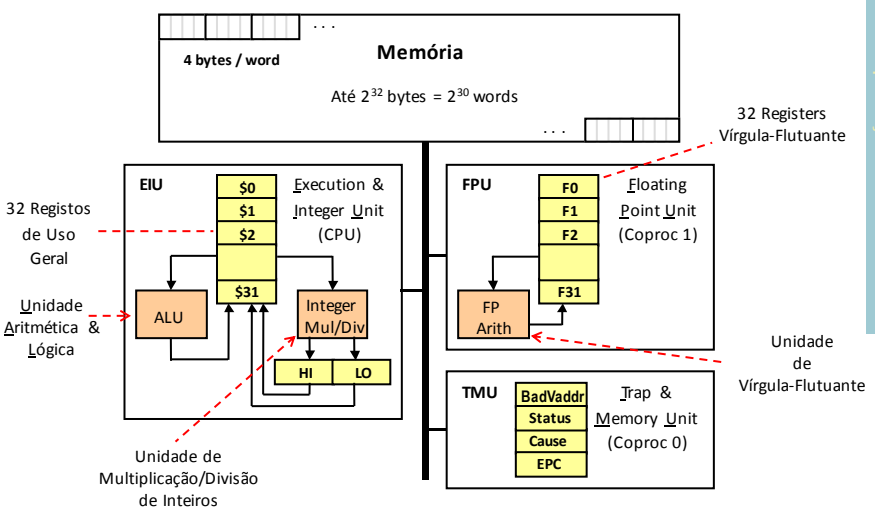
Introdução à Arquitetura de Computadores

Aula17

Assembly 2: Instruções do µP MIPS (cont)

- Mais Instruções
  - Multiplicação/Divisão de Inteiros
  - '*Saltos*' condicionais e incondicionais
- Estruturas de linguagens de Alto-Nível (C)
  - Controlo de fluxo de execução em ASM
    - Fluxo condicional: ***if, if-else***
    - Ciclos iterativos: ***while*** e ***for***

3 - Unidade de Multiplicação e Divisão (1) - µP MIPS



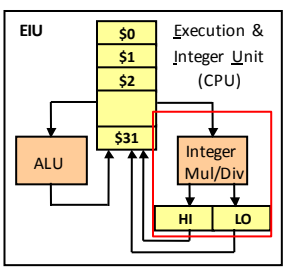
O CPU MIPS, para além da ALU e do Banco de Registos, inclui também uma unidade de Mul/Div de inteiros se 32bits.

3 - Unidade de Mult. e Divisão (2) - Registos LO e HI

- Registos especiais: LO(W), HI(GH)
- Multiplicação: 32 × 32, resultado: 64bits
  - **mult** $s0, $s1
    - Resultado de 64bits em {HI, LO}
- Divisão: 32-bits por 32-bits
  - **div** $s0, $s1 ; numa só instrução!
    - Quociente em LO
    - Resto em HI
- Instruções de acesso aos registos LO/HI:
  - **mflo** $s2 (move from lo to $s2)
  - **mfhi** $s3 (move from hi to $s3)

3 - Unidade de Mult. e Divisão (3) - Exemplos

Unidade de **MUL-DIV** de inteiros: residente no módulo de CPU



Registos auxiliares: **HI** e **LO**

Instruções:

mfhi e mflo (from)
mthi e mtlo (to)

Exemplo MULU:

```
Multiplicação (sem overflow):
        mulu $t1,$t2,$t3
Sets:
  HI to high-order 32bits,
  LO and $t1 to low-order 32bits
  of the product of $t2 and $t3.
```

Exemplo DIVU:

```
Divisão (sem overflow):
  divu $t1,$t2 # $t1/$t2
Sets:
  LO to quotient
  HI to remainder
  mflo $s0 # $s0 = quot = n/b
  mfhi $s1 # $s1 = rem  = n%b
```

3 - Instruções de 'Salto' (1) - Tipos

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

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

  Incondicional
  - jump (**j**)
  - jump register (**jr**)
  - jump and link (**jal**)

3 - 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:                              # label
  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).

3 - 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 de 'salto' beq e bne na próxima aula, quando dermos os vários Modos de endereçamento, e.g., PC-Relativo e Pseudo-Directo.

3 - 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
```

j é uma instrução do tipo-J.

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

```
# MIPS assembly - jump register
0x00002000      addi $s0, $0, 0x2010
0x00002004      jr   $s0
0x00002008      addi $s1, $0, 1
0x0000200C      sra  $s1, $s1, 2
0x00002010      lw   $s3, 44($s1)
```

jr é uma instrução do tipo-R.

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

- **Statements:**
  - **if**
  - **if-else**
- **Loops:**
  - **while**
  - **for**

4 - Execução condicional - If (1)

C Code

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

f = f - i;
```

MIPS assembly code

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


4 - Execução condicional - If (2) - ASM

C Code

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

f = f - i;
```

MIPS assembly code

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

Em C, a expressão (*statement*) condicional (f=g+h) é executada testando se a condição lógica (i==j) é verdadeira.
Em *Assembly* a condição lógica testada é a complementar (i!=j). Isto conduz a uma codificação mais eficiente (i.e., menos instruções *Assembly*).

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

4 - Execução condicional - If (3) – ASM Alternativas

C Code

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

f = f - i;
```

MIPS assembly code

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

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


4 - 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
```

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

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
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

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

4 - 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;
}
```

MIPS assembly code

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

O código *Assembly* dos ciclos de repetição é semelhante ao código dos ***if***'s com um **jump** 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 ;
      }
```

4 - 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;
}

// Convert while => if + goto
w_lp: if (pow != 128) {
        pow = pow * 2;
        x = x + 1;
        goto w_lp;
      }
```

MIPS assembly code

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

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

4 - Ciclos Iterativos - For (1)

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

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

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

4 - 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
```


4 - 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:
```

5 - Comparação: Set on Less Than (SLT) - (1)

Até aqui usamos só as instruções **beq** e **bne** para testar a igualdade ou a desigualdade e saltar para um dado *label*.

Existe ainda a instrução **slt** para comparar grandezas. A sintaxe e significado são:

```
slt   $r1, $r2, $r3   # $r1 = $r2 < $r3 ? 1 : 0
```

(**$r1** é **'1'** caso **$r2** < **$r3** e **'0'** caso contrário.)

A instrução **slt** é sempre seguida dum **beq** ou **bne** para testar o resultado e saltar.

De facto, as pseudo-instruções de salto **blt**, **bgt**, **ble**, **bge**, etc - são implementadas, pelo *assembler*, com recurso à instrução **slt**. Por exemplo, a instrução **bge** :

```
bge   $r1, $r2, LABEL   # jump to LABEL if $r1 >= $r2
```

É convertida da seguinte maneira:

```
slt   $r3, $r1, $r2     # $r3 = $r1 < $r2 ? 1 : 0
beq   $r3, $0, LABEL
```


5 - Comparação: Set on Less Than (SLT) - (2)

C Code

```
// add the powers of 2
// from 1 to 100
int sum = 0;
int i;

for (i=1;i < 101;i = i*2){
  sum = sum + i;
}
```

MIPS assembly code

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

5 - Comparação: Set on Less Than (SLT) - (3)

C Code

```
// add the powers of 2
// from 1 to 100
int sum = 0;
int i;

for (i=1;i < 101;i = i*2){
  sum = sum + i;
}
```

O *loop* termina quando i >=101

MIPS assembly code

```
# $s0 = i, $s1 = sum
      addi $s1,$0,0     # sum = 0
      addi $s0,$0,1     # i = 0
      addi $t0,$0,101   # $t0 = 101
     #bge $s0,$t0,done # pseudo-instr.
loop: slt  $t1,$s0,$t0 # $t1 =($s0<$t0)?1:0
      beq  $t1,$0,done # if($t1==0) done
      add  $s1,$s1,$s0 # sum = sum + i
      sll  $s0,$s0,1   # i = i*2
      j    loop
done:
```

$t1 = 1 if i < 101

Note-se que a instrução **slt** seguida do **beq** implementa a pseudo-instrução **bge**. De fato, no MARS podemos usar diretamente **bge** em vez de **slt** + **beq** !

5 - Set on Less Than Immediate (SLTI) - Tipo-I

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
       addi $s1, $0, 0
       addi $s0, $0, 1
       #addi $t0, $0, 101
loop:  slti $t1, $s0, 101
       beq  $t1, $0, done
       add  $s1, $s1, $s0
       sll  $s0, $s0, 1
       j    loop
done:
```

$t1 = 1 if i < 101

Para além da **slt** e **slti** existem ainda as variantes *unsigned*, **sltu** e **sltiu**, para comparar grandezas sem sinal!

Exemplo: **slti** $t1, $0, -1 e **sltiu** $t1, $0, -1, dão resultados diferentes, porquê?