INSTITUTO FEDERAL
SANTA CATARINA

Microprocessadores

Hugo Marcondes
hugo.marcondes@ifsc.edu.br

Aula 02

Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina

Programação Assembly: Documentação!

```
# Author:       your name
# Date:         current date
# Description:  high-level description of your program
```

```
.data
```

(constant and variable definitions)

```
.text
```

```
# Section 1: what this part does
# Pseudocode:
#   (algorithm in pseudocode)
# Register mappings:
#   (mapping from pseudocode variables to registers)
```

Comentários na mesma linha do código devem ajudar a esclarecer o pseudocódigo

.text onde está as instruções

.data local da memória

Expressões lógicas e aritméticas

- Composicionalidade:
  - Técnica útil e poderosa: sintetiza o código, facilita entendimento, reuso.
  - Expressões **matemáticas** convencionais são **composicionais**!
    - Ex: **a*b+(c/d)...** -> (Como em linguagem C);
  - Instruções em **assembly não**!
    - **mult $t0 (addi $t1, $t1, 3)** ???? – Não permitido!
    - Precisa sequenciar as instruções...

Em assembly é necessário sequenciar as instruções e levando algumas considerações.

O Mips vai fazer 1 operação por vez.

Sequenciando Instruções:

- Como sequenciar instruções para resolver expressões lógicas e aritméticas?
- Considerações:
  - Número limitado de instruções
    - MIPS - opera dois registradores ou registrador e imediato
    - Nem toda instrução possui a variante com imediatos
    - Uso do **li** para carregar constantes (tb maiores que 16b)
  - Número limitado de registradores
    - Pode ser necessário ler e armazenar resultados intermediários na memória (comum em programas mais complexos);

Sequenciando Instruções

- Estratégia: Decompor a expressão.
  - Separar a expressão em pequenas sub-expressões.
    - Respeite agrupamentos (entre parênteses) e precedência de operadores (assumam a precedência vista em linguagem C);
  - Traduza cada sub-expressão e salve os resultados intermediários;
  - Combine os resultados!

Example

```
# Pseudocode:
#   d = (a+b) * (c+4)
# Register mappings:
#   a: t0, b: t1, c: t2, d: t3
add  $t4, $t0, $t1    # tmp1 = a+b
addi $t5, $t2, 4      # tmp2 = c+4
mul  $t3, $t4, $t5    # d = tmp1 * tmp2
```

```
tmp1 = a+b
tmp2 = c+4
d = tmp1 * tmp2
```


Quebrar as expressões matemáticas em pequenas expressões.

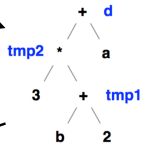
Salve cada operação em um operador temporário.

Sequenciando Instruções:

- Estratégia: Análise e tradução
  - Faça a análise sintática da expressão transformando-a em uma árvore de sintaxe abstrata
  - Atravesse a árvore em pós-ordem, armazenando o resultado das sub-árvores em registradores temporários
- Essa é a estratégia utilizada pelo compilador!

Example

```
# Pseudocode:
#   c = a + 3*(b+2)
# Register mappings:
#   a: t0, b: t1, c: t2
addi $t3, $t1, 2      # tmp1 = b+2
mul  $t4, $t3, 3      # tmp2 = 3*tmp1
add  $t2, $t0, $t4    # c = a + tmp2
```



Addi soma um registrador com um valor imediato.

add soma 2 regis

Otimizando o uso de registradores

- Geralmente é possível utilizar poucos registradores fazendo a acumulação do resultado.

```
# Pseudocode:
#   c = a + 3*(b+2)
# Register mappings:
#   a: $t0, b: $t1, c: $t2
#   tmp1: $t3, tmp2: $t4
```

```
# tmp1 = b+2
# tmp2 = 3*tmp1
# c = a + tmp2

addi $t3, $t1, 2
mul  $t4, $t3, 3
add  $t2, $t0, $t4
```

⇒

```
# c = b+2
# c = 3*c
# c = a + c

addi $t2, $t1, 2
mul  $t2, $t2, 3
add  $t2, $t0, $t2
```

tmp1 = 3+tmp1

A possiblidade de reaproveitar os temporários regis, portanto teriamos:

Exercício:

```
# Pseudocode:
#   d = a - 3 * (b + c + 8)
# Register mappings:
#   a: t0, b: t1, c: t2, d: t3
```

addi / muli/ subi --> terminador em i = imediato

```
addi $t3, $t2, 8     # d = b + c + 8
add  $t3, $t1, $t3
li   $t4, 3          # d = 3 * d
mul  $t3, $t4, $t3
sub  $t3, $t0, $t3   # d = a - d
```

load imediato --> addi $t4, $zero, 3

$at = temporário, mas não pode usar. Diretamente nao podemos usar ele, pois está ligado a fazer pseudo instruções do próprio compilador.

$at ele armazena o numero 3, para multiplicar, portanto se usar ele, vamos estar atualizando ele mesmo para 3 e multiplicando 3 = ERRO

A mul é uma pseudo instrução, assim ele vai usar o %at portanto nao dá de usa-lo.

$at sempre vai ser usado quando tiver uma pseudo instrução

pseudo instrução é quando o controlador não sabe decodificar a instrução e assim ele faz varias instruções para realizar aquela que desejada

Expressões lógicas

- Utilizadas em estruturas de controle:
  - branches (if-else);
  - loops (while, for);

> Logical expressions
> - values: True, False
> - boolean operators: not (`!`), and (`&&`), or (`||`)
> - relational operators: `==`, `!=`, `>`, `>=`, `<`, `<=`

- No MIPS, Falso = 0; Verdadeiro = 1;
- Operações lógicas não relacionais são **bit-a-bit**, não booleanas!

Usar "xori" para mudar apenas o numero 1 por 0.

00000000000000000000000000000001

xori -> 00000000000000000000000000000000

Caso usa-se a not como sempre usado em C, seria:

11111111111111111111111111111110

Instruções lógicas

- Operações de deslocamento (shift)
  - Deslocam todos os bits de uma palavra para esquerda ou direita, preenchendo os bits vazios com zero (não cíclico);
  - São instruções do tipo R (registrador)
  - A quantidade de bits a serem deslocados é especificada pelo campo shamt (shift amount)

Instruções lógicas

- **shift left logical (sll)**

```
sll  $t2, $s0, 4    # $t2 ← $s0 << 4
```

Conteúdo:

```
$s0   0110 1000 1111 0000 0111 0110 1111 1111
$t2   1000 1111 0000 0111 0110 1111 1111 0000
```


lembrando caso: 0001 → 0010 → 0100 → 1000 (2x, 2x, 2x)

Quero multiplicar um numero por 32 --> 2^5
--> Descolocar para esquerda 5 vezes.

Instruções lógicas

- **shift right logical (srl)**

```
srl  $t2, $s0, 8    # $t2 ← $s0 >> 8
```

Conteúdo:

```
$s0   0110 1000 1111 0000 0111 0110 1111 1111
$t2   0000 0000 0110 1000 1111 0000 0111 0110
```

Caso numero negativo, depois de deslocar ele vira positivo

Mesma ideia só que agora --> 1000 → 0100 (x2)

deslocar

Isto para inteiros!!!!!
Saber valor divisão

Saber resto → 3%2 → (0001)
10%4 = 2 (0010)

Instruções lógicas

- **shift right arithmetic (sra)**
  - Desloca bits à direita, preservando o sinal (compl. a 2)
  - Deslocamento de n bits corresponde à divisão por 2^n
  - Dica: para divisões com valores de tipo integer

```
sra  $t2, $s0, 4     # $t2 ← $s0 >> 4
```

Conteúdo:

```
$s0   1110 1000 1111 0000 0111 0110 1111 1111
$t2   1111 1110 1000 1111 0000 0111 0110 1111
```


Usar esse para divisão, pois preserva meu sinal

lembrando: Inverter sinais --> 10 complemento de 2

complemento de 2 --> 4 bits --> -8 a 7

01010 = 10 em compl 2

para inverter pega 10 inverte e +1 -->

10101
+1 --> 10110

Operações bit-a-bit

```
and  $t1, $t2, $t3    # $t1 = $t2 & $t3 (bitwise and)
or   $t1, $t2, $t3    # $t1 = $t2 | $t3 (bitwise or)
xor  $t1, $t2, $t3    # $t1 = $t2 ^ $t3 (bitwise xor)
```

Example: 0110 '**op**' 0011

and	or	xor
1010	1010	1010
0011	0011	0011
0010	1011	1001
1 iff both are 1	1 iff either is 1	1 iff exactly one 1

Immediate variants

```
andi $t1, $t2, 0x0F    # $t1 = $t2 & 0x0F (bitwise and)
ori  $t1, $t2, 0xF0    # $t1 = $t2 | 0xF0 (bitwise or)
xori $t1, $t2, 0xFF    # $t1 = $t2 ^ 0xFF (bitwise xor)
```


Lógica bit-a-bit vs. Lógica booleana

- Para **and**, **or**, **xor**:
  - **Equivalente** quando Falso=0 e Verdadeiro=1;
  - **Não equivalente** quando Falso=0 e Verdadeiro≠0 (como é em C);
- Not:
  - Macro fornecida pelo MARS: não equivalente ao "**not** lógico";
  - Inverte cada um dos bits:
    - Se Verdadeiro=1, not(Verdadeiro)=0xFFFFFFFE;

How can we implement logical not?

```
xori $t1, $t2, 1    # $t1 = not $t2 (logical not)
```


No mips para inverter (XORI)

Operadores relacionais

Logical expressions
- values: True, False
- boolean operators: not (!), and (&&), or (||)
- relational operators: ==, !=, >, >=, <, <=

```
seq  $t1, $t2, $t3    # $t1 = $t2 == $t3 ? 1 : 0
sne  $t1, $t2, $t3    # $t1 = $t2 != $t3 ? 1 : 0
sge  $t1, $t2, $t3    # $t1 = $t2 >= $t3 ? 1 : 0
sgt  $t1, $t2, $t3    # $t1 = $t2 >  $t3 ? 1 : 0
sle  $t1, $t2, $t3    # $t1 = $t2 <= $t3 ? 1 : 0
slt  $t1, $t2, $t3    # $t1 = $t2 <  $t3 ? 1 : 0

slti $t1, $t2, 42     # $t1 = $t2 <   42 ? 1 : 0
```


div --> divisão e joga pro reg o low e high -->

Low --> resultado
high --> resto

Exercício:

```
# Pseudocode:
#   c = (a < b) || ((a+b) == 10)
# Register mappings:
#   a: t0, b: t1, c: t2
```

```
add  $t3, $t0, $t1  # tmp = a+b
li   $t4, 10        # tmp = tmp == 10
seq  $t3, $t3, $t4
slt  $t2, $t0, $t1  # c = a < b
or   $t2, $t2, $t3  # c = c | tmp
```


Exercício:

```
# Pseudocode:
#   c = (a < b) && ((a+b) % 3) == 2
# Register mappings:
#   a: t0, b: t1, c: t2

#   tmp1: t3, tmp2: t4
```