



PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR NEO

ALUNOS:

Gabriel Carvalho de Araújo - 2201524449

Hermínio Barbosa de Freitas Júnior - 2201524475

Jeovane Araujo da Silva – 2201524420

Janeiro de 2018
Boa Vista/Roraima



PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR NEO

Janeiro de 2018
Boa Vista/Roraima

Resumo

Neste trabalho será abordado os principais pontos da construção e implementação do processador multiciclo NEO de 8 bits baseado na arquitetura MIPS. Será descrito com detalhes cada etapa do processo de sua construção levando em conta todos os componentes necessários para o seu funcionamento e os testes realizados durante a implementação. O processador terá capacidade de executar 16 instruções, já incluso, a soma de ponto flutuante, dando a possibilidade bastante abrangente de executar algoritmos.

A implementação do processador foi feita integralmente com a linguagem de descrição de hardware VHDL e os teste foram analisados através simulador ModelSim que gera waveforms, demonstrando assim o comportamento do processador.

Conteúdo

Sumário

1	Especificação	7
2	Plataforma de desenvolvimento	7
3	Conjunto de instruções	7
3.1	Tipo de Instruções:	8
3.2	Visão geral das instruções do Processador NEO:	9
4	Descrição do Hardware	9
4.1	Registrador Flip Flop.....	9
4.2	Memória de Instrução	10
4.3	Banco de Registradores.....	11
4.4	Extensor de Sinal de 2 para 8 bits	12
4.5	Extensor de Sinal de 4 para 8 bits.	12
4.6	Multiplexador de 2 entradas.....	13
4.7	Multiplexador de 3 entrada	13
4.8	ULA	14
4.9	Somador de Ponto Flutuante	15
4.10	Memória de Dados.....	17
4.11	Unidade de Controle	18
4.12	Bloco Operativo.....	20
5	Datapath.....	21
5.1	Datapath idealizado	22
5.2	Datapath.....	22
6	Simulações e Testes	23
6.1	N-ésimo termo de uma P.A:	23
6.2	Fatorial:	25
6.3	Verificação dos resultados no relatório da simulação	Erro! Indicador não definido.
7	Considerações finais.....	27

Lista de Tabelas

Tabela 1 – Tabela que mostra o formato da instrução do tipo R.....	8
Tabela 2 - Tabela que mostra a separação dos bits para instruções do tipo R.	8
Tabela 3 - Tabela que mostra o formato da instrução do tipo J.	8
Tabela 4 - Tabela que mostra a separação dos bits do tipo J.....	8
Tabela 5 - Tabela que mostra a lista de Opcodes utilizadas pelo processador NEO.	9
Tabela 6 - Organização de ponto flutuante para 32 bits.....	16
Tabela 7 - Adaptação e organização de ponto flutuante para 8 bits.	16
Tabela 8 - Demonstração da adaptação do ponto flutuante parte 1.	16
Tabela 9 - Demonstração da adaptação do ponto flutuante parte 2.	17
Tabela 10 - Código N-ésimo termo de uma P.A. para o processador NEO.	23
Tabela 11 - Código Fatorial para o processador NEO.....	25
Tabela 12 - Código para calcular a soma de ponto flutuante do processador NEO.	26

Lista de Figura

Figura 1 - Especificações no Quartus.	7
Figura 2 - Trecho de código da entidade do componente Flip Flop.	9
Figura 3 - RTL view do componente Flip Flop gerado pelo Quartus.	10
Figura 4 - Trecho de código da entidade do componente Memória de Instrução.	10
Figura 5 - RTL view do componente Memória de Instrução gerado pelo Quartus.	10
Figura 6 - Trecho de código da entidade do componente Banco de Registradores.	11
Figura 7 - RTL view do componente Banco de Registradores gerado pelo Quartus.	11
Figura 8 - Trecho de código da entidade do componente Extensor de Sinal de 2 para 8 bits.	12
Figura 9 - RTL view do componente Extensor de Sinal de 2 para 8 bits gerado pelo Quartus.	12
Figura 10 - Trecho de código da entidade do componente Extensor de Sinal de 4 para 8 bits.	12
Figura 11 - RTL view do componente Extensor de Sinal de 4 para 8 bits gerado pelo Quartus.	12
Figura 12 - Trecho de código da entidade do componente Multiplexador de 2 entradas.	13
Figura 13 - RTL view do componente Multiplexador de 2 entradas gerado pelo Quartus.	13
Figura 14 - Trecho de código da entidade do componente Multiplexador de 3 entradas.	13
Figura 15 - RTL view do componente Multiplexador de 3 entradas gerado pelo Quartus.	14
Figura 16 - Trecho de código da entidade do componente Multiplexador de 3 entradas.	14
Figura 17 - RTL view do componente ULA gerado pelo Quartus.	15
Figura 18 - Trecho de código do Somador de Ponto Flutuante.	15
Figura 19 - RTL view do componente Somador de Ponto Flutuante gerado pelo Quartus.	17
Figura 20 - Trecho de código da Memória de Dados.	17
Figura 21 - RTL view do componente Memória de dados gerado pelo Quartus.	18
Figura 22 - Trecho de código da Unidade de Controle.	18
Figura 23 - RTL view do componente Unidade de Controle gerado pelo Quartus.	19
Figura 24 - Máquina de Estados da Unidade de Controle gerada pelo Quartus.	19
Figura 25 - Trecho de código do Bloco Operativo.	20
Figura 26 - RTL view do componente Bloco Operativo gerado pelo Quartus.	21
Figura 27 - Datapath idealizado após a construção dos componentes.	22
Figura 28 - Trecho de código datapath do processador NEO.	22
Figura 29 - RTL view do Datapath gerado pelo Quartus.	23
Figura 30 - Waveform do teste do algoritmo N-ésimo termo de uma P.A.	24
Figura 31 - Waveform do teste do algoritmo N-ésimo termo de uma P.A.	25
Figura 32 - Waveform do teste do algoritmo para calcular fatorial.	26
Figura 33 - Waveform do teste do algoritmo de Soma de Ponto Flutuante.	27

1 Especificação

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador multiciclo NEO de 8 bits, bem como a descrição detalhada de cada etapa da construção do processador.

2 Plataforma de desenvolvimento

Para a implementação do processador NEO foi utilizado a IDE: Quartus Prime, versão 16.1, com o simulador ModelSim-Altera e toda a descrição do hardware foi feita com a linguagem VHDL.

Summary	
When you click Finish, the project will be created with the following settings:	
Project directory:	C:/Users/hermi/Desktop/processador
Project name:	processador
Top-level design entity:	processador
Number of files added:	0
Number of user libraries added:	0
Device assignments:	
Design template:	n/a
Family name:	Cyclone V (E/GX/GT/SX/SE/ST)
Device:	5CGXFC7C7F23C8
Board:	n/a
EDA tools:	
Design entry/synthesis:	<None> (<None>)
Simulation:	ModelSim-Altera (VHDL)
Timing analysis:	()
Operating conditions:	
Core voltage:	1.1V
Junction temperature range:	0-85 °C

Figura 1 - Especificações no Quartus.

3 Conjunto de instruções

O processador NEO possui 4 registradores no banco de registradores: S0, S1, S2, S3. Assim como 2 formatos de instruções de 8 bits cada, instruções do tipo **R** e **J**, seguem algumas considerações sobre as estruturas contidas nas instruções:

- **Opcode:** a operação básica a ser executada pelo processador, tradicionalmente chamado de código de operação;
- **RS:** o registrador contendo o primeiro operando fonte e adicionalmente para alguns tipos de instruções (ex. instruções do tipo R) é o registrador de destino;

- **RT:** o registrador contendo o segundo operando fonte;
- **IMM:** endereço de memória, para instruções do tipo jump condicional e incondicional.

3.1 Tipo de Instruções:

- **Formato do tipo R:** Este formato aborda instruções de Load, Store e instruções baseadas em operações aritméticas e instruções aritméticas imediatas.

Formato para escrita de código na linguagem NEO:

Tabela 1 – Tabela que mostra o formato da instrução do tipo R.

Tipo da Instrução	RS	RT
-------------------	----	----

Formato para escrita em código binário:

Tabela 2 - Tabela que mostra a separação dos bits para instruções do tipo R.

OPCODE	RS	RT	IMM
7-4	3-2	1-0	-
4	2	2	-

- **Formato do tipo J:** Este formato aborda instruções do tipo jump condicional e incondicional.

Formato para escrita de código na linguagem NEO:

Tabela 3 - Tabela que mostra o formato da instrução do tipo J.

Tipo da Instrução	LABEL
-------------------	-------

Formato para escrita em código binário:

Tabela 4 - Tabela que mostra a separação dos bits do tipo J.

OPCODE	RS	RT	IMM
7-4	-	-	3-0
4	-	-	4

3.2 Visão geral das instruções do Processador NEO:

O número de bits do campo **Opcode** das instruções é igual a quatro, sendo assim obtemos um total $(Bit(0 \text{ e } 1))^4 \therefore 2^4 = 16$) de 16 **Opcodes (0-16)** que são distribuídos entre as instruções, assim como é apresentado na Tabela 5.

Tabela 5 - Tabela que mostra a lista de Opcodes utilizadas pelo processador NEO.

Tipo	OP	RS	RT	IMM	Exemplo	Instrução
R	0000	00-11	00-11	-	add \$s1, \$s2	add
R	0001	00-11	00-11	-	addi \$s1, WORD	addi
R	0010	00-11	00-11	-	sub \$s1, \$s2	sub
R	0011	00-11	00-11	-	subi \$s1, WORD	subi
R	0100	00-11	00-11	-	mult \$s1, \$s2	mult
R	0101	00-11	00-11	-	multi \$s1, WORD	multi
R	0110	00-11	00-11	-	eq \$s1, \$s2	eq
R	0111	00-11	00-11	-	eqi \$s1, WORD	eqi
R	1000	00-11	00-11	-	move \$s1, \$s2	move
R	1001	00-11	00-11	-	movi \$s1, WORD	movi
R	1010	00-11	00-11	-	lw \$s1, WORD	lw
R	1011	00-11	00-11	-	sw \$s1, WORD	sw
R	1100	00-11	00-11	-	addf \$s1, \$s2	addf
J	1101	-	-	0000-1111	bne Label	bne
J	1110	-	-	0000-1111	beq Label	beq
J	1111	-	-	0000-1111	j Label	j

4 Descrição do Hardware

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

4.1 Registrador Flip Flop

```
entity registrador is
  Port(
    clock,
    enable: in std_logic;
    input : in std_logic_vector(7 downto 0);
    output: out std_logic_vector(7 downto 0)
  );
end registrador;
```

Figura 2 - Trecho de código da entidade do componente Flip Flop.

O registrador flip flop funciona da seguinte maneira: Caso o **clock** esteja em borda alta e **enable** possuir valor 1, o valor da entrada **input** é registrado e o **output** recebe o **input**.

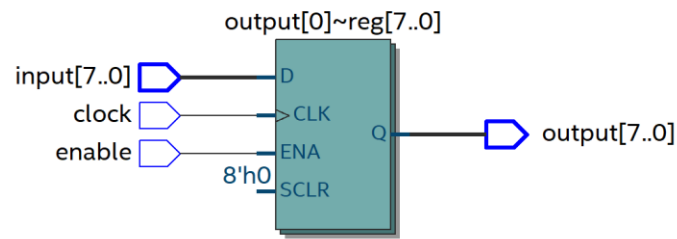


Figura 3 - RTL view do componente Flip Flop gerado pelo Quartus.

4.2 Memória de Instrução

```
entity memoriaInstrucao is
  port (
    clock : in std_logic;
    endereco: in std_logic_vector(7 downto 0);
    output : out std_logic_vector(7 downto 0)
  );
end memoriaInstrucao;
```

Figura 4 - Trecho de código da entidade do componente Memória de Instrução.

A memória de instrução funciona da seguinte maneira: Caso o **clock** esteja em borda alta, a saída **output** irá conter o valor da memória no endereço da entrada **endereco**.

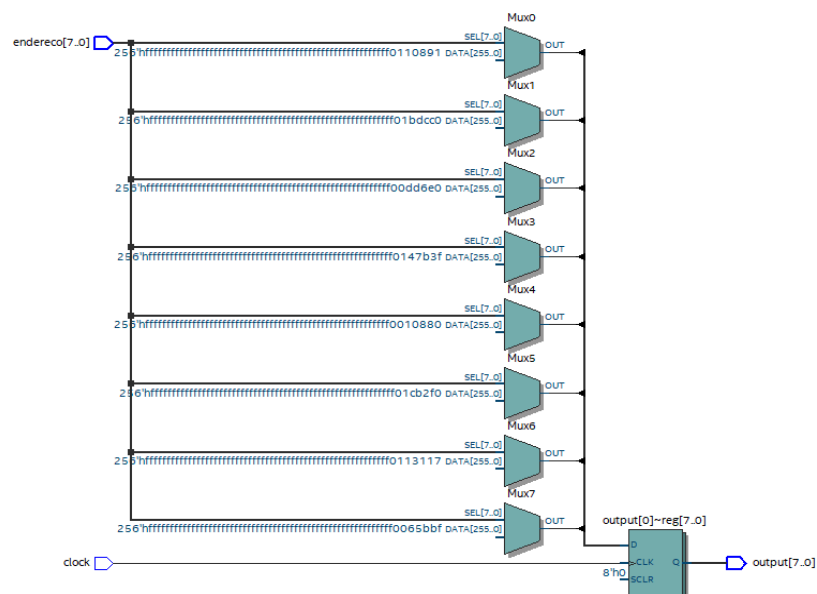


Figura 5 - RTL view do componente Memória de Instrução gerado pelo Quartus.

4.3 Banco de Registradores

```

entity bancoRegistadores is
port(
    clock,
    enablewrite : in std_logic;
    endereco_A,
    endereco_B : in std_logic_vector (1 downto 0);
    datain      : in std_logic_vector (7 downto 0);
    data_regA,
    data_regB   : out std_logic_vector(7 downto 0)
);
end bancoRegistadores;

```

Figura 6 - Trecho de código da entidade do componente Banco de Registradores.

O banco de registradores funciona da seguinte maneira: Funciona como um seletor de dados, caso o **enableWrite** esteja em 0, é procurado os dados registrados no **endereco_A** e **endereco_B**. Caso o **enableWrite** esteja em 1, o banco de registrador irá guardar o dado contido no **datain** no endereço **endereco_A** através do demultiplexador.

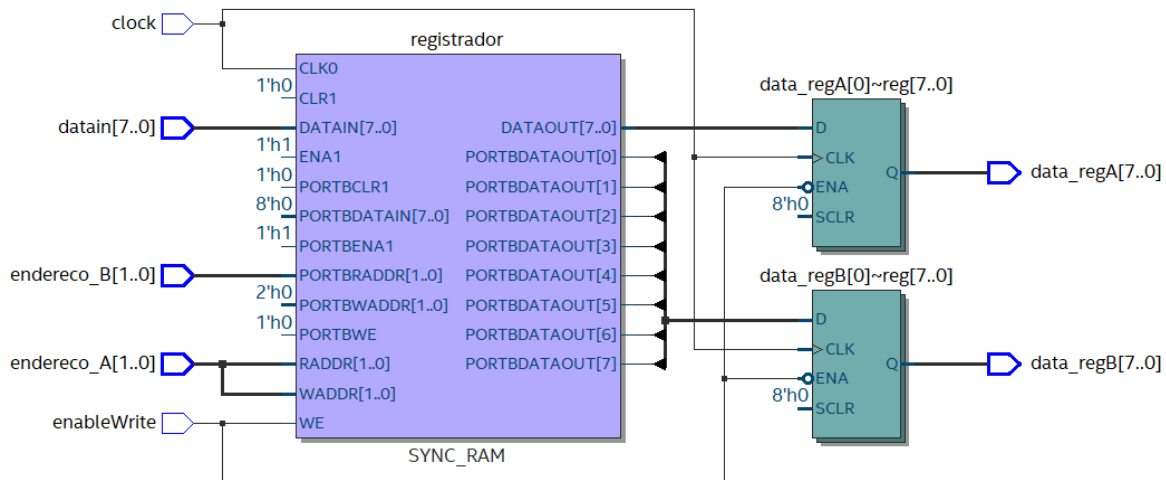


Figura 7 - RTL view do componente Banco de Registradores gerado pelo Quartus.

4.4 Extensor de Sinal de 2 para 8 bits

```
entity extensorSinal_2x8 is
  Port(
    input : in  std_logic_vector(1 downto 0);
    output: out std_logic_vector(7 downto 0)
  );
end extensorSinal_2x8;
```

Figura 8 - Trecho de código da entidade do componente Extensor de Sinal de 2 para 8 bits.

O extensor de sinal de 2 para 8 bits funciona da seguinte maneira: A entrada **input** com 2 bits será convertida para uma saída **output** com 8 bits.

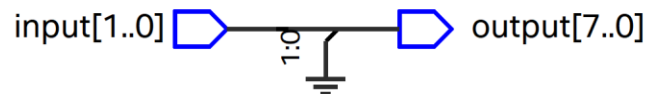


Figura 9 - RTL view do componente Extensor de Sinal de 2 para 8 bits gerado pelo Quartus.

4.5 Extensor de Sinal de 4 para 8 bits.

```
entity extensorSinal_4x8 is
  Port(
    input : in  std_logic_vector(3 downto 0);
    output: out std_logic_vector(7 downto 0)
  );
end extensorSinal_4x8;
```

Figura 10 - Trecho de código da entidade do componente Extensor de Sinal de 4 para 8 bits.

O extensor de sinal de 4 para 8 bits funciona da seguinte maneira: A entrada **input** com 4 bits será convertida para uma saída **output** com 8 bits.

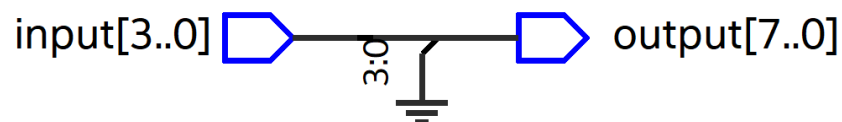


Figura 11 - RTL view do componente Extensor de Sinal de 4 para 8 bits gerado pelo Quartus.

4.6 Multiplexador de 2 entradas

```
entity mux_2x8 is
  port(
    selector: in  std_logic;
    E0,
    E1      : in  std_logic_vector(7 downto 0);
    output   : out std_logic_vector(7 downto 0)
  );
end mux_2x8;
```

Figura 12 - Trecho de código da entidade do componente Multiplexador de 2 entradas.

O multiplexador de 2 entradas funciona da seguinte maneira: É um componente onde terá 2 entradas e apenas uma saída que será definida através do **selector** dependendo do seu valor: caso o selector seja "00", a saída do MUX terá o valor de E0. Caso o seletor seja "01", a saída do MUX terá o valor de E1. Obs.: todos os casos necessitam do evento de borda alta no ciclo de clock.

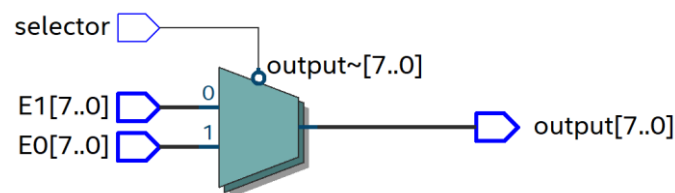


Figura 13 - RTL view do componente Multiplexador de 2 entradas gerado pelo Quartus.

4.7 Multiplexador de 3 entrada

```
entity mux_3x8 is
  port(
    selector: in  std_logic_vector(1 downto 0);
    E0,
    E1,
    E2      : in  std_logic_vector(7 downto 0);
    output   : out std_logic_vector(7 downto 0)
  );
end mux_3x8;
```

Figura 14 - Trecho de código da entidade do componente Multiplexador de 3 entradas.

O multiplexador de 2 entradas funciona da seguinte maneira: É um componente onde terá diversas entradas e apenas uma saída que será definida através do selector dependendo do seu

valor: caso o selector seja "00", a saída do MUX terá o valor de E0. caso o seletor seja "01", a saída do MUX terá o valor de E1. caso o seletor seja "10", a saída do MUX terá o valor de E2. Obs.: todos os casos necessitam do evento de borda alta no ciclo de clock

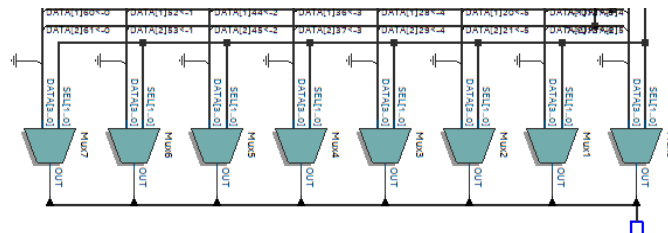


Figura 15 - RTL view do componente Multiplexador de 3 entradas gerado pelo Quartus.

4.8 ULA

```
entity ULA is
port(
    A,
    B      : in  std_logic_vector(7 downto 0);
    selector: in  std_logic_vector(3 downto 0);
    output  : out std_logic_vector(7 downto 0);
    zero    : out std_logic
);
end ULA;
```

Figura 16 - Trecho de código da entidade do componente Multiplexador de 3 entradas.

O componente ULA (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas, dentre elas: soma, subtração, multiplicação. Adicionalmente a ULA efetua operações de comparação de valor igual. O componente ULA recebe como entrada três valores: **A** – dado de 8bits para operação; **B** - dado de 8bits para operação e **OP** – identificador da operação que será realizada de 4bits. A ULA também possui duas saídas: **zero** – identificador de resultado (2bit) para comparações (1 se verdade e 0 caso contrário); e **result** – saída com o resultado das operações.

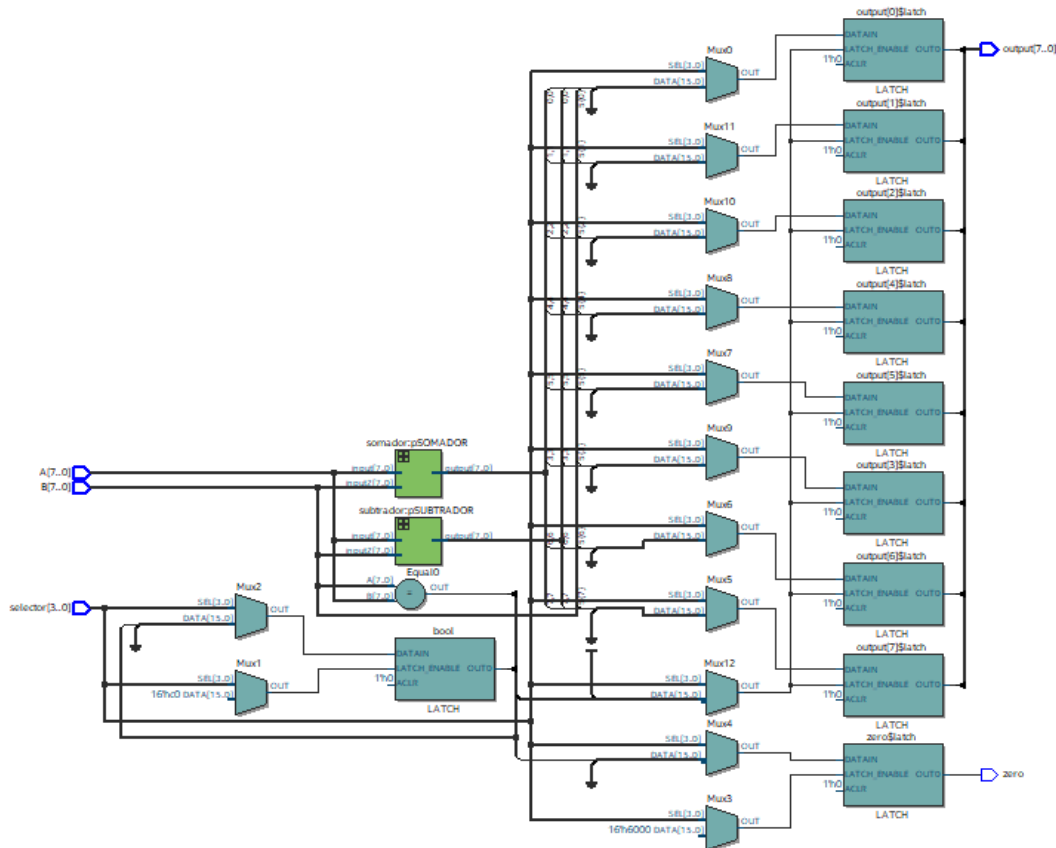


Figura 17 - RTL view do componente ULA gerado pelo Quartus.

4.9 Somador de Ponto Flutuante

Somador com entradas e saída usando a adaptação da representação para ponto flutuante que serão apresentadas neste documento.

```
entity ADD_PF_x8 is
    port(
        A : in std_logic_vector (7 downto 0);
        B : in std_logic_vector (7 downto 0);
        S : out std_logic_vector (7 downto 0)
    )
end ADD_PF_x8;
```

Figura 18 - Trecho de código do Somador de Ponto Flutuante.

O padrão IEEE 754 é uma forma de normalizar as operações com pontos flutuantes, tal padrão dá suporte apenas para arquiteturas de 16, 32 e 64 bits. Para um processador de 8 bits se fez necessário uma adaptação, que será mostrada a seguir.

Tabela 6 - Organização de ponto flutuante para 32 bits.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Expoiente											Significado																			

Para uma representação de 8 bits foi necessária uma distribuição onde há o máximo de precisão para essa arquitetura, foram distribuídos 3 bits para o expoente e 5 bits para o significando, nesse caso o bit de representação de sinal não foi levado em consideração, pois não há utilidade no processador NEO.

Tabela 7 - Adaptação e organização de ponto flutuante para 8 bits.

7	6	5	4	3	2	1	0
Expoente			Significando				

O número 5,3 em decimal equivale a $101,0100 \times 2^0$ em binário, normalizando este número ficaria:

$$10,1010 \times 2^1$$

$$-1,01010 \times 2^2$$

Agora o número está normalizado, pois há apenas um único 1 após a virgula. Para converter este número para a representação de 8 bits mostrada anteriormente são necessários passos simples:

Primeiro separamos o significando ignorando o 1 implícito em um binário normalizado.

Tabela 8 - Demonstração da adaptação do ponto flutuante parte 1.

7	6	5	4	3	2	1	0
			0	1	0	1	0
Expoente			Significando				

Depois é separado o expoente, o expoente deve sempre ser somado com 4, ou seja, $4 + \text{exp}$. Essa notação é chamada de Notação de excesso que é utilizada no padrão IEEE 754, logo, para o exemplo em questão ficaria $2 + 4 = 6$, esse número em binário é igual a 110.

Tabela 9 - Demonstração da adaptação do ponto flutuante parte 2.

7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	0
Expoente			Significando				

Logo abaixo podemos ver a estrutura do somador de ponto flutuante na RTL view:

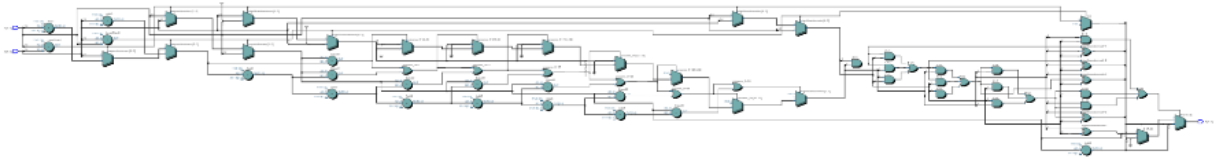


Figura 19 - RTL view do componente Somador de Ponto Flutuante gerado pelo Quartus.

4.10 Memória de Dados

```

entity MemoriaDados is
  port(
    clock,
    MemEscrita : in std_logic;
    endereco   : in std_logic_vector (1 downto 0);
    datain     : in std_logic_vector (7 downto 0);
    dataout    : out std_logic_vector (7 downto 0)
  );
end MemoriaDados;

```

Figura 20 - Trecho de código da Memória de Dados.

Podemos ver acima um trecho do código da memória de dados que demonstra todas as entradas e saídas. Basicamente ela funciona da seguinte maneira: quando houver um ciclo de borda alta no **clock** a escrita será habilitada, logo após o dado **datain** é armazenado no **endereco** estabelecido, no próximo ciclo de borda alta em que o **endereco** lido for o estabelecido obteremos o valor contido naquele endereço no **dataout**. Abaixo podemos ver a RTL view do componente especificado.

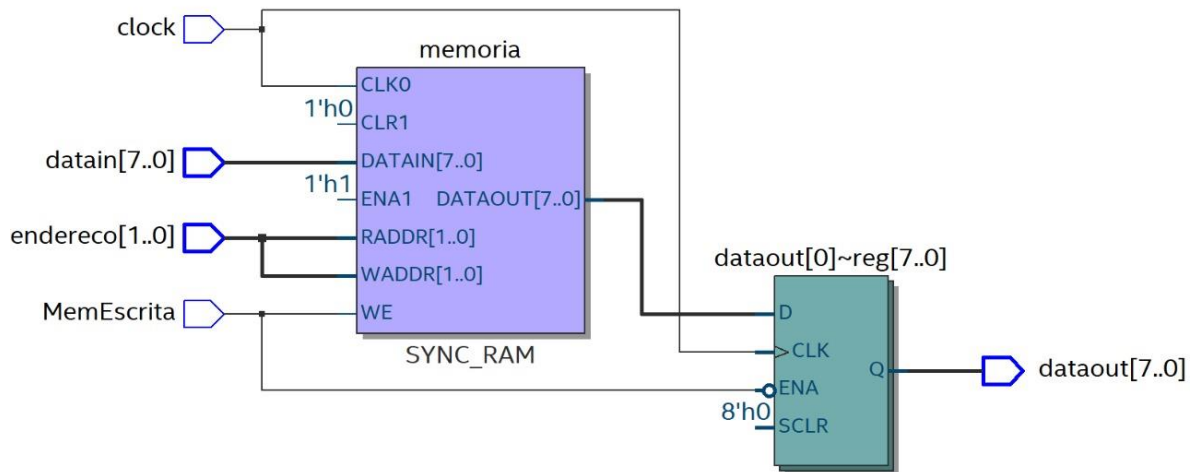


Figura 21 - RTL view do componente Memória de dados gerado pelo Quartus.

4.11 Unidade de Controle

```
entity blocoControle is
  port(
    clock,
    reset      : in std_logic;
    opcode     : in std_logic_vector(3 downto 0);
    PCescCond,
    PCesc,
    MemParaReg,
    regEscrita,
    ULAfonteA,
    MemLeitura,
    MemEscrita : out std_logic;
    ULAfonteB,
    PCfonte    : out std_logic_vector(1 downto 0);
    ULAop      : out std_logic_vector(3 downto 0);
    out_estado : out std_logic_vector(2 downto 0);
  );
end blocoControle;
```

Figura 22 - Trecho de código da Unidade de Controle.

Acima podemos ver um trecho do código da Unidade de Controle do processador NEO. A unidade de controle é componente responsável por ativar as flags e manter o fluxo correto das instruções, podendo suportar instruções do tipo R e J, e capaz de executar os estados básicos do mips, são eles: busca, decodificação, execução, cálculo de memória e escrita no registrador. Abaixo podemos ver na **Figura 23** a RTL view e na **Figura 24** a máquina de estados da unidade de controle.

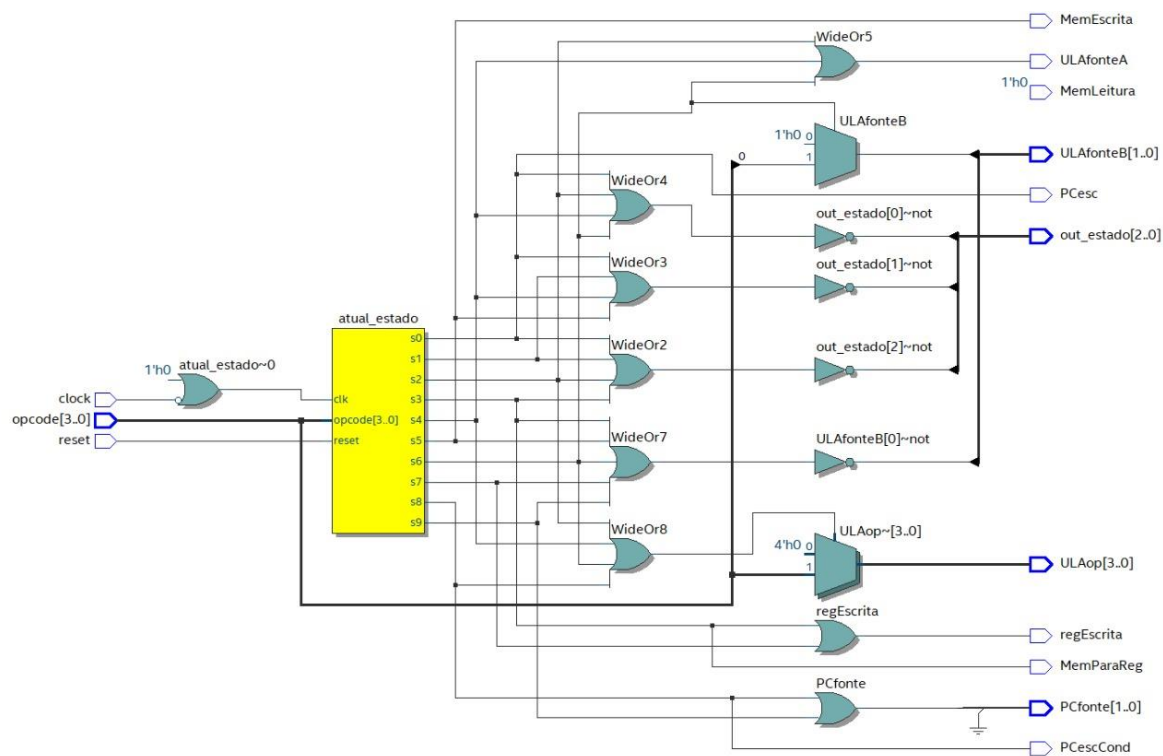


Figura 23 - RTL view do componente Unidade de Controle gerado pelo Quartus.

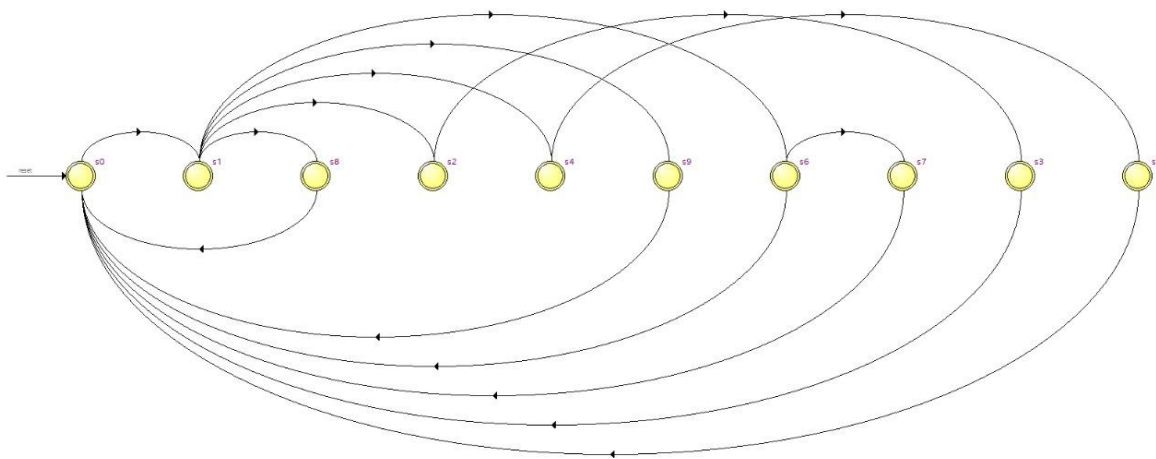


Figura 24 - Máquina de Estados da Unidade de Controle gerada pelo Quartus.

4.12 Bloco Operativo

```

entity blocoOperativo is
  port(
    clock: in std_logic;
    PCescCond,
    PCesc,
    MemParaReg,
    regEscrita,
    ULAfonteA,
    MemEscrita : in std_logic;
    ULAfonteB,
    PCfonte      : in std_logic_vector(1 downto 0);
    ULAop        : in std_logic_vector(3 downto 0);
    opcode: out std_logic_vector(3 downto 0);
    -- PORTAS PARA DEBUG --
    out_entrada_PC,
    out_saida_PC,
    out_saida_MemInstr,
    out_saida_BdRA,
    out_saida_BdRB,
    out_saida_Extensorsinal2p8,
    out_saida_Extensorsinal4p8,
    out_saida_RegA,
    out_saida_RegB,
    out_saida_MUXfonteA,
    out_saida_MUXfonteB,
    out_saida_ULA,
    out_saida_RegULAout,
    out_saida_MemDados,
    out_saida_MDR,
    out_saida_MUXbdRegin: out std_logic_vector(7 downto 0);
    out_habilitaPC,
    out_zeroULA: out std_logic
  );
end blocoOperativo;

```

Figura 25 - Trecho de código do Bloco Operativo.

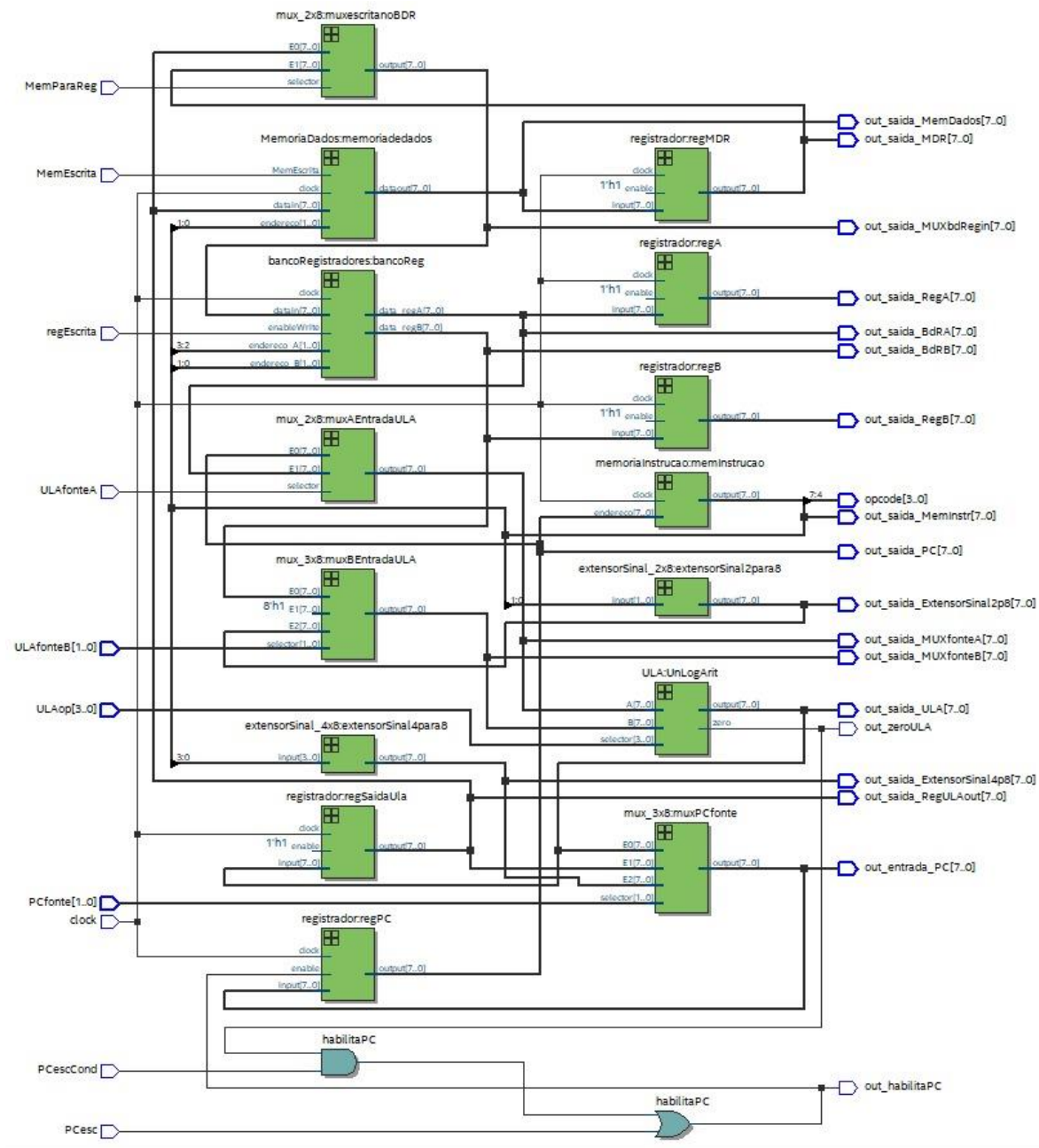


Figura 26 - RTL view do componente Bloco Operativo gerado pelo Quartus.

5 Datapath

É a conexão entre as unidades funcionais formando um único caminho de dados e acrescentando uma unidade de controle responsável pelo gerenciamento das ações que serão realizadas para diferentes tipos de instruções. Para o processador NEO foi decidido colocar a memória de dados e a memória de instruções, pois ambas possibilitam um gerenciamento melhor dos testes facilitando no momento da execução dos algoritmos que foram codificados.

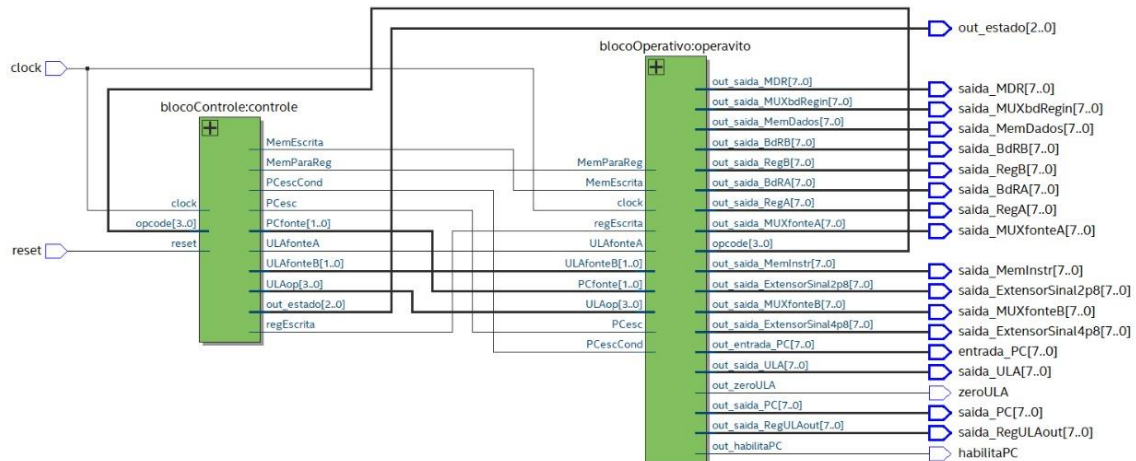


Figura 29 - RTL view do Datapath gerado pelo Quartus.

6 Simulações e Testes

Objetivando analisar e verificar o funcionamento do processador, efetuamos alguns testes analisando cada componente do processador em específico, em seguida efetuamos testes de cada instrução que o processador implementa. Para demonstrar o funcionamento do processador NEO utilizaremos como exemplo o código para calcular o N-ésimo termo de uma P.A, o fatorial de 3 e a soma de números de ponto flutuante. Os testes podem ser verificados logo abaixo.

6.1 N-ésimo termo de uma P.A:

Tabela 10 - Código N-ésimo termo de uma P.A. para o processador NEO.

Código	Endereço	Binário
main: movi \$s1, 3	0000	1001 00 11
addi \$s1, 3	0001	0001 00 11
addi \$s1, 3	0010	0001 00 11
addi \$s1, 1	0011	0001 00 01
movi \$s2, 3	0100	1001 01 11
addi \$s2, 2	0101	0001 01 10
movi \$s3, 3	0110	1001 10 11
loop: add \$s1, \$s3	0111	0000 00 10
subi \$s2, 1	1000	0011 01 01

eq \$s2, 0	1001	0111 01 00
bne loop	1010	1101 01 11

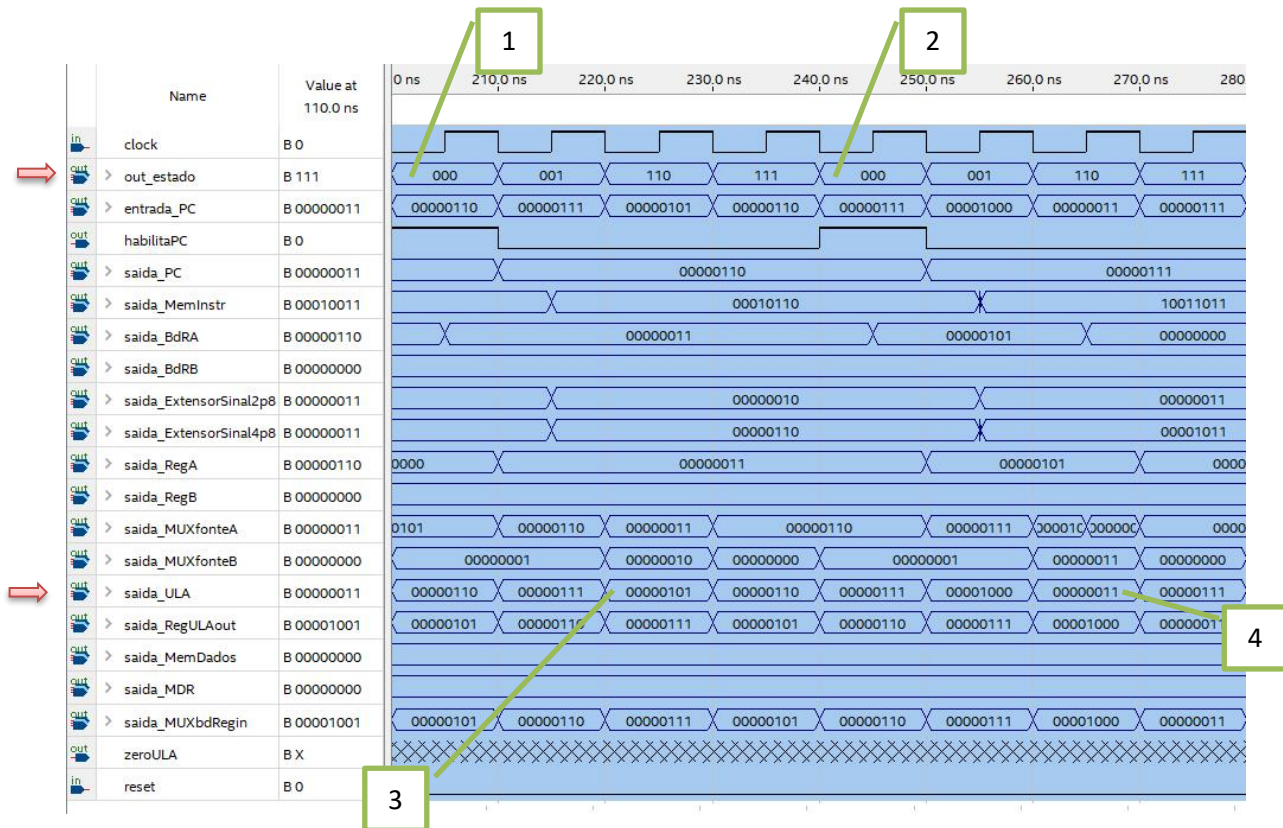


Figura 30 - Waveform do teste do algoritmo N-ésimo termo de uma P.A.

Na porta **out_estado**, podemos verificar em qual estado a instrução está, neste exemplo temos um trecho de onde é executado duas instruções do tipo R, no quadro 1, podemos verificar que a instrução (**addi \$s1, 2**) que está no estado S0 (busca da instrução e incremento do PC), passa pelo S1 (decodificação da instrução), passa pelo S6 (Execução da Instrução) e termina em S7 (Escrita no Registrador). O mesmo pode ser observado no quadro 2, onde começa a segunda instrução (**movi \$s2, 3**). No estado S6 (Execução da Instrução), podemos verificar na porta **saída_ULA**, no quadro 3 que o resultado da primeira instrução é cinco, e no quadro 4 é onde podemos verificar que o resultado da segunda instrução é três.

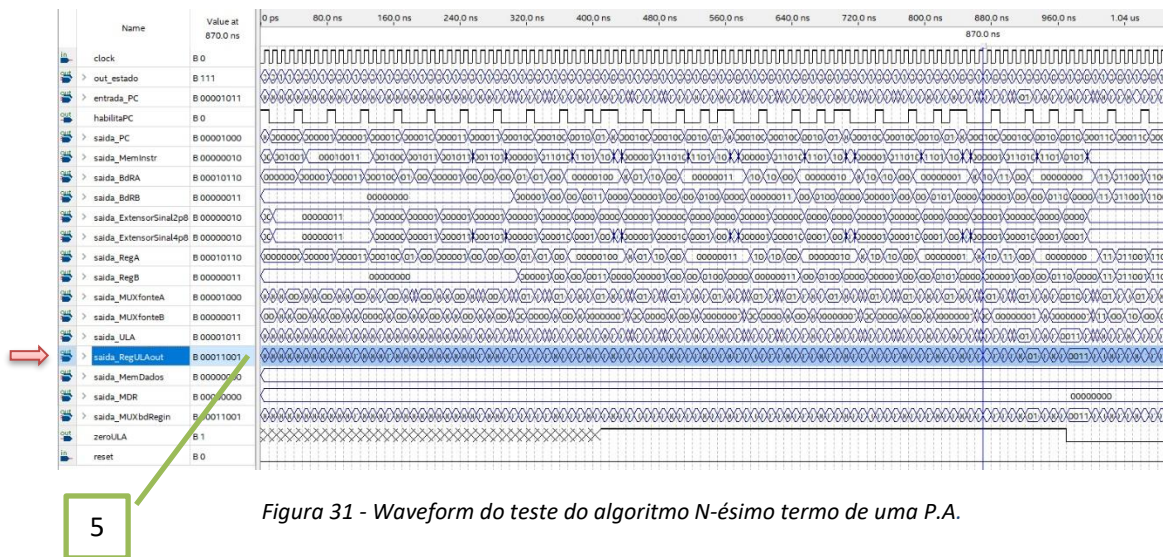


Figura 31 - Waveform do teste do algoritmo N-ésimo termo de uma P.A.

No quadro 5 podemos analisar na porta **saída_RegULAout** o resultado referente ao algoritmo do N-ésimo termo de uma P.A. Este algoritmo tem duração de 980ns.

6.2 Fatorial:

Tabela 11 - Código Fatorial para o processador NEO.

Código	Endereço	Binário
main: movi \$s0, 3;	0000	1001 00 11
move \$s1, \$s0;	0001	1000 01 00
subi \$s1, 1;	0010	0011 01 01
loop: mult \$s0, \$s1;	0011	0100 00 01
subi \$s1, 1;	0100	0011 01 01
eqi \$s1,0;	0101	0111 01 00
bne loop;	0110	1101 0110
exit:	-	-

Logo abaixo na Figura 32, podemos verificar os testes realizado com este algoritmo. No quadro 1 na porta **saída_RegULAout** podemos verificar o resultado obtido ao realizar o fatorial de 3. Neste algoritmo o tempo de duração de 400ns.

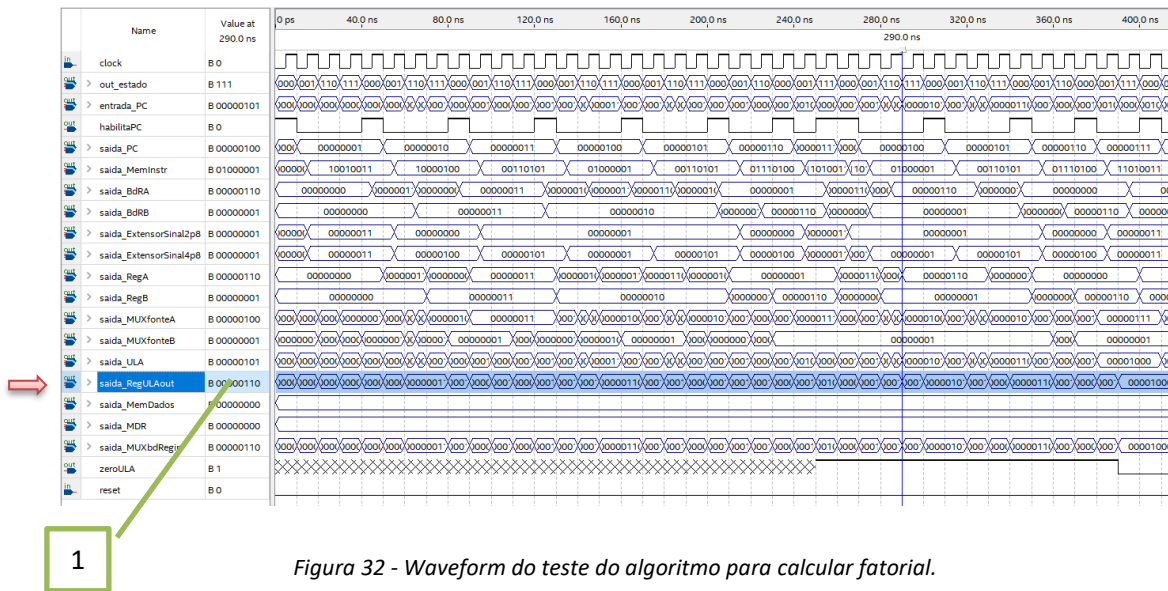


Figura 32 - Waveform do teste do algoritmo para calcular fatorial.

6.3 Soma Ponto Flutuante:

Tabela 12 - Código para calcular a soma de ponto flutuante do processador NEO.

Código	Endereço	Binário
main: movi \$s0, 3;	0000	1001 00 11
addi \$s0, 2;	0001	0001 00 10
movi \$s1, 2;	0010	1001 01 10
mult \$s0, \$s1;	0011	0100 00 01
addi \$s1, 3;	0100	0001 01 11
addi \$s1, 2;	0101	0001 01 10
move \$s2, \$s1;	0110	1000 10 01
mult \$s1, \$s2;	0111	0100 01 10
addf \$s0, \$s1;	1000	1100 00 01

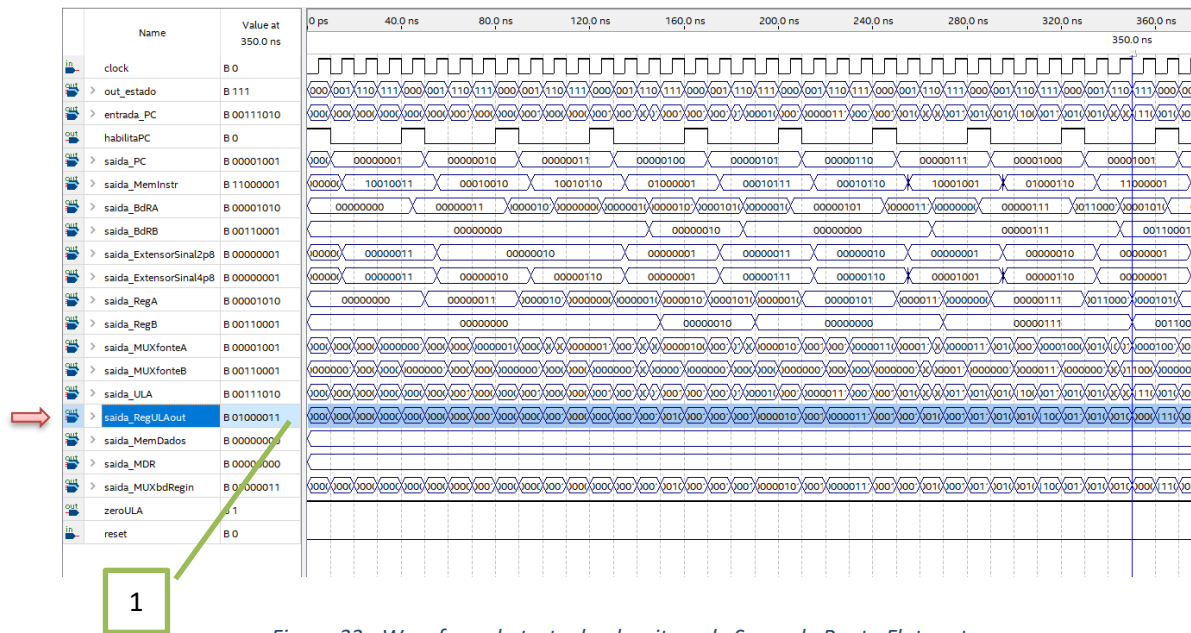


Figura 33 - Waveform do teste do algoritmo de Soma de Ponto Flutuante.

No quadro 1 podemos analisar o resultado na porta **saida_RegULAout** é 0,28 em decimal.

7 Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 8 bits denominado de NEO que é a junção dos conhecimentos adquiridos durante o período de ensino da disciplina Arquitetura e Organização de Computadores. Os conhecimentos passados pelo professor Herbert, foram suficientes para concluir a construção do processador com êxito.

O nome escolhido para o processador faz referência a trilogia matrix que tem como protagonista o personagem Neo interpretado pelo ator Keanu Reeves que possui a seguinte simbologia e significado: A palavra 'Neo' é um anagrama da palavra 'One'. Neo também é uma palavra latim que significa "novo", sugerindo assim uma pista para a sua missão na Matrix. Ainda no filme Neo é a pessoa escolhida da profecia, onde essa pessoa é chamada de "O escolhido" (The One). Por guarda tantos significados e por se tratar de uma referência a uma excelente trilogia escolhemos o nome NEO para o processador de 8 bits.