Universidade de Brasília

Organização e Arquitetura de Computadores

Engenharia de Computaçao

Trabalho III - Simulador MIPS

Aluno:
Matheus Veleci dos Santos
12/0130122

Professor:
Ricardo Jacobi

5 de outubro de 2015

1 Objetivo

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

2 Implementação

O código foi desenvolvido em C no sistema operacional MAC OS X 10.11 e foi utilizado um arquivo MakeFile para compilar o projeto(instrucoes para compilar no README.txt). O trabalho foi divido em 5 bibliotecas com funções pertinentes a cada etapa de desesnvolvimento do trabalho. Foi divido em:

- Carregar dados
- Funções de acesso a Mémoria
- Operações Simples
- Instruções

Esse projeto viabiliza 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 foi implementado duas funções: LoadText e LoadData.

A função LoadText lê o arquivo binário gerado pelo simulador MARS da parte a memória .text, que representa a memória de programa onde há 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 mesma instruções da função LoadText porém lê o arquivo que contém o conteúdo da memória de dados e os salva

na parte da memória reservado para dados que começa no endereço de memória 0x0002000.

2.2 Funções de acesso a Mémoria

As funções de acesso a memório 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 qunado 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 com a diferença de que os dados são salvos e não retornados.

A função load half-word e store half-word seguem a mesma lógica do endereçamento em bytes porém tem um detalhe, somente multiplos 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 multíplos 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 direfença 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 foi implementado básicamente 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 a instruções com seus campos (decode) e por fim executa a instrução em uma Unidade Lógica Aritmética ou outro circuito

Figura 1:

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

lógico.

A instrução **fetch** tem como unico 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. Para exemplificar os campos das instruções temos a figura 1[1].

A instrução **execute** é a principal função dessa biblioteca e possui todas a instrução aritméticas e lógicas do tipo R, I e J. Essa instrução ao seleciona qual instrução o processador deve executar de acordo com a sua OPCODE. Se sua 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 função 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 termino 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. E também das 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.

Figura 2:

```
enum OPCODES {  // lembrem que so sao considerados os 6 primeiros bits dessas constantes

        EXT=0x00,       LW=0x23,        LB=0x20,        LBU=0x24,
        LH=0x21,        LHU=0x25,       LUI=0x0F,       SW=0x2B,
        SB=0x28,        SH=0x29,        BEQ=0x04,       BNE=0x05,
        BLEZ=0x06,      BGTZ=0x07,      ADDI=0x08,      SLTI=0x0A,
        SLTIU=0x0B,     ANDI=0x0C,      ORI=0x0D,       XORI=0x0E,
        J=0x02,         JAL=0x03
};
```

Figura 3:

```
enum FUNCT {
        ADD=0x20,       SUB=0x22,       MULT=0x18,      DIV=0x1A,       AND=0x24,
        OR=0x25,        XOR=0x26,       NOR=0x27,       SLT=0x2A,       JR=0x08,
        SLL=0x00,       SRL=0x02,       SRA=0x03,       SYSCALL=0x0c,   MFHI=0x10, MFLO=0x12
};
```

Foi implementado as instruções com as seguintes OPCDODEs e FUNCT (Figura 2 e 3).

A maioria dessa foram resolvido como dito anteriormente com os operadores lógicos com o próprio C, apenas tomando o cuidado necessário à questão das operações unsigned e de sinal extendido e complemento de 2.

3 Testes e Resultados

Os teste realizado foram:

3.0.1 Teste 1 - Exemplo Roteiro

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

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

.text
```

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

loop:
beq $t1, $zero, exit #se processou todo o array, encerra
li $v0, 1 #servico de impressao de inteiros
lw $a0, 0($t0) #inteiro a ser exibido
syscall
li $v0, 4 #imprime separador
la $a0, space
syscall
addi $t0, $t0, 4 #incrementa indice array
addi $t1, $t1, -1 #decrementa contador
j loop #novo loop

exit:
li $v0, 10
syscall
```

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

3.0.2 Teste 2 - Exemplo Fibonacci

Exemplo do Fibonacci também realizado com êxito.

```
        .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) ti
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

3.0.3 Teste 3 - Soma Recursiva

Exemplo realizado com êxito.

```
.text
        addi $a0, $zero, 20

        jal Soma_recursiva

        add $a0, $zero, $v0
        li $v0, 1
        syscall

        li $v0, 10
        syscall

Soma_recursiva:
        addi $sp, $sp, -8       # prepara a pilha para receber 2 itens
        sw $ra, 4($sp)          # empilha $ra (End. Retorno)
        sw $a0, 0($sp)          # empilha $a0 (n)
        slti $t0, $a0, 1        # testa se n < 1
        beq $t0, $zero, L1      # se n>=1, va para L1
        add $v0, $zero, $zero   # valor de retorno e 0
        addi $sp, $sp, 8        # remove 2 itens da pilha
        jr $ra                  # retorne para depois de jal
L1:
```

```
addi $a0, $a0, -1        # argumento passa a ser (n-1)
jal Soma_recursiva       # calcula a soma para (n-1)
lw $a0, 0($sp)           # restaura o valor de n
lw $ra, 4($sp)           # restaura o endereco de retorno
addi $sp, $sp, 8         # retira 2 itens da pilha.
add $v0, $a0, $v0        # retorne n + soma_recursiva(n-1)
jr $ra                   # retorne para a chamadora
```

Resultado no Terminal:
210

3.0.4 Teste 4 - Multiplicação e Divisão

Exemplo realizado com êxito.

___

```
.data
x: .word 0x000fffff
y: .word 0x0f0f0f00

.text
        la $t0,x
        la $t1,y
        lw $t0,0($t0)
        li $t2, 20568714
        ori $t3,$t2,0xf0f0f0f0
        lw $t1,0($t1)
        mult $t0,$t1
        div $t0,$t1
```

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 implementação 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.

Referências

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

[2] Roteiro Trabalho III - Simulado MIPS. Disponibilizado no Moodle - OAC C