**Arquitetura de Computadores I**
**2ª série de problemas – soluções**
7.11.2014

## I – Invocação de funções

**1.** Converta a função C para assembly do MIPS. Siga as convenções de invocação de funções do MIPS.

```
unsigned int sum(unsigned int n)
{
  if (n == 0) return 0;
  else return n + sum(n-1);
}
```

Versão *assembly*:

```
sum:    bgt     $a0, $0, cont
        mov     $v0, $a0
        jr      $ra
cont:   addi    $sp, $sp, -8
        sw      $ra, 4($sp)
        sw      $a0, 0($sp)
        addi    $a0, $a0, -1
        jal     sum
        lw      $ra, 4($sp)
        lw      $a0, 0($sp)
        addi    $sp, $sp, 8
        add     $v0, $a0, $v0
        jr      $ra
```

**2.** Nas aulas foi traduzida para assembly a seguinte versão recursiva do cálculo de factorial(n):

```
int fact (int n)
{
 if (n < 1) return (1);
 else return n * fact(n - 1);
}
```

Faça a tradução de uma versão iterativa de factorial(n). Indique o conteúdo do stack durante a execução da versão recursiva e da versão iterativa.

```
int fact (int n)
{
 int i, produto;
 produto = 1;
 for (i = 1; i <= n; i++)
 {
      produto = produto*i;
 }
```

```
 return (produto);
}
```

Versão *assembly*:

1ª versão: sendo i e produto variáveis da função, segue-se a convenção de as alocar a registos $s:

Alocação de Registos: $a0 – n; $s0 – i; $s1 - produto

```
        addi    $sp, $sp, -8
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        li      $s1, 1
        li      $s0, 1
loop:   bgt     $s0, $a0, end
        mul     $s1, $s1, $s0
        addi    $s0, $s0, 1
        jmp     loop
end:    mov     $v0, $s1
        lw      $s1, 4($sp)
        lw      $s0, 0($sp)
        addi    $sp, $sp, 8
        jr      $ra
```

Stack (versão iterativa):

| Address | Antes | | Durante | | Depois | |
|---|---|---|---|---|---|---|
| | $sp → | | | | $sp → | |
| -4 | | | | $s1 | | |
| -8 | | | $sp → | $s0 | | |

2ª versão: sendo a função terminal (não invoca nenhuma função), alocar as suas variáveis a registos $t, dispensando assim a salvaguarda no stack:

Alocação de Registos: $a0 – n; $t0 – i; $t1 - produto

```
        li      $t1, 1
        li      $t0, 1
loop:   bgt     $t0, $a0, end
        mul     $t1, $t1, $t0
        addi    $t0, $t0, 1
        jmp     loop
end:    mov     $v0, $t1
        jr      $ra
```

Stack (versão recursiva- – ver slides teórica):

| | Valor do argumento de invocação | | | | Retorno | | | |
|---|---|---|---|---|---|---|---|---|
| Address | n | n-1 | … | 0 | … | | | |
| $sp$_{old}$ → | n | n | | n | | n | n | |
| -4 | L1+8 | L1+8 | | L1+8 | | L1+8 | L1+8 | |
| -8 $sp$_{new}$ | | n-1 | | n-1 | | n-1 | | |
| -12 | | L1+8 | | L1+8 | | L1+8 | | |
| -16 | | | | | | | | |
| -20 | | | | | | | | |
| … | … | … | … | … | … | | | |
| -8(n-1) $sp | | | | 0 | | | | |
| | | | | L1+8 | | | | |
| -8n $sp$_{new}$ | | | | | | | | |

## II - **Aritmética inteira**

**3.** Represente em 2's complement com 16 bits os operandos e execute as operações a seguir indicadas:

(a) 3 + 12 0000 0000 0000 0011 + 0000 0000 0000 1100 = 0000 0000 0000 1111

(b) 13 - 2 0000 0000 0000 1100 + 1111 1111 1111 1110 = 0000 0000 0000 1011

(c) 5 - 6 0000 0000 0000 0101 + 1111 1111 1111 1010 = 1111 1111 1111 1111

(d) -7 - (-7) 1111 1111 1111 1001 - (1111 1111 1111 1001) =
1111 1111 1111 1001 + 0000 0000 0000 0111 = 0000 0000 0000 0000

4. Dispõe-se de um multiplicador série, análogo ao apresentado nas aulas, para inteiros *unsigned* representados em 4-bits. Pretende-se utilizá-lo para multiplicar 1011 por 1010.

a) Preencha a tabela indicando o valor dos registos a cada passo da execução do algoritmo de multiplicação e fazendo a respetiva descrição (shift left, shift right, add / no add).

b) Que alterações teria de introduzir no desenho do multiplicador para multiplicar valores em complemento para 2 utilizando o algoritmo de Booth? Assuma o mesmo conjunto de bits para o multiplicando e o multiplicador, mas representando agora valores em complemento para 2, e preencha de novo a tabela com a indicação dos vários passos do algoritmo de Booth.

c) Que modificações teria de introduzir no algoritmo de Booth e no esquema do multiplicador para poder utilizá-lo para efetuar multiplicações tanto *signed* como *unsigned*? Preencha de novo a tabela com a descrição dos passos da execução da multiplicação no seu multiplicador de Booth dos mesmos operandos *unsigned*.

**a) Unsigned 11 x 10**

| Produto | Multiplicando | Multiplicador | Descrição | Step |
|---|---|---|---|---|
| 0000 0000 | 0000 1011 | 1010 | Valores Iniciais | Step 0 |
| 0000 0000 | 0000 1011 | 1010 | 0 - Shift | |
| 0000 0000 | 000 1011 | 101 | 1 - Add | Step 1 |
| 0001 0110 | 000 1011 | 101 | Shift | Step 2 |
| 0001 0110 | 00 1011 | 10 | 0 - Shift | Step 3 |
| **0001 0110** | 01 011 | 1 | 1 - Add | Step 4 |

**Resultado: 0110 1110 $(110)_{10}$**

### b) Booth 2's complement -5 x -6

| Produto | Multiplicando | Multiplicador | Descrição | Step |
|---|---|---|---|---|
| 0000 0000 | 1111 1011 | 1010 | Valores Iniciais | Step 0 |
| 0000 0000 | 1111 1011 | 1010 | 00 - Shift | Step 1 |
| 0000 1010 | 1111 011 | 1010 | 10 - Sub | Step 2 |
| 0000 1010 | 111 011 | 1010 | Shift | Step 3 |
| 1111 0110 | 1110 11 | 101 | 01 - Add | Step 4 |
| 1111 0110 | 110 11 | 101 | Shift | Step 5 |
| **0001 1110** | 101 1 | 10 | 10 - Sub | Step 6 |

**Resultado: 0001 1110 (+30)**

### c) Booth Unsigned 11 x 10

1º - representar os operandos em complemento para 2 (juntar bit de sinal):
01011 e 01010
2º - gama de valores dos operandos: $(0...2^4-1)$ =>
=> Resultado < $(2^8-1)$ => representável em 9 bits

| Produto | Multiplicando | Multiplicador | Descrição | Step |
|---|---|---|---|---|
| 0 0000 0000 | 0 0000 1011 | 01010 | Valores Iniciais | Step 0 |
| 0 0000 0000 | 0 0001 011 | 01010 | 00 - Shift | Step 1 |
| 1 1110 1010 | 0 0001 011 | 01010 | 10 - Sub | Step 2 |
| 1 1110 1010 | 0 0010 11 | 0101 | Shift | Step 3 |
| 0 0110 1110 | 0 0010 11 | 0101 | 01 - Add | Step 4 |
| 0 0110 1110 | 0 0101 1 | 010 | Shift | Step 5 |
| 1 1011 1110 | 0 0101 1 | 010 | 10 - Sub | Step 6 |
| 1 1011 1110 | 0 1011 | 01 | Shift | Step 7 |
| 0 **0110 1110** | 0 1011 | 01 | 01 - Add | Step 8 |

**Resultado (unsigned): 0110 1110** (o resultado é representável em 8 bits, pelo que o 9º bit é desnecessário)

5. Pretende-se efetuar a divisão de inteiros de 4 bits usando um divisor idêntico ao apresentado nas aulas e no livro. Preencha a tabela indicando o valor dos registos a cada passo da execução do algoritmo de divisão e fazendo a respetiva descrição (shift left, shift right, sub). O valor do Divisor é 4 (0100, com 0000 bits à direita para o right shift*), o Dividendo é 6 (inicialmente colocado no registo *Resto*).

* - o número de shifts à direita terá de ser de 4 para preservar o valor do divisor. (há um erro no P&H Computer Organization and Design – o numero de iterações do ciclo indicado é errado para a divisão inteira).

| Quociente | Divisor | Resto | Descrição | Step |
|---|---|---|---|---|
| 0000 | 0100 0000 | 0000 0110 | Valores Iniciais | Step 0 |
| | | 1100 0110 | Resto = Resto-Divisor | Step 1 |
| 0000 | | 0000 0110 | Resto<0 => Resto+Divisor | Step 2 |
| | 0010 0000 | | Shift rigth Divisor | Step 3 |
| | | 1110 0110 | Resto = Resto-Divisor | Step 4 |
| 0000 | | 0000 0110 | Resto<0 => Resto+Divisor | Step 5 |
| | 0001 0000 | | Shift rigth Divisor | Step 6 |
| | | 1111 0110 | Resto = Resto-Divisor | Step 7 |
| 0000 | | 0000 0110 | Resto<0 => Resto+Divisor | Step 8 |
| | 0000 1000 | | Shift rigth Divisor | Step 9 |
| | | 1111 1110 | Resto = Resto-Divisor | Step 10 |
| 0000 | | 0000 0110 | Resto<0 => Resto+Divisor | Step 11 |
| | 0000 0100 | | Shift rigth Divisor | Step 12 |
| 0001 | | 0000 0010 | Resto = Resto-Divisor | Step 13 |

**Quociente = 0001;** **Resto = 0010**

## **III – Vírgula Flutuante** (representação standard IEEE)

**6.** a) Qual o valor em decimal do número representado em precisão simples como

1 01111101 00100000000000000000000

Sinal: **-**
Expoente: 01111101 = 125 – 127 = -2
Significando: 1,001 = 1,125 Valor = **$1{,}125 \times 2^{-2}$**

b) Converta o valor para precisão dupla

Expoente: -2 + 1023 = 111 1111 1110 + 011 1111 1111 = 011 1111 1101

**1 01111111101 00100000000000000000000000000000000000000000000000000**

**7.** Neste problema usa-se um formato de representação em 8-bits IEEE 754-like, normalisado com 1 *sign bit*, 4 *exponent bits*, e 3 *mantissa bits*. O expoente é codificado em excesso-7. A ordem dos campos no número é (sign, exponent, mantissa). É usado *unbiased rounding to the nearest even* como especificado no standard IEEE.

a) Codifique os seguintes números neste formato:
(a) $0.0011011_2$

**0 1010 110**

(b) $16.0_{10}$

**0 1011 000**

b) Some 1.011 + 0.0011011 indicando o valor dos bits de guarda, de arredondamento e do *sticky bit*

1.011
+ 0.0011011
1.1001011 **guard bit = 1; round bit = 0; sticky bit = 1**

d) Qual o valor em decimal de 1 1010 101
**- 1,625x2$^3$**

e) Indique para cada um dos pares de numeros seguintes qual o que representa o maior valor

**(1)** 0 0100 100 **e** 0 0100 111 **0 0100 111**

**(2)** 0 1100 100 **e** 1 1100 101 **0 1100 100**

**8.** Represente no formato IEEE precisão simples o valor -11/16 (-0.6875)

**1 01111110 01100000000000000000000**

**9.** A e B estão representados no formato IEEE 754 precisão simples.

A = 0 10001100 11000000000000000000000
B = 1 01111111 01000000001100000000001

a) Determine A+B. Indique também o valor dos bits de guarda e de arredondamento e do sticky bit.
ExpA – ExpB = 1101 = 13

-SignificandB = 010111111110011111111111

SignificandA + (-SignificandB)x2$^{-13}$ =

11100000000000000000000
00000000000001011 111111001111 1111111
11100 0000000001011 111111001111 1111111

SignificandA+B =11000000000001011111111 Guard bit = 0; Round bit = 0; Sticky bit = 1

**A+B = 0 10001100 11000000000001011111111**

b) Determine B*A

SignA*B = 1
ExpA+ExpB = 10001100 + 01111111 – 01111111 = 10001100
SignificandA*SignificandB =

10100000000110000000001
10100000000110000000001
10100000000110000000001
100011000001010100000000111

guard round sticky

Normalizar resultado: ExpA*B = ExpA+ExpB+1 = 10001101

**A*B = 1 10001101 00011000001010100000001**