Instruction Set

The following subset of X86-64 instructions are to be discussed :

- Data movement
- Conversion Instruction
- Arithmetic Instruction
- Logical Instruction
- Control Instruction

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).

Operand Notation

`<src>`	Source operand. Operand value is unchanged after the instruction.
`<imm>`	Immediate value. May be specified in decimal, hex, octal, or binary.
`<mem>`	Memory location. May be a variable name or an indirect reference (i.e., a memory address).
`<op>` or `<operand>`	Operand, register or memory.
`<op8>`, `<op16>`, `<op32>`, `<op64>`	Operand, register or memory, with specific size requirement. For example, **op8** means a byte sized operand only and **reg32** means a double-word sized operand only.
`<label>`	Program label.

Data Movement

MOV (Move) is instructions are used to move data from
- Memory to register
- Register to memory
- Register to register

General format is

mov <dest>, <src>

Example:

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

what to do — *where to place* — *how much to get* — *memory location*

Size of data from source must be equal to size of destination

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

Data Movement

To access data in memory, brackets [ ] must be used. Omitting brackets means refering to memory address

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

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

Data Movement

Practice Session

Run the following program in debugging mode using DDD. Notice registers value carefully

```
global  _start

section .data
bVar    db      0
wVar    dw      0
qVar    dq      0

section .text
_start:
        mov     al,0    ;al =
        mov     rbx,0   ;rbx =
        mov     rbx,-1  ;rbx =
        mov     rbx,-2  ;rbx =
        mov     bl,byte [bVar]  ;rbx =
        mov     bx,word [wVar]  ;rbx =
        mov     rbx, 0x1234567812345678 ;rbx =
        mov     [qVar], rbx     ;qVar =

        mov     rax,60  ;exit to console
        mov     rdi,0
        syscall
```

yasm -f elf64 -g dwarf2 -o datamovement.o datamovement.s
ld -o datamovement datamovement.o
ddd ./datamovement &

Data Conversion

Unsigned conversion

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>

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`

Data Conversion

Signed conversion

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

movsx <dest>, <src>

`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]`

Assignment #1

Fibinacci

Create a program to iteratively find the ***n****th* Fibonacci number. 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.

Example

Practice session: unsigned conversion

```
File  Edit  View  Search  Terminal  Help
global _start

section .data
bVar1   db      0x12
bVar2   db      0xaa
wVar1   dw      0x0000
wVar2   dw      0xffee

section .text
_start:
        movzx   cx, byte [bVar1]
        movzx   cx, byte [bVar2]
        movzx   ebx, word [wVar1]
        movzx   ebx, word [wVar2]
        movzx   ebx, cx
        movzx   rbx, cl
        movzx   rbx, cx

        mov     rax, 60 ;exit
        mov     rdi, 0
        syscall
                          1,1           All
```

Example

Practice session: signed conversion

File Edit View Search Terminal Help

```
global _start

section .data
bVar1   db      0x12
bVar2   db      0xaa
wVar1   dw      0x0000
wVar2   dw      0xffee

section .text
_start:
        movsx   cx, byte [bVar1]
        movsx   cx, byte [bVar2]
        movsx   ebx, word [wVar1]
        movsx   ebx, word [wVar2]
        movsx   ebx, cx
        movsx   rbx, cl
        movsx   rbx, cx

        mov     rax, 60 ;exit
        mov     rdi, 0
        syscall
                            1,1           All
```

Arithmetic Instructions

Addition:

add <dest>, <src>	**; dest = dest + src**
inc <dest>	**; dest = dest + 1**

Subtraction:

sub <dest>, <src>	**;dest = dest – src**
dec <dest>	**; dest = dest - 1**

Practice Session

Practice session: addition

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyL...

File Edit View Search Terminal Help

```
global _start

section .data
        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

section .text
_start:
        ;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

        inc al
        inc ax
        inc eax
        inc byte [bAns]
        inc word [wAns]
        inc dword [dAns]

        mov rax, 60     ;exit
        mov rdi, 0
        syscall
(END)
```

Practice Session

Practice session: subtraction

watis@ThinkPad-E570 ~/Desktop/EN81...

File Edit View Search Terminal Help

```
global _start

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

section .text
_start:
        ;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

        dec al
        dec ax
        dec eax
        dec byte [bAns]
        dec word [wAns]
        dec dword [dAns]

        mov rax, 60     ;exit
        mov rdi, 0
        syscall
(END)
```

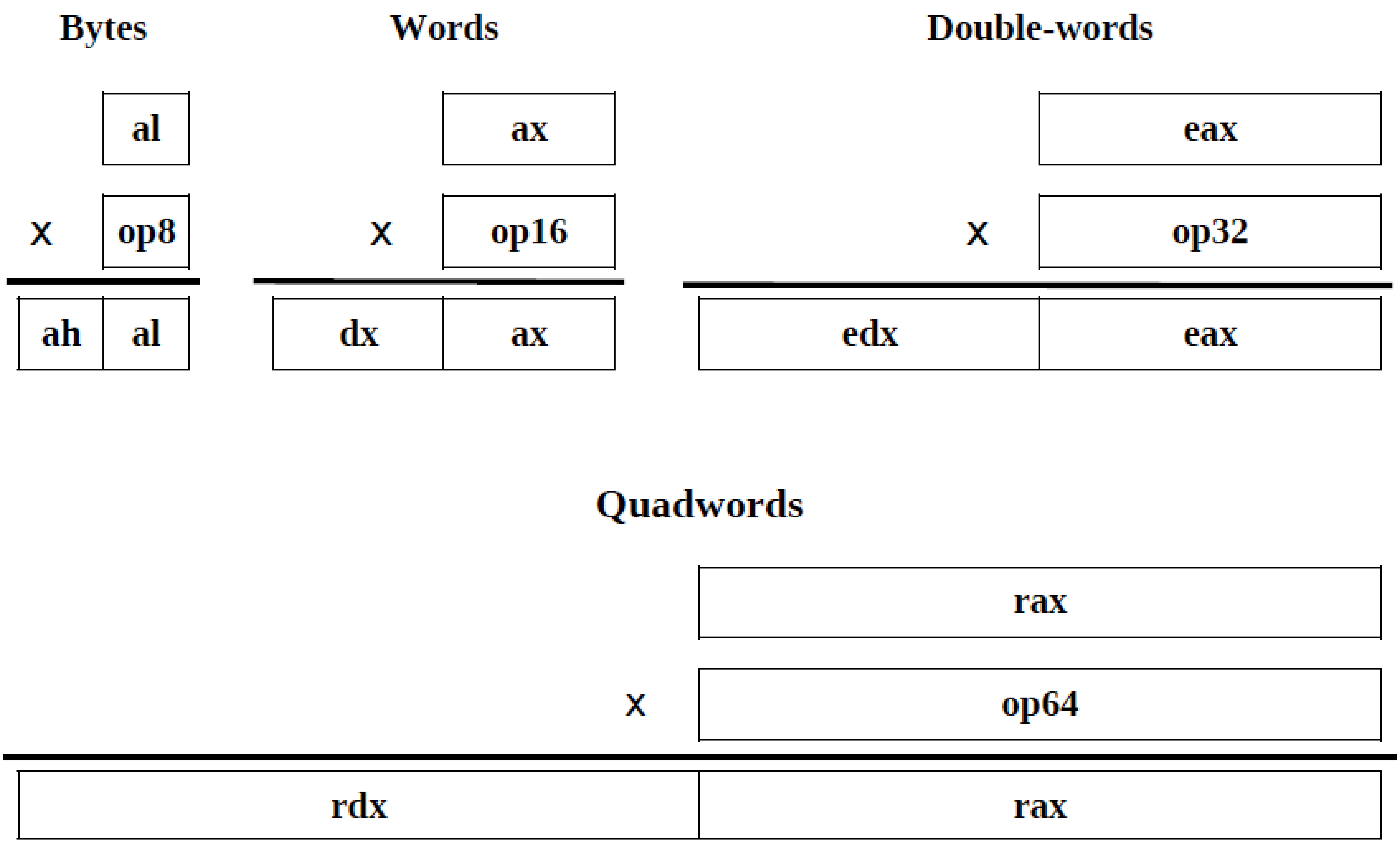
Arithmetic Instructions

Multiplication

Multiplication typically produces double sized results. That is, multiplying two n-bit values produces a 2n-bit result.

Unsigned multiplication

mul <src> ; result = reg. A x <src>



Arithmetic Instructions

Addition

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]**

Arithmetic Instructions

Unsigned multiplication

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyLanguageP...

File Edit View Search Terminal Help

```
global  _start

section .data
        bNumA   db 15
        bNumB   db 12
        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

section .text
_start:
        ; 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

        mov rax, 60     ;exit
        mov rdi, 0
        syscall

(END)
```

Arithmetic Instructions

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>

Arithmetic Instructions

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]

Arithmetic Instructions

Signed multiplication

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyLangu...

File Edit View Search Terminal Help

```
global  _start
section .data
        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

section .text
_start:
        ; 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

        mov     rax, 60         ;exit
        mov     rdi, 0
        syscall
```

Arithmetic Instructions

Integer division

$$\frac{\textit{dividend}}{\textit{divisor}} = \textit{quotient}$$

Division requires that the dividend must be a larger size than the divisor.

Bytes

$$\frac{\boxed{\textbf{ah}}\boxed{\textbf{al}}}{\boxed{\textbf{op8}}} = \boxed{\textbf{al}}$$

$$\textbf{rem} \quad \boxed{\textbf{ah}}$$

Words

$$\frac{\boxed{\textbf{dx}}\boxed{\textbf{ax}}}{\boxed{\textbf{op16}}} = \boxed{\textbf{ax}}$$

$$\textbf{rem} \quad \boxed{\textbf{dx}}$$

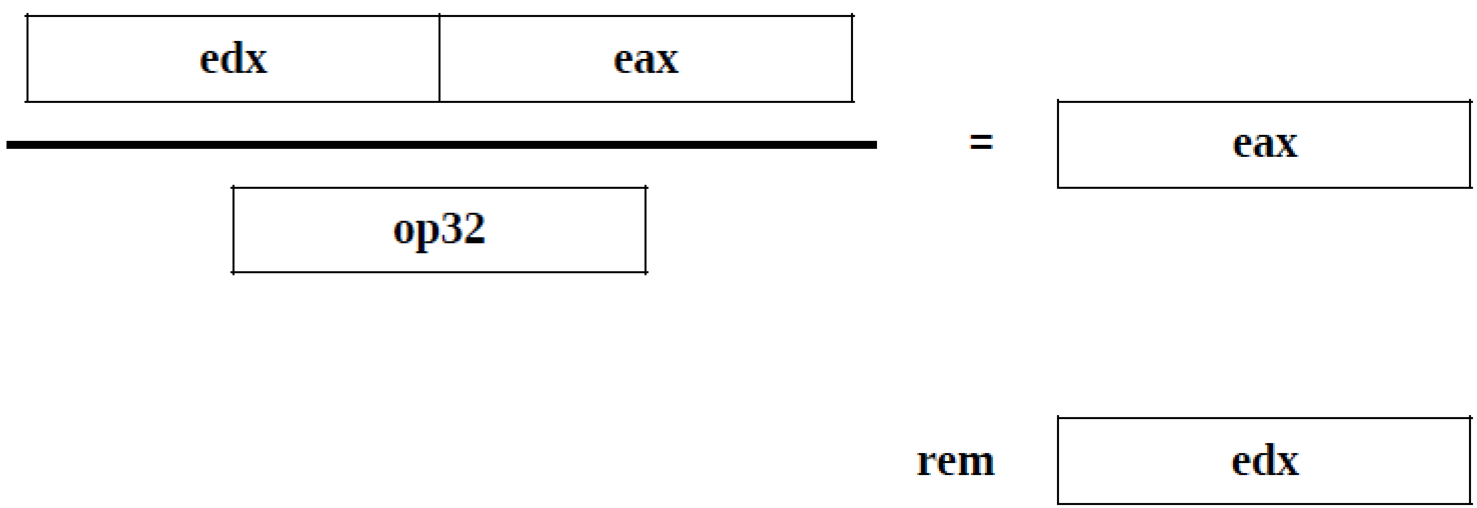
Arithmetic Instructions

Integer division

$$\frac{dividend}{divisor} = quotient$$

Division requires that the dividend must be a larger size than the divisor.

Double-words



Arithmetic Instructions

Integer division

$$\frac{\textit{dividend}}{\textit{divisor}} = \textit{quotient}$$

Division requires that the dividend must be a larger size than the divisor.

Quadwords

$$\frac{\boxed{\texttt{rdx} \mid \texttt{rax}}}{\boxed{\texttt{op64}}}$$

= rax

rem rdx

Arithmetic Instructions

Integer division

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

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

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]`

Arithmetic Instructions

Integer division

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

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

`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]`

Arithmetic Instructions

Integer division

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyLanguagePr...

File Edit View Search Terminal Help

```
global  _start
section .data
        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

section .text
_start:
; 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)
                                  1,1          Top
```

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyLanguagePr...

File Edit View Search Terminal Help

```
        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 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 edx:eax
        idiv dword [dNumB] ; eax = edx:eax/dNumB
        mov dword [dAns2], eax
        mov dword [dRem2], edx ; edx = edx:eax%dNu
mB
; 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]
                                  93,1-8        71%
```

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyLangua...

File Edit View Search Terminal Help

```
        cqo ; rax → edx:eax 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 → edx:eax rdx:rax
        idiv qword [qNumB] ; rax = rdx:rax/qNum
B
        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/qNum
B
        mov qword [qAns3], rax

        mov rax, 60     ;EXIT
        mov rdi, 0
        syscall
                                  112,1-8       Bot
```

Logical Instructions

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`

Logical Instructions

`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`

Logical Instructions

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`

Logical Instructions

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

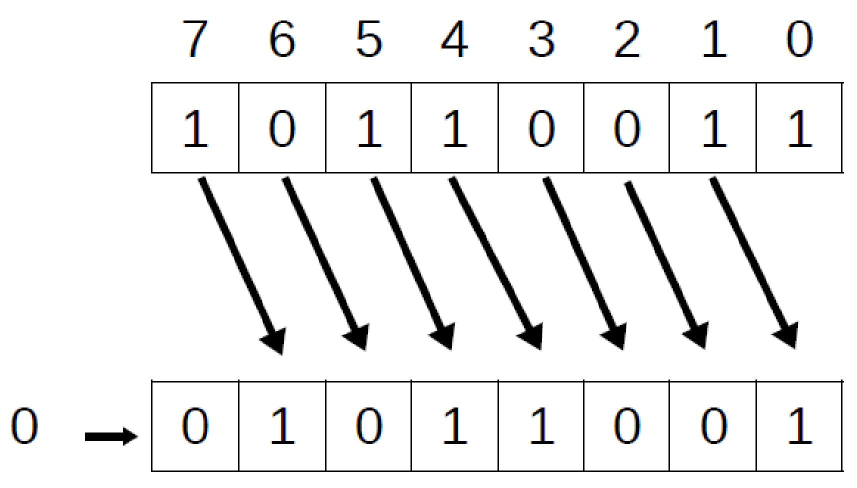
Examples:	`not    bx`
	`not    rdx`
	`not    dword [dNum]`
	`not    qword [qNum]`

Logical Instructions

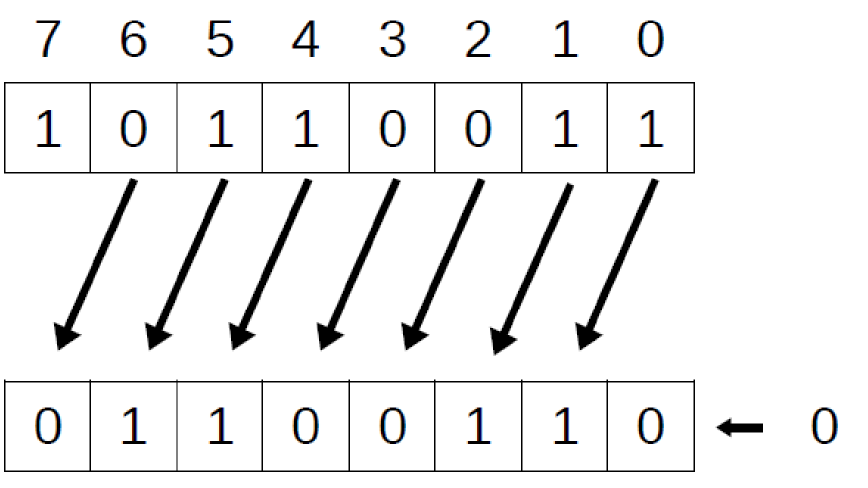
Shift instructions

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

Shift Right Logical



Shift Left Logical



Logical Instructions

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).

Shift Right Logical
Unsigned Division

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

Shift Left Logical
Unsigned Multiplication

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

Logical Instructions

Instruction	Explanation
`shl <dest>, <imm>` `shl <dest>, cl`	Perform logical shift left 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:	`shl ax, 8`
	`shl rcx, 32`
	`shl eax, cl`
	`shl qword [qNum], cl`

Logical Instructions

`shr <dest>, <imm>` `shr <dest>, cl`	Perform logical shift right 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.

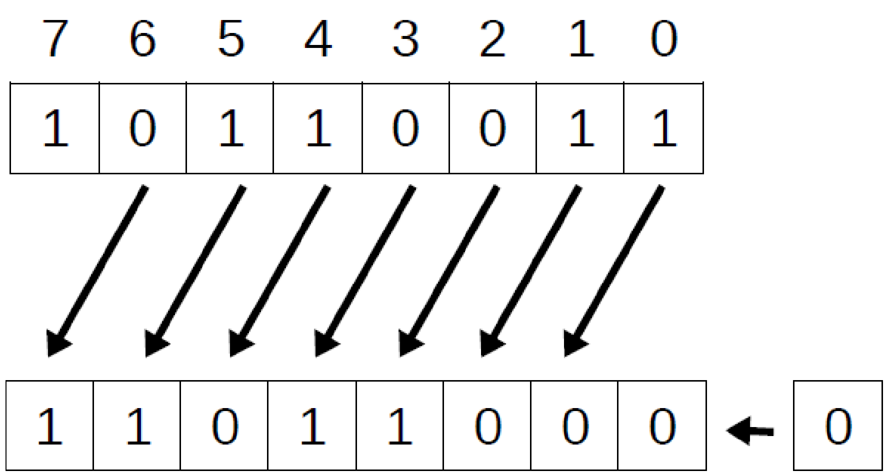
Examples:	`shr ax, 8`
	`shr rcx, 32`
	`shr eax, cl`
	`shr qword [qNum], cl`

Logical Instructions

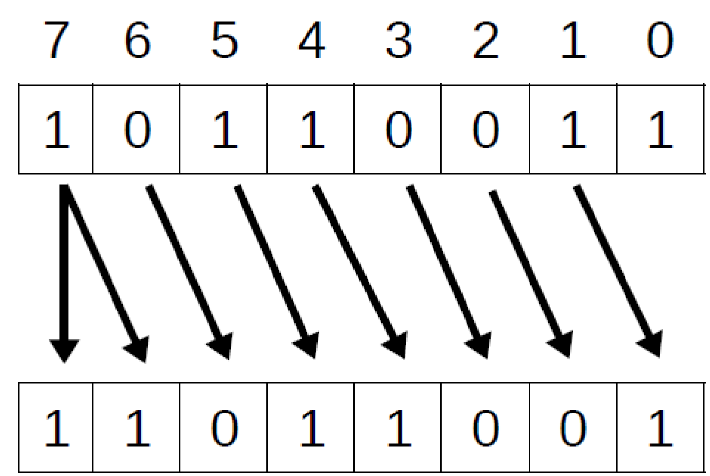
Arithmetic Shift

The original leftmost bit (the sign bit) is replicated to fill in all the vacant positions when performing arithmetic shift

Shift Left Arithmetic



Shift Right Arithmetic



Logical Instructions

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`

Logical Instructions

`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`

Logical Instructions

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.

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`

Logical Instructions

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

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

Control Instructions

Unconditional Control Instructions

The unconditional instruction provides an unconditional jump to a specific location in the program denoted with a program label.

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

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

Control Instructions

Conditional Control Instructions

The conditional control instructions provide a conditional jump based on a comparison.
This provides the functionality of a basic IF statement. The compare instruction will compare two operands and store the results of the comparison in the **rFlag** registers. The general form of the compare instruction is:

cmp <op1>, <op2>

This requires that the compare instruction is immediately followed by the conditional jump instruction.

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]`

Control Instructions

Jump if Equal	**je** <label>	; if <op1> == <op2>
Jump if Not Equal	**jne** <label>	; if <op1> != <op2>
Jump if Less than	**jl** <label>	; **signed**, if <op1> < <op2>
Jump if Less than or Equal	**jle** <label>	; **signed**, if <op1> <= <op2>
Jump if Greater than	**jg** <label>	; **signed**, if <op1> > <op2>
Jump if Greater than or Equal	**jge** <label>	; **signed**; if <op1> >= <op2>
Jump if Below	**jb** <label>	; unsigned, if <op1> < <op2>
Jump if Below or Equal	**jbe** <label>	; unsigned, if <op1> <= <op2>
Jump if Above	**ja** <label>	; unsigned, if <op1> > <op2>
Jump if Above or Equal	**jae** <label>	; unsigned, if <op1> >= <op2>

Control Instructions

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

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

```
section .data
    currNum     dq 0
    myMax       dq 0

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

Control Instructions

A more complex example might be as follows:

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

```
section .data
    TRUE     equ 1
    FALSE    equ 0
    x        dd 0
    y        dd 0
    ans      dd 0
    errFlg   db FALSE
```

```
section .text
    cmp dword [x], 0        ; if statement
    je doElse
    mov eax, dword [x]
    cdq                     ; double word to quad word
    idiv dword [y]
    mov dword [ans], eax
    mov byte [errFlg], FALSE
    jmp skpElse
doElse:
    mov dword [ans], 0
    mov byte [errFlg], TRUE
skpElse:
```

Control Instructions

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".

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

Replace *jne* with *jmp* if out-of-range error occurs

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

Control Instructions

Iteration

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

```
Section .data
        lpCnt   dq 15
        sum     dq 0
Section .text
        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
```

Control Instructions

Iteration

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.

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

<u>Code Set 1</u>

```
loop  <label>
```

<u>Code Set 2</u>

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

Control Instructions

Iteration

```
            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
```

Control Instructions

Example: Sum of square

watis@ThinkPad-E570 ~/Desktop/EN812700AssemblyLanguageProgrammin...

File Edit View Search Terminal Help

```
; sum of squares from 1 to n.
section .data
        SUCCESS         equ 0 ; Successful operation
        SYS_exit        equ 60 ; call code for terminate
        n               dd 10
        sumOfSquares dq 0
section .text
global _start
_start:
; -----
; Compute sum of squares from 1 to n.
; 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
last:
        mov rax, SYS_exit       ; call code for exit
        mov rdi, SUCCESS
        syscall
```

Assignment #1

Fibinacci

Create a program to iteratively find the ***n**th* Fibonacci number. 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.