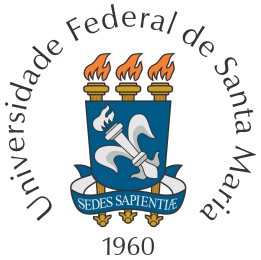


UNIVERSIDADE FEDERAL DE SANTA MARIA
DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO
ELC1011 - ORGANIZAÇÃO DE COMPUTADORES

PROCESSADOR MIPS: ESTUDO E ANÁLISE ATRAVÉS DO DESENVOLVIMENTO DE SIMULADOR

AUTORES: MARIA RITA PIEKAS E NATHÁLIA DE ALMEIDA ZÓFOLI
DOCENTE ORIENTADOR: GIOVANI BARATTO

Santa Maria, 2024

SUMÁRIO

OBJETIVOS

O presente trabalho possui os seguintes objetivos, categorizados em gerais e específicos:

2.1. OBJETIVOS GERAIS

- Simular o ciclo de processamento de informações realizado pelo processador MIPS;
- Compreender como um processador manipula os dados através da simulação de operações;
- Desenvolver habilidades lógicas e de programação;

2.2 OBJETIVOS ESPECÍFICOS

- Desenvolver uma aplicação em linguagem Assembly, utilizando o software Mars - versão 4.5, que simule a busca, decodificação e execução de instruções realizadas pelo processador;
- Observar o comportamento do algoritmo durante a leitura de um arquivo binário contendo as instruções para a execução da operação fatorial do número 5.

INTRODUÇÃO

O seguinte projeto teve embasamento teórico e metodológico na disciplina corrente de Organização de Computadores através dos arquivos-fonte do professor responsável e no livro texto da disciplina: PATTERSON, David A.; HENNESSY, John L. Organização e projeto de computadores : a interface hardware/software.

Na informática, computadores são dispositivos capazes de tratar algoritmos para realizar tarefas, são constituídos basicamente de Hardware e Software. Software faz referência aos algoritmos, códigos e instruções executados pelo computador, já o Hardware trata da parte física do computador é dividido em:

-Dispositivos de entrada e saída: os famosos dispositivos periféricos são os componentes que nos permitem interagir com a máquina, são as telas, teclados, mouses, etc;

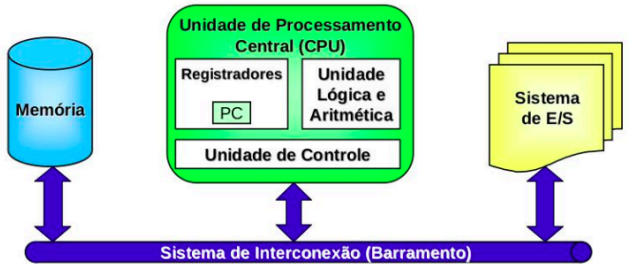
-Memória: a memória principal do computador serve para armazenar as informações de usuários e configurações básicas para o funcionamento do computador, como o sistema operacional. Atualmente o mercado dispõe de duas tecnologias principais: a dos SSDs (Solid State Drive) onde as informações são gravadas em chips de memória flash e dos HDs (Hard Disk) que nada mais é que em disco de metal que gira em alta velocidade;

-Unidade Central de Processamento: mais conhecido como processador, ele é o “cérebro” do computador, responsável por gerenciar todo o funcionamento da máquina.

O presente trabalho tem como objetivo o estudo de processadores, neste caso mais especificamente o processador MIPS, baseados na arquitetura de Von Neumann, os processadores MIPS são constituídos por:

- Subsistema de dados: no subsistema de dados temos um banco de registradores, que são pequenas unidades de memória temporária que armazenam n bits que estão sendo operados no momento, Unidade Lógica e Aritmética (ULA) que como o nome sugere, realiza as operações matemáticas e lógicas básicas, há também registradores especiais que são destinados a tarefas específicas;
- Subsistema de controle: responsável por gerenciar o ciclo busca-decodificação-execução dentro do processador;
- Há também a Unidade de Gerenciamento de Memória, Memória CACHE e Pipeline, que não são o foco deste trabalho.

A Arquitetura de Von Neumann



A interação hardware - software produz a computação, e se dá da seguinte maneira:

- O desenvolvedor de software cria um programa em linguagem de alto nível (que possui maior proximidade com a linguagem humana);
- Este programa passa por um compilador que faz uma análise léxica e semântica e traduz o código em alto nível para a linguagem de baixo nível, a linguagem de montagem ou Assembly que é mais próxima da linguagem de máquina mais ainda entendível por humanos;
- Então ele passa por um montador ou assembler que converte o programa em linguagem de montagem para a linguagem de máquina, gerando um Objeto (módulo em linguagem de máquina);
- Após, ele passa por Linker que combina o Objeto com rotinas de bibliotecas e assim cria um Executável: um programa em linguagem de máquina;
- O Executável é carregado para a memória do computador por um Loader e enfim está pronto para a execução.

A linguagem de máquina é composta apenas por 0s e 1s sequenciais, eles compõem o Executável que da memória é carregado para o processador que pega cada instrução do arquivo binário (o executável, escrito por 0s e 1s), separa em campos, decodifica estes campos e os executa.

No processador MIPS cada instrução é formada por 32 bits (algarismos em linguagem binária), este formato é adotado por razão da arquitetura do processador, que é do tipo RISC (Reduced Instruction Set Computing) que visa a rápida execução de tarefas simples, e é dividida em campos de acordo com o tipo de instrução. São três tipos de instruções:

- Instruções tipo R (register): São instruções que operam entre os registradores e são divididas nos campos OPCODE que indicam o tipo de instrução, no caso de instruções tipo R, é zero; RS que representa o registrador que guardará o resultado da operação, RT e RD que são os registradores que possuem os valores a serem operados; campo SHAMT que armazena o valores de deslocamento e o campo FUNCT que indica qual das funções do tipo R é;

- Instruções do tipo I (immediate): São instruções que operam valores de registradores e um valor, também chamado de imediato e são divididas nos campos OPCODE que neste caso varia para cada instrução, RS que representa o registrador de destino, RT que representa o registrador a ser operado e um campo que representa o valor da operação;
- Instruções tipo J (jump): São instruções que “saltam” de um ponto do código para outro, são divididas nos campos OPCODE que define qual operação está sendo tratada e o endereço destino do “salto”.

Instrução	Campos					
Formato	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Tipo R	op	rs	rt	rd	shamt	funct
Tipo I	op	rs	rt	Endereço/Imediato		
Tipo J	op	Endereço de Destino				

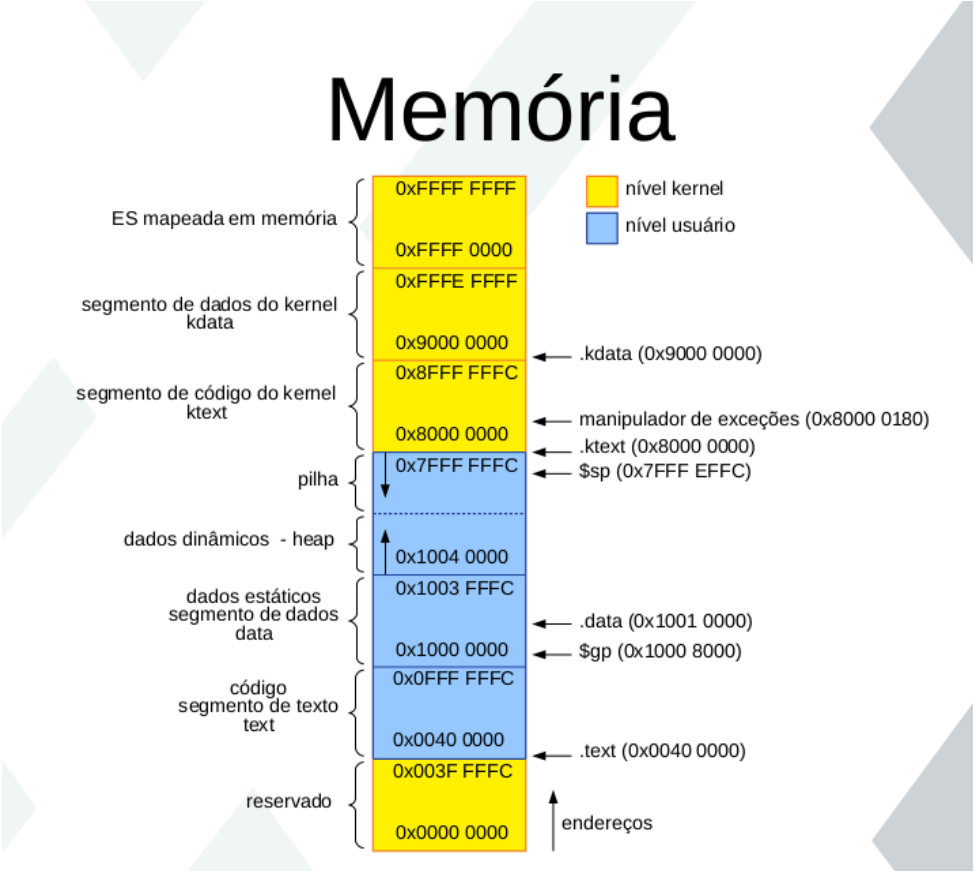
Diagrama das instruções no MIPS

O processador MIPS têm seus registradores divididos de acordo com o uso, que está exemplificado na imagem a seguir:

Número	Mnemônico	Uso Convencional
$0	$zero	Sempre 0
$1	$at	Temporário para o *assembly* (reservado)
$2, $3	$v0, $v1	Valor retornado por uma sub-rotina
$4-$7	$a0-$a3	Argumentos para uma sub-rotina
$8-$15	$t0-$t7	Temporário (não preservados na chamada à uma função)
$16-$23	$s0-$s7	Registradores salvos (preservados na chamada à uma função)
$24, $25	$t8, $t9	Temporários
$26, $27	$k0, $k1	Kernel (reservado para o sistema operacional)
$28	$gp	Ponteiro global
$29	$sp	Ponteiro para a pilha (*stack pointer*)
$30	$fp	Ponteiro para quadro *(frame pointer*)
$31	$ra	Endereço de retorno de procedimento

Diagrama dos registradores do MIPS

Outro detalhe importante é que tanto os registradores, quanto a memória operam com WORDS: “palavras” de 4 bytes, 32 bits. Na memória, cada palavra está em um endereço, assim sendo, ela é subdividida em diversos segmentos, há segmento específico para os dados, segmento específico para textos e programas e segmentos dedicados ao processador, os quais estão descritos na imagem seguinte:



Representação da distribuição da memória no MIPS

O computador, após este ciclo de decodificação, executa as instruções, opera os zeros e uns através dos circuitos, os circuitos são conjuntos de componentes elétricos que permitem ou não a passagem de energia, os zeros representam nível baixo de energia e os uns representam nível alto. Operando a passagem ou não de corrente, rapidamente e em grande escala, podemos armazenar, criar e manipular dados de acordo com nossos propósitos.

DESENVOLVIMENTO

METODOLOGIA

Uma maneira eficaz de se estudar uma arquitetura de hardware sem utilizar componentes eletrônicos é através de simuladores. Um simulador, segundo o Dicionário Priberam de Língua Portuguesa, é *"Dispositivo capaz de reproduzir o comportamento de um aparelho de que se deseja quer estudar o funcionamento, quer ensinar a utilização, ou de um corpo de que se pretende seguir a evolução."* Neste caso, vamos estudar o processador MIPS por meio da construção de um

simulador em software, utilizando uma IDE (interface de ambiente de desenvolvimento) que simula o processador MIPS e utilizando linguagem de máquina.

A IDE escolhia é o Software Mars - MIPS Assembler and Runtime Simulator (ver referências), desenvolvido pela Universidade Estadual do Missouri na linguagem de programação Java com base no MIPS 32 bits monociclo (opera a um ciclo de relógio, um sinal), em sua versão 4.5, ele possui 155 instruções e 370 pseudo-instruções (concatenamento de instruções básicas visando simplificar a programação), opções de debugger e interface gráfica simples, tornando-o uma ferramenta versátil.

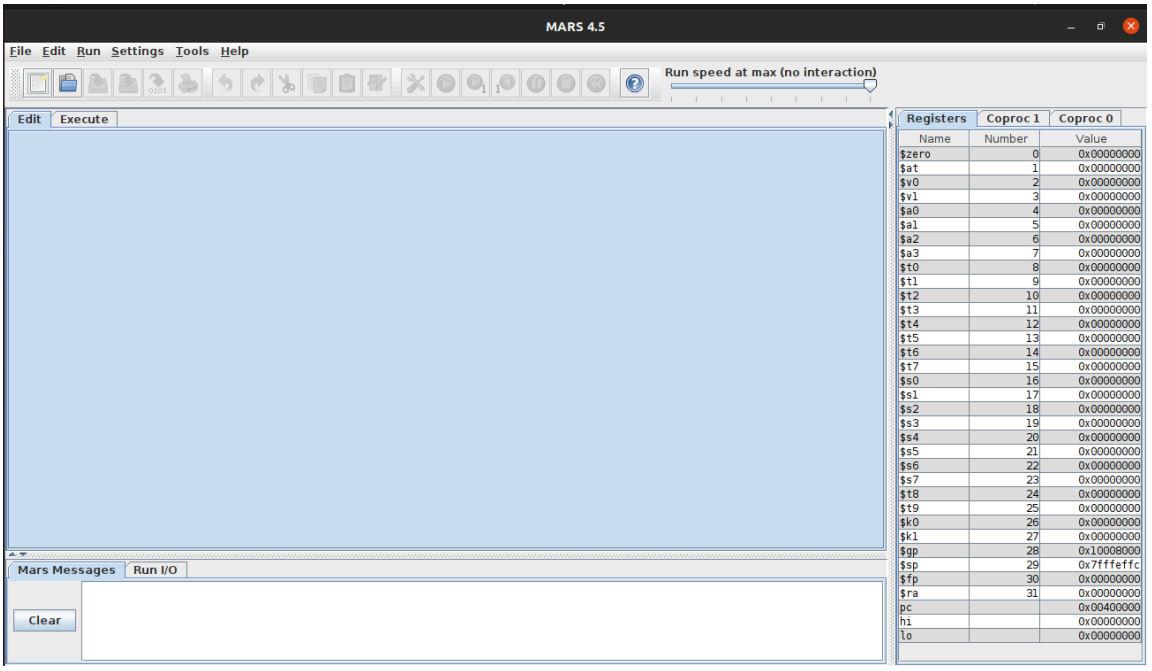


Imagem da interface da IDE Mars

O simulador desenvolvido foi programado com o propósito de receber um arquivo da memória, copiá-lo para a memória de texto, decodificá-lo e executá-lo a fim de calcular e imprimir no terminal a mensagem: "*O fatorial de 5 é 120*".

EXPERIMENTAÇÃO

O programa foi estruturado em procedimentos, trechos de códigos que desempenham uma tarefa específica, a fim otimizar e facilitar a leitura do código.

A criação do simulador foi dividida em diversas partes, partes estas compostas de vários passos: esquematização do projeto, estruturação no formato Top-Down, que consiste na modularização do código inteiro e após especificação e fase de testes e correções, os quais serão abordados detalhadamente na sequência.

Esquematização e estruturação:

Para simular o processador real, é necessário simular primeiramente o hardware, para isso primeiramente foram simulados os registradores no formato de array com 128 bits, reservando um espaço na memória do simulador real, foram simulados também os registradores especiais PC que contém o endereço da próxima instrução a ser executada, IR que contém o endereço da função que está sendo executada, Hi e Lo que são reservados para retorno de multiplicações e divisões. Em seguida foram simuladas as memórias de texto, dados e pilha e seus respectivos endereços, além das variáveis globais para o buffer de leitura, descritor de arquivo e endereços de arquivo. Todas essas informações fazem parte do segmento de dados do processador.

```
.data
   PC:              .word 0
   IR:              .word 0
   HI:          .word 0
   LO:          .word 0
   regs:            .space 128    ## Registradores do simulador
   buffer_leitura:      .space 4          ## Buffer para leitura do
arquivo de entrada. Armazena 4 bytes por vez
   memoria_text:        .space 1024
   memoria_data:        .space 1024
   memoria_pilha:       .space 1024
   endereco_texto:      .word 0x1001008c
   endereco_data:       .word 0x10010000
   endereco_pilha:      .word 0x7FFFEFFC
   local_arquivo:       .asciiz "trabalho_01-2024_1.bin"
   local_arquivo_data:      .asciiz "trabalh0_01-2024_1.dat"
   descritor_arquivo:   .word 0
   count: .word 0       # Variável para contar as iterações do loop
```

Com o hardware simulado, o próximo passo é realizar a leitura do arquivo e transportá-lo para a memória e prepará-lo para o processo de busca e decodificação .[1]

Para leitura de arquivos o computador faz uma chamada ao sistema operacional que solicita a leitura do arquivo, o sistema operacional contém uma lista de descritores dos arquivos e envia a solicitação para os gerenciadores de disco, buscando ler dados de um local específico do disco, o S.O então guarda os dados na memória e retorna o sistema ao chamador, destes passos, poucos são executados pelo processador e uma maneira de executar é dividindo o procedimento em abertura do arquivo, leitura do arquivo e fechamento do arquivo.

A leitura consiste em carregar o endereço e nome do arquivo, chamar o sistema operacional, armazenar o descritor do arquivo e testar a abertura verificando se o número de bytes lidos é compatível com o número de bytes do arquivo.

Para abrir o arquivo, foi desenvolvido um procedimento que que salva o nome, endereço e locais do arquivo, chama o sistema operacional para leitura, guarda o descritor do arquivo e verifica a leitura por meio do retorno do syscall, que retorna negativo em casos de erros.

O algoritmo desenvolvido:

```
aberturaArquivo:
    la $a0, local_arquivo       # carrega o endereço do arquivo em $a0
    li $a1, 0           # seta a flag do arquivo em 0 para modo de leitura
    la $a0, local_arquivo       # carrega o nome do arquivo em $a0
    li $v0, 13          # 0 = abre o arquivo em modo de leitura
    syscall                 # faz a chamada de sistema n. 13 para abrir o arquivo
    la $t0, descritor_arquivo    # armazena em $t0 o endereço do descritor do arquivo
    sw $v0, 0($t0)              # armazena em descritor_arquivo o valor de retorno da chamada
    ## Testa a abertura do arquivo:
    add $t1, $zero, $t0      # armazena em $t1 o valor de retorno da chamada
```

[1] todos os códigos exceto declarações estão no segmento .text

```
          bltz  $t1,  encerraPrograma     # se o retorno da chamada for
negativo, houve erro na abertura do arquivo


          lw $a0, 0($t0)              # armazena o valor do descritor do
arquivo em $a0
         jr  $ra
```

Já para a leitura do arquivo, precisamos apenas das informações do arquivo, realizar a leitura, que se dá por 4 bytes de cada vez, copiá-los para a memória, incrementar a memória em 4 para o próximo endereço, verificar se foi lido todos os bytes e seguir em loop até todos os caracteres terem sido lidos.

A proposta de algoritmo:

```
leituraArquivo:
    la $a1, buffer_leitura       # carrega em $a1 o endereço da variável
buffer_leitura
    li  $a2,  4              # carrega em $a2 o número de bytes que serão
lidos do arquivo
    li  $v0,  14             # carrega em $v0 a chamada que será feita ao
sistema (14 = leitura arquivo)
   syscall

## Armazena o valor de buffer_leitura para memoria_text e incrementa o
endereço de memoria_text
     lw  $a2,  endereco_texto          # carrega o endereço inicial de
memoria_texto em $a2
    lw $a3, 0($a1)           # carrega o valor contido no buffer_leitura
em $a3
    sw $a3, 0($a2)           # armazena o valor do buffer (endereço tá em
$a1) para o endereço de memoria_text

     la  $s0,  endereco_texto          # carrega o endereço da variável
endereco_texto
   addi $t1, $a2, 4          # incrementa 4 bytes no endereço de memoria
   sw $t1, 0($s0)            # armazena o novo valor do endereco_texto
## DO-WHILE:
    bgtz  $v0,  leituraArquivo     # se o número de caracteres lidos for
maior que 0, continua a leitura do arquivo
   jr  $ra
```

Por fim, para o fechamento do arquivo, precisamos apenas chamar o sistema operacional:

```
fechaArquivo:
   li $v0, 16
   syscall
   jr  $ra
```

Equivalentemente temos as mesmas operações para o arquivo de dados:

```
aberturaArquivoData:
   la $a0, local_arquivo_data    # carrega o endereço do arquivo em $a0
   li $a1, 0             # seta a flag do arquivo em 0 para modo de
leitura
   la $a0, local_arquivo_data    # carrega o nome do arquivo em $a0
   li $v0, 13            # 0 = abre o arquivo em modo de leitura
   syscall               # faz a chamada de sistema n. 13 para abrir o
arquivo
   la $t0, descritor_arquivo     # armazena em $t0 o endereço do
descritor do arquivo
   sw $v0, 0($t0)            # armazena em descritor_arquivo o valor de
retorno da chamada
   ## Testa a abertura do arquivo:
   add $t1, $zero, $t0       # armazena em $t1 o valor de retorno da
chamada
   bltz $t1, encerraPrograma     # se o retorno da chamada for
negativo, houve erro na abertura do arquivo

   lw $a0, 0($t0)            # armazena o valor do descritor do arquivo
em $a0
   jr  $ra
```

Leitura:

```
leituraArquivoData:
   la $a1, buffer_leitura      # carrega em $a1 o endereço da variável
buffer_leitura
   li $a2, 4           # carrega em $a2 o número de bytes que serão
lidos do arquivo
```

```
   li $v0, 14          # carrega em $v0 a chamada que será feita ao
sistema (14 = leitura arquivo)
   syscall

## Armazena o valor de buffer_leitura para memoria_text e incrementa o
endereço de memoria_text
   lw $a2, endereco_data       # carrega o endereço inicial de
memoria_texto em $a2
   lw $a3, 0($a1)          # carrega o valor contido no buffer_leitura
em $a3
   sw $a3, 0($a2)          # armazena o valor do buffer (endereço tá em
$a1) para o endereço de memoria_text

   la $s0, endereco_data       # carrega o endereço da variável
endereco_texto
   addi $t1, $a2, 4        # incrementa 4 bytes no endereço de memoria
   sw $t1, 0($s0)          # armazena o novo valor do endereco_texto
## DO-WHILE:
   bgtz $v0, leituraArquivoData    # se o número de caracteres lidos
for maior que 0, continua a leitura do arquivo
   jr  $ra
```

E fechamento de arquivo:

```
fechaArquivo:
   li $v0, 16
   syscall
   jr  $ra
```

Com as instruções do arquivo binário na memória, foi preciso simular o ciclo de busca - decodificação - execução.

Para iniciar o processo da busca, precisamos atualizar o registrador PC para a primeira instrução na memória:

```
inicioBuscaInstrucao:
## Carrega em PC o endereço inicial de memoria_text
   la $a0, PC          ## carrega endereço texto
   la $a1, endereco_texto      ## carrega o valor do endereço inicial
de memoria_text pra $a1
   sw $a1, 0($a0)          ## armazena em PC o valor de $a1
   jr  $ra
```

E iniciar o processo efetivo de busca:

```
## busca_instrucao: seta os valores de PC e IR para serem executados
buscaInstrucao:
   la $a0, PC          ## carrega endereço de PC em $a0
   lw $a1, 0($a0)          ## carrega o valor de PC em $a1
   lw $a2, 0($a1)          ## carrega o valor armazenado no endereço de
PC

   la $a3, IR          ## carrega o endereço de IR em $a3
   sw $a2, 0($a3)          ## armazena a instrução na variável IR

   ## Atualiza o PC para apontar para a próxima instrução
   addi $a1, $a1, 4
   sw $a1, 0($a0)          ## armazena o valor incrementado em PC

   jr  $ra
```

Com a busca realizada, PC e IR atualizados, iniciamos o processo de decodificação. A decodificação ocorre separando os campos da instrução contida em IR, selecionando os campos respectivos a cada tipo de função (I, R e J) e guardando-a em registradores separados, para otimizar, inicialmente mapeamos os registradores:

```
# Mapa Campos da Instrução - Decodifica Instrução

# INSTRUÇÕES R
# $s0 = campo OPCODE
# $s1 = campo funct
# $s2 = campo shamt
# $s3 = campo rd
# $s4 = campo rt
# $s5 = campo rs

# INSTRUÇÕES I
# $s0 = campo OPCODE
# $s1 = campo Immediate
# $s4 = campo rt
# $s5 = campo rs
```

```
# INSTRUÇÕES J
# $s0 = campo OPCODE
# $s1 = campo Address
```

E então decodificamos:

```
decodificaInstrucao:
# ideia addi    $sp, $sp, -4
# ideia sw  $ra, 0($sp)
   lw $a0, IR

   srl $s0, $a0, 26            ## $s0 = campo OPCODE
   andi $s1, $a0, 0x0000003F       ## $s1 = campo FUNCT
   srl $s2, $a0, 6             ## $s2 = campo shamt
   andi $s2, $s2, 0x0000001F
   srl $s3, $a0, 11            ## $s3 = campo rd
   andi $s3, $s3, 0x0000001F
   srl $s4, $a0, 16            ## $s4 = campo rt
   andi $s4, $s4, 0x0000001F
   srl $s5, $a0, 21            ## $s5 = campo rs
   andi $s5, $s5, 0x0000001F
   sll $s6, $s5, 1
   ## DIRECIONA PRO TIPO DE INSTRUÇÃO
   beqz    $s0,    instrucaoR      # if opcode == 0 instrucao do tipo r
   bge     $s0, 4, instrucaoI      # else if opcode >= 4 instrucao do
tipo i
   j       instrucaoJ          # else instrucao do tipo j

# ideia lw  $ra, 0($sp)
# ideia addi    $sp, $sp, 4

instrucaoR:
   jr  $ra

instrucaoI:
   sll $t0, $s3, 11            ## desloca campo rd 11 bits para
esquerda
   sll $t1, $s2, 6             ## desloca campo shamt 6 bits para
esquerda
   add $t2, $t1, $t0           ## soma shamt+rd
   add $s1, $t2, $s1           ## soma shamt+rd+funct
   jr  $ra
```

```
instrucaoJ:
   sll $t0, $s5, 21            ## desloca campo rs 21 bits para
esquerda
   sll $t1, $s4, 16            ## desloca campo rt 16 bits para
esquerda
   sll $t2, $s3, 11            ## desloca campo rd 11 bits para
esquerda
   sll $t3, $s2, 6             ## desloca campo shamt 6 bits para
esquerda
   add $t4, $t0, $t1           ## $t4 -> rs+rt
   add $t5, $t2, $t3           ## $t5 -> rd+shamt
   add $t4, $t4, $t5           ## $t4 + $t5
   sw  $t4, 0($s1)             ## $s1 -> address
   jr  $ra
```

Agora, com a instrução devidamente decodificada é cabível iniciar o processo de execução. A execução de uma instrução inicia identificando qual instrução está sendo referida, para isso basta apenas realizar uma comparação no formato Switch-Case, se o valor da função for igual ao da função, executa a função:

```
executaInstrucao:

   beqz    $s0,       comparaFunct
   beq $s0, 8,    instADDI
   beq $s0, 9,    instADDIU
   beq $s0, 0x1c, instMUL
   beq $s0, 5,    instBNE
   beq $s0, 2,    instJ
   beq $s0, 3,    instJAL
   beq $s0, 0x23,  instLW
   beq $s0, 0x2b,  instSW
   beq $s0, 0xf,   instLUI
   beq $s0, 0xd,  instORI

comparaFunct:
   beq $s1, 0xc,  instSYS
   beq $s1, 0x08, instJR
   beq $s1, 0x20, instADD
   beq $s1, 0x21, instADDU
   beq $s1, 0xb   instMOVE
```

O processo de executar cada função requer a simulação da mesma, para simular, é necessário converter os registradores do buffer para encontrar seus endereços reais As funções básicas necessárias para o cálculo do fatorial de um número são:

- ADD: A função ADD é simplesmente a soma com sinal de valores que estão em dois registradores e armazenamento em um terceiro. Estes registradores já foram identificados no processo de decodificação.

```
## add
instADD:
#rd = rs + rt
       ## rs
   la $t0, regs                 ## endereço inicial dos registradores
   sll $t1, $s5, 2              ## 4 * n. do registrador rs
   add $t1, $t1, $t0            ## endereço inicial de regs + 4 * n. do
registrador
   lw $t2, 0($t1)               ## $t2 = valor armazenado em regs[rs]
   ## rt
   sll $t1, $s4, 2              ## 4 * n. do registrador rt
   add $t1, $t1, $t0            ## endereço inicial de regs + 4 * n. do
registrador
   lw $t3, 0($t1)               ## $t3 = valor armazenado em regs[rt]
   ## rd
   sll $t1, $s3, 2              ## 4 * n. do registrador de destino (rd)
   add $t1, $t1, $t0            ## $t1 -> endereço do registrador rd
   add $t4, $t3, $t2            ## $t4 = rs+rt
   sw  $t4, 0($t1)
   jr  $ra
```

- ADDU: A função ADDU possui a mesma lógica da função ADD, a diferença é que a soma é desenvolvida sem sinal (unsigned).

```
# addu
instADDU:
#rd = rs + rt (sem sinal)
       ## rs
   la $t0, regs                 ## endereço inicial dos registradores
   sll $t1, $s5, 2              ## 4 * n. do registrador rs
```

```
   add $t1, $t1, $t0           ## endereço inicial de regs + 4 * n. do
registrador
   lw $t2, 0($t1)              ## $t2 = valor armazenado em regs[rs]
   ## rt
   sll $t1, $s4, 2             ## 4 * n. do registrador rt
   add $t1, $t1, $t0           ## endereço inicial de regs + 4 * n. do
registrador
   lw $t3, 0($t1)              ## $t3 = valor armazenado em regs[rt]
   ## rd
   sll $t1, $s3, 2              ## 4 * n. do registrador de destino
(rd)
   add $t1, $t1, $t0           ## $t1 -> endereço do registrador rd
   add $t4, $t3, $t2           ## $t4 = rs+rt
   sw  $t4, 0($t1)
   jr  $ra
```

- ADDI: A função ADDI por sua vez, soma com sinal o valor de um registrador e um valor imediato:

```
## addi
instADDI:
   la $t0, regs                ## endereço inicial dos registradores
   lw $t1, 0($s1)              ## valor Imm armazenado
   sll $t2, $s5, 2             ## 4 * valor armazenado em rs
   add $t3, $t2, $t0           ## endereço do registrador rs
   lw $t2, 0($t3)               ## $t2 -> valor armazenado em rs
   add $t2, $t2, $t1           ## rs+imm
   ## rt
   sll $t1, $s4, 2             ## 4 * valor armazenado em rt
   add $t3, $t1, $t0           ## $t3 -> endereço do registrador rt
   sw $t2, 0($t3)              ## registrador rt = rs +imm
   jr  $ra
```

- ADDIU: A função ADDIU retorna a soma com sinal do valor em um registrador com um valor imediato. Nas funções com sinal os valores são convertidos automaticamente.

```
# addiu
instADDIU:
   la $t0, regs                ## endereço inicial dos registradores
   lw $t1, 0($s1)              ## valor Imm armazenado
```

```
   sll $t2, $s5, 2            ## 4 * valor armazenado em rs
   add $t3, $t2, $t0          ## endereço do registrador rs
   lw $t2, 0($t3)             ## $t2 -> valor armazenado em rs
   add $t2, $t2, $t1          ## rs+imm
   ## rt
   sll $t1, $s4, 2            ## 4 * valor armazenado em rt
   add $t3, $t1, $t0          ## $t3 -> endereço do registrador rt
   sw $t2, 0($t3)             ## registrador rt = rs +imm
   jr  $ra
```

- SW: A função store word possui diversas aplicações, ela pode armazenar uma palavra de um registrador para a memória, entre registradores e de um registrador para outro com endereçamento base.

```
## sw
instSW:
   la  $t0, regs              ## endereço inicial dos registradores
   sll $t1, $s4, 2            ## $t1 -> valor do campo rt * 4
   add $t1, $t1, $t0          ## $t1 -> endereço efetivo do
registrador rt
   sll $t2, $s5, 2             ## $t2 -> valor do campo rs * 4
   add $t2, $t2, $t0          ## $t2 -> endereço efetivo do
registrador rs
   lw  $t3, 0($s1)            ## $t3 -> valor do campo immediate
   add $t3, $t3, $t2           ## $t3 -> imm + rs
   lw  $t0, 0($t1)
   sw  $t0, 0($t3)            ## armazena o valor de rt no endereço
imm + rs
   jr  $ra
```

- LW: A função load word, assim como store word, possui mais de uma sintaxe, ela copia a palavra de uma posição da memória para um registrador, copia palavra entre registradores ou entre um registrador e outro + endereço base.

```
## lw
instLW:
   la $t0, regs
   sll $t1, $s4, 2             ## $t1 -> valor do campo rt * 4
   add $t1, $t1, $t0          ## $t1 -> endereço efetivo do
registrador rt
   sll $t2, $s5, 2            ## $t2 -> valor do campo rs * 4
```

```
   add $t2, $t2, $t0            ## $t2 -> endereço efetivo do
registrador rs
   lw  $t3, 0($s1)              ## $t3 -> valor do campo immediate
   add $t3, $t3, $t2           ## $t3 -> imm + rs
   lw  $t1, 0($t3)
   jr  $ra
```

- LUI: A função load upper immediate carrega um valor de 16 bits para a parte mais significativa de um registrador ou memória e preenche o restante com zeros.

```
## lui
instLUI:
   la $t0, regs                 ## endereço inicial dos registradores
       sll $t1, $s5, 2             ## $t1 -> valor do campo rt * 4
       add $t2, $t1, $t0           # $t2 -> Endereço base + rt
       andi $t3, $s1, 0x00FF           # $t3 -> 8 bits menos
significativos de imm
       sw $t3, 0($t2)              # Armazena no endereço efetivo do
registrador rt
       jr  $ra
```

- ORI: A função or immediate realiza a operação OR entre um registrador e um valor imediato.

```
## ori
instORI:
   la $t0, regs                 ## endereço inicial dos registradores
       sll $t1, $s5, 2             ## $t1 -> valor do campo rs * 4
       add $t2, $t1, $t0           # $t2 -> Endereço base + rs
       lw $t3, 0($t2)              ## $t3 -> Valor armazenado em rs
       or $t4, $t3, $s1            ## $t4 -> or = rs e imm
       sll $t1, $s4, 2             ## $t1 -> valor do campo rt * 4
       add $t1, $t1, $t0           ## Endereço base + rt
       sw $t3, 0($t1)               # Armazena resultado de or no
endereço efetivo de rt
       jr  $ra
```

Um detalhe importante a ser mencionado é que as funções LUI e ORI em conjunto formam a pseudo-instrução LA (load address). O MIPS converte as pseudo-instruções por conveniência, neste caso, ele carrega um endereço usando LUI, este endereço fica na parte superior do registrador, visto que ambas as funções

operam com 16 bits, para completar o restante, é realizada a função ORI que não altera a parte superior, apenas a parte inferior.

- MOVE: A função MOVE copia o valor de um registrador para outro.

```
## move
instMOVE:
   la $t0, regs
       sll $t1, $s5, 2             ## $t1 -> valor do campo rt * 4
       add $t1, $t1, $t0           ## Calcula o endereço do rt
       lw $t5, 0($t1)              ## $t5 -> Carrega o valor do rt $t5
       sll $t3, $s4, 2             ## $t3 -> valor do campo rs * 4
       add $t3, $t3, $t0           ## Endereço base + rs
       sw $t5, 0($t3)              ## Armazena o valor de rt em rs
       jr  $ra
```

- BNE: A função branch if not equal, opera um desvio para uma parte do código se o valor de dois registradores é diferente.

```
## bne
instBNE:
   la  $t0, regs
   sll $t1, $s5, 2                 ## $t1 -> valor do campo rt * 4
       add $t1, $t1, $t0               ## Calcula o endereço do rt
       lw  $t5, 0($t1)                 ## $t5 -> Carrega o valor do rt
$t5
       sll $t3, $s4, 2                 ## $t3 -> valor do campo rs * 4
       add $t3, $t3, $t0               ## Calcula o endereço do rs
       lw  $t4, 0($t3)                 ## $t4 -> Carrega o valor do rs

       beq $t4, $t3, fimBNE            ## se valores forem iguais,
encerra a instrução e retorna pra main

   lw $t0, PC                    ## $t0 -> valor de PC
   lw $t1, 0($s1)                  ## $t1 -> valor do immediate
   sll $t1, $t1, 2                 ## Imm * 4 -> deslocamento de 4
bytes
   add $t2, $t0, $t1               ## $t2 -> PC + n. de instruções a
serem deslocadas
   sw $t2, PC

fimBNE:
```

```
   jr  $ra
```

- MUL: A função de multiplicação entre dois registradores, com sinal e sem overflow. O resultado é um valor de 64 bits, onde os primeiros 32 bits são armazenados no registrador especial HI e os últimos 32 bits são armazenados no registrador especial LO.

```
## mul
instMUL:
       la  $t0, regs
       sll $t1, $s5, 2                  ## $t1 -> valor do campo rt * 4
       add $t1, $t1, $t0                ## Calcula o endereço do rt
       lw  $t5, 0($t1)                  ## $t5 -> Carrega o valor do rt
$t5
       sll $t3, $s4, 2                  ## $t3 -> valor do campo rs * 4
       add $t3, $t3, $t0                ## Calcula o endereço do rs
       lw  $t4, 0($t3)                  ## $t4 -> Carrega o valor do rs

       sll $t1, $s3, 2                  ## $t1 -> valor do campo rd * 4
       add $t1, $t1, $t0                ## Calcula o endereço do rd
       mul $t2, $t4, $t5                ## $t2 -> multiplicação de rs+rt
       sw $t2, 0($t1)                   ## armazena o resultado em RD

       mfhi    $t0                 # $t0 -> valor de hi
       la  $t1, HI
       sw  $t0, 0($t1)             # Insere $t0 no endereço de hi
       mflo    $t2                 # $t2 -> valor de lo
       la  $t3, LO
       sw  $t2, 0($t3)             # Insere $t2 no endereço de lo

       jr  $ra
```

- J: A função jump parte incondicionalmente da parte atual do código para a parte indicada, requer uma *label,* um endereço para o salto.

```
## j
instJ:
   lw  $t0, 0($s1)          # $t0 -> address (26 bits)
       la  $t1, PC              # $t1 -> endereço de PC
```

```
        lw  $t2, endereco_texto     # $t2 -> endereço inicial da memória
de texto
        sll $t0, $t0, 2         # $t0 -> address * 4
        add     $t0, $t0, $t2           # $t0 <- endereço efetivo da
label desejada
        sw  $t0, 0($t1)         # Armazena em PC o endereço da instrução
definida pela label

        jr $ra
```

- JR: A função jump to register, como o nome sugere, muda o valor do PC para o valor contido no registrador. Ela pode também, mudar o endereço de retorno do registrador RA,no caso de ser uma chamada subrotina.

```
## jr
instJR:
   la $t0, regs                 ## endereço inicial dos registradores
   lw $t1, 124($t0)             ## endereço armazenado em $ra
   la $t2, PC                ## endereço armazenado em PC
   sw $t1, 0($t2)               ## armazena em PC endereço armazenado
em $ra
```

- JAL: A função jump and link desvia para o endereço de subrotina e copia o valor de PC para RA.

```
## jal
instJAL:
   la $t0, regs                 ## endereço inicial dos registradores
   lw $t1, PC                ## endereço armazenado em PC
   sw $t1, 31($t0)              ## armazena endereço da próxima
instrução em $ra

   j instJ
```

Tendo o processo de busca - decodificação - execução implementado, basta realizar este processo em loop:

```
main:
    jal     aberturaArquivo
    jal     leituraArquivo
    jal     fechaArquivo
    jal     aberturaArquivoData
    jal     leituraArquivoData
    jal     fechaArquivo
    jal     inicioBuscaInstrucao
    li $s7, 0       # Inicializa o contador ($s7)
    loop_main:
    jal buscaInstrucao
    jal decodificaInstrucao
    jal     executaInstrucao
    # Incrementa o contador
        lw $t1, count   # Carrega o valor atual de count para $t1
        addi $s7, $s7, 1    # Incrementa $s7
        sw $s7, count   # Armazena de volta em count

        # Verifica se o contador atingiu 4
        li $t2, 4       # Carrega o valor 4 para $t2
        bne $s7, $t2, loop  # Se $s7 != $t2, volta para loop
    jal     encerraPrograma
```

Resultados

Após o desenvolvimento e integração dos algoritmos, foi efetuada a depuração no software que resultou em mensagem de erro:
Error in: /home/usuário/Documentos/t1/simulador (1).asm line 216: Runtime exception at 0x00400284: arithmetic overflow
Go: execution terminated with errors.

O que indica overflow, ou estouro em uma operação de função ADD simulada, na soma da conversão do registrador para endereço efetivo:

```
add $t1, $t1, $t0                   ## endereço inicial de regs + 4 * n. do registrador
```

Este erro é decorrente de erros anteriores e expõe o não funcionamento do programa.

CONCLUSÃO

Como considerações técnicas, o não funcionamento do simulador foi identificado pela má manipulação dos registradores com possíveis trocas, erros de endereçamento, funções utilizadas erroneamente, erros de lógicas nas simulações

de funções e declaração das variáveis, além da não utilização efetiva da pilha de execução que se deu pela falta de conhecimento técnico e compreensão da atividade.

A ausência de testes é o fator primordial para a falha, o tempo necessário para o desenvolvimento do simulador foi calculado imprecisamente e a simultaneidade com outras atividades acadêmicas impediram a execução de uma das mais importantes etapas do processo, impossibilitando a correção dos erros e consequente êxito na execução da tarefa.

Como isso, podemos então indicar um estudo técnico a respeito do funcionamento da pilha de execução como sugestão de melhoria, além da realização dos testes individuais dos procedimentos e verificação na estrutura do código, análise lógica e correção das falhas. Com estas pequenas mudanças é possível obter a resolução para os problemas enfrentados.

Ademais, é de suma importância salientar o desenvolvimento pessoal envolvido no processo de trabalho, os conhecimentos adquiridos acerca da estrutura de um processador, do seu funcionamento, dos caminhos que os dados percorrem e a experiência com programação em linguagem Assembly, além da familiarização com as ferramentas como o software Mars, o que é indubitavelmente um indicativo de sucesso para as integrantes do trabalho.

BIBLIOGRAFIA

- https://dicionario.priberam.org/simulador#:~:text=Dispositivo%20capaz%20de%20reproduzir%20o,se%20pretende%20seguir%20a%20evolução. - ACESSO EM: 03/07

- https://www.google.com/imgres?q=instru%C3%A7%C3%A3o%20tipo%20r%20%2C%20i%20e%20j%20no%20mips&imgurl=https%3A%2F%2Fwww.researchgate.net%2Fpublication%2F342697219%2Ffigure%2Ffig5%2FAS%3A909739269578766%401593910152861%2FFigura-6-Formato-de-Instrucoes-do-MIPS-rs-e-rt-sao-os-registradore-com-os-dados-e-rd-o.ppm&imgrefurl=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FFigura-6-Formato-de-Instrucoes-do-MIPS-rs-e-rt-sao-os-registradore-com-os-dados-e-rd-o_fig5_342697219&docid=ZusAr8ueOquWcM&tbnid=mZF3LzGdfZejtM&vet=12ahUKEwiQzMLO3ouHAxVzkZUCHUDxC_EQM3oECBkQAA..i&w=850&h=199&hcb=2&ved=2ahUKEwiQzMLO3ouHAxVzkZUCHUDxC_EQM3oECBkQAA - ACESSO EM: 03/07

- https://www.inf.ufpr.br/wagner/ci243/GuiaMIPS.pdf - ACESSO EM: 03/07

- https://www.ufsm.br/pet/sistemas-de-informacao/2018/09/01/mars-ide-para-programacao-em-assembl - ACESSO EM: 03/07
- Material da disciplina Organização de Computadores por BARATTO, G.

REFERÊNCIAS:

SOFTWARE DE APOIO

Disponível em: https://courses.missouristate.edu/kenvollmar/mars/

Em linux:

Fazer o download do software, abrir o terminal, migrar para a pasta onde está o download, e digitar a linha de comando: java -jar Mars4_5.jar.

Problemas eventuais podem ocorrer pelo nome do arquivo que deve ser adaptado..