Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
Instituto Federal Catarinense
Campus Videira

CRISTIANO RODRIGUES PIROLLI

# TUCANO-8

Videira
2025

CRISTIANO RODRIGUES PIROLLI

# TUCANO-8

Projeto de Arquietura de Computadores apresentado ao Curso de graduação em Ciência da Computação do Instituto Federal Catarinense – Campus Videira.
Orientador: Angelita Rettore de Araujo Zanella

Videira
2025

# 1 INTRODUÇÃO

Foi proposto o projeto "Tucano-8", visando o desenvolvimento de uma arquitetura de processador de 8 bits customizada. O objetivo principal deste trabalho, conforme delineado na tarefa de projeto, foi consolidar os conhecimentos em arquitetura de computadores através da análise de requisitos de um sistema embarcado, o projeto de um banco de registradores adequado, a definição de um conjunto de instruções (ISA) eficiente e a documentação justificada de todas as escolhas de design.

A metodologia adotada partiu da análise de quatro aplicações práticas para o sistema agrícola, derivando-se delas os requisitos de hardware e software. Com base nesses requisitos, foram projetados o banco de registradores e a ISA. A implementação e validação da arquitetura foram realizadas utilizando a ferramenta de simulação Logisim, culminando em testes que verificaram a funcionalidade dos principais componentes e da lógica de controle. Este relatório documenta todas as fases do projeto, as decisões tomadas e os resultados obtidos.

# 2 ANÁLISE DAS APLICAÇÕES E REQUISITOS DO SISTEMA AGRÍCOLA INTELIGENTE

Para guiar o projeto do Tucano-8, foram analisadas quatro aplicações-chave:

**Aplicação 1: Irrigação Inteligente:** Monitora umidade (Sensor 0) e temperatura (Sensor 1). Ativa/desativa a irrigação (Atuador 1) se (Umidade < 30) E (Temperatura > 25). Requer monitoramento contínuo (loop) e lógica AND.

**Aplicação 2: Controle de Ventilação:** Monitora temperatura (Sensor 1). Ativa/desativa a ventilação (Atuador 2) se Temperatura >= 20. Requer lógica IF-ELSE.

**Aplicação 3: Nível de Água:** Monitora nível (Sensor 2) e estado da irrigação (Atuador 1). Ativa a bomba (Atuador 3) se (Nível < 10) E (Irrigação_Desativada). Requer lógica AND.

**Aplicação 4: Controle de Vento:** Monitora vento (Sensor 3). Fecha janelas (Atuador 0) se houver vento. Requer lógica IF e potencialmente sub-rotinas.

Sensores e Atuadores Identificados

**Sensores:** Umidade (S0), Temperatura (S1), Nível (S2), Vento (S3).
**Atuadores:** Janelas (A0), Irrigação (A1), Ventilação (A2), Bomba (A3).

# 3 BANCO DE REGISTRADORES

Baseado nos requisitos e visando a simplicidade, o banco de registradores do Tucano-8 foi projetado da seguinte forma:

**NÚMERO DE REGISTRADORES:** 4 (R0, R1, R2, R3).

*Justificativa:* Um número pequeno reduz a complexidade do hardware (decodificação de 2 bits para Rd/Rs) e é suficiente para as aplicações-alvo, que não exigem muitos cálculos complexos simultâneos. Isso promove o uso eficiente da memória para variáveis persistentes, enquanto os registradores servem para operações imediatas.

**TAMANHO DOS REGISTRADORES:** 8 bits.

*Justificativa:* Compatível com a largura de dados dos sensores e atuadores, e mantém a ULA e os barramentos de 8 bits, resultando em um design mais simples e econômico.

**ORGANIZAÇÃO:** Propósito geral.

*Justificativa:* Oferece máxima flexibilidade ao programador, simplificando a alocação de dados e a escrita de código, pois qualquer registrador pode ser usado em qualquer instrução que os aceite.

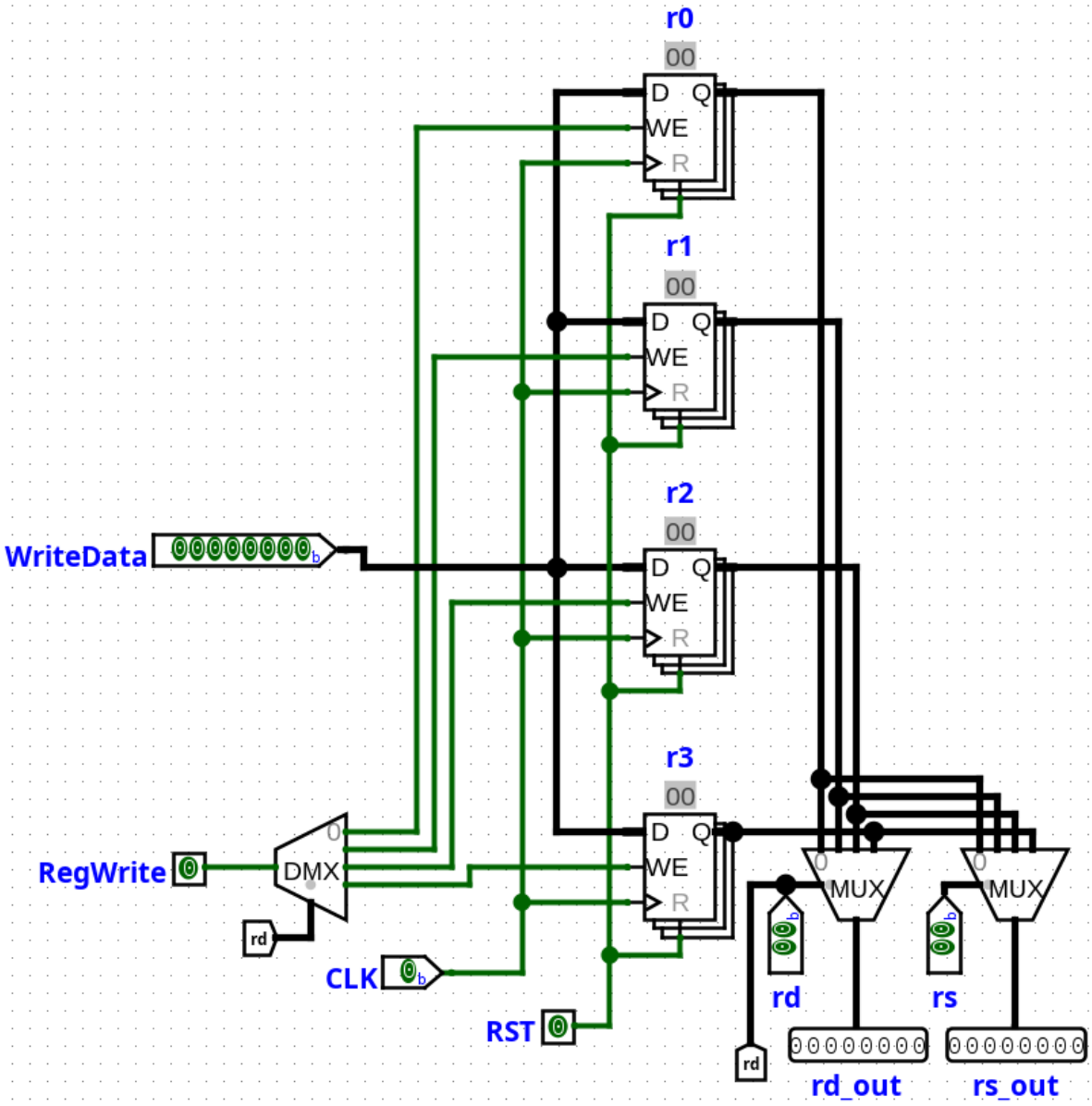



Figura 1 – Banco de Registradores
*Fonte: Elaboração do autor, 2025.*

4 CONJUNTO DE INSTRUÇÕES (ISA Tucano-8)

A ISA foi projetada para ser minimalista, mas funcional para as aplicações previstas.

Formatos de Instrução (8 bits):
**Tipo R** (Registrador): Opcode(4) | Rd(2) | Rs(2) - Para operações entre registradores.

**Tipo I/J** (Imediato/Jump/Memória): Opcode(4) | Endereço/Imediato(4) - Para saltos e acesso à memória (com Rd/Rs implícitos ou não usados).

**TABELA DE INSTRUÇÕES:**

| Opcode | Mnemônico | Operação | Descrição |
|---|---|---|---|
| **Operações Aritméticas e Lógicas (Tipo R)** | | | |
| 0000 | ADD | Rd = Rd + Rs | Soma o conteúdo de dois registradores. |
| 0001 | SUB | Rd = Rd - Rs | Subtrai o conteúdo de Rs de Rd. |
| 0010 | AND | Rd = Rd & Rs | Realiza a operação AND bit a bit. |
| 0011 | OR | Rd = Rd OR Rs | Realiza a operação OU bit a bit. |
| 0100 | NOT | Rd = ~Rd | Nega o valor de Rd. |
| 0101 | SHL | Rd = Rd << 1 | Desloca os bits de Rd para a esquerda. |
| 0110 | SHR | Rd = Rd >> 1 | Desloca os bits de Rd para a direita. |
| 0111 | GTH | Rd = Rd > Rs | Se Rd > Rs então Rd = 1, senão Rd =0 |
| 1000 | STH | Rd = Rd < Rs | Se Rd < Rs então Rd = 1, senão Rd =0 |
| 1001 | EQU | Rd = Rd == Rs | Se Rd igual a Rs então Rd = 1, senão Rd =0 |
| 1101 | MV | Rd = Rs | O registrador Rd recebe o conteúdo do registrador Rs |
| **Operações Imediatas (Tipo I)** | | | |
| 1010 | JMP | PC = Endereço | Salta para o endereço especificado. |
| 1011 | BEQ | Se (Flag == 1) então PC = Endereço | Salta se o resultado de uma comparação for igual. |
| 1100 | BNE | Se (Flag == 0) então PC = Endereço | Salta se o resultado de uma comparação for diferente. |
| **Operações de Memória** | | | |
| 1110 | READ | `Rd = Mem[Endereço]` | Carrega o valor armazenado em uma posição de memória e o transfere para um registrador. |
| 1111 | WRITE | `Mem[Endereço] = Rs` | Armazena o conteúdo de um registrador em uma posição especifica da memória. |

| **Tipo R** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **opcode** | | | | **Rd** | | **Rs** | |

| **Tipo I** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **opcode** | | | | **Rd** | | **Imediato** | |

| **Tipo J** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **opcode** | | | | **Endereço** | | | |

Figura 2 – Tabela de Instruções
Fonte: Angelita Rettore de Araujo Zanella, 2025.

5 CAMINHO DE DADOS

# **Tipo R:** Opcode(4) | Rd(2) | Rs(2)

**Exemplo:** Vamos usar **ADD R0, R1;**

Opcode ADD = 0000
Rd = R0 = 00
Rs = R1 = 01

Instrução Binária: 0000 0001
Instrução Hexadecimal: 01

**Fetch (busca):**

O PC (Program Counter) envia seu endereço atual (ex: 00h) para a Memória de Instruções (ROM);

A ROM retorna a instrução no endereço 0x00, que é 01;

Esta instrução 01 é carregada;

Simultaneamente, o PC é incrementado para 01 (PC+1), preparando-se para a próxima busca (assumindo que não haverá desvio).

**Decode (decodificação)**

A instrução 01 (0000 0001) é analisada;

Opcode (0000) vai para a UC (Unidade de Controle);

Os bits Rd (00) e Rs (01) vão para o Banco de Registradores como endereços de leitura (Rd como endereço de escrita também);

A UC reconhece o Opcode 0 e gera os sinais:
RegWrite = 1 (habilita escrita no registrador Rd)
ALUSrc = 0 (seleciona Rs como segunda entrada da ULA)
MemToReg = 0 (seleciona o resultado da ULA para escrita no registrador)

**Execute (execução)**

O Banco de Registradores usa os endereços 00 e 01:
Lê o valor de R0 e o envia para a Entrada A da ULA;
Lê o valor de R1;

Um MUX (controlado por ALUSrc = 0 ) seleciona o valor de R1 e o envia para a Entrada B da ULA;
A ULA, com Opcode = 0000, calcula: R0 + R1;
O resultado da soma sai da ULA;

#### Memory (acesso à memória)

Como MemRead = 0 e MemWrite = 0, nada significativo acontece aqui, o resultado da soma, apenas segue o seu caminho.

#### Write Back (fase de escrita)

O resultado chega a outro MUX controlado por MemToReg;
Este MUX seleciona o resultado e o envia para a porta Write_Data do Banco de Registradores;
O endereço do Rd é enviado para a porta;
O sinal RegWrite = 1, habilita a escrita;
No próximo sinal de clock, o resultado é escrito em Rd.

## **Tipo J:** Opcode(4) | Endereço(4)

**Exemplo:** Vamos usar Read E3;

#### Fetch

PC busca E3;

#### Decode

UC vê Opcode E (1110), ativa MemRead = 1, MemToReg = 1, RegWrite = 1. O endereço 03 é extrtaído.

#### Execute

O endereço vai para a memória de dados (RAM). (ULA não pode ser usada);

#### Memory

A RAM lê o dado no endereço 03;

#### Write Back

O dado lido da RAM passa pelo MUX  (MemToReg = 1) e é escrito no registrador, pois RegWrite = 1

# **Tipo I:** Opcode(4) | Rd(2) | Rs(2)

**Exemplo:** Vamos usar BNE C7;

#### Fetch

PC busca C7 (PC = x)

#### Decode

UC vê Opcode C (1011), ativa BNE_Enable = 1, ALUSrc = 1. O endereço 07 é extraído e extendido.

#### Execute

O PC já incrementado (x+1) vai para a entrada A da ULA. O Offset (07) vai para a entrada B (ALUSrc = 1). A ULA soma e calcula o Endereço_Destino.

#### Salto

A lógica de desvio vê BNE_Enable = 1, e Zero_Flag = 0, decide saltar.

#### Atualiza PC

O MUX do PC seleciona o Endereço_Destino calculado pela ULA. O PC é atualizado para este novo valor.

## 5 UNIDADE LÓGICO-ARITMÉTICA (ULA)

A Unidade Lógico-Aritmética (ULA), é um componente fundamental em qualquer processador. Ela atua como o "centro de cálculos" do processador, sendo responsável por realizar todas as operações aritméticas (como ADD (soma) e SUB (subtração)) e lógicas (como AND, OR, NOT) definidas pelo conjunto de instruções. No Tucano-8, a ULA foi projetada para executar as 7

operações básicas definidas na ISA, operando sobre dados de 8 bits e gerando a Zero_Flag, essencial para as instruções de desvio condicional.

**Componentes Utilizados:**

Para implementar as funcionalidades, foram selecionados os seguintes componentes básicos, disponíveis no Logisim Evolution:

1. Somador 8 bits (Adder): Utilizado para a operação ADD. Recebe duas entradas de 8 bits e gera uma saída de 8 bits.
2. Subtrator 8 bits (Subtractor): Utilizado para a operação SUB. Recebe duas entradas de 8 bits e gera uma saída de 8 bits.
3. Portas Lógicas 8 bits:
    - Porta AND: Para a operação AND bit a bit.
    - Porta OR: Para a operação OR bit a bit.
    - Porta NOT: Para a operação NOT bit a bit.
4. Deslocadores 8 bits (Shifter):
    - Um configurado para Shift Left Logical (SHL), deslocando 1 bit para a esquerda.
    - Um configurado para Shift Right Logical (SHR), deslocando 1 bit para a direita.
5. Multiplexador 8-para-1 (MUX): Um MUX de 8 bits com 16 entradas e 1 saída. Ele é o componente central que seleciona qual das operações será enviada para a saída final da ULA. A seleção é feita por um sinal de controle.
6. Flip-Flop Tipo D: Utilizado para armazenar o resultado do Detector de Zero, constituindo a Zero_Flag. Sua escrita é controlada pelo sinal FlagWrite vindo da UC.

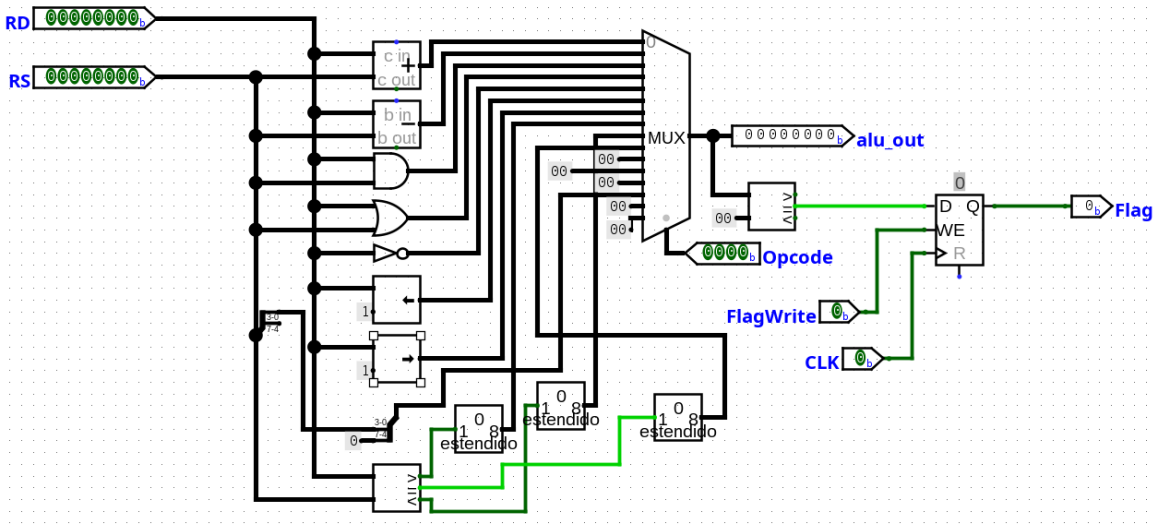



Figura 3 – Unidade Lógico-Aritmética.
*Fonte: Elaboração do autor, 2025.*

**Funcionamento**

A ULA Tucano-8 opera de forma síncrona com o restante do processador, mas seu núcleo é combinacional. Em cada ciclo de execução que requer uma operação lógica ou aritmética, a UC envia o Opcode apropriado. As entradas A e B, provenientes do banco de registradores ou do extensor de sinal (via ALUSrc), são processadas simultaneamente por todos os blocos funcionais (Somador, Subtrator, etc.). O MUX principal, controlado pelo Opcode, seleciona apenas um desses resultados para ser a saída (alu_out). Este resultado é então disponibilizado para ser escrito de volta no banco de registradores ou usado em outros cálculos. Paralelamente, o alu_out é verificado quanto a ser zero, e essa informação, se habilitada pelo FlagWrite, é armazenada na Flag para uso posterior por instruções de desvio.

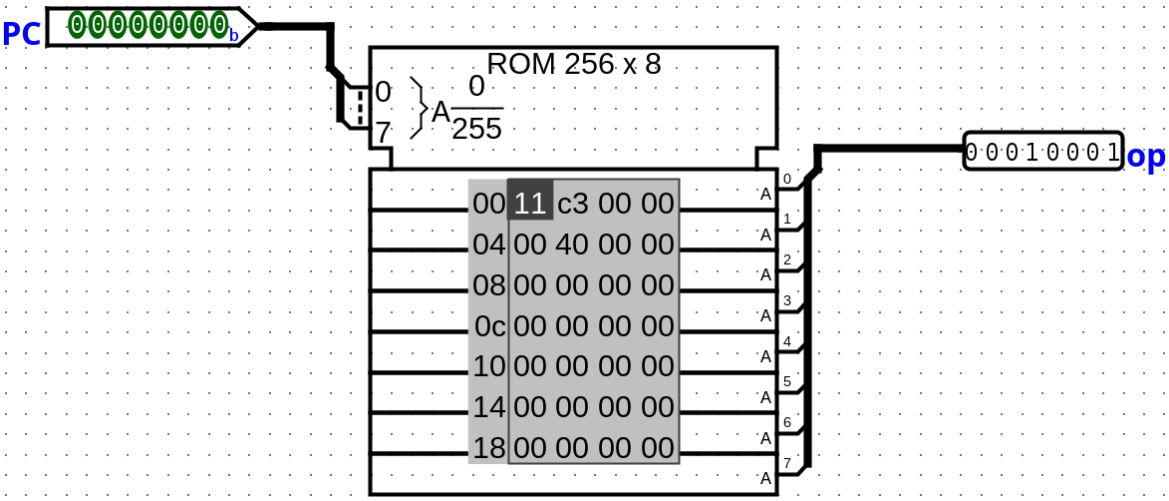
# 5 IMPLEMENTAÇÃO DA MEMÓRIA

A memória é um componente vital em qualquer sistema computacional, atuando como o repositório principal para as instruções que o processador executa e para os dados que ele manipula. No contexto do Tucano-8, a memória deve não apenas armazenar o programa de controle, mas também os dados lidos dos sensores, os comandos enviados aos atuadores e quaisquer variáveis temporárias necessárias. Esta seção detalha a definição, organização, implementação e teste do sistema de memória do Tucano-8.

### Escolha

A decisão de implementar memórias físicas maiores (256 bytes) do que o espaço atualmente endereçável (16 bytes) foi tomada estrategicamente. Ela visa permitir futuras expansões da ISA do Tucano-8. Versões futuras poderiam incluir instruções de 16 bits ou modos de endereçamento mais complexos (como endereçamento indireto via registrador) que permitiriam acessar todo o espaço de 256 bytes, sem a necessidade de modificar o hardware de memória base.

Figura 4 – Memória ROM

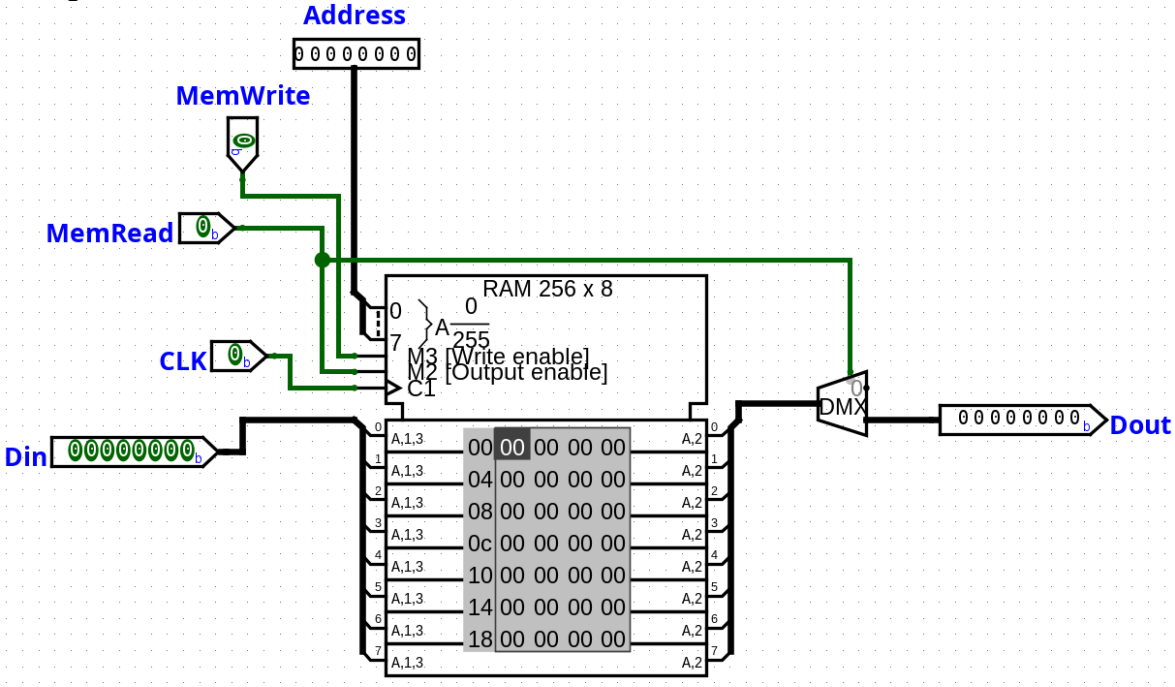



*fonte: Elaboração do autor, 2025.*

### *ROM*

A Memória de Instruções, implementada como uma ROM, é o componente responsável por armazenar o programa que o processador Tucano-8 executará. As instruções, codificadas no formato binário de 8 bits (representadas em hexadecimal para facilidade de carregamento), são lidas sequencialmente ou através de desvios, ditando as operações que o processador deve realizar. Por ser uma ROM, seu conteúdo é definido antes da execução e não é alterado pelo processador durante o funcionamento.

Figura 5 – Memória RAM




*fonte: Elaboração do autor, 2025.*

### **RAM**

A Memória de Dados, implementada como uma RAM, serve como o espaço de trabalho principal do Tucano-8 para armazenamento e recuperação de dados variáveis durante a execução do programa. Ela é utilizada para guardar os resultados de operações, valores lidos de sensores, comandos a serem enviados para atuadores e quaisquer outras informações temporárias que o programa necessite. Sendo uma RAM, seu conteúdo é volátil e pode ser lido e modificado pelo processador através das instruções READ e WRITE.

# 6 UNIDADE DE CONTROLE (UC)

A Unidade de Controle (UC) é o componente central do processador responsável pela interpretação das instruções e pela geração dos sinais de controle que orquestram as operações dos demais componentes do datapath (ULA, Banco de Registradores, Memória, PC). No Tucano-8, a UC recebe o Opcode de 4 bits da instrução atual e, com base nele, ativa ou desativa os sinais necessários para que a instrução seja executada corretamente.

### Tipo de Controle

O Tucano-8 utiliza uma Unidade de Controle Combinacional. Isso significa que os sinais de controle são gerados por um circuito lógico combinacional cujas saídas dependem exclusivamente das entradas atuais (Opcode). Esta abordagem foi escolhida pela sua simplicidade de projeto e implementação, adequada para um processador com um conjunto de instruções relativamente pequeno e fixo como o Tucano-8. Não há estados internos na UC; a cada Opcode, um conjunto fixo de sinais de controle é produzido.

ENTRADAS E SAÍDAS

Entrada principal da UC: Opcode;

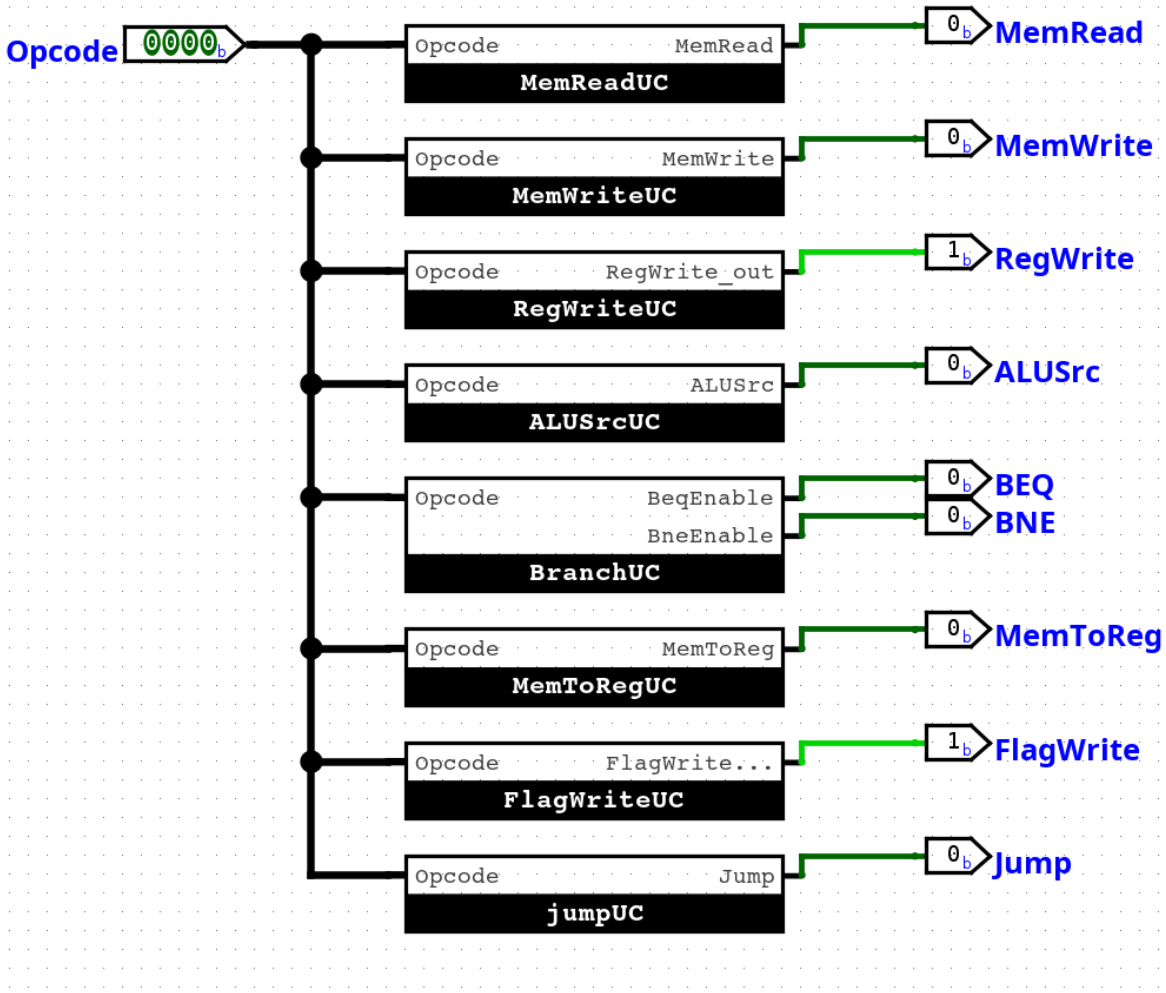
Saídas:

RegWrite (1 bit): Habilita a escrita no Banco de Registradores;
ALUSrc (1 bit): Seleciona a segunda entrada da ULA;
MemToReg (1 bit): Seleciona a origem do dado a ser escrito no registrador (ULA ou Memória de Dados);
MemRead (1 bit): Habilita a leitura da Memória de Dados;
MemWrite (1 bit): Habilita a escrita na Memória de Dados;
BEQ (1 bit): Indica que a instrução é um Salto do tipo BEQ;
BNE (1 bit): Indica que a instrução é um Salto do tipo BNE;
Jump (1 bit): Habilita o salto incondicional;
FlagWrite (1 bit): Habilita escrita na Zero_Flag;

### Lógica de Geração de Sinais

A UC foi implementada de forma modular, onde cada sinal de controle é gerado por um sub-circuito conceitual (...UC). Internamente, cada um desses sub-circuitos utiliza um Decodificador 4-para-16 que recebe o Opcode. As saídas do decodificador (cada uma correspondendo a um Opcode específico) são então combinadas usando Portas OR para gerar o sinal de controle final. Se um sinal de controle é ativado por apenas um Opcode, a saída correspondente do decodificador pode ser usada diretamente.

Figura 6 – Unidade De Controle




*fonte: Elaboração do autor, 2025.*

**Explicação do Funcionamento da UC**

A Unidade de Controle do Tucano-8 opera de forma puramente combinacional. Quando uma instrução é buscada da ROM e seu Opcode de 4 bits é apresentado à UC, este Opcode é imediatamente processado pela lógica interna.

Dentro da UC (de seus sub-blocos conceituais), um Decodificador 4-para-16 converte o Opcode de 4 bits em um sinal ativo em uma de suas 16 saídas. Cada uma dessas saídas representa unicamente uma das possíveis combinações de Opcode.

Para cada sinal de controle (RegWrite, ALUSrc, etc.), as saídas do Decodificador correspondentes às instruções que devem ativar aquele sinal são agrupadas através de Portas OR (ou usadas diretamente se apenas uma instrução ativar o

sinal). Por exemplo, para o sinal RegWrite, as saídas do decodificador para ADD, SUB, AND, OR, NOT, SHL, SHR e READ são conectadas a uma Porta OR. Se o Opcode de qualquer uma dessas instruções estiver presente, a saída da Porta OR para RegWrite se torna '1', ativando a escrita no banco de registradores.

Este processo ocorre simultaneamente para todos os sinais de controle, resultando em um conjunto de comandos que configuram o datapath para executar corretamente a instrução decodificada.

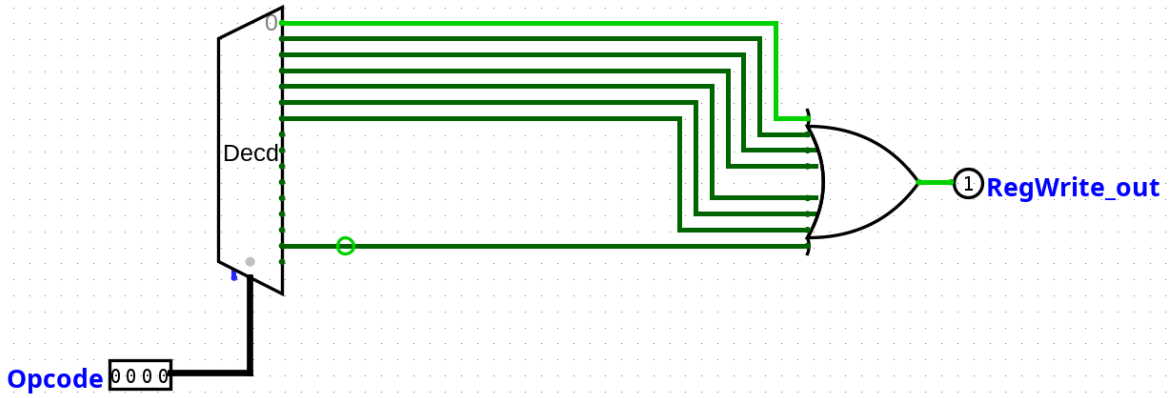



Figura 7 – Lógica para RegWrite

*fonte: Elaboração do autor, 2025.*

# 7 ERROS

## DESVIO CONDICIONAL

Um dos primeiros e mais significativos problemas foi o comportamento incorreto das instruções de desvio condicional BNE e BEQ. Observei que o processador parecia sempre seguir a lógica do BEQ, independentemente da instrução ou do resultado da operação anterior. A investigação revelou que a Zero_Flag estava permanentemente em nível lógico '1'. Isso ocorria porque não havia um mecanismo adequado para que a ULA calculasse o estado da flag (se o resultado da operação era zero ou não) e, crucialmente, para que esse estado fosse armazenado e atualizado condicionalmente.

## IMPLEMENTAÇÃO DA UNIDADE DE CONTROLE

Definir uma abordagem clara e eficiente para gerar os múltiplos sinais de controle necessários para orquestrar o datapath, a partir do Opcode da instrução. Houve uma consideração inicial em usar lógica customizada com portas AND/NOT individuais para cada Opcode que ativasse um sinal específico, contudo, algumas operações estava entrando em conflito e não passava valores ou dados corretamente.

Com isso, A UC foi conceitualizada em blocos (...UC), cada um responsável por um sinal de controle. Dentro de cada bloco, um Decodificador 4-para-16 recebe o Opcode. As saídas do decodificador correspondentes às instruções que ativam o sinal são conectadas a uma Porta OR. Se apenas uma instrução ativa o sinal, a saída do decodificador é usada diretamente. Esta abordagem provou ser significativamente mais limpa, fácil de verificar, depurar e manter.

**Link para o GITHUB do Projeto**

https://github.com/CristianoPirolli/RISC-V-processor

nele, consta esse mesmo arquivo, o arquivo com o projeto no logisim e alguns para carregar na ROM, com as instruções.