Eduardo Bischoff Grasel - 22200355
Universidade Federal de Santa Catarina - UFSC

Organização de Computadores

Florianópolis
2024

Objetivo Geral

Ao final desse projeto deveremos tem uma noção mais profunda em relação a subrotinas (funções em assembly mips), e como utilizar coisas como stack pointer e chamadas recursivas, o registrador $ra, passada de parâmetros através dos registradores $a0 até $a3, e o retorno do valor de funções, indo de $v0 e $v1.

Projeto 1:

Iniciamos o projeto 1 com a declaração do .data, que neste caso serão apenas três words: a, b, result, as duas primeiras para guardar os valores ditos pelo usuário, e a última como resultado.

```
.data
    a: .word 0
    b: .word 0
    result: .word 0
```

Para o início do .text, escolhemos declarar explicitamente a main, que será responsável por receber os inputs do usuário, chamar a função recursiva de somas sucessivas, e por fim mostrar o resultado e encerrar o programa.

```
.text
main:
    # Leitura dos números a e b
    li $v0, 5
    syscall
    move $t0, $v0 # Salva a em $t0

    li $v0, 5
    syscall
    move $t1, $v0 # Salva b em $t1

    # Chama a função de multiplicaçãoo recursiva
    move $a0, $t0 # Passa a para $a0
    move $a1, $t1 # Passa b para $a1
    jal mult_recursive

    # Salva o resultado em result
    move $s0, $v0

    # Imprime o resultado
    li $v0, 1
    move $a0, $s0
    syscall

    # Fim do programa
    li $v0, 10
    syscall
```

Agora entraremos na parte característica do trabalho, a função recursiva, como podemos ver no código acima, usamos a chamada de função jal juntamente com uma label para se referir à ela, os argumentos foram passados acima nos registradores corretos ($a0, $a1), através da instrução move.

Indo para dentro da função, teremos, em primeira análise, que reservar um espaço na pilha, stack pointer, para as informações passadas para a função e o seu retorno, no caso utilizaremos 12 bytes, e aguardamos os valores necessários na pilha.

```
# Função de multiplicação recursiva
mult_recursive:

    # Preserva o endereço de retorno e os registradores $a0 e $a1 na pilha
    addi $sp, $sp, -12
    sw $ra, 0($sp)
    sw $a0, 4($sp)
    sw $a1, 8($sp)
```

Faremos agora os condicionais para os casos base, caso algum dos valores, a ou b, seja 1 ou 0, caso um deles seja um, retornaremos o valor do maior multiplicando sem alterações, entretanto caso algum deles seja 0, retornaremos 0.

```
# Caso base: se b for 1, retorne a
beq $a1, 1, end_mult_recursive_a
# Caso base: se b for 0, retorne 0
beq $a1, $zero, end_mult_recursive_0
# Caso base: se a for 0, retorne 0
beq $a0, $zero, end_mult_recursive_0
# Caso base: se a for 1, retorne b
beq $a0, 1, end_mult_recursive_b
```

Passaremos agora para o funcionamento das somas sucessivas, primeiramente diminuímos 1 do b e chamamos a função novamente, abaixo da recursividade teremos a lógica, iniciamos movendo o valor de $v0 (retorno padrão da função), para um registrador temporário $t1, então somamos esse valor com o valor contido em $a0 (a) com o $t1 e guardamos ele em $v0 e carregamos o valor de cada uma das variáveis iniciais através do lw. Em última análise liberamos o espaço da stack que foi ocupada, retornando a função.

```
# Decrementa b
addi $a1, $a1, -1
# Chama recursivamente a função, passando a e b-1
jal mult_recursive
# Salva o resultado temporário
move $t1, $v0
# Adiciona a a $t1 (resultado temporário)
add $v0, $a0, $t1

# Restaura o endereço de retorno e os registradores $a0 e $a1
lw $ra, 0($sp)
lw $a0, 4($sp)
lw $a1, 8($sp)
addi $sp, $sp, 12
# Retorna
jr $ra
```

Ao fim da explicação, faltou apenas explicar as labels dos beq do caso base. No caso de algum dos dois valores ser 0, apenas limparmos a stack e retornaremos 0 como resultado em $v0.

```
end_mult_recursive_0:
    move $v0, $zero
    addi $sp, $sp, 12
    # Retorna
    jr $ra
```

Caso algum dos dois operandos for igual a 1, apenas moveremos o valor contrário do que vale 1, e retornaremos esse valor em $v0.

```
end_mult_recursive_a:
    # Restaura o endereço de retorno e o registrador $a0
    lw $ra, 0($sp)
    lw $a0, 4($sp)
    move $v0, $a0
    addi $sp, $sp, 12
    # Retorna
    jr $ra
```

```
end_mult_recursive_b:
    # Restaura o endereço de retorno e o registrador $a1
    lw $ra, 0($sp)
    lw $a1, 8($sp)
    move $v0, $a1
    addi $sp, $sp, 12
    # Retorna
    jr $ra
```

Dessa forma encerramos o programa, funcionando com todos os casos base e somas sucessivas funcionando.

```
10
1
10
-- program is finished running --


Reset: reset completed.

1
10
10


7
6
42
-- program is finished running --


Reset: reset completed.

6
7
42
-- program is finished running --
```

```
-- program is finished running --

10
0
0
-- program is finished running --


Reset: reset completed.

0
5
0
-- program is finished running --
```

Projeto 2

Começamos no .data declarando o “vetor” como uma sequência de words na memória, em seguida declaramos n: tamanho - 1 do array, pois ele iniciará em 0

```
.data
array:      .word   11, 2, 3, 14, 15 # array
N:          .word   4 #  tamanho do array (n-1)
```

Indo para o .text, dentro da main iniciamos os carregamentos do array e do tamanho nos registradores $a0 e $a1 simultaneamente, isso é, para enviá-los como parâmetros para a função, em seguida chamamos a função recursiva soma array.

```
.text
main:
    # Configurar registradores
    la $a0, array       # Carregar endereço base do array em $a0
    lw $a1, N           # Carregar N em $a1

    # Chamada da função soma_array
    jal soma_array
```

Agora já dentro da função, começamos alocando espaço para o registrador de retorno no stack pointer, seguidamente carregamos o tamanho do array e verificamos se é igual a 0, caso seja, retornaremos 0.

Iniciaremos a recursão, diminuindo de $a1 (tamanho) uma unidade, em seguida fazemos a re chamada da função, está responsável pela recursividade. Abaixo dela carregarmos em $t2 o primeiro elemento do array em $a0, em seguida adicionamos 4 ao $a0 para acessar a próxima posição, por último somamos $v0 com $v0 e $t2, lembrando que $v0 é o valor padrão retornado pela função. Pronto, agora podemos carregar o $ra, desalocar o espaço ocupado pelo stack pointer e retornar a função, que será mostrada através de uma syscall fora no main.

```
soma_array:
    # Configurar a pilha
    addi $sp, $sp, -4  # Alocar espaço para o retorno
    sw $ra, 0($sp)      # Salvar o endereço de retorno

    # Carregar n em $t1
    add $t1, $a1, $zero     # Carregar n em $t1

    # Condição de parada da recursão (n == 0)
    beqz $t1, base_case # Se n == 0, vá para base_case

    # Recursão (array[0] + soma_array(array + 1, n - 1))
    addi $a1, $a1, -1   # Decrementar n
    jal soma_array      # Chamada recursiva
    lw $t2, 0($a0)    # Carregar array[0] em $t2
    addi $a0, $a0, 4   # Avançar para o próximo elemento do array
    add $v0, $t2, $v0   # Adicionar array[0] ao resultado da chamada recursiva

    # Limpar a pilha e retornar
    lw $ra, 0($sp)      # Restaurar endereço de retorno
    addi $sp, $sp, 4   # Desalocar espaço para o retorno
    jr $ra              # Retornar
```

```
base_case:
    # Caso base: retorna 0
    li $v0, 0           # Carrega 0 no resultado
    jr $ra              # Retorna
```

```
.data
array:      .word   11, 2, 3, 14, 15 # array
N:          .word   4 #  tamanho do array (n-1)
```

11 + 2 + 3 + 14 + 15 =

45

```
45
-- program is finished running --
```

```
.data
array:      .word   20, 5, 54, 14, 35, 67 # array
N:          .word   5 #  tamanho do array (n-1)
```

20 + 5 + 54 + 14 + 35 + 67 =

195

```
195
-- program is finished running --
```

Conclusão

Ao fim do projeto conseguimos um entendimento a cerca de funções em assembly, entretanto a parte mais importante foi sem dúvidas o entendimento do stack pointer e a pilha em assembly, fora isso o maior desafio foi conseguir utilizar as chamadas recursivas sem quebrar o programa, considerando que a ordem a ser feita o decremento e somatório antes e depois da chamada da função dentro da função são muito confusas.