Unidade 1: Introdução à Arquitetura de Computadores

Prof. Daniel Caetano

Objetivo: Apresentar a evolução histórica dos computadores, os níveis de máquina e a importância da compreensão de diversas bases de contagem.

Bibliografia:

- MONTEIRO, M.A. **Introdução à Organização de Computadores**. 5ª. Ed. Rio de Janeiro: LTC, 2008.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.
- TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª.Ed. São Paulo: Prentice Hall, 2003.

INTRODUÇÃO

* O que é e como evoluiram os computadores?
* O que existe entre nosso programa e os sinais elétricos?
* Por que os computadores trabalham com números binários?

Atualmente é bastante incomum encontrar pessoas que jamais tenham visto ou operado um computador. Entretanto, para a maioria das pessoas o computador não passa de uma "caixa preta", isto é, um aparato "mágico" que produz resultados interessantes. O objetivo do curso de Arquitetura e Organização de Computadores é desvendar essa "caixa preta", apresentando os fundamentos que tornam possível seu funcionamento.

A compreensão dos computadores modernos sem conhecer a sua origem, entretanto, é muito mais árdua. Assim, esta primeira aula aborda a origem dos computadores e sua evolução, além de dar uma visão geral de toda a transição que ocorre entre aquilo que escrevemos na forma de um programa - o código - e aquilo que o computador realmente entende - sinais elétricos.

Finalmente, é apresentada uma breve introdução aos sistemas de numeração mais utilizados na informática, desvendando as razões pelas quais eles são utilizados.

1. EVOLUÇÃO DOS COMPUTADORES

* Equipamentos Mecânicos e Eletro-Mecânicos
* Equipamentos Eletrônicos e de Estado Sólido
    - Divisão em 6 Fases (Gerações)

A evolução dos equipamentos conhecidos hoje como "computadores" pode ser dividida duas grandes etapas: uma, inicial, envolve os equipamentos mecânicos e eletro-mecânicos. A segunda, mais recente, envolve os equipamentos eletrônicos e de estado sólido. Esta segunda etapa apresentou tantas transformações que acabou sendo dividida em diversas fases ou gerações.

1.1. Equipamentos Mecânicos e Eletro-Mecânicos

* 500 a.C. - Ábaco
* 1642 - Pascalene
* Fim do Século XIX - Hermann Hollerith cria máquina de cartões
  - Fundação da IBM

Esse universo eletrônico que hoje conhecemos como "mundo da informática" teve seu início em épocas bem mais precárias, há muitos séculos.

Tudo começou com os babilônios, por volta de 500a.C., com a invenção do ábaco, que era uma ferramenta de cálculo manual. A primeira evolução do ábaco só veio em 1642, com a invenção da Pascalene, pelo físico e matemático Blaise Pascal. A Pascalene era um equipamento mecânico capaz apenas de realizar somas e subtrações. A evolução destes dispositivos foi muito lenta e eles eram pouco usados, devido ao uso limitado e desajeitado.

Foi apenas com a invenção do motor elétrico, já no fim do século XIX, foi possível construir máquinas mecânicas muito mais complexas e "rápidas". Uma das primeiras máquinas deste tipo foi usada com o propósito de realizar a contabilização do censo dos Estados Unidos da América. Esta máquina foi projetada por Hermann Hollerith, fundador da IBM e também criador da máquina que realizava o cálculo do pagamento dos funcionários, produzindo um pequeno resumo de contabilidade que recebeu o seu nome, sendo chamado até hoje de "Olerite".

1.2. Equipamentos Eletrônicos

* Exigências militares: computadores humanos
* Segunda guerra: Enigma
  - Computadores Humanos: Insuficientes

Como os equipamentos existentes até o fim do seculo XIX não eram adequados para resolver grandes problemas, todos os problemas mais complexos precisavam ser solucionados por seres humanos.

Os militares, por exemplo, que frequentemente precisavam de soluções sistematizadas para problemas complexos - fosse para distribuir produtos ou para construir edificações -, usavam profissionais específicos da área de cálculo e lógica, que compunham uma espécie de linha de produção de soluções de problemas. Haviam os profissionais que, em tempos de guerra, estabeleciam a lógica de solução para os problemas de distribuição de armas e suprimentos; a atividade exercida por eles ficou conhecida como "Logística". Entretanto, os "logísticos" só descreviam os procedimentos de solução, que precisavam ter seus resultados computados para que pudessem ser postos em prática. Os profissionais que realizavam os cálculos eram chamados de "Computadores".

Esta organização era suficiente para todas as necessidades até que, na segunda guerra mundial, os engenheiros alemães criaram máquinas complexas de criptografia de dados, chamadas "Enigma". As forças militares aliadas, incluindo o exército norte-americano, eram capazes de captar as mensagens transmitidas pelos alemães mas, como estas estavam codificadas, não era possível compreendê-las. Percebendo que era fundamental decifrar tais mensagens para a vitória e, verificando que a decodificação pelos computadores humanos era impossível, engenheiros foram chamados para que fossem propostas máquinas capazes de decifrar as mensagens codificadas pelo Enigma.

Das pesquisas nesta área, deu-se início aos primeiros equipamentos eletrônicos de computação, substitutos dos computadores humanos.

1.2.1. Primeira Fase

* 1940 a 1955
* ENIAC: 5000 adições por segundo
  - O que isso representa hoje?
* Colossus
* Linguagem de Máquina: Wire-up
* Não existia conceito de Sistema Operacional
* Falta de Confiabilidade: executar várias vezes!
* EDVAC, IAS, UNIVAC I...

A primeira fase (ou primeira geração) dos computadores ocorreu aproximadamente durante o período entre 1940 e 1955, quando surgiram os primeiros computadores digitais usados durante a segunda guerra mundial.

O ENIAC (*Eletronic Numerical Integrator and Computer*) foi o primeiro computador de propósito geral, desenvolvido para cálculo de Balística. Era gigantesco e tinha uma capacidade de processamento em torno de 5000 adições por segundo, valor este muito inferior ao de uma calculadora atual da Hewlett-Packard. Outro computador desenvolvido à mesma época foi o Colossus, este de propósito específico: decodificar as mensagens criadas pela máquina Enigma (e posteriormente Enigma 2) dos Alemães.

Estes primeiros computadores eram baseados em válvulas e programados com o método chamado *wire-up*, isto é, conectando fisicamente, com fios, diversos polos, fechando bits de forma a construir um programa. A programação era, assim, feita diretamente em linguagem de máquina e o equipamento não oferecia qualquer recurso para facilitar este trabalho de programação.

Outros grandes computadores construídos na época incluem o EDVAC, o IAS e o UNIVAC I, construído pela IBM para a computação do censo americano de 1950.

1.3. Equipamentos de Estado Sólido

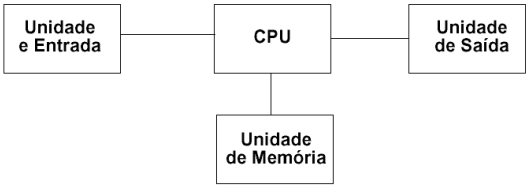
* Problemas dos equipamentos eletrônicos
* Transístores
* Modelo de Von Neumann

Os equipamentos eletrônicos produzidos na primeira fase possuíam diversas limitações. Para começar, consumiam uma quantidade monstruosa de energia. Adicionalmente, eram enormes, esquentavam demasiadamente, demoravam horas para poderem ser usados a partir do momento em que eram ligados e, para finalizar, alguns cálculos tinham de ser repetidos diversas vezes, pois a confiabilidade dos resultados não era exatamente alta.

Uma mudança radical neste cenário só foi possível com a criação dos transístores, que permitiram a eliminação das válvulas e, com elas, a maior parte dos transtornos mencionados acima. Entretanto, o transístor, por si só, não eliminava uma das principais limitações destes equipamentos: a programação física através de fios. Como a memória dos computadores eletrônicos era muito pequena, apenas os dados do processamento eram armazenados nela.

Com a possibilidade de memórias maiores - seja pelo uso de transístores ou pelas novas "fitas magnéticas", este problema foi resolvido com o conceito de *software*, isto é, um programa armazenado em memória, conforme descrito por John Von Neumann.

Na proposta de Von Neumann, todo equipamento de computação deveria possuir quatro componentes: memória, unidade de processamento, dispositivos de entrada e dispositivos de saída, conforme apresentado na figura a seguir. Adicionalmente, um certo conjunto de dados armazenado na memória deve ser interpretado pela CPU como *instruções*, eliminando a necessidade de fios para a programação do equipamento.



1.3.1. Segunda Fase

* 1955 a 1965
* Transístores => confiabilidade
* Memórias magnéticas
    - Maior velocidade e maior capacidade
* Programas armazenados em memória (Modelo de Von Neumann)
    - Funções de E/S => embrião dos Sistemas Operacionais
        + Eliminação de bugs
        + Interface padronizada para dispositivos de E/S
        + Escrita com Independência de Dispositivos
* Automação de Processo Sequenciais
    - Processamento Batch => vários programas em sequência
        + Cartão x Batch: qual a vantagem?
    - Sem intervenção
* Escrita direta entre dispositivos (DMA)

A segunda fase ocorreu aproximadamente entre 1955 e 1965, e a grande inovação era o uso de transístores, o que permitiu uma grande redução no tamanho dos equipamentos e aumento de sua velocidade, além do aumento da **confiabilidade** do processamento. Também é desta época o surgimento das memórias magnéticas, que permitiram um aumento na capacidade e velocidade do armazenamento.

Nesta fase surgiram as primeiras linguagens e compiladores e, pela primeira vez, surgiu o conceito de sistema operacional como um software para automatizar todas as tarefas repetitivas utilizadas por diversos softwares e que, até então, eram realizadas manualmente (processamento *batch*). Originalmente, sempre que se desejasse executar um programa, o programador deveria inserir este programa no equipamento (através de um cartão perfurado), este programa seria executado e finalmente o resultado seria impresso. Entretanto, em geral este processamento durava horas e era comum que se passassem horas até alguém perceber que o processamento havia finalizado. Isso fazia com que o equipamento ficasse ocioso mesmo que muitas tarefas ainda estivessem por ser executadas, já que o equipamento dependia que um ser humano fosse até ele e o alimentasse com um novo programa.

Nesta geração, então, passou a ser possível introduzir diversos programas e o "sistema operacional" existente era capaz de executá-los em seqüência, ou seja, assim que um terminava, ele iniciava o seguinte e assim por diante, eliminando o problema da ociosidade. Ainda no final desta fase a IBM criou o conceito de "canal", hoje comumente chamado de "DMA", que permitia a escrita direta entre dispositivos sem a necessidade de intervenção da CPU.

1.3.2. Terceira Fase

* 1965 a 1980
* Circuitos Integrados => redução de tamanho e custos
* IBM Série 360 / DEC PDP-8
* ...Sistemas Operacionais... Ex.: OS/360
  - Multiprogramação: multitarefa cooperativa
    + Processamento de um programa durante espera de outro
* Terminais de Impressão e de Vídeo
  - Interação Online
* PDP-7 (POSIX/UNIX e Linguagem C), Computadores Apple, CP/M

Esta fase ocorreu mais ou menos no período de 1965 a 1980, e foi marcada pela utilização dos circuitos integrados, que reduziram tamanho e custos e ampliaram enormemente a capacidade de armazenamento, processamento e confiabilidade dos computadores. Nesta época surgiu o conceito de família de processadores (IBM Série 360), em que vários equipamentos, com dispositivos diferentes conectados, eram compatíveis entre si. Também surgiram computadores de custo menor, como o PDP-8 da DEC.

Com o grande aumento de recursos destes equipamentos, os novos sistemas operacionais (como o OS/360) traziam novas possibilidades de gerência de processamento, permitindo que enquanto quando um programa esperava pela entrada de dados do usuário, outro fosse processado. Esta tecnologia ficou conhecida como *multiprogramação* e é uma técnica básica envolvida na *multitarefa cooperativa*. Nesta geração também passou a existir uma "interação online"; foram criados os primeiros terminais de vídeo e teclados para comunicação com o software *durante sua execução*.

Ainda nesta geração, a multiprogramação evoluiu de maneira a melhorar os tempos de resposta na interação com os usuários, desemvolvendo o conceito de *time-sharing*, isto é, cada processo compartilha a CPU por um intervalo de tempo. Este conceito é a base da *multitarefa preemptiva*.

Surgiu nesta fase, ainda, o sistema operacional UNIX, concebido inicialmente para o computador PDP-7, desenvolvido em Linguagem C, e tornou-se bastante conhecido por sua portabilidade. Outras grandes novidades desta época foram os computadores de 8 bits da Apple e o sistema operacional CP/M (Control Program Monitor).

Vale ressaltar que, nesta geração, houve a criação do padrão POSIX (Portable Operating System IX), que definiu uma interface mínima que sistemas UNIX devem suportar.

1.3.3. Quarta Fase

* 1980 a 1990
* Integração em Larga Escala (LSI e VLSI)
* Computadores Pessoais (no Brasil, do MSX ao IBM-PC)
  - Recursos limitados: DOS era suficiente (sem *time-sharing* etc)
* Computadores de Grande Porte
  - VMS: multitarefa monousuário
* Computadores Multiprocessados
* LANs, WANs, TCP/IP... Sistemas Operacionais de Rede

Nesta fase, que durou toda a década de 1980, a integração em larga escala (LSI e VLSI) permitiram uma redução substancial no tamanho e no preço dos equipamentos. Com isso houve o surgimento de diversos computadores menores mas muito potentes (variando desde os mais simples como o MSX até os mais poderosos IBM-PCs), ambiente no qual surgiu o DOS (Disk Operating System), base dos "computadores pessoais" do período. Estes equipamentos tinham processamento relativamente limitado e, portanto, o DOS não suportava muitas das características de multiprogramação, *time-sharing* e outros.

No campo dos computadores de grande porte, surgiu o sistema VMS (Virtual Machine System) que, implementando todos os recursos concebidos até então, criou oficialmente o conceito de *multitarefa* em um sistema monousuário.

Nesta fase surgiram os computadores capazes de *multiprocessamento*, com várias CPUs em paralelo, e os primeiros sistemas capazes de lidar com este tipo de característica também surgiram. Nesta fase houve proliferação das LANs e WANs, com o surgimento de diversos protocolos de comunicação e uma grande aceitação do protocolo TCP/IP.

Alguns autores (como Tanenbaum, 2003) não consideram a quinta e sextas fases, colocando os avanços posteriores ao da quarta fase dentro da própria quarta fase. Por questões didáticas, neste trabalho foi feita a opção pela separação.

1.3.4. Quinta Fase

* 1990 a 2000
* Enorme aumento da capacidade de processamento e armazenamento
* Multitarefa nos computadores pessoais

A quinta fase compreendeu basicamente a década de 1990, sendo a tônica principal o aumento da capacidade de processamento e armazenamento em proporções não previstas anteriormente, possibilitando aplicação de inteligência artificial, bancos de dados e mutlimídia em praticamente qualquer tipo de aplicação, tornando-as muito mais complexas.

Nesta fase os computadores se tornaram muito baratos e passaram a fazer parte da vida de praticamente todas as pessoas. O conceito de *processamento distribuído* passou a fazer parte das pesquisas e a *multitarefa* veio para os computadores pessoais.

1.3.5. Sexta Fase

* 2000 até hoje
* Rede sem fio ubíqua
* Limite físico para processamento de uma CPU
  - Multiprocessamento nos computadores pessoais
* Processamento Distribuído é comum
* Computação móvel

A sexta fase teve início juntamente com o século XXI e ainda não foi finalizada. As inovações trazidas são conhecidas pela maioria das pessoas, como a ubiquidade do acesso à rede, com redes sem fio e internet, com um aparente limite físico estabelecido da capacidade de processamento de uma unidade central de processamento e o *multiprocessamento* chegando aos computadores pessoais a baixos preços.

A quantidade de memória e velocidade de comunicação das redes permitem que grandes massas de dados sejam processadas e transmitidas, possibilitando video-conferências a um baixo custo. O processamento distribuído tornou-se uma realidade comum, embora ainda explorada apenas por aplicações científicas. A computação móvel tornou-se uma realidade, com a proliferação dos laptops e palmtops, levando os recursos computacionais a qualquer lugar.

2. NÍVEIS DE MAQUINAS

Na seção anterior foi possível verificar como os equipamentos foram evoluindo ao longo do tempo. Entretanto, a partir das máquinas de estado sólido, estabeleceu-se um paradigma de organização que facilita o estudo destes equipamentos.

Cada um dos níveis desta organização é denominado "nível de máquina" e, dentro de certos limites, cada um destes níveis de abstração pode ser estudado independentemente. Nestes termos, podem ser definidos 7 níveis de uma máquina, do mais alto para o mais baixo:

1. Programas Aplicativos
2. Linguagens de Alto Nível
3. Linguagem Assembly / de Máquina
4. Controle Microprogramado
5. Unidades Funcionais
6. Portas Lógicas
7. Transistores e Fios

Esta "independência" é que permite, na prática, que o usuário de um software qualquer não precise conhecer programação e que um programador não precise entender de eletrônica e portas lógicas, ficando esta tarefa apenas para os Engenheiros Eletrônicos.

A compreensão destes níveis é importante para que sejam compreendidos os diferentes níveis de compatibilidade que podem existir entre dois equipamentos.

Programas Aplicativos: é o nível com que, obviamente, o usuário de computador está mais familiarizado. É neste nível que o usuário interage com o computador, usando um programa (como jogos, editores gráficos ou de texto). Neste nível, quase nada (ou nada mesmo) da arquitetura interna é visível. Neste nível, existe a compatibilidade de "usabilidade", do tipo que você espera ao executar um programa como Microsoft Office ou Firefox independente de estar executando em um PC ou Mac.

Linguagens de Alto Nível: é o nível com que lidam os programadores de linguagens como C/C++, Pascal, Java etc. O programador lida com todos os detalhes de instruções e tipos de dados da linguagem que não necessariamente têm a ver com as instruções e tipos de dados da linguagem de máquina. É interessante citar a exceção do C/C++, onde algumas vezes o programador é obrigado a lidar com características especiais da linguagem de máquina. Por esta razão, o C/C++ às vezes é chamado, informalmente, de "a única linguagem de médio nível". Neste nível temos a chamada "compatibilidade de código fonte", em que um código escrito da maneira correta pode ser *compilado* para "qualquer" processador (ou CPU) e funcionar normalmente.

Linguagem Assembly / de Máquina: enquanto uma linguagem considerada de alto nível tem pouco a ver (ou nada a ver) com as instruções e estruturas de dados típicas de uma dada CPU, a linguagem de máquina (de baixo nível) é exatamente a linguagem desta CPU, com instruções próprias e tipos de dados intimamente ligados à forma como a CPU funciona. Estas instruções de uma CPU são chamadas de **conjunto de instruções** da máquina. Para programar neste nível, o programador precisa conhecer muito bem toda a arquitetura da máquina e também seu conjunto de instruções. Quando máquinas são compatíveis neste nível - ainda que o circuito seja completamente diferente de uma para outra, é dito que elas têm **compatibilidade binária**, pois uma é capaz de executar códigos de máquina da outra. A compatibilidade entre os diversos processadores Intel x86 e "compatíveis" vem até este nível.

Nos computadores digitais, a linguagem de máquina é composta por instruções binárias (longas seqüências de zeros e uns), também chamado de **código de máquina binário**. Entretanto, nenhum programdor com um mínimo de recursos disponíveis trabalha com tais códigos, por ser um trabalho extremamente tedioso e sujeito a erros de digitação. Ao trabalhar com programação de baixo nível é comum o uso de **montadores** (*assemblers*), que foram, certamente, um dos primeiros tipos de software escritos. Estes montadores permitem que usemos palavras chamadas **mnemônicos** para expressar instruções da CPU (LOAD, MOVE, JUMP etc.) e o trabalho destes montadores é justamente o de traduzir estes mnemônicos para códigos de máquina. O conjunto de mnemônicos cujas construções têm relação direta de um para um com a linguagem de máquina é chamada **linguagem de montagem** (linguagem *assembly*).

Controle Microprogramado: é o nível que faz a interface entre a linguagem de máquina (código de máquina) e os circuitos que realmente efetuam as operações, interpretando instrução por instrução, executando-as uma a uma. Nos processadores "compatíveis com x86", incluindo os da própria Intel, é nessa camada que é feita a "mágica" da compatibilidade. Este nível foi "criado" pela IBM com a série de computadores **IBM 360**, em meados da década de 1960. Existem duas formas de fazer a microprogramação: uma delas é através de circuitos lógicos (*hardwired*), o que é extremamente eficiente e rápido, mas de projeto bastante complexo. Uma outra solução é através do microprograma, que nada mais é que um pequeno programa escrito em uma linguagem de ainda mais baixo nível executado por um **microcontrolador**. Este microprograma é também chamado de **firmware**, sendo parte hardware e parte software.

Unidades Funcionais: a grande maioria das operações da Unidade de Controle (parte da CPU) são exatamente para mover dados para dentro e para fora das "unidades funcionais". Estas unidades têm esse nome porque executam alguma tarefa importante para o funcionamento da máquina e, dentre elas, temos os registradores da CPU (memórias internas da CPU que possuem um nome específico), a ULA (que realiza as contas de fato) e a memória principal.

Portas Lógicas, Transistores e Fios: este é o nível mais baixo que ainda remete ao funcionamento de mais alto nível. As unidades funcionais são compostas de **portas lógicas**, que por sua vez são compostas de **transistores** interconectados. Abaixo deste nível existem apenas detalhes de implementação de circuitos (como níveis de voltagem, atrasos de sinal etc).

3. BASES NUMÉRICAS

Como foi visto nos níveis de máquina, no nível mais baixo tudo ocorre através de portas lógicas, isto é, com informações lógicas do tipo "falso" ou "verdadeiro". Eletrônicamente falando, é comum considerar "falso" o sinal 0, isto é, um fio cuja tensão vale 0V, e considerar "verdadeiro" o sinal 1, isto é, um fio cuja tensão vale algo diferente de 0V (usualmente 3.3V ou 5V).

Como os fios do computador só podem estar "sem tensão" ou "com tensão" - 0 ou 1 -, o sistema de numeração mais próximo do funcionamento básico do computador é o sistema binário, em que apenas os dígitos 0 e 1 são usados. Assim, a menor unidade de informação tratada por um computador é chamada "bit", e tem o valor 0 ou o valor 1.

Por outro lado, para o programador, representar tudo através de números 0 e 1 pode ser bastante aborrecido (para não dizer chato, mesmo). Assim, nos níveis mais altos, é comum o uso de outras bases numéricas.

Nos primeiros computadores microprocessados, as instruções e informações eram organizadas em grupos de 4 bits; entretanto, apenas os 3 bits inferiores (os mais à direita)

eram "úteis"; como os computadores não eram exatamente confiáveis, o bit mais alto (o mais à esquerda) era usado como um dígito verificador (chamado "bit de paridade") e o programador tinha acesso a apenas 8 combinações numéricas. Assim, a notação OCTAL ganhou destaque, pois permitira exprimir cada sequência de 3 bits apenas com um dígito:

000	-	0	100	-	4
001	-	1	101	-	5
010	-	2	110	-	6
011	-	3	111	-	7

Por parecer pouca economia, mas observe o ganho no tempo de digitação e leitura. Cada tríade de bits era substituída por um único dígito.

000 010 111 100 001 000 100 101 010 <=> 0 2 7 4 1 0 4 5 2

Quando os computadores se tornaram mais confiáveis, as informações passaram a ser organizadas em grupos de 4 bits, denominado "*nibble*". Os "octais", de 0 a 7, não era mais suficientes, e o uso de decimais não era o mais adequado, como pode ser visto abaixo, devido à necessidade de dois dígitos para representar cada nibble.

0000	-	00	1000	-	08
0001	-	01	1001	-	09
0010	-	02	1010	-	10
0011	-	03	1011	-	11
0100	-	04	1100	-	12
0101	-	05	1101	-	13
0110	-	06	1110	-	14
0111	-	07	1111	-	15

Por essa razão, alguém teve a idéia de usar uma representação "hexadecimal", que os os dígitos considerados são 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F, onde A representa o 10, o B representa o 11, o C representa o 12, o D representa o 13, o E representa o 14 e, finalmente, o F representa o 15. Assim, a tabela anterior pode ser escrita da seguinte forma:

0000	-	0	1000	-	8
0001	-	1	1001	-	9
0010	-	2	1010	-	A
0011	-	3	1011	-	B
0100	-	4	1100	-	C
0101	-	5	1101	-	D
0110	-	6	1110	-	E
0111	-	7	1111	-	F

E o conjunto de nibbles abaixo pode ser convertido sem qualquer dúvida:

0000 0001 1000 0100 1100 1001 1110 <=> 0 1 8 4 C 9 E

Quando os dados passaram a ser considerado em 8 bits, percebeu-se que não seria viável o uso de novas letras, uma vez que há 256 combinações possíveis (de 0 a 255) com 8 bits e, passou-se então, a especificar os números em hexadecimal, com um dígito por *nibble*:

00101101 <=> 2D

Os números de 16 bits passaram a ser representados como 4 dígitos hexademais e os números de 32 bits passaram a ser representados por 8 dígitos hexadecimais, separados por : quatro a quatro:

00101101 11110000 : 11001110 00010110 <=> 2DF0:CE16

Um ganho significativo! Reduz-se o tempo de digitação e a probabilidade de erro.

Entretanto, o uso de notação binária, octal e hexadecimal não é natural para nós, seres humanos, que estamos habituados ao sistema decimal. Assim, nas próximas aulas trabalharemos bastante as conversões ente bases numéricas, para que todos se familiarizem com as novas notações, que aparecerão com frequência tanto nos projetos de circuitos digitais quanto nos códigos de programação.

4. BIBLIOGRAFIA

MONTEIRO, M.A. **Introdução à Organização de Computadores**. 5ª. Ed. Rio de Janeiro: LTC, 2008.

MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª.Ed. São Paulo: Prentice Hall, 2003.

Unidade 2: Sistemas de Numeração

Numerais Binários e Bases de Potência de Dois

Prof. Daniel Caetano

Objetivo: Apresentar as diferentes bases numéricas, preparando o aluno para compreender e trabalhar com lógica binária.

Bibliografia STALLINGS, 2003; MURDOCCA e HEURING, 2000.

INTRODUÇÃO

Como discutido na aula anterior, o computador trabalha com sinais elétricos do tipo "ligado" e "desligado" e, como uma consequência disso, quando usamos sinais em fios para representar números, torna-se mais natural uma representação em potências de dois.

Sendo assim, a capacidade de leitura/especificação de valores nas mais diversas bases de numeração torna-se fundamental a todo profissional que almeje trabalhar com lógica digital e a construção/manutenção de equipamentos baseados nesta.

Para propiciar familiaridade dos alunos com estas diferentes bases numércias, este será o assunto desta e das próximas aulas, antes de estudarmos o funcionamento interno dos equipamentos computacionais.

1. REPRESENTAÇÕES NUMÉRICAS

Primeiramente, é importante diferenciar o que são números do que são quantidades. A quantidade de elementos em um conjuto é um conceito abstrato oriundo da contagem dos elementos. É possível comparar quantidades - isto é, dizer se um conjunto é maior que outro - independentemente de existir um nome para essa quantidade. Por exemplo: os dois conjuntos abaixo possuem diferentes quantidades de bolinhas:

Conjunto 1	Conjunto 2
o o o o o	o o o o o o o o o o o o o o o

Os números são representações simbólicas convenientes para quantidades. Por exemplo: o Conjunto 1 tem 5 bolinhas e o Conjunto 2 tem 15 bolinhas. Entretanto, será que essa é a única forma de representar quantidades?

Na verdade, esta não apenas não é a única como também não foi a primeira. Esta forma de representação numérica é chamada "representação decimal com numerais indu-arábicos". Uma outra forma tradicional de representar os números é através dos numerais romanos, que não seguem uma base numércia tradicional usando letras como I, L, C, X, M e V para representar as quantidades. Segundo a representação romana, o primeiro conjunto tem V bolinhas e o seguindo tem XV bolinhas.

Existem outras representações "não-decimais" usando númerais indú-arábicos, como a binária e a octal. Existem aquelas que usam caracteres alfanuméricos para representar as quantidades, como a hexadecimal. A tabela a seguir mostra a contagem de 0 a 15 representada em diferentes formas.

Decimal	Romana	Binária	Octal	Hexadecimal
0	-	0	0	0
1	I	1	1	1
2	II	10	2	2
3	III	11	3	3
4	IV	100	4	4
5	V	101	5	5
6	VI	110	6	6
7	VII	111	7	7
8	VIII	1000	10	8
9	IX	1001	11	9
10	X	1010	12	A
11	XI	1011	13	B
12	XII	1100	14	C
13	XIII	1101	15	D
14	XIV	1110	16	E
15	XV	1111	17	F

Observe que, em cada linha, temos diferentes representações para uma mesma quantidade!

Qual a razão para essa "confusão" toda? Bem, algumas representações, como a romana, são muito antigas e, posteriormente, foram substituídas na maioria dos usos pela numeração indú-arábica decimal. A numeração decimal, por sua vez, parece ser a mais lógica para nós, já que somos capazes de contar com as mãos até 10 elementos.

Entretanto, em alguns casos - como no caso dos computadores, temos de representar as quantidades usando apenas **fios**, pelos quais pode (ou não) passar uma corrente. Nestes casos, somos obrigados a usar vários fios para representar um número, e ele acaba sendo, obrigatoriamente, representado na forma binária.

Na ***eletrônica digital***, cada fio/conexão devem indicar apenas um de dois valores: ligado (com tensão) ou desligado (sem tensão). Essa decisão tem um impacto bastante relevante na forma com que representamos as informações em um computador: considerando que o fio é a "mão" do computador, ele considera apenas dois dígitos: 0 e 1 - diferentemente de nós, que consideramos os dígitos de 0 a 9!

Ocorre que, como a representação biária é um tanto desajeitada - devido ao grande número de dígitos - frequentemente usamos representações que sejam equivalentes, de fácil conversão para o binário: octal e hexadecimal. Observe que, em octal, quando precisarmos de 2 dígitos, a representação em binário já está em 4 dígitos. Já na representação hexadecimal, o segundo dígito só será necessário quando a representação binária tiver 5 dígitos (não aparece nessa tabela). É um considerável ganho para economia de escrita e, adicionalmente, reduz a probabilidade de erros de digitação - bastante alta quando os números envolvem apenas longas sequências de zeros e uns.

Mas como isso é usado na eletrônica? Imagine, por exemplo, que você esteja projetando um equipamento que, como um mouse antigo, deva ser ligado à porta 2F8 (em hexadecimal). 2F8 em hexadecimal é o mesmo que 1011111000 em binário. Considerando um sistema de 16 bits de endereçamento, isso significa que, dos 16 fios, apenas alguns estarão com corrente:

Fio	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Corrente	0	0	0	0	0	0	1	0	1	1	1	1	1	0	0	0

Por essa tabela, o circuito digital a ser construído precisa ser ativado APENAS se houver corrente nos fios 3, 4, 5, 6, 7 e 9 e NÃO houver corrente nos demais.

Como pode haver confusão se usarmos diferentes representações numéricas em um mesmo texto, usa-se a seguinte convenção:

Números Decimais: são escritos normalmentem, SEM zero à esquerda.
Exemplos: 5, 30, 44.
Números Binários: são escritos com o acréscimo de um "b" ao final.
Exemplos: 101b, 11110b, 101100b.
Números Octais: são escritos como os decimais, mas COM um zero à esquerda.
Exemplos: 05, 036, 054.
Números Hexadecimais: são escritos com o acréscimo de um "h" ao final.
Exemplos: 5h, 1Eh, 2Ch

Nota: uma forma alternativa de representar os números hexadecimais é usada nas linguagens C/C++, Java e outras. Nestas, ao invés de se acrescentar o "h" ao final, acrescenta-se o prefixo "0x" no início. Exemplos: 0x5, 0x1E, 0x2C.

2. NOTAÇÃO POSICIONAL

Um grande avanço da notação indú-arábica decimal com relação à romana é o uso de notação posicional. A notação posicional significa que a quantidade que um número representa depende da posição em que ele aparece na reprentação completa. Por exemplo: qual a quantidade representada pelo símbolo "1"? Se você respondeu "1, oras!", errou! O símbolo 1 tem diferentes significados, de acordo com a posição no número!

Consideremos os números decimais. Neste caso, se o 1 estiver na primeira casa, ele vale **uma unidade**. Se estiver na segunda casa, ele vale **uma dezena**. Se estiver na terceira casa, ele vale **uma centena**... na quarta vale **uma unidade de milhar** e assim por diante. Observe:

1 : Um
10 : Dez
100 : Cem
1000 : Mil

1101 : Mil cento e um.

Neste último caso, observe que o número pode ser construído com uma soma de suas partes: 1101 = 1000 + 100 + 1. O mesmo vale quando temos outros números:

12345 = 10000 + 2000 + 300 + 40 + 5
4532 = 4000 + 500 + 30 + 2

Observe que a posição de um número indica quantos zeros devem ser acrescentados ao seu lado para identificarmos a quantidade que ele representa. Considere este exemplo:

4356 = 4000 + 300 + 50 + 6

Se conisderarmos esse número contando suas casas da direita para a esquerda, começando em zero, teremos uma correspondência direta:

Casa	3	2	1	0
Dígito	4	5	3	2
Quantidade	4.000	500	30	2

Observe: na casa 3, a quantidade real tem 3 zeros; na casa 2, a quantidade real tem 2 zeros... e assim por diante. Na tabela abaixo, escreveremos as quantidades de uma maneira diferente:

Casa	3	2	1	0
Dígito	4	5	3	2
Quantidade	4×10^3	5×10^2	3×10^1	2×10^0

Observe que o expoente do "10" é exatamente a posição do dígito em questão. Isso ocorre porque estamos usando, para construir os números, 10 dígitos diferentes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Sempre que não há mais dígitos em uma das posições, acrescentamos um à posição imediatamente à esquerda do número.

É por essa razão que esta representação é chamada de "decimal". A representação binária, por usa vez, usa apenas dois valores para os dígitos: 0 e 1. A representação octal usa oito valores diferentes para os dígitos: 0, 1, 2, 3, 4, 5, 6 e 7. Finalmente, a representação hexadecimal usa dezesseis valores diferentes para os dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F.

3. A NOTAÇÃO BINÁRIA

Como apresentado anteriormente, acredita-se que os humanos trabalhem com números decimais por conta da quantidade de dedos que temos nas mãos; os computadores, entretanto, foram construídos com uma outra característica e, portanto, a representação mais natural neste caso é a binária.

A informação que pode ser representada por um "fio" - 0 ou 1 - é denominada **bit**, e é a menor unidade de informação de um computador. Se um processador tivesse apenas 1 bit, ele só seria capaz de representar os números 0 e 1. Mas, e se ele tiver 2 bits? A tabela abaixo mostra todas as combinações possíveis:

00 01 10 11

E se ele tiver 3 bits?

000 001 010 011 100 101 110 111

E se ele tiver 4 bits?

0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111

Observe que o número de bits representa o número de dígitos binários; além disso, considerando **n** bits, o número de combinações possíveis é dado por 2^n: um computador com 8 bits pode representar até $2^8 = 256$ números e um de 16 bits pode representar até $2^{16} = 65536$ números... e assim por diante.

Nota: como 256 variações eram suficientes para representar a maior parte das informações necessárias nos primeiros computadores, o conjunto de 8 bits ganhou um nome específico: byte. Assim, um **byte é um conjunto de 8 bits**.

3.1. Conversão de Números Binários para Decimais

Mas que números esse valores representam, em nossa notação decimal?

Existe uma regra de conversão muito simples. Lembremos como representamos o número decimal anterioremente:

Dígito	3	2	1	0
Número	1	5	3	7

$1537 = 1*10^3 + 5*10^2 + 3*10^1 + 7*10^0$

Se fizermos o mesmo com um número binário, por exemplo, 1101, teremos:

Dígito	3	2	1	0
Número	1	1	0	1

$1101 \text{ binário} = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

Observe que os 10^n foram substituídos por 2^n; a razão para isso é que antes estávamos em uma base decimal, agora estamos em uma base binária. Façamos essa conta:

$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8 + 4 + 0 + 1 = 13$

Logo: 1101 binário = 13 decimal

Observe que agora temos uma confusão de representação: um número binário poderia ser lido erradamente como um número decimal. Para evitar esse problema, é usual acrescentar a letra "b" após valores binários, para evitar confusão. Em outras palavras, a representação é:

Texto	Valor (em decimal)
1101	1101
1101b	13

A tabela a seguir mostra a conversão das 16 combinações de números de 4 bits de binário para decimal:

Binário	Decimal	Binário	Decimal
0000b	0	1000b	8
0001b	1	1001b	9
0010b	2	1010b	10
0011b	3	1011b	11
0100b	4	1100b	12
0101b	5	1101b	13
0110b	6	1110b	14
0111b	7	1111b	15

3.2. Conversão de Números Decimais para Binários

A conversão de números decimais para binários é similar, e é realizada com um processo de sucessivas divisões inteiras por dois, parando quando a divisão valer 0. Os restos das divisões vão compondo o valor em binário, da esquerda para a direita. Por exemplo: vamos transformar o valor 13 em sua representação binária:

13 / 2 = 6 e sobra... 1
6 / 2 = 3 e sobra... 0
3 / 2 = 1 e sobra 1
1 / 2 = 0 e sobra 1

Assim, o valor 13 é representado em binário como 1101b. Tentemos novamente com outro número maior, 118:

118 / 2 = 59 e sobra... 0
59 / 2 = 29 e sobra... 1
29 / 2 = 14 e sobra... 1
14 / 2 = 7 e sobra... 0
7 / 2 = 3 e sobra... 1
3 / 2 = 1 e sobra... 1
1 / 2 = 0 e sobra... 1

Assim, o valor 118 é representado em binário como 1110110b

4. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 3: Sistemas de Numeração

Conversões Entre Quaisquer Bases e Aritmética em Bases Alternativas

Prof. Daniel Caetano

Objetivo: Apresentar métodos genéricos de conversão entre bases, bem como operações sem a necessidade de realizar conversões.

Bibliografia STALLINGS, 2003; MURDOCCA e HEURING, 2000.

INTRODUÇÃO

Na aula anterior foram apresentados os métodos para conversão entre as bases decimal e binária. Nesta aula veremos que o método aplicado serve para converter entre quaisquer bases, usando a decimal como intermediária.

Adicionalmente, veremos como lidar com números fracionários em potências de dois, em especial com binários, e como realizar as operações básicas com números representados em bases diferentes de 2.

1. RECORDANDO CONVERSÕES DECIMAL/BINÁRIO

1.1. Conversão de Números Binários para Decimais

Multiplica-se cada dígito pela correspondente potência de **dois**. Exemplo: converter o número 1101b para decimal:

Dígito	3	2	1	0
Número	1	1	0	1

$1101b = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8 + 4 + 0 + 1 = 13$

1.2. Conversão de Números Decimais para Binários

A conversão é feita com divisões sucessivas por **dois**, anotando os restos da divisão, que formam o número binário da direita para a esquerda. Exemplo: converter o valor 13 em sua representação binária:

13 / 2 = 6 e sobra... 1
6 / 2 = 3 e sobra... 0
3 / 2 = 1 e sobra 1
1 / 2 = 0 e sobra 1

Assim, o valor 13 = 1101b.

Outro exemplo: converter 118 para binário:

118 / 2 = 59 e sobra... 0
59 / 2 = 29 e sobra... 1
29 / 2 = 14 e sobra... 1
14 / 2 = 7 e sobra... 0
7 / 2 = 3 e sobra... 1
3 / 2 = 1 e sobra... 1
1 / 2 = 0 e sobra... 1

Assim, o valor 118 = 1110110b

2. CONVERTENDO DECIMAL/OCTAL

2.1. Conversão de Números Octais para Decimais

Multiplica-se cada dígito pela correspondente potência de **oito**. Exemplo: converter o número 03721 para decimal:

Dígito	3	2	1	0
Número	3	7	2	1

$03721 = 3*8^3 + 7*8^2 + 2*8^1 + 1*8^0 = 3*512 + 7*64 + 2*8 + 1*1 = 2001$

2.2. Conversão de Números Decimais para Octais

A conversão é feita com divisões sucessivas por **oito**, anotando os restos da divisão, que formam o número octal da direita para a esquerda. Exemplo: converter o valor 2001 em sua representação octal:

2001 / 8 = 250 e sobra... 1
250 / 8 = 31 e sobra... 2
31 / 8 = 3 e sobra... 7
3 / 8 = 0 e sobra... 3

Logo, 2001 = 03721.

3. CONVERTENDO DECIMAL/HEXADECIMAL

3.1. Conversão de Números Hexadecimal para Decimais

Multiplica-se cada dígito pela correspondente potência de **dezesseis**. Exemplo: converter o número 0x2F3C para decimal:

Dígito	3	2	1	0
Número	2	F	3	C

$0x2F3C=2*16^3 + 15*16^2 + 3*16^1 + 12*16^0 = 2*4096 + 15*16 + 3*16 + 12*1 = 12092$

3.2. Conversão de Números Decimais para Hexadecimais

A conversão é feita com divisões sucessivas por **dezesseis**, anotando os restos da divisão, que formam o número hexadecimal da direita para a esquerda. Exemplo: converter o valor 12092 em sua representação hexadecimal:

12092 / 16 = 755 e sobra...	12 (C)
755 / 16 = 47 e sobra...	3
47 / 16 = 2 e sobra...	15 (F)
2 / 16 = 0 e sobra...	2

Logo, 12092 = 0x2F3C

4. CONVERTENDO DECIMAL/QUALQUER BASE

4.1. Conversão de Números de Qualquer Base para Decimais

Multiplica-se cada dígito pela correspondente potência do **número da base**. Exemplo: converter o número **abcd**, na base **n**, para decimal:

Dígito	3	2	1	0
Número	a	b	c	d

abcd $= a*n^3 + b*n^2 + c*n^1 + d*n^0$

4.2. Conversão de Números Decimais para Qualquer Base

A conversão é feita com divisões sucessivas pelo **número da base**, anotando os restos da divisão, que formam o número na base em questão, da direita para a esquerda. Exemplo: converter o valor **x1** em sua representação na base **n**:

x1 / n = x2 e sobra... d
x2 / n = x3 e sobra... c
x3 / n = x4 e sobra... b
x4 / n = 0 e sobra... a

Logo, **x1** na base 10 = **abcd** na base n

5. CONVERTENDO NÚMEROS ENTRE QUAISQUER BASES

Sempre que for necessário converter números entre quaisquer bases, de maneira genérica, pode-se realizar a conversão usando a base 10 como intermediária. Por exemplo:

Converta o número **x1** na base **a** para a base **b**.

Esse problema pode ser decomposto em:

1) Converta **x1** na base **a** para a base **10**, obtendo **x2** na base 10.
2) Converta **x2** na base **10** para a base **b**, obtendo **x3** na base b.

As conversões envolvendo binários e octais e binários e hexadecimais podem ser feitas de forma direta, bastando decorar uma tabelinha básica de conversão.

A tabela abaixo, para conversões Binário<=>Octal, deve ser usada lembrando-se de que cada dígito octal viram 3 dígitos em binário... e que cada 3 dígitos em binário se torna um único em octal.

Binária	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Por exemplo, para converter o número 1011101101b em um número octal, separamos este número em grupos de 3 bits, da direita para a esquerda:

1 011 101 101

Como um bit ficou sozinho na esquerda, complementamos com mais dois bits zero:

001 011 101 101

Agora basta usar a tabela para converter:

001 011 101 101
1 3 5 5

Assim, 1011101101b = 01355.

A conversão contrária é idêntica: converter 01355 para binário:

1 3 5 5
001 011 101 101

Portanto 01355 = 1011101101b

A conversão entre binário e hexadecimal se faz da mesma forma, mas usa-se a tabela abaixo e deve-se lembrar que cada dígito hexadecimal corresponde a QUATRO dígitos binários.

Binária	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Por exemplo, para converter o número 1011101101b em um número hexadecimal, separamos este número em grupos de 4 bits, da direita para a esquerda:

10 1110 1101

Como dois bits ficaram sozinhos na esquerda, complementamos com mais dois bits zero:

0010 1110 1101

Agora basta usar a tabela para converter:

0010	1110	1101
2	E	D

Assim, 1011101101b = 0x2ED.

A conversão contrária é idêntica: converter 0x2ED para binário:

2	E	D
0010	1110	1101

Portanto 0x2ED = 1011101101b

6. NÚMEROS FRACIONÁRIOS EM OUTRAS BASES

Até agora só, lidamos com números inteiros. É possível lidar com conversões de números fracionários? SIM! É possível. Será apresentada a regra para números binários, sendo que a regra é análoga para as outras bases!

6.1. Conversão de Números Binários Fracionários para Decimais

Multiplica-se cada dígito pela correspondente potência de **dois**, lembrando que números depois da vírgula possuem expoente negativo! Exemplo: converter o número 1101,1001b para decimal:

Dígito	3	2	1	0	-1	-2	-3	-4
Número	1	1	0	1	1	0	0	1

$1101,1001b = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4}$
$= 8 + 4 + 0 + 1 + 0,5 + 0 + 0 + 0,0625$
$= 13,5625$

6.2. Conversão de Números Decimais Fracionários para Binários

A parte inteira é convertida com divisões sucessivas por **dois**, anotando os restos da divisão, que formam o número binário da direita para a esquerda - como já foi feito antes. A parte fracionária é convertida com multiplicações sucessivas por **dois**, "retirando" a parte inteira do número para compor a parte fracionária binária, da esquerda para a direita Exemplo: converter o valor 13,5625 em sua representação binária:

Parte Inteira:
13 / 2 = 6 e sobra... 1
6 / 2 = 3 e sobra... 0
3 / 2 = 1 e sobra 1
1 / 2 = 0 e sobra 1

Assim 13 = 1101b

Parte Fracionária:
0,5625 * 2 = **1**,125 => 1
0,125 * 2 = **0**,250 => 0
0,25 * 2 = **0**,5 => 0
0,5 * 2 = **1**,0 => 1
0,0 * 2 = **0**,0 => 0
0,0 * 2 = **0**,0 => 0
0,0 * 2 = **0**,0 => 0
0,0 * 2 = **0**,0 => 0
...
Assim, 0,5625 = 0,10010000b

Componto ambos:

13,5625 = 1101,10010000b = 1101,1001b

Lembrando que zeros à direita do número após a vírgula não modificam o valor do número. Em alguns casos, não é possível se chegar ao valor ZERO como nesse caso (por exemplo, se tentarmos converter PI (3,141592...) para binário. Neste caso, procede-se até obter a precisão desejada.

O processo para outras bases é análogo, substituindo o "2" pelo número da base.

7. ARITMÉTICA EM OUTRAS BASES

Quando trabalhamos com números decimais, fazemos operações diretas. Por exemplo:

```
 1
 15
+7
 22
```

No fundo, alinhamos as casas (unidade com unidade, dezena com dezena, centena com centena...) e depois somamos uma a uma, começando com a unidade e, em seguida, partindo para a dezena e centena. Quando o resultado de uma das casas é maior do que o valor da base (no exemplo, 5 + 7 = 12), subtraímos deste resultado o valor da base (12 - 10 = 2) e, fazemos o "vai um" (representado no exemplo como o pequeno algarismo 1 sobre o 15).

Realizar a soma em outras bases é exatamente a mesma coisa. Veja em binário:

```
  1  1
 1101b          =    13
+0101b          =   +5
10010b          =    18
```

Começando da direita para a esquerda:

Primeira Casa: 1 + 1 = 2; como 2 não pode ser representado em binário, subtraímos 2 (2-2 = 0) e fazemos o vai um.

Segunda Casa: 0 + 0 = 0, somando com o "1" que veio da casa anterior 0+1 = 1.

Terceira Casa: 1 + 1 = 2. Mais uma vez não é possível representar, deixamos zero (2-2) no lugar e vai um.

Quarta Casa: A soma é 1 + 0 = 1, mas ao somar com o "1" que veio da casa anterior, 1+1 = 2, deixando zero no lugar e, mais uma vez, "vai um".

Quinta Casa: Como ela não existe nos números originais, permanece apenas o "1" que veio da casa anterior.

O processo é análogo para outras bases:

```
    1
 0x25       =     37
+0x3C       =    +60
 0x61       =     97
```

Começando da direita para a esquerda:

Primeira Casa: 5 + C(12) = 17. 17 não pode ser representado... então indicamos 17-16 = 1 e vai um

Segunda Casa: 2 + 3 = 5. Somando com o "1" que veio da casa anterior: 5 + 1 = 6.

Será que podemos aplicar a mesma lógica para a subtração? É claro que sim! Vejamos primeiro com decimais:

```
 1
 25
-7
 18
```

Primeira Casa: temos 5 - 7; não é possível fazer, então "emprestamos um" da próxima casa, que aqui na unidade vale 10 e a nova conta é (10+5) - 7 = 8.

Segunda Casa: temos 2, mas que deve subtrair o "1" que foi emprestado pela primeira casa, então 2-1 = 1.

Vejamos agora em binário

```
  1 1 1
  1100b          =     12
 -0101b          =     -5
  0111b          =      7
```

Primeira Casa: 0 - 1; não é possível. Então "emprestamos 1" da próxima casa, que aqui na primeira casa vale 2. A nova conta é, então (2+0) -1 = 1

Segunda Casa: 0-0 = 0; entretanto, precisamos descontar o 1 que foi emprestado para a primeira casa; como 0-1 não é possível, somos obrigados a emprestar 1 da terceira casa, que aqui vale 2. A nova conta é: (2+0)-1 = 1.

Terceira Casa: 1-1 = 0... mas mais uma vez é preciso descontar o 1 que foi emprestado para a casa anterior... e, para isso, é preciso emprestar 1 da quarta casa! Daí (2+0)-1 = 1.

Quarta Casa: 1-0 = 1, que descontado o 1 que foi emprestado... 0.

O mesmo pode ser aplicado para a multiplicação. Façamos direto em binário:

```
   11b      =      3
  x10b      =     x2
   00b             6
  11b+
  110b
```

A divisão fica como exercício!

8. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 4: Sistemas de Numeração

Representação de Dados em Ponto Fixo

Prof. Daniel Caetano

Objetivo: Apresentar as representações mais utilizadas para ponto fixo.

Bibliografia STALLINGS, 2003; MURDOCCA e HEURING, 2000.

INTRODUÇÃO

Na aula anterior foram apresentados os métodos para conversão entre diversas bases e também como realizar cálculos diretamente nas bases de potências de dois. Nesta aula serão apresentadas maneiras de representar numerais de ponto fixo com sinal usando a representação binária, de maneira compatível com os cálculos apresentados anteriormente.

1. PONTO FIXO

Chamamos de representação em ponto fixo aquela que o numeral é guardado como um valor inteiro na memória, sem a especificação da vírgula. Esta última, por sua vez, é considerada **fixa** em uma posição (daí o nome, ponto fixo) no momento da interpretação do valor. Considere o número binário abaixo, por exemplo:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	1	0	1	1	0	0	1

Considerando esta representação sem sinal, em ponto fixo, ele pode ser interpretado como sendo um número inteiro: 11011001b, que é o mesmo que 217 em decimal. Se, por outro lado, considerarmos que a vírgula está entre o bit 3 e o bit 4, o número pode ser interpretaod como 1101,1001b, que é o mesmo que 13,5625.

Usualmente, o computador trata os números de ponto fixo como sendo inteiros, isto é, a vírgula situa-se à direita do bit 0, daí muitas vezes nos referirmos aos números inteiros como sendo números de ponto fixo (eles, de fato, o são!). Nesta aula, trataremos especificamente dos números inteiros.

2. REPRESENTAÇÃO DE SINAL

É muito comum a necessidade de representar números negativos. De fato, em especial quando fazemos contas, é frequente o surgimento dos números negativos. A primeira idéia

para representar números negativos é reservar o bit mais significativo (o de "número mais alto") como sendo o bit de sinal, não sendo considerado para a interpretação do valor. Considerando que se o bit de sinal contiver um valor 0 o número é positivo, se ele contiver o valor 1, o número será positivo.

Por exemplo: vamos representar o número 97:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	1	0	0	0	0	1

Agora, vamos representar o número -97:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	1	1	0	0	0	0	1

Observe que a diferença está apenas no bit do sinal.

Como sobraram apenas 7 bits (0 a 6) para indicar o número, não é mais possível fazer números entre 0 e 255; com 7 bits é possível apenas gerar números entre 0 e 127, inclusive. Considerando o bit do sinal, porém, temos duas faixas representáveis: de -127 a 0 e de 0 a 127.

Ou seja: com 1 byte podemos representar um número de 0 a 255 sem sinal ou, usando a representação indicada assim, de -127 a 127, considerando o sinal. Os bits que representam a parte numérica recebem o nome de **magnitude**.

Entretanto, essa representação não é comumente usada. Há dois problemas "graves" com ela. O primeiro é que há dois zeros na sequência de -127 a 127: o 0 e o -0, que, obviamente, deveriam ter a mesma representação (zero é sempre zero!):

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	0

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	0	0	0	0	0

O segundo problema é nas operações. Vejamos:

Decimal	Binário	Convertendo para Decimal
	1 1 1 1 1 1	
2	00000010b	2
-5	-00000101b	-5
-3	11111101b	-125

Opa, algo deu BEM errado! Vamos tentar repetir a operação, mas fazendo como 2 + (-5) e ver o resultado.

Binário	Convertendo para Decimal
1 1 1 1 1 1	
00000010b	2
+10000101b	-5
10000111b	-7

É, definitivamente essa representação não é muito boa!

2.1. Representação de Sinal com Complemento de Um

Os efeitos apresentados nas operações anteriores são advindos do fato que o bit de sinal está atrapalhando a continuidade da contagem binária: se tivermos o número 127 e somarmos 1, deveríamos ter uma contagem cíclica que levasse ao -127... Mas não é isso que ocorre:

Binário:	...	01111110b	01111111b	10000000b	10000001b	...
Decimal:	...	126	127	-0	-1	

Ainda: observe que ao **somarmos** 1 unidade positiva ao valor negativo -1, ele se torna -2, o que é totalmente inadequado. Para remediar essa solução, é usada a **representação em complemento**. O que significa isso? Significa que números negativos são obtidos invertendo **todos os bits** do número positivo.

Assim, se 97 positivo é assim:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	1	0	0	0	0	1

O número -97 fica assim:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	1	1	1	1	0

Observe que os bits da magnitude foram completamente invertidos! Observe que, neste caso, agora há continuidade:

Binário:	...	01111110b	01111111b	10000000b	10000001b	...
Decimal:	...	126	127	-127	-126	

Binário:	...	11111110b	11111111b	00000000b	00000001b	...
Decimal:	...	-1	-0	0	1	

Adicionalmente, se somarmos 1 unidad positiva a um número negativo, ele terá o comportamento adequado. Por exemplo: somando 1 (00000001b) a -1 (11111110b) o resultado é -0 (11111111b) = 0 (00000000b).

Esta representação funciona **muito** melhor que a representação simplificada com bit de sinal, mas ainda tem um problema: há duas representações para zero. Isso significa que há uma descontinuidade na conta: se eu somar -1 com 2, ao invés de obter 1, eu obterei...

Decimal	Binário (comp. de 1)	Binário para Decimal
-1	11111110b	-1
+2	-00000010b	+2
+1	00000000b	0

Como resolver este problema?

2.2. Representação de Sinal com Complemento de Dois

Para corrigir os efeitos das representações anteriores, ao invés de realizar o complemento de 1 (inverter os bits), realiza-se o complemento de 2, que é o seguinte:

"Para converter um número positivo para negativo, calcula-se seu complemento e soma-se 1".

Em outras palavras, calcula-se o complemento de 1 e soma-se um ao resultado.

Assim, se 97 é representado como abaixo:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	1	0	0	0	0	1

O número -97 em complemento de **um** fica assim:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	1	1	1	1	0

O número -97 em complemento de **dois** fica assim:

Bit 7 (SINAL)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	1	1	1	1	1

Para desinverter, isto é, transformar um negativo em um positivo, o processo é inverso: subtrai-se 1 e, em seguida, realiza-se a operação de complemento de 1.

Com isso, todos os problemas anteriores ficam solucionados. A continuidade existe:

Binário:	...	01111110b	01111111b	10000000b	10000001b	...
Decimal:	...	126	127	-128	-127	

Binário:	...	11111110b	11111111b	00000000b	00000001b	...
Decimal:	...	-2	-1	0	1	

Além de ser corrigido o problema anterior - repetição do zero -, agora há um número negativo adicional a ser representado: -128.

3. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 5: Sistemas de Representação

Números de Ponto Flutuante IEEE 754/2008 e Caracteres ASCII

Prof. Daniel Caetano

Objetivo: Compreender a representação numérica em ponto flutuante.

INTRODUÇÃO

Como foi visto em aulas anteriores, é possível armazernar na memória um valor que pode ser considerado fracionário usando a representação de **ponto fixo**, isto é, considerando a vírgula fixa em uma determinada posição. Assim, o número 110,101b, considerando ponto fixo com 4 dígitos depois da vírgula, este número poderia ser armazenado assim:

Parte Inteira				Parte Fracionária			
0	1	1	0	1	0	1	0

Essa representação é conveniente, porque permite representar e realizar operações com números reais; entretanto, ela tem o inconveniente de limitar a representação; o menor valor fracionário representável por este caso acima é $1/2^4 = 1/16 = 0,0625$. Não há como representar, por exemplo, o número 0,01. Isso é indesejável porque, na prática, alguns números que trabalhamos são de ordens de grandeza muito baixas, isto é, da ordem de 10^{-50}, que precisariam de um número binário muito grande para ser representado.

A solução para isso está em usar números de **ponto flutuante**.

1. REPRESENTAÇÃO EM PONTO FLUTUANTE

Quando os engenheiros se depararam com o problema de representação acima, tiveram que parar para pensar um pouco. Não demorou muito e a sugestão para resolver este problema veio, tendo como base a representação numérica científica - muito usada por engenheiros e físicos. Na base decimal, esta representação é a seguinte:

Tradicional	**Científica**	**Científica Reduzida**
125	$1,25 * 10^2$	1,25E2

A idéia é representar um número com apenas um dígito inteiro e ajustar a posição correta da vírgula com uma potência de 10; desta forma, um número com qualquer número de casas decimais fica representado por três números: a parte inteira (denominada **característica**), a parte fracionária (denominada **mantissa**) e o expoente da potência de 10 (denominado **expoente**). Obseve:

Tradicional	Científica	Característica	Mantissa	Expoente
125	1,25E2	1	25	2

Um número que seria muito difícil de escrever na forma decimal, por ser muito pequeno, pode ser facilmente escrito na notação científica e, portanto, representado como um número de ponto flutuante.

Científica	Característica	Mantissa	Expoente
1,25E-56	1	25	-56

O nome "ponto flutuante" vem da existência do expoente, que indica o número de dígitos que a vírgula deve ser deslocada; um expoente negativo significa que a vírgula deve ser deslocada à esquerda e um exponte positivo significa que a vírgula deve ser deslocada a direita.

2. PONTO FLUTUANTE COM NÚMEROS BINÁRIOS

Da mesma forma que representamos números decimais na forma de ponto flutuante, podemos representar números binários. Por exemplo:

Tradicional	Científica	Característica	Mantissa	Expoente
100b	$1,00b * 2^{2}$	1b	00b	2
101b	$1,01b * 2^{2}$	1b	01b	2
11,101b	$1,1101b * 2^{1}$	1b	1101b	1
0,1001b	$1,001b * 2^{-1}$	1b	001b	-1

Observe que a lógica é a mesma... e, inclusive, existe uma regra curiosa: a característica vale sempre 1! **Como a característica é sempre 1, ela não precisa ser representada!**

Assim, um número binário de ponto flutuante é representado apenas por sua mantissa e seu expoente, admitindo-se que sua característica é sempre o bit 1. No caso de números com sinal, jogamos o sinal na mantissa!

Tradicional	Científica	Mantissa	Expoente
100b	$1,00b * 2^{2}$	00b	2
101b	$1,01b * 2^{2}$	01b	2
11,101b	$1,1101b * 2^{1}$	1101b	1
0,1001b	$1,001b * 2^{-1}$	001b	-1

Mas, como representar isso na memória?

3. PONTO FLUTUANTE BINÁRIO NA MEMÓRIA

Anteriormente vimos uma forma simplificada de representar os números negativos, em que reservávamos um bit para indicar o sinal (0 = positivo e 1 = negativo), e usáamos os outros para representar o valor numérico, normalmente.

Sinal	Número						
0	1	1	0	1	0	1	0

Bem, a idéia é a mesma, mas agora teremos que reservar bits para:

a) Sinal do número
b) Sinal do expoente
c) Expoente
d) Mantissa

Para entender a idéia, vamos considerar primeiramente números de 8 bits. Em teoria, podemos reservar um bit para o sinal do número, um bit para o sinal do expoente, dois bits para o expoente e quatro bits para a mantissa:

Sinal	Sinal do Expoente	Expoente		Mantissa			
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Vejamos como o número 2,25 fica representado em ponto flutuante:

a) Primeiramente vamos convertê-lo em binário.
   Parte inteira: 2 = 10b
   Parte Fracionária: 0,25 = 0,01b
   Assim: 2,25 = 10,01b
b) Agora vamos reescrevê-lo em notação científica:
   $10,01b = 1,001b * 2^1$
c) Agora dividimos as partes:
   Sinal: 0 (positivo)
   Característica: 1b
   Mantissa: 001b
   Sinal Expoente: 0 (positivo)
   Expoente: 1
d) Agora representamos na memória

Sinal	Sinal do Expoente	Expoente		Mantissa			
0	0	0	1	0	0	1	0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Observe que o último dígito à direita da mantissa - um zero - foi adicionado para completar a representação. **Nunca** remova zeros à esquerda na mantissa: **eles são importantes**.

4. REPRESENTAÇÃO IEEE 754/2008

A representação anterior com 8 bits é adequada didaticamente, mas como é possível observar, os números que podem ser representados são muito limitados. Em especial, o número de bits da mantissa limita o número de dígitos que podem ser representados (dígitos significativos) e o número de bits do expoente limita a variação de magnitude do número.

Assim, na prática, são usadas representações que usam vários bytes de memória para indicar um único número; como a organização dos bits que representam sinal, expoente e mantissa é arbitrária, é preciso adotar uma padronização.

O IEEE é um órgão composto por engenheiros que define uma série de normas de engenharia. Uma destas normas, a 754 de 2008, define a representação de números mais usada em computadores modernos, definindo números de ponto flutuante de **precisão simples**, com 32 bits, e números de ponto flutuante de **precisão dupla**, com 64 bits.

A representação para precisão simples (32 bits) é a seguinte:

Sinal	Sinal do Expoente	Expoente	Mantissa
Bit 31	Bit 30	Bit 29 ~ Bit 23	Bit 22 ~ Bit 0

São, portanto, 1 bit para sinal, 8 bits para o expoente (incluindo o sinal) e 23 bits para a mantissa.

A representação para precisão dupla (64 bits), por sua vez, é a seguinte:

Sinal	Sinal do Expoente	Expoente	Mantissa
Bit 63	Bit 62	Bit 61 ~ Bit 52	Bit 51 ~ Bit 0

São, portanto, 1 bit para sinal, 11 bits para o expoente (incluindo o sinal) e 52 bits para a mantissa.

O detalhe nestas duas representações do IEEE é que o sinal do número (bit 31 e 63, respectivamente para simples e dupla precisão) é o tradicional, isto é, bit 0 é positivo e bit 1 é negativo; o **sinal do expoente**, entretanto, é **invertido**, isto é, bit 1 é positivo e bit 0 é

negativo. A razão foge ao escopo do curso, mas tem a ver com facilitar as operações matemáticas com números de ponto flutuante.

5. REPRESENTAÇÃO DO ZERO

O aluno, neste instante, pode estar se perguntando: se nas representações de ponto flututante binário o valor da característica é considerado fixo em 1 - e por essa razão nem é indicado -, como representar o valor zero?

A primeira alternativa seria a "gambiarra", isto é, representar uma mantissa vazia com o maior expoente negativo possível. Isso seria adequado na suposição de que 0,000000000000000001 = 0,0, por exemplo. Como essa é uma aproximação grosseira, o IEEE definiu uma maneira diferente de representar o número zero.

Sempre que o valor do EXPOENTE for igual a -0 (isto é, todos os bits zero), o valor da característica é considerado ZERO. Assim, se todos os bits do expoente forem zero, assim como todos os bits da mantissa forem zero, o valor representado será considerado exatamente igual a zero (independente do bit de sinal do número). Exemplo (simulando IEEE com 8 bits):

Sinal	Sinal do Expoente	Expoente		Mantissa			
0	0	0	0	0	0	0	0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Se o número representado no expoente for -0 e a mantissa for um outro valor, será considerado um valor de característica igual a 0. Por exemplo:

Sinal	Sinal do Expoente	Expoente		Mantissa			
0	0	0	0	1	1	0	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Esse valor representa o número binário 0,1101b. Repare que isso é totalmente diferente disso (expoente +0):

Sinal	Sinal do Expoente	Expoente		Mantissa			
0	1	0	0	1	1	0	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Cujo valor é o número binário 1,1101b

Assim, quando o expoente é -0, consideradmos que o valor está em notação científica **não-normalizada**, isto é, o número representado pela **característica+mantissa** gera um valor entre 0,0 e 1,0. Quando o expoente é diferente de -0, considera-se uma notação

científica **normalizada**, isto é, o número representado pela **característica+mantissa** gera um valor entre 1,0 e 2,0.

O IEEE também criou uma representação para o valor "infinito": quando todos os bits do expoente valerem 1 e todos os bits da mantissa valerem 0.

Assim, +infinito pode ser representado assim (simulando em 8 bits):

Sinal	**Sinal do Expoente**	**Expoente**		**Mantissa**			
0	1	1	1	0	0	0	0
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**

E o -inifinito, por sua vez, pode ser representado assim (simulando em 8 bits):

Sinal	**Sinal do Expoente**	**Expoente**		**Mantissa**			
1	1	1	1	0	0	0	0
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**

Observe que os valores da mantissa **devem** ser zero. Se todos os bits do expoente forem 1 e pelo menos um dos bits da mantissa for diferente de zero, o valor será considerado não-numérico (+NaN e -NaN: *Not A Number*).

6. OPERAÇÕES EM PONTO FLUTUANTE (OPCIONAL)

As operações com números em ponto flutuante são realizadas mediante muitas conversões e deslocamentos, já que para muitas delas asmbos os números precisam ter o mesmo expoente. Em algumas operações, a soma ocorre entre os expoentes e, em outras, ocorre apenas com a mantissa, após ajustes.

Visto que as operações não podem ser feitas de maneira direta, exigem um processamento diferenciado. Por essa razão, **nem todos os processadores são capazes de realizar aritmética de ponto flutuante**. Aqueles que as fazem, além da Unidade de Controle (UC) e Unidade Lógica Aritmética (ULA) possuem também uma UPF (Unidade de Ponto Flutuante) para realizar esses cálculos mais rapidamente.

A realização de cálculos de ponto flutuante em equipamentos que não possuem uma UPF exige que o cálculo seja feito "por software", isto é, exigem que um pequeno programa realize essas operações. Isso torna o processamento muito mais lento, sendo uma das razões pelas quais o uso de ponto flutuante é evitado em uma série de razões.

Adicionalmente, como o número de dígitos é limitado ao número de bits e, para realizar operações é frequente que ambos os números sendo operados precisem ser convertidos para o mesmo expoente, pode haver perda significativa de bits durante as

operações. Por exemplo, considere a soma de 2,25 com 0,5625, usando a representação simulada IEEE em 8 bits:

$2{,}25 = 10{,}01b = 1{,}001b * 2^1$

Sinal	Sinal do Expoente	Expoente		Mantissa			
0	1	0	1	0	0	1	0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

$0{,}5625 = 0{,}1001b = 1{,}001b * 2^{-1}$

Sinal	Sinal do Expoente	Expoente		Mantissa			
0	0	0	1	0	0	1	0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Bem, o resultado da soma deveria ser 2,25+0,5625 = 2,8125, certo? Vejamos o que ocorre!

Para realizar a soma das mantissas, é preciso que ambos os números estejam com o mesmo expoente. Sempre iremos converter o de menor expoente para se equiparar ao de maior expoente. Assim:

$1{,}001b * 2^{-1} = 0{,}1001b * 2^0 = 0{,}01001b * 2^1$

Se simplesmente somássemos os dois binários, teríamos:

$1{,}00100b * 2^1$
$\underline{0{,}01001b * 2^1}$
$1{,}01101b * 2^1 \quad = \quad 10{,}1101b \quad = \quad 2{,}8125$

Ora, antes, vamos representar este número 0,01001b na notação IEEE simulada com 8 bits (ignoremos a característica, já que estamos trabalhando com a soma das mantissas, fixando a característica em 1):

Sinal	Sinal do Expoente	Expoente		Mantissa				
0	1	0	1	0	1	0	0	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

O último bit se perdeu! Assim, a soma efetivamente realizada será:

$1{,}0010b * 2^1$
$\underline{0{,}0100b * 2^1}$
$1{,}0110b * 2^1 \quad = \quad 10{,}110b \quad = \quad 2{,}75$

Repare que 2,75 é **diferente** de 2,8125. Essa diferença, chamada erro, se reduz bastante à medida que se trabalha com números de maior precisão (32, 64, 128 bits... e assim por diante). Entretanto, esse erro sempre existe, consistindo em uma **limitação** da representação em ponto flutuante. Adicionalmente, diversos números que possuem representação finita na base decimal (como 0,1) se tornam "dízimas periódicas" na base dois (0,1 = 0,00011001100110011...b). Isso faz com que executar 1000 somas do número 0,1, por exemplo, não levem ao valor 100, consistindo uma **limitação** da representação binária em ponto flutuante para realizar operações com números decimais!

7. REPRESENTAÇÃO DE CARACTERES

Até o momento vimos como armazenar números de diferentes tipos na memória. Mas como armazenar letras? Bem, este foi um problema que surgiu nos primórdios da computação e, por esta razão, existe uma solução padrão, que é a chamada Tabela ASCII (ASCII significa *American Standard for Computer Information Interchange*). A tabela ASCII relaciona cada valor numérico de um byte a cada um dos códigos visuais usados por nós na atividade da escrita. A tabela de conversão é apresentada na página seguinte (fonte: Wikipédia).

Observe, porém, que nem todos os caracteres são definidos por essa tabela: em especial, os caracteres acentuados estão faltando. Mas não são apenas estes: também não estão presentes os caracteres japoneses, chineses, russos... dentre tantos outros.

Por essa razão, atualmente existem diversas outras "tabelas de código de caracteres" ou "páginas de código de caracteres" (do inglês *codepage*), que extendem a tabela abaixo indicando os símbolos faltantes aos códigos livres (não especificados pela tabela ASCII). Entretanto, com a grande troca de arquivos entre pessoas de países diferentes, isso começou a causar alguma confusão.

Foi assim que surgiram então os códigos Unicode, que são versões alternativas e universais à tabela ASCII. O padrão UTF (Unicode Transformation Format) define várias tabelas, sendo as mais conhecidas e usadas as tabelas UTF-8 e UTF16. A tabela UTF-8 define 256 caracteres, como a tabela ASCII, mas com um padrão que tenta alocar a grande maioria dos símbolos usados pela maioria das línguas. Já o UTF-16 define 65.536 caracteres, englobando a grande maioria dos caracteres de todas as línguas. Existe ainda o padrão UTF-32, com capacidade para definir até 4 bilhões de caracteres, mas que é muito pouco usado.

Binário	Decimal	Hexa	Glifo
0010 0000	32	20	
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binário	Decimal	Hexa	Glifo
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binário	Decimal	Hexa	Glifo
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

Unidade 6: Memórias

Prof. Daniel Caetano

Objetivo: Compreender os tipos de memória e como elas são acionadas nos sistemas computacionais modernos.

INTRODUÇÃO

Nas aulas anteriores foram apresentadas diversas maneiras de interpretar os bits na memória; essa compreensão é de extrema importância, mas não responde às perguntas: o que é, como funciona e como é acessada a memória?

O objetivo desta aula é apresentar uma introdução sobre os diferentes tipos de memórias existentes no computador, além de apresentar a forma com que a memória - e outros dispositivos - são acessados, através do **barramento de sistema**, que será detalhado em aulas posteriores.

1. O QUE É A MEMÓRIA?

Em palavras simples, a memória é um dispositivo físico capaz de armazenar e recuperar uma configuração elétrica em um "conjunto de fios". Uma vez que essa configuração elétrica estabelece um padrão de bits, ligados ou desligados, é possível dizer que a memória armazena e recupera **dados**.

Olhando como uma caixa preta, a memória é bastante simples. Observe a Figura 1.

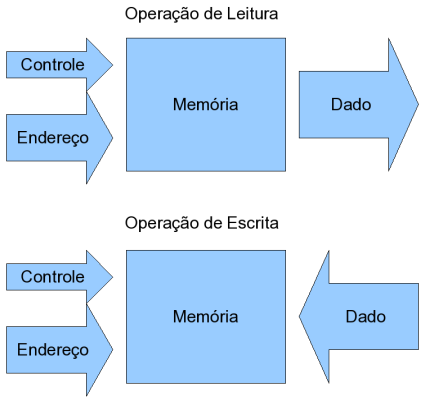


Figura 1: Operações de leitura e escrita na memória

Simplificadamente, o dispositivo memória recebe sinais de controle - que indicam se a operação é de leitura ou escrita na memória - e um endereço. Se a operação for de leitura, o

dispositivo memória responde emitindo o dado armazenado no endereço em questão; se a operação for de escrita, o dispositivo recebe o dado e o armazena na posição de memória indicada pelo endereço.

Apesar de ser um dispositivo de funcionamento aparentemente simples, as memórias são possivelmente os dispositivos com maior diversidade de implementações. Isso ocorre porque há diversas características que buscamos nas memórias como velocidade e capacidade, que não são atendidas plenamente por nenhum dos tipos de memória existente.

Há memórias que são rápidas, mas se forem desligadas perdem os dados armazenados e são muito caras; há memórias que são de velocidade média e possuem um preço razoável, mas se forem desligadas também perdem os dados. Há memórias muito baratas e que mantém a informação quando são desligadas, mas que são muito lentas...

Além disso, nem todas as memórias fornecem dados do "tamanho" que o processador "quer". O tamanho dos dados lidos pelo processador é denominado **palavra** e pode ter diferentes tamanhos, como 8, 16, 32, 64, 128bits... dentre outros. Isso é o que determina, em geral, a expressão "processador de 64 bits": o tamanho do dado que ele manipula diretamente. Algumas memórias simplesmente fornecem os dados em blocos maiores do que uma palavra, exigindo algum *"malabarismo*" para permitir seu uso direto com um dado processador.

Adicionalmente, para que se possa tirar máximo proveito de um sistema computacional, a velocidade da memória deve ser compatível com a velocidade do processador, de maneira que esse último não precise ficar esperando por respostas da memória por muito tempo. Em tese, considerando os processadores atuais, isso exigiria que toda a memória fosse muito rápida e, como consequência, os equipamentos seriam muito caros e praticamente não poderiam ser desligados.

Certamente esse não era um caminho viável e, por essa razão, cirou-se uma outra alternativa: usar diversos tipos de memória para obter o melhor desempenho ao menor custo.

2. HIERARQUIA DE MEMÓRIA

A quantidade de dados que um usuário medio armazena é gigantesca. Se considerarmos um servidor de uma grande empresa, essa quantidade de dados é ainda maior. Entretanto, a grande maioria desses dados **raramente** é usada pelo computador. Isso ocorre porque apenas um pequeno conjunto de programas e dados é usada rotineiramente.

Ainda assim, mesmo considerando os programas e dados que são processados com frequência, se medirmos a quantidade de tempo que o processador gasta com cada um destes bytes, veremos que a maior parte do tempo o computador está executando pequenos blocos de instruções e dados, realizando **tarefas repetidas**.

Observando este comportamento, os projetistas de *hardware* concluíram que poderiam equilibrar o custo de um equipamento se usassem um tipo de diferente de memória para cada tarefa:

a) Registradores e Memória Cache (Armazenamento Interno): Para armazenamento de curto prazo, de dados usados intensivamente pelo computador, adotam-se dispositivos de armazenamento volátil extremamente rápidos, mas de pequena capacidade devido ao **custo por bit ser muito alto**.

b) Memória Principal (Armazenamento Interno): Para armazenamento de médio prazo, de dados medianamente usados, adotam-se dispositivos de armazenamento volátil, cujo **custo por bit é médio**, proporcionando média capacidade com uma velocidade de acesso também média, já que estes dados são usados com alguma frequência.

c) Memória Secundária (Armazenamento Externo): Para armazenamento de longo prazo, de dados pouco usados, adotam-se dispositivos de armazenamento não volátil e cujo **custo por bit é baixo**, proporcionando grande capacidade, ainda que sejam lentos. A lentidão não é um problema, pois os dados aí contidos são pouco acessados.

d) Memória de Segurança (Armazenamento de Segurança): Para armazenamento de longuíssimo prazo, de dados que talvez nunca sejam necessários, adotam-se dispositivos de armazenamento não volátivo de **custo por bit extremamente baixo**, com enrome capacidade, ainda que extremamente lentos.

Estes quatro níveis formam a hierarquia de memória, lembrando que todos os dados úteis de um computador precisam estar armazenados na memória secundária, sendo transferidos para a memória principal na medida em que são necessários. Da mesma forma, os dados da memória principal são transferidos para o cache e para os registradores também na medida em que são necessários. A comunicação ocorre obedecendo a hierarquia, como pode ser visto na Figura 2.

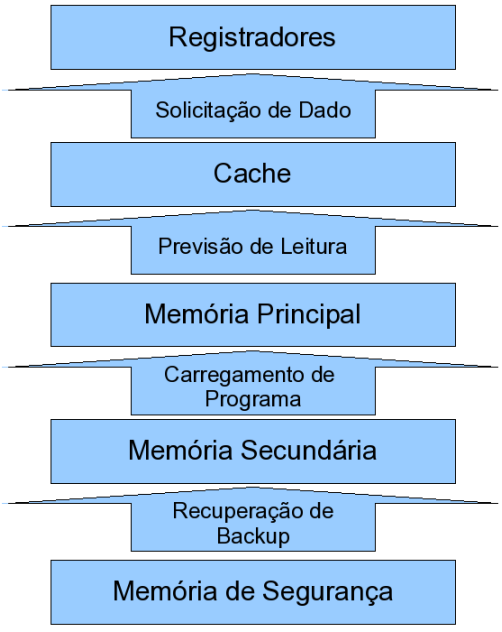


Figura 2: Situações de transferência de dados entre memórias (leitura)

3. TIPOS DE MEMÓRIA

As memórias utilizadas em cada uma destas camadas da hierarquia podem ser construídas com diferentes tecnologias. A diferenciação mais básica está entre os tipos RAM e ROM, mas existem diversos subtipos. Algumas delas estão descritas no quadro a seguir.

Tipo	Categoria	Apagamento	Escrita	Volatilidade	Palavra / Bloco	Interna / Externa	Velocidade	Usos	Custo por Bit
Memória de Acesso Aleatório Estática (SRAM)	Escrita e Leitura	Eletrica-mente	Eletrica-mente	Volátil	Bytes	Interna	Variada (pode ser tão rápida quanto o processador)	Registradores, cache, memória principal	De muito alto a alto
RAM Dinâmica (DRAM)	Escrita e Leitura	Eletrica-mente	Eletrica-mente	Volátil	Bytes	Interna	Média	Memória principal	Médio
Memória apenas de Leitura (ROM)	Apenas de leitura	Impossível	Máscaras	Não volátil	Bytes	Interna / Externa	Média para Rápida	Memória principal ou secundária	Baixo
ROM Programável (PROM)	Apenas de leitura	Impossível	Eletrica-mente	Não volátil	Bytes	Interna / Externa	Média para Rápida	Memória principal ou secundária	Baixo
PROM Apagável (EPROM)	Principal-mente de leitura	Luz UV	Eletrica-mente	Não volátil	Bytes	Interna / Externa	Rápida para leitura	Memória principal ou secundária	Baixo
EPROM Eletricamente apagável (EEPROM)	Principal-mente de leitura	Eletrica-mente	Eletrica-mente	Não volátil	Bytes ou Blocos	Interna / Externa	Rápida para leitura	Memória principal ou secundária	Médio
Memória Flash	Principal-mente de leitura	Eletrica-mente	Eletrica-mente	Não volátil	Blocos	Externa	Média para Lenta	Memória secundária	Médio
Disco Magnético	Escrita e Leitura	Magneti-camente	Magneti-camente	Não volátil	Blocos	Externa	Lenta	Memória secundária	Baixo
Disco Óptico	Leitura (e, opcio-nalmente, Escrita)	Óptica	Óptica	Não volátil	Blocos	Externa	Muito lenta	Memória secundária e de segurança	Muito baixo
Fita Magnética	Escrita e Leitura	Magneti-camente	Magneti-camente	Não volátil	Bytes	Externa (no passado, interna)	Extremamente Lenta	Memória secundária e de segurança (no passado, principal)	Extrema-mente baixo

Das memórias voláteis, qual a diferença entre a memória SRAM e DRAM? Bem, a SRAM é um dispositivo que basta estar ligado para preservar seus dados; a DRAM, por outro lado, exige que de tempos em tempos seja feita uma "simulação de leitura" em cada região da memória, para garantir que ela não seja perdida, em um processo chamado ***refresh***.

Como a maioria das memórias permite apenas **um acesso** por vez, isto é, ela só permite que um endereço de memória seja acessado de cada vez, durante o momento em que o ***refresh*** está sendo executado a memória DRAM fica **indisponível** para o processamento e, por essa razão, ela acaba sendo, no geral, mais lenta que a SRAM.

Projetar circuitos com SRAM é muito mais simples do que com DRAM; entretanto, a diferença de preço entre ambas faz com que, em geral, se a velocidade de uma SRAM não é necessária, os projetistas adotem o uso de DRAMs.

4. ACESSO A MEMÓRIA

Desde o início do curso é comentado que as partes do computador se comunicam por "fios". De fato, é isso que ocorre, embora esses "fios" muitas vezes sejam trilhas minúsculas em uma placa de circuito impresso.

Por exemplo, no caso da memória, apresentamos na Figura 1 "setas" que indicavam o fluxo de informações para a memória. Na prática, essas "setas" são fios ou trilhas, pelos quais trafegam sinais elétricos: a presença de sinal é interpretada como "1" e a ausência de sinal é interpretad como "0", formando os padrões desejados. Veja o exemplo da Figura 3.

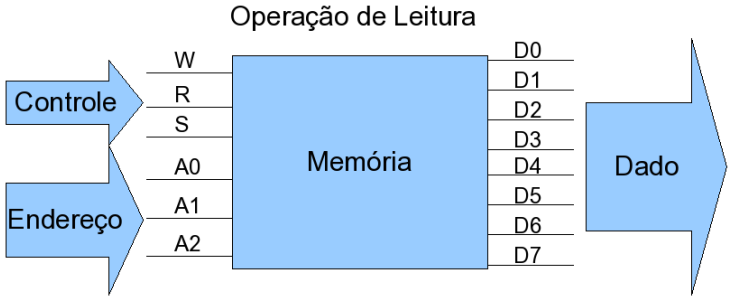


Figura 3: Nomenclatura dos "fios" que ligam a memória ao sistema

Nesta figura, os "fios" do controle foram nomeados de W (Write), R (Read) e S (Select). Estes fios controlam o funcionamento da memória. Sempre que a memória estiver sendo usada, S (ou MREQ) deverá estar com sinal em 1, indicando que a memória está selecionada. Quando S (ou MREQ) estiver em 1, os valores de W e R indicarão qual é a operação solicitada.

Quando a memória for usada para escrita, o sinal de W deve estar em 1 e R em 0; quando a memória for usada para leitura, R deverá estar em 1 e W em 0. O comportamento se S (MREQ), R e W estiverem todos simultaneamente em 1 é indefinido.

Os "fios" de endereço foram nomeados de A0 a A2 (A vem de Address). Isso significa que essa memória tem 3 bits de endereçamento, permitindo acesso a 8 dados (2^3). Em outras palavras, **temos 8 posições de memória**.

Os "fios" para representar os dados foram nomeados de D0 a D7 (D vem de Data). Isso significa que cada uma das posições de memória tem 8 bits (1 byte), proporcionando o armazenamento de 256 valores (2^8) diferentes em cada posição de memória.

Assim, como temos 8 posições de 1 byte cada, esta é uma memória de 8 bytes.

Vamos analisar agora como funcionaria a memória no momento de uma leitura. Num primeiro momento, o circuito do computador irá indicar nos "fios" do controle que ele deseja

ler a memória e, nos "fios" de endereço, vai indicar que deseja ler um determinado endereço. Suponhamos que o endereço a ser lido seja o endereço 6, ou seja, 110b. Observe a configuração na Figura 4.

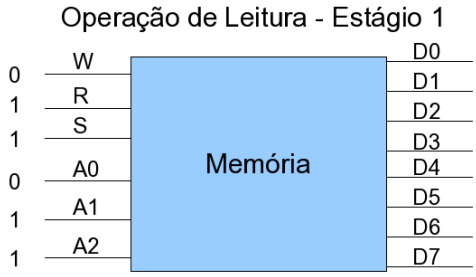


Figura 4: Configuração para a leitura do endereço de memória 6 (110b)

O circuito do computador coloca essa configuração elétrica nos "fios" de entrada da memória e, alguns instantes depois, a memória configura eletricamente os "fios" de dados (D0 a D7) com a informação que nela está armazenada, conforme indicado na Figura 5.

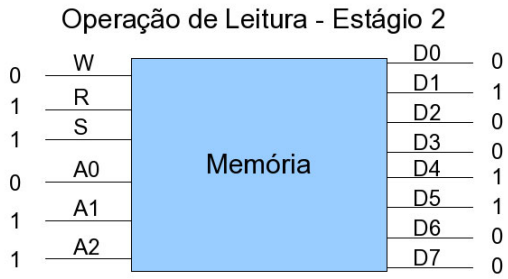


Figura 5: Resposta da memória à leitura do endereço de memória 6 (110b)

Ou seja: o dado armazenado na posição 6 da memória é: 00110010b, ou, interpretando como um decimal inteiro, 50.

Por terem funções **muito** distintas, cada um destes conjuntos de fios recebem nomes específicos. Os "fios" que controlam os dispositivos ligados ao computador são chamados de **barramento de controle**. Os "fios" que configuram endereços de memória e outros dispositivos são chamados de **barramento de endereços** e, finalmente, os "fios" que servem para a troca de dados entre os vários dispositivos são chamados de **barramento de dados**.

O exemplo desta aula foi feito com uma unidade de memória mas, de maneira geral, o procedimento é parecido para qualquer dispositivo que seja ligado no sistema computacional moderno.

Isso ocorre porque os computadores são projetados segundo um paradigma de arquitetura denominado "barramento de sistema", que será visto na aula que vem.

6. BIBLIOGRAFIA

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

Unidade 7: Barramento de Sistema

Prof. Daniel Caetano

Objetivo: Compreender a estrutura de comunicação dos sistemas computacionais modernos.

INTRODUÇÃO

Na aula anterior foi apresentada a memória e seu mecanismo de acionamento através dos barramentos de endereço, controle e dados. Mas o que são esses barramentos? Para que servem em um computador?

O objetivo desta aula é apresentar uma introdução ao modelo de barramentos de sistema e ressaltar sua importância para o funcionamento dos computadores modernos.

1. RECORDANDO...

Como foi visto na manipulação da memória, há "conjuntos de fios" com função muito diferenciada: os "fios" que controlam os dispositivos ligados ao computador são chamados de **barramento de controle**. Os "fios" que configuram endereços de memória e outros dispositivos são chamados de **barramento de endereços** e, finalmente, os "fios" que servem para a troca de dados entre os vários dispositivos são chamados de **barramento de dados**.

O exemplo da aula passada foi feito com uma unidade de memória mas, de maneira geral, o procedimento é parecido para qualquer dispositivo que seja ligado no sistema computacional moderno. Isso porque os computadores modernos são projetados segundo um paradigma de arquitetura denominado "barramento de sistema", que será visto com mais detalhes a seguir.

2. BARRAMENTO DE SISTEMA

O Modelo de Barramento de Sistema é composto de três componentes:

1. Unidade de Entrada e Saída - usada pela CPU para receber e fornecer dados (e instruções) ao usuário.
2. CPU - responsável por coordenar todo o funcionamento do computador.
3. Unidade de Memória - responsável por armazenar dados e instruções a serem utilizadas pela CPU.

Observe pela Figura 1 que agora todas as unidades estão interconectadas, permitindo assim algumas caraterísticas interessantes como uma unidade de entrada ler ou escrever na

memória sem a necessidade de intervenção da CPU (característica chamada DMA - Direct Memory Access).

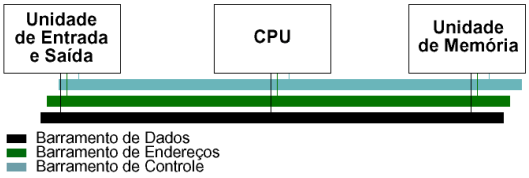


Figura 1: Modelo de Barramento de Sistema

Cada barramento consiste de um determinado conjunto de fios interligando os componentes do sistema. Quando dizemos, por exemplo, que um sistema tem 32 bits no barramento de dados, isso quer dizer que existem 32 fios para transmitir dados, e eles estão interligando todas as unidades do computador.

Um computador moderno tem, normalmente, três barramentos:

1. **Barramento de Dados** - para transmitir dados.
2. **Barramento de Endereços** - para identificar onde o dado deve ser lido/escrito.
3. **Barramento de Controle** - para coordenar o acesso aos barramentos de dados e endereços, para que não ocorra conflitos (do tipo vários periféricos querendo escrever na memória ao mesmo tempo).

Neste modelo, para a CPU ler a memória, ocorre (simplificadamente) o seguinte mecanismo:

1. CPU verifica se há uso da memória por algum dispositivo. Se houver, espera.
2. CPU indica no Barramento de Controle que vai ler a memória (impedindo que outros dispositivos tentem fazer o mesmo). Isso faz com que a memória se prepare para receber um endereço pelo barramento de endereços.
3. CPU coloca no barramento de endereços qual é o endereço de memória a ler.
4. Memória lê barramento de endereços e devolve o dado no barramento de dados.
5. CPU lê o dado solicitado no barramento de dados.

Repare que tudo isso existe uma coordenação temporal enorme, já que os barramentos não são "canos" onde você joga a informação e seguramente ela chega do outro lado. Na verdade, a atuação é mais como um sinal luminoso: quem liga uma lâmpada precisa esperar um tempo para que a outra pessoa (que recebe o sinal) veja que a lâmpada acendeu. Entretanto, a pessoa que liga a lâmpada não pode ficar com ela acessa indefinidamente, esperando que o recebedor da mensagem veja a lâmpada. Se isso ocorresse, a comunicação seria muito lenta.

Assim, tudo em um computador é sincronizado em intervalos de tempo muito bem definidos. Estes intervalos são os **ciclos de clock**. Assim, quando a memória coloca um dado no barramento de endereços, por exemplo, ela o faz por, digamos, 3 ciclos de clock. A CPU

precisa ler este dado nos próximos 3 ciclos de clock ou, caso contrário, a informação nunca será recebida pela CPU e temos um computador que não estará funcionando corretamente (um computador que não respeite essa sincronia perfeita não costuma sequer ser capaz de passar pelo processo de inicialização; quando a sincronia falha em algumas situações apenas, o resultado pode ser travamentos aparentemente sem explicação).

Como é possível notar, se a sincronia tem que ser perfeita e existe um controle, quando há um único barramento para comunicação (entrada e saída - relativo à CPU) de dados, há uma limitação: ou a CPU recebe dados ou a CPU envia dados, nunca os dois ao mesmo tempo. Em algumas arquiteturas específicas são feitos barramentos distintos - um para entrada e um para saída, de forma que entrada e saída possam ocorrer simultaneamente.

Além disso, o acesso a dispositivos pode ser de duas maneiras. Algumas arquiteturas exigem que os dispositivos sejam **mapeados em memória**, ou seja, para enviar uma informação a um dispositivo deste tipo, a CPU deve escrever em um (ou mais) endereço(s) de memória específico(s). Para receber informações do dispositivo, a CPU deve ler um (ou mais) endereço(s) de memória específico(s). Outras arquiteturas, mais flexíveis, possuem dois tipos de endereçamento: um endereçamento de memória e outro de entrada e saída (I/O). Neste caso, os dispositivos podem tanto ser mapeados em memória como **mapeados em portas** de I/O. O uso de mapeamento de dispositivos em portas de I/O permite que todo o endereçamento de memória esteja disponível, de fato, para o acesso à memória.

2.1. Arquitetura de Barramento Simples

Como já foi apresentado, em determinado momento os arquitetos de computador perceberam que seria interessante se todos os periféricos pudessem conversar entre si, sem intermediários. A primeira forma com que imaginaram isso ocorrendo seria através da interligação direta de todos os dispositivos entre si, como no diagama esquerdo da Figura 2.

Entretanto, não demorou para perceberem que isso traria problemas sérios, como por exemplo a quantidade de fios necessários nas placas para interligar todos os circuitos. Por esta razão, criou-se a idéia de **barramento**, que é um caminho comum de trilhas (fios) de circuito que interligam simultaneamente diversos dispositivos, em paralelo, como no diagrama direito da Figura 2.

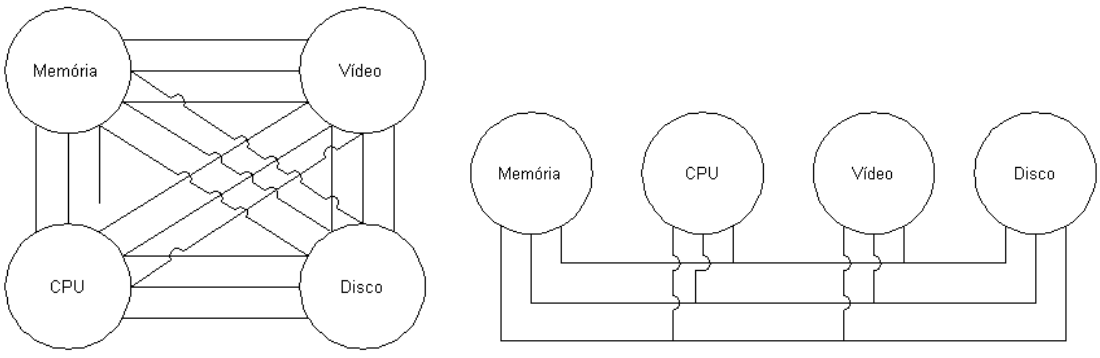


Figura 2: Elementos com ligação direta (esquerda) e através de barramento (direita)

O barramento é, fisicamente, um conjunto de fios que se encontra na *placa mãe* do computador, sendo um conjunto de fios que interliga a CPU (ou seu soquete) com todos os outros elementos. Em geral, o barramento (ou parte dele) pode ser visto como um grande número de trilhas de circuito (fios) paralelas impressas na placa mãe.

Quando se fala simplesmente em "barramento", na verdade se fala em um conjunto de três barramentos: o barramento de dados, o barramento de endereços e o barramento de controle. Alguns autores definem ainda o barramento de energia (cujas funções alguns incluem no barramento de controle).

Os barramentos de dados e endereços são usados para a troca de informações entre os diversos dispositivos e o barramento de controle é usado para a gerência do *protocolo de barramento*, sendo este o responsável por uma comunicação ordenada entre os dispositivos.

Este protocolo pode ser *síncrono* ou *assíncrono*. No caso do barramento síncrono, existe a necessidade de um circuito que forneça a cadência de operação, chamado de oscilador (ou relógio, ou *clock*, em inglês). Este dispositivo envia pulsos elétricos por um dos fios do barramento de controle, que podem ser entendidos como 0s e 1s, sendo estes pulsos utilizados por todos os outros dispositivos para *contar tempo* e executar suas tarefas nos momentos adequados. Observe na Figura 8 (MURDOCCA, 2000) o diagrama do sinal de *clock* em um barramento de 100MHz:

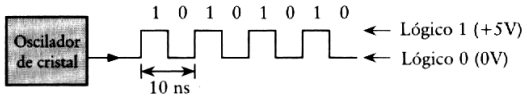


Figura 8: Sinal de *clock* (ao longo do tempo) de um barramento de 100 Mhz

Entretanto, a transição entre o sinal lógico 0 (0V) para 1 (5V) não é instantâneo e, na prática, é mais comum representar esta variação como uma linha inclinada, formando um trapézio.

O barramento freqüentemente opera a uma freqüência (*clock*) mais lenta que a CPU; além disso, uma operação completa (transação) no barramento usualmente demora vários ciclos deste *clock* de barramento. Ao conjunto dos ciclos de uma transação completa do barramento chamamos de ***ciclo do barramento***, que tipicamente compreendem de 2 a 5 períodos de *clock*.

Durante um ciclo do barramento, sempre existe **mestre do barramento**, que é o circuito que está responsável pelo barramento de controle. Todos os outros circuitos estarão em uma posição de **escravos**, isto é, todos os outros circuitos *observarão* os barramentos de controle, endereços e dados e atuarão quando solicitados.

3. BARRAMENTO SÍNCRONO

Os barramentos síncronos possuem algumas limitações, mas são os mais utilizados pela facilidade de implementação e identificação de problemas e erros (*debug*), como veremos mais adiante. O barramento síncrono funciona com base em uma temporização, definida pelo *clock do barramento* (ou relógio do barramento). O funcionamento do barramento fica sempre mais claro com um exemplo e, por esta razão, será usado o exemplo de uma operação de leitura de memória por parte da CPU.

O primeiro passo é a CPU requisitar o uso do barramento, ou seja, a CPU deve se tornar o circuito *mestre* do barramento. Por hora, será feita a suposição que a CPU conseguiu isso. Dado que ela é a mestra do barramento, a primeira coisa que ela fará, no primeiro ciclo T_1 desta transação, é indicar o endereço desejado no barramento de endereços (através do registrador MAR).

Como a mudança de sinais nas trilhas do circuito envolve algumas peças como capacitores, que demoram um tempo a estabilizar sua saída, a CPU aguarda um pequeno intervalo (para o sinal estabilizar) e então, ainda dentro do primeiro ciclo T_1, indica dois sinais no barramento de controle: o de requisição de memória (MREQ, de *Memory REQuest*), que indicará **com quem** a CPU quer trocar dados (neste caso, a memória), e o sinal de leitura (RD, de *ReaD*), indicando qual a operação que deseja executar na memória, no caso a operação de leitura.

Neste ponto, a CPU precisa esperar que a memória perceba que é com ela que a CPU quer falar (através do sinal MREQ) e que é uma leitura (através do sinal RD). Recebendo estes dois sinais, a memória irá usar o sinal do barramento de endereços para escolher um dado e irá colocar este dado no barramento de dados. Ocorre que a memória demora um certo tempo para fazer isso; por esta razão, a CPU espera em torno de um ciclo inteiro do barramento (T_2) e apenas no terceiro ciclo (T_3) é que a CPU irá ler a informação no barramento de dados (através do registrador MBR).

Assim que a CPU termina de adquirir o dado, ela desliga os sinais de leitura (RD) e de requisição de memória (MREQ), liberando o barramento de controle em seguida. Todo esse processo ao longo do tempo pode ser visto no diagrama da Figura 9.

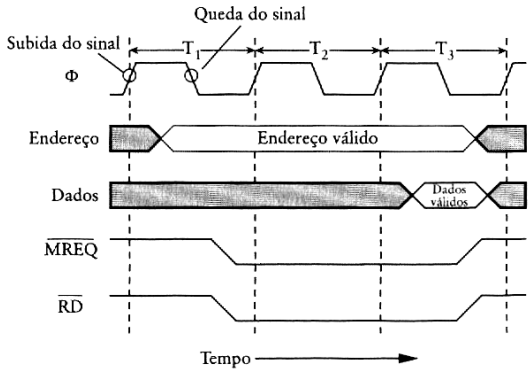


Figura 9: Um ciclo de barramento para leitura de memória
(fonte: Tanenbaum, 1999, apud Murdocca, 2000)

É importante ressaltar simbologia deste diagrama. A letra grega ϕ indica o sinal do *clock*. MREQ e RD possuem um traço em cima. Este traço indica que o acionamento ocorre quando a linha está em *zero*. Ou seja: se a linha MREQ estiver com sinal alto (1), a memória não responde. Quando MREQ estiver com o sinal baixo (0), a memória responderá. Ou seja: o traço em cima do nome indica que o acionamento do sinal é *invertido* com relação àquilo que se esperaria normalmente.

Anteriormente foi citado que este tipo de barramento tem uma limitação. Esta limitação é que seu protocolo é rigidamente ligado aos ciclos de *clock*. A CPU realiza sua operação em espaço de tempo pré-determinados e a memória precisa corresponder, em termos de velocidade. Se ela não corresponder, a CPU simplesmente lerá informações incorretas como se fossem corretas (qualquer "lixo" existente no barramento de dados no momento da leitura, em T_3, será lido como sendo um dado legítimo).

Por outro lado, não há ganho algum ao se substituir uma memória que atende aos requisitos de desempenho exigidos pela CPU por outra memória mais rápida: a CPU continuará demorando o mesmo número de ciclos de *clock* do barramento pera recuperar os dados: ela sempre irá esperar todo o ciclo T_2 sem fazer nada e irá ler a informação apenas no ciclo T_3 - mesmo que a memória seja rápida o suficiente para transmitir a informação já no ciclo T_2.

4. BARRAMENTOS EM PONTES

Como dito anteriormente, uma forma de superar as limitações do barramento síncrono (que exige, por exemplo, que CPU e dispositivos operem sobre o mesmo *clock*), é através do uso de circuitos de compatibilização. Estes circuitos são chamados de *bridges* ou, em português, **pontes**.

Na arquitetura Intel atual, o circuito que faz essa intermediação primária é chamada *North Bridge*. O North Bridge é uma espécie de benjamim adaptador: é ligado na CPU através do Barramento Frontal (*Front Side Bus*), que trabalha na velocidade da CPU, e também é ligado ao Memory Bus, que trabalha na velocidade das memórias.

Em equipamentos com conector do tipo AGP (*Advanced Graphics Port*), o North Bridge também conecta a CPU e a Memória com o barramento de gráficos, que trabalha na velocidade admitida pela placa de vídeo (daí denominações do tipo 1x, 2x, 4x, 8x...).

Uma outra parte do North Bridge é ligado ao South Bridge, que faz a ponte com os barramentos dos conectores internos (slots), sejam eles PCI-X, PCI ou ISA, cada um deles trabalhando em uma freqüência (*clock*) específicos. Um esquema de uma arquitetura Intel recente pode ser visto na figura 11.

Na figura, o North Bridge está claro; o South Bridge, não (a figura sugere North e South Bridge integrados).

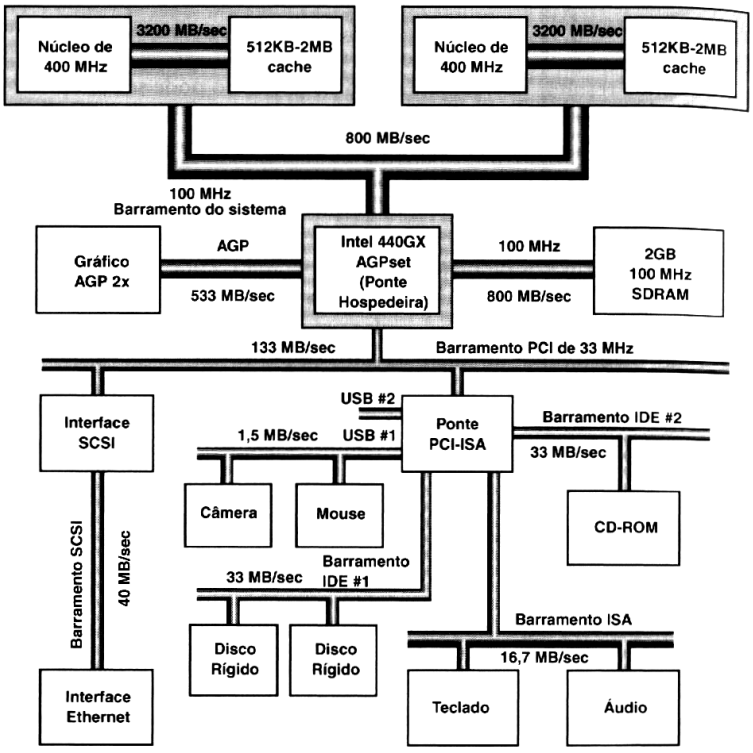


Figura 11: Arquitetura de Barramento em Pontes
(fonte: Intel apud Murdocca, 2000)

6. BIBLIOGRAFIA

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

Unidade 8: Dispositivos de Entrada e Saída

Prof. Daniel Caetano

Objetivo: Conhecer alguns dos dispositivos de entrada e saída mais comuns, além de compreender a lógica de comunicação entre os dispositivos de E/S mais comuns.

Bibliografia:

- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.
- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

INTRODUÇÃO

Grande parte da funcionalidade de um computador se deve à sua capacidade de se comunicar com dispositivos de entrada e saída. De fato, é usual que se dê o nome de "computador" apenas a dispositivos que realizem operações e que possuam pelo menos uma unidade de entrada e uma de saída.

Uma vez que os dispositivos existentes são os mais variados e, em geral, possuem uma velocidade de comunicação **muito inferior** à da memória, com **tamanhos de palavras** usualmente **distintos** daqueles trabalhados pelo computador, não é praticável instalá-los no mesmo barramento de alta velocidade da memória, o que faz com que normalmente possuam um barramento diferenciado, acessado através de uma das pontes, como visto em aulas anteriores.

Além disso, a maneira com que as informações destes dispositivos são transferidas para a memória pode variar em nível de complexidade, do mais simples e lento ao mais complexo e eficiente. Nesta aula serão apresentados os três modos de comunicação existentes nos computadores modernos. A maior diferença entre os três é o nível de interferência da CPU no processo de comunicação de um dado dispositivo com a memória.

No caso de maior intervenção da CPU no processo, é dado o nome de **Entrada e Saída Programada** (ou ***polling***). No caso de menor intervenção, é dado o nome de **Acesso Direto à Memória** (ou ***Direct Memory Access***, **DMA**). No caso intermediário, é dado o nome de **Entrada e Saída Controlada por Interrupção**.

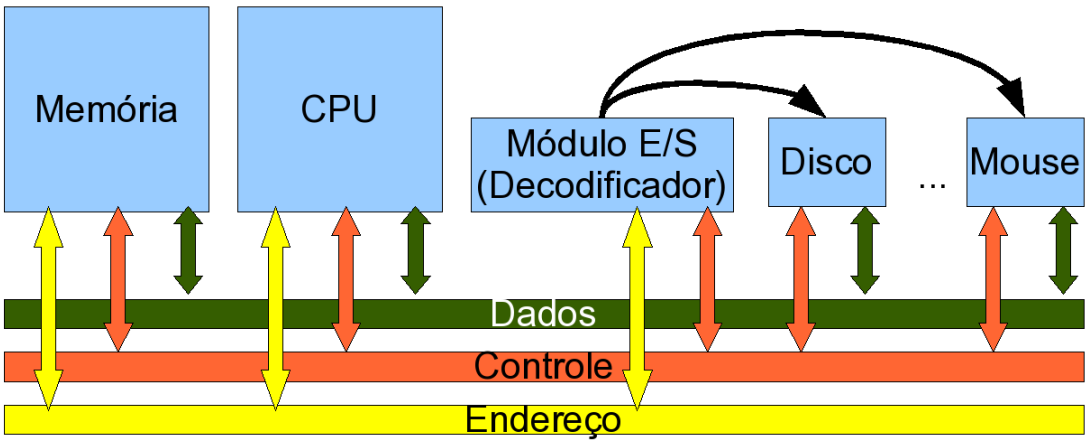
Após a visão geral sobre os métodos de transferência de dados entre dispositivos e memória, serão comentados alguns dos dispositivos mais comuns atualmente.

1. COMO SE ATIVA UM DISPOSITIVO DE E/S?

Um dispositivo de E/S, em certo aspecto, é bastante difernente da memória. Enquanto a memória recebe um sinal de ativação e ela é preparada para responder a todos os endereços (recebe os sinais A0 a AN), os dispositivos de E/S tem um funcionamento diferenciado, usualmente tendo apenas um (ou alguns poucos) sinal de ativação (A0).

Para diferenciar entre esses comportamentos, o processador tem, usualmente, formas alternativas para ativar o uso da a memória ou dos dispositivos de E/S. Em geral, o processador central tem um pino chamado MREQ (do barramento de controle) que é ligado diretamente ao seletor de ativação da memória e, no caso da memória, o mesmo ocorre com o barramento de endereços inteiro.

No caso dos dispositivos de E/S, a CPU os ativa por meio de um pino chamado IORQ (também do barramento de controle), que junto com o barramento de endereços aciona um (ou vários) dispositivo chamado **decodificador** que, dependendo da configuração do endereço, acionará um dispositivos adequado. Observe a figura abaixo, simplificada com um único barramento:



Note que este desenho é um exemplo, apenas para indicar como o **módulo de E/S** "interfere" na ativação do dispositivo, em uma funcionalidade chamada de **decodificação de endereço de E/S**.

Alguns dispositivos, como o vídeo e o *hard disk*, não possuem qualquer contato direto com o barramento do sistema, ficando completamente interligados ao módulo de E/S, que assume a responsabilidade de gereciar os três barramentos. Esse tipo de móldulo de E/S é chamado **interface controladora de dispositivo**. Cada dispositivo pode ter seu próprio controlador ou é possível ter um controlador para vários dispositivos.

Como exemplo desses módulos E/S temos a placa de vídeo ou a placa SCSI/SATA. Em alguns casos, o módulo decodificador está fisicamente integrado ao dispositivo, de maneira que não é possível observá-lo.

Note que o que se chama de **interface controladora** não é apenas um seletor e decodificador de endereços; usualmente ela tem muito mais recursos, permitindo, por

exemplo, que um determinado dispositivo "tome o controle" dos barramentos temporariamente, permitindo que o dispositivo converse diretamente com a memória (comum) ou com outros dispositivos (incomum).

A partir da próxima seção estudaremos quais são as formas com que a comunicação com o dispositivo pode ocorrer.

2. ENTRADA E SAÍDA POR *POLLING*

No esquema chamado *polling*, a CPU é responsável por todo o controle de transferências de dados de dispositivos. Isso significa que ela é responsável não só pela transferência de informações em si, mas também pela verificação constante dos dispositivos, para saber se algum deles tem dados a serem transferidos.

Isso significa que, de tempos em tempos, a CPU faz a seguinte 'pergunta', seqüencialmente, a todos os dispositivos conectados: "Você tem dados para serem transferidos para a memória?".

Quando algum dispositivo responder "sim", a CPU faz a transferência e continua perguntando aos outros dispositivos em seguida. Quando nenhum dispositivo necessitar de transferências, a CPU volta a fazer o que estava fazendo antes: executar um (ou mais) programas. Depois de algum tempo, ela volta a realizar a pergunta para todos os dispositivos novamente.

> Uma analogia que costuma ajudar a compreender a situação é a do garçom e a do cliente em um restaurante. No sistema de *polling* o garçom é a CPU e o cliente é o dispositivo. O cliente tem que esperar pacientemente até o garçom resolver atendê-lo e, enquanto o garçom atende a um cliente, ele não pode realizar qualquer outra tarefa. Depois que o cliente fez o pedido, ele ainda tem que esperar, pacientemente, o garçom trazer a comida.

Não é difícil ver que este sistema tem **três** problemas fundamentais:

a) A CPU gasta uma parcela considerável de tempo de processamento só para verificar se algum dispositivos tem dados a serem transferidos para a memória.
b) A CPU gasta uma parcela considerável de tempo de processamento apenas para transferir dados de um dispositivo para a memória.
c) Se um dispositivo precisar de um atendimento "urgente" (porque vai perder dados se a CPU não fizer a transferência imediata para a memória, para liberar espaço no dispositivo), não necessariamente ele terá.

A primeira questão é considerada um problema porque muitas vezes a CPU perde tempo perguntando para todos os dispositivos e nenhum deles tem qualquer dado a ser transferido. É tempo de processamento totalmente disperdiçado.

A segunda questão é considerada um problema porque cópia de dados é uma tarefa que dispensa totalmente a capacidade de processamento: é uma tarefa que mobiliza toda a

CPU mas apenas a Unidade de Controle estará trabalhando - e realizando um trabalho menos nobre. Adicionalmente, para a transferência de um dado do dispositivo para a memória, tendo que passar pela CPU, o barramento é ocupado duas vezes pelo mesmo dado, ou seja, são realizadas duas transferências: dispositivo=>CPU e depois CPU=>Memória. Isso acaba sendo um "retrocesso", transformando o barramento em um modelo de Von Neumann simples.

A terceira questão é considerada um problema porque, eventualmente, dados serão perdidos. Adicionalmente, se o dispositivo em questão precisar **receber** dados para tomar alguma atitude, problemas mais sérios podem ocorrer (como uma prensa hidráulica controlada por computador causar a morte de uma pessoa por falta de ordens em tempo hábil).

Por outro lado, o sistema de transferência por *polling* é de implementação muito simples, o que faz com que muitas vezes ele seja usado em aplicações onde os problemas anteriormente citados não são totalmente relevantes.

3. ENTRADA E SAÍDA POR INTERRUPÇÃO

No esquema chamado de entrada e saída por interrupção, a CPU fica responsável apenas pelas transferências em si. Isso significa que ela não tem que verificar os dispositivos, para saber se há dados a serem transferidos.

Mas se a CPU não faz a verificação, como ela vai perceber quando uma transferência precisa ser feita? Simples: o dispositivo dispara um sinal do barramento de controle chamado "Interrupção" (chamado de IRQ - *Interrupt ReQuest*). Quando a CPU percebe este sinal, ela sabe que algo precisa ser feito com algum dispositivo; normalmente uma transferência de dados (seja de entrada ou saída).

> Voltando a analogia do restaurante, o sistema com interrupções seria o fato de o cliente possuir uma sineta que, ao tocar, o garçom viria o mais rapidamente possível para atender ao cliente

Há sistemas em que há mais dispositivos que interrupções. Neste caso, o sistema ainda terá que fazer *polling* para saber qual foi o dispositivo que solicitou atenção; entretanto, o polling será feito somente quando ***certamente*** um dispositivo precisar de atenção da CPU (uma entrada ou saída de dados). Desta forma, elimina-se o problema de tempo perdido fazendo *pollings* quando nenhum dispositivo precisa de transferências.

A forma mais eficiente, entretanto, é quando temos pelo menos uma interrupção por dispositivo, de forma que a CPU saiba sempre, exatamente, qual é o dispositivo que está solicitando atenção e nenhum tipo de *polling* precise ser feito.

Essa característica resolve os problemas a) e c) existentes no sistema de *polling* puro, embora o problema b), relativo ao tempo de CPU gasto com as transferências em si, ainda esteja presente.

Entretanto, sempre que lidamos com sinais no barramento, temos que levantar uma questão: e se dois ou mais dispositivos solicitarem uma interrupção ao mesmo tempo? Normalmente os sistemas com várias interrupções possuem vários **níveis de prioridade de interrupção** (*interrupt level*). A CPU sempre atenderá a interrupção de maior prioridade primeiro (normalmente de "número" menor: IRQ0 tem mais prioridade que IRQ5).

Existe um outro problema também: quando a CPU recebe uma IRQ, ela **sempre** para o que está fazendo, momentaneamente, para realizar outra atividade qualquer. Ao finalizar esta atividade, ela volta ao que estava fazendo antes da IRQ ocorrer. Entretanto, há alguns processos em que talvez o programador não queria interrupções - em aplicações onde o controle de tempo é crítico, as interrupções podem causar problemas graves. Nestes casos, o programador pode usar uma instrução em linguagem de máquina que desliga as interrupções (normalmente chamada DI, de *Disable Interrupts*). Obviamente ele precisa ligá-las novamente depois (usando uma instrução normalmente chamada EI, de *Enable Interrupts*).

Entretanto, ao desligar as interrupções, o programador pode, potencialmente, causar um dano grave ao funcionamento do sistema operacional, por exemplo. Os sistemas operacionais modernos usam a interrupção para realizar a troca de aplicativos em execução, na chamada "multitarefa preemptiva". Por esta razão, as arquiteturas modernas possuem pelo menos uma interrupção chamada de Interrupção Não Mascarável (NMI, de *Non Maskable Interrupt*), que não pode ser desligada, nunca.

4. ENTRADA E SAÍDA POR DMA

No esquema chamado de entrada e saída por DMA (Acesso Direto à Memória), a CPU fica responsável apenas por coordenar as transferências. Isso significa que ela não tem que verificar os dispositivos, para saber se há dados a serem transferidos e nem mesmo transferir estes dados.

Mas se a CPU não faz a verificação, como ela vai perceber quando uma transferência precisa ser feita? Como já foi visto, pela interrupção (assim, DMA pressupõe interrupções). Mas se a CPU não faz a transferência, como os dados vão parar na memória? Simples: a CPU comanda um dispositivo responsável pela transferência, normalmente chamado simplesmente de DMA. Quando a CPU perceber o sinal de IRQ, ela verifica qual a transferência a ser feita e comanda o DMA, indicando o dispositivo origem, a posição origem dos dados, a posição destino dos dados e o número de bytes a transferir. O circuito do DMA fará o resto. Quando ele acabar, uma outra interrupção será disparada, informando que a cópia foi finalizada.

> Voltando a analogia do restaurante, o sistema com DMA seria como se, ao cliente tocar a sineta, o garçom (CPU) mandasse um outro garçom auxiliar (DMA) para fazer o que precisa ser feito, e o garçom "chefe" (CPU) continuasse a fazer o que estava fazendo antes. Ao terminar seu serviço, o garçom auxiliar (DMA) avisa ao garçom chefe (CPU) que está disponível novamente.

Este sistema resolve todos os problemas a), b) e c) apresentados anteriormente.

5. DISPOSITIVOS PRINCIPAIS DE ENTRADA E SAÍDA

Os dispositivos de entrada e saída dos computadores atuais são de conhecimento geral. Entretanto, veremos alguns detalhes de seu funcionamento interno.

5.1. HardDisks

Os harddisks são, em essência, muito similares aos disk-drives e disquetes comuns; entretanto, existem diferenças.

Dentro de um harddisk existem, normalmente, vários discos (de alumínio ou vidro) fixos a um mesmo eixo, que giram em rotações que variam de 3000 a 1000 rpm (rotações por minuto). Estes discos são, assim como os disquetes, cobertos por óxido de ferro, que pode ser regionalmente magnetizado em duas direções (indicando 0 ou 1). Estas regiões são acessíveis na forma de setores de uma trilha do disco. O tamanho de bits disponíveis em cada setor varia de caso para caso, mas 512 bytes é um valor comum.

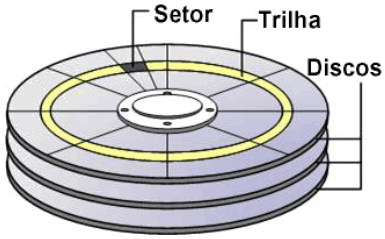


Figura 1: Discos, trilhas e setores

Para ler estes bits de um setor, existe normalmente uma cabeça de leitura para cada face de disco. No caso acima, seriam, normalmente, 6 cabeças de leitura, presas a um mesmo eixo. Apenas uma funciona por vez.

Quando é necessário ler um dado do disco, é preciso descobrir em que setor(es) ele está e ler este(s) setor(es). Um sistema de arquivos é exatamente uma maneira ordenada de guardar arquivos em diversos setores e o sistema operacional faz a parte mais chata de localização, fazendo com que os programadores possam lidar diretamente com o conceito lógico de "arquivos".

5.2. Discos Ópticos

Os discos ópticos são discos plásticos cobertos de uma camada de alumínio que é "entalhado" para representar os 0s e 1s: 1 quando não houver uma concavidade e 0 quando houver. Para a leitura há uma cabeça que é capaz de disparar um feixe de laser e também é capaz de recebê-lo de volta. O feixe de laser é disparado contra a superfície "entalhada" do disco; se encontrar uma concavidade, sofre um tipo de deflexão e a lente detecta esta deflexão como sendo um 0. Se não encontrar a concavidade, a deflexão será diferente, e a lente detectará esta outra deflexão como 1.

Os dados não são colocados em um disco óptico em um formato de trilhas e setores. Os dados em um disco óptico são colocados em forma espiral, que vai do interior para a borda do disco. O disco **não** gira com velocidade constante: gira mais lentamente à medida em que a cabeça se afasta do centro do disco.

5.3. Teclados

Independente da posição das teclas de um teclado, em geral elas são compreendidas pelo computador na forma de uma matriz com um determinado número de linhas e colunas ou simplesmente numeradas (juntando o número da linha e da coluna). Por exemplo: se a tecla "A" fica na linha 4 e coluna 2, então o número da tecla poderia ser 402, por exemplo.

Entretanto, a maioria dos teclados consegue enviar apenas um número de 7 ou 8 bits para o computador (0 a 127 ou 0 a 255), fazendo com que uma numeração tão simples não seja possível, mas a idéia permanece. Ainda assim, as teclas especiais que modificam o significado das teclas (como shift, alt, ctrl, etc) fariam com que 7 (ou 8) bits jamais fossem suficientes para transmitir todas as possíveis combinações. Por esta razão, alguns dos valores de 7 (ou 8) bits são separados para indicar o pressionamento de teclas especiais, de forma que os valores são transmitidos separadamente.

A função do "mapa de teclado" que é selecionado ao configurar um sistema operacional é justamente a de "mapear" cada número que o teclado irá enviar ao computador a uma letra correta. É por esta razão que se um teclado US-International for configurado com um mapa ABNT-2 as teclas não se comportarão como o esperado.

Mais uma vez o sistema operacional faz a maior parte do trabalho para o programador, que recebe diretamente o valor correto da tecla (letra ou número), ao invés de ter que lidar com números de teclas no teclado.

5.4. Mouses Ópticos

Os mouses ópticos são dispositvos extremamente complexos. Sua função é emitir uma luz sobre uma superfície e digitalizar a imagem da superfície onde ele se encontra. O mouse realiza essa operação milhares de vezes por segundo, comparando as imagens coletadas para detectar qual foi o **deslocamento** entre elas.

Depois de calculado este deslocamento, o mouse converte esse valor em um deslocamento proporcional que será enviado ao computador, que moverá o ponteiro do mouse na tela. É importante notar que o deslocamento na tela será tão maior quanto mais brusco for o movimento do mouse (um deslocamento rápido do mouse causa maior deslocamento na tela que o mesmo deslocamento de mouse feito lentamente).

5.5. Monitores de Vídeo LCD

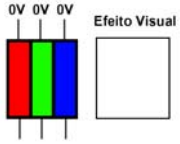
Os monitores de vídeo do tipo LCD são compostos de 3 partes:

a) um papel de fundo, onde são pintados todos os pixels, cada um deles divididos verticalmente em 3 cores: vermelho, verde e azul (RGB, Red Green Blue).
b) um sistema de iluminação (que deve iluminar uniformemente o papel de fundo)
c) uma tela de cristal líquido

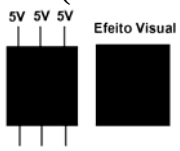
A tela de cristal líquido é composta por uma matriz de elementos de cristal líquido. O cristal líquido tem uma propriedade que é a de impedir que a luz passe em uma dada direção quando colocado sobre uma diferença de potencial suficientemente alta. A quantidade de luz que o cristal líquido deixa passar naquela direção é proporcional à diferença de potencial (campo elétrico). Pequena diferença de potencial faz com que muita luz passe; grande diferença de potencial faz com que pouca luz passe.

Cada pixel tem, então, três células de cristal líquido: uma para a região vermelha do papel, outra para a região verde do papel e outra para a região azul do papel. As cores são compostas variando o campo elétrico em cada célula de cristal líquido. Exemplos:

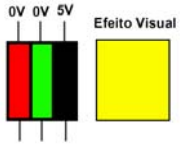
1) Campo baixo em R, G e B: cor branca (todas as cores "acesas")



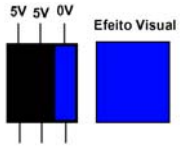
2) Campo alto em R, G, B: cor preta (todas as cores "apagadas")



3) Campo baixo em R e G e alto em B: cor amarela (só vermelho e verde "acesos")



4) Campo baixo em B e alto em R e G: cor azul (só azul "aceso")



As variações de intensidade e matiz das cores são obtidas variando a diferença de potencial em cada uma das células de cristal líquido.

6. BIBLIOGRAFIA

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

Unidade 09: Introdução aos Sistemas Operacionais

Prof. Daniel Caetano

Objetivo: Apresentar a lógica básica de funcionamento de um sistema operacional, de maneira a facilitar a compreensão do funcionamento da CPU, destacando os aspectos de interesse ao engenheiro eletrônico.

Bibliografia:

- MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.
- TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. São Paulo: Prentice Hall, 2003.
- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

* Problema: como tornar um computador útil?
* Quais as tarefas básicas que devem ser oferecidas?

Embora o uso de Sistemas Operacionais seja, hoje, parte do dia a dia da maioria das pessoas, muitas vezes sua função não é clara e é muito freqüente que algumas pessoas confundam as funções de alguns softwares aplicativos com a função do sistema operacional em si. Neste aula será apresentado os conceitos fundamentais que definem "o que é" e "para que serve" um sistema operacional, além de apresentar seus mecanismos mais básicos de funcionamento.

1. O CONCEITO DE "SISTEMA OPERACIONAL"

* Função: executar ou auxiliar a execução de tarefas básicas
    - Ex: Carregar um programa, gerenciar impressão de documento
* Sistema operacional faz tudo?
* O que é?
    - Conjunto de rotinas, em geral de baixo nível
    - Carregador de Programas x Infinidade de Funções
    - Padronização de Acesso a Recursos x Compartilhamento de Recursos

Sempre que se deseja usar um equipamento, algumas tarefas precisam ser executadas: carregar um programa na memória, enviar alguns dados para a impressora ou simplesmente escrever algo na tela. Essas estão entre algumas das funções para as quais um sistema operacional é projetado, seja para executá-las em sua completude, seja para facilitar a execução de algumas destas tarefas por parte do usuário ou de outros programas.

O nível de complexidade de um sistema operacional pode variar enormemente, assim como o nível de complexidade de utilização do mesmo. Isto significa que um sistema operacional pode fazer "mais" ou "menos" pelos usuários e pelos programas que nele irão ser executados. Independentemente de sua complexidade, um sistema operacional não passa, entretanto, de um conjunto de rotinas executadas pelo processador, como qualquer outro programa que um usuário possa desenvolver. A diferença fundamental é que, no sistema operacional, estas rotinas são, em geral, rotinas de baixo nível, rotinas que conversam diretamente com o hardware.

Um sistema operacional pode ser tão simples quanto um mero carregador de programas (praticamente o que o DOS era) ou possuir uma infinidade de funções (a maioria dos sistemas operacionais atuais.

1.1. Facilidade e Padronização do Acesso aos Recursos do Sistema

* Como facilitar o acesso a dispositivos?
    - Ex.: gravar um arquivo no HD
    - Como lidar com dispositivos de fabricantes diferentes?
* Virtualização de Dispositivos
    - Atuação como Intermediário
    - Ex.: Read / Write

Considerando que praticamente todo sistema computacional possui um grande número de dispositivos (drive de CD/DVD, monitores, impressoras, scanners etc) e que cada dispositivo tem uma operação bastante complexa, é interessante que exista uma maneira de poder utilizar tais dispositivos sem ter de se preocupar com o grande número de detalhes envolvidos no processo.

Por exemplo: o simples ato de escrever um arquivo em um *harddisk* exige um grande número de operações, envolvendo divisão do arquivo em blocos menores que caibam nos espaços livres disponíveis no disco e atualização das tabelas de diretório... para não falar em todos os detalhes envolvidos com a escrita em si, como posicionalmento da cabeça de gravação no disco, escrita propriamente dita, verificação de erros etc.

Ora, a maioria destas atividades é um processo repetitivo e metódico que não requer qualquer intervenção do usuário. As únicas informações que realmente o usuário precisa fornecer são: o nome do arquivo, quais são os dados a gravar e em que disco ele deseja que

estes dados sejam gravados. Todo o resto pode ser automatizado... e é exatamente o que o sistema operacional faz, neste caso.

Assim, o sistema operacional pode ser visto como uma interface entre o usuário/programa e o hardware, de forma a facilitar o acesso ao recursos do mesmo, como pode ser visto na figura 1.

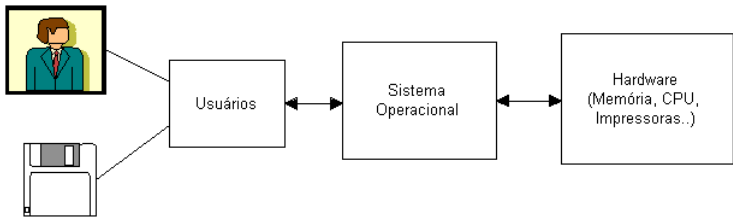


Figura 1 - Sistema Operacional como Interface entre usuários e recursos

Este tipo de "interface" que simplifica o acesso aos dispositivos (e os padroniza, em alguns casos) é também chamado de "**virtualização de dispositivos**". O caso mais comum deste tipo de virtualização são as operações "**read**" e "**write**" que, em geral, servem para ler e escrever em qualquer dispositivo, independente de seu funcionamento interno. Esta função de "virtualização" faz com que muitos usuários enxerguem o sistema operacional como uma "extensão da máquina" ou como uma *máquina extendida*.

É importante ressaltar, entretanto, que compiladores e bibliotecas fazem parte do Sistema Operacional. Embora sejam ferramentas muito importantes para o desenvolvimento, este tipo de software **não** faz parte do mesmo.

1.2. Compartilhamento de Recursos do Equipamento de Forma Organizada

* Compartilhar dispositivos?
* Vários programas tentando imprimir?
    a) Fazer o programa esperar
    b) Receber os dados e aguardar que a impressora esteja livre (spool)
* O que mais compartilhar?
    - Tela, teclado, mouse: múltiplas janelas
    - Disco, Rede?
* Múltiplos usuários
* Sistema Operacional: gerenciador de recursos!

Atualmente, o usuário médio está habituado a poder executar diversos programas ao mesmo tempo, em um mesmo equipamento. Ora, vários destes softwares podem precisar utilizar um mesmo dispositivo simultaneamente e, em geral, cada dispositivo só pode executar uma tarefa de cada vez. Como resolver este problema?

Mais uma vez surge uma das responsabilidades do sistema operacional, que é a de controlar o acesso a dispositivos de forma organizada, de maneira que os usuários deste sistema (no caso, softwares) possam compartilhar recursos. Assim, se um software está usando a impressora e um segundo deseja utilizá-la, o sistema operacional deve agir de uma de duas formas: **a) fazer com que o programa aguarde** ou **b) receber os dados a serem impressos e liberá-los para a impressora assim que ela esteja livre**. No caso de impressoras, é muito comum que a segunda alternativa seja implementada, e o nome deste recurso costuma ser *spool*.

Entretanto, não existem apenas equipamentos em que um usuário possui muitos programas. Em alguns sistemas, múltiplos usuários podem, ainda, estar usando o mesmo equipamento ao mesmo tempo. Neste caso, existe ainda mais a necessidade de um gerenciamento de recursos adequado, já que cada usuário terá seu conjunto de aplicativos, todos executando ao mesmo tempo, com os mesmos dispositivos, na mesma CPU e com a mesma memória.

Estas funções de gerenciamento fazem com que muitos usuários enxerguem o sistema operacional como um *gerenciador de recursos*.

2. GERENCIAMENTO DE PROCESSOS, DISPOSITIVOS E MEMÓRIA

Como os computadores modernos todos - inclusive os mais simples - executam com múltiplos processos, dentre as funções mais importantes do Sistema Operacional estão o gerenciamento de CPU, memória e processos.

2.1 Gerenciamento de CPU / PROCESSOS

Como há muitos processos e poucas CPUs (por simplicidade consideraremos apenas uma), se a CPU executar os programas de forma simplesmente sequencial pode haver um problema sério. Imagine que você abra o seu MSN e o seu programa de tocar MP3. Execução sequencial significaria que você teria que fechar o MSN para poder ouvir a música... ou desligar a música para usar o MSN.

Para que isso não ocorra, a CPU faz um **compartilhamento de CPU**. Isso significa que, como não existe uma CPU para cada processo/programa, cada programa poderá usar a CPU por um intervalo de tempo (por exemplo, 32ms) e, depois, terá de esperar pela execução de outros programas.

Esse compartilhmento de CPU é chamado de **Gerenciamento de Processos**. Mas o que seria um processo?

Simplificadamente, um *processo* pode ser definido como um **programa em execução**. Os processos ocupam memória, utilizam entrada e saída, ocupam o processador... e, inclusive, podem possuir *subprocessos* a serem executados simultaneamente.

O sistema operacional precisa permitir que tudo isso funcione em harmonia e, para tal, tem as seguintes funções diante relativas ao gerenciamento de processos:

a) Criação e remoção de processos (de sistema ou do usuário).
b) Suspensão e reativação de processos.
c) Sincronização de processos.
d) Comunicação entre processos.
e) Tratamento de impasses entre processos.

2.2 Gerenciamento de Memória

A memória principal é um dos principais recursos de um computador, sendo o local onde processos e seus dados ficam armazenados. É comum que um computador tenha menos memória principal que o necessário para carregar todos os programas e que, se esta memória não for utilizada de uma maneira ordenada, o equipamento "trave" ou não consiga executar algumas tarefas. Por exemplo: se seu computador tem 2GB e já estão ocupados com programas em torno de 1.75GB, se você quiser abrir uma grande imagem, de 0.5GB, em um trabalho... o computador poderia simplesmente lhe informar: "Falta Memória".

Para que isso não ocorra, o sistema operacional precisa fazer o gerenciamento de memória: como não há memória para todos os programas, aqueles programas que não estão sendo usados com tanta frequência serão transferidos para uma **memória virtual**, isto é, no HD, para que os programas realmente ativos tenham mais espaço. Quando o programa que for para o disco tiver de ser executado novamente, aí ele será trazido de volta à memória principal pelo sistema operacional.

Por outro lado, os processos não devem "saber" disso: eles devem ser executados de maneira que, para cada processo, é como se o equipamento (e a memória) fosse somente sua. Isto significa que todos os processos podem pensar que estão sendo executados no mesmo local da memória (mesmo endreço). Isto certamente não será verdade, mas os processadores modernos possuem recursos para permitir que os processos sejam executados com essa ilusão. Este é um recurso chamado de ***espaço de endereçamento virtual***.

Além dos processos sendo executados pela CPU, também os dispositivos acessam a memória através de mecanismos de DMA, já vistos anteriormente. Assim, o sistema operacional também precisa fornecer algumas funções relativas ao gerenciamento de memória. Algumas delas são:

a) Manter a informação de quais partes da memória estão em uso (e por quem).
b) Decidir quais processos devem ser carregados quando a memória ficar disponível.
c) Alocar e remover processos e dados da memória quando necessário.
d) Controlar o sistema de enedereçamento virtual dos processos.

2.3 Gerenciamento de Dispositivos

Os dispositivos de um computador são, normalmente, dispositivos complexos e, em muitos casos, não funcionariam bem se mais de um processo tentasse usá-los simultaneamente. O que aconteceria, por exemplo, se um programa estivesse imprimindo e outro tentasse imprimir ao mesmo tempo?

Certamente teríamos uma impressão toda embaralhada, com partes de um documento e partes de outro. Como não é isso que ocorre, alguma coisa deve ser feita pelo computador ou pelo sistema operacional para evitar esse problema. De fato, esse é o chamado **gerenciamento de dispositivos**.

O gerenciamento de dispositivos, dentre outras coisas, controla quantos processos podem acessá-lo e em qual instante. Dependendo da situação o dispositivo pode aparecer como indisponível, solicitar que o programa aguarde por sua vez ou, simplesmente, receber os dados e processá-los da forma adequada.

Mas o gerenciamento de dispositivos inclui uma outra parte muito importante.

Na aula de dispositivos vimos que cada dispositivo costuma receber um número para o computador, indicando a **porta** em que aquele dispositivo foi ligado. Se um programa quer escrever a instrução 0x37 em um dispositivo que está na porta 0x88A0, ele usa uma instrução do tipo OUT, que teria uma sintaxe parecida com esta:

```
OUT    (0x88A0),0x37
```

Em linguagem C, existe a instrução **outp**, que, para a mesma finalidade, teria a seguinte sintaxe:

```
outp (0x88A0,0x37)
```

Ora, quem é que escolheu a porta 0x88A0 para o dispositivo? Existem várias possibilidades mas, em geral, que escolhe este número é o **fabricante**. E o que significa a instrução 0x37? Certamente só o **fabricante** sabe. Por exemplo: em uma placa de vídeo nVidia isso pode significar "**desenhe uma reta**" e em uma placa da ATI isso pode singificar "**desenhe um ponto**". Como é que o sistema operacional vai saber todas essas informações?

A resposta é...

NÃO VAI!

O sistema operacional espera que o fabricante forneça uma **biblioteca** chamada **driver de dispositivo** que sabe todas essas informações. O código do driver da nVidia que desenha a reta seria algo como:

```
void linha(int x0, int y0, int x1, int y1, int cor) {
        OUT (0x88A0, 0x37);             /* Comando */
        OUT (0x88A0, x0);               /* Dados Esperados */
        OUT (0x88A0, y0);
        OUT (0x88A0, x1);
        OUT (0x88A0, y1);
        OUT (0x88A0, cor);
}
```

Já no **driver** da ATI a mesma função poderia ser algo como:

```
void linha(int x0, int y0, int x1, int y1, int cor) {
        OUT (0x0633, x0);               /* Dados */
        OUT (0x0634, y0);
        OUT (0x0635, x1);
        OUT (0x0636, y1);
        OUT (0x0637, cor);
        OUT (0x0638, 0x42);             /* Comando */
}
```

Observe que não apenas os números, mas a forma de trabalhar dos dispositivos pode ser diferente: no primeiro caso, apenas **uma porta** foi usada, e todos os dados foram mandados para lá. Neste caso, a ORDEM é importante! No segundo caso, foram usadas **6 portas** diferentes e a única "ordem" importante é que o comando deve ser dado por último.

O sistema operacional, por sua vez, quando quer desenhar uma linha na tela, simplesmente usa o comando:

```
linha(x0, y0, x1, y1, cor);
```

E quem deve fazer o serviço é o driver.

Quem faz o driver? A fabricante do dispositivo... ou seja: **engenheiros eletrônicos** e **engenheiros de computação**.

3. GERENCIAMENTO DE PROCESSOS

* Início: um computador = um programa
  - Acesso direto aos recursos
  - programas, tarefas, *jobs*...
* Problema: um computador = diversos programas?
  - Execução em ambiente "exclusivo"
  - *processo* : importante!
* Processo?
  - Programa: está no disco
  - "Cópia de Programa em Execução" => "código + estado"
* Mais processos que CPUs = troca de processo em execução
  - Quando processo finaliza
  - Quando processo espera algo ocorrer
  - Quando acaba o tempo do processo: *time slice* (fatia de tempo)
* Troca de processos: SO deve reconfigurar a CPU
  - Os dados de reconfiguração => o "tal" estado
    + Em que ponto da "receita" estávamos quando paramos?

No início, os computadores executavam apenas um programa de cada vez, que podiam acessar diretamente todos os seus recursos. Muitos nomes foram sugeridos para indicar as "coisas que o computador executa": programas, tarefas, *jobs*...

Entretanto, a partir do momento em que os computadores (e sistemas operacionais) passaram a executar diversos programas ao mesmo tempo, surgiu uma necessidade de criar uma "separação" entre estas tarefas, e dessa necessidade surgiu o conceito de *processo*, que se tornou unânime. Neste contexto, o conceito de processo é o mais importante de um sistema operacional moderno.

Simplificadamente, "processo" é um programa em execução. Entretanto, quando se trabalha em sistemas multitarefa, essa definição fica simplista demais. Em um sistema operacional multitarefa e multiusuário, o sistema operacional simula, para cada processo sendo executado, que apenas ele está sendo executado no computador. Pode-se dizer que um "programa" é uma entidade passiva (como um arquivo no disco) e que cada cópia dele em execução é um processo, incluindo não apenas o código original (único), mas dados e informações de estado que podem ser únicas de cada processo.

Como, em geral, há mais processos sendo executados do que CPUs disponíveis no equipamento, isso significa uma troca constante e contínua do processo sendo executado por cada CPU. O tempo que cada processo fica em execução é denominado "fatia de tempo" (*time slice*). O tempo da CPU é compartilhado entre processos.

Ora, considerando que cada processo tem requisitos específicos, a cada troca de processo em execução é necessário reconfigurar uma série de propriedades do processador. Um processo, então, envolve não apenas o "programa em execução", mas também todos os

dados necessários para que o equipamento seja configurado para que este processo incie (ou continue) sua execução. Assim, de uma forma mais abrangente, "processo" pode ser definido como o "um programa e todo o ambiente em que ele está sendo executado".

3.1. O Modelo de Processo (Opcional)

* Ambiente do Processo
    - Memória
        + Ganho 64K; Precisa de mais = ?
    - Ponto de Execução
    - Arquivos de Trabalho (Abertos)
    - Prioridades...
* Program Control Block (PCB) (Figura 1)
    - Estado do Hardware (Registradores, dispositivos etc.)
    - Estado do Software (Quotas, arquivos, "estado" do processo etc.)
    - Configuração da Memória (onde e quanto acessar)

Como foi dito, "processo é o ambiente onde se executa um programa", envolvendo seu código, seus dados e sua configuração. Dependendo de como o processo for definido, um programa pode funcionar ou não; por exemplo: se um programa for executado como um processo que permite que ele use apenas 64KB de RAM e este programa exigir mais, certamente ele será abortado.

Estas informações (sobre quanta RAM está disponível em um processo, em que ponto ele está em sua execução, qual seu estado e prioridades etc.) são armazenadas em uma estrutura chamada "Bloco de Controle de Processos" (Process Control Block - PCB). A figura 1 apresenta um exemplo de estrutura de PCB:

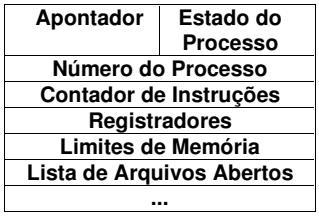


Figura 1: Exemplo de estrutura de um PCB

Os processos são gerenciados através de chamadas de sistema que criam e eliminam processos, suspendem e ativam processos, além de sincronização e outras ações.

A PCB usualmente armazena as seguintes informações: **estado do hardware**, **estado do software** e **configuração de memória**.

3.1.1. Estado do Hardware (Opcional)

As informações do estado do hardware são, fundamentalmente, relativas aos valores dos registradores dos processadores. Assim, o ponteiro da pilha (SP), do contador de programa (PC) e outros são armazenados quando um "programa sai de contexto", isto é, ele é momentaneamente interrompido para que outro possa ser executado.

Estas informações são armazenadas para que, quando o programa seja colocado novamente em execução, ele possa continuar exatamente onde estava, como se nada houvesse ocorrido e ele nunca tivesse deixado de ser executado.

Estas trocas, em que um processo deixa de ser executado, sua configuração é salva, e outro processo passa a ser executado (após devida configuração) são denominadas "trocas de contexto".

3.1.2. Contexto do Software (Opcional)

As informações de contexto do software são, essensialmente, identificações e definição de recursos utilizados.

Há sempre uma IDentificação de Processo (PID), que em geral é um número, podendo indicar também a identificação do criador deste processo, visando implementação de segurança.

Há também as limitações de quotas, como números máximos de arquivos que podem ser abertos, máximo de memória que o processo pode alocar, tamanho máximo do buffer de E/S, número máximo de processos e subprocessos que este processo pode criar etc.

Adicionalmente, algumas vezes há também informações sobre privilégios específicos, se este processo pode eliminar outros não criados por ele ou de outros usuários etc.

3.1.3. Configuração de Memória (Opcional)

Existe, finalmente, as informações sobre configuração de memória, indicando a região da memória em que o programa, onde estão seus dados etc. Mais detalhes sobre esta parte serão vistos quando for estudada a gerência de memória.

3.2. Estados de Processo

* Mais processos que CPU = fila de processos
* Quando um processo sai de execução, outro entra em execução
* Decisões dependem dos estados dos processos:
    - Novo
    - Execução (running)
    - Pronto (ready)
    - Espera/Bloqueado (wait / blocked)
        + Recurso próprio x de terceiros
    - Terminado
* Trocas
    - *Task Switch* (troca de processo)
        + Pronto => Execução
        + Execução => Pronto
    - Sincronia
        + Execução => Espera/Bloqueado
        + Espera/Bloqueado => Pronto
        - Sistemas Tempo Real: Espera/Bloqueado = > Execução!

Em geral, em um sistema multitarefa, há mais processos sendo executados do que CPUs. Desta forma, nem todos os processos estarão sendo executados ao mesmo tempo, uma boa parte deles estará esperando em uma *fila de processos*.

Em algum momento, o sistema irá paralisar o processo que está em execução (que irá para fila) para que outro processo possa ser executado ("sair" da fila). Os processos podem, então, estar em vários estados:

- Novo
- Execução (running)
- Pronto (ready)
- Espera/Bloqueado (wait / blocked)
- Terminado

O processo *novo* é aquele que acabou de ser criado na fila e ainda está em configuração. Funciona, para o resto do sistema, como um processo em *espera* - ou seja, ainda não pode ser executado.

O processo em *execução* é aquele que está ativo no momento (em sistemas com mais de uma CPU, mais de um pode estar neste estado ao mesmo tempo). Este processo é considerado "fora da fila", apesar de na prática ele ainda estar lá.

O processo *pronto* é aquele que está apenas aguardando na fila, esperando por sua "fatia de tempo" (*timeslice*) para ser executado.

O processo está em *espera* quando está na fila por aguardar algum evento externo (normalmente a liberaçao ou resposta de algum outro processo ou de algum dispositivo). Quando um processo espera pela liberação de um recurso, em geral se diz que ele está *bloqueado*. Quando o processo espera uma reposta de um recurso ao qual ele já tem acesso, diz-se que ele está simplesmente em *espera*.

O processo *terminado* é aquele que acabou de ser executado, mas está na fila porque ainda está sendo eliminado. Funciona, para o resto do sistema, como um processo em *espera* - ou seja, não pode ser executado.

Processos em espera ou prontos podem ser colocados na memória secundária (swap), mas para estar em execução ele sempre precisará ser movido para a memória principal.

3.2.1. Mudanças de Estado

O estado de um processo é alterado constantemente pelo sistema operacional; na simples troca de contexto, ou seja, qual processo está sendo executado, ocorre essa alteração de estado. Estas trocas podem ser:

Task Switch (troca de contexto)
- Pronto => Execução
- Execução => Pronto

Sincronia
- Excução => Espera/Bloqueado
- Espera/Bloqueado => Pronto

Apenas em sistemas Tempo de Real (*Real Time*) é permitido que um processo em espera ou bloqueado volte diretamente para o estado de execução.

3.3. Escalonamento de Processos (*Process Scheduling*)

* Gerenciador de Processos
    - Cria e Remove processos
    - Escalonador => troca processos
        + Usa informação do processo para isso
            = estado: pronto
            = prioridade
        + Evitar *starvation*

Uma das principais funções do Gerenciador de Processos de um sistema operacional multitarefa/multiusuário é realizar o Escalonamento de Processos, ou seja, realizar a troca de contexto dos processos. Esta tarefa pode ser executada de muitas maneiras, com auxílio ou não dos processos.

Basicamente, existe um número limitado de CPUs e um número usualmente maior de processos prontos para execução. O **escalonador** (*scheduler*) é o elemento do gerenciador de processos responsável pelo *escalonamento*, e sua função é **escolher** qual dos processos em estado de "pronto" será o próximo a receber o estado de "execução" na próxima fatia de tempo de execução (*timeslice*).

Seus objetivos são manter a CPU ocupada a maior parte do tempo (enquanto houver processos a serem executados, claro), além de balancear seu uso, ou seja, dividir o tempo entre os processos de acordo com sua *prioridade*, além de garantir um tempo de resposta razoável para o usuário e evitar que algum processo fique tempo demais sem ser executado (*starvation*).

Como um guia, o escalonador utiliza informações sobre os processos (se ele usa muito I/O, se ele usa muita CPU, se ele é interativo, se não é interativo etc.) para tentar acomodar da melhor maneira possível as necessidades.

4. GERENCIAMENTO DE MEMÓRIA

* Memória RAM => Recurso limitado
  - Tamanho e Velocidade
  - Seu gerenciamento é **muito** importante
* Início: apenas um processo: gerenciamento simples!
* Multitarefa/Multiusuário: cada byte conta!

Memória principal (RAM) sempre foi um dos recursos mais limitados em um sistema computacional, seja em termos de velocidade ou em termos de quantidade. Por esta razão, um eficiente sistema de gerenciamento de memória sempre foi uma das grandes preocupações no projeto de um sistema operacional.

O gerenciamento de memória principal era, inicialmente, simples. Os computadores executavam apenas um processo de cada vez e não eram necessários processos mais complicados para gerenciarmento de memória. Entretanto, à medida em que os sistemas multitarefa e multi-usuários surgiram, a gerência de memória se tornou mais complexa, com o objetivo de aproveitar melhor cada byte disponível.

Assim, serão apresentados os fundamentos da gerência de memória, desde os processos mais simples aos mais atuais, além de apresentar o conceito de endereçamento virtual e memória virtual.

4.1. Alocação Contígua Simples

* Sistemas monoprogramados
  - Existe isso?
* Dois processos: Programa + SO
  - Figura 3
* Processo: Respeitar o limite de memória
* Primeiras CPUs: nenhum controle de acesso
* CPUs posteriores: MMU: Memory Management Unit
  - Registrador que indica fim da área de sistema (Figura 4)
    + Programa do usuário não tem acesso lá!
      = *Access Violation* ou GPF
* Problema: ineficiente e inviável para multitarefa

Em sistemas monoprogramados, ou seja, aqueles em que há apenas um processo executando (e é necessário que ele termine para que outro seja carregado), não há a necessidade de um gerenciamento complexo.

Neste tipo de sistema, em geral, o sistema operacional é carregado em uma dada região da RAM (normalmente a parte mais baixa) e o programa sendo executado (o processo) é carregado em seguida ao sistema operacional, sempre na mesma área (Figura 2). O processo so precisa se preocupar em não ultrapassar o limite de memória disponível.

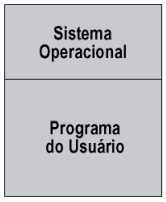


Figura 2: Alocação Contígua Simples

Nos primeiros equipamentos não existia qualquer proteção de memória para a área do sistema operacional. Isto significa que, se o processo executado sobrescrevesse a área do sistema operacional, poderia danificá-lo.

Em equipamentos posteriores, os processadores foram dotados de uma unidade simples, chamada MMU (*Memory Management Unit* - Unidade de Gerenciamento de Memória), que tinha como responsabilidade limitar o acesso do software à região do sistema oepracional, criando uma primeira versão simples de "modo supervisor" e "modo usuário".

Nestes primeiros sistemas o único controle existente na MMU era um registrador que indicava o endereço de memória onde acabava a área de sistema, ponto em que se iniciáva a área do software do usuário (Figura 3).

Figura 3: Separação entre Área de Sistema e Área de Usuário nas primeiras MMU

Em essência, qualquer código que fosse executado em uma área abaixo da área de sistema (endereço de memória menor ou igual ao valor contido no registrador da MMU) teria acesso à toda a memória; por outro lado, qualquer programa que fosse executado em um endereço **acima** do valor contido no registrador da MMU só poderia acessar a área acima da MMU; isto é, se o programa na área do usuário tentar escrever na área do sistema, um erro do sistema seria gerado, em geral causando o fechamento do aplicativo com uma mensagem do tipo *Access Violation* (Violação de Acesso) ou *General Protection Fault* (Falha de Proteção Geral).

Apesar de bastante simples e eficaz, este método de gerenciamento é ineficiente, muitas vezes sobrando uma grande parcela de memória inutilizada (nem pelo sistema, nem pelo software do usuário). Além disso, este esquema de gerenciamento é inviável em sistemas multitarefa ou multi-usuários.

4.2. Alocação com Particionamento Dinâmico

* Sistemas multiprogramados
  - Sistemas autais
* MMU com dois registradores: **início** e **fim**
* Área tem exatamente o tamanho necessário para o processo
  - Ex.: Figura 5
  - Ou programa "cabe" ou "não cabe" na RAM
* Novo problema: fragmentação (Figuras 6a e 6b)
  - Há 6KB disponível
  - Falta memória para processo de 6KB
  - Memória **contígua**
* Solução: desfragmentar memória (Figura 6)
  - Mas memória continua absoluta
  - Relocação em tempo de execução: lento!
  - Estratégia para reduzir a necessidade de desfragmentação

Como objetivo de permitir que vários programas pudessem ser executados simultaneamente, foi criado o esquema de alocação particionada. Neste esquema a memória é dividida em regiões, cada região estando disponível para um processo.

Neste modelo, há um sistema de **dois registradores** de MMU, um que marca o **início** e outro que marca o **fim** da área do processo, sendo seus valores determinados no momento de carregamento de um processo, exetamente do tamanho necessário para aquele programa (Figura 4). É claro que se um processo tentar acessar qualquer região fora de sua área permitida, um erro de violação de acesso será gerado.

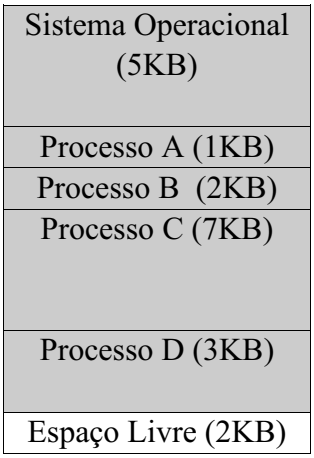


Figura 4: Particionamento Dinâmico da RAM

Este modelo, entretanto tem um problema chamado de **fragmentação**, que surge quando processos são inciados e finalizados. Suponha que, ao carregar os programas do usuário, a memória ficou completamente cheia de pequenos processos de 1KB (Figura 5a) e, posteriormente, alguns foram fechados, liberando pequenos trechos de 1KB espalhados pela RAM, espaços estes que, somados, totalizam 6KB (Figura 5b).

Sistema Operacional
Processo A (1KB)
Processo B (1KB)
Processo C (1KB)
Processo D (1KB)
Processo E (1KB)
Processo F (1KB)
Processo G (1KB)
Processo H (1KB)
Processo I (1KB)
Processo J (1KB)

Sistema Operacional
Processo A (1KB)
Livre (2KB)
Processo D (1KB)
Livre (2KB)
Processo G (1KB)
Livre (2KB)
Processo J (1KB)

Figuras 5a e 5b: Fragmentação da RAM no particionamento dinâmico.

Neste caso, se o sistema precisar carregar um processo de 6KB, não será possível, já que os processos precisam de memória **contígua**, já que a proteção de memória é feita apenas com 2 registradores, um indicando o início e outro o fim da região protegida.

A primeira solução que foi dada para este problema era "por software", isto é, foi criado um sistema de gerenciamento de particionamento dinâmico com relocação, em que os processos são relocados quando necessário (Figura 6), criando espaço para o novo processo.

Entretanto, este processo de relocação é bastante custoso computacionalmente (mesmo nos tempos atuais). Para evitar isso, os sistemas usam estratégias para minimizar a fragmentação da memória, **além de contar com mecanismos em hardware** para tanto.

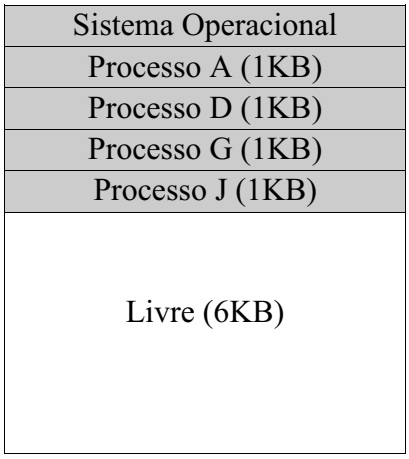


Figura 6: Desfragmentação da RAM no particionamento dinâmico com relocação.

4.3. Arquivo de Troca (*Swapping*)

* Processos precisam sempre estar na memória?
  - Quando estão executando a todo instante?
  - Quando estão esperando algo que pode demorar?
* Memória não é ilimitada!
  - Processo em longa espera: tirar da memória, temporariamente
  - Swapping: usar disco como memória-depósito-temporária
  - Volta à memória: quando processo for executar novamente
    + Será no mesmo lugar?
    + Relocação em tempo de execução: lento!

O sistema de alocação dinâmica apresentado anteriormente é muito interessante para aproveitar bem a memória. Entretanto, se ele for usado da forma descrita acima (o que, de fato, ocorre em muitos sistemas) ele tem uma severa limitação: todos os processos precisam permanecer na memória principal desde seu carregamento até o momento em que é finalizado.

Isso significa que os processos ocuparão a memória principal ainda que não estejam sendo processados como, por exemplo, quando estão esperando a resposta de algum dispositivo. Eventualmente, um processo pode esperar por horas para receber um dado sinal e, durante todo este tempo sem executar, estará ocupando a memória principal.

Isto não é exatamente um problema caso um sistema disponha de memória princial praticamente ilimitada, onde nunca ocorrerá um problema de falta de memória; entretanto, na prática, em geral a memória principal é bastante limitada e pode ocorrer de um novo processo

precisar ser criado mas não existir espaço na memória. Neste caso, se houver processos em modo de espera, seria interessante poder "retirá-los" da memória temporariamente, para liberar espaço para que o novo processo possa ser executado.

Esta é exatamente a função do mecanismo de "swapping": quando um novo processo precisa ser carregado e não há espaço na memória principal, o processo menos ativo (em geral, bloqueado ou em espera) será transferido para um arquivo em disco, denominado **arquivo de troca** ou **arquivo de swap**, no qual ele permanecerá até o instante em que precisar ser executado novamente. Com isso, uma área na memória principal é liberada para que o novo processo seja executado.

Entretanto, o uso da técnica de *swapping* cria um problema: quando um processo é colocado no swap, ele não pode ser executado. Assim, ele precisará **voltar à memória** principal para poder ser executado, quando seu processamento for liberado. Isso não traz nenhum problema a não ser o fato de que dificilmente ele poderá voltar a ser executado na mesma região da memória principal em que estava inicialmente. Isso implica que ele deve ser **realocado dinamicamente**, da mesma forma com que contece nas desfragmentação da memória, já citada anteriormente. E, também neste caso, este é um processo complexo e lento se tiver de ser executado por software (além de depender que o programa tenha sido preparado para isso).

4.4. Endereçamento Virtual

* Como evitar ter de mexer em todos os endereços de um programa?
  - Fazer com que ele pense que está sempre executando no mesmo lugar!
* MMU: **registro de endereço base**
  - Programa pensa sempre que está rodando a partir do endereço zero
  - Processador calcula endereço real:
    endereço indicado pelo programa **+** endereço base
  - Exemplos: tabela

Como vários processos de gerenciamento de memória esbarram no problema de relocação dinâmica, com necessidades de ajustes lentos em todos os programas em execução, é natural que os desenvolvedores de hardware tenham se dedicado a criar mecanismos que eliminassem esses problemas.

Com o objetivo específico de facilitar este processo de realocação dinâmica para o swapping e, em parte, na desfragmentação de memória, as MMUs dos processadores passaram a incorporar, além dos registradores de início e fim da área de operação do processo, o **registrador de endereço base**. Este registrador indica qual é a posição da memória inicial de um processo e seu valor é somado a todos os endereços referenciados dentro do processo.

Assim, não só os processos passam a enxergar apenas a sua própria região da memória, como todos eles acham que a sua região começa sempre no endereço ZERO (0). Como consequência, os endereços do software não precisam ser mais modificados, pois eles são sempre **relativos** ao "zero" da memória que foi destinada a ele.

Em outras palavras, se um programa é carregado no endereço 1000 da memória, o registrador de endereço base é carregado com o valor 1000. A partir de então, quando o processo tentar acessar o endereço 200, na verdade ele acessará o endereço 1200 da memória, já que o valor do endereço base será somado ao endereço acessado. Observe na tabela abaixo.

Registrador de Endereço Base		Endereço Referenciado		Endereço Físico
1000	+	200	=	1200
2000	+	200	=	2200
1000000	+	200	=	1000200
1000	+	123	=	1123
2000	+	1200	=	3200
1000000	+	123456	=	1123456

Assim para "relocar dinamicamente" um processo, basta alterar o valor do Registrador de Endereço Base no momento da execução daquele processo, apontando a posição inicial da área da memória principal em que aquele processo será colocado para execução, eliminando a necessidade de modificações no código dos programas que estão em execução.

4.4.1. Espaço Virtual de Endereçamento

* Memória acessível pelo aplicativo dividida em blocos: **páginas**
* MMU: registro de endereçamento base para cada página
    - Mapeamento direto: todos os registros em zero (Tabela)
    - Sobreposição de áreas (Tabela)
    - Inversão de áreas (Tabela)
    - Descotinuidade completa (Tabela)
* Estes dados ficam armazenados para CADA processo
    - Informação de "estado", como visto na unidade anterior
* Diminui x Elimina desfragmentação

O sistema de endereçamento virtual é, na verdade, uma combinação de vários dos recursos anteriormente citados e está presente na maioria dos sistemas operacionais modernos, devendo, para isso, ser suportado pelo hardware.

A idéia é simples e é baseada na idéia do registrador de endereço base, mas bastante mais flexível. Por exemplo, imagine que, ao invés de um registrador de endereço base, existam vários deles. Neste exemplo, é possível existir um registrador para cada bloco de memória. Ou seja: se há 1024 posições de memória, pode-se dividir este conjunto em 8

blocos de 128 bytes de memória, chamdas *páginas*. Cada página terá seu próprio registrador de endereço base:

Posição de Memória Referenciada
0 ~ 127
128 ~ 255
256 ~ 383
384 ~ 511
512 ~ 639
640 ~ 767
768 ~ 895
896 ~ 1023

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0

Com a configuração apresentada acima, existe um "mapeamento direto", ou seja, os valores dos registradores de endereço base são tais que a posição referenciada corresponde exatamente à posição de mesmo endereço na memória física. Entretanto, isso não é obrigatório: suponha que, por alguma razão, se deseje que quando forem acessados os dados de 128 ~ 255, na verdade sejam acessados os dados de 256 ~ 383. Como fazer isso? Simples: basta alterar o registrador de endereçamento número 1 (da área 128 ~ 255) com o valor 128. Assim, quando o endereço 128 for referenciado, o endereço físico acessado será 128 + 128 = 256. Quando o endereço 255 for referenciado, o endereço físico acessado será 255 + 128 = 383.

Posição de Memória Referenciada
0 ~ 127
128 ~ 255
256 ~ 383
384 ~ 511
512 ~ 639
640 ~ 767
768 ~ 895
896 ~ 1023

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento	Posição de Memória Física Acessada
0	0	0 ~ 127
1	128	256 ~ 383
2	0	256 ~ 383
3	0	384 ~ 511
4	0	512 ~ 639
5	0	640 ~ 767
6	0	768 ~ 895
7	0	896 ~ 1023

Pela mesma razão, pode ser interessante que os acessos à região de endereços de 256 ~ 383 fossem mapeados para os endereços físicos 128 ~ 255. Isso pode ser feito indicando no registrador de endereços 2 o valor -128:

Posição de Memória Referenciada
0 ~ 127
128 ~ 255
256 ~ 383
384 ~ 511
512 ~ 639
640 ~ 767
768 ~ 895
896 ~ 1023

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento	Posição de Memória Física Acessada
0	0	0 ~ 127
1	128	256 ~ 383
2	-128	128 ~ 255
3	0	384 ~ 511
4	0	512 ~ 639
5	0	640 ~ 767
6	0	768 ~ 895
7	0	896 ~ 1023

Observe que isso praticamente resolve de forma completa o problema da desfragmentação da memória, permitindo inclusive que um processo seja executado em porções não contíguas de memória: pelo ponto de vista do processo, ele **estará em uma região contígua**. Por exemplo: um programa de 256 bytes poderia ser alocado assim:

Posição de Memória Referenciada
0 ~ 127
128 ~ 255

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento	Posição de Memória Física Acessada
0	256	256 ~ 383
1	512	640 ~ 767

Para o processo, ele está em uma região contígua da memória principal, dos endereços 0 ao 255. Entretanto, na prática, ele está parte alocado na região 256 ~ 383 e 640 ~ 767. Este mecanismo se chama *espaço de endereçamento virtual com mapeamento de páginas*. Entretanto, cada processo precisa possuir uma tabela destas. E, de fato, esta tabela faz parte das informações do processo, dados estes armazenados na região de configuração de memória.

A memória virtual permite que a relocação de processos na memória seja simplificada (basta trocar os valores dos registradores de endereço) e diminui muito a necessidade de desfragmentação da memória. Mas ... porque diminui a necessidade ao invés de "elimina"?

A resposta é simples: por questões de economia de espaço, em geral se define que cada processo pode estar fragmentado em até X partes, sendo X um número fixo e, normalmente, não muito alto. Quanto maior for o número de fragmentos permitidos, maior é a tabela que deve ser armazenada por processo e maior é o desperdício de memória com estas informações. Se ocorrer uma situação em que um processo só caberia na memória se estivesse fragmentado em um número de fragmentos maior que X, então será necessário que o sistema desfragmente a memória. De qualquer forma, a desfragmentação fica bem mais rápida, já que a relocação de processos pode ser feita de forma simples, sem mudanças nos programas.

4.5. Memória Virtual (Opcional)

* Mapear em páginas regiões além da RAM física
  - Essas regiões ficam no disco: swapping
* Quando aplicativo tenta acessar uma dessas páginas
  - Sistema detecta
  - Traz página para a memória física
  - Ajusta registradores de endereço de página
  - Permite o acesso
* Processo transparente para o software
  - Perceptível para o usuário
    + Lentidão
    + *Thrashing*

Com base na idéia do sistema de endereçamento virtual apresentado anteriormente, muitos processadores e sistemas operacionais implementaram suporte para o que se convencionou chamar de "memória virtual", que nada mais é do que a incorporação da tecnologia de *swapping* em conjunto com o endereçamento virtual. O resultado é como se a memória total disponível para os aplicativos fosse a memória RAM principal física **mais** o espaço disponível na memória secundária. Como isso é feito?

Considere que um sistema tem 1GB de memória principal física (RAM) e vários GBs de disco. A idéia é que um registrador de endereçamento base **possa apontar posições de memória maiores do que a memória física**. Para o processo, ele é acarregado como se o sistema de fato tivesse mais memória disponível do que a física. Toda a parte do processo que ultrapassar a memória física disponível, será automaticamente jogada para o arquivo de troca (swap). Em outras palavras, parte do processo será colocada no disco.

Entretanto, quando o processo tentar acessar uma destas regiões que estão além da memória física, isso disparará uma ação do sistema operacional que irá recuperar do arquivo de swap o bloco que contém a informação desejada, colocando-a de volta na memória principal e corrigindo o registrador de endereçamento virtual para refletir a nova realidade e, assim, permitindo que o programa acesse aquela informação. Note que tudo isso ocorre de maneira absolutamente transparente para o processo.

Esta técnica permite o swapping de forma 100% transparente ao processo, facilitando muito o funcionamento do sistema operacional e mesmo a programação dos processos. O sistema acaba se comportando, realmente, como se a quantidade de memória disponível fosse bem maior. Entretanto, existe uma penalidade: como a memória secundária é, em geral, bastante mais lenta que a memória principal, o desempenho do equipamento como um todo é bastante prejudicado quando o uso de memória virtual (swap) é necessário.

Quando a atividade de swap cresce muito e o sistema se torna muito lento por causa disso, dá-se o nome de *thrashing* do sistema.

5. BIBLIOGRAFIA

DAVIS, W.S. **Sistemas Operacionais**: uma visão sistêmica. São Paulo: Ed. Campus, 1991.

MACHADO, F. B; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª. Ed. São Paulo: LTC, 2007.

SILBERSCHATZ, A; GALVIN, P. B. **Sistemas operacionais**: conceitos. São Paulo: Prentice Hall, 2000.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª.Ed. São Paulo: Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

Unidade 10: A Unidade Lógica Aritmética
e as Instruções em Linguagem de Máquina

Prof. Daniel Caetano

Objetivo: Apresentar as funções o mecanismo de atuação da Unidade Lógica Aritmética e iniciar a descrição dos tipos de instrução que o processador executa.

Bibliografia:

- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

* CPU: Duas partes principais
  - ULA + UC
* Iniciar pela ULA: execução dos cálculos

Nesta aula iniciaremos o estudo do funcionamento e operação de uma Unidade Central de Processamento (CPU), através do estudo de cada um de seus elementos. O primeiro elemento que será estudado é a Unidade Lógica Aritmética; não porque ele é o elemento mais simples ou porque ele é o primeiro elemento na longa seqüência de operações que uma CPU executa, mas porque ele é o elemento principal de uma CPU: é a Unidade Lógica Aritmética (ULA) quem executa a maior parte das instruções de uma CPU.

Como veremos posteriormente, praticamente todas as outras partes da CPU são voltadas a uma única finalidade: trazer instruções para que a ULA as processe, bem como armazenar os dados resultantes.

1. O PROCESSADOR E OS REGISTRADORES

É possível pensar no processador como um conjunto de funcionários praticamente sem memória alguma. É como se dois funcionários, chamados ULA e Controle, estivessem traduzindo um texto em um determinado andar da empresa, mas o dicionário, chamado Memória, ficasse apenas em outro andar da empresa. Isso significa que, para cada palavra que ULA fosse traduzir, o Controle teria que ir até o outro andar, procurar a palavra na Memória e só então voltar para traduzir.

Isso tem dois problemas: 1) é extremamente lento; 2) certamente o Controle já teria esquecido a informação que foi buscar quando chegasse de volta à sua mesa de trabalho.

A parte da lentidão foi resolvida com a contratação de um funcionário subalterno, chamado Cache. O Cache fica no elevador e, quando ULA precisa de um dado, ele pede ao Controle, que grita para o Cache, que vai buscar o dado, volta e grita de volta para o Controle a resposta. Isso ainda é um pouco lento, porque o Cache às vezes ainda tem que subir e descer o elevador, mas de alguma forma às vezes ele não precisa, porque ele adivinha o que o Controle vai querer saber e se informa antes! Em algumas situações, a resposta dele é imediata!

Bem, ocorre que mesmo com a contratação deste novo funcionário (o Cache), o Controle demora um pouco a responder às dúvidas do ULA que, neste meio tempo, se esquece do que estava traduzindo e, quando ele se lembra, tanto ele quanto o Controle já esqueceram a informação que o Cache havia acabado de passar.

Tomando conhecimento deste problema, a diretoria instituiu que os funcionários Controle e ULA passassem a adotar "pequenos papéis", verdadeiras "colas", para anotar uma informação assim que o Cache a informa; desta maneira, não havia mais erro: quando ULA se lembra o que ia fazer com a informação, ele olha a "cola" e a informação está lá, para que ele possa trabalhar com ela.

Como a empresa tem certificação de qualidade total ISO 9001, a diretoria achou por bem que estas colas fossem em uma quantidade pequena e que fossem reaproveitáveis, para contribuir com a natureza e não desperdiçar recursos. Para facilitar a identificação das mesmas (e para que ninguém de outro departamento levasse as "colas" embora), a diretoria indicou dois dizeres nos mesmos: a palavra "Registrador", pois é um local onde informações devem ser registradas e o nome deste registrador, que normalmente é alguma letra: A, B, C...

Os funcionários Controle e ULA decidiram que, como não há muitos *registradores*, para evitar confusão, algumas tarefas específicas usariam sempre os mesmos *registradores*. Definiram ainda que o Controle pode escrever e ler em qualquer *registrador*, mas a ULA pode escrever apenas em alguns (em especial no A), embora possa ler praticamente todos. Tanto o Controle quando a ULA perceberam ainda que havia informações que não caberiam em apenas um *registrador*. Por esta razão, eles combinaram que, nestes casos, seriam usados pares de *registradores* para registrar a informação completa.

A idéia funcionou tão bem que todas as empresas concorrentes copiaram, embora muitas vezes deem nomes diferentes para os *registradores*.

1.1. O Funcionamento Real

A analogia ajuda a entender como o computador funciona, mas ela não é exatamente precisa. Na prática, a Unidade de Controle é quem comanda as operações e a ULA apenas responde às requisições do Controle. Ocorre que a ULA não tem ligações com a memória

física do computador (o acesso a elas é bastante complexo); mas então, como ela recebe dados?

Bem, para que a ULA possa trabalhar, um processador tem uma pequena quantidade de memória (muito rápida) dentro de seu encapsulamento. Cada posição de memória do processador é chamada de *registrador* e é ligada diretamente à ULA e ao Controle.

Assim, quando o Controle pretende enviar uma ordem à ULA (ordem esta chamada *instrução*), ele precisa preencher os *registradores* previamente, com as informações que a ULA irá precisar para executar a operação. Observe a Figura 1 para compreender melhor o processo:

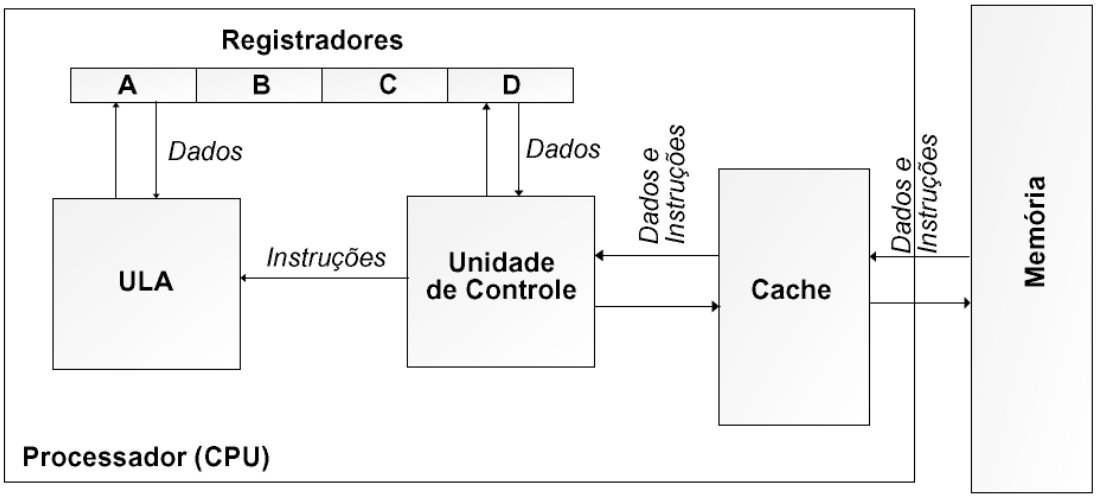


Figura 1: Interligações da ULA com Unidade de Controle e Registradores

Desta maneira, quando a Unidade de Conotrole envia uma instrução do tipo "ADD A,B" (adicionar B em A) para a ULA, esta última irá ler os registradores A e B, somá-los e colocar o resultado em A.

2. OPERAÇÕES EXECUTADAS PELA ULA

Como foi apresentado na seção anterior, a ULA responde à requisições da Unidade de Controle que, como veremos nas aulas seguintes, é responsável por preparar os registradores e entregar instruções decodificadas para a ULA.

A ULA só realizará, então, operações simples de aritmética e lógica, estando os parâmetros das mesmas em um ou mais registradores (mas nunca diretamente na memória). São operações como *adição* (ADD), *subtração* (SUB), *multiplicação* (MUL), *divisão* (DIV), *e* (AND), *ou* (OR), *ou exclusivo* (XOR), *não* (NOT) dentre outras mais específicas.

Os registradores mais importantes acessados pela ULA são o *Acumulador*, freqüentemente chamado de "A" (ou AX ou EAX na arquitetura x86). Outro registrador

importante, existente em algumas arquiteturas, é o *Flags*, normalmente chamado de "F" (ou FLAGS ou EFLAGS ou RFLAGS na arquitetura x86). Nas arquiteturas RISC, entretanto, os nomes de registradores são pouco significativos, normalmente variando de R1 a Rxx (a arquitetura RISC será melhor detalhada no futuro).

Note que instruções de leitura e escrita em memória (LOAD, MOVE, STORE etc) não são processadas pela ULA. Algumas instruções como saltos relativos à posição atual usam a ULA para o cálculo da nova posição de memória, mas grande parte do trabalho destas instruções (assim como no caso de instruções que fazem operações com dados na memória principal) é feito pela Unidade de Controle.

3. INSTRUÇÕES DA CPU

Na seção anterior falamos brevemente sobre as instruções da CPU. O que são essas instruções e quantas existem?

As "instruções" como comentadas anteriormente (ADD, LOAD, MOVE...) são, na verdade, "apelidos mnemônicos" dados a instruções binárias, para facilitar a sua memorização. Quando um computador lê uma instrução na memória, ele lê, na verdade, um conjunto de bits. Para entender melhor, vamos tomar um exemplo.

A instrução **ADD A,r** , do microprocessador Z80, serve para somar o valor de qualquer registrador **r** ao registrador A, isto é, realiza a seguinte tarefa: **A = A + r** . Pois bem, essa instrução, em bits, é descrita da seguinte forma:

1	0	0	0	0			
⟵ ADD A,r ⟶					⟵ r ⟶		

Dependendo dos valores dos bits "r", a operação executada usará um registrador diferente. Os valores que esses bits podem assumir são:

Registrador	**A**	**B**	**C**	**D**	**E**	**H**	**L**
Bits	111	000	001	010	011	100	101

Assim, a instrução ADD A,D - ou seja, A = A+D - pde ser representada assim:

1	0	0	0	0	0	1	0
⟵ ADD A,r ⟶					⟵ r ⟶		

Note que cada instrução tem uma "sintaxe de bits" diferente: em cada uma delas, os bits relativos à instrução e à designação dos operandos é diferente. Vejamos, como exemplo, uma outra instrução: **INC r** .

A instrução INC r **incrementa** o valor do registrador **r**, isto é, realiza uma operação que pode ser representada como **r = r + 1** . Em binário, essa instrução é especificada da seguinte forma:

0	0				1	0	0
← INC r →		← r →			← INC r →		

O código para indicar o registrador é o mesmo usado na instrução ADD A,r.

Ainda que esse tipo de especificação permita uma certa diversidade de operações, este número é um tanto limitado. Considerando que usaríamos sempre 3 bits para indicar um registrador, com 8 bits teríamos apenas mais 5 para indicar a instrução, o que nos limitaria a, basicamente, 32 instruções. Sendo assim, algumas instruções possuem mais de um byte (instruções de 16 bits, 32 bits...).

3.1. Instruções com Dados

Algumas instruções, como as apresentadas anteriormente, possuem toda a informação necessária em si mesma. Mas esse não é sempre o caso. Imagine, por exemplo, a instrução

ADD A,n

Essa instrução adiciona um número **n** qualquer, de 8 bits, ao valor existente no registrador A. É o equivalente a fazer **A = A + n** . Nesse caso específico, é usada **uma instrução de 8 bits que exige um dado de 8 bits em sequência**. Ou seja: a instrução ADD A,n exige que, na posição de memória imediatamente seguinte, exista o número a ser adicionado ao registrador A. Na memória isso fica assim:

1	1	0	0	0	1	1	0
← ADD A,n →							

← n →							

Assim, para realizar uma operação **ADD A,0x24** , indicamos da seguinte forma:

1	1	0	0	0	1	1	0
← ADD A,n →							

0	0	1	0	0	1	0	0
← n →							

NOTA: Como vimos, a ULA só tem acesso aos registradores. Como é possível somar um número da memória diretamente a um registrador? **Como** isso ocorre será explicado na próxima aula.

3.2. Conjuntos de Instruções

Cada CPU tem seu próprio conjunto de instruções (tanto em binário quanto a forma "mnemônica" oficial) e é um tanto inútil ficar estudando diversas delas antes de ser necessário. Entretanto, os tipos de instruções comuns na grande maioria dos processadores pode ser dividido em algumas categorias:

a) Transferência de Dados (MOVE, STORE, LOAD, EXCHANGE, PUSH, POP...)
b) Operações de Entrada/Saída (READ, WRITE...)
c) Operações Aritméticas (ADD, SUB, MULT, DIV, INC, DEC...)
d) Operações Lógicas (AND, OR, NOT, XOR, TEST, COMPARE, SHIFT...)
e) Transferência de Controle (JUMP, CALL, HALT...)
f) Operações de Conversão (TRANSLATE, CONVERT...)

As operações (a) e (b) são executadas basicamente pela Unidade de Controle (UC). As operações (c) a (f) podem envolver apenas a ULA (se forem operações apenas com registradores) ou podem envolver a ULA e a UC (se os dados estiverem na memória).

4. MODOS DE ENDEREÇAMENTO

Até agora, vimos instruções que trabalham com dois tipos de endereçamento:

a) Endereçamento a Registradores: aquelas em que o valor já está em um registrador (nenhum acesso à memória é necessário).
b) Endereçamento Imediato: aquelas que o valor segue a instrução, na memória.

Como foi comentado, como a ULA só tem acesso aos registradores (endereçamento do tipo (a)), o endereçamento imediato só é possível com o auxílio da Unidade de Controle.

> NOTA: A Unidade de Controle (UC) será o foco da próxima aula!

Entretanto, usar imediatos para acessar informações é extremamente limitado. Sendo assim, a CPU usualmente fornece uma grande quantidade de maneiras de ler dados da memória.

Uma das instruções mais simples que usa todas as possíveis maneiras de acesso à memória - e por isso vamos usá-la como exemplo - é a que lê um dado da memória para um registrador.

Essa instrução é, nromalmente, chamada de LOAD. Na arquitetura do Z80, que estamos usando como exemplo, ela é especificada como LD (de Load). Vejamos as formas possíveis de acessar a memória nessa instrução.

c) Endereçamento Direto: o número da posição e memória a ser lida é representado como um dado imediato. Por exemplo:

LD A,(e)

Essa instrução procura pelo endereço **e** na memória e lê o seu conteúdo no registrador A. No Z80, essa é uma instrução que, incluindo o endereço, tem 24 bits: 8 bits para a instrução e 16 bits para o endereço a ser lido (já que os endereços do Z80 possuem 16 bits). Exemplo: **LD A,(0x7200**)

Essa instrução **não** irá carregar o valor 0x7200 no registrador A, mas sim carregar o registrador A com o valor armazenado na posição de memória 0x7200.

d) Endereçamento Indireto: o número da posição e memória a ser lida está armazenado em um registrador. Por exemplo:

LD A,(rr)

Essa instrução procura na memória pelo endereço indicado pelo registrador **rr** e, quando o encontra, armazena o valor da memória no registrador A. No Z80, essa é uma instrução que tem apenas 8 bits, já que o registrador que armazena a informação do endereço já está incluído na instrução. Exemplo: **LD A,(HL**).

Se HL armazenar o valor 0x2010, essa instrução **não** irá carregar o valor 0x2010 no registrador A, mas sim carregar o registrador A com o valor armazenado na posição de memória 0x2010.

NOTA: Algumas arquiteturas permitem acesso indireto com um valor imediato, isto é, seria o mesmo que realizar LD A,((e)), em que o endereço de memória e indicaria o endereço do qual se quer ler o dado. Esse tipo de endereçamento é relativamente incomum.

e) Endereçamento por Deslocamento: o número da posição e memória a ser lida está armazenado em um registrador. Por exemplo:

LD A, (r + n)

Essa instrução calcula o endereço de memória a ser lido, somando o valor do registrador **r** a valor **n**. O endereço resultante é usado para ler o valor da memória para o registrador A. Exemplo: **LD A,(IX + 1**).

Nesse caso, se IX armazenar o valor 0x1010, essa instrução **não** irá carregar o valor 0x1011 no registrador A, mas sim carregar o registrador A com o valor armazenado na posição de memória 0x1011.

NOTA: dependendo do "tamanho em bits" do registrador e do número, esse tipo de endereçamento ganha um nome diferente: se o **registrador contém um endereço** e o número representa apenas um deslocamento, esse endereçamento se chama **Endereçamento via Registrador Base**. Por outro lado, se o **número é um endereço** e o registrador contém o deslocamento, esse acesso é chamado de **Endereçamento por Indexação**.

f) Endereçamento por Pilha: o número da posição e memória a ser lida ou escrita está em um registrador especial, que é operado pela unidad de controle. Neste caso, a instrução LOAD não funciona, teremos de usar o exemplo das instruções PUSH e POP.

PUSH r

Essa instrução insere o valor armazenado no registrador **r** na pilha, que é uma fila do tipo LIFO (Last In Firt Out ou, em português, o último a entrar é o primeiro a sair). PUSH serve para armazenar valores, POP serve para recuperá-los.

5. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 11: A Unidade de Controle

Prof. Daniel Caetano

Objetivo: Apresentar as funções o mecanismo de atuação da Unidade de Controle.

Bibliografia:

- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

Na aula anterior foi apresentada a Unidade Lógica Aritmética (ULA). Como vimos, a ULA é responsável por executar o processamento "de fato" de um computador, mas a ULA realiza apenas operações individuais. Para que a ULA processe seqüências de instruções, é necessário que algum dispositivo forneça tais instruções, na ordem correta.

Assim, nesta aula será continuado o estudo da Unidade Central de Processamento (CPU), apresentando a Unidade de Controle (UC) e alguns outros registradores importantes.

1. A UNIDADE DE CONTROLE

Uma das melhores analogias existentes entre a ULA e a UC é a analogia da calculadora. Enquanto a ULA é como uma calculadora simples, que executa um pequeno número de operações, a UC é como o operador da calculadora, que sabe onde buscar informações para alimentar a calculadora e também em que ordem estas infomações devem ser repassadas.

Em outras palavras, enquanto a ULA faz "partes" de um trabalho, a UC gerencia a execução destas partes, de forma que um trabalho mais complexo seja executado.

1.1. Algumas Responsabilidades da Unidade de Controle

- Controlar a execução de instruções, na ordem correta: uma vez que a ULA só cuida de executar instruções individuais, a UC tem o papel de ir buscar a próxima instrução e trazê-la para a ULA, no momento correto.

- Leitura da memória principal: Na aula anterior foi visto que a ULA não pode acessar diretamente a memória principal da máquina. A ULA só faz operações sobre os registradores, sendo que as instruções devem ser comandadas diretamente a ela. Assim, a UC

tem o papel não só de buscar as instruções na memória, como também verificar se a instrução exige dados que estejam na memória. Se for o caso, a UC deve recuperar os dados na memória e colocá-los em registradores especiais e, finalmente, solicitar que ULA execute a operação sobre estes valores.

- Escrita na memória principal: Da mesma forma que a leitura, a ULA não pode escrever na memória principal da máquina. Assim, quando for necessário armazenar o resultado de uma operação na memória principal, é tarefa da UC transferir a informação de um registrador para a memória.

- Controlar os ciclos de interrupção: praticamente toda CPU atual aceita sinais de interrupção. Sinais de interrupção são sinais que indicam para a UC que ela deve parar, momentaneamente, o que está fazendo e ir executar uma outra tarefa. As razões para as interrupções são as mais diversas, como o disparo de um timer ou uma placa de rede / model solicitando um descarregamento de seu buffer.

1.2. Rotina de Operação da CPU

Em geral, é possível dizer que uma CPU tem uma seqüência de ações a executar; algumas delas são atividades da ULA, outras da UC. Esta seqüência está apresentada a seguir:

a) Busca de instrução: quando a CPU lê uma instrução na memória.
b) Interpretação de Instrução: quando a CPU decodifica a instrução para saber quais os passos seguintes necessários.
c) Busca de dados: caso seja determinado na interpretação que dados da memória ou periféricos são necessários, a CPU busca estes dados e os coloca em registradores.
d) Processamento de dados: quando a instrução requer uma operação lógica ou aritmética, ela é executada neste instante.
e) Escrita de dados: se o resultado da execução exigir uma escrita na memória ou periféricos, a CPU transfere o valor do registrador para o destino final.
f) Avaliação de Interrupções: após finalizar a execução de uma instrução, a CPU verifica se foi requisitada uma interrupção (Interrupt Request). Se sim, toma as providências necessárias. Se não, volta para **a)**.

Pelas responsabilidades da ULA e da UC, é possível perceber que a atividade **d** é executada pela ULA e todas as outras pela UC.

1.3. Registradores Usados pela UC

Assim como a ULA tem seus registradores especiais (*Acumulador* para armazenar os resultados e o *Flags* para indicar informações sobre a última operação executada), também a UC precisa de alguns registradores para funcionar correamente.

O primeiro deles vem da necessidade da UC saber onde está a próxima instrução a ser executada. Em outras aplavras, ela precisa de um registrador que indique a posição de

memória em que a próxima instrução do programa estará armazenado (para que ela possa realizar a busca de instrução). Este registrador sempre existe, em todos os computadores microprocessados, mas seu nome varia de uma arquitetura para outra. Normalmente este registrador é chamado de **PC**, de Program Counter (Contador de Programa).

Sempre que é iniciado um ciclo de processamento (descrito na seção anterior), uma a UC busca a próxima instrução na memória, na posição indicada pelo PC. Em seguida, o PC é atualizado para apontar para a próxima posição da memória (logo após a instrução), que deve indicar a instrução seguinte.

Bem, como foi visto anteriormente, a UC precisa analisar esta instrução antes de decidir o que fazer em seguida. Por esta razão, costuma existir um registrador especial para armazenar a última instrução lida, chamado **IR**, de Instruction Register (Registrador de Instruções).

Para conseguir ler e escrever dados em memórias e periféricos, a UC também precisa de um contato com o barramento, o que é feito através de registradores especiais, de armazenamento temporário, chamados **MAR**, de Memory Address Register (Registro de Endereço de Memória) e o **MBR**, de Memory Buffer Register (Registro de Buffer de Memória). Assim, quando é preciso escrever na memória (ou em um periférico), a UC coloca o endereço no registrador MAR, o dado no registrador MBR e comanda a transferência pelo barramento de controle. Quando for preciso ler da memória (ou do periférico), a UC coloca o endereço no MAR, comanda a leitura pelo barramento de controle e então recupera o valor lido pelo MBR.

Adicionalmente a estes registradores, as CPUs costumam ter outros registradores que podem facilitar sua operação e mesmo sua programação. Alguns destes são os **registradores de propósito geral**, que servem para armazenar resultados intermediários de processamento, evitando a necessidade de muitas escritas e leituras da memória quando várias operações precisarem ser executadas em seqüência, a fim de transformar os dados de entrada nos dados de saída desejados. O nome destes registradores costuma ser letras diversas como B, C, D...

Existem também os **registradores de pilha**, normalmente com nomes como **SP**, de Stack Pointer (Ponteiro da Pilha) ou **BP** (Base da Pilha), que servem para que uma pilha seja usada pelo processador, na memória. Em essência, é onde o endereço de retorno é armazenado, quando um desvio é feito em linguagem de máquina; afinal, é preciso saber para onde voltar após a realização de uma chamada de subrotina. A pilha que o processador fornece pode ser usada com outros objetivos, como passagem de parâmetros etc.

Quase todas as arquiteturas fornecem os **registradores de índices**, que são registradores que permitem acessar, por exemplo, posições de uma matriz. Ele guarda uma posição de memória específica e existem instruções que permitem acessar o n-ésimo elemento a partir daquela posição. Seus nomes variam muito de uma arquitetura para outra, como **IX**, de IndeX, **SI**, de Source Index (Índice Fonte) ou ainda **DI**, de Destination Index (Índice Destino).

Arquiteturas com segumento possuem ainda os **registradores de segmento**, que definem o endereço "zero" da memória para um determinado tipo de informação. A arquitetura x86, por exemplo, possui diversos registradores deste tipo: **CS**, de Code Segment (Segmento de Código), **DS**, de Data Segment (Segmento de Dados), **SS**, de Stack Segment (Segmento de Pilha) e **ES**, de Extra data Segment (Segmento de Dados Extra). Quando estes segmentos existem, os endereços usados nos índices, contador de programa e outros são "somados" com os endereços do segmento para que a posição real na memória seja calculada. Por exemplo:

SS = 10000h => Endereço do segmento da pilha
SP = 1500h => Endereço da pilha (dentro do segmento)
Endereço real da pilha = SS + SP = 10000h + 1500h = 11500h

Isso permite que um programa possa rodar em qualquer parte da memória, mesmo que ele tenha sido criado para ser executado no endereço 0h: basta eu indicar no registrador CS o endereço inicial de carregamento deste programa e, para todas as instruções deste programa, vai ser como se ele estivesse no endereço 0h. A CPU, com os registradores de sergento, são responsáveis pela tradução do endereço "virtual" para o endereço real.

Vale lembrar que cada arquitetura tem nomes distintos para estes resigtradores e em algumas delas existem ainda outros; além disso, alguns destes registradores até existem em algumas arquiteturas, mas não são visíveis para o programador, isto é, o registrador está lá e é usado pela CPU, mas o programador não tem acesso direto a eles (embora muitas vezes tenha acesso indireto, como sempre ocorre com os registradores MAR e MBR).

2. CICLO DE INSTRUÇÃO

Nas seções anteriores já foi descrito o ciclo de instrução, que é a seqüência de passos que a UC segue até que uma instrução seja executada. Nesta parte será apresentado um diagrama genérico que mostra todos os passos que um processador comum executa para executar suas instruções. Os principais subciclos são os de busca de instruções, busca de dados, execução e interrupção.

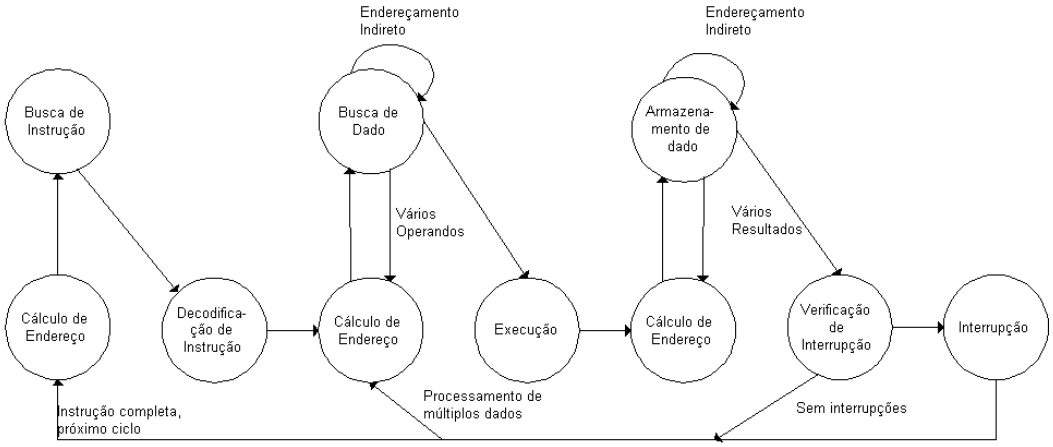


Figura 1 - Diagram de transição de estados do ciclo de instrução (STALLINGS, 2003)

Como é possível ver, o primeiro passo é a busca de instrução, onde a UC coloca o valor do PC no MAR, comanda leitura da memória e recebe o dado (que neste caso é uma instrução) pelo MBR, que em seguida copia para o IR.

Em seguida, a UC decodifica a instrução, avaliando se há a necessidade de busca de dados adicionais (por exemplo, se for uma instrução do tipo "ADD A,B", nenhum dado precisa ser buscado. Se houver a necessidade, o próximo passo é a busca do dado, onde a UC realiza o mesmo processo da leitura da instrução, mas agora para a leitura do dado. Esse processo é repetido até que todos os dados necessários tenham sido colocados em registradores.

O passo seguinte é a execução, onde a UC meramente comanda a ULA para executar a operação relevante. A ULA devolve um resultado em um registrador. Se este dado precisar ser armazenado externamente à CPU, a UC cloca este dado na MBR e o endereço destino na MAR e comanda a escrita, usando o barramento de controle. Se houver mais de um dado a armazenar, este ciclo é repetido.

Finalmente, a UC verifica se há *requisição de interrupção pendente*. Se houver, ela executa o ciclo da interrupção (que varia de arquitetura para arquitetura). Caso contrário, o funcionamento prossegue, com o cálculo do novo endereço de instrução e o ciclo recomeça.

3. A PIPELINE

A idéia de pipeline é a mesma da produção em série em uma fábrica: "quebrar" a produção de alguma tarefa em pequenas tarefas que podem ser executadas paralelamente. Isso significa que vários componentes contribuem para o resultado final, cada um executando sua parte.

Como foi possível ver pela seção anterior, existem vários passos em que a execução de uma instrução pode ser dividida. A idéia, então, é que cada um destes passos seja executado independentemente e por uma parte diferente da CPU, de forma que o processamento ocorra mais rapidamente.

Mas como isso ocorre? Imagine o processo explicado na seção 2. Nos momentos em que a comunicação com a memória é feita, por exemplo, a ULA fica ociosa. Nos momentos em que a ULA trabalha, a comunicação com a memória fica ociosa. Certamente o processamento linear não é a melhor forma de aproveitar os recursos.

Imagine então que emos duas grandes etapas (simplificando o processo explicado anteriormente): a etapa de busca e a etapa de execução. Se tivermos duas unidades na UC, uma para cuidar da busca e outra para cuidar da execução, enquanto a execução de uma instrução está sendo feita, a seguinte já pode estar sendo buscada! Observe as seqüênciasa seguir.

	SEM pipeline		COM pipeline	
Seqüência no Tempo	Busca	Execução	Busca	Execução
0	I1	-	I1	-
1	-	I1	I2	I1
2	I2	-	I3	I2
3	-	I2	I4	I3
4	I3	-	I5	I4

Observe que no tempo que foram executadas 2 instruções sem pipeline, com o pipeline de 2 níveis foram executadas 4 instruções. Entretanto, isso é uma aproximação grosseira, pois os tempos de execução de cada um destes estários é muito diferente, sendo que o aproveitamento ainda não é perfeito. Para um bom aproveitamente, precisamos dividir as tarefas em blocos que tomem mais ou menos a mesma fatia de tempo. Este tipo específico de pipeline é chamado de "Prefetch" (leitura antecipada).

Se quebrarmos, por exemplo, a execução em 6 etapas: Busca de Instrução (BI), Decodificação de Instrução (DI), Cálculo de Operandos (CO), Busca de Operandos (BO), Execução da Instrução (EI) e Escrita de Operando (EO), termos etapas mais balanceadas com relação ao tempo gasto. Observe na tabela abaixo o que ocorre:

	SEM Pipeline						COM Pipeline					
T	BI	DI	CO	BO	EI	EO	BI	DI	CO	BO	EI	EO
0	I1	-	-	-	-	-	I1	-	-	-	-	-
1	-	I1	-	-	-	-	I2	I1	-	-	-	-
2	-	-	I1	-	-	-	I3	I2	I1	-	-	-
3	-	-	-	I1	-	-	I4	I3	I2	I1	-	-
4	-	-	-	-	I1	-	I5	I4	I3	I2	I1	-
5	-	-	-	-	-	I1	I6	I5	I4	I3	I2	I1
6	I2	-	-	-	-	-	I7	I6	I5	I4	I3	I2
7	-	I2	-	-	-	-	I8	I7	I6	I5	I4	I3
8	-	-	I2	-	-	-	I9	I8	I7	I6	I5	I4
9	-	-	-	I2	-	-	I10	I9	I8	I7	I6	I5
10	-	-	-	-	I2	-	I11	I10	I9	I8	I7	I6
11	-	-	-	-	-	I2	I12	I11	I10	I9	I8	I7
12	I3	-	-	-	-	-	I13	I12	I11	I10	I9	I8

Basicamente, no tempo que foram executadas 2 instruções sem pipeline, foram executadas 8 instruções com pipeline. É claro que o tempo de execução de uma instrução sem pipeline neste caso de 6 estágios é aproximadamente o mesmo tempo de execução da mesma instrução sem pipeline no caso com 2 estágios; Isso ocorre porque, obviamente, cada um dos 6 estágios deste caso toma um tempo muito menor que cada um dos dois estágios do modelo anterior.

Bem, mas se o número de estágios aumenta o desempenho, porque não usar o máximo possível? Por algumas razões. Uma delas é que, a partir de um determinado número de estágios a quebra pode acabar fazendo com que dois estágios passem a gastar mais tempo de

execução do que o estágio original que foi dividido. Mas esta não é a razão fundamental: existe um gargalo mais evidente no sistema de pipelines: ele pressupõe que as instruções são independentes entre si; assim, para que o pipeline tenha o desempenho apresentado, uma instrução a ser executada não pode depender do resultado das instruções anteriores.

Quando uma instrução depende do resultado das anteriores, teremos alguns estágios "esperando" a execução da outra instrução terminar para que a "cadeia de montagme possa ter prosseguimento. Vamos dar um exemplo. Imagine que A instrução I2 dependa do resultado da instrução I1 para ser executada. Então, o que vai acontecer no pipeline é descrito a seguir:

	COM Pipeline					
T	BI	DI	CO	BO	EI	EO
0	I1	-	-	-	-	-
1	I2	I1	-	-	-	-
2	I3	I2	I1	-	-	-
3	I4	I3	I2	I1	-	-
4	I5	I4	I3	I2	I1	-
5	I5	I4	I3	I2	-	I1
6	I6	I5	I4	I3	I2	-
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10	I10	I9	I8	I7	I6	I5
11	I11	I10	I9	I8	I7	I6
12	I12	I11	I10	I9	I8	I7

Observe que, comparando com o quadro anterior, uma instrução a menos foi processada. Isso é pior ainda quando uma instrução do tipo "desvio condicional" precisa ser interpretada; isso porque a posição de leituras da próxima instrução vai depender da execução de uma instrução. Neste caso, quando ocorre este tipo de desvio, o pipeline é esvaziado e perde-se uma boa parte do desempenho.

Ocorre que a chance destes "problemas" acontecerem e os atrasos causados por eles aumentam com o número de níveis do pipeline. Desta forma, um número excessivo de níveis de pipeline podem acabar por degradas o desempenho, além de fazer com que uma CPU aqueça mais e mais. Para entender isso (o aquecimento) pense em uma fábrica: quanto mais funcionários, mais confusa é a movimentação dentro da fábrica. Nos circuitos, os níveis de pipeline fazem o papel dos funcionários da linha de montagem e, quanto maior número de movimentações internas de sinais, maior é o calor gerado.

A subdivisão excessiva dos pipelines foi o que matou a linha Intel Pentium IV, que foi abandonada. A Intel precisou retroceder sua tecnologia à do Pentium M (Pentium III móvel) e continuar o projeto em outra direção, com menos níveis de pipeline, o que deu origem aos processadores Pentium D, Core 2 Duo e os mais atuais Core i.

4. MICROPROGRAMAÇÃO

Apesar de os estágios de execução de uma instrução - como a busca de instrução, decodificação de instrução etc. - parecerem simples, sua implementação com circuitos lógicos é bastante complexa e, em geral, impõe uma grande dificuldade para modificações e expansões no conjunto de instruções.

Uma proposta para solucionar este problema foi a **microprogramação**. A microprogramação foi proposta ainda na década de 1950, mas, devido ao custo da tecnologia, que exigia memória rápida e barata, só na década de 1960 é que foi implementada em sistemas comerciais, tendo sido o IBM Série 360 o primeiro a usar um processador com essa característica.

Mas o que é a microprogramação?

Vamos relembrar, primeiramente, o que ocorre dentro do estágio de execução de uma instrução. Cada estágio de execução de busca de instrução é executado por várias tarefas e, assim que elas são terminadas, passa-se ao próximo estágio de execução.

Por exemplo: o estágio de busca de instrução pressupõe:

a) Configurar o barramento de endereços (MAR) com o valor do registrador PC;
b) Configurar o barramento de controle para leitura de memória;
c) Aguardar um intervalo referente ao tempo de resposta da RAM
d) Ler o valor do barramento de dados (MBR) para o IR
e) Remover os sinais do barramento de endereços e controle

Só depois disso é que a CPU parte para o próximo estágio. É possível, ainda, quebrar cada uma dessas tarefas em várias outras que envolvem apenas sinais elétricos e que podem ser executadas paralelamente. **Essas** pequenas tarefas que envolvem apenas sinais elétricos e que podem ser executasdas paralelamente são chamadas de **microinstruções**, e são compartilhadas para a execução de diversas (macro)instruções diferentes.

No fundo, é como se existisse uma espécie de Unidade de Controle dentro da Unidade de Controle, que "quebra" as (macro)instruções em tarefas muito menores e mais simples - como já dito, envolvendo apenas sinais elétricos e decisões lógicas mais elementares - e as executa para que, no conjunto, resultem no resultado da macroinstrução. Assim, cada instrução que a UC tem que executar é, na prática, o resultado de um programa gravado na CPU, chamado de **microprograma**. O conjunto de todos os microprogramas que formam o conjunto de instruções é o **firmware** da CPU.

5. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 12: Introdução ao Paralelismo:

Processadores Superescalares

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos fundamentais da arquitetura superescalar e as bases do paralelismo de processamento.

Bibliografia:

- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

Na aula anterior foi apresentada uma forma de organização das partes de um processador chamada "pipeline", que permitia que praticamente todos os elementos da CPU estivessem funcionando ao mesmo tempo. Apesar de acelerar bastante o processamento (em especial se o código foi otimizado para isso), o pipeline ainda representa uma forma sequêncial de processamento.

Na aula de hoje começaremos a estudar o processamento paralelo, conhecendo seus fundamentos iniciais. Iniciaremos com a arquitetura superescalar, que foi introduzida com o PowerPC e o Pentium II e, em seguida, serão apresentados os fundamentos do multiprocessamento.

1. ARQUITETURA SUPERPIPELINE

Na aula passada foi apresentado o conceito de "pipeline", isto é, uma forma de organizar o funcionamento da CPU para que ela seja capaz de processar as instruções mais rapidamente, minimizando a ocorrência de circuitos ociosos.

	SEM pipeline		COM pipeline	
Seqüência no Tempo	Busca	Execução	Busca	Execução
0	I1	-	I1	-
1	-	I1	I2	I1
2	I2	-	I3	I2
3	-	I2	I4	I3
4	I3	-	I5	I4

Alguns pesquisadores, entretanto, perceberam que, quando se subdividia a pipeline em um maior número de níveis, alguns destes estágios precisavam de muito menos tempo de execução que um ciclo de clock.

	SEM Pipeline						COM Pipeline					
T	BI	DI	CO	BO	EI	EO	BI	DI	CO	BO	EI	EO
0	I1	-	-	-	-	-	I1	-	-	-	-	-
1	-	I1	-	-	-	-	I2	I1	-	-	-	-
2	-	-	I1	-	-	-	I3	I2	I1	-	-	-
3	-	-	-	I1	-	-	I4	I3	I2	I1	-	-
4	-	-	-	-	I1	-	I5	I4	I3	I2	I1	-
5	-	-	-	-	-	I1	I6	I5	I4	I3	I2	I1
6	I2	-	-	-	-	-	I7	I6	I5	I4	I3	I2
7	-	I2	-	-	-	-	I8	I7	I6	I5	I4	I3
8	-	-	I2	-	-	-	I9	I8	I7	I6	I5	I4
9	-	-	-	I2	-	-	I10	I9	I8	I7	I6	I5
10	-	-	-	-	I2	-	I11	I10	I9	I8	I7	I6
11	-	-	-	-	-	I2	I12	I11	I10	I9	I8	I7
12	I3	-	-	-	-	-	I13	I12	I11	I10	I9	I8

Assim, propôs-se o uso de um duplicador (ou até triplicador) de clock interno na CPU, de maneira que muitos destes estágios do pipeline pudessem ser finalizados na metade do tempo, acelerando ainda mais o processamento final.

É importante notar, porém, que o aumento de desempenho é oriundo de um menor tempo de processamento de cada instrução, mas elas ainda estão sendo executadas uma a uma.

2. ARQUITETURA SUPERESCALAR

A Pipeline, que chegou aos computadores caseiros com o Pentium, trouxe um grande domínio da Intel no mercado de computadores domésticos, mas a empresa sentia que precisava ampliar ainda mais a capacidade de seus processadores, que vinham sendo seguidos de perto pelos de sua principal concorrente nesse segmento: a Advanced Micro Devices, AMD.

Expandindo a idéia de processar simultaneamente os diferentes estágios de insturções diferentes, foi sugerida a implementação de várias pipelines para a CPU, de maneira que várias instruções pudessem, de fato, serem executadas ao mesmo tempo, desde que fossem independentes. O princípio é o mesmo da pipeline já visto, mas nessas novas CPUs foram inseridas, em uma mesma CPU, diversas pipelines.

Essa estratégia é conhecida como **paralelismo em nível de instruções**, porque ela é capaz de detectar instruções independentes e executá-las simultaneamente. Em outras palavras, a Unidade de Controle, ao ler instruções, vai "jogando-as" para os diferentes

pipelines, alternadamente. Desta maneira, diversas instruções são processadas em um intervalo de tempo que só seria possível processar uma instrução, caso existisse um único pipeline.

2.1. O que São Instruções Independentes?

Talvez seja mais mais simples dizer o que são **instruções dependentes**. Instruções dependentes são aquelas cujo resultado de uma depende do resultado de outra e, assim, a ordem de execução deve ser preservada. Por exemplo:

```
LD    A,17
ADD   A,20
```

Resulta no valor 37 armazenado no registrador A. Entretanto, para que a instrução ADD A,20 possa ser executada, a instrução LD A,17 já precisa ter terminado.

Esse tipo de problema, como já foi comentado, atrapalha até mesmo o pipeline simples, mas vira um problema muito sério quando se usa uma arquitetura superescalar, pois o uso excessivo deste tipo de instruções acaba desperdiçando desempenho da CPU.

Como é impossível evitar essa dependência em muitos casos, a solução são os "compiladores otimizadores" ou mesmo o uso de Unidades de Controle mais inteligentes. O que eles fazem? Simples: analisam e **mudam a ordem do código** de acordo com as dependências. Por exemplo:

```
LD    A,17        ; Armazena 17 em A
ADD   A,20        ; Soma 20 em A
LD    C,A         ; Guarda resultado (37) em C
LD    A,30        ; Armazena 30 em A
ADD   A,10        ; Soma 10 em A
LD    D,A         ; Armazena o valor de A em D
```

Neste código temos dois blocos "quase" independentes:

```
LD    A,17        ; Armazena 17 em A
ADD   A,20        ; Soma 20 em A
LD    C,A         ; Guarda resultado (37) em C

LD    A,30        ; Armazena 30 em A
ADD   A,10        ; Soma 10 em A
LD    D,A         ; Armazena o valor de A em D
```

Estes dois blocos poderiam ser paralelizados trocando-se o registrador A do segundo bloco por outro registrador (se isso for possível), como o registrador B:

```
LD    A,17      ; Armazena 17 em A
ADD   A,20      ; Soma 20 em A
LD    C,A       ; Guarda resultado (37) em C
LD    B,30      ; Armazena 30 em B
ADD   B,10      ; Soma 10 em B
LD    D,B       ; Armazena o valor de B em D
```

Agora os blocos são totalmente independentes, embora as insutrções em cada um deles ainda precise ser executada na sequência. Como otimizar isso para o uso de arquitetura superescalar? Simples: vamos intercalar os blocos!

```
LD    A,17      ; Armazena 17 em A
LD    B,30      ; Armazena 30 em B
ADD   A,20      ; Soma 20 em A
ADD   B,10      ; Soma 10 em B
LD    C,A       ; Guarda resultado (37) em C
LD    D,B       ; Armazena o valor de B em D
```

Agora essas instruções podem ser processadas com otimalidade em uma arquitetura com duas pipelines!

Existem situações, entretanto, que a dependência é mais complexa, como as que envolvem deslocamento de execução, com JMP e, especialmente, com CALL. O CALL é uma forma de executar uma subrotina, o que significa que, quando se executa um CALL, em algum momento ocorrerá um RETurn. Ocorre que a instrução RET pode ter comportamento variado, dependendo do resultado de uma comparação. Por exemplo:

```
CP    A,B
RET   Z
```

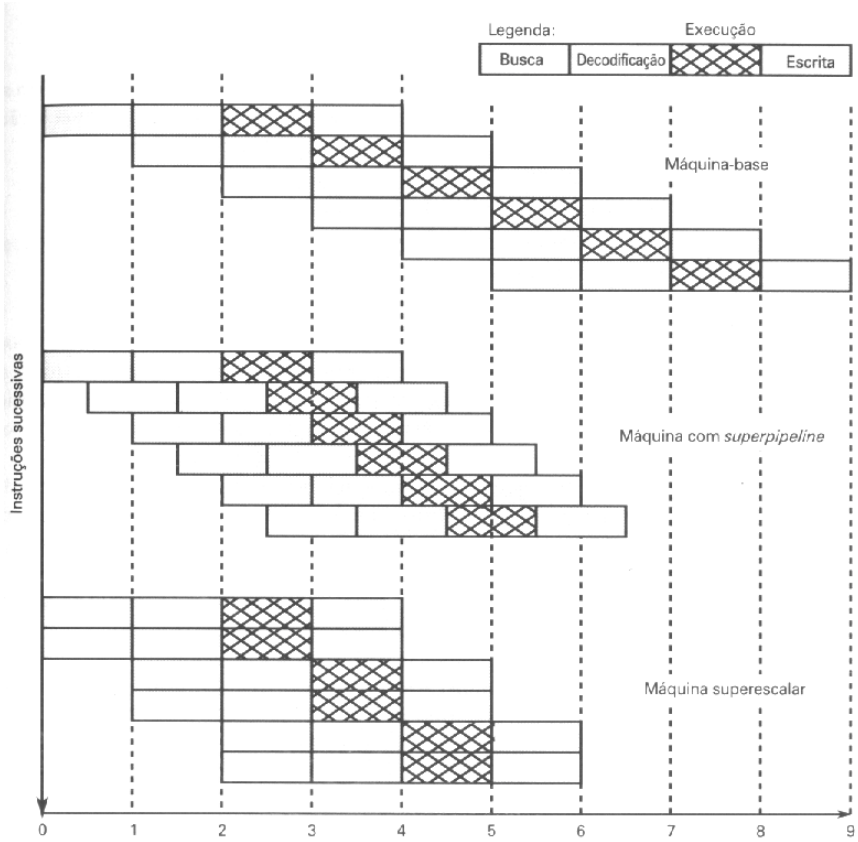
Esse RETurn Z indica que só haverá o retorno caso A e B sejam iguais. Esse tipo de dependência nem sempre é fácil de resolver, mas os bons compiladores e processadores são capazes de usar artifícios para contornar estas situações.

3. COMPARAÇÃO: PIPELINE x SUPERPIPELINE x SUPERESCALAR

O gráfico a seguir mostra o processamento superescalar comparado com o processamento em pipeline e superpipeline. Observe que, diferentemente das arquiteturas pipeline e superpipeline, na arquitetura superescalar temos, de fato, mais de uma instrução sendo processada ao mesmo tempo.

Isso é muito útil para processamento dados como imagens e outros tipos de matrizes, onde é possível processar diversos dados diferentes ao mesmo tempo pois, em geral, o

processamento de cada pixel ou elemento da matriz é independente do processamento dos outros pixels/elementos da matriz.



3.1. Lógica de Execução SuperEscalar

A sequência que o processador segue para executar um programa superescalar é indicada na figura a seguir:

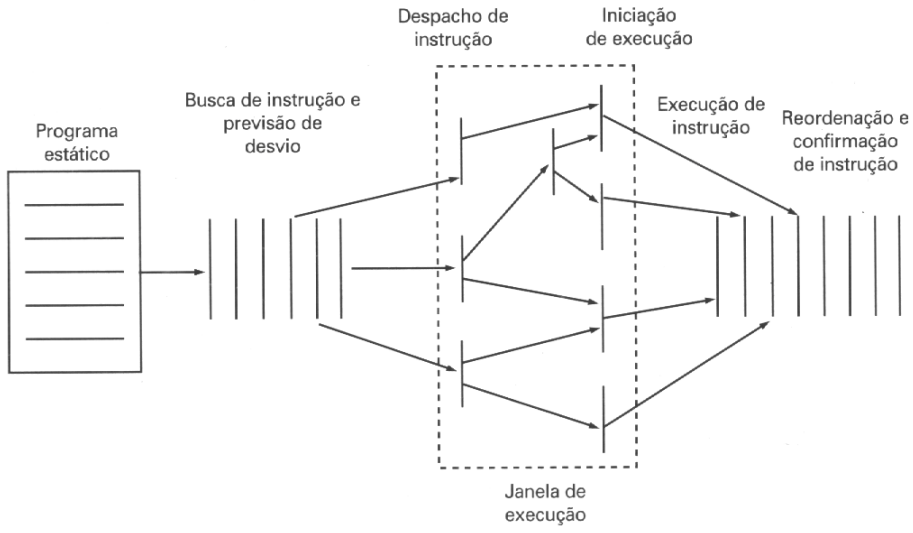


Figura 13.6 Representação conceitual de processamento superescalar (Smith e Sohi, 1995).

4. FUNDAMENTOS DO MULTIPROCESSAMENTO

O processamento escalar permite um certo nível de processamento paralelo, chamado "paralelismo em nível de instruções", como já citado. Entretanto, a idéia de execução de múltiplas tarefas simultaneamente pode ser expandida ainda mais.

De fato, o conceito de processamento paralelo é usado, atualmente, para se referir ao "paralelismo em nível de processos", em que diferentes processos são executados simultaneamente.

Existem três tipos básicos de processamento paralelo em uso corrente na atualidade:

SMP - Symetric MultiProcessing: multiprocessamento simétrico, em que vários processadores compartilham a mesma infraestrutura de computador (dispositivos e memória).

Clusters - multiprocessamento usando vários computadores de forma que eles todos trabalhem como se fossem apenas um.

NUMA - Non-Uniform Memory Access: acesso não uniforme à memória, em que vários processadores compartilham a mesma infraestrutura de computador, mas cada um deles trabalhando, em geral, em uma região específica da memória, diferente para cada um deles.

Na próxima aula veremos um pouco mais de detalhes sobre cada um desses, no momento iremos entender a taxonomia dessas formas de processamento paralelo, proposta por Flynn (1972 *apud* Stallings, 2003):

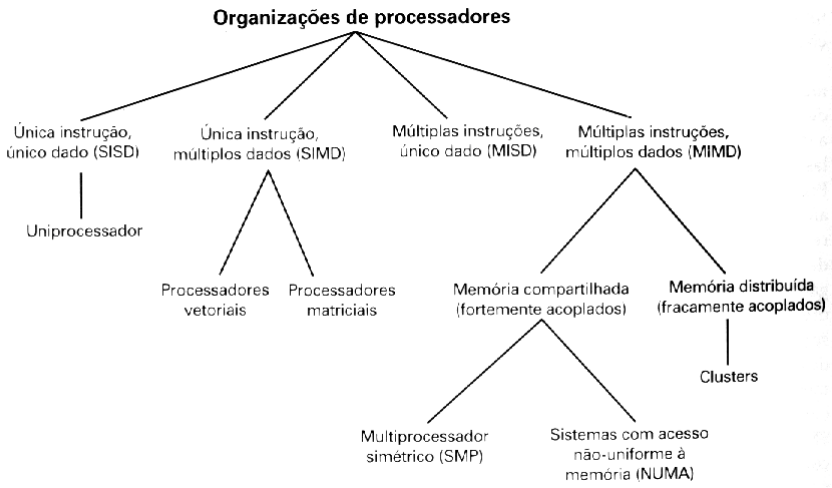
SISD - Single Instruction, Single Data (Única Instrução, Único Dado): sistemas monoprocessados. Pipelines e processadores escalares entram nessa categoria.

SIMD - Single Instruction, Multiple Data (Única Instrução, Múltiplos Dados): sistemas de processamento vetorial e matricial.

MISD - Multiple Instructions, Single Data (Múltiplas Instruções, Único Dado): seria um sistema em que diferentes conjuntos de instruções se aplicassem ao mesmo dado, simultaneamente. É um conceito teórico.

MIMD - Multiple Instructions, Multiple Data (Múltiplas Instruções, Múltiplos Dados): várias unidades de processamento, processando diferentes informações, como no caso dos sistemas SMP e clusters.

A figura a seguir esquematiza a relação entre as classificações e diferentes sistemas. Cada uma dessas formas de multiprocessamento será descrita na próxima aula, mas a maioria delas já foi brevemente apresentada, com exceção do processamento vetorial/matricial. O **processamento vetorial** se refere a sistemas que precisam executar uma única instrução sobre um grande conjunto de dados como, por exemplo, o processamento de uma imagem ou a simulação de partículas. Os computadores de processamento vetorial são frequentemente chamados de **supercomputadores**.



5. O CONCEITO DO COMPUTADOR COMPLETO (WHOLE COMPUTER)

Tanto os sistemas SMP/NUMA quando os Clusters são sistemas com uma arquitetura MIMD, isto é, são capazes de processar múltiplos dados com instruções diferentes de maneira simultânea. A diferença mais comum entre eles fica explícita pelo conceito de **computador completo**.

Um **computador completo** é uma unidade de processamento totalmente funcional, isto é, com sua própria memória, dispositivos e cpu. Os sistemas **SMP** e **NUMA** possuem multiprocessamento mas não são compostos por vários computadores completos, já que uma grande parte dos recursos de cada uma das unidades de processamento é compartilhado com as demais.

Já os **Clusters** são, em geral, compostos pela união de vários computadores completos, que trabalham em conjunto como se fossem um único computador. Cada um dos computadores completos que compõem um cluster recebe o nome de **nó**. Assim, cada nó possui sua própria memória e seus dispositivos, sendo uma unidade de processamento independente.

Resumidamente, um cluster com **N** nós pode ser separado em **N** computadores independentes. Um sistema SMP ou NUMA com **N** unidades de processamento não pode ser dividido em **N** computadores independentes.

6. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 13: Paralelismo:
SMP e Processamento Vetorial

Prof. Daniel Caetano

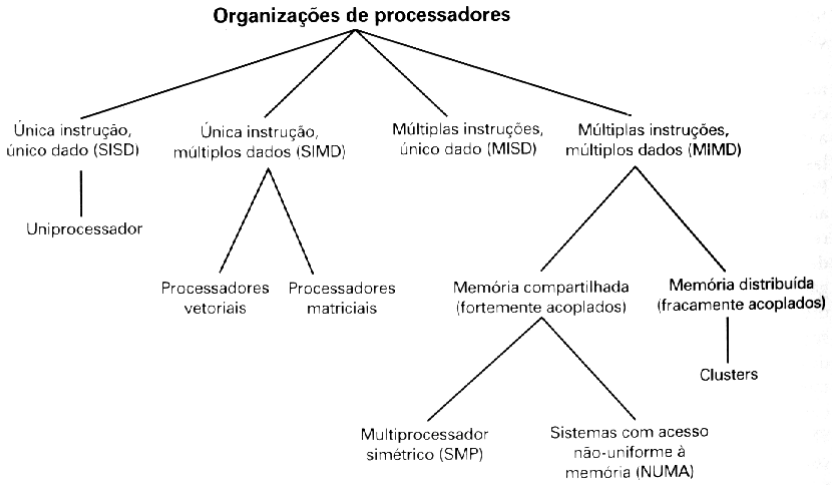
Objetivo: Apresentar os conceitos fundamentais da arquitetura SMP e alguns detalhes do processamento Vetorial.

Bibliografia:

- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

Como visto anteriormente, há diversas formas de se realizar processamentos paralelos, como indicado na figura a seguir. Nesta aula serão apresentados alguns detalhes sobre a arquitetura SMP e mais algumas informações sobre os tipos de cálculos realizados pelos processadrores vetoriais.



1. ARQUITETURA SMP (SYMMETRIC MULTI PROCESSING)

Até bem pouco tempo atrás, a arquitetura mais comum para os computadores pessoais tinha um único processador. Com a queda dos custos e a sempre crescente demanda, foi introduzida nos computadores pessoais a arquitetura SMP, que pode ser definida com as seguintes características:

1. Existência de dois ou mais processadores com capacidades comparáveis.
2. Todos os processadores compartilham a memória um barramento ou outro esquema de conexão qualquer, de forma que o tempo de acesso à memória é praticamente o mesmo para cada processador.
3. Os dispositivos de e/s são compartilhados, podendo ser acessados por canais comuns ou por canais distintos que levem aos mesmos dispositivos.
4. Todos os processadores podem executar as mesmas funções (simetria!)
5. O sistema é controlado por um sistema operacional integrado, que permite a interação entre os processadores e seus prorgamas - em nível de tarefas, arquivos e dados.

Ou seja, como decorrência do item 5, é o sistema operacional quem deve escalonar os processos e threads entre os diversos processadores.

As potenciais vantagens dos sistemas SMP são:

* **Desempenho**: desempenho nítidamente maior que o de máquinas monoprocessadas.
* **Disponibilidade**: como todos os processadores são equivalentes, a falência de 1 dos processadores não impede a máquina de continuar trabalhado, ainda que mais lentamente.
* **Crescimento Incremental**: é possível aumentar o desempenho do sistema adicionando processadores.
* **Escalabilidade**: os fabricantes podem oferecer uma larga faixa de produtos, com custos e desempenhos diferentes.

Mais recentemente, é possível até mesmo falar em maior economia de energia, visto que os equipamentos mais modernos são capazes de **desligar** as CPUs inativas, bem como **reduzir sua velocidade de processamento** quando a capacidade não é exigida.

2. ORGANIZAÇÃO DA ARQUITETURA SMP

O principal desafio da arquitetura SMP é reduzir os conflitos dos diversos processadores no seu acesso à memória. Uma vez que cada um dos processadores é independente e todos eles precisam acessar os dados, a organização para coordenar os acessos à memória é determinante para um bom desempenho.

Existem diversas estrturas para implementar essa coordenação na arquitetura SMP. Elas podem ser classificadas da seguinte forma: barramento comum, memória com múltiplas portas ou unidade de controle central.

2.1. Unidade de Controle Central

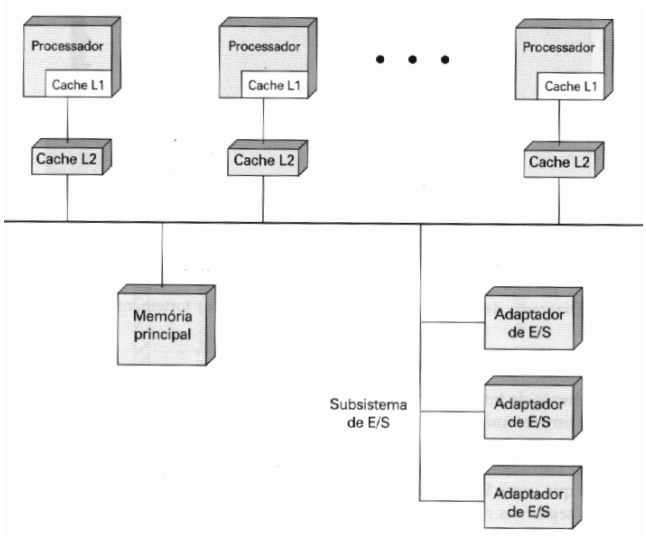
Uma vez que a grande dificuldade da arquitetura SMP é coordenar o acesso à memória pelas diferentes CPUs, uma das primeiras estratégias foi centralizar a Unidade de Controle: assim, existia uma unidade de controle única, mas várias unidades de processamento (ULA/FPU).

Uma das vantagens desta estrutura é uma coordenação completa dos acessos, já que a única unidade de controle tem capacidade de organizar todo o processo. Por outro lado, a Unidade de Controle, por si só, já é um circuito demasiadamente complexo e as unidades de controle centralizadas para vários processadores eram ainda mais complexas, em especial se fosse desejado um número dinâmico de unidades de processamento.

Este era o tipo de organização mais usado pela IBM na década de 1960 e 1970, mas, devido a alta complexidade, hoje ele praticamente caiu em desuso.

2.2. Barramento Comum (ou Tempo Compartilhado)

Na estrutura de barramento comum, também conhecida como "de tempo compartilhado", existem várias CPUs completas, todas elas conectadas ao mesmo barramento. As interconexões são praticamente as mesmas de um sistema monoprocessado, mas o gerenciamento do barramento é um pouco mais complexo (apesar de ele se manter essencialmente o mesmo), pois passa a ser necessário **distinguir os diferentes módulos** (e processadores), mantendo o **controle de arbitragem** e o **compartilhamento de tempo**.



Há diversas vantagens nessa estrutura:

*** Simplicidade**: é a forma mais simples de organizar um sistema multiprocessador, fica tudo muito parecido com um sistema monoprocessador.

*** Flexibilidade**: geralmente é fácil expandir o número de processadores no barramento.

*** Confiabilidade**: Como o barramento é passivo, a falha em um dispositivo/cpu não precisa, necessariamente, comprometer todo o funcionamento do sistema.

Há desvantagens, entretanto: há um grande gargalo no acesso a memória, já que todas as CPUs precisam acessar pelo mesmo barramento. Esse gargalo é, normalmente, mitigado

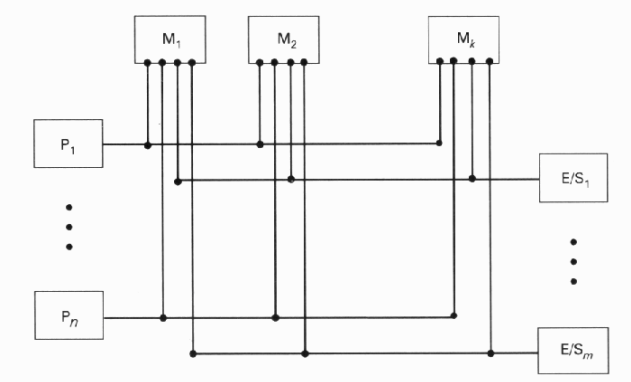
por uma memória Cache para cada processador. Infelizmente, isso introduz outras complexidades.

Como vários processadores podem estar trabalhando nas mesmas áreas da memória, se um processador escreve na memória, ele invalida os dados do cache de outro processador. Assim, é preciso haver também uma comunicação direta entre os processadores, para que os ajustes de cache sejam feitos, quando necessário.

Esses ajustes são feitos seguindo um protocolo de **coerência de cache**. Existem diversos mecanismos conhecido para isso, mas foge ao escopo do curso detalhá-los.

2.3. Memória com Múltiplas Portas

Esse tipo de estruturação fornece um barramento de acesso à memória para cada componente do sistema, permitindo que, em tese, todos os dispositivos possam solicitar acesso a uma mesma memória.



A grande vantagem é a redução dos conflitos de arbitragem de uso de barramento mas, por outro lado, exige o uso de memória com múltiplas portas (multiport memory) que são, usualmente, mais caras. A razão para que elas sejam mais caras é que a própria memória é responsável por resolver os conflitos que surgem.

Finalmente, esse tipo de estrutura dificulta o controle da coerência do cache, sendo necessário um mecanismo direto de controle por parte das CPUs, pois não existe uma forma conveniente dos outros processadores detectarem esses problemas pelo próprio barramento.

3. PROCESSAMENTO VETORIAL

Na aula passada foi comentada a existência de processadores vetoriais e matriciais. Esses processadores são usados em sistemas chamados **supercomputadores**.

Foge ao escopo do curso tratar detalhes sobre o funcionamento interno destes processadores, mas cabem alguns esclarecimentos sobre sua utilidade.

Há uma série de problemas, nas áreas de aerodinâmica, sismologia, meteorologia, física atômica - dentre outras - que exigem uma alta precisão numérica em cálculos de ponto flutuante, executando programas que aplicam repetidamente complexos conjuntos de operações em grandes vetores de números.

A maioria destes problemas pertence à categoria dos problemas de simulação de campos contínuos, isto é, um problema em que a situação física pode ser descrita por uma superfície ou uma região em 3 dimensões. Exemplos são a evolução climática na atmosfera terrestre, o campo de temperaturas e velocidades no jato de propulsão de um foguete ou mesmo as linhas de corrente de deslocamento de ar ao redor de uma asa de avião.

Um simples cálculo de soma destes problemas é expresso na forma de matrizes:

$$\begin{bmatrix} 1{,}5 \\ 7{,}1 \\ 6{,}9 \\ 100{,}5 \\ 0 \\ 59{,}7 \end{bmatrix} + \begin{bmatrix} 2{,}0 \\ 39{,}7 \\ 1000{,}003 \\ 11 \\ 21{,}1 \\ 19{,}7 \end{bmatrix} = \begin{bmatrix} 3{,}5 \\ 46{,}8 \\ 1006{,}903 \\ 111{,}5 \\ 21{,}1 \\ 79{,}4 \end{bmatrix}$$
$$A \quad + \quad B \quad = \quad C$$

É fácil imaginar porque o paralelismo de instruções simples sobre múltiplos dados (SIMD) ajuda neste tipo de cálculo. Quando o procedimento envolve multiplicação de matrizes, alguns truques são necessários, mas o paralelismo também ajuda.

Computadores capazes de resolver estes problemas com multiprocessamento vetorial/matricial são poucos no mundo, cada unidade custando alguns "poucos" milhões de dólares. Basicamente eles só existem nos grandes centros de pesquisa, onde são realmente necessários.

Como, mesmo com estes computadores, este tipo de problema requer, para resolver um problema simples, da ordem de 10^{13} operações aritméticas, tomando cerca de 2 a 3 dias para resolver alguns problemas não tão complexos, esta é, ainda, uma área em muito estudo e que, certamente, terá avanços num futuro próximo.

4. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
MURDOCCA, M. J; HEURING, V.P. **Introdução à Arquitetura de Computadores**. S.I.: Ed. Campus, 2000.

Unidade 14: Arquiteturas CISC e RISC

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos das arquiteturas CISC e RISC, confrontando seus desempenhos.

Bibliografia:

- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

Até o fim da década de 1970, praticamente todos os arquitetos de processadores e computadores acreditavam que melhorar os processadores estava diretamente relacionado ao aumento na complexidade das instruções (instruções que realizam tarefas cada vez maiores) e modos de endereçamento (acessos a estruturas de dados complexas diretamente pelas linguagens de máquina).

Sua crença não era descabida: como os tempos de acesso à memória eram bastante altos, reduzir o número de instruções a serem lidas era algo bastante positivo, já que reduzia a necessidade de comunicação com a memória. Entretanto, isso não perdurou: foi percebido que muitas destas instruções mais complexas praticamente nunca eram usadas, mas a existência das mesmas tornava outras instruções mais lentas, pela dificuldade em se implementar um pipeline adequado para processar igualmente instruções complexas e instruções simples.

Com o passar do tempo e o aumento da velocidade das memórias, os benefícios dos processadores com insturções complexas (CISC - Complex Instruction Set Computer) pareceu diminuir ainda mais, sendo que a tendência de novas arquiteturas acabou por ser uma eliminação destas instruções, resultando em processadores com um conjunto reduzido de instruções (RISC - Reduced Instruction Set Computer).

Nesta aula serão apresentadas algumas diferenças entre estas arquiteturas, vantanges e desvantagens, além da inerente controvérsia relacionada.

1. CISC - COMPLEX INSTRUCTION SET COMPUTER

Nos primórdios da era computacional, o maior gargalo de todo o processamento estava na leitura e escrita de dispositivos externos ao processador, como a memória, por exemplo. Como um dos acessos à memória indispensáveis para que o computador funcione é

a leitura de instruções (chamado de *busca de instrução*), os projetistas perceberam que se criassem instruções que executavam a tarefa de várias outras ao mesmo tempo seriam capazes de economizar o tempo de busca de instrução de praticamente todas elas.

Por exemplo, caso alguém desejasse copiar um bloco da memória de 500h bytes do endereço 1000h para o endereço 2000h, usando instruções simples teria de executar o seguinte código (em Assembly Z80):

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
COPIA:      LD    A, (HL)           ; Carrega byte da origem
            INC   HL                ; Aponta próximo byte de  origem em HL
            LD    (DE), A           ; Coloca byte no destino
            INC   DE                ; Aponta próximo byte de destino em HL
            DEC   BC                ; Decrementa 1 de BC
            LD    A,B               ; Coloca valor de B em A
            OR    C                 ; Soma os bits ligados de C em A
            ; Neste ponto A só vai conter zero se B e C eram zero
            JP    NZ,COPIA          ; Continua cópia enquanto A != zero
```

Isso é suficiente para realizar a tal cópia, mas foram gastas 8 instruções no processo, com um total de 10 bytes (de instruções e dados) a serem lidos da memória (fora as leituras e escritas de resultados de operações, como as realizadas pelas instruções LD A,(HL) e LD (DE),A). Ora, se o custo de leitura dos bytes da memória é muito alto, faz todo o sentido criar uma instrução que realiza este tipo de cópia, certo? No Z80, esta instrução se chama LDIR e o código acima ficaria da seguinte forma:

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
            LDIR                    ; Realiza cópia
```

A rotina de cópia passou de 8 instruções (com 10 bytes no total) para uma única instrução, que usa 2 bytes... definitivamente, um ganho substancial no que se refere à redução de tempo gasto com leitura de memória.

Como é possível ver, os projetistas tinham razão com relação a este aspecto. Entretanto, após algum tempo passou-se a se observar que estes ganhos eram limitados, já que o número de vezes que estas instruções eram usadas era mínimo. Um dos papas da arquitetura de computadores, Donald Knuth, fez uma análise das instruções mais usadas nos programas tradicionais. Os resultados foram:

Atribuição:	47%
If/Comparação:	23%
Chamadas de Função:	15%
Loops:	6%
Saltos simples:	3%
Outros:	7%

Observe que a rotina representada anteriormente se enquadra na categoria "Loops", que tem uma ocorrência que toma aproximadamente 6% de um programa. Considerando que é um loop bastante específico: um loop de cópia, provavelmente este porcentual é ainda menor. Possivelmente, substancialmente menor.

Com o surgimento dos pipelines, isso passou a causar algum problema; a existência destas instruções tornava a implementação do pipeline mais complexa e menos eficiente para todas as instruções (incluindo as de atribuição, que são a grande maioria de um programa!). Mas por que ocorre isso?

O que acontece é que instruções complexas como o LDIR apresentado representam uma severa complexidade para otimização por pipeline. Como visto, a implementação de um pipeline espera que todas as instruções possuam mais ou menos o mesmo número de estágios (que compõem o pipeline), estagios estes que normalmente são: *busca de instrução*, *interpretação de instrução*, *busca de operandos*, *operação na ULA* e *escrita de resultados*. Fica, claramente, difícil distribuir uma instrução como LDIR nestas etapas de uma maneira uniforme. Por conseqüência, um pipeline que otimize a operação de uma instrução LDIR (supondo que ele seja possíve), terá um certo número de estágios terão que *ignorar* as instruções simples, mas ainda assim consumirão tempo de processamento.

Este tipo de problema sempre depôs contra o uso instruções complexas; entretanto, se as memórias forem lentas o suficiente para justificar uma economia de tempo com a redução de leitura de instruções, o uso de arquitetura CISC (com instruções complexas) possa até mesmo ser justificado, em detrimento do uso de Pipeline (ou com o uso de um Pipeline "menos eficiente").

Mas o que tem ocorrido é um crescimento da velocidade das memórias, em especial com o uso dos diversos níveis de cache, o que tem tornado, pelos últimos 20 anos, um tanto discutível o ganho obtido com a redução do número de instruções obtidas por seguir uma arquitetura CISC.

2. RISC - REDUCED INSTRUCTION SET COMPUTER

Com a redução do "custo" da busca de instruções, os arquitetos de processadores trataram de buscar um esquema que otimizasse o desempenho do processador a partir daquelas operações que ele executava mais: as operações simples, como atribuições e comparações; o caminho para isso é simplificar estas operações - o que normalmente leva a CPUs mais simples e menores.

A principal medida escolhida para esta otimização foi a determinação de que leituras e escritas na memória só ocorreriam com instruções de **load** e **store**: instruções de operações lógicas e aritméticas direto na memória não seriam permitidas. Essa medida, além de simplificar a arquitetura interna da CPU, reduz a inter-dependência de algumas instruções e permite que um compilador seja capaz de perceber melhor esta inter-dependência. Como

conseqüência, essa redução da inter-dependência de instruções possibilita que o compilador as reorganize, buscando uma otimização do uso de um possível pipeline.

Adicionalmente, intruções mais simples podem ter sua execução mais facilmente encaixadas em um conjunto de estágios pequeno, como os cinco principais já vistos anteriormente:

1) Busca de Instrução
2) Decodificação
3) Busca de Operando
4) Operação na ULA
5) Escrita de resultado

Às arquiteturas construídas com base nestas linhas é dado o nome de **RISC: Reduced Instruction Set Computer**, pois normalmente o conjunto de instruções destas arquiteturas é menor, estando presentes apenas as de função mais simples.

Os processadores de arquitetura RISC seguem a seguinte filosofia de operação:

1) **Execução de prefetch**, para reduzir ainda mais o impacto da latência do ciclo de busca.
2) **Ausência de instruções complexas**, devendo o programador/compilador construir as ações complexas com diversas instruções simples.
3) **Minimização dos acessos a operandos em memória**, "exigindo" um maior número de registradores de propósito geral.
4) **Projeto baseado em pipeline**, devendo ser otimizado para este tipo de processamento.

Como resultado, algumas características das arquiteturas RISC passam a ser marcantes:

1) **Instruções de tamanho fixo**: uma palavra.
2) **Execução em UM ciclo de clock**: com instruções simples, o pipeline consegue executar uma instrução a cada tick de clock.
3) **As instruções de processamento só operam em registradores**: para usá-las, é sempre necessário usar operações de load previamente e store posteriormente. (Arquitetura LOAD-STORE)
4) **Não há modos de endereçamentos complexos**: nada de registradores de índice e muito menos de segmento. Os cálculos de endereços são realizados por instruções normais.
5) **Grande número de registradores de propósito geral**: para evitar acessos à memória em cálculos intermediários.

3. COMPARAÇÃO DE DESEMPENHO

A forma mais comum de cálculo para comparação de desempenho é a chamada de "cálculo de speedup". O speedup é dado pela seguinte fórmula:

$$S = 100 * (T_S - T_C) / T_C$$

Onde T_S é o "**Tempo Sem Melhoria**" (ou sem otimização) e o T_C é o "**Tempo Com Melhoria**" (ou com otimização). O resultado será uma porcentagem, indicando o quão mais rápido o software ficou devido à melhoria introduzida.

Esta fórmula pode ser expandida, pois é possível calcular os tempos de execução por outra fórmula:

$$T = N_I * C_{PI} * P$$

Tornando claro que N_I é o **Número de Instruções** de um programa, C_{PI} é o número de **Ciclos de clock médio Por Instrução** e P é o **Período**, que é o tempo de cada ciclo de clock (usualmente dado em nanossegundos), sendo o P = 1/freqüência.

Com isso, é possível estimar, teoricamente, o ganho de uma arquitetura RISC frente à uma arquitetura CISC. Suponhamos, por exemplo, um processador com o o Z80 rodando a 3.57MHz, com um C_{PI} de algo em torno de 10 Ciclos Por Instrução e um período de 280ns. Comparemos este com um processador RISC ARM, rodando nos mesmos 3.57MHz (baixíssimo consumo), com 1,25 Ciclos por Instrução (considerando uma perda de desempenho de pipeline em saltos condicionais) e um período de 280ns.

Considerando os programas analisados no item 1, de cópia de trechos de memória, temos que o primeiro (considerado um programa compatível com uma arquitetura RISC), tem um total de:

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
COPIA:      LD    A, (HL)           ; Carrega byte da origem
            INC   HL                ; Aponta próximo byte de  origem em HL
            LD    (DE), A           ; Coloca byte no destino
            INC   DE                ; Aponta próximo byte de destino em HL
            DEC   BC                ; Decrementa 1 de BC
            LD    A,B               ; Coloca valor de B em A
            OR    C                 ; Soma os bits ligados de C em A
            ; Neste ponto A só vai conter zero se B e C eram zero
            JP    NZ,COPIA          ; Continua cópia enquanto A != zero
```

- 3 instruções fixas, mais 8 instruções que se repetirão 500h (1280) vezes, num total de 10243 instruções. O tempo RISC $T_R = N_I * C_{PI} * P = 10243 * 1,25 * 280 = 3585050$ ns, que é aproximadamente 0,003585 segundos.

Já o segundo, compatível com uma arquitetura CISC, tem um total de:

```
LD    BC, 0500h         ; Número de bytes
LD    HL, 01000h        ; Origem
LD    DE, 02000h        ; Destino
; Aqui começa a rotina de cópia
LDIR                    ; Realiza cópia
```

- 3 instruções fixas e 1 que se repetirá 1280 vezes, num total de 1283 instruções. O tempo CISC $T_C = N_I * C_{PI} * P = 1283 * 10 * 280 = 3592400$ ns, que é aproximadamente 0,003592 segundos.

Comparando os dois tempos, $S = 100 * (T_C - T_R) / T_R$, temos que S = 0,21%, que representa praticamente um empate técnico. Entretanto, é importante ressaltar que a CPU RISC que faz este trabalho é significativamente menor, consome significativamente menos energia e esquenta significativamente menos: o que significa que ela custa significativamente MENOS que a CPU CISC. Por outro lado, significa também que é mais simples e barato aumentar a freqüência de processamento da RISC, para o dobro, por exemplo; isso faria com que, possivelmente, seu custo energético e financeiro fossem similares ao da CPU CISC, mas o aumento de desempenho seria sensível.

Recalculando o TR para o dobro do clock (metade do período), ou seja, P = 140ns, $T_R = N_I * C_{PI} * P = 10243 * 1,25 * 140 = 1792525$ ns, que é aproximadamente 0,001793 segundos. Com este valor, o speedup $S = 100 * (T_C - T_R) / T_R$ se torna: S = 100,41%, ou seja, um ganho de mais de 100% ao dobrar a velocidade.

3.1. Comparação Controversa

Apesar do que foi apresentado anteriormente, esta não passa de uma avaliação teórica. Na prática, este ganho pressupõe as considerações de custo e consumo feitas anteriormente e, para piorar, é dependente de qual programa está sendo avaliado.

Existe um consenso de que, em geral, uma arquitetura RISC é mais eficiente que uma arquitetura CISC; entretanto, é reconhecido que isso não é uma regra e, em alguns casos, uma arquitetura CISC pode ser mais eficiente. Isso porque pouco se escreve em linguagem de máquina nos tempos atuais e o desempenho de um software estará, então, ligado à eficiência de otimizaçã do compilador.

Arquiteturas RISC parecem melhores para os compiladores; programas gerados por bons compiladores para processadores RISC deixam muito pouco espaço para otimização manual. Com relação aos processadores CISC, em geral a programação manual acaba sendo mais eficiente.

Talvez também por esta razão, mas certamente pela compatibilidade reversa, a família de processadores mais famosa da atualidade - a família x86, seja uma arquitetura mista. Tanto os processadores da Intel quanto de sua concorrente AMD são processadores com núcleo

RISC mas com uma "camada de tradução" CISC: o programador vê um processador CISC, mas o processador interpreta um código RISC. O processo é representado muito simplificadamente abaixo:

1) Assembly x86 (CISC) Software

---

2) Controle Microprogramado (Tradutor CISC => RISC) CPU
3) Microprograma RISC
4) Controle Microprogramado/Superscalar (Tradutor RISC => sinais elétricos)
5) Core com operações RISC

O que ocorre é que um programa escrito como:

```
LD    BC, 0500h         ; Número de bytes
LD    HL, 01000h        ; Origem
LD    DE, 02000h        ; Destino
; Aqui começa a rotina de cópia
LDIR                    ; Realiza cópia
```

Ao passar da camada 2 para a 3 seria convertido para algo mais parecido com isso:

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
COPIA:      LD    A, (HL)           ; Carrega byte da origem
            INC   HL                ; Aponta próximo byte de  origem em HL
            LD    (DE), A           ; Coloca byte no destino
            INC   DE                ; Aponta próximo byte de destino em HL
            DEC   BC                ; Decrementa 1 de BC
            LD    A,B               ; Coloca valor de B em A
            OR    C                 ; Soma os bits ligados de C em A
            ; Neste ponto A só vai conter zero se B e C eram zero
            JP    NZ,COPIA          ; Continua cópia enquanto A != zero
```

Esta arquitetura permite que prorgramas antigos (que usam uma linguagem de máquina CISC) sejam capazes de ser executados em um processador RISC; a vantagem disto para a Intel e a AMD é que possibilitou um aumento expressivo de freqüência de operação do core (clock) a um custo mais baixo, além de permitir um uso mais eficiente dos pipelines. O único custo é um delay de uma camada de interpretação/tradução mas que, na prática, pode ser descrito, muito simplificadamente, como se fosse mais um nível de pipeline.

4. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.