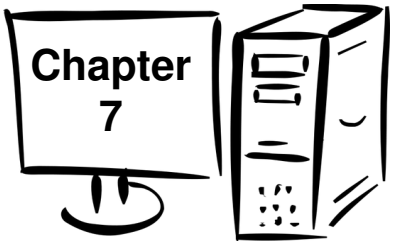
Why are math books sad?
Because they have so many problems.



7.0 Instruction Set Overview

This chapter provides a basic overview for a simple subset of the x86-64 instruction set focusing on the integer operations. This will cover only the subset of instructions required for the topics and programs discussed within the scope of this text. This will exclude some of the more advanced instructions and restricted mode instructions. For a complete listing of all processor instructions, refer to the references listed in Chapter 1.

The instructions are presented in the following order:

- Data Movement
- Conversion Instructions
- Arithmetic Instructions
- Logical Instructions
- Control Instructions

The instructions for function calls are discussed in the chapter in Chapter 12, Functions.

A complete listing of the instructions covered in this text is located in Appendix B for reference.

7.1 Notational Conventions

This section summarizes the notation used within this text which is fairly common in the technical literature. In general, 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.

7.1.1 Operand Notation

The following table summarizes the notational conventions used in the remainder of the document.

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.

By default, the immediate values are decimal or base-10. Hexadecimal or base-16 immediate values may be used but must be preceded with a **0x** to indicate the value is hex. For example, 15_{10} could be entered in hex as **0x0F**.

Refer to Chapter 8, Addressing Modes for more information regarding memory locations and indirection.

7.2 Data Movement

Typically, data must be moved into a CPU register from RAM in order to be operated upon. Once the calculations are completed, the result may be copied from the register and placed into a variable. There are a number of simple formulas in the example program that perform these steps. This basic data movement operation is performed with the move instruction.

The general form of the move instruction is:

```
mov   <dest>, <src>
```

The source operand is copied from the source operand into the destination operand. 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. Both operands cannot be memory. If a memory to memory operation is required, two instructions must be used.

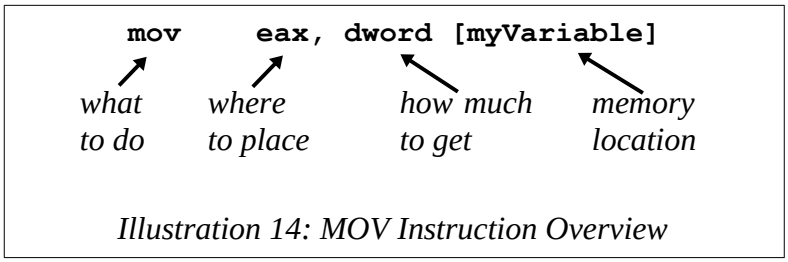


Illustration 14: MOV Instruction Overview

When the destination register operand is of double-word size and the source operand is of double-word size, the upper-order double-word of the quadword register is set to zero. This only applies when the destination operand is a double-word sized integer register.

Specifically, if the following operations are performed,

```
mov     eax, 100            ; eax = 0x00000064
mov     rcx, -1             ; rcx = 0xffffffffffffffff
mov     ecx, eax            ; ecx = 0x00000064
```

Initially, the **rcx** register is set to -1 (which is all 0xF's). When the positive number from the **eax** register (100_{10}) is moved into the **rcx** register, the upper-order portion of

the quadword register **rcx** is set to 0 over-writing the 1's from the previous instruction.

The move instruction is summarized as follows:

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`

A more complete list of the instructions is located in Appendix B.

For example, 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
```

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
```

For some instructions, including those above, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) can be omitted as the other operand will clearly define the size. In the text it will be included for consistency and good programming practice.

7.3 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
```

Since omitting the brackets is not an error, the assembler will not generate error messages or warnings. This can lead to confusion.

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]`

A more complete list of the instructions is located in Appendix B.

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

7.4 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.

7.4.1 Narrowing Conversions

Narrowing conversions are converting from a larger type to a smaller type (i.e., word to byte or double-word to word).

No special instructions are needed for narrowing conversions. The lower portion of the memory location or register may be accessed directly. For example, 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
```

This example is reasonable since the value of 50 will fit in a byte value. However, 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. The programmer is responsible for ensuring that narrowing conversions are performed appropriately. Unlike a compiler, no warnings or error messages will be generated.

7.4.2 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.

7.4.2.1 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. For example, 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
```

Since the **rbx** register was set to 0 and then the lower 8-bits were set to the value from **al** (50 in this example), the entire 64-bit **rbx** register is now 50.

This general process can be performed on memory or other registers. It is the programmer's responsibility to ensure that the values are appropriate for the data sizes being used.

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.

A summary of the instructions that perform the unsigned widening conversion are as follows:

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`

A more complete list of the instructions is located in Appendix B.

7.4.2.2 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.

This is performed by a sign-extend operation. Specifically, the upper-order bit of the original value indicates if the value is positive (with a 0) or negative (with a 1). The upper-order bit of the original value is extended into the higher bits of the new, widened value.

For example, 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**

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**

There are a series of dedicated instructions used to convert signed values in the **A** register from a smaller size into a larger size. These instructions work only on the **A** register, sometimes using the **D** register for the result. For example, the **cwd** instruction will convert a signed value in the **ax** register into a double-word value in the **dx** (upper-order portion) and **ax** (lower-order portion) registers. This is typically by convention written as **dx:ax**. The **cwde** instruction will convert a signed value in the **ax** register into a double-word value in the **eax** register.

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>
```

Which will perform the sign extension operation on the source argument. 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.

A summary of the instructions that perform the signed widening conversion are as follows:

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

Instruction	Explanation
`cqo`	Convert quadword in **rax** into word in double-quadword in **rdx:rax**. *Note*, only works for **rax** to **rdx:rax** registers.
Examples:	`cqo`

`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]`

A more complete list of the instructions is located in Appendix B.

7.5 Integer Arithmetic Instructions

The integer arithmetic instructions perform arithmetic operations such as addition, subtraction, multiplication, and division on integer values. The following sections present the basic integer arithmetic operations.

7.5.1 Addition

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

```
add   <dest>, <src>
```

Where operation performs the following:

```
<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. Both operands, cannot be memory. If a memory to memory addition operation is required, two instructions must be used.

For example, 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
```

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
```

For some instructions, including those above, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) can be omitted as the other operand will clearly define the size. It is included for consistency and good programming practice.

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>
```

Where operation is as follows:

```
<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.

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
```

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
```

The addition instruction operates the same on signed and unsigned data. It is the programmer's responsibility to ensure that the data types and sizes are appropriate for the operations being performed.

The integer addition instructions are summarized as follows:

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]`

A more complete list of the instructions is located in Appendix B.

7.5.1.1 Addition with Carry

The add with carry is a special add instruction that will include a carry from a previous addition operation. This is useful when adding very large numbers, specifically numbers larger than the register size of the machine.

Using a carry in addition is fairly standard. For example, consider the following operation.

```
  17
+ 25
----
  42
```

As you may recall, the least significant digits (7 and 5) are added first. The result of 12 is noted as a 2 with a 1 carry. The most significant digits (1 and 2) are added along with the previous carry (1 in this example) resulting in a 4.

As such, two addition operations are required. Since there is no carry possible with the least significant portion, a regular addition instruction is used. The second addition operation would need to include a possible carry from the previous operation and must be done with an add with carry instruction. Additionally, the add with carry must immediately follow the initial addition operation to ensure that the **rFlag** register is not altered by an unrelated instruction (thus possibly altering the carry bit).

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>
```

Where operation performs the following:

```
<dest> = <dest> + <src> + <carryBit>
```

Specifically, the source and destination operands along with the carry bit are added and the result is placed in the destination operand (over-writing the previous value). The carry bit is part of the **rFlag** register. 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. Both operands, cannot be memory. If a memory to memory addition operation is required, two instructions must be used.

For example, given the following declarations;

```
dquad1      ddq     0x1A000000000000000
```

```
dquad2      ddq     0x2C000000000000000
dqSum       ddq     0
```

Each of the variables ***dquad1***, ***dquad2***, and ***dqSum*** are 128-bits and thus will exceed the machine 64-bit register size. However, two 64-bit registers can be used for each of the 128-bit values. This requires two move instructions, one for each 64-bit register. For example,

```
mov     rax, qword [dquad1]
mov     rdx, qword [dquad1+8]
```

The first move to the **rax** register accesses the first 64-bits of the 128-bit variable. The second move to the **rdx** register access the next 64-bits of the 128-bit variable. This is accomplished by using the variable starting address, ***dquad1*** and adding 8 bytes, thus skipping the first 64-bits (or 8 bytes) and accessing the next 64-bits.

If the LSQ's are added and then the MSQ's are added including any carry, the 128-bit result can be correctly obtained. For example,

```
mov     rax, qword [dquad1]
mov     rdx, qword [dquad1+8]

add     rax, qword [dquad2]
adc     rdx, qword [dquad2+8]

mov     qword [dqSum], rax
mov     qword [dqSum+8], rdx
```

Initially, the LSQ of ***dquad1*** is placed in **rax** and the MSQ is placed in **rdx**. Then the **add** instruction will add the 64-bit **rax** with the LSQ of ***dquad2*** and, in this example, provide a carry of 1 with the result in **rax**. Then the **rdx** is added with the MSQ of ***dquad2*** along with the carry via the **adc** instruction and the result placed in **rdx**.

The integer add with carry instruction is summarized as follows:

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`

A more complete list of the instructions is located in Appendix B.

7.5.2 Subtraction

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

```
sub   <dest>, <src>
```

Where operation performs the following:

```
<dest> = <dest> - <src>
```

Specifically, 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. Both operands, cannot be memory. If a memory to memory subtraction operation is required, two instructions must be used.

For example, 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      73000000
qAns        dd      0
```

To perform, the basic operations of:

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

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
```

For some instructions, including those above, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) can be omitted as the other operand will clearly define the size. It is included for consistency and good programming practices.

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

```
dec   <operand>
```

Where operation performs the following:

```
<operand> = <operand> - 1
```

The result is exactly the same as using the subtract 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.

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
```

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
```

The subtraction instructions operate the same on signed and unsigned data. It is the programmer's responsibility to ensure that the data types and sizes are appropriate for the operations being performed.

The integer subtraction instructions are summarized as follows:

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.

Instruction	Explanation

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]`

A more complete list of the instructions is located in Appendix B.

7.5.3 Integer Multiplication

The multiply instruction multiplies two integer operands. Mathematically, there are special rules for handling multiplication of signed values. As such, different instructions are used for unsigned multiplication (**mul**) and signed multiplication (**imul**).

Multiplication typically produces double sized results. That is, multiplying two ***n***-bit values produces a 2***n***-bit result. Multiplying two 8-bit numbers will produce a 16-bit result. Similarly, multiplication of two 16-bit numbers will produce a 32-bit result, multiplication of two 32-bit numbers will produce a 64-bit result, and multiplication of two 64-bit numbers will produce a 128-bit result.

There are many variants for the multiply instruction. For the signed multiply, some forms will truncate the result to the size of the original operands. It is the programmer's responsibility to ensure that the values used will work for the specific instructions selected.

7.5.3.1 Unsigned Multiplication

The general form of the unsigned multiplication is as follows:

```
mul     <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 (**al** for 8-bits, **ax** for 16-bits, **eax** for 32-bits, and **rax** for 64-bit). The other operand can be a memory location or register, but not an immediate. Additionally, the result will be placed in the **A** and possibly **D** registers, based on the sizes being multiplied. The following table shows the various options for the byte, word, double-word, and quadword unsigned multiplications.

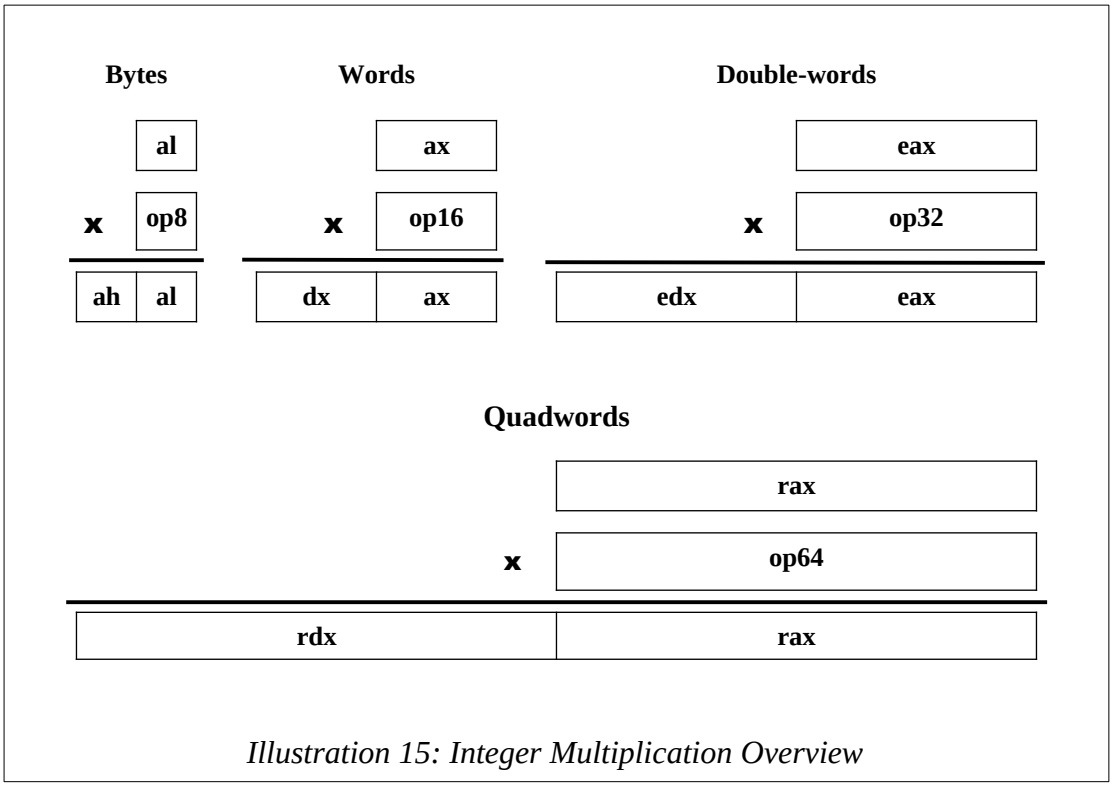


Illustration 15: Integer Multiplication Overview

As shown in the chart, for most cases the integer multiply uses a combination of the **A** and **D** registers. This can be very confusing.

For example, when multiplying a **rax** (64-bits) times a quadword operand (64-bits), the multiplication instruction provides a double quadword result (128-bit). This can be useful and important when dealing with very large numbers. Since the 64-bit architecture only has 64-bit registers, the 128-bit result is, and must be, placed in two different quadword (64-bit) registers, **rdx** for the upper-order result and **rax** for the lower-order result, which is typically written as **rdx:rax** (by convention).

However, this use of two registers is applied to smaller sizes as well. For example, the result of multiplying **ax** (16-bits) times a word operand (also 16-bits) provides a double-word (32-bit) result. However, the result is not placed in **eax** (which might be easier), it is placed in two registers, **dx** for the upper-order result (16-bits) and **ax** for the lower-order result (16-bits), typically written as **dx:ax** (by convention). Since the double-word (32-bit) result is in two different registers, two moves may be required to save the result.

This pairing of registers, even when not required, is due to legacy support for previous earlier versions of the architecture. While this helps ensure backwards compatibility, it can be quite confusing.

For example, assuming the following data declarations:

```
bNumA       db      42
bNumB       db      73
wAns        dw      0
wAns1       dw      0

wNumA       dw      4321
wNumB       dw      1234
dAns2       dd      0

dNumA       dd      42000
dNumB       dd      73000
qAns3       dq      0

qNumA       dq      420000
qNumB       dq      730000
dqAns4      ddq     0
```

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

; 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
```

For some instructions, including those above, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) is required to clearly define the size.

The integer unsigned multiplication instruction is summarized as follows:

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]**

A more complete list of the instructions is located in Appendix B.

7.5.3.2 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>
imul    <dest>, <src/imm>
imul    <dest>, <src>, <imm>
```

In all cases, the destination operand must be a register. For the multiple operand multiply instruction, byte operands are not supported.

When using a **single** operand multiply instruction, the **imul** is the same layout as the **mul** (as previously presented). However, the operands are interpreted only as signed.

When two operands are used, the destination operand and the source operand are multiplied and the result placed in the destination operand (over-writing the previous value).

Specifically, the action performed is:

```
<dest> = <dest> * <src/imm>
```

For two operands, the <**src/imm**> operand may be a register, memory location, or immediate value. 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 (64-bit) multiplications. The final result is truncated to the size of the destination operand. A byte sized destination operand is not supported.

When three operands are used, two operands are multiplied and the result placed in the destination operand. Specifically, the action performed is:

```
<dest> = <src> * <imm>
```

For three operands, the <**src**> operand must be a register or memory location, but not an immediate. The <**imm**> operand must be an immediate value. 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.

It should be noted that when the multiply instruction provides a larger type, the original type may be used. For this to work, the values multiplied must fit into the smaller size which limits the range of the data. For example, when two double-words are multiplied and a quadword result is provided, the least significant double-word (of the quadword) will contain the answer if the values are sufficiently small which is often the case. This is typically done in high-level languages when an **`int`** (32-bit integer) variable is multiplied by another **`int`** variable and assigned to an **`int`** variable.

For example, 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
```

To perform, the basic operations of:

```
wAns1 = wNumA * -13
wAns2 = wNumA * wNumB

dAns1 = dNumA * 113
dAns2 = dNumA * dNumB
```

```
qAns1 = qNumA * 7096
qAns2 = qNumA * qNumB
```

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

; 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
```

This example shows the three-operand multiply instruction using different registers.

In these examples, the multiplication result is truncated to the size of the destination operand. For a full-sized result, the single operand instruction should be used (as fully described in the section regarding unsigned multiplication).

For some instructions, including those above, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) may not be required to clearly define the size.

The integer signed multiplication instruction is summarized as follows:

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]**

A more complete list of the instructions is located in Appendix B.

7.5.4 Integer Division

The division instruction divides two integer operands. 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 $\frac{dividend}{divisor} = quotient$

Division requires that the dividend must be a larger size than the divisor. In order to divide by an 8-bit divisor, the dividend must be 16-bits (i.e., the larger size). Similarly, a 16-bit divisor requires a 32-bit dividend. And, a 32-bit divisor requires a 64-bit dividend.

Like the multiplication, for most cases the integer division uses a combination of the **A** and **D** registers. This pairing of registers is due to legacy support for previous earlier versions of the architecture. While this helps ensure backwards compatibility, it can be quite confusing.

Further, 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

Setting the dividend (top operand) correctly is a key source of problems. For the word, double-word, and quadword division operations, the dividend requires both the **D** register (for the upper-order portion) and **A** (for the lower-order portion).

Setting these correctly depends on the data type. If a previous multiplication was performed, the **D** and **A** registers may already be set correctly. Otherwise, a data item may need to be converted from its current size to a larger size with the upper-order portion being placed in the **D** register. For unsigned data, the upper portion will always be zero. For signed data, the existing data must be sign extended as noted in a previous section, *Signed Conversions*.

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. Refer to the *Integer Division Overview* table to see the layout more clearly.

The use of a larger size operand for the dividend matches the single operand multiplication. For simple divisions, an appropriate conversion may be required in order to ensure the dividend is set correctly. For unsigned divisions, the upper-order part of the dividend can set to zero. For signed divisions, the upper-order part of the dividend can be set with an applicable conversion instruction.

As always, division by zero will crash the program and damage the space-time continuum. So, try not to divide by zero.

The following tables provide an overview of the divide instruction for bytes, words, double-words, and quadwords.

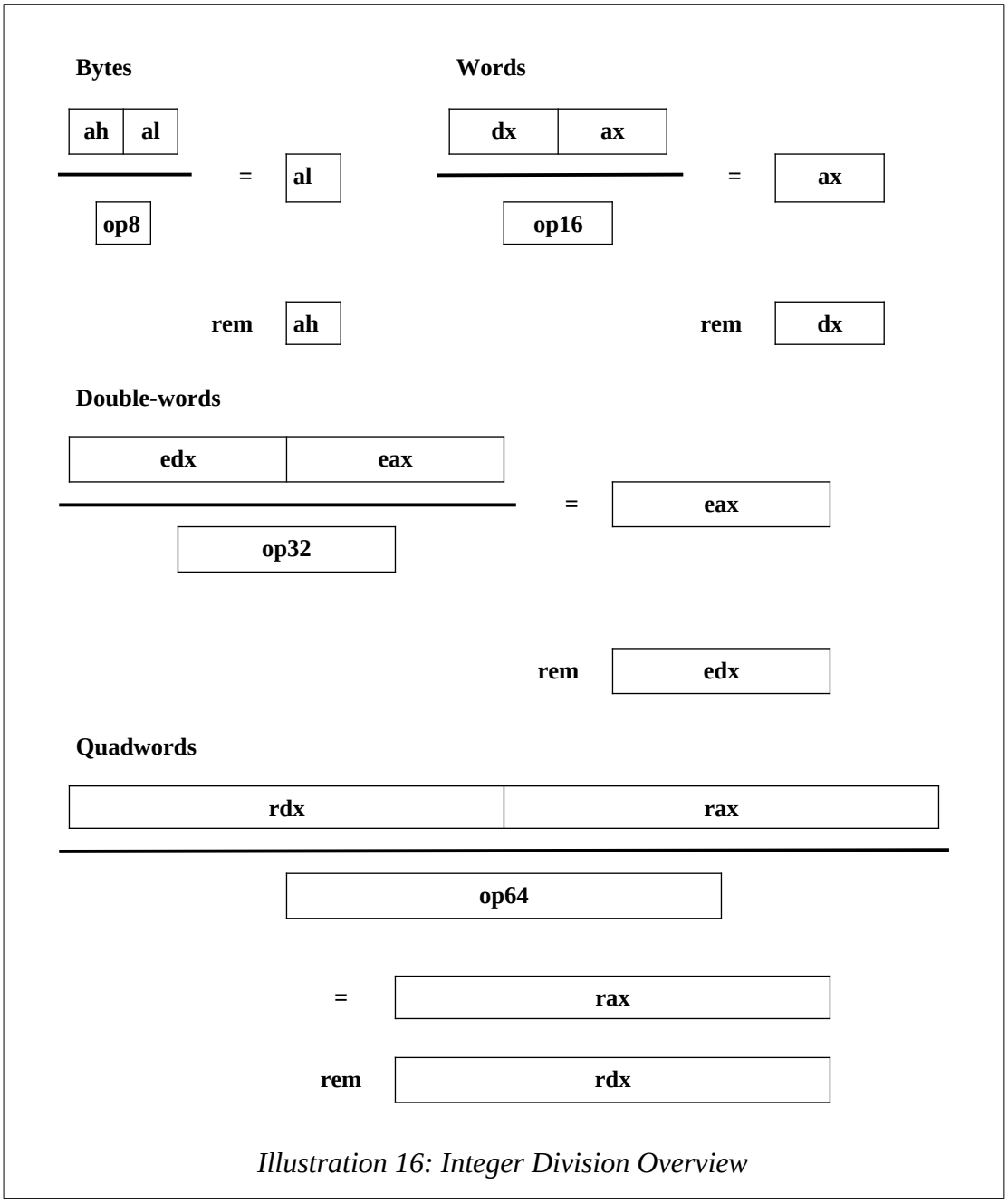


Illustration 16: Integer Division Overview

The signed and unsigned division instructions operate in the same manner. However, the range of values that can be divided is different. The programmer is responsible for ensuring that the values being divided are appropriate for the operand sizes being used.

The general forms of 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.

For example, 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
```

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
```

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 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 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 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
```

For some instructions, including those above, the explicit type specification (e.g., *byte*, *word*, *dword*, *qword*) is required to clearly define the size.

The integer division instructions are summarized as follows:

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]**
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]**

A more complete list of the instructions is located in Appendix B.

7.6 Logical Instructions

This section summarizes some of the more common logical instructions that may be useful when programming.

7.6.1 Logical Operations

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

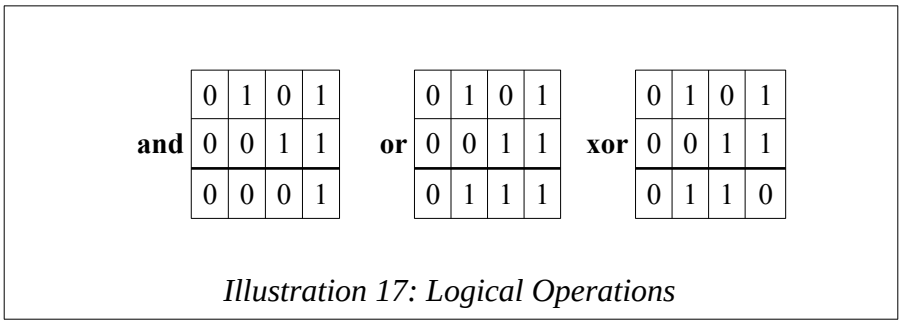


Illustration 17: Logical Operations

The logical instructions are summarized as follows:

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`

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]`

The **&** refers to the logical AND operation, the || refers to the logical OR operation, and the ^ refers to the logical XOR operation as per C/C++ conventions. The ¬ refers to the logical NOT operation.

A more complete list of the instructions is located in Appendix B.

7.6.2 Shift Operations

The shift operation shifts bits within an operand, either left or right. Two typical reasons for shifting bits include isolating a subset of the bits within an operand for some specific purpose or possibly for performing multiplication or division by powers of two. All bits are shifted one position. The bit that is shifted outside the operand is lost and a 0-bit added at the other side.

7.6.2.1 Logical Shift

The logical shift is a bitwise operation that shifts all the bits of its source register by the specified number of bits and places the result into the destination register. The bits can

be shifted left or right as needed. Every bit in the source operand is moved the specified number of bit positions and the newly vacant bit positions are filled in with zeros.

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

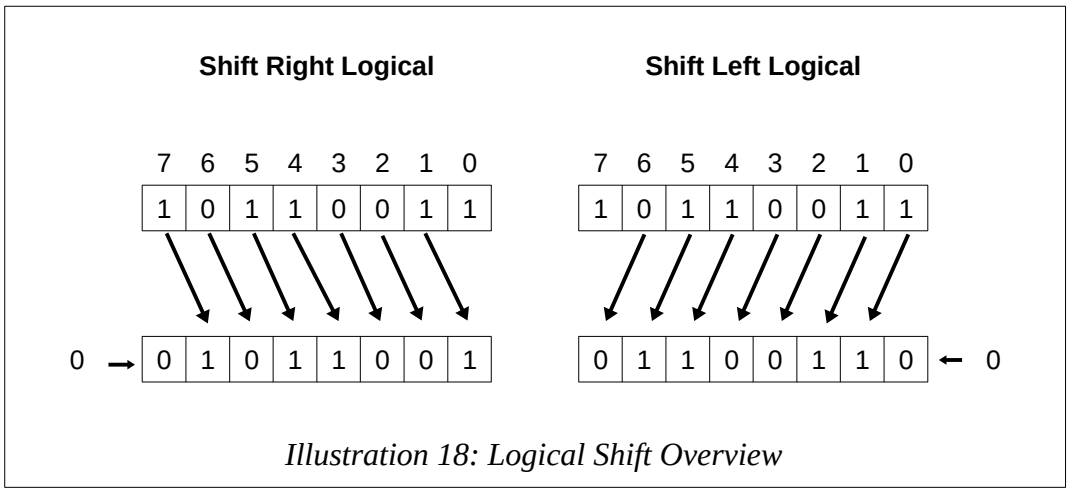


Illustration 18: Logical Shift Overview

The logical shift treats the operand as a sequence of bits rather than as a number.

The shift instructions may be used to perform unsigned integer multiplication and division operations for powers of 2. Powers of two would be 2, 4, 8, etc. up to the limit of the operand size (32-bits for register operands).

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.

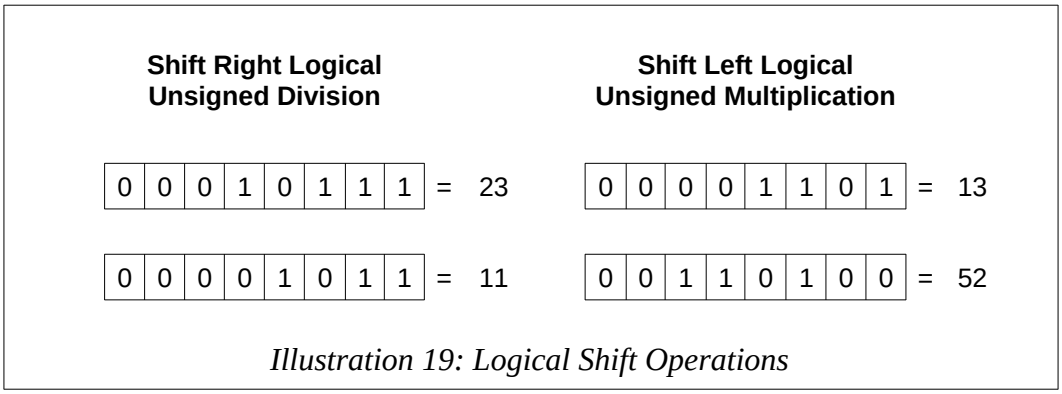


Illustration 19: Logical Shift Operations

As can be seen in the examples, a 0 was entered in the newly vacated bit locations on either the right or left (depending on the operation).

The logical shift instructions are summarized as follows:

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`

A more complete list of the instructions is located in Appendix B.

7.6.2.2 Arithmetic Shift

The arithmetic shift right is also a bitwise operation that shifts all the bits of its source register by the specified number of bits and places the result into the destination register. Every bit in the source operand is moved the specified number of bit positions, and the newly vacant bit positions are filled in. For an arithmetic left shift, the original leftmost bit (the sign bit) is replicated to fill in all the vacant positions. This is referred to as sign extension.

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

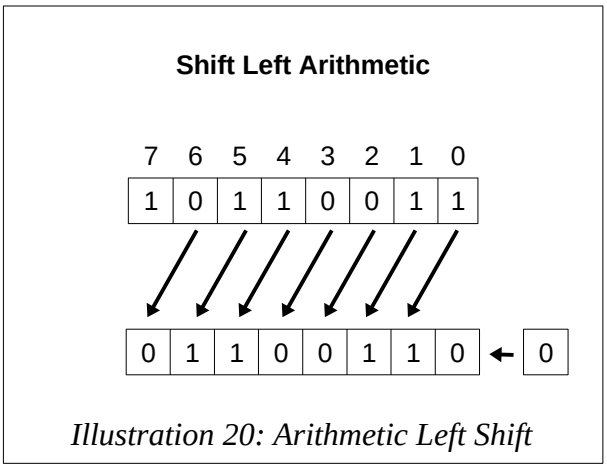


Illustration 20: Arithmetic Left Shift

The arithmetic left shift moves bits the number of specified places to the left and zero fills the from the least significant bit position (left). The leading sign bit is not preserved. The arithmetic left shift can be useful to perform an efficient multiplication by a power of two. If the resulting value does not fit an overflow is generated.

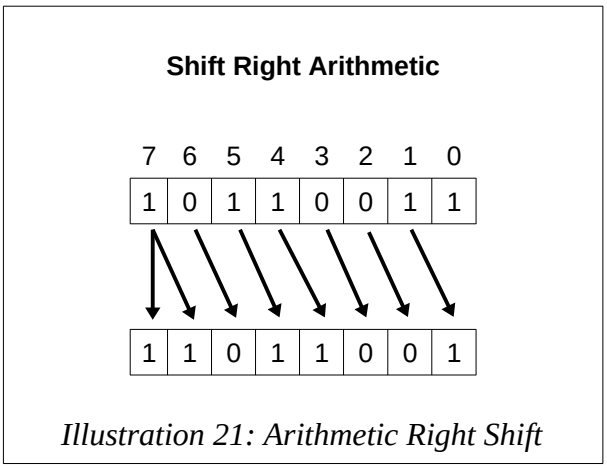


Illustration 21: Arithmetic Right Shift

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).

The arithmetic shift rounds always rounds down (towards negative infinity) and the standard divide instruction truncates (rounds toward 0). As such, the arithmetic shift is not typically used to replace the signed divide instruction.

The arithmetic shift instructions are summarized as follows:

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`

A more complete list of the instructions is located in Appendix B.

7.6.3 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.

For example, if a byte operand, 10010110_2, is rotated to the right 1 place, the result would be 01001011_2. If a byte operand, 10010110_2, is rotated to the left 1 place, the result would be 00101101_2.

The logical shift instructions are summarized as follows:

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`

A more complete list of the instructions is located in Appendix B.

7.7 Control Instructions

Program control refers to basic programming structures such as IF statements and looping.

All of the high-level language control structures must be performed with the limited assembly language control structures. For example, an IF-THEN-ELSE statement does not exist at the assembly language level. 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.

The control instructions refer to unconditional and conditional jumping. Jumping is required for basic conditional statements (i.e., IF statements) and looping.

7.7.1 Labels

A program label is the target, or a location to jump to, for control statements. For example, the start of a loop might be marked with a label such as “loopStart”. The code may be re-executed by jumping to the label.

Generally, a label starts with a letter, followed by letters, numbers, or symbols (limited to “_”), terminated with a colon (“:”). It is possible to start labels with non-letter characters (i.e., digits, “_”, “$”, “#”, “@”, “~” or “?”) . However, these typically convey special meaning and, in general, should not be used by programmers. Labels in **yasm** are case sensitive.

For example,

```
loopStart:
last:
```

are valid labels. Program labels may be defined only once.

The following sections describe how labels are used.

7.7.2 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.

The unconditional jump instruction is summarized as follows:

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

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

A more complete list of the instructions is located in Appendix B.

7.7.3 Conditional Control Instructions

The conditional control instructions provide a conditional jump based on a comparison. This provides the functionality of a basic IF statement.

Two steps are required for a comparison; the compare instruction and the conditional jump instruction. The conditional jump instruction will jump or not jump to the provided label based on the result of the previous comparison operation. The compare instruction will compare two operands and store the results of the comparison in the **rFlag** registers. The conditional jump instruction will act (jump or not jump) based on the contents of the **rFlag** register. This requires that the compare instruction is immediately followed by the conditional jump instruction. If other instructions are placed between the compare and conditional jump, the **rFlag** register will be altered and the conditional jump may not reflect the correct condition.

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.

The conditional control instructions include the jump equal (**je**) and jump not equal (**jne**) which work the same for both signed and unsigned data.

The signed conditional control instructions include the basic set of comparison operations; jump less than (**jl**), jump less than or equal (**jle**), jump greater than (**jg**), and jump greater than or equal (**jge**).

The unsigned conditional control instructions include the basic set of comparison operations; jump below than (**jb**), jump below or equal (**jbe**), jump above than (**ja**), and jump above or equal (**jae**).

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>
```

For example, given the following pseudo-code for signed data:

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

And, assuming the following data declarations:

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

Assuming that the values are updating appropriately within the program (not shown), 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:
```

Note that the logic for the IF statement has been reversed. The compare and conditional jump provide functionality for jump or not jump. As such, if the condition from the original IF statement is false, the code must not be executed. Thus, when false, in order to skip the execution, the conditional jump will jump to the target label immediately following the code to be skipped (not executed). While there is only one line in this example, there can be many lines of code.

A more complex example might be as follows:

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

This basic compare and conditional jump do not provide a typical IF-ELSE structure. It must be created. Assuming the ***x*** and ***y*** variables are signed double-words that will be set during the program execution, and the following declarations:

```
TRUE        equ     1
FALSE       equ     0
x           dd      0
y           dd      0
```

```
    ans         dd      0
    errFlg      db      FALSE
```

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
    idiv    dword [y]
    mov     dword [ans], eax
    mov     byte [errFlg], FALSE
    jmp     skpElse
doElse:
    mov     dword [ans], 0
    mov     byte [errFlg], TRUE
skpElse:
```

In this example, since the data was signed, a signed division (**idiv**) and the appropriate conversion (**cdq** in this case) were required. It should also be noted that the **edx** register was overwritten even though it did not appear explicitly. If a value was previously placed in **edx** (or **rdx**), it has been altered.

7.7.3.1 Jump Out of Range

The target label is referred to as a short-jump. Specifically, that means 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. 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:
```

Which accomplishes the same thing using an unconditional jump for the long jump and adding a conditional jump to a very close label.

The conditional jump instructions are summarized as follows:

Instruction	**Explanation**
`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.

Examples:	`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.
Examples:	`cmp  rax, 5` `je  wasEqual`
`jne  <label>`	Based on preceding comparison instruction, jump to <**label**> if <**op1**> **!=** <**op2**>. Label must be defined exactly once.
Examples:	`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.
Examples:	`cmp  rax, 5` `jl  wasLess`
`jle  <label>`	For signed data, based on preceding comparison instruction, jump to <**label**> if <**op1**> ≤ <**op2**>. Label must be defined exactly once.

Instruction	Explanation
Examples:	`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.
Examples:	`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.
Examples:	`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.
Examples:	`cmp rax, 5` `jb wasLess`
`jbe <label>`	For unsigned data, based on preceding comparison instruction, jump to <**label**> if <**op1**> ≤ <**op2**>. Label must be defined exactly once.
Examples:	`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.

Instruction	Explanation
Examples:	`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.
Examples:	`cmp rax, 5` `jae wasGreaterOrEqual`

A more complete list of the instructions is located in Appendix B.

7.7.4 Iteration

The basic control instructions outlined provide a means to iterate or loop.

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.

For example, 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
```

This is just one of many different ways to accomplish the odd integer summation task. In this example, **rcx** was used as a loop counter and **rax** was used for the current odd integer (appropriately initialized to 1 and incremented by 2).

The process shown using **rcx** as a counter is useful when looping a predetermined number of times. There is a special instruction, **loop**, provides looping support.

The general format is as follows:

```
    loop    <label>
```

Which will perform the decrement of the **rcx** register, comparison to 0, and jump to the specified label if rcx ≠ 0. The label must be defined exactly once.

As such, the loop instruction provides the same functionality as the three lines of code from the previous example program. The following sets of code are equivalent:

Code Set 1

```
loop  <label>
```

Code Set 2

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

For example, 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
```

Both code examples produce the exact same result in the same manner.

Since the **rcx** register is decremented and then checked, forgetting to set the **rcx** register could result in looping an unknown number of times. This is likely to generate an error during the loop execution, which can be very misleading when debugging.

The **loop** instruction can be useful when coding, but it is limited to the **rcx** register and to counting down. If nesting loops are required, the use of a loop instruction for both the inner and outer loop can cause a conflict unless additional actions are taken (i.e., save/restore **rcx** register as required for inner loop).

While some of the programming examples in this text will use the loop instruction, it is not required.

The loop instruction is summarized as follows:

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`

A more complete list of the instructions is located in Appendix B.

7.8 Example Program, Sum of Squares

The following is a complete example program to find the sum of squares from 1 to ***n***. For example, the sum of squares for 10 is as follows:

$$1^2 + 2^2 + \cdots + 10^2 \;=\; 385$$

This example main initializes the ***n*** value to 10 to match the above example.

```
;  Simple example program to compute the
;  sum of squares from 1 to n.
; ***********************************************
;  Data declarations

section    .data

; -----
;  Define constants

SUCCESS        equ     0          ; Successful operation
SYS_exit       equ    60          ; call code for terminate

;  Define Data.

n              dd     10
sumOfSquares   dq     0

; ****************************************************

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

; -----
;  Done, terminate program.

last:
    mov     rax, SYS_exit                   ; call code for exit
    mov     rdi, SUCCESS                    ; exit with success
    syscall
```

The debugger can be used to examine the results and verify correct execution of the program.

7.9 Exercises

Below are some quiz questions and suggested projects based on this chapter.

7.9.1 Quiz Questions

Below are some quiz questions based on this chapter.

1) Which of the following instructions is legal / illegal? As appropriate, provide an explanation.
    1. `mov   rax, 54`
    2. `mov   ax, 54`

3. **mov al, 354**
4. **mov rax, r11**
5. **mov rax, r11d**
6. **mov 54, ecx**
7. **mov rax, qword [qVar]**
8. **mov rax, qword [bVar]**
9. **mov rax, [qVar]**
10. mov rax, qVar
11. **mov eax, dword [bVar]**
12. **mov qword [qVar2], qword [qVar1]**
13. **mov qword [bVar2], qword [qVar1]**
14. **mov r15, 54**
15. **mov r16, 54**
16. **mov r11b, 54**

2) Explain what each of the following instructions does.
   1. **movzx rsi, byte [bVar1]**
   2. **movsx rsi, byte [bVar1]**

3) What instruction is used to:
   1. convert an *unsigned* byte in **al** into a word in **ax**.
   2. convert a *signed* byte in **al** into a word in **ax**.

4) What instruction is used to:
   1. convert an *unsigned* word in **ax** into a double-word in **eax**.
   2. convert a *signed* word in **ax** into a double-word in **eax**.

5) What instruction is used to:
   1. convert an *unsigned* word in **ax** into a double-word in **dx:ax**.
   2. convert a *signed* word in **ax** into a double-word in **dx:ax**.

6) Explain the difference between the **cwd** instruction and the **movsx** instructions.

7) Explain why the explicit specification (*dword* in this example) is required on the first instruction and is not required on the second instruction.

    1. `add   dword [dVar], 1`

    2. `add   [dVar], eax`

8) Given the following code fragment:

```
mov   rax, 9
mov   rbx, 2
add   rbx, rax
```

What would be in the **rax** and **rbx** registers after execution? Show answer in hex, full register size.

9) Given the following code fragment:

```
mov   rax, 9
mov   rbx, 2
sub   rax, rbx
```

What would be in the **rax** and **rbx** registers after execution? Show answer in hex, full register size.

10) Given the following code fragment:

```
mov   rax, 9
mov   rbx, 2
sub   rbx, rax
```

What would be in the **rax** and **rbx** registers after execution? Show answer in hex, full register size.

11) Given the following code fragment:

```
mov   rax, 4
mov   rbx, 3
imul  rbx
```

What would be in the **rax** and **rdx** registers after execution? Show answer in hex, full register size.

12) Given the following code fragment:

```
mov   rax, 5
cqo
```

```
mov   rbx, 3
idiv  rbx
```

What would be in the **rax** and **rdx** registers after execution? Show answer in hex, full register size.

13) Given the following code fragment:

```
mov   rax, 11
cqo
mov   rbx, 4
idiv  rbx
```

What would be in the **rax** and **rdx** registers after execution? Show answer in hex, full register size.

14) Explain why each of the following statements will not work.

1. `mov   42, eax`
2. `div   3`
3. `mov dword [num1], dword [num2]`
4. `mov   dword [ax], 800`

15) Explain why the following code fragment will not work correctly.

```
mov   eax, 500
mov   ebx, 10
idiv  ebx
```

16) Explain why the following code fragment will not work correctly.

```
mov   eax, -500
cdq
mov   ebx, 10
div   ebx
```

17) Explain why the following code fragment will not work correctly.

```
mov   ax, -500
cwd
mov   bx, 10
idiv  bx
mov   dword [ans], eax
```

18) Under what circumstances can the three-operand multiple be used?

7.9.2 Suggested Projects

Below are some suggested projects based on this chapter.

1) Create a program to compute the following expressions using unsigned byte variables and unsigned operations. *Note*, the first letter of the variable name denotes the size (**b** → byte and **w** → word).

    1. `bAns1 = bNum1 + bNum2`
    2. `bAns2 = bNum1 + bNum3`
    3. `bAns3 = bNum3 + bNum4`
    4. `bAns6 = bNum1 – bNum2`
    5. `bAns7 = bNum1 – bNum3`
    6. `bAns8 = bNum2 – bNum4`
    7. `wAns11 = bNum1 * bNum3`
    8. `wAns12 = bNum2 * bNum2`
    9. `wAns13 = bNum2 * bNum4`
    10. `bAns16 = bNum1 / bNum2`
    11. `bAns17 = bNum3 / bNum4`
    12. `bAns18 = wNum1 / bNum4`
    13. `bRem18 = wNum1 % bNum4`

    Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

2) Repeat the previous program using signed values and signed operations. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

3) Create a program to complete the following expressions using unsigned word sized variables. *Note*, the first letter of the variable name denotes the size (**w** → word and **d** → double-word).

    1. `wAns1 = wNum1 + wNum2`
    2. `wAns2 = wNum1 + wNum3`

3. `wAns3 = wNum3 + wNum4`
4. `wAns6 = wNum1 – wNum2`
5. `wAns7 = wNum1 – wNum3`
6. `wAns8 = wNum2 – wNum4`
7. `dAns11 = wNum1 * wNum3`
8. `dAns12 = wNum2 * wNum2`
9. `dAns13 = wNum2 * wNum4`
10. `wAns16 = wNum1 / wNum2`
11. `wAns17 = wNum3 / wNum4`
12. `wAns18 = dNum1 / wNum4`
13. `wRem18 = dNum1 % wNum4`

Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

4) Repeat the previous program using signed values and signed operations. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

5) Create a program to complete the following expressions using unsigned double-word sized variables. *Note*, the first letter of the variable name denotes the size (**d** → double-word and **q** → quadword).

1. `dAns1 = dNum1 + dNum2`
2. `dAns2 = dNum1 + dNum3`
3. `dAns3 = dNum3 + dNum4`
4. `dAns6 = dNum1 – dNum2`
5. `dAns7 = dNum1 – dNum3`
6. `dAns8 = dNum2 – dNum4`
7. `qAns11 = dNum1 * dNum3`
8. `qAns12 = dNum2 * dNum2`
9. `qAns13 = dNum2 * dNum4`

10. **dAns16 = dNum1 / dNum2**
11. **dAns17 = dNum3 / dNum4**
12. **dAns18 = qNum1 / dNum4**
13. **dRem18 = qNum1 % dNum4**

Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

6) Repeat the previous program using signed values and signed operations. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

7) Implement the example program to compute the sum of squares from 1 to ***n***. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

8) Create a program to compute the square of the sum from 1 to ***n***. Specifically, compute the sum of integers from 1 to ***n*** and then square the value. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results in both decimal and hexadecimal.

9) Create a program to iteratively find the ***n***th Fibonacci number[37]. The value for ***n*** should be set as a parameter (e.g., a programmer defined constant). The formula for computing Fibonacci is as follows:

$$fibonacci(n) = \begin{cases} n & \text{if } n=0 \text{ or } n=1 \\ fibonacci(n-2) + fibonacci(n-1) & \text{if } n \geq 2 \end{cases}$$

Use the debugger to execute the program and display the final results. Test the program for various values of ***n***. Create a debugger input file to show the results in both decimal and hexadecimal.

37 For more information, refer to: http://en.wikipedia.org/wiki/Fibonacci_number