Universidade de Brasília
INSTITUTO DE CIÊNCIAS EXATAS – IE
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – CIC

Disciplina: CIC 116394 – Organização e Arquitetura de Computadores – Turma C
Semestre: 1/2017
Professor: Marcelo Grandi Mandelli

TRABALHO 2 – SIMULADOR MIPS

OBJETIVOS

- Compreender o funcionamento do processador MIPS;
- Implementar um simulador de uma determinada ISA (Instruction Set Architecture);

ESPECIFICAÇÃO DO TRABALHO

Este trabalho consiste na implementação de um simulador da arquitetura MIPS em linguagem de alto nível (obrigatoriamente C ou C++). O simulador deve implementar as funções de busca da instrução (*fetch()*), decodificação da instrução (*decode()*) e execução da instrução (*execute()*). O programa binário a ser executado deve ser gerado a partir do montador MARS, juntamento com os respectivos dados. O simulador deve ler arquivos binários contendo o segmento de código e o segmento de dados para sua memória e executá-lo.

Arquivos de entrada do simulador
O simulador deverá receber como entrada dois arquivos, em formato **.bin**, provenientes de um programa assembly: um contendo o segmento de código, contendo as instruções desse programa; e outro contendo o segmento de dados.

O simulador MARS deverá ser utilizado, necessariamente, para a obtenção do segmento de código e do segmento de dados do programa assembly. Para ilustrar o procedimento de obtenção desses segmentos, considere o programa a seguir.

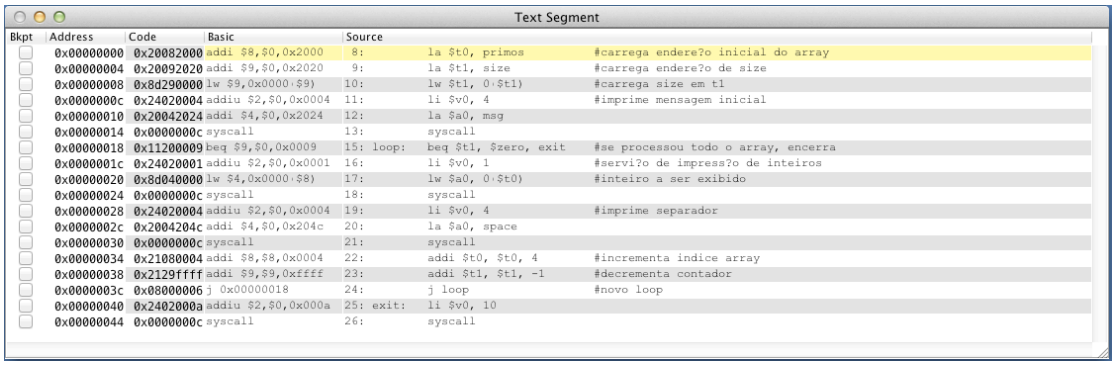
```
.data
primos:     .word 1, 3, 5, 7, 11, 13, 17, 19
size:       .word 8
msg:        .asciiz "Os oito primeiros numeros primos sao : "
space:      .ascii " "

.text
      la $t0, primos          #carrega endereço inicial do array
      la $t1, size            #carrega endereço de size
      lw $t1, 0($t1)          #carrega size em t1
      li $v0, 4               #imprime mensagem inicial
      la $a0, msg
      syscall

loop: beq $t1, $zero, exit    #se processou todo o array, encerra
      li $v0, 1               #serviço de impressão de inteiros
      lw $a0, 0($t0)          #inteiro a ser exibido
```

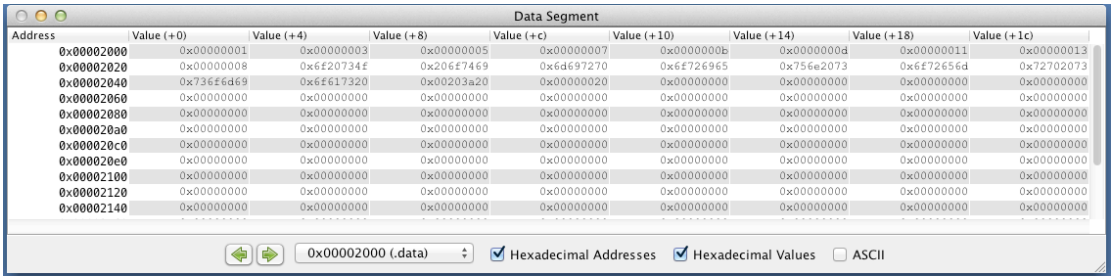
```
        syscall
        li $v0, 4               #imprime separador
        la $a0, space
        syscall
        addi $t0, $t0, 4        #incrementa indice array
        addi $t1, $t1, -1             #decrementa contador
        j loop                  #novo loop
exit:   li $v0, 10
        syscall
```

Esse programa deve ser montado utilizando o simulador MARS. Para isso, primeiro deve-se configurar o MARS através da opção:

Settings→Memory Configuration, opção Compact, Text at Address 0

Ao montar o programa (opção *Run→Assemble* ou F3), o MARS exibe na aba *Execute* os segmentos de texto (*Text Segment)* e dados (*Data Segment)*, apresentados abaixo.





O segmento de código (*Text Segment*) desse programa começa no endereço 0x00000000 de memória e se encerra no endereço 0x00000044, que contém a instrução *syscall.* O segmento de dados (*Data Segment*) começa na posição 0x00002000 e termina na posição 0x000204c.

A obtenção do segmento de código e do segmento de dados desse programa é efetuada através da opção:

File -> Dump Memory...

Para o salvamento do segmento de código selecione:

Memory segment: *.text (0x00000000 - 0x00000044)*

Dump Format: *binary*

Clique em *Dump To File...* e salve o arquivo com extensão **.bin**. Sugestão, salve como <programa>*_text.bin*.

Para o salvamento do segmento de dados selecione:

Memory segment: *.data (0x00002000 - 0x0000204c)*

Dump Format: *binary*

Clique em *Dump To File...* e salve o arquivo com extensão **.bin.** Sugestão, salve como <programa>*_data.bin*.

Memória do simulador

A memória do simulador deve ser modelada como um arranjo de inteiros de 32 bits:

```
#define MEM_SIZE 4096
int32_t mem[MEM_SIZE];
```

Ou seja, a memória é um arranjo de 4KWords, ou 16KBytes.

A memória será endereçada byte a byte.

O segmento de código e o segmento de dados contidos nos arquivos **.bin** de entrada devem ser lidos para a memória do simulador. O segmento de código deve começar na posição *0x00000000* da memória, enquanto o segmento de dados deve começar na posição 0*x00002000*.

Registradores

O simulador deve conter os 32 registradores do MIPS, mais os registradores *pc*, *ri*, *hi* e *lo*. Todos registradores serão declarados como variáveis globais. Os 32 registradores do MIPS, mais os registradores *hi* e *lo* serão do tipo i*nt* (*int32_t*). Já os registradores *pc* e *ri* podem ser do tipo *unsigned int (uint32_t)*, visto que não armazenam dados, apenas instruções.

Função void fetch()

Essa função lê uma instrução da memória e coloca-a em *ri*, atualizando o *pc* para apontar para a próxima instrução (soma 4).

Função void decode()

Essa função deve extrair todos os campos da instrução:

- opcode: código da operação
- rs: índice do primeiro registrador fonte
- rt: índice do segundo registrador fonte
- rd: índice do registrador destino, que recebe o resultado da operação
- shamt: quantidade de deslocamento em instruções *shift* e *rotate*
- funct: código auxiliar para determinar a instrução a ser executada
- k16: constante de 16 bits, valor imediato em instruções tipo I
- k26: constante de 26 bits, para instruções tipo J

Os campos da instrução devem ser definidos como variáveis globais.

Função void execute()

Essa função executa a instrução que foi lida pela função fetch() e decodificada por decode().

Função void step()

Essa executa uma instrução do MIPS:

step() => fetch(), decode(), execute()

Função void run()
Essa função executa o programa até encontrar uma chamada de sistema para encerramento, ou até o *pc* ultrapassar o limite do segmento de código (2k *words*).

Função void dump_mem(int start, int end, char format)
Essa função imprime o conteúdo da memória a partir do endereço *start* até o endereço *end*. O formato pode ser em hexa ('h') ou decimal ('d'). A memória deve ser impressa em palavras de 32 bits, de acordo com o formato abaixo, onde o lado esquerdo apresenta o endereço da memória e o lado direito mostra o valor contido nesse endereço.

```
MEMORY
0x00000000 = 0x00004000
0x00000004 = 0x0000ef00
0x00000008 = 0x0000ffff
0x0000000c = 0xf0caf0fa
```

Função void dump_reg(char format)
Essa função imprime o conteúdo dos registradores do MIPS, incluindo o banco de registradores e os registradores *pc*, *hi* e *lo*. O formato pode ser em hexa ('h') ou decimal ('d'). Os registradores devem ser impressos como mostrado abaixo:

```
REGISTERS
$zero = 0x00000000
$at = 0x00000000
$v0 = 0x00000000
$v1 = 0x00000000
$a0 = 0x00000000
$a1 = 0x00022000
$a2 = 0x00000000
$a3 = 0x00000000
$t0 = 0x00000000
$t1 = 0x00000000
$t2 = 0x00003000
$t3 = 0x00000000
$t4 = 0x00000000
$t5 = 0x00046000
$t6 = 0x00000000
$t7 = 0x000a0000
$s0 = 0x00000000
$s1 = 0x00000e00
$s2 = 0x00000000
$s3 = 0x00000000
$s4 = 0x00000000
$s5 = 0x00000000
$s6 = 0x00000000
$s7 = 0x00000000
$t8 = 0x00000000
$t9 = 0x00000000
$k0 = 0x00000000
$k1 = 0x00000000
$gp = 0x00000000
$sp = 0x00000000
$fp = 0x00000000
$ra = 0x00000000

PC = 0x00001004
HI = 0x00000000
LO = 0x00000000
```

Instruções a serem implementadas:

Tipo-R geral		Tipo-I Lógico-Aritméticas	Desvios, Jumps e Syscall	Load e Stores
ADD	**SLT**	**ADDI**	**J**	**LB**
SUB	**SLL**	**ANDI**	**JAL**	**LBU**
DIV	**SRL**	**ORI**	**JR**	**LH**
MULT	**SRA**	**XORI**	**BEQ**	**LHU**
AND	**MFHI**	**SLTI**	**BGTZ**	**LW**
OR	**MFLO**	**SLTIU**	**BLEZ**	**SB**
NOR	**XOR**	**ADDIU**	**BNE**	**SH**
				SW
				LUI

Syscall: implementar as chamadas para (ver *help* do MARS)

- imprimir inteiro
- imprimir string
- encerrar programa

Execução do simulador

O simulador deve ser executado via linha de comandos. Dessa forma, a execução do simulador será feita através de :

```
./<executável_do_simulador> <parâmetros>
```

O simulador contará com 3 parâmetros:

- parâmetro 1: caminho do arquivo contendo o segmento de código do programa em assembly em formato **.bin**. Exemplo: /home/mips_simulator/tests/primos_text.bin
- parâmetro 2: caminho do arquivo contendo o segmento de dados do programa em assembly em formato .bin. Exemplo: /home/mips_simulator/tests/primos_data.bin
- parâmetro 3: esse parâmetro aceita dois valores:
  - p : se o parâmetro 3 for igual a "p", o simulador abrirá uma interface com o usuário. A interface com usuário deverá conter um cardápio de funções e servir como um guia de ajuda de como utilizar cada função. O usuário poderá utilizar as seguintes funções: step(), run(), dump_mem(int start, int end, char format), dump_reg(char format) e *exit()* (para encerrar o simulador). Caso o usuário selecione uma função com parâmetros, estes devem ser pedidos ao usuário.
  - f : se o parâmetro 3 for igual a "f", o simulador deverá executar todo o programa assembly (função *run()*) e gerar dois arquivos: reg.txt, contendo o valor de todos os registradores em formato hexadecimal (função *dump_reg(h)*); e mem.txt, contendo o valor de toda a memória em formato hexadecimal (função *dump_mem(0, 4095, h)*). Os arquivos reg.txt e mem.txt serão gerados para a pasta corrente de chamada do simulador. Depois de gerar esses arquivos, o simulador encerra a execução.

Por exemplo, ao executar o comando:

```
./<executável_do_simulador> primos_text.bin primos_data.bin f
```

o simulador executará todo o programa assembly composto pelo segmento de código contido no arquivo "primos_text.bin" e segmento de dados contido no arquivo "primos_data.bin. Depois, gerará os arquivos reg.txt e mem.txt na pasta corrente ao qual foi chamado o simulador. E por fim encerrará a execução.

Agora, se o usuário executar o comando:

```
./<executável_do_simulador> primos_text.bin primos_data.bin p
```

o simulador mostrará uma inteface de usuário, como demonstrado abaixo:

```
• Defina o número função desejada:
   1. step
      <Descrição da função / Modo de usar>
   2. run
      <Descrição da função / Modo de usar>
   3. dump_mem
      <Descrição da função / Modo de usar>
   4. dump_reg
      <Descrição da função / Modo de usar>
   5. exit
```

Verificação do Simulador

1. Construa um programa para testar cada uma das instruções acima. Não precisa fazer nenhuma função específica, apenas verificar se cada instrução executa ok.

2. Para verificar o funcionamento do simulador, monte os programas exemplos fornecidos junto ao trabalho (números primos e fibonacci). Para cada programa exemplo:
   a. Salve o código e dados nos arquivos indicados.
   b. Leia os arquivos para a memória.
   c. Execute o programa.
   d. Utilize as funções dump_mem e dump_reg para mostrar os conteúdos
   e. Verifique o funcionamento de cada instrução do simulador

GRUPOS

O trabalho deverá ser realizado em grupos de 2 ou 3 alunos.

ENTREGA

Deverá ser entregue:

- um **relatório**, contendo
  - explicação do código implementado, incluindo descrição das funções implementadas. Além disso explicar como compilar e executar o código implementado (compilador / sistema operacional)

    - testes e resultados, demonstrando o funcionamento do simulador implementado
- o **código fonte do simulador**, em linguagem C ou C++

Entregar um arquivo compactado em formato .zip no moodle (aprender.unb.br) até às 23h55 do dia 08/05/2017

O nome do arquivo deve conter as matrículas dos integrantes do grupo:
matriculaaluno1_matriculaaluno2_matriculaaluno3.zip

Será descontado 2 pontos por dia de atraso na entrega do trabalho.

CRITÉRIOS DE AVALIAÇÃO

Funcionamento do simulador – 80% da nota

I - Testes Simples de Instruções – 20% da nota
- I.I - Instruções Tipo-R geral – 2% da nota
- I.II - Instruções Tipo-I Lógico-Aritméticas – 3% da nota
- I.III - Instruções de Loads e Stores – 5% da nota
- I.IV - Instruções de Desvios, Jumps e Syscall – 10% da nota

II - Testes com Programas prontos (Fibonacci, primos e outros – 45% da nota
- II.I - Teste dos primos – 10% da nota
- II.II - Teste de Fibonacci – 20% da nota
- II.III - Outros – 15% da nota

III - Testes de Dumps (memória e registradores) – 10% da nota
- III.I - Memória – 5% da nota
- III.II - Registradores – 5% da nota

IV – Execução do simulador (interface com usuário/geração arquivos) – 5% da nota

Relatório – 20% da nota

I - Explicação do código – 10% da nota
II - Testes e resultados – 10% da nota

Esta especificação pode ser atualizada para se efetuar correções de texto ou alterações para se deixar mais claro o que está sendo pedido.

Caso a especificação sofrer uma atualização, os alunos serão informados via moodle (aprender.unb.br).

Última atualização: 18/04/2017 às 13:00