Aula 6

- Representação de números inteiros com sinal: complemento para dois. Exemplos de operações aritméticas
- *Overflow* e mecanismos para a sua deteção
- Bloco funcional e operações de uma ALU de 32 bits
- Endereçamento imediato e uso de constantes
- Multiplicação de inteiros no MIPS
- Divisão de inteiros no MIPS. Divisão de inteiros com sinal

Bernardo Cunha, José Luís Azevedo

Representação de inteiros

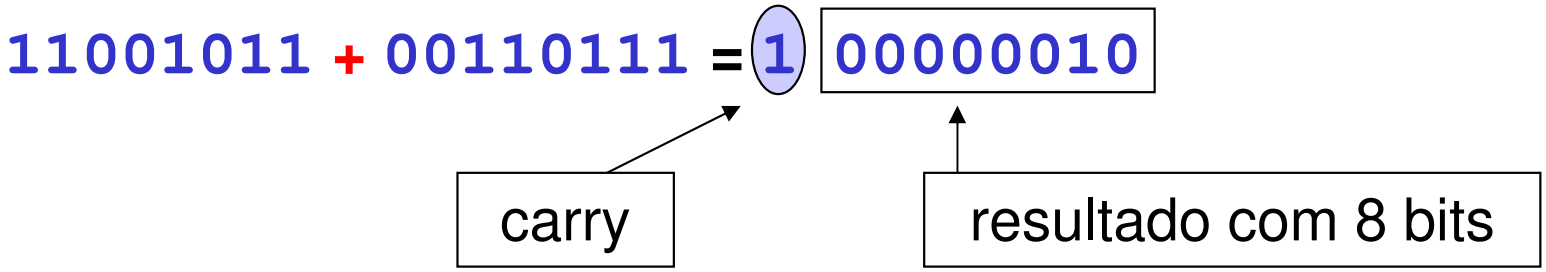
- Os computadores usam sistema binário; inteiros são representados em base 2 (0 e 1)
- Cada inteiro ocupa um número de bits igual à dimensão de um registo interno do CPU
- A gama de inteiros representáveis é finita: $\mathbf{N_{inteiros} = 2^n}$, sendo "**n**" o número de bits do registo; em MIPS, 32 bits:

$$2^{32} = 4.294.967.296, [0, 4.294.967.295]$$

- As operações aritméticas também são limitadas a este número de bits, funcionando em aritmética modular (**mod 2^n**), com representação circular: após o valor máximo (**$2^n - 1$**) volta-se ao zero

Representação de inteiros

- Num CPU com uma ALU de 8 bits, por exemplo, o resultado da soma dos números `11001011` e `00110111` seria:



- 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:

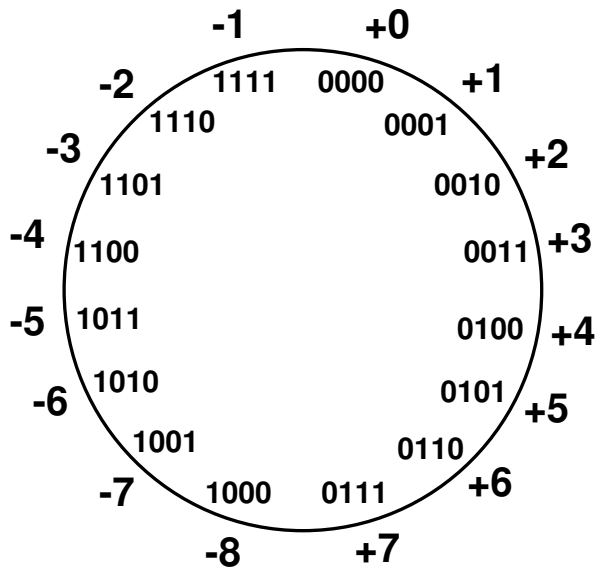
$$K^* = 2^n - K$$

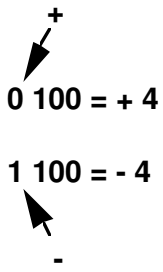
  em que "n" é o número de bits da representação
- **Exemplo**: determinar a representação de -5, com 4 bits

$$N = 5_{10} = 0101_2$$
$$2^n = 2^4 = 10000$$
$$2^n - N = 10000 - 0101 = 1011 = N^*$$

- **Método prático**: inverter todos os bits do valor original e somar 1 (**0101 => 1010; 1010 + 1 = 1011**)
  - Este método é reversível: C_1(**1011**)**=0100; 0100+1=0101**

Representação em complemento para dois





O bit mais significativo também pode ser interpretado como sinal:
0 = valor positivo,
1 = valor negativo

- 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:

$$-(a_{N-1}.2^{N-1})+(a_{N-2}.2^{N-2})+...+(a_1.2^1)+(a_0.2^0)$$

Representação em complemento para dois

- Uma quantidade de N bits codificada em complemento para 2 pode então ser representada pelo seguinte polinómio:

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

  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**: $10100101_2 = -(1x2^7)+(1x2^5)+(1x2^2)+(1x2^0)$
    $= -128 + 32 + 4 + 1 = -91_{10}$
  - **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}

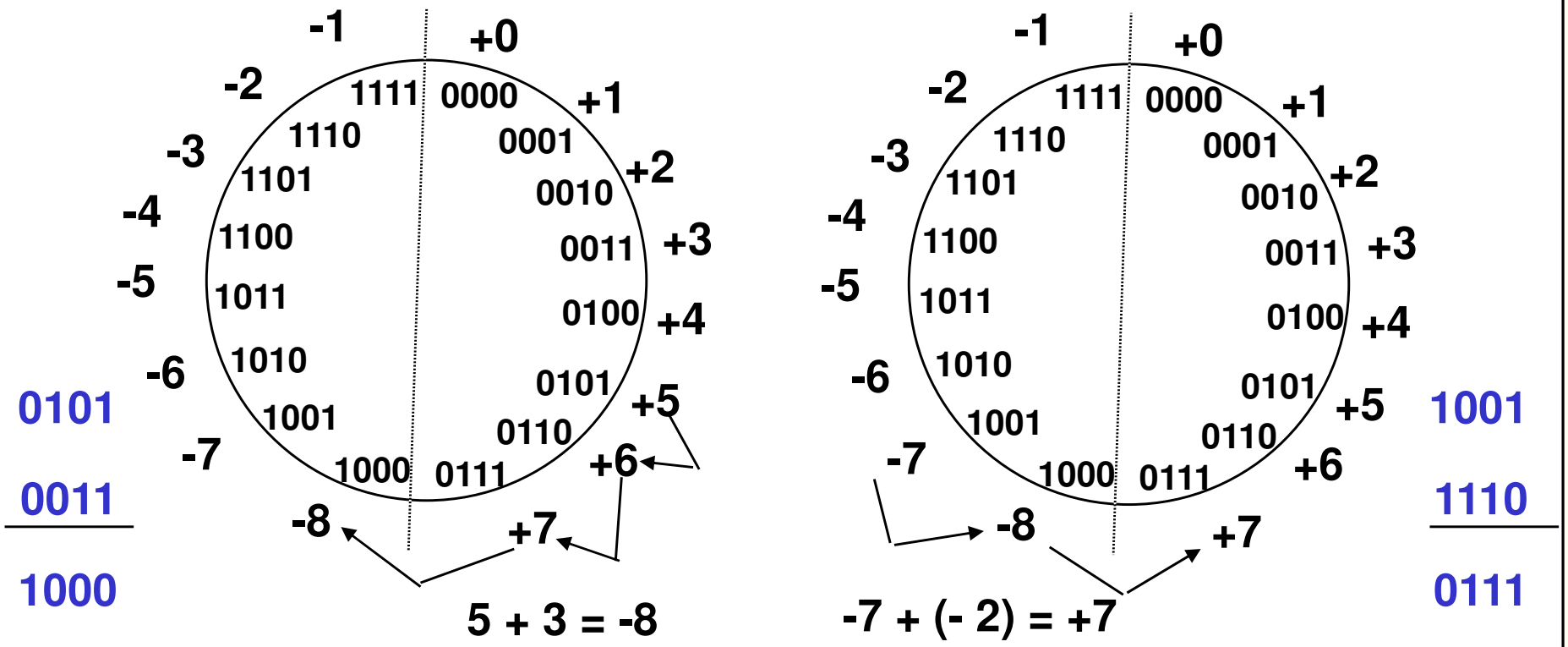
Operações em complemento para dois

- 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

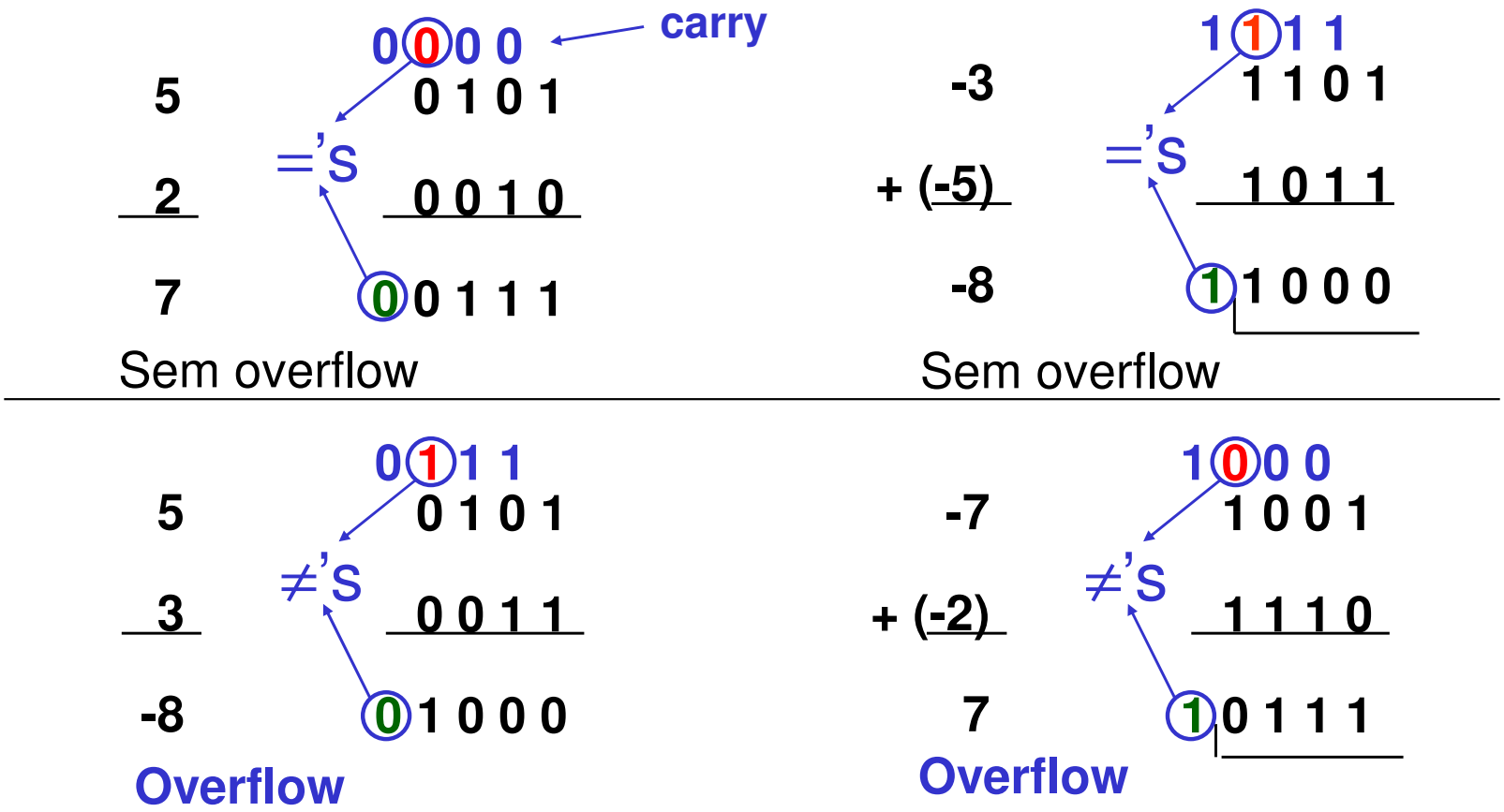
- Este esquema simples de adição com sinal torna o complemento para 2 o preferido para representação de inteiros em arquitetura de computadores

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

Overflow em complemento para 2



A situação de ***overflow*** **ocorre** quando o *carry-in* do bit mais significativo não é igual ao *carry-out*, ou seja, quando:

$$C_{n-1} \oplus C_n = 1$$

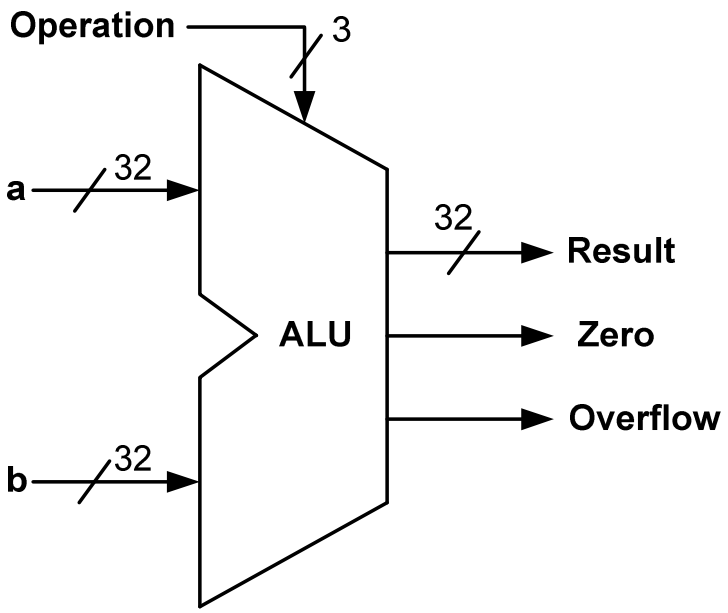
Overflow em operações aritméticas

- **Operandos interpretados em complemento para 2 (i.e. com sinal):**
  - Quando $A + B > 2^{n-1}-1$ ou $A + B < -2^{n-1}$
    - $OVF = (C_{n-1} \cdot \overline{C_n}) + (\overline{C_{n-1}} \cdot C_n) = C_{n-1} \oplus C_n$
  - Alternativamente, não tendo acesso aos bits intermédios de *carry*, (R = A + B):
    - $OVF = \overline{R_{n-1}} \cdot A_{n-1} \cdot B_{n-1} + R_{n-1} \cdot \overline{A_{n-1}} \cdot \overline{B_{n-1}}$
- **Operandos interpretados sem sinal:**
  - Quando $A+B > 2^n-1$ ou **A−B c/ B>A**
  - O bit de *carry* $C_n = 1$ sinaliza a ocorrência de *overflow*
- O MIPS apenas deteta *overflow* nas operações de adição com sinal (ADD, SUB, ADDI) e, quando isso acontece, gera uma exceção. ADDU, SUBU e ADDIU não detetam *overflow*

ALU de 32 bits - exemplo de bloco funcional

- A ALU deverá realizar as operações:
  - ADD, SUB
  - AND, OR
  - SLT (set if less than)
- Deverá ainda:
  - Detetar e sinalizar *overflow* (operandos em complemento para 2)
  - Sinalizar resultado igual a zero

Operation	ALU Action
0 0 0	And
0 0 1	Or
0 1 0	Add
1 1 0	Subtract
1 1 1	Set if less than



Bloco funcional correspondente a uma ALU de 32 bits

Manipulação de constantes inteiras

- Constante é um valor determinado com antecedência (quando o programa é escrito) e que não se pretende que seja ou possa ser mudado durante a execução do programa
- As constantes poderiam ser armazenadas na memória externa. Nesse caso, a sua utilização implicaria sempre o recurso a duas instruções:
  - leitura do valor residente em memória para um registo interno
  - operação com essa constante
- Para aumentar a eficiência, as arquiteturas disponibilizam um conjunto de instruções em que as **constantes se encontram armazenadas na própria instrução**
- Desta forma, com a leitura da instrução, o acesso à constante é "**imediato**", sem necessidade de recorrer a uma operação prévia de leitura da memória: "**endereçamento imediato**"

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?
  - Depende da instrução...

Manipulação de constantes no MIPS

- 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 0xFFFC
addi  $4, $2, 0x15    # 21_10
slti  $6, $7, 0xFFFF  # -1_10
```

  - Gama de representação da constante: [**-32768, +32767**]
  - A constante de 16 bits é estendida para 32 bits, preservando o sinal (ex: para -4, **0xFFFC** é estendido para **0xFFFFFFFC**)

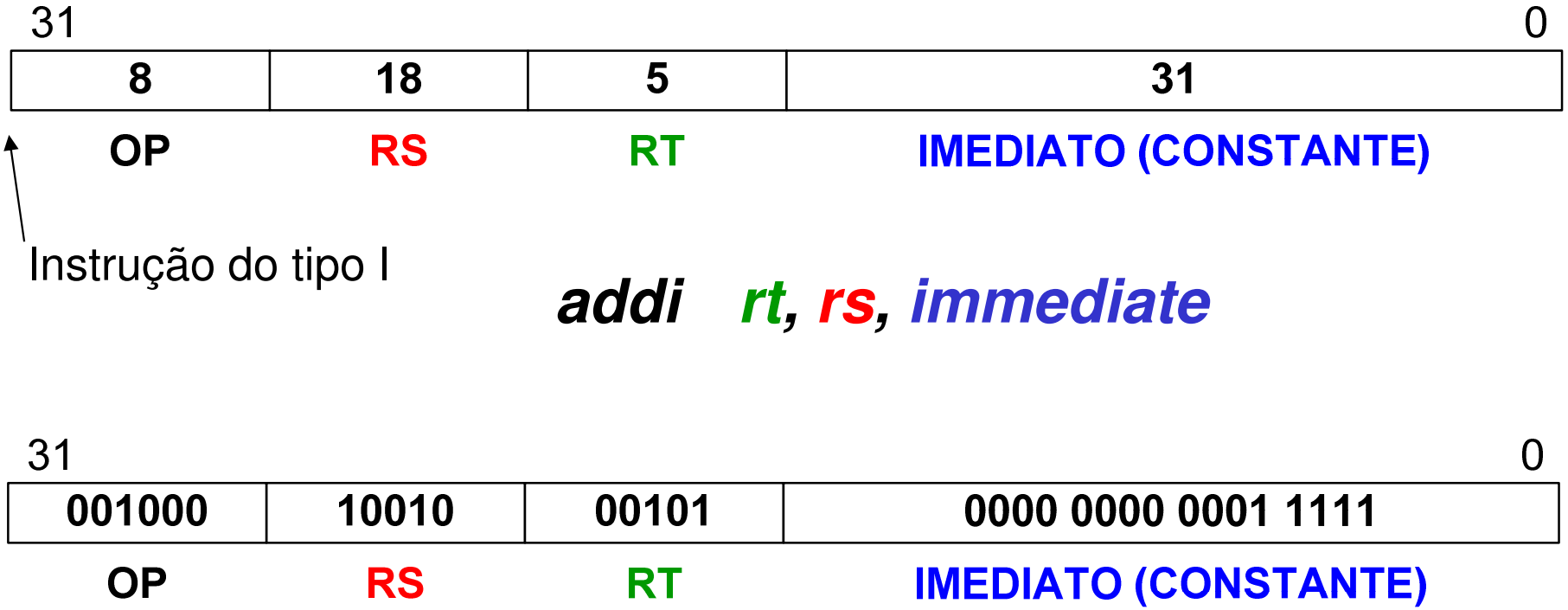
- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Estão neste grupo todas as instruções lógicas:

```
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**)

Codificação das instruções que usam constantes

Exemplo: **addi $5, $18, 31**



Cod. Máquina: 00100010010000101000000000011111 = 0x2245001F

Manipulação de constantes de 32 bits – LUI

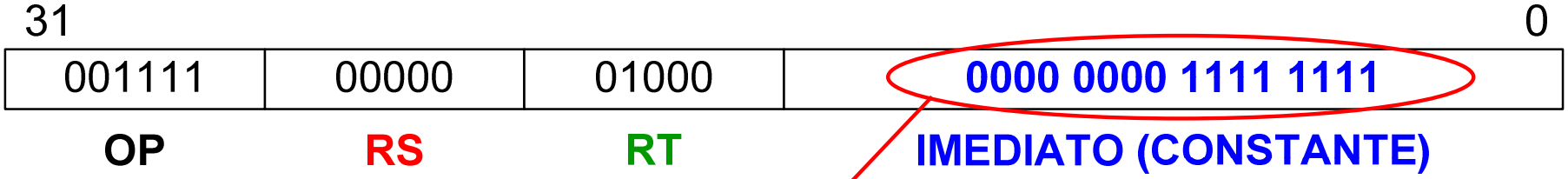
- Em alguns casos pode ser necessário manipular constantes que necessitem de um espaço de armazenamento com mais do que 16 bits (e.g., a referência explícita a um endereço)
- Como lidar com esses casos?
- Para permitir a manipulação de constantes com mais de 16 bits, o ISA do MIPS inclui a seguinte instrução, 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 `0x0000`

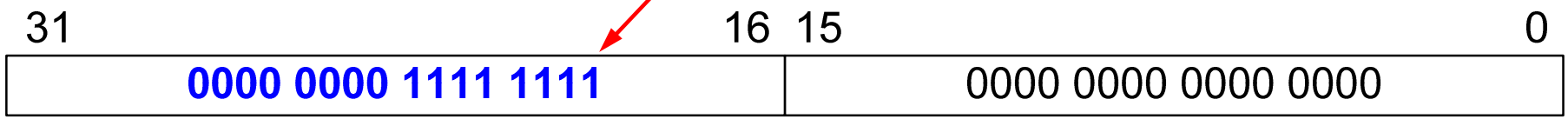
Manipulação de constantes de 32 bits – LUI

Exemplo: `lui $8, 255` `# 255`$_{10}$ `= 0xFF`



lui rt, immediate

Conteúdo do registo **$8** após a execução da instrução:



Valor que fica armazenado em $8 = 0x00FF0000

Os 16 bits menos significativos ficam com o valor 0

- Exemplo: inicializar o registo **$6** com o valor **0xF32864D9**

```
lui $6,0xF328     #$6=0xF3280000
ori $6,$6,0x64D9  #$6=0xF3280000 | 0x000064D9=0xF32864D9
```

Manipulação de constantes de 32 bits – LA / LI

A **instrução virtual** "**load address**"

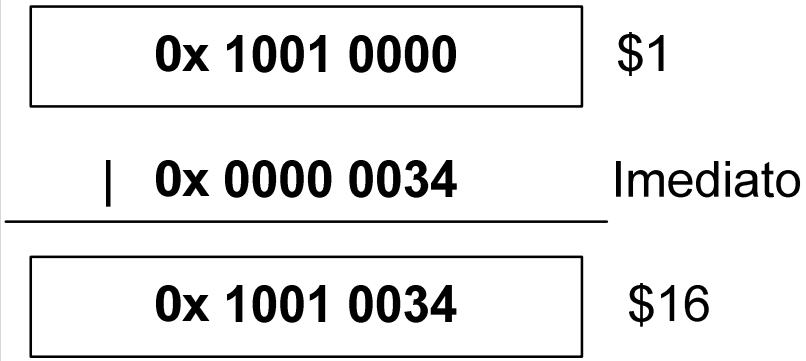
```
la   $16,MyData #Ex. MyData = 0x10010034
                #(segmento de dados em 0x10010000)
```

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**"

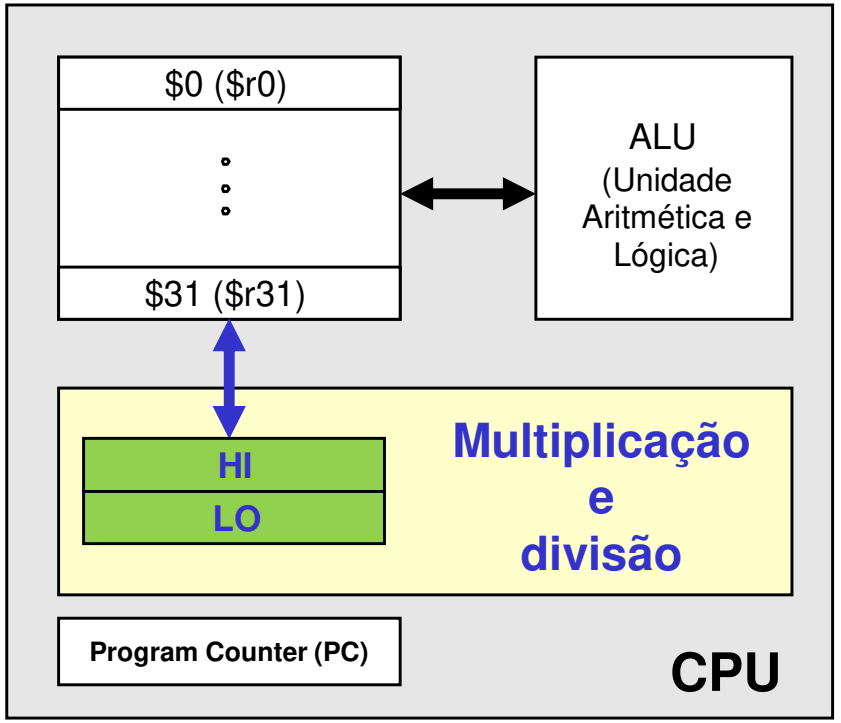


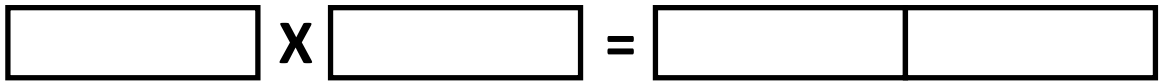
Multiplicação de inteiros

- Devido ao aumento de complexidade que daí resulta, nem todas as arquiteturas suportam, ao nível do *hardware*, a capacidade para efetuar operações aritméticas de multiplicação e divisão 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*N bits**

A Multiplicação de inteiros no MIPS

- 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





A Multiplicação de inteiros no MIPS

- 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"

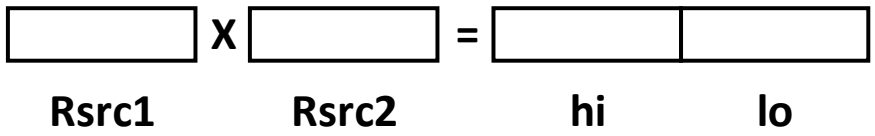
A Multiplicação de inteiros no MIPS

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

```
mult    Rsrc1, Rsrc2    # Multiply (signed)
multu   Rscr1, 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
```

Instruções virtuais de multiplicação

Multiplicação *signed*

mul Rdst, Rsrc1, Rsrc2		
	mult	Rsrc1, Rscr2
	mflo	Rdst

Multiplicação *unsigned*

mulu Rdst, Rsrc1, Rsrc2		
	multu	Rsrc1, Rscr2
	mflo	Rdst

Multiplicação *unsigned* com deteção de overflow

mulou Rdst, Rsrc1, Rsrc2		
	multu	Rsrc1, Rscr2
	mfhi	$1
	beq	$1, $0, cont
	break	
cont:	mflo	Rdst

Multiplicação *signed* com deteção de overflow

mulo Rdst, Rsrc1, Rsrc2		
	mult	Rsrc1, Rscr2
	mfhi	$1
	mflo	Rdst
	sra	Rdst, Rdst, 31
	beq	$1, Rdst, cont
	break	
cont:	mflo	Rdst

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        resto = -1
```

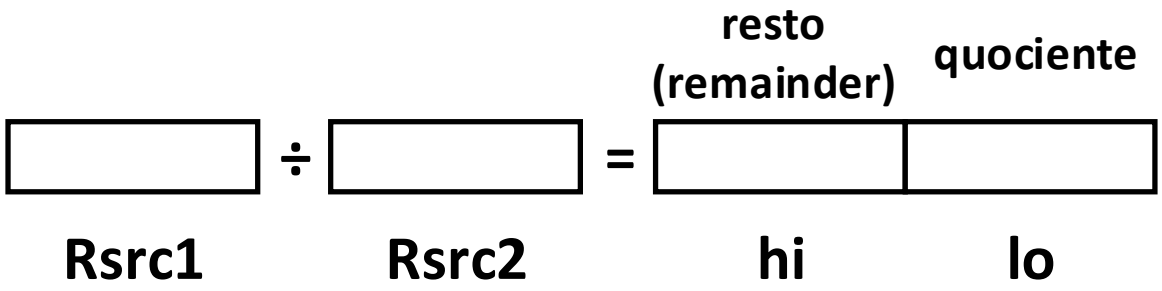
- Exemplo 2 (dividendo = 7, divisor = -3):

```
 7 / -3 = -2        resto =  1
```

Note que: **`Dividendo = Divisor * Quociente + Resto`**

A Divisão de inteiros no MIPS

- Tal como na multiplicação, continua a existir a necessidade de um registo de 64 bits para armazenar o resultado final na forma de um quociente e de um resto
- Os mesmos registos, **HI** e **LO**, que tinham já sido usados para a multiplicação, são igualmente utilizados para a divisão:
  - o registo **HI armazena o resto da divisão** inteira
  - o registo **LO armazena o quociente da divisão** inteira

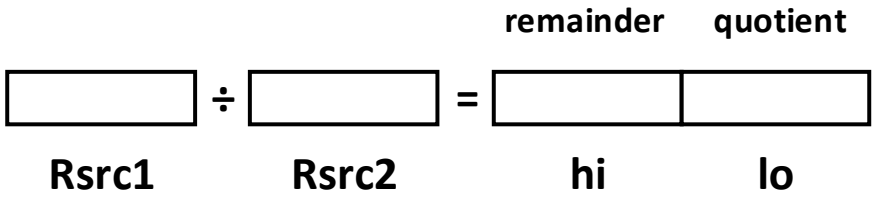


A Divisão de inteiros no MIPS

- No MIPS, as instruções *Assembly* de divisão são:

```
div     Rsrc1, Rsrc2    # Divide (signed)
divu    Rsrc1, Rsrc2    # Divide unsigned
```

- em que **Rsrc1** é o dividendo e **Rsrc2** o divisor. O **resultado** fica armazenado nos registos **HI (resto)** e **LO (quociente)**.



- **Exemplo**: obter o resto da divisão inteira entre os valores armazenados em $t0 e $t5, colocando o resultado em $a0

```
div     $t0, $t5    # hi = $t0 % $t5
                    # lo = $t0  /  $t5
mfhi    $a0         # $a0 = hi
```

Instruções virtuais de divisão

Divisão *signed*

div Rdst, Rsrc1, Rsrc2	
div	Rsrc1, Rscr2
mflo	Rdst

Divisão *unsigned*

divu Rdst, Rsrc1, Rsrc2	
divu	Rsrc1, Rscr2
mflo	Rdst

Resto da divisão *signed*

rem Rdst, Rsrc1, Rsrc2	
div	Rsrc1, Rscr2
mfhi	Rdst

Resto da divisão *unsigned*

remu Rdst, Rsrc1, Rsrc2	
divu	Rsrc1, Rscr2
mfhi	Rdst

Exercícios

- Para uma codificação em complemento para 2, apresente a gama de representação que é possível obter com 3, 4, 5, 8 e 16 bits (indique os valores-limite da representação em binário, hexadecimal e em decimal com sinal e módulo).
- Determine a representação em complemento para 2 com 16 bits das seguintes quantidades:
  - **`5`**, **`-3`**, **`-128`**, **`-32768`**, **`31`**, **`-8`**, **`256`**, **`-32`**
- Determine o valor em decimal representado por cada uma das quantidades seguintes, supondo que estão codificadas em complemento para 2 com 8 bits:
  - **`00101011`**$_2$, **`0xA5`**, **`10101101`**$_2$, **`0x6B`**, **`0xFA`**, **`0x80`**
- Determine a representação das quantidades do exercício anterior em hexadecimal com 16 bits (também codificadas em complemento para 2).

Exercícios

- Como é realizada a deteção de *overflow* em operações de adição com quantidades sem sinal? E com quantidades com sinal (codificadas em complemento para 2)?
- Para a multiplicação de dois operandos de "**m**" e "**n**" bits, respetivamente, qual o número de bits necessário para o armazenamento do resultado?
- Apresente a decomposição em instruções nativas da instrução virtual **`mul $5,$6,$7`**
- Determine o resultado da instrução anterior, quando **`$6=0xFFFFFFFE`** e **`$7=0x00000005`**.
- Apresente a decomposição em instruções nativas das instruções virtuais **`div $5,$6,$7`** e **`rem $5,$6,$7`**
- Determine o resultado das instruções anteriores, quando **`$6=0xFFFFFFF0`** e **`$7=0x00000003`**

Exercícios

- As duas sub-rotinas do slide seguinte permitem detetar *overflow* nas operações de adição com e sem sinal, no MIPS. Analise o código apresentado e determine o resultado produzido, pelas duas sub-rotinas, nas seguintes situações:
  - **`$a0=0x7FFFFFF1, $a1=0x0000000E;`**
  - **`$a0=0x7FFFFFF1, $a1=0x0000000F;`**
  - **`$a0=0xFFFFFFF1, $a1=0xFFFFFFFF;`**
  - **`$a0=0x80000000, $a1=0x80000000;`**
- Ainda no código das sub-rotinas, qual a razão para não haver salvaguarda de qualquer registo na *stack*?

Exercícios

```
    # Overflow detection, signed
    # int isovf_signed(int a, int b);
    isovf_signed:   ori  $v0,$0,0
                    xor  $1,$a0,$a1
                    slt  $1,$1,$0
                    bne  $1,$0,notovf_s
                    addu $1,$a0,$a1
                    xor  $1,$1,$a0
                    slt  $1,$1,$0
                    beq  $1,$0,notovf_s
                    ori  $v0,$0,1
    notovf_s:       jr   $ra

    # Overflow detection, unsigned
    # int isovf_unsigned(unsigned int a, unsigned int b);
    isovf_unsigned:ori  $v0,$0,0
                    nor  $1,$a1,$0
                    sltu $1,$1,$a0
                    beq  $1,$0,notovf_u
                    ori  $v0,$0,1
    notovf_u:       jr   $ra
```