Aulas de laboratório com o simulador MARS

2018

Referência principal: Apostilas do curso do Prof. Virgil Bistriceanu, 1996
[*www.cs.iit.edu/~**virgil**/cs470/**Labs/***]

Aluno(a): Luiza Batista Laquini
Matrícula: 2019107786
Curso: Engenharia Da Computação

1. AULA 1 – Iniciando com o MARS e a linguagem assembly MIPS

Objetivos

Após esse lab, você deverá ser capaz de:

- Carregar e executar programas em assembly MIPS
- Examinar posições de memória
- Examinar registradores
- Executar programas passo-a-passo

O MARS é um simulador escrito em Java que roda programas para as arquiteturas baseadas em MIPS (por exemplo, R2000 e R3000) . O simulador pode carregar e executar programas em *assembly*.

O processo de translação do código MIPS é transparente para o usuário. Isto significa que você não tem que tratar com o *assembler*, *linker* e *loader* diretamente. Após você escrever seu programa em *assembly*, o simulador vai se encarregar de executá-lo e apresentar o resultado da execução em uma das saídas disponíveis (em geral, a saída padrão definida em uma das janelas do MARS).

ETAPA 1 – Primeiros passos com o MARS

Usando a janela de edição do MARS, edite o programa P1. Como visto em sala de aula, (#) inicia um comentário, um nome seguido (:) é um *label* (identificador) e os nomes que se iniciam com (.) são diretivas para o montador (*assembler*).

```
.data
msg1: .asciiz "Please enter an integer number: "
.text
        li $s0, 0x00400000  # save return adress in $16
        li $v0, 4 # system call for print_str
        la $a0, msg1 # address of string to print
        syscall
# now get an integer from the user
        li $v0, 5 # system call for read_int
        syscall # the integer placed in $v0
# do some computation here with the number
        addu $t0, $v0, $0 # move the number in $t0
        sll $t0, $t0, 8 # last digit of your UFES number
# print the result
        li $v0, 1 # system call for print_int
        addu $a0, $t0, $0 # move number to print in $a0
        syscall
# restore now the return address in $ra and return from main
        addu $ra, $0, $s0 # return address back in $31
        jr $ra # return from main
```

Meu código:

```
.data
msg1: .asciiz "Please enter an integer number: "
.text
        li $s0, 0x00400000  # save return adress in $16
        li $v0, 4 # system call for print_str
        la $a0, msg1 # address of string to print
        syscall
# now get an integer from the user
        li $v0, 5 # system call for read_int
        syscall # the integer placed in $v0
# do some computation here with the number
        addu $t0, $v0, $0 # move the number in $t0
        sll $t0, $t0, 6 # last digit of your UFES number
# print the result
        li $v0, 1 # system call for print_int
        addu $a0, $t0, $0 # move number to print in $a0
        syscall
# restore now the return address in $ra and return from main
        addu $ra, $0, $s0 # return address back in $31
jr $ra # return from main
```

Monte o programa usando o menu Run→Assemble, ou F3. Execute o programa. Não se esqueça de trocar o dígito da linha indicada para o seu último dígito da sua matrícula. Observe o resultado para diferentes números digitados como entrada. Preencha a tabela com diversos valores entrada e de saída (inteiros, positivos e negativos) para a execução do programa.

Entrada	Saída
-2	-128
-1	-64
0	0
2	128
4	256

- Qual é a equação que define a transformação dos valores de entrada para saída?

R: y = 64x, sendo y o valor de saída e x o valor de entrada.

- Qual o valor armazenado em `$s0` e por que ele foi passado a `$ra`?

R: O valor é 0x00400000. Sabendo que $s0 é o início do programa, seu valor foi passado ao registrador temporário $ra para que fosse possível realizar um "jump" de volta para $s0 configurando um "reset" do programa.

ETAPA 2 – Usando o MARS para entender a arquitetura

Usando o MARS, é possível se observar a ocupação da memória e dos registradores, assim como o endereçamento de instruções e dados de um programa MIPS. Também é possível fazer a execução de um programa passo-a-passo, permitindo a observação detalhada da variação dos valores dos registradores e do fluxo de execução dos programas.

Etapa 2.1 – Modifique o programa Lab-P1, incluindo um *label* `L1` na 2ª linha do segmento de texto, que passará a ser:

```
L1: li $v0, 4 # system call for print_str
```

Observe a tabela de símbolos do programa. Dois *labels* devem aparecer (`L1` e `msg1`). Anote os valores desses *labels* e explique como esses valores foram definidos.

Símbolo	Valor
L1	0x24020004
msg1	0x61656c50

Observe a tabela de símbolos do programa. Dois *labels* devem aparecer (`L1` e `msg1`). Anote os valores desses *labels* (eles estão em hexadecimal, prefixados sempre com #0) e explique como esses valores foram definidos.

R: L1 foi definido pelo armazenamento da instrução que foi passada em seguida (li $v0, 4). Já msg1 foi definido pelo armazenamento dos 4 primeiros bytes da mensagem que foi passada (Plea – entretanto, na ordem inversa "a e l P").

Etapa 2.2 – Neste ponto, você já sabe como um programa é armazenado, como os valores dos labels são definidos e como os valores dos registradores são modificados. Importantes questões podem ser respondidas agora:

- Qual o tamanho das instruções do programa? Quantos bytes no total ele ocupa?

R: No MIPS, possui 32 bits ou 4 bytes.

- A instrução `li $s0, 0x00400000` tem qual propósito? E se esse valor fosse mudado, o que aconteceria?

R: li ("load immediate") passa o valor 0x00400000 diretamente para $s0, que é o ponto de partida.

- Essa instrução foi substituída por duas instruções. Quais? Por que isso foi necessário?

R: Foram as instruções: `lui $1, 64` e `ori $16, $1, 0` - que foram necessárias pois a instrução `li $s0, 0x00400000` ultrapassa a capacidade de armazenamento de 32 bits, sendo necessário uma maneira de armazenar números maiores. Isso se torna possível com essas duas novas instruções, em que "lui" faz um "shift-left" armazenando os maiores valores à esquerda e "ori" faz um "shift-right" armazenando os menores valores à direita.

- A instrução `la $a0, msg1` também foi substituída por 2 instruções. Quais? Por que isso foi necessário?

R: Foram as instruções: `lui $1, 4097` e `ori $4, $1, 0` - necessárias pelo mesmo motivo da `li $s0, 0x00400000`.

Etapa 2.3 – É possível fazer a execução passo-a-passo do programa usando o botão  ou a tecla F7. Execute o programa passo-a-passo e observe a mudança nos valores do registrador PC, assim como a instrução executada no momento, além das mudanças nos valores dos registradores utilizados no código.
Obs: O registrador PC muda a cada passo executado, ganhando 4 bytes.

Etapa 2.4 – Em 3 pontos do programa a instrução `syscall` foi utilizada. Note que essa instrução faz a chamada de um serviço do Sistema Operacional (SO) e que alguns argumentos devem ser passados ao SO antes da chamada. Anote na tabela a seguir os registradores que são alterados em cada uma das instruções `syscall` do programa, assim como os valores dos argumentos usados como entrada e saída nas chamadas.

Endereço	Tipo serviço	Input	Output
0x00400014	print string	-	$a0: 0x10010000
0x0040001c	armazenamento int	$v0: 0x00000005	-
0x00400030	print int	-	$a0: 0x00000040

Etapa 2.5 – Por questões práticas de projeto, a CPU MIPS foi projetada com:

- Uma unidade de inteiros (ULA dentro da CPU);
- Um coprocessador 0, com suporte ao tratamento de interrupções, exceções e memória virtual;
- Um coprocessador 1, correspondendo a uma unidade de operações em ponto flutuante.

Estes tópicos ainda não foram vistos no curso, porém os coprocessadores já podem ser testados. Para isso:

- Escolha a aba Coproc 0 na janela direita do simulador. Digite um valor de entrada "muito grande" e observe o erro de execução e sua indicação em um dos registradores. Qual registrador indica o erro? Qual o erro foi indicado? Como essa indicação poderia ser usada pelo SO?

R: O registrador que indica o erro é o $14 (epc). O erro indicado foi `"invalid integer input (syscall 5)"`. Essa indicação poderia ser utilizada pelo SO para evitar uma quebra do ciclo de Busca-Execução.

- Inclua uma declaração da variável x no segmento de dados com o comando `x: .float 1.5674`. Observe a tabela de labels do programa. O valor da variável `x` está correto? Que representação é essa?

R: Sim, o valor de x está correto. O valor dado (0x3fc8a090) é a representação do float (1.5674) em ieee 754.

Inclua as instruções a seguir no segmento de código após o último `syscall`:

```
l.s $f0, x
add.s $f2, $f0, $f0
```

Escolha a aba Coproc 1 na janela direita do simulador. Explique os valores que foram gerados nos registradores do coprocessador Ponto-flutuante MIPS.

R: O registrador $f0 simplesmente recebeu e armazenou o valor de x. Já o $f2 recebeu e armazenou x+x, ou seja, 2x.