https://cs.lmu.edu/~ray/notes/nasmtutorial/
https://www.rapidtables.com/convert/number/decimal-to-binary.html
http://www.penguin.cz/~literakl/intel/intel.html
https://faydoc.tripod.com/cpu/

CPU

Control unit is a decoder

Data Representation

Two's complement

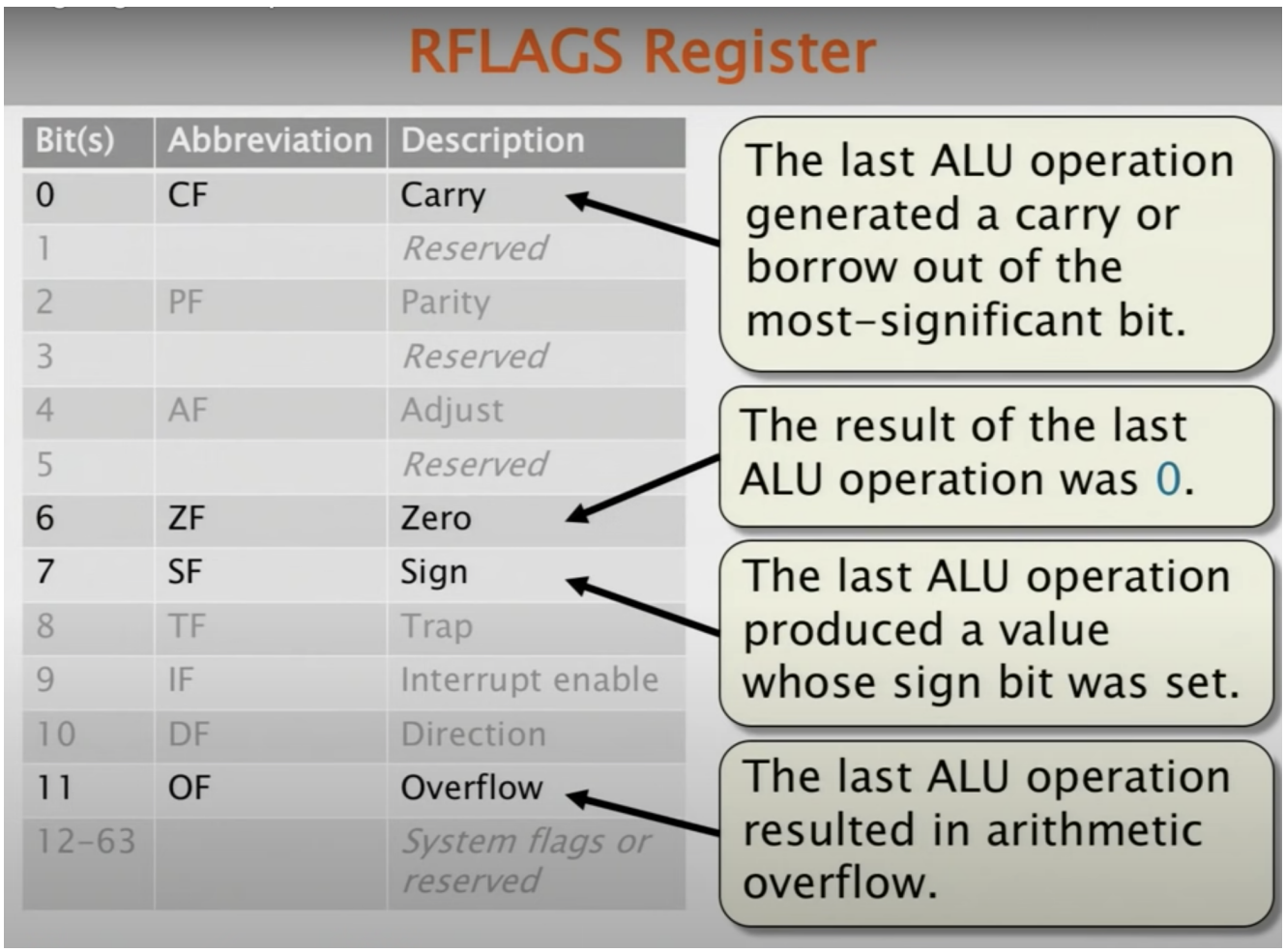
To take the two's complement of a number:

1. take the one's complement (negate)
2. add 1 (in binary)

Convert to floating-point representation

1. Conver to binary: e.g. 11.25 → 1101.01
2. convert to a normalized form $1101.1 * 2^1 -> 1 + 127 = 128$ biased exponent

Flags



Registers

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Programming

Program structure

```
; ------------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ------------------------------------------------------------------------------------------

            global    _start
```

```
          section   .text
_start:   mov       rax, 1                ; system call for write
          mov       rdi, 1                ; file handle 1 is stdout
          mov       rsi, message          ; address of string to output
          mov       rdx, 13               ; number of bytes
          syscall                         ; invoke operating system to do the write
          mov       rax, 60               ; system call for exit
          xor       rdi, rdi              ; exit code 0
          syscall                         ; invoke operating system to exit

          section   .data
message:  db        "Hello, World", 10    ; note the newline at the end
```

BSS section

the block starting symbol (abbreviated to . bss or bss) is **the portion of an object file, executable, or assembly language code that contains statically allocated variables that are declared but have not been assigned a value yet**. It is often referred to as the "bss section" or "bss segment".

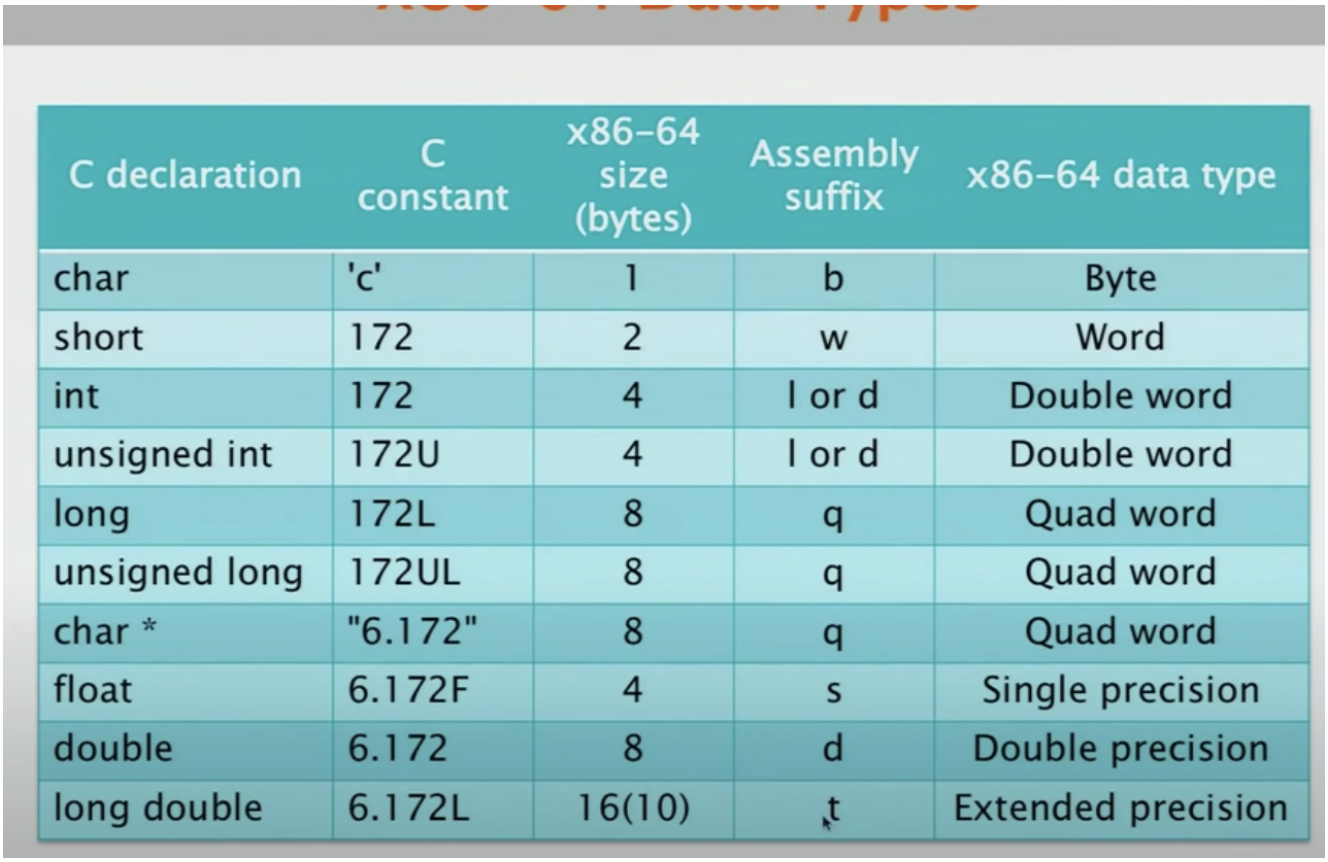
```
section .bss
        num: resb 4
```

Instructions

Type of operation		Examples
Data movement	Move	mov
	Conditional move	cmov
	Sign or zero extension	movs, movz
	Stack	push, pop
Arithmetic and logic	Integer arithmetic	add, sub, mul, imul, div, idiv, lea, sal, sar, shl, shr, rol, ror, inc, dec, neg
	Binary logic	and, or, xor, not
	Boolean logic	test, cmp
Control transfer	Unconditional jump	jmp
	Conditional jumps	j<condition>
	Subroutines	call, ret

Variables

data types



C declaration	C constant	x86-64 size (bytes)	Assembly suffix	x86-64 data type
char	'c'	1	b	Byte
short	172	2	w	Word
int	172	4	l or d	Double word
unsigned int	172U	4	l or d	Double word
long	172L	8	q	Quad word
unsigned long	172UL	8	q	Quad word
char *	"6.172"	8	q	Quad word
float	6.172F	4	s	Single precision
double	6.172	8	d	Double precision
long double	6.172L	16(10)	t	Extended precision

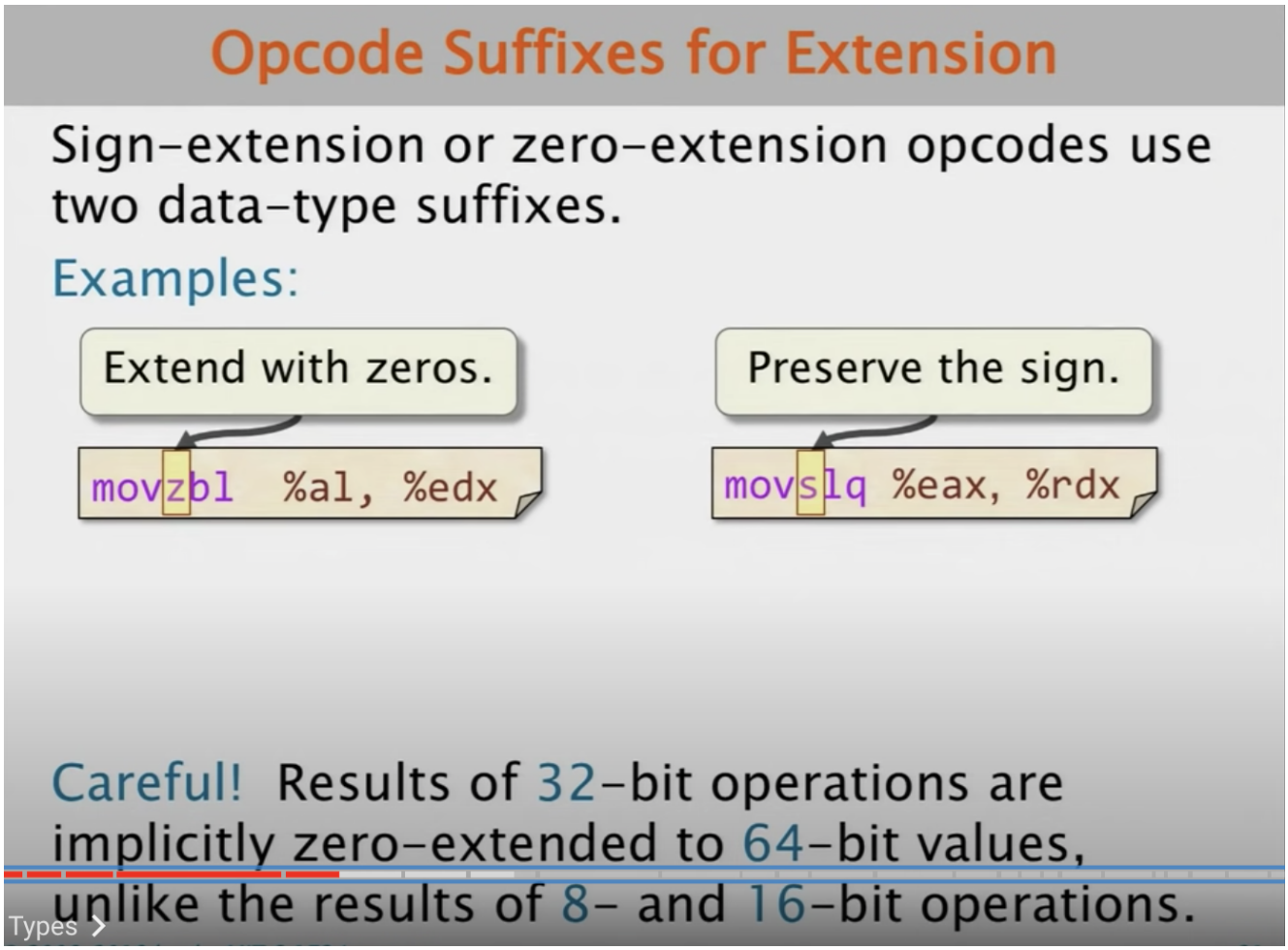
Define Data

```
bVar db 10 ; byte variable
cVar db "H" ; single character
strng db "Hello World" ; string
wVar dw 5000 ; 16-bit variable
dVar dd 50000 ; 32-bit variable
arr dd 100, 200, 300 ; 3 element array
flt1 dd 3.14159 ; 32-bit float
qVar dq 1000000000 ; 64-bit variable
```

Define Data

```
bArr resb 10 ; 10 element byte array
wArr resw 50 ; 50 element word array
dArr resd 100 ; 100 element double arra
qArr resq 200 ; 200 element quad array
```

Register Extension



Narrowing

Move from larger register/variable to smaller one. Have to make sure conversion is right.

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

Widening

unsigned

Method 1

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

Method 2 using `movzx` not available for 32-bit to 64-bit, solution: a mov instruction with a double-word register destination operand with a doubleword source operand will zero the upper-order double-word of the quadword destination
register

```
mov al, 50
movzx, rbx, al
```

For A register
cbw, cwd, cwde, cdq, cdqe, cdo

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`

Signed

```
mov bl, -50
movsx rax, bl
call writeNum
```

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]`

Arithmetic

Add

`add`
If a memory to memory addition operation is required, two instructions must be used
neg, not, adc(arry), sbb(orrow)

`inc`
When using a memory operand, the explicit type specification (e.g., byte, word, dword, qword) is required to clearly define the size.

```
inc rax

int byte [bNum]
```

`adc`
`<dest> = <dest> + <src> + <carryBit>`
Example adding 128-bit numbers

```
dquad1 ddq 0x1A000000000000000
dquad2 ddq 0x2C000000000000000
dqSum ddq 0

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
```

Sub

`sub`

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

`dec`

Mul

`mul`

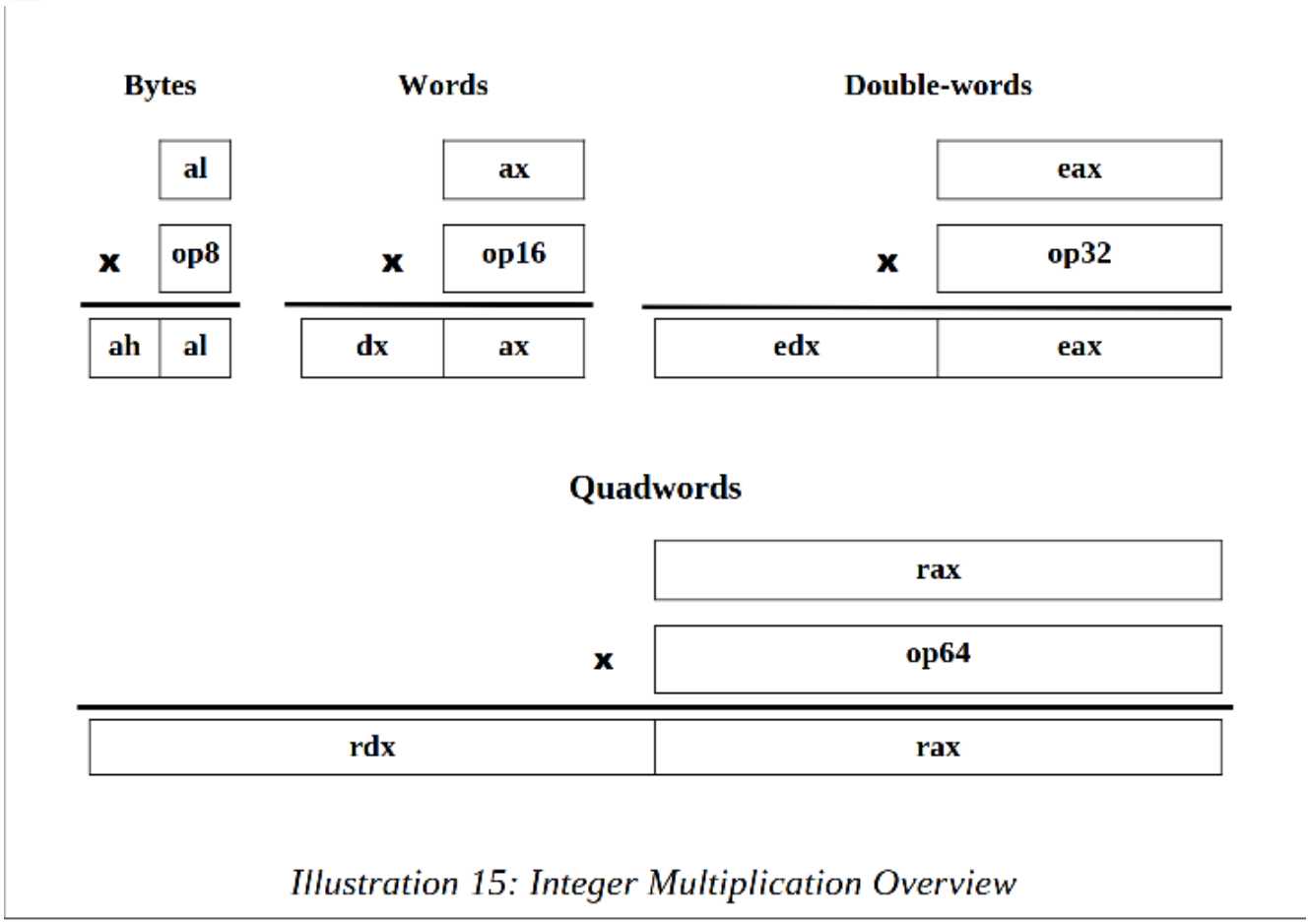


Illustration 15: Integer Multiplication Overview

```
; dqAns4 = qNumA * qNumB
mov rax, qword [qNumA]
mul qword [qNumB] ; result in rdx:rax
mov qword [dqAns4], rax
mov qword [dqAns4+8], rdx
```

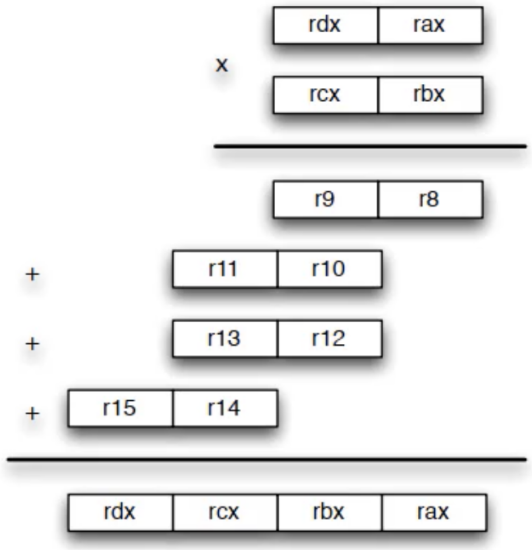
Multiply by a constant

```
mov rax, 50
imul rax, 10 ; rax = rax*10
```

imul

```
; dAns2 = dNumA * dNumB
mov eax, dword [dNumA]
imul eax, dword [dNumB] ; result in eax
mov dword [dAns2], eax
```

مثال: ضرب دو عدد ۱۲۸ بیتی




```
Mul128:
  mov r14, rax
  mov r15, rdx
  mul rbx
  mov r8, rax
  mov r9, rdx
  mov rax, r15
  mul rbx
  mov r10, rax
  mov r11, rdx
  mov rax, r14
  mul rcx
  mov r12, rax
  mov r13, rdx
  mov rax, r15
  mul rcx
  mov r14, rax
  mov r15, rdx
```

`imul`

Cannot use 8-bit operand as dest

Div

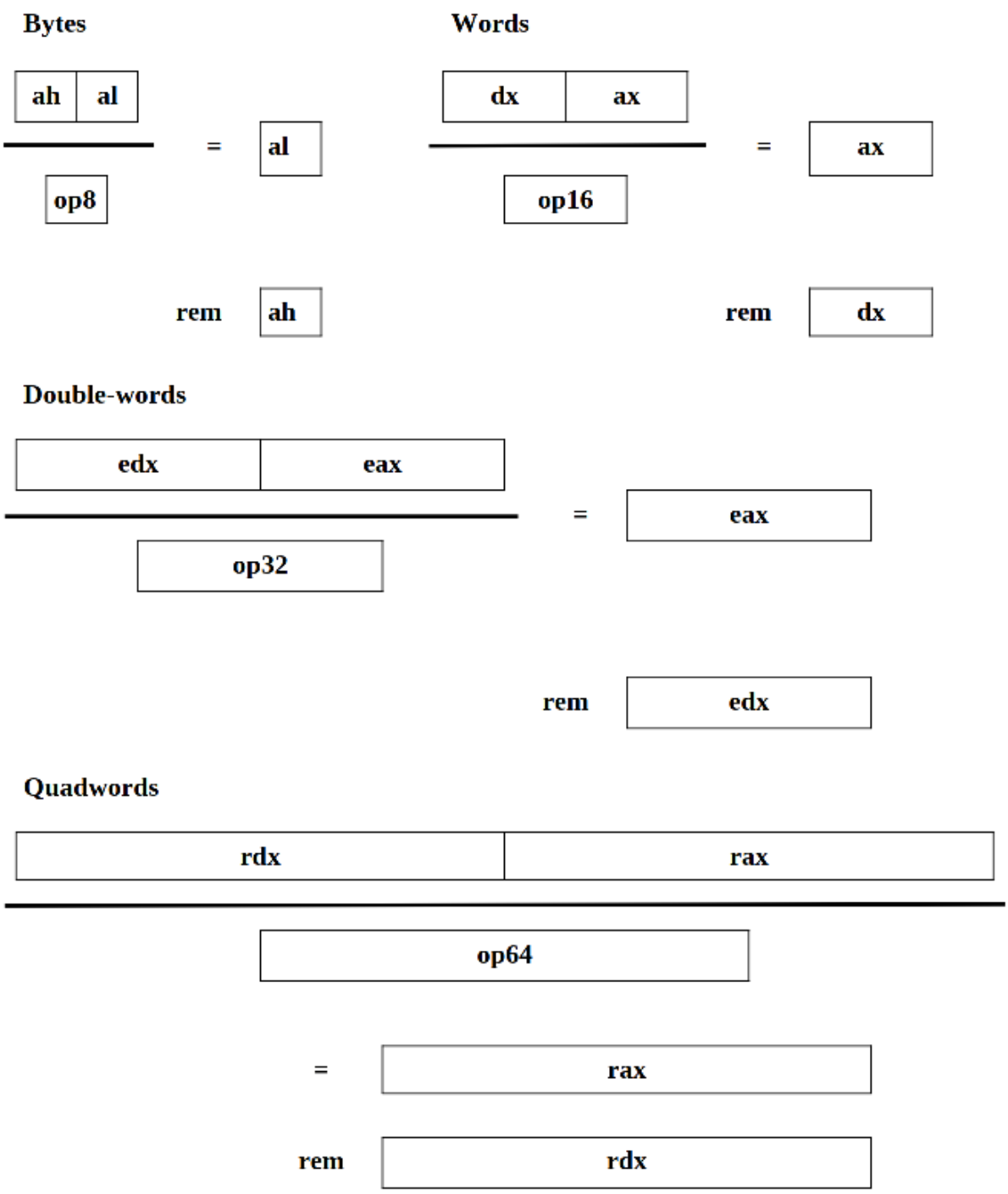


Illustration 16: Integer Division Overview

`div,idiv src`

Example for doubleWords

```
; 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
```

Shift

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

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

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

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

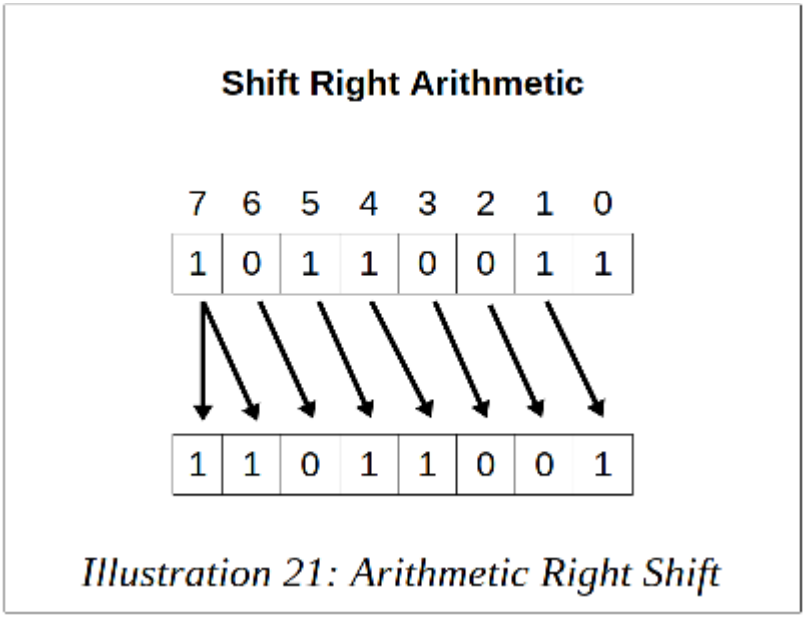


Illustration 21: Arithmetic Right Shift

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

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

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

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

Rotate

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`

Addressing

```
dquad1 ddq 0x1A000000000000000
mov rax, qword [dquad1]
mov rdx, qword [dquad1+8]
```

Access Array

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

lst dd 101, 103, 105, 107

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

Jumps

JUMPS Unsigned (Cardinal)			
JA	Jump if Above	JA Dest	(= JNBE)
JAE	Jump if Above or Equal	JAE Dest	(= JNB = JNC)
JB	Jump if Below	JB Dest	(= JNAE = JC)
JBE	Jump if Below or Equal	JBE Dest	(= JNA)
JNA	Jump if not Above	JNA Dest	(= JBE)
JNAE	Jump if not Above or Equal	JNAE Dest	(= JB = JC)
JNB	Jump if not Below	JNB Dest	(= JAE = JNC)
JNBE	Jump if not Below or Equal	JNBE Dest	(= JA)
JC	Jump if Carry	JC Dest	
JNC	Jump if no Carry	JNC Dest	

Cond Move

شکل کلی این دستورات به صورت زیر است:

$$\text{cmovcc } \{\text{r/m}\},\{\text{r/m}\}$$

که در آن

$$cc \in \{z,nz,e,ne,s,ns,c,nc,a,na,b,nb,ae,nae,be,\\ nbe,g,ng,l,nl,ge,nge,le,nle,p,np,o,no\}$$

داده‌ها در صورت برقراری شرط انتقال داده می شوند.
مثال:

```
mov     cx, 8
cmp     cx, 10
cmovz   ax, 100
cmova   ax, 40
cmovbe ax, 5
```

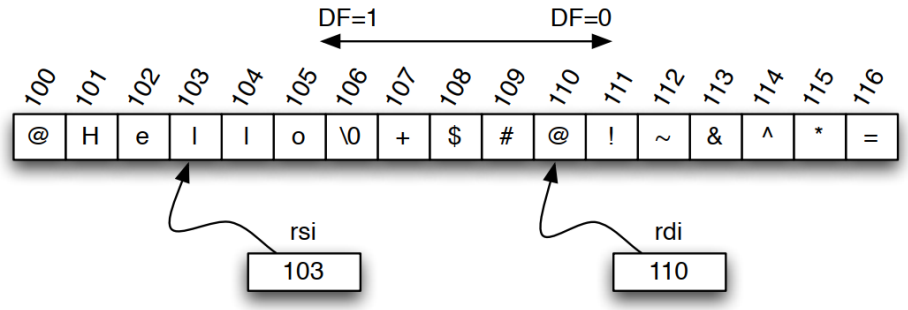
Loops

```
section   .data
     A      dw        50,70,30,20,0,80,40
     len    equ       ($-A)/2

section .text                ;Code Segment
     global _start

_start:
     mov       rax,-1
     mov       rcx, len
     mov       rsi, A
     xor       rbx, rbx
     dec       rbx
again:
     inc       rbx
     cmp       word [rsi+rbx*2], 0
     loopne    again
     cmove     rax, rbx
     call      writeNum
end:
     mov       rax, 60
     mov       rbx, 0
     syscall
```

String manipulation

دستورات رشته‌ای

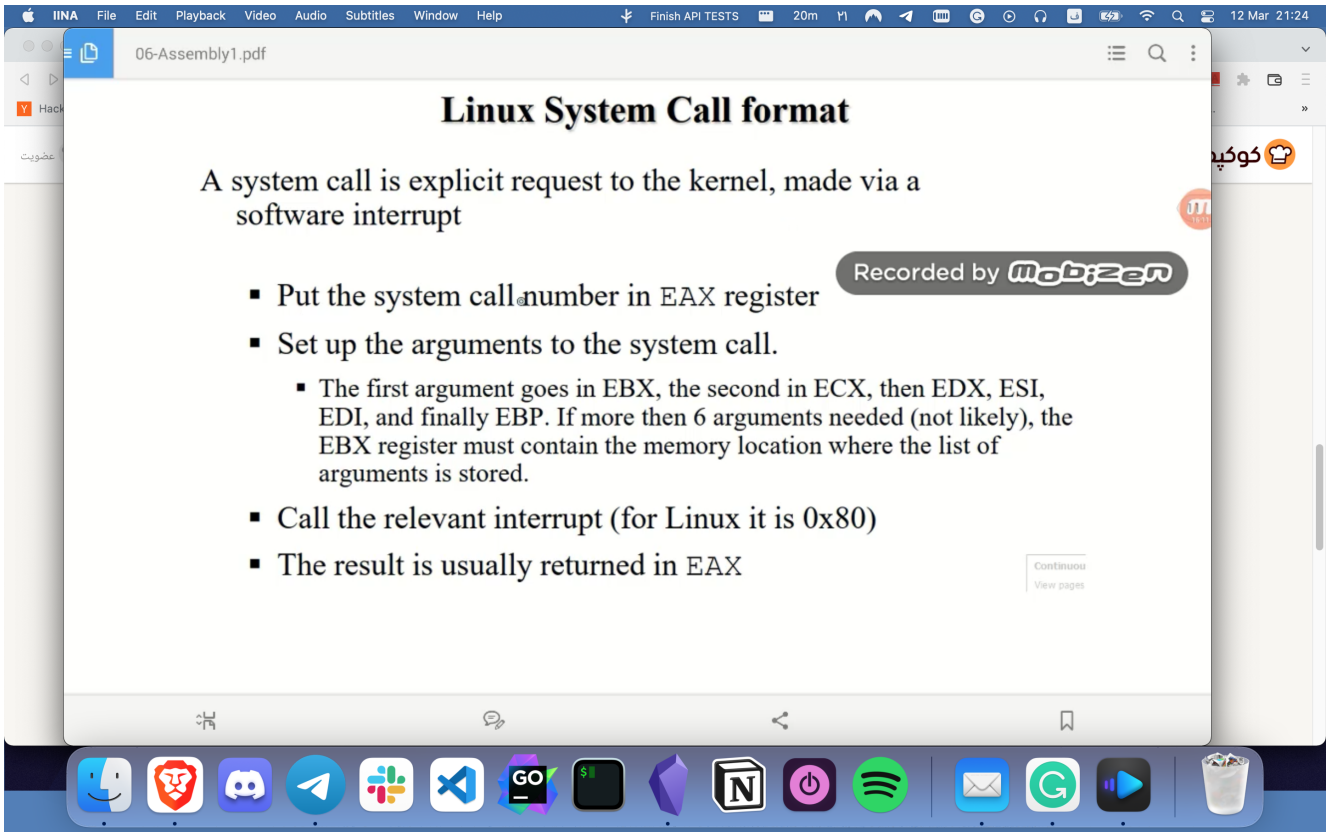
دستورات رشته‌ای از دو ثبات rsi و rdi به ترتیب برای اشاره به رشته اولیه و رشته نهایی استفاده می‌کنند (در حالت 16 بیتی از ثبات‌های si و di ، در حالت 32 بیتی از ثبات‌های esi و edi و در حالت 64 بیتی از ثبات‌های rsi و rdi استفاده می‌کنند). همچنین از پرچم DF [2] برای حرکت روی رشته به سمت جلو و یا برعکس استفاده می‌کنند.
اگر مقدار DF صفر باشد به سمت جلو و اگر یک باشد به سمت عقب جرکت می شود.



Check positive or negative

```
call    readNum
cmp     rax, 0
cmovl   r10,[minus]
cmovge  r10,[zero]
mov     rcx,0
mov     rbx, 10
```

Sys Calls



New line

https://stackoverflow.com/questions/41091375/insert-a-new-line-assembly-8086

Arrays

Save num in Array

Read input to array and print

```
%include "in_out.asm"

section .bss
    a:   resd 1000
section .data

section .text
    global _start
_start:
    call readNum
        mov r8, rax
        mov r9, 0
        mov rsi, a
loop:
        cmp r8, r9
        je printArray
    call readNum
    mov [rsi], al ; Move a number in range 0,255 to a
    inc rsi
        inc r9
    jmp loop

printArray:
    mov r9, 0
```

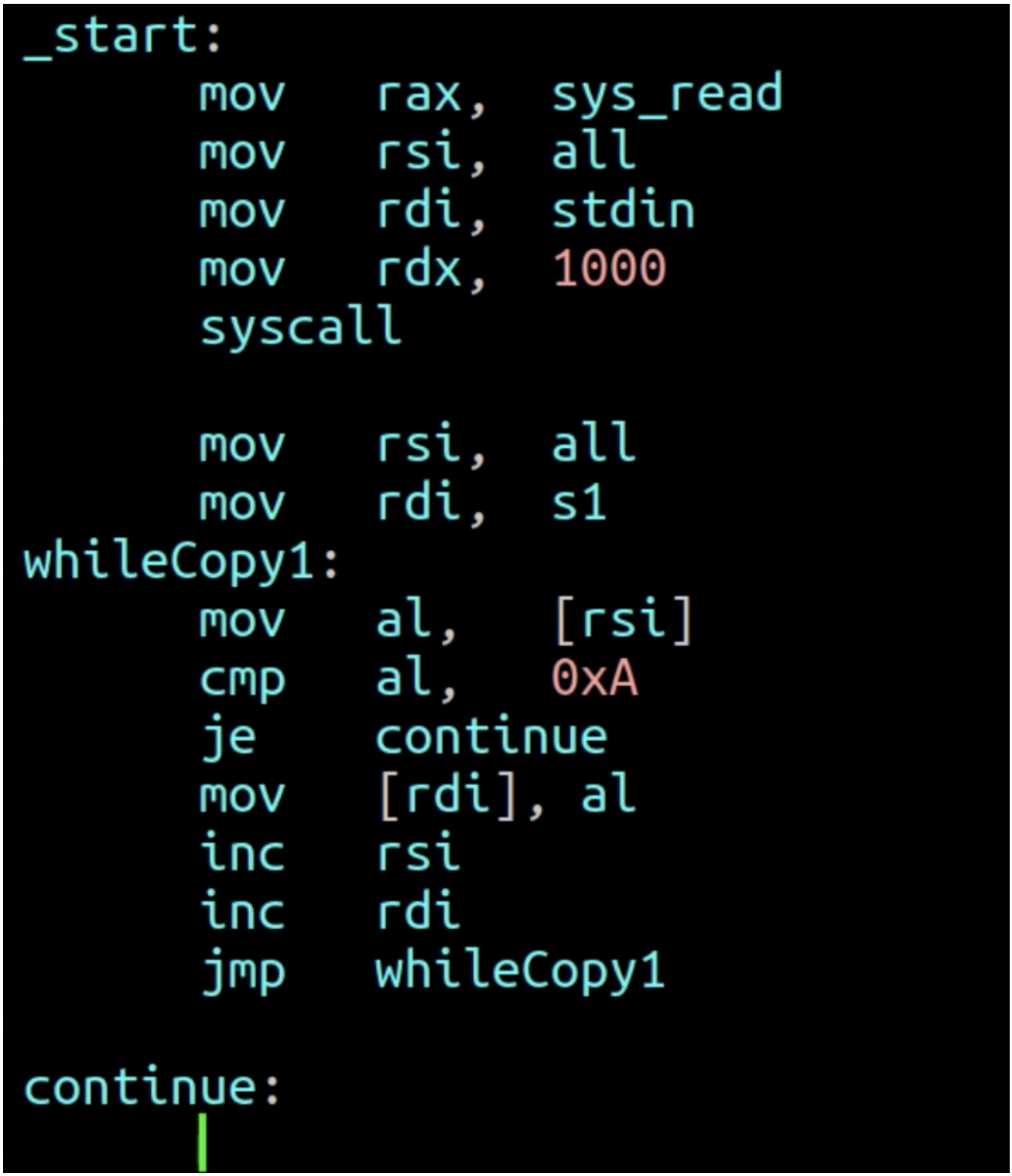
```
    mov rsi, a
printNextElement:
        cmp r8, r9
        je printArrayReverse
        mov al, [rsi]
        call writeNum
        inc r9
        inc rsi
        jmp printNextElement

printArrayReverse:
        call newLine
        mov r9, r8
        dec r9
        mov rsi, a
        add rsi, r9
printNextElementReverse:
        cmp r9, 0
        jb Exit
        mov al, [rsi]
        call writeNum
        dec rsi
        dec r9
        jmp printNextElementReverse


Exit:
    mov eax,1
    mov edi, 0
    int 80h
```

Read string to Array

```
section .bss
    a resd 1000
_start:
    mov rax, sys_read
    mov rsi, a
    mov rdi, stdin
    mov rdx, 100
    syscall

    mov rsi, a
    add rsi, 5 ; Skip first 5 characters from string
    call printString
```

Files

Read from file line by line



Idioms

0 the register

```
xor rax, rax
```

Check if register is zero

```
test rcx, rcx
je ... ; will jump if rcx is zero
```

```
test rax, rax
cmovne  ... ; will move data if rax is not zero
```

Floating Point

XMM1 - XMM15 are floating point registers

Stack

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`

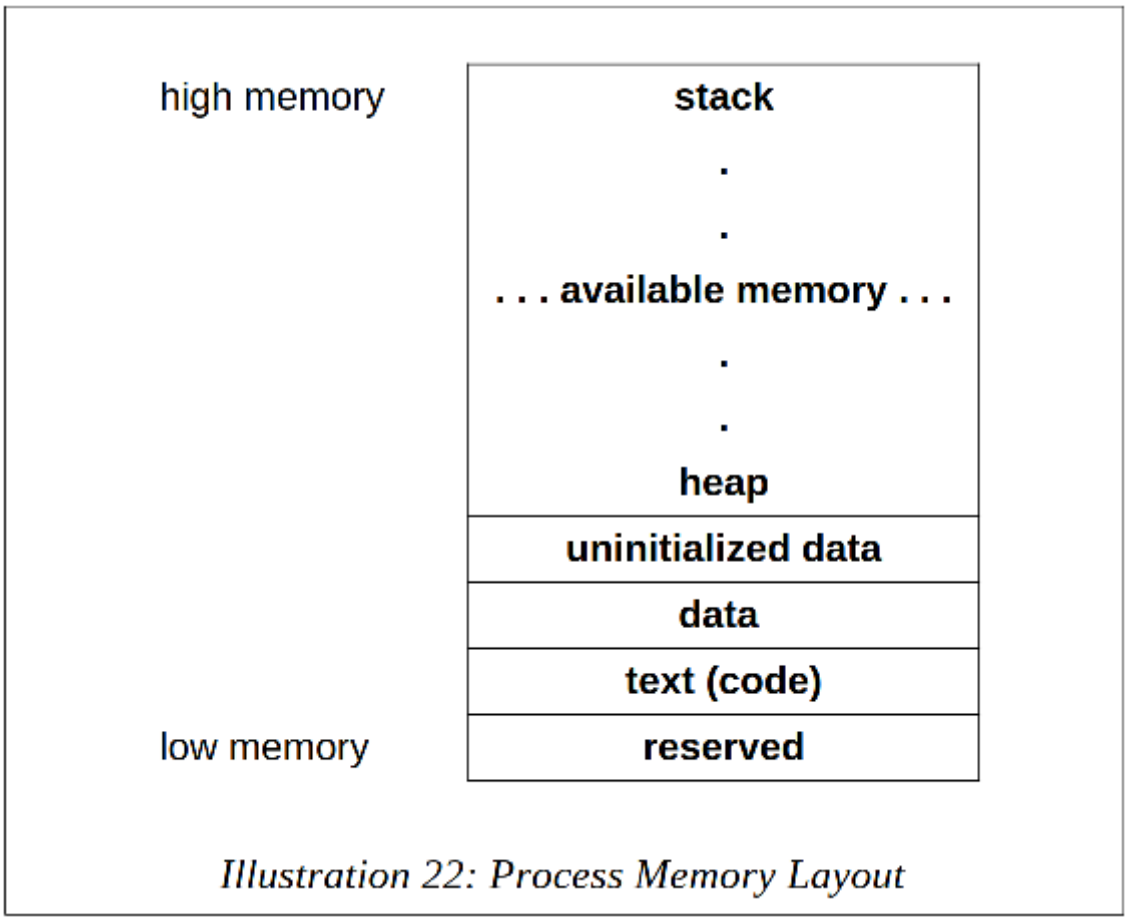


Illustration 22: Process Memory Layout

Macro

Macros will exand in the source code
Single line

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

Multiline:

```
%macro abs, 1; 1 is number of args
        cmp %1, 0 ; here we access the first argument
        jge %%done
        neg %1
%%done:

%endmacro
; invoke macro

mov eax, -3
abs eax
```

Command Line Arguments

Check dec usage for non registers
Move loop bounds to a variable

Functions

Creating a function and passing parameters with stack

```
F:
        enter 24, 0 ; set rsp to rbp so rbp-8 is the first var
        [rbp-8]

        leave ; move rbp to rsp
        ret
```

Access to array passed via stack in function:

```
func:
mov r8, qword [rbp+16] ; rbp+16 is the array address
mov rax, [r8] ; now [r8] is the first element
mov rax, [r8+1*8] ; second element
```