Notas da Aula 01 e 2: Funcionamento Geral de um Computador
Prof. Daniel Caetano

Objetivo: Revisar o conceito do modelo de Von Neumann, modelo de barramento e níveis das máquinas.

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

Introdução

Como foi apresentado na primeira disciplina, os computadores passaram por uma longa evolução até chegar em seu estágio atual. Da primeira máquina de somar, a Pascalene, criada por Blaise Pascal, passando pelo ENIGMA, ENIAC e tantos outros, até as máquinas de hoje, muita coisa mudou. Entretanto, nenhuma destas mudanças foi mais fundamental do que a capacidade de executar **programas armazenados**.

Pode parecer estranho, mas as primeiras máquinas (estas citadas acima) ou não eram capazes de executar qualquer tarefa diferente daquela para qual foram construídas (da mesma forma que as calculadoras mais simples de hoje em dia) ou, quando eram capazes de alterar sua função, esta alteração era limitada pois consistia em uma mudança do circuito do computador, algo normalmente feito através de placas de circuito. Este tipo de execução de programas era muito limitado, pois não permitia qualquer modificação no programa durante sua execução.

1. O Modelo Von Neumann

O modelo Von Neumann define basicamente os cinco componentes que compõem a quase totalidade dos computadores modernos, além de definir que **o computador deve ser capaz de executar programas armazenados**.

As cinco unidades básicas de um computador segundo o modelo Von Neumann são apresentadas na figura 1 e são melhor descritas a seguir.

1. Unidade de Entrada: dispositivo utilizado pelo usuário (ou pelo próprio computador) para receber informações e programas.
2. Unidade de Memória: local onde os dados de entrada (sejam instruções ou dados puros) são armazenados.
3. Unidade Lógica e Aritmética (ULA): responsável por processar os dados.

4. Unidade de Controle: responsável por controlar o processamento dos dados.
5. Unidade de Saída: dispositivo utilizado pelo computador para apresentar os dados ao usuário.

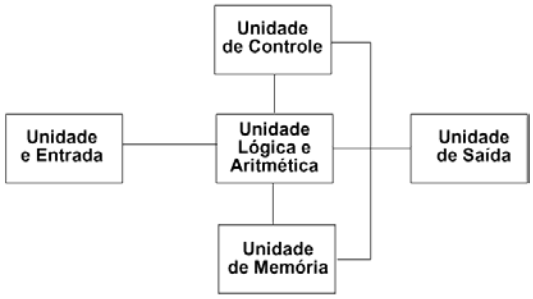


Figura 1: Componentes do Modelo Von Neumann

É freqüente que a ULA e a Unidade de Controle sejam chamadas em conjunto de CPU (Unidade Central de Processamento). Isso ocorre porque, em geral, uma está fortemente ligada à outra, dificilmente estando desassociadas. Um modelo simplificado que muitas vezes usamos é apresentado na figura 2.

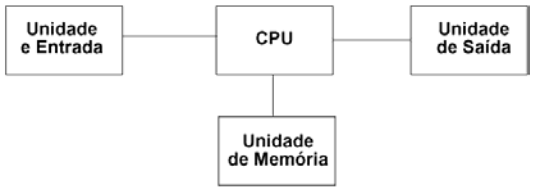


Figura 2: Modelo simplificado baseado no Modelo Von Neumann

Apesar do Modelo Von Neumann definir os componentes de um computador moderno, ele não reflete corretamente a organização arquitetural dos mesmos, já que limita a capacidade de comunicação entre estes componentes (indicada pelas ligações entre as caixas na fig.1 e na fig. 2). Assim, esse modelo foi refinado para o chamado Modelo de Barramento de Sistema, que será visto a seguir.

2. O Modelo de Barramento de Sistema

No Modelo de Barramento de Sistema já é presumida a união da ALU com a Unidade de Controle; além disso, são unificadas as unidades de entrada e saída. Desta forma, 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.

Entretanto, a maior mudança do Modelo Von Neumann para o Modelo de Barramento de Sistem não é a unificação de alguns componentes do modelo Von Neumann, mas sim uma mudança radical na maneira com que estes componentes se comunicam, como pode ser visto na figura 3.

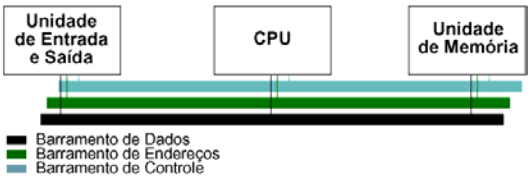


Figura 3: Modelo de Barramento de Sistema

Observe pela figura 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).

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 - usado para transmitir dados.
2. Barramento de Endereços - usado para identificar onde o dado deve ser lido ou escito.
3. Barramento de Controle - usado 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).

Por exemplo, 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 que deseja ler.
4. Memória lê barramento de endereços e devolve o dado solicitado 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.

3. Níveis das Máquinas

Como todo sistema complexo, um computador pode ser visto por várias perspectivas. Podemos dizer que existem vários níveis de abstração em um computador e uma das grandes vantagens dos computadores modernos é a enorme independência entre estes níveis de abstração.

Na prática, é isso que permite que o usuário de um software qualquer não precise conhecer programação e que um programador não precise entender de eletrônica e portas lógicas.

Uma outra grande vantagem disso é a possibilidade de desenolver sistemas compatíveis "para cima" ou *upward* compatíveis. Essa vantagem pode ser vista na prática: imagine o quanto mudaram os circuitos de um i8088 até os processadores mais atuais da família Intel. Certamente a forma de operar dos circuitos internos mudou, o que faria com que nenhum software pudesse mais ser executado numa mudança de processador. Entretanto, não é isso que ocorre, não é?

Devido à forma como os níveis de operação (e abstração) dos computadores estão organizados, existe uma hierarquia de níveis que permitem que mesmo alterando radicalmente o circuito básico de um processador, ele continue compatível com seus anteriores *quando se fala em instruções de linguagem de máquina.*

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

Estes níveis estão melhor detalhados a seguir.

3.1. Programas Aplicativos

Este é o nível com que, obviamente, o usuário de computador está mais familiarizado. É o nível com 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.

3.2. Linguagens de Alto Nível

Este 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.

3.3. 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.

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 foi, 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 diretamente 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*).

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.

3.4. Controle Microprogramado

Este 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.

3.5 Unidades Funcionais

A grande maioria das operações da Unidade de Controle 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 e a memória principal.

3.5 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).

4. Emuladores

Algumas arquiteturas possuem "instruções opcionais". Um grande exemplo disso são as antigas CPUs, que permitiam a existência de um co-processador matemático, para operações de ponto flutuante.

Neste caso, quando é chamada uma instrução de operação de ponto flutuante num computador sem o co-processador matemático, é disparada uma trap (armadilha) que permite que aquela operação seja implementada por software, ou seja, **emulada** por software. Para o programador tudo funciona como se a instrução de linguagem de máquina existisse, com uma única diferença na velocidade de execução. Aos programas que substituem a execução de instruções de máquina damos o nome de **emuladores**.

Hoje existem máquinas tão rápidas que tornou-se viável a emulação de computadores inteiros. Com o uso de emuladores é possível, hoje, executar praticamente programa de qualquer equipamento dentro de um PC comum, compatível com x86. A Apple também faz uso de emuladores, para manter compatibilidade com programas antigos na transição do processador M68000 dos MacIntoshes para o IBM/Motorola PowerPC. O mesmo está ocorrendo agora, na transição do IBM/Motorola Power PC para o Intel x86.

5. Bibliografia

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

Notas da Aula 03: Unidade Lógica Aritmética
Prof. Daniel Caetano

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

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

Nesta aula iniciaremos o estudo do funcionamento e operação de uma Unidade Central de Processamento (CPU), através do estudo 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. Os Registradores

1.1. Uma Analogia - O Computador e a Empresa

É possível pensar no processador como um conjuntoi de funcionários praticamente sem memória alguma. É como se dois funcionários, chamados ULA e Controle, estivesse 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 funcionário 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 bem que todas as empresas concorrentes copiaram, embora muitas vezes deem nomes diferentes para os *registradores*.

1.2. 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:

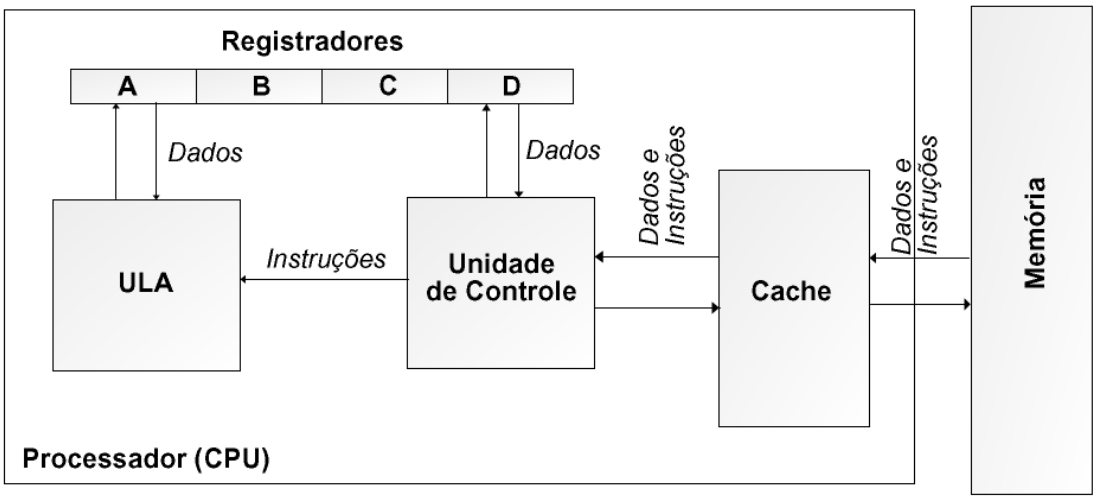


Figura 1: Interligações da ULA com Unidade de Controle e Registradores

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.

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. Lista de Exercícios L1

1) Qual foi a grande mudança introduzida com o modelo de Von Neumann?

2) Porque o modelo de Von Neumann teve de ser modificado? Qual a vantagem do modelo de barramentos de sistema? Faça um esquema de ambos.

3) A implementação de barramentos de sistema inclui um barramento de controle. Qual a função dele? O que aconteceria se ele não existisse?

4) Quais são os níveis de máquina para os quais se fala em compatibilidade? Qual é o nível de compatibilidade relacionado a cada nível de máquina?

5) O que é emulação? Para que ela serve?

6) O que são e para que servem os registradores de um processador?

7) O que é e para que serve a Unidade Lógica Aritmética?

8) Uma CPU funcionaria sem uma ULA? A máquina construída sem uma ULA seria uma máquina de Von Neumann?

9) Considerando o contexto da ULA, qual é o papel da Unidade de Controle?

10) Uma instrução do tipo ADD A, DADO_APONTADO_POR ENDEREÇO poderia ser executada diretamente pela ULA? Se sim, como isso seria feito? Se não, por quê?

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.

Notas da Aula 04: 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.

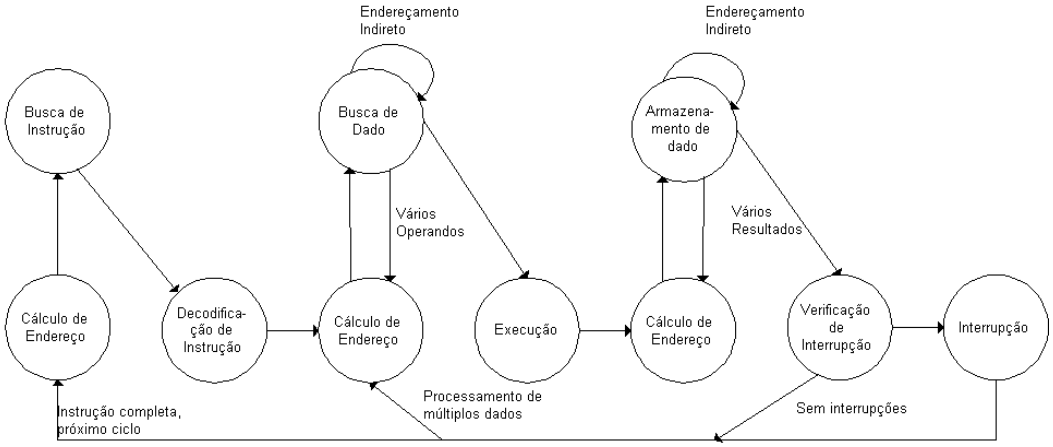


Figura 1 - Diagram de transição de estados do ciclo de instrução (STALLINGS, 2003)

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 Duo e Core 2 Duo.

4. Lista de Exercícios L1 - Adendo 1

1) Qual é a função da Unidade de Controle, em linhas gerais?
2) Para que serve o registrador PC?
3) O que acontece no ciclo de busca de dados?
4) O que é o "pre-fetch"?
5) Quala razão para o uso de pipeline? Qual a limitação desta técnica?

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.

Notas das Aulas 05: Identificação de Componentes
Prof. Daniel Caetano

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

Bibliografia:

- LE MUSÉE de l'informatique: Silicium, 15/09/2004. Disponível em: < http://www.silicium.org/ > Acesso em: 30/08/2007.
- CLUBE old bits: por dentro do TK 85, 09/09/2003. Disponível em < http://cobit.mma.com.br/ > Acesso em: 30/08/2007.
- JACK Baskin School of Engineering, 24/01/2002. Disponível em < http://www.soe.ucsc.edu/ > Acesso em: 30/08/2007.
- RED Hill Motherboards, 21/07/2005. Disponível em < http://redhill.net.au/ > Acesso em: 31/08/2007.
- ASUSTEK computer inc., 31/08/2007. Disponível em < http://br.asus.com > Acesso em: 31/08/2007.

Introdução

Nas aulas anteriores foram apresentados os componentes internos de um computador do ponto de vista teórico, além de uma avaliação da função de cada componente dentro de um computador.

Antes de partir para uma análise mais apurada de funcionamento do equipamento, através do estudo de uma linguagem de máquina, é interessante observar os componentes que constituem um computador em sua forma física.

Serão apresentados alguns equipamentos mais antigos e alguns mais atuais. Como será visto, os componentes existentes são essencialmente os mesmos: as mudanças estão basicamente no nível de integração, funcionamento interno dos componentes e nas interconexões.

1. Identificação de Componentes em Computadores Antigos

Apesar de não serem mais utilizados, os computadores de cerca 20 a 30 anos atrás são bastante interessantes para estudo, do ponto de vista didático, por apresentarem uma estrutura de barramento bastante simplificada que tornam a identificação dos componentes e de suas interconexões.

TK-85

Dentro deste espírito, o primeiro equipamento apresentado será o TK-85, da antiga empresa nacional Microdigital. Este equipamento era um clone de um Sinclair Spectrum, computador idealizado por Clive Sinclair, um "visionário" inglês.

Este computador é baseado no microprocessador Z80 e não tem uma unidade de vídeo dedicada. Além disso, sua memória de 48KB é dividida entre área de display de vídeo e memória principal. Por esta razão, há apenas um banco de memória. Não há também um chip de som dedicado na placa principal - que devia ser conectado como uma placa de som. O teclado e demais dispositivos eram controlados com lógica simples, não possuindo um circuito integrado especial para isso. Essas características tornam a identificação dos poucos componentes integrados muito simples, como pode ser visto na figura 1.

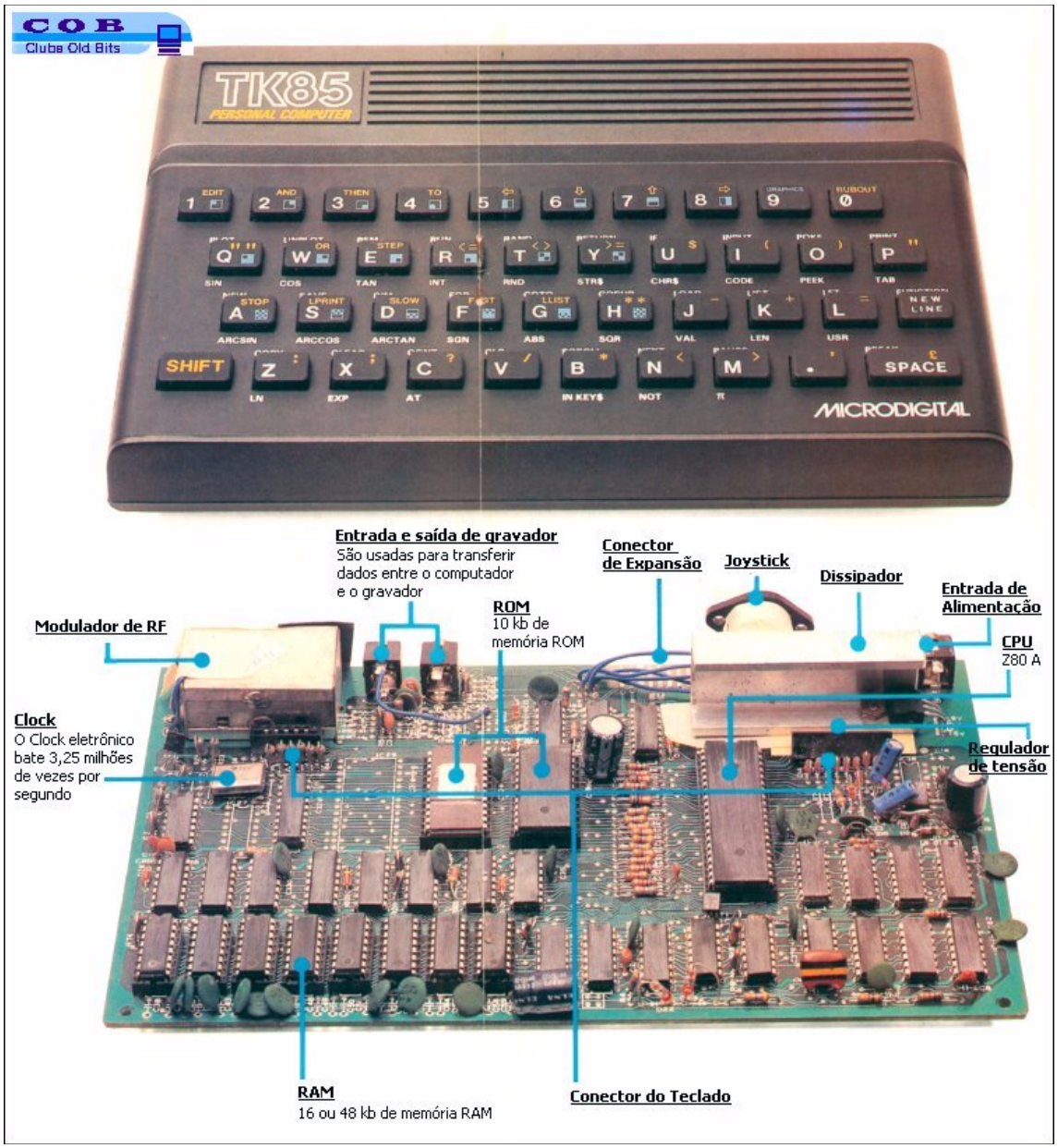


Figura 1: Microgital TK-85 - Visão externa e interna (CLUBE, 2003)

A figura indica, além da CPU e dos 8 chips de memória RAM, mostra também o gerador de clock (quartzo) de 3,25MHz, memória ROM ("sistema operacional" e linguagem BASIC) e alguns elementos do circuito e portas de comunicação.

MSX HB-8000 - HotBit

O HotBit foi a versão da Sharp brasileira de uma enorme linha de computadores compatíveis sob a especificação MSX, criada em 1983. A Gradiente também teve o seu computador compatível com a especificação MSX, chamado XP800 Expert.

A linhas MSX também era baseada em Z80 (a 3.57MHz), mas já era bem mais complexa que a linha Spectrum e, por esta razão, muitos outros circuitos já surgem, como um processador de vídeo dedicado (com memória própria), um processador de som dedicado e um controlador de dispositivos e bancos de memória. Os micros nacionais da linha MSX já vinham com 64KB de RAM de fábrica, mais 16KB de RAM de vídeo (comumente chamada de VRAM), fora os 32KB de ROM com o "sistema operacional" e o BASIC. A figura 2 mostra parte destes elementos, embora nem todos sejam facilmente visíveis:

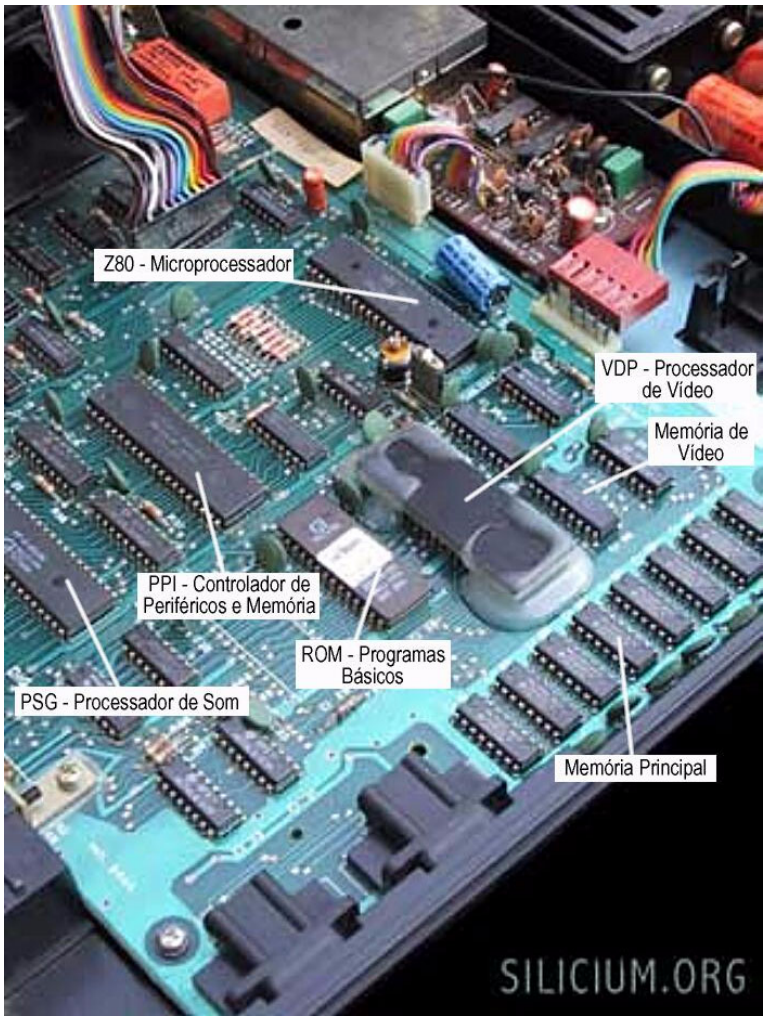


Figura 2: Sharp HB-8000 - Visão interna (LE MUSÉE, 2004)

Para quem teve a curiosidade, a aparência externa de um MSX HotBit é a apresentada na figura 3.

Figura 3: Sharp HB-8000 - Aparência externa (LE MUSÉE, 2004)

IBM PC 386

Do MSX damos um salto no tempo para o início da década de 90, onde as coisas começaram a ficar complexas. O computador em questão é um dos muitos clones da linha i80386, da Intel.

O PC 386 comum já era complexo o suficiente e muito do que vinha nas placas principais dos computadores antigos passou a vir em uma placa separada, como controlador de impressora e joystick, controlador de vídeo, controlador de áudio... até mesmo a memória passou a ser em pequenos "pentes". Adicionalmente, era comum as placas de 386 apresentarem um soquete para a adição de um co-processador matemático e a maior parte da circuitaria "glue-logic" foi concentrada em dois circuitos principais: o chamado North Bridge, que controla o acesso às memórias e o South Bridge, que controla o acesso aos slots e periféricos. Isso possibilitou, no futuro, que o "clock" de cada uma destas partes (CPU, slots e memórias) fossem distintos (como ocorre nos computadores atuais: Slots PCI a 33MHz, memória a 200MHz ~ 400MHz e CPU a 2GHz~3.5GHz). Adicionalmente, surgiu a já discutida memória cache, que permitia que os processadores de alto desempenho fossem melhor aproveitados, mesmo que com memórias bastante mais lentas que o processador central.

A figura 4 mostra alguns dos principais elementos de uma placa mãe de um IBM PC 386.

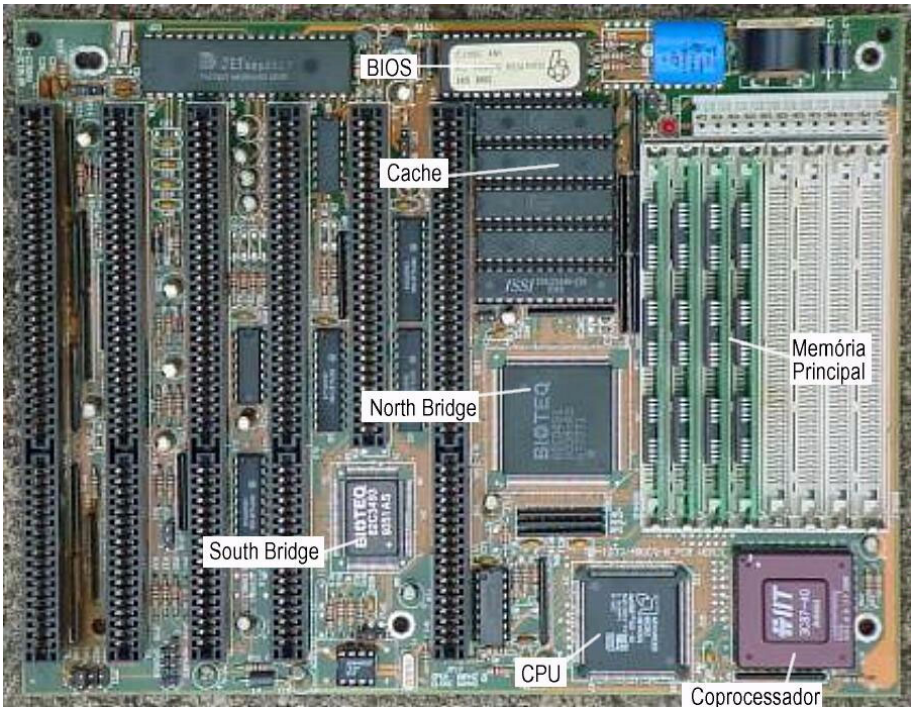


Figura 4: IBM PC 386-SX (AMD) - Visão interna (RED, 2005)

Repare, em específico, no cache externo. Note, também, a bateria azul, à direita da BIOS, acima da memória principal. Esta bateria passou a ser comum, para manter algumas configurações como data e hora, harddisks instalados, etc.

Observe, adicionalmente, que praticamente nenhum periférico é controlado diretamente por esta placa, além do teclado (conector no canto superior esquerdo). Todos os dispositivos são controlados através de placas externas. Repare que próximo a cada circuito integrado existe um pequeno capacitor: estes capacitores existem para fornecer o pico de tensão necessário aos circuitos no momento em que eles são solicitados. Se não houver estes capacitores, o circuito integrado simplesmente não funciona. corretamente.

3. Identificação de Componentes em Computadores Recentes

Nos equipamentos recentes, houve uma grande integração de componentes e quase tudo está dentro de um único circuito integrado, inclusive alguns periféricos como controlador de rede e som (ex. chipset nForce4, da nVidia).

A título de exemplo, a figura 5 apresenta uma motherboard atual, com seus elementos principais nomeados.

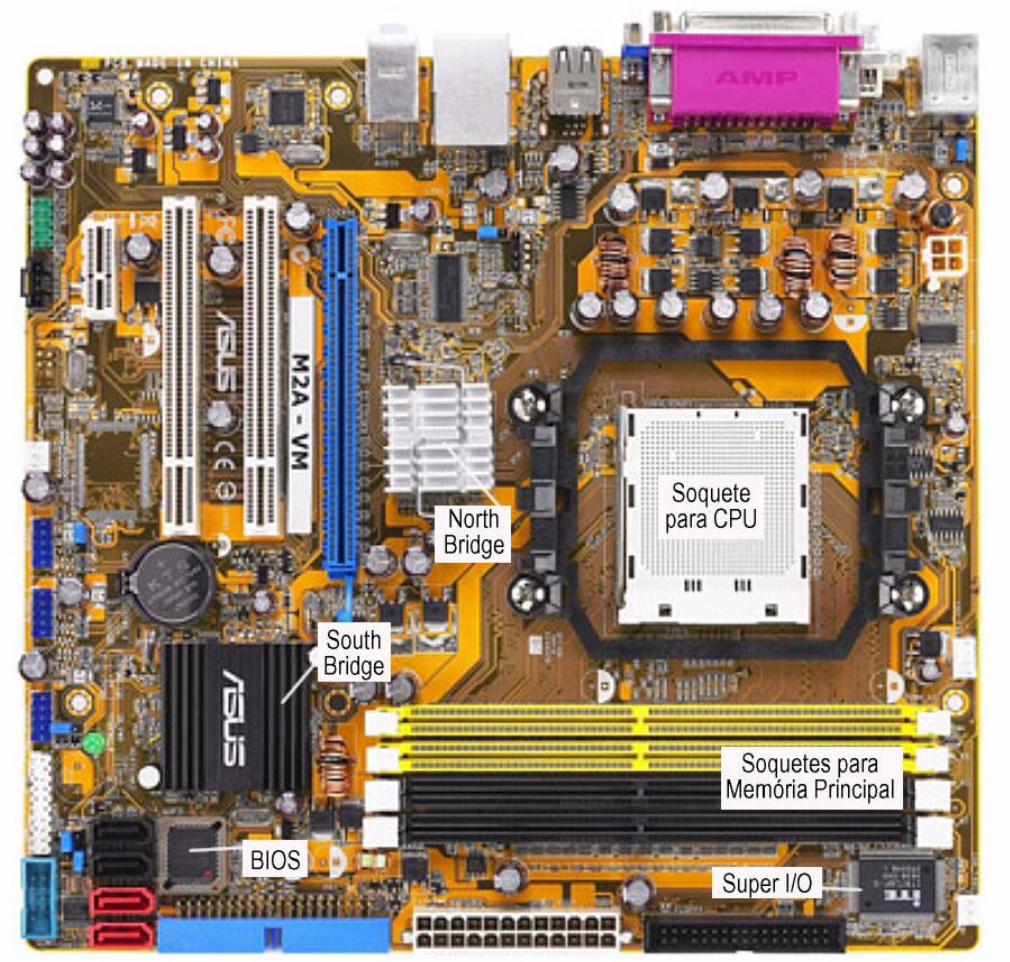


Figura 5: IBM PC - Placa ASUS M2A-VM para AMD M2 - Visão interna (ASUSTEK, 2007)

Observe a diferença da placa do 386 para essa: muitos conectores. Todo o controle dos conectores costuma estar no mesmo circuito integrado south bridge. Agora existem dissipadores sobre o north e south bridge, uma vez que eles concentram muito mais atividades e operam a velocidades muito altas. A BIOS deixou de ser um circuito integrado CMOS enorme e passou a ser muito menor, mas com mais capacidade.

O circuito da Super I/O (antiga placa que controlava HDs, discos, impressora etc) agora é um pequeno circuito integrado. Os soquetes de cores diferentes serve para que se use o chamado "Dual Channel", que é uma maneira que o North Bridge tem, hoje, de quase duplicar a taxa de transferência da CPU para a RAM e da RAM para a CPU.

Não se usa mais a bateria de Ni-Cd azul, o mais comum nos dias de hoje é uma bateria de lítio (redonda, acima do south bridge). Ao redor da CPU há o suporte para o dissipador de calor (heat sink) e o cooler (ventilador), já que as CPUs atuais não conseguem trabalhar sem dissipar uma quantidade excessiva de calor.

As CPUs mais novas demandam bastante corrente em algumas situação, causando picos. Os picos de corrente tendem a causar uma queda de tensão e, por essa razão, nas plascas mãe atuais é obrigatória a presença de reguladores de tensão parrudos (capacitores e

toróides acima do soquete da CPU). Em geral, quando uma placa mãe dá problemas (não liga mais, reinicia sem razão aparente, etc), é algum dos capacitores do regulador de tensão que se estragou.

4. Bibliografia

LE MUSÉE de l'informatique: Silicium, 15/09/2004. Disponível em: < http://www.silicium.org/ > Acesso em: 30/08/2007.

CLUBE old bits: por dentro do TK 85, 09/09/2003. Disponível em < http://cobit.mma.com.br/ > Acesso em: 30/08/2007.

JACK Baskin School of Engineering, 24/01/2002. Disponível em < http://www.soe.ucsc.edu/ > Acesso em: 30/08/2007.

RED Hill Motherboards, 21/07/2005. Disponível em < http://redhill.net.au/ > Acesso em: 31/08/2007.

ASUSTEK computer inc., 31/08/2007. Disponível em < http://br.asus.com > Acesso em: 31/08/2007.

Notas das Aulas 06: Linguagem Assembly
Prof. Daniel Caetano

Objetivo: Apresentar uma breve visão sobre uma linguagem assembly e como programas são construídos nesta linguagem.

Bibliografia:

- ROSSINI, F; LUZ, H.F. **Linguagem de Máquina**: Assembly Z80 - MSX. 1ed. São Paulo: Ed. Aleph, 1987.
- CARVALHO, J.M. **Assembler para o MSX**. 1ed. São Paulo: McGraw Hill, 1987.
- ASCII Corporation. **O Livro Vermelho do MSX**. 1ed. São Paulo: Ed. McGraw Hill, 1988.

Introdução

Nas aulas anteriores foram apresentados os elementos internos dos computadores, tendo sido o funcionamento interno dos principais deles descritos previamente. Todos estes conceitos são importantes para assimilar as limitações e potencialidades de cada equipamento que um programador tome contato; e é importante ressaltar que, diferentemente do que se pode pensar a princípio, hoje é dia é bastante comum a necessidade de programação para diferentes plataformas como videogames, celulares, PDAs, mídia players, computadores etc.

Como mais uma ferramenta para conhecer as limitações e potencialidades de uma máquina e também como uma forma de compreender melhor os conceitos previamente apresentados, nesta aula veremos uma breve introdução à linguagem de máquina.

1. Por que Usaremos Assembly do Z80?

O primeiro aspecto importante é que esta aula trata do assembly de um processador bastante antigo, do fim da década de 70. Mas porque iremos estudar isso? Por algumas razões, dentre elas:

- O Assembly do Z80 é, reconhecidamente, um dos mais amigáveis.
- O Z80 é o processador de maior sucesso da história, embora poucos falem nele hoje.
- O Z80 é usado ainda hoje, com a função de microcontrolador.
- O Z80 tem um número interessante de registradores de propósito geral.

Estas características possibilitam que até mesmo alguém que nunca programou aprenda a linguagem Assembly Z80, embora possa ser uma taréfa mais árdua para aqueles que jamais programaram, mas isso não tem a ver com a linguagem em si: assembly é, em essência, a linguagem mais fácil (e simples) que existe. Ao mesmo tempo, as inerentes limitações da linguagem assembly permitem, como nenhuma outra linguagem, o desenvolvimento da habilidade dos programadores em "dar soluções geniais" para problemas gerais.

2. Primeiros Passos no Assembly Z80

Primeiramente, vamos esquecer que o Assembly é uma linguagem de programação: pensemos que é uma seqüência de ordens que devem ser transmitidas às peças do computador. Pensando desta forma, tudo começa a fazer muito mais sentido, pois é isso que de fato ocorre quando programamos Assembly: damos ordens aos microchips.

Assembly também não é a linguagem de máquina (zeros e uns): Assembly é uma linguagem compreensível pelos seres humanos, ainda que ela represente exatamente as instruções da máquina. Por esta razão, é necessário o uso de um programa "Assembler", que converta o texto que escrevemos em código de máquina (em linguagem de máquina) para que o computador execute o programa. Isso será visto passo a passo.

2.1. Primeiro Software em Assembly Z80

Antes de mais nada, é preciso definir o objetivo. O objetivo desta introdução será usar o Assembly Z80 para escrever o famoso programa "Alo, Mundo". Então, claramente, a idéia é fazer um programa que escreva:

"Alo, Mundo"

Na tela do computador... mas isso é muito vago. Z80 é só um processador e ele precisa estar inserido em um equipamento inteiro para que essa tarefa possa ser realizada. Neste caso, será então selecionado um computador genérico que use Z80 (como um MSX), executando um sistema operacional compatível com CP/M (como o MSX-DOS). É claro que não usaremos a máquina real; usaremos um emulador, no caso o Z80Mu. Adicionalmente, precisaremos de um "assembler", que é um programa que converterá o que escrevemos em Assembly para a Linguagem de Máquina. Neste curso usaremos o M80/L80 da Microsoft.

Voltando ao problema, que já é bem mais definido: A idéia será escrever "Alo, Mundo" usando o CP/M, em um (emulador de) computador Z80. Isso é importante, porque, como já vimos, a linguagem Assembly muda de máquina para máquina.

Mas vamos por partes... se queremos escrever algo na tela do computador, lembrando que o computador é uma máquina de Von Neuman, a primeira coisa a se fazer é colocar a informação a ser impressa na **memória**. A memória dos computadors com Z80 tem 64KB, ou seja, 65536 *posições de memória*. Isso significa que você tem essas 65536 posições para colocar todos os dados e instruções do seu programa. Pode parecer pouco (de fato, a maioria dos documentos de Word tem pelo menos o dobro disso de tamanho), mas como vocês verão, em Assembly é muito difícil gastar tudo isso de memória.

Antes de vermos como solicitar que o computador imprima a frase na tela, vejamos como colocar **o que** queremos que ele escreva na memória. Assim, o primeiro passo será abrir o "NotePad" e digitar, na primeira linha, o seguinte:

```
Alo, Mundo
```

Ótimo, só que o computador não vai entender isso e irá simplesmente travar se tiver de executar esse "código". Para entender o porque disso, imagine um japonês tentando falar em russo com você, e você não sabe nem japonês nem russo. É mais ou menos assim que o computador vai encarar se isso for colocado na memória dele, deste jeito. Precisamos então indicar para o computador o que deve ser feito com esses dados, em uma linguagem que ele entenda.

Para compreender esta necessidade, não é preciso se colocar na posição do computador: isso que foi escrito no NotePad não tem significado sequer para uma pessoa: se alguém recebesse um papel com isso escrito, a pessoa simplesmente não ia entender o que significa ou para que serve. Com o computador, não seria diferente. Para isso fazer algum sentido para uma pessoa, o texto deveria ser algo do tipo:

```
"Pegue a frase abaixo:
Alo, Mundo
e cole na tela do monitor"
```

É claro que uma pessoa seria capaz de seguir estas ordens (embora ainda possa achá-las esquisitas). É importante lembrar que o computador sempre agirá exatamente desta forma: ele fará o que o programador mandar, desde que o programador mande direito. (pode-se dizer que ele faz exatamente o que o programador manda, o que não é necessariamente o que o programador quer...)

De qualquer forma, existe uma tarefa aí nesta seqüência de instruções que é complexa: "cole na tela do monitor"... como se faz isso? Com CTRL+V, com cola branca ou com post-it? Bem, dissemos anteriormente que iríamos trabalhar com o CP/M e, por sorte, o CP/M sabe exatamente como "colar um texto na tela do monitor". A instrução do CP/M que faz isso é a função número **9**. Embora pareça estranho no início, com o tempo o programador assembly se acostuma com "nomes" numéricos para tudo.

A função número 9 faz exatamente isso: "Pegue *a frase X* e a cole na tela do monitor". Ocorre que "a frase X" é algo que estará na memória do computador, como vimos anteriormente. E se "a frase X" está na memória do computador, ela deve ter um **endereço de memória**, que no caso das strings é (quase) sempre o endereço do primeiro caractere da string. Assim, se dizemos que a frase "Alo, Mundo" está no endereço 10000h, o que teremos na memória do computador é:

Posição de Mem.	Conteúdo
10000	A
10001	l
10002	o
10003	,
10004	
10005	M
10006	u
10007	n
10008	d
10009	o

A função 9 do CP/M exige que este valor inicial da frase esteja indicada por um registrador de uso geral chamado **DE**. Assim, se soubéssemos que a frase estava realmente no endereço 10000h, bastaria indicar este valor no registrador DE e em seguida chamar a função 9 do CP/M que a frase seria impressa na tela. Então, a primeira coisa que precisa ser feita após o texto "Alo, Munto", é indicar no registrador DE a posição de memória em que a frase estará... mas aí vem o primeiro problema: qual é esta posição?

Poderíamos definir um valor fixo; entretanto, não iremos fazê-lo. No início desta aula foi citado que usaríamos um programa "Assembler" para converter o código Assembly para Linguagem de Máquina. Existe uma maneira de deixar que o programa Assembler se preocupe com os endereços de memória por nós: usando apelidos.

Quando um apelido é definido para um dado, não importa onde o Assembler colocará aquele dado: podemos nos referir àquela posição de memória através do apelido. A indicação do apelido se faz usando a estrutura:

APELIDO: Dados

Por se tratar de uma string, é necessário indicar ao Assembler onde a string começa e onde ela termina, fazendo isso com as aspas simples: ' e ':

APELIDO: 'Uma String'

Por outro lado, também é necessário informar ao Assembler qual é o tipo de dado que está recebendo o apelido, que no caso é uma seqüência de bytes, que é indicado desta forma:

APELIDO: DB 'Uma String'

Onde DB significa "Data Bytes" ou "Bytes de Dados".

No nosso exemplo, vamos usar o apelido "FRASE" para a frase a ser impressa, e a indicação no programa fica assim:

```
FRASE:   DB   'Alo, Mundo'
```

Isso nos fornece o apelido "FRASE" para trabalhar. Se precisamos indicar o endereço da frase no registrador DE, basta carregar (ou ler) este valor no registrador DE. Isso é feito com a instrução LD (de LoaD) do Z80, cuja sintaxe é:

LD DE, *dado*

Onde *dado* é um número qualquer. No caso, deve ser indicada a posição de memória (ou seu apelido) da frase, para que ela possa ser impressa. Esta instrução precisa ser acrescentada antes de solicitarmos ao CP/M que imprima a frase. Isso pode ser feito como apresentado abaixo:

```
FRASE:   DB   'Alo, Mundo'
         LD   DE, FRASE
```

Assim, a função 9 já saberá onde encontrar a frase a ser impressa. A necessidade, agora, é informar ao CP/M que ele deve executar a função 9. Para isso, o CP/M solicita que o número da função a ser executada seja indicada no registrador de uso geral **C**. O formato é o mesmo já visto anteriormente:

LD C, *dado*

Desta forma, para indicar a função 9 para o CP/M, o seguinte deve ser escrito:

```
FRASE:   DB   'Alo, Mundo'
         LD   DE, FRASE
         LD   C, 9
```

Agora todo o cenário está preparado para que o CP/M execute a função 9 e o texto apareça escrito na tela, mas precisamos "solicitar" que ele execute a função selecionada no registrador C e isso pode ser feito com uma função que está na posição 5 da memória (ela foi carregada lá quando o CP/M foi iniciado). Como queremos que a execução do programa se desloque para lá temporariamente, faça o que tem de fazer e depois volte, usamos a instrução CALL do Assembly:

CALL [endereço]

Com essa chamada, o programa irá até a posição de memória indicada, executará o código que lá encontrar... até encontrar a instrução RET (RETurn), quando então volta para continuar a execução logo após o CALL.

O código, já implementando a instrução CALL para o endereço 5 está a seguir:

```
FRASE:   DB   'Alo, Mundo'
         LD   DE, FRASE
         LD   C, 9
         CALL 5
```

Ok, parece que isso vai funcionar: primeiro definimos os dados, depois indicamos os parâmetros nos registradores e finalmente chamamos a execução da função de impressão que o CP/M fornece... mas este programa está muito pouco legível. É possível melhorá-lo.

O Assembler que iremos usar, assim como a maioria dos Assemblers, permite que o programador dê *apelidos* a valores numéricos, usando uma *pseudo-instrução* chamada EQU (de EQUivalente). A forma desta instrução é:

APELIDO EQU VALOR

Assim, se quisermos dar um apelido interessante para a função 9 do CP/M, como por exemplo STROUT (de STRing OUT - "saída de texto", em português) podemos fazer da seguinte forma:

STROUT EQU 9

É possível fazer o mesmo com o endereço 5 de memória, dando um apelido BCPM (Bios do CP/M) a ele, como pode ser visto no código abaixo:

```
STROUT   EQU  9
BCPM     EQU  5

FRASE:   DB   'Alo, Mundo'

         LD   DE, FRASE
         LD   C, STROUT
         CALL BCPM
```

É possível tornar o código ainda mais legível com alguns comentários, que são indicados pelo caractere ";":

```
STROUT   EQU  9
BCPM     EQU  5

FRASE:   DB   'Alo, Mundo'

         LD   DE, FRASE      ; Indica endereço do texto
         LD   C, STROUT      ; Indica função do CPM
         CALL BCPM           ; Solicita execução pelo CPM
```

Este código já está com uma cara de programa, mas ainda não vai funcionar... e por quê? Porque começamos o nosso programa com uma seqüência de dados que nada têm a ver com um programa... e o Z80 (como qualquer outro processador) vai pensar que estes dados são, na verdade, instruções.

Isso nos remente ao modelo de Von Neuman: para o processador, não existe diferença física entre dados e instruções: ambos são números na memória. O que diferencia entre um e outro é a permissão que o programador dá para que a CPU execute um trecho da memória ou não. Por exemplo: se comandarmos no software a instrução:

```
CALL FRASE
```

Teremos dito para o Z80 "execute o que está na posição de memória indicada pelo apelido FRASE". Mas isso será um desastre, porque "FRASE" não aponta para um programa! "FRASE" aponta para um texto! Este tipo de coisa (CPU processando dados como instruções) normalmente faz com que o computador trave, simplesmente.

Mas a correção disso é simples: como não faz a menor diferença onde a definição da frase foi colocada (uma vez que o apelido será corrigido em qualquer posição que você a coloque), é possível colocá-la no fim do programa, e aí o problema acaba... ou quase:

```
STROUT   EQU  9
BCPM     EQU  5

         LD   DE, FRASE      ; Indica endereço do texto
         LD   C, STROUT      ; Indica função do CPM
         CALL BCPM           ; Solicita execução pelo CPM

FRASE:   DB   'Alo, Mundo'
```

Mas... por que quase? Observe a execução do programa: o que vai acontecer quando a execução voltar do "CALL BCPM"? Isso mesmo! O Z80 vai voltar a executar a frase 'Alo, Mundo', só que desta vez após a impressão da frase na tela. Para evitar isso, é preciso indicar para o Z80 onde o programa acaba... na verdade, iremos chamar um outro comando do CP/M que indica para que o programa seja finalizado e voltemos ao prompt. Isso pode ser feito com a instrução JP (de JumP, salto) que é o "pulo para nunca mais voltar", indicando para que o programa vá para o endereço 0:

```
JP      0
```

Inserindo esta instrução ao fim do programa, tudo estará pronto...:

```
STROUT   EQU  9
BCPM     EQU  5

         LD   DE, FRASE      ; Indica endereço do texto
         LD   C, STROUT      ; Indica função do CP/M
         CALL BCPM           ; Solicita execução pelo CP/M
         JP   0              ; Volta ao prompt do CP/M

FRASE:   DB   'Alo, Mundo'
```

Salve este arquivo na sua pasta de aluno, dentro de um diretório chamado \ASM, com o nome HELLO.MAC . Agora falta usar o Assembler para executar nosso programa.

2.2. Assemblando o Software HELLO.MAC

No computador já deve estar instalado o software M80/L80 da Microsoft. Para executá-lo, abra o prompt, mude para o diretório onde criou o arquivo HELLO.MAC e comande:

CL80 HELLO

A saída deverá ser algo do tipo:

```
M80/L80 Z80 Compiler - IBM PC
Ported by A&L Software

MSX.M-80  1.00  01-Apr-85  (c) 1981,1985 Microsoft
%No END statement
%No END statement

No Fatal error(s)

MSX.L-80  1.00  01-Apr-85  (c) 1981,1985 Microsoft

Data    0103    0118    <   21>

49189 Bytes Free
[ 0000   0118        1]
```

Isso terá gerado o arquivo HELLO.COM, que já é o executável pronto!

2.3. Executando o Software HELLO.COM

Agora que já temos o executável pronto, já no diretório em que reside o arquivo HELLO.COM, digite:

Z80MU

Isso fará com que entremos no prompt do CP/M. Neste prompt (roxo) basta digitar:

HELLO

E o programa criado será executado... só que o resultado não é exatamente o esperado! Após aparecer "Alo, Mundo" muito rapidamente, um monte de lixo é impresso e só um vidente pode dizer o que acontece em seguida.

Isso não devia estar acontecendo, afinal, tomamos todas as precauções para que o programa finalizasse corretamente, certo? Correto, mas faltou uma informação... e a falta dela está causando todo este transtorno.

A culpa não é sua, aluno. Faltou dizer que a função STROUT imprime uma string terminada pelo símbolo "$". Sem isso, a função STROUT vai imprimindo tudo que encontrar na memória, até achar um caractere desse (acredite, ela **não para** até achar!). Assim, saia do Z80Mu com o comando "QUIT" e experimente alterar o seu programa da seguinte forma:

```
STROUT   EQU  9
BCPM     EQU  5

         LD   DE, FRASE      ; Indica endereço do texto
         LD   C, STROUT      ; Indica função do CP/M
         CALL BCPM           ; Solicita execução pelo CP/M
         JP   0              ; Volta ao prompt do CP/M

FRASE:   DB   'Alo, Mundo$'
```

"Reassemble" o programa como indicado na seção 2.2 e re-execute o mesmo pelo emulador, como indicado na seção 2.3. Finalmente, o programa irá funcionar corretamente, imprimindo:

Alo, Mundo

E finalizando, voltando ao prompt do CP/M. Voilà! Seu programa funcionou perfeitamente! Esse é o seu primeiro programa em Assembly. Antes de finalizar, porém, é interessante comentar o seguinte. Você deve ter observado que, quando usou o CL80, os seguintes comentários apareceram:

MSX.M-80 1.00 01-Apr-85 (c) 1981,1985 Microsoft
%No END statement
%No END statement

Isso ocorre sempre que não indicamos ao M80 onde o programa inicia e onde ele termina. Para corrigir isso, basta acrescentar as *pseudo-instruções* START e END no programa, da seguinte forma:

```
STROUT   EQU  9
BCPM     EQU  5

START:
         LD   DE, FRASE      ; Indica endereço do texto
         LD   C, STROUT      ; Indica função do CP/M
         CALL BCPM           ; Solicita execução pelo CP/M
         JP   0              ; Volta ao prompt do CP/M

FRASE:   DB   'Alo, Mundo$'

         END
```

Se assemblá-lo novamente, verá que o M80 não reclama de mais nada em seu programa.

3. Bibliografia

ROSSINI, F; LUZ, H.F. **Linguagem de Máquina**: Assembly Z80 - MSX. 1ed. São Paulo: Ed. Aleph, 1987.

CARVALHO, J.M. **Assembler para o MSX**. 1ed. São Paulo: McGraw Hill, 1987.

ASCII Corporation. **O Livro Vermelho do MSX**. 1ed. São Paulo: Ed. McGraw Hill, 1988.

Notas das Aulas 07: Linguagem Assembly - Parte II
Prof. Daniel Caetano

Objetivo: Aprofundar os conceitos de programação em linguagem assembly.

Bibliografia:

- ROSSINI, F; LUZ, H.F. **Linguagem de Máquina**: Assembly Z80 - MSX. 1ed. São Paulo: Ed. Aleph, 1987.
- CARVALHO, J.M. **Assembler para o MSX**. 1ed. São Paulo: McGraw Hill, 1987.
- ASCII Corporation. **O Livro Vermelho do MSX**. 1ed. São Paulo: Ed. McGraw Hill, 1988.

Introdução

Na aula anterior foi apresentada a idéia de se escrever um programa, ou seja, uma seqüência de ordens para o computador, que escrevesse na tela a frase "Alo, Mundo". Bem, para fazer isso, foi usado o NotePad (Bloco de Notas) como editor e o Microsoft Macro Assembler (M80/L80).

Foram apresentados alguns conceitos sobre o ambiente de programação: há apenas 64K posições de memória, a linguagem assembly sendo usada é a do processador (Z80) e que para carregar dados em seus registradores usamos a instrução LoaD (LD). Foi visto que para o Z80 para executar uma função do CP/M (STROUT) é necessário indicar o numero da função no registrador C (LD C, STROUT) e mandar ele executar essa função no endereço BDOS (endereço 0005h), pedindo para que ele volte e continue com o programa após a execução da tarefa - usando para isso a instrução CALL (CALL BDOS). Finalmente foi apresentado como voltar ao prompt do CP/M, usando o comando JumP para o endereço 0 da memória (JP 0).

Adicionalmente, foram apresentados alguns conceitos sobre o *assembler* em uso (o M80/L80): como colocar dados na memória (APELIDO: DB 'dado') e como dar apelidos a endereços de memória (APELIDO EQU endereço). Foi apresentada a idéia de que dados e programas convivem na memória e que, se for comandado ao Z80 que ele execute um conjunto de dados (JP FRASE ou CALL FRASE), ele realmente vai tentar (e provavelmente o resultado não será bom); assim, é necessário evitar que o Z80 chegue a "executar dados".

Por fim, foi visto como "assemblar" o programa com o M80/L80, ou seja, como tornar o código assembly escrito algo legível para o computador, além de apresentar o uso do Z80MU.

Nesta aula os conceitos vistos anteriormente serão usados e extendidos, usando várias novas "funções" do CP/M, indicando como ler uma tecla e apresentar o valor lido.

1. Tornando o Programa Mais Completo

Inicialmente, é interessante relembrar o programa no estágio como ficou na aula anterior:

```
STROUT     EQU   9
BCPM EQU   5

START:
           LD    DE, FRASE          ; Indica endereço do texto
           LD    C, STROUT          ; Indica função do CP/M
           CALL  BCPM               ; Solicita execução pelo CP/M
           JP    0                  ; Volta ao prompt do CP/M

FRASE:     DB    'Alo, Mundo$'

           END
```

Antes de mais nada, é interessante criar um arquivo com esse conteúdo (PROG2.MAC, no diretório \ASM) para que as mudanças possam ser executadas. Em seguida, modifiquemos o apelido "FRASE" para um mais adequado, como **NOMEDOPRG** (Nome do Programa), alterando também a frase para **Programa 2 - Conversando com o usuário**.

```
STROUT     EQU   9
BCPM EQU   5

START:
           LD    DE, NOMEDOPROG     ; Indica endereço do nome
           LD    C, STROUT          ; Indica função do CP/M
           CALL  BCPM               ; Solicita execução pelo CP/M
           JP    0                  ; Volta ao prompt do CP/M

NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario$'

           END
```

Assim, o "nome do programa" sempre será apresentado quando o programa for iniciado. Se o programa for assemblado (CL80 PROG2) e depois executado dentro do Z80MU, ele deve imprimir a frase. Quando se programa em assembly é bastante útil "assemblar" e testar muito bem cada pequena mudança, para evitar dificuldades em resolver eventuais bugs.

Uma segunda alteração será acrescentar o nome do autor ao programa, o que pode ser feito da mesma maneira com que foi impresso o nome do programa, definindo-se um novo texto com um novo apelido:

```
STROUT     EQU   9
BCPM EQU   5

START:
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M
           JP    0                 ; Volta ao prompt do CP/M

NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario$'
AUTOR:     DB    ' Por Seu_Nome_Aqui$'

           END
```

Mas ainda falta a impressão do autor na tela; é preciso escrever o código para que o Z80 faça isso, sendo ele exatamente o mesmo que foi usado para imprimir o nome do programa:

```
STROUT     EQU   9
BCPM EQU   5

START:
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           JP    0                 ; Volta ao prompt do CP/M

NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario$'
AUTOR:     DB    ' Por Seu_Nome_Aqui$'

           END
```

Bem, o código começou a ficar meio sujo; uma forma de "sujar para limpar" é acrescentar alguns comentários de blocos. Comentários de blocos são comentários que ficam alinhados com as instruções e explicam em linhas gerais o que as próximas linhas de código fazem. Programadores experientes, em situações normais, fazem apenas comentários de blocos. Alguns comentários deste tipo são apresentados na listagem a seguir.

```
STROUT     EQU   9
BCPM EQU   5

START:
           ; Mostra nome do programa
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra nome do autor
           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
```

```
            CALL  BCPM              ; Solicita execução pelo CP/M

            JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario$'
AUTOR:      DB    ' Por Seu_Nome_Aqui$'

            END
```

Ao assemblar este programa, algo parece ter saído errado: o nome do autor e o nome do programa estão saindo "grudados". Bem, para que uma linha seja pulada, é preciso inserir um código no texto impresso, para solicitar essa "quebra" de linha. Na verdade, é preciso adicionar **dois** códigos: um que pula uma linha (Line Feed, ou LF) e outro que volta o cursor para a posiçao 0 (Carriage Return, ou CR). O Line Feed é representado pelo número 10 e o Carriage Return pelo código 13. Desta forma, basta adicionar estes números nos dados para que uma linha seja pulada, como por exemplo:

```
NOMEDOPRG: DB  'Programa 2 - Conversando com o usuario$',13,10
```

MAS, se isso for feito, nada mudará... porque o caractere que indica o "fim do texto" ($) está *antes* dos códigos 13 e 10 (e, portanto, estes códigos não estão sendo interpretados na impressão do texto). Assim, estes códigos precisam estar entre o fim do texto e o caractere $, como pode ser visto na listagem a seguir:

```
STROUT      EQU   9
BCPM EQU    5

START:
            ; Mostra nome do programa
            LD    DE, NOMEDOPROG    ; Indica endereço do nome
            LD    C, STROUT         ; Indica função do CP/M
            CALL  BCPM              ; Solicita execução pelo CP/M

            ; Mostra nome do autor
            LD    DE, AUTOR         ; Indica endereço do autor
            LD    C, STROUT         ; Indica função do CP/M
            CALL  BCPM              ; Solicita execução pelo CP/M

            JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:      DB    ' Por Seu_Nome_Aqui$'

            END
```

O próximo passo será solicitar ao usuário que pressione uma tecla, para que seja possível executar alguma atividade com base nesta resposta. Entretanto, para solicitar que o usuário pressione uma tecla, é preciso ainda imprimir mais uma informação na tela, o que pode ser feito imprimindo o texto PERGU1, como no código a seguir, lembrando de adicionar os códigos para pular linha na frase anterior.

```
STROUT     EQU   9
BCPM EQU   5

START:
           ; Mostra nome do programa
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra nome do autor
           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra pergunta
           LD    DE, PERGU1        ; Indica texto da pergunta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:     DB    ' Por Seu_Nome_Aqui',10,10,13,'$'
PERGU1:    DB    ' Pressione alguma tecla: $'

           END
```

Observe que foram adicionados dois códigos 10 na frase da etiqueta AUTOR. Isso faz com que sejam puladas duas linhas após o nome do autor. O resultado da execução deste programa deve ser algo assim:

```
A>PROG2
Programa 2 - Conversando com o usuário
  Por Daniel Caetano

  Pressione alguma tecla:
A>
```

Assim, o programa mostra tudo que é necessário, mas ainda falta receber a tecla do usuário. Bem, no CP/M isso é feito com a função 1 (apelidada, comumente, de CONIN, de CONsole IN). Para chamá-la, basta indicar seu número no registrador C e chamar o endereço 5 (CP/M) da mesma forma com que era feito para imprimir um texto:

```
LD    C,1   ; Indica a função que pega uma tecla
CALL  5     ; Chama CP/M
```

Ou, usando os apelidos,

```
CONIN EQU   1
BCPM  EQU   5
LD    C,CONIN     ; Indica a função que pega uma tecla
CALL  BCPM        ; Chama CP/M
```

Adicionando estes ao programa, o resultado será:

```
CONIN       EQU   1
STROUT      EQU   9
BCPM        EQU   5


START:
            ; Mostra nome do programa
            LD    DE, NOMEDOPROG    ; Indica endereço do nome
            LD    C, STROUT         ; Indica função do CP/M
            CALL  BCPM              ; Solicita execução pelo CP/M

            ; Mostra nome do autor
            LD    DE, AUTOR         ; Indica endereço do autor
            LD    C, STROUT         ; Indica função do CP/M
            CALL  BCPM              ; Solicita execução pelo CP/M

            ; Mostra pergunta
            LD    DE, PERGU1        ; Indica texto da pergunta
            LD    C, STROUT         ; Indica função do CP/M
            CALL  BCPM              ; Solicita execução pelo CP/M

            ; Recebe uma tecla
            LD    C, CONIN          ; Indica função do CP/M
            CALL  BCPM              ; Solicita execução pelo CP/M

            JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:      DB    ' Por Seu_Nome_Aqui',10,10,13,'$'
PERGU1:     DB    ' Pressione alguma tecla: $'

            END
```

Ao assemblar e executar este código, é possível ver que ele será executado normalmente, pedirá que uma tecla seja digitada e então o programa é finalizado. Mas como conseguir o valor digitado pelo usuário?

Como visto nas aulas teóricas, o lugar usual em que a ULA coloca os resultados das operações é o registrador A (Acumulador). Para não fundir a cabeça dos programadores, quase sempre se prepara funções que retornam valor para que também coloquem seus resultados neste registrador. Assim, como CONIN não é uma função estranha, o valor numérico da tecla pressionada está no registrador A (seu valor ASCII).

Para apresentar esta informação para o usuário, é preciso primeiro construir uma frase de resposta que será impressa, como por exemplo:

```
RESP1:      DB    13,10,10,'  A tecla pressionada foi: $'
```

Observe que o texto se inicia com a indicação para que duas linhas sejam puladas e nenhuma linha é pulada ao fim do texto. Isso ocorre para que as linhas sejam puladas após o usuário digitar o valor mas não entre o texto "a tecla foi pressionada: " e o valor ser

apresentado. A apresentação deste texto deve ser feita depois que o Z80 recebeu a tecla que foi pressionada, como pode ser visto no código a seguir:

```
CONIN      EQU   1
STROUT     EQU   9
BCPM       EQU   5

START:
           ; Mostra nome do programa
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra nome do autor
           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra pergunta
           LD    DE, PERGU1        ; Indica texto da pergunta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Recebe uma tecla
           LD    C, CONIN          ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Apresenta resposta
           LD    DE, RESP1         ; Indica texto da resposta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:     DB    ' Por Seu_Nome_Aqui',10,10,13,'$'
PERGU1:    DB    ' Pressione alguma tecla: $'
RESP1:     DB    13,10,10,'  A tecla pressionada foi: $'

           END
```

Quase tudo pronto, mas falta ainda mostrar a tecla pressionada na tela. Isso pode ser feito com a função CONOUT (número 2, CONsole OUT) do CP/M, que faz exatamente o inverso de CONIN: pega um código ASCII de uma tecla e apresenta o texto dela na tela. Uma diferença importante, entretanto, é que o CONIN recebe o valor do teclado no registrador A, enquanto o CONOUT envia o valor da tecla presente no registrador E para a tela; por isso, antes de enviar o valor para a tela, é preciso copiá-lo do registrador A para o E, usando a instrução LoaD:

```
LD    E,A
```

Observe que a instrução LD E, A executa algo similar a E := A, ou seja, o valor que existia em E será **sobrescrito** e o número que estava em A continua em A. Inserindo a nova função no código, é obtido:

```
CONIN      EQU   1
CONOUT     EQU   2
STROUT     EQU   9
BCPM       EQU   5

START:     ; Mostra nome do programa
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra nome do autor
           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra pergunta
           LD    DE, PERGU1        ; Indica texto da pergunta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Recebe uma tecla
           LD    C, CONIN          ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Apresenta resposta
           LD    DE, RESP1         ; Indica texto da resposta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Apresenta caractere digitado pelo usuário
           LD    E,A               ; Coloca em E o caractere
           LD    C, CONOUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:     DB    ' Por Seu_Nome_Aqui',10,10,13,'$'
PERGU1:    DB    ' Pressione alguma tecla: $'
RESP1:     DB    13,10,10,'  A tecla pressionada foi: $'

           END
```

Aparentemente está tudo ok, mas se alguém assemblar este programa e tentar executar este programa, ele sempre responderá que a tecla pressionada foi a tecla ' '. Onde está o problema? Este é um problema comum em assembly e linguagem de máquina e os programadores precisam ficar atentos.

Pelo código anterior, é possível observar que entre ser lido o caractere e a apresentação do caractere existe um trecho de código que imprime a primeira parte da resposta ("Apresenta Resposta"). Se esta parte for comentada (com um ";" no início de cada linha) e o código assemblado, será possível ver que o problema some (a tecla pressionada será repetida logo ao seu lado)! O que aconteceu?

O que aconteceu foi que quando foi chamada a função STROUT para imprimir o primeiro trecho da mensagem, ela **modificou** o valor do registrador A, onde havia sido armazenado o caractere digitado. Mas como evitar este problema? Existem duas formas: a primeira delas é descobrindo um registrador que a função STROUT não use, e armazenando o valor neste registrador. Infelizmente a função STROUT modifica quase todos os registradores, o que nos leva a uma outra alternativa: guardar este valor na **memória!**

Mas como fazer isso? Bem, primeiro é necessário criar um local para armazenamento, com uma etiqueta, dentro de nosso programa. Isso pode ser feito com a seguinte linha:

```
VAR1:       DB    0
```

Isso definiu um byte de memória onde se pode ler (como foi feito com os textos) ou escrever, com a etiqueta VAR1 (de VARiável 1), com o valor inicial igual a 0. Mas como colocar um valor neste endereço de memória? Exatamente com a instrução LoaD. Ela também serve para copiar coisas da **memória** para um **registrador**, ou de um **registrador** para a **memória**. Se a idéia é armazenar o dado do registrador A na posição de memória VAR1, nada mais natural do que um comando:

```
LD    VAR1, A
```

Mas, infelizmente, isso não funciona, porque esta ordem não é clara. É importante lembrar que VAR1 é uma etiqueta que é convertida para um número, no momento da assemblagem. Assim, no momento da assemblagem isso poderia virar uma instrução bizarra do tipo:

```
LD    1253, A
```

O que, em uma linguagem como Pascal ou Delphi equivaleria a escrever:

```
1253 := X ;
```

Certamente isso ia causar um erro no compilador porque não podemos mudar o valor de um número. 1253 é e sempre será 1253. Esta instrução escrita tentaria **mudar o nome do endereço de memória**, o que é impossível e nada desejável. O que se deseja é mudar o **conteúdo** deste endereço de memória. Para indicar isso, em assembly, usamos **parênteses** ao redor do número (ou etiqueta), da seguinte forma:

```
LD    (VAR1), A
```

Isso será compreendido pelo Z80 como "Escreva o valor do registrador A na posição de memória cujo nome é VAR1". Inserindo isso no código *logo após a leitura do caractere*, tem-se:

```
CONIN      EQU   1
CONOUT     EQU   2
STROUT     EQU   9
BCPM       EQU   5

START:     ; Mostra nome do programa
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra nome do autor
           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra pergunta
           LD    DE, PERGU1        ; Indica texto da pergunta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Recebe uma tecla
           LD    C, CONIN          ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M
           LD    (VAR1),A          ; Armazena valor lido na memória

           ; Apresenta resposta
           LD    DE, RESP1         ; Indica texto da resposta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Apresenta caractere digitado pelo usuário
           LD    E,A               ; Coloca em E o caractere
           LD    C, CONOUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:     DB    ' Por Seu_Nome_Aqui',10,10,13,'$'
PERGU1:    DB    ' Pressione alguma tecla: $'
RESP1:     DB    13,10,10,'  A tecla pressionada foi: $'
VAR1:      DB    0

           END
```

Agora o dado já está preservado na memória, só falta lê-lo de volta para o registrador E no momento em que for necessário usá-lo. Infelizmente, a instrução:

```
LD    E, (VAR1)
```

Não funciona. A Unidade de Controle do Z80 não sabe ler valores da memória diretamente para o registrador E, ela precisa usar o registrador A para isso... e só então será possível copiar o valor lido do registrador A para o E, com a seqüência:

```
LD    A, (VAR1)
LD    E, A
```

O resultado pode ser visto no próximo trecho de código:

```
CONIN      EQU   1
CONOUT     EQU   2
STROUT     EQU   9
BCPM       EQU   5

START:     ; Mostra nome do programa
           LD    DE, NOMEDOPROG    ; Indica endereço do nome
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra nome do autor
           LD    DE, AUTOR         ; Indica endereço do autor
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Mostra pergunta
           LD    DE, PERGU1        ; Indica texto da pergunta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Recebe uma tecla
           LD    C, CONIN          ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M
           LD    (VAR1),A          ; Armazena valor lido na memória

           ; Apresenta resposta
           LD    DE, RESP1         ; Indica texto da resposta
           LD    C, STROUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           ; Apresenta caractere digitado pelo usuário
           LD    A,(VAR1)          ; Recupera valor do caract. em A
           LD    E,A               ; Coloca em E o caractere
           LD    C, CONOUT         ; Indica função do CP/M
           CALL  BCPM              ; Solicita execução pelo CP/M

           JP    0                 ; Volta ao prompt do CP/M

; Dados do Programa
NOMEDOPROG: DB 'Programa 2 - Conversando com o usuario',10,13,'$'
AUTOR:     DB    ' Por Seu_Nome_Aqui',10,10,13,'$'
PERGU1:    DB    ' Pressione alguma tecla: $'
RESP1:     DB    13,10,10,'  A tecla pressionada foi: $'
VAR1:      DB    0

           END
```

3. Bibliografia

ROSSINI, F; LUZ, H.F. **Linguagem de Máquina**: Assembly Z80 - MSX. 1ed. São Paulo: Ed. Aleph, 1987.
CARVALHO, J.M. **Assembler para o MSX**. 1ed. São Paulo: McGraw Hill, 1987.
ASCII Corporation. **O Livro Vermelho do MSX**. 1ed. São Paulo: Ed. McGraw Hill, 1988.

Notas das Aulas 08: 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. Desempenho e Comparação Controversa

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 2.57MHz, 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 será convertido em:

```
            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.

Notas das Aulas 09: Barramentos
Prof. Daniel Caetano

Objetivo: Apresentar os tipos de barramentos 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

Como foi apresentado em aulas anteriores, a CPU precisa se comunicar com os periféricos, sendo esta comunicação feita através do barramento. Entretanto, os periféricos com que a CPU precisa se comunicar são dos mais diversos tipos, com as mais diversas velocidades. Por esta razão, alguns cuidados precisam ser tomados para que a comunicação possa ocorrer sem problemas.

Nesta aula serão apresentados os tipos mais comuns de barramento e seus mecanismos de funcionamento. Será apresentada também uma visão geral sobre os mecanismos de arbitragem de controle.

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.

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.

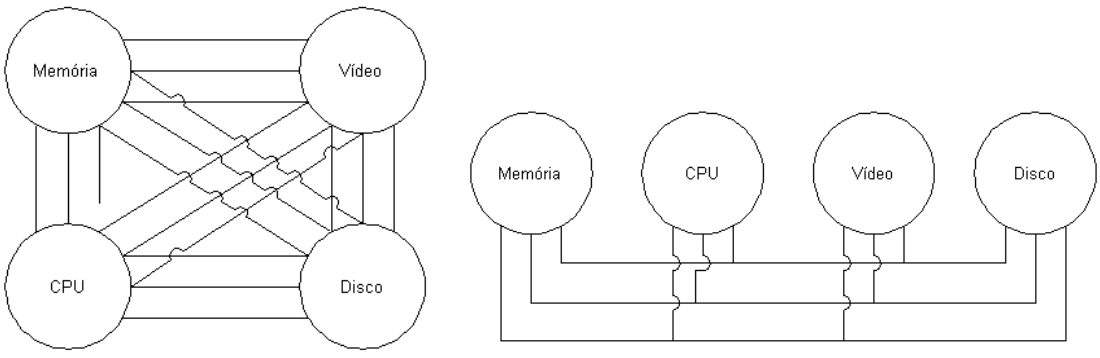


Figura 1: Elementos com ligação direta (esquerda) e através de barramento (direita)

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 2 (MURDOCCA, 2000) o diagrama do sinal de *clock* em um barramento de 100MHz:

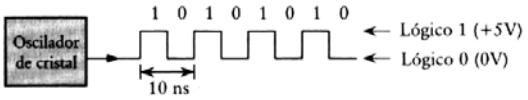


Figura 2: Sinal de *clock* (ao longo do tempo) de um barramento de 100 Mhz

Na pratica, 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 lento 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.

2. 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 3.

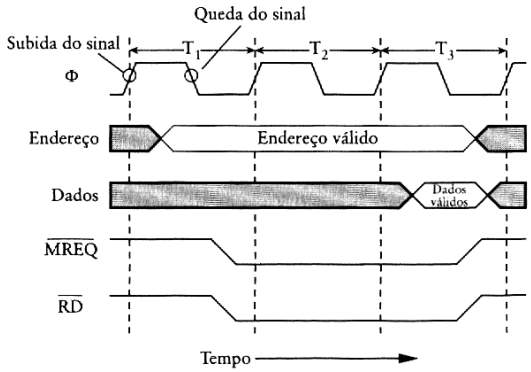


Figura 3: Um ciclo de barramento para leitura de memória
(fonte: Tanenbaum, 1999, apud Murdocca, 2000)

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.

3. Barramento Assíncrono

Ao contrário dos barramentos síncronos, os barramentos assíncronos não funcionam com base em ciclos de *clock* de barramento, mas sim com sinais que são chamados ***hand shakes*** (apertos de mão) de sincronização.

Este mecanismo de sincronização pode ser feito com duas linhas de controle adicionais: MSYN (*Master SYNchronization*, ou "Sincronização do Mestre", em português) e SSYN (*Slave SYNchronization*, ou "Sincronização do Escravo", em português). O sinal de *clock* não é mais necessário. Mais uma vez, o funcionamento será apresentado como um exemplo de leitura da memória por parte da CPU, para facilitar a comparação com o barramento síncrono.

Como no barramento síncrono, 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 é indicar o endereço desejado no barramento de endereços (através do registrador MAR).

Após a espera pelo tempo de estabilização dos sinais de endereço, a CPU indicará os outros dois sinais no barramento de controle: o de requisição de memória (MREQ) e o sinal de leitura (RD), indicando que deseja ler a memória.

Neste ponto, ao invés de simplesmente esperar a resposta da memória por um determinado tempo, a CPU indicará um terceiro sinal: MSYN, indicando para a memória que a CPU já finalizou a configuração dos barramentos e está apenas esperando a memória responder.

Assim que a memória perceber os sinais MREQ, RD, ela aguardará o MSYN, que será o aviso da CPU que, da parte dela, está tudo pronto. Ao perceber o sinal MSYN, a memória executará as operações necessárias, isto é, usará o endereço do barramento de endereços para escolher o dado e então colocará este dado no barramento de dados. A memória espera, então, o sinal do barramento de dados estabilizar e, em seguida, indica o sinal SSYN, informando à CPU que o dado está pronto para leitura, no barramento de dados.

Assim que a CPU perceber o sinal SSYN, ela lê o dado do barramento de dados e retira os sinais MREQ, RD e MSYN. A memória percebe que o MSYN foi inativado e inativa o sinal SSYN. Quando a CPU percebe que o SSYN foi inativado, libera o barramento de controle para outros circuitos poderem utilizá-lo. Todo esse processo ao longo do tempo pode ser visto no diagrama da figura 4.

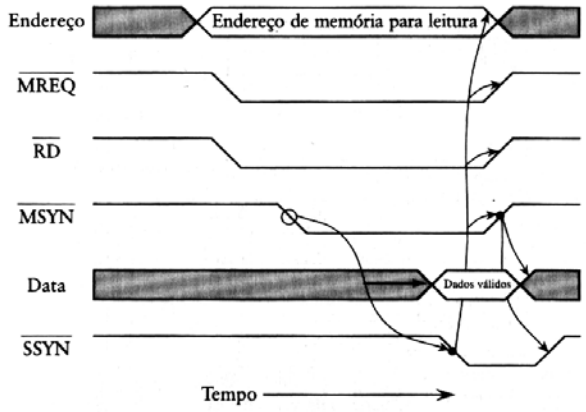


Figura 4: Um "ciclo" de barramento para leitura de memória
(fonte: Murdocca, 2000)

É importante observar que neste caso a cadência da operação é limitada somente pela velocidade dos componentes: quanto mais rápido eles operarem, mais rapidamente a operação como um todo será executada. Com isso, existe uma adeqüação automática da velocidade dos dispositivos.

Entretanto, a implementação deste tipo de barramento, como já dito anteriormente, é mais complexa devido à dificuldade de identificação de problemas de sincronia entre os dispositivos, perdas de dados etc. Por esta razão, não é comum que as CPUs se comuniquem diretamente com os dispositivos de forma assíncrona. Nos computadores pessoais, em geral a CPU se comunica de forma síncrona com um circuito gerenciador de acesso ao barramento, que se comunica de forma síncrona com os dispositivos. Globalmente, entretanto, tudo funciona como se a CPU se comunicasse de forma *assíncrona* com os periféricos, permitindo que CPU e periféricos (incluindo a memória) operem em velocidades (*clocks*) distintos. Note,

porém, que é uma comunicação *assíncrona aparente*: na prática temos comunicações síncronas, sendo que um dos circuitos intermedia as operações.

4. Barramentos Baseados 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 5. Na figura, o North Bridge está claro; o South Bridge, não.

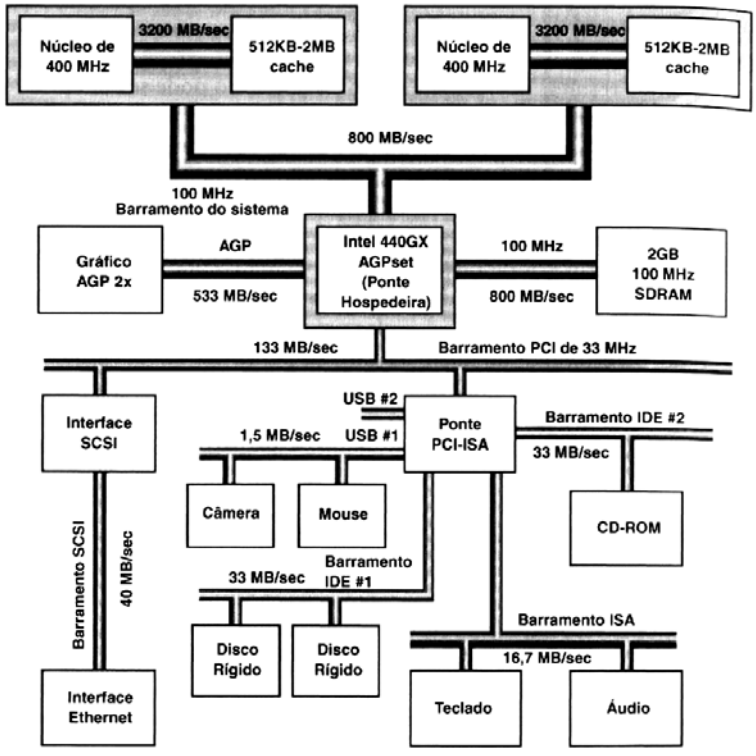


Figura 5: Arquitetura de Barramento em Pontes
(fonte: Intel apud Murdocca, 2000)

5. Arbitragem de Barramento

Como dito anteriormente, em algum momento é necessário decidir quem será o mestre do barramento (tornando escravos todos os outros dispositivos). Quando apenas um dispositivo solicita o barramento, esta é uma tarefa simples; por outro lado, quando mais de um dispositivo solicita o barramento *ao mesmo tempo*, é preciso haver um critério de *arbitragem*. Existem dois esquemas básicos para realizar esta arbitragem: o **centralizado** e o **descentralizado**.

Arbitragem Centralizada

No esquema centralizado deve existir um circuito que será o **árbitro**. Todo circuito que quiser utilizar o barramento, deve primeiramente solicitar permissão ao árbitro e só poderá se tornar o mestre do barramento quando o árbitro autorizar.

A ligação dos dispositivos com o árbitro é feita da seguinte forma: existe uma linha do barramento de controle que serve para a solicitação, chamada BUSRQ (BUS ReQuest). Esta linha é única, de forma que todos os dispositivos podem solicitar ao mesmo tempo, na mesma linha. Quando o árbitro percebe uma solicitação, ele não sabe de quem a solicitação veio, mas se o barramento estiver disponível, ele indica em uma linha chamda BUSACK (BUS ACKnowledge) que o circuito pode utilizar o barramento.

Certo, mas isso ainda não resolve o problema da arbitragem: se mais de um circuito solicitar ao mesmo tempo, qual deles irá receber a autorização? Simples: o mais próximo do circuito árbitro. Isso é conseguido porque a linha BUSACK não chega diretamente a todos os dispositivos: chega apenas ao primeiro. Caso este dispositivo **não** tenha solicitado o barramento, ele é **responsável** por passar o sinal BUSACK para o dispositivo seguinte. O dispositivo seguinte então, recebe o BUSACK e, se também ele não tiver solicitado o barramento, repassa o sinal adiante.

Este mecanismo funciona *em cascata* até que um circuito que tenha solicitado o barramento receba o sinal e se torne o mestre do barramento. Este tipo de ligação, em que os circuitos são responsáveis por passar a informação adiante quando for adequado é chamado de ligação em linha (*daisy chained*).

Este tipo de ligação estabelece automaticamente uma ordenação de prioridade que será cegamente respeitada. Assim, os dispositivos com mais prioridade de receber atenção do bus devem ser colocados mais próximos do circuito árbitro.

Uma forma mais incomum deste tipo de circuito é cada dispositivo ter uma linha BUSRQ e BUSACK própria (BUSRQn e BUSACKn). Este tipo de ligação permite que o próprio árbitro decida as questões de prioridade (e não a ligação física dos dispositivos). Entretanto, isso limita o número de dispositivos que se pode ligar no equipamento. Uma solução é adotar um mecanismo composto, em que há vários BUSRQ e vários BUSACK e, em cada um deles, vários dispositivos estão ligados em forma de *daisy chain*. Neste caso,

cada par BUSRQ e BUSACK representará uma prioridade principal, e a ordem dentro da *daisy chain* de cada um deles representará uma prioridade secundária.

A figura 6 (MURDOCCA, 2000) apresenta o esquema gráfico do esquema centralizado com uma linha de requisição e uma de permissão.

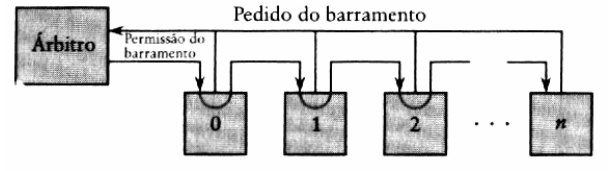


Figura 6: Arbitragem Centralizada

Arbitragem Descentralizada

Um outro esquema de abritragem é o descentralizado, onde a tarefa de definir quem comanda o barramento é de cada dispositivo.

Neste caso, existem três linhas: a de pedido de barramento (BUSRQ), a de barramento ocupado (BUSBSY) e a de premissão de uso do barramento (BUSACK). Estas linhas podem ser vistas na figura 7 (MURDOCCA, 2000).

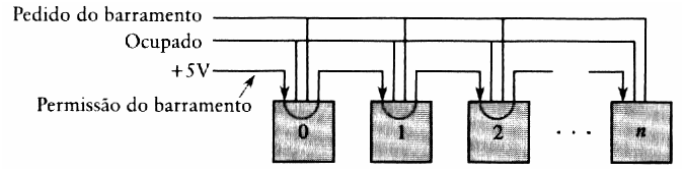


Figura 7: Arbitragem Descentralizada

Quando um dispositivo quer se tornar o mestre do barramento, primeiramente ele verifica se o barramento está ocupado, através do BUSBSY. Se não estiver, ele envia um sinal na linha BUSRQ, solicitando o uso do barramento. A permissão de barramento (BUSACK) será enviada apenas se BUSRQ estiver ativo **ao mesmo tempo** em qeu o BUSBSY estiver inativo.

Este sinal é, normalmente, uma simples tensão +5V, que é repassada pelos dispositivos na forma de *daisy chain*, estabelecendo as prioridades. Quando o dispositivo de maior prioridade que houver solicitado o barramento receber o sinal de permissão no BUSACK, ele retirará o sinal do BUSRQ, indicará sinal no BUSBSY e **não repassará** o BUSACK para os próximos dispositivos.

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.

Notas das Aulas 10: Dispositivos de Entrada e Saída
Prof. Daniel Caetano

Objetivo: Apresentar os métodos de comunicação com dispositivos de E/S e alguns destes dispositivos 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.

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 será 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. 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", agindo num sistema com barramentos como se fosse um sistema de Von Neumann.

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.

2. 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.

3. 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 antetiormente.

4. 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.

4.1. 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.

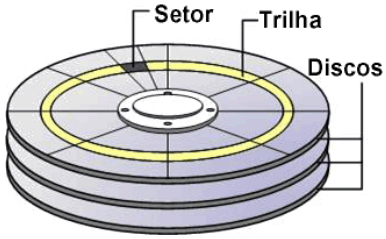


Figura 1: Discos, trilhas e setores

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".

4.2. 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.

4.3. 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.

4.4. 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).

4.5. 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).

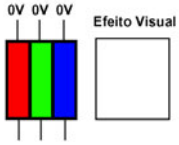
b) um sistema de iluminação (que deve iluminar uniformemente o papel de fundo)

c) uma tela de cristal líquido

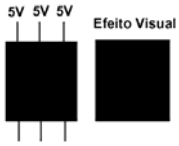
A tela de cristal líquido é composta por uma matriz de elementos de cristal líquido. O cristal líquido tem uma propriedade que é a de impedir que a luz passe em uma dada direção quando colocado sobre uma diferença de potencial suficientemente alta. A quantidade de luz que o cristal líquido deixa passar 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:

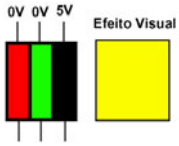
1) Campo baixo em R, G e B: cor branca (todas as cores "acesas")



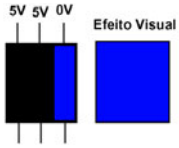
2) Campo alto em R, G, B: cor preta (todas as cores "apagadas")



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.

Notas das Aulas 11: Comunicação e Tecnologias Atuais
Prof. Daniel Caetano

Objetivo: Apresentar introdutoriamente os mecanismos de comunicação entre computadores e algumas tecnologias atuais no tocante à arquitetura de computadores.

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

Até alguns anos atrás, a grande maioria das funções de um computador eram desempenhadas "offline", ou seja, desconectado de uma rede. Os usuários estavam habituados a usar o computador apenas para tarefas que não requeriam comunicação em tempo real com outras pessoas ou outros computadores.

Nos últimos 10 ou 15 anos, entretanto, a situação se inverteu. Os usuários ainda utilizam muito o computador para atividades "isoladas", mas quase sempre usam, em paralelo, softwares de comunicação para trocar mensagens com conhecidos ou mesmo usam conexões com outros equipamentos para compartilhar arquivos e impressoras. Na realidade, a maior parte dos usuários atuais sente que se não houver comunicação entre computadores, a maior parte da funcionalidade do mesmo está perdida.

É claro que isso não é verdade, mas também é claro que a comunicação entre equipamentos é um aspecto importante dos computadores e sua arquitetura; a comunicação entre equipamentos já tem uma grande importância e esta mesma importância crescerá ainda mais nos próximos anos.

Mas nem só de comunicação vivem os computadores modernos. Essa afirmação é facilmente verificada na prática com a constante busca por *aceleramento do processamento*, que culminou, nos dias atuais, com a popularização dos sistemas multicore e multiprocessados.

Nesta aula serão apresentados alguns conceitos de arquitetura destas tecnologias.

1. Comunicação por Modem

Já nos primeiros computadores percebeu-se a necessidade de comunicação entre máquinas próximas. Essa comunicação era freqüentemente feita através das portas paralelas ou, mais comumente, das portas seriais.

As portas seriais nada mais são que dispositivos que servem para interligar dois equipamentos, formando um "túnel" de informações. Desta maneira, quando um computador escrevia em sua porta serial, o outro computador conectado a ela poderia ler este mesmo dado através de sua própria porta serial.

Este era um sistema simples, mas tinha um problema: a distância de comunicação era limitada, seja pelo comprimento do cabo, seja pela dificuldade de estender fios por todo o país interligando as máquinas.

Pensando nestes problemas, alguém percebeu que, na realidade, ambos já estavam resolvidos desde o fim do século XIX e início do século XX. A tecnologia que havia resolvido o problema era a tecnologia das **linhas telefônicas**. Esta pessoa pensou que, caso fosse possível fazer os computadores *conversarem por telefone*, o problema da comunicação entre computadores distantes estaria resolvido: bastava que houvesse uma linha telefônica por perto.

Para que os computadores pudessem conversar pelo telefone, os sinais elétricos trocados pela porta serial teriam de ser convertidos em forma de onda sonora, num processo chamado de **modulação**. Da mesma forma, para que o outro computador pudesse entender essa forma de onda sonora, ela teria de ser convertida de volta nos sinais que entrariam por sua porta serial, num processo chamado de **demodulação**. Por esta razão, os equipamentos que sabiam fazer ambas as funções (já que os dois computadores precisavam ser capazes de enviar e receber informações) ficou conhecido como **modulador-demodulador**, nome rapidamente abreviado para **modem**.

O modem que transmite nada mais faz do que pegar uma seqüência de bits e transformá-los em uma forma de onda sonora. Existem várias formas de "modular" uma onda: por **amplitude (AM)**, por **freqüência (FM)** e por **fase (PM)**. Por amplitude o sinal zero é representado por um volume médio menor e o sinal 1 por um volume médio maior. Por freqüência o sinal 0 é representado por uma freqüência menor da onda e o sinal 1 por uma freqüência maior. Por fase, o sinal 0 é represenado por uma forma de onda e o sinal 1 é representado por um deslocamento desta forma de onda. A figura 1 (MURDOCCA, 2000) mostra um exemplo.

É claro que podem ocorrer erros de sinal por deficiência da linha de comunicação, ocasionando perdas de dados. Por esta razão, é necessária a existência de protocolos de correção de erros, que fogem ao escopo deste curso. De qualquer forma, a incidência de erros é tão alta quanto maior for a velocidade de transmissão, sendo a ocorrência de erros um dos limitantes da velocidade de transmissão. Existem outros, entretanto, como o meio de transmissão (tipo de cabeamento etc).

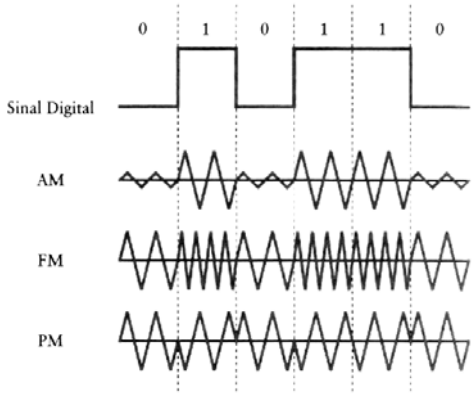


Figura 1: Exemplo de modulação de sinal digital

2. Comunicação por Redes

É claro que a comunicação entre dois computadores era interessante, mas a forma de funcionar das portas seriais e dos modems impunha um limite de comunicação: se três computadores tivessem que se comunicar entre si, um deles teria que possuir dois modems, usar duas linhas telefônicas e ainda por cima teria que ficar transportando dados entre os outros dois computadores (além de sua própria comunicação). Uma forma de resolver isso parcialmente seria dotar os três computadores de dois modems e passar a usar 3 linhas telefônicas, mas isso seria um enorme disperdício.

Entretanto, quando os computadores estavam próximos, decidiram que este esquema até seria interessante, mas não seria necessário usar uma linha telefônica. Desta forma, uma espécie de ligação direta entre portas seriais seria suficiente... ou quase. Quando vários computadores estão conversando por uma mesma linha, é razoável que a chance de erros aumente (por exemplo: dois computadores tentando "falar" ao mesmo tempo). Por esta razão, desenvolveram um dispositivo específico chamado "dispositivo de rede", que era capaz de coordenar a transmissão de dados. Este primeiro tipo de rede, onde um computador era interconectado no seguinte até formar um anel foi chamado de **token ring**. As informações eram passadas com o "endereço" do computador destino e, cada computador que recebesse uma informação que não era para ele, passava para o computador seguinte, até que a informação chegase no dono.

Este tipo de rede, entretanto, tinha vários inconvenientes e não é mais usado nos tempos atuais. O primeiro destes inconvenientes é que se um computador for removido do anel, a rede para de funcionar corretamente (a menos que uma ligação sem esse computador seja feita). Um outro inconveniente é a necessidade de as placas de rede ficarem repassando as informações que não dizem respeito ao seu computador.

Da mesma forma que o modelo Von Neumann evoluiu para o modelo de Barramento, alguém percebeu que as redes **token ring** também podiam tirar benefício da existência de um **barramento**: todos os computadores seriam ligados ao barramento; apenas um escreveria por vez no barramento e todos os outros ouviriam o **mesmo** barramento, colhendo apenas as

informações que lhe dissessem respeito. Assim, as placas de rede deixavam de ter de repassar as informações de um computador para outro e, adicionalmente, isso acabava com o problema de reconfiguração do cabeamento quando um computador era desligado: desligar um computador do barramento não muda em nada o funcionamento do barramento.

Mas como as placas podem detectar se é o momento delas transmitir ou de ouvir a rede? Bem, todas as placas ouvem a rede o tempo todo, mas apenas uma deve transmitir. Assim, existe um sinal chamado "portadora da rede" que indica se algum equipamento está transmitindo no momento. Se houver portadora, a placa de rede que está querendo transmitir desiste temporariamente da transmissão e aguarda um tempo aleatório antes de tentar novamente. Se não houver portadora, a placa de rede querendo transmitir coloca a portadora e começa a transmitir seus dados.

Em linhas gerais, essa é a essência da comunicação em rede. É claro que existem complexidades adicionais de correção de erro, colisões (quando dois computadores começam a transmitir exatamente ao mesmo tempo) etc. Mas estes assuntos fogem ao escopo do curso.

3. Tecnologias Atuais - Processadores Multi Core

Como foi dito anteriormente, nem só de comunicação vive o computador. A necessidade de evolução da capacidade de processamento sempre foi uma constante no desenvolvimento de novos processadores e novas arquiteturas.

Em princípio, a solução para o aumento de processamento era aumentar o clock de operação. Como cada instrução demora um certo número de ciclos para ser executada, se for possível aumentar o número de ciclos por segundo, há um aumento da capacidade de processamento.

Entretanto, o aumento de clock faz com que o processador esquente mais, afinal, mais trabalho é realizado em menos tempo. Além disso, complicações elétricas surgem ao forçar o aumento da velocidade de processamento. Para contornar estes problemas, são realizadas mudanças na arquitetura interna da CPU, são modificados os materiais elétricos usados na confecção do microprocessador e até mesmo a tecnologia de confecção do microprocessador.

Mas, apesar de tudo que se tem feito, este tipo de melhoria chegou praticamente a um limite nos primeiros anos do século XXI. Por algum tempo foi possível aumentar o desempenho, então, reduzindo o clock mas melhorando a forma de processar os dados, fazendo com que um número menor de ciclos fosse necessário para executar as mesmas instruções. Mas este tipo de melhoria também tinha limitações.

Foi assim que as grandes empresas de projeto de processadores, como AMD e Intel, decidiram trazer para o dia-a-dia algo que era comum apenas em grandes laboratórios de universidades e empresas de tecnologia de ponta: o multiprocessamento central. Ou seja,

decidiram que era hora do mercado SOHO se beneficiar do uso de mais de um processador central em uma mesma máquina.

Existem dois tipos de multiprocessamento. Um deles é aquele em que temos CPUs que sozinhas são capazes de se coordenar (e não são necessariamente iguais). O outro tipo é aquele em que temos duas CPUs idênticas (e por isso esse é chamado de Multiprocessamento Simétrico, ou SMP), interligadas, mas que compartilham a mesma memória. Adicionalmente, o sistema operacional tem autonomia para solicitar a execução de um serviço para uma ou outra CPU, sendo sua responsabilidade dividir as tarefas. Nestes casos, entretanto, um programa simples **não tem como selecionar a CPU em que quer rodar**, sendo que o sistema operacional é quem vai decidir isso.

De qualquer forma, é necessário haver uma comunicação entre as CPUs, já que elas compartilham a mesma memória e o trabalho de uma pode depender da outra. Originalmente, eram usadas duas CPUs físicas, separadas pela placa de circuito do computador (placa mãe), e isso impunha uma série de restrições na velocidade de comunicação entre elas.

A AMD, entretanto, conseguiu criar o primeiro processador multicore para o mercado SOHO, o chamado Athlon 64 X2. Este processador unia as duas CPUs dentro de uma mesma "pastilha de silício", isto é, dentro do mesmo chip, cada uma com seu próprio cache. Com isso, a comunicação entre estas duas CPUs passou a ser extremamente rápida.

Nesta arquitetura, a AMD introduziu também um sistema chamado Hyper Transport, que é um barramento interno da CPU, de altíssima velocidade, que as duas CPUs utilizam para se comunicar **e** para se comunicar com a memória do micro, tendo com isso eliminado a necessidade do North Bridge. Em outras palavras, a AMD trouxe para dentro da CPU o controlador de memória. Como foi eliminado um intermediário externo na comunicação com a memória, as CPUs da AMD passaram a ter um aproveitamento da banda de memória bastante superior.

A Intel, que havia ficado para trás, decidiu apostar em uma outra otimização com seu processador Core Duo. Observando a tecnologia da AMD, a Intel percebeu que o uso dos caches do processador X2 não era eficiente. O problema residia no seguinte fato: ambos os caches eram do mesmo tamanho (por exemplo, 1MB para cada core), mas muitas vezes um core precisa de mais cache que o outro (dependendo da operação realizada). Assim, ter um tamanho máximo fixo para cada core não era uma boa estratégia.

Assim, a Intel, ao invés de trazer o controlador de memória para dentro da CPU (o que, de fato, aumenta a velocidade, mas aumenta também o preço da CPU), resolveu unificar os caches dos dois cores, sendo a quantidade de cache disponível para cada um dinamicamente. Assim, com os mesmos 2MB de cache, é possível que cada core tenha 1MB de cache como nos processadores AMD, mas também seria possível que um core tivesse momentaneamente 1,5MB e outro apenas 0,5MB, se isso for a alternativa mais eficiente, fazendo com que a CPU denha um desempenho superior **com pouco ou nenhum aumento de de custo**.

No momento, o que ocorre é a tentativa da Intel implementar algo similar ao HyperTransport, trazendo o controlador da memória para dentro da CPU (para aumentar a eficiência de comunicação com a memória) e a AMD tentando implementar algo similar ao cache unificado da Intel, além de baratear o custo de seu HyperTransport.

Enquanto a AMD saiu na frente e domuniou o mercado por alguns anos com o X2, a Intel passou à frente com o Core 2 Duo, com um desempenho superior a preços mais competitivos. Este ano estão sendo colocados no mercado os primeiros processadores Quad-Core (quatro núcleos), onde a tecnologia HyperTransport pode fazer mais diferença, talvez, do que um cache unificado. Por outro lado, fica a questão de até que ponto os softwares SOHO atuais são capazes de aproveitar quatro núcleos em um mesmo computador.

Uma coisa pode-se afirmar, entretanto: muitas mudanças estão no horizonte e é difícil dizer qual tecnologia prevalecerá. Talvez estas tecnologias utilizadas nos processadores atuais estejam próximas de seus limites, tornando ainda mais difícil prever o que acontecerá pelos próximos anos.

4. 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.