Assembly (RISC-V)

Chamadas de Programa

PrintInt: **1**
PrintFloat: **2**
PrintDouble: **3**
PrintString: **4**
ReadInt: **5**
    retorno (inteiro lido): **a0**
ReadFloat: **6**
    retorno (inteiro lido): **a0**
ReadDouble: **7**
    retorno (inteiro lido): **a0**
ReadString: **8**
    parâmetro 1 (memória onde a string será escrita): **a0**
    parâmetro 2 (buffer/tamanho da string): **a1**
AllocateHeap: **9**
    retorno (endereço na heap): **a0**
Exit: **10**
PrintChar: **11**
    retorno (inteiro lido): **a0**
ReadChar: **12**

Registradores

Instruções

la reg_dst, label: carrega o endereço de um label no reg. | *load address*
li reg_dst, num: carrega o valor *num* no reg. | *load immediate*
mv reg_dst, reg_src: copia o valor de um reg. para outro | *move*
add reg_dst, reg_src, reg_src: *reg_dst = reg_src + reg_src* (valores)
addi reg_dst, reg_src, num: *reg_dst = reg_src + num* | *add immediate*

Acesso á Memória

Tipos

.word: *int* (4 bytes)
.half: *short int* (2 bytes)
.byte: *char* (1 byte)
.asciz: *string* (chars)

Align

.align n: alinha dados na memória para caberem em 2^n bytes.

Bytes: **.align 0** | 2^0 = 1 bytes = 8 bits | *chars* e *string*
Half Words: **.align 1** | 2^1 = 2 bytes = 16 bits | *short ints*
Words: **.align 2** | 2^2 = 4 bytes = 32 bits | *ints* e *instruções*

Instruções

lw reg_dst, offset(reg_src): carrega no reg. o valor (4 bytes) no endereço de memória guardado no outro reg. | load word
lh reg_dst, offset(reg_src): carrega no reg. o valor (2 bytes) no endereço de memória | load half
lb reg_dst, offset(reg_src): carrega no reg. o valor (1 byte) no endereço de memória | load byte
sw reg_src, offset(reg_dst): salva na memória o valor (4 bytes) do reg. | store word
sh reg_src, offset(reg_dst): salva na memória o valor (2 bytes) do reg. | store half
sb reg_src, offset(reg_dst): salva na memória o valor (1 byte) do reg. | store byte

Branches

b reg_1, reg_2, label: compara os valores de *reg_1* e *reg_2* e se for verdadeiro faz um jump para o label especificado

Instruções "puras":
- **beq**: branch if *equal* than
- **bne**: branch if *not equal* than
- **blt**: branch if *less* than
- **bge**: branch if *greater or equal* than

Pseudo-instruções:
- **bgt**: branch if *greater* than
- **ble**: branch if *less or equal* than

Exemplo 1 - strcpy() dinâmica

```
                .data # Dados do programa
                .align 0
str_src:        .asciz "Oi mae!!" # 9 bytes

                .align 2
ponteiro:       .word # Ponteiro que vai apontar para um espaço na heap

                .text # Código do programa
                .align 2 # Instruções de 32 bits
                .globl main # Ponto de entrada do código

main:           # Main
                la s0, str_src # Guardando em s0 o endereço do início de str_src
                li t1, 0 # Contador inicializado com 0

loopContChar:   # Conta o número de caracteres da string
                lb t0, 0(s0) # Armazenando em t0 o char lido de s0
                addi s0, s0, 1 # Avançando o ponteiro da string
                addi t1, t1, 1 # Incrementando o contador
                bne t0, zero, loopContChar  # Voltando para o inicio do loop caso não
                                              tenhamos chegado no \0

heap:           # Alocação dinâmica na heap
                mv a0, t1 # Passando como argumento da função (espaço a ser alocado) a
                quantidade de caracteres da string (incluindo \0)
                li a7, 9 # Instrução para alocar espaço na memória
                ecall # Chamada do programa - a0 agora guarda o endereço do 1º byte
                        endereçado

                # Fazendo ponteiro apontar para a heap
                la t2, ponteiro # Guardando em t2 o endereço do ponteiro
                sw a0, 0(t2) # Colocando o conteúdo de a0 na posição de memória
                               apontada por t2

copia:          # Preparar para copiar
                la s0, str_src # Colocando em s0 o endereço da string str_src
                lw s1, 0(t2) # Colocando em s1 o endereço do espaço na heap

loopCopia:      # Copiando string para a heap
                lb t0, 0(s0) # Carregando em t0 o caractere da string
                sb t0, 0(s1) # Escrevendo na heap o caractere guardado em t0
                beq t0, zero, fim # Vai para o fim do programa caso chegue no \0
                addi, s0, s0, 1 # Avançando para o próximo caractere da string
                addi s1, s1, 1 # Avançando para o próximo espaço na heap
                j loopCopia # Volta para o início do loop
```

```
fim:            # Imprime a string na heap e encerra o programa
                lw s1, 0(t2) # Colocando em s1 o endereço do espaço na heap
                mv a0, s1 # Colocando em s0 o endereço do espaço na heap
                li a7, 4 # Instrução para imprimir string
                ecall # Chamada do programa

                li a7, 10 # Instrução para encerrar o programa
                ecall
```

Exemplo 2 - Fatorial com função

```
                # Fatorial com função (jal e jr)
                .data # Dados do programa
                .align 0 # 1 byte = 4 bits
stringEnt:      .asciz "Digite o numero: "
stringSaida:    .asciz "Fatorial calculado: "

                .text # Código do programa
                .align 2 # Instruções/palavras de 32 bits
                .globl main # Ponto de entrada do programa
main:
                # Imprimindo string de entrada
                li a7, 4 # Instrução para imprimir string
                la a0,  stringEnt # Parâmetro da instrução: stringEnt
                ecall # Chamada do programa

                # Recebendo número do usuário
                li a7, 5 # Instrução para ler o valor digitado pelo usuário
                ecall # Chamada do programa, o valor digitado é armazenado em a0

                mv s0, a0 # Colocando o valor digitado pelo usr. (a0) em s0

                jal fatorial # Desvia para o label fatorial e salva a próxima linha como return
                             address

                mv s1, a1 # Colocando o valor retornado pela função fatorial (a1) em s1

                # Imprimindo o resultado
                li a7, 4 # Instrução para imprimir string
                la a0, stringSaida # Parâmetro da instrução: stringSaida
                ecall # Chamada do programa

                li a7, 1 # Instrução para imprimir inteiro
                mv a0, s1 # Parâmetro da função: resultado do fatorial (s1)
                ecall # Chamada do programa

                li a7, 10 # Instrução para encerrar o programa
                ecall # Chamada do programa
```

```
# Função fatorial
# a0: número a ser calculado
# a1: resultado (fatorial do número escolhido)
fatorial:
        li s1, 1 # Resposta
        mv t0, a0 # Copiando o número a ser calculado (a0) para t0
        beq  t0, zero, fimFat
loopFat:
        mul s1, s1, t0 # res = res * n
        addi t0, t0, -1 # n--
        bne t0, zero, loopFat # Volta para o início do loop caso t0 (n) seja diferente
                                                de 0
fimFat:
        mv a1, s1 # Colocando a resposta (s1) em a1
        jr ra # Voltando para o return address
```

Exemplo 3 - BubbleSort

```
        .data
        # Dados
        .align 1 #2^1 bytes = 2 bytes
vetor:  .half 7, 5, 2, 1, 1, 3, 4 # Vetor de (meio) inteiros a ser ordenado

        .text
        # Programa (Bubble Sort)
        .align 2 # Instruções de 32 bits
        .globl main
        # Main
        # Colocando em s0 o endereço do vetor
main:   la a0, vetor
        mv s0, a0

        # Colocando em t0 o contador i (loop externo), inicializado com o valor -1
        li t0, -1

        # Colocando em s1 o tamanho do vetor-2 (5), pois iremos acessar a posição
        j+1, com j indo até 5 (j+1 = posição 6 do vetor)
        li t1, 5
        mv s1, t1


# Loop Externo: percorre todo o vetor, executando o loop interno cada vez
loop_externo:  bgt t0, s1, fim # Vai para fim se i > 6
        # Senão:
        addi t0, t0, 1 #i++
        li t1, 0 # Inicializando contador do loop interno (j) com o valor 0
```

```
# Loop Interno: compara elementos do vetor 2 a 2 e os troca se necessário
loop_interno:   bgt t1, s1, loop_externo # Se j > tam-1, volta para o loop externo
                # Senão:
                add t2, t1, t1 # O offset (index) avançará de 2 em 2 bytes
                add t3, s0, t2 # Colocando em t3 o endereço de vetor[j]

                lh t4, 0(t3) # Carregando em t4 vetor[j] (conteúdo de t3 + 0 bytes)
                lh t5, 2(t3) # Carregando em t5 vetor[j+1] (conteúdo de t3 + 2 bytes)

                addi t1, t1, 1 # j++

                bgt t4, t5, swap # Se vetor[j] > vetor[j+1], swap

                j loop_interno # Volta para o começo do loop interno

# Swap
swap:           sh t4, 2(t3) # Carregando vetor[j] em vetor[j+1]
                sh t5, 0(t3) # Carregando veotr[j+1] em vetor [j]

                j loop_interno # Volta para o começo do loop interno

# Fim do programa
fim:
                li a7, 10 # Instrução para encerrar o programa
                ecall
```