CENTRO UNIVERSITÁRIO FEEVALE

ANDRÉ WAGNER

CRIAÇÃO DE UM KIT DE DESENVOLVIMENTO DE SOFTWARE PARA PROGRAMAÇÃO MODULAR DE EMULADORES

NOVO HAMBURGO, 2006

ANDRÉ WAGNER

CRIAÇÃO DE UM KIT DE DESENVOLVIMENTO DE SOFTWARE PARA PROGRAMAÇÃO MODULAR DE EMULADORES

Centro Universitário Feevale
Instituto de Ciências Exatas e Tecnológicas
Curso de Ciência da Computação
Trabalho de Conclusão de Curso

Professor orientador: Delfim Luiz Torok

Novo Hamburgo, novembro de 2006

RESUMO

Os emuladores podem ser usados para diversos fins, desde o desenvolvimento de software até a utilização de sistemas legados, e a recente explosão na utilização de dispositivos portáteis e celulares trouxe um aumento na demanda deste tipo de programa. No entanto, o desenvolvimento de um emulador ainda é bastante complexo, devido à falta de bibliografia e de ferramentas auxiliares nesta área. Este projeto visa suprir esta necessidade, através do desenvolvimento e implementação de uma biblioteca e de ferramentas auxiliares que facilitem a programação de novos emuladores de forma modular. Esta primeira parte do trabalho desenvolve um referencial teórico, descrevendo os diferentes tipos de emulador e suas técnicas de programação, o funcionamento do hardware de computador e a forma como a emulação deste hardware é tradicionalmente implementado em um emulador, bem como os problemas e dificuldades que o programador provavelmente enfrentará neste desenvolvimento. A conclusão introduz a segunda parte do trabalho, onde serão apresentadas uma nova biblioteca e ferramentas de desenvolvimento, que se aliarão a uma nova metodologia tornando a programação de emuladores mais fácil, rápida e reutilizável.

Palavras-chave: emulador, arquitetura de hardware, biblioteca.

ABSTRACT

Emulators can be used in variety of ways, from software development to the use of legacy systems, and the recent increase in the use of portable devices and cellphones brought a growth in the demand of this kind of program. The development of a new emulator, however, is still very complex, due to the lack of bibliography and auxiliary tools in this area. This project aims to fill this need through the development and implementation of a library and auxiliary tools that will ease the modular programming of new emulators. The first part of this paper presents a bibliographic reference, describing the different kinds of emulators and their programming techniques, the way a hardware architecture works and how this architecture is emulated, and also the problems and hardships that the programmer will find during the development. The conclusion introduces the second part of this paper, where there will be presented a new library a auxiliary tools that, together with a new methodology, will make the emulator development faster, easier e reusable.

Keywords: emulator, hardware architecture, library.

LISTA DE ABREVIATURAS E SIGLAS

ALU	*Aritmethic Logic Unit*
BIOS	*Basic Input/Output System*
CISC	*Complex Instruction Set Computer*
CPU	*Central Processing Unit*
EPROM	*Eraseble Programmable Read-Only Memory*
GPU	*Graphics Processing Unit*
HSYNC	*Horizontal Synchronization*
INT	*Interruption*
IP	*Instruction Pointer*
JVM	*Java Virtual Machine*
PC	*Program Counter*
PIC	*Progrmmable Interrupt Controller*
PROM	*Programmable Read-Only Memory*
RAM	*Random Access Memory*
RGB	*Red Green Blue*
RISC	*Reduced Instruction Set Computer*
SDL	*Simple DirectMedia Layer*
SP	*Stack Pointer*
TIA	*Television Interface Adapter*
UAL	Unidade Aritmética Lógica
UCP	Unidade Central de Processamento
VBLANK	*Vertical Blank*
VDP	*Video Display Processor*
VPU	*Video Processing Unit*
VSYNC	*Vertical Synchronization*
WINE	*Wine Is Not a Emulator*

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

INTRODUÇÃO

Um emulador é um programa de computador que imita o comportamento de uma arquitetura computacional[1], de modo que um software escrito para uma plataforma possa ser executado em outra (BRITISH, 2002). Os emuladores encontram diversos usos, dentre os quais se destacam o desenvolvimento para plataformas portáteis, o desenvolvimento de sistemas operacionais e o uso de aplicações legadas.

Os emuladores têm como objetivo principal a execução de programas de uma arquitetura computacional em outra, dando ao usuário a impressão de estar usando o computador original. Como um emulador é um programa que usa intensos recursos de processamento, geralmente é necessário deixar parte da precisão de lado para obter um desempenho equivalente ao do computador original. Isto diferencia os emuladores dos simuladores (que imitam uma plataforma com precisão absoluta em detrimento do desempenho) e dos virtualizadores (que permitem apenas a emulação da própria arquitetura em que estão sendo executados).

O desenvolvimento de um emulador é feito de forma modular, desenvolvendo uma série de módulos que emulam cada um dos dispositivos do sistema. Como existe uma infinidade de dispositivos que podem ser emulados, este trabalho se concentra nos quatro mais importantes, encontrados em todos os computadores: (i) processador; (ii) memória; (iii) sistema de vídeo e (iv) sistema de entrada.

Na implementação de um emulador, existe uma série de dificuldades e problemas que o programador enfrentará, devido à falta de uma metodologia de desenvolvimento de emuladores. Este trabalho busca suprir esta dificuldade, através do projeto, desenvolvimento e implementação de um Kit de Desenvolvimento de Software[2] que dê suporte à criação de novos emu-

1 O conjunto de tipos de dados, operações e características de um computador. Alguns autores incluem o sistema operacional junto na arquitetura (fazendo de um PC com Windows uma arquitetura diferente de um PC com Linux). Neste trabalho, a definição refere-se exclusivamente à arquitetura de hardware de um computador.

2 Pacote formado por um conjunto de ferramentas que auxiliam desenvolvedores de software na implementação de novos aplicativos. Geralmente é conhecido por SDK (do inglês *Software Development Kit*).

ladores, permitindo e facilitando a programação de emuladores de forma modular e reaproveitável, centralizando as operaçõespadrão e oferecendo uma interface gráfica ao usuário.

Para tanto, o primeiro passo é uma pesquisa de embasamento teórico envolvendo a modelagem tradicional de emuladores, visando compreender sua forma de funcionamento e identificar as atividades padrão – isto é, as atividades que se repetem na criação da maior parte dos emuladores.

O segundo passo é a realização de um estudo, buscando alistar as principais dificuldades e problemas existentes no modelo tradicional de desenvolvimento de emuladores. Neste estudo, procura-se também agrupar os componentes por tipo, buscando identificar as atividades padrão de cada tipo, de modo a oferecer ao programador uma biblioteca que permita o desenvolvimento modular.

Baseado nestes dois estudos, este trabalho é apresentado. O Primeiro Capítulo faz uma introdução ao assunto, localizando-o dentro da informática, diferenciando-o de assuntos semelhantes e especificando exatamente quais os tipos e técnicas que este trabalho enfocará

O Segundo Capítulo apresenta o embasamento teórico a respeito da arquitetura de hardware, fundamental para o desenvolvimento de um emulador e levando em conta apenas o que é importante para a criação destes. O capítulo enfoca os quatro dispositivos escolhidos: processadores, memória, sistema de vídeo e sistema de entrada, dividindo-os em tipos e falando das peculiaridades de cada um.

O Terceiro Capítulo apresenta as técnicas tradicionais de construção de emuladores, ou seja, a forma como um emulador é construído. Este capítulo apresenta a mesma divisão do segundo capítulo mas, ao invés de explicar o funcionamento dos dispositivos, apresenta a sua forma de implementação dentro de um emulador.

O Quarto Capítulo apresenta os problemas e dificuldades que a maioria dos programadores terá que enfrentar na construção de um novo emulador. A conclusão deste trabalho introduz a segunda parte do trabalho, a ser desenvolvida, discutindo os planos para a resolução dos problemas apresentados no último capítulo.

1 CONCEITOS SOBRE EMULADORES

Este Capítulo objetiva introduzir o assunto de emuladores, definindo o que são, quais são os seus tipos básicos e a sua filosofia de funcionamento, bem como situando os objetivos propostos por este trabalho dentro do universo dos tipos e das técnicas de emulação.

1.1 Emuladores

Um emulador é um programa de computador que imita o comportamento de uma arquitetura computacional, de modo que um software escrito para uma plataforma possa ser executado em outra. A Sociedade Britânica de Computação define a emulação como "[...] uma forma precisa de simulação que imita exatamente o comportamento ou as circunstâncias que se estão simulando. Um emulador permite que um tipo de computador opere como se fosse um tipo diferente de computador." (tradução nossa) (BRITISH, 2002, p. 30-31).

Um emulador assemelha-se bastante a uma máquina virtual, como a Máquina Virtual Java (JVM, do inglês *Java Virtual Machine*). Neste caso, o código é escrito na linguagem Java pelo programador, e compilado para um *bytecode*[3]. Este *bytecode* – semelhante a um código binário executável – é então interpretado por uma máquina virtual, como o JVM. Segundo Lindholm (1999, p. 12), "a Máquina Virtual Java é um computador abstrato. Como um computador real, possui um conjunto de instruções e manipula várias áreas de memória em tempo de execução" (tradução nossa).

A diferença essencial entre um emulador e uma máquina virtual está no fato da máquina virtual interpretar código escrito para uma máquina abstrata, enquanto um emulador procura interpretar código escrito para uma máquina real, geralmente diferente daquela na qual o emulador está sendo executado. No caso máquina virtual, o código foi escrito especificamente para ela, enquanto no caso do emulador, o código não foi escrito para o emulador, mas para outra ar-

3 Conjunto de instruções em forma binária, interpretáveis através de uma máquina virtual. Do inglês "código em bytes".

quitetura computacional.

Um exemplo de emulador é o fMSX. O fMSX é um software que emula o funcionamento de um computador MSX[4] em uma série de sistemas operacionais diferentes, como Windows, MS-DOS, MacOS, Linux, etc. Desta forma, é possível usar um computador PC[5] com Windows (ou MS-DOS, ou outros sistemas operacionais) e, ainda assim, executar qualquer programa escrito para o MSX (FAYZULLIN, 2006).

A performance de um emulador é muito importante, pois é necessário que o usuário tenha a impressão de que está usando uma máquina real. Para isso, o emulador precisa manter a sincronização, de modo a não ser executado nem muito rápida nem muito lentamente (DEL BARRIO, 2001, p. 11).

1.2 Níveis de Abstração

Abstração é o processo de reduzir a quantidade de informação necessária para um determinado conceito, com o objetivo de eliminar os detalhes irrelevantes e reter somente a informação relevante para um determinado propósito. Esta idéia é usada para gerenciar a complexidade de sistemas computacionais. SegundoKeller (1997, p. 1),

> estes sistemas geralmente consistem de milhões de pequenos componentes (bytes de memória, comandos de programa, portas lógicas, etc.). Tratar todos os componentes como um simples monolito é quase intelectualmente impossível. Portanto é comum, ao invés disso, ver o sistema como sendo composto por alguns poucos componentes interativos, cada qual compreendido em termos de *seus* componentes, e assim por diante, até que o nível mais básico seja atingido. (tradução nossa) (grifo do autor)

Tanenbaum (1990) define os computadores como tendo, em geral, seis níveis de abstração, partindo do nível mais baixo (hardware) e indo para o nível mais alto (aplicações), conforme apresentado na Figura 1.

4 Computador de arquitetura aberta criado por uma parceria entre Microsoft e ASCII, de modo a oferecer uma arquitetura padrão. Usava um microprocessador Zilog Z80. Foi muito popular no Brasil nos anos 80, onde era fabricado pela Gradiente (com o nome de Expert) e pela Sharp (com o nome de Hotbit).

5 Computador pessoal padrão IBM usando microprocessador da série x86. Do inglês *Personal Computer.*

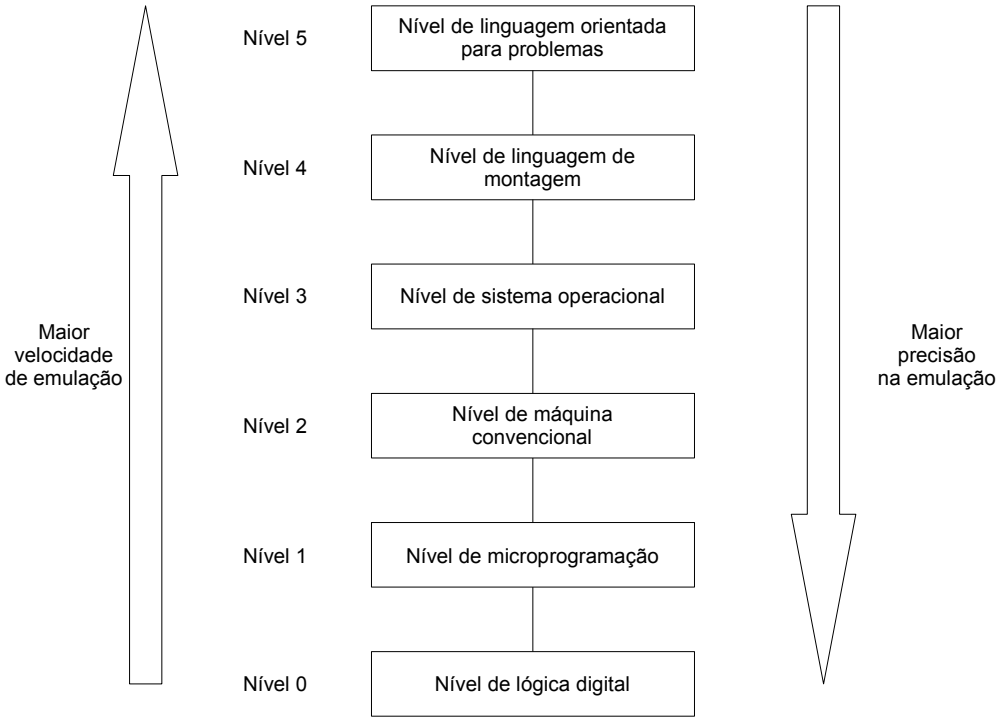

Figura 1 - Os níveis de abstração definidos por Tanenbaum.
Adaptado de TANENBAUM, 1990.

O nível mais baixo é o **nível de lógica digital**, que é o verdadeiro hardware da máquina. Este nível inclui especialmente as portas lógicas, que são dispositivos digitais que executam operações simples de lógica booleana[6], como as operações E e OU (TANENBAUM, 1990).

Os dois níveis seguintes são o **nível de microprogramação** e o **nível de máquina convencional**. Embora ambos pareçam bastante semelhantes, o nível de máquina convencional é aquele na qual as instruções *assembly*[7] são executadas. Portanto, quando o fabricante de um determinado processador distribui um manual de referência de linguagem *assembly*, este manual refere-se ao nível 2, e não ao nível 1. O nível 1, de microprogramação, refere-se a como estas instruções são implementadas dentro do processador (TANENBAUM, 1990).

O nível seguinte é o **nível de sistema operacional**. Em um sistema operacional, a maior parte das instruções executadas são idênticas às do nível de máquina convencional, porém

6 Sistema lógico baseado em valores digitais (VERDADEIRO e FALSO), criada por George Bool no século XIX, e atualmente muito usada na área da eletrônica e da computação.

7 Linguagem de montagem, onde as instruções numéricas enviadas para o processador (linguagem de máquina) são representadas por abreviaturas, de modo a facilitar o uso por um programador. Exemplos de instruções *assembly* comuns são MOV (mover), ADD (somar) e INT (gerar interrupção).

um novo conjunto de funcionalidades é acrescentado pelo sistema operacional. Segundo Tanenbaum (1990, p. 5), "algumas instruções de nível 3 são interpretadas pelo sistema operacional e outras são interpretadas diretamente pelo microprograma". As novas funcionalidades acrescentadas pelo sistema operacional são executadas através de chamadas de sistema[8].

O **nível de linguagem de montagem** e o **nível de linguagem orientada para problemas** são bastante diferentes dos outros níveis. Eles são voltados para programadores de aplicação que têm problemas específicos a serem solucionados. Estes níveis definem, respectivamente, os dois passos pela qual uma aplicação em código-fonte passa até se transformar em um arquivo binário executável: a montagem e a compilação. Um código-fonte[9] em uma linguagem de alto nível (como C, BASIC ou Pascal) é **compilado** para um código-fonte *assembly*, e o código-fonte *assembly* é **montado** em um arquivo binário executável (TANENBAUM, 1990).

1.3 Emuladores e Níveis de Abstração

Existem diferentes tipos de emuladores que simulam outras arquiteturas, e que variam de acordo com o nível de abstração que emulam. Quanto mais alto o nível de abstração a ser emulado, maior será a velocidade, e menor a precisão. O oposto também é verdadeiro - quanto mais exata a emulação, menor será a performance.

Simuladores são programas que buscam imitar o funcionamento de uma arquitetura com precisão absoluta – ou seja, emulam o nível 0. Geralmente são usados por engenheiros com o objetivo de testar novos tipos de hardware ou software, ou então para compreender como um sistema age internamente (um exemplo seria compreender como um determinado sistema operacional está usando a memória *cache*[10]). Devido à sua precisão, um sistema simulado geralmente não consegue atingir uma velocidade equivalente à do hardware original (DEL BARRIO, 2001, p. 12).

Um **emulador** funciona de forma semelhante a um simulador, mas busca obter uma velocidade compatível à do hardware original. Para isto, um emulador muitas vezes precisa deixar um pouco da precisão de lado para obter um desempenho superior (este fator será discutido com mais profundidade na Seção 3.4). Segundo Del Barrio (2001, p. 16),

8 Mecanismo utilizado por um aplicativo para requisitar um serviço do sistema operacional, geralmente através de uma interrupção de software.
9 Código em forma legível, usando uma linguagem de programação, que é traduzido para linguagem de máquina por um compilador, ou executado diretamente por um interpretador.
10 Memória de acesso muito rápido, onde uma cópia dos dados mais utilizados da memória RAM são armazenados.

> Outro aspecto da emulação que deve ser levado em conta é o nível de precisão que o emulador necessita. O tipo de emulador ao qual nos referimos não necessita ser preciso em um nível muito baixo (por exemplo, no funcionamento interno do microprocessador, nas funções de *cache* ou no tráfego de barramento) porque a intenção não é analisar a performance ou simular o computador. Queremos emular o comportamento externo dos *videogames* (ou aplicativos) executados no emulador o mais próximo possível do computador real. A intenção é que o emulador soe e se pareça o máximo possível com o sistema real. Isto significa que, às vezes, a precisão precisa ser sacrificada pelo desempenho, porque uma das coisas mais importantes a serem emuladas em um emulador é a "sensação de tempo", ou seja, o emulador precisa ser executado exatamente à mesma velocidade do sistema real. (tradução nossa)

Os emuladores estão no nível de abstração de máquina convencional definido por Tanenbaum, conforme ilustra a Figura 1, na página 14.

Um **emulador de alto nível** é um tipo especial, pois é construído com o objetivo de obter um desempenho superior. Nele, as chamadas de sistema dos aplicativos são capturadas e, ao invés de serem interpretadas, são executadas por instruções que estão dentro do próprio emulador. Isto significa que um emulador de alto nível abrange também o nível de sistema operacional. A desvantagem deste método é que ele é pouco preciso, e para cada software da plataforma que está sendo emulada, é necessário testar e adaptar o emulador. Um exemplo deste tipo de emulador é o UltraHLE, que emula um videogame do tipo Nintendo 64[11] (ULTRAHLE, 2006).

Um **virtualizador** é um tipo de emulador que implementa apenas a própria arquitetura onde está sendo executado. Ele faz isso emulando apenas os componentes e executando o código binário diretamente no processador, atingindo assim uma velocidade quase nativa. É usado, por exemplo, para executar um sistema operacional diferente do que se está sendo executado, mas que rode na mesma arquitetura (por exemplo, Windows e Linux). Um exemplo de virtualizador é o VMWare (VMWARE, 2006).

O virtualizador, assim como o emulador, abrange o nível de máquina convencional, sendo que a diferença entre os dois está no fato de que o emulador interpretar as instruções do processador, enquanto o virtualizador as executa diretamente.

Por último, existem ainda os **simuladores de sistema operacional**. Neste caso, eles agem como virtualizadores, mas ao invés de emular os componentes de hardware, eles capturam as chamadas de sistema e as implementam em uma plataforma diferente. Exemplos deste tipo de software são o *Cooperative Linux* (que permite usar programas de Linux dentro do Windows) e o WINE[12] (que permite usar programas de Windows no Linux) (ALONI, 2004).

---

11 Videogame lançado pela Nintendo em 1996, foi um dos primeiros videogames com gráficos 3D.

12 "*Wine* não é um Emulador" - programa que permite executar programas de Windows no Linux, emulando as chamadas de sistema à API do Windows (do inglês *Wine Is Not a Emulator*)

Levando em conta todos estes tipos de programas simuladores, a construção de emuladores que se enquadram no nível de máquina convencional (segundo definido por Tanenbaum) será o foco principal deste trabalho.

1.4 Filosofia de Funcionamento

Segundo Von Neumann, os computadores são formados por um conjunto de, no mínimo, cinco componentes: (i) uma unidade de memória (hoje geralmente conhecida por memória RAM[13]); (ii) uma unidade de entrada (um teclado, por exemplo); (iii) uma unidade de saída (um monitor de vídeo, por exemplo); (iv) uma unidade de controle e (v) uma unidade de lógica aritmética (os dois últimos hoje unificados em um só microprocessador), todos conectados através de um barramento (MURDOCCA, 2000).

Assim como um computador, um emulador funciona de forma modular, com os módulos do emulador tendo função semelhante àqueles do computador. Desta forma, existe um módulo que emula um microprocessador, um módulo que emula a memória, um módulo que emula os gráficos, e assim por diante. Estes módulos são unificados através de um laço[14], que de forma simplificada para uma máquina monoprocessada poderia ser o da Figura 2.

A emulação de um **microprocessador** é diferente da emulação dos outros tipos de dispositivos. O microprocessador busca dados da memória e executa as instruções equivalentes àqueles dados, retornando à memória o resultado da execução, ou então realizando alguma outra operação (por exemplo, comunicando-se com algum componente do sistema). A execução do microprocessador será interrompida com uma certa freqüência para tratar eventos de outros componentes, num processo chamado interrupção. (DEL BARRIO, 2001). A emulação de microprocessadores será discutida mais profundamente na Seção 3.2.

É necessário que haja uma sincronia entre o microprocessador e os outros componentes. Esta sincronia é atingida através de um sinal de ***clock***, que é um sinal digital usado para coordenar as ações de todos os circuitos (WIKIPEDIA, 2006a). No emulador, diferentemente da máquina real, todos os componentes são guiados pelo microprocessador – ele é o responsável por produzir a informação de quantos ciclos de *clock* foram gerados com a última instrução, e esta informação é passada aos outros componentes no momento de sua emulação (BORIS, 1999).

13 Memória de Acesso Randômico, significando uma memória que pode ser acessada em qualquer posição, ao contrário dos outros tipos de memória usados antigamente, como fitas (do inglês *Random Access Memory*)

14 Um conjunto de instruções de um programa que é definido apenas uma vez, mas é executado repetidamente. Também conhecido pela palavra inglesa *loop*.

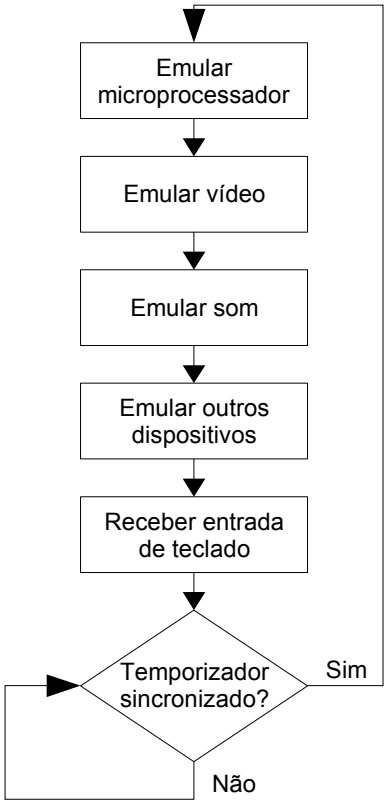

Figura 2 - Laço de um emulador simples.

Outros componentes internos do computador também devem ser emulados. Geralmente, eles se comunicam com o microprocessador e uns com os outros através de **áreas de memória** (onde cada componente tem uma área de memória reservada para si), de portas e de interrupções. Computadores modernos também usam um sistema de Acesso Direto à Memória (DMA, do inglês *Direct Memory Access*), segundo Murdocca (2000, p. 288), "um dispositivo com acesso direto à memória pode transferir dados diretamente da memória para para a mesma em vez de usar a CPU[15] como intermediário; portanto, pode diminuir a congestão do barramento do sistema."

Dos componentes internos do computador a serem emulados, geralmente o mais complexo e que utiliza mais de recursos de emulação é o **sistema gráfico**. De acordo com Del Barrio (2001, p. 107), "a emulação gráfica [...] ocupará mais de 50% do tempo da emulação, às vezes chegando até 80% ou 90%" (tradução nossa). Assim, na emulação do sistema gráfico – e, algumas vezes, de outros componentes – se faz necessário sacrificar a exatidão da emulação em prol do desempenho.

15 Termo comumente usado para referir-se ao microprocessador. Do inglês *Central Processing Unit*, ou Unidade de Processamento Central.

O **sistema de entrada**, geralmente baseado em teclado e mouse (no caso de micro-computadores) ou em *joystick*[16] (no caso de videogames), é normalmente a parte mais simples da emulação. Neste caso, quando uma ação for realizada na plataforma onde o emulador está sendo executado (por exemplo, o pressionamento de uma tecla), a ação correspondente será realizada na plataforma emulada (por exemplo, será escrito um byte na memória ou uma interrupção será gerada).

1.5 Técnicas de Emulação

Segundo Laureano (2006), os emuladores podem ser baseados em hardware, software ou alguma combinação entre os dois. Emuladores baseados em hardware são aqueles que são totalmente implementados em um componente físico, como um *microchip*. Emuladores baseados em software são aqueles que são executados em um computador, totalmente separados da implementação de hardware e onde o software provê todos os recursos para a emulação. Existem ainda as soluções híbridas, onde parte da emulação é feita em hardware, e parte em software.

Os emuladores baseados em software, que são o foco principal deste trabalho, podem ser implementados usando quatro técnicas diferentes.

Usando a técnica de i**nterpretação,** cada instrução é interpretada por um módulo que emula o microprocessador. O emulador do microprocessador lê a próxima instrução da memória, busca seu código de execução em uma tabela de instruções e executa-a, realizando assim a função equivalente. Esta técnica tem a vantagem de ser o tipo de emulação mais completo – qualquer plataforma pode ser emulada em qualquer plataforma, e até mesmo um emulador pode ser executado dentro de outro emulador. Este é, entretanto, a técnica de implementação cuja a execução é, geralmente, a mais lenta (DEL BARRIO, 2001).

No caso da **tradução binária** estática, um emulador usando esta técnica funciona de forma semelhante a um compilador. Nele, um aplicativo escrito para uma arquitetura é compilado para código binário executável em outra arquitetura. Desta forma, um programa executável de uma arquitetura se transforma num programa executável de outra arquitetura. Este é o tipo mais rápido de emulação, porém tem sérias limitações, ele não consegue lidar com código auto-modificável, o que acontece em todos os computadores, quando um aplicativo é carregado de uma unidade de armazenamento (como um disquete) para a memória. Portanto, serve apenas

16 Dispositivo de entrada usado em videogames, geralmente composto por uma manopla para o acionamento direcional e alguns botões.

para algumas plataformas específicas que não usam sistema operacional nem carregam programas para a memória RAM (como os videogames mais antigos) (DEL BARRIO, 2001).

Usando-se a técnica da **tradução binária dinâmica,** partes do código são compiladas (e recompiladas) durante a execução, sob demanda, de modo que o código gerado na memória venha a refletir as modificações que aconteceram durante a execução do emulador. Esta técnica é a mesma utilizada por máquinas virtuais do tipo *just-in-time* (JIT) (LAUREANO, 2006). Este tipo de emulador é mais rápido que o interpretador, mas mais lento que o tradutor binário estático, e seu grande problema é a extrema complexidade de implementação, devido à necessidade de técnicas heurísticas[17] para detecção de blocos de código, detecção de memória modificada, compilação e otimização (DEL BARRIO, 2001).

A **virtualização** é a técnica mais rápida de todas, mas pode apenas ser usada em ambientes onde a arquitetura em que o emulador será executado e a arquitetura que será emulada são a mesma. Neste caso, apenas os componentes são emulados, mas o código é executado nativamente pelo microprocessador. Esta técnica é muito utilizada em computadores PC, de modo a permitir que dois sistemas operacionais sejam executados na mesma máquina, ou então para manter vários serviços rodando isoladamente em um mesmo computador (LAUREANO, 2006).

Este trabalho tratará dos emuladores que usam a técnica de interpretação, por ser a única das técnicas que permite uma boa modularização e o reaproveitamento dos componentes implementados, bem como a unificação das atividades padrão.

1.6 Benefícios do Uso de Emuladores

Os emuladores podem ser usados para diferentes fins:

- Facilitar o desenvolvimento em plataformas onde a programação é difícil ou impossível. Um exemplo é o desenvolvimento de jogos de videogame onde, em muitos casos, o desenvolvimento é feito em um computador e testado em um emulador. Outro exemplo são os dispositivos móveis (computadores portáteis e celulares), onde a pequena tela dificultaria a programação, mas com o uso de um emulador o programador não só tem a facilidade de poder testar seu desenvolvimento em uma tela maior, como também a agilidade de não precisar fazer um *upload* para o dispositivo a cada nova compilação;

17 Métodos ou algoritmos exploratórios para definição de problemas em que as soluções são descobertas pela avaliação do progresso obtido na busca de um resultado final.

- Permitir o uso de aplicativos legados (por exemplos, editores de texto ou jogos antigos), que eram utilizados em equipamentos que não estão mais disponíveis, ou aos quais o usuário não têm acesso;

- Facilitar o desenvolvimento de novos sistemas operacionais, permitindo que desenvolvedor tenha acesso a tudo que acontece dentro de cada dispositivo, e dando a ele a facilidade de testar seu sistema com diferentes configurações;

- Facilitar o desenvolvimento de programas para múltiplas plataformas, oferecendo ao desenvolvedor uma gama de arquiteturas nas quais ele pode testar seus programas sem a necessidade de copiá-los a outros computadores.

Além dos benefícios citados, Laureano (2006, p. 35) acrescenta ainda que os emuladores perimitem "Testar configurações e situações diferentes do mundo real, como, por exemplo, mais memória disponível ou a presença de outros dispositivos de E/S" e "auxiliar no ensino prático de sistemas operacionais e programação ao permitir a execução de vários sistemas para comparação no mesmo equipamento".

1.7 Considerações Finais

O aumento do número de arquiteturas disponíveis no mercado trouxe um aumento na demanda por novos tipos de emuladores. Existe uma vasta gama de técnicas de emulação, mas a implementação de qualquer uma delas é tarefa custosa, exigindo meses de desenvolvimento e teste, e a implementação de complexos algoritmos de modo a obter um desempenho equivalente à arquitetura original.

Devido à extensa quantidade de tipos e técnicas de emulação, este trabalho se baseará em emuladores do nível de máquina convencional (conforme discutido na Seção 1.2), e que usem a interpretação como método de emulação, por ser o tipo que se encaixa de forma mais precisa com objetivos propostos neste trabalho.

2 ARQUITETURA DE HARDWARE PERTINENTE À CONSTRUÇÃO DE EMULADORES

Este capítulo discute a arquitetura interna de um computador. Visto que existe abundante literatura a respeito deste assunto, nesta Seção serão tratados somente os assuntos pertinentes ao segundo nível de abstração, que é o nível convencional de máquina, segundo discutido na Seção 1.2. Serão discutidos primeiramente cada um dos principais dispositivos que compõem um computador e, mais adiante, a forma como eles funcionam em conjunto dentro de uma arquitetura computacional.

As técnicas para a construção de emuladores dos dispositivos descritos neste Capítulo serão discutidas no Capítulo 3.

2.1 O Processador

O processador (também conhecido como microprocessador, CPU ou UPC[18]) é o "cérebro" do computador. Tem como propósito executar programas armazenados na memória, buscando nela as instruções e executando-as, uma após a outra (TANENBAUM, 1990). Um processador comum é capaz de executar um grande número de instruções em um curto período de tempo: um Pentium 4[19], por exemplo, é capaz de executar 1 bilhão e 500 milhões de instruções por segundo. Já o 80386[20], bem mais modesto, opera a 5 milhões de instruções por segundo (GILHEANY, 2006).

Um processador é composto de:

- uma **unidade de controle**, responsável pela busca de instruções da memória principal e determinação de seus tipos;
- uma **unidade lógica aritmética** (ALU, do inglês *Arithmetic Logic Unit*), res-

18 Unidade Central de Processamento
19 Sétima geração de processadores do padrão x86 construída pela Intel, fabricado a partir de 2000.
20 Terceira geração de processadores do padrão x86 da Intel, fabricado a partir de 1986.

ponsável pela execução das operações aritméticas e lógicas com os dados;

- uma pequena **memória interna** de alta velocidade, composta por registradores, cada um com uma função definida (TANENBAUM, 1990).

2.1.1 Conjunto de Instruções de Máquina

Segundo Murdocca (2000, p. 99), "o **conjunto de instruções** é a coleção de instruções que um processador pode executar" (grifo do autor). Em geral, estas instruções realizam operações bastante simples, como adições, subtrações, movimentação de dados dentro da memória, comparações entre posições da memória, etc. Esta simplicidade existe para aumentar a velocidade de execução, simplificar o *design* do processador e facilitar o trabalho do programador (PATTERSON, 2005). Uma vez que o processador executa milhões de operações por segundo, a complexidade da programação ficará nos níveis superiores de abstração, onde a programação geralmente é feita em linguagens de alto nível como C ou Pascal.

Para facilitar a programação, estas instruções são representadas através de mnemônicos. Num processador Intel da série x86, por exemplo, a instrução que gera uma interrupção de software é a instrução de número 0xCD[21]. Num código-fonte *assembly*, no entanto, este número será substituído pelo mnemônico INT (*Interruption*, do inglês "interrupção") (INTEL, 2006).

Nos processadores do tipo CISC[22], a instrução é composta por um byte. A maior parte das instruções, no entanto, recebem outros bytes que as acompanham. Estes bytes são chamados de **operandos**, e geralmente indicam a localização dos dados que serão manipulados durante a realização da operação (MONTEIRO, 1996). Num processador RISC[23], os operandos também estão presentes, embora não estejam em bytes separados – neste tipo de processador, todas as instruções (acrescidas de seus operandos) têm o mesmo número de bits (TANENBAUM, 1990).

Tanenbaum (1990) agrupa as instruções em sete tipos:

1. **instruções de transferência de dados**, que copiam dados de um lugar da memória para outro;

2. **operações diádicas**, que combinam dois operandos para produzir um resultado (por exemplo, uma instrução de soma);

21 Um conjunto de números ou letras precedido de "0x" significa que este é um número hexadecimal. Assim, 0xCD equivale ao número hexadecimal CD, ou ao número decimal 205.

22 Processador que pode executar várias operações de baixo nível em uma simples instrução. Do inglês *Complex Instruction Set Computer* (Computador com um Conjunto de Instruções Complexo).

23 Processador que possui um conjunto de instruções mais simples que o CISC. Do inglês *Reduced Instruction Set Computer* (Computador com um Conjunto de Instruções Reduzido).

3. **operações monádicas**, que têm um operando e produzem um resultado (por exemplo, uma instrução que incremente o valor de um determinado registrador em 1);

4. **comparações e desvios condicionais**, que comparam dois dados e transferem o controle do programa para uma posição especificada se eles forem iguais (ou diferentes, dependendo da instrução) – semelhante à instrução *if-then* (se-então) das linguagens de alto nível;

5. **instruções de chamada de procedimento**, que colocam o endereço atual na pilha[24] e transferem o controle do programa para outra posição – quando o procedimento termina, o endereço é desempilhado e o controle do programa é retornado à instrução seguinte de onde o procedimento foi chamado;

6. **controle de laço**, que permitem a execução repetida de partes de um programa, até que uma determinada condição seja satisfeita – de forma semelhante aos laços *do-while* (faça-enquanto) das linguagens de alto nível;

7. **entrada/saída**, através das quais é feita a comunicação com outros dispositivos. Neste grupo de instruções também entram as instruções de chamada de interrupção.

O Quadro 1 exemplifica a divisão acima, através de uma lista reduzida das operações do microprocessador MOS6502[25].

Quadro 1 - Conjunto reduzido de instruções do MOS 6502, agrupadas por tipo.

Tipo de Instrução	Instrução	Descrição da Operação
Transferência de Dados	LDA	Carrega dados da memória para o acumulador
Operações Diádicas	ADC	Soma dois operandos.
Operações Monádicas	INC	Incrementa uma posição da memória
Comparações e desvios condicionais	BEQ	Muda controle do programa se último resultado foi zero
	BPL	Muda controle do programa se último resultado foi negativo
Instruções de chamada de procedimento	JSR	Salta para sub-rotina
	RTS	Retorna de subrotina
Controle de laço		O 6502 usa instruções de desvio para o controle de laços.
Entrada/Saída	BRK	Gera uma interrupção de software

Fonte: JACOBS, 2002.

24 A pilha (*stack*) é uma área da memória que contém dados que são inseridos e lidos através de um algoritmo onde o último dado inserido é o primeiro a ser lido, semelhante a uma pilha de documentos.

25 Microprocessador de 8 bits criado pela MOS Techonology em 1975, extensivamente usado nos computadores pessoais e videogames nos anos 80.

2.1.2 Seqüência de Operação

O processador executa um ciclo repetitivo de operações, cada qual equivalente à execução de uma instrução. Este ciclo é chamado de **busca-decodificação-execução**, e é o centro da operação de todos os computadores (WEBER, 2000).

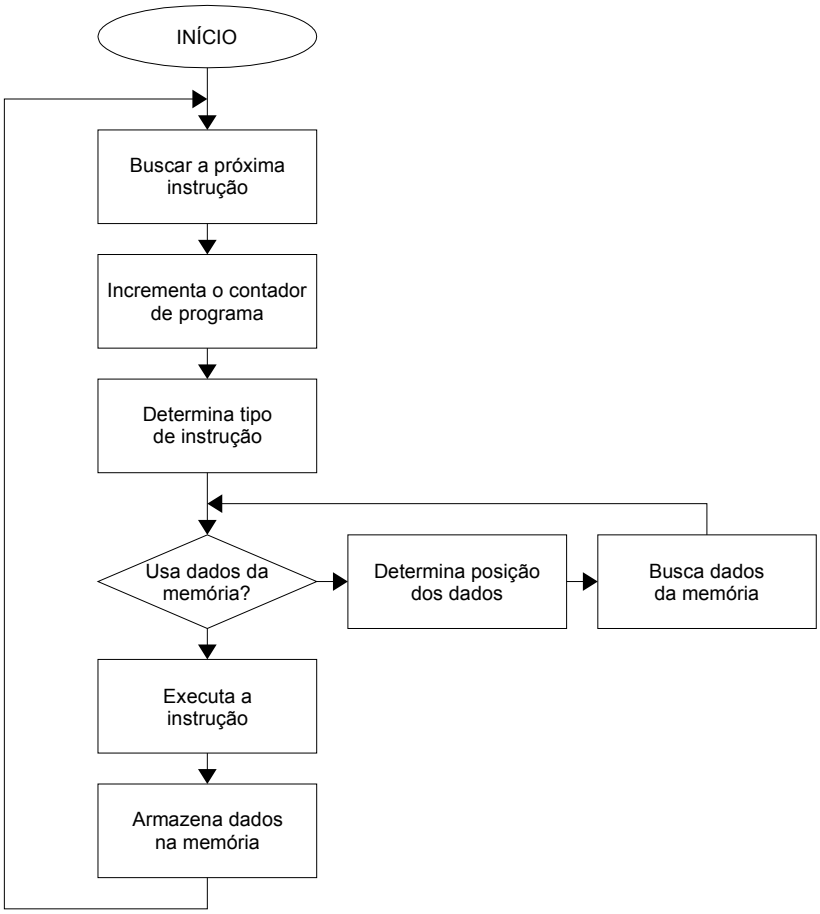

Figura 3 - Ciclo de execução de um microprocessador.

Tanenbaum (1990) divide a execução do ciclo em oito fases, apresentadas na Figura 3. O processador busca a próxima instrução da memória, guardando-a no registrador de instrução e atualizando o contador de programa. A seguir, ele determina o tipo da instrução e quantos operandos esta instrução utiliza, buscando estes operandos da memória. Finalmente, o processador executa a instrução, armazenando os resultados nos locais apropriados.

2.1.3 Registradores

O processador conta com uma pequena memória interna, que geralmente não passa de uns poucos bytes (desconsidera-se aqui a memória *cache*, que tem um propósito diferente). Se-

gundo Monteiro (1996, p. 137), "o resultado de uma operação aritmética ou lógica realizada pela UAL[26] deve ser armazenado temporariamente, de modo que possa ser reutilizado mais adiante (por outra instrução) ou apenas para ser, em seguida, transferido para a memória".

Para este propósito, o processador é fabricado com um conjunto de **registradores**. Os registradores são pequenos blocos de memória (geralmente entre 8 e 64 bits). Esta memória é de acesso muito rápido, mais rápido que qualquer outro bloco de memória do computador. Alguns destes registradores ficam disponíveis para serem usados pelo programador (os chamados registradores de propósito geral) enquanto outros têm propósitos específicos (MONTEIRO, 1996).

O **acumulador** é um registrador de propósito geral que é usado especialmente para operações lógicas e aritméticas. Em geral, todas as instruções lógicas e aritméticas de um processador podem ser realizadas utilizando os dados contidos no acumulador. Os modelos mais simples de processador contêm apenas um acumulador, enquanto nos modelos mais complexos encontram-se diversos acumuladores (WEBER, 2000).

O **ponteiro de instruções** (ou IP, do inglês *Instruction Pointer*) é um registrador que armazena o endereço de memória da próxima instrução a ser executada. É também chamado de **contador de programa** (ou PC, do inglês *Program Counter*) (WEBER, 2000). Seu valor geralmente será acrescido do número de bytes da instrução que está sendo executada, exceto nos casos em que a instrução modifica explicitamente o apontador de instruções, como no caso de uma instrução de desvio condicional ou de controle de laço.

O **ponteiro de pilha** (ou SP, do inglês *stack pointer*) é um registrador que armazena o endereço de memória da pilha. A pilha é uma área da memória principal onde o último dado a ser adicionado é o primeiro a ser retirado. Esta área é usada especialmente para armazenar o endereço de retorno em chamadas de sub-rotina, ou para armazenar os valores dos registradores em situações de troca de contexto[27] (MURDOCCA, 2000).

Os **sinalizadores**, ou *flags*, são um conjunto de informações de 1 bit que geralmente são armazenadas em um único registrador. Estas informações são geradas pela ALU, e representam a situação da última operação lógica ou aritmética, freqüentemente através de verdadeiro ou falso. Alguns sinalizadores também são usados para configurar certas funcionalidades do processador. O Quadro 2 exemplifica estes usos, mostrando os sinalizadores usados pelo processador MOS 6502, e são bastante semelhante ao sinalizadores usados por outros processadores.

26 Unidade Aritmética e Lógica, o mesmo que ALU.

27 Situação que ocorre em sistemas multitarefa onde, no momento em que o controle é passado de um processo para outro, todas as informações do processo devem ser armazenadas.

(JACOBS, 2002).

Quadro 2 - Sinalizadores do MOS 6502.

Nome	Função
Carry	Verdadeiro se a última operação resultou num transbordamento do primeiroou do último bit.
Zero	Verdadeiro se o resultado da última operação foi zero.
Interrupt	Enquanto for verdadeiro, o processador não responde a interrupções.
Decimal	Enquanto for verdadeiro, o processador usa o modo decimal nas operações aritméticas.
Break	Verdadeiro se uma interrupção foi gerada na última instrução.
Overflow	Verdadeiro se o resultado da última operação resultou num resultado inválido devido à falta de espaço (por exemplo, quando uma soma de dois números positivos resultou num número negativo devido ao fato do bit mais significativo ter sido mudado de 0 para 1).
Negative	Verdadeiro se a última operação resultou num número negativo.

Fonte: JACOBS, 2002.

2.1.4 Interrupções

Em sistemas multitarefa, é necessário que cada programa receba uma fatia de tempo de processamento, várias vezes por segundo. Isso significa que o processador não pode ficar parado, esperando uma resposta de requisição de E/S[28] (uma requisição para leitura de dados em um disquete, por exemplo) enquanto os outros programas ficam travados. Este problema é resolvido através do uso de **interrupções** (OLIVEIRA, 2001).

Segundo Tanenbaum (1990, p. 42),

> quando a CPU quer realizar E/S, ela carrega um programa especial para um dos canais, e diz ao canal para executá-lo. O canal manipula toda a E/S para e da memória principal, deixando a CPU livre para fazer outras coisas. Quando o canal termina, ele envia à CPU um sinal especial chamado **interrupção**, o que faz a CPU parar o que ela estava fazendo e dar atenção especial ao canal. (grifo do autor)

Uma interrupção é semelhante à chamada de uma sub-rotina – uma rotina é ativada e, no final do seu tratamento, o controle da execução é retornado ao programa principal. Mas como a interrupção pode ser gerada por hardware, ela pode ocorrer a qualquer momento (OLIVEIRA, 2001).

Os processadores geralmente admitem vários tipos de interrupções, cada uma delas

28 Entrada e Saída, referindo-se à comunicação entre dispositivos, ou à comunicação entre computador e usuário. Também referido pelo termo inglês I/O (*Input/Output*).

identificada por um número. Quando uma interrupção ocorre, o processador consulta a **tabela de vetores de interrupção** na memória principal para descobrir qual o endereço da rotina de tratamento da interrupção equivalente ao periférico que a gerou (OLIVEIRA, 2001).

Muitas vezes duas ou mais interrupções de hardware podem ocorrer simultaneamente. Para estes casos, existe um controle de prioridades por parte do processador ou de um **controlador de interrupções programável** (ou PIC, do inglês *Programmable Interrupt Controller*). Ainda assim, existem interrupções que não podem ser ignoradas – são as **interrupções não-mascaráveis** (ou NMI, do inglês *Non Maskable Interrupt*) (MURDOCCA, 2000).

As interrupções podem ser de três tipos:

- **interrupções de hardware**, geradas por algum componente do computador;
- **interrupções de software** (também chamadas de *traps*), geradas através de uma instrução do próprio processador egeralmente usadas para chamadas de sistema;
- **exceções**, geradas pelo próprio processador em decorrência de um erro, como uma divisão por zero ou a referência a um endereço de memória inexistente (OLIVEIRA, 2001).

2.2 Memória Endereçável

Tanenbaum (1990, p. 30) define a memória como sendo "[...] a parte do computador onde programas e dados são armazenados. [...] Sem uma memória onde os processadores possam ler e escrever informações, não haveria nenhum computador digital de programa armazenado."

Embora haja uma noção popular de que a memória de um computador é uma área contígua que pode ser lida e escrita livremente (como a memória RAM), esta definição está longe de ser verdadeira. Há regiões especiais de memória que mapeiam memória física, memória compartilhada com outros dispositivos (memória de vídeo, por exemplo), registradores de outros dispositivos, ROM[29], espelhamento de memória, etc (DEL BARRIO, 2001).

2.2.1 Memória RAM e ROM

O tipo mais comum de memória endereçável é a **memória RAM**. Esta sigla inglesa traduz-se por Memória de Acesso Aleatório (ou Randômico), ou seja, significa que qualquer re-

29 Memória somente para leitura. Do inglês *Read Only Memory*.

gião da memória pode ser acessada na mesma quantidade de tempo – ao contrário de uma fita de dados, por exemplo, onde o acesso é seqüencial e leva mais tempo para ler o final da fita do que o seu início. (MURDOCCA, 2000).

Do ponto de vista do nível convencional de máquina, a memória RAM é a mais simples de todas – é uma simples memória que permite livre leitura e escrita em qualquer posição. Seus bytes tem, via de regra, 8 bits (exceto em computadores muito antigos).

A **memória ROM** é um tipo de memória que permite somente a leitura. Do ponto de vista do nível convencional de máquina, a memória ROM funciona da mesma forma que a memória RAM, com uma diferença – a memória ROM não pode ser escrita (MURDOCCA, 2000). Exemplos de memória ROM são os cartuchos de videogameou a BIOS[30] nos computadores.

Existem ainda as memórias do tipo PROM (memórias somente-leitura programáveis, do inglês *Programmable Read-Only Memory*), que são unidades de memória ROM que podem ser reescritas (MURDOCCA, 2000). Do ponto de vista da emulação, este tipo de memória cai em uma das duas categorias acima – geralmente a memória do tipo EPROM[31] (que é reescrita por uma máquina especial) é acessada como uma memória ROM, enquanto a memória do tipo *flash*[32] é acessada de forma parecida com uma memória RAM.

2.2.2 Mapas de Memória

Segundo mencionado na Seção 2.2, a memória endereçável não é um seção contígua que pode ser acessada livremente. Parte desta memória é composta de regiões onde as leituras e escritas serão redirecionadas para um dispositivo, de forma a permitir a comunicação entre o programa e os dispositivos. O mapa contendo estas regiões é chamado **mapa de memória** (ou lista de regiões) (DEL BARRIO, 2001).

Por exemplo, o mapa de memória do videogame portátil *Game Boy*, fabricado pela Nintendo (de 64 kB de memória endereçável, no total) é apresentado no Quadro 3.

30 Sistema Básico de Entradas e Saídas: é o primeiro software que roda em um computador quando este é inicializado, e dá a seqüência para a inicialização do sistema operacional. Em alguns computadores possui também funções de acesso unificado ao hardware. Do inglês *Basic Input/Output System.*

31 Memória Somente Leitura Programável Apagável: tipo de pastilha de memória que pode ser apagado através de raios ultra-violeta e reescrito. Do inglês (*Eraseble Programmable Read-Only Memory*).

32 Memória do tipo PROM de alta velocidade que pode ter bytes específicos reescritos.

Quadro 3 - Mapa de memória do Game Boy.

Posição	Tamanho	Nome Interno	Descrição
0000-3FFF	16 kB		Memória ROM (banco 0)
4000-7FFF	16 kB		Memória ROM (banco 1)
8000-9FFF	8 kB	VRAM	Memória de vídeo
A000-BFFF	8 kB		Memória externa
C000-CFFF	4 kB	WRAM	Memória RAM (banco 0)
D000-DFFF	4 kB	WRAM	Memória RAM (banco 1)
E000-FDFF	7,5 kB	ECHO	Espelho da memória C000-DDFF
FE00-FE9F	160 bytes	OAM	Tabela de atributos de *sprites*
FEA0-FEFF	96 bytes		Não utilizado
FF00-FF7F	128 bytes		Portas de E/S
FF80-FFFE	127 bytes	HRAM	Memória alta
FFFF	1 byte		Registrador de Permissão de Interrupções

Fonte: KORTH, 2001, p. 1

No caso do Game Boy, o programa tem acesso a 64 kB de dados, mas apenas a 8 kB de memória RAM. As duas primeiras áreas de memória (que vão da posição 0x0 à posição 0x7FFF) são a memória ROM – ou seja, quando um cartucho de jogo é inserido no videogame, os dados contidos no cartucho são mapeados para estas posições de memória. Assim, quando o programa estiver acessando uma destas posições, ele estará acessando diretamente os dados do cartucho. Se um cartucho de memória extra for inserido no videogame, ele será mapeado para as posições de 0xA000 a 0xBFFF, e poderá ser usado livremente como a memória RAM (KORTH, 2001).

As posições de 0x8000 a 0x9FFF equivalem à memória de vídeo. Qualquer dado que for escrito em uma destas posições resultará em uma mudança na tela do usuário. Assim, esta região de memória é útil para exibir informações para o usuário, mas não serve para o armazenamento de dados diversos. Estes dados devem ser armazenados na região onde a memória é do tipo RAM.

2.2.3 Espelhamento de Memória

Em alguns casos, partes da memória são espelhadas em outras posições. Isso significa que o acesso a uma posição de memória equivale exatamente ao acesso da posição que a espelha. Por exemplo, em uma máquina hipotética em que a memória de posição 0x0 a 0xFF fosse

espelhada nas posição de 0x100 a 0x1FF, um dado escrito na posição 0x10 poderia ser subseqüentemente lido na posição 0x110.

Na realidade, embora exista uma diferença lógica no endereçamento, não há uma diferença física: ambos os endereços apontam para a mesma posição física. Isto é usado especialmente em computadores muito antigos, onde o programa é escrito em linguagem *assembly* levando em conta o menor consumo possível de ciclos de *clock* de um processador. Geralmente, o processador gasta menos ciclos acessando uma memória próxima do que uma memória distante[33] e, portanto, os espelhos de memória são úteis para deixar áreas de memória mais próximas.

Em computadores atuais, no entanto, algumas vezes o espelhamento é feito de forma física. Segundo a Microsoft (2006, p. 1), "memória redundante provê ao sistema um banco de memória de resgate quando um banco de memória falha. O espelhamento de memória divide os bancos de memória em um conjunto espelhado."

Um exemplo de memória espelhada pode ser visto no Quadro 3, onde as posições 0xE000 a 0xFDFF da memória do *Game Boy* espelham as posições 0xC000 a 0xDDFF.

2.2.4 Bancos de Memória

Em computadores mais antigos, o espaço de endereçamento muitas vezes era pequeno demais para acessar toda a memória física de uma vez só. Assim, foram desenvolvidos dispositivos especiais (colocados dentro do microprocessador) que permitissem selecionar qual bloco de memória seria acessado em um determinado momento. Estes blocos de memória física são chamados de **bancos**, e a técnica de troca de bancos é chamada de *bankswitching* (em inglês, literalmente, troca de bancos) (DEL BARRIO, 2001).

Um exemplo disto é o Atari 2600, videogame fabricado pela Atari entre 1977 e 1991. O Atari tem um espaço de apenas 4 kB para endereçamento de ROM, enquanto os jogos produzidos para ele iam de 2 kB a 32 kB. Os jogos de mais de 4 kB usavam bancos de memória, que eram trocados à medida que era necessário. Assim, os jogos de 8 kB usavam 2 bancos, os de 16 kB usavam 3 bancos, e assim por diante (KORTH, 2006).

2.3 Sistema de Vídeo

A visão é o método pelo qual os humanos absorvem a maior quantidade de informa-

33 Isto não acontece devido ao fato da memória ter diferentes taxas de acesso para diferentes posições, mas sim porque o processador muitas vezes precisa realizar operações como troca de página antes de acessar determinadas partes da memória.

ção, mais rapidamente. Por esta razão, o vídeo é o método mais importante de saída de informações (DEL BARRIO, 2001).

O sistema de vídeo é composto por duas partes: (i) a **unidade de processamento gráfico** (GPU, do inglês *Graphic Processing Unit*), geralmente composto por uma placa ou um *chip*, e (ii) a **saída de vídeo**, que geralmente é um monitor, um *display* de cristal líquido[34] ou uma televisão.

2.3.1 Saída de Vídeo

A saída de vídeo é responsável por exibir as imagens geradas pela unidade de processamento gráfico (GPU). Geralmente será composta por uma televisão (no caso de computadores pessoais antigos ou videogames), um monitor (no caso de computadores pessoais atuais ou estações de trabalho) ou um *display* de cristal líquido (no caso de dispositivos portáteis ou telefones celulares).

Estes tipos de saída de vídeo usam principalmente três técnicas de exibição de imagens. A primeira técnica desenvolvida, e a que hoje ainda é a mais usada (embora em declínio) é o **tubo de raios catódicos** (CRT, do inglês *Cathode Ray Tube*). Os monitores do tipo CRT são baseados em um feixe de elétrons que desenha, linha por linha, a imagem em uma tela de fósforo. Quando o fósforo é atingido pelo feixe de elétrons, ele brilha a uma determinada intensidade, compondo a imagem na tela. Como o fósforo mantém a cor apenas por um curto espaço de tempo, a imagem precisa ser redesenhada várias vezes por segundo (MONTEIRO, 1996). Veja a Figura 4.

Outras técnicas comuns para a exibição de imagens são o uso de **cristal líquido** e o uso do **plasma**. O monitor de cristal líquido utiliza moléculas de cristal líquido para exibição de imagens, enquanto o monitor de plasma utiliza uma mistura química de xenon e neon. No entanto, ambos os tipos de monitor têm o funcionamento semelhante a um monitor CRT quando conectados a um computador e, por isso, este será o tipo de monitor que este trabalho focará com mais profundidade (MONTEIRO, 1996).

34 Os *displays* são pequenos monitores de cristal líquido usados em dispositivos portáteis. Os cristais líquidos são uma classe de moléculas que estão em um estado da matéria entre o líquido e o gasoso conhecido como estado líquido cristalino.

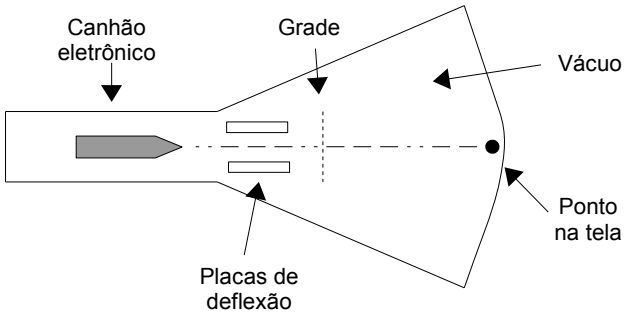

Figura 4 - Seção de um CRT.
Fonte: TANENBAUM, 1990.

2.3.1.1 O Caminho do Feixe de Elétrons

Os primeiros computadores utilizavam um sistema vetorial, onde o feixe de elétrons era movimentado livremente. Por exemplo, se o programa quisesse desenhar um triângulo na tela, o feixe era movimentado desenhando rapidamente três linhas, equivalentes aos três lados do triângulo.

Este método logo caiu em desuso devido a duas limitações. A primeira era que apenas os formatos das imagens podiam ser desenhados, mas elas não podiam ser preenchidas. Outro problema era que as imagens tinham que ser redesenhadas continuamente e, se houvessem muitas imagens na tela, havia uma grande sensação de *flicker*[35] (WIKIPEDIA, 2006b).

Por estas razões, os monitores vetoriais foram substituídos pelos monitores do tipo *raster* (do inglês, "rastro"). Em um monitor do tipo *raster* o feixe de elétrons desenha toda a tela seqüencialmente, de cima para baixo e da esquerda para a direita, várias vezes por segundo. Desta forma, a tela tem um número determinado de "pontos", que são o menor elemento de imagem possível de ser mostrado em uma só cor. Estes pontos são chamados *pixeis*[36] (DEL BARRIO, 2001).

O caminho percorrido pelo feixe de elétrons pode ser visto na Figura 5. A compreensão deste conceito é fundamental para a construção de emuladores.

35 Sensação de que a imagem está "piscando" muito rapidamente na tela. Isto ocorre devido ao redesenho muito lento da imagem, e pode causar desconforto e dores de cabeça no usuário.
36 Do inglês *picture element*, é a menor quantidade de informação possível de ser exibida em um monitor, ou seja, cada um dos pontos da imagem. Este elemento de informação não é um ponto ou um quadrado, mas sim uma amostra abstrata.

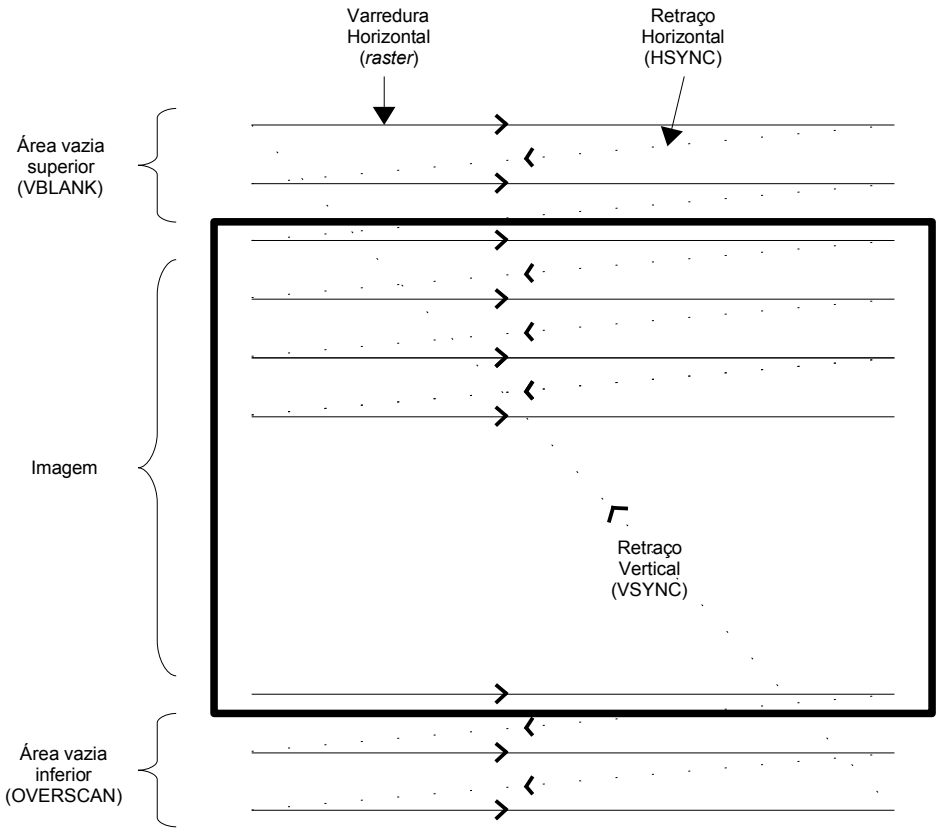

Figura 5 - Caminho percorrido pelo feixe de elétrons em um monitor CRT.

A primeira operação a ser executada é o **retraço vertical** (ou VSYNC, do inglês, *vertical synchronization*, ou sincronização vertical). No retraço vertical, o monitor recebe a informação de que o feixe de elétrons deve se deslocar para o canto superior esquerdo do usuário. Durante este tempo (que num televisor leva o equivalente à varredura de 3 linhas) o sinal é desligado, caso contrário a imagem na tela ficaria defomada (MONTEIRO, 1996).

Existem vários modelos de televisores e monitores, alguns exibindo um espaço de imagem extra na parte superior, outros na parte inferior, e outros em nenhuma. Assim, os computadores e videogames que se conectam em televisores geralmente usam um padrão onde a parte superior (VBLANK, do inglês *Vertical Blank*, ou espaço vertical em branco) e a parte inferior da imagem (OVERSCAN, do inglês, varredura posterior) não são exibidas. Segundo pesquisas realizadas pela empresa norte-americana Atari, um computador usando um VBLANK de 37 linhas de varredura e um OVERSCAN de 30 linhas de varredura funcionará em todos os tipos de televisores (WRIGHT, 1979). Monitores modernos não têm as áreas de VBLANK e OVERSCAN.

Após o VBLANK (ou o VSYNC em monitores modernos), a imagem começa a ser de-

senhada, linha por linha. Logo após a primeira linha ser desenhada (da esquerda para a direita), o elétron precisa retornar de volta para o lado esquerdo da tela. O período em que isto acontece é chamado **retraço horizontal** (HSYNC , do inglês *Horizontal Synchronization*, ou sincronização horizontal) (MONTEIRO, 1996). O retraço horizontal leva, em média, um quarto do tempo necessário para a varredura, e durante este tempo o feixe de elétrons é, obviamente, desligado (WRIGHT, 1979).

Depois da imagem ser desenhada até embaixo, o feixe de elétrons passa a área de OVERSCAN e novamente acontece o retraço vertical, assim sucessivamente, várias vezes por segundo.

O freqüência com que a imagem é desenhada na tela é chamada de **freqüência vertical** ou taxa de atualização. Esta taxa é medida em Hertz, e a medida é exatamente o número de vezes que a imagem é atualizada por segundo (por exemplo, um monitor operando a 60 Hz atualiza a imagem 60 vezes por segundo) (MONTEIRO, 1996).

Os televisores usam uma taxa fixa de atualização, que depende do país de origem. As duas principais taxas são as de 60 Hz (usada na América do Norte e no Brasil) e de 50 Hz (usada na Europa) (MURDOCCA, 2000).

Este fator é importante e deve ser levado em consideração na construção de emuladores de computadores e videogames mais antigos. No caso do Atari 2600, se um jogo escrito para um sistema de 50 Hz for executado num sistema de 60 Hz, o jogo rodará 17% mais rápido que no sistema original (porque, neste tipo de arquitetura, a velocidade do retraço vertical controla a velocidade de todo o resto do hardware) (WRIGHT, 1979).

Os monitores mais modernos aceitam várias freqüência verticais, geralmente variando entre 50 Hz e 90 Hz. Como os monitores modernos normalmente não utilizam entrelaçamento[37], a freqüência precisa ser maior que a da televisão para não irritar o olho humano.

2.3.2 Unidade de Processamento Gráfico

Uma unidade de processamento gráfico (GPU) é um *chip* ou uma placa presente em um computador que gera as imagens que serão enviadas para o monitor. As GPU também são conhecidas como placas de vídeo, VPU (*Video Processing Unit*, do inglês Unidade de Processa-

37 O entrelaçamento é uma técnica onde apenas a metade das linhas é atualizada em cada retraço vertical, na primeira vez as linhas pares, e na segunda as ímpares, sucessivamente. Isto faz com que um televisor operando a 50 Hz só complete um retraço completo 25 vezes por segundo, sem que o olho humano consiga perceber.

mento de Vídeo) ou VDP (*Video Display Processor*, do inglês Processador de Exibição de Imagens).

Existem vários sistemas de GPU, das quais este trabalho descreverá os principais. É importante lembrar que uma determinada placa pode ter mais de um sistema, embora use apenas um por vez. Exemplo disto são as placas atuais de PC, onde elas podem usar o modo terminal (modo texto), *framebuffer*[38] (modo gráfico 2D) ou o modo gráfico tridimensional.

2.3.2.1 Gráficos Vetoriais

Este modo já foi descrito no início Seção 2.3.1.1. Ao invés do tipo *raster*, onde os gráficos são desenhados na tela de cima a baixo, este modo controla diretamente o feixe de elétrons, usando-o para desenhar as formas na tela. É bastante usado em osciloscópios, mas muito raramente em computadores ou videogames. Uma exceção disto é o videogame Vectrex, produzido pela companhia americana General Consumer Electric entre 1982 e 1984 (WIKIPEDIA, 2006c).

Uma imagem do videogame Vectrex pode ser visto na Figura 6. Esta imagem foi gerada a partir do emulador Vecx. É possível notar a falta de cores e de preenchimento nas imagens. Como este tipo de gráfico foi raramente utilizado, a emulação dele não será mencionada neste texto. Todos os outros tipos de gráficos usam o modo *raster* de retraço no monitor.

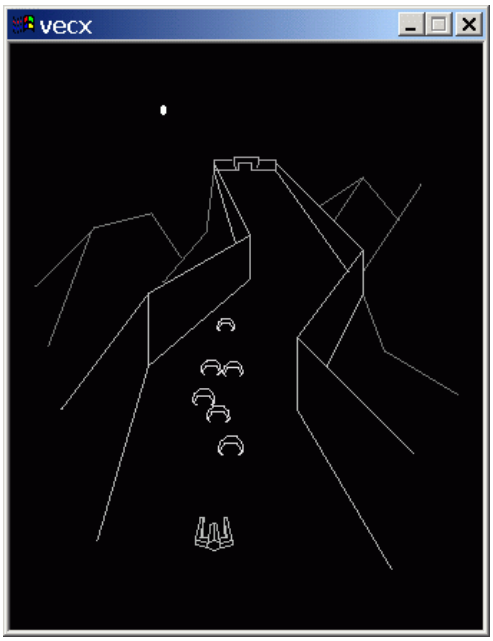

Figura 6 - Imagem do videogame Vectrex.

38 Sistema onde a memória possui um mapa de bits exatamente igual às informações que estão sendo exibidas na tela.

2.3.2.2 Gráficos Sincronizados

As GPU de gráfico sincronizados são um tipo de placas gráficas que possuem apenas uma pequena quantidade de memória. Como a memória é pequena, não é possível armazenar as posições dos pixeis, e o programa que está sendo executado precisa estar constantemente atualizando a tela, tarefa destinada à placa de vídeo nos outros tipos de processadores gráficos.

Um exemplo deste tipo de placa gráfica é o TIA (Adaptador de Interface de Televisão, do inglês *Television Interface Adapter*), que era usado no videogame Atari 2600. A atualização da tela do Atari 2600 tinha quatro fases: (i) 3 linhas de VSYNC; (ii) 37 linhas de VBLANK; (iii) 192 linhas de imagem; e (iv) 30 linhas de OVERSCAN. Assim, um jogo precisava estar sincronizado com o feixe de elétrons da televisão. Durante as 70 linhas de VSYNC, VBLANK e OVERSCAN, ele podia fazer o processamento do jogo (contar os pontos, verificar colisões, verificar dispositivos de entrada, etc), mas durante as 192 linhas de imagem, o processamento precisava parar para que o programa fizesse o desenho na tela. Isto acontecia 60 vezes por segundo. Isso significava que um jogo de Atari 2600 gastava cerca de 75% do seu tempo gerando a imagem, e apenas 25% fazendo o processamento (ISRAEL, 2004).

Por isto, a programação de jogos de Atari 2600 era considerada uma tarefa desafiadora, pois o programador precisava constantemente manter em mente o número de ciclos de *clock* de cada uma de suas instruções. Se o processamento acabasse ultrapassando as 70 linhas disponíveis para isso (ou, por descuido, o programador fizesse uma rotina que usasse menos de 70), a imagem ficaria fora de sincronia, resultando num programa inutilizável.

Vários programas utilizavam técnicas pouco ortodoxas de programação, de modo a obter o máximo do processador gráfico. Um exemplo disto é o Video Chess, um jogo de xadrez desenvolvido pela Atari em 1978 para o Atari 2600. O jogo permitia que o usuário jogasse xadrez contra o computador. Durante o processamento da jogada do computador, a tela ficava preta. Assim, o jogo não gastava ciclos com a atualização da imagem, de modo a acelerar o processamento. Uma vez que o computador estivesse terminado de processar a sua jogada, a imagem voltava a ser exibida.

2.3.3 Adaptadores Gráficos de *Tiles* e *Sprites*

Segundo Del Barrio (2001, p. 110), "motores gráficos baseados em *tiles* e *sprites* eram o tipo mais comum em videogames nos anos 80 e 90. Eles eram muito populares devido à sua habilidade em produzir bons gráficos e animações sem muito uso de CPU ou memória."

Nos jogos de computador e videogame, é muito comum existir uma repetição de padrões de imagem, representando terreno, cenário, parede de tijolos, etc. Um exemplo de uma imagem com estes padrões repetidos pode ser visto na Figura 7.

Observando-se a imagem da esquerda na Figura 7, nota-se a existência de padrões repetidos – as paredes, pedras, etc. Na imagem da direita, está um mapa com a legenda destes padrões. Estes padrões repetitivos são chamados de *tiles* (do inglês "azulejos").

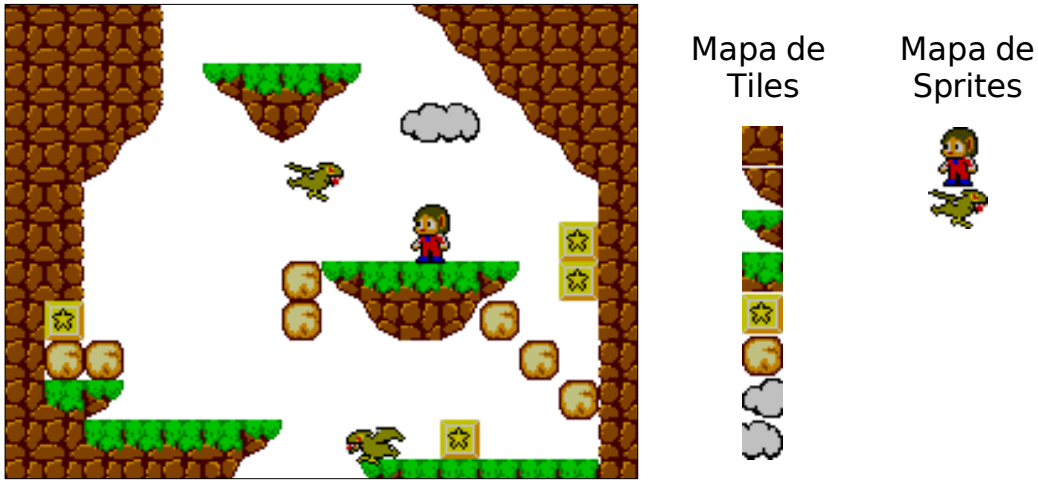

Figura 7 - Exemplo de mapa de *tiles* e *sprites*.

O uso de *tiles* minimiza bastante o uso de memória. Em uma posição da memória é guardado um *tileset* (do inglês, "conjunto de *tiles*"), como o da imagem da direita na Figura 7, enquanto em outra posição da memória é guardado um mapa das posições dos *tiles* na tela (chamado *tilemap*, do inglês "mapa de *tiles*"). Desta forma, é possível repetir um *tile* quantas vezes forem necessárias, sem ter que copiar toda a memória várias vezes. Isto, além de minimizar o uso de memória, acelera o processamento, permitindo que a cópia de um *tile* seja realizada apenas com uma ou duas instruções do processador, bem como facilitando a rolagem de tela.

Os *sprites* funcionam de maneira semelhante aos *tiles*, porém são mais completos. Enquanto os *tiles* geralmente são usados para o fundo da imagem, os *sprites* são usados para as personagens. Assim, o computador geralmente permite um ajuste mais exato da posição do *sprite*, permite que hajam modificações no formato do *sprite* e, como o *sprite* normalmente vai aparecer à frente dos *tiles*, parte dele é transparente. No mapa de *sprites* da Figura 7, pode-se notar que os *sprites* tem tamanhos variados.

Existe um tratamento de prioridades neste tipo de processador gráfico. Na maioria dos casos há mais de uma camada de *tiles* e mais de uma camada de *sprites*. Cada uma das camadas tem uma prioridade diferente – assim, quando dois *tiles* ou *sprites* ocuparem a mesma posição

na tela, o que tiver a prioridade menor ficará escondido debaixo daquele que tem a prioridade maior. Embora os *sprites* geralmente tenham uma prioridade maior que os *tiles*, este nem sempre é o caso. Alguns jogos, por exemplo, podem desejar que a personagem apareça à frente do cenário mas passe por detrás de uma coluna – neste caso, os *tiles* da coluna terão prioridade superior ao *sprite* do jogador.

Este tipo de processador gráfico geralmente possui tratamento de colisões. Assim, o jogo pode requisitar à placa de vídeo a informação sobre se dois *sprites* estão se sobrepondo (colidindo) na tela em um determinado momento. Num jogo de naves, por exemplo, é possível saber se o *sprite* do míssil disparado pela nave do jogador está colidindo com a nave adversária, acionando assim a subrotina que destrói a nave em questão (DEL BARRIO, 2001).

2.3.3.1 Framebuffer

O sistema gráfico mais simples de ser emulado é o *framebuffer*. Neste sistema existe na memória um mapa exato dos *pixeis* que aparecem na tela. Assim, se o programa escreve um byte na área da memória de vídeo, na próxima varredura vertical o pixel aparecerá automaticamente na posição equivalente. A palavra *framebuffer* significa "memória interna de quadro", já que o quadro inteiro da imagem é guardado na memória.

O *framebuffer* é, portanto, uma matriz de dados do tamanho da tela. Se a tela tiver 320x200 pixels, a matriz do *framebuffer* será de 640 por 480 unidades (ou seja, uma matriz de 307.200 unidades). O tamanho do dado das unidades vai depender do sistema usado (com ou sem paleta[39]) e do número de cores desejadas.

No sistema sem paleta, cada dado da matriz contém a informação de cor usando algum sistema de armazenamento de cores, como o RGB[40]. Quanto maior o dado, maior o número de cores que podem ser exibidas. Se o dado for de 8 bits, 256 cores podem ser exibidas. Nos computadores atuais, é comum o uso de 32 bits, permitindo a exibição de 4 bilhões de cores simultâneas.

No sistema com paleta, existe um mapa de cores, com um número correspondente a cada cor. No dado da matriz, é armazenado o número da cor. Desta forma é possível exibir, por exemplo, um universo de 4 bilhões de cores usando apenas 1 byte por pixel. Mas neste caso,

39 A paleta é um subconjunto da gama total de cores suportadas por um sistema gráfico. Ela funciona como um índice, onde cada cor na paleta recebe um número e, para cada pixel, um destes números é armazenado.

40 Sistema de armazenamento de cor onde a cor é formada pela mistura das intensidades de vermelho, verde e azul (por exemplo, a cor 0x2040A0 teria 0x20 de vermelho, 0x40 de verde e 0xA0 de azul, sendo 0x0 a ausência de cor e 0xFF a saturação total). Do inglês *Red Green Blue*, ou Verde Vermelho Azul.

apenas 256 cores podem ser exibidas simultaneamente.

O sistema *framebuffer* utiliza muito mais memória que os outros sistemas. Um exemplo seria a definição de vídeo usada amplamente nos Pcs atuais: 1024x768 pixeis, com 32 bits de cores. Isto significa que a memória de vídeo necessita de, no mínimo, 3,14 Mb (1024 x 768 x 4 bytes).

Para as operações gráficas, é usada uma técnica chamada *blit* (do inglês BLT - *Bit Block Transfer*, ou "transferência de bloco de bits"). Neste caso, uma grande quantidade de bits é transferida de uma porção da memória para outra, gerando assim a imagem. Por exemplo, se uma figura fosse ser exibida na tela, a operação de *blit* copiaria todos os bytes da memória principal (onde a imagem está armazenada) para a memória de vídeo, exibindo assim a imagem para o usuário. A técnica de *blit* geralmente faz algum tipo de modificação ou conversão na imagem, mudando sua resolução e tornando transparente as áreas que não devem ser exibidas.

A maior parte dos *framebuffers* atuais tem mais memória disponível do que o necessário. Isto permite o uso de uma técnica para acelerar a exibição gráfica, que é o armazenamento das imagens a serem coladas na tela principal na própria memória gráfica. Como a maioria das placas modernas possuem a técnica de *blit* implementada em hardware, a operação é muito mais rápida. Este tipo de placa é geralmente conhecida por "placa acelerada 2D" (DEL BARRIO, 2001).

Em algumas situações, a memória extra é igual ou maior que o tamanho de um quadro inteiro. Nestas ocasiões, pode-se usar uma técnica chamada de *page flipping* (do inglês "troca de página"), onde enquanto o quadro de exibição está sendo mostrado na tela, o quadro seguinte já pode ir sendo montado. Quando é feito o retraço vertical, o novo quadro é exibido e o que acabou de ser exibido se torna o secundário, através de uma instrução executada pelo acelerador gráfico (WIKIPEDIA, 2006d).

2.3.3.2 Terminais de Texto

O terminal de texto é um sistema gráfico que permite apenas a exibição de texto. São usados em praticamente todos os computadores, e funcionam internamente de maneira parecida com o sistema de *frambebuffer*. A diferença é que a matriz, ao invés de guardar informações de pixeis, guarda informações de caracter. Esta informação geralmente é guardada em 2 bytes, um contendo o código do caractere, e outro contendo as informações de atributo (como cor da letra e cor do fundo).

O sistema IBM PC usa, por padrão, 4000 bytes de memória de vídeo para este modo. São 25 linhas e 80 colunas de texto, cada caractere usando 2 bytes. Este sistema é, portanto, muito mais econômico que qualquer outro (TANENBAUM, 1990).

2.3.3.3 Placas Gráficas Tridimensionais

Placas gráficas tridimensionais ou placas 3D são processadores gráficos que contém um número de instruções criadas especificamente para exibição de gráficos tridimensionais, como tratamento de volume, distância, sombreamento, etc. Estas instruções variam de placa para placa, mas geralmente o acesso a elas é feita de forma unificada através de uma biblioteca como o OpenGL[41], Mesa 3D[42] ou Direct3D[43].

Devido à complexidade deste tipo de hardware, este assunto não será abordado neste trabalho.

2.4 Sistemas de Entrada

Dispositivos de entrada são dispositivos usados por um usuário para inserir dados em um computador. Dispositivos de entrada comuns em computadores são teclados e mouses, e joysticks nos videogames.

Toda vez que o usuário executa uma operação em algum destes dispositivos (por exemplo, pressiona uma tecla no teclado), esta informação é armazenada em um lugar específico da memória. Existem duas técnicas para detectar quando o usuário executou alguma operação (e, portanto, um dado da memória foi modificado), *polling* e interrupção.

A técnica de ***polling*** (do inglês "sondagem") é a mais simples das duas e geralmente é usada para sistemas monotarefa[44]. Neste tipo de sistema, o programa precisa verificar freqüentemente se alguma das posições de memória referentes ao dispositivo de entrada foi modificada, e agir de acordo.

Como isto se torna complicado nos sistemas multitarefa, existe outro sistema que onde cada dispositivo tem um **controlador**, que gera uma **interrupção** de hardware para o processa-

41 Biblioteca que contém a especificação gráfica de uma API multilinguagem e multiplataforma para escrever aplicativos que acessem aceleradores gráficos 3D.
42 Biblioteca gráfica semelhante ao OpenGL, mas de fonte aberto.
43 Biblioteca de acesso a aceleradores gráficos produzida pela Microsoft e parte do pacote DirectX, compatível apenas com sistemas operacionais Windows.
44 Sistema que, ao contrário dos sistemas multitarefa, executa apenas um programa por vez. Exemplos são o sistema operacional DOS e os videogames.

dor toda vez que o usuário efetua alguma operação no dispositivo (MURDOCCA, 2000).

2.5 Considerações Finais

Uma arquitetura de hardware é formada por um grande número de componentes, e a natureza expansível dos computadores modernos faz com que o número de componentes conectáveis seja ainda maior. Este trabalho se deterá apenas nos principais componentes, presentes em todos os computadores, que são o microprocessador, a memória, o vídeo e o sistema de entrada. Dos sistemas de vídeo mencionados, serão tratados apenas os síncronos, *tiles/sprites, framebuffer* e terminais de texto.

3 TÉCNICAS TRADICIONAIS NA CONSTRUÇÃO DE EMULADORES

Segundo foi explicado no Capítulo 1, um emulador é um programa que imita o comportamento de uma arquitetura de hardware, permitindo que programas escritos para aquela arquitetura possam ser executados em outra. Neste Capítulo 3 é discutida a forma como tradicionalmente são construídos os emuladores.

Neste Capítulo, serão citados vários blocos de código em C[45], que é uma das linguagem mais usadas para escrever emuladores. Em vários destes blocos, fica subentendida e pré-existência de certos procedimentos, onde aparece a sua chamada mas não têm sua implementação mostrada.

3.1 Estrutura Básica de um Emulador

Existem algumas diferenças fundamentais entre a construção de uma arquitetura física de hardware e a construção de um emulador.

A primeira diferença fundamental é que os diferentes componentes de hardware funcionam em paralelo, enquanto eles precisam ser emulados um após o outro em um emulador. Isto acontece devido ao fato de que um programa, a não ser que esteja sendo executado em um computador multiprocessado (isto é, que tenha mais de um processador rodando em paralelo), pode apenas executar uma operação de cada vez. Os sistemas operacionais multitarefa dão ao usuário a impressão de que os programas estão rodando em paralelo enquanto, na realidade, cada programa roda em uma fatia de tempo muito pequena (geralmente medida em milésimos de segundo), sucessiva e repetidamente. A este método dá-se o nome de *time sharing*, ou tempo compartilhado (DEL BARRIO, 2001).

45 Linguagem de programação imperativa e funcional criada por Dennis Ritchie e Ken Thompson nos anos 70. É uma linguagem de alto nível, mas que permite o acesso direto à memória e a partes do hardware. Linguagem extremamente popular, é a mais usada para escrita de aplicações de sistema.

E é exatamente este método de *time sharing* que é usado para fazer com que o usuário tenha a ilusão de que os vários componentes do emulador estão rodando em paralelo, que é fazer a emulação de um dispositivo um após o outro, milhares de vezes por segundo. Isto se atinge através de um laço, que para um emulador simples poderia ser o seguinte:

```
1       int ciclos;
2
3       while(continuar_emulacao)
4       {
5           emular_cpu(&ciclos);
6           emular_graficos(ciclos);
7           emular_som(ciclos);
8           emular_outros_dispositivos(ciclos);
9       }
```

No código acima, é primeiro feita a emulação do microprocessador, depois dos gráficos, do som e assim por diante. Como a execução este laço acontece milhares de vezes por segundo, o usuário não percebe que os dispositivos não estão sendo emulados simultaneamente.

Uma segunda diferença básica acontece em relação ao controle do tempo. Em uma arquitetura física de hardware, o sinal de *clock* é gerado por um cristal com uma certa freqüência, de modo a manter todos os dispositivos em sincronia. Já em um emulador, isto não acontece desta forma. Neste caso, o primeiro dispositivo a ser emulado, que é o microprocessador "gastar" um determinado número de ciclos de *clock* a cada operação (interpretação de uma instrução). Esta informação é então passada a cada um dos dispositivos que serão emulados a seguir, de forma que eles executem o número de instruções equivalente àquele número de ciclos de *clock* (DEL BARRIO, 2001).

Por exemplo, imagine-se a situação hipotética onde o emulador vai iniciar a emulação de um bloco de código. A função que emula o processador lê o primeiro byte da memória e descobre que este byte equivale a uma instrução de incremento de um registrador. A função executa então o incremento, retornando ao laço principal a informação de que a instrução consumiu 5 ciclos de *clock*. O emulador passa então à função que emula o vídeo, informando que esta deve executar o equivalente a 5 ciclos de *clock* (por exemplo, desenhar 5 pixeis na tela). Após isto o emulador passa à função que emula o som, emulando 5 ciclos de *clock*, e assim por diante, até que o laço esteja completo e o processo se inicie novamente com a próxima instrução (DEL BARRIO, 2001).

No código mostrado acima, isto pode ser visto através da variável ciclos. Na primeira

chamada à função emular_cpu, a variável é passada por referência[46], de modo que o número de ciclos gastos possa ser retornado. A seguir, a variável ciclos é passada por valor[47] para as outras funções, para que estas tenham condições de saber o quanto devem emular antes de retornar o controle ao laço principal.

Em muitos casos, dois componentes não funcionam na mesma velocidade, mas ao invés disto usam um **multiplicador de *clock***. Neste caso é necessário que se façam as devidas conversões. Um exemplo disto é o caso do Atari 2600. Nesta arquitetura, o processador roda a 1,19 Mhz, enquanto a placa de vídeo roda a 3,58 Mhz (portanto, 3 vezes mais rápido) (WRIGHT, 1979). Isto significa que o número de ciclos deve ser multiplicado por 3 quando for passado para a emulação do vídeo. Por exemplo, se uma instrução executada pela CPU levar 5 ciclos, um total de 15 ciclos deve ser passado ao vídeo.

3.2 Emulação do Microprocessador

A emulação do microprocessador é crucial para o desempenho do emulador, especialmente nos computadores modernos onde a CPU tem uma velocidade bastante alta. Na Seção 1.5 discutiram-se as diferentes técnicas utilizadas na construção de um emulador. Todas estas técnicas dizem respeito especialmente à emulação do microprocessador, e este trabalho se deterá na emulação por interpretação.

3.2.1 Registradores

É necessário manter as informações dos registradores e das *flags* entre a execução de uma instrução e outra. Estas informações são guardadas em variáveis.

Como a quantidade de memória gasta com os registradores é pequena, e a velocidade de acesso a eles é crucial, deve-se investir no acesso rápido, nem que haja um gasto de memória maior que o necessário. Geralmente vale a pena dividir o registrador de *flags* em várias variáveis, uma para cada *flag*. Isto torna o acesso mais rápido. Fayzullin (2006, p. 7) acrescenta ainda,

> Deve-se tentar usar apenas inteiros do tamanho da base suportada pelo microproces-

46 Quando uma variável é passada para uma função por referência, isto significa que, na verdade, um ponteiro da variável é passado. Isto permite que o valor da variável seja modificado dentro da função, e esta modificação se reflita fora da função.

47 Quando uma variável é passada para uma função por valor, isto significa que, na verdade, uma cópia da variável é passada. Qualquer modificação feita à variável se refletirá apenas dentro do escopo da função, sendo que uma vez que o controle seja retornado pela função, o valor da variável será o mesmo que antes da chamada.

> sador, isto é, use *int* ao invés de *short* ou *long*[48]. Isto reduzirá a quantidade de código que o compilador gera para fazer a conversão entre diferentes comprimentos de inteiros. Também diminuirá o tempo de acesso à memória, pois algumas CPUs funcionam mais rapidamente quando estão lendo ou escrevendo dados do tamanho da base alinhada ao tamanho da base do limite do endereço. (tradução nossa)

Alguns registradores permitem um uso da palavra reservada *register*, que faz com que aquela variável seja armazenada em um registrador, de modo a acelerar o acesso. Isto é uma boa idéia para o armazenamento de registradores de acesso mais comum, como o ponteiro de instruções e o acumulador. No entanto, a maioria dos compiladores modernos já calcula qual a variável será mais acessada e a coloca no registrador automaticamente (FAYZULLIN, 1997).

3.2.2 Seqüência de Operação

A Seção 2.1.2 apresentou o ciclo **busca-decodificação-execução** usado pelo processador para seu funcionamento, e é baseado neste ciclo que a emulação de um processador funciona.

O primeiro passo é buscar na memória o byte para o qual o contador de instruções aponta. Baseado neste byte, o processador pode saber se ele precisa buscar mais bytes da memória para complementar a instrução, isto é, se a instrução possui parâmetros. Isto acontecerá apenas em processadores CISC. Nos processadores RISC, todas as instruções acrescidas de seus parâmetros possuem o mesmo comprimento, e este passo separará a instrução de seu parâmetro (DEL BARRIO, 2001).

O segundo passo é a decodificação, ou seja, descobrir qual instrução equivale àquele byte buscado na memória. Existem várias maneiras de fazer isto, e a maneira mais comum é através de uma tabela de saltos. Isto é exemplificado no seguinte código:

```
1      void emular_processador(int &ciclos)
2      {
3          unsigned int instrucao = busca_instrucao();
4          unsigned int parametro;
5
6          switch(instrucao)
7          {
8              case 0x01:
9                  ciclos = 3;
10                 /* emular instrução 0x01 */
11                 break;
12             case 0x02:
13                 parametro = busca_parametro();
14                 ciclos = 7;
```

48 *int*, *short* e *long* são três tamanhos de variáveis inteiras usadas pela linguagem de programação C, sendo respectivamente os tamanhos médio, curto e longo.

```
15                  /* emular instrução 0x01 */
16                  break;
17              ...
18              case 0x..N:
19                  /* emular instrução 0x..N */
20                  break;
21              default:
22                  instrucao_ilegal();
23          }
24      }
```

Neste caso, o próximo byte de memória é buscado na linha 3. Na linha 3, a instrução *switch* determina que o fluxo do código deve saltar para a instrução que foi buscada da memória. Em cada instrução, a variável ciclos é alimentada, de modo que os outros dispositivos saibam quantos ciclos foram gastos com a instrução. Caso a instrução não esteja na lista, é uma instrução ilegal e deve ser tratada de acordo (com a reinicialização do computador, por exemplo, ou uma mensagem ao usuário).

Na linha 13 é ainda apresentado o exemplo de uma instrução que usa um parâmetro. Neste caso, o parâmetros deve ser buscado da memória e o apontador de instruções avançado em mais um byte (DEL BARRIO, 2001).

Existem ainda outras formas de implementação. Uma delas é através de blocos se-então, como no exemplo abaixo:

```
1       unsigned int instrucao = busca_instrucao();
2       if (instrucao == 0x01)
3       {
4           /* executa instrução 0x01 */
5       }
6       else if (instrucao == 0x02)
7       {
8           /* executa instrução 0x02 */
9       }
10      ...
11      else if (instrucao == 0x0..N)
12      {
13          /* executa instrução 0x0..N */
14      }
15      else
16          instrucao_ilegal();
```

O tipo de implementação a ser adotado vai depender do compilador usado. Alguns compiladores farão com que o programa produzido pelo primeiro código seja mais lento, devido ao grande número de saltos realizados, enquanto outros compiladores farão com que o segundo código seja mais lento, devido ao número de comparações envolvidas (DEL BARRIO, 2001).

Outra opção é o uso de um *array* (do inglês, "vetor") de funções para a execução das instruções. O grande problema deste método é que o desempenho pode ser drasticamente reduzido devido à geração de *overhead*[49] (será discutido com mais profundidade na Seção 3.7.1) (DEL BARRIO, 2001).

3.2.3 Emulação das Instruções

A implementação da emulação de cada uma das instruções normalmente é bastante simples mas, ainda assim, seu desempenho é chave para um bom desempenho do emulador. Os diferentes processadores costumam conter um conjunto relativamente padronizado de operações bastante simples (DEL BARRIO, 2001).

A tarefa de desenvolver a emulação de cada uma das instruções consiste em duas partes: (i) buscar a descrição da operação no manual de instruções ou *website* do fabricante (geralmente escrita em pseudocódigo[50]) e (ii) desenvolver a emulação da instrução no emulador.

Por exemplo, o manual de programação dos processadores da Intel (INTEL, 2006, p. 78) diz que a instrução equivalente ao mnemônico ADD é a seguinte:

```
DEST <- DEST + SRC;
```

Isto significa que a implementação em C seria, considerando-se que a variável oper1 e oper2 guardam os dois operandos:

```
oper1 += oper2;
```

A maior parte das instruções coloca valores nos *flags*, dependendo do resultado da operação. Para não implementar este código vez após vez, é útil o uso de macros[51]. Imaginando-se que a operação acima faz uso da *flag* de *carry*, que é setada toda vez que o resultado é maior que 255, a macro poderia ser a seguinte:

```
1      #define SETA_CARRY(x)
2          if(x > 255)
3          {
4              C = 1;
5              x = (unsigned char)x;
6          } else
7              C = 0;
```

No código acima, cada vez que o resultado é maior que 255, a flag de *carry* (C) é seta-

49 Excesso de instruções de máquina geradas pelo compilador no início e no final de cada função.
50 Linguagem de programação inexistente, escrita apenas de modo a ser compreendida por um programador.
51 Macros são identificadores que são substituído por blocos de código previamente especificados.

da, e o resultado é arredondado em bits para não estourar o tamanho de 255. Assim, a emulação da instrução seria:

```
oper1 = SETA_CARRY(oper1 + oper2);
```

Na maioria dos casos, as CPUs terão mais de uma instrução para executar a mesma operação. Por exemplo, uma determinada CPU pode ter uma instrução de subtração entre dois registradores, entre um registrador e uma posição de memória, e entre um registrador e um valor passado por parâmetro. Neste caso, torna-se útil o uso de macros.

Levando em consideração o exemplo acima, o código de um emulador poderia ser o seguinte:

```
1     unsigned int A;  /* acumulador */
2     unsigned int X;  /* registrador geral */
3     unsigned int Z;  /* flag de zeramento */
4
5     #define SUBTRAI(x,y) \
6         if((x - y) != 0)  \
7             Z = 1;       \
8         else             \
9             Z = 0;       \
10        A = (x - y);
11
12    void emula_cpu()
13    {
14        unsigned int parametro;
15        unsigned int instrução = busca_instrucao();
16
17        switch(instrucao)
18        {
19            ...
20            case 0xA0: /* SUB A,X */
21                SUBTRAI(A, X);
22                break;
23            case 0xA1: /* SUB A,mem */
24                parametro = busca_parametro();
25                SUBTRAI(A, ram[parametro]);
26                break;
27            case 0xA2: /* SUB A,val */
28                parametro = busca_parametro();
29                SUBTRAI(A, parametro);
30                break;
31            ...
32        }
33    }
```

No caso apresentado, as linhas de 5 a 10 contém uma macro que subtrai dois valores

armazenando o resultado em um acumulador, e colocando na *flag* de zeramento se o resultado foi zero ou não. Na tabela de saltos aparecem as três formas de chamada da função de subtração, conforme exemplificado acima. Nestes casos, é melhor usar uma macro do que uma função, pois com uma macro não há formação de *overhead* (ver seção 3.7.1).

Depois do emulador da CPU estar completamente pronto, é interessante procurar descobrir, através do uso de um *profiler*[52], quais são as instruções mais utilizadas e otimizá-las. No exemplo do videogame Nintendo, o seu processador (um MOS 6502) possui em torno de uma centena de instruções. Mas segundo Blargg (2006), a instrução LDA (carrega acumulador, do inglês *Load Accumulator*) é usada 11% das vezes, e a instrução BNE (tomar caminho se igual, do inglês *Branch if Not Equal*), 10% das vezes. Otimizar estas instruções é muito mais importante que otimizar instruções como a instrução CLC (limpar *carry*, do inglês *Clear Carry*), que é chamada apenas 1% das vezes.

3.2.4 Interrupções

Segundo discutido na Seção 2.1.4, as interrupções podem ser de três tipos: (i) de hardware; (ii) de software e (iii) exceções. As interrupções de software são causadas pela chamada a uma instrução específica, e as exceções são causadas por algum erro na execução de uma instrução (uma divisão por zero, por exemplo). Ambas acontecem internamente na CPU e, por isso, são implementadas da mesma forma que uma instrução normal.

As interrupções de hardware são causadas por dispositivos periféricos à CPU, e isto implica que o emulador deve prover mecanismos para que haja a comunicação entre estes dispositivos, o que pode ser feito através de uma função não-estática[53] colocada dentro do emulador da CPU.

É importante notar que a tarefa da CPU é apenas receber as interrupções de hardware, e não gerá-las. Estas interrupções são geradas pelos dispositivos periféricos, e a programação do emulador deve refletir este fato (DEL BARRIO, 2001).

3.3 Memória Endereçável

O acesso à memória diz respeito à toda memória endereçável, e não só à memória

52 Programa que analisa a performance de outro programa, identificando quais são as funções mais executadas, e o tempo de execução de cada uma.

53 Em C, as funções estáticas são visíveis apenas dentro do arquivo onde elas foram postas, enquanto as funções não-estáticas (ou externas) são visíveis a todo o programa.

RAM ou mesmo à memória física. De um modo geral, o acesso à memória é bastante simples, mas cresce em complexidade se são usados mapas de memória, espelhamento ou bancos de memória.

O primeiro passo é a inicialização a memória, que em C pode ser feito através da seguinte instrução:

```
char* memoria = malloc(64 * 1024);
```

Neste caso, uma memória de 64 kB (64 x 1024) está sendo inicializada. Para escrever dados na memória, o seguinte comando pode ser usado:

```
memoria[0x1234] = 0xAB;
```

Neste caso, o dado 0xAB está sendo escrito na posição 0x1234. É interessante colocar barreiras para impedir que o código acesse dados fora da área da memória (o que poderia causar uma falha de segmentação[54]), embora raramente este problema acontecerá com programas que foram escritos originalmente para a arquitetura que está sendo emulada.

A leitura de dados também é bastante simples. Para ler o dado da posição 0x1234 e transferí-lo para a variável "x", o seguinte comando pode ser usado:

```
x = memoria[0x1234];
```

O acesso à memória seria bastante simples, se não houvessem complicadores como mapas, bancos e espelhamento de memória. Na implementação de qualquer um destes casos, o seguinte conceito deve ser sempre levado em consideração – **a memória é muito mais freqüentemente lida do que escrita**. Portanto, o ideal é que a maior parte da complexidade seja posta na escrita, e não na leitura (FAYZULLIN, 1997).

3.3.1 Memória ROM

Na maior parte dos emuladores, é desnecessário fazer uma proteção contra a escrita de áreas de memória ROM, porque um software raramente tentará escrever neste tipo de área.

Fayzullin (1997, p. 6) coloca uma exceção:

> Alguns sofwares disponibilizados em cartuchos (como jogos de MSX, por exemplo) tentam escrever em sua própria memória ROM e se recusam a funcionar se a escrita for feita com sucesso. Isto é feito para proteção contra cópias.

Nestes casos, é necessário desenvolver um mecanismo que proteja estas áreas contra a

54 Erro que acontece quando um programa tenta acessar dados fora da sua área designada, fazendo com que o sistema operacional aborte o programa. É conhecido como operação ilegal nos sistemas Windows.

escrita.

3.3.2 Mapas de Memória

Segundo o que foi discutido na Seção 2.2.2, os mapas de memória contém as regiões de memória que são compartilhadas entre a CPU e algum outro dispositivo, permitindo que o programa se comunique com os dispositivos através destas regiões.

Isto é implementado no emulador através de uma tabela de estruturas como o seguinte:

```
1      typedef struct {
2          unsigned int inicio;
3          unsigned int fim;
4          (void*) funcao_escrita_memoria;
5      } MAPA_DE_MEMORIA;
```

Os campos "inicio" e "fim" contém o início e fim da área de memória, respectivamente. O campo "funcao_escrita_memoria" conterá um ponteiro para uma função, que deverá ser preenchido com o endereço da função correspondente ao acesso de memória em cada dispositivo.

Considerando um arquitetura que possuísse o mapa de memória apresentado na Figura 8, o preenchimento da estrutura de área de memória ficaria assim:

```
1      MAPA_DE_MEMORIA area[3] = {
2         {   0x0, 0x1FF, escrita_memoria_video },
3         { 0x200, 0x2FF, escrita_memoria_som },
4         { 0x300, 0x4FF, escrita_memoria_rede }
5      }
```

Neste caso, as funções "escrita_memoria_video", "escrita_memoria_som" e "escrita_memoria_rede" estariam implementadas nos módulos de vídeo, memória e rede, respectivamente. A implementação da função de escrita de dados na memória seria a seguinte:

```
1      void escreve_dados(int posicao, unsigned char dado)
2      {
3        int i;
4        for(i=0; i<numero_de_regioes; i++)
5          if(posicao>=area[i].inicio && posicao<=area[i].fim)
6            if(area[i].funcao_escrita_memoria(posicao,dado))
7              memoria[i] = dado;
8      }
```

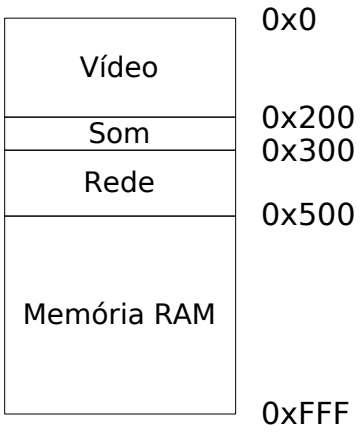

Figura 8 - Exemplo de mapa de memória de uma arquitetura hipotética.

Das linhas 4 a 8, é feito um laço para determinar se a posição está em alguma área de memória e chamar a função correspondente. É importante notar a linha 8: a função de chamada de cada dispositivo vai retornar se o dado deve ser escrito na memória ou não. Em alguns casos, o dado é guardado na memória do dispositivo para posterior acesso. Em outros, o dado escrito causa alguma mudança do dispositivo mas não deve ser armazenado. Isto acontece em alguns dispositivos onde um mesmo endereço é usado tanto para a execução de uma tarefa através da escrita, e do retorno de status completamente diferente através da leitura.

É importante lembrar que, sempre que possível, é necessário deixar a maior parte da complexidade para a rotina de escrita de memória, que é executada muito menos freqüentemente que a rotina de leitura.

3.3.3 Espelhamento de Memória

Como a rotina de escrita de memória é executada com muito mais freqüência do que a de leitura, geralmente vale a pena escrever os dados repetidamente em todos os espelhos de memória do que procurar os endereços dos espelhos de memória no momento da leitura. Isto significa que se os endereços de 0x100 a 0x1FF fossem espelhados em 0x200 a 0x2FF, um dado escrito em 0x100 faria com que o emulador escrevesse também o dado em 0x200. Embora esta abordagem consuma mais memória, vale a pena em termos de desempenho.

Na prática, os espelhos de memória vão ser implementados de forma semelhante aos mapas de memória (descritos na Seção 3.3.2), onde uma área de espelhamento será equivalente a uma área do mapa de memória, chamando uma função que execute a operação de espelhamento dos dados.

3.3.4 Bancos de Memória

O acesso a dados dos bancos de memória excessivamente lento para ler ou escrever de forma semelhante às áreas do mapa de memória. Além disso, os bancos de memória geralmente possuem tamanho igual, o que facilita o processo de escrita e leitura.

O código de implementação que possuísse três bancos de memória (de 0x0 a 0x3FFF, de 0x4000 a 0x7FFF e de 0x8000 a 0xBFFF) seria o seguinte:

```
1      unsigned int bancos[] = { 0x0, 0x4000, 0x8000 };
2      int banco_atual = 0;
3
4      void escreve_dados(int posicao, unsigned char dado)
5      {
6          memoria[posicao + bancos[banco_atual]] = dado;
7      }
```

No exemplo acima, a simples modificação da variável "banco_atual" seria suficiente para fazer com que todo acesso à memória fosse mudado para outro banco.

3.4 Sistema de Vídeo

O sistema de vídeo é a parte mais complexa na construção do emulador, e também a que exige mais recursos da emulação. Como na emulação o adaptador gráfico e a saída de vídeo estão intrinsicamente ligados, eles precisam ser construídos como se fossem um único bloco.

A primeira coisa que é importante definir é qual será a precisão da emulação do vídeo. Existem diferentes graus de precisão, e quanto menor a precisão maior a velocidade. Assim, o programador precisa conhecer bem não apenas a arquitetura computacional de destino, mas também quais técnicas que os programas escritos para aquela plataforma utilizam. Assim é possível saber se uma alta precisão é necessária ou se é um desperdício de recursos.

O nível mais alto de precisão é a emulação **por pixel**. Isso significa que à medida que as operações forem sendo realizadas, os pixeis estarão sendo desenhados na tela. O Atari 2600, por exemplo, desenha 3 pixeis na tela a cada ciclo de clock do seu microprocessador. Um emulador que usa esta técnica de emulação precisa desenhar os pixeis na tela tão logo a operação seja realizada (ISRAEL, 2004).

Um segundo nível de precisão é a emulação **por linha**. Neste caso, cada linha horizontal é desenhada por inteira, tão logo o feixe de elétrons tenha chegado ao seu final. E o terceiro nível de precisão é a emulação **por tela**, onde a tela inteira é desenhada de uma vez só.

Esta diferença existe devido ao fato de muitos programas mudarem parâmetros de exibição no meio da composição da imagem. Um exemplo disto pode ser visto na Figura 9, que mostra no lado esquerdo um jogo hipotético no estilo Pong[55] sendo executado em uma arquitetura que usa gráficos sincronizados, e na direita os comandos enviados pelo software para o dispositivo gráfico.

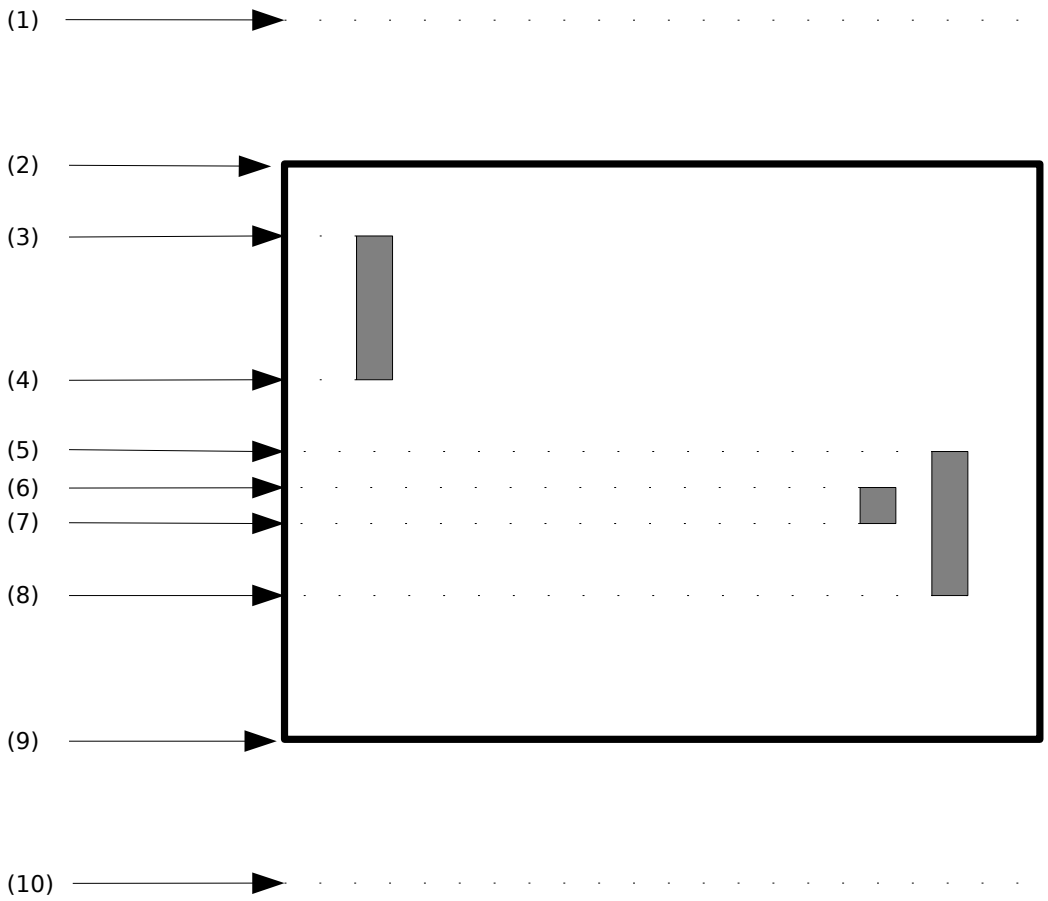

Figura 9 - Montagem de uma imagem em um computador que usa gráficos sincronizados.

Neste exemplo, durante a execução, programa envia os seguintes comandos à placa de vídeo, uma vez a cada exibição de imagem (segundo a numeração da figura):

1. Ativar o VBLANK;
2. desativar o VBLANK (iniciar a composição da imagem);
3. iniciar o desenho do jogador 1;
4. finalizar o desenho do jogador 1;
5. iniciar o desenho do jogador 2;

55 Tipo de jogo onde existe uma bola e duas "raquetes", uma de cada lado, e cujo objetivo é não deixar a bola passar pela raquete. A Figura 9 mostra um exemplo deste tipo de jogo.

6. iniciar o desenho da bola;
7. finalizar o desenho da bola;
8. finalizar o desenho do jogador 2;
9. ativar OVERSCAN (parar a composição da imagem);
10. executar a sincronização vertical (VSYNC).

Todos estes comandos são executados pelo programa, que durante toda a composição da imagem fica dedicado a montar a imagem, ficando livre para execução de outros processos apenas durante o tempo de VBLANK, OVERSCAN e sincronização vertical.

Existem outros tipos de gráficos (*framebuffers*, por exemplo) onde, ainda que seja possível a utilização da técnica de mudar os parâmetros do vídeo durante a imagem, isto não é utilizado, e uma emulação tão precisa consumiria preciosos recursos de tempo que poderiam ser melhor usados em outras áreas.

Nos sistemas onde esta técnica é utilizada com pouca freqüência, é possível ainda utilizar a emulação por tela e guardar estas informações enquanto a tela é construída, para depois usá-las no momento da montagem da tela.

Para a geração da imagem do vídeo no computador onde o emulador está sendo executado, geralmente é utilizada uma biblioteca gráfica 2D (ou 3D para gráficos tridimensionais), que permita que os dados de exibição possam ser facilmente alterados. Existe uma série de bibliotecas com este propósito, sendo que entre as mais proeminentes estão o Allegro[56], o SDL[57] e o DirectDraw[58].

3.4.1 *Framebuffer* e Terminais de Texto

Os *framebuffers* e os terminais de texto são os tipos de adaptadores gráficos mais fáceis de serem emulados. Como cada pixel (ou caractere) é definido no momento em que um dado é colocado na memória, basta colocar uma função que mapeie aquela área de memória, segundo já foi descrito na Seção 3.3.2. O tipo de emulação para este tipo de adaptador pode ser a emulação por tela, uma vez que raramente os gráficos serão modificados durante a exibição da

56 Biblioteca multiplataforma de código aberto voltada para a construção de jogos possuindo uma série de funções que auxiliam na criação de gráficos, som, entrada e temporização.
57 Biblioteca multiplataforma de código aberto que cria uma abstração sobre os gráficos de um sistema operacional. Do inglês *Simple DirectMedia Layer*, ou Camada Símples de Mídia Direta.
58 Biblioteca criada pela Microsoft semelhante ao SDL, mas de código fechado e roda apenas em plataformas Windows.

imagem (DEL BARRIO, 2001).

Os *framebuffers* geram uma **interrupção vertical** no momento que o feixe de elétrons termina de desenhar a última linha da imagem. No momento da interrupção vertical o emulador pode redesenhar a tela. A interrupção vertical também pode ser usada para auxiliar na temporização, segundo descrito na Seção 3.6 (DEL BARRIO, 2001).

3.4.2 Gráficos Sincronizados

Nos adaptadores gráficos do tipo sincronizado, é importante manter a perfeita sincronia, caso contrário a tela do monitor exibirá informações incorretas. Em geral, estes adaptadores gráficos não geram uma interrupção vertical e, portanto, não é possível sincronizar um emulador deste tipo que tenha saído de sincronia. Por isso, a construção de emuladores deste tipo é bastante complexo e requer mais testes do que os outros tipos de emuladores (DEL BARRIO, 2001).

O funcionamento deste tipo de adaptador aparece na Figura 9, na página 55. O desenho da imagem precisa ir acompanhando a execução do código, e o código irá ativar e desativar certos parâmetros da placa de vídeo quando o feixe de elétrons estiver em uma determinada posição exata. Qualquer erro de sincronia fará com que alguma imagem da tela seja mostrada numa posição incorreta, tornando o software impreciso ou inutilizável.

Os softwares mais avançados escritos para este tipo de sistema, além de ativarem e desativarem parâmetros em uma determinada posição vertical, o fazem também em posições horizontais. Isto geralmente é feito como forma de extrair mais do sistema do que ele originalmente oferece. Exemplo disto são alguns jogos escritos para o Atari 2600. O Atari 2600 permite a exibição de apenas 2 jogadores na tela. Alguns jogos burlam este limite ao ativar e desativar o registrador que ativa os jogadores mais de uma vez durante a varredura horizontal, mostrando assim mais de 2 jogadores de uma só vez na tela (DAVIE, 2003).

Por causa de todas estas complexidades, este tipo de adaptador gráfico geralmente é implementado por pixel. Se a implementação por pixel for lenta demais, é possível guardar as modificações feitas na linha (ou a linha toda em um *array*) e implementar o emulador por linha.

Outro problema que ainda deve ser levado em consideração é o fato de que não só o emulador compõe a sua imagem, mas o computador onde está sendo executado o emulador também. Este computador e o emulador podem possuir taxas diferentes de atualização, fazendo com que no momento que o computador fizer seu retraço vertical, o monitor que está sendo

emulado esteja composto pela metade, causando assim uma impressão de *page tearing*[59] ao usuário.

Este problema pode ser resolvido através de um *buffer*[60] de imagem. A imagem que está sendo montada não é exibida automaticamente (como acontece num monitor real) mas sim armazenada em um *buffer*. No momento em que o monitor compõe a sua imagem, o *buffer* é atualizado para a tela, dando assim ao usuário a impressão de uma atualização suave da tela.

3.4.3 Adaptadores Gráficos de Tiles e Sprites

Os adaptadores gráficos de *tiles* e *sprites*, discutidos na Seção 2.3.3, são bastante poderosos, e por essa razão bastante complexos de serem implementados. Geralmente são mais usados em videogames do que em computadores, mas há exceções.

A parte mais básica dos gráficos é o plano de fundo. O plano de fundo é composto de uma grade de informações, cada com um ponteiro para um *tile* da tabela de *tiles*. Por exemplo, se a tela tiver 160x160 pixeis, e cada *tile* for de 10x10 pixeis, a grade do plano de fundo terá o tamanho de 16x16. Existem outras informações armazenadas nesta tabela, como a prioridade (se o *tile* deverá aparecer na frente dos *sprites*) e a rotação do *tile*. O plano de fundo geralmente possui um mecanismo que permite a rolagem suave da tela, movendo todos os *tiles* um determinado número de pixeis para um dos lados (DEL BARRIO, 2001).

Os *sprites* são semelhantes aos *tiles*, mas mais poderosos. Eles podem ser movidos livremente pela tela, e possuem um fundo transparente. Na maior parte dos adaptadores gráficos, é possível realizar ainda outras operações com os *sprites*, como rotação, esticamento ou zoom. Os *sprites* possuem diferentes prioridades, e como são móveis, é bastante comum que eles se sobreponham durante a execução do software. Nestes casos, o emulador deve lidar corretamente com a prioridade de cada um, inclusive com a possibilidade de alguns *sprites* terem prioridade menor que determinados *tiles* (DEL BARRIO, 2001).

Este tipo de adaptador geralmente faz detecção de colisão, isto é, verifica se dois determinados *sprites* estão sobrepostos (ou se determinados *sprite* e *tile* estão sobrepostos), armazenando num registrador esta informação, de modo a facilitar a construção do programa e otimizar o seu processamento (DEL BARRIO, 2001).

---

59 *Page tearing* (ou "rasgamento de imagem") acontece quando os gráficos de um computador são atualizados durante a composição da imagem no monitor, fazendo com que a metade da tela apareça com uma imagem e a metade com outra. Como isto acontece muito rapidamente, o usuário tem a impressão de *flicker* em parte da tela.

60 *Buffer* é uma região de memória utilizada para armazenar dados temporários.

O tipo de emulação usado para estes adaptadores gráficosdependerá do tipo de software desenvolvido para a arquitetura. De um modo geral, o emulação por tela pode ser utilizada. Existem alguns programas, no entanto, que mudam parâmetros durante a composição da imagem e, portanto, exigem que o método utilizado seja a emulação por linha. Geralmente não é necessário uma precisão tão grande paraque seja necessário usar a emulação por pixel.

3.4.4 Sistemas Mistos

Embora a separação apresentada acima sirva como guia, nem sempre ela é assim tão precisa. Em algumas arquiteturas, algumas características de um tipo de adaptador se encontram presentes em outro. Um exemplo disto é o adaptador gráfico TIA, encontrado no Atari 2600. Este é um adaptador gráfico sincronizado, mas faz uso simplificado de *tiles* e *sprites* e detecta colisões (WRIGHT, 1979).

Outros tipos de adaptadores possuem registradores que permitem mudar o sistema gráfico. O sistema VGA e seus compatíveis (utilizados nos computadores PC atuais), por exemplo, permitem com que o modo de exibição seja mudado para modo texto (terminal de texto) ou modo gráfico (*framebuffer*) através da modificação de um registrador (NEAL, 1997).

3.5 Sistemas de Entrada

Os sistemas de entrada são bastante particulares a cada sistema, mas geralmente uma entrada resulta em alguma modificação na memória (nos casos de *polling*) ou na geração de alguma interrupção. Em todo caso, sempre será necessário que o emulador faça uma tradução, recebendo a tecla que foi digitada no computador onde o emulador está sendo executado e gerando a ação correspondente no emulador.

Em geral, não é necessário que esta rotina ocupe muito tempo de processamento. Embora os computadores estejam sempre prontos para receber uma entrada, o emulador pode verificar o pressionamento de alguma tecla apenas algumas vezes por segundo, pois é altamente improvável que o usuário pressione e solte a tecla em menos de uma fração de segundo.

3.6 Temporização

Uma das coisas mais importante que deve ser levada em consideração na construção de um emulador é o *feeling* (do inglês “sensação”), ou seja, a sensação do usuário estar executando a máquina real. Para isto, é necessário que a velocidade de execução do emulador seja o

mais próximo possível do hardware original (nem mais lento, nem mais rápido). Além disso, o emulador deve rodar de forma suave, isto é, não deve haver momentos onde a execução fica mais rápida ou mais lenta (ao menos não de forma perceptível ao usuário) (DEL BARRIO, 2001).

A tendência é que um código rode mais rápido numa máquina mais veloz, e mais lentamente numa máquina com menos recursos. É importante que isto não aconteça com um emulador pois, como o objetivo é imitar o computador original da forma mais semelhante possível, ele precisa rodar à mesma velocidade em qualquer computador.

A rotina de temporização é colocada em conjunto com a atualização do vídeo, e funciona da seguinte forma:

1. a emulação de todos os dispositivos é realizada sucessivamente, até que o feixe de elétrons tenha terminado de compor a última linha da tela, sendo que as modificações feitas no vídeo são armazenadas em um *buffer*;

2. o emulador congela a operação e aguarda até que tenha passado o tempo correspondente à composição da imagem na máquina real (por exemplo, se a freqüência do vídeo for de 50 Hz, isso significa que o emulador deve aguardar até que 1/50 segundo tenha passado);

3. a tela é atualizada a partir do *buffer*, e a operação 1 repetida, assim sucessivamente.

Para que o emulador não perca tempo com a atualização do *buffer*, é necessário começar a contagem do tempo antes do passo 3.

A Figura 10 mostra este processo em diferentes computadores. O computador 1 é o mais rápido de todos, e ele gasta a menor parte de seu tempo fazendo a emulação, e a maior parte de seu tempo aguardando. Isto é o correto para esta situação, e qualquer coisa diferente disto faria com que o emulador rodasse mais rápido que o desejado. O computador 2 é mais lento que o 1, e o 3 é mais lento que o 2, por isso eles gastam mais tempo emulando e menos tempo aguardando. O computador 4 é ainda mais lento e não consegue fazer toda a emulação a tempo. Neste caso, o código precisa ser otimizado, ou a execução ficará mais lenta que o computador original, conforme será esclarecido na próxima seção.

Este tempo de espera não será visualmente perceptível ao usuário. No computador original, a menor mudança visual que o usuário conseguiria perceber seria a varredura vertical, que

acontece dezenas de vezes por segundo, e que é mesma mudança visual que ele terá no emulador.

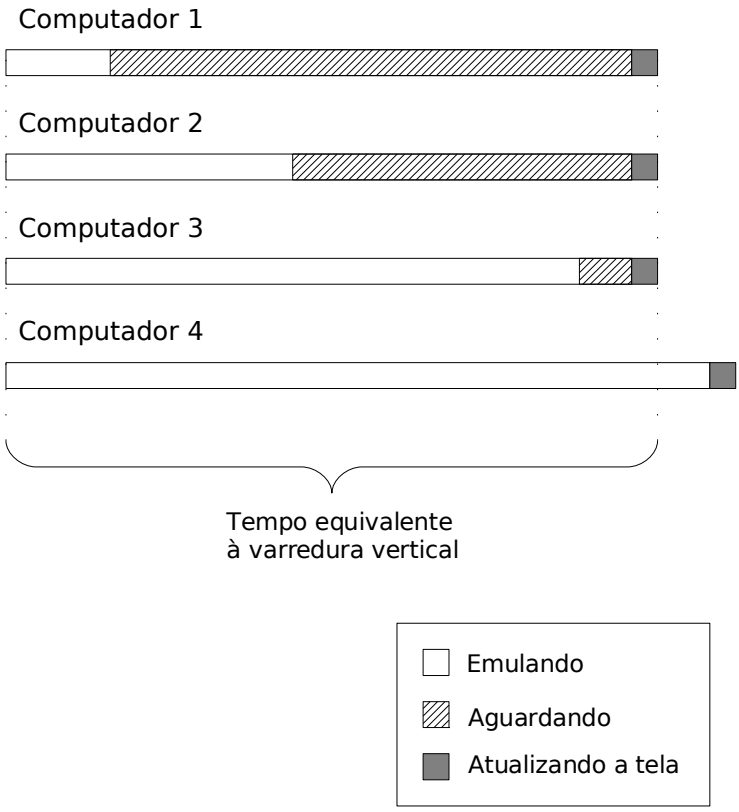

Figura 10 - Temporização em diferentes computadores.

3.7 Otimização

Num emulador, cada parte do código é executada milhares de vezes por segundo. Isto significa que o código deve ser o mais otimizado possível.

Manter a mesma velocidade do hardware original é tarefa desafiadora. Um emulador precisa executar milhares de instruções por segundo, e por isso o código precisa ser o mais otimizado possível. O uso de um *profiler* pode auxiliar na tarefa de encontrar quais linhas de código são executadas com maior freqüência, permitindo ao programador otimizá-las (FAYZULLIN, 1997).

Há uma grande diferença de performance entre a execução de programas escritos em diferentes linguagens de programação. É claro que a velocidade de execução vai depender do quão bem escrito é o código mas, em geral, programas escritos em assembly costumam ser mais rápidos que os demais, devido à proximidade da máquina (uma vez que estão se emulando pro-

cessadores, quanto mais controle tiverem do processador, melhor). A grande desvantagem do assembly é a falta de portabilidade, uma vez que um programa escrito em assembly não poderá ser executado em outro processador (e dificilmente em outro sistema operacional que use o mesmo processador).

A maioria dos emuladores utiliza assembly ou C. O uso de C é considerado a melhor opção para a escrita de emuladores portáveis, devido à sua proximidade ao hardware e capacidade de manipulação direta de memória. Alguns emuladores utilizam o C++[61], fazendo uso das funcionalidades de orientação a objeto, embora haja perda de performance. Existem ainda emuladores que usam linguagens de execução mais lenta, como Java[62], C#[63] e até Visual Basic[64], embora estes sejam exceções.

No caso do C, diversas otimizações podem ser feitas. O compilador geralmente possui opções para otimização do código gerado, e estas otimizações devem ser ativadas. Existem certas técnicas de otimização específicas para emulação de processadores, memória, placas gráficas, etc, já discutidas neste capítulo.

3.7.1 Overhead

Um dos maiores problemas de performance nos emuladores é o *overhead*. *Overhead* é o excesso de instruções de máquina geradas, de forma invisível ao programador, no início e no final de cada função. Estas instruções são colocadas lá pelo compilador para executar a troca de contexto, ou seja, salvar os registradores, o endereço da pilha e outros dados para que o processador entre "limpo" na nova função, segundo ilustrado na Figura 11. Do lado esquerdo, há uma função simples em C que retorna a soma de dois números. No lado direto, está o código assembly gerado pelo compilador (no caso, o gcc[65]). Pode-se ver que o compilador gerou 6 instruções assembly para esta função. Duas delas preparam o processador para entrar na função (*overhead* superior), duas executam a operação e duas preparam o processador para retornar à função principal (*overhead* inferior).

---

61 Linguagem de programação desenvolvida por Bjarne Stroustrup em 1983, baseada em C e acrescentando funcionalidade de orientação a objeto a ela.

62 Linguagem de programação desenvolvida pela Sun Microsystems no começo dos anos 90, orientada a objetos e voltada para execução em uma máquina virtual, permitindo assim uma portabilidade completa.

63 Linguagem de programação desenvolvida pela Microsoft, baseada em C++ e no Java.

64 Linguagem de programação desenvolvida pela Microsoft, baseada no Basic e ligada a uma ferramenta de desenvolvimento rápido própria.

65 Conjunto de compiladores de código aberto produzido pelo Projeto GNU e distribuído pela Free Software Foundation. Do inglês *GNU Compiler Collection*, que significa Coleção de Compiladores GNU.

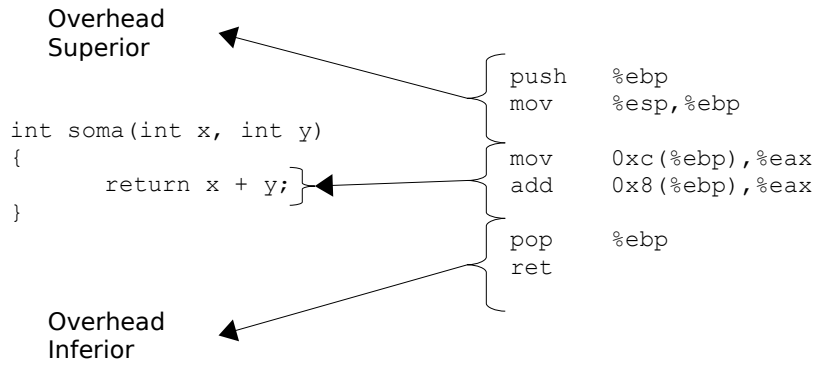

Figura 11 - *Overhead* em uma função C.

Na maioria dos programas, isto não é um problema, pois as funções são longas e não são chamadas com tanta freqüência. Num emulador, entretanto, o *overhead* é um problema sério, pois as funções são chamadas milhares (ou até milhões) de vezes por segundo. No exemplo acima, dois terços das instruções são gastos em *overhead*. Se tal tempo for perdido com o *overhead* na emulação, o emulador pode sofrer de séria perda de performance.

Uma das soluções propostas é a programação diretamente em assembly. Neste caso, o programador tem o controle de quanto *overhead* é realmente necessário, e pode se livrar daquilo que não é necessário. O grande problema da programação assembly, como já foi citado antes, é a falta de portabilidade.

Outra solução proposta é uso de funções *inline*. Neste caso, ao invés do compilador transformar uma função C em uma subrotina assembly, ele substitui a chamada da função pela própria função. Neste caso não há *overhead*, pois para o programa é como se o código da função chamada tivesse sido inserido dentro da função chamadora. A desvantagem deste método é que, se não for usado com cuidado, pode fazer com que o programa cresça demais, especialmente no caso de funções *inline* que são chamadas de vários lugares no programa (KETTLEWELL, 2003)

3.8 Considerações Finais

Neste Capítulo foram descritas algumas técnicas usadas para a construção de emuladores. Como o hardware pode ser radicalmente diferente de uma arquitetura para outra, o trabalho de construção de um emulador envolverá extensiva pesquisa e, provavelmente, o desenvolvimento de novos métodos que se encaixem melhor com a arquitetura proposta. Ainda assim, as orientações aqui propostas servem como regra geral.

Existe uma série de problemas e dificuldades que o programador encontrará durante a implementação de seus emuladores. O Capítulo seguinte descreve as mais comuns.

4 PROBLEMAS E DIFICULDADES NA CONSTRUÇÃO DE EMULADORES

Segundo descrito nos Capítulos anteriores, o desenvolvimento de um emulador exige uma ótima compreensão da arquitetura do hardware que se deseja emular. Além disso, o programador pode esperar encontrar uma série de outros problemas e dificuldades em seu trabalho. Este capítulo descreve algumas destas dificuldades, baseadas na experiência do autor deste trabalho em desenvolver emuladores.

4.1 Dificuldade de Reaproveitamento de Módulos

A modularização existente na arquitetura de hardware permite que componentes de uma arquitetura possam ser reaproveitados em outra. Por exemplo, os computadores Amstrad CPC, Colecovision, Commodore 128, MSX, Sinclair ZX-80, entre outros, utilizam o mesmo microprocessador, o Zilog Z80 (KROMEKE, 2006a). E os computadores Apple I, II e III, Atari 800, Commodre PET, entre outros, também utilizam o mesmo microprocessador, o MOS 6502 (KROMEKE, 2006b).

Não apenas os microprocessadores são reutilizados. O adaptador gráfico TMS9918 é usado nos computadores MSX, ColecoVision e TI-99. Versões modificadas deste chip são utilizadas nos videogames Master System, Game Gear e Mega Drive (WIKIPEDIA, 2006e).

Segundo apresentado nos Capítulos 1 e 3, o emulador tende a copiar o formato de uma arquitetura de hardware, que é modular. Geralmente, o desenvolvedor dedica um módulo para o processador, um módulo para o vídeo, e assim por diante, em partes bem definidas e separadas.

Portanto, pode parecer uma conclusão lógica imaginar que o código-fonte do módulo que emula um determinado dispositivo em um emulador possa ser facilmente reaproveitado em um emulador de uma arquitetura que utilize o mesmo dispositivo. Por exemplo, se existe um emulador de código aberto[66] que emula um MSX (que utiliza um microprocessador Zilog Z80),

66 Programas que têm o código-fonte publicamente disponível.

pode-se imaginar que a parte do código que emula o Zilog Z80 possa ser facilmente reaproveitado no emulador de um Colecovision, que utiliza o mesmo processador. Isto, porém, é bastante difícil na prática, devido a uma série de fatores.

Possivelmente, a principal razão deste problema seja a falta de uma metodologia na construção de emuladores. A bibliografia a respeito deste assunto é extremamente escassa, levando cada programador a procurar resolver os problemas da forma que lhe parece melhor. Isto, obviamente, acaba levando a uma inconsistência entre a forma como os emuladores são desenvolvidos.

Como exemplo disto, imagine-se que um programador esteja escrevendo um emulador de um computador TI-99/4A[67]. Este computador usa o microprocessador TMS9900[68], que é o mesmo processador do minicomputador TI-990[69]. Ele usa também o adaptador gráfico TMS9918[70], que é o mesmo adaptador gráfico de um MSX. Para tanto, ele obtém acesso ao código-fonte de dois emuladores, um de TI-990 e um de MSX, e obtém os módulos correspondentes. Mas ao unir os dois, existe uma grande possibilidade dele descobrir que os códigos não são compatíveis. Um dos emuladores pode fazer a temporização baseado nos ciclos da CPU, por exemplo, enquanto o outro o faz de acordo com a atualização do vídeo. Ao unir os dois, o programador descobre que os dispositivos não funcionam em sincronia, e que colocá-los em sincronia pode ser tão difícil quanto reescrever todo o módulo. A Figura 12 ilustra isso. O desenho superior mostra um cenário ideal, onde os módulos de diferentes autores se encaixam com perfeição. Já o desenho inferior mostra um cenário mais próximo da realidade, onde os módulos escritos por diferentes autores não se encaixam tão bem.

Outro problema comum é que muitas vezes o desenvolvedor de um emulador abre mão da modularização para obter um desempenho superior. Ele pode colocar parte do código destinado à atualização do vídeo dentro da CPU, de modo a acelerar o emulador. Devido a esta mistura de código, ocorre a perda da modularização, e o código escrito para um emulador não pode ser utilizado por outro emulador, a não ser com grandes modificações.

67 Computador criado pela Texas Instruments em 1981, usando um microprocessador TMS9900 e uma placa de vídeo TMS9918.
68 Microprocessador de 16 bits introduzido pela Texas Instruments em 1976.
69 Minicomputador produzido pela Texas Instruments na década de 80.
70 Processador gráfico produzido pela Texas Instruments na década de 80.

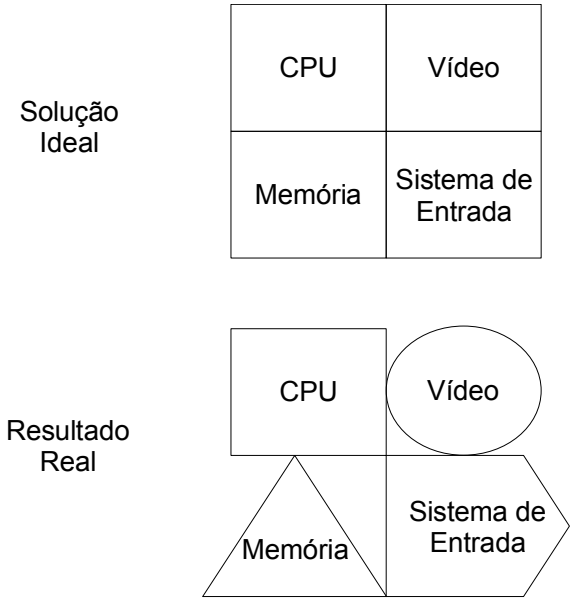

Figura 12 - Diferenças entre o ideal e o real na construção de emuladores.

4.2 Necessidade de Reimplementação das Tarefas Padrão

Existem uma série de tarefas que são realizadas por todos os emuladores, mas a falta de uma metodologia de desenvolvimento (e de ferramentas de apoio) faz com que os programadores precisem redesenvolver estas tarefas a cada novo emulador.

4.2.1 Tarefas Padrão na Plataforma de Origem

Como a arquitetura computacional é fundamentalmente semelhante em todos os computadores, a arquitetura dos emuladores também tende a ser. Exemplo disto é o fato que todos os computadores acessam algum tipo de memória. Todos os computadores recebem dados do usuário, e enviam dados para ele. Todos os computadores possuem um microprocessador que opera de forma semelhante. Embora a forma disto acontecer possa diferir um pouco de uma arquitetura para outra, estas são funções fundamentais encontradas em todas as arquiteturas.

Como um emulador imita um computador, estas também são tarefas realizadas por todos os emuladores. Mas a falta de uma biblioteca que uma estas funções faz com que estas tarefas tenham que ser reimplementadas na construção de cada novo emulador.

4.2.2 Tarefas Padrão na Plataforma de Destino

Assim como existem certas características internas de uma arquitetura que se repetem em todos as outras, existem também características específicas dos emuladores que se repetem, como a comunicação com o sistema operacional, com o objetivo de exibir de imagens e receber dados do usuário.

Segundo mencionado anteriormente, a saída de vídeo é a parte mais crítica da emulação. Isto significa que o emulador deve poder desenhar o mais rápido possível na tela, pois esta é uma das tarefas que mais tomam tempo da emulação. Existem uma série de técnicas para obter um desempenho superior na geração de gráficos, mas o programador pode não ter o conhecimento ou o tempo de implementar estas técnicas, que exigem bons conhecimentos de computação gráfica. Por exemplo, uma técnica utilizada para obter gráficos rápidos é usar a aceleração de placas 3D para gerar gráficos 2D. Mas existe uma boa possibilidade que o programador, profundo conhecedor do hardware que está implementando, não saiba como programar para placas 3D. Isto significa que seu trabalho irá custar mais tempo e, portanto, dinheiro.

Além disso, é comum existirem nos emuladores uma série de algoritmos criados para melhorar a saída de vídeo – estes algoritmos são chamados filtros gráficos. Um exemplo destes filtros é o 2xSaI[71]. Existem emuladores de computadores que possuem uma tela muito pequena (computadores portáteis e celulares, por exemplo), onde a saída de vídeo é ampliada no emulador para que o usuário possa visualizar melhor. Como os pixeis de uma imagem ampliada tendem a se tornarem grandes quadrados, filtros como o 2xSaI são usados para melhorar este tipo de imagem e arredondar os pixeis ampliados. (KIE FA, 2002).

Infelizmente, o desenvolvedor de um novo emulador é obrigado a reimplementar cada um destes filtros gráficos, se quiser que eles estejam disponíveis para os seus usuários. Ainda que muitos destes filtros sejam de código aberto, implementá-los em um novo emulador pode ser tarefa árdua e demorada.

4.2.3 Portabilidade

É sempre interessante que um emulador seja portável, isto é, que possa rodar em uma grande diversidade de sistemas operacionais e arquiteturas diferentes. Mas como o emulador faz bastante uso de tarefas específicas de cada sistema operacional (como acesso ao vídeo e às entradas do usuário), manter a portabilidade pode aumentar bastante a complexidade da programa-

71 Algoritmo de processamento de imagens digitais bidimensionais para gráficos computacionais.

ção.

4.2.4 Debuggers

Um *debugger* é um programa utilizado para identificar possíveis erros em um programa, permitindo corrigi-los. Ele possui a capacidade de executar o programa passo-a-passo, permitindo ao usuário analisar o conteúdo de variáveis e colocar pontos de parada (*breakpoints*) em posições planejadas pelo usuário. Ele pode mostrar ainda, se necessário, informações ainda mais técnicas, como o conteúdo da memória ou dos registradores do processador em um determinado momento (STALLMAN et al, 2006).

No desenvolvimento de um emulador, é indispensável que o programador escreva um *debugger* para cada dispositivo que desenvolver. Isto permitirá que ele saiba o que está acontecendo em cada momento dentro do emulador, o que está guardado na memória e nos registradores, qual instrução está sendo executada, quantos ciclos estão sendo gastos, etc. Desta forma, o programador pode testar diferentes tipos de rotinas e códigos assembly (escritos na linguagem do processador que ele está emulando), de forma a verificar se elas foram implementadas corretamente.

A Figura 13 mostra o *debugger* do emulador *Spectrum Emulator for Java*, que emula um computador ZX-Spectrum. Do lado esquero da janela aparecem as instruções que estão na memória, prontas para serem executadas pelo processador. Do lado direito, aparecem os dados contidos memória. Os números na parte inferior mostram o conteúdo dos registradores.

A tarefa de desenvolver um *debugger* é bastante custosa sendo que, em algumas ocasiões, o desenvolvimento do *debugger* pode ser até mais trabalhoso que o próprio desenvolvimento do módulo. E deve-se ainda levar em consideração o fato que o *debugger* é uma ferramenta somente para o programador, que geralmente será excluída da versão final do emulador.

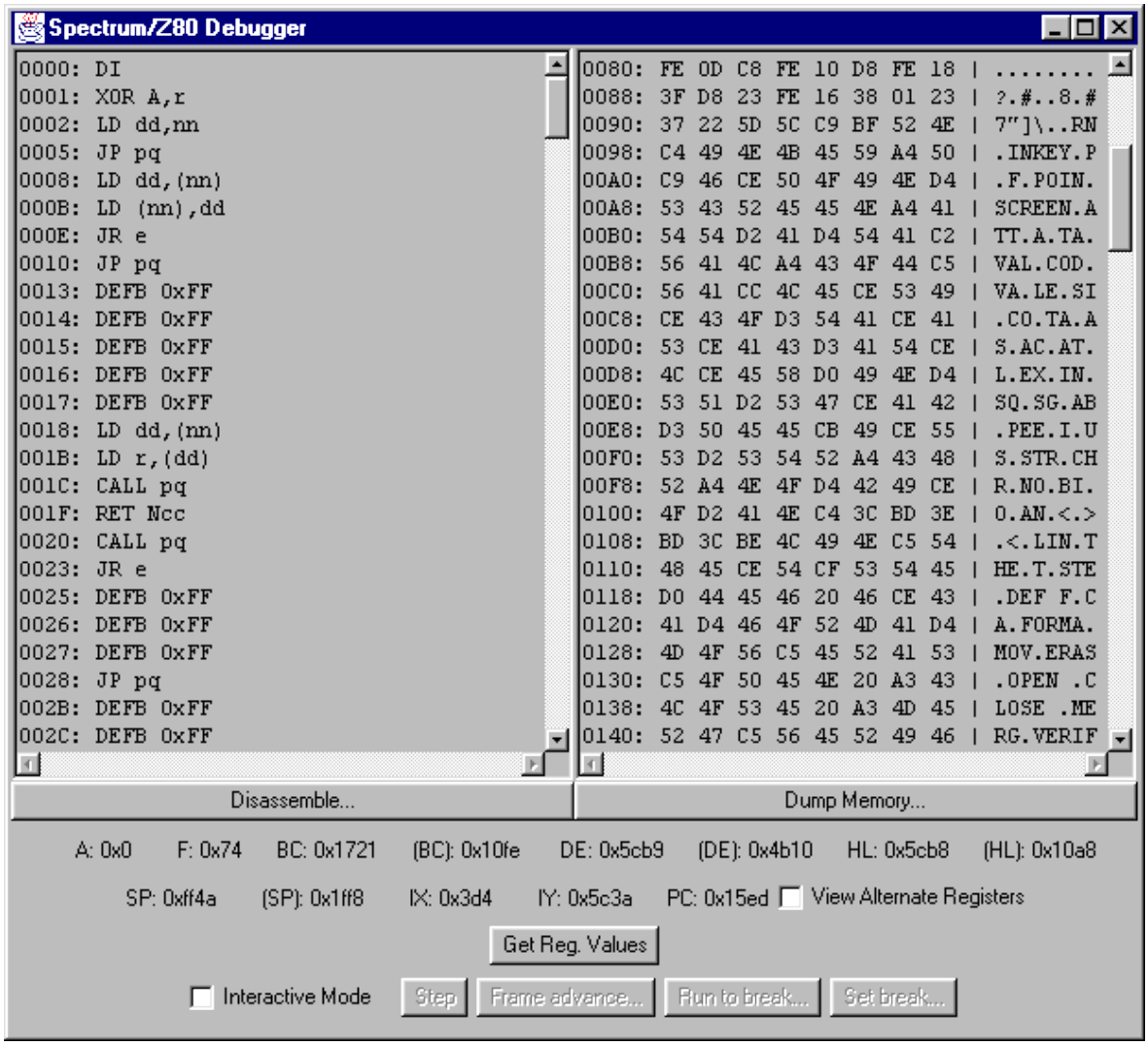

Figura 13 - Debugger do emulador Spectrum Emulator for Java.

Fonte: SUTHERLAND, 1997

4.3 Considerações Finais

Desenvolver um emulador é uma tarefa complexa, e parte da sua complexidade está nas dificuldades e problemas apresentados neste Capítulo, que encerra a primeira parte deste trabalho. Na próxima parte do trabalho, será apresentada uma nova metodologia baseada em novas ferramentas auxiliares que amenizarão estes problemas e facilitarão o desenvolvimento de novos emuladores.

CONCLUSÃO

Segundo foi apresentado no decorrer de toda a primeira parte deste trabalho, o desenvolvimento de um emulador não é uma tarefa fácil. O desenvolvedor precisa ter um grande conhecimento do funcionamento geral do hardware de um computador, e mais especificamente, um ótimo conhecimento a respeito da plataforma que deseja emular. Aliado a isso, segundo o Capítulo 4 apresentou, existem ainda uma série de outras dificuldades que o programador enfrenta no desenvolvimento de cada novo emulador, dificuldades estas que poderiam ser superadas através da criação de uma nova metodologia que tornasse o desenvolvimento mais modular, fornecendo ferramentas de apoio que a reforçassem e oferecessem uma biblioteca que unisse as tarefas padrão realizadas por todos os emuladores.

É exatamente a isto que a segunda parte deste trabalho, ainda por ser desenvolvida, se propõe – a criação de uma nova metodologia aliada ao desenvolvimento e implementação de um Kit de Desenvolvimento de Software. Este Kit de Desenvolvimento é composto por algumas ferramentas, entre as quais duas se destacam.

A primeira delas é um compilador que recebe o nome provisório de **emucc**. Esta ferramenta (montada sobre o compilador *gcc*) compilará cada um dos módulos de forma independente, transformando-os em bibliotecas de vínculo dinâmico[72] (ou DLL, do inglês *Dinamically Linked Library*). Isto permitirá que módulos possam ser distribuídos e reaproveitados, inclusive em formato binário.

A segunda ferramenta é uma biblioteca que recebe o nome provisório de **libemu**, sobre a qual os emuladores serão construídos. A biblioteca oferecerá três funções principais. A primeira será funcionar de maneira semelhante à placa-mãe de um computador moderno, permitindo a conexão, desconexão e configuração dos módulo gerados pelo compilador emucc, apresentado

72 Biblioteca que implementa o conceito de vínculo dinâmico, ou seja, que pode ser carregada ou descarregada durante a execução do programa.

acima. A segunda será oferecer uma série de funções, tanto internas ao emulador (como funções facilitando acesso à memória endereçável e aos sistemas de entrada, bem como a sincronização dos dispositivos) quanto externas (oferecendo funções de acesso ao vídeo do sistema operacional, por exemplo). A terceira tarefa da biblioteca será oferecer *debuggers*, que serão criados automaticamente pelo programa para cada um dos dispositivos.

As ferramentas permitem a criação de dispositivos de qualquer complexidade, desde microprocessadores mais simples até os atuais, mais avançados, sendo apenas limitado pela velocidade da emulação – segundo o que foi discutido no Capítulo 1, a técnica de virtualização é mais apropriada para emulação de microprocessadores modernos.

Ao final deste trabalho, um emulador será construído como forma de testar as qualidades e os defeitos da nova ferramenta, apresentar possíveis melhorias e funções que poderiam ser acrescentadas.

REFERÊNCIAS BIBLIOGRÁFICAS

ALONI, Dan. Cooperative Linux. **Proceedings of the Linux Symposium**, Ottawa, v. 1. 2004.

BLARGG. **Blargg's 6502 Emulation Notes**. 2006. Disponível em <http://www.slack.net/~ant/nes-emu/6502.html>. Acesso em 20 Out. 2006.

BORIS, Daniel. **How Do I Write An Emulator?**. 1999.. Disponível em <http://personals.ac.upc.edu/vmoya/docs/HowToDanBoris.txt>. Acesso em 28 ago. 2006.

BRITISH Computer Society. **A Glossary of Computing Terms**. 10ª ed. Boston: Addison Wesley, 2002. 390 p.

DAVIE, Andrew. **Section 15 - Playfield Continued**. 2003. Disponível em <http://www.atariage.com/forums/index.php?showtopic=28219>. Acesso em 29 Out. 2006.

DEL BARRIO, Victor Moya. **Study of the Techniques for Emulation Programming**. Barcelona: 2001. 152 p. (Tese de graduação) Facultat d'Informàtica de Barcelona. Disponível em <http://personals.ac.upc.edu/vmoya/docs/emuprog.pdf>. Acesso em 15 ago. 2006.

FAYZULLIN, Marat. **HOWTO: Writing a Computer Emulator**. 1997. Disponível em <http://fms.komkon.org/EMUL8/HOWTO.html>. Acesso em 15 Out. 2006.

FAYZULLIN, Marat. **fMSX: Portable MSX Emulator**. . Disponível em <http://fms.komkon.org/fMSX/>. Acesso em 27 ago. 2006.

GILHEANY, Steve. **Evolution of Intel Microprocessors 1971 to 2007**. 2006. Disponível em <http://www.cs.rpi.edu/~chrisc/COURSES/CSCI-4250/SPRING-2004/slides/cpu.pdf>. Acesso em 11 set. 2006.

INTEL Corporation. **IA-32 Intel Architecture Software Developer's Manual:** Volume 2A: Instruction Set Reference, A-M. 2006. Disponível em <ftp://download.intel.com/design/Pentium4/manuals/25366620.pdf>. Acesso em 11 set. 2006.

ISRAEL, Kirk. **2600 101: Into the Breach**. 2004. Disponível em <http://alienbill.com/2600/101/02breach.html>. Acesso em 24 set. 2006.

JACOBS, Andrew John. **6502 Instructions**. 2002. Disponível em <http://www.obelisk.demon.co.uk/6502/>. Acesso em 12 set. 2006.

KELLER, Robert M. **Computer Science:** Abstraction to Implementation. Harvey Mudd College: 1997. Disponível em <http://www.cs.hmc.edu/claremont/keller/webBook/ch01/sec01.html>. Acesso em 09 set. 2006.

KETTLEWELL, Richad. **Inline Functions in C**. 2003. Disponível em <http://www.greenend.org.uk/rjk/2003/03/inline.html>. Acesso em 29 Out. 2006.

KIE FA, Derek Liaw. **2xSaI : The advanced 2x Scale and Interpolation engine**. 2002. Disponível em <http://elektron.its.tudelft.nl/~dalikifa/>. Acesso em 30 Out. 2006.

KORTH, Martin. **Atari 2600 Specifications**. 2006. Disponível em <http://nocash.emubase.de/2k6specs.htm>.Acesso em 20 set. 2006.

KORTH, Martin. **Pan Docs (Gameboy Specifications)**. 2001. Disponível em <http://nocash.emubase.de/pandocs.htm>. Acesso em 20 set. 2006.

KROMEKE, Michael. **Computers using 6502 CPUs**. 2006b. Disponível em <http://www.zianet.com/kromeke/pastcomp/misc/cpu6502.htm>. Acesso em 30 Out. 2006.

KROMEKE, Michael. **Computer using Z80 CPUs**. 2006a. Disponível em <http://www.zianet.com/kromeke/pastcomp/misc/cpu_z80.htm>. Acesso em 30 Out. 2006.

LAUREANO, Marcos. **Máquinas Virtuais e Emuladores**. 1ª ed. São Paulo: Novatec, 2006. 184 p.

LINDHOLM, Tim; YELLIN, Frank. **The Java Virtual Machine**. 2ª ed. Boston: Addison Wesley, 1999. 473 p.

MICROSOFT Corporation. **Memory parity errors: Causes and suggestions**. 2006. Disponível em <http://support.microsoft.com/?kbid=101272>. Acesso em 20 set. 2006.

MONTEIRO, Mário A. **Introdução à Organização de Computadores**. 3ª ed. Rio de Janeiro: LTC, 1996. 397 p.

MURDOCCA, Miles J. e HEURING Vincent P. **Introdução à Arquitetura de Computadores**. 1ª ed. Rio de Janeiro: Campus, 2000. 512 p.

NEAL, Joshua. **VGA/SVGA Video Programming - Attribute Controller Registers**. 1997. Disponível em <http://osdever.net./FreeVGA/vga/attrreg.htm>. Acesso em 29 Out. 2006.

OLIVEIRA, Rômulo Silva de et al. **Sistemas Operacionais**. 2ª ed. Porto Alegre: Sagra Luzzatto, 2001. 247 p.

PATTERSON, David A. e HENNESSY, John L. **Organização e Projeto de Computadores**. 3ª ed. São Paulo: Elsevier, 2005. 484 p.

STALLMAN, Richard et al. **GDB User Manual**. 2006. Disponível em <http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html>. Acesso em 31 Out. 2006.

SUTHERLAND, James. **Spectrum Emulator for Java - Users' Guide**. 1997. Disponível em <http://www.guybrush.demon.co.uk/spectrum/docs/Debug.htm>. Acesso em 31 Out. 2006.

TANENBAUM, Andrew S.. **Organização Estruturada de Computadores**. 3ª ed. Rio de Janeiro: Prentice/Hall do Brasil, 1990. 460 p.

VMWARE Inc. **Virtualization Overview: VMWare Whitepaper**. . Disponível em <http://www.vmware.com/pdf/virtualization.pdf>. Acesso em 15 ago. 2006.

WEBER, Raul Fernando. **Fundamentos de Arquitetura de Computadores**. 1ª ed. Porto Alegre: Sagra Luzzatto, 2000. 262 p.

WIKIPEDIA, The Free Encyclopedia. **Texas Instruments TMS9918**. 2006e. Disponível em <http://en.wikipedia.org/w/index.php?title=Texas_Instruments_TMS9918&oldid=81452154>. Acesso em 30 Out. 2006.

WIKIPEDIA, The Free Encyclopedia. **Framebuffer**. 2006d. Disponível em <http://en.wikipedia.org/w/index.php?title=Framebuffer&oldid=76765516>. Acesso em 14 Out. 2006.

WIKIPEDIA, the Free Encyclopedia. **Vectrex**. 2006c. Disponível em <http://en.wikipedia.org/w/index.php?title=Vectrex&oldid=76443485>. Acesso em 24 set. 2006.

WIKIPEDIA, the Free Encyclopedia. **Cathode Ray Tube**. 2006b. Disponível em <http://en.wikipedia.org/w/index.php?title=Cathode_ray_tube&oldid=75878917>. Acesso em 20 set. 2006.

WIKIPEDIA, the Free Encyclopedia. **Clock Signal**. 2006a. Disponível em <http://en.wikipedia.org/w/index.php?title=Clock_signal&oldid=72530327>. Acesso em 28 ago. 2006.

WRIGHT, Steve. **Stella Programmer's Guide**. 1979. Disponível em <http://alienbill.com/2600/101/docs/stella.html>. Acesso em 21 set. 2006.

ULTRAHLE: Technical Information. 2006. Descreve informações técnicas do emulador UltraHLE. Disponível em <http://www.emuunlim.com/UltraHLE/old/techinfo.htm>. Acesso em 27. ago 2006.