**Sistemas Computacionais – COM210**

**Semana 8 - TEXTO DE REVISÃO**

A disciplina de Sistemas Computacionais aborda temas relevantes para que você apreenda conceitos e conhecimentos essenciais para o projeto e o desenvolvimento de sistemas computacionais, de modo a maximizar o desempenho e a utilização de todos os elementos de um computador. Nesse sentido, a disciplina também incorpora temas das tradicionais disciplinas de Arquitetura e Organização de Computadores e de Sistemas Operacionais. Contudo, o assunto explorado não consegue ser completo, uma vez que há uma variedade de conceitos e de tecnologias, desde *chip* a supercomputadores em *grids*. Além disso, as tecnologias de computadores estão em permanente e rápida evolução. Desse modo, os assuntos de sistemas computacionais relacionados às tecnologias dependem do estado atual da tecnologia, bem como seu desempenho e fabricação.

Na **Semana 1**, apresenta-se a evolução histórica dos computadores destacando os elementos básicos que os compunham. Em termos históricos, os computadores podem ser classificados de acordo com a operação, por essa razão, podem ser computadores analógicos ou digitais. Em relação ao uso, podem ser computadores científicos, comerciais, pessoais, ou de lazer. E para classificação tecnológica, tem-se computadores mecânicos e de 1ª a 4ª geração.

A história do computador mecânico começou por volta de 1623, quando o cientista francês Blaise Pascal construiu uma máquina de calcular à manivela e que pode ainda não ser considerado um computador. Trinta anos depois, um matemático alemão construiu outra máquina mecânica equivalente a uma calculadora de bolso de quatro operações. No século XIX, o inventor do velocímetro, o famoso pesquisador Babbage, criou uma máquina analítica, a qual tinha quatro componentes: armazenagem (memória), "moinho" (unidade de cálculo), seção de entrada (leitora de cartões perfurados) e seção de saída (saída perfurada e impressa). Essa máquina lia instruções de cartões perfurados e as executava. Babbage pode ser considerado o avô do computador digital moderno. A primeira geração de computadores tem como base as válvulas. Foram construídos computadores eletrônicos com operações internas realizadas em milisegundos. Nessa época, estava ocorrendo a Segunda Guerra Mundial, que serviu de estímulo para a criação do computador eletrônico. A idéia era decodificar mensagens interceptadas por rádio, a partir de submarinos alemães. Nesse contexto, o governo britânico construiu, em um laboratório ultrassecreto, o primeiro computador digital eletrônico do mundo, o Colossus. O famoso matemático britânico Alan Turing coordenou o projeto e a construção dessa máquina. No período, outra máquina construída foi o Eniac, com 18.000 válvulas e 1.500 relés e pesando 30 toneladas. Na época, a IBM criou seu primeiro computador, o 701, que foi o primeiro de uma série de máquinas científicas que dominaram o setor durante uma década. Alguns pesquisadores e usuários não consideravam a tecnologia de válvulas muito confiável.

Em 1948, alguns cientistas inventaram o transístor, que é um amplificador de cristal. O transístor revolucionou os computadores e, no fim da década de 1950, os computadores a válvulas estavam obsoletos. Nesse contexto, tinha-se a geração de circuitos eletrônicos transistorizados, que realizava operações internas realizadas em microsegundos. O primeiro computador transistorizado recebeu o nome de TX-0. Em 1961, a empresa DEC (do inglês, *Digital Equipment Corporation*) lançou uma máquina comercial muito parecida com o TX-0: o PDP-1 (do inglês, *Programmed Data Processor-1*), nascendo, assim, a indústria de minicomputadores. Alguns anos mais tarde, a mesma DEC lançou o PDP-8, o qual apresentava a inovação de um barramento único. Essa arquitetura passou a ser adotada por quase todos os computadores de pequeno porte.

Em 1964, a CDC (do inglês, *Control Data Corporation*) desenvolveu a 6600, uma máquina que era mais rápida do que qualquer outra existente na época. O segredo de sua velocidade era o fato de haver, dentro da CPU, uma máquina com alto grau de paralelismo. Naquela época, enquanto alguns

projetistas de empresas buscavam tornar o hardware mais barato (como a DEC) ou mais rápido (como a IBM e a CDC), um grupo de pesquisadores adotou uma linha de ação diferente: eles construíram uma máquina com a intenção de programá-la em linguagem Algol 60, uma precursora da linguagem C e Java. Nascia, assim a ideia de que o software também era importante.

A terceira geração de computadores foi caracterizada por circuitos integrados que realizavam operações internas em nanosegundos. A invenção, em 1958, do circuito integrado de silício permitiu que dezenas de transistores fossem colocados em um único chip. Esse empacotamento possibilitou a construção de computadores menores, mais rápidos e mais baratos que os transistorizados.

Em 1964, a IBM lançou uma linha de produtos baseada em circuitos integrados (chips) denominada System/360. Uma importante inovação dessa linha foi a multiprogramação (vários programas na memória ao mesmo tempo), resultando em um melhor desempenho da CPU e o imenso espaço de endereçamento na memória para a execução de programas. Os minicomputadores deram um grande passo quando a DEC lançou a série PDP-11, um sucessor de 16 bits do PDP-8. O PDP-11 teve enorme sucesso, em especial nas universidades, e manteve a liderança da DEC no mercado de minicomputadores.

Finalmente, por volta de 1980, a quarta geração de computadores, a VLSI (do inglês, *Very Large Scale Integration*) com integração em altíssima escala deu origem aos computadores pessoais. A tecnologia VLSI possibilitou a colocação de milhões de transistores em um único chip. Essa tecnologia resultou em computadores menores e mais rápidos. Os preços caíram tornando viável cada indivíduo ter o próprio computador.

Na época, empresas como a Apple e a IBM desenvolveram computadores pessoais. O IBM PC (do inglês, *Personal Computer*) se tornou o maior campeão de vendas de computadores da história. Foi também nos anos 1980 o desenvolvimento do sistema operacional Windows pela Microsoft, o qual passou a ser utilizado em grande parte dos computadores pessoais do mundo. Também em meados da década de 1980, as complicadas arquiteturas CISC começaram a ser substituídas pelas arquiteturas RISC, bem mais simples e rápidas. Enfim tínha-se tecnologia de firmware, com integração em escalas superiores e operações realizadas internas em picosegundos.

A CISC (do inglês, *Complex Instruction Set Computer*) é uma linha de arquitetura de processadores capaz de executar um grande conjunto de instruções complexas. Isso fez que a programação de computadores, com base nessa arquitetura, fosse mais fácil que em computadores com base em outros projetos. Por outro lado, a complexidade do conjunto de instruções tornou os circuitos da CPU e da unidade de controle bastante complicados. Para resolver essa questão, a programação é feita em dois níveis: uma instrução na linguagem de máquina não é executada diretamente pela CPU. A CPU realiza somente operações simples, chamadas micro-operações. Uma instrução complexa é transformada em um conjunto dessas operações simples que, então, são executadas pela CPU, o que torna a programação em Risc mais difícil e demorada que aquela realizada por outros projetos. A otimização desse tipo de arquitetura pode ocorrer com o armazenamento de número reduzido de operandos nos registradores. Nesse sentido, surgiram as arquiteturas RISC (ReducedInstruction Set Computer) na década de 1980. RISC apresentava CPUs incompatíveis com as outras do mesmo período, seus projetistas tinham liberdade para escolher novos conjuntos de instruções que maximizassem o desempenho total do sistema. Embora, a ênfase inicial estivesse dirigida a instruções simples, que podiam ser executadas rapidamente, logo percebeu-se que projetar instruções capazes de ser emitidas (iniciadas) com rapidez era a chave para um bom desempenho. O RISC manipulava um número relativamente pequeno de instruções disponíveis, em geral, cerca de 50, número muito menos expressivo que as 200 a 300 instruções de computadores como o VAX, da DEC, e os grandes mainframes da IBM.

A partir da máquina Intel i486, os processadores da Intel passaram a utilizar tanto a arquitetura RISC quanto a CISC. Eles contêm um núcleo RISC, que executa as instruções mais simples em um único ciclo do caminho de dados, enquanto interpreta as mais complicadas no modo CISC. O resultado é que as instruções comuns são rápidas e as menos comuns são lentas. Embora não tenha

a rapidez de um RISC puro, essa abordagem resulta em desempenho competitivo e permite que softwares antigos sejam executados sem precisar de modificação.

Na história, ainda é relevante mencionar modelos de computadores que marcaram a computação. Esses modelos históricos representam abstrações de arquiteturas de computadores projetados em diferentes épocas e que deixaram ideias e conceitos até hoje explorados nas atuais arquitetura dos computadores modernos. Os dois modelos de maior destaque são:

- o modelo de Turing, desenvolvido pelo matemático inglês Alan Turing em 1937. A Máquina Universal de Turing foi projetada para realizar qualquer cálculo se o programa apropriado for fornecido. Ela é considerada a primeira descrição de um computador moderno, pois é necessário apenas fornecer os dados de entrada e o programa – a descrição de como realizar o cálculo – para qualquer uma das máquinas. A inovação do modelo de Turing consistia na utilização de um programa, que tem a função de informar ao computador o que este deve fazer com os dados. Desse modo, os dados de saída passam a depender dos dados de entrada e do programa.
- o modelo de Von Neumann, que é a base da grande maioria dos computadores modernos dividida em quatro componentes: memória, unidade lógica e aritmética, E/S, unidade de controle. O modelo de Von Neumann determina que o programa deve ser armazenado na memória (retomando a ideia de programa de Turing e diferente da arquitetura dos primeiros computadores mecâninos com apenas dados armazenados na memória). Existem outros modelos diferentes e complementares ao modelo de Von Neumann, uma vez que esse modelo é sequencial e bastante funcional; contudo, seu projeto original impossibilita grandes velocidades de processamento.

Na **Semana 2**, ocorrem estudos com os elementos básicos distribuídos hierarquicamente no computador. Por essa razão, os elementos são detalhados por nível hierárquico, seus inter-relacionamentos e sua função. A estrutura de um computador apresenta esquematicamente quatro componentes:

- a unidade central de processamento ou CPU (do inglês, *Central Processing Unit*) controla o computador e faz processamento de dados;
- a memória principal armazena os dados;
- os periféricos de E/S (do inglês, *Input/Output* I/O) faz interface entre os dados, o computador e o meio externo;
- e as interconexões fazem a comunicação entre a CPU, memória principal e a E/S.

Em termos de funções, as funções básicas dos elementos de um computador são:

- processamento de dados para a manipulação dos dados em diferentes formas;
- armazenamento de dados para a retenção de dados a curto prazo para processamento imediato e a longo prazo para recuperação e atualização;
- Entrada/Saída (E/S) para o recebimento de dados do mundo exterior (Entrada) e sua devolução para o mundo exterior (Saída);
- controle para o comando das três funções anteriores, para a gerência dos recursos do computador e para a coordenação do desempenho das partes funcionais em resposta às instruções.

A CPU é o “cérebro” do computador e executa programas armazenados na memória principal, buscando, examinando e executando, em sequência, suas instruções. Especificamente, os componentes de uma CPU são:

- unidade de controle: controla a operação da CPU e, portanto, do computador. Ela comanda o funcionamento de cada subsistema sendo responsável por buscar instruções na memória principal e determinar seu tipo;
- unidade aritmética e lógica (ULA): efetua operações matemáticas para executar as instruções. Os dois aspectos mais importantes da aritmética computacional são o modo como os números são representados (o formato binário) e os algoritmos usados para as operações aritméticas básicas

(adição, subtração, multiplicação e divisão). Isso se aplica tanto para a aritmética de números inteiros quanto para a de números de ponto flutuante;

- registradores: oferece armazenamento interno à CPU. Registrador é uma pequena memória de alta velocidade utilizada para armazenar resultados temporários e para realizar algum controle de informações;
- interconexão da CPU: realiza a comunicação entre a unidade de controle, a ULA e os registradores. Os componentes de um computador são conectados por um barramento (fios paralelos que transmitem endereços, dados e sinais de controle entre a memória e os dispositivos de E/S, ou internos). Os computadores modernos que compramos contam com vários barramentos em suas placas e entre seus elementos.

Para executar as instruções do programa, a CPU utiliza ciclos de máquina. Um ciclo pode ter três fases:

- Busca: o sistema copia a próxima instrução no registrador de instruções da CPU. O endereço da instrução a ser copiada permanece no registrador contador de programa.
- Decodificação: a unidade de controle decodifica a instrução quando ela está no registrador de instrução. Dessa etapa resulta o código binário para uma operação que o sistema vai realizar.
- Execução: a ordem da tarefa é enviada pela unidade de controle para um componente da CPU. Por exemplo, a CPU pode dizer à ULA para somar o conteúdo de dois registradores de entrada e, depois, colocar o resultado em um registrador de saída.

A memória da CPU é composta de registradores, cada um deles com determinada função. Os registradores são locais de armazenamento rápido. Eles guardam dados temporariamente, os quais podem ser lidos e escritos em alta velocidade. Há diferentes tipos de registrados. Três tipos deles são:

- contador de programa: indica a próxima instrução a ser buscada para execução;
- registrador de instrução: mantém a instrução que está sendo executada;
- registrador de dados: mantém os dados de entrada, os resultados intermediários e os resultados finais.

A ULA efetua adição, subtração e outras operações simples sobre suas entradas, produzindo um resultado no registrador de saída. A ULA realiza três tipos de operação:

- Operações lógicas: tratam a entrada de dados e os resultados como padrões binários (geralmente AND, NOT, OR e XOR).
- Operações aritméticas: realizam as operações básicas da matemática (adição, subtração, divisão e multiplicação) e outras operações simples.
- Operações de deslocamento: são compostas por (i) deslocamento lógico que tem a função de deslocar padrões binários para a esquerda ou direita, e o (ii) deslocamento aritmético que é aplicado a números inteiros.

Como os demais componentes eletrônicos de um computador, a ULA é baseada em dispositivos lógicos digitais simples, capazes de armazenar dígitos binários e efetuar operações de lógica booleana.

A maioria dos computadores contém uma grande placa de circuito impresso denominada placa-mãe (do inglês, *motherboard*). A placa-mãe contém o chip da CPU, slots para os módulos de memória e vários chips de suporte. Ela apresenta também barramento ao longo do comprimento e soquetes, nos quais os conectores de borda das placas de E/S podem ser inseridos. Em geral, a CPU e a memória são conectadas por três grupos de barramentos:

- Barramento de dados: é composto de diversas linhas de conexão. Cada linha transporta 1 bit por vez. Desse modo, o número de linhas de conexão depende do tamanho das palavras utilizadas pelo computador. Uma palavra de 16 bits (2 bytes), por exemplo, necessita de um barramento de

dados com 16 linhas de conexão, de maneira que todos os 16 bits sejam transmitidos ao mesmo tempo.

- Barramento de endereços: permite que se acesse determinada palavra na memória. O número de linhas de conexão depende do espaço de endereçamento da memória. Portanto, se a memória tiver 2n palavras, o barramento de endereços precisa ter n conexões para poder transmitir n bits de cada vez.
- Barramento de controle: transporta a comunicação entre a memória e o CPU. O número de linhas de conexão depende da quantidade total de comandos de controle. Um computador com 2n ações de controle precisa que o barramento tenha n linhas de conexão. Assim, n bits podem definir 2n diferentes operações.

Memórias consistem em uma quantidade de células e cada uma delas pode armazenar uma informação. Cada célula tem um número, denominado endereço, pelo qual os programas podem se referir a ela. A unidade básica de memória é um dígito binário, denominado bit. Um bit pode conter um 0 ou um 1. Trata-se da unidade mais simples possível. Computadores que usam o sistema de números binários expressam endereços de memória como números binários. Se um endereço tiver m bits, o número máximo de células endereçaveis é 2m. A memória consiste em um conjunto de localizações de armazenamento, cada uma com um identificador único, chamado endereço (representado por números inteiros maiores que zero). O número total de localizações exclusivamente identificáveis na memória é chamado espaço de endereçamento. Se a memória tiver n células, elas vão ter endereços de 0 a (n – 1). Todas as células apresentam o mesmo número de bits.

As principais características das memórias são capacidade de armazenamento, tempo de acesso e custo. Essas características se relacionam entre si das seguintes formas:

- com tempo de acesso mais rápido, maior custo por bit;
- com maior capacidade, menor custo por bit;
- com maior capacidade, tempo de acesso mais lento.

Outras características das memórias podem ser consideradas como durabilidade e consumo de energia.

Os usuários de computador usualmente querem muita memória, de preferência muito rápida e de baixo custo. Porém, memórias velozes não são baratas. A solução tradicional para armazenar grandes quantidades de dados a preços acessíveis é lançar mão de uma hierarquia de memória com diferentes características para diferentes ocasiões. Algumas hierarquias de memória cruzam as características das necessidades com os tipos de memória. Outras hierarquias consideram apenas velocidade e colocam, no topo da pirâmide, os registradores (acessados à velocidade total da CPU), em seguida, a memória cache, que pode se usadas hierarquicamente com diferentes tempos de acesso, e a memória principal. Existem apresentações hierárquicas de memória que tem o cruzamento de memória com sua posição na estrutura do computador.

A escolha do uso de cada tipo de memória não pode ser uma escolha aleatória e as seguintes estratégias devem ser consideradas:

- explorar memórias de alta velocidade somente quando a velocidade é um fator fundamental;
- usar uma quantidade moderada de memória de média velocidade para armazenar dados que são acessados com frequência e, consequentemente, muita memória de baixa velocidade para dados que são acessados com pouca frequência.

A memória RAM (do inglês, *Random Access Memory*), simplesmente chamada de memória ou memória principal neste texto, caracteriza-se pelo fato de poder ser lida e escrita durante o processamento do computador. A memória cache é uma memória pequena e rápida cujo nome provém do francês *cacher*, que significa “esconder”. Ela é mais rápida do que a memória principal e mais lenta que os registradores e a CPU. A cache situa-se entre a CPU e a memória principal e pode

ser hierarquizada. A ideia básica de uma cache é simples: as células de memória utilizadas com mais frequência são mantidas nesse dispositivo. Quando a CPU precisa de uma informação, ela examina primeiro a cache. Somente se a palavra não estiver ali é que ela recorre à memória principal. Se uma fração substancial das palavras estiver na cache, o tempo médio de acesso pode ser muito reduzido. Visto que a memória RAM é muito mais lenta que os registradores da CPU, existe um overhead da comunicação pelo barramento. O uso da memória cache basicamente pretende obter velocidade de memória próxima das memórias mais rápidas, e disponibilizar uma memória de grande capacidade ao preço de memórias semicondutoras mais baratas.

Quando o processador tenta ler uma palavra da memória, é feita uma verificação para determinar se a palavra está na cache. Se estiver, ela é entregue ao processador. Caso não esteja, um bloco da memória principal, consistindo em algum número fixo de palavras, é lido para a cache; depois, a palavra é fornecida ao processador. A cache, então, armazena aqueles dados que são lidos com maior frequência, tornando, assim, o processamento mais rápido. Devido ao fenômeno de localidade de referência, quando um bloco de dados é levado para a cache para satisfazer uma única referência de memória, é provável que haja referências futuras a esse mesmo local da memória ou a outras palavras no mesmo bloco. O uso da cache é organizado em múltiplos níveis de cache denominados L1, L2 e L3. A cache L2 é mais lenta e maior que a cache L1, e a cache L3 é mais lenta e, em geral, maior que a cache L2.

A memória ROM (do inglês, *Read-Only Memory*) não pode ser alterada nem apagada. Os dados são inseridos durante sua fabricação. A única maneira de modificar o programa em uma ROM é substituindo o chip. Esse tipo de memória é utilizado em produtos cujos programa e dados básicos devem permanecer armazenados, mesmo quando o fornecimento de energia for interrompido, como carros, eletrodomésticos, brinquedos, e a placa-mãe para coordenar o processo de BOOT.

Outro tipo de memória presente nos computadores é a memória de massa, chamada também de memória secundária, memória de armazenamento ou apenas disco, como os discos rígidos ou HD (do inglês, *Hard Disk*) e outros dispositivos para armazenamento de longo período.

Os discos magnéticos podem ser flexíveis ou HD (giram de forma rápida, podendo alcançar 15 mil rotações por minuto). As velocidades mais comuns são 5.400 e 7.200 RPM. Os pratos desses discos são empilhados em um pino, e os cabeçotes de leitura/escrita são montados em um braço atuador giratório. O discos flexíveis giram em velocidades mais baixas do que os discos rígidos, em geral 300 ou 360 RPM e têm menor densidade, e organização e operação mais uniformes do que discos fixos.

Discos ópticos foram desenvolvidos inicialmente para gravar programas de televisão. Hoje são utilizados também para armazenar dados em computadores. Por sua grande capacidade e baixo preço, discos ópticos são usados para distribuir software, livros e dados, bem como para fazer *backup* de discos rígidos. Dois tipos de discos ópticos são CD-ROM (do inglês, *Compact Disc-Read Only Memory*) e DVD (do inglês, *Digital Versatile Disk*).

Sistemas de E/S movimentam dados codificados entre dispositivos externos e entre dois dispositivos do computador. A ideia de paralelismo e segurança motivou uma nova classe de dispositivos de E/S, denominados RAID (do inglês, *Redundant Array of Independent Disks*), que é um conjunto (arranjo) redundante de discos independentes para armazenamento de informações. A ideia fundamental de um RAID é instalar uma caixa cheia de discos, substituir a placa do controlador de disco por um controlador RAID, para copiar os dados para o RAID e, então, continuar a execução normal. Finalmente, terminais (teclado e monitor) de mainframes, uma grande variedade de teclados, diferentes tipos de mouses, diferentes monitores (LED, touchscreens etc.), impressoras, câmeras, e outros dispositivos de telecomunicações são outros exemplos de dispositivos de E/S.

Revisitando sistemas operacionais na **Semana 3**, obtém-se a definição de Sistema Operacional (SO) que "é o software que controla a execução de programas em um processador e gerencia os recursos do computador, realizando funções como o escalonamento de processos e o gerenciamento de memória". As funções do computador só podem ser executadas de modo rápido e eficiente se o SO dispuser de um suporte adequado do hardware do processador. Desse modo, quase todos os processadores dispõem desse suporte, em maior ou menor extensão, incluindo hardware de

gerenciamento de memória virtual e de gerenciamento de processos. Isso inclui registradores de propósito especial e áreas de armazenamento temporário, além de um conjunto de circuitos para realizar tarefas básicas de gerenciamento de recursos.

Além de gerenciar recursos, existem outras funções do SO, principalmente no que tange a interface entre usuários e o sistema. Nesse caso, o usuário vê o sistema de computação em termos de uma aplicação. No caso desta disciplina, a análise foca a arquitetura e a organização de computadores, então as tarefas do SO de mais interesse são escalonamento de processo e gerenciamento de memória.

No escalonamento de processos ou tarefas, o SO determina quais processos devem ser executados a cada instante. Tipicamente, o hardware interrompe periodicamente o processo que está sendo executado para permitir ao SO realizar uma nova decisão de escalonamento. Isso possibilita compartilhar o tempo do processador entre um determinado número de processos de modo imparcial. O SO faz também uma ação chamada de *swap* que é um chaveamento de processo entre memória e disco. O *swap in* ocorre do disco para memória e o *swap out* é da memória para o disco.

Existem alguns algoritmos específicos para a alocação de memória para processo: (i) melhor escolha (escolhe a área que mais encaixa em tamanho para o processo - *best fit*), (ii) pior escolha (*worst fit*) e (iii) primeira escolha (*first fit*).

Nos antigos mainframes e PCs, tinha-se um único processo executado no sistema por vez e isso era denominado de monoprogramação. Na multiprogramação há vários processos compartilhando um único processador. A multiprogramação exige a divisão da memória principal para manipular os diferentes processos com as seguintes abordagens:

- Particionamento fixo com tamanho igual;
- Particionamento fixo com tamanho variável;
- Particionamento variável ou dinâmico.

Programas cada vez maiores, maior grau de multiprogramação e execução de programas maiores que a memória motivaram a criação de técnicas para alocação de memória. Nesse contexto, memória virtual é uma técnica que usa a memória secundária como uma cache para partes de espaços dos processos. A maioria dos SO atuais inclui a capacidade de memória virtual, trazendo os seguintes benefícios: (i) um processo pode ser executado na memória principal sem que todas as instruções e dados do programa precisem estar armazenados na memória principal; e (ii) o espaço de memória total disponível para um programa pode exceder o tamanho da memória principal do sistema. Embora o gerenciamento de memória seja feito por software, o SO conta com suporte do hardware do processador, incluindo hardware de paginação e de segmentação da memória. Abordagens de memória virtual são:

- Paginação é quebra do tamanho do processo em páginas de tamanho fixo (mais ou menos 4KB). O endereçamento virtual é divido em páginas virtuais. Existe a mistura os tipos de dados na mesma página.
- Segmentação é a quebra em blocos de tamanho variáveis dividido por tipo texto, dado etc., para cada segmento. A segmentação permite lidar com blocos de tamanho variável, segmentos, cuja principal vantagem é definir características (proteções, detecção de acessos fora do segmento etc.) de forma adaptada ao tamanho de cada segmento que o programa define, em vez de forçar um tamanho fixo. No entanto, este mecanismo não é transparente para a programação e gera alguns problemas de implementação, portanto, acaba por ser menos geral do que a paginação. Também é possível combinar os dois, paginando segmentos grandes (segmentação paginada).

SO fazem uma mistura das duas técnicas para usufruir o benefício de cada. Os processos estão no disco, mas seu mapeamento vai para memória, mas a conversão é feita pelo MMU (do inglês, *Management Memory Unit*). MMU é um dispositivo de hardware (um circuito) que faz a tradução do endereço lógico (virtual ou lógico usado pelo programa) em físico (da memória de fato).

Página é uma unidade de tamanho fixo no dispositivo secundário. As páginas ficam armazenados no disco e nos frames na memória principal (RAM). A tabela de páginas é uma estrutura para mapear uma página ao frame correspondente. Cada processo tem a sua própria tabela de páginas, pois todos os processos podem acessar a mesma faixa de endereços virtuais e os espaços de endereçamento acessíveis aos processos são independentes. *Page fault* é a falta de página, quando a página não foi carregada na memória principal e está referenciada.

Para a tabela de páginas mapear as páginas virtuais em físicas deve haver um mecanismo de tradução entre endereço virtual e físico, usando os endereços virtuais que são compostos principalmente por bit de validade, número da página virtual vindo da tabela de páginas e o deslocamento. A busca antecipada ou por demanda de páginas na tabela de páginas pode ser binária, sequencial, ou mesmo combinando ambas as estratégias. Porém, existe *overhead* do gerenciamento de memória, sendo ideal que o endereço virtual na tabela de páginas seja um índice.

A fragmentação interna ocorre quando sobram espaços internos na memória decorrentes das com tamanho menores que podem gerar menos desperdício (menor fragmentação), porém a leitura é menos eficiente por causa da busca em tabelas grandes e mais tempo para carregar. As páginas maiores geram o oposto. A fragmentação externa é a soma de todas as fragmentações internas.

A tabela de páginas pode ser armazenada:

- na memória principal (RAM);
- no registrador;
- na cache (especificamente MMU).

O problema de armazenar a tabela de páginas na memória principal é gerar um acesso para tabela de página e outro para a memória propriamente dita. Uma solução é usar TLB (do inglês, *Translation Lookaside Buffer*) que é um mecanismo do tipo memória associativa que faz uma espécie uma cache da tabela de páginas. As páginas mais usadas num período de tempo ficam na TLB. Desse modo, primeiro busca-se na TLB, e depois vai na tabela de páginas levando um pouco mais de tempo. O conjunto de bits de saída da TLB, correspondente a cada número de página virtual de cada processo, é o seguinte: número de página física e bits de controle (válida, alterada etc.), para que a informação sobre a página possa ser usada quando a tradução de endereços é feita, sem acessar a memória.

Para busca rápida de informação com integração de memórias, tem-se a busca mais rápida dependente da disposição da cache e da memória principal. O caminho mais rápido é conseguido se o mapeamento virtual-físico estiver presente na TLB, a página referenciada estiver carregada em memória principal e o bloco acessado (dentro dessa página) estiver presente na cache.

Na **Semana 4**, o assunto estudado foi pipeline e arquiteturas paralelas. Uma CPU (ou processador) atende a ciclos de instrução para fornecer serviços, por exemplo, de aplicações. O ciclo de instrução, chamado de ciclo de busca e execução, coloca o processador em diferentes estados quando da execução de um ciclo de instrução. Na primeira ação – buscar – uma instrução na memória significa uma operação de leitura. Na segunda ação – executar – a ação significa interpretar e executar a operação, o que pode exigir leitura (voltar ao início) ou até termino. As etapas do ciclo de instrução se repetem indefinidamente até que o sistema seja desligado, ou ocorra algum tipo de erro, interrupção, ou seja encontrada uma instrução de parada. Além desse padrão de uso sequencial do processador, existem arquiteturas não von Neumann com vários processadores. A execução sequencial das ações do ciclo de instrução é lenta e pouco eficiente. Uma solução para esse problema é fazer paralelismo, isto é, fazer duas ou mais ações em paralelo, ou seja, ao mesmo tempo. Desse modo, a execução das tarefas ocorre, em geral, em menor tempo, por meio da execução em paralelo de diversas tarefas. O paralelismo pode ser obtido em hardware ou software, ele pode ser explorado dentro de instruções individuais para obter da máquina mais instruções por segundo (software) ou várias CPUs trabalhando juntas no mesmo problema (hardware).

Em termos de paralelismo no nível de instrução, o processo de buscar instruções na memória é um grande gargalo na velocidade de execução da instrução. Para amenizar esse problema, os computadores podem buscar instruções na memória antecipadamente (*prefetch*), de maneira que

elas estejam presentes quando necessárias. Essas instruções podem ser armazenadas em um conjunto de registradores (*buffers* de busca antecipada). Desse modo, sempre que necessária, uma instrução pode ser apanhada nesse *buffer*, em vez de aguardar a conclusão de uma leitura da memória. Na verdade, a busca antecipada divide a execução da instrução em duas partes: a busca e a execução propriamente dita. O conceito de pipeline amplia essa estratégia.

No pipelining, em vez de dividir a execução da instrução em apenas duas partes, muitas vezes ela é dividida em várias partes e cada uma delas é manipulada por uma parte dedicada do hardware, e todas elas podem ser executadas em paralelo. Em função do número de estágios do pipelining, o fator de aceleração se aproxima do número de instruções que podem ser introduzidas no pipeline sem desvio. Quanto maior o número de estágio no pipeline, maior a taxa de execução (aceleração ou *speedup*). Algumas situações que frustram essa proporcionalidade direta são: (i) o estágio de pipeline exige esforço extra para movimentar os dados e pode desacelerar o tempo total de execução (custo de implementação e atraso entre estágios), (ii) a lógica de controle da passagem entre estágios é mais complexa do que os estágios controlados.

Em pipelining, *hazards* de pipeline são situações que podem resultar em desempenho menor que a ótima. Eles ocorrem quando o pipeline, ou parte dele, precisa parar porque as condições não permitem a execução contínua. Os tipos de *hazard* e seus contextos são:

- *Hazard* de recurso (estrutural): duas ou mais instruções precisam do mesmo recurso;
- *Hazard* de dados: duas instruções (leitura ou escrita) querem acessar a mesma posição de operando;
- *Hazard* de controle (desvio): há necessidade de se tomar uma decisão com base nos resultados de uma instrução enquanto outras estão sendo executadas.

A solução para *hazard* é a busca antecipada (*prefetch*) de desvio, uso de *buffer* de laço de repetição (fica no *buffer* as últimas instruções recentemente lidas).

Para entender melhor, pode-se fazer uma analogia de pipeline com uma fábrica de bolos. Nessa fábrica, a operação de produção dos bolos e a operação da embalagem para expedição são separadas. Suponha que o departamento de expedição tenha uma longa esteira transportadora, ao longo da qual trabalham cinco funcionários (estágios de processamento). A cada dez segundos (ciclo de clock), o funcionário 1 coloca uma embalagem de bolo vazia na esteira. A caixa é transportada até o funcionário 2, que coloca um bolo dentro dela. Um pouco mais tarde, a caixa chega à estação do funcionário 3, na qual a embalagem é fechada e selada. Em seguida, prossegue até o funcionário 4, que coloca uma etiqueta na embalagem. Por fim, o funcionário 5 retira a caixa da esteira e a coloca em um grande contêiner, que mais tarde será despachado para um supermercado.

Em termos gerais, este é o modo como um pipeline de um computador funciona: cada instrução (bolo) passa por diversos estágios de processamento antes de aparecer concluída na extremidade final. Arquiteturas superescalares podem suportar mais de um pipeline. O termo “superescalar” descreve processadores que emitem múltiplas instruções – frequentemente, quatro ou seis – em um único ciclo de clock, e são arquiteturas paralelas.

Além de pipelining, a busca por computadores cada vez mais rápidos parece não terminar. Paralelismo no nível de instrução aprimora-os, mas pipelining raramente rende mais que um fator de 5 ou 10 vezes que a execução sequencial. Para obter ganhos de 50, 100 ou mais, a única maneira é projetar computadores com várias CPUs em arquiteturas paralelas. Existem diversas classificações para arquiteturas paralelas devido à constante evolução. A clássica classificação de Flynn (1972) baseia-se na unicidade e multiplicidade do fluxo de dados e instruções do seguinte modo:

- SISD (do inglês, *Single Instruction Stream, Single Data Stream* - único fluxo de instrução, único fluxo de dados) são os computadores convencionais como os de nossas residências (seriais de von Neumann), com instruções executadas serialmente, porém os estágios (busca da instrução, decodificação, busca do operando e execução) podem ser sobrepostos e usualmente empregam pipeline.

- MISD (do inglês, *Multiple Instruction Stream, Single Data Stream* - múltiplo fluxo de instruções, único fluxo de dados) são vários processadores que recebem instruções distintas, mas operam sobre o mesmo conjunto de dados. Ex.: múltiplos algoritmos de criptografia para decodificar uma mensagem.
- SIMD (do inglês, *Single Instruction Stream, Multiple Data Stream*, único fluxo de instruções, múltiplo fluxo de dados) são processadores matriciais, paralelos e associativos com uma única unidade de controle que envia um fluxo de instruções para vários processadores. Os processadores recebem a mesma instrução simultâneamente e atuam sobre diferentes fluxos de dado. Ex.: processadores vetoriais.
- MIMD (do inglês, *Multiple Instruction Stream, Multiple Data Stream*, múltiplo fluxo de instruções, múltiplo fluxo de dados) são vários processadores, cada um controlado por uma unidade de controle. Os processadores recebem instruções diferentes e operam sob fluxos de dados diferentes, que podem ser síncronos ou assíncronos. Ex: multiprocessadores (memória compartilhada) e multicomputadores (memória distribuída).

Multiprocessadores (MP) e multicomputadores (MC) são semelhantes quando se trata de interconexão, pois ambos trocam mensagem. A diferença fundamental entre ambos é a presença ou a ausência de memória compartilhada. Enfim, multiprocessador é um computador paralelo com CPUs que compartilham memória com um único espaço de endereço virtual mapeado para memória comum. Multicomputadores são vários computadores com suas CPUs e memórias privadas.

Na **Semana 5**, o estudo é focado em linguagem C, uma vez que ponteiros podem eficientemente manipular a memória. Ponteiro é uma variável que contém um endereço de memória, que é a posição de outra variável na memória. Diz-se que um ponteiro "aponta" para uma variável. A forma geral de declarar um ponteiro é <*tipo * identificador*>, sendo *tipo* qualquer modelo de dado válido em C, *identificador* como qualquer identificador válido em C, * como o símbolo para declaração de ponteiro. Alguns exemplos de declarações de ponteiros são: *int *p*; *char *p1*; *float *pf1, *pf2*. Os operadores e as operações com ponteiros são:

- & é um operador unário que devolve o endereço de memória do seu operando. Seu uso mais comum é durante inicializações de ponteiros. Um exemplo é:

```
int *p, acm = 35;
p = &acm; // p recebe "o endereço de" acm
```

- * é um operador unário que devolve o valor da variável apontada (o conteúdo do apontador). Um exemplo é:

```
int *p, q, acm = 35;
p = &acm;
q = *p; /* a variável q recebe o valor da variável "no endereço" p  */
```

- ++ e -- são operadores unários que caminham para frente (ou retrocede) o ponteiro na quantidade de bytes correspondente ao tipo base (incrementa o endereço do ponteiro).

A comparação entre ponteiros é possível, pois compara-se as posições de memória.

Em linguagem C, também é útil aprender *typedef* e *struct* para manipular variáveis e ponteiros, e simular ações como escalonamento e paginação. *typedef* permite compor novos tipos de dados, a partir de tipos pré-existentes (não cria um novo tipo de dado). A forma geral de declaração é <*typedef tipo novo_nome*>.

*strucut* é uma coleção de variáveis, possivelmente de diferentes tipos, organizadas em um único conjunto. As variáveis que compõem uma estrutura são comumente chamadas de elementos (ou campos). A definição é, por exemplo:

```
struct pessoa{
        char nome[30];
        int idade;
};
```

Para entender melhor o funcionamento de um computador, é interessante sabe como ele trata as informações. Nesse contexto, a **Semana 6** apresenta linguagens de montagem e de máquina, as quais tornam possível um computador entender o que se escreve em linguagem de alto nível como Ruby, Java, Phyton, C etc. Essas linguagens de comunicação mais direta com o computador são chamadas de linguagem de baixo nível, pois elas têm menor nível de abstração do que as linguagens de programação tradicionais. Linguagens de montagem manipulam representações simbólicas (SUB, ADD, MUL) e operandos com registradores (EAX, EDX), e linguagem de máquina manipula números binários ou hexadecimais.

A linguagem de montagem usa muitos nomes simbólicos (mnemónicos), incluindo a atribuição de nomes em posições específicas da memória principal e das instruções para operar o computador (Ex.: Assembly). Ela é composta também de instruções que não são executadas diretamente, mas que são úteis para o montador produzir o código de máquina.

A linguagem de máquina descreve instruções executadas diretamente pelo processador. Cada instrução é uma cadeia binária (0 e 1) que contém um opcode (código de operação básica da instrução), referências a operandos e, possivelmente, bits relacionados à execução.

Cada declaração de uma linguagem de montagem produz uma instrução de máquina. A linguagem de montagem é mais fácil de programar do que a linguagem de máquina, sendo que, em geral, é mais fácil lembrar de nomes e de endereços simbólicos em vez de binários ou hexadecimais. O programador de linguagem de montagem precisa se lembrar apenas dos nomes simbólicos porque o montador os traduz para instruções de máquina. O programador de linguagem de máquina deve sempre trabalhar com os valores numéricos dos endereços. As linguagens de montagem são úteis para:

- Localizar erros, analisando um código em linguagem de montagem gerado pelo compilador, ou a janela de montagem em um depurador;
- Criar compiladores, depuradores e outras ferramentas de desenvolvimento;
- Manipular sistemas embarcados, pois têm menos recursos do que PCs e mainframes;
- Construir drivers para hardware e códigos de sistemas que precisam acessar hardware, registradores de controle do sistema etc.;
- Acessar instruções que não são acessíveis a partir das linguagens de alto nível;
- Otimizar o tamanho do código para caber em uma cache;
- Otimizar código em desempenho checando a otimização feita pelo compilador;
- Compatibilizar bibliotecas de funções com compiladores e sistemas operacionais.

A estrutura de sentença da linguagem de montagem é composta por quatro elementos: rótulo, mnemônico, operando(s) e comentário. Rótulo é o identificador da instrução ou de uma constante, que são usados com mais frequência em instruções de desvio. Um exemplo é:

```
L2: SUB EAX, EDX
JG L2
```

Um rótulo possibilita que uma posição do programa seja mais fácil de ser localizada e lembrada. Mnemônico é o nome da operação ou função da sentença da linguagem de montagem. Ele serve para identificar uma operação ou função e pode corresponder a uma instrução de máquina (associado a determinado opcode), uma diretiva do montador ou uma macro. Operandos servem para

especificar dados necessários à operação. Uma sentença da linguagem de montagem inclui zero ou mais operandos. Cada operando identifica um tipo de referência: um valor imediato, um registrador ou uma posição de memória. A linguagem de montagem fornece convenções para: (i) distinguir os três tipos de referências de operandos, e (ii) indicar o modo de endereçamento.

Comentários começam com um caractere especial sinalizando para o montador que o restante da linha é um comentário e, por isso, deve ser ignorado pelo montador. Todas as linguagens de montagem permitem a inserção de comentários dentro do programa. O comentário pode estar do lado direito de um comando em linguagem ou ocupar uma linha inteira de texto. A Sentença-comentário consiste inteiramente de um comentário.

Instrução é o tipo de sentença de representações simbólicas de instruções de linguagem de máquina. As sentenças diretivas (pseudoinstruções) fazem parte da linguagem de montagem que não são diretamente traduzidas para instruções da linguagem de máquina e, portanto, não são executáveis. E finalmente, macro (ou sub-rotina) é uma seção do programa que pode ser usada diversas vezes, sendo chamada a partir de qualquer ponto do programa. Elas são consideradas pelo montador em tempo de montagem.

Quando se quer otimizar recursos e rastrear problemas recursos dos sistemas computacionais, na maioria das situações, os recursos disponibilizados por linguagens de alto nível são insuficientes. Nesses casos, usar linguagem de montagem e de máquina é fundamental para ativar a máquina via interrupções ou chamadas de sistema. O mecanismo de interrupção promove o compartilhamento de tempo da CPU entre os vários programas e os diferentes dispositivos periféricos num sistema multiprogramado. Enfim, interrupção constitui a base para a implementação desse esquema de paralelismo para compartilhamento entre CPU e periféricos. Por exemplo, para controlar E/S de dados, não é interessante que a CPU tenha que ficar continuamente monitorando e status de dispositivos como discos ou teclados. O mecanismo de interrupções permite que o hardware "chame a atenção" da CPU quando há algo a ser feito. Alguns tipos de interrupção são:

- interrupção de programa (como *traps*);
- interrupção de temporização (escalonamento de processo);
- falha de hardware (como falta de energia);
- interrupção de E/S (como fim de escrita no disco).

Quando ocorre uma interrupção, a CPU pára o processamento do programa em execução e executa um pedaço de código chamado de tratador de interrupção. Em muitos casos, após a execução do tratador, a CPU volta a executar o programa interrompido. O tratador de interrupções é feito por um bloco especial de código associado a uma condição de interrupção específica. Ele é chamado também de rotina de serviço de interrupção (ISR ou *handler*). Os *handlers* de interrupção são iniciados por interrupções de hardware, instruções de interrupção de software ou exceções de software. Ele é usado para implementar *drivers* de dispositivo ou transições entre modos protegidos de operação, como chamadas de sistema. A função do tratador de interrupções é (i) saber qual dispositivo lançou a interrupção, (ii) em que ponto do sistema operacional está o endereço inicial da rotina que trata esta interrupção para que o processador possa executar essa rotina no endereço especificado pelo tratador e, (iii) deixar o processador voltar ao seu curso de execução após a interrupção ter sido resolvida.

Em termos de fluxo de controle, a execução do tratador é um pouco como uma chamada de rotina, mas a chamada de rotina é iniciada por instruções do programa em execução e o tratador de interrupção é na maioria das vezes assíncrono (sem comunicação entre o programa interrompido e o tratador), mas existe interrupção síncrona, como o caso do *trap*.

Finalmente, chamada de sistema é o mecanismo pelo qual um programa de usuário solicita um serviço do sistema operacional. Por exemplo, para acessar o disco rígido, a criação e a execução de novos processos e comunicação com serviço do SO como escalonamento de processo. Chamada de sistema é uma interface essencial entre um processo e o SO.

A **Semana 7** aplica uma abordagem prática para exercitar a linguagem Assembly (ASM). Em linguagem C e em outras linguagens de programação, sabe-se que uma função tem argumentos. A declaração de instruções em Assembly é bem similar, mas o correto é dizer que a função é um opcode (código simbólico) e cada argumento vai ser chamada de operando. Os operandos podem ser registradores, endereços de memória (espaço na memória) e constantes (valores numéricos).

As quatro principais instruções ASM usam MOVE, ADD, SUB e JUMP. Em ADD ou SUB, as operações aritméticas de soma e subtração são realizadas, respectivamente. O primeiro operando é o registrador é o resultado na operação e o segundo operando vai ser o valor ou o registrador a ser somado ou subtraído do 1º operando. Em JUMP, o primeiro operando será a condição e o 2º operando é a localização para aonde pular. Um exemplo de comandos em ASM está na figura a seguir:

**Figura 7.1** Cálculo de *N = I + J* no x86.

| Etiqueta | *Opcode* | Operandos | Comentários |
|---|---|---|---|
| FORMULA: | MOV | EAX,I | ; registrador EAX = I |
| | ADD | EAX,J | ; registrador EAX = I + J |
| | MOV | N,EAX | ; N = I + J |
| | | | |
| I | DD | 3 | ; reserve 4 bytes com valor inicial 3 |
| J | DD | 4 | ; reserve 4 bytes com valor inicial 4 |
| N | DD | 0 | ; reserve 4 bytes com valor inicial 0 |

O exemplo tem 4 etiquetas ou rótulos I, J e N. O montador MASM requer dois pontos em rotulos de código, mas não requer nos dados. Outros assemblers ou montadores podem ser diferentes. Cada máquina tem alguns registradores, de modo que precisam de nomes. Os registradores de dados x86 tem nomes como EAX, EBX ECX e assim por diante. O campo *opcode* contém ou uma abreviatura simbólica para ele – se a declaração for uma representação simbólica para uma instrução de máquina – ou um comando para o próprio montador. A escolha de um nome adequado é apenas uma questão de gosto, e diferentes projestistas de linguagem de montagem muitas vezes fazem escolhas diferentes. Os projetistas do assembler MASM decidiram usar MOV para carregar o registrador, a partir da memória e para armazenar um registrador, mas poderiam ter escolhido MOVE ou LOAD e STORE. Os programas de montagem, em geral, precisam reservar espaço para variáveis. Os projetistas da linguagem de montagem MASM escolheram DD (*Define Double*), já que uma palavra no 8088 tinha 16 bits.

Usando um recurso do compilador C ou C++, o especificador *in-line,* é possível combinar comandos da linguagem C com comandos da linguagem Assembly. Os *in-line* instruem o compilador a inserir uma cópia do corpo da função em cada lugar em que a função é chamada. Sua estrutura básica é:

```
_asm_ ou asm(
"instrução\n” %assembly code
"Outra instrução” %assembly code
)
```

Em arquiteturas diferentes, existem diferentes sintaxes ASM. Na montagem *in-line* básica, tem-se apenas instruções. Na montagem estendida, também pode-se especificar os operandos. Ela nos permite especificar os registradores de entrada, registradores de saída e uma lista de registradores substituídos. Não é obrigatório especificar os registros a serem usados, pode-se deixar isso para o compilador C e isso provavelmente se encaixa melhor no esquema de otimização do GCC. A estrutura ASM estendida é:

```
asm (
```

```
assembler template
   : output operands               /* optional */
   : input operands                /* optional */
   : list of clobbered registers   /* optional */
   );
```

A estrutura estendida consiste em instruções de montagem. Cada operando é descrito por uma string de restrição de operando seguida pela expressão C entre parênteses. Dois pontos separam o template assembler do primeiro operando de saída e outro separa o último operando de saída da primeira entrada, se houver. As vírgulas separam os operandos dentro de cada grupo. O número total de operandos é limitado a dez ou ao número máximo de operandos em qualquer padrão de instrução na descrição da máquina, o que for maior. Se não houver operandos de saída, mas houver operandos de entrada, deve-se colocar dois dois-pontos consecutivos ao redor do local onde os operandos de saída iriam. Um exemplo estendido é:

```
int a=10, b;
asm ("movl %1, %%eax;
     movl %%eax, %0;"
    :"=r"(b)       /* output */
    :"r"(a)        /* input */
    :"%eax"        /* clobbered register */
  );
```

Esse exemplo converte o valor de 'b' declarado fora no ASM *in-line* igual ao de 'a', usando as instruções de montagem dentro do asm().

**FONTES:**


CORRÊA, A. G. D. **Organização e arquitetura de computadores**. Bibliografia Universitária Pearson. 1. ed. São Paulo: Pearson, 2017. ISBN: 9788543020327.

DAMAS, L. **Linguagem C**. São Paulo: Grupo GEN, 2006. VBID: 9788521632474.

DELGADO, J.; RIBEIRO, C. **Arquitetura de Computadores**. 5. ed. São Paulo: Grupo GEN, 2017. ISBN: 9788521633921.

FLYNN. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, v. C-21, p. 948-960, 1972.

KELLEY, A.; POHL, I. **A Book on C**. 4. ed., c. 1-2; 4. Boston: Addison-Wesley, 1998.

SCHILDT, H. C. **Completo e Total**. c. 2. São Paulo: Makron Books, 1997.

SOFFNER, R. **Algoritmos e Programação em Linguagem C**. 1. ed. São Paulo: Saraiva, 2013. VBID: 9788502207530.

STALLINGS, W. **Arquitetura e organização de computadores**. 10. ed. São Paulo: Pearson, 2017. ISBN: 9788543020532.

TANENBAUM, A.; AUSTIN, T. **Organização estruturada de computadores**. 6. ed. São Paulo: Pearson, 2013. ISBN: 9788581435398.