

14 Programação em linguagem <i>assembly</i> .....	3
14.1 Operações sobre valores numéricos.....	3
14.1.1 Operações aritméticas.....	3
14.1.2 Adição de valores a 32 bit.....	3
14.1.3 Afectação com valor constante.....	3
14.1.4 Deslocamento para a direita de um valor a 32 bit.....	4
14.1.5 Deslocamento para a esquerda um valor a 32 bit.....	4
14.1.6 Rotação de valores.....	4
14.1.7 Afectação de um bit de um registo com zero.....	4
14.1.8 Afectação de um bit de um registo com um bit de outro registo.....	4
14.1.9 Multiplicação por valor constante.....	5
14.2 Controlo da execução.....	5
14.3 Avaliação de condições.....	8
14.3.1 Teste do valor de um bit.....	8
14.3.2 Teste do valor zero.....	8
14.3.3 Comparação de valores numéricos.....	8
14.4 Acesso a valores em memória.....	9
14.4.1 Carregamento de endereço em registo.....	10
14.5 Acesso a valores em <i>array</i> .....	11
14.6 Funções.....	12
14.6.1 Função sem parâmetros.....	13
14.6.2 Função com parâmetros.....	14
14.7 Stack.....	16



## 14 PROGRAMAÇÃO EM LINGUAGEM ASSEMBLY

Neste capítulo exemplifica-se a programação em linguagem *assembly* para o P16, na implementação de abstrações de linguagem de alto nível presentes na linguagem C, designadamente: expressões aritméticas e lógicas, estruturas de controlo de execução *if/else*, *for* e *while*, acesso a *arrays* e chamada de funções.

### 14.1 Operações sobre valores numéricos

#### 14.1.1 Operações aritméticas

**a = a + b**

Considerando que a variável **a** está em R0 e a variável **b** está em R1:

```
add    r0, r0, r1
```

**a = c + b - d**

Considerando que a variável **a** está em R0, a variável **b** está em R1, a variável **c** está em R2 e a variável **d** está em R3:

```
add    r0, r2, r1
sub     r0, r0, r3
```

**a = b - 3**

Considerando que a variável **a** está em R0, e a variável **b** está em R1:

```
sub     r0, r1, 3
```

#### 14.1.2 Adição de valores a 32 bit

O primeiro operando está em R1:R0, a parte baixa em R0 e a parte alta em R1, o segundo operando está em R3:R2; o resultado é depositado em R5:R4.

```
add     r4, r2, r0
adc     r5, r3, r1
```

#### 14.1.3 Afectação com valor constante

Considera-se que uma variável do tipo **int** é representada a 16 bit.

<pre>int a = 3045;</pre>	<pre>; r0 - a mov     r0, 3045 &amp; 0xff movt    r0, 3045 &gt;&gt; 8</pre>
(a)	(b)

#### Programa 1: Afectação de registo com valor constante

O valor 3045 é igual a 1011 1110 0101 em binário. A instrução **mov** carrega o valor 1110 0101 na parte baixa de R0 e coloca a parte alta a zero. Se o valor da constante fosse inferior a 256 bastaria esta instrução. Sendo superior, é necessária a instrução **movt** para carregar 1011 na parte alta de R0 e assim formar o valor 3045 em R0. A instrução **movt** mantém a parte baixa do registo inalterada.

#### 14.1.4 Deslocamento para a direita de um valor a 32 bit

O valor a deslocar encontra-se em R1:R0.

Deslocar uma posição	Deslocar 4 posições
<pre>lsr    r1, r1, 1 rrx    r0, r0</pre>	<pre>lsr    r0, r0, 4 lsl    r2, r1, 16 - 4 add    r0, r0, r2 lsr    r1, r1, 4</pre>
(a)	(b)

**Programa 2: Deslocamento para a direita de um valor a 32 bits**

#### 14.1.5 Deslocamento para a esquerda um valor a 32 bit

O valor a deslocar encontra-se em R1:R0.

Deslocar uma posição.	Deslocar quatro posições.
<pre>lsl    r0, r0, 1 adc    r1, r1, r1</pre>	<pre>lsl    r1, r1, 4 lsr    r2, r0, 16 - 4 add    r1, r1, r2 lsl    r0, r0, 4</pre>
(a)	(b)

**Programa 3: Deslocamento para a esquerda de um valor a 32 bits**

#### 14.1.6 Rotação de valores

Rodar uma palavra para a direita significa inserir nas posições de maior peso, os *bits* que saem das posições de menor peso; rodar uma palavra para a esquerda significa inserir nas posições de menor peso os *bits* que saem das posições de maior peso.

Rodar o valor de R0 três posições para a direita.	Rodar o valor de R0 cinco posições para a esquerda.
<pre>ror    r0, r0, 3</pre>	<pre>ror    r0, r0, 16 - 5</pre>

#### 14.1.7 Afectação de um bit de um registo com zero

Afectar o *bit* de peso 12 de R0 com o valor zero, mantendo o valor dos restantes *bits* de R0.

```
mov    r1, 0b11111111
movt   r1, 0b11101111
and    r0, r0, r1
```

#### 14.1.8 Afectação de um bit de um registo com um bit de outro registo

Afectar o *bit* de peso 4 de R0 com o valor do *bit* de peso 13 de R1.

```
lsr    r2, r1, 13 - 4
mov    r3, 0b00010000
and    r2, r2, r3
orr    r0, r0, r2
```

```
orr    r0, r0, r2
```

### 14.1.9 Multiplicação por valor constante

A multiplicação de uma variável por uma constante pode ser realizada de forma eficiente, sem recorrer a instruções de multiplicação ou programa genérico de multiplicação. Veja-se o seguinte exemplo:

$$v * 23 = v * (16 + 4 + 2 + 1) = v * 16 + v * 4 + v * 2 + v * 1$$

A constante 23 é decomposta em parcelas de valor igual a potências de 2. As multiplicações são realizadas por instruções de deslocamento.

<pre>int a; int b = a * 23;</pre>	<pre>;ro - a, r1 - b mov    r1, r0      ; * 1 lsl    r0, r0, 1 add    r1, r1, r0  ; * 2 lsl    r0, r0, 1 add    r1, r1, r0  ; * 4 lsl    r0, r0, 2 add    r1, r1, r0  ; * 16</pre>
(a)	(b)

Programa 4: Multiplicar por constante

## 14.2 Controlo da execução

### if

<pre>if (i != 0)     f += 3; g--;</pre>	<pre>;r5 - i, f - r2, g - r4 add    r5, r5, 0 beq    if_end add    r2, r2, 3 if_end: sub    r4, r4, 1</pre>
(a)	(b)

Programa 5: if

As instruções de salto condicional baseiam-se no valor das *flags* do processador. A avaliação booleana de expressões consiste em realizar operações aritméticas ou lógicas que afectem as *flags* com valores que sejam conclusivos.

No exemplo acima, a instrução `add r5, r5, 0` afecta a *flag* Z com um se R5 for zero. A instrução `beq` salta sobre a instrução `add r2, r2, 3`, que corresponde a `f += 3`, se a *flag* Z for um, o que significa que `i` (R5) é igual a zero. A condição de salto da instrução *assembly* é contrária à condição definida na linguagem de alto nível.

### if/else

<pre>if (i == j) {     f &lt;=&amp; 1;     i++; } else {     f &gt;&gt;= 2;     i -= 2; }</pre>	<pre>1    ;r5 - i, r3 - j, r2 - f 2        cmp    r5, r3 3        bne    if_else 4        lsl    r2, r2, 1 5        add    r5, r5, 1 6        b      if_end 7    if_else:</pre>
---	---

j++;	8            lsr    r2, r2, 2
	9            sub    r5, r5, 2
	10   if_end:
	11           add    r3, r3, 1

(a)
(b)

**Programa 6: if/else**

A instrução **cmp r5, r3** realiza a subtracção  $R5 - R3$  e afecta a *flag* Z com um, se a diferença for zero, o que significa que **i** (R5) e **j** (R3) são iguais. No caso de **i** ser diferente de **j**, a *flag* Z é afectada com zero e a instrução **bne** salta por cima do bloco *if* – linhas 4, 5 e 6 – directamente para o código do bloco *else* – linhas 8 e 9. No caso de **i** ser igual a **j** a instrução **bne** não realiza o salto e executa o bloco *if*. Este bloco termina com um salto incondicional – linha 6, para não executar o bloco *else* posicionado nos endereços imediatos.

### switch/case

switch (v) {	1        ;r0 = v, r1 - a
case 1:	2        switch_case1:
a = 11;	3        mov    r2, 1
break;	4        cmp    r0, r2
case 10:	5        bne    switch_case10
a = 111;	6        mov    r1, 11
break;	7        b       switch_break;
default:	8        switch_case10:
a = 0;	9        mov    r2, 10
}	10       cmp    r0, r2
	11       bne    switch_default
	12       mov    r1, 111
	13       b       switch_break;
	14       switch_default:
	15       mov    r1, 0
	16       switch_break:

(a)
(b)

**Programa 7: switch/case**

A implementação do *switch/case* consiste em encadear um conjunto de *ifs*, um para cada caso (*case*).

### do while

do {	1        ;r0 = v, r1 - 1
v >> 1;	2        do_while:
l += 1;	3        lsr    r0, r0, 1
} while (v != 0) {	4        add    r1, r1, 1
	5        sub    r0, r0, 0
	6        bne    do_while

(a)
(b)

**Programa 8: do while**

A programação *assembly* segue a ordem de escrita e de execução da programação em C – primeiro executa o corpo de instruções – linhas 3 e 4 e no final avalia a condição – linhas 5 e 6.

## while

<pre>while (v != 0) {     v &gt;&gt; 1;     l += 1; }</pre>	<pre>1    ;r0 - v, r1 - 1 2    while: 3        sub    r0, r0, 0 4        beq    while_end 5        lsr    r0, r0, 1 6        add    r1, r1, 1 7        b      while 8    while_end: 9</pre>	<pre>1    ;r0 - v, r1 - 1 2    while: 3        b      while_cond 4    while_do: 5        lsr    r0, r0, 1 6        add    r1, r1, 1 7    while_cond: 8        sub    r0, r0, 0 9        bne    while_do</pre>
(a)	(b)	(c)

### Programa 9: while

O programa *assembly* da figura 9(b) é escrito e executado pela ordem da linguagem C – primeiro a avaliação da condição – linhas 3 e 4 – e depois o corpo de instruções do *while* – linhas 6 e 7. Com esta programação o processador executa 5 instruções durante o ciclo – linhas 3 a 7, entre elas duas instruções *branch* – linhas 4 e 7.

Na figura 9(c) o programa é escrito como num *do while*, com a avaliação da condição no fim – linhas 8 e 9. O *while* começa com um salto incondicional – linha 3 – para a avaliação da condição, porque esta deve ser executada em primeiro lugar. Esta programação resulta na supressão de uma instrução *branch* durante o ciclo, relativamente à programação apresentada na figura 9(b), o que a torna preferível.

## for

<pre>for (i = 0, a = 1; i &lt; n; ++i) {     a &lt;=&lt;= 1; }</pre>	<pre>1    ;r0 - i, r1 - a, r2 - n 2    mov    r0, 0 3    mov    r1, 1 4    b      for_cond 5    for: 6        lsl    r1, r1, 1 7        add    r0, r0, 1 8    for_cond: 9        cmp    r0, r2         blo    for</pre>
(a)	(b)

### Programa 10: for

A instrução

```
for (expression1; expression2; expression3)  
    statement;
```

é equivalente a

```
expression;  
while (exp2ression) {  
    statement;  
    expression3;  
}
```

A programação *assembly* apresentada na figura 10(b) reflecte esta equivalência, com o *while* implementado na variante mais eficiente – figura 9(c).

## 14.3 Avaliação de condições

### 14.3.1 Teste do valor de um bit

Testar o valor do *bit* da terceira posição (peso 2) do registo R0.

```
ror    r0, r0, 3
bcs    label
```

### 14.3.2 Teste do valor zero

```
add    r5, r5, 0
beq    label
```

### 14.3.3 Comparação de valores numéricos

<pre>if (a &lt; b)     c = a;</pre>	<pre>;r0 - a, r1 - b, r2 - c     cmp    r0, r1     bhs    if_end     mov    r2, r0 if_end:</pre>
(a)	(b)

Programa 11: Comparação “menor que”

O código do bloco *if* é colocado imediatamente após o código de avaliação da condição. A condição da instrução de salto – *high or same* – está relacionada com a instrução **cmp** anterior. Nestas instruções a mnemónica da condição aplica-se ao primeiro operando da instrução **cmp** anterior. Quem é maior ou igual? O valor do registo R0 que corresponde à variável **a**.

<pre>if (a &gt; b)     c = a;</pre>	<pre>;r0 - a, r1 - b, r2 - c     cmp    r1, r0     bhs    label     mov    r2, r0 if_end:</pre>
(a)	(b)

Programa 12: Comparação “maior que”

Para avaliar a condição contrária (**if (a < b)**) com a mesma instrução de comparação (**cmp r0, r1**) a condição de salto seria a contrária – *lower or same*. Como não existe no P16 instrução de salto com esta condição, a solução apresentada realiza a subtração com os operandos em posições invertidas (**cmp r1, r0**) e continua a usar **bhs**.

	operação	números naturais	números relativos
<b>if (a &lt; b)</b>	<b>cmp r0, r1</b>	<b>bhs</b>	<b>bge</b>
<b>if (a &lt;= b)</b>	<b>cmp r1, r0</b>	<b>blo</b>	<b>blt</b>
<b>if (a &gt; b)</b>	<b>cmp r1, r0</b>	<b>bhs</b>	<b>bge</b>
<b>if (a &gt;= b)</b>	<b>cmp r0, r1</b>	<b>blo</b>	<b>blt</b>

Tabela 1: Comparação de números

Na tabela 1 apresentam-se soluções de programação para as quatro relações possíveis.



Na comparação de números relativos, codificados em código de complementos, devem ser utilizadas as instruções **bge** (*branch greater or equal*) ou **blt** (*branch less than*).

#### 14.4 Acesso a valores em memória

No P16, o acesso a variáveis em memória faz-se utilizando o modo de endereçamento indirecto. Este modo consiste em utilizar o conteúdo de registos do processador como endereço de memória. Antes de realizar a operação de acesso à variável (LDR ou STR) é necessário carregar previamente o endereço da variável num registo do processador.

Transferência do conteúdo de uma variável alojada em memória para os registos do processador.

Variável <b>x</b> , representada a 8 bits, alojada na posição de memória de endereço 5	Variável <b>y</b> , representada a 16 bits, alojada nas posições de memória de endereços 6 e 7
<pre>mov r1, x ldrb r0, [r1]</pre>	<pre>mov r1, y ldr r0, [r1]</pre>
<p>A instrução <b>mov</b> afecta os 8 bits menos significativos de R1 com o endereço da variável <b>x</b> (valor 5) e afecta o 8 bits mais significativos com zero.</p> <p>A instrução <b>ldrb</b> copia o conteúdo da posição de memória de endereço 5 (valor 0x23) para os 8 bits menos significativos de R0 e afecta os 8 bits mais significativos com zero.</p>	<p>A instrução <b>mov</b> afecta os 8 bits menos significativos de R1 com o endereço da variável <b>y</b> (valor 6) e afecta o 8 bits mais significativos com zero.</p> <p>A instrução <b>ldr</b> copia dois bytes da memória para o registo R0. O conteúdo da posição de memória de endereço 6 (valor 0x7a) para os 8 bits menos significativos de R0 e o conteúdo da posição de memória de endereço 7 (valor 0x3e) para os 8 bits mais significativos – <i>little ended</i>.</p>

Afectação de uma variável alojada em memória com o conteúdo de registos do processador.

Variável <b>x</b> , representada a 8 bits, alojada na posição de memória de endereço 5.	Variável <b>y</b> , representada a 16 bits, alojada nas posições de memória de endereços 6 e 7
<pre>mov r1, x</pre>	<pre>mov r1, y</pre>

<code>strb r0, [r1]</code>	<code>str r0, [r1]</code>
A instrução <b>mov</b> afecta os 8 <i>bits</i> menos significativos de R1 com o endereço da variável <b>x</b> (valor 5) e afecta o 8 bits mais significativos com zero.	A instrução <b>mov</b> afecta os 8 <i>bits</i> menos significativos de R1 com o endereço da variável <b>y</b> , valor 6, e afecta o 8 bit mais significativos com zero.
A instrução <b>strb</b> copia o valor dos 8 bits menos significativos de R0 para a posição de memória de endereço 5 (valor 0x9b).	A instrução <b>str</b> copia o conteúdo do registo R0 para a memória. O valor dos 8 <i>bits</i> menos significativos de R0 para a posição de memória de endereço 6 (valor 0xa4) e o valor dos 8 <i>bits</i> mais significativos de R0 (valor 0x67) para a posição de memória de endereço 7 – <i>little ended</i> .

Esta forma de carregar o endereço da variável no registo do processador com a instrução **mov**, tem a limitação de apenas poder carregar endereços na gama 0 – 255 porque a dimensão do campo destinado à constante, no código da instrução **mov**, é de 8 *bits*.

#### 14.4.1 Carregamento de endereço em registo

A solução geral para carregamento de endereços nos registos do processador passa por utilizar a instrução **ldr rd, label**. Esta instrução copia um valor representado a 16 *bits* para o registo indicado. Esse valor, que pode ser o endereço de uma *label* ou outra constante, está alojado em memória no endereço definido por **label**.

Para aceder à posição de memória definida por **label**, esta instrução usa um método de endereçamento relativo ao PC. O código binário desta instrução tem um campo de 6 *bits* para codificar, em número de palavras de 16 *bits* (*words*), a distância, no sentido crescente, a que **label** se encontra do PC.

Na figura 1, ilustra-se a disposição em memória, assim como os códigos binários da sequência de instruções para carregamento do conteúdo da variável **x**, localizada no endereço 0x6037, no registo R0.

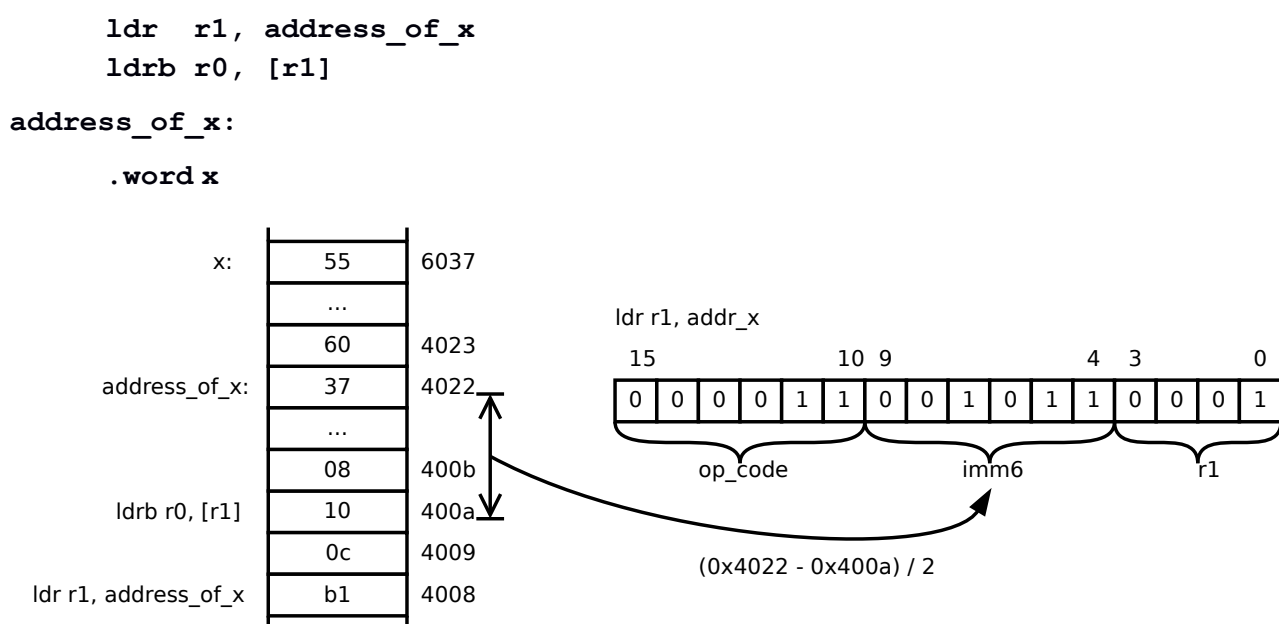


Figura 1 – Carregamento em registo do valor de uma *label*

A instrução `ldr r1, address_of_x` carrega em R1 o endereço da variável **x** (0x6037) que está armazenado em memória na posição indicada pela *label* **address\_of\_x** com endereço 0x4022. Esta instrução determina o endereço de **address\_of\_x** adicionando ao **PC** (0x400a) a distância representada no campo **imm6** ( $0x4022 = 0x400a + 0xb * 2$ ). (Na fase de execução de uma instrução, o PC tem o endereço da instrução seguinte.)

## 14.5 Acesso a valores em *array*

Os *arrays* são conjuntos de variáveis do mesmo tipo alojadas em posições de memória contíguas. As posições do *array* são definidas pelo índice.

Os acessos aos elementos do *array* são realizados pelas seguintes instruções de endereçamento baseado - indexado:

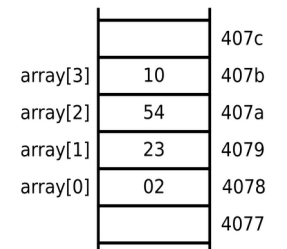
```
ldr rd, [rn, rm]    ldr rd, [rn, imm3*2]
str rd, [rn, rm]    str rd, [rn, imm3*2]
```

se se tratar de um *array* de *words* ou

```
ldrb rd, [rn, rm]   ldrb rd, [rn, imm3]
strb rd, [rn, rm]   strb rd, [rn, imm3]
```

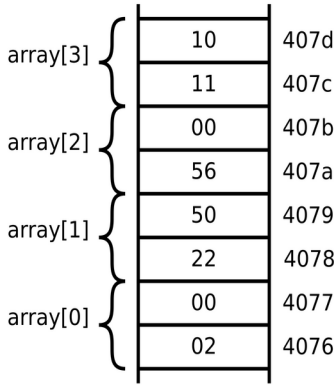
se se tratar de um *array* de *bytes*.

Estas instruções determinam o endereço de acesso somando o valor de **rn** com **rm** ou com a constante **imm3\*2** ou **imm3**. Em **rn** carrega-se o endereço da primeira posição do *array* e através da segunda componente – **rm**, **imm3\*2** ou **imm3** – define-se a posição a que se pretende aceder.

<pre>char array[] = {     2, 0x23, 0x54, 0x10};  for (i = 0; i &lt; 10; ++i)     a += array[i]</pre>	 <p>array[3] 10 407b array[2] 54 407a array[1] 23 4079 array[0] 02 4078 4077</p>	<pre>; r0 - array, r1 - i, r2 - a mov    r1, 0 mov    r4, 10 b      for_cond for: ldrb   r3, [r0, r1] add    r2, r2, r3 add    r1, r1, 1 for_cond: cmp    r1, r4 blo    for</pre>
(a)	(b)	(c)

**Programa 13: Acesso a *array* de *bytes***

No programa 13 assume-se que o endereço inicial do *array* foi previamente carregado no registo R0 – valor 0x4078. Cada posição deste *array* de *bytes* ocupa uma posição de memória. Determinar o endereço duma dada posição do *array* consiste em somar o índice da posição ao endereço inicial, operação realizada pela instrução `ldrb r3, [r0, r1]`.

<pre>int array[] = {     2, 0x5022, 0x56, 0x1011};  for (i = 0; i &lt; 10; ++i)     a += array[i]</pre>		<pre>; r0 - array, r1 - i ; r2 - a     mov    r1, 0     mov    r4, 10     b      for_cond for:     lsl    r3, r1, 1     ldr    r3, [r0, r3]     add    r2, r2, r3     add    r1, r1, 1 for_cond:     cmp    r1, r4     blo    for</pre>
(a)	(b)	(c)

**Programa 14: Acesso a array de words**

No programa 14, os elementos do *array* são valores representados a 16 *bits* – ocupam duas posições de memória. O acesso ao elemento **array[i]** é realizado pela instrução **ldr r3, [r0, r3]** que acede à posição de memória que resulta da soma de R0 com R3. Assume-se que R0 tem o endereço da primeira posição do *array* (endereço de **array[0]**) e R3 a distância, em posições de memória, entre o endereço de **array[i]** e o endereço de **array[0]**. Esta distância é determinada pela instrução **lsl r3, r1, 1** que multiplica o índice, em R1, pela dimensão de cada elemento (2 *bytes*).

## 14.6 Funções

“Função” é o termo que se usa na linguagem C para designar uma sequência de instruções que realizam uma tarefa específica. Também se usam na programação em geral termos como “rotina”, “subrotina”, “método” (em linguagens OO) ou “procedimento” (SQL), com aproximadamente o mesmo significado.

O objectivo da sua utilização é subdividir e organizar os programas, eventualmente extensos e complexos, em operações mais pequenas, mais simples e reutilizáveis.

Neste texto usa-se o termo “função”, por coerência com a utilização da linguagem C.

Uma funcionalidade importante num processador é o suporte à implementação de funções, ou seja, a existência de um mecanismo que possibilite invocar um mesmo troço de programa a partir de qualquer ponto do programa e retornar ao ponto de invocação. Esta funcionalidade é concretizada pela acção de chamada que transfere o fluxo de execução para o endereço onde reside a rotina (de modo semelhante a uma instrução de salto) e simultaneamente memoriza o valor corrente do PC. A acção de retorno consiste em transferir o valor anteriormente memorizado e para o PC.

No P16, durante a execução de uma instrução, o PC contém o endereço da instrução seguinte. Se este valor for guardado como o endereço de retorno, fica assegurado o retorno à posição correcta.

A solução adoptada no P16 para memorizar o endereço de retorno foi utilizar o registo R14. Devido a esta funcionalidade este registo tem também o nome de registo de ligação (*link register* ou LR). A instrução **bl**, antes de afectar o PC com o endereço da rotina, transfere o valor actual do PC para o LR. O retorno ao ponto de invocação, faz-se copiando o conteúdo de LR para o PC, por exemplo, com a instrução **mov pc, lr**.

### 14.6.1 Função sem parâmetros

Considere-se a sequência de duas chamadas à função **void delay()**

...	1	...
<b>delay()</b> ;	2	<b>bl delay</b>
...	3	...
<b>delay()</b> ;	4	<b>bl delay</b>
...	5	...

(a) (b)

**Programa 15: Chamada a função sem parâmetros**

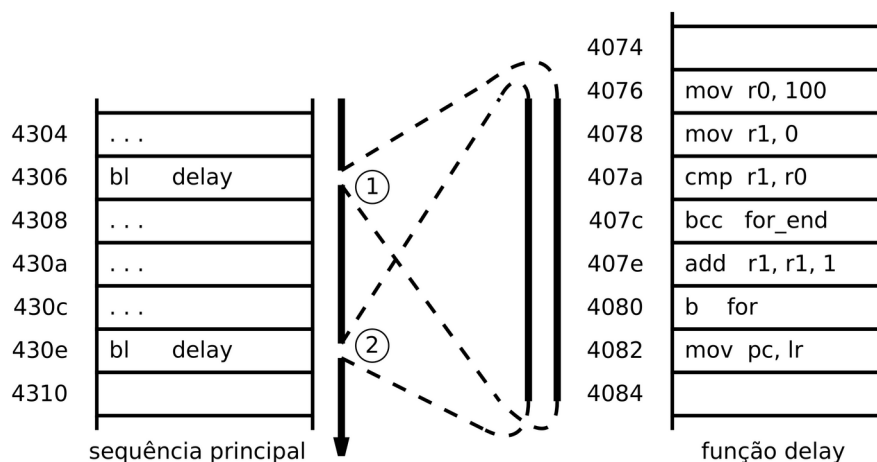
A chamada a funções sem parâmetros e sem valor de retorno corresponde apenas à execução da instrução **bl** para a *label* que define o início da função. No programa 15, **bl delay** nas linhas 2 e 4.

<b>void delay()</b> { <b>for</b> ( <b>int</b> <b>i</b> = 0; <b>i</b> < 100; ++ <b>i</b> ) ; }	1 <b>delay:</b> 2 <b>mov</b> <b>r0</b> , 100 3 <b>mov</b> <b>r1</b> , 0 4 <b>for:</b> 5 <b>cmp</b> <b>r1</b> , <b>r0</b> 6 <b>bcc</b> <b>for_end</b> 7 <b>add</b> <b>r1</b> , <b>r1</b> , 1 8 <b>b</b> <b>for</b> 9 <b>for_end:</b> 10 <b>mov</b> <b>pc</b> , <b>lr</b>
--	--

(a) (b)

**Programa 16: Programação de função sem parâmetros**

Na função **delay** a variável local **i** tem apenas o âmbito do *for*, pode ser suportada num registo do processador, no caso do programa 16, o registo R1.



**Figura 2- Chamada de função sem parâmetros**

Na primeira chamada, no endereço 0x4306, o processador começa por transferir o conteúdo de PC (0x4308) para LR e em seguida afecta o PC com o endereço de **delay** (0x4076). Na segunda chamada, no endereço 0x430e, o processador realiza acções semelhantes, com a diferença do valor de LR ser 0x4310.

Ao executar a instrução **mov pc, lr** posicionada no final da função, no endereço 0x4082, o processamento regressa ao endereço 0x4308 no caso da primeira chamada e regressa ao endereço 0x4310 no caso da segunda chamada.

### 14.6.2 Função com parâmetros

Considere-se a sequência de duas chamadas à função **multiply** com a seguinte assinatura:

```
uint16_t multiply(uint8_t multiplying, uint8_t multiplier);
```

...	1	mov	r0, 4
product[2] = multiply(4, 10);	2	mov	r1, 10
product[3] = multiply(8, 10);	3	bl	multiply
...	4	str	r0, [r4, 4]
	5	mov	r0, 8
	6	mov	r1, 10
	7	bl	multiply
	8	str	r0, [r4, 6]

(a)

(b)

#### Programa 17: Chamada de função com parâmetros

A função **multiply** tem dois parâmetros – **multiplying** e **multiplier** e retorna valor. Na fase de chamada, antes da execução de **bl** é necessário passar os argumentos. O que significa colocar os valores dos argumentos no local que dá suporte aos parâmetros. Nesta função utilizam-se o registo R0 para passar o primeiro argumento e o registo R1 para passar o segundo argumento. Na primeira chamada os valores dos argumentos são 4 e 10 e são carregados em **r0** e **r1**, respectivamente – linhas 1 e 2, na segunda chamada os valores dos argumentos são 8 e 10 e são passados do mesmo modo.

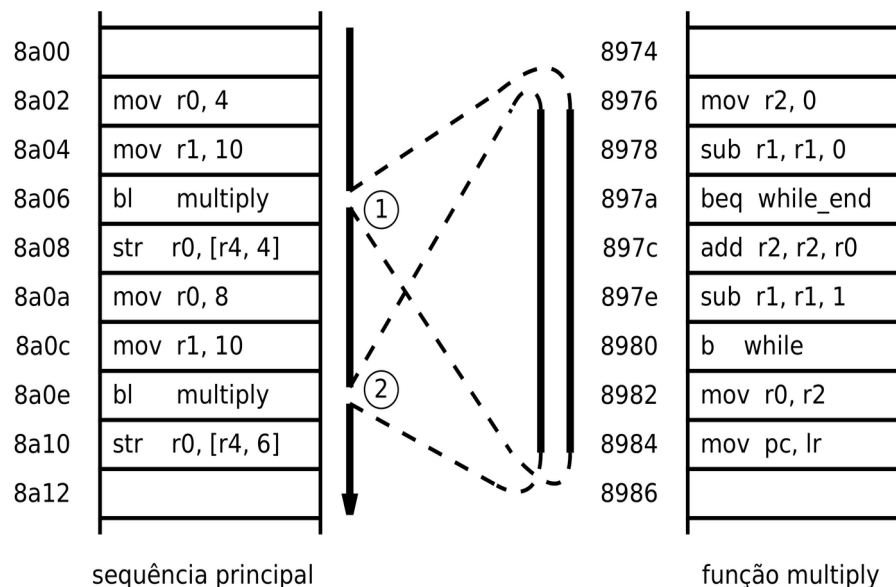
uint16_t multiply(uint8_t multiplying,	1	multiply:
uint8_t multiplier) {	2	mov r2, 0
uint16_t product = 0;	3	while:
while ( multiplier > 0 ) {	4	sub r1, r1, 0
product -= multiplying;	5	beq while_end
multiplier--;	6	add r2, r2, r0
}	7	sub r1, r1, 1
return product;	8	b while
}	9	while_end:
	10	mov r0, r2
	11	mov pc, lr

(a)

(b)

#### Programa 18: Programação de função com parâmetros

Ao programar a função **multiply** em *assembly* assume-se que os registos de suporte aos parâmetros – R0 e R1 – já contêm os argumentos. A variável local **product** como não prevalece para além do âmbito desta função é suportada no registo R2, entre as linhas 2 e 10. O valor da função – o resultado da multiplicação – é depositado no registo R0 – linha 10. Em ambas as chamadas, o valor retornado pela função é guardado em *array* – linhas 4 e 8 do programa 17.



**Figura 3 – Chamada de função com parâmetros**

Na linha 11, a instrução **mov pc, lr** faz o processador retornar ao endereço 0x8a08 na primeira chamada e ao endereço 0x8a10 na segunda chamada. Nestas posições encontram-se instruções para processamento do valor retornado pela função no registo R0.

### 14.7 Stack

O *stack* é uma zona de memória para salvaguarda temporária de dados do programa de diversa natureza. O seu nome advém do tipo de estrutura de dados que implementa ser do tipo *last-in-first-out* (LIFO), também designada por *stack*. Esta estrutura de dados tem um funcionamento análogo a uma pilha de objectos: só se consegue retirar da pilha o objecto que se encontra no topo – o que foi lá colocado mais recentemente – e só se consegue colocar um novo objecto sobre o topo da pilha – apenas sobre o objecto anteriormente lá colocado.

O P16 dispõe de um registo específico e duas instruções para manusear o *stack*. O registo R13, neste contexto designado *stack pointer* (SP), destina-se a guardar permanentemente o endereço corrente do topo do *stack*. A instrução **push** coloca o conteúdo de um registo no topo do *stack* e a instrução **pop** retira um valor do topo do *stack*, colocando-o num registo.

As instruções **push** e **pop** transferem o conteúdo completo de um registo (uma *word*), ou seja, não é possível transferir apenas um *byte* como acontece com as instruções **ldrb** e **strb**.

A instrução **push** começa por decrementar o registo SP de duas unidades e em seguida transfere o conteúdo do registo indicado para a posição do *stack* definida por SP.

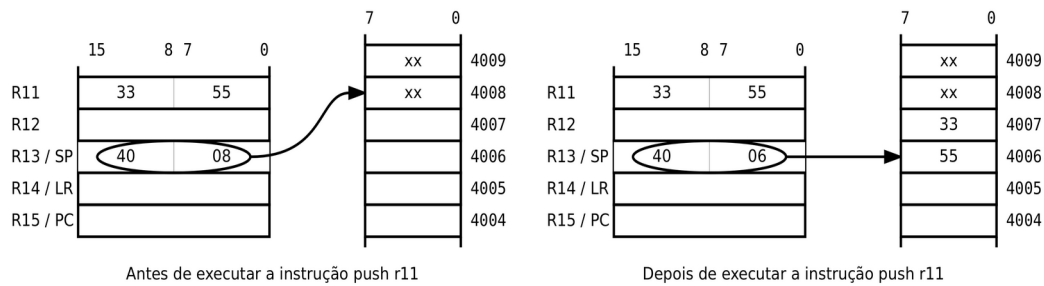
**push rs** é equivalente a

```
sub sp, sp, 2
str rs, [sp]
```

A instrução **pop** faz a operação inversa do **push**. Começa por incrementar o registo SP de duas unidades e em seguida transfere o conteúdo da posição do *stack* definida por SP para o registo indicado.

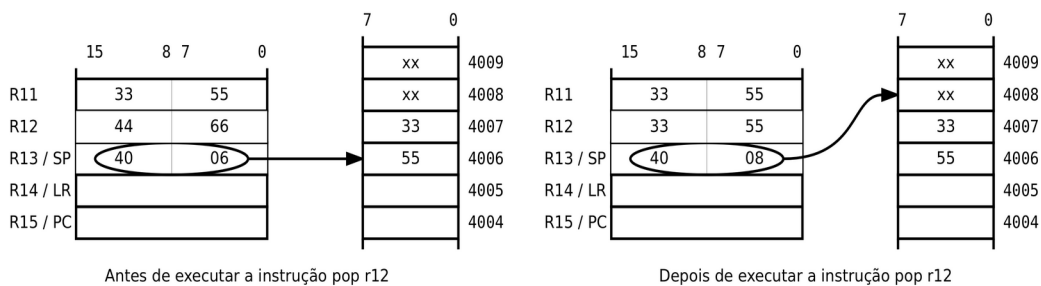
**pop rd** é equivalente a

```
ldr rd, [sp]
add sp, sp, 2
```



**Figura 4- Funcionamento da instrução push**

A figura 4 ilustra o efeito da execução da instrução **push r11**. Antes da sua execução o SP contém o endereço 0x4008. Ao executar a instrução **push** o processador começa por decrementar o SP de duas unidades passando para 0x4006. Em seguida escreve o *byte* menos significativo de R11 (0x55) na posição de endereço 0x4006 e o *byte* mais significativo de R11 (0x33) na posição de endereço 0x4007. A posição dos *bytes* segue o critério *little-ended*.



**Figura 5 - Funcionamento da instrução pop**

A figura 5 ilustra o efeito da execução da instrução **pop r12**. Antes da sua execução o SP contém o endereço 0x4006. Ao executar a instrução **pop**, o processador começa por transferir o conteúdo da posição de endereço 0x4006 (0x55) para o *byte* menos significativo de R12 e o conteúdo da posição de endereço 0x4007 (0x33) para o *byte* mais significativo de R12. Em seguida incrementa o SP para o endereço 0x4008. O conteúdo das posições de memória 0x4006 e 0x4007 não é alterado, mas estas posições ficam disponíveis para serem reutilizadas na próxima instrução **push**.