5 de Maio de 2017

Trabalho II

Eduardo Said Calil Vilaça - 13/0154253

Lukas Machado - 12/0127377

Raphael Queiroz - 13/0154989

Professor: Prof. Dr. Marcelo Grandi Mandelli
Matéria: Organização e Arquitetura de Computadores

Departamento de Ciência da Computação
Universidade de Brasília

1 Objetivo

O objetivo deste trabalho foi implementar um simulador da arquitetura MIPS para entender como funciona a organização, a decodificação e execução em linguagem de assembly e linguagem de máquina.

2 Implementação

O código foi desenvolvido em C e foi utilizado o arquivo MakeFile para compilar o projeto (instruções para compilar no README.txt). O trabalho foi dividido em quatro bibliotecas com funções relativas a cada etapa de desenvolvimento do trabalho. As divisões foram:

- Carregar dados
- Funções de acesso à Memória
- Operações Simples
- Instruções

Esse projeto permite a simulação das instruções em assembly MIPS a partir de arquivos binários gerados pelo simulador MIPS MARS. Portanto, temos como entrada arquivos organizados de acordo com a hierarquia de memória(.text e .data) que estão em linguagem de máquina(bits), com isso, devemos simular o processo de execução de cada instrução, decodificando e simulando as mesmas.

2.1 Carregar Dados

Nesta etapa, foram implementadas duas funções: LoadText e LoadData.

A função LoadText lê o arquivo binário gerado pelo simulador MARS da parte da memória .text, que representa a memória de programa, onde se encontram as intruções em linguagem de máquina. Essa função lê o arquivo com auxílio da função 'fread' na qual lê do arquivo uma word por vez e guarda na memoria a partir do endereço determinado para memória de programa(0x00000000).

A função LoadData executa basicamente as mesmas instruções da função LoadText, porém lê o arquivo que contém o conteúdo da memória de dados e os salva na parte da memória reservada para dados, que começa no endereço de memória 0x0002000.

2.2 Funções de acesso a Memória

As funções de acesso a memória são: load word, load byte, load half-word, store word, store byte, store half-word, assim como instruções para impressão da memória e dos registradores.

A função load word tem como parâmetros um endereço e um 'offset' que quando somados temos um endereço em bytes da memória, logo, ao dividirmos por 4 iremos saber qual word da memória que o usuário deseja carregar. A função store word segue a mesma lógica. A diferença é que os dados são salvos e não retornados.

A função load half-word e store half-word seguem a mesma lógica de endereçamento em bytes, porém existe um detalhe, somente múltiplos de dois são acessíveis. Logo, ao descobrir em qual word da memória está a half-word, devemos verificar o resto da divisão por quatro. Como só temos múltiplos de 2, teremos duas opções(halfwords na word) 0 e 2 que são os endereços base de qualquer half-word em uma palavra.

Assim como as funções anteriores, as funções load byte e store byte seguem o padrão do endereçamento em bytes com a diferença na análise do resto. Verificando o resto da divisão por 4 sabemos qual byte que o usuário deseja, logo podemos retorna-lo ou altera-lo.

Além das funções de acesso a mémoria temos as funções que exibem na tela o conteúdo dos registradoes, de uma região da memória ou da memória completa.

2.3 Operações Simples

Na biblioteca de operações simples foram implementadas basicamente 3 funções, com as devidas variações de execução direta e execução passo a passo, para melhor visualização do usuário final. Essas três instruções foram: fetch,decode e execute. Essas instruções simulam o processo no qual o processador MIPS realiza as instruções a partir da linguagem de máquina. Primeiramente ele pega a instrução na memória (fetch), com isso decodifica as instruções com seus campos (decode) e, por fim, executa a instrução em uma Unidade Lógica Aritmética ou outro circuito lógico.

A instrução **fetch** tem como único objetivo e funcionalidade colocar a instrução contida na memória de programa e apontada pelo registrador PC (contador de programa) no registrador de instruções RI e incrementar PC para apontar para a próxima instrução.

A instrução **decode** decodifica a instrução , ou seja, com a instrução contida em RI realiza operações lógicas e aritméticas para selecionar os campos das intruções, sendo ela do tipo I,R ou J.

A instrução **execute** é a principal função dessa biblioteca e possui todas as instruções aritméticas e lógicas do tipo R, I e J. Essa instrução seleciona qual instrução o processador deve executar, de acordo com o seu OPCODE. Se seu OPCODE é 0x00 ele é do tipo R, então vamos para outra função que seleciona as funções a partir do campo FUNCT. Ao selecionar a instrução chamamos as funções referentes ao seu código de outra biblioteca que será explicada mais a frente.

A função que agrega todos os passos para excecutar uma instrução é denominada **step** e a função que roda a função step até o término do programa apontado pela operação SYSCALL ou até o fim da memória de programa 0x0001000 é a função **run()**.

2.4 Instruções

Cada instrução foi implementada com o auxílio das operações lógicas do C, como: OR,AND,XOR. Também foram utilizadas as operações aritméticas, como +,-,x,/. Com essas

operações e com os comparadores(<, >, =, <=, >=) podemos implementar todas as instruções determinadas no roteiro do trabalho, logo não foi enfrentada nenhuma dificuldade em implementar as instruções, pois podem ser resolvidas com as próprias operações em C.

A maioria das operações MIPS foram resolvidas com os operadores lógicos com o próprio C, apenas tomando o cuidado necessário com a questão das operações unsigned, de sinal extendido e complemento de 2.

3 Testes

Os testes realizados foram:

3.1 Teste 1 - Exemplo Fibonacci

O código disponibilizado no roteiro foi executado com sucesso e obteve os mesmos resultados que o simulado MARS. Segue abaixo:

```
# Compute first twelve Fibonacci numbers and put in array, then print
        .data
fibs:   .word   0 : 12          # "array" of 12 words to contain fib values
size:   .word   12              # size of "array"
        .text
        la    $t0, fibs         # load address of array
        la    $t5, size         # load address of size variable
        lw    $t5, 0($t5)       # load array size
        li    $t2, 1            # 1 is first and second Fib. number
        sw    $t2, 0($t0)       # F[0] = 1
        sw    $t2, 4($t0)       # F[1] = F[0] = 1
        addi  $t1, $t5, -2      # Counter for loop, will execute (size-2)
```

```
loop: lw   $t3, 0($t0)      # Get value from array F[n]
      lw   $t4, 4($t0)      # Get value from array F[n+1]
      add  $t2, $t3, $t4    # $t2 = F[n] + F[n+1]
      sw   $t2, 8($t0)      # Store F[n+2] = F[n] + F[n+1] in array
      addi $t0, $t0, 4      # increment address of Fib. number source
      addi $t1, $t1, -1     # decrement loop counter
      bgtz $t1, loop        # repeat if not finished yet.
      la   $a0, fibs        # first argument for print (array)
      add  $a1, $zero, $t5  # second argument for print (size)
      jal  print            # call print routine.
      li   $v0, 10          # system call for exit
      syscall               # we are out of here.

########## routine to print the numbers on one line.

      .data
space:.asciiz  " "          # space to insert between numbers
head: .asciiz  "The Fibonacci numbers are:\n"
      .text
print:add  $t0, $zero, $a0  # starting address of array
      add  $t1, $zero, $a1  # initialize loop counter to array size
      la   $a0, head        # load address of print heading
      li   $v0, 4           # specify Print String service
      syscall               # print heading
out:  lw   $a0, 0($t0)      # load fibonacci number for syscall
      li   $v0, 1           # specify Print Integer service
      syscall               # print fibonacci number
      la   $a0, space       # load address of spacer for syscall
      li   $v0, 4           # specify Print String service
      syscall               # output string
      addi $t0, $t0, 4      # increment address
      addi $t1, $t1, -1     # decrement loop counter
      bgtz $t1, out         # repeat if not finished
      jr   $ra              # return
```

Resultado no Terminal: The Fibonacci numbers are: 1 1 2 3 5 8 13 21 34 55 89 144

3.2 Teste 2 - Exemplo Números Primos

Exemplo também realizado com êxito.

```
.data
primos: .word 1,3,5,7,11,13,17,19
size:   .word 8
msg:    .asciiz "Os oito primeiros numeros primos sao: "
space:  .asciiz " "

.text
        la $t0,primos
        la $t1,size
        lw $t1,0($t1)
        li $v0,4
        la $a0,msg
        syscall

loop:
        beq $t1,$zero,exit
        li $v0,1
        lw $a0,0($t0)
        syscall
        li $v0,4
        la $a0,space
        syscall
        addi $t0,$t0,4
        addi $t1,$t1,-1
        j loop

exit:
        li $v0,10
        syscall
```

Resultado no Terminal: Os oito primeiros numeros primos sao : 1 3 5 7 11 13 17 19

4 Conclusão

O projeto do simulador atingiu os resultados esperados assim como atingiu seus objetivos de entender o funcionamento da arquitetura de processadores MIPS, a partir da sua organização e implementaçãpo de instruções, tanto em linguagem assembler como em linguegem de máquina.

Com os testes realizados e compilados, o programa desenvolvido em C executa e simula tanto passo a passo como de uma vez só, assim como imprime a memória e registradores e atende todas as especificações descritas no roteiro do trabalho disponibilizado no moodle.

5 Referências

[1] D. A. Patterson e J. L. Hennessy. Organização e Projeto de Computadores - Interface Hardware/Software, Capítulo 2, 2005.