

Tipos

A norma ANSI C define os seguintes tipos básicos:

`char` `short` `int` `long` `float` `double` `long double`

As dimensões impostas para cada tipo são as seguintes:

`sizeof(char)` ≥ 8 bit `sizeof(short)` ≥ 16 bit
`sizeof(int)` ≥ 16 bit `sizeof(long)` ≤ 32 bit

`sizeof(char)` \leq `sizeof(short)` \leq `sizeof(int)` \leq `sizeof(long)`
`sizeof(float)` \leq `sizeof(double)` \leq `sizeof(long double)`

`sizeof(int)` é igual à dimensão da palavra natural do processador, medida em bytes.

Os tipos derivados (**struct**, **union**, **array**, **class**) são agregados de tipos básicos.

Os ponteiros são endereços da memória.

O compilador GNU suporta o tipo `long long` que tem uma capacidade de codificação que é o dobro de `long`. Para a arquitectura ARM o `long long` é codificado a 64 bits.

Os qualificadores **signed** e **unsigned** são suportados por instruções adequadas.

Dimensões e alinhamento

Tipo	Dimensão	Alinhamento
char	1	1
short	2	2
int	4	4
long	4	4
long long	8	4
float	4	4
double	8	4
pointer	4	4
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

<pre>struct A { char a; int b; char c; };</pre>	<code>sizeof (struct A) == 12</code>	4
<pre>struct A { char a, b, c, d; };</pre>	<code>sizeof (struct A) == 4</code>	4

Portabilidade

Se se pretender portabilidade nos programas em C terá que se prestar atenção ao domínio de valores dos tipos utilizados. Na linguagem C, esses domínios podem diferir entre arquitecturas de processador diferentes. Num processador de 16 bits o tipo int suporta valores de -32768 a +32767 enquanto num processador a 32 bits o tipo int suporta valores de -2147483648 a +2147483647.

Uma forma de manter o domínio de valores, associado a um tipo, em diversas arquitecturas, é criar novos tipos (U8, U16, ...) e, por redefinição desses tipos para cada arquitectura, assegurar a dimensão adequada.

Para um processador de 32 bits poderemos ter:

```
typedef unsigned char    U8;
typedef unsigned short   U16;
typedef unsigned long    U32;
typedef unsigned long long U64;

typedef char             I8;
typedef short            I16;
typedef long             I32;
typedef long long        I64;

#define U32_MAX 4294967295UL
#define U16_MAX 65535U
#define U8_MAX 255

#define I32_MAX 2147483647L
#define I16_MAX 32767
#define I8_MAX 127

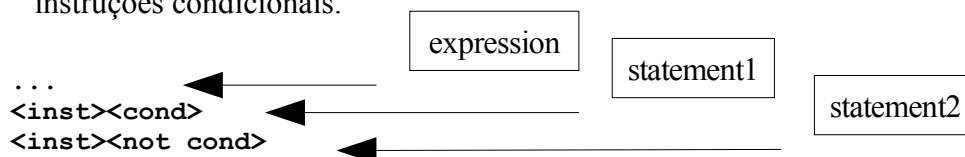
#define I32_MIN (-2147483648L)
#define I16_MIN (-32768)
#define I8_MIN  (-128)
```

Estruturas de controlo de fluxo

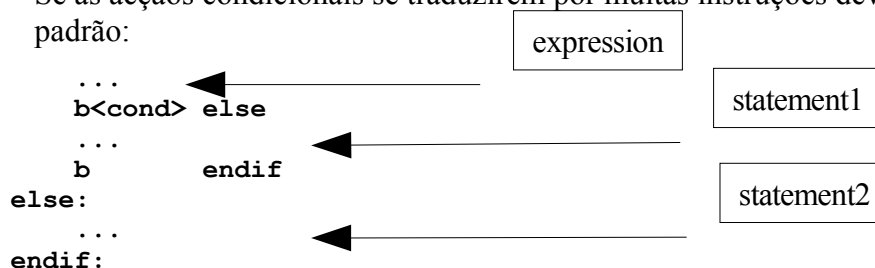
if ... else

if (expression) statement1 else statement2

- Se as acções condicionais se traduzirem por poucas instruções (até 3) devem usar-se instruções condicionais.



- Se as acções condicionais se traduzirem por muitas instruções deve usar-se o seguinte padrão:



switch

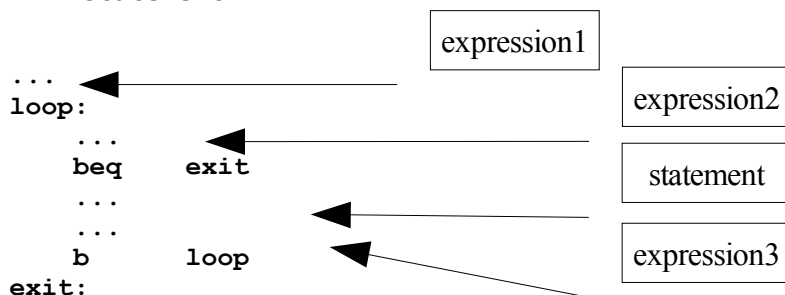
<pre> switch (expression) { case constant1: statements1 case constant2: statements2 case constant3: statements3 default: statementsD } </pre>	<pre> ldr r3, [fp, #-32] ldr r3, [r3] cmp r3, #6 ldrls pc, [pc, r3, asl #2] b .L2 .L10: .word .L3 .word .L4 .word .L5 .word .L6 .L3: ... b .L2 .L4: ... b .L2 .L5: ... b .L2 .L6: ... b .L2 .L2: ... </pre>
---	--

for

```

for (expression1; expression2; expression3)
    statement

```

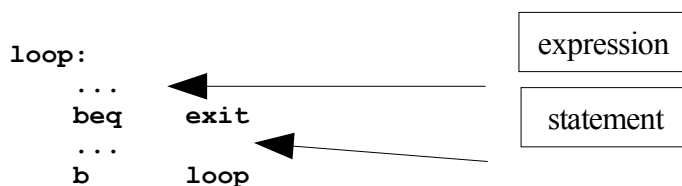


while

```

while (expression)
    statement

```



do while

```

do {
    statement
} while (expression);

```



```

...
...
bne    loop

```

break

Saltar para a instrução a seguir ao ciclo.

continue

Saltar para a avaliação da condição de permanência no ciclo.

goto

Saltar para a instrução da *label* indicada.

return

Saltar para o epílogo da função ou executar logo a instrução de retorno.

Operações com bits

Deslocar um valor para a esquerda n posições	b = a << n;	mov r1, r0, lsl r0
Colocar o bit da posição n a zero	b = a & ~(1 << n);	mov r2, #1 mvn r2, r2, lsl r0 and r1, r1, r2
Colocar o bit da posição n a um	b = a 1 << n;	mov r2, #1 orr r1, r2, lsl r0
Testar o valor do bit na posição n	if (a & (1 << n))	mov r2, #1 tst r1, r2, lsl r0
Obter o campo de n bits a começar na posição p . A posição 0 é a mais à direita.	unsigned getbits(unsigned x, int p, int n){ return (x >> p) & ~(~0 << n); }	getbits: mvn r3, #0 mvn r3, r3, lsl r2 and r0, r0, r3, lsr r1 mov pc, lr

Ponteiros

Aritmética de ponteiros.

Considerando:

```

int * p, *q;
int n;

```

p + n	aponta para o elemento n posições à frente do apontado por p	a = *(p + n);	ldr r0, [r1, r2, lsl #2]	a - r0 p - r1 n - r2
p - q	representa o número de elementos entre o	n = p - q;	sub r0, r0, r1 mov r2, r0 asr #2	p - r0 q - r1

	apontado por p e o apontado por q , mais um			n - r2
--	---	--	--	---------------

Ponteiros e arrays

Os ponteiros são endereços de memória. Na arquitectura ARM são representados a 32 bits.

Quando se declara um array (`int array[10];`) estabelece-se um símbolo que representa um ponteiro para o primeiro elemento do array.

<code>char a, b;</code> <code>int i, j;</code> <code>char * cp;</code> <code>int * ip;</code>	.data <code>a: .byte 0</code> <code>b: .byte 0</code> <code>i: .int 0</code> <code>j: .int 0</code> <code>cp: .word 0</code> <code>ip: .word 0</code>	<code>b = *cp;</code>	<code>ldr r0, =cp</code> <code>ldr r0, [r0]</code> <code>ldrb r0, [r0]</code> <code>ldr r1, =b</code> <code>strb r0, [r1]</code>
<code>cp = &a;</code>	.text <code>ldr r0, =a</code> <code>ldr r1, =cp</code> <code>str r0, [r1]</code>	<code>i = *ip;</code>	<code>ldr r0, =ip</code> <code>ldr r0, [r0]</code> <code>ldr r0, [r0]</code> <code>ldr r1, =i</code> <code>str r0, [r1]</code>
<code>cp++;</code>	<code>ldr r0, =cp</code> <code>ldr r1, [r0]</code> <code>add r1, r1, #1</code> <code>str r1, [r0]</code>	<code>ip++;</code>	<code>ldr r0, =ip</code> <code>ldr r1, [r0]</code> <code>add r1, r1, #4</code> <code>str r1, [r0]</code>
<code>ip = ip + i</code>	<code>ldr r0, =ip</code> <code>ldr r1, [r0]</code> <code>ldr r3, =i</code> <code>ldr r2, [r3]</code> <code>add r1, r1, r2, lsl #2</code> <code>str r1, [r0]</code>	<code>j = *(ip + i);</code> <code>j = ip[i];</code>	<code>ldr r0, =ip</code> <code>ldr r1, [r0]</code> <code>ldr r3, =i</code> <code>ldr r2, [r3]</code> <code>ldr r2, [r1, r2, lsl #2]</code> <code>ldr r1, =j</code> <code>str r2, [r1]</code>

Memória de um programa

As componentes de um programa (funções, variáveis, ...) em C são alojadas na memória segundo critérios que permitem:

- agrupar componentes com as mesmas características (“código com código”, “variáveis com variáveis”, ...).
- manipular separadamente zonas de memória (alojar em RAM ou em ROM, comprimir, iniciar).
- otimizar a dimensão da memória ocupada e a eficiência dos acessos.

A especificação **elf** define a seguinte composição básica da memória de um programa:

.stack zona de stack do programa.

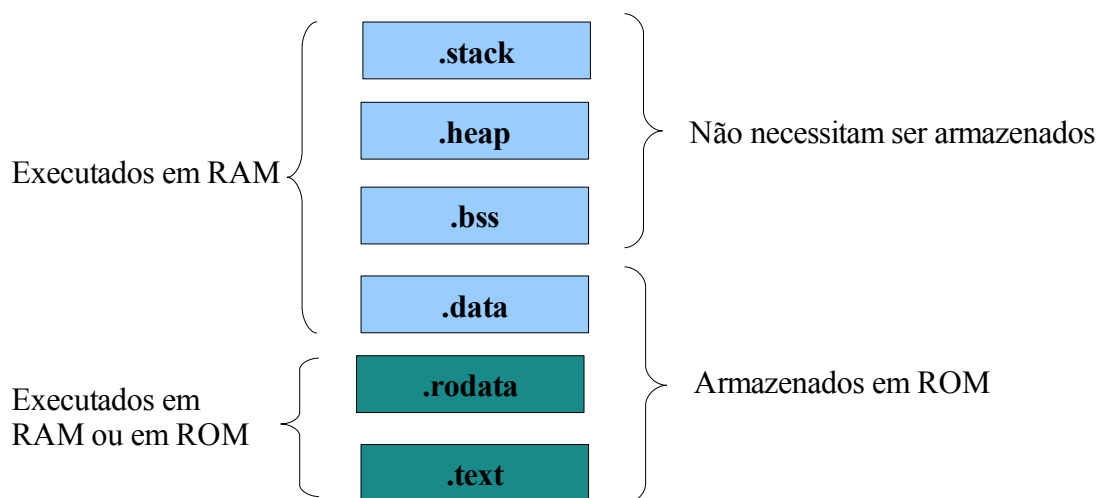
.heap zona de heap do programa.

.bss variáveis iniciadas a zero.

.data variáveis com valor inicial definido.

.rodata constantes.

.text código das instruções.



Alojamento

```
int i;                                .bss      4
static int j;                        .bss      4
char nomes[100][20];                 .bss    200
int x = 20;                           .data     4
static int y = 24;                    .data     4
char n[] = "Joaquim";                 .data     8
const char n[] = "Joaquim";           .rodata    8
char * c = "Francisco";               .data     4   .rodata  10
char * const c = "Francisco";         .rodata   16
const char * c = "all";               .data     4   .rodata   4

int main() {
    static int i = 55;                .data          4
    int j;                            .stack     4
    int k = 34;                       .stack     4
    const int = 20;                   .stack     4
    static const int = 20;            .rodata     4
}
```

APCS (ARM Procedure Call Standard)

Utilização dos registos

Nome APCS	Registo	Utilização
a1-a4	r0-r3	Argumentos
a1-a4, ip	r0-r3, r12	Não é necessário preservar
v1-v7	r4-r10	É necessário preservar
fp	r11	Frame pointer
sp	r13	Stack pointer

lr	r14	Endereço de retorno
pc	r15	Program counter
	r0 e r1	Retorno de valores

Stack

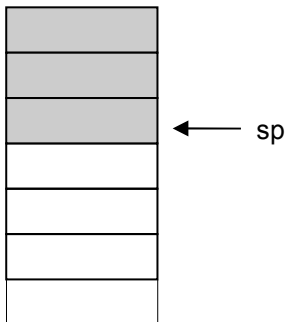
O stack cresce de endereços altos para endereços baixos.

Uma função folha pode não precisar de stack frame.

O stack pointer está sempre alinhado num endereço múltiplo de 4.

O stack pointer contém o endereço da palavra mais recente armazenada no stack.

```
push - stmdb sp!, { . . . } = stmfd sp!, { . . . }
pop  - ldmia sp!, { . . . } = ldmfd sp!, { . . . }
top  - ldmia sp, { . . . } = ldmfd sp, { . . . }
```



Passagem de parâmetros

Os tipos char, short, int, long e pointer ocupam uma word.

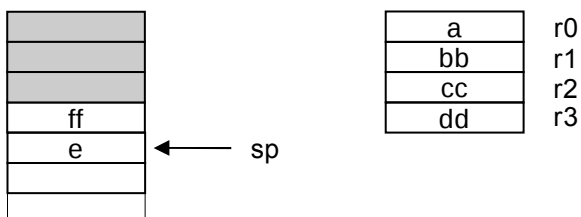
Agregados ocupam sempre um número múltiplo de words.

Valores representados em virgula flutuante ocupam um número múltiplo de words.

Os argumentos forma uma sequência de words com a ordem com que estão escritos.

Até às primeiras quatro words são passadas nos registros r0 a r3. As restantes são passadas pelos stack, empilhadas por ordem inversa.

```
void f(int a, int b, int c, int d, int e, int f);
```



Retorno de valores

Os tipos char, short, int, long, float e pointer são retornados em r0.

Os agregados (struct ou union) de dimensão igual ou inferior a uma word são retornados em r0.

O tipos double e long long são retornados em r1:r0.

Os outros valores são depositados numa zona de memória fornecida para o efeito, pelo chamador. O endereço desta zona de memória é passado como mais um parâmetro inserido automaticamente pelo compilador na primeira posição.

```

struct {
    int a;
    int b;
    char c;
} t;

int arg;

t = function(arg);

```

A chamada a function é processada como se estivesse escrita da seguinte forma:

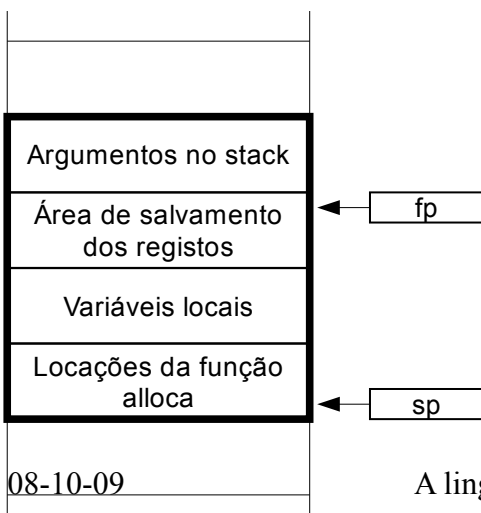
```
void function(&t, arg);
```

Exemplos de codificação de funções

<pre>void f0() { }</pre>	<pre>f0: mov pc, lr</pre>
<pre>int f1(int x) { return x; }</pre>	<pre>f1: mov pc, lr; r0 mantém-se inalterado</pre>
<pre>int sum(int x, int j) { return x + j; }</pre>	<pre>sum: add r0, r0, r1 mov pc, lr</pre>
<pre>size_t strlen(char * str) { size_t count = 0; while (*str) ++count; return count; }</pre>	<pre>strlen: mov r1, r0 mov r0, #0 1: ldrb r2, [r1], #1 cmp r2, #0 moveq pc, lr add r0, r0, #1 b 1b</pre>

Estas funções são funções folha, isto é, não chamam outras funções. Em chamadas sucessivas é necessário preservar o valor de lr antes de iniciar uma nova chamada. No caso de ser necessário usar os registos r4 a r10 estes devem ser também preservados. No caso de os registos não serem suficientes para alojar as variáveis locais é necessário alojá-las no stack. A APCS define o formato do stack em cada activação de uma função de modo a dar resposta e estas necessidades.

Stack frame



Nas funções com um número de parâmetros variável, depois de eventuais argumentos no stack, é reservado um espaço para colocação de argumentos anónimos passados em registos.

Padrão de código para entrada numa função.

```
mov     r12, sp
stmfd   sp!, {..., fp, r12, lr, pc}
sub     fp, r12, #4
sub     sp, sp, #...
```

Padrão de código para saída de uma função.

```
ldmdb   fp, {..., fp, sp, pc}
```

Exemplos de codificação de funções

<pre>int sum(int i, int j) { return i + j; } int triple(int x) { return sum(x, sum(x, x)); } int main() { return triple(2); }</pre>	<pre>sum: add r0, r0, r1 mov pc, lr triple: mov r12, sp stmdb sp!, {r4, fp, r12, lr, pc} sub fp, r12, #4 mov r4, r0 mov r1, r0 bl sum mov r1, r4 bl sum ldmbd fp, {r4, fp, sp, pc} main: mov r12, sp stmdb sp!, {fp, r12, lr, pc} sub fp, r12, #4 mov r0, #2 bl triple ldmbd fp, {fp, sp, pc}</pre>
---	--

Função com número de parâmetros variável

```
void mysprintf(char * str, char * fmt, ...) {
    va_list ap;
    char * p;
    va_start(ap, fmt);
    for (p = fmt; *p; ++p) {
        if (*p != '%')
            *str++ = *p;
        else {
            int base = 0;
            switch (*++p) {
                case 'x':
                    base += 6;
                case 'd':
                    base += 2;
                case 'o':
```

```

        base += 6;
    case 'b': {
        int ival = va_arg(ap, int);
        base += 2;
        str = int_to_str(ival, str, base);
        break; }
    case 's': {
        char * s;
        for (s = va_arg(ap, char *); *s ; ++s)
            *str++ = *s;
        break; }
    default:
        *str++ = *p;
    }
}
}
va_end(ap);
*str = 0;
}

```

Acesso aos parâmetros

1. Deve ser definido um apontador para a lista de parâmetros.

```
va_list ap;
```

2. O apontador para a lista de parâmetros deve ser iniciado.

```
va_start(va_list ap, lastarg);
```

3. A macro `va_arg` produz um valor do tipo especificado e ajusta o apontador para o próximo argumento da lista.

```
va_arg(va_list ap, type);
```

4. Para executar no fim do processamento e antes de sair da função.

```
va_end(va_list ap);
```

stdarg.h

```

#ifndef _STDARG_H
#define _STDARG_H

typedef char * va_list;

/*
    Dimensao do tipo x arredondado para multiplo de sizeof(int)
*/
#define _size(x) ((sizeof(x) + sizeof(int)-1) & ~(sizeof(int)-1))

/*
    Assume-se que o buraco e' nos enderecos superiores

    va_arg(ap, char) num processador de 32 bits

    -----

```

```

|   |car| * | * | * |   |   |   |   |
-----
      ^ap                ^ap + _size(char)
*/

#define va_start(p, arg)    (void) ((p) = (char *) & (arg) + _size(arg))
#define va_arg(p, T)       ( * (T *) ( ((p) += _size(T)) - _size(T) ) )
#define va_end(p)          (void)0

#endif

```

Exercício de codificação em assembly ARM

main.c

```

#include <stdio.h>
#include <stdarg.h>

void my_sprintf(char * str, char * fmt, ...);

char buffer[200], * p = buffer;
int i = 20;
char * str = "ola";

int main() {
    my_sprintf(p, "decimal = %d ; hexadecimal = %x ; octal = %o ; "
        "binario = %b ; &i = %x ; str = %s", i, i, i, i, (int)&i, str);
    return 0;
}

```

int_to_str.c

```

char * int_to_str(int i, char * str, int base) {
    char * begin = str, * end;
    if (i == 0) {
        *str++ = '0';
        *str = 0;
        return str;
    }
    do {
        *str++ = i % base < 10 ? i % base + '0' : i % base - 10 + 'a';
        i /= base;
    } while (i > 0) {
        *str = 0;
        end = str - 1;
        while (begin < end) {
            char aux = *begin;
            *begin++ = *end;
            *end-- = aux;
        }
        return str;
    }
}

```

my_printf.s

```

void my_sprintf(char * str, char * fmt, ...) {
                                r0          r1      r2   r3   stack
|           |
+-----+
|   p6   |
+-----+

```

```

|   p5   | <- sp
+-----+
|   r3   |
+-----+
|   r2   | <- ap
+-----+
|   r1   | <- fmt
+-----+
|   r0   | <- str
+-----+
|   pc   | <- fp
+-----+
|   lr   |
+-----+
|   sp   |
+-----+
|   fp   | <- sp'
+-----+
|       |

*/
.global my_printf
my_printf:
    mov     r12, sp
    stmfd   sp!, {r0 - r3}
    stmfd   sp!, {r4 - r6, fp, r12, lr, pc}
    sub     fp, r12, #20
    add     r4, fp, #12      /* r4 = ap = va_start(ap, fmt) */
    mov     r5, r1          /* r5 = p = fmt */
    mov     r6, r0          /* r6 = str */
1:
    ldrb    r0, [r5], #1     /* *p */
    cmp     r0, #0
    beq     endfor
    cmp     r0, #'%'         /* if (*p != '%') */
    beq     2f
    strb    r0, [r6], #1     /* *str++ = *p */
    b       1b
2:
                                /* else */
    mov     r2, #0           /* base = 0 */
    ldrb    r0, [r5], #1     /* switch (*++p) */
    cmp     r0, #'x'
    beq     x
    cmp     r0, #'d'
    beq     d
    cmp     r0, #'o'
    beq     o
    cmp     r0, #'b'
    beq     b
    cmp     r0, #'s'
    beq     s
    b       default
x:
    add     r2, r2, #6
d:
    add     r2, r2, #2
o:
    add     r2, r2, #6
b:
    add     r2, r2, #2        /* r2 = base */
    ldr     r0, [r4], #4     /* ival = va_arg(ap, int) */
    mov     r1, r6          /* r1 = str */

```

```

        bl      int_to_str
        mov     r6, r0          /* str = int_to_string( */
        b      1b
s:
        ldr     r1, [r4], #4     /* s = va_arg(ap, * char) */
2:
        ldrb    r0, [r1], #1
        cmp     r0, #0
        beq     1b
        strb    r0, [r6], #1
        b      2b
default:
        strb    r0, [r6], #1
        b      1b

endfor:
        mov     r0, #0
        strb    r0, [r6]
        ldmdb   fp, {r4 - r6, fp, sp, pc}

```

int_to_str.s

```

        .global int_to_str
int_to_str:
        mov     r12, sp
        stmfd   sp!, {r4 - r6, fp, r12, lr, pc}
        sub     fp, r12, #4
        mov     r4, r1          /* r4 = str */
        mov     r5, r2          /* r5 = base */
        mov     r6, r1          /* r6 = begin = str */

        cmp     r0, #0
        bne     1f              /* if (i == 0) */
        mov     r0, #'0'
        strb    r0, [r1], #1
        mov     r0, #0
        strb    r0, [r1]
        mov     r0, r1
        ldmdb   fp, {r4 - r10, fp, sp, pc}
1:
        mov     r1, r5
        bl      division        /* r0 = i, r1 = base, return r0 = i / base, r1 = i %
base */
        cmp     r1, #10
        addle   r1, r1, # '0'
        addge   r1, r1, # - 10 + 'a'
        strb    r1, [r4], #1
        cmp     r0, #0          /* while (i > 0) */
        bgt     1b
        mov     r0, #0
        strb    r0, [r4]        /* *str = 0 */
        sub     r3, r4, #1      /* r3 = end = str - 1, r6 = begin */
3:
        cmp     r6, r3          /* while (begin < end) */
        bge     4f
        ldrb    r0, [r3]
        ldrb    r1, [r6]
        strb    r1, [r3], #-1
        strb    r0, [r6], #1
        b      3b

```

```

4:
    mov     r0, r4
    ldmdb   fp, {r4 - r6, fp, sp, pc}

```

division.s

```

/*  int division ( int numerator, int denominator );
    r0              r0              r1
*/
    .text
    .global  division
division:
    cmp     r1, #0                /*  Evitar divisão por zero */
    beq     .

    mov     r2, #0                /*  r2 - resto          */
    mov     r3, #0                /*  r3 - quociente      */
    mov     r12, #32              /*  r12 - 32 iterações */

1:
    movs    r0, r0, lsl #1        /*  dividendo = dividendo << 1 */
    adc     r2, r2, r2            /*  resto = resto << 1 + bit de maior peso
                                @ do dividendo

    cmp     r2, r1                /*  carry = 1 se resto(r2) >= divisor(r1) */
    subcs   r2, r2, r1            /*  se resto >= divisor então
                                resto = resto - divisor */
    adc     r3, r3, r3            /*  quociente = quociente << 1 + carry */
    subs    r12, r12, #1
    bne     1b
    mov     r0, r3
    mov     r1, r2
    mov     pc, lr

```

Referências:

ARM Software Programação com módulos Development Toolkit, Chapter 9, ARM Procedure Call Standard

Programação com módulos

A divisão dos programas em módulos (ficheiros) é uma prática indispensável que facilita a produção, manutenção e a reutilização de *software*.

Uma parte de um programa num módulo interage com outra parte noutro módulo através do acesso a variáveis ou funções.

As referências para variáveis ou para funções, são feitas através de símbolos.

Um símbolo tem subjacente o endereço do elemento referenciado – endereço da variável ou endereço da função.

Para que um símbolo definido num módulo seja visível noutro módulo é necessário que este seja globalmente visível.

Na linguagem *assembly* GNU, por omissão, um símbolo é visível apenas no módulo onde é declarado. Para o tornar globalmente visível é necessário explicitar através da directiva `.global`.

Um símbolo globalmente visível precisa ser conhecido no módulo onde é referenciado.

O **as** assume que um símbolo referenciado e não declarado no módulo actual é global e é definido noutro módulo.

Na linguagem C, por omissão, um símbolo é globalmente visível. Para o tornar visível apenas no módulo onde é definido deve usar-se o atributo `static`.

int x; x é globalmente visível.

static int y; y é visível apenas no módulo onde é definido.

Para se poder usar um símbolo definido noutro módulo é necessário que este seja declarado no módulo onde é usado:

extern int x; declara que existe um símbolo **x**, do tipo **int**, definido noutro módulo.

int strlen(char *str); declara que existe um símbolo **strlen**, que representa uma função que recebe um ponteiro para char e devolve um int. A sua definição encontra-se algures noutro módulo.

Definições:

Declarar uma entidade consiste em anunciar a sua existência.

Definir uma entidade implica reservar espaço e explicitar as instruções que determinam o seu comportamento. (Programação em C++, pág. 36).