UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

AFFONSO DICK NETO

Simuladores em plataforma web de máquinas hipotéticas para estudo Arquitetura de computadores.

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Sérgio Luis Cechin

Porto Alegre
2021

“Os programas são feitos para serem lidos por humanos
e apenas incidentalmente para serem executados por computadores.”
— Donald Knuth

AGRADECIMENTOS

Agradeço a minha mãe que sempre me apoiou nas dificuldades enfrentadas. Sem seu esforço este trabalho não seria possível.

A esta universidade, seu corpo docente, direção e administração que me oportunizaram as condições necessárias para que eu alcançasse meus objetivos.

Aos programas governamentais de incentivo a educação e cultura, principalmente o Ciências sem Fronteiras. Sem eles, inúmeras oportunidades e experiências estariam fora do meu alcance.

E por fim, a todos que contribuíram para a realização deste trabalho, seja de forma direta ou indireta, fica registrado aqui, o meu muito obrigado!

RESUMO

Ferramentas de ensino auxiliam o alunos a aprenderem e fixarem conhecimentos complexos de forma prática. As disciplinas de arquitetura e organização de computadores do Instituto de Informática da UFRGS utiliza máquinas hipotéticas para ensinar de maneira gradual o funcionamento de computadores modernos. São elas Neander, Ahmes, Ramses e Cesar. Para mostrar o seu funcionamento de maneira prática simuladores dessas arquiteturas foram criados para o uso dos alunos .

Estes simuladores são suportados em apenas um sistema operacional e são implementados em uma linguagem caindo em desuso com pouco suporte ao passar dos anos. Para solucionar os dois problemas é sugerida a suas reimplementação na plataforma web, devido a suas grande popularidade e disponibilidade em múltiplos ambientes.

Este trabalho apresenta uma implementação destes simuladores em plataforma web. Além da descrição comportamental destas máquinas hipotéticas, detalhes de implementação deste projeto são mostrados que envolve uma pilha tecnológica multidisciplinar que são elas HTML, CSS, JS, Rust e WebAssembly. Também são apresentados dois conjuntos de testes usados para a verificação e validação dos simuladores e comparação de desempenho com as soluções existentes.

Palavras-chave: Máquina hipotéticas. Simuladores. Arquitetura de computadores. WebAssembly. Rust.

Simuladores de máquinas hipotéticas para a disciplina de Arquitetura e organização de computadores.

ABSTRACT

Teaching tools help students to learn and firm complex knowledge in a practical way. The disciplines of architecture and computer organization at the UFRGS Institute of Informatics use hypothetical machines to gradually teach the operation of modern computers. They are Neander, Ahmes, Ramses, and Cesar. To show how it works in a practical way, simulators of these architectures were created for the use of students.

These simulators are supported on only one operating system and are implemented in a language falling out of favor with little support over the years. To solve both problems, it is suggested to reimplement it on the web platform, due to its great popularity and availability in multiple environments.

This work presents an implementation of these simulators on a web platform. In addition to the behavioral description of these hypothetical machines, details of the implementation of this project are shown, which involves a multidisciplinary technological stack, which are HTML, CSS, JS, Rust and WebAssembly. Also presented are two sets of tests used to verify and validate the simulators and compare performance with existing solutions.

Keywords: Hypotetical machines. Simulators. Computer architecture. WebAssembly. Rust. .

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CSS	Cascating Stylers Sheet
FPGA	Field Programable Gate Arrays
HTML	Hyper Text Markup Language
IPS	Instruções por segundo
JS	Javascript
NPM	Node Package Manager
ULA	Unidade de Logica e Aritmética
SIMD	Single Instruction Multiple Data
W3C	World Wide Web Consortium
WASM	Web Assembly

SUMÁRIO

1 INTRODUÇÃO

Nas disciplinas de arquitetura e organização de máquinas, ministradas para os cursos de Ciências e Engenharia da Computação da UFRGS, os alunos são introduzidos a conceitos chave para o entendimento de computadores modernos. Com a evolução dos computadores ao passar dos anos tais conceitos são bastante complexos para serem ensinados imediatamente de forma prática. Para facilitar a curva de aprendizado foram arquitetadas máquinas mais simples, com um conjunto de instruções reduzidos e componentes simples que incrementalmente são adicionados novas funcionalidades e capacidades a elas. Isso permite o ensino destes conceitos de forma linear sem que o aluno fique sobrecarregado com o imenso conjunto de instruções e complexidade de processadores modernos.

Foram criadas um total de quatro máquinas hipotéticas que hoje são utilizadas como ferramenta de ensino, são elas: Neander, Ahmes, Ramses, Cesar (WEBER, 2009). Elas implementam um conjunto de instruções de máquina básicas, sendo que Neander $\subset$ Ahmes e Neander $\subset$ Ramses. Elas são apresentadas para os alunos na seguinte ordem de complexidade: Neander, Ahmes, Ramses e Cesar.

Por serem máquinas hipotéticas não existe uma implementação em hardware prontamente disponível para uso. Para isso simuladores foram desenvolvidos para cada uma delas. Simuladores são softwares que imitam o comportamento de uma máquina de forma artificial. Estes são fornecidos na disciplina aos alunos para que seja possível um aprendizado pratico com elas. Esse simulador é feito em Pascal utilizando o *framework* Delphi para a criação da interface e atualmente está disponível apenas para o sistema operacional Windows.

Outras implementações de algumas dessas máquinas existem como algumas implementações feita em FPGA das máquinas Neander e Ahmes (CAMPOS, 2016) e Cesar (ORTH, 2010), porém acesso a placas FPGA é restrito para muitos e a configuração de uma requer conhecimentos avançados. Outra implementação é mantida pelo PET da UFGRS (PROJETO. . . , 2021), porém sua compilação deve ser feita para cada plataforma separadamente. Ou uma implementação simples em JS do Neander e Ahmes (ARAUJO, 2021). Além da UFRGS, outras institutos usam a essas arquiteturas para ensino, ICMC-USP e UFRJ, e possuem extensões do simulador Neander com simulador (SILVA; BORGES, 2016).

1.1 Objetivo e Motivação

Para tornar essas ferramentas de aprendizado modernas e de fácil uso é necessário tornar os simuladores mais acessíveis para os alunos a fim de encorajar o seu uso. A internet é atualmente a plataforma mais acessada e usada mundialmente e isso se deve em grande parte pela homogeneidade de acesso e desenvolvimento nela desde o avanço da Web 2.0. O acesso uniforme de funcionalidades em praticamente qualquer máquinas, *desktop* ou *mobile*, torna a plataforma web muito atraente para se desenvolver nela com um baixo esforço e grandes benefícios. Esse acesso mais pervasivo e ubíquo da plataforma web é um excelente candidato para um aplicativo com o simuladores dessas máquinas.

Para facilitar o aprendizado e modernizá-los, é proposta a implementação de simuladores destas máquinas em uma plataforma web onde seja possível os alunos exercitarem e experimentarem com as máquinas hipotéticas. Esse projeto tem como objetivo aumentar o alcance e usabilidade dessas ferramentas sem perder o desempenho alcançado em um ambiente *desktop*. Para isso são usadas diferentes tecnologias que trabalham juntas para alcançar esse objetivo.

1.2 Estrutura

A estrutura deste trabalho é descrita a seguir.

O capítulo 2 descreve as máquinas hipotéticas a serem implementadas: Neander, Ahmes, Ramses, e Cesar. Uma breve explicação de alguns conceitos comuns entre todas máquinas é dado inicialmente. Para cada uma são apresentadas as principais características e o conjunto de instruções.

A seguir no capítulo 3 são apresentadas as tecnologias usadas na implementação dos simuladores. Detalhando algumas de suas características e explicando o motivo pela qual foram escolhidas. Elas são divididas em tecnologias usadas para implementar a interface web, usada para interação com o usuário, e para implementar os modelos dos simuladores.

O capítulo 4 descreve os detalhes de implementação dos simuladores. Explicando sua organização e decisões de arquitetura de código para interface e modelo. Ao fim desse capítulo os problemas relacionados ao desempenho enfrentados são apresentados, analisados, e a solução encontrada é mostrada.

Por fim, no capítulo 5 relata os objetivos alcançados e problemas enfrentados no desenvolvimento deste trabalho. Também são apresentadas uma série de possíveis melhorias para trabalhos futuros.

2 MÁQUINAS HIPOTÉTICAS

Nesse capítulo será explicada a arquitetura das máquinas hipotéticas e suas diferenças e similaridades, como é descrito no livro (WEBER, 2009). Três delas são de 8 bits, Neander, Ahmes e Ramses, e um de 16 bits, Cesar. A arquitetura do Neander serve de base para o Ahmes e o Ramses, sendo assim, programas feitos para o Neander podem ser executados sem problemas nos dois. Cesar é o processador de 16 bits, sendo assim, não compatível com as máquinas 8 bits.

Todos processadores podem possuem dois módulos centrais básicos: Registradores, Unidade de Lógica Aritmética (ULA) e bloco de memória.

A representação numérica usada na ULA de todos processadores é em complemento de 2, ou seja, a inversão do sinal de um número é feita invertendo os bits e somando 1 ao resultado. Todas operações ativam sinais de condição, de acordo com a última operação feita, que podem ser usados para controlar o fluxo de um programa.

O bloco de memória possui o endereçamento por bytes, ou seja, o limite de tamanho da memória é de definido pela sua capacidade de endereçar elas. Ficando 256 (2^8) bytes para os processadores de 8 bits e 65536 (2^{16}) bytes para o processador de 16 bits.

A execução de instruções em todas as máquinas segue os seguintes passos:

1. Busca a próxima instrução na memória, indicado pelo apontador de memória.
2. Decodifica a instrução.
3. Executa a instrução.

Para a criação de programas um montador, Daedalus, é usado para que um arquivo de código *Assembly* gere o estado de memória, arquivo **.mem**, que execute o programa. Exemplo de código pode ser visto na Figura 2.1).

Figura 2.1: Exemplo de código assembly para o Neander

```
1 LDA A ; carrega valor de A no acumulador
2 ADD A ; soma valor de A ao acumulador
3 STA B ; guarda valor no acumulador em B
4 HLT   ; finaliza programa
5
6 A: 2
7 B: 0
```

Tabela 2.1: Conjunto de instruções do Neander

Código binário	*Mnemônico*		*Descrição*
0000 xxxx	NOP		Nenhuma operação
0001 xxxx	STA	*end*	armazena acumulador - (store)
0010 xxxx	LDA	*end*	carrega acumulador com valor em memória - (load)
0011 xxxx	ADD	*end*	soma
0100 xxxx	OR	*end*	'ou' logico
0101 xxxx	AND	*end*	'e' logico
0110 xxxx	NOT		inverte (complementa) acumulador
1000 xxxx	JMP	*end*	desvio incondicional - (jump)
1001 xxxx	JN	*end*	desvio condicional - (jump on negative)
1010 xxxx	JZ	*end*	desvio condicional - (jump on zero)
1111 xxxx	HLT		término da execução

Fonte: (WEBER, 2009, p.61)

2.1 Neander

Características principais do processador Neander são:

- Largura de dados e endereços de 8 bits
- Possui 11 instruções
- Um registrador de 8 bits para uso geral, acumulador (AC)
- Um registrador de programa (PC) de 8 bits
- Dois sinais de condição: negativo (N) e zero (Z)
- Modo de endereçamento direto

Todas instruções pode ser representadas por 1 ou 2 bytes, o identificador de instrução e o operador de instrução (*end*). Apenas os 4 bits mais representativos do identificador são usados no Neander e os 4 bits restantes são ignorados, por padrão o montador completa eles com '0'. Todas operações válidas estão na Tabela 2.1. Bits não relevantes na instrução são representados com 'x' nela.

O resultado de todas operações aritméticas (AND, OR, NOT, e ADD) é guardado no registrador AC. O conteúdo de AC é sempre usado como primeiro operador. Algumas instruções usam um segundo operando *end* que está no byte seguinte a instrução.

Tabela 2.2: Conjunto de instruções estendidas do Ahmes

Código binário	*Mnemônico*		*Descrição*
0111 xxxx	SUB	end	subtração
1001 00xx	JN	end	desvio condicional - herdado - (jump if negative)
1001 01xx	JP	end	desvio condicional - herdado - (jump if positive)
1001 10xx	JV	end	desvio condicional - (jump if overflow)
1001 11xx	JN	end	desvio condicional - (jump if not overflow)
1010 00xx	JZ	end	desvio condicional - (jump if zero)
1010 01xx	JNZ	end	desvio condicional - (jump if not zero)
1011 00xx	JC	end	desvio condicional - (jump if carry)
1011 01xx	JNC	end	desvio condicional - (jump if not carry)
1011 10xx	JB	end	desvio condicional - (jump if borrow)
1011 11xx	JNB	end	desvio condicional - (jump if not borrow)
1110 xx00	SHR	end	deslocamento para a direita
1110 xx01	SHL	end	deslocamento para a esquerda
1110 xx00	ASR	end	rotação para a direita
1110 xx11	ASL	end	rotação para a esquerda

Fonte: (WEBER, 2009, p.69)

2.2 Ahmes

O conjunto de instruções da máquina Ahmes é uma extensão do conjunto de instruções do Neander, as características adicionadas incluem:

- 13 instruções adicionadas, total de 24
- 3 novo sinais de condição, total de 5
  - Estouro de representação, ou overflow (V)
  - vai-um, ou carry (C)
  - empresta-um, ou borrow (B)

Todas instruções do Neander são válidas no Ahmes, ou seja, o código de instrução é o mesmo para os dois. Porém algumas instruções são representadas com mais do que 4 bits, particularmente duas herdadas do Neander, são representadas por mais bits no Ahmes, JN e JZ. Como o código gerado pelo montador para bits irrelevantes é sempre 0, isso não afeta o funcionamento de programas Neander no Ahmes. Na Tabela 2.2 de instruções é possível ver isso.

Tabela 2.3: Codificador de registrador no Ramses

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

Fonte: (WEBER, 2009, p.193)

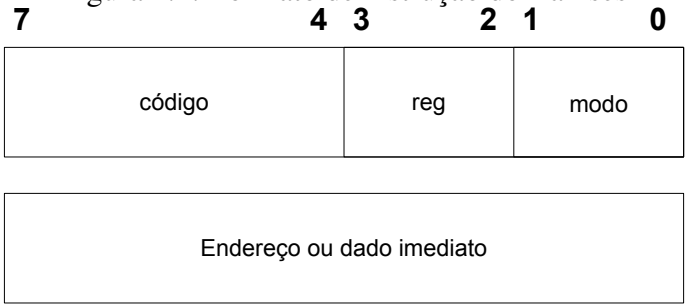
2.3 Ramses

Assim como o Ahmes, Ramses é uma extensão do Neander, porém não compatíveis com Ahmes. Suas principais características de arquitetura comparando com Neander são:

- 4 instruções adicionadas, total de 15
- 2 registradores de 8bits de uso geral, A e B
- 1 registrador de 8 bits de índice, X
- 3 modos de endereçamento, total de 4
- 1 novo sinal de condição, carry (C), total de 3

Com os novos registradores e modo de leitura, cada instrução precisa definir qual registrador e modo de endereçamento será usado. Para codificar cada um é necessário no mínimo 2 bits para cada, nas Tabelas 2.4 e 2.3 mostram a codificação de cada um. Os 4 bits menos significativos no identificador de instrução é usado para isso, o formato final das instruções pode ser visto na Figura 2.2.

Figura 2.2: Formato de instrução do Ramses



Fonte: (WEBER, 2009, p.191)

A Tabela 2.5 mostra todas instruções, *r* é usado para indicar o registrador a ser usado pela instrução.

A instrução de controle de sub-rotina (JSR), mostrado na Figura 2.3, funciona de forma simples. Ao executar JSR *sub* o endereço da instrução seguinte é guardada em

Tabela 2.4: Modos de endereçamento do Ramses

Código	*Modo*	*Mnemônico*	*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
00	imediato	#*end*	O operando segue a palavra de código da instrução
00	indexado	*end*,X	O deslocamento é somado ao registrador X para formar o endereço do operando. Deslocamento representado em complemento de 2

Fonte: (WEBER, 2009, p.191-193)

Tabela 2.5: Conjunto de instruções do Ramses

Código binário	*Mnemônico*	*Descrição*
0000 xxxx	NOP	Nenhuma operação
0001 xxxx	STR *r end*	armazena registrador - (store)
0010 xxxx	LDR *r end*	carrega registrador com valor em memória - (load)
0011 xxxx	ADD *r end*	soma
0100 xxxx	OR *r end*	'ou' logico
0101 xxxx	AND *r end*	'e' logico
0110 xxxx	NOT *r*	inverte (complementa) registrador
0111 xxxx	SUB *r end*	subtração
1000 xxxx	JMP *end*	desvio incondicional - (jump)
1001 xxxx	JN *end*	desvio condicional - (jump on negative)
1010 xxxx	JZ *end*	desvio condicional - (jump on zero)
1011 xxxx	JC *end*	desvio condicional - (jump on carry)
1100 xxxx	JSR *end*	desvio para sub-rotina
1101 xxxx	NEG *r*	troca sinal
1110 xxxx	SHR *r*	deslocamento para a direita
1111 xxxx	HLT	término da execução

Fonte: (WEBER, 2009, p.193)

sub e o apontador de programa é modificado para *sub*+1. Por esse motivo o primeiro byte de uma sub-rotina não deve conter nenhuma instrução relevante sendo recomendado usar um NOP por ocupar apenas um byte. Não há uma instrução de retorno de sub-rotina, ele deve ser feito manualmente através de um JMP indireto, JMP (*sub*). Com isso apenas um endereço de retorno pode ser guardado por vez visto que ele será sobrescrito na próxima chamada de sub-rotina, não permitindo que ela seja chamada de novo antes da mesma chegar ao fim sem que fluxo do programa seja afetado ou precauções extras sejam tomadas.

Figura 2.3: Exemplo de uso de sub-rotina no Ramses

```
LDR A #0
JSR add1
JSR add1
STR A b
HLT
b:0 ; 2 ao fim da execucao
add1: NOP
ADD A #1
JMP (sub1)
```

2.4 Cesar

- Largura de dados e endereço de 16 bits.
- 64 KB de memória.
- Uso de notação *big endian* para dados em memória.
- 8 registradores utilizáveis (R0 a R7) de 16 bits, sendo R6 usado como ponteiro de pilha (SP) e R7 como apontador de programa (PC).
- Quatro códigos de condição: negativo (N), zero (Z), overflow (V), carry (C).
- 41 instruções disponíveis, incluindo:
  - 15 instruções de desvio condicional.
  - 12 instruções aritméticas de um operando.
  - 6 instruções aritméticas de dois operandos.
  - 2 instruções de controle de sub-rotinas.
  - 1 instrução de laço.
- Visor digital de 36 caracteres.
- Acesso a teclado para entrada de dados

2.4.1 Modos de endereçamento

O Cesar possui oito modos de endereçamento, codificados em 3 bits. Na Tabela 2.6 contém a codificação e explicação de cada endereçamento. Note que os endereçamentos que incrementam ou decrementam os registradores o fazem por 2 já que a palavra do Cesar ocupa 2 bytes. Para desvios o modo de endereçamento Registrador é inválido, já que desvios recebem o *endereço de desvio* e não o *endereço do operando*, nesses casos o

desvio é interpretado como uma instrução NOP. Como a arquitetura Cesar permite acesso ao PC (R7) é possível fazer operações relativas a ele. Durante uma instrução o R7 estará apontando a palavra que segue a instrução é possível fazer o endereçamento imediato usado para definir um valor de uma constante diretamente no código. Também é possível fazer o endereçamento absoluto que possui a mesma funcionalidade do endereçamento direto das máquinas anteriores.

Tabela 2.6: Modos de endereçamento do Cesar

Código	*Nome*	*Mnem.*	*Descrição*
000	Registrador	Rx	Registrador contem o operando
001	Pós-incrementado	(Rx)+	Registrador contém o endereço do operando. Após o uso 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 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: (WEBER, 2009, p.223 e 234)

2.4.2 Instruções simples

As instruções do Cesar podem ocupar 1 ou 2 bytes de acordo com o seu tipo. Elas são divididas em dez tipos identificados pelos 4 bits mais significativos da instrução como visto na Tabela 2.7.

Para a instrução NOP apenas o 4 bit mais significativos importam, o restante é ignorado, como visto na Figura 2.4. O mesmo acorre para a instrução HLT.

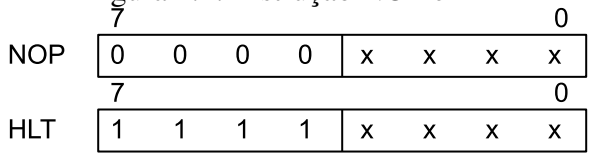
As instruções de controle de sinais (CCC e SCC) ocupam apenas um byte. Quais sinais de condição devem ser alterados pela instrução é indicado nos quatro bits menos significativos da instrução como mostrado na Figura 2.5, caso o bit correspondente

Tabela 2.7: Tipos de instruções

Código	*Tipo*
0000	Instrução NOP
0001 e 0010	Instruções sobre os códigos de condição
0011	Instruções de desvio condicional
0100	Instruções de desvio incondicional (JMP)
0101	Instruções de controle de laço (SOB)
0110	Instruções de desvio de sub-rotina (JSR)
0111	Instruções de retorno de sub-rotina (JSR)
1000	Instruções de um operando
1001 e 1110	Instruções de dois operandos
1111	Instrução de parada (HLT)

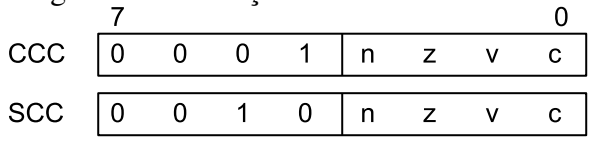
Fonte: (WEBER, 2009, p.193)

Figura 2.4: Instrução NOP e HLT



esteja ativo o sinal será alterado.

Figura 2.5: Instruções de controle de sinais



2.4.3 Controle de desvios

As instruções de desvio condicional ocupam dois bytes no formato visto na Figura 2.6. O segundo byte, representado por *dddddddd*, indica o deslocamento, complemento de dois, que será feito caso o desvio aconteça. O desvio é relativo a posição atual, ou seja, esse valor é somado ao PC para se ter o endereço final. Qual desvio será feito é indicado por *c* e poder ser visto na Tabela 2.8.

Figura 2.6: Instrução de desvio condicional

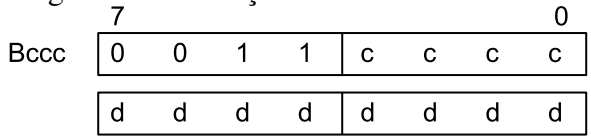


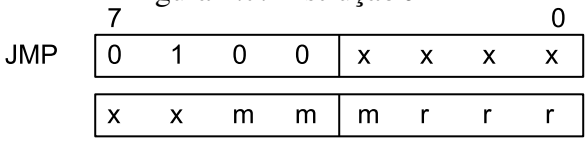
Tabela 2.8: Instruções de desvio condicional

cccc	mnemônico	condição de desvio
0000	BR (always)	sempre
0001	BNE (Not Equal)	Z=0
0010	BEQ (EQual)	Z=1
0011	BPL (PLus)	N=0
0100	BMI (MInus)	N=1
0101	BVC (oVerflow Clear)	V=0
0110	BVS (oVerflow Set)	V=1
0111	BCC (Carry Clear)	C=0
1000	BCS (Carry Set)	C=1
1001	BGE (Greater or Equal)	N=V
1010	BLT (Less Than)	N <> V
1011	BGT (Greater Than)	N=V and Z=0
1100	BLE (Less or Equal)	N<>V and Z=1
1101	BHI (HIgher)	C=0 and Z=0
1110	BLS (Lower or Same)	C=0 and Z=1

Fonte: (WEBER, 2009, p. 229)

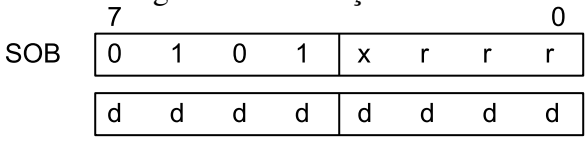
A instrução de desvio (JMP) ocupa dois bytes e tem o formato visto na Figura 2.4.3. O valor a ser carregado no PC é o endereço do operando calculado usando *mmm* como modo e *rrr* para o registrador. Caso o modo seja registrador (000) a instrução age como um NOP.

Figura 2.7: Instrução JMP



A instrução de controle de laço (SOB) tem o formato mostrado na Figura 2.8. O registrador indicado em *r* é decrementado, se o valor resultante for diferente de 0, o deslocamento indicado em *ddddddd* é *subtraído* de PC. Caso seja 0 o fluxo do programa continua para a próxima instrução. Note que pelo desvio ser apenas um byte, desvios condicionais têm o alcance de apenas -127 e +128 posições de memória em torno da instrução.

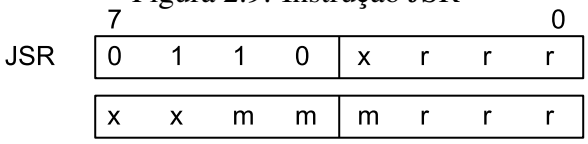
Figura 2.8: Instrução SOB



2.4.4 Sub-rotinas

Para iniciar um sub-rotina instrução JSR é usada, o formato dela pode ser visto na Figura 2.9.

Figura 2.9: Instrução JSR



Seu funcionamento é mais complexo do que o visto no Ramses, a sequência de operações a seguir é feita.

$$temporário \leftarrow endereço\ da\ subrotina$$
$$pilha \leftarrow registrador$$
$$registrador \leftarrow PC$$
$$PC \leftarrow temporário$$

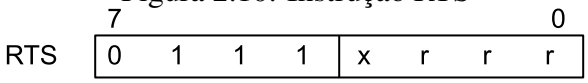
O endereço usado como pilha deve estar indicado em SP (R6), o registrador guardado nela é o primeiro indicado na instrução. O endereço da sub-rotina é dado pelo segundo registrador e modo indicado. Se o primeiro registrador usado é o PC essa sequência efetivamente guarda na pilha o endereço de retorno da sub-rotina, assim como em JMP o modo 0 é ignorado e a instrução é tratada como NOP.

Para o retorno da sub-rotina (RTS) segue o formato da Figura 2.10. Onde *rrr* é o registrador de retorno, para isso os seguintes passos são feito:

$$PC \leftarrow registrador$$
$$registrador \leftarrow topo\ da\ pilha$$

Se o registrador PC é usado, o topo da pilha é usado como endereço de retorno.

Figura 2.10: Instrução RTS



2.4.5 Instruções de um operando

Instruções de um operando ocupam dois bytes e seguem o formato na Figura 2.11. Os campos *mmm* e *rrr* são usados para calcular o operando, *cccc* é o identificador da ope-

ração. Na Tabela 2.9 estão todas operações de uma instrução, com a ação feita assim como quais sinais de condição são modificados. Os sinais de condição *not(t)* são usados para instruções que pode ocorrer *borrow*, invertendo o seu valor e armazenando em C. Os valores *lsb* e *msb* são o bits menos e mais significativos, respectivamente, do operando antes da execução da instrução. *xor* para o sinal overflow indica que este é carregado com o resultado da operção de 'ou exclusivo' entre os sinais de condição N e C, após os mesmos terem sido alterados pela instrução.

As operações SHR e SHL são respectivamente as operações bit a bit *shift right* e *shift left*. Enquanto ROR e ROL são rotações para a direta e esquerda, ASR e ASL são deslocamentos aritméticos onde o comportamento é similar a um *shift* quando para a esquerda, porém para a direita ao invés de 0 o resultado é completado com o bit mais significativo antes da operação. A operação TST somente testa o valor do operando, e atualiza o sinais de condição.

Figura 2.11: Instruções de um operando

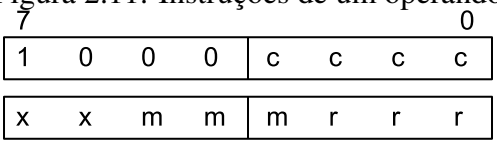


Tabela 2.9: Instruções de um operando

Código	*Instrução*	*Significado*	*N*	*Z*	*C*	*V*
0000	CLR	$op \leftarrow 0$	t	t	0	0
0001	NOT	$op \leftarrow NOTop$	t	t	1	0
0010	INC	$op \leftarrow op + 1$	t	t	t	t
0011	DEC	$op \leftarrow op - 1$	t	t	not(b)	t
0100	NEG	$op \leftarrow -op$	t	t	not(b)	t
0101	TST	$op \leftarrow op$	t	t	0	0
0110	ROR	$op \leftarrow SHR(C\&op)$	t	t	lsb	xor
0111	ROL	$op \leftarrow SHL(op\&C)$	t	t	msb	xor
1000	ASR	$op \leftarrow SHR(msb\&op)$	t	t	lsb	xor
1001	ASL	$op \leftarrow SHL(op\&0)$	t	t	msb	xor
1010	ADC	$op \leftarrow op + C$	t	t	t	t
1011	SBC	$op \leftarrow op - C$	t	t	t	t

Fonte: (WEBER, 2009, p. 232)

2.4.6 Instruções de dois operandos

Instruções de dois operandos usam todos os bits de dois bytes, o seu formato pode ser visto na Figura 2.12. Os dois operandos usados são referênciados como ori-

gem (*src*) e destino (*dst*) para facilitar o entendimento. Na Tabela 2.10 é mostrado as operações de dois operandos assim como os sinais de condição alterados pela instrução. Assim como nas instruções onde pode ocorrer *borrow*, o valor do é armazenado em C de forma inversa. A operação CMP somente altera os códigos de condição descartando o resultado da operação feita.

Figura 2.12: Instruções de dois operandos

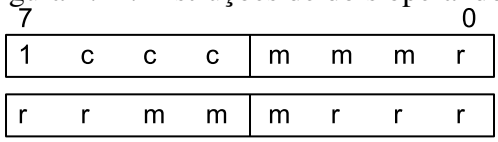


Tabela 2.10: Instruções de dois operandos

Código	*Instrução*	*Significado*	*N*	*Z*	*V*	*C*
001	MOV	$dst \leftarrow src$	t	t	0	-
010	ADD	$dst \leftarrow dst + src$	t	t	t	t
011	SUB	$dst \leftarrow dst - src$	t	t	t	not(b)
100	CMP	$src - dst$	t	t	t	not(b)
101	AND	$dst \leftarrow dst\ AND\ src$	t	t	0	-
110	OR	$dst \leftarrow dst\ OR\ src$	t	t	0	-

Fonte: (WEBER, 2009, p. 233)

2.4.7 Interface externa

A máquina Cesar possui suporte a dispositivos de entrada e saída de dados rudimentar. Para entrada é um teclado que possibilita a leitura de um caractere e teste se uma tecla foi digitada. A saída é feita em um visor alfanumérico de 36 posições, ele permite visualizar letras, dígitos, e alguns caracteres especiais.

A entrada e saída de dados é feita através de um mapeamento de memória das 38 últimas posições, faixa de endereçamento de 65498 a 65535. Nesta faixa de valores as instruções usam uma palavra do tamanho de 1 byte, ou seja, ao ler uma posição apenas ela é lida como um número de 8 bits e ao escrever apenas o byte menos significativo é usado. Por exemplo: escreve o número 65535 (FFFF em hexadecimal), apenas 255 (FF em hexadecimal) será escrito.

O conteúdo do visor é mapeado entre 65500 e 65535, cada byte sendo um caractere. É usada a codificação ASCII e apenas valores entre 35 e 126 são visíveis, o restante dos caracteres são mostrados como espaço vazio. O teclado usa a posição 65499 para sinalizar o último caractere digitado e a posição 65498 sinaliza o estado do teclado.

Ao digitar um caractere o byte de sinalização é mudado para 128 (80 em hexadecimal) e o caractere digitado é atualizado, caso a sinalização estiver ativa novos caracteres são ignorados. Dessa forma o programa sendo executado é obrigado a zerar o conteúdo do sinalizador para poder ler novos caracteres.

3 TECNOLOGIAS USADAS

Este capítulo mostra as tecnologias usadas para desenvolver os simuladores e o motivo de algumas dessas escolhas, assim como possíveis alternativas e o porque não foram escolhidas.

3.1 Plataforma web

Aplicações web modernas requerem cada vez mais recursos computacionais para dar ao usuário uma boa experiência, sem perder níveis de segurança alcançados ao longo dos anos. Para endereçar o problema, engenheiros dos quatro principais navegadores de mercado desenvolveram a linguagem de baixo nível WebAssembly (Wasm) (HAAS et al., 2017). Wasm descreve uma abstração sobre uma máquina de pilha, ou seja, ela é independente de linguagem, hardware, e plataforma. Programas em uma linguagem arbitrária podem ser compilados usando como alvo Wasm. Ou seja, tais programas serão executados na maquina cliente acessando a página, com velocidade superior ao Javascript, que é o padrão web para processamentos arbítrios.

Isso permite que processamentos custosos possam ser feitos diretamente no navegador de forma eficiente e independente de sistema operacional. Ao usar essa plataforma para implementar os simuladores todo o processamento necessário para simulação será feito pelo navegador, aumentando a eficiência e responsividade dos simuladores. Isto também reduz a complexidade necessária para o projeto, pois não há a necessidade de comunicação entre outros agente para o processamento, comum em páginas web que requerem processamento mais avançado fazendo o uso de servidores, já que o processamento necessário no servidor é mínimo.

Wasm surgiu por meados de 2017 como um aprimoramento de *asm.js* (HERMAN; WAGNER; ZAKAI, 2014), um sub-set de instruções Javascript (JS) (SEVERANCE, 2012) que aumentava o desempenho, evoluindo para resolver o problemas de desempenho restritos por conta do interpretador JS. Apesar de recente, está tendo uma ascensão rápida pelas contribuições de gigantes da indústria (Mozilla, Microsoft, Google, Apple, entre outros) e em 2019 passou a ser um dos padrões recomendados para navegadores pela W3C (World Wide Web Consortium). Por isso Wasm já é suportado nos principais navegadores do mercado, Chrome, Firefox, Edge, Safari.

Sua compatibilidade com navegadores sem suporte pode ser superada usando

Polyfill, funções que detectam o suporte do navegador e simulam elas assim trocando velocidade por compatibilidade nesses ambientes. Tornado a plataforma compatível com a grande maioria dos navegadores do mercado, mesmo sem suporte ao formato.

Outra vantagem do Wasm que será explorada nesse trabalho é a possibilidade de programas já existentes poderem ser compilados em Wasm com poucas ou nenhuma mudança no código original. Um exemplo disso é o BROWSIX (JANGDA et al., 2019), um kernel unix que permite que programas feitos para sistemas POSIX (LEWINE, 1991) sejam executados sem mudanças. Com isso é possível fazer comparativos de desempenho mais fiéis (JANGDA et al., 2019). Essa possibilidade incentiva novas tecnologias de computação distribuída (Edge computing) (MENDKI, 2020), que aproveitam o processamento disponível nos clientes para diminuir cargas de processamento em servidores sem grandes alterações em código já disponível.

Alguns artigos mostram que o desempenho se comparando com o código sendo executado nativamente é igual ou maior (MALLE et al., 2018) dependendo do navegador, ou pode ter perdas que giram em torno de 1,2 a 2 vezes de desempenho (WANG et al., 2019), enquanto outros mostram perdas de até 100 vezes (GADEPALLI et al., 2019). Isto mostra que os ganhos de desempenho são dependentes da aplicação, por isso deve ser feita uma análise de custo benefício antes. Essas divergências de desempenho podem ser explicadas ainda pela imaturidade dos compiladores wasm, que estão competindo com compiladores robusto e com anos de melhorias como GCC ou Clang. Para endereçar isso foram criadas propostas de melhorias, como suporte a algumas otimizações padrão da indústria, por exemplo, instruções para múltiplos dados (SIMD) e *threads* (W3C MOZILLA, 2021).

3.2 Linguagem alvo para os simuladores

Escolher a plataforma Wasm flexibiliza escolha da linguagem alvo a ser usada para implementar os simuladores das máquinas. Com isso o foco passa a ser no desempenho de linguagem e compatibilidade e suporte a plataforma, por ela ser recente. Duas linguagens se destacam por esse parâmetros: C++ com o compilador Emscriptem (ZAKAI, 2011), e Rust (MATSAKIS; KLOCK, 2014) com o seu compilador nativo.

Rust foi escolhido para esse projeto por interesse pessoal de aprender a linguagem. Em termos de desempenho Rust possui uma velocidade comparável com C/C++ em ambientes comuns, *benchmarks* comparativos mostram ganhos de desempenho em

alguns casos podendo ser até mais rápidos (BAGLEY, 2021). Por ser concebida pela Mozilla Research, uma das empresas que ajudou a criar Wasm, a sua compatibilidade com a plataforma é mais robusta que outras linguagens com uma comunidade ativa para a solução de problemas. O gerenciador de pacotes **cargo** é fornecido junto com a instalação da linguagem, facilitando o desenvolvimento e replicação de ambiente, assim como o fácil acesso a bibliotecas. As configurações do projeto estão no arquivo *Cargo.toml*. A configuração do compilador Rust para a plataforma Wasm é feita com o comando:

```
rustup target add wasm-unknown-unknown
```

E o compilando com :

```
cargo build --target wasm32-unkown-unkown
```

Apesar de semelhante à C++, Rust possui algumas diferenças que serão explicadas para melhor entendimento da implementação dos simuladores. Para ganho de desempenho Rust faz uso de um sistema **verificador de empréstimo** (*borrow checker*) para eliminar a necessidade de um coletor de lixo (*garbage collector*). Isso é feito usando o conceito de donos de áreas de memória (*ownership*). Explicando de forma simplificada, o compilador atribui a cada variável criada um escopo que será dono dela. Essa variável pode ser **emprestada** (*borrow*), assim como um empréstimo de um objeto físico não é possível emprestar para mais de uma parte de código ao mesmo tempo, ou **passada adiante** para outra parte de programa usá-la, a região de memória não é copiada permanecendo a mesma. Mantendo o controle quem é o dono e quem está usando cada variável possibilita que o compilador infira quem e quando estão usando cada variável. Uma consequência disto é que se sabe quando uma variável deixa de ser usada, permitindo saber em tempo de compilação quando ela pode ser limpa da memória. Isso permite remover a necessidade de um coletor de lixo (*garbage colector*) em tempo de execução e removendo a sua sobrecarga no desempenho.

O sistema de *borrow checker* traz uma diferença profunda no funcionamento comparando com outras linguagens, a chamada de função usando argumentos por referência ou por valor. Passagem por referência é muito similar a *borrow* em Rust. Mas passagem por valor não existe, o que ocorre é que a variável é 'dada' para o escopo da função sendo chamada, fazendo com que ela seja limpa após o término da função se ela não for 'devolvida' para o escopo chamador da função, exemplo disso pode ser visto na Figura 3.1. Esses conceitos são importantes para que o programador não lute contra o *borrow checker* por seu código não compilar.

Figura 3.1: Exemplo de código (passagem de propriedade)

```
fn foo(a: i32) -> u32{
    // faz alguma coisa e retorna o mesmo valor
    return a
}
let a = 0;
let b = foo(b);

println!("{}", b); // Imprime o mesmo valor de a
// Erro de compilacao, a nao existe mais nesse escopo
println!("{}",a);
```

Figura 3.2: Exemplo de código (estrutura, *trait*, e implementação)

```
trait Caracteristica {
    fn foo(&self) {println!("foo de Caracteristica");}
}
#[derive(Debug)]
struct Baz { a: u32 };

impl Caracteristica for Baz {}

let baz = Baz{a:0};
println!("{} {}". baz, baz.foo());
```

Rust não faz o uso de classes ao invés usa **estruturas** de forma similar a C, ou seja, são estruturas de dados que **implementam** funções que podem modificar o seu estado. Por não serem classes não é possível fazer herança entre classes, como substituto é oferecido o mecanismo de características (*traits*). Elas funcionam de forma similar a classes abstratas, porém somente com funções e sem variáveis. Para uma estrutura possuir essas funções é preciso implementar um *trait*, modificando ou não as funções definidas pela *trait*, como visto na Figura 3.2. É importante notar que uma *trait* serve como identificador de tipo, ou seja, uma função que requer a *trait* **A** irá aceitar qualquer estrutura que a implemente.

Essas restrições relacionadas com a segurança de memória causam com que o código final seja bastante idiomático. Para facilitar a vida do programador Rust oferece macros geradores de código. Dois exemplos podem ser visto na Figura 3.2, os macros ***#[derive(Debug)]*** e ***println!***. O macro ***derive*** implementa automaticamente a *trait* ***Debug***, usada para imprimir na tela os elementos de uma estrutura. ***println!*** é uma função macro usada para formatar uma *string* e imprimi-la na tela.

Entender como um macro funciona internamente não é importante para Rust,

Figura 3.3: Exemplo de código (declaração bindgen)

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

mas entender ele de maneira rasa pode ajudar o entendimento de alguns erros de compilação comuns. O principal exemplo de uso de macros nesse projeto é a biblioteca *bindgen* (THE..., 2021), usada para gerar a camada de comunicação entre Wasm e JS. A Figura 3.3 mostra como a função *alert* é importada e *greet* é exportada para JS de forma simples usando o macro ***#[wasm_bindgen]***.

3.3 Interface

Desenvolver uma página web envolve várias disciplinas e uma pilha tecnológica grande. Para implementar a interface web foram utilizadas as linguagens padrão de indústria, Hyper Text Markup Language (HTML) (BERNERS-LEE; CONNOLLY, 1995), Cascading Style Sheets (CSS) (BOS et al., 2005), JavaScript (JS) (SEVERANCE, 2012). Elas foram escolhidas para reduzir a complexidade e dependências adicionais ao projeto. Basicamente HTML é utilizado para definir a estrutura da página, CSS para definir o estilo visual, e JS para definir o comportamento dos elementos e da interação do usuário com a página.

Para a estilização da página foi usada o *framework* Bootstrap 4.6 (KRAUSE, 2016). Ela define estilos em CSS através de classes de fácil uso e o posicionamento dos elementos é feito com um sistema de grade.

Para a organização e empacotamento do projeto final foi usado *npm* e *webpack*. Essas ferramentas são usadas apenas para o ambiente de desenvolvimento. *Npm* é um gerenciador de pacotes voltado para aplicações web, ele é usado principalmente para especificar as versões das dependências do projeto de maneira simples e replicável. Todas dependências usadas se encontrão no arquivo *packages.json*. Para instalar as dependên-

cias basta usar o comando no arquivo raiz do projeto:

```
npm ci
```

Webpack é usado para gerar um pacote final com todos arquivo necessários para hospedar o website, incluindo os arquivos com os modelos Wasm. O comando usado para isso é:

```
npm run build
```

4 IMPLEMENTAÇÃO

A implementação dos simuladores é dividida em duas partes, modelo e interface. Modelo se refere a implementação dos simuladores das máquinas hipotéticas seguindo as suas especificações. A interação com o usuário é referida como interface. Sendo assim, este capítulo descreve a implementação das duas e como é dada a comunicação entre elas.

4.1 Simuladores

Os simuladores foram desenvolvidos seguindo as suas especificações comportamental, ou seja, alguns detalhes específicos da sua organização de hardware não são implementadas. A implementação dos simuladores da família Neander foi feita de maneira similar, para aproveitar suas similaridades de arquitetura no reuso de código. O simulador Cesar foi feito do zero, por isso sua implementação foi feita de forma diferente. Isso se deve principalmente aos aprendizados com as dificuldades com a implementação dos simuladores anteriores. Por isso essa sessão será dividida nessas duas iniciativas.

4.1.1 Neander e extensões

Para o compartilhamento e extensão de comportamento entre os simuladores Neander, Ahmes, e Ramses a implementação foi divida em quatro funcionalidades básicas:

- Controle de memória.
- Controle de registradores.
- Operações aritmética.
- Decodificação e execução de instruções.

O controle de memória é feito pela *trait* **Memory** foi criada para ter um acesso unificado a memória e também a contagem de acessos. Para representação da memória da máquina, todos os simuladores usam uma lista contínua de bytes, tipo *u8* em Rust, com 256 elementos. Na sua forma simples apenas acesso modo direto é possível, para implementar o modos usados pelo Ramses **Memory** foi estendida em conjunto com **Re-**

gisterBank para que se tenha acesso ao registrador **rX** e seja possível fazer o acesso de modo indexado.

Para o controle de registradores foi criada a *trait* **RegisterBank**. Ela é simples e possui apenas funções de acesso e modificação aos registradores:

- Apontador de programa (PC).
- Registrador de instrução (RI) sendo executada no ciclo atual.
- Registradores gerais enumeráveis, acumulador é identificado como 0.

As operações aritméticas são implementadas por **SimpleAlu**, ela implementa funções aritméticas do Neander e controles dos sinais de condições. Cada função altera os sinais de condição de acordo com a especificação, porém esse comportamento é diferente em cada um dos três simuladores por terem diferentes sinais. Para manter a compatibilidade com Ahmes e Ramses sem que funções sejam reescritas todos os sinais de condição são definidos (Zero, Negativo, Carry, Overflow, e Borrow) e alterados de acordo, dessa forma não há perda de informação e cada simulador usa os sinais especificados em cada. **ExtendedAlu** é criada para adicionar mais funções usadas por Ahmes e Ramses de acordo com o comportamento adequado de cada.

A *trait* **Runner** unifica as características mostradas anteriormente para definir o fluxo de funcionamento do simulador. A busca de instrução pode ser definida da mesma maneira para todos simuladores, porém a decodificação deve ser implementada por cada um individualmente. Isso requer duplicação de código, mas devido à inexperiência outra solução mais elegante não foi descoberta. Na Figura 4.1 a implementação do decodificador para o Neander pode ser vista, nela é usado *pattern matching* nos 4 bits mais significativos da instrução para identificar qual operação deve ser executada. É esperado um booleano de retorno para indicar se uma instrução de parada foi atingida (*halt*) ou um valor inesperado.

4.1.2 Cesar

Como o Cesar possui um conjunto de instruções e comportamento mais complexo que o Neander as suas estruturas de controle foram reescritas. Um comportamento adicional fundamental são as interrupções de hardware para entrada de dados pelo teclado. Ela somente é processada ao fim da execução de uma instrução. Sendo assim, o comportamento de um ciclo do Cesar segue os seguintes passos:

Figura 4.1: Exemplo de código (Decodificação e execução de instrução no Neander)

```
fn decode_and_execute(&mut self) -> bool {
    let operator = (self.get_ri() & 0b1111_0000) >> 4;
    match operator {
        0x1 => self.str(),//store
        0x2..=0x5 => self.ula_operation(),//ula operation
        0x6 => self.op_not(),// NOT
        0x8 => self._jmp_if(true),
        0x9 => self._jmp_if(self.get_negative()), //JN
        0xA => self._jmp_if(self.get_zero()), // JZ
        0x0 => return true,// NOP
        0xF => return false,// HLT
        _ => return false
    }
    return true
}
```

1. Busca instrução de memória
2. Decodifica instrução
3. Executa instrução
4. Processa entrada de dados do teclado.

A decodificação de instrução funciona de forma diferente. Enquanto nos simuladores anteriores a informação era extraída no momento de ser usada, diretamente do código de instrução, no Cesar uma estrutura decodificador foi criada para gerar uma instância com todas informações relevantes da instrução. Com isso o código de execução de instruções se tornou mais claro e idiomático. Reduzindo o números de erros gerados. A leitura de memória usando modos ficou clara pela mesma razão, pois *enum* são usados para descrevê-los.

A memória do simulador é representada pela estrutura **MemoryBank**. Sua implementação é bastante simples, contendo apenas funções de leitura e escrita imediata na memória, representada por uma lista continua de 64k elementos do tipo *u8*, e contagem de acessos.

Uma grande diferença em relação aos simuladores da família Neander é a implementação das operação aritméticas. Todas foram simplificadas em funções que recebem além dos operandos uma referência com informação de variáveis de condição que são usadas e/ou modificadas, de acordo com a instrução.

Ao fim de um ciclo de instrução, é verificado se não houve algum dado de entrada do teclado e se ele deve ser processado. A função usada recuperar o conteúdo do

visor retorna os caracteres ASCII equivalentes aos valores em memória, sem modificar o contador de memória. Caso seja o valor de um ASCII não visível ele é substituído por um caractere de espaço (32 em ASCII).

4.1.3 Camada de comunicação entre Rust e JS

Com os simuladores compilados em Wasm, ainda é necessário criar a camada de comunicação entre Wasm e JS, ou seja, criar os *bindings* entre elas para que uma possa usar funções e objetos da outra com chamadas específicas para cada tipo de dados e conversões necessárias em alguns casos, gerando uma repetição de código considerável. Isso é abstraído pelo pacote Rust *wasm-bindgen* onde apenas criando uma estrutura de interface simples em Rust ela é exportada para JS automaticamente.

Foi criada uma interface simples para controle dos simuladores, as principais funções são:

- Criação de um instância de simulador
- Executar ciclos do simulador.
- Verificar e definir estado do simulador, registradores, memória, estatísticas de uso.
- Carregar a memória do simulador.
- Para o simulador Cesar: controle de entrada e saída de dados.

4.2 Interface

Para manter a familiaridade dos simuladores já existentes, a interface foi criada com uma estrutura similar. Um comparativo das interfaces será mostrado ao final desta sessão. As funcionalidades básicas, para o controle do simulador são:

- Importar arquivo de memória dos simuladores **.mem**, para carregar o simulador.
- Visualizar a memória do simulador, e modificar esses valores.
- Visualizar os registradores e mudar o seus valores.
- Mostrar estatísticas de acesso de memória por instruções e contador de instruções executadas.
- Executar o programa passo a passo ou continuamente até o fim do programa .

Para o simulador Cesar, duas funcionalidades adicionais são necessárias:

- Visualização de um visor de 36 caracteres.
- Capturar e enviar entradas de teclado para o simulador.

O processo de implementação da interface começou com o uso de elementos básicos HTML, *tables* e *labels*, para a prototipação das classes de controle e interação com o simuladores. Estas classes de controle foram criadas usando o padrão de arquitetura *Model-View-Controller* (MVC) com controles essenciais para o uso dos simuladores.

A partir desses protótipos foi feita a estilização da página. Utilizando a biblioteca CSS de estilos *Bootstrap*, os elementos da página foram dispostos de maneira semelhante aos simuladores originais. Uma descrição das decisões de estilização da página é mostrada, assim como imagens comparativas da interface original e a implementada nesse trabalho.

Logo após é apresentado as dificuldades de desempenho enfrentadas na visualização da página. Analisando suas causas a partir de ferramentas dadas pelo navegador, e soluções chegadas para as mesmas.

4.2.1 Classes de controle

As estruturas de controle foram feitas de forma bem simples e direta. Todos os simuladores compartilham código para o controle das três áreas (registradores, dados, e programa) de interface. Cada simulador estende essas classes para selecionarem os elementos da página que serão controlados e adaptar o controle para elementos adicionais.

Áreas de interface controladas e breve descrição:

- Visualização de programa: Tabela com valores de memória do simulador. Possui três colunas:
  - Endereço de memória.
  - Valor de memória, mostrando valor em base decimal e hexadecimal.
  - Mnemônicos de instruções decodificados para cada linha, linha em branco fazem parte de uma instrução de uma linha anterior.
- Visualização de dados: Similar a visualização de dados, porém sem a coluna de Mnemônicos.
- Visualização de registradores e controle: Mostra os principais registradores e alguns registradores internos como registrador de instrução, de contagens de acesso de

memória e instruções. Assim como controles dos simuladores.

Uma classe **ProgramTableView** responsável controlar a área da interface que mostra a memória com mnemônicos para as instruções e controles para modificar a memória interna do simulador. A coluna de mnemônicos é criada usando um decodificador simples criado em JS. Ela é criada uma vez quando a memória é carregada, para valores modificados em poucas linhas apenas os mnemônicos afetados são ajustados, para evitar o recálculo de toda memória. Para o controle da tabela de dados a classe **DataTableView** estende **ProgramTableView** para que a coluna de mnemônicos não seja mostrada. Para que o usuário possa ver diferentes áreas de memória ao mesmo tempo, cada tabela é controlada de forma independente. As tabelas são renderizadas de forma especial, como será explicado na sub-sessão 4.2.3, para melhorias de desempenho.

A classe **RegisterView** é responsável pela visualização e controle da área de registradores. Nela é mostrado o estado dos seguintes componentes do simulador: registradores, sinais de controle, registrador de instrução (RI), e contadores de acesso a memória e instruções executadas. É permitido que o estado dos registradores possa ser editado também. O valor de RI é atualizado juntamente com o seu mnemônico, o decodificador utilizado é o mesmo usado pelas classes de controle de tabela.

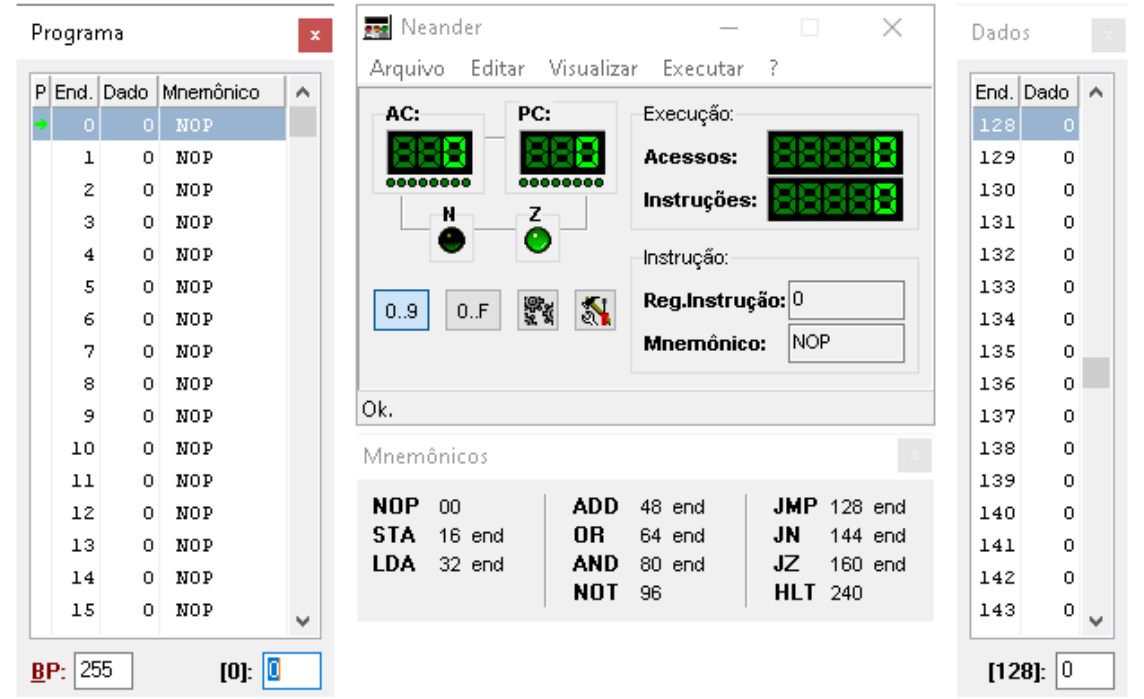
Todas essas classes são gerenciadas pela classe **ProcessorController**. Ela se encarrega de inicializar todas classes, conectando os elementos HTML as classes pertinentes. Também é responsavel em controlar o modelo dos simuladores e atualizar o seu estado para as outras classes **View**.

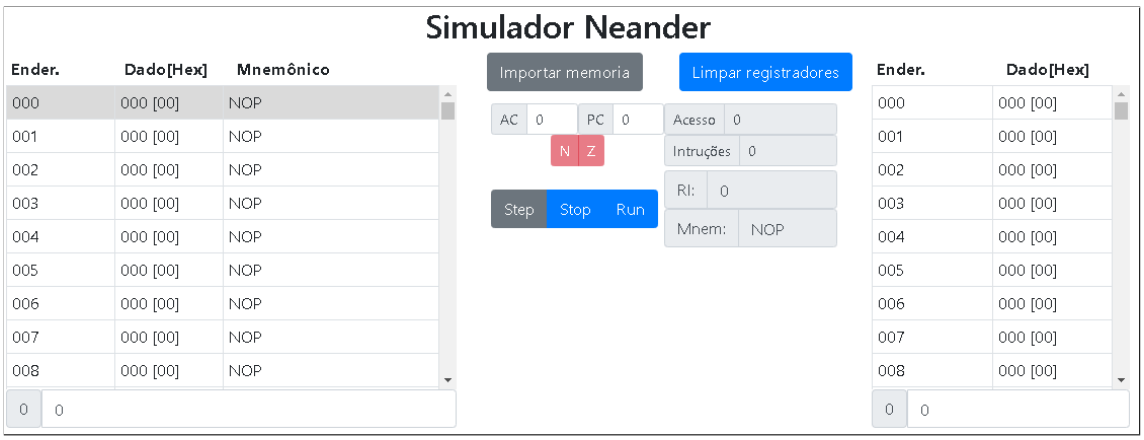
O simulador Cesar ainda possui suporte para entrada e saída de dados. A entrada é feita através de um teclado e a saída é um visor de 36 caracteres ASCII. A implementação é bem simples e segue a especificação. Para capturar inputs de teclado um *listener* é adicionado a página para enviar, com a função *input_keyboard*, ao simulador quando uma tecla, com valor ASCII menor que 255, é pressionado. O simulador é encarregado de salvar em memória o valor, respeitando a especificação.

4.2.2 Estilização

Para manter a familiaridade com os simuladores já existentes, a disposição dos elementos mostrados para controle dos simuladores foi mantida a mesma. Com a única diferença sendo como é mostrado os valores de memória, originalmente era possível es-

Figura 4.2: Simulador original X web para o Neander





colher mostrar eles em base decimal ou base hexadecimal. Os dois valores são mostrados lado a lado para que o aluno se familiarize e associe valores, facilitando a conversão de valores mentalmente para usos futuros. Nas Figuras 4.2, 4.3, 4.4 e 4.5 pode ser visto a semelhança da estruturas dos simuladores, note as diferenças na coluna **Dado**.

A página foi feita voltada para telas grandes (1080p ou maiores) num primeiro momento. Portanto os elementos podem ficar deslocados com tamanhos menores de tela. Essa disposição será ajustada em trabalhos futuros. Assim como configurações corretas de acessibilidade.

Figura 4.3: Simulador original X web para o Ahmes

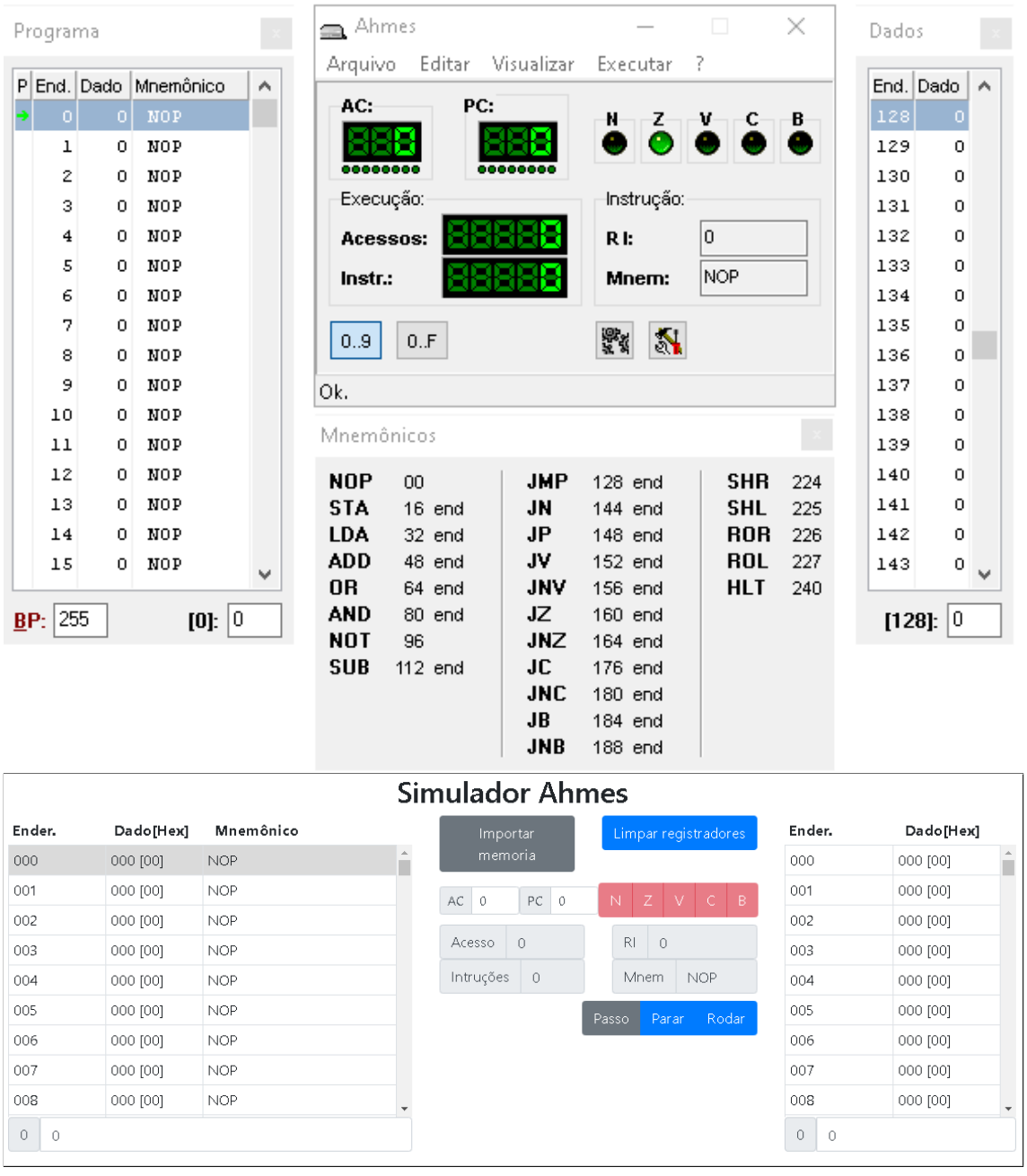


Figura 4.4: Simulador original X web para o Ramses

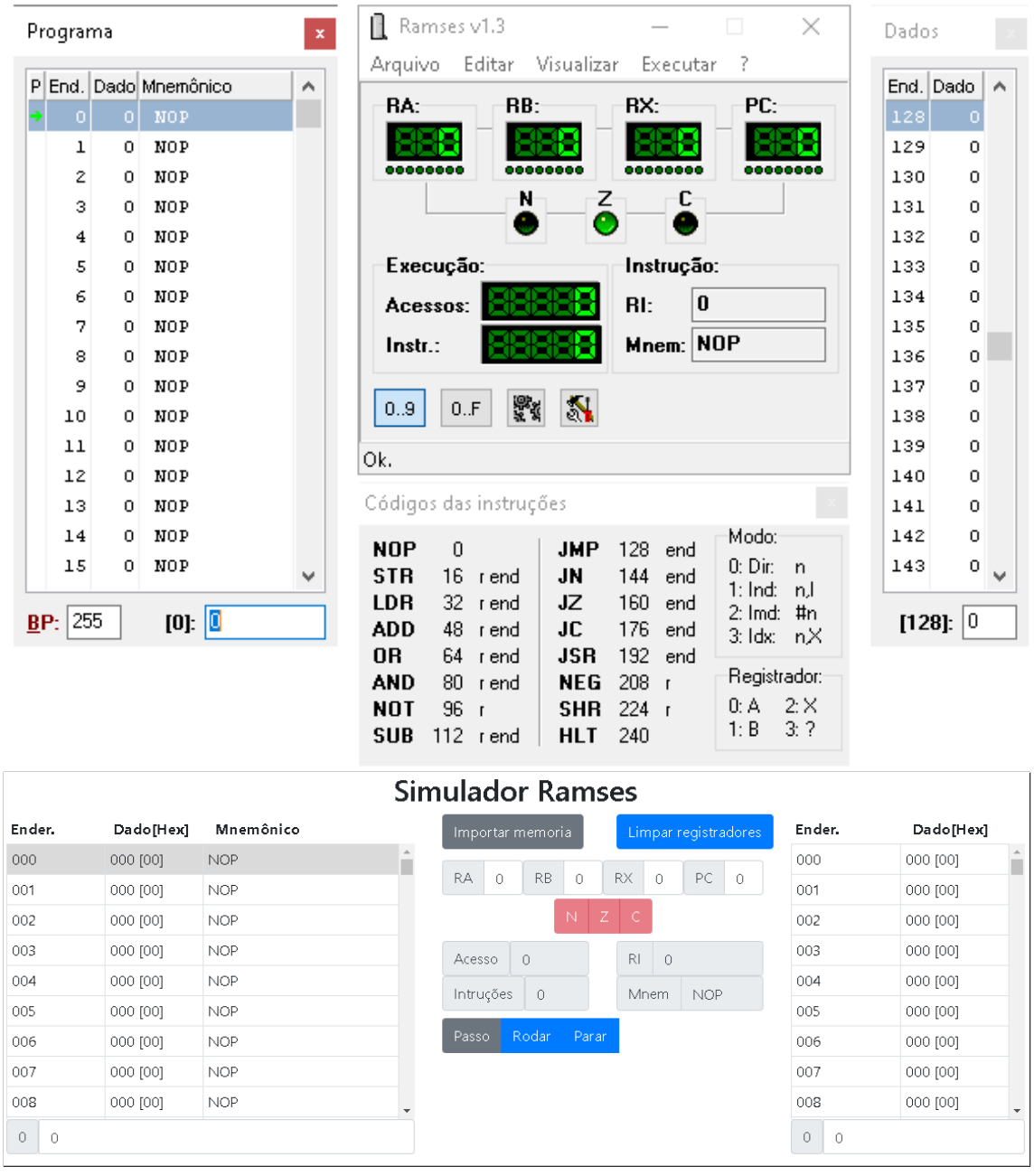
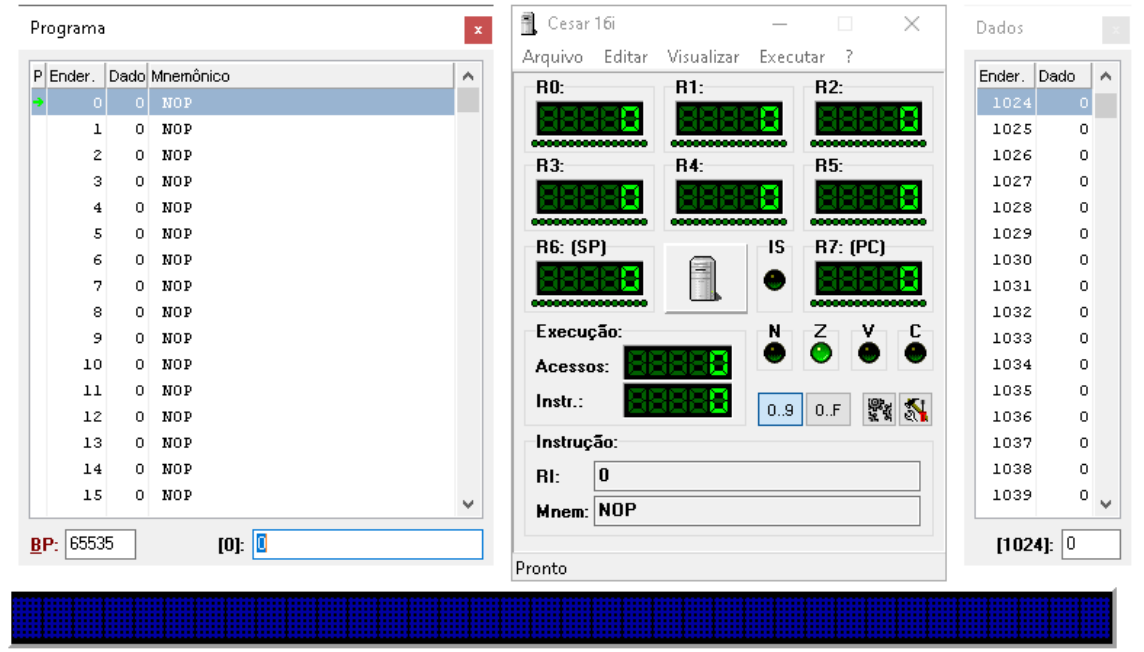


Figura 4.5: Simulador original X web para o Cesar



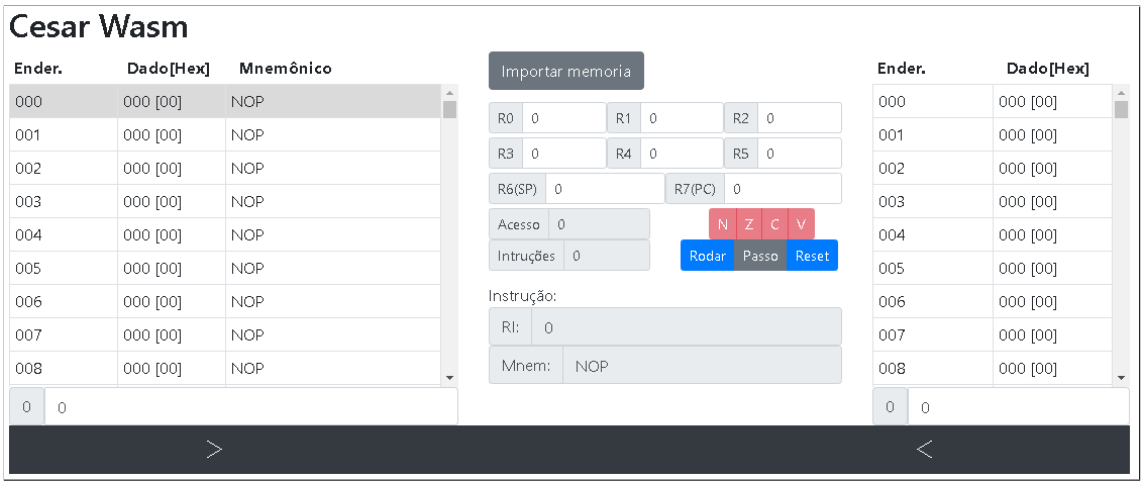
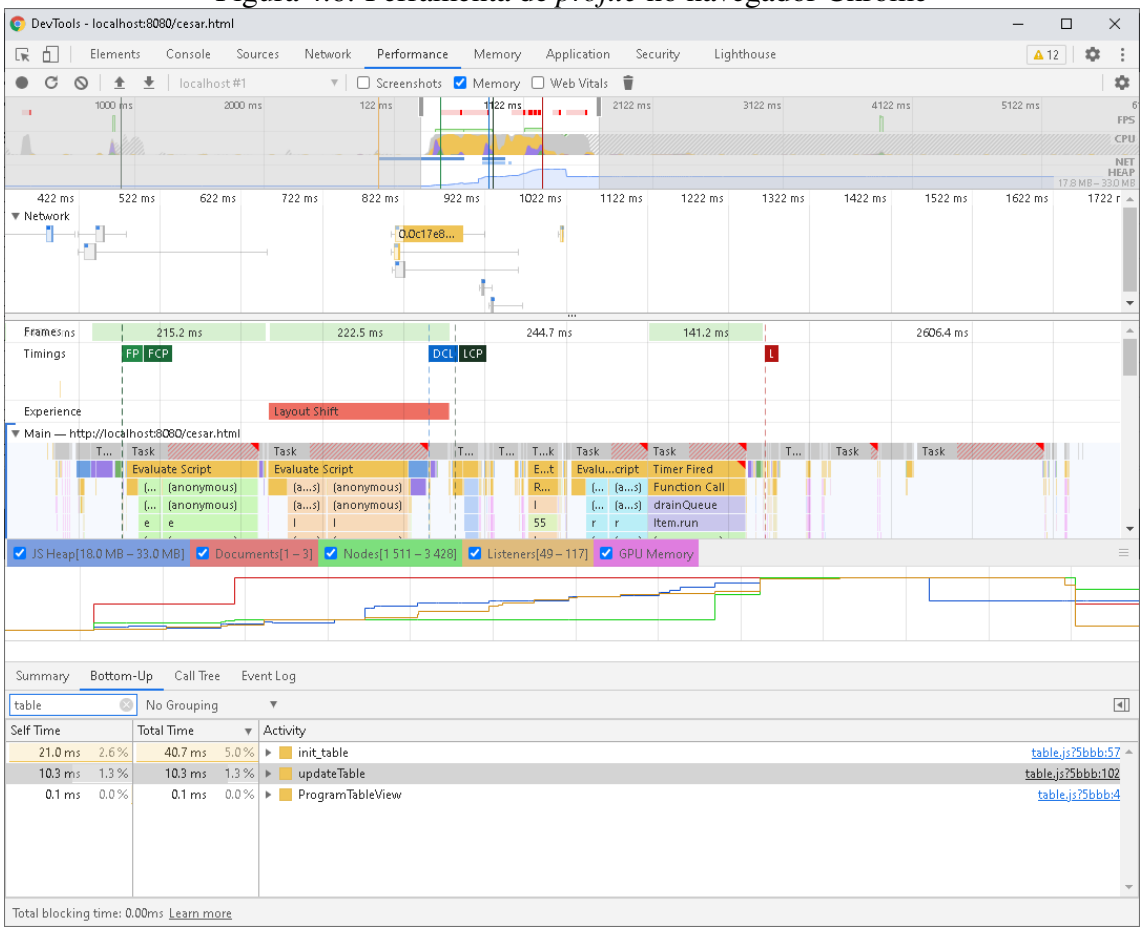


Figura 4.6: Ferramenta de *profile* no navegador Chrome



4.2.3 Melhorias no desempenho

Os simuladores Neander, Ahmes, e Ramses funcionaram como o previsto, sem diferenças notáveis de desempenho. Diferente do simulador Cesar, ao acessar a sua página nada era mostrado, ela apenas continuava em um estado de carregamento por um longo tempo. A primeira suspeita do causador disto foi a renderização das cercas de 65,000 linhas de memória do Cesar. Para testar essa hipótese foi colocado um limite de 1000 linha de memória na visualização da página, que carregou sem problemas.

Usando as ferramentas de desenvolvedor fornecidas pelo navegador Google Chrome. A Figura 4.6 mostra a aba de desempenho onde é possível coletar métricas precisas durante o carregamento da página, inclusive discretizados por linha de comando como pode ser visto na Figura 4.7

Com esses dados foi possível apontar os causadores desses problemas. O primeiro ponto da queda de desempenho é a renderização das tabelas de dados e programa. O primeiro causador é a forma como uma tabela em HTML calcula seu tamanho, o tama-

Figura 4.7: Ferramenta de *profile* mostrando o tempo total de processamento por linha

```
55          }
56
57          init_table() {
58  20.7 ms     this.currentData = Array.from({length: this.size},
59   2.2 ms         () => ([0, "NOP"]));
60
61          var tb = this.table_node.cloneNode();
62          tb.onclick = this.select_row.bind(this);
```

nho de cada coluna depende do maior conteúdo entre todas as linhas daquela coluna, e a redesenha por completo. Apenas carregar a página com o limite de 5000 linhas de dados levava cerca de 1 minuto, a renderização da memória completa nunca foi testada.

A implementação ingênua inicializava a tabela adicionando uma linha por vez com o mesmo conteúdo, causando o recálculo do tamanho da tabela a cada linha adicionada. Criando a toda tabela apenas em memória e então adicionando-a à página quando concluída resolve este problema para a inicialização dela.

Porém o problema permanece enquanto simulador estiver sendo executado, já que a cada mudança de memória causa o redesenho de toda a tabela. Qualquer mudança de valores na tabela causava o recálculo do tamanho da tabela inteira, causando a espera do mesmo tempo para cada mudança de memória que ocorria durante o funcionamento de um programa.

Para resolver esse problema foi usado a mesma funcionalidade usada para criar páginas 'infinitas', comumente usada na pagina inicial de redes sociais como Twitter ou Facebook, para a visualização continua de novos conteúdo carregados sobre demanda. O *IntersectionObservers* (FOUNDATION, 2021) é a funcionalidade central por trás disso, ele permite que elementos da página sejam observados e uma função seja chamada assim que ele se tornar visível na tela. O algoritmo básico para ter uma tabela 'infinita' envolve ter uma tabela com N linhas usada para mostrar M elementos, sendo $M > N$, e dois elementos vazios espaçadores em cima e abaixo da tabela. Assim é possível reciclar a tabela, mostrando apenas N elementos e mudando quais elementos são mostrados conforme o usuário move a rolagem (*scroll*) da área visível. Os espaçadores são usados para ajustar o tamanho total de área de rolagem, dando a ilusão que uma tabela com todos elementos está sendo mostrada.

Após resolver os problemas relacionados com a renderização da página, outro gargalo de desempenho foi encontrado a interface de comunicação entre Wasm e JS. Por padrão a passagem de valores, parâmetros de função e valores resultantes, é feito por cópia. Dependendo do tipo de valor ele também deve ser convertido, nesse caso todos valores serão convertidos de $u8$ para $Number$, ou seja, um vetor de 64k elementos serão

convertidos e copiados para a camada JS. Quando a atualização passo a passo está ativada a cada instrução rodada no simulador toda a memória do simulador é copiada para o JS para verificar se houve alguma mudança e atualizar a tabela.

Para solucionar esse problema a memória do modulo Rust é acessada diretamente, isso é possível através da variável do modulo Wasm *wasm_module.memory.buffer*. A interface Rust foi modificada para retornar o ponteiro de memória para a memória do simulador, com operador $*variavel$ ou a função $varivel.as\_ptr()$, os dois retornam a posição de memória do modulo Wasm em que a variável se encontra. Usando o ponteiro é possível calcular qual região está a lista que representa a memória do simulador e criar um objeto *Uint8Arrray* para poder visualizar esses dados. Dando acesso direto a memória do simulador sem criar uma cópia da lista.

5 TESTES

Foram feitos dois conjuntos de testes para serem usados neste projeto. Testes de aceitação para verificar se os simuladores seguem o mesmo comportamento dos simuladores existentes. Testes de desempenho para comparação da página web e os simuladores Wasm.

5.1 Testes de aceitação

Verificar se os simuladores seguem a suas especificações se torna rapidamente uma tarefa impossível de ser feito manualmente. Para automatizar esse processo testes unitários foram utilizados inicialmente. Porém houve uma grande dificuldade de desenvolver eles sem que o princípio de caixa preta fosse ferido, ou seja, os testes 'sabiam' como o simuladores eram implementados.

Visto que já haviam soluções testadas e validadas de todos os simuladores eles podem ser usados para comparar com os simuladores desenvolvidos nesse projeto. De forma ideal seriam comparados as mudanças de estado, registradores e memória, do simulador ao executar uma instrução, acesso aos registradores não é possível apenas o estado de memória poder ser facilmente acessado através dos arquivo **.mem**. Carregando um programa no simulador, executando-o até sua parada e por fim salvando o estado de memória final podemos capturar parte do comportamento dos simuladores. Com isso podemos comparar as duas implementações. O requisito principal dos programas teste usados para eles terem alguma utilidade é que eles devem modificar a memória do programa baseado no efeito colateral das instruções executadas nele.

Foi criado um programa teste para cada instrução de cada simulador, para diminuir a complexidade de cada testes e facilitar a identificação de um possível problema. Todos programas assumem que todo o conjunto de instruções do simulador estão seguindo a sua especificação, exceto a instrução alvo do teste. Resultados parciais de testes são salvos em memória durante o teste se possível e desvios condicionais são usados para verificar se sinais de condição foram mudadas corretamente ao executar uma instrução. Assim, se todos os programas testes passarem para um simulador, temos a garantia que todas as instruções seguem o mesmo comportamento dos simuladores existentes, portanto seguem a especificação.

Foram criados um total de 80 programas testes. Os testes para o Neander são

Figura 5.1: Exemplo de código: Programa teste para instrução LDA do Neander

```
; Testa guardar na memoria
LDA MAX ; AC = 255(-1)
STA T0
LDA T0  ; AC = 255(-1)
JZ ERR
JN P1
JMP ERR
P1:
ADD ONE ; AC = 255 + 1 = 0 e sobra 1
STA T1
JN ERR
JZ END
JMP ERR

ERR:
LDA ONE
STA E
END:
HLT

ORG 128
T0: 1
T1: 1
ONE: 1
MAX: 255; Valor maximo em byte
E: 0    ; Variavel para indicar que teve um erro no teste
```

usados para testar o Ahmes e Ramses para garantir compatibilidade. Cada simulador ficou com o seguintes números de testes:

- Neander: 8 testes
- Ahmes: 17 testes
- Ramses: 14 testes
- Cesar: 41 testes

Com estes testes em mãos foi possível implementar os simuladores de forma rápida e iterativa seguindo o princípio de desenvolvimento voltado a testes (*Test Driven Development*). Visto que esses testes são independes da implementação do simulador, eles podem ser usados para verificação de outros simuladores ou implementações em hardware de futuros simuladores.

Tabela 5.1: Resultado dos testes comparativos

Teste	Média (IPS)	Variância	*t*	*t* critico	*p*>0.05
Neander / Ahmes / Ramses					
Existente	8,01E+05	1,94E+12	-	-	-
Chrome	4,46E+05	8.37E+07	1,61	2,02	0,11
Firefox	5,03E+05	2,9E+08	1,35	2,02	0,18
Cesar					
Existente	1,00E+06	2,34E+08	-	-	-
Chrome	2,49E+06	4,55E+10	-3,19E+01	2,09	5,99E-19
Firefox	9,41E+04	5,26E+06	3,71E+02	2,02	2,53E-77

Comparação dos resultados dos testes de desempenho $(N = 40)$

5.2 Testes de desempenho

As implementações dos simuladores foram comparadas utilizando a medida de instruções por segundo (IPS). Em primeiro momento foi usado o valor do contador de instruções fornecido nos simuladores originais. Porém esse contador tem um limite máximo de 99999, ao atingir esse limite o valor voltava a 0, inviabilizando o seu uso para fins de testes.

Para contornar isso foi criado um programa de teste que executa todas as instruções da máquina alvo em sequência, já que diferentes instruções executam em tempos variados, e incrementa um contador ao fim e retornando ao início do programa, código dos programas são encontrados em anexo. Dessa forma sabe-se o número de instruções executadas a cada repetição, podendo calcular o total de instruções executadas em *N* segundos, assim o valor IPS pode ser calculado com a fórmula $(Loops/10) * Instr_{loop}$. A coleta de medições foi feita usando ferramentas de automações de teclados, o programa é executado do início por exatamente 10 segundos e parado em simuladores antigos. Na solução deste trabalho a automação foi feita utilizando o console do navegador, o mesmo processo de medição foi feito. Todos testes foram feitos com a atualização de dados cada ciclo desativada, para ser comparado o desempenho do simuladores sem interferência. Testes foram feitos em dois navegadores, Google Chorme e Mozilla Firefox.

A Tabela 5.1 mostra os dados coletados ao longo desses testes. Foram feitos testes apenas para o Ramses e Cesar, testes com Neander e Ahmes são considerados equivalente ao Ramses por terem a mesma base de implementação.

Para os simuladores da família Neander, não há uma diferença estatística significativa comparando com o simulador existente.

A maior surpresa é a comparação do simulador Cesar. Testes feitos no Chrome

mostram melhoras de cerca de 2.5 vezes enquanto no Firefox há uma queda de desempenho em ordens de magnitude. A plataforma Wasm ainda possui diferenças entre suas implementações na indústria, por isso uma possível explicação é uma otimização presente no Chrome ainda não presente no Firefox. Uma hipótese envolve como a decodificação de instrução é feita pelo simulador Cesar, maior diferença entre as duas implementações, envolvendo a criação de uma instância de *Enum* para cada instrução decodificada.

Essa diferença pode ser mais acentuada em outras plataformas, por isso um cuidado deve ser tomado para a adaptação para outras plataformas com menos capacidade de processamento, por exemplo mobile.

6 CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS

Comparando os objetivo proposto no início deste trabalho com o produto final, pode-se concluir que boa parte do proposto foi alcançado. Uma solução foi desenvolvida para a plataforma web com desempenho comparável as soluções existentes, inclusive mostrando melhorias significativas de desempenho em alguns casos.

Os resultados atingidos também trazem promessas para a plataforma Wasm para implementações de outros projetos que possam se aproveitar da facilidade de distribuição dela e do esforço mínimo para adaptação de código já existente. Ferramentas de ensino são um bom exemplo disso, facilitando o acesso do aluno a elas sem a ingrime curva de introdução muitas vezes presente na instalação de diversas ferramentas necessárias para o auxílio no aprendizado.

Este trabalho apenas criou uma página simples como prova de conceito. Várias melhorias podem ser feitas na mesma, elas envolvem desde ajustes para outros dispositivos à integração de novas funcionalidades. Algumas delas são:

- O uso dos simuladores depende da importação de programas, armazenados em arquivos **.mem**, estes são gerados por um montador que compila um arquivo texto para arquivo de memória. O montador no presente momento é o programa Daedalus na plataforma *Windows* que permite a edição e montagem de programas. CCom isso é recomendado a implementação de um montador em conjunto com um editor para remover a dependência de plataforma simuladores.
- Para tornar a página mais didática seria interessante a integração de conteúdos das disciplinas, mostrando materiais e exemplos para o uso da página.
- Melhorias de acessibilidade podem ser feitas utilizando o padrão *web* já existente com *tags* do tipo **aria** para democratizar o acesso dos alunos.
- Ajustes na estilização da página para dispositivos de tamanhos variados de tela, incluindo dispositivos mobile.
- Adição de outros simuladores usados na disciplina, como o intel x8086.
- Integração da página com um sistema de armazenamento e sincronização de arquivos.
- Adicionar mais funcionalidades de análise do programa no simulador, por exemplo, *breakpoints* condicionais para parar a execução do programa assim que a condição for verdadeira.

REFERÊNCIAS

ARAUJO, V. de. **neander - Simulators for the Neander and Ahmes didactical architectures**. 2021. Acessado em: 06 de maio de 2021. Disponível em: <https://elmord.org/code/neander/>.

BAGLEY, D. **Rust versus C gcc**. 2021. Acessado em: 15 de maio de 2021. Disponível em: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html>.

BERNERS-LEE, T.; CONNOLLY, D. **Hypertext markup language-2.0**. [S.l.]: RFC 1866, November, 1995.

BOS, B. et al. Cascading style sheets level 2 revision 1 (css 2.1) specification. **W3C working draft, W3C, June**, Citeseer, 2005.

CAMPOS, K. C. Implementação de máquinas hipotéticas (neander e ahmes) e interface vga. **Ufrgs.br**, 2016. Disponível em: <https://lume.ufrgs.br/handle/10183/150939>.

FOUNDATION, M. **Intersection Observer API - Web APIs | MDN**. 2021. Acessado em: 15 de maio de 2021. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API>.

GADEPALLI, P. K. et al. Challenges and opportunities for efficient serverless computing at the edge. In: . [S.l.: s.n.], 2019. ISSN 10609857.

HAAS, A. et al. Bringing the web up to speed with webassembly. In: **Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2017. p. 185–200.

HERMAN, D.; WAGNER, L.; ZAKAI, A. **asm.js**. 2014. Disponível em: <http://asmjs.org/spec/latest/>.

JANGDA, A. et al. Not so fast: Analyzing the performance of webassembly vs. native code. In: . [S.l.: s.n.], 2019.

KRAUSE, J. **Introducing Bootstrap 4**. [S.l.]: Springer, 2016.

LEWINE, D. **POSIX programmers guide**. [S.l.]: "O'Reilly Media, Inc.", 1991.

MALLE, B. et al. The need for speed of ai applications: Performance comparison of native vs. browser-based algorithm implementations. **arXiv**, arXiv, 2 2018. Disponível em: <http://arxiv.org/abs/1802.03707>.

MATSAKIS, N. D.; KLOCK, F. S. The rust language. **Ada Lett.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 3, p. 103–104, oct. 2014. ISSN 1094-3641. Disponível em: <https://doi.org/10.1145/2692956.2663188>.

MENDKI, P. Evaluating webassembly enabled serverless approach for edge computing. In: . [S.l.]: Institute of Electrical and Electronics Engineers (IEEE), 2020. p. 161–166. ISBN 9781728182667.

ORTH, G. K. Implementação em hardware da arquitetura do computador hipotético cesar. **www.lume.ufrgs.br**, 2010. Disponível em: <https://www.lume.ufrgs.br/handle/10183/27969>.

PROJETO Hidra. 2021. Acessado em: 26 de maio de 2021. Disponível em: <https://github.com/petcomputacaoufrgs/hidracpp>.

SEVERANCE, C. Javascript: Designing a language in 10 days. **Computer**, IEEE, v. 45, n. 2, p. 7–8, 2012.

SILVA, G. P.; BORGES, J. A. S. **O Simulador Neander-X: Para o Ensino de Arquitetura de Computadores**. 1a edição. ed. Gabriel P. Silva, 2016. Disponível em: <https://www.amazon.com.br/Simulador-Neander-X-Ensino-Arquitetura-Computadores-ebook/dp/B01DX2VLJA>.

THE 'bindgen' User Guide. 2021. Acessado em: 06 de maio de 2021. Disponível em: <https://rust-lang.github.io/rust-bindgen/>.

W3C MOZILLA, M. G. A. **WebAssembly Roadmap**. 2021. Acessado em: 15 de maio de 2021. Disponível em: <https://webassembly.org/roadmap/>.

WANG, S. et al. Leveraging webassembly for numerical javascript code virtualization. **IEEE Access**, v. 7, 2019. ISSN 21693536.

WEBER, R. **Fundamentos de Arquitetura de Computadores - Vol.8: Série Livros Didáticos Informática UFRGS**. Bookman Editora, 2009. ISBN 9788540701434. Disponível em: <https://books.google.com.br/books?id=UAiPQ60dRjMC>.

ZAKAI, A. Emscripten: an llvm-to-javascript compiler. In: **Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion**. [S.l.: s.n.], 2011. p. 301–312.