Introdução à Arquitetura de Computadores

Aula17

Assembly 2: Instruções do µP MIPS

Linguagens de Alto-Nível e o *Assembly*

Instruções Lógicas e de Deslocamento (*Shift*)

- Lógicas: *and*, *or*, *xor*, *nor*
- Deslocamento (*Shift*)*:* Lógico e Aritmético

Constantes: 16 e 32 bits

Instruções de '*Salto*'

- Condicional (*beq, bne*) e incondicional (*j* e *jr*)

Controlo de fluxo de execução

- Fluxo condicional: *if, if-else*
- Ciclos iterativos: *while* e *for*

*A. Nunes da Cruz / **DETI - UA*** *Maio / 2021*

1 - Linguagens de Alto Nível (1)

Exemplos

- Python, Java, C
- Utilizam um nível de abstração mais elevado (i.e., cada *statement* é traduzido em múltiplas instruções *Assembly*).

Estruturas de *software* comuns:

- Execução condicional:
  - if e if-else
- Ciclos iterativos:
  - for
  - while
- *Arrays*: estrutura de dados contígua
- Funções: blocos de código reutilizáveis

instruções de alto nível chamam-se statements

Precisam de várias instruções para serem executados

Cuidado

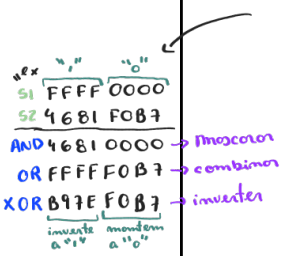
1 - Linguagens de Alto Nível e o Assembly MIPS

- As Linguagens de Alto-Nível são suportadas pelo MIPS através dum conjunto de instruções Lógicas, Aritméticas, de *Shift*, de *Salto* e de Invocação e de Retorno de função.
- Apresentamos sucintamente as instruções *Assembly* usadas na implementação de estruturas de Alto-Nível com valor semântico equivalente.
- Iniciamos com as instruções Lógicas e de deslocamento; Seguem-se as instruções de controlo do fluxo de execução (*if, if-else*) e ainda os ciclos de iteração (*while, for*).

2 - Instruções Lógicas (1): AND,OR, XOR, NOR

and, or, xor, nor (tipo-R)

- **and**: **mascara** bits
  Ex: mascarar todos os *bytes* exceto o menos significativo duma word :
  0xF234012F **and** 0x000000FF = 0x0000002F
- **or**: **combina** *bitfields*
  Ex: Combinar 0xF2340000 com 0x000012BC:
  0xF2340000 **or** 0x000012BC = 0xF23412BC
- **xor e nor**: **invertem** bits:
  Ex: Inverter todos os bits: A **nor** $0 = **not** A



andi, ori, xori (tipo-I)

- 16-bit imediato é *zero-extended*.
- a instrução **nori** não existe.

Na instrução **addi** (aritmética) a constante de 16-bits é sign-extended!

2 - Instruções Lógicas (2): Exemplo 1 - tipo-R

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
`and $s3,$s1,$s2`	$s3								
`or $s4,$s1,$s2`	$s4								
`xor $s5,$s1,$s2`	$s5								
`nor $s6,$s1,$s2`	$s6								

Estes tipo de exercícios fazem parte do guião do TP6.

2 - Instruções Lógicas (3): Exemplo 1 - tipo-R

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
→ `and $s3,$s1,$s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
→ `or $s4,$s1,$s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
→ `xor $s5,$s1,$s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
→ `nor $s6,$s1,$s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

and: **mascara** bits
0xFFFF0000 **and** 0x46A1F0B7
= 0x46A10000

xor: **inverte** com '1', **não-inverte** com '0'
0xFFFF0000 **xor** 0x46A1F0B7
= 0xB95EF0B7

or: **combina** bits
0xFFFF0000 **or** 0x46A1F0B7
= 0xFFFFF0B7

nor: **força** a '0' com '1', **inverte** com '0'
0xFFFF0000 **or** 0x46A1F0B7
= 0x00000F48

2 - Instruções Lógicas (4): Exemplo 2 - tipo-I

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code

```
andi $s2,$s1,0xFA34
ori  $s3,$s1,0xFA34
xori $s4,$s1,0xFA34
```

Result

$s2								
$s3								
$s4								

2 - Instruções Lógicas (5): Exemplo 2 - tipo-I

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code

```
andi $s2,$s1,0xFA34
ori  $s3,$s1,0xFA34
xori $s4,$s1,0xFA34
```

Result

$s2	0000	0000	0000	0000	0000	0000	0011	0100
$s3	0000	0000	0000	0000	1111	1010	1111	1111
$s4	0000	0000	0000	0000	1111	1010	1100	1011

Nas instruções lógicas: o valor imediato de 16-bits é *zero-extended* (não *sign-extended*)

3 - Instruções de Shift (1) - Valor de 'shift' Constante

sll: *shift left logical*

- Desloca à esquerda e preenche com zeros os bits à direita
- **sll** *i* bits = multiplicar por 2^i
- **Exemplo:** **sll** $t0, $t1, 5 # $t0 := $t1 << 5

srl: *shift right logical*

- Desloca à direita e preenche com zeros os bits à esquerda
- **srl** *i* bits = dividir por 2^i (operandos unsigned)
- **Exemplo:** **srl** $t0, $t1, 5 # $t0 := $t1 >>> 5

sra: *shift right arithmetic*

- *Shift* à direita e preenche com o bit de sinal os bits à esquerda
- **sra** *i* bits = dividir por 2^i (operandos signed)
- **Exemplo:** **sra** $t0, $t1, 5 # $t0 := $t1 >> 5

3 - Instruções de Shift (2) - Valor de 'shift' variável

sllv: *shift left logical variable*

- **Exemplo:** **sllv** $t0, $t1, $t2 # $t0 := $t1 << $t2

srlv: *shift right logical variable*

- **Exemplo:** **srlv** $t0, $t1, $t2 # $t0 := $t1 >>> $t2

srav: *shift right arithmetic variable*

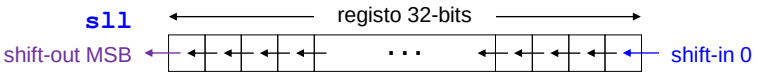
- **Exemplo**: **srav** $t0, $t1, $t2 # $t0 := $t1 >> $t2

3 - Shift (3) - Shift Left Logical (SLL) (<<)

Deslocar k bits à esquerda é equivalente a multiplicar um número por $\mathbf{2^k}$.

Exemplo:

```
ori  $t1,$0,4    # $t1 = 4
sll  $t1,$t1,3   # $t1 = $t1*2^3 = $t1*8 = 4*8 = 32
```

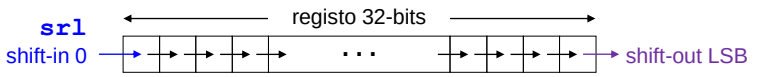



```
$t1 = 0b0000 0000 ... 0000 0100 = 4
após sll $t1,$t1,3
$t1 = 0b0000 0000 ... 0010 0000 = 32 = (4 *2^3)
```

violeta = bits ignorados; azul = bits admitidos

3 - Shift (4) - Shift Right Logical (SRL) (>>>)

srl comporta-se como **sll** mas desloca para direita em vez de para a esquerda. Corresponde a dividir por $\mathbf{2^K}$ mas só em UB (binário sem sinal).



Exemplo:

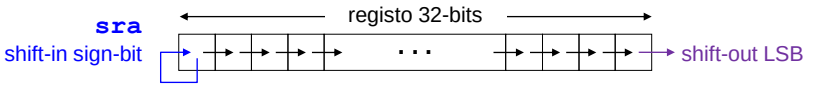
```
$t1 = 0b0000 0000 ... 0100 0000 = 0x0040  = 64
após srl  $t1,$t1,2
$t1 = 0b0000 0000 ... 0001 0000 = 0x0010  = 16
                                = 64*2^-2 = 64/4
```

azul = bits admitidos; violeta = bits ignorados

3 - Shift (5) - Shift Right Arithmetic (SRA) (>>) (1)

sra também desloca para direita, mas preserva o bit de sinal.

Deslocar e preservar o bit-de-sinal corresponde a dividir por $\mathbf{2^k}$ em **2C**.



Exemplo: -127 (em 2C):

-127_{10} = 0b1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 000 0001

Dividindo-o por 8 = $\mathbf{2^3}$ deveria dar -127/**8** = -15.875 ~=-16

Divisão inteira (resto desaparece)

Possível usando **sra** e deslocando 3 bits para a direita?

->>

3 - Shift (5) - Shift Right Arithmetic (SRA) (>>) (2)

Possível, -127/**8** ~= -16, usando **sra** deslocando 3 bits para a direita?

```
addi  $t1, $0, -127  # addi's imm is signed!
sra   $t1, $t1, 3
```

127/8 ≃ 15
-127/8 ≃ -16
Arredondamento é sempre para baixo

$t1 antes de ***sra*** :

-127_{10} = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 000 0001

= **0xFFFF FF81**

$t1 após o ***sra*** :

= 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0000

= **0xFFFF FFF0**= -16_{10}

O 2C de $0xFFFF\ FFF0_{16}$ é $0x0000\ 000F_{16}$ + 1 = $0x0000\ 0010_{16}$ = 16_{10}

*Em C/C++ o operador '>>' executa um *shift*-aritmético se a variável é um inteiro com sinal e um *shift*-lógico em inteiros sem sinal. Em Java é usado um operador distinto: '>>>' (srl).

3 - Instruções de Shift (6) - Exemplos de Codificação

Tabela B.2 - **tipo-R** (pg. 621/2)

Assembly Code

```
sxx  RD, RT, shamt
sll  $t0, $s1, 2
srl  $s2, $s1, 2
sra  $s3, $s1, 2
```

Tipo - R

Shift variable: sxxv RD, RT, RS

Field Values

op	rs	rt	rd	shamt	funct
0	0	17	8	2	0
0	0	17	18	2	2
0	0	17	19	2	3
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

shamt = *shift amount* (número de bits a deslocar)

4 - Uso de constantes (1) - 16Bits

- Constantes de 16-bits com **addi**:

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0,$0,0x4f3c
# $s0 = 0x00004f3c
```

A instrução nativa **addi** é útil para para carregar constantes com 16-bits num registo, quer sejam positivas quer sejam negativas!

Faz o sign-extended

Como o valor imediato da instrução **addi** tem sinal (2C), este é sempre estendido em sinal (*sign-extended*).

C Code

```
// int is a 32-bit signed word
int b = -0x8000; //-32768 (-2^15)
```

MIPS assembly code

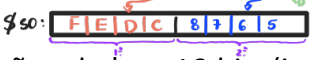
```
# $s0 = b
addi $s0,$0,-32768
# $s0 = 0xffff8000
```

É conveniente dizer algo sobre o uso de constantes (16- e 32-bits) em *Assembly*, antes de prosseguir com a descrição de mais tipos de instruções .

4 - Uso de constantes (2) - 32Bits

- Constantes de 32-bits requerem duas instruções: *load upper immediate* (**lui**) e **ori**:

```
C Code                  MIPS assembly code
                        # $s0 = a
int a = 0xFEDC8765;     lui $s0,0xFEDC      # $s0 = 0xFEDC0000
                        ori $s0,$s0,0x8765  # $s0 = 0xFEDC8765
```



- Quando o valor 'não cabe' em 16-bits (i.e., não é representável em 16-bits), é necessário usar duas instruções **lui** e **ori**; O MARS faz isso automagica/ através da pseudo-instrução **li** (*load immediate*).

Do ponto de vista do programador é obviamente mais cómodo usar as pseudo-instruções, todavia numa disciplina como IAC será necessário saber os detalhes (i.e., as instruções da máquina real) para compreender o funcionamento do *Datapath* do CPU.

5 - Instruções de 'Salto' (1) - Tipos

- Permitem a execução de código duma forma não-sequencial. (i.e., a instrução seguinte a ser executada não reside necessáriamente no endereço de memória igual a PC + 4)

Tipos de 'salto':

Condicional

- *branch if equal* (**beq**)
- *branch if not equal* (**bne**)

Incondicional

- *jump* (**j**)
- *jump register* (**jr**)
- *jump and link* (**jal**) <- próxima aula

5. Instruções de 'Salto'

5 - Instruções de 'Salto' Condicional (1) - beq

MIPS assembly - branch if equal

```
        addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
        addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
        sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
        beq     $s0, $s1, target    # branch is taken
        addi    $s1, $s1, 1         # not executed
        sub     $s1, $s1, $s0       # not executed
target:
        add     $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Os ***Labels*** (etiquetas) indicam o **endereço** de memória da instrução. Não podem ser usadas palavras reservadas (e.g., uma instrução) e devem ter o sufixo ':' (dois pontos).

5 - Instruções de 'Salto' Condicional (2) - bne

MIPS assembly - branch if not equal

```
        addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
        addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
        sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
        bne     $s0, $s1, target    # branch not taken
        addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
        sub     $s1, $s1, $s0       # $s1 = 5 - 4 = 1
target:
        add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Abordaremos a codificação das instruções *beq* e *bne* quando dermos os vários Modos de endereçamento, e.g., PC-Relativo e Pseudo-Directo.

5 - Instruções de 'Salto' Incondicional (1) - j

MIPS assembly - j(ump)

```
        addi    $s0, $0, 4          # $s0 = 4
        addi    $s1, $0, 1          # $s1 = 1
        j       target              # jump to target
        sra     $s1, $s1, 2         # not executed
        addi    $s1, $s1, 1         # not executed
        sub     $s1, $s1, $s0       # not executed
target:
        add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Salta e pronto. Sem condição

j é uma instrução do tipo-J.

5 - Instruções de 'Salto' Incondicional (2) - jr

MIPS assembly - j(ump) r(egister)

```
0x00002000      addi $s0, $0, 0x2010
0x00002004      jr   $s0
0x00002008      addi $s1, $0, 1
0x0000200C      sra  $s1, $s1, 2
0x00002010      lw   $s3, 44($s1)
```

→ Como $s0 tem 0x2010 a próx instrução será a 2010

jr é uma instrução do tipo-R.

6 - Controlo de Fluxo de Execução em Assembly (1)

Execução Condicional

- **`if`**
- **`if-else`**

Ciclos Iterativos

- **`while`**
- **`for`**

6 - Execução condicional - If (1) - ASM

C Code

se condição é verdadeira "ficas aqui"

```
if(i == j)
        f = g + h;
f = f - i;
```

C: A expressão condicional (f = g + h) é executada se a condição lógica (i == j) for verdadeira.

MIPS assembly code

Pensamos ao contrário

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3,$s4,nx  # if (i != j)
do: add $s0,$s1,$s2 # f = g + h
nx: sub $s0,$s0,$s3 # f = f - i
```

se for diferente "salta"

Asm: A condição lógica testada é a complementar (i != j). Isto conduz a uma codificação mais eficiente (i.e., menos instruções *Assembly*).

Fica mais fácil de seguir o código em assembly

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

6 - Execução condicional - If (2) - ASM Alternativa

C Code

```
if(i == j)
      f = g + h;
f = f - i;
```

C: A expressão condicional (f = g + h) é executada se a condição lógica (i == j) for verdadeira.

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3,$s4,nx  # if (i != j)
do: add $s0,$s1,$s2 # f = g + h
nx: sub $s0,$s0,$s3 # f = f - i

# Alternativa menos eficiente
    beq $s3,$s4,do  # if (i == j)
    j   nx          # +1 jump!
do: add $s0,$s1,$s2 # f = g + h
nx: sub $s0,$s0,$s3 # f = f - i
```

3 instruções

se forem iguais → se forem diferentes

4 instruções

Usa mais um *j* no final do *if* para saltar o bloco *do*.

6 - Execução condicional - If-else (1)

C Code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

6 - Execução condicional - If-else (2) - ASM

C Code

```
if (i == j)
   f = g + h;
else
   f = f - i;
```

Não pode executar se i==j

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, else
      add $s0, $s1, $s2
      j   done        <- Salta por cima do else
else: sub $s0, $s0, $s3
done:
```

Requere um *j* no final do *if* para saltar o bloco *else*.

7 - Ciclos Iterativos - While (1)

C Code

```
// determines the power of x
// such that 2^x = 128
int pow = 1;
int x   = 0;
   while (pow != 128) {
        pow = pow * 2;
        x = x + 1;
   }
```

Princípio é o mesmo, só contrário da condição salta

MIPS assembly code

```
# $s0 = pow, $s1 = x
```

O código *Assembly* dos ciclos de repetição é semelhante ao código dos *if* 's com um *j*ump para trás!

Conversão dum ciclo *while* num *if* com um salto para trás.

```
while ( i < j ){
   k++ ;
   i = i * 2 ;
}
```

```
W_LP: if ( i < j ){
        k++ ;
        i = i * 2 ;
        goto W_LP ;   <-
      }
```

7 - Ciclos Iterativos - While (2)

C Code

```
// determines the power of x
// such that 2^x = 128
int pow = 1;
int x   = 0;
   while (pow != 128) {
        pow = pow * 2;
        x = x + 1;
   }
```

MIPS assembly code

```
# $s0 = pow, $s1 = x
      addi $s0, $0, 1   # pow=1
      add  $s1, $0, $0  # x=0

      addi $t0, $0, 128
wh:   beq  $s0, $t0, done
      sll  $s0, $s0, 1  # pow*=2
      addi $s1, $s1, 1  # x+=1
      j    wh
done:
```

shift left logic multiplica por 2^i

beq → se forem iguais salta

j wh → testa a condição outra vez

- Na última vez saltamos para cima para depois salta para baixo, se alterarmos esse teste lógico para baixo fazemos um do...while

Assembly tests for the opposite case (pow == 128) of the C code (pow != 128).


```
do {
  (...)
} while(t0 = t1);
```

```
do:
 :
beq $t0, $t1, do
```

7 - Ciclos Iterativos - For (1)

for (*inicialização*; *condição*; *oper_iterativa*) **{**
  statement(s);
}

- *inicialização*: executada antes do *loop* começar
- *condição*: testada no início de cada iteração
- *statement(s)*: executado(s) sempre que a condição é satisfeita
- *oper_iterativa*: executada no final de cada iteração

O ciclo *for* é semelhante ao ciclo *while* com a vantagem de incluir uma variável de controlo do número de iterações.

7 - Ciclos Iterativos - For (2)

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

  for (i=0; i!=10; i = i+1){
     sum = sum + i;
  }
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```


7 - Ciclos Iterativos - For (3)

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

  for (i=0; i!=10; i = i+1){
     sum = sum + i;
  }
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
#
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1 # i++
        j    for
done:
```

Igual ao while mas tem a variável de controlo