Laboratório 1 - Desenvolvimento de Aplicações em Assembly MIPS

Marcos Vinícius Marques - 14/0071989
Afonso Dias de Oliveira Conceição Silva - 14/0055771
Guilherme Andreúce Sobreira Monteiro - 14/0141961
Universidade de Brasília - UnB

5 de Maio de 2019

Organização e Arquitetura de Computadores - turma B

marcos10@outlook.com, afonso199626@gmail.com, guilhermeandreuce@gmail.com

Abstract

Neste laboratório realizamos a partir de uma entrada um arquivo texto ASCII com o código-fonte elaborado por instruções assembly MIPS (arquivos com a extensão ".asm"), em que este seja capaz de gerar um código objeto montado em Hexadecimal em arquivo de texto ASCII, no formato MIF (Memory Inicialization File) de uma listagem de instruções pré-definidas, e contidas especificamente nas áreas .text e .data do arquivo de entrada (.ams) fornecido pelo usuário da aplicação. Geramos na saída um arquivo, também em codificação ASCII, com o mesmo nome do arquivo de entrada, com a extensão ".mif" (um arquivo para a área .data e outro para a área .text). A aplicação contém como argumento de entrada, além de todo o leque de registradores inteiros da CPU MIPS, incluindo as máscaras atribuídas aos registradores, e permite a entrada no campo imediato de números inteiro e/ou decimais, ambos inteiros e sinalizados.

1. Objetivos

O projeto teve como objetivo geral a familiarização com a linguagem Assembly MIPS e a aprendizagem de metodologias de aplicações eficientes e otimizadas.
Também teve como intuito formar espírito crítico de avaliação a respeito do desempenho real provido pelo sistema computacional, propiciando assim melhorias na compreensão do funcionamento destes tipos de sistemas.
Além disso, visou complementar e reforçar o conteúdo programático da disciplina Organização e Arquitetura de Computadores e desenvolver a capacidade de observação, análise e compreensão das metodologias de organização e arquitetura de computadores.

2. Introdução

Assembly ou linguagem de montagem é uma notação legível por humanos para o código de máquina que uma arquitetura de computador específica usa, utilizada para programar códigos entendidos por dispositivos computacionais, como microprocessadores e microcontroladores. O código de máquina torna-se legível pela substituição dos valores em bruto por símbolos chamados mnemónicos.

Por exemplo, enquanto um computador sabe o que a instrução-máquina IA-32 (B0 61) faz, para os programadores é mais fácil recordar a representação equivalente em instruções mnemónicas MOV AL, 61h. Tal instrução ordena que o valor hexadecimal 61 (97, em decimal) seja movido para o registrador 'AL'. Embora muitas pessoas pensem no código de máquina como valores em binário, ele é normalmente representado por valores em hexadecimal.

A tradução do código Assembly para o código de máquina é feita pelo montador ou assembler. Ele converte os mnemónicos em seus respectivos opcodes, calcula os endereços de referências de memória e faz algumas outras operações para gerar o código de máquina que será executado pelo computador. Para carregar nossos problemas de montagem MIPS manuscritos (ou gerados por compilador) em nossa ROM de instruções, precisamos de uma maneira de montá-los em linguagem de máquina e depois salvar esses programas em um arquivo de texto onde as instruções binárias são representadas como uma sequência de valores hexadecimais ASCII.

Um arquivo de inicialização de memória (.mif) é um arquivo de texto ASCII (com a extensão .mif) que especifica o conteúdo inicial de um bloco de memória (CAM, RAM ou ROM), ou seja, os valores iniciais de cada endereço. É esse arquivo que estaremos formulando aqui.

3. Materiais e Métodos

Uma observação é que nossas instruções funcionam com espaço depois da vírgula. Neste laboratório foram usados o ambiente Mars v. 4.5 para programação em Assembly Mips. O MARS é um ambiente de desenvolvimento interativo leve (IDE) para programação em linguagem assembly MIPS, destinado ao uso em nível educacional. A listagem de instruções a serem compiladas e montadas pela aplicação desenvolvida são: lw $t0, OFFSET($s3)
add/sub/and/or/nor/xor $t0, $s2, $t0
sw $t0, OFFSET($s3)
j LABEL
jr $t0
jal LABEL
beq/bne $t1, $zero, 0xXXXXX
slt $t1, $t2, $t3
lui $t1, 0xXXXX
addu/subu $t1, $t2, $t3
sll/srl $t2, $t3, 10
addi/andi/ori/xori $t2, $t3, -10
mult $t1, $t2
div $t1, $t2
li $t1, XX (incluindo na forma de pseudoinstrução)
mfhi/mflo $t1
bgez $t1, LABEL
clo $t1, $t2
srav $t1, $t2, $t3

Inicialmente é lido um arquivo de entrada com codigo fonte em assembly, cada linha é processada por um parser que identifica, com um comportamento de maquina de estados, a instrução ou label, ou alocação de memória. Esse parser é feito usando uma grande estrutura de IF's (Testes condicionais) que manipulam o código a fim de gerar a conversão em hexadecimal da instrução. Por exemplo, se o primeiro caracter de uma linha for "a", então o parser trabalha com a possibilidade de ser uma instrução add ou and. No final dessa arvore de decisão existem nós folha para cada possibilidade de instrução, que então aloca alguns registradores $s e $t para a construção em binario que representa as instruções, conforme a tabela da figura 1, no futuro esse codigo será convertido em hexadecimal. As instruções em Mips possuem formatos parecidos, variando em apenas algumas colunas.

Essa execução pode ser vista mais detalhadamente tomando como exemplo a instrução "or"na figura 2.

No começo o procedimento lida com a leitura da linha, caractere à caractere, em seguida os campos opcode, shamt, func são separados em registradores, para finalmente serem concatenadas. A função pegaregistrador coordena a seleção dos diferentes tipos de registradores coordenando e chamando seus reespectivos procedimentos. Cada registrador

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Figura 1. Codificação de instruções MIPS. Na tabela acima, "reg" significa um número de registro entre 0 e 31, "endereço" significa um endereço de 16 bits e "n.a" (não aplicável) significa que este campo não aparece neste formato. Observe que as instruções add e sub têm o mesmo valor no campo op; o hardware usa o campo funct para decidir a variante da operação: add (32) ou subtrair (34).

```
503     i_or:
504       move $t0, $zero #contador de registrador (0 registrador rd)
505       jal readchar  #le caracter
506       bne $v0, 114, undefined  #se o proximo caracter não for 'r', instrução não definida.
507       jal readchar  #le caracter
508       #beq $v0, 105, i_ori  #se o proximo caracter for 'i', funçao de escrita da addi
509       bne $v0, 32, undefined #se o proximo caracter não for um 'espaço', instrução não definida.
510       jal readchar  #le caracter
511       bne $v0, 36, undefined  #se o proximo caracter não for um '$', instrução não definida
512       jal readchar #le caracter
513       la $s0, s_opcode_add_sub_and_or_nor_xor_jr_slt_addu_subu_sll_srl_mult_div_mfhi_mflo_srav  #coloca opcode add em s0
514       la $s4, s_shamtlipor #coloca shamt em tipos r em s4
515       la $t9, s_function_or #coloca o function do add em t9
516       jal pegaregistrador #função que pega registrador
517       jal readchar
518       addi $t0, $t0, 1  #incrementa contador
519       bne $v0, 44, undefined #se o proximo caracter não for um ',', instrução não definida.
520       jal readchar  #le caracter
521       bne $v0, 32, undefined #se o proximo caracter não for um 'espaço', instrução não definida.
522       jal readchar  #le caracter
523       bne $v0, 36, undefined  #se o proximo caracter não for um '$', instrução não definida
524       jal readchar #le caracter
525       jal pegaregistrador #função que pega registrador
526       jal readchar
527       addi $t0, $t0, 1  #incrementa contador
528       bne $v0, 44, undefined #se o proximo caracter não for um ',', instrução não definida.
529       jal readchar  #le caracter
530       bne $v0, 32, undefined #se o proximo caracter não for um 'espaço', instrução não definida.
531       jal readchar  #le caracter
532       bne $v0, 36, undefined  #se o proximo caracter não for um '$', instrução não definida
533       jal readchar #le caracter
534       jal pegaregistrador #função que pega registrador
535       j concatenate
```

Figura 2. Parser da instrução OR

possui sua propria configuração (Figura 3).

```
1120      i_tnumero9:
1121
1122        beq $t0, 0, i_tnumero9rd
1123        beq $t0, 1, i_tnumero9rs
1124        beq $t0, 2, i_tnumero9rt
1125        j undefined
1126        i_tnumero9rd:
1127        la $s3, s_t9_25_em_bin  #coloca string t9 em s3 (rd)
1128        j i_tnumero9continue
1129        i_tnumero9rs:
1130        la $s1, s_t9_25_em_bin  #coloca string t9 em s1 (rs)
1131        j i_tnumero9continue
1132        i_tnumero9rt:
1133        la $s2, s_t9_25_em_bin  #coloca string t9 em s2 (rt)
1134        j i_tnumero9continue
1135      i_tnumero9continue:
1136        lw $ra, 0($sp)  #lê valor de ra que estava na pilha
1137        addi $sp, $sp, 4 #zera a pilha
1138        jr $ra
```

Figura 3. Modelo de configuração de um registrador $t.

Depois disso temos que concatenar essas strings e convertendo-as para hexadecimal.

Com a instrução convertida para hexadecimal podemos escreve-la no arquivo de saída. Na Figura 5 podemos ver a seleção do caractere em hexadecimal que será escrito em arquivo.

Todo o processo é repetido em loop, inclusive para .word

```
1604   i_vetordecaracteresparadecimal: #monta vetor de caracteres em decimal em buffer_caracter_decimal
1605       addi $sp, $sp, -4  #prepara pilha pra receber 1 item
1606       sw $ra, 0($sp)     #salva o endereço de $ra em sp
1607       move $t0, $zero  #anda de 4 em 4
1608       move $t2, $zero  #salva quantos numeros foram lidos
1609       move $t1, $zero
1610     vetordecaracteresparadecimalstart:
1611     jal readchar
1612     beq $v0, 48, digito_0 #se digito 0
1613     beq $v0, 49, digito_1 #se digito 1
1614     beq $v0, 50, digito_2 #se digito 2
1615     beq $v0, 51, digito_3 #se digito 3
1616     beq $v0, 52, digito_4 #se digito 4
1617     beq $v0, 53, digito_5 #se digito 5
1618     beq $v0, 54, digito_6 #se digito 6
1619     beq $v0, 55, digito_7 #se digito 7
1620     beq $v0, 56, digito_8 #se digito 8
1621     beq $v0, 57, digito_9 #se digito 9
1622     beq $v0, 10, gobackcaller #func que volta pra caler
1623     bge $t1, 64, undefined
1624     j undefined
1625   gobackcaller:
1626           lw $ra, 0($sp)     #salva o endereço de $ra em sp
1627           addi $sp, $sp, 4  #prepara pilha pra receber 1 item
1628           jr $ra
1629
1630    digito_0:
1631     addi $t1, $v0, -48
1632     sw $t1, buffer_caracter_decimal($t0)
1633     addi $t0, $t0, 4
1634     addi $t2, $t2, 1
1635     j vetordecaracteresparadecimalstart
```

Figura 4. Conversão de char para decimal

```
2455  verificawdigito:
2456      beq $t0, 48, wdigito_0 #se digito 0
2457      beq $t0, 49, wdigito_1 #se digito 1
2458      beq $t0, 50, wdigito_2 #se digito 2
2459      beq $t0, 51, wdigito_3 #se digito 3
2460      beq $t0, 52, wdigito_4 #se digito 4
2461      beq $t0, 53, wdigito_5 #se digito 5
2462      beq $t0, 54, wdigito_6 #se digito 6
2463      beq $t0, 55, wdigito_7 #se digito 7
2464      beq $t0, 56, wdigito_8 #se digito 8
2465      beq $t0, 97, wdigito_a #se digito 9
2466      beq $t0, 98, wdigito_b #se digito a
2467      beq $t0, 99, wdigito_c #se digito b
2468      beq $t0, 100, wdigito_d #se digito c
2469      beq $t0, 101, wdigito_e #se digito d
2470      beq $t0, 102, wdigito_f #se digito e
2471      j undefined
2472
2473  wdigito_0:
2474
2475  li   $v0, 15       # system call for write to file
2476  move $a0, $s7      # file descriptor for text stored in s7
2477  la   $a1,  s_tohex0# address of buffer from which to write
2478  li   $a2, 1       # hardcoded buffer length (size of buffer_data_init in decimal)
2479  syscall            # write to file
2480  jr $ra
2481  wdigito_1:
```

Figura 5. Escrevendo hexadecimal em arquivo.

e .text, até o final da leitura do arquivo de entrada.

É importante ressaltar que dependendo da arquitetura do computador, onde o simulador Mars esta executando o programa, é preciso ajustar o alinhamento de words. Nos testes realizados algumas words ficaram desalinhadas na memória de emulação do PC no indice 3.

4. Resultados

Usando um arquivo de entrada apenas com instruções tipo R, pois não terminamos de implementar as outras instruções, obtemos os seguintes resultados: (Figura 7)

A partir da ferramenta Intruction Counter do Mars, obteve-se uma porcentagem de tipos de instruções utilizadas. Em seguida podemos ver a contagem de instruções. (Figura 8).

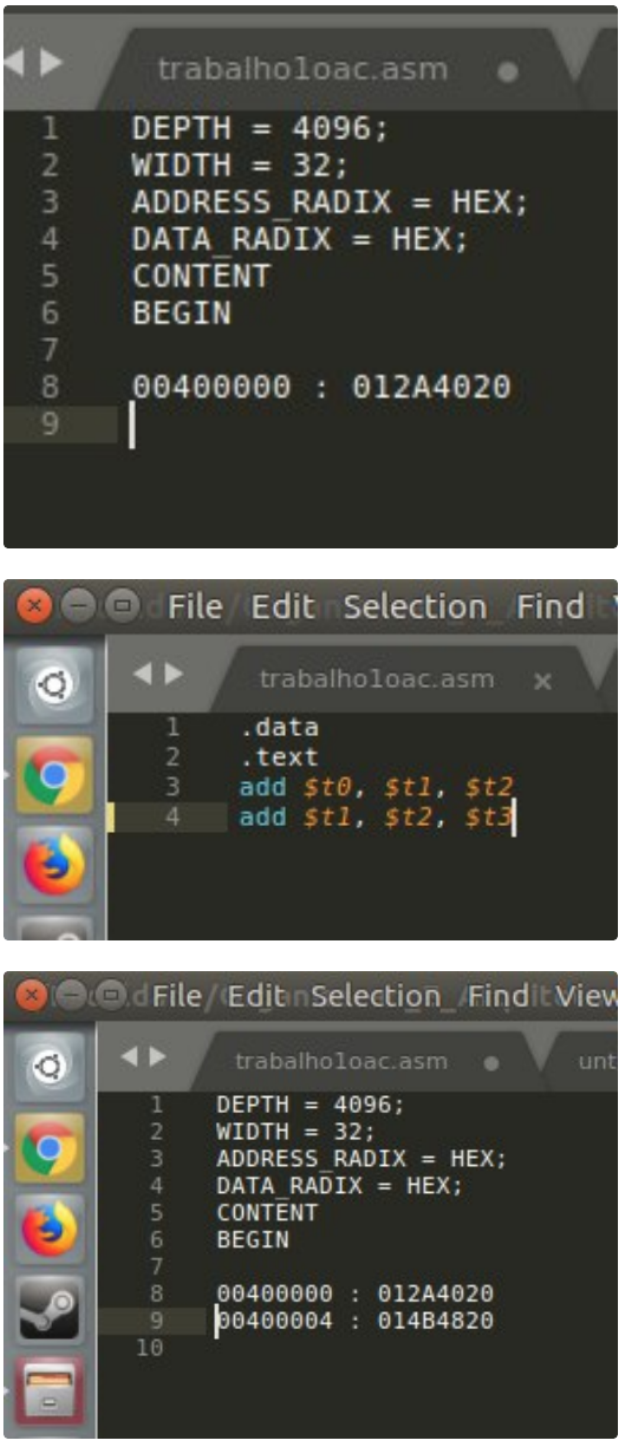

Figura 6. Desalinhamento de words

5. Discussão e Conclusões

A proposta de criar um programa em Assembly Mips exercita diversos pontos recém aprendidos do funciona-

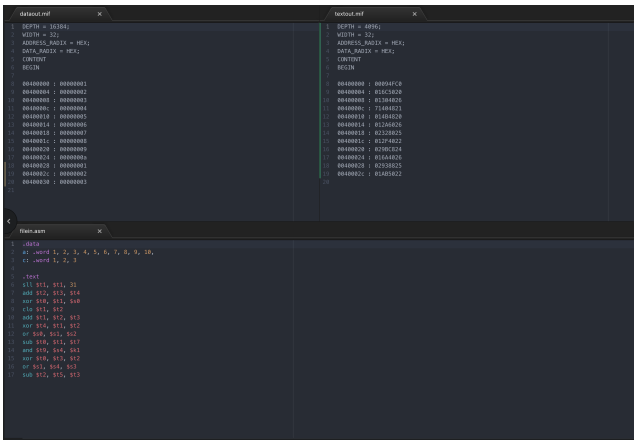

Figura 7. Entrada e saidas da execução do programa.

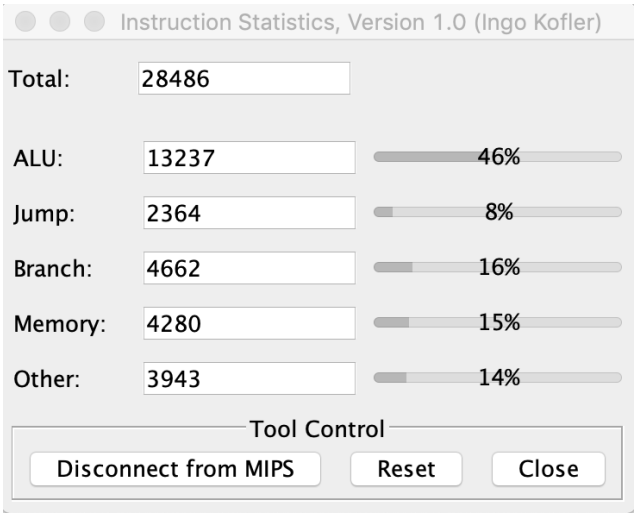

Figura 8. Estatisticas de instruções.

mento da arquitetura dos computadores modernos. O desafio foi lidar com uma estrutura de decisão tão grande, acrescido de entender o arquivo de inicialização do Mips e gerar corretamente a conversão em hexadecimal das instruções propostas. Inicialmente o programa recebe um arquivo com codigo fonte Assembly Mips (.asm) com um leque de instruções, uma área .text e uma área .word. Nosso trabalho então foi interpretar cada linha e gerar um código equivalente em hexadecimal, que no futuro será utilizado como arquivo de entrada de um programa em um processador Mips projetado no Altera Quartus II. Nosso fluxo de execução começa no parser, na identificação da instrução representada na linha. Em separar a lógica para cada caso especifico, e em criar procedimentos pontuais para tratar cada caso. Como armazenamento em pilha, alocação de registradores, conversões de base e outros. Finalmente convertemos os valores codificados que representam cada instrução em hexadecimal e escrevemos em ordem no arquivo. Durante o desenvolvimento do projeto pode-se perceber a dificuldade em se trabalhar com códigos não modularizados e também a incrível quantidade de instruções necessárias para se realizar pequenas tarefas, que em linguagem de alto nivel são relativamente simples. Isso impactou no nosso entendimento sobre desempenho de código e como isso é uma área de foco na computação. Dessa forma aprendemos muito sobre arquitetura do processador Mips, sobre assembly e, principalmente, sobre organização e arquitetura de computadores modernos e o que a compilação e a manipulação de código representa para a computação em significância para o desempenho dos computadores.

6. Bibliografia

1. Assembly
   https://pt.wikipedia.org/wiki/Assembly
2. MARS (MIPS Assembler and Runtime Simulator)
   https://courses.missouristate.edu/KenVollmar/MARS/
3. Arquitetura MIPS
   https://pt.wikipedia.org/wiki/Arquitetura$_MIPS$
4. Arquitetura e Organização de Computadores (4a edição) - Patterson e Hennesy