UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Bacharelado em Ciência da Computação

Gabriel Victor da Silva
Lucas Apolonio de Amorim
Pedro Lucas Medeiros do Nascimento

Organização de Computadores:

NATAL
2025

Gabriel Victor da Silva
Lucas Apolonio de Amorim
Pedro Lucas Medeiros do Nascimento

Organização de Computadores:

Relatório técnico apresentado como avaliação da disciplina (Organização de Computadores) ministrada pelo professor Dr. Marcio Edurado Kreutz para o curso de Ciência da Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte – Campus Central de Natal.

NATAL
2025

Sumário

1 INTRODUÇÃO

A área de sistemas computacionais é marcada por um dinamismo ímpar, resultado direto do contínuo avanço da tecnologia da informação. Longe de ser um campo estático e monótono, a computação representa uma das forças motrizes mais significativas da economia moderna, sendo responsável por uma parcela considerável do produto interno bruto em países como os Estados Unidos. Esse setor, impulsionado pelo ritmo acelerado previsto pela Lei de Moore, favorece a inovação em uma escala sem precedentes, na qual novas arquiteturas e computadores revolucionários são rapidamente superados por projetos ainda mais avançados.

Desde o surgimento da computação eletrônica na década de 1940, os progressos têm sido notáveis. Para ilustrar, se a indústria dos transportes tivesse evoluído na mesma velocidade da computação, hoje seria possível atravessar o oceano Atlântico em um segundo, a um custo insignificante. Tal comparação evidencia o impacto profundo que os avanços computacionais exercem sobre a sociedade, afetando não apenas o cotidiano das pessoas, mas também a forma como o conhecimento científico é produzido.

Atualmente, fala-se em uma "terceira revolução"da civilização: a revolução da informação. Assim como as revoluções agrícola e industrial transformaram o mundo, a computação vem ampliando exponencialmente a capacidade intelectual humana. Isso tem promovido uma nova abordagem científica, com o surgimento da ciência computacional ao lado das abordagens teórica e experimental, viabilizando avanços em diversas áreas, como astronomia, biologia, química e física.

O potencial de transformação da computação cresce a cada nova redução significativa no custo do processamento. Aplicações que outrora pertenciam ao campo da ficção científica tornam-se, gradualmente, viáveis e acessíveis (PATTERSON; HENNESSY, 2013)

2 Decisões de Projeto

Nesta seção são descritas as principais escolhas arquiteturais adotadas para a implementação de um processador baseado no paradigma RISC, com suporte a pipeline de cinco estágios. O projeto segue os princípios de simplicidade, modularidade e eficiência, característicos deste tipo de arquitetura.

A abordagem adotada visou um equilíbrio entre complexidade de implementação, clareza estrutural e capacidade funcional suficiente para execução de programas simples com instruções aritméticas, lógicas, de controle e acesso à memória.

2.1 Organização Geral da Arquitetura

A arquitetura foi estruturada com base em um conjunto reduzido de instruções (RISC), facilitando a implementação das etapas de decodificação, controle e execução. A divisão em cinco estágios clássicos — *Instruction Fetch (IF)*, *Instruction Decode (ID)*, *Execute (EX)*, *Memory Access (MEM)* e *Write Back (WB)* — permitiu a implementação eficiente de um pipeline com paralelismo instrucional.

2.2 Tamanho da Instrução e Formatos

Todas as instruções possuem largura fixa de 32 bits, facilitando a lógica de busca e decodificação e contribuindo para a previsibilidade do pipeline. A escolha desse tamanho garante alinhamento natural na memória e espaço adequado para representar diferentes tipos de instruções com seus operandos.

Foram definidos três formatos de instrução:

Tipo R (Registrador-Registrador) Usado para operações aritméticas e lógicas entre registradores. Esse tipo inclui três registradores (Rs, Rt, Rd), cada um representado por 4 bits. O opcode ocupa 6 bits, e os 14 bits restantes são reservados (zeros ou extensão futura). A escolha por três operandos explícitos permite instruções do tipo `ADD R1, R2, R3` com independência entre leitura e escrita.

Tipo I (Imediato) Usado para operações que envolvem constantes ou endereçamento imediato (como carga e armazenamento). Contém dois registradores (Rs, Rt), um campo de imediato de 18 bits (sinalizado) e 6 bits para o opcode. O tamanho do imediato permite acesso a um espaço de endereçamento significativo ou uso de valores constantes diretamente embutidos na instrução.

Tipo J (Salto) Projetado para instruções de controle de fluxo, como desvios e saltos incondicionais ou condicionais. Com 6 bits de opcode e 26 bits para o endereço de destino,

essa configuração permite uma faixa ampla de endereçamento, o que é útil em programas maiores, mesmo com instruções alinhadas a 4 bytes.

2.3 Opcode e Espaço de Instruções

O campo de opcode foi definido com 6 bits, permitindo até 64 instruções distintas. Apesar do conjunto de instruções implementado ser reduzido (aproximadamente 15), essa escolha oferece espaço para futuras expansões e experimentos com instruções complexas ou especializadas sem necessidade de reformulação no formato.

2.4 Racional por Trás das Escolhas

As decisões acima foram tomadas levando em consideração a viabilidade de implementação em SystemC, a clareza do projeto educacional e a escalabilidade futura. O uso de campos de tamanho padronizado (como 4 bits por registrador e 6 bits de opcode) contribui para modularidade no hardware e simplificação nos módulos de controle e decodificação. O projeto prioriza a clareza e eficiência, evitando complexidade desnecessária como modos de endereçamento indiretos ou instruções multifunção.

2.5 Subsistemas Fundamentais

O banco de registradores foi projetado com 16 posições de 32 bits cada, representadas por 4 bits nos campos Rs, Rt e Rd das instruções. Esse tamanho foi escolhido como um compromisso entre simplicidade de implementação e capacidade de armazenamento temporário para operandos. Entre os registradores, um possui uma função especial: o registrador \$0, cujo conteúdo é fixo em zero e não pode ser alterado — útil como constante em operações.

A memória do sistema é dividida em duas unidades fisicamente separadas: uma memória de instruções e uma memória de dados. Cada unidade possui capacidade de armazenamento para 256 palavras de 32 bits, com acesso direto e alinhamento por palavra. O espaço de endereçamento de 8 bits permite referenciar até 256 posições distintas, o que é suficiente para aplicações de pequeno porte e reduz a complexidade da lógica de controle. Por causa disso, e dado que os dados são todos de 32 bits, é necessário dividir o endereço por 4 para referenciar a posição correta. Essa separação segue o modelo Harvard e favorece a execução em pipeline, permitindo que as instruções sejam buscadas enquanto dados são acessados ou gravados, sem contenção de memória.

A comunicação entre os subsistemas ocorre por meio de três barramentos distintos: o barramento de dados (32 bits), que interliga registradores, memória e ULA; o barramento de endereços (32 bits), que especifica posições de leitura e escrita nas memórias; e o barramento de controle, que transporta sinais de habilitação, escrita, leitura, seleção de

operação e outros comandos essenciais à coordenação do funcionamento dos módulos. O uso de barramentos separados facilita o paralelismo entre os estágios e evita colisões de sinal.

2.6 Organização do Pipeline

A execução das instruções é dividida em cinco estágios:

1. Busca de Instrução (IF)
2. Decodificação (ID)
3. Execução (EX)
4. Acesso à Memória (MEM)
5. Escrita no Registrador (WB)

2.7 Conjunto de Instruções

O conjunto de instruções (ISA) foi desenvolvido para atender às necessidades fundamentais da computação, mantendo o princípio de simplicidade. Estão incluídas operações aritméticas e lógicas como `ADD`, `SUB`, `AND`, `OR`, `XOR`, além de comparações (`CMP`), transferências de dados (`LD`, `ST`), operações com imediatos (`ADDI`, `SUBI`) e instruções de controle de fluxo, como saltos incondicionais (`J`) e condicionais (`JZ`, `JN`). Essas instruções fornecem a base para a execução de algoritmos básicos, como ordenação, soma de vetores e controle de loops.

Formatos das Instruções

As instruções foram classificadas em três formatos principais: Tipo R, Tipo I e Tipo J, conforme a operação a ser executada.

- **Tipo R** (Registrador): utilizado para operações aritméticas e lógicas entre registradores.

  `R: <opcode[31:26]> <rs[25:22]> <rt[21:18]> <rd[17:14]> <free[13:0]>`

  - AND 000011
  - OR 000010
  - XOR 000001
  - CMP 000100
  - ADD 000111

  - SUB 001000

- **Tipo I** (Imediato): utilizado para operações entre registrador e valor imediato, além de operações de carga e armazenamento de dados.

  `I: <opcode[31:26]> <rs[25:22]> <rd[21:18]> <immediate[17:0]>`

  - NOT 100000
  - LD 100110
  - ST 101001
  - ANDI 100011
  - ORI 100010
  - XORI 100001
  - CMPI 100100
  - ADDI 100111
  - SUBI 101000

- **Tipo J** (Jump): utilizado para controle de fluxo com saltos baseados em condição ou incondicionais.

  `J: <opcode[31:26]> <immediate[25:0]>`

  - J 111111
  - JZ 110000
  - JN 111000

3 Implementação

3.1 Diagrama de Blocos

O diagrama de blocos da arquitetura representa graficamente a interconexão entre os principais módulos da Parte Operativa (PO) e a Parte de Controle (PC). Ele evidencia a separação entre os estágios do pipeline e os fluxos de dados, endereços e sinais de controle ao longo do processador.

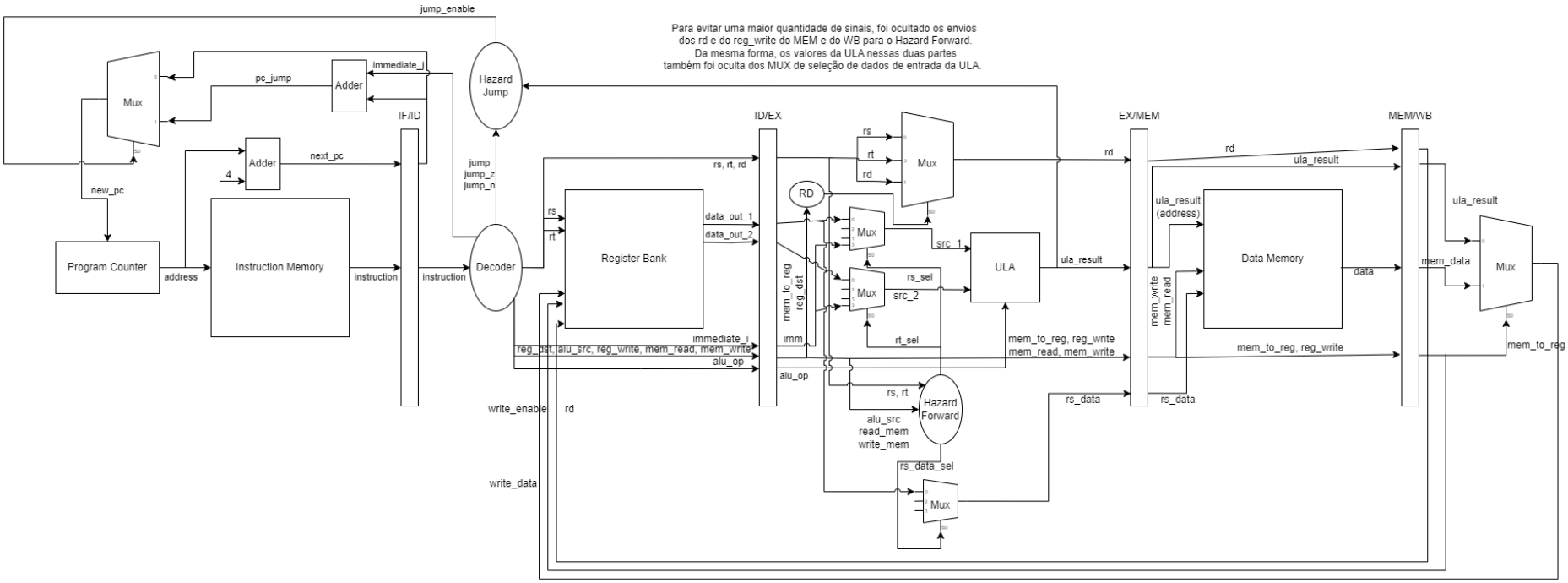

Figura 1: Diagrama de blocos do microprocessador.

Entre os blocos representados, destacam-se:

- **Memória de Instruções**: responsável pela entrega da instrução atual com base no valor do contador de programa (PC);
- **Banco de Registradores**: acessado nos estágios de decodificação (ID) e escrita (WB), permite leitura e escrita simultânea de registradores;
- **ULA (Unidade Lógica e Aritmética)**: executa as operações aritméticas e lógicas no estágio de execução (EX), utilizando operandos vindos dos registradores ou do imediato;
- **Unidade de Controle**: gera sinais que determinam o fluxo de dados, a seleção de caminhos nos multiplexadores e o comportamento da ULA, com base no campo de opcode;
- **Memória de Dados**: acessada no estágio MEM para operações de carga e armazenamento;
- **MUXes**: utilizados em diversos pontos do caminho de dados, para selecionar entre diferentes fontes de entrada (por exemplo, entre registrador e imediato, ou entre endereço sequencial e salto).

O diagrama também mostra a estrutura dos cinco estágios do pipeline (IF, ID, EX, MEM, WB), conectados por registradores intermediários que armazenam os resultados parciais entre os ciclos de clock. Os barramentos de controle, dados e endereços interligam esses módulos, permitindo a comunicação sincronizada de acordo com os sinais gerados pela unidade de controle.

Essa representação visual serviu como referência principal durante a integração e simulação do processador, garantindo que os caminhos de dados e controle estivessem corretos e que as dependências entre os estágios fossem tratadas adequadamente.

3.2 Etapas do Desenvolvimento

O processo de implementação foi conduzido de forma modular, com o desenvolvimento individual de cada componente funcional do processador. Foram construídos os seguintes blocos básicos: contador de programa, banco de registradores, memória de dados e instruções, multiplexadores genéricos com diferentes números de entradas, unidade lógica e aritmética (ULA), a unidade de decodificação e de controle, o Hazard Foward e o Hazard Control.

Com os blocos básicos concluídos, foram desenvolvidos os módulos correspondentes às etapas do pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX),

Memory Access (MEM) e Write Back (WB). Cada estágio foi construído interligando os blocos necessários à sua função específica dentro do ciclo de execução da instrução, incluindo a integração com a unidade de controle responsável pela coordenação do fluxo operacional.

Posteriormente, os estágios foram integrados progressivamente na construção do módulo principal da arquitetura. Esse módulo representa a parte operativa do processador, responsável por conectar os blocos funcionais por meio dos barramentos de dados, endereços e controle, estabelecendo o caminho completo da execução em pipeline.

A abordagem modular adotada facilitou a organização do projeto e permitiu uma evolução incremental, promovendo clareza estrutural e flexibilidade durante o desenvolvimento.

4 Teste Unitário

Antes da integração completa do processador, cada componente desenvolvido foi submetido a testes unitários com o uso de *testbenches* específicos. Essa etapa foi essencial para garantir que todos os blocos funcionassem conforme o esperado individualmente.

Foram realizados testes em diversos módulos fundamentais da arquitetura, incluindo os registradores individuais e o banco de registradores, as memórias de dados e de instruções, os multiplexadores com diferentes quantidades de entradas, a Unidade Lógica e Aritmética (ULA), o somador e a unidade de controle. Cada componente foi avaliado de forma isolada com diferentes estímulos de entrada, assegurando o correto funcionamento das operações conforme especificado no projeto.

Embora tenhamos implementado testes unitários para cada componente, a fim de simplificação, apresentamos abaixo apenas um exemplo do teste desenvolvido para a ULA.

Os demais testes podem ser encontrados no repositório do projeto, disponível em: https://github.com/lucasaamorim/chip-pipeline

4.1 Testes Unitários da ULA

Para validar o funcionamento da Unidade Lógica e Aritmética (ULA), foi desenvolvido um testbench em SystemC, onde diferentes operações foram simuladas com combinações variadas de operandos.

As operações testadas incluíram: soma (ADD), subtração (SUB), operação lógica AND e comparação (CMP).

Código de Teste (Resumo)

Listing 1: Trecho do testbench da ULA

```
sc_uint<32> a, b;

    // AND (op=0)
    func.write(0);
    a = 0xAAAAAAAA; b = 0x55555555;
    in1.write(a); in2.write(b); wait(10, SC_NS);
    print_test(0, a, b, out.read());

    // OR (op=1)
    func.write(1);
    a = 0xAAAAAAAA; b = 0x55555555;
    in1.write(a); in2.write(b); wait(10, SC_NS);
    print_test(1, a, b, out.read());
```

```
// XOR (op=2)
func.write(2);
a = 0xAAAAAAAA; b = 0x55555555;
in1.write(a); in2.write(b); wait(10, SC_NS);
print_test(2, a, b, out.read());

// GREATER_THAN (op=3)
func.write(3);
a = 100; b = 50;
in1.write(a); in2.write(b); wait(10, SC_NS);
print_test(3, a, b, out.read());
```

Resultados Obtidos

```
------------------------------------------------
Time: 14 ns | Operation: AND (0)
Inputs: A = 2863311530 (0x0aaaaaaaa), B = 1431655765 (0x055555555)
Result: 0 (0x000000000)
------------------------------------------------
Time: 24 ns | Operation: OR (1)
Inputs: A = 2863311530 (0x0aaaaaaaa), B = 1431655765 (0x055555555)
Result: 4294967295 (0x0ffffffff)
------------------------------------------------
Time: 34 ns | Operation: XOR (2)
Inputs: A = 2863311530 (0x0aaaaaaaa), B = 1431655765 (0x055555555)
Result: 4294967295 (0x0ffffffff)
------------------------------------------------
Time: 44 ns | Operation: GREATER_THAN (3)
Inputs: A = 100 (0x000000064), B = 50 (0x000000032)
Result: 1 (0x000000001)
------------------------------------------------
Time: 54 ns | Operation: GREATER_THAN (3)
Inputs: A = 50 (0x000000032), B = 100 (0x000000064)
Result: 0 (0x000000000)
------------------------------------------------
Time: 64 ns | Operation: ADD (4)
Inputs: A = 50 (0x000000032), B = 25 (0x000000019)
Result: 75 (0x00000004b)
------------------------------------------------
Time: 74 ns | Operation: SUB (5)
Inputs: A = 50 (0x000000032), B = 25 (0x000000019)
Result: 25 (0x000000019)
------------------------------------------------
Time: 84 ns | Operation: NOT (6)
Inputs: A = 2863311530 (0x0aaaaaaaa), B = 0 (0x000000000)
Result: 1431655765 (0x055555555)
```

Tabela 1: Resultados dos testes aplicados à ULA.

A operação `CMP` retorna 0 quando os operandos são iguais. Nos casos de desigualdade, o resultado é diferente de zero (tipicamente 1 ou -1, dependendo da implementação), o que pode ser utilizado por instruções de desvio condicional.

Esses testes demonstram que a ULA opera corretamente nas instruções essenciais do conjunto, com resposta adequada dos resultados, o que garante sua confiabilidade para integração no processador.

5 Desvantagens

Uma das limitações do projeto desenvolvido está relacionada à correção de hazards envolvendo instruções de `Load`. Embora seja comum utilizar a abordagem de inserção de uma bolha no pipeline para lidar com esse tipo de dependência, optamos por uma solução alternativa: primeiro realizamos a leitura da memória e, em seguida, enviamos o dado para a ULA. Essa abordagem evita a inserção de bolhas na execução, mas exige que o estágio de EX aguarde a conclusão total do estágio MEM, o que implica em um ciclo de clock mais longo.

Por esse motivo, consideramos essa solução uma desvantagem, já que, em sistemas de hardware, um processador com ciclos mais longos pode ter desempenho médio inferior em comparação com outro que apenas adiciona bolhas à execução, mas mantém um clock mais rápido.

6 Conclusão

O desenvolvimento deste microprocessador com arquitetura pipeline permitiu explorar diversos conceitos fundamentais de arquitetura de computadores, como a divisão em estágios, o controle de fluxo e o tratamento de dependências de dados. Ao longo do projeto, foi possível observar o aumento da taxa de instruções por ciclo através do pipeline, mas também as dificuldades impostas por ele. A decisão de não tratar o hazard decorrido por dependência de dados em funções `Load`, torna o microprocessador passível de erros de dependência, não funcionando corretamente para alguns conjuntos de instruções.

Em síntese, esse trabalho atingiu seus objetivos educacionais, permitindo uma compreensão mais aprofundada do funcionamento de um processador em pipeline, a sincronização dos elementos com base no clock e o tratamento de hazards.

Referências

[1] PATTERSON, David A.; HENNESSY, John L. *Computer Organization and Design: The Hardware/Software Interface*. 5. ed. Burlington: Morgan Kaufmann, 2013.