TUTORIAL DE INSTRUÇÕES MIPS ASSEMBLY
AUTOR: João Ricardo dos Santos Rosa

1. As instruções

1.1. As instruções MIPS

O assembly é uma linguagem de programação de baixo nível sendo uma representação simbólica de códigos binários, também chamados de linguagem de máquina.

O assembly possui micro-instruções, que indicam operações que o processador deve realizar. Um conjunto ordenado de zeros e uns constitui uma instrução. As instruções do processador MIPS são compostas por 32 bits.

1.2. Instruções de leitura e escrita

1.2.1. li

SINTAXE: li <registrador>,<valor imediato>

A instrução load immediately (ler imediatamente) é uma facilidade de leitura que foi implantada na linguagem para atribuir um valor inteiro diretamente a um registrador.

Exemplo:

```
li $t0, 5      # Atribuindo o valor 5 ao registrador $t0
li $t1, 3      # Atribuindo o valor 3 ao registrador $t1
li $t2, -2     # Atribuindo o valor -2 ao registrador $t2
li $t3, 10     # Atribuindo o valor 10 ao registrador $t3
```

1.2.2. la

SINTAXE: la <registrador>,<endereço de memória variável>

A instrução load address (ler pelo endereço de memória da variável) é uma facilidade de leitura que foi implantada na linguagem para atribuir o valor armazenado numa variável diretamente num registrador. Usamos esse comando no código para exibir mensagem string na tela, no exemplo do Olá Mundo, no subitem 5.1.

```
.data                   # Diretiva de dados
  msg: .asciiz          "Olá Mundo!"
.text                   # Diretiva texto
  la $a0,  msg          # $a0 = msg
  li  $v0,  4           # Imprime uma string
  syscall               # Executa
```

Veja este outro exemplo:

Este outro código, criei uma variável chamada idade que tem um valor de 43. Fiz a leitura do endereço de memória da variável no registrador **$t0**. Vamos ver como ficou a tela de execução no MARS:

```
.data                           # Diretiva de dados
   idade:   .word      43       # idade = 43
.text                           # Diretiva de texto
la $t0,  idade                  # $t0 = idade
li  $v0,  10                    # Sair do programa
syscall                         # Executa
```

Registradores:

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	268500992
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0

Segmento de dados:

Data Segment

Address	Value (+0)	Value (+4)
268500992	43	0
268501024	0	0
268501056	0	0
268501088	0	0
268501120	0	0
268501152	0	0
268501184	0	0
268501216	0	0

Veja no segmento de dados que na posição de memória **268500992** tem um valor 43 em Value (+0). Este mesmo valor está sendo mostrado em **$t0** e em **$at** (registrador para **a**rgumento **t**emporário, usado pelo compilador do simulador). Com a instrução **la** podemos apontar um endereço de memória e armazenar esse valor num registrador para posteriormente localizar o conteúdo através de outra instrução.

1.2.3. lw

SINTAXE: lw <registrador>,<variável>

As instruções de leitura e escrita utilizam acesso direto à memória principal. As memórias possuem dois status: grava ou lê. Usaremos a instrução **l<tipagem>** para ler um dado na memória RAM e gravar num registrador. A tipagem depende da definição do tipo de constante ou variável que você criou no início.
Exemplo:

Criamos uma constante do tipo **.word** que registra números inteiros. Lembre-se que a área de declaração de variáveis no código MIPS assembly fica na diretiva **.data** ou segmento de dados. Vejamos:

```
.data                   # Diretiva de dados
idade: .word    40      # idade = 40
```

Na diretiva **.text**, nós podemos faremos a leitura do valor da constante e gravaremos num registrador.

Usaremos a instrução l<tipagem> (load) para ler o valor da constante diretamente num registrador. Escolheremos neste exemplo o registrador **$t0** para receber o valor constante idade que vale 40 (inteiro). Como a constante é do tipo **.word**, usaremos a instrução: **lw**

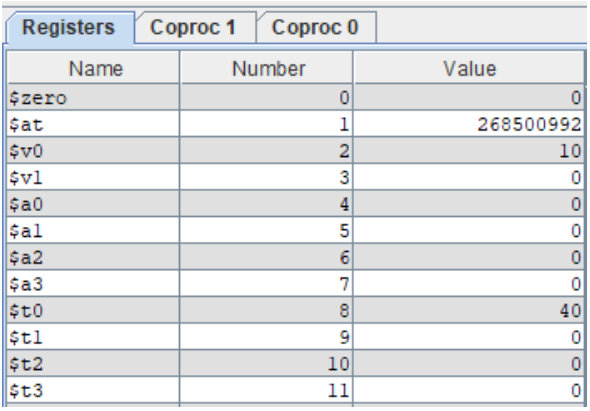
```
.text                   # Diretiva de texto
  lw  $t0, idade        # $t0 = idade
  syscall               # Executa
```

Abra seu simulador MARS, e digite esse código abaixo. Salve seu código e depois clique na ferramenta para assemblar seu código.

Veja o registrador **$t0**. Se tudo deu certo o número 40 deverá estar dentro do registrador.

```
.data                          # Diretiva de dados
    idade:      .word  40      # idade = 40
.text                          # Diretiva de texto
  lw $t0,  idade               # $t0 = idade
  li  $v0,  10                 # Sair do programa
  syscall                      # Executa
```

Veja o valor no registrador **$t0** como ficou:



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	40
$t1	9	0
$t2	10	0
$t3	11	0

1.2.4.sw

SINTAXE: sw <registrador>,<variável>

Usaremos a instrução **sw** (save) para gravar um dado de um registrador diretamente na memória RAM. Esta instrução é o inverso da instrução **lw** (load).

Neste exemplo, criaremos uma variável chamada idade, do tipo **.word**, inicializada com o valor zero.

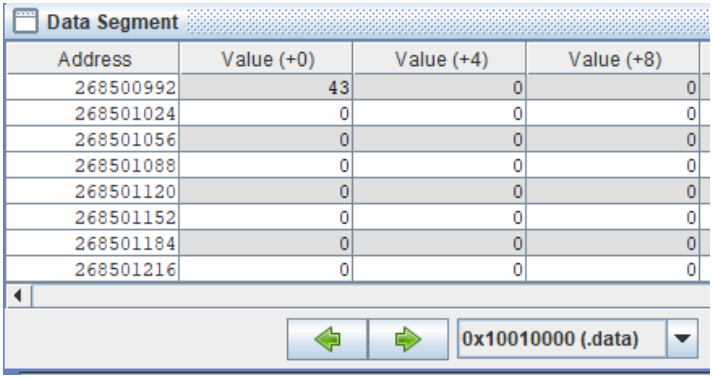
O programa irá pedir ao usuário que digite um número inteiro e este número inteiro será gravado na variável idade. O registrador que será manipulado será o **$t0**. Ao final da execução do código será possível ver o valor digitado no registrador **$t0** e na variável idade que estará na área Value +0 no simulador MIPS.

```
.data                       # Diretiva de dados
   idade:  .word    0       # idade = 0

.text                       # Diretiva de texto
```

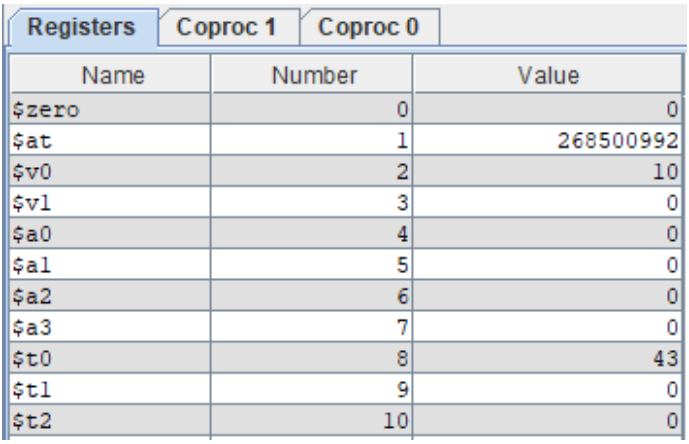
```
# Pedindo para o usuário digitar um valor
  li $v0, 5                    # Ler um número inteiro
  syscall                      # Executa
  move $t0, $v0                # $t0 recebe $v0

# Salvando na variável o número digitado
  sw $t0, idade                # $t0 = idade

# Finalizando o programa
  li  $v0, 10                  # Sair do programa
  syscall                      # Executa
```

Quando executamos o código, foi digitado o valor 43 e na tela do MARS apresentou o valor 43 no registrador **$t0** e na área de segmentação de dados (Data Segment) na coluna Value (+0).

Segmento de Dados:



Registradores:



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	43
$t1	9	0
$t2	10	0

Vejamos este outro exemplo, usando a instrução **sw** para construir um vetor de dados inteiros (**.word**).

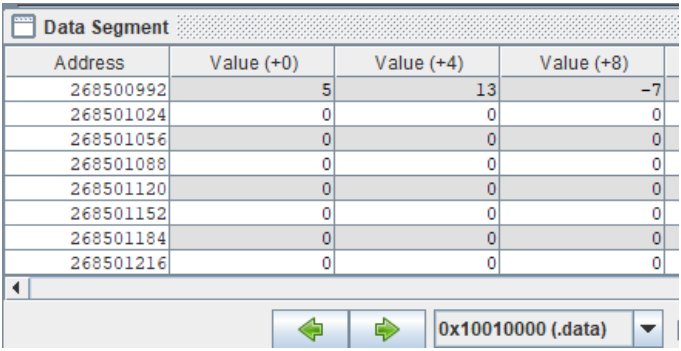
Vamos ver como ficaria um código em MIPS assembly trabalhando com vetores (Array).

```
.data                    # Diretiva de dados
   vetor1:  .space  12   # vetor1 = 3 números
.text                    # Diretiva de texto
   la $t0,vetor1         # Carrega o vetor no registrador
   li $t1,5              # $t1 = 5 Value(+0)
   sw $t1,0($t0)         # Registra o primeiro elemento
   li $t1,13             # $t1 = 13 Value(+4)
   sw $t1,4($t0)         # Registra o segundo elemento
   li $t1,-7             # $t1 = -7 Value(+8)
   sw $t1,8($t0)         # Registra o terceiro elemento
   li $v0, 10            # Sair do programa
   syscall               # Executa
```

No código acima, mostra que existe um vetor chamado vetor1 do tipo **.space**, separando 12 espaços de memória. Este código vai armazenar 3 valores do tipo

.word (inteiro), cada valor **.word** ocupará 4 espaços de memória. Após a execução do código a área de segmentação de dados ficou assim:

Segmento de dados:



A segmentação de dados mostra que em Value (+0) foi armazenado o valor 5, em Value (+4), o valor 13 e em Value (+8) valor -7.

1.2.5. lb e sb

As instruções **lb** (load byte) e **sb** (save byte), tratam dados do tipo caractere. O byte é usado para armazenar o código da representação dos caractere na tabela ASCII. É só lembrar que 8 bits forma 1 byte e 1 byte é o espaço de representação de 1 caractere, seja letra, número ou símbolo. Vejamos o código:

```
.data                    # Diretiva de dados
  var1:   .byte  'A'     # var1 = 'A'
.text                    # Diretiva de texto
  lb $t0, var1           # $t0 = var1
  li $t1, 97             # $t1 = 97
  sb $t1, var1           # $t1 = var1
```

Neste exemplo criei uma variável chamada de var1 do tipo **.byte**, contendo o caractere 'A'.

Sabemos que o caractere 'A' tem o valor 65 na tabela ASCII. Coloquei em **$t1** o valor de 97 e depois alterei o valor da variável var1 também para 97 usando o valor armazenado em **$t0**.

É possível ver na área de segmento de dados que está armazenado o valor de 97 lá também.

Vejamos a tela do MARS após a execução do código:

Registradores:

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	65
$t1	9	97
$t2	10	0

Segmentação de dados:

Data Segment

Address	Value (+0)	Value (+4)
268500992	97	0
268501024	0	0
268501056	0	0
268501088	0	0
268501120	0	0
268501152	0	0
268501184	0	0
268501216	0	0

É possível ver o char 'a' em Value (+0), na área de segmentação de dados quando você alterar a exibição para [x] Ascii. Experimenta alterar o modo de exibição dos dados e veja se você consegue ver o caractere 'a' armazenado.

1.3. Instruções de movimentação

1.3.1. move

SINTAXE:
move <registrador_destino>,<registrador_origem>

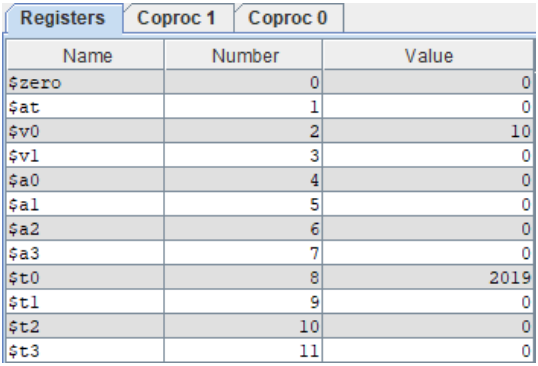
A instrução **move** copia os dados do registrador origem para o registrador destino. É uma instrução de movimentação de dados, de transferência de dados. Usamos muito a instrução **move** quando setamos o registrador **$v0** para receber dados de entrada do usuário, por exemplo, pedir ao usuário que ele digite um número inteiro.

O dado que é recebido, fica armazenado no registrador **$v0** temporariamente, sendo então necessária a movimentação do valor armazenado em **$v0** para outro registrador, pelo seguinte motivo: com a manipulação do registrador **$v0** para que o sistema realize outras tarefas perderemos a informação que fora antes armazenada.
Usamos a instrução **move** da seguinte forma:

```
.data                    # Diretiva de dados
.text                    # Diretiva de texto
  li $v0, 5              # Ler um número inteiro do teclado
  syscall                # Executa
```

```
move $t0, $v0       # Salvar no registrador $t0
li $v0 ,10          # Sair do programa
syscall             # Executa
```

O número digitado será armazenado inicialmente no registrador **$v0** e o comando **move** fará uma movimentação do número para o registrador **$t0**. No momento da execução do código, foi digitado o número inteiro 2019 como entrada. Veja como ficou nos registradores:



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	2019
$t1	9	0
$t2	10	0
$t3	11	0

Fizemos a movimentação do dado que estava em **$v0** para **$t0**. Veja que depois manipulamos o registrador **$v0** para ele receber o número inteiro 10 que faz uma chamada de sistema para finalizar o programa. Se não fosse a movimentação com a instrução **move**, o número 2019 que estava registrado anteriormente teria sido perdido.

1.3.1. mfhi

SINTAXE: mfhi <registrador_destino>

A instrução **mfhi** copia a informação que está no registrador **hi** para o registrador destino. O registrador **hi** é usado na multiplicação e na divisão para ganho de funcionalidade. Por exemplo, na divisão o registrador **hi** pode conter o resto da divisão, nos possibilitando assim sabermos se um determinado número é par ou ímpar.

A instrução que vou mostrar agora é a instrução **div** que realiza a divisão do número contido no registrador 1 pelo registrador 2.

Exemplo:

X = 5; Y = 2

Se eu dividir 5 / 2 e for buscar o resto desta divisão encontraremos 1 como resultado, indicando que 5 é ímpar.

Vou colocar o número 5 no registrador **$t1** e o número 2 no registrador **$t2**.

Usarei a instrução **div** para dividir o registrador **$t1** pelo registrador **$t2** e em seguida farei a movimentação dos dados do registrador **hi** para o registrador **$t0**.

Ao final o registrador **$t0** deverá mostrar o valor 1 indicando que 5 é ímpar.

```
.data                   # Diretiva de dados
.text                   # Diretiva de texto
    li $t1, 5           # $t1 = 5
    li $t2, 2           # $t2 = 2
    div $t1, $t2        # hi = $t1/$t2
    mfhi $t0            # Registrador $t0 recebe hi
```

```
li $v0, 10          # Sair do programa
syscall             # Executa
```

Vejamos nos registradores:

$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194328
hi		1
lo		2

O registrador **hi** depois da divisão recebeu o resto igual a 1 (um).

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	1
$t1	9	5
$t2	10	2
$t3	11	0
$t4	12	0

Após a execução da instrução **mfhi $t0**, houve a movimentação dos dados do registrador **hi** para **$t0**.

Note que **$t0** também está com o valor de 1. Isto indica que o 5 é ímpar.

Faça o teste com este mesmo código acima, troque o número do registrador **$t1** de 5 para 6, salve o código e execute-o. Veja se o registrador hi e o registrador **$t0** ficarão com o valor 0. Isto indicará que 6 é par.

1.3.1. **mflo**

SINTAXE: mflo <registrador_destino>

A instrução mflo copia a informação que está no registrador **lo** para o registrador destino. O registrador **lo** é usado na multiplicação e na divisão para o resultado do cálculo.

Imagine o seguinte cálculo:

X = 18

Y = 3

Z = X / Y, qual será o valor de Z ?

No MIPS assembly, precisamos colocar os valores de X e de Y nos registradores.

Vamos colocar em **$t1** o valor 18 e no registrador **$t2** o valor 3.

Então vamos dividir 18 por 3 com a instrução **div**.

O resultado da divisão estará no registrador **lo**, precisando apenas que realizemos a movimentação dos dados para obter o valor num registrador temporário.

No exemplo, colocarei o resultado em **$t0**.

```
.data                   # Diretiva de dados
.text                   # Diretiva de texto
    li $t1, 18          # $t1 = 18
    li $t2, 2           # $t2 = 3
    div $t1,$t2         # hi = $t1/$t2
    mflo $t0            # Registrador $t0 recebe lo
    li $v0, 10          # Sair do programa
    syscall             # Executa
```

Note que no meu exemplo eu usei o registrador temporário **$t0**. Você pode fazer a movimentação com a instrução mflo para qualquer outro registrador.

Registrador **lo**:

$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194328
hi		0
lo		6

Então, 18 dividido por 3 é 6.
Encontramos esse valor no registrador **lo**.
Também iremos encontrar o mesmo valor.

Registradores:

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	6
$t1	9	18
$t2	10	3
$t3	11	0

1.4. Instruções aritméticas

1.4.1. Somar com as instruções **add** e **addi**

Podemos usar muitas instruções para fazer a soma aritmética. Mostraremos duas instruções em específico: **add** e **addi.** Usaremos 3 registradores para manipular o processo de cálculo. Vejamos a sintaxe:

SINTAXE:
add <registrador_destino>,<registrador1>,<registrador2>

O registrador1 e o registrador2 são os registradores que conterão os números para os cálculos. O registrador_destino conterá o resultado da soma dos registradores 1 e 2. Da mesma forma a instrução **addi** será usada.

SINTAXE:
addi <registrador_destino>,<registrador1>,<registrador2>

Também com três registradores, sendo o primeiro registrador o que irá conter o resultado da soma aritmética.

Então você deve estar se perguntando, e qual a diferença entre **add** para **addi** ? Bem, a instrução **add**, soma registradores, e a instrução **addi** pode somar números inseridos diretamente no local do registrador.

Exemplo:

```
li $t1, 10                    # $t1 = 10
li $t2, 20                    # $t2 = 20
add $t0, $t1, $t2       # $t0 = $t1 + $t2
```

Ao executarmos a instrução de soma com o **add**, o resultado da soma dos registradores **$t1** e **$t2** estará no registrador **$t0**.

Já utilizando a instrução **addi** poderíamos lançar os números na própria instrução imediatamente, sem precisar antes carregar os dados em todos os registradores, bastando apenas um registrador ser carregado, enquanto que o outro número pode ser colocado diretamente.

```
li $t1, 10          # $t1 = 10
addi $t0, $t1, 5    #$t0 = $t1 + 5, ou seja $t0 = 15
```

Se formos ao registrador **$t0** após a execução da instrução veremos o valor 15 armazenado.

Vejamos este outro exemplo como ficou a simulação no MARS e como foram apresentados os valores nos registradores. Se tudo ocorreu bem, o registrador **$t3** deverá apresentar os resultados.

```
.data                          # Diretiva de dados
.text                          # Diretiva de texto
  li $t1, 5                    # $t1 = 5
  li $t2, 2                    # $t2 = 2
  add $t0, $t1, $t2            # $t0 = $t1 + $t2
  addi $t3, $t0, 10            # $t3 = $t0 + 10
  li $v0, 10                   # Sair do programa
  syscall                      # Executa
```

Vamos ver como ficaram os registradores após a execução das instruções:

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	7
$t1	9	5
$t2	10	2
$t3	11	17
$t4	12	0

O registrador **$t1** foi carregado com o número 5, o registrador foi carregado com o número 2; a soma dos registradores **$t1** com o **$t2** foi armazenado no

registrador **$t0**, carregado com o número 7 (5+2). E para finalizar somamos diretamente o valor 10 ao registrador **$t0** e salvamos o resultado do cálculo no registrador **$t3**, indicando na imagem o número 17.

1.4.2. Subtrair com as instruções **sub** e **subu**

As instruções **sub** e **subu** são semelhantes as instruções **add** e **addi**. A primeira instrução **sub**, trabalha com três registradores com dois carregados com dados, enquanto que a instrução **subu** pode carregar um valor imediatamente a própria instrução **subu**. Veja a sintaxe das instruções:

SINTAXE:
sub <registrador_destino>,<registrador1>,<registrador2> e
subu <registrador_destino>,<registrador1>,<registrador2>

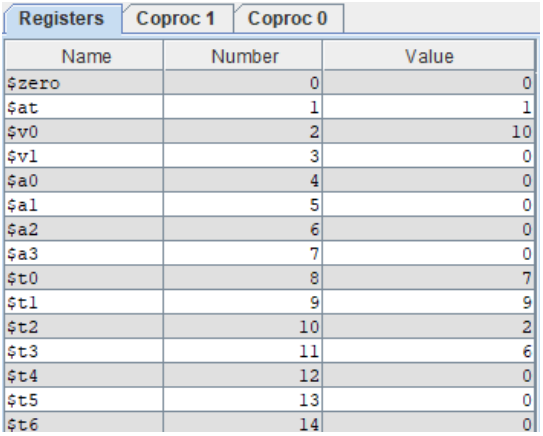
Vamos fazer um exemplo:

```
.data                          # Diretiva de dados
.text                          # Diretiva de texto
      li $t1, 9                # $t1 = 9
      li $t2, 2                # $t2 = 2
      sub $t0, $t1, $t2        # $t0 = $t1 - $t2
      subu $t3, $t0, 1         # $t3 = $t0 – 1
      li $v0, 10               # Sair do programa
      syscall                  # Executa
```

Neste exemplo, o registrador **$t1** é carregado com o valor 9 e o registrador **$t2** é carregado com o valor 2. Imediatamente é calculado **$t1** menos **$t2** e armazenado

o resultado no registrador **$t0** (**$t0** = 7). Na sequência a instrução subu é executada com o registrador **$t0** valendo 7 e diretamente no segundo registrador de entrada sendo colocado o valor 1, para então armazenar o resultado da subtração em **$t3** que será o valor 6 (7-1).

Vejamos nos registradores:



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	1
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	7
$t1	9	9
$t2	10	2
$t3	11	6
$t4	12	0
$t5	13	0
$t6	14	0

1.4.3. Multiplicar com as instruções **mult** e **mul**

As instruções **mult** e **mul** se diferenciam uma da outra pela quantidade de registradores. Quando usamos o mult trabalhamos com dois registradores e o resultado da multiplicação vai para o registrador **lo**. Temos que usar o mflo para fazer a movimentação do dado para um registrador temporário.

SINTAXE:
mult <registrador1>,<registrador2>

Já com a instrução **mul**, usamos três registradores, sendo o primeiro registrador usado para receber o resultado dos outros dois registradores usados na sequencia.

SINTAXE:
mul <registrador_destino>,<registrador1>,<registrador2>

Vamos aos códigos? Apresentaremos dois códigos, um usando o **mult** e o mflo e outro código usando apenas a instrução **mul**.

```
.data                   # Diretiva de dados
.text                   # Diretiva de texto
    li $t0, 3           # $t0 = 3
    li $t1, 2           # $t1 = 2
    mult $t0, $t1       # lo = $t0 * $t1
    mflo $t2            # Registrador $t2 recebe lo
    li $v0, 10          # Sair do programa
    syscall             # Executa
```

Neste primeiro código, o resultado é armazenado no registrador **lo** e depois é feito a movimentação para o registrador **$t2**.

Neste segundo código, iremos usar o terceiro registrador, mas mudaremos a instrução para apenas **mul**.

Também é possível carregar um valor diretamente na instrução **mul** (semelhante ao que vimos na instrução **addi**).

Vejamos:

```
.data                   # Diretiva de dados
.text                   # Diretiva de texto
    li $t0, 3           # $t0 = 3
    li $t1, 2           # $t1 = 2
    mul $t2, $t0, $t1   # $t2 = $t0 * $t1
    li $v0, 10          # Sair do programa
    syscall             # Executa
```

Veja como ficam os registradores:

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	3
$t1	9	2
$t2	10	6
$t3	11	0

Veja como fica o registrador lo:

$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194324
hi		0
lo		6

Continua recebendo o resultado do cálculo da multiplicação.

1.4.4. Dividir com a instrução **div**

SINTAXE:
div <registrador_destino>,<registrador1>,<registrador2>

A instrução **div** pode ser usada juntamente com os registradores **hi** e **lo** ou não. Se você usar apenas dois registradores para o cálculo o assembly usará os registradores **hi** e **lo** para registrarem os resultados ou caso você use três registradores para o cálculo, o primeiro registrador será usado como retorno do cálculo.

Neste primeiro exemplo abaixo, usaremos os registradores **hi** e **lo** para receberem o resultado da divisão entre 5 e 2. Apenas os registradores do Coprocessador 1 são projetados para trabalharem com ponto flutuante, no caso float ou double. Os resultados da instrução div sempre serão truncados e mostrados de forma inteira. Exemplo: 5/2 = 2,5. Neste caso será

registrado apenas o valor 2 sendo descartado, desprezado os resultados após a casa decimal. Para calcular e exibir os resultados usando a casa decimal será necessário a codificação usando a tipagem **.float** ou **.double** e a utilização dos registradores do Coproc1, além de usar comandos próprios para manipular os dados nos registradores do Co-processador 1. Não abordaremos neste livro o uso dos coprocessadores.

Vejamos o código:

```
.data                   # Diretiva de dados
.text                   # Diretiva de texto
    li $t0, 5           # $t0 = 3
    li $t1, 2           # $t1 = 2
    div $t0, $t1        # lo = $t0/$t1
    mflo $t2            # Registrador $t2 recebe lo
    mfhi $t3            # Registrador $t3 recebe hi
    li $v0, 10          # Sair do programa
    syscall             # Executa
```

O registrador foi carregado com o número 5 e o registrador **$t1** foi carregado com o valor 2, ambos números inteiros. Quando usado a instrução **div $t0**,$t1 no código, os registradores **hi** e **lo** foram usados para armazenarem o resultado.

Vejamos como ficou:

$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194332
hi		1
lo		2

O registrador **hi** armazenou o resto da divisão e o registrador **lo** armazenou o resultado da divisão truncando para exibir apenas a parte inteira do cálculo.

Imediatamente foi feito a movimentação dos dados dos registradores **hi** e **lo** e os resultados também foram carregados nos registradores **$t2** e **$t3**. O registrador lo foi movimentado para o registrador **$t2** e o registrador **hi** foi movimentado para o registrador **$t3**.

Vamos ver:

Registers | Coproc 1 | Coproc 0

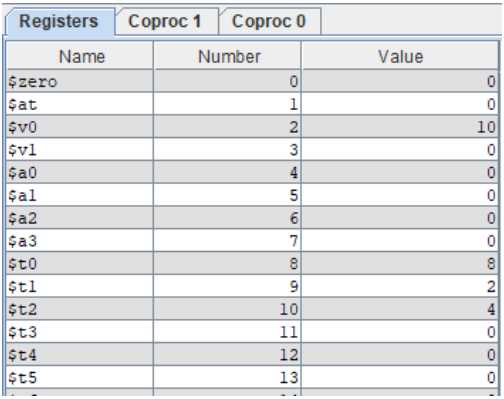
Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	5
$t1	9	2
$t2	10	2
$t3	11	1
$t4	12	0

O próximo código nós vamos mostrar usando três registradores, veja como fica mais simples. A diferença é que não consigo saber o resto da divisão, apenas o resultado da divisão inteira.

Vejamos o segundo código com o **div** usando três registradores:

```
.data                    # Diretiva de dados
.text                    # Diretiva de texto
    li $t0, 8            # $t0 = 8
    li $t1, 2            # $t1 = 2
    div $t2, $t0, $t1    # $t3 = $t0/$t1
    li $v0, 10           # Sair do programa
    syscall              # Executa
```

Veja os registradores:



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	8
$t1	9	2
$t2	10	4
$t3	11	0
$t4	12	0
$t5	13	0

O registrador **$t0** armazenou 8, o registrador **$t1** armazenou 2 e a instrução **div** efetuou a divisão entre **$t0**/**$t1** e armazenou o resultado no registrador **$t2**.

1.5. Instruções de desvios condicionais

A partir de agora apresentaremos um novo recurso que é a criação dos desvios de códigos.

Nós podemos colocar um nome aos desvios.

Lembra que no início criamos uma diretiva **.globl** na qual informamos a função principal? Pois bem, nós criamos um bloco de código etiquetado com o nome principal e dissemos que o assembly deveria executar este bloco em primeiro lugar.

Você poderá relembrar indo para o subitem 4.5.

Nossos códigos terão o bloco de execução principal com uma condicional.

Neste subitem trataremos das estruturas condicionais, ou melhor, desvios condicionais. Existem várias instruções no MIPS assembly que realizam os desvios condicionais, vejamos alguns dos principais: **beq**, **blt**, **ble**, **bgt**, **bge** e **bne.**

O **beq** realiza o desvio caso o conteúdo do primeiro registrador seja igual ao segundo registrador analisado, caso positivo devemos informar qual o bloco de comandos ele deverá ir. Por exemplo:

```
beq $t0, $t1, Iguais
```

Em alguma parte do código eu preciso ter um bloco chamado **Iguais**. Lembra-se de como declaramos uma variável no segmento de dados? De forma idêntica iremos nomear os blocos de códigos usando os dois pontos **(:).**

```
Iguais:
    Instrução1
    Instrução2
    Instrução3
```

Todas as instruções seguirão este mesmo padrão, sempre desviando para um bloco de execução que devemos definir.

Vejamos abaixo a sintaxe de cada instrução de desvio condicional.

1.5.1. As instruções **beq** e **bne**

Branch for Equal – Desvie se for igual. A instrução **beq** desvia quando o primeiro registrador é igual a o segundo registrador. Podemos usar a instrução **beq** para um programa de menu ou para identificar uma condição específica dentro do código. Veja um exemplo:

```
beq $t0, $t1, bloco        # desvia se $t0 = $t1
```

Branch for Not Equal – Desvie se não for igual ou desvie se for diferente. A instrução **bne** desvia quando o primeiro registrador é diferente do segundo registrador. Podemos usar a instrução **bne** como um caso contrário ao **beq**. Veja um exemplo:

```
bne $t0, $t1, bloco        # desvia se $t0 <> $t1
```

Vejamos um exemplo de código usando estas duas instruções. Nosso programa fará a análise de um número inteiro, caso esse número for zero o programa dará uma mensagem na tela, caso contrário ele dará outra mensagem na tela segundo a tabela abaixo:

CONDIÇÃO	MENSAGEM
Igual a zero (**beq**)	Foi detectado o número ZERO!
Diferente de zero (**bne**)	Este número não é zero!

Vamos ao código: Abra o simulador MARS, digite o código abaixo e faça a simulação para identificar os desvios condicionais.

```
.data                          # Diretiva de dados
  pergunta:      .asciiz       "Digite um número: "
  eh_zero:       .asciiz       "Foi detectado o Zero!\n"
  nao_eh_zero:   .asciiz       "Este número não é Zero!\n"
.text                          # Diretiva de texto
  # Imprimir
  li $v0, 4                    # Imprime uma string
  la $a0, pergunta             # Carrega a string no registrador
  syscall                      # Executa
  # Imprimir
  li $v0, 5                    # Lê um número inteiro
  syscall                      # Executa
  # Grava o número digitado em $t0
  move $t0, $v0                # $t0 = $v0
  # Desvios condicional usando beq e bne
  beq $t0,0,seforzero          # Se $t0 for 0,
                               # executa "seforzero"

  bne $t0,0,senaoforzero       # Se $t0 não for 0,
                               # executa "senaoforzero"

  # Condição se for zero
  seforzero:
  # Imprimir
  li $v0, 4                    # Imprime uma string
  la $a0, eh_zero              # Carrega a string no registrador
```

```
syscall                    # Executa
#Depois finaliza o programa
li $v0, 10                 # Sair do programa
syscall                    # Executa

# Condição se não for zero
senaoforzero:
# Imprimir
li $v0, 4                  # Imprime uma string
la $a0, nao_eh_zero        # Carrega a string no registrador
syscall                    # Executa
# Depois finaliza o programa
li $v0, 10                 # Sair do programa
syscall                    # Executa
```

O **beq** e o **bne** estão se comportando neste código como um SE e um SE NÃO. Em ambas as condições depois de mostrarem a mensagem proposta, o programa é finalizado.

1.5.2. As instruções **bge** e **blt**

As instruções **bge** **e** **blt** são antagônicas. A instrução **bge** desvia o fluxo de dados se o número contido no registrador 1 for maior ou igual ao do registrador 2, enquanto que a instrução **bl**t desvia o fluxo se o valor contido no registrador 1 for menor do que o valor contido no registrador 2.

```
bge $t0, $t1, bloco        # desvia se $t0 >= $t1
```

Com a instrução **bge**, desviamos o fluxo para a sub-rotina chamada bloco quando **$t0** for maior ou igual a **$t1**.

```
blt $t0, $t1, bloco          # desvia se $t0 < $t1
```

Com a instrução **blt**, desviamos o fluxo para a sub-rotina chamada bloco quando **$t0** for menor do que **$t1**.

Com essas instruções, podemos, por exemplo, criar um programa que identificaria através idade do usuário se ele poderia ou não tirar sua carteira de motorista, identificando sua maior idade e também o seu caso contrário.

1.5.3. As instruções ble e bgt

As instruções **ble** e **bgt** também são antagônicas. A instrução **ble**, desvia o fluxo de dados se o número contido no registrador 1 for menor ou igual ao do registrador 2, enquanto que a instrução **bgt**, desvia o fluxo se o valor contido no registrador 1 for maior do que o valor contido no registrador 2.

```
ble $t0, $t1, bloco          # desvia se $t0 <= $t1
```

Com a instrução **ble**, desviamos o fluxo para a sub-rotina chamada bloco quando **$t0** for menor ou igual a **$t1**.

```
bgt $t0, $t1, bloco        # desvia se $t0 > $t1
```

Com a instrução **bgt**, desviamos o fluxo para a sub-rotina chamada bloco quando **$t0** for maior do que **$t1**.

1.6. Desvio incondicional com a instrução **j**

A instrução que realiza o desvio incondicional é aquela que não precisa de condição para realizar a execução de um desvio.

Usamos este artifício na programação quando queremos modularizar nossos códigos, quando executamos uma função e até mesmo quando executamos algum procedimento.

O desvio incondicional mostrado neste subitem é a instrução **j** (jump) que fará saltos para outros blocos de execução.

Também podemos encontrar o conceito de saltos incondicionais.

O desvio incondicional também cria uma estrutura de repetição, loop infinito ou finito, iterativo ou não.

Exemplo:

```
.data                      # Diretiva de dados
.text                      # Diretiva de texto
.globl principal           # Diretiva global
principal:
   Instrução_1
   Instrução_2
   Instrução_3
   j principal             # O laço infinito foi criado
```

Veja que criamos um bloco de códigos chamado de **principal** e que ele executa 3 instruções e depois ele executa o desvio incondicional (ou um pulo) para o início do bloco de códigos **principal**, ou seja, ele está num loop infinito.

Se quiser que o programa saia do loop infinito, será necessário colocar uma instrução de desvio condicional para que, quando a condição for satisfeita o programa faça outro salto para outro bloco de códigos saindo assim do laço.

Por exemplo:

Vamos criar um código que receba do usuário um número. Caso o usuário digite zero, o programa termina, caso contrário ele continua no loop pedindo que o usuário insira um novo número. O laço só vai ser quebrado se o usuário digitar o valor zero.

Veja:

```
.data                          # Diretiva de dados
   pergunta:      .asciiz      "Digite um número: "
.text                          # Diretiva de texto
.globl principal               # Diretiva global

principal:                     #### Bloco principal
   li $v0, 4                   # Imprimir string
   la $a0, pergunta            # Carrega a string
   syscall                     # Executa
   li $v0, 5                   # Lê um numero inteiro
   syscall                     # Executa
   move $t0, $v0               # Move o número para $t0
   beq $t0,0,sair              # Se $t0 = 0, vai para bloco sair
   j principal                 # vai para o início de principal
```

```
sair:                    # Bloco chamado sair
   li $v0,10             # Sair do programa
   syscall               # Executa
```

1.7. Chamada de sub-rotina com a instrução jal

A instrução **jal** (jump and link) é usada para chamar um procedimento, chamar uma função. O **jal** é um salto (jump) na intenção de interagir e retornar para a mesma parte do código que foi chamado. A instrução **jal** trabalha com a instrução **jr** (jump register).

Todas as vezes que programamos para desviar o fluxo, o sistema armazena o último endereço de execução no registrador **$ra**. A instrução **jr**, usa o registrador **$ra** para retornar para o ponto em que foi chamado.

Por exemplo, imagine que um programa em sua execução chama uma função dobro. O número é multiplicado por dois e depois de feito é necessário o programa voltar para a linha de execução em que foi feita a chamada da função. Usamos o **jr $ra** voltar.

A instrução **jal** chama um bloco de códigos, que funcionará como um procedimento ou função, e o **jr $ra** volta para a linha do jal que o chamou, assim continuando o código sequencialmente.

Vamos criar um contador, será uma espécie de for. Chamaremos a instrução **jal** para fazer a verificação.

```
.data                          # Diretiva de dados
   mensagem:     .asciiz       "Fim da contagem..."
   espaco:       .asciiz       ", "

.text                          # Diretiva de texto
prog:
    addi $t0, $zero, 0         #Registrador t0 = 0

enquanto:
    bgt $t0, 10, sair          # se for maior do que 10
                               # vá para a função sair
    jal imprimirnumero         # Chama a função
    addi $t0, $t0, 1           # i = i + 1
    j enquanto                 # Volta para enquanto/loop

sair:                          # Função SAIR
   li $v0, 4                   # Imprime a string
   la $a0, mensagem            # Imprime na tela a variável
   syscall                     # Executa
   li $v0, 10                  # Finaliza o programa
   syscall                     # Executa

imprimirnumero:                # Função IMPRIMENUMERO
    li $v0, 1                  # Imprime número inteiro
    add $a0, $t0, $zero        # Exibe o conteúdo de $t0
    syscall                    # Executa
    li $v0,4                   # Imprime mensagem
    la $a0,espaco              # Exibe a variável na tela
    syscall                    # Executa
    jr $ra                     # Retorna a função
```