Notas de aula

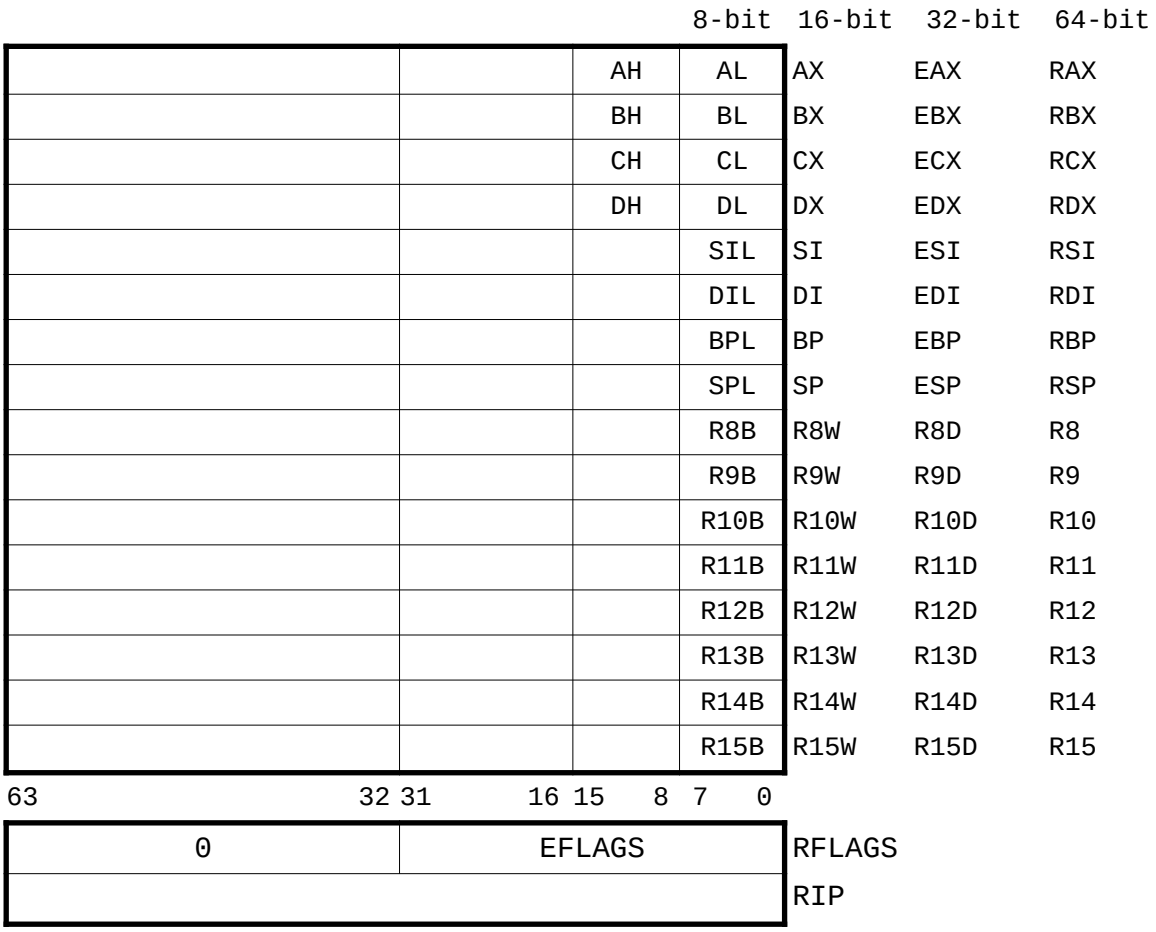
Programação *assembly x86-64*

Ezequiel Conde

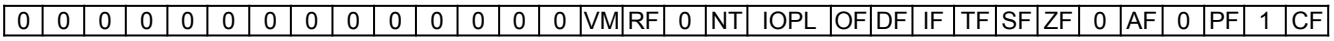
Código dos exemplos disponível em https://github.com/econde/psc_code

Processador x86-64

Registos



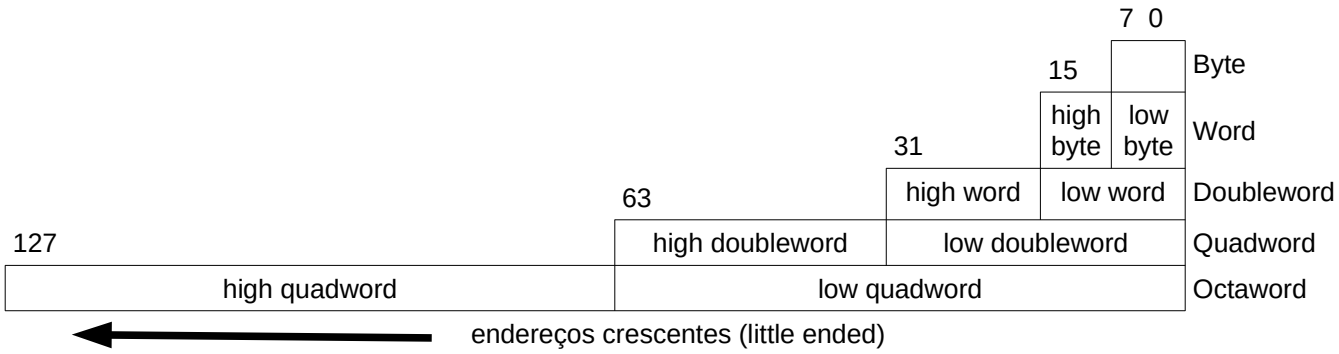
Flags



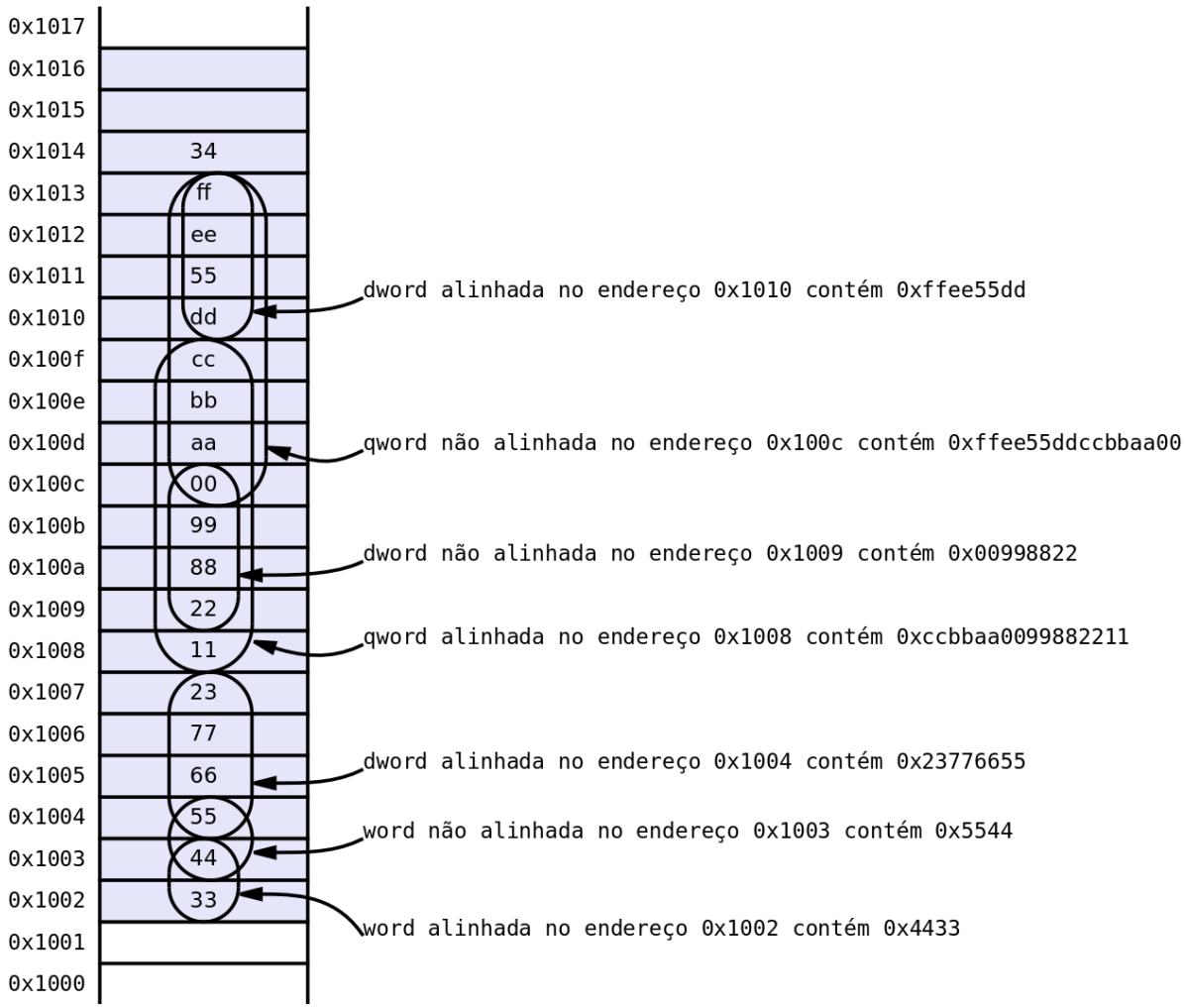
CF- Carry Flag
PF- Parity Flag
AF- Auxiliary Flag
ZF- Zero Flag
SF- Sign Flag
TF- Trap Flag
IF- Interrupt Flag
DF- Direction Flag
OF- Overflow Flag
IOPL- I/O Privilege Level
NT- Nested Task Flag
RF- Resume Flag
VM- Virtual 8086 Mode

Organização dos dados em memória

Designação dos tipos de palavras na terminologia Intel



Declaração C	Designação Intel	Dimensão (Bytes)	Alinhamento
char	byte	1	1
short	word	2	2
int	double word	4	4
unsigned int	double word	4	4
long int	quad word	8	8
unsigned long int	quad word	8	8
float	single precision	4	4
double	double precision	8	8
pointer (char *)	quad word	8	8
struct union		A dimensão de um tipo composto é múltipla do seu alinhamento.	O alinhamento de um tipo composto é igual ao maior alinhamento interno.


0x1017
0x1016
0x1015
0x1014
0x1013
0x1012
0x1011
0x1010
0x100f
0x100e
0x100d
0x100c
0x100b
0x100a
0x1009
0x1008
0x1007
0x1006
0x1005
0x1004
0x1003
0x1002
0x1001
0x1000
34
ff
ee
55
dd
cc
bb
aa
00
99
88
22
11
23
77
66
55
44
33
dword alinhada no endereço 0x1010 contém 0xffee55dd
qword não alinhada no endereço 0x100c contém 0xffee55ddccbbaa00
dword não alinhada no endereço 0x1009 contém 0x00998822
qword alinhada no endereço 0x1008 contém 0xccbbaa0099882211
dword alinhada no endereço 0x1004 contém 0x23776655
word não alinhada no endereço 0x1003 contém 0x5544
word alinhada no endereço 0x1002 contém 0x4433

Instruções (comparação com P16)

P16			Sintaxe AT&T	
ldr[b]	rd, [rn, <imm3>]	rd = memory[rn + imm3]	mov	imm3(%rn), %rd
ldr[b]	rd, [rn, rm]	rd = [rn + rm]	mov	(%rn, %rm), %rd
str[b]	rs, [rn, <imm3>]	memory[rn + imm3] = rd	mov	%rd, imm3(%rn)
str[b]	rs, [rn, rm]	[rn + rm] = rd	mov	%rd, (%rn, %rm)
pop	rd	rd = stack[sp]; sp += 2	pop	%rd
push	rs	sp -= 2; stack[sp] = rs	push	%rs
ldr	rd, label	rd = memory[pc + imm6]	mov	label, %rd
add	rd, rm, rn	rd = rm + rn	add	%rn, %rd
adc	rd, rm, rn	rd = rm + rn + cy	adc	%rn, %rd
sub	rd, rm, rn	rd = rm - rn	sub	%rn, %rd
sbb	rd, rm, rn	rd = rm - rn - cy	sbb	%rn, %rd
add	rd, rm, <imm4>	rd = rm + imm4	add	$immediate, %rd
adc	rd, rm, <imm4>	rd = rm + imm4 + C	adc	$immediate, %rd
sub	rd, rm, <imm4>	rd = rm - imm4	sub	$immediate, %rd
sbb	rd, rm, <imm4>	rd = rm - imm4 - C	sbb	$immediate, %rd
cmp	rn, rm	rn – rm; flags afected	cmp	%rm, %rn
and	rd, rm, rn	rd = rm & rn	and	%rn, %rd
orr	rd, rm, rn	rd = rm | rn	or	%rn, %rd
eor	rd, rm, rn	rd = rm ^ rn	xor	%rn, %rd
rrx	rd, rm	rd = rm >> 1 + c << 16	rcr	$1, %rd
lsl	rd, rn, <imm4>	rd = rn << imm4	shl/sal	$immediate, %rd
lsr	rd, rn, <imm4>	rd = rn >> imm4	shr	$immediate, %rd
asr	rd, rm, <imm4>	rd = rm >> imm4 (signed)	asr	$immediate, %rd
ror	rd, rm	rd = rm >> 1	ror	$immediate, %rd
mov	rd, rs	rd = rs	mov	%rs, %rd
mvn	rd, rs	rd = ~rs	not	%rd
movs	pc, lr		iret	
msr	cpsr, rs	cpsr = rs	popf	
msr	spsr, rs	spsr = rs		
mrs	rd, cpsr	rd = cpsr	setXX; pushf	
mrs	rd, spsr	rd = spsr		
mov	rd, <imm8>	rd = imm8	mov	$immediate, %rd
movt	rd, <imm8>	rd += imm8 << 8		
beq/bzs	label	if (Z == 1) PC += offset	je/jz	label
bne/bzc	label	if (Z == 0) PC += offset	jne/jnz	label
bcs/blo	label	if (C == 1) PC += offset	jc/jb/jnae	label
bcc/bhs	label	if (C == 0) PC += offset	jnc/jae/jnb	label
bge	label	if (S == V) PC += offset	jge/jnl	label
blt	label	if (S != V) PC += offset	jl/jnge	label
b	label	PC += offset	jmp	label
bl	label	LR = PC; PC += offset	call	label

Sintaxe

Sintaxe Intel

O primeiro argumento de uma instrução é o destino do resultado e o primeiro operando.

```
mov   rax, 100
```

Sintaxe AT&T

O segundo argumento de uma instrução é o destino do resultado e o primeiro operando.

```
mov   $100, %rax
```

Instruções

Formato geral das instruções:

```
<instrução>    <operando2>, <operando1/destino>
```

Operandos

Os operandos são valores representados a 8, 16, 32 ou 64 bits.

Os operandos são definidos como:

- valores **imediatos** `mov $1, %rax`
- valores em **registo** `sub %rcx, %rdx`
- ou valores em **memória** `add %rax, (%rbx)`

Há instruções com operandos implícitos:

```
div, mul, inc
```

Um operando fonte pode ser um valor **imediato**, um valor em **registo** ou um valor em **memória**. Um operando destino apenas pode ser um registo ou uma posição de memória.

Não existem instruções que operem dois operandos em memória:

```
sub       (%rdx), (%rbx)
```

Operandos em memória

Um operando em memória é definido por 4 componentes:

```
displacement(base register, index register, scale)
```

Displacement		Base		Index		Scale
		rax, rbx, rcx, rdx	+	rax, rbx, rcx, rdx	*	1
constante a 8 bit com sinal	+	rsp, rbp, rsi, rdi		rbp, rsi, rdi		2
constante a 32 bit com sinal		r8, r9, r10, r11		r8, r9, r10, r11		4
		r12, r13, r14, r15		r12, r13, r14, r15		8

Exemplos de definição de operandos:

Syntaxe	Valor	Designação	Exemplo
$Imm	Imm	Imediato	`mov $20, %r10`
ra	R[ra]	Registo	`mov rax, %r10`
Imm	M[Imm]	Direto ou absoluto	`mov var, %rax`
(ra)	M[R[ra]]	Indireto	`mov (%r10), rax`
Imm(rb)	M[Imm + R[rb]]	Baseado	`mov 2(%r12), rax`
(rb, ri)	M[R[rb] + R[ri]]	Indexado	`mov (%rcx, %r12), rax`
Imm(rb, ri)	M[Imm + R[rb] + R[ri]]	Indexado	`mov count(%rcx, %r12), rax`
(, ri, s)	M[R[ri] * s]	Indexado escalado	`mov (, %r12, 4), rax`
Imm(, ri, s)	M[Imm + R[ri] * s]	Indexado escalado	`mov array(, %r12, 2), rax`
(rb, ri, s)	M[R[rb] + R[ri] * s]	Indexado escalado	`mov (%r14, %r12, 2), rax`
Imm(rb, ri, s)	M[Imm + R[rb] + R[ri] * s]	Indexado escalado	`mov array(%rdx, %r12, 8), rax`

Na definição de operandos em memória algumas componentes podem ser omitidas.

No acesso a variáveis em memória usa-se endereçamento indireto com base em RIP.

```
displacement(%rip)
```

O ISA também dispõe de endereçamento direto com endereço definido a 64 bit.

Exemplo 1

```
      mov   $0x80001000, %rbx
      mov   (%rbx), %rcx
      and   $0x00ff00ff, %rcx
      mov   %rcx, 4(%rbx)
      mov   $0, %rdx
L1:
      mov   12(%rbx, %rdx), %al
      sub   $' ', %al
      mov   %al, 12(%rbx, %rdx)
      inc   %rdx
      cmp   $3, %rdx
      jnz   L1
```

rax	
rbx	
rcx	
rdx	

0x8000100e	'g'
0x8000100d	'f'
0x8000100c	'e'
0x8000100b	'd'
0x8000100a	'c'
0x80001009	'b'
0x80001008	'a'
0x80001007	88
0x80001006	77
0x80001005	66
0x80001004	55
0x80001003	44
0x80001002	33
0x80001001	22
0x80001000	11

Lista das Instruções (sintaxe AT&T)

Instruções para movimento de dados

`mov <origem>, <destino>`

Copia **`origem`** para **`destino`**. A origem pode ser um valor imediato, o conteúdo de um registo ou de uma posição de memória. O destino pode ser um registo ou posição de memória.

`mov $0, %rax`	**`rax`** é afectado com 0
`mov %rbx, %rax`	**`rax`** é afectado com o valor de **`rbx`**
`mov var, %rax`	**`rax`** é afectado com o valor da variável var
`mov (%rbx), %rax`	**`rax`** é afectado com o valor da memória cujo endereço está em **`rbx`**

`movzx / movsx <origem, destino>`

Copia um valor representado a 8, 16 ou 32 bits para um destino representado com maior número de bits. Os bits de maior peso são acrescentados com zero ou com o bit de sinal.

`xchg <operando1>, <operando2>`

São trocados os conteúdos dos registos ou posições de memória entre **`operando1`** e **`operando2`**. Não é possível trocar o conteúdo de duas posições de memória na mesma instrução.

Trocar o conteúdo de duas *quadwords* em memória:

```
mov   (%rbx), %rax
mov   8(%rbx), %rcx
mov   %rcx, (%rbx)
mov   %rax, 8(%rbx)
```

```
mov   (%rbx), %rax
xchg  8(%ebx), %rax
mov   %rax, (%rbx)
```

`cbw / cwd / cdq`

Converte um valor para uma representação com maior número de *bits*, assumindo codificação em código de complementos (operandos implícitos).

`cbw` - **`al`** para **`ax`**; **`cwd`** - **`ax`** para **`eax`**; **`cdq`** - **`eax`** para **`edx`**:**`eax`**

`lea <origem>, <destino>`

Coloca o endereço de **`origem`** em **`destino`**.

`lea var(%rip), %rax`	coloca em **`rax`** o endereço da variável **`var`**
`lea (%rbx, %rdi), %rax`	coloca em **`rax`** o endereço que resulta da soma de **`rbx`** com **`rdi`**

Exemplo 2

`int a, b;`
 	
`a = b;`
	`a:  .int 0`
	`b:  .int 0`
	`mov  b(%rip), %eax`
	`mov  %eax, a(%rip)`

`char c, d, *p;`
 	
`p = &c;`
 	
`*p = d;`
	`c:  .byte 0`
	`d:  .byte 0`
	`p:  .long 0`
	
	`lea  c(%rip), %rbx`
	`mov  %rbx, p(%rip)`
	
	`mov  d(%rip), %al`
	`mov  p(%rip), %rbx`
	`mov  %al, (%rbx)`

`int array[] = {1, 2, 3};`
`int i = 2;`
`int *pi;`
 	
`array[i] = 6023;`
 	
`pi = &array[i];`
	`array: .int  1, 2, 3`
	`i:  .int  2`
	`pi:  .quad 0`
	
	`mov i(%rip), %rcx`
	`lea array(%rip), %rax`
	`mov $6023, (%rax, %rcx, 4)`
	
	`lea array(%rip), %rax`
	`mov i(%rip), %rcx`
	`lea (%rax, %rcx, 4), %rdx`
	`mov %rdx, pi(%rip)`

`push <operando>`

Decrementa o registo **`rsp`** de 8 e copia o **`operando`** para a posição de memória definida por **`rsp`**. O **`operando`** pode ser um valor imediato, um valor em registo ou numa posição de memória.

```
push  %rdx
```
é equivalente a
```
sub $8, %rsp
mov %rdx, (%rsp)
```

Exemplo 3

```
push  %rax
push  $0x800000000a080
push  (%rbx)
```

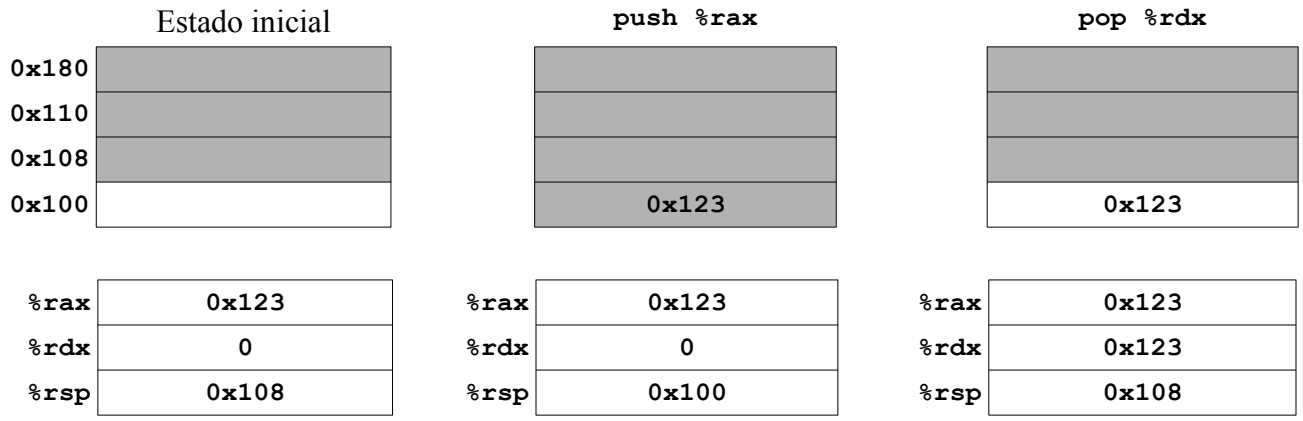
`pop <operando>`

Copia o valor da posição de memória indicada por **`rsp`** para o **`operando`** e incrementa o registo **`rsp`** de 8. O **`operando`** pode ser um registo ou uma posição de memória.

```
pop   %rax
```
é equivalente a
```
mov (%rsp), %rax
add $8, rsp
```

Exemplo 4

```
pop   %rdx
pop   var(%rbx)
```



Instruções aritméticas

add <operando2>, <operando1 / destino>

Adiciona o valor de **operando1** com o de **operando2** e coloca o resultado em **operando1**. O **operando2** pode ser um valor imediato, um registo ou uma posição de memória. O **operando1** pode ser um registo ou uma posição de memória.

`add %rbx, %rax`	afecta **rax** com **rax** + **rbx**.
`add $10, 4(%rbx)`	adiciona 10 a um valor em memória. O endereço do operando em memória é calculado pela adição de **rbx** com **4**

adc <operando2>, <operando1 / destino>

Adiciona o valor de **operando1** como o de **operando2** e com a ***flag carry***. Coloca o resultado em **operando1**. O **operando2** pode ser um valor imediato, um registo ou uma posição de memória. **operando1** pode ser um registo ou uma posição de memória.

Somar valores representados a 128 bits - primeira parcela em **rbx:rax**; segunda parcela em **rdx:rcx** resultado em **rbx:rax.**

```
add   %rcx, %rax
adc   %rdx, %rbx
```

sub <subtraendo>, <diminuendo / destino>

Subtrai **subtraendo** de **diminuendo** e coloca o resultado em **destino** que é coincidente com **diminuendo**. O **subtraendo** pode ser um valor imediato, um registo ou uma posição de memória. O **diminuendo** pode ser um registo ou uma posição de memória.

`sub %rbx, %rax`	afeta **rax** com **rax** – **rbx**.

sbb <subtraendo>, <diminuendo / destino>

Subtrai **subtraendo** e a ***flag carry*** de **diminuendo**. O **subtraendo** pode ser um valor imediato,

um registo ou uma posição de memória. O **destino**/**diminuendo** pode ser um registo ou uma posição de memória. É assumido que a *flag carry* contém a indicação de *borrow* resultante de uma subtração anterior.

Subtrair valores representados a 128 bits - diminuendo em **rbx:rax** subtraendo em **rdx:rcx** resultado em **rbx:rax**.

```
sub   %rcx, %rax
sbb   %rdx, %rbx
```

inc <operando>

Incrementa uma unidade ao **operando**. O **operando** pode ser um registo ou uma posição de memória.

`inc %rax`	afecta **rax** com **rax + 1**

dec <operando>

Decrementa uma unidade ao `operando`. O `operando` pode ser um registo ou uma posição de memória.

`dec %rax`	afecta **rax** com **rax - 1**
`decl (%rbx, %rcx)`	decrementa uma unidade a um valor em memória. O endereço de memória o operando é dado pela adição de **rbx** com **rcx**. O sufixo de **dec** indica a dimensão do operando.

neg <operando>

Nega o valor numérico representado no **operando**. O **operando** pode ser um registo ou uma posição de memória.

Simétrico de um valor inteiro – nega o valor de **rax**

```
neg   rax, rax
```

mul <operando>

Multiplica o argumento da instrução pelo valor de **al**, **ax**, **eax** ou **rax** deixando o resultado em **ax**, **eax, rax** ou **rdx:rax**, respetivamente.

div <operando>

Divide o conteúdo de **ax**, **dx:ax**, **edx**:**eax** ou **rdx:rax** por **operando** deixando os resultados (quociente – resto) em **al** – **ah**, **ax** – **dx**, **eax** – **edx** ou **rax** – **rdx**.

Instruções lógicas

`and <operando2>, <operando1 / destino>`

Executa a operação lógica AND entre os *bits* da mesma posição de ambos os operandos. Coloca o resultado em destino.

`or <operando2>, <operando1 / destino>`

Executa a operação lógica OR entre os bits da mesma posição de ambos os operandos. Coloca o resultado em **`destino`**.

`xor <operando2>, <operando1 / destino>`

Executa a operação lógica EXCLUSIVE OR entre os bits da mesma posição de ambos os operandos. Coloca o resultado em **`destino`**.

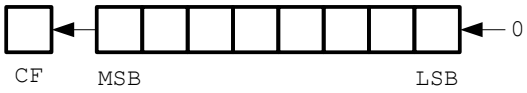
`not <operando>`

Inverte o valor lógico de todos os bits de **`operando`**. O operando pode ser um registo ou uma posição de memória.

Instruções deslocamento (*shift*)

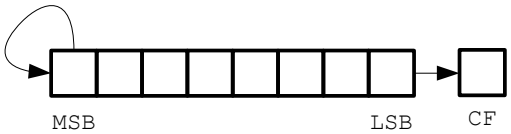
`sal/shl <nbits>, <operando>`

Desloca o conteúdo de **`operando`** para esquerda inserindo zero nos *bits* de menor peso.



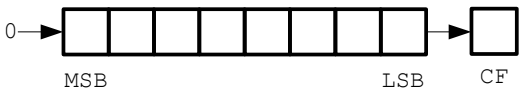
`sar <nbits>, <operando>`

Desloca o conteúdo de **`operando`** para direita propagando o valor do *bit* de maior peso.



`shr <nbits>, <operando>`

Desloca o conteúdo de **`operando`** para direita inserindo zero nos *bits* de maior peso.



shld <nbits>, <operando1>, <operando2>

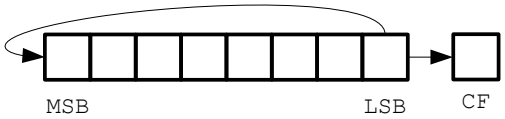
Desloca os conteúdos de **operando1** e **operando2** para esquerda inserindo os bits de maior peso de **operando1** nos de menor peso de **operando2**. Apenas o **operando2** é alterado, o **operando1** mantém-se.

shrd <nbits>, <operando1>, <operando2>

Desloca os conteúdos de **operando1** e **operando2** para direita inserindo os bits de menor peso de **operando1** nos de maior peso de **operando2**. Apenas o **operando2** é alterado, o **operando1** mantém-se.

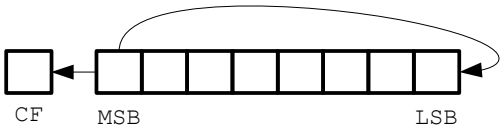
ror <nbits>, <operando>

Desloca o conteúdo de operando para a direita inserindo, em cada passo, o bits de menor peso na posição de maior peso e na carry flag.



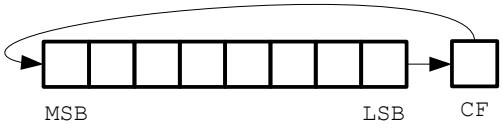
rol <nbits>, <operando>

Desloca o conteúdo de operando para a esquerda inserindo, em cada passo, o bits de maior peso na posição de menor peso e na carry flag.



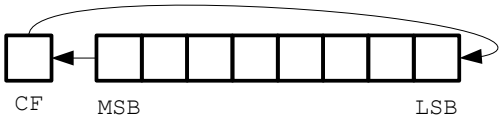
rcr <nbits>, <operando>

Desloca o conteúdo de operando para a direita inserindo, em cada passo, o bits de menor peso na carry flag e o conteúdo de carry flag na posição de maior peso.



rcl <nbits>, <operando>

Desloca o conteúdo de operando para a esquerda inserindo, em cada passo, o bits de maior peso na carry flag e o conteúdo de carry flag na posição de menor peso.



Exemplo 5

Considerando **a** um valor expresso a 64 bit, previamente carregado em RAX.

Deslocar **p** posições para a esquerda	`b = a << p;`	`mov p(%rip), %cl` `shl %cl, %rax`

Afetar o *bit* da posição **p** com zero	`b = a & ~(1 << p);`	`mov p(%rip), %cl`
		`mov $1, %rdx`
		`shl %cl, %rdx`
		`not %rdx`
		`and %rdx, %rax`

Afetar o *bit* da posição **p** com um	`b = a | 1 << p;`	`mov p(%rip), %cl`
		`mov $1, %rdx`
		`shl %cl, %rdx`
		`or %rdx, %rax`

Testar o valor do *bit* da posição **p**	`if (a & (1 << p))`	`mov p(%rip), %cl`
		`mov $1, %rdx`
		`shl %cl, %rdx`
		`test %rdx, %rax`
		`jz label`

Obter o campo de **n** bits a começar na posição **p**.	`return (a >> p) & ~(~0 << n);`
	(n = 64 provoca *overflow*)
		`mov $~0, %rdx`
		`mov n(%rip), %cl`
		`shl %cl, %rdx`
		`not %rdx`
		`mov p(%rip), %cl`
		`shr %cl, %rax`
		`and %rdx, %rax`

Multiplicação por constante	`rax * 13 = (rax * 3) * 4 + rax`	`lea (%rax, %rax, 2), %rdx` `lea (%rax, %rdx, 4), %eax`

Considerando **a** um valor expresso a 128 bit previamente carregado em RDX:RAX .

Deslocar 1 posição para a esquerda	`a <<= 1;`	`shl $1, %rax` `rcl $1, %rbx`
Deslocar 1 posição para a direita	`a >>= 1;`	`shr $1, %rbx` `rcr $1, %rax`

Deslocar **p** posições para a esquerda	`a <<= p;`	`mov p(%rip), %cl`
		`shld %cl, %rax, %rdx`
		`shl $cl, %rax`

Deslocar **N** posições para a direita	`a >>= N;`	`shrd $N, %rbx, %rax` `shr $N, %rbx`

Instruções para controlo da execução

`jmp <endereço>`

Executa o salto para o endereço definido pelo argumento da instrução. O endereço pode ser definido em absoluto, através de um valor imediato, pelo conteúdo de um registo ou de uma posição de memória, ou pode ser definido de forma relativa, através de um valor imediato a adicionar ao valor corrente de RIP.

`jmp .L2`	.L2 representa um valor constante
`jmp *%rax`	salta para o endereço que está em RAX.
`jmp *(rbx)`	salta para o endereço que em memória no endereço indicado por RBX.

`call <endereço>`

Guarda RIP no *stack* (**`push %rip`**). Nessa altura RIP contém o endereço da instrução seguinte. Depois executa o salto para o endereço indicado (**`jmp <endereço>`**). O endereço de salto pode ser um valor imediato, o conteúdo de um registo ou de uma posição de memória.

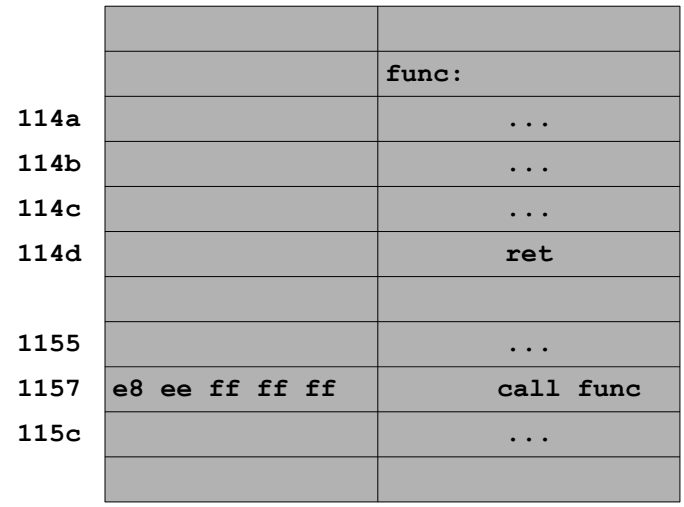
`call <endereço>` é equivalente a

```
push %rip
jmp <endereço>
```

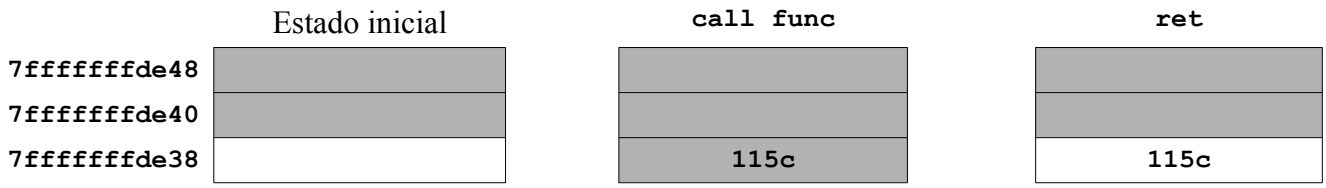
`ret`

Coloca o valor que está no topo do *stack* em RIP (**`pop rip`**). Sendo o valor que está no topo do *stack* o endereço empilhado pela última instrução **`call`**, esta instrução provoca o retorno do processamento à instrução a seguir a essa instrução **`call`**.

Memória de código



Stack



Registos

%rsp	7fffffffde40	%rsp	7fffffffde38	%rsp	7fffffffde40
%rip	1157	%rip	114a	%rip	115c

`enter`

Na sua forma básica, é equivalente a:

```
push  %rbp
mov   %rsp, %rbp
```

É usada no inicio das funções para preparar a *stack frame*.

`leave`

É equivalente a:

```
mov   %rbp, %rsp
pop   %rbp
```

É usada no fim das funções para restaurar o *stack pointer* e o *base pointer*.

`loop`

(em falta)

`loope`

(em falta)

`loopne`

(em falta)

`jcxz`

(em falta)

Exemplo 6

my_sub.c

```
int a = 10, b = 20, c;

int sub(int a, int b) {
      return a - b;
}

void subp(int *op1, int *op2, int *res) {
      *res = sub(*op1, *op2);
}

int main() {
      subp(&a, &b, &c);
}
```

```
$ gcc -Og -o my_sub my_sub.c

$ ls -l

drwxrwxr-x 2 ezequiel ezequiel  4096 mar 30 12:13 .
drwxrwxr-x 4 ezequiel ezequiel  4096 mar 30 11:47 ..
-rwxrwxr-x 1 ezequiel ezequiel 16496 mar 30 12:06 my_sub
-rw-rw-r-- 1 ezequiel ezequiel   171 mar 30 12:06 my_sub.c

$ objdump -d my_sub

000000000000114a <sub>:
    114a:   89 f8                   mov    %edi,%eax
    114c:   29 f0                   sub    %esi,%eax
    114e:   c3                      retq

000000000000114f <subp>:
    114f:   53                      push   %rbx
    1150:   48 89 d3                mov    %rdx,%rbx
    1153:   8b 36                   mov    (%rsi),%esi
    1155:   8b 3f                   mov    (%rdi),%edi
    1157:   e8 ee ff ff ff          callq  114a <sub>
    115c:   89 03                   mov    %eax,(%rbx)
    115e:   5b                      pop    %rbx
    115f:   c3                      retq

0000000000001160 <main>:
    1160:   48 8d 15 b5 2e 00 00    lea    0x2eb5(%rip),%rdx        # 401c <c>
    1167:   48 8d 35 a2 2e 00 00    lea    0x2ea2(%rip),%rsi        # 4010 <b>
    116e:   48 8d 3d 9f 2e 00 00    lea    0x2e9f(%rip),%rdi        # 4014 <a>
    1175:   e8 d5 ff ff ff          callq  114f <subp>
    117a:   b8 00 00 00 00          mov    $0x0,%eax
    117f:   c3                      retq
```

Instruções sobre blocos de memória

`movs`

Move o conteúdo da memória indicada por RSI para a memória indicada por RDI.

`cmps`

Compara o conteúdo da memória indicada por RSI com o conteúdo da memória indicada por RDI.

`scas`

Compara o conteúdo de RAX com conteúdo da memória apontada por RDI.

`ldos`

(em falta)

`stos`

(em falta)

`cld`

(em falta)

`std`

(em falta)

`rep repe repz repne repnz`

Prefixos de repetição. Provocam a repetição da operação da instrução seguinte enquanto o registo RCX for maior que zero e se verificar a condição indicada.

Instruções relacionadas com as *flags*

`stc`

(em falta)

`clc`

(em falta)

`cmc`

(em falta)

`lahf`

(em falta)

`sahf`

(em falta)

`pushf / pushfd`

(em falta)

`popf / popfd`

(em falta)

`test <operando2>, <operando1>`

Executa a operação lógica E entre os bits das mesmas posições dos operandos. Não aproveita o resultado mas afeta as *flags*.

```
test  $0x80, %eax
jz    label
```

Testa o valor do *bit* da posição 7 – executa EAX & 0x80 e afeta as *flags* conforme o resultado.

`cmp <subtraendo>, <diminuendo>`

Subtrai **`subtraendo`** ao **`diminuendo`**. O resultado não é aproveitado, apenas as *flags* são afetadas. O **`subtraendo`** pode ser um valor imediato, um registo ou uma posição de memória. O **`diminuendo`** pode ser um registo ou uma posição de memória.

`cmp  %ebx, %eax` realiza EAX – EBX e afeta as *flags*

`j<cond> <endereço>`

Se a condição for verdadeira, muda o endereço de execução para o endereço definido pelo argumento. A verificação da condição é realizada sobre o estado das *flags*. O endereço de salto é definido de forma relativa, através de um valor imediato a adicionar ao valor corrente de RIP.

Encarando os operandos como números naturais

Mnemonic	Condition Tested	"Jump If..."

JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even

Encarando os operandos como números relativos

Mnemonic	**Condition Tested**	**"Jump If..."**
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (positive, including 0)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

Linguagem assembly

Sintaxe GNU

Um programa é composto por uma sequência de *statements*. Um *statemant* começa opcionalmente com uma *label* ao que se segue, também opcionalmente, uma instrução ou diretiva, pode ter um comentário e acaba com o fim da linha ou com o caráter separador ';' (ponto e vírgula).

```
label:
      .directive    followed by something

another_label:                # This is an empty statement.
      instruction   operand_1, operand_2, ...
```

label	Símbolo seguido do caráter '**:**' (dois pontos). Define o endereço do elemento seguinte – instrução ou variável. Pode ser composto por letras, dígitos e os caracteres '.' (ponto) e '_' (sublinhado). As *labels* locais são definidas por um dígito na forma N: com N de 0 a 9 e referidos por Nb ou Nf.
instruction	Qualquer instrução de processador.
comment	O caracter # indica comentário até ao fim da linha. Podem também ser inseridos comentários como em C com **/*** no início e ***/** no fim.

Sintaxe Intel

```
mov   rax, rbx
call  [rdi]
shr   rcx, 2
mov   [rbp + rdi * 4], rax
```

Sintaxe geral para operandos em memória – **displacement[*base* + *index* * *scale*]**

Sintaxe AT&T

```
movl  %ebx, %eax
call  *(%edi)
shrl  $2, %ecx
movl %eax, (%ebp, %edi, 8)
```

Sintaxe geral para operandos em memória – **displacement(*base*, *index*, *scale*)**

Sufixos - as mnemónicas podem ter o sufixo **b**, **w**, **l** ou **q** para indicar, respetivamente, que o operando tem a dimensão de 8, 16, 32 ou 64 bits.

Expressões

Uma expressão é equivalente a um valor numérico. Esse valor é determinado diretamente, por substituição de um símbolo ou por aplicação de operações a outras expressões.

Uma expressão pode aparecer em qualquer instrução onde seja esperada uma constante.

Os valores numéricos podem ser representados em decimal, hexadecimal (0xddd), octal (0ddd),

binário (0bdddd) ou caracteres entre ' ' (plicas) . Exemplos: 34, 0x3f, 034, 0b0101, 'K'

Operadores unários

- `-` Negação em complemento para 2.
- `~` Negação bit a bit.

Operadores binários

`*`	Multiplicação	`|`	Disjunção bit a bit	`+`	adição
`/`	Divisão inteira	`&`	Conjunção bit a bit	`-`	subtracção
`%`	Resto da divisão inteira	`^`	Disjunção exclusiva bit a bit		
`<<`	Deslocar para a esquerda	`>>`	Deslocar para a direita		

Directivas

Resumo das principais diretivas.

`.align expression`	Insere bytes a zero até um endereço múltiplo do parâmetro.
`.ascii "string"`	Insere os caracteres que compõem a string.
`.asciz "string"`	Insere os caracteres que compõem a string com terminação a zero.
`.byte expression`	Insere o valor especificado representado a 8 bits (1 byte).

`.2byte expression`
`.word expression`
`.hword expression`
`.short expression`
	Insere o valor especificado representado a 16 bits (2bytes).

`.4byte expression`
`.long expression`
`.int expression`
	Insere o valor especificado representado a 32 bits (4bytes).

`.8byte expression` `.quad expression`	Insere o valor especificado representado a 64 bits (8bytes).
`.octa expression`	Insere o valor especificado representado a 128 bits (16bytes).
`.space size, fill` `.skip size, fill`	Insere size bytes com o valor fill.
`.zero size`	Insere um bloco com dimensão **size** preenchido com zero

`.text`
`.data`
`.bss`
`.rodata`
	Passa a inserir na secção assinalada.
	A secção **.text** é para o código das instruções; a secção **.data** para as variáveis iniciadas; a secção **.bss** para as variáveis não iniciadas e a secção **.rodata** para os dados constantes.

`.section name`	Passa a inserir na secção com o nome name.
`.global symbol`	Declara symbol visível para os outros módulos.
`.extern symbol`	Declara que symbol é definido noutro módulo.
`.include "file"`	Insere o conteúdo de file na posição desta directiva.

`.if expression`
`.else if expression`
`.else`
`.endif`
	Para compilação condicional.

`.equ symbol, expression` `.set symbol, expression`	Define symbol com o valor de expression.
`.err`	Imprime uma mensagem de erro e termina a compilação.

Sufixos

O sufixo serve para definir a dimensão da palavra de dados processada pela instrução.

Sempre que um dos operandos da instrução é um registo, o sufixo é dispensável. O *assembler* infere a dimensão da palavra pela designação do registo.

Instrução	Dimensão dos dados
`mov $8, %al`	8 bits
`mov (%rbx), %ax`	16 bits
`add %r8d, %r10d`	32 bits
`push $0`	64 bits – as instruções push e pop manipulam sempre valores a 64 bits

Quando não é possível ao *assembler* determinar a dimensão da palavra através dos argumentos da instrução, o uso do sufixo é necessário.

Instrução	Dimensão dos dados
`incb (%rbx)`	8 bits
`movw $7, (%rbx)`	16 bits
`shrl %cl, mask(%rip)`	32 bits
`decq 16(%rbp, %rio, 8)`	64 bits

Há instruções com sufixos sintaticamente obrigatórios. É o caso das instruções de extensão de bits.

Instrução	Dimensão dos dados
`movsbq %dl, %rbx`	Extensão com o valor do bit de sinal de 8 para 64 bits
`movzbq (%rbx),%r10`	Extensão com zero 8 para 64 bits
`movswl mask(%rip), %eax`	Extensão com o valor do bit de sinal de 16 para 32 bits

Declaração C	Designação	Sufixo	Dimensão	Alinhamento

	Intel	GAS	(Bytes)	
char	byte	b	1	1
short	word	w	2	2
int	double word	l	4	4
unsigned int	double word	l	4	4
long int	quad word	q	8	8
unsigned long int	quad word	q	8	8
float	single precision	s	4	4
double	double precision	d	8	8
pointer (char *)	quad word	q	8	8
struct union			A dimensão de um tipo composto é múltipla do seu alinhamento.	O alinhamento de um tipo composto é igual ao maior alinhamento interno.

Ferramentas

GCC

Algumas opções **gcc** relacionadas com a geração de código.

`-S`	Geração de código em linguagem *assembly*
`-masm=intel`	Acrescentar à opção anterior para gerar sintaxe Intel
`-Og`	Geração de código com otimização legível
`-fno-stack-protector`	Suprimir a geração de código de verificação de integridade do *stack*
`-fno-stack-clash-protection`	Suprimir a geração de código de mitigação da vulnerabilidade s*tack clash*

GNU AS

O GNU a*ssembler* usa sintaxe AT&T por omissão. Para usar sintaxe Intel, deve incluir-se no ficheiro fonte a diretiva:

```
.intel_syntax noprefix
```

Invocação do **`as`** (GNU *assembler)* na linha de comando (shell).

```
$ as [-a[=file]] [-defsym sym=val] [--gstabs] [-I dir] [-o objfile] srcfile
```

`-a`	Gerar ficheiro listagem.
`=file`	Criar a listagem no ficheiro indicado.
`-defsym`	Definir um símbolo.
`--gstabs`	Incluir informação para *debugging* no ficheiro de saída (*object relocation file*).
`-I`	Definir diretoria de pesquisa para ficheiros de *include*.
`-o`	Definir o nome do ficheiro de saída.

`srcfile` O ficheiro com a fonte do programa.

Objdump

Visualizar o conteúdo de um ficheiro executável em linguagem em *assembly*.

```
$ objdump -d <program>
```

Debugger

O Insight pode ser configurado para apresentar a sintaxe Intel ou AT&T
Preferences/Source.../Disassembly flavor

Linguagem C

Estruturas de controlo (padrões de código)

if

```
if (expression)
    statement1;
else
    statement2;
```

```
                    . . .                   código de avaliação de expression
                    j<cond> 1f
                    . . .                   código de statement1
                    jmp 2f
                1:  . . .                   código de statement2
                2:                          instrução a seguir ao if
```

while

```
while (expression)
    statement;
```

```
                2:  . . .                   código de avaliação de expression
                    j<cond> 1f
                    . . .                   código de statement
                    jmp 2b
                1:                          instrução a seguir ao while
```

do while

```
do {
    statement;
} while (expression);
```

```
                1:  . . .                   código de statement
                    . . .                   código de avaliação de expression
                    j<cond> 1b
                                            instrução a seguir ao do while
```

for (variante 1)

```
for (exp1; exp2; exp3)
    statement;
```

```
                    . . .                   código da expressão exp1
                2:  . . .                   código de avaliação de exp2
                    j<cond> 1f
                    . . .                   código de statement
                    . . .                   código de exp3
                    jmp 2b
                1:                          instrução a seguir ao for
```

for (variante 2)

```
for (exp1; exp2; exp3)
    statement;
```

```
        . . .          código da expressão exp1
        jmp 1f
2:      . . .          código de statement
        . . .          código de exp3
1:      . . .          código de avaliação de exp2
        j<cond> 2b
                       instrução a seguir ao for
```

switch

```
switch (expression) {
    case value1:
        statement1;
    case value2:
        statement2;
};
```

```
        ...                    código de cálculo de expression
        cmp  $value1, %edi
        je   L1
        cmp  $value2, %edi
        je   L2
L1:     . . .                  código de statement1
L2:     . . .                  código de statement2
```

Funções

A chamada a uma função é feita com a instrução **`call`**. O argumento desta instrução é o endereço da função, que pode ser especificado nas seguintes formas:

`call` ***`label`***	salto relativo; é calculada a distância até à *label*
`call *`* ***`operand`***	salto absoluto; o endereço é o conteúdo de *operand*, que pode ser um registo ou conteúdo de memória.

A instrução **`call`** empilha o endereço da próxima instrução no topo do *stack* – o endereço de retorno – e executa um salto para o endereço de início da função chamada (***callee***). Para retornar à função chamadora (***caller***), a função chamada executa, em último lugar, a instrução **`ret`**. Esta instrução desempilha para o registo RIP, o endereço empilhado pelo **`call`** anterior, como se fosse um **`pop rip`**, provocando o regresso à função chamadora.

Passagem de argumentos e retorno de valores

Os argumentos são passados nos registos **`rdi`**, **`rsi`**, **`rdx`**, **`rcx`**, **`r8`** e **`r9`**, por esta ordem, e até ao sexto argumento. Os argumentos seguintes são passados em *stack*.

O retorno é feito em **`al`**, **`ax`**, **`eax`**, **`rax`** ou em **`rdx:rax`**, conforme o tipo. No caso de retorno de uma **`struct`** com dimensão superior a 128 *bits*, o chamador passa um argumento extra, que é o endereço de uma zona de memória, reservada pelo chamador, onde a função vai depositar o resultado.

Exemplo 7

```
int sum(int a, int b) {
    return a + b;
}
```

```
sum:
      mov   %edi, %eax
      add   %esi, %eax
      ret
```

```
...
sum(3, sum(4, 5));
...
```

```
      ...
      mov   $5, %esi
      mov   $4, %edi
      call  sum
      mov   %eax, %esi
      mov   $3, %edi
      call  sum
      ...
```

Exemplo 8

```
size_t my_strlen(const char *str)
{
      site_t i;
      for (i = 0; str[i]; ++i)
          ;
      return i;
}
```

my_strlen.c

```
my_strlen:
      mov   $0, %rax
my_strlen_for:
      cmpb  $0, (%rdi, %rax)
      je    my_strlen_return
      inc   %rax
      jmp   my_strlen_for
my_strlen_return:
      ret
```

my_strlen_asm.s

Exemplo 9

```
void my_strcpy(char *dst, const char *src)
{
    while(*dst++ = *src++)
        ;
}
```

my_strcpy.c

```
my_strcpy:
      mov   (%rsi), %al
      mov   %al, (%rdi)
      inc   %rdi
      inc   %rsi
      cmp   $0, %al
      jnz   my_strcpy
      ret
```

my_strcpy_asm.s

Acesso a dados

Em linguagem C quando se define uma variável global (**`char a`** ou **`char *cp`**) estabelece-se um símbolo (**`a`** ou **`cp`**) que representa o conteúdo dessa variável.

Em linguagem *assembly* esse símbolo corresponde a uma *label* que define o endereço dessa variável em memória. O acesso ao conteúdo destas variáveis é realizado com endereçamento relativo ao RIP.

```
mov    a(%rip), %al
```

`char a, b;`	`a:  .byte  0` `b:  .byte  0`

`int  i, j;`	`i:  .int  0` `j:  .int  0`

`char *cp;`
`int *ip, *iq;`
	`cp:  .quad  0`
	`ip:  .quad  0`
	`iq:  .quad  0`

Em linguagem C quando se define um *array* (**`int ia[10];`**) estabelece-se um símbolo (**`ia`**) que representa o ponteiro para o primeiro elemento do *array*.

Em *assembly* este símbolo corresponde a uma *label* que define o endereço inicial de uma zona de memória onde é alojado o *array*. Para aceder aos elementos do *array* começa-se por carregar num registo o endereço atual dessa *label*

```
lea    ai(%rip), %rax
```

em seguida utiliza-se o registo onde se carregou esse endereço – RAX – como base de formação do endereço do elemento a aceder, em que RCX é o índice do *array* e o valor 4 o fator de escala.

```
mov    (%rax, %rcx, 4), %edx
```

`char ca[10];` `int ia[10];`	`ca:  .space  10, 0` `ia:  .space  10 * 4, 0`

Operações com ponteiros

`cp = &a;`	`lea  a(%rip), %rax` `mov  %rax, cp(%rip)`
`cp++;`	`incq  cp(%rip)`
`ip++;`	`addq  $4, ip(%rip)`

`ip = ip + i`	`mov  i(%rip), %eax`
	`shl  $2, %rax`
	`add  %rax, ip(%rip)`

`j = ip - iq;`	`mov  iq(%rip), %rax`
	`sub  ip(%rip), %rax`
	`shr  $2, %rax`
	`mov  %eax, j(%rip)`

`b = *cp;`	`mov  cp(%rip), %rax`
	`mov  (%rax), %al`
	`mov  %al, b(%rip)`

`i = *ip;`	`mov  ip(%rip), %rax`
	`mov  (%rax), %eax`
	`mov  %eax, i(%rip)`

`j = *(ip + i);`
ou equivalente
`j = ip[i];`
	`mov  ip(%rip), %rax`
	`mov  i(%rip), %esi`
	`mov  (%rax, %rsi, 4), %eax`
	`mov  %eax, j(%rip)`

Programas de exemplo

Programa 1

```
long getbits(long x, int p, int n) {
    return (x >> p) & ((1L << n) - 1);
}
```

```
    .text
    .global getbits
getbits:
```

```
n = 64 provoca overflow
```

getbits.c

```
    mov     $1, %rax
    mov     %dl, %cl
    shl     %cl, %rax
    dec     %rax
    mov     %sil, %cl
    shr     %cl, %rdi
    and     %rdi, %rax
    ret
```

getbits_asm.s

```
long x = 30;
long y = 0;

long getbits(long x, int p, int n);

int main() {
      y = getbits(x, 4, 3);
}
```

main.c

```
    .data
x:
    .quad   30

    .bss
y:
    .zero   8

    .text
    .global main
main:
    sub     $8, %rsp
    mov     $3, %edx
    mov     $4, %esi
    mov     x(%rip), %rdi
    call    getbits
    mov     %rax, y(%rip)
    mov     $0, %eax
    add     $8, %rsp
    ret
```

main.s

```
$ gcc -Og -g main.c getbits.s -o main

$ insight main
```

Programa 2

```
size_t i = 0;

char message[] = "abcdef";

int main() {
      i = my_strlen(message);
      return i;
}
```

main.c

```
    .data
message:
    .string "abcdef"

    .bss
i:
    .zero     8

    .text
    .global main
main:
    lea     message(%rip), %rdi
    call    my_strlen
    mov     %rax, i(%rip)
    ret
```

main.s

```
$ gcc -Og -g main.c my_strlen.s -o main

$ insight main
```

Programa 3

```
char dst[100];
char *src = "abcdefghijk";

void my_strcpy(char*, char*);

int main() {
    my_strcpy(dst, src);
}
```

main.c

```
    .bss
dst:
    .zero   100

    .rodata
.LC0:
    .string "abcdefghijk"

    .data
src:
    .quad   .LC0

    .text
    .global main
main:
    mov     src(%rip), %rsi
    lea     dst(%rip), %rdi
    call    my_strcpy
    mov     $0, %eax
    ret
```

main.s

```
$ gcc -Og -g main.c my_strcpy.s -o main

$ insight main
```

Programa 4

```
int sub(int a, int b) {
    return a - b;
}
```

sub.c

```
    .text
    .global sub
sub:
    mov     %edi, %eax
    sub     %esi, %eax
    ret
```

sub_asm.s

```
void subp(int *op1, int *op2, int *res) {
    *res = sub(*op1, *op2);
}
```

subp.c

```
    .text
    .global subp
subp:
    pushq   %rbx
    movq    %rdx, %rbx
    movl    (%rsi), %esi
    movl    (%rdi), %edi
    call    sub
    movl    %eax, (%rbx)
    popq    %rbx
    ret
```

subp_asm.s

```
int a = 10, b = 20, c;




int main() {
    subp(&a, &b, &c);
}
```

main.c

```
    .data
a:
    .int   10
b:
    .int   20

    .bss
c:
    .zero  4

    .text
    .global main
main:
    lea     c(%rip), %rdx
    lea     b(%rip), %rsi
    lea     a(%rip), %rdi
    call    subp
    mov     $0, %eax
    ret
```

main_asm.s

```
$ gcc -g gcc main_asm.s subp_asm.s sub_asm.s -o main

$ insight main
```

Programa 5

Procurar o maior elemento num *array* de valores inteiros.

```
int find_bigger(int array[],
                size_t array_size) {
    int bigger = array[0];
    for (size_t i = 1; i < array_size; ++i)
        if (array[i] > bigger)
            bigger = array[i];
    return bigger;
}
```

find_bigger.c

```
    .text
    .global find_bigger
find_bigger:
    mov     (%rdi), %eax
    mov     $1, %rdx
for:
    cmp     %rsi, %rdx
    jnb     for_end
    mov     (%rdi,%rdx,4), %ecx
    cmp     %eax, %ecx
    jle     if_end
    mov     %ecx, %eax
if_end:
    add     $1, %rdx
    jmp     for
for_end:
    ret
```

find_bigger_asm.s

```
int find_bigger(int array[], size_t array_size);

int a[] = { 10, 40, 30, 5};

int main() {
```

```
    .data
a:
    .int 10, 40, 30, 5

    .text
```

```
      int b = find_bigger(a, 4);
}
```

main.c

```
    .global main
main:
    movl    $4, %esi
    lea     a(%rip), %rdi
    call    find_bigger
    mov     $0, %eax
    ret
```

main_asm.s

```
$ gcc -Og -g main.c find_bigger_asm.s -o main

$ insight main
```

Programa 6

Procurar a pessoa mais alta num *array* de **`struct person`**. Retornar o nome da pessoa encontrada.

```
typedef struct person {
    char name[20];
    int age;
    int weight;
    int height;
} Person;
```

```
Person *get_taller(Person *people, size_t n_people) {
    int taller = 0;
    for (int i = 1; i < n_people; ++i)
        if (people[i].height > people[taller].height)
            taller = i;
    return &people[taller];
}
```

`get_taller.c`

```
      .global     get_taller
get_taller:
      movq  $0, %rax          #     size_t taller = 0;
      movq  $1, %rdx          #     for (size_t i = 1; ...
      jmp   for_cond
for:
      addq  $1, %rdx          #     for (... ; ... ; ++i)
for_cond:                     #     for ( ...; i < n_people; ...)
      cmpq  %rsi, %rdx
      jnb   for_end
      movq  %rdx, %r8
      salq  $5, %r8           #     r8 = i * sizeof people[0]
      movq  %rax, %rcx
      salq  $5, %rcx          #     rcx = taller * sizeof people[0]
      movl  28(%rdi, %rcx), %ecx    #     ecx = people[taller].height
      cmpl  %ecx, 28(%rdi, %r8)     #     if (people[i].height > ecx)
      jle   for
      movq  %rdx, %rax              #     taller = i;
      jmp   for
for_end:
      salq  $5, %rax                # rax = taller * sizeof people[0]
      addq  %rdi, %rax              # rax = &people[taller]
      ret
```

`get_taller_asm.s`

```
Person people[] = {
      {"Ana", 30, 70, 177},
      {"Manuel", 30, 70, 187}
};

int main() {
    char *taller_name = get_taller(
                people, ARRAY_SIZE(people))->name;
}
```

```
    .data
people:
    .string "Ana"
    .zero   16
    .int    30, 70, 77
    .string "Manuel"
    .zero   13
    .long   30, 70, 187
main:
      subq  $8, %rsp
      movl  $2, %esi
      leaq  people(%rip), %rdi
      call  get_taller
      movl  $0, %eax
      addq  $8, %rsp
```

	`ret`

`main.c` **`main_asm.s`**

Programa 7

Considerar um *array* de ícones em que cada ícone é representado por um *array* bidimensional de pixeis, com dimensões **dx** e **dy**.

A função **gey_pixel** procura num *array* de ícones o ícone com identificador **id** e retornar a cor do pixel de coordenadas **x** e **y**.

```
typedef struct {
    int id;
    unsigned dx;
    unsigned dy;
    unsigned *bitmap;
} Icon;

unsigned get_pixel(Icon **icons, int id, unsigned int x, unsigned int y) {
    for (Icon *icon = *icons; icon != NULL; icon = *++icons)
        if (icon->id == id) {
            unsigned int idx = icon->dx * y + x;
            return (icon->bitmap[idx >> 5] >> (idx & 0x1f)) & 1;
        }
    return -1;
}
```

get_pixel.c

```
get_pixel:
      movq  (%rdi), %rax
for:
      testq %rax, %rax
      je    for_end
      cmpl  %esi, (%rax)
      je    if_then
      addq  $8, %rdi
      movq  (%rdi), %rax
      jmp   for
if_then:
      imull 4(%rax), %ecx
      addl  %edx, %ecx
      movq  16(%rax), %rdx
      movl  %ecx, %eax
      shrl  $5, %eax
      movl  %eax, %eax
      movl  (%rdx,%rax,4), %eax
      shrl  %cl, %eax
      andl  $1, %eax
      ret
for_end:
      movl  $-1, %eax
      ret
```

get_pixel_asm.s

```
unsigned image1[] = {0xff, ..., 0xff};

unsigned image2[] = {0xff, ..., 0xff};
```

```
    .data
icons:
    .quad   icon1
    .quad   icon2
```

```
Icon icon1 = {33, 64, 64, image1};
Icon icon2 = {44, 64, 64, image2};

Icon *icons[] = {&icon1, &icon2};

int main() {
      get_pixel(icons, 33, 20, 20);
}
```

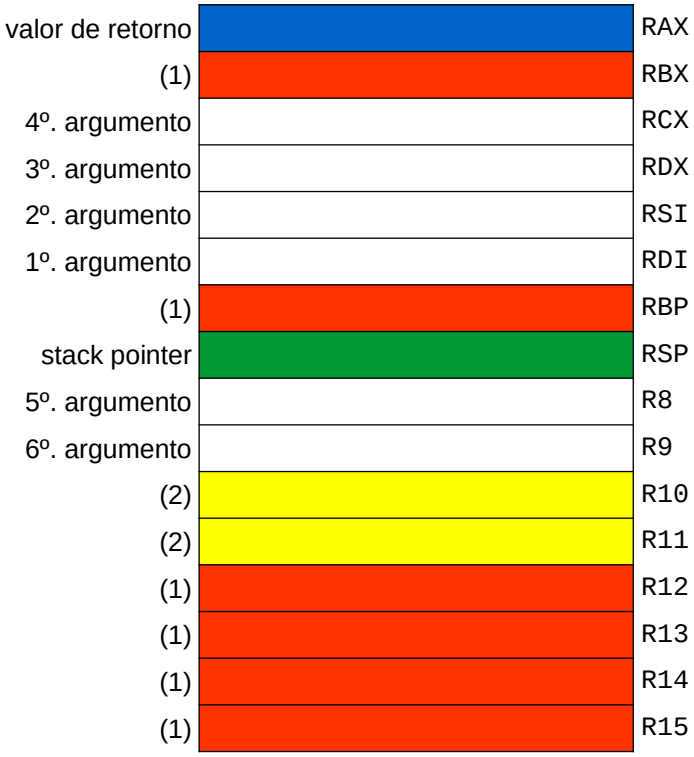
main.c

```
icon2:
    .long   44
    .long   64
    .long   64
    .zero   4
    .quad   image2
icon1:
    .long   33
    .long   64
    .long   64
    .zero   4
    .quad   image1
image2:
    .long   0xff
     ...
    .long   0xff
image1:
    .long   0xff
    ...
    .long   0xff

    .text
    .global main
main:
    movl    $20, %ecx
    movl    $20, %edx
    movl    $33, %esi
    leaq    icons(%rip), %rdi
    call    get_pixel
    movl    $0, %eax
    ret
```

main_asm.s

Critérios de utilização dos registos



(1) – salvo pelo chamado (*calee saved*)
(2) – salvo pelo chamador (*caller saved*)

Os registos **RAX**, **RCX**, **RDX**, **RSI**, **RDI**, **R8, R9, R10 e R11** podem ser modificados pela função chamada, os registos **RBX**, **RBP**, **R12**, **R13**, **R14** e **R15**, se forem utilizados, devem ser preservados.

A cadeia de chamadas a funções num programa pode ser visualizada como uma árvore em que a função **main** se situa na posição da raiz, as funções que são chamadas e que também chamam outras funções, situam-se nas posições dos ramos e as funções que apenas são chamadas situam-se nas posições das folhas.

Para efeitos de escolha dos registos a utilizar interessa classificar as funções como "funções chamadas" ou como "funções chamadoras". As funções folha são funções chamadas, as restantes são funções chamadoras.

Função chamada (folha)

- Deve-se operar os argumentos diretamente no registos que os transportam.
- Deve-se preferir utilizar os registos *caller saved.*
- Se tiver que se utilizar os registos *calee saved* deve-se assegurar à saída da função o mesmo conteúdo que tinham à entrada.

Exemplo 10

Todos os exemplos anteriores de funções de acesso a dados são casos de funções folha.

Função chamadora (ramo)

- Reutiliza os registos de parâmetros na chamada a outras funções.
- Deve-se salvar os argumentos recebidos em registos *calee saved* ou em *stack*.
- Deve-se alojar variáveis locais em registos *calee saved* ou em *stack*.
- Se se optar por utilizar registos *caller saved* ou manter os argumentos recebidos no registos originais deve-se salvar esses registos antes de proceder à chamada de outra função.

Exemplo 11

```
void sort(int array[], int dim) {
    for (int i = 0; i < dim - 1; ++i)
        for (int j = 0; j < dim - 1 - i; ++j)
            if (array[j] > array[j + 1])
                swap(&array[j], &array[j + 1]);
}
```

sort_int.c

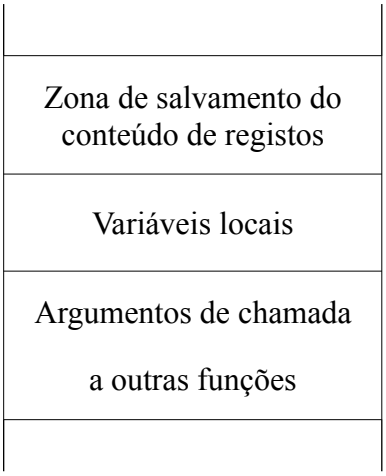
```
    .text
    .global sort
 sort:
        push   %r14
        push   %r12
        push   %rbp
        push   %rbx
        mov    %rdi,%rbp            /*  array   */
        mov    %esi,%r14d           /*  dim  */
        dec    %r14d                /*  dim - 1  */
        mov    $0x0,%r12d           /*  i = 0  */
        jmp    sort_for1_cond
sort_for1:
        mov    $0x0, %ebx           /*  j = 0  */
        jmp    sort_for2_cond
sort_for2:
        movslq %ebx,%rax
        lea    0x0(%rbp,%rax,4),%rdi
        lea    0x4(%rbp,%rax,4),%rsi
        mov    (%rsi),%eax
        cmp    %eax,(%rdi)           /* if (array[j] > array[j + 1])  */
        jle    sort_if_end
        callq  swap
sort_if_end:
        inc    %ebx                  /* ++j */
sort_for2_cond:
        mov    %r14d, %eax           /* j < dim - 1 - i  */
        sub    %r12d, %eax
        cmp    %ebx, %eax
        jg     sort_for2
        inc    %r12d                 /* ++i  */
sort_for1_cond:
        cmp    %r12d,%r14d           /* i < dim - 1  */
        jg     sort_for1
 sort_for1_end:
        pop    %rbx
        pop    %rbp
        pop    %r12
        pop    %r14
        ret
```

sort_int_asm.s

Organização da *stack frame*

Na *stack frame* alojam-se os conteúdos dos registos a preservar, as variáveis locais e argumentos de

chamada a outras funções.



Variáveis locais em stack

As variáveis locais são alojadas em *stack* se:

- a sua quantidade excede o número de registos disponíveis;
- a sua dimensão não permite o alojamento em registo – é o caso dos *arrays;*
- é necessário aceder a essas variáveis através de ponteiros.

Exemplo 12

Acesso a variáveis locais **a**, **b** e **c** através de ponteiros.

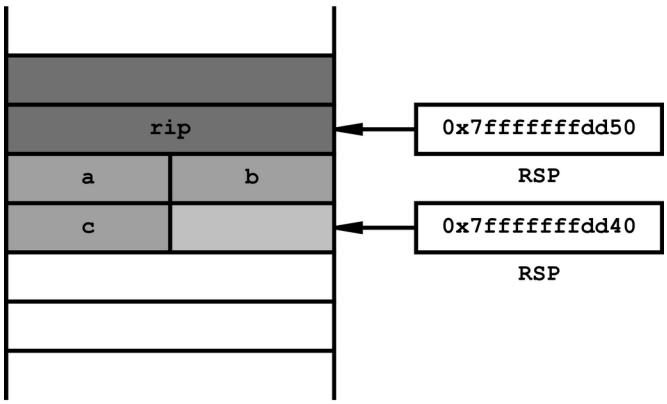
```
void addp(int *, int *, int *);

int main() {
      int a = 10, b = 20, c;
      addp(&a, &b, &c);
}
```

main.c

```
1  main:
2        sub   $16, %rsp
3        movl  $10, 12(%rsp)
4        movl  $20, 8(%rsp)
5        lea   4(%rsp), %rdx
6        lea   8(%rsp), %rsi
7        lea   12(%rsp), %rdi
8        call  addp
9        mov   $0, %eax
10       add   $16, %rsp
11       ret
```

main.s




```
$ gcc -S -Og -c -fno-stack-protector main.c
```

À entrada da função o registo RSP apresenta-se na posição de memória **`0x7fffffffdd50`**.
A instrução **`sub $16, %rsp`** ao subtrair 16 a RSP, reserva espaço para alojar as variáveis **`a`**, **`b`** e **`c`**. Para alojar três variáveis do tipo **`int`** são necessários 12 *bytes*. Como RSP deve estar sempre alinhado num endereço múltiplo de 8, são subtraídas 16 posições de memória.
As posições de memória entre **`0x7fffffffdd40`** e **`0x7fffffffdd43`** não são utilizadas.
Nas linhas 3 e 4 procede-se à inicialização das variáveis **`a`** e **`b`**. Não existe código de inicialização da variável **`c`** porque, segundo a definição da linguagem C, as variáveis locais não inicializadas têm valor inicial indefinido.
Nas linhas 5, 6 e 7 são preparados os argumentos para a chamada à função **`addp`**, que são os ponteiros para as variáveis **`a`**, **`b`** e **`c`**.
A instrução **`add $16, %rsp`** reposiciona RSP na posição inicial.

Passagem de argumentos em *stack*

Antes da chamada a uma função, os parâmetros são colocados nos registos e, no caso de serem mais que seis, são empilhados no *stack* pela ordem inversa da lista de parâmetros da função. O parâmetro mais à esquerda é o que fica no topo do *stack*.

Exemplo 13

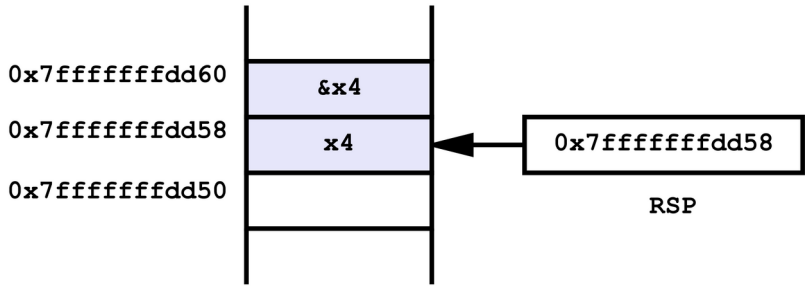
Consideremos a chamada à função **`proc`**, que tem 8 parâmetros.

```
1   void proc(long a1, long *a1p, int a2, int *a2p,
2             short a3, short *a3p, char a4, char *a4p);
3
4   long x1 = 1;
5   int x2 = 2;
6   short x3 = 3;
7   char x4 = -4;
8
9   long call_proc() {
10      proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
11      return (x1 + x2) * (x3 - x4);
12  }
```

`call_proc.c`

```
1       .data
2   x1:
3       .quad 1
4   x2:
5       .int  2
6   x3:
7       .word 3
8   x4:
9       .byte -4
10
11      .text
12      .global call_proc
13  call_proc:
14      sub     $8, %rsp
15      lea     x4(%rip), %rax
16      push    %rax
17      movsbl  x4(%rip), %eax
18      push    %rax
19      lea     x3(%rip), %r9
20      movswl  x3(%rip), %r8d
21      lea     x2(%rip), %rcx
22      mov     x2(%rip), %edx
23      lea     x1(%rip), %rsi
24      mov     x1(%rip), %rdi
25      call    proc
26      movslq  x2(%rip), %rax
27      add     $16, %rsp
28      add     x1(%rip), %rax
29      movswl  x3(%rip), %edx
30      movsbl  x4(%rip), %ecx
31      sub     %ecx, %edx
32      movslq  %edx, %rdx
        imul    %rdx, %rax
        add     $8, %rsp
        ret
```

call_proc_asm.s



Na linha 16 empilha-se o oitavo argumento – o ponteiro para **x4**. Na linha 18 empilha-se o sétimo argumento – o valor de **x4**. Note-se que apesar de ser do tipo **char** a passagem é feita numa palavra de 64 bits. Entre as linhas 19 e 24 procede-se à passagem dos restantes seis argumentos.

A convenção de chamada a funções define que na altura da execução da instrução **call** o registo RSP deve estar alinhado num endereço múltiplo de 16. Como consequência, à entrada de uma função, o RSP está sempre desalinhado de endereço múltiplo de 16. Assim, a função atual pode basear-se neste pressuposto para efeito de alinhamento do RSP ao realizar outras chamadas.

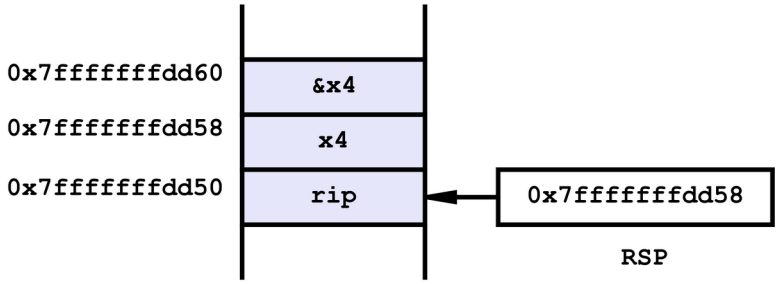
A instrução **sub $8, %rsp** na linha 14 serve para cumprir esta convenção. Até à instrução **call** na linha 25, o RSP vai ser decrementado de 24 ficando alinhado num endereço múltiplo de 16.

```
1   void proc(long a1, long *a1p, int a2, int *a2p,
2             short a3, short *a3p, char a4, char *a4p) {
3       *a1p += a1;
4       *a2p += a2;
```

```
5       *a3p += a3;
6       *a4p += a4;
7   }
```

proc.c

```
1       .text
2       .globl  proc
3   proc:
4       add     %rdi, (%rsi)
5       add     %edx, (%rcx)
6       add     %r8w, (%r9)
7       mov     8(%rsp), %dl
8       mov     16(%rsp), %rax
9       add     %dl, (%rax)
10      ret
```



proc_asm.s

No início da execução de uma função o endereço de retorno apresenta-se no topo do *stack,* depois dos argumentos da função. O acesso aos argumentos é realizado com base em RSP.

O acesso a **a4p** é realizado na linha 8. **16(%rsp)** equivale ao endereço **0x7fffffffdd60** que é o local do *stack* onde ser encontra o argumento **&x4.**
O acesso a **a4** é realizado na linha 7. **8(%rsp)** equivale ao endereço **0x7fffffffdd58** que é o local do stack onde ser encontra o argumento x4.

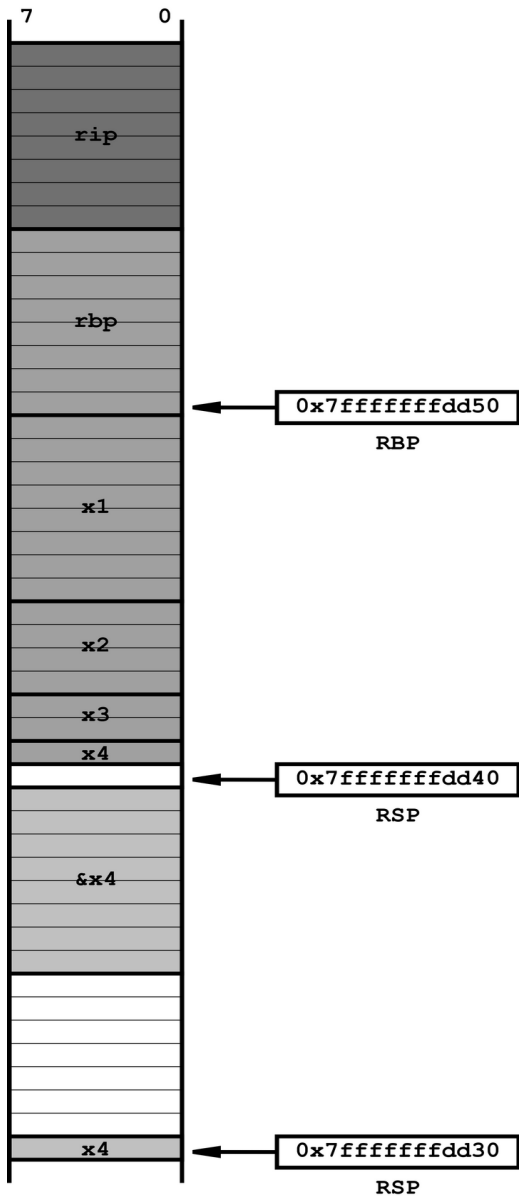
Exemplo 14

Neste exemplo vai ser mostrada uma utilização mais ampla do *stack*. Além de utilizado na passagem de argumentos vai também ser utilizado para alojamento de variáveis locais.

O exemplo é semelhante ao anterior com a diferença das variáveis **`x1`**, **`x2`**, **`x3`** e **`x4`** serem locais à função **`call_proc`**.

```
 1 long call_proc() {
 2
 3     long x1 = 1;
 4     int x2 = 2;
 5     short x3 = 3;
 6     char x4 = -4;
 7
 8     proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
 9     return (x1 + x2) * (x3 - x4);
10 }
```

```
 1      .text
 2      .global call_proc
 3  call_proc:
 4      push    %rbp
 5      mov     %rsp, %rbp
 6      sub     $16, %rsp
 7      movq    $1, -8(%rbp)
 8      movl    $2, -12(%rbp)
 9      movw    $3, -14(%rbp)
10      movb    $-4, -15(%rbp)
11      lea     -15(%rbp), %r8
12      push    %r8
13      movb   -15(%rbp), %dil
14      push    %rdi
15      lea     -14(%rbp), %r9
16      movw    -14(%rbp), %r8w
17      lea     -12(%rbp), %rcx
18      movl    -12(%rbp), %edx
19      lea     -8(%rbp), %rsi
20      movq    -8(%rbp), %rdi
21      call    proc
22      movslq  -12(%rbp), %rdx
23      movq    -8(%rbp), %rcx
24      add     %rdx, %rcx
25      movswl  -14(%rbp), %edx
26      movsbl  -15(%rbp), %eax
27      sub     %eax, %edx
28      mov     %edx, %eax
29      cltq
30      imul    %rcx, %rax
31      mov        %rbp, %rsp
32      pop     %rbp
33      ret
```



O bloco de código inicial linhas 4 a 6 designa-se por **preâmbulo** da função. No preâmbulo, linhas 4 e 5, o registo RBP é preparado para acesso aos valores em *stack* - argumentos e variáveis locais. Na linha 6 a adição de 16 a RSP reserva espaço de memória em *stack* para as variáveis locais.

O registo RSP sofre um decremento de 40 unidades até à instrução call o que garante o alinhamento a múltiplo de 16.

O registo RBP indica sempre a mesma posição do *stack* – a seguir ao endereço de retorno, onde é salvo o conteúdo de RBP da função chamadora – e mantém-se fixo durante a execução da função.

Para aceder aos parâmetros são usados deslocamentos positivos em relação a RBP: **`8(%rbp)`** corresponderia ao primeiro argumento em *stack*, **`16(%rbp)`** corresponderia ao argumento segundo e assim sucessivamente.

Para aceder às variáveis locais são usados deslocamentos negativos em relação a RBP : **`-8(%rbp)`** corresponde à primeira variável local - **`x1`**, **`-12(%rbp)`** corresponde à segunda variável local - **`x2`**, -

`14(%rbp)` corresponde à terceira variável local **`-x3`** e **`-15(%rbp)`** corresponde à terceira variável local.

Ao retornar da função é necessário repor RSP exatamente no estado inicial – libertar o espaço usado para passagem de argumentos, libertar o espaço reservado para variáveis locais e repor em RBP o conteúdo original.

O código responsável por esta operação designa-se por **epílogo** e neste caso é formado pelas instruções das linhas 31 e 32. A instrução **`movq %rsp, %rbp`** ao reposicionar RSP na posição de RBP liberta, eficazmente o espaço ocupado em *stack* que no alojamento de variáveis locais quer na passagem de argumentos.

Exemplo 15

Alojamento de *array* local de dimensão variável em *stack*

Consideremos a função **`get_year`** que extrai a componente ano, na forma de inteiro, de uma data representada numa string com o formato "2020-9-3".

```
1  int get_year(const char *date) {
2      char buffer[strlen(date) + 1];
3      strcpy(buffer, date);
4      return atoi(strtok(buffer, "-/ "));
5  }
```

`get_year.c`

```
1        .section .rodata
2  sep:  .asciz      "/- "
3
4        .text
5        .global get_year
6  get_year:
7        push  %rbp
8        mov   %rsp, %rbp
9        push  %rbx
10       mov   %rdi, %rbx
11       sub   $8, %rsp
12       call  strlen
13       inc   %eax
14       add   $15, %eax
15       and   $-16, %eax
16       sub   %rax, %rsp
17       mov   %rsp, %rdi
18       mov   %rbx, %rsi
19       call  strcpy
20       mov   %rsp, %rdi
21       lea   sep(%rip), %rsi
22       call  strtok
23       mov   %rax, %rdi
24       call  atoi
25       mov   -8(%rbp), %rbx
26       mov   %rbp, %rsp
27       pop   %rbp
28       ret
```

`get_year_asm.s`

A reserva de espaço para o *array* local **`buffer`** é realizada nas linhas 14, 15 e 16. A dimensão necessária é estabelecida na linha 13 – valor retornado por **`strlen`** mais um.

Na linha 14 e 15 essa dimensão (EAX) é arredondada por excesso para um valor múltiplo de 16.

((EAX + 15) / 16) * 16 (o sinal / representa divisão inteira).

Na linha 16 esse valor é subtraído a RSP consumando a reserva de espaço de memória para o *array* local **`buffer`**.

No final da função, RSP é restabelecido com o valor de RBP – linha 26. Sendo uma solução simples para de reajuste do RSP e libertação do espaço de memória reservado.

Nas circunstâncias em que o espaço de memória a reservar em *stack* é variável, o gcc por omissão gera código de mitigação do efeito *stack clash*. O código apresentado acima foi gerado pelo gcc sob o efeito da opção **`-fno-stack-clash-protection`**.

Exercício

Programar a função copy_if em linguagem assembly x86-64.

```
size_t copy_if(void *dst, size_t dst_size,
               void *src, size_t src_size, size_t elem_size,
               int (*predicate)(const void *, const void *),
               const void *context) {

    char *src_ptr = src, *src_last = (char *)src + src_size * elem_size;
    char *dst_ptr = dst, *dst_last = (char *)dst + dst_size * elem_size;

    for (src_ptr = src; src_ptr < src_last; src_ptr += elem_size)
        if (predicate(src_ptr, context) && dst_ptr < dst_last) {
            memcpy(dst_ptr, src_ptr, elem_size);
            dst_ptr += elem_size;
        }
    return (dst_ptr - (char*)dst) / elem_size;
}
```

copy_if.c

```
typedef struct person {
      char name[20];
      int age;
} Person;

Person people[] = {
      {"Luis", 20},
      {"António", 30},
      {"Manuel", 50}
};

Person found_people[3];

int older_than(const void *elem, const void *age) {
      return ((Person *)elem)->age > (int)(long)age;
}

int main() {
      size_t n = copy_if(found_people, ARRAY_SIZE(found_people),
                         people, ARRAY_SIZE(people), sizeof(people[0]),
                         older_than, (const void *)25);
      for (size_t i = 0; i < n; ++i)
            printf("%s, %d\n", found_people[i].name, found_people[i].age);
}
```

main.c

Exemplo 16

Determinar experimentalmente a dimensão de *stack* disponível.

```
 1 #include <stdio.h>
 2
 3 unsigned long get_sp();
 4
 5 void recurse() {
 6       char array[1024*1024];
 7       printf("%lx\n", get_sp());
 8       recurse();
 9 }
10
11 int main() {
12       recurse();
13 }
```

print_sp.c

```
1       .text
2       .global     get_sp
3 get_sp:
4       mov   %rsp, %rax
5       ret
```

get_sp.s

Exemplo 17

Buffer overflow

A função **`gets`** lê caracteres do *standard input* e escreve-os no *array* passado em parâmetro até ler uma marca de fim de linha - \n. Se o *array* tiver uma dimensão inferior ao número de caracteres lidos a função **`gets`** escreve-os para além do limite do *array*, corrompendo informação armazenada na vizinhança.

No exemplo, se o número de caracteres lidos for superior a 7, irá ocorrer falha. Quais as consequências dessa falha?

Esta função foi retirada da biblioteca normalizada da linguagem C pelo potencial de falha que pode introduzir num programa.

```c
#include <stdio.h>

void print_secret(int n) {
      ...
}

void secrets() {
      struct {
            int a;
            char buffer[7];
            short b;
      } x;

      x.a = 'a';
      x.b = 'b';

      gets(x.buffer);

      if (x.b == 'B')
            print_secret(1);

      if (x.a == 'A')
            print_secret(2);
}

int main() {
      printf("Secrets\n");
      secrets();
}
```

main.c

```
secrets:
      pushq %rbp
      movq  %rsp, %rbp
      subq  $16, %rsp
      movl  $'a', -16(%rbp)
      movw  $'b', -4(%rbp)
      leaq  -16(%rbp), %rax
      addq  $4, %rax
      movq  %rax, %rdi
      movl  $0, %eax
      call  gets
      movl  -4(%rbp), %eax
      cmpl  $'B', %eax
      jne   .L5
      movl  $1, %edi
      call  print_secret
.L5:
      movl  -16(%rbp), %eax
      cmpl  $'A', %eax
      jne   .L7
      movl  $2, %edi
      call  print_secret
.L7:
      nop
      leave
      ret
```

secrets_asm.s

```
$ gcc secret.c -fno-stack-protector -o secret
```

Exercício

1. Fazer com que a sequência de caracteres introduzida provoque a execução de **`print_secret(1)`**.

2. Fazer com que a sequência de caracteres introduzida provoque a execução de **`print_secret(2)`**.

   Sistema ASLR (Address Space Layout Randomization) do Linux.

   Visualizar o estado:

   0 = Disabled
   1 = Conservative Randomization
   2 = Full Randomization

   ```
   $ cat /proc/sys/kernel/randomize_va_space
   ```

   Alterar o estado:

   ```
   $ sudo sysctl -w kernel.randomize_va_space=0
   ```

Referências

- Calling conventions for different C++ compilers and operating systems
  https://www.agner.org/optimize/calling_conventions.pdf

- System V Application Binary Interface

  AMD64 Architecture Processor Supplement

  https://wiki.osdev.org/System_V_ABI#x86-64

- Application Binary Interface for the Arm® Architecture - The Base Standard

  https://developer.arm.com/documentation/ihi0036/d/?lang=en#the-generic-c-abi

- Compiler Explorer, https://gcc.godbolt.org/

- Using AS, https://sourceware.org/binutils/docs-2.25/as/index.html

- Intel® 64 and IA-32 Architectures Software Developer Manuals,
  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html