Sugestões no Projeto do Simulador MIPS

Variáveis do Programa

No simulador, serão necessários 3 segmentos de memória: segmento para instruções, segmento de dados e o segmento da pilha. Temos que definir o endereço inicial e o endereço final de cada um destes segmentos de memória. Podemos usar a diretiva **.eqv** para definirmos estes endereços. Esta diretiva é semelhante a diretiva **#define** da linguagem C. Os valores padrões inicialmente usados nos registradores (PC, $sp e $gp) também podem ser definidos por meio desta diretiva. A seguir, apresentamos um exemplo de definição para os endereços de memória e para os valores padrão (default) de alguns registradores.

```
.text
# definicoes dos endereços e valores padrão do simulador
# endereço inicial e final dos segmentos de memória para instruções, dados e a pilha
.eqv  ei_memoria_instrucoes  0x00400000      #  endereço  inicial  da  memória  de
instruções
.eqv ef_memoria_instrucoes 0x0040FFFF   # endereço final da memória de instruções
.eqv ei_memoria_dados      0x10010000   # endereço inicial da memória de dados
.eqv ef_memoria_dados      0x1001FFFF   # endereço final da memória de dados
.eqv ei_memoria_pilha      0x7FFF0000   # endereço inicial da memória da pilha
.eqv ef_memoria_pilha      0x7FFFFFFF   # endereço final da memória da pilha

# valores padrão (default) para alguns registradores
.eqv PC_DEFAULT            0x00400000   # valor padrão do PC (program counter, contador de programa)
.eqv SP_DEFAULT            0x7FFFEFFC   # valor padrão do SP (stack pointer, apontador para a pilha)
.eqv GP_DEFAULT            0x10008000   # valor padrão do GP (global pointer, apontador global)
```

Registradores e Memória

Para o programa simulador, será necessário definir variáveis para os segmentos de memória, registradores e os campos dos registradores. Um exemplo é apresentado a seguir[1]:

```
.data

#################################################################################
# variáveis para o simulador
#################################################################################

# registradores de uso geral
.align 2
registradores:  .space 128          # 32 registradores de uso geral (registradores de 32 bits)

# outros registradores
```

1 Veja o arquivo dicas.s.

```
PC:              .space 4          # contador de programa (contém o endereço da instrução atual)
IR:              .space 4 # registrador de instrução (contém a instrução que é decodificada e executada)
hi:              .space 4        # registrador hi, usado nas operações de multiplicação e divisão
lo:              .space 4        # registrador lo, usado nas operações de multiplicação e divisão

# campos do registrador IR, usados na decodificação e execução
IR_campo_op:     .space 4            # campo opcode ou op - código de operação
IR_campo_rs:     .space 4            # endereço (número) do registrador rs
IR_campo_rt:     .space 4            # endereço (número) do registrador rt
IR_campo_rd:     .space 4            # endereço (número) do registrador rd
IR_campo_shamt:  .space 4            # campo usado nas operações de deslocamento
IR_campo_funct:  .space 4            # usado para completar a decodificação da instrução
IR_campo_imm:    .space 4            # campo imediato, um valor de 16 bits
IR_campo_j:      .space 4            # usado nas operações j, possui 26 bits

# segmentos de memória para as instruções, dados e pilha

# segmento de memória      (endereço final - endereço inicial) + 1
# as memórias são organizadas em bytes.
memoria_instrucoes: .space 65536    # segmento de memória para instruções (.text)
memoria_dados:      .space 65536    # segmento de memória para os dados (.data)
memoria_pilha:      .space 65536    # segmento de memoria da pilha
```

No código anterior, o banco de registradores de uso geral ($0 a $31) e os segmentos de memória são variáveis tipo vetores. A variável registradores é um vetor de palavras (inteiros com 4 bytes). Os segmentos de memória são representados por vetores de bytes. Escrever ou ler na variável registradores é semelhante a ler e escrever um inteiro em um vetor de inteiros. Escrever ou ler na variável memoria_instrucoes, memoria_dados ou memoria_pilha é semelhante a escrever um caractere em uma *string*.

Na execução das instruções, vamos precisar ler e escrever registradores e endereços de memória corretamente. Estamos apresentando no arquivo dicas.s estas funções:

- leia_registrador – faz a leitura de um registrador do banco de registradores. O banco de registradores possui 32 registradores de uso geral. O dado lido é uma palavra de 32 bits (4 bytes).
- escreve_registrador – faz a escrita de um dado (palavra com 4 bytes) em um dos registradores.
- escreve_memoria – escreve um byte em um dos segmentos de memória. O procedimento verifica se o endereço pertence a um segmento de memória. Se pertence, o dado é escrito. Se o endereço não está contido em nenhum dos segmentos de memória, uma mensagem de erro é apresentado no console do simulador MARS. Neste procedimento basta indicar o endereço e o dado. Ele é transparente. Não precisamos nos preocupar em identificar qual o segmento de memória do endereço.
- leia_memoria – Este procedimento é semelhante ao procedimento de escrita na memória. Basta colocar o endereço como argumento. Pelo endereço, o procedimento encontra qual a variável representado os segmentos de memória deve ser usada e realiza a leitura. Se o endereço não existe nos segmentos de memória definidos, o procedimento escreve uma mensagem de erro no console do simulador MARS.

Procedimento Principal (main)

O procedimento principal (main) pode chamar apenas 3 procedimentos:

- inicializa_variaveis: usados para colocar os valores padrões nos registradores.
- leia_linha_keyboard: lê uma linha da ferramenta keyboard and display MMIO simulator. Esta linha pode ser armazenada em uma string (buffer_linha).
- executa_comando: decodifica a linha lida de keyboard e executa o comando lt, ld, m, d ou r, se ele existe e está correto.

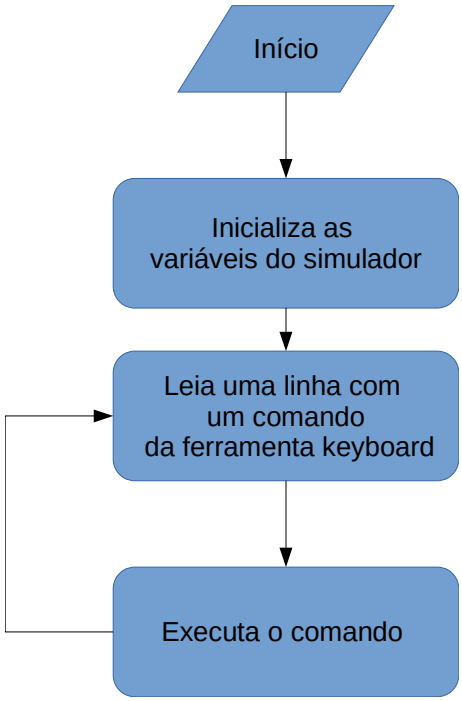

Figura 1 – Fluxograma do procedimento main.

Para trabalhar com a ferramenta keyboard and display MMIO simulator, do simulador MARS, veja os procedimentos leia_linha e imprime_string do arquivo keyboard_display.s:

leia_linha – Este procedimento lê uma linha da ferramenta keyboard. O retorno deste procedimento ocorre no momento que pressionamos a tecla <enter>. A linha é armazenada temporariamente em um buffer (como uma string).
imprime_string – Faz uma string aparacer na tela da ferramenta display.

Procedimento para executar um comando

O fluxograma para o procedimento que executa os comandos do simulador é apresentado na figura a seguir.

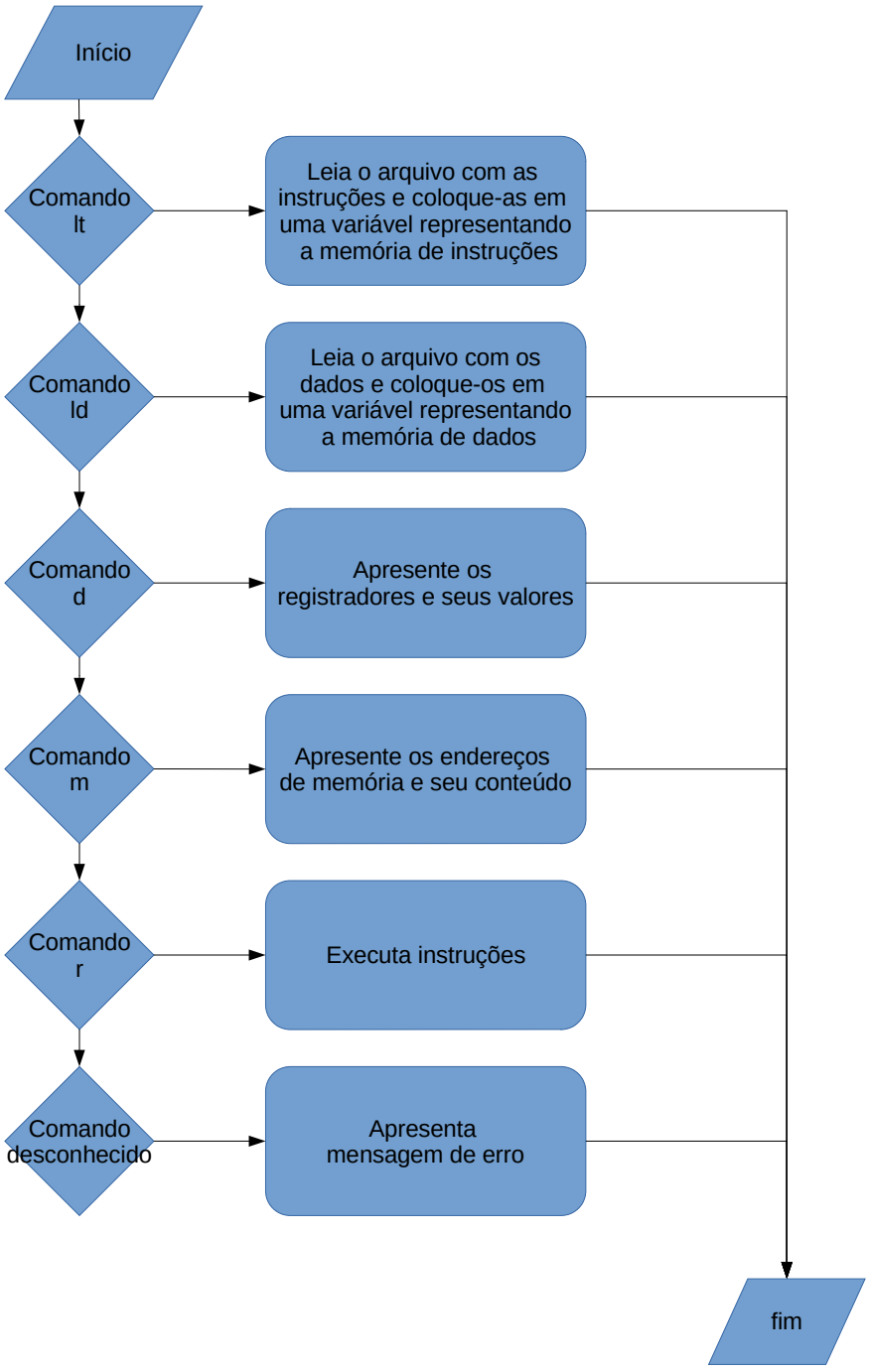

Figura 2: Fluxograma do procedimento usado para executar um comando do simulador.

- Comando lt – lê um arquivo e armazena na variável representando o segmento de instruções (.text). A primeira instrução deve ser colocada no endereço 0x00400000. As instruções são armazenadas no formato *little endian*. Se desejar, usar o procedimento leia_memoria.
- Comando ld – lê um arquivo e armazena os bytes na variável representando o segmento de dados (.data). O primeiro byte deve ser armazenado no endereço 0x10010000. Se desejar usar o procedimento escreve_memoria.
- Comando d – fazemos a leitura (palavras de 4 bytes) de cada elemento da variável representado os registradores e imprimimos o número do registrador e o seu conteúdo. Usar o procedimento imprime_string.
- Comando m – fazemos a leitura de endereços de memória e apresentamos na ferramenta display. Usar o procedimento leia_memoria para fazer a leitura do conteúdo da memória e imprime_string para apresentar os valores.
- Comando r – este é o comando mais importante do projeto. Executamos as instruções armazenadas na variável representando o segmento de instruções. Este procedimento lê uma instrução, decodifica e executa a instrução. Cada uma destas tarefas pode ser realizada com um procedimento.

Os comandos lt, ld, e r possuem 1 argumento: um nome de arquivo ou um valor numérico. O comando m tem 2 argumentos: o endereço inicial e o número de endereços, desde o endereço inicial, que queremos apresentar na ferramenta display. Os valores numéricos, argumentos destes comandos, são substrings. Temos que converter estas substrings em seu correspondente valor numérico, em um número. No arquivo dicas.s apresentamos dois procedimentos para realizar a conversão de string para um valor numérico:

- converte_string_decimal – converte uma string decimal em um valor numérico.
- converte_string_hexadecimal_para_decimal – converte uma string hexadecimal em um valor numérico.

Nestes procedimentos, os argumentos são dois endereços: endereço da variável string com o valor a ser convertido e endereço de uma variável (inteiro com 4 bytes) contendo o estado da conversão. Lemos a string com o seu endereço. Realizamos a conversão para um valor numérico. Se a conversão da string para um valor numérico não pode ser realizada, o valor retornado da conversão será 0. Para verificarmos se este valor é o resultado de uma conversão ou se é porque a conversão não pôde ser realizada, usamos o endereço de uma variável inteira. Se a conversão foi realizada com sucesso, armazenamos o valor 1, senão, 0. Para verificar se o valor da conversão pode ser utilizado, basta fazer a leitura desta variável.

Procedimento para executar instruções

O fluxograma do procedimento para simular a execução das instruções é apresentado na figura 3.

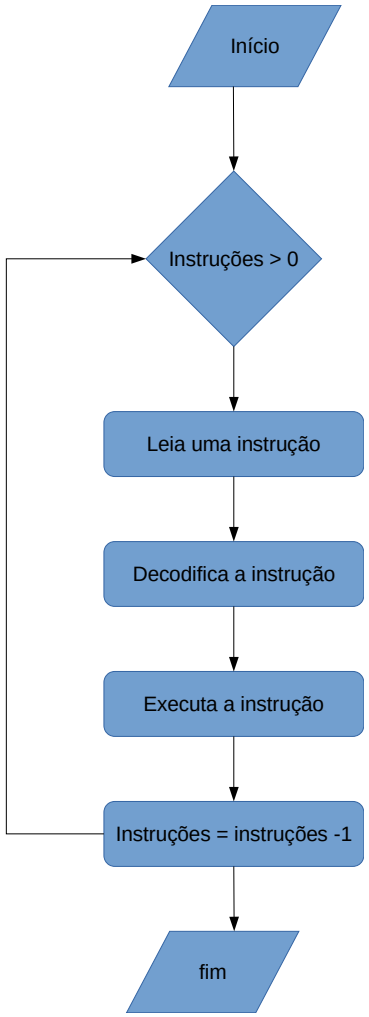

Figura 3: Fluxograma do comando para simular a execução das instruções em linguagem de máquina.

Se existem instruções que serão simuladas, executamos 3 procedimentos:

- Busca da instrução
- Decodificação da instrução
- Execução da instrução

Busca da instrução

Leia uma instrução de um endereço de memória e armazene no registrador IR[2] (*instruction register* ou registrador de instruções). O registrador IR contém a instrução atual, que é decodificada e executada. Usamos o procedimento leia_memoria, do arquivo dicas.s, para ler os endereços de memória. O endereço de memória da instrução está no registrador PC. Precisamos ler 4 bytes da memória. Como a memória é organizada em bytes, vamos chamar 4 vezes o procedimento leia_memoria. Cada byte lido é armazenado na posição correta no registrador IR. Usamos o formato little endian, o contendo dos endereços menores de memória são armazenados nos bytes menos significativos do registrador IR:

```
IR[7:0] ← leia_memoria(PC)
IR[15:8] ← leia_memoria(PC+1)
IR[23:16] ← leia_memoria(PC+2)
IR[31:24] ← leia_memoria(PC+3)
```

Decodificação da instrução

Neste procedimento, incrementamos o registrador PC e extraímos todos os campos da instrução:

- opcode ou código de operação
- registrador rs
- registrador rt
- registrador rd
- campo shamt
- campo funct
- campo imediato
- campo imediato para instruções tipo j

Cada um dos campos é colocado na sua respectiva variável. Para extrair o campo usamos um deslocamento para a direita e em seguida realizamos a operação AND com uma máscara[3]. Por exemplo, para extrairmos o campo rt (veja a figura 4), realizamos as seguintes operações[4]:

```
la      $t0, IR             # $t0 <- endereço da variável IR
lw      $t1, 0($t0)         # carregamos o valor de IR em $t1
srl     $t2, $t1, 16        # deslocamos IR para a direita em 16 bits
andi    $t2, $t2, 0x001F    # aplicamos uma máscara isolando os 5 bits de rt
la      $t0, IR_campo_RT    # $t0 <- endereço da variável que armazena o campo rt
sw      $t2, 0($t0)         # armazenamos o campo rt
```

2 Este registrador não aparece explicitamente no simulador MARS.
3 Ou isolamos os bits do campo e aplicamos um deslocamento para a direita.
4 Se carregamos em IR a instrução 0x014B4820 = add $t1, $t2, $t3, após a execução das instruções, temos armazenado em IR_campo_RT o valor 0x0B (11). Este valor é igual ao valor de rt na instrução: $11 = $t3.

Na fase de decodificação não conhecemos o que a instrução realiza, por isso, todos os campos possíveis devem ser extraídos do registrador IR. Para a decodificação das instruções utilize como referência das instruções o apêndice B.10 do livro texto.

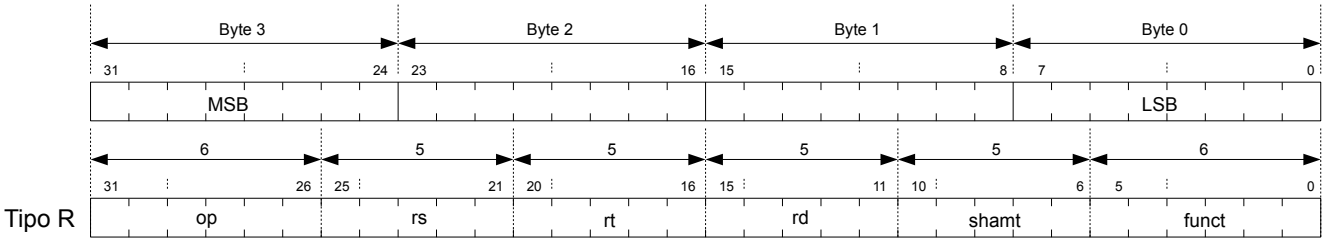

Figura 4: Ilustração de uma instrução tipo R, com o campo rt. Nas outras instruções, este campo sempre está na mesma posição, entre os bits 20 e 16.

O campo imediato possui 16 bits. Este campo representa normalmente um valor com sinal. Quando armazenamos em uma variável de 32 bits, ou nas operações, que são realizadas em 32 bits, precisamos fazer a extensão do sinal. Para fazer a extensão, replique o bit mais significativo do campo imediato, bit 15, para os outros bits mais significativos do número estendido (bits 31 a 16).

No procedimento para a decodificação da instrução, também incrementamos o valor de PC. No simulador, as instruções são de 4 bytes e ao valor de PC adicionamos o valor 4.

```
PC = PC + 4
```

Execução da Instrução

No procedimento para execução usaremos os campos extraídos do procedimento anterior, o procedimento de decodificação da instrução. Tomamos o campo opcode ou op. De acordo com o seu valor, chamamos um determinado procedimento[5] que executa a instrução. Na execução da instrução, usamos os outros campos extraídos na decodificação e possivelmente a leitura e escrita nos registradores e memória. Para a leitura e escrita dos registradores e memória podemos usar os procedimentos: leia_registrador, escreve_registrador, leia_memoria e escreve_memoria, do arquivo dicas.s. Para algumas instruções, será necessário observar outros campos para determinarmos a instrução que deve ser executada. Exemplos são instruções com o campo opcode 0 ou 1C. Nos dois casos a instrução somente será reconhecida após a análise do campo funct. Por exemplo, todas as instruções R tem o campo opcode igual a 0. A operação sobre os registradores rs e rt é conhecida após a análise do campo funct. Por exemplo, se o campo funct possui o valor 0x20, a operação será de adição.

5 Equivalente às microoperações do processador.

Vamos apresentar um exemplo de execução de uma instrução do simulador. Seja a instrução 0x8d090000 = lw $t0, 0($t1). Na fase de execução verificamos o valor do campo opcode ou op. Neste exemplo, o valor do opcode é 0x23. Qualquer instrução que tenha o opcode 0x23 chama o procedimento para executar a instrução lw (vamos chamar este procedimento de executa_instrucao_lw). Este procedimento utiliza os campos encontrados na fase de decodificação. Neste procedimento executamos as seguintes operações:

- Lemos o conteúdo do registrador rs. O número do registrador rs está em um dos campos extraídos do registrador de instruções (IR), na fase de decodificação.

```
eb ← registradores[rs]
```

- Encontramos o endereço efetivo (ef) da instrução lw, somando o valor do registrador rs com o campo imediato. O sinal do campo imediato deve estar estendido para 32 bits.

```
ef ← eb + campo_imediato[31:0]
```

- Lemos o conteúdo do registrador rt. O número do registrador rt está em um dos campos extraídos do registrador IR, no procedimento de decodificação da instrução.

```
dado ← registradores[rt]
```

- Escrevemos o conteúdo do registrador rt no endereço efetivo ef. O conteúdo do registrador rt é uma palavra com 4 bytes e a memória é organizada em bytes: um byte por endereço. Para escrevermos a palavra na memória podemos usar 4 vezes o procedimento leia_memoria do arquivo dicas.s. Escrevemos os bytes do registrador rt, do byte menos significativo para o mais significativo, nos endereços ef, ef+1, ef+2 e ef+3.

```
escreve_memoria(ef) ← dado[7:0]
escreve_memoria(ef+1) ← dado[15:8]
escreve_memoria(ef+2) ← dado[23:16]
escreve_memoria(ef+3) ← dado[31:24]
```

Terminamos o procedimento da execução desta instrução e o ciclo se repete para as próximas instruções: busca, decodificação e execução das instruções.