UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO" – UNESP

INSTITUTO DE GEOCIÊNCIAS E CIÊNCIAS EXATAS

Dalton Varussa de Oliveira Lima - 141152257
Lucas Augusto Pinheiro - 141150688
Stefany Lacroux - 141150955
Tadeu Martines - 141152214
Victor Henrique Ribeiro - 141153237

TRABALHO DE ARQUITETURA
PARSER MIPS ASSEMBLY LANGUAGE
2ª Parte

Rio Claro
2015

Trabalho apresentado como requisito parcial para aprovação na disciplina Arquitetura de Computadores do curso de Ciência da Computação, UNESP.

Prof. Dr. Maurício Acconcia Dias

SUMÁRIO

1 - INTRODUÇÃO

Este programa visa desenvolver os conhecimentos básicos sobre os processadores MIPS. O programa foi desenvolvido em linguagem C.
Esta segunda parte, utilizou conceitos e funções previamente desenvolvidas na primeira parte deste trabalho.

2 - DESENVOLVIMENTO

O trabalho foi desenvolvido com o auxílio de um sistema de controle de versão, o GIT. Todo o programa, e seu código fonte com todas as diferentes versões pode ser encontrado no seguinte endereço: https://github.com/daltonbr/MIPS.

Devido ao escasso tempo dos alunos ao fim deste semestre letivo, optamos por simplificar o projeto, a fim de conseguirmos um resultado expressivo. Para tanto definimos alguns pontos:

- O programa irá ler somente as funções já pré-configuradas no código (*hardcoded*).

- A arquitetura é monociclo. Uma instrução é executada até o fim, antes de uma nova ser carregada.
- Durante todo o processo os valores dos Registradores e Banco de Memória são atualizados, refletindo os dados que estão sendo trabalhados. (para simplificação mostramos somente os valores relevantes)

Escolhemos algumas operações que mais caracterizavam o uso das estruturas básicas do MIPS. Para tal escolhemos as funções LW (Load Word), SW (Store Word), ADD, ADDI, BEQ (Branch Equal).

Montamos o seguinte código em assembly:

```
lw $t0,4($t1)           // $t0 <= $t1+4 = [4+4] = [8] = 10 // t0<= 10
addi $t1,$zero,8        // $t1 <= 0+8 = 8
add $t2,$t0,$t1         // $t2 <= $t0 + $t1 = 10 + 8 = 18
beq $t2,$t3,loop        // compara $t2 e $t3 - Devem ser iguais 18 = 18
addi $t3,$zero,0        // logo pulamos esta linha (que "setaria" $t3 = 0)
loop:
sw $t3,4($t1)           // $t3 = 18 (na memória) em t1+4 = 8+4 = 12/4 =3
Portanto escreve o valor 18 na posição 3 da memória (4 linha)
```

O código acima foi colocado no Parser (1ª parte deste trabalho), o qual gerou o seguinte código em binário:

```
10001101000010010000000000000100          // lw $t0,4($t1)
00100001001000000000000000001000          // addi $t1,$zero,8
00000001001010100100000000100000          // add $t2,$t0,$t1
00010001010010110000000000000001          // beq $t2,$t3,loop  (offset +1)
00100001011000000000000000000000          //addi $t3,$zero,0
10101101011010010000000000000100          //loop: sw $t3,4($t1)
```

Em seguida este código foi colocado internamente no código da versão final.

Definimos também valores iniciais para os **registradores** e **banco de memória**, mostraremos aqui somente os mais relevantes, (todos os demais seguem inalterados durante a execução do programa).

- **valores INICIAIS registradores**

$t0 = 0000 0000 0000 0000 0000 0000 0000 0000 = [0]base 10

$t1 = 0000 0000 0000 0000 0000 0000 0000 0100 = [4]base 10

$t2 = 0000 0000 0000 0000 0000 0000 0000 0000 = [0]base 10

$t3 = 0000 0000 0000 0000 0000 0000 0001 0010 = [18]base 10

- **valores INICIAIS memória**

[01] 0000 0000 0000 0000 0000 0000 0000 0000 = [0]base 10

[02] 0000 0000 0000 0000 0000 0000 0000 1010 = [10]base 10

[03] 0000 0000 0000 0000 0000 0000 0000 0000 = [0]base 10

Após todas as operações executadas, verifica-se as seguintes posições finais dos **registradores** e **banco de memória:**

- **valores FINAIS registradores**

$t0 = 0000 0000 0000 0000 0000 0000 0000 1010 = [10]base 10

$t1 = 0000 0000 0000 0000 0000 0000 0000 1000 = [8]base 10

$t2 = 0000 0000 0000 0000 0000 0000 0001 0010 = [18]base 10

$t3 = 0000 0000 0000 0000 0000 0000 0001 0010 = [18]base 10

- **valores FINAIS do banco de memória**

[01] 0000 0000 0000 0000 0000 0000 0000 0000 = [0]base 10

[02] 0000 0000 0000 0000 0000 0000 0000 1010 = [10]base 10

[03] 0000 0000 0000 0000 0000 0000 0001 0010 = [18]base 10

3 - FUNCIONAMENTO DO PROGRAMA

O funcionamento do programa é bem simples. Cada instrução é carregada de uma única vez. Todos os dados relevantes àquela instrução aparecem na tela. Optamos por deixar algumas informações de DEBUG na versão final, pois elas podem ajudar na análise do desdobramento das instruções.

Ao final de cada instrução, uma mensagem no fim da tela orienta o usuário a apertar qualquer tecla para seguir para a próxima instrução.

4 - CONCLUSÃO

Este programa foi muito interessante para conhecer mais a fundo o funcionamento de um processador como o MIPS.

Gostaríamos de salientar como crítica construtiva o fato de que os requisitos da segunda parte do programa não foram divulgados com mais antecedência. Sendo assim, detalhes importantes da arquitetura e organização do programa teriam sido tomados de forma diferentes; o que facilitaria muito no desenvolvimento do programa.

5 – DETALHES TÉCNICOS

Os arquivos fonte principais são: *mips.c, mips2.h.*
Existem arquivos secundários de documentação nas demais pastas internas.
Todos estes arquivos podem ser encontrados e baixados gratuitamente em:
https://github.com/daltonbr/MIPS

Para compilar utilizamos e testamos o seguinte compilador:

- Microsoft C/C++ Optimizing Compiler Version 19.00.23026 for x86
  Encontrado junto ao Microsoft Visual Studio 2015.

A versão entregue foi a 3.0.0 de 31/08/15

6 - BIBLIOGRAFIA

Patterson, David A.; Hennessy, John L. Organização e Projeto de Computadores. 3 ed. Rio de Janeiro: ELSEVIER, 2006.