16 DESENVOLVIMENTO DE PROGRAMAS

O desenvolvimento de programas para computador é um processo iterativo que engloba a concepção e a escrita do programa em linguagem de programação, a tradução para código máquina e o teste. O desenvolvimento termina quando na fase de teste se verificar o correcto funcionamento do programa.

Os tradutores recebem os programas em ficheiros de texto simples, cujo conteúdo é formado unicamente pelos códigos dos caracteres visíveis, pelas tabulações e pelas mudanças de linha e de página. Estes ficheiros são produzidos em editores de programa. Os editores de documentos em linguagem natural não são adequados, porque introduzem informação relacionada com a formatação do texto que os tradutores não estão habilitados a processar.

A principal operação do tradutor consiste em traduzir os programas expressos em linguagem de programação para código máquina executável. O tradutor também pode produzir informação secundária para ajuda ao desenvolvimento. Designadamente, texto legível que mostra exaustivamente a correspondência entre as instruções da linguagem de programação e o código máquina produzido.

Um tradutor de linguagem *assembly* é geralmente designado por *assembler*. Os tradutores de linguagens de alto nível como C/C++ ou Java, são designados por compiladores.

O teste envolve a cópia do código máquina produzido pelo tradutor, para a memória dum sistema computacional – operação designada por carregamento – e a sua execução controlada.

Neste capítulo é descrito o processo de produção de programas para o microprocessador P16, escritos em linguagem *assembly*.

16.1 Linguagem *assembly* do P16

Um programa em linguagem *assembly* é formado por uma sequência de linhas de texto. Cada linha contém uma instrução.

```
        sub   r0, r0, 1
```

No exemplo, **sub** é a mnemónica da instrução e **r0, r0** e **1** são os operandos da instrução.

O P16 dispõe das instruções listadas na Tabela 16.1. A quantidade e o tipo dos operandos dependem da instrução em causa e podem ser registos do processador, constantes numéricas ou símbolos.

Se for necessário referenciar a instrução, por exemplo para saltar para ela desde outro ponto do programa, precede-se a instrução de uma *label*.

```
cycle:
        sub   r0, r0, 1
        . . .
        b     cycle
```

A *label* define um símbolo – no exemplo acima o símbolo **cycle**, cujo valor é o endereço da memória onde está alojado o código máquina da instrução **sub r0, r0, 1**.

Sintacticamente, é formada por uma palavra iniciada por uma letra seguida de mais letras ou dígitos e terminada pelo carácter '**:**'. Pode conter o carácter '**_**', tanto no início da palavra, como entre caracteres. O compilador distingue letras maiúsculas de minúsculas na definição de símbolo do tipo *label*.

Para melhor evidência a *label* costuma colocar-se na linha anterior à da instrução a que se refere.

Os comentários podem ser inseridos em qualquer lugar quando delimitados pelas marcas '***/****' *e* '****/***' ou entre o carácter '**;**' e o fim da linha.

```
cycle:
            sub   r0, r0, 1   ; decrement counter
            . . .
            b     cycle
```

ldr	rd, [pc, <imm6>]
pop	rd
push	rs
ldr	rd, [rn, <imm3>]
ldrb	rd, [rn, <imm3>]
ldr	rd, [rn, rm]
ldrb	rd, [rn, rm]
str	rs, [rn, <imm3>]
strb	rs, [rn, <imm3>]
str	rs, [rn, rm]
strb	rs, [rn, rm]
msr	cpsr, rs
msr	spsr, rs
mrs	rd, cpsr
mrs	rd, spsr

add	rd, rn, rm
sub	rd, rn, rm
adc	rd, rn, rm
sbc	rd, rn, rm
add	rd, rn, <imm4>
sub	rd, rn, <imm4>
cmp	rn, rm
bzs/beq	label
bzc/bne	label
bcs/blo	label
bcc/bhs	label
bge	label
blt	label
b	label
bl	label

and	rd, rn, rm
orr	rd, rn, rm
eor	rd, rn, rm
mvn/not	rd, rs
lsl	rd, rn, <imm4>
lsr	rd, rn, <imm4>
asr	rd, rn, <imm4>
ror	rd, rn, <imm4>
rrx	rd, rn
mov	rd, rs
movs	pc, lr
mov	rd, <imm8>
movt	rd, <imm8>

Tabela 15.1 – Instruções do P16

16.1.1 Directivas

Directivas de compilação são comandos que permitem ao programador controlar a operação do *assembler*.

Sintacticamente uma directiva é identificada por uma palavra chave precedida do carácter '**.**'. No texto do programa, uma directiva e os seus parâmetros ocupam a mesma posição da mnemónica da instrução e dos respetivos operandos.

Na linguagem *assembly* do P16 existem directivas para definir os dados do programa, para controlar a localização dos dados e do código máquina em memória e para definir símbolos.

.byte	<sequência de valores>	Define uma porção de memória que aloja a 'sequência de valores' composta por valores representados a 8 *bits*.
.word	<sequência de valores>	O mesmo que o anterior, para o caso dos valores serem representados a 16 *bits*.
.space	<dimensão> [,<valor inicial>]	Define um bloco de memória com a dimensão em *bytes* indicada pelo parâmetro dimensão. Se a definição de <valor inicial> for omitida o bloco será preenchido com zero.
.ascii	"<sequência de carateres>"	Define um bloco de memória iniciado com os caracteres que formam a 'sequência de carateres', codificados em ASCII.
.asciz	"<sequência de carateres>"	O mesmo que o anterior acrescido do valor zero no final da 'sequência de caracteres'.
.align	[<n>]	Avança o contador de localização até um valor múltiplo de 2^n. O novo valor terá zero nos 'n' *bits* de menor peso. A omissão de argumento é equivalente a **.align 1**.

Tabela 15.2 – Directivas para definição de dados do programa

Utilização da directiva **.word** para definição de uma variável de 16 *bits* identificada pelo símbolo **counter** iniciada com o valor zero.

```
counter:
        .word       0
```

Utilização da directiva **.byte** para definição de um *array* de valores representados a 8 *bits* com três posições, iniciadas com os valores 3, 4 e 5, sucessivamente.

```
array:
        .byte       3, 4, 5
```

Utilização da directiva **.asciz** para definição de um *array* de caracteres iniciado com a *string* **"Portugal"**, no formato da linguagem C. Neste formato cada posição do *array* guarda o código de um carácter, começando no endereço mais baixo e pela ordem de escrita. A terminação da string é assinalada com o valor zero na posição a seguir à do último carácter. Neste exemplo são ocupadas nove posições de memória, oito para os códigos dos caracteres e uma para o terminador.

```
message:
        .asciz      "Portugal"
```

.section	<nome>	Define uma secção, designando-a com o 'nome' indicado.
.text		Define uma secção com o nome '.text'.
.data		Define uma secção com o nome '.data'.

Tabela 15.3 – Directivas para definição de secções

.equ	<nome>, <valor>	Define o símbolo 'nome' como sendo equivalente a 'valor'.

Tabela 15.4 – Directiva para definição de símbolos

16.1.2 Símbolos

Existem duas formas de definir símbolos: através de *labels* ou através da directiva **.equ**. No caso das *labels* o símbolo é equivalente ao endereço da instrução ou da variável indicada. No caso da directiva **.equ** o símbolo é equivalente ao valor da constante ou expressão associada.

```
.equ MASK, 0b00001110
```

Neste exemplo, a directiva **.equ** é usada para definir o símbolo **MASK** como equivalente ao valor binário **1110**.

16.1.3 Contador de localização

O contador de localização é uma variável interna do *assembler* e contém o endereço onde o código da instrução corrente pode eventualmente ser carregado em memória. Quando a tradução do programa começa, esta variável é inicializada a zero. À medida que as instruções ou directivas são processadas, o contador de localização é aumentado da dimensão necessária para armazenar o código máquina da instrução ou o conteúdo da variável.

Existe um contador de localização para cada secção.

A linguagem *assembly* do P16 usa o símbolo '**.**' (um ponto isolado) como identificador do contador de localização. No contexto da linguagem *assembly* este símbolo substitui a *label* referente à instrução ou directiva corrente.

```
b     .
```

No exemplo, a instrução *branch* realiza um salto para a posição onde ela própria se encontra, confinando o processamento à execução repetitiva desta instrução.

16.1.4 Secções

As secções são zonas de memória que alojam elementos do programa – o código de instruções ou os dados do programa, segundo critérios de afinidade. O caso mais simples consiste em agrupar o código das instruções numa secção e as variáveis noutra secção.

Antes de especificar qualquer instrução ou directiva deve-se definir a secção que as vai conter. A secção corrente é definida pela directiva **.section** ou pelas directivas especificas **.text** ou **.data**.

O programa seguinte é composto pela secção **.data** onde se alojam as variáveis **x**, **y** e **z** e pela secção **.text** onde se aloja o código máquina do programa. A secção **.data** está localizada no endereço 0x20a0 e tem dimensão seis. A secção **.text** está localizada no endereço 0xb000 e tem a dimensão 22 (0x16). Os valores dos endereços usados neste exemplo são arbitrários. Conforme veremos mais adiante, os endereços das secções são atribuídos em fase posterior à da escrita do programa (Secção 16.4.1).

```
1                           .data
2                    x:
3 20A0 1E00                 .word 30
4                    y:
5 20A2 0400                 .word 4
6                    z:
```

```
 7 20A4 0000                .word 0
 8
 9                          .text
10                      main:
11 B000 700C                ldr   r0, addr_x
12 B002 0000                ldr   r0, [r0]
13 B004 610C                ldr   r1, addr_y
14 B006 1100                ldr   r1, [r1]
15 B008 8080                add   r0, r0, r1
16 B00A 410C                ldr   r1, addr_z
17 B00C 1020                str   r0, [r1]
18 B00E 0FB7                mov   pc, lr
19
20                      addr_x:
21 B010 A020                .word x
22                      addr_y:
23 B012 A220                .word y
24                      addr_z:
25 B014 A420                .word z
```

Uma secção pode ser fragmentada ao longo do texto do programa. Por exemplo, para que as variáveis possam ser definidas junto ao código das funções que as utilizam, mas alojadas na secção **`.data`**, haverá fragmentos de **`.text`** entrecortados por fragmentos de **`.data`**.

Durante a compilação, os fragmentos de uma secção são concatenados, pela ordem em que aparecem ao longo da descrição do programa para formar a composição final de cada uma delas.

```
 1                      /*-------------------------------------------
 2                          Function strtok
 3                      */
 4                          .data
 5                      ptr:
 6 1000 0000                .word 0
 7
 8                          .text
 9                      strtok:
10 3000 200C                ldr   r0, addr_ptr
11 3002 0000                ldr   r0, [r0]
12                          ; ...
13 3004 0FB7                mov   pc, lr
14
15                      addr_ptr:
16 3006 0010                .word ptr
17
18                      /*-------------------------------------------
19                          Function accumulate;
20                      */
21                          .data
22                      counter:
23 1002 00                  .byte 0
24
25                          .text
26                      accumulate:
27 3008 200C                ldr   r0, addr_counter
28 300A 0008                ldrb  r0, [r0]
29                          ; ...
30 300C 0FB7                mov   pc, lr
```

```
31
32                      addr_counter:
33 300E 0210                  .word counter
```

Neste exemplo, mostram-se os endereços calculados de forma coerente, embora tendo por base valores novamente arbitrários. A variável **ptr** é alojada na secção **.data** e a sua definição surge junto da função **strtok** que a utiliza. O mesmo acontece com a variável **counter** e a função **accumulate**. A secção **.text** está separada em dois fragmentos o primeiro contém o código da função **strtok** e o segundo contém o código da função **accumulate**. Na memória as duas variáveis ocupam posições contíguas – **ptr** ocupa as posições de endereço 0x1000 e 0x1001 e **counter** a posição de endereço 0x1002. O código das funções **strtok** e **accumulate** ocupam também zonas de memória contíguas, respectivamente, a gama de endereços 0x3000 a 0x3007 e a gama de endereços 0x3008 a 0x300f.

16.2 Convenções de programação

Para reutilizar as mesmas funções em diversos programas e para que essas funções possam ser escritas por programadores diferentes, é conveniente usar regras que facilitem a interoperabilidade. Designadamente, tipos de variáveis, parâmetros de funções, retorno de valor de funções e vocação dos registos.

Nos exemplos de programa apresentados são utilizadas as convenções descritas seguidamente.

16.2.1 Tipos

Os tipos de dados utilizados são tipos numéricos simples ou *arrays* destes valores. Os tipos numéricos são codificados em código binário natural ou em código dos complementos para dois, usando 8, 16 ou 32 *bits*.

char, int8_t	- inteiro relativo a 8 *bits*	**uint8_t**	- inteiro natural a 8 *bits*
int, int16_t	- inteiro relativo a 16 *bits*	**uint16_t**	- inteiro natural a 16 *bits*
long, int32_t	- inteiro relativo a 32 *bits*	**uint32_t**	- inteiro natural a 32 *bits*

16.2.2 Parâmetros

As funções que comportam parâmetros recebem os argumentos nos registos do processador, ocupando a quantidade necessária, por ordem: **r0**, **r1**, **r2** e **r3**.

```
void f(uint8_t a,  uint16_t b,  int8_t c,  int16_t d)
            r0           r1          r2          r3
```

Se os argumentos ocuparem mais que os quatro registos, os restantes são passados no *stack*. Sendo o argumento mais à direita o primeiro a ser empilhado.

```
void f(uint8_t a,  uint16_t b,  int8_t c,  int16_t d,  int8_t e,  int16_t f)
            r0           r1          r2          r3      stack       stack
```

Se o tipo do parâmetro for um valor codificado a 32 *bits* a passagem utiliza dois registos consecutivos. Cabendo ao registo de menor índice a parte de menor peso do parâmetro.

```
void f(uint8_t a,   uint32_t b,  int8_t c)
            r0          r2:r1       r3
```

Se o parâmetro for um *array*, independentemente do tipo dos elementos, o que é passado como argumento é o endereço da primeira posição do *array*.

```
void f(uint8_t array[], int8_t dim)
            r0                  r1
```

16.2.3 Valor de retorno

O valor de retorno de uma função, caso exista, é devolvido no registo **r0**. Se for um valor representado a 32 *bits* é devolvido no par de registos **r1:r0**, com a parte de menor peso em **r0**.

16.2.4 Preservação de registos

Uma função pode usar os registos de **r0** a **r3** sem ter de preservar o seu conteúdo original. Os restantes registos – **r4** a **r12** – devem ser preservados.

Na perspectiva de função chamadora, uma função chamada pode modificar os registos **r0** a **r3**, **lr** e **cpsr**;

Na perspectiva da função chamada, os conteúdos dos registos de **r4** a **r12** têm de ser mantidos, independentemente de serem ou não utilizados.

16.3 Estrutura dos programas

Para sustentar a adequada versatilidade na utilização do espaço de endereçamento, um programa completo é organizado em secções. As secções mais comuns são: secção para código de inicialização – secção **.startup**; secção para o *stack* – **.stack**; secção para alojar as variáveis – **.data** e secção para o código das instruções do programa – **.text**.

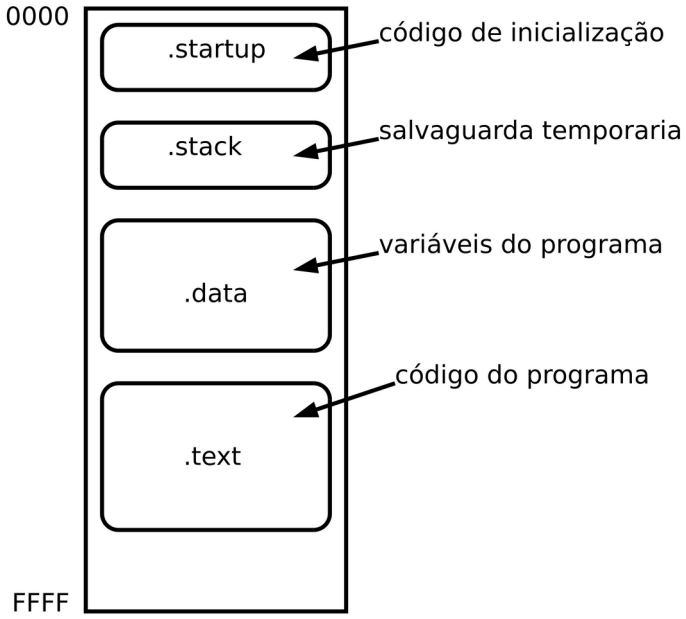


Figura 15.1 – Composição de um programa por secções

Após a acção *reset* o P16 passa a executar código a partir do endereço 0x0000. Por isso, a secção **.startup** deve ser localizada no endereço zero (é o que acontece por omissão, se, como é uso, for esta a primeira secção do ficheiro fonte). As restantes secções podem ser localizadas em qualquer endereço do espaço de endereçamento.

```
1           .section .startup
2           b     _start
3           b     .
4
5     _start:
6           mov   sp, stack_top
7           bl    main
8           b     .
9
10          .section    .stack
11          .space      128
12    stack_top:
13
14          .data
15          ;  ... variáveis do programa
16
17          .text
18    main:
19          ; ... main code
20          mov   pc, lr
```

Como o endereço 0x0002 é reservado ao atendimento de interrupções, a primeira instrução a executar, localizada no endereço 0x0000, é **b _start** – para prosseguir a execução noutro local. Mesmo quando não se utilizam as interrupções, o endereço 0x0002 deve ser preenchido pela instrução **b .** (surge necessariamente na linha 3). Se, por algum erro, o processador atender uma interrupção inesperada o processamento não se descontrola – o processador ficará retido a executar indefinidamente esta instrução.

Para suporte à execução do programa, entendido como uma cadeia hierárquica de chamadas a funções, conforme ocorre na linguagem C, é necessário definir a área de memória dedicada ao *stack* e a inicialização do registo *stack pointer* (SP) antes de se chamar a função **main**. No exemplo, a área de memória para *stack* é definida com a directiva **.space** com a dimensão de 128 *bytes,* confinada entre o início da secção **.stack** e a *label* **stack_top.** O registo **sp** é inicializado com o valor da *label* **stack_top** – que corresponde ao endereço a seguir ao endereço mais alto da secção **.stack** – porque no P16 o empilhamento evolui no sentido descendente com decremento prévio do apontador (*full descending stack*).

A instrução **b .** que vemos na linha 8, mantém a execução controlada no caso da função **main** retornar.

No programa anterior existem duas limitações quanto à localização das secções:

- a primeira limitação está na forma de iniciar o registo SP, a utilização da instrução **mov sp, stack_top** limita o topo do *stack* a endereços inferiores a 256. A solução geral para a iniciação do SP, implementada nas linhas 5, 12 e 13 do programa seguinte, utiliza o método descrito no Capítulo13, Secção 4.4;

- a segunda limitação é devida ao alcance da instrução **bl**. Esta instrução utiliza endereçamento relativo, limitado a +1022 ou -1024 posições de memória. Se a distância entre **bl main** e a *label* **main** for superior a estes valores, esta solução não pode ser empregada. Esta limitação pode ser suprimida manipulando directamente o PC – simulando a instrução **bl main** – como no código apresentado nas linhas 6 a 9 do programa seguinte.

```
1    .section .startup
2          b     _start
3          b     .
4    _start:
5          ldr   sp, addr_stack_top
6          ldr   r0, addr_main
7          mov   r1, pc
8          add   lr, r1, 4
9          mov   pc, r0
10         b     .
11
12   addr_stack_top:
13         .word stack_top
14   addr_main:
15         .word main
16
17         .section    .stack
18         .space      1024
19   stack_top:
```

16.4 Ambiente de programação

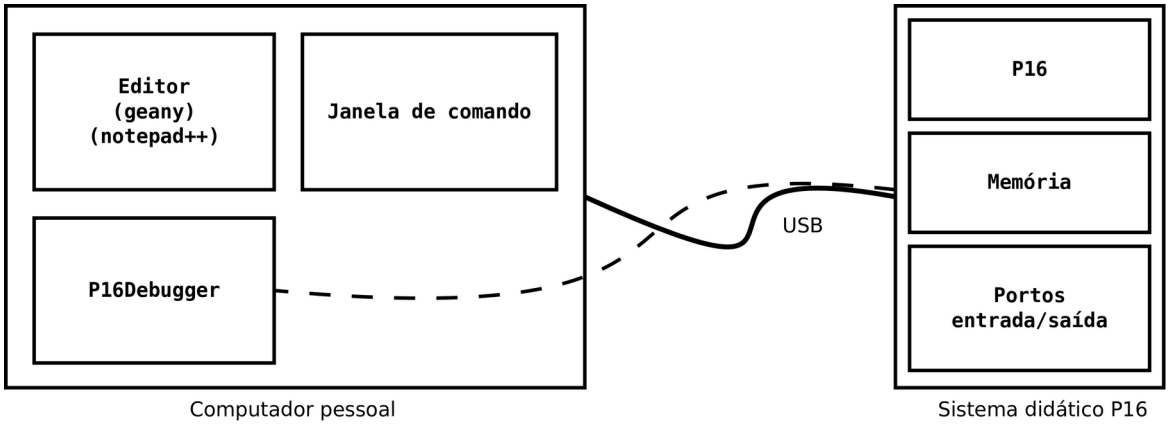


Figura 15.2 – Cenário de desenvolvimento de programas

Edição

Os programas são produzidos com recurso a um editor de programa e armazenados em ficheiros de texto simples, com extensão **.s**. Exemplos de editores de programa: geany, notepad++, SciTE.

Tradução

A tradução de ficheiro em linguagem *assembly* P16 para código máquina é realizada pelo *assembler* **PAS**. O manuseamento deste programa é feito através da janela de comando.

Teste

No teste estão envolvidos *debugger* – **P16Debugger** e um sistema computacional baseado no P16.

O **P16Debugger** é uma aplicação para computador pessoal (Microsoft Windows) que permite invocar as acções que são necessárias durante o teste de programas sobre sistemas baseados no processador P16. Este programa permite realizar as seguintes operações:

- transferir o código máquina do programa para a memória do sistema computacional;
- visualizar o conteúdo da memória e dos registos internos do processador;
- executar o programa controladamente – uma instrução de cada vez, uma rotina, ou correr a execução até um endereço pré-definido;

O **SDP16** é um sistema computacional autónomo baseado no P16 com suporte para acesso à memória e aos registos internos e controlo de execução dos programas. A sua operação pode ser feita remotamente, usando um *debugger* (P16Debugger) ou localmente por meio de botões e visualizadores LED. Liga fisicamente ao computador pessoal através de cabo USB que serve como alimentação de energia e para comunicação.

O **P16Simulator** é um programa simulador de sistema computacional baseado no P16. É usado em substituição de um sistema real e permite o teste de programas usando o computador pessoal como único recurso material. Simula o processador P16 e a memória, virtualizando todo o espaço de endereçamento. Não simula periféricos nem o sistema de interrupção. A sua operação é feita remotamente através de *debugger* (P16Debugger).

16.4.1 *Assembler* PAS

O programa PAS é um *assembler* para o processador P16 que, a partir de um ficheiro de texto em linguagem *assembly*, produzido num editor de programas, gera ficheiros com o código máquina específico do P16.

O PAS é um *assembler* de uma única passagem que, por razões didácticas, processa apenas um ficheiro fonte e localiza o programa. A localização consiste em atribuir endereço absoluto ao programa, tarefa que não é normalmente realizada pelo *assembler*.

No PAS não é possível definir símbolos iguais a mnemónicas de instruções. Por exemplo, não pode existir um símbolo "b" porque coincide com a mnemónica da instrução *branch*.

O PAS é invocado na janela de comando segundo a seguinte sintaxe:

```
c:> pas [options] <program filename>
options:
      --verbose
      -h, --help
      -v, --version
      -o, --output filename
      -s, --section sectionname=address
```

O ficheiro com o texto do programa (**<program filename>**) tem normalmente a extensão **s**.

Os erros e avisos são assinalados na janela de comando usada na invocação.

No caso do compilador não detectar erros, são gerados ficheiros com o nome (**filename**) definido na opção **--output** e com as extensões **lst** e **hex**. Se esta opção for omitida, os ficheiros produzidos

terão o mesmo nome base que ficheiro fonte.

A opção **`--section`** permite definir o endereço de localização das secções.

Localização das secções

A definição da localização em memória de cada secção pode ser explícita ou implícita.

Na localização explícita o programador indica o endereço da secção através da opção **`--section`** na invocação do *assembler*.

Na localização implícita a secção é localizada no endereço a seguir ao último endereço da secção anterior, pela ordem em que surgem no ficheiro fonte que contém o programa.

No caso de não ser explicitada a localização da primeira secção definida no programa, esta é localizada no endereço 0x0000.

No caso da secção ser fragmentada, a localização implícita é definida pela posição do primeiro fragmento.

O endereço inicial de uma secção é localizado automaticamente num endereço par.

16.4.2 Exemplo

Considere-se o seguinte programa como conteúdo do ficheiro **`multiply.s`**.

```
 1       .section .startup
 2       b     _start
 3       b     .
 4 _start:
 5       ldr   sp, addr_stack_top
 6       ldr   r0, addr_main
 7       mov   r1, pc
 8       add   lr, r1, 4
 9       mov   pc, r0
10       b     .
11
12 addr_stack_top:
13       .word stack_top
14 addr_main:
15       .word main
16
17       .section .stack
18 stack:
19       .space      1024
20 stack_top:
21
22 /*-------------------------------------------------
23 uint8_t m = 20, n = 3;
24 uint16_t p, q;
25 */
26       .data
27 m:
28       .byte 20
29 n:
30       .byte 3
31 p:
```

```
32        .word 0
33  q:
34        .word 0
35  /*-------------------------------------------------
36  int main() {
37        p = multiply(m, n);
38        q = multiply(4, 7);
39  }
40  */
41        .text
42  main:
43        push  lr
44        ldr   r0, addr_m
45        ldrb  r0, [r0]
46        ldr   r1, addr_n
47        ldrb  r1, [r1]
48        bl    multiply
49        ldr   r1, addr_p
50        str   r0, [r1]
51        mov   r0, 4
52        mov   r1, 7
53        bl    multiply
54        ldr   r1, addr_q
55        str   r0, [r1]
56        pop   pc
57
58  addr_m:
59        .word m
60  addr_n:
61        .word n
62  addr_p:
63        .word p
64  addr_q:
65        .word q
66
67  /*-------------------------------------------------
68  int multiply(<r0> int multiplicand, <r1> int multiplier) {
69        <r2> int product = 0;
70        while (multiplier > 0) {
71              product += multiplicand;
72              multiplier--;
73        }
74        <r0> return product;
75  }
76  */
77  multiply:
78        mov   r2, 0
79        add   r1, r1, 0
80        bzs   while_end
81  while:
82        add   r2, r2, r1
83        sub   r1, r1, 1
84        bzc   while
85  while_end:
86        mov   r0, r2
87        mov   pc, lr
```

O comando

```
pas simple.s -s .data=0x500 -s .text=0x600
```

traduz e localiza o programa. Com as opções **-s** definem-se os endereços das secções **.data** e **.text**. Por omissão, a secção **.startup** é localizada no endereço 0x0000 e a secção .**stack** no endereço 20 (0x0014).

Se o programa tiver erros, estes serão assinalados na janela de comandos. Foram introduzidos dois erros de sintaxe apenas para exemplificar.

```
multiply.s (46):        ld    r0, addr_m
----------------        ^^
ERROR!      syntax error

multiply.s (48):        ldr   addr_n
----------------              ^^^^^^
ERROR!      syntax error
```

Se o programa fonte não tiver erros, são produzidos dois ficheiros adicionais **multiply.lst** com informação legível e **multiply.hex** com o código máquina.

A emissão de avisos não impede a geração do código binário como no seguinte caso:

```
multiply.s (86):        sub   r1, r1, 17
----------------                      ^^
WARNING! Expression's value = 17 (0x11) not encodable in 4 bits, truncate to 1 (0x1)
```

Faz parte de uma boa prática de programação corrigir o programa até suprimir a emissão de mensagens de aviso.

Por uma questão de organização, é conveniente criar especificamente uma directoria para alojar os ficheiros relacionados com um dado programa. No exemplo seguinte a directoria **multiply** aloja todos os ficheiros relacionados com este programa: **multiply.s**, **multiply.lst** e **multiply.hex**.

```
disciplinas
 |-- pe
 |-- ss
 |-- ac
      |-- documents
      |-- p16_code
           |-- divide
           |-- multiply
                |-- multiply.s
                |-- multiply.lst
                |-- multiply.hex
```

Em seguida apresenta-se o conteúdo do ficheiro **lst**. Este contém a tabela de secções, a tabela de símbolos e a listagem das instruções.

Na tabela de secções listam-se as secções existentes, as gamas de endereços que ocupam e as respectivas dimensões.

Na tabela de símbolos listam-se os símbolos definidos através de *label* ou através da directiva **.equ**. Por cada símbolo é dada a seguinte informação: identificador, tipo, valor associado e secção a que

pertence.

Na listagem das instruções, são apresentados do lado esquerdo, na primeira coluna o número da linha do ficheiro fonte, na segunda coluna os endereços da memória e na terceira coluna o respectivo conteúdo.

Na arquitectura do P16 as palavras formadas por dois *bytes* – designadas por *word* – ocupam duas posições de memória, o *byte* de menor peso toma a posição de endereço menor e o *byte* de maior peso, a posição de endereço maior – *little ended format*.

O conteúdo da memória – código das instruções ou valor das variáveis – é escrito na terceira coluna como uma sequência de *bytes* pela ordem dos endereços que ocupam na memória. Por exemplo, na linha 7 o código máquina da instrução **`mov r1, pc`** que ocupa os endereços 8 e 9, e tem o valor 0xb781, é representado pela sequência de *bytes* **`81 B7`**. Na linha 29, a variável **`m`** do, tipo **`.byte`**, ocupa o endereço 0x500 e o seu valor é 20 (0x14).

```
P16 assembler v1.2 (Apr  8 2019)    multiply.lst      Sun Apr 14 09:28:15 2019

Sections
Index   Name            Addresses       Size
0       .startup        0000 - 0013     0014 20
1       .stack          0014 - 0413     0400 1024
2       .data           0414 - 0419     0006 6
3       .text           041A - 044D     0034 52

Symbols
Name                    Type      Value      Section
while                   LABEL     0444 1092  .text
addr_n                  LABEL     0438 1080  .text
multiply                LABEL     043E 1086  .text
addr_m                  LABEL     0436 1078  .text
q                       LABEL     0418 1048  .data
p                       LABEL     0416 1046  .data
addr_main               LABEL     0012 18    .startup
line#3                  LABEL     0002 2     .startup
stack                   LABEL     0014 20    .stack
addr_q                  LABEL     043C 1084  .text
_start                  LABEL     0004 4     .startup
stack_top               LABEL     0414 1044  .stack
addr_stack_top          LABEL     0010 16    .startup
line#10                 LABEL     000E 14    .startup
main                    LABEL     041A 1050  .text
while_end               LABEL     044A 1098  .text
addr_p                  LABEL     043A 1082  .text
m                       LABEL     0414 1044  .data
n                       LABEL     0415 1045  .data

Code listing
   1                            .section .startup
   2 0000 0158                  b     _start
   3 0002 FF5B                  b     .
   4                            _start:
   5 0004 5D0C                  ldr   sp, addr_stack_top
   6 0006 500C                  ldr   r0, addr_main
   7 0008 81B7                  mov   r1, pc
   8 000A 1EA2                  add   lr, r1, 4
```

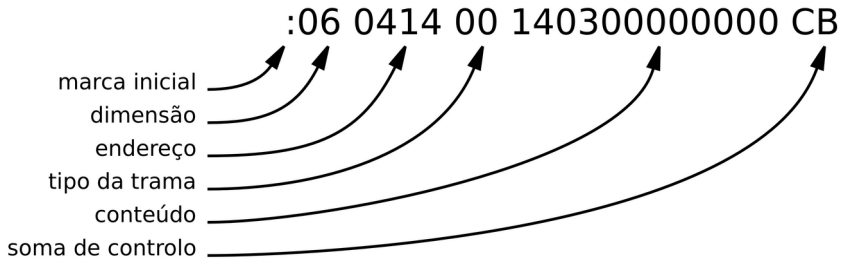
```
 9 000C 0FB0                mov   pc, r0
10 000E FF5B                b     .
11
12                      addr_stack_top:
13 0010 1404                .word stack_top
14                      addr_main:
15 0012 1A04                .word main
16
17                          .section .stack
18                      stack:
19 0014 00000000            .space      1024
19 0018 00000000
19 001C 00000000
19 0020 00000000
20                      stack_top:
21
22                      /*-------------------------------------------------
23                      uint8_t m = 20, n = 3;
24                      uint16_t p, q;
25                      */
26                          .data
27                      m:
28 0414 14                  .byte 20
29                      n:
30 0415 03                  .byte 3
31                      p:
32 0416 0000                .word 0
33                      q:
34 0418 0000                .word 0
35                      /*-------------------------------------------------
36                      int main() {
37                          p = multiply(m, n);
38                          q = multiply(4, 7);
39                      }
40                      */
41                          .text
42                      main:
43 041A 0E24                push  lr
44 041C C00C                ldr   r0, addr_m
45 041E 0008                ldrb  r0, [r0]
46 0420 B10C                ldr   r1, addr_n
47 0422 1108                ldrb  r1, [r1]
48 0424 0C5C                bl    multiply
49 0426 910C                ldr   r1, addr_p
50 0428 1020                str   r0, [r1]
51 042A 4060                mov   r0, 4
52 042C 7160                mov   r1, 7
53 042E 075C                bl    multiply
54 0430 510C                ldr   r1, addr_q
55 0432 1020                str   r0, [r1]
56 0434 0F04                pop   pc
57
58                      addr_m:
59 0436 1404                .word m
60                      addr_n:
61 0438 1504                .word n
62                      addr_p:
63 043A 1604                .word p
```

```
  64                    addr_q:
  65 043C 1804                .word q
  66
  67                    /*-------------------------------------------------
  68                    int multiply(<r0> int multiplicand, <r1> int multiplier)
{
  69                          <r2> int product = 0;
  70                          while (multiplier > 0) {
  71                                product += multiplicand;
  72                                multiplier--;
  73                          }
  74                          <r0> return product;
  75                    }
  76                    */
  77                    multiply:
  78 043E 0260              mov   r2, 0
  79 0440 11A0              add   r1, r1, 0
  80 0442 0340              bzs   while_end
  81                    while:
  82 0444 A280              add   r2, r2, r1
  83 0446 91A8              sub   r1, r1, 1
  84 0448 FD47              bzc   while
  85                    while_end:
  86 044A 00B1              mov   r0, r2
  87 044C 0FB7              mov   pc, lr
```

O ficheiro de extensão **hex**, em formato Intel HEX[1], contém apenas o código binário das instruções e os valores iniciais das variáveis, com a indicação dos endereços de memória onde serão carregados.

```
:100000000158FF5B5D0C500C81B71EA20FB0FF5B67
:0400100014041A04B6
:0604140014030000000CB
:10041A000E24C00C0008B10C11080C5C910C1020C1
:10042A0040607160075C510C10200F04140415041D
:10043A0016041804026011A00340A28091A8FD4787
:04044A0000B10FB737
:00000001FF
```

O seu conteúdo é composto por tramas, formadas por uma marca inicial, a dimensão dos dados, o endereço onde os dados serão carregados, o tipo da trama, os dados contidos na trama e um código para detecção de eventual corrupção dos dados – soma de controlo.



A última trama (:00 0000 01 FF) tem dimensão zero, invoca virtualmente o endereço zero e é do tipo "01" – *end of file*, o que conjuntamente suscita a soma de controlo "FF". Serve para terminar o ficheiro.

1 - https://en.wikipedia.org/wiki/Intel_HEX

16.5 Recomendações para escrita em linguagem a*ssembly*

Na escrita de programas em geral, usam-se convenções de formatação para facilitar a leitura. Em seguida lista-se um conjunto de regras geralmente utilizadas na programação em linguagem *assembly* e que são aplicadas nos programas de exemplo.

- O texto do programa é escrito em letra minúscula, excepto os identificadores de constantes.
- Nos identificadores formados por várias palavras usa-se como separador o carácter '_' (sublinhado).
- O programa é disposto na forma de uma tabela de quatro colunas. Na primeira coluna insere-se apenas a *label* (se existir), na segunda coluna a mnemónica da instrução ou a directiva, na terceira coluna os parâmetros da instrução ou da directiva e na quarta coluna os comentários até ao fim da linha (começados por '**;**' ou envolvidos por '**/* */**'). Cada linha contém apenas uma *label*, uma instrução ou uma directiva.
- Para definir as colunas deve usar-se o carácter TAB configurado com a largura de oito espaços.
- As linhas com *label* não devem conter menhum outro elemento. Isso permite usar *labels* compridas sem desalinhar a tabulação e cria separações na sequência de instruções.

16.6 Exemplos de programação

Os exemplos de programação que se seguem devem executar sem erros, bastando para isso juntar a definição das secções **`.startup`** e **`.stack`** apresentadas na Secção 16.3 deste documento.

16.6.1 Multiplicação

Nos processadores que não dispõem de instrução de multiplicação, esta operação poderá ser realizada por programação. Um algoritmo comummente utilizado baseia-se na aplicação das seguintes expressões:

$$P = M \,.\, m = M \,.\, (2^{n-1} \,.\, m_{n-1} + 2^{n-2} \,.\, m_{n-2} + \ldots + 4 \,.\, m_2 + 2 \,.\, m_1 + m_0) =$$

$$= M \,.\, 2^{n-1} \,.\, m_{n-1} + M \,.\, 2^{n-2} \,.\, m_{n-2} + \ldots + M \,.\, 4 \,.\, m_2 + M \,.\, 2 \,.\, m_1 + M \,.\, m_0$$

onde $\mathbf{m_x}$ representa cada um dos dígitos do multiplicador **m**, expresso em binário, com **x** a variar entre 0 e n – 1, sendo n o número de *bits* do multiplicador **m**.
A última expressão é um somatório de parcelas da forma $\mathbf{M \,.\, 2^x \,.\, m_x}$. A multiplicação de **M** por uma potência inteira de 2 ($\mathbf{M.2^x}$), pode ser realizada por uma instrução de deslocamento para a esquerda do multiplicando **M**. O produto de $\mathbf{M.2^x}$ por $\mathbf{m_x}$, será igual a $\mathbf{M.2^x}$ se o *bit* de índice **x** do multiplicador **m** for igual a um, ou será zero, no caso de $\mathbf{m_x}$ ser zero. A sequência conjunta destas três acções – **DESLOCAMENTO** de **M**, **PRODUTO LÓGICO** pelo *bit* $\mathbf{m_x}$ e **ADIÇÃO** deste resultado parcial, em somatório, até final – justifica a designação *shift-and-add*, pela qual o algoritmo ficou conhecido (podendo também surgir a designação *add-shift*).

Em seguida apresenta-se a função **`multiply`** em linguagem C que aplica o princípio enunciado acima. Na sua programação em linguagem *assembly* assume-se que os parâmetros da função, multiplicando e multiplicador, são previamente colocados nos registos R0 e R1, respectivamente. O resultado da função

– produto de multiplicando por multiplicador – é deixado no registo R0. Considera-se que o valor dos argumentos nunca é superior a 255 e representam números naturais.

```
1   /*-------------------------------------------------------
2   uint8_t m = 20, n = 3;
3   uint16_t p, q;
4   */
5         .data
6   m:
7         .byte 20
8   n:
9         .byte 3
10  p:
11        .word 0
12  q:
13        .word 0
14
15  /*-------------------------------------------------------
16  int main() {
17        p = multiply(m, n);
18        q = multiply(4, 7);
19  }
20  */
21        .text
22  main:
23        push  lr
24        ldr   r0, addr_m
25        ldr   r0, [r0]
26        ldr   r1, addr_n
27        ldr   r1, [r1]
28        bl    multiply
29        ldr   r1, addr_p
30        str   r0, [r1]
31        mov   r0, 4
32        mov   r1, 7
33        bl    multiply
34        ldr   r1, addr_q
35        str   r0, [r1]
36        pop   pc
37
38  addr_m:
39        .word m
40  addr_n:
41        .word n
42  addr_p:
43        .word p
44  addr_q:
45        .word q
46
47  /*-------------------------------------------------------------
48  int multiply(<r0> int multiplicand, <r1> int multiplier)) {
49        <r2> int product = 0;
50        while ( multiplier > 0 ) {
51              if ( (multiplier & 1) != 0 )
52                    product += multiplicand;
53              multiplier >>= 1;
54              multiplicand <<= 1;
```

```
55        }
56        <r0> return product;
57  }
58  */
59  multiply:
60        mov   r2, 0       ; <r2> int product = 0;
61  mul_while:
62        add   r1, r1, 0   ; while ( multiplier > 0 )
63        beq   mul_return
64        lsr   r1, r1, 1   ; if ( (multiplier & 1) != 0 )
65        bcc   mul_if_end
66        add   r2, r2, r0  ; product += multiplicand;
67  mul_if_end:
68        lsl   r0, r0, 1   ; multiplicand <<= 1;
69        b     mul_while
70  mul_return:
71        mov   r0, r2
72        mov   pc, lr      ; <r0> return product;
```

16.6.2 Divisão

O programa seguinte implementa o algoritmo de divisão *shift-and-subtract*. Este algoritmo executa um número de iterações igual ao número de *bits* dos operandos.

```
 1 /*---------------------------------------------------------------
 2 int x = 30, y = 4, z;
 3
 4 int main() {
 5       z = div(x, y);
 6 }
 7 */
 8       .data
 9 x:
10       .word 30
11 y:
12       .word 4
13 z:
14       .word 0
15
16       .text
17 main:
18       push  lr
19       ldr   r0, addr_x
20       ldr   r0, [r0]
21       ldr   r1, addr_y
22       ldr   r1, [r1]
23       bl    divide
24       ldr   r1, addr_z
25       str   r0, [r1]
26       pop   pc
27
28 addr_x:
29       .word x
30 addr_y:
31       .word y
```

```
32 addr_z:
33       .word z
34
35 /*-------------------------------------------------------------------------
36 <r0> uint16_t int divide(<r0> uint16 dividend, <r1> uint16_t divisor){
37       <r2> uint16_t i = 16;
38       <r3> uint16_t remainder = 0, <r4> quotient = 0;
39       do {
40             uint16 dividend_msb = dividend >> 15;
41             dividend <<= 1;
42             rest = ( remainder << 1) | dividend_msb;
43             quotient <<= 1;
44             if (remainder >= divisor) {
45                   remainder -= divisor;
46                   quotient += 1;
47             }
48       } while (--i > 0);
49       return quotient (r0);
50 }
51 */
52
53 divide:
54       push  r4
55       mov   r3, 0       ; remainder = 0;
56       mov   r4, 0       ; quotient = 0;
57       mov   r2, 16      ; uint16_t i = 16;
58 div_while:              ; uint16 dividend_msb = dividend >> 15;
59       lsl   r0, r0, 1   ; dividend <<= 1;
60       adc   r3, r3, r3  ; rest = (rest << 1) | dividend_msb;
61       lsl   r4, r4, 1   ; quotient <<= 1;
62       cmp   r3, r1      ; if ( remainder >= divisor) {
63       blo   div_if_end
64       sub   r3, r3, r1  ; remainder -= divisor;
65       add   r4, r4, 1   ; quotient += 1;
66 div_if_end:
67       sub   r2, r2, 1   ; } while (--i > 0);
68       bne   div_while
69       mov   r0, r4      ; return quotient;
70       pop   r4
71       mov   pc, lr
```

Na função **main**, linhas 17 a 26, é feita a invocação da função **div**. Nas linhas 19 e 20 é passado o valor da variável **x** como primeiro argumento de **divide** através de **r0**. Na linha 19 começa por se carregar o endereço da variável **x**. Na linha 20, é efectivamente carregado em **r0** o valor **x**. O carregamento do segundo argumento, valor da variável **y** é realizado da mesma forma nas linhas 21 e 22.

Depois de retornar, o valor de retorno da função **divide** encontra-se em **r0**. Este valor é guardado na variável **z** pela instrução **str r0, [r1]**. O endereço de **z** foi carregado em **r1** pelo mesmo método que os endereços das variáveis **x** e **y**.

Na função **divide** a instrução **push** da linha 54 começa por salvar o valor do registo **r4** no *stack*. Este registo vai ser usado na função e para cumprir o protocolo de interoperabilidade deve ser preservado. Na linha 70 a instrução **pop** restaura o valor original do registo **r4**, descarregando do *stack* o valor que tinha sido salvo pela instrução da linha 54.

Entre as linhas 55 e 68 implementa-se o algoritmo da operação de divisão descrito em linguagem C nas linhas 36 a 50. Nesta descrição, junto das variáveis locais e dos parâmetros são anotados entre '<' e '>' os registos que os suportam.

Nas linhas 55, 56 e 57 iniciam-se respectivamente as variáveis **`remainder`**, **`quotient`** e **`i`**.

Na linha 59 a instrução **`lsl r0, r0, 1`** realiza as duas instruções C descritas nas linhas 41 e 42. Desloca **`dividend`** uma posição para a esquerda – operação da linha 41 – e insere o *bit* mais significativo no registo Carry do processador – operação da linha 42.

A instrução **`adc r3, r3, r3`**, linha 60, desloca **`remainder`** para a esquerda ao mesmo tempo que insere o valor do registo Carry no *bit* menos significativo. As duas instruções das linhas 59 e 60 em conjunto, transferem o *bit* da posição mais significativa de **`dividend`** para a posição menos significativa de **`remainder`** enquanto deslocam ambas variáveis para a esquerda.

O quociente é sempre deslocado para a esquerda – linha 61.

O valor corrente do resto – variável **`remainder`** – é comparado com o divisor – linha 62. Se o resto for maior ou igual ao divisor – não se executa o salto condicional da linha 63: ao valor corrente do resto é subtraído o valor do divisor – linha 64 – e, coerentemente, o quociente é incrementado de uma unidade – linha 65.

Este algoritmo realiza dezasseis iterações baseadas na variável **`i`** – registo **`r2`**. Na linha 67 **`i`** é decrementada e na linha 68 é verificado, através da *flag* Z, se já chegou a zero.

A instrução **`mov r0, r4`**, na linha 69, copia o resultado final para o registo **`r0`** que, por convenção, é onde as funções retornam o valor obtido.

Na linha 71, a instrução **`mov pc, lr`** transfere o endereço de retorno, guardado no *link register*, para o *program counter* o que provoca o retorno do programa à instrução da linha 24 na função **`main`**.

16.6.3 Pesquisa de um valor num *array*

O programa seguinte percorre um *array*, com o intuito de descobrir a primeira posição eventualmente ocupada por um dado valor. Na ausência do valor pesquisado não há índice que sirva, pelo que a função devolve o valor -1.

```
 1 /*-------------------------------------------------------------------
 2       #define ARRAY_SIZE(a)   (sizeof(a) / sizeof(a[0]))
 3       uint16 table1[] = {10, 20, 5, 6, 34, 9};
 4       uint16 table2[] = {11, 22, 33};
 5       int16 p, q;
 6
 7       int main() {
 8             p = search(table1, ARRAY_SIZE(table1), 20);
 9             q = search(table2, ARRAY_SIZE(table2), 31);
10       }
11 */
12       .data
13       .equ  TABLE1_SIZE, (table1_end - table1) / 2
14 table1:
15       .word 10, 20, 5, 6, 34, 9
```

```
16 table1_end:
17 table2:
18       .word 11, 22, 33
19 table2_end:
20
21 p:
22       .word 0
23 q:
24       .word 0
25
26       .text
27 main:
28       push  lr
29       ldr   r0, addr_table1
30       mov   r1, TABLE1_SIZE
31       mov   r2, 20
32       bl    search
33       ldr   r1, addr_p
34       str   r0, [r1]
35
36       ldr   r0, addr_table2
37       mov   r1, (table2_end - table2) / 2
38       mov   r2, 44
39       bl    search
40       ldr   r1, addr_q
41       str   r0, [r1]
42       mov   r0, 0
43       pop   pc
44
45 addr_table1:
46       .word table1
47 addr_table2:
48       .word table2
49 addr_p:
50       .word p
51 addr_q:
 5       .word q
 2
53 /*-------------------------------------------------------------
54 <r0> int16 search(<r0> uint16 array[], <r1> uint8 array_size,
55              <r2> uint16 value) {
56       for (<r3> uint8 i = 0; i < array_size && array[i] != value; ++i)
57             ;
58       if( i < array_size)
59             return i;
60       return -1;
61 }
62 */
63       .text
64 search:
65       push  r4
67       mov   r3, 0             /* r3 - i */
68 search_for:
69       cmp   r3, r1            /* i - array_size */
70       bhs   search_for_end
71       ldr   r4, [r0, r3]      /* array[i] != value */
72       cmp   r4, r2
```

```
 73       beq   search_for_end
 74       add   r3, r3, 2          /* ++i */
 75       b     search_for
 76 search_for_end:
 77       cmp   r3, r1             /* if (i < array_size) */
 78       bhs   search_if_end
 79       lsr   r0, r3, 1          /* return i */
 80       b     search_end
 81 search_if_end:
 82       mov   r0, 0              /* return -1 */
 83       sub   r0, r0, 1
 84 search_end:
 85       pop   r4
 86       mov   pc, lr
```

16.6.4 Ordenação de dados

O programa seguinte realiza a ordenação de um *array* de números naturais por ordem crescente, do início ao fim, utilizando uma variante do algoritmo de ordenação *bubble sort*.

```
1    /*-------------------------------------------------------------------------
2    uint16_t array[] = { 20, 3, 45, 7, 5, 9, 15, 2};
3
4    int main() {
5          sort(array, sizeof(array) / sizeof(array[0]));
6    }
7    */
8          .data
9    array:
10         .word 20, 3, 45, 7, 5, 9, 15, 2
11   array_end:
12
13         .text
14   main:
15         push  lr
16         ldr   r0, addr_array
17         mov   r1, (array_end - array) / 2
18         bl    sort
19         pop   pc
20
21   addr_array:
22         .word array
23
24   /*-------------------------------------------------------------------------
25   typedef     enum boolean {false = 0, true = !false} Boolean;
26
27   void sort(<r0> uint16_t a[], <r1> int dim)  {
28       <r2> Boolean swapped;
29       do {
30           swapped = false;
31           for (<r3> int i = 0; i < dim - 1; i++)
32               if ( a[i] > a[i + 1]) {
33                   int aux = a[i];
34                   a[i] = a[i + 1];
35                   a[i + 1] = aux;
36                   swapped = true;
```

```
37              };
38            dim--;
39        } while (swapped);
40    }
41    */
42          .equ  false, 0
43          .equ  true, !false
44    sort:
45          push  r4
46          push  r5
47          push  r6
48          sub   r1, r1, 1         ; dim - 1
49    sort_do:
50          mov   r2, false         ; do {
51          mov   r3, 0             ; i = 0
52          mov   r4, r0            ; r4 = address of a[0]
53    sort_for:
54          cmp   r3, r1            ; i - (dim - 1)
55          bhs   sort_for_end      ; if (i < dim-1)
56          ldr   r5, [r4]          ; r0 = a[i]
57          ldr   r6, [r4, 2]       ; r4 = a[i + 1]
58          cmp   r6, r5            ; a[i + 1] - a[i]
59          bge   sort_if_end       ; if (a[i] < a[i + 1])
60          str   r6, [r4]          ; swaps a[i] by a[i + 1]
61          str   r5, [r4, 2]
62          mov   r2, true          ; swap = true
63    sort_if_end:
64          add   r3, r3, 1         ; i++
65          add   r4, r4, 2         ; r6 = address of a[i]
67          b     sort_for
68    sort_for_end:
69          sub   r1, r1, 1         ; dim--
70          mov   r4, true
71          cmp   r2, r4
72          beq   sort_do           ; } while (swapped)
73          pop   r6
74          pop   r5
75          pop   r4
            mov   pc, lr            ; return
```

16.6.5 Chamadas encadeadas a funções diversas

```
 1 /*---------------------------------------------------------------------
 2 int8_t is_leap(<r0> uint16_t year) {
 3       return year % 4 == 0;
 4 }
 5 */
 6       .text
 7 is_leap:
 8       mov   r1, 3
 9       and   r0, r0, r1
10       mrs   r0, cpsr
11       lsr   r0, r0, 1
12       mov   r0, 0
13       adc   r0, r0, r0
14       mov   pc, lr
15
```

```
16 /*-------------------------------------------------------------------
17 int16_t month_days[] =
18     {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
19
20 int16_t year_days(<r0> uint16_t year,
21                <r1> uint8_t month, <r2> uint8_t day) {
22       return month_days[month - 1]
23             + (month > 2 && is_leap(year) ? 1 : 0) + day - 1;
24 }
25 */
26       .text
27 month_days:
28       .word 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365
29 year_days:
30       push  lr
31       push  r4
32       ldr   r3, addr_month_days
33       sub   r4, r1, 1   ; month - 1
34       add   r4, r4, r4  ; scale index of array by two
35       ldr   r3, [r3, r4]      ; month_days[month - 1]
36       sub   r1, r1, 3   ; month - 3
37       blo   year_days_if_end
38       bl    is_leap
39       add   r3, r3, r0  ; + is_leap(year)
40 year_days_if_end:
41       add   r3, r3, r2  ; + day
42       sub   r0, r3, 1   ; - 1;
43       pop   r4
44       pop   pc
45
46 addr_month_days:
47       .word month_days
48 /*-------------------------------------------------------------------
49 uint16_t days_since(<r5> <r0> uint16_t year_base, <r6> <r1> uint16_t year,
50                     <r7> <r2> uint8_t month, <r8> <r3> uint8_t day {
51       <r4> uint16_t days  = 0;
52       for ( <r5> uint16_t y = year_base; y < year; ++y)
53             days += 365 + is_leap(y);
54       return days + year_days(year, month, day);
55 }
56 */
57       .text
58 days_since:
59       push  lr
60       push  r4
61       push  r5
62       push  r6
63       push  r7
64       push  r8
65       push  r9
66       mov   r5, r0
67       mov   r6, r1
68       mov   r7, r2
69       mov   r8, r3
70       mov   r4, 0
71       mov   r9, 365 & 0xf
72       movt  r9, 365 >> 8
```

```
 73       b     days_since_for_cond
 74 days_since_for:
 75       add   r4, r4, r9
 76       mov   r0, r5
 77       bl    is_leap
 78       add   r4, r4, r0
 79       add   r5, r5, 1
 80 days_since_for_cond:
 81       cmp   r5, r6
 82       blo   days_since_for
 83       mov   r0, r6
 84       mov   r1, r7
 85       mov   r2, r8
 86       bl    year_days
 87       add   r0, r4, r0
 88       pop   r9
 89       pop   r8
 90       pop   r7
 91       pop   r6
 92       pop   r5
 93       pop   r3
 94       pop   pc
 95 /*-------------------------------------------------------------------
 96 uint16_t days;
 97 int main() {
 98       days = days_since(1900, 2017, 3, 30)
 99            - days_since(1900, 1995, 3, 8);
100 }
101 */
102       .data
103 days:
104       .word 0
105
106       .text
107 main:
108       push  lr
109       mov   r0, 1900 & 0xff
110       movt  r0, 1900 >> 8
111       mov   r1, 2017 & 0xff
112       movt  r1, 2017 >> 8
113       mov   r2, 3
114       mov   r3, 30
115       bl    days_since
116       mov   r4, r0
117       mov   r0, 1900 & 0xff
118       movt  r0, 1900 >> 8
119       mov   r0, 1995 & 0xff
120       movt  r0, 1995 >> 8
121       mov   r2, 3
122       mov   r3, 8
123       bl    days_since
124       sub   r0, r4, r0
125       ldr   r1, addr_days
126       str   r0, [r1]
127       pop   pc
128
129 addr_days:
```

```
130       .word       days
```

16.6.6 Chamada recursiva de funções

```
 1 /*-------------------------------------------------------------
 2 uint16_t a = 8, fa;
 3 uint16_t fb;
 4
 5 int main() {
 6     fa = factorial(a);
 7     fb = factorial(8);
 8 }
 9 */
10       .data
11 a:
12       .word 8
13 fa:
14       .word 0
15 fb:
16       .word 0
17
18       .text
19 main:
20       push  lr
21       ldr   r0, addr_a
22       ldr   r0, [r0]
23       bl    factorial
24       ldr   r1, addr_fa
25       str   r0, [r1]
26       mov   r0, 8
27       bl    factorial
28       ldr   r1, addr_fb
29       str   r0, [r1]
30       pop   pc
31 addr_a:
32       .word a
33 addr_fa:
34       .word fa
35 addr_fb:
36       .word fa
37 /*-------------------------------------------------------------
38 uint16_t factorial(uint16_t n) {
39       if (n == 0)
40             return 1;
41       else
42             return(n * factorial(n - 1));
43 }
44 */
45 factorial:
46       add   r0, r0, 0   ; if (n == 0)
47       beq   factorial_1
48       push  lr          ; save return address
49       push  r0          ; save n in stack
50       sub   r0, r0, 1   ; factorial(n - 1)
51       bl    factorial
52       pop   r1          ; recover n
53       bl    mul16       ; n * factorial(n - 1)
```

```
54       add   r1, r1, 0   ; multiplication overflow ?
55       beq   factorial_2
56       b     .
57 factorial_2:
58       pop   pc          ; restore return address and return
59 factorial_1:
60       mov   r0, 1
61       mov   pc, lr
62
63 /*-------------------------------------------------------------
64 uint32_t mul16(<r0> uint16_t multiplicand, <r1> uint16_t multiplier)) {
65       <r2:r0> uint32_t multiplicandi = (uint32_t) multiplicand;
66       <r4:r3> uint32_t product = 0;
67       while ( multiplier > 0 ) {
68             if ( (multiplier & 1) != 0 )
69                   product += multiplicandi;
70             multiplier >>= 1;
71             multiplicandi <<= 1;
72       }
73       <r1:r0> return product;
74 }
75 */
76 mul16:
77       push  r4
78       mov   r2, 0             ; <r2:r0> uint32_t multiplicandi
79                               ;     = (uint32_t) multiplicand;
80       mov   r3, 0             ; <r4:r3> uint32_t product = 0;
81       mov   r4, 0
82 mul16_while:
83       add   r1, r1, 0         ; while ( multiplier > 0 )
84       beq   mul16_return
85       lsr   r1, r1, 1         ; if ( (multiplier & 1) != 0 )
86       bcc   mul16_if_end
87       add   r3, r3, r0        ; product += multiplicandi;
88       adc   r4, r4, r2
89 mul16_if_end:
90       lsl   r0, r0, 1         ; multiplicandi <<= 1;
91       adc   r2, r2, r2
92       b     mul16_while
93 mul16_return:
94       mov   r0, r3            ; <r1:r0> return product;
95       mov   r1, r4
96       pop   r4
97       mov   pc, lr
```

16.7 Sintaxe da linguagem *assembly* do P16

A linguagem *assembly* do P16 é semelhante à usada pelo *assembler* AS da GNU quando usado no desenvolvimento de programas para a arquitectura ARM. O objectivo é facilitar ao estudante a transição para essa arquitectura.

A sintaxe das instruções apesar de definida no âmbito da arquitectura P16 é semelhante à sintaxe unificada da arquitectura ARM.

Em seguida descrevem-se, na notação *Wirth syntax notation* (WSN), as regras sintácticas a aplicar na escrita de programas em linguagem *assembly* do P16.

[a]	o elemento **a** é opcional
a | b	**a** ou **b** são elementos alternativos
{a}	o elemento **a** pode não existir ou repetir-se indefinidamente
"a"	elemento terminal

```
program = statement { statement }.

statement =
      [label] [instruction | direcive] "EOL" .

directive =
      ( ".section" symbol )
      | ".text"
      | ".data"
      | ".align" [ expression ]
      | ".equ" symbol "," expression )
      | ( ".byte" | ".word" ) expression { "," expression }
      | ".space" expression [ "," expression ] )
      | ( ".ascii" | ".asciz" ) string { "," string } .

instruction =
      "ldr" reg0-15 ","
                  ( "[" ("pc" | "r15") "," expression "]" ) | identifier

      ( ( "ldr" | "str" ) ["b"]
            reg0-15 "," ( "[" reg0-7 ["," (reg0-15 | expression)] "]" )
      | ( "mov" | "movt" ) reg0-15, (reg0-15 | expression)
      | ( "push" | "pop" ) ["{"] reg0-15
      | ( "add" | "sub" ) reg0-15, reg0-7, (reg0-15 | expression)
      | ( "adc" | "sbc" ) reg0-15, reg0-7,  reg0-15
      | "cmp" reg0-7, reg0-15
      | ( "and" | "orr" | "eor" ) reg0-15, reg0-7, reg0-15
      | ( "mvn" | "not") reg0-15, reg0-15
      | ( "lsl" | "lsr" | "asr" | "ror" ) reg0-15, reg0-7, expression
      | "rrx" reg0-15, reg0-7
      | "msr" psw "," reg0-15
      | "mrs" reg0-15 "," psw
      | ( "bzs" | "beq" | "bzc" | "bne" | "bcs" | "blo" | "bcc" | "bhs"
            | "blt" | "bge" | "bl" | "b" ) symbol
      | "movs pc, lr" .
```

```
reg0-7 = "r0" | "r1" | "r2" | "r3" | "r4" | "r5" | "r6" | "r7" .

reg0-15 = reg0-7
            | "r8" | "r9" | "r10" | "r11" | "r12" | "r13" | "r14" | "r15"
            | "sp" | "lr" | "pc" .

psw = "cpsw" | "spsw" .

expression = logical_or_expression
      | logical_or_expression "?" expression ":" expression .

logical_or_expression = logical_and_expression
      | logical_or_expression "||" logical_and_expression .

logical_and_expression = inclusive_or_expression
      | logical_and_expression "&&" inclusive_or_expression .

inclusive_or_expression = exclusive_or_expression
      | inclusive_or_expression "|" exclusive_or_expression .

exclusive_or_expression = and_expression
      | exclusive_or_expression "^" and_expression .

and_expression = equality_expression
      | and_expression "&" equality_expression .

equality_expression = relational_expression
      | equality_expression "==" relational_expression
      | equality_expression "!=" relational_expression .

relational_expression = shift_expression
      | relational_expression "<" shift_expression
      | relational_expression ">" shift_expression
      | relational_expression "<=" shift_expression
      | relational_expression ">=" shift_expression .

shift_expression = additive_expression
      | shift_expression "<<" additive_expression
      | shift_expression ">>" additive_expression .

additive_expression = multiplicative_expression
      | additive_expression "+" multiplicative_expression
      | additive_expression "-" multiplicative_expression .

multiplicative_expression = unary_expression
      | multiplicative_expression "*" unary_expression
      | multiplicative_expression "/" unary_expression
      | multiplicative_expression "%" unary_expression .

unary_expression = primary_expression
      | "+" primary_expression
      | "-" primary_expression
      | "!" primary_expression
      | "~" primary_expression .

primary_expression = literal | identifier | "(" expression ")" .

identifier = (alphabet | "_") { alphabet | number | "_" }.
```

```
label =  identifier ":" .

literal = decimal | hexadecimal | octal | binary | "'" character "'" .

decimal = "0" | (("1" | ... | "9") { decimal_digit } ) .

hexadecimal = "0" ("x" | "X") hexadecimal_digit { hexadecimal_digit } .

octal = "0" ("1" | ... | "7") { octal_digit } .

binary = "0" ("b" | "B") ("0" | "1") { "0" | "1" } .

octal_digit = "0" | "1" | ... | "6" | "7" .

decimal_digit = "0" | "1" | ... | "8" | "9" .

hexadecimal_digit = decimal_digit | "a" | ... | "f" | "A" | ... | "F" .

alphabet = "a" | ... | "z" | "A" | ... | "Z" .

symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
      | "=" | "|" | "&" | "%" | "$" | "#" | "/" | "?" | "!" | "_" | "*"
      | "\b" | "\t" | "\n" | "\f" | "\r" | "\\" | "\"" | "\'"
      | ( "\" ( decimal | hexadecimal | octal | binary ) ) .

character = alphabet | decimal_digit | symbol .

string = "\"" character { character } "\"" .

"EOL" = control character for end of line
```