Trabalho 1 de Organização e Arquitetura de Computadores

Marcos Paulo Cayres Rosa (14/0027131)
Departamento de Ciência da Computação
Organização e Arquitetura de Computadores
Brasília, Brasil

***Abstract*—Esse documento tem como finalidade demonstrar como foi construído um simulador de alguams das funções do Mars Mips em linguagem C (*Abstract*)**

Keywords—mips, mars, simulador, assembly

I. INTRODUÇÃO

O problema apresentado consiste em desenvolver 8 funções básicas do Assembly e simular sua execução no Mars Mips, considerando os registradores que podem estar em uso e, especialmente, as modificações na memória de instruções e dados. Para tal deveria se criar um array que correspondesse as 16384 posições da memória, tendo em vista que cada uma representa 1 byte e considerando que na simulação cada um dos 4096 índices terá 32 bits (ou 4 bytes).

Além dos detalhes apresentados anteriormente, também era necessário desenvolver códigos ASM para gerar instruções e dados ("text.bin" e "data.bin", respectivamente) que seriam salvos em binário e, após isso, na memória do simulador. Por fim, deve-se imprimir e comparar os resultados obtidos diretamente pelo MIPS e os simulados em C.

A cerca dos detalhes do código fonte, o programa foi compilado e executado através de um sistema Ubuntu 14.04.1, sem o uso de IDEs, apenas editores básicos de texto. Ademais, o código foi escrito em linguagem C e compilado com o gcc 4.8.4.

II. INSTRUÇÕES

Para compilação e execução do programa é preciso seguir os passos descritos abaixo:

1. Utilizando um terminal Ubuntu, abrir a pasta na qual se encontra o arquivo 140027131.c e mips.h;
2. Efetuar o comando a seguir no terminal: "gcc 140027131.c -o Nome_Executavel". Observação: durante os testes o programa foi compilado com "gcc 140027131.c -ansi -Wall -o Nome_Executavel" para garantir que este segue o padrão Ansi e que não há nenhum "warning flag" ativado;
3. Executar o arquivo criado com "./Nome_Executavel";
4. Durante a execução será necessário que o usuário aperte "Enter" entre as diferentes etapas (impressão das informações advindas dos binários, efetivazação das instruções do MIPS - testes das funções, impressão dos resultados obtidos).
5. Em relação aos testes das funções, estes ocorrem em base as instruções passadas pelo binário e, por causa disso, o usuário deve utilizar-se dos arquivos "text" e "data" que já vem inclusos na pasta compactada, podendo também modificar o código ASM ou criar um novo para ser executado pelo simulador - gerando os binários necessários. Para isso só é preciso conferir se o binário do "data" é correspondente ao seu estado inicial e, a fim de que os resultados gerados entre o MIPS e seu simulador sejam iguais, os comandos correpondam apenas as 8 funções elaboradas no código fonte - tendo

em vista que somente essas serão executadas em ambos.

III. METODOLOGIA

A seguir estão apresentadas as lógicas e formas de implementação das 8 funções requisitadas:

1. Load Word (lw)

A lógica dessa função corresponde a: lw $X, IMM($Y) => $X = MEMORY [$Y + IMM]. Ou seja, o endereço armazenado em $Y somado ao offset IMM indicará a posição da memória da qual o dado será retirado e $X corresponde ao registrador no qual este será salvo. Um exemplo disso é lw $t1, 4($t2), que salva em $t1 os bits salvos na memória com posição $t2 deslocado 4 bytes.

No código fonte isso foi implementado do seguinte modo: primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 4, cessando o procedimento caso não seja, e depois retorna o dado encontrada na posição solicitada da memória, sendo que este é dividido por 4 nessa implementação, por cada elemento do array possuir 4 bytes. Ademais, durante a execução das instruções, é especificado o registrador no qual deve ser salvo o valor retornado.

2. Load HalfWord (lh)

Similar ao Load Word, o endereço armazenado em $Y somado ao offset IMM indicará a posição da memória da qual o dado será retirado e $X corresponde ao registrador no qual este será salvo. O que difere é que o dado salvo não completará os 32 bits possíveis de armazenamento, mas metade disso (2 bytes). No caso, opera em cima de uma metade determinada do dado, completando os demais bits estendendo o sinal. Um exemplo disso é lh $t1, 2($t2), que salva em $t1 metade dos dados salvos na memória com posição $t2 deslocado 2 bytes e completa os demais bits com o sinal correspondente.

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 2, cessando o procedimento caso não seja, depois utiliza uma máscara para pegar somente os bits especificados e repete o sinal encontrado. Então, retorna o dado encontrada na posição solicitada da memória, sendo que este é dividido por 4 nessa implementação, por cada elemento do array possuir 4 bytes. Ademais, durante a execução das instruções, é especificado o registrador no qual deve ser salvo o valor retornado.

Observação: as máscaras usadas são 0x0000ffff e 0xffff0000 e aplica-se com elas a operação “AND” (&). Para o sinal, caso o primeiro bit seja 1, é usada ao final a máscara inversa com a operação “OR” (|). Por fim, caso os bytes coletados sejam do fim da palavra, deslocasse 16 bits antes de retornar o valor.

3. Load HalfWord Unsigned (lhb)

Possui quase o mesmo modo de operação do Load HalfWord, com o endereço armazenado em $Y somado ao offset IMM indicando a posição da memória da qual o dado será retirado e $X correspondendo ao registrador no qual este será salvo. Além disso, o dado salvo também não completará os 32 bits possíveis de armazenamento, mas metade disso (2 bytes). A diferença está em não estender o sinal, mas mantê-lo como positivo em todos os casos. Um exemplo disso é lhb $t1, 2($t2), que salva em $t1 metade dos dados salvos na memória com posição $t2 deslocado 2 bytes e completa os demais bits com o sinal positivo (0).

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 2, cessando o procedimento caso não seja, depois utiliza uma máscara para pegar somente os bits especificados. Então, retorna o dado

encontrada na posição solicitada da memória, sendo que este é dividido por 4 nessa implementação, por cada elemento do array possuir 4 bytes. Ademais, durante a execução das instruções, é especificado o registrador no qual deve ser salvo o valor retornado.

Observação: as máscaras usadas são 0x0000ffff e 0xffff0000 e aplica-se com elas a operação “AND” (&). Por fim, caso os bytes coletados sejam do início da palavra, deslocasse 16 bits antes de retornar o valor.

4. Load Byte (lb)

Similar ao Load Word, o endereço armazenado em $Y somado ao offset IMM indicará a posição da memória da qual o dado será retirado e $X corresponde ao registrador no qual este será salvo. O que difere é que o dado salvo não completará os 32 bits possíveis de armazenamento, mas um quarto disso (1 byte). Dessa forma, opera apenas em um byte determinado da palavra, completando os demais bits estendendo o sinal. Um exemplo disso é lb $t1, 1($t2), que salva em $t1 o byte salvo na memória com posição $t2 deslocado 1 byte e completa os demais bits com o sinal correspondente.

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 1, cessando o procedimento caso não seja, depois utiliza uma máscara para pegar somente os bits especificados e repete o sinal encontrado. Então, retorna o dado encontrada na posição solicitada da memória, sendo que este é dividido por 4 nessa implementação, por cada elemento do array possuir 4 bytes. Ademais, durante a execução das instruções, é especificado o registrador no qual deve ser salvo o valor retornado.

Observação: as máscaras usadas são 0x000000ff, 0xff000000, 0x00ff0000 e 0x0000ff00 e aplica-se com elas a operação “AND” (&). Para o sinal, caso o primeiro bit seja 1, é usada ao final a máscara complementar com a operação “OR” (|). Por fim, caso os bytes coletados não sejam do fim da palavra, deslocasse 24, 16 ou 8 bits antes de retornar o valor (respectivos as três últimas máscaras indicadas acima).

5. Loab Byte Unsigned (lbu)

Possui quase o mesmo funcionamento do Load Byte, com o endereço armazenado em $Y somado ao offset IMM indicando a posição da memória da qual o dado será retirado e $X correspondendo ao registrador no qual este será salvo. Da mesma forma, o dado salvo não completará os 32 bits possíveis de armazenamento, mas um quarto disso (1 byte), só que estenderá o sinal positivo em todos os casos. Um exemplo disso é lbh $t1, 1($t2), que salva em $t1 o byte salvo na memória com posição $t2 deslocado 1 byte e completa os demais bits com o sinal positivo (0).

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 1, cessando o procedimento caso não seja, depois utiliza uma máscara para pegar somente os bits especificados. Então, retorna o dado encontrada na posição solicitada da memória, sendo que este é dividido por 4 nessa implementação, por cada elemento do array possuir 4 bytes. Ademais, durante a execução das instruções, é especificado o registrador no qual deve ser salvo o valor retornado.

Observação: as máscaras usadas são 0x000000ff, 0xff000000, 0x00ff0000 e 0x0000ff00 e aplica-se com elas a operação “AND” (&). Por fim, caso os bytes coletados não sejam do fim da palavra, deslocasse 24, 16 ou 8 bits antes de retornar o valor (respectivos as três últimas máscaras indicadas acima).

6. Store Word (sw)

A lógica dessa função corresponde a: sw $X, IMM($Y) => MEMORY [$Y + IMM] = $X. Ou seja, $X indicará o registrador do qual o dado será retirado e o endereço armazenado em $Y somado ao offset IMM corresponde a posição da memória na qual este será salvo. Um exemplo disso é sw $t1, 4($t2), que salva na memória, com posição $t2 deslocado 4 bytes, os bits encontrados em $t1.

No código fonte isso foi implementado do seguinte modo: primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 4, cessando o procedimento caso não seja, e depois, na posição especificada (dividida por 4 nessa implementação, por cada elemento do array possuir 4 bytes), salvando o dado passado como parâmetro, sendo que este durante a execução das instruções, é especificado a partir do registrador explicitado pelo comando.

7. Store HalfWord (sh)

Similar ao Store Word, $X indicará o registrador do qual o dado será retirado e o endereço armazenado em $Y somado ao offset IMM corresponde a posição da memória na qual este será salvo. O que difere é que o dado salvo não completará os 32 bits possíveis de armazenamento, mas metade disso (2 bytes), com os demais bits correspondente ao que havia previamente na memória. Um exemplo disso é sh $t1, 2($t2), que salva na memória, com posição $t2 deslocado 2 bytes, os bits encontrados em $t1.

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 2, cessando o procedimento caso não seja, e depois, na posição especificada (dividida por 4 nessa implementação, por cada elemento do array possuir 4 bytes), salvando o dado passado como parâmetro, sendo que este durante a execução das instruções, é especificado a partir do registrador explicitado pelo comando.

Observação: as máscaras usadas são 0x0000ffff e 0xffff0000 e aplica-se com elas a operação “AND” (&). Além disso, o resultado encontrado é somado ao encontrado a partir da mesma operação “AND” aplicada na posição de memória especificada com a máscara complementar. Por fim, caso os bytes coletados sejam do fim da palavra, deslocasse 16 bits para a esquerda antes de retornar o valor.

8. Store Byte (sb)

Similar ao Store Word, $X indicará o registrador do qual o dado será retirado e o endereço armazenado em $Y somado ao offset IMM corresponde a posição da memória na qual este será salvo. O que difere é que o dado salvo não completará os 32 bits possíveis de armazenamento, mas um quarto disso (1 byte), com os demais bits correspondente ao que havia previamente na memória. Um exemplo disso é sb $t1, 1($t2), que salva na memória, com posição $t2 deslocado 1 byte, os bits encontrados em $t1.

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 1, cessando o procedimento caso não seja, e depois, na posição especificada (dividida por 4 nessa implementação, por cada elemento do array possuir 4 bytes), salvando o dado passado como parâmetro, sendo que este durante a execução das instruções, é especificado a partir do registrador explicitado pelo comando.

Observação: as máscaras usadas são 0x000000ff, 0xff000000, 0x00ff0000 e 0x0000ff00 e aplica-se com elas a operação “AND” (&). Além disso, o resultado encontrado é somado ao encontrado a partir da mesma operação “AND” aplicada na posição de memória especificada com a máscara complementar. Por fim, caso os bytes coletados não sejam do fim da

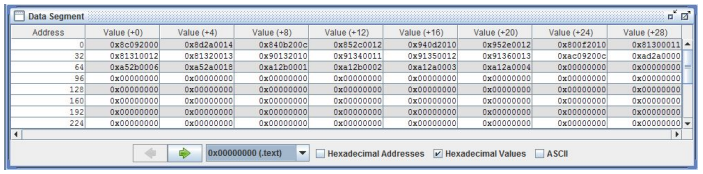
palavra, deslocasse 24, 16 ou 8 bits antes de retornar o valor (respectivos as três últimas máscaras indicadas acima).

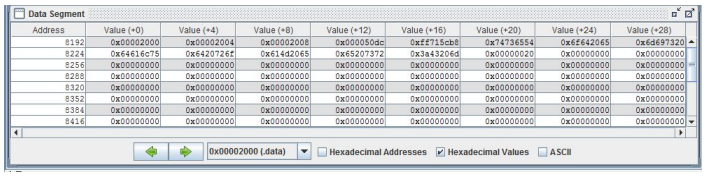
Para poder testar e analisar as funções acima mencionadas existem outras auxiliares criadas:

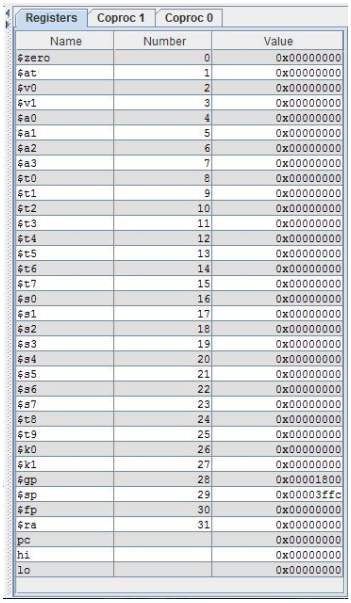
- leitura, recebe os arquivos binários e salva eles na memória;
- init, inicializa a memória e os registradores;
- instrExec, usa o array da memória para determinar as instruções a serem executadas, a partir de informações como op code, rs, rt e imm, chamando-as em seguida;
- gerarData, imprime os resultados na tela e gera um binário "dataSim", com o estado final do espaço dos dados na memória.

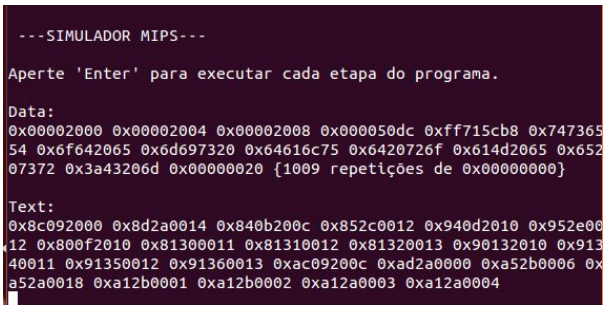
IV. RESULTADOS

Conforme os testes executados foram a partir do código ASM (MIPS-140027131.asm), as comparações de resultado são feitas a partir de imagens mostrando os dados armazenados tanto na memória quanto nos registradores em cada etapa tanto no MIPS quanto em seu simulador. Tendo isso em vista, a seguir encontram-se as análises obtidas.

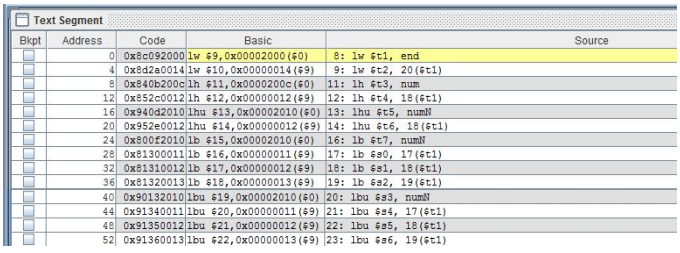


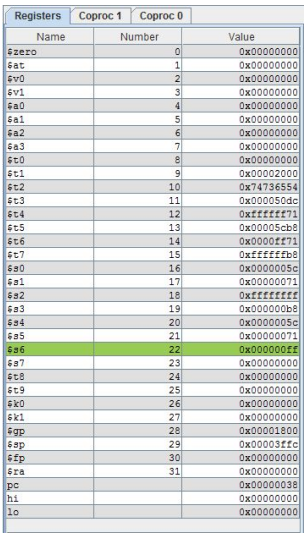






Acima estão indicados os estados iniciais do "Data Segment" (com a instruções e dados, respectivamente) e dos registradores, sendo que, em comparação ao código desenvolvido, a maior diferença está no fato de não haver a implementação da operação dos registradores como $gp e $sp, já que o foco atual era somente salvar e acessar dados dos demais (mais especificamente $t1-$t9 e $s1-$s7). Ademais, isso indica que a leitura dos binários foi executada corretamente.





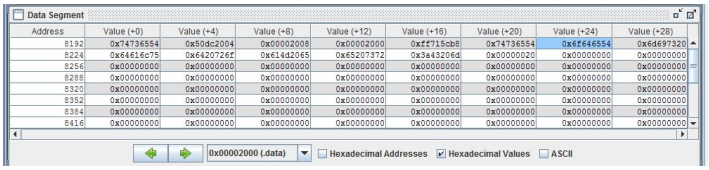

```
Execução das instruções:
lw $9,8192($0)
>> $9 = 0x00002000

lw $10,20($9)
>> $10 = 0x74736554

lh $11,8204($0)
>> $11 = 0x000050dc

lh $12,18($9)
>> $12 = 0xffffff71

lhu $13,8208($0)
>> $13 = 0x00005cb8

lhu $14,18($9)
>> $14 = 0x0000ff71

lb $15,8208($0)
>> $15 = 0xfffffbb8

lb $16,17($9)
>> $16 = 0x0000005c

lb $17,18($9)
>> $17 = 0x00000071

lb $18,19($9)
>> $18 = 0xffffffff

lbu $19,8208($0)
>> $19 = 0x000000b8

lbu $20,17($9)
>> $20 = 0x0000005c

lbu $21,18($9)
>> $21 = 0x00000071

lbu $22,19($9)
>> $22 = 0x000000ff
```

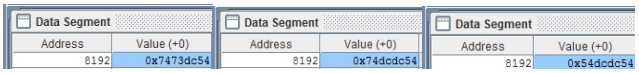
```
Registradores após a simulação:
$0 = 0x00000000 || $1 = 0x00000000 || $2 = 0x00000000 || $3 = 0
x00000000 || $4 = 0x00000000 || $5 = 0x00000000 || $6 = 0x00000
000 || $7 = 0x00000000 || $8 = 0x00000000 || $9 = 0x00002000 ||
 $10 = 0x74736554 || $11 = 0x000050dc || $12 = 0xffffff71 || $1
3 = 0x00005cb8 || $14 = 0x0000ff71 || $15 = 0xfffffbb8 || $16 =
 0x0000005c || $17 = 0x00000071 || $18 = 0xffffffff || $19 = 0x
000000b8 || $20 = 0x0000005c || $21 = 0x00000071 || $22 = 0x000
000ff || $23 = 0x00000000 || $24 = 0x00000000 || $25 = 0x000000
00 || $26 = 0x00000000 || $27 = 0x00000000 || $28 = 0x00000000
|| $29 = 0x00000000 || $30 = 0x00000000 || $31 = 0x00000000
```

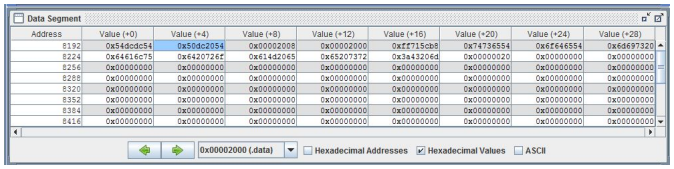
Funções de carregamento executadas, modificando os registradores de número 9 a 22 da mesma forma tanto no MIPS quando na sua implementação em C, sendo mostrado neste quando elas foram identificadas (com a impressão de cada uma e o resultado correspondente) e como se encontrava o estado final dos registradores. Novamente, os que diferem são somente os de modo de operação do MIPS ainda não implementados (28 e 29), mas todos as execuções de Load sucederam.

A cerca do que foi testado, números negativos e positivos foram utilizados em todas as funções e, quando havia mais de uma condição (máscaras e deslocamentos distintos para halfword e byte), há ao menos uma instrução que opere em cima do distintos casos. Especialmente, esses foram os testes que trouxeram maior auxílio para ajustar falhas no código, pois era a parte mais complexa a ser entendida e executada e foi onde consegui perceber certas confusões iniciais, como determinar quais eram as máscaras corretas e em quais situações o deslocamento de bits era necessário.

```
56 0xac09200c sw $9,0x0000200c($0)   25: sw $t1, num
60 0xad2a0000 sw $10,0x00000000($9)  26: sw $t2, 0($t1)
64 0xa52b0006 sh $11,0x00000006($9)  28: sh $t3, 6($t1)
68 0xa52a0018 sh $10,0x00000018($9)  29: sh $t2, 24($t1)
72 0xa12b0001 sb $11,0x00000001($9)  31: sb $t3, 1($t1)
76 0xa12b0002 sb $11,0x00000002($9)  32: sb $t3, 2($t1)
80 0xa12a0003 sb $10,0x00000003($9)  33: sb $t2, 3($t1)
84 0xa12a0004 sb $10,0x00000004($9)  34: sb $t2, 4($t1)
```








```
sw $9,8204($0)
>> mem[$0+8204] = 0x00002000

sw $10,0($9)
>> mem[$9+0] = 0x74736554

sh $11,6($9)
>> mem[$9+6] = 0x50dc2004

sh $10,24($9)
>> mem[$9+24] = 0x6f646554

sb $11,1($9)
>> mem[$9+1] = 0x7473dc54

sb $11,2($9)
>> mem[$9+2] = 0x74dcdc54

sb $10,3($9)
>> mem[$9+3] = 0x54dcdc54

sb $10,4($9)
>> mem[$9+4] = 0x50dc2054

Fim das instruções.
```

```
Data obtida pela simulação:
0x54dcdc54 0x50dc2054 0x00002008 0x00002000 0xff715cb8 0x747365
54 0x6f646554 0x6d697320 0x64616c75 0x6420726f 0x614d2065 0x652
07372 0x3a43206d 0x00000020 {2034 repetições de 0x00000000}
```

Seguindo o que foi feito anteriormente, com as funções de armazenamento também foram testados números negativos e positivos e as diferentes possíveis condições. Ademais, acima há imagens repetidas do segmento de dados na

memória, pois há mais de uma chamada em um mesmo espaço de memória, conferindo se os bits com os quais não se estão operando se mantém (ou seja, não são sobrescritos). Do mesmo modo, houve sucesso com a execução dos testes, em vista que a memória permanece igual em todos os estados do MIPS em comparação ao seu simulador.