AULA PRÁTICA N.º 2

# Objetivos:

Utilização de instruções lógicas e de deslocamento sobre inteiros no MIPS. Utilização das diretivas do *assembler* do MARS.

# Conceitos básicos:

* Lógica *bitwise* e operações com máscaras. Instruções lógicas.
* Deslocamento (*shift*) lógico e aritmético. Instruções de deslocamento.
* Diretivas do *assembler* do MARS.

# Guião:

1. Instruções lógicas.
   1. Codifique um programa em *assembly* do MIPS que determine o resultado das operações lógicas bit a bit (*bitwise*) AND1, OR, NOR e XOR, considerando como operandos os valores armazenados nos registos **$t0** e **$t1**; os resultados devem ser armazenados nos registos **$t2**, **$t3**, **$t4** e **$t5**, respetivamente.

**.data**

**.text**

**.globl main**

**main: ori $t0,$0,val\_1 # substituir val\_1 e val\_2 pelos ori $t1,$0,val\_2 # valores de entrada desejados and $t2,$t0,$t1 # $t2 = $t0 & $t1 (and bit a bit) or ... # $t3 = $t0 | $t1 (or bit a bit) (...) #**

**jr $ra # fim do programa**

* 1. Execute o programa no MARS, preencha a tabela e confirme (calculando manualmente) os resultados para os seguintes pares de valores:

**val\_1 = 0x1234, val\_2 = 0x000F val\_1 = 0x1234, val\_2 = 0xF0F0 val\_1 = 0x5C1B, val\_2 = 0xA3E4**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **$t0** | **$t1** | **$t2 (AND)** | **$t3 (OR)** | **$t4 (NOR)** | **$t5 (XOR)** |
| **0x1234** | **0x000F** | 0x00000004 | 0x0000123f | 0xffffedc0 | 0x0000123b |
| **0x1234** | **0xF0F0** | 0x00001030 | 0x0000f2f4 | 0xffff0d0b | 0x0000e2c4 |
| **0x5C1B** | **0xA3E4** | 0x00000000 | 0x0000ffff | 0xffff0000 | 0x0000ffff |

* 1. O ISA do MIPS não disponibiliza uma instrução de negação bit a bit. Usando as instruções lógicas disponíveis, sugira uma forma de efetuar a negação bit a bit do conteúdo de um registo e implemente-a (registo de entrada **$t0**, registo de saída **$t1**). Teste o seu programa com os seguintes valores e confirme manualmente os resultados obtidos:

|  |  |
| --- | --- |
| **$t0 (Val)** | **$t1 (Val\)** |
| **0x0F1E** | 0xfffff0e1 |
| **0x0614** | 0xfffff9eb |
| **0xE543** | 0xffff1abc |

1 Em linguagem C, os operadores lógicos *bitwise* representam-se por: **&** (**and**); **| (or)**; **^** (**xor**); **~** (**not**)

1. Instruções de deslocamento.
   1. Para além das instruções que implementam operações lógicas bit a bit, o MIPS disponibiliza ainda operações de deslocamento2 (*shift*), nomeadamente, deslocamento à esquerda lógico, deslocamento à direita lógico e deslocamento à direita aritmético. Em todas estas instruções o número de bits a deslocar é especificado na instrução (campo **Imm**):

**sll Rdst,Rsrc,Imm # Shift left logical srl Rdst,Rsrc,Imm # Shift right logical sra Rdst,Rsrc,Imm # Shift right arithmetic**

Escreva um programa que efetue as 3 operações de deslocamento, considerando como operandos os registos **$t0** e a constante **Imm** (valor e número de bits a deslocar, respetivamente) e colocando os resultados nos registos **$t2**, **$t3** e **$t4**. Execute o programa, e observe os resultados, para os seguintes pares de valores (a instrução virtual "**li**" permite a inicialização de um registo com uma constante de 32 bits)3:

**(0x12345678, 1)**

**(0x12345678, 4)**

**(0x12345678, 16)**

**(0x862A5C1B, 2)**

**(0x862A5C1B, 4)**

**.data**

**.text**

**.globl main**

**main: li $t0,0x12345678 # instrução virtual (decomposta**

**# em duas instruções nativas)**

**sll $t2,$t0,1 #**

**srl $t3,$t0,1 #**

**sra $t4,$t0,1 #**

**jr $ra # fim do programa**

* 1. Preencha a tabela seguinte e confirme manualmente os resultados para cada um dos pares de valores de entrada.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **$t0** | **Imm** | **$t2 (sll)** | **$t3 (srl)** | **$t4 (sra)** |
| **0x12345678** | **1** | 0x2468acf0 | 0x091a2b3c | 0x091a2b3c |
| **0x12345678** | **4** | 0x23456780 | 0x01234567 | 0x01234567 |
| **0x12345678** | **16** | 0x56780000 | 0x00001234 | 0x00001234 |
| **0x862A5C1B** | **2** | 0x18a9706c | 0x218a9706 | 0xe18a9706 |
| **0x862A5C1B** | **4** | 0x62a5c1b0 | 0x0862a5c1 | 0xf862a5c1 |

* 1. Observe, no MARS, a decomposição da instrução virtual "**li $t0,0x862A5C1B**" (coluna "Native Instruction"); anote os endereços de memória onde as instruções resultantes da decomposição estão armazenadas.
  2. A conversão de uma quantidade codificada em binário natural para o equivalente em código Gray poder ser feita do seguinte modo:

**gray = bin ^ (bin >> 1);**

Traduza para *assembly* a expressão anterior, usando os registos **$t0** e **$t1** para o armazenamento das variáveis "**bin**" e "**gray**", respetivamente. Teste o seu programa para diferentes valores de entrada (por exemplo 2, 4, 5, …).

2 Em linguagem C o deslocamento à direita representa-se por **>>** e o deslocamento à esquerda por **<<**

3 O MIPS não disponibiliza, por razões que serão compreendidas mais tarde, uma única instrução que permita o preenchimento de uma quantidade de 32 bits num registo do CPU).

* 1. A conversão de uma quantidade codificada em código Gray para binário natural (a operação inversa da descrita na alínea anterior) pode ser feita, de forma não iterativa e para quantidades de 8 bits, do seguinte modo:

**num = gray;**

**num = num ^ (num >> 4); num = num ^ (num >> 2); num = num ^ (num >> 1); bin = num;**

Traduza para *assembly* o programa anterior, usando os registos **$t0**, **$t1** e **$t2** para o armazenamento das variáveis "**gray**" "**num**" e "**bin**", respetivamente. Teste o seu programa para diferentes valores de entrada (por exemplo 2, 7, 13, 15, …).

1. Diretivas do *assembler.*

Os programas que efetuam a tradução de código *assembly* para código máquina (designados em inglês por *assemblers*) disponibilizam um conjunto de instruções que permitem ao programador controlar alguns aspetos do processo de tradução. Estas instruções (não confundir com as instruções do CPU) são normalmente designadas por diretivas e são usadas exclusivamente pelo *assembler* durante o processo de tradução do código.

No caso do *assembler* para o MIPS usado no MARS, as diretivas são constituídas por um identificador, cujo primeiro carater é sempre o símbolo "**.**", e, em alguns casos, por um ou mais parâmetros. Exemplos de diretivas: **.text**, para definir o início da zona de código do programa; **.data**, para definir o início da zona de dados do programa.

A diretiva **.eqv** permite atribuir a um identificador literal uma quantidade numérica (por exemplo: **.eqv print\_string,4**). A utilização desta diretiva tem como objetivo melhorar a legibilidade do código *assembly*, ao permitir utilizar o identificador literal em vez de um número (cabe ao *assembler* a tarefa de substituir o identificador pelo valor que lhe foi atribuído na diretiva).

Para além das diretivas anteriores há uma outra que será usada com frequência e que permite a declaração de *strings* (sequências de carateres alfanuméricos delimitadas pelo carater “**"**”). Por exemplo, a declaração da *string* **"AC1 – P"**, pode ser efetuada do seguinte modo:

**.data**

**str1: .asciiz "AC1 – P"**

em que **str1** é um identificador (*label*), que é uma sequência de carateres alfanuméricos, cujo primeiro carater não pode ser um algarismo.

A diretiva **.asciiz** reserva, em memória, espaço para alojar todos os carateres da *string*, e ainda para um carater especial que explicita o fim da *string*, designado por terminador. Em *assembly* e em linguagem C o terminador é o carater **'\0'**, isto é, o *byte* **0x00**. De referir ainda que cada carater é codificado, de acordo com o código ASCII, com 1 *byte*.

* 1. Edite e compile no MARS, o seguinte código:

**.data**

**str1: .asciiz "Uma string qualquer" str2: .asciiz "AC1 – P"**

**.text**

**.globl main main: jr $ra**

* 1. Sabendo que o segmento de dados tem início no endereço **0x10010000** da memória (os endereços no MIPS são quantidades de 32 bits), preencha a tabela seguinte com o código ASCII e o endereço onde está armazenado cada um dos carateres da *string* **str2**. Confirme os códigos dos carateres numa tabela ASCII e verifique a sua localização na memória através da janela de dados do MARS. Preencha a tabela seguinte com todos os endereços de memória ocupados pela *string* **str2** e respetivos valores (não se esqueça que o terminador, '**\0**', faz parte integrante da *string*).

|  |  |  |  |
| --- | --- | --- | --- |
| **Endereço** | **Valor** | **Endereço** | **Valor** |
| **0x100100** | **0x41** |  |  |
|  | **0x43** |  |  |
|  | **0x31** |  |  |
|  | **0x20** |  |  |
|  |  |  |  |

* 1. O identificador da *string* (*label*) permite que o endereço inicial dessa *string* seja referenciado por uma instrução *assembly*. Por exemplo, a utilização da *system call* **print\_string()** requer que, antes da sua chamada, o registo **$a0** do CPU seja preenchido com o endereço inicial da *string* a imprimir. No MIPS, a obtenção do endereço a que corresponde o identificador da *string* pode ser feita através da instrução virtual "**la**", iniciais de *load address*.

O programa para imprimir a string **str2**, usando a *system call* **print\_string()**, fica então:

**.data**

**str1: .asciiz "Uma string qualquer" str2: .asciiz "AC1 – P"**

**.eqv print\_string,4**

**.text**

**.globl main**

**main: la $a0,str2 # instrução virtual, decomposta pelo**

**# assembler em 2 instruções nativas ori $v0,$0,print\_string # $v0 = 4**

**syscall # print\_string(str2);**

**jr $ra # fim do programa**

Edite, compile e execute este código.

* 1. Traduza para *assembly*, e teste no MARS a seguinte sequência de código C:

**print\_string("Introduza 2 numeros ");**

**a = read\_int();**

**b = read\_int();**

**print\_string("A soma dos dois numeros e': "); print\_int10(a + b);**

Tradução incompleta para *assembly*:

**.data**

**str1: .asciiz "Introduza 2 numeros\n"**

**str2: .asciiz "A soma dos dois numeros e': "**

**.eqv print\_string,4**

**.eqv read\_int,??**

**.eqv print\_int10,??**

**.text**

**.globl main main: la $a0,str1**

**ori $v0,$0,print\_string**

**syscall # print\_string(str1); ori $v0,$0,read\_int**

**syscall # valor lido e' retornado em $v0 or $t0,$v0,$0 # $t0=read\_int()**

**(...)**

**jr $ra # fim do programa**

**Anexo**:

|  |  |  |
| --- | --- | --- |
| **u** | **v** | **w = u or v** |
| 0  1 | x  x | x  1 |
| **u** | **V** | **w = u and v** |
| 0  1 | x  x | 0  x |
| **u** | **v** | **w = u xor v** |
| 0  1 | x  x | x  x\ |
| **u** | **v** | **w = u nor v** |
| 0  1 | x  x | x\  0 |

|  |  |  |
| --- | --- | --- |
| **U** | **V** | **W = U or V** |
| 0  F | X  X | X  F |
| **U** | **V** | **W = U and V** |
| 0  F | X  X | 0  X |
| **U** | **V** | **W = U xor V** |
| 0  F | X  X | X  X\ |
| **U** | **V** | **W = U nor V** |
| 0  F | X  X | X\  0 |

Notas:

1. Na tabela da esquerda apresentam-se alguns casos particulares com operandos de 1 bit das operações lógicas mais comuns (o símbolo **\** significa negação).
2. Na tabela da direita apresentam-se alguns casos particulares com operandos de 4 bits (1 dígito hexadecimal) das operações lógicas mais comuns (o símbolo **\** significa negação bit a bit, ou seja, complemento para 1 do operando; **X + X\ = F**).

**PDF criado em 04/09/2024**