

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS  
*CAMPUS* DIVINÓPOLIS  
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

José Marconi de A. Júnior

SIMULADOR ARQUITETURA MIPS



Divinópolis  
2025

# Sumário

<b>Sumário</b>	<b>iii</b>
<b>1 Metodologia</b>	<b>vii</b>
1.1 Instruções	vii
1.2 Implementação	vii
1.2.1 Escalonador	viii
1.2.2 Função executeProcess() da Classe Core	ix
1.2.3 Função execute() Classe Process	ix
<b>2 Resultados</b>	<b>xi</b>

---

**Capítulo****1**

---

# Metodologia

## 1.1 Instruções

O README no GitHub especifica como devem ser feitas as instruções. Para esta etapa do trabalho, foram inseridos novos arquivos de instruções (nomeados `instructiosN.txt`, para  $N \geq 0$ ), enquanto é utilizado um mesmo arquivo para inicializar os registradores para todas as instruções.

Foi adotada a lógica de que cada arquivo será um processo e cada processo será uma thread. Sendo assim, cada processo/thread pode conter mais de uma linha, o que aumenta o seu tempo de execução.

## 1.2 Implementação

Foram criadas duas novas classes, denominadas `Processos` e `Escalonador`. Também foram realizadas modificações na classe `Core`, alterando a função antiga `activate()` para `executeProcess()`.

A classe `Processos` cria os processos e realiza a leitura dos arquivos de instruções, efetuando a escrita na RAM (`StructionsLoad()`) e nos registradores, por meio do `RegisterLoad()` que chama uma função na unidade de controle responsável por acionar uma função da Pipeline.

O que é chamado de processo é o próprio PCB do mesmo, sendo composto por:

- Id: Identificador único do processo;
- State: estado de vida do processo;

- Quantum: quantum máximo que o processo pode utilizar;
- ActualInstruction: índice da instrução atual no vetor de instruções do processo;
- Regs: registradores utilizados na instrução atual;
- Files: nome do arquivo que contém as instruções sendo executadas;
- Quantum total: quantos ciclos de clock ao todo o processo utilizou.

O quantum utilizado foi fixado em 12 para todos os processos.

A classe contém a função `execute()` responsável por executar as instruções (a primeira ação é alterar o estado do ciclo de vida para `RUNNING`), interagindo com a unidade de controle e a Pipeline. Também possui as funções `block()` e `unlock()` para alterar o estado do ciclo de vida do processo para `BLOCK` e `READY`, respectivamente.

O estado de vida do processo pode ser resumido como:

- `READY`: quando o processo está pronto para ser executado na fila;
- `RUNNING`: quando o processo está sendo executado;
- `BLOCK`: quando o quantum excede e o processo deve ser parado;
- `TERMINATED`: quando o processo é finalizado;

Sempre que o processo sai da execução, seu estado é verificado. Caso esteja em `BLOCK`, ele é colocado novamente no fim da fila; caso esteja em `TERMINATED`, ele é finalizado e retirado da fila.

### 1.2.1 Escalonador

A classe `Scheduler` implementa o escalonador, utilizando a política `First Come First Service` (FCFS), ou seja, o primeiro que entra é o primeiro a ser executado.

Primeiramente, o construtor é chamado na função `main`, recebendo as memórias e o vetor de Cores como referência. Ainda no construtor, é chamada a função `createAndAddProcess()` que cria um novo processo e o adiciona à fila de processos. Em seguida, a função `schedule()` é chamada.

Na função `schedule()`, a política é aplicada. Um mutex é utilizado para proteger a execução, e o vetor de threads a ser executado é definido. Há um laço `while` que roda

enquanto a fila não estiver vazia ou enquanto algum Core estiver ocupado. Dentro dele, há uma condicional que verifica se a fila não está vazia. Nesse caso, entra em um laço que percorre os Cores. Se um Core estiver disponível, uma thread é criada, chamando a função `executeProcess()` da classe `Core`, que será responsável pela execução do processo, passando as devidas referências. Logo após, o vetor de threads é iniciado com a função `detach()`, permitindo que as threads sejam executadas de forma independente.

### 1.2.2 Função `executeProcess()` da Classe `Core`

A função utiliza um mutex (`coutMutex`) para sincronizar o acesso à saída padrão. Isso é necessário em um ambiente multithreaded para evitar que múltiplas threads escrevam na saída padrão ao mesmo tempo, o que pode gerar mensagens misturadas e ilegíveis.

Nela, a função `execute()` da classe `Process` é chamada para realizar a execução do processo. Em seguida, o estado de vida do processo é analisado. Caso esteja em `BLOCK`, ele é reposicionado no fim da fila. Caso esteja em `TERMINATED`, é exibida uma mensagem indicando a conclusão do processo. Por fim, o Core é liberado para a próxima execução.

### 1.2.3 Função `execute()` Classe `Process`

A função retorna informações sobre o que está sendo executado, incluindo o ID do processo, a instrução, os registradores, o arquivo fonte e o clock ao final da execução da instrução.

O controle de quantum é feito por meio de um contador. Dentro da lógica, a execução de uma instrução pode ultrapassar o quantum máximo por alguns poucos ciclos de clock. Isso ocorre porque o controle desse clock não é realizado na Pipeline, mas apenas na instrução. Assim, o clock retornado da execução na Pipeline pode exceder ligeiramente o quantum.

De forma geral, a função controla o quantum e retorna o output, enquanto a execução das instruções é realizada com o uso da unidade de controle e da Pipeline.

## Capítulo 2

# Resultados

A saída foi conforme o esperado, considerando o que foi implementado:

```
Processo 1 sendo executado no core 1

[Processo 1] Executando instrução: PC=0 Opcode=0 Destino=R1 Valor1=10 Valor2=5
State:1
Arquivo fonte: data/instructions0.txt
Quantum total utilizado: 13
clock = 13

--- PROCESSO BLOQUEADO, QUANTUM EXCEDIDO ---
Processo 1 bloqueado no Core 1

Processo 2 sendo executado no core 2

Processo 3 sendo executado no core 1

[Processo 2] Executando instrução: PC=0 Opcode=0 Destino=R1 Valor1=10 Valor2=5
State:1
Arquivo fonte: data/instructions1.txt
Quantum total utilizado: 10
clock = 10

[Processo 2] Executando instrução: PC=4 Opcode=1 Destino=R4 Valor1=15 Valor2=5
State:1
Arquivo fonte: data/instructions1.txt
Quantum total utilizado: 14
clock = 14
```

Figura 1 – Exemplo de output.

É possível visualizar quando um processo foi bloqueado e em qual core isso aconteceu. Também é exibido em qual core cada processo está sendo executado. É importante lembrar

que cada core possui um clock total diferente. O clock final total de cada core pode ser visualizado ao final dos últimos processos que foram executados e concluídos.

```
Processo 5 sendo executado no core 2

Processo 1 sendo executado no core 1

[Processo 5] Executando instrução: PC=28 Opcode=1 Destino=R9 Valor1=0 Valor2=20
State:1
Arquivo fonte: data/instructions4.txt
Quantum total utilizado: 96
clock = 158

----- Processo 5 concluído-----

Processo 5 finalizado no Core 2

[Processo 1] Executando instrução: PC=32 Opcode=3 Destino=R21 Valor1=7 Valor2=-7
State:1
Arquivo fonte: data/instructions0.txt
Quantum total utilizado: 117
clock = 167

----- Processo 1 concluído-----

Processo 1 finalizado no Core 1
```

Figura 2 – Exemplo de output.

Observa-se que o core 1 realizou, ao todo, 167 ciclos de clock, enquanto o core 2 executou 158 ciclos de clock. É possível confirmar os valores realizando o cálculo dos quantus totais de todos os processos, nesse exemplo o somatório de ambos os cores é de 325 ciclos de clock total. A seguir é possível visualizar que o somatório de quantum total dos processos, foi o mesmo 325:

Process 1	117
Process 2	56
Process 3	17
Process 4	24
Process 5	96
Process 6	15
Total	325

Por fim, é exibido o estado final da RAM e uma mensagem indicando que todos os processos foram finalizados:

```
Endereço 0 -> Opcode: 0, Destino: R1, R1: 2, R2: 3
Endereço 1 -> Opcode: 1, Destino: R10, R1: 5, R2: 2
Endereço 2 -> Opcode: 0, Destino: R7, R1: 9, R2: 21
Endereço 3 -> Opcode: 1, Destino: R19, R1: 11, R2: 4
Endereço 4 -> Opcode: 0, Destino: R21, R1: 13, R2: 9
Endereço 5 -> Opcode: 1, Destino: R9, R1: 20, R2: 2
Endereço 6 -> Opcode: 0, Destino: R12, R1: 12, R2: 5
Endereço 7 -> Opcode: 1, Destino: R20, R1: 1, R2: 3
Endereço 8 -> Opcode: 3, Destino: R21, R1: 4, R2: 19
Endereço 9 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 10 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 11 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 12 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 13 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 14 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 15 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 16 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 17 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 18 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 19 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 20 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 21 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 22 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 23 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 24 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 25 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 26 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 27 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 28 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 29 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 30 -> Opcode: 0, Destino: R0, R1: 0, R2: 0
Endereço 31 -> Opcode: 0, Destino: R0, R1: 0, R2: 0

Todos os processos foram finalizados.
```

Figura 3 – Exemplo da ram ao final da execução geral.