	Examples
jbe <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** <= **<op2>**. Label must be defined exactly once.	**cmp rax, 5** **jbe wasLessOrEqual**
ja <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1> > <op2>**. Label must be defined exactly once.	**cmp rax, 5** **ja wasGreater**
jae <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** >= **<op2>**. Label must be defined exactly once.	**cmp rax, 5** **jae wasGreaterOrEqual**

Iteration

- A basic loop can be implemented consisting of a counter which is checked at either the bottom or top of a loop with a compare and conditional jump.

Iteration

Ex1. Assuming the following declarations:

```
    lpCnt   dq      15
    sum     dq      0
```

The following code would sum the odd integers from 1 to 30:

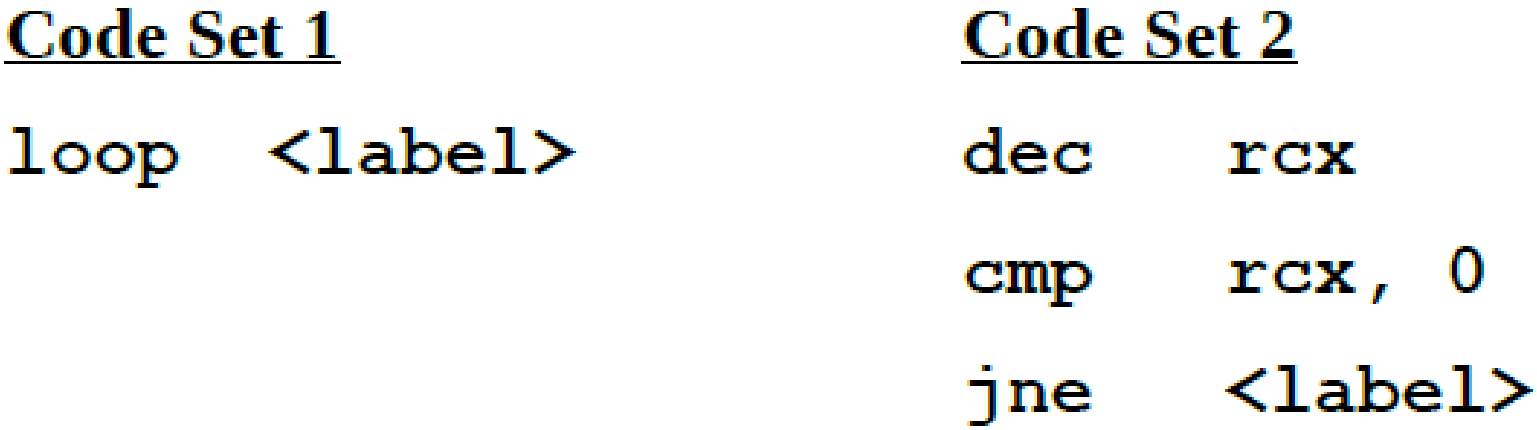
```
    mov   rcx, qword [lpCnt]    ; loop counter
    mov   rax, 1                ; odd integer counter
  sumLoop:
    add   qword [sum], rax  ; sum current odd integer
    add   rax, 2                ; set next odd integer
    dec   rcx                 ; decrement loop counter
    cmp   rcx, 0
    jne   sumLoop
```

General Format of Iteration

- The general format is as follows:

  loop <label>

- The following sets of code are equivalent:



Code Set 1

```
loop   <label>
```

Code Set 2

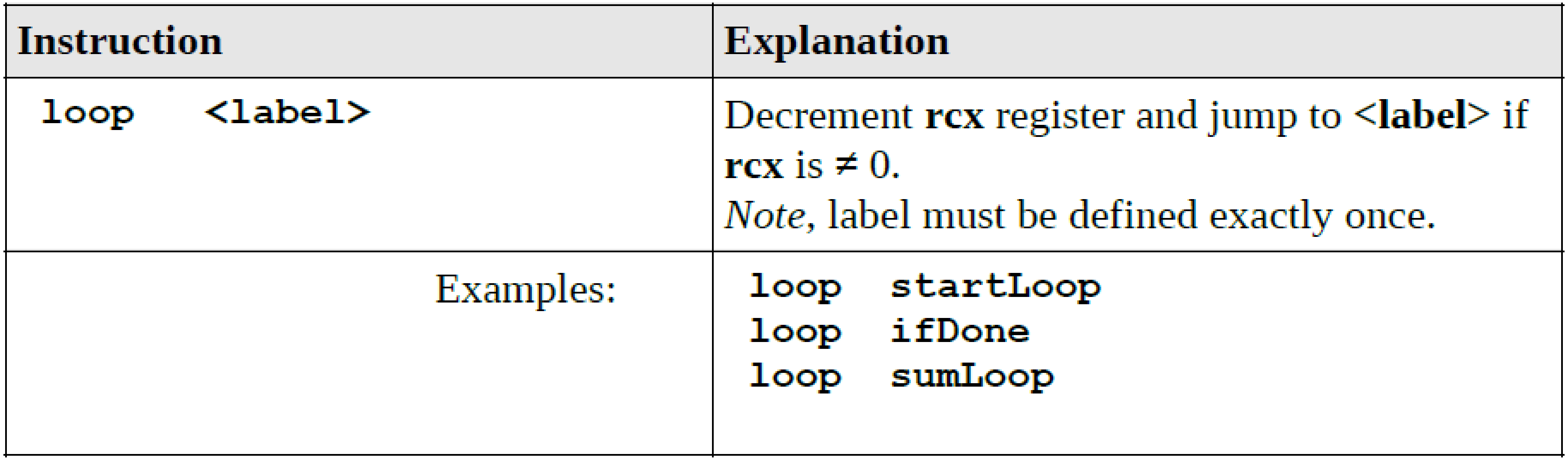
```
dec    rcx
cmp    rcx, 0
jne    <label>
```

General Format of Iteration

- Ex2. The previous program can be written as follows:

```
    mov  rcx, qword [maxN]    ; loop counter
    mov  rax, 1               ; odd integer counter
 sumLoop:
    add   qword [sum], rax  ; sum current odd integer
    add   rax, 2              ; set next odd integer
    loop  sumLoop
```

Summary of loop Instruction



Instruction	Explanation
`loop <label>`	Decrement **rcx** register and jump to **<label>** if **rcx** is ≠ 0. *Note*, label must be defined exactly once.

Examples:	`loop startLoop`
	`loop ifDone`
	`loop sumLoop`

Example Program, Sum of Squares

Example: $1^2 + 2^2 + \cdots + 10^2 = 385$

```
; Simple example program to compute the
; sum of squares from 1 to n.
; ****************************************
; Data declarations
section   .data
; -----
; Define constants
SUCCESS         equu    0               ; Successful operation
SYS_exit        equ     60              ; call code for terminate
; Define Data.
n               dd      10
sumOfSquares    dq      0
```

Example: $1^2 + 2^2 + \cdots + 10^2 = 385$

```
; *******************************************************
section .text
global _start
_start:
; -----
; Compute sum of squares from 1 to n (inclusive).
; Approach:
; for (i=1; i<=n; i++)
; sumOfSquares += i^2;
        mov     rbx, 1                          ; i
        mov     ecx, dword [n]
sumLoop:
        mov     rax, rbx                        ; get i
        mul     rax                             ; i^2
        add     qword [sumOfSquares], rax
        inc     rbx
        loop    sumLoop
```

Example: $1^2 + 2^2 + \cdots + 10^2 = 385$

```
; -----
; Done, terminate program.
last:
        mov     rax, SYS_exit                   ; call code for exit
        mov     rdi, SUCCESS                    ; exit with success
        syscall
```

End of Chapter 7