UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

EDUARDO NACCACHE MARTINS DA COSTA

UM SIMULADOR DO PROCESSADOR SAPIENS NO NODEMCU (ESP8266)

RIO DE JANEIRO

2018

EDUARDO NACCACHE MARTINS DA COSTA

UM SIMULADOR DO PROCESSADOR SAPIENS NO NODEMCU (ESP8266)

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Gabriel Pereira da Silva, D.Sc.

RIO DE JANEIRO

2018

CIP - Catalogação na Publicação

C837s

Costa, Eduardo Naccache Martins da
Um simulador do processador Sapiens no NodeMCU (ESP8266) / Eduardo Naccache Martins da Costa. -- Rio de Janeiro, 2018.
59 f.

Orientador: Gabriel Pereira da Silva.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2018.

1. Processador. 2. Simulador. 3. Internet das coisas. I. Silva, Gabriel Pereira da, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

EDUARDO NACCACHE MARTINS DA COSTA

UM SIMULADOR DO PROCESSADOR SAPIENS
NO NODEMCU (ESP8266)

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 07 de JUNHO de 2018 .

BANCA EXAMINADORA:

Prof. Gabriel Pereira da Silva, D.Sc.

Prof. José Antonio dos Santos Borges, D.Sc.

Profa. Silvana Rossetto, D.Sc.

AGRADECIMENTOS

Agradeço, primeiramente, aos meus amigos do colégio, em especial Felipe e Diogo, que me ajudaram em diversas situações durante a graduação, tanto emocional quanto didaticamente.

Também agradeço aos meus colegas de faculdade, pelas incontáveis horas de estudo em grupo e ajudas em momentos de necessidade.

Agradeço a todos os professores que me acompanharam durante a gradução, em especial ao meu orientador Prof. Gabriel, que me guiou durante a realização deste trabalho.

RESUMO

O projeto desenvolvido consiste num simulador do código de máquina do processador hipotético Sapiens para ser rodado no NodeMCU, uma placa de desenvolvimento que conta com o ESP8266, um *chip* de baixíssimo custo que possui Wi-Fi. Foi montado um circuito numa *protoboard* com dispositivos para permitir uma interação com o simulador, como um *display*, um *keypad* e botões. Como o ESP8266 possui Wi-Fi e pode se conectar à rede, também foi desenvolvida uma interface *web*, bem mais completa, que permite um acompanhamento melhor da execução e depuração do programa, além de possuir mais dispositivos de entrada/saída e recursos como *breakpoints*. Foram feitas adições a instruções do Sapiens, de modo que seja possível controlar o valor dos pinos do NodeMCU diretamente pelo programa em execução no simulador. Com isso, é possível desenvolver programas para o Sapiens em linguagem de montagem para interagir com dispositivos físicos na *protoboard*, como LEDs, *speakers* e *displays*. Espera-se, assim, despertar um maior interesse dos alunos em disciplinas de Ciência da Computação como Circuitos Lógicos, Arquitetura de Computadores e, ainda, disciplinas eletivas como Internet das Coisas.

Palavras-chave: Processador. Simulador. IoT.

ABSTRACT

This project consists of a simulator of the machine code of the hipotetical processor Sapiens, to be executed on the NodeMCU, a development board containing the ESP8266, a extremely low cost chip which features Wi-Fi. A circuit was assembled on a breadboard, with devices such as a display, a keypad and some buttons, allowing the interaction with the simulator. As the ESP8266 features Wi-Fi and can connect to the network, a web interface was also developed, bringing a more complete experience, allowing a better view of the execution and debug of the program, while also featuring more in/out devices, and breakpoints. Modifications were made to Sapiens instructions, to make possible the control of NodeMCU pin values directly within programs running on the simulator. Thereby, it is now possible to develop assembly language Sapiens programs to interact with physical devices on the breadboard, such as LEDs, speakers and displays. It is expected, therefore, to make students more interested in Computer Sciente disciplines such as Logical Circuits, Computer Architecture, and Internet of Things.

Keywords: Processor. Simulator. IoT.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

I2C	Inter-Integrated Circuit
SoC	System on a Chip
RAM	Random-Access Memory
IRAM	Instruction Random-Access Memory
ROM	Read-Only Memory
LED	Light-Emitting Diode
GPIO	General-Purpose Input/Output
E/S	Entrada/Saída
I/O	Input/Output
IoT	Internet of Things
MHz	Megahertz
USB	Universal Serial Bus
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
HTML	Hypertext Markup Language
PWM	Pulse-Width Modulation
IP	Internet Protocol
JSON	JavaScript Object Notation

SUMÁRIO

1 INTRODUÇÃO

Uma grande dificuldade no ensino de arquitetura de computadores é fazer com que os alunos compreendam corretamente o funcionamento de um processador. O uso de simuladores de processadores, sejam de processadores comerciais ou didáticos, é uma estratégia frequentemente utilizada para alcançar esse objetivo. Para a formação de um conhecimento sólido do funcionamento dos processadores é bastante importante a visão da arquitetura do processador, do formato das instruções e dos passos necessários para a execução de um programa por parte do estudante.

Há mais de 10 anos, foi lançado o simulador Neanderwin, para o processador didático Neander-X [21]. Este simulador foi utilizado por muito tempo pelos autores e por diversos outros professores universitários do país, em cursos de Arquitetura de Computadores.

Após diversos anos de uso do Neanderwin, e observando suas limitações, foi desenvolvido, por Gabriel P. Silva e José Antônio dos S. Borges, o processador hipotético Sapiens e o simulador SimuS, apresentados no livro "SimuS: Um Simulador Didático para Arquitetura de Computadores" [23].

No processador Sapiens, a arquitetura e o conjunto de instruções inicialmente propostos para o Neander-X foram estendidos, para se obter uma arquitetura com menos limitações. Dentre as melhorias, temos o maior espaço de endereçamento de memória, instruções para chamada e retorno de procedimentos, e novos dispositivos de entrada/saída. Apesar das mudanças, os códigos desenvolvidos para o Neander-X são totalmente compatíveis com o Sapiens [22] em nível de linguagem de montagem.

Notadamente, no Sapiens foram introduzidas instruções especiais de TRAP, que permitem ao usuário acessar dispositivos de entrada e saída mais complexos de uma maneira mais fácil, de modo similar a uma chamada de sistema, como ocorre com os modernos processadores e sistemas operacionais.

O simulador SimuS, disponível para Windows e Linux [1], conta com um ambi-

[1] `https://github.com/sottam/simus`

ente integrado de desenvolvimento, para auxiliar na criação, compilação, execução e depuração do código para o processador hipotético Sapiens.

Neste projeto, é apresentado um simulador do processador Sapiens desenvolvido para rodar no NodeMCU [7], uma placa de desenvolvimento baseada no ESP8266 [5]. O ESP8266 é um *chip* de baixíssimo custo, que integra processador RISC de 32 bits, memória *flash* de 4 MB e memória RAM de 160 KB. Além disso, possui pinos digitais para leitura e escrita, um pino para leitura de valores analógicos, pinos para a geração de sinais PWM, além de interface SPI, I2C e WiFi.

O NodeMCU foi montado numa placa *protoboard*, com dispositivos físicos acoplados, como um *display*, um *keypad* de 12 teclas e 4 botões, para auxiliar na controle do simulador e acompanhamento da execução dos programas. Dessa forma, é possível executar programas desenvolvidos para o Sapiens em um hardware fora do computador, e visualizar os valores dos registradores em tempo real no *display*.

Adicionalmente, foi desenvolvida uma interface *web*, bem mais completa, através da qual pode ser feita a interação do usuário com o simulador no NodeMCU. O NodeMCU é conectado à rede WiFi e, acessando a página através de um computador na rede, é possível realizar diversas funções, como:

- enviar o seu programa;
- utilizar dispositivos virtuais de entrada e saída;
- controlar a execução;
- visualizar toda a memória em tempo real;
- adicionar breakpoints para auxiliar a depuração do código.

Novos tipos de TRAP foram adicionados ao simulador desenvolvido, de modo que o valor dos pinos do NodeMCU possa ser controlado diretamente pelo programa em execução no simulador. Com isso, é possível fazer programas que interajam com dispositivos físicos na *protoboard*, como LEDs, *speakers*, *displays*, etc.

Com este projeto, espera-se despertar um maior interesse por parte dos alunos nas disciplinas básicas do curso de Ciência da Computação, como Circuitos Lógicos,

Arquitetura de Computadores e, ainda, disciplinas eletivas como Internet das Coisas. Os programas desenvolvidos em linguagem de máquina do Sapiens agora poderão ir além das fronteiras da tela do computador.

1.1 TRABALHOS CORRELATOS

Existem muitas ferramentas para simulação de processadores, tanto comerciais quanto hipotéticos. Temos os simuladores de processadores comerciais mais usados em projetos de equipamentos atuais, como os de 8051, ATmega, PIC e variantes de processadores do tipo RISC. Porém, quase todos são destinados ao uso profissional, e alguns deles possuem licenças de uso pagas e ferramentas de uso complexo [22].

Há também simuladores didáticos de processadores comerciais, como o Abacus [26] e o GNUSim8085 [19], que simulam o 8085 da Intel que tem sido utilizado amplamente no ensino de arquitetura de computadores, devido a sua simplicidade.

Também existem os simuladores didáticos de processadores hipotéticos como o R2DSim [18], para simular uma arquitetura RISC de 16 bits, e o SIMAEAC [24], que implementa um subconjunto da arquitetura do 8085 [22]. O simulador SimuS, introduzido junto com o processador Sapiens, se encaixa nesta categoria. O projeto desenvolvido tem bastante relação com o SimuS na parte da execução do código de máquina do Sapiens. Porém, o SimuS conta com muitos outros recursos, detalhados na seção 2.2.

Além disso, há também a emulação de processadores em placas de FPGA, muitas vezes utilizada para reproduzir o comportamento de processadores antigos de forma mais eficiente e fiel. Como exemplos, há projetos como o Apple2fpga, em que foi implementado o antigo Apple II+ na placa de FPGA Altera DE2 [13].

O projeto com mais semelhança encontrado foi uma adaptação da versão de Linux do SimuS para rodar no Raspberry Pi, com modificações para tornar possível o uso dos pinos GPIO. Apesar de ser o próprio simulador do Sapiens para PC rodando no Raspberry Pi, tem como semelhança a possibilidade de controlar os pinos de GPIO através de instruções de TRAP do Sapiens. Este projeto é detalhado

no livro “SimuS: Um Simulador Didático para Arquitetura de Computadores” [23].

2 CONCEITOS GERAIS

Neste capítulo, serão apresentados conceitos básicos necessários para a compreensão do projeto. Será introduzido o processador que é simulado e a placa de desenvolvimento utilizada no projeto.

2.1 O PROCESSADOR SAPIENS

O Sapiens é um processador hipotético utilizado para o ensino de arquitetura de computadores, e foi desenvolvido como uma evolução do processador Neander-X, que possuia algumas limitações. O Neander-X [21] foi um processador hipotético com um conjunto simplificado de instruções, utilizado durante anos no ensino de arquitetura de computadores, pelos autores e por diversos professores em universidades do país [22]. Apesar de ajudar na introdução dos aspectos principais de um processador, o Neander-X possuia algumas limitações, tais como [23]:

- Quantidade muito limitada de memória (256 bytes);
- Ausência de *flags* para suporte à aritmética *multi-byte*;
- Limitação dos modos de endereçamento: apenas o modo direto.

O processador hipotético Sapiens pode ser visto como uma evolução do Neander-X. A arquitetura do Sapiens resolve muitas das limitações do Neander-X, e conta com detalhes que tornam possível a sua utilização em problemas menos simples, que eram muito difíceis ou até impossíveis de serem desenvolvidos no Neander-X [22]. O diagrama em blocos do Sapiens pode ser visto na Figura 1.

Algumas melhorias importantes do Sapiens em relação ao Neander-X são:

- Modo indireto e modo imediato de endereçamento;
- *Flag* de *carry* para "vai-um" e "vem-um";
- Expansão do conjunto de instruções para inclusão de diversos tipos de desvio condicional, novas operações lógicas e aritméticas, entre outras;

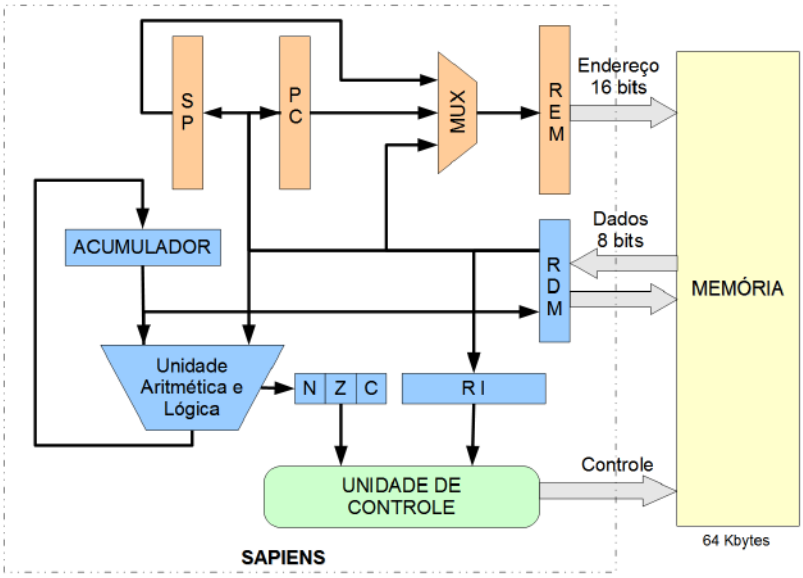

Figura 1: Diagrama em blocos do processador hipotético Sapiens [23]

- Aumento do tamanho do apontador de instruções para 16 bits, permitindo endereçar até 64K posições de memória;
- Inclusão de um apontador de pilha, também de 16 bits;
- Instruções para chamada e retorno de procedimento;
- Instruções para a manipulação de dados na pilha.

As instruções do Sapiens em linguagem de máquina podem ter um, dois ou três bytes, como mostra a Figura 2. O primeiro byte contém o código de operação nos 6 bits mais significativos e, quando for o caso, o modo de endereçamento nos 2 bits menos significativos [23].

Figura 2: Formato das instruções do processador Sapiens [23]

As instruções com apenas 1 byte não tem outro operando além do acumulador,

que é um operando implícito para quase todas as instruções. As instruções com 2 bytes são aquelas que utilizam, além do acumulador, um dado imediato de 8 bits como operando no segundo byte da instrução. Nas instruções com 3 bytes, o conteúdo do segundo e terceiro bytes da instrução depende do modo de endereçamento. A codificação para o modo de endereçamento e o conteúdo dos dois últimos bytes da instrução é a seguinte:

- **00 - Direto**: o segundo e terceiro bytes da instrução contêm o endereço de memória do operando;
- **01 - Indireto**: o segundo e terceiro bytes da instrução contêm o endereço da posição de memória com o endereço do operando (ou seja, é o endereço do ponteiro para o operando);
- **10 - Imediato 8 bits**: o segundo byte da instrução é o próprio operando;
- **11 - Imediato 16 bits**: os dois bytes seguintes à instrução são utilizados como operando.

Note que nos dois primeiros modos de endereçamento, apesar do endereço da posição de memória possuir 2 bytes, o operando final em memória possui apenas 1 byte de comprimento.

Na linguagem de montagem, utiliza-se o "@" (arrôba) para indicar que um operando é indireto, e o "#" (tralha) para indicar que um operando é imeditado.

2.2 O SIMULADOR SIMUS

O simulador SimuS[1] foi introduzido junto com o processador Sapiens e está disponível para Windows e Linux. A janela principal da interface de usuário pode ser vista na Figura 3.

[1] `https://github.com/sottam/simus`

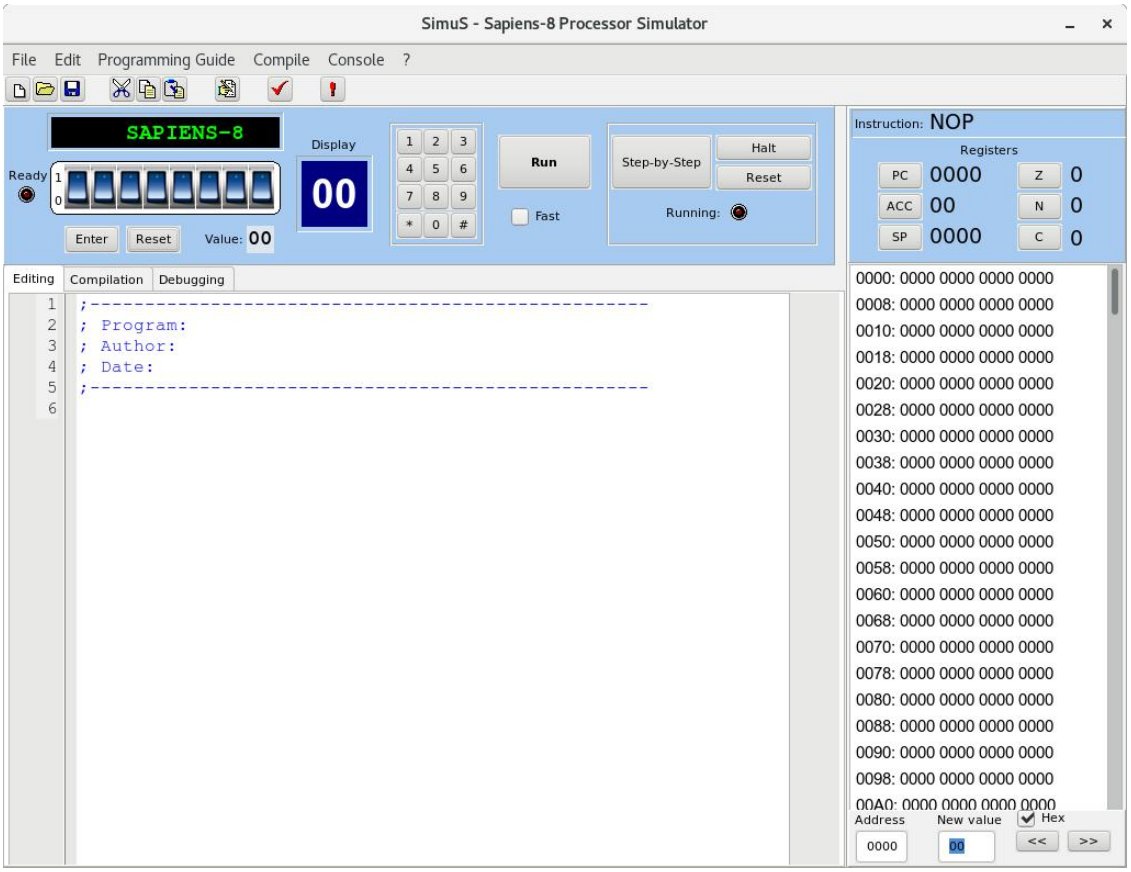

Figura 3: Interface do simulador SimuS[2]

Alguns dos recursos do SimuS são:

- Editor de texto;
- Montador (*assembler*) para gerar o código objeto final em linguagem de máquina;
- Execução contínua ou passo a passo (uma por vez), e possibilidade de pausar a execução;
- Modificação do valor dos registradores, de *flags*, e de valores da memória;
- *Breakpoints* para auxiliar na depuração;
- Simulador de dispositivos de entrada e saída, como um painel de chaves, um visor hexadecimal, um teclado de 12 teclas e um painel de 1 linha com 16 caracteres;

[2] `https://github.com/sottam/simus`

- Exportação de um *dump* de memória no formato Intel HEX.

Além disso, estão disponíveis algumas diretivas do compilador, que não são instruções do Sapiens, mas sim diretivas que dão instruções para o compilador, como posições de variáveis na memória ou indicação da posição de início do código.

2.3 O NODEMCU

No projeto, foi utilizada a placa de desenvolvimento NodeMCU, exibida na Figura 4, que conta com um ESP8266, memória *flash* de 4 MB e um conversor *serial*-USB [7] [1].

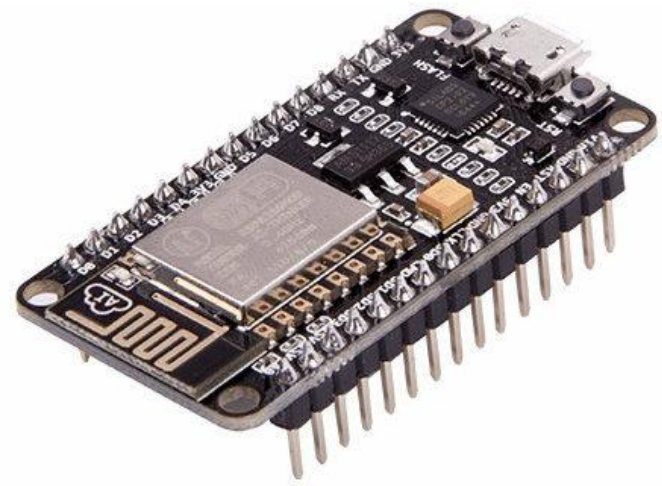

Figura 4: A placa de desenvolvimento NodeMCU [20]

O ESP8266 é um SoC (*System on a Chip*) de baixo custo, com um processador de 32 bits e frequência de relógio de 80 MHz. Ele é largamente utilizado em aplicações de Internet das Coisas. Possui 160 KB de memória RAM, onde 80 KB estão disponíveis para o usuário. A divisão das áreas de memória do ESP8266 é mostrada com detalhes na seção 2.3.1. Além disso, possui diversos pinos de entrada/saída e WiFi [5].

O ESP8266 por si só é apenas um *chip* de poucos milímetros, exibido na Figura 5, e seria necessário o uso de solda para poder utilizá-lo, além de ser trabalhoso o carregamento de programas. Para permitir que ele seja utilizado de maneira mais fácil, foram criadas "placas de desenvolvimento", como o NodeMCU, utilizado no

projeto. Estas placas de desenvolvimento incluem o ESP8266 já soldado, com pinos para encaixar numa *protoboard*, memória *flash* externa e conversor *serial*-USB. Com isso, pode ser ligado ao PC com um simples cabo USB e o carregamento de programas pode ser feito sem maiores complicações.

Figura 5: O *system on a chip* ESP8266 avulso [4]

O termo "NodeMCU" pode ser utilizado para se referir tanto à placa de desenvolvimento utilizada no projeto, quanto a um dos *firmware* disponíveis para ela, detalhado a seguir. Dessa forma, sempre que o termo for utilizado aqui, será indicado de forma explícita a qual dos dois a referência está sendo feita.

O desenvolvimento para o ESP8266 pode ser feito de diversas formas, sendo possível escolher entre diversas linguagens e ambientes de desenvolvimento. Os três principais são:

- O *firmware* NodeMCU[3], para rodar *scripts* em Lua. A linguagem Lua tem a vantagem de tornar mais fácil o desenvolvimento, por ser de altíssimo nível. Por ser dinamicamente tipada e interpretada, o desenvolvimento de aplicações é mais rápido e fácil. Não é necessário se preocupar com alguns detalhes como na linguagem C. Ele é anunciado como uma plataforma de IoT simples e rápida, com exemplos de códigos de poucas linhas para rodar um servidor HTTP, por exemplo.

- Espruino[4], que conta com um interpretador de JavaScript, com diversas otimizações que melhoram o desempenho e o consumo de memória em relação ao

[3] `https://github.com/nodemcu/nodemcu-firmware`

[4] `http://www.espruino.com/EspruinoESP8266`

JavaScript original. Inclui a opção de compilar o código em JavaScript para melhorar ainda mais o desempenho. O JavaScript, assim como a Lua, é uma linguagem de script de altíssimo nível, o que torna o desenvolvimento mais simples e rápido.

- IDE do Arduino[5], programando em C/C++. Através dela, é possível fazer uso do grande número de bibliotecas disponíveis para os mais diferentes módulos de *hardware*. O uso da linguagem C/C++ traz vantagens em relação ao consumo de memória e ao desempenho, em comparação com Lua e JavaScript. Como Lua e JavaScript são interpretadas e dinamicamente tipadas, é fácil chegar nos limites do ESP8266. Com o C ou C++, é possível aproveitar melhor e de forma mais eficiente o que o ESP8266 tem a oferecer.

2.3.1 Memória

O ESP8266 possui, no total, 160 KB de memória RAM. No entanto, nem tudo está disponível para o usuário. A divisão de memória é feita da seguinte forma [14] [10]:

- **96 KB**: DRAM (*Data RAM*), memória para dados. Dividida em:
  - **80 KB** de memória disponível para o usuário. Porém, parte disso é ocupada com bibliotecas e sistema operacional. Na prática, mesmo com programas bem simples, apenas 50 KB ou menos estarão disponíveis.
  - **16 KB** de ETS *System Data* RAM, memória reservada para o uso da ROM.
- **64 KB**: IRAM (*Instruction RAM*), memória para instruções. Dividida em duas partes:
  - **32 KB** iniciais para armazenamento do código que **não** foi marcado com a diretiva ICACHE_FLASH_ATTR, detalhada ao final desta seção. Esta parte do código será copiada no início do carregamento do programa.

[5] `https://github.com/esp8266/Arduino`

  - **32 KB** finais para um *cache* do código marcado com a diretiva ICACHE_FLASH_ATTR. O código marcado com esta diretiva ficará na memória *flash* externa, e será copiado pelo ESP8266 para esta parte da IRAM à medida que for utilizado [12].

Duas diretivas podem ser utilizadas no código em C para forçar que determinada variável ou função não seja carregada inicialmente na memória RAM:

- PROGMEM (ou ICACHE_RODATA_ATTR): Diretiva usada para indicar que variáveis devem ficar na memória *flash* externa, para não ocupar espaço na RAM. Utilizado normalmente para evitar que *strings* grandes ocupem espaço na RAM, por exemplo [6]. Na medida que forem utilizadas, as variáveis são copiadas para a RAM.

- ICACHE_FLASH_ATTR: Diretiva utilizada para indicar parte do código (funções) que devem ficar na memória *flash* externa, e não ser copiada inicialmente para a IRAM [10]. Como dito anteriormente, a IRAM possui, no total, 64 KB. Nos 32 KB iniciais, o código das funções que **não** foram marcadas com esta diretiva é copiado, no início do carregamento do programa. Nos 32 KB finais da IRAM, é mantido um *cache* de parte do código das funções que **foram** marcadas com esta diretiva. Na medida que for utilizado, parte do código marcado com esta diretiva será copiado da memória *flash* para estes 32 KB finais da IRAM. Este gerenciamento é feito de forma automática pelo ESP8266. Rotinas de tratamento de interrupções e código que seja executado com muita frequência não devem ser marcados com este atributo.

3 O PROJETO

Neste capítulo, será apresentada a idéia principal do projeto desenvolvido. Em seguida, será detalhado o formato do arquivo de entrada esperado. Por fim, será abordado o método de acesso aos pinos do NodeMCU através de instruções do Sapiens.

3.1 VISÃO GERAL

O projeto desenvolvido consiste num simulador do código de máquina do Sapiens, a ser rodado no ESP8266. Um *dump* de memória do programa compilado, no formato Intel HEX [15], é enviado para o ESP8266, que o executará.

O usuário pode interagir com o simulador de duas maneiras:

- Através de elementos físicos, numa *protoboard*, como botões, um teclado numérico e um *display*.
- Através de uma interface *web*, utilizando um navegador num computador na rede.

Na interface *web* é possível ter a experiência mais completa. É possível inserir *breakpoints*, visualizar a memória completa e interagir utilizando todos os dispositivos de entrada/saída. Na *protoboard*, por limitações físicas, nem todos os dados podem ser vistos no *display*, e não estão disponíveis todos os dispositivos de entrada/saída. As duas maneiras serão detalhadas nas seções 4 e 5, respectivamente.

O simulador, que roda no ESP8266, foi desenvolvido em C++, através da IDE do Arduino. A interface *web* foi desenvolvida em HTML e JavaScript. O código está disponível no GitHub e pode ser acessado pelo endereço `https://github.com/edunmc/tcc-ufrj`.

3.2 LIMITAÇÕES DE MEMÓRIA E SOLUÇÃO DESENVOLVIDA

O Sapiens endereça até 64 KB de memória RAM. Portanto, seria necessário alocar 64 KB de RAM para o perfeito funcionamento do simulador. O ESP8266 possui 80 KB de memória RAM disponível para o usuário, mas, após carregar o programa desenvolvido com todas as bibliotecas, restam apenas em torno de 30 KB para uso do simulador. Dessa forma, não é possível armazenar a memória inteira a ser simulada no ESP8266.

Para contornar este problema, os 64 KB de memória foram separados em páginas (blocos) de 1 KB, e cada página só é alocada quando é requisitada a escrita em algum endereço dentro dela. Assim, um programa pode utilizar, por exemplo, partes diferentes e distantes da memória, sem complicações.

Cada grupo de 1024 bytes forma uma página, e cada página é alocada por completo:

- Página 0, posições de 0 a 1023;
- Página 1, posições de 1024 a 2047;
- e assim por diante.

Este gerenciamento da memória e alocação de páginas é feito pela classe "Memoria". Neste classe, o método `ref` é utilizado para pegar a referência a uma posição de memória, para ser feita uma operação de escrita. O método `pega` é utilizado para se obter o valor de determinada posição de memória, ou seja, uma operação de leitura. Assim, só é necessário passar a posição de memória para os métodos correspondentes, e todo o trabalho de descobrir a página à qual a posição pertence e alocar a página caso necessário é abstraído por esta classe. A seguir são mostrados exemplos do uso destes métodos.

Um exemplo de uma operação de escrita na memória pode ser visto no código abaixo. Como 1025 é a segunda posição de memória da segunda página, esta página será alocada caso ainda não tenha sido por uma operação anterior. Um vetor de 1024 posições será alocado, e uma referência à segunda posição deste vetor será

retornada pela função `ref`. Em seguida, o valor 10 será colocado neste endereço.

```
// Escrever o valor 10 na posicao 1025 de memoria
memoria.ref(1025) = 10;
```

No código a seguir, vemos o exemplo de uma operação de leitura. O método `pega` identifica que a posição 1025 pertence à segunda página da memória, e acessa o vetor relativo a ela. Neste vetor, acessa o segundo elemento e retorna seu valor, que é colocado na variável `valor`.

```
// Ler o valor da posicao 1025 de memoria
int valor = memoria.pega(1025);
```

Desta forma, o gerenciamento de páginas é feito de maneira completamente automática e abstraída, e a memória pode ser utilizada no resto do programa de forma transparente e sem preocupações.

A limitação da solução desenvolvida é a ausência da substituição de páginas. Pode-se dizer que foi implementada apenas uma paginação parcial. Como o acesso a memória externa no ESP8266 é muito custoso, não foi implementado um método para substituição de páginas, e só será possível alocar páginas até que a memória do ESP8266 lote. Como as páginas alocadas vão ocupando a memória disponível, e inicialmente há em torno de 30 KB disponíveis, só será possível alocar por volta de 30 páginas. Porém, na prática, isto seria muito difícil ocorrer, visto que o Sapiens é utilizado para fins didáticos. Dificilmente um programa desenvolvido nessas circunstâncias usará mais do que alguns kilobytes de memória.

3.2.1 Código da página HTML

Quando se utiliza a interface *web* para interagir com o simulador, o código da página servida é armazenado no ESP8266 como uma grande *string*. Ao acessar o IP através do navegador, o servidor *web* retorna esta *string* com o código HTML e JavaScript da página. Porém, este código tem um tamanho razoável, chegando a quase 30 KB. Normalmente, isso ocuparia quase metade da memória RAM disponível

para o usuário. Porém, foi utilizada a diretiva PROGMEM (ver seção 2.3.1), para indicar que a *string* deveria ficar na memória *flash* externa, e não na RAM. Com esta diretiva, a *string* não ocupa espaço na memória RAM, e é copiada apenas temporariamente quando é acessada. Mais detalhes sobre a interface *web* podem ser vistos no capítulo 5. O processo de configuração dos servidores HTTP e WebSockets pode ser encontrado na seção 5.4.

3.3 ENVIO DE CÓDIGO PARA O SIMULADOR

Para o envio do código de máquina para o simulador, foi escolhido o formato Intel HEX. O simulador SimuS, em sua versão mais recente, possui opção para exportar o *dump* de memória no formato Intel HEX [15]. Para isso, basta ir no menu Arquivo → Exportar Hexa, após a compilação do código. Este formato tem algumas vantagens, como:

- Formato bem simples e organizado;
- Possibilidade de indicar dados tanto da memória quanto de registradores;
- Tipo específico de linha para indicar o fim do arquivo;
- *Checksum* ao fim de cada linha, tornando possível a verificação da integridade dos dados transmitidos.

O formato de cada linha no Intel HEX é detalhado a seguir. Cada linha consiste em seis campos (partes) que aparecem na seguinte ordem:

1. **Início de linha** (1 caractere ":")
   O caractere ":" indicando o início de uma linha.
2. **Tamanho dos dados** (2 dígitos hexadecimais representando 1 byte)
   Indica o tamanho do campo "dados".
3. **Endereço de início** (4 dígitos hexadecimais representando 2 bytes)
   Indica o endereço de memória de início (*offset*) dos dados.

4. **Tipo** (4 dígitos hexadecimais representando 2 bytes)
   Indica o tipo do dado sendo transmitido naquela linha. 0 – dados da memória, 1 – fim de arquivo, 2 a 5 – registradores.

5. **Dados** (2 dígitos hexadecimais para cada byte do tamanho especificado no campo "tamanho")
   Os dados em si. A linha de fim de arquivo não possui este campo.

6. ***Checksum*** (2 dígitos hexadecimais).
   Usados para checar a integridade da linha.

Como exemplo, tomemos a linha `:0300300002337A1E`. Separando nos 6 campos, temos:

`:`	`03`	`0030`	`00`	`02337A`	`1E`

- **Tamanho dos dados**: 3 bytes
- **Endereço de início**: 30
- **Tipo**: 0 – dados de memória
- **Dados**: os bytes `02`, `33` e `7A`.
- ***Checksum***: o byte `1E` pode ser usado para checar a integridade da linha.

Logo, esta linha está informando que o conteúdo da memória nas posições 30, 31 e 32 é, respectivamente, `02`, `33` e `7A`.

3.4 ACESSO AOS PINOS GPIO DO ESP8266

Através do simulador desenvolvido, é possível acessar os pinos de GPIO do ESP8266. Com isso, códigos desenvolvidos para o Sapiens agora podem realizar operações como ler o sinal de algum pino, ou definir o nível lógico de saída de um pino como alto ou baixo. Dessa forma, é possível interagir com dispositivos físicos numa *protoboard*, como LEDs e *speakers*, por exemplo.

O acesso aos pinos de GPIO é feito através de instruções de TRAP. Na Tabela 1 podem ser vistos os tipos de TRAP incluídos originalmente no Sapiens, e suportados

pelo simulador SimuS.

Tipo	**Descrição**
1	Leitura de um caractere da console
2	Escrita de um caractere na console
3	Leitura de uma linha inteira da console
4	Escrita de uma linha inteira na console
5	Chama uma rotina de temporização
6	Chama uma rotina para tocar um tom
7	Chama uma rotina para retornar um número pseudo-aleatório de 0 a 99
8	Define a semente inicial da rotina de números aleatórios

Tabela 1: Tipos de TRAP originais do Sapiens [23]

Além dos tipos de TRAP originais do Sapiens, no simulador desenvolvido foram adicionados novos tipos de TRAP para a comunicação com os pinos de GPIO do ESP8266.

Vale lembrar que os pinos GPIO da placa de desenvolvimento NodeMCU, utilizada, não são os números impressos na placa. Há uma equivalência entre os números de pino impressos na placa e os números de pinos GPIO, mostrada na Tabela 2. Dessa forma, para acessar, por exemplo, o pino D5 da placa, o número do pino GPIO utilizado no código deve ser 14.

Na placa	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
GPIO	16	5	4	0	2	14	12	3	15	3	1

Tabela 2: Equivalência entre pinos impressos na placa e pinos GPIO [17]

3.4.1 Novos tipos de TRAP

No Sapiens, o tipo de TRAP é passado no acumulador, e os parâmetros da operação a ser feita, no endereço passado como operando da instrução de TRAP.

Abaixo são listados os novos tipos de TRAP introduzidos no simulador desenvolvido, assim como os parâmetros requeridos em cada um deles.

#101 – Configurar pino como entrada ou saída

Neste tipo, os parâmetros que são passados no endereço de memória definido pelo operando da instrução TRAP são:

- Número do pino (1 byte): número do pino de GPIO do ESP8266.
- Modo (1 byte): 0 para entrada, 1 para saída.

Exemplo: definindo o pino 0 como saída.

```
LDA #101
TRAP OPERANDO

OPERANDO: DB 0, 1   ; pino, modo
```

#102 – Escreve o valor lógico de saída num pino

Neste tipo, os parâmetros que são passados no endereço de memória definido pelo operando da instrução TRAP são:

- Número do pino (1 byte): número do pino de GPIO do ESP8266.
- Valor (1 byte): 0 para nível baixo, 1 para nível alto.

Para este tipo de TRAP ser usado, o pino deve ter sido previamente configurado como saída (TRAP #101).

Exemplo: escrevendo o valor lógico alto no pino 0.

```
LDA #102
TRAP OPERANDO

OPERANDO: DB 0, 1   ; pino, valor
```

#103 – Lê o valor de entrada de um pino para o acumulador

Neste tipo, os parâmetros que são passados no endereço de memória definido pelo operando da instrução TRAP são:

- Número do pino (1 byte): número do pino de GPIO do ESP8266.

O valor lido será 0 ou 1, e será escrito no acumulador.

Exemplo: lendo o nível lógico do pino 0 para o acumulador

```
LDA #103
TRAP OPERANDO

OPERANDO: DB 0   ; pino
```

#104 – Configurar resistencia *pull-up* ou *pull-down* num pino

O ESP8266 possui resistência *pull-up* em todos os pinos, exceto no 16, que possui apenas *pull-down*. Por isso, este tipo de TRAP ativará a resistência *pull-up* nos pinos 0 a 15, e *pull-down* no 16. Neste tipo, os parâmetros que são passados no endereço de memória definido pelo operando da instrução TRAP são:

- Número do pino (1 byte): número do pino de GPIO do ESP8266.
- PUD (1 byte): 0 para "sem resistor", 1 para ativar o resistor de *pull-down* caso o pino seja 16 ou *pull-up* caso o pino seja qualquer outro.

Para este tipo de TRAP ser usado, o pino deve ser configurado necessariamente como entrada (TRAP #101).

Exemplo: Ativando o resistor de *pull-up* do pino 5

```
LDA #104
TRAP OPERANDO

OPERANDO: DB 5, 1   ; pino, pud
```

#105 - *Duty cycle* do PWM do pino

Neste tipo, os parâmetros que são passados no endereço de memória definido

pelo operando da instrução TRAP são:

- Número do pino (1 byte): número do pino de GPIO do ESP8266.
- Valor (2 byte): valor do *duty cycle*, de 0 a 1023. Este valor representa a porcentagem de tempo em que o pino fica em alto.

Para este tipo de TRAP ser usado, o pino deve ser configurado necessariamente como saída (TRAP #101).

Exemplo: Definindo o *duty cycle* do pino 5 como 512

```
LDA #105
TRAP OPERANDO

OPERANDO: DB 5    ; pino
          DW 512  ; duty cycle
```

3.5 INTERAGINDO COM O SIMULADOR

É possível interagir com o simulador de duas maneiras:

- Físicamente, através de botões, um *display* e um *keypad*, na *protoboard*; ou
- Através de uma interface *web*.

As duas maneiras serão exploradas com detalhes nos capítulos seguintes.

4 *PROTOBOARD*

Neste capítulo, é proposto um circuito com alguns dispositivos físicos que permitem uma interação básica com o simulador. Uma interação mais completa está disponível com a utilização da interface *web*, detalhada no capítulo seguinte.

4.1 CIRCUITO PROPOSTO

Uma das formas de interagir com o simulador desenvolvido, é a partir de dispositivos físicos num circuito montado numa *protoboard*. Foi montado o circuito exibido na Figura 6, com a placa de desenvolvimento NodeMCU, que conta com o ESP8266.

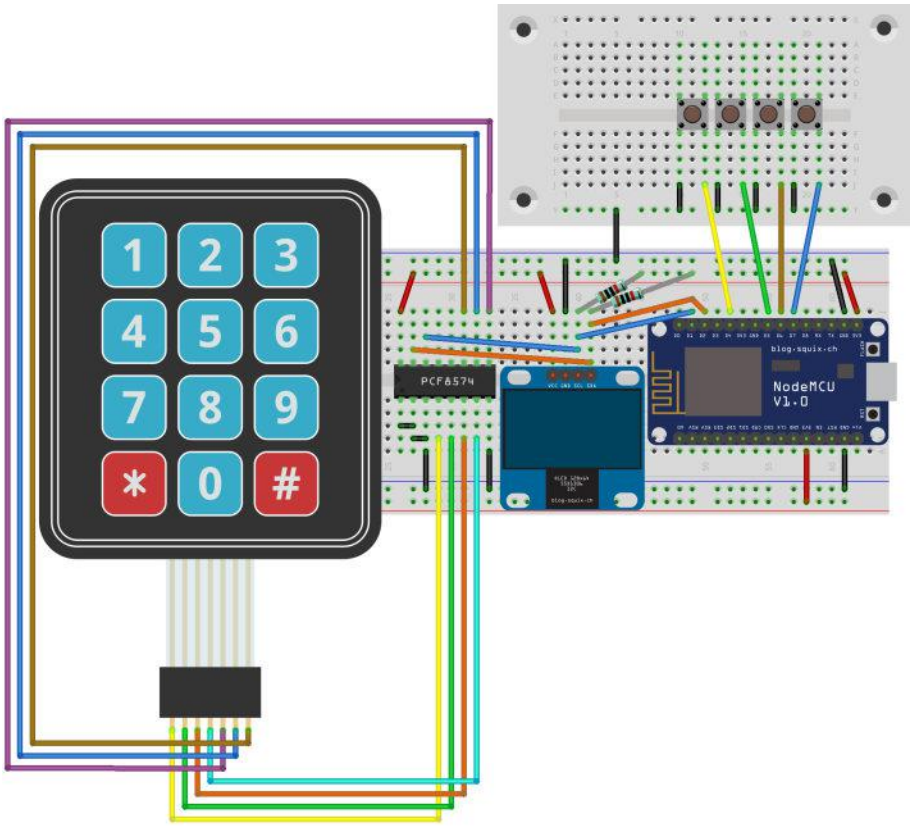

Figura 6: Circuito montado para o projeto, desenhado no Fritzing[1]

No circuito, temos:

- A placa de desenvolvimento NodeMCU;

[1] `http://fritzing.org`

- O *display* SSD1306, de resolução 128x64, usando o protocolo I2C para se conectar ao ESP8266;
- Um *keypad* de 12 teclas;
- Um circuito integrado PCF8574, para converter os 7 pinos paralelos do *keypad* para usar o protocolo I2C, que usa apenas 2 pinos.
- Quatro botões (*push buttons*) para controlar o fluxo de execução: iniciar execução contínua, passo a passo, pausar e resetar, respectivamente.

Na Figura 7, é possível ver uma foto real do circuito montado, com o simulador em execução. No *display*, é exibido o valor dos registradores e o *banner* de 16 caracteres.

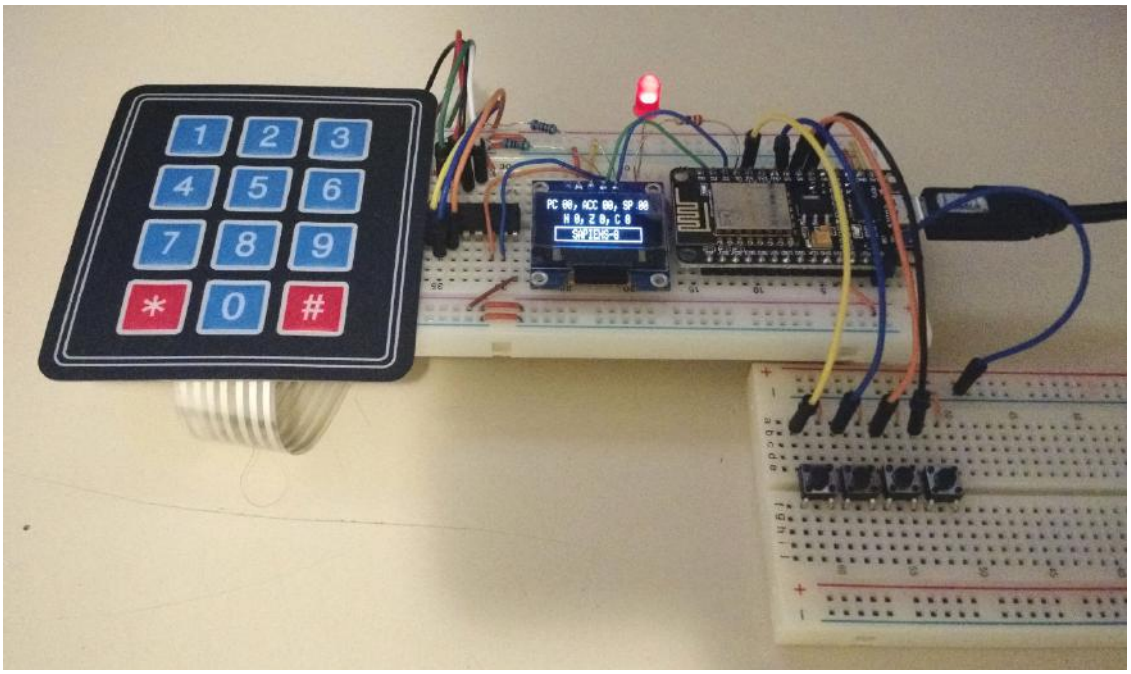

Figura 7: Foto do circuito em funcionamento

Nem todos os dispositivos de E/S estão disponíveis na *protoboard*, por limitações de espaço, assim como também não há como ver o conteúdo da memória. Para estes casos, deve ser utilizada a interface web, mais completa, detalhada no capítulo 5.

4.2 DISPOSITIVOS DE ENTRADA/SAÍDA

O *display* utilizado é o SSD1306 [8]. Ele tem resolução de 128x64, e utiliza o protocolo I2C para se conectar ao ESP8266. Foi utilizada a biblioteca Adafruit_SSD1306[2]. O *display* é usado para a exibição do valor dos registradores e do *banner* de 16 caracteres. Na Figura 8, é possível ver uma foto do *display* durante a execução de um programa. Nele, são exibidos os valores dos registradores (incluindo as *flags*: *negative*, *zero* e *carry*) e o *banner* de 16 caracteres.

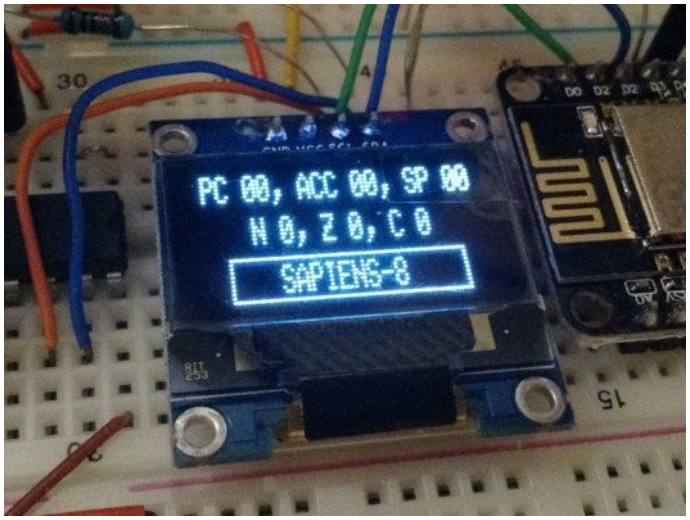

Figura 8: O *display* durante a execução de um programa

O *keypad* de 12 teclas pode ser usado como no simulador SimuS. Quando uma tecla é pressionada, o status "pronto" do keypad é definido, e o valor da última tecla pressionada é salvo.

O *keypad* utiliza uma matriz com um pino por linha e um por coluna, totalizando 7 pinos (matriz de 3×4). Como o número de pinos disponível para o usuário na placa de desenvolvimento é limitado, usar 7 pinos apenas para um teclado numérico seria demais. Por isso, foi utilizado o circuito integrado PCF8574, que converte até 8 bits paralelos para serial através do protocolo I2C, que utiliza apenas 2 pinos. Neste protocolo, todos os dispositivos utilizam o mesmo barramento, e são identificados por endereços diferentes. Desta forma, como o *display* já utilizava I2C, os 2 pinos

[2] `https://github.com/adafruit/Adafruit_SSD1306`

utilizados para o *keypad* foram os mesmos. Para a utilização do *keypad* com o protocolo I2C, foi utilizada a biblioteca Keypad_I2C[3].

4.3 INTERAÇÃO COM O SIMULADOR

Sem a utilização da interface *web*, o envio de arquivos pode ser feito pela *serial*, com a placa de desenvolvimento conectada ao computador pela USB.

Para o envio do código do programa a ser executado, deve ser utilizado o *dump* de memória no formato Intel HEX do código de máquina do Sapiens. Vale lembrar que o simulador desenvolvido é um simulador do código de máquina do Sapiens. Dessa forma, caso o usuário deseje programar em *assembly* para o Sapiens, é preciso que o *assembly* seja compilado antes de ser enviado para o simulador. Para isso, é possível utilizar o SimuS, no computador. Em sua versão mais recente, o SimuS possui opção para exportar um *dump* de memória no formato Intel HEX. Dessa forma, é possível escrever o *assembly*, compilar, e exportar o conteúdo da memória com o código de máquina.

Ao ser iniciado, o ESP8266 tentará se conectar à rede Wi-Fi utilizando o nome e senha definidos no código, no início do arquivo "web.ino". Após se conectar, ou depois de alguns segundos sem sucesso na tentativa de conexão, será ativado o modo em que o envio de um arquivo é aguardado, como mostra a Figura 9.

[3] `https://github.com/joeyoung/arduino_keypads/tree/master/Keypad_I2C`

Figura 9: Aguardando o envio de um arquivo no formato Intel HEX

Neste momento, o envio do arquivo pode ser feito pela *serial*. No Linux, isso pode ser feito através do comando "`cat arquivo.hex > /dev/ttyUSB0`", caso o dispositivo esteja mapeado em `/dev/ttyUSB0`. Como o formato Intel HEX possui um identificador de fim de arquivo, é possível saber que o envio terminou, e o simulador identifica automaticamente. No *display*, isso poderá ser identificado pela exibição dos registradores e do banner. Neste momento, será possível iniciar a execução através de um dos botões.

Após o envio do código, é possível controlar o fluxo de execução através dos quatro *push buttons*. São eles, da esquerda para a direita: iniciar execução contínua, passo a passo, pausar execução e resetar.

Infelizmente, não é possível visualizar o conteúdo da memória através do *display* físico. Por conta de seu tamanho pequeno e resolução baixa, os registradores e o *banner* já ocupam grande parte do espaço disponível. Para a visualização da memória, o uso de outros dispositivos de entrada/saída, assim como a disponibilidade de outros recursos como *breakpoints*, está disponível a outra forma de interagir com o simulador: a interface *web*, bem mais completa, detalhada no capítulo seguinte.

5 INTERFACE *WEB*

Neste capítulo, será introduzida a interface *web* do simulador. São detalhadas suas várias funcionalidades, muitas delas não disponíveis na interação através da *protoboard*. A seguir, é discutida a arquitetura e comunicação da interface *web* com o ESP8266, com detalhes sobre protocolos utilizados e formato de mensagens enviadas. Por fim, são apresentadas as dificuldades encontradas durante seu desenvolvimento, assim como soluções utilizadas para superá-las.

5.1 A INTERAÇÃO ATRAVÉS DA INTERFACE *WEB*

A interface *web* é a maneira mais completa de interagir com o simulador. Para acessá-la, basta acessar o endereço IP do ESP8266 através de um computador na mesma rede. Na Figura 10, é possível ver a interface *web* assim que a página é acessada. O desenvolvimento foi feito em HTML e JavaScript.

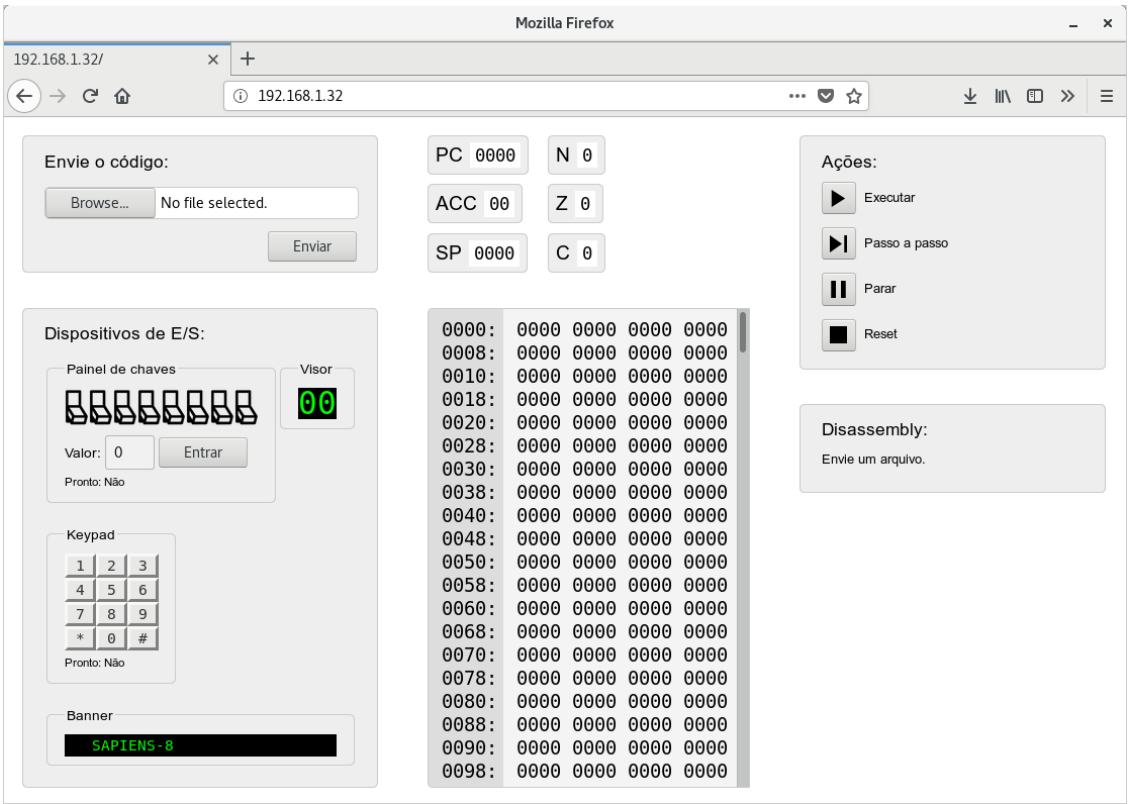

Figura 10: Interface *web* do simulador

Na esquerda, há um campo para *upload* do arquivo com o código de máquina em formato Intel HEX no topo, e os dispositivos de entrada e saída embaixo. No centro, são exibidos os registradores e a memória completa. À direita, temos no topo os botões de ações como: executar, parar e passo a passo, para controlar o fluxo de execução; e embaixo, um *disassembler*, que exibe o *assembly* do código gerado a partir do código de máquina do arquivo enviado.

O endereço 192.168.1.32 acessado foi o endereço IP atribuído ao ESP8266 pela rede. Este IP é exibido no *display*, logo que o programa é iniciado. Ao ser iniciado, é exibida a mensagem "Conectando..." no display, e é feita a tentativa de conexão na rede com os dados informados no código (nome e senha), no início do arquivo "web.ino". Depois de alguns segundos, caso a conexão seja bem sucedida, será exibida a mensagem "Conectado." e o IP do servidor *web* para ser acessado, como mostrado na Figura 11.

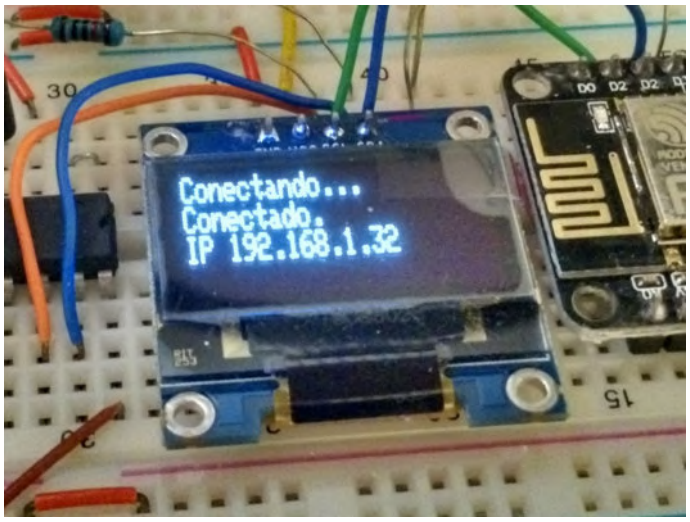

Figura 11: Display exibindo informações da conexão Wi-Fi

Todos os recursos do simulador estão disponíveis na interface *web*. São eles:

- Todos os dispositivos de E/S: painel de chaves, um visor hexadecimal, um teclado de 12 teclas e um *banner* de 1 linha com 16 caracteres;
- Toda a memória pode ser vista;
- *Disassembler* do código: a partir do conteúdo de memória carregado no simulador, é possível ver as instruções às quais os valores correspondem;

- Registradores;
- *Breakpoints*: no *disassembler*, é possível clicar numa linha de código para adicionar um breakpoint;
- *Upload* do código de forma simples.

A Tabela 3 compara os recursos do simulador e os dispositivos de entrada/-saída disponíveis fisicamente na *protoboard* e na interface *web*. Os dispositivos de entrada/saída são discutidos com mais detalhes na sessão seguinte.

	Protoboard	**Interface web**
Registradores	Sim	Sim
Memória	Não	Sim
Disassembler do código	Não	Sim
Breakpoint	Não	Sim
Painel de chaves	Não	Sim
Visor hexadecimal	Não	Sim
Teclado de 12 teclas	Sim	Sim
Banner de 16 caracteres	Sim	Sim

Tabela 3: Comparação de recursos do simulador na interface *web* e na *protoboard*

5.2 DISPOSITIVOS DE ENTRADA/SAÍDA

À esquerda da interface *web*, estão disponíveis os dispositivos de entrada e saída, como exibidos na Figura 12. Como dispositivos de entrada, há um painel de chaves, com 8 chaves que representam os bits de um número a ser inserido, e um teclado de 12 teclas. No painel de chaves, há um botão "Entrar" que define uma *flag* para informar que o valor desejado já foi definido com as chaves e pode ser lido. Isto é informado pelo status "Pronto", logo abaixo do valor. No teclado de 12 teclas, sempre que uma tecla é pressionada, o status pronto é definido, e a última tecla

pressionada pode ser lida. Como dispositivos de saída, há o visor hexadecimal de 2 dígitos e o *banner* de 16 caracteres.

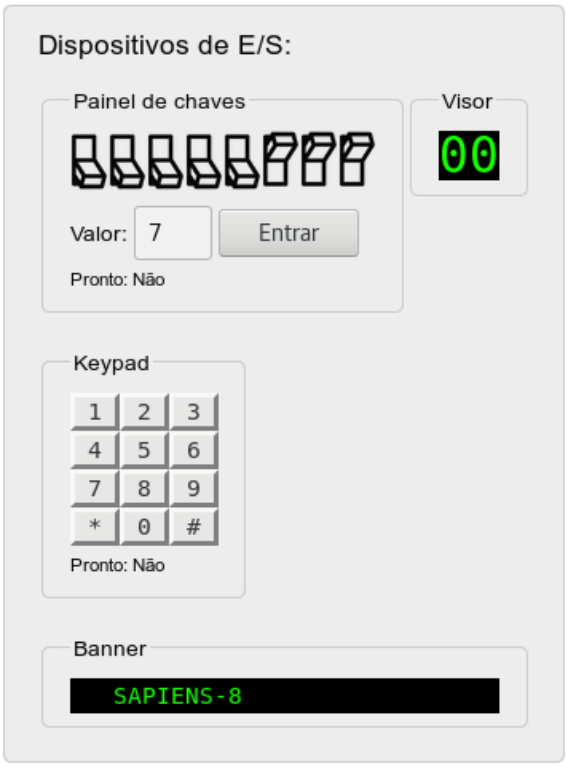

Figura 12: Dispositivos de entrada/saída

5.3 BREAKPOINTS

No canto inferior direito há o *disassembler*, que mostra o *assembly* gerado a partir do código de máquina do arquivo de entrada, com uma instrução por linha. Nele, é possível clicar numa linha para definir um *breakpoint*. Um *breakpoint* é indicado por um pequeno círculo à esquerda da linha, como mostra a Figura 13. Para removê-lo, basta clicar na linha novamente. Quando um *breakpoint* está definido, a execução contínua é interrompida logo antes da execução da instrução daquela linha.

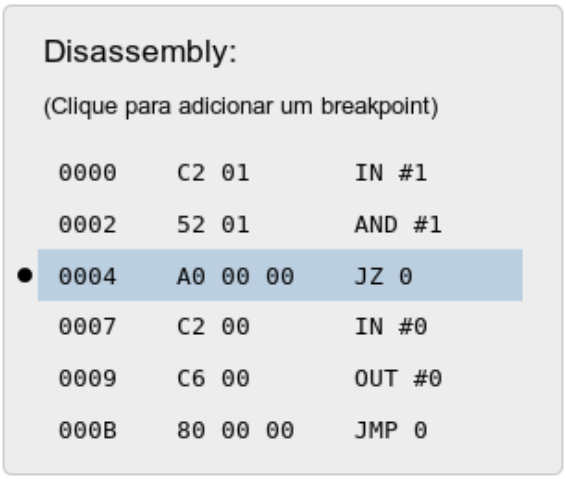

Figura 13: Exemplo do *disassembler* com um *breakpoint* inserido

5.4 ARQUITETURA

Quando o usuário acessa a interface *web* através de um navegador de um computador na rede, é feita uma requisição para o servidor HTTP do ESP8266, na porta 80. O servidor HTTP irá, então, responder a requisição com o código HTML/-JavaScript da página da interface *web*. Ao carregar esta página, será executado o código JavaScript, que abrirá uma conexão com o servidor WebSockets na página 81. Esta conexão é bidirecional e ficará aberta o tempo todo. Toda a comunicação entre a interface *web* e o ESP8266 ocorrerá através dela: sempre que o simulador alterar algum dado por conta da simulação, uma mensagem com as alterações será enviada para a interface *web*; sempre que o usuário interagir de alguma forma na interface *web* (clicando num botão ou enviando um arquivo, por exemplo), uma mensagem será enviada para o ESP8266. Na Figura 14 é possível ver um diagrama da arquitetura utilizada para a interface *web* e sua comunicação com o ESP8266.

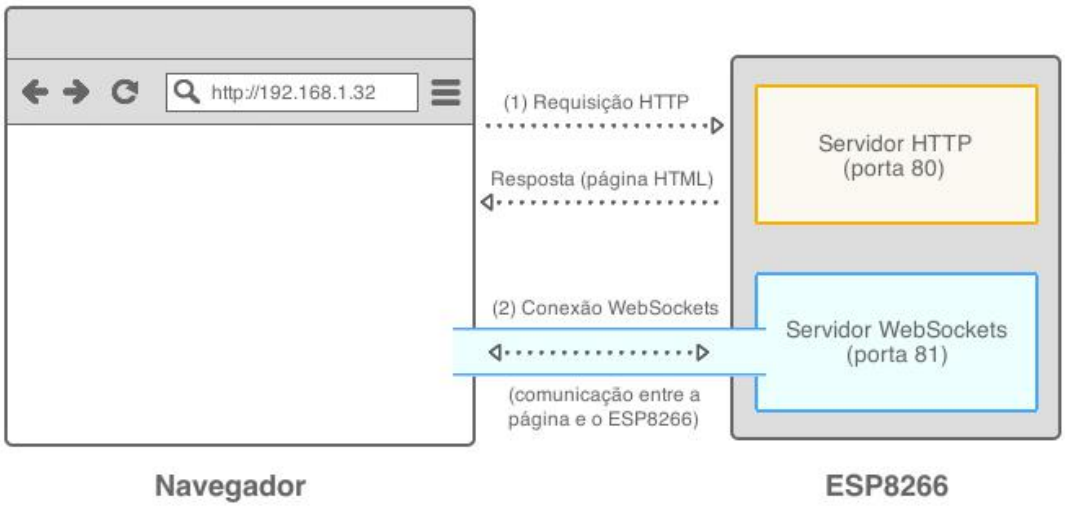

Figura 14: Diagrama da arquitetura utilizada para a interface *web*

Portanto, para o funcionamento da interface *web*, são utilizados dois servidores:

- Servidor HTTP, na porta 80, através da biblioteca *ESP8266WebServer*[1], para servir a página HTML;
- Servidor WebSockets, na porta 81, através da biblioteca *arduinoWebSockets*[2], para a comunicação bidirecional entre a página e o ESP8266.

A seguir, serão exibidos trechos do código necessário para a utilização desses servidores, com explicações detalhadas sobre seu funcionamento.

O projeto foi desenvolvido com o foco em apenas 1 cliente. Apesar de não haver problemas com a conexão de mais de 1 cliente para a visualização da simulação, a interação com o simulador deve ser feita por apenas um único cliente.

5.4.1 Servidor HTTP

Para a utilização da biblioteca *ESP8266WebServer* para a execução do servidor HTTP, é incluído no código o arquivo `ESP8266WebServer.h`, através da diretiva `#include`.

[1] `https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WebServer`
[2] `https://github.com/Links2004/arduinoWebSockets`

```
#include <ESP8266WebServer.h>
```

Após isso, é instanciado um objeto do tipo `ESP8266WebServer`, passando o número da porta para o construtor:

```
ESP8266WebServer server(80);
```

Depois, são registrados os caminhos onde são servidas as páginas, com a função `on(caminho, handler)`, que associa um caminho a uma função que irá tratar a requisição. Quando for feita uma requisição para um endereço, a função associada àquele caminho é chamada. Neste projeto, apenas o caminho raiz (`/`) é usado:

```
server.on("/", []() {
  server.send(200, "text/html", pagina);
});
```

Aqui, a função para o tratamento da requisição chama a `send(status, mime, pagina)`, que envia uma resposta para o cliente. Na função `send`, os parâmetros são, respectivamente: o código do *status* HTTP da resposta, o *mime type* da resposta, e o conteúdo. Aqui, o código de *status* é 200 (OK) e o conteúdo é o código HTML da página, salvo na *string* `pagina`.

Após o registro dos caminhos, é utilizada a função `begin()` para iniciar a execução do servidor:

```
server.begin();
```

No loop principal, a função `handleClient()` é chamada, para verificar se há alguma requisição para ser atendida:

```
server.handleClient();
```

5.4.2 Servidor WebSockets

Para a utilização da biblioteca *arduinoWebSockets* para a execução de um servidor WebSockets, é incluído no código, através da diretiva `#include`, o arquivo `WebSocketsServer.h`.

```
#include <WebSocketsServer.h>
```

Após isso, é instanciado um objeto do tipo `WebSocketsServer`, passando o número da porta para o construtor:

```
WebSocketsServer webSocket(81);
```

Depois, o servidor é iniciado com `begin()`, e é registrada uma função para tratar os eventos.

```
webSocket.begin();
webSocket.onEvent(webSocketEvent);
```

A função `webSocketEvent` é chamada pela biblioteca sempre que ocorre algum evento, tal como: nova conexão aberta, mensagem recebida. Parte dela é exibida no código abaixo.

```
void webSocketEvent(uint8_t num, WStype_t type,
                    uint8_t * payload, size_t length) {
  switch (type) {
    case WStype_CONNECTED: {
        envia_memoria_atual(true); // envia toda a memoria
        envia_io(true);
      }
      break;

    case WStype_TEXT: {
      String acao = (char *) payload;
      if (acao.startsWith("envia,")) {
        String arquivo = acao.substring(6);
        PC = 0; SP = 0; ACC = 0; ALU_N = 0; ALU_Z = 0; ALU_C = 0;
```

```
        breakpoint = -1;
        memoria.limpa();
        processa_arquivo(arquivo);
        envia_memoria_atual(true); // envia toda a memoria
      }
      else if (acao.equals("play"))   simula_play();
      else if (acao.equals("step"))   simula_step();
      else if (acao.equals("pause"))  simula_pause();
      else if (acao.equals("stop"))   simula_stop();
    }
  }
}
```

Note que o conteúdo da memória inteira só é enviado no início (evento *CONNECTED*) e ao receber um novo arquivo, depois de processá-lo.

Como pode ser visto, a função `webSocketEvent` é apenas um grande `switch`, que realiza alguma ação dependendo do tipo de evento. Os principais tipos de eventos são:

- **WStype_CONNECTED** – uma conexão foi estabelecida;
- **WStype_DISCONNECTED** – uma conexão foi fechada;
- **WStype_TEXT** – uma mensagem de texto foi recebida;
- **WStype_BIN** – uma mensagem binária foi recebida;
- **WStype_ERROR** – um erro ocorreu.

Para o envio de mensagens, é possível enviar para um cliente específico, ou fazer *broadcast*, onde a mensagem é sempre enviada para todos os clientes conectados no momento. Neste projeto, como o foco é apenas 1 cliente, o resultado seria o mesmo, pois o *broadcast* acabaria enviando apenas para este único cliente. Portanto, foi utilizado o *broadcast* por questões de simplicidade. Dessa forma, não foi preciso guardar o id do cliente e utilizá-lo sempre que for feito algum envio de mensagem.

Além disso, é possível enviar mensagens do tipo texto ou binário. Estão dispo-

níveis as funções `sendTXT`, `sendBIN`, `broadcastTXT` e `broadcastBIN`, onde as duas primeiras enviam para um cliente específico, e as duas últimas fazem *broadcast*. Foi utilizada a `broadcastTXT`, passando uma string com os dados no formato JSON. Abaixo é mostrado um exemplo de uso da função, com o envio de uma mensagem de texto com dados no formato JSON. O conteúdo dessas mensagens é discutido com detalhes na seção 5.5.2.

```
String dados =
    " {" +
    " 'registradores': {  " +
    "    'pc':  '0000',   " +
    "    'acc': 'BB',     " +
    "    'sp':  '00CC',   " +
    "    'n':   '0',      " +
    "    'z':   '1',      " +
    "    'c':   '0'       " +
    " }";

webSocket.broadcastTXT(dados);
```

No loop principal, a função `loop()` é chamada, para verificar se há alguma mensagem nova recebida:

```
webSocket.loop();
```

5.5 COMUNICAÇÃO DA INTERFACE *WEB* COM O ESP8266

Toda a comunicação entre a interface *web* e o ESP8266 é feita através do protocolo WebSockets. O protocolo WebSockets mantem uma conexão aberta, e os dois lados podem enviar mensagens. Dessa forma, não há a necessidade de fazer *pooling* a cada segundo, por exemplo, para perguntar se há dados novos [9]. No protocolo WebSockets, a conexão fica aberta, e sempre que há alguma alteração no valor de algum registrador ou posição de memória, o ESP8266 envia para a interface *web*.

Da mesma forma, sempre que alguma ação for feita na página *web*, como a inser-

ção de um dado num dispositivo de entrada/saída ou um clique no botão de iniciar ou pausar a execução, uma mensagem é enviada para o ESP8266, para que a ação seja refletida no simulador. As mensagens que a página envia para o microcontrolador seguem um protocolo próprio, detalhado a seguir.

5.5.1 Formato das mensagens enviadas da interface *web* para o ESP8266

Nos itens a seguir, são detalhadas todas as mensagens que são enviadas da interface *web* para o ESP8266. Nas tabelas abaixo, o seguinte padrão é adotado:

- Cada célula identifica um byte da mensagem.
- Células com fundo cinza denotam variáveis, e seus significados estão nas linhas seguinte.
- Células com fundo branco denotam caracteres.
- Todo fim de mensagem é indicado pelo caractere `\0`.

5.5.1.1 Alteração de I/O

Enviado sempre que algum dispositivo de I/O é alterado na interface *web*.

i	o	,	keyReg	keyStatusReg	kbdReg	kbdStatusReg	\0

- keyReg – dado do painel de chaves
- keyStatusReg – status de "dado disponível" do painel de chaves
- kbdReg – dado do *keypad* de 12 teclas
- kbdStatusReg – status de "dado disponível" do *keypad*

5.5.1.2 Breakpoint - adicionar

Enviado sempre que é definido um *breakpoint* através de um clique numa linha do *disassembler*.

b	p	,	byte1	byte2	\0

- byte1 – primeiro byte do endereço da posição do *breakpoint*

- byte2 – segundo byte do endereço da posição do *breakpoint*

5.5.1.3 *Breakpoint* - remover

Enviado sempre que é removido um *breakpoint* na interface *web* através de um clique numa linha do *disassembler* que já possuía um *breakpoint*.

b	p	n	\0

5.5.1.4 *Upload* de *dump* de memória

e	n	v	i	a	,	<conteúdo do arquivo>

- <conteúdo do arquivo> – todo o conteúdo do arquivo no formato Intel HEX

5.5.1.5 Iniciar execução (contínua)

p	l	a	y	\0

5.5.1.6 Passo a passo

s	t	e	p	\0

5.5.1.7 Pausar execução

p	a	u	s	e	\0

5.5.1.8 Parar execução e resetar

s	t	o	p	\0

5.5.2 Formato das mensagens enviadas do ESP8266 para a interface *web*

Todas as mensagens que o ESP8266 envia para a página são no formato JSON, formato tipicamente usado em aplicações de IoT. Este formato tem a facilidade de poder ser transformado num objeto nativo do JavaScript, linguagem utilizada na interface *web*, e, assim, ser manipulado de maneira bem simples. São mostrados exemplos a seguir.

5.5.2.1 Registradores e memória

Sempre que o simulador faz alguma alteração na memória, apenas as posições alteradas da memória são enviadas. Quando há apenas alterações em registradores, o valor do campo "memoria" é vazio. O inverso (apenas alterações em memória) nunca ocorre, pois sempre que alguma alteração é feita na memória, foi resultado da execução de alguma instrução, e ao menos o registrador PC será alterado.

Exemplo:

1 – Alteração apenas nos registradores:

```
{
    "registradores": {
        "pc": "00AA",
        "acc": "BB",
        "sp": "00CC",
        "n": "0",
        "z": "1",
        "c": "0"
    },

    "memoria": {
    }
}
```

2 – Alteração nos registradores e na memória:

```
{
  "registradores": {
    "pc": "00AA",
    "acc": "BB",
    "sp": "00CC",
    "n": "0",
    "z": "1",
    "c": "0"
  },

  "memoria": {
    "4": "15",
    "9": "C6",
    "7": "16"
  }
}
```

No JavaScript, o JSON pode ser transformado em um objeto nativo, e o acesso aos elementos fica bastante fácil. Por exemplo: caso o objeto se chame `dados`, para acessar a posição 4 de memória é usado apenas `dados.memoria[4]`, e para acessar o registrador PC, `dados.registradores.pc`.

5.6 OTIMIZAÇÕES

Algumas otimizações foram feitas, por conta de complicações que surgiam durante o desenvolvimento.

Inicialmente, a troca de mensagens da interface *web* com o ESP8266 era feita com requisições HTTP comuns, através de *pooling*. A cada segundo, a página enviava uma requisição, e o simulador retornava como resposta o conteúdo dos registradores e de todas as posições de memória diferentes de 0. Isso acabava sendo ruim por diversos motivos: primeiramente, era preciso fazer uma conexão HTTP por segundo, o que inclui todo o *overhead* de abrir a conexão no início, e fechar no final. Além disso, muitas vezes a conexão era feita à toa: se não houvesse nenhuma modificação, o conteúdo seria enviado de novo. Em terceiro lugar, o envio da memória inteira

adicionava um grande custo. Em programas maiores, acabava sendo uma quantidade não tão pequena de dados, para ser enviada a cada segundo.

Para contornar estes problemas, primeiramente foi utilizado o protocolo WebSockets. Neste protocolo, uma conexão é mantida aberta entre o servidor e cliente, e os dois lados podem enviar mensagem quando algum evento ocorrer [9]. Dessa forma, a conexão pôde ficar aberta sem transmitir nada a menos que seja necessário. Quando alguma mudança no valor do registrador ou da memória é feita no simulador, ele envia as alterações para a interface *web*. Não é mais preciso que a conexão seja iniciada pela interface *web*. Além disso, outra mudança foi o envio apenas das modificações da memória. Assim, a memória é enviada apenas quando há alteração, e apenas os bytes alterados são enviados. A interface *web* recebe esses dados alterados e substitui na memória salva. Somente no início é enviada a memória inteira.

Outro problema encontrado foi a exibição da memória inteira na interface *web*. Como existem 64K posições de memória, renderizar tudo isso de uma só vez é muito pesado. Nas versões iniciais, onde a memória era renderizada por completo, o navegador chegava a travar por cerca de 5 segundos no início da exibição da página.

Para solucionar isto, foi utilizada a biblioteca "Clusterize.js" [3]. Esta bilioteca auxilia na exibição de dados em quantidades muito grandes. Com ela, os elementos não são todos renderizados no início. Apenas parte dos dados é exibida, e conforme a barra de rolagem é utilizada, o resto dos dados é renderizado. Para o usuário, é totalmente imperceptível: a barra de rolagem é exibida normalmente, e ao rolar os conteúdos são exibidos da forma esperada.

5.7 COMENTÁRIOS ADICIONAIS

Vale notar que a aplicação *web* desenvolvida se encaixa fielmente no conceito de IoT. O protocolo WebSockets e o formato JSON, utilizados na comunicação entre o ESP8266 e a interface *web*, são soluções bem utilizadas no contexto de IoT [16] [25]

[3] `https://clusterize.js.org/`

[11]. Além disso, o envio apenas dos dados que foram alterados, ao invés de reenviar sempre tudo, é bastante comum em aplicações deste contexto. É interessante notar que, inicialmente, muitas destas soluções não eram utilizadas. Conforme o projeto foi sendo desenvolvido e problemas foram sendo encontrados, o projeto foi caminhando para o uso destas soluções.

6 CONCLUSÃO

6.1 VISÃO GERAL

Foi apresentado um simulador do código de máquina do processador hipotético Sapiens para o microcontrolador ESP8266. O simulador SimuS, por ser de código aberto, pôde servir de inspiração em diversos trechos do simulador desenvolvido. Foi interessante ver a união entre as áreas de arquitetura e de circuitos, sendo possível utilizar código de máquina para a interação com um circuito na *protoboard*, além da possibilidade da utilização da interface *web* para a interação com o simulador, sendo também um excelente exemplo de IoT.

Ao longo do desenvolvimento do projeto, foram encontradas algumas dificuldades. Inicialmente, o *overhead* das conexões HTTP ao se fazer *pooling* para se obter o conteúdo da memória de tempos em tempos não permitia que esta atualização fosse feita com muita frequência. Com a utilização do protocolo WebSockets, que mantem uma conexão aberta e permite o envio de mensagem pelos dois lados, e o envio apenas das modificações da memória, foi possível solucionar este problema.

Outra dificuldade foi a exibição da memória completa na interface *web*. Como existem 64K posições de memória, renderizar tudo de uma só vez era muito pesado, e o *scroll* ficava muito lento. Foi utilizada a biblioteca "Clusterize.js", que esconde a área não visível, e só renderiza a área sendo exibida, conforme o conteúdo é rolado. Com isto, a memória inteira pôde ficar disponível, e o *scroll* pôde ocorrer de forma fluida.

6.2 TRABALHOS FUTUROS

6.2.1 Novos dispositivos

Uma possível adição ao projeto seria a inclusão de novos dispositivos físicos de entrada e saída na *protoboard*. Por exemplo, o uso de alguns interruptores (*switches*) físicos para fazer um painel de chaves, e *displays* de 7 segmentos para o visor

hexadecimal.

6.2.2 Uso do ESP32

Outro possível trabalho futuro interessante seria portar o simulador para o ESP32. O ESP32 é sucessor do ESP8266, e conta com um processador melhor, mais memória, e algumas outras adições [3] [2]. Como o ESP32 possui 520 KB de RAM, muito mais do que os 64 KB de memória endereçados pelo Sapiens, não haveria os problemas de memória abordados na seção 2.3.1. Toda a memória poderia ser alocada. Além disso, o ESP32 também possui Bluetooth, abrindo um leque de novas possibilidades de dispositivos de entrada e saída.

Durante o desenvolvimento do projeto, foram feitos testes em um ESP32, para verificar se seria viável rodar o mesmo código que foi usado no ESP8266, ou ao menos um código com pequenas alterações. No entanto, foi descoberto que algumas bibliotecas utilizadas no projeto ainda não foram portadas para o ESP32. Não foram encontradas bibliotecas para o uso do protocolo WebSockets, utilizado para na comunicação do simulador com a interface *web*, nem para o uso do *keypad* de 12 teclas com o protocolo I2C. Como o ESP32 ainda é bem recente, ainda não conta com toda a comunidade de desenvolvedores e disponibilidade de bibliotecas e códigos que o ESP8266 já tem.

6.2.3 Melhorias na interface *web*

Algumas adições à interface *web* que seriam relevantes são, primeiramente, a possibilidade da edição da memória, como existe no simulador SimuS, através de um clique na posição de memória. Ao clicar numa posição, um campo seria exibido, para que fosse possível inserir um novo valor para aquele endereço. Seria interessante, inclusive, haver um "mini *assembler*", que permitisse que o usuário digitasse a instrução em linguagem de montagem, para que os bytes fossem modificados de acordo.

Outra mudança que poderia ser útil à interface *web* seria a utilização de biblio-

tecas apontando para CDNs (*Content Delivery Network*), em vez de incluir todo o código das bibliotecas no arquivo HTML servido. Dessa forma, as bibliotecas seriam baixadas da internet pelo navegador na hora do acesso à interface *web*, e não seria necessário que todo o código da biblioteca fosse armazenado na memória limitada do NodeMCU. Isso inclusive abriria portas para o uso de mais bibliotecas em JavaScript na interface *web*, o que seria importante conforme novas funcionalidades sejam adicionadas. No projeto desenvolvido, todo o código JavaScript era todo armazenado junto com o HTML, e a maior parte dele era JavaScript nativo (sem a utilização de bibliotecas), visando sempre diminuir o espaço usado para armazenar o código da página.

6.2.4 Novos tipos de TRAP

Outra adição interessante seria a inclusão de novos tipos de TRAP. Além das TRAPs desenvolvidas, como leitura e escrita de pinos digitais, seria interessante implementar a leitura de valor analógico. Dessa forma, seria possível ler o valor de sensores, por exemplo.

Também seria interessante uma TRAP para o uso da interface I2C, e, a partir disto, utilizar o display. No circuito montado para o projeto, o display é usado para exibir informações dos registradores; porém, caso seja utilizada a interface *web*, estas informações já podem ser vistas por lá e o display se torna desnecessário. Nestes casos, seria interessante o usuário ter a possibilidade de desenhar ou escrever no display, por exemplo.

Por fim, TRAPs para o controle de dispositivos específicos também seriam uma boa adição. Controlar, por exemplo, um motor servo, especificando o pino e a posição desejada, seria uma boa funcionalidade. Este controle poderia inclusive ser implementado através de bibliotecas do Arduino já existentes, já que o projeto foi desenvolvido com a IDE do Arduino.

6.2.5 Avaliação qualitativa do uso

O simulador desenvolvido pode ser usado para auxiliar o ensino em disciplinas de graduação como Arquitetura e Internet das Coisas. Seria bem importante obter uma avaliação qualitativa do uso do simulador por parte dos alunos, visando identificar pontos que podem ser melhorados no projeto desenvolvido, para que alunos seguintes pudessem ter uma experiência ainda melhor.

6.3 OBSERVAÇÕES FINAIS

Durante os seis meses de desenvolvimento do projeto, foram acumuladas diversas experiências. Primeiramente, foi possível observar e entender de perto os detalhes de um simulador de código de máquina, vendo que uma idéia simples e organizada pode levar a um resultado grande e significativo. Também foi interessante notar que existem dispositivos de tão baixo custo que integram tecnologias como WiFi, tornando o desenvolvimento de projetos na área de IoT bem viável e barato. Por fim, foi proveitoso trabalhar num ambiente onde a preocupação com a memória é real. No dia a dia, é comum a utilização de ambientes onde há memória de sobra; então, o trabalho com memória restrita foi uma experiência diferente do padrão.

Com este projeto, esperamos que seja despertado um maior interesse dos alunos em disciplinas de Arquitetura de Computadores e de Circuitos Lógicos. Agora, é possível interagir com dispositivos físicos na *protoboard* a partir do código de máquina do processador Sapiens, e os limites vão além do mundo virtual do computador. A placa de desenvolvimento de baixíssimo custo permite que estudantes e professores possam adquirir com facilidade, e utilizar em seus cursos. A partir de modificações no circuito proposto, é simples adicionar no projeto outros dispositivos físicos e utilizar programas simples escritos em linguagem de montagem para interagir com eles. Todos os códigos do projeto desenvolvidos estão disponíveis no GitHub[1], e são livres para serem modificados e utilizados.

[1] `https://github.com/edunmc/tcc-ufrj`

REFERÊNCIAS

[1] ESP-12E WiFi Module. Disponível em: <https://www.kloppenborg.net/images/blog/esp8266/esp8266-esp12e-specs.pdf>. Acesso em: 06 maio 2018.

[2] ESP32 Datasheet: Version 2.3. 2018. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf>. Acesso em: 06 maio 2018.

[3] ESP32 Overview. 2018. Disponível em: <https://www.espressif.com/en/products/hardware/esp32/overview>. Acesso em: 06 maio 2018.

[4] ESP8266 12e WIFI MODULE. 2017. Disponível em: <https://modernelectronics.com.pk/product/esp8266-12e-wifi-module/>. Acesso em: 29 abr. 2018.

[5] ESP8266EX Datasheet. 2018. Disponível em: <https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf>. Acesso em: 06 maio 2018.

[6] Guide to PROGMEM on ESP8266 and Arduino IDE. 2017. Disponível em: <https://gist.github.com/sticilface/e54016485fcccd10950e93ddcd4461a3>. Acesso em: 15 abr. 2018.

[7] NodeMCU DEVKIT V1.0. 2018. Disponível em: <https://github.com/nodemcu/nodemcu-devkit-v1.0/blob/master/README.md>. Acesso em: 06 maio 2018.

[8] SSD1306. 2008. Disponível em: <https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf>. Acesso em: 06 maio 2018.

[9] WebSockets. 2018. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API>. Acesso em: 06 maio 2018.

[10] Espressif FAQ. 2016. Disponível em: <https://www.espressif.com/sites/default/files/documentation/espressif_faq_en.pdf>, 2016. Acesso em: 29 abr. 2018.

[11] Ada, L. All the Internet of Things - Episode Two. 2017. Disponível em: <https://learn.adafruit.com/alltheiot-protocols?view=all#whats-the-difference>. Acesso em: 06 maio 2018.

[12] Casner, D. Guidelines for writing code for the ESP8266. 2015. Disponível em: <http://www.danielcasner.org/guidelines-for-writing-code-for-the-esp8266/>. Acesso em: 29 abr. 2018.

[13] Edwards, S. A. Apple2fpga: Reconstructing an Apple II+ on an FPGA. 2007. Disponível em: <http://www.cs.columbia.edu/~sedwards/apple2fpga/>. Acesso em: 29 abr. 2018.

[14] Filippov, M. Memory Map. 2015. Disponível em: <https://github.com/esp8266/esp8266-wiki/wiki/Memory-Map>. Acesso em: 29 abr. 2018.

[15] Intel. Hexadecimal Object File Format Specification. 1998. Disponível em: <https://web.archive.org/web/20160607224738/http://microsym.com/editor/assets/intelhex.pdf>. Acesso em: 15 abr. 2018.

[16] Madan, D. Unleashing the power of HTML5 WebSocket for Internet of Things. 2015. Disponível em: <https://www.hcltech.com/blogs/unleashing-power-html5-websocket-internet-things-iot>. Acesso em: 06 maio 2018.

[17] ESP8266. Arduino/pins_arduino.h - esp8266/Arduino. 2018. Disponível em: <https://github.com/esp8266/Arduino/blob/master/variants/nodemcu/pins_arduino.h#L37-L59>. Acesso em: 06 maio 2018.

[18] Moreira, A. A.; Martins, C. A. P. S. R2DSim: Simulador Didático do RISC Reconfigurável. In: WORKSHOP SOBRE EDUCAÇÃO EM ARQUITETURA DE COMPUTADORES, 2009, São Paulo. Anais... [S.l.: s.n.], 2009.

[19] Ranganathan, A. Gnusim8085. 2018. Disponível em: <https://launchpad.net/gnusim8085>. Acesso em: 29 abr. 2018.

[20] Seeed Technology Co., L. NodeMCU v2 - Lua based ESP8266 development kit. 2017. Disponível em: <https://www.seeedstudio.com/NodeMCU-v2-Lua-based-ESP8266-development-kit-p-2415.html>. Acesso em: 29 abr. 2018.

[21] Silva, G. P.; Borges, J. A. dos S. Neanderwin - Um Simulador Didático para uma Arquitetura do Tipo Acumulador. 2006. Disponível em: <http://dcc.ufrj.br/~gabriel/WEAC2006.pdf>. Acesso em: 15 abr. 2018.

[22] Silva, G. P.; Borges, J. A. dos S. Simus: Um Simulador Para o Ensino de Arquitetura de Computadores. 2016. Disponível em: <http://www2.sbc.org.br/ceacpad/ijcae/v5_n1_dec_2016/IJCAE_v5_n1_dez_2016_paper_2_vf.pdf>. Acesso em: 15 abr. 2018.

[23] Silva, G. P.; Borges, J. A. dos S. SimuS: Um Simulador Didático para Arquitetura de Computadores. Rio de Janeiro: Ed. do Autor, 2017.

[24] Verona, A. B.; Martini, J. A.; Gonçalves, T. L. SIMAEAC: Um simulador acadêmico para ensino de arquitetura de computadores. 2009. Disponível em: <https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxub3ZvbWlsZW5pbzIwMTQwMXxneDo0MTE2YzA3M2JjMzYz%20Yjhi>. Acesso em: 29 abr. 2018.

[25] Viswanath, A. How JSON and Big Data Will Shape the Internet of Things. 2013. Disponível em: <http://java.sys-con.com/node/2881856>. Acesso em: 06 maio 2018.

[26] Ziller, R. Microprocessadores: Conceitos Importantes. 2. ed. Florianópolis: R. M. Ziller, 2000.