ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

CAPÍTULO 4 - COMO PODEMOS OBTER E ARMAZENAR AS INFORMAÇÕES E COMO MANIPULÁ-LAS EM MÁQUINAS CISC, RISC E SUPERESCALARES?

Fernando Cortez Sica

INICIAR

Introdução

Nessa parte dos estudos em Arquitetura e Organização de Computadores, vamos estudar armazenamento, interfaceamento com o mundo externo, e um pouco mais de processamento.

Para começar, vamos abordar a questão de armazenamento das informações na memória interna do computador. A partir disso, podemos deduzir a questão inicial: apesar de um pedaço da *cache* estar dentro do núcleo do processador, ela

pode ser considerada interna, igualmente à memória principal que está na placa-mãe? Vamos entender isso e também os conceitos e aspectos físicos destes dois tipos de memória.

Aproveitando o assunto, vamos estudar outro tipo de memória: a ROM (Memória Somente de Leitura, em inglês, *Read Only Memory*).

Com esse conhecimento sobre memórias, poderemos avançar para outro questionamento: como podemos obter dados do mundo externo, usando os dispositivos de entrada e de saída (E/S)? Essa pergunta será respondida abordando alguns pontos sobre a estruturação e sobre como podemos manipular os módulos de E/S.

Por fim, entraremos no aspecto de processamento, abordando uma dúvida comum: quais as diferenças entre os processadores RISC e CISC? A família x86 é, afinal, RISC ou CISC? Aproveitando o gancho, um outro assunto que estudaremos é em relação aos processadores superescalares. Você pode, então, ampliar a sua pergunta sobre a família x86: ela é superescalar? E o que seria superescalaridade?

Esses pontos, sobre RISC, CISC e superescalaridade, serão abordados também neste capítulo.

Com base no conteúdo que veremos a seguir, você terá condições de classificar melhor os sistemas computacionais existentes, e de solucionar problemas pela adoção de equipamentos mais específicos para a sua aplicação.

Desejamos bons estudos!

4.1 Memória interna

O sistema de memória é essencial para um sistema computacional, pelo fato de ser o responsável por prover o processador com as informações por ele demandadas (instruções e dados). Ele pode ser dividido em níveis hierárquicos, desde os registradores (nível mais alto), até a memória secundária (nível mais baixo). Quanto mais alto o nível da memória, mais rápido é seu processamento, mais cara e menor a sua capacidade de armazenamento (PATTERSON; HENNESSY, 1998).

A memória interna do computador pode ser classificada em relação ao seu tipo, volatilidade e tecnologia de fabricação.

Quanto ao tipo de memória, podemos encontrar as de acesso aleatório (RAM – *Random Access Memory*) e as ROM (*Read Only Memory*). Em relação à memória do tipo RAM, o seu acesso é não sequencial, ou seja, podemos acessar um certo conteúdo, em qualquer localização, por intermédio da passagem de um endereço. Por exemplo, é possível ingressar no conteúdo de um vetor cuja célula é endereçada pelo índice. Por sua vez, as memórias do tipo ROM são aquelas representadas por sua capacidade apenas de atender às requisições de leitura.

Veremos, nessa seção, a memória interna em sua concepção mais física. Essa memória interna será classificada, aqui, como memória dinâmica (DRAM), memória estática (SRAM) e memória apenas de leitura (ROM).

4.1.1 Memória DRAM e SRAM – RAM dinâmica e estática

Antes de começarmos a falar sobre as memórias do tipo DRAM e SRAM, vamos dar uma visão geral da interface de um módulo de memória. Basicamente, um módulo de memória poder ser representado na figura abaixo:

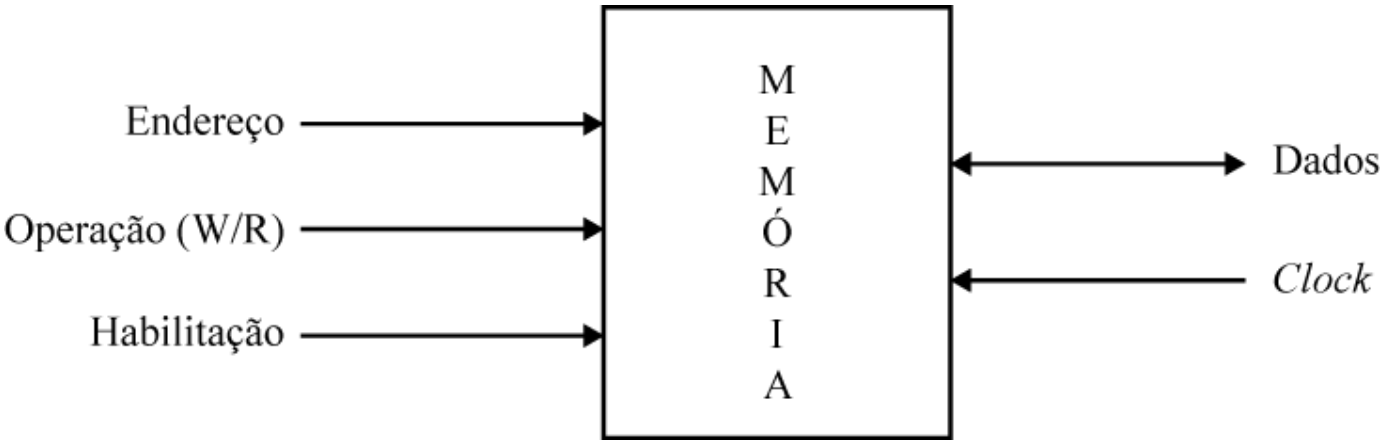


Figura 1 - Módulo básico de memória. Outros pinos poderão ser encontrados nas implementações reais de memória. Fonte: Elaborada pelo autor, 2018.

Na figura acima, temos uma possível pinagem de um módulo de memória. Nesta memória hipotética, podemos encontrar os seguintes pinos:

- **endereço**: palavra produzida pelo processador que contém a localização da memória a ser acessada;

- **operação** (W/R): define a operação a ser realizada. Neste caso, a linha sobre o caracter “W” indica que, caso esse pino receber o valor zero, será realizada uma operação de escrita (“W”: *write*). Caso contrário, ou seja, se esse pino tiver o valor um, foi demandada uma operação de leitura (“R”: *read*);
- **habilitação**: esse pino, também chamado como “*enable*”, habilita a memória a proceder a operação. Esse pino existe, pois a pinagem relativa ao endereço é ligada ao barramento compartilhado. Sendo assim, no barramento trafegarão sinais envolvendo os dispositivos e E/S. Assim, o pino *enable* (quando ativado) informa, módulo de memória, que o sinal de endereço é destinado à própria memória;
- **dados**: os pinos “dados” representam as informações que serão entregues ou coletadas da memória. Você pode notar que, ao contrário dos demais, esses pinos são bidirecionais;
- ***clock***: por fim, esse pino entrega o pulso de *clock* para a sincronização das ações realizadas pela memória para proceder a ação de leitura ou gravação das informações.

As memórias do tipo RAM podem ser diferenciadas em relação à tecnologia de fabricação e suas utilidades principais. Com os dois tipos de memória RAM, a dinâmica (DRAM), e a estática (SRAM), faremos uma pergunta de provocação: porque a DRAM é usada na memória principal e a SRAM é usada na *cache* (níveis L1 e L2, principalmente)? A resposta está relacionada com a tecnologia de fabricação.

A figura abaixo, ilustra a concepção básica de uma DRAM e de uma SRAM.

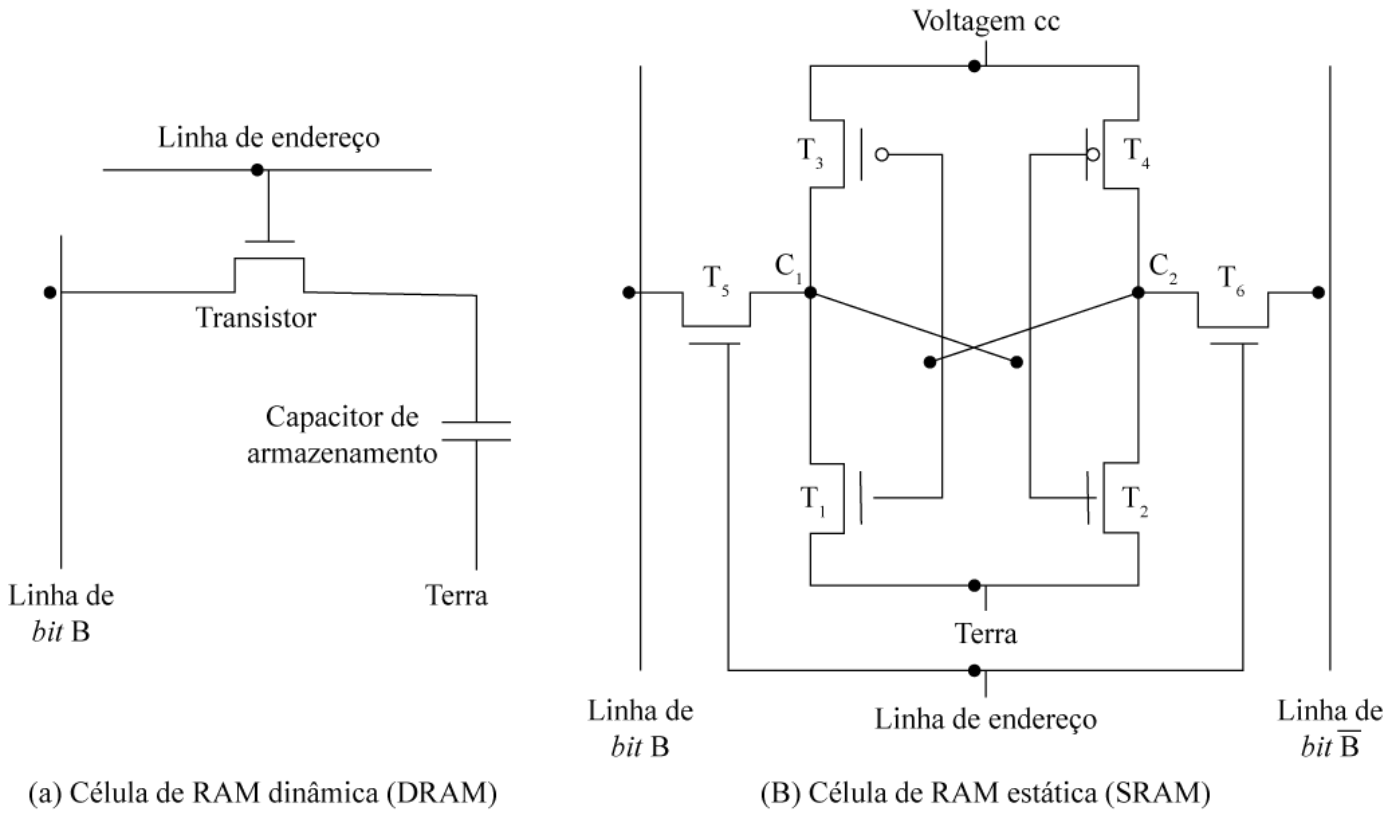


Figura 2 - Estrutura básica de uma célula de memória. Em (a), tem-se uma célula da DRAM e, em (b), uma configuração básica da SRAM. Fonte: STALLINGS, 2010, p. 130.

Na figura acima, observa-se, em (a), integrando uma célula de memória DRAM, um componente eletrônico denominado “capacitor”. Um capacitor tem a capacidade de armazenar energia temporariamente. No caso da memória, essa carga armazenada é o próprio *bit* armazenado. Porém, como mencionado, a carga do capacitor é temporária (mesmo que o circuito continue recebendo tensão de alimentação). Esse é o motivo deste tipo de memória ser chamada de dinâmica.

Em função da descarga do capacitor, antes dele perder a referência do *bit* armazenado, o circuito de memória precisa ser “relembrado” (*refresh*). De tempos em tempos, ocorre o evento de *refresh* da memória (o tempo é em função da taxa de *refresh*). Durante todo o ciclo de *refresh*, a memória fica bloqueada tanto para operações de escrita, quanto para operações de leitura. Esse bloqueio impacta, de forma negativa, diretamente na performance das operações de memória.

Por outro lado, memórias SRAM, item (b) da figura acima, a base para a sua implementação são componentes baseados em semicondutores (transistores). Ao contrário da memória DRAM, seu armazenamento ocorre de forma estática, enquanto houver alimentação suprindo-a de energia.

Em função da implementação com transistores ocupando mais espaço, a memória SRAM se torna menos densa em relação à DRAM (menos quantidade de *bits* por área). Além da densidade, podemos falar que a SRAM é mais rápida e apresenta um custo mais alto (STALLINGS, 2010).

VOCÊ QUER VER?

Você tem dúvidas sobre as diferenças entre as memórias DDR3 e DDR4 e qual a melhor escolha para a montagem dos computadores? Assista o vídeo que vai te ajudar a diferenciar as memórias: <https://www.youtube.com/watch?v=Ual7ELvPsJc (https://www.youtube.com/watch?v=Ual7ELvPsJc)>.

Atualmente, encontramos memórias DRAM do tipo SDRAM (DRAM Síncrona – *Synchronous* DRAM), equipando os computadores. Uma SDRAM é uma memória híbrida, sincronizada pelo *clock* do sistema computacional, envolvendo a DRAM e a SRAM (TANENBAUM, 2013). A grande vantagem da SDRAM, frente à DRAM é a possibilidade de eliminação de sinais de sincronização entre a memória e o processador – a sincronização é feita somente com os pulsos de *clock*. Este artifício permite que a memória opere no modo rajada (*burst*), no qual a memória pode transmitir dados com endereçamentos subsequentes, sem a intervenção do processador em enviar os próprios endereços demandados. Com isso, permite-se um melhor aproveitamento do *pipeline*, pois há a maior probabilidade de deixá-lo cheio.

VOCÊ SABIA?

Você sabia que existem memórias *flash* do tipo NOR e do tipo NAND? Para saber mais, recomendamos o especial da revista Guia do Hardware (MORIMOTO, 2007), sobre memórias: <https://www.hardware.com.br/static/media/RevistaGDH_04.pdf (https://www.hardware.com.br/static/media/RevistaGDH_04.pdf)>.

A evolução da memória SDRAM culminou nas memórias SDRAM DDR (SDRAM Taxa Dupla de Dados – SDRAM *Dual Data Rate*). Neste tipo de implementação, a memória consegue entregar dois dados a cada ciclo de *clock*, um na subida da onda de *clock*, e outro na descida.

4.1.2 Tipos de ROM e correção de erro

Um outro tipo de memória interna, que encontramos nos computadores, é a memória tipo ROM. Esse tipo de memória, não volátil, é utilizada em ocasiões nas quais requer apenas operações de leitura, como por exemplo: bibliotecas de funções, para que sejam usadas frequentemente (tal como BIOS – *Basic Input/Output System*), tabelas de funções ou informações fixas a serem demandadas pelo sistema computacional.

Nesse ponto, você pode estar perguntando: mas, se é apenas de leitura e pode conter código (instruções), como foi feita a gravação das instruções na ROM? A memória ROM pode ser classificada em relação ao momento no qual poderá ser realizada a gravação, da seguinte forma (STALLINGS, 2010):

- ROM – no tipo ROM original, as instruções eram gravadas no momento da fabricação do *chip;*
- PROM (ROM programável – *Programmable ROM*) – a escrita, neste caso, poderá ser realizada uma única vez. Para tanto, será necessário um ambiente de programação que possua uma interface física (por exemplo, usando um cabo serial) para que seja realizada a programação do dispositivo;
- EPROM (PROM apagável – *Erasable PROM*) – semelhante à PROM, porém permite várias gravações. Para se gravar um novo conjunto de informações (dados ou código), é necessário realizar a etapa de apagamento, submetendo o dispositivo a um banho de luz ultravioleta. Neste tipo de componente, existe, no invólucro, uma região translúcida para permitir que a luz ultravioleta atinja a região interna do componente;
- EEPROM (PROM apagável eletricamente – *Electrically Erasable PROM*) – igualmente à EPROM, permite o apagamento das informações para que seja feita uma nova gravação. Porém, em vez de se utilizar luz

ultravioleta, o processo de apagamento se faz mediante sinais elétricos.

VOCÊ SABIA?

Você sabia que o BIOS (Sistema Básico de Entrada e Saída – *Basic Input-Output System*) foi criado a mais de 25 anos e ainda persiste nos computadores? Existe uma proposta para substituir o BIOS, chamada de UEFI (*Unified Extensible Firmware Interface*, em português, Interface Unificada Extensível para o *Firmware*) (PACHECO, 2010). Maiores detalhes no artigo: <https://www.hardware.com.br/artigos/bios/ (https://www.hardware.com.br/artigos/bios/)>.

Quando se fala em armazenamento de informações e transmissão de dados, na maioria das vezes, temos que nos preocupar com a corretude da informação armazenada/transmitida. Corretude, neste caso, significa detectar ou, até mesmo, corrigir os dados em caso de erros.

Erros (ou falhas) podem ser permanentes ou transientes. Falhas permanentes ocorrem por falha física do componente – ocasionadas por defeitos de fabricação ou deterioração do material pelo tempo de uso. Falhas transientes também podem ser causadas pelo desgaste do material em função do tempo do uso ou, ainda, por influência de interferências externas.

Independentemente da causa ou tipo de falha, temos a necessidade de verificar se a informação que foi armazenada ou transmitida encontra-se coerente com os dados originais.

Um dos mecanismos mais simples consiste no cálculo do *bit* de paridade. Chama-se paridade par quando se adiciona, à palavra, o *bit* 1 para que a quantidade total de *bits* com valor 1 seja par (ou paridade ímpar para completar a quantidade ímpar de *bits* iguais a 1).

Por exemplo, caso a palavra a ser armazenada seja `01101101`. Como, na palavra, temos 5 *bits* iguais a 1, adiciona-se um nono *bit* com valor 1 para completar a quantidade par – neste caso, armazena-se `011011011`. Por outro lado, caso tivéssemos a palavra `10001101` (que já apresenta um número par de *bits* iguais a

um), adicionaríamos o *bit* zero, ficando 100011010. O *bit* de paridade par é conseguindo aplicando-se o operador “ou exclusivo” (XOR) em todos os *bits* da palavra.

Porém, caso a interferência atingisse dois *bits* e não somente um, o que aconteceria? Vejamos alterando-se dois *bits* do primeiro exemplo, passando de 01101101 para 01011101. . O *bit* de paridade par continuaria sendo um (010111011). Neste caso, a utilização de *bit* de paridade não é muito seguro. Dependendo do ambiente e necessidade, usa-se algum outro método de detecção de erros, com é o caso do código de Hamming.

VOCÊ O CONHECE?

Richard Wesley Hamming (1915-1998), foi um matemático nascido nos Estados Unidos, que marcou seu nome na ciência da computação, com o chamado código de Hamming. O código é usado para detecção e correção de erros, não somente frente ao armazenamento das informações mas, também, no mundo das telecomunicações. Para saber mais, leia a tradução de um texto do livro “*The Art of Doing Science and Engineering*” (HAMMING, 2016): <http://www.de.ufpe.br/~hmo/TraducaoHamming.pdf (http://www.de.ufpe.br/~hmo/TraducaoHamming.pdf)>.

O código de Hamming permite não somente a detecção da ocorrência de erros, mas, também, que os dados sejam corrigidos. A ideia central baseia-se na formação de subconjuntos com intersecções. A partir dos *bits* de paridade destas intersecções, consegue-se verificar a ocorrência e corrigir os erros ocorridos sobre a palavra de informação (STALLINGS, 2010).

4.2 Sistemas de entrada e saída

Além do processador, memória e barramento, fazem parte, também, do computador, os sistemas de entrada e de saída (E/S – I/O – *Input/Output*). Sistemas de E/S tem por objetivo interfacear com dispositivos externos, tais como vídeo, teclado e disco rígido.

Você pode estar se perguntando: como o computador consegue manipular dados provenientes de um dispositivo que opera blocos (registros), como os discos rígidos, e também de dispositivos que transmitam suas informações de forma serial, como os dispositivos ligados a uma porta USB (*Universal Serial Bus*)? Podemos responder que os dispositivos não estão ligados diretamente ao barramento e, sim, são interfaceados por um módulo de E/S, também chamado de controlador de E/S.

Nesta seção, vamos falar sobre os módulos de E/S e as suas formas de processamento.

4.2.1 Estrutura do módulo de E/S

Os módulos de E/S devem ser implementados de forma a exportar funcionalidades de interfaceamento entre o barramento e os próprios dispositivos de E/S, sob sua responsabilidade. A figura abaixo ilustra o interfaceamento básico de um módulo de E/S.

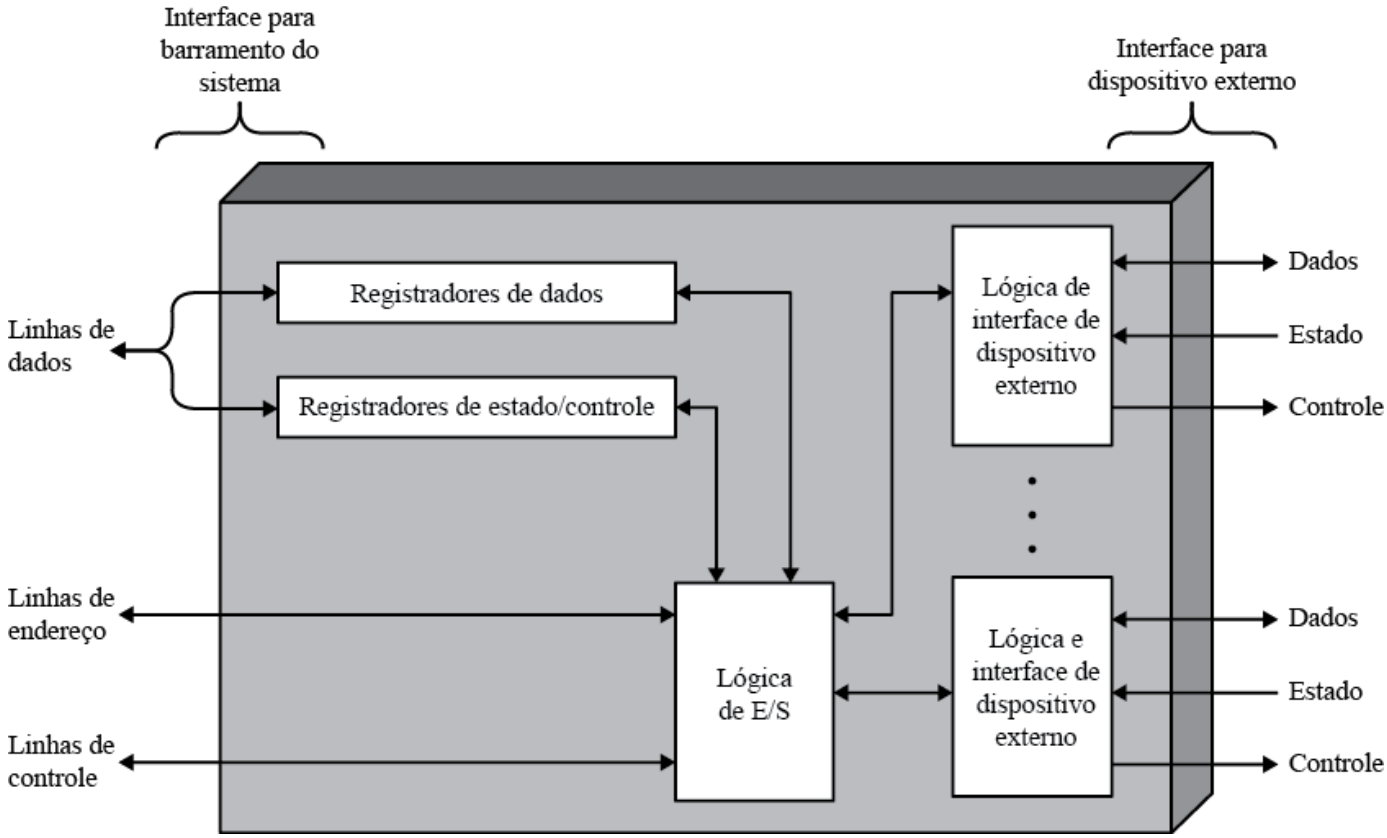


Figura 3 - Estrutura básica de um módulo de E/S, identificando seus componentes externos e suas interfaces para a interligação com o barramento. Fonte: STALLINGS, 2010, p. 130.

Na figura acima podemos ver a estrutura básica de um módulo de E/S. Nela, notamos os pinos de interfaceamento com o barramento (linhas de dados, endereço e de controle) e o interfaceamento com os dispositivos propriamente ditos. Também identificamos as linhas de endereço que transmitem a identificação do módulo de E/S gerada pelo processador. A partir de seu conteúdo, o módulo verifica se a requisição é ou não direcionada a ele. Veremos, adiante, que um módulo pode ser identificado como um endereço de memória, ou como um identificador próprio de E/S. Por sua vez, os registradores de estado/controle armazenam as condições atuais do módulo, por exemplo, se ele se encontra ocupado (*BUSY*), pronto para receber novas demandas (*READY*) ou, ainda, informar estados de erro. Por fim, as linhas de controle são responsáveis pelo envio das requisições realizadas pelo processador ou usados quando solicitadas operações envolvendo DMA (*Direct Memory Access*, em português, acesso direto à memória) – que será visto adiante.

A unidade de lógica de E/S desempenha a função de gerenciamento dos dispositivos de E/S além de ser o módulo responsável pelo recebimento das requisições (pelas linhas de controle e de endereço) e instanciações ou recebimento das informações por meio dos registradores de dados e de estado/controle.

Como funcionalidades dos módulos de E/S podemos citar (STALLINGS, 2010):

- controle e temporização: o controle e temporização são funcionalidades fundamentais para permitir a sincronização de acesso e uso dos dispositivos sob a coordenação do módulo de E/S;
- interação com o processador e com os dispositivos de E/S: a interação com o processador permite o recebimento das demandas (representadas por sinais e de palavras de controle) e, também, permite o envio ou o recebimento das informações a serem manipuladas. Essa demanda é completada pela ativação dos dispositivos de E/S para a coleta ou envio dos dados perante o mundo externo;
- detecção de erros: já que os dispositivos de E/S envolvem armazenamento e transmissão, como mencionado anteriormente, necessitam, na maior parte dos casos, de mecanismos para detecção de erros;

- *bufferização* das informações sob manipulação: a técnica de armazenar temporariamente (*bufferizar*) uma informação, permite compatibilizar as taxas de transferência entre os dispositivos e os barramentos envolvidos, uma vez que os dispositivos de E/S tendem a ser mais lentos em relação ao barramento e ao processador.

Mais especificamente em relação à funcionalidade de controle e temporização, podemos nos atentar ao fato de que o mecanismo para acesso ao dispositivo de E/S é baseado no *handshake*. Tal mecanismo é dividido em duas fases: inicialmente, o processador consulta ao módulo de E/S para verificar a disponibilidade de um dispositivo. Caso haja uma resposta positiva, então, somente neste momento, é que o processador envia um comando de E/S para efetivar a transferência das informações.

Convém salientar que a comunicação entre o processador e o módulo de E/S requer o uso do barramento. Sendo assim, deve-se, ainda, seguir a sincronização imposta pelo árbitro do barramento, para consumar a comunicação entre o processador e o módulo de E/S.

Há uma diferença entre um canal de E/S (ou processador de E/S) e um controlador de E/S (ou controlador de dispositivo). Os canais de E/S, usados normalmente em computadores de maior porte (exemplo, os *mainframes*), são módulos com um grau de independência elevado e a sua complexidade permite que eles assumam grande parte do processamento inerente à coleta ou transmissão de informações junto aos dispositivos de E/S. Os controladores de E/S, encontrados geralmente nos computadores pessoais, são módulos primitivos e requerem maior controle por parte do processador (STALLINGS, 2010).

Mas, como, efetivamente, os módulos de E/S são manipulados? A seguir, abordaremos esse assunto.

4.2.2 Instrução de E/S e processamento de E/S

Para que possamos manipular os módulos de E/S devemos, primeiro, saber o modo de manipulação do módulo em específico. Três formas são admitidas: E/S programada, E/S controlada por interrupções e DMA (Acesso Direto à Memória – *Direct Memory Access*). Nos dois primeiros modos, o processador intermedeia a

transferência de informações entre o módulo de E/S e a memória. Já, no último modo (DMA), a transferência para a memória é realizada diretamente pelo módulo de E/S.

A figura a seguir, mostra fluxogramas destes três tipos de manipulação para que, depois, possamos explanar sobre o assunto.

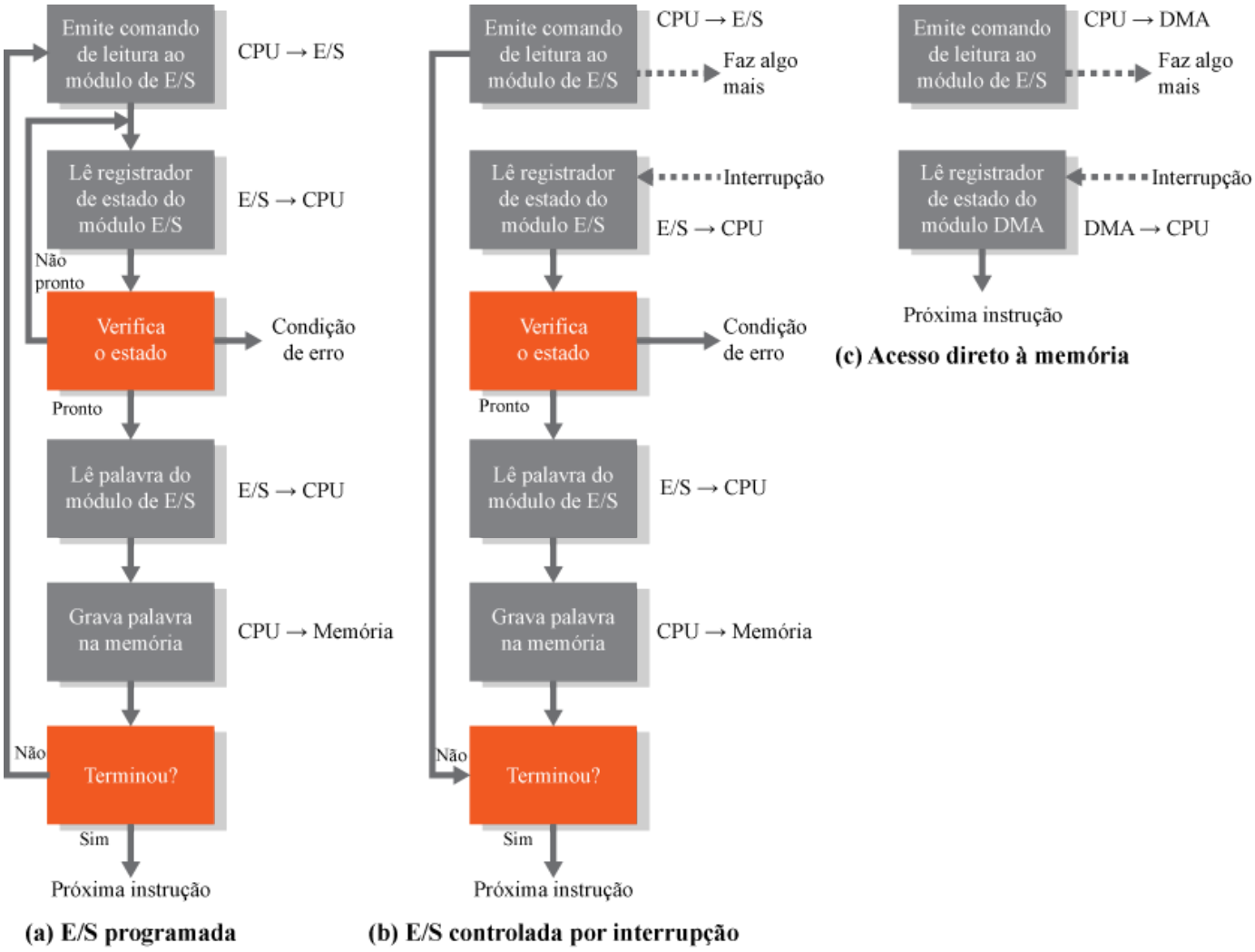


Figura 4 - Fluxogramas relacionados aos três modos de manipulação dos módulos de E/S: programada, controlada por interrupção e DMA. Fonte: STALLINGS, 2010, p. 130.

Observando a figura acima, você pode perguntar: mas, praticamente não há diferença entre o modo de E/S programada e o modo baseado em interrupção? No fluxograma existe uma sutil diferença, mas, na prática, essa sutil diferença representa uma boa economia de consumo computacional. Para sanar essa dúvida, vamos explicar cada um dos modos.

No primeiro modo, na E/S programada, há a necessidade de utilizarmos instruções de acesso ao módulo. Essas instruções variam em função do tipo de mapeamento de adotado pelo computador em questão, ou seja, existem processadores que fazem com que o módulo de E/S seja acessado como se fosse uma posição de memória enquanto outros necessitam de operações especiais tais como: `inport` e `outport` (instruções de C ANSI). Nesta última forma de acesso, os módulos são endereçados como portos de E/S.

VOCÊ QUER LER?

Às vezes, para acessar dispositivos de E/S, há a necessidade de se escrever um *device driver*. No artigo “Aprendendo a escrever *drivers* para o Linux” (SCHARDONG, 2012), você tem uma orientação inicial: <http://www.jack.eti.br/aprendendo-a-escrever-drivers-para-o-linux/ (http://www.jack.eti.br/aprendendo-a-escrever-drivers-para-o-linux/)>.

Porém, independentemente do tipo de acesso (mapeamento como memória ou porto de E/S), há a necessidade de se verificar, constantemente, o estado do módulo até que ele se encontre pronto para ser operado. Traduzindo essa afirmação em linha de código teremos: "`while(módulo_ocupado);`"

Nota-se que o laço de repetição é vazio (sem nenhuma instrução a ser realizada). Isso denota a “espera ocupada”, ocupando recurso computacional sem realizar um processamento efetivo.

Para evitar a espera ocupada, pode-se adotar o modelo no qual o módulo de E/S é controlado por interrupção. Neste tipo de modelo, a programação consiste em ativar uma interrupção para que seja acessado o módulo de E/S requerido. A requisição é enfileirada pelo processador de interrupções que a atenderá assim que o dispositivo estiver pronto.

Como essas operações envolvem apenas sinais, não há a necessidade de ter um trecho de código que teste continuamente o estado de dispositivo ocupado. Porém, os dois modos descritos têm o inconveniente de requerer, continuamente,

a intervenção do processador para proceder a transferência de informações entre a memória e o módulo de E/S. Essa necessidade é eliminada quando é utilizado o modo DMA.

No modo DMA, retira-se a responsabilidade de gerenciar a transferência das informações entre o módulo de E/S e a memória. Neste caso, há somente a intervenção do processador para configurar o controlador de DMA e, depois, somente quando recebe, do controlador de DMA, o sinal indicando a finalização da transferência. A figura a seguir ilustra um módulo básico de DMA.

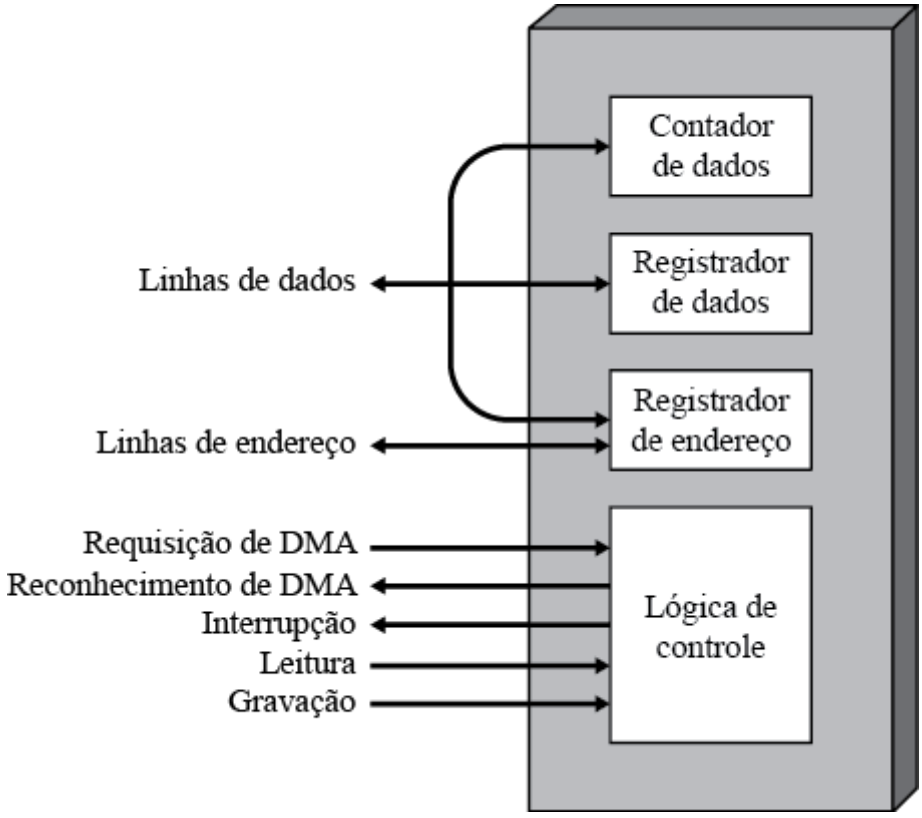


Figura 5 - Módulo básico de DMA.

Nota-se a presença das linhas de dados, de endereço e de controle (divididas em linhas específicas para a lógica de controle). Fonte: STALLINGS, 2010, p. 192.

Na figura acima, temos as linhas de dados sendo utilizadas em dois momentos distintos: na configuração do DMA e na transferência efetiva das informações. Na etapa de configuração, o processador envia, para o DMA, a quantidade de *bytes* a ser transferida, o endereço inicial da fonte das informações e do destino na memória principal. Após este momento, durante o decurso da transferência, as vias servirão para o transporte efetivo das informações (os dados são disponibilizados pelo DMA pelo registrador de dados).

As linhas de endereço serão utilizadas, também, no momento da transferência, de modo que o sistema de memória possa receber a localização na qual escreverá os dados transferidos.

Em relação aos sinais de controle, destacam-se aqueles relacionados à interrupção, pois o DMA também é baseado na utilização de interrupções para o seu funcionamento e interfaceamento com o processador. Interrupções que ocorrem durante a reivindicação do processador ao controlador do DMA e, também, no momento do término de transferência – para que o DMA possa sinalizar para o processador.

Todos esses modos e técnicas precisarão ser lembrados no momento em que você tiver a necessidade de implementar, quando necessário, métodos de acesso aos dispositivos de E/S – seja utilizando interfaces padrões como USB, FireWire e InfiniBand, ou algum outro dispositivo implementado para uma finalidade bem específica.

4.3 Evolução dos computadores RISC/CISC

Os processadores podem ser classificados em relação à arquitetura e à organização. Esses dois modos de abstração possuem um elemento em comum: o seu conjunto de instruções.

O projeto do conjunto de instruções permite definir um processador como sendo CISC (*Complex Instruction Set Computer*, em português, Computador com Conjunto de Instruções Complexas) ou RISC (*Reduced Instruction Set Computer*, em português, Computador com Conjunto de Instruções Reduzidas), o que impacta diretamente sua organização como por exemplo, a estrutura de seu *pipeline*.

Alguns pontos são primordiais na diferenciação dos computadores baseados na metodologia CISC, ou RISC. Dentre os quais, podemos mencionar, em relação ao RISC (STALLINGS, 2010):

- banco de registradores envolvendo um número maior de GPRs (*General Purpose Registers*, em português, Registradores de Propósito

Geral). A quantidade maior de registradores permite uma maior otimização de seu uso – seja pela maior probabilidade de reaproveitamento de valores previamente carregados, ou seja, pela possibilidade de usar o renomeamento de registradores em caso de *hazards* de dados;

- conjunto de instruções simples: as instruções de uma máquina RISC são simples. Sendo simples, pode-se aproveitar melhor os conceitos inerentes ao sistema de *pipeline*;
- otimização de *pipeline*: devido às próprias características das instruções, o *pipeline* pode ter sua otimização feita de forma mais agressiva;

Com base nestas afirmativas, abordaremos, a seguir, as características da execução e conceitos que cercam o modelo RISC.

4.3.1 Características da execução das instruções

Como mencionado, uma das características do modelo RISC é o fato de seu conjunto de instruções contemplar instruções mais simples (reduzidas). Para que a concepção do processador RISC fosse idealizada, alguns levantamentos foram feitos para que a ISA (*Instruction Set Architecture*, em português, Arquitetura de Conjunto de Instruções) fosse projetada. Tais levantamentos referiram-se a:

- **frequência de uso das operações**: o índice permitiu consolidar quais operações necessitariam mais otimizações e como ficaria o relacionamento do processador com o sistema de memória. O referido índice permitiu, também, estudos sobre o emprego de superescalaridade nos processadores RISC;
- **frequência de uso dos operandos**: essa métrica permitiu que o sistema de memória fosse projetado, incluindo a estimativa de tamanho e os mecanismos associados ao banco de registradores e memória *cache*;
- **sequência de execução das instruções**: a análise da sequência permitiu um estudo mais aprofundado da estrutura do *pipeline*.

Os levantamentos citados acima foram essenciais, não somente para a concepção do conjunto de instruções do processador RISC mas, também, para otimizações do *pipeline* e do sistema de memória (em relação ao banco de registradores e memória *cache*). Resumindo, podemos falar que *pipeline* e o sistema de banco de registradores são os elementos que mais contribuem com a performance dos processadores RISC.

Falando em banco de registradores, convém lembrar que, quanto mais se consegue reaproveitar as informações previamente nos registradores, melhor será o desempenho do processador pois atenua-se o número de acessos ao sistema de memória. Para permitir uma otimização, existem duas abordagens uma baseada em *software* e outra baseada em *hardware*.

Na abordagem de *software*, há a necessidade do compilador realizar uma análise do programa, de modo que os registradores sejam usados mais massivamente, armazenando as variáveis que serão mais utilizadas.

Um exemplo de abordagem de *hardware* consiste na utilização de janelas de registradores. Para explicarmos sobre janelas de registradores, vamos imaginar a estrutura de um programa diferenciando as variáveis locais e globais. Imaginemos, também, passagem de parâmetros, quando uma função evoca outra. Para continuarmos na explanação, vamos supor, ainda, que o programa possui quatro funções (procedimentos) que serão chamadas sequencialmente, ou seja, função `A` evoca a função `B`, que evoca a função `C`, e assim por diante. A figura abaixo ilustra a janela de registradores envolvendo essas quatro funções.

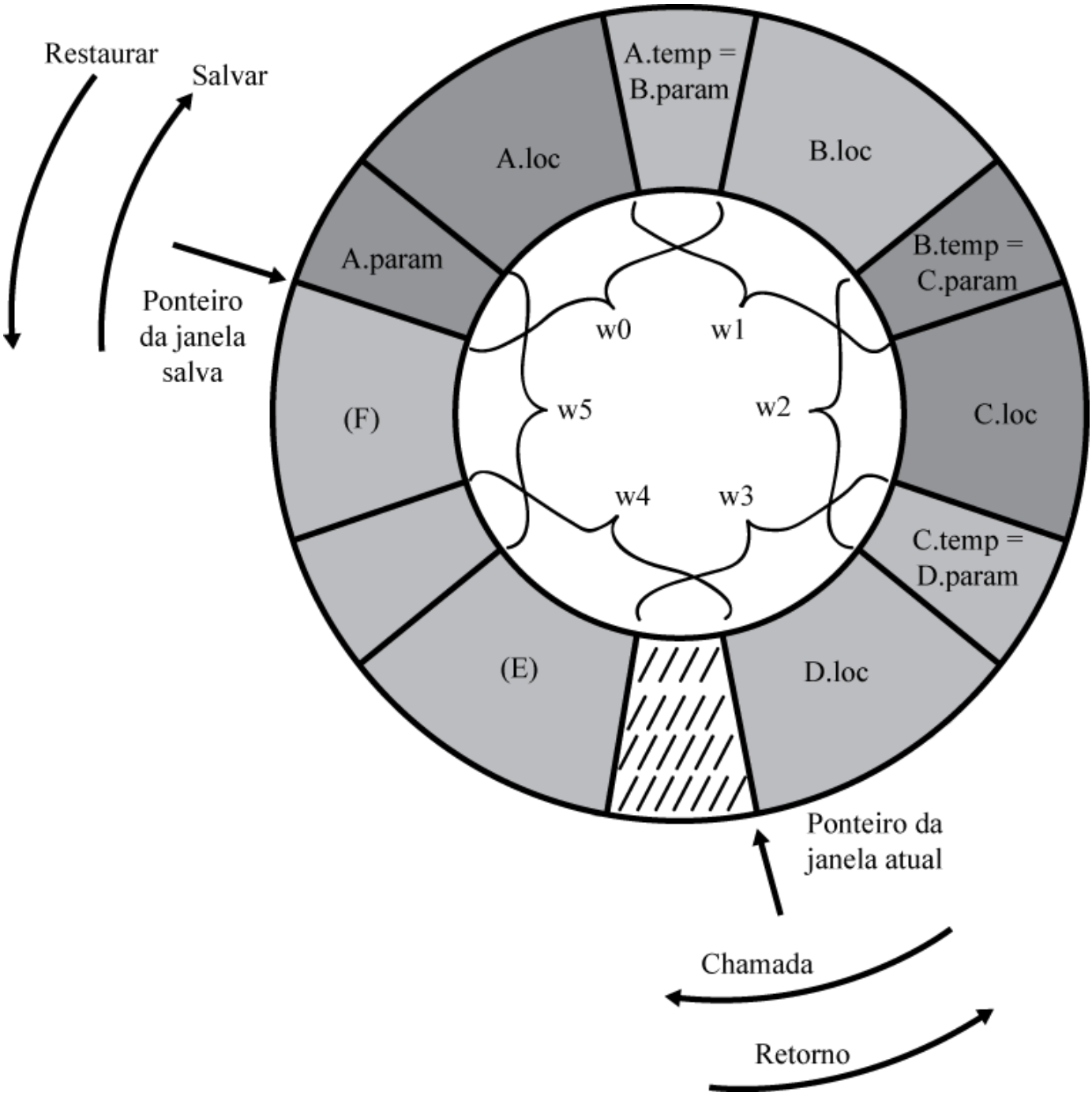


Figura 6 - Exemplo de janelas de registradores sendo usadas por quatro funções aninhadas. A função A passa parâmetros para a função B, e assim por diante. Fonte: STALLINGS, 2010, p. 402.

Na figura acima vemos a alocação de uma janela para cada função. Na medida em que as funções vão sendo chamadas, ocorre a alocação de uma janela (de forma sequencial) para que possam ser armazenadas as variáveis locais de cada função. A transição das janelas é usada para armazenar os valores relativos aos parâmetros passados de uma função para outra.

No momento do retorno, na finalização de uma função, a janela ativa volta a ser a antecessora. Dessa forma, as variáveis não são perdidas e, consequentemente, não precisam ser buscadas, novamente, junto à memória.

No caso das janelas de registradores, listamos algumas perguntas que podem surgir.

- E se tivermos muitas chamadas aninhadas de funções e chegarmos ao limite do número de janelas? O que acontece com as variáveis que estavam armazenadas, relativas às funções mais antigas? Neste caso, há o uso da memória *cache* para armazenar as variáveis mais antigas.
- O uso das janelas de registradores não é ineficiente, pois vários registradores poderão ficar ociosos? Para responder a esta pergunta, vamos comparar com a *cache*. A *cache* manipula blocos de informações. Em alguns (ou vários) momentos, será necessário manipular apenas uma variável dentro do bloco, ou seja, todas as demais informações serão carregadas “à toa”. Então, poderemos ter uma ociosidade de uso na *cache*. Na janela de registradores, a unidade de manipulação é a própria variável (e não blocos) podendo ter registradores não alocados. De uma forma ou de outra (nas janelas de registradores ou na *cache*) temos ociosidade, ou de registradores não alocados, ou de itens carregados na *cache* e não utilizados.

As janelas de registradores atuam como um *buffer* circular, no qual as janelas podem ser sobrepostas. Diante disso, a otimização de uso se faz com base nas características encontradas nas linguagens de alto nível, ou seja, o uso frequente de chamadas de procedimentos. Por conclusão, é por esse motivo que os levantamentos descritos no início dessa seção foram essenciais para o desenvolvimento da concepção da ideia que envolve os processadores RISC.

4.3.2 Instruções reduzidas

Como já mencionado, máquinas RISC tem como principal característica, o fato de possuírem, em sua implementação, instruções reduzidas. Instrução reduzida significa instrução simples, otimizada. Temos quatro características básicas de uma instrução reduzida (STALLINGS, 2010):

- o tempo de execução de uma instrução corresponde a um ciclo de máquina. Ciclo de máquina representa o tempo necessário para o acesso e coleta de informações, junto aos registradores, executar uma operação e escrever o resultado junto ao banco de registradores;
- as operações são do tipo registrador-registrador, ou seja, não existem instruções que misturem registrador-memória, exceto aquelas de carga e escritas na memória;
- os modos de endereçamento são simples, não possuindo, por exemplo, endereçamento indireto;
- utilização de formatos simples de instruções é essencial para se projetar unidades de controle mais simples, nas quais o processo de decodificação e busca dos operandos, torna-se mais ágil, em função da menor complexidade do *hardware*.

;O

e a comparação, para saber qual a melhor tecnologia: CISC ou RISC? Cada uma apresenta pontos sobre a outra. Por exemplo, processadores RISC tem uma implementação mais simples, consomem rgia e fazem uso mais eficiente de seu *pipeline* e do banco de registradores. Processadores CISC manipular, de forma mais eficiente, instruções complexas. Mas, então, por que não juntar os dois

nisso e na compatibilidade em relação a modelos anteriores, a família x86 passou a adotar, desde o o (1995), o modelo híbrido. Esse processador era um CISC dotado de um núcleo RISC. Na ocasião, o r contava com decodificadores responsáveis pela conversão de instruções complexas para sequências de simples. Com isso, introduziu-se, também, a execução fora de ordem.

, praticamente todos os processadores que equipam computadores pessoais adotam o modelo híbrido Você pode ler mais sobre as metodologias e esse hibridismo no artigo (CARVALHO, 2017): .linkedin.com/pulse/arquitetura-de-computadores-diferen%C3%A7a-entre-risc-e-carlos- (https://pt.linkedin.com/pulse/arquitetura-de-computadores-diferen%C3%A7a-entre-risc-e-ardo)>.

Todas essas características do RISC permitem que o *pipeline* também seja otimizado, assim como as interfaces internas e externas do processador. Isso favorece a simplicidade do circuito como um todo, pois são necessários, por exemplo, menos elementos de roteamento interno de informações. Como exemplo de processadores RISC, podemos citar: SPARC, MIPS R4000 e micro-RISC e ARM.

Processadores que adotam padrão RISC são mais empregados em sistemas embarcados e aplicações industriais e em tempo real, pois são mais deterministas em relação ao tempo de processamento das instruções.

4.4 Máquinas escalares e superescalares

Quando se fala em *pipeline*, uma das maiores preocupações é tentar mantê-lo cheio, para obter a máxima vazão (*throughput*) de processamento das instruções. Em máquinas escalares, que carregam a possibilidade de executar apenas uma instrução por vez, o *pipeline* pode não ser atendido plenamente, devido aos conflitos e dependências (*hazards*) de dados, estrutural, ou de controle.

Para se resolver esse problema, devemos ter mecanismos que possam antecipar instruções que estejam aptas a serem executadas, invertendo a ordem daquelas que devem esperar para que alguma dependência seja resolvida. Essa técnica é chamada execução fora de ordem e, para se conseguir, devemos adotar mecanismos que permitam o paralelismo em nível de instruções (ILP – *Instruction Level Parallelism*).

Uma das técnicas existentes para se chegar ao ILP é a adoção da superescalaridade. Superficialmente falando, superescalaridade consiste na replicação das unidades funcionais.

VOCÊ QUER LER?

No artigo “Arquiteturas Superescalares” (FREITAS, 2016), podemos entender mais sobre as principais limitações que ocorrem na execução paralela de instruções na arquitetura. Também podemos conhecer os principais processadores de mercado com arquitetura superescalar, para aprofundar nossos estudos. Leia o artigo completo: <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/029043-superescalar.pdf (http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/029043-superescalar.pdf)>.

Vamos mais a fundo em superescalaridade, para entender alguns outros aspectos relacionados ao paralelismo em nível de instruções.

4.4.1 Características

A motivação principal para o desenvolvimento de arquiteturas superescalares consiste na possibilidade de se executar instruções de forma paralela. Para tanto, são replicadas unidades funcionais, sendo que, cada uma, apresenta o seu próprio *pipeline*.

A figura abaixo, ilustra uma implementação de um processador superescalar típico.

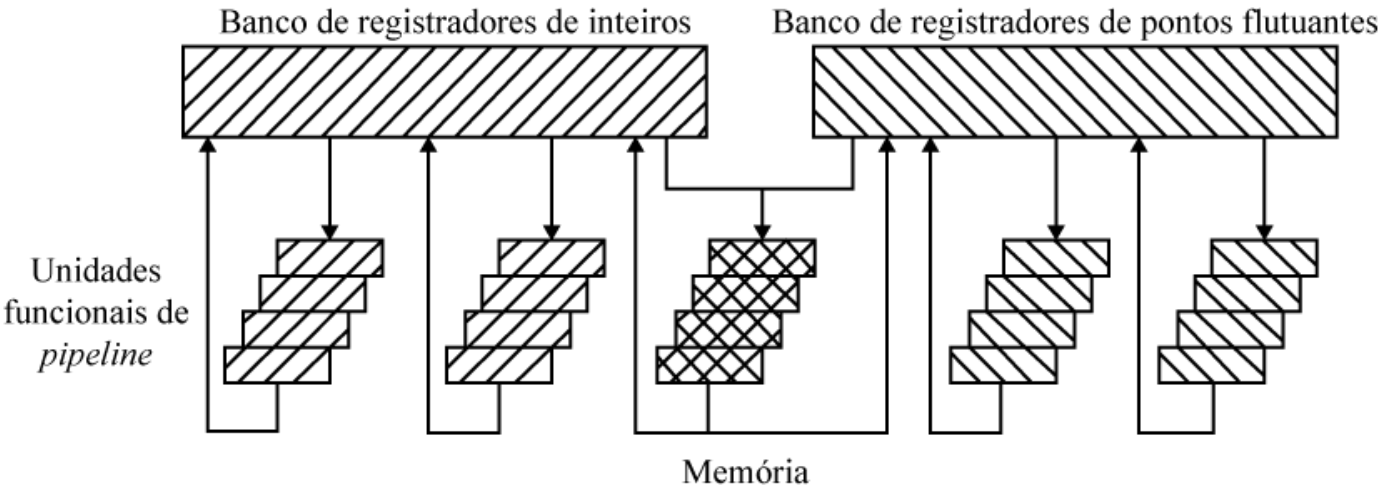


Figura 7 - Estrutura de um processador superescalar típico. Nota-se a presença de unidades funcionais replicadas. Fonte: STALLINGS, 2010, p. 430.

Como podemos perceber na figura acima, a existência de unidades funcionais replicadas dotadas de *pipeline*, faz com que haja a possibilidade de se executar instruções independentes de forma paralela, o chamado ILP.

Sabemos que o ILP permite a execução fora de ordem. Mas, detectar as dependências de forma que possa “pular” instruções impossibilitadas de serem executadas? Como fazer com que as dependências sejam eliminadas, em alguns casos?

Para responder às questões acima, vamos citar alguns mecanismos adotados pelos processadores atuais.

O primeiro mecanismo que podemos mencionar consiste na separação dos bancos de registradores. Como podemos ver na figura anterior, há a separação do banco de registradores para atender as operações que envolvem dados inteiros e as que envolvem dados de ponto flutuante. Essa abordagem já atenua a frequência de *hazards* estruturais, devido à possibilidade de dois acessos simultâneos aos registradores.

Dependências de controle podem ser atenuadas com a previsão de desvio. Ao se deparar com uma instrução de desvio, o processador tenta prever qual será a próxima instrução: a que inicia o bloco do desvio tomado, ou a que inicia o bloco do desvio não tomado? Para tentar manter a eficiência computacional, ele toma uma decisão com base no histórico do processamento. Caso a decisão seja acertada, a instrução é efetivada. Caso contrário, descarta-se os resultados obtidos pelas instruções executadas erroneamente.

Para encaminhar as instruções para as unidades funcionais, adota-se a emissão múltipla de instruções. A unidade de despacho é capaz de coletar um bloco de instruções e armazená-las em um *buffer*, chamado de janela de execução.

Após uma decodificação e análise de dependências, as instruções contidas na janela de execução são direcionadas aos módulos funcionais correspondentes. Como a emissão, nos processadores superescalares pode ser realizada fora de ordem, há a necessidade de efetivar os resultados respeitando-se a ordem original de forma que os resultados sejam ordenados.

Os resultados podem ser efetivados pela ação do evento de conclusão (*commit*) ou refutados – no caso, por exemplo, daquelas envolvidas em instruções de desvios tomados erroneamente pela previsão de desvios.

Por fim, no caso de conflitos em relação à geração de valores, pode-se adotar o renomeamento de registradores (HENNESSY; PATTERSON, 2014). Para tanto, vamos supor o seguinte código:

```
(i)   a = b / c;
(ii)  d = a - f;
(iii) a = g + h;
(iv)  i = a + j;
```

No código acima, podemos notar a presença de alguns conflitos e dependências. A linha (ii) depende o resultado obtido na linha (i). Por sua vez, a linha (iv) depende de (iii). Mas, como as instruções poderão ser despachadas simultaneamente, ou com uma defasagem mínima de tempo, devemos considerar que, como a operação da linha (i) é uma divisão, ela demorará mais tempo a ser executada em relação às demais. E se ela finalizar a produção de "*a*", logo após a geração de "*a*" pela linha (iii), qual o valor a ser usado na linha (iv): o "*a*" obtido da linha (i), ou o da linha (iii)?

Para evitar essa dúvida, usa-se, como alternativa, o renomeamento de registradores. No caso específico, troca-se o registrador "`a`", da linha (iii), e todos os subsequentes, por outro registrador. No exemplo adotado, ficaria:

```
(i)   a = b / c;
(ii)  d = a - f;
(iii) aa = g + h;
(iv)  i = aa + j;
```

Com o renomeamento, este tipo conflito é eliminado, porém aumenta-se a complexidade do processador pelo fato de ter a necessidade de possuir uma tabela que fizesse o relacionamento entre a referência original do registrador e o resultado do renomeamento.

4.4.2 Tendências

Como você pode notar, estamos em uma ascensão em relação ao uso de paralelismo nos computadores. Historicamente, tivemos como ponto de partida, a introdução do *pipeline,* que representa um pseudoparalelismo. Na sequência, houve a incorporação da superescalaridade, adicionando, de fato, o paralelismo para dentro dos processadores.

Porém, a tendência é que não fiquemos limitados ao paralelismo em nível de instrução, mas, sim, que possamos ampliar o paralelismo em nível de *threads* e de núcleos.

No paralelismo em nível de *threads*, o processador tem condições de escalonar *threads* na ocorrência de conflitos no *pipeline*. Tratando-se de um escalonamento em nível de *hardware*, torna-se muito mais eficiente, se compararmos com o escalonamento realizado pelo sistema operacional (LEME, 2016).

Por sua vez, o paralelismo em nível de núcleos, incorpora a ideia das máquinas multiprocessadores. Os vários processadores foram transformados em núcleos – cada núcleo independente, com suas estruturas de controle e memória *cache* L1.

Não devemos deixar de destacar a volta da ênfase aos processadores vetoriais. Uma arquitetura vetorial consiste em um modelo paralelo, dotado de *pipeline*, do tipo SIMD (*Single Instruction, Multiple Data*, em português, Instrução Única, Múltiplos Dados).

Neste modelo, vários dados são executados por uma única instrução. Como exemplo, podemos falar das GPUs (*Graphics Processing Unit*, em português, Unidade de Processamento Gráfico), no qual os dados a serem processados são representados por vetores e as instruções são otimizadas para esse tipo de manipulação.

Por fim, diversas são as pesquisas para explorar outras formas de paralelismo, não se limitando às abstrações arquiteturais (OLIVEIRA; CAVALCANTE, 2010). Podemos deixar, por exemplo, uma provocação para sua pesquisa e, por consequência, para sua formação acadêmica, sugerindo um estudo sobre paralelismo em nível de aplicações, por exemplo, o paralelismo em nível de dados (DLP – *Data Level Parallelism*).

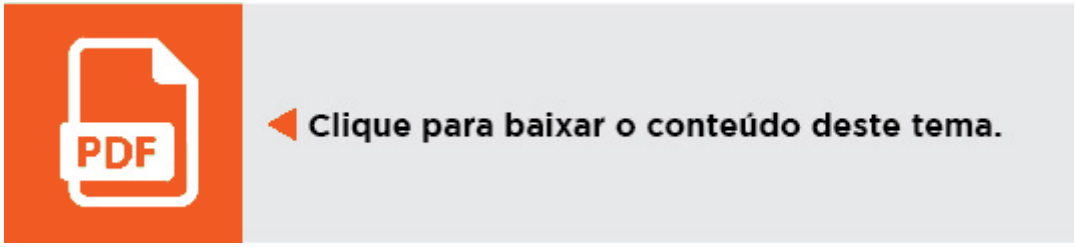
Síntese

Chegamos ao final de mais um capítulo. Esperamos que a sua trajetória tenha sido satisfatória e que o conteúdo aqui abordado tenha ampliado seus horizontes e conhecimentos. Neste capítulo, estudamos alguns aspectos físicos da memória

interna, assim como pudemos abordar aspectos inerentes aos módulos de E/S. Pudemos ver, também, aspectos relacionados à execução das instruções em máquinas RISCs e CISCs e superescalares.

Neste capítulo, você teve a oportunidade de:

- reconhecer aspectos inerentes à memória interna, no quesito de sua implementação e funcionalidades;
- analisar e comparar as estruturas e formas de manipulação dos módulos de E/S;
- discutir e identificar os modelos computacionais RISC e CISC, situando-os nos atuais modelos de processadores;
- esboçar modelos baseados na superescalaridade e em algumas outras formas de paralelismo. As abstrações contraídas poderão ser aplicadas em futuros projetos de desenvolvimento de sistemas computacionais.



Bibliografia

CARVALHO, C. E. M. **Arquitetura de Computadores – A diferença entre máquinas RISC e CISC**. Portal Linkedin, publicado em 19/03/2017. Disponível em: <https://pt.linkedin.com/pulse/arquitetura-de-computadores-diferen%C3%A7a-entre-risc-e-carlos-eduardo (https://pt.linkedin.com/pulse/arquitetura-de-computadores-diferen%C3%A7a-entre-risc-e-carlos-eduardo)>. Acesso em: 21/07/2018.

CHIPTEC. **Entenda realmente a diferença entre memória RAM DDR3 e DDR4**. Canal ChipTec, YouTube, publicado em 06/03/2018. Disponível em <https://www.youtube.com/watch?v=Ual7ELvPsJc (https://www.youtube.com/watch?v=Ual7ELvPsJc)>. Acesso em: 22/07/2018.

FREITAS, M. E.; **Arquiteturas Superescalares**. Fundação CPqD Centro de Pesquisa e Desenvolvimento em Telecomunicações. Campinas: Unicamp, 2016. Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/029043-superescalar.pdf (http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/029043-superescalar.pdf)>. Acesso em: 21/07/2018.

HAMMING, R. **Você e sua pesquisa**. Tradução de Edmar Candeia Gurjão. Universidade Federal de Pernambuco. Departamento de Estatística. Publicado em 13/06/2016. Disponível em: <http://www.de.ufpe.br/~hmo/TraducaoHamming.pdf (http://www.de.ufpe.br/~hmo/TraducaoHamming.pdf)>. Acesso em: 21/07/2018.

HENNESSY, J. L.; PATTERSON, D. A. **Arquitetura de Computadores**: uma Abordagem Quantitativa. 5. ed. Rio de Janeiro: Campus, 2014.

LEME, F.; **Processadores SMT e paralelismo em nível de *threads***. Fundação CPqD Centro de Pesquisa e Desenvolvimento em Telecomunicações. Campinas: Unicamp, 2016. Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/930886-thread.pdf (http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/930886-thread.pdf)>. Acesso em: 21/07/2018.

MORIMOTO, C. E.; Especial Memória RAM. **Revista GDH**, ano 1, n. 4. 2007. Disponível em: <https://www.hardware.com.br/static/media/RevistaGDH_04.pdf (https://www.hardware.com.br/static/media/RevistaGDH_04.pdf)>. Acesso em: 21/07/2018.

OLIVEIRA, R. G.; CAVALCANTE, S. V.; Explorando o paralelismo de dados e de *thread* para atingir a eficiência energética do ponto de vista do desenvolvedor de *software*. **Anais**. IX Workshop em Desempenho de Sistemas Computacionais e de Comunicação. Belo Horizonte, 2010. pp 1776-1789. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/wperformance/2010/002.pdf (http://www.lbd.dcc.ufmg.br/colecoes/wperformance/2010/002.pdf)>. Acesso em: 21/07/2018.

PACHECO, E.; **Era uma vez, o BIOS...** Portal Guia do Hardware.net, publicado em 17/11/2010. Disponível em: <https://www.hardware.com.br/artigos/bios/ (https://www.hardware.com.br/artigos/bios/)>. Acesso em: 21/07/2018.

PATTERSON, D. A.; HENNESSY, J. L.; **Organização e Projeto de Computadores**: a interface *hardware/software*. 2. ed. Rio de janeiro: Morgan Kaufmann Publishers, 1998.

SCHARDONG, F.; **Aprendendo a desenvolver drivers para o Linux**. Blog Pensando a tecnologia, publicado em 21/02/2012. Disponível em: <http://www.jack.eti.br/aprendendo-a-escrever-drivers-para-o-linux/ (http://www.jack.eti.br/aprendendo-a-escrever-drivers-para-o-linux/)>. Acesso em: 21/07/2018.

STALLINGS, W.; **Arquitetura e Organização de Computadores**. 8. ed. São Paulo: Pearson Prentice Hall, 2010. Disponível na Biblioteca Virtual Ânima: <https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset (https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)>. Acesso em: 21/07/2018.

STALLINGS, W.; **Arquitetura e Organização de Computadores**. 8. ed. São Paulo: Pearson Prentice Hall, 2010. Disponível na Biblioteca Virtual Ânima: <https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset (https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)>. Acesso em: 21/07/2018.