Introdução à Arquitetura de Computadores

Aula16

Assembly 1: Instruções do µP MIPS

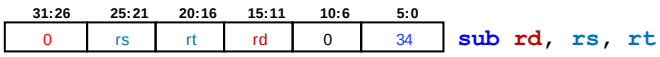
- Codificação de Instruções
  - Tipo-R, Tipo-I e Tipo-J
  - Exemplos: Instruções Aritméticas e LW/SW
- Programa em Memória
  - Inicialização e Execução
  - Interpretação do código máquina
- Instruções Lógicas e de Deslocamento (*Shift*)
  - Lógicas: AND, OR, XOR, NOR
  - Deslocamento: Lógico e Aritmético
- Constantes em *Assembly*
  - 16 e 32 bits

*A. Nunes da Cruz / **DETI - UA*** *Abril / 2018*

1 - Linguagem Máquina do µP MIPS

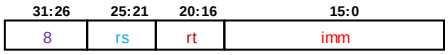
Codificação binária de instruções

- **Todas** as instruções têm 32-bits!
- Três formatos de instrução:
  - **Tipo-R:** dois operandos contidos em registos



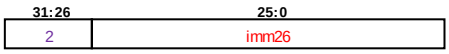
`sub rd, rs, rt`

  - **Tipo-I:** um dos operandos é uma constante



`addi rt,rs,imm`

  - **Tipo-J:** um dos operandos é um endereço



`j imm26`

Objetivos: Compreender o funcionamento do Datapath e a Unidade de Controlo do CPU.

1 - Instruções do Tipo-R(egister) (1)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- 3 Registos:
  - **rs, rt**: 2 registos fonte (**rs**=primeiro e **rt**=segundo) ou operandos
  - **rd**: 1 registo destino (ou resultado)
- Outros campos:
  - **op**: *opcode* ou *código de operação* (0 para instruções do Tipo-R)
  - ***funct***: a *função* que juntamente com o *opcode*, especifica a operação a ser executada
  - **shamt**: o *shift amount* - constante usada só nas instruções de *shift* (i.e., deslocamento de bits à esquerda ou à direita), nas outras instruções possue o valor 0.

Tipo-R: Instruções Aritméticas e Lógicas com 3-registos

1 - Instruções do Tipo-R (2) - Assembly vs Máquina (1)

Código Assembly

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Código Máquina

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Começamos com o tipo de instruções mais simples (isto é, aquelas cuja execução envolve só a interação entre o Banco de Registos e a Unidade Aritmética e Lógica (UAL ou ALU).

As instruções do tipo-R têm as seguintes particularidades:

1. O código de operação é zero sendo a operação a ser executada determinada pelo campo funct.

2. As instruções do tipo-R add e sub possuem um shamt igual a zero.

Em geral, a conversão das instruções Assembly para código Máquina é feita através da consulta de tabelas. Neste caso, precisamos duma tabela para os Registos: rs, rt e rd e outra tabela para o código de função, funct.

Exemplo: Codificação de Instruções Aritméticas tipo-R

1 - Instruções do Tipo-R (2) - Assembly vs Máquina (2)

Código Assembly

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Código Máquina

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tabela 6.1 - (pg. 300)

```
$s0 = 16, $s1 = 17, $s2 = 18
$t0 = 8,  $t3 = 11, $t5 = 13
```

Para todas as instruções, o valor dos registos é obtido por consulta da tabela 6.1.

Tabela B.2 - (pg. 622)

Funct	Name
100000 (32)	add rd, rs, rt
100001 (33)	addu rd, rs, rt
100010 (34)	sub rd, rs, rt

Para as instruções add e sub, o código de função é obtido da tabela B.2.
add=32 e sub=34.

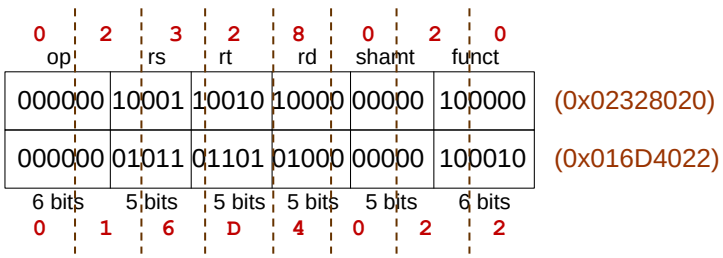
1 - Instruções do Tipo-R (2) - Assembly vs Máquina (3)

Código Assembly

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Código Máquina

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



(0x02328020)

(0x016D4022)

Em *Assembly*: **add rd, rs, rt**
Em *Máquina*: **<0> rs, rt, rd,<funct>**

1 - Instruções do Tipo-R (3) - Exercício Codificação (1)

Qual é o código máquina da seguinte instrução Assembly?

add $t0, $s4, $s5

Resposta:

1. Sabemos que **add** é uma instrução do tipo-R

2. Da Tabela 6.1 (pg. 300), tiramos:

$t0=**rd**=8, $s4=**rs**=20 e $s5=**rt**=21

onde **rd**=registo destino; **rs** = registo op1 e **rt** = registo op2

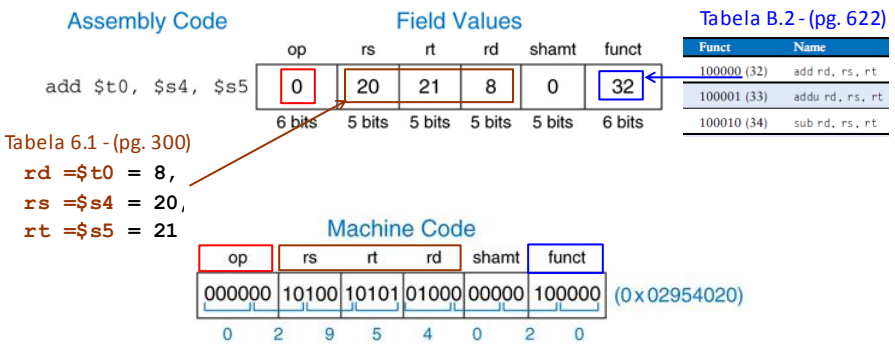
3. Da Tabela B.2 (pg. 622), o código de função de **add** é 32 (por ser do tipo-R, o código de operação é 0).

Convem escrever primeiro em binário e só depois em hexadecimal

1 - Instruções do Tipo-R (3) - Exercício Codificação (2)

Qual é o código máquina da seguinte instrução Assembly?

add $t0, $s4, $s5



Convem escrever em binário primeiro e só depois em hexadecimal

2 - Instruções do Tipo-I(mmediate) (1)

op	rs	rt	imm: constante ou endereço
6 bits	5 bits	5 bits	16 bits

- 2 registos:
  - **rs**: registo fonte
  - **rt**: registo destino para algumas (e.g., addi, lw) e fonte para outras (e.g., sw)
  - **imm**: valor imediato de 16-bits (em 2C, exceto nas lógicas)
    constante: -2^{15} a $+2^{15}-1$ (mas nas lógicas: 0 a $2^{16}-1$)
    endereço: *offset* adicionado ao endereço-base em **rs**
- Outros campos:
  - **op**: *opcode* ou *código de operação* (diferente de zero). (presente em todas as instruções do Tipo-I);
    a operação a ser executada é inteiramente determinada pelo *opcode* (só!).

Tipo-I: Usadas em instruções Aritméticas/Lógicas Imediatas e de *Load/Store*

2 - Instruções do Tipo-I (2) - Ex: Aritmética_i + lw/sw

Código Assembly

```
addi   $s0, $s1, 5
addi   $t0, $s3, -12
lw     $t2, 32($s0)
sw     $s1,  4($t1)
```

Valor dos Campos

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Código Máquina

	op	rs	rt	imm	
addi rt, rs, imm	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
addi rt, rs, imm	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
lw rt, imm(rs)	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
sw rt, imm(rs)	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	

2 - Instruções do Tipo-I (3) - Exercício Codificação (1)

Qual é o código máquina da seguinte instrução* Assembly?

lw $s3, -24($s4)

Resposta:

1. Sabemos que **lw** é uma instrução do tipo-I

2. Da Tabela 6.1 (pg. 300), tiramos $s3=**rt**=19 e $s4=**rs**=20, onde **rt**=registo destino; **rs** = registo (com o) endereço-base

3. Da Tabela B.1 (pg. 620), o código de operação para **lw** é 35

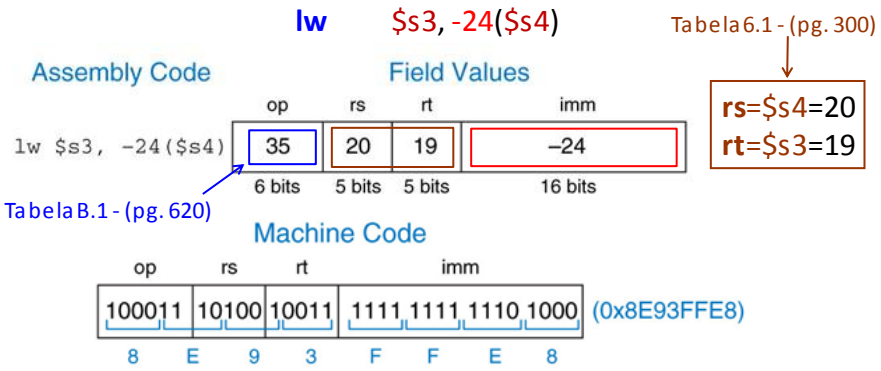
4. O valor imediato, -24, representa o *offset* (16-bits em 2C) a adicionar ao endereço-base (**rs**) para gerar o endereço efetivo.

*Esta instrução lê uma palavra de 32-bits (*word*) do endereço de memória dado por: "$s4 - 24" e coloca-a no registo $s3.

1 - Instruções do Tipo-I (3) - Exercício Codificação (2)

Qual é o código máquina da seguinte instrução Assembly?

lw $s3, -24($s4)

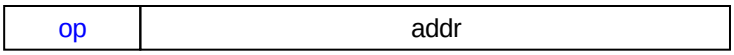


Valor imediato, de 16-bits, em 2C :

24_{10} = 0000 0000 0001 1000$_2$

-24_{10} = 1111 1111 1110 1000$_2$ = $FFE8_{16}$

3 - Instruções do Tipo-J(ump)



- 1 único operando:
  - **addr**: endereço com 26-bits
- Outros campos:
  - **op**: o *código de operação* da instrução **j**ump (2)

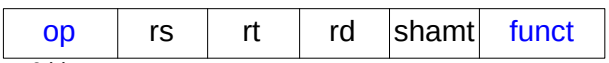
```
# MIPS assembly - j(ump)
   addi    $s0, $0, 4      # $s0 = 4
   addi    $s1, $0, 1      # $s1 = 1
   j       target          # jump to target
   sra     $s1, $s1, 2     # not executed
   addi    $s1, $s1, 1     # not executed
target:
   add     $s1, $s1, $s0   # $s1 = 1 + 4 = 5
```

próxima aula!

Tipo-J - Usadas em instruções do tipo j (jump) e jal (jump and link).

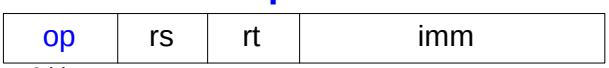
4 - Formato das Instruções do µP MIPS - Resumo

Tipo-R



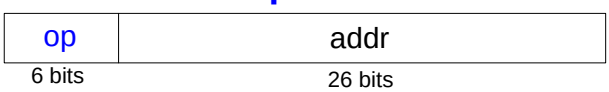
- **3 registos**: operandos (**rs, rt**) e resultado (**rd**)

Tipo-I



- **2 registos**: operando (**rs**) e resultado (**rt**) ; operando (**imm**)

Tipo-J



- um único operando (**addr**)

5 - Programa* em Memória (1)

- Programa em memória (o que é?)
  - Conjunto de instruções e dados
  - Diferença entre duas aplicações:
    a sequência de instruções e as estruturas de dados
- Execução de um novo programa
  - Não é necessário refazer ligações elétricas
  - Basta armazenar um novo programa na memória
- A execução do programa (como?)
  - O CPU lê instruções da memória (sequencial/).
  - Em seguida, descodifica a instrução e executa a operação associada.

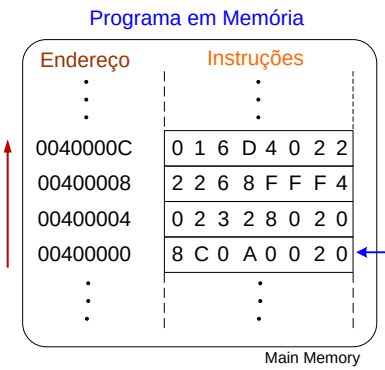
*Cria uma máquina cujo modo de funcionamento é (re)programável!

5 - Programa em Memória (2) - Inicialização

Código *Assembly*	Código *Máquina*
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

Tradução Assembly->Máquina:
Código *Assembly* e a respectiva tradução em código máquina.

Programa em Memória



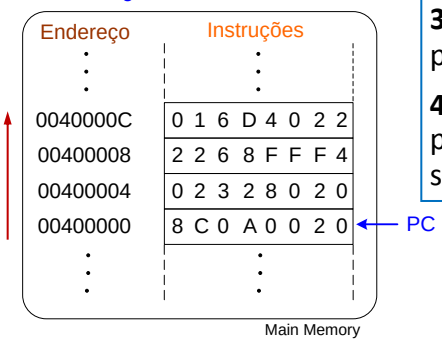
Programa em Memória:
O código máquina carregado em Memória (de Instruções).

Program Counter (PC):
Registo que aponta para instrução a ser executada (pelo CPU).

5 - Programa em Memória (3) - Início de Execução

Código *Assembly*			Código *Máquina*
`lw`	`$t2, 32($0)`		`0x8C0A0020`
`add`	`$s0, $s1, $s2`		`0x02328020`
`addi`	`$t0, $s3, -12`		`0x2268FFF4`
`sub`	`$t0, $t3, $t5`		`0x016D4022`

Programa em Memória



1. O código *Assembly* é traduzido em código máquina e carregado em memória.

2. O registo PC é inicializado para apontar para a 1ª instrução.

3. O CPU lê a instrução apontada pelo PC, descodifica-a e executa-a.

4. O valor do PC é incrementado para apontar para a instrução seguinte, i.e., PC = PC + 4.

5 - Prog em Mem (4) - Descodificação* da Instrução (1)

- Começamos com o *opcode* (6-bits mais signif.)
  - Se for igual a zero (0)
    É uma instrução do tipo-R
    Os 6 bits da função determinam a operação
  - Caso contrário
    É uma instrução do tipo-I ou do tipo-J
    O *opcode*, por si só, determina a operação

Exemplo:

Converter o código máquina seguinte em intruções *Assembly:*

0x02F34022
0x2237FFF1

*Interpretação ou DisAssembly. Processo inverso à codificação de ASM em Cód. Máquina.

5 - Prog em Mem (5) - Descodificação da Instrução (2)

As intruções *Assembly:*

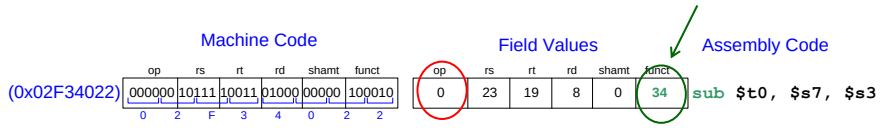
0x02F34022
0x2237FFF1

Consideremos a primeira instrução.

1. Verificamos que os 6bits mais significativos são iguais a zero. Logo estamos em presença duma instrução do **tipo-R**.

1. O *opcode* (6-bits MS)

Tabela B.2 - Tipo-R (pg. 621/2)



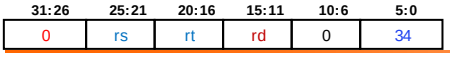
2. Sendo do tipo-R, precisamos de analisar o campo funct (6bits menos signif.) para determinar qual a operação a ser executada. A tabela B.2 dá-nos que a operação 34 corresponde a sub.

Funct	Name
100000 (32)	add rd, rs, rt
100001 (33)	addu rd, rs, rt
100010 (34)	sub rd, rs, rt

3. O valores dos registos rs,rt e rd é-nos dado pela tabela 6.1:

```
rs = $s7 = 23,
rt = $s3 = 19,
rd = $t0 = 8,
```

```
sub $t0, $s7, $s3
```




```
sub rd, rs, rt
```


5 - Prog em Mem (5) - Descodificação da Instrução (3)

As intruções *Assembly:*

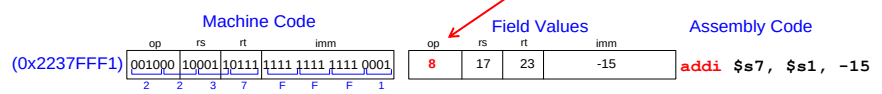
0x02F34022
0x2237FFF1

Consideremos a segunda instrução.

1. Verificamos que os 6bits mais significativos são diferentes de zero. Logo **não** estamos em presença duma instrução do tipo-R.

1. O *opcode* (6-bits MS)

Tabela B.1 - (pg. 620)



2. Consultando a tabela B.1, verificamos que a instrução com o *opcode* igual a 8 é addi.

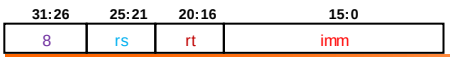
Opcode	Name	Description
001000 (8)	addi rt, rs, imm	add immediate

3. O valores dos registos rs,rt é-nos dado pela tabela 6.1:

```
rs = $s1 = 17,
rt = $s7 = 23,
```

4. O valor imediato (16 bits menos signif.) é uma constante em 2C:
   $0xFFF1_{16}$ que corresponde a $-0x000F_{16}$ ou -15_{10}.

```
addi $s7, $s1, -15
```




```
addi rt,rs,imm
```

6 - Programação - Linguagens de Alto Nível (1)

- Linguagens de Alto-Nível:
  e.g., Python, Java, C
  Escritas a um nível mais elevado de abstração

- Estruturas comuns em *software* de Alto-Nível:
  if/else *statements*
  for *loops*
  while *loops*
  function calls (invocação de funções)
  arrays (dados)

6 - Estruturas de Alto Nível e o Assembly MIPS

- Para implementar as estruturas usadas nas linguagens de Alto-Nível, o MIPS disponibiliza um conjunto de instruções Lógicas, Aritméticas, de *shift* e de salto.

- Faremos uma apresentação sucinta das instruções *Assembly* para entendermos a sua utilização na implementação de estruturas de Alto-Nível com o mesmo valor semantico.

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

- **and, or, xor, nor** (tipo-R)
  - **and**: útil para **mascarar** bits
    - Mascarar todos os bytes exceto o menos significativo duma word :

      0xF234012F **AND** 0x000000FF = 0x0000002F
  - **or**: útil para **combinar** *bit-fields* (subconjunto de bits)
    - Combine 0xF2340000 com 0x000012BC:

      0xF2340000 **OR** 0x000012BC = 0xF23412BC
  - **nor**: útil para **inverter** bits:
    - A **NOR** $0 = **NOT** A
- **andi, ori, xori** (tipo-I)
  - 16-bit imediato é *zero-extended* (não *sign-extended*)
  - a instrução **nori** não existe

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

6 - 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

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

Result

$s3								
$s4								
$s5								
$s6								

Estes tipo de exercícios fazem parte do guião do TP6.

6 - 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

6 - Instruções Lógicas (4): Exemplo 2 - Tipo-I

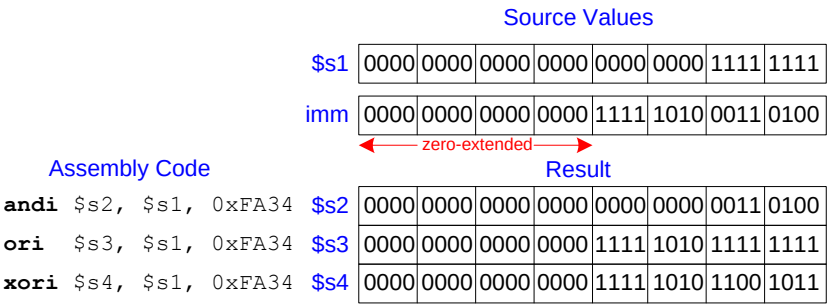
Source Values

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

zero-extended

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

6 - Instruções Lógicas (5): Exemplo 2 - Tipo-I



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

- **sll:** shift left logical
  - Deslocar à esquerda e preencher 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
  - Deslocar à direita e preencher 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 preencher 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

Estes tipo de exercícios fazem parte do guião do TP6.

6 - Instruções de Shift (2) - Valor de 'shift' Variável

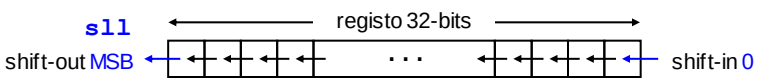
- sllv: shift left logical variable
  - **Exemplo: sllv** $t0, $t1, $t2 # $t0 <= $t1 << $t2
- srlv: shift right logical variable
  - **Exemplo: srlv** $t0, $t1, $t2 # $t0 <= $t1 >> $t2
- srav: shift right arithmetic variable
  - **Exemplo**: **srav** $t0, $t1, $t2 # $t0 <= $t1 >>> $t2

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

Deslocar k bits à esquerda é equivalente a multiplicar um número por 2^k.

Exemplo:

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



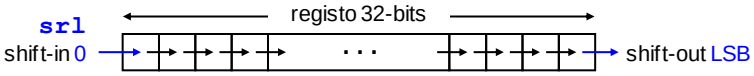
Exemplo:

```
      $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

6 - 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:

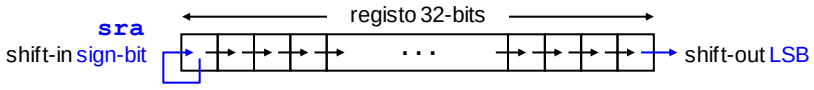
```
    $t1  = 0b0000 0000 ... 0100 0000 = 0x0040 = 64
após srl  $t1,$t1,2 :
    $ t1 = 0b0000 0000 ... 0001 0000 = 0x0010 = 16
                                     = 64*2^-2  = 64/4
```

azul = bits admitidos; violeta = bits ignorados

6 - Shift (5) - Shift Right Arithmetic (SRA) (>> *) (1)

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

Admitir o bit-de-sinal a partir da esquerda, corresponde a dividir por $\mathbf{2^k}$ em **2C**. (**2C** - Complemento a 2).



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 1 000 0001

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

Possível, esta divisão, usando a instrução **sra** deslocando 3 bits para a direita?

*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. O autor do livro usa '>>>' válido em Verilog (?).

6 - 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
```

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**

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 0000 = **0xFFFF FFF0** = -16_{10}

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

azul = bits admitidos; violeta = bits ignorados

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

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

Assembly Code / Field Values

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 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* (valor imediato a deslocar)

6 - Uso de constantes* (1) - 16Bits

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

```
C Code                                  MIPS assembly code
// int is a 32-bit signed word          # $s0 = a
int a = 0x4f3c;                        addi $s0, $0, 0x4f3c
                                        # $s0 = 0x00004f3c
```

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

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

```
C Code                                  MIPS assembly code
// int is a 32-bit signed word          # $s0 = b
int b = -0x8000; // -32768 (-2^15)      addi $s0,$0,-32768
                                        # $s0 = 0xffff8000
```

* Antes de continuar com a descrição de mais tipos de instruções é conveniente dizer algo sobre o uso de constantes (16- e 32-bits) em *Assembly*.

6 - Uso de constantes (2) - 32Bits

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

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

- A instrução **addi** é útil para para carregar constantes de 16-bits num registo, quer sejam positivas ou negativas.
- Mas quando o valor 'não cabe' em 16-bits, é necessário usar duas instruções **lui** e **ori**, embora o MARS o faça automagica/ por nós 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 da implementação das instruções na máquina real, para compreender o funcionamento do Datapath do CPU.