**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO**

# SOFTWARE SIMULADOR DO MICROCONTROLADOR M+++

**JEAN CARLOS KLANN**

**BLUMENAU**
**2017**

**JEAN CARLOS KLANN**

# SOFTWARE SIMULADOR DO MICROCONTROLADOR M+++

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof(a). Miguel Alexandre Wisintainer, Mestre - Orientador

**BLUMENAU**
**2017**

# SOFTWARE SIMULADOR DO

# MICROCONTROLADOR M+++

Por

**JEAN CARLOS KLANN**

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

| | |
|---|---|
| Presidente: | Prof(a). Miguel Alexandre Wisintainer, Mestre – Orientador, FURB |
| Membro: | Prof(a). Mauro Marcelo Mattos, Doutor – FURB |
| Membro: | Prof(a). Gilvan Justino, Mestre – FURB |

Blumenau, 3 de julho de 2017

Dedico este trabalho a minha família e a todos as pessoas que me ajudaram nesta jornada.

## AGRADECIMENTOS

Agradeço aos meus pais, Maria Salete Klann e Valdemar Klann e a minha namorada, Rúbia Cristina da Silva, que sempre me apoiaram em todas as fases da minha vida.

Ao meu orientador Miguel Alexandre Wisintainer pela dedicação na orientação deste trabalho.

Para se ter sucesso, é necessário amar de verdade o que se faz. Caso contrário, levando em conta apenas o lado racional, você simplesmente desiste. É o que acontece com a maioria das pessoas.

Steve Jobs

# RESUMO

Este trabalho apresenta o desenvolvimento de uma ferramenta de programação, depuração e simulação do microcontrolador M+++, visando o auxílio na aprendizagem de alunos que estudam ou desejam estudar as matérias da área de sistemas embarcados, computação digital, arquitetura de computadores, entre outras afins. O software permite a programação, importação, exportação e montagem do código para ser executado em um ambiente controlado, na qual é possível inserir pontos de parada, ajuste da velocidade de simulação, execução passo a passo, entre outras características. Para a programação e depuração foi utilizado o *framework* Scintilla.NET, o qual permite realizar o realce de sintaxe. Permite também a montagem de circuitos eletrônicos digitais, podendo anexar e fazer associações de portas lógicas, *displays*, decodificadores, entre outros. Para o desenvolvimento desta montagem de circuito foi implementado um interpretador de circuitos eletrônicos próprio para esta ferramenta. Além disso, é permitido a visualização interna deste microcontrolador, podendo reconhecer os sinais internos entre cada componente.

Palavras-chave: Microcontrolador. Simulador. Depuração. Emulador.

# ABSTRACT

This work presents the development of a tool for programming, debugging and simulate the M +++ microcontroller, aiming at helping students to learn the subjects of embedded systems, digital computing, computer architecture, among others. The software allows programming, import, export and build the code to be executed in a controlled environment, in which it is possible to define breakpoints, simulation speed, step by step, among other characteristics. For programming and debugging it was used the framework Scintilla.NET, which make the syntax highlighting. It also allows building digital electronic circuits, being able to attach and make associations of logic gates, displays, decoders, among others. For the development of this circuit, it was implemented an electronic circuit interpreter for this tool. In addition, it is allowed the internal visualization of this microcontroller, being able to recognize the internal signals between each component.

Keywords: Microcontroller. Simulator. Debugging. Emulator.

# LISTA DE FIGURAS

# LISTA DE QUADROS

# LISTA DE ABREVIATURAS E SIGLAS

DLL – Dynamic-link library

EOI – End of Instruction

HD – High Decoder

Hz – Hertz

IPS – Instruções por Segundo

RAM – Random Access Memory

ROM – Read Only Memory

UCP – Unidade Central de Processamento

ULA – Unidade Lógica e Aritmética

# SUMÁRIO

# 1 INTRODUÇÃO

Atualmente existem diversas plataformas dedicadas ao desenvolvimento de software, sendo que apenas uma parcela destas é especializada ou possui ferramentas para desenvolvimento de aplicações para microcontroladores (MOREIRA; MAIA, 2010). Os microcontroladores geralmente são embarcados, ou seja, são dedicados ao dispositivo ou sistema que controla (NOVA ELETRÔNICA, [2014?], p. 1). Eles "[...] são muito utilizados atualmente, pois a capacidade que os microcontroladores apresentam de gerenciar e otimizar as funções de dispositivos é consideravelmente alta" (NOVA ELETRÔNICA, 2016, p. 1).

A Unidade Central de Processamento (UCP) hipotética M++ foi desenvolvida por Borges (2003) e atualizada por Jung (2014), que a nomeou M+++. A UCP atualizada M+++ possui uma memória de programa, chamada de memória Read Only Memory (ROM) duas memórias Random Access Memory (RAM) (uma externa de dados, e uma interna de pilha), 16 entradas e 16 saídas (JUNG, 2014). A maioria dos simuladores eletrônicos que implementam a memória ROM e as portas lógicas são capazes de simular microcontroladores como a M+++, porém, muitas vezes não conseguem chegar à velocidade real, dependendo da complexidade do circuito e da precisão desejada (MCCLURE, 2014). Uma parte destes, por sua vez, não possui uma interface de programação direta para esta UCP simulada, visto que cada microcontrolador possui um código de máquina diferente, sendo necessário escrever diretamente o código por uma interface do software simulador, ou através de um montador externo (LENZ, [2012?]).

Diante do exposto, este trabalho apresenta uma ferramenta de auxílio no ensino-aprendizagem capaz de programar, depurar e simular o microcontrolador M+++, podendo visualizar os sinais internos do mesmo. Esta ferramenta poderá auxiliar o professor e aluno e poderá ser aplicado nas matérias de sistemas embarcados, computação digital, arquitetura de computadores, entre outras afins.

## 1.1 OBJETIVOS

O objetivo deste trabalho é implementar uma ferramenta cuja função é auxiliar os alunos e professores no ensino-aprendizagem de microcontroladores.

Os objetivos específicos são:

a) disponibilizar uma ferramenta didática para os alunos que estudam matérias de sistemas embarcados, computação digital, arquitetura de computadores, entre outras afins;
b) permitir a programação, inspeção de bugs, e uma rápida simulação (próxima ao

tempo real) do programa;

c) disponibilizar um modo de depuração na qual é possível criar *breakpoints* (pontos de parada) de forma que facilite as análises e testes do programa desenvolvido pelo aluno;
d) disponibilizar uma interface de montagem de circuitos, permitindo que sejam inseridos módulos de display de 7-segmentos, LEDs, botões e portas lógicas (AND, XOR, NOT, NOR, NAND e XNOR) ao microcontrolador, além de poder visualizar todos os sinais internos do mesmo de forma que ajude no aprendizado de circuitos eletrônicos digitais.

## 1.2 ESTRUTURA

Esta monografia está organizada em quatro capítulos. O primeiro apresenta a introdução, identificando o assunto da pesquisa e descrevendo os objetivos do projeto. O segundo capítulo descreve a fundamentação teórica, na qual é descrito a estrutura e funcionalidades da ferramenta atual utilizada, e alguns trabalhos correlatos identificados na pesquisa do mesmo. Já o terceiro capítulo apresenta o desenvolvimento da ferramenta, junto com os requisitos propostos, a sua especificação, implementação e a análise dos resultados. O quarto capitulo expõe as conclusões deste simulador e identifica algumas alternativas para pesquisas futuras relacionadas ao mesmo.

# 2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação teoria é composta por duas seções. A 2.1 descreve a máquina M+++ desenvolvida por Borges (2003) e Jung (2014). Já na 2.2 são descritos os trabalhos correlatos identificados para esta ferramenta.

## 2.1 A MÁQUINA M+++

A máquina M+++ é um microcontrolador originalmente criado por Borges (2003) e atualizado por Jung (2014) na Universidade Regional de Blumenau (FURB). Foi desenvolvida no software Logisim, e é atualmente utilizada nas aulas de Arquitetura I no curso de Ciência da Computação na FURB. O Apêndice A mostra o esquema do circuito interno da M+++ atualizado por Jung (2014).

Segundo Jung (2014), as principais características da M+++ são:

a) memória RAM com 8 bits de endereçamento;
b) memória de pilha com 8 bits de endereçamento;
c) memória ROM com 16 bits de endereçamento;
d) 4 registradores de entrada;
e) 4 registradores de saída;
f) 10 operações da ULA;
g) 5 operações de salto do programa;
h) linguagem de programação semelhante ao *assembly*;
i) *flag* End of Instruction (EOI);
j) *flag carry*;
k) *flag zero*;
l) 4 registradores com 8 bits (B – E) + acumulador com 8 bits (A).

No projeto inicial haviam 3 bits que identificavam o código de operação da ULA. O módulo de controle ganhou um bit extra no endereço de instruções do registrador, então poderiam passar a ser utilizados 15 posições. Este bit adicional é uma instrução especial, chamada High Decoder (HD), gerado automaticamente pelo montador, sendo assim, o programador *assembly* não precisa se preocupar com isso. Com a reescrita e melhorias em 2014 foi adicionado um contador com 5 bits ao módulo de controle, a partir disso pode-se usar todas as posições de instruções do registrador. O contador HD é incrementado nas chamadas sucessivas da operação High Decoder, atuando como um marcador de páginas. Se é desejado utilizar uma instrução que está na terceira página, é dado um comando de “vire a

página três vezes", e na sequência, "use a segunda instrução nesta página". A instrução `JMP inicio`, por exemplo, chama esta troca de página uma vez e em sequência utiliza a quarta instrução dela. (JUNG, 2014)

O Quadro 1 mostra a relação entre cada byte das instruções e como elas são divididas. Como é um microcontrolador de 8 bits e apenas 3 destes são usados para a operação, não caberiam todas as funções necessárias em um único byte, sendo necessário a utilização do HD.

Quadro 1 - Separação das instruções

| Operação ULA | Registrador / Porta de entrada ou saída | Fluxo de operação |
|---|---|---|
| 3 bits | 2 bits | 3 bits |

Fonte: adaptado de Jung (2014).

O Quadro 2 mostra os registradores disponíveis junto com o código binário apropriado. Note que o mesmo corresponde aos registradores internos, as portas de entrada e as de saída. Esta escolha entre cada tipo é feita pelo fluxo de operação. Também vale destacar que o acumulador não faz parte desta listagem, pois o mesmo encontra-se fora do banco de registradores e está ligado diretamente à ULA. Nota-se também que todos eles operam com um conjunto 8 bits. (JUNG, 2014)

Quadro 2 - Endereços dos registradores

| Código | Nome |
|---|---|
| 00 | B / IN0 / OUT0 |
| 01 | C / IN1 / OUT1 |
| 10 | D / IN2 / OUT2 |
| 11 | E / IN3 / OUT3 |

Fonte: adaptado de Jung (2014).

O Quadro 3 apresenta o conjunto de operações da ULA, assim como o código binário correspondente e a descrição da mesma. Destaca-se também que estas funções sempre são executadas internamente pela unidade independente se é desejado fazer alguma operação com a ULA ou não, esta, por sua vez, apenas joga os valores no barramento de dados caso seja necessário. Também vale destacar que algumas operações utilizam de um ou dois valores adicionais para a sua execução, como é o caso das operações `NOT`, `MOV` e `INC`. (JUNG, 2014)

Quadro 3 - Conjunto de operações da ULA

| Binário | Operação | Descrição |
|---|---|---|
| 000 | ADD | Faz a soma |
| 001 | SUB | Faz a subtração |
| 010 | AND | Faz a operação E |
| 011 | OR | Faz a operação OU |
| 100 | XOR | Faz a operação OU EXCLUSIVO |
| 101 | NOT | Faz a operação NÃO |
| 110 | MOV | Faz a cópia do valor |
| 111 | INC | Incrementa em 1 o valor |

Fonte: adaptado de Jung (2014).

O Quadro 4 expõe os fluxos de controle do microcontrolador, assim como os respectivos códigos de página, código binário e uma descrição detalhada. Nota-se que para o mesmo código binário existem mais de uma instrução, fazendo-se necessário o uso do *high decoder*.

Quadro 4 - Conjunto de fluxo de operação

| Página de código | Binário | Fluxo | Descrição |
|---|---|---|---|
| 00000 | 000 | A*A → A | Faz a operação da ULA do acumulador com o acumulador, e o resultado desta operação é copiado para o acumulador. |
| 00000 | 001 | A*A → reg | Faz a operação da ULA do acumulador com o acumulador, e o resultado desta operação é copiado para o registrador. |
| 00000 | 010 | A*A → RAM | Faz a operação da ULA do acumulador com o acumulador, e o resultado desta operação é copiado para a memória RAM. |
| 00000 | 011 | A*A → OUT | Faz a operação da ULA do acumulador com o acumulador, e o resultado desta operação é copiado para a saída. |
| 00000 | 100 | A*reg → A | Faz a operação da ULA do acumulador com o registrador, e o resultado desta operação é copiado para o acumulador. |
| 00000 | 101 | A*RAM → A | Faz a operação da ULA do acumulador com a memória RAM, e o resultado desta operação é copiado para o acumulador. |
| 00000 | 110 | A*IN → A | Faz a operação da ULA do acumulador com a entrada, e o resultado desta operação é copiado para o acumulador. |
| 00000 | 111 | *High decoder* | Indicador de troca de página. Esta operação pode ser chamada independente da página de código atual. |
| 00001 | 000 | A*ROM → A | Faz a operação da ULA do acumulador com um valor especificado, e o resultado desta operação é copiado para o acumulador. |
| 00001 | 001 | A*ROM → reg | Faz a operação da ULA do acumulador com um valor especificado, e o resultado desta operação é copiado para o registrador. |
| 00001 | 010 | A*ROM → RAM | Faz a operação da ULA do acumulador com um valor especificado, e o resultado desta operação é copiado para |

| | | | |
|---|---|---|---|
| | | | a memória RAM. |
| 00001 | 011 | JMP <end> | Salta para o endereço especificado. |
| 00001 | 100 | JMPC <end> | Salta para o endereço especificado se a *flag carry* estiver ativa. |
| 00001 | 101 | JMPZ <end> | Salta para o endereço especificado se a *flag zero* estiver ativa. |
| 00001 | 110 | CALL <end> | Chama o procedimento no endereço. |
| 00010 | 000 | RET | Retorna do procedimento. |
| 00010 | 001 | A*DRAM → A | Faz a operação da ULA do acumulador com o endereço da memória ram dinâmico (escolhido pelo valor do registrador especificado), e o resultado desta operação é copiado para o acumulador. |
| 00010 | 010 | A*A → DRAM | Faz a operação da ULA do acumulador com o acumulador, e o resultado desta operação é copiado para o endereço da memória ram dinâmico (escolhido pelo valor do registrador especificado). |
| 00010 | 011 | PUSH <reg> | Adiciona o valor do registrador na memória de pilha. |
| 00010 | 100 | POP <reg> | Retira o valor da pilha e copia para o registrador. |
| 00010 | 101 | PUSHA | Adiciona o valor do acumulador na memória de pilha. |
| 00010 | 110 | POPA | Retira o valor da pilha e copia para o acumulador. |

Fonte: adaptado de Jung (2014).

A chamada do fluxo de operação do HD pode ser chamada consecutivas vezes (JUNG, 2014). Ressalta-se que os códigos de salto, como `JMP`, `JMPC`, `JMPZ` e `CALL`, possuem dois bytes a mais para indicar o endereço na memória ROM para onde deverá fazer o salto. O primeiro byte guarda o valor “alto”, ou mais significativo, e o segundo byte guarda o valor “baixo”, ou menos significativo. O mesmo acontece para alguns fluxos de operação que utilizam endereços da memória RAM ou dados da ROM, como o comando `MOV 42H, A`, porém, estes comandos precisam de apenas um byte a mais. O Quadro 5 mostra uma relação de alguns códigos válidos da M+++ e seus respectivos códigos em binário.

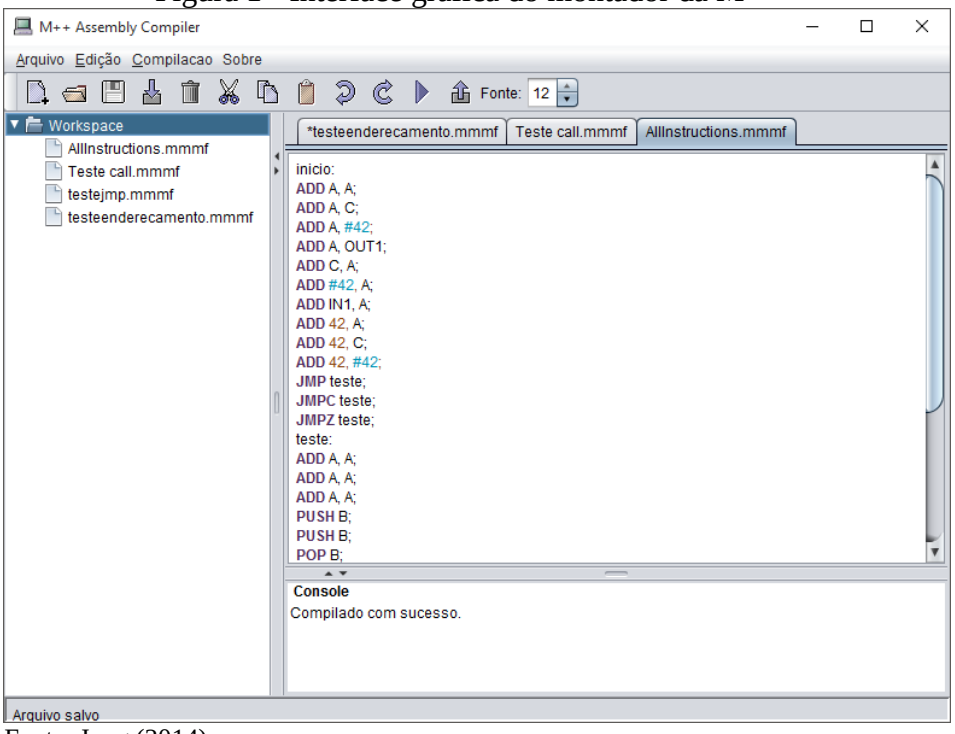
Quadro 5 - Exemplos de códigos e binários

| Código | Binário | Descrição |
|---|---|---|
| PUSH C | 000 00 111 | *High decoder* |
| | 000 00 111 | *High decoder* |
| | 000 01 011 | Instrução PUSH com o registrador C |
| POP D | 000 00 111 | *High decoder* |
| | 000 00 111 | *High decoder* |
| | 000 10 100 | Instrução POP com o registrador D |
| JMP #0056H | 000 00 111 | *High decoder* |
| | 000 00 011 | Instrução JMP |
| | 000 00 000 | Endereço - parte alta: 0 em hexadecimal |
| | 010 10 110 | Endereço - parte baixa: 56 em hexadecimal |
| MOV 42H, A | 000 00 111 | *High decoder* |
| | 110 00 000 | Operação MOV A*ROM → A |
| | 010 00 010 | Valor 42 em hexadecimal |
| INC D, A | 111 10 100 | Operação INC A*Reg → A com o registrador D |
| AND E, A | 010 11 100 | Operação AND A*Reg → A com o registrador E |

Fonte: elaborador pelo autor.

Jung (2014) também desenvolveu um montador para ser utilizado na máquina M+++. Ele funciona como uma pasta de códigos, chamada de `workspace`. A Figura 1 ilustra a interface gráfica do mesmo.

Figura 1 - Interface gráfica do montador da M+++




Fonte: Jung (2014).

## 2.2 TRABALHOS CORRELATOS

Foram analisadas quatro ferramentas com características semelhantes aos principais objetivos da proposta. A seção 2.2.1 detalha o EDSIM51, desenvolvido e mantido por James Rogers, o item 2.2.2 relata sobre o PIC Simulator IDE, uma ferramenta criada pela

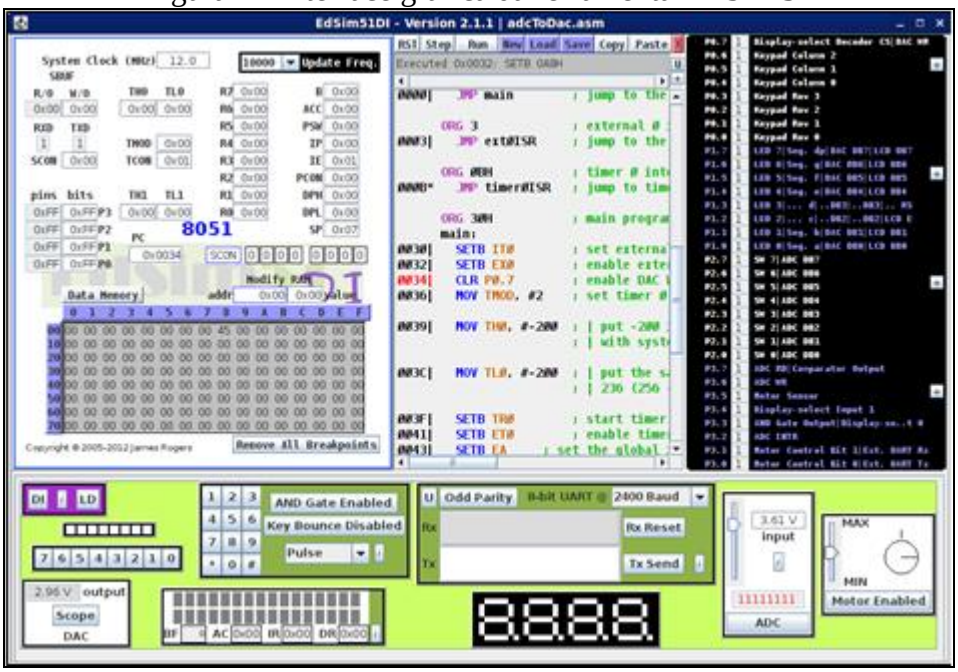
OshonSoft, o item 2.2.3 descreve o GPSIM, desenvolvido e mantido por Franklin, Rankin e Dattalo, já o item 2.2.4 mostra sobre o modelo online da UCP criado por Scott e Filbinger (2016).

### 2.2.1 EDSIM51

O EDSIM51 é uma ferramenta gratuita, desenvolvida por James Rogers, “diferente de muitos simuladores dos padrões de indústria que estão amplamente disponíveis, este tem o aluno em mente.” (ROGERS, 2013, p. 1, tradução nossa). Ela é focada no microcontrolador 8051 e nela é possível programar em assembly, percorrer pelo código, podendo colocar *breakpoints*, e observar os efeitos para cada linha na memória interna e nos periféricos externos (ROGERS, 2016).

A ferramenta não possibilita a edição do circuito eletrônico, tampouco a visualização do circuito dos registradores (somente os valores) e portas lógicas internamente, porém possui vários módulos que podem ser ligados ao microcontrolador, dentre eles estão: LEDs, chaves, conversor analógico-digital, teclado numérico, entre outros (ROGERS, 2016). Segundo o guia do usuário do simulador, é possível escolher a frequência de *clock* e a de atualização da tela, porém, este não deixa claro se consegue chegar próximo ao tempo real com a frequência padrão (12MHz) (ROGERS, 2013). A Figura 2 exemplifica a interface da ferramenta EDSIM51 com todos os módulos na parte inferior.

Figura 2 - Interface gráfica da ferramenta EDSIM51



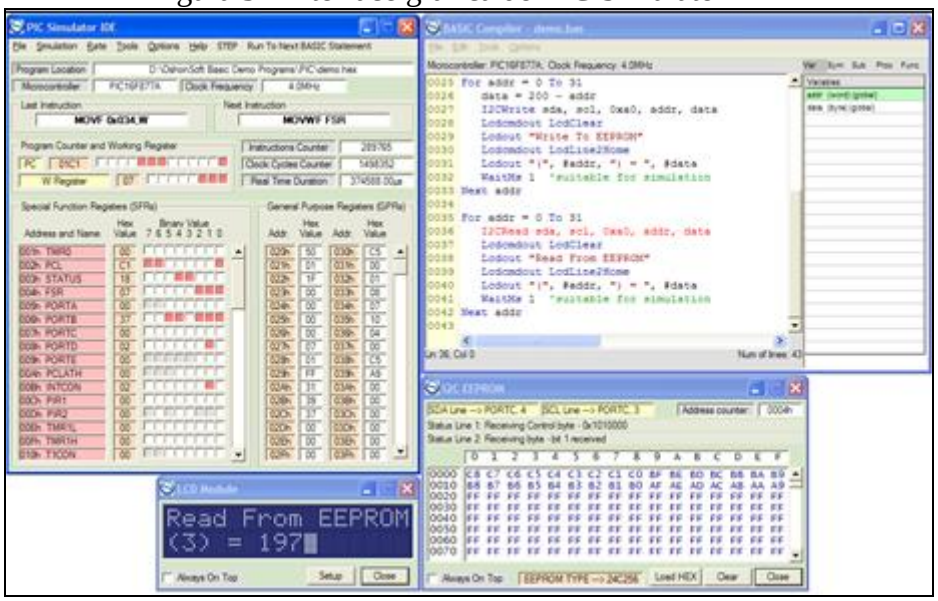

Fonte: Rogers (2016).

### 2.2.2 PIC Simulator IDE

Desenvolvido pela OshonSoft (2016), o PIC Simulator IDE é um software que integra um simulador, compilador, montador, desmontador e depurador. A ferramenta é voltada para a linha de produtos da arquitetura PIC de 8-bits da Microchip, sendo compatível com modelos PIC16F, PIC12F e PIC10F. Assim como o simulador anterior apresentado, este não possibilita a edição do circuito eletrônico e a visualização dos componentes internos do microcontrolador, também possui alguns módulos que podem ser adicionados ao microcontrolador, dentre eles estão: display de 7-segmentos, display gráfico LCD, memória externa, motor de passos, entre outros (OSHONSOFT, 2016).

Também é possível fazer a depuração do código podendo inserir *breakpoints*, sendo possível programar e compilar diretamente na ferramenta na linguagem Basic. O desenvolvedor não deixa claro a que velocidade pode chegar na simulação, mas mostra que para a versão *extremely fast* depende da velocidade do computador, sendo que na versão *ultimate* é possível chegar a velocidades ainda maiores (OSHONSOFT, 2016). A Figura 3 exemplifica a interface gráfica do PIC Simulator IDE assim como a programação em Basic utilizada para a programação deste microcontrolador.

Figura 3 - Interface gráfica do PIC Simulator IDE




Fonte: Oshonsoft (2016).

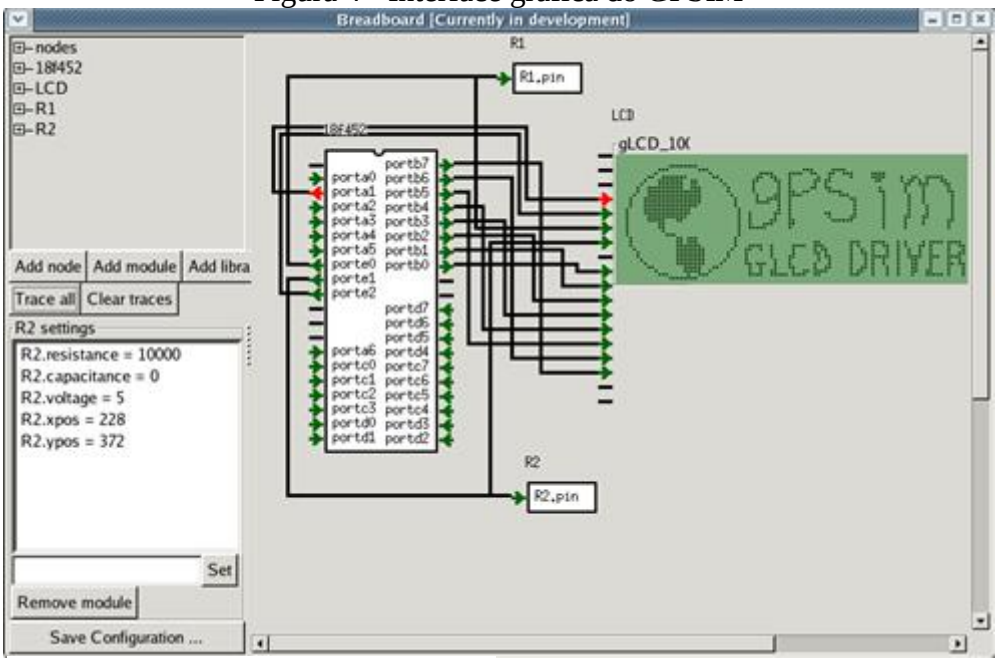
### 2.2.3 GPSIM

O GPSIM é uma ferramenta de simulação para microcontroladores PIC desenvolvida por Franklin, Rankin e Dattalo (2015) e está sendo distribuído sob licença GNU GPL. Foi desenhado para ser o mais preciso possível, desde o núcleo até os pinos de entrada e saída,

incluindo todos os periféricos internos. O simulador possui um inspetor de código fonte, porém, não é possível editá-lo por esta ferramenta, sendo necessária uma aplicação externa para programá-lo. Também não é possível visualizar os sinais internos do chip, tampouco os registradores e portas lógicas, porém, ele é capaz de editar o circuito eletrônico fora do mesmo, podendo adicionar portas lógicas, LEDs, displays de 7-segmentos, chaves, entre outros (FRANKLIN; RANKIN; DATTALO, 2015).

Segundo Franklin, Rankin e Dattalo (2015), a velocidade da simulação é capaz de se aproximar ao real (25MHz em um PII 400MHz), entretanto, com a adição e interação de vários componentes a velocidade diminui. A Figura 4 exemplifica a edição do circuito, assim como um exemplo da utilização do módulo de display LCD.

Figura 4 - Interface gráfica do GPSIM




Fonte: Franklin, Rankin e Dattalo (2015).

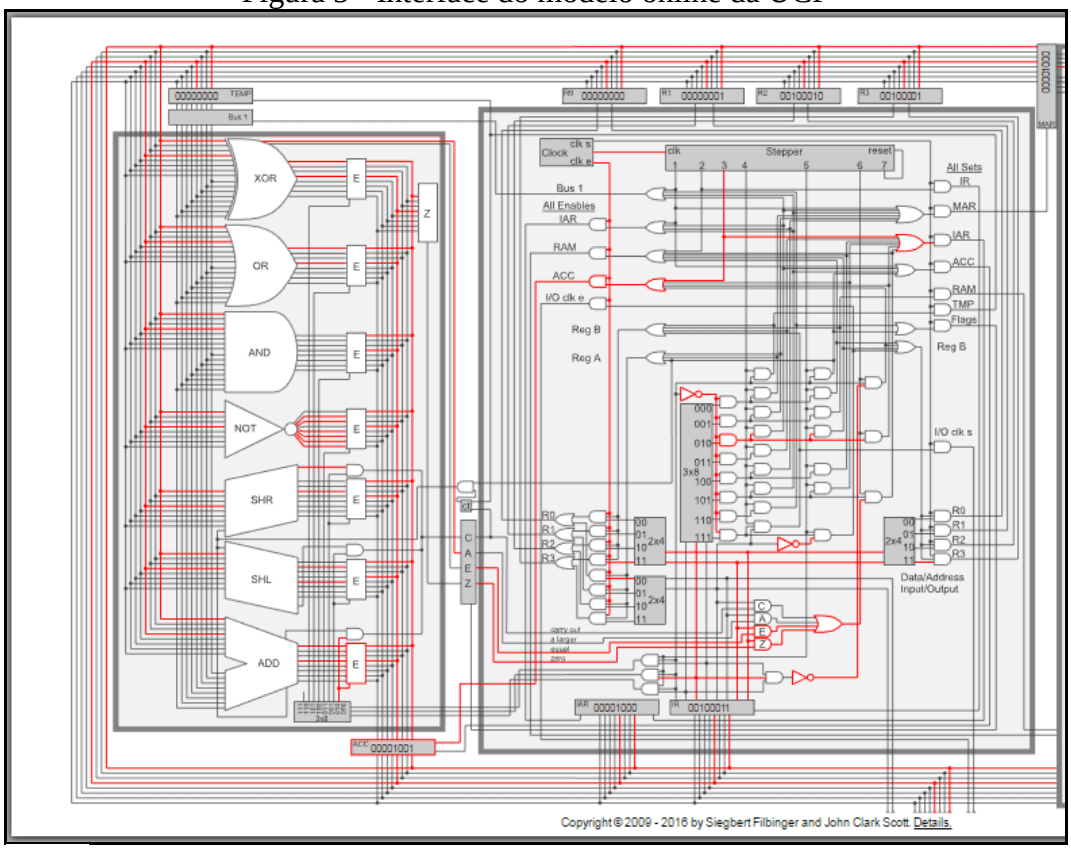
### 2.2.4 But How Do It Know CPU Online Model

O modelo da UCP desenvolvido por Scott e Filbinger (2016) possui uma versão online e uma versão desenvolvida no Microsoft Excel. Estes dois modelos foram desenvolvidos para auxiliar no aprendizado de arquitetura de computadores e áreas afins, além de não serem vendidos comercialmente (exceto o livro). No modelo online, é possível programar o microcontrolador por um menu lateral e após a programação, é gerado o código binário correspondente desse programa na parte inferior.

A simulação deste modelo permite uma velocidade máxima de até 10Hz e é possível visualizar todos os sinais internos deste microcontrolador. Além disso, também não é possível inserir *breakpoints* no programa, dificultando a sua análise e tampouco editar este modelo

online do circuito, e não possui quaisquer componentes externos. Porém, no livro de Scott (2009), são exemplificados a construção de vários componentes externos, como um display, um teclado, entre outros. A Figura 5 apresenta o circuito interno deste microcontrolador mostrando os sinais internos.

Figura 5 - Interface do modelo online da UCP




Fonte: Scott e Filbinger (2016).

# 3 DESENVOLVIMENTO DA FERRAMENTA

O desenvolvimento deste trabalho foi dividido em quatro etapas. A seção 3.1 identifica os requisitos funcionais e não funcionais da ferramenta. Já a 3.2 descreve a especificação mostrando os diagramas empregues. Na sequência, a 3.3 exibe a implementação do software, dividindo-se em Técnicas e ferramentas utilizadas, Ambiente de programação e depuração e Ambiente de circuito. Por fim, a 3.4 faz a análise dos resultados que foram obtidos.

## 3.1 REQUISITOS

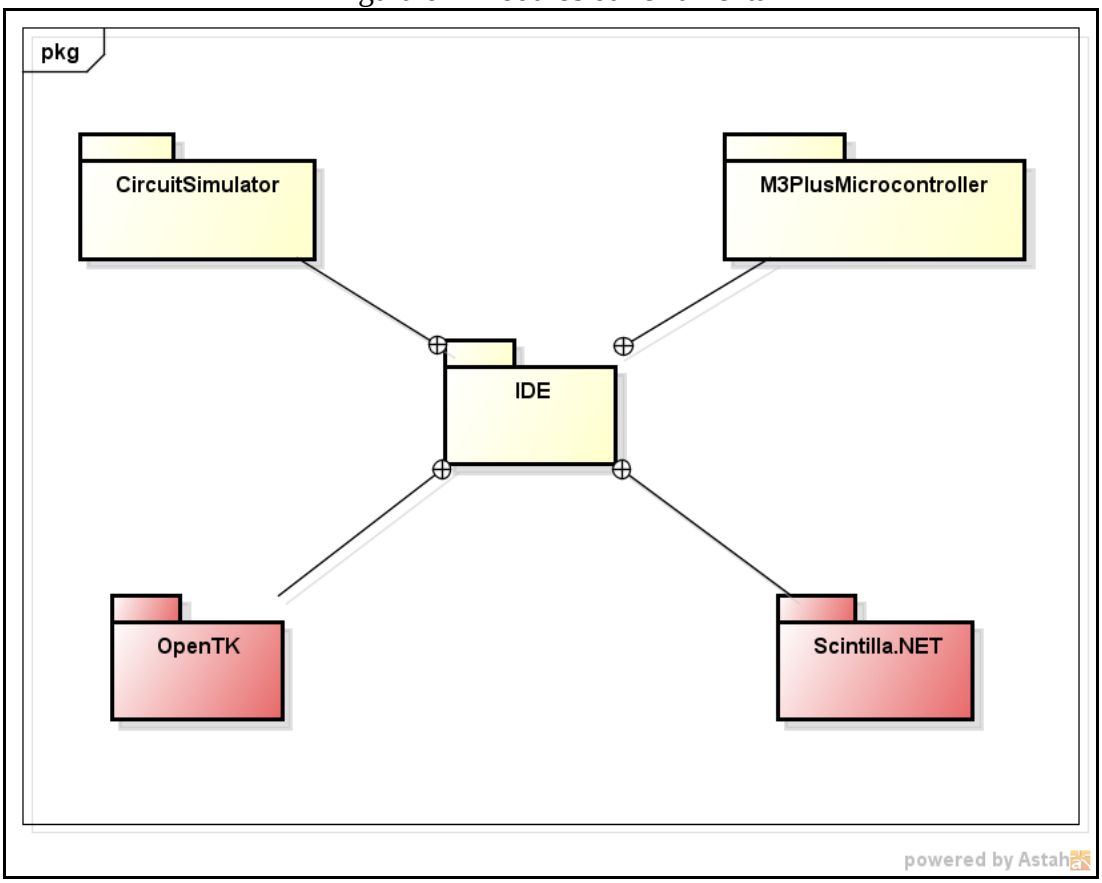
A seguir, são descritos os requisitos funcionais (RF) e os requisitos não funcionais (RNF) da ferramenta proposta:

a) salvar e carregar projetos (RF);
b) abrir arquivos já compilados (RF);
c) programar e montar os programas na mesma ferramenta (RF);
d) disponibilizar a depuração do software com pontos de parada (breakpoints) (RF);
e) possibilitar a configuração da velocidade de depuração (RF);
f) permitir a construção de circuitos eletrônicos (RF);
g) visualizar os sinais internos do microcontrolador (RF);
h) ser implementado na linguagem C# (RNF);
i) ter compatibilidade com o sistema operacional Windows 7 (RNF);
j) rodar em tempo real na velocidade de *clock* de 1 MHz (RNF).

## 3.2 ESPECIFICAÇÃO

A ferramenta foi dividida em cinco módulos. A Figura 6 apresenta um diagrama representando os mesmos. Nota-se que as extensões `OpenTK` e `Scintilla.NET`, são os *frameworks* de terceiros, já o `CircuitSimulator`, `M3PlusMicrocontroller` e `IDE` são pacotes/soluções desenvolvidas neste trabalho. O `IDE` importa todas as outras, sendo assim, os outros módulos podem ser gerados separadamente da aplicação principal (`IDE`) substituindo apenas a Dynamic-link library (DLL) compilada.

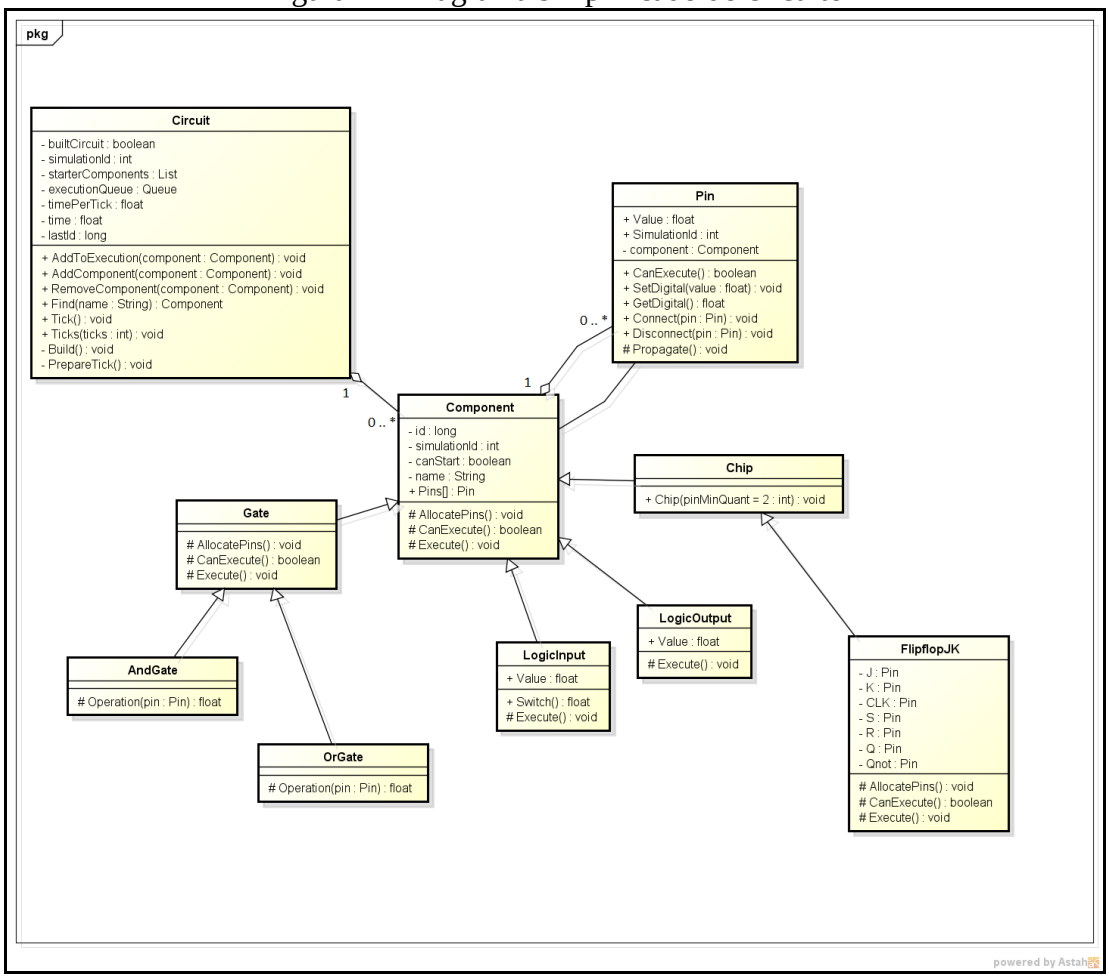
Figura 6 – Módulos da ferramenta




Fonte: elaborado pelo autor.

No desenvolvimento do simulador de circuitos, identificado por `CircuitSimulator` na Figura 6, foi necessário criar três classes principais, são elas: `Circuit`, `Component` e `Pin`. Todos os componentes eletrônicos presentes no simulador herdam de `Component`, e alguns deles de outras classes intermediárias. A Figura 7 exemplifica um diagrama de classes simplificado (algumas classes foram omitidas) com alguns componentes do módulo de circuitos. A classe `Circuit` é a base da simulação, nela contém a referência para todos os componentes eletrônicos, uma fila de execução e mais algumas informações que são esclarecidas na seção 3.3.3.

Figura 7 – Diagrama simplificado do circuito




Fonte: elaborado pelo autor.

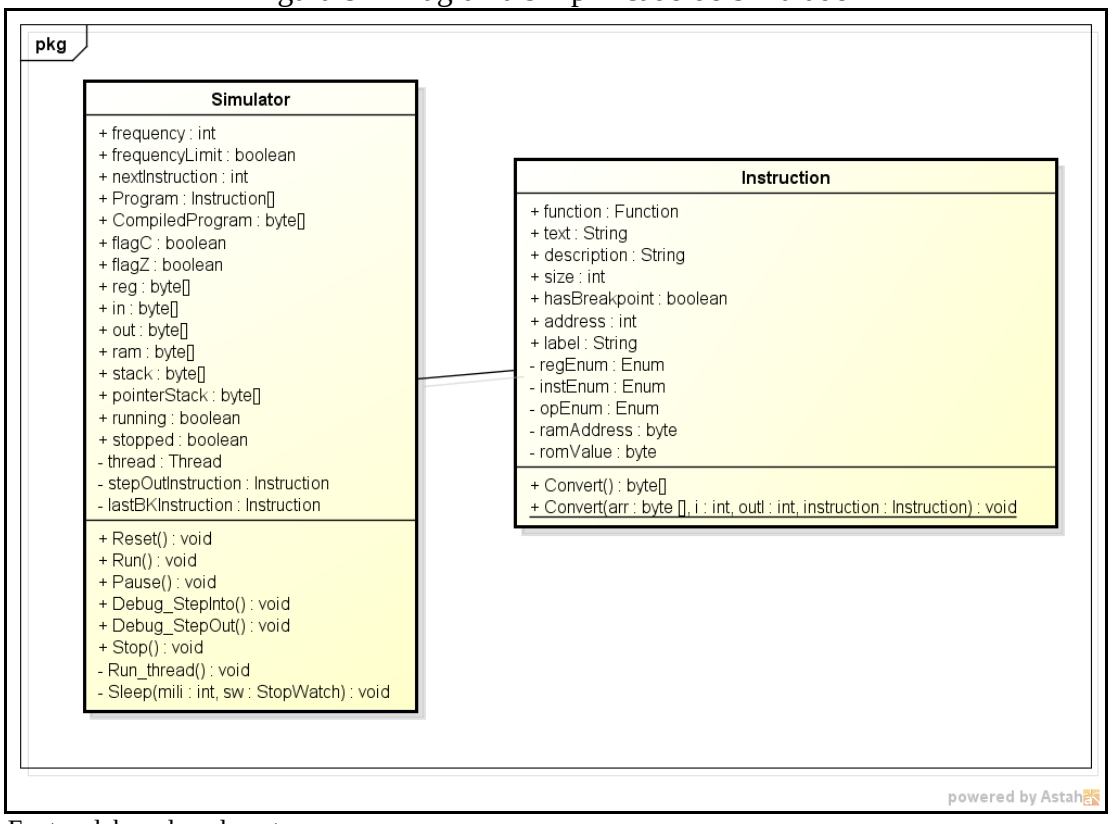
A Figura 8 ilustra o interpretador do microcontrolador, identificado por `M3PlusMicrocontroller` na Figura 6. Foi necessário implementar apenas duas classes principais: `Simulator`, que faz a simulação em si e guarda o estado do microcontrolador, e `Instruction`, que guarda o procedimento a ser executado pelo simulador e outros dados necessários para a execução e depuração. Vale destacar que, ao chamar o método `Run` da classe `Simulator`, esta cria uma `thread` que executa a sequência de instruções em paralelo. Esta `thread`, além de chamar as funções, também utiliza dos métodos `Sleep` para fazer a sincronia da frequência desejada de execução e `Pause` quando a próxima instrução a ser executada está com a *flag* `hasBreakpoint` ativa.

A parte mais importante da classe `Instruction` é a variável `function`, que é executado pelo `Simulator`. Esta variável, na qual é um método do tipo `delegate`, muda o estado do `Simulator`, podendo alterar os valores das seguintes variáveis:

a) `nextInstruction`: usado como ponteiro para a próxima instrução. É alterada quando são utilizadas operações como `JMP`, `JMPC`, `JMPZ`, `CALL` e `RET`;
b) `flagC`: usada para indicar a *flag carry* da máquina. Todas as operações que utilizam a ULA modificam esta variável, e a operação `JMPC` lê o conteúdo da mesma;
c) `flagZ`: usada para indicar a *flag zero* da máquina. Todas as funções que usam a ULA modificam ela, e a operação `JMPZ` faz a leitura desta *flag*;
d) `reg`: são os registradores A, B, C, D e E da máquina. Praticamente todas as funções usam os valores desta variável;
e) `in`: são as entradas da máquina M+++, podendo ser alterado pelo simulador de circuitos ou pela interface de depuração;
f) `out`: são as saídas da máquina M+++;
g) `ram`: é um conjunto de bytes de toda a memória RAM;
h) `stack`: é um conjunto de bytes de toda a memória de pilha. As instruções que utilizam a pilha e as de salto leem ou alteram os valores desta variável;
i) `pointerStack`: é o apontador da pilha. Ele é alterado toda vez que ela é utilizada.

Destaca-se também os dois métodos `Convert` da classe `Instruction`. O primeiro deles é utilizado para a conversão da instrução para código binário, na qual é usado para a memória ROM da M+++ e para exportação do código para as outras versões. Já o segundo método é a operação inversa, na qual recebe os bytes e a converte para ser utilizada na interpretação do simulador.

Figura 8 – Diagrama simplificado do simulador




Fonte: elaborado pelo autor.

## 3.3 IMPLEMENTAÇÃO

A implementação da ferramenta é dividida em quatro seções: Técnicas e ferramentas utilizadas, Ambiente de programação e depuração, Ambiente de circuito e Problemas e melhorias que são descritas na sequência.

### 3.3.1 Técnicas e ferramentas utilizadas

O software mais utilizado para a implementação desta ferramenta foi o Microsoft Visual Studio Community 2015 e em algumas ocasiões, foi necessário a manipulação das macros do software Notepad++ para facilitar algumas partes da programação da ferramenta. Também foi feito o versionamento dos arquivos deste projeto, utilizando o sistema de versionamento GIT pelo site github.com. Todas as entregas parciais e publicação de testes e versões finais estão disponíveis nesta plataforma.

Foram utilizados dois frameworks de terceiros: Scintilla.NET, que é empregado nos ambientes de programação e depuração para fazer o realce da sintaxe, e o OpenTK, que é aplicado para fazer os gráficos e desenhos no ambiente de circuito. Além disso, foi usado o software Astah Community e Adobe Illustrator para fazer os diagramas.

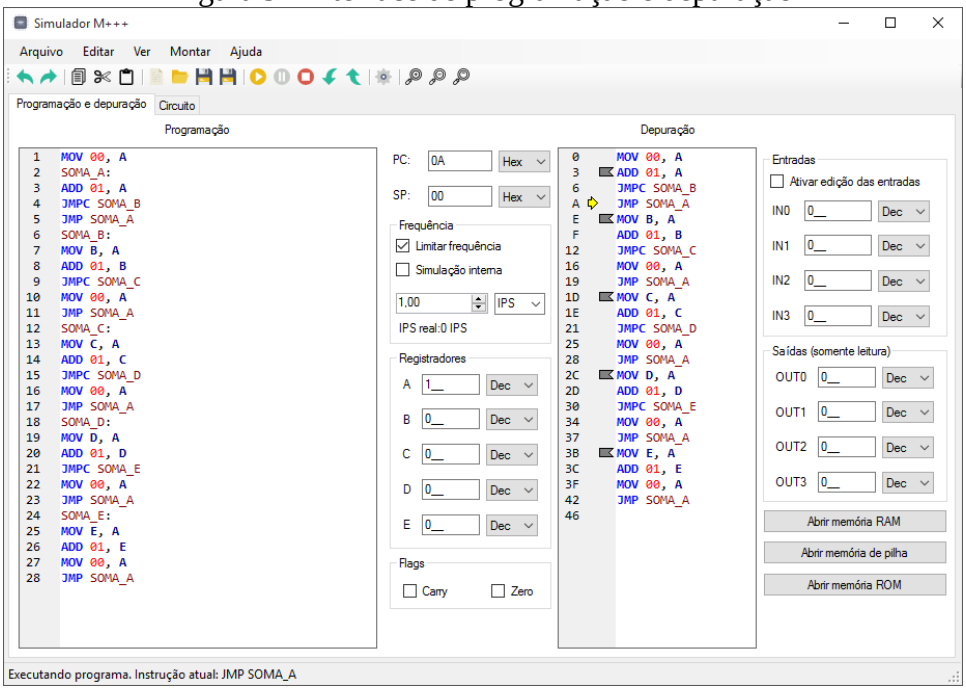
### 3.3.2 Ambiente de programação e depuração

Na etapa do desenvolvimento do painel de programação do projeto foi utilizado o porte de framework chamado Scintilla, desenvolvido por Hodgson (2003) e portado por Slusser (2016). Este framework é responsável por fazer o realce da sintaxe (também conhecido por *syntax highlighting*) no código, mostrar as linhas, o número do endereço, e criar uma borda ao lado do código para a adição de *breakpoints* e *labels*. Com isso foi desenvolvido um analisador léxico próprio para o Scintilla, para que, conforme o programador for programando, este vai adicionando o estilo correspondente àquele código. Os estilos empregados foram: Padrão, InstrucaoCPU, Registrador, Identificador, Numero, Endereco e Comentario. Na Figura 9 pode-se observar alguns desses estilos aplicados, como o Identificador, InstrucaoCPU, Numero e Registrador.

Esta interface permite a inserção de pontos de parada no programa clicando ao lado direito do número da linha. Ao montar o programa, eles são enviados para a tela de depuração e adicionados no endereço correspondente àquela instrução/linha com o ponto de parada. A interface de programação também possibilita a inserção de comentários no meio do programa facilitando a leitura do mesmo. Todo o texto que há após o caractere `;` (ponto e vírgula) ou de `//` (duas barras) até o final da linha são considerados como comentário. Comentários e linhas em branco são ignoradas pelo interpretador e compilador.

Na implementação do painel de depuração foi necessário desenvolver um emulador (interpretador da M+++) e um montador. A Figura 9 mostra a interface gráfica deste ambiente ao lado direito, no qual, é possível visualizar o código montado sem comentários, espaços extras, tabulações e linhas em branco. Pode-se adicionar pontos de parada com o programa já montado, clicando ao lado direito dos endereços e os *labels* são visualizados clicando nos marcadores em cinza ao lado esquerdo das instruções. Nesta mesma região dos *breakpoints* também há uma seta em amarelo indicando qual será a próxima instrução a ser executada, o programa é pausado sempre que esta seta se encontra com algum ponto de parada.

Figura 9 - Interface de programação e depuração

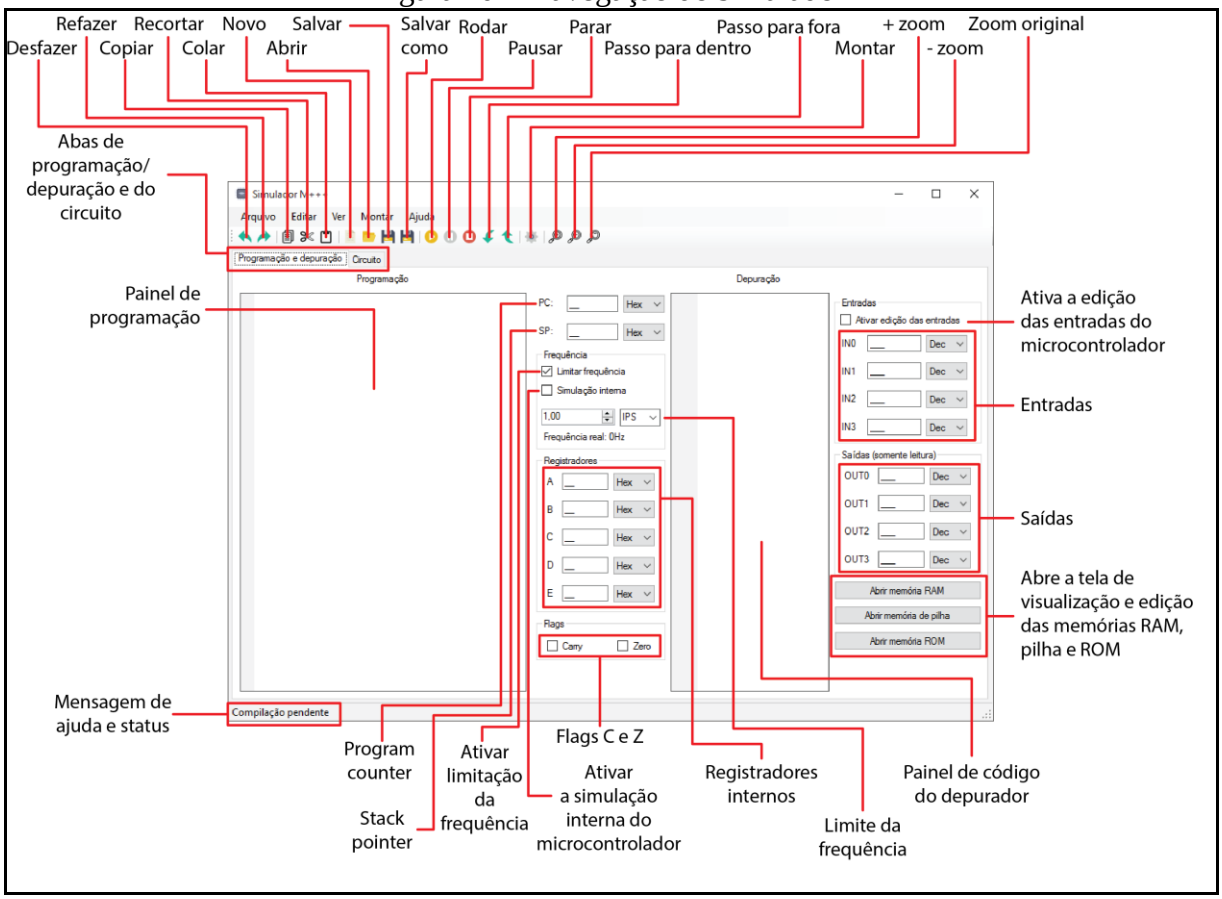



Fonte: elaborado pelo autor.

É possível visualizar e editar alguns registradores, *flags* e velocidade de execução ao lado esquerdo deste ambiente. Além disso, selecionando o *checkbox* `Simulação interna` faz com que o emulador passe a simular o circuito interno da M+++. A velocidade desta foi limitada a 20 Hertz (Hz) devido a alguns problemas de desempenho que podem ocorrer e comprometer os dados. Nota-se que ao clicar neste botão, a caixa de seleção ao lado, muda de Instruções por Segundo (IPS) para Hz, isto se deve ao fato de que a interpretação do programa não conta quantos pulsos de *clock* são necessários para executar cada instrução, diferente da simulação interna (e real) da máquina M+++.

O limite de frequência pode ser desativado, caso não esteja selecionado o *checkbox* `Simulação interna`, fazendo com que a *thread* principal da simulação não passe pelos comandos de *delay*, operando na velocidade máxima que o computador é capaz de atingir. Registradores de entrada são editados somente se a *checkbox* `Ativar edição das entradas` estiver ativa, caso contrário, elas são modificadas pelo Ambiente do Circuito. É possível também visualizar e editar os valores da memória RAM e pilha clicando nos botões `Abrir memória RAM` e `Abrir memória de pilha` no canto inferior direito. Não é permitido editar a memória ROM, somente a visualização da mesma, clicando no botão `Abrir memória ROM`. A Figura 10 exemplifica a navegação do simulador.

Figura 10 – Navegação do simulador




Fonte: elaborado pelo autor.

Laureano (2006) classifica os emuladores em quatro tipos básicos: totalmente baseada em hardware, parcialmente baseada em hardware, parcialmente baseada em software e totalmente baseada em software. Para o desenvolvimento da ferramenta, foi utilizado o tipo totalmente baseado em software, pois de acordo com Laureano (2006, p. 20)

> O emulador não precisa de nenhum hardware para prover a emulação, ou seja, o software provê todos os recursos para isso. Esse tipo de emulador é o mais popular pela adaptabilidade (o código-fonte de um emulador pode ser aproveitado para emular várias plataformas parecidas) e portabilidade. Emuladores de plataformas Atari e Amiga são alguns exemplos. (LAUREANO, 2006, p. 20)

Assim sendo, para a construção do emulador primeiramente foram necessárias três análises que são muito comuns em compiladores: análise léxica, sintática e semântica. Assim que o programador conclui o código e clica na ação `Analisar e montar`, a primeira etapa a ser feita é passar pela análise léxica. Segundo Aho et al. (2010, p. 70), a "tarefa principal do analisador léxico é ler os caracteres da entrada do programa fonte, agrupá-los em lexemas e produzir como saída uma sequência de *tokens* para cada lexema no programa fonte" (AHO et al., 2010, p. 70).

Os *tokens* são categorizados de acordo com os tipos de instruções, registradores, dados etc. São eles:

a) entradas: `IN0`, `IN1`, `IN2` e `IN3`;
b) saídas: `OUT0`, `OUT1`, `OUT2` e `OUT3`;
c) registradores: `A`, `B`, `C`, `D` e `E`;
d) instruções da UCP: `ADD`, `SUB`, `AND`, `OR`, `XOR`, `NOT`, `MOV`, `INC`, `JMP`, `JMPC`, `JMPZ`, `CALL`, `RET`, `PUSH`, `POP`, `PUSHA` e `POPA`;
e) separador de fluxo de controle: caractere `,` (vírgula);
f) separador de identificador: caractere `:` (dois pontos);
g) número: números em hexadecimais com 2 caracteres, com 3 caracteres sendo o último `h`, decimais com mais de 3 caracteres podendo ou não o último ser `d`;
h) endereço: números em hexadecimais com 3 caracteres, com 4 caracteres sendo o último `h`, decimais com mais de 4 caracteres podendo ou não o último ser `d`. Em todas estas situações, o primeiro caractere deve ser `#` para ser considerado como endereço.

Os *tokens*, diferentemente da versão anterior, não são *case-sensitive*, entretanto, anteriormente era permitido o salto para um endereçamento direto, algo semelhante a isto: `JMP #0010`. Esta versão não permite este tipo de instrução, pois caso seja feito o endereçamento incorreto, pode causar bugs e diferenças na simulação comparado com uma máquina real. Além disso, também foi retirada a necessidade de um ponto e vírgula no final de cada instrução. Após ter separado todos os *tokens* e categorizados, a etapa em sequência é a análise sintática, ela verifica se os tipos dos *tokens* gerados na etapa anterior estão na sequência correta para a geração dos procedimentos do interpretador e do código binário. Segundo Aho et al. (2010, p. 122),

> o analisador sintático recebe do analisador léxico uma cadeia de *tokens* representando o programa fonte, [...] e verifica se essa cadeia de *tokens* pertence à linguagem gerada pela gramática. O analisador deve ser projetado para emitir mensagens para quaisquer erros de sintaxe encontrados no programa. (AHO et al., 2010, p. 122)

A próxima etapa é a análise semântica, na qual verifica se a ação relacionada com a instrução está correta. Nela normalmente incluem a verificação de tipos, de fluxo de controle e da unicidade da declaração de variáveis (RICARTE, 2003). Neste caso, a única análise

semântica necessária é a verificação de existência dos *labels*. Ex.: `JMP REPETE`. Internamente, estes *labels* são subtituidos por endereços. Segundo Marangon (2017, p. 1),

> a análise semântica é responsável por verificar aspectos relacionados ao significado das instruções, essa é a terceira etapa do processo de compilação e nesse momento ocorre a validação de uma serie de regras que não podem ser verificadas nas etapas anteriores. (MARANGON, 2017, p. 1)

A última etapa de um compilador é a geração de código objeto (MARANGON, 2017). Para esta geração foi desenvolvido um procedimento correspondente para cada instrução e fluxo de controle. Nestas funções são colocadas as características de cada instrução, tais como a descrição, e um `delegate`. Os `delegates` da linguagem C# são similares ao conceito de ponteiros para funções usados nas linguagens C e C++ (MACORATTI, 2011). Estes possuem a implementação da instrução em si, como pode ser observado no Quadro 6. Cada um destes procedimentos é uma ação semântica (ou uma instrução) que serão executadas assim que o programador iniciar o programa.

Quadro 6 - Exemplo de instrução

```
1  public delegate void Function();
2  public class Instruction {
3  public Function function = delegate() { };
4  //...
5  //Instrução MOV B, A
6  instruction.function = delegate () {
7        simulator.Reg[0] = simulator.Reg[1];
8        simulator.Flag_Z = simulator.Reg[0] == 0;
9        simulator.Flag_C = false;
10 };
11 //...
12 public class Simulator{
13       Instruction[] Instructions;
14 //...
15 }
```

Fonte: elaborado pelo autor.

Fayzullin (2000), exemplifica como criar um emulador de uma UCP, e mostra que estes simuladores se apropriam de uma técnica que converte os *opcodes* em instruções reais para a linguagem de programação utilizada. Porém, ele emprega um `switch` de cada *opcode* para a execução. No caso do simulador da M+++ foi feito um `array` com os procedimentos já montados. Antes de fazer dessa maneira, era proposto que, ao rodar/compilar o programa, o simulador geraria um código intermediário (era imaginado em C# também). O resultado disso tem todos os procedimentos da M+++ em sequência de forma que se pareça com um programa normal em C#, algo semelhante ao Quadro 7.

Quadro 7 - Exemplo anteriormente proposto das intruções

```
1 //OR B, A
2 Reg[0] = (byte)(Reg[1] | Reg[0]);
3 Flag_Z =Reg[0] == 0;
4 Flag_C = false;
5 //XOR 42, C
6 Reg[2] = (byte)(42 ^ Reg[0]);
7 Flag_Z = Reg[2] == 0;
8 Flag_C = false;
```

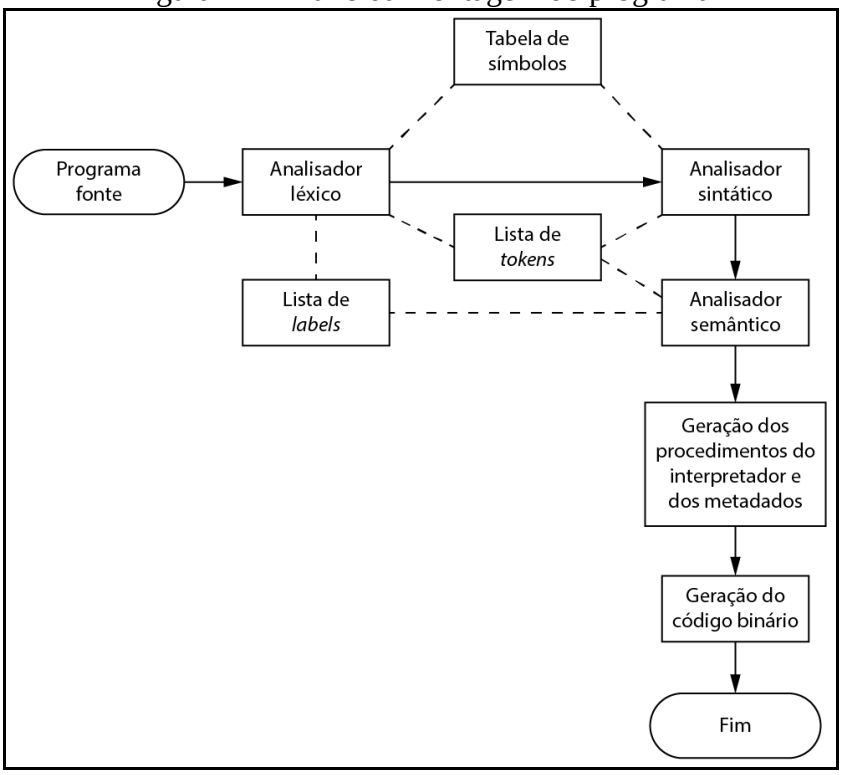
Fonte: elaborado pelo autor.

Após gerar o programa, compilar e executar, seria anexado um depurador a este código. Mikołajczak (2011) exemplifica a criação de um depurador simples, porém, era nítida a dificuldade do desenvolvimento deste depurador de forma com que o código execute sincronizado com a frequência desejada, pontos de parada, entre outros problemas encontrados durante o protótipo. Após alguma pesquisa de alternativas, viu-se que o tipo `delegate` para a geração das instruções, de fácil implementação e com um alto desempenho na execução. Após a adição destes procedimentos, o programa está pronto para rodar.

Ao iniciar, o simulador adiciona a quantidade de bytes que aquela instrução possui no Program Counter (PC) e a executa (alguns procedimentos podem modificar novamente o valor do PC, como a instrução `JMP`). Na interface gráfica é possível visualizar os valores do PC, dos registradores, das entradas, saídas, memória RAM, memória de pilha, memória ROM, a frequência desejada para a simulação e as *flags carry* e *zero*. Ao selecionar uma opção na lista de instruções é mostrada uma descrição mais detalhada do que cada uma faz, também é possível ver os *labels* correspondentes àquele endereço ao clicar nos marcadores em cinza.

A montagem do programa (para a interpretação do circuito interno, ou para exportação) é gerada a partir de uma tradução do código, pois antes disso, já são feitas as análises léxicas, sintáticas, semânticas e os métodos das instruções. Junto com estes métodos, também são gerados alguns metadados, na qual têm as informações de endereçamento, fluxo de controle, registrador, operação, entre outros. A partir disso, é gerado o binário correspondente àqueles metadados, conforme é especificado por Jung (2014). Este binário pode ser em formato puro (as instruções são transformadas em bytes), em texto hexadecimal ou texto no formato do trabalho anterior feito no Logisim. A Figura 11 mostra o fluxo da montagem do programa até a geração do código binário.

Figura 11 – Fluxo da montagem do programa



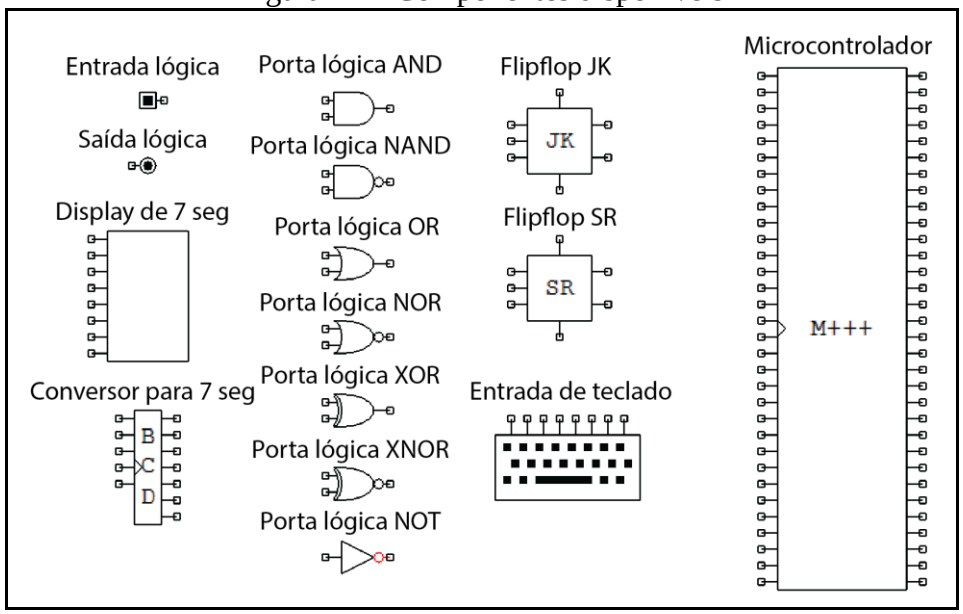

Fonte: elaborado pelo autor.

Na importação do código (também conhecida como desmontador ou *disassembler*), o programa lê os dados e converte para os metadados correspondentes, e a partir destes metadados o programa gera as instruções na tela de programação. Nota-se que ao importar e exportar o código são perdidas algumas informações, como: tabulação, espaços, linhas em branco, comentários e os *labels* (visto que no binário da M+++ não existem estas informações). Estes *labels* que estão faltando para endereços de salto são substituídos por "Byte_X", na qual X é o endereço correspondente àquele *label*. Vale também observar que, como a importação não passa pelas etapas de análise léxica, sintática e semântica, a importação pode falhar e não mostrar onde está este erro.

### 3.3.3 Ambiente de circuito

O ambiente de circuito é separado em três partes: a renderização da tela, a interpretação da eletrônica digital, e o circuito interno. Para a primeira delas, foi utilizado o *framework* OpenTK, um *wrapper* em C# para o OpenGL (APOSTOLOPOULOS, 2016). A partir deste *framework* foram desenhados os componentes eletrônicos, vários destes (como as portas lógicas) são exemplificados por Tocci e Widmer (2003), outros são baseados no software Logisim, desenvolvido por Burch (2005). Os componentes disponíveis podem ser observados na Figura 12.

Figura 12 – Componentes disponíveis




Fonte: elaborado pelo autor.

No desenvolvimento da interpretação eletrônica digital é feita uma verificação de todos os componentes que estão inseridos na tela e os instancia em um módulo que foi chamado de `CircuitSimulator`. A ideia inicial era utilizar o *framework* SharpCircuit criado por Godard (2014), o qual é um porte para C# do simulador de circuitos de Falstad (2016). Este *framework* já traz uma série de componentes eletrônicos, como resistores, capacitores indutores, transistores, e até componentes um pouco mais complexos, como o 555 *timer chip*, amplificador operacional, e *flip flops* (FALSTAD, 2016). Porém, após vários testes e pesquisas foi decidido abandoná-lo pela sua instabilidade e desenvolver um simulador de circuitos próprio para esta ferramenta.

O módulo `CircuitSimulator` foi parcialmente baseado no *framework* de Falstad (2016), e possui uma base de simulação que contém uma fila de execução com todos os componentes que serão interpretados a seguir. O componente `Entrada lógica`, por exemplo, sempre será adicionado automaticamente nas primeiras posições desta fila, já outros, como uma porta lógica, são adicionados somente quando o simulador os identifica pela propagação daqueles executados anteriormente. O Quadro 8 esquematiza o método `execute` para o componente `Flipflop JK`, na qual ele muda o estado dos pinos de saída dependendo dos estados dos pinos de entrada.

Quadro 8 - Método `execute` do `Flipflop jk`

```
1  base.Execute();
2  if (S.GetDigital() == Pin.HIGH) { //S = 1
3      Q.Value = Pin.HIGH;
4      Qnot.Value = Pin.LOW;
5  } else if (R.GetDigital() == Pin.HIGH) { // R = 1
6      Q.Value = Pin.LOW;
7      Qnot.Value = Pin.HIGH;
8  } else if(CLK.Value == Pin.LOW && lastClk == Pin.HIGH) { //Clock desc
9          if (J.GetDigital() == Pin.HIGH) {
10         if(K.GetDigital() == Pin.HIGH) { // J = 1, K = 1
11             Q.Value = Q.Neg();
12             Qnot.Value = Qnot.Neg();
13         } else { // J = 1, K = 0
14             Q.Value = Pin.HIGH;
15             Qnot.Value = Pin.LOW;
16         }
17     } else {
18         if(K.GetDigital() == Pin.HIGH) { // J = 0, K = 1
19             Q.Value = Pin.LOW;
20             Qnot.Value = Pin.HIGH;
21         } else { // J = 0, K = 0
22             //Não faz nada, pois a saída mantém a mesma
23         }
24     }
25 }
26 lastClk = CLK.Value;
27 Q.Propagate();
28 Qnot.Propagate();
```

Fonte: elaborado pelo autor.

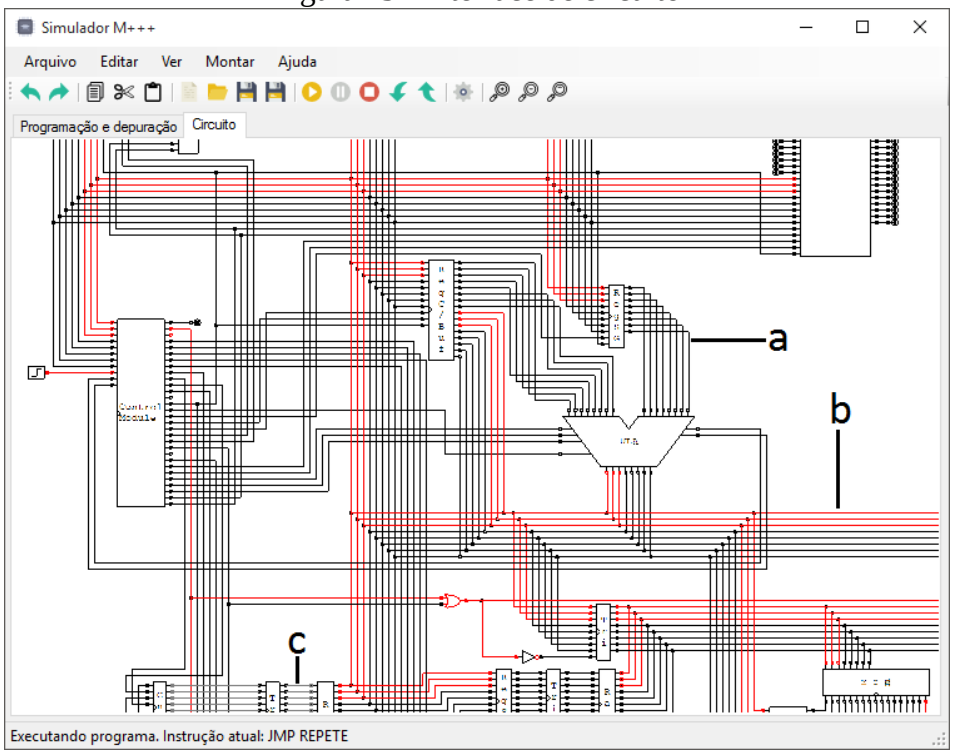
As linhas 27 e 28 do Quadro 8 chamam os métodos `Propagate` nos pinos `Q` e `Qnot`. Estes métodos são necessários para propagar o sinal de saída para os próximos componentes. Ele verifica quais estão conectados a ele e caso passe em alguns testes de validação, o próximo componente conectado a estes pinos de saída são adicionados na fila de execução do simulador. Com base nisso, é possível montar um circuito eletrônico mais complexo, fazendo associações de portas lógicas, display de 7 segmentos com decodificadores, e utilizar das portas de saída do microcontrolador M+++ para acionar outros.

Para a programação do módulo de controle foi necessário copiar a memória de controle da máquina M+++ desenvolvida por Jung (2014) para uma matriz e montar uma planilha (para análise e testes) com estes sinais contendo as instruções atreladas a cada endereço destes sinais interno. Além disso, também foi necessário modificar estes sinais, pois a lógica de funcionamento do Logisim (software utilizado na M+++) e do interpretador de circuitos deste trabalho diferem.

A criação de algum componente é dada por um clique com o botão direito do mouse na tela do circuito e selecionado no menu o item desejado a ser criado, a movimentação deles são feitos pela seleção dos mesmos (clicando neles) e arrastando-os para onde é desejado. Já para a movimentação da tela basta clicar e arrastar com o botão direito do mouse. A Figura 13

destaca uma parte do circuito interno do microcontrolador, assim como os sinais internos sendo executados em uma instrução `JMP`.

Figura 13 - Interface de circuito




Fonte: elaborado pelo autor.

Existem 3 cores de sinais na simulação:

a) preto: 0V ou sinal baixo;

b) vermelho: 5V ou sinal alto;

c) cinza: terceiro estado.

Foi necessário adicionar um número identificador de iteração em todos os componentes para não ocorrerem loops na simulação e travar o programa. Este identificador é incrementado a cada iteração (aproximadamente 62 vezes por segundo) e quando o componente é executado é passado este identificador para o mesmo. Caso ele já tenha este identificador, não será executado. Este mesmo mecanismo foi aplicado nos pinos dos componentes.

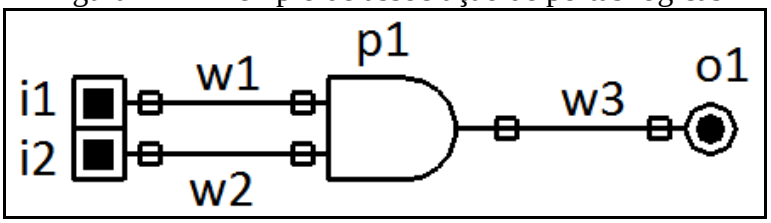
O terceiro estado funciona também como forma de detectar erros no circuito, pois os fios e terminais que não estiverem com o mesmo identificador da simulação ficam em terceiro estado. Por exemplo: Se for adicionado um componente `Flipflop JK` no circuito mas deixar as entradas em aberto, não será executado fazendo com que as suas saídas mantenham as mesmas sem alterar o identificador de iteração. Segundo Tocci (2003, p. 398),

> Esse tipo de saída aproveita a vantagem da operação em alta velocidade das configurações de saída pull-up/pull-down ao mesmo tempo, permitindo que as

> saídas sejam conectadas juntas para compartilharem um fio comum. Ela é denominada tristate porque permite três estados na saída: ALTO, BAIXO e alta impedância (Hi-Z). (TOCCI, 2003, p. 398)

A Figura 14, mostra um exemplo de montagem de uma associação de portas lógicas, na qual, i1 e i2 são as entradas lógicas, w1, w2 e w3 são fios, p1 é a porta lógica do tipo E e o1 é a saída lógica. Vale destacar também os terminais indicados por um pequeno quadrado próximo ao desenho dos componentes. Já o Quadro 9 mostra o passo a passo da execução do simulador com o escopo do componente e a descrição do passo referente ao circuito utilizado na Figura 14.

Figura 14 – Exemplo de associação de portas lógicas




Fonte: elaborado pelo autor.

Quadro 9 - Passo a passo da execução

| Passo | Escopo | Descrição |
|---|---|---|
| 1 | Simulador | É incrementado o identificador de iteração (anteriormente 0) |
| 2 | Simulador | São adicionados na fila de execução todos os componentes que podem iniciar a simulação. São eles: i1 e i2 |
| 3 | i1 | O componente é executado: É passado o valor atual dele (0) para o pino de saída e executado a operação `Propagate` no pino. |
| 4 | i1:terminal | O terminal identifica que o terminal 1 do componente w1 está conectado a ele, então este é adicionado na fila de execução. |
| 5 | i2 | O componente é executado: É passado o valor atual dele (0) para o pino de saída e executado a operação `Propagate` no pino. |
| 6 | i2:terminal | O terminal identifica que o terminal 1 do componente w2 está conectado a ele, então este é adicionado na fila de execução. |
| 7 | w1 | O componente é executado: verifica-se que o terminal 1 deste componente possui o identificador atualizado, com isso, ele passa o valor do terminal 1 para o terminal 2 e executa a operação `Propagate` no terminal 2. |
| 8 | w1:terminal 2 | O terminal identifica que o terminal 1 do componente p1 está conectado a ele, porém, o terminal 2 não está com o identificador atualizado. Então ele não é adicionado à fila de execução. |
| 9 | w2 | O componente é executado: verifica-se que o terminal 1 deste componente possui o identificador atualizado, com isso, ele passa o valor do terminal 1 para o terminal 2 e executa a operação `Propagate` no terminal 2. |
| 10 | w2:terminal 2 | O terminal identifica que o terminal 2 do componente p1 está conectado a ele, e, neste passo ambos os terminais de entrada estão com os identificadores atualizados, e com isso, é adicionado o componente p1 na fila de execução. |
| 11 | p1 | O componente é executado: é feito a operação “E” entre os terminais 1 e 2 do componente e o resultado é passado para o terminal 3 (saída) e executa a operação `Propagate` no terminal 3. |
| 12 | p1:terminal 3 | O terminal identifica que o terminal 1 do componente w3 está conectado a ele, então este é adicionado na fila de execução. |
| 13 | w3 | O componente é executado: verifica-se que o terminal 1 deste componente possui o identificador atualizado, com isso, ele passa o valor do terminal 1 para o terminal 2 e executa a operação `Propagate` no terminal 2. |
| 14 | w3:terminal 2 | O terminal identifica que o terminal 1 do componente o1 está conectado a ele, então o adiciona na fila de execução. |
| 15 | o1 | O componente é executado: lê o valor do terminal de entrada e o guarda em uma variável interna. Posteriormente uma tarefa sendo executada em segundo plano lê este valor interno e o deixa com a cor vermelha ou preta. |
| 16 | Simulador | Não há mais componentes na fila de execução. Fim da execução. |

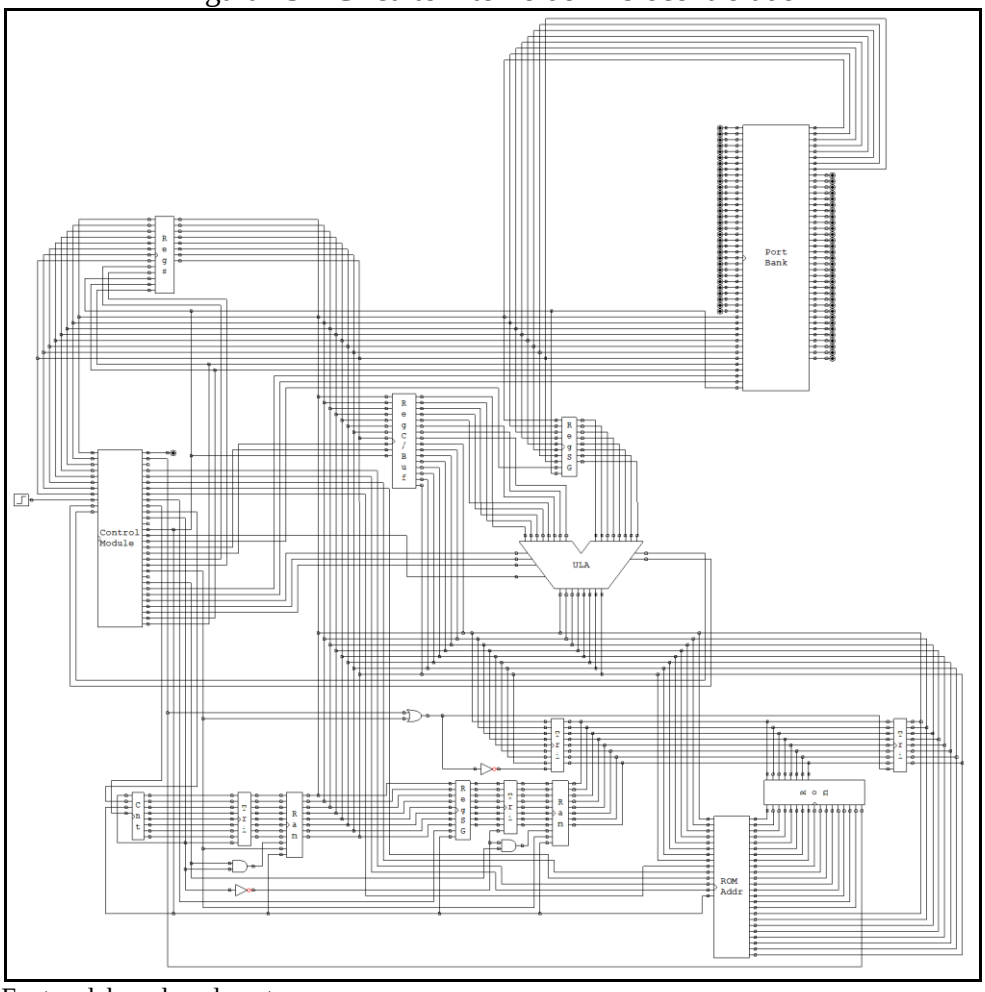
Fonte: elaborado pelo autor.

Foi detectado um problema na versão anterior da M+++: ao executar uma operação `PUSH <reg>` o microcontrolador escreverá o mesmo valor do registrador no primeiro endereço da memória RAM. Este erro ocorria (e foi corrigido neste trabalho) pois esta instrução jogará o valor no barramento de dados principal ao mesmo tempo que dava um pulso de *clock* para a escrita na memória RAM (o mesmo pulso de *clock* é usado na memória de pilha). Ele foi

corrigido com a adição do identificador de iteração: ao executar a operação `PUSH <reg>` os dados de endereçamento na memória RAM estão com o identificador de iteração desatualizado, impedindo a execução da memória RAM no pulso de *clock* para a escrita na memória RAM.

Inicialmente a ideia principal do circuito interno do microcontrolador era que todos os seus componentes, inclusive os fios, fossem interpretados de acordo com a instrução atual que estava simulando, como se fosse um grande `switch`. Cada uma teria uma quantidade de ciclos de *clock* para a execução da mesma, e além do índice da instrução atual também teria um índice do ciclo de *clock* atual. Mas a construção desta ideia iria demorar muito tempo e achou-se mais fácil fazer a utilização deste simulador eletrônico internamente também. Para isso foi necessário criar e implementar todos os componentes internos, como o módulo de controle, o endereçador de memória, memória RAM, registrador, banco de registradores etc. Observa-se na Figura 15 o circuito interno do microcontrolador desenvolvido neste trabalho.

Figura 15 – Circuito interno do microcontrolador




Fonte: elaborado pelo autor.

### 3.3.4 Problemas e melhorias

Durante o desenvolvimento deste trabalho foram identificados vários problemas e melhorias sugeridos pelo orientador e pelos alunos. Os bugs mais complicados e mais graves são descritos na sequência, porém, vários outros menores também ocorreram durante o desenvolvimento da ferramenta, entre elas estão: descrição incorreta de algumas instruções, ao montar o programa gerava-se outra instrução, ao executar a função zoom era também executada em outras telas que não estava abertas etc. Ao todo foram feitos mais de 70 *commits* no repositório da ferramenta hospedado no Github e várias versões betas. A primeira versão beta foi entregue para o orientador no dia 01/05/2017, dando o nome de “Beta 1”.

Na segunda versão do simulador, foi identificado que em alguns computadores ocorriam problemas na renderização das texturas do ambiente de circuito. Estas texturas são usadas para mostrar os textos presentes neste ambiente e ao invés de renderizar o texto, mostrava uma textura totalmente branca. Este problema ocorria em computadores que possuem uma placa de vídeo muito antiga, com os drivers desatualizados ou sem os mesmos corretamente instalados, tornando-os incompatíveis com as texturas Non Power Of Two (NPOT), na qual, exige que os tamanhos estejam em potência de dois (1, 2, 4, 8, 16, ...) (OPENGL WIKI CONTRIBUTORS, 2015).

Foi necessário retirar a instrução de carregamento da tela e inserção dinâmica dos componentes em uma `thread` separada ao abrir as telas de memória RAM e memória de pilha. Esta alocação travava o software em algumas ocasiões e não foi possível descobrir o motivo disto. Agora, ao abrir uma destas telas o software apresenta uma pequena pausa, pois ele está instanciando os elementos na `thread` principal.

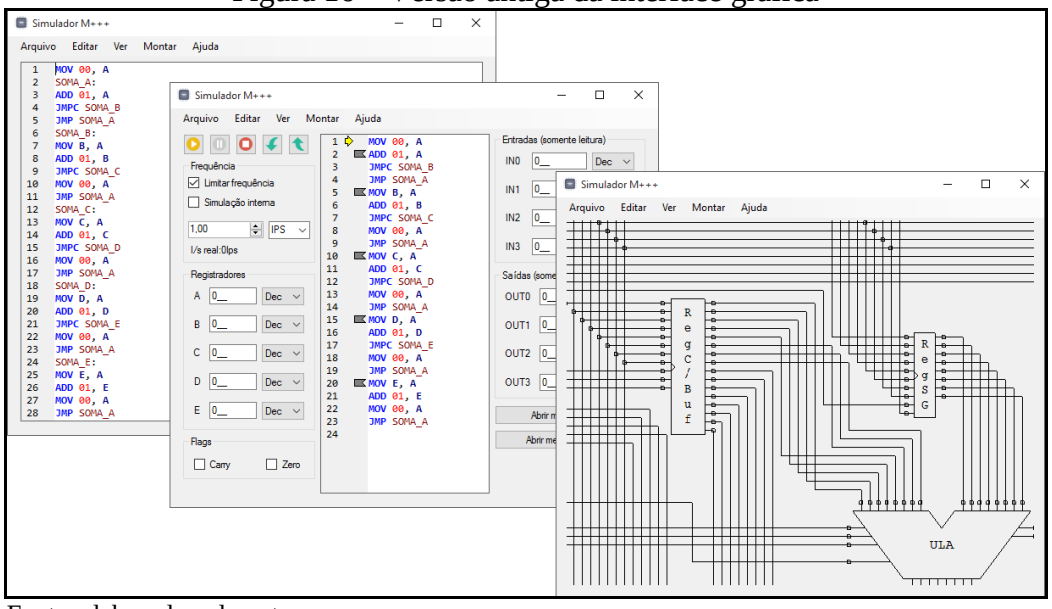
Na versão três foram arrumados alguns bugs que estavam ocorrendo na limitação da frequência de simulação e foi invertida a ordem dos pinos do circuito decodificador. Além disso, foi feito uma pequena melhoria para aprimorar o desempenho na renderização dos componentes eletrônicos.

Para a quarta versão do software foi adicionado o PC na tela inicial do simulador, melhorando as informações de compilação, foi trocado o número das linhas do programa gerado para o endereço atual da instrução, colocado a identificação de cor nos próprios terminais dos componentes, aplicado o conteúdo interno da memória ROM, os ciclos de *clock* por passo na simulação interna e a instrução que está sendo executada no momento.

Durante o desenvolvimento também foi proposto uma interface gráfica dividida em três telas: Ambiente de Programação, Ambiente de Depuração e Ambiente de circuito. Estas

poderiam ser acessadas pelo menu `Ver > Código | Depuração | Circuito`, porém, era pouco intuitivo acessá-las por este menu. A versão antiga da interface gráfica é ilustrada pela Figura 16. Na quarta versão do simulador foi colocado um modelo de abas que unifica o ambiente programação e depuração em apenas uma aba, e outra com o ambiente de circuitos. Além disso, este novo modelo de interface gráfica também adicionou um painel superior com os principais controles, como `Rodar`, `Pausar`, `Parar`, `Salvar` etc. A quinta versão somente conserta um problema que não mostrava as instruções enquanto executava o passo de ciclo de *clock*.

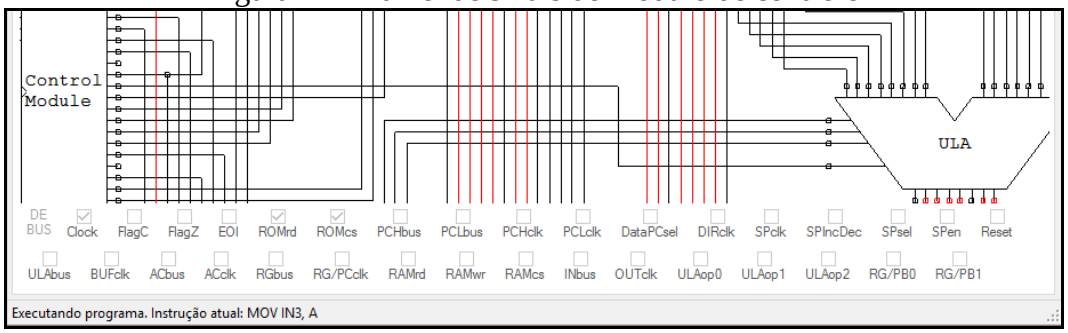
Figura 16 – Versão antiga da interface gráfica




Fonte: elaborado pelo autor.

Foi sugerido, para a versão seis, que o ambiente de circuito mostre os sinais que saem do módulo de controle com mais clareza, pois não era possível identificar rapidamente, por exemplo, o que estava no *bus* de dados. Para isto, foi adicionado um pequeno campo abaixo da tela do circuito em que mostra os valores que estão ativos e inativos no módulo de controle. Além disso, foi alterado o endereçamento da memória RAM e da pilha para hexadecimal e mais alguns pequenos problemas encontrados. A Figura 17 exibe o painel dos sinais que saem do módulo de controle.

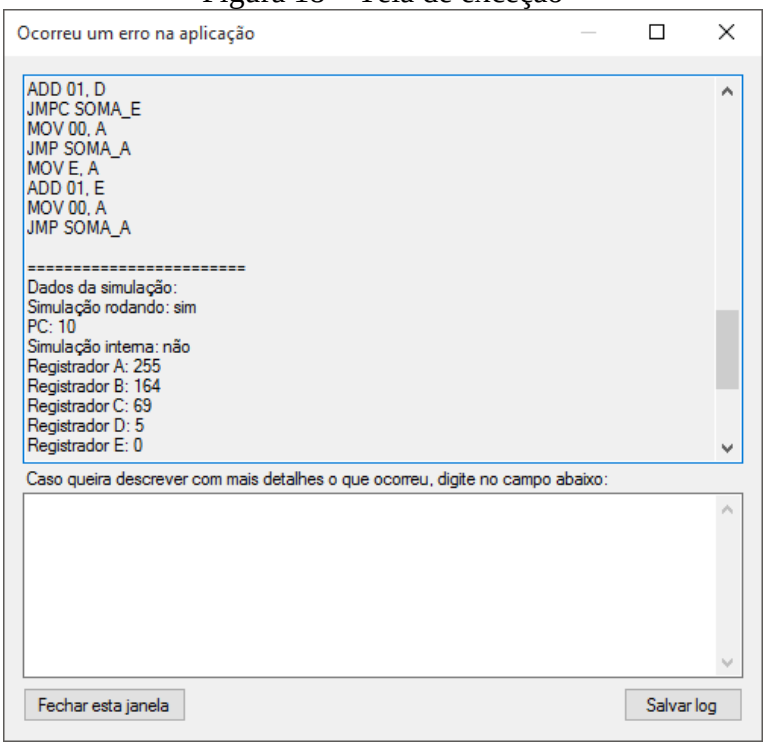
Figura 17 – Painel de sinais do módulo de controle




Fonte: elaborado pelo autor.

Para facilitar e também por sugestão para a versão seis, foi adicionada uma tela para reportar bugs no software. Ela coleta alguns dados referente à simulação atual, os códigos digitados e gerados, e outras informações relevantes. Ela é aberta automaticamente caso ocorra alguma exceção não tratada com as informações do erro e a sua pilha. É possível salvar estas informações em um arquivo para posteriormente tratar este problema. A Figura 18 mostra a tela de exceção junto com algumas informações relevantes para a reprodução do erro.

Figura 18 – Tela de exceção



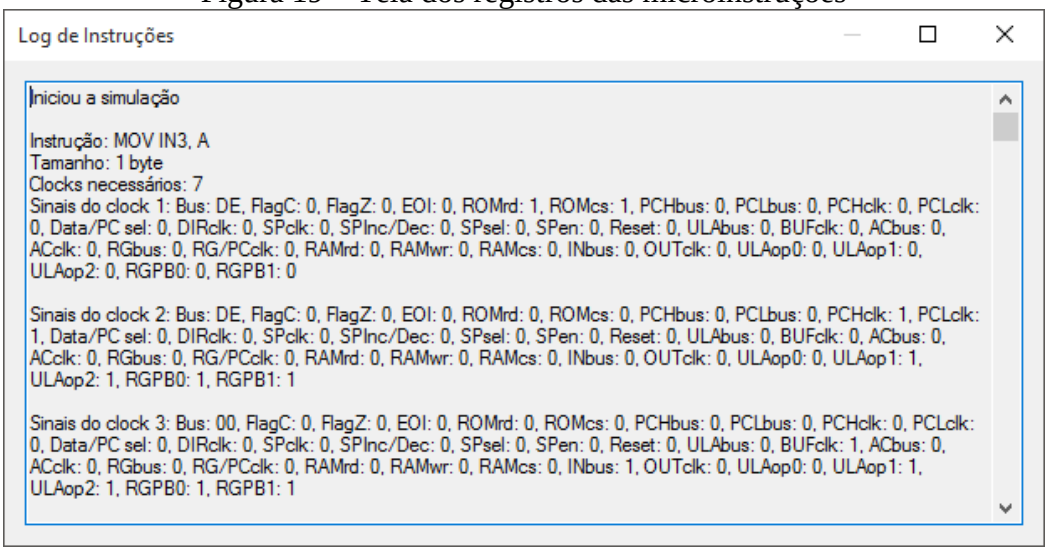

Fonte: elaborado pelo autor.

As versões sete e oito consertava um problema que gerava uma exceção geral ao tentar compilar com os *labels* inválidos. Inicialmente era proposto utilizar somente o caractere ; (ponto e vírgula) para comentários, porém, para ficar totalmente retro compatível foi

adicionado também a `string` `//` (duas barras) como tipo de comentário para a versão nove do software.

Outra sugestão, adicionada na versão 10, foi a inserção de uma tela dos registros das microinstruções executadas. Isso facilita na análise do programa feito, e ajuda na compreensão do microcontrolador pois esta registra todos os procedimentos simulados, o tamanho, a quantidade ciclos necessários, e os sinais que foram gerados em cada sinal de *clock*. A Figura 19 ilustra esta tela com algumas informações geradas pela operação `MOV IN3, A`.

Figura 19 – Tela dos registros das microinstruções




Fonte: elaborado pelo autor.

A versão 11 do simulador corrige dois bugs: sob algumas circunstâncias não era limpa a tela de código ou do circuito ao criar um novo projeto, antes também era possível instanciar várias janelas simultaneamente da memória RAM, ROM e de pilha.

Até o momento, ao clicar no botão `Abrir memória RAM` ou `Abrir memória de pilha`, o simulador faz uma pequena pausa. Isto ocorre porque estão sendo criados seus componentes destas telas. Foi sugerido criar uma tela inicial com o carregamento do programa e sua versão. Esta também foi aproveitada ao abrir a memória RAM e memória *Stack*, porém, o simulador também apresenta uma pausa. A Figura 20 ilustra esta tela de carregamento assim como a sua versão em beta 12 no canto inferior direito.

Figura 20 – Tela de carregamento


Fonte: elaborado pelo autor.

A 13ª versão do simulador adiciona uma mensagem de erro quando o programa tenta retirar algo da memória de pilha, ocasionando um *stack underflow*, e quando coloca muitos valores nela, mostrando um *stack overflow*.

A versão 14 do software corrige alguns bugs relacionados a contagem de instruções na tela `Log de Instruções` e faz com que os registradores A, B, C, D e E no painel de depuração aparecem por padrão em hexadecimal.

## 3.4 ANÁLISE DOS RESULTADOS

Analisando as ferramentas correlatas mostradas nas seções 2.2.1, 2.2.2, 2.2.3 e 2.2.4 é notável perceber que este trabalho teve praticamente as mesmas características apresentadas em comum, mas a grande vantagem é a visualização interna do microcontrolador para estudar melhor a arquitetura, e posicionar os componentes livremente, semelhante ao GPSIM (FRANKLIN, RANKIN E DATTALO, 2015). Também vale destacar o alto desempenho alcançado na emulação do microcontrolador. A ferramenta GPSIM simula um programa simples a velocidade de 25MHz em um computador equipado com um PII 400MHz (FRANKLIN, RANKIN E DATTALO, 2015). Este software, entretanto, alcançou a 120MIPS em uma máquina equipada com um i7 4790K @ 4.0GHz. O Quadro 10 relaciona as principais características entre os trabalhos estudados e a atual desenvolvida.

Quadro 10 - Diferenças ferramentas estudadas

| | Rogers (2016) | Oshonsoft (2016) | Franklin, Rankin e Dattalo (2015) | Scott e Filbinger (2016) | Este projeto |
|---|---|---|---|---|---|
| Velocidade da simulação | Não deixa claro | Não deixa claro | 25MHz @ PII 400MHz | 10Hz | 120MIPS @ i7 4790k 4.0GHz |
| Permite a depuração e inserção de *breakpoints* | Sim | Sim | Sim | Não | Sim |
| Permite programar o microcontrolador diretamente na ferramenta | Sim | Sim | Não | Sim | Sim |
| Possibilita a montagem de circuitos eletrônicos | Não | Não | Sim | Não | Sim |
| Quantidade de componentes eletrônicos | 9 | 24 | 11 | 0 | 14[1] |

Fonte: elaborado pelo autor.

O simulador também foi utilizado pelos alunos da matéria de Arquitetura I, lecionado pelo prof. Miguel Alexandre Wisintainer, no primeiro semestre de 2017, na Universidade Regional de Blumenau. Realizou-se uma pesquisa para observar como foi a experiência do simulador desenvolvido neste trabalho. A mesma pode ser observada no Apêndice B.

Conforme os resultados apresentados no Apêndice C, observa-se que o simulador teve resposta bastante positiva com relação ao aprendizado que o software proporcionou. Segundo vários estudantes, ainda ocorriam alguns bugs que fechava o software, estes ainda não puderam ser corrigidos devido à falta de detalhamento de como ocorriam, impossibilitando a reprodução dos mesmos. Ainda assim, os acadêmicos gostaram da solução e a maioria apreciaria compreender mais a arquitetura de computadores.

[1] Desconsiderando os componentes internos do microcontrolador.

# 4 CONCLUSÕES

Como observado na seção 3.4, este simulador é uma solução viável para auxiliar no desenvolvimento e qualificação dos alunos das matérias de sistemas embarcados, computação digital, arquitetura de computadores, entre outras afins.

Em relação ao trabalho anterior, esta nova versão possui grandes vantagens de velocidade, e análise do programa, visto que é possível adicionar pontos de parada e em apenas uma interface tem-se todos os valores de registradores importantes do microcontrolador. Entretanto, possui uma grande desvantagem para a edição do circuito, pois na versão anterior tinha-se liberdade total para edita-lo, podendo visualizar e alterar toda a parte interna do microcontrolador e uma gama maior de componentes eletrônicos.

## 4.1 EXTENSÕES

Como sugestões de extensões para este trabalho, é proposto:

a) aumentar a quantidade de componentes eletrônicos;
b) permitir a criação de sub-circuitos ou macros, parecido com software Logisim;
c) ampliar a quantidade de instruções disponíveis no microcontrolador;
d) corrigir alguns problemas não solucionados que fecham o software e não puderam ser reproduzidos neste trabalho;
e) tornar o log mais amigável visualmente;
f) tornar o circuito mais amigável visualmente.

# REFERÊNCIAS


AHO, Alfred V. et al. **Compiladores:** princípios, técnicas e ferramentas. 2. ed. São Paulo: Pearson, 2010. 634 p. Tradução de: Daniel Vieira.

APOSTOLOPOULOS, Stefanos. **Open Tool Kit.** 2016. Disponível em: <https://github.com/opentk/opentk>. Acesso em: 07 nov. 2016.

BORGES, Jonathan Manoel. **M++:** Simulador de UCP. 2003. Trabalho final da disciplina de Arquitetura de Computadores, sob orientações dos professores Miguel Alexandre Wisintainer e Antônio Carlos Tavares. Disponível em: <http://www.inf.furb.br/~maw/mmaismais/>. Acesso em: 22 ago. 2016.

BURCH, Carl. **Logisim.** 2005. Disponível em: <http://www.cburch.com/logisim/index.html>. Acesso em: 10 maio 2017.

FALSTAD, Paul. **Circuit Simulator.** 2016. Disponível em: <http://www.falstad.com/circuit/>. Acesso em: 07 nov. 2016.

FAYZULLIN, Marat. **How To Write a Computer Emulator.** 2000. Disponível em: <https://fms.komkon.org/EMUL8/HOWTO.html>. Acesso em: 22 maio 2017.

FRANKLIN, Craig; RANKIN, Roy; DATTALO, Scott. **GPSim.** 2015. Disponível em: <http://gpsim.sourceforge.net/>. Acesso em: 02 set. 2016.

GODARD, Riley Mervill. **SharpCircuit.** 2014. Disponível em: <https://github.com/Mervill/SharpCircuit>. Acesso em: 07 nov. 2016.

HODGSON, Neil. **Scintilla.** 2003. Disponível em: <http://www.scintilla.org/>. Acesso em: 07 nov. 2016.

JUNG, Jean. **M+++.** 2014. Disponível em: <https://github.com/jejung/maquina-plus-plus/blob/master/README.md>. Acesso em: 10 maio 2017.

LAUREANO, Marcos. **Máquinas Virtuais e Emuladores:** Conceitos, Técnicas e Aplicações. [s. L.]: Novatec, 2006. 184 p.

LENZ, André Luis. **Linguagem para Programar Microcontroladores:** Assembly, C ou Basic?. [2012?]. Disponível em: <http://www.ebah.com.br/content/ABAAAAdWoAC/linguagem-programar-microcontroladores-assembly-c-basic>. Acesso em: 07 set. 2016.

MACORATTI, José Carlos. **C# - Delegates e Eventos:** Conceitos básicos. 2011. Disponível em: <http://www.macoratti.net/11/05/c_dlg1.htm>. Acesso em: 16 maio 2017.

MARANGON, Johni Douglas. **Compiladores para Humanos.** 2017. Disponível em: <https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/semantic-analysis.html>. Acesso em: 10 maio 2017.

MCCLURE, Tucker. **How Simulations Work.** 2014. Disponível em: <http://www.anuncommonlab.com/articles/how-simulations-work/>. Acesso em: 07 set. 2016.

MIKOŁAJCZAK, Marcin. **Writing an automatic debugger in 15 minutes.** 2011. Disponível em: <https://tripleemcoder.com/2011/12/10/writing-an-automatic-debugger-in-15-minutes-yes-a-debugger/>. Acesso em: 07 nov. 2016.

MOREIRA, L. B.; MAIA, R. F. **Ambiente multiplataforma para o processo de ensino-aprendizagem de programação de microcontroladores.** In: Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE). 1., 2010, João Pessoa. *Anais...* São Bernardo do Campo, 2010.

NOVA ELETRÔNICA. **Qual a diferença entre Microprocessador e Microcontrolador.** [2014?]. Disponível em: <http://blog.novaeletronica.com.br/qual-diferenca-entre-microprocessador-e-microcontrolador/>. Acesso em: 07 set. 2016.

OPENGL WIKI CONTRIBUTORS. **NPOT Texture.** 2015. Disponível em: <http://www.khronos.org/opengl/wiki_opengl/index.php?title=NPOT_Texture&oldid=12502>. Acesso em: 25 maio 2017.

OPENTK. **Open Toolkit Library.** 2017. Disponível em: <https://opentk.github.io/>. Acesso em: 10 maio 2017.

OSHONSOFT (Org.). **PIC Simulator IDE.** 2016. Disponível em: <http://www.oshonsoft.com/pic.html>. Acesso em: 02 set. 2016.

RICARTE, Ivan Luiz Marques. **Análise semântica.** 2003. Disponível em: <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node71.html>. Acesso em: 17 maio 2017.

ROGERS, James. **EDSIM51.** 2016. Disponível em: <http://www.edsim51.com/index.html>. Acesso em: 06 set. 2016.

ROGERS, James. **EDSIM51:** About the Simulator. 2013. Disponível em: <http://www.edsim51.com/introduction.html>. Acesso em: 06 set. 2016.

SCOTT, John Clark. **But How Do It Know?:** The Basic Principles of Computers for Everyone. Oldsmar: John Clark Scott, 2009. 222 p.

SCOTT, John Clark; FILBINGER, Siegbert. **But How Do It Know?** 2016. Disponível em: <http://www.buthowdoitknow.com/>. Acesso em: 22 ago. 2016.

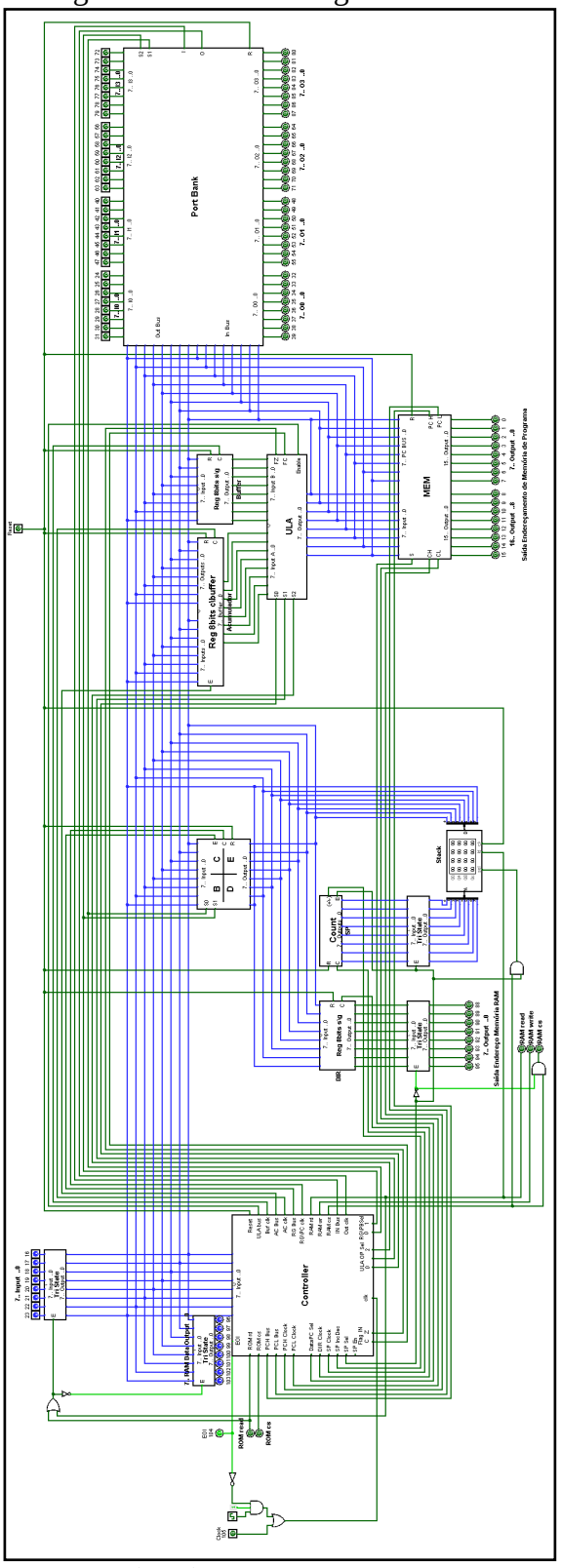
SLUSSER, Jacob. **ScintillaNET.** 2016. Disponível em: <https://github.com/jacobslusser/ScintillaNET>. Acesso em: 07 nov. 2016.

TOCCI, Ronald J.; WIDMER, Neal S.. **Sistemas Digitais:** Princípios e Aplicações. 8. ed. São Paulo: Pearson Prentice Hall, 2003. 755 p. Tradução de: José Lucimar do Nascimento.

## APÊNDICE A – Circuito interno da M+++

A Figura 21 mostra o circuito interno da M+++.

Figura 21 – Circuito lógico da M+++




Fonte: Jung (2014).

## APÊNDICE B – Relação de perguntas enviado aos alunos

A Figura 22, Figura 23 e Figura 24 mostra a relação das perguntas enviadas as turmas de Arquitetura I lecionado pelo professor Miguel Alexandre Wisintainer no primeiro semestre de 2017.

Figura 22 – Feedback do simulador parte 1

**Feedback simulador M+++**

Este simulador está nas fases finais de desenvolvimento na matéria de TCC2 pelo aluno Jean Carlos Klann e com orientação do Professor Miguel Alexandre Wisintainer.

*Obrigatório

1. **Digite seu nome:** *

2. **Você estuda em qual turno?** *
   *Marcar apenas uma oval.*
   - Matutino
   - Vespertino
   - Noturno

3. **O simulador contribuiu para os seus estudos?** *
   *Marcar apenas uma oval.*
   - Sim
   - Não

4. **Você teve vontade de aprender mais sobre a arquitetura de computadores?** *
   *Marcar apenas uma oval.*
   - Sim
   - Não

5. **Com que frequência você utilizou o simulador nas últimas aulas?** *
   *Marcar apenas uma oval.*

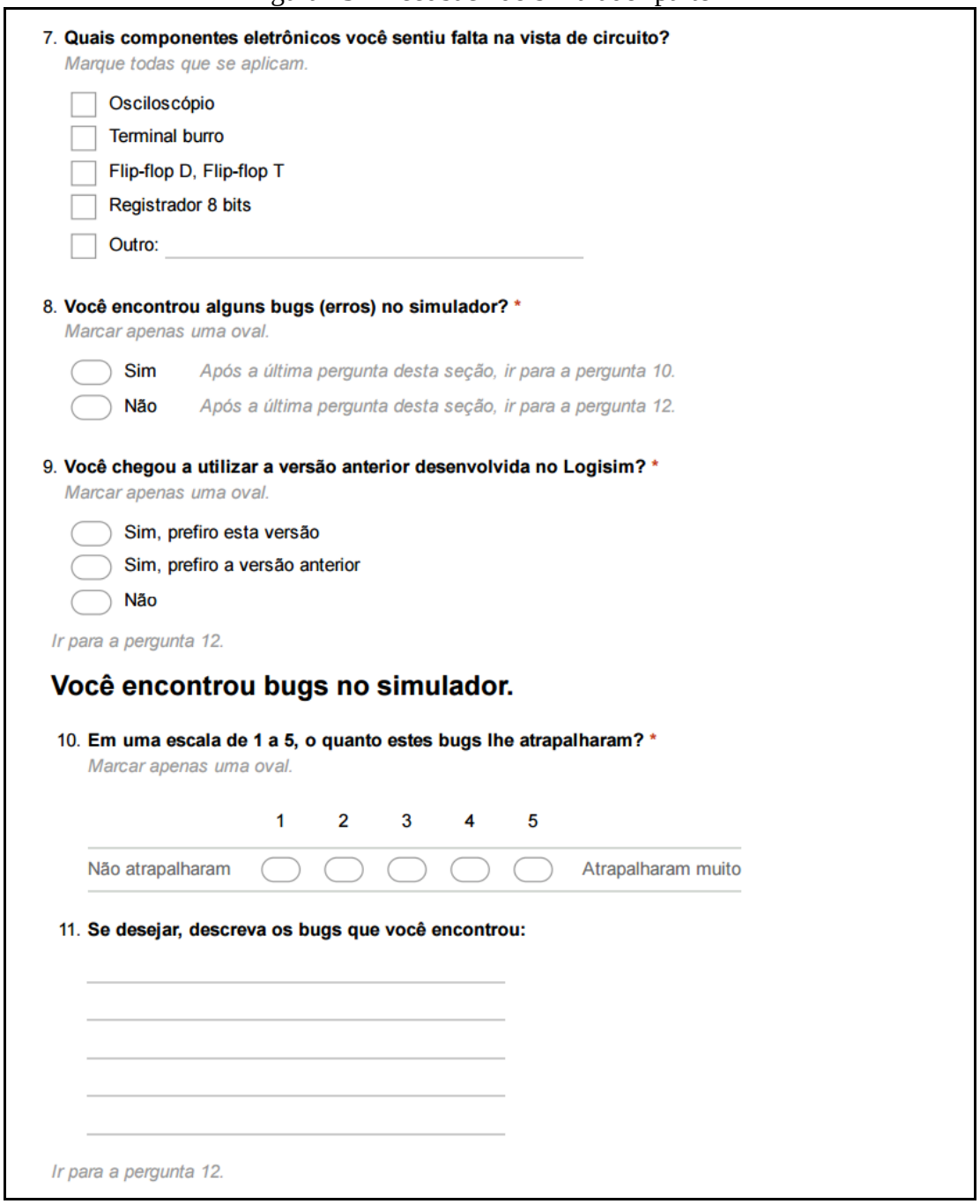
| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Nunca foi utilizado | | | | | | Todas as últimas aulas |

6. **Em uma escala de 1 a 5, o quão fácil foi operar no simulador?** *
   *Marcar apenas uma oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Foi difícil, tive muita dificuldade | | | | | | Foi fácil, não tive dificuldades |

Fonte: elaborado pelo autor.

Figura 23 – Feedback do simulador parte 2



7. **Quais componentes eletrônicos você sentiu falta na vista de circuito?**
*Marque todas que se aplicam.*

- [ ] Osciloscópio
- [ ] Terminal burro
- [ ] Flip-flop D, Flip-flop T
- [ ] Registrador 8 bits
- [ ] Outro: _____

8. **Você encontrou alguns bugs (erros) no simulador?** *
*Marcar apenas uma oval.*

- ◯ Sim — *Após a última pergunta desta seção, ir para a pergunta 10.*
- ◯ Não — *Após a última pergunta desta seção, ir para a pergunta 12.*

9. **Você chegou a utilizar a versão anterior desenvolvida no Logisim?** *
*Marcar apenas uma oval.*

- ◯ Sim, prefiro esta versão
- ◯ Sim, prefiro a versão anterior
- ◯ Não

*Ir para a pergunta 12.*

## Você encontrou bugs no simulador.

10. **Em uma escala de 1 a 5, o quanto estes bugs lhe atrapalharam?** *
*Marcar apenas uma oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Não atrapalharam | ◯ | ◯ | ◯ | ◯ | ◯ | Atrapalharam muito |

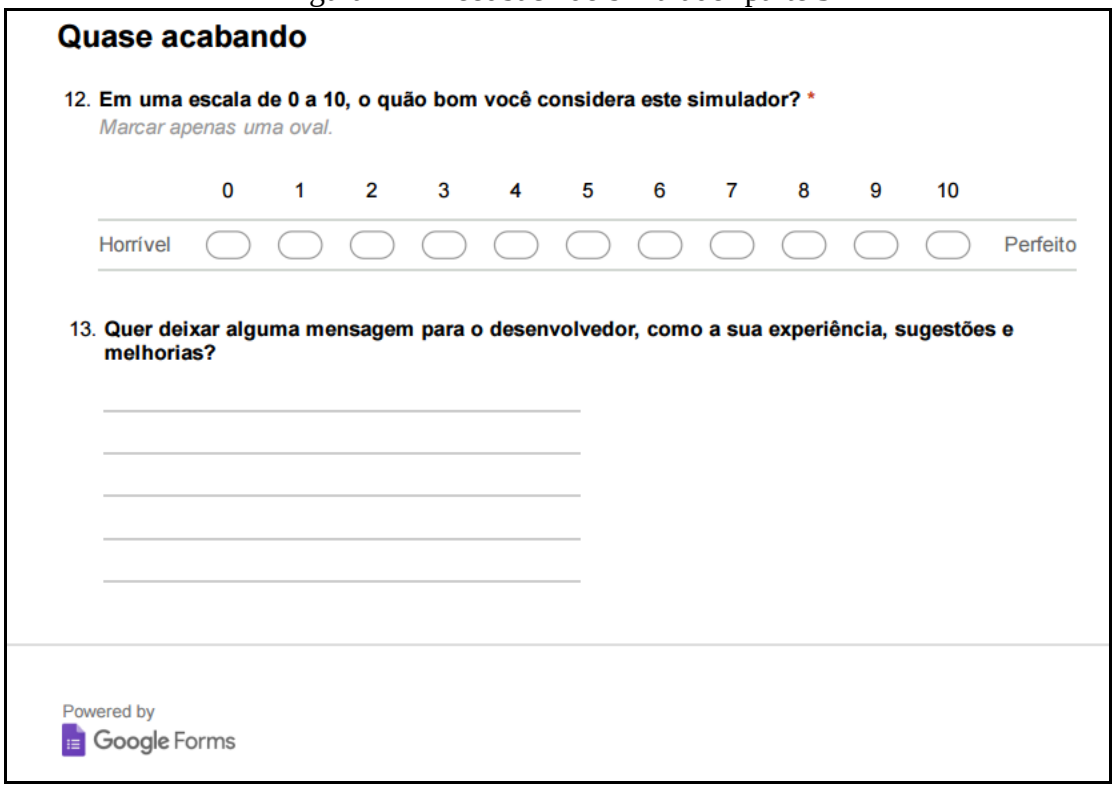
11. **Se desejar, descreva os bugs que você encontrou:**

_____
_____
_____
_____
_____

*Ir para a pergunta 12.*

Fonte: elaborado pelo autor.

Figura 24 – Feedback do simulador parte 3

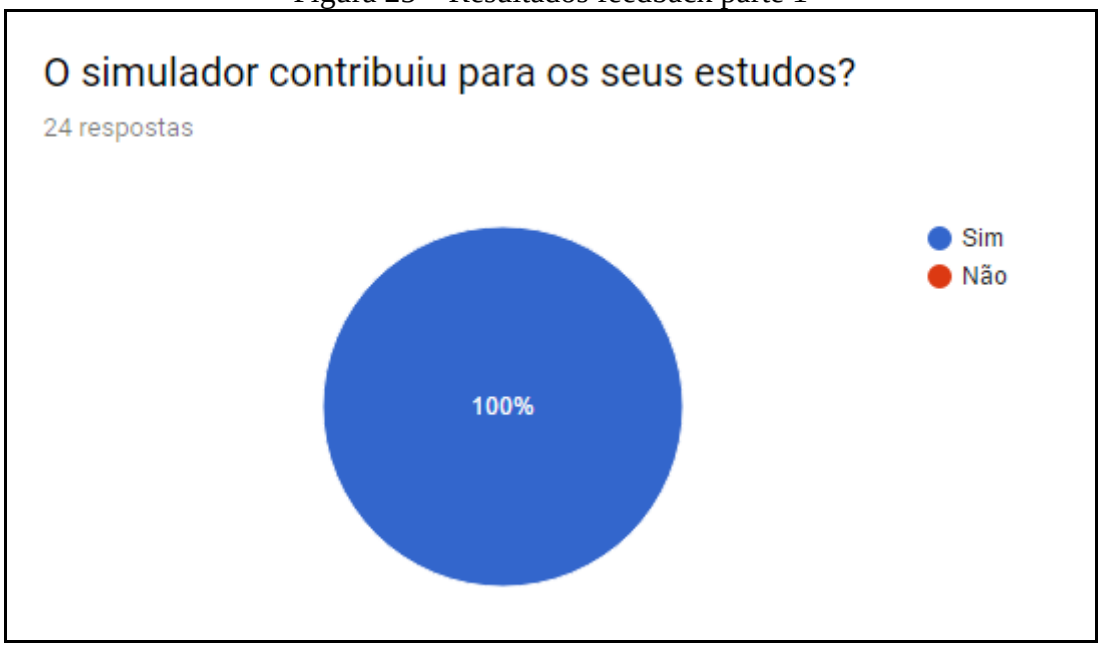



Fonte: elaborado pelo autor.

## APÊNDICE C – Resultado do questionário

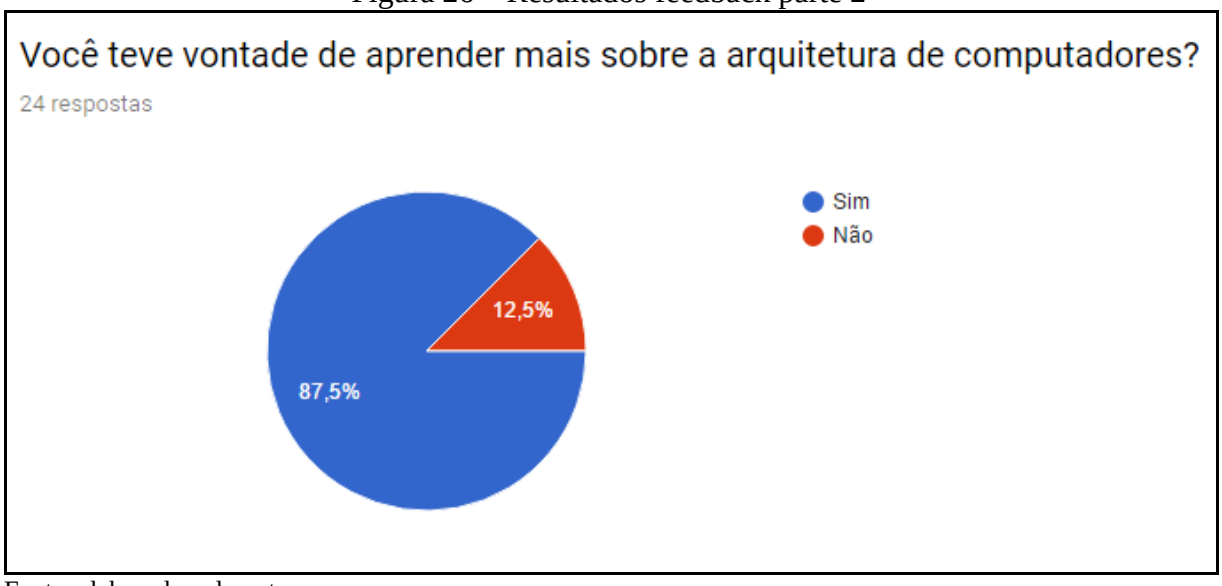
As figuras de 23 a 31 mostram a relação dos resultados das perguntas que estão no Apêndice A enviadas para as turmas de Arquitetura I lecionado pelo professor Miguel Alexandre Wisintainer no primeiro semestre de 2017.

Figura 25 – Resultados feedback parte 1



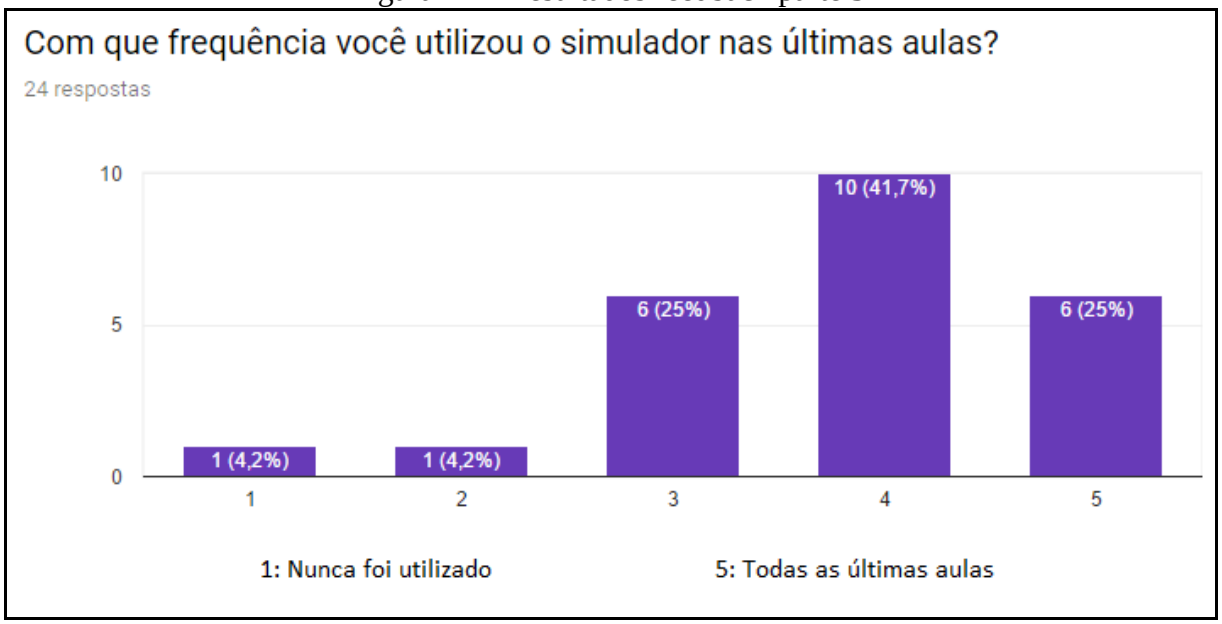

Fonte: elaborado pelo autor.

Figura 26 – Resultados feedback parte 2



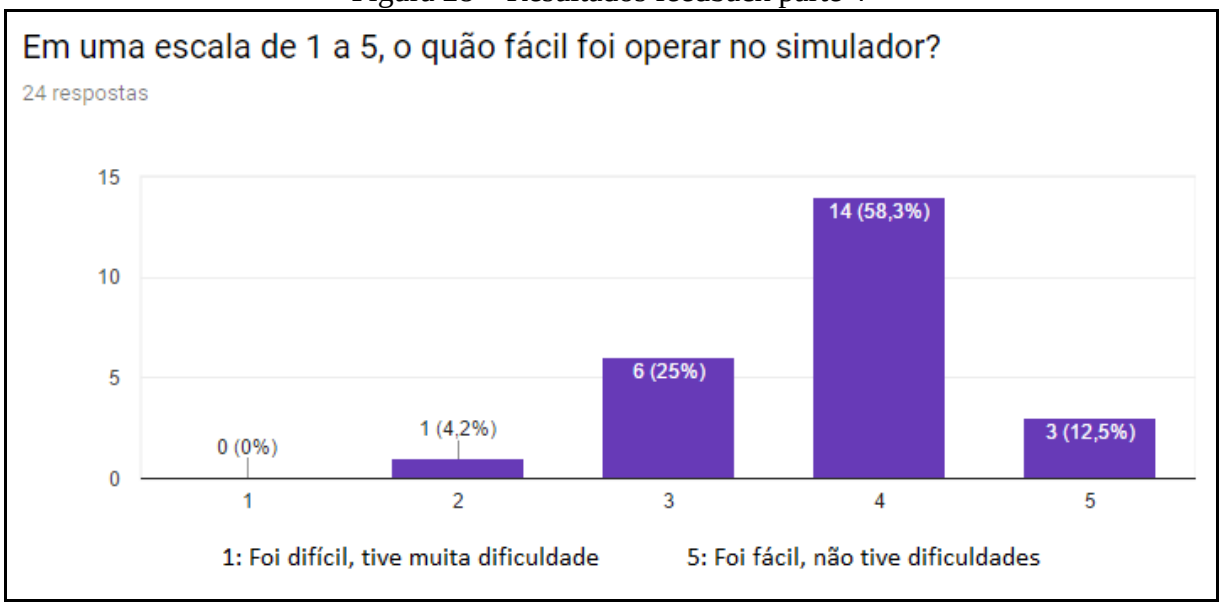

Fonte: elaborado pelo autor.

Figura 27 – Resultados feedback parte 3



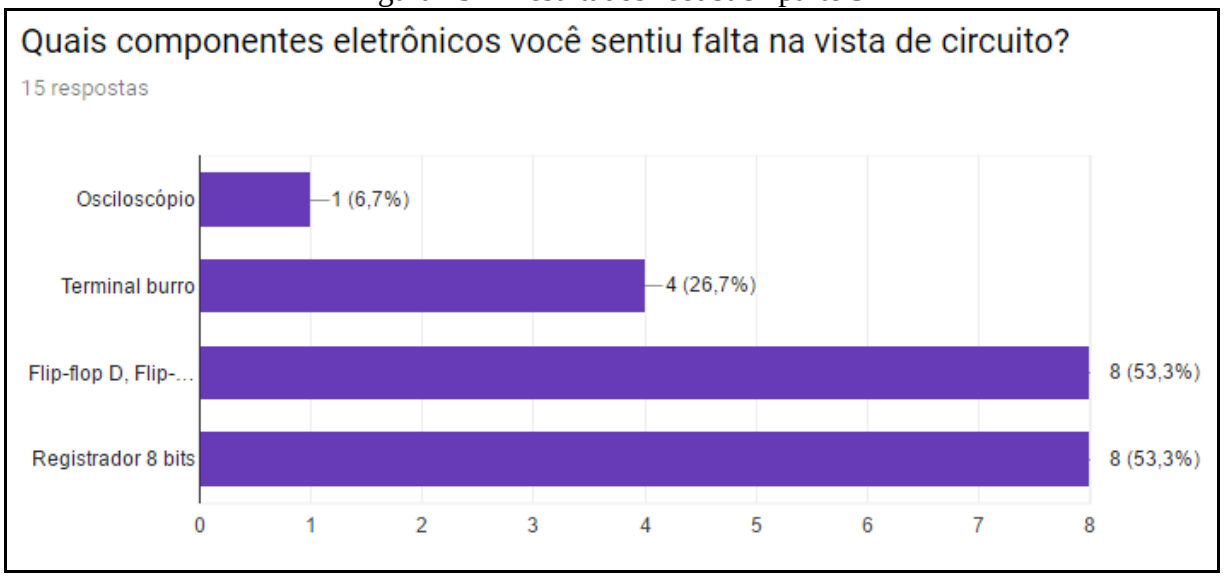

Fonte: elaborado pelo autor.

Figura 28 – Resultados feedback parte 4



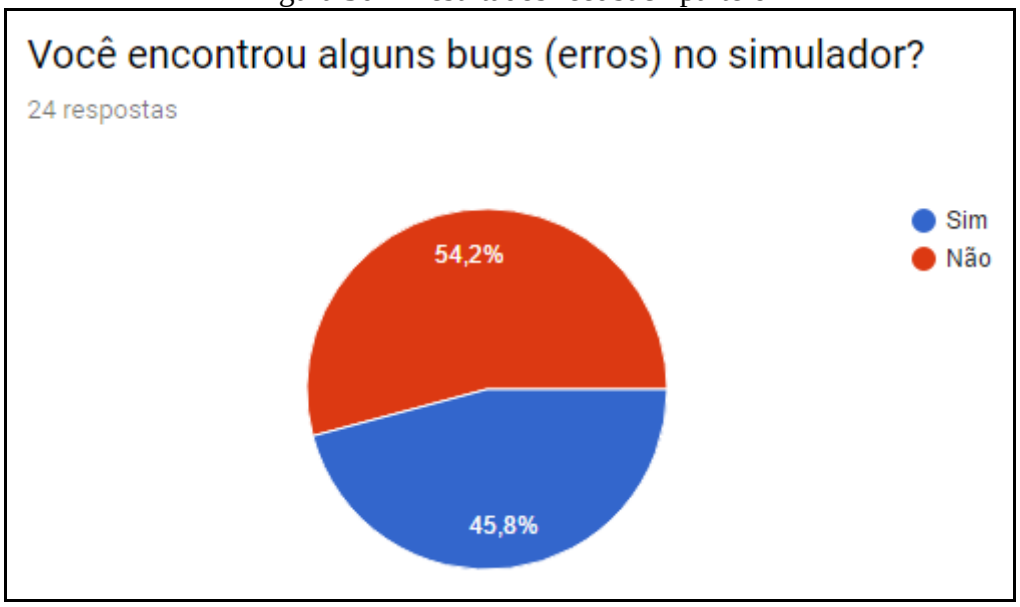

Fonte: elaborado pelo autor.

Figura 29 – Resultados feedback parte 5



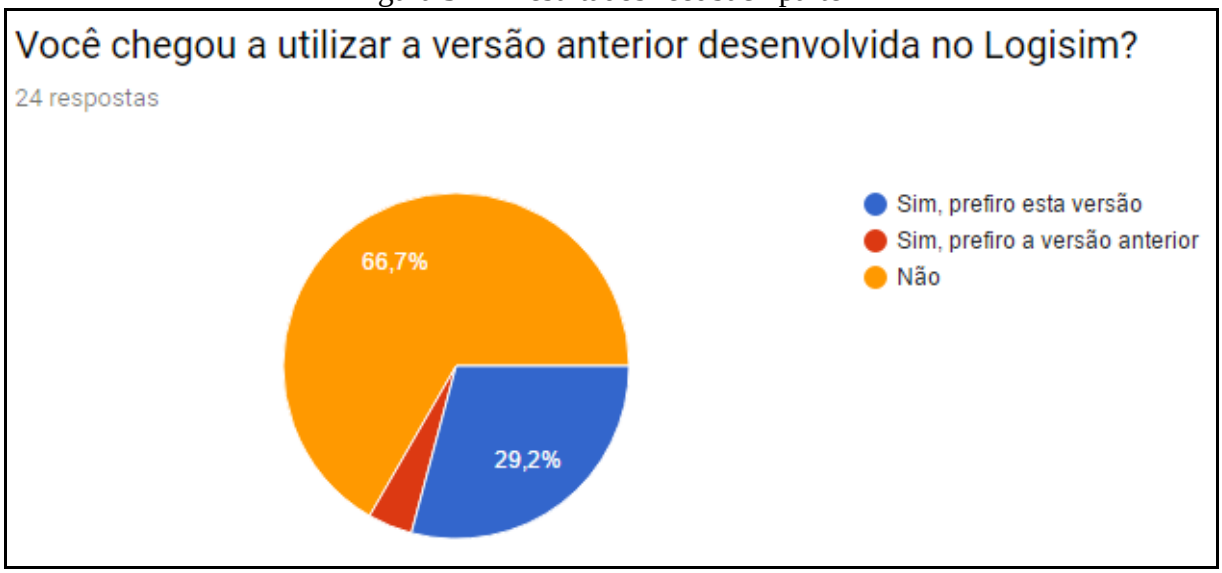

Fonte: elaborado pelo autor.

Figura 30 – Resultados feedback parte 6



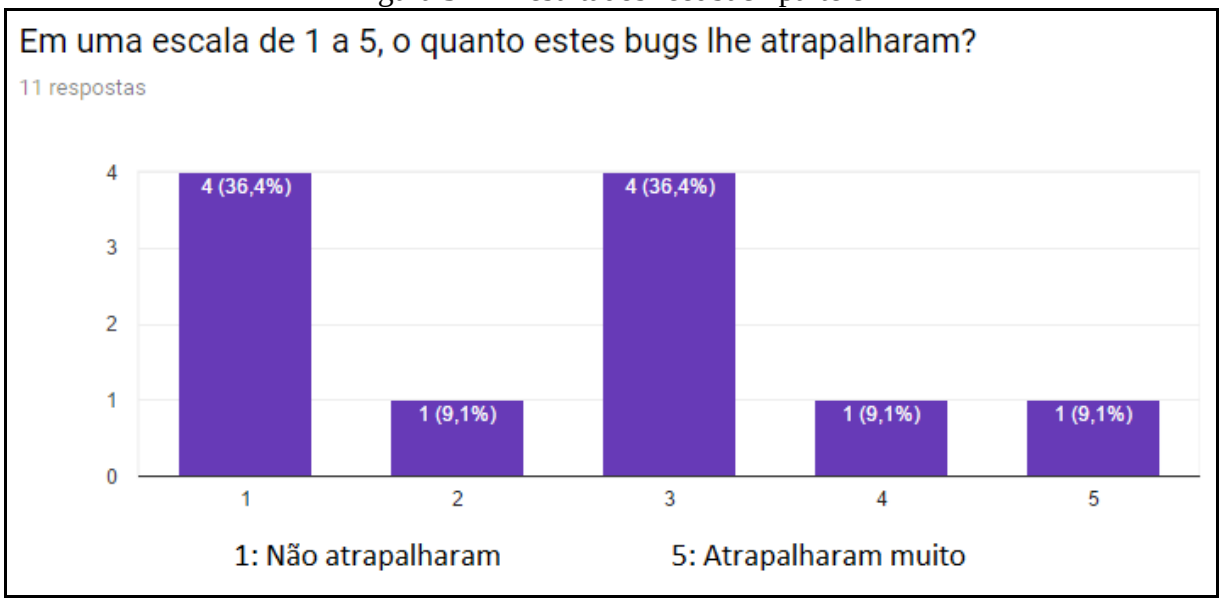

Fonte: elaborado pelo autor.

Figura 31 – Resultados feedback parte 7



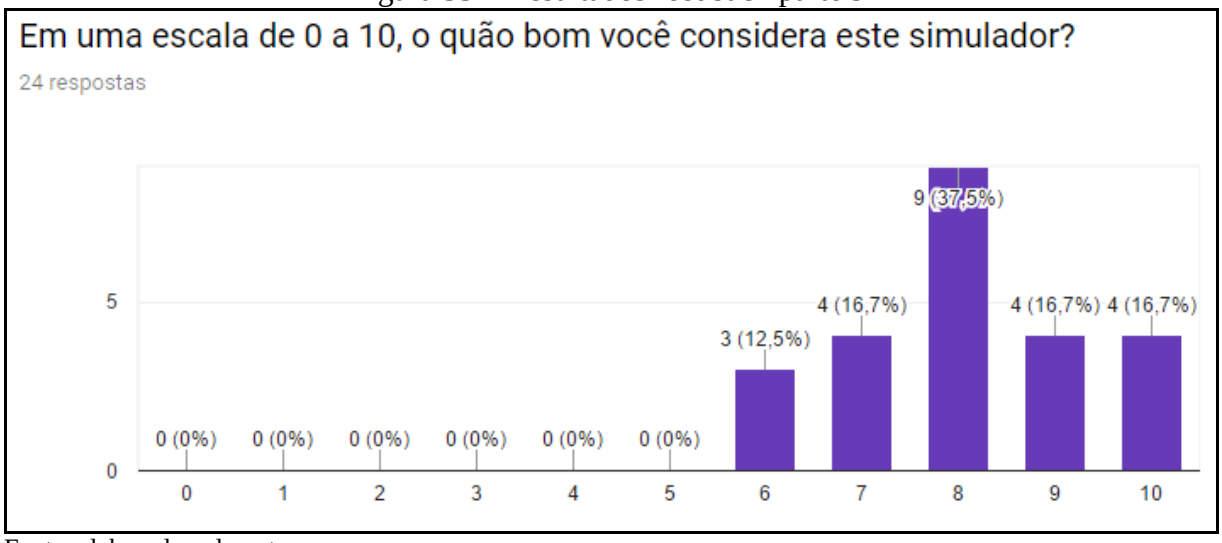

Fonte: elaborado pelo autor.

Figura 32 – Resultados feedback parte 8




Fonte: elaborado pelo autor.

Figura 33 – Resultados feedback parte 9




Fonte: elaborado pelo autor.