Universidade Federal
do Espírito Santo

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Arquitetura de Computadores I – Turmas 01 e 02 (EARTE) – 2021/2
Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br

Laboratório III – A máquina Virtual MARS e as Variáveis de Programa

Integrantes do Grupo:
- Dionatas Brito
- Maria Julia Damasceno
- Otávio Sales

Objetivos:

- Verificar como as instruções sintéticas ou pseudoinstruções MIPS podem ser reconhecidas no código;
- Compreender como as instruções sintéticas se expandem para sequências de instruções em linguagem de máquina;
- Entender como o MIPS endereça a memória de dados.

Descrição:

O MIPS baseia-se em uma arquitetura RISC "típica": possui um conjunto de instruções simples e regular, com apenas um endereçamento de memória modo (base mais deslocamento) e instruções são de tamanho fixo (32 bits). Surpreendentemente, é muito simples escrever programas de montagem para MIPS (e para qualquer máquina RISC).

A razão para isso é que a programação não deve ser feita em Linguagem Assembly. A ISA RISC foi projetada de tal forma a:

- Simplificar o trabalho dos compiladores;
- Permitir uma implementação em hardware mais eficiente.

Em algumas situações particulares, o programador poderá utilizar uma instrução em Linguagem Assembly MIPS que não têm correspondência direta com uma instrução em Linguagem de Máquina da CPU MIPS. Estas instruções que pertencem ao conjunto de instruções da linguagem de montagem MIPS e que não pertencem ao conjunto de instruções da máquina MIPS, implementadas pela CPU MIPS são chamadas de instruções sintéticas ou pseudoinstruções. O único objetivo destas instruções é "simplificar" a programação. Não há representação binária direta para as pseudoinstruções. Na verdade, o montador as substitui por uma ou mais instruções nativas equivalentes e que executam a lógica desejada.

Um exemplo bastante usado em programas é a carga de um endereço do segmento de dados em um registrador da CPU, como no caso da pseudoinstrução *la $t0, str1*. Esta instrução carrega o endereço reservado para *str1* (la=load address) no segmento de dados para o registrador *$t0*.

Responda:
Observe que se *str1* é um endereço, ele é representado em 32 bits e, portanto, não há como a instrução citada ser uma instrução nativa. Por quê?

> *Porque excede o valor de 16 bits, e é necessário usar duas instruções nativas, que são " lui" e "or"i, para conseguir armazenar nos registradores o valor de 32 bits*

Resposta:

1. Usando instruções sintéticas ou pseudoinstruções

Este exercício fará com que você se familiarize com instruções sintéticas no conjunto de instruções da máquina MIPS virtual. Existem muitas instruções sintéticas, mas só exploraremos algumas delas durante este laboratório. Veremos mais deles enquanto continuamos a explorar o conjunto de instruções em outras sessões de laboratório. A arquitetura MIPS é do tipo load/store e por isso:

- Todas as operações são realizadas em registros. O acesso à memória (de dados) é feito apenas pelas instruções load e store. Não há instrução aritmética ou lógica que tenha parte de operandos em registradores e parte na memória: operandos para tais instruções estão sempre em registradores da CPU;
- Todas as instruções aritméticas e lógicas possuem três operandos, um registrador de destinmo que é sempre listado imediatamente após o nome da instrução. Existem sempre dois registradores de origem. A instrução de máquina *add $t0, $t1, $t2* adiciona os registros de origem *$t1* e *$t2* e armazena o resultado no registro de destino *$t0*.

No programa MIPS a seguir, substitua a variável X pelos 3 últimos dígitos de sua matrícula:

```
.data 0x10010000
var1: .word 0x55          # var1 is a word (32 bit) with the initial value 0x55
var2: .word 0xaa

.text
.globl main
main: addu $s0, $ra, $0 # save $31 in $16
li $t0, X
move $t1, $t0
la $t2, var2
lw $t3, var2
sw $t2, var1

# restore now the return address in $ra and return from main
addu $ra, $0, $s0         # return address back
jr $ra                    # return from main
```

Atividade:
Salve o programa anterior como *lab2.asm* e o execute no MARS. Identifique instruções sintéticas e preencha a tabela a seguir. Lembre-se de que as instruções sintéticas são aquelas que estão no conjunto de instruções da linguagem de montagem mas não no conjunto de instruções da linguagem de máquina. Você pode reconhecê-los porque eles são diferentes na memória ao se comparar com o arquivo de origem. Às vezes há uma substituição de um para um (uma instrução sintética é substituída por uma instrução nativa), outras vezes duas ou mais instruções nativas são usadas para substituir uma instrução sintética. Acrescente quantas linhas na tabela forem necessárias! E lembre-se que o fato de que um nome de registrador (como *$t0*) é substituído por um número de registro (*$8*) não denota uma instrução sintética.

Resposta:

Endereço	Instrução sintética	Instruções nativas	Efeito
0x00400004	*li $t0, 307*	addiu $8,$0,0x00000133	$t0=307

0x00400008	move $t1, $t0	addu$9,$0,$8	$t1= $t0
0x0040000c	la $t2, var2	lui $1,0x00001001 ori $10,$0x00000004	$t2= var2
0x00400014	lw $t3, var2	lui $1,0x00001001 lw $10,$0x00000004($1)	$t3= var2
0x0040001c	sw $t2, var1	lui $1,0x00001001 sw $10,$0x00000000($1)	memória[$t2]=var1
0x00400024	-	addu$31,$0,$16	return address back
0x00400028	-	jr$31	return from main

Na coluna "Efeito", indique qual é a operação efetivamente realizada em cada instrução nativa. Exemplo:
add $t0, $t1, $t2 Efeito: *$t0 = $t1 + $t2*

2. Carregando um endereço em um registrador

A instrução *lui* é usada para carregar uma constante imediata de 32 bits em um registrador. Como todas as instruções são do mesmo tamanho (32 bits), não há como uma instrução inicializar um registrador com um valor imediato de 32 bits, pois a instrução "não caberia" nos 32 bits disponíveis para codificar a instrução de máquina. Sendo assim, foi criado um mecanismo para permitir a carga carregar uma constante de 32 bits em um registrador, baseado numa execução em duas etapas. Na primeira etapa, o mecanismo usa uma instrução do tipo *lui* para carregar 16 bits na parte superior do registrador (geralmente usa-se o registrador *$1*, que é um registro reservado para o montador). Estes 16 bits correspondem à parte "superior" da constante a ser armazenada no registrador. Na segunda etapa, uma instrução *ori* é utilizada para que se completem os 16 bits "inferiores" do valor da constante a ser armazenada no registrador.

Atividade:

Na tabela abaixo, insira a constante que você encontra com os primeiros *lui* e *ori* em seu programa, em formato hexadecimal. Explique o que estas constantes representam no seu programa.

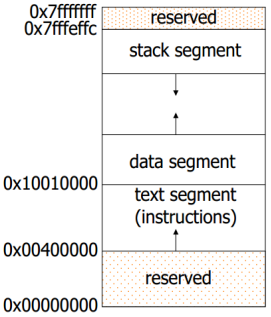
Instrução	Constante	Significado da constante
lui $1,0x00001001 ori $10,$0x00000004	0x10010004	Endereço inicial do código da palavra var2

Resposta:

`lui $1,0x00001001`	`10: la $t2, var2`
`ori $10,$1,0x00000004`	

3. Endereçamento em MIPS

A única maneira pela qual a CPU pode acessar a memória no MIPS é por meio de instruções do tipo load/store. Existe apenas um modo de endereçamento para os dados: base + deslocamento. Ter apenas um modo de endereçamento faz parte da filosofia RISC de manter as instruções simples, permitindo assim uma estrutura de controle simples e uma execução eficiente. A figura abaixo mostra o layout da memória para sistemas MIPS. O espaço do usuário é reservado para programas do usuário:



Os espaços iniciais não podem ser acessados diretamente pelos programas e são reservado para as chamadas de sistema (*syscall*). O segmento de texto contém o código do usuário em linguagem de máquina que o sistema computacional está executando.

O segmento de dados tem duas seções:

- Dados alocados de forma estática, que contém o espaço para variáveis estáticas e globais;
- Dados alocados de forma dinâmica, que é o espaço alocado para objetos de dados em tempo de execução (geralmente usando espaços alocados com instruções do tipo malloc() em C).

Atividades

a) Criar um programa em assembly MIPS que tenha as seguintes características:

- Reserve espaço na memória para quatro variáveis chamadas var1 a var4 do tamanho da palavra.
- Defina valores iniciais para estas variáveis
- Reserve espaço na memória para duas variáveis chamadas “primeiro” e "último" do tipo Byte. O valor inicial de primeiro deve ser a primeira letra do seu primeiro nome e o valor inicial do último deve ser a primeira letra do seu último nome.
- O programa troca os valores das variáveis na memória: o novo valor de var1 será o valor inicial de var4, o novo valor de var2 será o valor inicial de var3, var3 receberá o valor inicial de var2, e finalmente o var4 obterá o valor inicial de var1.

Você deve priorizar o uso dos registradores *$t0* a *$t8* em seu programa. Você pode usar o conjunto de instruções estendido com pseudoinstruções MIPS. Comente cada linha no programa com comentários indicando o que a instrução MIPS faz.

Resposta:

Código:

lab3.asm | Endereçamento em MIPS.asm*

```
#Reserve espaço na memoria para 4 variaveis (var1,var2,var3,var4) de tamanho da palavra

#1-passo = segmento de data
.data  0x10010000 # segmento de data, aqui é onde ira começar o segmento de data

#2-passo = criar as variaveis do tipo word (no segmento de data) com valores iniciais
var1: .word 4
var2: .word 8192
var3: .word 4096
var4: .word 2048

#3-passo = reservar espaço (no segmento data) para duas variaveis (primeiro e ultimo) do tipo Byte
#primeiro: .byte D
#ultimo: .byte B

#4-passo = fechamento do segmento data e inicio do segmento de texto
.text
.globl main
main: addu $s0,$ra,$0 #salvar 31 em 16

#5 passo = O programa troca os valores das variáveis na memória:
#o novo valor de var1 será o valor inicial de var4,
#o novo valor de var2 será o valor inicial de var3,
#var3 receberá o valor inicial de var2,
#e finalmente o var4 obterá o valor inicial de var1

#5.1 passso (carregar) nos registradores
#Os registradores estao armazendando os valores das variaveis tipo word, carregando a palavra nos registradores %t0 até $t3
lw $t0, var1
lw $t1, var2
lw $t2, var3  #lw destino, origem (levar da memoria para os registradores)
lw $t3, var4

#5.2 passo (mover)
#Apos carregar, os valores armazenados pelos registradores,
#irão ser movidos para outros registradores e assim ser possivel realizar a troca
move $t4,$t0
move $t5,$t1   # move destino,origem (mover)
move $t6,$t2
move $t7,$t3

#5.3 passo (mover inversamente)
#Os registradores 4-7, serao movidos para 0-3
move $t0,$t7
move $t1,$t6
move $t2,$t5
move $t3,$t4

#5.4 passo (mover para a memoria)
#Os registradores invertidos (0-3) serão levados para a memoria
sw $t0,var1
sw $t1,var2   # sw origem,destino (levar) -
sw $t2,var3
sw $t3,var4
```

Resultado;

$t0	8	2048
$t1	9	4096
$t2	10	8192
$t3	11	4

b) Encontre manualmente (faça as contas) o valor do deslocamento usado para determinar o endereço de cada variável no seu programa desde o início do segmento de dados. O deslocamento será a distância em Bytes entre o início do segmento de dados e o local onde a variável é armazenada. Em seguida, compare estes valores de deslocamento com os que aparecem na tabela de labels do seu programa no MARS. Use as chamadas de sistema (syscall) para verificar o que está armazenado na memória no segmento de dados (basta imprimir o que está na memória, comparando com os valores armazenados na tabela "Data Segment" do MARS). Apresente os resultados e comparações entre o valor esperado e o valor real.

Resposta:

Segmento de dados antes da troca

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10
0x10010000	4	8192	4096	2048	

Segmento de dados depois da troca

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	2048	4096	8192	4

Labels (de início)

Endereçamento em MIPS...	
var4	0x1001000c
var3	0x10010008
var2	0x10010004
var1	0x10010000

local de início: 0x10010000
var1: 0x1001000c -> deslocamento = c
var2: 0x10010008 -> deslocamento = 8
va3: 0x 10010004 -> deslocamento = 4
var4: 0x10010000 -> deslocamento = 0
Os valores de deslocamento da tabela de labels são iguais aos valores apresentados no segmento de dados.

c) Quantas instruções assembly e quantas instruções de máquina foram utilizadas no seu programa? Como você determina o tamanho do programa, em Bytes, a partir destas informações?

Resposta:
16 instruções em assembly e 16 instruções de máquina, como tem 32 instruções (total de 32 bits x 32 instruções=1024 bits) e cada instrução tem 4 Bytes, o tamanho total em Bytes do programa, seria 4x32 = 128 Bytes.

4. Execução

- Grupos de até 3 (três) alunos;
- Submissão até 09/12 (9h);