



UNIVERSIDADE
FEDERAL DE
SERGIPE

Busca Binária em MIPS Assembly

.....

COMP0415 - Arquitetura de Computadores - T03

Gustavo Gomes Tavares

Índice

1. Objetivo
2. A busca binária
3. Implementação recursiva em C
4. Implementação em MIPS Assembly
 - 4.1 .data
 - 4.2 A função "main"
 - 4.3 A função "bSearch"
 - 4.4 funções auxiliares
5. Conclusões
6. Referências

Objetivos do trabalho

- Aprender sobre Assembly e, mais especificamente, MIPS Assembly;
- Colocar em prática os conhecimentos da disciplina de Arquitetura de Computadores;
- Se divertir aprendendo.

A busca binária

A busca binária é um algoritmo de busca para vetores ordenados de complexidade de pior caso $O(\log n)$ que pode ser implementado de forma iterativa ou recursiva.[1]

Ela é ideal para buscar itens em vetores ordenados de forma eficiente, com um pior caso menor que a busca sequencial ($O(N)$).

Seu funcionamento consiste em:

- Obter o pivô do vetor a ser pesquisado
- Caso o item seja o pivô, retorna a posição desse item. Caso seja maior que o pivô, pesquisa na segunda parte da lista (de pivô+1 a tamanho-1), e caso seja menor, pesquisa na primeira parte (de início a pivô-1).[1][2]

A busca binária

Elemento desejado: 80

Início = 0;

Final = $10 - 1 = 9$;

Vetor inicial (pivô = -1)

p

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | 60 | 80 | 98 |
|---|---|---|---|----|----|----|----|----|----|

A busca binária

Elemento desejado: 80

Início = 0;

Final = 10 - 1 = 9;

É feito o cálculo do pivô:

$$\text{pivô} = \frac{(\text{início} + \text{final})}{2} = \left\lfloor \frac{0 + 9}{2} \right\rfloor = 4$$

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| p | | | | | | | | | |
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | 60 | 80 | 98 |

A busca binária

Elemento desejado: 80

Início = pivô + 1 = 6;

Final = 10 - 1 = 9;

Como $\text{vetor}[5] = 13$ e $13 < 80$, a busca ocorre apenas na segunda parte do vetor:

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | 60 | 80 | 98 |
|---|---|---|---|----|----|----|----|----|----|

A busca binária

Elemento desejado: 80

Início = 6;

Final = 9;

O pivô é recalculado:

$$\text{pivô} = \frac{(\text{início} + \text{final})}{2} = \lfloor \frac{6+9}{2} \rfloor = 7$$

| | | | | | | | | | |
|---|---|---|---|----|----|----|--------------------|----|----|
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | ^p 60 | 80 | 98 |
|---|---|---|---|----|----|----|--------------------|----|----|

A busca binária

Elemento desejado: 80

Início = pivô + 1 = 8;

Final = 9;

Como vetor[7] = 60 e $60 < 80$, a busca ocorre novamente apenas na segunda parte do vetor:

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | 60 | 80 | 98 |
|---|---|---|---|----|----|----|----|----|----|

A busca binária

Elemento desejado: 80

Início = 8;

Final = 9;

O pivô é recalculado mais uma vez:

$$\text{pivô} = \frac{(\text{início} + \text{final})}{2} = \left\lfloor \frac{8+9}{2} \right\rfloor = 8$$

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | 60 | 80 | 98 |
|---|---|---|---|----|----|----|----|----|----|

p

A busca binária

Elemento desejado: 80

Início = 8;

Final = 9;

Como o elemento do pivô é igual ao elemento desejado, é possível afirmar que o pivô está na posição 8

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| 2 | 6 | 8 | 9 | 13 | 15 | 20 | 60 | 80 | 98 |
|---|---|---|---|----|----|----|----|----|----|

Implementação recursiva em C

Aqui está a implementação recursiva da busca binária em C:

```
1  int binSearchRec(int valor, int *array, int inicio, int
   ↪  final){
2      if(inicio<=final){
3          int pivo = (final+inicio)/2;
4          if(array[pivo] == valor) return pivo;
5          if(array[pivo] > valor)
6              return binSearchRec(valor,array,inicio,pivo-1);
7          else
8              return binSearchRec(valor,array,pivo+1,final);
9      }
10     return -1;
11 }
```

Implementação em MIPS Assembly

A implementação em MIPS Assembly utiliza recursividade, a fim de compreender melhor o uso da *stack*, dos registradores **\$ra** (return address) e **\$sp** (stack pointer), bem como o uso da *stack* para armazenar dados e endereços ao longo da execução do código.

Para a implementação, os registradores foram utilizados consistentemente para armazenar os seguintes valores:

\$s0 = endereço do array

\$s1 = valor a ser procurado

\$t2 = delimitador inicial(low)

\$t3 = delimitador final

\$t4 = pivo $[(\text{inicio} + \text{final})/2]$

\$t5 = valor armazenado no pivo

\$t6 = pivo * 4 (offset)

\$t7 = end. do array + offset do pivo

\$s3 = valor final

Implementação em MIPS Assembly

A seção `.data`

A seção `.data` do código ficou da seguinte maneira:

```
1  .data
2      #array contendo valores a serem buscados (ordenado)
3      array: .word
           ↪ 1,2,8,17,20,40,42,60,75,80,81,82,84,90,125,162,200,4096,8192
4      tamArray: .word 19 # tamanho do array
5      lineFeed: .asciiz "\n" #quebra de linha
6      fraseInicial: .asciiz "Insira o valor que deseja procurar: "
7      fraseFinal: .asciiz "O valor esta armazenado na posicao "
8      fraseNaoEncontrado: .asciiz " (nao encontrado)"
```

Implementação em MIPS Assembly

A função "main"

A primeira parte da função main carrega o endereço do array (**la**) e pede ao usuário o valor a ser procurado, com chamadas de sistema (**syscall**) para exibir o texto inicial e solicitar o valor de um inteiro.

A função também carrega o valor **0** ao registrador de delimitador inicial, e (**tamanho - 1**) ao registrador de delimitador final pois a busca começa com o pivo inicial calculado com $\frac{0 + (\text{tamanho} - 1)}{2}$.

Após isso, o valor em **\$s3** (valor final) é definido como **-1** por padrão, pois caso o valor procurado não esteja no vetor, basta imprimir **-1**.

Por fim, a busca é iniciada, com o valor de retorno da busca armazenado em **\$ra**.

Implementação em MIPS Assembly

A função "main"

```
1  main:
2      la $s0, array    #carrega o array
3
4      #carrega e imprime a frase inicial
5      la $a0, fraseInicial
6      li $v0, 4 #código de syscall para imprimir string
7      syscall
8
9      #le o valor digitado e o coloca em $s1
10     li $v0, 5 #código de syscall para armazenar inteiro
11     syscall
12     move $s1, $v0
13
14     move $t2, $zero    #Aloca o delimitador inicial (0) em $t2
```

Implementação em MIPS Assembly

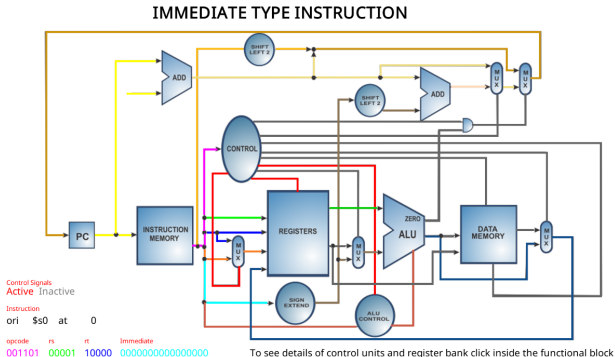
A função "main"

```
15  # obtem o tamanho do array -1 e o coloca em $t3
16  lw $t3, tamArray
17  subi $t3, $t3, 1
18
19  #define registrador de valor final como -1 (não encontrado)
20  li $s3, -1
21  #inicia a busca
22  jal bSearch
```

Implementação em MIPS Assembly

A função "main"

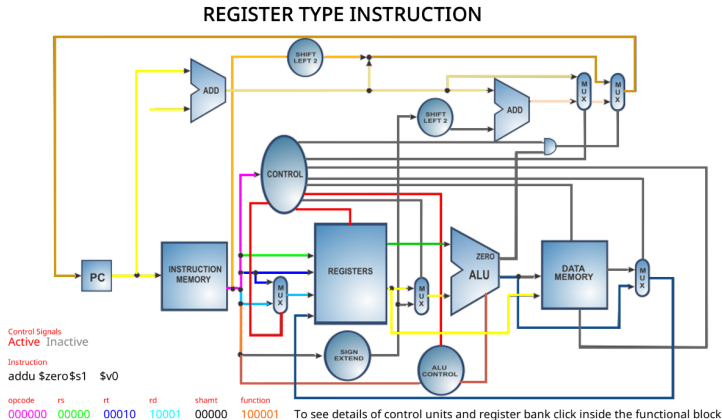
Os sinais e módulos utilizados para a pseudoinstrução **la** (load address), dividida em **lui** e **ori**, bem como para as instruções tipo I podem ser visualizados a partir do plugin X-Ray do simulador MARS:



Implementação em MIPS Assembly

A função "main"

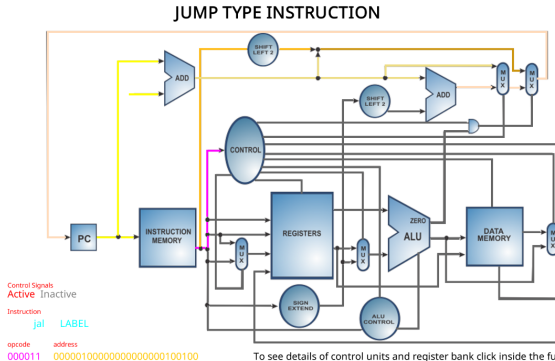
Para instruções tipo R, como a pseudoinstrução **move**, que move um endereço de um registrador para outro, a visualização do Xray é:



Implementação em MIPS Assembly

A função "main"

Para instruções tipo J, como o **jal** (jump-and-link), que além de pular para o endereço da função determinada, armazena o endereço de retorno do ponto atual de execução, os sinais executados são:



Implementação em MIPS Assembly

A função "main"

A segunda parte da função main é executada após a busca binária ser concluída, e as informações finais são impressas. Primeiramente, é impressa uma quebra de linha, e depois a frase final, seguida da posição em que o valor desejado está armazenado no vetor.

Caso o valor seja -1, indicando que o valor não está armazenado no vetor, o programa imprime ainda um texto de "não encontrado" para fácil visualização do resultado.

Por fim, uma chamada de sistema encerra o programa.

Implementação em MIPS Assembly

A função "main"

```
23  #impressão da frase final e da posicao do numero procurado
24  la $a0, lineFeed    # '\n'
25  li $v0, 4
26  syscall
27
28  la $a0, fraseFinal   #imprime a frase final...
29  syscall
30
31  move $a0, $s3
32  li $v0, 1            #...e a posição do valor desejado
33  syscall
34
35  beq $s3, -1, naoEncontradoTexto
36  li $v0, 10 #encerra o programa
37  syscall
```

Implementação em MIPS Assembly

A função "*bSearch*"

Essa função é a parte recursiva da busca, responsável por armazenar o endereço de retorno na *stack*, utilizando o *\$sp*.

Caso o delimitador inicial seja maior que o delimitador final, indicando que o valor não foi encontrado no vetor, o fluxo do programa é desviado para a função que finaliza a execução com valor não encontrado. Caso contrário, o fluxo continua.

Após essa condicional, o valor do pivô é calculado, com a soma do valor inicial com o valor final, e a divisão desse resultado por 2 (utilizando *shift right* por eficiência).

Implementação em MIPS Assembly

A função "bSearch"

Com o pivô calculado, é necessário determinar o offset no array. Considerando que cada inteiro possui 4 bytes, o valor do pivô é multiplicado por 4 com um *shift left*.

O endereço do pivô no array é obtido a partir da soma do endereço do array (em `$s0`) com o offset do pivô.

Caso o valor do pivô seja o valor desejado, a função para aqui. Caso contrário, um de dois caminhos é tomado:

- Se $[pivo] > valDesejado$, desvio para *pivoMaiorQueValor*
- Se $[pivo] < valDesejado$, desvio para *pivoMenorQueValor*

Implementação em MIPS Assembly

A função "bSearch"

```
1  bSearch:
2      addi $sp,$sp, -4    #aloca espaço na stack para armazenar o
   ↪   return address (em $ra)
3      sw $ra, 0($sp)     #armazena o valor de $ra na stack
4
5      #caso base: valor não encontrado.
6      bgt, $t2,$t3, finalizar #se del. inicial > del. final, finaliza
7
8      add $t4, $t2, $t3   # pivo = inicio+final
9      srl $t4, $t4, 1     # pivo = pivo/2
10
11     sll $t6, $t4, 2      #offset pivo (pivo*4)
12     add $t7,$s0,$t6      #array+offset pivo
13     lw $t5, 0($t7)       #carrega valor do pivo no registrador
```

Implementação em MIPS Assembly

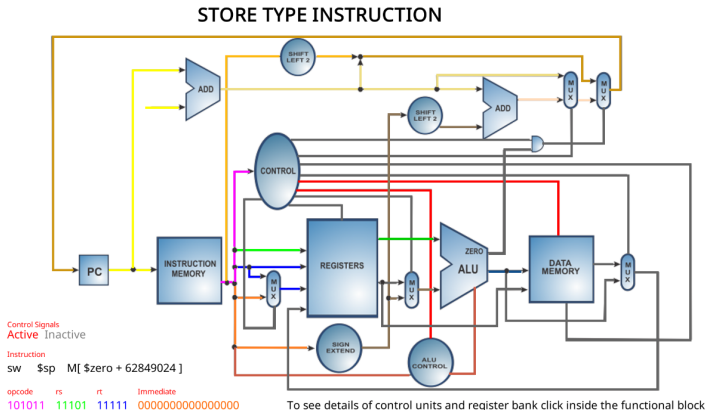
A função "bSearch"

```
14  beq $t5, $s1, encontrado #se o valor do pivo for o valor
    ↳ desejado, desvia para finalizar função
15
16  bgt $t5, $s1, pivoMaiorQueValor # se o valor do pivo for maior
    ↳ que o valor desejado, desvia para determinar novos
    ↳ delimitadores
17
18  blt $t5, $s1, pivoMenorQueValor # se o valor do pivo for menor
    ↳ que o valor desejado, desvia para determinar novos
    ↳ delimitadores
```

Implementação em MIPS Assembly

A função "bSearch"

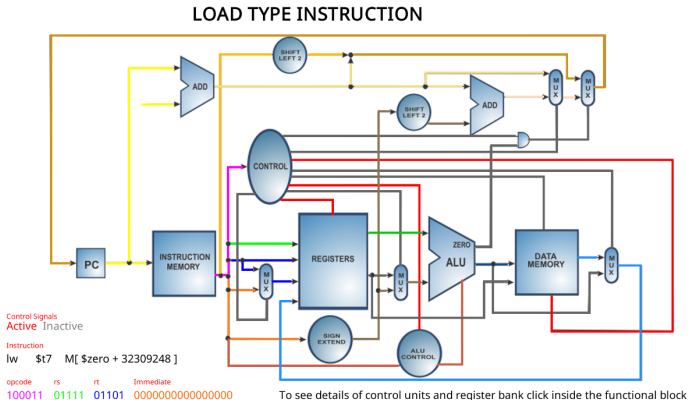
O diagrama gerado pelo MIPS Xray para a instrução **sw**, que armazena uma palavra em uma posição na memória, é:



Implementação em MIPS Assembly

A função "bSearch"

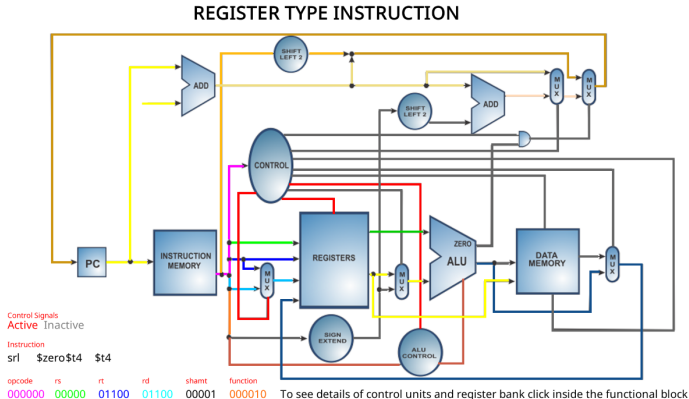
Já para a instrução **lw**, que carrega uma palavra da memória, o diagrama gerado é:



Implementação em MIPS Assembly

A função "bSearch"

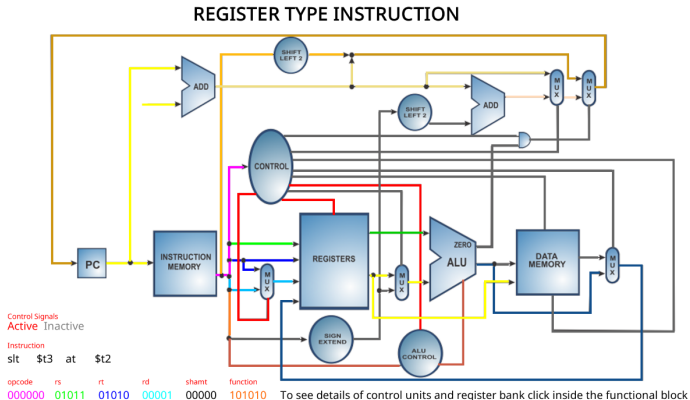
Para as instruções de *shift left* e *shift right*, que deslocam os bits de uma palavra para a esquerda (para multiplicar) e para a direita (para dividir), o mapa gerado é:



Implementação em MIPS Assembly

A função "bSearch"

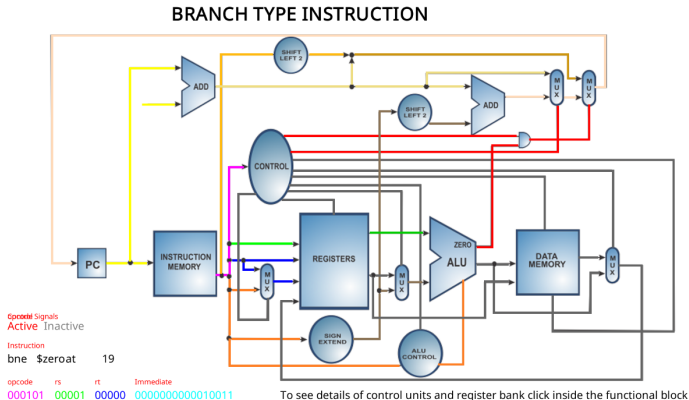
Para as instruções de desvio condicional, como **bgt** e **blt**, é usada uma combinação de **slt** e **bne**. O diagrama para **slt** é :



Implementação em MIPS Assembly

A função "bSearch"

Já para o **bne**, e similarmente, para o **beq**, que compara dois registradores ou um registrador e um valor imediato, o diagrama é:



Implementação em MIPS Assembly

Funções auxiliares

As funções auxiliares são utilizadas para definir as novas condições do passo recursivo, ou para finalizar a recursão:

- **pivoMenorQueValor**: modifica o delimitador inicial para que seja a posição do *pivo* + 1, e faz uma nova chamada de **bSearch**;
- **pivoMaiorQueValor**: modifica o delimitador final para que seja a posição do *pivo* - 1, e faz uma nova chamada de **bSearch**;
- **encontrado**: salva o índice contido no *pivo* como o resultado da busca e inicia o retorno das chamadas;
- **finalizar**: carrega o endereço de retorno armazenado na *stack* em **\$ra** e retorna o fluxo da chamada anterior.

Implementação em MIPS Assembly

Funções auxiliares

```
1  pivoMenorQueValor:
2      addi $t2, $t4, 1  #define pivo+1 como delimitador inicial
3      jal bSearch      #faz mais um passo recursivo em bSearch
4      j finalizar      #retorno recursivo de bSearch
5  pivoMaiorQueValor:
6      subi $t3, $t4, 1  #define pivo-1 como delimitador final
7      jal bSearch      #faz mais um passo recursivo em bSearch
8      j finalizar      #retorno recursivo de bSearch
9
10 encontrado:
11     move $s3, $t4      #define o valor final como o valor do pivo
12     j finalizar        #finaliza a busca com o valor encontrado
13 finalizar:
14     lw $ra, 0($sp)     #carrega o endereço de retorno da stack no $ra
15     addi $sp, $sp, 4   #"desaloca" espaço na stack
16     jr $ra             #retorna para o valor em "$ra"
```

Implementação em MIPS Assembly

Funções auxiliares

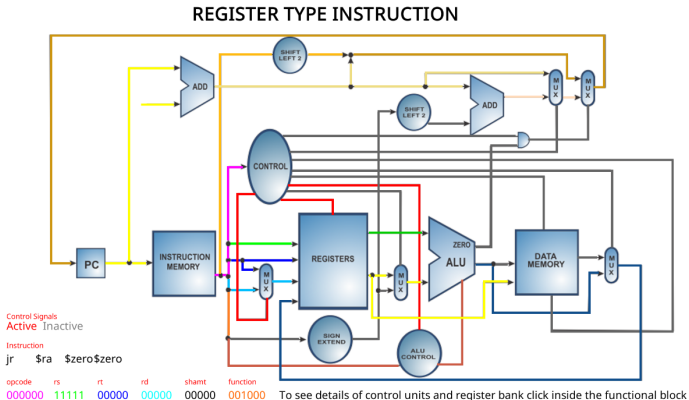
A última função auxiliar serve para imprimir o texto de "não encontrado", caso, na **main**, seja identificado que o valor armazenado no registrador de "valor final" seja **-1**.

```
17  #se o valor desejado não estiver no array, imprime o "-1", bem como a  
    ↪ mensagem de "não encontrado"  
18  naoEncontradoTexto:  
19      la $a0, fraseNaoEncontrado  
20      li $v0, 4  
21      syscall  
22  
23      li $v0, 10 #encerra o programa  
24      syscall
```

Implementação em MIPS Assembly

Funções auxiliares

O diagrama produzido pelo Xray para a instrução **jr**, que retorna para o endereço armazenado no **\$ra** é:



Conclusões

Após a implementação das versões iterativa e recursiva da busca binária, foi possível compreender muito bem a programação em baixo nível, com o gerenciamento de registradores, da stack e as otimizações necessárias para o funcionamento do código.

A produção do código em Assembly se baseou em reproduzir quase que fielmente a versão em C, a fim de se obter parâmetros de comparação para o funcionamento do programa.

Junto a isso, o código em C também foi compilado para assembly para motivos de comparação, e é evidente a quantidade de otimizações na produção de um código em assembly do zero se comparado com a conversão baixo nível → alto nível.

Conclusões

Ainda é possível otimizar mais o código em assembly, com pontos como:

- Junção dos valores de "lineFeed" e "Frase final" em uma só variável, reduzindo o número de instruções e chamadas de sistema;
- Eliminação de `blt $t5, $s1, pivoMenorQueValor` de `bSearch`, afinal a próxima função executada já é `pivoMenorQueValor`;
- Transformar a função de recursiva para iterativa também garante melhor tempo de execução, pois não é necessário salvar endereços na pilha.

Referências

- [1] B. O. P. Prado, “Estuturas de dados - busca binária e interpolada,” 2025.
- [2] A. Levitin, *Introduction to the design & analysis of algorithms*. Addison-Wesley, Pearson, 3rd ed., 2011.