Universidade Federal do Cariri

Irlan Barros e Matheus Bezerra

ARQUITETURA DE COMPUTADORES

DESENVOLVIMENTO DO MICROPROCESSADOR BLDIM

Juazeiro do Norte-CE

2024

IRLAN BARROS E MATHEUS BEZERRA

ARQUITETURA DE COMPUTADORES

DESENVOLVIMENTO DO MICROPROCESSADOR BLDIM

Trabalho em dupla para obtenção de nota complementar para a cadeira de arquitetura de computadores, ministrada pelo prof. Ramon Nepomuceno.

Juazeiro do Norte-CE

2024

IRLAN BARROS E MATHEUS BEZERRA

ARQUITETURA DE COMPUTADORES

DESENVOLVIMENTO DO MICROPROCESSADOR BLDIM

EXAMINADOR

_____

Prof. Ramon Nepomuceno

SUMÁRIO

1. INTRODUÇÃO

Nos últimos anos, o avanço da tecnologia de microprocessadores tem desempenhado um papel crucial no desenvolvimento de sistemas embarcados, computadores pessoais, dispositivos móveis e uma variedade de outros dispositivos eletrônicos. Diante desse cenário, o projeto apresentado neste trabalho visa a criação de um microprocessador denominado BLDIM, utilizando a ferramenta de simulação de circuitos digitais Logisim.

A multiplicação de matrizes é uma operação fundamental em diversas áreas da ciência da computação, matemática aplicada e engenharia. No entanto, quando se trata de matrizes de grande porte, o desafio computacional aumenta significativamente devido à complexidade do algoritmo envolvido. Sendo assim, o projeto apresentado neste trabalho tem como principal objetivo a criação de um microprocessador denominado BLDIM, capaz de realizar a multiplicação de matrizes 4 por 4 de forma eficiente.

Utilizando a ferramenta de simulação de circuitos digitais Logisim, o BLDIM será projetado e implementado com o propósito específico de realizar operações de multiplicação de matrizes de tamanho 4 por 4. Essa escolha se deve à relevância dessa operação em diversas aplicações práticas, como processamento de imagens, simulações físicas e análise de dados.

Além da implementação dos circuitos lógicos necessários para a multiplicação de matrizes, este projeto também incluirá a definição de uma arquitetura de instruções customizada em linguagem Assembly, otimizada para realizar essa operação de forma eficiente e rápida. Adicionalmente, será desenvolvido um compilador em linguagem Python para traduzir programas escritos em Assembly para o conjunto de instruções do microprocessador BLDIM.

Ao longo deste documento, serão abordados os detalhes do projeto, desde a concepção da arquitetura do microprocessador até a implementação dos circuitos no Logisim. Serão discutidos os desafios encontrados durante o desenvolvimento, as decisões de design tomadas e os resultados obtidos em relação à eficiência e desempenho da multiplicação de matrizes. Por fim, serão apresentadas as conclusões e possíveis direções futuras para aprimoramento do BLDIM e suas aplicações.

2. METODOLOGIA

A metodologia adotada neste projeto baseia-se em uma abordagem integrada, combinando o uso da ferramenta de simulação de circuitos digitais Logisim, o desenvolvimento de um compilador em linguagem Python e a definição de um conjunto de instruções em Assembly. Cada uma dessas etapas desempenha um papel fundamental no desenvolvimento do microprocessador BLDIM e na realização eficiente da multiplicação de matrizes 4 por 4. Abaixo, detalhamos o porquê da escolha de cada uma dessas ferramentas:

1. Logisim:

A escolha do Logisim como ferramenta principal para a implementação do microprocessador BLDIM deve-se à sua versatilidade e facilidade de uso. O Logisim permite a construção de circuitos digitais de forma intuitiva, utilizando componentes lógicos como portas AND, OR e flip-flops para criar sistemas complexos. Além disso, sua interface gráfica facilita a visualização e depuração dos circuitos, tornando-o ideal para o desenvolvimento de projetos de microprocessadores.

2. Compilador em linguagem Python:

O desenvolvimento de um compilador em linguagem Python é essencial para traduzir programas escritos em Assembly para o conjunto de instruções do microprocessador BLDIM. Python foi escolhido devido à sua simplicidade, flexibilidade e ampla disponibilidade de bibliotecas para manipulação de strings e análise sintática. O compilador será responsável por converter as instruções Assembly em códigos de máquina compreensíveis pelo BLDIM, garantindo a execução correta dos programas desenvolvidos.

3. Instruções em Assembly:

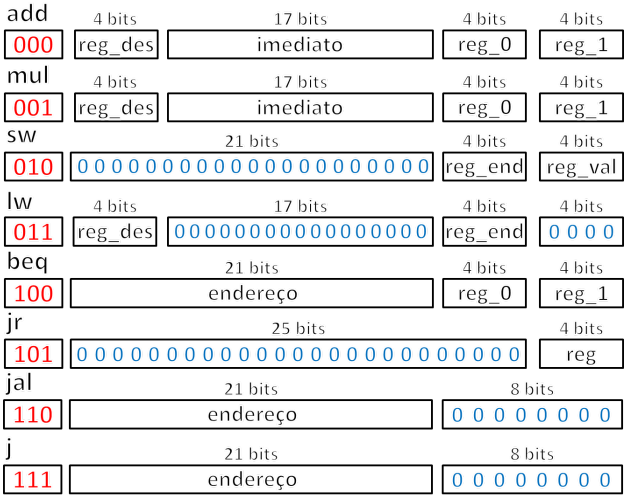
A definição de um conjunto de instruções em linguagem Assembly personalizada é fundamental para garantir o funcionamento eficiente do microprocessador BLDIM. As instruções serão projetadas especificamente para realizar operações de multiplicação de matrizes 4 por 4, otimizando o desempenho e minimizando o tempo de execução dos programas. A linguagem Assembly foi escolhida devido à sua proximidade com a linguagem de máquina e à sua capacidade de representar operações de baixo nível de forma concisa e eficiente.

Ao combinar essas ferramentas e abordagens, espera-se alcançar um microprocessador BLDIM eficaz e capaz de realizar a multiplicação de matrizes 4 por 4 de forma rápida e precisa. O uso do Logisim para implementação dos circuitos, do compilador em Python para tradução de programas e das instruções em Assembly para operações de baixo nível proporcionará uma solução completa e robusta para o problema proposto.

3. IMPLEMENTAÇÃO

Para desenvolver o processador BLDIM utilizando o Logisim e criando suas intruções em linguagem Assembly tivemos o auxílio do professor orientador Ramon Nepomuceno que disponibilizou em aula, como exemplo, o circuito do processador MIPS. Dessa forma, utilizamos o circuito disponibilizado como base para o desenvolvimento do BLDIM. Todo o processo de desenvolvimento foi feito passo a passo, fazendo cada parte de forma detalhista e cuidadosa.

3.0 Estrutura das Funções do BLDIM - Assembly



As instruções da linguagem BLDIM-Assembly foram definidas com 32 bits cada, sendo 8 funções no total, como mostrado ao lado.

Na imagem, vemos também a quantidade de bits destinados a cada componente das funções, onde reg_des = registrador de destino, reg_0 e reg_1 são registradoes utilizados na ULA, reg_end = registrador de endereço e reg_val = registrador de valor.

As funções aição (add) e multiplicação (mul) são compostas de dois registradores de operandos, reg_0 e reg_1, um imediato e um registrador de destino (reg_des). Como resultado da função, teremos o efeito:

$reg_des = $reg_0 + (*) $reg_1 + (*) imediato

A função sw acessa o endereço da memória de dados que corresponde ao valor em $reg_end e guarda nesse endereço o valor contido em $reg_val.

Semelhantemente, lw acessa o endereço da memória de dados correspondente ao valor em $reg_end e guarda o valor presente nesse endereço em $reg_des.

beq compara os vaores contidos em $reg_0 e $reg_1, produzindo um salto para o endereço indicado caso os valores sejam iguais.

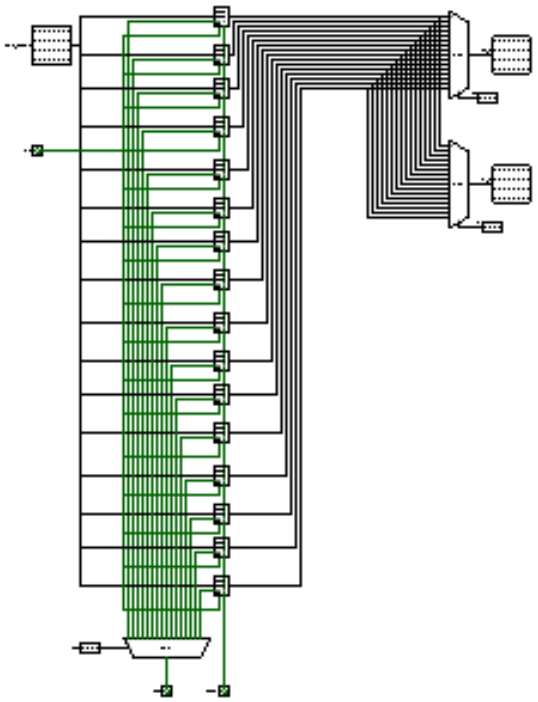
jr produz salto para o endereço correspondente ao valor em $reg;

jal salta para o endereço indicado e guarda em $ra o endereço seguinte da memória de instruções.

j simplesmente salta para o endereço indicado.

Com esse conjunto de funções, somos capazes de realizar o procedimento proposto de multiplicação de matrizes.

3.1 Banco de Registradores



O banco de registradores é uma parte vital de um microprocessador, responsável por armazenar temporariamente dados e resultados intermediários das operações realizadas pela unidade de processamento. Composto por registradores de propósito geral, ele serve para guardar operandos de operações aritméticas, resultados temporários, endereços de memória e outros dados importantes. Geralmente, microprocessadores modernos possuem dezenas de registradores, acessíveis diretamente pela unidade de controle e de processamento através de sinais específicos. Os registradores são utilizados em diversas operações, como aritméticas, lógicas e de transferência de dados, e estão conectados ao barramento de dados para uma comunicação eficiente com outras partes do sistema. Apesar de sua rápida capacidade de acesso, os dados nos registradores são voláteis e perdidos quando o microprocessador é desligado, exigindo programas e algoritmos projetados para minimizar essa dependência. Em suma, o banco de registradores contribui significativamente para o desempenho e eficiência do processamento de um microprocessador. No caso do processador BLDIM, contamos com 16 registradores que é um número significativo e suficiente para o fim necessário do projeto que é a multiplicação de matrizes 4x4. Desses, os registradores referentes aos códgios 0000 e 0001 são reservados a guardar os valores fixos 0 e 1, respectivamente, pela sua utilização recorrente em processos de adição e multiplicação. De 0010 a 1011 temos os registradores temporários, de 1100 a 1110 temos registradores salvos e reservamos o registrador 1111 para desempenhar a função de registrador de endereço, utilizado em funções jal. Vide seção 3.7 para identificar o código específico de cada registrador.

3.2 Memória de Dados

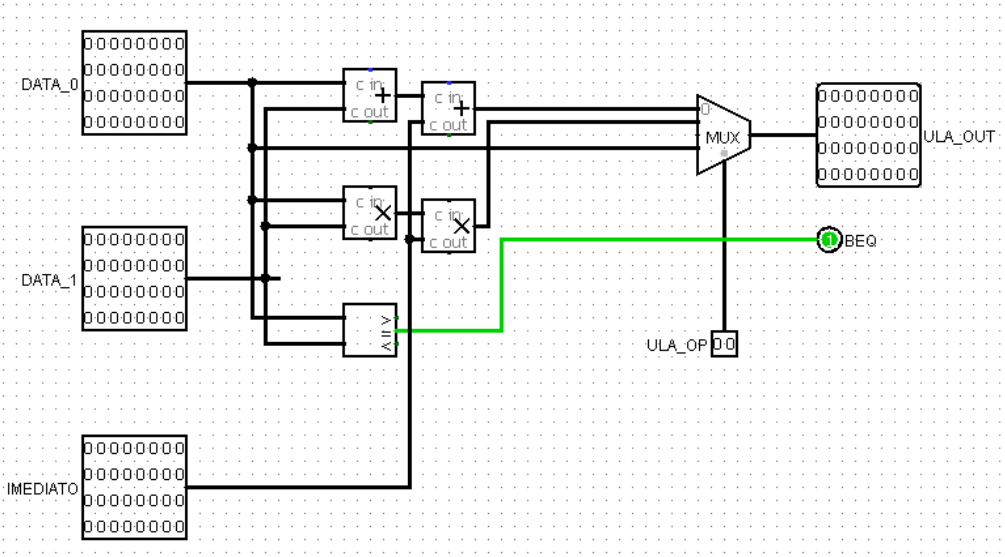
A memória de dados de um microprocessador é responsável por armazenar temporariamente os dados utilizados durante a execução de programas. Ela difere da memória de programa, que contém as instruções do programa. Organizada em células de memória, cada uma com um endereço único, a memória de dados permite que o microprocessador acesse e manipule os dados armazenados. Sua capacidade de armazenamento varia de acordo com a arquitetura do microprocessador, sendo essencial para a execução de programas complexos. O acesso à memória é feito por instruções de leitura e escrita, garantindo a manipulação adequada dos dados. Integrada à hierarquia de memória do sistema, que inclui memória cache, RAM e armazenamento secundário,

a memória de dados contribui para o desempenho e a capacidade de armazenamento do sistema. No entanto, os dados armazenados nela são voláteis e perdidos quando o microprocessador é desligado, destacando a importância de salvar dados importantes em memória não volátil, como discos rígidos ou SSDs, para preservá-los entre diferentes execuções. Em suma, a memória de dados é fundamental para o funcionamento eficiente do microprocessador, fornecendo uma área de armazenamento acessível durante a execução de operações.

3.3 Memória de Instruções

A memória de instruções em um microprocessador é crucial para armazenar as sequências de instruções que o processador deve executar. Ao contrário da memória de dados, que contém os dados necessários para os programas, a memória de instruções guarda o código de máquina que representa as operações a serem realizadas. Essa memória é organizada em células com endereços únicos, e sua capacidade de armazenamento varia conforme a arquitetura do processador. Durante a execução de um programa, o processador busca e executa as instruções sequencialmente, decodificando cada uma para determinar a operação a ser realizada e os dados envolvidos. No entanto, os dados na memória de instruções são voláteis, perdidos quando o microprocessador é desligado, portanto, é essencial armazenar o código do programa em dispositivos não voláteis para preservar os programas entre as execuções. Em resumo, a memória de instruções é vital para o funcionamento dos programas, fornecendo ao processador as instruções necessárias para executar tarefas computacionais com eficiência e precisão.

3.4 Unidade Lógico-Aritmética (ULA)



A Unidade Lógico-Aritmética (ULA), também conhecida como ALU (Arithmetic Logic Unit), é uma parte fundamental de um microprocessador responsável por realizar operações aritméticas e lógicas em dados. Ela é projetada para executar uma variedade de operações, incluindo adição, subtração, multiplicação, divisão, bem como operações lógicas como AND, OR, NOT e XOR.

A ULA recebe dados de entrada, chamados de operandos, realiza a operação

especificada e produz um resultado. Ela é capaz de trabalhar com números inteiros, números em ponto flutuante e bits individuais, dependendo da arquitetura do microprocessador.

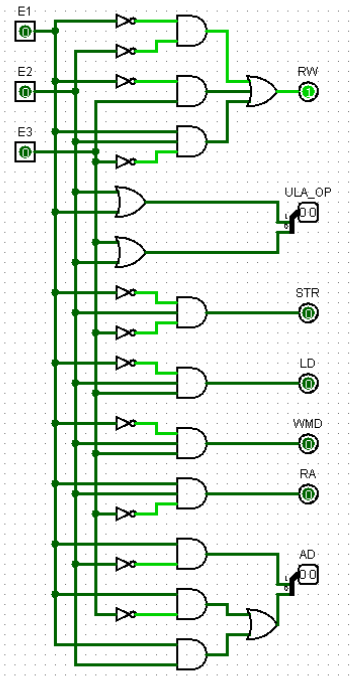
Além das operações aritméticas básicas, a ULA também é responsável por operações de comparação. Por exemplo, ela pode comparar dois números para determinar se são iguais, maiores ou menores. Isso é essencial para a tomada de decisões durante a execução de programas, como em instruções de desvio condicional.

A ULA é geralmente controlada por instruções de máquina enviadas pela unidade de controle do microprocessador. Essas instruções especificam a operação a ser realizada e os operandos envolvidos. Dependendo da arquitetura do microprocessador, a ULA pode ter um conjunto fixo de operações ou ser configurável para suportar uma variedade de operações definidas pelo usuário.

Em resumo, a ULA desempenha um papel crucial no processamento de dados de um microprocessador, realizando operações aritméticas, lógicas e de comparação. Sua capacidade de executar essas operações de forma eficiente e rápida contribui significativamente para o desempenho geral do sistema computacional.

Na ULA do BLDIM contamos com um conjunto de apenas 4 operações, sendo elas: somar ou multiplicar os valores de DATA_0, DATA_1 e IMEDIATO, emitindo o resultado na saída ULA_OUT, passar diretamente o valor de DATA_0 para ULA_OUT ou emitir 1 na saída BEQ caso DATA_0 e DATA_1 tenham o mesmo valor e 0 caso contrário.

3.5 Unidade de Controle (UC)



A Unidade de Controle (UC) de um microprocessador é responsável por coordenar e controlar as operações de todos os componentes do processador, garantindo que as instruções sejam executadas corretamente e na sequência adequada. Ela interpreta as instruções de máquina do programa, decodifica-as e gera os sinais de controle necessários para coordenar as operações dos outros componentes do processador, como a Unidade Lógico-Aritmética (ULA), o banco de registradores, a memória e os dispositivos de entrada e saída.

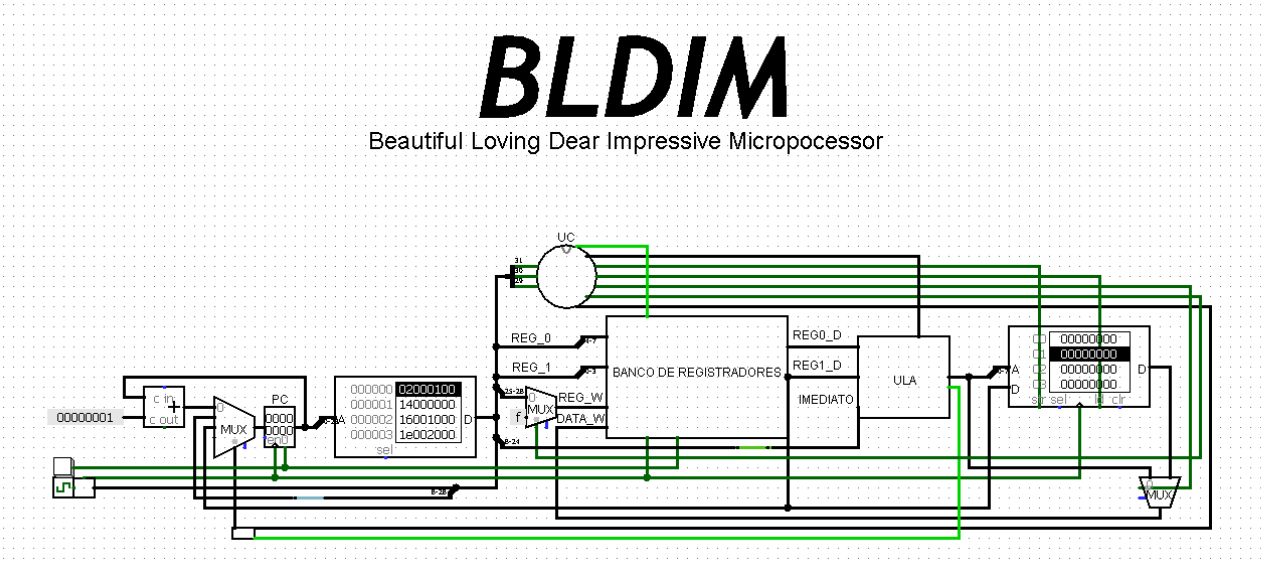
A UC geralmente opera em ciclos de instrução, onde cada ciclo representa a execução de uma instrução do programa. Durante cada ciclo, a UC busca a próxima instrução na memória de instruções, decodifica-a para entender qual operação deve ser realizada e em que dados ela deve operar e, em seguida, gera os sinais de controle apropriados para coordenar as operações dos outros componentes do processador.

Além de coordenar as operações de execução de instruções, a UC também lida com situações de exceção e interrupção, como erros de execução, acesso indevido à memória ou solicitações de entrada/saída. Quando ocorrem exceções ou interrupções, a UC interrompe a execução normal do programa, salva o estado atual

do processador e direciona o fluxo de controle para lidar com a situação de maneira apropriada, como tratamento de erro, chamada de rotinas de serviço ou alteração do contexto de execução.

Em resumo, a Unidade de Controle desempenha um papel crítico no funcionamento de um microprocessador, coordenando e controlando as operações de execução de instruções, gerenciando exceções e interrupções e garantindo o correto funcionamento do processador como um todo.

3.6 Processador



O processador é o resultado final da conexão dos demais circuitos-base apresentados. Assim, temos um percurso natural pelo qual as instruções passarão dentro do BLDIM: A partir do PC acessamos a instrução atual, que é desmembrada entre os bits de interesse da UC, e os bits de endereços de registradores e número imediato. Então, com os sinais de controle ofertados pela UC, acessamos e tratamos as saídas, caso necessário, do Banco de Registradores, da ULA, da Memória de Dados e então novamente do PC, reiniciando o ciclo, nessa ordem.

Multiplexadores ao longo do processador definem, com o auxílio dos sinais da UC, se haverá um salto advindo de uma instrução BEQ, se devemos saltar para algum endereço que não o próximo nas instruções, se o registrador de escrita é definido pelos bits na instrução ou se é o $ra, e se escreveremos no Banco um dado originário da ULA ou da Memória.

3.7 Compilador em Python

Vamos analisar o código linha por linha:

```python
assembly = [
    "add $um 1 $zero $zero",
    "add $s0 0 $zero $zero",
    "add $s1 16 $zero $zero",
    "add $v0 32 $zero $zero",
    "add $t7 4 $zero $zero",
    "add $s2 0 $zero $zero",
    "beq $zero 33 $s2 $t7",
    "add $s3 0 $zero $zero",
    "beq $zero 31 $s3 $t7",
    "add $t0 0 $zero $zero",
    "add $s4 0 $zero $zero",
    "mul $t1 1 $t7 $s2",
    'mul $t2 1 $um $s3',
    'beq $zero 28 $s4 $t7',
    'mul $t3 1 $um $s4',
    'add $t3 0 $s0 $t3',
    'add $t3 0 $t3 $t1',
    'lw $t4 0 $t3 $zero',
    'mul $t3 1 $t7 $s4',
    'add $t3 0 $s1 $t3',
    'add $t3 0 $t3 $t2',
    'lw $t5 0 $t3 $zero',
    'mul $t4 1 $t4 $t5',
    'add $t0 0 $t0 $t4',
    'add $s4 1 $s4 $zero',
    'j $zero 13 $zero $zero',
    'add $t1 0 $v0 $t1',
    'add $t1 0 $t1 $t2',
    'sw $zero 0 $t1 $t0',
    'add $s3 1 $s3 $zero',
    'j $zero 8 $zero $zero',
    'add $s2 1 $s2 $zero',
    'j $zero 6 $zero $zero'
    ]
```

```python
registers = {
    "$zero": '0000',
    "$um": '0001',
    "$t0": '0010',
    "$t1": '0011',
    "$t2": '0100',
    "$t3": '0101',
    "$t4": '0110',
    "$t5": '0111',
    "$t6": '1000',
    "$t7": '1001',
    "$s0": '1010',
    "$s1": '1011',
    "$s2": '1100',
    "$s3": '1101',
    "$s4": '1110',
    "$v0": '1111'
    }

functions = {
    "add": '000',
    "mul": '001',
    "sw": '010',
    "lw": '011',
    "beq": '100',
    "jr": '101',
    "jal": '110',
    "j": '111'
    }
```

```python
Hex = {
    "0000": "0",
    "0001": "1",
    "0010": "2",
    "0011": "3",
    "0100": "4",
    "0101": "5",
    "0110": "6",
    "0111": "7",
    "1000": "8",
    "1001": "9",
    "1010": "a",
    "1011": "b",
    "1100": "c",
    "1101": "d",
    "1110": "e",
    "1111": "f"
}
```

```python
for line in assembly:
    func, reg1, imdt, reg2, reg3 = line.split(" ")
    bin_line = []
    hex_line = []
    for k in functions.keys():
        if func == k:
            bin_line.append(functions[k])
            break
    for k in registers.keys():
        if reg1 == k:
            bin_line.append(registers[k])
            break
    q = int(imdt)
    imdt = []
    while q > 0:
        r = q % 2
        q = q // 2
        imdt.append(str(r))
    imdt.reverse()
    tam = len(imdt)
    for i in range(17-tam):
        imdt.insert(0, "0")
    imdt = "".join(imdt)
    bin_line.append(imdt)
    for k in registers.keys():
        if reg2 == k:
            bin_line.append(registers[k])
            break
    for k in registers.keys():
        if reg3 == k:
            bin_line.append(registers[k])
            break
    bin_line = "".join(bin_line)
    for i in range(0, 32, 4):
        for k in Hex.keys():
            if bin_line[i:i+4] == k:
                hex_line.append(Hex[k])
    hex_line = "".join(hex_line)
    print(hex_line)
```

1. assembly = [...]: Aqui, uma lista chamada `assembly` é definida, contendo várias strings que representam instruções em linguagem de montagem. Cada instrução consiste em uma operação seguida por quatro argumentos: dois registradores e dois valores imediatos.

2. registers = {...}: Em seguida, um dicionário chamado `registers` é definido, mapeando os nomes dos registradores para seus valores binários correspondentes.

3. functions = {...}: Outro dicionário chamado `functions` é definido, mapeando os nomes das funções (operações) para seus códigos binários correspondentes.

4. bin_assembly = []: Uma lista vazia chamada `bin_assembly` é inicializada para armazenar as instruções convertidas para binário.

5. for line in assembly:: Inicia um loop que percorre cada linha na lista `assembly`.

6. func, reg1, imdt, reg2, reg3 = line.split(" "): Divide cada linha da lista `assembly` em cinco partes separadas por espaços em branco e as atribui às variáveis `func`, `reg1`, `imdt`, `reg2` e `reg3`.

7. bin_line = []: Inicializa uma lista vazia chamada `bin_line` para armazenar a versão binária da linha atual.

8. for k in functions.keys(): ...: Itera sobre as chaves do dicionário `functions`.

9. if func == k: ...: Verifica se o valor da variável `func`, que representa a função da instrução, corresponde à chave atual `k` no dicionário `functions`.

10. bin_line.append(functions[k]): Se a função corresponder, adiciona o código binário correspondente ao `bin_line`.

11. Os próximos dois loops for k in registers.keys(): ... fazem algo semelhante, mas para os registradores especificados na instrução.

12. bin_line.append("000000000"): Adiciona uma sequência de zeros para representar a parte de destino da instrução.

13. bin_line.append(imdt): Adiciona o valor imediato (imdt) especificado na instrução ao bin_line.

14. O próximo loop for k in registers.keys(): ... é repetido duas vezes mais, para os últimos dois registradores especificados na instrução.

15. bin_line = "".join(bin_line): Junta todos os elementos da lista bin_line em uma única string, representando a instrução completa em binário.

16. print(bin_line): Imprime a instrução convertida para binário.

Este código traduz as instruções Assembly para sua representação binária correspondente, usando os dicionários `functions` e `registers` para obter os códigos de operação e os valores dos registradores, respectivamente.

3.8 Código em BLDIM-Assembly da Multiplicação de Matrizes

```
add $um, 1, $zero, $zero
add $s0, 0, $zero, $zero
add $s1, 64, $zero, $zero
add $ra, 128, $zero, $zero
add $t7, n, $zero, $zero
add $s2, 0, $zero, $zero
beq 33, $s2, $t7 (begin_for_i)
        add $s3, 0, $zero, $zero
        beq 31, $s3, $t7 (begin_for_j)
                add $t0, 0, $zero, $zero
                add $s4, 0, $zero, $zero
                mul $t1, 1, $t7, $s2            # t1 = i * n
                mul $t2, 1, $um, $s3            # t2 = j
                beq 26, $s4, $t7 (begin_for_k)
                        mul $t3, 1, $um, $s4    # t3 = k
                        add $t3, 0, $s0, $t3
                        add $t3, 0, $t3, $t1
                        lw $t4, $t3             # t4 = mat1[i][k]
                        mul $t3, 1, $t7, $s4    # t3 = k * n
                        add $t3, 0, $s1, $t3
                        add $t3, 0, $t3, $t2
                        lw $t5, $t3             # t5 = mat2[k][j]
                        mul $t4, 1, $t4, $t5
                        add $t0, 0, $t0, $t4    # $t0 += mat1[i][k] * mat2[k][j]
                        add $s4, 1, $s4, $zero
                        j 13
                add $t1, 0, $ra, $t1 (end_for_k)
                add $t1, 0, $t1, $t2
                sw $t0, $t1
                add $s3, 1, $s3, $zero
                j 8
        add $s2, 1, $s2, $zero (end_for_j)
        j 6
(end_for_i)
```

Vamos analisar o código linha por linha:

1. `add $um, 1, $zero, $zero`: Aqui, estamos adicionando o valor 1 ao registrador `$zero` (que é sempre zero) e armazenando o resultado no registrador `$um`. Basicamente, estamos inicializando `$um` com o valor 1.

2. `add $s0, 0, $zero, $zero`: Similar ao anterior, estamos inicializando o registrador `$s0` com o valor 0.

3. `add $s1, 64, $zero, $zero`: Aqui, estamos definindo o registrador `$s1` com o valor 64.

4. `add $ra, 128, $zero, $zero`: O registrador `$ra` (Return Address) está sendo configurado com o valor 128.

5. `add $t7, n, $zero, $zero`: O registrador `$t7` está sendo inicializado com o valor da variável `n`.

6. `add $s2, 0, $zero, $zero`: Inicialização do registrador `$s2` com o valor 0.

7. `beq 33, $s2, $t7 (begin_for_i)`: Aqui, estamos verificando se o valor de `$s2` é igual a 33. Se for verdadeiro, o fluxo de execução vai para a etiqueta `begin_for_i`.

8. `add $s3, 0, $zero, $zero`: Inicialização do registrador `$s3` com o valor 0.

9. `beq 31, $s3, $t7 (begin_for_j)`: Verificamos se o valor de `$s3` é igual a 31. Se sim, o programa vai para a etiqueta `begin_for_j`.

10. A partir daqui, entramos em um loop aninhado com três níveis (para `i`, `j` e `k`). Vou explicar o padrão geral:

- `mul $t1, 1, $t7, $s2`: Multiplicamos o valor de `$t7` (que é `n`) pelo valor de `$s2` (que varia com o loop `i`). O resultado é armazenado em `$t1`.

- `mul $t2, 1, $um, $s3`: Multiplicamos o valor de `$um` (que é 1) pelo valor de `$s3` (que varia com o loop `j`). O resultado é armazenado em `$t2`.

- `lw $t4, $t3`: Carregamos o valor da memória na posição indicada por `$t3` (que é uma combinação de índices `i` e `k`) e armazenamos em `$t4`.

- `lw $t5, $t3`: Carregamos o valor da memória na posição indicada por `$t3` (que é uma combinação de índices `k` e `j`) e armazenamos em `$t5`.

- `mul $t4, 1, $t4, $t5`: Multiplicamos os valores de `$t4` e `$t5` e armazenamos o resultado em `$t4`.

- `add $t0, 0, $t0, $t4`: Adicionamos o valor de `$t4` ao acumulador `$t0`.

- `add $s4, 1, $s4, $zero`: Incrementamos o contador `k`.

- `j 13`: Saltamos para a etiqueta `begin_for_k`.

11. O processo se repete para os loops `j` e `i`.

12. No final, o resultado final está em `$t0`.

4. CONCLUSÃO

Ao longo deste trabalho, desenvolvemos um projeto ambicioso e desafiador: a criação de um microprocessador denominado BLDIM, com o objetivo principal de realizar a multiplicação de matrizes 4 por 4. Utilizamos a ferramenta Logisim para a construção do microprocessador em nível de circuitos lógicos, além de implementar nossas próprias instruções em linguagem Assembly para alcançar esse objetivo.

Durante o desenvolvimento do projeto, enfrentamos diversos desafios, desde a compreensão dos conceitos fundamentais de arquitetura de computadores até a implementação prática das instruções e do circuito do microprocessador. A necessidade de criar um compilador em linguagem Python para traduzir as instruções Assembly para a linguagem de máquina do BLDIM acrescentou uma camada adicional de complexidade ao projeto, mas também proporcionou uma oportunidade única de explorar diferentes aspectos da computação.

Ao finalizar o projeto, alcançamos nosso principal objetivo de implementar com sucesso a multiplicação de matrizes 4 por 4 no microprocessador BLDIM. Este resultado representa não apenas uma conquista técnica, mas também uma demonstração do poder e da versatilidade dos microprocessadores customizados e das linguagens de programação de baixo nível.

Além disso, ao longo do processo, adquirimos uma compreensão mais profunda da arquitetura de computadores, dos circuitos lógicos e dos princípios de programação de sistemas embarcados. Também desenvolvemos habilidades práticas em design de circuitos, programação Assembly e desenvolvimento de software, especialmente na criação do compilador em Python.

Em resumo, o projeto do microprocessador BLDIM representa uma jornada de aprendizado significativa e gratificante, que nos permitiu explorar novos horizontes na área de computação. Esperamos que este trabalho possa servir como inspiração para futuros projetos e contribuir para o avanço contínuo da tecnologia de microprocessadores e sistemas computacionais.