Escola de Artes, Ciências e Humanidades
da Universidade de São Paulo

EDGAR HENRIQUE DE OLIVEIRA LIRA (12717266)
VICTOR SACCHI (12542776)

EXERCÍCIO PROGRAMA MIPS

SÃO PAULO
2021

EDGAR HENRIQUE DE OLIVEIRA LIRA (12717266)
VICTOR SACCHI (12542776)

EXERCÍCIO PROGRAMA MIPS

Trabalho apresentado à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo como requisito da disciplina de Organização e Arquitetura de Computadores I.

Prof. Dr.ª Gisele da Silva Craveiro.

SÃO PAULO
2021

SUMÁRIO

1 ORGANIZAÇÃO E ARQUITETURA MIPS

1.1 Linguagem de montagem e o MIPS

A linguagem de montagem é uma linguagem muito próxima da linguagem de máquina, geralmente as instruções são traduzidas para uma linguagem de máquina de uma maneira muito simples, já que as referências aos operandos estão na própria linha de comando de uma maneira mais explícita que em uma linguagem de alto nível, em contrapartida a programação em linguagem de montagem tem um processo muito mais lento e complexo, suas desvantagens que mais se destacam são no tempo de desenvolvimento, a facilidade de cometer erros, a difícil depuração do código, além da manutenção desses códigos não serem tão simples. Entretanto, podemos ressaltar algumas vantagens, as linguagens de montagem tem uma imensa fatia de mercado para produtos embarcados que devem ter um uso muito baixo de memória, além de que são nessas linguagens que são feitas as bibliotecas compatíveis com vários sistemas operacionais e que possuem funções que são utilizadas por muitos programadores; além disso, para que seja possível fazer bons compiladores é preciso entender como funciona uma linguagem de montagem e suas funções para enfim ocorrer a otimização do código como um todo.

O MIPS surge neste contexto das linguagens de montagens, com o objetivo de suprir tendências do desenvolvimento de sistemas embarcados e passou a ter uma fatia de mercado importante, principalmente para roteadores, dispositivos com sistemas embarcados, roteadores e alguns vídeo games.

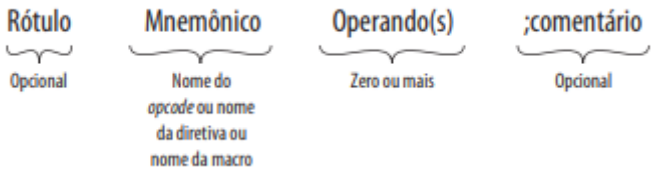
1.2 Elementos de uma linguagem de montagem

Uma linguagem de montagem tem quatro elementos básicos, os rótulos, mnemônicos, operandos e comentários.

Os rótulos definem o endereço para a instrução gerada, dessa maneira o programador não precisa calcular endereços relativos e facilita a montagem do programa. Os mnemônicos são os nomes da operações, e quando um deles é utilizado ele pode vir acompanhado ou não de operandos, que são endereços para um dado na memória, além disso são os mnemônicos que definem o opcode de uma operação e que por conseguinte definirá a instrução que deve ser executada. Os Operandos podem identificar um valor imediato, um registrador ou um espaço na memória para buscar o dado. Por fim os comentários podem ser

adicionados ao final da linha ou na linha inteira, eles não realizam nenhuma instrução, mas servem muitas vezes para explicar o que está sendo feito naquela linha ou função.(Stallings, 2009)

Estrutura da sentença de instrução:



1.3 Tipos de sentença MIPS

Existem quatro tipos de sentença no MIPS, instrução, diretiva, macro e comentário.

Instruções, as instruções em MIPS são representações simbólicas de instruções de linguagem de máquina, portanto o montador resolve quaisquer tipos de instruções e o transforma em cadeias binárias

Diretivas, as diretivas, também chamadas de pseudoinstruções, são sentenças que não são traduzidas para linguagem de máquina, mas servem para definir constantes, designar áreas da memória para guardar dados, inicializar dados da memória, permitir referências para outros programas e colocar tabelas ou outros dados fixos na memória.

No MIPS temos algumas diretivas como o “.asciiz” para definir dados de uma maneira mais natural, “.byte” definindo os lugares de memória que serão guardados os dados, “.data” que diz ao montador para guardar a string no segmento de memória, por fim temos o “.text” que diz ao montador para guardar as instruções em seu segmento de texto.

Macro, é uma seção do código que o programador escreve uma vez e pode ser chamada várias vezes, evitando processos repetitivos, elas são identificadas em tempo de montagem e após a montagem elas são indistinguíveis de um programa que não utilizou macros. No MIPS utilizamos o .macro e eles utilizam o registrador $arg como parâmetro para um função que é chamada diversas vezes.(Patterson, Hennessy, 2020)

1.4 Montador MIPS

Um montador é um software que recebe como entrada um programa em linguagem de máquina e o transforma em um arquivo binário. O montador em MIPS é dividido em duas etapas, a primeira é encontrar todos os locais de memória dos rótulos, e ler linha a linha do código para identificar o espaço de memória para cada instrução, função, além de se preparar para cada tipo de sentença identificado, já no segundo passo todas as instruções são traduzidas para binário, obedecendo todas as sentenças utilizadas e como saída ele entrega um arquivo objeto que contém as instruções de máquina, dados e informações. (Stallings, 2009)

Arquivo Objeto:

Cabeçalho do arquivo objeto	Segmento de texto	Segmento de dados	Informações de relocação	Tabela de símbolos	Informações de depuração

1.5 Link-Editor

Muitos programas são feitos em compilação separada, ou seja, divisão de um programa em arquivos diferentes e para “juntá-los” novamente é usado uma ferramenta chamada link-editor, ele se utiliza dos arquivos objetos, das bibliotecas padrões e busca resolver todas referências entre os arquivos e por fim gera um arquivo executável para o sistema operacional.(Patterson, Hennessy, 2020)

1.6 Princípios de projetos MIPS

Antes de tratarmos os princípios de projetos do MIPS é importante ressaltar que durante o início do projeto foi definido que eles que iriam seguir um projeto baseado em Risc (reduced instruction set computer), ou seja, haveriam poucos conjuntos de instruções, todas as instruções com o mesmo tamanho, poucos modos de endereçamento, um controle por hardware simples, e também a ideia de buscar sempre 1 instrução por ciclo de clock, dado todos esses fatores os projetistas decidiram os princípios de projeto, que são:

- Princípio 1: Simplicidade favorece a regularidade
    - A regularidade deixa a implementação mais simples e também permite maior desempenho por menor custo. Temos por exemplo, Instruções de tamanho fixo, Poucos formatos de instruções e o opcode sempre utilizando os primeiros 6 bits
- Princípio 2: Menor é mais rápido
    - Repertório de instruções limitadas, quantidade de registradores limitados, tendo 32 registradores de 32 bits cada um, além de números reduzidos de modos de endereçamento.
- Princípio 3: faço o caso comum rápido
    - Utilizar de constantes em instruções e operandos imediatos que evitam instrução de load, por exemplo o uso de instruções imediatas como ‘addi’ para evitar-se o uso instruções de load, ou seja, nem toda instrução precisa ser executada somente com os registradores.
- Princípio 4: Um bom projeto exige bons compromissos.
    - Quando o MIPS começou a ser desenvolvido os projetistas optaram por manter todos opcodes e instruções com o mesmo tamanho (6 e 32 bits respectivamente), portanto eles mantiveram estes compromissos para padronizar todo o projeto.
    - O compromisso escolhido pelos projetistas do MIPS é manter todas as instruções com o mesmo tamanho, exigindo, assim, diferentes tipos de formatos para diferentes tipos de instruções.(Patterson, Hennessy, 2020)

Principais registradores do MIPS e seus usos convencionais:

Registradores	**Uso**
$a0… $a3	Usados em argumentos
$t0… $t7	Registradores temporários, valores não precisam ser salvos entre chamadas
$zero	Constante 0, usado em diferentes operações que usa-se 0
$s0… $s7	Registradores com dados que precisam ser salvos entre as chamadas

Registradores	Uso
$sp	Stack Pointer, aponta para o último local na pilha
$ra	Usado como endereço de retorno de uma chamada
$v0 e $v1	Avaliação de expressão e resultado de uma função

1.7 Formatos de instruções

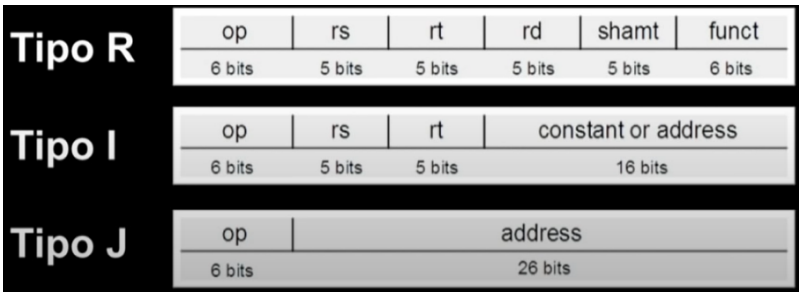
Como anteposto o MIPS possui diferentes formatos de instrução, sendo eles o tipo R, I e J, cada um sendo usado por um diferente tipo de função.

O formato de instrução R possui um campo para o Opcode, um para o operando de destino, dois para os operandos de origem, um para estender a função do opcode e outro para deslocamento de bits, sendo que esse tipo de formato é usado para instruções aritméticas como a soma e subtração.

O formato de instrução I é usado para guardar endereços e carregar endereços com determinados dados, ou seja, dar load, store e também realizar operações do tipo imediato, tendo um campo para o Opcode, um operando de destino e um de origem, e também um campo para guardar endereços e constantes.

Por fim temos o tipo J, formato esse usado para jumps, contendo apenas o opcode e o endereço do campo.(Patterson, Hennessy, 2020)

Formatos de instrução:



2 EXPLICAÇÃO DO PROBLEMA E CÓDIGO EM ALTO NÍVEL

2.1 Problema: Números perfeitos, abundantes ou deficientes.

Números abundantes são números em que a soma de seus divisores próprios é maior que esse número. Números perfeitos são números em que a soma de seus divisores próprios é igual a esse número. Por fim, números deficientes são números em que a soma de seus divisores próprios é menor que este número, por exemplo:

Número	Divisores Próprios	Soma	= / > / <	Tipo
12	1, 2, 3 e 6	16	16 > 12	Abundante
6	1, 2 e 3	6	6 = 6	Perfeito
3	1	1	1 < 3	Deficiente

2.2 Código em alto nível - Linguagem C

```
#include <stdio.h>
int main(void) {
	int x = 0;
	soma = 0;
	scanf("%d", &x);

	for (int i = 1; i < x; i++){
		if (x % i == 0)
			soma = soma + i;
	}
	if(soma > x)
		printf("Número Abundante\n");
	if(soma == x)
		printf("Número Perfeito\n");
	if(soma < x)
		printf("Número Deficiente\n");

	return 0;
}
```

3 CÓDIGO EM ASSEMBLY DESENVOLVIDO

```
.data
# Variáveis que guardam uma String
digite: .asciiz "Digite um número:"
perfeito: .asciiz "Número Perfeito"
deficiente: .asciiz "Número Deficiente"
abundante: .asciiz "Número Abundante"

.text
.globl main
main:
        li $a1,0            # inicia o local do número que será digitado
        li $a2,0            # inicia o contador
        li $a3,0            # inicia o lugar onde ficará a soma dos inteiros

        jal scanInt         # vai para função de escanear um inteiro
        jal divisores       # chama divisores
        jal resultado       # Chama o resultado

        li $v0,10           # termina
        syscall             # chama a função de v0

#Recebe um inteiro digitado pelo usuário
scanInt:
        la $a0, digite      # Da um load no endereço de digite no registrador $a0
        li $v0, 4           # Carrega a instrução 4 para o registrador $v0
        syscall             # Executa a instrução que está no registrador $v0
        li $v0, 5           # Carrega a instrução 5 para o registrador $v0
        syscall             # Executa a instrução que está no registrador $v0
        move $a1, $v0       # Move os dados de $v0 para $a1
        jr $ra              # Retorna para última posição salva
```

```
divisores:
        add $a2,$a2,1       # soma 1 do contador
        blt $a2,$a1,cond    # verifica se chegou no final
        j divisores_fim     # vai para o final

cond:
        div $a1, $a2                #Divide os valores contido em $a1 e $a2
        mfhi $s4            #Move o resto da soma do registrador hi para $s4
        beq $s4, 0, salva_e_soma  #Se $s4 e 0 são iguais ele pula para função
        j divisores                 #jump simples para divisores

salva_e_soma:

      subu $sp,$sp,8        # Abre um espaço na pilha
      sw $a2,4($sp)         # Salva o $a2
      sw $ra,0($sp)         # Salva o endereço de retorno
      jal divisores         # chama o fatorial
      lw $ra,0($sp)         # Recupera o valor original do $a2 em $t0
      lw $t0,4($sp)         # Recupera o valor original do $a2 em $t0
      addu $sp,$sp,8        # Retira o espaço ocupado na pilha
      add $a3,$a3,$t0       # soma $a3 = $a3 + $t0

divisores_fim:
      jr $ra                # retorna para o endereço contido em $ra

# Função que compara a soma dos divisores próprios do número com o número
resultado:

        beq $a3, $a1, nPerfeito     # Vai para a função nPerfeito
        blt $a3, $a1, nDeficiente   # Vai para a função nDeficiente
        bgt $a3, $a1, nAbundante    # Vai para a função nAbundante
```

nPerfeito: # Printa ‘Número perfeito’ na tela

```
li $v0, 4              # Carrega a instrução 4 para o registrador $v0
la $a0, perfeito       # Carrega a string de perfeito para o registrador $a0
syscall                # Mostra na tela o conteúdo do registrador $a0
jr $ra                 # Retorna para o endereço salvo no $ra
```

nDeficiente: # Printa ‘Número deficiente’ na tela

```
li $v0, 4              #Carrega a instrução 4 para o registrador $v0
la $a0, deficiente     #Carrega a string de deficiente para o registrador $a0
syscall                # Mostra na tela o conteúdo do registrador $a0
jr $ra                 # Retorna para o endereço salvo no $ra
```

nAbundante: # Printa ‘Número abundante’ na tela

```
li $v0, 4              # Carrega a instrução 4 para o registrador $v0
la $a0, abundante      # Carrega a string de abundante para o registrador $a0
syscall                # Mostra na tela o conteúdo do registrador $a0
jr $ra                 # Retorna para o endereço salvo no $ra
```

4 EXPLICAÇÃO DETALHADA DAS INSTRUÇÕES NO CÓDIGO

Tabela de instruções usadas

Instrução	Explicação	Exemplo	Ciclo de Execução
li - load immediate	Carrega o valor passado como parâmetro para o registrador destino	li, $s0, 5	$s0 <- 5
la - load address	Carrega o endereço passado como parâmetro para o registrador destino	la, $s0, $s1	$s0 <- $s1
sw - store word	Instrução de movimentação de dados do registrador para a memória	sw $s1,4($s2)	Memór[$s2+4] <- $s1
lw- load word	Instrução de movimentação de dados da memória para registrador	lw, $s0, 0($s2)	$s0 <- Memór[$s2+0]
move	Move o conteúdo de um registrador para outro	move, $s0, $s1	$s0 <- ($s1)
j - jump	Pula para o endereço da função passada como parâmetro	j label	pc <- pc + 1
jal - jump and link	Pula para o endereço da função e salva o momento em que estava	jal label	$ra <- (pc) pc <- pc + 1
jr - jump register	Pula incondicionalmente para instrução cujo endereço está no	jr $ra	pc <- ($ra)

Instrução	Explicação	Exemplo	Ciclo de Execução
	registrador passado como parâmetro		
blt - branch less than	Compara o conteúdo de dois registradores e se r1 < r2 vai para o endereço da função que foi passada como parâmetro	blt, $s0, $s1, lab	if $s0 < $s1 then: pc <- pc + 4 + lab
bgt - branch gran than	Compara o conteúdo de dois registradores e r1 > r2 vai para o endereço da função que foi passada como parâmetro	bgt, $s0, $s1, lab	if $s0 > $s1 then: pc <- pc + 4 + lab
beq - branch equals	Compara o conteúdo de dois registradores e se são iguais vai para o endereço da função que foi passada como parâmetro	beq, $s0, $s1, lab	if $s0 == $s1 then: pc <- pc + 4 + lab
div - division	Recebe 2 parâmetros e divide, guardando a parte inteira no registrador lo e a parte decimal no registrador hi	div, $s0, $s1	[hi, lo] <- $s0 / $s1
add	Coloca a soma dos operando no registrador destino	add, $s0, $s1, $s2	$s0 <- ($s1+$s2)

Instrução	Explicação	Exemplo	Ciclo de Execução
addu - add unsigned	Coloca a soma dos operando no registrador destino	addu, $s0, $s1, $s2	$s0 <- ($s1+$s2)
sub - subtraction	Coloca a subtração dos operandos no registrador destino	sub, $s0, $s1, $s2	$s0 <- ($s1-$s2)
subu - subtraction unsigned	Coloca a subtração dos operandos no registrador destino	subu, $s0, $s1, $s2	$s0 <- ($s1-$s2)
mfhi - move from hi	move o conteúdo do registrador hi para o registrador destino	mfhi, $s0	$s0 <- (hi)

5 REFERÊNCIAS

Apêndice A - “Organizações e Projeto de Computadores - A interface Hardware/Software”, David A. Patterson & John L Hennessy, Campus, 7° edição, 2020

Seção 2 - “Organizações e Projeto de Computadores - A interface Hardware/Software”, David A. Patterson & John L Hennessy, Campus, 7° edição, 2020

Apêndice B - “Arquitetura e Organização de Computadores”, W. Stallings 8° edição, 2009