# Capítulo 3

# Uma Abordagem Prática para Aprendizagem em Arquitetura e Organização de Computadores com Apoio do Simulador Computacional CompSim

Guilherme Álvaro Rodrigues Maia Esmeraldo (IFCE), Eduardo Carlos Pereira da Silva Proto (IFCE), Edson Barbosa Lisboa (IFS) e Edna Natividade da Silva Barros (UFPE)

***Abstract***

*This work presents a teaching-learning approach supported by a simulation environment for the discipline of Computer Architecture and Organization. In this chapter, initially, the discipline and its particularities are presented, justifying the need for simulation to support practical learning in design of computational systems. Following, the main concepts of Computer Organization and Architecture are worked on in the context of CompSim, a simulation environment with integrated graphical resources that simplify the design of new virtual (developed in software) or mixed (which can interact with hardware using Arduino platforms) computational systems.*

***Resumo***

*Este trabalho apresenta uma abordagem de ensino-aprendizagem com suporte de ambiente de simulação para a disciplina de Arquitetura e Organização de Computadores. Neste capítulo, inicialmente, apresenta-se a disciplina e suas particularidades, justificando a necessidade de simulação para apoiar o aprendizado prático em projetos de sistemas computacionais. Na sequência, são trabalhados os principais conceitos de Organização e de Arquitetura de Computadores no contexto do CompSim, um ambiente de simulação com recursos gráficos integrados que simplificam o projeto de novos sistemas computacionais virtuais (desenvolvidos em software) ou mistos (que podem interagir com hardware físico utilizando plataformas Arduino).*

## 3.1. Introdução

O computador é um sistema de alta complexidade, composto de hardware e software. Devido à alta escala de integração, os subsistemas que compõem o hardware podem incluir bilhões de componentes eletrônicos menores e elementares, tornando-se exponencialmente complexo, o que impacta no seu estudo e entendimento. Assim, didaticamente, o computador pode ser definido e estudado de diferentes maneiras, como, por exemplo, em: 1) Stallings (2010), que cita que o computador é um sistema eletrônico que possui uma estrutura com subsistemas interrelacionados, e que cada subsistema também pode ser subdividido em novos subsistemas, formando uma estrutura hierárquica. Dessa forma, pode-se estudar e projetar cada subsistema independentemente e em momentos distintos, considerando sua estrutura e funções que serão desempenhadas; e 2) Tanenbaum e Austin (2013), que consideram que o computador é estruturado em camadas, e que, em cada camada, há uma linguagem específica para a programação do computador. Nesse contexto, entende-se que a linguagem de uma determinada camada depende da linguagem da sua camada antecessora. Assim, um programador que cria um programa de computador utilizando a linguagem de uma determinada camada, não precisa se preocupar com tradução para linguagens das camadas subjacentes.

Analisando as duas abordagens apresentadas, pode-se concluir que, de uma forma geral, um computador é um sistema eletrônico complexo com funções programáveis, e, dependendo da abordagem metodológica empregada, o seu estudo e aprendizado pode ser impactado.

### 3.1.1. Arquitetura e Organização de Computadores

Arquitetura e Organização de Computadores (AOC) é uma disciplina presente em cursos técnicos e superiores nas áreas de Computação e Engenharias Elétrica, Eletrônica e Mecatrônica, Automação Industrial, dentre outras [SBC 2005][ACM and IEEE 2013]. A disciplina inclui o estudo dos componentes do computador, das suas funções e dos modelos de comunicação, bem como dos aspectos visíveis ao programador [Stallings 2010]. Esses conhecimentos são fundamentais à operação, programação, projeto e otimização de desempenho de sistemas computacionais, além de estarem alinhados com as novas tendências tecnológicas, tais como: Internet das Coisas (IoT), Indústria 4.0, Smart Cities, Robótica, Computação de Alto Desempenho, entre outras.

Arquitetura de computadores trata dos aspectos visíveis ao programador, que são os aspectos que tratam da execução lógica dos programas de computador. Já a Organização de computadores trata dos componentes do computador, suas estruturas e submódulos, suas funções e as formas de comunicação entre si [Stallings 2010]. É importante observar que os conteúdos trabalhados na disciplina variam de acordo com o perfil de cada curso. Por exemplo, em cursos de graduação em Engenharia da Computação, há necessidade de um maior aprofundamento teórico-prático, o que já não ocorre em cursos de Sistemas de Informação.

As diretrizes curriculares da ACM e IEEE [ACM and IEEE 2013] incluem habilidades práticas que devem ser necessariamente desenvolvidas pelos estudantes, com objetivo de promover o aprofundamento teórico e permitir que os estudantes

possam explorar as características dos sistemas mais modernos [Nikolic et al. 2009]. O desenvolvimento das habilidades práticas pelos estudantes em AOC necessita de laboratórios de hardware especializados, com disponibilidade de componentes de hardware, ferramentas para manuseio e instrumentos de medição de diferentes grandezas em componentes eletrônicos, entre outros. Percebe-se assim que, compor e manter um laboratório desse porte exigirá maiores recursos financeiros e não é uma tarefa simples, pois necessitará de um projeto estrutural bem elaborado, de apoio de técnico de laboratório para preparação e acompanhamento dos experimentos, de manutenção dos equipamentos e de reposição dos componentes eletrônicos.

Este capítulo tem como objetivo apresentar uma abordagem prática de ensino-aprendizagem em AOC com suporte do simulador CompSim [CompSim 2021]. O Simulador CompSim é um ambiente virtual que inclui as principais características dos simuladores da literatura [Esmeraldo et al. 2019] e recursos gráficos integrados que permitem criar, programar, simular, visualizar e avaliar o desempenho de novos sistemas computacionais. Além disso, o CompSim traz suporte para integração do ambiente de simulação com a plataforma de prototipação Arduino, o que permite a criação de projetos de sistemas computacionais mistos (que envolvem tanto software quanto hardware físico). Este suporte minimiza a necessidade de uso de laboratórios especializados, tornando o simulador CompSim uma ferramenta com grande potencial para exploração e aplicação prática dos conceitos estudados na disciplina de AOC.

O uso de simuladores computacionais, ou ambientes virtuais, como prática pedagógica complementar, não é uma atividade nova [Balamuralithara and Woods 2009]. Estudos, como os apresentados em [Uribe et al. 2016] [Garcia, Pacheco and Garcia 2014] [Balamuralithara and Woods 2009], mostram que, ao se utilizar ambientes virtuais para fins educacionais, foi possível aumentar o desempenho acadêmico em cursos de tecnologia e engenharia. Os simuladores são ferramentas importantes no processo de apropriação do conhecimento, pois possibilitam o desenvolvimento de habilidades e experiências práticas, de forma assíncrona, em cenários virtuais que se assemelham aos reais [Wolffe et al. 2002]. São também fundamentais para compor laboratórios específicos na ausência de infraestrutura [Garcia, Pacheco and Garcia 2014] ou ainda quando há necessidade de se reduzir custos, realizar configurações rápidas e obter resultados instantâneos [Uribe et al. 2016]. Por fim, os simuladores são particularmente úteis para representação ou abstração de cenários complexos [Bahk et al. 2013] e frequentemente abordam os conteúdos presentes no estado da arte [Wolffe et al. 2002].

### 3.1.2. Uma Proposta de Abordagem Metodológica

Considerando as diferentes ementas, contextos locais e cursos de computação, a disciplina de AOC pode sofrer variações quanto à extensão e verticalização de seus conteúdos. Desta forma, sugere-se a leitura do trabalho em [Lisboa et al. 2019], o qual apresenta uma proposta de metodologia de ensino-aprendizagem que aborda, de forma flexível, um subconjunto de conteúdos comuns aos cursos de AOC, tais como: introdução à aritmética computacional, componentes do computador e suas funções, conjunto de instruções do processador, modos de endereçamento, entrada/saída, programação em baixo nível e análise de desempenho.

### 3.1.3. Conteúdo Programático

Este capítulo apresenta o simulador CompSim como um ambiente integrado com diferentes recursos para dar suporte aos processos de ensino-aprendizado prático em AOC. O capítulo está dividido da seguinte maneira:

**Seção 3.2 - Introdução ao Simulador CompSim**: esta seção apresenta brevemente o CompSim, destacando os principais recursos para apoio ao aprendizado em AOC;

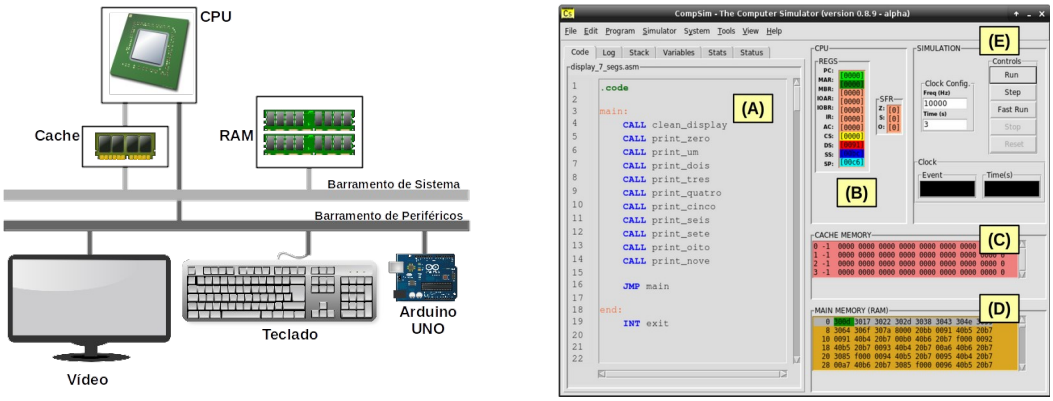
**Seção 3.3 - Organização de Computadores com o CompSim**: nesta seção, realiza-se uma breve introdução à organização de computadores, destacando as características dos principais componentes do computador (Processador, memórias RAM e Cache, Barramento e Subsistema de Entrada/Saída) no contexto do CompSim;

**Seção 3.4 - Arquitetura de Computadores com o CompSim**: esta seção trata de aspectos de arquitetura de computadores, tais como: modelo de memória e estrutura de um programa de computador; alocação e manipulação de dados em memória; processamento de dados; controle de fluxo de execução; modos avançados de acesso à memória; entrada/saída de dados; e modularização de programas; e

**Seção 3.5 - Conclusões**: Por fim, serão abertos espaços para discussões acerca do tema e de novos trabalhos.

## 3.2. Introdução ao Simulador CompSim

CompSim consiste de um ambiente virtual para apoio ao ensino-aprendizado em AOC [Esmeraldo and Lisboa 2017]. Ele segue a abordagem de projetos baseados em plataforma [Keutzer et al. 2000], na qual há uma Plataforma de Hardware Virtual (PHV) simulável e customizável, baseada em microprocessador, que inclui os principais componentes do computador, tais como processador, memórias, barramentos e periféricos, como pode ser vista na Figura 3.1(a).




**(a) PHV do CompSim.** **(b) Interface Gráfica do CompSim.**

**Figura 3.1. PHV e Interface Gráfica do Simulador CompSim.**

O simulador também conta com uma interface gráfica para dar suporte à configuração dos componentes da PHV, ajustar parâmetros de simulação e suportar a codificação de aplicações em baixo nível (linguagem Assembly). A Figura 3.1(b) mostra a interface gráfica do CompSim.

Na Figura 3.1(b) pode-se visualizar os seguintes componentes gráficos: A) Editor de código: inclui recursos para simplificar a codificação de uma aplicação, como número de linhas, teclas de atalho para recursos de edição (*copy*, *cut*, *paste*, *undo*/*redo*, *got to line*, etc.), um assistente de codificação (permite auxiliar a construção de instruções de código com a sintaxe correta), destaque de palavras-chave (*syntax highlight*), entre outros. Este componente está integrado a um montador (Assembler) que realiza análises léxica, sintática e semântica no código-fonte da aplicação, tradução deste para *bytecodes*, carregamento dos *bytecodes* na memória RAM, além de gerar um relatório do programa, que inclui a tabela de símbolos, endereços de memória dos segmentos e *bytecodes* gerados; B) Processador: durante uma simulação, exibe os registradores do processador e respectivos valores assumidos. Os registradores de endereçamento possuem cores diferenciadas, onde as respectivas cores são utilizadas para indicar as diferentes posições referenciadas na memória RAM; C) Memória cache: exibe as linhas da cache e respectivos valores, destaca também as linhas e as palavras endereçadas pelo processador durante uma simulação; D) Memória RAM: exibe os conteúdos (instruções e dados) de todos os endereços do componente virtual memória RAM (os componentes gráficos Memória RAM e processador estão vinculados, de forma que, durante uma simulação, as posições de memória são destacadas de acordo com as respectivas cores dos registradores de endereçamento); e E) Componentes de controle de configuração e execução de simulação: inclui controles que permitem configurar o tempo total de simulação e a frequência de relógio de sistema (*clock*), iniciar, executar (em modos normal, passo a passo ou rápido), parar e reiniciar uma simulação. Além desses, o CompSim conta com recursos para: registro de *logs* de simulação, acompanhamento dos status da pilha e variáveis do programa, geração de gráficos estatísticos após uma simulação, conversor de números inteiros, com e sem sinal, entre bases decimal, hexadecimal e binária, conversor de código-caractere ASCII, entre outros.

O simulador CompSim pode ser obtido no website do projeto [CompSim 2021], na seção "*Download*", e é distribuído para os sistemas operacionais MS/Windows e GNU/Linux. Após o download, a instalação consiste apenas em descompactar o arquivo obtido.

## 3.3. Organização de Computadores com o CompSim

Um computador é um dispositivo eletrônico que desempenha quatro funções principais: processamento, armazenamento e transferência de dados, bem como controle de operações, que envolve a coordenação das demais funções para permitir que os computadores realizem tarefas variadas [Stallings 2010]. Um computador é dividido em diferentes subsistemas, ou componentes, onde cada um deles possui uma estrutura e comportamento bem definidos. Assim, esses componentes devem se comunicar para trocar informações visando compor e realizar as funções do computador.

As subseções a seguir trazem as características básicas dos principais componentes do computador, que são Processador, Memória Principal, Memória Cache, Barramento e Subsistema de Entrada/Saída (E/S), no contexto dos componentes da PHV do CompSim.

### 3.3.1. Processador

O processador (*Central Processing Unit* - CPU) é o elemento central de qualquer computador. Ele é responsável por buscar instruções na memória, decodificá-las para compreender as tarefas que devem ser realizadas e executá-las [Tanenbaum and Austin 2013]. Além disso, ele interage com todos componentes do computador, dentre eles a memória e os periféricos. A CPU é o componente mais complexo do computador e, portanto, possui diversos subsistemas.

O CompSim inclui uma CPU, denominada de processador "Cariri", que apresenta as principais características de processadores reais. Dentre suas características, pode-se destacar:

- Possui arquitetura de 16-bits baseada em acumulador, onde, na maioria das suas operações, utiliza-se implicitamente o registrador de propósito geral, chamado "Acumulador" (AC). Por exemplo, em uma operação de adição, que necessita de dois operandos, a instrução deve referenciar apenas um dos operandos, pois considera-se que o outro operando já está contido em AC. Arquiteturas baseadas em acumulador são mais simples de projetar e programar, pois as instruções do processador possuem tamanhos e complexidade reduzidos [Null and Lobur 2009];
- Possui uma Unidade Lógica e Aritmética (ULA), responsável por realizar o processamento de dados, e uma Unidade de Controle (UC), que é responsável por coordenar todas as operações do processador;
- Possui um circuito Contador, que é um componente utilizado para incrementar automaticamente os endereços das instruções, contidos no registrador Contador de Programa (*Program Counter* - PC), que serão buscadas pelo processador na memória principal. Com suporte do contador, as instruções do programa são buscadas e executadas de forma sequencial;
- Conta com um Banco de Registradores, para suportar diferentes ações do processador, tais como: busca na memória e decodificação de instruções, operações lógicas e aritméticas, operações de E/S, de acesso à memória e à pilha de programa para armazenamento de dados, entre outras;
- Possui espaço de endereçamento diferenciado para operações de E/S e de acesso à memória principal. É importante destacar que, em muitos processadores, há apenas um espaço de endereços para acesso à memória e aos periféricos;
- Conta com 16 instruções de baixo nível, para diferentes operações tais como: transferência de dados, operações aritméticas e lógicas, controle de fluxo de programa e de sistema. As instruções são de 16-bits, sendo que 4-bits são reservados para o código de operação, ou seja, determinam a operação da instrução, e os demais 12-bits são reservados para o operando da instrução;

- Suporta dois tipos de operandos: inteiro de 16-bits com sinalização (*signed int*) e caractere (*char* ou *byte*); e
- Suporta os modos de endereçamento de dados: 1) imediato: onde o operando está na própria instrução; 2) direto: onde o operando pode ser acessado pelo endereço de memória na instrução (há um acesso à memória para busca do operando); 3) indireto: a instrução contém um endereço de memória que aponta para um outro endereço de memória, que é o endereço do operando (há dois acessos à memória para busca do operando); 4) registrador: o operando está contido em um registrador; 5) implícito: não há necessidade de informar onde está o operando.

### 3.3.2. Memória Principal

A memória principal (*Random Access Memory* - RAM) é o componente do computador onde as instruções e os dados de um programa em execução estão temporariamente armazenados. Para a execução de um programa, o processador deve ler instruções e trocar dados com a memória RAM [Stallings 2010]. As memórias RAM frequentemente são organizadas em matrizes, que não necessariamente precisam ser quadradas, onde cada célula é utilizada para armazenamento de um dado [Tanenbaum and Austin 2013]. Desta forma, um endereço de memória RAM é dividido em duas partes, sendo uma para seleção da linha e a outra para a coluna da matriz de dados, de forma a tornar disponível uma célula de armazenamento para leitura ou escrita de dados.

Na memória principal do CompSim, o modelo de endereçamento proposto utiliza endereços de 12-bits, sendo que os 9 bits mais significativos são utilizados para endereçamento de linha e os 3-bits menos significativos para endereçamento de coluna. Com essa configuração, é possível endereçar até 512 ($2^9$) linhas de memória, onde cada linha possui 8 ($2^3$) colunas, totalizando assim 4096 (512 x 8) diferentes endereços. Como a memória do CompSim armazena palavras de 16-bits - o termo "palavra", ou "*word*", refere-se a um conjunto de bits ou bytes, que consiste da unidade de informação que pode ser armazenada, transmitida e/ou processada em um computador -, sua capacidade de armazenamento total é de 65.536 (512 x 8 x 16) bits, ou simplesmente 8 KBytes.

### 3.3.3. Memória Cache

Na execução de um programa, o processador busca instruções e dados na memória RAM. Algumas instruções, por exemplo, necessitam realizar mais acessos à memória para busca dos operandos. E, considerando que um acesso à memória RAM leva muito tempo (consome vários ciclos de relógio de sistema), comparado à taxa de execução do processador, percebe-se que os acessos podem degradar o desempenho do processador e, por consequência, do sistema. Em resumo, o processador atua em uma taxa de execução muito mais alta do que a taxa de entrega de dados da memória RAM, caracterizando assim um modelo de comunicação frágil (muitos autores utilizam o termo "gargalo de comunicação") e que tem grande impacto no desempenho do sistema [Monteiro 2007]. Buscando otimizar o desempenho de comunicação Processador/Memória RAM, os projetistas de computadores criaram um tipo de

memória de armazenamento temporário, localizada entre o processador e a memória RAM e que é fabricada com a mesma tecnologia do processador, chamada de Memória Cache. A memória cache tem como função acelerar a entrega de instruções e dados, minimizando assim os ciclos de espera pelo processador.

A memória cache tem uma organização diferente da memória RAM. Enquanto que a memória RAM inclui uma matriz para armazenamento de dados, a memória cache armazena os blocos transferidos da memória RAM em linhas. Como a capacidade de armazenamento de blocos da memória cache é bastante inferior à da memória RAM, cada linha da cache pode ser utilizada para armazenar diferentes blocos, em momentos distintos. A abordagem de escolha de armazenamento de determinado bloco de memória RAM em uma determinada linha da memória cache se chama Mapeamento. Para maximizar o desempenho do sistema pelo uso de memórias Cache, é importante que todas as linhas da Cache estejam ocupadas com blocos transferidos da memória RAM. Porém, dependendo do(s) programa(s) em execução, haverá necessidade de retirar blocos armazenados na memória cache para ceder espaço para novos blocos que estão sendo transferidos (esse processo é conhecido como Substituição). Ao modificar um dado na memória Cache, é necessário manter a coerência com respectivo dado presente na memória RAM, daí deve-se utilizar alguma Política de Escrita (ou Política de Atualização) em memória cache.

A memória cache do CompSim possui as seguintes características: 1) Número de linhas parametrizável: é possível configurar o número total de linhas da memória cache; 2) Bloco de 8 palavras: cada linha poderá armazenar 8 palavras. A memória RAM do CompSim armazena 8 palavras de 16-bits por linha da sua matriz de dados. Assim, uma linha da memória cache poderá armazenar os dados contidos em uma linha da memória RAM; 3) Técnicas de Mapeamento: são suportadas as técnicas de Mapeamento Direto, Associativo e Associativo por Conjunto; 4) Algoritmos de Substituição: são suportadas as técnicas de substituição Menos Recentemente Usado (*Least Recently Used* - LRU), Fila Circular (*First-In First-Out* - FIFO) e Aleatório (*Random*); e Políticas de Escrita: são suportadas as políticas de escrita com acerto *Write-Through* e *Write-Back*, e as políticas de escrita com falta *Write-Allocate* e *Write-Around*.

### 3.3.4. Barramento

Um barramento é um recurso utilizado para conectar os componentes do computador [Null and Lobur 2009] e possibilita a comunicação de dados e controle entre eles. Os barramentos compartilhados são subdivididos em três tipos: 1) Endereço: é utilizado para informar um endereço de algum componente para realizar algum tipo de comunicação; 2) Controle: é utilizado para informar o tipo de comunicação que será realizada com o componente endereçado no barramento de endereço; e 3) Dados: é utilizado para a transferência de dados entre os componentes comunicantes. As larguras, ou os números de linhas ou de sinais, dos barramentos de dados e endereços, são parâmetros importantes que têm grande impacto no desempenho de comunicação e capacidade de endereçamento do sistema computacional. A largura do barramento de dados define a quantidade de bits que podem ser transferidos em paralelo em uma única comunicação. Já a largura do barramento de endereços define a quantidade de unidades

endereçáveis, quer sejam o total de endereços de memória ou número de periféricos. Os barramentos podem ser classificados em Síncronos, que são aqueles onde a sequência de eventos de uma transmissão de dados (transação) é controlada por eventos gerados pelo relógio do sistema (*clock*); e Assíncronos, cujas linhas de controle coordenam a transmissão de dados através de um protocolo de comunicação (*handshaking*).

No CompSim, como o processador Cariri possui espaços de endereçamento diferenciados para comunicação com a memória e com os periféricos, a PHV inclui um barramento síncrono de sistema para conectar processador, memória cache e memória RAM, e um barramento assíncrono de periféricos para conectar o processador Cariri ao subsistema de E/S para se comunicar com os periféricos. O barramento de sistema possui largura de barramentos de dados de 16-bits (comunicações de 2 bytes) e de endereços de 12-bits (endereça até 4.096 posições de memória RAM) e suporta as operações de leitura e escrita de dados; enquanto que o barramento de periféricos possui larguras de barramentos de dados e de endereços de 8-bits (comunicações de 1 byte e endereçamento de até 256 periféricos) e também suporta operações de leitura e escrita.

### 3.3.5. Subsistema de E/S

Os periféricos são componentes fundamentais em computadores, pois são eles que possibilitam a interação entre o computador e o usuário (*Human Readable*), o computador e outros sistemas computacionais (*Machine Readable*) e comunicação entre dispositivos remotos (*Communication*). Cada tipo de periférico possui características próprias, tais como tecnologia de fabricação, funcionalidades, mecanismos de operação, taxas de transferência de dados, formato de dados e tamanhos de palavra [Stallings 2010]. O processador se comunica com os periféricos através do barramento de periféricos, o qual, por sua vez, não possui capacidade para realizar conexão com as interfaces nativas de cada um dos tipos de periféricos. Em outras palavras, é necessário que cada periférico, independentemente da sua natureza, inclua um mecanismo que permita sua conexão com uma interface padronizada presente no barramento de periféricos. Essa interface com o barramento de periféricos é conhecido como Subsistema de E/S, Módulo de E/S ou ainda Controlador do Periférico.

O simulador CompSim inclui um subsistema de E/S conectado ao barramento de periféricos da PHV. Esse subsistema de E/S permite que novos periféricos sejam conectados de forma automática ao barramento de periféricos, bastando que o periférico implemente a interface de comunicação padrão com o barramento. Quanto aos periféricos, é possível ter periféricos do tipo virtual (que são aqueles implementados em software e emulam o comportamento de periféricos reais), e do tipo físico (que são divididos em software, para implementar a interface padrão com o barramento, e em hardware para compor o periférico físico em si e seu comportamento).

## 3.4. Arquitetura de Computadores com o CompSim

A CPU é o componente mais importante do computador. Segundo Tanenbaum e Austin (2013), ela é considerada o "cérebro" do computador, pois sua função é executar os programas armazenados na memória RAM, buscando suas instruções, interpretando-as e executando-as uma a uma, de maneira sequencial.

Independentemente da linguagem de programação utilizada para criar os programas de computador, a CPU somente reconhece o seu próprio conjunto de instruções da arquitetura (*Instruction Set Architecture* - ISA). Assim, para que seja possível executar os programas criados em linguagem de alto nível pelos programadores, suas instruções devem ser traduzidas, por um compilador, para um novo programa que estará descrito em termos da ISA da CPU. Algumas CPUs possuem uma ISA reduzida, com instruções simples, de tamanho fixo e que requerem poucos ciclos de *clock* para serem executadas (abordagem *Reduced Instruction Set Computer* - RISC); já outras trazem um conjunto mais amplo, com instruções de tamanho variado e que requerem vários ciclos de *clock* para serem executadas (abordagem *Complex Instruction Set Computer* - CISC). A arquitetura RISC, por conter instruções mais simples, pode ser considerada mais rápida do que a arquitetura CISC, por outro lado, os programas tendem a conter mais instruções. Atualmente, dispositivos móveis, tais como *smartphones* e *tablets*, incluem processadores RISC. Já os computadores pessoais, tais como *desktop* e *notebook*, possuem processadores CISC.

A CPU do CompSim trata exclusivamente de palavras de 16-bits. Isso significa que suas instruções e os tipos de dados suportados possuem larguras de 16-bits. Em cada instrução, foram reservados 4-bits para o campo de código de operação ("*Opcode*") e 12-bits para o campo Operando. Com opcodes de 4-bits, a CPU suporta 16 ($2^4$) instruções, que podem ser dos tipos de transferência de dados, aritméticas, lógicas, entrada/saída, transferência de controle e controle de sistema. Ressalta-se que, apesar do processador Cariri oferecer uma ISA restrita (poucas instruções), suas instruções possuem versatilidade suficiente para: incluir duas ou mais operações em uma única instrução; suportar diferentes modos de endereçamento e, com isso, ampliar os recursos de manipulação de dados pelo processador; e suportar a composição de combinações de instruções para implementar outras instruções e/ou operações mais complexas. As instruções da CPU podem suportar um tipo de operando numérico ou um tipo de dado não numérico. O tipo numérico consiste de números inteiros de 16-bits com sinal (*signed int*), codificados em notação "Complemento a 2". Já o tipo não numérico consiste de caracteres (*char*), codificados no padrão ASCII.

As subseções a seguir apresentam alguns dos aspectos mais abordados no estudo de arquitetura de computadores, ilustrando-os com aplicações no simulador CompSim.

### 3.4.1. A Estrutura de um Programa

Em cada célula de dados da memória RAM, é possível armazenar uma palavra, que consiste de uma sequência de bits ou de bytes. As células podem ser utilizadas para armazenamento de diferentes tipos de dados e de instruções, desde que estejam codificados em bits.

Para diferenciar dados e instruções, bem como reservar mais recursos na memória RAM, normalmente, os programas de computador, antes de serem alocados em memória, são divididos em segmentos, tais como: 1) Código ou de Texto (*Code*/*Text*): inclui todas as instruções do programa; 2) Dados (*Data*): inclui todas as variáveis e estruturas de dados globais ou estáticas, que foram definidas (inicializadas com valores na sua criação); 3) BSS (*Block Started by Symbol*): inclui todas as variáveis e estruturas de dados globais ou estáticas, que foram apenas declaradas (não foram

inicializadas); 4) *Heap*: é um segmento que contém espaços de memória não utilizados, os quais podem ser demandados em operações de alocação dinâmica de memória, tais como pelo uso das instruções “malloc” e “calloc” da linguagem de programação C; 5) Pilha (*Stack*): este segmento é utilizado para implementar a pilha do programa, que é basicamente é uma estrutura de dados do tipo LIFO (*Last-In*, *First-Out*) e pode ser utilizada para armazenamento temporário de dados, passagem de parâmetros de entrada e de retorno em funções, implementação de variáveis locais, entre outros.

A CPU do CompSim conta com 4 registradores para dar suporte à segmentação da memória de um programa. São eles: 1) Registrador de Segmento de Código (*Code Segment* - CS): armazena o endereço da primeira posição de memória alocada para o segmento de código; 2) Registrador de Segmento de Dados (*Data Segment* - DS): armazena o endereço da primeira posição de memória alocada para o segmento de dados; 3) Registrador de Segmento de Pilha (*Stack Segment* - SS): armazena o endereço da primeira posição de memória alocada para o segmento de pilha; e 4) Registrador Apontador de Topo de Pilha (*Stack Pointer* - SP): aponta para o topo da pilha do programa. Com suporte dos registradores de segmentos, o processador Cariri tem ciência dos limites de cada um dos segmentos.

O código a seguir ilustra a estrutura de um programa em linguagem de baixo nível (*Assembly*), no CompSim.

| Linha | Código |
|---|---|
| 1 | `.code` |
| ... | ... |
| ... | `.data` |
| ... | ... |
| ... | `.bss` |
| ... | ... |
| ... | `.stack 10` |

Na estrutura do código anterior, observa-se que ela inclui as palavras-chave “.code”, “.data”, “.bss” e “.stack”, que são delimitadores de seções em um código *Assembly* e indicam: o início da seção que conterá todas as instruções do programa; o início da seção que conterá a definição (declaração com inicialização) de variáveis e estruturas de dados globais e estáticas do programa; o início da seção que conterá a declaração (sem inicialização) das variáveis e estruturas de dados globais e estáticas do programa; e o tamanho da pilha, respectivamente. A palavra-chave “.stack” deve ser sucedida por um número inteiro positivo, o qual informa a quantidade de posições de memória que serão alocadas para a pilha do programa.

### 3.4.2. Alocação e Manipulação de Dados em Memória

No CompSim, a criação de variáveis e estruturas de dados é dada pelo uso de pseudo-instruções do montador (Assembler). Uma pseudo-instrução é um tipo de instrução que existe na linguagem *Assembly*, porém somente é reconhecida pelo montador (não faz parte do conjunto de instruções da CPU). Após o processo de montagem do código-fonte, durante o carregamento do programa em memória, o montador fica responsável

por alocar memória suficiente para comportar as estruturas de dados criadas com as respectivas pseudo-instruções no código-fonte.

As definições de variáveis dos tipos inteiro e caractere podem ser realizadas através das pseudo-instruções DD e DB, respectivamente. Já as definições de *arrays* dos tipos inteiro e caractere podem ser realizadas através das pseudo-instruções INITD e INITB, respectivamente. Neste capítulo, será utilizado o termo "*array*" para fazer referência à estrutura de dados matriz, quer seja uni ou multidimensional, visto que, do ponto de vista de alocação de memória, as linhas de uma matriz multidimensional são armazenadas sequencialmente em memória, assumindo assim uma disposição linear (vetorial). Um *array* de caracteres consiste, na realidade, em uma cadeia de caracteres (*string*), a qual, por sua vez, em termos de alocação em memória, apresenta-se como um *array* de números inteiros, visto que os caracteres da cadeia são codificados no padrão ASCII. É importante ressaltar que, na definição de variáveis e *arrays* do tipo inteiro, os valores de inicialização podem ser informados em bases decimal, hexadecimal e binária.

Uma variável, ou uma estrutura de dados, pode ser declarada e não possuir um valor de inicialização. As declarações devem ser realizadas exclusivamente na seção ".bss" do código *Assembly* do processador Cariri. Desta forma, o montador reconhece, na declaração, o tipo e a quantidade de elementos que a estrutura de dados necessita e, com isso, aloca espaços de memória suficientes para comportá-la. As declarações de variáveis e de *arrays* do tipo inteiro e caractere podem ser realizadas através da pseudo-instruções RESD e RESB, respectivamente.

No código a seguir, ilustra-se a criação de diferentes estruturas de dados.

| Linha | Código |
|---|---|
| 1 | `.code` |
| ... | `...` |
| 8 | `.data` |
| 9 | `...` |
| 10 | `   var1:   DD 10` |
| 11 | `   var2:   DB 'A'` |
| 12 | `   array1: INITD 10,11,12` |
| 13 | `   array2: INITB "ABC",0` |
| 14 | `.bss` |
| 15 | `   var3:   RESD 1` |
| 16 | `   array3: RESB 3` |
| 17 | `.stack 10` |

No código anterior, pode-se visualizar: a definição de uma variável do tipo inteiro inicializada com o valor 10 ("var1" na Linha 10) e uma do tipo caractere com valor 'A' ("var2" na Linha 11); a definição de um *array* inicializado com os números inteiros 10, 11 e 12 ("array1" na Linha 12) e de um *array* de caracteres inicializado com a *string* "ABC\0" ("array2" na Linha 13). Nas Linhas 15 e 16, são declarados, respectivamente, a variável "var3" do tipo inteiro (reservou-se 1 espaço de memória para ela) e o *array* de caracteres "array3", para o qual foi reservado espaço para armazenamento de 3 caracteres.

A manipulação de dados é realizada na CPU. A CPU do CompSim possui instruções para realizar operações de leitura e de escrita de dados em memória. Nessas instruções, o acesso aos dados alocados em memória pode ser realizado a partir de um nome de uma variável ou de um *array*, ou ainda por um endereço de memória. As instruções para leitura e gravação de um dado em memória são LDA e STA, respectivamente.

O programa a seguir ilustra como pode ser realizada a atribuição de um valor de uma variável "var1" à uma variável "var2". Uma atribuição de valor entre variáveis consiste em realizar a leitura do valor de uma variável, em uma determinada posição de memória, pela CPU, e escrevê-lo na posição de memória referente à outra variável. No código, na seção ".data", a variável "var1" é definida na Linha 11, sendo inicializada com o inteiro 10. Já a variável "var2" é declarada na seção ".bss", na Linha 14, por não ser inicializada.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   LDA var1          ;as duas instrucoes realizam: var2 = var1` |
| 3 | `   STA var2` |
| 4 | |
| 5 | `end:` |
| 6 | `   INT exit` |
| 7 | |
| 8 | `.data` |
| 9 | `   ;syscall exit` |
| 10 | `   exit: DD 25` |
| 11 | `   var1: DD 10       ;int var1 = 10` |
| 12 | |
| 13 | `.bss` |
| 14 | `   var2: RESD 1      ;int var2` |
| 15 | `.stack 10` |

As instruções de um programa devem ser inseridas na seção ".code". Assim, no código anterior, a Linha 2 inclui a instrução LDA que implementa a operação de leitura do inteiro 10, contido na variável "var1" em memória, para o registrador AC (Acumulador). Já na Linha 3, a instrução STA realiza a gravação do dado contido no registrador AC para a posição de memória referente à variável "var2".

Ainda no código, as Linhas 5 e 6 apresentam um rótulo "end:" e a respectiva instrução "INT exit" (as instruções em linguagem *Assembly* podem conter rótulos, também chamados de *labels*, e são muito importantes para dar suporte ao controle de fluxo de execução, modularização do programa, entre outros). A instrução INT, que recebe, como operando, uma variável ou um endereço de memória, possui múltiplas funções, dependendo do valor lido em memória, como será visto ao longo deste capítulo. Na Linha 6, a instrução INT realiza a leitura do valor da variável "exit", a qual contém o inteiro 25 (a variável "exit" é definida na Linha 10). Uma instrução INT com parâmetro inteiro 25, quando decodificada pela CPU, será compreendida como a instrução de controle de parada de simulação (*Halt*). Com isso, ao simular essa aplicação, a instrução "INT exit" instrui a CPU do CompSim a encerrar sua execução.

### 3.4.3. Processamento de Dados

No computador, o processamento de dados é realizado na CPU ou, mais precisamente, na ULA da CPU. A ULA é um circuito do tipo combinacional, que dependendo da sua entrada, logo apresenta uma saída. A ULA deve dar suporte aos tipos de processamento de dados demandados pelas instruções presentes na ISA da CPU. Dependendo da instrução de processamento de dados, a UC decodifica a instrução e configura a ULA para realizar tal operação.

Geralmente, nas CPUs, é possível encontrar diferentes tipos de instruções para processamento de dados, sendo as mais comuns as operações aritméticas, tais como adição, subtração, multiplicação e divisão, bem como as operações lógicas, tais como os operadores lógicos AND, OR, NOT e XOR e operações de deslocamento de bits. A CPU do CompSim conta com apenas 4 instruções para processamento de dados, sendo 2 delas para operações aritméticas (adição - ADD e subtração - SUB), 1 lógica (negação da conjunção - NAND) e 1 para deslocamento aritmético de bits (SHIFT).

O código a seguir ilustra como pode-se implementar a instrução “var1 = var1 + var2”, em linguagem Assembly do CompSim. As variáveis “var1” e “var2” estão definidas nas linhas 12 e 13, e são inicializadas com os inteiros 10 e 20, respectivamente. Na Linha 2, é realizada a leitura do valor contido na posição de memória referente à “var1” para AC (“AC = var1”); na Linha 3 realiza-se a soma dos inteiros em AC e no endereço de memória referente à “var2”, sendo o resultado guardado no próprio AC (“AC = AC + var2”); por fim, na Linha 4, o inteiro em AC é gravado na posição de memória referente à variável “var1” (“var1 = AC’).

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   LDA var1          ;este bloco realiza: var1 = var1 + var2` |
| 3 | `   ADD var2` |
| 4 | `   STA var1` |
| 5 | |
| 6 | `end:` |
| 7 | `   INT exit` |
| 8 | |
| 9 | `.data` |
| 10 | `   ;syscall exit` |
| 11 | `   exit: DD 25` |
| 12 | `   var1: DD 10       ;int var1 = 10` |
| 13 | `   var2: DD 20       ;int var2 = 20` |
| 14 | `.stack 10` |

No CompSim, as operações lógicas são implementadas a partir da instrução NAND. A operação lógica NAND é bastante versátil, pois, com ela, é possível compor outras operações e funções lógicas, utilizando algumas propriedades da álgebra booleana, por isso ela é conhecida como “porta lógica universal” [Patrick, Fardo and Chandra, 2020]. Para compreensão de como funções lógicas podem ser compostas a partir de operações lógicas universais, é importante ter conhecimentos básicos de

álgebra booleana. Por exemplo, a composição da operação lógica AND a partir de operações NAND, se dá da seguinte maneira:

$$A.B = \neg ( (\neg (A.B) ) . (\neg (A.B) ) ) \quad (1)$$

, onde “A e “B” são variáveis lógicas, “¬” representa o operador lógico de negação; e “.” representa o operador lógico de conjunção. Partido da equação (1), o código a seguir ilustra como pode ser implementada uma operação lógica de conjunção (AND) entre as variáveis “var1” e “var2”.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   LDA var1          ;este bloco realiza: tmp = NAND(var1, var2)` |
| 3 | `   NAND var2` |
| 4 | `   STA tmp` |
| 5 | |
| 6 | `   NAND tmp          ;realiza: tmp = NAND(AC, tmp)` |
| 7 | |
| 8 | `end:` |
| 9 | `   INT exit` |
| 10 | |
| 11 | `.data` |
| 12 | `   ;syscall exit` |
| 13 | `   exit: DD 25` |
| 14 | `   var1: DD 1111111111111111b      ;int var1 = 0xffff` |
| 15 | `   var2: DD 0000000011111111b      ;int var1 = 0x00ff` |
| 16 | `   tmp:  DD 0` |
| 17 | `.stack 10` |

As variáveis “var1” e “var2”, que estão sendo definidas nas Linhas 14 e 15, são inicializadas com os inteiros em base binária “1111111111111111b” e “0000000011111111b” (65.535 e 255, respectivamente). Nas Linhas 2 à 4, realiza-se a operação lógica NAND entre os bits das variáveis “var1” e “var2”, e o resultado é armazenado na variável “tmp”, tendo assim, portanto: “tmp = ¬(var1 . var2)”. O próximo passo, considerando a equação (1), será realizar uma nova operação NAND com os bits em “tmp”. Assim, a Linha 6 realiza a operação lógica NAND entre os bits em AC e na variável “tmp” (neste caso, os bits em AC estão idênticos aos de “tmp”), tendo assim “AC = ¬(AC . tmp)”. O resultado desta operação, guardado em AC, contém o valor lógico equivalente ao da operação AND entre os bits das variáveis “var1 e “var2”.

A instrução lógica de deslocamento de bits SHIFT da CPU do CompSim realiza deslocamento aritmético de 1-bit em palavras de 16-bits, nos sentidos à direita ou à esquerda. Essa operação tem várias utilidades, sendo uma delas a implementação da operação de multiplicação por 2 (deslocamento à esquerda) ou divisão por 2 (deslocamento à direita). A combinação da instrução SHIFT com a de adição ADD pode ser utilizada para implementar outras multiplicações tal como, por exemplo, por 5.

Em tese, uma multiplicação de um inteiro 10 por 5, poderia ser implementada com adições sucessivas (10+10+10+10+10). Porém, com o uso de deslocamento de bits, 10 poderia ser multiplicado por 4 (através de dois deslocamentos sucessivos à esquerda), seguido de uma uma soma por 10, como mostra o código a seguir.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   LDA var1          ;este bloco realiza: (var1*2)*2 + var1` |
| 3 | `   SHIFT esquerda` |
| 4 | `   SHIFT esquerda` |
| 5 | `   ADD var1` |
| 6 | |
| 7 | `end:` |
| 8 | `   INT exit` |
| 9 | |
| 10 | `.data` |
| 11 | `   ;syscall exit` |
| 12 | `   exit:      DD 25` |
| 13 | `   var1:      DD 10` |
| 14 | `   esquerda:  DD 1` |
| 15 | `.stack 10` |

No código anterior, são definidas as variáveis “var1” (na Linha 13), que é inicializada com o inteiro 10, e a variável “esquerda” (na Linha 14), que é inicializada com o inteiro 1 (esta variável será utilizada pela instrução SHIFT e o inteiro 1 indica que será realizado um deslocamento de 1-bit à esquerda). Na Linha 2, o valor de “var1” é carregado em AC. As instruções SHIFT, nas Linhas 3 e 4, atuam da seguinte maneira: ao ler o valor da variável “esquerda”, verifica-se que consiste de um deslocamento de 1-bit à esquerda no valor contido em AC. Assim, com as instruções das Linhas 3 e 4, o valor em AC sofrerá 2 deslocamentos de 1-bit à esquerda, o que equivale à multiplicação por 4 (“AC = AC*2*2”). Por fim, a instrução na Linha 5, adiciona o valor de “var1” ao contido em AC, compondo a multiplicação por 5. Para realizar o deslocamento de 1-bit à direita, a instrução SHIFT, ao ler o conteúdo de uma variável, necessita do valor 0.

É importante destacar que a aplicação das operações lógicas e aritméticas pode resultar, além de um valor positivo, também pode resultar em valores zero, negativo e que supera o limite de 16-bits de uma palavra (*overflow*). Nesses casos, a CPU notifica-os por meio de um registrador de estados (*Status Flags Register* - SFR), que possui 1-bit para sinalizar resultado igual a zero (Z), 1-bit para resultado negativo (S) e 1-bit para *overflow* (O).

### 3.4.4. Controle de Fluxo de Execução

Um programa de computador consiste em um sequência de instruções, as quais são lidas e executadas respeitando uma ordem de execução. Um controle de fluxo de execução, em termos gerais, pode consistir em, dada uma determinada condição, alterar a ordem em que as instruções de um programa são executadas. Controle de fluxo pode ser importante em várias situações, tais como: dada uma condição, executar uma de duas

possíveis ações; repetir uma sequência de ações até que uma condição não mais seja atendida; repetir determinadas ações por uma quantidade predefinida de vezes; entre outras.

A CPU do CompSim possui três instruções de controle de fluxo, sendo uma delas de salto incondicional (JMP) e duas de salto condicional (JZ e JN). A instrução JMP (*Jump*), ao ser executada, transfere seu operando, que consiste de um endereço de memória, para o registrador contador de programa (PC), de forma que, no próximo ciclo de instrução da CPU, o programa passará a executar a instrução apontada no novo endereço em PC. Já as instruções de salto condicional, antes de atualizarem o endereço em PC, avaliam se determinados bits no registrador de status SFR estão configurados em 1. No caso, a instrução JZ (*Jump if Zero*) avalia o bit Z do registrador SFR, enquanto que a instrução JN (*Jump if Negative*) avalia o bit S de SFR. Por exemplo, se estiver sendo executada uma instrução JZ e se o bit Z de SFR estiver configurado em 1, a instrução fará com que o endereço de memória contido em seu operando seja copiado para PC. Dessa forma, no próximo ciclo de instrução da CPU, será executada a instrução apontada pelo novo endereço em PC. Após um salto condicional, as instruções JZ e JN configuram em 0 os bits Z e S de SFR, respectivamente. Com o uso das instruções JMP, JZ e JN é possível implementar construções de linguagens de alto nível para controle de fluxo, tais como IF..ELSE, FOR..DO e WHILE..DO.

O código a seguir ilustra a implementação de uma estrutura IF..ELSE, na qual verifica-se se o valor da variável “var1” é superior ao da variável “var2”. Dependendo do resultado da comparação, será atribuído o conteúdo da variável de maior valor à de menor valor.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `if:` |
| 3 | `   LDA var1          ;este bloco realiza: if (var1 >= var2)` |
| 4 | `   SUB var2` |
| 5 | `   JN else` |
| 6 | `then:                ;este bloco realiza: then` |
| 7 | `   LDA var1             ;var2 = var1` |
| 8 | `   STA var2` |
| 9 | `else:                ;este bloco realiza: else` |
| 10 | `   LDA var2             ;var1 = var2` |
| 11 | `   STA var1` |
| 12 | |
| 13 | `end:` |
| 14 | `   INT exit` |
| 15 | |
| 16 | `.data` |
| 17 | `   ;syscall exit` |
| 18 | `   exit: DD 25` |
| 19 | `   var1: DD 10` |
| 20 | `   var2: DD 20` |
| 21 | `.stack 10` |

No código, para a comparação, as Linhas 3 e 4 realizam a subtração entre os valores contidos nas variáveis “var1” e “var2”. Como o valor da variável “var1” é inferior ao da variável “var2” (ver suas definições nas Linhas 19 e 20), a CPU configurará em 1 o bit S no registrador SFR. Seguindo o fluxo de execução, na Linha 5, a instrução JN verificará que o bit S de SFR está configurado em 1, então, ela atribuirá o endereço da instrução rotulada como “else” ao registrador PC, provocando assim, um desvio do fluxo para essa instrução, no próximo ciclo de execução da CPU.

### 3.4.5. Acesso Avançado à Memória

Viu-se, nas subseções anteriores, que a CPU possui recursos que permitem realizar o acesso e o processamento de dados armazenados em memória. Dependendo do conjunto de dados e do modelo de processamento da aplicação, pode ser necessário utilizar mecanismos adicionais para ampliar as formas de acesso ao conjunto de dados e aos seus elementos individualmente, para, desta forma, reduzir a complexidade e otimizar a aplicação.

Um desses recursos é o ponteiro. Um ponteiro, ou apontador, pode ser definido como uma variável que armazena números inteiros sem sinal com dimensão suficiente para acesso a todos os endereços de memória do espaço de endereçamento da CPU [Santos and Langlois 2018]. Os ponteiros podem ser utilizados em diversas situações, tais como: referenciar variáveis, provendo um caminho alternativo para acesso aos dados das variáveis; acesso aos elementos de estruturas de dados com armazenamento contínuo em memória, como *arrays*, e não alocadas em endereços contíguos de memória, tais como árvores e grafos; passagem e retorno de parâmetros por referência em funções; entre outras.

A CPU do Compsim possui instruções que permitem definir um ponteiro (DD), ler (LDI) e gravar (STI) o conteúdo apontado na memória, bem como atribuir o endereço de uma variável a um ponteiro (MOV). O código a seguir ilustra uma aplicação que utiliza cada uma dessas funções.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   LDI ptr          ;este bloco realiza: *ptr = *ptr + var2` |
| 3 | `   ADD var2` |
| 4 | `   STI ptr` |
| 5 | |
| 6 | `   MOV var2         ;este bloco realiza: ptr = &var2` |
| 7 | `   STA ptr` |
| 8 | |
| 9 | `end:` |
| 10 | `   INT exit` |
| 11 | |
| 12 | `.data` |
| 13 | `   ;syscall exit` |
| 14 | `   exit: DD 25` |
| 15 | `   var1: DD 10` |
| 16 | `   var2: DD 20` |
| 17 | `   ptr:  DD var1    ;int * ptr = &var1` |
| 18 | `.stack 10` |

No código, são definidas as variáveis “var1” e “var2”, nas Linhas 15 e 16. Na Linha 17, define-se um ponteiro, chamado “ptr”, que é do tipo inteiro, e é inicializado com o endereço de memória da variável “var1” (equivalente à instrução em linguagem C “int * ptr = &var1”). Na Linha 2, o conteúdo apontado por “ptr” é lido para AC, através da instrução LDI (“AC = *ptr”). Na Linha 3, é acrescido ao conteúdo lido o valor da variável “var2” (“AC = AC + var2”). Na Linha 4, o resultado da operação de adição anterior é gravado na posição de memória apontada por “ptr”, através da instrução STI (“*ptr = AC”). Por fim, nas Linhas 6 e 7, o endereço de memória da variável “var2” é copiado para AC (“AC = &var2”) e, em seguida, é gravado na variável “ptr” (“ptr = AC”), fazendo com que a mesma passe a apontar para a posição de memória referente à “var2”.

Com o suporte das instruções de alocação e de manipulação de dados em memória, controle de fluxo de execução e de acesso avançado à memória, é possível criar programas mais elaborados, tais como para manipulação de estruturas de dados mais complexas, como matrizes, registros, listas encadeadas, árvores e grafos.

### 3.4.6. Entrada/Saída

A interação com o computador se dá através dos periféricos. Viu-se que há diferentes tipos de periféricos, onde cada tipo possui suas próprias características; e que a comunicação entre a CPU e os periféricos se dá pelo barramento de periféricos e pelo subsistema de E/S.

O subsistema de E/S do CompSim permite a conexão automatizada de novos periféricos ao barramento de periféricos, bastando que o periférico implemente a interface de comunicação padrão com o barramento. Quanto aos periféricos, no CompSim, é possível ter periféricos do tipo virtual, que são aqueles implementados em software e emulam o comportamento de periféricos reais, e  físico, que são divididos em software, para implementar a interface padrão com o barramento de periféricos, e em hardware para compor o periférico físico em si e seu comportamento. Por padrão, o CompSim não é distribuído com periféricos. Porém, no website do projeto [CompSim 2021], na seção “*Download*” é possível realizar o *download* para instalação dos periféricos virtuais Video, Keyboard e VArduino (Arduino Virtual) e do periférico físico Arduino UNO, tanto para o sistema operacional MS/Windows quanto GNU/Linux. A instalação de um periférico é bastante simples: consiste apenas em descompactar o arquivo referente ao periférico, obtido no website do CompSim, na pasta onde se encontra o arquivo executável do simulador CompSim.

A CPU do CompSim possui a instrução INT, a qual, até este ponto do capítulo, somente havia sido utilizada com o parâmetro 25 para encerrar a execução do programa (*Halt*). Porém, a instrução INT também pode ser utilizada em operações de E/S com periféricos. Para tanto, a instrução necessita, como parâmetros, dos códigos de operação 20 e 21 para leitura e escrita de dados em um determinado periférico, respectivamente. As operações de E/S, com a instrução INT, devem atender às seguintes condições: 1) Em operações de leitura/escrita, os 8-bits mais significativos do registrador acumulador AC serão utilizados para endereçamento de periféricos. Com isso, é possível ter até 256

endereços de periféricos (“portas”) diferentes ($2^8$ = 256); 2) Em operações de saída, os 8-bits menos significativos do registrador acumulador AC serão utilizados para guardar o byte que será escrito em algum periférico; e 3) Em operações de entrada, após a leitura de dados de periférico, o dado lido estará disponível no registrador acumulador AC, nos 8-bits menos significativos, caso o valor lido seja um byte, ou, nos 16-bits de AC, caso o valor lido seja um número inteiro. Desta forma, nas operações de E/S, o registrador acumulador AC do processador Cariri passa a incluir os campos “AC High”, utilizado para endereçamento de porta de periférico, e “AC Low”, utilizado para transferência de dados, como podem ser vistos na Figura 3.2.


**Figura 3.2. Campos do registrador AC em operações de E/S.**

Considerando que o periférico Video está devidamente instalado, o código a seguir ilustra os passos para impressão do caractere “A”.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   LDA char          ;AC Low recebe 'A'` |
| 3 | `   ADD video_port    ;AC High recebe a porta de Video` |
| 4 | `   INT output        ;realiza operacao de output` |
| 5 | |
| 6 | `end:` |
| 7 | `   INT exit` |
| 8 | |
| 9 | `.data` |
| 10 | `   ;syscall exit` |
| 11 | `   exit:       DD 25` |
| 12 | `   char:       DB 'A'       ;caractere que será impresso em Video` |
| 13 | `   output:     DD 21        ;codigo de output da instrucao INT` |
| 14 | `   video_port: DD 0x0000    ;porta do periferico Video` |
| 15 | `.stack 10` |

No código anterior, na Linha 12 define-se a variável “char”, que é do tipo caractere e é inicializada com o caractere “A”. Na Linha 13, é definida a variável “output”, que inclui o código de operação 21, o qual refere-se à operação da instrução INT para escrita de dados em periférico. Na Linha 14, a variável “video_port” inclui a porta do periférico Video, que por padrão tem o endereço de E/S igual a 0.

Ainda no código anterior, na Linha 2, o caractere “A” é carregado nos 8-bits menos significativos do registrador acumulador AC (campo “AC Low”). Em seguida, adiciona-se ao campo “AC High” de AC a porta do periférico Video (na Linha 3), e, por fim, na Linha 4, realiza-se a operação de escrita de dados no periférico Video, através da instrução INT com parâmetro 21 (contido na variável “output”). A Figura 3.3

apresenta, à direita, a visualização do periférico Video após a operação de escrita, onde observa-se que houve de fato a impressão do caractere “A”.

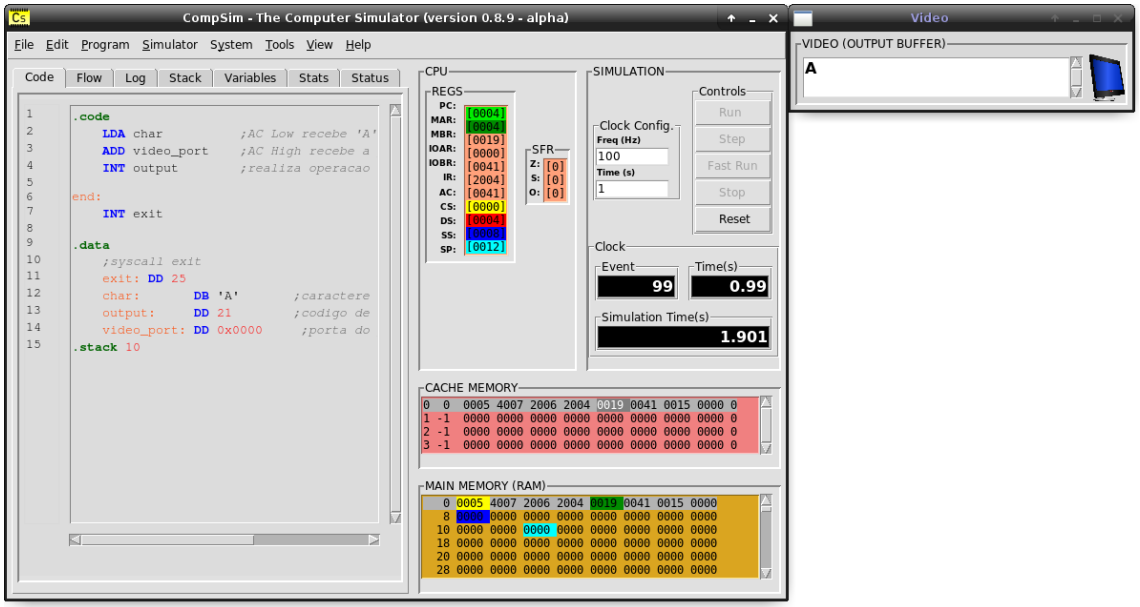



**Figura 3.3. Visualização de escrita de caractere no periférico Video.**

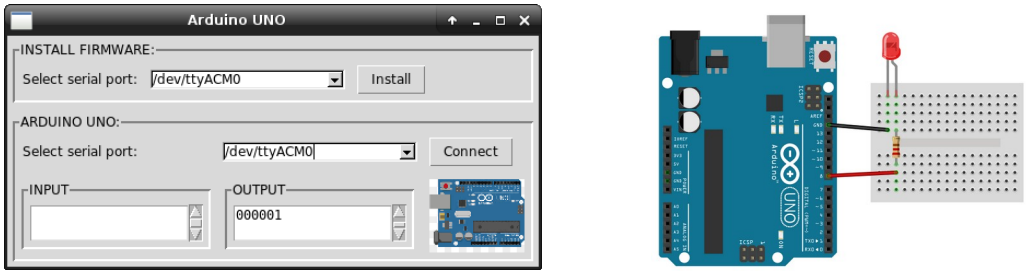
O periférico físico Arduino UNO se destaca por permitir a interação da PHV do CompSim com componentes eletrônicos reais. Arduino é uma plataforma eletrônica baseada em microcontrolador, com especificação aberta e tem sido largamente utilizada em projetos de sistemas eletrônicos [Arduino 2018].

Arduino UNO é a mais utilizada dentre as diversas versões do Arduino. Seu microcontrolador gerencia a interação com componentes eletrônicos externos através dos registradores PortB, PortC e PortD. Os bits desses registradores estão diretamente conectados aos pinos de E/S (*General Purpose Input/Output* - GPIO) em uma placa de circuitos (*board*) do Arduino UNO. Assim, através de operações de leitura e/ou escrita de dados nesses registradores é possível interagir com componentes eletrônicos conectados a uma *board* Arduino UNO, através do pinos de GPIO.

No CompSim, o periférico físico Arduino UNO, cuja interface gráfica pode ser vista na Figura 3.4(a), por padrão, mapeia os bits dos registradores PortB, PortC e PortD nas portas de 2 a 10. Com este modelo, a CPU, ao realizar operações de E/S nas portas mapeadas nos registradores de um Arduino UNO, pode ativar operações de leituras e/ou escritas digitais ou analógicas nos GPIOs.

Para ilustrar a interação entre o simulador CompSim e componentes eletrônicos reais, propõe-se o cenário exposto na Figura 3.4(b). Nele há: uma *board* Arduino UNO; uma *protoboard*, utilizada para conectar os componentes eletrônicos *led* vermelho a um resistor de 200 Ohm; e *jumpers* (fios) que conectam o circuito eletrônico montado na *protoboard* ao Arduino UNO. O *jumper* vermelho liga o GPIO 8 da *board* Arduino UNO ao resistor, que por sua vez está ligado ao polo positivo do *led*, e o *jumper* preto

liga o polo negativo do *led* ao pino de isolamento (*ground* - GND) da *board* Arduino UNO. O circuito funciona da seguinte maneira: ao realizar uma operação de escrita no registrador PortB do Arduino UNO, dependendo do valor escrito, pode-se ativar (ou desativar) uma corrente elétrica no GPIO 8, que percorrerá o circuito resistor-*led* até o GND, fazendo com que o *led* acenda.




**(a) O periférico físico Arduino UNO.** **(b) Circuito eletrônico para "pisca-led".**

**Figura 3.4. O periférico Arduino UNO e o circuito "pisca-led".**

Para a execução do cenário proposto, é necessário que: o periférico Arduino UNO esteja instalado no CompSim; o circuito apresentado na Figura 3.4(b) esteja montado e configurado; e a *board* Arduino UNO esteja conectada a uma porta USB do computador.

O passo seguinte será instalar o *firmware* no microcontrolador do Arduino UNO para que seja possível realizar a comunicação com o CompSim. Para tanto, na Figura 3.4(a), no painel "*INSTALL FIRMWARE*", deve-se escolher a porta serial na qual a *board* Arduino UNO está conectada e clicar no botão "*Install*". Em seguida, após a instalação, deve-se estabelecer uma conexão lógica entre o CompSim e a *board* Arduino UNO, selecionando novamente a porta serial e clicando no botão "*Connect*", ambos no painel "ARDUINO UNO", como mostra a Figura 3.4(a). O último passo é programar e executar uma aplicação em *Assembly* no CompSim que realize operações de escrita de dados na porta 2, a qual está mapeada no registrador PortB do microcontrolador do Arduino UNO, cujos bits estão ligados aos GPIOs 8 a 13 de uma *board*.

O código a seguir ilustra uma aplicação para ligar e desligar o led ("pisca-led") conectado na GPIO 8. No código, nas Linhas 14 e 15 estão definidos, nas variável "liga" e "desliga", os bits que serão escritos na PortB do Arduino UNO, para acender e apagar o led, respectivamente. Para a escrita dos bits em PortB, a variável "portB_arduino", definida na Linha 17, guarda a porta do periférico Arduino UNO referente ao registrador PortB. Nas Linhas 3 à 5, carrega-se os bits na variável "liga" em AC Low, acrescenta-se a porta contida na variável "portB_arduino" em AC High e realiza-se a operação de escrita para acender o *led*, respectivamente. Já as instruções nas Linhas 7 à 9 fazem com que o led desligue (os procedimentos são análogos aos de ligar o *led*, com exceção de que carrega-se em AC Low os bits da variável "desliga"). Por último, a instrução na Linha 11 realiza um desvio de fluxo de execução para o início do

programa, criando assim um loop infinito no programa e, por consequência, a aplicação ligará e desligará o *led* indefinidamente ou até que se encerre a simulação.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `loop:` |
| 3 | `   LDA liga              ;AC Low recebe os bits para ligar o led` |
| 4 | `   ADD portB_arduino     ;AC High recebe a porta de PortB` |
| 5 | `   INT output            ;realiza operacao de output` |
| 6 | |
| 7 | `   LDA desliga           ;AC Low recebe os bits para desligar o led` |
| 8 | `   ADD portB_arduino     ;AC High recebe a porta de PortB` |
| 9 | `   INT output            ;realiza operacao de output` |
| 10 | |
| 11 | `   JMP loop` |
| 12 | |
| 13 | `.data` |
| 14 | `   liga:          DD 00001b   ;5-bits para escrita em PortB` |
| 15 | `   desliga:       DD 00000b   ;5-bits para escrita em PortB` |
| 16 | `   output:        DD 21       ;codigo de output da instrucao INT` |
| 17 | `   portB_arduino: DD 0x0200   ;porta de PorB do Arduino UNO` |
| 18 | `.stack 10` |

Outro periférico que se destaca é o VArduino, por apresentar um periférico virtual do Arduino UNO com disponibilidade de componentes eletrônicos simuláveis, tais como *leds*, botões, sensores, *display* de 7-segmentos, entre outros. Ele é bastante útil quando não há disponibilidade dos respectivos componentes eletrônicos físicos e apresenta o mesmo modelo de comunicação de E/S do periférico Arduino UNO (os programas escritos em *Assembly* são compatíveis entre os periféricos Arduino UNO e VArduino) [Cartaxo et al. 2020].

Finalmente, ressalta-se que o Subsistema de E/S do CompSim é bastante versátil, pois, além dos periféricos disponíveis no website do projeto, permite conectar dinamicamente, à PHV do CompSim, outros tipos desenvolvidos pelos usuários. Para tanto, na criação dos novos periféricos, para que tenham o suporte de conexão automática, deve-se implementar um modelo de interface padrão, descrito com maiores detalhes em [Esmeraldo et al. 2020].

### 3.4.7. Modularização de Programas

Dependendo da aplicação, os programas de computador podem se tornar maiores, por incluir mais instruções, e, por consequência, se tornarem muito difíceis de ler e compreender, realizar depuração para correções, manutenção e evolução. Ao longo dos anos, viu-se que a melhor forma de desenvolver e manter um programa maior é modularizá-lo, através da sua divisão em partes menores (também chamadas de "módulos"), que são mais simples de tratar em relação ao programa como um todo. Há vários motivos para modularizar um programa, entre eles estão: 1) Dividir um problema maior em problemas menores, que são mais simples de tratar, e agrupar suas soluções para compor a solução final do problema maior; 2) Evitar repetição de código, onde ao se criar funções, que são blocos de instruções que realizam tarefas específicas, elas

podem ser chamadas para execução sob demanda, em diferentes pontos do programa; e 3) Abstrair o código, pois ao chamar uma função para execução, não é necessário compreender como a mesma está implementada, apenas utiliza-se a sua assinatura, como são os casos da funções “printf”, “scanf” e “malloc”, da linguagem de programação C.

A CPU do CompSim inclui duas instruções que implementam, respectivamente, os passos para chamada (CALL) e retorno (RET) de uma função. No CompSim, para a definição de uma função basta definir um bloco de instruções, em que a primeira delas terá um rótulo, que será o identificador da função, e a última será a instrução RET. Com isso, pode-se utilizar a instrução CALL com o identificador da função para chamá-la para execução (o endereço da instrução de retorno é adicionado na pilha do programa e o fluxo de execução do programa é desviado para a primeira instrução do bloco de instruções da função). Ao final da execução da função, a instrução RET faz com que o fluxo de execução seja novamente desviado, retornando à instrução posterior à da chamadora da função, cujo endereço foi previamente guardado na pilha do programa.

O código a seguir ilustra a criação de uma função que realiza a impressão do caractere “A” no periférico Video e como ela pode ser chamada para execução.

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `   CALL printA` |
| 3 | `   CALL printA` |
| 4 | `   CALL printA` |
| 5 | |
| 6 | `end:` |
| 7 | `   INT exit` |
| 8 | |
| 9 | `printA:` |
| 10 | `   LDA char          ;AC Low recebe 'A'` |
| 11 | `   ADD video_port    ;AC High recebe a porta de Video` |
| 12 | `   INT output        ;realiza operacao de output` |
| 13 | `   RET` |
| 14 | |
| 15 | `.data` |
| 16 | `   ;syscall exit` |
| 17 | `   exit:       DD 25` |
| 18 | `   char:       DB 'A'       ;caractere que sera impresso em Video` |
| 19 | `   output:     DD 21        ;codigo de output da instrucao INT` |
| 20 | `   video_port: DD 0x0000    ;porta do periferico Video` |
| 21 | `.stack 10` |

No código anterior, nas Linhas 9 à 13, encontra-se a definição da função chamada “printA”, em que: na Linha 9, observa-se o rótulo “printA” da instrução na Linha 10 (o rótulo consiste do identificador da função); o bloco de instruções nas Linhas 10 à 12 realiza a impressão do caractere “A” no periférico Video; e, na Linha 13, a instrução RET encerra a execução da função, ao realizar o desvio de fluxo de execução para retorno. Observa-se, nas Linhas 1 à 3, que a função “printA” é chamada 3 vezes para execução, por meio da instrução “CALL printA”. Ao executar esse

programa, espera-se que sejam impressos 3 caracteres “A” no periférico Video, como mostra a Figura 3.5.

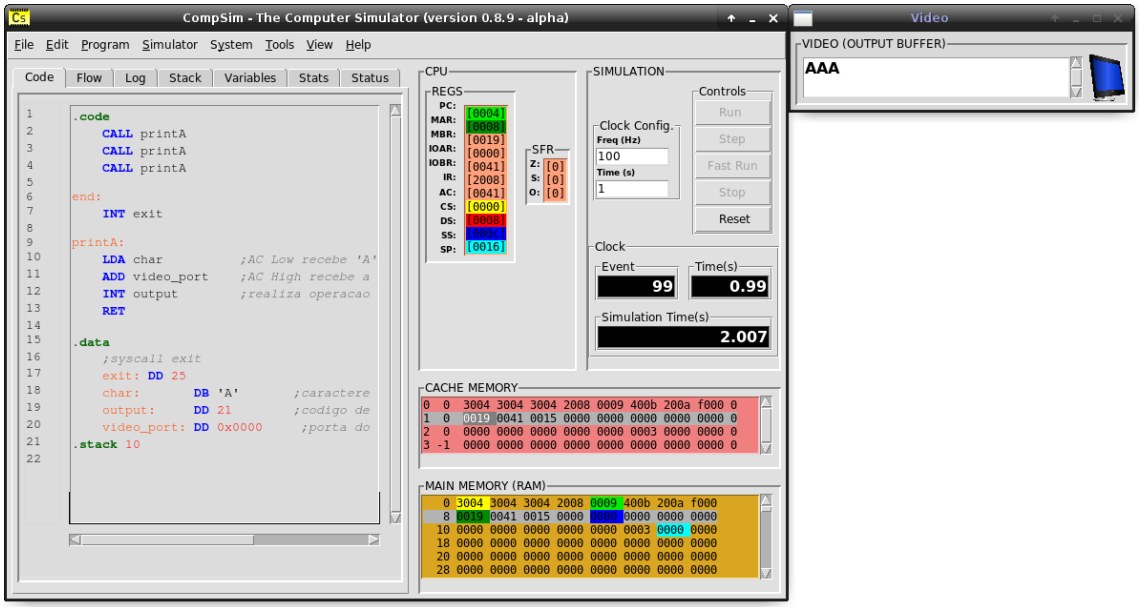



**Figura 3.5. Visualização de escrita de caracteres no periférico Video por chamadas de função.**

O código a seguir ilustra a modularização do código para ligar e desligar o *led* conectado na GPIO 8 (programa “pisca-*led*”).

| Linha | Código |
|---|---|
| 1 | `.code` |
| 2 | `loop:` |
| 3 | `    CALL ligaLed` |
| 4 | `    CALL desligaLed` |
| 5 | `    JMP loop` |
| 6 | |
| 7 | `ligaLed:` |
| 8 | `    LDA liga              ;AC Low recebe os bits para ligar o led` |
| 9 | `    ADD portB_arduino     ;AC High recebe a porta de PortB` |
| 10 | `    INT output            ;realiza operacao de output` |
| 11 | `    RET` |
| 12 | `desligaLed:` |
| 13 | `    LDA desliga           ;AC Low recebe os bits para desligar o led` |
| 14 | `    ADD portB_arduino     ;AC High recebe a porta de PortB` |
| 15 | `    INT output            ;realiza operacao de output` |
| 16 | `    RET` |
| 17 | |
| 18 | `.data` |
| 19 | `    liga:          DD 00001b   ;5-bits para escrita em PortB` |
| 20 | `    desliga:       DD 00000b   ;5-bits para escrita em PortB` |
| 21 | `    output:        DD 21       ;codigo de output da instrucao INT` |
| 22 | `    portB_arduino: DD 0x0200   ;porta de PorB do Arduino UNO` |
| 23 | `.stack 10` |

Nas Linhas 7 à 11, está definida a função “ligaLed”, cujas instruções fazem com que seja ativada a corrente elétrica que liga o led, e, nas Linhas 12 à 16, define-se a função “desligaLed”, responsável por desativar a corrente elétrica. As Linhas 2 à 5 implementam um *loop* infinito, em que a instrução da Linha 3 chama a função “ligaLed” e a instrução da Linha 4 chama a função “desligaLed”. Dessa forma, a aplicação executará com a chamada iterativa das funções para ligar e desligar o *led* indefinidamente ou até que se encerre a simulação.

Com os recursos do simulador CompSim, além da definição e chamada de funções para execução, é possível explorar outros aspectos, tais como: passagem de parâmetros por valor e por referência; passagem de parâmetros em variáveis globais, registradores e pilha do programa; retorno de função; implementação de variáveis locais, através de variáveis globais, alocação de memória e pilha do programa; definição e execução de funções aninhadas e de funções recursivas.

## 3.5. Conclusões

Este capítulo apresentou uma proposta de abordagem de uso do simulador CompSim para apoio ao aprendizado prático de conceitos de Arquitetura e Organização de Computadores. O simulador abordado inclui uma plataforma de hardware virtual que contém os principais componentes do computador, tais como CPU, memórias, barramentos e periféricos, cujas características são semelhantes às dos respectivos componentes reais. Além disso, o simulador conta com uma interface gráfica, que inclui recursos integrados para simplificar a configuração, programação, simulação, visualização e análise de desempenho de sistemas computacionais criados no ambiente virtual.

Ao longo do capítulo, foram apresentados alguns dos recursos do simulador CompSim e como eles podem ser utilizados para estimular a aplicação prática dos conceitos aprendidos em aulas teóricas e, com isso, otimizar o processo de ensino-aprendizagem em projetos de sistemas computacionais. Observa-se que os elementos apresentados neste capítulo não esgotam o leque de recursos e potencialidades de aprendizagem pelo uso do simulador CompSim. Desta forma, recomenda-se explorar o website do projeto [CompSim 2021], onde estão disponíveis, além do simulador, materiais didáticos variados, relação de publicações científicas, link para grupo de discussão sobre o projeto, entre outros. Sugere-se ainda a leitura dos trabalhos: [Lisboa et al. 2018], que detalha a interface de comunicação entre o CompSim-Arduino UNO; [Esmeraldo et al. 2018], que traz descrições de mais recursos gráficos do simulador; [Cartaxo et al. 2020], para aprofundamento teórico na dinâmica de simulação de sistemas computacionais com o periférico VArduno; e [Esmeraldo et al. 2020], para compreender a interface do subsistema de E/S e como pode-se criar novos periféricos virtuais e físicos para a PHV do CompSim.

Cabe destacar que o CompSim tem sido utilizado por diferentes turmas de cursos de Técnico Integrado em Eletrônica e de Bacharelado em Sistemas de Informação e os resultados vêm mostrando que, ao utilizar o simulador CompSim em

aulas práticas, o aprendizado em projetos de sistemas computacionais se torna mais atrativo, dinâmico, produtivo e efetivo.

Por fim, ressalta-se que o simulador está em contínuo desenvolvimento e que o feedback dos estudantes e o apoio de diversas instituições parceiras têm sido muito importantes para adição de novos recursos, bem como para o aprimoramento do simulador e da experiência de uso, bem como do aprendizado em projetos de sistemas computacionais.

## Referências


[ACM and IEEE 2013] ACM, Association for Computing Machinery and IEEE Computer Society (2013) "Curriculum Guidelines for Undergraduate Degree Programs in Computer Science", The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society.

[Arduino 2018] Arduino (2018) "What is Arduino?" https://www.arduino.cc/en/Guide/Introduction

[Bahk et al. 2013] Bahk, J. H.; Youngs, M.; Yazawa, K.; Shakouri, A. and Pantchenko, O. (2013) "An online simulator for thermoelectric cooling and power generation", In: Frontiers in education Conference, IEEE, 2013.

[Balamuralithara and Woods 2009] Balamuralithara, B. and Woods, P. C. (2009) "Virtual laboratories in engineering education: The simulation lab and remote lab", In: Computer Applications in Engineering Education, Vol. 17(1). pp. 108–118.

[Cartaxo et al. 2020] Cartaxo, L. F., Mendes, C. S. R., Lisboa, E. B. and Esmeraldo, G. A. R. M. (2020) "Utilizando VArduino para Criação de Periféricos Virtuais baseados em Arduino UNO para o Simulador CompSim", In: Revista Tecnologias na Educação, v. 33, pp. 1-17.

[CompSim 2021] CompSim (2021) "CompSim - The Computer Simulator". http://compsim.crato.ifce.edu.br/

[Esmeraldo et al. 2018] Esmeraldo, G., Cartaxo, L. F., Mendes, C. S. R. and Lisboa, E. B. (2018) "Um Simulador Educacional para Apoio ao Projeto de Sistemas Computacionais: Hardware, Software e suas Interfaces", In: XXVI Workshop sobre Educação em Computação (WEI 2018) - XXXVIII Congresso da Sociedade Brasileira de Computação (CSBC 2018).

[Esmeraldo et al. 2019] Esmeraldo, G. A. R. M., Mendes, C. S. R., Cartaxo, L. F. and Lisboa, E. B. (2019) "Apoio ao Aprendizado em Arquitetura e Organização de Computadores: Um Estudo Comparativo entre Simuladores Computacionais", In: Revista Tecnologias na Educação, v. 31, pp. 1-17.

[Esmeraldo et al. 2020] Esmeraldo, G. A. R. M., Lisboa, E. B., Mendes, C. S. R., Cartaxo, L. F., Ribeiro, C. V., Morato, L. F. B., Santos, P. S. and Nascimento, M. S. (2020) "Uma Abordagem Integrada de Hardware e Software para o Aprendizado de Subsistemas de Entrada/Saída em Projetos de Sistemas Computacionais", In: International Journal of Computer Architecture Education, v. 9, pp. 1-9.

[Garcia, Pacheco and Garcia 2014] Garcia, I. A.; Pacheco, C. L. and Garcia, J. N. (2014) “Enhancing education in electronic sciences using virtual laboratories developed with effective practices”, In: Computer Applications in Engineering Education, Vol. 22(2), pp. 283–296.

[Keutzer et al. 2000] Keutzer, K., Newton, A. R., Rabaey, J. M. and Sangiovanni-Vincentelli, A. (2000) "System-level design: orthogonalization of concerns and platform-based design", In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19(12), pp. 1523-1543.

[Lisboa et al. 2018] Lisboa, E. B., Cartaxo, L. F., Mendes, C. S. R. and Esmeraldo, G. A. R. M. (2018) “Ambiente Integrado de Hardware e Software Aplicado ao Ensino de Projeto de Sistemas Computacionais”, In: III Congresso sobre Tecnologias na Educação (Ctrl+E 2018).

[Lisboa et al. 2019] Lisboa, E. B., Cartaxo, L. F., Mendes, C. S. R. and Esmeraldo, G. A. R. M. (2019) “Uma Metodologia Educacional para Aprendizado Prático de Organização e Arquitetura de Computadores com Apoio de Simulador Computacional”, In: Brazilian Journal of Development, v. 5, pp. 31062-31068.

[Monteiro 2007] Monteiro, M. A. (2007) “Introdução à Organização de Computadores”. 4a. Ed. LTC.

[Nikolic et al. 2009] Nikolic, B.; Radivojevic, Z.; Djordjevic, J. and Milutinovic, V. A. (2009) “Survey and Evaluation of Simulators Suitable for Teaching Courses in Computer Architecture and Organization”, In: IEEE Transactions on Education, Vol. 52, No. 4.

[Null and Lobur 2009] Null, L. and Lobur, J. (2009) “Princípios Básicos de Arquitetura e Organização de Computadores”. Ed. Bookman.

[Patrick, Fardo and Chandra, 2020] Patrick, D. R., Fardo, S. W. and Chandra, V. (2020) “Electronic Digital System Fundamentals”. 1st Ed. River Publishers.

[SBC 2005] SBC. Sociedade Brasileira de Computação (2005) “Currículo de Referência da SBC para Cursos de Graduação em Bacharelado em Ciência da Computação e Engenharia de Computação”. https://www.sbc.org.br/documentos-da-sbc/category/131-curriculos-de-referencia

[Santos and Langlois 2018] Santos, P. R. and Langlois, T. (2018) “Compiladores: da Teoria à Prática”. LTC.

[Stallings 2010] Stallings, W. (2010) “Computer Organization and Architecture. Designing for Performance”. 8th Ed. Prentice Hall.

[Tanenbaum and Austin 2013] Tanenbaum, A. S. and Austin, T. (2013) “Organização estruturada de computadores”, 6a. Ed. Pearson.

[Uribe et al. 2016] Uribe, M. D. R.; Magana, A. J.; Bahk, J.-H. and Shakouri, A. (2016) “Computational Simulations as Virtual Laboratories for Online Engineering Education: A Case Study in the Field of Thermoelectricity”, In: Computer Applications in Engineering Education, Vol. 24(3). pp. 428–442.

[Wolffe et al. 2002] Wolffe, G. S.; Yurcik, W.; Osborne, H.; Holliday and M. A. (2002) "Teaching computer organization/architecture with limited resources using simulators", In: Proceedings of the 33rd SIGCSE technical symposium on Computer science education, ACM SIGCSE Bulletin. Vol. 34, No. 1.