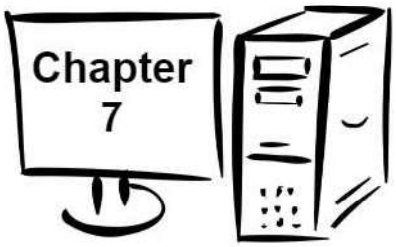
Why are math books sad?
Because they have so many problems.



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.

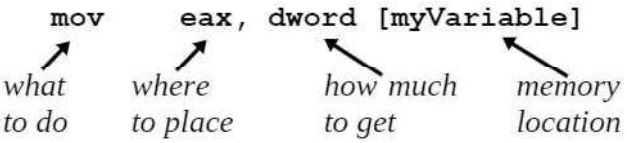


Illustration 14: MOV Instruction Overview

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`

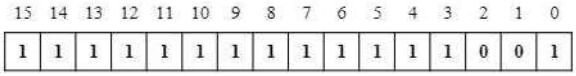
A more complete list of the instructions is located in Appendix B.

7.4.2.2 Signed Conversions

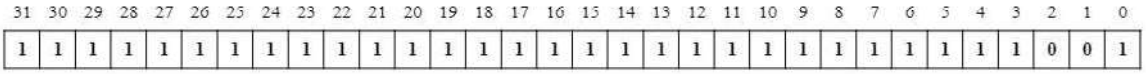
For signed widening conversions, the upper-order bits must be set to either 0's or 1's depending on if the original value was positive or negative.

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:



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:



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

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]`

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.

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.

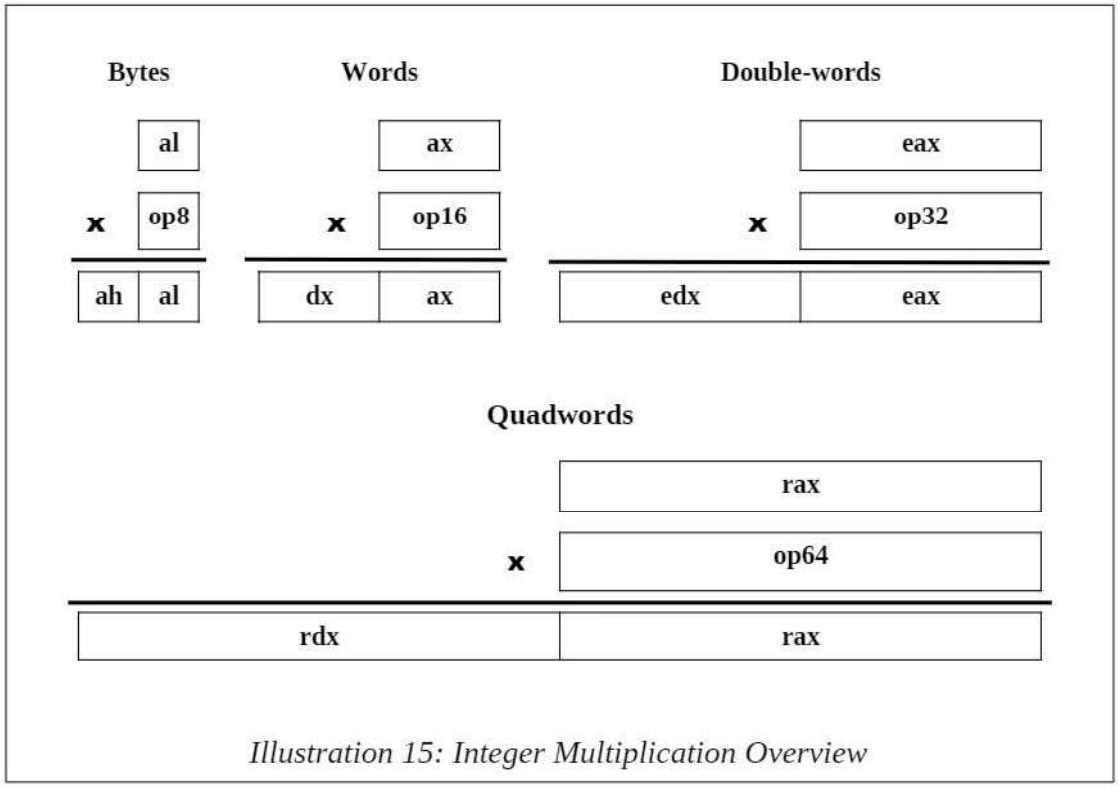


Illustration 15: Integer Multiplication Overview

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).

IIowever, 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.

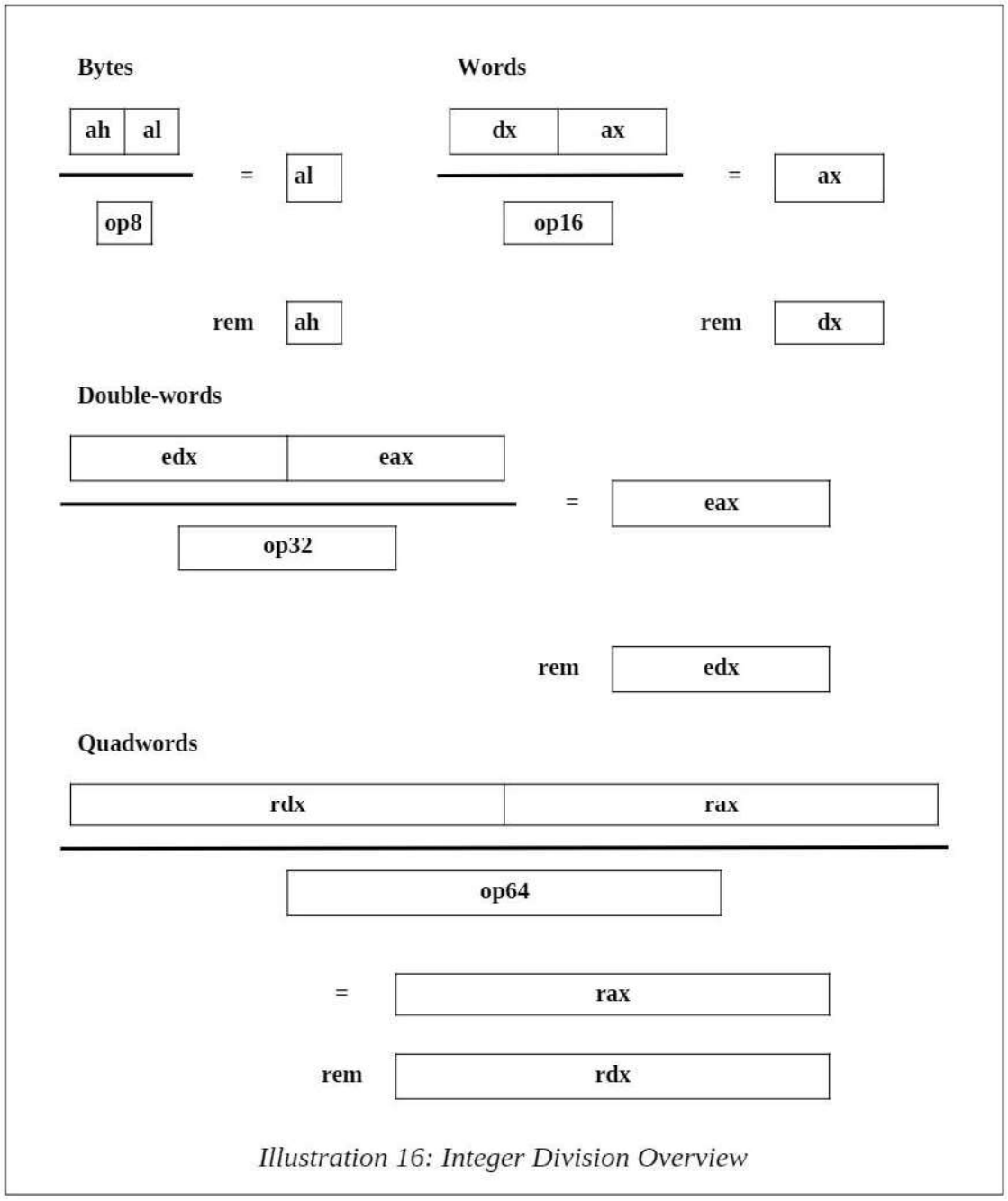


Illustration 16: Integer Division Overview

The signed and unsigned division instructions operate in the same manner. However, the range 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 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.

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;

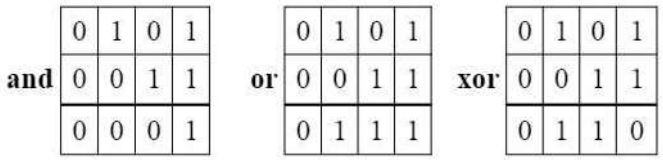


Illustration 17: Logical Operations

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.

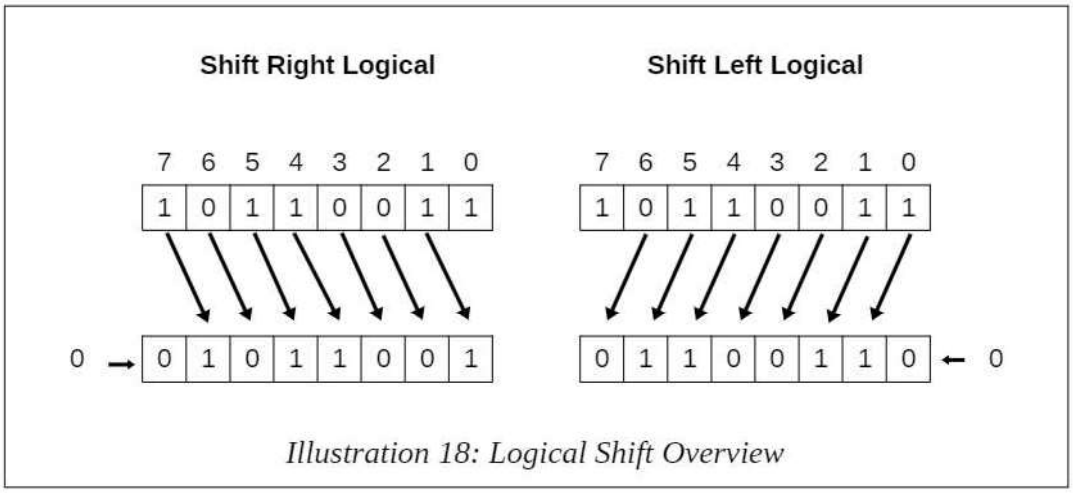


Illustration 18: Logical Shift Overview

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.

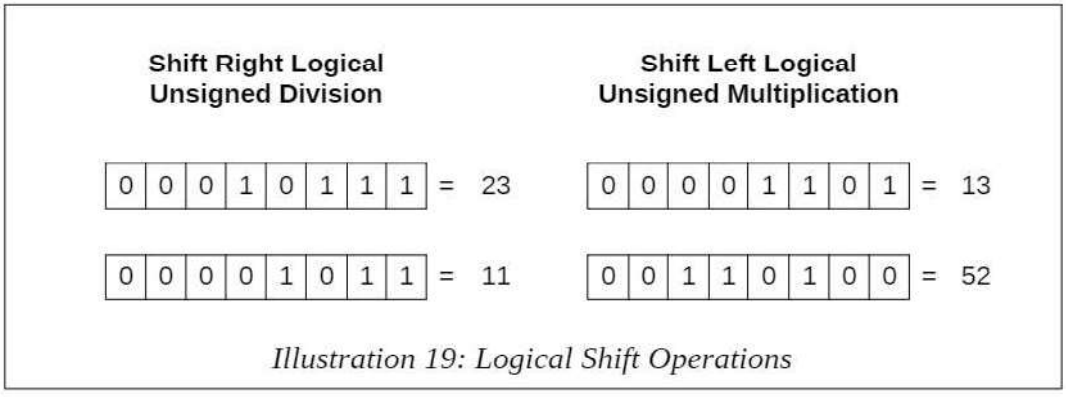


Illustration 19: Logical Shift Operations

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.

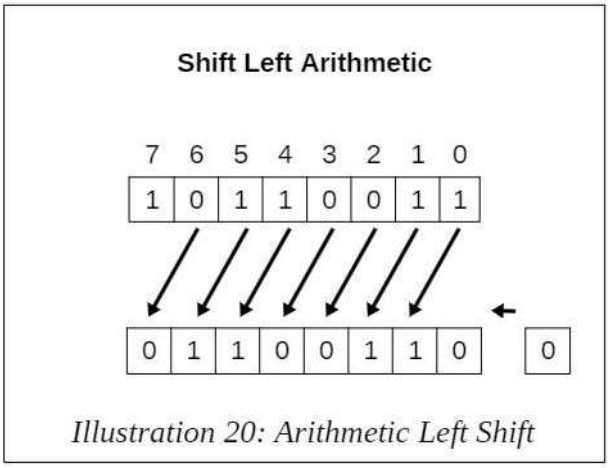


Illustration 20: Arithmetic Left Shift

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.

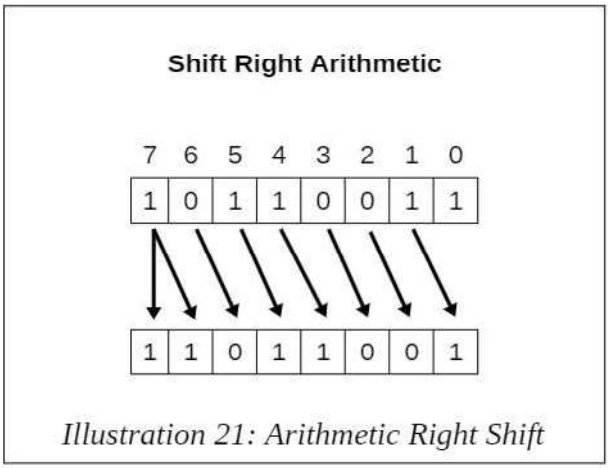


Illustration 21: Arithmetic Right Shift

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.
Examples:	`cmp rax, 5` `jle wasLessOrEqual`

Instruction	Explanation
`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.
Examples:	`cmp rax, 5` `ja wasGreater`

Instruction	Explanation
`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.

Instruction	Explanation

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 type 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 [num1]`
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.

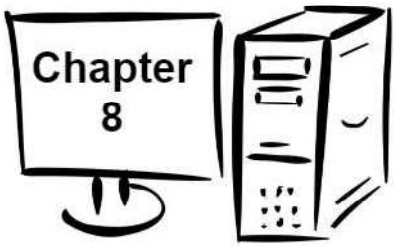
9) Create a program to iteratively find the ***n***th Fibonacci number[37]. The value for ***n*** should be set as a parameter (e.g., a programmer defined constant). The formula for computing Fibonacci is as follows:

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

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

Why did the programmer quit his job?
Because he didn't get arrays.



8.0 Addressing Modes

This chapter provides some basic information regarding addressing modes and the associated address manipulations on the x86-64 architecture.

The addressing modes are the supported methods for accessing a value in memory using the address of a data item being accessed (read or written). This might include the name of a variable or the location in an array.

The basic addressing modes are:

- Register
- Immediate
- Memory

Each of these modes is described with examples in the following sections. Additionally, a simple example for accessing an array is presented.

8.1 Addresses and Values

On a 64-bit architecture, addresses require 64-bits.

As noted in the previous chapter, 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.

When accessing memory, in many cases the operand size is clear. For example, the instruction

```
mov eax, [rbx]
```

moves a double-word from memory. However, for some instructions the size can be ambiguous. For example,

```
inc [rbx]                          ; error
```

is ambiguous since it is not clear if the memory being accessed is a byte, word, or double-word. In such a case, operand size must be specified with either the *byte*, *word*, or *dword*, *qword* size qualifier. For example,

```
inc byte [rbx]
inc word [rbx]
inc dword [rbx]
```

each instruction requires the size specification in order to be clear and legal.

8.1.1 Register Mode Addressing

Register mode addressing means that the operand is a CPU register (**eax**, **ebx**, etc.). For example:

```
mov     eax, ebx
```

Both **eax** and **ebx** are in register mode addressing.

8.1.2 Immediate Mode Addressing

Immediate mode addressing means that the operand is an immediate value. For example:

```
mov eax, 123
```

The destination operand, **eax** , is register mode addressing. The **123** is immediate mode addressing. It should be clear that the destination operand in this example cannot be immediate mode.

8.1.3 Memory Mode Addressing

Memory mode addressing means that the operand is a location in memory (accessed via

an address). This is referred to as *indirection* or *dereferencing*.

The most basic form of memory mode addressing has been used extensively in the previous chapter. Specifically, the instruction:

```
mov   rax, qword [qNum]
```

Will access the memory location of the variable ***qNum*** and retrieve the value stored there. This requires that the CPU wait until the value is retrieved before completing the operation and thus might take slightly longer to complete than a similar operation using an immediate value.

When accessing arrays, a more generalized method is required. Specifically, an address can be placed in a register and indirection performed using the register (instead of the variable name).

For example, assuming the following declaration:

```
lst    dd     101, 103, 105, 107
```

The decimal value of 101 is 0x00000065 in hex. The memory picture would be as follows:

	Value	Address	Offset	Index
	00	0x6000ef	lst + 15	
	00	0x6000ee	lst + 14	
	00	0x6000ed	lst + 13	
	6b	0x6000ec	lst + 12	**lst[3]**
	00	0x6000eb	lst + 11	
	00	0x6000ea	lst + 10	
	00	0x6000e9	lst + 9	
	69	0x6000e8	lst + 8	**lst[2]**
	00	0x6000e7	lst + 7	
	00	0x6000e6	lst + 6	
	00	0x6000e5	lst + 5	
	67	0x6000e4	lst + 4	**lst[1]**
	00	0x6000e3	lst + 3	
	00	0x6000e2	lst + 2	
	00	0x6000e1	lst + 1	
lst →	65	0x6000e0	lst + 0	**lst[0]**

The first element of the array could be accessed as follows:

```
mov     eax, dword [lst]
```

Another way to access the first element is as follows:

```
mov     rbx, list
mov     eax, dword [rbx]
```

In this example, the starting address, or base address, of the list is placed in **rbx** (first line) and then the value at that address is accessed and placed in the **rax** register (second line). This allows us to easily access other elements in the array.

Recall that memory is "byte addressable", which means that each address is one byte of information. A double-word variable is 32-bits or 4 bytes so each array element uses 4 bytes of memory. As such, the next element (103) is the starting address (lst) plus 4, and the next element (105) is the starting address (**lst**) 8.

Increasing the offset by 4 for each successive element. A list of bytes would increase by 1, a list of words would increase by 2, a list of double-words would increase by 4, and a list of quadwords would increase by 8.

The offset is the amount added to the base address. The index is the array element number as used in a high-level language.

There are several ways to access the array elements. One is to use a base address and add a displacement. For example, given the initializations:

```
mov     rbx, lst
mov     rsi, 8
```

Each of the following instructions access the third element (105 in the above list).

```
mov     eax, dword [lst+8]
mov     eax, dword [rbx+8]
mov     eax, dword [rbx+rsi]
```

In each case, the starting address plus 8 was accessed and the value of 105 placed in the **eax** register. The displacement is added and the memory location accessed while none of the source operand registers (**rbx**, **rsi**) are altered. The specific method used is up to the programmer.

In addition, the displacement may be computed in more complex ways.

The general format of memory addressing is as follows:

```
[ baseAddr + (indexReg * scaleValue ) + displacement ]
```

Where ***baseAddr*** is a register or a variable name. The ***indexReg*** must be a register. The ***scaleValue*** is an immediate value of 1, 2, 4, 8 (1 is legal, but not useful). The ***displacement*** must be an immediate value. The total represents a 64-bit address.

Elements may be used in any combination, but must be legal and result in a valid address.

Some example of memory addressing for the source operand are as follows:

```
mov     eax, dword [var1]
mov     rax, qword [rbx+rsi]
mov     ax, word [lst+4]
mov     bx, word [lst+rdx+2]
mov     rcx, qword [lst+(rsi*8)]
mov     al, byte [buff-1+rcx]
mov     eax, dword [rbx+(rsi*4)+16]
```

For example, to access the 3rd element of the previously defined double-word array (which is index 2 since index's start at 0):

```
mov     rsi, 2                          ; index=2
mov     eax, dword [lst+rsi*4]          ; get lst[2]
```

Since addresses are always *qword* (on a 64-bit architecture), a 64-bit register is used for the memory mode addressing (even when accessing double-word values). This allows a register to be used more like an array index (from a high-level language).

For example, the memory operand, **[lst+rsi*4]**, is analogous to **lst[rsi]** from a high-level language. The **rsi** register is multiplied by the data size (4 in this example since each element is 4 bytes).

8.2 Example Program, List Summation

The following example program will sum the numbers in a list.

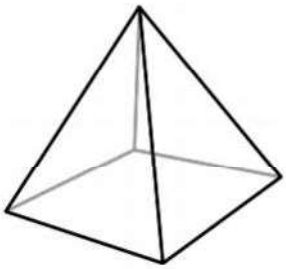
```
; Simple example to the sum and average for
; a list of numbers.

; ****************************************************
;  Data declarations

section    .data
```

```
; -----
;  Define constants

EXIT_SUCCESS   equ 0          ; successful operation
SYS_exit       equ 60         ; call code for terminate

; -----
;   Define Data.

section    .data
    lst    dd     1002, 1004, 1006, 1008, 10010
    len    dd     5
    sum    dd     0

; ************************************************************
section    .text
global _start
_start:

; -----
;  Summation loop.

    mov    ecx, dword [len]             ; get length value
    mov    rsi, 0                       ; index=0

sumLoop:
    mov    eax, dword [lst+(rsi*4)]     ; get lst[rsi]
    add    dword [sum], eax             ; update sum
    inc    rsi                          ; next item
    loop   sumLoop

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

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

The **()**'s within the **[ ]**'s are not required and added only for clarity. As such, the **[lst+(rsi*4)]**, is exactly the same as **[lst+rsi*4]**.

8.3 Example Program, Pyramid Areas and Volumes

This example is a simple assembly language program to calculate some geometric information for each square pyramid in a series of square pyramids. Specifically, the program will find the lateral total surface area (including the base) and volume of each square pyramid in a set of square pyramids.



Once the values are computed, the program finds the minimum, maximum, sum, and average for the total surface areas and volumes.

All data are unsigned values (i.e., uses **mul** and **div**, not **imul** or **idiv**).

This basic approach used in this example is the loop to calculate the surface areas and volumes arrays. A second loop is used to find the sum, minimum, and maximum for each array. To find the minimum and maximum values, the minimum and maximum variables are each initialized to the first value in the list. Then, every element in the list is compared to the current minimum and maximum. If the current value from the list is less than the current minimum, the minimum is set to the current value (over-writing the previous value). When all values have been checked, the minimum will represent the true minimum from the list. If the current value from the list is more than the current maximum, the maximum is set to the current value (over-writing the previous value). When all values have been checked, the maximum will represent the true maximum from the list.

```
;  Example assembly language program to calculate the
;  geometric information for each square pyramid in
;  a series of square pyramids.

;  The program calculates the total surface area
;  and volume of each square pyramid.

;  Once the values are computed, the program finds
;  the minimum, maximum, sum, and average for the
;  total surface areas and volumes.

; -----
;  Formulas:
;   totalSurfaceAreas(n) = aSides(n) *
;                  (2*aSides(n)*sSides(n))
;   volumes(n) = (aSides(n)^2 * heights(n)) / 3
```

```
; ************************************************

section    .data

; -----
;  Define constants

EXIT_SUCCESS   equ    0              ; successful operation
SYS_exit       equ    60             ; call code for terminate

; -----
;  Provided Data

aSides     db     10,    14,    13,    37,    54
           db     31,    13,    20,    61,    36
           db     14,    53,    44,    19,    42
           db     27,    41,    53,    62,    10
           db     19,    18,    14,    10,    15
           db     15,    11,    22,    33,    70
           db     15,    23,    15,    63,    26
           db     24,    33,    10,    61,    15
           db     14,    34,    13,    71,    81
           db     38,    13,    29,    17,    93

sSides     dw   1233,  1114,  1773,  1131,  1675
           dw   1164,  1973,  1974,  1123,  1156
           dw   1344,  1752,  1973,  1142,  1456
           dw   1165,  1754,  1273,  1175,  1546
           dw   1153,  1673,  1453,  1567,  1535
           dw   1144,  1579,  1764,  1567,  1334
           dw   1456,  1563,  1564,  1753,  1165
           dw   1646,  1862,  1457,  1167,  1534
           dw   1867,  1864,  1757,  1755,  1453
           dw   1863,  1673,  1275,  1756,  1353


heights    dd  14145, 11134, 15123, 15123, 14123
           dd  18454, 15454, 12156, 12164, 12542
           dd  18453, 18453, 11184, 15142, 12354
           dd  14564, 14134, 12156, 12344, 13142
           dd  11153, 18543, 17156, 12352, 15434
           dd  18455, 14134, 12123, 15324, 13453
           dd  11134, 14134, 15156, 15234, 17142
```

```
            dd  19567, 14134, 12134, 17546, 16123
            dd  11134, 14134, 14576, 15457, 17142
            dd  13153, 11153, 12184, 14142, 17134

length      dd  50

taMin       dd  0
taMax       dd  0
taSum       dd  0
taAve       dd  0

volMin      dd  0
volMax      dd  0
volSum      dd  0
volAve      dd  0

; -----
; Additional variables

ddTwo       dd  2
ddThree     dd  3

; ------------------------------------------------
; Uninitialized data

section     .bss
totalAreas      resd    50
volumes         resd    50

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

section     .text
global _start
_start:

;  Calculate volume, lateral and total surface areas

    mov     ecx, dword [length]             ; length counter
    mov     rsi, 0                          ; index

calculationLoop:

;   totalAreas(n) = aSides(n) * (2*aSides(n)*sSides(n))

    movzx   r8d, byte [aSides+rsi]          ; aSides[i]
```

```
    movzx   r9d, word [sSides+rsi*2]         ; sSides[i]
    mov     eax, r8d
    mul     dword [ddTwo]
    mul     r9d
    mul     r8d
    mov     dword [totalAreas+rsi*4], eax

;   volumes(n) = (aSides(n)^2 * heights(n)) / 3

    movzx   eax, byte [aSides+rsi]
    mul     eax
    mul     dword [heights+rsi*4]
    div     dword [ddThree]
    mov     dword [volumes+rsi*4], eax

    inc     rsi
    loop    calculationLoop

; -----
;  Find min, max, sum, and average for the total
;  areas and volumes.

    mov     eax, dword [totalAreas]
    mov     dword [taMin], eax
    mov     dword [taMax], eax

    mov     eax, dword [volumes]
    mov     dword [volMin], eax
    mov     dword [volMax], eax

    mov     dword [taSum], 0
    mov     dword [volSum], 0

    mov     ecx, dword [length]
    mov     rsi, 0

statsLoop:
    mov     eax, dword [totalAreas+rsi*4]
    add     dword [taSum], eax

    cmp     eax, dword [taMin]
    jae     notNewTaMin
    mov     dword [taMin], eax
```

```
notNewTaMin:
    cmp     eax, dword [taMax]
    jbe     notNewTaMax
    mov     dword [taMax], eax

notNewTaMax:
    mov     eax, dword [volumes+rsi*4]
    add     dword [volSum], eax
    cmp     eax, dword [volMin]
    jae     notNewVolMin
    mov     dword [volMin], eax

notNewVolMin:
    cmp     eax, dword [volMax]
    jbe     notNewVolMax
    mov     dword [volMax], eax
notNewVolMax:

    inc     rsi
    loop    statsLoop

; -----
;  Calculate averages.

    mov     eax, dword [taSum]
    mov     edx, 0
    div     dword [length]
    mov     dword [taAve], eax

    mov     eax, dword [volSum]
    mov     edx, 0
    div     dword [length]
    mov     dword [volAve], eax

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

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

This is one example. There are multiple other valid approaches to solving this problem.

8.4 Exercises

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

8.4.1 Quiz Questions

Below are some quiz questions based on this chapter.

1) Explain the difference between the following two instructions:

    1. `mov  rdx, qword [qVar1]`
    2. `mov  rdx, qVar1`

2) What is the address mode of the source operand for each of the instructions listed below. Respond with *Register*, *Immediate*, *Memory*, or *Illegal Instruction*.

    Note, `mov  <dest>, <source>`

```
mov    ebx, 14
mov    ecx, dword [rbx]
mov    byte [rbx+4], 10
mov    10, rcx
mov    dl, ah
mov    ax, word [rsi+4]
mov    cx, word [rbx+rsi]
mov    ax, byte [rbx]
```

3) Given the following variable declarations and code fragment:

```
ans1   dd   7

mov    rax, 3
mov    rbx, ans1
add    eax, dword [rbx]
```

    What would be in the **eax** register after execution? Show answer in hex, full register size.

4) Given the following variable declarations and code fragment:

```
list1     dd   2, 3, 4, 5, 6, 7

mov rbx, list1
add rbx, 4
mov eax, dword [rbx]
mov edx, dword [list1]
```

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

5) Given the following variable declarations and code fragment:

```
lst    dd   2, 3, 5, 7, 9

       mov      rsi, 4
       mov      eax, 1
       mov      rcx, 2
lp:    add      eax, dword [lst+rsi]
       add      rsi, 4
       loop     lp
       mov      ebx, dword [lst]
```

What would be in the **eax**, **ebx**, **rcx**, and **rsi** registers after execution? Show answer in hex, full register size. *Note*, pay close attention to the register sizes (32-bit vs. 64-bit).

6) Given the following variable declarations and code fragment:

```
list      dd    8, 6, 4, 2, 1, 0

      mov      rbx, list
      mov      rsi, 1
      mov      rcx, 3
      mov      edx, dword [rbx]
lp:   mov      eax, dword [list+rsi*4]
      inc      rsi
      loop     lp
      imul     dword [list]
```

What would be in the **eax**, **edx**, **rcx**, and **rsi** registers after execution? Show answer in hex, full register size. *Note*, pay close attention to the register sizes (32-bit vs. 64-bit).

7) Given the following variable declarations and code fragment:

```
list      dd    8, 7, 6, 5, 4, 3, 2, 1, 0

          mov   rbx, list
          mov   rsi, 0
          mov   rcx, 3
          mov   edx, dword [rbx]
lp:       add   eax, dword [list+rsi*4]
          inc   rsi
          loop  lp
          cdq
          idiv  dword [list]
```

What would be in the **eax**, **edx**, **rcx**, and **rsi** registers after execution? Show answer in hex, full register size. *Note*, pay close attention to the register sizes (32-bit vs. 64-bit).

8) Given the following variable declarations and code fragment:

```
list      dd   2, 7, 4, 5, 6, 3

          mov   rbx, list
          mov   rsi, 1
          mov   rcx, 2
          mov   eax, 0
          mov   edx, dword [rbx+4]
lp:       add   eax, dword [rbx+rsi*4]
          add   rsi, 2
          loop  lp
          imul  dword [rbx]
```

What would be in the **eax**, **edx**, **rcx**, and **rsi** registers after execution? Show answer in hex, full register size. *Note*, pay close attention to the register sizes (32-bit vs. 64-bit).

8.4.2 Suggested Projects

Below are some suggested projects based on this chapter.

1) Implement the example program to sum a list of numbers. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

2) Update the example program from the previous question to find the maximum, minimum, and average for the list of numbers. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

3) Implement the example program to compute the lateral total surface area (including the base) and volume of each square pyramid in a set of square pyramids. Once the values are computed, the program finds the minimum, maximum, sum, and average for the total surface areas and volumes. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

4) Write an assembly language program to find the minimum, middle value, maximum, sum, and integer average of a list of numbers. Additionally, the program should also find the sum, count, and integer average for the negative numbers. The program should also find the sum, count, and integer average for the numbers that are evenly divisible by 3. Unlike the median, the 'middle value' does not require the numbers to be sorted. *Note*, for an odd number of items, the middle value is defined as the middle value. For an even number of values, it is the integer average of the two middle values. Assume all data is unsigned. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

5) 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.

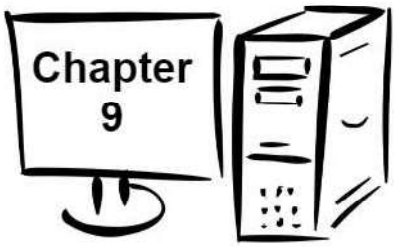
6) Create a program to sort a list of numbers. Use the following bubble sort[38] algorithm:

```
for ( i = (len-1) to 0 ) {
     swapped = false
     for ( j = 0 to i-1 )
         if ( lst(j) > lst(j+1) ) {
             tmp = lst(j)
             lst(j) = lst(j+1)
             lst(j+1) = tmp
             swapped = true
         }
     if ( swapped = false ) exit
}
```

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

38 For more information, refer to: http://en.wikipedia.org/wiki/Bubble_sort

A programmer is heading out to the grocery store, and is asked to "get a gallon of milk, and if they have eggs, get a dozen." He returns with 12 gallons of milk.



9.0 Process Stack

In a computer, a stack is a type of data structure where items are added and then removed from the stack in reverse order. That is, the most recently added item is the very first one that is removed. This is often referred to as Last-In, First-Out (LIFO).

A stack is heavily used in programming for the storage of information during procedure or function calls. The following chapter provides information and examples regarding the stack.

Adding an item to a stack is referred to as a ***push*** or push operation. Removing an item from a stack is referred to as a ***pop*** or pop operation.

It is expected that the reader will be familiar with the general concept of a stack.

9.1 Stack Example

To demonstrate the general usage of the stack, given an array, `a = {7, 19, 37}`, consider the operations:

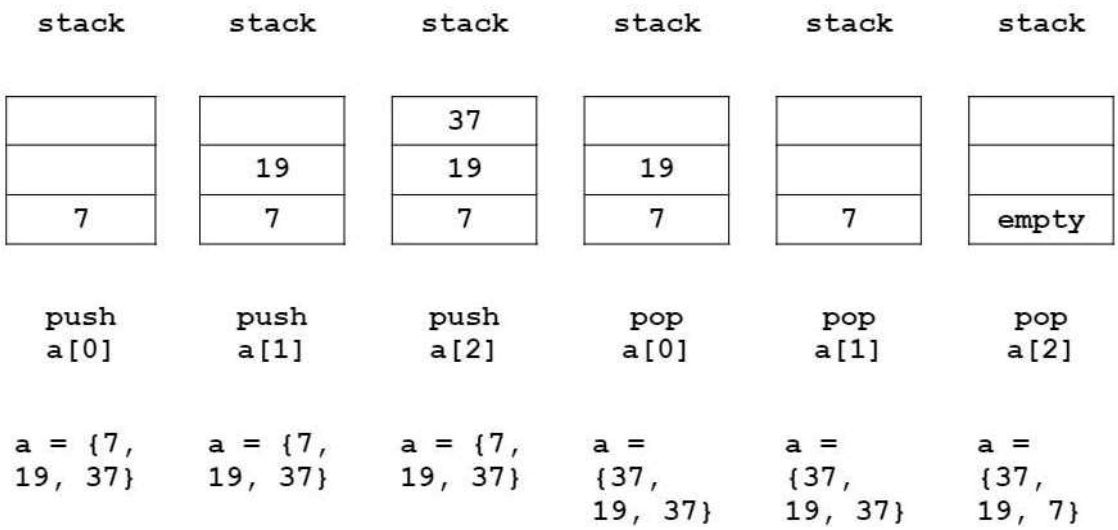
```
push    a[0]
push    a[1]
push    a[2]
```

Followed by the operations:

```
pop     a[0]
pop     a[1]
pop     a[2]
```

The initial push will push the 7, followed by the 19, and finally the 37. Since the stack is last-in, first-out, the first item popped off the stack will be the last item pushed, or 37 in this example. The 37 is placed in the first element of the array (over-writing the 7). As this continues, the order of the array elements is reversed.

The following diagram shows the progress and the results.



The following sections provide more detail regarding the stack implementation and applicable stack operations and instructions.

9.2 Stack Instructions

A push operation puts things onto the stack, and a pop operation takes things off the stack. The format for these commands is:

```
push    <operand64>
pop     <operand64>
```

The operand can be a register or memory, but an immediate is not allowed. In general, push and pop operations will push the architecture size. Since the architecture is 64-bit, we will push and pop quadwords.

The stack is implemented in reverse in memory. Refer to the following sections for a detailed explanation of why.

The stack instructions are summarized as follows:

Instruction	Explanation
`push <op64>`	Push the 64-bit operand on the stack. First, adjusts **rsp** accordingly (**rsp**-8) and then copy the operand to [**rsp**]. The operand may not be an immediate value. Operand is not changed.

Examples:	`push rax`
	`push qword [qVal] ; value`
	`push qVal ; address`

`pop <op64>`	Pop the 64-bit operand from the stack. Adjusts **rsp** accordingly (**rsp**+8). The operand may not be an immediate value. Operand is overwritten.

Examples:	`pop rax`
	`pop qword [qVal]`
	`pop rsi`

If more than 64-bits must be pushed, multiple push operations would be required. While it is possible to push and pop operands less than 64-bits, it is not recommended.

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

9.3 Stack Implementation

The **rsp** register is used to point to the current top of stack in memory. In this architecture, as with most, the stack is implemented growing downward in memory.

9.3.1 Stack Layout

As noted in Chapter 2, Architecture, the general memory layout for a program is as follows:

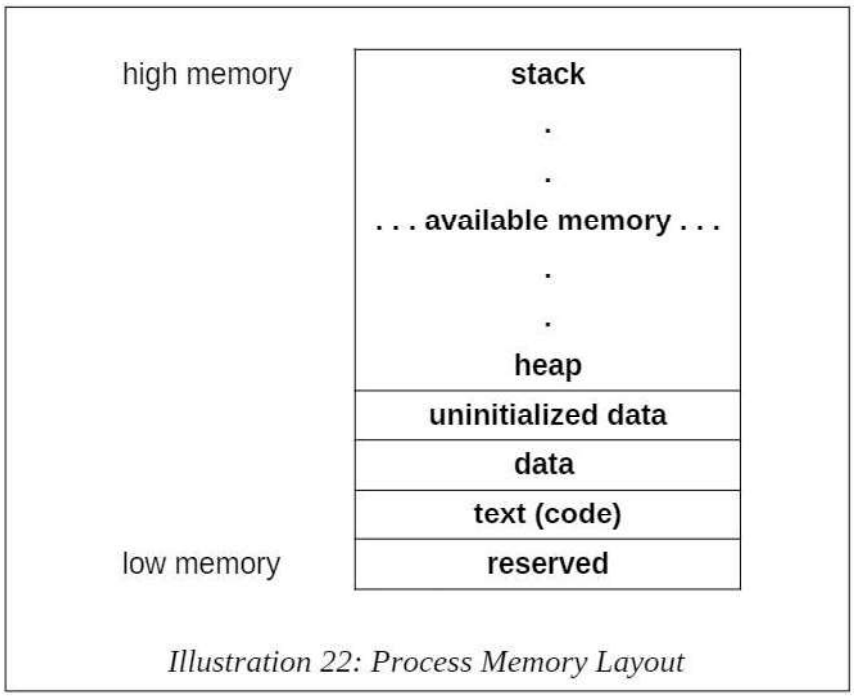


Illustration 22: Process Memory Layout

The heap is where dynamically allocated data will be placed (if requested). For example, items allocated with the C++ **new** operator or the C **malloc()** system call. As dynamically allocated data is created (at run-time), the heap typically grows upward. However, the stack starts in high memory and grows downward. The stack is used to temporarily store information such as call frames for function calls. A large program or a recursive function may use a significant amount of stack space.

As the heap and stack expand, they grow toward each other. This is done to ensure the most effective overall use of memory.

A program (Process A) that uses a significant amount of stack space and a minimal amount of heap space will function. A program (Process B) that uses a minimal amount of stack space and a very large amount of heap space will also function.

For example:

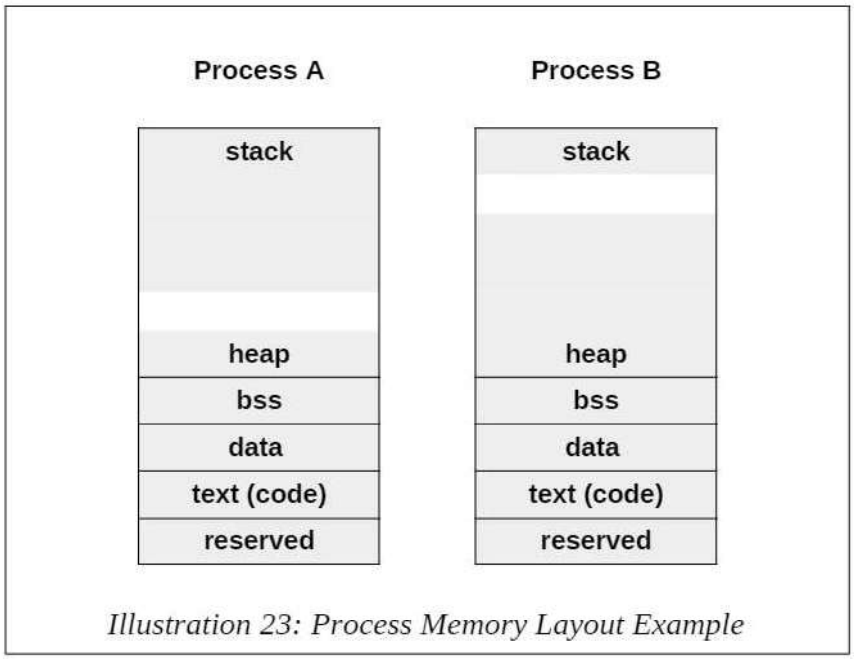


Illustration 23: Process Memory Layout Example

Of course, if the stack and heap meet, the program will crash. If that occurs, there is no memory available.

9.3.2 Stack Operations

The basic stack operations of push and pop adjust the stack pointer register, **rsp**, during their operation.

For a push operation:

1. The **rsp** register is decreased by 8 (1 quadword).
2. The operand is copied to the stack at **[rsp]**.

The operand is not altered. The order of these operations is important.

For a pop operation:

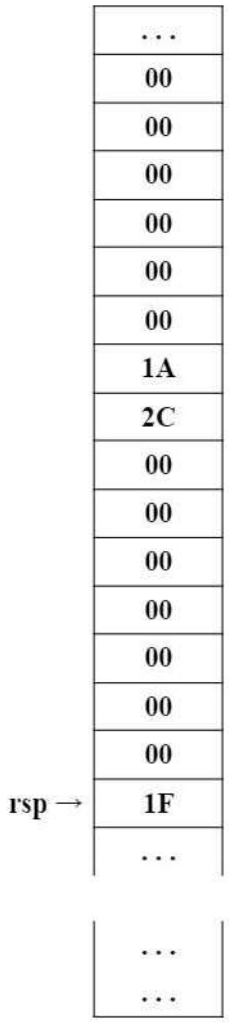
1. The current top of the stack, at **[rsp]**, is copied into the operand.
2. The **rsp** register is increased by 8 (1 quadword).

The order of these operations is the exact reverse of the push. The item popped is not actually deleted. However, the programmer cannot count on the item remaining on the stack after the pop operation. Previously pushed, but not popped, items can be accessed.

For example:

```
mov     rax, 6700          ; 6700₁₀ = 00001A2C₁₆
push    rax
mov     rax, 31            ;   31₁₀ = 0000001F₁₆
push    rax
```

Would produce the following stack configuration (where each box is a byte):



The layout shows the architecture is little-endian in that the least significant byte is placed into the lowest memory location.

9.4 Stack Example

The following is an example program to use the stack to reverse a list of quadwords in place. Specifically, each value in a quadword array is placed on the stack in the first loop. In the second loop, each element is removed from the stack and placed back into the array (over-writing) the previous value.

```
; Simple example demonstrating basic stack operations.

; Reverse a list of numbers - in place.
; Method: Put each number on stack, then pop each number
;         back off, and then put back into memory.

; ***************************************************
;  Data declarations

section    .data

; -----
;  Define constants

EXIT_SUCCESS   equ    0       ; successful operation
SYS_exit       equ    60      ; call code for terminate

; -----
;  Define Data.

numbers        dq     121, 122, 123, 124, 125
len            dq     5

; ***************************************************
section    .text
global _start
_start:

;  Loop to put numbers on stack.

    mov    rcx, qword [len]
    mov    rbx, numbers
    mov    r12, 0
    mov    rax, 0

pushLoop:
    push   qword [rbx+r12*8]
```

```
    inc     r12
    loop    pushLoop

; -----
;  All the numbers are on stack (in reverse order).
;  Loop to get them back off.   Put them back into
;  the original list...

    mov     rcx, qword [len]
    mov     rbx, numbers
    mov     r12, 0
popLoop:
    pop     rax
    mov     qword [rbx+r12*8], rax
    inc     r12
    loop    popLoop

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

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

There are other ways to accomplish this function (reversing a list), however this is meant to demonstrate the stack operations.

9.5 Exercises

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

9.5.1 Quiz Questions

Below are some quiz questions based on this chapter.

1) Which register refers to the top of the stack?
2) What happens as a result of a **`push rax`** instruction (two things)?
3) How many ***bytes*** of data does the **`pop rax`** instruction remove from the stack?

4) Given the following code fragment:

```
        mov     r10, 1
        mov     r11, 2
        mov     r12, 3
        push    r10
        push    r11
        push    r12
        pop     r10
        pop     r11
        pop     r12
```

What would be in the **r10** , **r11**, and **r12** registers after execution? Show answer in hex, full register size.

5) Given the following variable declarations and code fragment:

```
lst     dq   1, 3, 5, 7, 9

        mov     rsi, 0
        mov     rcx, 5
lp1:    push    qword [lst+rsi*8]
        inc     rsi
        loop    lp1
        mov     rsi, 0
        mov     rcx, 5
lp2:    pop     qword [lst+rsi*8]
        inc     rsi
        loop    lp2
        mov     rbx, qword [lst]
```

Explain what would be the ***result*** of the code (after execution)?

6) Provide one advantage to the stack growing downward in memory.

9.5.2 Suggested Projects

Below are some suggested projects based on this chapter.

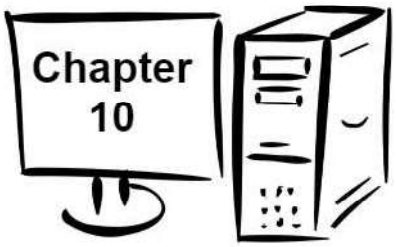
1) Implement the example program to reverse a list of numbers. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

2) Create a program to determine if a NULL terminated string representing a word is a palindrome[39]. A palindrome is a word that reads the same forward or backwards. For example, "anna", "civic", "hannah", "kayak", and "madam" are palindromes. This can be accomplished by pushing the characters on the stack one at a time and then comparing the stack items to the string starting from the beginning. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

3) Update the previous program to test if a phrase is a palindrome. The general approach using the stack is the same, however, spaces and punctuation must be skipped. For example, "A man, a plan, a canal – Panama!" is a palindrome. The program must ignore the comma, dash, and exclamation point. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

39 For more information, refer to: http://en.wikipedia.org/wiki/Palindrome

CAPS LOCK – Preventing login since 1980.



10.0 Program Development

Writing or developing programs is easier when following a clear methodology. The main steps in the methodology are:

- Understand the Problem
- Create the Algorithm
- Implement the Program
- Test/Debug the Program

To help demonstrate this process in detail, these steps will be applied to a simple example problem in the following sections.

10.1 Understand the Problem

Before attempting to create a solution, it is important to fully understand the problem. Ensuring a complete understanding of the problem can help reduce errors and save time and effort. The first step is to understand what is required, especially the applicable input information and expected results or output.

Consider the problem of converting a single integer number into a string or series of characters representing that integer. To be clear, an integer can be used for numeric calculations, but cannot be displayed to the console (as it is). A string can be displayed to the console but not used in numeric calculations.

For this example, only unsigned (positive only) values will be considered. The small extra effort to address signed values is left to the reader as an exercise.

As an unsigned double-word integer, the numeric value 1498_{10} would be represented as 0x000005DA in hex (double-word sized). The integer number 1498_{10} (0x000005DA) would be represented by the string "1", "4", "9", "8" with a NULL termination. This would require a total of 5 bytes since there is no sign or leading spaces required for this

specific example. As such, the string "1498" would be represented as follows:

Character	"1"	"4"	"9"	"8"	NULL
ASCII Value (decimal)	49	52	57	56	0
ASCII Value (hex)	0x31	0x34	0x39	0x38	0x0

The goal is to convert the single integer number into the appropriate series of characters to form a NULL terminated string.

10.2 Create the Algorithm

The algorithm is the name for the unambiguous, ordered sequence of steps involved in solving the problem. Once the program is understood, a series of steps can be developed to solve that problem. There can be, and usually are, multiple correct solutions to a given problem.

The process for creating an algorithm can be different for different people. In general, some time should be devoted to thinking about possible solutions. This may involve working on some possible solutions using a scratch piece of paper. Once an approach is selected, that solution can be developed into an algorithm. The algorithm should be written down, reviewed, and refined. The algorithm is then used as the outline of the program.

For example, we will consider the integer to ASCII conversion problem outlined in the previous section. To convert a single digit integer (0-9) into a character, 48_{10} (or "0" or 0x30) can be added to the integer. For example, 0x01 + 0x30 is 0x31 which is the ASCII value of "1". It should be obvious that this trick will only work for single digit numbers (0-9).

In order to convert a larger integer (10) into a string, the integer must be broken into its component digits. For example, 123_{10} (0x7B) would be 1, 2, and 3. This can be accomplished by repeatedly performing integer division by 10 until a 0 result is obtained.

For example;

$$\frac{123}{10} = 12 \quad remainder 3$$

$$\frac{12}{10} = 1 \quad remainder 2$$

$$\frac{1}{10} = 0 \quad remainder 1$$

As can be seen, the remainder represents the individual digits. However, they are obtained in reverse order. To address this, the program can push the remainder and, when done dividing, pop the remainders and convert to ASCII and store in a string (which is an array of bytes).

This process forms the basis for the algorithm. It should be noted, that there are many ways to develop this algorithm. One such approach is shown as follows:

```
;  Part A - Successive division
;       digitCount = 0
;       get integer
;   divideLoop:
;       divide number by 10
;       push remainder
;       increment digitCount
;       if (result > 0) goto divideLoop

;  Part B - Convert remainders and store
;       get starting address of string (array of bytes)
;       idx = 0
;   popLoop:
;       pop intDigit
;       charDigit = intDigit + "0" (0x030)
;       string[idx] = charDigit
;       increment idx
;       decrement digitCount
;       if (digitCount > 0) goto popLoop
;       string[idx] = NULL
```

The algorithm steps are shown as program comments for convenience. The algorithm is typically started on paper and then more formally written in pseudo-code as shown above. In the unlikely event the program does not work the first time, the comments are the primary debugging checklist.

Some programmers skip the comments and will end up spending much more time debugging. The commenting represents the algorithm and the code is the implementation of that algorithm.

10.3 Implement the Program

Based on the algorithm, a program can be developed and implemented. The algorithm is expanded and the code added based on the steps outlined in the algorithm. This allows the programmer to focus on the specific issues for the current section being coded including the data types and data sizes. This example addresses only unsigned data so the unsigned divide (DIV, not IDIV) is used. Since the integer is a double-word, it must be converted into a quadword for the division. However, the result and the remainder after division will also be a double-words. Since the stack is quadwords, the entire quadword register will be pushed. The upper-order portion of the register will not be accessed, so its contents are not relevant.

One possible implementation of the algorithm is as follows:

```
;  Simple example program to convert an
;  integer into an ASCII string.

; ******************************************************
;  Data declarations

section    .data

; -----
;  Define constants

NULL              equ     0
EXIT_SUCCESS      equ     0             ; successful operation
SYS_exit          equ     60            ; code for terminate

; -----
;  Define Data.

intNum         dd      1498
```

```
section    .bss
strNum         resb    10

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

section    .text
global _start
_start:

; Convert an integer to an ASCII string.

; -----
;  Part A - Successive division

    mov     eax, dword [intNum]          ; get integer
    mov     rcx, 0                       ; digitCount = 0
    mov     ebx, 10                      ; set for dividing by 10

divideLoop:
    mov     edx, 0
    div     ebx                          ; divide number by 10

    push    rdx                          ; push remainder
    inc     rcx                          ; increment digitCount

    cmp     eax, 0                       ; if (result > 0)
    jne     divideLoop                   ;   goto divideLoop

; -----
;  Part B - Convert remainders and store

    mov     rbx, strNum                  ; get addr of string
    mov     rdi, 0                       ; idx = 0

popLoop:
    pop     rax                          ; pop intDigit

    add     al, "0"                      ; char = int + "0"

    mov     byte [rbx+rdi], al           ; string[idx] = char
    inc     rdi                          ; increment idx
    loop    popLoop                      ; if (digitCount > 0)
                                         ;   goto popLoop
    mov     byte [rbx+rdi], NULL         ; string[idx] = NULL
```

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

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

There are many different valid implementations for this algorithm. The program should be assembled to address any typos or syntax errors.

10.4 Test/Debug the Program

Once the program is written, testing should be performed to ensure that the program works. The testing will be based on the specific parameters of the program.

In this case, the program can be executed using the debugger and stopped near the end of the program (e.g., at the label "last" in this example). After starting the debugger with **ddd**, the command **`b last`** and **`run`** can be entered which will run the program up to, but not executing the line referenced by the label "last". The resulting string, ***strNum*** can be viewed in the debugger with **`x/s &strNum`** will display the string address and the contents which should be "1498". For example;

```
(gdb) x/s &strNum
0x600104:  "1498"
```

If the string is not displayed properly, it might be worth checking each character of the five (5) byte array with the **`x/5cb &strNum`** debugger command. The output will show the address of the string followed by both the decimal and ASCII representation.

For example;

```
(gdb) x/5cb &strNum
0x600104:  49 '1' 52 '4' 57 '9' 56 '8' 0 '\000'
```

The format of this output can be confusing initially.

If the correct output is not provided, the programmer will need to debug the code. For this example, there are two main steps; successive division and conversion/storing the remainders. The second step requires the first step to work, so the first step should be verified. This can be done by using the debugger to focus only on the first section. In this example, the first step should iterate exactly 4 times, so **rcx** should be 4. Additionally, 8, 9, 4, and 1 should be pushed on the stack in that order. This is easily

verified in the debugger by looking at the register contents of **rdx** when it is pushed or by viewing the top 4 entries in the stack.

If that section works, the second section can be verified. Here, the values 1, 4, 9, and 8 should be coming off the stack (in that order). If so, the integer is converted into a character by adding "0" (0x30) and that stored in the string, one character at a time. The string can be viewed character by character to see if they are being entered into the string correctly.

In this manner, the problem can be narrowed down fairly quickly. Efficient debugging is a critical skill and must be honed by practice.

Refer to Chapter 6, DDD Debugger for additional information on specific debugger commands.

10.5 Error Terminology

In case the program does not work, it helps to understand some basic terminology about where or what the error might be. Using the correct terminology ensures that you can communicate effectively about the problem with others.

10.5.1 Assembler Error

Assembler errors are generated when the program is assembled. This means that the assembler does not understand one or more of the instructions. The assembler will provide a list of errors and the line number of each error. It is recommended to address the errors from the top down. Resolving an error at the top can clear multiple errors further down.

Typical assembler errors include misspelling an instruction and/or omitting a variable declaration.

10.5.2 Run-time Error

A run-time error is something that causes the program to crash.

10.5.3 Logic Error

A logic error is when the program executes, but does not produce the correct result. For example, coding a provided formula incorrectly or attempting to compute the average of a series of numbers before calculating the sum.

If the program has a logic error, one way to find the error is to display intermediate values. Further information will be provided in later chapters regarding advice on finding logic errors.

10.6 Exercises

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

10.6.1 Quiz Questions

Below are some quiz questions based on this chapter.

1) What is an algorithm?
2) What are the four main steps in algorithm development?
3) Are the four main steps in algorithm development applicable only to assembly language programming?
4) What type of error, if any, occurs if the one operand multiply instruction uses an immediate value operand? Respond with assemble-time or run-time.
5) If an assembly language instruction is spelled incorrectly (e.g., "mv" instead of "mov"), when will the error be found? Respond with assemble-time or run-time.
6) If a label is referenced, but not defined, when will the error be found? Respond with assemble-time or run-time.
7) If a program performing a series of divides on values in an array divides by 0, when will the error be found? Respond with assemble-time or run-time.

10.6.2 Suggested Projects

Below are some suggested projects based on this chapter.

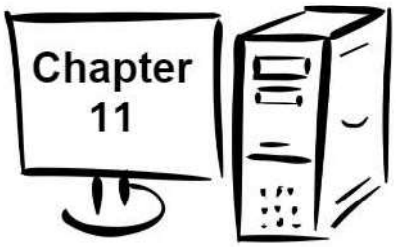
1) Implement the example program to convert an integer into a string. Change the original integer to a different value. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.
2) Update the example program to address signed integers. This will require including a preceding sign, "+" or "-" in the string. For example, -123_{10} (0xFFFFFF85) would be "-123" with a NULL termination (total of 5 bytes). Additionally, the signed divide (IDIV, not DIV) and signed conversions (e.g., CDQ) must be used. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

3) Create a program to convert a string representing a numeric value into an integer. For example, given the NULL terminated string "41275" (a total of 6 bytes), convert the string into a double-word sized integer (0x0000A13B). You may assume the string and resulting integer is unsigned. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

4) Update the previous program to address strings with a preceding sign ("+" or "-"). This will require including a sign, "+" or "-" in the string. You must ensure the final string is NULL terminated. You may assume the input strings are valid. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

5) Update the previous program to convert strings into integers to include error checking on the input string. Specifically, the sign must be valid and be the first character in the string, each digit must be between "0" and "9", and the string NULL terminated. For example, the string "-321" is valid while "1+32" and "+1R3" are both invalid. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

Why did C++ decide not to go out with C?
Because C has no class.



11.0 Macros

An assembly language macro is a predefined set of instructions that can easily be inserted wherever needed. Once defined, the macro can be used as many times as necessary. It is useful when the same set of code must be utilized numerous times. A macro can be useful to reduce the amount of coding, streamline programs, and reduce errors from repetitive coding.

The assembler contains a powerful macro processor, which supports conditional assembly, multi-level file inclusion, and two forms of macros (single-line and multi-line), and a 'context stack' mechanism for extra macro power.

Before using a macro, it must be defined. Macro definitions should be placed in the source file ***before*** the data and code sections. The macro is used in the text (code) section. The following sections will present a detailed example with the definition and use.

11.1 Single-Line Macros

There are two key types of macros; single-line macros and multi-line macros. Each of these is described in the following sections.

Single-line macros are defined using the **`%define`** directive. The definitions work in a similar way to C/C++; so you can do things like:

```
%define   mulby4(x)   shl x, 2
```

And, then use the macro by entering:

```
mulby4  (rax)
```

in the source, which will multiply the contents to the **rax** register by 4 (via shifting two bits).

11.2 Multi-Line Macros

Multi-line macros can include a varying number of lines (including one). The multi-line macros are more useful and the following sections will focus primarily on multi-line macros.

11.2.1 Macro Definition

Before using a multi-line macro, it must first be defined. The general format is as follows:

```
%macro  <name>  <number of arguments>

  ; [body of macro]

%endmacro
```

The arguments can be referenced within the macro by **%<number>**, with **%1** being the first argument, and **%2** the second argument, and so forth.

In order to use labels, the labels within the macro must be prefixing the label name with a **%%**.

This will ensure that calling the same macro multiple times will use a different label each time. For example, a macro definition for the absolute value function would be as follows:

```
%macro  abs  1
      cmp  %1, 0
      jge  %%done
      neg  %1
%%done:

%endmacro
```

Refer to the sample macro program for a complete example.

11.2.2 Using a Macro

In order to use or "invoke" a macro, it must be placed in the code segment and referred to by name with the appropriate number of arguments.

Given a data declaration as follows:

```
qVar    dq    4
```

Then, to invoke the "abs" macro (twice):

```
mov  eax, -3
abs  eax

abs  qword [qVar]
```

The list file will display the code as follows (for the first invocation):

```
27 00000000 B8FDFFFFFF          mov  eax, -3
28                              abs  eax
29 00000005 3D00000000  <1>  cmp %1, 0
30 0000000A 7D02        <1>  jge %%done
31 0000000C F7D8        <1>  neg %1
32                      <1> %%done:
```

The macro will be copied from the definition into the code, with the appropriate arguments replaced in the body of the macro, *each* time it is used. The **<1>** indicates code copied from a macro definition. In both cases, the **%1** argument was replaced with the given argument; **eax** in this example.

Macros use more memory, but do not require overhead for transfer of control (like functions).

11.3 Macro Example

The following example program demonstrates the definition and use of a simple macro.

```
; Example Program to demonstrate a simple macro

; ************************************************
;  Define the macro
;   called with three arguments:
;       aver  <lst>, <len>, <ave>

%macro     aver     3
    mov    eax, 0
    mov    ecx, dword [%2]          ; length
    mov    r12, 0
    lea    rbx, [%1]
```

```
%%sumLoop:
    add     eax, dword [rbx+r12*4]    ; get list[n]
    inc     r12
    loop    %%sumLoop

    cdq
    idiv    dword [%2]
    mov     dword [%3], eax

%endmacro

; ************************************************
;  Data declarations

section    .data

; -----
;  Define constants

EXIT_SUCCESS   equ    0                 ; success code
SYS_exit       equ    60                ; code for terminate

;   Define Data.

section    .data
list1      dd   4, 5, 2, -3, 1
len1       dd   5
ave1       dd   0

list2      dd   2, 6, 3, -2, 1, 8, 19
len2       dd   7
ave2       dd   0
; ************************************************

section    .text
global _start
_start:

; -----
;  Use the macro in the program

    aver   list1, len1, ave1               ; 1st, data set 1
```

```
    aver    list2, len2, ave2                ; 2nd, data set 2

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

last:
    mov rax, SYS_exit                        ; exit
    mov rdi, EXIT_SUCCESS                    ; success
    syscall
```

In this example, the macro is invoked twice. Each time the macro is used, it is copied from the definition into the text section. As such, macros typically use more memory.

11.4 Debugging Macros

The code for a macro will not be displayed in the debugger source window. When a macro is working correctly, this is very convenient. However, when debugging macros, the code must be viewable.

In order to see the macro code, display the machine code window (**View → Machine Code Window**). In the window, the machine code for the instructions are displayed. The step and next instructions will execute the entire macro. In order to execute the macro instructions, the **stepi** and **nexti** commands must be used.

The code, when viewed, will be the expanded code (as opposed to the original macro's definition).

11.5 Exercises

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

11.5.1 Quiz Questions

Below are some quiz questions based on this chapter.

1) Where is the macro definition placed in the assembly language source file?
2) When a macro is invoked, how many times is the code placed in the code segment?
3) Explain why, in a macro, labels are typically preceded by a **%%** (double percent sign).
4) Explain what might happen if the **%%** is not included on a label?
5) Is it legal to jump to a label that does not include the **%%**? If not legal, explain why. If legal, explain under what circumstances that might be useful.

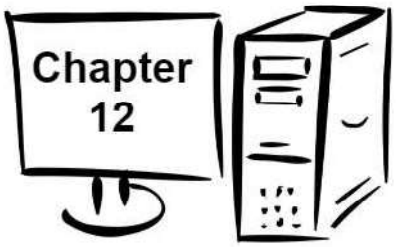
6) When does the macro argument substitution occur?

11.5.2 Suggested Projects

Below are some suggested projects based on this chapter.

1) Implement the example program for a list average macro. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.
2) Update the program from the previous question to include the minimum and maximum values. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.
3) Create a macro to update an existing list by multiplying every element by 2. Invoke the macro at least three times of three different data sets. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.
4) Create a macro from the integer to ASCII conversion example from the previous chapter. Invoke the macro at least three times of three different data sets. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

Why do programmers mix up Halloween and Christmas?
Because 31 Oct = 25 Dec.



12.0 Functions

Functions and procedures (i.e., void functions) help break-up a program into smaller parts making it easier to code, debug, and maintain. Function calls involve two main actions:

- Linkage
  - Since the function can be called from multiple different places in the code, the function must be able to return to the correct place in which it was originally called.
- Argument Transmission
  - The function must be able to access parameters to operate on or to return results (i.e., access call-by-reference parameters).

The specifics of how each of these actions are accomplished is explained in the following sections.

12.1 Updated Linking Instructions

When writing and debugging functions, it is easier for the C compiler (either GCC or G++) to link the program as the C compiler is aware of the appropriate locations for the various C/C++ libraries.

For example, assuming that the source file is named *example.asm*, the commands to compile, assemble, link, and execute as follows:

```
yasm -g dwarf2 -f elf64 example.asm -l example.lst
gcc -g -o example example.o
```

Note, Ubuntu 18 will require the **no-pie** option on the gcc command as shown:

```
gcc -g -no-pie -o example example.o
```

This will use the GCC compiler to call the linker, reading the *example.o* object file and creating the *example* executable file. The "-g" option includes the debugging information in the executable file in the usual manner. The file names can be changed as desired.

12.2 Debugger Commands

When using the debugger to debug programs with functions, a review of the ***step*** and ***next*** debugger commands may be helpful.

12.2.1 Debugger Command, *next*

With respect to a function call, the debugger ***next*** command will execute the entire function and go to the next line. When debugging functions, this is useful to quickly execute the entire function and then just verify the results. It will not display any of the function code.

12.2.2 Debugger Command, *step*

With respect to a function call, the debugger ***step*** command will step into the function and go to the first line of the function code. It will display the function code. When debugging functions, this is useful to debug the function code.

12.3 Stack Dynamic Local Variables

In a high-level language, non-static local variables declared in a function are stack dynamic local variables by default. Some C++ texts refer to such variables as automatics. This means that the local variables are created by allocating space on the stack and assigning these stack locations to the variables. When the function completes, the space is recovered and reused for other purposes. This requires a small amount of additional run-time overhead, but makes a more efficient overall use of memory. If a function with a large number of local variables is never called, the memory for the local variables is never allocated. This helps reduce the overall memory footprint of the program which generally helps the overall performance of the program.

In contrast, statically declared variables are assigned memory locations for the entire execution of the program. This uses memory even if the associated function is not being executed. However, no additional run-time overhead is required to allocate the space since the space allocation has already been performed (when the program was initially loaded into memory).

12.4 Function Declaration

A function must be written before it can be used. Functions are located in the code segment. The general format is:

```
global <procName>
<procName>:

    ; function body

ret
```

A function may be defined only once. There is no specific order required for how functions are defined. However, functions cannot be nested. A function definition should be started and ended before the next function's definition can be started.

Refer to the sample functions for examples of function declarations and usage.

12.5 Standard Calling Convention

To write assembly programs, a standard process for passing parameters, returning values, and allocating registers between functions is needed. If each function did these operations differently, things would quickly get very confusing and require programmers to attempt to remember for each function how to handle parameters and which registers were used. To address this, a standard process is defined and used which is typically referred to as a *standard calling convention*[40]. There are actually a number of different standard calling conventions. The 64-bit C calling convention, called **System V AMD64 ABI**[41] [42], is described in the remainder of this document.

This calling convention is also used for C/C++ programs by default. This means that interfacing assembly language code and C/C++ code is easily accomplished since the same calling convention is used.

It must be noted that the standard calling convention presented here applies to Linux-based operating systems. The standard calling convention for Microsoft Windows is slightly different and not presented in this text.

40 For more information, refer to: http://en.wikipedia.org/wiki/Calling_convention
41 For more information, refer to: https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI
42 For complete details, refer to: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

12.6 Linkage

The linkage is about getting to and returning from a function call correctly. There are two instructions that handle the linkage, `call <funcName>` and `ret` instructions.

The `call` transfers control to the named function, and `ret` returns control back to the calling routine.

- The `call` works by saving the address of where to return to when the function completes (referred to as the *return address*). This is accomplished by placing contents of the **rip** register on the stack. Recall that the **rip** register points to the next instruction to be executed (which is the instruction immediately after the call).
- The `ret` instruction is used in a procedure to return. The `ret` instruction pops the current top of the stack (**rsp**) into the **rip** register. Thus, the appropriate return address is restored.

Since the stack is used to support the linkage, it is important that within the function the stack must not be corrupted. Specifically, any items pushed must be popped. Pushing a value and not popping would result in that value being popped off the stack and placed in the **rip** register. This would cause the processor to attempt to execute code at that location. Most likely the invalid location will cause the process to crash.

The function calling or linkage instruction is summarized as follows:

Instruction	Explanation
`call <funcName>`	Calls a function. Push the 64-bit **rip** register and jump to the <***funcName***>.
Examples:	`call printString`
`ret`	Return from a function. Pop the stack into the **rip** register, effecting a jump to the line after the call.
Examples:	`ret`

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

12.7 Argument Transmission

Argument transmission refers to sending information (variables, etc.) to a function and obtaining a result as appropriate for the specific function.

The standard terminology for transmitting values to a function is referred to as *call-by-value*. The standard terminology for transmitting addresses to a function is referred to as *call-by-reference*. This should be a familiar topic from a high-level language.

There are various ways to pass arguments to and/or from a function.

- Placing values in register
  - Easiest, but has limitations (i.e., the number of registers).
  - Used for first six integer arguments.
  - Used for system calls.
- Globally defined variables
  - Generally poor practice, potentially confusing, and will not work in many cases.
  - Occasionally useful in limited circumstances.
- Putting values and/or addresses on stack
  - No specific limit to count of arguments that can be passed.
  - Incurs higher run-time overhead.

In general, the calling routine is referred to as the ***caller*** and the routine being called is referred to as the ***callee***.

12.8 Calling Convention

The function *prologue* is the code at the beginning of a function and the function *epilogue* is the code at the end of a function. The operations performed by the prologue and epilogue are generally specified by the standard calling convention and deal with stack, registers, passed arguments (if any), and stack dynamic local variables (if any).

The general idea is that the program state (i.e., contents of specific registers and the stack) are saved, the function executed, and then the state is restored. Of course, the function will often require extensive use of the registers and the stack. The prologue code helps save the state and the epilogue code restores the state.

12.8.1 Parameter Passing

As noted, a combination of registers and the stack is used to pass parameters to and/or from a function.

The first six integer arguments are passed in registers as follows:

Argument Number	Argument Size			
	64-bits	32-bits	16-bits	8-bits
1	`rdi`	`edi`	`di`	`dil`
2	`rsi`	`esi`	`si`	`sil`
3	`rdx`	`edx`	`dx`	`dl`
4	`rcx`	`ecx`	`cx`	`cl`
5	`r8`	`r8d`	`r8w`	`r8b`
6	`r9`	`r9d`	`r9w`	`r9b`

The seventh and any additional arguments are passed on the stack. The standard calling convention requires that, when passing arguments (values or addresses) on the stack, the arguments should be pushed in reverse order. That is "`someFunc (one, two, three, four, five, six, seven, eight, nine)`" would imply a push order of: *nine*, *eight*, and then *seven*.

For floating-point arguments, the floating-point registers **xmm0** to **xmm7** are used in that order for the first eight float arguments.

Additionally, when the function is completed, the calling routine is responsible for clearing the arguments from the stack. Instead of doing a series of pop instructions, the stack pointer, **rsp**, is adjusted as necessary to clear the arguments off the stack. Since each argument is 8 bytes, the adjustment would be adding [(number of arguments) * 8] to the **rsp**.

For value returning functions, the result is placed in the **A** register based on the size of the value being returned.

Specifically, the values are returned as follows:

Return Value Size	Location
byte	`al`
word	`ax`
double-word	`eax`
quadword	`rax`
floating-point	`xmm0`

The **rax** register may be used in the function as needed as long as the return value is set appropriately before returning.

12.8.2 Register Usage

The standard calling convention specifies the usage of registers when making function calls. Specifically, some registers are expected to be preserved across a function call. That means that if a value is placed in a *preserved register* or *saved register*, and the function must use that register, the original value must be preserved by placing it on the stack, altered as needed, and then restored to its original value before returning to the calling routine. This register preservation is typically performed in the prologue and the restoration is typically performed in the epilogue.

The following table summarizes the register usage.

Register	Usage
`rax`	Return Value
`rbx`	Callee Saved
`rcx`	4th Argument
`rdx`	3rd Argument
`rsi`	2nd Argument
`rdi`	1st Argument
`rbp`	Callee Saved
`rsp`	Stack Pointer
`r8`	5th Argument
`r9`	6th Argument
`r10`	Temporary

r11	Temporary
r12	Callee Saved
r13	Callee Saved
r14	Callee Saved
r15	Callee Saved

The temporary registers (**r10** and **r11**) and the argument registers (**rdi**, **rsi**, **rdx**, **rcx**, **r8**, and **r9**) are not preserved across a function call. This means that any of these registers may be used in the function without the need to preserve the original value.

Additionally, none of the floating-point registers are preserved across a function call. Refer to Chapter 18 for more information regarding floating-point operations.

12.8.3 Call Frame

The items on the stack as part of a function call are referred to as a *call frame* (also referred to as an *activation record* or *stack frame*). Based on the standard calling convention, the items on the stack, if any, will be in a specific general format.

The possible items in the call frame include:

- Return address (required).
- Preserved registers (if any).
- Passed arguments (if any).
- Stack dynamic local variables (if any).

Other items may be placed in the call frame such as static links for dynamically scoped languages. Such topics are outside the scope of this text and will not be discussed here.

For some functions, a full call frame may not be required. For example, if the function:

- Is a leaf function (i.e., does not call another function).
- Passes its arguments only in registers (i.e., does not use the stack).
- Does not alter any of the saved registers.
- Does not require stack-based local variables.

This can occur for simpler, smaller leaf functions. However, if any of these conditions is not true, a full call frame is required.

For more non-leaf or more complex functions, a more complete call frame is required.

The standard calling convention does not explicitly require use of the frame pointer register, **rbp**. Compilers are allowed to optimize the call frame and not use the frame pointer. To simplify and clarify accessing stack-based arguments (if any) and stack dynamic local variables, this text will utilize the frame pointer register. This is similar to how many other architectures use a frame pointer register.

As such, if there are any stack-based arguments or any local variables needed within a function, the frame pointer register, **rbp**, should be pushed and then set pointing to itself. As additional pushes and pops are performed (thus changing **rsp**), the **rbp** register will remain unchanged. This allows the **rbp** register to be used as a reference to access arguments passed on the stack (if any) or stack dynamic local variables (if any).

For example, assuming a function call has eight (8) arguments and assuming the function uses **rbx**, **r12**, and **r13** registers (and thus must be pushed), the call frame would be as follows:

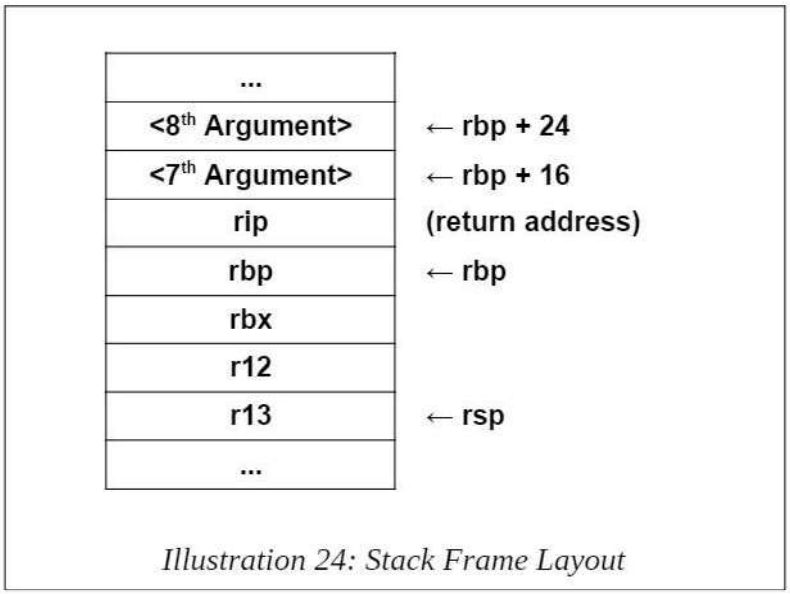


Illustration 24: Stack Frame Layout

The stack-based arguments are accessed relative to the **rbp**. Each item push is a quadword which uses 8 bytes. For example, **[rbp+16]** is the location of the first passed argument (7^{th} integer argument) and **[rbp+24]** is the location of the second passed argument (8^{th} integer argument).

In addition, the call frame would contain the assigned locations of local variables (if any). The section on local variables details the specifics regarding allocating and using local variables.

12.8.3.1 Red Zone

In the Linux standard calling convention, the first 128-bytes after the stack pointer, **rsp**, are reserved. For example, extending the previous example, the call frame would be as follows:

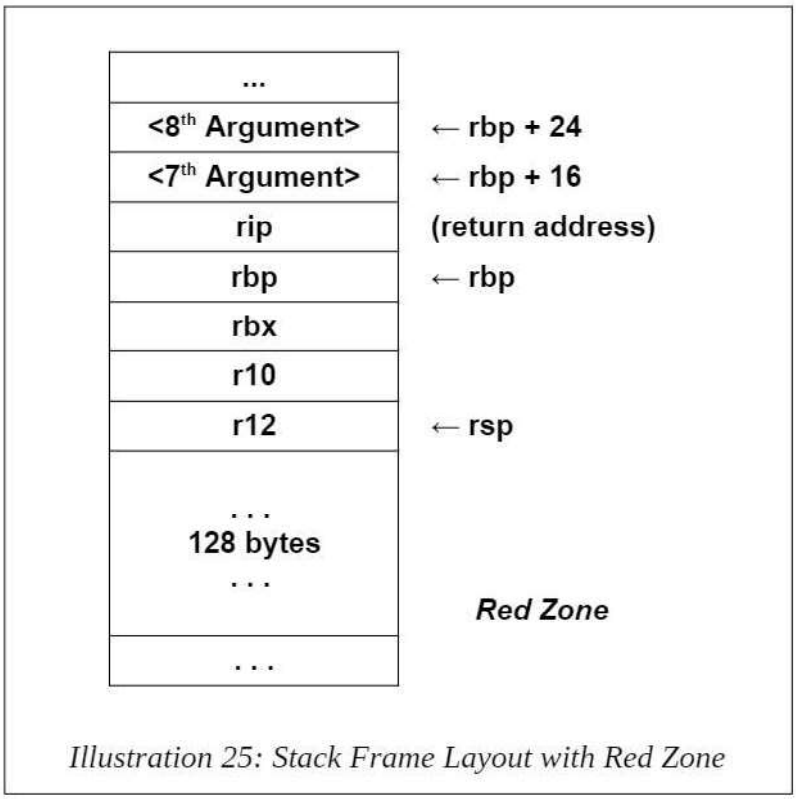


Illustration 25: Stack Frame Layout with Red Zone

This red zone may be used by the function without any adjustment to the stack pointer. The purpose is to allow compiler optimizations for the allocation of local variables. This does not directly impact programs written directly in assembly language.

12.9 Example, Statistical Function 1 (leaf)

This simple example will demonstrate calling a simple void function to find the sum and average of an array of numbers. The High-Level Language (HLL) call for C/C++ is as follows:

```
stats1(arr, len, sum, ave);
```

As per the C/C++ convention, the array, ***arr***, is call-by-reference and the length, ***len***, is call-by-value. The arguments for ***sum*** and ***ave*** are both call-by-reference (since there are no values as yet). For this example, the array ***arr***, ***sum***, and ***ave*** variables are all signed double-word integers. Of course, in context, the ***len*** must be unsigned.

12.9.1 Caller

In this case, there are 4 arguments, and all arguments are passed in registers in accordance with the standard calling convention. The assembly language code in the calling routine for the call to the stats function would be as follows:

```
; stats1(arr, len, sum, ave);
mov     rcx, ave                        ; 4th arg, addr of ave
mov     rdx, sum                        ; 3rd arg, addr of sum
mov     esi, dword [len]                ; 2nd arg, value of len
mov     rdi, arr                        ; 1st arg, addr of arr
call    stats1
```

There is no specific required order for setting the argument registers. This example sets them in reverse order in preparation for the next, extended example.

Note, the setting of the **esi** register also sets the upper-order double-word to zero, thus ensuring the **rsi** register is set appropriately for this specific usage since length is unsigned.

No return value is provided by this void routine. If the function was a value returning function, the value returned would be in the **A** register (of appropriate size).

12.9.2 Callee

The function being called, the callee, must perform the prologue and epilogue operations (as specified by the standard calling convention) before and after the code to perform the function goal. For this example, the function must perform the summation of values in the array, compute the integer average, return the sum and average values.

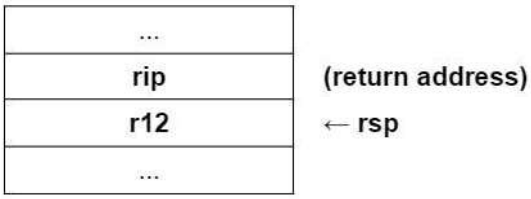
The following code implements the **stats1** example.

```
;  Simple example function to find and return
;  the sum and average of an array.

;  HLL call:
;   stats1(arr, len, sum, ave);
; -----
;  Arguments:
;   arr, address - rdi
;   len, dword value - esi
```

```
;   sum, address - rdx
;   ave, address - rcx

global stats1
stats1:
    push    r12                         ; prologue

    mov     r12, 0                      ; counter/index
    mov     rax, 0                      ; running sum
sumLoop:
    add     eax, dword [rdi+r12*4]      ; sum += arr[i]
    inc     r12
    cmp     r12, rsi
    jl      sumLoop

    mov dword [rdx], eax                ; return sum

    cdq
    idiv    esi                         ; compute average
    mov     dword [rcx], eax            ; return ave

    pop     r12                         ; epilogue
    ret
```

The choice of the **r12** register is arbitrary, however a 'saved register' was selected.

The call frame for this function would be as follows:



The minimal use of the stack helps reduce the function call run-time overhead.

12.10 Example, Statistical Function2 (non-leaf)

This extended example will demonstrate calling a simple void function to find the minimum, median, maximum, sum and average of an array of numbers.

The High-Level Language (HLL) call for C/C++ is as follows:

```
stats2(arr, len, min, med1, med2, max, sum, ave);
```

For this example, it is assumed that the array is sorted in ascending order. Additionally, for this example, the median will be the middle value. For an even length list, there are two middle values, ***med1*** and ***med2***, both of which are returned. For an odd length list, the single middle value is returned in both ***med1*** and ***med2***.

As per the C/C++ convention, the array, ***arr***, is call-by-reference and the length, ***len***, is call-by-value. The arguments for ***min***, ***med1***, ***med2***, ***max***, ***sum***, and ***ave*** are all call-by-reference (since there are no values as yet). For this example, the array ***arr***, ***min***, ***med1***, ***med2***, ***max***, ***sum***, and ***ave*** variables are all signed double-word integers. Of course, in context, the ***len*** must be unsigned.

12.10.1 Caller

In this case, there are 8 arguments and only the first six can be passed in registers. The last two arguments are passed on the stack. The assembly language code in the calling routine for the call to the stats function would be as follows:

```
; stats2(arr, len, min, med1, med2, max, sum, ave);
push    ave                             ; 8th arg, add of ave
push    sum                             ; 7th arg, add of sum
mov     r9, max                         ; 6th arg, add of max
mov     r8, med2                        ; 5th arg, add of med2
mov     rcx, med1                       ; 4th arg, add of med1
mov     rdx, min                        ; 3rd arg, addr of min
mov     esi, dword [len]                ; 2nd arg, value of len
mov     rdi, arr                        ; 1st arg, addr of arr
call    stats2
add     rsp, 16                         ; clear passed arguments
```

The 7th and 8th arguments are passed on the stack and pushed in reverse order in accordance with the standard calling convention. After the function is completed, the arguments are cleared from the stack by adjusting the stack point register (**rsp**). Since two arguments, 8 bytes each, were passed on the stack, 16 is added to the stack pointer.

Note, the setting of the **esi** register also sets the upper-order double-word to zero, thus ensuring the **rsi** register is set appropriately for this specific usage since length is unsigned.

No return value is provided by this void routine. If the function was a value returning function, the value returned would be in the **A** register.

12.10.2 Callee

The function being called, the callee, must perform the prologue and epilogue operations (as specified by the standard calling convention). Of course, the function must perform the summation of values in the array, find the minimum, medians, and maximum, compute the average, return all the values.

When call-by-reference arguments are passed on the stack, two steps are required to return the value.

- Get the address from the stack.
- Use that address to return the value.

A common error is to attempt to return a value to a stack-based location in a single step, which will not change the referenced variable. For example, assuming the double-word value to be returned is in the **eax** register and the 7th argument is call-by-reference and where the **eax** value is to be returned, the appropriate code would be as follows:

```
mov     r12, qword [rbp+16]
mov     dword [r12], eax
```

These steps cannot be combined into a single step. The following code

```
mov     dword [rbp+16], eax
```

Would overwrite the address passed on the stack and not change the reference variable.

The following code implements the **stats2** example.

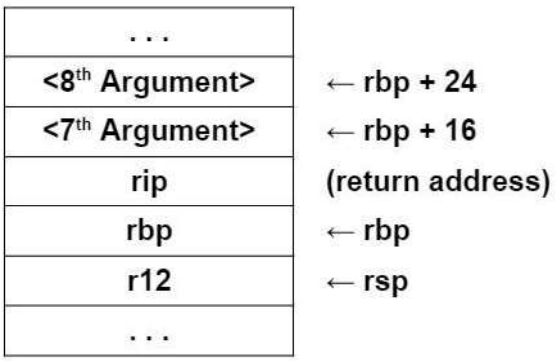
```
;  Simple example function to find and return the minimum,
;  maximum, sum, medians, and average of an array.
; -----
;  HLL call:
;   stats2(arr, len, min, med1, med2, max, sum, ave);

;  Arguments:
;   arr, address - rdi
;   len, dword value - esi
;   min, address - rdx
;   med1, address - rcx
;   med2, address - r8
;   max, address - r9
;   sum, address - stack (rbp+16)
```

```
;   ave, address - stack (rbp+24)

global stats2
stats2:
    push    rbp                             ; prologue
    mov     rbp, rsp
    push    r12

; -----
;  Get min and max.

    mov     eax, dword [rdi]                ; get min
    mov     dword [rdx], eax                ; return min

    mov     r12, rsi                        ; get len
    dec     r12                             ; set len-1
    mov     eax, dword [rdi+r12*4]          ; get max
    mov     dword [r9], eax                 ; return max

; -----
;  Get medians

    mov     rax, rsi
    mov     rdx, 0
    mov     r12, 2
    div     r12                             ; rax = length/2

    cmp     rdx, 0                          ; even/odd length?
    je      evenLength

    mov     r12d, dword [rdi+rax*4]         ; get arr[len/2]
    mov     dword [rcx], r12d               ; return med1
    mov     dword [r8], r12d                ; return med2
    jmp     medDone

evenLength:
    mov     r12d, dword [rdi+rax*4]         ; get arr[len/2]
    mov     dword [r8], r12d                ; return med2
    dec     rax
    mov     r12d, dword [rdi+rax*4]         ; get arr[len/2-1]
    mov     dword [rcx], r12d               ; return med1
medDone:

; -----
;  Find sum
```

```
    mov     r12, 0                          ; counter/index
    mov     rax, 0                          ; running sum

sumLoop:
    add     eax, dword [rdi+r12*4]          ; sum += arr[i]
    inc     r12
    cmp     r12, rsi
    jl      sumLoop

    mov     r12, qword [rbp+16]             ; get sum addr
    mov     dword [r12], eax                ; return sum

; -----
;  Calculate average.

    cdq
    idiv    rsi                             ; average = sum/len
    mov     r12, qword [rbp+24]             ; get ave addr
    mov     dword [r12], eax                ; return ave

    pop     r12                             ; epilogue
    pop     rbp
    ret
```

The choice of the registers is arbitrary with the bounds of the calling convention.

The call frame for this function would be as follows:



In this example, the preserved registers, **rbp** and then **r12**, are pushed. When popped, they must be popped in the exact reverse order **r12** and then **rbp** in order to correctly restore their original values.

12.11 Stack-Based Local Variables

If local variables are required, they are allocated on the stack. By adjusting the **rsp** register, additional memory is allocated on the stack for locals. As such, when the function is completed, the memory used for the stack-based local variables is released (and no longer uses memory).

Further expanding the previous example, if we assume all array values are between 0 and 99, and we wish to find the mode (number that occurs the most often), a single double-word variable ***count*** and a one hundred (100) element local double-word array, ***tmpArr[100]*** might be used.

As before, the frame register, **rbp**, is pushed on the stack and set pointing to itself. The frame register plus an appropriate offset will allow accessing any arguments passed on the stack. For example, **rbp+16** is the location of the first stack-based argument (7^{th} integer argument).

After the frame register is pushed, an adjustment to the stack pointer register, **rsp**, is made to allocate space for the local variables, a 100-element array in this example. Since the count variable is a one double-word, 4-bytes is needed. The temporary array is 100 double-word elements, 400 bytes is required. Thus, a total of 404 bytes is required. Since the stack is implemented growing downward in memory, the 404 bytes is subtracted from the stack pointer register.

Then any saved registers, **rbx** and **r12** in this example, are pushed on the stack.

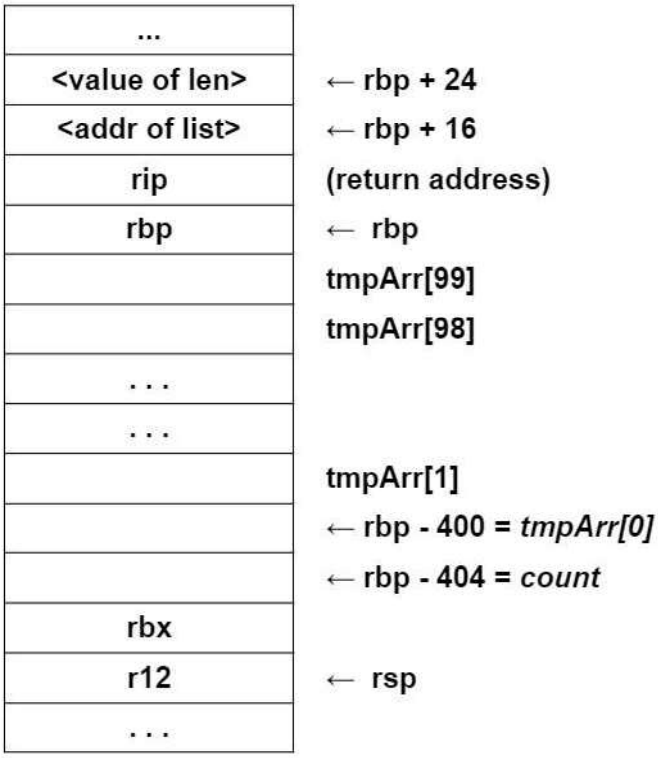
When leaving the function, the saved registers and then the locals must be cleared from the stack. The preferred method of doing this is to pop the saved registers and then top copy the **rbp** register into the **rsp** register, thus ensuring the **rsp** register points to the correct place on the stack.

```
mov     rsp, rbp
```

This is generally better than adding the offset back to the stack since allocated space may be altered as needed without also requiring adjustments to the epilogue code.

It should be clear that variables allocated in this manner are uninitialized. Should the function require the variables to be initialized, possibly to 0, such initializations must be explicitly performed.

For this example, the call frame would be formatted as follows:



The layout and order of the local variables within the allocated 404 bytes is arbitrary.

For example, the updated prologue code for this expanded example would be:

```
push    rbp                                 ; prologue
mov     rbp, rsp
sub     rsp, 404                            ; allocate locals
push    rbx
push    r12
```

The local variables can be accessed relative to the frame pointer register, **rbp**. For example, to initialize the count variable, now allocated to **rbp-404**, the following instruction could be used:

```
mov    dword [rbp-404], 0
```

To access the ***tmpArr***, the starting address must be obtained which can be performed with the lea instruction. For example,

```
lea    rbx, dword [rbp-400]
```

Which will set the appropriate stack address in the **rbx** register where **rbx** was chosen arbitrarily. The **dword** qualifier in this example is not required, and may be misleading, since addresses are always 64-bits (on a 64-bit architecture). Once set as above, the ***tmpArr*** starting address in **rbx** is used in the usual manner.

For example, a small incomplete function code fragment demonstrating the accessing of stack-based local variables is as follows:

```
; ----------------------------------------
;  Example function

global  expFunc
expFunc:
    push    rbp                             ; prologue
    mov     rbp, rsp
    sub     rsp, 404                        ; allocate locals
    push    rbx
    push    r12

; -----
;  Initialize count local variable to 0.

    mov     dword [rbp-404], 0

; -----
;  Increment count variable (for example)...

    inc     dword [rbp-404]                 ; count++

; -----
;  Loop to initialize tmpArr to all 0's.

    lea     rbx, dword [rbp-400]            ; tmpArr addr
    mov     r12, 0                          ; index
zeroLoop:
    mov     dword [rbx+r12*4], 0            ; tmpArr[index]=0
    inc     r12
    cmp     r12, 100
    jl      zeroLoop

; -----
;  Done, restore all and return to calling routine.

    pop     r12                             ; epilogue
```

```
pop     rbx
mov     rsp, rbp                        ; clear locals
pop     rbp
ret
```

Note, this example function focuses only on how stack-based local variables are accessed and does not perform anything useful.

12.12 Summary

This section presents a brief summary of the standard calling convention requirements which are as follows:

Caller Operations:

- The first six integer arguments are passed in registers
  - **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**
- The 7th and on arguments are passed on the stack-based
  - Pushes the arguments on the stack in reverse order (right to left, so that the first stack argument specified in the function call is pushed last).
  - Pushed arguments are passed as quadwords.
- The caller executes a **`call`** instruction to pass control to the function (callee).
- Stack-based arguments are cleared from the stack.
  - `add  rsp, <argCount*8>`

Callee Operations:

- Function Prologue
  - If arguments are passed on the stack, the callee must save **rbp** to the stack and move the value of **rsp** into **rbp**. This allows the callee to use **rbp** as a frame pointer to access arguments on the stack in a uniform manner.
    - The callee may then access its parameters relative to **rbp**. The quadword at **[rbp]** holds the previous value of **rbp** as it was pushed; the next quadword, at **[rbp+8]**, holds the return address, pushed by the **call**. The parameters start after that, at **[rbp+16]**.

- If local variables are needed, the callee decreases **rsp** further to allocate space on the stack for the local variables. The local variables are accessible at negative offsets from **rbp**.
- The callee, if it wishes to return a value to the caller, should leave the value in **al**, **ax**, **eax**, **rax**, depending on the size of the value being returned.
  - A floating-point result is returned in **xmm0**.
- If altered, registers **rbx**, **r12**, **r13**, **r14**, **r15** and **rbp** must be saved on the stack.

- Function Execution
  - The function code is executed.
- Function Epilogue
  - Restores any pushed registers.
  - If local variables were used, the callee restores **rsp** from **rbp** to clear the stack-based local variables.
  - The callee restores (i.e., pops) the previous value of **rbp**.
  - The call returns via `ret` instruction (return).

Refer to the sample functions to see specific examples of the calling convention.

12.13 Exercises

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

12.13.1 Quiz Questions

Below are some quiz questions based on this chapter.

1) What are the two main actions of a function call?
2) What are the two instructions that implement *linkage*?
3) When arguments are passed using *values*, it is referred to as?
4) When arguments are passed using *addresses*, it is referred to as?
5) If a function is called fifteen (15) times, how many times is the code placed in memory by the assembler?
6) What happens during the execution of a ***call*** instruction (two things)?
7) According to the standard calling convention, as discussed in class, what is the purpose of the initial pushes and final pops within most procedures?

8) If there are six (6) 64-bit integer arguments passed to a function, where specifically should each of the arguments be passed?

9) If there are six (6) 32-bit integer arguments passed to a function, where specifically should each of the arguments be passed?

10) What does it mean when a register is designated as temporary?

11) Name two temporary registers?

12) What is the name for the set of items placed on the stack as part of a function call?

13) What does it mean when a function is referred to as a *leaf function*?

14) What is the purpose of the `add rsp, <immediate>` after the call statement?

15) If ***three*** arguments are passed on the stack, what is the value for the `<immediate>`?

16) If there are seven (7) arguments passed to a function, and the function itself pushes the **rbp**, **rbx**, and **r12** registers (in that order), what is the correct offset of the stack-based argument when using the standard calling convention?

17) What, if any, is the limiting factor for how many times a function can be called?

18) If a function must return a result for the variable ***sum***, how should the ***sum*** variable be passed (call-by-reference or call-by-value)?

19) If there are eight (8) arguments passed to a function, and the function itself pushes the **rbp**, **rbx**, and **r12** registers (in that order), what are the correct offsets for each of the two stack-based arguments (7^{th} and 8^{th}) when using the standard calling convention?

20) What is the advantage of using stack dynamic local variables (as opposed to using all global variables)?

12.13.2 Suggested Projects

Below are some suggested projects based on this chapter.

1) Create a main and implement the ***stats1*** example function. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

2) Create a main and implement the ***stats2*** example function. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

3) Create a main program and a function that will sort a list of numbers in ascending order. Use the following selection[43] sort algorithm:

```
begin
    for  i = 0 to len-1
        small = arr(i)
        index = i
        for  j = i to len-1
            if ( arr(j) < small ) then
                small = arr(j)
                index = j
            end_if
        end_for
        arr(index) = arr(i)
        arr(i) = small
    end_for
end_begin
```

The main should call the function on at least three different data sets. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

4) Update the program from the previous question to add a stats function that finds the minimum, median, maximum, sum, and average for the sorted list. The stats function should be called after the sort function to make the minimum and maximum easier to find. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

5) Update the program from the previous question to add an integer square root function and a standard deviation function. To estimate the square root of a number, use the following algorithm:

$$iSqrt_{est} = iNumber$$

$$iSqrt_{est} = \frac{\left(\frac{iNumber}{iSqrt_{est}}\right) + iSqrt_{est}}{2} \qquad \text{iterate 50 times}$$

43 For more information, refer to: http://en.wikipedia.org/wiki/Selection_sort

The formula for standard deviation is as follows:

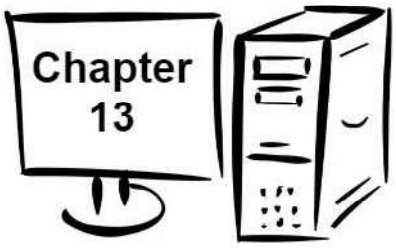
$$iStandardDeviation = \sqrt{\frac{\sum_{i=0}^{length-1} (list[i] - average)^2}{length}}$$

Note, perform the summation and division using integer values. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

6) Convert the integer to ASCII macro from the previous chapter into a void function. The function should convert a signed integer into a right-justified string of a given length. This will require including any leading blanks, a sign ("+" or "-"), the digits, and the NULL. The function should accept the value for the integer and the address of where to place the NULL terminated string, and the value of the maximum string length - in that order. Develop a main program to call the function on a series of different integers. The main should include the appropriate data declarations. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

7) Create a function to convert an ASCII string representing a number into an integer. The function should read the string and perform appropriate error checking. If there is an error, the function should return FALSE (a defined constant set to 0). If the string is valid, the function should convert the string into an integer. If the conversion is successful, the function should return TRUE (a defined constant set to 1). Develop a main program to call the function on a series of different integers. The main should include the appropriate data declarations and applicable the constants. Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

Linux is basically a simple operating system, but you have to be a genius to understand the simplicity.



13.0 System Services

There are many operations that an application program must use the operating system to perform. Such operations include console output, keyboard input, file services (open, read, write, close, etc.), obtaining the time or date, requesting memory allocation, and many others.

Accessing system services is how the application requests that the operating system perform some specific operation (on behalf of the process). More specifically, the *system call* is the interface between an executing process and the operating system.

This section provides an explanation of how to use some basic system service calls. More information on additional system service calls is located in Appendix C, System Service Calls.

13.1 Calling System Services

A system service call is logically similar to calling a function, where the function code is located within the operating system. The function may require privileges to operate which is why control must be transferred to the operating system.

When calling system services, arguments are placed in the standard argument registers. System services do not typically use stack-based arguments. This limits the arguments of a system services to six (6), which does not present a significant limitation.

To call a system service, the first step is to determine which system service is desired. There are many system services (see Appendix C). The general process is that the system service call code is placed in the **rax** register. The call code is a number that has been assigned for the specific system service being requested. These are assigned as part of the operating system and cannot be changed by application programs. To simplify the process, this text will define a very small subset of system service call codes to a set of constants. For this text, and the associated examples, the subset of

system call code constants are defined and shown in the source file to help provide complete clarity for new assembly language programmers. For more experienced programmers, typically developing larger or more complex programs, a complete list of constants is in a file and included into the source file.

If any are needed, the arguments for system services are placed in the **rdi**, **rsi**, **rdx**, **r10**, **r8**, and **r9** registers (in that order). The following table shows the argument locations which are consistent with the standard calling convention.

Register	Usage
`rax`	Call code (see table)
`rdi`	1st argument (if needed)
`rsi`	2nd argument (if needed)
`rdx`	3rd argument (if needed)
`r10`	4th argument (if needed)
`r8`	5th argument (if needed)
`r9`	6th argument (if needed)

This is very similar to the standard calling convention for function calls, however the 4th argument, if needed, uses the **r10** register.

Each system call will use a different number of arguments (from none up to 6). However, the system service call code is always required.

After the call code and any arguments are set, the **syscall** instruction is executed. The **syscall** instruction will pause the current process and transfer control to the operating system which will attempt to perform the service specified in the **rax** register. When the system service returns, the process will be resumed.

13.2 Newline Character

As a refresher, in the context of output, a newline means move the cursor to the start of the next line. In many languages, including C, it is often noted as "\n" as part of a string. C++ uses **endl** in the context of a **cout** statement. For example, "Hello World 1" and "Hello\nWorld 2" would be displayed as follows:

```
Hello World 1
Hello
World 2
```

Nothing is displayed for the newline, but the cursor is moved to the start of the next line as shown.

In Unix/Linux systems, the linefeed, abbreviated LF with an ASCII value of 10 (or 0x0A), is used as the newline character. In Windows systems, the newline is carriage return, abbreviated as CR with an ASCII value 13 (or 0x0D) followed by the LF. The LF is used in the code examples in the text.

The reader may have seen instances where a text file is downloaded from a web page and displayed using older versions Windows Notepad (pre-Windows 10) where all the formatting is lost and it looks like the text is one very long line. This is typically due to a Unix/Linux formatted file, which uses only LF's, being displayed with a Windows utility that expects CR/LF pairs and does not display correctly when only LF's are found. Other Windows software, like Notepad++ (open source text editor) will recognize and handle the different newline formats and display correctly.

13.3 Console Output

The system service to output characters to the console is the system write (SYS_write). Like a high-level language characters are written to standard out (STDOUT) which is the console. The STDOUT is the default file descriptor for the console. The file descriptor is already opened and available for use in programs (assembly and high-level languages).

The arguments for the write system service are as follows:

Register	SYS_write
rax	Call code = SYS_write (1)
rdi	Output location, STDOUT (1)
rsi	Address of characters to output
rdx	Number of characters to output

Assuming the following declarations:

```
STDOUT      equ     1               ; standard output
SYS_write   equ     1               ; call code for write

msg         db      "Hello World"
msgLen      dq      11
```

For example, to output “Hello World” (it’s traditional) to the console, the system write (SYS_write) would be used. The code would be as follows:

```
    mov     rax, SYS_write
    mov     rdi, STDOUT
    mov     rsi, msg                    ; msg address
    mov     rdx, qword [msgLen]         ; length value
    syscall
```

Refer to the next section for a complete program to display the above message. It should be noted that the operating system does not check if the string is valid.

13.3.1 Example, Console Output

This example is a complete program to output some strings to the console. In this example, one string includes new line and the other does not.

```
; Example program to demonstrate console output.
; This example will send some messages to the screen.

; ------------------------------------------

section    .data

; -----
;  Define standard constants.

LF              equ     10              ; line feed
NULL            equ     0               ; end of string
TRUE            equ     1
FALSE           equ     0

EXIT_SUCCESS    equ     0               ; success code

STDIN           equ     0               ; standard input
STDOUT          equ     1               ; standard output
STDERR          equ     2               ; standard error

SYS_read        equ     0               ; read
SYS_write       equ     1               ; write
SYS_open        equ     2               ; file open
SYS_close       equ     3               ; file close
SYS_fork        equ     57              ; fork
SYS_exit        equ     60              ; terminate
SYS_creat       equ     85              ; file open/create
```

```
SYS_time        equ     201             ; get time

; -----
;  Define some strings.

message1        db      "Hello World.", LF, NULL
message2        db      "Enter Answer: ", NULL
newLine         db      LF, NULL

;-----------------------------------------------------

section     .text
global _start
_start:

; -----
;  Display first message.

    mov     rdi, message1
    call    printString

; -----
;  Display second message and then newline

    mov     rdi, message2
    call    printString

    mov     rdi, newLine
    call    printString

; -----
;  Example program done.

exampleDone:
    mov     rax, SYS_exit
    mov     rdi, EXIT_SUCCESS
    syscall

; ******************************************************
;  Generic function to display a string to the screen.
;  String must be NULL terminated.
;  Algorithm:
;   Count characters in string (excluding NULL)
;   Use syscall to output characters
```

```
;  Arguments:
;   1) address, string
;  Returns:
;   nothing

global printString
printString:
    push    rbx

; -----
;  Count characters in string.

    mov     rbx, rdi
    mov     rdx, 0
strCountLoop:
    cmp     byte [rbx], NULL
    je      strCountDone
    inc     rdx
    inc     rbx
    jmp     strCountLoop
strCountDone:

    cmp     rdx, 0
    je      prtDone

; -----
;  Call OS to output string.

    mov     rax, SYS_write          ; system code for write()
    mov     rsi, rdi                ; address of chars to write
    mov     rdi, STDOUT             ; standard out
                                    ; RDX=count to write, set above
    syscall                         ; system call

; -----
;  String printed, return to calling routine.

prtDone:
    pop     rbx
    ret
```

The output would be as follows:

```
Hello World.
Enter Answer:_
```

The newline (LF) was provided as part of the first string (***message1***) thus placing the cursor on the start of the next line. The second message would leave the cursor on the same line which would be appropriate for reading input from the user (which is not part of this example). A final newline is printed since no actual input is obtained in this example.

The additional, unused constants are included for reference.

13.4 Console Input

The system service to read characters from the console is the system read (SYS_read). Like a high-level language, for the console, characters are read from standard input (STDIN). The STDIN is the default file descriptor for reading characters from the keyboard. The file descriptor is already opened and available for use in program (assembly and high-level languages).

Reading characters interactively from the keyboard presents an additional complication. When using the system service to read from the keyboard, much like the write system service, the number of characters to read is required. Of course, we will need to declare an appropriate amount of space to store the characters being read. If we request 10 characters to read and the user types more than 10, the additional characters will be lost, which is not a significant problem. If the user types less than 10 characters, for example 5 characters, all five characters will be read plus the newline (LF) for a total of six characters.

A problem arises if input is redirected from a file. If we request 10 characters, and there are 5 characters on the first line and more on the second line, we will get the six characters from the first line (5 characters plus the newline) and the first four characters from the next line for the total of 10. This is undesirable.

To address this, for interactively reading input, we will read one character at a time until a LF (the Enter key) is read. Each character will be read and then stored, one at a time, in an appropriately sized array.

The arguments for the read system service are as follows:

Register	SYS_read
rax	Call code = SYS_read (0)
rdi	Input location, STDIN (0)

rsi	Address of where to store characters read
rdx	Number of characters to read

Assuming the following declarations:

```
STDIN       equ     0               ; standard input
SYS_read    equ     0               ; call code for read

inChar      db      0
```

For example, to read a single character from the keyboard, the system read (SYS_read) would be used. The code would be as follows:

```
mov     rax, SYS_read
mov     rdi, STDIN
mov     rsi, inChar                     ; msg address
mov     rdx, 1                          ; read count
syscall
```

Refer to the next section for a complete program to read characters from the keyboard.

13.4.1 Example, Console Input

The example is a complete program to read a line of 50 characters from the keyboard. Since space for the newline (LF) along with a final NULL termination is included, an input array allowing 52 bytes would be required.

This example will read up to 50 characters from the user and then echo the input back to the console to verify that the input was read correctly.

```
; Example program to demonstrate console output.
; This example will send some messages to the screen.
; ***********************************************

section     .data

; -----
;  Define standard constants.

LF              equ     10              ; line feed
NULL            equ     0               ; end of string
```

```
TRUE            equ     1
FALSE           equ     0

EXIT_SUCCESS    equ     0               ; success code

STDIN           equ     0               ; standard input
STDOUT          equ     1               ; standard output
STDERR          equ     2               ; standard error

SYS_read        equ     0               ; read
SYS_write       equ     1               ; write
SYS_open        equ     2               ; file open
SYS_close       equ     3               ; file close
SYS_fork        equ     57              ; fork
SYS_exit        equ     60              ; terminate
SYS_creat       equ     85              ; file open/create
SYS_time        equ     201             ; get time

; -----
;  Define some strings.

STRLEN          equ     50

pmpt            db      "Enter Text: ", NULL
newLine         db      LF, NULL

section   .bss
chr             resb    1
inLine          resb    STRLEN+2            ; total of 52

;-----------------------------------------------------

section     .text
global _start
_start:

; -----
;  Display prompt.

    mov     rdi, pmpt
    call    printString

; -----
;  Read characters from user (one at a time)
```

```
    mov     rbx, inLine             ; inLine addr
    mov     r12, 0                  ; char count
readCharacters:
    mov     rax, SYS_read           ; system code for read
    mov     rdi, STDIN              ; standard in
    lea     rsi, byte [chr]         ; address of chr
    mov     rdx, 1                  ; count (how many to read)
    syscall                         ; do syscall

    mov     al, byte [chr]          ; get character just read
    cmp     al, LF                  ; if linefeed, input done
    je      readDone

    inc     r12                     ; count++
    cmp     r12, STRLEN             ; if # chars ≥ STRLEN
    jae     readCharacters          ;   stop placing in buffer

    mov     byte [rbx], al          ; inLine[i] = chr
    inc     rbx                     ; update tmpStr addr

    jmp     readCharacters
readDone:
    mov     byte [rbx], NULL        ; add NULL termination

; -----
;  Output the line to verify successful read

    mov     rdi, inLine
    call    printString

; -----
;  Example done.

exampleDone:
    mov     rax, SYS_exit
    mov     rdi, EXIT_SUCCESS
    syscall

; ******************************************************
;  Generic procedure to display a string to the screen.
;  String must be NULL terminated.
;  Algorithm:
;     Count characters in string (excluding NULL)
```

```
;     Use syscall to output characters

;  Arguments:
;   1) address, string
;  Returns:
;   nothing

global printString
printString:
    push    rbx

; -----
;  Count characters in string.

    mov     rbx, rdi
    mov     rdx, 0

strCountLoop:
    cmp     byte [rbx], NULL
    je      strCountDone
    inc     rdx
    inc     rbx
    jmp     strCountLoop
strCountDone:

    cmp     rdx, 0
    je      prtDone

; -----
;  Call OS to output string.

    mov     rax, SYS_write    ; system code for write()
    mov     rsi, rdi            ; address of char's to write
    mov     rdi, STDOUT         ; standard out
                                ; RDX=count to write, set above
    syscall                   ; system call

; -----
;  String printed, return to calling routine.

prtDone:
    pop     rbx
    ret
```

If we were to completely stop reading at 50 (STRLEN) characters and the user enters more characters, the characters might cause input errors for successive read operations. To address any extra characters the user might enter, the extra characters are read from the keyboard but not placed in the input buffer (***inLine*** above). This ensures that the extra input is removed from the input steam and but does not overrun the array.

The additional, unused constants are included for reference.

13.5 File Open Operations

In order to perform file operations such as read and write, the file must first be opened. There are two file open operations, open and open/create. Each of the two open operations are explained in the following sections.

After the file is opened, in order to perform file read or write operations the operating system needs detailed information about the file, including the complete status and current read/write location. This is necessary to ensure that read or write operations pick up where they left off (from last time).

If the file open operation fails, an error code will be returned. If the file open operation succeeds, a file descriptor is returned. This applies to both high-level languages and assembly code.

The file descriptor is used by the operating system to access the complete information about the file. The complete set of information about an open file is stored in an operating system data structure named File Control Block (FCB). In essence, the file descriptor is used by the operating system to reference the correct FCB. It is the programmer's responsibility to ensure that the file descriptor is stored and used correctly.

13.5.1 File Open

The file open requires that the file exists in order to be opened. If the file does not exist, it is an error.

The file open operation also requires the parameter flag to specify the access mode. The access mode must include one of the following:

- Read-Only Access → O_RDONLY
- Write-Only Access → O_WRONLY
- Read/Write Access → O_RDWR

One of these access modes must be used. Additional access modes may be used by OR'ing with one of these. This might include modes such as append mode (which is not addressed in this text). Refer to Appendix C, System Services for additional information regarding the file access modes.

The arguments for the file open system service are as follows:

Register	SYS_open
rax	Call code = SYS_open (2)
rdi	Address of NULL terminated file name string
rsi	File access mode flag

Assuming the following declarations:

```
SYS_open        equ     2           ; file open

O_RDONLY        equ     000000q     ; read only
O_WRONLY        equ     000001q     ; write only
O_RDWR          equ     000002q     ; read and write
```

After the system call, the **rax** register will contain the return value. If the file open operation fails, **rax** will contain a negative value (i.e., < 0). The specific negative value provides an indication of the type of error encountered. Refer to Appendix C, System Services for additional information on error codes. Typical errors might include invalid file descriptor, file not found, or file permissions error.

If the file open operation succeeds, **rax** contains the file descriptor. The file descriptor will be required for further file operations and should be saved.

Refer to the section on Example File Read for a complete example that opens a file.

13.5.2 File Open/Create

A file open/create operation will create a file. If the file does not exist, a new file will be created. If the file already exists, it will be erased and a new file created. Thus, the previous contents of the file will be lost.

A file access mode must be specified. Since the file is being created, the access mode must include the file permissions that will be set when the file is created. This would include specifying read, write, and/or execute permissions for the *user*, *group*, or *world* as is typical for Linux file permissions. The only permissions addressed in this example are for the user or owner of the file. As such, other users (i.e., using other accounts) will not be able to access the file our program creates. Refer to Appendix C, System