

UNIVERSIDADE DE BRASÍLIA  
INSTITUTO DE CIÊNCIAS EXATAS  
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

**116394 ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES**

Trabalho II: Simulador MIPS

## **OBJETIVO**

Este trabalho consiste na implementação de um simulador da arquitetura MIPS em linguagem de alto nível (C/C++/Java). 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, juntamente 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.

## **DESCRIÇÃO**

### ***Geração dos arquivos***

As instruções e dados de um programa MIPS para este trabalho devem vir necessariamente de arquivos montados pelo MARS. Para ilustrar o procedimento, considere o exemplo 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
```

### ***Montagem do programa***

Antes de montar o programa deve-se configurar o MARS através da opção:

*Settings->Memory Configuration*, opção Compact, Text at Address 0

Ao montar o programa (F3), o MARS exibe na aba “Execute” os segmentos *Text* e *Data*, apresentados abaixo. O segmento de código (*Text*) deste programa começa no endereço 0x00000000 de memória e se encerra no endereço 0x00000044, que contém a instrução *syscall*. O segmento de dados começa na posição 0x00002000 e termina na posição 0x000204c. Verifique a ordem dos caracteres da mensagem *msg* no segmento de dados usando a opção ASCII de visualização.

O armazenamento destas informações em arquivo é obtido com a opção:

*File -> Dump Memory...*

As opções de salvamento devem ser:

Código:

.text (0x00000000 - 0x00000044) - que é o valor *default* para este exemplo

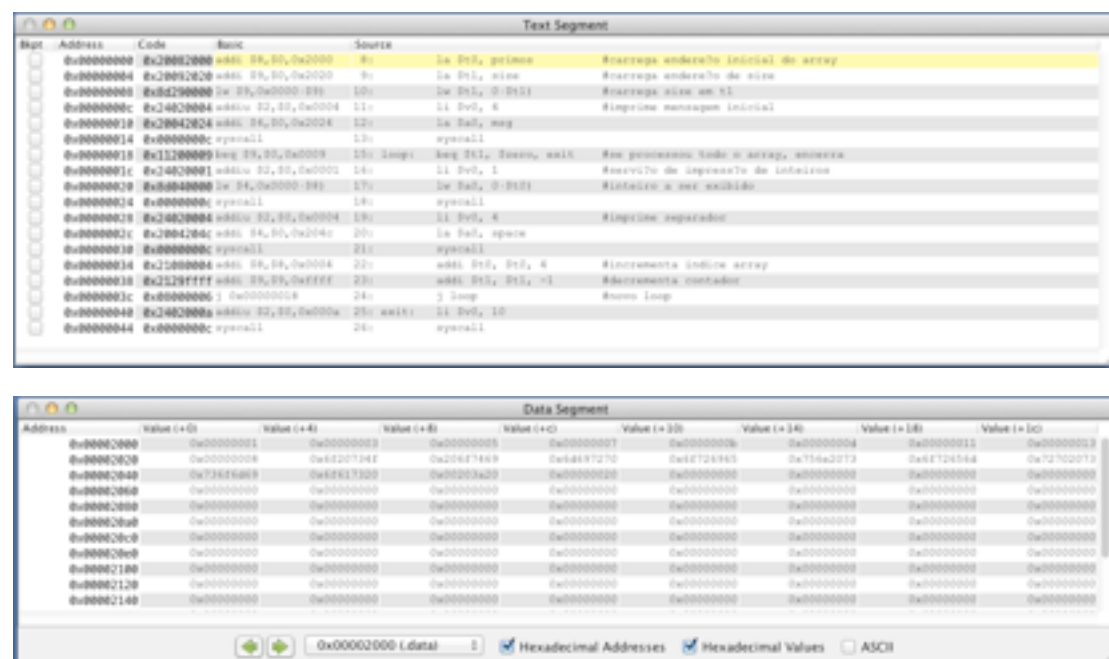
*Dump Format: binary*

Dados:

.data (0x0000200 - 0x0000204c) - que contém os dados de interesse para o exemplo.

*Dump Format: binary*

Gere os arquivos com nomes text.bin e data.bin.



## Leitura do código e dos dados

O código e os dados contidos nos arquivos devem ser lidos para a memória do simulador.

A memória deve ser modelada como um arranjo de inteiros:

```
#define MEM_SIZE 8192
int mem[MEM_SIZE];
```

Ou seja, a memória é um arranjo de 8KWords, ou 32KBytes.

### **Acesso à Memória**

Desenvolver as funções:

```
int read_int(unsigned int end)          # lê um inteiro alinhado - endereços múltiplos de 4
int read_byte(unsigned int end)        # lê um byte - retorna inteiro com sinal
int read_ubyte(unsigned int end)       # lê um byte - retorna inteiro sem sinal
void write_int(unsigned int end, int dado)    # escreve um inteiro alinhado na memória
void write_byte(unsigned int end, char byte)  # escreve um byte na sua posição na memória
```

Os endereços são todos de *byte*. A operação de leitura de *byte* retorna um inteiro com o *byte* lido na posição menos significativa. A escrita de um *byte* deve colocá-lo na posição correta dentro da palavra de memória.

### **Registradores**

Os registradores *pc* e *ri*, e também os campos da instrução (*opcode*, *rs*, *rt*, *rd*, *shamt*, *funct*) devem ser definidos como variáveis globais. *pc* e *ri* podem ser do tipo *unsigned int*, visto que não armazenam dados, apenas instruções.

### **Função fetch()**

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

### **Função decode()**

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
- shamnt: 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

### **Função execute()**

A função `void execute()` executa a instrução que foi lida pela função `fetch()` e decodificada por `decode()`.

### Função *step()*

A função *step()* executa uma instrução do MIPS:

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

### Função *run()*

A função *run()* executa o programa até encontrar uma chamada de sistema para encerramento, ou até o pc ultrapassar o limite do segmento de código (2kwords).

### Função *dump\_mem(int start, int end, char format)*

Imprime o conteúdo da memória a partir do endereço *start* até o endereço *end*. O formato pode ser em hexa ('h') ou decimal ('d').

### Função *dump\_reg(char format)*

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').

### Instruções a serem implementadas:

```
enum OPCODES { // lembrem que so sao considerados os 6 primeiros bits dessas constantes
```

```
    EXT=0x00,    LW=0x23,    LB=0x20,    LBU=0x24,
    LH=0x21,    LHU=0x25,    LUI=0x0F,    SW=0x2B,
    SB=0x28,    SH=0x29,    BEQ=0x04,    BNE=0x05,
    BLEZ=0x06,    BGTZ=0x07,    ADDI=0x08,    SLTI=0x0A,
    SLTIU=0x0B,    ANDI=0x0C,    ORI=0x0D,    XORI=0x0E,
    J=0x02,      JAL=0x03
};
```

```
enum FUNCT {
    ADD=0x20,    SUB=0x22,    MUL=0x18,    DIV=0x1A,    AND=0x24,
    OR=0x25,     XOR=0x26,    NOR=0x27,    SLT=0x2A,     JR=0x08,
    SLL=0x00,    SRL=0x02,    SRA=0x03,    SYSCALL=0x0c
};
```

Syscall: implementar as chamadas para

- \* imprimir inteiro
- \* imprimir string
- \* encerrar programa

### Verificação do Simulador

1. Construa um programa para testar cada uma das instruções acima. Não precisa fazer nenhuma função específica, apenas verifica se cada instruções executa ok.
2. Monte o programa na primeira página e a série fibonacci fornecida junto com o MARS.

para cada exemplo:

3. Salve o código e dados nos arquivos indicados.

4. Leia os arquivos para a memória.
5. Execute o programa.
6. Utilize as funções `dump_mem` e `dump_reg` para mostrar os conteúdos

## **Entrega**

Entregar:

- relatório da implementação:
  - descrição do problema
  - descrição sucinta das funções implementadas
  - testes e resultados
- o código fonte do simulador, com a indicação da plataforma utilizada:
  - qual compilador empregado
  - sistema operacional
  - IDE (Eclipse, XCode, etc)

Entregar no Moodle em um arquivo compactado.

Prazo de entrega: 28/04/14