Atividades - Assembly RISC-V

- Autor: Rafael Grossi

Atividade 1

Implemente um programa que declare um vetor de 5 posições e chame a função acima para multiplicar todos os elementos por 10.

```
.data
vetor:
    .word 10
    .word 20
    .word 30
    .word 40
    .word 50

.text
MultiplicaVetor:
    addi sp, sp, -16
    sw   s0, sp, 12
    sw   s1, sp, 8
    sw   s2, sp, 4
    sw   ra, sp, 0

    mv   s0, a0
    mv   s1, a1
    mv   s2, a2

for:
    beq  s0, zero, fim
    lw   a0, s1, 0
    mv   a1, s2
    call Multiplica
    sw   a0, s1, 0
    addi s1, s1, 4
    addi s0, s0, -1
    j    for
```

```
fim:
    lw   ra, sp, 0
    lw   s2, sp, 4
    lw   s1, sp, 8
    lw   s0, sp, 12
    addi sp, sp, 16
    ret

Multiplica:
    mv t0, a0
    li t1, 1

for1:
    beq a1, t1, m_end
    add a0, a0, t0
    addi a1, a1, -1
    j for1

m_end:
    ret

main:
    lui s0, %hi(vetor)

    addi a0, zero, 5
    # add a1, a0, s0
    addi a1, s0, %lo(vetor)
    addi a2, zero, 10

    call MultiplicaVetor

    ret
```

Atividade 2

Implemente a função `unsigned Multiplica(unsigned x, unsigned y)`.

```
.data
vetor:
    .word 10
    .word 20
    .word 30
    .word 40
    .word 50

.text
MultiplicaVetor:
    addi sp, sp, -16
    sw   s0, sp, 12
    sw   s1, sp, 8
    sw   s2, sp, 4
    sw   ra, sp, 0

    mv   s0, a0
    mv   s1, a1
    mv   s2, a2

for:
    beq  s0, zero, fim
    lw   a0, s1, 0
    mv   a1, s2
    call Multiplica
    sw   a0, s1, 0
    addi s1, s1, 4
    addi s0, s0, -1
    j    for

fim:
    lw   ra, sp, 0
    lw   s2, sp, 4
    lw   s1, sp, 8
    lw   s0, sp, 12
    addi sp, sp, 16
    ret

Multiplica:
    mv t0, a0
    li t1, 1
```

```
for1:
    beq a1, t1, m_end
    add a0, a0, t0
    addi a1, a1, -1
    j for1

m_end:
    ret

main:
    lui s0, %hi(vetor)

    addi a0, zero, 5
    # add a1, a0, s0
    addi a1, s0, %lo(vetor)
    addi a2, zero, 10

    call MultiplicaVetor

    ret
```

Atividade 3

Implemente a função `int SomaVetor(unsigned N, unsigned *v)` utilizando apenas os registradores t e a. Altere seu código anterior para chamar essa função no lugar da `MultiplicaVetor`.

```
.data
vetor:
    .word 10
    .word 20
    .word 30
    .word 40
    .word 50

.text
SomaVetor:
    li t0, 0
```

```
    for:
        beq a0, zero, end
        lw t1, 0(a1)
        add t0, t0, t1

        addi a1, a1, 4
        addi a0, a0, -1

        j    for
    end:
        mv a0, t0
        ret


main:
    lui s0, %hi(vetor)

    addi a0, zero, 5
    addi a1, s0, %lo(vetor)

    call SomaVetor

    ret
```

Atividade 4

Crie uma função `int TamanhoString()` que tenha uma string como variável local (deve ser guardada na pilha). Leia a string do teclado e retorne apenas o tamanho da string. Você deve criar e utilizar também uma função chamada `int strlen(char *s)` para calcular o tamanho da string de acordo com as convenções do simulador (igual você já fez anteriormente).

```
.text
TamanhoString:
    # stack begin
    addi sp, sp, -108
    sw   ra, 104(sp)
    sw   s0, 100(sp)

    mv   s0, sp
```

```
    # read string
    li   t0, 6
    mv   a0, s0
    li   a1, 100
    ecall

    # strlen(*str: a0)
    mv   a0, s0
    call strlen

    # stack end
    lw   s0, 100(sp)
    lw   ra, 104(sp)
    addi sp, sp, 108
    ret

strlen:
    mv   t0, a0
    li   t2, 10
    # check against space
    li   t3, 32

    strlen_loop:
        lbu  t1, 0(t0)
        beq  t1, zero, strlen_end
        beq  t1, t2, strlen_end
        beq  t1, t3, check_next

        addi t0, t0, 1
        j    strlen_loop

    check_next:
        lbu  t4, 1(t0)
        beq  t4, t2, strlen_end
        beq  t4, t3, strlen_end
        beq  t4, zero, strlen_end

        addi t0, t0, 1
        j    strlen_loop
```

```
    strlen_end:
        # size is end - start
        sub  a0, t0, a0
        ret

main:
    call TamanhoString

    mv   s0, a0

    li   t0, 1
    mv   a0, s0
    ecall
```

Atividade 5

Implemente uma calculadora com 2 operações: Soma e Subtração. Você deve ler um número, um caracter e outro número do teclado e imprimir o resultado da operação correspondente ao caracter lido. Utilize a estrutura Operacao para armazenar as funções.

```
.data
    operations:
        .word 43
        .word 0x2c #add_func
        .word 45
        .word 0x34 #sub_func

.text

add_func:
    add a0, a0, a1
    ret

sub_func:
    sub a0, a0, a1
    ret

main:
    # read first number
```

```
    li t0, 4
    ecall
    mv s0, a0

    # read operator character
    addi sp, sp, -8
    li t0, 6
    mv a0, sp
    li a1, 2
    ecall
    lbu s1, 0(sp)
    addi sp, sp, 8

    # read second number
    li t0, 4
    ecall
    mv s2, a0

    # load function pointer
    lui t0, %hi(operations)
    addi t0, t0, %lo(operations)

    li t1, 2

find_operation:
    beqz t1, end

    lw t2, 0(t0)
    beq t2, s1, operation_found

    # move to next operation
    addi t0, t0, 8
    addi t1, t1, -1
    j find_operation

operation_found:
    lw t3, 4(t0)

    # arguments
    mv a0, s0
    mv a1, s2
```

```
    # call function
    jalr ra, t3, 0

    # print result
    li t0, 1
    ecall

end:
    ret
```

Atividade 6

Implemente uma função recursiva que calcule o fatorial de um número. Seu programa deve ler um número de entrada e imprimir o fatorial dele após chamar a função. Utilize a função Multiplica que você já implementou anteriormente.

```
.text

multiply:
    # a0 x a1 saves in a0
    li t0, 0

multiply_loop:
    beqz a1, multiply_end
    add t0, t0, a0
    addi a1, a1, -1
    j multiply_loop

multiply_end:
    mv a0, t0
    ret

factorial:
    # save return address and argument
    addi sp, sp, -8
    sw ra, 4(sp)
    sw a0, 0(sp)

    # base case if n less than 1, return 1
```

```
    li t0, 1
    ble a0, t0, factorial_base

    # recursive case: n * factorial(n-1)
    addi a0, a0, -1
    call factorial

    # a0 now has factorial(n-1)
    # multiply by n (local var in stack)
    lw a1, 0(sp)

    # save factorial(n-1) result
    mv s0, a0

    # prepare multiply arguments
    mv a0, s0
    call multiply

    # restore and return
    lw ra, 4(sp)
    addi sp, sp, 8
    ret

factorial_base:
    li a0, 1
    lw ra, 4(sp)
    addi sp, sp, 8
    ret

main:
    # read number
    li t0, 4
    ecall

    # save input
    mv s0, a0

    # call factorial
    mv a0, s0
    call factorial
```

```
    # print result
    li t0, 1
    ecall

    # end
```