Referência – Programação IA-32 em NASM

Nota: esta referência rápida é apenas um auxiliar e não dispensa o estudo dos acetatos.
Uma referência ainda mais condensada (ver ficheiro no CLIP) estará disponível no teste prático.

Registos do CPU					
eax (32 bits)			**esi** (32 bits)		
	ax (16 bits)			**si** (16 bits)	
	ah (8 bits)	**al** (8 bits)			
ebx (32 bits)			**edi** (32 bits)		
	bx (16 bits)			**di** (16 bits)	
	bh (8 bits)	**bl** (8 bits			
ecx (32 bits)			**ebp** (32 bits)		
	cx (16 bits)			**bp** (16 bits)	
	ch (8 bits)	**cl** (8 bits			
edx (32 bits)			**esp** (32 bits)		
	dx (16 bits)			**sp** (16 bits)	
	dh (8 bits)	**dl** (8 bits			

Instruções para movimento de dados		
op1 e *op2* não podem ser ambos endereços de memória; **não** afectam as flags. Na coluna da esquerda está um exemplo do uso de cada instrução.		
mov *op1, op2*	*op1 = op2;*	`mov eax, [x]`
Instruções aritméticas		
op1 e *op2* não podem ser ambos endereços de memória		
inc *op*	*op ++;*	`inc esi`
dec *op*	*op --;*	`dec ecx`
add *op1, op2*	*op1 = op1 + op2;*	`add eax, [i]`
sub *op1, op2*	*op1 = op1 – op2;*	`sub bl, al`
cmp *op1, op2*	(*op1 – op2*)	`cmp ecx, 0` `je ciclo`
neg *op*	*op = –op;*	`neg dword [n]`
Instruções lógicas e de bits		
op1 e *op2* não podem ser ambos endereços de memória; *n* ≤ número de bits de *op*		
or *op1, op2*	*op1 = op1 | op2;*	`or eax, 0x20`
and *op1, op2*	*op1 = op1 & op2;*	`and eax, 1`
test *op1*, op2	(*op1 & op2*)	`test al, 0x80` `jz fim`
xor *op1, op2*	*op1 = op1 ^ op2;*	`xor eax, eax`
shl *op, n*	*op = op << n; (unsigned)*	`shl al, 1`
shr *op, n*	*op = op >> n; (unsigned)*	`shr edx, 2`
sal *op, n*	*op = op << n; (signed)*	`sal dword[z], 4`
sar *op, n*	*op = op >> n; (signed)*	`sar ebx, 1`
ror *op, n*	Rotação de bits p/ direita	`ror al, 1`
rol *op, n*	Rotação de bits p/ esquerda	`rol ah, 2`

Multiplicações e Divisões		
op tem de ser um registo ou uma posição de memória – não pode ser um valor imediato operações ilustradas para **32 bits** – para **16 bits** usam **ax**, **dx** e **edx:eax** em vez de **eax**, **edx** e **edx:eax**; para **8 bits** usam **al**, **ah** e **ax** em vez de **eax**, **edx** e **edx:eax** **edx:eax** significa combinar eax e edx como um único registo de **64 bits**		
mul *op*	`edx:eax = eax * op`	sem sinal
imul *op*	`edx:eax = eax * op`	com sinal
div *op*	`eax = edx:eax / op` `edx = edx:eax % op`	sem sinal
idiv *op*	`eax = edx:eax / op` `edx = edx:eax % op`	com sinal

Flags			
CF – Carry flag	**ZF** – Zero flag	**OF** – Overflow flag	**SF** – Signal flag
1 se houve transporte (números sem sinal)	**1** se resultado = 0	**1** se houve overflow (números com sinal)	**1** se resultado negativo (números com sinal)
Saltos			
jmp *label*	Salto incondicional	**Igualdade (ZF)**	
Zero Flag		**je** *label*	Salta se igual
jz *label*	Salta se **ZF** = 1	**jne** *label*	Salta se diferentes
jnz *label*	Salta se **ZF** = 0	**Números sem sinal (CF e ZF)**	
Carry Flag		**ja** *label*	Salta se maior
jc *label*	Salta se **CF** = 1	**jae** *label*	Salta se maior ou igual
jnc *label*	Salta se **CF** = 0	**jb** *label*	Salta se menor
Overflow Flag		**jbe** *label*	Salta de menor ou igual
jo *label*	Salta se **OF** = 1	**Números com sinal (SF, OF e ZF)**	
jno *label*	Salta se **OF** = 0	**jg** *label*	Salta se maior
Signal Flag		**jge** *label*	Salta se maior ou igual
js *label*	Salta se **SF** = 1	**jl** *label*	Salta se menor
jns *label*	Salta se **SF** = 0	**jle** *label*	Salta de menor ou igual

Pilha e Subrotinas			
push *op*	empilhar(*op)*	**call** *label*	chamada de subrotina
pop *op*	*op* = desempilhar()	**ret**	retorno de subrotina
Várias			
int *n*	*Interrupção por software*	*n* – código da interrupção - 8 bits	**nop** no operation

Directiva		Modificador		Tamanho
db	`db 'a'`	**byte**	`mov byte [ebx], 'x'`	8 bits
dw	`dw 0`	**word**	`add word [ebx + esi*2], 5`	16 bits
dd	`dd 123`	**dword**	`cmp dword [x], 5`	32 bits
dq	`dq 1.0`	**qword**	`fld qword [x]`	64 bits
dt	`dt 2.1`	**tword**	`fstp tword [y]`	80 bits

Modos de endereçamento		
Na coluna da direita estão os modos de endereçamento básicos do CPU – note com nasm podemos usar labels e expressões constantes para expressar o endereço mais comodamente, como ilustrado pelos exemplos na coluna da esquerda. *Escala* pode ser 1, 2, 4 ou 8		
Constante	Imediato	`mov al, '0'`
Reg	Registo	`mov eax, ebx`
[*constante*]	Directo	`mov edx, [x]`
[*reg32bits*]	Indirecto por registo	`mov bl, [esi]`
[*constante* + *reg32bits*escala*]	Indexado	`mov eax, [v+(esi+1)*4]`
[*reg32bits* + *constante*]	Baseado	`mov esi, [ebp+8]`
[*reg32bits* + *reg32bits*escala* + *constante*]	Baseado Indexado	`mov ch, [ebx + edi]`

Unidade de Reais – Máquina de Pilha

Transferência de dados			
op têm que ser uma posição de memória – não pode ser registo nem constante			
fld *op*	empilhar real na fpu	`fld qword [x]`	empilhar x (real 64 bits)
fild *op*	empilhar inteiro na fpu	`fld dword [i]`	empilhar i (inteiro 32 bits)
fldz	empilhar 0 na fpu		
fld1	empilhar 1 na fpu		
fldpi	empilhar π na fpu		
fstp *op*	Desempilhar da fpu; Guardar real na memória	`fstp dword[x]`	desempilhar para x (real de 32 bits)
fistp *op*	Desempilhar da fpu Guardar inteiro na memória	`fistp word[i]`	desempilhar para i (inteiro de 16 bits)

Manipulação da pilha	
fld `st0`	Duplicar topo da pilha da fpu
ffreep `st0`	Eliminar topo da pilha da fpu
fxch	Trocar st0 e st1

Aritméticas			
faddp `st1`	st1 = st1 + st0; pop	**fmulp** `st1`	st1 = st1 * st0; pop
fsubp `st1`	st1 = st1 - st0; pop	**fdivp** `st1`	st1 = st1 / st0; pop
fsubrp `st1`	st1 = st0 – st1; pop	**fdivrp** `st1`	st1 = st0 / st1; pop
fchs	st0 = -st0	**fabs**	st0 = |st0|

Comparação				
fcomi `st1`	(st0– st1)	afecta **CF** e **ZF**	`fcomi st1` `jae label`	Salta para label se st0 ≥ st1

Funções matemáticas			
fsin	st0 = sin(st0) (radianos)	**fpatan**	st1 = arctan(st1/st0); pop
fcos	st0 = cos(st0) (radianos)	**fsqrt**	st0 = $\sqrt{st0}$
fptan	st0 = tan(st0) (radianos)		