O que é o MIPS?

O MIPS é o nome de uma arquitetura de processadores baseados no uso de registradores. As suas instruções tem à disposição um conjunto de 32 registradores para realizar as operações. Entretanto, alguns destes registradores não podem ser usados por programadores, pois são usados pela própria máquina para armazenar informações úteis.

Processadores MIPS são do tipo RISC (Reduced Instruction Set Computer - ou seja, Computadores com Conjunto de Instruções Reduzidas). Isso significa que existe um conjunto bastante pequeno de instruções que o processador sabe fazer. Combinando este pequeno número, podemos criar todas as demais operações.

Instruções Aritméticas Simples

```
add $r1, $r2, $r3  # Esta instrução soma o conteúdo dos registradores
                   # $r2 e $r3 colocando o conteúdo no registrador $r1

addi $r4, $r1, 9   # Agora estamos somando o conteúdo do registrador
                   # $r1 com o valor imediato 9 e armazenando o
                   # resultado em $r4. O número imediato deve ter
                  # 16 bits.

addu $r5, $r6, $r4 # Quase igual ao add. Mas agora assumimos que
                   # todos os valores são não-negativos.

addiu $r7, $r8, 10 # Somamos o conteúdo de $r8 com o valor imediato
                   # 10 e armazenamos o resultado em $r7. Assume-se
                   # que todos os valores são não-negativos.

sub $r1, $r2, $r3  # Subtrai-se o conteúdo de $r3 do conteúdo de $r2
                   # e coloca-se em $r1. Também existe subi, subu e
                   # subiu que tem comportamento semelhante a addi,
                   # addu e addiu, mas para a subtração.
```

Instruções de Operadores Lógicos [editar | editar código-fonte]

```
and $r1, $r2, $r3  # Realiza uma operação AND bit-a-bit entre $r3 e $r2.
                   # O resultado é armazenado em $r1.

andi $r1, $r2, 42  # Realiza uma operação AND bit-a-bit entre $r2 e o valor
                   # imediato 42. O resultado é armazenado em $r1. O número
                   # imediato deve caber em 16 bits.

or $r1, $r2, $r3   # Realiza uma operação OR bit-a-bit entre $r3 e $r2.
                   # O resultado é armazenado em $r1.

ori $r1, $r2, 42   # Realiza uma operação OR bit-a-bit entre $r2 e o valor
                   # imediato 42. O resultado é armazenado em $r1. O número
                   # imediato deve caber em 16 bits.
```

Instruções de Uso de memória [editar | editar código-fonte]

As instruções do uso da memória seguem uma lógica diferente das instruções de operações aritméticas. Pra começar, existem três tipos de instruções capazes de copiar dados da memória para os registradores.

```
lw $r1, 4($r2) # Load Word: Esta instrução carrega uma palavra (estrutura de 4 bytes)
               # localizada no endereço representado pela soma do valor
               # armazenado no registrador $r2 mais 4. O resultado é armazenado em $r1.

lh $r1, 6($r3) # Load Half: Esta instrução carrega uma estrutura de 2 bytes localizada
               # no endereço representado pela soma do valor armazeado no
               # registrador $r3 mais o número 6. O resultado é armazenado em $r1.

lb $r1, 16($r2)# Load Byte: Esta instrução carrega um byte (8 bits) localizado no
               # endereço representado pela soma do valor armazenado em $r2 mais o
               # número 16. O resultado é armazenado em $r1.
```

Perceba que desta forma é rápido de carregar o conteúdo de um valor em um vetor. Basta saber o endereço do valor inicial do vetor e o índice de sua posição. Por exemplo, suponha que eu possua um vetor de caracteres **char vetor[5]**. Supondo que o registrador $r1 contenha o endereço de vetor[0], para armazenar o valor de vetor[3] no registrador $r2 e assumindo que cada caractere possui um byte, basta usar a seguinte instrução:

```
lb $r2 24($r1)
```

Colocamos o 24 lá porque cada caractere ocupa 8 bits e queremos pegar o quarto caractere da seqüência. Logo, precisamos nos deslocar 24 bits para acharmos o caractere certo. Afinal, a distância entre o primeiro elemento do vetor armazenado em $r1 e o quarto item do vetor é de 24:

```
[0][ ][ ][ ][ ][ ][ ][ ].[1][ ][ ][ ][ ][ ][ ][ ].[2][ ][ ][ ][ ][ ][ ][ ].[3][ ][ ][ ][ ][ ][ ]
```

Para armazenarmos conteúdo de um registrador na memória também existem 3 comandos:

```
sw $r1, 4($r2) # Store Word: Esta instrução carrega uma palavra (estrutura de 4 bytes)
               # localizada no registrador $r1 e armazena no endereço representado
               # pela soma do valor armazenado no registrador $r2 mais 4.

sh $r1, 4($r2) # Store Half: Esta instrução carrega uma estrutura de 2 bits
               # localizada no registrador $r1 e armazena no endereço representado
               # pela soma do valor armazenado no registrador $r2 mais 4.

sb $r1, 4($r2) # Store Byte: Esta instrução carrega um byte (8 bits)
               # localizado no registrador $r1 e armazena no endereço representado
               # pela soma do valor armazenado no registrador $r2 mais 4.
```

Instruções de Controle de Fluxo

Agora veremos instruções capazes de controlar o fluxo de execução de instruções de um computador. A primeira delas é a instrução **beq**, ou *Branch if Equal*:

```
beq $r1, $r2, DESTINO
```

O que esta instrução faz é verificar se o valor de $r1 é igual à $r2. Caso isso seja verdadeiro, ela muda o valor do registrador PC (Program Counter) que guarda o endereço da próxima instrução a ser executada. Se o valor passado como destino for 0, nada acontece. Nenhuma instrução é pulada, o programa executa a próxima instrução normalmente, independente de $r1 == $r2 ser verdadeiro ou falso. Se o valor passado for

"1", pula-se uma instrução se $r1 for igual a $r2. Se o valor passado for "2", pulam-se duas instruções e assim por diante. Também pode-se passar valores negativos. Um valor de "-1" faz com que o programa entre em um loop infinito, nunca mais passando para a próxima instrução. Uma "-2" faz com que voltemos uma instrução anterior, e assim por diante.

Entretanto, o valor de DESTINO só pode ser no máximo um valor com 16 bits. Com um valor destes, só podemos pular no máximo cerca de 32 000 instruções para frente ou para trás. Isso não é um problema na maioria das vezes é muito raro existir um programa que exija que você pule uma quantidade tão grande de instruções. Assumindo que 4 linhas de código e Assembly correspondem a 1 linha em código C, para que pulemos uma quantidade tão grande de instruções, teríamos que fazer um **if** com um bloco com mais de 8.000 linhas de código. Isso é algo extremamente difícil de ocorrer em um programa real.

Mas caso você realmente precise pular mais de 32 000 instruções, isso é perfeitamente possível, desde que você altere antes o valor do registrador PC. Com isso, perde-se um pouco de desempenho, mas torna desvios de código extremamente grandes possível. E o importante é que o caso comum (programas em C com blocos menores de 8.000 linhas) roda mais rápido (regra 4 da Filosofia de Projeto do MIPS).

Além do **beq**, temos também o **bne**, ou *Branch if Not Equal*:

```
bne $r1, $r2, DESTINO
```

Ele funciona da mesma forma que o **beq**. A diferença é que ele pula um determinado número de instruções somente se o valor dos dois registradores for diferente.

E finalmente, temos a instrução **j**, ou *Jump*:

```
j ENDEREÇO
```

Ele faz com que o programa passe a executar a instrução que é encontrada no endereço dado. O endereço passado para a instrução **j** é sempre um número de 26 bits. Entretanto, os endereços da máquina sempre são números de 32 bits. Então como será possível saber qual o endereço em que está a instrução desejada?

É muito simples. Instruções sempre estarão em endereços múltiplos de 4. Então sabemos que os dois últimos bits do endereço são sempre 0. Não precisamos armazená-los. Já os dois primeiros bits, retiramos do PC, pois é sempre muito mais provável que nós usemos o controle de fluxo para saltar para uma posição não muito distante. Caso precisemos saltar para um posição muito distante, basta alterarmos o valor do PC primeiro. Da mesma forma que foi visto no **beq**, o caso comum executa mais rápido desta forma. Isso é muito bom, mesmo que tenhamos mais trabalho e perda de desempenho nas pouquíssimas vezes nas quais precisamos dar saltos muito longos.

Instruções de Comparações

Por fim, precisamos ver ainda instruções de comparação. Um exemplo é o **slt**, ou *Set Less Than*:

```
slt $r1, $r2, $r3
```

Ela armazena 1 em $r1 se $r2 < $r3 e 0 caso contrário.

Resumo dos Modos de Endereçamento

Nas instruções do MIPS podemos representar os endereços de dados das seguintes formas:

- **A Registrador**: Representamos o dado passando o nome do registrador no qual ele está contido. Ex: **add $r1, $r2, $r2**.
- **Base-Deslocamento**: Representamos o dado passando o endereço de um vetor no qual ele está e a quantidade de bits a serem deslocados. Ex: **lw $r5, 4($r65)**.
- **Imediato**: Passamos o dado escrevendo o seu valor imediato. Ex: **addi $r1, $r2, 456**.
- **Relativo ao PC**: Passamos o dado descrevendo o seu valor relativo ao endereço da instrução atual. Ex: **beq $r1, $r2, DESTINO**.
- **Absoluto**: passamos o valor informando o seu endereço (pseudo-)absoluto. Ex: **j DESTINO**.

Existem apenas estas 5 formas de endereçamento no MIPS. Através delas, efetuamos todas as operações necessárias.