Aula 11

- Representação de números inteiros com sinal: complemento para dois. Exemplos de operações aritméticas
- *Overflow* e mecanismos para a sua deteção
- Construção de uma ALU de 32 bits
- A multiplicação de inteiros no MIPS
- Divisão de inteiros no MIPS. Divisão de inteiros com sinal

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Representação de inteiros

- Sendo um computador um sistema digital binário, a representação de inteiros faz-se sempre em base 2 (símbolos 0 e 1).
- **Tipicamente, um inteiro pode ocupar um número de bits igual à dimensão de um registo interno do CPU**.
- A gama de valores inteiros representáveis é, assim, finita, e corresponde ao número máximo de combinações que é possível obter com o número de bits de um registo interno.
- No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representável é:

$$N_{inteiros} = 2^{32} = 4.294.967.296_{10}$$

Representação de inteiros

- 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 $\mathbf{mod(2^n)}$ em que "**n**" é o número de bits de representação
- O maior valor que um resultado aritmético pode tomar será portanto $\mathbf{2^n-1}$, sendo o valor inteiro imediatamente a seguir o valor zero (representação circular)

Representação de inteiros

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

11001011 + 00110111 = 1 $\boxed{\texttt{00000010}}$

carry

resultado com 8 bits

- No caso em que os operandos são do tipo ***unsigned***, o bit ***carry*** 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*** não tem qualquer significado e é ignorado

Representação em complemento para dois

- O método mais usado na codificação de quantidades inteiras com sinal (*signed*) é "complemento para dois"
- **Definição**: Se N é um número positivo, então N* é o seu complemento para 2 (complemento verdadeiro) e é dado por:

  $$N^* = 2^n - N$$

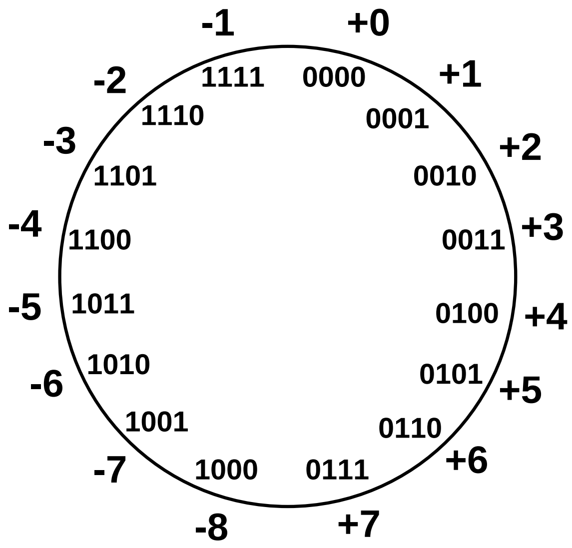
  em que “n” é o número de bits da representação
- **Exemplo**: determinar a representação de -5, com 4 bits

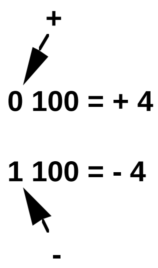
  $$N = 5_{10} = 0101_2$$

  $$2^n = 2^4 = 10000$$

  $$2^n - N = 10000 - 0101 = 1011 = N^*$$
- **Método prático**: inverter todos os bits do valor original positivo 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))`**

Representação em complemento para dois

- Uma quantidade de 32 bits codificada em complemento para 2 pode ser representada pelo seguinte polinómio:

  $$-(a_{31}.2^{31})+(a_{30}.2^{30})+...+(a_1.2^1)+(a_0.2^0)$$

  Onde o bit indicador de sinal (a_{31}) é multiplicado por -2^{31} e os restantes pela versão positiva do respetivo peso
- **Exemplo**: Qual o valor representado em base 10 pela quantidade 10100101_2, supondo uma representação com 8 bits e uma codificação em complemento para 2?
  - **R1**: $10100101_2 = -(1x2^7)+(1x2^5)+(1x2^2)+(1x2^0)$ $= -128 + 32 + 4 + 1 = -91_{10}$
  - **R2**: Complemento para 2 de **10100101 = 01011010 + 1** $= 01011011_2 = 5B_{16} = 91_{10}$ (o módulo da quantidade é **91**; logo o valor representado é -91_{10})

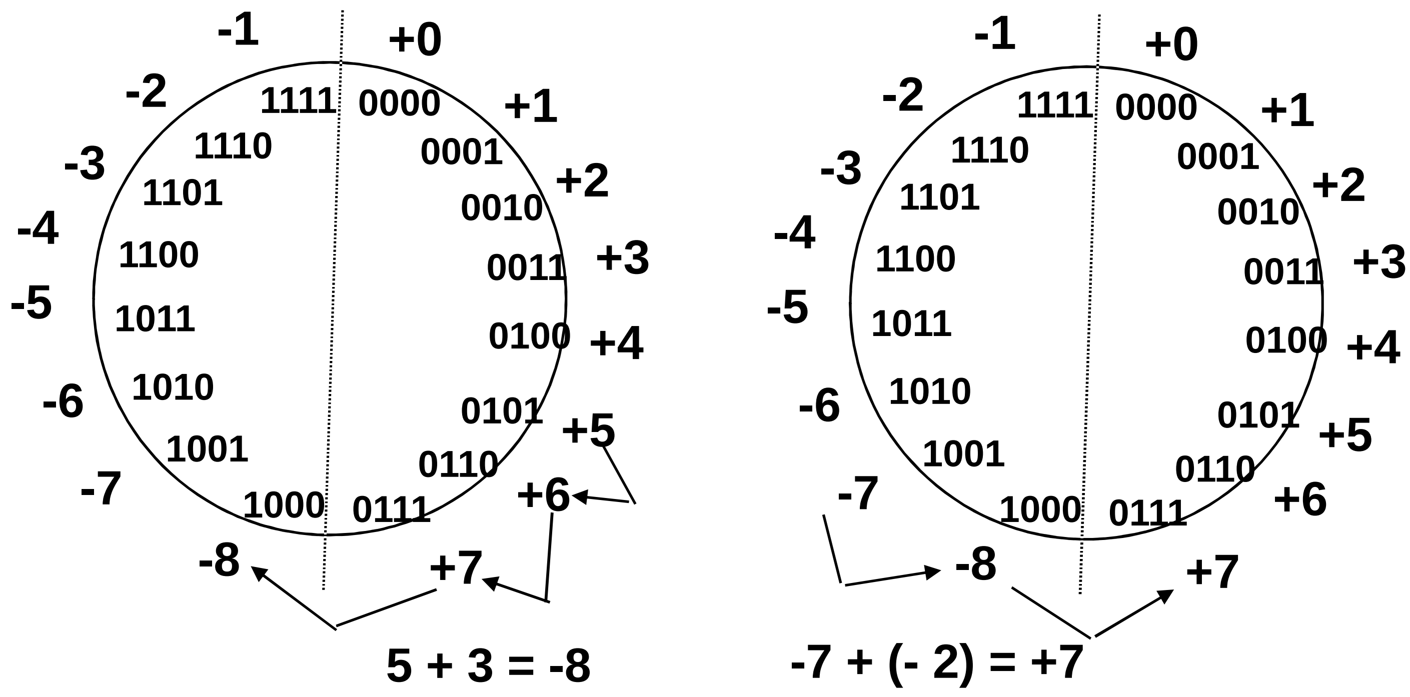
Representação em complemento para dois

- Exemplos de operações

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

	0 0 0 0 ← carry		1 1 1 1
5	0 1 0 1	-3	1 1 0 1
2	0 0 1 0	+ (-5)	=’S 1 0 1 1
7	0 0 1 1 1	-8	1 1 0 0 0
Sem overflow		Sem overflow	

	0 1 1 1		1 0 0 0
5	0 1 0 1	-7	1 0 0 1
3	≠’S 0 0 1 1	+ (-2)	≠’S 1 1 1 0
-8	0 1 0 0 0	7	1 0 1 1 1
Overflow		**Overflow**	

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

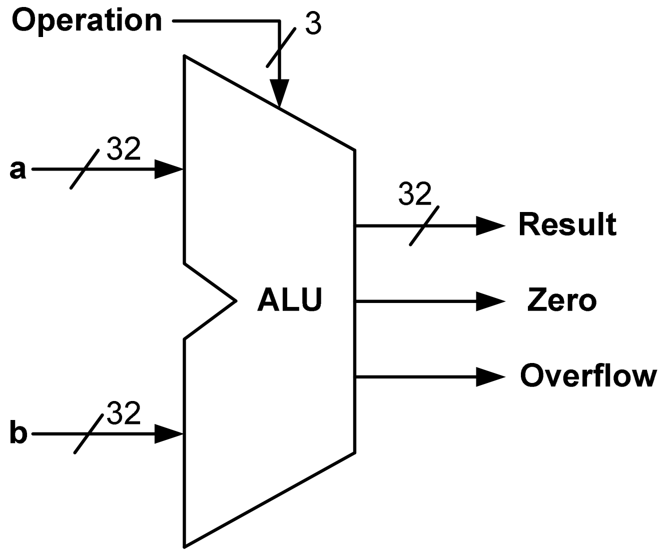
$$C_{n-1} \oplus C_n = 1$$

Overflow em operações aritméticas

- **Em operações 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*
- **Em operações com sinal:**
  - Quando $A + B > 2^{n-1}-1$ ou $A + B < -2^{n-1}$
    - $OVF = (C_{n-1}.\overline{C_n}) + (\overline{C_{n-1}}.C_n) = C_{n-1} \oplus C_n$
  - Alternativamente, não tendo acesso aos bits intermédios de *carry*, (R = A + B):
    - $OVF = R_{n-1}.\overline{A_{n-1}}.\overline{B_{n-1}} + \overline{R_{n-1}}.A_{n-1}.B_{n-1}$
- O MIPS apenas deteta *overflow* nas operações de adição com sinal e, quando isso acontece, gera uma exceção:
  - Como detetar *overflow* nas operações sem sinal?
  - Como detetar *overflow* antes da realização das operações com sinal, de modo a evitar a ocorrência da exceção?

Construção de uma ALU de 32 bits

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



Bloco funcional correspondente a uma ALU de 32 bits

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

Construção de uma ALU de 32 bits – VHDL

```
entity alu32 is
 port( a     :  in  std_logic_vector(31 downto 0);
       b     :  in  std_logic_vector(31 downto 0);
       oper  :  in  std_logic_vector(2  downto 0);
       res   :  out std_logic_vector(31 downto 0);
       zero  :  out std_logic;
       ovf   :  out std_logic);
end alu32;

architecture Behavioral of alu32 is
 signal s_res : std_logic_vector(31 downto 0);
 signal s_b   : unsigned(31 downto 0);
begin
 s_b  <= not(unsigned(b)) + 1 when oper = "110" else
            unsigned(b); -- complemento para 2 (se subtração)
 res  <= s_res;
 zero <= '1' when s_res = X"00000000" else '0';
 ovf  <= (not a(31) and not s_b(31) and s_res(31)) or
           (a(31) and s_b(31) and not s_res(31));
 --(continua)
```

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

Construção de uma ALU de 32 bits (continuação)

```vhdl
 process(oper, a, b, s_b)
 begin
  case oper is
     when "000" =>   -- AND
        s_res <= a and b;
     when "001" =>   -- OR
        s_res <= a or b;
     when "010" =>   -- ADD
        s_res <= std_logic_vector(unsigned(a) + s_b);
     when "110" =>   -- SUB
        s_res <= std_logic_vector(unsigned(a) + s_b);
     when "111" =>   -- SLT
        if(signed(a) < signed(b)) then
           s_res <= X"00000001";
        else
           s_res <= (others => '0');
        end if;
     when others =>
        s_res <= (others => '-');
  end case;
 end process;
end Behavioral;
```

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

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

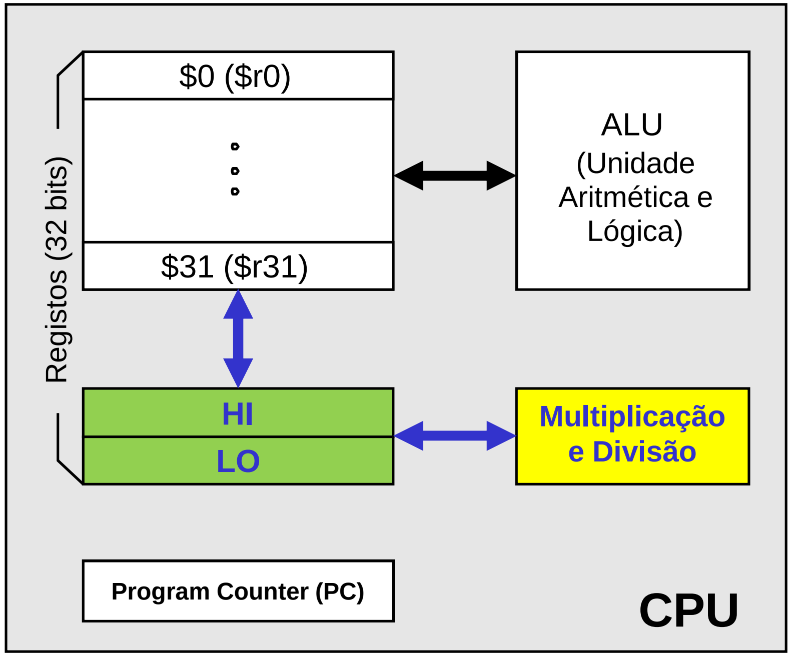
- No caso do MIPS, essas operações são asseguradas por uma unidade especial de multiplicação e divisão de inteiros

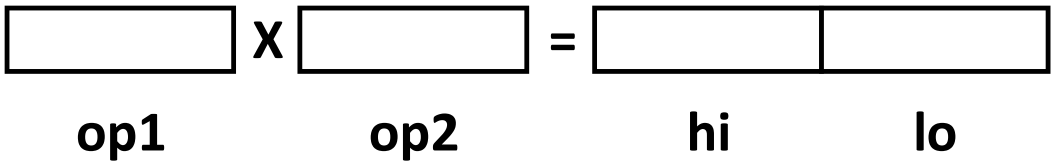
- Note-se que uma multiplicação que envolva **dois operandos de *n* bits** carece de um espaço de armazenamento, para o resultado, de **2*n bits**

- Tal implica que o **resultado**, no caso do MIPS, deverá ser armazenado com **64 bits**, o que determina a existência de **registos especiais** para esse mesmo armazenamento

A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação e a divisão são asseguradas por um módulo independente da ALU
- Os resultados da multiplicação e divisão são armazenados num par de registos especiais designados por **HI** e **LO**
- 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   $reg    # move from hi - Copia HI para $reg
mflo   $reg    # move from lo - Copia LO para $reg
```

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

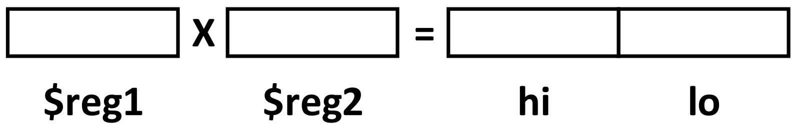
A Multiplicação de inteiros no MIPS

- Em *Assembly*, a multiplicação é efetuada pela instrução

```
mult    $reg1, $reg2    # Multiply (signed)
multu   $reg1, $reg2    # Multiply unsigned
```

  em que **$reg1** e **$reg2** 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)

```
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 de dividendo e divisor forem diferentes
  - O resto tem o mesmo sinal do dividendo

- Exemplo 1 (**`dividendo = -7`**, **`divisor = 3`**):

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

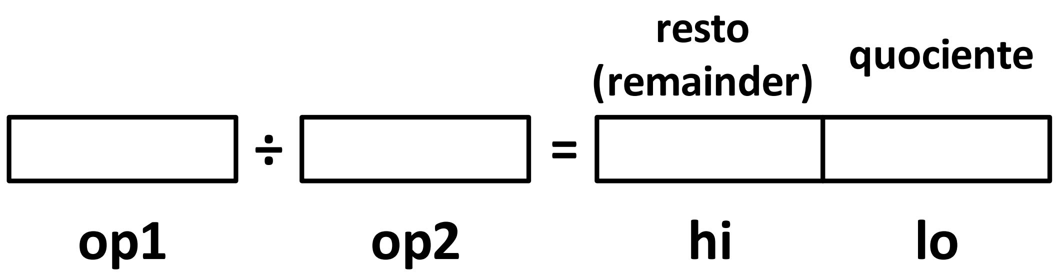
- Exemplo 2 (dividendo = 7, divisor = -3):

```
 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 apresentados para o caso da 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

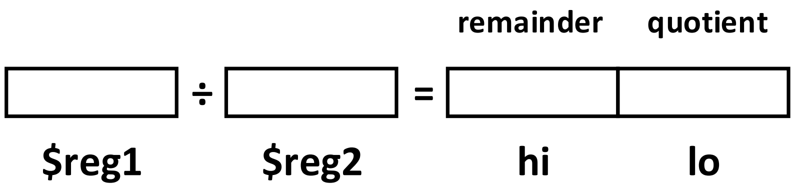


A Divisão de inteiros no MIPS

- Em *Assembly*, a divisão é efetuada pela instrução

```
div     $reg1, $reg2    # Divide (signed)
divu    $reg1, $reg2    # Divide unsigned
```

- em que **$reg1** é o dividendo e **$reg2** 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
```

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:
  - $\mathbf{00101011_2}$, **0xA5**, $\mathbf{10101101_2}$, **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 **`mult $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
```