

Introdução à Arquitetura e Organização de Computadores

Professor: Diego de Assis Monteiro Fernandes

Aula 8 - Aritmética de Computadores

1 Introdução

Uma das principais tarefas de um computador é a de manipular números. Por isso, o conjunto dos números naturais deve ser modelado pelo computador. Nessa modelagem, cada número natural é representado como um número binário, pois é a base compreendida pelo computador, e armazenado em palavras, que, como já foi visto, é um agrupamento pré-definido de *bits*. Nessa aula, são empregados os conceitos de bases numéricas, algoritmos aritméticos na base binária e conversão entre bases, que estão resumidos na Aula 1.

Considerando que um número deve possuir uma representação através de palavras, é possível levantar algumas questões:

- Como os números negativos são representados?
- Qual o maior número representado em uma palavra?
- O que acontece se uma operação cria um número maior do que a representação máxima?
- Como são representados números reais e fracionários?

Nas seções seguintes são elucidadas algumas destas questões.

2 Números com e sem sinal

As palavras em MIPS tem o tamanho de 32 *bits*. Desse modo é possível representar 2^{32} padrões diferentes, que permite para os números inteiros positivos abranger o intervalo que vai desde 0 até $2^{32} - 1$ ou 4,294,967,295. Como já foi constatado através de algumas das instruções da arquitetura MIPS, o hardware pode ser projetado para somar, subtrair, multiplicar e dividir os números em sua forma binária. É possível que essas operações produzam um número resultante que não pode ser apropriadamente representado em uma palavra. Nesse caso, é registrada a ocorrência de um *overflow*.

Computadores manipulam tanto números positivos como negativos. Os números decimais negativos são modelados pela representação binária em **complemento de dois**. Essa representação em uma palavra de 32 *bits* modela o seguinte intervalo de números: -2,147,483,648 à 2,147,483,647. A representação em complemento de dois apresenta algumas características importantes, além de facilitar as operações aritméticas binárias, que é conduzida de forma semelhante a representação binária direta. Outra característica importante é a de que o *hardware* precisa verificar somente o *bit* mais significativo para descobrir se um número é positivo ou negativo (0 é positivo e 1 é negativo). Além disso, um *overflow* ocorre quando o número (representado em complemento de dois) resultante de uma operação é negativo e tem seu *bit* mais significativo 0 ou é positivo e tem seu *bit* mais significativo 1.

Programas podem manipular números negativos e positivos ou, algumas vezes, somente números positivos. Quando somente números positivos são manipulados, a representação em complemento de 2 não é adotada. Com base nisso, grande parte das operações lógico-aritméticas não são apropriadas para tratar os dois casos. Para diferenciar esses casos, MIPS oferece duas versões de algumas instruções. Por exemplo, as

instruções de comparação **slt** e **slti** manipulam inteiros (int) e as instruções **sltu** e **sltiu** manipulam inteiros positivos. Uma diferença simples de ser constatada está nas instruções MIPS com formato do tipo-I (**addi**, **slt**, **lw**, ...) que tem em seu campo de 16 *bits* uma constante representada em complemento de 2. Ao contrário disso, a instrução **sltu** não utiliza complemento de 2.

3 Adição e Subtração

Os algoritmos de soma e subtração de números binários já são conhecidos. Com base nesses algoritmos, a ocorrência de um *overflow* é detalhada a seguir.

3.1 Overflow

Um *overflow* ocorre quando o resultado de uma operação não pode ser representada pelo *hardware*, no caso do caso da arquitetura MIPS, em uma palavra de 32 *bits*. **Na soma um *overflow* nunca vai ocorrer para os casos em que os operandos possuem sinais diferentes.** Pois, para os operandos x e $-y$, o resultado sempre será no intervalo formado pelos dois operandos. Na subtração, o princípio é o oposto: **quando os sinais dos operandos forem o mesmo, um *overflow* não acontece.**

Um *overflow* ocorre quando a soma de dois números positivos resulta em um número negativo, ou vice-versa. Na subtração, um *overflow* ocorre quando é subtraído um número negativo de um positivo e o resultado obtido é negativo, ou vice-versa.

Em alguns casos é interessante que as operações ignorem a ocorrência de um *overflow* (no endereçamento de memória, por exemplo). Por isso, o projetista de uma máquina deve prover um modo de ignorá-lo em alguns casos e reconhecê-los em outros. A solução em MIPS é ter dois tipos de instruções aritméticas para reconhecer a duas escolhas:

- **add**, **addi** e **sub** causa exceções na ocorrência de um *overflow*.
- **addu**, **addiu** e **subu** ignora a possível ocorrência de um *overflows*.

4 Operações Lógicas

Embora os computadores concentrem suas operações na manipulação de palavras completas, também é importante manipular campos de *bits* dentro da palavra ou mesmo *bits* individuais (um exemplo é a instrução **sb** já vista que manipula *bytes*).

Uma classe de instruções que permitem desse tipo operação é chamada de instruções de **deslocamento** (*shifts*). Tais instruções executam operações que movem todos os *bits* contidos em uma palavra para a esquerda ou para a direita direita, completando com 0 os *bits* que ficaram “vazios” devido ao deslocamento.. Por exemplo, se o registrador **\$s0** contiver:

```
0000 0000 0000 0000 0000 0000 0000 1101
```

E a instrução de deslocamento para a esquerda for executada com o parâmetro 8, o novo valor seria:

```
0000 0000 0000 0000 0000 1101 0000 0000
```

A instruções MIPS de deslocamento para a esquerda e direita são, respectivamente, **sll** (*shift left logical*) e **srl** (*shift right logical*). A instrução que realiza a operação do exemplo mostrado é a seguinte, assumindo que o resultado é armazenado em **\$t2**:

```
sll $t2, $s0, 8
```

Essas instruções possuem o formato tipo-R e utilizam o campo **shamt**, que armazena a quantidade de deslocamento a ser realizada. A versão em linguagem de máquina da instrução **sll \$t2, \$s0, 8** é mostrada a seguir.

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

Outro tipo de operação lógica utilizada para isolar campos de bits em uma palavra é executada pela instrução **and**, que examina *bit a bit* dois registradores e armazena, para cada comparação, o valor 1 quando ambos os *bits* dos operandos possuem o valor 1. O valor 0 é retornado para os outros casos. Em outras palavras, a instrução **and** executa a operação E-lógico para os bits de dois registradores. Por exemplo, se o registrador **\$t2** contém:

```
0000 0000 0000 0000 0000 1101 0000 0000
```

E o registrador **\$t1**:

```
0000 0000 0000 0000 0011 1100 0000 0000
```

Depois de executar a instrução MIPS:

```
and $t0, $t1, $t2
```

O valor do registrador **\$t0** será:

```
0000 0000 0000 0000 0000 1100 0000 0000
```

Além da operação **and** existe também a **or**, que verifica dois registradores *bit a bit*, armazenando o valor 1 quando o *bit* de algum dos operandos é 1, ou seja, fazendo um OU-lógico *bit a bit*. Assumindo os valores do exemplo anterior, a instrução

```
or $t0, $t1, $t2
```

Armazenaria o seguinte valor em **\$t0**:

```
0000 0000 0000 0000 0011 1101 0000 0000
```

Para os casos onde as constantes são necessárias, MIPS provê as instruções **andi** e **ori**. A Tabela 1 resume as operações lógicas apresentadas.

Operações	Operadores em C	Instruções MIPS
deslocamento à esquerda	<<	sll
deslocamento à direita	>>	srl
AND bit a bit	&	and, andi
OR bit a bit		or, ori

Tabela 1: Operações lógicas e suas operações correspondentes em C e MIPS.

Exemplo

O seguinte código em C aloca três campos em uma palavra chamada **receiver**.

```
int data;
struct
{
    unsigned int ready: 1;
    unsigned int enable: 1;
    unsigned int receivedByte: 8;
}receiver;
...
data = receiver.receivedByte;
receiver.ready = 0;
receiver.enable = 1;
```

Qual é o código MIPS para a sequência? Assumindo que data e receiver estão alocados em \$s0 e \$s1.

```
srl $s0, $s1, 2 #move o campo de 8 bits para a direita
andi $s0, $s0, 0x00ff #limpa (com a máscara 0x00ff) os bits que não pertencem ao campo
andi $s1,$s1, 0xfffe # valor 0 para o bit 0
ori $s1, $s1, 0x0002 # valor 1 para o bit 1
```