

x86-64

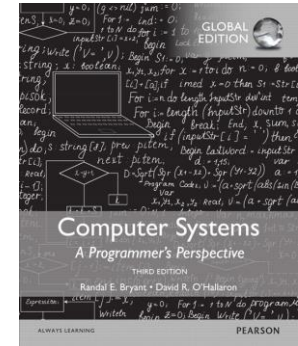
Intel architecture (AT&T syntax)

Bib: Computer Systems: A Programmer's Perspective
x86-64 Machine-Level Programming (adenda ao cap. 3 do livro anterior)

Programação em Sistemas Computacionais

João Pedro Patriarca (joao.patriarca@isel.pt), Gabinete F.0.23 do edifício F

ISEL, ADEETC, LEIC



Agenda

- Características base da arquitetura x86-64
- Instruções *assembly* de uso geral (*general purpose*)

Agenda

- Características base da arquitetura x86-64
- Instruções *assembly* de uso geral (*general purpose*)

Desafio

Interpretar a funcionalidade da função *xpto*

```
00000000000001169 <xpto>:
```

```
1169: f3 0f 1e fa endbr64  
116d: b8 00 00 00 00      mov     $0x0,%eax  
1172: 48 85 ff              test    %rdi,%rdi  
1175: 74 0c                 je      1183 <xpto+0x1a>  
1177: 89 fa                 mov     %edi,%edx  
1179: 83 e2 01              and     $0x1,%edx  
117c: 01 d0                 add     %edx,%eax  
117e: 48 d1 ef              shr     %rdi  
1181: eb ef                 jmp     1172 <xpto+0x9>  
1183: c3                    retq
```

Utilitário *objdump*

- Permite realizar o *disassembly* de um ficheiro objeto (código nativo)

```
$ objdump -d xpto.o > xpto.od
```

-d: disassembla apenas conteúdo de secções com código para o ficheiro *xpto.od*

```
$ objdump -D xpto.o > xpto.od
```

-D: disassembla conteúdo de todas as secções para ficheiro *xpto.od*

- permite observar o conteúdo das secções de dados e constantes

```
$ objdump -j .data -s xpto.o
```

-j: apresenta informação apenas da secção *.data*

-s: apresenta todo o conteúdo da secção referida com a opção *-j*

- O *output* é colocado, por omissão, na consola
 - Usar '|' (*pipe*) entre a aplicação *objdump* e *less* para navegar no *output*
- O ficheiro com código fonte deve ser compilado com a opção de otimização *-Og*

```
$ gcc -Wall -pedantic -Og xpto.c -o xpto
```

Principais características

- Ponteiros e inteiros longos de 64 bits
- Suporte de operações aritméticas entre inteiros de 8, 16, 32 e 64 bits
- Conjunto de 16 registros para uso geral constituídos por 64 bits
- Maior parte do estado local de um programa mantido em registros
- Passagem de argumentos a funções através de registros (até 6 argumentos)
- Registro RIP (*Instruction Pointer* – 64 bits): mantém a posição atual da execução do programa
- Registro RFLAGS: mantém estado atual da execução do programa (inclui *flags* produzidas pela ALU)

Tipos de instruções (GP) e tipos de operandos

- Tipos de instruções
 - Transferência / Transferência condicional
 - Aritméticas / Lógicas / Deslocamento
 - Transferência de controlo
- Instruções com
 - 0 operandos: *ret*
 - 1 operando: *push %rbx*
 - 2 operandos: *mov %ax, %cx*
- Tipos de operandos:
 - Imediato: *\$456; \$0x2E*
 - Registo: *%rdx; %eax; %bx; %ch; %si*
 - Memória: *var_name; (%esi); 9(%r10); (%rac, %edx); 10(%r8, %eax, 4)*

Registos do CPU de uso geral

63	0	31	0	15	0	15	8	7	0	Convenção C
%rax		%eax		%ax		%ah		%al		Return value
%rbx		%ebx		%bx		%bh		%bl		Callee saved
%rcx		%ecx		%cx		%ch		%cl		4th argument
%rdx		%edx		%dx		%dh		%dl		3rd argument
%rsi		%esi		%si				%sil		2nd argument
%rdi		%edi		%di				%dil		1st argument
%rbp		%ebp		%bp				%bpl		Callee saved
%rsp		%esp		%sp				%spl		Stack pointer
%r8		%r8d		%r8w				%r8b		5th argument
%r9		%r9d		%r9w				%r9b		6th argument
%r10		%r10d		%r10w				%r10b		Caller saved
%r11		%r11d		%r11w				%r11b		Caller saved
%r12		%r12d		%r12w				%r12b		Callee saved
%r13		%r13d		%r13w				%r13b		Callee saved
%r14		%r14d		%r14w				%r14b		Callee saved
%r15		%r15d		%r15w				%r15b		Callee saved

Tipos de dados do C e correspondência com o x86-64

Declaração C	Tipo de dados Intel	GAS suffix	x86-64 size (bytes)
<i>char</i>	Byte	b	1
<i>short</i>	Word	w	2
<i>int</i>	Double word	l	4
<i>unsigned</i>	Double word	l	4
<i>long int</i>	Quad word	q	8
<i>unsigned long</i>	Quad word	q	8
<i>char *</i>	Quad word	q	8
<i>float</i>	Single precision	s	4
<i>double</i>	Double precision	d	8
<i>long double</i>	Extended precision	t	16

8 bits – byte Ex: *movb %al, %bh*

16 bits – word Ex: *movw %ax, %bx*

32 bits – double word Ex: *movl %eax, %ebx*

64 bits – quad word Ex: *movq %rax, %rbx*

Modos de endereçamento

- Forma genérica: endereço = Imm(R_b , R_i , Scale)
 - $R_{b,i}$ - qualquer registo a 32 ou 64 bits
 - Scale - 1, 2, 4 ou 8
 - Imm - definido a 8, 16 ou 32 bits (valores positivos e negativos)

$$\text{Imm}(R_b, R_i, \text{Scale}) \Leftrightarrow R_b + R_i * \text{Scale} + \text{Imm}$$

- A definição de um endereço não exige a presença de todas as componentes
 - Endereço = Imm , (direto) \Leftrightarrow Mem[Imm]
 - Endereço = (R_b) , (indireto) \Leftrightarrow Mem[R_b]
 - Endereço = Imm(R_b) , (baseado) \Leftrightarrow Mem[$R_b + \text{Imm}$]
 - Endereço = (R_b, R_i) , (indexado) \Leftrightarrow Mem[$R_b + R_i$]
 - Endereço = (R_b, R_i, Scale) , (indexado escalado) \Leftrightarrow Mem[$R_b + \text{Scale} * R_i$]

Exemplo de modos de endereçamento

```
mov*1 16(%esi,%edi,4), %eax    ; %eax = Meml[%esi+%edi*4+16]
mov*1 -10(,%ebx,2), %ax       ; %ax  = Memw[%ebx*2-10]
mov*1 -12(%ebp), %ecx         ; %ecx = Meml[%ebp-12]
mov*1 0x54, %rbx              ; %rbx = Memq[0x54]
movl*2 $0x55, 20(%ecx)        ; Meml[%ecx+20] = 0x00000055
movb*2 $0x55, 10(%ecx)        ; Memb[%ecx+10] = 0x55
```

- ^{*1}- opcional a adição do sufixo porque o compilador consegue inferir a dimensão da palavra a transferir pelos registros
- ^{*2}- obrigatória a adição do sufixo porque o compilador não consegue inferir a dimensão da palavra a transferir pelos registros

Exercício com modos de endereçamento

- Considere os seguintes valores nos registos e na memória

%rax	0x100
%rcx	0x1
%rdx	0x3

0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

- Qual o valor que fica no registo %b1 para cada uma das instruções?

<code>movb %ah, %b1</code>	%b1	
<code>movb 0x104, %b1</code>	%b1	
<code>movb \$104, %b1</code>	%b1	
<code>movb (%rax), %b1</code>	%b1	
<code>movb 4(%rax), %b1</code>	%b1	
<code>movb 9(%rax,%edx), %b1</code>	%b1	
<code>movb 0xFC(,%ecx,4), %b1</code>	%b1	
<code>movb (%rax,%rdx,4), %b1</code>	%b1	

Exercício com modos de endereçamento

- Considere os seguintes valores nos registos e na memória

%rax	0x100	0x100	0xFF
%rcx	0x1	0x104	0xAB
%rdx	0x3	0x108	0x13
		0x10C	0x11

- Qual o valor que fica no registo %b1 para cada uma das instruções?

<code>movb %ah, %b1</code>	%b1	1	%b1 = %ah
<code>movb 0x104, %b1</code>	%b1	0xAB	%b1 = Mb[0x104]
<code>movb \$104, %b1</code>	%b1	104	%b1 = 104
<code>movb (%rax), %b1</code>	%b1	0xFF	%b1 = Mb[%rax]
<code>movb 4(%rax), %b1</code>	%b1	0xAB	%b1 = Mb[%rax+4]
<code>movb 9(%rax,%edx), %b1</code>	%b1	0x11	%b1 = Mb[%rax+%edx+9]
<code>movb 0xFC(,%ecx,4), %b1</code>	%b1	0xFF	%b1 = Mb[%ecx*4+0xFC]
<code>movb (%rax,%rdx,4), %b1</code>	%b1	0x11	%b1 = Mb[%rax+%rdx*4]

Agenda

- Características base da arquitetura x86-64
- Instruções *assembly* de uso geral (*general purpose*)

Instruções de transferência (1 de 3)

Instrução		Efeito	Exemplo
mov	<i>S, D</i> reg, reg reg, mem mem, reg imm, reg imm, mem	$D \leftarrow S$	mov %rax, %r10 mov %ebx, var1 mov array(%esi), %cx mov \$stack_top, %rsp movl \$0x1234, var2
movabs	<i>I, R</i>	$R \leftarrow I$	movabs \$0x123456789abcdef0, %r12
movsx	<i>S, R</i> reg8, reg16 reg8, reg32 reg8, reg64 reg16, reg32 reg16, reg64 mem8, reg16 mem8, reg32 mem8, reg64 mem16, reg32 mem16, reg64	$R \leftarrow \text{SignExtended}(S)$	movsbw %ch, %r9w movsbl %sil, %eax movsbq %dl, %r15 movswl %ax, %edx movswq %r10w, %r12 movsbw var0, %r9w movsbl var1, %eax movsbq (%rsp), %r15 movswl 8(%rbx), %edx movswq (%rbx,%eax), %r12

Instruções de transferência (2 de 3)

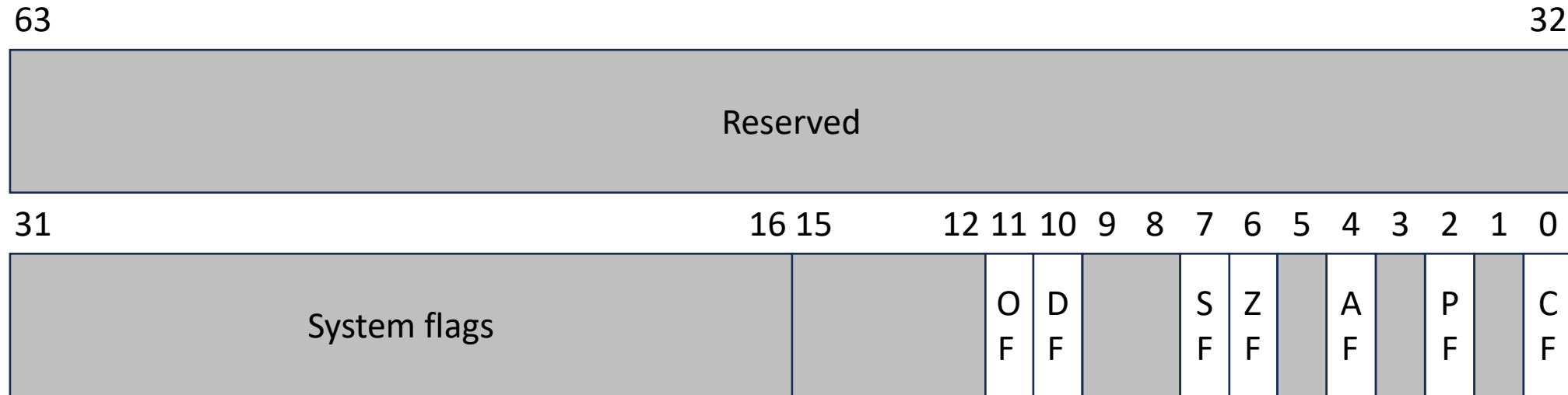
Instrução		Efeito	Exemplo
movslq	S32, R64 reg32, reg64 mem32, reg64	$R64 \leftarrow S32$	movslq %eax, %r10 movslq array(%esi), %rcx
movzx	S, R	$R \leftarrow ZeroExtended(S)$	O mesmo tipo de operandos que a instrução movsx
push	S reg64 mem64 imm64	$\%rsp \leftarrow \%rsp - 8;$ $M[\%rsp] \leftarrow S$	push %r11 push (%rbx) push \$0
pop	D reg64 mem64	$D \leftarrow M[\%rsp];$ $\%rsp \leftarrow \%rsp + 8$	pop %r8 pop var
xchg	D, R mem, reg reg, reg	$tmp \leftarrow D$ $D \leftarrow R$ $R \leftarrow tmp$	xchg var, %eax xchg %r8, %r9
lea	M, D mem, reg	Load effective address $D \leftarrow address(M)$	lea var, %rbx lea 16(%rsi,%rcx,4), %r10

Instruções de transferência (3 de 3)

Moves condicionais

Instrução		Sinónimo	Condição	Descrição
<code>cmove</code>	S, D	<code>cmovz</code>	ZF	Equal/Zero
<code>cmovne</code>	S, D	<code>cmovnz</code>	$\sim ZF$	Not equal / not zero
<code>cmovs</code>	S, D		SF	Negative
<code>cmovns</code>	S, D		$\sim SF$	Nonnegative
<code>cmovg</code>	S, D	<code>cmovnl</code>	$\sim (SF \wedge OV) \& \sim ZF$	Greater (signed)
<code>cmovge</code>	S, D	<code>cmovnl</code>	$\sim (SF \wedge OV)$	Greater or equal (signed)
<code>cmovl</code>	S, D	<code>cmovnge</code>	$SF \wedge OV$	Less (signed)
<code>cmovle</code>	S, D	<code>cmovng</code>	$(SF \wedge OV) / ZF$	Less or equal (signed)
<code>cmova</code>	S, D	<code>cmovnbe</code>	$\sim CF \& \sim ZF$	Above (unsigned)
<code>cmovae</code>	S, D	<code>cmovnb</code>	$\sim CF$	Above or equal (unsigned)
<code>cmovb</code>	S, D	<code>cmovnae</code>	CF	Below (unsigned)
<code>cmovbe</code>	S, D	<code>cmovna</code>	CF / ZF	Below or equal (unsigned)

Registo RFlags (*flags* visíveis ao software aplicativo)



OF – *Overflow Flag*

DF – *Direction Flag*

SF – *Sign Flag*

ZF – *Zero Flag*

AF – *Auxiliary Carry Flag*

PF – *Parity Flag*

CF – *Carry Flag*

System flags – *Flags visíveis para software de sistema*

Instruções de manipulação de *flags*

Instrução		Descrição	Flags ODITSZAPC	Exemplo
lahf		AH = EFLAGS & 0xD5 7 6 4 2 0 = S Z A P C	-----	
sahf		EFLAGS = AH & 0xD5 S Z A P C = 7 6 4 2 0	----MMMM	
pushf		RSP = RSP - 8; Mem[RSP] = RFLAGS	-----	
popf		RFLAGS = Mem[RSP]; RSP = RSP + 8	MMMMMMMM	
clc		CF = 0 (clear Carry Flag)	-----0	
cmc		CF = ~CF (complement Carry Flag)	-----M	
stc		CF = 1 (set Carry Flag)	-----1	
cld		DF = 0 (clear Direction Flag)	-0-----	
std		DF = 1 (set Direction Flag)	-1-----	
cli		IF = 0 (clear Interrupt flag)	--0-----	
sti		IF = 1 (set Interrupt Flag)	--1-----	
setXX	<i>D</i> <i>reg</i> <i>mem</i>	Byte set or clear based on condition (conditions on next slide) D = XX == true	-----	setXX %al setXX res

Condições (baseadas nas *flags* do registo EFLAGS)

Menmónica	Descrição	Condição
g / nle	greater / not less nor equal (com sinal)	CF == OF && ZF == 0
ge / nl	greater or equal / not less (com sinal)	CF == OF
l / nge	less / not greater nor equal (com sinal)	CF != OF
le / ng	less or equal / not greater (com sinal)	CF != OF ZF == 1
a / nbe	above / not below nor equal (sem sinal)	CF == 0 && ZF == 0
ae / nb	above or equal / not below (sem sinal)	CF == 0
b / nae	below / not above nor equal (sem sinal)	CF == 1
be / na	below or equal / not above (sem sinal)	CF == 1 ZF == 1
p / pe	paraity / parity even	PF == 1
np / po	not parity / parity odd	PF == 0
o	overflow	OF == 1
no	not overflow	OF == 0
s	sign	SF == 1
ns	not sign	SF == 0
e / z	equal / zero	ZF == 1
ne / nz	not equal / not zero	ZF == 0
c	carry	CF == 1
nc	not carry	CF == 0

Instruções aritméticas (1 de 4)

Instrução		Descrição	Flags ODITSZAPC	Exemplo
add	S, D reg, reg mem, reg reg, mem imm, reg imm, mem	$D \leftarrow D + S$	M - - - M M M M M	add %rax, %rbx add name(%ecx), %r8 add %bh, var add \$0x55, %ax addq \$1, i
adc	S, D	$D \leftarrow D + S + CF$	M - - - M M M M M	Mesmos operandos que ADD
inc	D reg mem	$D \leftarrow D + 1$	M - - - M M M M M	inc %al inc var
sub	S, D	$D \leftarrow D - S$	M - - - M M M M M	Mesmos operandos que ADD
sbb	S, D	$D \leftarrow D - S - CF$	M - - - M M M M M	Mesmos operandos que ADD
dec	S, D	$D \leftarrow D - 1$	M - - - M M M M M	Mesmos operandos que INC
neg	D	$D \leftarrow -D$	M - - - M M M M M	Mesmos operandos que INC
cmp	S, D	$D - S$	M - - - M M M M M	Mesmos operandos que ADD

Instruções aritméticas (2 de 4)

Instrução		Descrição	Flags ODITSZAPC	Exemplo
div idiv	op op reg8 mem8 reg16 mem16 reg32 mem32 reg64 mem64	Divisão de números sem sinal Divisão de números com sinal AL = AX / byte AH = AX % byte AX = DX:AX / word DX = DX:AX % word EAX = EDX:EAX / dword EDX = EDX:EAX % dword RAX = RDX:RAX / qword RDX = RDX:RAX % qword	U - - - UUUUU	div %cl divb alpha div %bx divw table(%rsi) div %ebx divl (%rsi) div %rbx divq (%rsi)
mul	op reg8 mem8 reg16 mem16 reg32 mem32 reg64 mem64	Multiplicação de números sem sinal AX = AL * op (byte) DX:AX = AX * op (word) EDX:EAX = EAX * op (dword) RDX:RAX = RAX * op (qword)	M - - - UUUUM	mul %bl mulb month(%rsi) mul %cx mulw baund_rate mul %ebx mull (%rsi) mul %rbx mulq (%rsi)

Instruções aritméticas (3 de 4)

Instrução		Descrição	Flags ODITSZAPC	Exemplo
imul	<i>[[op3], op2], op1</i>	Multiplicação de números com sinal	U- - -UUUUU	
	reg8	AL = AL * op1 (byte)		imul %cl
	mem8			imulb rate
	reg16	DX:AX = AX * op1 (word)		imul %bx
	mem16			imulw red(%rbp, %rdi)
	reg32	EDX:EAX = EAX * op1 (dword)		imul %ebx
	mem32			imulw (%rsi)
	reg64	RDX:RAX = RAX * op1 (qword)		imul %r10
	mem64			imulq (%r10)
	reg, reg	op1 = op1 * op2		imul %rax, %rbx
	mem, reg			imul m, %r14
	imd, reg			imul \$5, %r12
	imd, reg, reg	op1 = op2 * op3		imul \$54, %ax, %bx
	imd, mem, reg			imul \$3, n, %r13

Instruções aritméticas (4 de 4)

Instrução	Descrição	Flags ODITSZAPC	Exemplo
cbw	Estende o sinal de AL para AX	-----	cbw
cwde	Estende o sinal de AX para EAX	-----	cwde
cdqe / cltq	Estende o sinal de EAX para RAX	-----	cdqe
cwd	Estende o sinal de AX para DX:AX	-----	cwd
cdq / cltd	Estende o sinal de EAX para EDX:EAX	-----	cdq
cqo / cqto	Estende o sinal de RAX para RDX:RAX	-----	cqo

Instruções lógicas

Instrução		Descrição	Flags ODITSZAPC	Exemplo
and	<i>S, D</i> reg, reg mem, reg reg, mem imm, reg imm, mem	$D \leftarrow D \& S$	0---MMUM0	and %rax, %rbx and name(%ecx), %r8 and %bh, var and \$0x55, %ax andq \$1, i
test	<i>S, D</i>	$D \& S$	0---MMUM0	Mesmos operandos que AND
or	<i>S, D</i>	$D \leftarrow D \mid S$	0---MMUM0	Mesmos operandos que AND
xor	<i>S, D</i>	$D \leftarrow D \wedge S$	0---MMUM0	Mesmos operandos que AND
not	<i>D</i> reg mem	$D \leftarrow \sim D$	-----	not %al notw var

Instruções de deslocamento

Instrução		Descrição	Flags ODITSZAPC	Exemplo
shld	<i>count, R, D</i> imm, reg, reg imm, reg, mem CL, reg, reg CL, reg, mem	temp = count & 1fh value = concatenate(D, S) value = value << temp D = value	-----	shld \$4, %rbx, %rax shld \$1, %r8w, var shld %cl, %r10d, %ebx shld %cl, %rax, 8(%rsi)
shrd	<i>count, R, D</i>	value = value >> temp	-----	Mesmos operandos que shld
sal/ shl	<i>count, D</i> CL, reg imm8, reg CL, mem imm8, mem	$D \leftarrow D \ll count$	M-----M	sal %cl, %rax shl \$8, %dx shll %cl, (%ebx) salq \$10, var
shr	<i>count, D</i>	$D \leftarrow D \gg count$	M-----M	Mesmos operandos que sal
sar	<i>count, D</i>	$D \leftarrow D \gg count$	M-----M	Mesmos operandos que sal
rol	<i>count, D</i>	$D \leftarrow \text{rotate_left}(D, count)$	M-----M	Mesmos operandos que sal
rcl	<i>count, D</i>	$D \leftarrow \text{rotate_left_C}(D, count)$	M-----M	Mesmos operandos que sal
ror	<i>count, D</i>	$D \leftarrow \text{rotate_right}(D, count)$	M-----M	Mesmos operandos que sal
rcr	<i>count, D</i>	$D \leftarrow \text{rotate_right_C}(D, count)$	M-----M	Mesmos operandos que sal

Instruções de manipulação de bits

Instrução		Descrição	Flags ODITSZAPC	Exemplo
bsf	<i>target, index</i> reg, reg mem, reg	Scan bit forward for(i = 0; target[i] == 0 && i <= 15(31)(63); i++); index = i;	U - - -UMUUU	bsf %rbx, %rax bsf var, %cx
bsr	<i>target, index</i>	Scan bit reverse for(i=15(31)(63); target[i] == 0 && i >= 0; i--); index = i;	U - - -UMUUU	Mesmos operandos que bsf
bt	<i>index, target</i> imm8, reg imm8, mem reg, reg mem, reg	Test bit CF = target[index]	U - - -UUUUM	bt \$53, %rax btw \$13, var bt %ecx, %rdx bt idx, %r10
btc	<i>index, target</i>	Test bit and complement CF = target[index] target[index] = ~ target[index]	U - - -UUUUM	Mesmos operandos que bt
btr	<i>index, target</i>	Test bit and reset	U - - -UUUUM	Mesmos operandos que bt
bts	<i>index, target</i>	Test bit and set	U - - -UUUUM	Mesmos operandos que bt

Instruções de controlo de fluxo (1 de 2)

Instrução		Descrição	Flags ODITSZAPC	Exemplo
jmp	<i>target</i> label reg mem	RIP += offset8(16)(32) RIP = reg RIP = [mem]	-----	jmp .L1 jmp %rbx jmp *switch(%rsi)
jXX	<i>disp</i> disp8 disp64	if (XX is TRUE) RIP += disp (XX – ver slide com as condições)	-----	jXX label
call	<i>target</i> label reg mem	push RIP; RIP += offset8(16)(32) push RIP; RIP = reg push RIP; RIP = [mem]	-----	call strcmp call %rax call *table(%rsi)
ret	<i>[count]</i>	pop RIP pop RIP; RSP = RSP + count	-----	ret ret \$4

Instruções de controlo de fluxo (2 de 2)

Instrução		Descrição	Flags ODITSZAPC	Exemplo
jcxz	<i>disp</i>	jmp if CX is zero	-----	jcx count_done
jecz	<i>disp</i>	jmp if ECX is zero	-----	jec count_done
jrcz	<i>disp</i>	jmp if RCX is zero	-----	jrcz count_done
loop	<i>disp</i>	RCX = RCX – 1; jmp if RCX != 0	-----	loop again
loope/loopz	<i>disp</i>	RCX = RCX – 1; jmp if RCX != 0 && ZF == 1	-----	loope again
loopne/loopnz	<i>disp</i>	RCX = RCX – 1; jmp if RCX != 0 && ZF == 0	-----	loopne again

Análise e observação de código em *assembly*

Opção `-Og` para o compilador gerar código próximo do que seria escrito por um humano

```
> gcc -Og <filename.c> -o <filename>
```

Opção `-d` para *disassembly* da secção `.text`; aplicação `less` para navegar no código apresentado

```
> objdump -d <filename> | less
```

Opções `-j .data -S` para apresentar o conteúdo da secção `.data`

```
> objdump -j .data -S <filename>
```

Análise e observação de código em *assembly*

- Expressões aritméticas
- Deslocamentos e divisões por potências inteiras de 2
- Comparação entre inteiros com e sem sinal
- Alinhamento de valores em memória
- Representação de dados em memória (*endianness*):
 - *little-endian* versus *big-endian*
- Alinhamento dos campos de uma estrutura

Expressões aritméticas

```
long calc_expression_and_ret_long(int a, long b, char c) {  
    long res = a + b + c;  
    return res;  
}
```

```
int calc_expression_and_ret_int(int a, long b, char c) {  
    int res = (int)(a + b + c);  
    return res;  
}
```

```
short calc_expression_and_ret_short(int a, long b, char c) {  
    short res = (short)(a + b + c);  
    return res;  
}
```


Expressões aritméticas - *assembly*

```
calc_expression_and_ret_long:
```

```
    movslq    %edi,%rdi  
    add       %rsi,%rdi  
    movsbq    %dl,%rax  
    add       %rdi,%rax  
    retq
```

```
calc_expression_and_ret_int:
```

```
    lea       (%rdi,%rsi,1),%eax  
    movsbl    %dl,%edx  
    add       %edx,%eax  
    retq
```

```
calc_expression_and_ret_short:
```

```
    lea       (%rdi,%rsi,1),%eax  
    movsbw    %dl,%dx  
    add       %edx,%eax  
    retq
```

Deslocamentos e divisões por potências inteiras de 2

```
int shift_right_uint(uint v, int n) {  
    return v >> n;  
}  
  
int shift_right_int(int v, int n) {  
    return v >> n;  
}  
  
int divide_by_four_uint(uint v) {  
    return v / 4;  
}  
  
int divide_by_four_int(int v) {  
    return v / 4;  
}
```

Deslocamentos e divisões por potências inteiras de 2 - *assembly*

shift_right_uint:

```
mov    %edi,%eax
mov    %esi,%ecx
shr    %cl,%eax
retq
```

shift_right_int:

```
mov    %edi,%eax
mov    %esi,%ecx
sar    %cl,%eax
retq
```

divide_by_four_uint:

```
mov    %edi,%eax
shr    $0x2,%eax
retq
```

divide_by_four_int:

```
lea    0x3(%rdi),%eax
test   %edi,%edi
cmovns %edi,%eax
sar    $0x2,%eax
retq
```

Comparação entre inteiros com e sem sinal

```
int int_cmp(int v1, int v2) {  
    if (v1 < v2)  
        return -1;  
    if (v1 > v2)  
        return 1;  
    return 0;  
}  
  
int uint_cmp(uint v1, uint v2) {  
    if (v1 < v2)  
        return -1;  
    if (v1 > v2)  
        return 1;  
    return 0;  
}
```

Comparação entre inteiros com e sem sinal - *assembly*

```
int_cmp:
    cmp    %esi,%edi
    jl     int_cmp_0x10
    jg     int_cmp_0x16
    mov     $0x0,%eax
    retq
int_cmp_0x10:
    mov     $0xffffffff,%eax
    retq
int_cmp_0x16:
    mov     $0x1,%eax
    retq
```

```
uint_cmp:
    cmp     %esi,%edi
    jb     uint_cmp_0x10
    ja     uint_cmp_0x16
    mov     $0x0,%eax
    retq
uint_cmp_0x10:
    mov     $0xffffffff,%eax
    retq
uint_cmp_0x16:
    mov     $0x1,%eax
    retq
```


Alinhamento de valores em memória

```
int v1 = 0x12345678;
long v2 = 0x1234567890abcdef;

void print_int_long_alignment(int *pv1, long *pv2) {
    printf("pv1=%p\n", (void*)pv1);
    printf("pv2=%p\n", (void*)pv2);
}

void f1_alignment() {
    int *pv1 = &v1;
    long *pv2 = &v2;

    print_int_long_alignment(pv1, pv2);
}
```



pv1=0x555555558030
pv2=0x555555558028

Alinhamento de valores em memória - *assembly*

```
int v1 = 0x12345678;
long v2 = 0x1234567890abcdef;

void print_int_long_alignment(
    int *pv1, long *pv2)
{
    printf("pv1=%p\n", (void*)pv1);
    printf("pv2=%p\n", (void*)pv2);
}

void f1_alignment() {
    int *pv1 = &v1;
    long *pv2 = &v2;
    print_int_long_alignment(
        pv1, pv2);
}
```

Disassembly of section .data:

```
...
00000000000004028 <v2>:
    4028: ef cd ab 90 78 56 34 12
00000000000004030 <v1>:
    4030: 78 56 34 12

00000000000001265 <f1_alignment>:
    ...
    126d: lea     0x2db4(%rip),%rsi
    1274: lea     0x2db5(%rip),%rdi
    127b: callq   print_int_long_alignment
    ...
    1297: retq
```

Representação de dados em memória (*endianness: little-endian versus big-endian*)

```
int v1 = 0x12345678;
long v2 = 0x1234567890abcdef;

void print_int_long_lsb(int *pv1, long *pv2) {
    uchar *pc1 = (uchar *)pv1, *pc2 = (uchar*)pv2;
    printf("*pv1=0x%x\n", *pc1);
    printf("*pv2=0x%x\n", *pc2);
}

void f1_endianness() {
    int *pv1 = &v1;
    long *pv2 = &v2;

    print_int_long_lsb(pv1, pv2);
}
```



*pv1=0x78
*pv2=0xef

Representação de dados em memória (*endianness: little-endian versus big-endian*) - assembly

```
int v1 = 0x12345678;
long v2 = 0x1234567890abcdef;

void print_int_long_lsb(int *pv1,
                        long *pv2)
{
    uchar *pc1 = (uchar *)pv1,
    uchar *pc2 = (uchar*)pv2;
    printf("*pv1=0x%x\n", *pc1);
    printf("*pv2=0x%x\n", *pc2);
}

void f1_endianness() {
    int *pv1 = &v1;
    long *pv2 = &v2;

    print_int_long_lsb(pv1, pv2);
}
```

Disassembly of section .data:

```
...
00000000000004028 <v2>:
    4028: ef cd ab 90 78 56 34 12
00000000000004030 <v1>:
    4030: 78 56 34 12

00000000000001265 <f1_endianness>:
    ...
    1280: lea     0x2da1(%rip),%rsi
    1287: lea     0x2da2(%rip),%rdi
    128e: callq   print_int_long_lsb
    ...
    1297: retq
```

Alinhamento dos campos de uma estrutura

```
struct {  
    long a; int b; char c;  
    char d; long e;  
} struct_ex = {0x1,0x2,0x3,0x4,0x5};
```

```
void f2_struct_fields_addr() {  
    long * pa = &struct_ex.a;  
    int * pb = &struct_ex.b;  
    char * pc = &struct_ex.c;  
    char * pd = &struct_ex.d;  
    long * pe = &struct_ex.e;
```

```
    print_struct_fields_addr(  
        pa, pb, pc, pd, pe
```

```
    );
```

```
}
```

Struct fields address:

```
.a = 0x557f39fa1010  
.b = 0x557f39fa1018  
.c = 0x557f39fa101c
```

```
void print_struct_fields_addr(  
    long * pa, int * pb,  
    char * pc, char * pd,  
    long * pe)  
{  
    printf("Struct fields address:\n"  
        "\t.a = %p\n"  
        "\t.b = %p\n"  
        "\t.c = %p\n"  
        "\t.d = %p\n"  
        "\t.e = %p\n"  
        , (void*)pa, (void*)pb  
        , (void*)pc, (void*)pd  
        , (void*)pe);  
}
```

```
.d = 0x557f39fa101d  
.e = 0x557f39fa1020
```

Alinhamento dos campos de uma estrutura

```
struct {  
    long a; int b; char c;  
    char d; long e;  
} struct_ex = {0x1, 0x2, 0x3, 0x4, 0x5};  
  
...
```

Disassembly of section .data:

```
...  
0000000000004010 <struct_ex>:  
    long a; int b; char c;  
    char d; long e;  
} struct_ex = {  
    0x1, 0x2, 0x3, 0x4, 0x5  
};
```

```
4010:  01 00 00 00 00 00 00 00  
      02 00 00 00 03 04 00 00  
4020:  05 00 00 00 00 00 00 00
```

Exercícios

- Implemente a função *Long abs(Long v)* que retorna o valor absoluto de *v*
- Implemente a função *int toupper(int car)* que retorna a letra maiúscula correspondente à letra minúscula recebida em *car*. Caso o valor recebido não corresponda a uma letra minúscula, retorna o próprio caracter.