Conversão de programas em C para *assembly* do MIPS

Tomás Oliveira e Silva

***Abstract* – Neste documento apresenta-se uma maneira sistemática possível de gerar código *assembly* (para o MIPS), a partir de código em C. O objectivo principal do documento é o de mostrar como se pode decompor o código em C em pedaços elementares, que depois podem ser convertidos individualmente para assembly sem dificuldade. Em muitos casos é possível depois optimizar o código gerado desta maneira; isso não será descrito neste documento.**

Conteúdo

Conteúdo 1
I Introdução 1
II O *assembly* do MIPS 1
III A linguagem C 3
III-A Variáveis 3
III-B Acesso à memória usando ponteiros 5
III-C Funções 5
III-D Expressões aritméticas e lógicas 7
III-E Controlo do fluxo de execução 9
III-F Estruturas e uniões 11
III-G Miscelânea 13
III-H Exemplo final 13
Bibliografia 16

I. Introdução

Nas aulas práticas da disciplina de Arquitectura de Computadores I (AC I) é pedido aos alunos para converterem programas de uma linguagem de alto nível para uma linguagem de baixo nível. Devido às suas características relativamente simples, a linguagem de alto nível escolhida é o C e a de baixo nível é o *assembly* do MIPS (apenas versão inicial de 32 bits, **sem** *delay slot*). Não é objectivo deste documento ensinar nem C nem *assembly* do MIPS. Apenas se pretende mostrar, através de exemplos, que a tradução de C para *assembly* não é difícil; é, no entanto, necessário prestar muita atenção (para evitar enganos) ao sítio onde cada variável irá ser armazenada (registo ou memória).

Um programa em C é constituído por um ou mais ficheiros. Tal como para outras linguagens de alto nível, é utilizado um compilador para converter o código fonte (neste caso C) do programa para uma versão (linguagem máquina) que pode ser executada directamente no processador pretendido. Um compilador típico faz isso em várias etapas [1, secção 2.15, no CD]:

1. Em primeiro lugar é feito o chamado *parsing* do código fonte, sendo gerado código equivalente numa linguagem intermédia simples (veremos mais adiante exemplos disso). Isto é feito no chamado *front-end* do compilador; um compilador pode ter *front-ends* para várias linguagens de programação.
2. O código na linguagem intermédia pode depois ser optimizado. Por exemplo, pode-se eliminar código que de certeza nunca será executado, e pode-se reutilizar (partes de) expressões que foram calculadas previamente. Também se podem aplicar optimizações muito mais sofisticadas cuja descrição sai fora do âmbito deste documento.
3. O código na linguagem intermédia resultante da etapa anterior é depois convertido para *assembly* do processador pretendido. Isto é feito no chamado *back-end* do compilador; um compilador pode ter *back-ends* para vários processadores diferentes.
4. O *assembly* gerado na etapa anterior pode depois ser optimizado. Por exemplo, certas sequências de instruções elementares podem ser substituidas por uma ou mais instruções especializadas que são suportadas pelo processador pretendido.
5. Na penúltima etapa, o *assembly* é convertido em linguagem máquina do processador pretendido. Isto é feito por um *assembler*.
6. Na última etapa os vários pedaços do programa são juntos e é gerado um ficheiro executável. Isto é feito por um *linker*. É nesta fase que os símbolos que não estão definidos no *assembly* mas que são usados no programa são procurados em arquivos de ficheiros (as chamadas bibliotecas); um símbolo encontrado num dos ficheiros do arquivo faz com que o código (e dados) armazenados nesse ficheiro também passe a fazer parte do ficheiro executável.

Neste documento vamos descrever como se pode chegar até à fase 4, fazendo apenas as optimizações mais simples, e fazendo mentalmente toda a actividade de *parsing* e de geração de código *assembly*. A nossa linguagem intermédia (que apenas será usada para facilitar a exposição das ideias relevantes, não sendo necessário utilizá-la na prática), será um subconjunto restrito da própria linguagem *C*.

II. O *assembly* do MIPS

O processador MIPS (de 32 bits) usa uma arquitectura do tipo *load*/*store*. Isto significa que existem instruções para carregar o conteúdo de uma posição de memória (*load*) para um registo e para guardar o conteúdo de um registo na memória (*store*), e que existem instruções que manipulam os valores armazenados em registos; não existem instruções que manipulam directamento o conteúdo de posições de memória.

O processador tem 32 registos de 32 bits, normalmente utilizados em operações aritméticas ou lógicas que envolvem números inteiros, e, caso o coprocessador 1 esteja disponível, 32 registos de 32 bits nesse coprocessador, normalmente utilizados em operações aritméticas que envolvem números representados em vírgula flutuante. Em precisão simples (32 bits, instruções com o sufixo `.s`), nas

(Caso pretenda imprimir este documento, se possível use uma impressora a cores.)

TABELA I
OS 32 REGISTOS DE USO GERAL DO MIPS

nome(s)	número(s)	convenção de uso	preservar?
`$zero`	0	não existe convenção de uso: é **sempre** igual a zero	—
`$at`	1	utilizado na conversão de instruções virtuais em instruções nativas	não
`$v0`–`$v1`	2–3	valores inteiros retornados por uma função; valores temporários de expressões	não
`$a0`–`$a3`	4–7	primeiros quatro argumentos inteiros de uma função	não
`$t0`–`$t7`	8–15	valores temporários que podem ser alterados por uma função	não
`$s0`–`$s7`	16–23	valores temporários que têm de ser preservados por uma função	**sim**
`$t8`–`$t9`	24–25	valores temporários que podem ser alterados por uma função	não
`$k0`–`$k1`	26–27	registos reservados para uso do sistema operativo	**não utilizar**
`$gp`	28	*global pointer*; utilizado pelo compilador para tornar mais eficiente o acesso a variáveis globais	**sim**
`$sp`	29	*stack pointer*; guarda o endereço da posição de memória mais baixa utilizada pelo *stack* (o *stack* cresce no sentido dos endereços mais baixos)	**sim**
`$fp`	30	*frame pointer*; pode ser usado para apontar para a zona do *stack* onde estão armazenadas as variáveis locais de uma função (pode também ser usado como `$s8` se o *frame pointer* não for necessário)	**sim**
`$ra`	31	*return address*; armazena o endereço de retorno de uma função	**sim**

instruções aritméticas e de comparação apenas podem ser utilizados os registos com números pares, pelo que na realidade temos apenas 16 registos disponíveis. Em precisão dupla (64 bits, instruções com o sufixo `.d`), os registos são usados aos pares, sendo apenas especificado na instrução o número do registo par – o menor – do par de registos; mais uma vez, temos apenas 16 registos disponíveis. Note que para transformar código em *assembly* de precisão simples para precisão dupla, ou vice-versa, é "apenas" necessário trocar os sufixos `.s` por `.d`, as directivas `.float` por `.double` e reservar a quantidade de memória apropriada nas directivas `.space`.

Aos 32 registos de uso geral, numerados de 0 a 31, são normalmente dados nomes; por exemplo, em *assembly*, o registo número 0 é designado por `$0` ou por `$zero`. Na tabela I, apresentamos os nomes dos 32 registos, bem como as suas convenções habituais de utilização. Os registos cujo conteúdo deve ser preservado por uma função podem ser alterados dentro dessa função; é, no entanto, preciso garantir que os valores que esses registos tinham quando se entrou na função são os mesmos quando se sai da função (o pedaço de código que chamou a função está à espera que os valores armazenados nesses registos não se alterem).

Os registos do coprocessador 1 são normalmente designados apenas pelos seus números, precedidos por `$f`. Por exemplo, em operações aritméticas de precisão simples o registo número 4 designa-se por `$f4`, e em operações de precisão dupla o par de registos com os números 4 e 5 (64 bits no total) também de designa por `$f4`. No caso da precisão simples, nas instruções aritméticas apenas se podem usar os registos com números pares, visto que os com os números ímpares correspondentes podem ser afectados por essas instruções (trata-se de pormenores de implementação da vírgula flutuante, inexistentes no simulador `MARS`).

Também existem convenções para o uso dos registos em vírgula flutuante: os registos `$f0` a `$f3` são usados para retornar valores de uma função, os registos `$f12` a `$f15` são usados para passar parâmetros para uma função, os registos `$f20` a `$f31` devem ser preservados por uma função (por precaução, em precisão simples deve-se preservar o par de registos), e os restantes registos (`$f4` a `$f11` e `$f16` a `$f19`) são de uso livre.

Um aspecto fundamental da tradução de `C` para *assembly* diz respeito à utilização dos registos, tendo em atenção as suas convenções de uso. Por exemplo, o primeiro argumento inteiro de uma função deve ser passado no registo `$a0` e o segundo em `$a1`. Note-se que os ponteiros (que são endereços de algo que está na memória), para efeitos de *assembly* também são números inteiros de 32 bits (sem sinal). Os argumentos que não podem ficar em registos (a partir do quinto inteiro e do terceiro em vírgula flutuante) são colocados no *stack* pela ordem em que aparecem na chamada da função, de modo a que o primeiro que lá tem de residir fica no endereço mais baixo. (Nota: em AC I nunca será necessário passar argumentos no *stack*. A maneira exacta de o fazer é algo complicada, visto que é necessário reservar no *stack* espaço para os registos que foram passados em registos! O aluno/a curioso/a em verificar como isso é feito poderá, mais tarde, analisar o código gerado pelo compilador de `C` usado nas aulas práticas de AC II.)

Uma boa parte das instruções que manipulam o conteúdo de registos toma a forma

```
INST    DST,SRC1,SRC2
```

onde `INST` é a mnemónica (nome) da instrução (por exemplo, `add`), `DST` é o nome do registo onde é guardado o resultado, `SRC1` é o nome do registo que fornece o primeiro argumento da instrução, e `SRC2` é ou o nome do registo que fornece o segundo argumento ou um valor imediato (se a instrução for do tipo I). As instruções nativas de multiplicação e de divisão usam dois registos implícitos especiais (que não fazem parte dos 32 descritos anteriormente), designados por `$hi` e `$lo`. O leitor ou leitora deve consultar o **apêndice B de [1]** ou o material de apoio de AC I para obter uma lista das instruções disponíveis. Essas instruções, quer nativas quer virtuais (ver explicação a

seguir), serão utilizadas no resto deste documento sem se apresentar mais detalhes. No entanto, por uma questão de princípio, não utilizaremos instruções virtuais para as quais exista uma instrução nativa equivalente.

Para além das instruções habituais, para facilitar a vida aos programadores, a grande maioria dos *assemblers* para o `MIPS` também aceitam instruções ditas virtuais. Estas instruções, que ocorrem inúmeras vezes na maioria dos programas, são convertidas automaticamente pelo *assembler* numa sequência de instruções nativas. Por exemplo, a instrução virtual

```
li      $t0,0x12345678
```

é convertida nas instruções

```
lui     $t0,0x1234
ori     $t0,$t0,0x5678
```

(o registo intermédio também pode ser o `$at`). Se o *assembler* for minimamente inteligente, então a instrução virtual

```
li      $t0,0x9876
```

será convertida apenas na instrução

```
ori     $t0,$zero,0x9876
```

e a instrução virtual

```
li      $t0,-1
```

será convertida apenas na instrução

```
addiu   $t0,$zero,0xFFFF
```

(verifique se o *assembler* que está a usar faz isto). Em alguns casos a conversão de uma instrução virtual para uma ou mais instruções nativas requer o uso de um registo temporário (`$at`), que se encontra reservado precisamente para esse fim.

III. A linguagem C

Cada ficheiro de um programa em `C` [2] pode ter uma mistura de (i) directivas para o preprocessador, que não discutiremos aqui, (ii) declarações de tipos de dados, por exemplo, de estruturas, (iii) declarações de variáveis e de funções, e (iv) definições de variáveis e de funções.

Uma declaração é apenas um anúncio para o compilador de que a variável ou função está definida noutro sítio (ou mais adiante no mesmo ficheiro, ou noutro ficheiro, ou numa biblioteca). Numa definição é reservado espaço para armazenar o conteúdo de uma variável e é produzido código para uma função. Como é evidente, os tipos de dados usados numa declaração têm de coincidir com os da correspondente definição. Por exemplo, se num ficheiro se declarar

```
extern int sum;
extern int square(int x);
```

então daí para diante podemos passar a usar a variável `sum` e a função `square` como se elas já estivessem definidas. O `extern` pode ser omitido na declaração da função, mas não o pode ser na declaração da variável. Mais adiante no mesmo ficheiro, ou noutro ficheiro, podemos defini-las:

```
int sum;
int square(int x) { return x * x; }
```

É apenas nesta altura que o compilador armazena espaço para a variável e produz código para a função. No nosso caso, teriamos, por exemplo,

```
        .data
        .globl  sum
        .align  2
sum:    .space  4

        .text
        .globl  square
square: mult    $a0,$a0
        mflo    $v0
        jr      $ra
```

(em vez da directivas `.align` e `.space` podiamos ter usado, neste caso, a directiva `.word`).

É preciso declarar ou definir uma variável ou função antes de a poder utilizar. Se uma ou mais variáveis ou funções forem utilizadas num programa mas nunca forem definidas (incluindo nas bibliotecas) a última etapa da compilação falha (o *linker* mostra-nos uma mensagem de erro indicando os nomes das que não foram definidas).

Uma variável ou função é reconhecida como tal a partir do momento da sua declaração ou definição até ao fim do bloco em que está inserida (um bloco é delimitado por chavetas: `{` para o iniciar e `}` para o terminar) ou até ao fim do ficheiro se não estiver inserida num bloco. Uma função não pode ser definida dentro de um bloco. Por omissão, as funções e as variáveis definidas fora de um bloco podem ser utilizadas, mediante uma declaração no sítio apropriado, noutros ficheiros (usa-se a directiva *assembly* `.globl`, usualmente abreviada para `.globl`, para tornar o nome da função ou variável visível noutros ficheiros). Quando se pretende que isso não aconteça usa-se `static` antes da definição. Por omissão, as variáveis definidas dentro de um bloco só existem dentro desse bloco (isto é, só é reservado espaço para armazenar o seu conteúdo, em registos ou na memória, dentro do bloco); se se usar `static` antes da sua definição uma variável passa a ser persistente (isto é, apesar de não poder ser utilizada directamente fora do bloco onde foi definida, o seu conteúdo é preservado numa zona de memória usada em exclusivo para esse fim).

A. Variáveis

Para o processador `MIPS` de 32 bits o número de bits e de bytes dos tipos de dados predefinidos são os seguintes (na última linha, tipo representa um qualquer tipo de dados, e a última coluna mostra o valor a colocar a seguir a um `.align` quando se pretende reservar memória para uma variável desse tipo usando a directiva `.space`):

tipo	bits	bytes	`.align`	inicialização
`void`	0	0		

`char`	8	1		`.byte`
				`.ascii`
				`.asciiz`

`short`	16	2	1	`.half`
`int`	32	4	2	`.word`
`long`	32	4	2	`.word`
`float`	32	4	2	`.float`
`double`	64	8	3	`.double`
tipo `*`	32	4	2	`.word`

Cada um dos quatro tipos inteiros pode ser `signed` ou `unsigned`; se não se disser nada a maior parte dos compi-

ladores considera que o tipo é `signed`. Todos os ponteiros têm 32 bits, qualquer que seja o tipo de dados para que esteja a apontar (um ponteiro armazena sempre um endereço, que num processador de 32 bits tem geralmente também 32 bits). O tipo `void` é utilizado para indicar que uma função não retorna nada ou que não tem argumentos; também é utilizado em ponteiros para um tipo arbitrário.

Por exemplo, as linhas

```
int a,b,*c;
double d,*e;
float f;
```

definem seis variáveis capazes de armazenar: dois inteiros (`a` e `b`) e um ponteiro para inteiro (`c`), um número em vírgula flutuante de precisão dupla (`d`) e um ponteiro para um número desse tipo (`e`), e, finalmente, um número em vírgula flutuante de precisão simples (`f`). As variáveis `a`, `b`, `c` e `e` ocupam 4 bytes, podendo cada uma delas residir ou num registo de uso geral ou na memória, a variável `d` ocupa 8 bytes, podendo residir ou num par de registos do coprocessador 1 ou na memória, e a variável `f` ocupa 4 bytes, podendo residir num registo do coprocessador 1 ou na memória.

É possível forçar uma mudança de tipo usando um *cast*. Por exemplo, apesar de a seguinte linha de código ser válida

```
d = a;
```

(nestes casos o compilador gera automaticamente código para converter o inteiro armazenado em `a` num número representado em vírgula flutuante de precisão dupla e armazena-o em `d`), este método de conversão automática entre tipos é por muitos considerada uma má prática de programação. O que se deve fazer é tornar a conversão explícita, como se segue:

```
d = (double)a;
```

O leitor ou leitora poderá consultar [2] ou um outro livro de referência sobre a linguagem de programação `C` para ver que conversões automáticas de tipos são efectuadas quando se misturam vários tipos na mesma expressão.

Quando uma variável inteira com menos de 32 bits é carregada para um registo é necessário inicializar os bits que não são utilizados pelo tipo com o valor correcto: zeros para tipos `unsigned` e cópias do bit do sinal para tipos `signed` (é para isso que existem, por exemplo, as instruções `lbu` e `lb`). Deste modo, no registo fica armazenado o valor da variável promovido para o tipo `int` (sem ou com sinal conforme for o caso), o que torna possível, por exemplo, efectuar comparações (as instruções do `MIPS` para comparar inteiros trabalham sempre com os 32 bits dos registos).

A conversão de tipos inteiros de/para tipos de vírgula flutuante pode ser feita recorrendo às instruções que o `MIPS` disponibiliza para esse efeito. Retomando as variáveis definidas anteriormente, o código

```
d = (double)a;
b = (int)f;
```

dá origem ao seguinte código, quando as variáveis `a`, `b`, `d` e `f` estão armazenadas respectivamente nos registos `$t0`, `$t1`, `$f4`–`$f5` e `$f6`:

```
        mtc1    $t0,$f4
        cvt.d.w $f4,$f4
        cvt.w.s $f8,$f6
        mfc1    $t1,$f8
```

Note que foi necessário utilizar um registo de vírgula flutuante extra (`$f8`) na conversão para inteiro. Se por acaso o valor da variável `f` não fosse mais necessário teria sido possível reutilizar o registo `$f6`, não sendo nesse caso necessário usar qualquer registo extra. Note ainda que depois da instrução `mtc1` o valor armazenado no registo `$f4` ainda está representado em complemento para 2; a instrução apenas copia os bits de um lado para o outro, **sem** alterar a sua representação. É pois necessário alterar a maneira como o número é representado, o que é feito pela instrução `cvt.d.w`. O mesmo se passa, mas por ordem inversa, na outro conversão.

As variáveis persistentes (definidas usando a palavra reservada `static`) têm de ser armazenadas em memória. Por exemplo, o seguinte pedaço de código

```
static int a = 1,b = -1,*c = &a;
double d = 1.0,*e = &d;
float f = -2.0f;
```

pode ser traduzido para *assembly* da seguinte maneira:

```
        .data
a:      .word   1
b:      .word   -1
c:      .word   a
e:      .word   d
d:      .double 1.0
f:      .float  -2.0
```

A ordem pela qual as variáveis são definidas é arbitrária; tomámos a liberdade de definir `e` antes de `d` para ilustrar que, em *assembly*, se pode utilizar uma *label* antes de ela ser definida. Note que uma *label* é o nome que o programador dá a uma variável ou função em *assembly*: nada obriga a que a *label* tenha o mesmo nome que a correspondente variável ou função em `C`, se bem que é usual que assim seja. Note que para o *assembler* a *label* `a` é o endereço onde a variável `a` vai ser armazenada. Logo, a sua utilização no texto em *assembly* não se refere ao valor da variável mas sim ao valor do ponteiro da variável. Para copiar o valor da variável para, por exemplo, o registo `$s0`, terá de ser utilizado código do género

```
        la      $t9,a
        lw      $s0,0($t9)
```

(note a utilização do registo `$t9` para armazenar temporariamente o endereço da variável). Finalmente, note que em *assembly* é geralmente possível somar ou subtrair uma constante (sempre em número de bytes) a um ponteiro e subtrair dois ponteiros (obtendo-se sempre o número de bytes que os separa). Infelizmente o simulador `MARS` é, neste respeito, um bocado limitado; por exemplo, na sua versão 4.2, `la $t0,a-1` não é aceite, mas `la $t0,a+-1` já o é. Estas operações aritméticas com ponteiros **diferem** em *assembly* do que acontece em `C`, como se explica na próxima subsecção. Tanto em `C` como em *assembly*, não é possível somar dois ponteiros, visto que isso não faz qualquer sentido (já subtrair dois ponteiros pode ser usado, por exemplo, para determinar o número de bytes usado pelas nossas *strings*).

B. Acesso à memória usando ponteiros

Como já foi mencionado anteriormente, um ponteiro é um endereço. Na posição de memória correspondente a esse endereço encontra-se armazenada a informação pretendida. Note que o próprio endereço também precisa de espaço para ser armazenado. No nosso caso, como o espaço de endereçamento é de 32 bits, cada endereço pode ser armazenado ou num registo ou numa *word* (4 bytes) de memória.

Um ponteiro para uma variável `x` é obtido colocando um `&` antes do nome da variável (neste caso seria `&x`). O nome de um *array* é um ponteiro para o seu primeiro elemento. Para se obter o valor apontado por um ponteiro usa-se um `*` antes do ponteiro, ou `[0]` depois do ponteiro. Por exemplo, se `p` for um ponteiro para um inteiro, então `*p` e `p[0]` são o inteiro; a segunda notação permite aceder aos inteiros à volta do inteiro apontado pelo ponteiro, como se este fosse o primeiro elemento de um *array* (índices negativos são permitidos); por exemplo, `p[1]` será o inteiro armazenado imediatamente a seguir a `*p` e `p[-1]` será o inteiro armazenado imediatamente antes de `*p`, como se ilustra na figura 1. Como exemplo, considere o seguinte pedaço de código:

```
{
  static int a[4] = { 0,1,2,3 };
  int i,j,*p = &a[3],*q;

  i = p[-1];
  j = *p;
  q = p + 1;
}
```

Neste caso `p` aponta para o terceiro elemento do *array* `a`, pelo que `p[-1]` é o valor armazenado no segundo elemento (se fosse necessário um ponteiro para esse valor ele seria `p-1` ou `&a[2]`). Pressupondo que as variáveis `i`, `j`, `p` e `q` estão armazenadas respectivamente nos registos `$t0`, `$t1`, `$t2` e `$t3`, e tomando em consideração que a variável `a` é persistente (por causa do `static`), o código anterior dá origem ao seguinte código *assembly*:

```
        .data
a:      .word   0,1,2,3

        .text
        la      $t2,a+3*4   # p
        lw      $t0,-4($t2) # i
        lw      $t1,0($t2)  # j
        addiu   $t3,$t2,1*4 # q
```

Note que como se pretende o endereço de `a[3]` foi preciso somar ao endereço do início do *array* 3 vezes o número de bytes ocupados por um inteiro, ou seja, foi preciso somar `3*4`. Se em vez de `p = &a[3]` se tivesse `p = &a[k]` com o inteiro `k` armazenado no registo `$t4`, a inicialização de `p` passaria a ser:

```
        la      $t2,a
        sll     $t9,$t4,2   # 4*k
        addu    $t2,$t2,$t9 # p
```

Note que foi necessário usar um registo adicional (neste caso `$t9`) para armazenar o resultado intermédio da multiplicação de `k` por 4; multiplicações por potências de 2 podem e devem ser feitas usando a instrução `sll`.

Somar a um ponteiro um número inteiro i (ou subtrair $-i$) dá origem a um outro ponteiro, que aponta para i elementos para a frente (se i for negativo, na realidade é para trás). Em *assembly* o novo ponteiro será obtido somando ao ponteiro original i vezes o número de bytes do tipo de dados para onde o ponteiro aponta (ver figura 1).

endereço	ponteiro	memória
...	...	...
0x10000010	p+1	*(p+1) ou p[1]
0x1000000C	p ou p+0	*p ou p[0]
0x10000008	p-1	*(p-1) ou p[-1]
...	...	...

Fig. 1 - Exemplo de acesso a uma zona de memória usando ponteiros. Neste caso p é um ponteiro para um tipo de dados que ocupa 4 bytes (por exemplo `int` ou `float`), e tem o valor `0x1000000C`.

C. Funções

No `MIPS` uma chamada de uma função é feita geralmente usando a instrução `jal` e o retorno de uma função é geralmente feito usando a instrução `jr`. A instrução `jal` armazena no registo `$ra` o endereço da instrução que se lhe segue e depois armazena no *program counter* o endereço que se encontra codificado na própria instrução, passando deste modo a execução do programa para a função que foi chamada. Para que este mecanismo funcione como deve ser quando existem chamadas encadeadas a funções (isto é, uma função chama outra, que por sua vez chama outra, etc.), todas as funções que chamem qualquer função **têm** de salvaguardar no *stack* o seu próprio endereço de retorno, que se encontra no registo `$ra`, antes de chamar uma função. Uma função só não precisa de salvaguardar o conteúdo do registo `$ra` se não chamar qualquer função, visto que neste caso o registo `$ra` não será alterado.

Outro aspecto muito importante relacionado com uma função tem a ver com o sítio (registo ou memória no *stack*) onde cada uma das suas variáveis locais será armazenada. Variáveis locais cujo valor é necessário **antes** e **depois** de uma chamada a uma função têm de ser armazenadas ou num registo do tipo `s`, ou numa zona de memória do *stack*; só assim é que há garantia que o seu valor é preservado (se as convenções de uso dos registos forem respeitadas!). As outras variáveis locais podem ser armazenadas em registos do tipo `t`, `a` ou `v`, caso haja registos desses tipos em número suficiente (se não houver, vão também para o *stack*). Todas as decisões sobre a localização de cada variável local devem ser tomadas antes de se começar a escrever código.

O esqueleto típico em *assembly* de uma função será (aqui ilustrado para uma função com o nome `f`):

```
        .text
        .globl  f
f:      PRÓLOGO
        CÓDIGO
        EPÍLOGO
        jr      $ra
```

Este esqueleto é composto por três blocos:

PRÓLOGO No primeiro, todas os registos cujo conteúdo tem de ser preservado é copiado para o *stack*. Além disso, também é reservado espaço no *stack* para as

variáveis locais que não vão ficar armazenadas em registos, e, caso seja necessário, os parâmetros de entrada da função são copiados para outros registos. O valor do registo `$sp` é ajustado (logo no início) da maneira apropriada.

`CÓDIGO` No segundo, coloca-se o código *assembly* que resulta da tradução do código `C`. O valor de retorno da função, caso o haja, é colocado no registo apropriado.

`EPÍLOGO` No terceiro, é reposto o conteúdo original nos registos cujo conteúdo tinha de ser preservado. No fim, o conteúdo original do registo `$sp` também é reposto.

Pode acontecer que não seja necessário preservar o conteúdo de qualquer registo e que não seja preciso reservar espaço no *stack* para variáveis locais. Se isso acontecer os blocos `PRÓLOGO` e `EPÍLOGO` ficarão vazios, tal como acontece com a função `square` da página 3

Como primeiro exemplo, considere a seguinte função:

```
static int fact(int n)
{
  if(n == 0)
    return 1;
  else
    return n * fact(n - 1);
}
```

Como esta função pode chamar uma função (neste caso, a própria), é necessário preservar o registo `$ra`. Também é preciso guardar o valor da variável `n` num registo que é preservado, já que só é possível efectuar a multiplicação depois de se obter o resultado de `fact(n - 1)`; vamos pois guardar `n` num registo do tipo `s`, por exemplo `$s0`. Será então preciso guardar o conteúdo de dois registos no *stack* (8 bytes ao todo). Neste exemplo simples, é possível armazenar o valor a retornar pela função directamente no registo `$v0`. Sendo assim, uma versão em *assembly* possível desta função será (omitiu-se o `.globl` porque a função é `static`):

```
        .text
fact:   addiu   $sp,$sp,-8
        sw      $ra,0($sp)
        sw      $s0,4($sp)
        or      $s0,$0,$a0        # n
        bne     $a0,$zero,fact_1
        ori     $v0,$zero,1
        j       fact_2
fact_1: addi    $a0,$s0,-1        # n-1
        jal     fact
        mult    $v0,$s0
        mflo    $v0
fact_2: lw      $ra,0($sp)
        lw      $s0,4($sp)
        addiu   $sp,$sp,8
        jr      $ra
```

Note que o *stack* cresce para baixo e que o registo `$sp` contém o endereço da posição de memória mais baixa **ocupada** pelo *stack*. Logo, para reservar 8 bytes no *stack* a primeira coisa a fazer é subtrair 8 ao valor de `$sp`. Só depois se pode usar o novo espaço.

Como segundo exemplo, considere a seguinte função:

```
float f(int n)
{
  static int a[7] = { 0,2,4,6,1,3,5 };
  int b[7];
  float r;

  r = 1.0;
  while(n != 0)
  {
    n = n - 1;
    b[n] = a[n];
    a[n] = a[n] + 1;
    r += f(n);
    a[n] = b[n];
  }
  return r;
}
```

Neste caso, é necessário preservar o conteúdo do registo `$ra` e o das variáveis `n` e `r`; optámos por armazenar estas variáveis nos registos `$s0` e `$f20`. Além disso, o *array* `b` é local, pelo que tem de ser armazenado no *stack*. No total, é então necessário reservar espaço para 40 bytes no *stack*, correspondentes a 9 inteiros de 32 bits e a um número em vírgula flutuante de precisão simples, também de 32 bits. Temos de decidir em que posição no *stack* vamos colocar cada um dos registos que têm de ser preservados e onde vamos colocar as variáveis locais (neste caso apenas o *array* `b`). Uma possibilidade é a seguinte:

endereço	memória	comentário
$\$sp + 12 + 4k$	`b[k]`	$k = 0, 1, \ldots, 6$
$\$sp + 8$	`$f20`	o registo vai armazenar `r`
$\$sp + 4$	`$s0`	o registo vai armazenar `n`
`$sp`	`$ra`	

O código *assembly* resultante será então, por exemplo:

```
        .data
a:      .word   0,2,4,6,1,3,5

        .text
        .globl  f
f:      addiu   $sp,$sp,-40
        sw      $ra,0($sp)
        sw      $s0,4($sp)
        s.s     $f20,8($sp)
        or      $s0,$zero,$a0 # n
        ori     $t0,$zero,1
        mtc1    $t0,$f20
        cvt.s.w $f20,$f20     # r
f_1:    beq     $s0,$zero,f_2
        addi    $s0,$s0,-1
        sll     $t0,$s0,2     # 4*n
        addiu   $t1,$sp,12
        addu    $t1,$t1,$t0   # &b[n]
        la      $t2,a
        addu    $t2,$t2,$t0   # &a[n]
        lw      $t0,0($t2)
        sw      $t0,0($t1)
        addi    $t0,$t0,1
        sw      $t0,0($t2)
        or      $a0,$0,$s0
```

```
        jal     f
        add.s   $f20,$f20,$f0
        sll     $t0,$s0,2
        addiu   $t1,$sp,12
        addu    $t1,$t1,$t0
        la      $t2,a
        addu    $t2,$t2,$t0
        lw      $t0,0($t1)
        sw      $t0,0($t2)
        j       f_1
f_2:    mov.s   $f0,$f20
        lw      $ra,0($sp)
        lw      $s0,4($sp)
        l.s     $f20,8($sp)
        addiu   $sp,$sp,40
        jr      $ra
```

Tal como no exemplo anterior, foi necessário no prólogo transferir a variável `n` do registo `$a0` para o registo `$s0`.

O último exemplo que vamos apresentar sobre este assunto é mais complicado. Como **sai fora** do âmbito da matéria leccionada em AC I, deve apenas ser analizado pelos mais curiosos (e corajosos). Alguns compiladores disponibilizam uma função chamada `alloca` que reserva espaço para uma variável no *stack*. Esse espaço é automaticamente libertado quando a função que chamou a função `alloca` retorna. O uso desta função é mais cómodo e mais eficiente que a utilização das funções `malloc` e `free` que são habitualmente utilizadas para pedir e libertar uma zona de memória. A sua implementação *inline* é trivial quando se usa um *stack frame* e o registo `fp`. O *stack frame*, por sua vez, é de grande utilidade quando se está a correr um *debugger*, pois permite descobrir que funções foram chamadas (e com que argumentos) até se chegar a uma determinada linha de código. Para simplificar, vamos ignorar aqui esses pormenores.

Suponha que o seu código tem uma função em que um dos argumentos é um ponteiro para um *array* e que essa função modifica o conteúdo do *array*. Suponha ainda que quer chamar essa função mas que não quer que o *array* seja modificado. Uma solução possível será fazer uma cópia do *array* para outra zona de memória e depois chamar a função usando como ponteiro o ponteiro para a cópia. No fim, liberta-se a memória ocupada pela cópia. É isso que a seguinte função faz (aproveitamos este exemplo para usar um ponteiro para uma função: em `C` o nome de uma função é um ponteiro para a sua primeira instrução):

```
int f(int n,int *a,int (*g)(int,int *))
{
  int i,*b,*bb;

  b = (int *)alloca(n * sizeof(int));
  for(bb = b,i = n;i != 0;i--)
    *bb++ = *a++;
  return g(n,b);
}
```

Usando o registo `$fp` de uma maneira **não *standard*** o código assembly correspondente será:

```
        .text
        .globl  f
f:      addiu   $sp,$sp,-8
        sw      $ra,0($sp)
        sw      $fp,4($sp)
        or      $fp,$zero,$sp
        sll     $t0,$a0,2     # 4*n
        subu    $sp,$sp,$t0   # b
        or      $t1,$zero,$sp # bb
        or      $t0,$zero,$a0 # i
f_1:    beq     $t0,$zero,f_2
        lw      $t2,0($a1)    # *a
        addiu   $a1,$a1,4     # a++
        sw      $t2,0($t1)    # *bb
        addiu   $t1,$t1,4     # bb++
        addi    $t0,$t0,-1    # i--
        j       f_1
f_2:    or      $a1,$zero,$sp
        jalr    $a2
        lw      $ra,0($fp)
        lw      $fp,4($fp)
        addiu   $sp,$fp,8
        jr      $ra
```

Note que `sizeof(int)` é igual a quatro (é o número de bytes utilizados para armazenar um inteiro), e que a função `alloca` foi implementada *inline*, isto é, no próprio local (sem `jal` e correspondente `jr $ra`), em apenas duas linhas de código. Note ainda que neste caso a variável `b` está armazenada no registo `$sp`! Para chamar a função `f` usando como último argumento uma função com o nome, por exemplo, `ttt`, para inicializar o registo `$a2` com o endereço dessa função basta fazer

```
        la      $a2,ttt
```

no nosso código *assembly*; é claro que `ttt` terá de ser a *label* colocada no início dessa função.

D. Expressões aritméticas e lógicas

Em `C` uma atribuição (em Inglês, *assignment*) tem a forma

```
lvalue = rvalue;
```

onde `lvalue` é normalmente o nome de uma variável e onde `rvalue` é uma expressão. Note que `lvalue` não pode ser uma constante, isto é, tem de ser qualquer coisa associada a um sítio (registo ou memória) onde o seu valor está armazenado. Por exemplo, se `a` for uma variável inteira e `p` for um ponteiro para inteiro, o código `p = &a` é válido, mas o código `&a = p` não o é, apesar de ambos os lados terem o mesmo tipo (o endereço de `a` não pode ser alterado!). A atribuição

```
lvalue = lvalue op rvalue;
```

onde `op` é uma operador binário, pode ser escrita na forma compacta

```
lvalue op= rvalue;
```

Por exemplo, para substituir `a` pelo seu dobro pode-se usar

```
a *= 2;
```

ou

```
a <<= 1;
```

Uma variável usada num `rvalue` pode, opcionalmente, ser pré- ou pós-incrementada (`++`) ou pré- ou pós-decrementada (`--`). Por exemplo, se numa expressão aparecer

TABELA II
OPERAÇÕES ELEMENTARES E SUA TRADUÇÃO (**PARA INTEIROS COM SINAL**), AGRUPADOS POR ORDEM DECRESCENTE DE PRECEDÊNCIA.

operação	implementação
`t1 = t2 * t3;`	`mul $t1,$t2,$t3`
`t1 = t2 / t3;`	`div $t1,$t2,$t3`
`t1 = t2 % t3;`	`rem $t1,$t2,$t3`
`t1 = t2 + t3;`	`add $t1,$t2,$t3`
`t1 = t2 - t3;`	`sub $t1,$t2,$t3`
`t1 = t2 << t3;`	`sllv $t1,$t2,$t3`
`t1 = t2 >> t3;`	`srav $t1,$t2,$t3`
`t1 = t2 < t3;`	`slt $t1,$t2,$t3`
`t1 = t2 <= t3;`	`sle $t1,$t2,$t3`
`t1 = t2 > t3;`	`sgt $t1,$t2,$t3`
`t1 = t2 >= t3;`	`sge $t1,$t2,$t3`
`t1 = t2 == t3;`	`seq $t1,$t2,$t3`
`t1 = t2 != t3;`	`sne $t1,$t2,$t3`
`t1 = t2 & t3;`	`and $t1,$t2,$t3`
`t1 = t2 ^ t3;`	`xor $t1,$t2,$t3`
`t1 = t2 | t3;`	`or $t1,$t2,$t3`
`t1 = t2 && t3;`	ver texto
`t1 = t2 || t3;`	ver texto

`a++` o valor de `a` é usado na expressão e só depois é que o seu valor é incrementado (isto é, `a` é substituido por `a+1`); se numa expressão aparecer `--a` o valor de `a` é primeiro decrementado (isto é, `a` é substituido por `a-1`) e só depois é que o seu valor é usado. Como numa expressão a ordem de avaliação de operadores comutativos é arbitrária, deve-se evitar escrever expressões do género `a + a++`; dependendo do compilador, o primeiro `a` tanto se pode estar a referir ao valor antigo como ao novo valor da variável.

Os bits de um valor inteiro usado numa expressão podem ser todos negados usando o operador `~`. Por exemplo, como 1 em binário é $0000\ldots0001_2$, `~1` é, em binário, $1111\ldots1110_2$. A sua tradução para *assembly* usa a instrução virtual `not`, que é implementada com a instrução nativa `nor`. Existe também o operador `!` que transforma um valor inteiro igual a zero no número inteiro 1 e que transforma um valor inteiro diferente de zero no número inteiro 0; trata-se pois de uma espécie de negação que não é feita bit a bit. Por exemplo, `!a` toma o valor 0 se `a` for diferente de zero e toma o valor 1 se `a` for igual a zero. A sua tradução para *assembly* usa a instrução nativa `sltiu` com um valor imediato de 1 (verifique!).

Todas as expressões podem ser decompostas no encadeamento de uma ou mais operações elementares. Cada operação elementar é da forma `arg1 op arg2`, onde `op` é um operador binário. Existem operadores mais prioritários que outros; por exemplo, as multiplicações são feitas antes das adições. É possível alterar a ordem com que as operações são efectuadas utilizando parêntesis. Na tabela II apresentam-se as várias operações elementares disponíveis (guardando o resultado numa variável), bem como a sua tradução para *assembly* para o tipo `signed int`. Como exercício, o leitor ou leitora deverá implementar os operadores aritméticos e os relacionais (comparações) para os tipos `unsigned int`, `float` e `double`.

Os operadores `&&` e `||` são especiais. O primeiro é igual a 1 apenas quando os seus dois operandos são diferentes de zero (caso contrário é igual a 0), e o segundo é igual a 0 apenas quando os seus dois operandos são iguais a zero (caso contrário é igual a 1); em ambos os casos o segundo operando **só é avaliado** caso o primeiro operando não seja suficiente para decidir qual é o resultado final. Por exemplo, em

```
i = j || f(k);
```

a função `f` só é chamada se `j` for igual a 0 (isto porque se `j` for diferente de 0 já se sabe que o resultado final vai ser 1). A tradução de cada um destes dois operadores para *assembly* requer pois saltos condicionais.

Um compilador decompõe uma expressão complexa numa sequência de operações elementares, encadeadas umas nas outras. Os resultados das operações elementares intermédias têm de ser armazenados em variáveis temporárias, para posteriormente serem utilizados como operandos em operações elementares subsequentes. Em expressões complexas, descobrir qual é o número mínimo de variáveis temporárias a utilizar requer algum engenho.

Como primeiro exemplo, considere a seguinte linha de código (todas as variáveis são do tipo `int` e têm nomes que sugerem em que registos estão armazenadas):

```
s0 = (s1 + s2 - 1) * (s3 + 3 * s4);
```

Um compilador, ao converter esta linha de código para a sua linguagem intermédia (no nosso caso, a linguagem intermédia é uma versão simplificada do `C` que só suporta operações elementares), gera código como o seguinte (incluimos em comentários o correspondente código em *assembly*):

```
t9 = s1 + s2; // add  $t9,$s1,$s2
t9 = t9 - 1;  // subi $t9,$t9,1
t8 = 3 * s4;  // mul  $t8,$s4,3
t8 = t8 + s3; // add  $t8,$t8,$s3
s0 = t9 * t8; // mul  $s0,$t9,$t8
```

Mais concretamente, o compilador começa por avaliar as sub-expressões mais internas (neste caso, as que estão dentro de parêntesis), substituindo-as pelos valores calculados, e depois continua a aplicar o mesmo método (calcular as sub-expressões mais internas, substituindo-as pelos valores calculados), até se obter o resultado final. Cada linha do código anterior corresponde a apenas uma instrução em *assembly*. Com prática, o leitor ou leitora será capaz de gerar *assembly* directamente a partir do código original.

Repare que usámos os registos temporários com os números mais altos; uma segunda expressão reutilizaria esses registos. Esta maneira de fazer as coisas simplifica a gestão dos nomes dos registos se se utilizarem os registos com os números mais baixos para armazenar as variáveis locais.

Como segundo exemplo, considere a seguinte linha de código:

```
s0 = (1 + s1 * (s2 + 1)) * (s2 + 1);
```

O código gerado será, por exemplo:

```
t9 = s2 + 1;  // addi $t9,$s2,1
t8 = s1 * t9; // mul  $t8,$s1,$t9
t8 = t8 + 1;  // addi $t8,$t8,1
s0 = t8 * t9; // mul  $s0,$t8,$t9
```

Repare que o valor de `t9` foi usado duas vezes, visto que `s2 + 1` apareceu em dois sítios diferentes na mesma expressão.

Como terceiro exemplo, considere a seguinte linha de código:

```
s0 = (s1 == 0) && (f(s2) == 0);
```

(Mais tarde veremos que expressões deste género, sem a atribuição, aparecem com frequência em linhas de código relacionadas com o controlo do fluxo de execução de um programa. Compare este exemplo com o que será apresentado mais adiante.) A variável `s0` apenas pode ficar com dois resultados possíveis: 0 ou 1. Uma maneira possível de conseguir isto, na nossa linguagem `C` simplificada, será:

```
      if(s1 != 0) goto val0;
      v0 = f(s2);
      if(v0 != 0) goto val0;
val1: s0 = 1;
      goto done;
val0: s0 = 0;
done:
```

que em *assembly* dá:

```
        bne     $s1,$zero,val0
        or      $a0,$zero,$s2
        jal     f
        bne     $v0,$zero,val0
val1:   ori     $s0,$zero,1
        j       done
val0:   ori     $s0,$zero,0
done:
```

Repare que se o teste `(s1 == 0)` for falso o resultado final é conhecido imediatamente. Como as instruções de salto condicional alteram o fluxo de execução do programa quando a condição testada é verdadeira, a condição a testar é a negação da condição original, ou seja, será `!(s1 == 0)`, que é o mesmo que `(s1 != 0)`. O mesmo se passa com o segundo teste.

Como exemplo final, considere a seguinte linha de código (muito parecida com a anterior):

```
s0 = (s1 == 0) || (f(s2) == 0);
```

Neste caso a tradução desta linha de código para a nossa linguagem intermédia será, por exemplo,

```
      if(s1 == 0) goto val1;
      v0 = f(s2);
      if(v0 == 0) goto val1;
val0: s0 = 0;
      goto done;
val1: s0 = 1;
done:
```

que em *assembly* dá:

```
        bne     $s1,$zero,val1
        or      $a0,$zero,$s2
        jal     f
        bne     $v0,$zero,val1
val0:   ori     $s0,$zero,0
        j       done
val1:   ori     $s0,$zero,1
done:
```

Repare que se o teste `(s1 == 0)` for verdadeiro o resultado final é conhecido imediatamente.

E. Controlo do fluxo de execução

Em `C` existem várias palavras reservadas que estão relacionadas com a alteração (possivelmente condicional) do fluxo de execução de um programa, a saber: `goto`, `if`, `else`, `for`, `while`, `do`, `break`, `continue`, `switch`, `case` e `default`.

- O `goto`, cuja existência numa linguagem de programação de alto nível é por muitos considerada perniciosa por poder dar origem a código muito pouco estruturado (e potencialmente difícil de entender), permite transferir o fluxo de execução para outra parte do código. O seu uso em `C` e *assembly* é semelhante (em *assembly*, a instrução correspondente a `goto label;` é `j label` (ou `beq $zero,$zero,label`).
- O `if` toma ou a forma

```
if(exp)
  BLOCO_V
```

ou a forma

```
if(exp)
  BLOCO_V
else
  BLOCO_F
```

(Se um bloco for composto por um único comando as chavetas (`{` e `}`), que normalmente delimitam o conteúdo do bloco, podem ser omitidas.) Se a expressão `exp` for diferente de zero, então o controlo do fluxo de execução passa para o início do bloco `BLOCO_V`; se for igual a zero, e se o `else` estiver presente, então o controlo do fluxo de execução passa para o início do bloco `BLOCO_F`. Por exemplo, o seguinte código (comparar com o terceiro exemplo da subsecção anterior)

```
if((s1 == 0) && (f(s2) == 0))
  BLOCO_V
else
  BLOCO_F
```

pode ser traduzido para *assembly* como se segue:

```
if:     bne     $s1,$zero,else
        or      $a0,$zero,$s2
        jal     f
        bne     $v0,$zero,else
then:   BLOCO_V
        j       end
else:   BLOCO_F
end:
```

Repare que não foi preciso criar uma variável temporária para guardar o valor da expressão `exp`. A *label* `val1` do exemplo da secção anterior (que resolvemos passar a chamar aqui `then`) é o sítio certo para por o código do bloco `BLOCO_V`, e a *label* `val0` (agora com o nome `else`) é o sítio certo para por o código do bloco `BLOCO_F`. Por uma questão de coerência com o que se segue, a *label* `done` passou a ter o nome `end`. A *label* no ínicio do bloco pode ser útil se for necessário transferir o fluxo de execução do programa para lá.

Note que um `else` está sempre associado ao `if` anterior (tendo em atenção as chavetas que possam existir). Assim, o `else` do código

```
if(t1)      // if1:   beq $t1,$zero,end1
  if(t2)    // if2:   beq $t2,$zero,else2
    t3 = 1; //        ori $t3,$zero,1
  else      //        j   end2
    t3 = 0; // else2: ori $t3,$zero,0
            // end2:
            // end1:
```

é o `else` do `if(t2)`, tal como a indentação do código sugere, enquando que o `else` do código

```
if(t1)      // if1:   beq $t1,$zero,else1
{           // if2:   beq $t2,$zero,end2
  if(t2)    //        ori $t3,$zero,1
    t3 = 1; // end2:  j   end1
}           // else1: ori $t3,$zero,0
else        // end1:
  t3 = 0;
```

é o `else` do `if(t1)`.

• O `while` toma a forma

```
while(exp)
  BLOCO
```

O `BLOCO` é executado enquanto a expressão `exp` for diferente de zero. Uma sua implementação possível é (as *labels* podem em alguns casos não ser necessárias mas são uma ajuda preciosa para a explicação que se segue)

```
while:  if((exp) == 0)
          goto end;
body:   BLOCO
        goto while;
end:
```

Dentro do `BLOCO` um `break` permite sair do `while` (isto consegue-se fazendo `goto end`), e um `continue` dá um salto para o fim do bloco (isto consegue-se fazendo um `goto` para uma *label* colocada no fim do bloco, ou, neste caso, fazendo `goto while`).

• O `do`–`while` toma a forma

```
do
  BLOCO
while(exp);
```

O `BLOCO` é executado uma vez, e depois, enquanto a expressão `exp` for diferente de zero, volta a ser executado. Uma sua implementação possível é

```
do:    BLOCO
test:  if(exp)
         goto do;
end:
```

Tal como para o caso anterior, dentro do `BLOCO` um `break` permite sair do `do`–`while` (isto consegue-se fazendo `goto end`), e um `continue` dá um salto para o fim do bloco (isto consegue-se fazendo um `goto` para uma *label* colocada no fim do bloco, ou, neste caso, fazendo `goto test`).

• O `for` toma a forma

```
for(com1;exp2;com3)
  BLOCO
```

que é equivalente a

```
for:  com1;
loop: if((exp2) == 0)
        goto end;
body: BLOCO
next: com3;
      goto loop;
end:
```

Como de costume, dentro do `BLOCO` um `break` permite sair do `for` (isto consegue-se fazendo `goto end`), e um `continue` dá um salto para o fim do bloco (isto consegue-se fazendo um `goto` para uma *label* colocada no fim do bloco, ou, neste caso, fazendo `goto next`).

• O `switch` toma a forma

```
switch(int_exp)
{
  case N1:
    BLOCO_1
    break;
  case N2:
    BLOCO_2
    break;
   ⋮
  default:
    BLOCO_D
    break;
}
```

(`N1`, `N2`, etc., designam números inteiros distintos). A expressão inteira `int_exp` é avaliada e se o seu valor for igual ao número de um `case`, o bloco correspondente é executado. Se isso não acontecer, e se existir um `default`, o bloco `BLOCO_D` é executado. Se um `break` for omitido, a execução continua no bloco seguinte; caso contrário, a execução passa para o fim do `switch`. A ordem dos `case` e do `default` é arbitrária (é usual colocar o `default` no fim, mas pode ficar em qualquer sítio).

Uma implementação possível do `switch` em `C` é (existem muitas outras que reduzem ou mesmo eliminam os testes que estão presentes nesta implementação)

```
switch: tmp = int_exp;
        if(tmp == N1)
        { switch1: BLOCO_1; goto end; }
        if(tmp == N2)
        { switch2: BLOCO_2; goto end; }
         ⋮
        switchD: BLOCO_D; goto end;
end:
```

(Não existe um `if(tmp == ...)` antes da linha com o `BLOCO_D`.) Um `break` fora de um `for`, `while` ou `do-while` implica a colocação de um `goto end` nesse sítio; a sua ausência no fim de um bloco do `switch` implica a colocação de um `goto` para o início do bloco seguinte, saltando por cima do teste que foi introduzido pela nossa implementação do `switch`. Por exemplo, se não existisse um `break` depois do `BLOCO_1` no exemplo anterior, o `goto end;` que está na implementação logo a seguir a esse bloco seria substituido por `goto switch2;`.

F. Estruturas e uniões

Em C é possível criar tipos de dados que agregam vários tipos elementares. Por exemplo,

```
struct nome1
{
  int ix;
  float fx;
};
```

declara um novo tipo de dados, chamado `struct nome1` que tem dois campos, o primeiro do tipo `int` e o outro do tipo `float`. O código `struct nome1 p;` pode ser usado para definir uma variável deste tipo com o nome `p`. Estar sempre a escrever `struct` sempre que se declara ou define uma variável que é uma estrutura pode ser uma maçada. Felizmente tal não é preciso se se declarar o tipo como se segue:

```
typedef struct nome1
{
  int ix;
  float fx;
}
nome2;
```

O nome da estrutura, neste caso `nome1`, pode ser omitido. O nome do tipo é apenas `nome2` (podia ser sido utilizado o mesmo nome em cima e em baixo sem qualquer problema). Agora basta escrever `nome2 p;` para definir a variável `p`.

Ao contrário dos *arrays*, com estruturas o nome da variável representa toda a estrutura e não um ponteiro para o seu início. O acesso aos campos da estrutura é feito colocando a seguir ao nome da variável um ponto (`.`) seguido pelo nome do campo. Se a variável for um ponteiro para a estrutura, em vez do ponto usa-se `->`. O código

```
static nome2 p = { 1,2.0f };
static void f(nome2 *q)
{
  p.ix = (int)(q->fx);
}
```

mostra um exemplo de como se podem usar os campos de uma estrutura (neste caso, os parêntesis à volta de `q->fx` são opcionais). Na memória esta estrutura ocupa duas *words* consecutivas: primeiro o `int` (com um *offset* de 0) e depois o `float` (com um *offset* de 4). O código anterior pode ser traduzido para *assembly* da seguinte maneira:

```
        .data
p:      .word   1
        .float  2.0

        .text
f:      l.s     $f4,4($a0) # q->fx
        cvt.w.s $f4,$f4    # to int
        la      $t0,p      # &p
        s.s     $f4,0($t0) # p.ix
        jr      $ra
```

Note que se usou a instrução `s.s` para transferir directamente o inteiro para a memória, sem passar primeiro por um registo de uso geral.

Na figura 2 apresentamos a declaração de um novo tipo (uma estrutura). Para cada campo indicamos qual é o alinhamento que o campo tem de ter na memória (o endereço desse campo tem de ser um múltiplo do alinhamento), qual é o seu tamanho (em bytes), e qual é o seu *offset* (isto é, qual é a sua posição em relação ao início da estrutura). O alinhamento da estrutura é o maior dos alinhamentos dos seus campos, e o seu tamanho é o *offset* que um campo extra com um alinhamento igual ao da estrutura teria se fosse colocado no fim da estrutura. Em C uma variável deste tipo pode ser definida e inicializada da seguinte maneira:

código C	alinhamento	tamanho	*offset*
`typedef struct`			
`{`			
`  char var1;`	1	1	0
`  int var2;`	4	4	4
`  char var3[6];`	1	6×1	8
`  float var4[4];`	4	4×4	16
`  char var5;`	1	1	32
`}`			
`nome3;`	4	36	

Fig. 2 - Uma estrutura mais complicada.

```
nome3 p =
{
  'a',
  3,
  "meio", // ou { 'm','e','i','o','\0' }
  { 1.0f,2.0f },
  'f'
};
```

a que corresponde, em *assembly*:

```
        .data
        .globl  p
p:      .byte   'a',0,0,0
        .word   3
        .byte   'm','e','i','o',0,0,0,0
        .float  1.0,2.0,0.0,0.0
        .byte   'f',0,0,0
```

(note que os elementos não explicitamente inicializados dos *arrays* são postos a zero, e que se introduziram alguns bytes de *padding* (a verde) logo a seguir aos campos do tipo `char` para que tudo fique alinhado correctamente). Neste caso é possível reduzir o tamanho da estrutura para 28 bytes se se colocarem todos os `char`s juntos.

Como exemplo final, considere o seguinte código:

```
int f(nome3 *q)
{
  int i,c; // i->$t0, c->$v0

  p.var1 = q->var1;
  for(i = c = 0;i < 4;i++)
    if(q->var4[i] > 0.0f)
      p.var3[i] = '0' + c++;
  p.var2 = c;
  p.var5 = '\0';
  *q = p;
  return c;
}
```

Uma maneira possível que traduzir este código para *assembly*, respeitando a atribuição de registos sugerida pelo comentário, é a seguinte:

```
        .text
        .globl  f
f:      la      $t1,p       # &p
        lb      $t2,0($a0)  # q->var1
        sb      $t2,0($t1)  # q.var1
        mtc1    $zero,$f6   # 0.0f
        ori     $t0,$zero,0
        ori     $v0,$zero,0
loop1:  bge     $t0,4,end1
if2:    sll     $t2,$t0,2   # 4*i
        addu    $t2,$t2,$a0
        l.s     $f4,16($t2) # q->var4[i]
        c.gt.s  $f4,$f6
        bc1f    next1
then2:  addu    $t2,$t1,$t0
        addi    $t3,$v0,'0'
        addi    $v0,$v0,1   # c++
        sb      $t3,8($t2)  # p.var3[i]
next1:  addi    $t0,$t0,1   # i++
        j       loop1
end1:   sw      $v0,4($t1)
        sb      $zero,32($t1)
        ### início de *q = p;
        ### &p em $t1, q em $a0
        ori     $t2,$zero,9 # 36/4
copy3:  lw      $t3,0($t1)
        addiu   $t1,$t1,4
        sw      $t3,0($a0)
        addiu   $a0,$a0,4
        addi    $t2,$t2,-1
        bne     $t2,$zero,copy3
        ### fim de *q = p;
        jr      $ra
```

Note que a linha `*q = p` dá origem a uma copia de 36 bytes (o tamanho da estrutura), isto é, $36/4 = 9$ palavras, da zona de memória onde está armazenado `p` para a zona de memória apontada por `q`.

Uma união tem um princípio de funcionamento igual ao de uma estrutura e usa-se da mesma maneira. A única diferença é que todos os campos têm um *offset* de 0, estando por isso sobrepostos. O tamanho de uma união é igual ao menor múltiplo do maior alinhamento dos seus campos que não é inferior ao maior dos tamanhos dos seus campos. Por outras palavras, o tamanho de uma união é o maior dos tamanhos dos seus campos, ao qual se poderá ter de adicionar algum *padding* para garantir que o tamanho da união é um múltiplo do maior dos alinhamentos dos seus campos. As uniões são declaradas ou definidas usando `union` em vez de `struct`. O programador tem de ter o cuidado de se lembrar qual o tipo de dados que foi colocado mais recentemente numa união, de modo a evitar surpresas desagradáveis.

Uma união pode ser utilizada para ver como um tipo de dados é representado. Por exemplo, uma união com dois campos, um deles um `int` e o outro um *array* de 4 `char`s pode ser usado para inspecionar (ou modificar) individualmente cada um dos bytes do inteiro (que estamos a supor tem 32 bits, sem ter de usar operações aritméticas e lógicas. Em particular, código do género

```
union
{
  int i;
  unsigned char c[4];
}
u;
unsigned int x;

u.i = 0x11223344;
x = (unsigned int)u.c[1];
```

colocará na variável `x` o valor `0x33` se a arquitectura for ***little endian***, na qual o byte **menos** significativo de um inteiro é armazenado no endereço mais baixo, ou o valor `0x22` se a arquitectura for ***big endian***, na qual o byte **mais** significativo de um inteiro é armazenado no endereço mais baixo (o MIPS que usamos nas aulas práticas de AC I é *little endian*). O seguinte exemplo mostra outra aplicação possível de uma união:

```
int f_as_i(float f)
{
  union
  {
    int i;
    float f;
  }
  u;

  u.f = f;    /* write as float */
  return u.i; /* read as int    */
}
```

Neste código copia-se um `float` para um `int`, sem efectuar a conversão de `float` para `int`. Sem usar uma união nem instruções em *assembly*, a única outra maneira de fazer o mesmo em `C` é guardar o `float` numa posição de memória (`s.s` em *assembly*) e depois ler o que lá está como se fosse um `int` (`lw` em *assembly*). Neste caso particular, uma versão não optimizada do código *assembly* desta função será

```
        .data
u:      .word   0
        .text
        .globl  f_as_i
f_as_i: la      $t0,u       # $t0 = &u
        s.s     $f12,0$(t0) # u.f = f
        lw      $v0,0($t0)  # $v0 = u.i
        jr      $ra
```

e uma versão optimizada, usando todas as instruções que temos ao nosso dispôr no `MIPS`, será:

```
        .text
        .globl  f_as_i
f_as_i: mfc1    $v0,$f12
        jr      $ra
```

(note que a instrução `mfc1` copia apenas os bits, sem efectuar qualquer conversão, que é exactamente o que se pretende).

G. Miscelânea

Em C código do género

```
((exp) ? (val_true) : (val_false))
```

permite que o resultado de uma expressão dependa de um teste condicional: `val_true` será o valor da expressão quando `exp` é diferente de zero e `val_false` será o valor da expressão quando `exp` é igual a zero. Na grande maioria dos casos os parêntesis podem ser omitidos.

Em C também é possível concatenar *statements* usando uma vírgula. Por exemplo

```
for(i = 0,j = 1;i < 10;i++,j <<= 2)
  f(i,j);
```

é equivalente a

```
i = 0; j = 1;
while(i < 10)
{
  f(i,j);
  i++; j <<= 2;
}
```

H. Exemplo final

Converta a seguinte função para *assembly* do MIPS:

```
int f(int x)
{
  int c,i,j,k;
  c = 0;

  for(i = 0;i < x;i++)
  {
    if(c < 100 || c > 1000)
    {
      for(j = 0;j <= i;j++)
      {
        c += f(i & j);
      }
    }
    else
    {
      for(j = i;j < x;j <<= 2)
      {
        if((c & 1) == 0 && (i & 2) != 0)
        {
          for(k = i;k <= j;k++)
          {
            c += j | k;
          }
        }
      }
    }
  }
  return c;
}
```

Para poupar espaço mais tarde, vamos reescrever esta função na forma mais compacta:

```
int f(int x)
{
  int c,i,j,k;
  c = 0;
  for(i = 0;i < x;i++)
    if(c < 100 || c > 1000)
      for(j = 0;j <= i;j++)
        c += f(i & j);
    else
      for(j = i;j < x;j <<= 2)
        if((c & 1) == 0 && (i & 2) != 0)
          for(k = i;k <= j;k++)
            c += j | k;
  return c;
}
```

A primeira coisa a fazer é decidir onde é que se vão guardar as variáveis locais (e argumentos, caso seja necessário) usadas pela função. Numa arquitectura *load/store* com muitos registos disponíveis, como é o caso do MIPS, é geralmente possível utilizar apenas registos para guardar todas as variáveis locais. No nosso caso a função chama-se a si própria numa zona onde é necessário preservar as variáveis `i`, `x` (argumento), `c` e `j`. Logo estas variáveis têm de ser guardadas em registos do tipo `s`. Isso já não acontece com a variável `k`, que só é usada no `else` do primeiro `if`, pelo que esta pode ser guardada num registo do tipo `t`. A tabela seguinte mostra a escolha que fizemos.

variável	registo
x	$s0
c	$s1
i	$s2
j	$s3
k	$t0

Tal como uma cebola a nossa função tem várias camadas. Podemos começar por codificar a camada exterior, **deixando** o miolo da função **para mais tarde** (ou vice-versa). Em C temos

```
int f(int x)
{
  int c,i,j,k;

  BLOCO_1
  return c;
}
```

onde `BLOCO_1` é

```
  c = 0;
  for(i = 0;i < x;i++)
    if(c < 100 || c > 1000)
      for(j = 0;j <= i;j++)
        c += f(i & j);
    else
      for(j = i;j < x;j <<= 2)
        if((c & 1) == 0 && (i & 2) != 0)
          for(k = i;k <= j;k++)
            c += j | k;
```

Esta maneira de pensar reduz substancialmente os detalhes que é preciso tem em atenção em cada fase da codificação. No nosso caso é preciso salvaguardar no início da função (prólogo) os registos do tipo `s` que vão ser utilizados e o registo `$ra`, e no fim (epílogo) é preciso voltar a repor os seus valores originais antes de sair da função. Uma codificação possível da função `f()` em *assembly* do MIPS será então

```
        .text
        .globl  f
f:      addu    $sp,$sp,-20
        sw      $ra,0($sp)
        sw      $s0,4($sp)
        sw      $s1,8($sp)
        sw      $s2,12($sp)
        sw      $s3,16($sp)
        or      $s0,$zero,$a0
        BLOCO_1
        or      $v0,$zero,$s1
        lw      $ra,0($sp)
        lw      $s0,4($sp)
        lw      $s1,8($sp)
        lw      $s2,12($sp)
        lw      $s3,16($sp)
        addu    $sp,$sp,20
        jr      $ra
```

Note que o valor do argumento `x` é transferido para o registo `$s0` antes do `BLOCO_1` e que o valor da variável `c` (registo `$s1`) é transferido para o registo de saída (`$v0`) depois do `BLOCO_1`.

Continuando a descascar a função observa-se que no nosso caso o `BLOCO_1` tem a seguinte forma:

```
c = 0;
for(i = 0;i < x;i++)
{
  BLOCO_2
}
```

onde `BLOCO_2` é

```
if(c < 100 || c > 1000)
  for(j = 0;j <= i;j++)
    c += f(i & j);
else
  for(j = i;j < x;j <<= 2)
    if((c & 1) == 0 && (i & 2) != 0)
      for(k = i;k <= j;k++)
        c += j | k;
```

Uma codificação possível para o `BLOCO_1` será

```
        ori     $s1,$zero,0
for1:   ori     $s2,$zero,0
loop1:  bge     $s2,$s0,end1
body1:  BLOCO_2
next1:  addi    $s2,$s2,1
        j       loop1
end1:
```

Note que se utilizaram *labels* (`for1`, `loop1`, `body1`, `next1` e `end1`) para assinalar o princípio de cada um dos pontos para onde podem existir saltos (condicionais ou não) no ciclo `for`. É possível que algumas destas *labels* não venham a ser utilizadas. Por exemplo, a *label* `body1` **neste caso** não será utilizada (se a condição de permanência do ciclo fosse, for exemplo, `i < x || c < 0` então esta *label* seria utilizada [**confirme**]). A sua utilização permite delimitar bem as zonas de código correspondentes a cada parte da função, o que torna o código mais legível e torna mais fácil verificar a sua correção.

O bloco `BLOCO_2` tem a seguinte forma:

```
if(c < 100 || c > 1000)
{
  BLOCO_3
}
else
{
  BLOCO_4
}
```

onde `BLOCO_3` é

```
for(j = 0;j <= i;j++)
  c += f(i & j);
```

e onde `BLOCO_4` é

```
for(j = i;j < x;j <<= 2)
  if((c & 1) == 0 && (i & 2) != 0)
    for(k = i;k <= j;k++)
      c += j | k;
```

Uma codificação possível para o `BLOCO_2` será

```
if2:    blt     $s1,100,then2
        ble     $s1,1000,else2
then2:  BLOCO_3
        j       end2
else2:  BLOCO_4
end2:
```

Uma codificação possível para o `BLOCO_3` será (seria possível subdividir o `BLOCO_3` na parte relacionada com o controlo do ciclo `for` e na parte que está no seu interior, mas como neste caso esta última é muito simples isso não foi feito)

```
for3:   ori     $s3,$zero,0
loop3:  bgt     $s3,$s2,end3
body3:  and     $a0,$s2,$s3
        jal     f
        add     $s1,$s1,$v0
next3:  addi    $s3,$s3,1
        j       loop3
end3:
```

O bloco `BLOCO_4` tem a seguinte forma:

```
for(j = i;j < x;j <<= 2)
{
  BLOCO_5
}
```

onde `BLOCO_5` é

```
if((c & 1) == 0 && (i & 2) != 0)
  for(k = i;k <= j;k++)
    c += j | k;
```

Uma codificação possível para o `BLOCO_4` será

```
for4:   or      $s3,$zero,$s2
loop4:  bge     $s3,$s0,end4
body4:  BLOCO_5
next4:  sll     $s3,$s3,2
        j       loop4
end4:
```

O bloco `BLOCO_5` tem a seguinte forma:

```
if((c & 1) == 0 && (i & 2) != 0)
{
  BLOCO_6
}
```

onde BLOCO_6 é

```
for(k = i;k <= j;k++)
  c += j | k;
```

Uma codificação possível para o BLOCO_5 será

```
if5:    andi    $t1,$s1,1
        bnez    $t1,end5
        andi    $t1,$s2,2
        beqz    $t1,end5
then5:  BLOCO_6
end5:
```

e uma codificação possível para o BLOCO_6 será (tal como para o BLOCO_3, o interior do ciclo `for` do BLOCO_6 é muito simples, pelo que pode ser codificado na sua totalidade sem dificuldade)

```
for6:   move    $t0,$s2
loop6:  bgt     $t0,$s3,end6
body6:  or      $t1,$s3,$t0
        add     $s1,$s1,$t1
next6:  addi    $t1,$t1,1
        j       loop6
end6:
```

Para terminar basta incorporar o código de cada bloco no código do bloco imediatamente a montante (nível anterior), obtendo-se o código apresentado na coluna do lado direito. Pode-se verificar que algumas *labels* não são utilizadas (podendo portanto ser eliminadas), e que existem *labels* seguidas sem código pelo meio (podendo portanto ser aglomeradas numa única *label*). Não o fizemos para tornar o código mais fácil de perceber.

Com prática a subdivisão que aqui foi detalhada poderá ser feita ou mentalmente ou numa folha de papel, como ilustrado em baixo. Para funções pequenas pode-se mesmo converter a função em *assembly* pela ordem em que a função foi escrita (registando mentalmente o que falta ainda fazer das estruturas de controlo de fluxo de execução). Note que a conversão em separado de cada bloco elementar torna um problema que à partida parecia difícil num problema fácil de resolver (dividir para conquistar).

```
int f(int x)
{ // x->$s0, c->$s1, i->$s2, j->$s3, k->$t0
  int c,i,j,k;

  c = 0;                                          1
  for(i = 0;i < x;i++)
    if(c < 100 || c > 1000)                       2
      for(j = 0;j <= i;j++)                       3
        c += f(i & j);
    else
      for(j = i;j < x;j <<= 2)                    4
        if((c & 1) == 0 && (i & 2) != 0)          5
          for(k = i;k <= j;k++)                   6
            c += j | k;
  return c;                                       0
}
```

```
        .text
        .globl  f
f:      subu    $sp,$sp,20                        0
        sw      $ra,0($sp)
        sw      $s0,4($sp)
        sw      $s1,8($sp)
        sw      $s2,12($sp)
        sw      $s3,16($sp)
        or      $s0,$zero,$a0
        ori     $s1,$zero,0                       1
for1:   ori     $s2,$zero,0
loop1:  bge     $s2,$s0,end1
body1:
if2:    blt     $s1,100,then2                     2
        ble     $s1,1000,else2
then2:
for3:   ori     $s3,$zero,0                       3
loop3:  bgt     $s3,$s2,end3
body3:  and     $a0,$s2,$s3
        jal     f
        add     $s1,$s1,$v0
next3:  addi    $s3,$s3,1
        j       loop3
end3:
        j       end2
else2:
for4:   or      $s3,$zero,$s2                     4
loop4:  bge     $s3,$s0,end4
body4:
if5:    andi    $t1,$s1,1                         5
        bnez    $t1,end5
        andi    $t1,$s2,2
        beqz    $t1,end5
then5:
for6:   or      $t0,$zero,$s2                     6
loop6:  bgt     $t0,$s3,end6
body6:  or      $t1,$s3,$t0
        add     $s1,$s1,$t1
next6:  addi    $t1,$t1,1
        j       loop6
end6:
end5:
next4:  sll     $s3,$s3,2
        j       loop4
end4:
end2:
next1:  addi    $s2,$s1,1
        j       loop1
end1:
        or      $v0,$zero,$s1
        lw      $ra,0($sp)
        lw      $s0,4($sp)
        lw      $s1,8($sp)
        lw      $s2,12($sp)
        lw      $s3,16($sp)
        addu    $sp,$sp,20
        jr      $ra
```

BIBLIOGRAFIA

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, fourth edition, 2009.

[2] Peter Pintz and Tony Crawford. *C in a Nutshell: A Desktop Quick Reference*. O'Reilly, 2006.