UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

CENTRO DE CIÊNCIAS EXATAS E DA TERRA - CCET

DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA - DIMAp

Turma: 2º período em Ciência da Computação, 2024.2 (DIM0119)

Docente: MÁRCIO EDUARDO KREUTZ

Relatório - Processadores

Carlos Alberto de Lima Neto

Claudivan Costa de Lima Junior 🦄

Yuri Maximiliano Brasileiro Santos 🏳️

Natal - RN2024

SUMÁRIO

Neander

2.Ramsés

3.César

Neander

1.Introdução

1.1 Objetivo do Código

O código implementa uma simulação do processador Neander, um modelo educacional usado para demonstrar conceitos básicos de arquitetura de computadores. Ele simula componentes como registradores, unidade lógica e aritmética (ULA), unidade de controle (UC), memória e instruções.

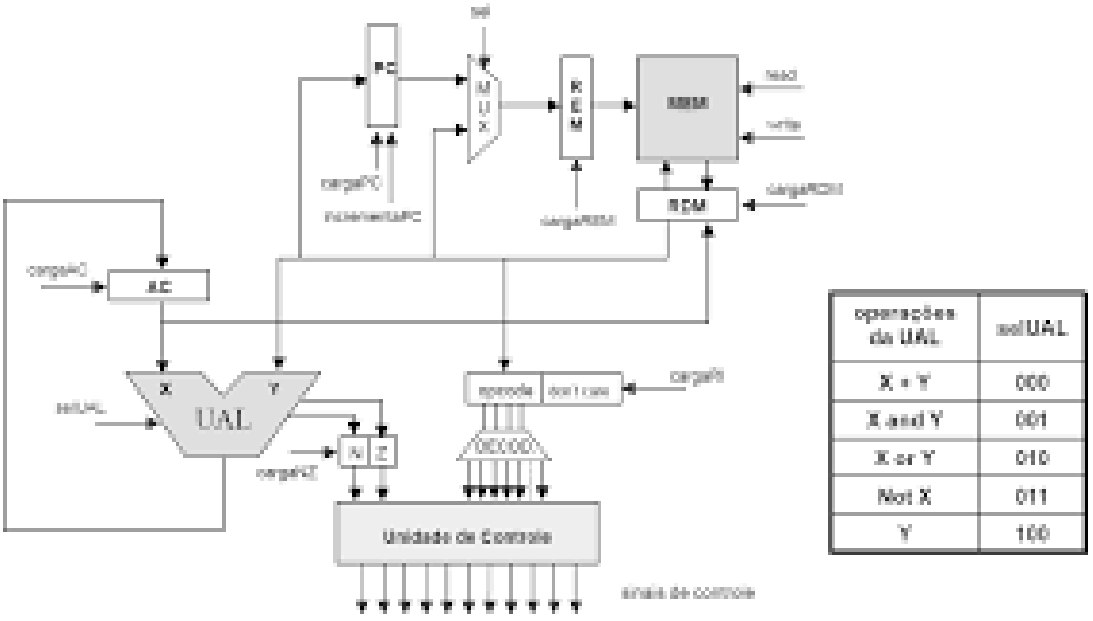
1.2 Estrutura Geral

A classe `Neander` encapsula os principais componentes de um processador e é organizada em métodos para simular operações como leitura, escrita, decodificação de instruções e execução de comandos.

Principais Componentes:

1. **Registradores:**
   - `AC`: Acumulador, usado para armazenar resultados intermediários.
   - `PC`: Contador de programa, que aponta para a próxima instrução a ser executada.
   - `RDM`: Registrador de dados da memória, que armazena o dado lido ou a ser escrito na memória.
   - `REM`: Registrador de endereço de memória, que aponta para o endereço da operação.
   - `RI`: Registrador de instrução, que armazena a instrução atual.
2. **Unidade de Controle (UC):** Controla os sinais que coordenam as operações do processador, como leitura/escrita, incremento do PC e seleção de entradas.
3. **Unidade Lógica e Aritmética (ULA):** Realiza operações como soma, AND, OR e atribuição direta de valores.
4. **Memória:**
   - Representada pela classe `memória`, armazena as instruções e os dados usados pelo processador.
5. **NZ (Flags):** Indicadores de sinal (N para negativo e Z para zero) baseados no resultado da ULA.

2. Metodologia



2.1 Funcionamento

Inicialização:

O construtor da classe `Neander` inicializa todos os registradores e sinais de controle com valores padrão (geralmente 0).

Execução Geral:

1. **Busca (`busca`):**
   - Incrementa o PC.
   - Lê a instrução da memória e a armazena no `RI`.
2. **Decodificação e Execução (`selecao`):**
   - Decodifica a instrução armazenada no `RI`.
   - Executa ações específicas com base no opcode.

Instruções Implementadas:

- **LDA (32):** Carrega um valor da memória no acumulador (`AC`).
- **ADD (48):** Soma um valor da memória ao acumulador.
- **STA (16):** Armazena o valor do acumulador em um endereço de memória.
- **HLT (240):** Interrompe a execução.
- **NOP (0):** Não faz nada.
- **NOT (96):** Nega o operando.

Unidade Lógica e Aritmética (`operatio`):

Executa operações com base no sinal de controle `sel_ula`:

- `0`: Soma.
- `1`: Operação AND.
- `2`: Operação OR.
- `3`: Atribuição direta.

Leitura e Escrita na Memória:

- **Leitura (`read`):** Carrega o dado da memória para o `RDM`.
- **Escrita (`write`):** Armazena o valor do `RDM` em um endereço de memória.

Entrada de Instruções (`lertudo`):

Lê instruções de um arquivo e carrega na memória.

2.2 Recursos de Depuração

O código inclui métodos para exibir o estado interno do processador e os sinais de controle:

- `printState`: Mostra os valores dos registradores principais.
- `print_UnitControl`: Exibe o estado dos sinais da unidade de controle.

Essas funções auxiliam na visualização das operações e no acompanhamento do ciclo de execução.

3.Questões

3.1 Justificar o tamanho dos programas (em número de instruções):

O tamanho do programa de dá pelo fato, de ter 11 instruções que tem que ser implementadas e ao mesmo tempo mostrar os sinais de controle para simular exatamente como um processador funciona.

3.2 Comentar sobre facilidades/dificuldades de programação):

-Realizar operações da ula

-Colocar resultados no acumulador

3.3 Comparar desempenho em ciclos de relógio, considerando:

Tem instruções do ciclo de busca e ciclo de instruções, existem instruções que usam dois ciclos e só duas usam 1 ciclo então no geral as instruções usam 2 ciclos sendo bem rápido de realizar.

3.4 Ciclo para instruções lógico/ aritméticas:

Tem instruções de busca e execução, então a instrução de busca é uma instrução de lógica e a execução uma instrução aritmética e tem instruções que operam diretamente no acumulador.

4.Conclusão

O desenvolvimento do simulador para a arquitetura Neander foi essencial para consolidar conhecimentos sobre os princípios de funcionamento de processadores. A implementação dos principais componentes, como a Unidade de Controle (UC), Unidade Lógica e Aritmética (ULA), registradores e memória, permitiu compreender como ocorrem as operações básicas de um processador, incluindo o ciclo de busca, decodificação e execução das instruções.

O trabalho evidenciou a importância do controle do fluxo de dados e da coordenação precisa entre os diferentes módulos para garantir o correto funcionamento do sistema. Além disso, possibilitou a aplicação prática de conceitos teóricos de Arquitetura de Computadores, reforçando habilidades como análise de instruções, manipulação de dados e depuração de sistemas.

A simulação dos programas no Neander demonstrou que a arquitetura é funcional e capaz de realizar operações básicas, como soma, armazenamento e carregamento de dados, respeitando as características originais da ISA. Esse projeto serve como base para futuros estudos e implementações de arquiteturas mais complexas, além de proporcionar uma visão prática dos fundamentos da computação.

Ramsés

1. INTRODUÇÃO

1.1 Objetivo do Código

O simulador do processador Ramsés foi desenvolvido com o objetivo de replicar o funcionamento básico de uma arquitetura computacional simplificada, explorando os conceitos de processamento de instruções, interação entre registradores, memória e unidade de controle. O projeto visa proporcionar um ambiente prático para o estudo de arquiteturas de processadores, destacando a execução de instruções e ciclos de máquina.

1.2 Estrutura Geral

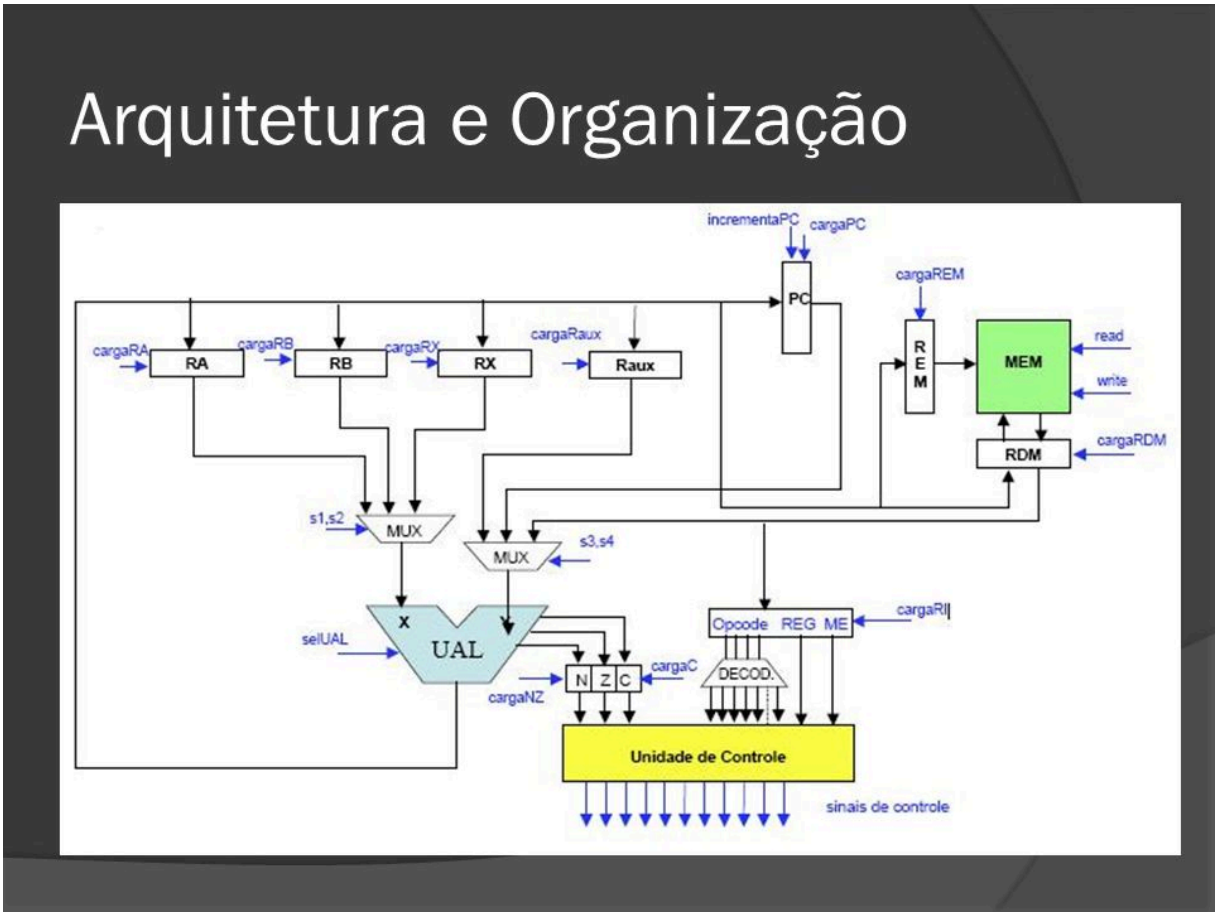
A arquitetura do Ramsés segue um modelo simples com os seguintes componentes principais:

- **Registradores**: Inclui AC (acumulador), AC1 e AC2 (auxiliares), PC (contador de programa), RDM (registrador de dados da memória), REM (registrador de endereços de memória) e RI (registrador de instruções).
- **Unidade Lógica e Aritmética (ULA)**: Realiza operações aritméticas e lógicas.
- **Memória**: Armazena as instruções e os dados utilizados pelo processador.
- **Unidade de Controle (UC)**: Controla o fluxo de dados e a execução das operações com base na instrução carregada.

O código é implementado na linguagem C++ e organiza os componentes do processador em métodos e classes que interagem de forma modular.

---

2. METODOLOGIA



2.1 Funcionamento

O simulador é baseado no ciclo de busca-decodificação-execução e implementa as seguintes instruções:

- **NOP (00)**: Não realiza nenhuma operação.
- **LDR (32)**: Carrega um valor da memória para o acumulador.
- **ADD (48)**: Soma o valor do acumulador com outro valor especificado.
- **STR (16)**: Armazena o valor do acumulador em uma posição de memória.
- **NOT (96)**: Realiza a operação lógica NOT no acumulador.
- **JMP (128)**: Salto incondicional para um endereço específico.
- **JZ (160)**: Salto condicional caso o acumulador seja zero.
- **SUB (112)**: Subtrai um valor do acumulador.
- **JC (176)**: Salto condicional baseado no bit de carry.

O ciclo operacional pode ser resumido em:

1. Buscar a instrução na memória usando o PC.
2. Decodificar a instrução no RI.
3. Executar a instrução com base no seu opcode.

2.2 Recursos de Depuração

O simulador possui um sistema de depuração que permite acompanhar o estado dos registradores e da unidade de controle após cada instrução. Isso auxilia na identificação de erros e na validação do comportamento esperado. Os dados de saída incluem:

- Estados dos registradores (AC, PC, RI, RDM, REM, etc.).
- Sinalizações da UC (carga, leitura, escrita, incremento do PC, etc.).

- Resultado de operações da ULA.

---

3. QUESTÕES

3.1 Justificar o Tamanho dos Programas (em Número de Instruções)

O tamanho dos programas é diretamente influenciado pela simplicidade da arquitetura Ramsés e pela necessidade de programar em baixo nível. Cada operação, incluindo saltos e manipulação de memória, deve ser representada como uma instrução separada. Isso resulta em programas mais extensos quando comparados a arquiteturas de mais alto nível.

3.2 Comentar sobre Facilidades/Dificuldades de Programação

Facilidades:

- A arquitetura simples do Ramsés facilita a compreensão das operações fundamentais de um processador.
- A implementação modular do simulador permite isolar e depurar cada componente (registradores, ULA, etc.).

Dificuldades:

- A ausência de instruções complexas exige que operações mais elaboradas sejam compostas por várias instruções simples, aumentando a complexidade do código.
- Gerenciar saltos condicionais e fluxos de controle pode ser desafiador, especialmente em programas mais extensos.

3.3 Comparar Desempenho em Ciclos de Relógio, Considerando:

O desempenho em ciclos de relógio varia dependendo da instrução. Instruções como **NOP** consomem o menor número de ciclos (apenas a busca e decodificação), enquanto operações como **LDR** e **STR** envolvem acesso à memória, aumentando o tempo de execução. Em média, as instruções lógicas/aritméticas exigem menos ciclos que as instruções de manipulação de memória.

3.4 Ciclo para Instruções Lógico/Aritméticas

As instruções aritméticas e lógicas (como **ADD**, **SUB** e **NOT**) são executadas em um ciclo mais curto, pois envolvem apenas a interação entre os registradores e a ULA. O tempo de execução depende do número de registradores envolvidos e do caminho de dados, mas, em geral, essas operações são mais rápidas comparadas às instruções que requerem acesso à memória (como **LDR** ou **STR**).

---

4. CONCLUSÃO

O simulador do processador Ramsés provou ser uma ferramenta eficaz para a compreensão de arquiteturas de processadores simples. Ele permite visualizar de forma prática o funcionamento de componentes fundamentais, como registradores, ULA e UC. Apesar das limitações de uma arquitetura simplificada, o projeto fornece uma base sólida para o estudo de sistemas computacionais mais complexos. O desafio de programar em baixo nível foi uma experiência valiosa para compreender o impacto de cada instrução no desempenho e funcionamento geral do processador.

Cesar

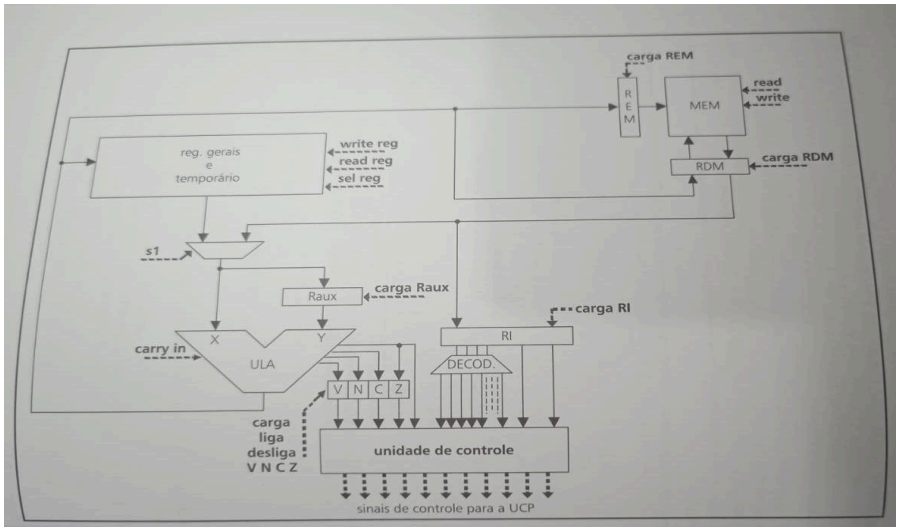
1. INTRODUÇÃO

1.1 Objetivo do código O código tem como objetivo simular o funcionamento do processador Cesar, um processador didático baseado em arquitetura reduzida. Ele é capaz de executar instruções básicas como operações aritméticas, lógicas e de controle de fluxo, sendo amplamente utilizado em ambientes educacionais para estudo de arquiteturas de computadores.

1.2 Estrutura geral O simulador do Cesar foi desenvolvido em C++ e possui uma estrutura modular que inclui:

- **Memória**: Um vetor de 256 posições que armazena instruções e dados.
- **Registradores principais**:
    - **ACC**: Acumulador, usado para operações aritméticas e lógicas.
    - **PC**: Contador de programa, que indica o endereço da próxima instrução a ser executada.
- **Controle de execução**: Indicador que define se o processador está em execução ou parado.
- **Decodificação de instruções**: Implementação de um conjunto de instruções, incluindo LDA, ADD, STA, SUB, JMP, JZ, JN e HLT.

2. METODOLOGIA



2.1 Funcionamento O funcionamento do simulador baseia-se na execução cíclica de três etapas principais:

1. **Busca**: O PC aponta para o endereço da memória onde está localizada a próxima instrução. Esta instrução é carregada no decodificador.
2. **Decodificação**: A instrução é interpretada, dividindo-se o opcode (4 bits mais significativos) e o operando (4 bits menos significativos).
3. **Execução**: A instrução decodificada é executada, alterando os registradores ou a memória, e o PC é atualizado.

2.2 Recursos de depuração O simulador conta com mensagens de erro que auxiliam na identificação de problemas durante a execução:

- **Acesso fora dos limites da memória**: Exibe um erro caso o PC ou o operando estejam fora do intervalo de 0 a 255.
- **Instruções desconhecidas**: Interrompe a execução e exibe uma mensagem caso uma instrução inválida seja encontrada.

3. QUESTÕES

3.1 Justificar o tamanho dos programas (em número de instruções) O tamanho dos programas depende da simplicidade do conjunto de instruções do Cesar. A ausência de instruções complexas, como múltiplos ou divisão, exige que táticas mais elaboradas sejam implementadas com sequências maiores de instruções básicas. Isso pode aumentar o tamanho dos programas para tarefas mais complexas.

3.2 Comentar sobre facilidades/dificuldades de programação **Facilidades**:

- A estrutura simples do Cesar permite compreender facilmente o funcionamento de cada instrução.
- Programas podem ser desenvolvidos com foco em conceitos fundamentais de arquitetura.

Dificuldades:

- A limitação no tamanho da memória e no conjunto de instruções pode tornar certas tarefas mais longas e complexas de implementar.
- A ausência de suporte a operações complexas requer implementações manuais, o que aumenta a propensão a erros.

3.3 Comparar desempenho em ciclos de relógio, considerando: O desempenho em ciclos de relógio depende da implementação de cada instrução:

- **Instruções de memória (LDA, STA)**: Mais lentas, pois envolvem acesso à memória principal.
- **Instruções aritméticas (ADD, SUB)**: Geralmente mais rápidas, pois são executadas diretamente nos registradores.
- **Instruções de controle (JMP, JZ, JN)**: Dependem de condições e alteram o fluxo de execução, mas são relativamente eficientes.

3.4 Ciclo para instruções lógico/aritméticas As instruções lógico/aritméticas do César, como ADD e SUB, geralmente requerem um ciclo para busca, um para decodificação e outro para execução. Assim, essas operações consomem tipicamente 3 ciclos de relógio.

4. CONCLUSÃO

O simulador do processador César é uma ferramenta educativa valiosa para explorar conceitos fundamentais de arquitetura de computadores. Embora limitado em termos de conjunto de instruções e memória, ele oferece uma experiência didática rica. A simplicidade de sua estrutura facilita o aprendizado, mas também impõe desafios ao programador, especialmente em tarefas mais complexas. Melhorias futuras podem incluir a implementação de funcionalidades de depuração mais avançadas e a expansão do conjunto de instruções para ampliar suas capacidades.