Mário Allan Lehmkuhl de Abreu

Implementação de processadores didáticos utilizando linguagem de descrição de hardware

São José – SC

Agosto / 2014

Mário Allan Lehmkuhl de Abreu

Implementação de processadores didáticos utilizando linguagem de descrição de hardware

Monografia apresentada à Coordenação do Curso Superior de Tecnologia em Sistemas de Telecomunicações do Instituto Federal de Santa Catarina para a obtenção do diploma de Tecnólogo em Sistemas de Telecomunicações.

Orientador:
Prof. Roberto de Matos, M. Eng.

CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS DE TELECOMUNICAÇÕES
INSTITUTO FEDERAL DE SANTA CATARINA

São José – SC

Agosto / 2014

Monografia sob o título *"Implementação de processadores didáticos utilizando linguagem de descrição de hardware"*, defendida por Mário Allan Lehmkuhl de Abreu e aprovada em 27 de Agosto de 2014, em São José, Santa Catarina, pela banca examinadora assim constituída:

Prof. Roberto de Matos, M. Eng.
Orientador
IFSC

Prof. Marcos Moecke, Dr.
IFSC

Prof. Eraldo Silveira e Silva, Dr.
IFSC

As grandes ideias surgem da observação
dos pequenos detalhes.
Augusto Cury

Agradecimentos

Primeiramente gostaria de agradecer a Deus pela vida e saúde que possuo. Depois aos meus pais, irmão, irmã e aos familiares pelo incentivo e suporte. A todos os amigos, colegas e professores que contribuíram durante todo o curso. Agradeço também ao meu orientador por aceitar me orientar, pela boa metodologia de pesquisa e por sempre incentivar pela busca de novos conhecimentos. Assim agradeço a todos, pois através de todas essas ações foi possível a realização desse TCC.

Resumo

Todos os anos vários processadores são criados para desempenhar de maneira mais eficiente, diversas ou especificas funções nos equipamentos eletrônicos. Os processadores desde os mais simples até os mais sofisticados são todos compostos por um conjunto de características de funcionamento e por componentes interligados. Na linguagem do processador esse conjunto de características e componentes é chamado respectivamente de arquitetura e organização do processador. A arquitetura é a parte mais abstrata do processador, é a parte que um programador enxerga, isto é, mostram quais são as instruções que são suportadas, com quantos bits essas instruções trabalham, quais as funções são executadas entre outras informações. A organização é a parte mais detalhada, ela mostra como os componentes que formam o processador são interligados e como as instruções do processador são executadas apartir desses componentes. Contudo qualquer processador que é criado independente da tarefa que vai executar possui como base uma arquitetura e uma organização básica. Dessa forma em disciplinas como arquitetura de computadores, microprocessadores ou outras disciplinas do gênero, para que os alunos tenham uma noção de como um processador funciona, os professores utilizam para o estudo a arquitetura e organização básica que formam a base do mesmo. Entretanto esse estudo possui um foco maior na parte teórica do assunto, deixando uma lacuna na parte prática. Ela até possui alguma demonstração, mas é direcionada apenas na arquitetura do processador. Assim a implementação de processadores didáticos utilizando linguagem de descrição de hardware é um trabalho que demonstra na prática através da linguagem de hardware VHDL como é o funcionamento básico de um processador, com o uso de processadores didáticos apartir das suas arquiteturas e organizações. Com isso o trabalho permite que os aluno e/ou professor das disciplinas mencionadas tenham uma noção completa de como um processador funciona basicamente. Os processadores didáticos escolhidos são chamados de Neander e Ramses que pertencem a um conjunto de processadores didáticos criados para fins de ensino por um grupo de professores da Universidade Federal do Rio Grande do Sul (UFRGS). Todos eles são formados pela arquitetura e organização básica que formam os processadores. No entanto cada um possui diferenças de complexidade que aumentam gradativamente em relação a cada processador. Neander e Ramses são os menos complexos do conjunto respectivamente. Para implementação/simulação foi utilizada à linguagem de descrição de hardware VHDL [VHSIC (Very High Speed Integrated Circuits) Hardware Description Language], que permite descrever e simular circuitos digitais, os quais formam os componentes básicos dos processadores. Além disso, por ser uma tecnologia independente de fabricante, o código gerado é portável e reutilizável em diversas tecnologias de prototipação de lógica configurável.

Palavras chaves: processador didático, VHDL.

Abstract

Every year many processors are designed to perform more efficiently, or several specific functions in electronic equipment. Processors from the simplest to the most sophisticated are all composed of a set of operating characteristics and interconnected components. In the language of the processor that set of features and components are called respectively the processor architecture and organization. Architecture is the most abstract of the processor is the part that a programmer sees, that is, show what are the instructions that are supported, how many bits these instructions work, what functions are performed among other information. The organization is the most detailed part, it shows how the components that make up the processor are interconnected and how processor instructions are executed starting these components. However any processor that is created independently of the task that will have to run as a base architecture and a basic organization. Thus in disciplines such as computer architecture, microprocessors or other subjects of gender, so that students have a sense of how a processor works, the teachers use to study the architecture and basic organization that form the basis of the same. However this study has a greater focus on the theoretical part of the subject, leaving a gap in the practical part. She even has some demonstration, but it is directed only at the processor architecture. Thus the implementation of teaching processors using hardware description language is a work that shows in practice by VHDL hardware language like is the basic operation of a processor, using didactic processors starting their architectures and organizations. With this work allows the student and / or teacher of the mentioned disciplines have a complete understanding of how a processor works basically. The chosen didactic processors are called Neander and Ramses belonging to a set of instructional processors created for educational purposes by a group of professors from the Federal University of Rio Grande do Sul (UFRGS). They are all formed by the architecture and basic organization forming processors. However each has different complexity that increase gradually in each processor. Neander and Ramses are the least complex of the set respectively. For implementation / simulation was used to hardware description language VHDL [VHSIC (Very High Speed Integrated Circuits) Hardware Description Language], which allows to describe and simulate digital circuits, which form the basic components of processors. Moreover, being an independent technology manufacturer, the generated code is portable and reusable in various prototyping configurable logic technology

Key words: teaching processor, VHDL.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AC - Acumulador

C - Carry

CI - Circuito Integrado

CPU - Central Processing Unit

FMS - Finite State Machine

HDL - Hardware Description Language

IEEE - Instituto de Engenheiros Eletricistas e Eletrônicos

MSB - Most Significant Bit

N - Negativo

PC - Contador de Programa

RDM - Registrador de Dados de Memória

REM - Registrador de Endereços de Memória

RI - Registrador de Instrução

RST - Registrador de Estado

UC - Unidade de Controle

UFRGS - Universidade Federal do Rio Grande do Sul

ULA - Unidade Lógica e Aritmética

UPF - Unidade de Ponto Flutuante

V - Overflow

VHDL - VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

Z - Zero

Sumário

1 Introdução

Todos os anos vários processadores são criados para desempenhar de maneira mais eficiente, diversas ou especificas funções nos equipamentos eletrônicos. Os processadores desde os mais simples até os mais sofisticados são todos compostos por um conjunto de características de funcionamento e de componentes interligados. O processador é um circuito integrado que tem como característica a função de calcular, endereçar, resolver ou preparar um volume de dados representados em um determinado numero de bits. Ele trabalha apenas com linguagem de máquina, isto é, padrão binário (0 e 1). Para a execução de um programa, o mesmo busca, decodifica e executa cada instrução armazenada em uma memória. Este ciclo é sempre repetido até que seja executada uma instrução que pare o processamento. Na linguagem do processador essas características de funcionamento são chamadas de arquitetura do processador e o modo como os componentes que formam o mesmo são interligados é chamado de organização do processador. Cada processador possui sua arquitetura e organização que na hora da fabricação irá determinar quantas instruções, funções e componentes ele possuirá. Entretanto todos os processadores possuem como base uma arquitetura e organização básica.

Seguindo essa filosofia, professores da Universidade Federal do Rio Grande do Sul (UFRGS), que trabalham com as disciplinas de Arquitetura de Computadores e Microprocessadores, criaram apartir dessa arquitetura e organização básica uma série de processadores didáticos para melhorar o ensinamento do assunto aos alunos dessas disciplinas. Os processadores foram chamados de Neander, Ahmes, Ramses e Cesar, no qual cada um possui um nível de complexidade diferente. Para estudo ambos possuem uma arquitetura e organização aberta. Porém o foco é voltado mais para a parte teórica, deixando uma lacuna na parte prática. Ela até possui alguma demonstração, mas é direcionada apenas para as arquiteturas desses processadores no qual eles possuem ferramentas para criar programas (Montador Assembler) utilizando suas instruções e simuladores que decodificam esses programas. Contudo, esses processadores não demonstram como o ciclo de busca, decodificação e execução das instruções ocorrem através das suas organizações.

Nesse trabalho os processadores escolhidos para a implementação seriam o Neander e o Ahmes, mas devido à escassez do tempo o único processador implementado e simulado foi o Neander. Dessa forma, esse trabalho propõem a implementação do processador didático Neander para que qualquer aluno e/ou professor das disciplinas de microprocessadores, arquitetura de computadores e outras disciplinas do gênero consigam entender de uma forma mais clara como é o funcionamento básico de um processador e como ele realmente processa as informações através da sua arquitetura e organização. Os alunos apartir desses conhecimentos também poderão criar programas para executar tarefas reais com o mesmo.

Para implementação do Neander utilizou-se a linguagem de descrição de hardware VHDL, que descrever e simula circuitos digitais, os quais formam os componentes básicos dos processadores. Além disso, por ser uma tecnologia independente de fabricante, o código gerado é portável e reutilizável em diversas tecnologias de prototipação de lógica configurável.

1.1 Motivação

A motivação desse trabalho foi o interesse em entender de forma mais detalhada o funcionamento básico de um processador, isto é, como ele realmente processa na as informações através da sua arquitetura e organização. Porque em disciplinas como arquitetura de computadores, microprocessadores ou outras disciplinas do gênero, para que os alunos tenham uma noção de como um processador funciona, os professores utilizam para o estudo a arquitetura e organização básica que formam a base do mesmo. Entretanto esse estudo possui um foco maior na parte teórica do assunto, deixando uma lacuna na parte prática. Ela até possui alguma demonstração, mas é direcionada apenas na arquitetura do processador que utiliza montadores assembler para a criação de programas e depois usa simuladores desses processadores para interpretar os programas criados. Contudo não é mostrado como essa interpretação ocorreu através de todos os seus componentes, ou seja, através da sua organização.

1.2 Objetivos

O objetivo geral do trabalho é entender o funcionamento básico de um processador através da arquitetura e organização do processador didático Neander e implementar/simular o mesmo apartir da linguagem de hardware VHDL. Assim, alunos e professores de disciplinas como arquitetura de computadores, microprocessadores ou outras disciplinas do gênero irão ter uma visão mais completa de como esse processo realmente ocorre. Como objetivo específico, serão

criados alguns programas para serem executados no processador Neander.

1.3 Organização do texto

O texto está organizado da seguinte forma: No capítulo 2 são apresentados os principais conceitos e componentes da área de processadores. Em seguida são exibidas informações sobre os processadores didáticos. No final do capítulo são mostradas as características da linguagem de descrição de hardware VHDL. No capítulo 3 é descrito a arquitetura do processador didático Neander, isto é, é apresentado as suas características de funcionamento, que tipos de instruções ele suporta e como um programa é formado com essas instruções. O capítulo 4 apresenta a organização do Neander, ou seja, é mostrado como os componentes que compõe o processador são interligados e como as instruções são executadas a partir desses componentes. No final são informados quais os tempo de clock que cada instrução necessita para que a execução da instrução como um todo ocorra de forma sincronizada com os componentes utilizados. Também é explicado como é formada a máquina de estado que compõe a unidade de cotrole do processador Neander. No capítulo 5 é exposto as etapas da implementação em VHDL de todos os componentes que constituem o processador Neander. O capítulo 6 exibe as simulações e os resultados da implementação do Neander e seus componentes. Finalmente, no capítulo 7 são apresentadas as conclusões e trabalhos futuros.

2 *Fundamentação Teórica*

Neste capítulo serão apresentados os principais conhecimentos envolvidos no trabalho. A primeira seção descreve com mais profundidade os conceitos genéricos da área de processadores. A segunda seção faz um levantamento das informações relacionadas aos processadores didáticos e por fim será descrito a linguagem de descrição de hardware VHDL.

2.1 Microprocessador

Microprocessador ou CPU (Central Processing Unit) é basicamente um CI (Circuito Integrado) composto por um conjunto de características de funcionamento e por um conjunto de componentes interligados como registradores e unidades de execução e controle. Mas para executar as informações além dos componentes internos, o processador também se interliga com componentes externos tais como sinais de controle e memória. A figura 2.1 ilustra como é a composição básica de um processador.

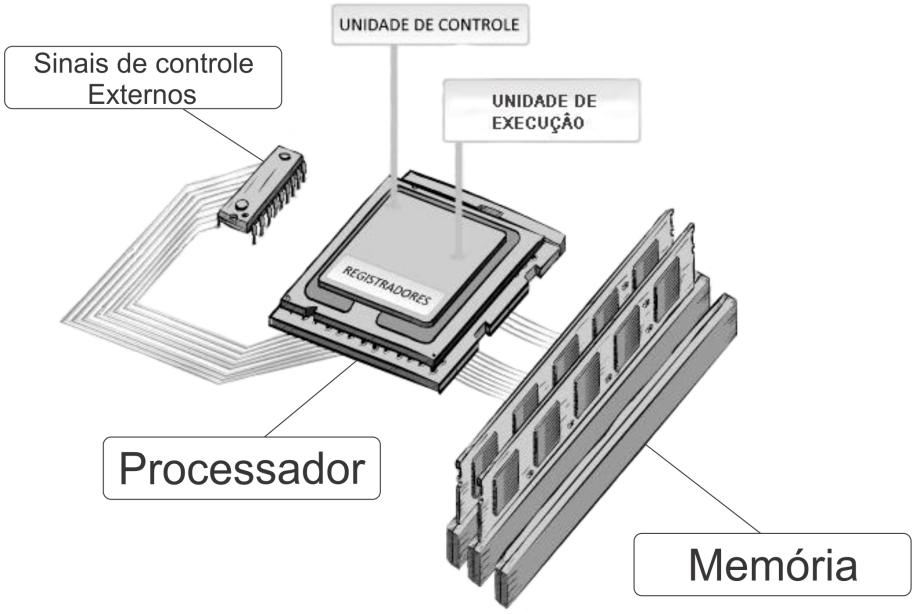

Figura 2.1: Composição básica de um processador.

As principais funções que uma CPU realiza são a busca de instruções e de dados na memória, a transferência de dados entre memória e dispositivos de entrada e saída, a decodificação de instruções, a execução de operações lógicas e aritméticas sinalizadas pelas instruções, resposta aos sinais de controle externos como, por exemplo, o RESET, entre outras funções.

Todas essas funções são executadas por circuitos lógicos, que controlam o que ocorrem na CPU baseados nas instruções que são colocadas na memória. Isso torna a CPU versátil e flexível, pois quando quisermos alterar a operação, o que só é preciso fazer é trocar o programa na memória ao invés de mudar todos os circuitos eletrônicos (hardware) da CPU (TOCCI, 2003).

Na linguagem da CPU as características que descrevem como ela é formada são chamadas de arquitetura do processador e os componentes interligados que o constitui recebem o nome de organização do processador. A figura 2.2 ilustra numa visão mais abstrata a composição básica de um processador apartir dessas duas denominações (ZEFERINO,).

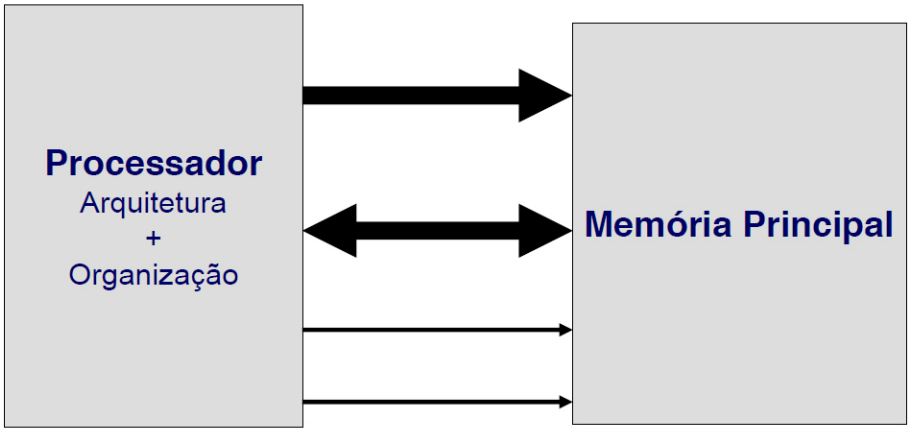

Figura 2.2: Visão abstrata da CPU apartir da arquitetura e organização (ZEFERINO,).

2.2 Arquitetura do processador

A arquitetura do processador descreve apartir de uma visão mais abstrata às características de como ele funciona, isto é, os atributos que são visíveis ao programador do processador (programação em linguagem de montagem). Dessa forma os atributos arquiteturais de um processador são:

- Tamanho da palavra de instrução
- Tamanho da palavra de dados
- Tipos de dados
- Formato das instruções
- Modos de endereçamento
- Registradores
- Conjunto de instruções

2.2.1 Tamanho da palavra de instrução

O tamanho da palavra de instrução refere-se ao número de bits usados para representar uma instrução de programa (ZEFERINO,).

2.2.2 Tamanho da palavra de dados

O tamanho da palavra de dados se refere ao número de bits do dado manipulado pelo processador (ZEFERINO,).

2.2.3 Tipos de dados

Quais os tipos de dados manipulados pelo processador: inteiro (com ou sem sinal), real, etc. (ZEFERINO,).

2.2.4 Formato das instruções

O formato das instruções mostra como é a estrutura utilizada para organização das instruções, a largura (em bits) do campo do código da operação (OpCode), o número de operandos e a largura dos operandos (em bits) (ZEFERINO,).

2.2.5 Modos de endereçamento

Os modos de endereçamento referem-se aos métodos de acesso aos dados processados pelas instruções (ZEFERINO,).

2.2.6 Registradores

São os registradores de uso geral e específico que tem influência nas instruções. Assim apenas alguns registradores são apresentados na arquitetura (ZEFERINO,).

2.2.7 Conjunto de instruções

O conjunto de instruções mostra quais são às instruções suportadas pelo processador, quais seus códigos de operação (OpCode) e como são representadas através dos mnemônico (apelido) (ZEFERINO,).

2.3 Organização do processador

A organização do processador mostra através de uma visão mais detalhada quem são os componentes que o constitui e como são interligados. Nesses componentes é implementado a arquitetura do processador além de outros atributos que não são visíveis ao programador, como por exemplo, o ciclo de busca e a execução das instruções apartir dos componentes que compõe o mesmo, entre outros atributos. Assim essa parte possui o foco de atenção do engenheiro de computação (projetista de hardware). A figura 2.3 mostra a diferença entre arquitetura/organização do processador (ZEFERINO,).

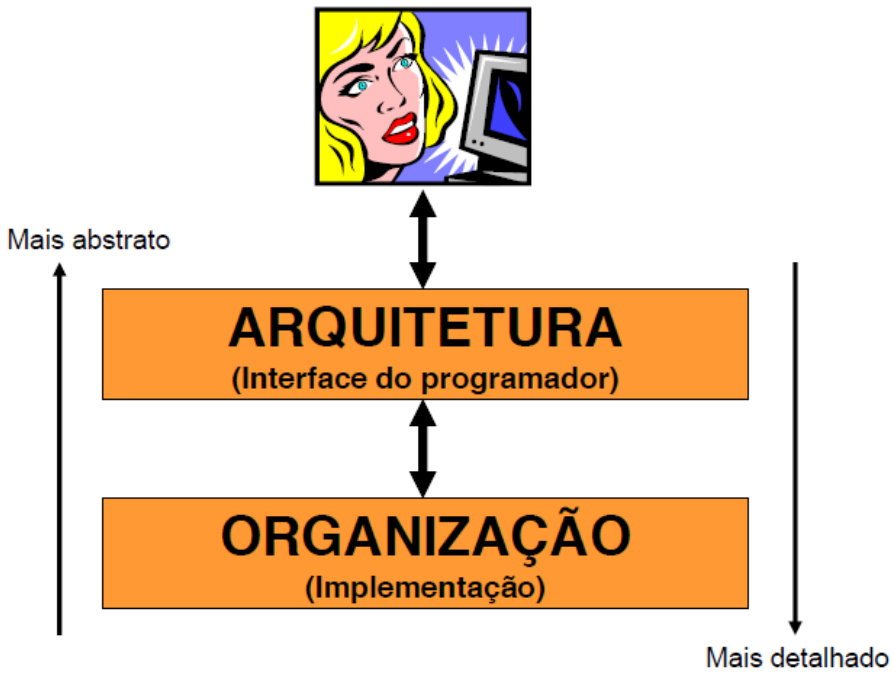

Figura 2.3: Diferença entre arquitetura/organização do processador (ZEFERINO,).

2.4 Circuito Combinacional VS Circuito Sequencial

Um circuito combinacional é um circuito composto por portas lógicas onde o valor da saída depende somente do valor atual da entrada. A figura 2.4 mostra o modelo de um circuito combinacional (PEDRONI, 2010).

Figura 2.4: Circuito Combinacional.

Já um circuito sequencial é formado por um circuito combinacional, mas que possui um elemento de armazenamento de informação (memória) e um laço de realimentação. Dessa forma o valor da saída não depende só do valor atual da entrada. A figura 2.5 exibe o modelo de um circuito sequencial (MORENO, 2003)MUX..

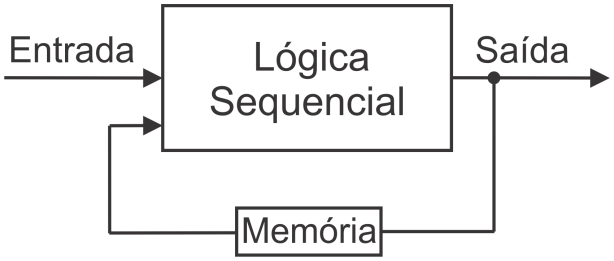

Figura 2.5: Circuito Sequencial.

2.5 Instrução

Instrução é uma operação a ser executada por algo ou alguém. No nosso caso será executada pelo processador. A instrução do processador pode ser formada apenas pela operação ou formada pela operação seguida de um operando, onde a operação indica a função a ser desempenhada e o operando é o valor (dado) a ser usado pela operação (WEBER, 2008).

2.6 Programa

Programa é uma sequência de instruções pré-definidas por alguém, por exemplo, o programador, com o objetivo de descrever uma ou várias tarefas que serão executadas pelo processador (WEBER, 2008).

2.7 Sinais de Controle

Sinais de controle ou entrada de controle são sinais responsáveis por ativar ou selecionar uma determinada operação nos elementos digitais (WEBER, 2008).

2.8 Memória

Memória é um circuito combinacional que tem a função de armazenar instruções e dados. Organizada em posições onde cada posição é identificada por um endereço. Podemos pensar nessas posições como elementos de uma matriz no qual cada elemento possui um endereço (TOCCI, 2003).

2.9 Barramento

Barramento é o caminho físico que interliga e permite o transporte dos dados entre os elementos digitais. Um barramento só pode receber dados de um elemento de cada vez. Assim seu compartilhamento deve ser feito através de tempos de clock. Caracteriza-se pela sua largura em bits. Dessa forma a largura em bits do barramento deve suportar a taxa de transmissão também em bits dos elementos por ele transportados, como por exemplo, as instruções, os operandos, os sinais de controle, entre outros (WEBER, 2008).

2.10 Multiplexador

O multiplexador (Mux) é um circuito combinacional com a finalidade de selecionar, através de um sinal de controle, uma de suas entradas, conectando-a a uma única saída. A figura 2.6 exibe o modelo de um Mux 2x1 (PEDRONI, 2010).

A tabela 2.1 mostra a tabela verdade genérica de um Mux 2X1.

s	y
0	a
1	b

Tabela 2.1: Tabela Verdade genérica do Mux.

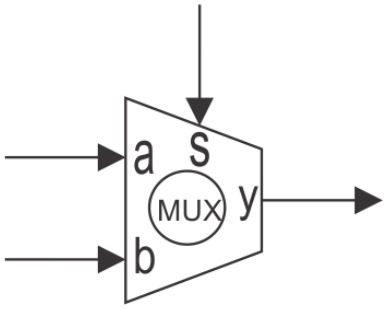

Figura 2.6: Mux.

2.11 Incrementador

O incrementador é um circuito combinacional que executa a operação aritmética de soma apartir do valor da sua entrada e um valor pré-configurado. A figura 2.7 exibe o modelo de um incrementador que soma a sua entrada com mais um (PEDRONI, 2010).

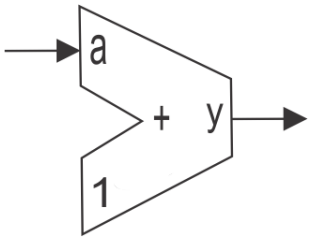

Figura 2.7: Incrementador.

A tabela 2.2 mostra a tabela verdade genérica de um incrementador.

y
a + 1

Tabela 2.2: Tabela verdade genérica do Incrementador.

2.12 Busca-Decodificação-Execução de Instruções

São tarefas básicas de um processador a busca, decodificação e a execução de instruções. Essa sequência de tarefas é feita repetidamente, até que seja decodificada uma instrução que finalize e execução, como por exemplo, a instrução HLT. Na fase de busca é feita a leitura de uma instrução da memória. A fase de decodificação é caracterizada pela definição dos sinais de controle internos da UC que serão ativados baseados na instrução que será executada. Na fase de execução a ação indicada pela instrução é executada através dos componentes do processador, escolhidos a partir dos sinais de controle da UC que foram ativados. A figura 2.8 exibe esse processo da Busca-Decodificação-Execução (WEBER, 2008).

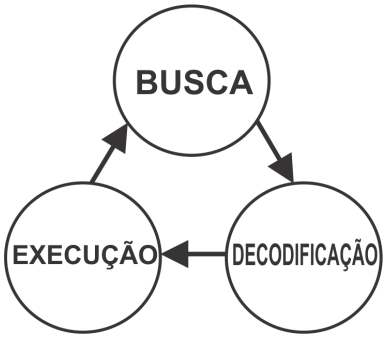

Figura 2.8: Busca-Decodificação-Execução.

2.13 Relógio

O relógio ou clock (*clk*) em circuitos digitais é um sinal de controle usado para coordenar as ações de dois ou mais circuitos eletrônicos. Um sinal de *clk* oscila entre os estados alto (1) e baixo (0) e é fornecido por uma onda quadrada que é criada a partir de um cristal e de um CI gerador de frequência. Circuitos digitais que usam o sinal de *clk* para sincronização podem se tornar ativos na subida, na descida ou em ambos os momentos do sinal de *clk*. A figura 2.9 mostra como é representado o sinal do *clk* nos circuitos digitais (WEBER, 2008).

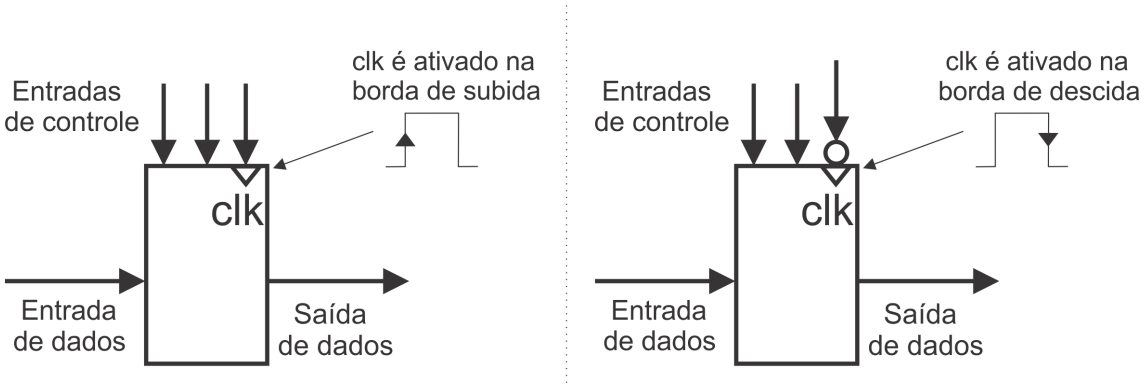

Figura 2.9: Representação do sinal de *clk* nos circuitos digitais.

2.14 Registrador

O registrador é um conjunto de elementos de memória (flip-flops ou latches) que tem a capacidade de guardar n bits de informação. Possui n entradas e saídas de dados, além de também possuir um sinal de controle de *clk* que determina quando novos conteúdos são armazenados. A pulsação ou mudança no sinal do *clk* faz com que o registrador mude ou retenha o valor de saída baseado no valor de entrada. O registrador é tipicamente usado como dispositivo de armazenamento temporário.

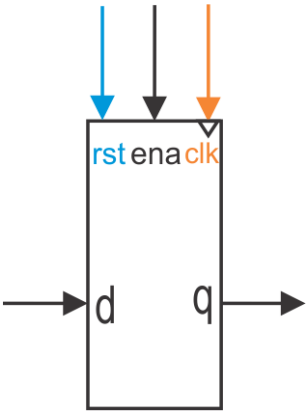

Figura 2.10: Registrador genérico.

A figura 2.10 mostra como é composto um registrador genérico. Ele é composto por mais dois sinais de controle, o reset (*rst*) e o enable (*ena*) além de informar que o *clk* é ativado na borda de subida. O *rst* quando esta ativo, reseta a(s) saída(s), isto é, zerar o valor da(s) saída(s) do registrador. O *ena* quando ativado tem a função de transferir o valor da entrada para a saída. Mas essa ação só é executada quando no mesmo tempo de *clk* o mesmo estiver na borda de subida. Se essa ação for válida, no próximo período do *clk* o valor de entrada é transferido para a saída. O valor só é transferido no próximo ciclo de *clk* porque como já mencionado o *clk* desse registrador se inicia na sua subida (MORENO, 2003). Para ativação dos sinais de controle nesse trabalho, foi definido que o *rst* é ativado no estado baixo (0) e o *ena* é ativado no estado alto (1). A tabela 2.3 mostra a tabela verdade genérica do registrador genérico.

clk	rst	ena	q
X	0	X	0
X	1	0	q0
↑	1	1	d
↓	1	1	q0

Tabela 2.3: Tabela verdade genérica do Registrador genérico.

Legenda:

q0 = Mantém valor do estado anterior

2.15 Conjunto de registradores

Os principias registradores que compõem um processador são apresentados a seguir. Relembrando que todos os registradores usados nesse trabalho são do tipo registrador genérico.

2.15.1 Registrador de Endereços de Memória

O Registrador de Endereços de Memória (REM) tem a função de armazenar um valor que indica qual endereço de memória vai ser selecionado para ser lido ou escrito pela memória (WEBER, 2008).

2.15.2 Registrador de Dados de Memória

O Registrador de Dados de Memória (RDM) após a leitura de um endereço de memória possui a função de armazenar o conteúdo (instrução ou operando) desse endereço. Também armazena um operando que posteriormente vai ser escrito no conteúdo de um endereço de memória selecionado (WEBER, 2008).

2.15.3 Acumulador

O registrador Acumulador (AC) ou registrador A, tem a função de armazenar um operando ou um resultado fornecido pela Unidade Lógica e Aritmética (ULA) que compõe a unidade de execução (WEBER, 2008).

2.15.4 Registrador de Estado

O Registrador de Estado (RST) possui a função de armazenar os códigos de condição gerados pela ULA (WEBER, 2008).

2.15.5 Registrador de Instrução

O Registrador de Instrução (RI) tem a função de armazenar a instrução que vai ser decodificada pela UC e que posteriormente irá ser executada pelo processador (WEBER, 2008).

2.15.6 Contador de Programa

O registrador Contador de Programa (PC) que também é chamado de apontador de instruções possui a função de armazenar um valor que indica qual é o endereço de memória a ser lido pela memória. A cada instrução executada, o PC é incrementado para que o programa avance para a instrução seguinte, ou seja, aponte para a próxima instrução a ser executada (WEBER, 2008).

2.16 Unidade de Execução

A unidade de execução é como é chamado o componente que executa as operações de lógica e aritmética do processador. Os processadores possuem dois tipos de unidade de execução, a Unidade Lógica e Aritmética (ULA) que trabalha com números inteiros e a Unidade de Ponto Flutuante (UPF) que trabalha com números reais. Em processadores simples a única unidade que é utilizada é a ULA (TORRES, 2005).

2.16.1 ULA

As principais operações realizadas pela ULA são:

- Operações aritméticas (adição, subtração, multiplicação, divisão) com números inteiros;
- Operações lógicas (AND, NOT, OR, Negação, Complemento de Dois) bit a bit;
- Operações de deslocamento de bits, para a esquerda ou para direita;
- Operações de rotação de bits, para a esquerda ou para direita;

A escolha da operação a ser usada pela ULA é determinada pela UC. A ULA também fornece códigos de condição que são indicações sobre a operação que foi realizada. A seguir alguns exemplos de códigos de condição normalmente utilizados:

- Zero (Z): Indica se o resultado da operação é zero.
- Negativo (N): Informa se o resultado da operação é negativo.
- Carry (C): Para soma ou subtração pode representar o bit vai-um (carry-out) ou vem-um (borrow-out) respectivamente. Em operações de deslocamento serve para guardar ou fornecer o bit deslocado.
- Overflow (Estouro de Campo) (V): Indica que o resultado de uma operação aritmética não pode ser representado no tamanho de bits disponível.

A figura 2.11 exibe como é formada uma ULA (WEBER, 2008).

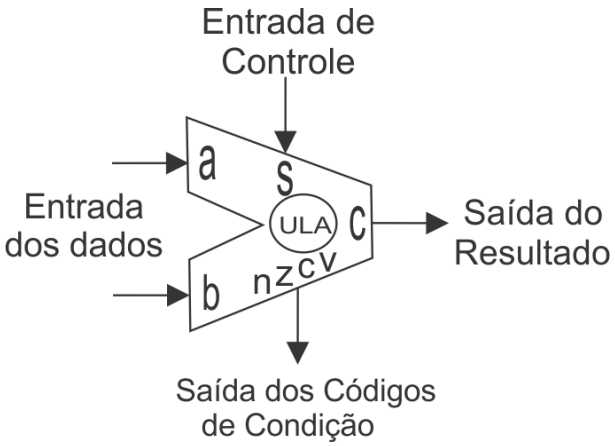

Figura 2.11: ULA.

2.17 Unidade Controle

A Unidade de Controle (UC) é um componente formado por lógica sequencial que tem seu comportamento representado por máquina de estados finitos ou FMS (Finite State Machine). É responsável por controlar todos os elementos do processador fornecendo sinais de controle e temporização. Possui sinais de entradas de controle vindos de dispositivos externos para a inicialização, reset, etc. Recebe sinais de entrada de dados que podem ser usados para determinação do próximo estado. Também fornece sinais de saída de controle usados para comunicação com os componentes do processador. Cada sinal fornecido executa uma determinada tarefa nos componentes do processador, no qual pode ser a ativação de um registrador, a seleção de entrada de um multiplexador, a leitura ou escrita da memória, a seleção da operação da ULA entre outras tarefas de controle (WEBER, 2008). A figura 2.12 exibe a simbologia de uma UC genérica.

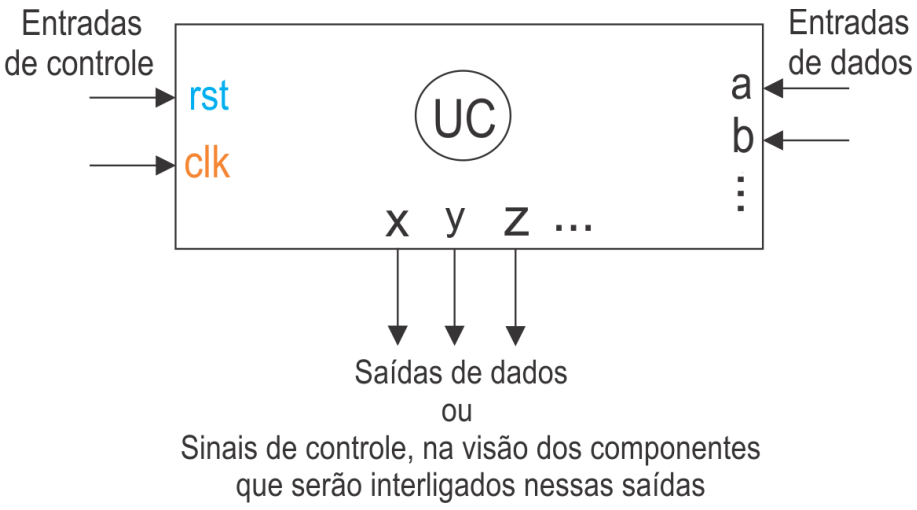

Figura 2.12: UC genérica.

2.17.1 Máquina de Estados Finitos

A máquina de estados finitos ou FMS é um modelo de comportamento usado para representar programas de computadores ou circuitos digitais (HARRIS, 2013). Nesse trabalho ela será usada para representar um circuito digital, no caso, a UC. Ela é composta por:

- Estados
- Transições
- Ações

Os estados armazenam informações sobre o passado refletindo as modificações das entradas do início até o presente momento. As transições indicam uma troca de estado e podem ou não precisar de uma condição que habilita a modificação de estado. A ação descreve a atividade que deve ser executada em um determinado tempo de clock (MATOS, 2013). A figura 2.13 exibe a representação de uma FSM genérica.

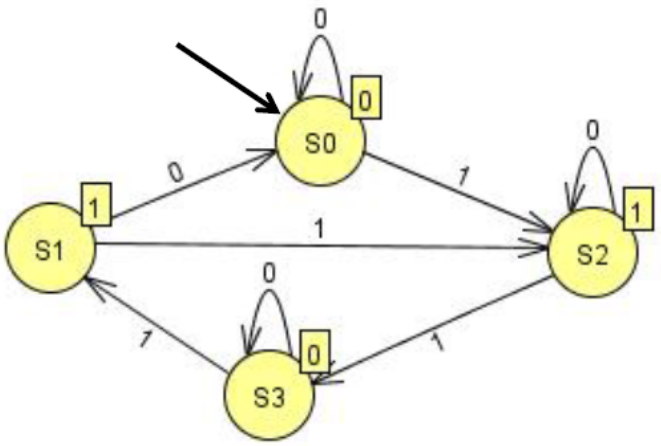

Figura 2.13: FSM genérica (MATOS, 2013).

2.18 Processadores Didáticos

Processadores didáticos ou hipotéticos são processadores simplificados criados para fins de ensino com o objetivo de facilitar a explicação de como funciona um processador. São criados através da arquitetura e organização básica dos processadores, ou seja, a partir dos conceitos, elementos e instruções básicas que qualquer processador independente da aplicação

possui tornando o aprendizado menos complicado. Também permitir o aprimoramento desse conjunto básico de informações possibilitando o aumento gradativo do nível de complexidade, isto é, conceito a conceito.

Seguindo essa filosofia, professores da UFRGS, que trabalham com as disciplinas de Arquitetura de Computadores e Microprocessadores, criaram um conjunto de processadores didáticos para melhorar o ensinamento do assunto aos alunos dessas disciplinas. Os processadores foram nomeados de Neander, Ahmes, Ramses e Cesar. Cada um possui um nível de complexidade diferente e a sequência dos nomes informa o respectivo nível dessa complexidade. Eles possuem uma arquitetura e organização aberta, ferramentas para criar programas através de suas respectivas instruções conhecidos como montadores assembler e simuladores de suas arquiteturas que interpretam esses programas criados apartir dos Montadores. Nesse trabalho os processadores implementados seriam o Neander e o Ahmes, mas devido à escassez do tempo o único processador implementado foi o Neander. Assim a figura 2.14 exibe o montador assembler chamado Daedalus utilizado no estudo do Neander e para a criação de programas para o Simulador da arquitetura do mesmo. Na imagem percebe-se que o montador não tem a opção da arquitetura do Neander para a criação de programas. Mesmo assim é possível criar programas para o Neander através da arquitetura do Ahmes porque ele foi criado apartir da arquitetura do Neander.

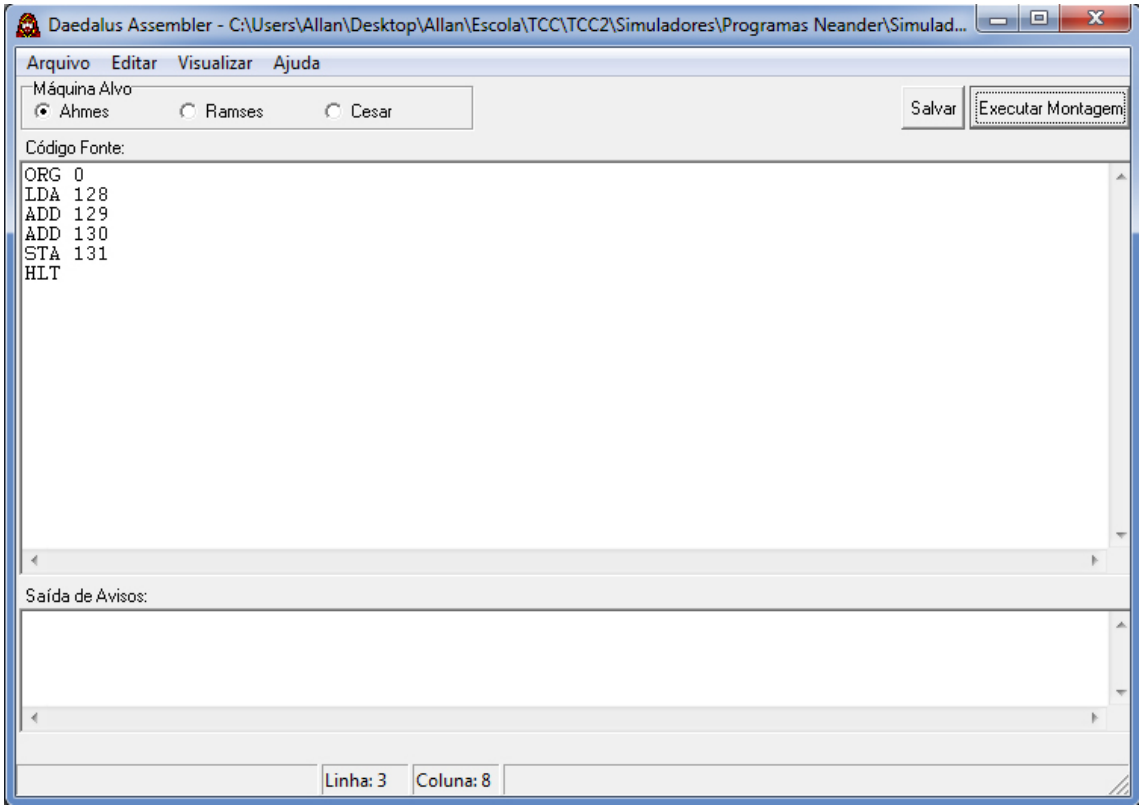

Figura 2.14: Montador Daedalus.

Depois de criado o programa é usado o simulador para executar o programa. A figura 2.15 mostra o simulador da arquitetura do Neander utilizado para o estudo desse processador e para a interpretação dos programas criados com o Daedalus.

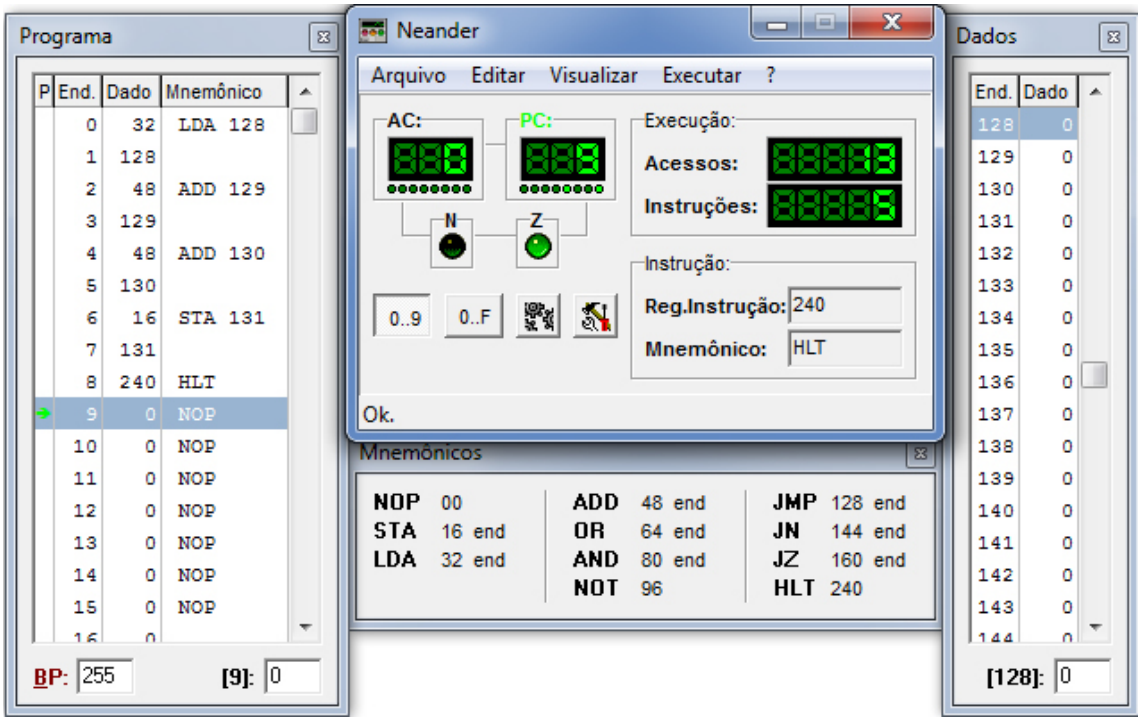

Figura 2.15: Simulador da arquitetura do Neander.

Não entrando em muitos detalhes, o simulador é composto por quatro janelas, no qual duas delas representam a memória RAM e estão divididas em Programa e Dados. Outra janela informa quais são as instruções suportadas pelo simulador, e na ultima janela, de todas as informações apresentadas, o que interessa saber no momento é que nela são apresentados apenas alguns dos componentes que formam o Neander, que são os registradores AC e PC, além da sinalização dos códigos de condição N e Z da ULA. Isso acontece porque como já foi mencionado na motivação do trabalho, essas ferramentas apenas focam nas arquiteturas dos processadores e esses são os componentes necessários para entender a arquitetura do Neander. Assim não é mostrado como a interpretação dos programas ocorre através de todos os componentes que formam o processador, ou seja, através da sua organização.

2.19 Linguagem de descrição de hardware

A linguagem de descrição de hardware ou HDL (Hardware Description Language) é uma classe de linguagens utilizadas para projetar hardware ou uma parte dele. São padrões de expressões baseados em texto da estrutura espacial, temporal e comportamental dos sistemas

eletrônicos. Descrevem o funcionamento do circuito, a sua concepção e organização, e ainda é possível testá-lo para verificar seu funcionamento por meio de simulação. Dessa forma essa linguagem tem a capacidade de modelar um hardware ou parte dele antes de ser criado fisicamente, reduzindo o tempo e os custos. São usadas extensivamente na indústria para projetos de sistemas digitais variando de microprocessadores, há componentes dentro de aparelhos de consumo.

As HDLs têm uma grande semelhança com as linguagens de programação, pois também possuem sintaxe e semântica. No entanto, são especificamente orientadas à descrição das estruturas e do comportamento do hardware. Outra diferença em relação às linguagens de programação é que essa linguagem tem a capacidade de modelar vários processos paralelos, ou seja, cada processo é executado independentemente um do outro. Os exemplos mais comuns de HDL são o VHDL, VERILOG, AHDL, MyHDL e Handel-C (MORENO, 2003).

Nesse trabalho como o próprio titulo já menciona, é utilizado o VHDL para projetar (implementar) o processador didático Neander. Ele foi escolhido porque foi o único HDL que o autor do trabalho aprendeu durante o curso de Tecnólogo em Sistemas de Telecomunicações.

2.19.1 VHDL

VHDL significa [VHSIC (Very High Speed Integrated Circuits) Hardware Description Language] é uma linguagem padronizada para descrever componentes digitais, permitindo a transferência de componentes ou projetos para qualquer tecnologia em construção de hardware existente ou que ainda será desenvolvida. Ela fornece uma rica variedade de construções que permitem modelar o hardware em um alto nível de abstração.

VHDL foi resultado de uma iniciativa financiada pelo Departamento de Defesa dos Estados Unidos na década de 1980. Foi à primeira linguagem de descrição de hardware padronizado pelo IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos). Apesar de amplamente utilizada na descrição e síntese de sistemas digitais, a linguagem VHDL foi primeiramente definida com objetivos de simulação.

VHDL proporciona algumas vantagens que facilitam a vida dos desenvolvedores de hardware. As principais vantagens são a redução de tempo e dos custos do desenvolvimento dos projetos, o aumento do nível de abstração na implementação do hardware, a não dependência de uma tecnologia específica e a grande facilidade nas atualizações dos projetos por permitir a simulação do hardware. Por fim, através dessas vantagens, VHDL já dispõe de um grande número de usuários que interagem com essa linguagem.

Para trabalhar com VHDL empresas do ramo eletrônico fornecem ferramentas de síntese e simulação. A tabela 2.4 exibe as principais empresas e suas ferramentas para síntese e simulação em VHDL (PEDRONI, 2010).

Empresa	Ferramenta de Síntese	Ferramenta se Simulação
Altera	Quartus II	QSim
Xilinx	ISE	ISE
Mentor Graphics	Precision RTL	ModelSim

Tabela 2.4: Empresas e suas ferramentas de síntese e simulação de VHDL.

Nesse trabalho a ferramenta usada na síntese foi o Quartus II 12.1 sp1 - Web Edition da Altera e a ferramenta de simulação foi o ModelSim Starter Edition 10.1b da Mentor Graphics.

Na sintetização de um código em VHDL, a estrutura básica do mesmo consiste em três partes como mostra a tabela 2.5 no qual é apresentado o código da porta lógica AND (figura 2.16) em VHDL.

```
LIBRARY ieee;                                          –Primeira parte
USE ieee.std_logic_1164.all;
ENTITY porta_and IS                                    –Segunda parte
   PORT ( a: IN STD_LOGIC;
          b: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END porta_and;
ARCHITECTURE porta_and_codigo OF porta_and IS          –Terceira parte
BEGIN
   y <= a AND b;
END porta_and_codigo;
```

Tabela 2.5: Porta AND em VHDL.

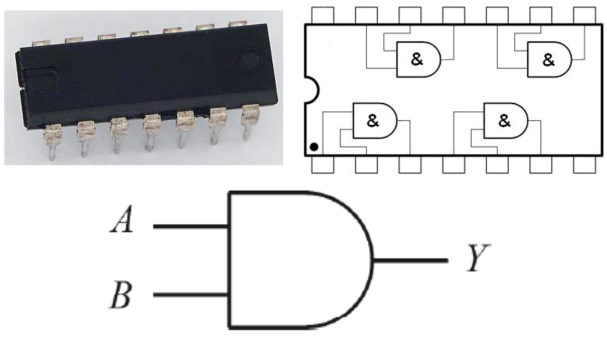

Figura 2.16: Porta lógica AND que compõe os circuitos digitais.

A primeira parte do código é definida pelas Declarações de bibliotecas e pacotes. Ela contém a lista das bibliotecas e pacotes (LIBRARY e USE respectivamente) que o compilador vai usar para processar o projeto.

A segunda parte é composta pela Entity (Entidade). Ela possui duas seções de código denominadas ”PORT”e ”GENERIC”. Podem ser usadas sozinhas ou ao mesmo tempo. Nessas seções são feitas as declarações das entradas e saídas que compõe o circuito digital implementado, além de informar quantos bits cada entrada e saída irá suportar.

A terceira parte do código é definida pela Architecture (Arquitetura). É formada por declarações de sinais, constantes, componentes, subprogramas, etc. Também contém comandos como blocos, atribuições a sinais, chamadas a subprogramas, instanciação de componentes, processos, entre outros. Resumindo ela armazena a lógica de funcionamento do circuito digital implementado (PEDRONI, 2010).

3 *Arquitetura do Neander*

O Neander foi criado com uma arquitetura de 8 bits. Dessa maneira a memória do mesmo possui 256 endereços para armazenar as instruções e os operandos para execução dos programas, como mostra a figuras 3.1.

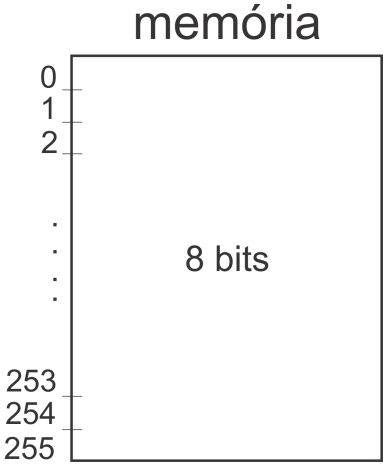

Figura 3.1: Memória 8 bits.

3.1 Tamanho da palavra de instrução

O Neander possui 8 bits no tamanho da palavra de instrução para representar uma instrução de programa (WEBER, 2008).

3.2 Tamanho da palavra de dados

O Neander possui 8 bits no tamanho da palavra de dados para ser manipulado pelo processador (WEBER, 2008).

3.3 Tipos de dados

Os tipos de dados manipulados pelo Neander são do tipo inteiro com sinal. Dessa forma os números negativos são representados em complemento de dois (WEBER, 2008). O complemento de dois é um padrão utilizado nos números binários para representar os números negativos. Foi convencionado que o bit mais significativo ou MSB (Most Significant Bit) indica o sinal do valor armazenado, independente do tamanho de bits do dado. O numero é considerado positivo se o valor do MSB for zero e negativo se for um.

Assim tendo 8 bits no tamanho da palavra de dados e o uso do complemento de dois nos números negativos, o Neander apenas irá trabalhar com números positivos de 0 à 127 e de -1 à -128 com os números negativos. A tabela 3.1 mostra exemplos do uso do complemento de dois em valores de 8 bits.

Binário	Interpretação de complemento de dois	Interpretação sem sinal
00000000	0	0
00000001	1	1
...	...	...
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
...	...	...
11111101	-3	253
11111101	-2	254
11111101	-1	255

Tabela 3.1: Complemento de dois em valores de 8 bits.

3.4 Formato das instruções

As instruções do Neander são formadas por uma ou duas palavras (bytes) como mostram as figuras 3.2 e 3.3 respectivamente.

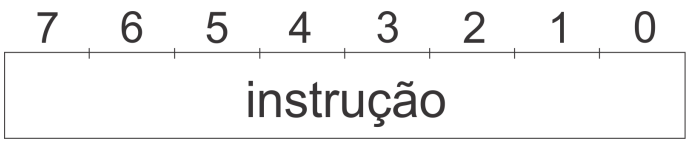

Figura 3.2: Instrução de uma palavra.

A instrução de uma palavra é dividida em dois campos, o do OpCode e o dos bits restantes. O OpCode possui 4 bits e esta localizado nos 4 MSB da mesma. É nele que se encontra a

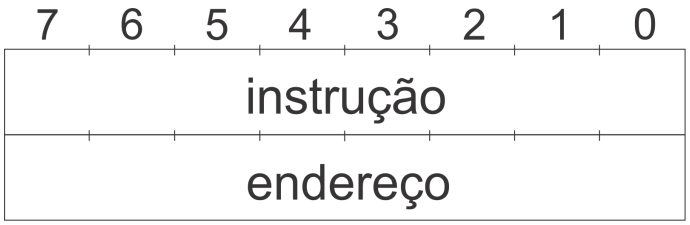

Figura 3.3: Instrução de duas palavras.

instrução. Os bits restantes não são importantes e não interferem na identificação da instrução como ilustra a figura 3.4.

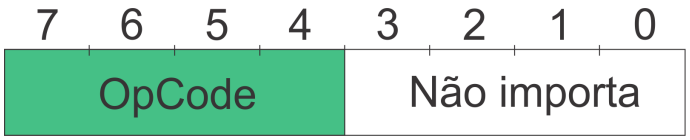

Figura 3.4: Mais detalhes sobre a instrução de uma palavra.

Na instrução de duas palavras, a primeira palavra tem a mesma estrutura da instrução de uma palavra e assim ela também contém só a instrução. Já a segunda palavra possui só um campo, o do endereço direto que é composto por todos os bits da palavra, isto é, os 8 bits. Esse endereço informa onde se encontra o operando que vai ser usado junto à instrução (WEBER, 2008). A figura 3.5. mostra esses detalhes sobre a instrução de duas palavras.

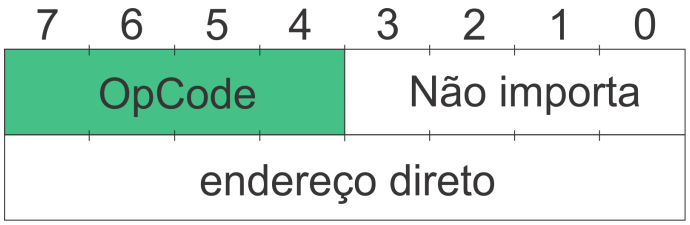

Figura 3.5: Mais detalhes sobre a instrução de duas palavras.

3.5 Modos de endereçamento

O Neander possui só um modo de endereçamento, o endereçamento direto. Endereçamento direto é como é chamado o endereço de memória que acompanha a instrução de duas palavras. Como já foi mencionado esse endereço informa onde se encontra o operando que vai ser usado junto à instrução (WEBER, 2008). A figura 3.6 mostra como funciona o endereçamento direto.

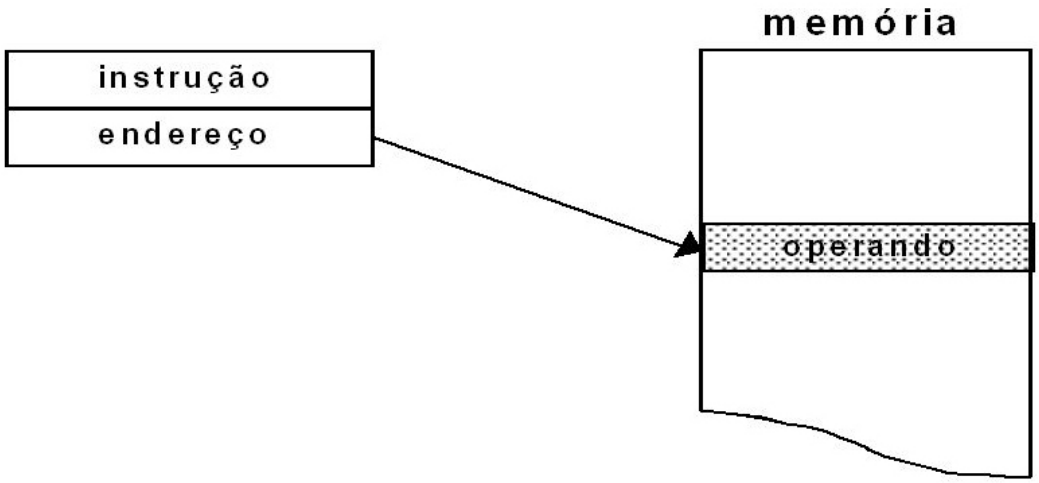

Figura 3.6: Endereçamento direto (WEBER, 2008).

3.6 Registradores

Os registradores uso geral e específico que tem influência nas instruções do Neander e são conhecidos na sua arquitetura são o AC e o PC respectivamente.

3.7 Conjunto de instruções

O Neander compreende um conjunto de 11 instruções que podem ser formadas por uma ou duas palavras. Elas são classificadas em:

- Formada por uma palavra
  - Instrução de controle.
- Formada por duas palavras.
  - Instrução de transferência.
  - Instrução de aritmética e lógica.
  - Instrução de desvio.

A tabela 3.2 mostra as informações sobre essas instruções.

Legenda:

← = *Atribuição*

Mem(end) = Conteúdo do endereço

Tipo	OpCode	Mnemônico	Ação	Comentário
Controle	0000	NOP	Não executa nada	Não executa nada
Acesso	0001	STA end	MEM(end) ← AC	Copia o operando do AC para um endereço
Acesso	0010	LDA end	AC ← MEM(end)	Copia o operando de um endereço para o AC
Aritmética.	0011	ADD end	AC ← MEM(end) + AC	Soma o operando de um endereço com o operando do AC
Lógica	0100	OR end	AC ← MEM(end) OR AC	Faz operação OU do operando de um endereço com o operando do AC
Lógica	0101	AND end	AC ← MEM(end) AND AC	Faz operação E do operando de um endereço com o operando do AC
Lógica	0110	NOT	AC ← NOT AC	Inverte todos os bits do operando do AC
Desvio	1000	JMP end	PC ← end	Desvio incondicional (Jump) Faz o desvio para um endereço
Desvio	1001	JN end	If N=1 then PC ← end	Desvio condicional (Jump on Negative) Faz o desvio para um endereço se o código de condição Negativo for igual a um.
Desvio	1010	JZ end	If Z=1 then PC ← end	Desvio condicional (Jump on Zero) Faz o desvio para um endereço se o código de condição Zero for igual a um.
Controle	1111	HLT	Termina execução	Termina execução

Tabela 3.2: Instruções do Neander.

Através das instruções percebe-se que além dos registradores AC e PC, outro componente também tem influência nas instruções do Neander e aparece na sua arquitetura. Esse componente é a ULA que além de participar da execução das instruções de lógicas, aritmética e de transferência (AND, OR, NOT, ADD, LDA), também fornece os seguintes valores de códigos de condição que são usados pelas instruções JN e JZ:

- Negativo (N): indica o sinal do resultado. A ULA verifica o MSB do resultado. Se o valor for um (1) o resultado é negativo, se for zero (0) o resultado é positivo.

- Zero (Z): indica se o resultado é igual à zero. A ULA faz a operação lógica NOR com todos os bits do resultado, se o valor final for um (1) o resultado é zero, se for zero (0) o resultado é diferente de zero.

As demais instruções STA, JMP, JN, JZ, NOP e HLT não fornecem nenhum valor aos códigos de condição da ULA (WEBER, 2008).

Assim apartir de todas as características apresentadas à visão em blocos do Neander através da sua arquitetura é ilustrada na figura 3.7.

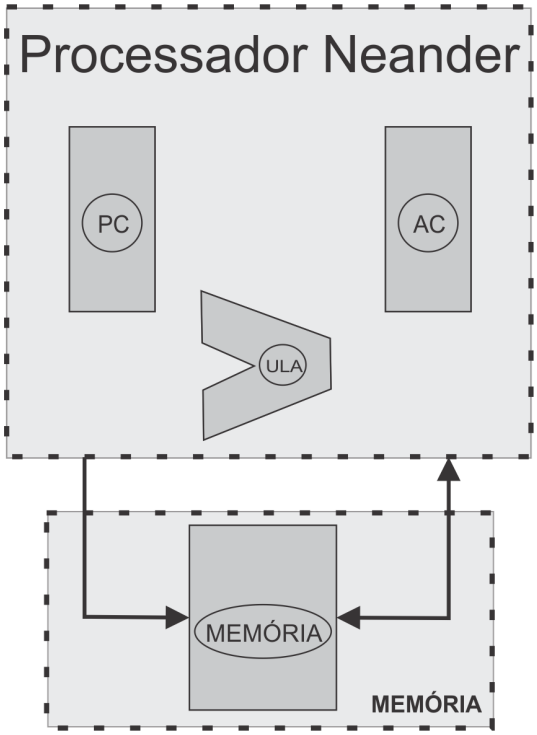

Figura 3.7: Visão em blocos do Neander através da sua arquitetura.

3.8 Exemplo de programa

A tabela 3.3 mostra um programa representado por mnemônicos que realiza a soma de três posições consecutivas da memória e armazena o resultado numa quarta posição.

Porém quando o programa é executado pelo Neander, os mnemônicos são convertidos em linguagem de máquina por um montador, ou seja, todas as informações do programa serão representadas por números binários, 0 e 1 como é apresentado na tabela 3.4.

Programa		
Mnemônico	Ação	Comentários
LDA 128	AC ← MEM(128)	Copia o operando do end.128 para o AC
ADD 129	AC ← MEM(129) + AC	Soma o operando do end.129 com o operando do AC
ADD 130	AC ← MEM(130) + AC	Soma o operando do end.130 com o operando do AC
STA 131	MEM(131) ← AC	Copia o operando do AC para o end.131
HLT	Termina execução	Termina execução

Tabela 3.3: Programa que soma três posições consecutivas.

Mnemônico	Decimal	Hexadecimal	Binário
LDA 128	32 128	20 80	00100000 10000000
ADD 129	48 129	30 81	00110000 10000001
ADD 130	48 130	30 82	00110000 10000010
STA 131	16 131	10 83	00010000 10000011
HLT	240	F0	11110000

Tabela 3.4: Linguagem mnemônica e de máquina.

Depois de montado, o programa é colocado na memória. A tabela 3.5 demonstra como o programa é carregado na memória através da linguagem de mnemônicos. Para critérios de organização o Neander separa metade da memória para as instruções e a outra metade para os operandos. Assim tendo 256 endereços de memória, os primeiros 128 endereços vão armazenar as instruções e os outros 128 vão armazenar os operandos. No qual como pode ser visto os operandos foram representados com variáveis.

MEMÓRIA - 8 bits			
Área das Instruções		Área dos Operandos	
end	**Conteúdo**	**end**	**Conteúdo**
0	LDA	128	w
1	128	129	x
2	ADD	130	y
3	129	131	z
4	ADD	132	
5	130	133	
6	STA	134	
7	131	135	
8	HLT	...	...
...	..	255	

Tabela 3.5: Memória carregada com o programa que soma três números (mnemônicos).

A tabela 3.6 ilustra como o programa é carregado na memória através da linguagem de máquina. Para simplificar a linguagem de máquina, os endereços e conteúdos da memória são representados em hexadecimal.

MEMÓRIA - 8 bits			
Área das Instruções		Área dos Operandos	
end	**Conteúdo**	**end**	**Conteúdo**
00H	20H	80H	w
01H	80H	81H	x
02H	30H	82H	y
03H	81H	83H	z
04H	30H	84H	
05H	82H	85H	
06H	10H	86H	
07H	83H	87H	
08H	F0H	...	...
...	..	FFH	

Tabela 3.6: Memória carregada com o programa que soma três números (hexadecimal).

Para entender a lógica do programa a figura 3.8 exibe o seu fluxograma. O programa inicia com a soma dos dois primeiros operandos. O resultado gerado é somado com o terceiro operando. Para finalizar o resultado da ultima soma é armazenado em um quarto endereço de memória da área dos operandos.

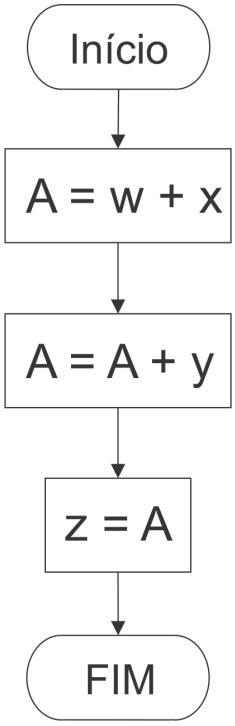

Figura 3.8: Fluxograma do programa que soma três números.

Assim seguindo o exemplo desse programa, é possível criar outros programas com as instruções do Neander, pois a lógica vai ser sempre a mesma.

4 Organização do Neander

4.1 Componentes

O Neander é composto pelos seguintes componentes:

- Unidade de Controle (UC);
- ULA de 8 bits com 2 estados: Negativo (N) e Zero (Z);
- Incrementador de 8 bits;
- AC de 8 bits;
- PC de 8 bits;
- RST (NZ) de 2 bits para códigos de condição: Negativo (N) e Zero (Z) .
- RI de 4 bits;
- REM de 8 bits;
- RDM de 8 bits.;

4.2 ULA do Neander

Na organização é possível ver com mais detalhes como é a composição da ULA do Neander, no qual é mostrado na figura 4.1. Também é visto na tabela 4.1 sua lógica de funcionamento.

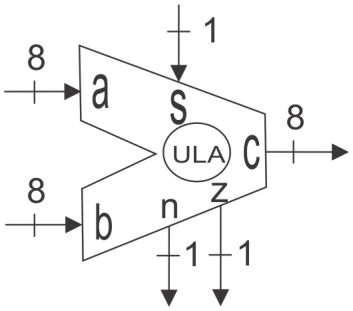

Figura 4.1: Composição da ULA do Neander

s	c	n	z
000	a + b	MSB(c)	NOR(c)
001	a AND b	MSB(c)	NOR(c)
010	a OR b	MSB(c)	NOR(c)
011	NOT a	MSB(c)	NOR(c)
100	a	MSB(c)	NOR(c)

Tabela 4.1: Lógica de funcionamento da ULA do Neander

4.3 Interligação dos componentes

A figura 4.2 ilustra o modo de interligação escolhida dos componentes que compõe o Neander.

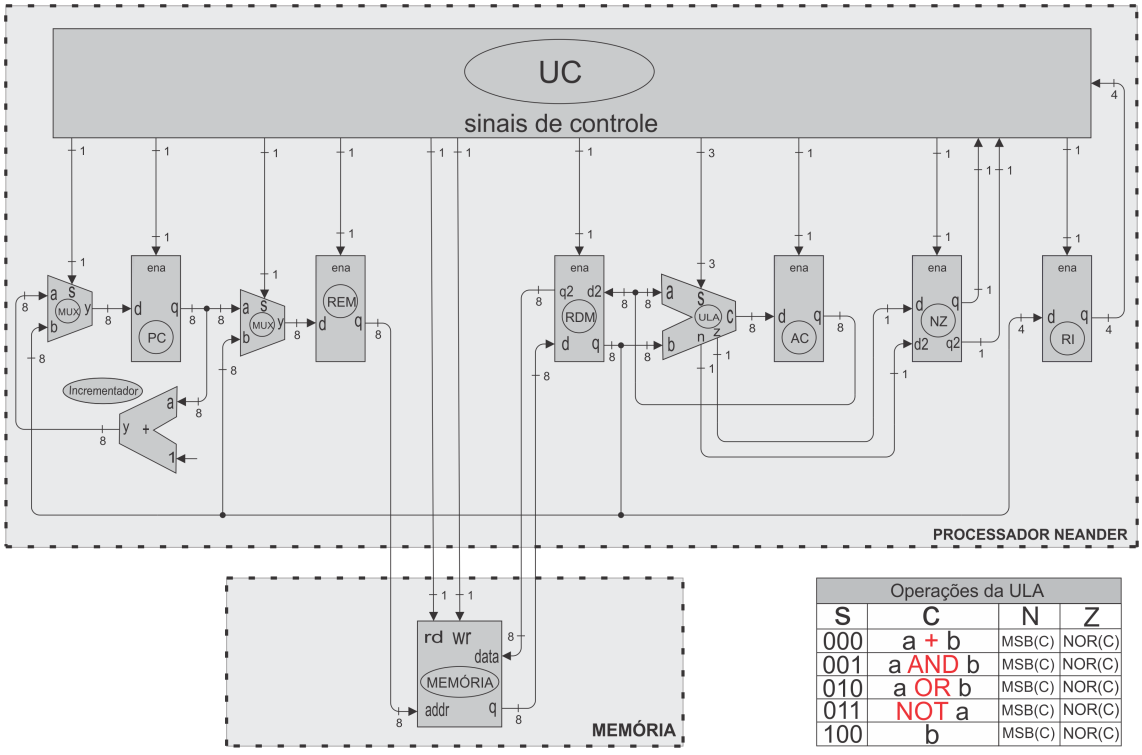

Figura 4.2: Interligação dos componentes que compõe o Neander.

Como já foi informada a memória não faz parte da composição do processador, por isso ela esta separada. Os números nas extremidades das setas representam a quantidade de bits das entradas e saídas dos componentes. Nas setas menores só há um numero, e nesse caso, o número representa o valor da entrada e da saída que a mesma interliga. Todas as setas representam a interligação dos componentes e mostram o sentido que a informação trafega.

O registrador RST esta com o nome NZ para facilitar a identificação dos estados que o mesmo armazena. Os registradores PC e REM são os únicos registradores que recebem dados de duas fontes diferentes. Para solucionar esse problema foram adicionados na interligação dois multiplexadores. Entretanto, o multiplexador do PC pode ser removido se forem feitas algumas alterações nas ligações. Por exemplo, o incremento do PC em vez de usar um somador próprio, poderia ser substituído por um registrador contador, assim o próprio PC iria fazer o incremento e o multiplexador seria removido. Outra maneira de incrementar o PC seria usando a ULA em vez do somador próprio, mas dessa forma continuaria a necessidade do multiplexador. Isso demonstra a possibilidade de pelo menos três diferentes organizações que impactam diretamente no consumo de recursos e na lógica da Unidade de Controle. Essa organização foi escolhida pela simplicidade de implementação.

A UC é desenvolvida utilizando a teoria de máquina de estado e é responsável por gerar todos os sinais de controle dos componentes do Neander e da memória, além de gerenciar os tempos de *clk* em que cada componente vai ser habilitado. As informações que são enviadas em suas entradas, também influenciam no tipo de estado que vai ser usado e que consequentemente ira afetar a sequência dos sinais de controle. Os sinais de controle tem a função de informar aos componentes que caminho ou operação serão escolhidos, se irão permitir ou não que a informação prossiga adiante e se a memória vai ser lida ou escrita. Além dos sinais de controle gerados pela UC, o Neander também possui sinais de controle externos, como o *rst* e o *clk*.

O *rst* é responsável por zerar as saídas dos registradores. Além disso, quando a UC é resetada os sinais de controle dos registradores e da memória também são zerados para que nenhuma informação circule durante o reinício do processamento. O *clk* tem a função de sincronizar a UC, os registradores e a memória, pois através dessa sincronia a UC consegue gerenciar os instantes em que a memória e os registradores serão utilizados para que ambos não enviem informações em momentos errados. Detalhando mais as informações, os registradores e a memória transferem as informações de suas entradas para as saídas na borda de subida do clock. Na UC também é feito na borda de subida do clock a transferência do valor dos sinais de controle para os componentes. Já a mudança dos estados da FSM é feito na borda de descida (WEBER, 2008). A figura 4.3 mostra a interligação dos componentes do Neander junto com os

sinais de controles externos.

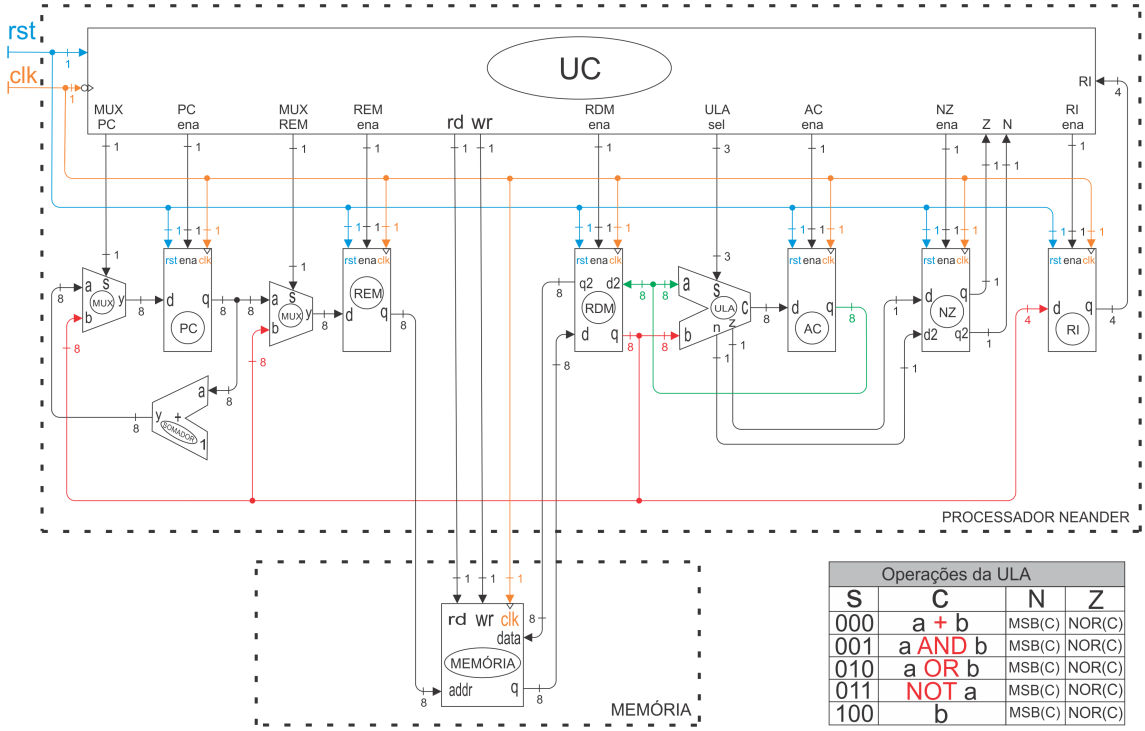

Figura 4.3: Interligação dos componentes do Neander com os sinais de controle externos.

4.4 Ciclo de busca e execução das instruções apartir dos registradores

O ciclo de busca apartir dos registradores é igual para todas as instruções do Neander e é mostrada na tabela 4.2.

Ciclo	Execução apartir dos registradores
Busca	REM ← PC
	Read; PC ← PC + 1
	RI ← RDM

Tabela 4.2: Ciclo de busca através dos registradores.

O ciclo de execução das instruções apartir dos registradores é ilustrada na tabela 4.3 no qual cada instrução segue um determinado caminho, com exceção do NOP que não faz nada. Após a execução da instrução o processador inicia o ciclo de busca novamente, pois como já foi mencionado esse processo sempre irá se repetir até que seja executada uma instrução que termine a execução (HLT).

Legenda:

← = Atribuição

; = Separador de ações executadas em paralelo no mesmo de tempo de clock.

Read = Sinal de controle de leitura da memória.

Write = Sinal de controle de escrita da memória.

Ciclo	**Execução apartir dos registradores**	
NOP	Não executa nada	
STA	REM ← PC Read; PC ← PC + 1 REM ← RDM RDM ← AC Write	
LDA	REM ← PC Read; PC ← PC + 1 REM ← RDM Read AC ← RDM; atualiza N e Z	
ADD OR AND	REM ← PC Read; PC ← PC + 1 REM ← RDM Read AC ← AC (+, OR, AND) RDM; atualiza N e Z	
NOT	AC ← AC NOT(AC); atualiza N e Z	
JMP	REM ← PC Read PC ← RDM	
JN	**Se N=1 (desvio ocorre)** REM ← PC Read PC ← RDM	**Se N=0 (desvio não ocorre)** PC ← PC + 1
JZ	**Se Z=1 (desvio ocorre)** REM ← PC Read PC ← RDM	**Se Z=0 (desvio não ocorre)** PC ← PC + 1
HLT	Termina execução	

Tabela 4.3: Ciclo da execução das Instruções através dos registradores.

4.5 Ciclo de busca e execução das instruções apartir dos componentes

Para facilitar a compreensão do ciclo de busca e execução das instruções apartir dos componentes, todas as figuras usadas de exemplo são baseadas na interligação dos componentes do Neander sem os sinais de controles externos.

4.5.1 Ciclo de busca

O ciclo de busca apartir dos componentes é mostrado na figura 4.4.

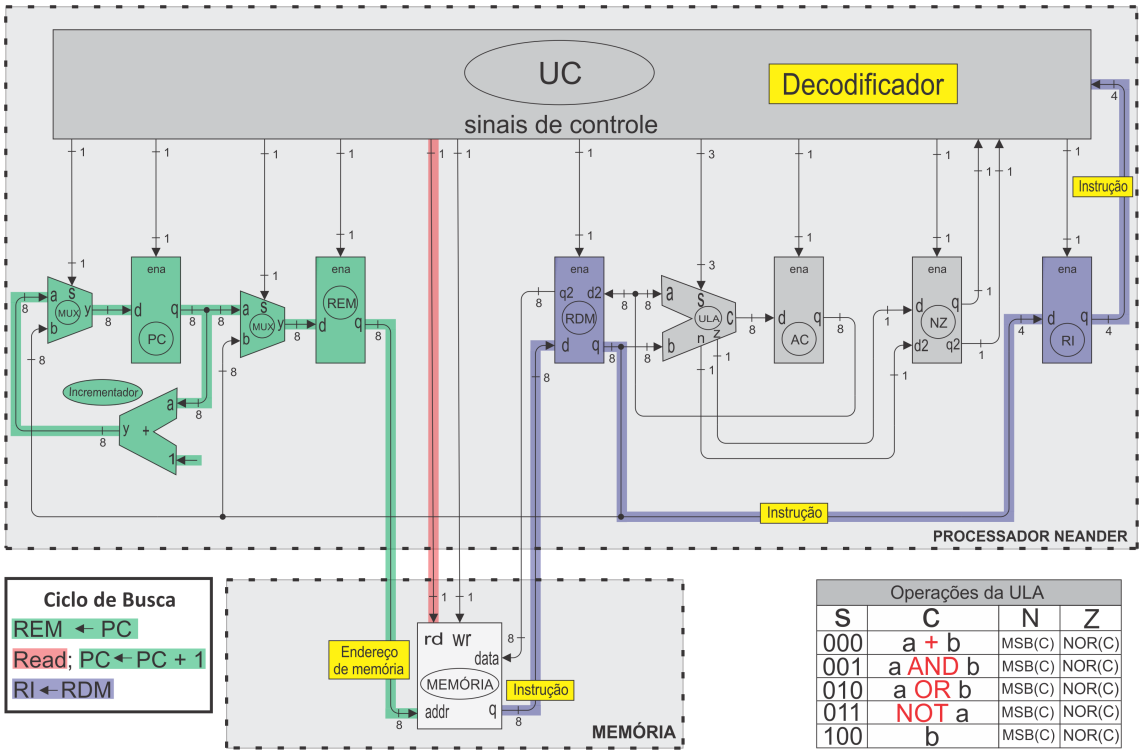

Figura 4.4: Ciclo de busca apartir dos componentes.

O ciclo de busca começa quando o Neander é iniciado ou resetado (ambos pelo sinal externo *rst*), no qual todos os registradores iniciam com o valor zero em suas saídas. Dessa forma o primeiro endereço de memória que o PC sempre vai apontar vai ser o endereço zero. Assim o ciclo de busca se inicia apartir do PC que envia o valor zero para o REM. O REM informa a memória qual é endereço escolhido, então é ativado o *rd* pra ler o endereço e descobrir o valor do seu conteúdo. O valor do conteúdo é então repassado ao RDM que envia para o RI. O RI encaminha para a UC que decodifica a instrução, preparando os instantes de tempo e os sinais de controle necessários para a execução da mesma. Assim é executada a fase de busca. Após uma leitura na memória apontada pelo PC, o conteúdo desse registrador deve ser incrementado

para apontar para o próximo endereço de memória. Essa operação pode ser feita a qualquer instante de tempo após o envio do valor do PC para o REM, mas também deve ser feito antes do próximo uso do PC. Na descrição o incremento é feito junto com a ação de leitura (*rd*) da memória.

4.5.2 Instrução STA, LDA, ADD, OR e AND.

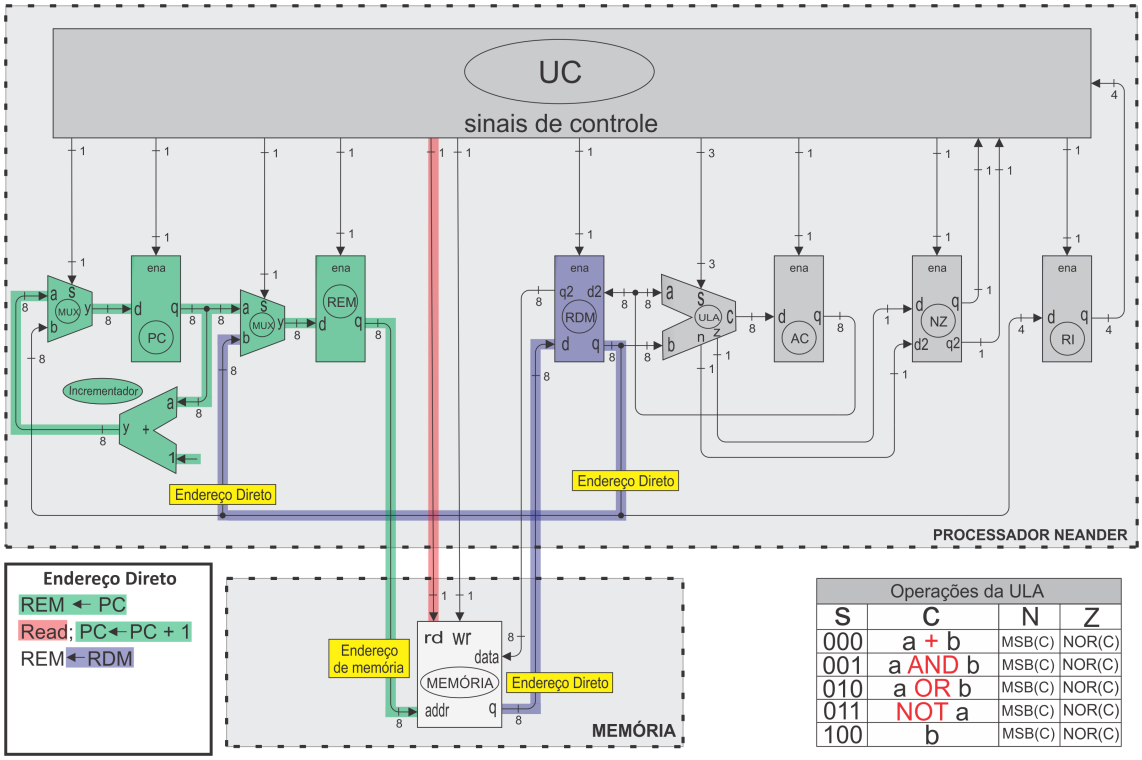

Figura 4.5: Execução da segunda palavra do STA, LDA, ADD, OR e AND - parte 1.

A ilustração da execução das instruções de duas palavras STA, LDA, ADD, OR e AND apartir dos componentes é dividida em duas partes. O ciclo de busca já apresentado decodifica a primeira palavra dessas instruções e identifica qual é a instrução. A figura 4.5 mostra a primeira parte de como a segunda palavra dessas instruções que são compostas pelo endereço direto é executada. Ela é semelhante com o ciclo de busca, contudo quando o conteúdo do endereço de memória é transferido ao RDM, o mesmo dessa vez repassa a informação ao REM. Essa informação contém o endereço direto que representa um endereço de memória e através do REM é que se descobre onde se encontra o operando contido nesse endereço que é usado junto a essas instruções. No caso da instrução STA o endereço direto informa ao REM qual endereço vai receber a escrita (*wr*) de um operando. Novamente após uma leitura na memória apontada pelo PC, o conteúdo desse registrador deve ser incrementado para apontar para o próximo endereço de memória.

A segunda parte mostra o fim do ciclo de execução da segunda palavra de cada uma dessas instruções. A figura 4.6 apresenta a segunda parte da instrução STA. Como já informado, o endereço direto informa ao REM qual é o endereço de memória que é selecionado para receber a escrita (*wr*) de um operando. O operando é repassado do AC para o RDM. Assim com o REM e o RDM preparados com as informações, à memória só finaliza ativando a escrita (*wr*).

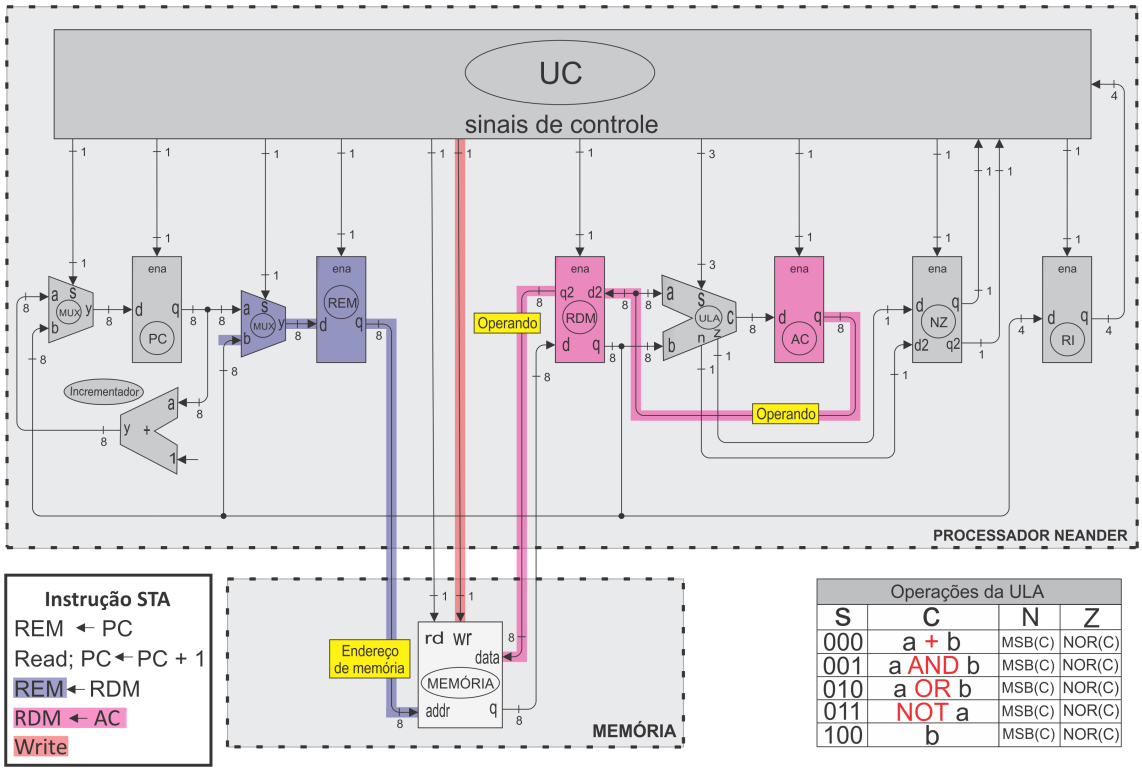

Figura 4.6: Execução da segunda palavra da instrução STA - parte 2.

Já a figura 4.7 mostra a segunda parte da instrução ADD. Como mencionado o REM transfere para a memória o valor do endereço direto, ela lê (*rd*), encontra o operando contido nesse endereço e repassa ao RDM. O RDM depois repassa para a ULA no qual também recebe um operando do AC. A ULA então ativa o sinal de controle com o código de operação correspondente a instrução ADD (000) e assim é processada a operação. O resultado é repassado ao AC que dessa forma possui um novo valor de operando. Para finalizar, através das saídas de códigos de condições, a ULA informa ao NZ que depois repassa a UC, se o resultado processado é zero ou não e se é negativo ou positivo. Essas informações são fornecidas a UC para serem usadas pelas instruções JN e JZ quando elas forem decodificadas. Uma observação é que essa ilustração também serve para as instruções LDA, OR e AND, pois a única diferença para cada uma dessas instruções é o código de operação da ULA.

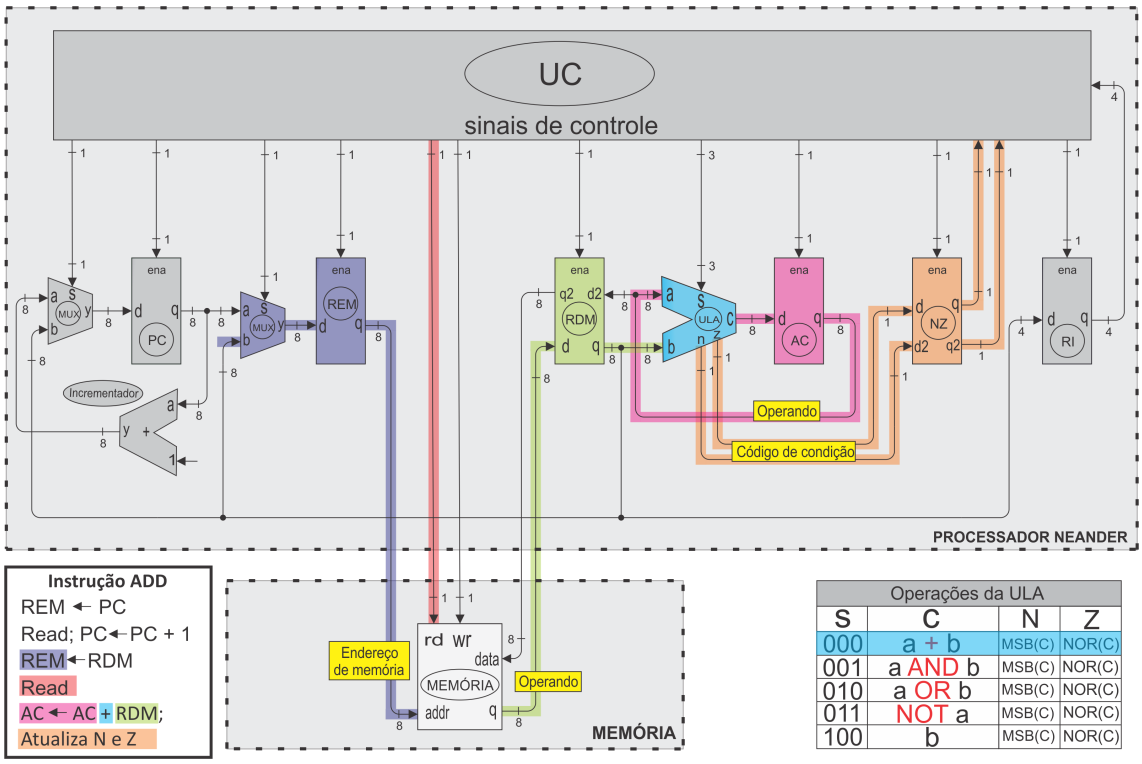

Figura 4.7: Execução da segunda palavra da instrução ADD.

4.5.3 Instrução NOT

O ciclo de execução da instrução de uma palavra NOT é mostrado na figura 4.8.

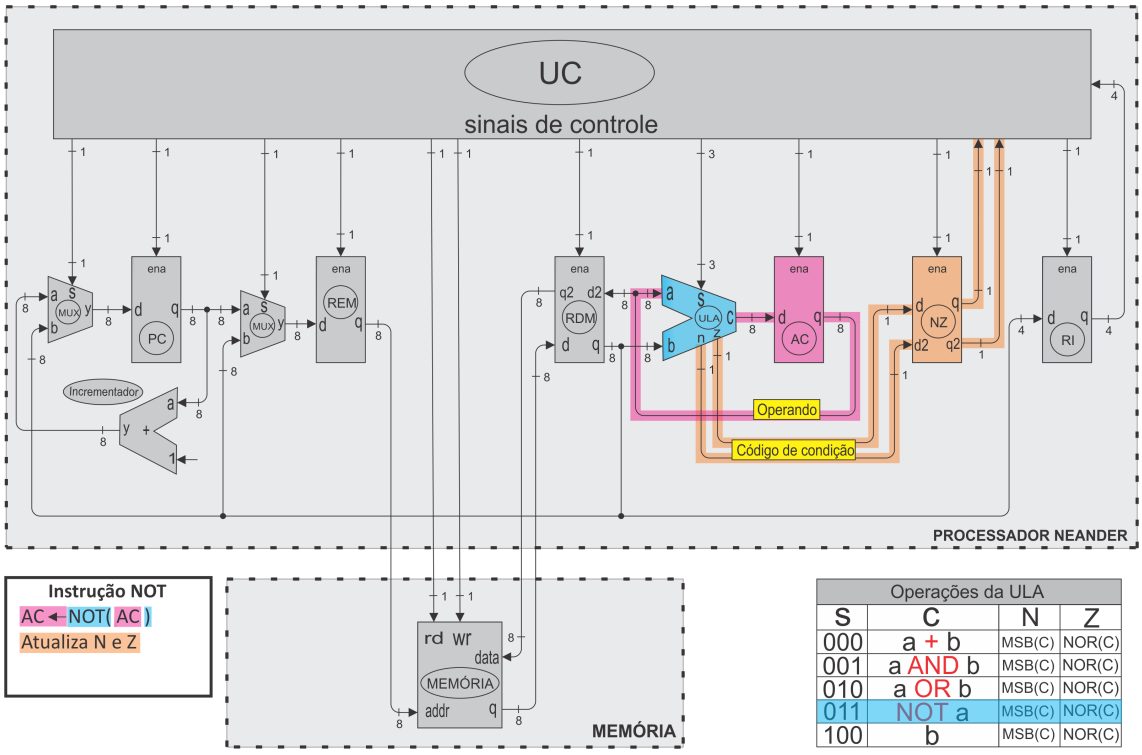

Figura 4.8: Instrução NOT apartir dos componentes.

Depois que o ciclo de busca é executado e identifica essa instrução, a execução da mesma começa com o AC repassando um operando para a ULA. Ela então ativa o sinal de controle com o código de operação da instrução NOT (011) e executa a operação da mesma. Depois de processado o resultado é enviado ao AC que fica com um novo valor de operando. Novamente para finalizar, através das saídas de códigos de condições, a ULA repassa ao NZ que depois repassa a UC, se o resultado processado é zero ou não e se é negativo ou positivo. Essas informações são fornecidas a UC para serem usadas pelas instruções JN e JZ quando elas forem decodificadas.

4.5.4 Instrução JMP, JN (IF N=1) e JZ (IF Z=1)

O ciclo de execução das instruções de duas palavras JMP, JN (IF N=1) e JZ (IF Z=1) também é ilustrado em duas partes. Essas duas partes seguem a mesma lógica das instruções de duas palavras (STA, ADD,...) já apresentadas. Assim depois que o ciclo de busca executou a primeira palavra, a figura 4.9 mostra a primeira parte da execução da segunda palavra da instrução JMP, no qual essa representação também vale para as a instruções JN (IF N=1) e JZ (IF Z=1). O PC envia o valor do endereço para o REM, o REM repassa para a memória. A memória vai e lê (*rd*) esse endereço.

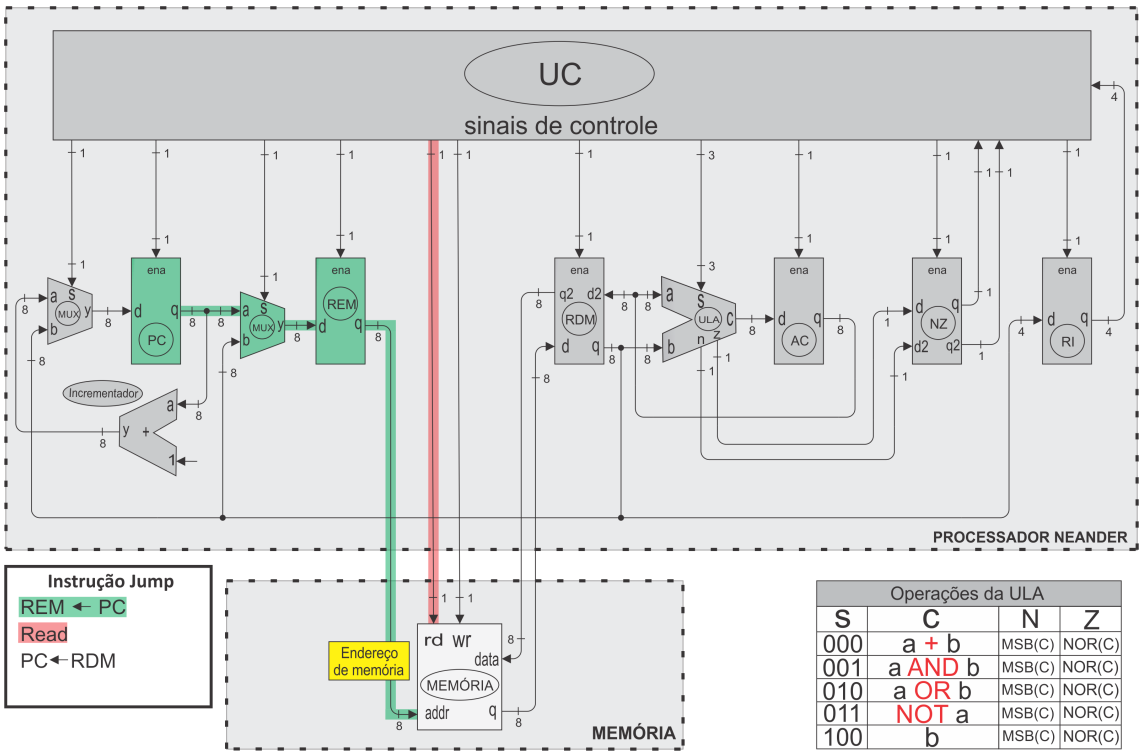

Figura 4.9: Execução da instrução de duas palavras JMP - parte 1.

A figura 4.10 ilustra a segunda parte da execução da segunda palavra da instrução JMP. O conteúdo do endereço lido que é um endereço direto é passado ao RDM. O RDM então envia o valor do endereço para o PC. Dessa forma o PC não é incrementado, interrompendo a sequencia dos endereços e assim ocorre o desvio para o endereço de memória escolhido onde se encontra a próxima instrução a ser executada.

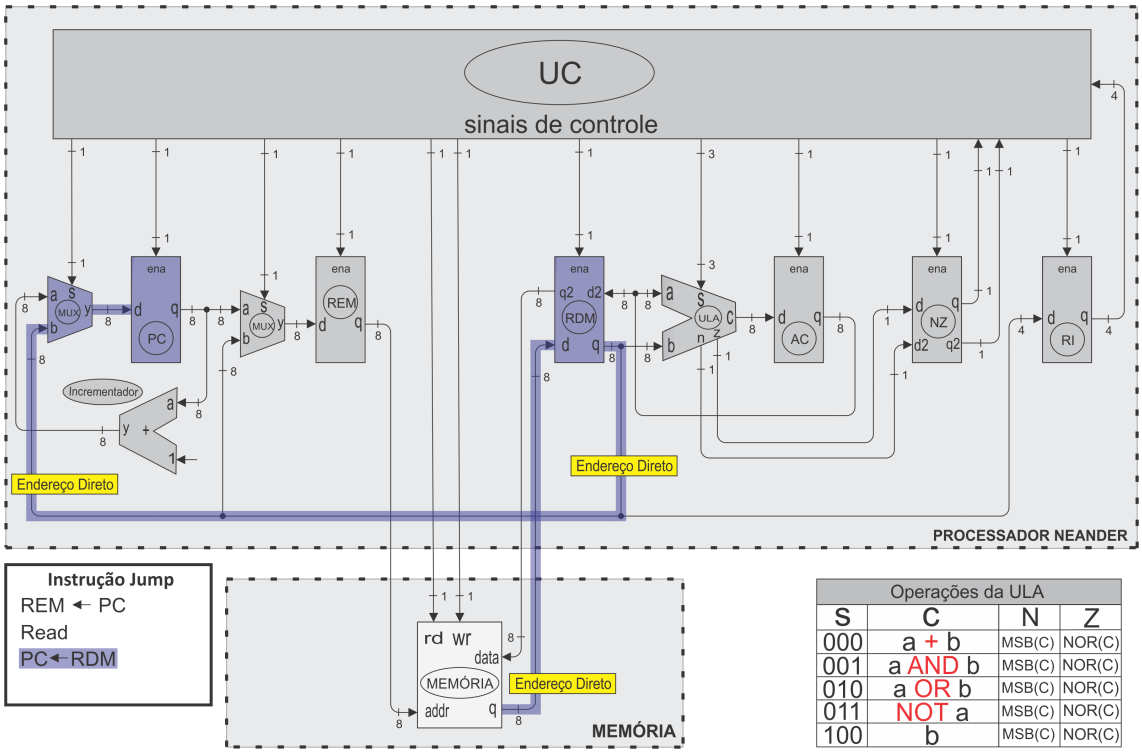

Figura 4.10: Execução da instrução de duas palavras JMP - parte 2.

4.5.5 Instrução JN (IF N=0) e JZ (IF Z=0)

O ciclo de execução das instruções JN (IF N=0) e JZ (IF Z=0) quando o desvio condicional não se realiza é representado na figura 4.11. Se não é preciso fazer o desvio, então o que não pode ocorrer é a leitura o valor do endereço direto. Assim, basta incrementar o valor do PC, para que o mesmo pule sobre o endereço de memória que contém o endereço direto e passe a apontar para a instrução seguinte.

4.5.6 Instrução HLT

Por ultimo o ciclo de execução da instrução de uma palavra HLT é mostrado na figura 4.12. Sua execução faz com que a UC desative todos os componentes, assim nenhuma informação irá trafegar no processador.

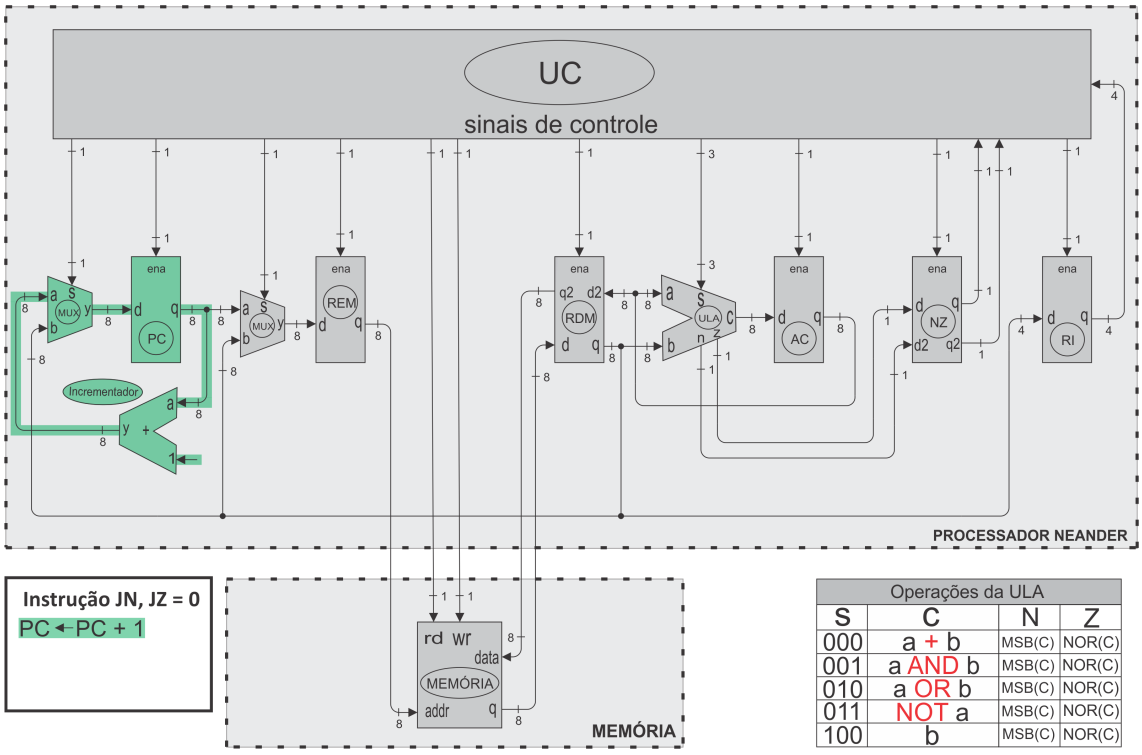

Figura 4.11: Instrução JN (IF N=0) e JZ (IF Z=0) quando o desvio não acontece.

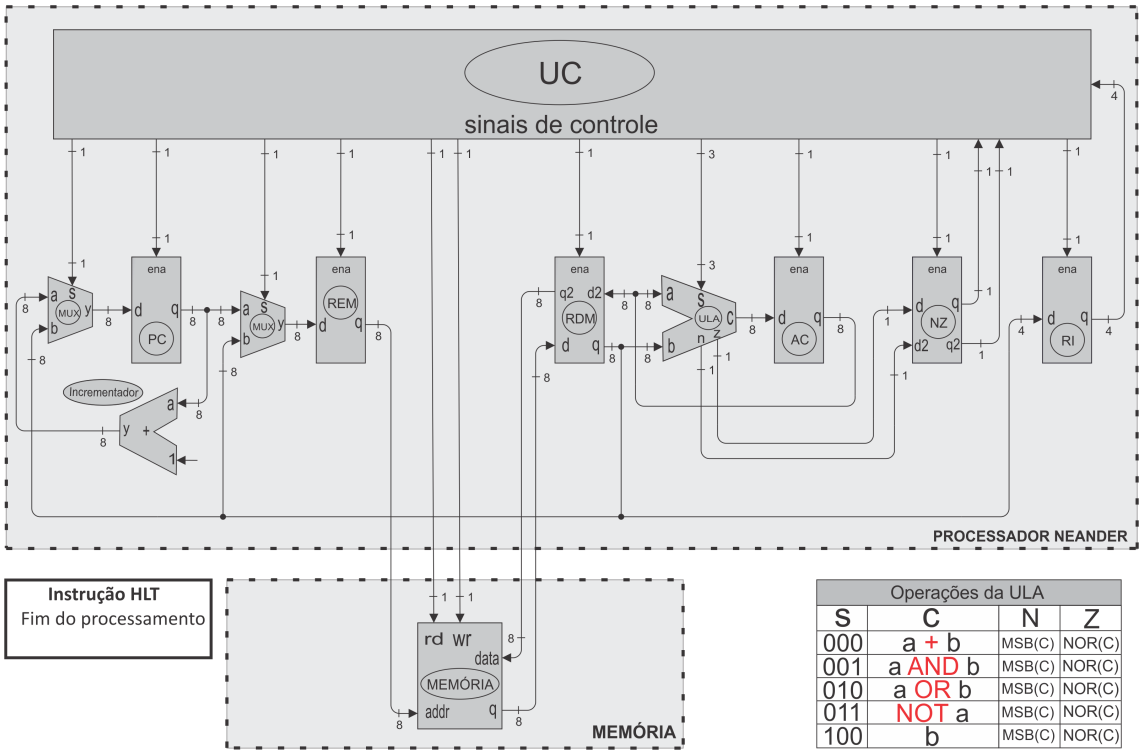

Figura 4.12: Instrução HLT apartir dos componentes.

4.6 Programa processado através dos registradores

Entendendo como o ciclo de busca e de execução das instruções funciona através dos componentes do Neander, o próximo passo é ver como o programa que soma três posições consecutivas e guarda o resultado numa quarta posição já visto na seção 3.8 é processado apartir dos registradores. A tabela 4.4 exibe novamente o programa, e a tabela 4.5 exibe o mesmo carregado na memória em linguagem de máquina (Hexadecimal).

Programa	
Mnemônico	Comentários
LDA 128	Copia o operando do end.128 para o AC
ADD 129	Soma o operando do end.129 com o operando do AC
ADD 130	Soma o operando do end.130 com o operando do AC
STA 131	Copia o operando do AC para o end.131
HLT	Termina execução

Tabela 4.4: Programa que soma três posições consecutivas.

MEMÓRIA - 8 bits			
Área das Instruções		Área dos Operandos	
end	**Conteúdo**	**end**	**Conteúdo**
00H	20H	80H	w
01H	80H	81H	x
02H	30H	82H	y
03H	81H	83H	z
04H	30H	84H	
05H	82H	85H	
06H	10H	86H	
07H	83H	87H	
08H	F0H	...	...
...	..	FFH	

Tabela 4.5: Programa na memória em linguagem de máquina (hexadecimal).

A tabela 4.6 mostra como o programa carregado na memória representado em linguagem de máquina (hexadecimal) é processado através dos registradores.

Ciclo	Execução apartir dos registradores
Busca	REM (00H) ← PC (00H) Read; PC (01H) ← PC (00H) + 1 RI (20H) ← RDM (20H)
LDA	REM (01H) ← PC (01H) Read; PC (02H) ← PC (01H) + 1 REM (80H) ← RDM (80H) Read AC (w) ← RDM (w); atualiza N e Z
Busca	REM (02H) ← PC (02H) Read; PC (03H) ← PC (02H) + 1 RI (30H) ← RDM (30H)
ADD	REM (03H) ← PC (03H) Read; PC (04H) ← PC (03H) + 1 REM (81H) ← RDM (81H) Read AC (A) ← AC (w) (+) RDM (x); atualiza N e Z
Busca	REM (04H) ← PC (04H) Read; PC (05H) ← PC (04H) + 1 RI (30H) ← RDM (30H)
ADD	REM (05H) ← PC (05H) Read; PC (06H) ← PC (05H) + 1 REM (82H) ← RDM (82H) Read AC (A) ← AC (A) (+) RDM (y); atualiza N e Z
Busca	REM (06H) ← PC (06H) Read; PC (07H) ← PC (06H) + 1 RI (10H) ← RDM (10H)
STA	REM (07H) ← PC (07H) Read; PC (08H) ← PC (07H) + 1 REM (83) ← RDM (83H) RDM (z) ← AC (A) Write
Busca	REM (08H) ← PC (08H) Read; PC (09H) ← PC (08H) + 1 RI (F0H) ← RDM (F0H)
HLT	Termina execução

Tabela 4.6: Programa processado através dos registradores.

4.7 Temporização do ciclo de busca e das Instruções

Com o objetivo de explicar o funcionamento do ciclo de busca e execução das instruções de forma mais direta, a exibição das figuras de ambos os ciclos foi demonstrado de maneira resumida. Porém é gerada a falsa impressão de que os componentes utilizados para o processamento das informações são executados quase que todos em um só ciclo de clock, mas na prática não é isso que acontece. Os componentes como ULA, incrementador e os multiplexadores que são assíncronos, são os únicos que executam suas funções imediatamente. Já os registradores, a memória e a UC que são síncronos, só repassam a informação de suas entradas para as saídas no próximo ciclo do clock. No caso dos registradores e da memória, depois que o sinal de controle da UC permite a passagem da informação da entrada para a saída, o resultado só é válido no próximo ciclo de clock. Em relação à UC como já foi mencionado, o envio do valor dos sinais de controle para os componentes é feito na borda de subida do clock e as mudanças dos estados são feitas na borda de descida (WEBER, 2008).

Assim a seguir é apresentada a organização dos sinais de controle levando em consideração os ciclos de clock para a execução da busca e das instruções no Neander. Os sinais de controle utilizados nessa organização, isto é, os sinais de controle externos (*rst* e *clk*) e da UC do Neander são mostrados novamente na figura 4.13.

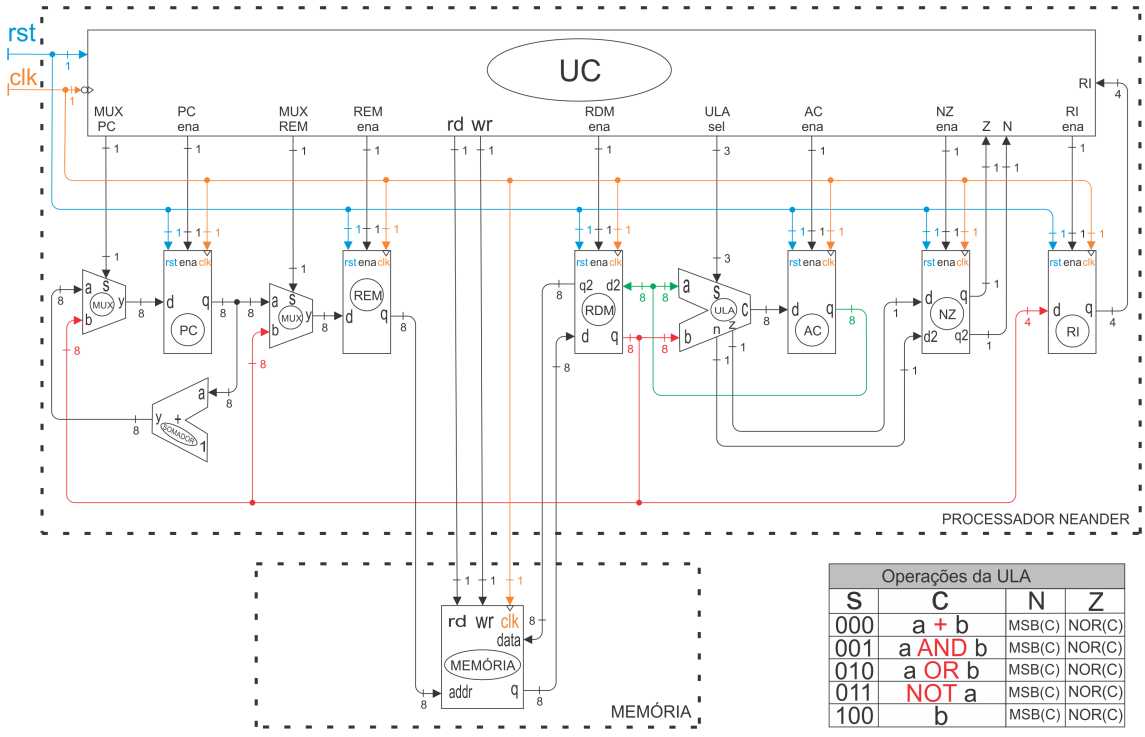

Figura 4.13: Sinais de controle externos e da UC do Neander.

Tudo começa a partir dos sinais de controle externos clock e reset. O clock sincroniza todos

os componentes síncronos e o reset além de zerar as saídas dos registradores também configura os sinais de controle da UC para que nenhuma informação circule no processador, ou seja, desativa todos os componentes.

- MUX PC = X
- PC ena = 0
- MUX REM = X
- REM ena = 0
- rd = 0
- wr = 0
- RDM ena = 0
- ULA sel = XXX
- AC ena = 0
- NZ ena = 0
- RI ena= 0

Depois que os componentes estão sincronizados e os sinais de controle estão carregados com esses valores, a UC só vai gerenciar os ciclos e clock em que ela vai escolher qual caminho ou operação será selecionada, se ira permitir ou não que a informação prossiga adiante e se a memória vai ser lida ou escrita.

As tabelas 4.7, 4.8 e 4.9 descrevem os ciclos de clock em que foi dividido o ciclo de busca, a execução de cada uma das instruções e os sinais relevantes para aquele ciclo de clock, ou seja, só são mostrados os sinais de controle que serão ativados para que a informação circule assim os sinais restantes vão estar desativados.

O ciclo de busca e a decodificação ocorrem nos ciclos de clock de t0 á t5, por isso todas as instruções tem os mesmos sinais de controle e valores nesses respectivos ciclos. Os ciclos de clock seguintes mostram a execução das instruções. Também é mostrado que nem todas as instruções possuem os mesmo numero de ciclos clock para concluir sua respectiva execução. Ao final da execução, com exceção da instrução HLT, o processador retorna para o ciclo de clock t0 e inicia o ciclo de busca novamente.

Clk	STA	LDA	ADD	OR	AND
t0	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1
t1	rd = 1	rd = 1	rd = 1	rd = 1	rd = 1
t2	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1
t3	RI ena = 1	RI ena = 1	RI ena = 1	RI ena = 1	RI ena = 1
t4	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1
t5	Todos desativados	Todos desativados	Todos desativados	Todos desativados	Todos desativados
t6	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1
t7	rd = 1	rd = 1	rd = 1	rd = 1	rd = 1
t8	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1
t9	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1
t10	MUX REM = 1 REM ena = 1	MUX REM = 1 REM ena = 1	MUX REM = 1 REM ena = 1	MUX REM = 1 REM ena = 1	MUX REM = 1 REM ena = 1
t11	RDM ena = 1	rd = 1	rd = 1	rd = 1	rd = 1
t12	wr = 1 Volta para t0	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1
t13		ULA sel = 100 AC ena = 1 NZ ena = 1 Volta para t0	ULA sel = 000 AC ena = 1 NZ ena = 1 Volta para t0	ULA sel = 010 AC ena = 1 NZ ena = 1 Volta para t0	ULA sel = 001 AC ena = 1 NZ ena = 1 Volta para t0

Tabela 4.7: Temporização dos sinais de controle da UC com os ciclos de clock - parte 1.

Clk	JUMP	JN, N=1	JN, N=0	JZ, Z=1	JZ, Z=0
t0	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1
t1	rd = 1	rd = 1	rd = 1	rd = 1	rd = 1
t2	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1	RDM ena = 1
t3	RI ena = 1	RI ena = 1	RI ena = 1	RI ena = 1	RI ena = 1
t4	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1
t5	Todos desativados	Todos desativados	Todos desativados	Todos desativados	Todos desativados
t6	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX PC = 0 PC ena = 1 Volta para t0	MUX REM = 0 REM ena = 1	MUX PC = 0 PC ena = 1 Volta para t0
t7	rd = 1	rd = 1		rd = 1	
t8	RDM ena = 1	RDM ena = 1		RDM ena = 1	
t9	MUX PC = 1 PC ena = 1 Volta para t0	MUX PC = 1 PC ena = 1 Volta para t0		MUX PC = 1 PC ena = 1 Volta para t0	

Tabela 4.8: Temporização dos sinais de controle da UC com os ciclos de clock - parte 2.

Clk	NOP	NOT	HLT
t0	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1	MUX REM = 0 REM ena = 1
t1	rd = 1	rd = 1	rd = 1
t2	RDM ena = 1	RDM ena = 1	RDM ena = 1
t3	RI ena = 1	RI ena = 1	RI ena = 1
t4	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1	MUX PC = 0 PC ena = 1
t5	Todos desativados	Todos desativados	Todos desativados
t6	Volta para t0	ULA sel = 011 AC ena = 1 NZ ena = 1 Volta para t0	Todos desativados

Tabela 4.9: Temporização dos sinais de controle da UC com os ciclos de clock - parte 3.

Dessa forma essas tabelas exibem como realmente acontece a ativação de cada componente para a execução do ciclo de busca e da execução das instruções. Pois anteriormente foi mostrada a execução da busca e das instruções apartir dos componentes de forma resumida, mas a razão dessa escolha surgiu pelo fato de haver muitos ciclos de clock na busca e nas instruções e isso acarretaria em muitas figuras para a demonstração.

4.8 FSM do Neander

Como já descrito a lógica da UC é composta por uma FSM. Dessa maneira a FSM da UC do Neander foi criada apartir das informações das tabelas de temporizações dos ciclos de clock já apresentadas. A figura 4.14 ilustra como foi à criação da FSM através dessas tabelas. Cada círculo com o retângulo em cima representa um estado com suas respectivas ações, no qual essas ações são os valores dos sinais de controle da UC. Cada estado representa um ciclo de clock das respectivas tabelas e abaixo de seus nomes há a indicação do respectivo clock. Os ciclos de clock não foram usados como os nomes dos estados, pois nem todos os ciclos de clock de todas as instruções possuem em comum os mesmos valores dos sinais de controle. Entretanto foram escolhidos nomes que já indicam o papel de cada estado na execução da FSM. O estado RESET monitora o sinal de controle externo *rst*, os estados B0 à B4 são responsáveis pelo ciclo de busca, o estado D cuida da decodificação, os estados E0 à E9 e os com os nomes das instruções são responsáveis pela execução das mesmas. A transição de um estado para outro ocorre a cada ciclo de clock e pode ou não depender de uma condição.

Assim a execução da FSM se inicia apartir do estado RESET que desativa os valores dos sinais de controle da UC. Apartir desses valores as ações dos outros estados também vão estar desativados com exceção dos sinais de controle relevantes de cada ciclo de clock que vão estar ativados como mostrado nas tabelas da seção passada.

Seguindo a sequencia dos estados, nos próximos ciclos de clock ocorre o ciclo de busca e a decodificação, pois como é visto as transições do estado B0 ao D não necessitam de nenhuma condição. Mas chegando ao estado D, para que a sequencia continue, agora além do clock também é preciso de uma condição, no qual essas condições são compostas pelos sinais de entrada RI, N e Z da UC. Dependendo do estado a ser seguido, pode ser preciso usar só o RI ou o RI junto com o N ou Z. A condição RI na FSM representa os OpCode das instruções do Neander e o N e o Z são os códigos de condição gerados pela ULA usados nas instruções JN e JZ.

Os estados seguintes ao D como já mencionado são todos responsáveis pela execução das instruções. A execução das instruções, com exceção do NOP que não faz nada e retorna para o ciclo de busca, pode ser feita apatir de só um estado no caso do NOT, JN (IF N=0), JZ (IF Z=0) e HLT ou podem ser executados por um conjunto de estados no caso das demais instruções STA, LDA, ADD, OR, AND, JMP, JN (IF N=0) e JZ (IF Z=0). Para criação desse conjunto de estados utilizou-se os sinais de controle em comum que as demais instruções executam no seu processamento. Dessa forma se economizaram estados e otimizou-se a FSM. Porém quando a instrução não possui mais sinais de controle em comum foram criados estados que ativam os sinais de controle necessário pra a sua execução. Após o processamento do último estado das instruções com exceção do HLT que termina a execução, acontece o retornou para o ciclo de busca e recomeça o processo da busca, decodificação e execução das instruções do processador Neander.

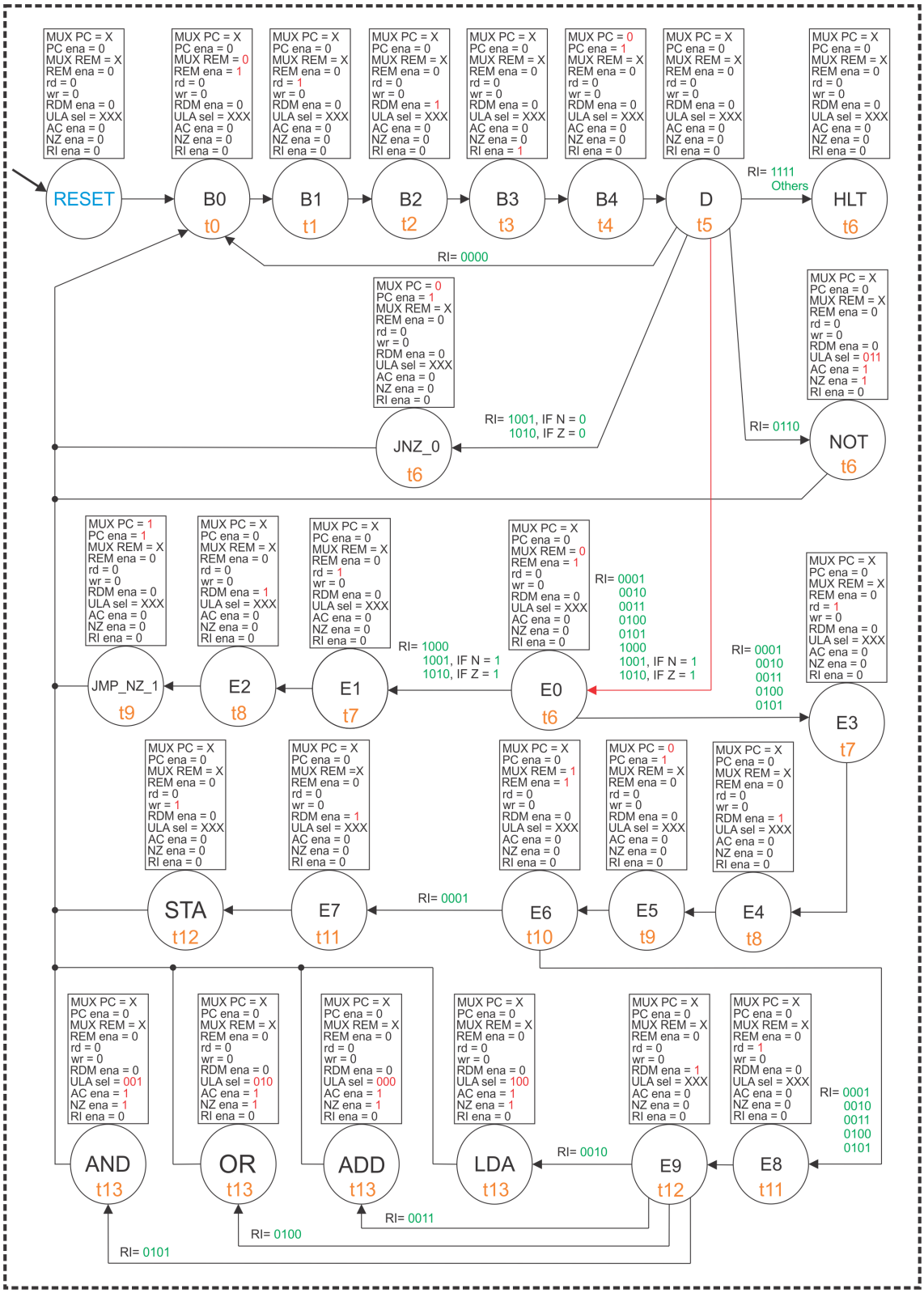

Figura 4.14: FSM da UC do Neander.

5 *Implementação em VHDL*

A implementação em VHDL do processador Neander iniciou-se apartir de etapas que são exibidas na figura 5.1 e descrevem como foi todo o processo. Uma observação sobre a memória é que ela foi criada através do Megawizard Plugin-in Manager, uma ferramenta do Quartus que automatiza a criação da memória em VHDL e assim não foi preciso escrever o seu código.

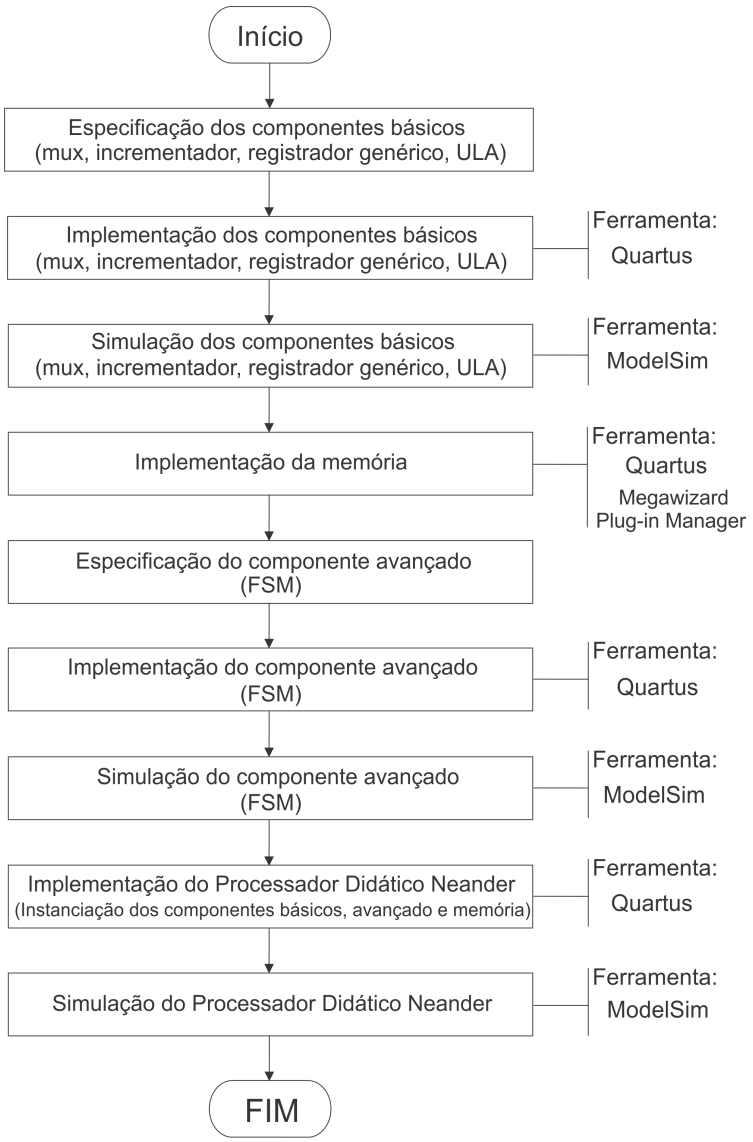

Figura 5.1: Etapas da implementação em VHDL do processador Neander.

6 *Simulação e Resultados*

São exibidos as simulações e resultados gerados por cada componente para validar seu funcionamento.

6.1 Simulação do Mux

Entendendo as características de funcionamento do Mux do Neander através da figura 6.1 e da tabela genérica mostrada na tabela 6.1 foi elaborado o vetor de teste do Mux exibido na tabela 6.2 . Apartir daí a próxima etapa foi a simulação do Mux no ModelSim configurando os valores das entradas de acordo com os do vetor de testes. Dessa forma a figura 6.2 mostrou que o resultado da simulação foi validado com o do vetor de teste do Mux.

A simulação foi feita através do Testbench tipo 1 (PEDRONI, 2010), no qual foi configurado que os valores das entradas só aparecem depois de um determinado ciclo de clock. Por isso no começo do teste os valores das entradas estão com os valores desconhecidos e não inicializados.

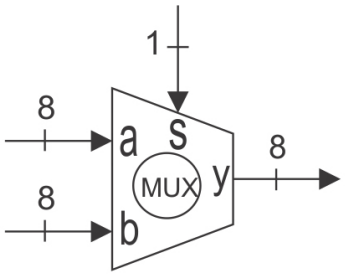

Figura 6.1: Mux do Neander.

s	y
0	a
1	b

Tabela 6.1: Tabela verdade genérica do Mux do Neander.

Entradas			Saída
a	b	s	y
00H	FFH	0	00H
00H	FFH	1	FFH

Tabela 6.2: Vetor de teste do Mux do Neander.

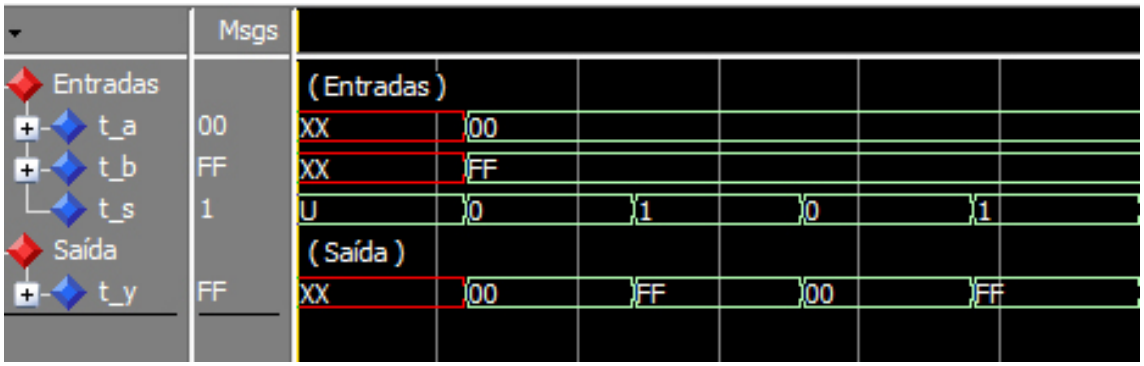

Figura 6.2: Simulação do Mux do Neander no ModelSim.

Legenda:

X = Valor desconhecido

U = Valor não inicializado

6.2 Simulação do Incrementador

O mesmo processo acontece no incrementador do Neander e os demais componentes, no qual se entendendo suas características de funcionamento apartir da figura 6.3 e da tabela genérica exibida na tabela 6.3, foi desenvolvido o vetor de teste do incrementador exibido na tabela 6.4. A etapa seguinte foi a simulação do incrementador no ModelSim configurando os valores das entradas de acordo com os do vetor de testes. A figura 6.4 ilustrou que o resultado da simulação foi validado com o do vetor de teste do incrementador.

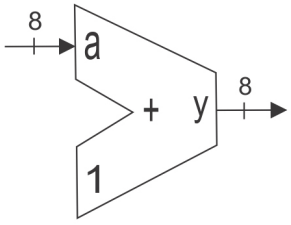

Figura 6.3: Incrementador do Neander.

y
a + 1

Tabela 6.3: Tabela verdade genérica do Incrementador do Neander.

Entrada	Saída
a	b
02H	03H
03H	04H
04H	05H

Tabela 6.4: Vetor de teste do Incrementador do Neander.

Figura 6.4: Simulação do Incrementador do Neander no ModelSim.

6.3 Simulação do Registrador genérico

A simulação do registrador genérico vale para ambos os tipos de registradores do Neander, ou seja, o de uma ou duas entradas e saídas. Porque as características de funcionamento são as mesmas, mudando só o número de informações que são transferidas da entrada para a saída. Assim entendendo as características de funcionamento do registrador genérico do Neander através da figura 6.5 e da tabela genérica apresentada na tabela 6.5 foi desenvolvido o vetor de teste desse registrador exibido na tabela 6.6. Apartir daí a próxima etapa foi a simulação do registrador genérico no ModelSim configurado os valores das entradas de acordo com os do vetor de testes. A figura 6.6 demonstrou que o resultado da simulação foi validado com o do vetor de teste do registrador genérico. Uma observação que já foi mencionada, é que como o registrador inicia o seu *clk* na sua subida, a transferência da informação da entrada para a saída quando o ena esta ativado e o *clk* esta na subida, só é realizada no próximo ciclo de *clk*.

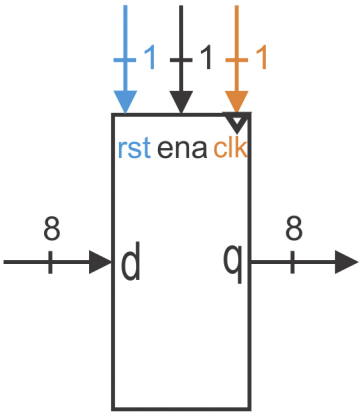

Figura 6.5: Registrador genérico do Neander.

clk	rst	ena	q
X	0	X	0
X	1	0	q0
↑	1	1	d
↓	1	1	q0

Tabela 6.5: Tabela verdade genérica do Registrador genérico do Neander.

Entradas				**Saída**
d	clk	rst	ena	q
01H	X	0	X	00H
01H	X	1	0	q0
01H	↑	1	1	01H
01H	↓	1	1	q0

Tabela 6.6: Vetor de teste do Registrador genérico do Neander.

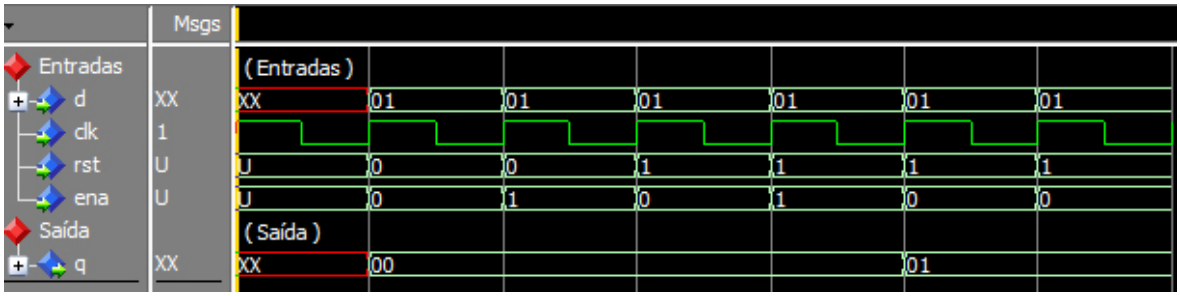

Figura 6.6: Simulação do Registrador genérico do Neander no ModelSim.

6.4 Simulação da ULA

Conhecendo as características de funcionamento da ULA do Neander apartir da figura 6.7 e da tabela genérica através da tabela 6.7, foi desenvolvido o vetor de teste da ULA exibido na tabela 6.8. A etapa seguinte foi a simulação da ULA no ModelSim configurando os valores das entradas de acordo com os do vetor de testes. A figura 6.8 mostrou que o resultado da simulação foi validado com o do vetor de teste da ULA.

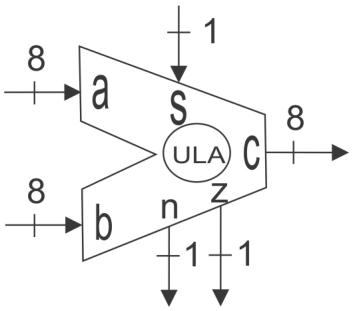

Figura 6.7: ULA do Neander.

s	c	n	z
000	a + b	MSB(c)	NOR(c)
001	a AND b	MSB(c)	NOR(c)
010	a OR b	MSB(c)	NOR(c)
011	NOT a	MSB(c)	NOR(c)
100	a	MSB(c)	NOR(c)

Tabela 6.7: Tabela verdade genérica da ULA do Neander.

Entradas			**Saídas**		
a	b	s	c	n	z
80H	80H	000	00H	0	1
80H	80H	001	80H	1	0
80H	80H	010	80H	1	0
80H	80H	011	7FH	0	0
80H	80H	100	80H	1	0
80H	80H	XXX	00H	0	1

Tabela 6.8: Vetor de teste da ULA do Neander.

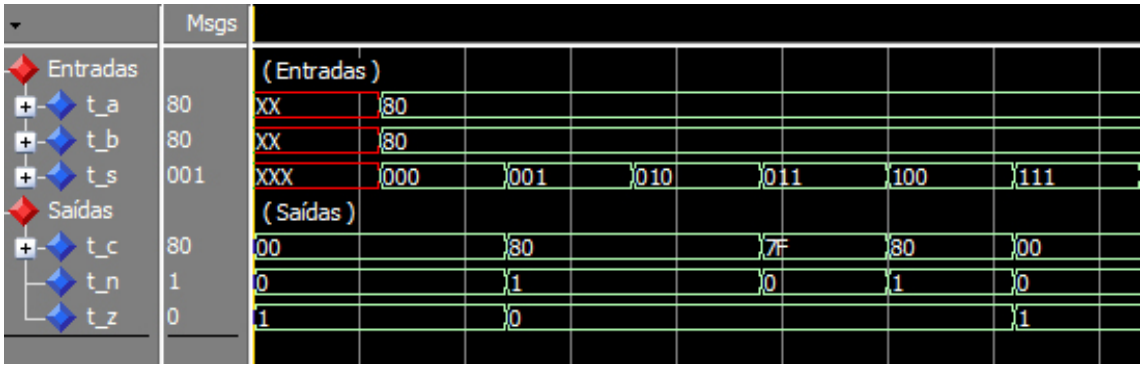

Figura 6.8: Simulação da ULA do Neander no ModelSim.

6.5 Simulação da UC

A simulação da UC foi executada com todas as instruções do Neander, contudo todas possuem a mesma lógica de execução, assim a simulação da UC a seguir foi feita apenas com a instrução LDA (0010). A figura 6.9 ilustra as características de composição da UC do Neander e apartir dela foi criado o vetor de teste através da instrução LDA que foi dividido em duas partes e exibido na tabela 6.9 e na tabela 6.10. A próxima etapa foi a simulação da UC no ModelSim configurando os valores das entradas de acordo com os do vetor de testes. A figura 6.10 e a figura 6.11 mostraram que o resultado da simulação foi validado com o do vetor de teste da UC.

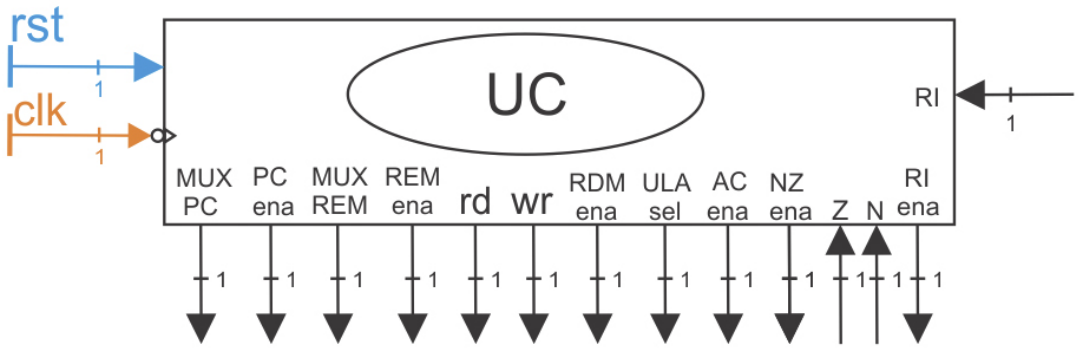

Figura 6.9: UC do Neander.

Entradas					Estados	
clk	rst	N	Z	RI	EA (Estado Atual)	PE (Próximo Estado)
Saídas (↑) / Estados (↓)	0	1	1	0010	RESET	B0
Saídas (↑) / Estados (↓)	1	1	1	0010	B0	B1
Saídas (↑) / Estados (↓)	1	1	1	0010	B1	B2
Saídas (↑) / Estados (↓)	1	1	1	0010	B2	B3
Saídas (↑) / Estados (↓)	1	1	1	0010	B3	B4
Saídas (↑) / Estados (↓)	1	1	1	0010	B4	D
Saídas (↑) / Estados (↓)	1	1	1	0010	D	E0
Saídas (↑) / Estados (↓)	1	1	1	0010	E0	E3
Saídas (↑) / Estados (↓)	1	1	1	0010	E3	E4
Saídas (↑) / Estados (↓)	1	1	1	0010	E4	E5
Saídas (↑) / Estados (↓)	1	1	1	0010	E5	E6
Saídas (↑) / Estados (↓)	1	1	1	0010	E6	E8
Saídas (↑) / Estados (↓)	1	1	1	0010	E8	E9
Saídas (↑) / Estados (↓)	1	1	1	0010	E9	LDA
Saídas (↑) / Estados (↓)	1	1	1	0010	LDA	B0

Tabela 6.9: Vetor de teste da UC do Neander com a instrução LDA (0010) - parte 1.

Estados	Saídas										
EA (Estado Atual)	MUX PC	PC ena	MUX REM	REM ena	rd	wr	RDM ena	ULA sel	AC ena	NZ ena	RI ena
RESET	X	0	X	0	0	0	0	XXX	0	0	0
B0	X	0	0	1	0	0	0	XXX	0	0	0
B1	X	0	X	0	1	0	0	XXX	0	0	0
B2	X	0	X	0	0	0	1	XXX	0	0	0
B3	X	0	X	0	0	0	0	XXX	0	0	1
B4	0	1	X	0	0	0	0	XXX	0	0	0
D	X	0	X	0	0	0	0	XXX	0	0	0
E0	X	0	0	1	0	0	0	XXX	0	0	0
E3	X	0	X	0	1	0	0	XXX	0	0	0
E4	X	0	X	0	0	0	1	XXX	0	0	0
E5	0	1	X	0	0	0	0	XXX	0	0	0
E6	X	0	1	1	0	0	0	XXX	0	0	0
E8	X	0	X	0	1	0	0	XXX	0	0	0
E9	X	0	X	0	0	0	1	XXX	0	0	0
LDA	X	0	X	0	0	0	0	100	1	1	0

Tabela 6.10: Vetor de teste da UC do Neander com a instrução LDA (0010) - parte 2.

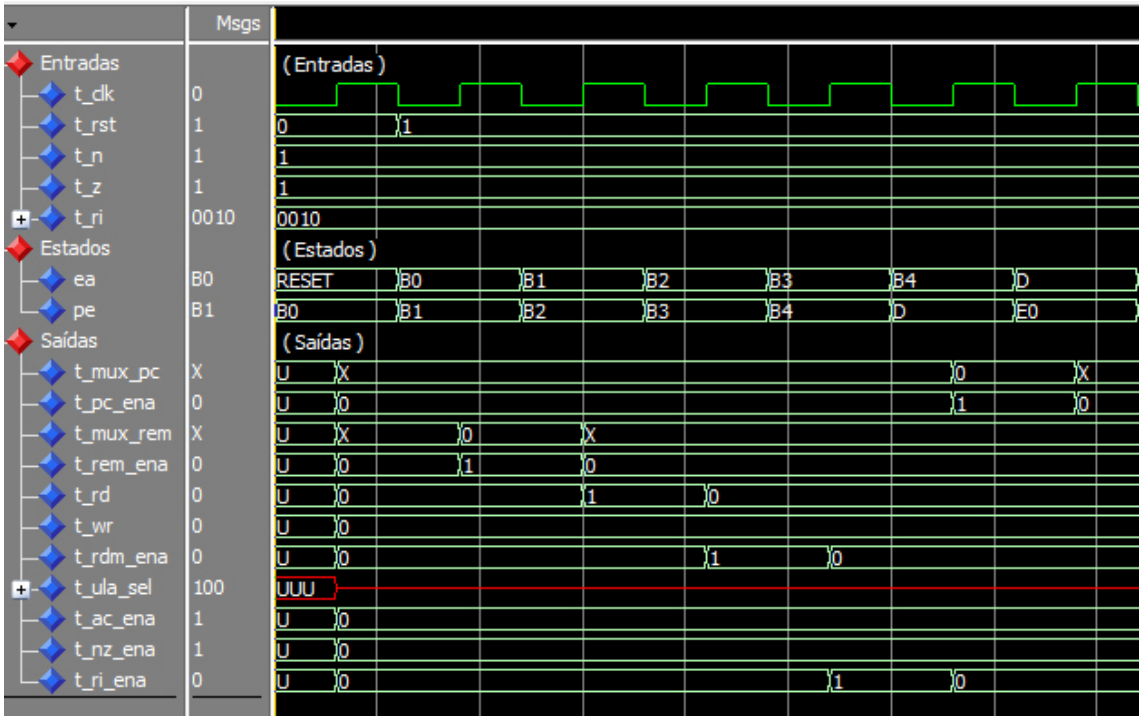

Figura 6.10: Simulação da UC do Neander com o LDA (0010) no ModelSim - parte 1.

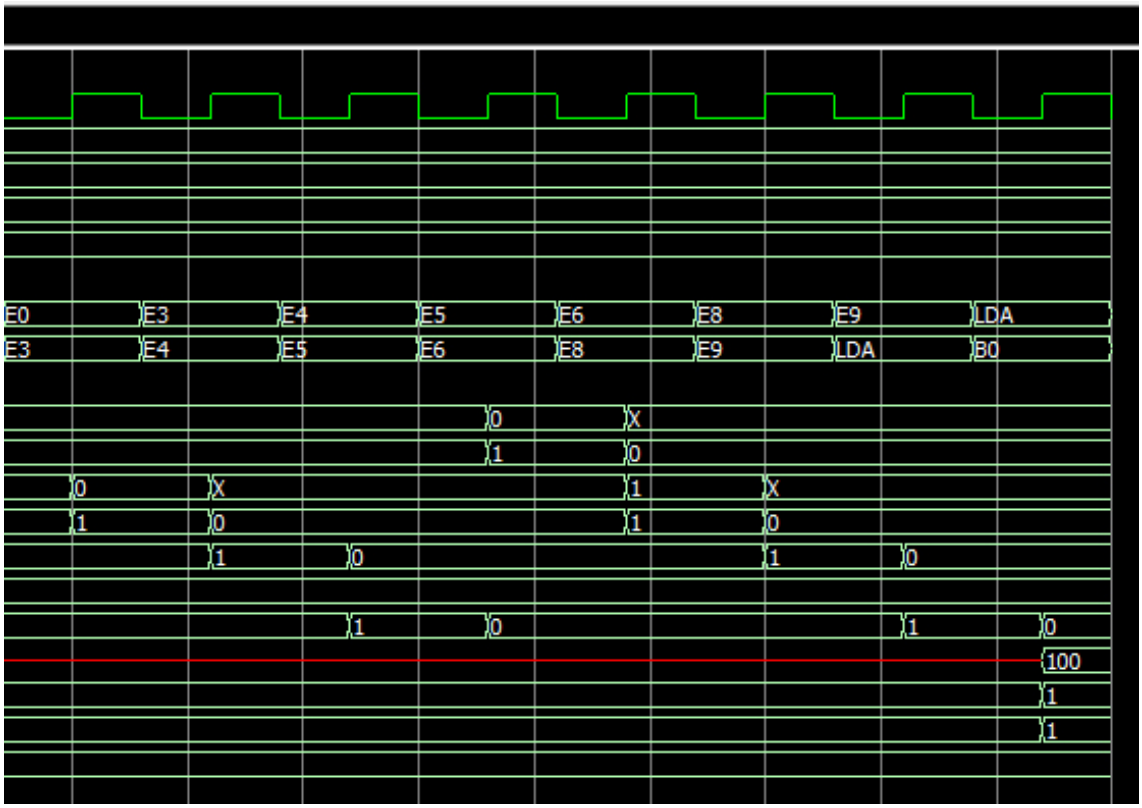

Figura 6.11: Simulação da UC do Neander com o LDA (0010) no ModelSim - parte 2.

6.6 Simulação da Memória

A primeira etapa para a simulação da memória do Neander começou vendo como são suas características de composição através da figura 6.12. Na sequencia foi visto como acontece a leitura (*rd*) e escrita (*wr*) nos endereços da memória. A próxima etapa foi à elaboração do vetor de testes da memória exibido na tabela 6.11 através dessas explicações. A ultima etapa foi a simulação da memória carregada com informações nos seus primeiros endereços no ModelSim configurando os valores das entradas de acordo com os do vetor de teste. Dessa maneira a figura 6.15 ilustra que o resultado da simulação foi validado com o do vetor de teste da memória. Uma observação que já foi mencionada, é que como a memória inicia o seu *clk* na sua subida, a transferência do conteúdo do endereço de memória lido para a saída, só é realizada no próximo ciclo de *clk*.

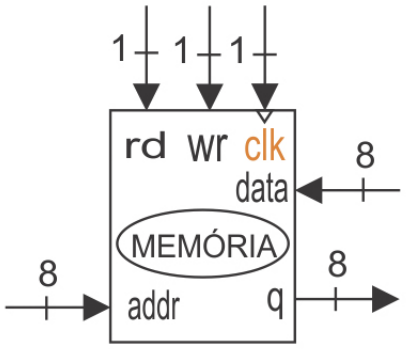

Figura 6.12: Memória do Neander.

A figura 6.13 demonstra a memória carregada com informações nos seus primeiros endereços e executando a leitura do endereço 00H, no qual para esta ação é ativado o sinal de controle *rd (1)* e desativado o *wr (0)*. Assim na saída q é repassado o valor 20H que é o conteúdo do endereço 00H.

A figura 6.14 demonstra a mesma memória da explicação anterior só que agora é executado à escrita do conteúdo FFH da entrada data no endereço 02H, no qual para esta ação é ativado o sinal de controle *wr* (1) e desativado o *rd* (0). Assim o conteúdo do endereço 02H na memória é alterado pelo valor escrito e a saída q mantém o valor do estado anterior.

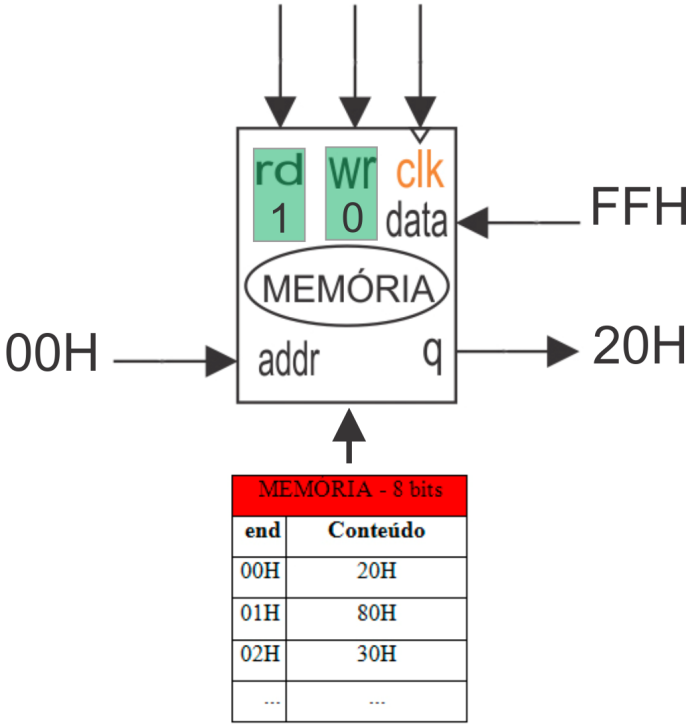

Figura 6.13: Leitura (*rd*) do end. 00H na memória.

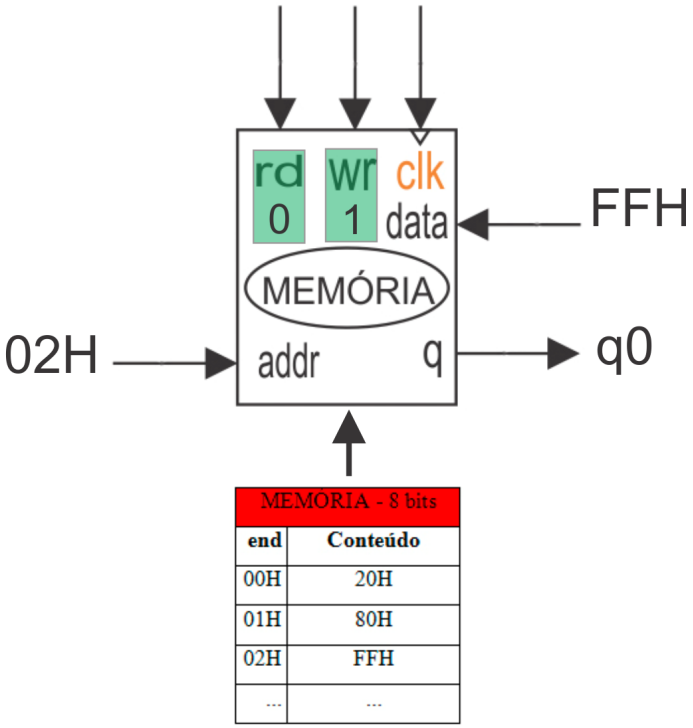

Figura 6.14: Escrita (*wr*) do end. 02H na memória

Entradas					Saídas
address	data	rd	wr	clk	q
00H	FFH	0	0	X	00H
00H	FFH	1	0	↑	20H
00H	FFH	1	0	↓	q0
01H	FFH	1	0	↑	80H
02H	FFH	1	0	↑	30H
02H	FFH	0	1	X	q0
02H	FFH	1	0	↑	FFH
02H	FFH	1	1	X	q0

Tabela 6.11: Vetor de teste da Memória do Neander.

Uma observação sobre a simulação da memória, é que a lógica da memória foi crida apartir da ferramenta do Quartus, Megawizard Plugin-in Manager. Assim algumas informações do vetor de testes foram preenchidas com o resultado que foi gerado na simulação da memória no ModelSim. Como é o caso quando o *rd* e o *wr* são ao mesmo tempo zero (0) e um (1).

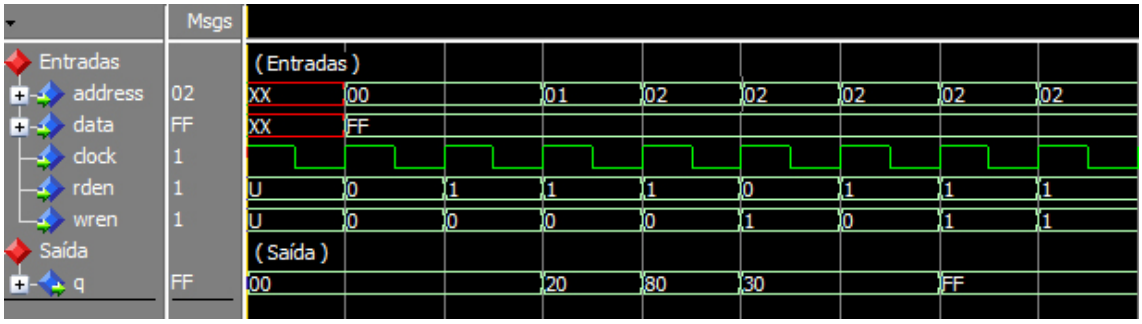

Figura 6.15: Simulação da Memória do Neander no ModelSim.

6.7 Simulação do Processador Neander

Para iniciar a simulação do processador do Neander a primeira etapa foi criação de um novo programa que utiliza as três classes do mesmo. Esse programa tem a função de comparar três valores e dizer qual deles é o maior. Esses três valores são colocados em três posições sequencias da área de operandos da memória. Feito a comparação, o valor maior é colocado no quarto endereço dessas sequencia. A tabela 6.12 mostra como foi montado esse programa. O próximo passo foi o carregamento do programa na memória junto com os valores. Depois é explicado como funciona a lógica do programa através de um fluxograma do programa. Por ultimo é demonstrado à simulação do processador Neander carregado com o programa de comparação de valores e nela é comprovado que o Neander executou o programa de acordo com o esperado.

Programa	
Mnemônico	Comentários
LDA 129	Copia o operando do end.129 para o AC
NOT	Inverte todos os bits do operando do AC
ADD 132	Soma o operando do end.132 com o operando do AC
ADD 128	Soma o operando do end.128 com o operando do AC
JN 30	Desvia para o end.30 se o cód. de cond. Negativo = 1
LDA 130	Copia o operando do end.130 para o AC
NOT	Inverte todos os bits do operando do AC
ADD 132	Soma o operando do end.132 com o operando do AC
ADD 128	Soma o operando do end.128 com o operando do AC
JN 24	Desvia para o end.24 se o cód. de cond. Negativo = 1
LDA 128	Copia o operando do end.128 para o AC
STA 131	Armazena o valor do AC no end.131
JMP 43	Desvia para o end.43
LDA 130	Copia o operando do end.130 para o AC
STA 131	Armazena o valor do AC no end.131
JMP 43	Desvia para o end.43
LDA 130	Copia o operando do end.130 para o AC
NOT	Inverte todos os bits do operando do AC
ADD 132	Soma o operando do end.132 com o operando do AC
ADD 129	Soma o operando do end.129 com o operando do AC
JN 24	Desvia para o end.24 se o cód. de cond. Negativo = 1
LDA 129	Copia o operando do end.129 para o AC
STA 131	Armazena o valor do AC no end.131
LDA 131	Copia o operando do end.131 para o AC
HLT	Termina execução

Tabela 6.12: Programa de comparação de três valores.

A tabela 6.13 mostra o programa carregado na memória em mnemônico. Nos três primeiros endereços da área dos operandos foram colocadas quatro variáveis e o numero um (1) no quinto endereço. As três primeiras variáveis representam os três valores que são comparados. A variável z representa o maior valor da comparação, ou seja, no final da comparação z pode ser igual a w, x ou y. O numero um no quinto endereço tem a função de junto com as instruções NOT e ADD converter um numero positivo em negativo, isto é, o conjunto dos três faz a lógica do complemento de dois. A lógica do programa pra fazer a comparação é pegar dois valores e aplicar a operação de subtração neles, assim é possível descobrir qual deles é o maior. Como o Neander não possui uma instrução de subtração foi feito essa configuração para que fosse possível fazer a comparação.

MEMÓRIA - 8 bits					
Área das Instruções				Área dos Operandos	
end	**Conteúdo**	**end**	**Conteúdo**	**end**	**Conteúdo**
0	LDA	24	LDA	128	w
1	129	25	130	129	x
2	NOT	26	STA	130	y
3	ADD	27	131	131	z
4	132	28	JMP	132	1
5	ADD	29	43	133	
6	128	30	LDA	134	
7	JN	31	130	135	
8	30	32	NOT	...	
9	LDA	33	ADD		
10	130	34	132		
11	NOT	35	ADD		
12	ADD	36	129		
13	132	37	JN		
14	ADD	38	24		
15	128	39	LDA		
16	JN	40	129		
17	24	41	STA		
18	LDA	42	131		
19	128	43	LDA		
20	STA	44	131		
21	131	45	HLT		
22	JMP				
23	43	...	...	255	

Tabela 6.13: Memória carregada com o programa de comparação de três valores.

A figura 6.16 exibe o fluxograma da lógica do programa de comparação. Nele é mostrada a lógica da subtração para descobrir qual valor é o maior.

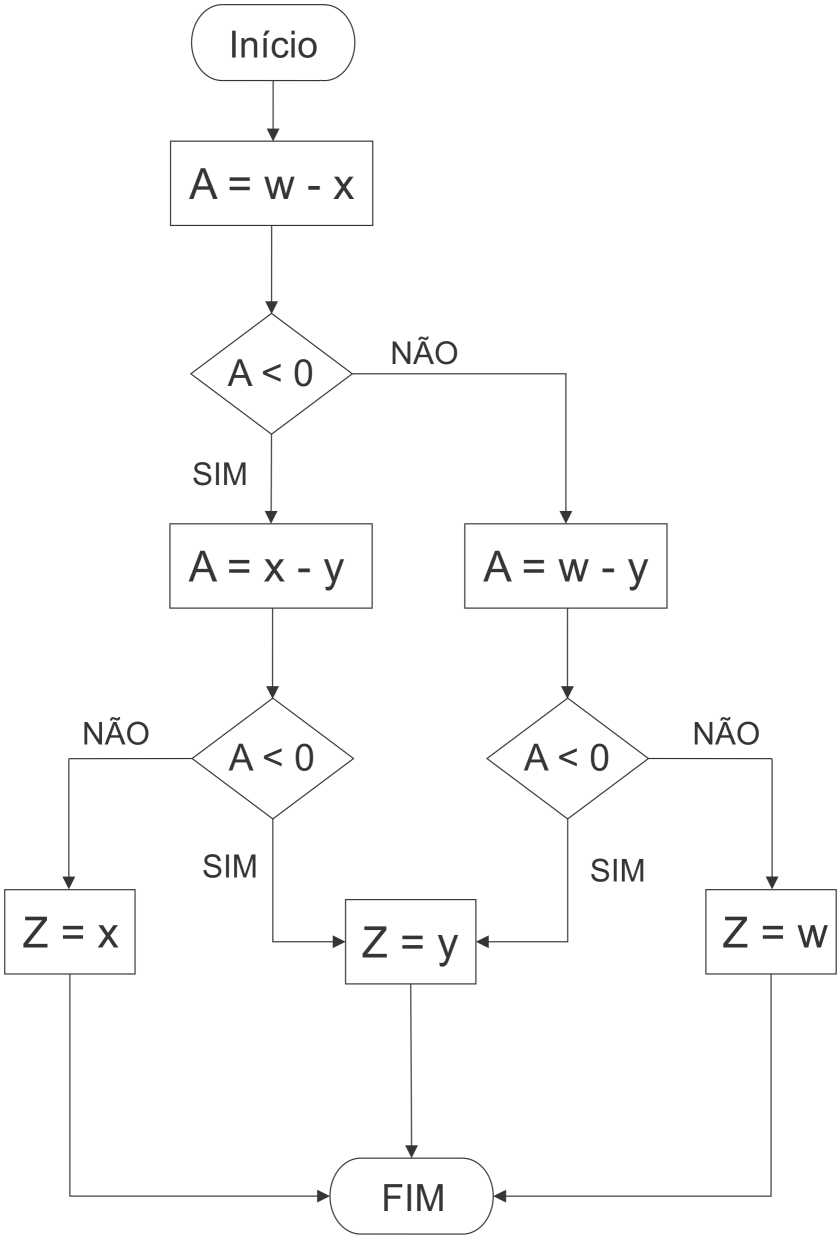

Figura 6.16: Fluxograma do programa de comparação de três valores.

A figura 6.17 ilustra a simulação do processador Neander carregado com o programa de comparação de três valores no ModelSim. Na simulação as variáveis foram configuradas com w=30, x=20 e y=10. Através da simulação se concluiu que a implementação do processador Neander em VHDL e o processamento do programa carregado no mesmo foram executados com sucesso. Pois o resultado final como esperado e repassado ao endereço de memória 131 (83H) foi o numero 30, no qual era o maior valor. Uma observação sobre a figura é que os valores de N e Z com o resultado 30 ficaram com os valores em zero (0), pois o valor 30 não é zero e nem negativo, mas no zoom da figura não é possível enxergar esses valores.

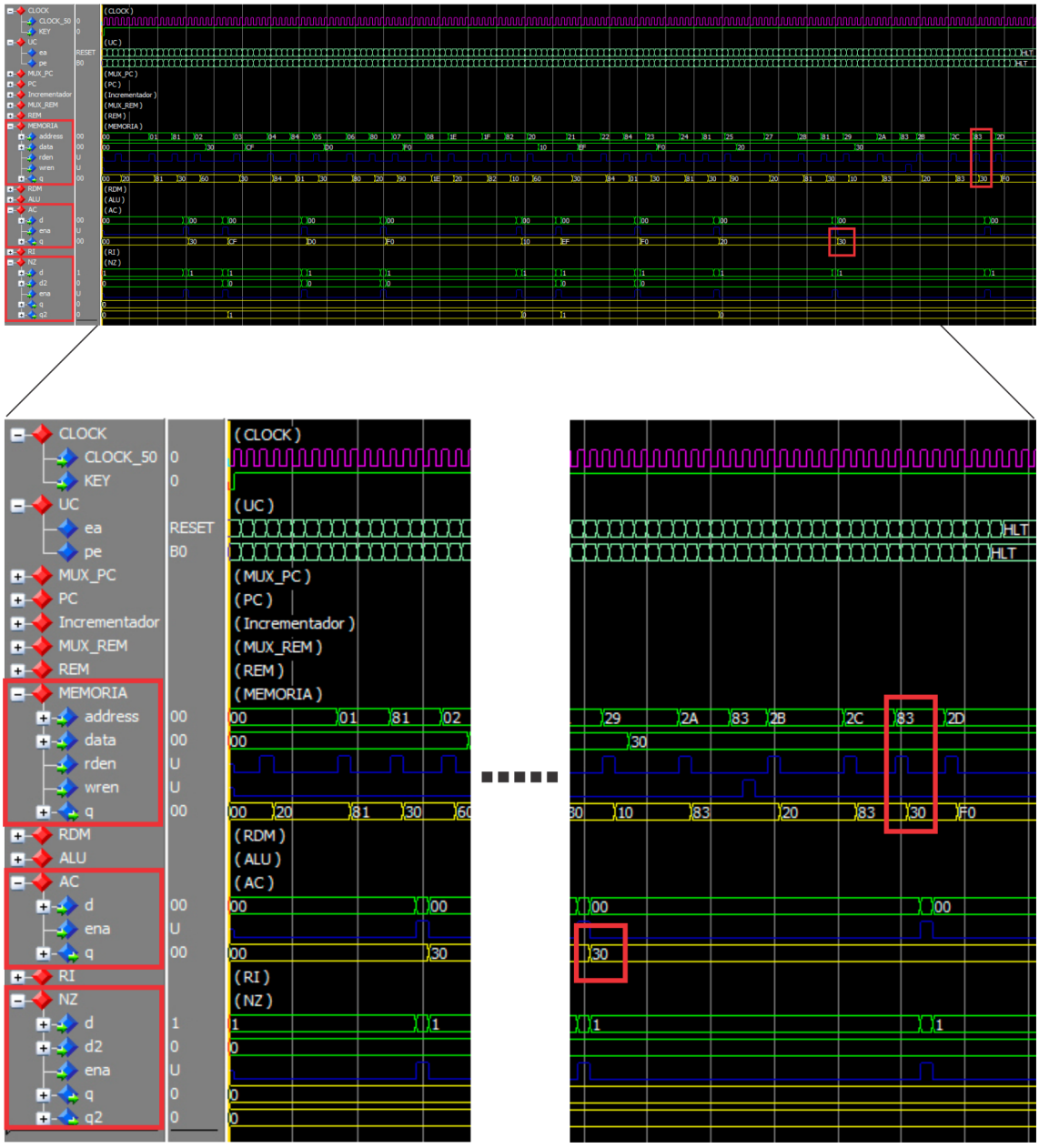

Figura 6.17: Neander processando o programa de comparação de três valores.

7 Conclusões e Trabalhos futuros

Este trabalho apresentou as características básicas de um processador nos quais são divididas em arquitetura e organização. Para demonstrar essas características foi implementado o processador didático Neander utilizando a linguagem de hardware VHDL. Ela foi escolhida porque permite descrever e simular circuitos digitais, no qual os componentes básicos dos processadores são formados. Além de ser uma tecnologia independente de fabricante, o que torna o código gerado portável e reutilizável em outras tecnologias do gênero. Também porque proporciona algumas vantagens na criação de projetos como a redução de tempo e dos custos do desenvolvimento, o aumento do nível de abstração na implementação do hardware e a grande facilidade nas atualizações dos projetos por permitir a simulação do hardware.

Durante a elaboração deste trabalho foram estudados as principais características de um processador, além da arquitetura e organização do processador didático Neander, os quais proporcionaram um conhecimento prático de como um processador funciona. Também se estudou os componentes básicos que compõem um processador, como registradores, unidades de execução e a UC. Esse estudo permitiu um aprofundamento no funcionamento de cada componente e nos conhecimentos de lógica digital combinacional e sequencial.

Os testes práticos descritos mostraram como é o processamento e a composição básica de um processador. Também foi demonstrado como um programa é executado e como as informações desse programa são interpretadas através dos componentes básicos que formam um processador.

Assim alunos e professores de disciplinas como arquitetura de computadores, microprocessadores ou outras disciplinas do gênero, apartir deste trabalho terão uma aprendizagem mais detalhada de como realmente ocorre o processamento das informações em um processador. Também apartir desse conhecimento, poderão criar vários programas para serem o processados no Neander.

Como trabalhos futuros, sugere-se a execução do processador Neander em um kit de desenvolvimento FPGA, no qual através dos displays de 7-segmentos e/ou o de cristal liquido seria configurado as informações de alguns registradores como o AC, PC, N, Z e RI, com o objetivo de mostrar as informações internas que estão sendo executadas durante o processamento de um programa. Seria tipo o simulador virtual da arquitetura do Neander apresentado na seção 2.18, só que esse seria um simulador físico.

Outra sugestão seria a implementação dos dois processadores mais complexos do conjunto de processadores didáticos comentados nesse trabalho, isto é, o Ramses e o Cesar que possuem uma arquitetura e organização mais complexa entre outras características que os diferenciam do Neander. O Ahmes não é recomendado porque possui uma unidade de execução ligeiramente mais complexa, entretanto é bastante semelhante ao que foi implementado no Neander. Também pode ser explorada a criação de um sistema de apoio as aulas relacionadas com o tema.

Referências Bibliográficas

MATOS, P. R. *Síntese de máquinas de estado (FSM)*. 2013. Disponível em: <http://wiki.$sj.ifsc.edu.br/wiki/images/f/f6/Sst-lab6_fsm.pdf$>. *Acesso em*: 28 out. 2014.

HARRIS, D. M. *Digital Design and Computer Architecture (2 ed.)*. [S.l.]: Wyman Street, Waltham: Morgan Kaufmann is an imprint of Elsevier, 2013.

MORENO, E. *Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)*. [S.l.]: São Paulo: BLESS Gráfica e Editora Ltd, 2003.

PEDRONI, V. *Eletrônica Digital Moderna e VHDL*. [S.l.]: Rio de Janeiro: Elsevier, 2010.

TOCCI, R. J. *Sistemas Digitais - Princípios e Aplicações (8 ed.)*. [S.l.]: São Paulo: Pearson Prentice Hall, 2003.

TORRES, G. *Como os processadores funcionam*. 2005. Disponível em: <http://www.clubedohardware.com.br/: http://www.clubedohardware.com.br/artigos/Como-os-Processadores-Funcionam/1145/7>. Acesso em: 18 ago. 2014.

WEBER, R. F. *Fundamentos de Arquitetura de Computadores (3 ed.)*. [S.l.]: Porto Alegre: Bookman: Instituto de Informática da UFRGS, 2008.

ZEFERINO, P. D. *Projetando um computador*. Disponível em: <http://wiki.sj.ifsc.edu.br/wiki/images/8/8e/Bip.pdf>. Acesso em: 27 out. 2014.

APÊNDICE A – Códigos fontes e configurações

Para ver como ficou configurado o código fonte em VHDL do processador didádito Neander e seus componentes, envie um e-mail para **marioallan.la@gmail.com**.