Arquitetura de Computadores II

Manipulação de bits

José Luís Azevedo, Manuel Bernardo Cunha, Tomás Oliveira e Silva

Índice

1 Introdução

Um bit de informação pode representar o valor de uma variável Booleana:

- 0 (zero) representa falso,
- 1 (um) representa verdadeiro.

Uma função Booleana de n variáveis, $n \geqslant 1$, pode ser representada por uma tabela de verdade. A tabela de verdade enumera todas as combinações possíveis de valores dessas variáveis e, para cada uma delas, especifica o valor da função Booleana. São 2^n casos ao todo. Neste documento descrevemos, na secção 2 apenas quatro funções Booleanas:

- **not** — negação ($n = 1$)
- **and** — e lógico ($n \geqslant 2$), também designado por produto lógico, ou conjunção,
- **or** — ou lógico ($n \geqslant 2$), também designado por soma lógica, ou disjunção,
- **xor** — ou exclusivo lógico ($n \geqslant 2$), também designado por função paridade, ou, para $n = 2$, por função diferença.

Todos os processadores modernos têm instruções que efetuam estas operações lógicas (para $n = 2$ no caso do **and**, **or** e **xor**) em paralelo para cada um dos bits dos seus registos. Na secção 3 isto é descrito com mais detalhe para o caso do MIPS e na secção 4 são apresentadas algumas aplicações da utilização dessas instruções (incluindo código equivalente na linguagem de programação C).

2 Algumas funções Booleanas

2.1 A negação (not)

A tabela de verdade da função lógica **negação** (abreviado em Inglês por **not**), que tem apenas um argumento, é muito simples:

x	not x
0	1
1	0

Por vezes, tahbém se representa a negação com uma barra: **not** $x = \overline{x}$. Pressupondo que o bit que representa cada valor lógico é interpretado como um 0 ou 1 numérico, então também temos (complemento para 1)

$$\textbf{not}\, x = 1 - x.$$

2.2 O e lógico (and)

A operação lógica **e** (abreviado em Inglês por **and**) apenas dá verdadeiro quando todos os seus argumentos são verdadeiros. Sendo assim, temos

$$x_1 \textbf{ and } x_2 \textbf{ and } \ldots \textbf{ and } x_n = \begin{cases} 0, & \text{quando pelo menos um dos } x_i \text{ é } 0; \\ 1, & \text{quando todos os } x_i \text{ são } 1. \end{cases}$$

Em particular, temos

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

Pressupondo que o bit que representa cada valor lógico é interpretado como um 0 ou 1 numérico, então temos

$$x_1 \textbf{ and } x_2 \textbf{ and } \ldots \textbf{ and } x_n = \textbf{min}(x_1, \ldots, x_n).$$

Também temos (o produto aritmético tem o mesmo elementro neutro e o mesmo elemento absorvente que o **and** lógico)

$$x_1 \textbf{ and } x_2 \textbf{ and } \ldots \textbf{ and } x_n = x_1 \cdot x_2 \cdot \ldots \cdot x_n = \prod_{i=1}^{n} x_i.$$

É por causa desta segunda maneira de interpretar esta operação lógica que ela é usualmente representada matemáticamente pelo sinal de produto.

Para $n = 2$, a operação lógica **and** tem as seguintes propriedades:

- comutatividade: x **and** $y = y$ **and** x,
- idempotência: x **and** $x = x$,
- o zero é o elemento absorvente: x **and** $0 = 0$ (muito útil para forçar um bit a zero),
- o um é o elemento neutro: x **and** $1 = x$ (muito útil para deixar passar um bit sem alterar o seu valor),
- também se verifica que x **and** $($**not** $x) = 0$.

2.3 O ou lógico (or)

A operação lógica **ou** (abreviado em Inglês por **or**) apenas dá falso quando todos os seus argumentos são falsos. Sendo assim, temos

$$x_1 \textbf{ or } x_2 \textbf{ or } \ldots \textbf{ or } x_n = \begin{cases} 0, & \text{quando todos os } x_i \text{ são } 0; \\ 1, & \text{quando pelo menos um dos } x_i \text{ é } 1. \end{cases}$$

Em particular, temos

x	y	x **or** y
0	0	0
0	1	1
1	0	1
1	1	1

Pressupondo que o bit que representa cada valor lógico é interpretado como um 0 ou 1 numérico, então temos

$$x_1 \textbf{ or } x_2 \textbf{ or } \ldots \textbf{ or } x_n = \textbf{max}(x_1, \ldots, x_n).$$

Também temos

$$x_1 \textbf{ or } x_2 \textbf{ or } \ldots \textbf{ or } x_n = \begin{cases} 0, & \text{se } \sum_{i=1}^{n} x_i = 0, \\ 1, & \text{se } \sum_{i=1}^{n} x_i \geqslant 1. \end{cases}$$

(Trata-se de uma soma com saturação a 1: valores da soma maiores do que 1 são convertidos em 1.) É por causa desta segunda maneira de interpretar esta operação lógica que ela é usualmente representada matemáticamente pelo sinal de soma.

Para $n = 2$, a operação lógica **or** tem as seguintes propriedades:

- comutatividade: x **or** $y = y$ **or** x,
- idempotência: x **or** $x = x$,
- o zero é o elemento neutro: x **or** $0 = x$ (muito útil para deixar passar um bit sem alterar o seu valor),
- o um é o elemento absorvente: x **or** $1 = 1$ (muito útil para forçar um bit a um),
- também se verifica que x **or** (**not** x) $= 1$.

2.4 O ou exclusivo lógico (xor)

A operação lógica **ou exclusivo** (abreviado em Inglês por **xor**) dá verdadeiro quando o número de argumentos verdadeiros é ímpar. Sendo assim, temos

$$x_1 \textbf{ xor } x_2 \textbf{ xor } \ldots \textbf{ xor } x_n = \begin{cases} 0, & \text{quando o número de } x_i \text{ iguais a 1 é par;} \\ 1, & \text{quando o número de } x_i \text{ iguais a 1 é ímpar.} \end{cases}$$

Em particular, temos

x	y	x **xor** y
0	0	0
0	1	1
1	0	1
1	1	0

Pressupondo que o bit que representa cada valor lógico é interpretado como um 0 ou 1 numérico, então também temos

$$x_1 \,\textbf{xor}\, x_2 \,\textbf{xor} \ldots \textbf{xor}\, x_n = \left(\sum_{i=1}^{n} x_i\right) \textbf{mod}\ 2,$$

onde a notação **mod** 2 representa o resto da divisão por 2 (módulo 2). A soma de números inteiros módulo 2 é por vezes representada por $\oplus$ e é por isso que esta operação lógica é usualmente representada por este símbolo.

Para $n = 2$, a operação lógica **xor** tem as seguintes propriedades:

- comutatividade: $x \,\textbf{xor}\, y = y \,\textbf{xor}\, x$,
- aniquilação: $x \,\textbf{xor}\, x = 0$,
- zero é o elemento neutro: $x \,\textbf{xor}\, 0 = x$ (muito útil para deixar passar um bit sem alterar o seu valor),
- um é o elemento inversor: $x \,\textbf{xor}\, 1 = \textbf{not}\, x = \overline{x}$ (muito útil para inverter [*toggle*] o valor de um bit),
- também se verifica que $x \,\textbf{xor}\, (\textbf{not}\, x) = 1$.

3 Algumas instruções do MIPS que usam funções Booleanas

Todos os processadores modernos têm instruções que aplicam as quatro operações lógicas descritas na secção 2 em paralelo aos bits de um ou dois registos (*bitwise operators*). Em particular, no caso do MIPS, temos a (pseudo) instrução

```
        not    destination_reg,source_reg
```

que faz o seguinte ($d_i = \textbf{not}\ s_i$):

source_register	s_{31}	s_{30}	s_{29}	s_{28}	$\cdots$	s_3	s_2	s_1	s_0
	not	not	not	not		not	not	not	not
destination_register	d_{31}	d_{30}	d_{29}	d_{28}	$\cdots$	d_3	d_2	d_1	d_0

E, por exemplo, a instrução

```
        and    destination_reg,source1_reg,source2_reg
```

faz o seguinte ($d_i = s_{1,i}\ \textbf{and}\ s_{2,i}$):

source1_register	$s_{1,31}$	$s_{1,30}$	$s_{1,29}$	$s_{1,28}$	$\cdots$	$s_{1,3}$	$s_{1,2}$	$s_{1,1}$	$s_{1,0}$
source2_register	$s_{2,31}$	$s_{2,30}$	$s_{2,29}$	$s_{2,28}$	$\cdots$	$s_{2,3}$	$s_{2,2}$	$s_{2,1}$	$s_{2,0}$
	and	and	and	and		and	and	and	and
destination_register	d_{31}	d_{30}	d_{29}	d_{28}	$\cdots$	d_3	d_2	d_1	d_0

O comportamento das instruções **or** e **xor** é semelhante.

4 Aplicações

As operações lógicas descritas acima são extremamente úteis para alterar alguns bits de um registo sem modificar os outros bits. Apresentamos a seguir alguns exemplos, quer em assembly do MIPS quer na linguagem de programação C. Nesta última, os operadores lógicos *bitwise* disponíveis são os seguintes:

o que se pretende	como se faz em C
$b =$ **not** a	`b = ~a;`
$c = a$ **and** b	`c = a & b;`
$c = a$ **or** b	`c = a | b;`
$c = a$ **xor** b	`c = a ^ b;`

Tal como descrito na secção 3, estes operadores lógicos aplicam-se, em paralelo, a cada um dos bits das variáveis a e b.

Em todos os exemplos apresentados a seguir vamos apenas trabalhar com os 16 bits menos significativos dos números inteiros; os outros poderão ficar com valores arbitrários (não nos interessam). Existem duas razões principais para trabalharmos apenas com 16 bits:

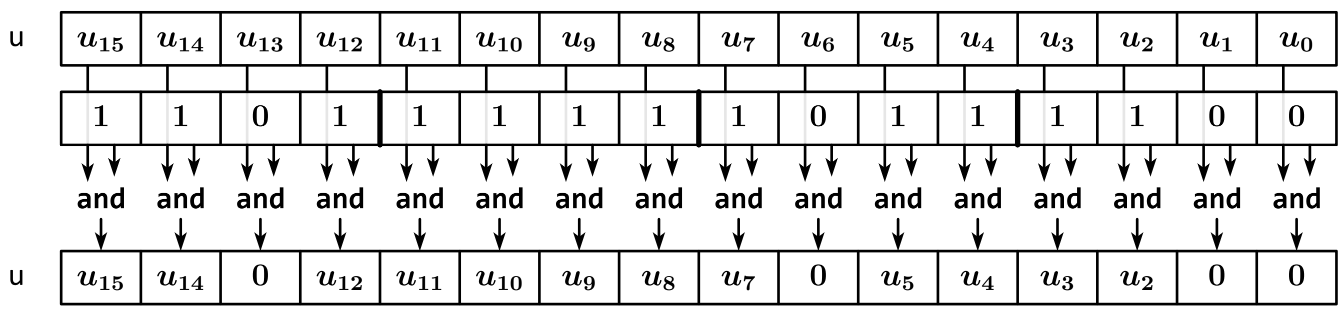
- é o que é preciso para o PIC32, e
- é conveniente porque as instruções lógicas do tipo I do MIPS têm constantes de 16 bits.

É trivial generalizar os exemplos para um número diferente de bits.

É altamente desaconselhado que as técnicas descritas a seguir sejam decoradas. Tal não é necessário e é mesmo contraproducente. É apenas suficiente conhecer quais são os elementos neutros e absorventes das funções Booleanas **and** e **or**, conhecer também as propriedades do **xor**, e conhecer bem o comportamento dos deslocamentos lógicos para a esquerda e para a direita (os bits que entram são todos zero) e do deslocamento aritmético para a direita (os bits que entram são iguais ao bit do sinal, que é o bit mais significativo). Depois é só usar estes conhecimentos para deduzir, na hora, o que é necessário fazer para se obter o resultado pretendido. É isso que os autores deste documento fazem, e até agora nunca se deram mal com isso.

4.1 Como forçar um conjunto predefinido de bits de um registo a zero sem alterar os outros bits

Como se pretende forçar bits a zero, devemos usar a operação lógica que tem o zero como elemento absorvente e o um como elemento neutro. É pois o **and**. No exemplo seguinte, pretende-se colocar os bits 13, 6, 1 e 0 a zero, sem alterar os outros bits. Isto pode ser feito da seguinte maneira:



Em C, isto pode ser feito da seguinte maneira (note que **1101 1111 1011 1100**$_2$ é, em hexadecimal, **DFBC**$_{16}$):

```
u &= 0xDFBC;
```

Em assembly do MIPS, supondo que a variável u está no registo `$t0`, temos

```
        andi    $t0,$t0,0xDFBC
```

4.2 Como forçar um conjunto predefinido de bits de um registo a um sem alterar os outros bits

Como se pretende forçar bits a um, devemos usar a operação lógica que tem o um como elemento absorvente e o zero como elemento neutro. É pois o **or**. No exemplo seguinte, pretende-se colocar os bits 11, 7, 4 e 3 a um, sem alterar os outros bits. Isto pode ser feito da seguinte maneira:

u	u_{15}	u_{14}	u_{13}	u_{12}	u_{11}	u_{10}	u_9	u_8	u_7	u_6	u_5	u_4	u_3	u_2	u_1	u_0
	0	0	0	0	1	0	0	0	1	0	0	1	1	0	0	0
	or	or	or	or	or	or	or	or	or	or	or	or	or	or	or	or
u	u_{15}	u_{14}	u_{13}	u_{12}	1	u_{10}	u_9	u_8	1	u_6	u_5	1	1	u_2	u_1	u_0

Em C, isto pode ser feito da seguinte maneira (note que $\mathbf{0000\ 1000\ 1001\ 1000_2}$ é, em hexadecimal, $\mathbf{0898_{16}}$):

```
u &= 0x0898;
```

Em assembly do MIPS, supondo que a variável u está no registo `$t0`, temos

```
        ori     $t0,$t0,0x0898
```

4.3 Como forçar alguns bits a zero e outros a um

É uma combinação dos dois métodos anteriores! Por exemplo, para colocar os bits 14, 9 e 4 a zero e os bits 15, 6, 2 e 0 a um, sem alterar os outros bits, faz-se:

u	u_{15}	u_{14}	u_{13}	u_{12}	u_{11}	u_{10}	u_9	u_8	u_7	u_6	u_5	u_4	u_3	u_2	u_1	u_0
	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1
	and	and	and	and	and	and	and	and	and	and	and	and	and	and	and	and
	u_{15}	0	u_{13}	u_{12}	u_{11}	u_{10}	0	u_8	u_7	u_6	u_5	0	u_3	u_2	u_1	u_0
	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
	or	or	or	or	or	or	or	or	or	or	or	or	or	or	or	or
u	1	0	u_{13}	u_{12}	u_{11}	u_{10}	0	u_8	u_7	1	u_5	0	u_3	1	u_1	1

Em C, isto pode ser feito da seguinte maneira:

```
u &= 0xBDEF;
u |= 0x8045;
```

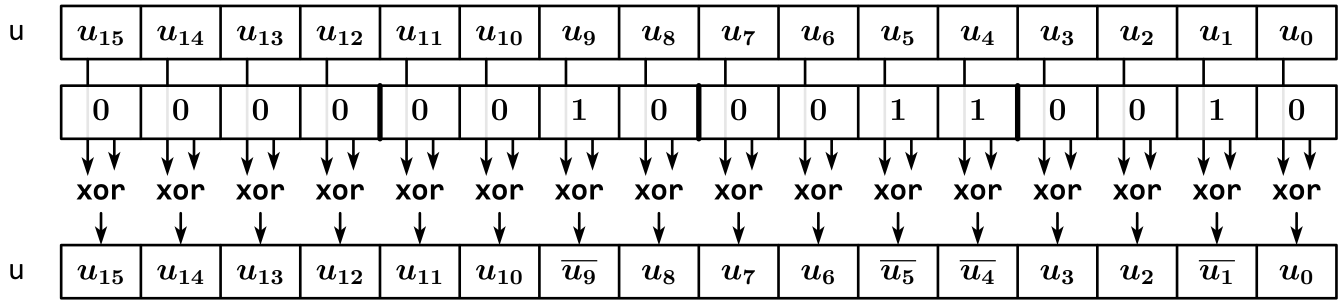
Em assembly do MIPS, supondo que a variável u está no registo `$t0`, temos

```
        andi    $t0,$t0,0xBDEF
        ori     $t0,$t0,0x8045
```

Note que a ordem do **and** e do **or** é arbitrária; o **or** podia ter sido feito primeiro.

4.4 Como inverter o valor (*toggle*) de um conjunto predefinido de bits de um registo sem alterar os outros bits

Como se pretende inverter alguns bits, devemos usar a operação lógica que tem um elemento neutro e um elemento inversor. É pois o **xor**, sendo o zero o elemento neutro e o um o inversor. No exemplo seguinte, pretende-se inverter os bits 9, 5, 4 e 1, sem alterar os outros bits. Isto pode ser feito da seguinte maneira:



Em C, temos

```
u ^= 0x0232;
```

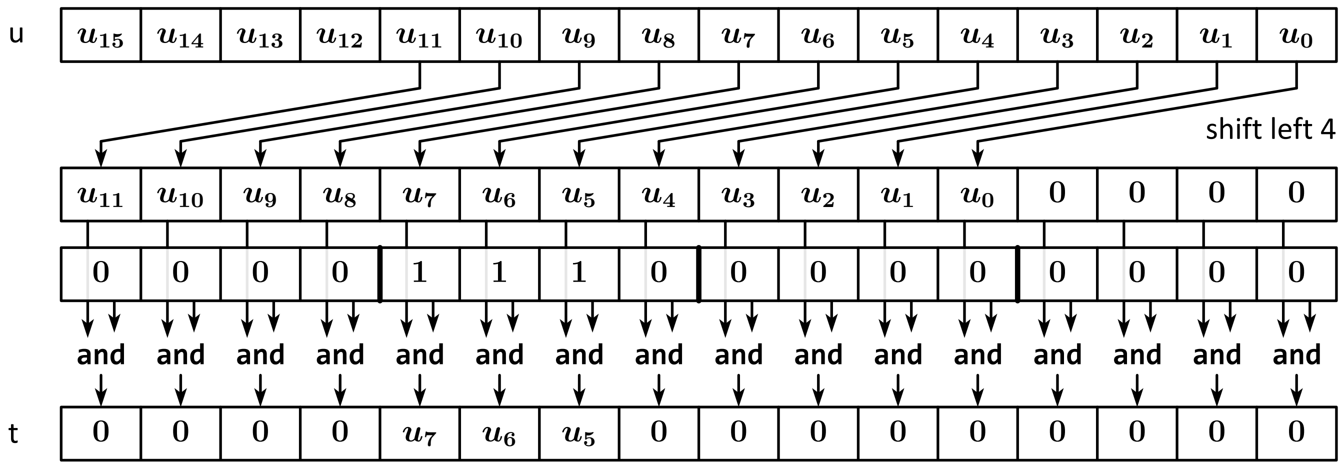
Em assembly do MIPS, supondo que a variável u está no registo `$t0`, temos

```
        xori    $t0,$t0,0x0232
```

4.5 Como copiar alguns bits consecutivos de um registo para outro registo

Como exemplo final, suponha que se pretende copiar os bits 7 a 5 da variável u para os bits 11 a 9 da variável `v`, sem alterar os restantes bits de `v`. Podemos fazer isso com um **or** de dois **and**, ou com um **and** de dois **or** (em ambos os casos com um deslocamento para a esquerda no início). Um maneira possível de fazer isso, usando uma variável temporária `t`, é a seguinte.

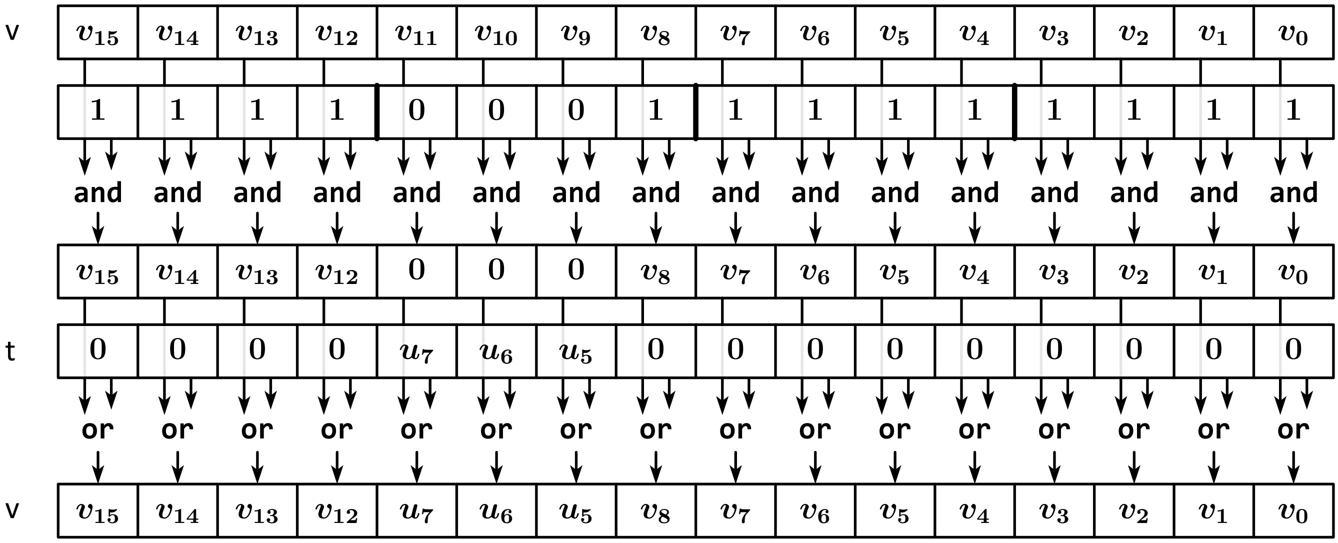
Passo 1:



Note que podiamos ter efetuado o deslocamento para a esquerda depois de aplicar o **and**. Note ainda que o deslocamento é de $11 - 7 = 4$.

A variável `t` já tem os bits 7 a 5 da variável u na posição correta (bits 11 a 9), prontos a serem colocados na variável `v`. Para isso, basta colocar os bits 11 a 9 de v a zero usando um **and** e efetuar depois um **or** com `t`! Isto funciona porque o zero é o elemento neutro do **or**.

Passo 2:



Em C, temos

```
t = (u << 4) & 0x0E00;
v = (v & 0xF1FF) | t;
```

(Seria simples fazer a coisa sem a variável `t`.) Em assembly do MIPS, supondo que as variáveies u, v e `t` estão, respetivamente, nos registos `$t0`, `$t1` e `$t2`, temos

```
        sll     $t2,$t0,4
        andi    $t2,$t2,0x0E00
        andi    $t1,$t1,0xF1FF
        or      $t1,$t2,$t2
```

Exercícios extra:

- Resolva um problema semelhante, mas no qual se pretende que os bits 11 a 6 da variável u sejam copiados para os bits 6 a 1 da variável v, sem alterar os restantes bits de v.
- Resolva o problema original com um deslocamento para a esquerda, dois **or** e um único **and**. [Pista: qual é o elemento neutro do **and**?]
- Se se fizer primeiro o deslocamento, e se este for para a direita, existe alguma diferença entre usar um deslocamento lógico ou usar um aritmético?