Trabalho 2 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 algumas das funções do Mars Mips em linguagem C, além de implementar modos de execução das intruções fornecidas (*Abstract*)**

Keywords—mips, mars, simulador, assembly

I. INTRODUÇÃO

O problema apresentado consiste em desenvolver 38 instruções básicas do Assembly e simular suas execuções 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 utilizar ou desenvolver códigos ASM (números primos, fibonacci e outros de teste) 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 os arquivos simulador.c, mips.h e instrct.h;
2. Efetuar o comando a seguir no terminal: "gcc simulador.c -o Nome_Executavel";
3. Executar o arquivo criado com "./Nome_Executavel";
4. Durante a execução será necessário que o usuário escolha dentre as diferentes opções do menu, precisando somente seguir as informações explicitadas no terminal. No caso, seguirá a ordem: escolha do código a ser executado (código de teste, originalmente o de número primos - “text” e “data” - ou de fibonacci - “textF” e “dataF”); definir se quer ver os dados e instruções salvas na memória impressos; optar entre rodar o programa inteiro, executar cada etapa individualmente, impressão dos resultados obtidos (pela memória ou os registradores) e finalizar a execução do programa. Observação: ao executar Syscall o texto fica em vermelho para deixar mais evidente, em vista que não há uma caixa separada de saída e são impressos todas as instruções e seus respectivos resultados ao executar Run ou cada etapa de Step.
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 instruções 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 resumidas das 38 intruções requisitadas. Para maiores detalhes das 8 primeiras, recorrer ao relatório do trabalho 1. Observações: quando os registradores estiverem referenciados como S, T ou D significa que são decodificado a partir de rs, rt ou rd, respectivamente.

1. Load Word (lw)

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.

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, difere pelo fato do dado salvo não completar os 32 bits possíveis de armazenamento, mas metade disso (2 bytes). No caso, opera em cima de uma metade determinada do dado, estendendo o sinal para os demais bits.

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. Ademais, segue a lógica do Load Word.

3. Load HalfWord Unsigned (lhu)

Possui quase o mesmo modo de operação do Load HalfWord, com a diferença estando em não estender o sinal, mas mantê-lo como positivo em todos os casos.

4. Load Byte (lb)

Similar ao Load Word, difere pelo dado salvo não completar os 32 bits possíveis de armazenamento, mas um quarto disso (1 byte). Dessa forma, opera apenas em um byte determinado da palavra, estendendo o sinal nos demais bits.

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. Ademais, segue a lógia do Load Word.

5. Loab Byte Unsigned (lbu)

Possui quase o mesmo funcionamento do Load Byte, só que estenderá o sinal positivo em todos os casos.

6. Store Word (sw)

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.

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, difere pelo dado salvo não completar os 32 bits possíveis de armazenamento, mas metade disso (2 bytes), com os demais bits correspondendo ao que havia previamente na memória.

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 2, cessando o procedimento caso não seja. Ademais, opera da mesma forma que o Store Word.

8. Store Byte (sb)

Similar ao Store Word, difere pelo dado salvo não completar os 32 bits possíveis de armazenamento, mas um quarto disso (1 byte), com os demais bits correspondente ao que havia previamente na memória.

Na implementação, primeiramente é analisado se a soma do endereço com o offset é um múltiplo de 1, cessando o procedimento caso não seja. Ademais, segue a lógica do Store Word.

9. Load Upper Immediate (lui)

O valor imediato é deslocado 16 bits para esquerda e salvo no registrador T, com os 16 bits inferiores permanecendo como zero.

10. Branch on Equal (beq)

Caso os valores salvos nos registradores S e T sejam iguais, avança o pc com o valor correspondente ao imediato deslocado 2 bits para esquerda.

11. Branch on Not Equal (bne)

Similar ao Branch on Equal, porém analisa se os registrados são diferentes.

12. Branch on Less than or Equal to Zero (blez)

Similar ao Branch on Equal, porém analisa se o registrador especificado é menor ou igual à zero.

13. Branch on Greater than Zero (bgtz)

Similar ao Branch on Equal, porém analisa se o registrador especificado é maior do que zero.

14. Add Immediate (addi)

Faz adição entre um registrador S e um valor imediato (com sinal estendido) e salva o resultado em um registrador T.

15. Set on Less than Immediate (slti)

Se o registrador S é menor do que o imediato, T é colocado como 1, caso contrário, 0.

16. Set on Less than Immediate Unsigned (sltiu)

Mesmo processo do slti, mas com o imediato sem sinal.

17. Bitwise And Immediate (andi)

Executa um operação and entre um registrador S e um imediato e salva em T.

18. Bitwise Or Immediate (ori)

Mesma lógica de andi com or ao invés de and.

19. Bitwise Exclusive Or Immediate (xori)

Mesma lógica de andi com xor ao invés de and.

20. Jump (j)

Iguala pc ao endereço especificado pelo imediato (deslocado 2 bits para esquerda).

21. Jump And Link (jal)

Executa as funções de j, mas também salva no registrador $31 o endereço atual do pc.

22. Add Immediate Unsigned (addiu)

Instrução colocada a mais das que foram pedidas originalmente, por usar em alguns testes. Executa o mesmo que addi, porém sem overflow.

23. Add (add)

Adição entre dois registradores S e T, salvando em D o resultado.

24. Subtract (sub)

Subtração entre dois registradores S e T, salvando em D o resultado.

25. Multiply (mult)

Multiplicação entre dois registradores S e T, salvando o resultado em $lo.

26. Divide (div)

Divisão entre dois registradores S e T, salvando o quociente em $lo e o resto em $hi.

27. Bitwise And (and)

Operação and entre dois registradores S e T, salvando o resultado em D.

28. Bitwise or (or)

Operação or entre dois registradores S e T, salvando o resultado em D.

29. Bitwise Exclusive Or (xor)

Operação xor entre dois registradores S e T, salvando o resultado em D.

30. Bitwise Not Or (nor)

Operação nor entre dois registradores S e T, salvando o resultado em D.

31. Set on Less Than (slt)

Se S é menor do que T, D é igualado à 1, caso contrário, 0.

32. Jump Register (jr)

Iguala pc ao endereço condito em S.

33. Shift Left Logical (sll)

Salva em D o valor deslocado de T pela quantidade especificada pelo shamt.

34. Shift Right Logical (srl)

Salva em D o valor deslocado de T pela quantidade espeificada pelo shamt.

35. Shift Right Arithmetic (sra)

Mesma operação de srl, porém com o sinal deslocado junto.

36. Move From HI (mfhi)

O conteúdo de $hi é colocado em D.

37. Move From LO (mflo)

O conteúdo de $lo é colocado em D.

38. Syscall

Gera uma interrupção de software, sendo que três das opções foram implementadas:

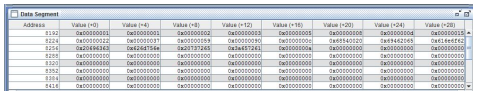
I. Registrador $2 igual a 1 faz imprimir valor inteiro salvo no registrador $4.
II. Registrador $2 igual a 4 faz imprimir caracteres salvos a partir do registrador $4 até encontrador um comando de parada.
III. Registrador $2 igual a 10 finaliza a execuação do código.

Para poder testar de fato o simulador e as instruções mencionadas acimas, existem funções criadas com o propósito de imitar o funcionamento do MIPS. Sendo que estas se utilizam de um array de memória de tamanho 4096 (cada elemento com 32 bits) e os 32 registradores mais pc, hi, lo e ri.

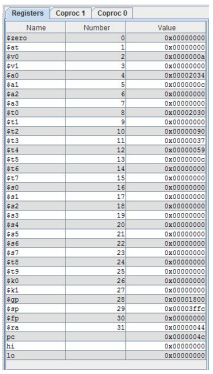
- init, inicializa a memória e os registradores;
- leitura, recebe os arquivos binários e salva eles na memória;
- fetch, lê uma instrução da memória, associa esta ao campo ri e coloca pc para indicar a próxima a ser executada;
- decode, usa o array da memória para determinar as instruções a serem executadas, a partir de informações como opcode, rs, rt, rd, shamt, funct e constantes de 16 ou 24 bits;
- execute, de acordo com os campos descodificados, executa a instrução demandada;
- step, chama as funções fetch, decode e execute caso o pc não tenha excedido o máximo e o syscall de finalização não tenha sido chamado;
- run, executa as instruções de modo similar ao step, porém não chama uma distinta a cada vez que é utilizado, mas todas de uma vez até chegar ao máximo valor para pc ou a chamada de um syscall de finalização;
- dump_reg, imprime na tela e gera um binário "regSim" com os valores dos registradores;
- dump_data, imprime na tela e gera um binário "dataSim" com os valores encontrados no espaço especificado da memória, recomendado ser o mínimo e máximo dos dados.

IV. RESULTADOS

Conforme os testes executados foram a partir dos códigos ASM fibonacci.asm e teste.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.

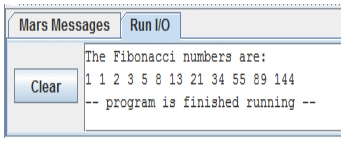


```
Data obtida pela simulação:
0x00000001 0x00000001 0x00000002 0x00000003 0x00000005 0x000000
08 0x0000000d 0x00000015 0x00000022 0x00000037 0x00000059 0x000
00090 0x0000000c 0x68540020 0x69462065 0x616e6f62 0x20696363 0x
626d756e 0x20737265 0x3a657261 0x0000000a {2027 repetições de 0
x00000000}
```



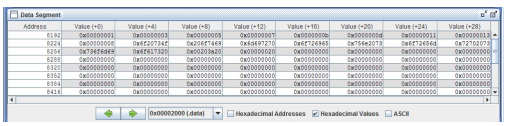
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00002034
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00002030
$t1	9	0x00000000
$t2	10	0x00000090
$t3	11	0x00000037
$t4	12	0x00000059
$t5	13	0x0000000c
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10001800
$sp	29	0x00003ffc
$fp	30	0x00000000
$ra	31	0x00000044
pc		0x0000004c
hi		0x00000000
lo		0x00000000

```
Registradores após a simulação:
$0 = 0x00000000   $1 = 0x00000000   $2 = 0x0000000a   $3 = 0x00
000000   $4 = 0x00002034   $5 = 0x0000000c   $6 = 0x00000000
$7 = 0x00000000   $8 = 0x00002030   $9 = 0x00000000   $10 = 0x0
0000090   $11 = 0x00000037   $12 = 0x00000059   $13 = 0x0000000
c   $14 = 0x00000000   $15 = 0x00000000   $16 = 0x00000000   $1
7 = 0x00000000   $18 = 0x00000000   $19 = 0x00000000   $20 = 0x
00000000   $21 = 0x00000000   $22 = 0x00000000   $23 = 0x000000
00   $24 = 0x00000000   $25 = 0x00000000   $26 = 0x00000000   $
27 = 0x00000000   $28 = 0x00000000   $29 = 0x00000000   $30 = 0
x00000000   $31 = 0x00000044   pc = 0x0000004c   hi = 0x00000000
lo = 0x00000000
```

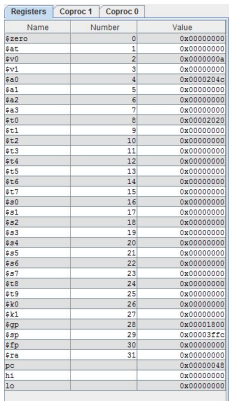


Acima estão indicados os estados finais do "Data Segment" (com a instruções e dados, respectivamente) e dos registradores ao utilizar os binários correspondentes ao fibonacci, sendo que, em comparação ao código desenvolvido, a maior diferença está no fato de não haver a implementação da operação dos registradores como $gp, $sp e $ra, já que o foco atual era a manipulação com os demais registradores, incluindo $lo, $hi e $pc. Ademais, isso indica que a execução obteve sucesso, em vista dos resultados serem iguais.

A cerca do que foi testado, diversas instruções foram utilizadas e tiveram sua execução conforme esperada. Acima não se encontram imagens com cada uma delas pela quantidade, mas isso pode ser visto ao executar o programa, assim como o resultado impresso no terminal pelo Syscall (em vermelho no terminal) - saída obtida pelo simulador foi idêntica a do MIPS.

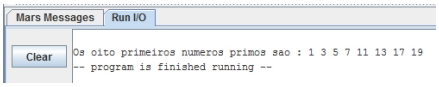


```
Data obtida pela simulação:
0x00000001 0x00000003 0x00000005 0x00000007 0x0000000b 0x000000
0d 0x00000011 0x00000013 0x00000008 0x6f20734f 0x206f7469 0x6d6
97270 0x6f726965 0x756e2073 0x6f72656d 0x72702073 0x736f6d69 0x
6f617320 0x00203a20 0x00000020 {2028 repetições de 0x00000000}
```



Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x0000204c
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00002020
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00003ffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00000048
hi		0x00000000
lo		0x00000000

```
Registradores após a simulação:
$0 = 0x00000000   $1 = 0x00000000   $2 = 0x0000000a   $3 = 0x00
000000   $4 = 0x0000204c   $5 = 0x00000000   $6 = 0x00000000
$7 = 0x00000000   $8 = 0x00002020   $9 = 0x00000000   $10 = 0x0
0000000   $11 = 0x00000000   $12 = 0x00000000   $13 = 0x0000000
0   $14 = 0x00000000   $15 = 0x00000000   $16 = 0x00000000   $1
7 = 0x00000000   $18 = 0x00000000   $19 = 0x00000000   $20 = 0x
00000000   $21 = 0x00000000   $22 = 0x00000000   $23 = 0x000000
00   $24 = 0x00000000   $25 = 0x00000000   $26 = 0x00000000   $
27 = 0x00000000   $28 = 0x00000000   $29 = 0x00000000   $30 = 0
x00000000   $31 = 0x00000000   pc = 0x00000048  hi = 0x00000000
lo = 0x00000000
```



Seguindo o que foi feito anteriormente, com o código de teste que descobre os 8 primeiros números primos, houve sucesso com a execução dos testes, em vista que a memória e os registradores são iguais ao final da execução, diferindo apenas nos registradores $gp e $ra que não foram implementados. Além disso, nos testes feitos foi perceptível que a cada instrução a operação do MIPS se igualava a do simulador, sendo que isso fica mais evidente ao rodar o programa no terminal e observar cada etapa.