DESENVOLVIMENTO DE MÁQUINA VIRTUAL PARA SISTEMA EMBARCADO

Regimar Francisco dos Santos, regimar.santos@gmail.usf.edu.br
Renan Romão Oliveira, renan.oliveira@mail.usf.edu.br
Fábio Andrijauskas, fabio.andrijauskas@usf.edu.br

Universidade São Francisco, Av. Sen. Lacerda Franco, 360 - Centro, Itatiba - SP, 13250-400

Resumo: *Computadores são máquinas utilizadas nas mais variadas atividades, porém estas têm um funcionamento relativamente complexo, o que acaba por distanciar indivíduos que não são da área de computação a obterem conhecimento sobre o funcionamento e programação destas máquinas. Porém tais máquina são divididas em diferentes níveis, e têm sua complexidade mitigada quando analisadas a níveis mais baixos, o que pode ser um ótimo artifício didático, além de não existirem ferramentas consagradas no mercado com o objetivo de passarem tal tipo de conhecimento de forma lúdica. O presente trabalho visa o desenvolvimento de uma máquina virtual com linguagem de máquina própria, voltada para a didática e possibilitando o aprendizado de desenvolvimento de software e arquitetura de computadores por leigos ou iniciantes da área, partindo de um ambiente de baixo nível e, consequentemente, de menor complexidade. Para tal será desenvolvido um interpretador, que terá como plataforma de execução um sistema embarcado de baixo custo, mais especificamente um microcontrolador ESP32. A aplicação desenvolvida ainda será de código aberto, permitindo que terceiros desenvolvam extensões e melhorias, mantendo o ambiente atualizado.*

Abstract: *Computers are machines used in the most varied activities, but these have a relatively complex operation, which ends up distancing individuals who are not from the computing area to obtain knowledge about the functioning and programming of these machines. However, these machines are divided into different levels, and have their complexity mitigated when analyzed at lower levels, this can be a great didactic artifice, in addition to the fact that there are no established tools in the market with the objective of passing on this type of knowledge in a playful way. The present work aims at the development of a virtual machine with its own machine language, focused on didactics and enabling the learning of software development and computer architecture for dummies or beginners in the area, starting from a low level and, consequently, less complex environment.An interpreter will be developed with this purpose, it will have a low cost embedded system as its execution platform, more specifically an ESP32 microcontroller. The application developed will also be open source, allowing third parties to develop extensions and improvements, keeping the environment updated..*

Palavras-chave: *desenvolvimento de software, arquitetura de computadores, compilador, máquina virtual, sistema embarcado.*

1. INTRODUÇÃO

Computadores digitais são máquinas muito complexas, por conta de sua infinidade de componentes internos e teorias envolvidas em seu funcionamento, porém, cientistas da computação desenvolveram teorias e modelos, para descrever o funcionamento de tais máquinas, permitindo sua utilização ou mesmo, em alguns casos, programação por usuários que não possuam compreensão de todas as áreas do conhecimento e processos internos envolvidos nelas. De forma resumida, Tanenbaum, **et al** (2013) descreve os computadores como máquinas que podem resolver problemas para as pessoas, executando instruções que lhe são dadas, tal afirmação deixa claro como a codificação de programas e sua execução são processos essenciais dentro do campo da computação, porém o funcionamento de certa forma mistificado destas máquinas e de como executam instruções de código, faz com que leigos ou mesmo profissionais de campos relacionados à tecnologia, não demonstrem interesse pelo desenvolvimento de software, tornando esta uma área tida como difícil e reservada a poucos, uma vez que existem poucas ferramentas capazes de elevarem o aprendizado de programação de um nível mais básico para o intermediário, existem muitas aplicações voltadas para completos iniciantes como programadores com código em bloco, jogos com mecânicas baseadas em pseudocódigo, ou ferramentas para usuários mais avançados, como linguagens de programação de sintaxe simplificada, porém tais opções não oferecem a possibilidade de um usuário básico/intermediário produzir uma aplicação funcional, o que seria crucial para deixá-lo engajado.

O presente trabalho tem como objetivo a criação de uma máquina virtual, com uma linguagem de máquina própria visando a simplicidade e didática no que se refere a seu desenvolvimento e execução, mas mesmo assim permitindo o desenvolvimento de aplicações úteis e interessantes por meio desta. Tal produção será constituída de um interpretador que será uma máquina teórica e servirá de ambiente de execução para seu próprio código de

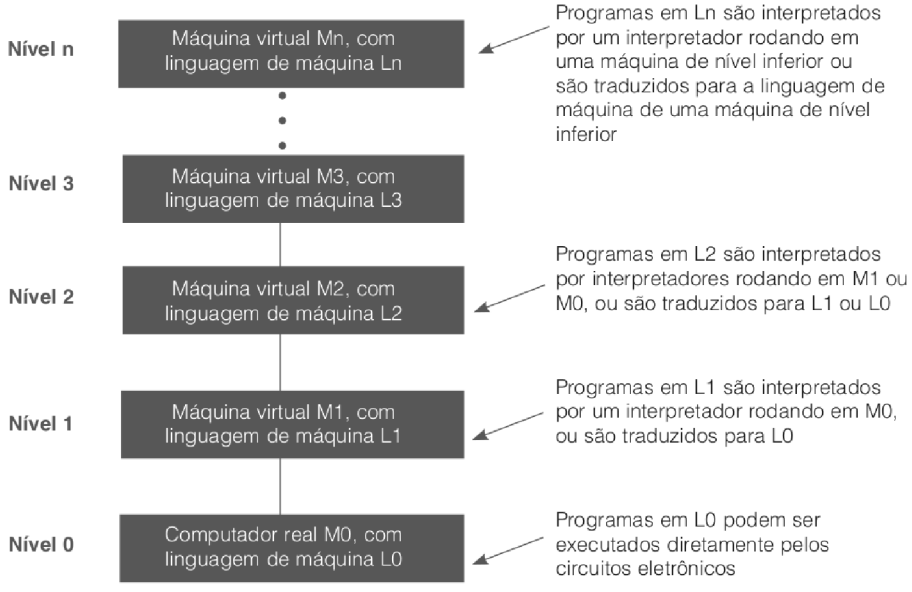
máquina, esta máquina será executada em um ESP32, um sistema embarcado de baixo custo, com baixo consumo de energia e um número considerável de interfaces de entrada e saída, como terminais digitais e analógicos, compatibilidade nativa com protocolos de comunicação variados e até mesmo interfaces de comunicação sem fio, permitindo ao usuário o desenvolvimento de aplicativos que realizam interações físicas, a compreensão do funcionamento de um computador digital tendo como base um dos níveis mais básicos e a evolução gradativa do seu conhecimento na área.

Esta pesquisa tem como justificativa o fato de que a maioria das linguagens de programação possuem estruturas rígidas e complexas, dificultando o aprendizado por iniciantes e até mesmo aqueles que já são iniciados na computação e querem expandir seus conhecimentos como hobbistas, talvez podendo se tornarem profissionais futuramente. Algo importante de se ter em vista é que o produto apresentado consiste apenas de parte de um sistema computacional, sendo um dos componentes mais básicos, estando enquadrado na chamada computação de baixo nível, sabendo disso, o sistema será criado desde o início para ser modular e expansível, e apesar de funcionar perfeitamente apenas com seus componentes mais básicos, não apresentará, de início, muitas das facilidades necessárias para os usuários mais iniciantes, como uma interface gráfica para operação deste e uma linguagem de montagem, ou *assembly* com sintaxe mais simples que aquela da linguagem de máquina.

2. LEVANTAMENTO BIBLIOGRÁFICO

Computadores digitais nada mais são do que máquinas capazes de executar determinadas instruções, que formam uma linguagem, denominada linguagem de máquina, Tanenbaum, **et al** (2013) define estes como máquinas multiníveis, isto porque são divididos em níveis, da forma que cada um desses níveis pode ser visto como uma máquina diferente e é capaz de executar as instruções de sua própria linguagem de máquina. O nível mais baixo de um computador é sempre o Nível lógico Digital que consiste de circuitos eletrônicos que realizam operações com valores binários. Por se tratar de eletrônica digital e, consequentemente, executarem instruções muito básicas e que não estão no campo de conhecimento de muitos, os projetistas de computadores desenvolvem uma máquina de nível superior ao Lógico Digital capaz de executar um conjunto mais restrito de instruções tendo como base as instruções do nível abaixo, a partir daí os níveis podem ser considerados máquinas virtuais, por se tratarem de entidades abstratas cada qual com a sua própria linguagem de máquina, a figura 1 apresenta a estrutura simplificada de uma máquina multinível.

Figura 1 - Máquinas multiníveis.



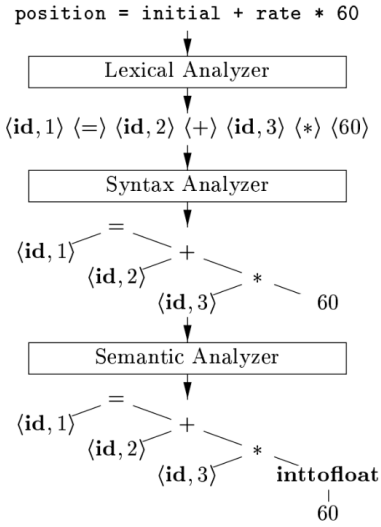
Fonte: Organização estruturada de computadores, 2014.

É possível escrever código na linguagem de uma máquina virtual e executá-lo nela como se a máquina de fato existisse, as máquinas mais distantes do nível do hardware, e que, consequentemente, são aquelas operadas por usuários comuns, executam linguagens chamadas de alto nível, que geralmente possuem um conjunto de instruções mais orientado aos usuários ou ao domínio para o qual a linguagem é utilizada. Ainda de acordo com os conceitos da organização de computadores por Tanenbaum, **et al** (2013), as máquinas virtuais são divididas em, compiladores, interpretadores e implementações híbridas, que nada mais são do que uma combinação de ambas as anteriores, todas essas abordagens são discutidas nas seções a seguir.

2.1. Compiladores e Interpretadores

Os compiladores são descritos por Aho, **et al** (2007) como programas que leem um programa escrito numa linguagem, a linguagem fonte, e a traduzem para outra linguagem, a linguagem alvo. Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte. A compilação é um processo complexo, que por convenção é dividido em duas etapas, análise e síntese. A análise tem a função de ler o programa escrito na linguagem fonte e dividi-lo de forma organizada de acordo com sua estrutura em uma representação intermediária, verificando assim se o código é válido e se haverá erros no programa gerado, esta etapa ainda pode ser subdividida em três: a análise léxica ou *lexer*, tem como objetivo identificar na entrada do usuário as sequências de caracteres que formam os símbolos da linguagem, chamados de lexemas ou tokens, esse processo geralmente é realizado utilizando expressões regulares; a análise sintática ou *parser*, utiliza os lexemas gerados no processo anterior como entrada, agrupa estes em frases gramaticais, de acordo com a estrutura sintática da linguagem, gerando assim a representação do programa inteiro por uma árvore gramatical, ou gramática livre de contexto, estrutura de dados ligada que descreve a execução e relação entre as estruturas do programa a ser desenvolvido; e, por último, a análise semântica, que complementa as anteriores realizando verificação de erros semânticos, como operações realizadas com variáveis de tipos inválidos, e coletando informações para o processo de síntese do programa-alvo. A seguir, na figura 2, é demonstrado o funcionamento de cada umas das etapas da análise, bem como mostrados exemplos de quais seriam suas respectivas saídas.

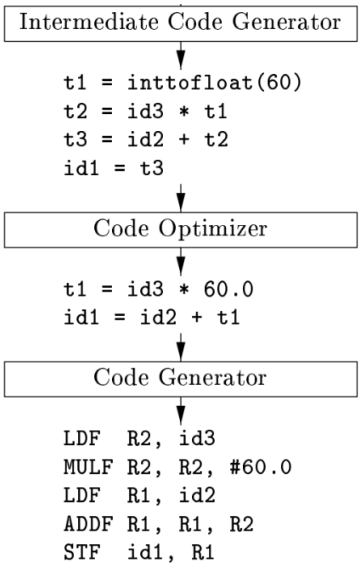
Figura 2 - Etapas da análise e seus resultados.



Fonte: Compilers: Principles, Techniques, & Tools, adaptado, 2007.

A síntese tem como objetivo gerar o código de máquina, que será de fato executado na máquina alvo, com base na árvore sintática gerada nas etapas da análise, aqui também existem subdivisões, porém estas variam dependendo da linguagem, com algumas podendo ser realizadas em uma única passada ou mesmo omitidas, de acordo com Aho, **et al** (2007), elas se resumem em: geração de código intermediário, consiste na geração de uma representação intermediária do programa baseada em instruções, porém independente da arquitetura de hardware, esta deve ser fácil de reproduzir e de traduzir no programa alvo; otimização de código, aqui o código intermediário é otimizado, tendo como resultado um código de máquina mais eficiente em tempo de execução, otimizações geralmente consistem em ações como, conversões de tipos das variáveis, antes mesmo da compilação, e remoção de registros temporários que se mostram desnecessários; geração de código, é a única fase necessariamente dependente da máquina alvo, aqui as representações intermediárias são efetivamente traduzidas em instruções de máquina, assim o compilador tem a responsabilidade de alocar memória para cada variável, gerenciar escopos e registradores do processador, dente outras. Exemplos populares de linguagens compiladas são C/C++, C# e Pascal. Na figura 3, a seguir, é mostrada de forma simplificada o funcionamento das etapas da síntese, como exemplos de resultados de cada uma.

Figura 3 - Etapas da síntese e seus resultados.



Fonte: Compilers: Principles, Techniques, & Tools, adaptado, 2007.

Os interpretadores utilizam o código escrito na linguagem fonte diretamente como dados de entrada, ou seja, interpretam cada instrução escrita em uma linguagem de alto nível, executando a sequência de instruções equivalentes em uma máquina de nível mais baixo, desta forma, diferentemente do que ocorre com a compilação, não é gerado um novo programa na linguagem de nível inferior e a máquina virtual de nível superior trabalha ativamente a toda execução do software. Exemplos populares de linguagens interpretadas são PHP, Python e JavaScript.

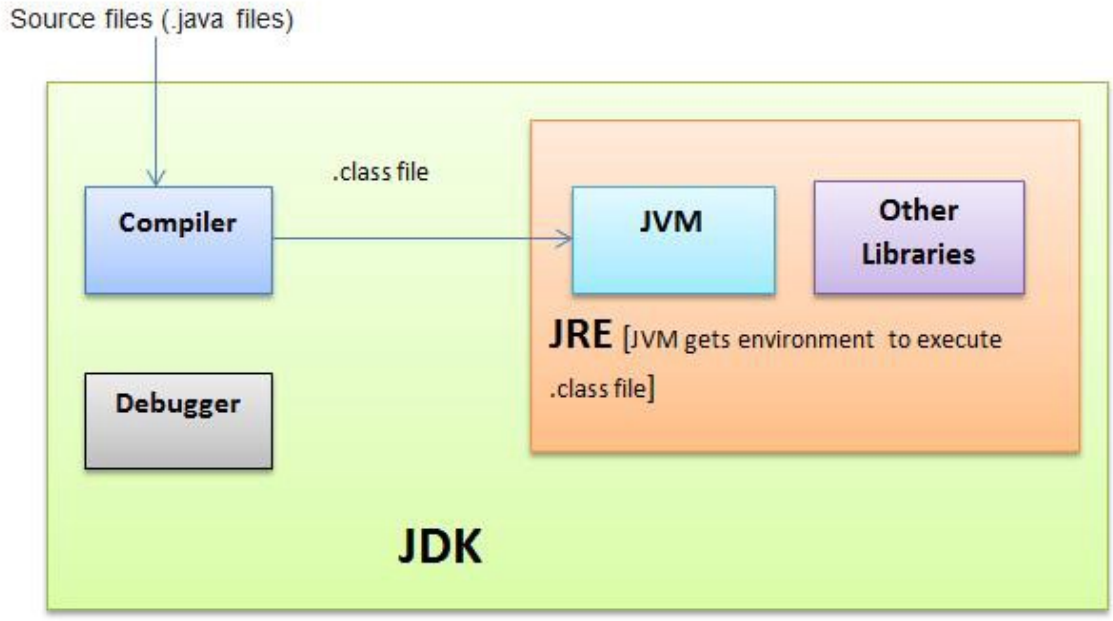
2.2. Implementações híbridas

Além dos tradicionais compiladores e interpretadores, outra abordagem de execução de software popular consiste em utilizar características de ambas as anteriores, GOUGH (2001) diz que esta é uma tendência do desenvolvimento de software que visa obter portabilidade de execução dos programas, por meio da compilação do código fonte em uma representação intermediária baseada em uma máquina abstrata, mais especificamente trata-se de uma abordagem que separa o *front-end* de um compilador, que tem como função compilar o código fonte para uma representação intermediária e verificar a semântica desta, e o *back-end* que utiliza este código intermediário para executar as instruções na máquina de destino, podendo esta execução ser por meio de compilação direta para linguagem de máquina, muitas vezes utilizando técnicas como compilação JIT (*Just in Time*) ou AOT (*Ahead of Time*), ou ser uma execução por meio de interpretador, que como já foi apresentado, é uma simulação da máquina virtual na máquina de destino, executando cada instrução do código intermediário, que neste caso é chamado de *bytecode*.

Na emulação de hardware virtual, as máquinas ainda podem ser segmentadas de acordo com a sua estrutura interna, sendo divididas em máquinas de arquitetura baseada em pilhas, e de arquitetura baseada em registradores. As arquiteturas baseadas em pilhas, de acordo com GOUGH (2001), tem como exemplos mais notáveis a Máquina P-Code utilizada no sistema operacional UCSD p-System, a JVM (Java Virtual Machine) desenvolvida pela Sun Microsystems e, atualmente, o ambiente de execução mais utilizado e portado do mundo, e o Motor de Execução .NET da Microsoft, como sua própria nomenclatura sugere, estas têm seu ciclo de execução baseada em estruturas de dados do tipo pilha ou LIFO (*Last In First Out* ou último a entrar primeiro a sair), utilizadas para armazenar e operar sobre variáveis em tempo de execução, tal tipo de implementação é exclusivo de máquinas abstratas. As arquiteturas baseadas em registradores, por sua vez, trabalham de maneira semelhante aos computadores físicos e utilizam registradores abstratos porém com a mesma funcionalidades daqueles encontrados nos chips de processadores, armazenar valores e realizar operações nestes para a execução das instruções, as máquinas virtual Dalvik e ART, criadas pelo Google para os sistemas Android, são exemplos de utilização deste método. De acordo com KAHN **et al** (2009), as arquiteturas de máquinas virtuais baseadas em pilhas têm sido mais utilizadas por conta de sua facilidade de implementação e por possuírem arquivos executáveis menores, o que são fatores muito importantes se considerado que estas geralmente devem ser executadas em diversos tipos diferentes de hardware, já as máquinas baseadas em registradores são mais performáticas por conta de executarem menos instruções do que aquelas baseadas em pilhas para realizarem as mesmas tarefas, o que foi crucial para a escolha do Google por tal abordagem, quando estava desenvolvendo o Android. Na figura 4, logo a seguir, é mostrado um diagrama que demonstra, de forma simplificada, o funcionamento do JDK (*Java Development Kit*), utilizado para compilar código fonte Java em bytecode, e do

JRE (*Java Runtime Environment*) utilizado para executar esse bytecode em diversas implementações da JVM, percebe-se como a plataforma Java utiliza de uma implementação híbrida para concretizar seu slogan "*write once, run anywhere"* (ou "escreva uma vez, execute em qualquer lugar", em português), com implementações de seu tempo de execução para as mais diversas plataformas.

Figura 4 - Diagrama simplificado do funcionamento da JVM.



Fonte: The Secret of Java- JDK, JRE, JVM difference, Medium, 2016.

3. METODOLOGIA

Para iniciar este trabalho foi realizado um estudo na literatura, e também em projetos semelhantes sobre técnicas e metodologias necessárias para desenvolvimento de uma máquina virtual para desenvolvimento de aplicações simples porém com retorno funcional para o usuário. Com base em todo o aprendizado assimilado, foram selecionadas ferramentas e técnicas das quais achamos mais adequadas para o desenvolvimento do projeto. Neste trabalho foi optado pelo desenvolvimento de uma máquina abstrata baseada em registradores, por conta desta abordagem ser mais simples e próxima de um hardware real, de forma que, exista mais material de pesquisa disponível, incluindo materiais de arquiteturas de hardware que podemos tomar como base. A aplicação desenvolvida consistirá, no entanto, apenas do back-end de um sistema mais completo para uma linguagem de programação interpretada, pois diferentemente dos exemplos abordados na literatura e utilizados comercialmente, esta irá consistir apenas do componente de execução de código da máquina virtual (*bytecode*), e não apresentando funcionalidades mais complexas como *Garbage Collector* e execução de códigos no paradigma de Programação Orientada a Objetos. Apesar das limitações apresentadas, o design da aplicação trabalhada será completamente modular, assim como descrito nas especificações de máquinas multiníveis encontradas na literatura, permitindo que sejam adicionados novos níveis ao sistema posteriormente, deixando este mais completo e com mais possibilidades de implementação.

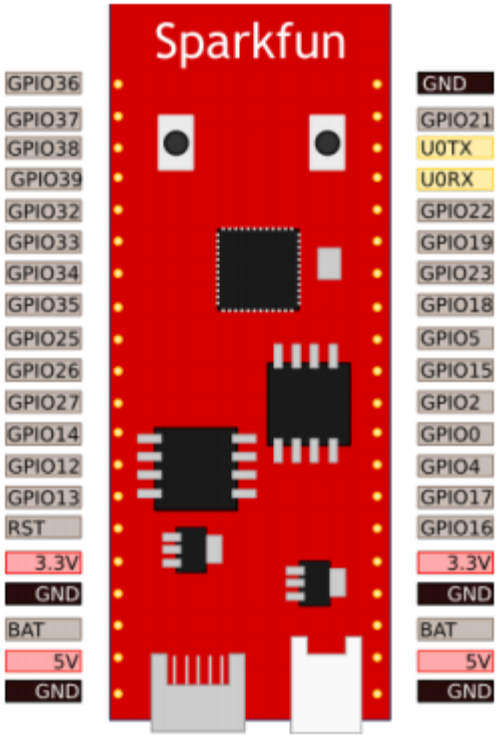
3.1. Plataforma de hardware

Como especificado na teoria, o sistema desenvolvido tem como hardware de execução um sistema embarcado, mais precisamente optamos por utilizar um microcontrolador da série ESP32, produzido pela empresa chinesa Espressif Systems e voltado especificamente para aplicações de IoT (*Internet of Things*). Os microcontroladores são equipamentos programáveis de pequeno porte e de baixo custo, desenvolvidos para atuar sobre a administração de ações e eventos, são compostos basicamente por uma unidade processadora, memórias, entradas e saídas, controle temporal e conversores analógicos e digitais entre outros, que tem como função realizar ações de controle de maneira remota em sistemas embarcados. Esse tipo de dispositivo, segundo SILVA (2007), surgiu em meados da década de 70, criado por uma equipe da Texas Instruments, derivado dos então microprocessadores criados para realizar cálculos e posteriormente tomada de decisões, foi incorporado memórias e outros componentes através de um chip, evoluindo desde então e resultando nos equipamentos que conhecemos atualmente. A característica integradora dos microcontroladores por terem ligações com os meios

externos através de suas entradas e saídas os tornam muito práticos nas execuções de funções complexas, principalmente na automação e ou automatização, sua conectividade também é algo fundamental para explicar a grande difusão e aplicações desses aparatos, resultando na capacidade de atuar junto a internet, apresentando se como a melhor opção para desenvolver um sistema de controle e monitoramento integrado com outros dispositivos, podendo servir de host para uma página web, que baseado em linguagens elementares como o Hiper Text Markup Language (HTML), possibilite o usuário manter a supervisão e gestão dos equipamentos integrados ao microcontrolador remotamente, além do potencial em realizar ajustes e parametrizações de maneira simples e eficiente (SANTOS, JUNIOR, 2019, 22).

O ESP32 é amplamente utilizado para desenvolvimento de sistemas embarcados móveis, por possuir uma arquitetura RISC (*acrônimo de Reduced Instruction Set Computer*) de 32 bits, microcontrolador dual-core com clock máximo de 240 MHZ com baixo consumo e uma excelente performance, possui grande capacidade de memória ROM de 448 KB usada para boot e funções principais e memória SRAM de 520 KB usada para dados e instruções de programas. O ESP32 possui muitos recursos o que torna-o muito interessante quando seu uso destina a internet das coisas, por possuir 34 GPIO, 3 SPI, 2 I2S, 18 canais ADC, 3 UART, 10 pinos de leitura capacitiva e PWM, na figura 5 podemos ver os respectivos pinos do ESP32, ele pode ser utilizado de forma completamente autônoma ou de modo escravo para uma MCU (*Microcontroller Unit*) host, dentre as interfaces de comunicação, podemos destacar o Bluetooth híbrido (clássico e BLE) e Wi-Fi, das quais as iterações podem ser feitas através das interfaces, SPI/ SDIO ou I2C/UART, isso o faz especial para controles de dispositivos com a atual revolução da Tecnologia 4.0 a um baixo custo.

Figura 5 - ESP32 e sua pinagem.



Fonte: Kolban (2018).

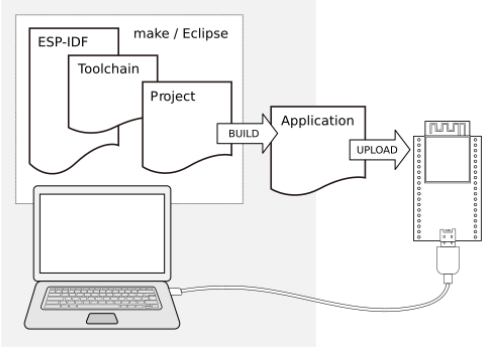
3.2. Plataforma de software

O módulo ESP32 é muito difundido entre os desenvolvedores, e portanto permite a utilização de diversas linguagens de programação como C/C++, Javascript, Java, Python, entre outras, o qual pode ser desenvolvido através do programa SDK (*Software Development Kit*) fornecido pela própria fabricante do dispositivo, ou pode ser programado através da interface do Software Arduino IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado). O Arduino IDE é um ambiente multiplataforma de desenvolvimento escrito em Java e derivada dos projetos Processing e Wiring, para poder programar o ESP32 usando o Arduino IDE é necessário instalar biblioteca específica para o NodeMCU-32S. No entanto neste trabalho foi optado pela utilização do ESP-IDF (*Espressif IoT Development Framework*) ferramenta oficial de desenvolvimento para o ESP32 fornecida pela Espressif Systems, juntamente com outros componentes de um ambiente de desenvolvimento tradicional para sistemas embarcados, com todos os seus componentes sendo fornecidos pelo

fabricante e/ou de código aberto, na figura 6 temos o diagrama do processo do desenvolvimento de firmware para o ESP32, o qual se assemelha muito a qualquer ambiente de desenvolvimento tradicional para sistemas embarcados, os componentes que formam tal processo estão listados e explicados a seguir.

- **Toolchain:** conjunto de ferramentas de software utilizado para compilar código para arquiteturas diferentes daquela da máquina que está sendo utilizada para programar, fornecida pela própria Espressif Systems;
- **Ferramentas de compilação:** softwares utilizados para executar a compilação de código fonte de forma automatizada e/ou otimizada, como o CMake e o Ninja, ambos ferramentas de código aberto e recomendados pela Espressif Systems;
- **ESP-IDF:** conjunto de ferramentas fornecido pela fabricante para manusear a toolchain e outras ferramentas e gerar *firmware* (binários executáveis ou código de máquina) compatível com a plataforma ESP32;
- **Editor de código:** software utilizado para facilitar escrita de código fonte, auxiliar com erros de sintaxe e automatizar a execução das ferramentas de compilação, optamos por utilizar o Eclipse, uma ferramenta gratuita e de código aberto e recomendada pela Espressif Systems.

Figura 6 - Diagrama do processo de desenvolvimento de firmware para o ESP32.



Fonte: ESP-IDF Programming Guide, latest version, 2020.

as principais vantagens para programação do ESP32 utilizando o ESP-IDF em relação a utilização do Arduino IDE, são o suporte a todas features disponíveis no sistema, incluindo Bluetooth, Flash Encryption e Secure Boot, e a configuração total do sistema, como eFuses, clock, watchdogs, memória dinâmica para Wi-Fi e timers.

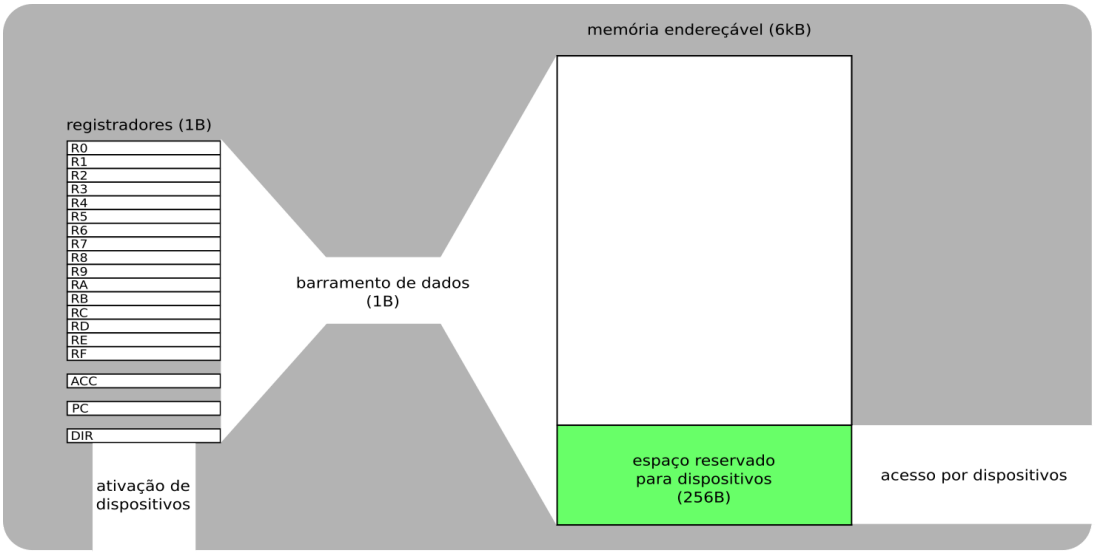
A linguagem de programação padrão utilizada para desenvolvimento para o ESP32 com o ESP-IDF, e que é utilizada neste trabalho, é a linguagem C, esta tem uma sintaxe e funcionalidades familiares para a maioria dos programadores, além de uma ótima eficiência de execução, por ser compilada e gerar código de máquina muito otimizado. Apesar de ser considerada uma linguagem de programação de alto nível, a linguagem C permite a realização de operações de baixo nível, como manuseio de ponteiros e alocação dinâmica de memória, que, apesar de serem funcionalidades relativamente avançadas da linguagem, estão presentes em sua biblioteca padrão e são muito úteis para o desenvolvimento de uma máquina virtual, uma vez que com esta é possível criar estruturas de dados customizadas (structs), para o armazenamento dos dados necessários na execução dos programas, tanto variáveis definidas pelo programador quanto os valores utilizados na execução das aplicações, como os endereços de retorno após a execução de rotinas, e armazenar essas estruturas em memória alocada em tempo de execução, na linguagem escolhida esta tarefa é factível de forma rápida e sem gerar muitos metadados, dados utilizados para manuseamento do programa, otimizando a utilização da memória que é um recurso um pouco escasso em um sistema embarcado, além de permitir futuras otimizações de desempenho e alterações de design mais facilmente, em contrapartida essa abordagem faz com que seja necessária a verificação dos recursos utilizados e disponíveis, para não causar erros como um estouro da pilha de execução.

4. RESULTADOS

Utilizando o conteúdo teórico e as técnicas abordadas neste trabalho foi possível o desenvolvimento de uma máquina virtual de baixo nível, de arquitetura simples mas capaz de ser extendida e executar aplicações de maneira determinística em um microcontrolador ESP32. A arquitetura desta máquina está exemplificada no diagrama da figura 7, neste pode-se ver os principais componentes que fazem parte desta; os registradores, que são utilizados para armazenamento de valores temporários durante a execução das instruções, contando inclusive com registradores especiais como o acumulador (ACC) utilizado para operações aritméticas, o Contador de

Programa (PC) que aponta para a próxima instrução a ser realizada e o Registrador de Interrupção dos Dispositivos (DIR), utilizado na operação dos dispositivos conectados à máquina virtual; conectada aos registradores pelo barramento de dados está a memória endereçável de 6 Kilobytes, na qual ficam armazenados tanto os códigos do programa em execução, quanto os dados trabalhados por este programa, além de ainda contar com um espaço reservado, de 256 bytes, para a utilização por dispositivos conectados. Mais detalhes sobre os componentes da arquitetura estão nos parágrafos a seguir.

Figura 7 - Diagrama da arquitetura da máquina virtual desenvolvida



Fonte: Figura de autoria própria.

A arquitetura da máquina virtual desenvolvida é baseada em **registradores**, assim como àquela de um processador físico, esta tem dezenove registradores de dezesseis bits cada um, sendo estes:

- Dezesseis registradores de propósito geral (R0 a RF, indexados por um algarismo hexadecimal), chamado de GRs ou *General Registers*, estes são utilizados livremente pelo programador durante o desenvolvimento das aplicações;
- Um acumulador (ACC) utilizado pela máquina para realização de operações aritméticas, este ainda pode ser referenciado diretamente pelo usuário em instruções que utilizam GRs;
- Um contador de programa, chamado de PC ou *Program Counter*, utilizado pela máquina virtual para saber qual a posição de memória da próxima instrução que será executada, este registro é incrementado automaticamente a cada instrução e modificado em instruções de pulo de execução do programa;
- Um registrador de interrupções dos dispositivos, chamado de DIR ou *Device Interrupts Register*, utilizado para ativar as interrupções de dispositivos externos, permitindo que estes tomem controle da aplicação, o funcionamento dos dispositivos é explicado na seção dispositivos externos.

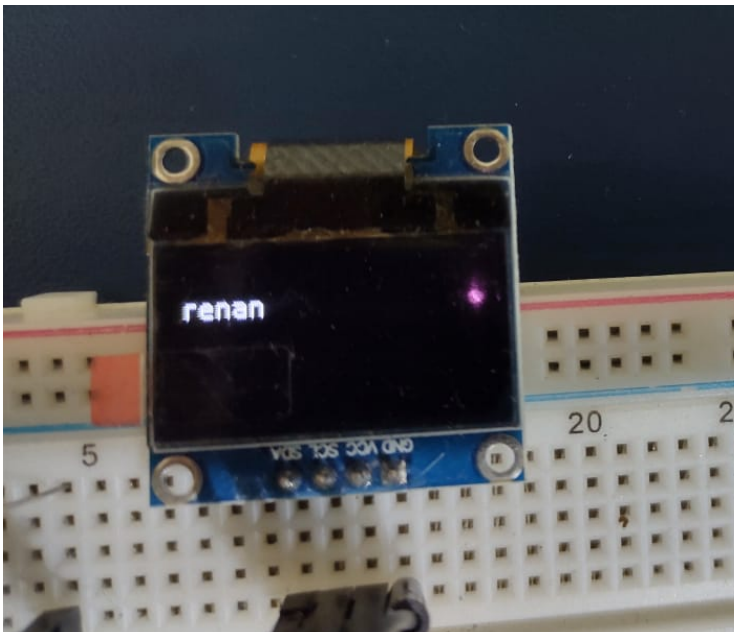
A máquina virtual também possui seis kilobytes, ou 65536 bytes, de memória endereçável, ou seja, todos os endereços possíveis do contador de programa de dezesseis bits podem ser utilizados, porém os 256 primeiros bytes da memória (endereços 0x0000 a 0x0099 em hexadecimal) são utilizados para acesso por dispositivos, sendo assim, o valor do contador de programa no início da execução do código é 256 (0x0100 em hexadecimal). A aplicação desenvolvida também foi baseada na arquitetura de Von Neumann, portanto as instruções do código de máquina e os dados utilizados no programa ficam armazenados na mesma memória.

A aplicação desenvolvida também implementa o conceito de **dispositivos externos virtuais**, estes consistem de programas que podem utilizar parte da memória da máquina para realizar operações que envolvam a interação da máquina com o meio externo, também conhecidas com operações de entrada e saída (*I/O* ou *Input/Output*). Como foi comentado anteriormente a área reservada para uso dos dispositivos externos consiste dos bytes nos endereços zero a 255, sendo que cada dispositivo reserva uma quantidade de bytes. Os dispositivos são conectados a máquina antes da execução do código de fato e funcionam basicamente tomando controle da execução após uma instrução por meio de uma interrupção e utilizando a memória reservada a eles, as interrupções de dispositivos são chamadas pelo próprio programa, ou seja são interrupções internas, colocando um bit relativo à interrupção como alto (*high* ou *1*) no registrador de interrupções dos dispositivos (o DIR), assim como no caso da memória, a quantidade de bits reservados por dispositivo varia, e sempre se inicia do bit menos significativo do registrador, na ordem em que os dispositivos são conectados à máquina.

Apenas dispositivos de saída foram utilizados em testes na máquina virtual, pois, uma vez que todas as interrupções são internas, dispositivos de entrada não são tão efetivos, já que as chamadas seriam síncronas e partiriam do próprio programa executado, tendo isso em vista que a implementação de sub-rotinas e interrupções externas está entre as possíveis melhorias futuras para a arquitetura da máquina virtual.

O dispositivo mais utilizado e deixado nativamente na máquina foi um mini display oled, que pode ser observado na figura 8 a seguir, no qual é possível escrever linhas de texto de forma semelhante a um terminal, a opção pela utilização deste se deu por conta de ser uma forma muito prática de obter saída dos programas durante a realização de testes.

Figura 8 - Display OLED utilizado como dispositivo de saída da máquina virtual, em funcionamento.



Fonte: Figura de autoria própria.

O **código de máquina** da arquitetura desenvolvida consiste de 37 instruções, cada qual com um código de operação de um byte, além de seus parâmetros que variam tamanho dependendo da instrução, existindo desde instruções sem parâmetros, que têm apenas um byte de tamanho, até instruções com um valor literal, de dezesseis bits, e um endereço de memória, também de dezesseis bits, totalizando cinco bytes de tamanho total da instrução, o tempo de execução da máquina é capaz de lidar com as instruções de diferentes tamanhos automaticamente. As instruções implementadas foram categorizadas em cinco grupos diferentes, dependendo de suas funcionalidades, são eles:

- **Instruções de movimentação:** têm a função de mover valores entre diferentes localidades, como registradores e posições de memória;
- **Instruções de pulo:** têm como função realizar pulos na execução do programa, ou seja, alterar o valor do contador de programa para outra posição da memória dependendo de certas condições;
- **Instruções de adição:** realizam somas de valores no valor de registradores;
- **Instruções de subtração:** realizam subtrações de valores no valor de registradores;
- **Instruções de incremento/decremento:** incrementa ou decrementa o valor de registradores;
- **Instruções de mudança de bits:** realizam operações binárias de mudança de bits no valor de registradores;
- **Instruções de operações lógicas**: realizam operações de lógica binária (AND, OR, XOR e NOT) no valor de registradores;

Por conta de ter sido implementada de uma maneira que preza pela simplicidade, o ciclo de instruções da máquina, ou ciclo de busca-execução, é realizado por uma única função, que chama uma macro da linguagem, esta macro, que pode ser observada na figura 9, primeiramente decodifica a instrução, apenas comparando os bytes mais e menos significativos de seu opcode com valores preestabelecidos para definir a categoria da instrução e, logo depois, a instrução em particular dentro da sua categoria, realizando este processo em uma estrutura de seleção na forma de árvore, após a decodificação da instrução, a operação é executada instantaneamente com os operandos necessários para ela, já que, uma vez que os tipos de operandos são definidos por instrução, nenhuma verificação necessita ser realizada. Ainda é possível ver no próprio código,

qual instrução de máquina é executada por cada linha da macro, e quais tipos de operandos essas utilizam, por meio dos comentários que estão na frente de cada uma das linhas, aqui as instruções estão representadas na forma de mnemônicos semelhantes aos de uma linguagem *assembly*, assim este trecho crítico do código já conta com uma forma de documentação, básica porém eficiente, junto do próprio código.

Figura 9 - Macro que decodifica e executa todas as instruções da arquitetura da máquina virtual.

```
#define DECODE_INSTRUCTIONS(a,b) \
    a(0x0, \
        b(0x0, WORD; BYTE; Ry = x;);                    /* MOV lit, reg */ \
        b(0x1, BYTE; BYTE; Ry = Rx;);                   /* MOV reg, reg */ \
        b(0x2, BYTE; WORD; write_word(y, Rx););         /* MOV reg, addr */ \
        b(0x3, WORD; BYTE; Ry = read_word(x););         /* MOV addr, reg */ \
        b(0x4, WORD; WORD; write_word(y, x););          /* MOV lit, addr */ \
        b(0x5, BYTE; BYTE; Ry = read_word(Rx););        /* MOV addr_ptr(reg), reg */ \
        b(0x6, BYTE; BYTE; write_word(Ry, Rx););        /* MOV reg, addr_ptr(reg) */ \
    ) \
    a(0x1, \
        b(0x0, WORD; WORD; if (x != _ACC) _PC = y;);    /* JNE lit, addr */ \
        b(0x1, BYTE; WORD; if (Rx != _ACC) _PC = y;);   /* JNE reg, addr */ \
        b(0x2, WORD; WORD; if (x == _ACC) _PC = y;);    /* JEQ lit, addr */ \
        b(0x3, BYTE; WORD; if (Rx == _ACC) _PC = y;);   /* JEQ reg, addr */ \
        b(0x4, WORD; WORD; if (x < _ACC) _PC = y;);     /* JLT lit, addr */ \
        b(0x5, BYTE; WORD; if (Rx < _ACC) _PC = y;);    /* JLT reg, addr */ \
        b(0x6, WORD; WORD; if (x > _ACC) _PC = y;);     /* JGT lit, addr */ \
        b(0x7, BYTE; WORD; if (Rx > _ACC) _PC = y;);    /* JGT reg, addr */ \
        b(0x8, WORD; WORD; if (x <= _ACC) _PC = y;);    /* JLE lit, addr */ \
        b(0x9, BYTE; WORD; if (Rx <= _ACC) _PC = y;);   /* JLE reg, addr */ \
        b(0xa, WORD; WORD; if (x >= _ACC) _PC = y;);    /* JGE lit, addr */ \
        b(0xb, BYTE; WORD; if (Rx >= _ACC) _PC = y;);   /* JGE reg, addr */ \
        b(0xc, WORD; _PC = x;);                         /* UCJ addr */ \
    ) \
    a(0x2, \
        b(0x0, BYTE; BYTE; _ACC = Rx + Ry;);            /* ADD reg, reg */ \
        b(0x1, WORD; BYTE; _ACC = x + Ry;);             /* ADD lit, reg */ \
    ) \
    a(0x3, \
        b(0x0, BYTE; BYTE; _ACC = Rx - Ry;);            /* SUB reg, reg */ \
        b(0x1, WORD; BYTE; _ACC = x - Ry;);             /* SUB lit, reg */ \
    ) \
    a(0x4, \
        b(0x0, BYTE; Rx++;);                            /* INC reg */ \
        b(0x1, BYTE; Rx--;);                            /* DEC reg */ \
    ) \
    a(0x5, \
        b(0x0, BYTE; WORD; Rx = Rx << y;);              /* LSF reg, lit */ \
        b(0x1, BYTE; BYTE; Rx = Rx << Ry;);             /* LSF reg, reg */ \
        b(0x2, BYTE; WORD; Rx = Rx >> y;);              /* RSF reg, lit */ \
        b(0x3, BYTE; BYTE; Rx = Rx >> Ry;);             /* RSF reg, reg */ \
    ) \
    a(0x6, \
        b(0x0, BYTE; WORD; _ACC = Rx & Ry;);            /* AND reg, lit */ \
        b(0x1, BYTE; BYTE; _ACC = Rx & y;);             /* AND reg, reg */ \
        b(0x2, BYTE; WORD; _ACC = Rx | Ry;);            /* OR  reg, lit */ \
        b(0x3, BYTE; BYTE; _ACC = Rx | y;);             /* OR  reg, reg */ \
        b(0x4, BYTE; WORD; _ACC = Rx ^ Ry;);            /* XOR reg, lit */ \
        b(0x5, BYTE; BYTE; _ACC = Rx ^ y;);             /* AND reg, reg */ \
        b(0x6, BYTE; _ACC = ~Rx;);                      /* NOT reg */ \
    )
```

Fonte: Figura de autoria própria.

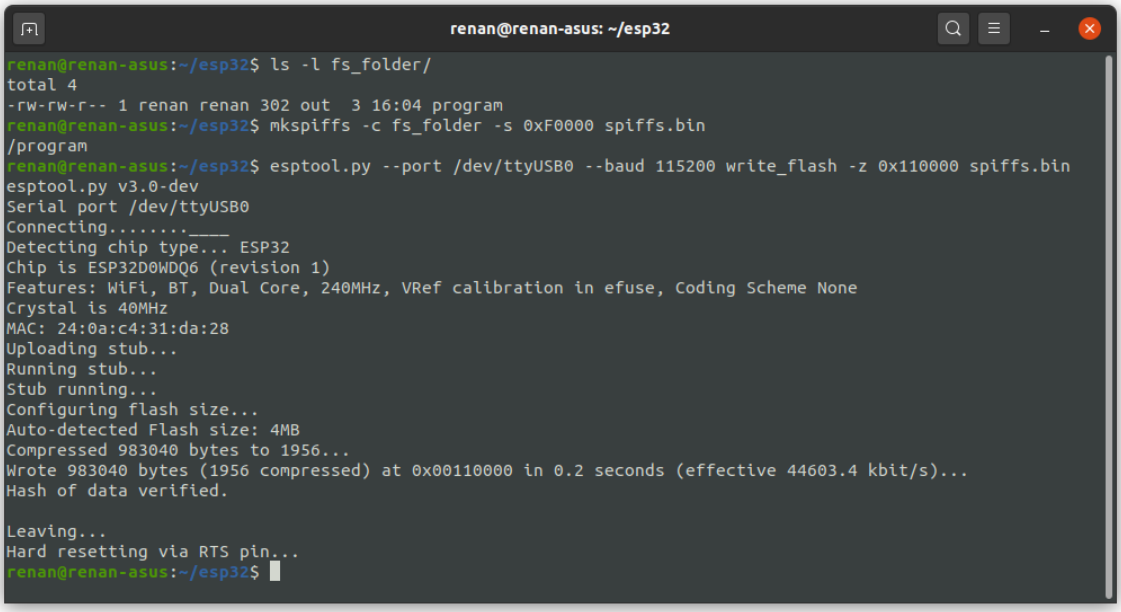
Após a execução de cada instrução, uma função é chamada para verificar se alguma interrupção de dispositivo foi chamada, em caso positivo a máquina verifica quais dispositivos foram acionados realizando operações de máscara de bits no registrador de interrupção de dispositivos (o DIR), e passa a execução do programa para eles realizarem as operações necessárias com seus respectivos trechos de memória mapeados.

Esta implementação, utilizando-se de metaprogramação, as macros no caso, se mostrou muito eficiente e facilmente extensível, uma vez que a maioria das instruções foram adicionadas e muitas alterações na execução de instruções foram realizadas após o ciclo de execução da máquina já estar completamente implementado, provando que esta é uma maneira completamente eficaz de se implementar o funcionamento de um processador virtual, já que este realiza uma vasta gama de tarefas simples, que muitas vezes precisam ser modificadas, ou incrementadas.

Foi implementado um **sistema de arquivos** com persistência na aplicação da máquina virtual, permitindo que o código a ser executado na mesma possa ser carregado de um computador por meio de cabo e que tal código permaneça gravado no dispositivo mesmo após este ser desligado e ligado novamente. O sistema de arquivo implementado foi o SPIFFS (*SPI Flash File System*), que é de código aberto, foi desenvolvido

especificamente para funcionar na memória flash de dispositivos embarcados e além disso tem suporte nativo pela plataforma do ESP32. Com tal funcionalidade implementada, tudo que precisa ser feito para inserir código na máquina virtual, por meio de um cabo USB é exemplificado na figura 10, nesta é mostrada a utilização da ferramenta gratuita *mkspiffs*, que compacta uma pasta, aqui de nome de *fs_folder*, contendo apenas um arquivo, que sempre deve ser chamado *program* e é o código do programa em formato binário, para uma imagem de partição SPIFFS, que aqui se chama *spiffs.bin* e tem um tamanho de 960 Kilobytes ou 0xF0000 bytes em hexadecimal, este tamanho deve ser sempre o mesmo pois é predefinido na tabela de partições do sistema, logo após, é utilizada a ferramenta de linha de comando do ESP-IDF *esptool.py* para a gravação da imagem gerada na flash da placa, no *offset* correto, 0x110000 em hexadecimal, que também é um valor que foi definido na tabela de partições. Apesar dos comandos no exemplo terem sido executados em um sistema operacional Linux, todas as ferramentas utilizadas estão disponíveis para a maioria dos sistemas operacionais de computadores desktop populares, como o Microsoft Windows e o macOS da Apple.

Figura 10 - Ferramentas utilizadas para enviar código à máquina virtual por cabo USB.



Fonte: Figura de autoria própria

Até o momento esta é a única forma de enviar código para a máquina virtual, porém outras maneiras podem ser desenvolvidas futuramente, uma vez que o sistema de arquivos pode ser acessado por código inclusive para a escrita, e levando em conta as diversas opções de conectividade oferecidas pelo hardware da ESP32, como Wi-Fi e Bluetooth, uma forma de gravação de código sem fios, provavelmente atrelada a um ambiente de desenvolvimento, futuramente poderia trazer muito mais praticidade aos usuários da aplicação.

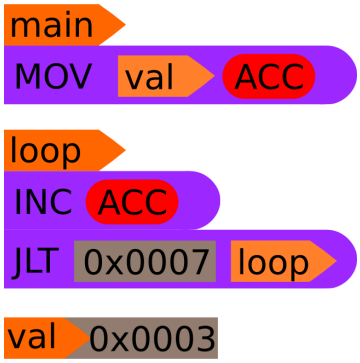
Era parte do planejado para o projeto da plataforma de desenvolvimento embarcado a criação de uma **interface para programação visual**, utilizando de uma linguagem que chamamos de *assembly visual*, esta combinaria a facilidade de das linguagens de programação em blocos, com a robustez e simplicidade do *assembly*, também chamado de linguagem de montagem, linguagem na qual as instruções da arquitetura da máquina são traduzidas para código chamados de mnemônicos, que visam ser mais simples do que decorar todos os opcodes possíveis em uma arquitetura, uma vez que remetem a funcionalidade da instrução. O compilador, ou *assembler*, da linguagem de máquina de nossa máquina virtual não chegou a ser desenvolvido, por limitações de tempo no projeto, porém a linguagem em si, com todos os seus mnemônicos e funcionalidades, foi definida, de forma que é possível traduzir um programa para esta, mesmo que tal implementação não seja funcional esta se mostra de muito mais fácil compreensão. Com a sintaxe desenvolvida, todas as 37 instruções foram simplificadas em dezenove mnemônicos, uma vez que as instruções variam de acordo com o tipo de parâmetro utilizado, mesmo que a funcionalidade destas seja basicamente a mesma, são esses:

- **MOV**: para instruções onde valores são transportados entre localizações de memória;
- **JNE**: para instruções de pulo condicional comparando se valores são diferentes;
- **JEQ**: para instruções de pulo condicional comparando se valores são iguais;
- **JLT**: para instruções de pulo condicional comparando se valores são menores;
- **JGT**: para instruções de pulo condicional comparando se valores são maiores;

- **JLE**: para instruções de pulo condicional comparando se valores são menores ou iguais;
- **JGE**: para instruções de pulo condicional comparando se valores são maiores ou iguais;
- **UCJ**: para a instrução de pulo incondicional;
- **ADD**: para instruções de adicionar valores;
- **SUB**: para instruções de subtrair valores;
- **INC**: para a instrução de incrementar registradores;
- **DEC**: para a instrução de decrementar registradores;
- **LSF**: para instruções de bit shifting à esquerda;
- **RSF**: para instruções de bit shifting à direita;
- **AND**: para instruções de e lógico;
- **OR**: para instruções de ou lógico;
- **XOR**: para instruções de ou exclusivo lógico;
- **NOT**: para a instrução de não lógico;
- **HLT**: para a instrução de terminar a execução de um programa.

Além dos mnemônicos outra funcionalidade muito útil do assembly são os labels, que nada mais são do que nomes dados a certo endereço de memória, aqui os labels podem ser usados tanto para nomear trechos do código, e realizar pulos condicionais para a localidade destes trechos, quanto para inserir valores constantes diretamente na memória, como se fossem variáveis em uma linguagem de programação de alto nível, desta forma não é necessário manter o endereço físico de tais localidades da memória anotado, como acontece quando se programa diretamente em código de máquina. A seguir, na figura 11, temos um exemplo de código em assembly visual, mostrando ambos os usos dos labels.

Figura 11 - Trecho de código em assembly visual, mostrando utilização dos labels.



Fonte: Figura de autoria própria.

Na figura acima, podemos observar a label *main*, que sempre aponta o início da execução do programa, a primeira instrução movimenta o valor apontado pela label *val* para o registrador acumulador (o ACC), logo após temos a label *loop*, na qual é executada uma instrução de incremento no acumulador e, depois disso, uma instrução de pulo condicional, que retoma a execução do programa para a localização apontada pela label *loop*, caso o valor do acumulador seja menor do que sete, ou seja, o trecho executa uma iteração, contando do valor três até sete, algo similar a um loop (ou laço) em uma linguagem de programação de alto nível.

Uma observação importante aqui é como os formatos e cores têm significados definidos, a seta laranja sempre representa uma label, a elipse vermelha um registrador e o retângulo cinza um valor literal, estas foram escolhidas apenas para prova de conceito de como o artifício do assembly visual tornaria a programação mais simples se comparada àquela utilizado uma linguagem assembly tradicional, em texto.

5. TESTES

A seguir é apresentado um exemplo de programa para a máquina virtual, este tem uma funcionalidade muito básica, mostrar uma mensagem de “olá mundo” no display utilizado como dispositivo de saída, para tal foram utilizadas seis das 37 instruções disponíveis na arquitetura, estas são mostradas e explicadas no quadro 1, logo a seguir.

Quadro 1 - Instruções utilizadas no programa de exemplo.

Opcode	Mneumônico	Operandos		Descrição
		1	2	
0x00	MOV	literal (16 bits)	registrador (8 bits)	Move o valor literal para o registrador especificado
0x05	MOV	registrador (8 bits)	registrador (8 bits)	Move o valor do endereço de memória apontado pelo registrador 1 para o registrador 2
0x06	MOV	registrador (8 bits)	registrador (8 bits)	Move o valor do registrador 1 para o endereço de memória apontado pelo registrador 2
0x10	JNE	literal (16 bits)	endereço (16 bits)	Pula a execução do programa para o endereço de memória, caso o valor do acumulador seja diferente do literal
0x40	INC	registrador (8 bits)		Incrementa o valor no registrador
0xFF	HLT			Termina a execução do programa

Fonte: Quadro de autoria própria.

O código do programa de exemplo, da mesma forma em que está distribuído na memória da máquina virtual, em formato hexadecimal, é mostrado no quadro 2, logo a seguir. Lembrando que as instruções de um programa começam no endereço 256 (ou 0x0100 em hexadecimal) pois a memória antes desta posição é reservada para os dispositivos conectados à máquina, e, sendo assim, ficam vazios no início da execução de um programa.

Quadro 2 - Código em hexadecimal do programa de exemplo.

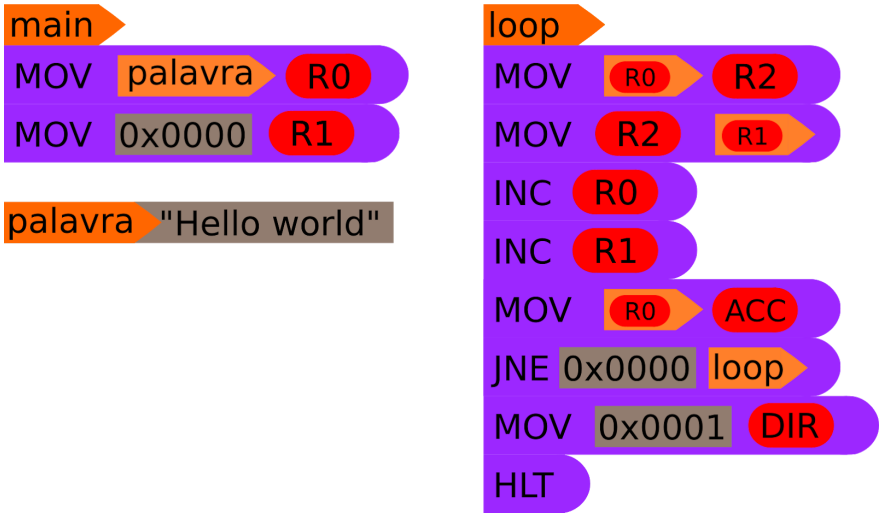
	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0x0100	00	01	1F	00	00	00	00	01	05	00	02	06	02	01	40	00
0x0110	40	01	05	00	10	10	00	00	01	08	00	00	01	11	FF	48
0x0120	65	6C	6C	6F	20	77	6F	72	6C	64	00	00	00	00	00	00

Legenda	
	Opcodes
	Parâmetros
	Dados
	Não utilizado

Fonte: Quadro de autoria própria.

Uma outra forma de escrever um programa em baixo é utilizando-se da linguagem *assembly*, como já foi dito anteriormente neste trabalho, o *assembly visual* para a máquina virtual não foi desenvolvido, porém os mnemônicos e funcionalidades deste foram projetados. A figura 12, mostra como ficaria o programa de teste referido nesta seção, se desenvolvido na linguagem *assembly* visual que foi projetada para a máquina virtual.

Figura 12 - Código em assembly visual do programa de exemplo.



Fonte: Figura de autoria própria.

Na figura 13, é possível observar o resultado da execução do programa, que consiste na visualização de uma mensagem no display OLED ligado à máquina virtual como um dispositivo externo.

Figura 13 - Resultado da execução do programa de exemplo.

Fonte: Figura de autoria própria.

Existem nas literaturas diferentes formas de desenvolver uma máquina virtual e com finalidades distintas, portanto, através dos estudos realizados e nos fundamentos do levantamento bibliográfico, foi desenvolvido uma máquina virtual para sistema embarcado totalmente funcional e modular, apresentando o funcionamento de um computador real através da virtualização, isso foi feito de modo que pode auxiliar o iniciante da computação a entender o funcionamento das categorias de código, plataformas, estruturas de arquiteturas de computadores por meio de programação utilizando a plataforma de *assembly visual*.

Com base nos resultados obtidos podemos notar de forma transparente as infinidades aplicações e diferença se comparado a trabalhos desenvolvido por outros autores, o qual não apresenta uma forma de modularização da plataforma desenvolvida, e tão pouco indubitável para o iniciante na programação.

6. CONCLUSÃO

Com o desenvolvimento deste trabalho, foi comprovado ser completamente possível o desenvolvimento de implementações virtuais de computadores completamente do zero, baseando-se totalmente em conteúdos relativos a arquiteturas físicas, que antes tinham sido abordados apenas de forma teórica pelos desenvolvedores desta aplicação. No desenvolvimento foram utilizados conceitos de programação intermediários, como os ponteiros de memória e macros da linguagem C, assim foi criada outro nível de abstração, capaz de executar aplicações completamente funcionais e desenvolvidas de forma significativamente mais simples do que aquelas para o nível inferior a este, sobre o qual a máquina virtual foi desenvolvida, podendo inclusive ser uma alternativa no ensino de desenvolvimento de software e arquitetura de computadores para usuários mais leigos.

Além de tudo isso, o projeto em questão ainda é de código aberto, e foi desenvolvida de maneira completamente extensível, tanto nas instruções, quanto nos dispositivos virtuais, permitindo que toda a comunidade de desenvolvedores o utilizem e realizem alterações e melhorias neste como bem entenderem, tendo isso em mente, algumas sugestões para futuras melhorias são:

- Implementação de subrotinas;
- Implementação de interrupções externas para entrada por dispositivos;
- Desenvolvimento de mais dispositivos padrão, como acesso aos GPIOs, saída de áudio e timer;
- Criação do ambiente de desenvolvimento para a máquina virtual em assembly visual;
- Implementação de gravação de código na máquina virtual de forma sem fio.
- Documentação detalhada da máquina virtual e seus dispositivos.

7. REFERÊNCIAS

TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização estruturada de computadores**. 6ª edição. São Paulo: Pearson Education do Brasil, 2013.

RODRIGUES, Erich. **Gramáticas e linguagens**. uCoder, 2015. Disponível em: <https://ucoder.com.br/problems/1082/html/>. Acesso em: 21 de março de 2020.

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D.. **Compilers: Principles, Techniques, & Tools**. Second Edition. Boston: Pearson Education, 2007.

ALEXANDRINI, Fábio et al. **Desenvolvimento do Compilador da Linguagem Basico**. ln: XI SIMPÓSIO DE EXCELÊNCIA EM GESTÃO E TECNOLOGIA (SEGET), 2014, Resende, RJ.

STÄRK, Robert; SCHMID, Joachim; BÖRGER, Egon. **Java and the Java Virtual Machine: Definition, Verification, Validation**. Berlin: Springer-Verlag, 2001.

GOUGH, K. John. **Stacking them up: a Comparison of Virtual Machines**. ln: 6th AUSTRALIAN COMPUTER SYSTEMS ARCHITECTURE CONFERENCE (ACSAC), 2001, Queensland, Australia.

KHAN, Sohail et al. **Analysis of Dalvik Virtual Machine and Class Path Library**. ln: CONSTRAINED INTENTS: EXTENDING ANDROID SECURITY FOR INTENT POLICIES (EASIP), 2009, Institute of Management Science, Peshawar, Paquistão.

PEMBERTON, Steven. DANIELS, Martin. **Pascal Implementation: The P4 Compiler and Interpreter.** Disponível em: <https://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html>. Acesso em 15 de junho de 2020.

MANOUSARIDIS, Konstantinos. MAVRIDIS, Apostolos. ANAGNOSTOPOULOS, Konstantinos. KALOGIANNIS, Gregory. **Introducing an innovative robot-based mobile platform for programming learning**. ln: INTERNATIONAL CONFERENCE ON INTERACTIVE MOBILE COMMUNICATION TECHNOLOGIES AND LEARNING (IMCL), Thessalônica, Grécia, 2015.

SILVA, Lucas M.. **Uma máquina virtual para uso didático.** 2012. 236 f. Monografia (Graduação em Sistemas de Informação) - Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis, 2012.

ESPRESSIF SYSTEMS. **ESP-IDF Programming Guide, latest version.** Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html>. Acesso em 20 de junho de 2020.

MEDIUM. **The Secret of Java- JDK, JRE, JVM difference - by Mann Verma**. Disponível em: <https://medium.com/@mannverma/the-secret-of-java-jdk-jre-jvm-difference-fa35201650ca>. Acesso em 20 de junho de 2020.

SANTOS, Jean Willian. JUNIOR, Renato Capelin de Lara. **Sistema de automatização residencial de baixo custo controlado pelo microcontrolador ESP32 e monitorado via smartphone.** 2019. 46 f. Trabalho de conclusão de curso - Departamento acadêmico de Eletrônica Curso superior de Tecnologia em Automação Industrial, Campus Ponta Grossa, Universidade Federal do Paraná, Paraná, 2019.

GITHUB - PELLEP/SPIFFS. **Wear-leveled SPI flash file system for embedded devices.** Disponível em: <https://github.com/pellepl/spiffs>. Acesso em 08 de agosto de 2020.

GITHUB - IGRR/MKSPIFFS. **Tool to build and unpack SPIFFS images.** Disponível em: <https://github.com/igrr/mkspiffs>. Acesso em 08 de agosto de 2020.

KOLBAN, Neil. **Kolban's book on ESP32**. [S.l.]: Leanpub, 2018.