UNIVERSIDADE ESTADUAL DE SANTA CRUZ

ALLANA DOS SANTOS CAMPOS

MODELO SEMÂNTICO DE CONTROLE DE FLUXO E CHAMADA E RETORNO DE SUB-ROTINA PARA HARDWARE VIRTUAL

ILHÉUS - BAHIA

2022

ALLANA DOS SANTOS CAMPOS

MODELO SEMÂNTICO DE CONTROLE DE FLUXO E CHAMADA E RETORNO DE SUB-ROTINA PARA HARDWARE VIRTUAL

Pré-projeto de trabalho de conclusão de curso apresentado à Universidade Estadual de Santa Cruz como parte das exigências para graduação do curso de Bacharelado em Ciência da Computação.

Área de concentração: Virtualização, Hardware Virtual.

Orientador: Prof. Dr. César Alberto Bravo Pariente.

ILHÉUS - BAHIA

2022

ALLANA DOS SANTOS CAMPOS

MODELO SEMÂNTICO DE CONTROLE DE FLUXO E CHAMADA E RETORNO DE SUB-ROTINA PARA HARDWARE VIRTUAL

Trabalho de conclusão de curso apresentado à Universidade Estadual de Santa Cruz como parte das exigências para graduação do curso de Bacharelado em Ciência da Computação.

Área de concentração: Virtualização, Hardware Virtual.

Orientador: Prof. Dr. César Alberto Bravo Pariente.

Ilhéus, 06 de Julho de 2022.

Prof. Dr. César Alberto Bravo Pariente

UESC/DCET

(Orientador)

Prof. Dr. Marcelo Ossamu Honda

UESC/DCET

Prof. Dr. Sergio Fred Ribeiro Andrade

UESC/DCET

AGRADECIMENTOS

Á minha mãe, Marineide, que foi uma mãe e um pai, cuidou e acreditou em mim, me apoiou e me incentivou durante toda minha vida, me ensinou a ser como sou hoje.

Ao meu pai, Orlando, por ser um homem amoroso e cuidadoso e, mesmo que tenha partido tão cedo deixou seus ensinamentos.

Á minha irmã, Alexsandra, por estar presente em meus momentos de estresse, ter me incentivado a não desistir, e por se interessar um pouco sobre o meu mundo particular.

Ao meu namorado, Vinícius, por sempre me escutar independente do assunto, por me levar a sério independente do problema e por ficar feliz a cada conquista mesmo que seja pequena.

Ao meu cachorro, Toddy, por sempre estar feliz ao me ver, me fazer companhia durante meus anos, se divertir junto comigo, e mesmo sendo um ser tão pequeno, me defender.

Á minha amiga, Rebeca, por continuar sendo minha amiga durante 18 anos, por não se esquecer nem se afastar mesmo quando não estávamos próximas fisicamente.

Á minha amiga, Joildes, por estar presente na minha vida, me dando concelhos, indicações de livros, e por alegrar meus dias com sua coleção de figurinhas animadas.

Aos meus amigos por me proporcionarem o prazer de sua companhia, me permitindo momentos de descontração e também de reflexão.

Ao prof. Dr. César Alberto Bravo Pariente por ter me concedido a oportunidade de participar em seus projetos, por me orientar durante esta jornada e pelos seus ensinamentos.

Á FAPESB pelo incentivo em forma de bolsa de iniciação científica nas cotas de 2020/2021 e 2021/2022.

Por fim, aos professores que me ensinaram e participaram positivamente durante os meus anos na UESC.

MODELO SEMÂNTICO DE CONTROLE DE FLUXO E CHAMADA E RETORNO DE SUB-ROTINA PARA HARDWARE VIRTUAL

RESUMO

Este trabalho descreve a implementação de um modelo semântico de controle de fluxo e chamada e retorno de sub-rotina para hardware virtual e, apresenta seu desempenho comparado com a máquina P-code, especificada por Niklaus Wirth no livro “Algorithms + Data Structures = Programs”, e a máquina descrita por Brookshear em seu livro “Ciência da Computação: Uma Visão Abrangente”. A virtualização permite a utilização de vários softwares baseados em diferentes sistemas operacionais em uma mesma máquina física; para que ocorra a virtualização comumente utiliza-se um monitor de máquinas virtuais, conhecido também como hipervisor. O hipervisor serve como uma interface entre a máquina física e as máquinas virtuais, ele é responsável por controlar e distribuir os recursos de hardware entre as máquinas virtuais instaladas. A máquina virtual desenvolvida, AllanaVM, possui 2^{32} células de memória com 64 bits de comprimento, conjunto de instruções contendo 25 instruções de 32 bits cada, 256 registradores, arquitetura Von Neumann e, um modelo de controle de fluxo e chamada e retorno de sub-rotina, permitindo recursão. A máquina de Brookshear também possui arquitetura Von Neumann porém não possui infraestrutura para chamada e retorno de sub-rotina; já a P-code possui infraestrutura para chamada e retorno de sub-rotina porém, arquitetura Harvard. A arquitetura da AllanaVM baseou-se na arquitetura da máquina de Brookshear e estendeu-se com o modelo de controle de fluxo e chamada e retorno de sub-rotina e o conjunto de instruções da P-code, tais instruções são armazenadas na memória da AllanaVM e foram desenvolvidas utilizando o conjunto de instruções da máquina de Brookshear. Para os testes do funcionamento da AllanaVM, desenvolveu-se algoritmos para o cálculo do fatorial e da sequência de Fibonacci, com implementações iterativas e recursivas, os mesmos algoritmos também foram desenvolvidos para as máquinas de Brookshear e P-code, para a comparação do desempenho entre as mesmas. Mesmo que as arquiteturas das três máquinas sejam diferentes, comparou-se o desempenho entre a AllanaVM e a P-code e, entre a AllanaVM e a máquina de Brookshear, como resultados, em 25% dos testes a máquina AllanaVM é a mais veloz na execução de seus algoritmos e, em 100% dos testes a máquina AllanaVM permite maior quantidade de iterações ou chamadas de sub-rotina.

Palavras-chave: Virtualização. Máquina Virtual. Hipervisor.

SEMANTIC MODEL OF FLOW CONTROL AND SUBROUTINE CALL AND RETURN FOR VIRTUAL HARDWARE

ABSTRACT

This work describes the implementation of a semantic model of flow control and subroutine call and return for virtual hardware and, presents its performance compared to the P-code machine, specified by Niklaus Wirth in the book “Algorithms + Data Structures = Programs”, and the machine described by Brookshear in his book “Computer Science: A Comprehensive View”. Virtualization allows the use of various software based on different operating systems on the same physical machine; for virtualization to occur, a virtual machine monitor, also known as a hypervisor, is commonly used. The hypervisor serves as an interface between the physical machine and the virtual machines, it is responsible for controlling and distributing the hardware resources among the installed virtual machines. The virtual machine developed, AllanaVM, has 2^{32} memory cells with 64 bits in length, an instruction set containing 25 instructions of 32 bits each, 256 registers, Von Neumann architecture and a flow control model and subroutine call and return, allowing recursion. Brookshear's machine also has Von Neumann architecture but does not have infrastructure for subroutine call and return; P-code has infrastructure for subroutine call and return, however, Harvard architecture. The AllanaVM architecture was based on the Brookshear machine architecture and extended with the flow control model and subroutine call and return and the P-code instruction set, such instructions are stored in the AllanaVM memory and were developed using the Brookshear machine instruction set. To test AllanaVM’s functionatity, algorithms were developed to calculate the factorial and the Fibonacci sequence, with iterative and recursive implementations, the same algorithms were also developed for the Brookshear and P-code machines, to compare the performance between them. Even though the architectures of the three machines are different, the performance was compared between AllanaVM and P-code and, between AllanaVM and Brookshear's machine, as a result, in 25% of the tests the AllanaVM machine is the most fast in the execution of its algorithms and, in 100% of the tests, the AllanaVM machine allows more iterations or subroutine calls.

Keywords: Virtualization. Virtual Machine. Hypervisor.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

A virtualização permite que aplicações de diversos sistemas operacionais sejam processados em uma mesma máquina, para isso é necessário um sistema operacional executado por uma máquina física e um sistema virtualizado. Para a comunicação entre esses dois fatores, comumente é utilizado um Monitor de Máquina Virtual, também conhecido como hipervisor, que controla os recursos utilizados pelo sistema virtualizado (TANENBAUM; BOS, 2016).

Nos anos que se seguiram ao surgimento dos primeiros computadores era comum que cada computador possuísse seu próprio sistema operacional, não sendo possível a execução de um mesmo software, em duas máquinas com diferentes sistemas operacionais; a virtualização foi introduzida como um meio para contornar este problema.

Em 1936, Turing foi o primeiro a idealizar uma máquina virtual ao descrever a máquina universal de Turing, que podia simular qualquer outra máquina de Turing; no entanto a primeira implementação das máquinas virtuais ocorreu nos anos 60 com o lançamento do System/360 pela (IBM, 2022), uma linha de mainframes que permitia que aplicações desenvolvidas em modelos da mesma série fossem simulados nas demais máquinas, mantendo sua compatibilidade (TANENBAUM; BOS, 2016),

Com a virtualização o sistema operacional de preferência do usuário poderia ser instalado em sua máquina; consequentemente houve a extinção de vários outros sistemas operacionais que se tornaram obsoletos, diminuindo a quantidade disponível para aplicações que formam um pequeno grupo nos dias atuais.

Existem vários tipos de virtualização, como a virtualização de rede, na qual os recursos de uma rede são compartilhados entre diferentes canais, ou a virtualização de aplicativo, que permite executar um aplicativo separado do dispositivo que o acessa; um outro tipo de virtualização comum, utilizado neste projeto, é a virtualização de hardware, que possibilita a criação de uma máquina virtual com um comportamento semelhante à um computador real.

O fluxo de execução de um código é controlado pelas instruções do programa, sem operações de desvios o fluxo de execução é contínuo, de modo que as instruções são acessadas na ordem que foram escritas; os controles de fluxo são operações que manipulam o fluxo de execução, são estas, operações de desvios, como *jump* e *go to*,

que podem ser condicionais ou não e, iterações, que são os laços de repetição como *while* e *for* (KUBRUSLY, 2021).

Para que seja possível a execução de sub-rotinas em um programa, é necessário a utilização de operações de desvio para a chamada da sub-rotina e, para que uma sub-rotina seja executada recursivamente é essencial o armazenamento do estado atual da sub-rotina antes da nova chamada; para isso comumente se utilizam registros de ativação, que podem ser implementados em posições consecutivas de uma pilha, contendo as variáveis locais, parâmetros, dentre outros componentes da sub-rotina.

O modelo semântico da pilha de execução, contendo os registros de ativação para cada sub-rotina, apresenta as conexões entre as sub-rotinas através da utilização de ponteiros, indicando a ordem de execução das sub-rotinas.

1.2 Motivação

A máquina de Turing em sua idealização inicial é uma máquina monolítica, na qual se permite a utilização de instruções de desvio, porém não permite uma melhor estruturação de controle, como iterações, subdivisões ou recursões, consequentemente não oferece infraestrutura para chamada e retorno de sub-rotinas; tal problema foi contornado por (HERMES, 1965).

Na atualidade qualquer linguagem de programação implementa e permite chamada e retorno de sub-rotina mas para isso a máquina objeto tem que oferecer infraestrutura; em particular, nos cursos de Ciência da Computação, na matéria de Conceitos de Linguagem de Programação se estuda de forma conceitual as infraestruturas de chamada e retorno de sub-rotina.

Para implementar um compilador na matéria de Compiladores é necessário conhecer o suporte de hardware porém, os microprocessadores modernos são complexos que os alunos não tem a vivência necessária para trabalhar em um microprocessador real; sendo conveniente a implementação de uma ferramenta didática, como um hardware virtual para melhor entendimento dos alunos.

Cada arquitetura de computadores possui suas vantagens e desvantagens; como por exemplo a arquitetura Harvard proporciona maior segurança à memória de programa e a arquitetura Von Neumann permite a alteração do programa em tempo de execução; com a versatilidade da arquitetura é possível a implementação de um escalonador de processos, que é uma parte essencial do sistema operacional, razão pela qual optou-se

por desenvolver um hardware virtual com tal arquitetura, aliada a dificuldade do alunado narrada anteriormente pela complexidade de entendimento dos microprocessadores.

Dentre os simuladores de hardwares virtuais disponíveis encontrou-se o de (BROOKSHEAR, 2005) que possui arquitetura Von Neumann mas não apresenta infraestrutura de chamada e retorno de sub-rotina e, a P-code, descrita por (WIRTH, 1990), que possui infraestrutura de chamada e retorno de sub-rotina porém arquitetura Harvard; para alcançar o hardware virtual desejado, optou-se por expandir a arquitetura da máquina de Brookshear com infraestrutura de chamada e sub-rotina da P-code.

1.3 Objetivo geral

Este trabalho tem como objetivo apresentar a descrição e implementação de um modelo semântico de controle de fluxo e chamada e retorno de sub-rotina para hardware virtual.

1.4 Objetivos específicos

Este trabalho possui os seguintes objetivos específicos:

a) determinar a arquitetura para máquina virtual;
b) desenvolver um conjunto de instruções para a máquina virtual;
c) desenvolver um modelo semântico para controle de fluxo de chamada e retorno de sub-rotina;
d) implementar algoritmos para teste de controle de fluxo de chamada e retorno de sub-rotina.

1.5 Justificativa

Para (GHANNOUM; RODRIGUES, 2018), a virtualização proporciona várias vantagens ao usuário da tecnologia, como:

a) otimização da utilização dos recursos, em vez de manter a unidade central de processamento (CPU) dedicada a apenas um servidor, a virtualização permite que os recursos da máquina sejam distribuídos entre os servidores aproveitando o máximo de capacidade de computação do hardware;
b) melhor gerenciamento do hardware: com a virtualização é possível gerenciar a utilização do hardware por máquina virtual, assim quando uma máquina virtual estiver utilizando menos recursos em determinada tarefa, em um determinado

tempo, pode-se realocar os recursos que estão ociosos para outras máquinas, além da deleção ou criação das mesmas de maneira rápida;

c) diminuição dos custos, por possibilitar o processamento de diferentes sistemas operacionais em uma mesma máquina, a quantidade de máquinas em funcionamento diminui, proporcionando menores custos com manutenção, energia e espaço utilizado;
d) suporte a aplicações que utilizam tecnologias antigas, caso seja necessário a utilização de algum aplicativo, software. que não foi atualizado, de modo que não é compatível com os sistemas operacionais atuais, pode-se fazer uso de uma máquina virtual para execução de tais aplicações;
e) segurança, como cada máquina fica isolada das demais, caso utilize cada uma para um serviço, a vulnerabilidade do sistema se torna menor, além de que a virtualização permite a escolha do melhor ambiente para execução de cada serviço e, com isso a utilização das ferramentas de segurança existentes no mesmo, como backup.

1.6 Organização

O presente trabalho visa apresentar a descrição e implementação de um modelo semântico de controle de fluxo de chamada e retorno de sub-rotina para hardware virtual, organizado da seguinte maneira: o capítulo 2 contém o referencial teórico, o capítulo 3 descreve os materiais e métodos utilizados, o capítulo 4 apresenta a discussão dos resultados obtidos e o capítulo 5 as conclusões e trabalhos futuros. Ao final encontram-se as referências utilizadas e os apêndices A, B e C que descrevem o conjunto de instruções de cada máquina virtual utilizada.

2 REFERENCIAL TEÓRICO

Este capítulo define o que é virtualização e os componentes que são necessários para sua utilização, descreve como o fluxo de controle é determinado e os componentes para organizar as chamadas e retornos de sub-rotinas e, apresenta trabalhos semelhantes ao desenvolvido e o estado da arte das máquinas virtuais.

2.1 Virtualização

A virtualização teve seu início nos anos 1960, na época os mainframes eram computadores com muitos recursos e grande velocidade de processamento que, por muitas vezes eram mantidos ociosos pois, os processos eram gerenciados manualmente pelo operador da máquina, para contornar o problema a IBM desenvolveu o System/360, uma família de computadores que permitiam executar diversos sistemas operacionais simultaneamente, além de poderem trabalhar juntos, iniciando uma era de compatibilidade (IBM, 2022).

Em 1972 a IBM lançou o System/370, o sucessor do System/360 e a virtualização se difundiu nos anos 1980 por conta do baixo custo para utilização da técnica; em 1990 o System/370 foi substituído pelo System/390 e nos anos 2000 a IBM lançou a série z, suportando endereçamento de 64 bits mantendo ainda a compatibilidade com as outras séries (TANENBAUM; BOS, 2016).

No início da ascensão da plataforma x86 juntamente aos sistemas operacionais Windows e Linux a virtualização não era suportada. Em 1997 a Connectix introduziu a virtualização através do Virtual PC (WIKIPEDIA, 2022), um aplicativo de Macintosh para System 7.5 e, a partir de 1999 a virtualização foi introduzida na plataforma x86 pela (VMWARE, 2021), de modo que a partir de 2005 os fabricantes de processadores melhoraram seu suporte via hardware, permitindo a virtualização completa (HIALINX, 2019).

A primeira versão do Virtual PC para Windows foi lançada em 2001; em 2003 a Microsoft adquiriu da Connectix o Virtual PC juntamente ao Virtual Server, que até então não havia sido lançado. O Virtual PC obteve sua última atualização em 2011 e se mantinha compatível para versões do Windows 7 e anteriores, já o Virtual Server foi lançado em 2004 e obteve sua última atualização em 2007, sendo compatível com Windows XP, Windows Vista e Windows Server 2003; ambos foram sucedidos pelo Hyper-

V em 2008, iniciando sua compatibilidade com Windows 8 e a mantendo com as versões mais atuais do Windows (WIKIPEDIA, 2022).

Além do Hyper-V hoje se encontram outras opções para virtualização, como a Virtual Box que foi desenvolvida pela (INNOTEK, 2007), comprada pela (SUN MICROSYSTEMS. 2006) em 2008 e adquirida em 2010 pela Oracle e, a VMware Workstation e VMware vSphere lançadas pela (VMWARE, 2021) em 1999 e 2009 respectivamente, e continuam com suporte até os dias atuais (LAWRENCE, 2015).

De acordo com Veras (2019, p. 99), “a virtualização pode ser conceituada de duas principais formas:

- É o particionamento de um servidor físico em vários servidores lógicos.
- É uma camada de abstração entre o hardware e o software que protege o acesso direto do software aos recursos físicos do hardware”.

Dessa forma a virtualização permite que várias máquinas virtuais (VMs), cada uma possuindo seu próprio sistema operacional, sejam executadas em um único computador, que serve como hospedeiro das mesmas (TANENBAUM; BOS, 2016). Para que ocorra a virtualização comumente utiliza-se um Monitor de Máquina Virtual (VMM), que também é conhecido como hipervisor para controlar o hardware do hospedeiro em função das VMs em execução (REMZI, ARPACI-DUSSEAU, 2018).

2.1.1 Máquina virtual (VM)

De acordo com Zanoello (2017, p. 9), a máquina virtual é um contêiner de software, que possui seu próprio sistema operacional e aplicativos além de ser autônoma. A utilização de várias VMs em um único servidor físico viabiliza o uso de diversos sistemas operacionais ao mesmo tempo sem necessidade de um novo computador para executá-los.

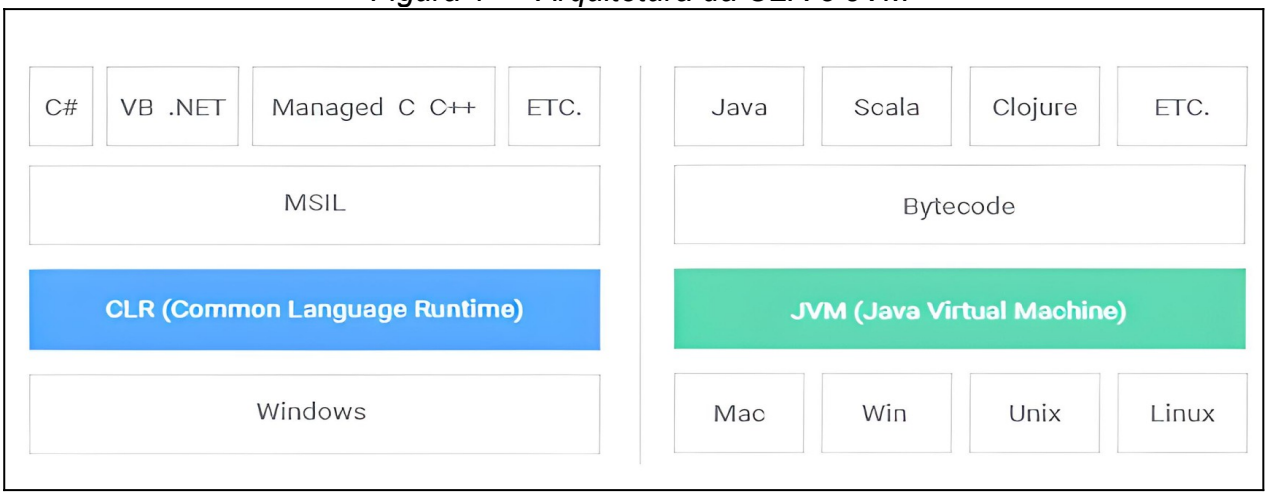
Para (SINGH, 2019) as máquinas virtuais são criadas para executar tarefas específicas que podem ser prejudiciais ao computador hospedeiro. Como a máquina virtual é separada do sistema do host caso algum erro ou danificação ocorra na máquina virtual o mesmo não ocorrerá com o computador físico.

Além das VMs de sistema, que providenciam um sistema operacional completo, também se utilizam as VMs de processo, que são designadas para executar um processo

ou programa independente do ambiente (SMITH; NAIR, 2005). Como exemplos de VMs de processo temos Oracle Java Virtual Machine (JVM) e Microsoft Common Language Runtime (CLR), apresentadas na Figura 1.

A JVM é um produto da Oracle e a CLR um ambiente em tempo de execução do .NET, um framework da Microsoft, ambas apresentam diversas similaridades, elas empregam operações baseadas em pilha, sistema de coleta de lixo, incluem segurança em nível de tempo de execução e, métodos para tratamento de exceções, porém também apresentam diferenças: a CLR possui instruções para corrotina e declaração ou manipulação de ponteiros mas a JVM não, a JVM é compatível com diversos sistemas operacionais enquanto a CLR, originalmente apenas compatível com Windows, atualmente possui algumas versões compatíveis com Linux (SOROKER, 2018).

Figura 1 — Arquitetura da CLR e JVM



Fonte: (*SOROKER, 2018*)

2.1.2 Hipervisor

O hipervisor é responsável por abstrair os recursos da máquina física e permitir a hospedagem de máquinas virtuais. Os recursos da máquina hospedeira podem ser compartilhados entre as VMs; para isso, o hipervisor controla como os recursos são acessados e aloca dinamicamente recursos de computação para cada VM conforme necessário (SINGH, 2019).

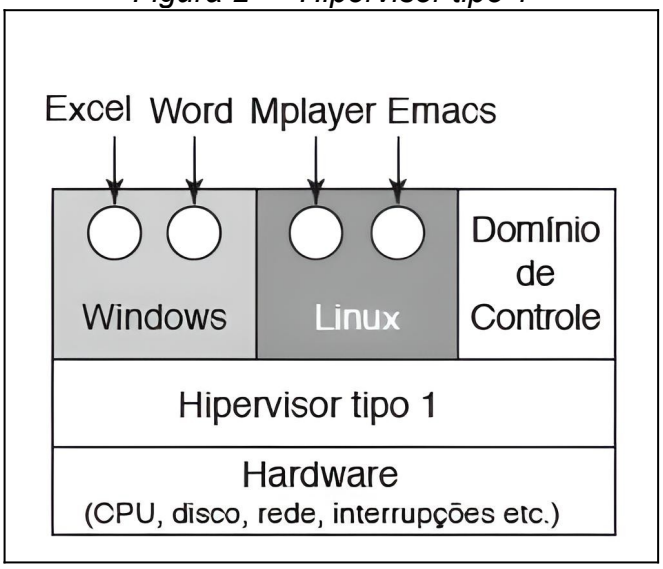
De acordo com Goldberg (1973, p22), existem dois tipos de hipervisores:

1. Executa diretamente na máquina hospedeira, de modo privilegiado. Ex: Oracle VM.

2. Necessita de um host, como um sistema operacional para ser executado. Ex: VMware Workstation.

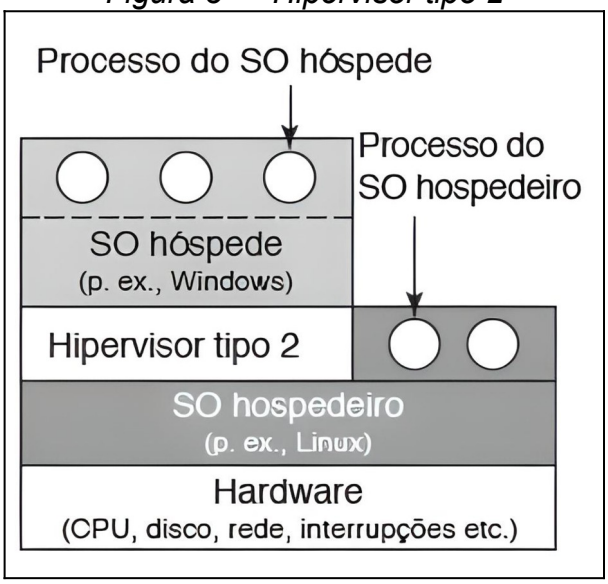
Na Figura 2 observa-se a ilustração de um hipervisor tipo 1, ele executa diretamente na máquina física, dando suporte as máquinas virtuais instaladas, multiplexando o hardware e proporcionando o necessário para a execução de seus aplicativos. Ex: Microsoft Hiper-V, VMware vSphere, Oracle VM.

Figura 2 — Hipervisor tipo 1



Fonte: (*TANENBAUM; BOS, 2016*)

Figura 3 — Hipervisor tipo 2



Fonte: (*TANENBAUM; BOS, 2016*)

Na Figura 3 encontra-se uma ilustração de um hipervisor de tipo 2, ele executa sobre um sistema operacional, sendo limitado pelo mesmo, já que o sistema operacional disponibiliza apenas uma parte do hardware para que o hipervisor controle e redistribua entre as máquinas virtuais instaladas. Ex: VMware Workstation, Oracle VM Virtual Box.

Independente do tipo, o hipervisor, cria a ilusão de que o sistema operacional de cada máquina virtual controla o hospedeiro, porém o hipervisor na verdade está controlando o hardware de modo a multiplexar os recursos físicos do computador entre todas as VMs (REMZI; ARPACI-DUSSEAU, 2018). A utilização de um hipervisor, impede que um erro em uma máquina virtual atrapalhe o desempenho das demais (TANENBAUM; BOS, 2016).

2.2 Controle de fluxo

O fluxo de execução está relacionado à ordem em que as instruções de um programa devem ser executadas (TANENBAUM; AUSTIN, 2013). Chamadas de procedimento ou exceções causam alterações no fluxo de execução, ocasionando a interrupção do procedimento em execução que deverá ser retornado após o encerramento da chamada de procedimento.

A maioria das instruções não causam desvios no fluxo de execução de um programa: ao se encerrar a execução de alguma instrução a seguinte é buscada na memória, seguindo a ordem do programa; quando uma instrução causa um desvio, a ordem das instruções presentes no programa não é mais válida, sendo a instrução de desvio responsável por informar a seguinte instrução a ser executada (TANENBAUM; AUSTIN, 2013).

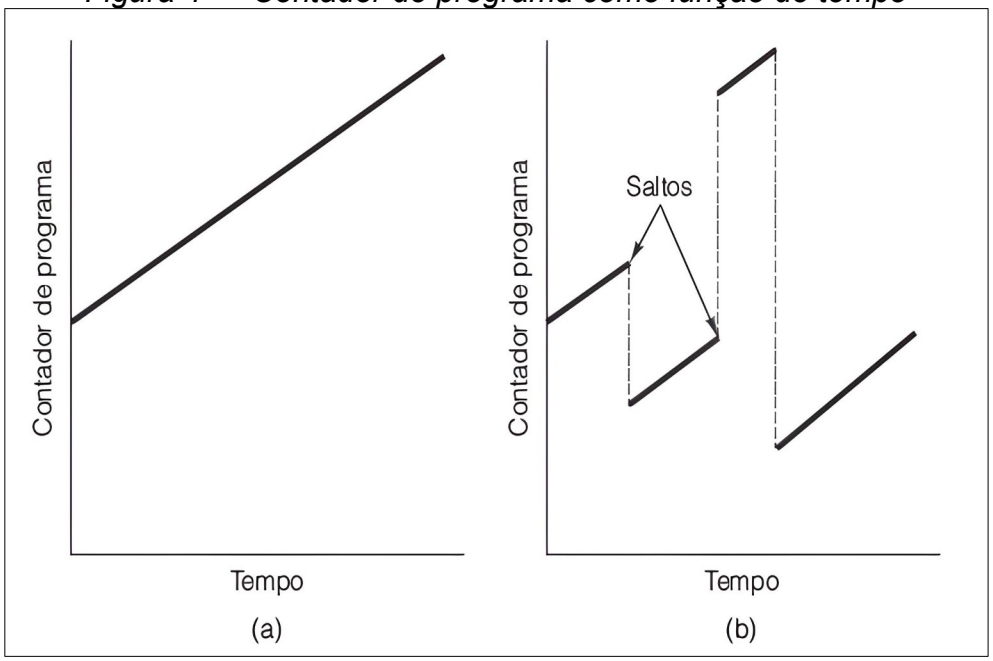
A execução de um programa um computador opera em ciclos de instrução, com uma instrução de máquina por ciclo (STALLINGS, 2010). Cada ciclo possui três estágios: **fetch**, **decode** e **execute**, em CPUs antigas, os ciclos são executados sequencialmente, mas em CPUs modernas (Ex: RISC) é possível a execução de ciclos concorrentes ou em paralelo. Para cada ciclo são utilizados 2 registradores específicos da máquina:

- IR – Registrador de instruções;
- PC – Contador de programa.

Durante o estágio de **fetch**, a próxima instrução é lida da memória e salva no IR, e o PC é incrementado, de forma que aponte para a instrução seguinte. No estágio de **decode** o código da instrução é interpretado e no estágio **execute** a unidade de controle da CPU passa a informação decodificada como uma sequência de sinais para a unidade relevante (lógica ou aritmética) referente a instrução sendo executada. O ciclo até o fim do programa (SEBESTA, 2012).

Na Figura 4-a encontra-se o contador de programa como função de tempo em um programa sem desvios, e com desvios na Figura 4-b, na qual o programa pode receber instruções que estão alocadas em posições posteriores ou anteriores a qual se encontra.

Figura 4 — Contador de programa como função de tempo



Fonte: (*TANENBAUM;* AUSTIN*, 2013*)

2.2.1 Subprogramas

Sebesta (2012, p. 395) categoriza os subprogramas em funções e procedimentos, sendo funções semanticamente modeladas como funções matemáticas, e chamadas ao utilizarem seus nomes e parâmetros no lado direito de instruções de atribuição ou expressões aritméticas, enquanto os procedimentos são chamados apenas invocando o nome do procedimento como uma instrução no programa chamador.

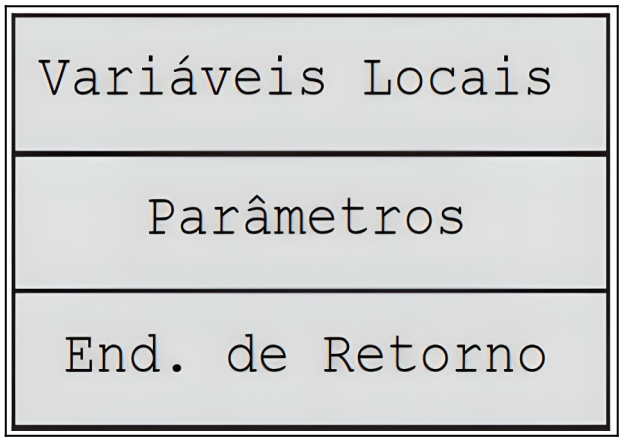
Um subprograma consiste em duas partes distintas, o código do subprograma, que é constante, e as variáveis locais, que mudam ocasionalmente quando o subprograma é executado. As partes do subprograma que não pertencem ao código são chamadas de registro de ativação (SEBESTA, 2012).

Um registro de ativação (ARI) são posições contíguas na pilha de execução que contém, em um *layout* simples, as variáveis locais, os parâmetros e o endereço de retorno de um subprograma, como ilustrado na Figura 6.

As linguagens que utilizam pilhas com alocação dinâmica necessitam armazenar no ARI um ponteiro, **dynamic link**, que referencia a base do registro de ativação do subprograma chamador. Seus registros de ativação assemelham-se à Figura 5.

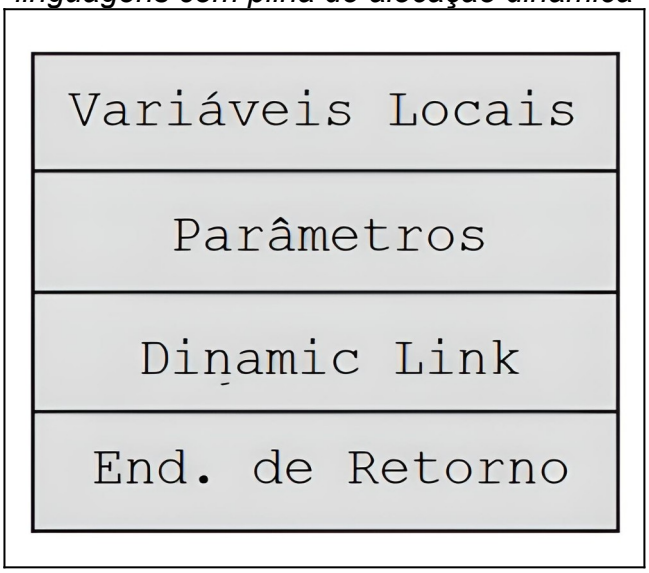
Segundo (TANENBAUM; AUSTIN, 2013) “a técnica mais importante para estruturar programas é o procedimento”. A chamada de um procedimento altera o controle de fluxo porém ao seu termino devolve o controle a instrução seguinte a chamada.

Figura 6 — Registro de Ativação Simples



Fonte: Baseado em (SEBESTA, 2012)

Figura 5 — Registro de ativação para linguagens com pilha de alocação dinâmica



Fonte: Baseado em (SEBESTA, 2012)

Para (SEBESTA, 2012) a semântica para chamada de um subprograma necessita de algumas ações do subprograma chamador:

1. Criar uma instância de chamada de registro de ativação (ARI);
2. Salvar o status de execução da unidade de programa atual no seu ARI;
3. Calcular e passar os parâmetros;
4. Passar o endereço de retorno para o subprograma chamado;
5. Transferir o controle para o ARI do subprograma chamado.

Enquanto as ações de prólogo (antes de executar) do subprograma chamado são:

1. Salvar o EP (ponteiro de ambiente, utilizado para acessar parâmetros e variáveis locais durante a execução do subprograma) na pilha como **dynamic link** e criar seu novo valor;
2. Alocar as variáveis locais.

E as ações de epílogo do subprograma chamado (antes do retorno) são:

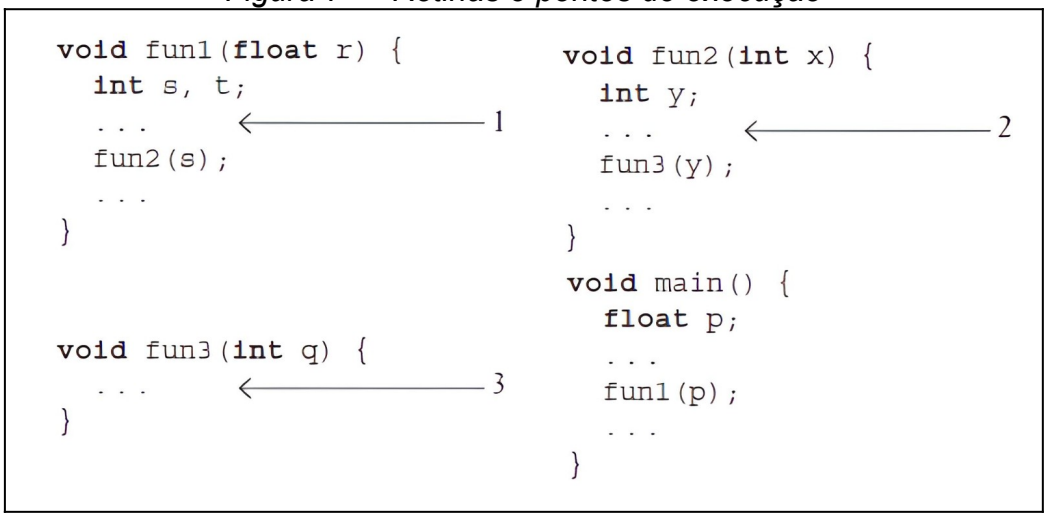
1. Caso tenha parâmetros passados por resultado ou modo de saída, tais valores são movidos para os parâmetros correspondentes;
2. Se o subprograma é uma função, seu resultado é movido para um local acessível ao subprograma chamador;
3. Restaurar o ponteiro de pilha pelo valor atual do EP menos um e atualizar o EP com o valor do **dynamic link** anterior;
4. Restaurar o status de execução para o chamador;
5. Transferir o controle de volta para o chamador.

Para gerenciar subprogramas recursivos é necessário o armazenamento de seus parâmetros e variáveis locais em uma pilha, de forma que para cada chamada, uma nova instância do ARI na pilha é alocada e, ao término da mesma, essa instância do ARI é liberada e a chamada anterior mais recente é executada (TANENBAUM; AUSTIN, 2013).

2.2.2 Pilha de execução

Em cada chamada de subprograma um registro de ativação referente a mesma é alocada na memória do computador na pilha de execução, e quando o subprograma se encerra, seu registro de ativação é retirado da pilha e o controle de fluxo retorna para o subprograma anterior. Como exemplo, considere o programa descrito na Figura 7.

Figura 7 — Rotinas e pontos de execução



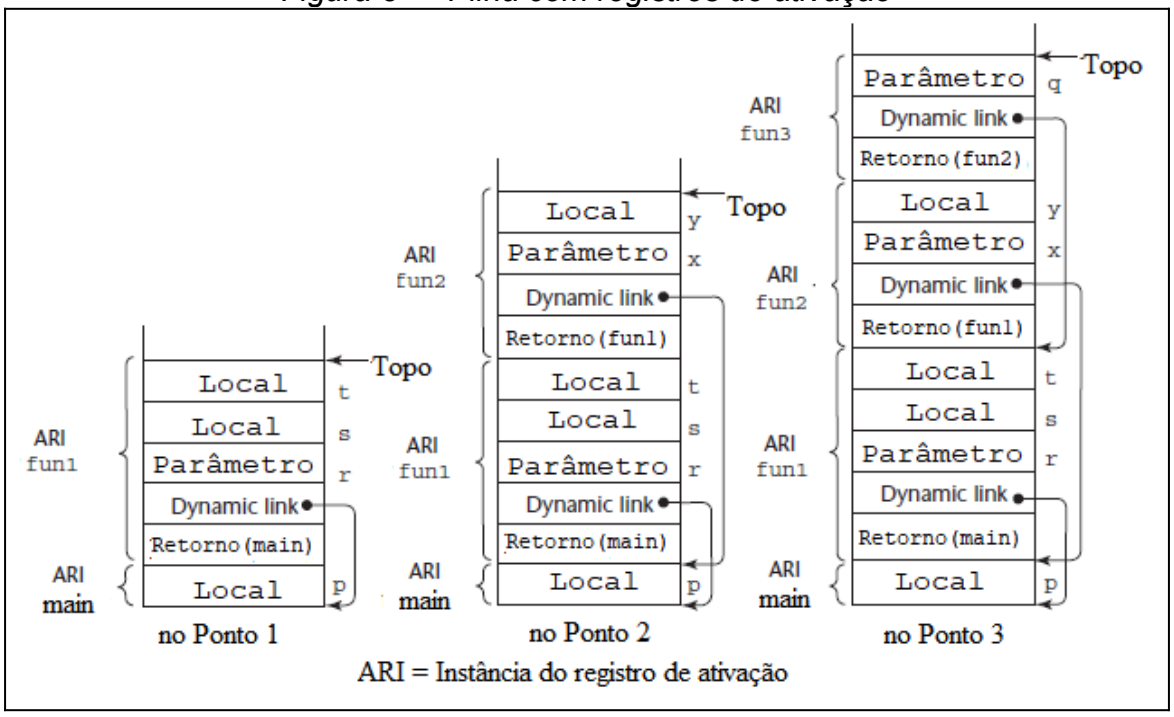
Fonte: (*SEBESTA, 2012*)

Na Figura 8 encontra-se a pilha contendo os registros de ativação para cada ponto de execução das rotinas na Figura 7, sendo a sequência de chamada das funções:

1. main chama fun1;

2. fun1 chama fun2;
3. fun2 chama fun3.

Figura 8 — Pilha com registros de ativação



Fonte: Baseado em (*SEBESTA, 2012*)

2.3 Trabalhos Relacionados

Alpert (1979) teve como finalidade desenvolver um interpretador para P-code que consiga compilar Pascal, para utilização na máquina Stanford EMMY (NEUHAUSER, 1977), que é configurada como parte do Laboratório de Emulação de Stanford, para isso a P-code foi montada em uma representação binária e interpretada por um microprograma no armazenamento de controle do EMMY.

Weber (2016) propôs o desenvolvimento de uma representação intermediária para o Portal de Algoritmos da UCS (Universidade de Caxias do Sul), para tradução de comandos, estrutura de dados e controle da linguagem utilizada. O Portal de Algoritmos permite a edição e interpretação de código em Português Estruturado, Weber descreve que, com a nova representação, o portal pode ser expandido para outras linguagens de programação.

Rubinius (2016) é uma a máquina virtual que suporta várias linguagens de programação e, iniciou-se como uma implementação alternativa de Ruby criada por Evan Phoenix. Rubinius executa em macOS e vários sistemas de operação Linux/Unix, porém

não é suportado em Windows. Um aspecto do Rubinius é que seu conjunto de instruções deve representar qualquer semântica de máquina, contrastando com sua implementação inicial que dependia de primitivos C++ modelados na máquina virtual Smaltalk 80.

Para obter um interpretador de Ruby que não seja lento, Sasada (2005) desenvolveu o YARV (Yet Another Ruby VM), uma máquina virtual baseada em uma arquitetura de máquina de pilha com recursos para otimização para execução de programas em Ruby em alta velocidade. YARV é escrita em C, e possui um conjunto de instruções designado especificamente para Ruby.

Parrot (2021) é uma máquina virtual de processo baseada em registradores, designada para compilar e executar bytecode para linguagens dinâmicas; sua implementação foi iniciada pela comunidade de Perl, lançada em 2009 como software livre e de código aberto, como Parrot não obteve atualizações desde 2016 foi oficialmente descontinuada em 2021.

O papel de Parrot como VM para Perl 6, renomeado em 2019 como Raku, foi preenchido por MoarVM (2022), uma máquina virtual utilizada pela maioria dos programadores de Raku, possuindo suporte para threads e, recursos de linguagem como exceções e carregamento de código em tempo de execução.

2.4 Estado da Arte

A Oracle possui sua própria máquina virtual, a Oracle VM VirtualBox, um software de virtualização de plataforma cruzada e código aberto. A VirtualBox foi desenvolvida para reduzir o custo operacional e diminuir o tempo de execução, recomendada para teste, desenvolvimento, demonstração e implantação de soluções através de múltiplas plataformas em um único dispositivo, é suportada em Windows, Linux, macOS e Oracle Solaris (ORACLE, 2020).

Além da Oracle VM VirtualBox a Oracle também possui a Oracle VM, uma plataforma que fornece um ambiente equipado para virtualização; a Oracle VM permite a implantação de sistemas operacionais e softwares em um ambiente virtualizado. A principal diferença entre a Oracle VM Virtual Box e a Oracle VM é que a Oracle VM não necessita de um sistema operacional para ser executado, sendo o mesmo um hipervisor tipo 1, enquanto que a Oracle VM Virtual Box necessita, sendo um hipervisor tipo 2 (ORACLE, 2021).

Em 1999, a VMware desenvolveu a VMware Workstation Pro, um hipervisor que executa em sistemas operacionais Windows e Linux com processadores AMD ou Intel de 32 ou 64 bits, e se mantém no mercado até os dias atuais, sendo sua versão mais recente a Vmware Workstation Pro 16.0 (VMWARE, 2021).

O Hyper-V é uma tecnologia de virtualização de hardware desenvolvida pela Microsoft e, a partir do Windows 8 substituiu o Windows Virtual PC. O Hyper-V permite a criação de discos rígidos virtuais, computadores virtuais e uma série de dispositivos virtuais que podem ser adicionados a máquinas virtuais (MICROSOFT, 2022b).

Xen é um hipervisor de código aberto que permite a execução de vários sistemas operacionais simultaneamente no mesmo hardware e atualmente se encontra na versão 4.16 que introduziu recursos permitindo o aperfeiçoamento da performance, segurança, funcionalidade e suporte de hardware (MONNE, 2021).

A (NUTANIX, 2022) apresenta como um de seus produtos o Nutanix AHV, que oferece uma solução de virtualização baseado em virtualização Linux de código aberto comprovada, com simplicidade de gerenciamento e baixo custo operacional.

Openstack é um conjunto de componentes de software que oferece serviços para infraestrutura de nuvem para máquinas virtuais, Bare Metal e contêiners; ele gerencia através de APIs ou um painel, pools de recursos de computação, armazenamento e rede (OPENSTACK, 2022).

Assim como o Openstack, Microsoft Azure é uma plataforma que oferece serviços para infraestrutura de nuvem, permitindo a criação de máquinas virtuais, aplicativos com utilização de IA, sites escalonáveis, banco de dados como MySQL e PostgreSQL, dentre outros (MICROSOFT, 2022a).

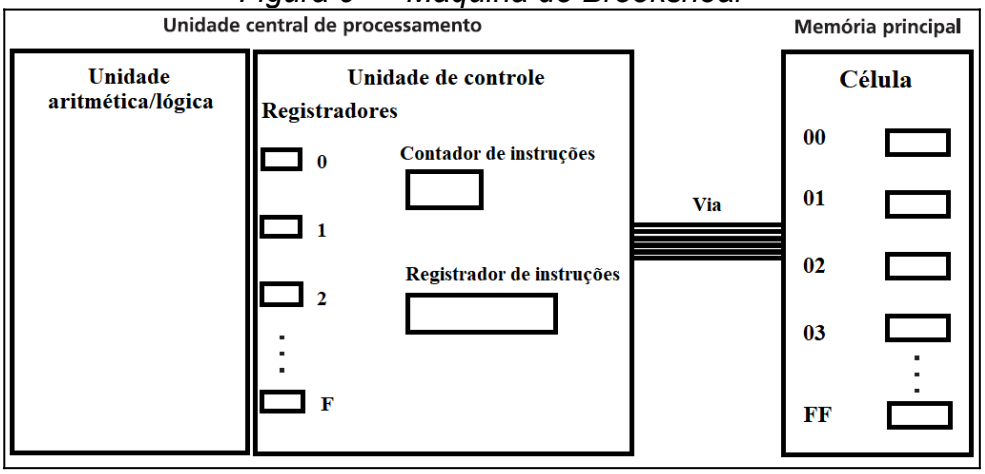
3 DESENVOLVIMENTO

Para alcançar o objetivo de apresentar um modelo semântico de controle de fluxo e chamada e retorno de sub-rotina para hardware virtual, foi desenvolvida para este trabalho a AllanaVM, os passos utilizados para sua implementação são apresentados nas seções 3.1 à 3.4 . Todas as implementações descritas nesta seção podem ser encontradas no GitHub (https://github.com/AllanaCampos/VMs) e nos Apêndices D E e F.

3.1 Escolha da arquitetura do hardware virtual

Dentre os simuladores de hardware disponíveis, encontrou-se a máquina de Brookshear e a P-code. Os elementos principais da máquina de Brookshear podem ser visualizados na Figura 9, possuindo arquitetura Von Neumann, 16 registradores, 256 células de memória com 8 bits de comprimento e, um conjunto de instruções formado por 12 instruções, com 16 bits cada, descritas no APÊNDICE B — Conjunto de Instruções da Máquina Descrita por Brookshear.

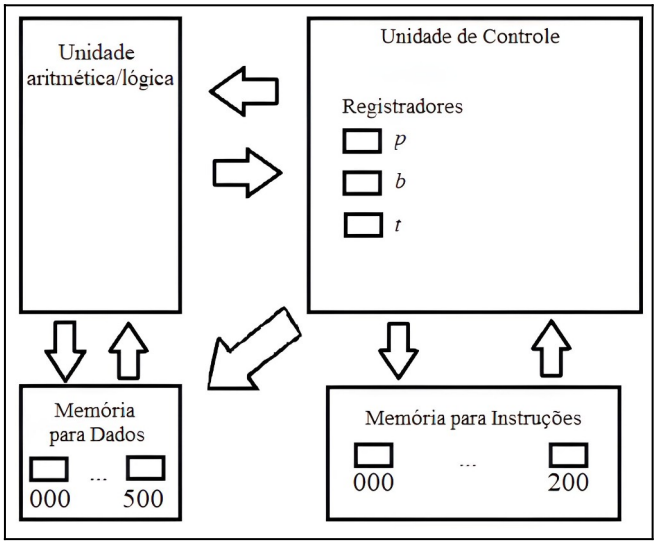
Figura 9 — Máquina de Brookshear



Fonte: (Brookshear, 2005)

Os elementos principais da máquina P-code, podem ser visualizados na Figura 10, possuindo 3 registradores e um conjunto de instruções formado por 8 instruções, descritas no APÊNDICE A — Conjunto de Instruções e Operações da P-code Machine, sendo a instrução OPR utilizada para chamada de 14 operações aritméticas e lógicas e, para permitir a recursão a máquina utiliza uma pilha contendo os registros de ativação, que indica os valores de retorno de chamada além dos dados utilizados em cada sub-rotina.

Figura 10 — P-code Machine



Fonte: Autora

A arquitetura Harvard proporciona segurança à memória de programa enquanto que, a arquitetura Von Neumann permite a alteração do programa em tempo de execução, sendo mais versátil portanto, foi escolhida; como a máquina de Brookshear possui a arquitetura Von Neumann, optou-se por utilizar como elementos principais do hardware virtual desenvolvido os mesmos elementos da máquina de Brookshear e, por conta da variedade de instruções da P-code, a mesma foi utilizada no desenvolvimento do conjunto de instruções da AllanaVM.

3.2 Definição do conjunto de instruções

Para implementar um modelo de chamada e retorno de sub-rotinas precisa-se utilizar registros de ativação contendo os dados da sub-rotina, portanto é necessário instruções que permitam a alocação de espaço para o registro de ativação na pilha de execução assim como instruções de retorno da sub-rotina.

Como o conjunto de instruções da máquina de Brookshear não possui as instruções necessárias para chamadas e retornos de sub-rotinas, optou-se por estender as o conjunto de instruções da máquina de Brookshear com as instruções aritméticas e lógicas que a P-code possui e a Brookshear não apresenta para formar o conjunto de instruções da AllanaVM.

As instruções extendidas da P-code foram implementadas utilizando as instruções da máquina de Brookshear, as mesmas foram armazenadas na memória RAM da

AllanaVM; para isso foi necessário a expansão da memória original da máquina de Brookshear que passou de 2^8 células de 8 bits cada para 2^{32} células de 64 bits, com a expansão houve alteração no comprimento das instruções, passando de 16 bits para 128 bits. Todas as instruções estão indicadas no APÊNDICE C — Conjunto de Instruções da AllanaVM.

3.3 Implementação do modelo de controle de fluxo e chamada e retorno de sub-rotina

O modelo de controle de fluxo e chamada e retorno de sub-rotina da AllanaVM foi implementado com base no modelo apresentado pela P-code, onde, cada ARI apresenta nas duas posições seguintes à base, valores referentes à posição de retorno da sub-rotina e o **dynamic link** indicando o ARI anterior e, nas demais posições as variáveis locais e parâmetros que podem ser alterados no decorrer da execução do programa.

3.3.1 Protocolo de chamada

Para chamar uma sub-rotina, deve-se primeiro utilizar a instrução INT, responsável pelo incremento do apontador ao topo da pilha, reservando o espaço necessário para armazenar o registro de ativação; como INT é uma instrução derivada da P-code, na AllanaVM, o código que implementa INT se encontra em uma posição da memória e, para chamá-la deve-se utilizar uma instrução de JMP, responsável pelos desvios no programa, indicando tal posição. A instrução INT chama uma função que foi desenvolvida com objetivo de salvar o conteúdo da sub-rotina no registro de ativação antes da chamada da mesma. Ao encerrar as instruções retorna-se ao programa principal utilizando outra função de JMP, para isso deve-se salvar o valor da posição de retorno na posição de memória $FC07_{16}$.

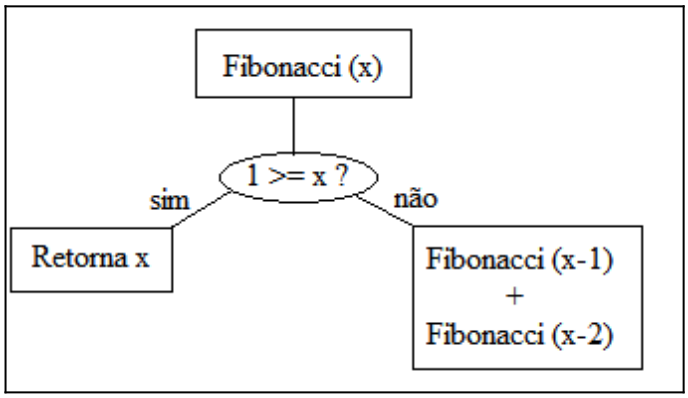
A instrução INT utiliza-se dos registradores F_{16}, E_{16} e FD_{16}, que devem conter respectivamente o primeiro parâmetro da sub-rotina, o segundo parâmetro, se houver e a posição de retorno. Logo, antes de chamar a instrução deve-se inserir tais valores nos registradores indicados, após a chamada da instrução INT pode-se utilizar a instrução de JMP para executar a sub-rotina desejada.

3.3.2 Pilha resultante da AllanaVM

Como instruções de multiplicação, divisão, dentre outras, são derivadas da P-code, as mesmas são executadas através de instruções advindas da máquina descrita por

Brookshear, como soma ou instruções de desvio, de modo que para a AllanaVM tais instruções se tornam uma sub-rotina, logo a pilha de chamadas para uma sub-rotina que envolva as instruções derivadas da P-code são maiores que a pilha de chamadas da P-code, como por exemplo, considerando como entradas números naturais, a sequência de Fibonacci implementada recursivamente segue o diagrama descrito na Figura 11.

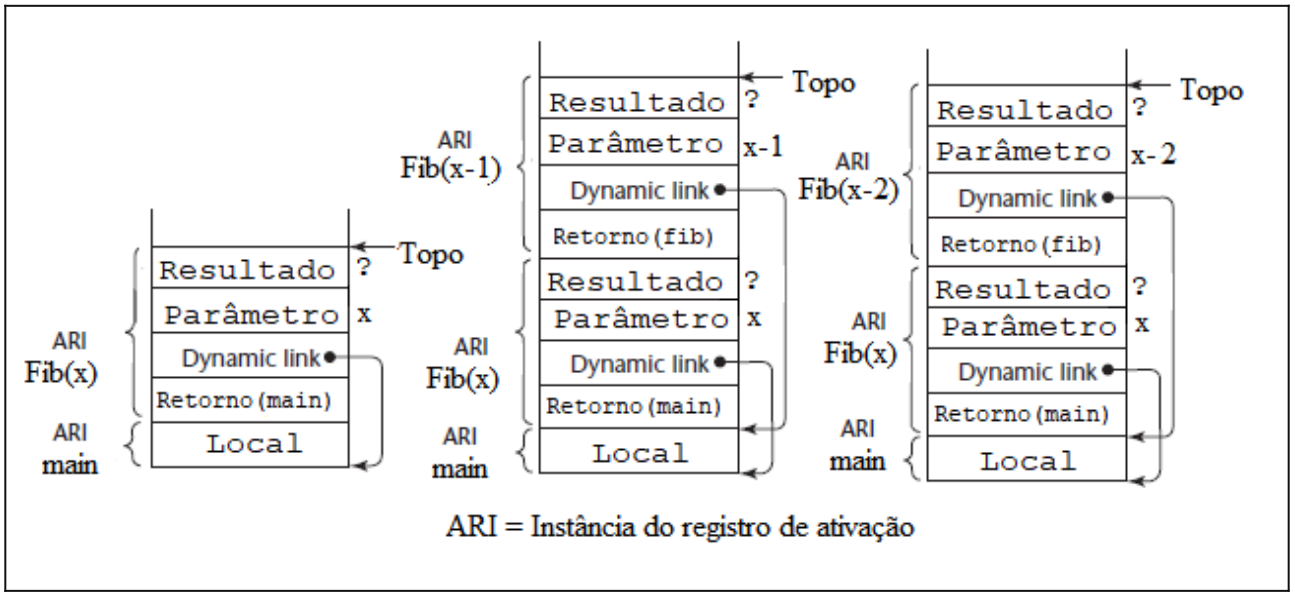
Figura 11 — Diagrama para sequência de Fibonacci



Fonte: Autora

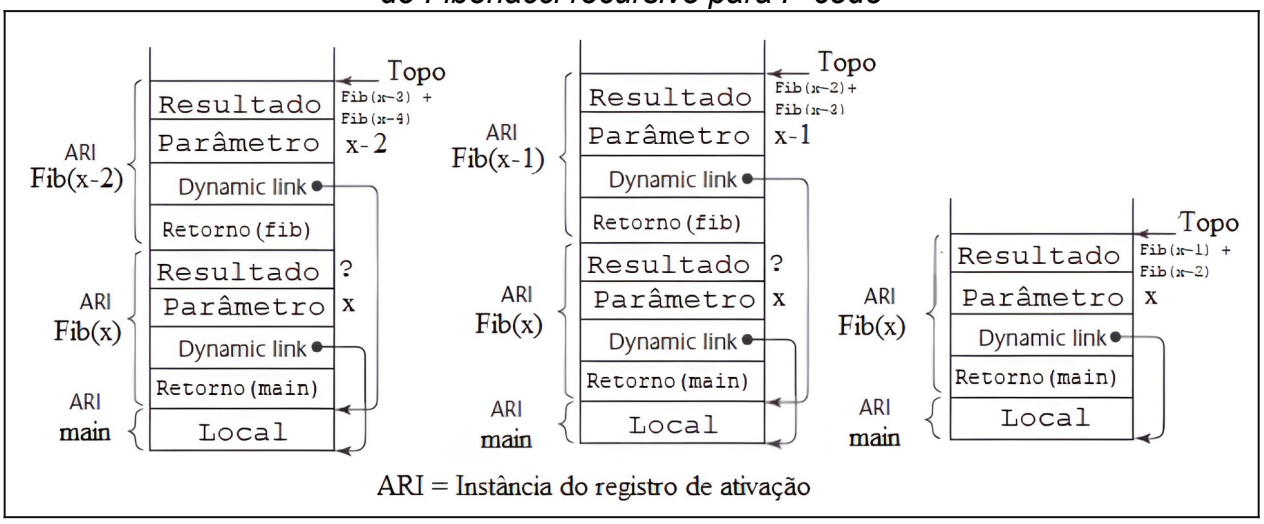
Para verificar se um valor é maior ou igual ao outro utiliza-se a instrução GEQ (≥), advinda da P-code, porém para AllanaVM tal instrução é implementada como uma função de biblioteca, como descrito na seção 3.2 , e utiliza como auxílio das instruções GTR (>), que verifica se um valor é maior que o outro, e EQL (=), que verifica se um valor é igual ao outro, também advindas da P-coode, logo a pilha de execução da mesma função para a P-code, descrita na Figura 12 com seu retorno na Figura 13, é menor que para AllanaVM, descrita na Figura 14 com retorno da Figura 15.

Figura 12 — Pilha de execução para o cálculo da sequência de Fibonacci recursivo para P-code



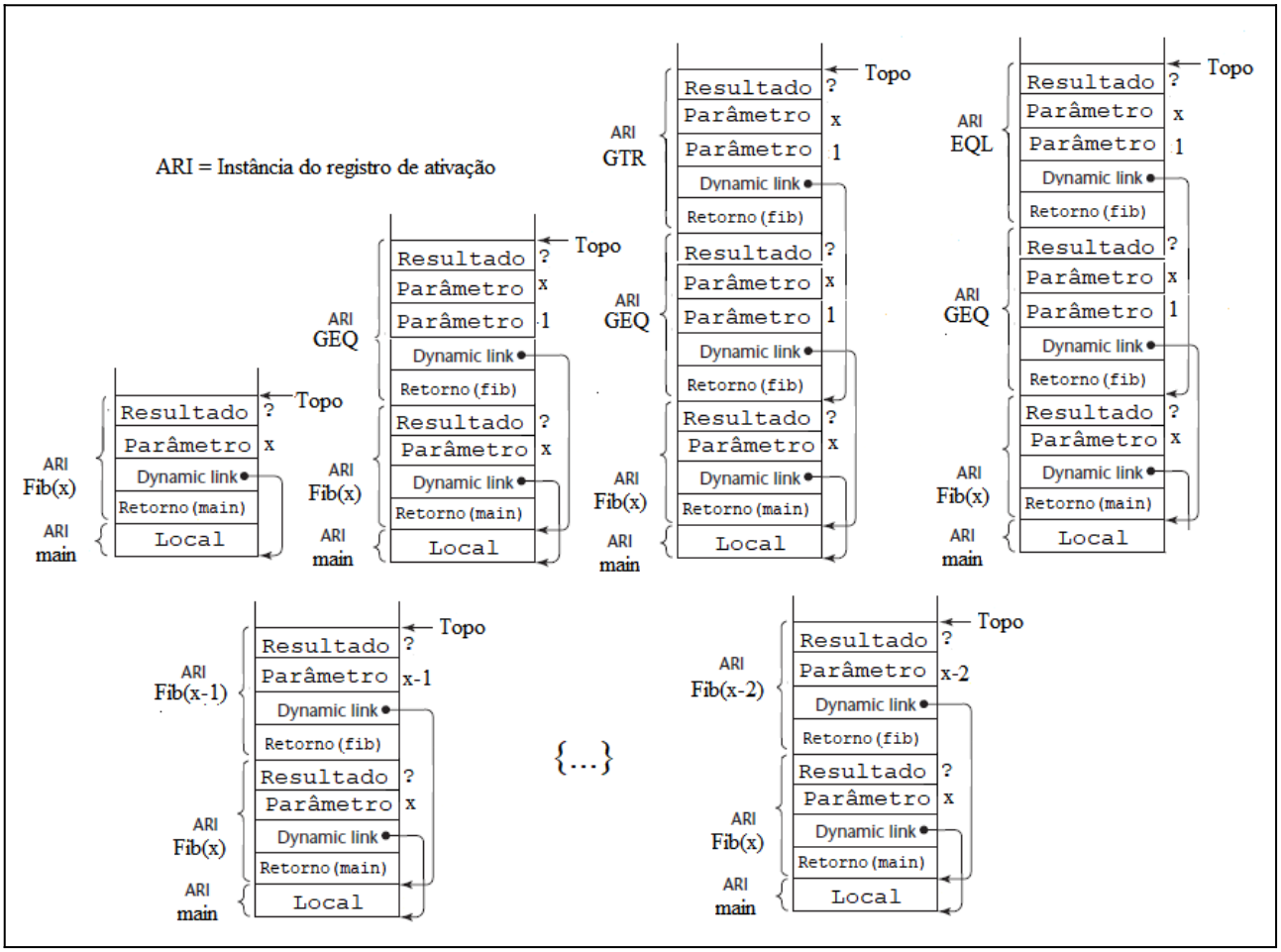
Fonte: Baseado em (*SEBESTA, 2012*)

Figura 13 — Retorno de chamadas na pilha de execução para o cálculo da sequência de Fibonacci recursivo para P-code



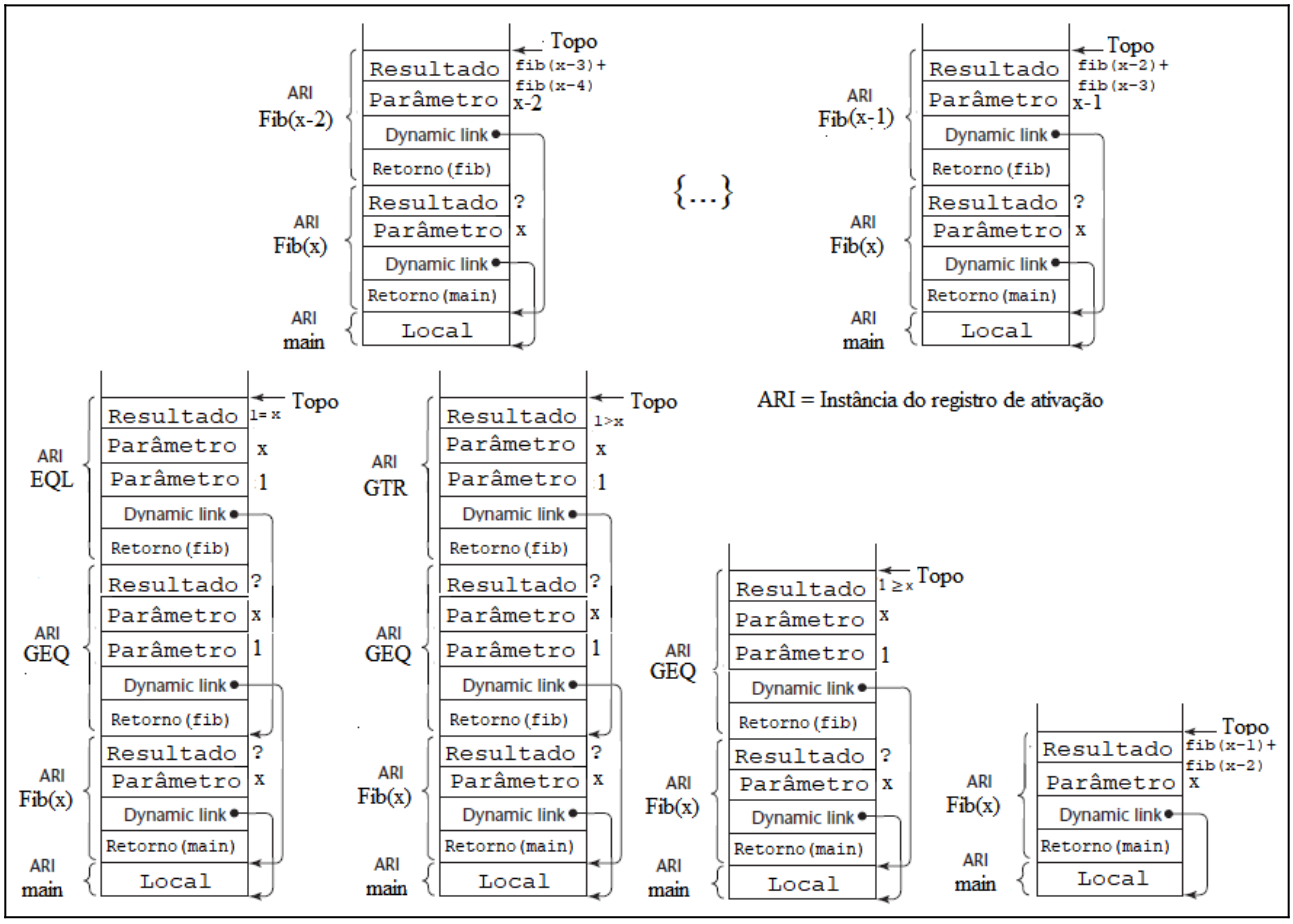
Fonte: Baseado em (SEBESTA, 2012)

Figura 14 — Pilha de execução para o cálculo da sequência de Fibonacci recursivo para AllanaVM



Fonte: Baseado em (SEBESTA, 2012)

Figura 15 — Pilha de execução mostrando os retornos de chamadas para o cálculo da sequência de Fibonacci recursivo para AllanaVM



Fonte: Baseado em (*SEBESTA, 2012*)

Nas Figuras 14 e 15 entre os ARIs presentes na pilha de execução de Fibonacci(x-1) e, Fibonacci(x-2) e Fibonacci(x) existem momentaneamente, ARIs referentes às chamadas das operações GEQ, GTR e EQL, de forma que caso o resultado de tais operações indiquem que 1 é maior ou igual ao valor de entrada o resultado do Fibonacci é a própria entrada, logo a pilha de execução não empilha mais nenhum ARI, caso contrário ocorre a chamada recursiva, então a pilha de execução se torna maior com o ARI referente à função chamada.

3.4 Implementações para testes

O tempo de execução e a quantidade de armazenamento são fatores relevantes ao se escolher qual máquina utilizar, portanto, para comparar tais fatores, além da AllanaVM, implementou-se as máquinas P-code e Brookshear utilizando a linguagem de programação C, as mesmas são apresentadas nas seções 3.4.1 e 3.4.2 .

3.4.1 P-code

A pilha com os registros de ativação da P-code é implementada como um vetor que inicia com o valor -1; os registradores, *p*, *b* e *t*, da máquina são representados por variáveis do tipo inteiro, sendo que *p* é o registrador que contém a posição da seguinte instrução a ser executada, *b* é o registrador que contém a base do registro de ativação e *t* é o registrador que contém o topo do registro de ativação.

O rótulo de cada instrução contida na máquina é armazenada em um ponteiro de caracteres que é utilizado na impressão das instruções em tempo de execução, o mesmo ocorre para as operações aritméticas.

O código executado na P-code é armazenado em um registro de struct, que é formada por três variáveis, *f*, *l* e *a*, onde *f* representa a função, *l* o nível, e *a* o endereço da pilha ou valor de entrada de cada instrução, mas, para as operações aritméticas e lógicas o valor de *a* indica a operação que deve ser executada.

A máquina é formada por 3 funções, *main*, *base* e *getInstructionName* que são descritas a seguir:

a) Main

Na função *main* é declarado o algoritmo a ser executado na P-code e o comportamento da máquina para cada instrução, a função ainda é responsável pela execução do algoritmo e pela impressão das instruções juntamente a pilha com os registros de ativação em tempo de execução; para encerrar um algoritmo o ponteiro indicando a seguinte instrução, *p*, deve possuir valor 0.

b) Base

A função *base* utiliza o valor do nível da instrução para alterar o valor do registrador de base da pilha, e é utilizada na execução das instruções de *store*, *load* e *call*. Por exemplo, sendo a instrução em execução *store 1 3*, a função base indicará o nível da pilha de ativação na qual o valor do topo da pilha será armazenado, se a pilha possua 2 registros de ativação o valor será armazenado no primeiro registro, se a pilha possua 3 registros de ativação o valor será armazenado no segundo registro.

c) GetInstructionName

A função *getInstructionName* recebe como parâmetro o valor referente à instrução a ser executada e retorna o rótulo da instrução ou operação que está sendo executada, esta função é utilizada durante a impressão das instruções.

3.4.2 Brookshear

Como as instruções possuem 16 bits e a memória apenas 8 bits é necessário utilizar duas posições da memória para armazenar uma instrução. Na implementação da máquina de Brookshear o rótulo de cada instrução é armazenado em um ponteiro de caracteres que é utilizado na impressão das instruções em tempo de execução, a memória RAM e os registradores são declarados como vetores, e o código executado na máquina é armazenado em um vetor de struct, que é formada por quatro variáveis, *op*, *opr1*, *opr2* e *opr3*, onde *op* representa a operação e, *opr1*, *opr2* e *opr3* são os operandos.

Em seu total a implementação apresenta 7 funções, *main*, *sum_flut*, *sum_comp2*, *rot*, *trans*, *transformInstruction* e *getInstructionName* que são descritas a seguir:

a) Main

Na função *main* é declarado o algoritmo a ser executado na máquina; durante a execução do algoritmo a função controla o comportamento da máquina para cada instrução e imprime seus rótulos com o conteúdo dos registradores, quando recebe a instrução de HALT a máquina encerra a execução do algoritmo.

b) Sum_flut

A função *sum_flut* é responsável pela soma dos valores pertencentes a dois registradores em notação de ponto flutuante. Como cada registrador possui 8 bits de comprimento, durante a soma em ponto flutuante, o bit mais significativo indica o sinal do valor armazenado, os 3 bits seguintes representam o expoente em excesso de 4 e os 4 bits menos significativos indicam a mantissa.

c) Sum_comp2

A função *sum_comp2* é responsável pela soma dos valores pertencentes a dois registradores em complemento de dois, sendo que o bit mais significativo de cada valor indica o sinal do mesmo.

d) Rot

A função *rot* é responsável por calcular a rotação em sentido horário dos bits do registrador informado na chamada da instrução, a quantidade de vezes que um bit será rotacionado também é informado na chamada da instrução.

e) Trans

A função *trans* é responsável por retornar uma posição da memória indicada pelos valores dos operandos *opr2* e *opr3*, de forma que o operando *opr2* indique os bits mais significativos da posição e o operando *opr3* os bits menos significativos.

f) TransformInstruction

A função *transformInstruction* é responsável por indicar qual é a operação e os operandos de uma instrução.

g) GetInstructionName

A função *getInstructionName* retorna o rótulo da instrução ou operação que está sendo executada, esta função é utilizada durante a impressão das instruções.

4 RESULTADOS

Para o teste do funcionamento de cada máquina foram implementados algoritmos para calculo de fatorial e da sequência de Fibonacci, que são descritos no APÊNDICE G — Implementação C dos programas de teste da AllanaVM, Brookshear e P-code assim como no GitHub (https://github.com/AllanaCampos/VMs). Os testes objetivam verificar o desempenho de cada máquina para algoritmos iterativos e recursivos, logo, para todas as máquinas, desenvolveu-se algoritmos iterativos para ambos casos de teste e para as máquinas que suportam recursividade, P-code e AllanaVM, também implementou-se os mesmos algoritmos em forma recursiva.

Para manter os testes em um ambiente comum entre as máquinas, utilizou-se um computador Intel® Core™ i7 7500U CPU @ 2.70GHz, 2.90GHz, 8.00GB, 64 bits, Windows 10 para executar os algoritmos, de modo que para os mesmos testes, resultados diferentes aos apresentados podem ser obtidos ao utilizar computadores com configurações diferentes.

As figuras apresentadas nas seções 4.1 e 4.2 indicam os resultados para a quantidade de ciclos de clock para cada entrada, onde *n* representa as entradas para cada algoritmo, ou seja, para os algoritmos de fatorial, calcula-se o fatorial de *n* e, para os algoritmos de Fibonacci, calcula-se o número da sequência de Fibonacci na posição *n*.

Cada algoritmo foi executado três vezes para cada máquina. As tabelas apresentadas nas seções 4.1 e 4.2 indicam a média da quantidade de ciclos de clock, das execuções realizadas, que foram necessárias para o cálculo da entrada limitante.

Na Tabela 1, são indicados os valores máximos de armazenamento para AllanaVM, P-code e Brookshear de acordo com o tamanho da palavra de cada máquina. Na Tabela 2, os limites para parâmetros de entrada para as mesmas em cada algoritmo, com o resultado referente ao cálculo de tais entradas, além do resultado para a seguinte entrada, indicando overflow.

Tabela 1 — Tamanho da Palavra

AllanaVM	Brookshear	P-code
64 bits	8 bits	32 bits
18.446.744.073.709.551.615 (8 bytes)	255 (1 byte)	4.294.967.295(4 bytes)

Fonte: Autora

Tabela 2 — Limites para Parâmetros de Entrada

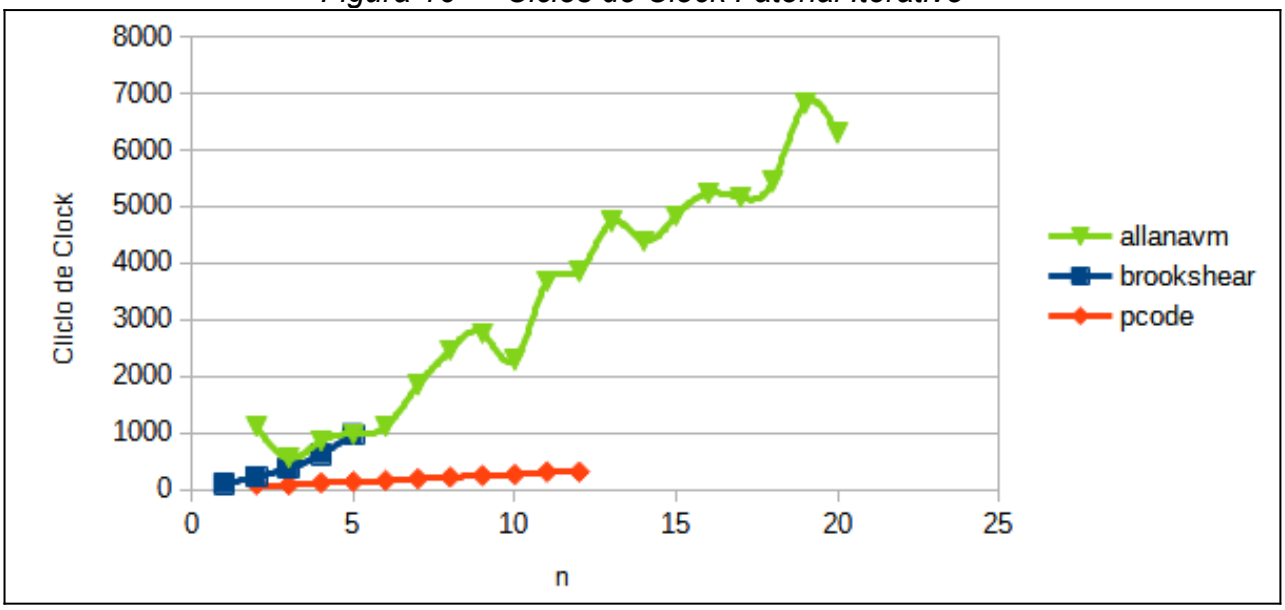
Algoritmos		AllanaVM	Brookshear	P-code
Fatorial	Limite	20! = 2.432.902.008.176.640.000	5! = 120	12! = 479.001.600
	Overflow	21! = 51.090.942.171.709.000.000	6! = 720	13! = 6.227.020.800
Fibonacci	Limite	F(93) = 12.200.160.415.121.876.738	F(13) = 233	F(47) = 2.479.001.600
	Overflow	F(94) = 19.740.274.219.868.223.167	F(14) = 377	F(48) = 4.807.526.976

Fonte: Autora

4.1 Implementações Iterativas

Como entrada para o fatorial iterativo as máquinas AllanaVM e P-code iniciaram calculando o fatorial de 2 e, a cada iteração acresceu 1 até atingirem o calculo do fatorial de 20 e de 12 respectivamente e, a máquina de Brookshear iniciou calculando o fatorial de 1 e, a cada iteração acresceu 1 até atingir o cálculo do fatorial de 5. A quantidade de ciclos de clock para execução de cada entrada pode ser observado na Figura 16 e, os limites para parâmetros de entrada de cada máquina estão descritos na Tabela 3.

Figura 16 — Ciclos de Clock Fatorial Iterativo



Fonte: Autora

Tabela 3 — Limites Fatorial Iterativo

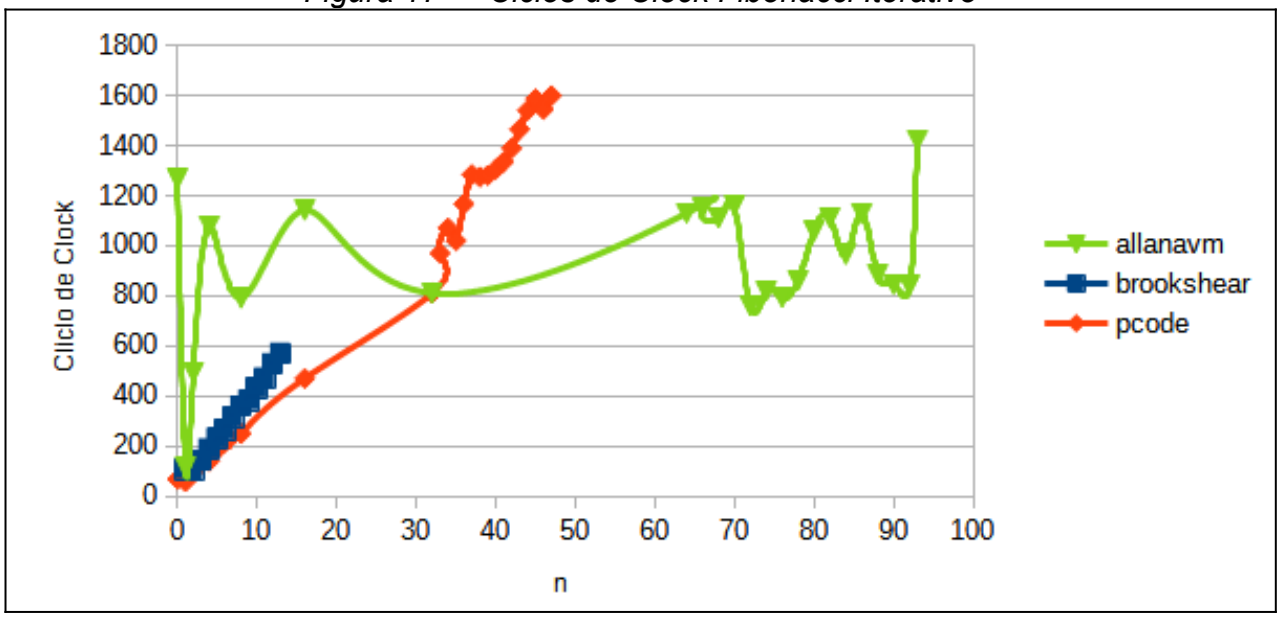
Máquina	n	Ciclos de Clock
AllanaVM	20	6295
Brookshear	5	980
P-code	12	314

Fonte: Autora

Para o Fibonacci iterativo, para a AllanaVM calculou-se os valores da sequência para potências de 2 até obter o resultado correspondente a 64 passando a múltiplos de 2 até atingir o cálculo da sequência de Fibonacci na posição 92 encerrando-se com o cálculo da posição 93; para a máquina de Brookshear calculou-se o valor da sequência de 1 à 13; a P-code calculou-se os valores para 0, 1 e 2, e depois continuou-se com o calculo para as potências de 2 até atingir o valor 32 passando a incrementos de 1 até calcular-se a sequência na posição 46 e encerrando o cálculo da sequência na posição 47.

A quantidade de ciclos de clock para execução de cada entrada pode ser observado na Figura 17 e, os resultados referentes aos limites para cada máquina são apresentados na Tabela 4.

Figura 17 — Ciclos de Clock Fibonacci Iterativo



Fonte: Autora

Tabela 4 — Limites Fibonacci Iterativo

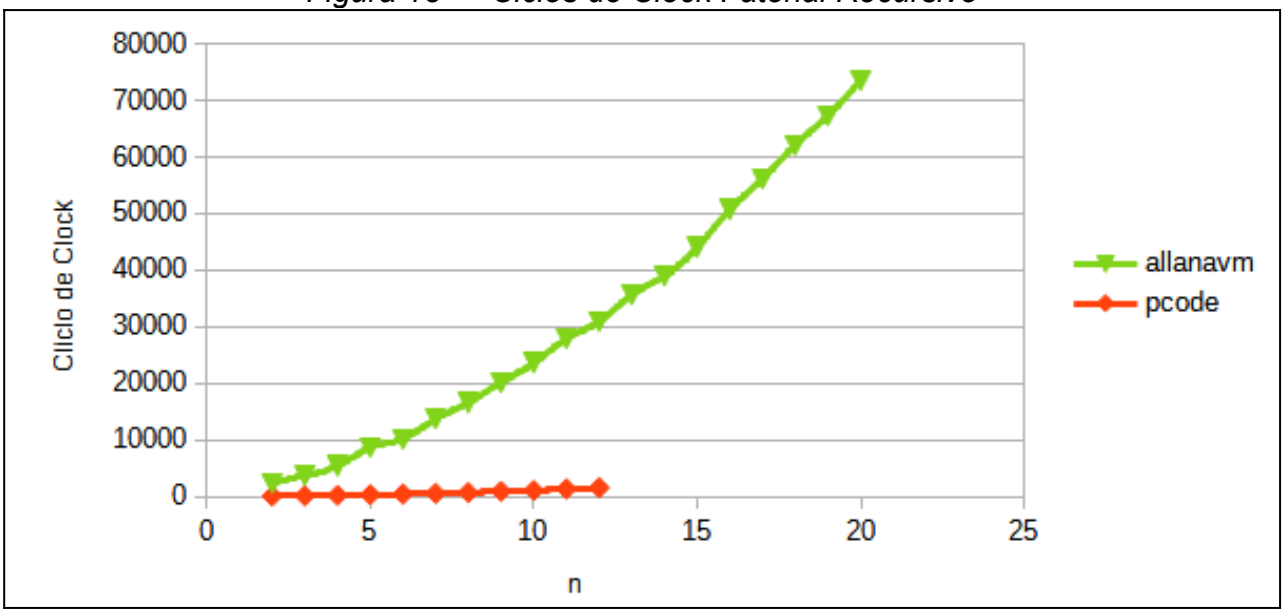
Máquina	n	Ciclos de Clock
AllanaVM	93	1423
Brookshear	13	570
P-code	47	1599

Fonte: Autora

4.2 Implementações recursivas

As execuções do fatorial recursivo, para a AllanaVM calculou-se de 1 à 20 e para a P-code calculou-se de 1 à 12. Os resultados para a quantidade de ciclos de clock para execução de cada máquina pode ser observado na Figura 18 e seus limites são descritos na Tabela 5.

Figura 18 — Ciclos de Clock Fatorial Recursivo



Fonte: Autora

Tabela 5 — Limites Fatorial Recursivo

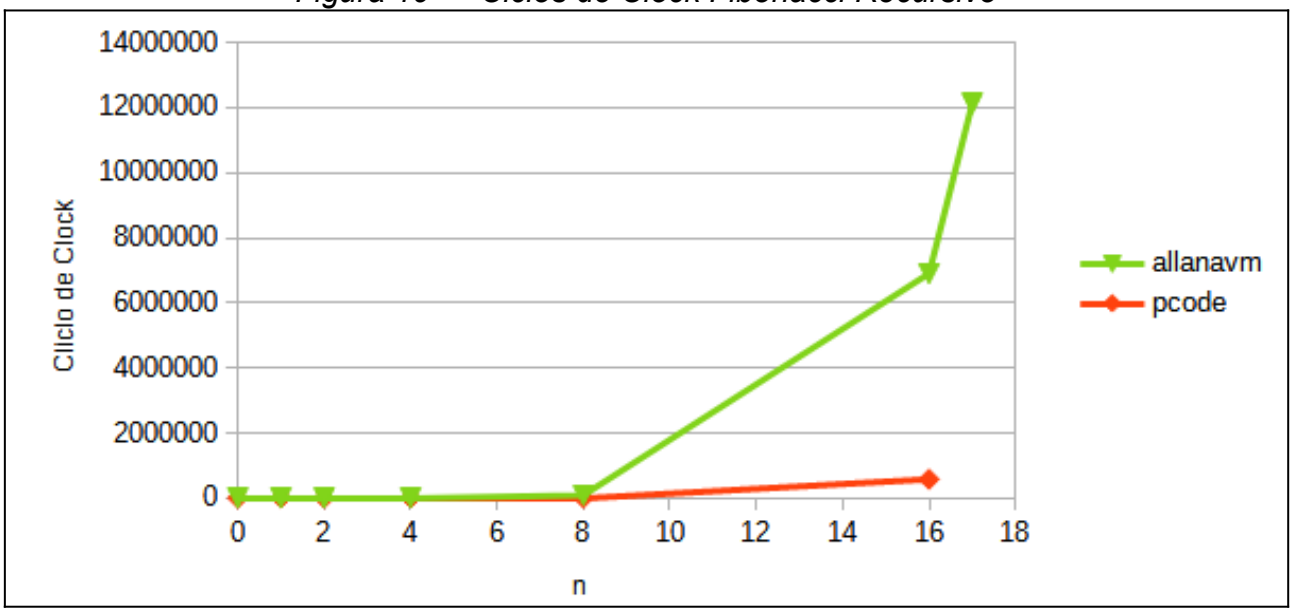
Máquina	n	Ciclos de Clock
AllanaVM	20	73532
P-code	12	1586

Fonte: Autora

Para o Fibonacci recursivo, a AllanaVM calculou os valores para 0, 1 e 2, e continuou-se o cálculo para as potências de 2 até 16 onde encerrou calculando os valores para sequência de Fibonacci na posição 17. A P-code calculou os valores para 0, 1 e 2, onde continuou-se com os múltiplos de 2 até 16.

A quantidade de ciclos de clock para execução do Fibonacci recursivo referente a cada entrada para cada máquina pode ser observado na Figura 19 e seus respectivos limites são apresentados na Tabela 6.

Figura 19 — Ciclos de Clock Fibonacci Recursivo



Fonte: Autora

Tabela 6 — Limites Fibonacci Recursivo

Máquina	n	Ciclos de Clock
AllanaVM	17	12113182
P-code	16	573018

Fonte: Autora

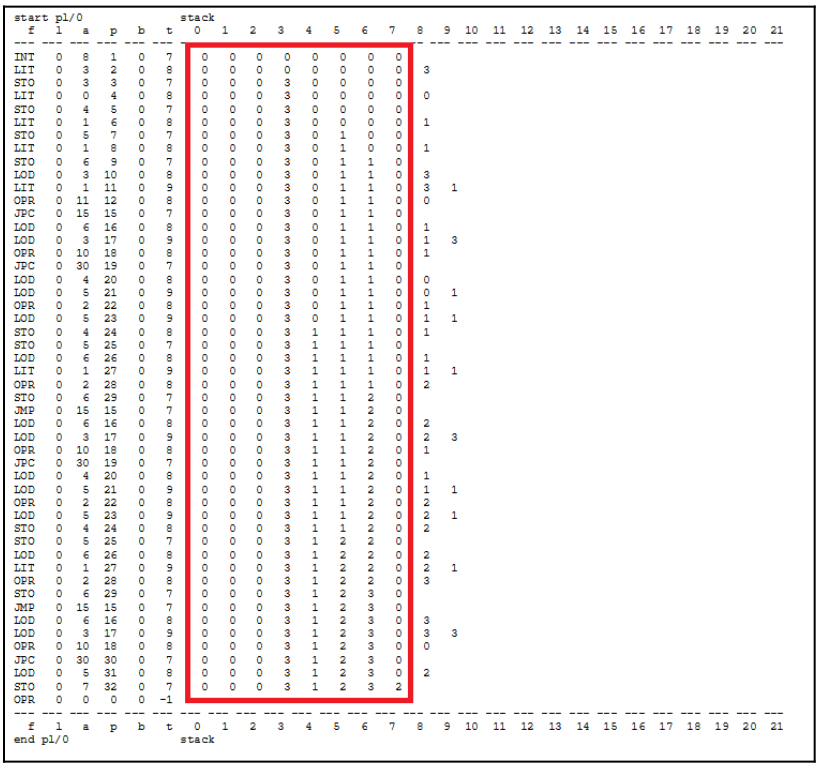
Mesmo que a AllanaVM possa calcular a sequência de Fibonacci até 93 e a P-code até 47, não foi possível a visualização da execução de tais valores na versão recursiva do programa, como se observa na Figura 19; deve-se ao fato de que ambas máquinas requerem maior tempo de processamento com funções recursivas, e especificamente na execução da função de Fibonacci a função de tempo apresenta tendências exponenciais, portanto optou-se pelo encerramento da execução ao chegar nos cálculos para os valores de 17 para AllanaVM e 16 para P-code.

4.3 Pilhas de execução da AllanaVM e P-code

A máquina de Brookshear possui compatibilidade com a AllanaVM, portanto foi utilizada como meio de comparação entre as máquinas, porém, a mesma não apresenta infraestrutura de chamada e retorno de sub-rotina, logo não possui uma pilha de execução, portanto nessa seção apenas são apresentados resultados referentes a AllanaVM e P-code. Utilizou-se para apresentação as pilhas de execução das funções Fibonacci(3) e fatorial(3) para as máquinas AllanaVM e P-code.

A pilha de execução da função iterativa de Fibonacci(3) para P-code observa-se na Figura 20, onde o bloco em vermelho indica o ARI da função main. O resultado da função pode ser observado na última linha do bloco, na posição 7 da pilha.

Figura 20 — Pilha da função iterativa de Fibonacci(3) para P-code



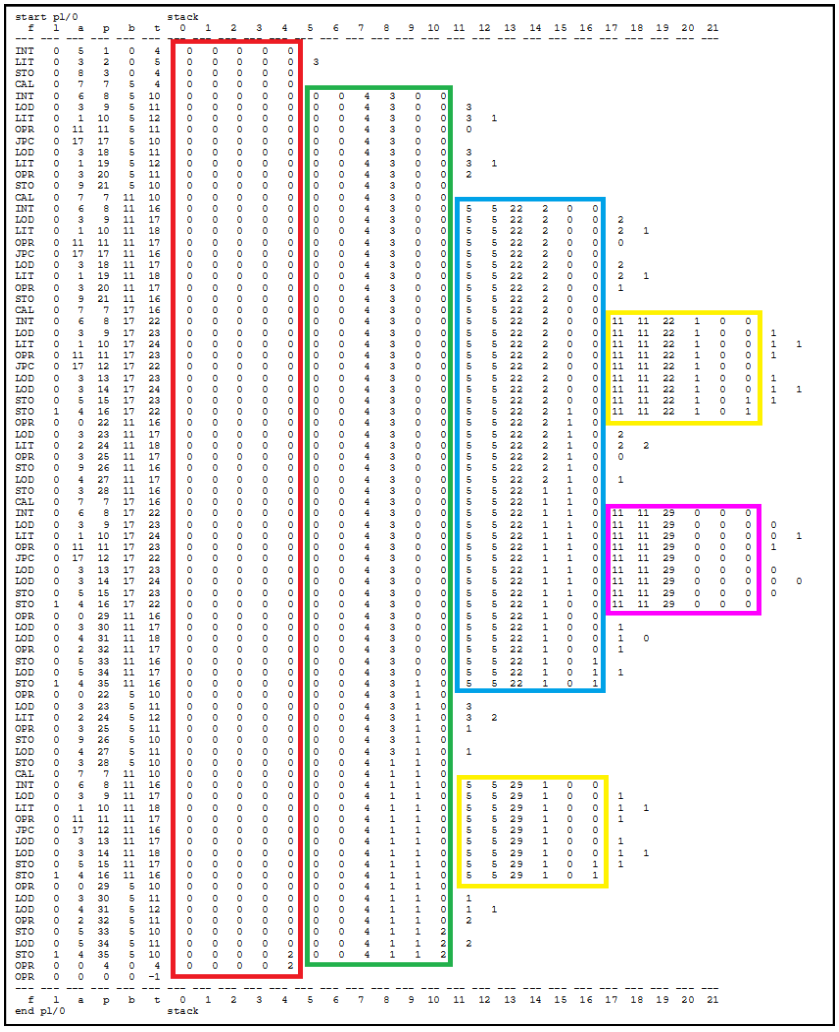
start pl/0						stack																			
f	l	a	p	b	t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
INT	0	8	1	0	7	0	0	0	0	0	0	0	0														
LIT	0	3	2	0	8	0	0	0	0	0	0	0	0	3													
STO	0	3	3	0	7	0	0	0	3	0	0	0	0														
LIT	0	0	4	0	8	0	0	0	3	0	0	0	0	0													
STO	0	4	5	0	7	0	0	0	3	0	0	0	0														
LIT	0	1	6	0	8	0	0	0	3	0	0	0	0	1													
STO	0	5	7	0	7	0	0	0	3	0	1	0	0														
LIT	0	1	8	0	8	0	0	0	3	0	1	0	0	1													
STO	0	6	9	0	7	0	0	0	3	0	1	1	0														
LOD	0	3	10	0	8	0	0	0	3	0	1	1	0	3													
LIT	0	1	11	0	9	0	0	0	3	0	1	1	0	3	1												
OPR	0	11	12	0	8	0	0	0	3	0	1	1	0	0													
JPC	0	15	15	0	7	0	0	0	3	0	1	1	0														
LOD	0	6	16	0	8	0	0	0	3	0	1	1	0	1													
LOD	0	3	17	0	9	0	0	0	3	0	1	1	0	1	3												
OPR	0	10	18	0	8	0	0	0	3	0	1	1	0	1													
JPC	0	30	19	0	7	0	0	0	3	0	1	1	0														
LOD	0	4	20	0	8	0	0	0	3	0	1	1	0	0													
LOD	0	5	21	0	9	0	0	0	3	0	1	1	0	0	1												
OPR	0	2	22	0	8	0	0	0	3	0	1	1	0	1													
LOD	0	5	23	0	9	0	0	0	3	0	1	1	0	1	1												
STO	0	4	24	0	8	0	0	0	3	1	1	1	0	1													
STO	0	5	25	0	7	0	0	0	3	1	1	1	0														
LOD	0	6	26	0	8	0	0	0	3	1	1	1	0	1													
LIT	0	1	27	0	9	0	0	0	3	1	1	1	0	1	1												
OPR	0	2	28	0	8	0	0	0	3	1	1	1	0	2													
STO	0	6	29	0	7	0	0	0	3	1	1	2	0														
JMP	0	15	15	0	7	0	0	0	3	1	1	2	0														
LOD	0	6	16	0	8	0	0	0	3	1	1	2	0	2													
LOD	0	3	17	0	9	0	0	0	3	1	1	2	0	2	3												
OPR	0	10	18	0	8	0	0	0	3	1	1	2	0	1													
JPC	0	30	19	0	7	0	0	0	3	1	1	2	0														
LOD	0	4	20	0	8	0	0	0	3	1	1	2	0	1													
LOD	0	5	21	0	9	0	0	0	3	1	1	2	0	1	1												
OPR	0	2	22	0	8	0	0	0	3	1	1	2	0	2													
LOD	0	5	23	0	9	0	0	0	3	1	1	2	0	2	1												
STO	0	4	24	0	8	0	0	0	3	1	1	2	0	2													
STO	0	5	25	0	7	0	0	0	3	1	2	2	0														
LOD	0	6	26	0	8	0	0	0	3	1	2	2	0	2													
LIT	0	1	27	0	9	0	0	0	3	1	2	2	0	2	1												
OPR	0	2	28	0	8	0	0	0	3	1	2	2	0	3													
STO	0	6	29	0	7	0	0	0	3	1	2	3	0														
JMP	0	15	15	0	7	0	0	0	3	1	2	3	0														
LOD	0	6	16	0	8	0	0	0	3	1	2	3	0	3													
LOD	0	3	17	0	9	0	0	0	3	1	2	3	0	3	3												
OPR	0	10	18	0	8	0	0	0	3	1	2	3	0	0													
JPC	0	30	30	0	7	0	0	0	3	1	2	3	0														
LOD	0	5	31	0	8	0	0	0	3	1	2	3	0	2													
STO	0	7	32	0	7	0	0	0	3	1	2	3	2														
OPR	0	0	0	0	-1																						
f	l	a	p	b	t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
end pl/0						stack																					

Fonte: Autora

Na Figura 21 encontra-se a pilha de execução da função recursiva de Fibonacci(3) para a P-code, onde o bloco em vermelho indica o ARI da função main, em verde o ARI para Fibonacci(3), em azul o ARI para Fibonacci(2) e em amarelo o ARI para Fibonacci(1)

que se repete duas vezes, sendo chamado por Fibonacci(2) e por Fibonacci(3) além do ARI para Fibonacci(0) em magenta.

Figura 21 — Pilha da função recursiva de Fibonacci(3) para P-code

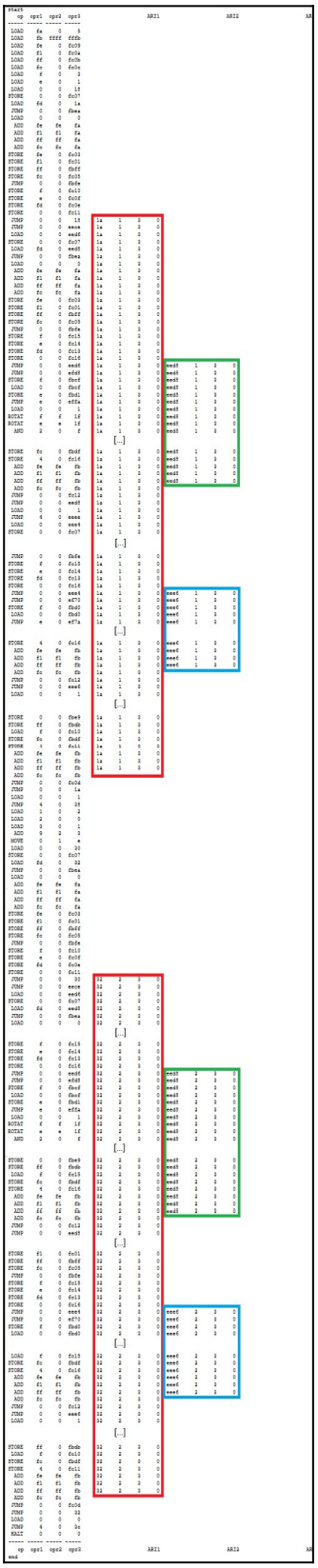


```
start pl/0                stack
  f   l   a   p   b   t   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
INT   0   5   1   0   4   0   0   0   0   0
LIT   0   3   2   0   5   0   0   0   0   0   3
STO   0   8   3   0   4   0   0   0   0   0
CAL   0   7   7   5   4   0   0   0   0   0
INT   0   6   8   5  10   0   0   0   0   0   0   0   4   3   0   0
LOD   0   3   9   5  11   0   0   0   0   0   0   0   4   3   0   0   3
LIT   0   1  10   5  12   0   0   0   0   0   0   0   4   3   0   0   3   1
OPR   0  11  11   5  11   0   0   0   0   0   0   0   4   3   0   0   0
JPC   0  17  17   5  10   0   0   0   0   0   0   0   4   3   0   0
LOD   0   3  18   5  11   0   0   0   0   0   0   0   4   3   0   0   3
LIT   0   1  19   5  12   0   0   0   0   0   0   0   4   3   0   0   3   1
OPR   0   3  20   5  11   0   0   0   0   0   0   0   4   3   0   0   2
STO   0   9  21   5  10   0   0   0   0   0   0   0   4   3   0   0
CAL   0   7   7  11  10   0   0   0   0   0   0   0   4   3   0   0
INT   0   6   8  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0
LOD   0   3   9  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0   2
LIT   0   1  10  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0   2   1
OPR   0  11  11  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0   0
JPC   0  17  17  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0
LOD   0   3  18  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0   2
LIT   0   1  19  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0   2   1
OPR   0   3  20  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0   1
STO   0   9  21  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0
CAL   0   7   7  17  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0
INT   0   6   8  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0
LOD   0   3   9  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0   1
LIT   0   1  10  17  24   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0   1   1
OPR   0  11  11  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0   1
JPC   0  17  12  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0
LOD   0   3  13  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0   1
LOD   0   3  14  17  24   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   0   1   1
STO   0   5  15  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   0   0  11  11  22   1   0   1   1
STO   1   4  16  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0  11  11  22   1   0   1
OPR   0   0  22  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0
LOD   0   3  23  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0   2
LIT   0   2  24  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0   2   2
OPR   0   3  25  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0   0
STO   0   9  26  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0
LOD   0   4  27  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   2   1   0   1
STO   0   3  28  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0
CAL   0   7   7  17  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0
INT   0   6   8  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0
LOD   0   3   9  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0   0
LIT   0   1  10  17  24   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0   0   1
OPR   0  11  11  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0   1
JPC   0  17  12  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0
LOD   0   3  13  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0   0
LOD   0   3  14  17  24   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0   0   0
STO   0   5  15  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   1   0  11  11  29   0   0   0   0
STO   1   4  16  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   0  11  11  29   0   0   0
OPR   0   0  29  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   0
LOD   0   3  30  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   0   1
LOD   0   4  31  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   0   1   0
OPR   0   2  32  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   0   1
STO   0   5  33  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   1
LOD   0   5  34  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  22   1   0   1   1
STO   1   4  35  11  16   0   0   0   0   0   0   0   4   3   1   0   5   5  22   1   0   1
OPR   0   0  22   5  10   0   0   0   0   0   0   0   4   3   1   0
LOD   0   3  23   5  11   0   0   0   0   0   0   0   4   3   1   0   3
LIT   0   2  24   5  12   0   0   0   0   0   0   0   4   3   1   0   3   2
OPR   0   3  25   5  11   0   0   0   0   0   0   0   4   3   1   0   1
STO   0   9  26   5  10   0   0   0   0   0   0   0   4   3   1   0
LOD   0   4  27   5  11   0   0   0   0   0   0   0   4   3   1   0   1
STO   0   3  28   5  10   0   0   0   0   0   0   0   4   1   1   0
CAL   0   7   7  11  10   0   0   0   0   0   0   0   4   1   1   0
INT   0   6   8  11  16   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0
LOD   0   3   9  11  17   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0   1
LIT   0   1  10  11  18   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0   1   1
OPR   0  11  11  11  17   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0   1
JPC   0  17  12  11  16   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0
LOD   0   3  13  11  17   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0   1
LOD   0   3  14  11  18   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   0   1   1
STO   0   5  15  11  17   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   1   1
STO   1   4  16  11  16   0   0   0   0   0   0   0   4   1   1   0   5   5  29   1   0   1
OPR   0   0  29   5  10   0   0   0   0   0   0   0   4   1   1   0
LOD   0   3  30   5  11   0   0   0   0   0   0   0   4   1   1   0   1
LOD   0   4  31   5  12   0   0   0   0   0   0   0   4   1   1   0   1   1
OPR   0   2  32   5  11   0   0   0   0   0   0   0   4   1   1   0   2
STO   0   5  33   5  10   0   0   0   0   0   0   0   4   1   1   2
LOD   0   5  34   5  11   0   0   0   0   0   0   0   4   1   1   2   2
STO   1   4  35   5  10   0   0   0   0   0   2   0   4   1   1   2
OPR   0   0   4   0   4   0   0   0   0   2
OPR   0   0   0   0  -1
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
  f   l   a   p   b   t   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
end pl/0                  stack
```

Fonte: Autora

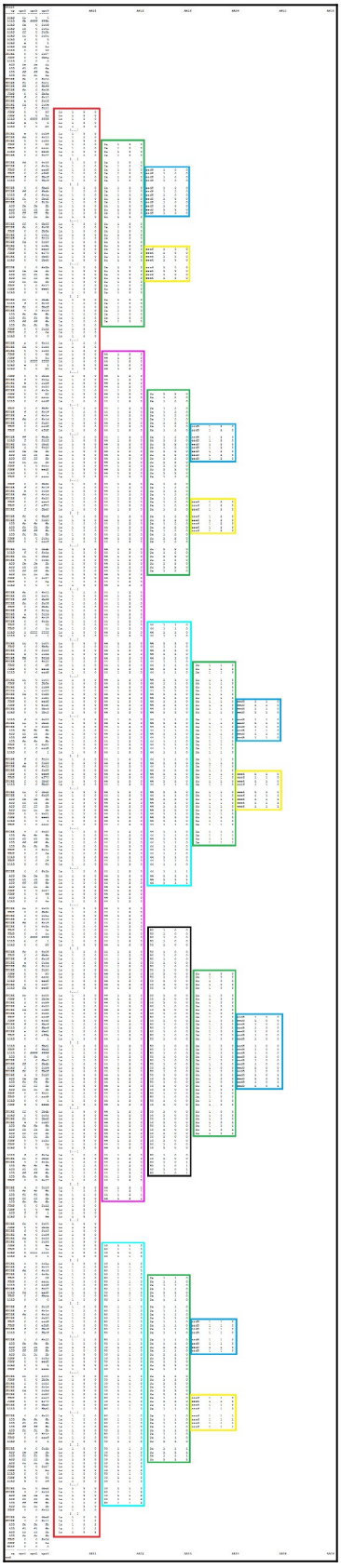
As funções para AllanaVM não possui ARI referente ao main. Os ARIs para função iterativa de Fibonacci(3) para AllanaVM encontram-se na Figura 23, onde os blocos em vermelho indicam as operações de GEQ, em verde as operações de GTR e em azul as operações de EQL. Observa-se que os ARIs possuem espaçamentos em branco; como existem muitas instruções para execução da função, optou-se por ocultar algumas instruções para melhor visualização.

Figura 23 — ARIs da função iterativa de Fibonacci(3) para AllanaVM



Fonte: Autora

Figura 22 — ARIs da função recursiva de Fibonacci(3) para AllanaVM



Fonte: Autora

A pilha de execução da função recursiva de Fibonacci(3) para AllanaVM é visualizada na Figura 22, onde o bloco em vermelho indica o ARI para Fibonacci(3), em verde são para as operações de GEQ, em azul as operações de GTR, em amarelo as operações de EQL, em magenta o ARI para Fibonacci(2), em ciano o ARI para Fibonacci(1) que se repete duas vezes, sendo chamado por Fibonacci(2) e por Fibonacci(3) e, em preto o ARI para Fibonacci(0). Assim como a função iterativa, optou-se por ocultar algumas instruções para melhor visualização.

A pilha de execução da função iterativa de fatorial(3) para P-code observa-se na Figura 24, onde o bloco em vermelho indica o ARI da função main.

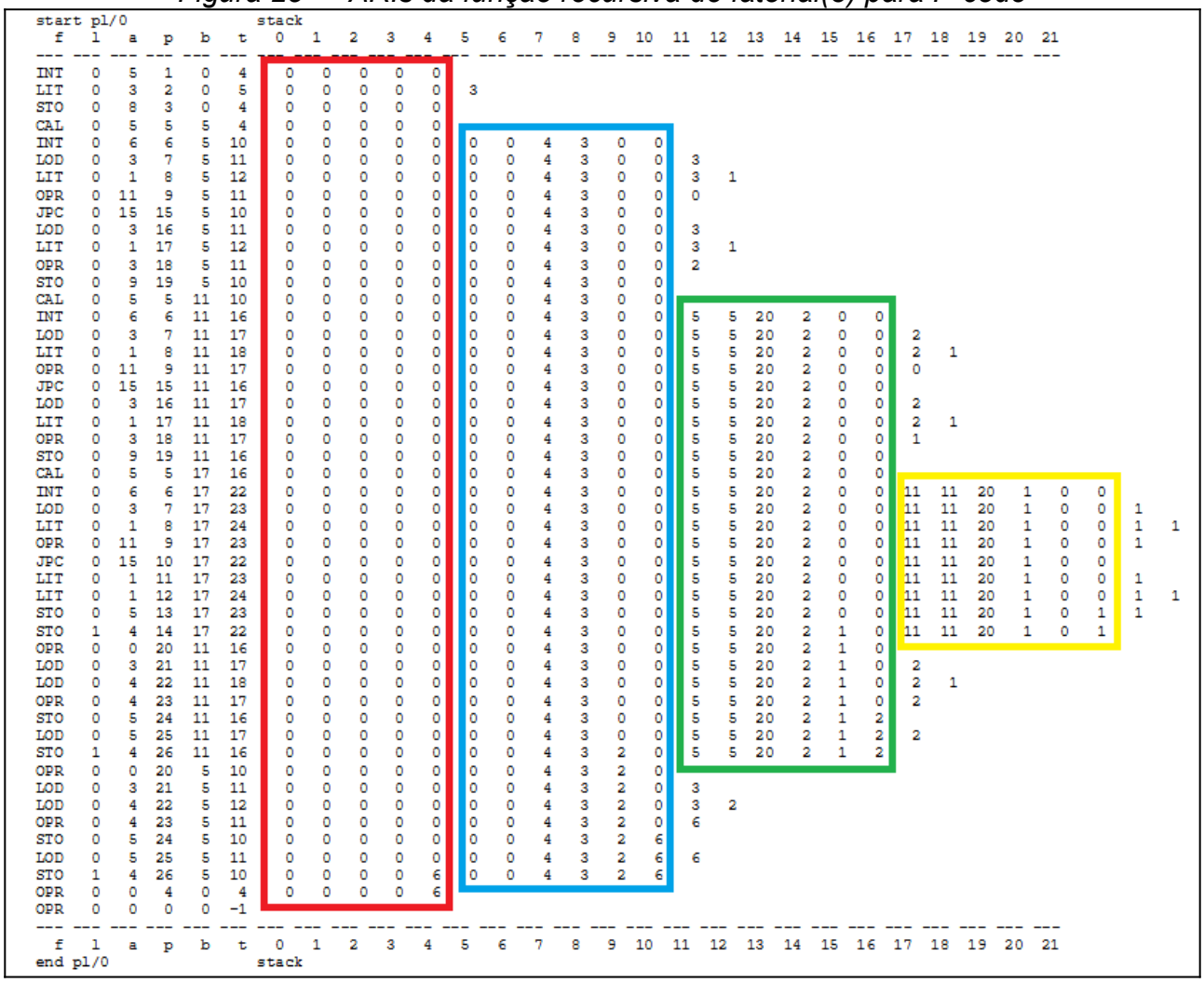
Figura 24 — Pilha da função iterativa de fatorial(3) para P-code

```
start pl/0                stack
  f   l   a   p   b   t   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
INT   0   5   1   0   4   0   0   0   0   0
LIT   0   1   2   0   5   0   0   0   0   0   1
STO   0   3   3   0   4   0   0   0   1   0
LIT   0   1   4   0   5   0   0   0   1   0   1
STO   0   4   5   0   4   0   0   0   1   1
LOD   0   3   6   0   5   0   0   0   1   1   1
LIT   0   3   7   0   6   0   0   0   1   1   1   3
OPR   0  13   8   0   5   0   0   0   1   1   0
JPC   0  10  10   0   4   0   0   0   1   1
LOD   0   3  11   0   5   0   0   0   1   1   1
LIT   0   1  12   0   6   0   0   0   1   1   1   1
OPR   0   2  13   0   5   0   0   0   1   1   2
STO   0   3  14   0   4   0   0   0   2   1
LOD   0   3  15   0   5   0   0   0   2   1   2
LOD   0   4  16   0   6   0   0   0   2   1   2   1
OPR   0   4  17   0   5   0   0   0   2   1   2
STO   0   4  18   0   4   0   0   0   2   2
JMP   0   5   5   0   4   0   0   0   2   2
LOD   0   3   6   0   5   0   0   0   2   2   2
LIT   0   3   7   0   6   0   0   0   2   2   2   3
OPR   0  13   8   0   5   0   0   0   2   2   0
JPC   0  10  10   0   4   0   0   0   2   2
LOD   0   3  11   0   5   0   0   0   2   2   2
LIT   0   1  12   0   6   0   0   0   2   2   2   1
OPR   0   2  13   0   5   0   0   0   2   2   3
STO   0   3  14   0   4   0   0   0   3   2
LOD   0   3  15   0   5   0   0   0   3   2   3
LOD   0   4  16   0   6   0   0   0   3   2   3   2
OPR   0   4  17   0   5   0   0   0   3   2   6
STO   0   4  18   0   4   0   0   0   3   6
JMP   0   5   5   0   4   0   0   0   3   6
LOD   0   3   6   0   5   0   0   0   3   6   3
LIT   0   3   7   0   6   0   0   0   3   6   3   3
OPR   0  13   8   0   5   0   0   0   3   6   1
JPC   0  10   9   0   4   0   0   0   3   6
OPR   0   0   0   0  -1
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
  f   l   a   p   b   t   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
end pl/0                  stack
```

Fonte: Autora

Na Figura 25 encontram-se os ARIs para a chamada e retorno da função recursiva de fatorial(3) para a P-code, onde o bloco em vermelho indica o ARI da função main, em verde o ARI para fatorial(3), em azul o ARI para fatorial(2) e em amarelo o ARI para fatorial(1).

Figura 25 — ARIs da função recursiva de fatorial(3) para P-code



```
start pl/0                                  stack
  f   l   a   p   b   t   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
INT   0   5   1   0   4   0   0   0   0   0
LIT   0   3   2   0   5   0   0   0   0   0   3
STO   0   8   3   0   4   0   0   0   0   0
CAL   0   5   5   5   4   0   0   0   0   0
INT   0   6   6   5  10   0   0   0   0   0   0   0   4   3   0   0
LOD   0   3   7   5  11   0   0   0   0   0   0   0   4   3   0   0   3
LIT   0   1   8   5  12   0   0   0   0   0   0   0   4   3   0   0   3   1
OPR   0  11   9   5  11   0   0   0   0   0   0   0   4   3   0   0   0
JPC   0  15  15   5  10   0   0   0   0   0   0   0   4   3   0   0
LOD   0   3  16   5  11   0   0   0   0   0   0   0   4   3   0   0   3
LIT   0   1  17   5  12   0   0   0   0   0   0   0   4   3   0   0   3   1
OPR   0   3  18   5  11   0   0   0   0   0   0   0   4   3   0   0   2
STO   0   9  19   5  10   0   0   0   0   0   0   0   4   3   0   0
CAL   0   5   5  11  10   0   0   0   0   0   0   0   4   3   0   0
INT   0   6   6  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0
LOD   0   3   7  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0   2
LIT   0   1   8  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0   2   1
OPR   0  11   9  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0   0
JPC   0  15  15  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0
LOD   0   3  16  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0   2
LIT   0   1  17  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0   2   1
OPR   0   3  18  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0   1
STO   0   9  19  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0
CAL   0   5   5  17  16   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0
INT   0   6   6  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0
LOD   0   3   7  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0   1
LIT   0   1   8  17  24   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0   1   1
OPR   0  11   9  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0   1
JPC   0  15  10  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0
LIT   0   1  11  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0   1
LIT   0   1  12  17  24   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   0   1   1
STO   0   5  13  17  23   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   0   0  11  11  20   1   0   1   1
STO   1   4  14  17  22   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   0  11  11  20   1   0   1
OPR   0   0  20  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   0
LOD   0   3  21  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   0   2
LOD   0   4  22  11  18   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   0   2   1
OPR   0   4  23  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   0   2
STO   0   5  24  11  16   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   2
LOD   0   5  25  11  17   0   0   0   0   0   0   0   4   3   0   0   5   5  20   2   1   2   2
STO   1   4  26  11  16   0   0   0   0   0   0   0   4   3   2   0   5   5  20   2   1   2
OPR   0   0  20   5  10   0   0   0   0   0   0   0   4   3   2   0
LOD   0   3  21   5  11   0   0   0   0   0   0   0   4   3   2   0   3
LOD   0   4  22   5  12   0   0   0   0   0   0   0   4   3   2   0   3   2
OPR   0   4  23   5  11   0   0   0   0   0   0   0   4   3   2   0   6
STO   0   5  24   5  10   0   0   0   0   0   0   0   4   3   2   6
LOD   0   5  25   5  11   0   0   0   0   0   0   0   4   3   2   6   6
STO   1   4  26   5  10   0   0   0   0   6   0   0   4   3   2   6
OPR   0   0   4   0   4   0   0   0   0   6
OPR   0   0   0   0  -1
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
  f   l   a   p   b   t   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
end pl/0                                    stack
```

Fonte: Autora

Os ARIs para a função iterativa de fatorial(3) para AllanaVM encontram-se na Figura 27, onde os blocos em vermelho indicam as operações de MUL, que é responsável pela multiplicação entre dois valores.

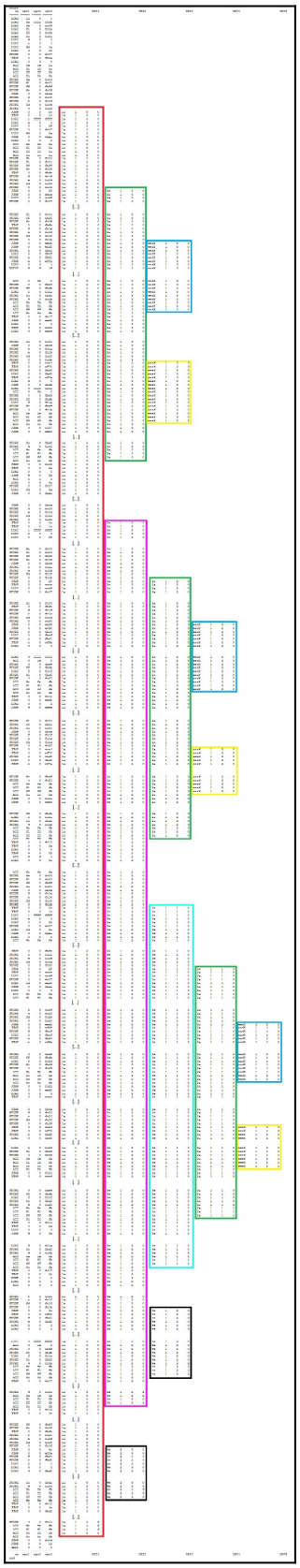
Os ARIs para chamada e retorno da função recursiva de fatorial(3) para AllanaVM é visualizada na Figura 26, onde o bloco em vermelho indica o ARI para fatorial(3), em verde são para as operações de GEQ, em azul as operações de GTR, em amarelo as operações de EQL, em magenta o ARI para fatorial(2), em ciano o ARI para fatorial(1) e, em preto o ARI para a operação de MUL. Observa-se que a pilha possui espaçamentos em branco; como existem muitas instruções para execução da função, optou-se por ocultar algumas instruções para melhor visualização.

Figura 27 — ARIs da função iterativa de fatorial(3) para AllanaVM

```
start
   op  opr1  opr2  opr3                    ARI1
----- ----- ----- -----
 LOAD    fa     0     5
 LOAD    fb  ffff  fffb
 LOAD    fe     0  fc09
 LOAD    f1     0  fc0a
 LOAD    ff     0  fc0b
 LOAD    fc     0  fc0c
 LOAD     f     0     3
 MOVE     0     f     4
 LOAD     5  ffff  ffff
 LOAD     0     0     1
 JUMP     f     0    26
  ADD     f     f     5
 MOVE     0     4     e
 LOAD     0     0    22
STORE     0     0  fc07
 LOAD    fd     0    24
 JUMP     0     0  fbea
 LOAD     0     0     0
  ADD    fe    fe    fa
  ADD    f1    f1    fa
  ADD    ff    ff    fa
  ADD    fc    fc    fa
STORE    fe     0  fc03
STORE    f1     0  fc01
STORE    ff     0  fbff
STORE    fc     0  fc05
 JUMP     0     0  fbfe
STORE     f     0  fc10
STORE     e     0  fc0f
STORE    fd     0  fc0e
STORE     0     0  fc11
 JUMP     0     0    22    24     3     2     0
 JUMP     0     0  ef8e    24     3     2     0
STORE     f     0  fbd1    24     3     2     0
STORE     e     0  fbd0    24     3     2     0
 LOAD     2     0     1    24     3     2     0
 LOAD     3     0     1    24     3     2     0
 LOAD     4     0  fbd0    24     3     2     0
 LOAD     0     0  fbd1    24     3     2     0
 LOAD     1     0  fbd0    24     3     2     0
 JUMP     2     0  fbd2    24     3     2     0
  ADD     4     4     1    24     3     2     0
  ADD     2     2     3    24     3     2     0
 LOAD     0     0     1    24     3     2     0
 JUMP     3     0  ef98    24     3     2     0
 LOAD     0     0  fbd1    24     3     2     0
 LOAD     1     0  fbd0    24     3     2     0
 JUMP     2     0  fbd2    24     3     2     0
 LOAD     0  ffff  ffff    24     3     2     0
  ADD     0    fe     0    24     3     2     0
STORE     0     0  fbe9    24     3     2     0
STORE    ff     0  fbdb    24     3     2     0
 LOAD     f     0  fc10    24     3     2     0
STORE    fc     0  fbdf    24     3     2     0
STORE     4     0  fc11    24     3     2     6
  ADD    fe    fe    fb    24     3     2     6
  ADD    f1    f1    fb    24     3     2     6
  ADD    ff    ff    fb    24     3     2     6
  ADD    fc    fc    fb
 JUMP     0     0  fc0d
 JUMP     0     0    24
 JUMP     0     0    12
 LOAD     0     0     1
 JUMP     f     0    26
  ADD     f     f     5
 MOVE     0     4     e
 LOAD     0     0    22
STORE     0     0  fc07
 LOAD    fd     0    24
 JUMP     0     0  fbea
 LOAD     0     0     0
  ADD    fe    fe    fa
  ADD    f1    f1    fa
  ADD    ff    ff    fa
  ADD    fc    fc    fa
STORE    fe     0  fc03
STORE    f1     0  fc01
STORE    ff     0  fbff
STORE    fc     0  fc05
 JUMP     0     0  fbfe
STORE     f     0  fc10
STORE     e     0  fc0f
STORE    fd     0  fc0e
STORE     0     0  fc11
 JUMP     0     0    22    24     6     1     0
 JUMP     0     0  ef8e    24     6     1     0
STORE     f     0  fbd1    24     6     1     0
STORE     e     0  fbd0    24     6     1     0
 LOAD     2     0     1    24     6     1     0
 LOAD     3     0     1    24     6     1     0
 LOAD     4     0  fbd0    24     6     1     0
 LOAD     0     0  fbd1    24     6     1     0
 LOAD     1     0  fbd0    24     6     1     0
 JUMP     2     0  fbd2    24     6     1     0
 LOAD     0  ffff  ffff    24     6     1     0
  ADD     0    fe     0    24     6     1     0
STORE     0     0  fbe9    24     6     1     0
STORE    ff     0  fbdb    24     6     1     0
 LOAD     f     0  fc10    24     6     1     0
STORE    fc     0  fbdf    24     6     1     0
STORE     4     0  fc11    24     6     1     6
  ADD    fe    fe    fb    24     6     1     6
  ADD    f1    f1    fb    24     6     1     6
  ADD    ff    ff    fb    24     6     1     6
  ADD    fc    fc    fb
 JUMP     0     0  fc0d
 JUMP     0     0    24
 JUMP     0     0    12
 LOAD     0     0     1
 JUMP     f     0    26
 HALT     0     0     0
----- ----- ----- -----
   op  opr1  opr2  opr3                    ARI1
end
```

Fonte: Autora

Figura 26 — ARIs da função recursiva de fatorial(3) para AllanaVM



Fonte: Autora

5 CONCLUSÃO

Em 100% dos testes AllanaVM conseguiu calcular valores maiores que as outras máquinas e em 25% dos testes a quantidade de ciclos de clock necessários da AllanaVM é menor quando que as outras máquinas.

Como os resultados dos cálculos são armazenados na memória, o valor máximo permitido para armazenagem em uma máquina dependerá do tamanho de palavra da mesma, como as células da memória da máquina AllanaVM possuem 64 bits de comprimento, a mesma consegue armazenar valores maiores que a Brookshear (8 bits) e a P-code (32 bits); consequentemente a AllanaVM alcançou maiores valores resultantes em seus cálculos que as demais máquinas.

Em 75% dos testes a AllanaVM não alcançou uma resposta mais rápida para cada cálculo que as demais máquinas por dois fatores:

a) a P-code possui arquitetura Harvard, que contém dois barramentos de memória, um para os dados e outro para as instruções, permitindo que dados e instruções se movimentem ao mesmo tempo entre a CPU e a Unidade Lógica e Aritmética (ULA);
b) as instruções de máquina AllanaVM que são derivadas da P-code são consideradas sub-rotinas, como explicadas na sessão 3.3.2 e observados nas Figuras 23, 22, 26 e 27, portanto o tempo de execução de tais instruções na AllanaVM demanda mais tempo do que na P-code.

Tabela 7 — Comparação entre as máquinas AllanaVM, Brookshear e P-code

	AllanaVM	Brookshear	P-code
Registradores	256	16	3
Memória	2^{64}	2^{8}	200 - instruções 500 - dados
Tamanho da Palavra	64 bits	8 bits	32 bits
Instruções	25 - 128 bits	12 - 16 bits	8 - 32 bits
Arquitetura	Von Neumann	Von Neumann	Harvard
Infraestrutura de chamada/retorno de sub-rotina	Sim	Não	Sim

Fonte: Autora

Na Tabela 7 encontra-se informações da arquitetura de cada máquina, onde em verde apresenta-se os valores máximos para cada componente, em amarelo os valores intermediários e em vermelho os valores mínimos, levando em conta as máquinas AllanaVM, Brookshear e P-code.

Ao observar a Tabela 7 percebe-se que AllanaVM possui a maior quantidade de registradores, mais células de memória e o maior conjunto de instruções, enquanto as máquinas Brookshear e P-code apresentam em sua maior parte componentes com valores intermediários e mínimos. A AllanaVM e a P-code são as únicas que possuem infraestrutura para chamada e retorno de sub-rotina. As arquiteturas das máquinas AllanaVM e Brookshear são Von Neumann, que lhes permitem a alteração do código em tempo de execução, enquanto que a P-code possui arquitetura Harvard, que protege a memória de programa.

Conclui-se que para programas que necessitam utilizar mais memória ou armazenar valores altos em um tempo razoável de execução, a melhor indicação de máquina é a AllanaVM, além de que o tempo de execução das instruções derivadas da P-code podem ser alterados ao otimizar as sub-rotinas referentes as mesmas.

Como trabalho futuro se pretende aproveitar a versatilidade que a arquitetura Von Neumann proporciona e implementar um escalonador de processos para o desenvolvimento posterior de um sistema operacional.

REFERÊNCIAS

ALPERT; Donald. **A pascal P-code interpreter for the Stanford EMMY**. Universidade de Stanford, Laboratório de Sistemas de Computação. Nota técnica nº 164. California, 1979. Disponível em: <http://bitsavers.informatik.uni-stuttgart.de/pdf/stanford/sel_techReports/TN164_A_Pascal_P-Code_Interpreter_for_the_Stanford_Emmy_Sep79.pdf>. Acesso em: 03 de Junho de 2022.

BROOKSHEAR, J. Glenn. **Ciência da Computação:** Uma visão abrangente. Tradução Cheng Mei Lee. 7 ed. Porto Alegre: Bookman, 2005. 515 p.

CAMPOS, Allana S. **VMs**. 2022. Disponível em: <https://github.com/AllanaCampos/VMs>.

GHANNOUM, Rodrigo G; RODRIGUES, Fábio B. Virtualização de Servidores: Vantagens e Desvantagens. **Revista Mirante**. Goiânia, abr. 2018. v. 11, n. 6 (edição especial).

GOLDBERG, Robert P. **Archtectural Principles for Virtual Computer Systems**. 1973. 249 p. Tese (Doutorado em Engenharia e Física Aplicada) – Harvard University, Cambridge, 1973.

HERMES, H. **Enumerability Decidability Computability**. Vol 127. Nova York: Heidelberg, 1965. 255 p.

HIALINX. **O surgimento da virtualização e as grandes mudanças que ela teve**. 2019. Disponível em: <https://www.hialinx.com.br/post/o-surgimento-da-virtualização-e-as-grandes-mudanças-que-ela-trouxe>. Acesso em: 14 de Junho de 2022.

IBM. **System/360 From Computers to Computer Systems**. 2022. Disponível em: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/system360/>. Acesso em: 13 de Junho de 2022.

INNOTEK. **InnoTek VirtualBox User Manual**. 2007. Disponível em: <https://www.mclibre.org/descargar/webapps/virtualbox-1-3-8-user-manual.pdf>. Acesso em: 07 de Julho de 2022.

KREUTZ, Diego L; MACEDO, Douglas D. J; ARBIZA, Lucas M. R. **Virtualização: Conceitos, Aplicações, Mercado e Prática**. Escola Regional de Redes de Computadores, 2009. Capítulo1, SBC.

KUBRUSLY, Jessica. **Uma introdução à programação com R**. Bookdown, 2021. 343 p. Disponível em: <https://bookdown.org/jessicakubrusly/programacao-estatistica/>. Acesso em: 23 de Junho de 2022.

LAWRENCE, Geoff. **An Introduction to Virtual Box**. 2015. Disponível em: <https://capgemini.github.io/infrastructure/an-introduction-to-virtualbox/>. Acesso em: 28 de Junho de 2022.

MICROSOFT. **Azure**. 2022a. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-azure/>. Acesso em: 11 de Julho de 2022.

MICROSOFT. **Introdução ao Hyper-V no Windows 10**. 2022b. Disponível em: <https://docs.microsoft.com/pt-br/virtualization/hyper-v-on-windows/about/>. Acesso em: 14 de Junho de 2022.

MOARVM. **MoarVM**. 2022. Disponível em: <https://moarvm.org>. Acesso em: 23 de Junho de 2022.

MONNE, Roger P. **Xen Project ships version 4.16 with focus on improved performance security and hardware support**. 2021. Disponível em: <https://xenproject.org/2021/12/02/xen-project-version-4-16/>. Acesso em: 14 de Junho de 2022.

NEUHAUSER, C. **EMMY system processor: principles of operation**. Standord Univ. United States, 1997.

NUTANIX. **AHV**. 2022. Disponível em: <https://www.nutanix.com/br/products/ahv>. Acesso em: 09 de Julho de 2022.

ORACLE. **Oracle VM**. 2021. Disponível em: <https://docs.oracle.com/en/virtualization/oracle-vm/3.4/concepts/E64081.pdf>. Acesso em: 23 de Junho de 2022.

ORACLE. **Oracle VM VirtualBox**. 2020. Disponível em: <https://www.oracle.com/br/a/ocom/docs/oracle-vm-virtualbox-ds-1655169.pdf>. Acesso em: 14 de Junho de 2022.

OPENSTACK. **The Most Widely Deployeed Open Source Cloud Software in the World**. 2022. Disponível em: <https://www.openstack.org>. Acesso em: 09 de Julho de 2022.

PARROT. **Parrot Virtual Machine**. 2021. Disponível em: <https://github.com/parrot/parrot>. Acesso em: 23 de Junho de 2022.

REMZI, H; ARPACI-DUSSEAU; ARPACI-DUSSEAU, Andrea C. **Operating Systems: Three Easy Pieces**. 1ª Edição. Arpaci-Dusseau Books, 2018.

ROSENBLUM, Mendel; Stanford Universit; VMWare. **The Reincarnation of Virtual Machines**. ACMQueue. Agosto 2004. vol 2 issue 5. Disponível em: <https://queue.acm.org/detail.cfm?id=1017000>. Acesso em: 24 de Junho de 2022.

RUBINIUS, Inc. **The Rubinius Book**. 2016. Disponível em: <https://book.rubinius.com>. Acesso em: 07 de Junho de 2022.

SASADA, Koichi. **YARV: Yet Another RubyVM**. San Diego: OOSPLA ‘05, 2005. Disponível em: <http://www.atdot.net/yarv/oopsla2005eabstract-rc1.pdf>. Acesso em: 07

de Junho de 2022.

SEBESTA, Robert W. **Concepts of Programming Languages**. 10ª ed. Pearson, 2012. 816 p.

SINGH, Himanshu. **Next-Gen Virtualization for Dummies, Vmware Special Edition**. New Jersey: John Wiley & Sons, Inc. 2019. 64 p.

SMITH, James E; NAIR, Ravi. **Virtual Machines**. Califórnia: Elsevier Inc, 2005. 661 p.

SOROKER, Tali. **CLR vs JVM: How the battle between C# and Java extends to the VM -Level**. 2018. Disponível em: <https://www.overops.com/blog/clr-vs-jvm-how-the-battle-between-net-and-java-extends-to-the-vm-level/>. Acesso em: 21 de Junho de 2022.

STALLINGS, William. **Arquitetura e Organização de Computadores**. Tradução Daniel Vieira e Ivan Bosnic. 8ª ed. São Paulo: Pearson Pratice Hall, 2010. 643 p.

SUN MICROSYSTEMS. **Company Info**. 2006. Disponível em: <https://web.archive.org/web/20060828042628/http://sun.com/aboutsun/company/facts.jsp>. Acesso em: 07 de Junho de 2022.

TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização Estruturada de computadores**. Tradução Daniel Vieira. 6ª ed. São Paulo: Pearson Education, 2013. 626 p.

TANENBAUM, Andrew S.; BOS, Herbert. **Sistemas Operacionais Modernos**. Tradução Jorge Ritter. 4ª ed. São Paulo: Pearson Education, 2016. 778 p.

VERAS, Manoel. **Virtualização: Componente Central do Datacenter**. Rio de Janeiro: Brasport Livros e Multimídia Ltda, 2011. 342 p.

VMWARE. **Using Vmware Workstation Pro**. 2021. Disponível em: <https://docs.vmware.com/en/Vmware-Workstation-Pro/16.0/workstation-pro-16-user-guide.pdf>. Acesso em: 14 de Junho de 2022.

WEBER, Gabriel S. **Desenvolvimento de um Compilador de Português Estruturado para a JVM no Portal de Algoritmos da UCS**. 2016. 88 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade de Caxias do Sul, Caxias do Sul, 2016.

WIRTH, Niklaus. **Algorithms + Data Structures = Programs**. Nova Jersei: Prentice-Hall, 1990. 381 p.

WIKIPEDIA. **Windows Virtual PC**. 2022. Disponível em: <https://en.wikipedia.org/wiki/Windows_Virtual_PC#Virtual_PC_by_Connectix>. Acesso em: 28 de Junho de 2022.

ZANOELLO, Thiago B. **Curso de introdução a criação e uso de Máquinas Virtuais**. 1ª Edição. São Paulo: Seção Técnica de Informática – USP, 2017. 28 p.

APÊNDICE A — Conjunto de Instruções e Operações da P-code Machine

A Tabela 8 descreve o conjunto de instruções da P-code Machine com código e operandos que podem ser utilizados, a Tabela 9 descreve os operandos que podem ser utilizados na instrução de código 2 “OPR”, para utilizar as operações basta utilizar os operandos informados na Tabela 9 ao chamar a instrução de código 2.

Tabela 8 — Conjunto de instruções da P-code machine

Código	Operando	Descrição
1	0, a	LIT: insere uma constante *a* na pilha.
2	0, a	OPR: executa uma operação *a.*
3	l, a	LOD: insere a variável que se encontra no nível *l*, posição *a*, no topo da pilha.
4	l, a	STO: armazena o valor do topo da pilha na posição *a* do nível *l*.
5	l, a	CAL: chama a sub-rotina na posição *a* do nível *l*.
6	0, a	INT: incrementa o registrador de topo da pilha por *a*.
7	0, a	JMP: pula para a instrução no endereço *a*.
8	0, a	JPC: retira o valor do topo da pilha, caso seja 0 pula para a instrução no endereço *a*, caso contrário segue a execução normalmente.

Fonte: Autora

Tabela 9 — Conjunto de operações da P-code machine

Código	Operando	Descrição continua
2	0, 0	RTN: Encerra a execução da sub-rotina
2	0, 1	NEG: Nega o valor do topo da pilha.
2	0, 2	ADD: Soma os dois valores no topo da pilha.
2	0, 3	SUB: Subtrai os dois valores no topo da pilha.
2	0, 4	MUL: Multiplica os dois valores no topo da pilha.
2	0, 5	DIV: Divide os dois valores no topo da pilha.
2	0, 6	ODD: Retira o valor do topo da pilha, caso seja ímpar, insere 1 no topo da pilha, caso contrário insere 0.
2	0, 7	MOD: Retira o valor do topo da pilha, insere o resto da divisão por

		2 no topo da pilha.
Código	Operando	Descrição conclusão
2	0, 8	EQL: Retira os dois valores da pilha, caso sejam iguais insere 1 no topo da pilha, caso contrário insere 0.
2	0, 9	NEQ: Retira os dois valores da pilha, caso sejam diferentes insere 1 no topo da pilha, caso contrário insere 0.
2	0, 10	LSS: Retira os dois valores da pilha, se o penúltimo valor for menor que o valor do topo insere 1 no topo da pilha, caso contrário insere 0.
2	0, 11	LEQ: Retira os dois valores da pilha, se o penúltimo valor for menor ou igual ao valor do topo insere 1 no topo da pilha, caso contrário insere 0.
2	0, 12	GTR: Retira os dois valores da pilha, se o penúltimo valor for maior que o valor do topo insere 1 no topo da pilha, caso contrário insere 0.
2	0, 13	GEQ: Retira os dois valores da pilha, se o penúltimo valor for maior ou igual ao valor do topo insere 1 no topo da pilha, caso contrário insere 0.

Fonte: Autora

APÊNDICE B — Conjunto de Instruções da Máquina Descrita por Brookshear

A Tabela 10 descreve o conjunto de instruções utilizado pela máquina descrita por Brookshear.

Tabela 10 — Conjunto de instruções da máquina descrita por Brookshear

Código	Operando	Descrição
1	RXY	LOAD: insere no registrador R o padrão de bits encontrado na posição de memória XY.
2	RXY	LOAD: insere no registrador R o valor XY.
3	RXY	STORE: Insere o padrão de bits encontrado no registrador R na posição de memória XY.
4	0RS	MOVE: insere no registrador R o padrão de bits contido no registrador S.
5	RST	ADD: soma os bits dos registradores S e T em notação de complemento de dois e armazena seu resultado no registrador R.
6	RST	ADD: soma os bits dos registradores S e T em notação de ponto flutuante e armazena seu resultado no registrador R.
7	RST	OR: executa a operação de OR sobre os bits dos registradores S e T e armazena seu resultado no registrador R.
8	RST	AND: executa a operação de AND sobre os bits dos registradores S e T e armazena seu resultado no registrador R.
9	RST	EXCLUSIVE OR: executa a operação de XOR sobre os bits dos registradores S e T e armazena seu resultado no registrador R.
A	R0X	ROTATE: rotaciona para a direita o padrão de bits no registrador R por X.
B	RXY	JUMP: pula para a instrução na posição de memória XY se o registrador R obtiver o mesmo conteúdo que o registrador 0.
C	0	HALT: termina a execução do programa.

Fonte: Autora

APÊNDICE C — Conjunto de Instruções da AllanaVM

A Tabela 11 descreve o conjunto de instruções da máquina AllanaVM obtidas através da máquina descrita por Brookshear.

Tabela 11 — Conjunto de instruções da máquina AllanaVM

Código	Operando	Descrição
1	RXY	LOAD: insere no registrador R o padrão de bits encontrado na posição de memória XY.
2	RXY	LOAD: insere no registrador R o valor XY.
3	RXY	STORE: Insere o padrão de bits encontrado no registrador R na posição de memória XY.
4	0RS	MOVE: insere no registrador R o padrão de bits contido no registrador S.
5	RST	ADD: soma os bits dos registradores S e T em notação de complemento de dois e armazena seu resultado no registrador R.
6	RST	ADD: soma os bits dos registradores S e T em notação de ponto flutuante e armazena seu resultado no registrador R.
7	RST	OR: executa a operação de OR sobre os bits dos registradores S e T e armazena seu resultado no registrador R.
8	RST	AND: executa a operação de AND sobre os bits dos registradores S e T e armazena seu resultado no registrador R.
9	RST	EXCLUSIVE OR: executa a operação de XOR sobre os bits dos registradores S e T e armazena seu resultado no registrador R.
A	R0X	ROTATE: rotaciona para a direita o padrão de bits no registrador R por X.
B	RXY	JUMP: pula para a instrução na posição de memória XY se o registrador R obtiver o mesmo conteúdo que o registrador 0.
C	0	HALT: termina a execução do programa.

Fonte: Autora

Na Tabela 12 são descritas as instruções que foram expandidas através do conjunto de instruções da P-code, como as instruções estão armazenadas na memória da AllanaVM, para executá-las utiliza-se uma instrução de JUMP.

Tabela 12 — Conjunto de instruções expandidas da P-code machine

Código	Operando	Descrição
B	00FBEA	INT: incrementa o registrador de topo da pilha e salva os parâmetros do subprograma a ser chamado.
B	00FBD2	RTN: Encerra a execução da sub-rotina.
B	00EF84	NEG: Nega o valor do registrador F_{16}.
B	00EF8E	MUL: Multiplica os valores dos registradores F_{16} e E_{16} e armazena no registrador 4_{16}.
B	00EFA6	DIV: Divide os valores dos registradores F_{16} e E_{16} e armazena no registrador 4_{16}.
B	00EF7E	ODD: Verifica o valor do registrador F_{16}, caso seja ímpar, insere 1 no registrador 4_{16}, caso contrário insere 0.
B	00EF2C	MOD: Verifica o valor do registrador F_{16}, insere o resto da divisão por 2 no registrador 4_{16}.
B	00EF70	EQL: Verifica os valores dos registradores F_{16} e $E_{16,}$ caso sejam iguais insere 1 no registrador 4_{16}, caso contrário insere 0.
B	00EF62	NEQ: Verifica os valores dos registradores F_{16} e E_{16}, caso sejam diferentes insere 1 no registrador 4_{16}, caso contrário insere 0.
B	00EF16	LSS: Verifica os valores dos registradores F_{16} e E_{16}, se o valor do registrador E_{16} for menor que o valor do registrador F_{16} insere 1 no registrador 4_{16}, caso contrário insere 0.
B	00EEF2	LEQ: Verifica os valores dos registradores F_{16} e E_{16}, se o valor do registrador E_{16} for menor ou igual ao valor do registrador F_{16} insere 1 no registrador 4_{16}, caso contrário insere 0.
B	00EFD8	GTR: Verifica os valores dos registradores F_{16} e E_{16}, se o valor do registrador E_{16} for maior que o valor do registrador F_{16} insere 1 no registrador 4_{16}, caso contrário insere 0.
B	00EECE	GEQ: Verifica os valores dos registradores F_{16} e E_{16}, se o valor do registrador E_{16} for maior ou igual ao valor do registrador F_{16} insere 1 no registrador 4_{16}, caso contrário insere 0.

Fonte: Autora

APÊNDICE D — Implementação C da AllanaVM

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<inttypes.h>
#include<math.h>
typedef struct{
    uint64_t op;        // operacao
    uint64_t opr1;      // operando1
    uint64_t opr2;      // operando2
    uint64_t opr3;      // operando3
} Tinstruction;
// List of available instructions
char * instructionString[] = { "LOAD", "LOAD", "STORE", "MOVE", "ADD", "ADD",
"OR", "AND", "EXCLS", "ROTAT", "JUMP", "HALT", "NEG", "MULT"};
// Prototypes
char * getInstructionName(uint64_t inst);
uint64_t sum_comp2(uint64_t val1, uint64_t val2);
uint64_t sum_flut(uint64_t val1, uint64_t val2);
uint64_t rot(uint64_t val1, uint64_t val2);
uint64_t trans(uint64_t val1, uint64_t val2);
uint64_t neg(uint64_t val1);
uint64_t mult(uint64_t val1, uint64_t val2);
uint64_t divide(uint64_t val1, uint64_t val2);
uint64_t mod(uint64_t val1, uint64_t val2);
Tinstruction transformInstruction(uint64_t a, uint64_t b);
int main(){
    uint64_t a = 0x00000005;
    uint64_t RAM[65536];
    uint64_t reg[65536]; // registradores
    Tinstruction i;
    printf("start                    \n");
// Aqui voce preenche as instrucoes na RAM
// RAM[0x0] = 0xOpOpr1;          RAM[0x1] = 0xOpr2Opr3;
    //GEQ
    RAM[0xeece] = 0x0000000200000000;           RAM[0xeecf] =
0x000000000000eed6;       //RAM[0xeece]: LOAD  00 eed6 // REG[0x00] <- eed6
    RAM[0xeed0] = 0x0000000300000000;           RAM[0xeed1] =
0x000000000000fc07;       //RAM[0xeed0]: STORE 00 fc07 // RAM[0xfc07] <-
REG[0x00]
    RAM[0xeed2] = 0x00000002000000fd;           RAM[0xeed3] =
0x000000000000eed8;       //RAM[0xeed2]: LOAD  fd eed8 // REG[0xfd] <- eed8
    RAM[0xeed4] = 0x0000000b00000000;           RAM[0xeed5] =
0x000000000000fbea;       //RAM[0xeed4]: JUMP  00 fbea // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbea;}
    RAM[0xeed6] = 0x0000000b00000000;           RAM[0xeed7] =
0x000000000000efd8;       //RAM[0xeed6]: JUMP  00 efd8 // if (REG[0x00] ==
REG[0x00]) {p <- 0xefd8;} // GTR
    RAM[0xeed8] = 0x0000000200000000;           RAM[0xeed9] =
0x0000000000000001;       //RAM[0xeed8]: LOAD  00 01   // REG[0x00] <- 01
    RAM[0xeeda] = 0x0000000b00000004;           RAM[0xeedb] =
0x000000000000eeee;       //RAM[0xeeda]: JUMP  04 eeee // if (REG[0x00] ==
REG[0x04]) {p <- 0xeeee;}
    RAM[0xeedc] = 0x0000000200000000;           RAM[0xeedd] =
0x000000000000eee4;       //RAM[0xeedc]: LOAD  00 eee4 // REG[0x00] <- eee4
```

```
    RAM[0xeede] = 0x0000000300000000;           RAM[0xeedf] =
0x000000000000fc07;       //RAM[0xeede]: STORE 00 fc07 // RAM[0xfc07] <-
REG[0x00]
    RAM[0xeee0] = 0x00000002000000fd;           RAM[0xeee1] =
0x000000000000eee6;       //RAM[0xeee0]: LOAD  fd eee6 // RAM[0xfd] <-
REG[0xeee6]
    RAM[0xeee2] = 0x0000000b00000000;           RAM[0xeee3] =
0x000000000000fbea;       //RAM[0xeee2]: JUMP  00 fbea // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbea;}
    RAM[0xeee4] = 0x0000000b00000000;           RAM[0xeee5] =
0x000000000000ef70;       //RAM[0xeee4]: JUMP  00 ef70 // if (REG[0x00] ==
REG[0x00]) {p <- 0xef70;} // EQL
    RAM[0xeee6] = 0x0000000200000000;           RAM[0xeee7] =
0x0000000000000001;       //RAM[0xeee6]: LOAD  00 01   // REG[0x00] <- 01
    RAM[0xeee8] = 0x0000000b00000004;           RAM[0xeee9] =
0x000000000000eeee;       //RAM[0xeee8]: JUMP  04 eeee // if (REG[0x00] ==
REG[0x04]) {p <- 0xeeee;}
    RAM[0xeeea] = 0x0000000200000004;           RAM[0xeeeb] =
0x0000000000000000;       //RAM[0xeeea]: LOAD  04 00   // REG[0x04] <- 00
    RAM[0xeeec] = 0x0000000b00000000;           RAM[0xeeed] =
0x000000000000fbd2;       //RAM[0xeeec]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    RAM[0xeeee] = 0x0000000200000004;           RAM[0xeeef] =
0x0000000000000001;       //RAM[0xeeee]: LOAD  04 01   // REG[0x04] <- 01
    RAM[0xeef0] = 0x0000000b00000000;           RAM[0xeef1] =
0x000000000000fbd2;       //RAM[0xeef0]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    //LEQ
    RAM[0xeef2] = 0x0000000200000000;           RAM[0xeef3] =
0x000000000000eefa;       //RAM[0xeef2]: LOAD  00 eefa // REG[0x00] <- eefa
    RAM[0xeef4] = 0x0000000300000000;           RAM[0xeef5] =
0x000000000000fc07;       //RAM[0xeef4]: STORE 00 fc07 // RAM[0xfc07] <-
REG[0x00]
    RAM[0xeef6] = 0x00000002000000fd;           RAM[0xeef7] =
0x000000000000eefc;       //RAM[0xeef6]: LOAD  fd eefc // REG[0xfd] <- eefc
    RAM[0xeef8] = 0x0000000b00000000;           RAM[0xeef9] =
0x000000000000fbea;       //RAM[0xeef8]: JUMP  00 fbea // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbea;}
    RAM[0xeefa] = 0x0000000b00000000;           RAM[0xeefb] =
0x000000000000efd8;       //RAM[0xeefa]: JUMP  00 efd8 // if (REG[0x00] ==
REG[0x00]) {p <- 0xefd8;} // GTR
    RAM[0xeefc] = 0x0000000200000000;           RAM[0xeefd] =
0x0000000000000000;       //RAM[0xeefc]: LOAD  00 00   // REG[0x00] <- 00
    RAM[0xeefe] = 0x0000000b00000004;           RAM[0xeeff] =
0x000000000000ef12;       //RAM[0xeefe]: JUMP  04 ef12 // if (REG[0x00] ==
REG[0x04]) {p <- 0xef12;}
    RAM[0xef00] = 0x0000000200000000;           RAM[0xef01] =
0x000000000000ef08;       //RAM[0xef00]: LOAD  00 ef08 // REG[0x00] <- ef08
    RAM[0xef02] = 0x0000000300000000;           RAM[0xef03] =
0x000000000000fc07;       //RAM[0xef02]: STORE 00 fc07 // RAM[0xfc07] <-
REG[0x00]
    RAM[0xef04] = 0x00000002000000fd;           RAM[0xef05] =
0x000000000000ef0a;       //RAM[0xef04]: LOAD  fd ef0a // REG[0xfd] <- ef0a
    RAM[0xef06] = 0x0000000b00000000;           RAM[0xef07] =
0x000000000000fbea;       //RAM[0xef06]: JUMP  00 fbea // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbea;}
    RAM[0xef08] = 0x0000000b00000000;           RAM[0xef09] =
0x000000000000ef70;       //RAM[0xef08]: JUMP  00 ef70 // if (REG[0x00] ==
REG[0x00]) {p <- 0xef70;} // EQL
    RAM[0xef0a] = 0x0000000200000000;           RAM[0xef0b] =
```

```
0x0000000000000001;       //RAM[0xef0a]: LOAD  00 01   // REG[0x00] <- 01
    RAM[0xef0c] = 0x0000000b00000004;           RAM[0xef0d] =
0x000000000000ef12;       //RAM[0xef0c]: JUMP  04 ef12 // if (REG[0x00] ==
REG[0x04]) {p <- 0xef12;}
    RAM[0xef0e] = 0x0000000200000004;           RAM[0xef0f] =
0x0000000000000000;       //RAM[0xef0e]: LOAD  04 00   // REG[0x04] <- 00
    RAM[0xef10] = 0x0000000b00000000;           RAM[0xef11] =
0x000000000000fbd2;       //RAM[0xef10]: JUMP  00 fdb2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfdb2;}
    RAM[0xef12] = 0x0000000200000004;           RAM[0xef13] =
0x0000000000000001;       //RAM[0xef12]: LOAD  04 01   // REG[0x04] <- 01
    RAM[0xef14] = 0x0000000b00000000;           RAM[0xef15] =
0x000000000000fbd2;       //RAM[0xef14]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    //LSS
    RAM[0xef16] = 0x0000000200000000;           RAM[0xef17] =
0x000000000000ef1e;       //RAM[0xef16]: LOAD  00 ef1e // REG[0x00] <- 0xef1e
    RAM[0xef18] = 0x0000000300000000;           RAM[0xef19] =
0x000000000000fc07;       //RAM[0xef18]: STORE 00 fc07 // RAM[0xfc07] <-
REG[0x00]
    RAM[0xef1a] = 0x00000002000000fd;           RAM[0xef1b] =
0x000000000000ef20;       //RAM[0xef1a]: LOAD  fd ef20 // REG[0xfd] <- ef20
    RAM[0xef1c] = 0x0000000b00000000;           RAM[0xef1d] =
0x000000000000fbea;       //RAM[0xef1c]: JUMP  00 fbea // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbea;}
    RAM[0xef1e] = 0x0000000b00000000;           RAM[0xef1f] =
0x000000000000eece;       //RAM[0xef1e]: JUMP  00 eece // if (REG[0x00] ==
REG[0x00]) {p <- 0xeece;} // GEQ
    RAM[0xef20] = 0x0000000200000000;           RAM[0xef21] =
0x0000000000000000;       //RAM[0xef20]: LOAD  00 00   // REG[0x00] <- 00
    RAM[0xef22] = 0x0000000b00000004;           RAM[0xef23] =
0x000000000000ef28;       //RAM[0xef22]: JUMP  04 ef28 // if (REG[0x00] ==
REG[0x04]) {p <- 0xef28;}
    RAM[0xef24] = 0x0000000200000004;           RAM[0xef25] =
0x0000000000000000;       //RAM[0xef24]: LOAD  04 00   // REG[0x04] <- 00
    RAM[0xef26] = 0x0000000b00000000;           RAM[0xef27] =
0x000000000000fbd2;       //RAM[0xef26]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    RAM[0xef28] = 0x0000000200000004;           RAM[0xef29] =
0x0000000000000001;       //RAM[0xef28]: LOAD  04 01   // REG[0x04] <- 01
    RAM[0xef2a] = 0x0000000b00000000;           RAM[0xef2b] =
0x000000000000fbd2;       //RAM[0xef2a]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    //MOD
    RAM[0xef2c] = 0x000000030000000f;          RAM[0xef2d] = 0x000000000000fbd0;
//RAM[0xef2c]: STORE 0f fbd0 // RAM[0xf6] <- REG[0x0f]
    RAM[0xef2e] = 0x000000010000000c;          RAM[0xef2f] = 0x000000000000fbd0;
//RAM[0xef2e]: LOAD  0c fbd0 // REG[0x0c] <- RAM[0xf6]
    RAM[0xef30] = 0x0000000200000000;          RAM[0xef31] = 0x000000000000ef38;
//RAM[0xef30]: LOAD  00 ef38 // REG[0x00] <- ef38
    RAM[0xef32] = 0x0000000300000000;          RAM[0xef33] = 0x000000000000fc07;
//RAM[0xef32]: STORE 00 fc07 // RAM[0xfc07] <- REG[0x00]
    RAM[0xef34] = 0x00000002000000fd;          RAM[0xef35] = 0x000000000000ef3a;
//RAM[0xef34]: LOAD  fd ef3a // REG[0xfd] <- ef3a
    RAM[0xef36] = 0x0000000b00000000;          RAM[0xef37] = 0x000000000000fbea;
//RAM[0xef36]: JUMP  00 fbea // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;}
    RAM[0xef38] = 0x0000000b00000000;          RAM[0xef39] = 0x000000000000efa6;
//RAM[0xef38]: JUMP  00 efa6 // if (REG[0x00] == REG[0x00]) {p <- 0xefa6;} //
DIV
    RAM[0xef3a] = 0x000000030000000c;          RAM[0xef3b] = 0x000000000000fbd0;
```

```
//RAM[0xef3a]: STORE 0c fbd0 // RAM[0xf6] <- REG[0x0c]
    RAM[0xef3c] = 0x000000010000000f;          RAM[0xef3d] = 0x000000000000fbd0;
//RAM[0xef3c]: LOAD  0f fbd0 // REG[0x0f] <- RAM[0xf6]
    RAM[0xef3e] = 0x000000030000000e;          RAM[0xef3f] = 0x000000000000fbcf;
//RAM[0xef3e]: STORE 0e fbcf // RAM[0xf5] <- REG[0x0e]
    RAM[0xef40] = 0x0000000300000004;          RAM[0xef41] = 0x000000000000fbd1;
//RAM[0xef40]: STORE 04 fbd1 // RAM[0xf7] <- REG[0x04]
    RAM[0xef42] = 0x000000010000000e;          RAM[0xef43] = 0x000000000000fbd1;
//RAM[0xef42]: LOAD  0e fbd1 // REG[0x0e] <- RAM[0xf7]
    RAM[0xef44] = 0x0000000200000000;          RAM[0xef45] = 0x000000000000ef4c;
//RAM[0xef44]: LOAD  00 ef4c // REG[0x00] <- ef4c
    RAM[0xef46] = 0x0000000300000000;          RAM[0xef47] = 0x000000000000fc07;
//RAM[0xef46]: STORE 00 fc07 // RAM[0xfc07] <- REG[0x00]
    RAM[0xef48] = 0x00000002000000fd;          RAM[0xef49] = 0x000000000000ef4e;
//RAM[0xef48]: LOAD  fd ef4e // REG[0xfd] <- ef4e
    RAM[0xef4a] = 0x0000000b00000000;          RAM[0xef4b] = 0x000000000000fbea;
//RAM[0xef4a]: JUMP  00 fbea // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;}
    RAM[0xef4c] = 0x0000000b00000000;          RAM[0xef4d] = 0x000000000000ef8e;
//RAM[0xef4c]: JUMP  00 ef8e // if (REG[0x00] == REG[0x00]) {p <- 0xef8e;} //
MULT
    RAM[0xef4e] = 0x0000000300000004;          RAM[0xef4f] = 0x000000000000fbd1;
//RAM[0xef4e]: STORE 04 fbd1 // RAM[0xf7] <- REG[0x04]
    RAM[0xef50] = 0x000000010000000f;          RAM[0xef51] = 0x000000000000fbd1;
//RAM[0xef50]: LOAD  0f fbd1 // REG[0x0f] <- RAM[0xf7]
    RAM[0xef52] = 0x0000000200000000;          RAM[0xef53] = 0x000000000000ef5a;
//RAM[0xef52]: LOAD  00 ef5a // REG[0x00] <- ef5a
    RAM[0xef54] = 0x0000000300000000;          RAM[0xef55] = 0x000000000000fc07;
//RAM[0xef54]: STORE 00 fc07 // RAM[0xfc07] <- REG[0x00]
    RAM[0xef56] = 0x00000002000000fd;          RAM[0xef57] = 0x000000000000ef5c;
//RAM[0xef56]: LOAD  fd ef5c // REG[0xfd] <- ef5c
    RAM[0xef58] = 0x0000000b00000000;          RAM[0xef59] = 0x000000000000fbea;
//RAM[0xef58]: JUMP  00 fbea // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;}
    RAM[0xef5a] = 0x0000000b00000000;          RAM[0xef5b] = 0x000000000000ef84;
//RAM[0xef5a]: JUMP  00 ef84 // if (REG[0x00] == REG[0x00]) {p <- 0xef84;} //
NEG
    RAM[0xef5c] = 0x000000010000000f;          RAM[0xef5d] = 0x000000000000fbcf;
//RAM[0xef5c]: LOAD  0f fbcf // REG[0x0f] <- RAM[0xf5]
    RAM[0xef5e] = 0x0000000500000004;          RAM[0xef5f] = 0x0000000f00000004;
//RAM[0xef5e]: ADD   04 f4   // REG[0x04] <- REG[0x0f] + REG[0x04]
    RAM[0xef60] = 0x0000000b00000000;          RAM[0xef61] = 0x000000000000fbd2;
//RAM[0xef60]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x0f]) {p <- 0xfbd2;}
    //NEQ
    RAM[0xef62] = 0x000000030000000f;           RAM[0xef63] =
0x000000000000fbd0;      //RAM[0xef62]: STORE 0f fbd0 // RAM[0xf6] <- REG[0x0f]
    RAM[0xef64] = 0x0000000100000000;           RAM[0xef65] =
0x000000000000fbd0;      //RAM[0xef64]: LOAD  00 fbd0 // REG[0x00] <- RAM[0xf6]
    RAM[0xef66] = 0x0000000b0000000e;           RAM[0xef67] =
0x000000000000ef68;      //RAM[0xef66]: JUMP  0e ef68 // if (REG[0x00] ==
REG[0x0e]) {p <- 0xef68;}
    RAM[0xef68] = 0x0000000200000004;           RAM[0xef69] =
0x0000000000000001;      //RAM[0xef68]: LOAD  04 01   // REG[0x04] <- 01
    RAM[0xef6a] = 0x0000000b00000000;           RAM[0xef6b] =
0x000000000000fbd2;      //RAM[0xef6a]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    RAM[0xef6c] = 0x0000000200000004;           RAM[0xef6d] =
0x0000000000000000;      //RAM[0xef6c]: LOAD  04 00   // REG[0x04] <- 00
    RAM[0xef6e] = 0x0000000b00000000;           RAM[0xef6f] =
0x000000000000fbd2;      //RAM[0xef6e]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    //EQL
```

```
    RAM[0xef70] = 0x000000030000000f;           RAM[0xef71] =
0x000000000000fbd0;      //RAM[0xef70]: STORE 0f fbd0 // RAM[0xf6] <- REG[0x0f]
    RAM[0xef72] = 0x0000000100000000;           RAM[0xef73] =
0x000000000000fbd0;      //RAM[0xef72]: LOAD  00 fbd0 // REG[0x00] <- RAM[0xf6]
    RAM[0xef74] = 0x0000000b0000000e;           RAM[0xef75] =
0x000000000000ef7a;      //RAM[0xef74]: JUMP  0e ef7a // if (REG[0x00] ==
REG[0x0e]) {p <- 0xef7a;}
    RAM[0xef76] = 0x0000000200000004;           RAM[0xef77] =
0x0000000000000000;      //RAM[0xef76]: LOAD  04 00   // REG[0x04] <- 00
    RAM[0xef78] = 0x0000000b00000000;           RAM[0xef79] =
0x000000000000fbd2;      //RAM[0xef78]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    RAM[0xef7a] = 0x0000000200000004;           RAM[0xef7b] =
0x0000000000000001;      //RAM[0xef7a]: LOAD  04 01   // REG[0x04] <- 01
    RAM[0xef7c] = 0x0000000b00000000;           RAM[0xef7d] =
0x000000000000fbd2;      //RAM[0xef7c]: JUMP  00 fbd2 // if (REG[0x00] ==
REG[0x00]) {p <- 0xfbd2;}
    //ODD
    RAM[0xef7e] = 0x000000020000000e;           RAM[0xef7f] = 0x0000000000000001;
//RAM[0xef7e]: LOAD  0e 01   // REG[0x0e] <- 01
    RAM[0xef80] = 0x0000000800000004;           RAM[0xef81] = 0x0000000e0000000f;
//RAM[0xef80]: AND   04 ef   // REG[0x04] <- REG[0x0e] && REG[0x0f]
    RAM[0xef82] = 0x0000000b00000000;           RAM[0xef83] = 0x000000000000fbd2;
//RAM[0xef82]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;}
    //NEG
    RAM[0xef84] = 0x000000020000000e;           RAM[0xef85] = 0x0000000000000001;
//RAM[0xef84]: LOAD  0e 01   // REG[0x0e] <- 01
    RAM[0xef86] = 0x0000000200000002;           RAM[0xef87] = 0xffffffffffffffff;
//RAM[0xef86]: LOAD  02 -1   // REG[0x02] <- -1
    RAM[0xef88] = 0x0000000900000004;           RAM[0xef89] = 0x000000020000000f;
//RAM[0xef88]: XOR   04 2f   // REG[0x04] <- REG[0x02] XOR REG[0x0f]
    RAM[0xef8a] = 0x0000000500000004;           RAM[0xef8b] = 0x000000040000000e;
//RAM[0xef8a]: ADD   04 4e   // REG[0x04] <- REG[0x04] + REG[0x0e]
    RAM[0xef8c] = 0x0000000b00000000;           RAM[0xef8d] = 0x000000000000fbd2;
//RAM[0xef8c]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x0f]) {p <- 0xfbd2;}
    //MULT
    RAM[0xef8e] = 0x000000030000000f;           RAM[0xef8f] = 0x000000000000fbd1;
//RAM[0xef8e]: STORE 0f fbd1 // RAM[0xf7] <- REG[0x0f]
    RAM[0xef90] = 0x000000030000000e;           RAM[0xef91] = 0x000000000000fbd0;
//RAM[0xef90]: STORE 0e fbd0 // RAM[0xf6] <- REG[0x0e]
    RAM[0xef92] = 0x0000000200000002;           RAM[0xef93] = 0x0000000000000001;
//RAM[0xef92]: LOAD  02 01   // REG[0x02] <- 01
    RAM[0xef94] = 0x0000000200000003;           RAM[0xef95] = 0x0000000000000001;
//RAM[0xef94]: LOAD  03 01   // REG[0x03] <- 01
    RAM[0xef96] = 0x0000000100000004;           RAM[0xef97] = 0x000000000000fbd0;
//RAM[0xef96]: LOAD  04 fbd0 // REG[0x04] <- RAM[0xf6]
    RAM[0xef98] = 0x0000000100000000;           RAM[0xef99] = 0x000000000000fbd1;
//RAM[0xef98]: LOAD  00 fbd1 // REG[0x00] <- RAM[0xf7]
    RAM[0xef9a] = 0x0000000100000001;           RAM[0xef9b] = 0x000000000000fbd0;
//RAM[0xef9a]: LOAD  01 fbd0 // REG[0x01] <- RAM[0xf6]
    RAM[0xef9c] = 0x0000000b00000002;           RAM[0xef9d] = 0x000000000000fbd2;
//RAM[0xef9c]: JUMP  02 fbd2 // if (REG[0x00] == REG[0x02]) {p <- 0xfbd2;}
    RAM[0xef9e] = 0x0000000500000004;           RAM[0xef9f] = 0x0000000400000001;
//RAM[0xef9e]: ADD   04 41   // REG[0x04] <- REG[0x04] + REG[0x01]
    RAM[0xefa0] = 0x0000000500000002;           RAM[0xefa1] = 0x0000000200000003;
//RAM[0xefa0]: ADD   02 23   // REG[0x02] <- REG[0x02] + REG[0x03]
    RAM[0xefa2] = 0x0000000200000000;           RAM[0xefa3] = 0x0000000000000001;
//RAM[0xefa2]: LOAD  00 01   // REG[0x00] <- 01
    RAM[0xefa4] = 0x0000000b00000003;           RAM[0xefa5] = 0x000000000000ef98;
//RAM[0xefa4]: JUMP  03 ef98 // if (REG[0x00] == REG[0x03]) {p <- 0xef98;}
```

```
    //DIV
    RAM[0xefa6] = 0x000000030000000f;          RAM[0xefa7] = 0x000000000000fbd0;
//RAM[0xefa6]: STORE 0f fbd0 // RAM[0xf6] <- REG[0x0f]
    RAM[0xefa8] = 0x000000030000000e;          RAM[0xefa9] = 0x000000000000fbd1;
//RAM[0xefa8]: STORE 0e fbd1 // RAM[0xf7] <- REG[0x0e]
    RAM[0xefaa] = 0x0000000200000001;          RAM[0xefab] = 0x0000000000000001;
//RAM[0xefaa]: LOAD  01 01   // REG[0x01] <- 01
    RAM[0xefac] = 0x0000000200000003;          RAM[0xefad] = 0x0000000000000001;
//RAM[0xefac]: LOAD  03 01   // REG[0x03] <- 01
    RAM[0xefae] = 0x0000000100000002;          RAM[0xefaf] = 0x000000000000fbd0;
//RAM[0xefae]: LOAD  02 fbd0 // REG[0x02] <- RAM[0xf6]
    RAM[0xefb0] = 0x000000050000000f;          RAM[0xefb1] = 0x0000000f00000002;
//RAM[0xefb0]: ADD   0f f2   // REG[0x0f] <- REG[0x0f] + REG[0x02]
    RAM[0xefb2] = 0x0000000200000000;          RAM[0xefb3] = 0x000000000000efba;
//RAM[0xefb2]: LOAD  00 efba // REG[0x00] <- efba
    RAM[0xefb4] = 0x0000000300000000;          RAM[0xefb5] = 0x000000000000fc07;
//RAM[0xefb4]: STORE 00 fc07 // RAM[0xfc07] <- REG[0x00]
    RAM[0xefb6] = 0x00000002000000fd;          RAM[0xefb7] = 0x000000000000efbc;
//RAM[0xefb6]: LOAD  fd efbc // REG[0xfd] <- efbc
    RAM[0xefb8] = 0x0000000b00000000;          RAM[0xefb9] = 0x000000000000fbea;
//RAM[0xefb8]: JUMP  00 fbea // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;}
    RAM[0xefba] = 0x0000000b00000000;          RAM[0xefbb] = 0x000000000000efd8;
//RAM[0xefba]: JUMP  00 efd8 // if (REG[0x00] == REG[0x00]) {p <- 0xefd8;} //
GTR
    RAM[0xefbc] = 0x0000000200000000;          RAM[0xefbd] = 0x0000000000000000;
//RAM[0xefbc]: LOAD  00 00   // REG[0x00] <- 00
    RAM[0xefbe] = 0x0000000b00000004;          RAM[0xefbf] = 0x000000000000efc4;
//RAM[0xefbe]: JUMP  04 efc4 // if (REG[0x00] == REG[0x04]) {p <- 0xefc4;}
    RAM[0xefc0] = 0x0000000500000003;          RAM[0xefc1] = 0x0000000300000001;
//RAM[0xefc0]: ADD   03 31   // REG[0x03] <- REG[0x03] + REG[0x01]
    RAM[0xefc2] = 0x0000000b00000000;          RAM[0xefc3] = 0x000000000000efae;
//RAM[0xefc2]: JUMP  00 efae // if (REG[0x00] == REG[0x00]) {p <- 0xefa2;}
    RAM[0xefc4] = 0x0000000100000000;          RAM[0xefc5] = 0x000000000000fbd1;
//RAM[0xefc4]: LOAD  00 fbd1 // REG[0x00] <- RAM[0xf7]
    RAM[0xefc6] = 0x0000000b0000000f;          RAM[0xefc7] = 0x000000000000efce;
//RAM[0xefc6]: JUMP  0f efce // if (REG[0x00] == REG[0x0f]) {p <- 0xefce;}
    RAM[0xefc8] = 0x0000000300000003;          RAM[0xefc9] = 0x000000000000fbd1;
//RAM[0xefc8]: STORE 03 fbd1 // RAM[0xf7] <- REG[0x03]
    RAM[0xefca] = 0x0000000100000004;          RAM[0xefcb] = 0x000000000000fbd1;
//RAM[0xefca]: LOAD  04 fbd1 // REG[0x04] <- RAM[0xf7]
    RAM[0xefcc] = 0x0000000b00000000;          RAM[0xefcd] = 0x000000000000fbd2;
//RAM[0xefcc]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;}
    RAM[0xefce] = 0x0000000500000003;          RAM[0xefcf] = 0x0000000300000001;
//RAM[0xefce]: ADD   03 31   // REG[0x03] <- REG[0x03] + REG[0x01]
    RAM[0xefd0] = 0x0000000300000003;          RAM[0xefd1] = 0x000000000000fbd1;
//RAM[0xefd0]: STORE 03 fbd1 // RAM[0xf7] <- REG[0x03]
    RAM[0xefd2] = 0x0000000100000004;          RAM[0xefd3] = 0x000000000000fbd1;
//RAM[0xefd2]: LOAD  04 fbd1 // REG[0x04] <- RAM[0xf7]
    RAM[0xefd4] = 0x0000000b00000000;          RAM[0xefd5] = 0x000000000000fbd2;
//RAM[0xefd4]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x0f]) {p <- 0xfbd2;}
    //GTR
    RAM[0xefd8] = 0x000000030000000f;          RAM[0xefd9] = 0x000000000000fbcf;
//RAM[0xefd8]: STORE 0f fbcf // RAM[0xf5] <- REG[0x0f]
    RAM[0xefda] = 0x0000000100000000;          RAM[0xefdb] = 0x000000000000fbcf;
//RAM[0xefda]: LOAD  00 fbcf // REG[0x00] <- RAM[0xf5]
    RAM[0xefdc] = 0x000000030000000e;          RAM[0xefdd] = 0x000000000000fbd1;
//RAM[0xefdc]: STORE 0e fbd1 // RAM[0xf7] <- REG[0x0e]
    RAM[0xefde] = 0x0000000b0000000e;          RAM[0xefdf] = 0x000000000000effa;
//RAM[0xefde]: JUMP  0e effa // if (REG[0x00] == REG[0x0e]) {p <- 0xeffa;}
    RAM[0xefe0] = 0x0000000200000000;          RAM[0xefe1] = 0x0000000000000001;
```

```
//RAM[0xefe0]: LOAD  00 01   // REG[0x00] <- 01
    RAM[0xefe2] = 0x0000000a0000000f;          RAM[0xefe3] = 0x0000000f0000001f;
//RAM[0xefe2]: ROT   0f 0f1f // REG[0x0f] <- REG[0x0f] >> 1f
    RAM[0xefe4] = 0x0000000a0000000e;          RAM[0xefe5] = 0x0000000e0000001f;
//RAM[0xefe4]: ROT   0e 0e1f // REG[0x0e] <- REG[0x0e] >> 1f
    RAM[0xefe6] = 0x0000000800000002;          RAM[0xefe7] = 0x000000000000000f;
//RAM[0xefe6]: AND   02 0f   // REG[0x02] <- REG[0x00] && REG[0x0f]
    RAM[0xefe8] = 0x0000000800000004;          RAM[0xefe9] = 0x000000000000000e;
//RAM[0xefe8]: AND   04 0e   // REG[0x04] <- REG[0x00] && REG[0x0e]
    RAM[0xefea] = 0x000000090000000d;          RAM[0xefeb] = 0x0000000200000004;
//RAM[0xefea]: XOR   0d 24   // REG[0x0d] <- REG[0x02] XOR REG[0x04]
    RAM[0xefec] = 0x0000000b0000000d;          RAM[0xefed] = 0x000000000000eff0;
//RAM[0xefec]: JUMP  0d eff0 // if (REG[0x00] == REG[0x02]) {p <- 0xeff0;}
    RAM[0xefee] = 0x0000000b00000000;          RAM[0xefef] = 0x000000000000efe2;
//RAM[0xefee]: JUMP  00 efe2 // if (REG[0x00] == REG[0x00]) {p <- 0xefe2;}
    RAM[0xeff0] = 0x0000000b00000002;          RAM[0xeff1] = 0x000000000000effa;
//RAM[0xeff0]: JUMP  02 efea // if (REG[0x00] == REG[0x02]) {p <- 0xefea;}
    RAM[0xeff2] = 0x0000000200000004;          RAM[0xeff3] = 0x0000000000000001;
//RAM[0xeff2]: LOAD  04 01   // REG[0x04] <- 01
    RAM[0xeff4] = 0x000000010000000f;          RAM[0xeff5] = 0x000000000000fbcf;
//RAM[0xeff4]: LOAD  0f fbcf // REG[0x0f] <- RAM[0xf5]
    RAM[0xeff6] = 0x000000010000000e;          RAM[0xeff7] = 0x000000000000fbd1;
//RAM[0xeff6]: LOAD  0e fbd1 // REG[0x0e] <- RAM[0xf7]
    RAM[0xeff8] = 0x0000000b00000000;          RAM[0xeff9] = 0x000000000000fbd2;
//RAM[0xeff8]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;}
    RAM[0xeffa] = 0x0000000200000004;          RAM[0xeffb] = 0x0000000000000000;
//RAM[0xeffa]: LOAD  04 00   // REG[0x04] <- 00
    RAM[0xeffc] = 0x000000010000000f;          RAM[0xeffd] = 0x000000000000fbcf;
//RAM[0xeffc]: LOAD  0f fbcf // REG[0x0f] <- RAM[0xf5]
    RAM[0xeffe] = 0x000000010000000e;          RAM[0xefff] = 0x000000000000fbd1;
//RAM[0xeffe]: LOAD  0e fbd1 // REG[0x0e] <- RAM[0xf7]
    RAM[0xf000] = 0x0000000b00000000;          RAM[0xf001] = 0x000000000000fbd2;
//RAM[0xf000]: JUMP  00 fbd2 // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;}
    //RET
    RAM[0xfbd2] = 0x0000000200000000;          RAM[0xfbd3] = 0xffffffffffffffff;
// RAM[0xfbd2]: LOAD  00 -1   // REG[0x00] <- -1
    RAM[0xfbd4] = 0x0000000500000000;          RAM[0xfbd5] = 0x000000fe00000000;
// RAM[0xfbd4]: ADD   00 fe00 // REG[0x00] <- REG[0xfe] + REG[0x00]
    RAM[0xfbd6] = 0x0000000300000000;          RAM[0xfbd7] = 0x000000000000fbe9;
// RAM[0xfbd6]: STORE 00 fbe9 // RAM[0xfbe9] <- REG[0x00]
    RAM[0xfbd8] = 0x00000003000000ff;          RAM[0xfbd9] = 0x000000000000fbdb;
// RAM[0xfbd8]: STORE ff fbdb // RAM[0xfbdb] <- REG[0xff]
    RAM[0xfbda] = 0x000000010000000f;
// RAM[0xfbda]: LOAD  0f      // REG[0x0f] <- RAM[]
    RAM[0xfbdc] = 0x00000003000000fc;          RAM[0xfbdd] = 0x000000000000fbdf;
// RAM[0xfbdc]: STORE fc fbdf // RAM[0xfbdf] <- REG[0xfc]
    RAM[0xfbde] = 0x0000000300000004;
// RAM[0xfbde]: STORE 04      // RAM[] <- REG[0x04]
    RAM[0xfbe0] = 0x00000005000000fe;          RAM[0xfbe1] = 0x000000fe000000fb;
// RAM[0xfbe0]: ADD   fe fefb // REG[0xfe] <- REG[0xfe] + REG[0xfb] // base
    RAM[0xfbe2] = 0x00000005000000f1;          RAM[0xfbe3] = 0x000000f1000000fb;
// RAM[0xfbe2]: ADD   f1 f1fb // REG[0xf1] <- REG[0xf1] + REG[0xfb] // parametro
    RAM[0xfbe4] = 0x00000005000000ff;          RAM[0xfbe5] = 0x000000ff000000fb;
// RAM[0xfbe4]: ADD   ff fffb // REG[0xff] <- REG[0xff] + REG[0xfb] // parametro
    RAM[0xfbe6] = 0x00000005000000fc;          RAM[0xfbe7] = 0x000000fc000000fb;
// RAM[0xfbe6]: ADD   fc fcfb // REG[0xfc] <- REG[0xfc] + REG[0xfb] // resultado
    RAM[0xfbe8] = 0x0000000b00000000;
// RAM[0xfbe8]: JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    //INT
    RAM[0xfbea] = 0x0000000200000000;          RAM[0xfbeb] = 0x0000000000000000;
```

```
// RAM[0xfbea]: LOAD  00 00   // REG[0xf0] <- 00
    RAM[0xfbec] = 0x00000005000000fe;          RAM[0xfbed] = 0x000000fe000000fa;
// RAM[0xfbea]: ADD   fe fefa // REG[0xfe] <- REG[0xfe] + REG[0xfa] // base
    RAM[0xfbee] = 0x00000005000000f1;          RAM[0xfbef] = 0x000000f1000000fa;
// RAM[0xfbea]: ADD   f1 f1fa // REG[0xf1] <- REG[0xf1] + REG[0xfa] // parametro
    RAM[0xfbf0] = 0x00000005000000ff;          RAM[0xfbf1] = 0x000000ff000000fa;
// RAM[0xfbea]: ADD   ff fffa // REG[0xff] <- REG[0xff] + REG[0xfa] // parametro
    RAM[0xfbf2] = 0x00000005000000fc;          RAM[0xfbf3] = 0x000000fc000000fa;
// RAM[0xfbea]: ADD   fc fcfa // REG[0xfc] <- REG[0xfc] + REG[0xfa] // resultado
    RAM[0xfbf4] = 0x00000003000000fe;          RAM[0xfbf5] = 0x000000000000fc03;
// RAM[0xfbea]: STORE fe fc03 // RAM[0xfc03] <- REG[0x04] // return
    RAM[0xfbf6] = 0x00000003000000f1;          RAM[0xfbf7] = 0x000000000000fc01;
// RAM[0xfbea]: STORE f1 fc01 // RAM[0xfc01] <- REG[0x04] // parametro
    RAM[0xfbf8] = 0x00000003000000ff;          RAM[0xfbf9] = 0x000000000000fbff;
// RAM[0xfbea]: STORE ff fbff // RAM[0xfbff] <- REG[0x04] // parametro
    RAM[0xfbfa] = 0x00000003000000fc;          RAM[0xfbfb] = 0x000000000000fc05;
// RAM[0xfbea]: STORE fc fc05 // RAM[0xfc05] <- REG[0x04] // resultado
    RAM[0xfbfc] = 0x0000000b00000000;          RAM[0xfbfd] = 0x000000000000fbfe;
// RAM[0xfbea]: JUMP  00 fbfe // if (REG[0x00] == REG[0x00]) {p <- 0xfbfe;}
    //Save STACK
    RAM[0xfbfe] =
0x000000030000000f;                                          // RAM[0xfbfe]:
STORE 0f      // RAM[0x] <- REG[0x0f]
    RAM[0xfc00] =
0x000000030000000e;                                          // RAM[0xfc00]:
STORE 0e      // RAM[0x] <- REG[0x0e]
    RAM[0xfc02] =
0x00000003000000fd;                                          // RAM[0xfc02]:
STORE fd      // RAM[0x] <- REG[0xfd]
    RAM[0xfc04] =
0x0000000300000000;                                          // RAM[0xfc04]:
STORE 00      // RAM[0x] <- REG[0x00]
    RAM[0xfc06] =
0x0000000b00000000;                                          // RAM[0xfc06]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    //STACK
    RAM[0xfc08] =
0x0000000b00000000;                                          // RAM[0xfc08]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc0d] =
0x0000000b00000000;                                          // RAM[0xfc0d]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc12] =
0x0000000b00000000;                                          // RAM[0xfc12]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc17] =
0x0000000b00000000;                                          // RAM[0xfc17]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc1c] =
0x0000000b00000000;                                          // RAM[0xfc1c]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc21] =
0x0000000b00000000;                                          // RAM[0xfc21]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc26] =
0x0000000b00000000;                                          // RAM[0xfc26]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc2b] =
0x0000000b00000000;                                          // RAM[0xfc2b]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
```

```
    RAM[0xfc30] =
0x0000000b00000000;                                                 // RAM[0xfc30]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc35] =
0x0000000b00000000;                                                 // RAM[0xfc35]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc3a] =
0x0000000b00000000;                                                 // RAM[0xfc3a]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc3f] =
0x0000000b00000000;                                                 // RAM[0xfc3f]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc44] =
0x0000000b00000000;                                                 // RAM[0xfc44]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc49] =
0x0000000b00000000;                                                 // RAM[0xfc49]:
JUMP  00      // if (REG[0x00] == REG[0x00]) {p <- 0x;}
    RAM[0xfc4e] = 0x0000000b00000000;
    RAM[0xfc53] = 0x0000000b00000000;
    RAM[0xfc58] = 0x0000000b00000000;
    RAM[0xfc5d] = 0x0000000b00000000;
    RAM[0xfc62] = 0x0000000b00000000;
    RAM[0xfc67] = 0x0000000b00000000;
    RAM[0xfc6c] = 0x0000000b00000000;
    RAM[0xfc71] = 0x0000000b00000000;
    RAM[0xfc76] = 0x0000000b00000000;
    RAM[0xfc7b] = 0x0000000b00000000;
    RAM[0xfc80] = 0x0000000b00000000;
    RAM[0xfc85] = 0x0000000b00000000;
    RAM[0xfc8a] = 0x0000000b00000000;
    RAM[0xfc8f] = 0x0000000b00000000;
    RAM[0xfc94] = 0x0000000b00000000;
    RAM[0xfc99] = 0x0000000b00000000;
    RAM[0xfc9e] = 0x0000000b00000000;
    RAM[0xfca3] = 0x0000000b00000000;
    RAM[0xfca8] = 0x0000000b00000000;
    RAM[0xfcad] = 0x0000000b00000000;
    RAM[0xfcb2] = 0x0000000b00000000;
    RAM[0xfcb7] = 0x0000000b00000000;
    RAM[0xfcbc] = 0x0000000b00000000;
    RAM[0xfcc1] = 0x0000000b00000000;
    RAM[0xfcc6] = 0x0000000b00000000;
    RAM[0xfccb] = 0x0000000b00000000;
    RAM[0xfcd0] = 0x0000000b00000000;
    RAM[0xfcd5] = 0x0000000b00000000;
    RAM[0xfcda] = 0x0000000b00000000;
    RAM[0xfcdf] = 0x0000000b00000000;
    RAM[0xfce4] = 0x0000000b00000000;
    RAM[0xfce9] = 0x0000000b00000000;
    RAM[0xfcee] = 0x0000000b00000000;
    RAM[0xfcf3] = 0x0000000b00000000;
    RAM[0xfcf8] = 0x0000000b00000000;
    RAM[0xfcfd] = 0x0000000b00000000;
    RAM[0xfd02] = 0x0000000b00000000;
    RAM[0xfd07] = 0x0000000b00000000;
    RAM[0xfd0c] = 0x0000000b00000000;
    RAM[0xfd11] = 0x0000000b00000000;
    RAM[0xfd16] = 0x0000000b00000000;
```

```
RAM[0xfd1b] = 0x0000000b00000000;
RAM[0xfd20] = 0x0000000b00000000;
RAM[0xfd25] = 0x0000000b00000000;
RAM[0xfd2a] = 0x0000000b00000000;
RAM[0xfd2f] = 0x0000000b00000000;
RAM[0xfd34] = 0x0000000b00000000;
RAM[0xfd39] = 0x0000000b00000000;
RAM[0xfd3e] = 0x0000000b00000000;
RAM[0xfd43] = 0x0000000b00000000;
RAM[0xfd48] = 0x0000000b00000000;
RAM[0xfd4d] = 0x0000000b00000000;
RAM[0xfd52] = 0x0000000b00000000;
RAM[0xfd57] = 0x0000000b00000000;
RAM[0xfd5c] = 0x0000000b00000000;
RAM[0xfd61] = 0x0000000b00000000;
RAM[0xfd66] = 0x0000000b00000000;
RAM[0xfd6b] = 0x0000000b00000000;
RAM[0xfd70] = 0x0000000b00000000;
RAM[0xfd75] = 0x0000000b00000000;
RAM[0xfd7a] = 0x0000000b00000000;
RAM[0xfd7f] = 0x0000000b00000000;
RAM[0xfd84] = 0x0000000b00000000;
RAM[0xfd89] = 0x0000000b00000000;
RAM[0xfd8e] = 0x0000000b00000000;
RAM[0xfd93] = 0x0000000b00000000;
RAM[0xfd98] = 0x0000000b00000000;
RAM[0xfd9d] = 0x0000000b00000000;
RAM[0xfda2] = 0x0000000b00000000;
RAM[0xfda7] = 0x0000000b00000000;
RAM[0xfdac] = 0x0000000b00000000;
RAM[0xfdb1] = 0x0000000b00000000;
RAM[0xfdb6] = 0x0000000b00000000;
RAM[0xfdbb] = 0x0000000b00000000;
RAM[0xfdc0] = 0x0000000b00000000;
RAM[0xfdc5] = 0x0000000b00000000;
RAM[0xfdca] = 0x0000000b00000000;
RAM[0xfdcf] = 0x0000000b00000000;
RAM[0xfdd4] = 0x0000000b00000000;
RAM[0xfdd9] = 0x0000000b00000000;
RAM[0xfdde] = 0x0000000b00000000;
RAM[0xfde3] = 0x0000000b00000000;
RAM[0xfde8] = 0x0000000b00000000;
RAM[0xfded] = 0x0000000b00000000;
RAM[0xfdf2] = 0x0000000b00000000;
RAM[0xfdf7] = 0x0000000b00000000;
RAM[0xfdfc] = 0x0000000b00000000;
RAM[0xfe01] = 0x0000000b00000000;
RAM[0xfe06] = 0x0000000b00000000;
RAM[0xfe0b] = 0x0000000b00000000;
RAM[0xfe10] = 0x0000000b00000000;
RAM[0xfe15] = 0x0000000b00000000;
RAM[0xfe1a] = 0x0000000b00000000;
RAM[0xfe1f] = 0x0000000b00000000;
RAM[0xfe24] = 0x0000000b00000000;
RAM[0xfe29] = 0x0000000b00000000;
RAM[0xfe2e] = 0x0000000b00000000;
RAM[0xfe33] = 0x0000000b00000000;
RAM[0xfe38] = 0x0000000b00000000;
```

```
RAM[0xfe3d] = 0x0000000b00000000;
RAM[0xfe42] = 0x0000000b00000000;
RAM[0xfe47] = 0x0000000b00000000;
RAM[0xfe4c] = 0x0000000b00000000;
RAM[0xfe51] = 0x0000000b00000000;
RAM[0xfe56] = 0x0000000b00000000;
RAM[0xfe5b] = 0x0000000b00000000;
RAM[0xfe60] = 0x0000000b00000000;
RAM[0xfe65] = 0x0000000b00000000;
RAM[0xfe6a] = 0x0000000b00000000;
RAM[0xfe6f] = 0x0000000b00000000;
RAM[0xfe74] = 0x0000000b00000000;
RAM[0xfe79] = 0x0000000b00000000;
RAM[0xfe7e] = 0x0000000b00000000;
RAM[0xfe83] = 0x0000000b00000000;
RAM[0xfe88] = 0x0000000b00000000;
RAM[0xfe8d] = 0x0000000b00000000;
RAM[0xfe92] = 0x0000000b00000000;
RAM[0xfe97] = 0x0000000b00000000;
RAM[0xfe9c] = 0x0000000b00000000;
RAM[0xfea1] = 0x0000000b00000000;
RAM[0xfea6] = 0x0000000b00000000;
RAM[0xfeab] = 0x0000000b00000000;
RAM[0xfeb0] = 0x0000000b00000000;
RAM[0xfeb5] = 0x0000000b00000000;
RAM[0xfeba] = 0x0000000b00000000;
RAM[0xfebf] = 0x0000000b00000000;
RAM[0xfec4] = 0x0000000b00000000;
RAM[0xfec9] = 0x0000000b00000000;
RAM[0xfece] = 0x0000000b00000000;
RAM[0xfed3] = 0x0000000b00000000;
RAM[0xfed8] = 0x0000000b00000000;
RAM[0xfedd] = 0x0000000b00000000;
RAM[0xfee2] = 0x0000000b00000000;
RAM[0xfee7] = 0x0000000b00000000;
RAM[0xfeec] = 0x0000000b00000000;
RAM[0xfef1] = 0x0000000b00000000;
RAM[0xfef6] = 0x0000000b00000000;
RAM[0xfefb] = 0x0000000b00000000;
RAM[0xff00] = 0x0000000b00000000;
RAM[0xff05] = 0x0000000b00000000;
RAM[0xff0a] = 0x0000000b00000000;
RAM[0xff0f] = 0x0000000b00000000;
RAM[0xff14] = 0x0000000b00000000;
RAM[0xff19] = 0x0000000b00000000;
RAM[0xff1e] = 0x0000000b00000000;
RAM[0xff23] = 0x0000000b00000000;
RAM[0xff28] = 0x0000000b00000000;
RAM[0xff2d] = 0x0000000b00000000;
RAM[0xff32] = 0x0000000b00000000;
RAM[0xff37] = 0x0000000b00000000;
RAM[0xff3c] = 0x0000000b00000000;
RAM[0xff41] = 0x0000000b00000000;
RAM[0xff46] = 0x0000000b00000000;
RAM[0xff4b] = 0x0000000b00000000;
RAM[0xff50] = 0x0000000b00000000;
RAM[0xff55] = 0x0000000b00000000;
RAM[0xff5a] = 0x0000000b00000000;
```

```
    RAM[0xff5f] = 0x0000000b00000000;
    RAM[0xff64] = 0x0000000b00000000;
    RAM[0xff69] = 0x0000000b00000000;
    RAM[0xff6e] = 0x0000000b00000000;
    RAM[0xff73] = 0x0000000b00000000;
    RAM[0xff78] = 0x0000000b00000000;
    RAM[0xff7d] = 0x0000000b00000000;
    RAM[0xff82] = 0x0000000b00000000;
    RAM[0xff87] = 0x0000000b00000000;
    RAM[0xff8c] = 0x0000000b00000000;
    RAM[0xff91] = 0x0000000b00000000;
    RAM[0xff96] = 0x0000000b00000000;
    RAM[0xff9b] = 0x0000000b00000000;
    RAM[0xffa0] = 0x0000000b00000000;
    RAM[0xffa5] = 0x0000000b00000000;
    RAM[0xffaa] = 0x0000000b00000000;
    RAM[0xffaf] = 0x0000000b00000000;
    RAM[0xffb4] = 0x0000000b00000000;
    RAM[0xffb9] = 0x0000000b00000000;
    RAM[0xffbe] = 0x0000000b00000000;
    RAM[0xffc3] = 0x0000000b00000000;
    RAM[0xffc4] = 0x0000000b00000000;
    RAM[0xffc8] = 0x0000000b00000000;
    RAM[0xffcd] = 0x0000000b00000000;
    RAM[0xffd2] = 0x0000000b00000000;
    RAM[0xffd7] = 0x0000000b00000000;
    RAM[0xffdc] = 0x0000000b00000000;
    RAM[0xffe1] = 0x0000000b00000000;
    RAM[0xffe6] = 0x0000000b00000000;
    RAM[0xffeb] = 0x0000000b00000000;
    RAM[0xfff0] = 0x0000000b00000000;
    RAM[0xfff5] = 0x0000000b00000000;
    RAM[0xfffa] = 0x0000000b00000000;
    RAM[0xffff] = 0x0000000b00000000;
    printf("%5s %5s %5s %5s %24s %24s %24s %24s %24s
%24s","op","opr1","opr2","opr3", "ARI1", "ARI2", "ARI3", "ARI4", "ARI5",
"ARI6");
    printf("\n");
    printf("----- ----- ----- -----");
    printf("\n");
    uint64_t p = 0;
    do {
        i = transformInstruction(RAM[p], RAM[p+1]);
        p+=2;
        switch (i.op) {
            case 1: reg[i.opr1] = RAM[trans(i.opr2, i.opr3)];
break; // LOAD registrador <- RAM
            case 2: reg[i.opr1] = trans(i.opr2, i.opr3);
break; // LOAD registrador <- constante
            case 3: RAM[trans(i.opr2, i.opr3)] = reg[i.opr1];
break; // STORE
            case 4: reg[i.opr3] = reg[i.opr2];
break; // MOVE
            case 5: reg[i.opr1] = sum_comp2(reg[i.opr2], reg[i.opr3]);
break; // ADD complemento de 2
            case 6: reg[i.opr1] = sum_flut(reg[i.opr2], reg[i.opr3]);
break; // ADD ponto flutuante
            case 7: reg[i.opr1] = reg[i.opr2] | reg[i.opr3];
break; // OR
```

```
            case 8: reg[i.opr1] = reg[i.opr2] & reg[i.opr3];
break; // AND
            case 9: reg[i.opr1] = reg[i.opr2] ^ reg[i.opr3];
break; // OR EXCLUSIVO
            case 0xA: reg[i.opr1] = rot(reg[i.opr2], i.opr3);
break; // ROTACAO
            case 0xB: if(reg[i.opr1] == reg[0]) {p = trans(i.opr2, i.opr3);};
break; // JUMP
            case 0xC: p = 0; }
        printf("%5s %5x %5x %5x ", getInstructionName(i.op), i.opr1, i.opr2,
i.opr3);
        uint64_t stop = reg[0xfc];
        if(p > 0xfbf2 && p <= 0xfc06){
                stop = stop - 5; }
        for (uint64_t h = 0; h < 16; h++) {
           printf(" %8ld", reg[h]); }
        printf("\n");
    } while (p != 0);
    printf("----- ----- ----- -----");
    printf("\n");
    printf("%5s %5s %5s %5s %24s %24s %24s %24s %24s
%24s","op","opr1","opr2","opr3", "ARI1", "ARI2", "ARI3", "ARI4", "ARI5",
"ARI6");
    printf("\nend                      \n");}
uint64_t sum_comp2(uint64_t val1, uint64_t val2){
    uint64_t sinal1 = val1 >> 31;
    uint64_t sinal2 = val2 >> 31;
    uint64_t result;
    if(sinal1 == 0){
        if(sinal2 == 0){
            result = val1 + val2;
            if(result > (unsigned long long)9223372036854775807){
                result -= (unsigned long long)9223372036854775807; } }
        else{
            result = val1 - ((unsigned long long)18446744073709551615 - val2 +
1);
            if(result < 0){
                result *= -1;
                result = (unsigned long long)18446744073709551615 - result +
1; } } }
    else{
        if(sinal2 == 0){
            result =  val2 - ((unsigned long long)18446744073709551615 - val1 +
1);
            if(result < 0){
                result *= -1;
                result = (unsigned long long)18446744073709551615 - result +
1; } }
        else{
            result = ((unsigned long long)18446744073709551615 - val1 + 1) +
((unsigned long long)18446744073709551615 - val2 + 1);
            result = (unsigned long long)18446744073709551615 - result + 1; } }
    return result;}
uint64_t sum_flut(uint64_t val1, uint64_t val2){
    uint64_t sinal1, sinal2, expv1, expv2, mant1, mant2, result, soma;
    sinal1 = val1>>63;
    sinal2 = val2>>63;
    expv1 = val1<<1; expv1 = expv1>>33;
    expv2 = val2<<1; expv2 = expv2>>33;
```

```
    mant1 = val1 & 0xffffffff;
    mant2 = val2 & 0xffffffff;
    if(expv1 != expv2){
        uint64_t aux;
        if(expv1 > expv2){
            aux = expv1-expv2;
            mant2>>aux; }
        else{
            aux = expv2-expv1;
            mant1>>aux; } }
    if(sinal1 == 0){
        if(sinal2 == 0){
            soma = mant1 + mant2;
            result = sinal1<<63;
            result += expv1<<32;
            result += soma;
            return result; }
        else{
            soma = abs(mant1 - mant2);
            if(mant2 > mant1) { result = sinal2<<63; }
            else { result = sinal1<<63;}
            result += expv1<<32;
            result += soma;
            return result; } }
    else{
        if(sinal2 == 0){
            soma = abs(mant1 - mant2);
            if(mant1 > mant2) { result = sinal1<<63; }
            else { result = sinal2<<63;}
            result += expv1<<32;
            result += soma;
            return result; }
        else{
            soma = abs(mant1 + mant2);
            result = sinal1<<63;
            result += expv1<<32;
            result += soma;
            return result; } }}
uint64_t rot(uint64_t val1, uint64_t val2){
    uint64_t a = val1>>val2 ;
    uint64_t b = val1<<(64-val2);
    uint64_t c = a+b;
    return c;}
uint64_t trans(uint64_t val1, uint64_t val2){
    uint64_t a1 = val1<<32;
    uint64_t b = a1 + val2;
    return b;}
Tinstruction transformInstruction(uint64_t a, uint64_t b){
    Tinstruction result;
    result.op = a>>32;
    result.opr1 = a & (uint64_t)4294967295;
    result.opr2 = b>>32;
    result.opr3 = b & (uint64_t)4294967295;
    return result;}
char * getInstructionName(uint64_t inst){
    return instructionString[inst - 1];}
```

APÊNDICE E — Implementação C da máquina descrita por Brookshear

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<inttypes.h>
#include<math.h>
typedef struct{
    unsigned char op;        // operacao
    unsigned char opr1;      // operando1
    unsigned char opr2;      // operando2
    unsigned char opr3;      // operando3
} Tinstruction;
int p;
Tinstruction i;
uint8_t RAM[256];
uint8_t reg[16];   // registradores
int h;
// List of available instructions
char * instructionString[] = { "LOAD", "LOAD", "STORE", "MOVE", "ADD", "ADD",
"OR", "AND", "EXCLS", "ROTAT", "JUMP", "HALT"};
// Prototypes
char * getInstructionName(int inst);
uint8_t sum_comp2(uint8_t val1, int val2);
uint8_t sum_flut(uint8_t val1, int val2);
uint8_t rot(uint8_t val1, uint8_t val2);
uint8_t trans(uint8_t val1, uint8_t val2);
Tinstruction transformInstruction(uint8_t a, uint8_t b);
int main (){
      printf("start                   registradores\n");
    h = 0; p=0;
// Aqui voce preenche as instrucoes na RAM
// RAM[0x0] = 0xOpOpr1;          RAM[0x1] = 0xOpr2Opr3;
    printf("%5s %5s %5s %5s","op","opr1","opr2","opr3");
    for(h = 0; h <16; h++) {
        printf(" %8x", h); }
    printf("\n");
    printf("----- ----- ----- -----");
    for(h = 0; h <16; h++) {
        printf(" --------"); }
    printf("\n");
    do {
            i = transformInstruction(RAM[p], RAM[p+1]);
            p+=2;
            switch (i.op) {
                  case 1: reg[i.opr1] = RAM[trans(i.opr2, i.opr3)];
```

```
                break; // LOAD registrador <- RAM
                  case 2: reg[i.opr1] = trans(i.opr2, i.opr3);
                  break; // LOAD registrador <- constante
                  case 3: RAM[trans(i.opr2, i.opr3)] = reg[i.opr1];
            break; // STORE
                  case 4: reg[i.opr3] = reg[i.opr2];
                        break; // MOVE
                  case 5: reg[i.opr1] = sum_comp2(reg[i.opr2], reg[i.opr3]);
            break; // ADD complemento de 2
                  case 6: reg[i.opr1] = sum_flut(reg[i.opr2], reg[i.opr3]);
            break; // ADD ponto flutuante
                  case 7: reg[i.opr1] = reg[i.opr2] | reg[i.opr3];
                  break; // OR
                  case 8: reg[i.opr1] = reg[i.opr2] & reg[i.opr3];
                  break; // AND
                  case 9: reg[i.opr1] = reg[i.opr2] ^ reg[i.opr3];
                  break; // OR EXCLUSIVO
                  case 0xA: reg[i.opr1] = rot(reg[i.opr1], i.opr3);
                  break; // ROTACAO
                  case 0xB: if(reg[i.opr1] == reg[0]) {p = trans(i.opr2,
i.opr3);};  break; // JUMP
                  case 0xC: p = 0;
                              break; // HALT }
          printf("%5s %5x %5x %5x ", getInstructionName(i.op), i.opr1, i.opr2,
i.opr3);
            for (h = 0; h < 16; h++) {
                  int aux = reg[h];
            for(int j = 7; j > -1; j--) {
                if ( (aux/pow(2, j)) >= 1 ) { aux -= pow(2, j); printf("1"); }
                else { printf("0"); } }
            printf(" "); }
            printf("\n");
    } while (p != 0);
    printf("----- ----- ----- -----");
    for(h = 0; h <16; h++) {
        printf(" --------"); }
    printf("\n");
    printf("%5s %5s %5s %5s","op","opr1","opr2","opr3");
    for(h = 0; h <16; h++) {
        printf(" %8x", h); }
    printf("\nend                     registradores\n"); }
uint8_t sum_comp2(uint8_t val1, int val2){
    uint8_t comp2_val1 = (255 - val1) + 0b00000001;
    uint8_t comp2_val2 = (255 - val2) + 0b00000001;
    uint8_t result = comp2_val1 + comp2_val2;
    result = (255 - result) + 0b00000001;
    return result;}
uint8_t sum_flut(uint8_t val1, int val2){
    uint8_t sinal1, sinal2, expv1, expv2, mant1, mant2, result, soma;
    sinal1 = val1>>7;
```

```
    sinal2 = val2>>7;
    expv1 = val1<<1; expv1 = expv1>>5;
    expv2 = val2<<1; expv2 = expv2>>5;
    mant1 = val1<<4; mant1 = mant1>>4;
    mant2 = val2<<4; mant2 = mant2>>4;
    if(expv1 != expv2){
        int aux;
        if(expv1 > expv2){
            aux = expv1-expv2;
            mant2>>aux; }
        else{
            aux = expv2-expv1;
            mant1>>aux; } }
    if(sinal1 == 0){
        if(sinal2 == 0){
            soma = mant1 + mant2;
            result = sinal1<<7;
            result += expv1<<4;
            result += soma;
            return result; }
        else{
            soma = abs(mant1 - mant2);
            if(mant2 > mant1) { result = sinal2<<7; }
            else { result = sinal1<<7;}
            result += expv1<<4;
            result += soma;
            return result; } }
    else{
        if(sinal2 == 0){
            soma = abs(mant1 - mant2);
            if(mant1 > mant2) { result = sinal1<<7; }
            else { result = sinal2<<7;}
            result += expv1<<4;
            result += soma;
            return result; }
        else{
            soma = abs(mant1 + mant2);
            result = sinal1<<7;
            result += expv1<<4;
            result += soma;
            return result; } } }
uint8_t rot(uint8_t val1, uint8_t val2){
    uint8_t a = val1>>val2 ;
    uint8_t b = val1<<(8-val2);
    uint8_t c = a+b;
    return c;}
uint8_t trans(uint8_t val1, uint8_t val2){
```

```
    uint8_t a = val1*16;
    uint8_t b = a + val2;
    return b;}
Tinstruction transformInstruction(uint8_t a, uint8_t b){
    Tinstruction result;
    result.op = a>>4;
    result.opr1 = a<<4; result.opr1 = result.opr1>>4;
    result.opr2 = b>>4;
    result.opr3 = b<<4; result.opr3 = result.opr3>>4;
    return result;}
char * getInstructionName(int inst){
      return instructionString[inst - 1];}
```

APÊNDICE F — Implementação C da P-code

```
/*
    Author: Gustavo B. Fragoso
    2021-10-26: Updated by Allana Campos
*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>
#define MAX_INSTRUCTIONS 50 // Define your own
#define STACKSIZE 1024
typedef struct{
    int f;
    int l;
    int a;
} Tinstruction;
int p;                  // Points to an i in the program area
int b;                        // Points to the b address in the s for the
current invocation of a given procedure
int t;                        // Points to the current t of the s
int totalInstrctns = 0;
Tinstruction i;
int s[STACKSIZE] = {};        // Vector with only integers, used as a datastore
int h;
// Array of instructions (Input)
Tinstruction code[MAX_INSTRUCTIONS];
// List of available instructions
enum { LIT, OPR, LOD, STO, CAL, INT, JMP, JPC } instructionCode;
char * instructionString[] = { "LIT", "OPR", "LOD", "STO", "CAL", "INT", "JMP",
"JPC"};
// List of available arithmetic operations
enum { RTN, NEG, ADD, SUB, MUL, DIV, ODD, MOD, EQL, NEQ, LSS, LEQ, GTR, GEQ }
operationCode;
char * operationString[] = { "RTN", "NEG", "ADD", "SUB", "MUL", "DIV", "MOD",
"ODD", "EQL", "NEQ", "LSS", "LEQ", "GTR", "GEQ"};
// Prototypes
void executeInstruction();
void stackOperation(int a);
int base(int l);
int getInstructionCode(char *c);
char * getInstructionName(int inst);
int main(){
      printf("start pl/0              stack\n");
    h = 0;
    t = -1; b = 0; p = 0;
// Aqui voce preenche as instrucoes no vetor code
// code[ 0].f = OPR; code[ 0].l = 0; code[ 0].a = 0;
    printf("%3s %3s %3s %3s %3s %3s","f","l","a","p","b","t");
    for(h = 0; h <22; h++) {
        printf(" %3d" , h); }
    printf("\n");
    printf("--- --- --- --- --- ---");
    for(h = 0; h <22; h++) {
        printf(" ---"); }
    printf("\n");
```

```
    do {
            i = code[p];
            p++;
            switch (i.f) {
                  // LIT, OPR, LOD, STO, CAL, INT, JMP, JPC
                  // 0  1      2      3      4  5      6      7
                  case LIT:  t++; s[t] = i.a; break;
                  case OPR:
                        switch (i.a) {
                              // RTN, NEG, ADD, SUB, MUL, DIV, ODD, MOD, EQL,
NEQ, LSS, LEQ, GTR, GEQ
                              // 0  1      2      3      4  5      6      7    8
      9      10     11   12   13
                              case 0: t = b - 1; p = s[t+3]; b = s[t+2]; break;
                              case 1: s[t]= -s[t]; break;
                              case 2: t--; s[t] += s[t+1]; break;
                              case 3: t--; s[t] -= s[t+1]; break;
                              case 4: t--; s[t] *= s[t+1]; break;
                              case 5: t--; s[t] /= s[t+1]; break;
                              case 6: t--; s[t] %= s[t+1]; break;
                              case 7: s[t] = (s[t]%2 == 1); break;
                              case 8: t--; s[t] = (s[t] == s[t+1]); break;
                              case 9: t--; s[t] = (s[t] != s[t+1]); break;
                              case 10: t--; s[t] = (s[t] < s[t+1]); break;
                              case 11: t--; s[t] = (s[t] <= s[t+1]); break;
                              case 12: t--; s[t] = (s[t] > s[t+1]); break;
                              case 13: t--; s[t] = (s[t] >= s[t+1]); break; }
                        break;
                  case LOD: t++; s[t] = s[ base(i.l) + i.a ]; break;
                  case STO: s[ base(i.l) + i.a ] = s[t]; t--; break;
                  case CAL: s[t+1] = base( i.l ); s[t+2] = b; s[t+3] = p; b =
t+1; p = i.a; break;
                  case INT: t += i.a; break;
                  case JMP: p = i.a; break;
                  case JPC: if ( s[t] == 0 ) { p = i.a; } t--; break; }
            printf("%3s %3d %3d %3d %3d %3d ", getInstructionName(i.f), i.l,
i.a, p, b, t);
            for (h = 0; h < t+1; h++){
                  printf(" %3d", s[h]); }
            printf("\n");
      } while (p != 0);
    printf("--- --- --- --- --- ---");
    for(h = 0; h <22; h++) {
        printf(" ---"); }
    printf("\n");
    printf("%3s %3s %3s %3s %3s %3s","f","l","a","p","b","t");
    for(h = 0; h <22; h++) {
        printf(" %3d" , h); }
    printf("\nend pl/0                stack\n");}
int base(int l){
    int newBase;
      newBase = b;
      while (l>0){
            newBase = s[newBase];
            l--; }
      return newBase;}
char * getInstructionName(int inst){
      return instructionString[inst];}
```

APÊNDICE G — Implementação C dos programas de teste da AllanaVM, Brookshear e P-code

1. Fatorial() iterativo para AllanaVM; para inserir o valor de entrada necessita alterar o conteúdo da RAM[0xd] de *a* para o valor desejado.

```
    RAM[ 0x0] = 0x00000002000000fa;           RAM[ 0x1] = 0x0000000000000005;
//RAM[ 0x0]: LOAD  0a 02   // REG[0xfa] <- 5
    RAM[ 0x2] = 0x00000002000000fb;           RAM[ 0x3] = 0xfffffffffffffffb;
//RAM[ 0x2]: LOAD  fb -5   // REG[0xfb] <- -5
    RAM[ 0x4] = 0x00000002000000fe;           RAM[ 0x5] = 0x000000000000fc09;
//RAM[ 0x4]: LOAD  fe fc09 // REG[0xfe] <- 0xfc09 // posicao return
    RAM[ 0x6] = 0x00000002000000f1;           RAM[ 0x7] = 0x000000000000fc0a;
//RAM[ 0x6]: LOAD  f1 fc0a // REG[0xf1] <- 0xfc0a // posicao param
    RAM[ 0x8] = 0x00000002000000ff;           RAM[ 0x9] = 0x000000000000fc0b;
//RAM[ 0x8]: LOAD  ff fc0b // REG[0xff] <- 0xfc0b // posicao param
    RAM[ 0xa] = 0x00000002000000fc;           RAM[ 0xb] = 0x000000000000fc0c;
//RAM[ 0xa]: LOAD  fc fc0c // REG[0xfc] <- 0xfc0c // posicao result
    RAM[ 0xc] = 0x000000020000000f;           RAM[ 0xd] = a;
//RAM[ 0xc]: LOAD  0f a    // REG[0x0f] <- a      // Parametros
    RAM[ 0xe] = 0x0000000400000000;           RAM[ 0xf] = 0x0000000f00000004;
//MOVE
    RAM[0x10] = 0x0000000200000005;           RAM[0x11] = 0xffffffffffffffff;
//LOAD
    RAM[0x12] = 0x0000000200000000;           RAM[0x13] = 0x0000000000000001;
//RAM[ 0xe]: LOAD  0e 01   // REG[0x0e] <- 0x0001 // Parametros
    RAM[0x14] = 0x0000000b0000000f;           RAM[0x15] = 0x0000000000000026;
//JUMP
    RAM[0x16] = 0x000000050000000f;           RAM[0x17] = 0x0000000f00000005;
//ADD
    RAM[0x18] = 0x0000000400000000;           RAM[0x19] = 0x000000040000000e;
//MOVE
    RAM[0x1a] = 0x0000000200000000;           RAM[0x1b] = 0x0000000000000022;
//RAM[0xee94]: LOAD  00 28    // REG[0x00] <- 28
    RAM[0x1c] = 0x0000000300000000;           RAM[0x1d] = 0x000000000000fc07;
//RAM[0xee96]: STORE 00 fc07  // RAM[0xfc07] <- REG[0x00]
    RAM[0x1e] = 0x00000002000000fd;           RAM[0x1f] = 0x0000000000000024;
//RAM[0xee98]: LOAD  fd 2a    // REG[0xfd] <- 2a
    RAM[0x20] = 0x0000000b00000000;           RAM[0x21] = 0x000000000000fbea;
//RAM[0xee9a]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} //
LIT
    RAM[0x22] = 0x0000000b00000000;           RAM[0x23] = 0x000000000000ef8e;
//RAM[0xee9c]: JUMP  00 ef8e  // if (REG[0x00] == REG[0x00]) {p <- 0xef8e;} //
MULT
    RAM[0x24] = 0x0000000b00000000;           RAM[0x25] = 0x0000000000000012;
//JUMP
    RAM[0x26] = 0x0000000c00000000;           RAM[0x27] = 0x0000000000000000;
//RAM[0x1a]: HALT  00 00   // END
```

2. Fatorial() recursivo para AllanaVM; para inserir o valor de entrada necessita alterar o conteúdo da RAM[0xd] de *a* para o valor desejado.

```
     RAM[ 0x0] = 0x00000002000000fa;           RAM[ 0x1] = 0x0000000000000005;
//RAM[ 0x0]: LOAD  0a 02   // REG[0xfa] <- 5
```

```
    RAM[ 0x2] = 0x00000002000000fb;           RAM[ 0x3] = 0xfffffffffffffffb;
//RAM[ 0x2]: LOAD  fb -5   // REG[0xfb] <- -5
    RAM[ 0x4] = 0x00000002000000fe;           RAM[ 0x5] = 0x000000000000fc09;
//RAM[ 0x4]: LOAD  fe fc09 // REG[0xfe] <- 0xfc09 // posicao return
    RAM[ 0x6] = 0x00000002000000f1;           RAM[ 0x7] = 0x000000000000fc0a;
//RAM[ 0x6]: LOAD  f1 fc0a // REG[0xf1] <- 0xfc0a // posicao param
    RAM[ 0x8] = 0x00000002000000ff;           RAM[ 0x9] = 0x000000000000fc0b;
//RAM[ 0x8]: LOAD  ff fc0b // REG[0xff] <- 0xfc0b // posicao param
    RAM[ 0xa] = 0x00000002000000fc;           RAM[ 0xb] = 0x000000000000fc0c;
//RAM[ 0xa]: LOAD  fc fc0c // REG[0xfc] <- 0xfc0c // posicao result
    RAM[ 0xc] = 0x000000020000000f;           RAM[ 0xd] = a;
//RAM[ 0xc]: LOAD  0f a    // REG[0x0f] <- a      // Parametros
    RAM[ 0xe] = 0x000000020000000e;           RAM[ 0xf] = 0x0000000000000001;
//RAM[ 0xe]: LOAD  0e 01   // REG[0x0e] <- 0x0001 // Parametros
    RAM[0x10] = 0x00000002000000fd;           RAM[0x11] = 0x000000000000001a;
//RAM[0x10]: LOAD  fd 1a   // REG[0xfd] <- 0x001a // retorno da chamada
FIBONACCI
    RAM[0x12] = 0x0000000200000000;           RAM[0x13] = 0x0000000000000018;
//RAM[0x12]: LOAD  00 18   // REG[0x00] <- 0x0018 // posicao de retorno da
chamada LIT
    RAM[0x14] = 0x0000000300000000;           RAM[0x15] = 0x000000000000fc07;
//RAM[0x14]: STORE 00 fc07 // RAM[0xfc07] <- REG[00] // salva retorno da chamada
LIT
    RAM[0x16] = 0x0000000b00000000;           RAM[0x17] = 0x000000000000fbea;
//RAM[0x16]: JUMP  00 fbea // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} // LIT
    RAM[0x18] = 0x0000000b00000000;           RAM[0x19] = 0x000000000000001c;
//RAM[0x18]: JUMP  00 1c   // if (REG[0x00] == REG[0x00]) {p <- 0x001c;} //
FIBONACCI
    RAM[0x1a] = 0x0000000c00000000;           RAM[0x1b] = 0x0000000000000000;
//RAM[0x1a]: HALT  00 00   // END
    //FATORIAL
    RAM[0x1c] = 0x0000000200000001;           RAM[0x1d] = 0xffffffffffffffff;
//RAM[0xee90]: LOAD  01 -1    // REG[0x01] <- -1
    RAM[0x1e] = 0x000000020000000e;           RAM[0x1f] = 0x0000000000000001;
//RAM[0xee92]: LOAD  0e 01    // REG[0x0e] <- 01
    RAM[0x20] = 0x0000000200000000;           RAM[0x21] = 0x0000000000000028;
//RAM[0xee94]: LOAD  00 28    // REG[0x00] <- 28
    RAM[0x22] = 0x0000000300000000;           RAM[0x23] = 0x000000000000fc07;
//RAM[0xee96]: STORE 00 fc07  // RAM[0xfc07] <- REG[0x00]
    RAM[0x24] = 0x00000002000000fd;           RAM[0x25] = 0x000000000000002a;
//RAM[0xee98]: LOAD  fd 2a    // REG[0xfd] <- 2a
    RAM[0x26] = 0x0000000b00000000;           RAM[0x27] = 0x000000000000fbea;
//RAM[0xee9a]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} //
LIT
    RAM[0x28] = 0x0000000b00000000;           RAM[0x29] = 0x000000000000eece;
//RAM[0xee9c]: JUMP  00 eece  // if (REG[0x00] == REG[0x00]) {p <- 0xeece;} //
GEQ
    RAM[0x2a] = 0x0000000200000000;           RAM[0x2b] = 0x0000000000000000;
//RAM[0xee9e]: LOAD  00 00    // REG[0x00] <- 00
    RAM[0x2c] = 0x0000000b00000004;           RAM[0x2d] = 0x0000000000000032;
//RAM[0xeea0]: JUMP  04 38    // if (REG[0x00] == REG[0x04]) {p <- 0x38;}
    RAM[0x2e] = 0x0000000200000004;           RAM[0x2f] = 0x0000000000000001;
//RAM[0xeea2]: LOAD  04 01    // REG[0x04] <- 01
    RAM[0x30] = 0x0000000b00000000;           RAM[0x31] = 0x000000000000fbd2;
//RAM[0xeeaa]: JUMP  00 fbd2  // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;}
    RAM[0x32] = 0x000000050000000f;           RAM[0x33] = 0x0000000f00000001;
//RAM[0xeeac]: ADD   0f f1    // REG[0x0f] <- REG[0x0f] + REG[0x01]
    RAM[0x34] = 0x0000000200000000;           RAM[0x35] = 0x000000000000003c;
//RAM[0xeeae]: LOAD  00 3c    // REG[0x00] <- 3c
    RAM[0x36] = 0x0000000300000000;           RAM[0x37] = 0x000000000000fc07;
```

```
//RAM[0xeeb0]: STORE 00 fc07  // RAM[0xfc07] <- REG[0x00]
    RAM[0x38] = 0x00000002000000fd;           RAM[0x39] = 0x000000000000003e;
//RAM[0xeeb2]: LOAD  fd 3e    // REG[0xfd] <- 3e
    RAM[0x3a] = 0x0000000b00000000;           RAM[0x3b] = 0x000000000000fbea;
//RAM[0xeeb4]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} //
LIT
    RAM[0x3c] = 0x0000000b00000000;           RAM[0x3d] = 0x000000000000001c;
//RAM[0xeeb6]: JUMP  00 1c    // if (REG[0x00] == REG[0x00]) {p <- 0xee90;} //
FATORIAL
    RAM[0x3e] = 0x0000000400000000;           RAM[0x3f] = 0x000000040000000e;
//RAM[0xeeb8]: MOVE  00 4e    // REG[0x0e] <- REG[0x04]
    RAM[0x40] = 0x0000000200000003;           RAM[0x41] = 0x0000000000000001;
//RAM[0xeeb8]: LOAD  03 01    // REG[0x03] <- 01
    RAM[0x42] = 0x000000050000000f;           RAM[0x43] = 0x0000000f00000003;
//RAM[0xeeb8]: ADD   0f f3    // REG[0x0f] <- REG[0x0f] + REG[0x03]
    RAM[0x44] = 0x0000000200000000;           RAM[0x45] = 0x000000000000004c;
//RAM[0xeeba]: LOAD  00 4c    // REG[0x00] <- 4c
    RAM[0x46] = 0x0000000300000000;           RAM[0x47] = 0x000000000000fc07;
    RAM[0x48] = 0x00000002000000fd;           RAM[0x49] = 0x000000000000004e;
//RAM[0xeebe]: LOAD  fd 4e    // REG[0xfd] <- 4e
    RAM[0x4a] = 0x0000000b00000000;           RAM[0x4b] = 0x000000000000fbea;
//RAM[0xeec0]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} //
LIT
    RAM[0x4c] = 0x0000000b00000000;           RAM[0x4d] = 0x000000000000ef8e;
//RAM[0xeec2]: JUMP  00 1c    // if (REG[0x00] == REG[0x00]) {p <- 0xef8e;} //
MULT
    RAM[0x4e] = 0x0000000b00000000;           RAM[0x4f] = 0x000000000000fbd2;
//RAM[0xeec8]: JUMP  00 fbd2  // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;} //
RET
```

3. Fibonacci() iterativo para AllanaVM; para inserir o valor de entrada necessita alterar o conteúdo da RAM[0xd] de *a* para o valor desejado.

```
      RAM[ 0x0] = 0x00000002000000fa;           RAM[ 0x1] = 0x0000000000000005;
//RAM[ 0x0]: LOAD  0a 02   // REG[0xfa] <- 5
    RAM[ 0x2] = 0x00000002000000fb;           RAM[ 0x3] = 0xfffffffffffffffb;
//RAM[ 0x2]: LOAD  fb -5   // REG[0xfb] <- -5
    RAM[ 0x4] = 0x00000002000000fe;           RAM[ 0x5] = 0x000000000000fc09;
//RAM[ 0x4]: LOAD  fe fc09 // REG[0xfe] <- 0xfc09 // posicao return
    RAM[ 0x6] = 0x00000002000000f1;           RAM[ 0x7] = 0x000000000000fc0a;
//RAM[ 0x6]: LOAD  f1 fc0a // REG[0xf1] <- 0xfc0a // posicao param
    RAM[ 0x8] = 0x00000002000000ff;           RAM[ 0x9] = 0x000000000000fc0b;
//RAM[ 0x8]: LOAD  ff fc0b // REG[0xff] <- 0xfc0b // posicao param
    RAM[ 0xa] = 0x00000002000000fc;           RAM[ 0xb] = 0x000000000000fc0c;
//RAM[ 0xa]: LOAD  fc fc0c // REG[0xfc] <- 0xfc0c // posicao result
    RAM[ 0xc] = 0x000000020000000f;           RAM[ 0xd] = a;
//RAM[ 0x8]: 2f02 LOAD  0f 02 //Parametros
    RAM[ 0xe] = 0x000000020000000e;           RAM[ 0xf] = 0x0000000000000001;
//RAM[ 0xa]: 2e07 LOAD  0e 07 //Parametros
    RAM[0x10] = 0x0000000200000000;           RAM[0x11] = 0x0000000000000018;
//RAM[0x4c]: 2054 LOAD  00 54 // REG[0x00] <- 16 // posicao de retorno da
chamada LIT
    RAM[0x12] = 0x0000000300000000;           RAM[0x13] = 0x000000000000fc07;
//RAM[0x4e]: 30a7 STORE 00 a7 // RAM[0xa7] <- REG[0x00]// salva retorno da
chamada LIT
    RAM[0x14] = 0x00000002000000fd;           RAM[0x15] = 0x000000000000001a;
//RAM[0x50]: 29a1 LOAD 09 a1 // RAM[0xa1] <- REG[0x09] // retorno da chamada GEQ
    RAM[0x16] = 0x0000000b00000000;           RAM[0x17] = 0x000000000000fbea;
//RAM[0x52]: b0a0 JUMP  00 a0 // if (REG[0x00] == REG[0x00]) {p <- 0xa0;} // LIT
```

```
    RAM[0x18] = 0x0000000b00000000;           RAM[0x19] = 0x000000000000eece;
//RAM[0x58]: b05a JUMP  00 5a // GEQ
    RAM[0x1a] = 0x0000000200000000;           RAM[0x1b] = 0x0000000000000001;
//RAM[ 0xa]: 2e07 LOAD  0e 07 //Parametros
    RAM[0x1c] = 0x0000000b00000004;           RAM[0x1d] = 0x0000000000000038;
//RAM[0x1a]: b088 JUMP  00 88 //00
    RAM[0x1e] = 0x0000000200000001;           RAM[0x1f] = 0x0000000000000002;
//RAM[ 0xa]: 2e07 LOAD  0e 07 //Parametros
    RAM[0x20] = 0x0000000200000002;           RAM[0x21] = 0x0000000000000000;
//RAM[ 0xa]: 2e07 LOAD  0e 07 //Parametros
    RAM[0x22] = 0x0000000200000003;           RAM[0x23] = 0x0000000000000001;
//RAM[ 0xa]: 2e07 LOAD  0e 07 //Parametros
    //01
    RAM[0x24] = 0x0000000500000009;           RAM[0x25] = 0x0000000200000003;
//RAM[ 0xc]: 2c1c LOAD  0c 16
    RAM[0x26] = 0x0000000400000000;           RAM[0x27] = 0x000000010000000e;
    RAM[0x28] = 0x0000000200000000;           RAM[0x29] = 0x0000000000000030;
//RAM[0x4c]: 2054 LOAD  00 54 // REG[0x00] <- 54// posicao de retorno da chamada
LIT
    RAM[0x2a] = 0x0000000300000000;           RAM[0x2b] = 0x000000000000fc07;
//RAM[0x4e]: 30a7 STORE 00 a7 // RAM[0xa7] <- REG[0x00]// salva retorno da
chamada LIT
    RAM[0x2c] = 0x00000002000000fd;           RAM[0x2d] = 0x0000000000000032;
//RAM[0x50]: 29a1 LOAD 09 a1 // RAM[0xa1] <- REG[0x09]// retorno da chamada GEQ
    RAM[0x2e] = 0x0000000b00000000;           RAM[0x2f] = 0x000000000000fbea;
//RAM[0x52]: b0a0 JUMP  00 a0 // if (REG[0x00] == REG[0x00]) {p <- 0xa0;}// LIT
    RAM[0x30] = 0x0000000b00000000;           RAM[0x31] = 0x000000000000eece;
//RAM[0x58]: b05a JUMP  00 5a // GEQ
    RAM[0x32] = 0x0000000200000000;           RAM[0x33] = 0x0000000000000000;
//RAM[ 0xa]: 2e07 LOAD  0e 07 //Parametros
    RAM[0x34] = 0x0000000b00000004;           RAM[0x35] = 0x000000000000003c;
//RAM[0x1a]: b088 JUMP  00 88 //02
    RAM[0x36] = 0x0000000400000000;           RAM[0x37] = 0x0000000900000004;
    RAM[0x38] = 0x0000000c00000000;           RAM[0x39] = 0x0000000000000000;
//RAM[0x1c]: c000 HALT  00 00
    RAM[0x3a] = 0x0000000400000000;           RAM[0x3b] = 0x0000000f00000004;
//RAM[ 0xc]: 2c1c LOAD  0c 16 //00
    RAM[0x3c] = 0x0000000c00000000;           RAM[0x3d] = 0x0000000000000000;
//RAM[0x1c]: c000 HALT  00 00
    RAM[0x3e] = 0x0000000400000000;           RAM[0x3f] = 0x0000000300000002;
//02
    RAM[0x40] = 0x0000000400000000;           RAM[0x41] = 0x0000000900000003;
    RAM[0x42] = 0x0000000200000009;           RAM[0x43] = 0x0000000000000001;
//RAM[ 0xe]: 2016 LOAD  00 14
    RAM[0x44] = 0x0000000500000001;           RAM[0x45] = 0x0000000100000009;
    RAM[0x46] = 0x0000000b00000000;           RAM[0x47] = 0x0000000000000022;
//RAM[0x1a]: b088 JUMP  00 88 //01
```

4. Fibonacci() recursivo para AllanaVM; para inserir o valor de entrada necessita alterar o conteúdo da RAM[0xd] de *a* para o valor desejado.

```
      RAM[ 0x0] = 0x00000002000000fa;           RAM[ 0x1] = 0x0000000000000005;
//RAM[ 0x0]: LOAD  0a 02   // REG[0xfa] <- 5
    RAM[ 0x2] = 0x00000002000000fb;           RAM[ 0x3] = 0xfffffffffffffffb;
//RAM[ 0x2]: LOAD  fb -5   // REG[0xfb] <- -5
    RAM[ 0x4] = 0x00000002000000fe;           RAM[ 0x5] = 0x000000000000fc09;
//RAM[ 0x4]: LOAD  fe fc09 // REG[0xfe] <- 0xfc09 // posicao return
    RAM[ 0x6] = 0x00000002000000f1;           RAM[ 0x7] = 0x000000000000fc0a;
```

```
//RAM[ 0x6]: LOAD  f1 fc0a // REG[0xf1] <- 0xfc0a // posicao param
    RAM[ 0x8] = 0x00000002000000ff;           RAM[ 0x9] = 0x000000000000fc0b;
//RAM[ 0x8]: LOAD  ff fc0b // REG[0xff] <- 0xfc0b // posicao param
    RAM[ 0xa] = 0x00000002000000fc;           RAM[ 0xb] = 0x000000000000fc0c;
//RAM[ 0xa]: LOAD  fc fc0c // REG[0xfc] <- 0xfc0c // posicao result
    RAM[ 0xc] = 0x0000000200000000f;          RAM[ 0xd] = a;
//RAM[ 0xc]: LOAD  0f a    // REG[0x0f] <- a      // Parametros
    RAM[ 0xe] = 0x0000000200000000e;          RAM[ 0xf] = 0x0000000000000001;
//RAM[ 0xe]: LOAD  0e 01   // REG[0x0e] <- 0x0001 // Parametros
    RAM[0x10] = 0x00000002000000fd;           RAM[0x11] = 0x000000000000001a;
//RAM[0x10]: LOAD  fd 1a   // REG[0xfd] <- 0x001a // retorno da chamada
FIBONACCI
    RAM[0x12] = 0x0000000200000000;           RAM[0x13] = 0x0000000000000018;
//RAM[0x12]: LOAD  00 18   // REG[0x00] <- 0x0018 // posicao de retorno da
chamada LIT
    RAM[0x14] = 0x0000000300000000;           RAM[0x15] = 0x000000000000fc07;
//RAM[0x14]: STORE 00 fc07 // RAM[0xfc07] <- REG[00] // salva retorno da chamada
LIT
    RAM[0x16] = 0x0000000b00000000;           RAM[0x17] = 0x000000000000fbea;
//RAM[0x16]: JUMP  00 fbea // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} // LIT
    RAM[0x18] = 0x0000000b00000000;           RAM[0x19] = 0x000000000000001c;
//RAM[0x18]: JUMP  00 1c   // if (REG[0x00] == REG[0x00]) {p <- 0x001c;} //
FIBONACCI
    RAM[0x1a] = 0x0000000c00000000;           RAM[0x1b] = 0x0000000000000000;
//RAM[0x1a]: HALT  00 00   // END
    RAM[0x1a] = 0x0000000c00000000;           RAM[0x1b] = 0x0000000000000000;
//RAM[0x1a]: HALT  00 00   // END
    //FIBONACCI
    RAM[0x1c] = 0x0000000200000001;           RAM[0x1d] = 0xffffffffffffffff;
//RAM[0xee90]: LOAD  01 -1    // REG[0x01] <- -1
    RAM[0x1e] = 0x0000000200000000e;          RAM[0x1f] = 0x0000000000000001;
//RAM[0xee92]: LOAD  0e 01    // REG[0x0e] <- 01
    RAM[0x20] = 0x0000000200000000;           RAM[0x21] = 0x0000000000000028;
//RAM[0xee94]: LOAD  00 28    // REG[0x00] <- 28
    RAM[0x22] = 0x0000000300000000;           RAM[0x23] = 0x000000000000fc07;
//RAM[0xee96]: STORE 00 fc07  // RAM[0xfc07] <- REG[0x00]
    RAM[0x24] = 0x00000002000000fd;           RAM[0x25] = 0x000000000000002a;
//RAM[0xee98]: LOAD  fd 2a    // REG[0xfd] <- 2a
    RAM[0x26] = 0x0000000b00000000;           RAM[0x27] = 0x000000000000fbea;
//RAM[0xee9a]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;} //
LIT
    RAM[0x28] = 0x0000000b00000000;           RAM[0x29] = 0x000000000000eece;
//RAM[0xee9c]: JUMP  00 eece  // if (REG[0x00] == REG[0x00]) {p <- 0xeece;} //
GEQ
    RAM[0x2a] = 0x0000000200000000;           RAM[0x2b] = 0x0000000000000000;
//RAM[0xee9e]: LOAD  00 00    // REG[0x00] <- 00
    RAM[0x2c] = 0x0000000b00000004;           RAM[0x2d] = 0x0000000000000038;
//RAM[0xeea0]: JUMP  04 38    // if (REG[0x00] == REG[0x04]) {p <- 0x38;}
    RAM[0x2e] = 0x0000000100000009;           RAM[0x2f] = 0x00000000000000f4;
//RAM[0xeea2]: LOAD  09 f4    // REG[0x09] <- RAM[0xf4]
    RAM[0x30] = 0x0000000400000000;           RAM[0x31] = 0x0000000f00000004;
//RAM[0xeea4]: MOVE  00 f4    // REG[0x04] <- REG[0x0f]
    RAM[0x32] = 0x0000000500000004;           RAM[0x33] = 0x0000000400000009;
//RAM[0xeea6]: ADD   04 49    // REG[0x04] <- REG[0x04] + REG[0x09]
    RAM[0x34] = 0x0000000300000004;           RAM[0x35] = 0x00000000000000f4;
//RAM[0xeea8]: STORE 04 f4    // RAM[0xf4] <- REG[0x04]
    RAM[0x36] = 0x0000000b00000000;           RAM[0x37] = 0x000000000000fbd2;
//RAM[0xeeaa]: JUMP  00 fbd2  // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;}
    RAM[0x38] = 0x000000050000000f;           RAM[0x39] = 0x0000000f00000001;
//RAM[0xeeac]: ADD   0f f1    // REG[0x0f] <- REG[0x0f] + REG[0x01]
```

```
    RAM[0x3a] = 0x0000000200000000;           RAM[0x3b] = 0x0000000000000042;
//RAM[0xeeae]: LOAD  00 42    // REG[0x00] <- 42
    RAM[0x3c] = 0x0000000300000000;           RAM[0x3d] = 0x000000000000fc07;
//RAM[0xeeb0]: STORE 00 fc07  // RAM[0xfc07] <- REG[0x00]
    RAM[0x3e] = 0x00000002000000fd;           RAM[0x3f] = 0x0000000000000044;
//RAM[0xeeb2]: LOAD  fd 44    // REG[0xfd] <- 44
    RAM[0x40] = 0x0000000b00000000;           RAM[0x41] = 0x000000000000fbea;
//RAM[0xeeb4]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;}
    RAM[0x42] = 0x0000000b00000000;           RAM[0x43] = 0x000000000000001c;
//RAM[0xeeb6]: JUMP  00 1c    // if (REG[0x00] == REG[0x00]) {p <- 0xee90;} //
FIBONACCI
    RAM[0x44] = 0x0000000500000000f;          RAM[0x45] = 0x0000000f00000001;
//RAM[0xeeb8]: ADD   0f f1    // REG[0x0f] <- REG[0x0f] + REG[0x01]
    RAM[0x46] = 0x0000000200000000;           RAM[0x47] = 0x000000000000004e;
//RAM[0xeeba]: LOAD  00 4c    // REG[0x00] <- 4c
    RAM[0x48] = 0x0000000300000000;           RAM[0x49] = 0x000000000000fc07;
    RAM[0x4a] = 0x00000002000000fd;           RAM[0x4b] = 0x0000000000000050;
//RAM[0xeebe]: LOAD  fd 4e    // REG[0xfd] <- 4e
    RAM[0x4c] = 0x0000000b00000000;           RAM[0x4d] = 0x000000000000fbea;
//RAM[0xeec0]: JUMP  00 fbea  // if (REG[0x00] == REG[0x00]) {p <- 0xfbea;}
    RAM[0x4e] = 0x0000000b00000000;           RAM[0x4f] = 0x000000000000001c;
//RAM[0xeec2]: JUMP  00 1c    // if (REG[0x00] == REG[0x00]) {p <- 0xee90;} //
FIBONACCI
    RAM[0x50] = 0x0000000100000009;           RAM[0x51] = 0x00000000000000f4;
//RAM[0xeec4]: LOAD  09 f4    // REG[0x09] <- RAM[0xf4]
    RAM[0x52] = 0x0000000400000000;           RAM[0x53] = 0x0000000900000004;
//RAM[0xeec6]: MOVE  00 94    // REG[0x04] <- REG[0x09]
    RAM[0x54] = 0x0000000b00000000;           RAM[0x55] = 0x000000000000fbd2;
//RAM[0xeec8]: JUMP  00 fbd2  // if (REG[0x00] == REG[0x00]) {p <- 0xfbd2;} //
RET
```

5. Fatorial() iterativo para Brookshear; para inserir o valor de entrada necessita alterar o conteúdo da RAM[0x1] de *a* para o valor desejado.

```
RAM[ 0x0] = 0x20;         RAM[ 0x1] = a;      //RAM[ 0x0]: 2003 LOAD 0 03
RAM[ 0x2] = 0x21;         RAM[ 0x3] = 0x01;   //RAM[ 0x2]: 2101 LOAD 1 01
RAM[ 0x4] = 0x22;         RAM[ 0x5] = 0x01;   //RAM[ 0x4]: 2201 LOAD 2 01
RAM[ 0x6] = 0x2f;         RAM[ 0x7] = 0x01;   //RAM[ 0x6]: 2f01 LOAD f 01
RAM[ 0x8] = 0x30;         RAM[ 0x9] = 0x34;   //RAM[ 0x8]: 3034 STORE 0 34
RAM[ 0xa] = 0x31;         RAM[ 0xb] = 0x35;   //RAM[ 0xa]: 3135 STORE 1 35
RAM[ 0xc] = 0x32;         RAM[ 0xd] = 0x36;   //RAM[ 0xc]: 3236 STORE 2 36
RAM[ 0xe] = 0xb1;         RAM[ 0xf] = 0x32;   //RAM[ 0xe]: b132 JUMP 1 32
RAM[0x10] = 0x23;         RAM[0x11] = 0x01;   //RAM[0x10]: 2301 LOAD 3 01
RAM[0x12] = 0x10;         RAM[0x13] = 0x35;   //RAM[0x12]: 1035 LOAD 0 35
RAM[0x14] = 0xb3;         RAM[0x15] = 0x24;   //RAM[0x14]: b334 JUMP 3 24
RAM[0x16] = 0x14;         RAM[0x17] = 0x36;   //RAM[0x16]: 1436 LOAD 4 36
RAM[0x18] = 0x54;         RAM[0x19] = 0x42;   //RAM[0x18]: 5442 ADD 4 42
RAM[0x1a] = 0x34;         RAM[0x1b] = 0x36;   //RAM[0x1a]: 3436 STORE 4 36
RAM[0x1c] = 0x53;         RAM[0x1d] = 0x3f;   //RAM[0x1c]: 533f ADD 3 3f
RAM[0x1e] = 0x33;         RAM[0x1f] = 0x37;   //RAM[0x1e]: 3337 STORE 3 37
RAM[0x20] = 0x10;         RAM[0x21] = 0x37;   //RAM[0x20]: 1037 LOAD 0 37
RAM[0x22] = 0xb3;         RAM[0x23] = 0x12;   //RAM[0x22]: b310 JUMP 3 12
RAM[0x24] = 0x12;         RAM[0x25] = 0x36;   //RAM[0x24]: 1236 LOAD 2 36
RAM[0x26] = 0x10;         RAM[0x27] = 0x34;   //RAM[0x26]: 1034 LOAD 0 34
RAM[0x28] = 0xb1;         RAM[0x29] = 0x32;   //RAM[0x28]: b132 JUMP 1 32
RAM[0x2a] = 0x51;         RAM[0x2b] = 0x1f;   //RAM[0x2a]: 510f ADD 1 1f
RAM[0x2c] = 0x31;         RAM[0x2d] = 0x35;   //RAM[0x2c]: 3135 STORE 1 35
RAM[0x2e] = 0x10;         RAM[0x2f] = 0x35;   //RAM[0x2e]: 1035 LOAD 0 35
```

```
RAM[0x30] = 0xb1;          RAM[0x31] = 0x10;    //RAM[0x30]: b110 JUMP 1 10
RAM[0x32] = 0xc0;          RAM[0x33] = 0x00;    //RAM[0x32]: c000 HALT 0 00
```

6. Fibonacci() iterativo para Brookshear; para inserir o valor de entrada necessita alterar o conteúdo da RAM[0x1] de *a* para o valor desejado.

```
RAM[ 0x0] = 0x20;          RAM[ 0x1] = a;        //RAM[ 0x0]: 2003 LOAD 0 03
RAM[ 0x2] = 0x21;          RAM[ 0x3] = 0x01;     //RAM[ 0x2]: 2101 LOAD 1 01
RAM[ 0x4] = 0x22;          RAM[ 0x5] = 0x00;     //RAM[ 0x4]: 2200 LOAD 2 00
RAM[ 0x6] = 0x23;          RAM[ 0x7] = 0x01;     //RAM[ 0x6]: 2301 LOAD 3 01
RAM[ 0x8] = 0x2f;          RAM[ 0x9] = 0x01;     //RAM[ 0x8]: 2f01 LOAD f 01
RAM[ 0xa] = 0xb1;          RAM[ 0xb] = 0x16;     //RAM[ 0xa]: b116 JUMP 1 16
RAM[ 0xc] = 0x55;          RAM[ 0xd] = 0x23;     //RAM[ 0xc]: 5523 ADD  5 23
RAM[ 0xe] = 0x40;          RAM[ 0xf] = 0x32;     //RAM[ 0xe]: 4032 MOVE 0 32
RAM[0x10] = 0x40;          RAM[0x11] = 0x53;     //RAM[0x10]: 4053 MOVE 0 53
RAM[0x12] = 0x51;          RAM[0x13] = 0x1f;     //RAM[0x12]: 511f ADD  1 1f
RAM[0x14] = 0xb0;          RAM[0x15] = 0x0a;     //RAM[0x14]: b00a JUMP 3 0a
RAM[0x16] = 0x40;          RAM[0x17] = 0x34;     //RAM[0x16]: 4034 MOVE 0 34
RAM[0x18] = 0xc0;          RAM[0x19] = 0x00;     //RAM[0x18]: c000 HALT 0 00
```

7. Fatorial() iterativo para P-code; para inserir o valor de entrada necessita alterar o conteúdo de code[6].a de *a* para o valor desejado.

```
      code[ 0].f = 5;     code[ 0].l = 0;    code[ 0].a = 5;   //INT 0 5 // main
      code[ 1].f = 0;     code[ 1].l = 0;    code[ 1].a = 1;   //LIT 0 1
      code[ 2].f = 3;     code[ 2].l = 0;    code[ 2].a = 3;   //STO 0 3
      code[ 3].f = 0;     code[ 3].l = 0;    code[ 3].a = 1;   //LIT 0 1
      code[ 4].f = 3;     code[ 4].l = 0;    code[ 4].a = 4;   //STO 0 4
      code[ 5].f = 2;     code[ 5].l = 0;    code[ 5].a = 3;   //LOD 0 3
      code[ 6].f = 0;     code[ 6].l = 0;    code[ 6].a = a;   //LIT 0 3
      code[ 7].f = 1;     code[ 7].l = 0;    code[ 7].a = 13;  //OPR 0 13 // OPR
0 >=
      code[ 8].f = 7;     code[ 8].l = 0;    code[ 8].a = 10;  //JPC 0 10
      code[ 9].f = 1;     code[ 9].l = 0;    code[ 9].a = 0;   //OPR 0 0
      code[10].f = 2;     code[10].l = 0;    code[10].a = 3;   //LOD 0 3
      code[11].f = 0;     code[11].l = 0;    code[11].a = 1;   //LIT 0 1
      code[12].f = 1;     code[12].l = 0;    code[12].a = 2;   //OPR 0 2 // OPR
0 ADD
      code[13].f = 3;     code[13].l = 0;    code[13].a = 3;   //STO 0 3
      code[14].f = 2;     code[14].l = 0;    code[14].a = 3;   //LOD 0 3
      code[15].f = 2;     code[15].l = 0;    code[15].a = 4;   //LOD 0 4
      code[16].f = 1;     code[16].l = 0;    code[16].a = 4;   //OPR 0 4 // OPR
0 MUL
      code[17].f = 3;     code[17].l = 0;    code[17].a = 4;   //STO 0 4
      code[18].f = 6;     code[18].l = 0;    code[18].a = 5;   //JMP 0 5
```

8. Fatorial() recursivo para P-code; para inserir o valor de entrada necessita alterar o conteúdo de code[1].a de *a* para o valor desejado.

```
      code[ 0].f = 5;     code[ 0].l = 0;     code[ 0].a = 5;     //INT 0 5 //
main
      code[ 1].f = 0;     code[ 1].l = 0;     code[ 1].a = a;     //LIT 0 a
      code[ 2].f = 3;     code[ 2].l = 0;     code[ 2].a = 8;     //STO 0 8 //
passagem de parametro do main para fatorial
      code[ 3].f = 4;     code[ 3].l = 0;     code[ 3].a = 5;     //CAL 0 5
```

```
      code[ 4].f = 1;     code[ 4].l = 0;     code[ 4].a = 0;     //OPR 0 0 //
retorno do main
      code[ 5].f = 5;     code[ 5].l = 0;     code[ 5].a = 6;     //INT 0 6 //
factorial
      code[ 6].f = 2;     code[ 6].l = 0;     code[ 6].a = 3;     //LOD 0 3
      code[ 7].f = 0;     code[ 7].l = 0;     code[ 7].a = 1;     //LIT 0 1
      code[ 8].f = 1;     code[ 8].l = 0;     code[ 8].a = 11;    //OPR 0 11 //
OPR 0 <=
      code[ 9].f = 7;     code[ 9].l = 0;     code[ 9].a = 15;    //JPC 0 15
      code[10].f = 0;     code[10].l = 0;     code[10].a = 1;     //LIT 0 1
      code[11].f = 0;     code[11].l = 0;     code[11].a = 1;     //LIT 0 1
      code[12].f = 3;     code[12].l = 0;     code[12].a = 5;     //STO 0 5
      code[13].f = 3;     code[13].l = 1;     code[13].a = 4;     //STO 1 4
      code[14].f = 1;     code[14].l = 0;     code[14].a = 0;     //OPR 0 0 //
retorno da chamada do fatorial
      code[15].f = 2;     code[15].l = 0;     code[15].a = 3;     //LOD 0 3
      code[16].f = 0;     code[16].l = 0;     code[16].a = 1;     //LIT 0 1
      code[17].f = 1;     code[17].l = 0;     code[17].a = 3;     //OPR 0 3 //
OPR 0 SUB
      code[18].f = 3;     code[18].l = 0;     code[18].a = 9;     //STO 0 9 //
passagem de parametro
      code[19].f = 4;     code[19].l = 0;     code[19].a = 5;     //CAL 0 5 //
chamada recursiva
      code[20].f = 2;     code[20].l = 0;     code[20].a = 3;     //LOD 0 3
      code[21].f = 2;     code[21].l = 0;     code[21].a = 4;     //LOD 0 4
      code[22].f = 1;     code[22].l = 0;     code[22].a = 4;     //OPR 0 4
      code[23].f = 3;     code[23].l = 0;     code[23].a = 5;     //STO 0 5
      code[24].f = 2;     code[24].l = 0;     code[24].a = 5;     //LOD 0 5
      code[25].f = 3;     code[25].l = 1;     code[25].a = 4;     //STO 1 4
      code[26].f = 1;     code[26].l = 0;     code[26].a = 0;     //OPR 0 0 //
retorno da chamada do fatorial
```

9. Fibonacci() iterativo para P-code; para inserir o valor de entrada necessita alterar o conteúdo de code[1].a de *a* para o valor desejado.

```
      code[ 0].f = 5;     code[ 0].l = 0;     code[ 0].a = 8;     //INT 0 8 //
main
      code[ 1].f = 0;     code[ 1].l = 0;     code[ 1].a = a;     //LIT 0 a
      code[ 2].f = 3;     code[ 2].l = 0;     code[ 2].a = 3;     //STO 0 3
      code[ 3].f = 0;     code[ 3].l = 0;     code[ 3].a = 0;     //LIT 0 0
      code[ 4].f = 3;     code[ 4].l = 0;     code[ 4].a = 4;     //STO 0 4
      code[ 5].f = 0;     code[ 5].l = 0;     code[ 5].a = 1;     //LIT 0 1
      code[ 6].f = 3;     code[ 6].l = 0;     code[ 6].a = 5;     //STO 0 5
      code[ 7].f = 0;     code[ 7].l = 0;     code[ 7].a = 1;     //LIT 0 2
      code[ 8].f = 3;     code[ 8].l = 0;     code[ 8].a = 6;     //STO 0 6
      code[ 9].f = 2;     code[ 9].l = 0;     code[ 9].a = 3;     //LOD 0 3
      code[10].f = 0;     code[10].l = 0;     code[10].a = 1;     //LIT 0 1
      code[11].f = 1;     code[11].l = 0;     code[11].a = 11;    //OPR 0 11 //
OPR 11 <=
      code[12].f = 7;     code[12].l = 0;     code[12].a = 15;    //JPC 0 15
      code[13].f = 2;     code[13].l = 0;     code[13].a = 3;     //LOD 0 3
      code[14].f = 3;     code[14].l = 0;     code[14].a = 7;     //STO 0 7
      code[15].f = 2;     code[15].l = 0;     code[15].a = 6;     //LOD 0 6
      code[16].f = 2;     code[16].l = 0;     code[16].a = 3;     //LOD 0 3
      code[17].f = 1;     code[17].l = 0;     code[17].a = 10;    //OPR 0 10 //
OPR 10 <
      code[18].f = 7;     code[18].l = 0;     code[18].a = 30;    //JPC 0 30
      code[19].f = 2;     code[19].l = 0;     code[19].a = 4;     //LOD 0 4
```

```
      code[20].f = 2;     code[20].l = 0;     code[20].a = 5;     //LOD 0 5
      code[21].f = 1;     code[21].l = 0;     code[21].a = 2;     //OPR 0 2 //
OPR 2 ADD
      code[22].f = 2;     code[22].l = 0;     code[22].a = 5;     //LOD 0 5
      code[23].f = 3;     code[23].l = 0;     code[23].a = 4;     //STO 0 4
      code[24].f = 3;     code[24].l = 0;     code[24].a = 5;     //STO 0 5
      code[25].f = 2;     code[25].l = 0;     code[25].a = 6;     //LOD 0 6
      code[26].f = 0;     code[26].l = 0;     code[26].a = 1;     //LIT 0 1
      code[27].f = 1;     code[27].l = 0;     code[27].a = 2;     //OPR 0 2 //
OPR 2 ADD
      code[28].f = 3;     code[28].l = 0;     code[28].a = 6;     //STO 0 6
      code[29].f = 6;     code[29].l = 0;     code[29].a = 15;    //JMP 0 15
      code[30].f = 2;     code[30].l = 0;     code[30].a = 5;     //LOD 0 5
      code[31].f = 3;     code[31].l = 0;     code[31].a = 7;     //STO 0 7
      code[32].f = 1;     code[32].l = 0;     code[32].a = 0;     //OPR 0 0
```

10. Fibonacci() recursivo para P-code; para inserir o valor de entrada necessita alterar o conteúdo de code[1].a de *a* para o valor desejado.

```
      code[ 0].f = 5;     code[ 0].l = 0;     code[ 0].a = 5;     //INT 0 5 //
main
      code[ 1].f = 0;     code[ 1].l = 0;     code[ 1].a = a;     //LIT 0 a
      code[ 2].f = 3;     code[ 2].l = 0;     code[ 2].a = 8;     //STO 0 8 //
passagem de parametro do main para fibonacci
      code[ 3].f = 4;     code[ 3].l = 0;     code[ 3].a = 7;     //CAL 0 7
      code[ 4].f = 2;     code[ 4].l = 0;     code[ 4].a = 4;     //LOD 0 4
      code[ 5].f = 3;     code[ 5].l = 0;     code[ 5].a = 3;     //STO 0 3
      code[ 6].f = 1;     code[ 6].l = 0;     code[ 6].a = 0;     //OPR 0 0 //
retorno do main
      code[ 7].f = 5;     code[ 7].l = 0;     code[ 7].a = 6;     //INT 0 6 //
fibonacci
      code[ 8].f = 2;     code[ 8].l = 0;     code[ 8].a = 3;     //LOD 0 3
      code[ 9].f = 0;     code[ 9].l = 0;     code[ 9].a = 1;     //LIT 0 1
      code[10].f = 1;     code[10].l = 0;     code[10].a = 11;    //OPR 0 11 //
OPR 0 <=
      code[11].f = 7;     code[11].l = 0;     code[11].a = 17;    //JPC 0 17
      code[12].f = 2;     code[12].l = 0;     code[12].a = 3;     //LOD 0 3
      code[13].f = 2;     code[13].l = 0;     code[13].a = 3;     //LOD 0 3
      code[14].f = 3;     code[14].l = 0;     code[14].a = 5;     //STO 0 5
      code[15].f = 3;     code[15].l = 1;     code[15].a = 4;     //STO 1 4
      code[16].f = 1;     code[16].l = 0;     code[16].a = 0;     //OPR 0 0 //
retorno da chamada do fibonacci
      code[17].f = 2;     code[17].l = 0;     code[17].a = 3;     //LOD 0 3
      code[18].f = 0;     code[18].l = 0;     code[18].a = 1;     //LIT 0 1
      code[19].f = 1;     code[19].l = 0;     code[19].a = 3;     //OPR 0 3 //
OPR 0 SUB
      code[20].f = 3;     code[20].l = 0;     code[20].a = 9;     //STO 0 9 //
passagem de parametro
      code[21].f = 4;     code[21].l = 0;     code[21].a = 7;     //CAL 0 7 //
chamada recursiva
      code[22].f = 2;     code[22].l = 0;     code[22].a = 3;     //LOD 0 3
      code[23].f = 0;     code[23].l = 0;     code[23].a = 2;     //LIT 0 2
      code[24].f = 1;     code[24].l = 0;     code[24].a = 3;     //OPR 0 3 //
OPR 0 SUB
      code[25].f = 3;     code[25].l = 0;     code[25].a = 9;     //STO 0 9 //
passagem de parametro
      code[26].f = 2;     code[26].l = 0;     code[26].a = 4;     //LOD 0 4
      code[27].f = 3;     code[27].l = 0;     code[27].a = 3;     //STO 0 3
```

```
      code[28].f = 4;     code[28].l = 0;     code[28].a = 7;     //CAL 0 7 //
chamada recursiva
      code[29].f = 2;     code[29].l = 0;     code[29].a = 3;     //LOD 0 3
      code[30].f = 2;     code[30].l = 0;     code[30].a = 4;     //LOD 0 4
      code[31].f = 1;     code[31].l = 0;     code[31].a = 2;     //OPR 0 2
      code[32].f = 3;     code[32].l = 0;     code[32].a = 5;     //STO 0 5
      code[33].f = 2;     code[33].l = 0;     code[33].a = 5;     //LOD 0 5
      code[34].f = 3;     code[34].l = 1;     code[34].a = 4;     //STO 1 4
      code[35].f = 1;     code[35].l = 0;     code[35].a = 0;     //OPR 0 0 //
retorno da chamada do fibonacci
```