UFRR

UNIVERSIDADE FEDERAL DE RORAIMA

CIÊNCIA DA COMPUTAÇÃO - GRADUAÇÃO

RELATÓRIO DO PROJETO: PROCESSADOR LM

Boa vista – Roraima

2025

MARCUS VINÍCIUS MAIA DOS SANTOS

ÍGOR PEREIRA DA SILVA

RELATÓRIO DO PROJETO: PROCESSADOR LM

Relatório técnico apresentado ao Prof. Dr. Herbert Oliveira Rocha, como requisito de obtenção de nota parcial na disciplina DCC 301 - Arquitetura e Organização de Computadores.

Boa vista – Roraima

2025

RESUMO

Este trabalho aborda o projeto e implementação de um processador LM, utilizando do software Quartus Prime, da linguagem Assembly MIPS e da linguagem VHDL (Very High-Speed Integration Circuit HDL). O processador em questão é um RISC (Reduced Instruction Set Computer) de 8 bits.

O relatório então servirá de instrumento de avaliação dos alunos para a disciplina de AOC (Arquitetura e Organização de Computadores), ministrada pelo professor Herbert Oliveira Rocha.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. ESPECIFICAÇÃO

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador LM, bem como a descrição detalhada de cada etapa da construção do processador.

1.1. PLATAFORMA DE DESENVOLVIMENTO

Para a implementação do processador LM foi utilizada a IDE *Quartus Prime*:

Figura 1 – Especificações no *Quartus Prime*

Flow Status	Successful - Mon Mar 10 20:57:38 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	CPU_LM
Top-level Entity Name	CPU_LM
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	5 / 56,480 (< 1 %)
Total registers	8
Total pins	70 / 268 (26 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)

Fonte: Elaborada pelos autores.

1.2. Conjunto de Instruções

O processador LM possui 2 registradores: $s0 e $s1. Assim como 11 formatos de instruções de 8 bits cada, instruções do tipo R, I e J. Seguem algumas considerações sobre as estruturas contidas nas instruções:

- Opcode: indica ao processador qual a instrução a ser executada;

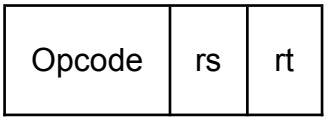
· Reg1: o registrador contendo o primeiro operando fonte e adicionalmente para alguns tipos de instruções (ex. Instruções do tipo R) é o registrador de destino;

· Reg2: o registrador contendo o segundo operando fonte.

Tipos de Instruções:

- Tipo R: Este tipo de instrução trata de operações aritméticas.

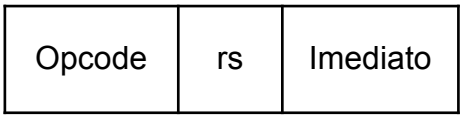
  - Formato para escrita de código na linguagem MIPS:



  - Formato para escrita em código binário:

Instrução do tipo R		
Opcode	rs	rt
4bits	2bits	2bits
7-4	3-2	1-0

- Tipo I: Este tipo de instrução aborda carregamentos diretos na memória.
  - Formato para escrita de código na linguagem MIPS:



  - Formato para escrita em código binário:

Instrução do tipo I		
Opcode	rs	Imediato
4bits	2bits	2bits
7-4	3-2	1-0

- Tipo J: Este tipo de instrução é responsável por desvios condicionais e incondicionais.
  - Formato para escrita de código na linguagem MIPS:

Opcode	Endereço

- Formato para escrita em código binário:

Instrução do tipo J	
Opcode	Endereço
4bits	4bits
7-4	3-0

Visão geral das instruções do Processador LM:

O número de bits do campo Opcode das instruções é igual a quatro, sendo assim obtemos um total de 16 Opcodes (0-15) que são distribuídos entre as instruções, assim como é apresentado na Tabela 1.

Tabela 1 – Opcodes suportados pelo Processador LM

Opcode	Sintaxe	Formato	Significado	Exemplos
0000	add	R	Soma	add $s0, $s1
0001	addi	I	Soma Imediata	addi $s0, 3
0010	sub	R	Subtração	sub $s0, $s1
0011	subi	I	Subtração Imediata	subi $s0, 6
0100	lw	I	Load Word	lw $s0 ram (00)
0101	sw	I	Store Word	sw $s0 ram (00)
0110	li	I	Load Imediato	li $s0 2
0111	beq	J	Branch Equal	beq endereço
1000	if	J	If Equal	If $s0 $s1
1001	J	J	Jump	j endereço (0000)

Fonte: Elaborada pelos autores.

1.3. Descrição do Hardware

Nesta seção são descritos os componentes do hardware que compõem o processador LM, incluindo uma descrição de suas funcionalidades, valores de entrada e saída.

1.3.1. ALU

O componente ALU (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas (considerando apenas resultados inteiros), dentre elas: soma e subtração. Adicionalmente, ela efetua operações de comparação de valor como BEQ.

O componente ALU recebe como entrada quatro valores:

• CLOCK: dado de 1 bit;

• A e B: dados de 1 byte;

• OP: opcode de 4 bits.

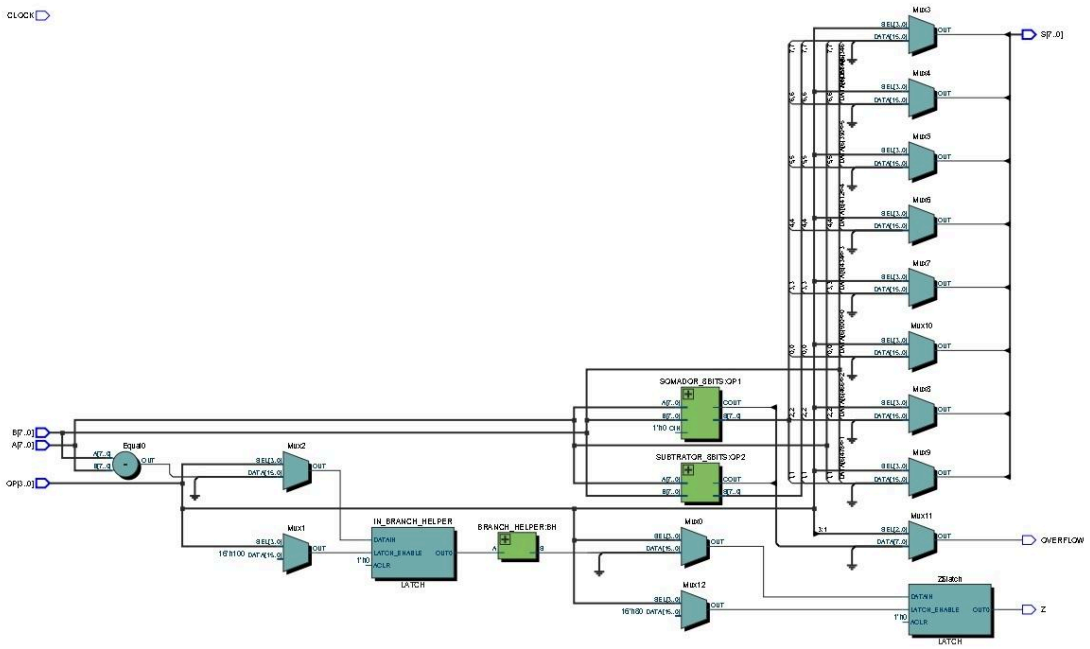
A ALU possui três saídas:

• S: resultado de 1 byte;

• Z: resultado de 1 bit para verificar se o valor retornado é zero;

• OVERFLOW: resultado de 1 bit para verificar se a operação resulta num overflow.

Figura 2 – RTL Viewer da ALU



Fonte: Elaborada pelos autores.

1.3.2. BANCO_REGISTRADORES

O componente BANCO_REGISTRADORES tem como principal objetivo escrever, ler e armazenar valores nos registradores.

O componente BANCO_REGISTRADORES recebe como entrada quatro valores:

• CLOCK: dado de 1 bit;

• REG_WRITE: dado de 1 bit que indica se há escrita em registradores;

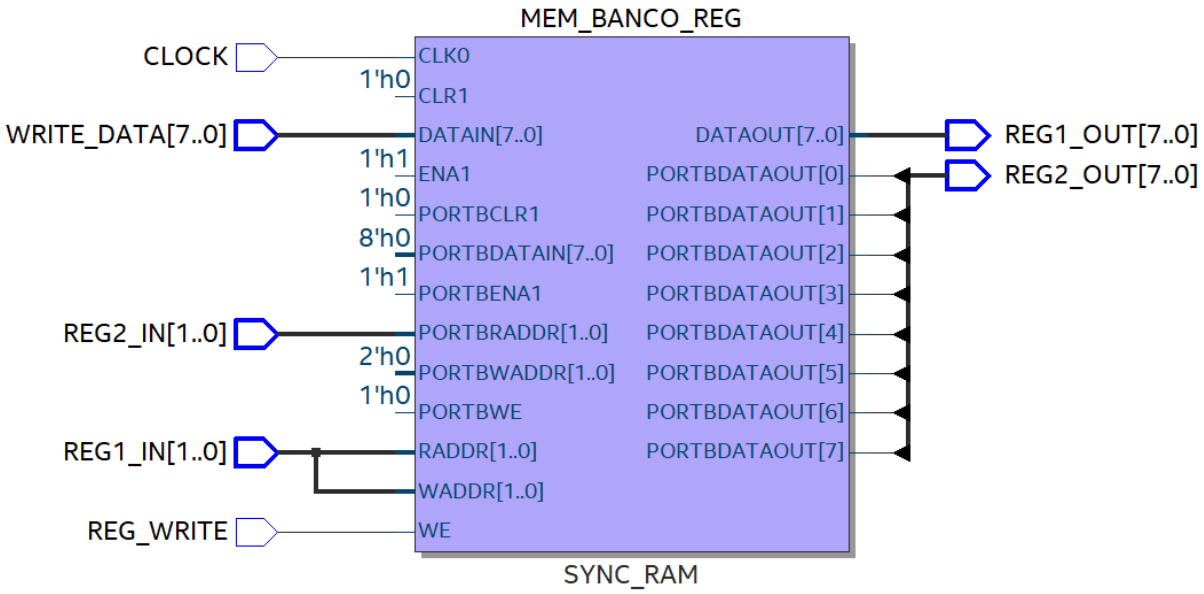
• REG1_IN e REG2_IN: dados de dois bits que indicam sobre quais registradores devem ser executados as operações;

• WRITE_DATA: dado de 1 byte a ser escrito em registrador.

O componente BANCO_REGISTRADORES tem duas saídas:

• REG1_OUT e REG2_OUT: dados de 1 byte armazenados nos registradores.

Figura 3 – RTL Viewer do BANCO_REGISTRADORES



Fonte: Elaborada pelos autores.

1.3.3. CONTADOR_SINCRONO

O componente CONTADOR_SINCRONO tem como objetivo somar 1 ao PC, avançando assim para a próxima linha de código do programa na memória ROM.

O componente CONTADOR_SINCRONO recebe como entrada dois valores:

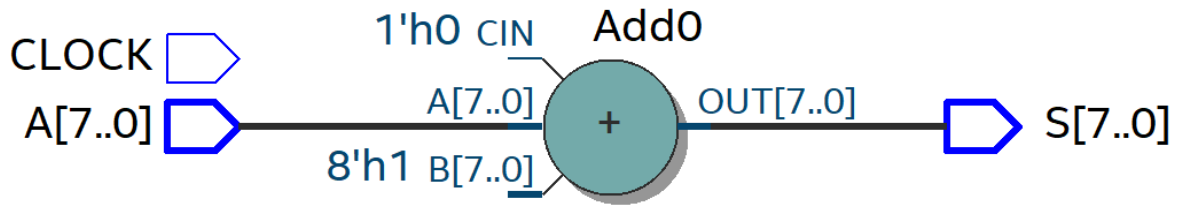
• CLOCK: dado de 1 bit;

• A: dado de 1 byte equivalente ao endereço no PC.

O componente CONTADOR_SINCRONO tem como saída o valor:

• S: PC + 1.

Figura 4 – RTL Viewer do CONTADOR_SINCRONO



Fonte: Elaborada pelos autores.

1.3.4. DIV_INSTRUCAO

O componente DIV_INSTRUCAO divide a instrução recebida em 4 trilhas, que são utilizadas para definir qual o tipo de instrução a ser executado e quais seus operandos.

O componente DIV_INSTRUCAO recebe como entrada:

• INSTRUCTION: dado de 1 byte com a instrução.

O componente DIV_INSTRUCAO tem como saída:

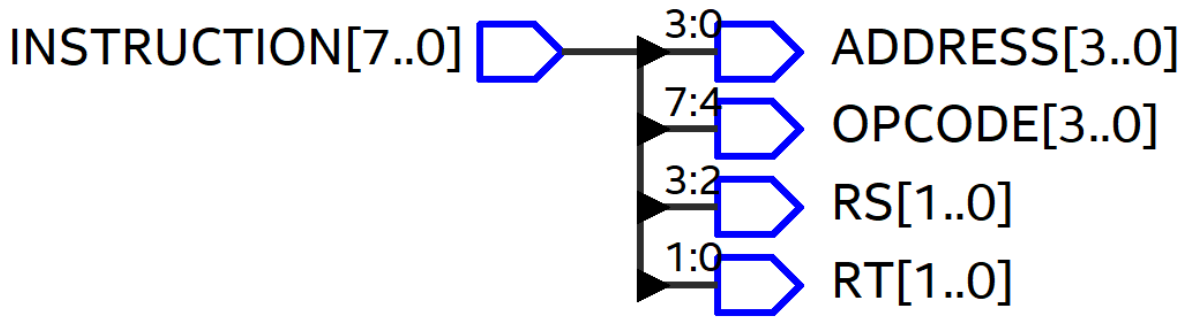
• ADDRESS: dado de 4 bits com endereço;

• OPCODE: dado de 4 bits com o opcode;

• RS: dado de 2 bits com o primeiro operando;

• RT: dado de 2 bits com o segundo operando.

Figura 5 – RTL Viewer do DIV_INSTRUCAO



Fonte: Elaborada pelos autores.

1.3.5. EXTENSOR_2X8

O componente EXTENSOR_2X8 extende um sinal de 2 bits para um de 1 byte.

O componente EXTENSOR_2X8 recebe como entrada:

• A: dado de 2 bits a ser extendido.

O componente EXTENSOR_2X8 tem como saída:

• S: dado de 1 byte extendido.

Figura 6 – RTL Viewer do EXTENSOR_2X8

Fonte: Elaborada pelos autores.

1.3.6. EXTENSOR_4X8

O componente EXTENSOR_4X8 extende um sinal de 4 bits para um de 1 byte.

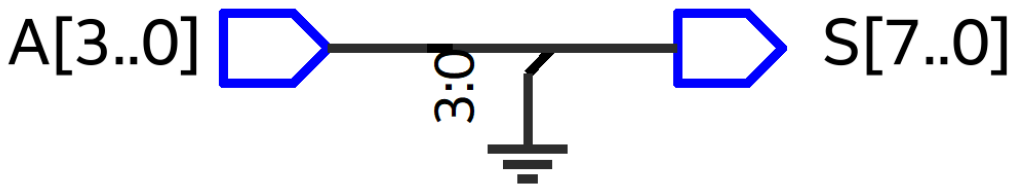
O componente EXTENSOR_4X8 recebe como entrada:

• A: dado de 4 bits a ser extendido.

O componente EXTENSOR_4X8 tem como saída:

• S: dado de 1 byte extendido.

Figura 7 – RTL Viewer do EXTENSOR_4X8



Fonte: Elaborada pelos autores.

1.3.7. MUX_2X1

O componente MUX_2X1 seleciona um entre dois dados de 1 byte com base num seletor.

O componente MUX_2X1 recebe como entrada:

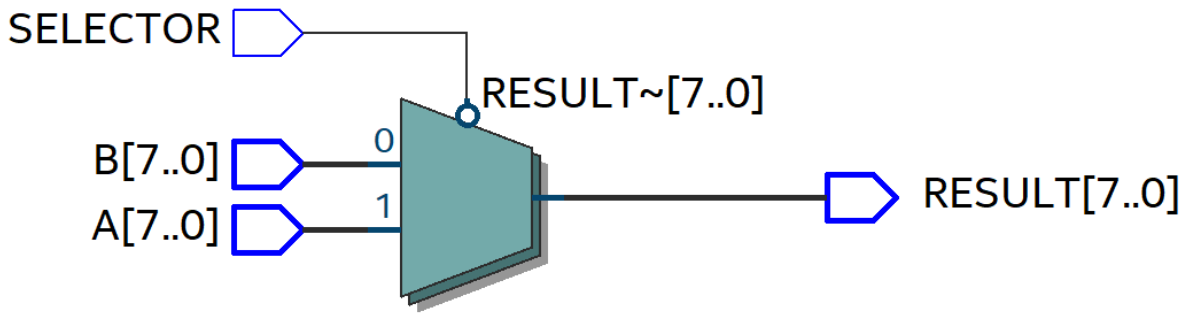
• A e B: dados de 1 byte;

• SELECTOR: dado de 1 bit.

O componente MUX_2X1 tem como saída:

• RESULT: dado de 1 byte que foi selecionado.

Figura 8 – RTL Viewer do MUX_2X1



Fonte: Elaborada pelos autores.

1.3.8. PC

O componente PC é responsável por passar a linha de código do programa que deve ser executada.

O componente PC recebe como entrada:

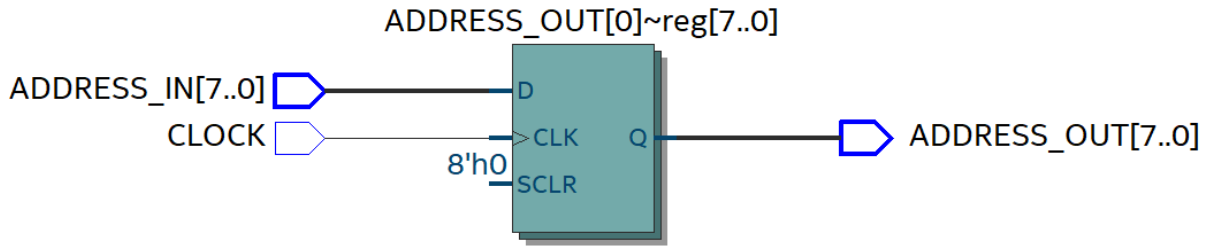
• CLOCK: dado de 1 bit;

• ADDRESS_IN: dado de 1 byte com a linha atualizada.

O componente PC tem como saída:

• ADDRESS_OUT: dado de 1 byte com a linha atual.

Figura 9 – RTL Viewer do PC



Fonte: Elaborada pelos autores.

1.3.9. RAM

O componente RAM é responsável por armazenar e ler até 8 dados de 1 byte por meio de instruções load e store.

O componente RAM recebe como entrada:

• CLOCK: dado de 1 bit;

• A: dado de 1 byte a ser escrito;

• ADDRESS: dado de 1 byte com endereço;

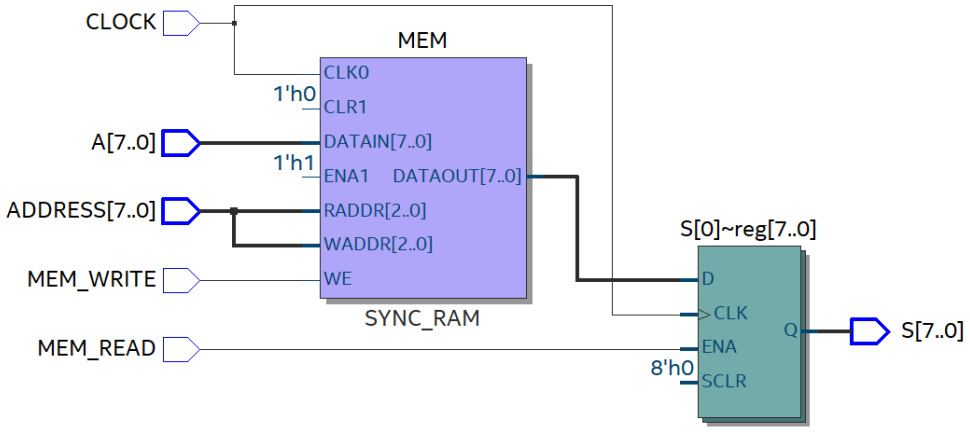
• MEM_WRITE: dado de 1 bit que serve como flag;

• MEM_READ: dado de 1 bit que serve como flag.

O componente RAM tem como saída:

• S: dado de 1 byte com o resultado.

Figura 10 – RTL Viewer da RAM



Fonte: Elaborada pelos autores.

1.3.10. ROM

O componente ROM é responsável por armazenar o programa, com até 256 linhas de código.

O componente ROM recebe como entrada:

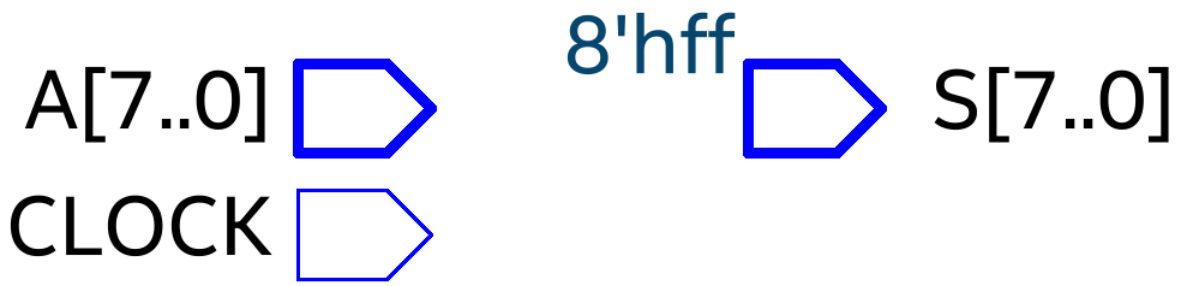
• CLOCK: dado de 1 bit;

• A: dado de 1 byte com o endereço da linha a ser lida.

O componente ROM tem como saída:

• S: dado de 1 byte armazenado naquela linha.

Figura 11 – RTL Viewer da ROM



Fonte: Elaborada pelos autores.

1.3.11. SOMADOR_8BITS

O componente SOMADOR_8BITS tem como principal objetivo efetuar operações de soma entre dois dados de 1 byte.

O componente SOMADOR_8BITS recebe como entrada:

• A e B: dados de 1 byte com valores a serem somados;

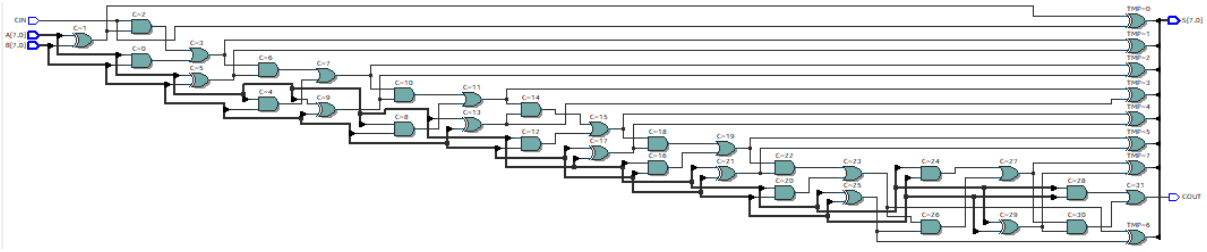
• CIN: dado de 1 byte com o carry in.

O componente SOMADOR_8BITS tem como saída:

• S: dado de 1 byte com o resultado;

• COUT: dado de 1 bit com o carry out.

Figura 12 – RTL Viewer do SOMADOR_8BITS



Fonte: Elaborada pelos autores.

1.3.12. SUBTRATOR_8BITS

O componente SUBTRATOR_8BITS tem como principal objetivo efetuar operações de subtração entre dois dados de 1 byte.

O componente SUBTRATOR_8BITS recebe como entrada:

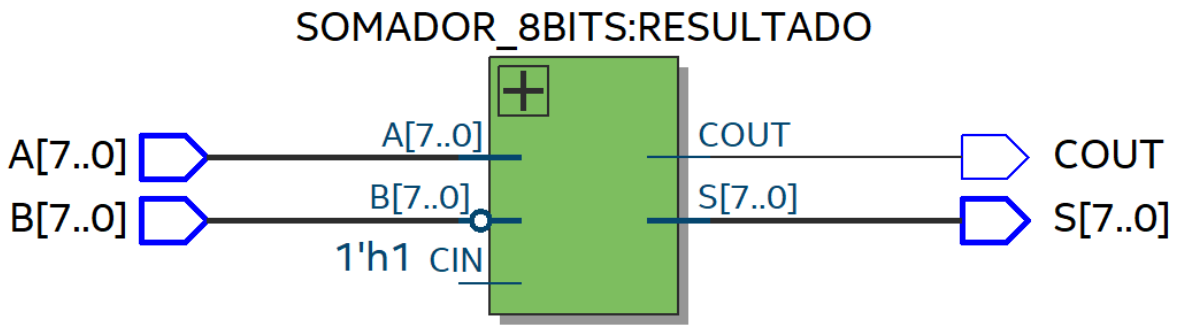
• A e B: dados de 1 byte com valores a se subtrair;

O componente SUBTRATOR_8BITS tem como saída:

• S: dado de 1 byte com o resultado;

• COUT: dado de 1 bit com o carry out.

Figura 13 – RTL Viewer do SUBTRATOR_8BITS



Fonte: Elaborada pelos autores.

1.3.13. UNIDADE_DE_CONTROLE

O componente UNIDADE_DE_CONTROLE tem como principal objetivo administrar as flags necessárias para cada tipo de instrução.

O componente UNIDADE_DE_CONTROLE recebe como entrada:

• CLOCK: dado de 1 bit;

• OPCODE: dado de 3 bits, indicando a operação a ser realizada;

O componente UNIDADE_DE_CONTROLE tem como saída:

• JUMP: dado de 1 bit que serve de flag se há uma operação do tipo jump;

• BRANCH: dado de 1 bit que serve de flag se há uma operação característica de instrução tipo J;

• MEMREAD: dado de 1 bit que server de flag se há leitura de memória;

• MEMTOREG: dado de 1 bit que serve de flag se há escrita de um dado da memória num registrador;

• ALUOP: dado de 3 bits que indica a operação a ser realizada na ALU;

• MEMWRITE: dado de 1 bit que serve de flag se há escrita na memória;

• ALUSRC: dado de 1 bit que serve de flag se há necessidade de executar operação com registrador ou imediato;

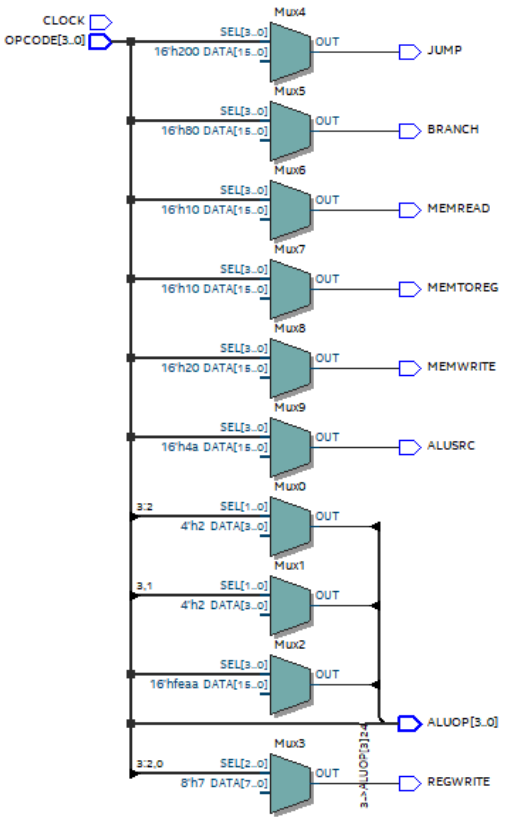
• REGWRITE: dado de 1 bit que serve de flag se há escrita de dados no banco de registradores.

Tabela 2 - Relações entre OPCODES e flags na UNIDADE_DE_CONTROLE

OPCODE	JUMP	BRANCH	MEMREAD	MEMTOREG	ALUOP	MEMWRITE	ALUSRC	REGWRITE
0000	0	0	0	0	0	0	0	1
0001	0	0	0	0	0	0	1	1
0010	0	0	0	0	0	0	0	1
0011	0	0	0	0	0	0	1	1
0100	0	0	1	1	0	0	0	1
0101	0	0	0	0	0	1	0	0
0110	0	0	0	0	0	0	1	1
0111	0	1	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0
1001	1	0	0	0	0	0	0	0

Fonte: Elaborada pelos autores.

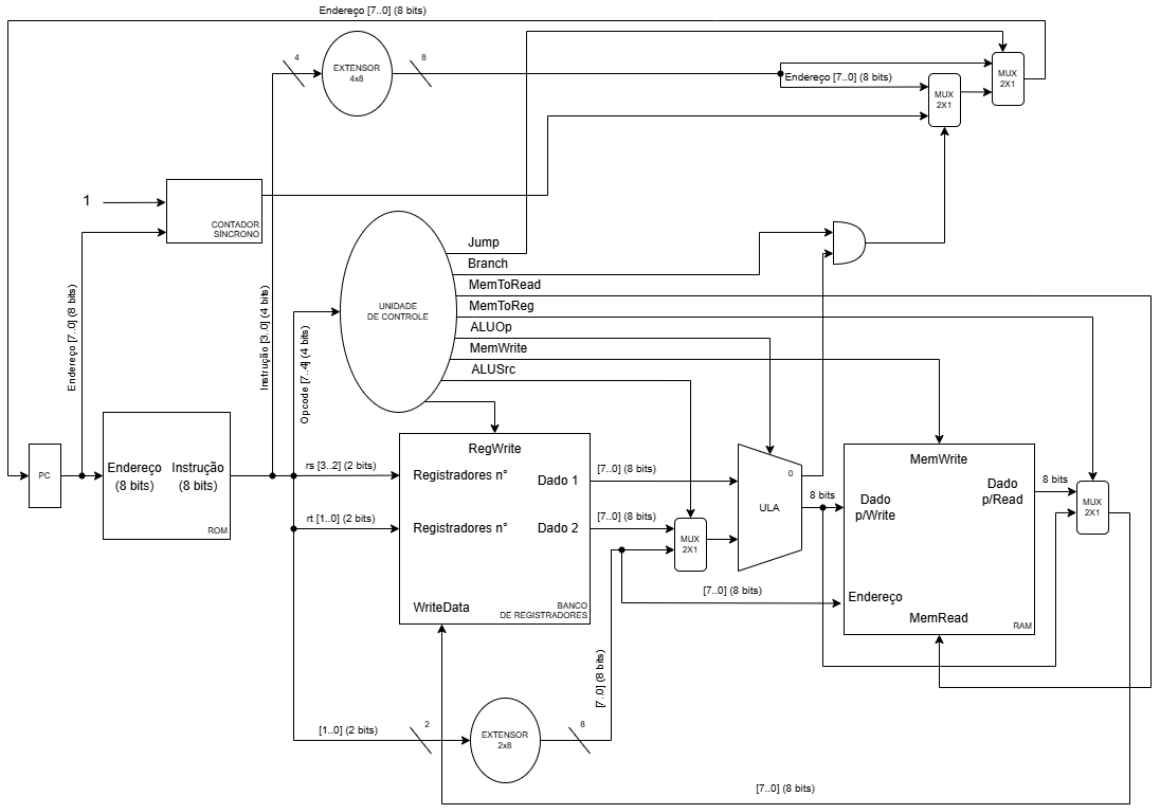
Figura 14 – RTL Viewer da UNIDADE_DE_CONTROLE



Fonte: Elaborada pelos autores.

1.4. Datapath

Figura 15 – Datapath

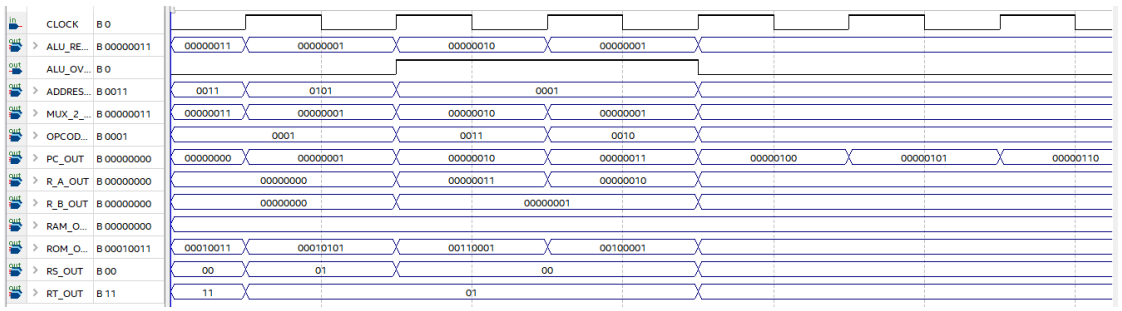


Fonte: Elaborada pelos autores.

2. Simulações e testes

2.1. Teste ADDI, SUB, SUBI

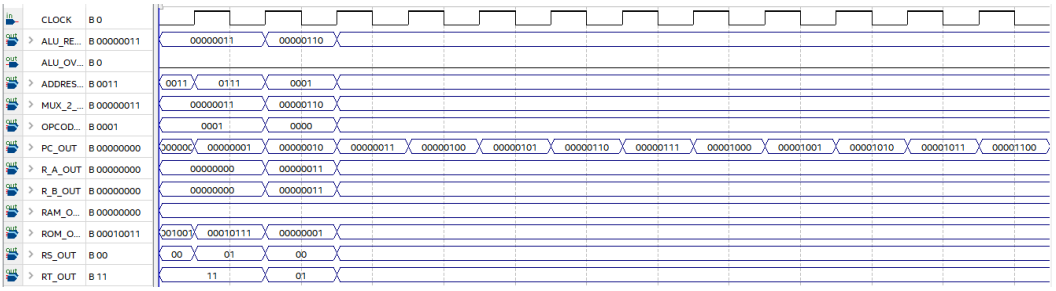
```
-- TESTE DE SUB E SUBI
        0 => "00010011", -- ADDI S0 3
        1 => "00010101", -- ADDI S1 1
        2 => "00110001", -- SUBI S0 1
        3 => "00100001", -- SUB S0 S1
```



Fonte: Elaborada pelos autores.

2.2. Teste ADD e ADDi

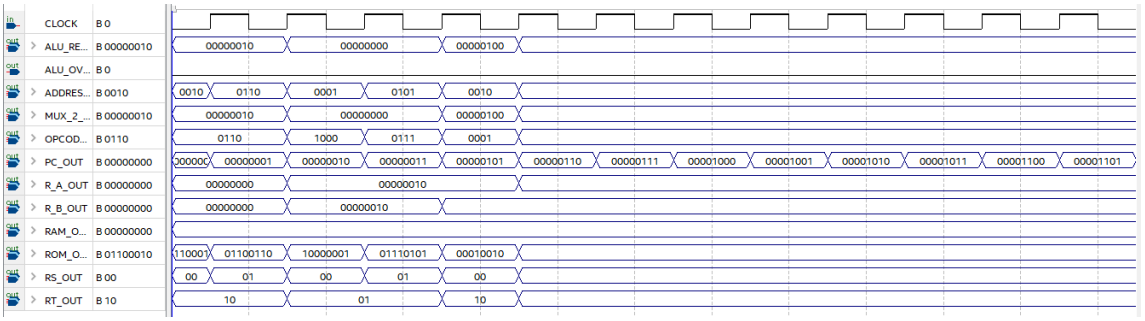
```
-- TESTE DE ADD E ADDI
        0 => "00010011", -- ADDI S0 3
        1 => "00010111", -- ADDI S1 3
        2 => "00000001", -- ADD S0 S1
```



Fonte: Elaborada pelos autores.

2.3. Teste BEQ

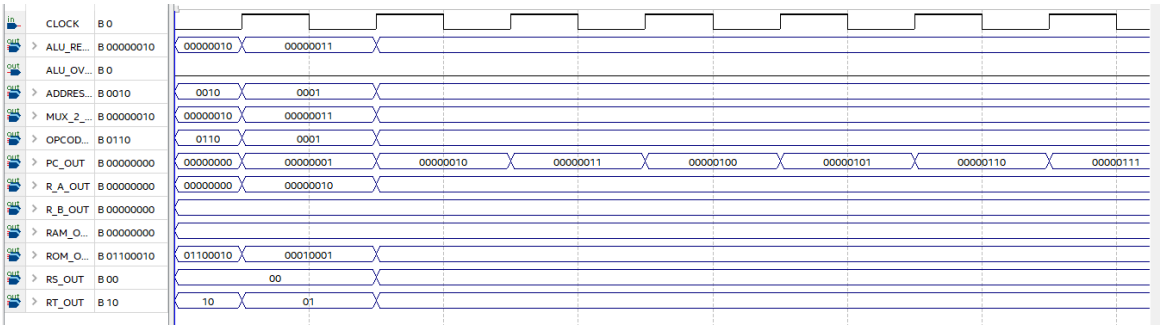
```
-- TESTE BEQ
        0 => "01100010", -- LI S0 2
        1 => "01100110", -- LI S1 2
        2 => "10000001", -- IF S0 == S1
        3 => "01110101", -- BEQ S0 == S1 JUMP 0101
        4 => "00010001", -- ADDI S0 1
        5 => "00010010", -- ADDI S0 2
```



Fonte: Elaborada pelos autores.

2.4. Teste LI

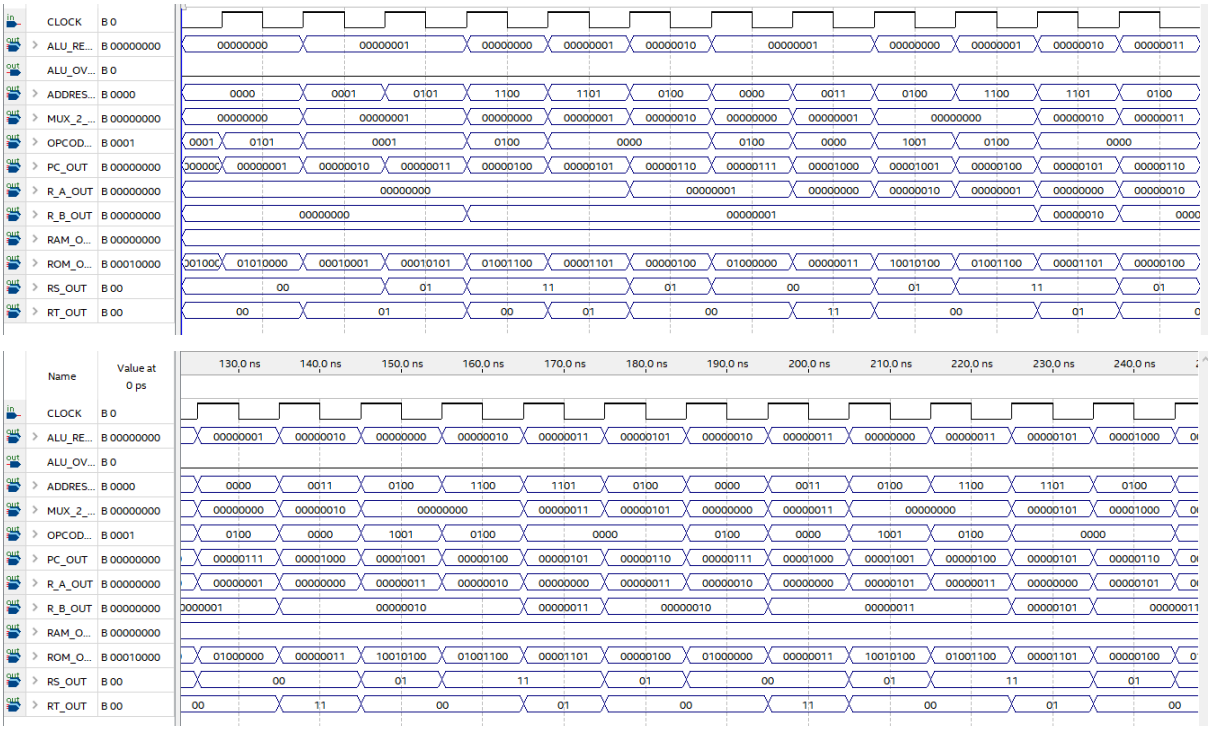
```
-- TESTE LI
    0 => "01100010", -- LI S0 2
    1 => "00010001", -- ADDI S0 1
```



Fonte: Elaborada pelos autores.

2.5. Teste Fibonacci

```
-- TESTE FIBONACCI
    0 => "00010000", -- ADDI S0 0
    1 => "01010000", -- SW S0
    2 => "00010001", -- ADDI S0 1
    3 => "00010101", -- ADDI S1 1
    4 => "01001100", -- LW S3 00
    5 => "00001101", -- ADD S3 S1
    6 => "00000100", -- ADD S1 S0
    7 => "01000000", -- LW S0 00
    8 => "00000011", -- ADD S0 S3
    9 => "10010100", -- J 0100
```



Fonte: Elaborada pelos autores.

3. Considerações finais

Este trabalho apresentou o projeto e a implementação do processador de 8 bits denominado LM, proporcionando uma valiosa oportunidade para aplicar na prática os conhecimentos adquiridos na disciplina de AOC e esclarecer diversos pontos que antes eram de difícil compreensão.

4. Repositório

https://github.com/marcusv0/AOC_MarcusViniciusIgorPereira_UFRR_2024