

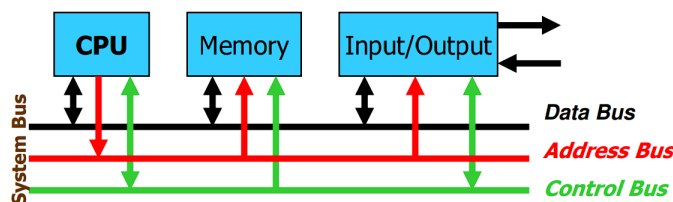
# Resumos AL-1

## A máquina e a sua linguagem

- Princípios básicos dos computadores atuais:
  - As instruções são representadas da mesma forma que os números
  - Os programas são armazenados em memória, para serem lidos e escritos, tal como os números
- Estes princípios formam os fundamentos do conceito da arquitetura "stored-program"
  - O conceito "stored-program" implica que na memória possa residir, ao mesmo tempo, informação de natureza tão variada como: o código fonte de um programa em C, um editor de texto, um compilador, e o próprio programa resultante da compilação.

## Arquitetura básica de um sistema computacional

- Unidades fundamentais que constituem um computador
  - CPU – responsável pelo processamento da informação através da execução de uma sequência de instruções (programa) armazenadas em memória
  - Memória – responsável pelo armazenamento de:
    - Programas
    - Dados para processamento
    - Resultados
  - Unidades de I/O – responsáveis pela comunicação com o exterior
    - Unidades de entrada – permitem a receção de informação vinda do exterior (dados, programas) e que é armazenada em memória
    - Unidades de saída – permitem o envio de resultados para o exterior
- Um computador é um sistema digital complexo
- Modelo de von Neumann



- Data Bus: barramento de transferência de informação (CPU ↔ memória, CPU ↔ Input/Output)
  - Address Bus: identifica a origem/destino da informação (na memória ou nas unidades de input/output)
  - Control Bus: sinais de protocolo que especificam o modo como a transferência de informação deve ser feita
- Endereço (address) – um número (único) que identifica cada registo de memória. Os endereços são contados sequencialmente, começando em 0 □ Exemplo: o conteúdo da posição de memória 0x2000 é 0x32 – (0x2000 é o endereço, 0x32 o valor armazenado)
  - Espaço de endereçamento (address space) – a gama total de endereços que o CPU consegue referenciar (depende da dimensão do barramento de endereços). Exemplo: um CPU com um barramento de endereços de 16 bits pode gerar endereços na gama: 0x0000 a 0xFFFF (i.e., 0 a  $2^{16}-1$ )

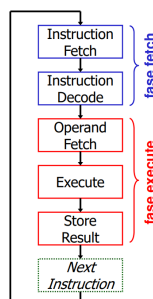
## Arquitetura básica do CPU

- Secção de dados (datapath) – elementos operativos/funcionais para encaminhamento, processamento e armazenamento de informação
  - Multiplexers
  - Unidade Aritmética e Lógica (ALU) – Add, Sub, And, Or...

- Registos internos
- **Unidade de controlo** – responsável pela coordenação dos elementos do datapath, durante a execução de um programa
  - Gera os sinais de controlo que adequam a operação de cada um dos recursos da secção de dados às necessidades da instrução que estiver a ser executada
  - Dependendo da arquitetura, pode ser uma máquina de estados ou um elemento meramente combinatória
- Independentemente da Unidade de Controlo ser combinatória ou sequencial, **o CPU é sempre uma máquina de estados síncrona**

### *Ciclo-base de execução de uma instrução*

- **Instruction fetch**: leitura do código máquina da instrução (instrução reside em memória)
- **Instruction decode**: decodificação da instrução pela unidade de controlo
- **Operand fetch**: leitura do(s) operando(s)
- **Execute**: execução da operação especificada pela instrução
- **Store result**: armazenamento do resultado da operação no destino especificado na instrução



### *Arquitetura do Conjunto de Instruções (ISA)*

- **Instruction Set**: coleção de todas as operações/instruções que o processador pode executar
- **Instruction Set Architecture (ISA)**
  - ... os atributos de um sistema computacional tal como são vistos pelo programador, i.e. a estrutura concetual e o comportamento funcional, de forma distinta e independente da organização do fluxo de informação e dos respetivos elementos de controlo, do desenho lógico e da implementação física. – Amdahl, Blaaw, and Brooks, 1964
- Arquitetura de Computadores =

### **Arquitetura do Conjunto de Instruções (ISA) + Organização da Máquina**

- Também designada por "modelo de programação"
- Descreve tudo o que o programador necessita de saber para programar corretamente, em assembly, um determinado processador
- Uma importante abstração que representa a **interface entre o nível mais básico de software e o hardware**
- Descreve a funcionalidade, de forma independente do hardware que a implementa. Pode assim falar-se de "arquitetura" e "implementação de uma arquitetura"

### *Classes de instruções*

- Um dada arquitetura pode ter um ISA com centenas de instruções
- É possível, no entanto, considerar a existência de um grupo limitado de classes de instruções comuns à generalidade das arquiteturas
- Classes de instruções:
  - Processamento
    - Aritméticas e lógicas
  - Transferência de informação

- Cópia entre registos internos e entre registos internos e memória
- Controlo de fluxo de execução
  - Alteração da sequência de execução (estruturas condicionais, ciclos, chamadas a funções,...)

### *Instruções e implementação hardware*

- No projeto de um processador a definição do **instruction set** exige um delicado compromisso entre múltiplos aspetos, nomeadamente:
  - as facilidades oferecidas aos programadores (por ex. instruções de manipulação de strings )
  - a complexidade do hardware envolvido na sua implementação
- Quatro princípios básicos estão subjacentes a um bom design ao nível do hardware:
  - A regularidade favorece a simplicidade
  - Quanto mais pequeno mais rápido
  - O que é mais comum deve ser mais rápido
  - Um bom design implica compromissos adequados

### *ISA – formato e codificação das instruções*

- Codificação das instruções com um número de bits variável
  - Código mais pequeno
  - Maior flexibilidade
  - Instruction fetch em vários passos
- Codificação das instruções com um número de bits fixo
  - Instruction fetch e decode mais simples
  - Mais simples de implementar em pipeline

### *ISA – número de registos internos do CPU*

- Vantagens de um número pequeno de registos
  - Menos hardware
  - Acesso mais rápido
  - Menos bits para identificação do registo
  - Mudança de contexto mais rápida
- Vantagens de um número elevado de registos
  - Menos acessos à memória
  - Algumas variáveis dos programas podem residir em registos
  - Certos registos podem ter restrições de utilização

### *ISA – localização dos operandos das instruções*

- Arquiteturas baseadas em **acumulador**
  - Resultado das operações é armazenado num registo especial designado de acumulador
    - `add a           # acc ← acc + a`
- Arquiteturas baseadas em **Stack**
  - Operandos e resultado armazenados numa stack (pilha) de registos
    - `add           # tos ← tos + next (tos = top of stack)`
- Arquiteturas **Register-Memory**
  - Operandos das instruções aritméticas e lógicas residem em registos internos do CPU ou em memória
    - `load r1, [a] # r1 ← mem[a]`
    - `add r1, [b] # r1 ← r1 + mem[b]`
    - `store [c], r1 # mem[c] ← r1`
- Arquiteturas **Load-store**
  - Operandos das instruções aritméticas e lógicas residem em registos internos do CPU de uso geral (mas nunca na memória).
    - `load r1, [a] # r1 ← mem[a]`

- `load r2, [b] # r2 ← mem[b]`
- `add r3, r1, r2 # r3 ← r1 + r2`
- `store [c], r3 # mem[c] ← r3`

### Aspetos chave da arquitetura MIPS

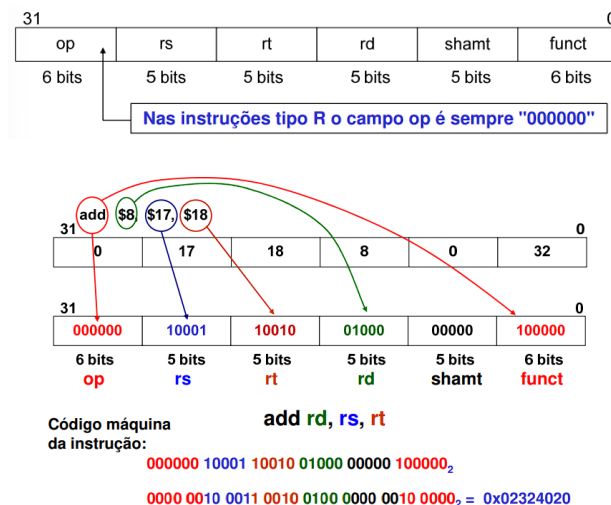
- 32 Registos de uso geral, de 32 bits cada (1 word  $\leftrightarrow$  32 bits)
- ISA baseado em **instruções de dimensão fixa** (32 bits)
- Arquitetura **load-store** (register-register operation)
- Memória organizada em bytes (memória byte addressable)
- Espaço de endereçamento de 32 bits (232 endereços possíveis, i.e. máximo de 4 GB de memória)
- Barramento de dados externo de 32 bits

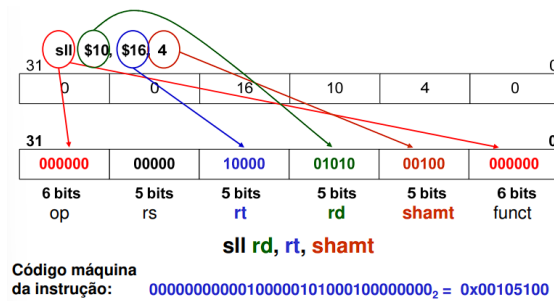
### Os registos internos do MIPS

- Em assembly são, normalmente, usados nomes alternativos para os registos (nomes virtuais):
  - `$zero` (`$0`)
  - `$at` (`$1`)
  - `$v0` e `$v1` (`$2` e `$3`)
  - `$a0` a `$a3`
  - `$t0` a `$t9`
  - `$s0` a `$s7`
  - `$sp` (`$29`)
  - `$ra` (`$31`)
- Registo `$0` tem sempre o valor `0x00000000` (apenas pode ser lido)

### Codificação de instruções no MIPS – formato R

- O formato R é um dos três formatos de codificação de instruções no MIPS
- Campos da instrução:
  - **op**: opcode (é sempre zero nas instruções tipo R)
  - **rs**: Endereço do registo que contém o 1º operando fonte
  - **rt**: Endereço do registo que contém o 2º operando fonte
  - **rd**: Endereço do registo onde o resultado vai ser armazenado
  - **shamt**: **shift amount** (útil apenas em instruções de deslocamento)
  - **funct**: código da operação a realizar





### Instruções de transferência entre registos internos

- Transferência entre registos internos:  $R_{dst} = R_{src}$
- Registo  $\$0$  do MIPS tem sempre o valor  $0x00000000$  (apenas pode ser lido)
- Utilizando o registo  $\$0$  e a instrução lógica OR é possível realizar uma operação de transferência entre registos internos:
  - `or Rdst, Rsrc, $0`       $\# R_{dst} = (R_{src} \mid 0) = R_{src}$
  - Exemplo: `or $t1, $t2, $0`       $\# \$t1 = \$t2$
- Para esta operação é habitualmente usada uma **instrução virtual** que melhora a legibilidade dos programas - "move".
- No processo de geração do código máquina, o assembler substitui essa instrução pela instrução nativa anterior:
  - `move Rdst, Rsrc`       $\# R_{dst} = R_{src}$
  - Exemplo: `move $t1, $t2`       $\# \$t1 = \$t2$  (or  $\$t1, \$t2, \$0$ )

### Instruções de controlo de fluxo de execução

- As instruções que permitem a tomada de decisões pertencem à classe "controlo de fluxo de execução"
- No MIPS as instruções básicas de controlo de fluxo de execução são:

`beq Rsrc1, Rsrc2, Label` # branch if equal

`bne Rsrc1, Rsrc2, Label` # branch if not equal

e são conhecidas como "**branches**" (saltos / jumps) **condicionais**

- O endereço para onde o salto é efetuado (no caso de a condição ser verdadeira) designa-se por **endereço-alvo** da instrução de branch (**branch target address**)
- O ISA do MIPS suporta ainda um conjunto de instruções que **comparam diretamente com zero**:
  - `bltz Rsrc, Label` # Branch if  $R_{src} < 0$
  - `blez Rsrc, Label` # Branch if  $R_{src} \leq 0$
  - `bgtz Rsrc, Label` # Branch if  $R_{src} > 0$
  - `bgez Rsrc, Label` # Branch if  $R_{src} \geq 0$
- Nestas instruções o **registo  $\$0$  está implícito** como o segundo registo a comparar

### Instrução SLT

Para além das instruções de salto com base no critério de igualdade e desigualdade, o MIPS suporta ainda a instrução:

```
slt Rdst, Rsrc1, Rsrc2  # slt = "set if less than"
```

```
# set Rdst if Rsrc1 < Rsrc2
```

**Descrição:** O registo "Rdst" toma o valor "1" se o conteúdo do registo "Rsrc1" for inferior ao do registo "Rsrc2". Caso contrário toma o valor "0".

```
slti Rdst, Rsrc1, Imm  # slt = "set if less than"
```

```
# set Rdst if Rsrc1 < Imm
```

A utilização das instruções "bne", "beq", "slt" e "slti", em conjunto com o registo \$0, permite a implementação de todas as condições de comparação entre dois registos e também entre um registo e uma constante: (A = B), (A ≠ B), (A > B), (A ≥ B), (A < B), (A ≤ B)

### Instruções virtuais de branch do MIPS

- Nos programas Assembly, podem ser utilizadas instruções de salto não diretamente suportadas pelo MIPS (instruções virtuais), mas que são decompostas pelo assembler em instruções nativas. Essas instruções são:
  - blt Rsrc1, Rsrc2, Label # Branch if Rsrc1 < Rsrc2
  - ble Rsrc1, Rsrc2, Label # Branch if Rsrc ≤ Rsrc2
  - bgt Rsrc1, Rsrc2, Label # Branch if Rsrc > Rsrc2
  - bge Rsrc1, Rsrc2, Label # Branch if Rsrc ≥ Rsrc2

### Decomposição das instruções virtuais BGT e BGE (é costume sair no teste)

A instrução virtual "bge" (branch if greater or equal than):

```
bge $4, $7, exit      # if $4 ≥ $7 goto exit  
  
# (i.e. goto exit if !($4 < $7))
```

É decomposta nas instruções nativas:

```
slt $1, $4, $7        # $1 = 1 if $4 < $7 ($1=0 if $4 ≥ $7)  
  
beq $1, $0, exit      # if $1 = 0 goto exit
```

De modo similar, a instrução virtual "bgt" (branch if greater than):

```
bgt $4, $7, exit      # if $4 > $7 goto exit  
  
# (i.e. goto exit if $7 < $4 )
```

É decomposta nas instruções nativas:

```
slt $1, $7, $4        # $1 = 1 if $7 < $4 ($1=1 if $4 > $7)  
  
bne $1, $0, exit      # if $1 ≠ 0 goto exit
```

## Armazenamento de informação – registos internos

- Nos exemplos das aulas anteriores apenas se fez uso de registos internos do CPU para o armazenamento de informação (variáveis):

add \$8, \$17, \$18

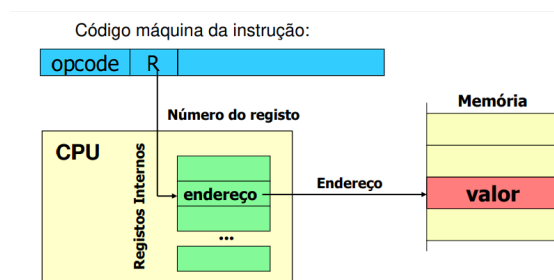
add \$9, \$19, \$20

- E se a informação a processar residir na memória externa (por exemplo um array de inteiros) ?
- Recorde-se que a arquitetura MIPS é do tipo **load-store**, ou seja, não é possível operar diretamente sobre o conteúdo da memória externa
- Terão que existir instruções para transferir informação entre os registos do CPU e a memória externa

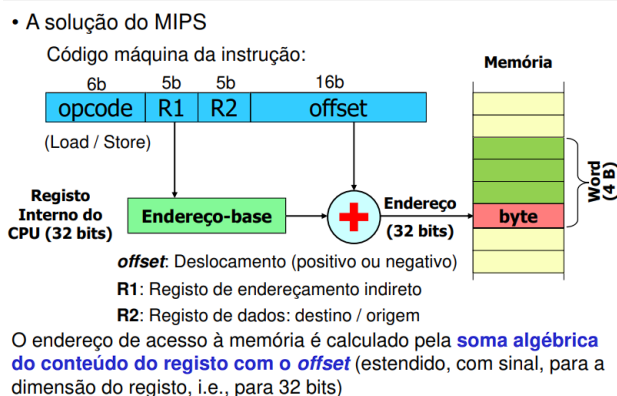
## Modos de endereçamento

- O **método** usado pela arquitetura para **aceder ao elemento que contém a informação** que irá ser processada por uma dada instrução é genericamente designado por “**Modo de Endereçamento**”
- Nas instruções aritméticas e lógicas (codificadas com o formato R), os operandos residem ambos em registos internos
  - Este modo é designado por **endereçamento tipo registo**

## Endereçamento indireto por registo

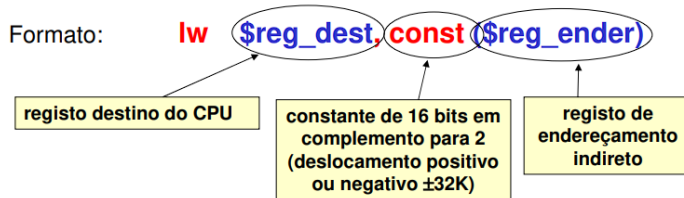


## Endereçamento indireto por registo com deslocamento



## Leitura da memória – instrução LW

- LW - (load word)** transfere uma palavra de 32 bits da memória para um registo interno do CPU (**1 word é armazenada em 4 posições de memória consecutivas**)

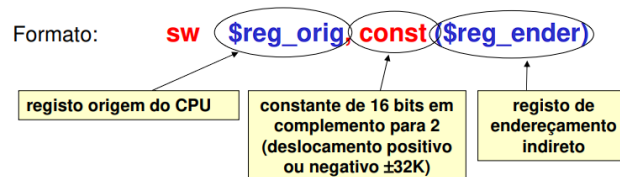


Exemplo:

**lw \$5, 4 (\$2)** # copia para o registo \$5 a *word* de 32 bits armazenada  
# a partir do endereço de memória calculado como:  
# **addr = (conteúdo do registo \$2) + 4**

*Escrita na memória – instrução SW*

**SW - (store word)** transfere uma palavra de 32 bits de um registo interno do CPU para a memória (**1 word é armazenada em 4 posições de memória consecutivas**)

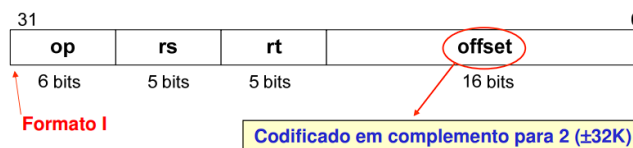


Exemplo:

**sw \$7, -8 (\$4)** # copia a *word* armazenada no registo \$7 para a  
# memória, a partir do endereço calculado como:  
# **addr = (conteúdo do registo \$4) - 8**

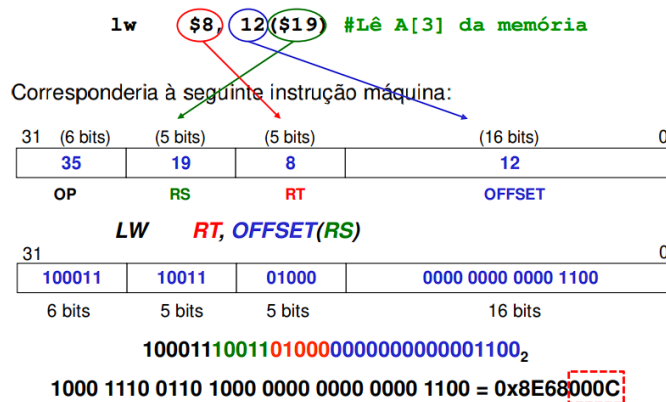
*Codificação das instruções de acesso à memória no MIPS*

A necessidade de codificação de uma constante de 16 bits, obriga à definição de um novo formato de codificação, o **formato I**



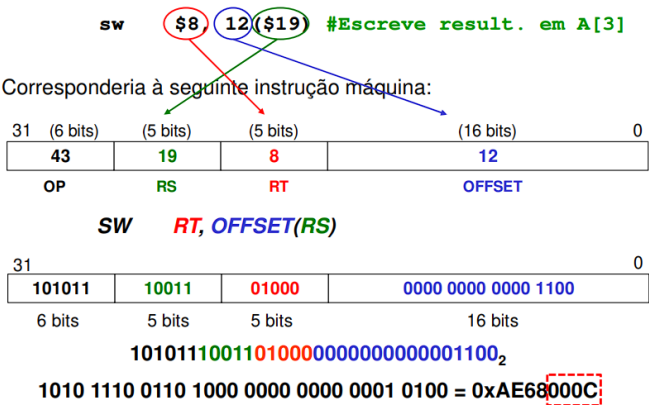
- Gama de representação da constante de 16 bits
  - [-32768, +32767]

*Codificação da instrução LW (Load Word)*





Codificação da instrução SW (Store Word)



Exemplo de codificação

• O seguinte trecho de código assembly:

```
lw $8, 12($19)      # Lê A[3] da memória
add $8, $18, $8      # Calcula novo valor
sw $8, 12($19)      # Escreve resultado em A[3]
```

Corresponde à codificação:

31						0
0x23	0x13	0x08	0x000C			Formato I
0x00	0x12	0x08	0x08	0x00	0x20	Formato R
0x2B	0x13	0x08	0x000C			Formato I

• Resultando no código máquina:

100011100110100000000000001100<sub>2</sub> = 0x8E68000C

00000010010010000100000000100000<sub>2</sub> = 0x02484020

101011100110100000000000001100<sub>2</sub> = 0xAE68000C

Restrições de alinhamento nas instruções LW e SW

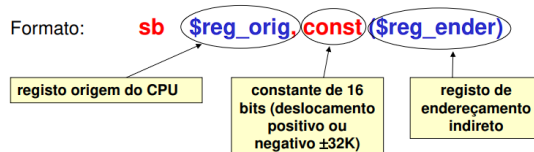
- Externamente o barramento de endereços do MIPS só tem disponíveis 30 dos 32 bits: A31...A 2. Ou seja, qualquer combinação nos bits A1 e A0 é ignorada no barramento de endereços exterior.
- Assim, do ponto de vista externo, só são gerados endereços **múltiplos de 2^2 = 4** (ex: ...0000, ...0100, , ...1000, ...1100 )

O acesso a words só é possível em endereços múltiplos de 4

- Se, numa instrução de leitura/escrita de uma word, for calculado um endereço não múltiplo de 4, quando o MIPS tenta aceder à memória a esse endereço verifica que o endereço é inválido e gera uma exceção, terminando aí a execução do programa
- Como se evita o problema ?
  - Garantindo que as variáveis do tipo word estão armazenadas num endereço múltiplo de 4
- Diretiva .align n do assembler (força o alinhamento do endereço de uma variável num valor múltiplo de 2^n)

Instrução de escrita de 1 byte na memória - SB

- SB - (store byte) transfere um byte de um registo interno para a memória – só são usados os 8 bits menos significativos

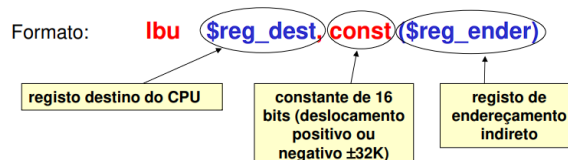


Exemplo:

**sb \$7, 5 (\$4)** # transfere o **byte** armazenado no registro \$7 (8 bits menos significativos) para o endereço de memória calculado como:  
# **addr = (conteúdo do registro \$4) + 5**

### Instrução de leitura de 1 byte na memória - LBU

- **LBU - (load byte unsigned)** transfere um byte da memória para um registro interno - **os 24 bits mais significativos do registro destino são colocados a 0**

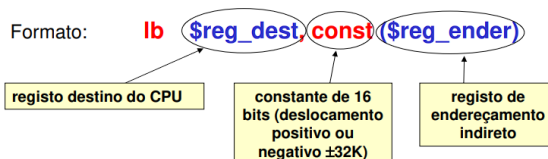


Exemplo:

**lbu \$5, -3 (\$2)** # transfere para o registro \$5 o **byte** armazenado no endereço de memória calculado como:  
# **addr = (conteúdo do registro \$2) - 3**  
# os 24 bits mais significativos de \$5 são colocados a zero

### Instrução de leitura de 1 byte na memória - LB

- **LB - (load byte)** transfere um byte da memória para um registro interno, **fazendo extensão de sinal do valor lido de 8 para 32 bits**



Exemplo:

**lb \$5, 0 (\$2)** # transfere para o registro \$5 o **byte** armazenado no endereço de memória calculado como:  
# **addr = (conteúdo do registro \$2) + 0**  
# o bit mais significativo do **byte** transferido é replicado nos 24 bits mais significativos de \$5

Na leitura/escrita de 1 byte de informação o problema do alinhamento, do ponto de vista do programador, não se coloca.

### Organização das words de 32 bits na memória

- A memória no MIPS está organizada em bytes (byte-addressable memory)
- Se a quantidade a armazenar tiver uma dimensão superior a 8 bits vão ser necessárias várias posições de memória consecutivas (por exemplo, para uma word de 32 bits são necessárias 4 posições de memória)
- Exemplo: 0x012387A5 (4 bytes: 01 23 87 A5 )
- Qual a ordem de armazenamento dos bytes na memória? Duas alternativas:
  - byte mais significativo armazenado no endereço mais baixo da memória (**big-endian**)
  - byte menos significativo armazenado no endereço mais baixo da memória (**little-endian**)

- Exemplo: **0x012387A5** (**0x01 23 87 A5**)

Address	Data		Address	Data	
0x10010008	?		0x10010008	?	
0x10010009	?		0x10010009	?	
0x1001000A	?		0x1001000A	?	
0x1001000B	?		0x1001000B	?	
0x1001000C	0x01	↓	0x1001000C	0xA5	↑
0x1001000D	0x23		0x1001000D	0x87	
0x1001000E	0x87		0x1001000E	0x23	
0x1001000F	0xA5		0x1001000F	0x01	
0x10010010	?		0x10010010	?	
0x10010011	?		0x10010011	?	
0x10010012	?		0x10010012	?	
0x10010013	?		0x10010013	?	
0x10010014	?		0x10010014	?	
...	...		...	...	

Big-Endian

Address	Data
0x10010008	?
0x10010009	?
0x1001000A	?
0x1001000B	?
0x1001000C	0xA5
0x1001000D	0x87
0x1001000E	0x23
0x1001000F	0x01
0x10010010	?
0x10010011	?
0x10010012	?
0x10010013	?
0x10010014	?
...	...

Little-Endian

Big-Endian

Little-Endian

- O simulador MARS implementa "little-endian"

### Diretivas do Assembler

- Diretivas são comandos especiais colocados num programa em linguagem assembly destinados a instruir o assembler a executar uma determinada tarefa ou função
- Diretivas **não são instruções** da linguagem assembly (não fazem parte do ISA), não gerando qualquer código máquina
- As diretivas podem ser usadas com diversas finalidades:
  - reservar e inicializar espaço em memória para variáveis
  - controlar os endereços reservados para variáveis em memória
  - especificar os endereços de colocação de código e dados na memória
  - definir valores simbólicos

### NOTA:

- Utilizar a diretiva ".align" sempre que se pretender que o endereço subsequente esteja alinhado
- A diretiva ".word" alinha automaticamente num endereço múltiplo de 4

### Linguagem C: ponteiros e endereços – o operador &

Um ponteiro é uma variável que contém o endereço de outra variável – o acesso à 2ª variável pode fazer-se indiretamente através do ponteiro • Se var é uma variável, então &var dá-nos o seu endereço

### Ponteiros e endereços – o operador \*

O operador " \* ":

- trata o seu operando como um endereço
- permite o acesso ao endereço para obter ou alterar o respetivo conteúdo
- Exemplo:

px = &x;                      // px é um ponteiro para x

y = \*px;                      // \*px é o valor de x

Atribui a y o mesmo valor que a expressão: y = x;

- O operador " \* ", é designado por operador de indireção

### Acesso sequencial a elementos de um array

## Acesso indexado

- endereço do elemento a aceder = endereço inicial do array + (índice \* dimensão em bytes de cada posição do array)

Address	Data	
0x00000020	0x45	a[0]
0x00000021	0x12	
0x00000022	0x3A	
0x00000023	0xF3	
0x00000024	0xC9	a[1]
0x00000025	0x7D	
0x00000026	0xB3	
0x00000027	0x9D	
0x00000028	0x47	a[2]
0x00000029	0x5F	
0x0000002A	0x6D	
0x0000002B	0x4A	
0x0000002C	0xFD	a[3]
0x0000002D	0xC0	
0x0000002E	0x5A	
0x0000002F	0x7C	
0x00000030	0x1D	
...	...	

&a[0] →

&a[2] →

## Acesso por ponteiro

- endereço do elemento seguinte = endereço actual + dimensão em bytes de cada posição do array

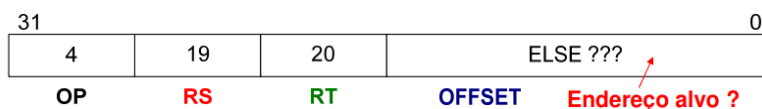
Address	Data	
0x00000020	0x45	a[0]
0x00000021	0x12	
0x00000022	0x3A	
0x00000023	0xF3	
0x00000024	0xC9	a[1]
0x00000025	0x7D	
0x00000026	0xB3	
0x00000027	0x9D	
0x00000028	0x47	a[2]
0x00000029	0x5F	
0x0000002A	0x6D	
0x0000002B	0x4A	
0x0000002C	0xFD	a[3]
0x0000002D	0xC0	
0x0000002E	0x5A	
0x0000002F	0x7C	
0x00000030	0x1D	
...	...	

p →

p+1 →

p+2 →

## Codificação de branches – método geral



- Se o endereço alvo fosse codificado diretamente nos 16 bits menos significativos da instrução, isso significaria que o programa não poderia ter uma dimensão superior a 216 (64K)...
- Em vez de um endereço absoluto, o campo offset pode ser usado para codificar a diferença entre o valor do endereço-alvo e o endereço onde está armazenada a instrução de branch
- O offset é interpretado como um valor em complemento para dois, permitindo o salto para endereços anteriores (offset negativo) ou posteriores (offset positivo) ao PC
- Durante a execução da instrução de branch o seu endereço está disponível no registo PC, pelo que o processador pode calcular o endereço-alvo como: Endereço-alvo = PC + offset
- Endereçamento relativo ao PC (PC-relative addressing)
- No MIPS, na fase de execução de um branch, o PC corresponde ao endereço da instrução seguinte (o PC é incrementado na fase "fetch" da instrução)

- Por essa razão, na codificação de uma instrução de branch, a referência para o cálculo do offset é o endereço da instrução seguinte
- As instruções estão armazenadas em memória em endereços múltiplos de 4 (e.g., 0x00400004, 0x00400008,...) pelo que o offset é também um valor múltiplo de 4 (2 bits menos significativos são sempre 0)
- e modo a otimizar o espaço disponível para o offset na instrução, os dois bits menos significativos não são representado

Considere-se o seguinte exemplo:

```

0x00400000    bne    $19, $20, ELSE
0x00400004    add    $16, $17, $18
0x00400008    j      END_IF
0x0040000C    ELSE:  sub    $16, $16, $19
0x00400010    END_IF:

```

Durante o *instruction fetch* o PC é incrementado (i.e. PC=0x00400004)

O "offset" seria portanto:

ELSE - [ PC ] =

0x0040000C - 0x00400004 = 0x008

O endereço correspondente ao label ELSE é 0x0040000C

No entanto, como **cada instrução ocupa sempre 4 bytes** na memória (a partir de um endereço múltiplo de 4), o "offset" é também múltiplo de 4 Logo:

**"offset" = 0x08 / 4 = 0x02 (offset em número de instruções!!!)**

31	5	19	20	0
				0x0002

Código máquina: 0001011001101000000000000000010 = 0x16740002

Uma instrução de salto condicional pode referenciar qualquer endereço de uma outra instrução que se situe até **32K instruções** antes ou depois dela própria.

### Execução de uma instrução de branch

- O campo offset do código máquina da instrução de branch é então usado para codificar a diferença entre o valor do endereço-alvo e o valor do endereço seguinte ao da instrução de branch, dividida por 4
- Durante a execução da instrução, o processador calcula o endereço-alvo como:

$$\text{Endereço\_alvo} = \text{PC\_atual} + (\text{offset} * 4)$$

ou:

$$\text{Endereço\_alvo} = \text{PC\_atual} + (\text{offset} << 2)$$

**Exemplo**

```

0x00400028    ...
0x0040002C    ...
0x00400030    0X1509FFFD
0x00400034    ...

```

```

0001 0101 0000 1001 1111 1111 1111 1101
000101 01 0000 1001 1111 1111 1111 1101
bne

```

Instrução do tipo I

Offset = 1111 1111 1111 1101

1111 1111 1111 1111 1111 1111 1111 1101

1111 1111 1111 1111 1111 1111 1111 0100 (0xFFFFF4)

Op rs rt offset

bne \$8, \$9, ???

O valor do PC foi incrementado na fase *fetch* da instrução

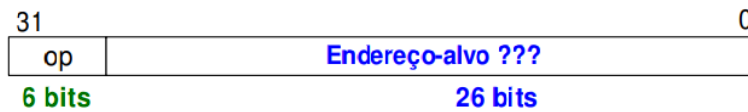
Endereço alvo = PC + (offset << 2) = 0x00400034 + 0xFFFFF4 = 0x00400028

Instrução decodificada: bne \$8, \$9, 0x00400028

### Codificação da instrução de salto incondicional

- No caso da instrução de salto incondicional (" j "), é usado endereçamento pseudo-direto, i.e. o código máquina da instrução codifica diretamente parte do endereço alvo

• Formato J:



- Endereço alvo da instrução "j" é sempre múltiplo de 4 (2 bits menos significativos são sempre 0)



- Exemplo: j Label # com Label = 0x001D14C8

0x001D14C8: 0000 0000 0001 1101 0001 0100 1100 1000

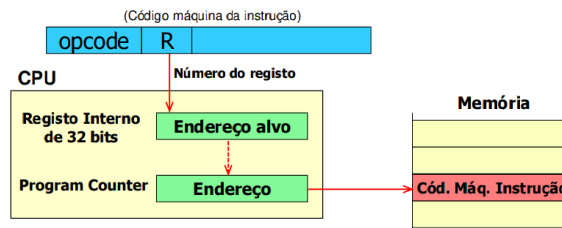
(26 bits) 00 0000 0111 0100 0101 0011 0010

- Código máquina (opcode do "j" é 0x02):

0000 1000 0000 0111 0100 0101 0011 0010 = 0x08074532

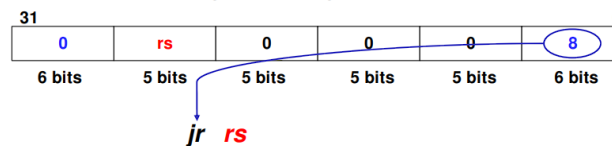
### Salto incondicional – endereçamento indireto por registo

- Haverá maneira de especificar, numa instrução que realize um salto incondicional, um endereço-alvo de 32 bits?
- Há! Utiliza-se endereçamento indireto por registo. Ou seja, um registo interno (de 32 bits) armazena o endereço alvo da instrução de salto (instrução JR - Jump register)



### Instrução JR (jump on register)

O formato de codificação da instrução JR é o formato R:



### Manipulação de constantes no MIPS

- As instruções aritméticas e lógicas que manipulam constantes (do tipo imediato) são identificadas pelo sufixo "i".

```
addi $3,$5,4      # $3 = $5 + 0x0004
andi $17,$18,0x3AF5 # $17 = $18 & 0x3AF5
ori $12,$10,0x0FA2 # $12 = $10 | 0x0FA2
slti $2,$12,16     # $2 = 1 se $12 < 16
                  # ($2 = 0 se $12 ≥ 16)
```

- Estas instruções são codificados usando o formato I. Logo apenas 16 bits podem ser usados para codificar a constante
- Este espaço é geralmente suficiente para armazenar as constantes mais frequentemente utilizadas (geralmente valores pequenos)

- Se há apenas 16 bits dedicados ao armazenamento da constante, qual será a gama de representação dessa constante?
- No caso mais geral, a constante representa uma quantidade inteira, positiva ou negativa, codificada em complemento para dois. É o caso das instruções:

```
addi $3, $5, -4      # equivalente a 0xFFFFC
addi $4, $2, 0x15    # 2110
slti $6, $7, 0xFFFF # -110
```

- Gama de representação da constante: [-32768, +32767]
- A constante de 16 bits é estendida para 32 bits, preservando o sinal (ex: para -4, 0xFFFFC é estendido para 0xFFFFF0FC)
- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Estão neste grupo todas as instruções lógicas:

```
andi $3, $5, 0xFFFF
```

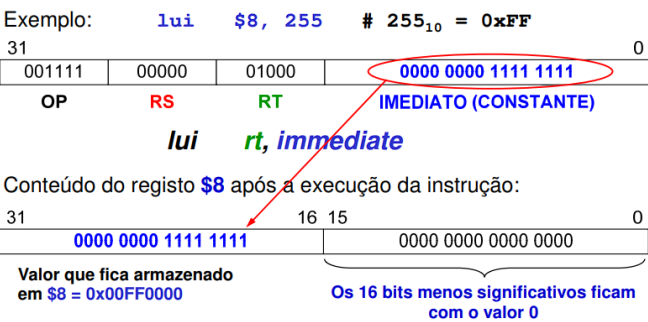
- Gama de representação da constante: [0, 65535]
- A constante de 16 bits é estendida para 32 bits, sendo os 16 mais significativos 0x0000 (para o exemplo: 0x0000FFFF)

### Manipulação de constantes de 32 bits – LUI

- Para permitir a manipulação de constantes com mais de 16 bits, o ISA do MIPS inclui a seguinte instrução, também codificada com o formato I:

```
lui $reg, immediate
```

- A instrução lui ("Load Upper Immediate"), coloca a constante "immediate" nos 16 bits mais significativos do registo destino (\$reg)
- Os 16 bits menos significativos ficam com 0x000



### Manipulação de constantes de 32 bits – LA / LI

A instrução virtual "load address"

```
la $16, MyData #Ex. MyData = 0x10010034
# (segmento de dados em 0x1001000)
```

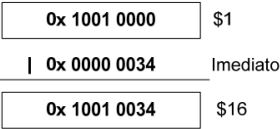
é executada no MIPS pela sequência de instruções nativas:

```
lui $1, 0x1001      # $1 = 0x10010000
ori $16, $1, 0x0034 # $16 = 0x10010000 | 0x00000034
```

Notas:

• O registo \$1 (\$at) é reservado para o Assembler, para permitir este tipo de decomposição de instruções virtuais em instruções nativas.

• A instrução "li" (load immediate) é decomposta em instruções nativas de forma análoga à instrução "la"



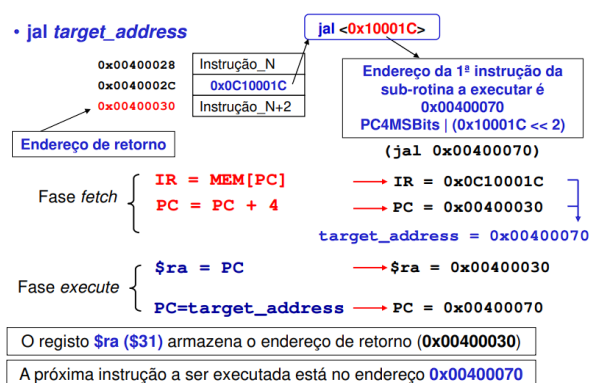
## Porque se usam funções (sub-rotinas)?

- Há três razões principais que justificam a existência de funções\*:
  - A reutilização no contexto de um determinado programa
  - A reutilização no contexto de um conjunto de programas
  - A organização e estruturação do código

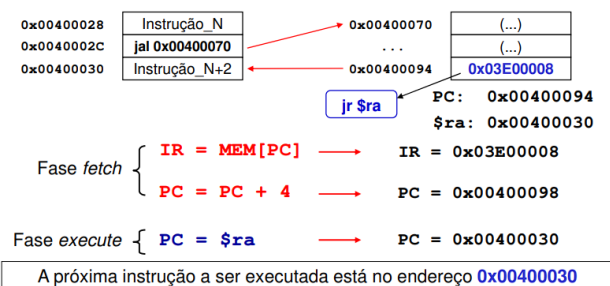
(\*) No contexto da linguagem Assembly, as funções e os procedimentos são genericamente conhecidas por sub-rotinas!

## Sub-rotinas no MIPS: a instrução JAL

- A ligação (link) entre o chamador e o chamado (sub-rotina) é feita pela instrução JAL (jump and link)
- A instrução JAL é uma instrução de salto incondicional, que armazena o valor atual do Program Counter no registo \$ra
- A instrução JAL é codificada do mesmo modo que a instrução



- Como **regressar** à instrução que sucede à instrução "jal" ?
- Aproveita-se o endereço de retorno armazenado em \$ra durante a execução da instrução "jal" (instrução "jr register")



## Instruções JAL e JALR

- A instrução "jal" é codificada do mesmo modo que a instrução "j": formato j em que os 26 bits menos significativos são obtidos dos 28 bits menos significativos do endereço-alvo, deslocados à direita dois bits
- Durante a execução, a obtenção do endereço-alvo é feita do mesmo modo que na instrução "j"
- A especificação de um endereço-alvo de 32 bits é possível através da utilização da instrução "jalr" (jump and link register)
- A instrução "jalr" funciona de modo idêntico à instrução "jal", exceto na obtenção do endereço-alvo:
  - o endereço da sub-rotina é lido do registo especificado na instrução (endereçamento indireto por registo); ex: jalr \$t2
- A instrução "jalr" é codificada com o formato R

## Sub-rotinas



- A reutilização de sub-rotinas é essencial em programação.
- As sub-rotinas surgem frequentemente agrupadas em bibliotecas.
- A utilização de sub-rotinas escritas por outros para serviço dos nossos programas, não deverá implicar o conhecimento dos detalhes da sua implementação
- Na perspetiva do programador, a sub-rotina que este tem a responsabilidade de escrever é um trecho de código isolado
- Torna-se óbvia a necessidade de definir um conjunto de regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”:
  - definição da interface entre ambos, i.e., quais os parâmetros de entrada e como os passar para a sub-rotina e como devolver resultados ao programa chamador
  - princípios que assegurem uma “sã convivência” entre os dois, de modo a que um não destrua os dados do outro

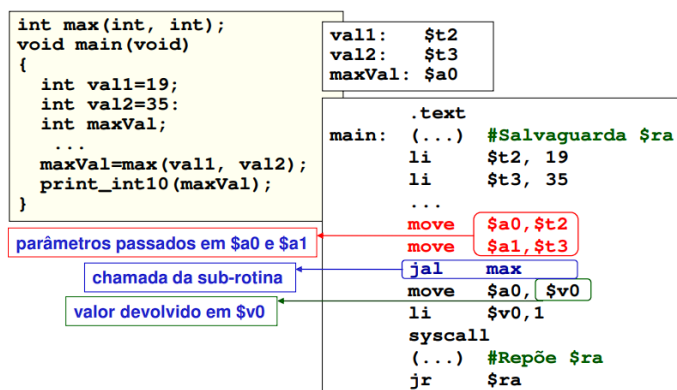
### Regras a definir entre chamador e a sub-rotina chamada

- Ao nível da interface:
  - Como passar parâmetros do “chamador” para a sub-rotina, identificar quantos e onde são passados
  - Como devolver, para o “chamador”, resultados produzidos pela sub-rotina
- Ao nível das regras de “sã c:onvivência”:
  - Que registos do CPU podem “chamador” e sub-rotina usar, sem que haja alteração indevida de informação por parte da sub-rotina
  - Como partilhar a memória usada para armazenar dados, sem risco de sobreposição

### Convenções do MIPS (passagem e devolução de valores)

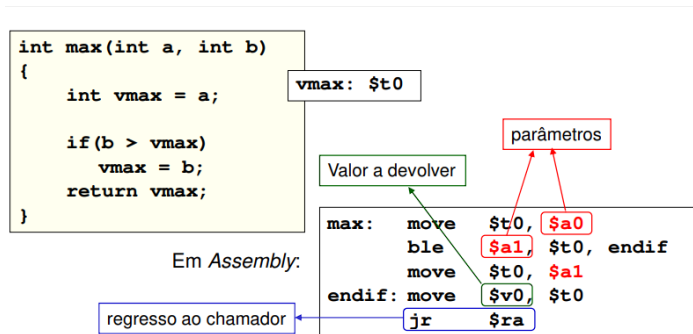
- Os parâmetros que possam ser armazenados na dimensão de um registo (32 bits, i.e., char, int, ponteiros) devem ser passados à sub-rotina nos registos \$a0 a \$a3 (\$4 a \$7) por esta ordem
  - o primeiro parâmetro sempre em \$a0, o segundo em \$a1 e assim sucessivamente
- A sub-rotina pode devolver um valor de 32 bits ou um de 64 bits:
  - Se o valor a devolver é de 32 bits é utilizado o registo \$v0
  - Se o valor a devolver é de 64 bits, são utilizados os registos \$v1 (32 bits mais significativos) e \$v0 (32 bits menos significativos)

### Exemplo (chamador)



- Para escrever o programa “chamador”, não é necessário conhecer os detalhes de implementação da sub-rotina

### Exemplo (sub-rotina)



- Para escrever o código da sub-rotina, não é necessário conhecer os detalhes de implementação do “chamador”
- Será necessário salvar o valor de \$ra?

### Estratégias para a salvaguarda de registos

- Que registos pode usar uma sub-rotina, sem que se corra o risco de que os mesmos registos estejam a ser usados pelo programa “chamador”, potenciando assim a destruição de informação vital para a execução do programa como um todo?
- Estratégia “caller-saved”
  - Deixa-se ao cuidado do programa “chamador” a responsabilidade de salvar o conteúdo da totalidade dos registos antes de chamar a sub-rotina □ Cabe-lhe também a tarefa de repor posteriormente o seu valor
  - No limite, é admissível que o “chamador” salve apenas os registos de cujo conteúdo venha a precisar mais tarde
- Estratégia “callee-saved”
  - Entrega-se à sub-rotina a responsabilidade pela prévia salvaguarda dos registos que vai usar
  - Assegura, igualmente, a tarefa de repor o seu valor imediatamente antes de regressar ao programa “chamador”.

### Convenção para salvaguarda de registos no MIPS

- Os registos \$t0 a \$t9, \$v0 e \$v1, e \$a0 a \$a3 podem ser livremente utilizados e alterados pelas sub-rotinas
- Os registos \$s0 a \$s7 não podem, na perspetiva do chamador, ser alterados pelas sub-rotinas
  - Se uma dada sub-rotina precisar de usar qualquer um dos registos \$s0 a \$s7 compete a essa sub-rotina salvar previamente o seu conteúdo, repondo-o imediatamente antes de terminar
  - Ou seja, é seguro para o programa chamador usar um registo \$sn para armazenar um valor que vai necessitar após a chamada à sub-rotina, uma vez que tem a garantia que esta não o modifica

### Considerações práticas sobre a utilização da convenção

- sub-rotinas terminais (sub-rotinas folha, i.e., que não chamam qualquer sub-rotina)
  - Só devem utilizar registos que não têm a responsabilidade de salvar (t0..t9, v0..v1 e a0..a3)
- sub-rotinas intermédias (sub-rotinas que chamam outras sub-rotinas)
  - Devem utilizar os registos \$s0..\$s7 para o armazenamento de valores que pretendam preservar durante a chamada à sub-rotina seguinte
    - A utilização de qualquer um dos registos \$s0 a \$s7 implica a sua prévia salvaguarda na memória externa logo no início da sub-rotina e a respetiva reposição no final
  - Devem utilizar os registos \$t0..\$t9, \$v0..\$v1 e \$a0..\$a3 para os restantes valores

### Representação de inteiros

- Tipicamente, um inteiro pode ocupar um número de bits igual à dimensão de um registo interno do CPU.
- No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representável é:

$$N_{\text{inteiros}} = 2^{32} = 4.294.967.296_{10} = [0, 4.294.967.295_{10}]$$

- Os circuitos que realizam operações aritméticas estão igualmente limitados a um número finito de bits, geralmente igual à dimensão dos registos internos do CPU
- Os circuitos aritméticos operam assim em aritmética modular, ou seja em  $\text{mod}(2^n)$  em que "n" é o número de bits da representação
- O maior valor que um resultado aritmético pode tomar será portanto  $2^{n-1}$ , sendo o valor inteiro imediatamente a seguir o valor zero (representação circular)
- Num CPU com uma ALU de 8 bits, por exemplo, o resultado da soma dos números 11001011 e 00110111 seria:

$$11001011 + 00110111 = 1\ 00000010$$

carry      resultado com 8 bits

- No caso em que os operandos são do tipo unsigned, o bit carry, se igual a '1', sinaliza que o resultado não cabe num registo de 8 bits, ou seja sinaliza a ocorrência de overflow
- No caso em que os operandos são do tipo signed (codificados em complemento para 2) o bit de carry, por si só, não tem qualquer significado, e não faz parte do resultado

### Representação em complemento para dois

- O método usado em sistemas computacionais para a codificação de quantidades inteiras com sinal (signed) é "complemento para dois"
- Definição: Se K é um número positivo, então  $K^*$  é o seu complemento para 2 (complemento verdadeiro) e é dado por:

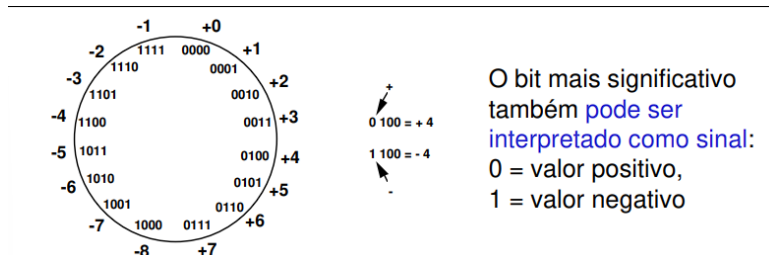
$$K^* = 2^n - K$$

em que "n" é o número de bits da representação

- Exemplo: determinar a representação de -5, com 4 bits

$$\begin{aligned}
 N &= 5_{10} = 0101_2 \\
 2^n &= 2^4 = 10000 \\
 2^n - N &= 10000 - 0101 = 1011 = N^*
 \end{aligned}$$

- Método prático: inverter todos os bits do valor original e somar 1 (0101  $\Rightarrow$  1010; 1010 + 1 = 1011 )
  - Este método é reversível:  $C1(1011)=0100$ ;  $0100+1=0101$



- Uma única representação para 0
- Codificação assimétrica (mais um negativo do que positivos)
- A subtração é realizada através de uma operação de soma com o complemento para 2 do 2.º operando:  $(a-b) = (a+(-b))$
- Uma quantidade de N bits codificada em complemento para 2 pode ser representada pelo seguinte polinómio:

$$-(a_{N-1} \cdot 2^{N-1}) + (a_{N-2} \cdot 2^{N-2}) + \dots + (a_1 \cdot 2^1) + (a_0 \cdot 2^0)$$

Onde o bit indicador de sinal ( $a_{N-1}$ ) é multiplicado por  $-2^{N-1}$  e os restantes pela versão positiva do respetivo peso

- **Exemplo:** Qual o valor representado em base 10 pela quantidade  $10100101_2$ , supondo uma representação em complemento para 2 com 8 bits?

• **R1:**  $10100101_2 = -(1 \times 2^7) + (1 \times 2^5) + (1 \times 2^2) + (1 \times 2^0)$   
 $= -128 + 32 + 4 + 1 = -91_{10}$

• **R2:** O valor é negativo, calcular o módulo (simétrico de  $10100101$ ):  $01011010 + 1 = 01011011_2 = 5B_{16} = 91_{10}$   
o módulo da quantidade é 91; logo o valor representado é  $-91_{10}$

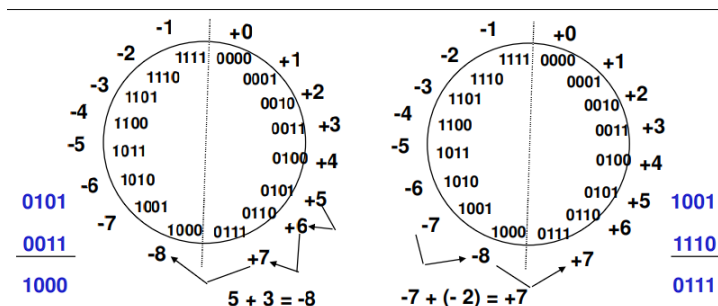
- Exemplos de operações, com 4 bits

( 4 + 3 )	4	0100	( -4 - 3 )	-4	1100
	+3	0011		+(-3)	1101
	7	0111		-7	11001

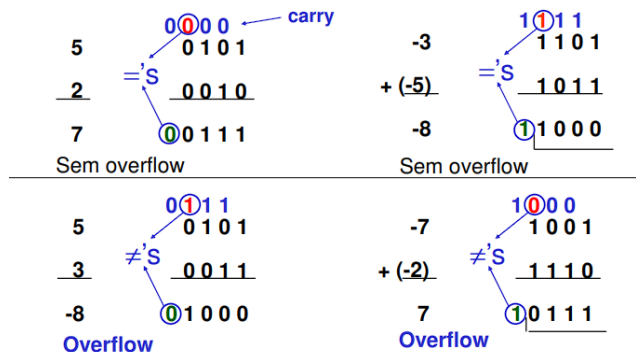
( 4 - 3 )	4	0100	( -4 + 3 )	-4	1100
	+(-3)	1101		+3	0011
	1	10001		-1	1111

- Este esquema simples de adição com sinal torna o complemento para 2 o preferido para representação de inteiros em arquitetura de computadores

### Overflow em complemento para 2



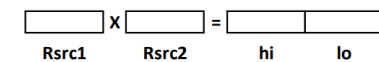
- Ocorre overflow quando é ultrapassada a gama de representação. Isso acontece quando:
  - se somam dois positivos e o resultado obtido é negativo
  - se somam dois negativos e o resultado obtido é positivo



A situação de overflow ocorre quando o carry-in do bit mais significativo não é igual ao carry-out.

## Multiplicação de inteiros

- Multiplicação de quantidades sem sinal: algoritmo clássico que é usado na multiplicação em decimal
- Multiplicação de quantidades com sinal (representadas em complemento para dois): algoritmo de Booth
- Uma multiplicação que envolva dois operandos de N bits carece de um espaço de armazenamento, para o resultado, de  $2 \times N$  bits
- No MIPS, a multiplicação e a divisão são asseguradas por um módulo independente da ALU
- Os operandos são registos de 32 bits. Na multiplicação, tal implica que o resultado tem de ser armazenado com 64 bits
- Os resultados são armazenados num par de registos especiais designados por HI e LO, cada um com 32 bits
- Estes registos são de uso específico da unidade de multiplicação e divisão de inteiros



- O registo HI armazena os 32 bits mais significativos do resultado
- O registo LO armazena os 32 bits menos significativos do resultado
- A transferência de informação entre os registos HI e LO e os restantes registos de uso geral faz-se através das instruções mfhi e mflo:

mfhi Rdst # move from hi: copia HI para Rdst

mflo Rdst # move from lo: copia LO para Rdst

- A unidade de multiplicação pode operar considerando os operandos com sinal (multiplicação signed) ou sem sinal (multiplicação unsigned); a distinção é feita através da mnemónica da instrução:
  - mult – multiplicação "signed"
  - multu – multiplicação "unsigned"

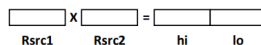
• Em *Assembly*, a multiplicação é então efetuada pelas instruções

**mult** **Rsrc1, Rsrc2** # Multiply (signed)

**multu** **Rsrc1, Rsrc2** # Multiply unsigned

em que **Rsrc1** e **Rsrc2** são os dois registos a multiplicar

• O **resultado** fica armazenado nos **registos HI e LO**



• **Exemplo:** Multiplicar os registos \$t0 e \$t1 e colocar o resultado nos registos \$a1 (32 bits mais significativos) e \$a0 (32 bits menos significativos); os operandos devem ser interpretados com sinal

**mult** **\$t0, \$t1** # resultado em hi e lo

**mfhi** **\$a1** # copia hi para registo \$a1

**mflo** **\$a0** # copia lo para registo \$a0

## Divisão de inteiros com sinal

- A divisão de inteiros com sinal faz-se, do ponto de vista algorítmico, em sinal e módulo
- Nas divisões com sinal aplicam-se as seguintes regras:
  - Divide-se dividendo por divisor, em módulo
  - O quociente tem sinal negativo se os sinais do dividendo e do divisor forem diferentes
  - O resto tem o mesmo sinal do dividendo
- Exemplo 1 (dividendo = -7, divisor = 3):

$$-7 / 3 = -2 \quad \text{resto} = -1$$

- Exemplo 2 (dividendo = 7, divisor = -3):

$$7 / -3 = -2 \quad \text{resto} = 1$$

mfhi    \$a0    # \$a0 = hi