

# Montador e Simulador

## IJVM

**Andressa G. Moreira, Carlos Augusto M. de Pinho, Stefane Adna dos Santos, Cleton M. Soares.**

Universidade Federal do Ceará (UFC) – Campus Sobral, CE - Brasil

andressa\_gomes.12@hotmail.com, cmelodepinho@yahoo.com,  
stefaneadnas@gmail.com, cletonmoraissouares@gmail.com

**Abstract.** *This article aims to show the development of the IJVM assembler and simulator, as well as the receipt of its mnemonics and the binary equivalent based on the behavior of the IJVM virtual machine. In addition, for each instruction presented indicates its operation in execution, as well as the changes in the registers, presenting their values in each cycle and the modification in the data of the memory stack.*

**Resumo.** *Este artigo tem como objetivo mostrar o desenvolvimento do montador e simulador IJVM, assim como o recebimento dos seus mnemônicos e o equivalente em binário baseado no comportamento da máquina virtual IJVM. Ademais, para cada instrução apresentada indica-se a sua operação em execução, bem como as alterações nos registradores, apresentando seus valores em cada ciclo e a modificação nos dados da pilha de memória.*

## 1. Introdução

Microarquitetura é a forma como um determinado conjunto de instruções (ISA) é implementado em um processador, podendo ser implementado com microarquiteturas diferentes. As implementações podem variar devido a diferentes objetivos de um dado projeto ou a mudanças na tecnologia. A microarquitetura inclui os elementos constitutivos do processador e como estes interligam e interoperam para implementar o ISA. A ISA é aproximadamente o mesmo que o modelo de programação de um processador como visto por um programador de linguagem Assembly ou escritor de compilador. O ISA inclui o modelo de execução, registradores do processador, endereço e formatos de dados, entre outras coisas.

Máquina virtual Java – JVM, é um programa que carrega e executa os aplicativos Java, convertendo os bytecodes em código executável de máquina. A JVM é responsável pelo gerenciamento dos aplicativos, à medida que são executados. Graças à máquina virtual Java, os programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma versão da JVM, tornando assim essas aplicações independentes da plataforma onde funcionam.

A IJVM desenvolvido por Andrew S. Tanenbaum é um exemplo de microarquitetura, que tem por função implementar o nível ISA. O nível ISA está posicionado logo acima da microarquitetura, e é ele que define como a microarquitetura deve ser construída. A IJVM faz parte da JVM e possui apenas instruções que lidam com inteiros.

É comum em grande parte das linguagens de programação o uso de métodos. Esses métodos possuem variáveis locais que precisam ser armazenadas em algum lugar. Essas variáveis não possuem um endereço absoluto em memória, dessa forma é necessário o uso de dois registradores que ficarão responsáveis por controlar quais variáveis pertencem a determinada instrução/método. Uma área de memória, denominada pilha, é reservada para variáveis, mas variáveis individuais não obtêm endereços absolutos na memória. Por exemplo, dado um procedimento A, a estrutura de dados entre SP e LV é denominado quadro de variáveis locais de A. Dessa forma a memória é alocada apenas para procedimentos que estão ativos em dado momento.

Quando um procedimento retorna, a faixa de memória que estava sendo utilizada é liberada para o próximo procedimento. Essa mesma pilha pode ser utilizada para realizar cálculos aritméticos.

O modelo da memória JVM pode ser vista de duas maneiras: um arranjo de memória de 4.294.967.926 bytes (4 GB) ou um arranjo de 1.073,741.824 palavras, cada uma consistindo em 4 bytes. Em qualquer instante as seguintes áreas de memória são definidas:

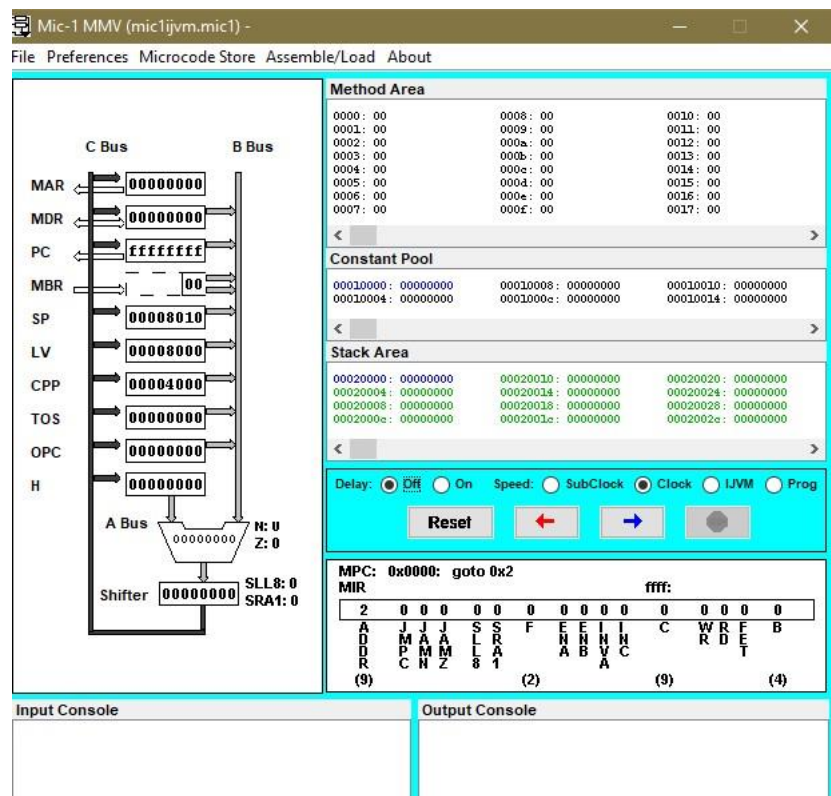
- **Conjunto de constantes:** Essa área não pode ser escrita por um programa JVM e consiste em constantes, cadeia de ponteiros para outras áreas da memória que podem ser referenciadas.
- **O quadro de variáveis locais:** Para cada invocação de um método é alocada uma área para armazenar variáveis durante o tempo de invocação, denominada quadro de variáveis locais.
- **A pilha de operandos:** É garantido que o quadro não exceda um certo tamanho, calculando com antecedência pelo compilador Java.
- **A área de método:** Por fim há uma região da memória que contém o programa, à qual é referido como área de ‘texto’ em um processo UNIX. Há um registrador implícito que contém o endereço da instrução a ser buscada em seguida. Esse ponteiro é denominado contador de programa (Program Counter) ou PC. Diferente de outras regiões da memória, a área de método é tratada como um arranjo de bytes.

Para o conjunto de instruções JVM cada instrução consiste em um UPCODE e às vezes um operando, tal como um deslocamento de memória ou uma constante. São fornecidas instruções para passar a pilha uma palavra que pode vir de diversas fontes. Essas fontes podem ser o conjunto de constantes (LDC\_W), quadro de variáveis locais (ILOAD) e a própria instrução (BIPUSH). Uma variável também pode ser retirada da pilha e armazenada no quadro de variáveis locais (ISTORE). Algumas instruções têm vários formatos, o que permite uma forma abreviada para versões comumente usadas.

2. MIC-I

O MIC-1 é uma arquitetura de processador que consiste em uma unidade de controle muito simples que executa o microcódigo criado utilizando-se instruções da IJVM. Essa arquitetura tem a função de interpretar instrução por instrução do nível ISA. Ademais, a figura 01, mostra a interface do MIC-I da IJVM.

Figura 01. MIC-I da IJVM



3. Apresentação problemática – IJVM

Sabe-se que IJVM é uma arquitetura de conjunto de instruções criada por Andrew Tanenbaum para sua arquitetura MIC-1. Por conseguinte, praticamente todas as linguagens de programação suportam o conceito de procedimentos, que tem variáveis locais, tais variáveis podem ser acessadas dentro dos procedimentos, mas deixam de ser acessíveis assim que o procedimento é devolvido. Dessa forma, é necessário um lugar da memória para manter essas variáveis. Assim, uma área na memória, denominada pilha, é reservada para o armazenamento de variáveis locais de um procedimento. Ademais, tem-se o conjunto de instrução da IJVM. Dessa forma, a tabela 01 ilustra tais instruções, nas quais a primeira coluna dá a codificação hexadecimal da instrução, a segunda fornece o

mnemônico em linguagem de montagem, enquanto a terceira fornece uma breve descrição do seu efeito.

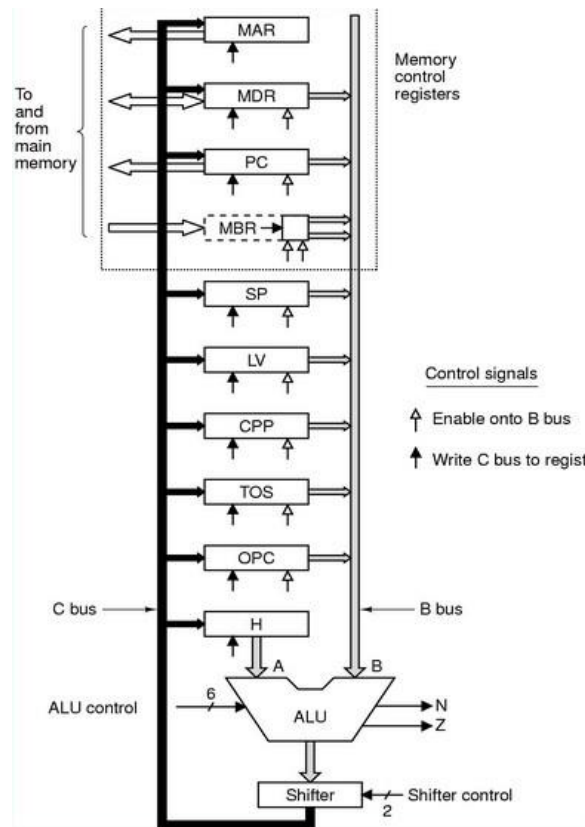
**Tabela 01. Instruções da IJVM**

Hexa	Mnemônico	Significado
0x10	BIPUSH <i>byte</i>	Colocar um byte na pilha
0x59	DUP	Copia a palavra do topo da pilha e coloca a cópia no topo da pilha
0xA7	GOTO <i>deslocamento</i>	Desvio incondicional
0x60	IADD	Retira da pilha as duas palavras do topo; coloca no topo da pilha o resultado da soma dessas palavras
0x7E	IAND	Retira da pilha as duas palavras do topo; coloca no topo da pilha o resultado da operação AND booleana dessas palavras
0x99	IFEQ <i>deslocamento</i>	Retira da pilha a palavra do topo; desvia se ela for igual a zero
0x9B	IFLT <i>deslocamento</i>	Retira da pilha a palavra do topo; desvia se ela for menor que zero
0x9F	IF_ICMPEQ <i>deslocamento</i>	Retira da pilha as duas palavras do topo; desvia se elas forem iguais
0x84	IINC <i>varnum const</i>	Soma uma constante a uma variável local
0x15	ILOAD <i>varnum</i>	Coloca uma variável local no topo da pilha
0xB6	INVOKEVIRTUAL <i>deslocamento</i>	Chama um procedimento
0x80	IOR	Retira da pilha as duas palavras do topo; coloca no topo da pilha o resultado da operação OR booleana dessas palavras
0xAC	IRETURN	Retorna de um procedimento trazendo um valor inteiro
0x36	ISTORE <i>varnum</i>	Retira a palavra do topo da pilha; armazena essa palavra numa variável local
0x64	ISUB	Retira da pilha as duas palavras do topo; coloca no topo da pilha o resultado da subtração dessas palavras
0x13	LDC_W <i>índice</i>	Coloca no topo da pilha uma constante vinda do pool de constantes
0x00	NOP	Não faz nada
0x57	POP	Retira da pilha a palavra do topo
0x5F	SWAP	Troca de posição as duas palavras do topo da pilha

#### 4. Registradores do MIC -1.

O caminho de dados é a parte do processador que contém a ULA e todas as suas entradas e saídas. A figura 02 apresenta uma simplificação do modelo de arquitetura MIC-1 e tem como principal característica uma das entradas da ULA (barramento A) sempre relacionado ao registrador “H”. Por outro lado, o barramento B pode receber dados de quaisquer outros registradores.

**Figura 02. Registradores do MIC-I**



Nota-se que os quatro primeiros registradores (MAR, MDR, PC e MBR) são registradores que trabalham diretamente com os endereços na memória. Enquanto que MAR e PC apenas enviam para a memória, MBR recebe e é de comunicação exclusiva com a memória, não recebendo dados de outros registradores através do barramento C. Já MDR trabalha tanto enviando como recebendo dados referentes à memória. Os demais registradores de controle podem receber dados de outros registradores através do barramento C, e trabalham enviando dados ao barramento B, ligado à ULA. Outro fator importante é que a ULA apresenta duas “flags” de controle, indicando se um resultado de uma operação é nulo ou não (flag “Z”) ou se é negativo (flag “N”). Assim, os registradores e suas respectivas funções são:

- **MDR:** Registrador de Dados de Memória para o barramento de 32 bits, apontado por MAR.
- **MBR:** Registrador de Dados de Memória para o barramento de 8 bits, apontado por PC.

- **PC:** Contador de Programa. Aponta para a Memória em um Barramento de 8 bits, que contém instruções.
- **SP:** Apontador de Pilha.
- **LV:** Apontador para a base das Variáveis Locais, localizada na pilha.
- **CPP:** Aponta para o POOL de constantes e apontadores para outras áreas da memória.
- **TOS e OPC:** Registradores Temporários.
- **H:** Acumulador.

## 5. Implementação do Código – O Montador.

O montador é responsável por receber as diversas instruções que foram escritas com os mnemônicos da IJVM e convertê-las em linguagem de máquina. Para que essa conversão pudesse ocorrer de forma satisfatória, foi feito um código em Python que realizava manipulação de *Strings*, que possuía diversas funções e estruturas de repetição, de modo a verificar a sintaxe do código que foi escrito, se haviam erros na escrita de alguma instrução, verificar se havia a entrada de valores na pilha, o armazenamento em alguma variável ou a retirada desse valor da pilha e por fim, mostrar ao usuário, depois de escrever as instruções em IJVM, o equivalente de cada uma delas em suas respectivas equivalentes em hexadecimal.

Também, foi criada uma interface gráfica com o auxílio do software QT Design. Nesse programa, só é possível a criação de interfaces gráficas em C++. Mas, com o auxílio do download de uma Biblioteca chamada de PyQt, é possível a implementação de códigos utilizando a linguagem Python quando essa biblioteca é importada no código e com isso, mesmo a interface gráfica sendo feita em C++, é possível manipular seus atributos, como os botões e as janelas, utilizando-se conceitos de Programação Orientada a Objetos voltados para a linguagem Python. Em outras palavras, essa biblioteca é especializada em converter um código escrito em C++ que foi desenvolvido no QT Design para um código em Python.

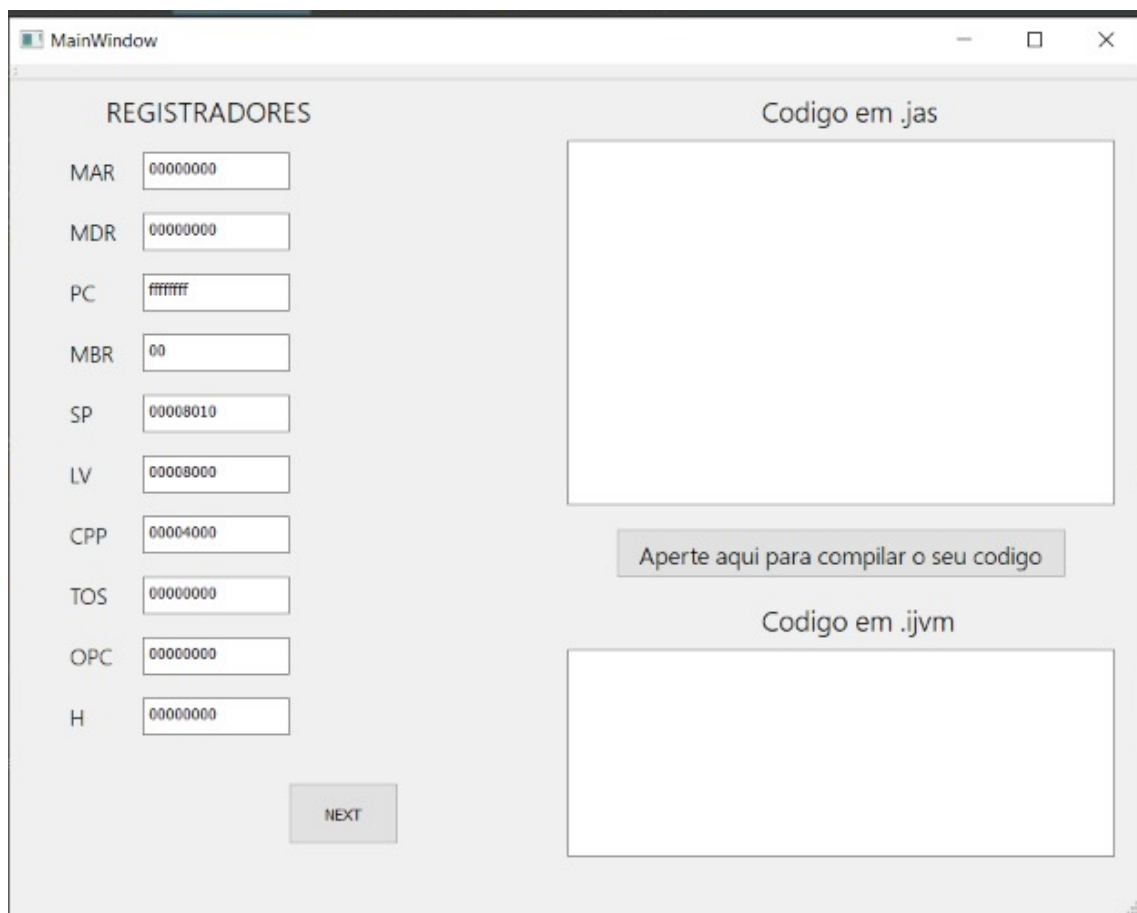
No código fonte feito em Python, foram feitas funções que conseguem manipular o código fonte escrito em um arquivo em formato .jas. Por exemplo, para a função BIPUSH, foi feita uma função que quando o programa, ao verificar que foi escrita essa instrução, converte essa palavra em seu equivalente Hexadecimal, e também o número que vem seguido dessa palavra. Além disso, foram criadas outras funções para verificar a presença das instruções ILOAD, ISTORE, e GOTO, que são funções em que o montador terá de retirar

algum valor da pilha ou fazer um desvio incondicional nas linhas do código, o que poderá ter alguma mudança na representação do código em linguagem de máquina com seus equivalente em Hexadecimal. Essas funções foram importantes para se fazerem as conversões adequadas.

Também, foram criadas funções com o objetivo de verificar o início do programa, que começa com a instrução e em seguida com a declaração das variáveis e só termina quando é escrito o .end-main.

A interface criada no QT design para o montador e simulador IJVM é a mostrada na figura abaixo:

**Figura 3 - Interface gráfica do Montador e Simulador IJVM**



Pode-se observar que nessa interface, existe um editor de texto onde pode-se inserir o código em .jas, e nele digita-se qualquer instrução da IJVM, e quando clica-se no botão ‘Aperte aqui para compilar o seu código’, na janela seguinte do programa, aparece o código em IJVM, com seus equivalentes em Hexadecimal, realizando assim a função do montador. Como as instruções da IJVM são colocadas em uma fila de operandos onde há o caminho de

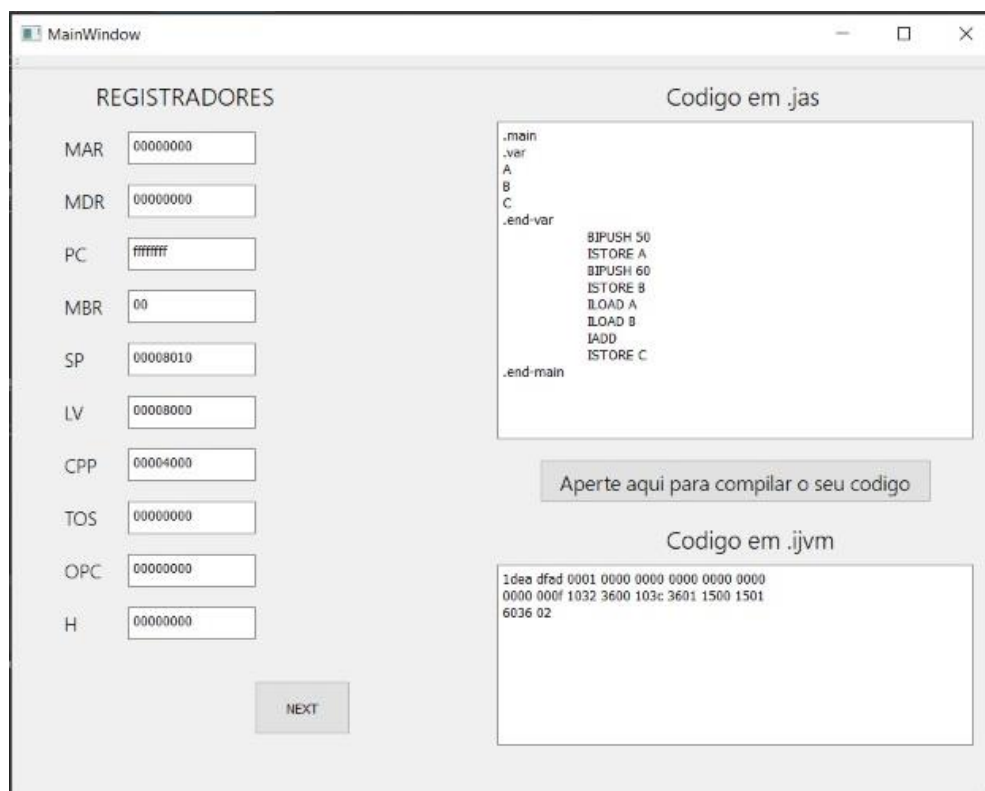


dados do processador, sabe-se que haverá inúmeras mudanças nos registradores quando uma única instrução é lida. Essa interface também mostra como ocorrerá a atualização dos valores de cada registrador ao longo do caminho de dados. Cada um dos registradores tem uma função específica, e de acordo com as instruções escritas, até o final da execução, eles apresentarão seus valores correspondentes.

## Testes

Nesse simulador IJVM, utilizou-se alguns exemplos de modo a verificar o funcionamento dessa interface, como mostra-se nas figuras a seguir:

**Figura 4 - Simulação de um exemplo na Interface Gráfica**

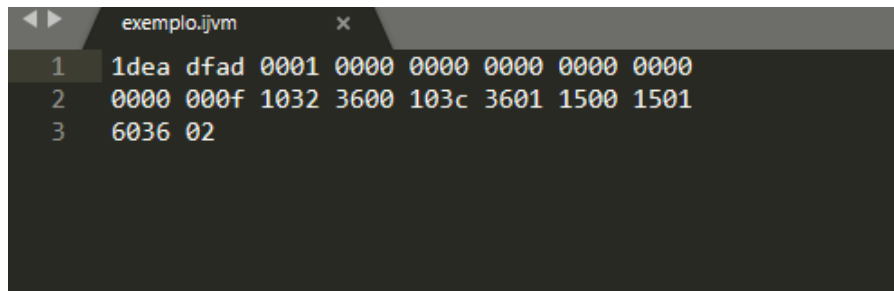


Pode-se ver que nesse exemplo, por meio da instrução BIPUSH, coloca-se o número 50 na pilha e armazena-se ele na variável A(Instrução ISTORE A). Após isso, insere-se o número 60 também na pilha e o armazena na variável B(Instrução ISTORE B). Daí, coloca-se as variáveis A e B no topo da pilha(por meio das instruções ILOAD A e ILOAD B) e a instrução seguinte, que é a IADD, faz com que elas sejam somadas e armazena-se o valor dessa soma na variável C (ISTORE C). Quando se escreve. end-main, marca-se então onde a execução deve parar. Com isso, ao se clicar no botão 'Aperte aqui para compilar seu código', pode-se notar que os equivalentes em hexadecimal aparecem mostrados na tela.

Pode-se notar que esse código de instruções serve para somar dois números que são inseridos na pilha de operandos.

Quando simulamos esse mesmo conjunto de instruções na microarquitetura MIC-1, pode-se notar que o resultado do código em IJVM será o mesmo:

**Figura 5 - Código em IJVM ao ser executado no MIC-1.**

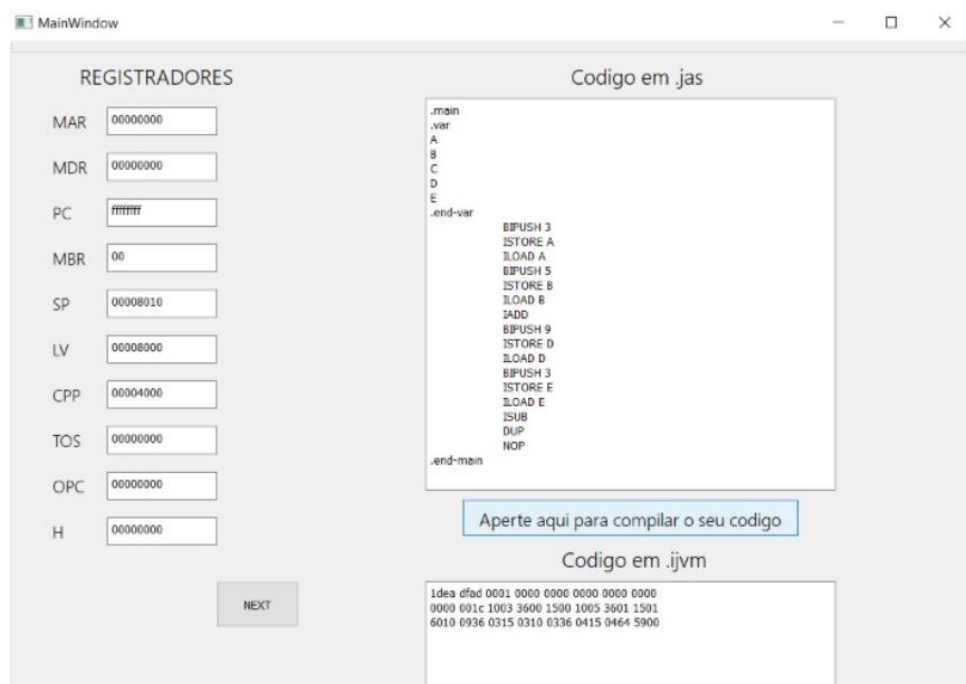


```
1 1dea dfad 0001 0000 0000 0000 0000 0000
2 0000 000f 1032 3600 103c 3601 1500 1501
3 6036 02
```

Pode-se notar que o código IJVM com seu equivalente em Hexadecimal mostrado na microarquitetura MIC-1 foi o mesmo desenvolvido na interface gráfica do simulador IJVM. Os valores dos registradores desse programa também serão atualizados, instrução por instrução, cada vez que o botão NEXT for pressionado, pra assim analisar-se como se comportaram os registradores a cada instrução que foi lida no simulador.

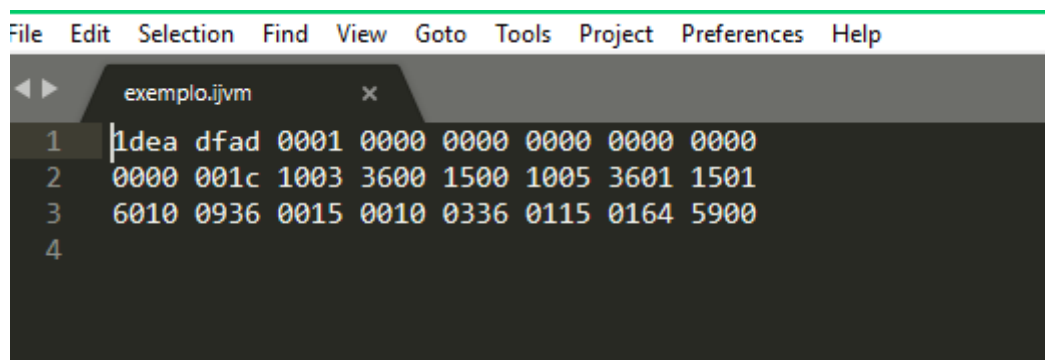
Fazendo outro exemplo um tanto mais complexo, inseriu-se na interface gráfica o seguinte código:

**Figura 6 -Simulador IJVM com outro exemplo.**



Nesse código em .jas, por meio da função BIPUSH, insere-se os valores 3 e 5 na pilha, e os mesmos são armazenados nas variáveis A e B, respectivamente, onde as mesmas são colocadas no topo da pilha. Em seguida, esses valores são somados, por conta da presença da instrução IADD. Daí, novamente, os valores 9 e 3 são inseridos na pilha, e armazenados nas variáveis D e E, e as variáveis são colocadas no topo da pilha. Em seguida, esses valores foram subtraídos, por conta da instrução ISUB. Daí, os valores foram duplicados na pilha por meio da instrução DUP. A instrução NOP no final indica que não deve mais ser feito NADA. Abaixo do editor de texto, pode-se observar o código em IJVM com suas equivalentes em Hexadecimal. Quando esse código é executado na microarquitetura MIC-1, o resultado do código em IJVM é o mostrado a seguir:

**Figura 7 - Código em IJVM ao ser executado no MIC-1.**



The image shows a screenshot of a text editor window titled 'exemplo.ijvm'. The editor contains four lines of code, each with a line number on the left. The code consists of IJVM instructions followed by their hexadecimal equivalents. Line 1: 'ldea dfad 0001 0000 0000 0000 0000 0000'. Line 2: '0000 001c 1003 3600 1500 1005 3601 1501'. Line 3: '6010 0936 0015 0010 0336 0115 0164 5900'. Line 4: (empty line).

```
1 ldea dfad 0001 0000 0000 0000 0000 0000
2 0000 001c 1003 3600 1500 1005 3601 1501
3 6010 0936 0015 0010 0336 0115 0164 5900
4
```

Pode-se ver que o código em IJVM da microarquitetura MIC-1 é o mesmo da interface gráfica do simulador. Com isso, pode-se observar que o montador e simulador atende a forma de como funciona a conversão das instruções escritas da IJVM para os seus equivalentes em Hexadecimal e também no que diz respeito ao funcionamento dos Registradores.

## **6. Conclusão**

A priori, foi mister entender os conceitos de microarquitetura, na qual é a forma como um determinado número de instruções em nível ISA serão implementadas no processador. Por conseguinte, o MIC-1 é uma arquitetura de processador que consiste em executar o microcódigo criado utilizando-se instruções da IJVM, e tem como principal função interpretar instrução por instrução do nível ISA.

Desse modo, tornou-se possível desenvolver um montador instruções IJVM que recebe os mnemônicos e transforma no equivalente em binário. Utilizou-se a linguagem de alto nível Python para desenvolver tal programa. Ademais, fez-se o simulador, na qual interessou-se por simular o comportamento da arquitetura da máquina virtual IJVM, ler o código gerado pelo montador e carregar em sua memória como mnemônico.

Por fim, pode-se analisar o comportamento das instruções da IJVM, na qual geralmente são escritos em mnemônicos. Também se entendeu sobre a função dos registradores da MIC1 e como ocorre a modificação dos dados na pilha de memória.

## 7. Referências

- [1] TANENBAUM, Andrew S. Organização Estruturada de Computadores. 5°. São Paulo: Pearson Prentice Hall, 2013.
  
- [2] WIKPÉDIA. Nível Microarquitetura. Disponível em: <<https://pt.wikipedia.org/wiki/Microarquitetura> .> Acesso 19 de junho de 2019.
  
- [3] PREZI. Exemplos de execuções de instrução IJVM. Disponível em: <<https://prezi.com/km-p7mwyqaow/exemplo-de-execucao-de-instrucoes-ijvm/> >. Acesso em 20 de junho de 2019.
  
- [4] JENSE BRASILEIRO, Microarquiteturas MIC. Disponível em: <<https://jansebp.wordpress.com/2013/08/20/microarquiteturas-mic-1-e-mic-2-parte-i-apresentacao/>> Acesso em 22 de junho de 2019.