UNIVERSIDADE DE RIBEIRÃO PRETO
CENTRO DE CIÊNCIAS EXATAS, NATURAIS E TECNOLÓGICAS
ENGENHARIA DE COMPUTAÇÃO

FELIPE ALBANEZ CONTIN

SIMULADOR COMPUTADOR NEANDER WEB E MOBILE

Ribeirão Preto
2024

FELIPE ALBANEZ CONTIN

SIMULADOR COMPUTADOR NEANDER WEB E MOBILE

Monografia apresentada ao Centro de Ciências Exatas, Naturais e Tecnológicas da Universidade de Ribeirão Preto como parte dos requisitos para obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Otávio Marson Junior

Ribeirão Preto
2024

Ficha catalográfica preparada pelo Centro de Processamento
Técnico da Biblioteca Central da UNAERP

- Universidade de Ribeirão Preto -

CONTIN, FELIPE ALBANEZ, 2001-

C762s SIMULADOR COMPUTADOR NEANDER WEB E MOBILE / FELIPE ALBANEZ CONTIN. - Ribeirão Preto, 2024.
75 f. : il. color.

Orientador: Prof.º Dr.º Otávio Marson Junior.

Trabalho de Conclusão de Curso (Graduação) - Universidade de Ribeirão Preto, UNAERP, Engenharia de Computação, 2024.

1. Simulador. 2. Assembly. 3. Neander. 4. SAP. II. Título.

CDD 621.3

FOLHA DE APROVAÇÃO

833875 – Felipe Albanez Contin
Simulador computador Neander *web* e *mobile*

Monografia apresentada ao Centro de Ciências Exatas, Naturais e Tecnológicas da Universidade de Ribeirão Preto como parte dos requisitos para obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Otávio Marson Junior

Aprovada em: 09/12/2024

Banca Examinadora

Prof. Dr. Rodrigo de Oliveira Plotze
Instituição: Universidade de Ribeirão Preto – UNAERP

Prof. Me. Fernando Marco Perez Campos
Instituição: Universidade de Ribeirão Preto – UNAERP

Prof. Me. Geraldo Henrique Neto
Instituição: Universidade de Ribeirão Preto – UNAERP

DEDICATÓRIA

A Deus por sempre colocar as oportunidades e as pessoas certas na minha caminhada.

Ao meu pai que sempre me apoiou e me ensinou o respeito e o dever acima de tudo, e trouxe muita sabedoria para minha vida.

À minha mãe que sempre cuidou de mim e proporcionou muito amor e carinho.

AGRADECIMENTOS

Ao professor Otávio por proporcionar um aprendizado fantástico e cativante no decorrer do curso, profissional de extrema competência que traz muita riqueza ao ensino.

Aos meus colegas que sempre me ajudaram quando podiam e trouxeram bons momentos.

A Universidade de Ribeirão Preto por fornecer toda a infraestrutura e suporte durante todo o curso.

"Hard work. will power and dedication. For a man with these qualities, the sky is the limit."

Milkha Singh

RESUMO

ALBANEZ. F. C. Simulador computador Neander *web* e *mobile*. 75 p. 2024 Monografia (Graduação em Engenharia de Computação) – Centro de Ciências Exatas, Naturais e Tecnológicas, Universidade de Ribeirão Preto, Ribeirão Preto-SP, 2024.

A prática de conceitos teóricos sobre arquitetura de computadores em conjunto com um simulador didático potencializa a compreensão do estudante. O computador Neander é uma pseudo-máquina criada pelos Professores Raul Weber e Taisy Weber com a proposta de proporcionar uma experiência prática para seus estudantes a partir de um simulador educacional. A realidade atual dessas ferramentas são que se encontram desatualizadas e obsoletas, logo a desencorajar novos estudantes dos quais podem julgar os *softwares* como ultrapassados, a fim de contribuir na solução deste panorama, este trabalho propôs o desenvolvimento de um novo simulador de computador Neander, onde conta com os recursos convencionais do simulador além de novos como diagramação em tempo de execução. Com o intuito de promover a disponibilidade do *software*, o desenvolvimento está direcionado para navegadores *web* e dispositivos *mobile*. A definição da arquitetura Neander é fundamentada pela arquitetura *Simple as Possible* (SAP), proposta por Albert Paul Malvino e Jerald A. Brown na obra *Digital Computer Electronics*. De forma mais específica, as versões enfatizadas são SAP-1 e SAP-2, as quais, neste estudo, são amplamente comparadas com a finalidade de ratificar o equilíbrio adequado do Neander entre simplicidade e complexidade para um modelo prático didático. As 11 instruções Neander são demonstradas em contexto de arquitetura para o entendimento além do *assembly*. São apresentadas as demais máquinas baseadas no Neander, Ahmes, Ramses e Cesar, igualmente desenvolvidas pelos professores Weber, como propostas alternativas para o estudo a fim de compará-las com o Neander. A temporização dos sinais de controle do Neander é apresentada a fim de orientar um desenvolvimento pulso a pulso da simulação. As tecnologias de desenvolvimento são definidas e discutidas a fim de atingir um resultado profissional e satisfatório. A diagramação em tempo de execução da arquitetura é detalhada, com o intuito de ser um recurso inovador para o entendimento da máquina em funcionamento. Um módulo de *clock* customizável é apresentado como mecanismo para controle de velocidade de execução da máquina. A compatibilidade de importação e exportação de arquivos entre o simulador desenvolvido e o simulador Neander original é analisada em detalhes. Os pontos principais da lógica de programação que constituem a simulação de comportamento da máquina hipotética são minuciosamente descritos. A Pesquisa de validação, embasada na escala Likert, realizada com alunos do curso de Engenharia da Computação da Universidade de Ribeirão Preto – UNAERP é discutida, pois é um dos fatores chaves que demonstram a eficácia do simulador vista pelos estudantes. As telas desenvolvidas são apresentadas e tem seu funcionamento explicado, além disso se discute fatores de responsividade e imersão. O objetivo do trabalho foi satisfatoriamente atingido, visto que o simulador desenvolvido contempla todas as funcionalidades já existentes no original e faz se o complemento de novos recursos marcantes, bem como os resultados positivos da pesquisa de validação realizada com os estudantes.

Palavras chave: Simulador, Assembly, Neander, SAP.

ABSTRACT

ALBANEZ, F. C. Simulador computador Neander *web* e *mobile*. 75 p. 2024 Monografia (Graduação em Engenharia de Computação) – Centro de Ciências Exatas, Naturais e Tecnológicas, Universidade de Ribeirão Preto, Ribeirão Preto-SP, 2024.

The practice of theoretical concepts about computer architecture in conjunction with a educational simulator enhances the student's understanding. The Neander computer is a pseudo-machine created by Professors Raul Weber and Taisy Weber with the aim of providing a practical experience for their students using an educational simulator. The current reality of these tools is that they are outdated and obsolete, thus discouraging new students who may judge the software as outdated. In order to contribute to solving this situation, this work proposed the development of a new Neander computer simulator, which has the conventional features of the simulator in addition to new ones such as runtime diagramming. In order to promote the availability of the software, development is aimed at web browsers and mobile devices. The definition of the Neander architecture is based on the simple as possible (SAP) architecture, proposed by Albert Paul Malvino and Jerald A. Brown in the work Digital Computer Electronics. More specifically, the versions emphasized are SAP-1 and SAP-2, which, in this study, are extensively compared with the purpose of ratifying Neander's appropriate balance between simplicity and complexity for a practical didactic model. The 11 Neander instructions are demonstrated in architectural context for understanding beyond assembly. The other machines based on Neander, Ahmes, Ramses and Cesar, also developed by Professors Weber, are presented as alternative proposals for study in order to compare them with Neander. The timing of Neander control signals is presented in order to guide a pulse-by-pulse development of the simulation. Development technologies are defined and discussed in order to achieve a professional and satisfactory result. The runtime diagram of the architecture is detailed, with the aim of being an innovative resource for understanding the machine in operation. A customizable clock module is presented as a mechanism for controlling the machine's execution speed. The compatibility of importing and exporting files between the developed simulator and the original Neander simulator is analyzed in detail. The main points of the programming logic that constitute the simulation of the hypothetical machine's behavior are described in detail. The validation research, based on the Likert scale, carried out with students of the Computer Engineering course at the Universidade de Ribeirão Preto - UNAERP is discussed, since it is one of the key factors that demonstrate the effectiveness of the simulator as seen by the students. The developed screens are presented and their operation is explained, in addition to discussing responsiveness and immersion factors. The objective of the work was satisfactorily achieved, since the developed simulator includes all the functionalities already existing in the original and is complemented by new striking resources, as well as the positive results of the validation research carried out with the students.

Keywords: Simulator, Assembly, Neander, SAP.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

ACC	*Accumulator*
ALU	*Arithmetic logic unit*
CISC	*Complex instruction set computer*
CLR	*Clear*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
HLT	*Halt*
HMR	*Hot Module Replacement*
HTML	*HyperText Markup Language*
IR	*Instruction register*
ISA	*Instruction Set Architecture*
JB	*Jump borrow*
JC	*Jump carry*
JMP	*Jump*
JN	*Jump negative*
JNB	*Jump not borrow*
JNC	*Jump not carry*
JNV	*Jump not overflow*
JNZ	*Jump not zero*
JP	*Jump positive*
JSR	*Jump subroutine*
JV	*Jump overflow*
JZ	*Jump zero*
LDA	*Load accumulator*
LDR	*Load register*
LED	*Light-emitting diode*
MAR	*Memory address register*
NEG	*Negate sign*
NOP	*No operation*
NPM	*Node Package Manager*
NPN	*Negative, Positive, Negative*
PC	*Program counter*
PNP	*Positive, Negative, Positive*
RAM	*Random access memory*
RISC	*Reduced instruction set computer*
ROL	*Rotate left*
ROM	*Read only memory*
ROR	*Rotate right*
SAP	*Simple as Possible*
SHL	*Shift left*
SHR	*Shift right*
SPA	*Single-page application*
STA	*Store Accumulator*
STR	*Store register*
SUB	*Subtraction*
TMP	*Temporary*
TTL	*Transistor-transistor logic*

SUMÁRIO

1 INTRODUÇÃO

Simulações são muito usadas na engenharia para analisar e predizer o comportamento de um sistema dada determinada situação, um método muito eficaz afim de validar um processo sem comprometer recursos reais, além disso simulações são muito bem aplicadas como ferramentas pedagógicas com o intuito de praticar e fixar os conceitos estudados.

Um estudante de engenharia da computação, ao se deparar pela primeira vez com a prática dos conceitos de organização de computadores, é comum que apresente alguma adversidade na abstração de operações nas quais um modelo de arquitetura de computador realiza a fim de chegar a um resultado. De modo a facilitar a compreensão, foi criado o simulador Neander, que tem como inspiração a arquitetura *Simple as Possible* (SAP), uma máquina extremamente simples, mas que por definição é um computador, por meio deste simulador é possível obter uma demonstração de execução das operações de um programa, desta forma é oferecida uma percepção mais objetiva de como uma máquina teórica computa dados e instruções.

O estudante, por sua vez, tem a oportunidade de praticar ao ser capaz de escrever simples programas, em formato *assembly*, para testar a arquitetura, bem como validar seu entendimento. Isto faz com que o simulador Neander seja, portanto, uma ferramenta didática com alto potencial educativo. Entretanto, os simuladores disponíveis atualmente estão obsoletos e desatualizados, consequentemente isso pode desencorajar o uso, seja por uma interface ultrapassada ou o risco de baixar um *software* não homologado. Além disso, pode se afirmar que:

> Os simuladores atualmente disponíveis para ensino no Brasil (o que muitas vezes significa serem sistemas gratuitos), apresentam uma interface de usuário pouco elaborada e com poucos recursos operacionais, e mesmo considerando o uso de arquiteturas mais simples, é usual por parte dos alunos uma certa dificuldade de trabalhar com eles (SILVA; BORGES, 2006, p. 1).

Com a finalidade de contornar esses incômodos e trazer uma ferramenta com o melhor que um simulador Neander tem a oferecer, este trabalho propõe o desenvolvimento de um *software* de tecnologia *webview*, tendo disponibilidade para navegadores ou dispositivos *mobiles*, desta forma o usuário não precisa instalar nenhum *software* desconhecido em seu computador, basta acessar um *link online* para começar a usar a aplicação, e para aqueles que necessitam de portabilidade o simulador pode ser instalado em um dispositivo *mobile* e ser usado sem a exigência de estar conectado à internet.

Com o intuito de facilitar a assimilação do fluxo de dados, este *software* propõe uma novidade em relação aos simuladores já existentes, apresentar uma diagramação da estrutura de módulos em tempo real, desta maneira é possível acompanhar o fluxo de dados e endereços conforme a movimentação dos mesmos pelos registradores, memória, *Arithmetic logic unit* (ALU) e unidade de controle, de tal modo a gerar uma apresentação gráfica. Logo o estudante tem a possibilidade de assimilar tais conceitos de forma visual e interativa.

1.1 OBJETIVO GERAL

Desenvolver um simulador didático de computador do tipo Neander, *Simple as Possible* (SAP).

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

- Estudar e analisar a arquitetura de um computador SAP.
- Estudar e analisar o simulador Neander e suas variantes.
- Desenvolver aplicação para *browser* e *mobile* do simulador Neander.

1.3 ESTRUTURA DA MONOGRAFIA

Esta monografia está dividida em cinco capítulos, sendo que no primeiro capítulo há Introdução, Os Objetivos Geral e Específicos e a Estrutura da Monografia.

No segundo capítulo há a revisão de literatura que contempla o estudo e a fundamentação dos conceitos das arquiteturas SAP e Neander, o conjunto de instruções computacionais que compõe a programação Neander, as principais máquinas baseadas no Neander junto a suas características, e por fim, a justificativa de escolha da máquina Neander como ferramenta pedagógica ideal para os estudantes de arquitetura de computadores.

No terceiro capítulo encontra-se os principais materiais e métodos empregados na realização do trabalho.

No quarto capítulo apresentam-se os resultados e a discussão.

No quinto capítulo tem-se a conclusão do estudo.

E, por último, são descritas as referências utilizadas para a elaboração desta monografia.

2 REVISÃO DE LITERATURA

Neste capítulo será apresentada a revisão de literatura que contempla a arquitetura de um computador SAP, a arquitetura da máquina hipotética Neander e suas derivações, os conjuntos de instruções para operação das máquinas hipotéticas, e as vantagens dos simuladores como ferramentas pedagógicas.

2.1 ARQUITETURA SAP

Segundo Malvino e Brown (1993), a proposta principal de uma arquitetura SAP é demonstrar para um iniciante as ideias cruciais das operações de um computador sem criar complicações com detalhes desnecessários. Existem três gerações diferentes de SAP, sendo denominadas como, SAP-1, SAP-2 e SAP-3.

2.1.1 Gerações SAP

A primeira geração, SAP-1, é definida como um computador, pois é capaz de armazenar dados e um programa em sua memória, em seguida realiza as instruções de programa e as operações correspondentes, sem a necessidade de uma interferência humana (MALVINO; BROWN, 1993).

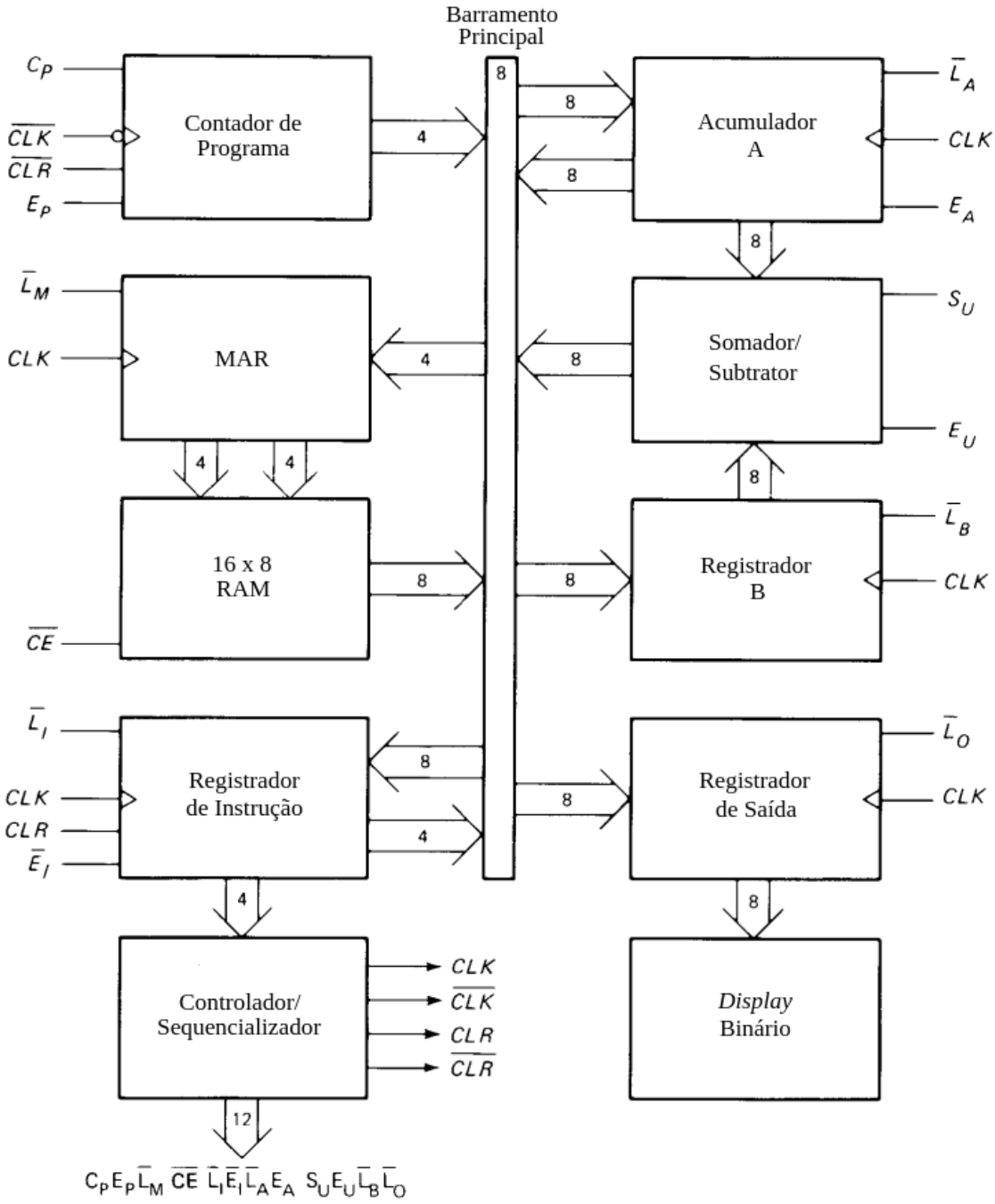
Com o SAP-1 é possível escrever diversos programas, porém existe uma limitação que impede que algoritmos mais robustos e com lógica de repetição sejam definidos, por conta desta dificuldade o SAP-2 foi projetado para incluir instruções de *jump*, com esse tipo de comando é possível pular partes do programa ou repeti-las indefinidamente, isso traz muito mais flexibilidade e poder de computação para a arquitetura.

Já o SAP-3 é uma versão que inclui todas as instruções do SAP-2, mas dessa vez com mais registradores e novas instruções. Nesse trabalho será abordado apenas o conteúdo de SAP-1 e SAP-2, já que o simulador Neander, geralmente, é uma abstração que está entre essas duas arquiteturas, pois o SAP-1 se mostra incompleto na necessidade de *flags* para condicionamento e operações de *jump* para iteração, e o SAP-2 exagera no número extra de registradores e instruções para servir de referência como modelo didático ideal.

2.1.2 SAP-1

A Figura 1, demostra em detalhes a arquitetura do SAP-1:

Figura 1 – Arquitetura SAP-1



Fonte: Adaptado de MALVINO; BROWN (1993, p. 141).

Todos os registradores que têm suas saídas conectadas com o Barramento Principal, são de 3 estados, nível lógico baixo, nível lógico alto e alta impedância (MALVINO; BROWN, 1993). Desta forma, é possível gerenciar e controlar a transferência de dados entre cada bloco.

Nesta arquitetura não existe um bloco referenciado como "Unidade de controle", no qual tem como objetivo centralizar todos os circuitos de controle em um único bloco ou um bloco apenas de entrada e saída de dados, normalmente conhecido como "*I/O*". O SAP-1 enfatiza o uso dos registradores de forma mais descentralizada para atender tais necessidades (MALVINO; BROWN, 1993).

2.1.2.1 Contador de programa

Todo programa é armazenado no começo da memória, com a primeira instrução no endereço binário de 0000, a segunda em 0001, e assim por diante de forma consecutiva. O papel do contador de programa é dizer em qual endereço se encontra a instrução atual do programa, assim como avançar para a próxima instrução. Quando o computador é iniciado o contador é sempre resetado para a primeira instrução (MALVINO; BROWN, 1993).

2.1.2.2 MAR

O bloco MAR significa *memory address register*, ou registrador de endereço de memória. O objetivo deste bloco é receber 4 *bits* de endereço e enviá-los para a RAM para que uma operação de leitura seja feita (MALVINO; BROWN, 1993).

2.1.2.3 RAM

A RAM, do SAP-1 é definida como 16 x 8 estática TTL, o que significa dizer 16 endereços para palavras de 8 *bits*, estática TTL pois seu arranjo de memória é constituído de transistores NPN e PNP que simulam circuitos de *flip-flop*.

Durante a execução, a RAM recebe 4 *bits* de endereço, enviados por MAR, uma operação de leitura é feita e a saída, por fim, é disponibilizada no barramento principal (MALVINO; BROWN, 1993).

Uma observação é que a RAM deve ser programada com as instruções de programa e palavras de dados antes do computador ligar, por causa da natureza volátil da memória, portanto os dados são perdidos na ausência de alimentação elétrica.

2.1.2.4 Registrador de instrução

O registrador de instrução faz parte da unidade de controle. É responsável por ler a instrução no barramento, disponibilizada pela RAM, e separá-la pela metade em duas partes, 4 *bits* cada, uma parte é enviada para uma saída de 3 estados conectada ao barramento, e outra para uma saída de 2 estados conectada ao bloco de Controlador/Sequencializador (MALVINO; BROWN, 1993).

2.1.2.5 Controlador/Sequencializador

No momento que o computador é ligado ou resetado, este bloco envia 2 sinais de *clear* (CLR) um para o contador de programa para resetá-lo para 0000, e outro para o registrador de instrução para remover qualquer instrução anterior.

Um sinal de *clock* é enviado para todos os registradores com o intuito de sincronizar as operações e garantir que todas as transferências ocorram juntas na borda de subida do *clock*.

Os 12 *bits* de saída formam o barramento de controle, cada *bit* representa uma *flag* para comunicar os outros blocos do que deve ser feito (MALVINO; BROWN, 1993).

2.1.2.6 Acumulador

O acumulador é um registrador que armazena respostas intermediárias durante a execução do programa, uma saída de 2 estados é conectada ao Somador/Subtrator, outra saída de 3 estados é conectada ao Barramento Principal que pode ser habilitada para sua informação ser disponibilizada para outras partes (MALVINO; BROWN, 1993).

2.1.2.7 Somador e Subtrator

Este bloco contém um circuito combinatório de Somador e Subtrator por complemento de 2, isto significa que independe de um *clock* para funcionar, sua saída é atualizada logo após a mudança das entradas, quando S_U está em nível lógico baixo realiza operação de soma, caso

contrário realiza operação de soma com complemento de 2, que resulta em uma operação de subtração (MALVINO; BROWN, 1993).

Sendo seus 2 números de entradas A e B, fornecidos pelo acumulador, como A, e pelo registrador B, como B.

Possui uma saída de 3 estados que quando habilitada disponibiliza seu resultado para o Barramento Principal.

2.1.2.8 Registrador B

Tem o propósito de servir como o registrador secundário para as operações aritméticas. Pode ser alimentado pelo barramento, e sempre envia seu valor para o Somador/Subtrator, no qual representa o número B para ser usado na operação de soma ou subtração (MALVINO; BROWN, 1993).

2.1.2.9 Registrador de saída

Este registrador tem o objetivo de servir como a saída final da execução, quando o acumulador contiver o resultado final, seu valor pode ser transferido para o registrador de saída (MALVINO; BROWN, 1993).

2.1.2.10 Display binário

Possui 8 *LEDs* em sequência que são alimentados pelo valor do registrador de saída, desta forma podemos ver a saída final da operação (MALVINO; BROWN, 1993).

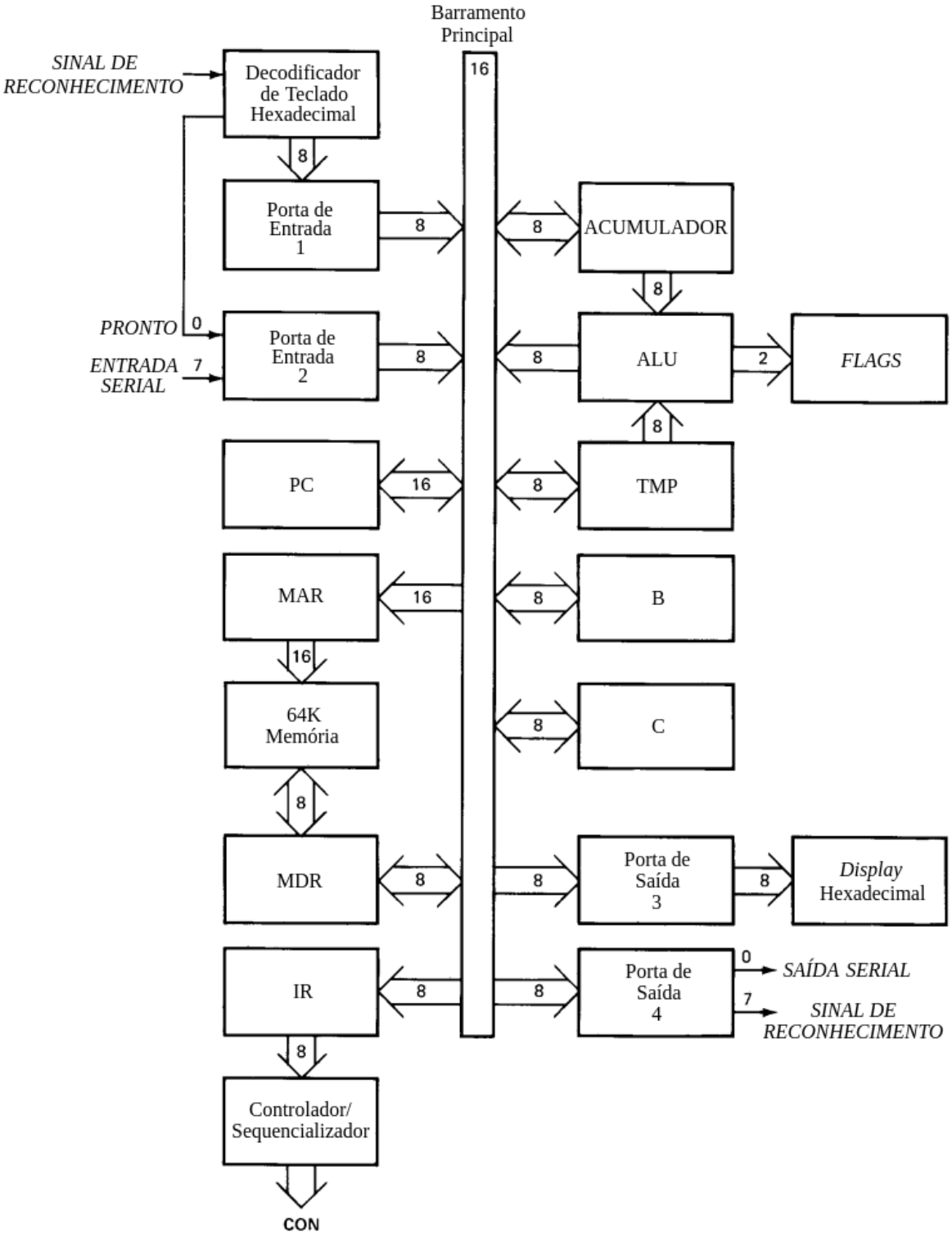
2.1.3 SAP-2

Por definição, o SAP-1 é considerado um computador por ser capaz de executar uma série de operações sem a intervenção humana, porém de forma muito limitada, a falta de instruções de *jump*, condicionamento, e escrita na memória o torna muito primitivo para solucionar problemas mais complexos.

O SAP-2 é a evolução direta do SAP-1, por proporcionar um conjunto de memória que totaliza 64K endereços e 16 *bits* de palavra, o Barramento Principal expandido para 16 *bits*, comparado com o SAP-1 é uma diferença colossal, mais registradores auxiliares, e novas

instruções, incluindo instruções de escrita na memória e de *jump* para laços de repetição ou pular partes do programa quando necessário (MALVINO; BROWN, 1993). A Figura 2, demostra em detalhes a arquitetura do SAP-2.

Figura 2 – Arquitetura SAP-2



Fonte: Adaptado de MALVINO; BROWN (1993, p. 141).

2.1.3.1 Portas de entrada

o SAP-2 possui 2 portas de entradas, nomeadas de Porta de Entrada 1 e Porta de Entrada 2, um teclado hexadecimal é conectado à porta 1, enquanto a porta 2 é usada para validar quando os dados da porta 1 estão prontos para leitura e receber uma entrada *serial* (MALVINO; BROWN, 1993).

2.1.3.2 Contador de programa

Desta vez o contador de programa possui 16 *bits*, em notação hexadecimal vai de 0000(H) até FFFF(H), ou em decimal de 0 até 65535, tem o mesmo objetivo que no SAP-1, armazenar o endereço da instrução atual do programa (MALVINO; BROWN, 1993).

2.1.3.3 Memória

A memória do SAP-2 é dividida em duas memórias, a primeira memória de 2K ROM contém um programa responsável por inicializar o computador ao ser ligado, e um interpretador para as entradas do teclado, a segunda memória de 62K RAM é usada para armazenar as instruções do programa, e variáveis ou constantes acessadas durante a execução do programa (MALVINO; BROWN, 1993).

2.1.3.4 MAR

Como no SAP-1, existe um registrador responsável para definir o endereçamento da memória para uma operação de leitura ou escrita, dessa vez com um barramento de 16 *bits* (MALVINO; BROWN, 1993).

2.1.3.5 MDR

Memory Data Register (MDR), ou registrador de dados da memória, é um registrador bidirecional responsável pela troca de dados entre a memória e o Barramento Principal, quando a memória realiza uma operação de leitura o MDR envia os dados da memória para o barramento, em operação de escrita o MDR envia os dados do barramento para a memória,

possui um barramento de 8 *bits* para acompanhar o tamanho da palavra da memória (MALVINO; BROWN, 1993).

2.1.3.6 Registrador de instrução

O SAP-2 possui 42 instruções, desta forma foi necessário aumentar seu tamanho de 4 *bits* para 8 *bits* em relação ao SAP-1. Funciona da forma semelhante ao SAP-1, faz a leitura da instrução no barramento e envia para Controlador/Sequencializador, porém desta vez a instrução não é dividida pela metade (MALVINO; BROWN, 1993).

2.1.3.7 Controlador/Sequencializador

Segue a mesma ideia do SAP-1, porém dessa vez o barramento de controle possui uma palavra maior, tem o objetivo de instruir os registradores como devem reagir na próxima borda de subida do *clock* (MALVINO; BROWN, 1993).

2.1.3.8 Acumulador

Segue o mesmo princípio do SAP-1, faz o papel de um registrador intermediário para as operações aritméticas executadas pela ALU (MALVINO; BROWN, 1993).

2.1.3.9 ALU

Diferente do SAP-1 no qual este bloco é identificado como Somador/Subtrator, no SAP-2 ele foi generalizado para *Arithmetic logic unit* (ALU), ou Unidade lógica e aritmética, possui o mesmo objetivo de realizar operações matemáticas, mas desta vez com a adição de duas novas *flags*, *sign flag* ou *negative flag*, ligada quando o acumulador contém um número negativo e *zero flag*, ligada quando o acumulador está em 0 (MALVINO; BROWN, 1993).

2.1.3.10 Registradores TMP, B e C

No lugar do registrador B no SAP-1, o registrador TMP, ou registrador temporário, é usado no SAP-2, possui o objetivo de fornecer a ALU o segundo número utilizado na

operação. Os registradores B e C foram adicionados como suportes para fornecer flexibilidade em operações de transferência de dados (MALVINO; BROWN, 1993).

2.1.3.11 Portas de saída

SAP-2 possui 2 portas de saída, a primeira conectada a um *display* hexadecimal para que os dados processados possam ser visualizados, e a segunda porta é usada para a saída *serial* dos dados e para enviar um sinal de confirmação ao codificador hexadecimal do *display* (MALVINO; BROWN, 1993).

2.2 SIMULADOR NEANDER

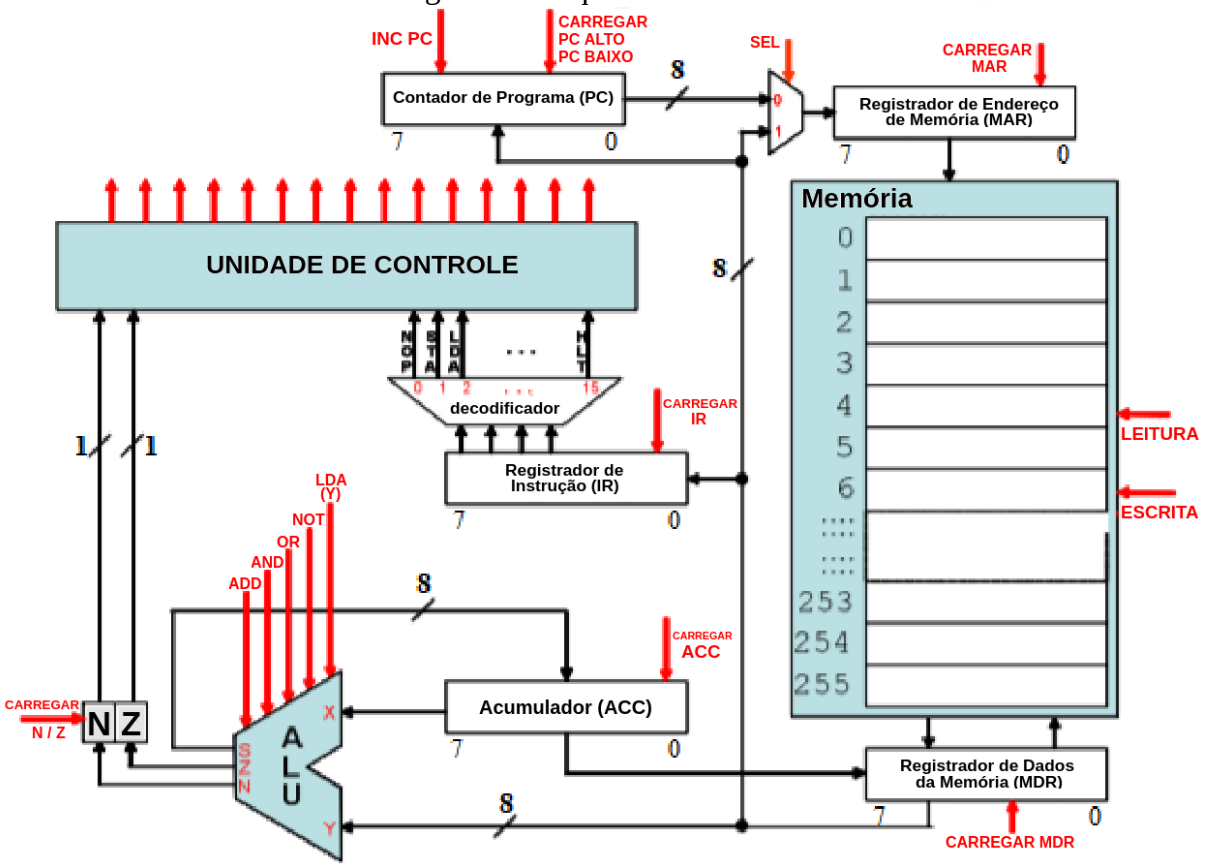
O Neander foi desenvolvido pelos professores Raul Weber e Taisy Weber, a definição oficial de suas características se encontram na obra Fundamentos de arquitetura de computadores, de autoria por Raul Weber. O computador Neander é constituído de um barramento de 8 *bits*, uma memória RAM de 256x8, um registrador acumulador, um registrador de contador de programa e um barramento de estado com 2 condições, *sign flag* e *zero flag* (WEBER, 2012).

Assim como os computadores SAP, na simulação Neander também é feito o uso de instruções *assembly* para a rotina de execução do programa, estas instruções são comandos definidos por palavras ou siglas que possuem seu respectivo valor binário, isto permite a unidade de controle reconhecer a instrução e fazer o tratamento necessário no barramento de controle para cada comando. O computador Neander normalmente possui 11 instruções que são detalhadas abaixo no capítulo 2.2.2.

2.2.1 Arquitetura Neander

A arquitetura original do Neander possui um barramento de palavra de 8 *bits* (SCHNEIDER, 2004), conhecendo a arquitetura do SAP-1 e SAP-2 é possível identificarmos os componentes necessários para o seu funcionamento teórico. Por exemplo, sabemos que o computador Neander tem instruções de leitura e escrita na memória, assim deduzimos que sua arquitetura deve conter registradores de MAR e MDR. A Figura 3, ilustra sua arquitetura.

Figura 3 – Arquitetura Neander



Fonte: Adaptado de CAGLIARI; BUTZEN; BRUM (2020, p. 1).

O simulador Neander possui uma memória de 256x8, diferente do SAP-1 e SAP-2 que possuem uma memória de 16x8 e 64Kx16 respectivamente, um decodificador para tradução das instruções, um registrador acumulador, uma ALU para as operações aritméticas, um contador de programa (PC), registradores MAR e MDR para operações de leitura e escrita na memória, um registrador de instrução e um barramento de estado da ULA contendo os sinais de negativo e zero (WEBER, 2012). Todos esses componentes operam juntos a partir do barramento de controle determinado pela unidade de controle que garante a lógica de funcionamento correta.

Dada a arquitetura Neander, é pertinente concluir que:

> NEANDER é um computador muito simples, desenvolvido apenas para fins didáticos. Processadores modernos são muito mais complexos que o NEANDER. Entretanto, mesmo processadores utilizados nas mais sofisticadas estações de trabalho são baseados nos conceitos elementares que você aprendeu com NEANDER (WEBER, 2012, p. 54).

2.2.2 Instruções Neander

Segundo Tanenbaum e Austin (2013, p. 542) "Todo computador tem uma ISA (*Instruction Set Architecture* – arquitetura do conjunto de instrução), que é um conjunto de registradores, instruções e outras características visíveis para seus programadores de baixo nível". Portanto este conjunto de instruções, também denominado de linguagem de máquina ou *assembly*, é o que possibilita a programação da rotina de operações na máquina.

Para realizar a programação do computador Neander são definidas 11 instruções, das quais 8 são de modo de endereçamento absoluto direto, onde o próprio endereço do operando é o endereço referenciado na memória, e 3 são de modo de endereçamento inerente, onde não é necessário a referência de nenhum endereço da memória ou registrador para a execução da operação (WEBER, 2012). Em comparação com o SAP-1 e SAP-2, esses possuem 5 e 42 de operações no total, respectivamente, isso expressa como o SAP-1 é muito simplório e primitivo para programas que exigem uma lógica com certa complexidade, e o SAP-2 se torna muito intimidante, com muitas instruções disponíveis, para ser usado como modelo didático para novos estudantes que estão em um processo de familiarização com a arquitetura do computador e a programação *assembly*.

O Quadro 1, descreve as 11 operações do computador Neander:

Quadro 1 – Instruções Neander

Binário	**Decimal**	**Hexadecimal**	**Instrução**	**Descrição**
0000 0000	0	00	NOP	Nenhuma operação
0001 0000	16	10	STA *end*	Armazena acumulador
0010 0000	32	20	LDA *end*	Carrega acumulador
0011 0000	48	30	ADD *end*	Soma
0100 0000	64	40	OR *end*	Operação lógica OR
0101 0000	80	50	AND *end*	Operação lógica AND
0110 0000	96	60	NOT	Inverte bits do acumulador
1000 0000	128	80	JMP *end*	Desvio incondicional
1001 0000	144	90	JN *end*	Desvio condicional *negative*
1010 0000	160	A0	JZ *end*	Desvio condicional *zero*
1111 0000	240	F0	HLT	Encerra execução

Fonte: Adaptado de SILVA; BORGES (2006, p. 3).

É plausível constatar que apenas com estas poucas operações já se tem uma flexibilidade computacional admissível para a engenharia de algoritmos com complexidades mais expressivas, porém em contrapartida simples programas ainda mantêm uma fácil leitura e compreensão do algoritmo, pois são constituídos de instruções descomplicadas de abstrair. Desta forma o computador Neander possui um bom equilíbrio entre conveniência e número de instruções, segue uma filosofia de arquitetura RISC de forma mais pragmática em comparação com o conjunto de instruções do SAP-2.

2.2.2.1 NOP

Instrução nula, ou ausência de instrução, por padrão todos os endereços na memória começam com o valor 0, isso significa que o programa começa com todas as instruções em NOP. Esta instrução não realiza nenhuma operação, apenas gasta o *clock* de sua leitura (WEBER, 2012).

2.2.2.2 STA

Instrução *store*, ou instrução de armazenar, recebe junto um operando de endereço no qual é carregado no registrador MAR para operação de escrita no endereço informado, os dados do registrador acumulador são carregados no registrador MDR, desta forma os dados do acumulador são salvos no endereço referenciado (WEBER, 2012).

2.2.2.3 LDA

Instrução *load*, ou instrução de carregar, semelhante ao STA, mas com o intuito de leitura, recebe um endereço no operando, os dados do endereço referenciado são carregados no acumulador (WEBER, 2012).

2.2.2.4 ADD

Instrução *add*, ou instrução de soma, recebe também um endereço no operando, carrega os dados do endereço na ALU e realiza uma soma por complemento de 2 com o valor do acumulador, e o resultado é salvo no acumulador (WEBER, 2012).

2.2.2.5 OR

Instrução *or*, ou instrução "ou", possui endereço no operando, carrega os dados do endereço na ALU e realiza uma operação de *or* com o acumulador, no qual os *bits* correspondentes se mantêm 1 caso pelo menos algum seja 1, caso contrário recebem 0, este resultado é salvo no acumulador (WEBER, 2012).

2.2.2.6 AND

Instrução *and*, ou instrução "e", muito semelhante a instrução *or*, a diferença está que em vez da lógica booleana *or*, é usado *and*, logo os *bits* correspondentes se mantêm 1 caso ambos sejam 1, caso contrário recebem 0 (WEBER, 2012).

2.2.2.7 NOT

Instrução *not*, ou instrução de negação, esta instrução é de modo de endereçamento inerente, logo não realiza nenhuma operação na memória e não é necessário um endereço no operando, realiza uma operação de inversão nos *bits* do acumulador, portanto os *bits* 0 são transformados em 1, e os 1 para 0 (WEBER, 2012).

2.2.2.8 JMP

Instrução de *jump*, ou instrução de salto, está instrução manuseia o registrador PC, logo é possível voltar ou pular parte do programa em execução, muito usado para criar laços de repetição, o endereço referenciado no operando é carregado no registrador de contador de programa, portanto no próximo *clock* da máquina a próxima instrução não necessariamente será a instrução consecutiva a anterior (WEBER, 2012).

2.2.2.9 JN

Instrução de *jump negative*, ou instrução de salto negativo, isto significa um condicionamento na *negative flag*, logo semelhante a instrução de JMP um salto é realizado, mas desta vez, apenas se a *flag* de negativo estiver acionada, caso contrário a operação é ignorada e o registrador PC segue normalmente, muito útil para executar ou pular blocos de códigos apenas em determinadas situações (WEBER, 2012).

2.2.2.10 JZ

Instrução de *jump zero*, ou instrução de salto zero, semelhante a instrução de JN, porém com a *zero flag*, desta forma o salto só acontece se o valor do acumulador ser igual a 0 (WEBER, 2012).

2.2.2.11 HLT

Instrução *halt*, ou instrução de parada, esta instrução pausa a execução da máquina interrompendo o *clock*, geralmente usada para encerrar o programa (WEBER, 2012).

2.3 SIMULADOR AHMES

Como proposta alternativa para o simulador Neander, foi desenvolvido um novo computador pelo professor Raul Weber, o simulador Ahmes, igualmente com o intuito de ser uma máquina didática, mas desta vez desenhado com o foco em operações aritméticas mais complexas como multiplicação e divisão. Ahmes possui total compatibilidade com Neander em seu conjunto de instruções, além disso possui novas instruções e *flags* de estado na ALU para contribuir nas operações aritméticas (WEBER, 2012).

O computador Ahmes pode ser pensado como uma extensão ou um aperfeiçoamento do computador Neander, porém ainda está longe da complexidade proposta pela arquitetura e instruções do modelo SAP-2, por ser compatível com as instruções Neander, segue da mesma forma os dados representados por complemento de 2 e o modo de endereçamento de suas instruções em absoluto direto ou inerente.

2.3.1 Arquitetura Ahmes

Semelhante ao computador Neander, Ahmes possui uma palavra de barramento de 8 *bits*, um acumulador de 8 *bits* no qual a ALU usa como uma das entradas e como saída para as operações aritméticas, um contador de programa de 8 *bits* com o propósito de apontar a próxima instrução a ser executada, e sua ALU possui um registrador de estado com 5 condições, sendo elas *negative flag* (N), *zero flag* (Z), *carry out flag* (C), *borrow out flag* (B) e *overflow flag* (V).

Sua arquitetura é constituída de maneira muito similar ao Neander, logo o restante dos seus componentes como a memória e os outros registradores não mencionados são os mesmos

do Neander. A vantagem dessa arquitetura são os 3 novos estados da ULA que desempenham um papel fundamental para a programação de operações aritméticas mais avançadas como multiplicação e divisão.

Overflow flag quando ligada indica a representação de um estouro após uma instrução de soma ou subtração, isto significa que o resultado não pode ser representado apenas com uma combinação de 8 bits. *Carry out flag* quando ligada indica que houve a existência de um "vai-um" na operação de soma. *Borrow out flag* semelhante a *carry out*, indica que houve um "empresta-um" quando ligada, mas desta vez apenas para a operação de subtração. *Zero flag* e *negative flag* funcionam exatamente igual ao Neander (WEBER, 2012).

2.3.2 Instruções Ahmes

O conjunto de instruções de Ahmes possui no total 24 instruções, 13 instruções a mais comparadas com o Neander, sendo essas novas instruções para *jump* e deslocamento. O Quadro 2, descreve todas as operações de Ahmes:

Quadro 2 – Instruções Ahmes

Binário	**Decimal**	**Hexadecimal**	**Instrução**	**Descrição**
0000 0000	0	00	NOP	Nenhuma operação
0001 0000	16	10	STA *end*	Armazena acumulador na memória *store*
0010 0000	32	20	LDA *end*	Carrega acumulador da memória *load*
0011 0000	48	30	ADD *end*	Soma
0100 0000	64	40	OR *end*	“Ou” lógico
0101 0000	80	50	AND *end*	“E” lógico
0110 0000	96	60	NOT	Inverte acumulador
0111 0000	112	70	SUB *end*	Subtração
1000 0000	128	80	JMP *end*	Desvio incondicional *jump*
1001 0000	144	90	JN *end*	Desvio condicional *jump if negative*
1001 0100	148	94	JP *end*	Desvio condicional *jump if positive*
1001 1000	152	98	JV *end*	Desvio condicional *jump if overflow*
1001 1100	156	9C	JNV *end*	Desvio condicional *jump if not overflow*
1010 0000	160	A0	JZ *end*	Desvio condicional *jump if zero*
1010 0100	164	A4	JNZ *end*	Desvio condicional *jump if non-zero*
1011 0000	176	B0	JC *end*	Desvio condicional *jump if carry*

1011 0100	180	B4	JNC *end*	Desvio condicional *jump if not carry*
1011 1000	184	B8	JB *end*	Desvio condicional *jump if borrow*
1011 1100	188	BC	JNB *end*	Desvio condicional *jump if not borrow*
1110 0000	224	E0	SHR	Deslocamento para direita *shift right*
1110 0001	225	E1	SHL	Deslocamento para esquerda *shift left*
1110 0010	226	E2	ROR	Rotação para direita *rotate right*
1110 0011	227	E3	ROL	Rotação para esquerda *rotate left*
1111 0000	240	F0	HLT	Término de execução *halt*

Fonte: Adaptado de WEBER (2012, p. 56).

O Ahmes possui as mesmas instruções do Neander além de mais instruções de *jump* condicional e operações de descolamento. No total são 11 instruções dedicadas de *jump*, sendo a instrução JMP sem condicionamento e as demais instruções verificam o valor das *flags* de estado da ALU sendo 2 instruções para cada *flag*, uma para o estado acionado e outra na forma negada. As instruções de deslocamento e rotação tem o objetivo de facilitar operações de multiplicação e divisão, no qual cada *bit* é deslocado para o *bit* ao lado conforme a orientação do sentido de forma respectiva, nas operações de SHR e SHL os *bits* deslocados para fora são perdidos, já as operações de rotação ROR e ROL o deslocamento é feito de forma circular e nenhum *bit* é perdido (WEBER, 2012).

As demais instruções contidas no Neander já comentadas no capítulo 2.2 não serão explicadas novamente pois funcionam de forma idêntica no Ahmes.

2.3.2.1 SUB

Instrução de *subtraction*, ou instrução de subtração, equivalente a instrução de ADD com uma instrução de NOT no número a se tornar negativo somado com 1 para simular uma subtração por soma de complemento de 2, esta instrução serve como um atalho para realizar uma subtração, a fim de simplificar o processo de uma subtração por soma de complemento de 2. Portanto “no lugar de a – b realize-se a + not(b) + 1” (WEBER, 2012, p. 61).

2.3.2.2 JP

Instrução de *jump positive*, ou salto positivo, é a instrução oposta à instrução de JN, isto significa que o salto só acontece caso o valor do acumulador não seja negativo (WEBER, 2012).

2.3.2.3 JV

Instrução *jump overflow*, ou salto estouro, o salto só acontece se a *flag* de *overflow* estiver acionada, logo o valor resultado no acumulador deve ter sofrido *overflow* da última instrução que alterou o valor do acumulador para o salto acontecer (WEBER, 2012).

2.3.2.4 JNV

Instrução *jump not overflow*, ou salto sem estouro, é a instrução de condição oposta a JV, isto significa que o salto só acontece caso o resultado no acumulador não tenha sofrido um *overflow* (WEBER, 2012).

2.3.2.5 JNZ

Instrução *jump not zero*, ou salto não zero, é a instrução de condição oposta a JZ, o estado de *zero flag* deve estar desligado para o salto acontecer, qualquer valor diferente de zero no acumulador aprova o salto (WEBER, 2012).

2.3.2.6 JC

Instrução *jump carry*, ou salto vai-um, para este salto acontecer a operação de soma anterior deve ter ocorrido a presença de um "vai-um", que pode acontecer quando a soma de dois números, geralmente um positivo e outro negativo em complemento de 2, resulta em um número no qual o *bit* mais significativo sofreu um *carry out* ou "vai-um". Por exemplo na operação de soma entre 7 e 251 (-5 em complemento de 2), a *carry out flag* é acionada, pois aquele que seria o 9º *bit* está valendo 1 porém por ele não ser considerado na representação do número a soma resultou no número 2 (WEBER, 2012).

2.3.2.7 JNC

Instrução *jump not carry*, ou salto não vai-um, realiza um salto se a *carry out flag* não estiver acionada, é a condição oposta à JC, logo para o salto acontecer se tiver havido uma operação de soma e seu resultado ainda estiver no acumulador, esse valor não pode ter a presença de um "vai-um" em seu *bit* mais significativo (WEBER, 2012).

2.3.2.8 JB

Instrução *jump borrow*, ou salto empresta-um, este salto é condicionado pela *borrow out flag*. *Borrow* é o inverso de *carry*, portanto em uma operação de subtração com a instrução SUB o valor de *borrow* é resultado do oposto de *carry*, isto significa que sempre que *carry* for 1 *borrow* será 0 e vice e versa, entretanto, mesmo *carry* e *borrow* sendo um inverso ao outro, suas *flags* respectivas são independentes, logo a *borrow out flag* é reservada para a operação de SUB e a *carry out flag* é reservada para a operação de ADD. Portanto na instrução JB para o salto ocorrer a última operação de subtração deve ter acionado a *borrow out flag* e seu resultado ainda deve estar no acumulador, para assim manter a *flag* acionada e permitir o salto (WEBER, 2012).

2.3.2.9 JNB

Instrução *jump not borrow*, ou salto não empresta-um, é a instrução de condição oposta a JB, desta maneira o salto só acontece se a *borrow out flag* não estiver acionada, logo se tiver havido uma operação de subtração anteriormente e seu resultado ainda estiver no acumulador, o salto só será possível se a operação não tiver resultado em um valor que ocorreu o "empresta-um" (WEBER, 2012).

2.3.2.10 SHR

Instrução *shift right*, ou deslocamento direita, esta operação faz com que todos os *bits* do acumulador movam para o *bit* a direita. Por exemplo o número 240 (1111 0000) ao sofrer um deslocamento para direita seu valor passa para 120 (0111 1000), note que essa operação é o equivalente a dividir o número por 2, pois em qualquer sistema de numeração mover os dígitos para direita equivale a dividir pela base desse sistema (WEBER, 2012). A instrução SHR é muito poderosa para programar rotinas de operações aritméticas de divisão.

2.3.2.11 SHL

Instrução *shift left*, ou deslocamento esquerda, semelhante a operação SHR, porém no sentido oposto, os *bits* do acumulador são deslocados para a esquerda, isto significa que esta operação é o equivalente a multiplicar o número por 2. Extremamente útil para programar operações aritméticas de multiplicação (WEBER, 2012).

2.3.2.12 ROR

Instrução *rotate right*, ou rotacionar direita, funciona de forma semelhante a SHR, todos os *bits* são deslocados para a direita, entretanto na operação SHR o *bit* menos significativo é perdido, já na instrução ROR o *bit* menos significativo passa a ser o *bit* mais significativo (WEBER, 2012).

2.3.2.13 ROL

Instrução *rotate left*, ou rotacionar esquerda, segue a mesma ideia de ROR, porém nesta instrução os *bits* são deslocados para esquerda, isto significa, inclusive, que o *bit* mais significativo é deslocado para o *bit* menos significativo (WEBER, 2012).

2.4 SIMULADOR RAMSES

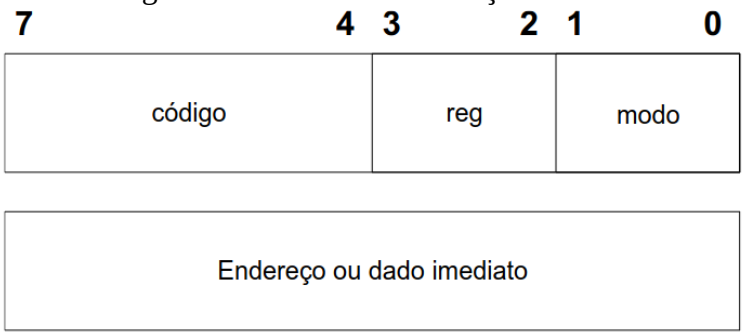
O computador Ramses faz parte das máquinas hipotéticas baseadas no computador Neander, também desenvolvido pelo professor Weber, o Ramses não prioriza tanto o foco em operações aritméticas como o Ahmes, porém possui suporte a uma execução mais dinâmica com registradores adicionais e novos modos de endereçamento.

Segundo Neto (2021, p. 17) “Assim como o Ahmes, Ramses é uma extensão do Neander, porém não compatível com Ahmes”. Desta forma a arquitetura hipotética de Ramses segue o mesmo princípio de Ahmes, novos componentes adicionados, porém toda a sua base de arquitetura se baseia no Neander. Ramses possui 4 novas instruções, 2 novos registradores de 8 *bits* para uso geral, um novo registrador exclusivo para endereçamento indexado, suporte aos modos de endereçamento indireto e indexado, e uma nova condição de estado *carry out flag* (WEBER, 2012).

Diferente do Neander e do Ahmes que só possuem no máximo 2 operandos em suas instruções, o primeiro como código da instrução e o segundo como um endereço da memória,

o Ramses, por sua vez, permite a escolha do registrador e do modo de endereçamento. A Figura 4, demonstra como é definido o formato de instrução no Ramses.

Figura 4 – Formato de instrução Ramses



Fonte: Adaptado de WEBER (2012, p. 191)

O Ramses reserva 2 *bits* para a identificação do registrador na instrução, seus valores são definidos no Quadro 3.

Quadro 3 – Código de identificação do Registrador no Ramses

Código	**Registrador**
00	A
01	B
10	X
11	Nenhum registrador é selecionado

Fonte: Adaptado de WEBER (2012, p.193)

São reservados 2 *bits* para o formato do modo de endereçamento, o Quadro 4, detalha cada uma das opções.

Quadro 4 – Modos de endereçamento do Ramses

Código	**Modo**	**Operação**	**Descrição**
00	Direto	*end*	O endereço segue a palavra de código da instrução
01	Indireto	*end*, I	O endereço do endereço do operando segue a palavra de código da instrução.
10	Imediato	*#end*	O operando segue a palavra de código da instrução.
11	Indexado	*end*, X	O deslocamento é somado ao registrador X para formar o endereço do operando. Deslocamento representado em complemento de 2.

Fonte Adaptado de WEBER (2012, p. 191-193)

O conjunto de instruções do Ramses possui 16 comandos no total, sendo 2 delas exclusivas em relação ao Neander e ao Ahmes, o Quadro 5 mostra as instruções disponíveis na máquina Ramses, onde *r* representa o registrador a ser usado pela instrução.

Quadro 5 – Instruções Ramses

Binário	**Decimal**	**Hexadecimal**	**Instrução**	**Descrição**
0000 0000	0	00	NOP	Nenhuma operação
0001 0000	16	10	STR *r end*	Armazena registrador *store*
0010 0000	32	20	LDR *r end*	Carrega registrador pela memória *load*
0011 0000	48	30	ADD *r end*	Soma
0100 0000	64	40	OR *r end*	"Ou" lógico
0101 0000	80	50	AND *r end*	"E" lógico
0110 0000	96	60	NOT *r*	Inverte registrador
0111 0000	112	70	SUB *r end*	Subtração
1000 0000	128	80	JMP *end*	Desvio incondicional *jump*
1001 0000	144	90	JN *end*	Desvio condicional *jump if negative*
1010 0000	160	A0	JZ *end*	Desvio condicional *jump if zero*
1011 0000	176	B0	JC *end*	Desvio condicional *jump if carry*
1100 0000	192	C0	JSR *end*	Desvio para sub-rotina
1101 0000	208	D0	NEG *r*	Troca sinal
1110 0000	224	E0	SHR *r*	Deslocamento para direita
1111 0000	240	F0	HLT	Término da execução

Fonte: Adaptado de WEBER (2012, p. 193)

Dada as instruções de Ramses é notável que seu foco está mais voltado em disponibilidade e dinamismo, a possibilidade de mais registradores para uso geral no algoritmo, facilita o gerenciamento dos dados e operações. O uso de sub-rotinas com a instrução JSR fornece mais flexibilidade à execução.

O Ramses traz mais possibilidade de programação e execução em relação ao Neander, com 4 novas instruções e o acesso a registradores complementares, porém, de forma que não sacrifica consideravelmente a simplicidade de seu conjunto de instruções.

2.5 SIMULADOR CESAR

Das máquinas hipotéticas baseadas no Neander, o computador Cesar é o mais extenso e complexo de todos, igualmente desenvolvido pelo professor Raul Weber como ferramenta didática, entretanto o Cesar é recomendado para estudantes mais avançados, pois possui muitos modos de endereçamento e um amplo conjunto de instruções.

Diferente das outras arquiteturas, o Cesar possui uma largura de dados e endereço de 16 *bits*. Dispõe de uma memória de 64 KB. Acesso a 8 registradores (R0 a R7), sendo 6 deles para uso geral, um reservado para ponteiro de pilha e um para contador de programa. Contém 4 *flags* de estados, sendo elas *negative flag* (N), *zero flag* (Z), *overflow flag* (V), *carry out flag* (C). O Cesar apresenta 41 instruções disponíveis em seu conjunto, dessas, 15 são instruções de desvio condicional, 12 instruções aritméticas de um operando, 6 instruções aritméticas de dois operandos, 2 instruções de controle de sub-rotinas e 1 instrução de laço de repetição. A arquitetura de Cesar, inclusive, propõe um *display* de 36 caracteres como saída de dados e um teclado para a entrada de dados (NETO, 2021).

O Cesar contempla no total 8 modos de endereçamento, no qual 3 *bits* são reservados na instrução para ocupar a seleção do endereçamento correspondente. O Quadro 6 descreve os modos de endereçamento disponíveis no Cesar.

Quadro 6 – Modos de endereçamento do Cesar

Código	**Endereçamento**	**Operação**	**Descrição**
000	Registrador	Rx	Registrador contém o operando.
001	Pós-incrementado	(Rx)+	Registrador contém o endereço do operando. Após o acesso o conteúdo do registrador é incrementado por 2.
010	Pré-decrementado	-(Rx)	Registrador é decrementado por 2 e contém o endereço do operando.
011	Indexado	ddd (Rx)	Registrador somado a palavra que segue para formar o endereço do operando.
100	Registrador Indireto	(Rx)	Registrador contém o endereço do endereço do operando.
101	Pós-incrementado indireto	((Rx)+)	Registrador contém o endereço do endereço do operando. Após o registrador é incrementado por 2.
110	Pré-decrementado Indireto	(-(Rx))	Registrador é decrementado por 2 e contém o endereço do endereço do operando.

111	Indexado Indireto	(ddd (Rx))	Registrador somado a palavra que segue para formar o endereço do endereço do operando.

Fonte: Adaptado de WEBER (2012, p. 223 e 224)

O Cesar possui suporte para modos de endereçamento do tipo registrador, direto indireto, indexado, incrementado e decrementado, logo o Cesar dispõe de muita flexibilidade em sua execução, porém a simplicidade é abandonada em favor de sua arquitetura dinâmica.

O conjunto de instruções do Cesar é extremamente extenso, havendo 41 instruções no total, desta forma as instruções estão divididas em tipos para facilitar o estudo. O Quadro 7 contém os tipos de instruções que fazem parte do Cesar.

Quadro 7 – Tipos de instruções do Cesar

Código	**Tipo de Instrução**
0000	Nenhuma instrução.
0001 e 0010	Instruções de códigos de condição.
0011	Instruções de desvio condicional.
0100	Instruções de desvio incondicional.
0101	Instruções de controle de laço de repetição.
0110	Instruções de desvio de sub-rotina.
0111	Instruções de retorno de sub-rotina.
1000	Instruções de um operando.
1001 e 1110	Instruções de dois operandos.
1111	Instrução de parada.

Fonte: Adaptado de WEBER (2012, p. 193)

O Cesar, portanto, tem uma proposta mais similar a sugerida no SAP-2, ambos com mais de 40 instruções e modos de endereçamento para registradores e memória, seguem o princípio de um computador do tipo *Complex Instruction Set Computer* (CISC). Cesar é uma máquina muito poderosa e eficaz para solução de problemas mais complexos e trabalhosos, entretanto é voltada para usuários experientes que possuem mais conhecimento e familiaridade com os conceitos de arquitetura de computadores, principalmente com tipos de endereçamento avançados e programação de sub-rotinas.

2.6 TEMPORIZAÇÃO NEANDER

Para cada instrução é necessário que haja uma sequência de microinstruções utilizadas pela unidade de controle para que a execução da instrução possa acontecer. Uma instrução Neander pode ter até 8 estados de temporização, partindo de T0 até T7 (WEBER, 2010). A temporização das instruções Neander é dada nos Quadros 8 e 9.

Quadro 8 – Temporização dos sinais de controle Neander parte 1

Tempo	**STA**	**LDA**	**ADD**	**OR**	**AND**	**NOT**
T0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
T1	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC
T2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
T3	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	UAL(NOT), carga AC, carga NZ, *goto* t0
T4	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	
T5	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	
T6	carga RDM	*read*	*read*	*read*	*read*	*read*
T7	*write*, *goto* t0	UAL(Y), carga AC, carga NZ, *goto* t0	UAL(ADD), carga AC, carga NZ, *goto* t0	UAL(OR), carga AC, carga NZ, *goto* t0	UAL(AND), carga AC, carga NZ, *goto* t0	

Fonte: Adaptado de WEBER (2010, p. 26).

Quadro 9 – Temporização dos sinais de controle Neander parte 2

Tempo	**JMP**	**JN, N = 1**	**JN, N = 0**	**JZ, Z = 1**	**JZ, Z = 0**	**NOP**	**HLT**
T0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
T1	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC	*read*, incrementa PC
T2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
T3	sel=0, carga REM	sel=0, carga REM	incrementa PC, *goto* t0	sel=0, carga REM	incrementa PC, *goto* t0	goto t0	*halt*
T4	*read*	*read*		*read*			
T5	carga PC, *goto* t0	carga PC, *goto* t0		carga PC, *goto* t0			
T6							
T7							

Fonte: Adaptado de WEBER (2010, p. 26).

2.7 NEANDER COMO FERRAMENTA PEDAGÓGICA

A simulação de uma máquina hipotética em simultâneo com diagramações e animações da arquitetura do computador em paralelo à execução, enriquece a assimilação do estudante sobre a engenharia em operação de todo o conjunto computacional. A realização de um estudo sobre o uso da simulação no ensino de engenharia conclui:

> Por meio da análise de nove artigos, foi possível identificar que o principal objetivo da simulação é a visualização e animação, de forma a demonstrar conceitos e teorias, contribuindo para a compreensão e consequente aprendizado dos estudantes. A simulação associada a outras metodologias de ensino propicia o desenvolvimento principalmente de competências profissionais, mas também interpessoais e pessoais. (GARBIN; KAMPFF, 2021, p. 9).

Segundo Costa e Silva (2018, p. 29) "Uma grande dificuldade no ensino de arquitetura de computadores é fazer com que os alunos compreendam corretamente o funcionamento de um processador." A escolha do computador Neander como principal ferramenta pedagógica para o estudo de arquitetura de computadores em relação aos outros simuladores é justificada

pela sua simplicidade de funcionamento e o seu potencial de execução. A máquina Neander aborda os principais conceitos de operações de uma arquitetura de computador, porém sem a sobrecarga de complexidade com modos de endereçamento mais avançados ou um número excessivo de instruções. Com o Neander é possível desenvolver um simples algoritmo que contempla os fundamentos primordiais de trabalho de um computador, ler um dado na memória, realizar um processamento com este dado e armazená-lo de volta a memória, desta forma, além do estudante se familiarizar com a programação *assembly*, todas as etapas do procedimento ficam compreensíveis.

Outro ponto relevante que fomenta a necessidade de um novo simulador Neander para auxílio na disciplina de arquitetura de computadores é que:

> Ainda há carência de ferramentas educacionais para auxiliar o ensino de determinados conceitos e tecnologias relacionados a esta disciplina. Como exemplo, pode-se citar a carência de ferramentas de apoio ao ensino de tópicos relacionados à organização e à execução de instruções no computador Neander. (RUDOLFO *et al*., 2014, p. 1648).

O computador Neander tem a arquitetura hipotética mais fiel em relação ao conceito de *simple as possible* (SAP) apresentado por Albert Paul Malvino e Jerald A. Brown, e é a máquina mais semelhante a proposta em SAP-1, na qual pretende ser um computador de forma mais simplória possível, entretanto o Neander não dispõe das limitações de desvio com condicionamento e escrita na memória.

As variantes do Neander apresentadas como Ahmes, Ramses e Cesar são propostas excelentes para aquele estudante que deseja aprofundar seus conhecimentos e buscar mais experiências práticas, porém para um iniciante mais instruções e mais modos de endereçamento podem parecer intimidantes. Portanto o Neander entrega as operações essenciais da forma mais elementar possível, a fim de simplificar o entendimento e manter o foco nos conceitos principais de funcionamento de uma arquitetura de computador.

3 MATERIAL E MÉTODOS

Nesse capítulo são descritos os material e métodos utilizados para a realização desse trabalho. Foi desenvolvido um *software* de simulação do computador Neander que comtempla os principais processos de configuração da memória de programa e dados, execução direcionada com unidade de *clock* ajustável, visualização e manipulação de registradores, visualização de saída de *flags* e sinais de controle, diagramação de máquina em tempo de execução, importação e exportação de programa, compatíveis com simulador original. O simulador foi projetado com base em tecnologias *web*, desta forma, a ferramenta se torna disponível para navegadores e dispositivos *mobiles* sem a obrigatoriedade de nenhum *download* por parte do usuário.

3.1 TECNOLOGIAS DE DESENVOLVIMENTO

Atualmente aplicativos *mobile* e *websites* são muito mais populares e desejáveis em relação a aplicações *desktops* quando não existe uma demanda computacional alta, isso favorece que a simulação do computador Neander aconteça em um ambiente *web*. Tecnologias como Apache Cordova permitem a compilação nativa de soluções *web* para *mobile*, desta forma o *software* final fica disponível para sistemas operacionais como Android e IOS além de ser acessível em um domínio na rede mundial de computadores em formato de *website* através de qualquer navegador.

A construção do conteúdo foi feita usando *HyperText Markup Language* (HTML), a estilização e o layout foram projetados com o uso de *Cascading Style Sheets* (CSS), e a programação completa do simulador, junto com a captura de eventos e atualização de conteúdos na interface, foi utilizado a linguagem de programação JavaScript. Entretanto para um desenvolvimento mais profissional e eficiente foi utilizado o *framework* Quasar, de código aberto e baseado por cima de outra tecnologia de *framework web* Vue.js, no qual proporcional o desenvolvimento de uma aplicação *single-page application* (SPA), compilação para Android e IOS por meio da tecnologia Cordova, componentização de recursos, atualização dinâmica do *Document Object Model* (DOM) do HTML, uma série de componentes de interface pré-configurados para auxiliar o desenvolvimento, o uso de recursos modernos do ECMAScript8 (ES8) no qual complementa a linguagem de programação JavaScript, e uma gama de outras vantagens.

Para o gerenciamento de pacotes e acesso a bibliotecas de terceiros foi utilizado a tecnologia *Node Package Manager* (NPM) que fornece um gerenciador de pacotes para o Node.JS, a tecnologia Node.js disponibiliza a execução de códigos JavaScript fora de um navegador *web*. Intrinsecamente, a partir do Quasar, o Node.js e o NPM também são responsáveis por providenciar as ferramentas de compilação e disponibilizar um servidor local de desenvolvimento, este servidor de desenvolvimento é construído e gerenciado pela tecnologia Vite que provê um monitoramento dos arquivos e atualiza a aplicação ao detectar qualquer mudança a partir de um processo conhecido como *Hot Module Replacement* (HMR).

Como tecnologia complementares, foi usado a linguagem de programação Stylus para o pré-processamento dinâmico de instruções CSS, e o ESLint uma ferramenta de análise estática de códigos para assegurar uma qualidade de escrita de *software* profissional seguindo regras padronizadas de formatação.

Como tecnologias de ambientes externas à aplicação foram empregados o sistema operacional Linux Ubuntu e o editor de códigos Visual Studio Code.

Para o versionamento do *software* foi aplicado a tecnologia Git, um sistema de controle de versões distribuído que disponibiliza um repositório profissional do projeto com todo histórico de andamento do projeto. O repositório Git é agregado ao GitHub, uma plataforma de hospedagem para garantir a disponibilidade do projeto.

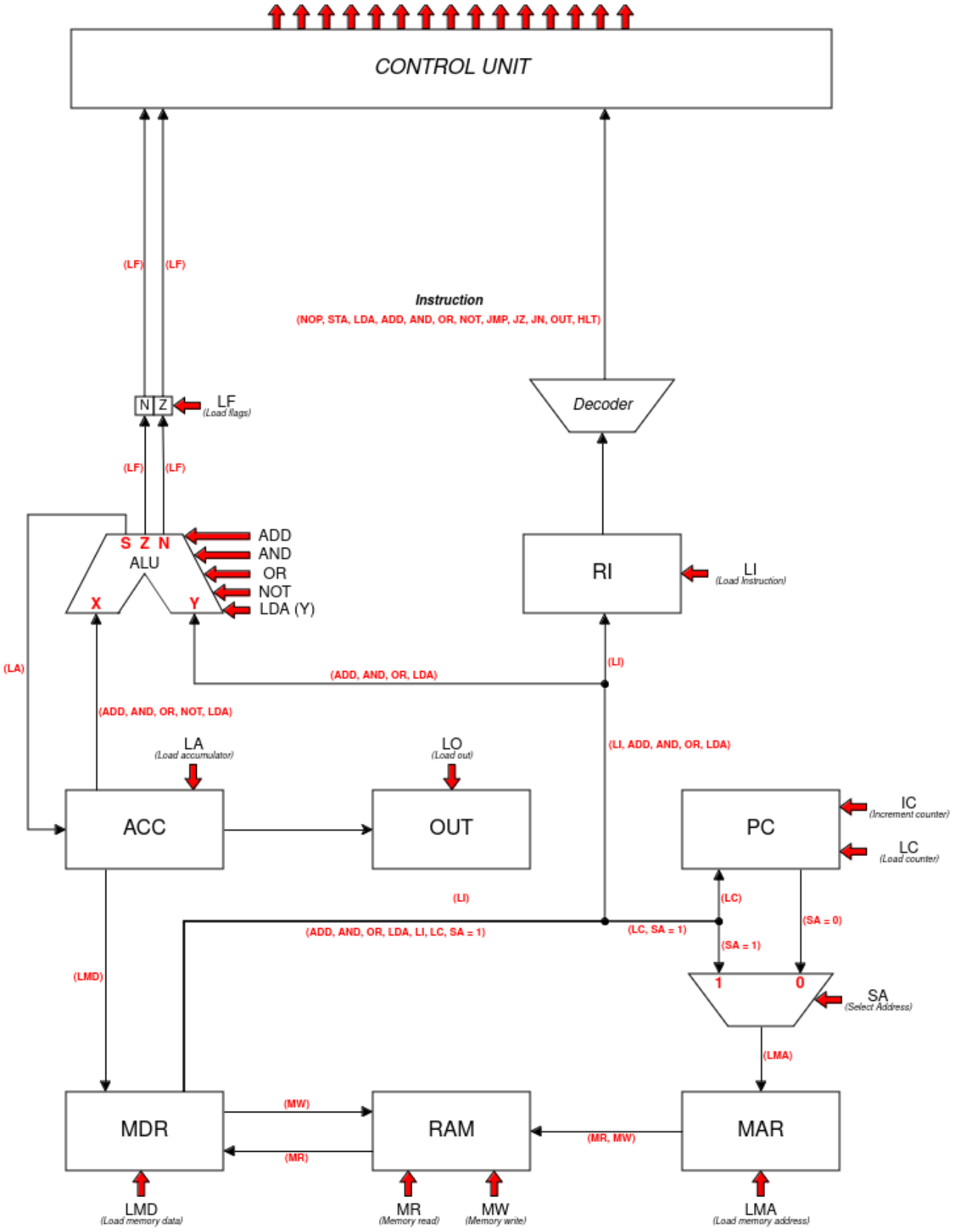
3.2 DIAGRAMAÇÃO EM TEMPO DE EXECUÇÃO

Com o intuito de proporcionar uma experiência completa de simulação a fim de auxiliar na absorção dos conceitos estudados em arquitetura de computador, este trabalho contempla uma funcionalidade exclusiva de diagramação que reflete o estado da máquina atual no contexto da simulação, por meio de blocos os componentes como registradores, seletores, decodificadores de instrução, ALU e a unidade de controle são representados, e por meio de setas a comunicação entre a unidade de controle com os demais componentes são demonstradas.

A principal tecnologia usada para a construção dessa funcionalidade é o HTML5 Canvas, no qual é uma área *bitmap* de modo imediato da tela que pode ser manipulada com JavaScript. Modo imediato se refere à maneira como a tela renderiza *pixels* na tela (FULTON, S.; FULTON, J. 2013).

A figura 5, apresenta um desenho de como deve ser a diagramação do computador Neander.

Figura 5 – Diagramação de Simulação do computador Neander



Fonte: Autoria própria

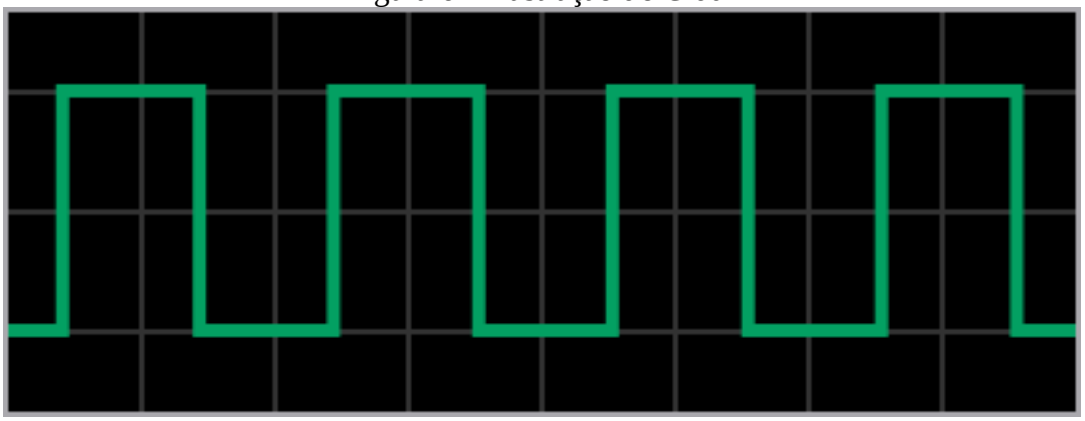
3.3 SIMULAÇÃO DE CLOCK

Para manter a fidelidade de uma execução de circuito computacional, foi projetado um módulo ajustável para simular o sinal de um *clock*, é possível ajustar seu funcionamento para trabalhar com um controle monoestável ou astável, bem como ajustar sua frequência de oscilação. Desta forma, o usuário tem a possibilidade de executar seu programa *tick* a *tick*, microinstrução a microinstrução, a fim de possibilitar uma análise mais minuciosa de seu algoritmo e do funcionamento da arquitetura.

A oportunidade de possuir um total controle do mecanismo de *Clock* agrega muito valor para o entendimento de como a máquina opera, pois, cada passo pode ser analisado e discutido, desta forma os conceitos de arquitetura de computador se tornam muito mais fáceis de serem apresentados e explicados.

A Figura 6 apresenta uma ilustração de como o sinal de *clock* é exibido para o usuário de maneira dinâmica e interativa.

Figura 6 – Ilustração do *Clock*



Fonte: Autoria Própria

A ferramenta utilizada para construir esse módulo de *Clock*, é a mesma usada para a construção da diagramação do computador, o HTML5 Canvas, desta forma é possível desenhar o sinal de *clock* representado em uma escada de tempo e atualizado de maneira dinâmica com a linguagem de programação JavaScript. O seu controle é feito por componentes de UI externos ao Canvas.

3.4 COMPATIBILIDADE COM SIMULADOR ORIGINAL

Com o propósito incentivar o uso do simulador desenvolvido, o sistema de salvamento, importação e exportação de arquivos, possui total compatibilidade com o simulador original WNeander projetado por Raul Weber.

WNeander trabalha com arquivos no formato MEM, que pode ser usado para representar blocos de memória, o arquivo gerado pelos simuladores representa a memória do Neander composta de 256 *bytes*, entretanto o arquivo gerado possui 516 *bytes*, os 4 primeiros *bytes* são reservados como cabeçalho do arquivo, e para cada linha são reservados 2 *bytes*, por mais que uma instrução Neander sempre ocupará 1 *byte*, de tal forma que o outro *byte* reservado sempre está zerado.

A Figura 7 exemplifica a visualização hexadecimal de um arquivo MEM de um programa feito no Simulador Neander para calcular a sequência de Fibonacci.

Figura 7 – Representação Hexadecimal de um arquivo tipo MEM

```
00000000: 034e 4452 2000 3300 1000 3d00 2000 3200  .NDR .3...=. .2.
00000010: c800 3000 3d00 1000 3e00 2000 3d00 1000  ..0.=...>. .=...
00000020: 3c00 2000 3e00 1000 3d00 2000 3c00 9000  <. .>...=. .<...
00000030: 0000 8000 0600 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0100 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 1500 2200  ..............".
00000080: 2200 0000 0000 0000 0000 0000 0000 0000  "...............
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000200: 0000 0000                                ....
```

Fonte: Autoria Própria

A estratégia usada para ler e escrever arquivos MEM é trabalhar com *arrays* do tipo *unsigned int* de 16 *bits*, desta forma cada elemento representa uma linha de endereço do computador Neander.

O código de leitura, importação, de um arquivo MEM é demonstrado na Figura 8, o arquivo recebido é convertido para uma sequência binária, logo após o *buffer* genérico é definido como um *array* do tipo *unsigned int* de 16 *bits*, com isso é possível identificar se o arquivo em questão é um MEM válido por meio de um *checksum* no cabeçalho, assim é considerado os 4 primeiros *bytes*, neste caso uma soma com os 2 primeiros elementos do *array*, caso a soma resultar no número em específico, que representa a soma original, é positivo para um arquivo MEM válido, em seguida os dados da RAM são carregados como inteiros desconsiderando os elementos de cabeçalho.

Figura 8 – Código JavaScript para Leitura de arquivo MEM

```
1  function readFile (file) {
2      file.arrayBuffer().then(buffer => {
3          const arr = new Uint16Array(buffer)
4          // if filetype .mem
5          if (arr[0] + arr[1] === 41031) {
6              this.computer.RAM = [...arr.slice(2)]
7              this.configStore.saveRAM(this.computer.RAM)
8              return
9          }
10         // ...
11     })
12 }
```

Fonte: Autoria Própria

O código de escrita, exportação, demonstrado na Figura 9, simplesmente compacta os valores da memória RAM do Neander para o tipo *unsigned int* 16 *bits* e adiciona 2 números de prefixos que representam o cabeçalho de um arquivo MEM, em seguida seu conteúdo é encapsulado em um objeto *Blob* que posteriormente pode ser usado pela aplicação para disponibilizar um *download* do arquivo ao usuário.

Figura 9 – Código JavaScript para Escrita de arquivo MEM

```
1  function intArrayToMem (arr) {
2      const prefix = [19971, 21060]
3      return new Blob([new Uint16Array([...prefix, ...arr])])
4  }
```

Fonte: Autoria Própria

3.5 OPERADORES BITWISE

Para simular as operações da ALU, existe um recurso que muitas linguagens de programação proporcionam que é o uso dos operadores *bitwise*, são usados para realizar operações em nível de *bits* com números inteiros, isto significa que os valores das variáveis no código da simulação que representam a memória, registradores e barramento podem ser do tipo inteiro, mantendo a simplicidade do código sem a necessidade de funções complicadas para simular as operações binárias trabalhando com *strings* representando *bits*.

O Quadro 10, contém as principais operações *bitwise* disponíveis na linguagem JavaScript:

Quadro 10 – Operadores *Bitwise*

Operador	**Instrução**	**Descrição**
&	AND	Definir cada bit para 1 caso ambos os bits sejam 1
|	OR	Definir cada bit para 1 caso algum dos bits sejam 1
^	XOR	Definir cada bit para 1 apenas se um dos bits seja 1
~	NOT	Inverte todos os bits

Fonte: Adaptado de FLANAGAN (2011, p. 62).

Para demonstrar as operações, o Quadro 11 contém alguns exemplos:

Quadro 11 – Exemplos de operações *Bitwise*

NOT	0111 = 1000 10101011 = 01010100
AND	0101 0011 = 0001 0011 0010 = 0010
OR	0101 0011 = 0111 0010 1000 = 1010
XOR	0101 0011 = 0110 0010 1010 = 1000

Fonte: Autoria própria.

Este recurso, portanto, facilita muito a programação do comportamento da ALU em linguagens de programação de alto nível.

3.6 LÓGICAS DE SIMULAÇÃO PARA ARQUITETURA DE MÁQUINA

Para definir o comportamento de execução de uma máquina Neander, devem ser considerados o padrão de dados de sua memória RAM, os dados de seus registradores, o estado de unidade de controle, bem como seu contador interno para temporização de instrução, e o estado das *flags*. A técnica utilizada foi adotar a utilização da programação orientada a objetos para facilitar o gerenciamento da relação entre suas variáveis, desta forma uma classe Neander é declarada. A variável que referência a memória RAM é definida como um *array* do tipo *int*, não há necessidade de se preocupar com valores em formato numérico binário ou hexadecimal, já que os operadores de *bitwise* são empregados, desta forma o tipo *int* se torna mais conveniente. Os registradores são declaradas variáveis do tipo *int* correspondentes para cada registrador da máquina. A representação do estado da unidade de controle e o estado das *flags* são declarados objetos de estrutura *key-value*, no qual a *key* identifica o atributo referenciado e o *value* o valor armazenado nesse atributo.

A continuidade de sua execução é definida por um método nomeado como *next*, no qual representa, em termos da arquitetura do computador Neander, a borda de subida de um sinal de *clock*, o momento chave que os componentes reconhecem a necessidade da mudança de estado que representa os intervalos de qualquer circuito combinatório. Esse método é responsável pela execução do reconhecimento da microinstrução a ser executada, bem como a atribuição de saída da unidade de controle correspondente a microinstrução identificada, e por fim a execução da microinstrução orientada pelo estado da unidade de controle. Os métodos de identificação e execução da microinstrução são detalhados neste trabalho com maior aprofundamento.

3.6.1 Lógica de Execução de Microinstrução

No meio físico, a execução acontece de maneira paralela entre os componentes do *hardware*, pois os sinais de controle são conectados de maneira independente a cada componente, por exemplo, o sinal de incremento de PC é conectado diretamente ao módulo contador responsável por incrementar o registrador que representa o endereço da instrução a executar no programa, desta forma o restante dos componentes não precisam ter ciência do que ocorreu, assim cada um executa a sua função, paralelamente, de maneira independente ao comando de seu respectivo sinal da unidade de controle.

No contexto da programação, na maioria dos casos, o código é lido e executado de forma sequencial, assim a estratégia adotada foi condicionar, de maneira independente, cada sinal da unidade de controle, e no bloco de condicionamento executar sua respectiva operação com os componentes envolvidos. A figura 10 possui o trecho de código referente a função de execução de microinstrução em JavaScript no qual realiza a lógica comentada.

Figura 10 – Código JavaScript de Execução de Microinstrução Neander

```
function runMicroInstruction () {
    if (this.controlUnit.LMA) { this.MAR = this.controlUnit.SA ? this.MDR : this.PC }
    if (this.controlUnit.MR)  { this.MDR = this.read(this.MAR) }
    if (this.controlUnit.MW)  { this.write(this.MAR, this.MDR) }
    if (this.controlUnit.LMD) { this.MDR = this.ACC }
    if (this.controlUnit.IC)  { this.incrementPC() }
    if (this.controlUnit.LC)  { this.PC = this.MDR }
    if (this.controlUnit.LI)  { this.RI = this.MDR }
    if (this.controlUnit.LO)  { this.OUT = this.ACC }

    let ALU = 0
    if (this.controlUnit.ADD) { ALU = (this.ACC + this.MDR) & 255 }
    if (this.controlUnit.AND) { ALU = this.ACC & this.MDR }
    if (this.controlUnit.OR)  { ALU = this.ACC | this.MDR }
    if (this.controlUnit.NOT) { ALU = this.ACC ^ 255 }
    if (this.controlUnit.LDA) { ALU = this.MDR }

    if (this.controlUnit.LA)  { this.ACC = ALU }
    if (this.controlUnit.LF)  { this.updateFlags(ALU) }

    if (this.controlUnit.T0)  { this.gotoT0() } else { this.controlTime += 1 }
    if (this.controlUnit.HLT) { this.HLT() }
}
```

Fonte: Autoria Própria

3.6.2 Lógica de Identificação de Microinstrução

São considerados dois fatores para se determinar a microinstrução a ser executada, a decodificação da instrução em RI e valor do contador interno de temporização de instrução da unidade de controle. A decodificação de instrução, no contexto do *hardware*, é traduzir o número armazenado no registrador de instrução em um sinal de tensão baixa ou alta para a unidade de controle, desta forma a unidade de controle possui uma entrada de sinal para cada instrução existente, isso facilita a lógica interna da unidade de controle em reconhecer, de forma combinatória, as saídas correspondentes. Entretanto uma instrução necessita de vários tempos de *clock*, pois uma instrução Neander pode conter até 8 microinstruções, de T0 a T7, isso cria a necessidade de contagem, ou temporização, para se determinar qual intervalo o circuito se encontra, para isso é necessário que a unidade de controle possua um contador interno, o valor desse contador é usado como uma das entradas no circuito combinatório para

definir as saídas correspondentes da unidade de controle, e no fim de cada microinstrução esse contador é incrementado, com exceção da microinstrução *goto T0* que reseta o contador.

Para o desenvolvimento do código da simulação, é necessário um padrão de dados que represente tanto as microinstruções quanto a instrução decodificada do número armazenado em RI, como o Javascript é uma linguagem de programação com tipagem dinâmica fraca, não há necessidade de sofisticar padrões de dados mais consistentes, caso contrário isso aumentaria a complexidade e o tamanho do código consideravelmente, desta forma o conceito usado foi representar as instruções e microinstruções como *strings*, pois além de facilitar a leitura, o código é simplificado, já que o Javascript permite manobrar os dados de maneira direta e objetiva.

Os tempos de T0 a T2 é conhecido como o momento de *fetch instruction*, pois é o ciclo responsável por obter a instrução correspondente do programa em execução na memória e transferir para o registrador de instrução, nesse período o valor de RI não importa, já que para todas as instruções existentes esse ciclo obrigatoriamente sempre acontecerá, desta forma o código condiciona esses três tempos apenas com a temporização de instrução. Os tempos de T3 a T7 representam o ciclo de *execution instruction*, nesses são necessários considerar tanto a temporização de instrução quanto a instrução de RI.

O retorno do método de identificação de microinstruções é composto sempre por um *array* de *strings* que cada elemento representa uma microinstrução que a unidade de controle deve definir. A figura 11 apresenta o trecho de código usado comentado.

Figura 11 – Código JavaScript de Obtenção de Microinstruções Neander

```
function getMicroInstruction () {
    // from t0 to t2 (fetch)
    if (this.controlTime < 3) return this.controlTime === 0 ? ['LMA'] : (this.controlTime
        === 1 ? ['MR', 'IC'] : ['LI'])

    // from t3 to t7 (execution)
    const timingMicrocodes = {
        NOP: [['T0']],
        STA: [['LMA'], ['MR', 'IC'], ['SA', 'LMA'], ['LMD'], ['MW', 'T0']],
        LDA: [['LMA'], ['MR', 'IC'], ['SA', 'LMA'], ['MR'], ['LDA', 'LA', 'LF', 'T0']],
        ADD: [['LMA'], ['MR', 'IC'], ['SA', 'LMA'], ['MR'], ['ADD', 'LA', 'LF', 'T0']],
        OR:  [['LMA'], ['MR', 'IC'], ['SA', 'LMA'], ['MR'], ['OR', 'LA', 'LF', 'T0']],
        AND: [['LMA'], ['MR', 'IC'], ['SA', 'LMA'], ['MR'], ['AND', 'LA', 'LF', 'T0']],
        NOT: [['NOT', 'LA', 'LF', 'T0']],
        JMP: [['LMA'], ['MR'], ['LC', 'T0']],
        JN: this.flags.N ? [['LMA'], ['MR'], ['LC', 'T0']] : [['IC', 'T0']],
        JZ: this.flags.Z ? [['LMA'], ['MR'], ['LC', 'T0']] : [['IC', 'T0']],
        OUT: [['LO', 'T0']],
        HLT: [['HLT', 'T0']]
    }
    return timingMicrocodes[this.decoder(this.RI)][this.controlTime - 3]
}
```

Fonte: Autoria Própria

3.7 PESQUISA DE VALIDAÇÃO DO SIMULADOR

A fim de obter resultados válidos e fundamentados sobre a efetividade do simulador, uma pequena pesquisa de avaliação (Apêndice I) foi conduzida com uma amostra de alunos do curso de Engenharia da Computação da Universidade de Ribeirão Preto - UNAERP. A ferramenta utilizada, responsável pela coleta de dados do questionário e gerenciamento de pesquisa, foi o Google Forms.

O modelo de questões foi elaborado seguindo a escala Likert, por definição:

> A escala Likert foi criada para medir "atitude" de uma forma cientificamente aceita e validada em 1932. Uma atitude pode ser definida como formas preferenciais de se comportar/reagir em uma circunstância específica enraizada em uma organização relativamente duradoura de crenças e ideias (em torno de um objeto, um sujeito ou um conceito) adquiridas por meio de interações sociais (JOSHI, 2015, v. 7, p. 396-403).

Desta forma, para cada um dos itens do questionário, o participante deveria indicar a opção que representasse sua percepção sobre o uso do *software*, levando em conta:

1. Concordo totalmente;
2. Concordo;
3. Não concordo, nem discordo;
4. Discordo;
5. Discordo totalmente.

Pouco antes do processo de avaliação, o simulador foi apresentado e explicado para o grupo de alunos, após isso cada aluno teve a oportunidade de conduzir a própria utilização da aplicação, em seguida o questionário foi aberto e disponibilizado para os participantes. As questões do formulário foram elaboradas referentes a aspectos de funcionalidade, eficiência e usabilidade. As questões são:

1. O *design* e a interface do simulador são visualmente agradáveis e intuitivos.
2. Eu achei o simulador fácil de usar e manipular.
3. O simulador responde rapidamente e sem falhas.
4. O simulador oferece funcionalidades suficientes para que eu possa estudar e aprimorar meus conhecimentos de arquitetura de computador.
5. Eu considero o simulador apresentado como uma nova versão mais completa do simulador original Neander.
6. O simulador facilita a abstração de conceitos de arquitetura de computador de maneira simples e intuitiva.

7. O simulador atende e apresenta os mecanismos básicos de uma máquina hipotética Neander.
8. O simulador contribuiu para o meu entendimento de como uma máquina Neander funciona.
9. Eu estou satisfeito(a) com o simulador como um todo.
10. Eu recomendaria este simulador para outros estudantes de engenharia da computação.

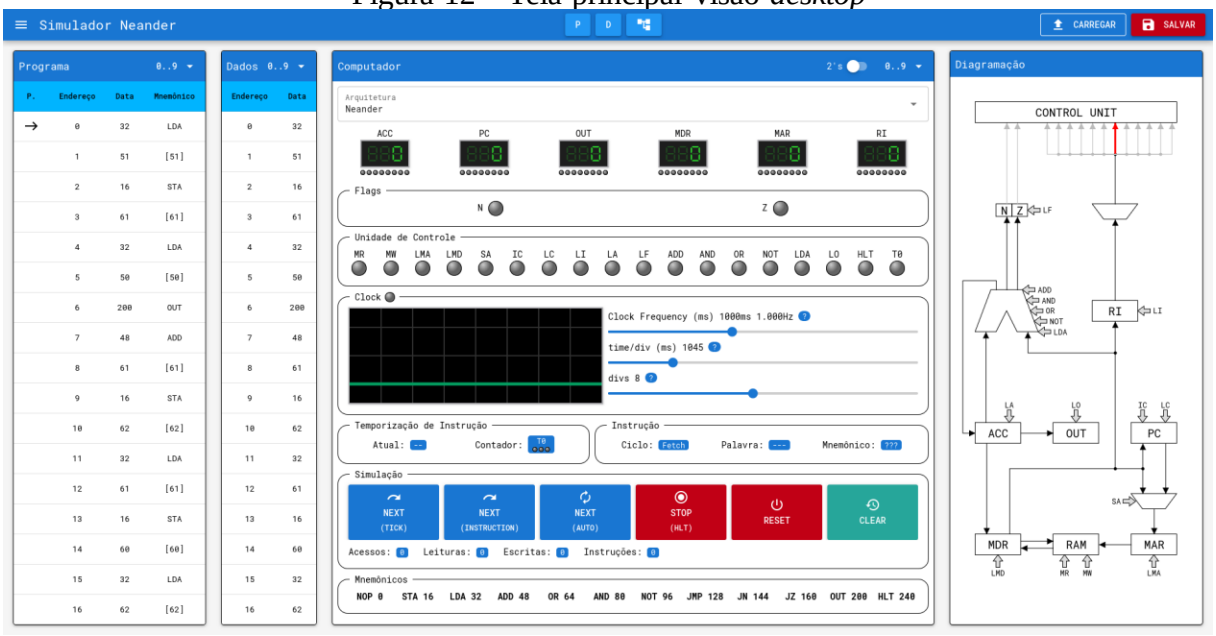
Com o resultado das respostas fornecidas pelos participantes é possível estimar a aceitação do simulador, bem como sua eficácia em apresentar os conceitos de arquitetura de computador a partir da máquina Neander.

4 RESULTADOS E DISCUSSÃO

Neste capítulo são apresentadas e descritas as funcionalidades presentes no *software* desenvolvido para simulação do computador Neander, por meio de suas telas e alguns exemplos todos os principais recursos são demonstrados, e os resultados obtidos na pesquisa de validação realizada são analisados.

Na Figura 12 é apresentada a tela completa do simulador na visão *desktop*, a interface completa é constituída de uma barra de topo com a finalidade de título e menu de tarefas, e quatro janelas independentes responsáveis pela interação e exibição da máquina Neander, cada uma dessas janelas possui um papel fundamental na composição da simulação, e são comentadas separadamente neste capítulo.

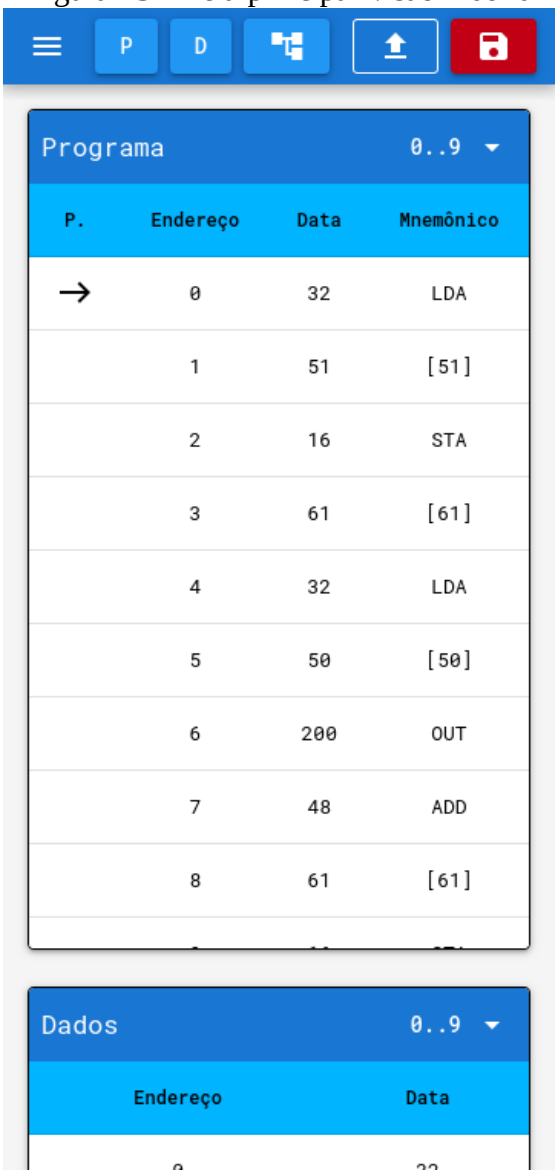
Figura 12 – Tela principal visão *desktop*



Fonte: Autoria Própria

As interfaces foram projetadas para suportar visões *mobiles* e *desktops*, dessa forma possuem responsividade dinâmica em função do dimensionamento da tela. Na Figura 13 é apresentada a mesma tela principal da Figura 12, mas desta vez, em visão *mobile*, as janelas são apresentadas em formato de coluna ao invés de linha, logo é possível uma acessibilidade satisfatória dos recursos mesmo em dispositivos com telas pequenas.

Figura 13 – Tela principal visão *mobile*



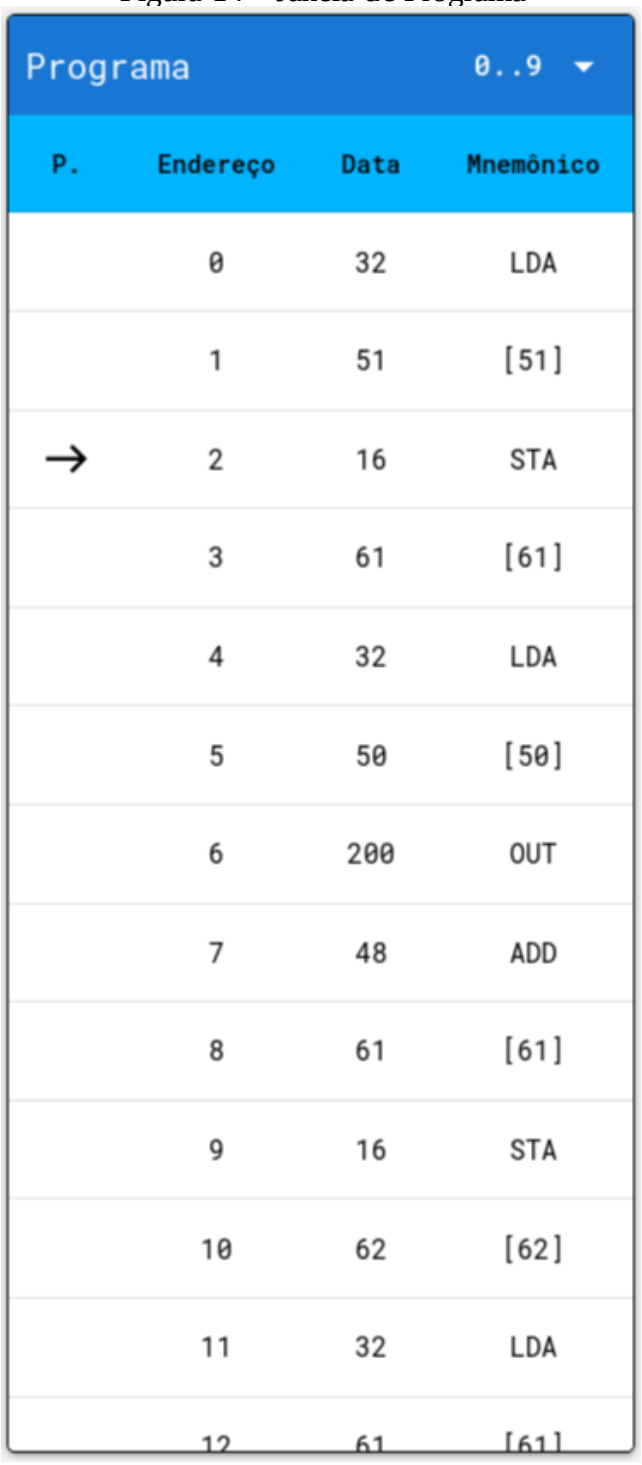
Fonte: Autoria Própria

A primeira janela é definida como "Programa", tem a finalidade de representar e manipular as instruções *assembly* do programa presentes na memória RAM do computador, é por meio dela que geralmente é programado a rotina de execução das instruções. Os dados da memória RAM são organizados em formato de tabela, possuem quatro colunas, Ponteiro ou "P.", Endereço, Dado e Mnemônico, respectivamente. A coluna de Ponteiro reflete a instrução que será executada, espelha o mesmo valor do registrador PC. A coluna de Endereço representa o índice da linha da memória, no Neander a memória é definida de 0 a 255. A coluna de Dados representa o dado armazenado na linha da memória, o dado, da mesma forma no endereço, pode variar de 0 a 255. E por fim, a coluna Mnemônico representa a decodificação do dado para formato de instrução.

A edição do dado é feita ao clicar na linha da tabela, um *popup* ao lado é aberto com a possibilidade de escolher uma instrução ou digitar um valor para o dado. É possível editar a visualização da base numérica, no campo de seleção na parte superior direita da janela, para binário, decimal ou hexadecimal.

A Figura 14 apresenta a janela de Programa.

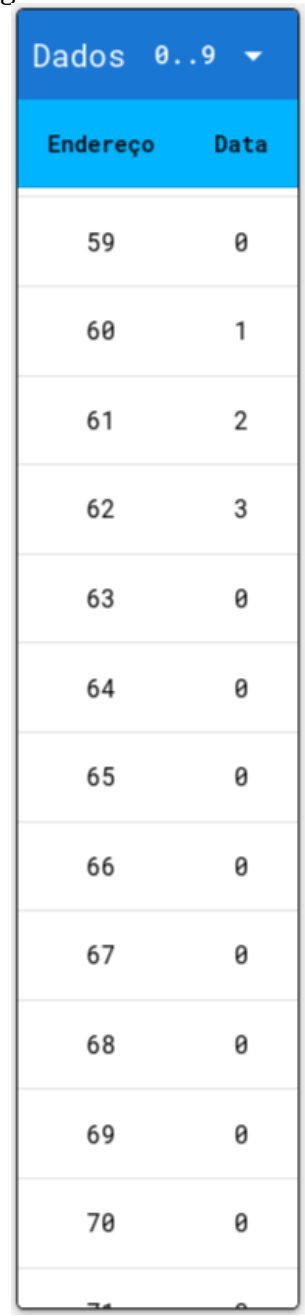
Figura 14 – Janela de Programa



Fonte: Autoria Própria

A segunda janela é apresentada como "Dados", muito semelhante a janela de Programa, porém seu intuito é mais voltado para definição de constantes e variáveis usadas na execução do programa, porém devido a mesma memória RAM ser usada para programa e dados, ambas as janelas compartilham as mesmas informações, dessa forma a edição de uma é refletida na outra. A janela de Dados possui as mesmas funcionalidades comentadas na janela de Programa, porém possui apenas as colunas de Endereço e Dado. A Figura 15 apresenta a janela de Dados.

Figura 15 – Janela de Dados



Fonte: Autoria Própria

A terceira janela é a principal janela que exibe e controla grande parte dos recursos do simulador, definida como "Computador", nela é possível ver e editar os dados dos registradores, observar a saída das *flags* e do barramento da unidade de controle, observar o funcionamento do *clock*, bem como definir frequência e exibição de sinal no quadro de osciloscópio, observar dados de temporização de instrução e palavra de instrução, e por fim controlar a simulação por meio dos botões de ação.

Esta janela foi projetada com a ideia de ser o mais interativa e imersiva possível, dessa forma o *design* dos componentes foi inspirado em componentes de *hardware*, os registradores são representados em *displays* de 7 segmentos, os dados binários são representados por LEDs apagados e acessos, o *clock* além de possuir um LED de sinal tem um quadro de osciloscópio dedicado com a animação do sinal, e o restante dos componentes possuem todos animações de *click* e *hover*, logo proporciona uma experiência confortante e chamativa ao usuário.

Existem seis botões de ação responsáveis por controlar o funcionamento do computador Neander, são eles "Próximo instante", "Próxima instrução", "Próximo automático", "Parar HLT", "Reiniciar" e "Limpar". O botão "Próximo instante" executa apenas um sinal de *clock*, desta forma apenas um intervalo na temporização é avançado, logo o usuário tem a oportunidade de acompanhar cada ciclo da máquina no seu próprio ritmo, essa operação é equivalente a funcionalidade de monoestável de um gerador de pulso. O botão "Próxima instrução" continua a execução até que o contador da temporização atinja T0, logo apenas os pulsos necessários para finalizar a execução da instrução atual são enviados. O botão "Próximo automático" continua a execução da máquina indefinidamente até que alguma instrução de parada seja enviada, isto é equivalente a definir o *clock* no modo astável, assim os pulsos de *clock* são gerados continuamente. O botão "Parar HLT" representa a ação de pausar a máquina, mesmo processo da instrução *Halt*, isso é equivalente a fazer o *clock* retornar para o modo monoestável, desta forma os pulsos de *clock* param de ser enviados até que algum botão de próximo seja pressionado. O botão "Reiniciar" volta a máquina ao estado original, instância a máquina novamente e zera todos os valores dos registradores, apenas mantém os dados da memória RAM. E por fim, o botão "Limpar" zera os dados de contagem da simulação, os valores de quantas leituras, escritas, acessos e instruções ocorreram são zerados.

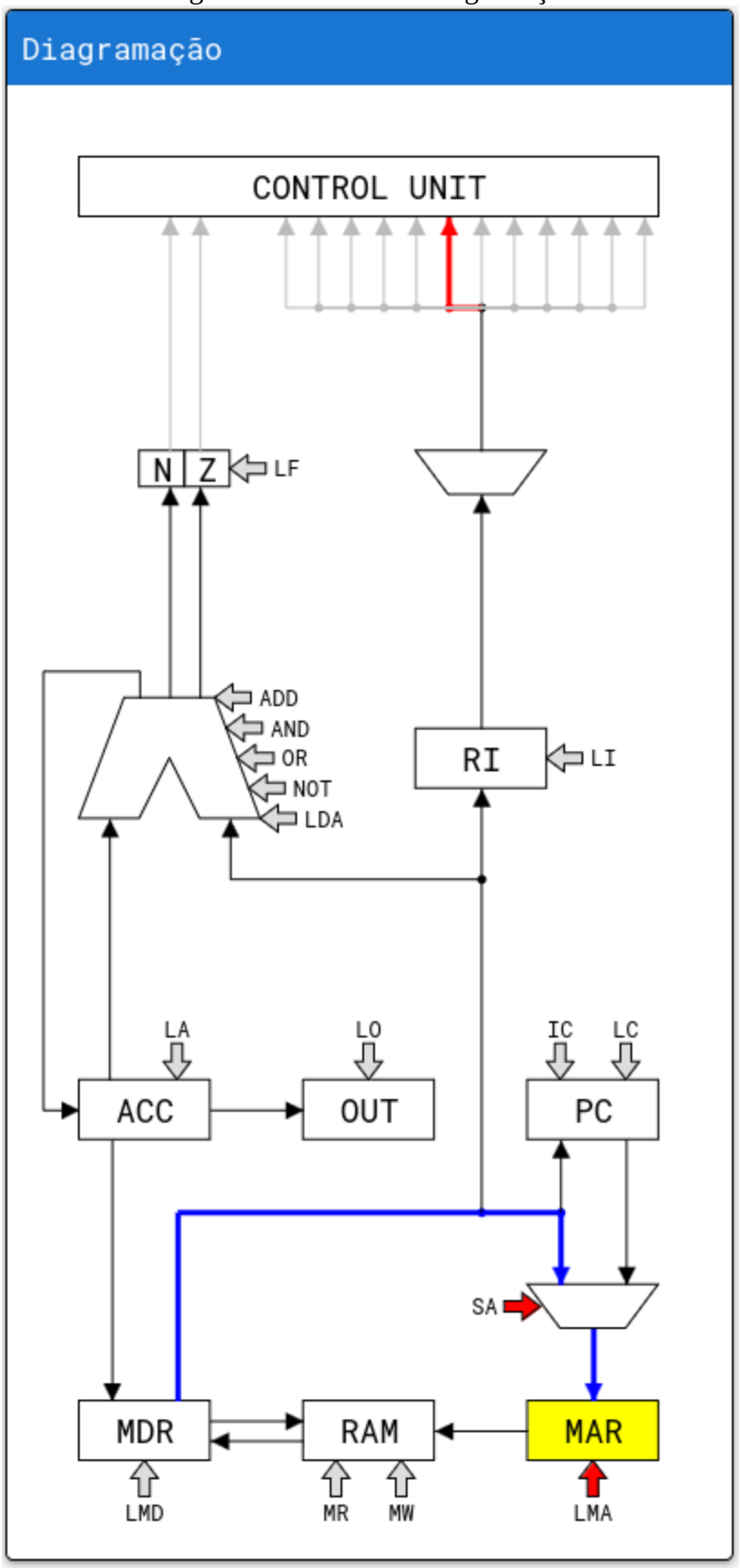
A Figura 16 apresenta a janela de Computador.

Figura 16 – Janela de Computador

Fonte: Autoria Própria

A última janela é definida como "Diagramação", onde é feito, de maneira dinâmica, a representação do fluxo de temporização da máquina, um diagrama com todos os componentes da arquitetura do computador são desenhados, os itens em colorido representam um processamento ou uma troca de dados em andamento, desta forma a cada novo instante de tempo na simulação este diagrama é atualizado para refletir a operação em execução, logo o usuário em tempo real tem a possibilidade de observar como a arquitetura trabalha, e o que de fato ocorre na máquina quando o seu programa escrito é executado. Esta dinâmica de obter uma resposta visual da arquitetura por meio da execução do próprio código *assembly* programado no simulador facilita a abstração dos conceitos estudados e colabora para um aprendizado mais profundo e interativo. A Figura 17 apresenta a janela de Diagramação.

Figura 17 – Janela de Diagramação

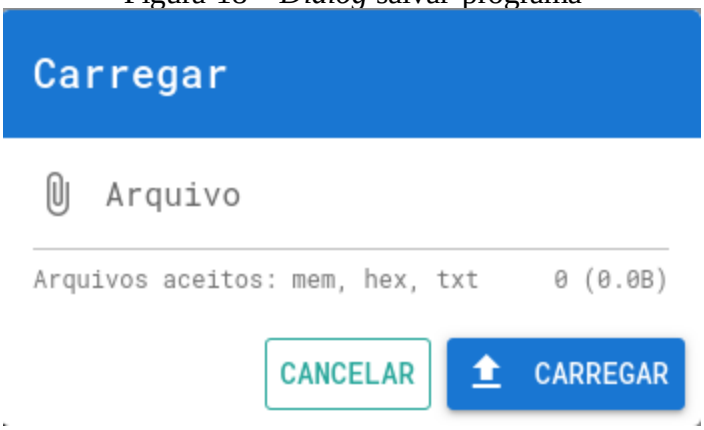


Fonte: Autoria Própria

No canto superior direito da tela na barra de topo existem dois botões responsáveis pela importação e exportação de programas, é possível salvar os dados da memória RAM assim como carregá-los de um arquivo externo, quando pressionados uma janela do tipo *dialog* é carregada da respectiva operação.

A Figura 18 apresenta o *dialog* responsável por salvar o programa, existem dois campos que devem ser preenchidos, o primeiro campo é a escolha do formato de arquivo que será gerado, e o segundo é o nome do arquivo que será criado, após isso é possível clicar no botão Salvar, em seguida o arquivo com os dados da memória é baixado.

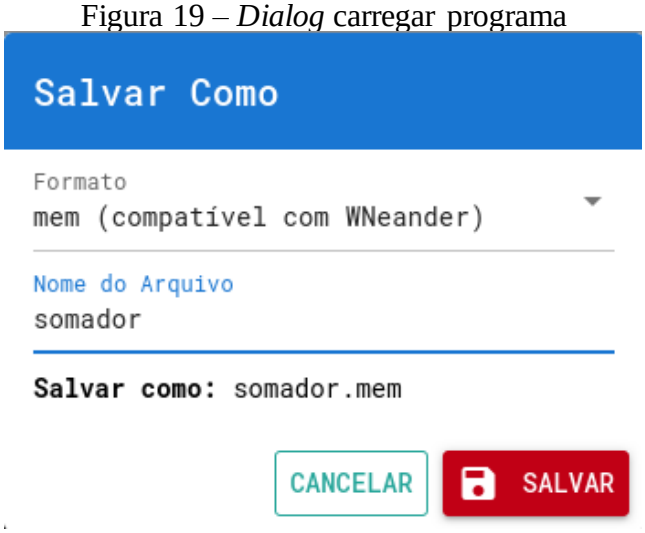
Figura 18 – *Dialog* salvar programa



Fonte: Autoria Própria

A Figura 19 apresenta o *dialog* responsável por carregar um arquivo com um programa, o usuário pode escolher ou arrastar o arquivo desejado para o campo Arquivo, e em seguida clicar em Carregar, após isso os dados da memória RAM são definidos por meio do arquivo selecionado.

Figura 19 – *Dialog* carregar programa

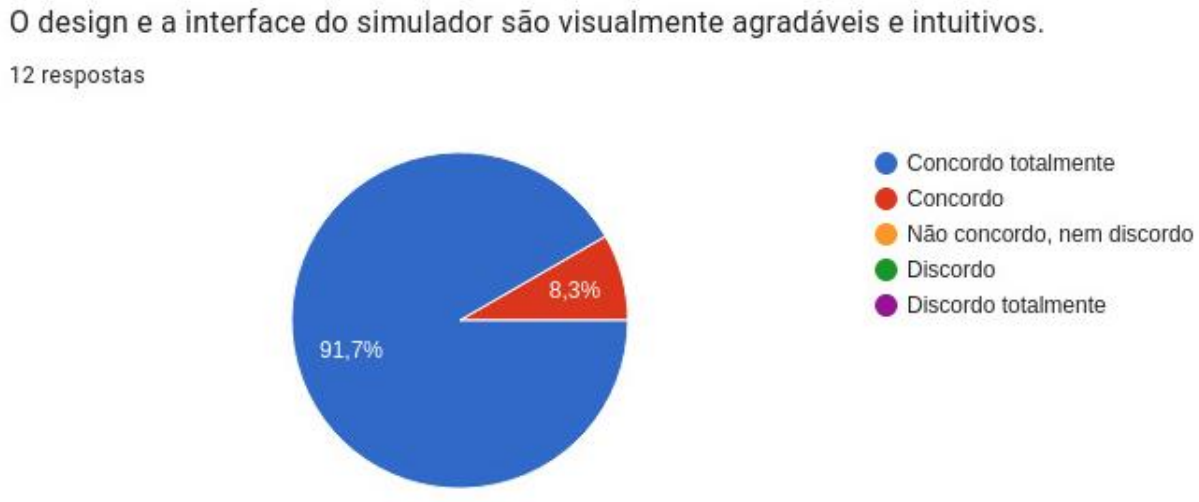


Fonte: Autoria Própria

A pesquisa de validação contou com a participação de 12 alunos do curso de Engenharia da Computação da Universidade de Ribeirão Preto - UNAERP, com as respostas obtidas é possível estimar, de forma geral e simplificada, se o simulador desenvolvido foi bem aceito pelos estudantes.

A Figura 20 ilustra as respostas obtidas da Questão 1 "O *design* e a interface do simulador são visualmente agradáveis e intuitivos", obteve 91.7% das respostas como “Concordo totalmente”, e 8.3% das respostas como “Concordo”. Esta questão está vinculada com o aspecto de usabilidade, portanto, dadas as respostas, o *design* e a interface cumpriram com a expectativa de proporcionar telas visualmente agradáveis.

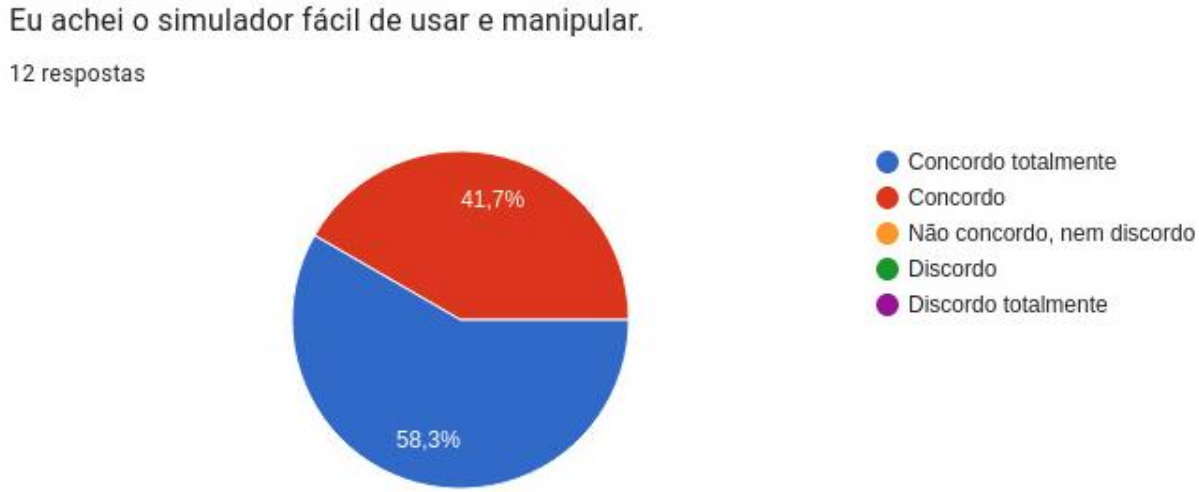
Figura 20 – Questão 1 da Pesquisa de Validação



Fonte: Autoria Própria

A Figura 21 ilustra as respostas obtidas da Questão 2 "Eu achei o simulador fácil de usar e manipular", obteve 58.3% das respostas como “Concordo totalmente”, e 41.7% das respostas como “Concordo”. Esta questão está associada com a particularidade de usabilidade, portanto, dadas as respostas, o simulador apresenta mecanismos de uso de forma intuitiva com facilidade de utilização.

Figura 21 – Questão 2 da Pesquisa de Validação



Fonte: Autoria Própria

A Figura 22 ilustra as respostas obtidas da Questão 3 "O simulador responde rapidamente e sem falhas", obteve 75% das respostas como “Concordo totalmente”, 16.7% das respostas como “Concordo”, e 8.3% das respostas como "Não concordo, nem discordo". Esta questão faz referência ao desempenho e execução das funcionalidades do simulador, portanto, dadas as respostas, o simulador demonstra uma performance consistente e eficiente.

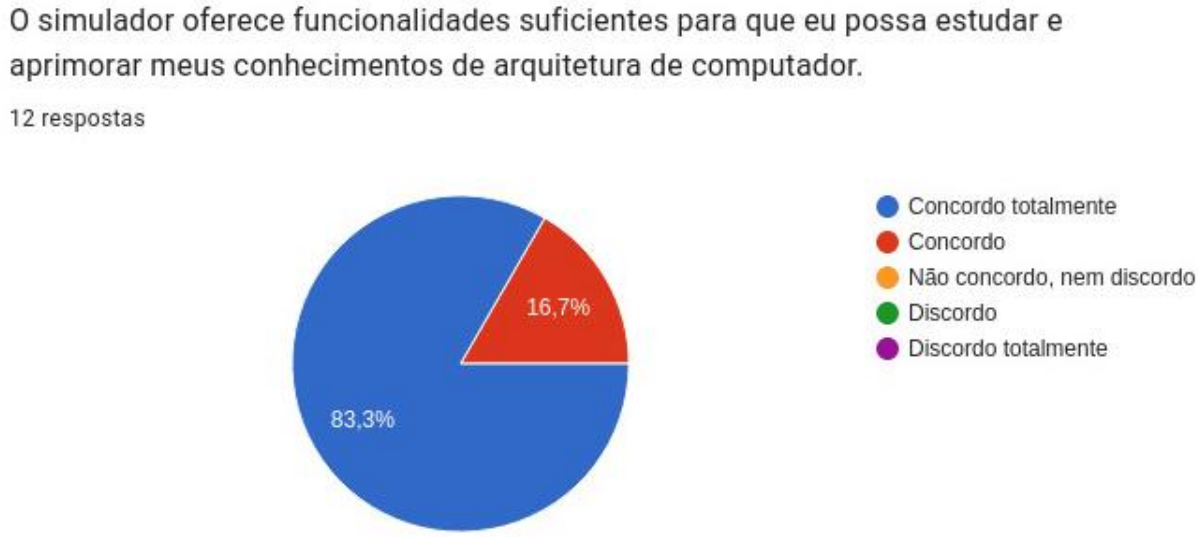
Figura 22 – Questão 3 da Pesquisa de Validação

Fonte: Autoria Própria

A Figura 23 ilustra as respostas obtidas da Questão 4 "O simulador oferece funcionalidades suficientes para que eu possa estudar e aprimorar meus conhecimentos de arquitetura de computador", obteve 83.3% das respostas como “Concordo totalmente”, e 16.7% das respostas como “Concordo”. Esta questão está relacionada com a eficiência com

que o simulador colabora com o estudo dos conceitos de arquitetura de computador, portanto, dadas as respostas, o simulador definitivamente auxilia os usuários em seus aprendizados.

Figura 23 – Questão 4 da Pesquisa de Validação



Fonte: Autoria Própria

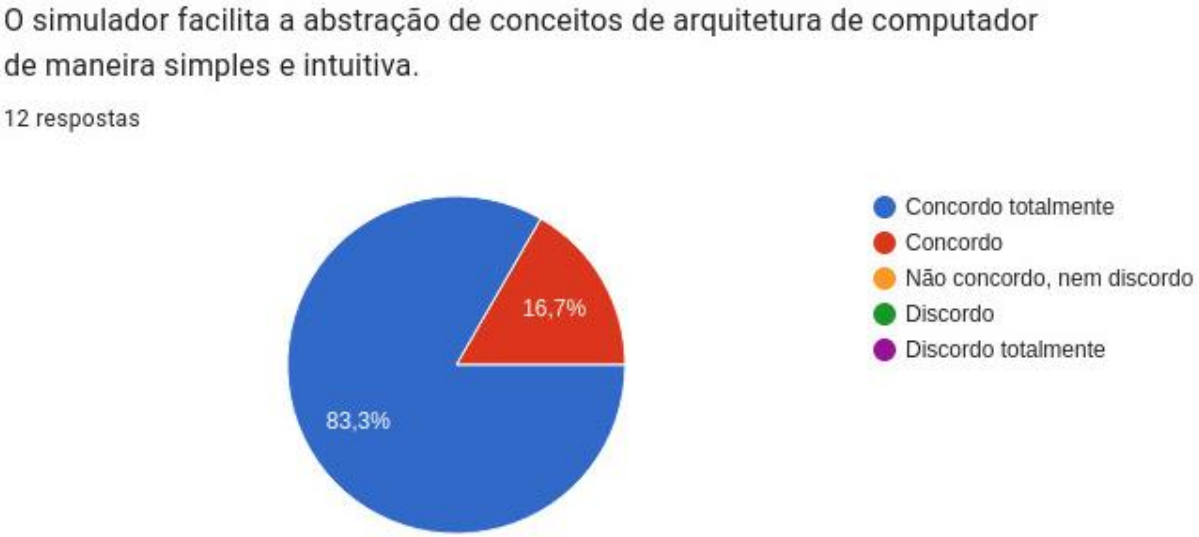
A Figura 24 ilustra as respostas obtidas da Questão 5 "Eu considero o simulador apresentado como uma nova versão mais completa do simulador original Neander", obteve 83.3% das respostas como “Concordo totalmente”, e 16.7% das respostas como “Concordo”. Está questão compara, de forma geral, o simulador desenvolvido com o simulador original Neander, desta forma o simulador desenvolvido deve possuir as funcionalidades já conhecidas, bem como novos recursos para agregar o processo de simulação e apresentação. Dadas as respostas, o simulador desenvolvido, portanto, atingiu a expectativa de se apresentar como um sucessor do simulador original Neander.

Figura 24 – Questão 5 da Pesquisa de Validação

Fonte: Autoria Própria

A Figura 25 ilustra as respostas obtidas da Questão 6 "O simulador facilita a abstração de conceitos de arquitetura de computador de maneira simples e intuitiva", obteve 83.3% das respostas como “Concordo totalmente”, e 16.7% das respostas como “Concordo”. Está questão está ligada a eficiência com que o simulador facilita o aprendizado do funcionamento de uma arquitetura de computador, portanto, dadas as respostas o simulador auxilia de maneira considerável o estudo dos conceitos associados.

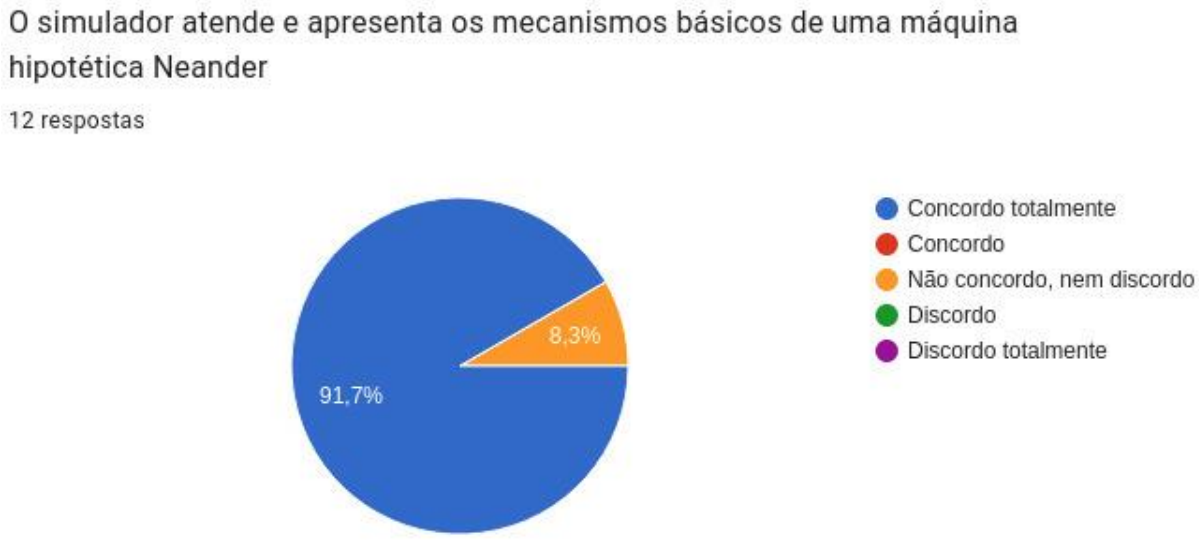
Figura 25 – Questão 6 da Pesquisa de Validação



Fonte: Autoria Própria

A Figura 26 ilustra as respostas obtidas da Questão 7 "O simulador atende e apresenta os mecanismos básicos de uma máquina hipotética Neander", obteve 91.7% das respostas como “Concordo totalmente”, e 8.3% das respostas como “Não concordo, nem discordo”. Está questão está relacionada as funcionalidades apresentadas no simulador referente ao computador Neander, portanto, dadas as respostas, o simulador desenvolvido evidenciou as operações e informações necessárias para atender os requisitos de um simulador da máquina hipotética Neander.

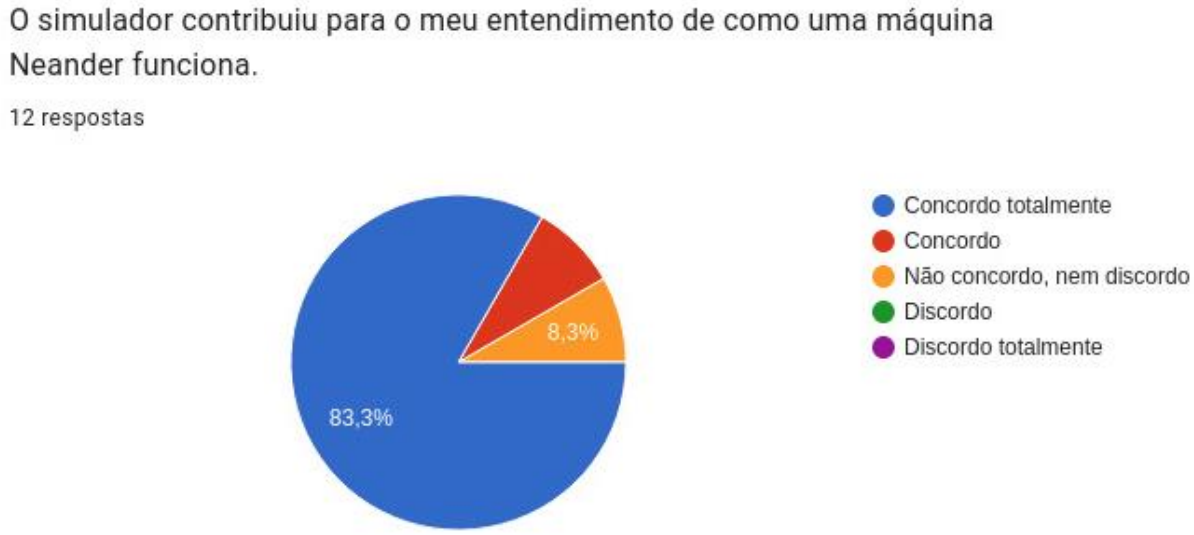
Figura 26 – Questão 7 da Pesquisa de Validação



Fonte: Autoria Própria

A Figura 27 ilustra as respostas obtidas da Questão 8 "O simulador contribuiu para o meu entendimento de como uma máquina Neander funciona", obteve 83.3% das respostas como “Concordo totalmente”, 8.3% das respostas como “Concordo”, e 8.3% das respostas como "Não concordo, nem discordo". Semelhante à questão 6, está questão tem o intuito de validar a eficiência com que o simulador contribui para os estudos de arquitetura de computador, portanto, dadas as respostas, o simulador desenvolvido atingiu as expectativas na contribuição do aprendizado.

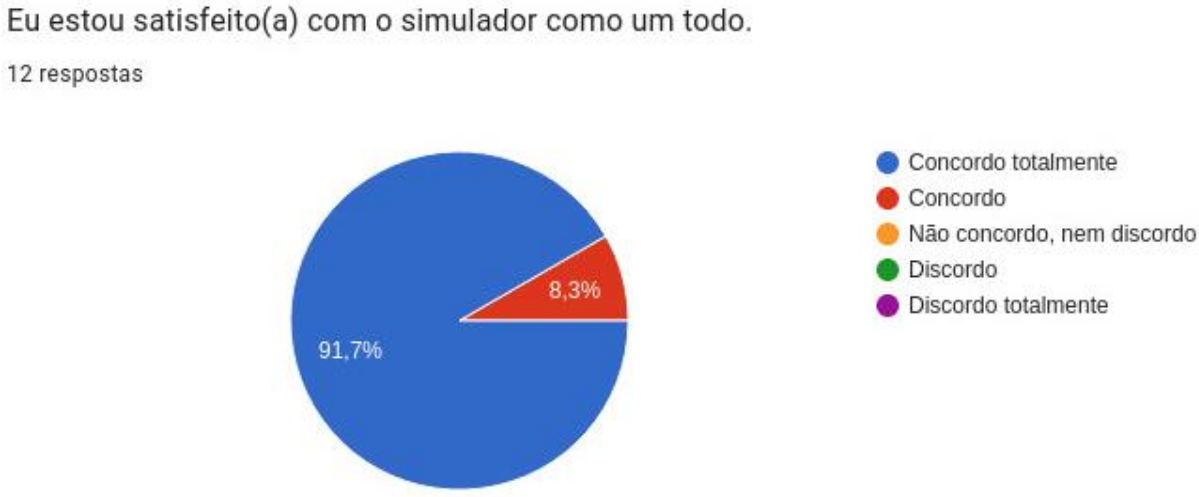
Figura 27 – Questão 8 da Pesquisa de Validação



Fonte: Autoria Própria

A Figura 28 ilustra as respostas obtidas da Questão 9 "Eu estou satisfeito(a) com o simulador como um todo", obteve 91.7% das respostas como “Concordo totalmente”, e 8.3% das respostas como “Concordo”. Está questão avalia o simulador de maneira geral, portanto, dadas as respostas, o simulador desenvolvido foi bem-sucedido em proporcionar satisfação de uso.

Figura 28 – Questão 9 da Pesquisa de Validação

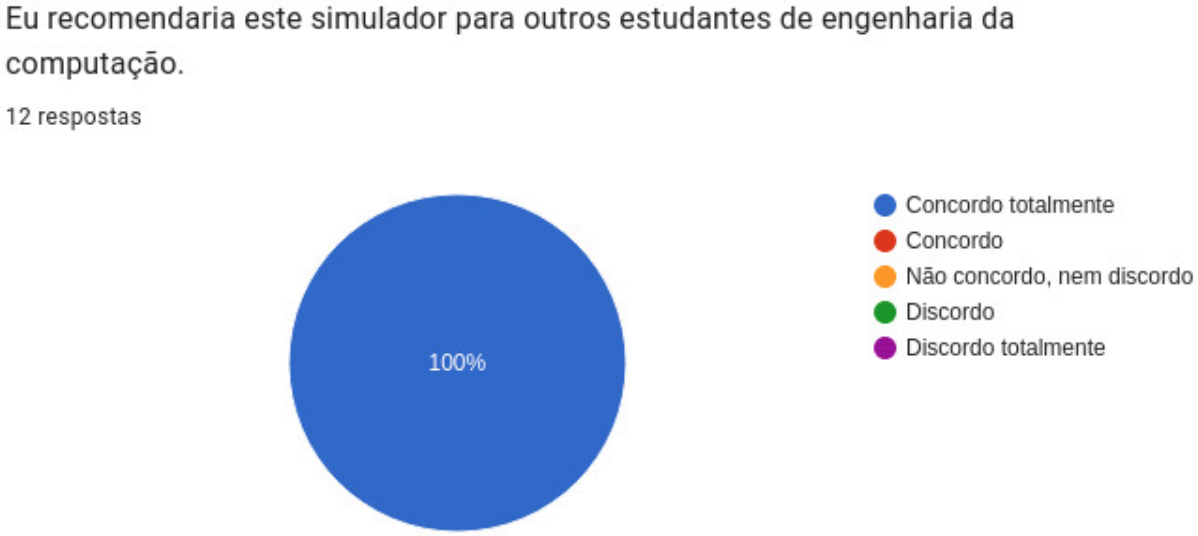


Fonte: Autoria Própria

A Figura 29 ilustra as respostas obtidas da Questão 10 "Eu recomendaria este simulador para outros estudantes de engenharia da computação", obteve 100% das respostas como “Concordo totalmente”. Semelhante à questão 9, está questão tem o objetivo de avaliar

a aceitação e satisfação do simulador de forma geral, portanto, dadas as respostas, o simulador foi bem aceito pelos participantes e seria recomendado para outros estudantes.

Figura 29 – Questão 10 da Pesquisa de Validação



Fonte: Autoria Própria

5 CONCLUSÃO

O objetivo de desenvolver um simulador didático da máquina hipotética Neander foi satisfatoriamente atingido dado o resultado do produto final, o *software* inclui todas as funcionalidades já existentes no simulador Neander original e faz se o complemento de novos recursos marcantes como a diagramação em tempo de execução e um módulo de *clock* ajustável, bem como a habilidade de executar a máquina pulso a pulso, diferente do original que se limita a uma execução apenas instrução a instrução. O simulador desenvolvido é extenso e rico em detalhes comparado ao produto original que omite muitas das informações, como os registradores que constituem a arquitetura e o barramento da unidade de controle.

Por meio de um desenvolvimento feito com tecnologias modernas e atenção aos detalhes a interface está responsiva para telas de dispositivos *desktop* e *mobile*, além disso cada componente recebeu o devido cuidado para que se tenha uma experiência interativa e imersiva, na qual traz o uso de recursos modernos que a tecnologia *web* tem a oferecer.

A pesquisa de validação realizada com os alunos do curso de Engenharia da Computação da Universidade de Ribeirão Preto – UNAERP obteve resultados extremamente positivos perante a aceitação e eficiência do simulador desenvolvido.

O *software* possui total compatibilidade com o simulador original na importação e exportação de arquivos de código fonte a fim de facilitar uma migração para o novo produto sem haver perdas ou retrabalhos.

O foco do simulador foi trabalhar com a arquitetura Neander, entretanto o seu desenvolvimento foi estrategicamente elaborado para permitir futuras integrações com outras arquiteturas, como por exemplo o SAP-1 ou o SAP-2, portanto este trabalho tem o potencial de ser estendido como um simulador de arquitetura de computador que comtempla vários modelos de máquinas.

REFERÊNCIAS

CAGLIARI, Bruna C; BUTZEN, Paulo F; BRUM, Raphael M. 20th Microelectronics Student Forum. **Towards a Nonvolatile Implementation of Neander, an Accumulator Based 8-bit Processor**. Porto Alegre, RS, agosto/2020.

COSTA, E.; SILVA, G. P. **Um Simulador Didático para o Ensino de Arquitetura de Computadores e Internet das Coisas**. International Journal of Computer Architecture Education, v. 7, n. 1, p. 29–38, 1 dez. 2018.

FLANAGAN, David. **JavaScript: The Definitive Guide**. 6ª ed. O'Reilly, 2011.

FULTON, Steve; FULTON, Jeff. **HTML5 Canvas**. [s.l.]: "O'Reilly Media, Inc.", 2013.

GARBIN, Fernanda Gobbi de Boer; KAMPFF, Adriana Justin Cerveira. COBENGE. **Uso da simulação para o ensino de engenharia: aplicações em cursos de graduação brasileiros**. Evento Online, setembro/2021.

JOSHI, A. et al. **Likert Scale: Explored and Explained**. British Journal of Applied Science & Technology, v. 7, n. 4, p. 396–403, 2015.

MALVINO, Albert Paul; BROWN, Jerald. **Digital Computer Electronics**. 3ª ed. New york: GLENCOE McGraw-Hill, 1993.

NETO. Affonso Dick; **Simuladores em plataforma web de máquinas hipotéticas para estudo Arquitetura de computadores**. Universidade Federal do Rio Grande do Sul, Porto Alegre, 2021.

RUDOLFO, M. et al. **NeanderSIM: Simulador Gráfico de Apoio ao Ensino de Arquitetura de Computadores**. XXXIV Congresso da Sociedade Brasileira de Computação, p. 1647–1656, 28 jul. 2014.

SCHNEIDER, Felipe R. et al. **Design of a 4-bit Processor for Evaluating of the E/D nMOS Technology from CCS/UNICAMP**. Instituto de Informática UFRGS, Porto Alegre, RS, 2004.

SILVA, Gabriel P; BORGES, José Antonio S. **NeanderWin – Um Simulador Didático para uma Arquitetura do Tipo Acumulador**. Instituto de Computação da Universidade Federal do Rio de Janeiro, Rio Janeiro, 2006.

TANENBAUM, Andrew S; AUSTIN, Todd. **Organização Estruturada de Computadores**. 6ª ed. São Paulo: Pearson Education, 2013.

WEBER, Raul Fernando. **Fundamentos de arquitetura de computadores**. 4ª ed. Bookman, 2012.

WEBER, Raul Fernando. **Organização Neander**. São Carlos. ICMC. 19 ago. 2010. Apresentação de Power Point. 28 slides. Disponível em: http://wiki.icmc.usp.br/images/5/58/SSC0610-Aula03s-Neander.pdf. Acesso em 23 out. 2024.

APÊNDICE I

Pesquisa de Validação do Projeto TCC Simulador Computador Neander Web e Mobile

Pesquisa realizada com alunos do curso de engenharia da computação a fim de obter resultados de validação do projeto desenvolvido.

felipe.contin@sou.unaerp.edu.br Mudar de conta

* Indica uma pergunta obrigatória

E-mail *

Seu e-mail

O design e a interface do simulador são visualmente agradáveis e intuitivos. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

Eu achei o simulador fácil de usar e manipular. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

O simulador responde rapidamente e sem falhas. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

O simulador oferece funcionalidades suficientes para que eu possa estudar e aprimorar meus conhecimentos de arquitetura de computador. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

Eu considero o simulador apresentado como uma nova versão mais completa do simulador original Neander. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

O simulador facilita a abstração de conceitos de arquitetura de computador de maneira simples e intuitiva. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

O simulador atende e apresenta os mecanismos básicos de uma máquina hipotética Neander *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

O simulador contribuiu para o meu entendimento de como uma máquina Neander funciona. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

Eu estou satisfeito(a) com o simulador como um todo. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

Eu recomendaria este simulador para outros estudantes de engenharia da computação. *

- ◯ Concordo totalmente
- ◯ Concordo
- ◯ Não concordo, nem discordo
- ◯ Discordo
- ◯ Discordo totalmente

Você gostaria de deixar uma opinião ou sugestão? (Opcional)

Sua resposta

Enviar Limpar formulário

Nunca envie senhas pelo Formulários Google.

Este formulário foi criado em Universidade de Ribeirão Preto. Denunciar abuso

Google Formulários