Unidade II

3 ORGANIZAÇÃO DE SISTEMAS COMPUTACIONAIS

3.1 Unidade central de processamento (CPU)

O processador é considerado o "cérebro" do computador, controlando suas tarefas, como processar, gravar ou interpretar dados e/ou instruções, operando sobre números binários (0 e 1).

Figura 45 – Unidade central de processamento (CPU) Intel Core i7

Lembrete

A CPU é constituída por diferentes partes distintas em unidade de controle, responsável pela busca e decodificação de instruções e pela unidade lógica e aritmética (ULA), que efetua operações lógicas (AND, OR, NOT, NAND, NOR, XOR, XNOR), assim como operações aritméticas, como adição e subtração.

3.1.1 Processo de fabricação da CPU

As CPUs são empacotadas em um *chip* multiprocessado constituído por um núcleo de silício ligado a um conjunto de pinos capazes de realizar toda a comunicação de dados/instruções eletronicamente. A figura a seguir mostra parte do processo de manufatura de um processador a partir de um *wafer* até a impressão litográfica de portas lógicas no *chip*.

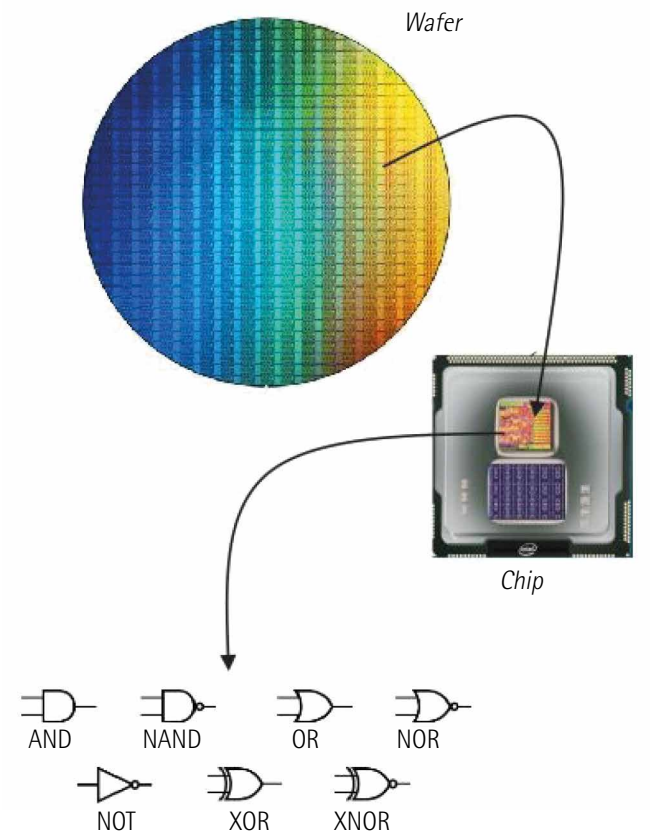


Figura 46 – Etapas do processo de fabricação de um *chip*

O processo para a fabricação de um processador envolve muitos estágios, desde a obtenção do silício em um alto grau de pureza até o estágio final de litografia (impressão de circuitos eletrônicos e portas lógicas) e empacotamento do processador. Inicialmente, o silício em estado sólido é triturado até ficar reduzido ao tamanho de pequenos grãos.

Figura 47 – Pedaço de silício com alto grau de pureza

Após a obtenção do silício em pó, ele é inserido em uma máquina de crescimento epitaxial para que possa ser aquecido a aproximadamente 1.000 °C e, através de um processo de rotação, seja possível obter uma pré-forma ou "tarugo" de silício.

Figura 48 – Máquina de crescimento epitaxial

Essa pré-forma é então fatiada transversalmente, facilitando o próximo estágio da fabricação, que será a litografia. Na sequência, é possível observar o próximo estágio de fabricação do processador, em que se retira um pequeno pedaço do *wafer*, que será empacotado num *chip*.

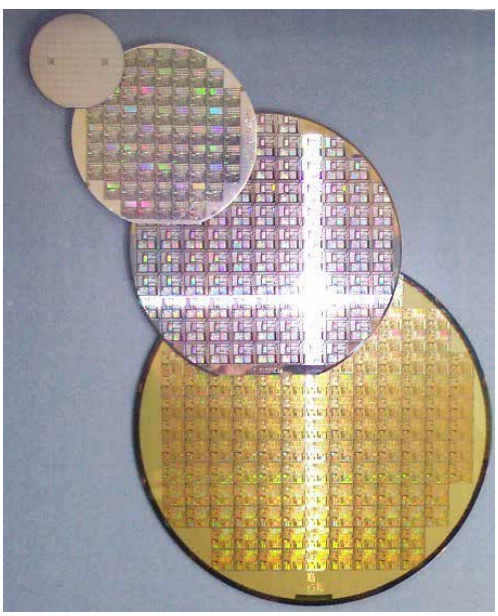


Figura 49 – Vários processadores em diferentes tamanhos de *wafer*

Saiba mais

Aprofunde seus conhecimentos sobre os avanços na fabricação de novos processadores:

NANOFABRICAÇÃO: refazendo a matéria átomo por átomo. *Inovação Tecnológica*, 1º ago. 2016. Disponível em: https://bit.ly/3rCZbPl. Acesso em: 2 fev. 2021.

3.1.2 *Chips* de CPU

Um *chip* de CPU é basicamente dividido em três tipos: endereço, dados e controle. Dessa forma, eles poderão ser conectados a pinos similares na memória e a *chips* de E/S através de um conjunto de fios paralelos, também conhecidos como barramento. Ao buscar uma instrução na memória, inicialmente a CPU coloca o endereço da posição de memória desejada daquela instrução em seus pinos de endereços. Após esse primeiro passo, ela ativa uma ou mais linhas de controle com a finalidade de informar à memória que ela necessita ler uma palavra. Daí a memória responde colocando a palavra requisitada nos pinos de dados da CPU e ativa um sinal que informará o que acabou de ser realizado. Ao receber esse sinal, a CPU aceita a palavra e executa a instrução solicitada. A instrução solicitada pode ser uma requisição para realizar leitura ou escrita de palavra de dados, em que todo o processo deverá ser repetido para cada palavra adicional processada.

Nos *chips* de CPU, existem dois parâmetros fundamentais que determinam seu desempenho, que são a quantidade de pinos de endereços e o número de pinos para dados. Um *chip* com *m* pinos de endereços pode realizar o endereçamento de até 2^m localizações de memória. Os valores comuns do número de pinos *m* são 16, 32 e 64 (TANENBAUM; AUSTIN, 2013). De modo semelhante, um *chip* consistindo em *n* pinos de dados pode ler ou escrever uma palavra de n bits em uma única operação. Dessa forma, um *chip* de 64 pinos de dados será muito rápido, porém com um custo muito mais caro.

Além dos pinos para endereçamento de dados, a CPU também possui alguns pinos de controle. Os pinos de controle servem para regular o fluxo e a temporização de dados oriundos da CPU e que passam por ela, além de outras funcionalidades. As CPUs também possuem pinos para alimentação de energia elétrica (entre 1,2 volt a 1,5 volt), um pino de aterramento e um pino para o sinal de sincronismo de *clock* (onda quadrada de frequência bem definida). Existem outros diversos pinos que também realizam o controle e que podem ser agrupados nas seguintes categorias:

- Controle de barramento.
- Interrupções.
- Arbitragem de barramento.
- Sinalização de coprocessador.

- Estado.
- Diversos.

A figura a seguir mostra um *chip* de CPU genérico baseado no uso desses grupos de pinos de sinais.

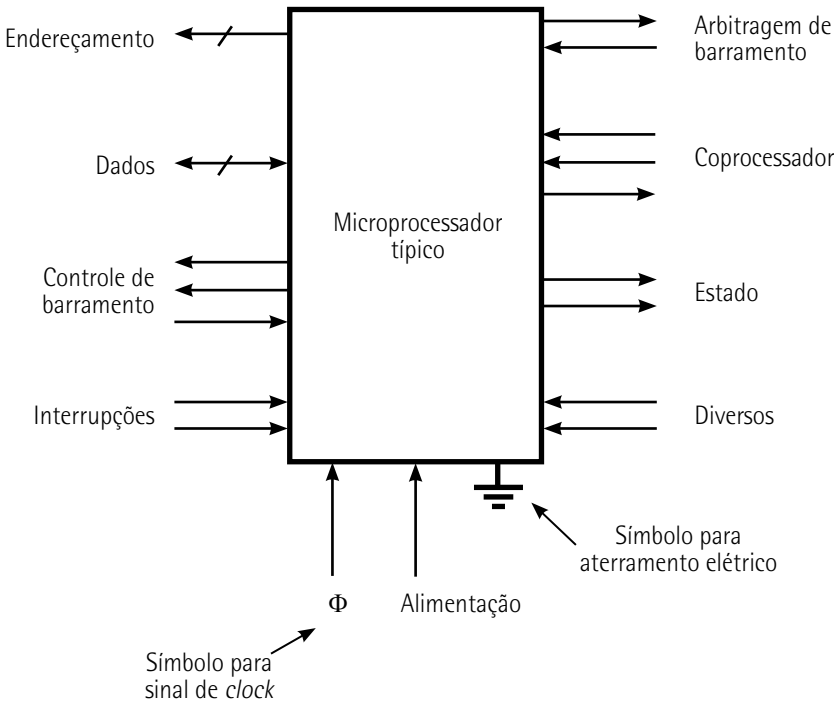


Figura 50 – Pinagem lógica de uma CPU genérica

Como observado na figura, as setas indicam sinais de entrada e sinais de saída, e os segmentos de reta diagonais indicam que são utilizados vários pinos para realizar a comunicação de dados e instruções. Também é possível observar a pinagem para alimentação elétrica, aterramento e entrada de sincronismo de *clock*.

Observação

A maioria dos pinos de controle do barramento são na verdade saídas de comunicação da CPU para o barramento; portanto, são linhas de comunicação para E/S e para memória principal.

Outros pinos como os de interrupção são entradas oriundas dos dispositivos de E/S que se conectam à CPU. A principal função nesse caso é a de controlar algum dispositivo de E/S, ativando um sinal dos pinos para interromper a CPU ou fazê-la realizar algum tipo de serviço para o dispositivo de E/S, por exemplo a verificação de ocorrências de erros de comunicação de E/S. Algumas CPUs, como a observada na figura anterior, possuem um pino de saída para confirmação do sinal de interrupção. Os pinos de arbitragem de barramento são utilizados para realizar o controle do tráfego no barramento, de modo a impedir

que dois ou mais dispositivos tentem usá-lo simultaneamente. Outros *chips* de CPU são projetados para funcionar como processadores auxiliares ou coprocessadores, como os de ponto flutuante, como visto na figura a seguir, além de outros coprocessadores especializados em processamento gráfico.

3.1.3 Intel Core i7

O processador Intel Core i7, lançado na sua primeira geração em 2008, é uma evolução da CPU Intel 8088 com 29 mil transistores utilizada no *desktop* IBM PC. Em 2008, o i7 possuía cerca de 731 milhões de transistores distribuídos em quatro processadores operando em uma frequência de *clock* de 3,2 GHz (3,2 bilhões de hertz ou ciclos por segundo), utilizando uma largura de linha (fios de cobre de ligação entre os transistores) de 45 nanômetros (45×10^{-9} metros). Quanto menor for a largura de linha, o que também implica transistores menores, mais transistores são empacotados no *chip*, aumentando assim o poder de processamento da CPU. A primeira geração da arquitetura i7 era baseada na arquitetura Nahalem, que na época substituiu a arquitetura Core e evoluiu para uma arquitetura Sandy Bridge, composta por 1,16 bilhão de transistores, trabalhando a uma velocidade de até 3,5 GHz, com largura de linha de 32 nanômetros (TANENBAUM; AUSTIN, 2013). Outra diferença com os processadores mais antigos é que o i7 é uma máquina completa de 64 bits (tamanho da palavra), *multicore* (múltiplos núcleos), vendida com um número variável de núcleos que vão de 2 a 6.

O i7 também é compatível para uso de processamento paralelo e *threads* de *hardware* e *software*. O sistema de *hyper-threading*, também conhecido como *multithreading* simultâneo, permite que latências muito curtas (quantidade de tempo para acesso a outro *hardware*) sejam utilizadas quando há falta de memória *cache*.

Além disso, todos os processadores Core i7 possuem três níveis de memória *cache* (L1, L2 e L3), em que cada núcleo possui uma *cache* de dados L1 (*layer* 1) com 32 KB (kilobytes) e uma *cache* L1 de instruções também com 32 KB. Os núcleos dos processadores i7 também possuem sua própria *cache* L2 com 256 KB, além de compartilharem entre todos os núcleos uma unificada L3, com tamanhos que variam de 4 a 15 MB (megabytes).

Como todos os processadores Core i7 possuem múltiplos núcleos com *caches* específicas para dados, ocorre um problema quando a CPU modifica a palavra na *cache* privada que também esteja contida em outro núcleo. Se outro processador tenta ler a mesma palavra da memória, poderá obter um valor ultrapassado, visto que as palavras de *cache* modificadas não são escritas imediatamente de volta na memória. Assim, para manter uma consistência de informações da memória, cada núcleo em um sistema microprocessado realiza a operação *snoops* (escuta) no barramento de memória, em busca de referências de palavras contidas na *cache*. Quando ela acha uma dessas referências, são fornecidos os dados antes que a memória faça a leitura deles.

Outro problema encontrado no i7 é referente ao consumo de energia, que também ocorre com outras CPUs. Um processador Core i7 consome entre 17 a 150 watts, dependendo de seu modelo ou frequência de operação. A fim de impedir que algum dano ocorra no silício devido ao aquecimento excessivo, em todo projeto de processadores, a Intel utiliza métodos de resfriamento e de dissipadores de calor, para que o calor fique afastado do substrato do processador e o silício queime. O projeto do Core i7 também

possui um soquete de conexões *land grid array* (LGA) quadrado com 37,5 milímetros de borda e 1.155 pinos em sua parte inferior, em que 286 são para alimentação e 360 para aterramento. Os pinos estão dispostos como um quadrado de 40 x 40 pinos com 17 x 25 pinos faltando no meio. Outros 20 pinos também estão faltando no perímetro de forma assimétrica, para impedir que o *chip* seja inserido de forma incorreta na sua base. Essa disposição física da pinagem é observada na figura a seguir.

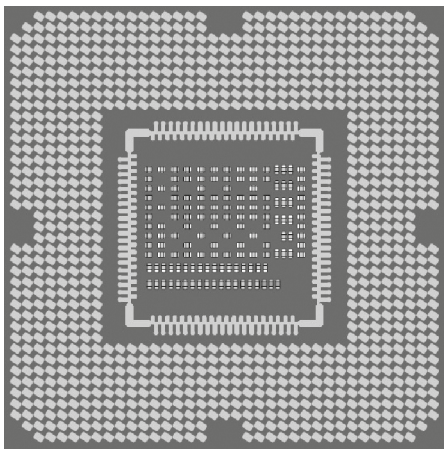


Figura 51 – Disposição física dos pinos do Intel Core i7

A pinagem lógica de funcionamento é mostrada na figura a seguir, em que é possível observar no seu lado esquerdo cinco grupos principais de sinais de barramento, incluindo comunicações com a memória principal através de comunicação *double data rate* (DDR); e no seu lado direito, uma diversidade de sinais, para diagnósticos, monitoramento térmico, detecção e gerenciamento de energia etc.

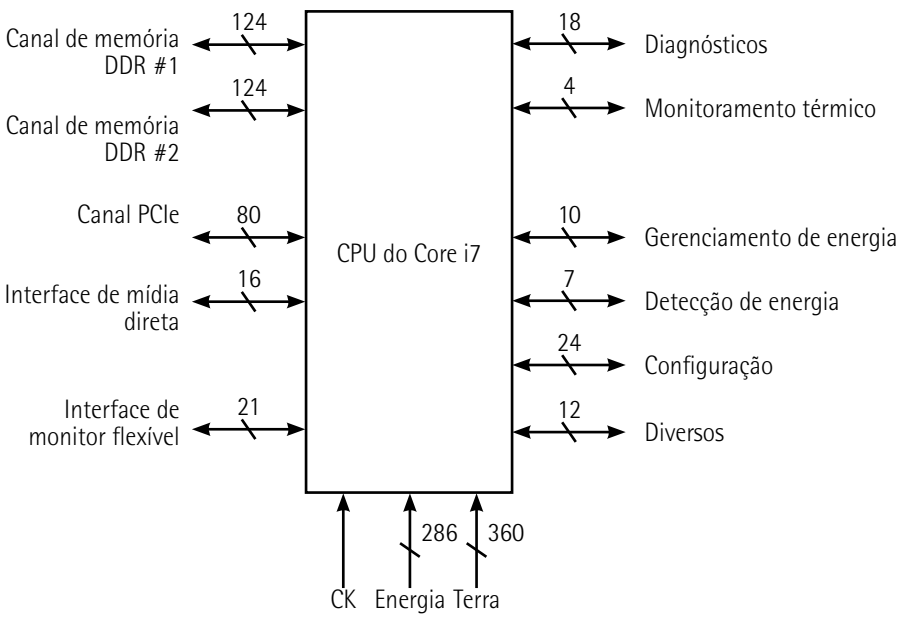


Figura 52 – Pinagem lógica do Intel Core i7

Depois de estudar a parte externa de um processador com a discussão de sua pinagem lógica e física, agora é necessário abordar as características e o funcionamento interno de um processador.

3.1.4 Organização geral de um processador

O processador pode ser organizado de forma funcional em categorias:

- **Buscar instruções**: o processador deve ser capaz de realizar a leitura de uma ou várias instruções que ocupam endereços específicos na memória (registrador, *cache*, memória principal).
- **Interpretar instruções**: as instruções são decodificadas a fim de se determinar qual ação é requerida naquele momento.
- **Obter dados**: o ato da execução de uma ou mais instruções pode requerer a leitura de dados da memória ou algum dispositivo de E/S.
- **Processar dados**: a execução de instruções pode requerer a realização de operações lógicas e/ou aritméticas com os dados.
- **Gravar dados**: os resultados obtidos através do processamento de dados podem ser gravados na memória ou dispositivos de E/S.

3.1.5 Microarquitetura de processadores

A microarquitetura de um processador é constituída por um conjunto de instruções contendo operações complexas, além de três subsistemas principais: unidades de execução, banco de registradores e lógica de controle. Essas unidades também podem ser descritas como o caminho dos dados do processador, pois os dados e instruções fluem regularmente por eles (CARTER, 2002).

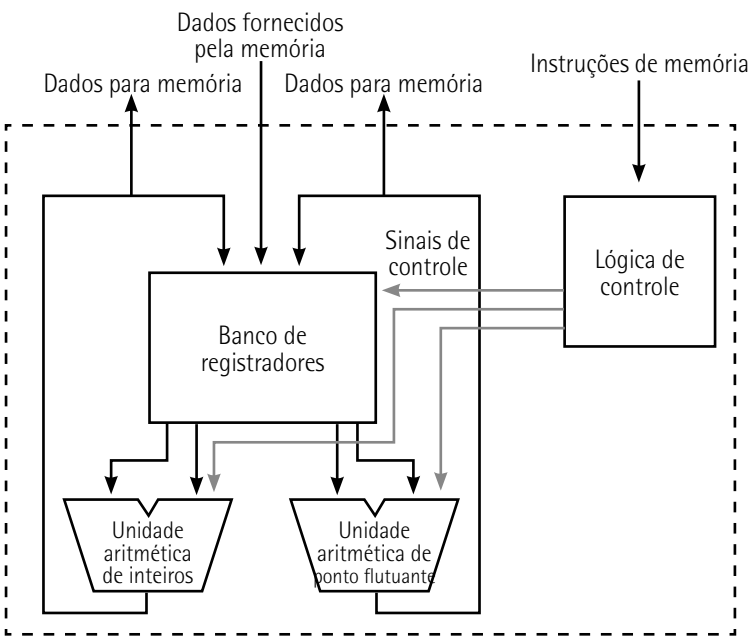


Figura 53 – Diagrama de blocos de um processador moderno

O diagrama da figura anterior exemplifica quais são as etapas envolvidas na execução de uma ou mais instruções e como os módulos do processador interagem entre si durante o processamento e decodificação da instrução. Inicialmente, o processador busca a instrução que está contida na memória principal, externa ao processador, e a direciona para ser decodificada pela lógica de controle. Após a decodificação da instrução, ela é representada em um padrão de *bits* a fim de informar ao *hardware* como ela deverá ser executada. Esse padrão de *bits* é então enviado para a próxima sessão da unidade de execução da instrução, através de sinais de controle, que lê as entradas da instrução na memória interna do processador (banco de registradores). Assim que a instrução é decodificada e os valores são armazenados nos registradores, as instruções são executadas em uma unidade lógica aritmética de inteiros e/ou ponto flutuante, a fim de se obter resposta ao processamento desejado. Por fim, os dados obtidos são devolvidos como resposta à memória principal do computador, primeiro passando pelo banco de registradores através do barramento interno do processador.

3.1.6 Unidade lógica e aritmética (ULA)

A ULA é o principal dispositivo do processador e realiza efetivamente as operações lógicas/matemáticas sobre dados e instruções.

Qualquer ULA é considerada um conjunto de circuitos lógicos e componentes eletrônicos simples que, ao se integrarem, realizam operações lógicas (AND, OR, XOR), operações de deslocamento, incremento, complemento, soma, subtração, multiplicação e divisão (MONTEIRO, 2019). Uma ULA geralmente possui duas entradas de dados conectadas à saída (resultado da operação efetuada), entrada para sinais de controle para determinação da operação a ser realizada, além de saídas de comunicação com registradores e para sinalização de *flags*. Em processadores mais antigos, o barramento interno para dados é utilizado para interligar a ULA ao registrador acumulador, conhecido também como AC ou ACC (*accumulator*), e aos demais registradores, e na sequência à memória principal, conforme observado na figura a seguir.

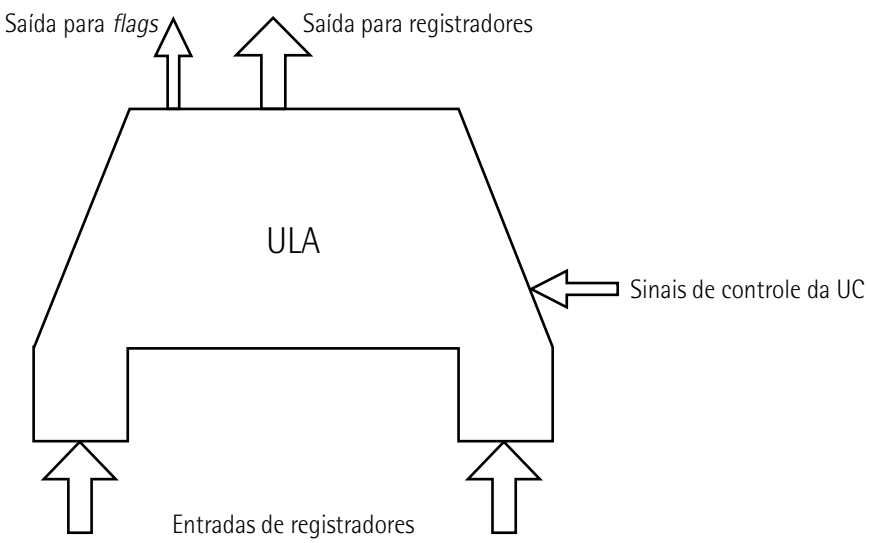


Figura 54 – ULA e suas conexões com a CPU

Alguns fabricantes de processadores atuais, como a Intel, têm substituído o nome unidade lógica e aritmética (ULA) por unidade de cálculo ou unidade de execução; outros, como a AMD, têm chamado a ULA de *integer unit* (UI). Alguns processadores atuais também são constituídos por unidades responsáveis por cálculos fracionários, representados em ponto flutuante, denominadas unidade de ponto flutuante (*floating point unit* – FPU).

3.1.7 Unidade de controle (UC)

A UC é considerada o dispositivo mais complexo da CPU. Ela é responsável pela movimentação de dados e instruções no processador através de sinais de controle sincronizados pelo relógio (*clock*). A UC opera sobre micro-operações (pequenos passos no ciclo de leitura), em que a cada início de um ciclo de instrução ocorrerá uma busca (*fetch*) da instrução solicitada, trazendo uma cópia dela para o processador, que será armazenada no registrador de instrução (*instruction register* – IR). Cada micro-operação é inicializada por um pulso oriundo da UC em decorrência de uma programação prévia, realizada diretamente no *hardware*. A estrutura básica diagramada de funcionamento da UC é observada na figura a seguir.

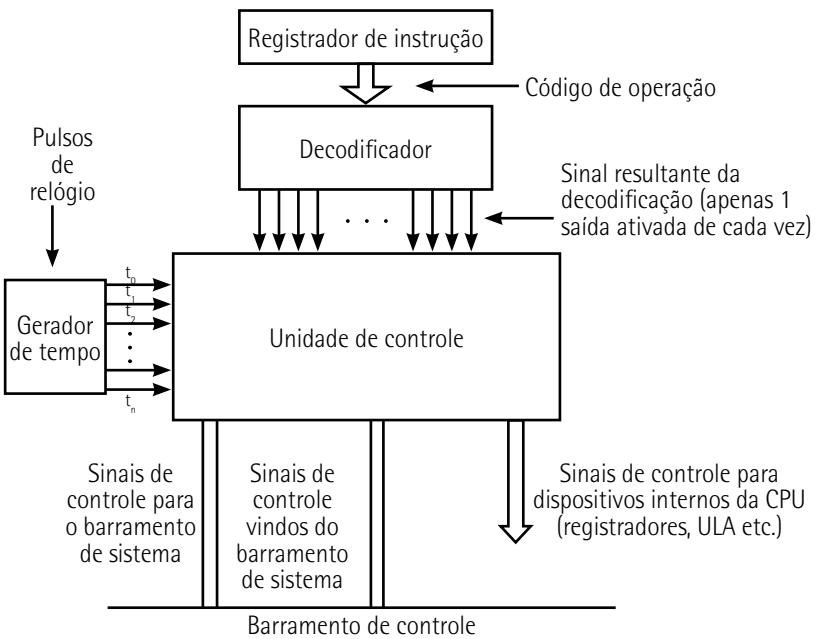


Figura 55 – Diagrama de bloco da função de controle

Os *opcodes* oriundos do registrador IR primeiramente são decodificados, ou seja, as instruções são interpretadas para servirem de sinais de execução na UC. Como é observado na figura anterior, um gerador de tempo está conectado à UC para ditar o ritmo (pulsos) em que as instruções devem ser inseridas no barramento para alguma ação a ser executada pela ULA. A UC possui também requisitos funcionais que são:

- Definição dos elementos básicos do processador.

- Descrição das micro-operações que o processador executa.

- Determinação das funções que a UC deve realizar para que as micro-operações sejam efetuadas.

Além desses requisitos funcionais, a UC também desempenha duas tarefas básicas: sequenciamento e execução:

- **Sequenciamento**: as micro-operações devem ser executadas em série, baseando-se no programa que está sendo executado.

- **Execução**: a UC deve garantir que cada micro-operação seja executada.

Para que a UC realize a operação da CPU de forma funcional, ela deve ser constituída por sinais de controle de entrada, como:

- **Registrador de instrução**: é utilizado para definir qual *opcode* deverá ser executado no ciclo de instrução corrente.

- ***Flags***: são necessárias para que a UC determine o estado do processador e das saídas das operações anteriores da ULA.

- **Sinais de controle do barramento de controle**: fazem parte do barramento do sistema para o fornecimento de sinais para a UC.

- **Sinais de controle dentro do processador**: são sinais que fazem os dados serem movidos de um registrador para outro registrador, além de ativarem funções específicas da ULA.

- **Sinais de controle para barramento de controle**: são constituídos pelos sinais de controle para memória principal e sinais de controle para os módulos de E/S.

- ***Clock***: é responsável pelo tempo de ciclo do processador ou tempo de ciclo de *clock* de operação do processador.

3.2 Desempenho de operação do processador

Os fabricantes de *chips*, por exemplo a Intel ou AMD, buscam incessantemente o aumento no desempenho de seus processadores, mesmo que a evolução dos processadores esteja atrelada à lei de Moore. Como existem limitações para aumentar a velocidade do *chip*, atualmente três técnicas básicas são utilizadas para contornar esse problema (STALLINGS, 2010):

- A primeira implica aumentar a velocidade de *hardware* do processador, em que o aumento deve se fundamentar na diminuição de tamanho das portas lógicas contidas no processador. Assim, quanto mais portas lógicas reunidas dentro de um mesmo *chip*, maior será a taxa de *clock*.

- A segunda implica aumentar a capacidade e velocidade das memórias auxiliares do tipo *cache*, inseridas entre o processador e a memória principal.

- A terceira técnica implica a realização de mudanças na arquitetura e organização de um processador, de modo a buscar o aumento na velocidade da execução das instruções, o que pode envolver algum processo de paralelismo na execução de tarefas.

Entre os fatores citados, o dominante, para que haja ganho no desempenho, tem sido o aumento na densidade de transistores no processador, aumentando assim a capacidade de *clock*. A figura a seguir mostra essa tendência nos processadores Intel, em que, à medida que a densidade de transistores aumenta, ocorrem problemas na dissipação de energia, além de uma estabilização na capacidade de *clock* que não evolui junto com a densidade.

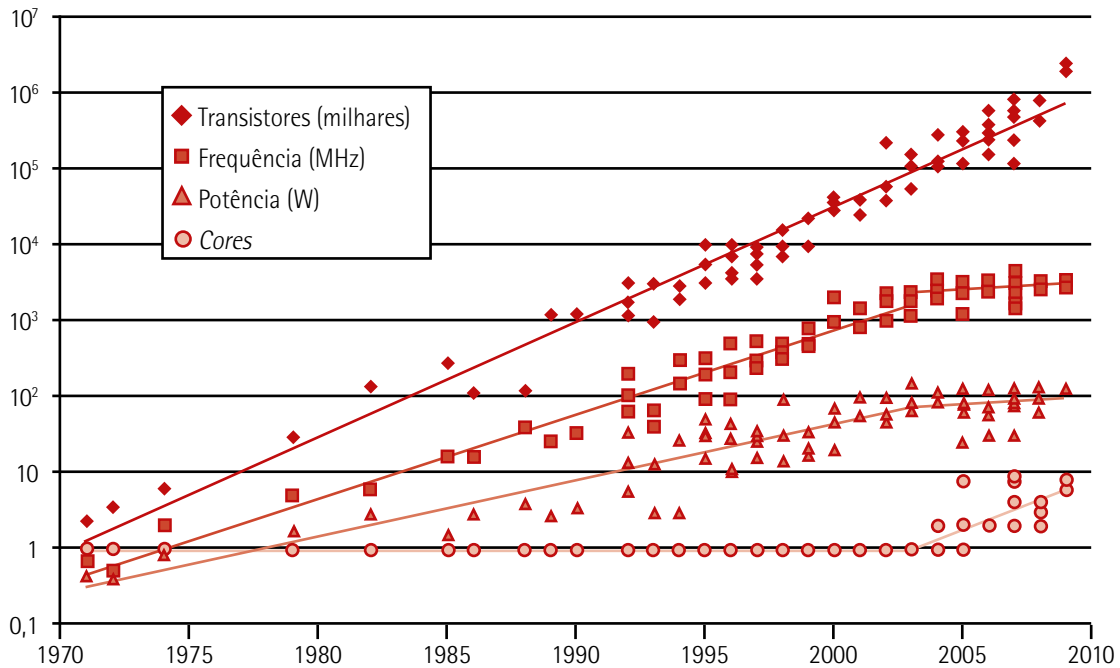


Figura 56 – Tendência de desempenho dos processadores

Os problemas que podem ser apontados como limitadores do aumento da velocidade de *clock* e, por consequência, do desempenho do processador também estão relacionados aos seguintes fatores:

- **Potência**: aumentar a densidade de transistores e, consequentemente, a velocidade de *clock* do processador também aumentará a densidade de potência dada em watts/cm^2. Isso levará a uma dificuldade para dissipar o calor gerado no *chip*, de forma que o calor será um sério problema a ser considerado no projeto.

- **Atraso de resistência e capacitância (RC)**: existe uma limitação física na velocidade em que os elétrons fluem no *chip* entre os transistores, que é dada pela resistência e capacitância dos fios metálicos que conectam os transistores. Assim, o atraso pode aumentar à medida que o produto RC aumenta. Consequentemente, com a diminuição dos componentes do *chip*, as interconexões

de fios metálicos se tornam mais finas, o que aumenta a resistência elétrica. E, ao diminuir o tamanho e a distância entre os transistores, os transistores também ficaram mais próximos um dos outros, aumentando a capacitância.

- **Latência de memória**: a velocidade de memória baixa limita a velocidade e o desempenho do processador.

Saiba mais

Conheça mais sobre os impactos da lei de Moore:

ROTMAN, D. We're not prepared for the end of Moore's Law. *MIT Technology Review*, 24 fev. 2020. Disponível em: https://bit.ly/38t8IpY. Acesso em: 3 fev. 2020.

3.2.1 *Clock* (relógio)

Como é de conhecimento, os processadores e as memórias são componentes digitais constituídos por portas lógicas. Os circuitos digitais mudam de estado ou comutam de estado (0 e 1) milhões de vezes por segundo ao executarem tarefas específicas, de acordo com instruções de um programa. Essas operações precisam estar sincronizadas para que executem tarefas de forma ordenada. O componente que realiza a sincronização é o relógio, conhecido também como *clock* (MONTEIRO, 2019).

O *clock* nada mais é do que um contador de tempo, e foi desenvolvido para gerar pulsos, cuja duração é denominada ciclo de *clock*. De forma geral, os pulsos comutam de valor para intensidade alta, que corresponde ao *bit* 1, para a intensidade baixa, que corresponde ao *bit* 0. Essa alternância de estado faz com que seja possível sincronizar e cadenciar as ações ou atividades do processador.

De certa forma, o *clock* também pode ser entendido como um dispositivo que realiza um controle, sincronizando todos os componentes do computador. A figura a seguir mostra um esquema com um conjunto de pulsos gerados por um *clock* e alguns de seus elementos principais.

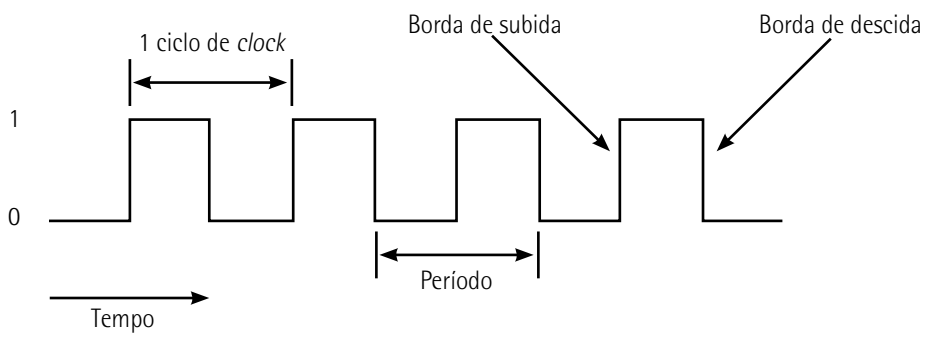


Figura 57 – Diagrama de pulsos de *clock*

Os elementos contidos na figura anterior são:

- **Ciclo de *clock* ou ciclo de relógio**: determina o intervalo de tempo entre o início da borda de subida (ou descida) do pulso, até o início da próxima borda de subida (ou descida) do outro pulso.
- **Período ou ciclo de tempo**: é o intervalo de tempo necessário para que o pulso execute uma oscilação completa.
- **Borda de subida**: é constituída pelo período utilizado pelo pulso para realizar a transição de subida.
- **Borda de descida**: é constituída pelo período utilizado pelo pulso para realizar a transição de descida.
- **Frequência ou taxa de *clock***: é a quantidade de ciclos por segundo do *clock*, determinada também pelo inverso do período e medida em hertz (Hz), em que 1 Hz será igual a um ciclo por segundo.

A figura a seguir mostra um diagrama de ciclo de *clock* original (T_0) de um processador constituído por outros cinco pulsos gerados (T_1, T_2, T_3, T_4 e T_5). O ciclo de *clock* geralmente é relacionado a uma operação elementar que ocorre durante o ciclo de instrução. Porém, mesmo as operações elementares não se realizam em um único passo de ciclo, de forma que um ciclo pode se subdividir em outros ciclos menores, conhecidos como subciclos. Ou subciclos podem estar defasados no tempo, de maneira que cada um pode acionar um passo diferente da operação elementar inicial. Essas diferenças de operações do ciclo fundamental também são conhecidas como micro-operações e possibilitam, entre outras coisas, o processamento paralelo.

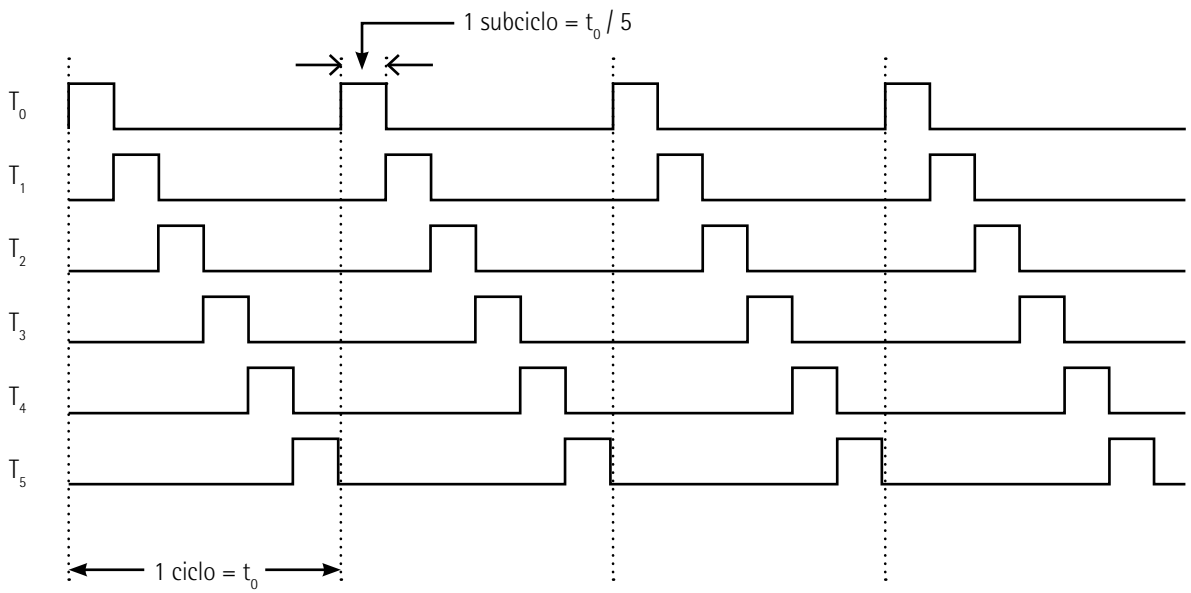


Figura 58 – Diagrama de ciclos de um processador com cinco subciclos

3.2.2 Taxa de execução de instruções por segundo

Como já definido antes, um processador é controlado através de um *clock* em uma frequência constante f ou, de forma equivalente, por um tempo de ciclo constante , em que: $\tau = \frac{1}{f}$.

Exemplo de aplicação

A partir da frequência de operação de um processador de 133 MHz, encontre o tempo de duração de cada ciclo.

Resolução

Utilizando a fórmula: $\tau = \frac{1}{f}$, tem-se: $\tau = \frac{1}{133 \times 10^6} = 7{,}51 \times 10^{-9}$ segundos ou 7,51 nanossegundos (ns).

A tabela a seguir mostra alguns dos prefixos para valores na base decimal (base 10) e base binária (base 2), organizados pela International Electrotechnical Commission.

Tabela 4 – Prefixo de valores e bases utilizados em computação

Prefixo	Símbolo	Potência de 10	Potência de 2	Prefixo	Símbolo	Potência de 10	Potência de 2
Kilo	K	1 mil = 10^3	2^{10} = 1024	Mili	m	1 milésimo = 10^{-3}	2^{-10}
Mega	M	1 milhão = 10^6	2^{20}	Micro	μ	1 milionésimo = 10^{-6}	2^{-20}
Giga	G	1 bilhão = 10^9	2^{30}	Nano	n	1 bilionésimo = 10^{-9}	2^{-30}
Tera	T	1 trilhão = 10^{12}	2^{40}	Pico	ˆpˆp	1 trilionésimo = 10^{-12}	2^{-40}
Peta	P	1 quadrilhão = 10^{15}	2^{50}	Femto	f	1 quadrilionésimo = 10^{-15}	2^{-50}
Exa	E	1 quintilhão = 10^{18}	2^{60}	Atto	a	1 quintilionésimo = 10^{-18}	2^{-60}
Zetta	Z	1 sextilhão = 10^{21}	2^{70}	Zepto	z	1 sextilionésimo = 10^{-21}	2^{-70}
Yotta	Y	1 setilhão = 10^{24}	2^{80}	Yocto	y	1 setilionésimo = 10^{-24}	2^{-80}

Adaptada de: Null e Lobur (2010).

Outro fator preponderante no cálculo da execução de instruções é a contagem de instruções I_C, para um certo programa que execute todas as instruções até um intervalo de tempo definido. Dessa forma, pode-se determinar um parâmetro importante, que é a média de ciclos por instrução (CPI – *cycles per instruction*).

Para instruções que exigem o mesmo número de ciclos de *clock* para ser executadas, a CPI será dada por um valor constante, de acordo com a frequência do processador. Dessa forma, é possível calcular o tempo T necessário para a execução de um determinado programa, dado por: $T = IC \times CPI \times \tau$.

Exemplo de aplicação

Qual será o tempo total de resposta para a execução de um determinado programa que possua 440 instruções (I_C), 8 ciclos por instrução e uma frequência (f) de processamento de 1,4 GHz?

Resolução

$T = I_C \times CPI \times \tau$, mas é importante lembrar que $\tau = \frac{1}{f}$, então:

$$T = I_c \times CPI \times \frac{1}{f} \rightarrow T = 440 \times 8 \times \left(\frac{1}{1,4 \times 10^9} \right) \rightarrow T = 2,51 \text{ microssegundos (2,51 } \mu s).$$

Outra medida muito utilizada para o desempenho de um processador é a taxa de execução, expressa em milhões de instruções por segundo (*millions of instructions per second* – MIPS), dada por: $\frac{f}{CPI \times 10^6}$.

Exemplo de aplicação

Considere a execução de um programa contendo 1.000 instruções, executados em um processador capaz de operar 0,6 ciclos por instrução a uma frequência de 200 MHz. Qual será a quantidade de MIPS para esses valores?

Resolução

Substituindo na fórmula:

$$\frac{f}{CPI \times 10^6} \rightarrow \frac{200 \times 10^6}{0,6 \times 10^6} = 333,33 \text{ MIPS ou 333,33 milhões de instruções por segundo.}$$

3.3 Máquina de von Neumann

3.3.1 Computador IAS

Embora atualmente a microarquitetura de um processador seja bem aceita, no início da computação, principalmente durante as gerações dos computadores das décadas de 1940 e 1950, os computadores ainda não possuíam um padrão bem definido de quais componentes eram necessários para que seu funcionamento ocorresse de forma otimizada, assim como não possuíam uma estruturação de quais componentes deveriam conter para melhor tratar dados/instruções.

Coube ao matemático húngaro John von Neumann (1903-1957), que trabalhava no Instituto de Estudos Avançados da Universidade de Princeton, nos Estados Unidos, realizar modificações no projeto do computador Eniac, em 1946. As tarefas de processamento e armazenamento de dados no Eniac eram extremamente enfadonhas. Von Neumann verificou que a programação poderia ser facilitada se o programa fosse representado de forma adequada para a realização do armazenamento na memória junto com os dados. Assim, um computador poderia obter suas instruções lendo-as diretamente da memória, e um programa poderia ser criado ou alterado através de endereços da memória, de forma que essa ideia ficou conhecida como conceito de programa armazenado (STALLINGS, 2010).

A publicação do novo projeto de von Neumann ocorreu em 1945 e foi base para a construção de um computador com uma nova proposta de arquitetura, o *Electronic Discrete Variable Automatic Computer* (EDVAC), terminado em 1949. Após o término do projeto do EDVAC, von Neumann se dedicou à construção de um novo computador, desenvolvido nos laboratórios do Instituto de Estudos Avançados de Princeton, cujo nome ficou conhecido como computador IAS (Institute of Advanced Estudies). O IAS ficou pronto em 1952 e é conhecido como o protótipo para todos os computadores modernos de uso geral. A figura a seguir mostra como foi projetada a estrutura geral do computador IAS.

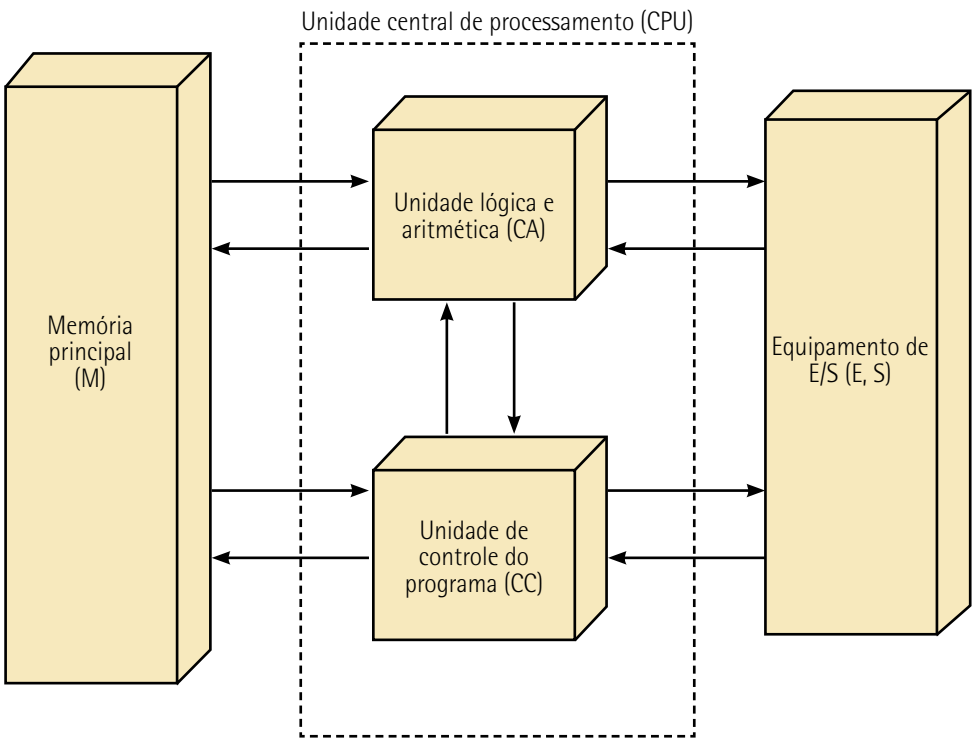


Figura 59 – Estrutura simplificada de um computador IAS

Von Neumann definiu que todos os computadores deveriam possuir as seguintes características:

- Memória principal para armazenamento dos dados.
- Unidade lógica e aritmética (ULA) para realizar operações em dados binários.

- Unidade de controle (UC) para interpretar/executar todas as instruções da memória principal.
- Dispositivos de entrada e saída (E/S) controlados pela UC.

Segundo von Neumann, um computador precisa ser capaz de realizar as operações elementares da aritmética (adição, subtração, multiplicação e divisão). Assim, é imprescindível que ele contenha as unidades especializadas para realizar tais tarefas. O controle lógico do dispositivo tem como função organizar a sequência apropriada de como as operações devem ser executadas. Von Neumann estabeleceu também que, para que os dispositivos possam executar operações em sequências longas e complicadas, se faz necessário o uso de uma memória que seja capaz de armazenar um volume grande de dados. Um computador deve ser capaz de estabelecer contato de entrada e saída com dispositivos internos/externos com a finalidade de ler/gravar dados. Como praticamente todos os computadores atuais possuem a mesma função e estrutura, eles também são conhecidos como máquinas de von Neumann.

A memória do computador IAS era constituída de 1.000 locais para armazenamento, chamados de palavras (*words*), com 40 dígitos binários. Uma palavra pode ser definida como um conjunto ordenado de bytes no qual a informação pode ser armazenada, processada e transmitida dentro de um computador. Geralmente, um processador possui um conjunto de instruções de tamanho fixo, de forma que o tamanho da instrução será igual ao tamanho da palavra. Por exemplo, um computador de 64 bits processará palavras de 64 bits ou mesmo duas palavras de 32 bits. A figura A a seguir mostra a formatação desses dados, em que cada número é representado por 1 bit de sinal e um valor de 39 bits contendo a palavra.

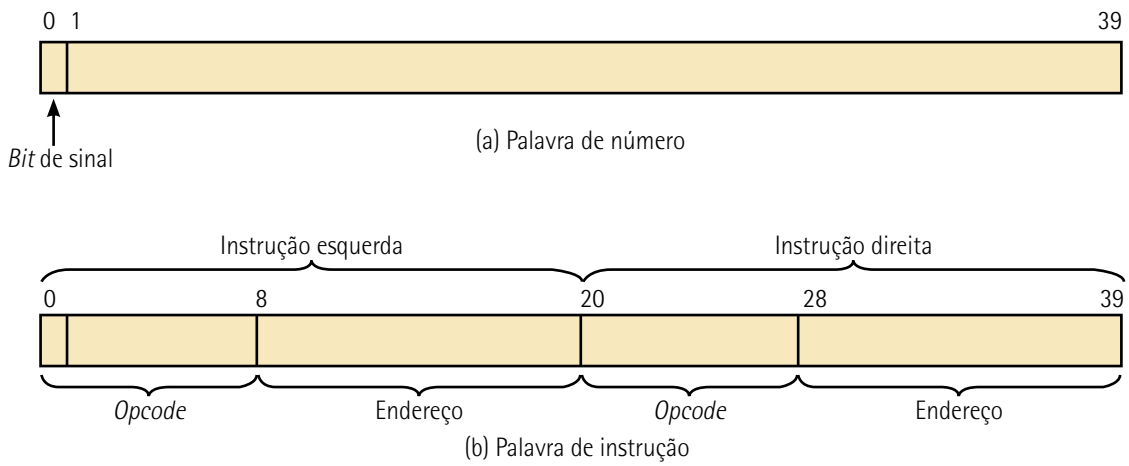


Figura 60 – Estrutura simplificada de um computador IAS

Uma palavra também pode ser subdividida em duas instruções de 20 bits (figura B), com cada instrução constituída por um *opcode* (código de operação) de 8 bits que especifica a operação realizada naquele momento, e um endereço de 12 bits que designa uma das palavras na memória, com valores numerados de 0 a 999.

A máquina de von Neumann foi desenvolvida basicamente utilizando componentes de memória (registradores) que auxiliam em toda as tarefas da UC e da ULA.

Saiba mais

Conheça mais sobre a máquina de von Neumann:

FREIBERGER, P. A.; SWAINE, M. R. Von Neumann machine. *Encyclopedia Britannica*, 14 nov. 2016. Disponível em: https://bit.ly/3byu6GW . Acesso em: 3 fev. 2021.

3.3.2 Arquitetura Harvard

A arquitetura Harvard teve seu início com o desenvolvimento do computador eletromecânico Mark III, em 1950. Esse computador possuía memórias diferentes para dados e instruções.

A grande diferença entre a arquitetura de von Neumann e a arquitetura Harvard é que, enquanto a arquitetura von Neumann utiliza o mesmo barramento para envio/recebimento de dados e instruções (o que pode ocasionar gargalos na execução de tarefas de busca), a arquitetura Harvard utiliza barramentos diferentes. Como a arquitetura Harvard é capaz tanto de ler instruções ou dados ao mesmo tempo, há um grande benefício para a utilização de processamento paralelo como o *pipeline*, o que aumenta a velocidade de execução das aplicações. A figura a seguir mostra de forma simplificada a diferença entre essas duas arquiteturas.

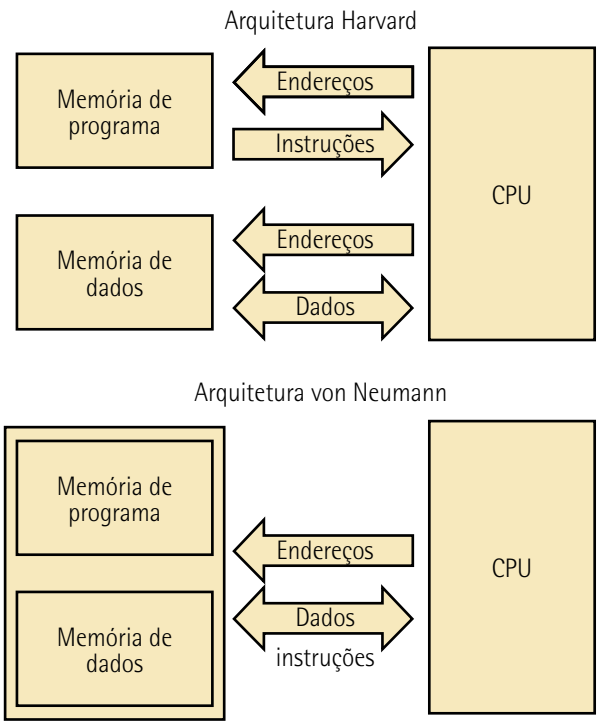


Figura 61 – Diferenças básicas entre as arquiteturas Harvard e von Neumann

3.4 Organização dos registradores

Os registradores são dispositivos de memória, constituídos basicamente de portas lógicas, que fornecem armazenamento temporário para dados/instruções dentro do processador.

Os registradores estão no topo da hierarquia da memória, pois são mais rápidos, ou seja, possuem maior velocidade de transferência de dados/instruções para dentro ou fora do processador. Apesar disso, os registradores possuem uma capacidade de armazenamento menor, se comparada a outras memórias internas, e também possuem custo mais elevado em relação a outras memórias. Os registradores geralmente são classificados de acordo com sua funcionalidade em: registradores de propósito geral e registradores de controle e estado (STALLINGS, 2010).

3.4.1 Registradores de propósito geral ou "visíveis" ao usuário

Os registradores de propósito geral possibilitam que o programador de linguagem de baixo nível (linguagem de máquina), como Assembly, realize referências diretamente à memória principal. Esses registradores podem ser subdivididos, ainda, como: de uso geral, dados, endereços e códigos condicionais.

Os registradores de uso geral são atribuídos a uma variedade de funções de acordo com a necessidade do programador de Assembly. Essa categoria de registradores pode conter um operando para realizar qualquer operação, por exemplo operações de ponto flutuante ou operações que envolvam processo de pilhas.

Em algumas situações, os registradores de uso geral também podem ser utilizados para funções de endereçamento ou dados. Como registradores de dados, eles podem ser utilizados somente para guardar dados temporários, e não podem ser empregados no cálculo do endereço de um operando. Como registradores de endereços, eles podem se dedicar exclusivamente ao modo de endereçamento de ponteiros de segmento, quando o endereço não está somente em um lugar exclusivo, mas sim segmentado. Podem se comportar como registradores de índice para indexar endereços ou como ponteiros de pilha, caso exista a necessidade de empilhamento de endereços.

3.4.2 Registradores de controle e estado

Os registradores de controle e estado são utilizados no funcionamento e organização do processador e, diferentemente dos "visíveis" ao usuário, eles estão ocultos para programação por desenvolvedores de linguagens de baixo nível como Assembly, ficando apenas disponíveis para o sistema operacional. Esses registradores são organizados da seguinte forma:

- **Registrador de *buffer* de memória (*memory buffer register* – MBR)**: é o registrador que recebe uma ou várias palavras que serão armazenadas na memória ou enviadas para alguma unidade de E/S. Pode-se entender a palavra também como algum dado a ser processado pelo computador que ficará temporariamente armazenado aguardando instruções, ou uma palavra pode ser interpretada como o resultado do processamento e seu próximo passo será o envio para memória principal e unidades de E/S.

- **Registrador de instrução (*instruction buffer register* – IBR)**: é utilizado para armazenar temporariamente a próxima instrução a ser executada.

- **Registrador de endereço de memória (*memory address register* – MAR)**: é o registrador que especifica o endereço na memória principal a ser lido/escrito.

- **Registrador de instrução (*instruction register* – IR)**: é o registrador que contém um código de operação (*opcode*) que está em execução.

- **Registrador contador de programa (*program counter* – PC)**: é o registrador que contém o endereço para busca de um par de instruções contidas na memória principal.

Além dos registradores de controle e estado, a máquina de von Neumann também possui os registradores acumulador (AC) e multiplicador (MQ), que são empregados para manter temporariamente os operandos (que especificam os dados que serão modificados) e resultados de operações da ULA.

A figura a seguir mostra uma estrutura detalhada envolvendo todos os registradores do IAS, assim como o processador constituído pela ULA e UC e as comunicações via barramento de dados/instruções conectando a memória principal e algum equipamento de E/S.

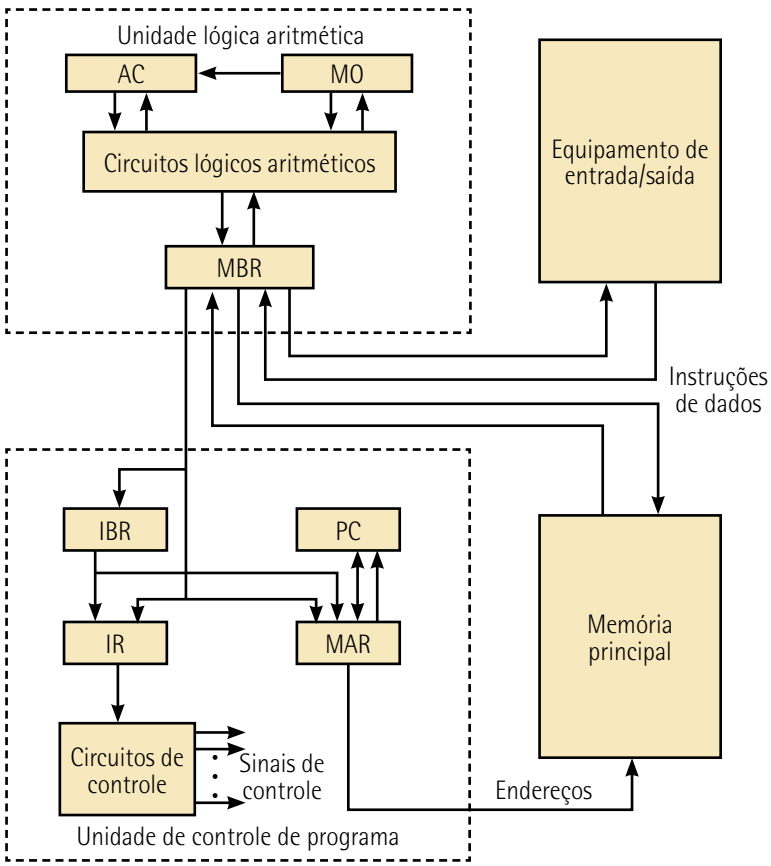


Figura 62 – Estrutura expandida de um computador IAS

A máquina de von Neumann realiza repetidamente um ciclo de instrução em que cada ciclo consiste em dois subciclos. Durante o primeiro ciclo (ciclo de busca de instrução ou *fetch cycle*), o código de operação (*opcode*) da próxima operação é carregado no registrador de instrução (IR), e parte do endereço é carregada no registrador de endereço de memória (MAR). A instrução então poderá ser retirada do IBR ou ser recebida da memória principal carregando-se uma palavra no MBR e, na sequência, para IBR, IR e MAR. O computador IAS possuía um total de 21 instruções agrupadas como segue:

- **Transferência de dados**: são instruções que movimentam os dados entre a memória e os registradores da ULA ou mesmo entre dois registradores diretamente na ULA.

- **Desvio incondicional**: são instruções que podem modificar a sequência de execução das instruções para facilitar operações repetitivas.

- **Desvio condicional**: são instruções que podem se tornar dependentes de uma condição do programa, permitindo que ele possua pontos de decisão.

- **Aritméticas**: são instruções realizadas pela ULA consistindo em operações binárias de adição, subtração, multiplicação e divisão.

- **Modificação de endereço**: são instruções que permitem que os endereços sejam calculados na ULA e inseridos em instruções armazenadas na memória, permitindo uma flexibilidade para realização do endereçamento.

Não são todos os processadores que possuem registradores designados para tarefas específicas como o MAR e o MBR, porém é necessário que haja algum mecanismo de armazenamento temporário (*buffer*). Geralmente, o processador atualiza o registrador *program counter* (PC) após ler cada instrução para que o PC sempre possa apontar para as próximas instruções a serem executadas. Uma instrução de desvio condicional ou incondicional também irá modificar o conteúdo do registrador PC, de forma que a instrução é lida e colocada no registrador IR, em que o *opcode* será analisado. Os dados são transferidos entre a memória e os registradores MAR e MBR através do barramento interno do processador.

Como observado, existem diversos caminhos para execução dos dados na máquina de von Neumann, a qual é realizada por registradores contidos na ULA e na UC assim como pelos diversos barramentos que conectam esses dispositivos. Por exemplo, um banco de registradores genéricos recebe os valores A e B para realização da operação aritmética de adição na ULA. A ULA efetuará a operação nessas entradas, produzindo um resultado no registrador de saída que é armazenado em um outro registrador e que pode, na sequência, ser armazenado na memória principal, a fim de ser enviado para algum dispositivo de E/S. A figura a seguir mostra uma operação de adição de dois valores.

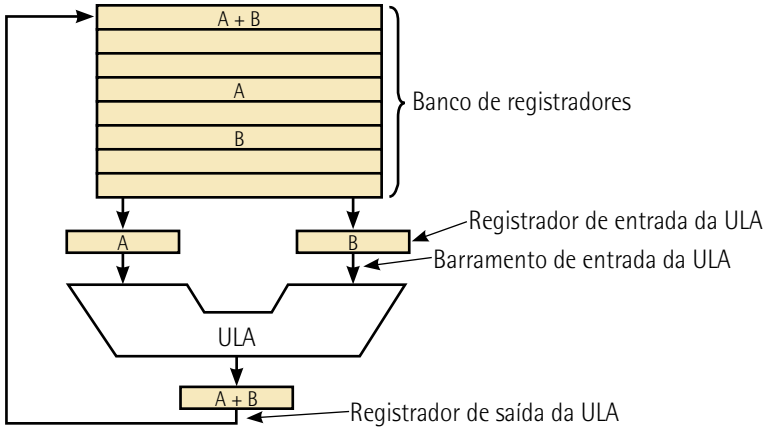


Figura 63 – Caminho de execução de instruções na máquina de von Neumann

3.4.3 Organização de registradores em diferentes microprocessadores

Muitos modelos de processadores podem incluir um conjunto de registradores conhecidos como de palavra de estado do programa (*program status word* – PSW), que contém as informações de códigos condicionais ou informação de estado ou *flags*, como:

- **Sinais**: *bits* resultantes da última operação aritmética.
- **Zero**: quando o resultado da operação for zero.
- ***Carry***: quando uma operação resulta em um transporte (adição) para empréstimo (subtração) de um *bit* de ordem superior. Também é utilizado em operações aritméticas para tratamento de múltiplas palavras.
- **Igual**: utilizado para realizar uma comparação lógica de igualdade.
- **Overflow**: utilizado para indicar uma sobrecarga aritmética.
- **Habilitar/desabilitar interrupções**: utilizado para habilitar ou desabilitar interrupções.
- **Supervisor**: utilizado na identificação do uso do processador em modo supervisor ou modo de usuário.

Além desses registradores, vários outros também são utilizados na organização dos registradores de controle e estado de um processador e variam de acordo com o modelo dele, por exemplo caso haja um ponteiro para um bloco de memória contendo informações adicionais sobre algum processo. Outros modelos de processadores podem utilizar máquinas baseadas em interrupções vetorizadas, fazendo-se necessário um registrador que atenda a esse tipo de requisição de endereçamento. Em outros processadores, se faz necessário o uso de registradores do tipo pilha, quando se precisa realizar chamadas de sub-rotinas que estão em um topo da pilha. Há também registradores utilizados como ponteiros de tabela de páginas da memória virtual, além dos registradores utilizados para operações de E/S.

Em um projeto de processadores, há uma série de fatores que influenciam na inclusão ou não de certos registradores de controle e estado. Uma das questões relacionadas a esse projeto remete ao sistema operacional. Geralmente, um projetista de sistema operacional deve ter o conhecimento funcional dos registradores para que eles possam funcionar de modo a otimizar a conexão entre o *hardware* e o *software*. Outra questão fundamental do projeto dos registradores é referente à alocação de informações e comunicação entre registradores e a memória. Em geral, se dedicam algumas centenas ou milhares de palavras de memória e controle, e o projetista de *hardware* deve decidir quanto da informação será armazenada nos registradores e quanto da informação ficará na memória. Alguns processadores de 16 ou 32 bits são baseados nessa arquitetura de organização de registradores, podem servir de exemplo o Motorola MC68000, o Intel 8086 e o Pentium 4 (STALLINGS, 2010).

O MC68000 possui registradores de 32 bits que são divididos em oito registradores para manipulação de dados e nove registradores de endereços indexados. Por possuir registradores de 32 bits, nesse tipo de processador há uma facilidade de realizar operações de dados de 8, 16 ou 32 bits determinadas pelo código da operação realizada. Dada a capacidade de 32 bits dos registradores de endereços, não há a necessidade de segmentação de endereçamento; além disso, dois desses nove registradores podem ser utilizados como ponteiros de pilha.

Outros registradores possuem funcionalidades bem conhecidas, como para uso do sistema operacional, contador de programa, controle e estado e registradores para uso do próprio usuário. A figura A a seguir mostra a organização dos registradores do MC68000.

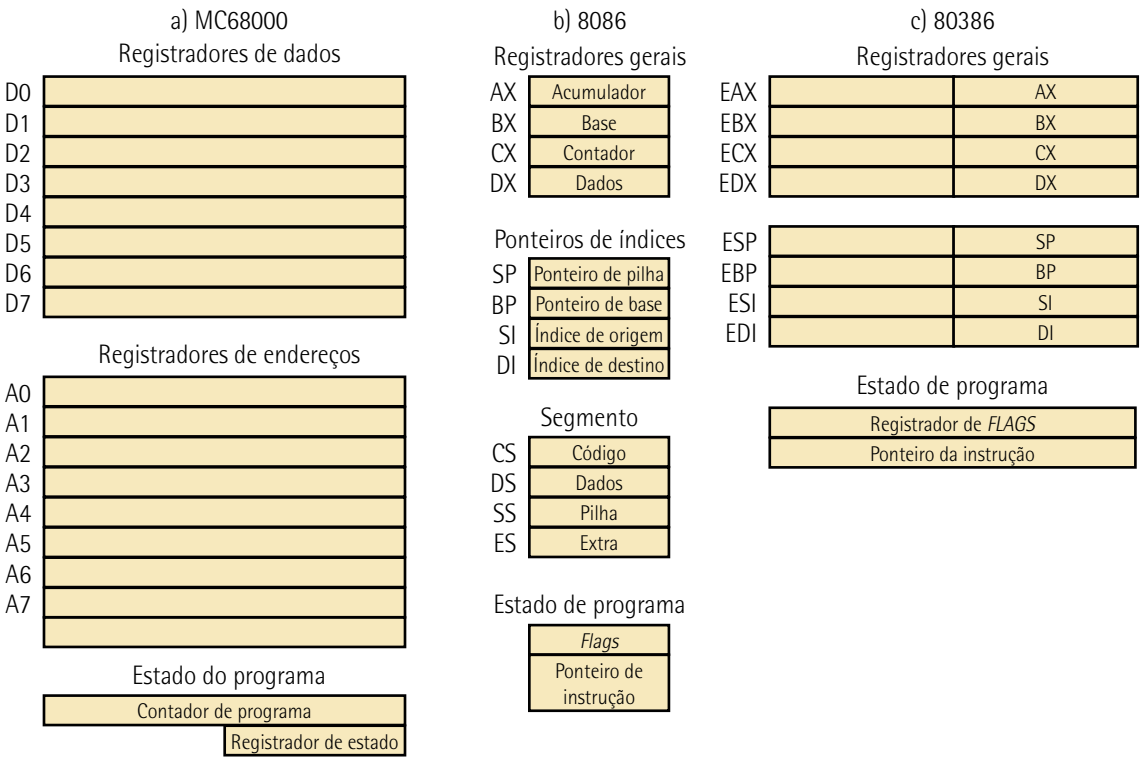


Figura 64 – Exemplo de organização de registradores em processadores

Observação

O processador 8086, primeiro processador de 16 bits fabricado pela Intel, utiliza uma abordagem diferente na organização de seus registradores, pois cada registrador possui um uso específico.

Alguns registradores podem ser utilizados para uso geral, enquanto outros podem ser utilizados para realizar endereçamento ou armazenamento de dados e instruções, além de registradores de 16 bits para atuarem como índices e ponteiros. Além desses, o 8086 possui quatro registradores de segmento de instruções, como para realização de desvio, segmento de dados e segmento de pilha.

Na figura 64C, pode-se observar a organização do processador 80386, que possui registradores de 32 bits constituídos pelos registradores de uso geral AX, BX, CX e DX, além dos de controle SP, BP, SI e DI e de estado (registrador de *flags* e ponteiro de instruções).

O uso de um processador de 32 bits também permite a compatibilidade de programas escritos para arquiteturas de 8 ou 16 bits, o que possibilitou aos projetistas maior flexibilidade ao projetar a organização dos registradores.

3.5 Arquitetura do processador Intel x86 e sua evolução

A arquitetura x86 teve sua origem no processador de apenas 4 bits Intel 4004, fabricado em larga escala a partir de 1971 para concorrer com a crescente indústria japonesa de eletrônicos. Esse processador apresentava como característica possuir um *chip* do tipo *dual in-line package* (DYP), com circuitos eletrônicos (transistor) totalmente integrados.

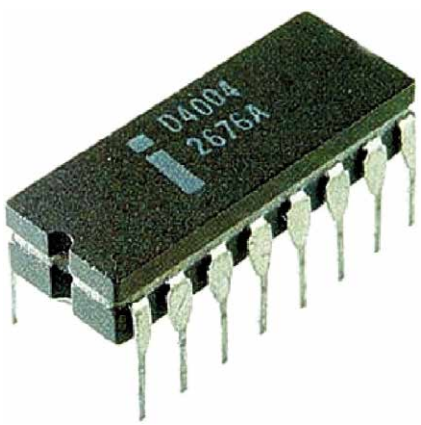


Figura 65 – Processador Intel 4004

Após o sucesso do 4004, a Intel desenvolveu melhorias na sua linha de processadores, e lançou em 1972 um processador de 8 bits para dados e 16 bits para palavras, denominado Intel 8008. Esse processador possuía surpreendentes 500 KHz de velocidade de *clock*.

Figura 66 – Processador Intel 8008

Como evolução natural do processador 8008, a Intel lançou em 1974 o primeiro processador de uso geral do mundo, o 8080. Esse processador possuía 7 registradores de uso geral e capacidade de endereçamento de memória de 64 Kbytes, além de um *clock* de 2 MHz, que foram utilizados no computador da Xerox, o Altair.

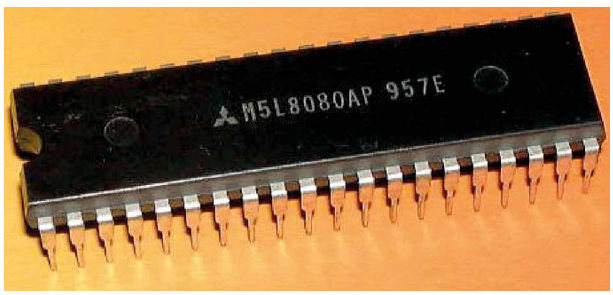


Figura 67 – Mitsubishi 8080

O processador 8086 foi o precursor da arquitetura x86 e serve de base para os processadores atuais. Ele era um circuito muito poderoso para época (1978) e operava a 16 bits, tanto para dados quanto para palavras, além da possibilidade de endereçamento de 1 Mb de posições, direcionados em um caminho para dados (barramento) de 20 bits.

Figura 68 – Processador Intel 8086

O processador 8086 possuía uma estrutura simples, o que facilitava sua programação em linguagem de montagem. Internamente, o Intel 8086 é dividido em duas unidades: unidade de execução e unidade de interface com o barramento (PANNAIN; BEHRENS; PIVA JR., 2012):

- **Unidade de execução (*execution unit* – EU)**: a EU está situada na ULA e é responsável pela execução das operações lógicas e aritméticas e das instruções de máquina.

- **Unidade de interface com o barramento (*bus interface unit* – BIU)**: é a unidade responsável pela comunicação de dados entre a ULA e os meios externos, como memória e dispositivos de E/S. Também é responsável pela transmissão de sinais de endereçamento, dados, controle e busca de instruções.

A figura a seguir mostra a arquitetura do processador 8086 e seus registradores operacionais.

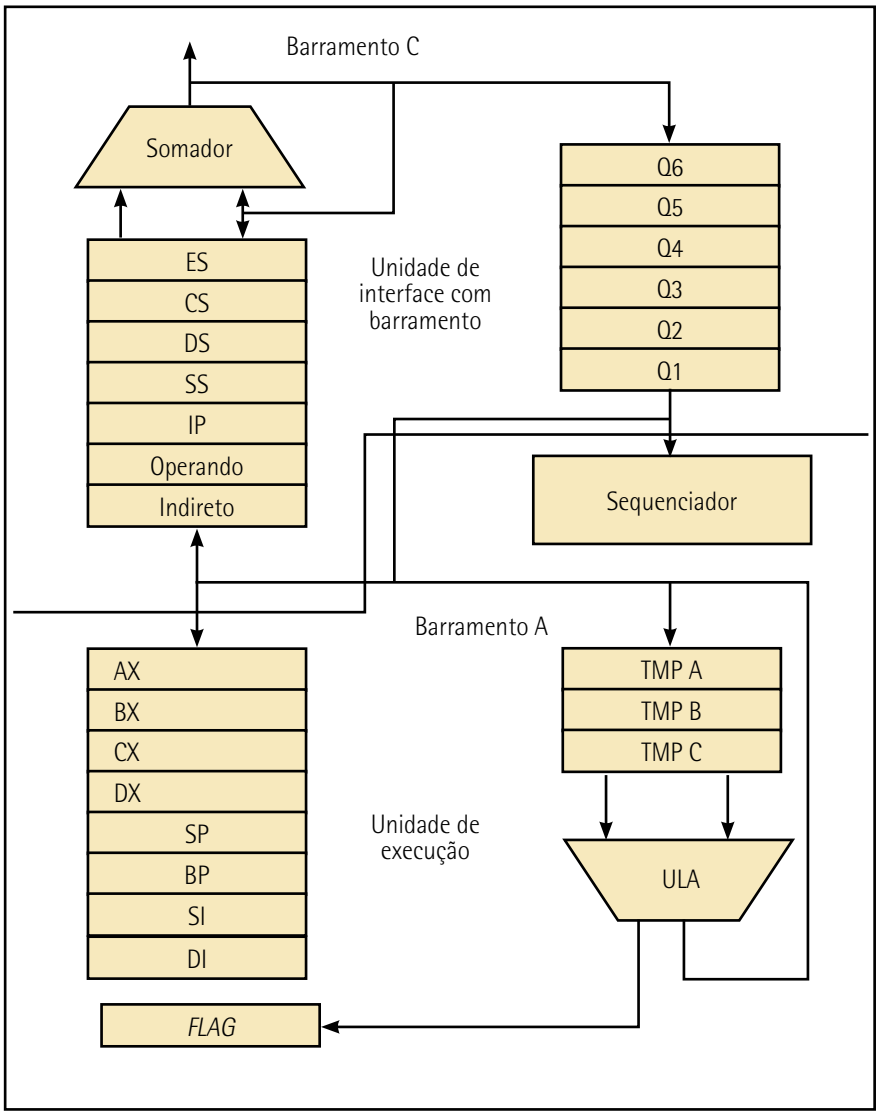


Figura 69 – Organização básica do processador Intel 8086

Os registradores do Intel 8086 estão divididos em registradores de propósito geral e registradores específicos ou de estado e controle. Registradores específicos geralmente são utilizados como segmentadores, apontadores e índices de endereçamento, além de sinalizadores de estado e controle (*flags*). O 8086 possuía quatro registradores de propósito geral de 16 bits, que também podiam operar a 8 bits (AX, BX, CX e DX), utilizados para operações lógicas e aritméticas:

- **Registrador AX (acumulador)**: utilizado em operações lógicas e aritméticas, para instruções de E/S e em instruções de operações de ajuste *binary coded decimal* (BCD – codificação binária decimal).

- **Registrador BX (base)**: utilizado como registrador base para referenciação a posições da memória principal, ou seja, é capaz de armazenar o endereço base de uma tabela ou vetor de dados, onde as posições são obtidas adicionando-se um valor de deslocamento (*offset*).

- **Registrador CX (contador)**: utilizado em operações iterativas e repetitivas para contagem de *bits*, *bytes* ou palavras, sendo incrementado ou decrementado quando necessário.

- **Registrador DX (dados)**: utilizado em operações para armazenamento de parte de um produto de 32 bits ou em operações também de 32 bits para armazenar o resto de uma operação de divisão.

Os registradores específicos ou de controle e estado podem ser classificados em: segmento de código, segmento de dados, segmento de pilha e segmento extra (também utilizado para dados). Todos esses registradores são de 16 bits e são utilizados para trabalhar com blocos de memória de 64 Kbytes:

- **Registrador de segmento de código (*code segment* – CS)**: usado para apontar para uma área de memória que contém o código do programa que está sendo executado.

- **Registrador de segmento de dados (*data segment* – DS)**: usado para apontar para um segmento de memória que será utilizada no armazenamento de dados do programa em execução.

- **Registrador de segmento de pilha (*stack segment* – SS)**: utilizado na identificação do segmento que será utilizado como pilha (*stack*), com o objetivo de armazenar os dados temporários do programa em execução.

- **Registrador de segmento extra de dados (*extra segment* – ES)**: utilizado para determinar um segmento extra de dados, como no armazenamento e acesso da memória de vídeo.

Observação

O endereçamento no Intel 8086 pode apresentar algumas diferenças no armazenamento de dados na memória principal ou em uma pilha de registradores.

Em cada situação, serão alocados segmentos ativos endereçáveis para alocar os 64 Kbytes. Os segmentos ativos são: segmento de código endereçado por CS, segmento de dados endereçado por DS, segmento de pilha endereçado por SS e segmento extra de dados endereçado por ES. Os registradores citados anteriormente (CS, DS, SS e ES) armazenam o endereço base de seu segmento; então, há uma necessidade de haver outros registradores para armazenar o deslocamento (*offset*) dentro de um dado segmentado, como mostra o esquema da figura a seguir.

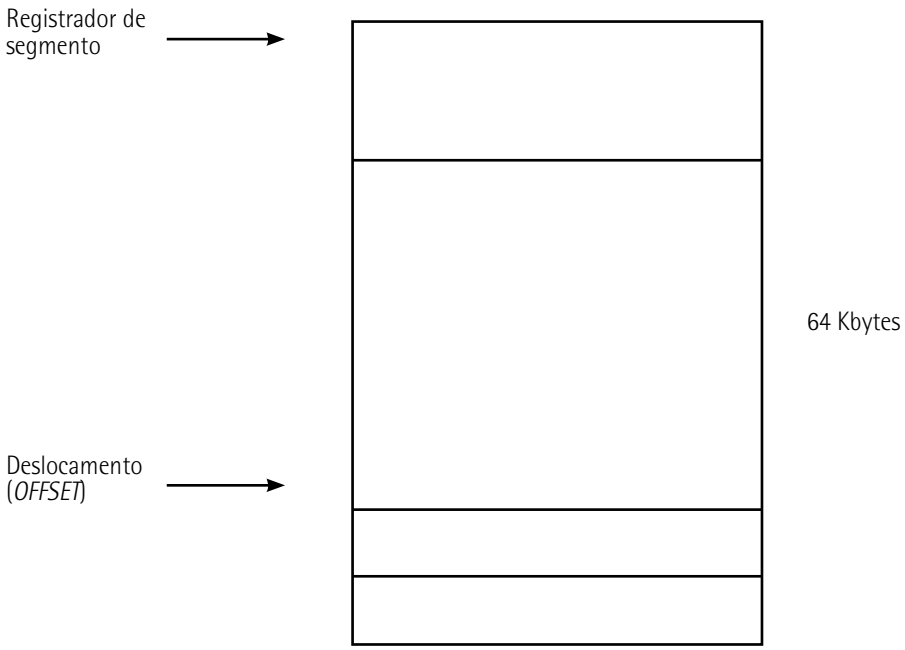


Figura 70 – Segmento de endereço base e deslocamento

O Intel 8086 possui outros registradores de 16 bits auxiliares, como:

- **Registrador IP (*instruction pointer*)**: é considerado o *program counter* (PC) do 8086 e é utilizado em conjunto com o registrador CS para localizar uma posição, em um segmento de código em operação, da próxima instrução que será executada. Esse segmento será automaticamente incrementado em função do número de *bytes* da instrução após a operação de busca dessa instrução.

- **Registrador SP (*stack pointer*)**: utilizado em conjunto com o registrador SS no acesso à área de pilha contida na memória principal, apontando sempre para o topo da pilha.

- **Registrador BP (*base pointer*)**: utilizado no acesso ao conjunto de dados que está dentro do segmento de pilha, como um vetor de dados.

- **Registrador SI (*souce index*)**: usado como registrador de índice em modos de endereçamento indireto, em conjunto com o registrador DS.

- **Registrador DI (*destination index*)**: atua em conjunto com ES, quando utilizado em instruções de manipulação de *string*, ou com DS em acesso a vetores armazenados no segmento de dados.

- **Registrador de sinalizadores (*flags*)**: é um registrador de 16 bits que tem como função indicar um estado do processador após a execução de uma instrução.

Após o lançamento do 8086, a Intel continuou a desenvolver processadores mais robustos, como o 80286 (figura a seguir), capaz realizar um endereçamento de memória de 16 MB e que tinha 6 MHz de *clock*.

Figura 71 – Processador Intel 80286

Em 1985, a Intel lançou seu primeiro processador multitarefa (*multitask*) de 32 bits, o 80386 (figura a seguir), capaz de executar várias tarefas simultâneas e que seria muito utilizado em minicomputadores e *mainframes*.

Figura 72 – Processador Intel 80386

Como resultado do contínuo avanço da Intel em desenvolver novas tecnologias, em 1989 foi lançado o 80486 (figura a seguir), com um poder de processamento de 25 MHz de *clock*, mais do que dobrando a capacidade do 80386, que tinha apenas 12 MHz de *clock*. O 80486 oferecia um coprocessador matemático interno, auxiliando a CPU na realização das tarefas matemáticas complexas.

Figura 73 – Processador Intel 80486

Apesar de a família de processadores da Intel possuir essa codificação utilizando o número 86, embora sua arquitetura ainda continuasse baseada no x86, em 1993 foi lançado o Pentium (figura a seguir), baseado em um poder de processamento de 66 MHz.

Figura 74 – Processador Intel Pentium

Observação

Esse processador foi um grande avanço nas arquiteturas da Intel, pois se baseava em técnicas superescalares, que permitiam que múltiplas instruções fossem executadas em paralelo.

A sequência de lançamentos da Intel ocorre com o Pentium II (1997), baseado na tecnologia MMX, projetada para processamento eficaz de áudio e vídeo e poder de processamento de 233 MHz. Fabricado em 1999 e com uma frequência de *clock* de 450 MHz, o Pentium III incorpora melhorias nas operações em ponto flutuante, melhorando seu desempenho em gráficos 3D.

A sétima geração de processadores da Intel veio com o lançamento do Pentium 4 em 2000, e foi um dos maiores sucessos tecnológicos da empresa. O Pentium 4 possuía surpreendentes 1,3 GHz de *clock*, melhorias de desempenho em processos multimídia e de ponto flutuante. Antes do lançamento das famílias i3, i5, i7 e i9 em 2009, a Intel ainda lançou em 2006 o Intel Core e o Intel Core 2. Esses processadores foram os primeiros com arquitetura de 64 bits e núcleo de quatro processadores em um único *chip*.

Saiba mais

Para conhecer um pouco mais sobre a evolução dos processadores da Intel e sua arquitetura interna, leia:

JORDÃO, F. Tabela de processadores: Intel. *Tecmundo*, 2 jul. 2013. Disponível em: https://bit.ly/3t7kkS4. Acesso em: 3 fev. 2021.

4 CONJUNTO DE INSTRUÇÕES

4.1 Ciclo de instrução

A maior parte das instruções pode ser dividida em duas categorias, relacionadas a como elas interagem com o *hardware*: registrador-memória ou registrador-registrador. As instruções do tipo registrador-memória permitem que palavras da memória possam ser buscadas em registradores para serem utilizadas como entradas pela ULA em instruções sequenciais. No outro tipo de instrução (registrador-registrador), uma instrução típica busca dois operandos nos registradores, os traz para a ULA e efetua alguma operação aritmética ou booleana com eles, e armazena o resultado em um dos registradores. Esse método é conhecido como ciclo de caminho de dados e é uma das principais funções dos registradores das CPUs, de forma que, quanto mais rápido for o ciclo do caminho de dados, mais rápido será o funcionamento do computador.

Para que ocorra essa comunicação de instruções pelos registradores e/ou pela memória, é necessário entender como de fato ocorre o ciclo de busca e execução de instruções. Quando se trata do ciclo direto de execução de instruções em sua forma mais simplificada, primeiro o processador busca as instruções (lê a próxima instrução da memória dentro do processador) uma por vez, e as executa (interpreta o código de operação e efetua a operação indicada).

Lembrete

A execução de um programa é baseada em repetir o processo de busca e execução da instrução quando houver mais instruções a serem executadas em um mesmo ciclo. Interrupções também podem ocorrer durante a execução das instruções.

Nesses casos, salva-se o estado do processo atual e atende-se à instrução de interrupção indicada. Após o término da execução das instruções, o processo é finalizado. A figura a seguir mostra um ciclo de instrução básico.

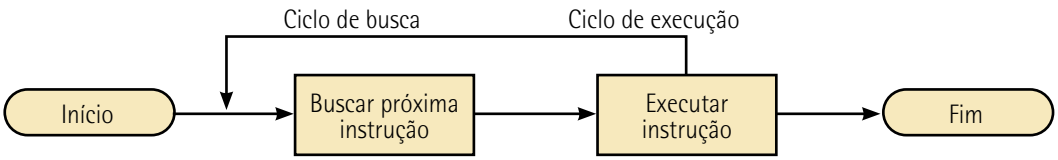


Figura 75 – Ciclo de instrução

A execução de instruções no ciclo é dependente do tipo de instrução que será executada e, mais intrinsicamente, de qual operando será utilizado nas operações.

4.2 Tipos de operandos

Um operando nada mais é do que uma entrada ou argumento em uma dada instrução, e geralmente é dividido em categorias: números, caracteres, endereços e dados lógicos. O endereçamento é uma forma de dados e pode conter cálculos realizados sobre a referência de memória do operando em uma dada instrução.

4.2.1 Números

Todas as linguagens de máquina como o Assembly incluem algum tipo de dado numérico. Mesmo algum tipo de processamento de dados não numérico terá a necessidade de que haja números atuantes, como contadores, tamanho de campo etc. Existem basicamente três tipos de dados numéricos encontrados em um computador:

- **Inteiros binários ou ponto fixo binário**: representados pelos dígitos zero e um, sinal de menos e vírgula fracionada.
- **Ponto flutuante binário**: representado por um intervalo de inteiros positivos e negativos centrados em zero.
- **Decimal**: organizado em casas decimais utilizando a vírgula para indicar a que ordem o número pertence.

4.2.2 Caracteres

Baseado no uso de texto ou *strings* de caracteres, esse tipo de codificação foi desenvolvido para facilitar o armazenamento de dados e instruções. Diversos códigos foram desenvolvidos, e geralmente são representados por uma sequência de *bits*, sendo o *American Standard Code for Information Interchange* (ASCII) o mais utilizado atualmente. Cada caractere no ASCII é representado por um padrão exclusivo de 7 bits, de modo que 128 caracteres diferentes poderão ser representados. Como os computadores atuais são orientados a *byte*, cada caractere no ASCII será armazenado em um *byte* separado, conforme pode ser observado nas tabelas a seguir.

Tabela 5 – Parte do conjunto de caracteres ASCII de controle

Hexa	Nome	Significado	Hexa	Nome
0	NUL	Null	10	DLE
1	OH	tart	Of	Heading
2	TX	tart	Of	Text
3	ETX	End	Of	Text
4	EOT	End	Of	Transmission
5	ENQ	Enquiry	15	NAK
6	ACK	ACKnowledgement	16	YN
7	BEL	BELI	17	ETB
8	BS	BackSpace	18	CAN
9	HT	Horizontal	Tab	19
A	LF	Line	Feed	1A
B	VT	Vertical	Tab	1B
C	FF	Form	Feed	1C
D	CR	Carriage	Return	1D
E	O	hift	Out	1E
F	I	hift	In	1F

Adaptada de: Tanenbaum e Austin (2013, p. 108).

Tabela 6 – Parte do conjunto de caracteres ASCII de uso geral

Hexa	Nome	Significado
0	NUL	Null
1	OH	tart
2	TX	tart
3	ETX	End
4	EOT	End
5	ENQ	Enquiry
6	ACK	ACKnowledgement
7	BEL	BELI

Hexa	Nome	Significado
8	BS	BackSpace
9	HT	Horizontal
A	LF	Line
B	VT	Vertical
C	FF	Form
D	CR	Carriage
E	O	hift
F	I	hift

Adaptada de: Tanenbaum e Austin (2013, p. 108).

A tabela ASCII contém os códigos de 0 a 1F (hexadecimal) para realizar o controle operacional, e de 20 a 7F utilizados como caracteres de uso geral do computador.

4.2.3 Dados lógicos

Um dado lógico consiste em uma palavra ou unidade endereçável, tratadas como uma única unidade de dados. Nesse caso, é útil considerar essa unidade constituída de n bits de dados de 1 bit, com cada *bit* valendo 0 ou 1.

Há duas vantagens no uso da orientação a *bits*; primeiro, é possível armazenar um *array* (coleção de elementos) de itens de dados binários, em que cada valor pode assumir verdadeiro (1) e falso (0). A partir desses dados lógicos, a memória pode ser utilizada de forma mais eficiente no armazenamento; segundo, existem situações em que se deseja manipular os *bits* de um item de dados, por exemplo se as operações de ponto flutuante forem implementadas diretamente no *software*. Assim, é necessário deslocar os *bits* significativos em algumas operações.

4.3 Ciclo indireto

A execução de uma instrução geralmente vai envolver um ou mais operandos lidos da memória; e, se algum endereçamento indireto for utilizado, então, acessos adicionais à memória serão necessários. Nessas situações, um endereço indireto será considerado como um estágio a mais no ciclo de instrução, conforme pode ser observado na figura a seguir.

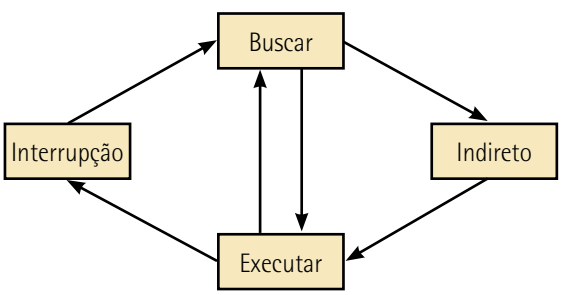


Figura 76 – Ciclo de instrução indireto

Nesse ciclo, a instrução é lida e examinada para determinar se há algum endereçamento indireto envolvido. Se houver, os operandos necessários são obtidos através de endereçamento indireto. Havendo interrupções durante a execução do ciclo de instrução, ela será processada antes da obtenção da próxima instrução.

4.4 Busca e execução de instruções

Em cada início de ciclo de instrução, o registrador *program counter* (PC – contador de programa) mantém o endereço da instrução a ser buscada. A cada busca, o PC será incrementado a fim de buscar a próxima instrução da sequência, ou seja, realizará a busca da próxima instrução com o endereço de memória mais alto. Por exemplo, considerando um computador em que cada instrução ocupará uma palavra de 16 bits, o contador de programa poderá estar alocado no endereço de número 300. Assim, o processador irá buscar a próxima instrução no local 300, e nos próximos ciclos de instrução ele irá buscar nos endereços 301, 302, 303 etc. Cada instrução é lida e carregada no registrador IR (registrador de instrução), que interpreta a instrução e realiza a ação desejada. Em geral, as ações a serem realizadas estão separadas em quatro categorias (STALLINGS, 2010):

- **Processador – memória**: quando os dados podem ser transferidos do processador para a memória ou da memória para o processador.

- **Processador – E/S**: quando os dados podem ser transferidos de, ou para, um dispositivo de entrada e saída.

- **Processamento de dados**: quando o processador realiza alguma operação lógica/aritmética sobre os dados.

- **Controle**: quando uma instrução especifica que a sequência de execução pode ser alterada. Por exemplo, o processador pode buscar uma instrução no local de memória 149, que especifica que a próxima instrução estará no local 182. Assim, o processador guardará essa informação, definindo que o contador de programa será definido como 182; dessa forma, o próximo ciclo de busca da instrução será retirado do endereço 182 em vez do endereço 150.

Em outro exemplo, baseando-se em um computador hipotético, que inclui algumas características como instrução/palavra de 16 bits de extensão, oferecendo 4 bits para o *opcode* e podendo haver até $2^4 = 16$ *opcodes* diferentes e até $2^{12} = 4.096$ palavras de memória endereçadas diretamente, registrador único (acumulador – AC) e memória também de 16 bits, o ciclo de instrução pode envolver mais de uma referência à memória e especificar uma operação de E/S. Através de um diagrama de estados, pode-se entender melhor o processo de busca e execução de uma instrução, que envolve as seguintes etapas:

- **Cálculo de endereço de instrução (*instruction address calculation* – IAC)**: estágio responsável pela determinação do endereço da próxima instrução a ser executada.

- **Busca da instrução (*instruction fetch* – IF)**: estágio responsável pela leitura da instrução do seu local da memória para o processador.

- **Decodificação da operação da instrução (*instruction operation decoding* – IOD)**: estágio responsável pela determinação do tipo de operação e os operandos a serem executados.

- **Cálculo do endereço do operando (*operation address calculation* – OAC)**: se uma operação envolve alguma referência a um operando na memória.

- **Busca do operando (*operation fetch* – OF)**: estágio em que ocorre a busca pelo operando da memória ou de algum dispositivo de E/S.

- **Operação dos dados (*data operation* – DO)**: estágio que realiza a operação solicitada pela instrução.

- **Armazenamento do operando (*operand store* – OS)**: estágio responsável por armazenar (escrever) o resultado na memória ou enviar o resultado para algum dispositivo de E/S.

Os estágios envolvidos no ciclo de instrução, observados na figura a seguir, envolvem uma troca entre o processador e a memória ou algum módulo de E/S.

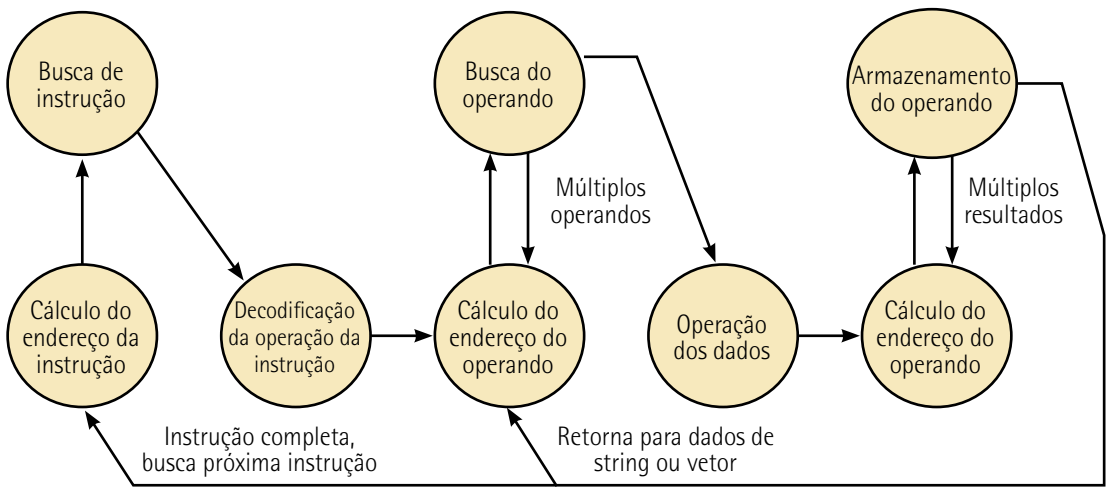


Figura 77 – Diagrama dos estágios do ciclo de instrução

O estado de cálculo do operando aparece duas vezes na figura, pois uma instrução pode envolver uma leitura e/ou uma escrita, utilizando somente um identificador de estado para as duas situações.

4.5 Formatos de instrução

Uma instrução consiste em um código de operação (*opcode*) formado por um conjunto de informações como o armazenamento dos resultados das operações realizadas.

As instruções operações geralmente possuem um formato bem definido, contendo além do *opcode*, um ou mais endereçamentos em uma mesma operação. Alguns modelos de formatos de instruções podem ser observados na figura a seguir.

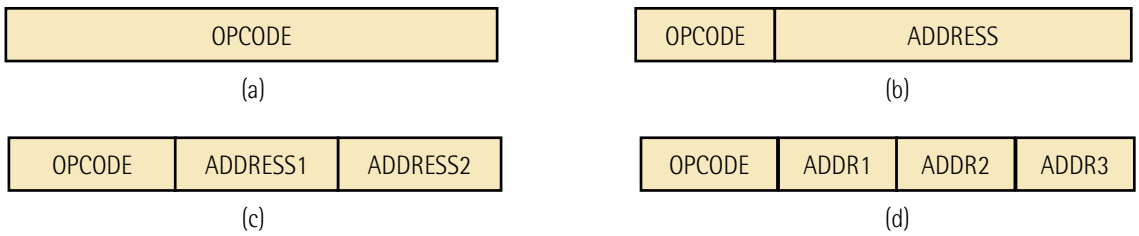


Figura 78 – Modelos simplificados de formatos de instrução

Na figura 78A, observa-se uma instrução que não possui endereçamento, sendo interpretada como uma instrução do tipo imediata, ou seja, não envolve outros registradores ou a memória principal, pois a própria instrução já está contida no registrador em uso. Já nas figuras 78B, C e D, nota-se que sempre há um ou mais endereços (*address*) relacionados ao *opcode*. Alguns computadores possuem instruções com o mesmo comprimento (em *bits*), porém em outros o tamanho pode ser diferente, variando inclusive em relação ao tamanho da palavra. Instruções que possuem o mesmo tamanho da palavra facilitam a decodificação, embora possam desperdiçar espaço para o armazenamento. Na figura a seguir, é possível observar as diferentes relações entre as instruções com diferentes comprimentos e larguras.

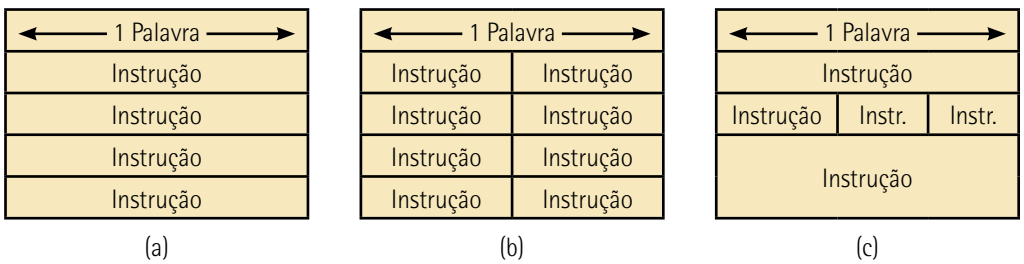


Figura 79 – Relações entre diferentes formatos de instrução

Na figura 79A, é notório que o tamanho da palavra é exatamente o mesmo tamanho das instruções, o que facilita a interpretação realizada pela unidade de controle do computador. Na figura 79B, as instruções possuem exatamente metade do tamanho da palavra, por exemplo para uma palavra de 64 bits tem-se uma instrução de 32 bits. Já na figura 79C, nota-se que há uma variação nos tamanhos das instruções, facilitando o armazenamento ao custo de uma maior dificuldade de padronização na decodificação de cada instrução.

Ao se projetar um computador, deve-se planejar criteriosamente qual será o tamanho das instruções, pois uma máquina moderna deve ser capaz de trabalhar com diferentes tamanhos de instruções para poder "sobreviver", por tempo suficiente, às mudanças de tecnologia impostas por diferentes gerações de processadores (TANENBAUM; AUSTIN, 2013).

A eficiência de determinada *instruction set architecture* (ISA) depende muito da quantidade de alterações que essa tecnologia passará ao longo dos anos. Por exemplo, se os acessos à memória forem rápidos, o uso de acesso a registradores ou memória baseado no conceito de pilha será eficiente, porém, se os acessos forem lentos, será mais eficiente se o computador possuir um conjunto muito grande de registradores como a CPU *advanced risc machine* (ARM).

Ao diminuir o tamanho das instruções, elas tornarão os processadores mais rápidos devido à menor quantidade de largura de banda (número de *bits* que a memória pode fornecer) de transmissão do processador para a memória principal.

Se a largura de banda de comunicação de uma memória *cache* para instruções *for t bps* (*bits* por segundo) e o comprimento médio da instrução for r bits, a memória *cache* poderá entregar no máximo t/r instruções por segundo. Esse é o limite superior da taxa a qual o processador pode executar instruções, o que também é limitado pelo comprimento da instrução.

Instruções mais curtas significam que o processamento será mais veloz, e como os processadores atuais são capazes de executar mais instruções a cada ciclo de *clock*, quanto maior for a quantidade de busca, decodificação e execução de instruções, mais potente é o computador. Dessa forma, o aspecto do tamanho da instrução e da *cache* de instrução é um fator muito importante no projeto de um computador e implicará diretamente no seu desempenho.

Outro critério em um projeto está relacionado ao espaço requerido pelo tamanho da instrução para que ela possa expressar todas as operações desejadas. Um computador com 2^n operações que tenha instruções menores do que n bits será impossível, pois simplesmente não haveria espaço suficiente no *opcode* para indicar qual instrução será necessária. Um terceiro critério se refere ao número de *bits* contidos em um campo de endereço.

4.5.1 Expansão de *opcodes*

Considere agora uma instrução de (n + k) bits com um *opcode* de k bits e um endereçamento único de n bits. Essa instrução pode permitir que ocorram até 2^k operações diferentes e utilizar 2^n células de memória endereçáveis. Em outra possibilidade, os mesmos (n + k) bits podem ser desmembrados em um *opcode* de (k – 1) bits e um endereço de (n + 1) bits, que significa apenas a metade do número de instruções, mas duas vezes mais memória endereçável, e que pode ser traduzido também como a mesma quantidade de memória, mas possuindo o dobro da resolução.

Um *opcode* de (k + 1) bits e um endereço de (n – 1) bits terá mais operações, porém à custa de um menor número de células endereçáveis ou uma menor resolução. Uma possibilidade para resolver esses impasses é conhecida como expansão de *opcodes*. Esse conceito pode ser observado da seguinte forma: considere um computador em que as instruções possuem 16 bits de comprimento e os endereços possuem 4 bits de comprimento, como mostra a figura a seguir.

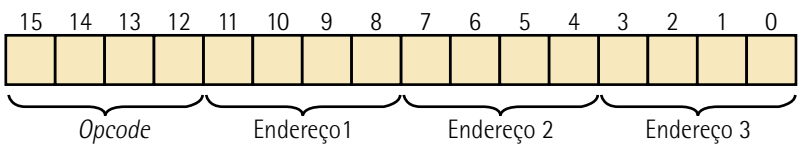


Figura 80 – *Opcode* de 4 bits e três campos de endereços de 4 bits

Considere nesse exemplo que a máquina possua 16 registradores também com endereços de 4 bits, em que ocorrem todas as operações aritméticas, como observado na figura a seguir.

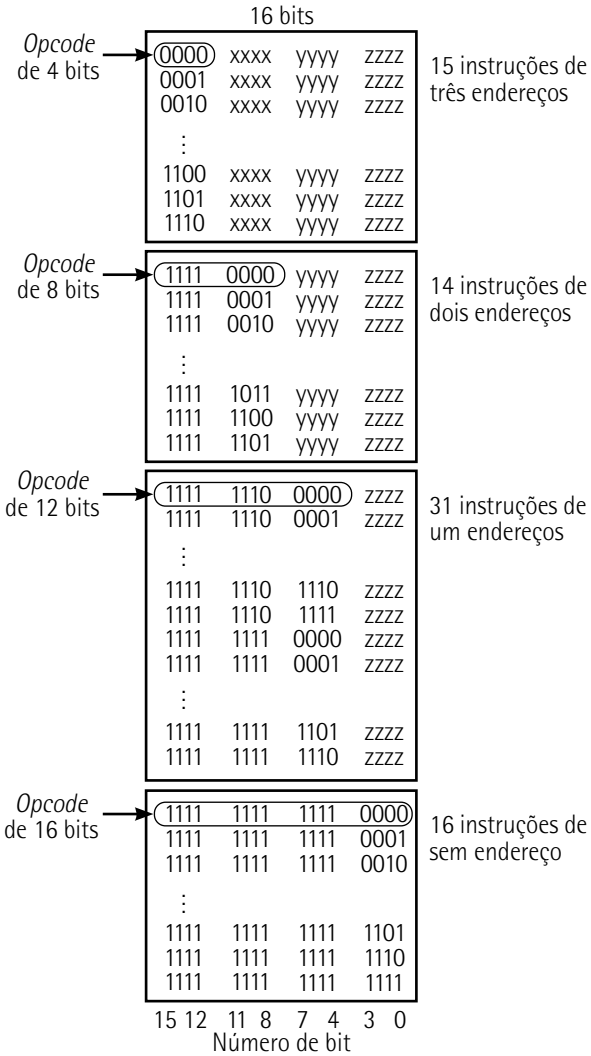


Figura 81 – Expansão de *opcode*

O *opcode* 15 significa que esse código de operação está contido nos *bits* que vão de 8 a 15 em vez de 12 a 15. Os *bits* de 0 a 3 e os de 4 a 7 formarão mais dois endereços. As 14 instruções de dois endereços possuem todas 1111 nos 4 bits da extrema esquerda e números que vão de 0000 a 1101 nos *bits* de 8 a 11. Instruções que possuem 1111 nos 4 bits da extrema esquerda e 1110 ou 1111 nos bits de 8 a 11 possuem um tratamento especial, pois seus *opcodes* podem estar alocados entre os *bits* de 4 a 15. Dessa forma, o resultado será 32 novos *opcodes*. Porém como são necessários somente 31 *opcodes*, o *opcode* codificado como 111111111111 é interpretado como significado de que está contido nos *bits* de 0 a 15, resultando em 16 instruções sem nenhum endereço.

Essa ideia de expansão de *opcode* pode demonstrar um compromisso entre o espaço para *opcodes* e o espaço para outras informações utilizadas no processo. Em termos práticos, os *opcodes* expandidos não são tão regulares como no exemplo; na verdade, há dois modos individuais para explorar a capacidade de uso dos diferentes tamanhos dos *opcodes*. Primeiro, todas as instruções que precisam de mais *bits* para especificarem outras funções; e segundo, o tamanho da instrução pode ser minimizado através da escolha dos *opcodes* mais curtos em instruções comuns e *opcodes* mais longos em instruções utilizadas raramente.

4.6 Instruções no Intel Core i7

As instruções no Intel Core i7 são geralmente de maior complexidade e irregulares, podendo ter até seis campos de comprimento variáveis, cinco dos quais são opcionais. O padrão das instruções do i7 é mostrado na figura a seguir.

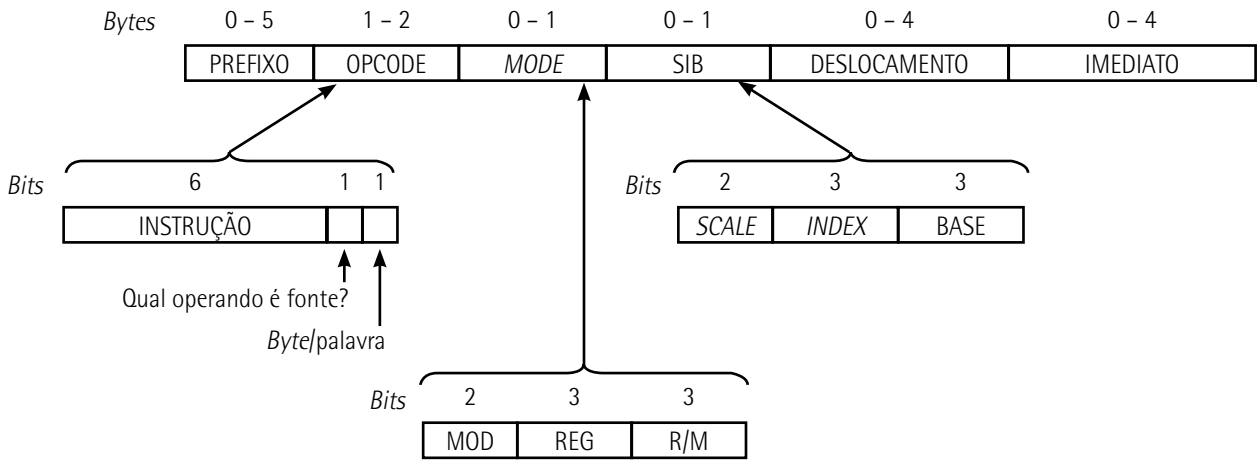


Figura 82 – Formatos de instrução no Intel Core i7

Como pode ser observado, a instrução pode conter alguns campos, como o prefixo para a identificação da instrução, o código da operação que relaciona a instrução com os operandos de origem, o modo de endereçamento, além de campos relacionados ao deslocamento de busca dos dados na memória principal ou em outros registradores. Nas primeiras arquiteturas da Intel, os *opcodes* possuíam 1 byte de prefixo e que também é interpretado como um *opcode* extra introduzido na frente da instrução a fim de alterar sua ação.

Os *bits* individuais nos *opcodes* contidos no i7 não contêm muitas informações sobre a instrução referida. Uma única estrutura no campo de *opcode* pode ser utilizada para um *bit* de ordem baixa com algumas instruções indicando o *byte*/palavra, e a utilização de um *bit* adjacente é usada para indicar se o endereço da memória é de origem ou destino.

Em geral, o *opcode* deve ser decodificado por completo para que seja possível determinar a qual classe de operação esta será executada e, consequentemente, qual deverá ser o comprimento da instrução. Isso pode dificultar as implementações que requerem um alto desempenho, pois será preciso uma decodificação extensiva antes mesmo de se determinar onde a próxima instrução irá começar.

Logo na sequência ao *byte* de *opcode* de referência a um operando, segue um segundo *byte* para informar tudo sobre o operando, sendo separados em 8 bits, subdivididos em um campo denominado MOD de 2 bits e dois campos com registradores de 3 bits (REG e R/M). Geralmente, os três primeiros *bits* desse *byte* são utilizados como extensão para o *opcode*, o que resulta em um total de 11 bits. Porém, o campo de modo de 2 bits significa que há apenas quatro modos de endereçamento dos operandos e um deles deve ser sempre um registrador.

4.7 Endereçamento de instruções

Os campos de endereços para instruções simples geralmente possuem um formato pequeno. É de interesse que haja uma grande variedade de técnicas de endereçamento envolvendo alguma troca de intervalos de endereços ou mesmo uma flexibilidade para realização do endereçamento. Além disso, existe uma necessidade de um grande número de referências à memória que estão contidas dentro da própria instrução e que devem ser atendidas, incluindo o próprio cálculo de endereçamento. Os principais modos de endereçamento para memória e/ou para registradores são: imediato, direto, indireto, por registradores, indireto por registradores, por deslocamento e endereçamento de pilha.

4.7.1 Endereçamento imediato

O endereçamento imediato é o formato mais simples para o endereçamento, em que o valor do operando está presente na própria instrução, por exemplo:

Operando = A

Esse tipo de endereçamento é geralmente utilizado para definir valores iniciais das variáveis. Via de regra, o número será armazenado em duas formas complementares com o *bit* à esquerda do operando para *bit* de sinal, de modo que, quando o operando é carregado em um registrador de dados, o *bit* de sinal será estendido para a esquerda até que o tamanho total da palavra de dados seja completado. Em outros casos, o valor em binário imediato é interpretado como um número inteiro e sem sinal (negativo ou positivo).

Uma das vantagens em se utilizar o endereçamento imediato é que não há necessidade de referenciar algum endereço da memória principal, pois a instrução já está contida no próprio operando, economizando ciclos de memória principal ou memória *cache*. Entretanto, a desvantagem desse tipo de endereçamento é em relação ao tamanho do campo de endereço, que pode ser considerado pequeno se comparado ao tamanho da palavra.

4.7.2 Endereçamento direto

No formato de endereçamento direito, o próprio campo de endereço contém o endereço efetivo do operando que será utilizado, por exemplo:

EA = A

Essa técnica de endereçamento era muito utilizada nas primeiras gerações dos computadores, mas não é utilizada nas arquiteturas atuais. Sua grande vantagem é requerer apenas uma referência de memória e nenhum cálculo de endereçamento especial. A principal desvantagem é sua limitação em relação ao espaço para o endereçamento dos operandos.

4.7.3 Endereçamento indireto

O endereçamento indireto possui como característica o tamanho do campo de endereço menor do que o tamanho da palavra, limitando dessa maneira o intervalo de endereços. Uma possível solução para essa limitação é ter um campo de endereço com referência ao endereço de uma palavra da memória que, por sua vez, conterá o endereço completo da operação a ser executada. Por exemplo:

EA = (A)

No exemplo da operação, os parênteses são interpretados como o conteúdo de um operando, no caso o A. Uma das vantagens visíveis desse tipo de endereçamento consiste no tamanho *N* da palavra, pois será disponibilizado um endereçamento de 2^N. A desvantagem é que a execução da instrução irá requerer duas referências da memória para que o operando seja obtido, ou seja, um para obter o endereço do operando e outra para obter o valor do operando. Caso o número de palavras que podem ser endereçados seja 2^N, o número de endereços diferentes que poderão ser referenciados em qualquer momento será limitado a 2^k, em que *k* será o tamanho do campo de endereço.

Em uma variação do endereçamento indireto, é possível o uso de vários níveis de endereços para acesso a diferentes conteúdos, como é observado no exemplo:

EA = (...(A)...)

4.7.4 Endereçamento de registradores

Esse tipo de endereçamento se assemelha ao endereçamento direto, sendo sua única diferença o campo de endereço se referir a um registrador em vez de um endereço na memória principal, determinado como:

EA = R

Em que R significa referência a um registrador. Por exemplo, se um campo de endereço de um registrador dentro de uma instrução for 3, então o registrador R3 será o endereço pretendido, e o valor do operando está contido em R3. Geralmente, um campo de endereço para referência de registradores possui de 3 a 5 bits, com um total de 8 a 32 registradores de uso geral.

Uma das vantagens do endereçamento por registradores consiste em utilizar apenas um pequeno campo de endereço contido na instrução, além de não necessitar de nenhuma referência à memória. Como desvantagem pode-se citar o espaço para o endereçamento de registradores ser pequeno e limitado. Quando há um uso muito grande de endereçamento por registradores em um conjunto de instruções, por consequência haverá um uso exacerbado dos registradores contidos no processador. Como o processador possui um número limitado de registradores, o uso exacerbado destes só faz sentido de modo eficiente. Se cada operando for inserido no registrador a partir da memória e utilizado apenas uma vez e depois enviado à memória principal, então seu uso não será otimizado, sendo necessário um passo intermediário no processo, aumentando uma etapa no ciclo de instrução. Por outro lado, se o

operando permanecer no registrador durante algumas operações, então haverá economia de etapas no ciclo de busca e execução dos operandos.

4.7.5 Endereçamento indireto por registradores

Da mesma maneira que o endereçamento por registradores é semelhante ao endereçamento direto (memória), o endereçamento indireto por registradores, por sua vez, é semelhante ao endereçamento indireto (memória). Nos dois casos, a única diferença é em relação ao campo de endereço, que difere ao se referenciar a um registrador ou à memória principal. Outra característica no endereçamento indireto por registradores é quanto a menor utilização de referências à memória, se comparado ao endereçamento indireto.

4.7.6 Endereçamento por deslocamento

O formato de endereçamento por deslocamento combina as capacidades de endereçamento direto e indireto por registradores e pode ser representado como:

EA = A + (R)

O endereçamento por deslocamento necessita que a instrução possua dois campos de endereçamento, dos quais um deverá ser explícito, ou seja, o valor contido em um campo de endereço como (A) deverá ser utilizado diretamente. Em um outro campo de endereço, será utilizada uma referência implícita baseada no *opcode*, que por sua vez refere-se a um registrador cujo conteúdo é adicionado em A de modo a produzir um endereço efetivo.

Há três usos comuns para o endereçamento por deslocamento: endereçamento relativo, endereçamento por registrador base e indexação.

Endereçamento relativo

O endereçamento relativo ocorre quando o registrador *program conter* (PC) é referenciado, ou seja, o endereço da próxima instrução será adicionado ao campo de endereço da operação. Assim, o endereço efetivo é o deslocamento relativo para a operação em uso. O uso de endereçamentos relativos economiza *bits*, pois a maioria das referências de memória já está na próxima instrução a ser executada.

Endereçamento por registrador base

Nessa variação, o registrador base contém um endereço de memória, e o campo de endereço contém um deslocamento, geralmente um número inteiro sem sinal. As referências aos registradores nesse caso podem ser de forma explícita ou implícita, e, também, é considerada uma forma eficiente para implementar a segmentação de instruções. Em algumas implementações de endereçamento, um único registrador de segmento pode ser utilizado implicitamente. Já em outras situações, o programador de *hardware* pode escolher um registrador para guardar o endereço base de um segmento, e a instrução deverá referenciá-lo explicitamente. Nessa última opção, se o tamanho do campo de endereço for k e o

número de possíveis registradores for N, então uma instrução pode referenciar qualquer uma de N áreas de 2^k palavras.

Indexação

Nessa situação, o campo de endereço referencia uma posição que contém um endereço na memória principal, e o registrador que for referenciado conterá um deslocamento positivo desse endereço. Devido ao fato de o campo de endereço ser considerado um endereço de memória na indexação, geralmente ele conterá mais *bits* quando comparado a um campo de endereço de uma instrução com endereçamento por registrador base. Apesar disso, o método para calcular EA será o mesmo utilizado para o endereçamento por registrador base e indexação, e nos dois casos a referência do registrador será algumas vezes explícita e outras, implícita, em diferentes processadores.

Uma opção na indexação é a permissão para que algum mecanismo possa efetuar operações iterativas. Por exemplo, suponha uma lista de números armazenados, iniciando-se na posição A. Suponha também que se queira adicionar 1 a cada elemento de uma lista. Será necessário obter cada valor, adicionar 1 a esse número e armazená-lo de volta. A sequência de endereçamentos efetivos necessários será A, A + 1, A + 2, A + 3..., até a última posição dessa lista. Ao se utilizar o endereçamento por indexação, essa tarefa é realizada de modo mais fácil, pois o valor A será armazenado no campo de endereço da instrução, e o registrador indexador será inicializado com zero. Após cada operação, o registrador indexador será incrementado por 1. Os registradores indexadores são comumente utilizados em tarefas iterativas, o que torna normal a necessidade de incrementar e decrementar o registrador indexador após cada referência a ele. Por se tratar de uma operação muito comum, alguns sistemas fazem isso de forma automática no próprio ciclo de instrução, o que é conhecido como autoindexação. Se forem utilizados registradores de uso geral para a indexação, a operação de autoindexação pode necessitar de uma sinalização feita por 1 bit dentro da própria instrução. Pode-se descrever como exemplo de autoindexação para o uso de incremento:

$$EA = A + (R)$$

$$(R) \leftarrow (R) + 1$$

Em alguns computadores, o endereçamento indireto e a indexação fazem parte de sua estrutura, sendo possível utilizar as duas possibilidades em uma mesma instrução. A indexação é executada antes ou depois da indireção. Se a indexação for executada depois da indireção, ela será chamada pós-indexação e pode ser representada como:

$$EA = (A) + (R)$$

O conteúdo dos campos de endereço é geralmente utilizado para acesso ao local de memória que contém o endereço direto. Esse endereço será então indexado por um valor contido no registrador. Tal técnica será útil para o acesso do interior de vários blocos de dados de formato fixo. Por exemplo, nos sistemas operacionais é necessário implementar um bloco de controle de processos (*program control block* – PCB), em que todos os dados referentes ao processo são armazenados. Assim, a operação executada

será a mesma, independentemente de qual bloco está sendo manipulado no momento, de forma que os endereços nas instruções que fazem referência ao bloco podem apontar para um local (valor=A) contendo um ponteiro para o início do PCB. Ao utilizar a pré-indexação antes da indireção, tem-se:

EA = (A + (R))

O endereçamento é calculado da mesma forma que na indexação simples, pois, nesse caso, o endereço calculado não contém o operando, mas o endereço de onde o operando se encontra. Por exemplo, pode-se citar uma tabela contendo múltiplos endereços de desvio. Em um certo ponto do programa, pode haver uma ramificação para uma série de posições diferentes que dependem de sua condição. Uma tabela de endereços poderá ser definida com início em A. Baseando-se na indexação dessa tabela, a localização desejada poderá ser encontrada.

4.7.7 Endereçamento de pilha

Uma pilha é um vetor linear de posições e é definida por um conjunto de elementos em que somente um deles pode ser acessado de cada vez. Uma pilha pode ter tamanho variável, e o ponto de acesso à pilha é denominado topo da pilha, assim como o último elemento da pilha é conhecido como base da pilha. Os itens adicionados ou removidos da pilha só podem ser acessados do seu topo e, por esse motivo, uma pilha também pode ser conhecida como último a entrar, primeiro a sair (*last in first out* – LIFO). A figura a seguir mostra algumas operações básicas de uma pilha.

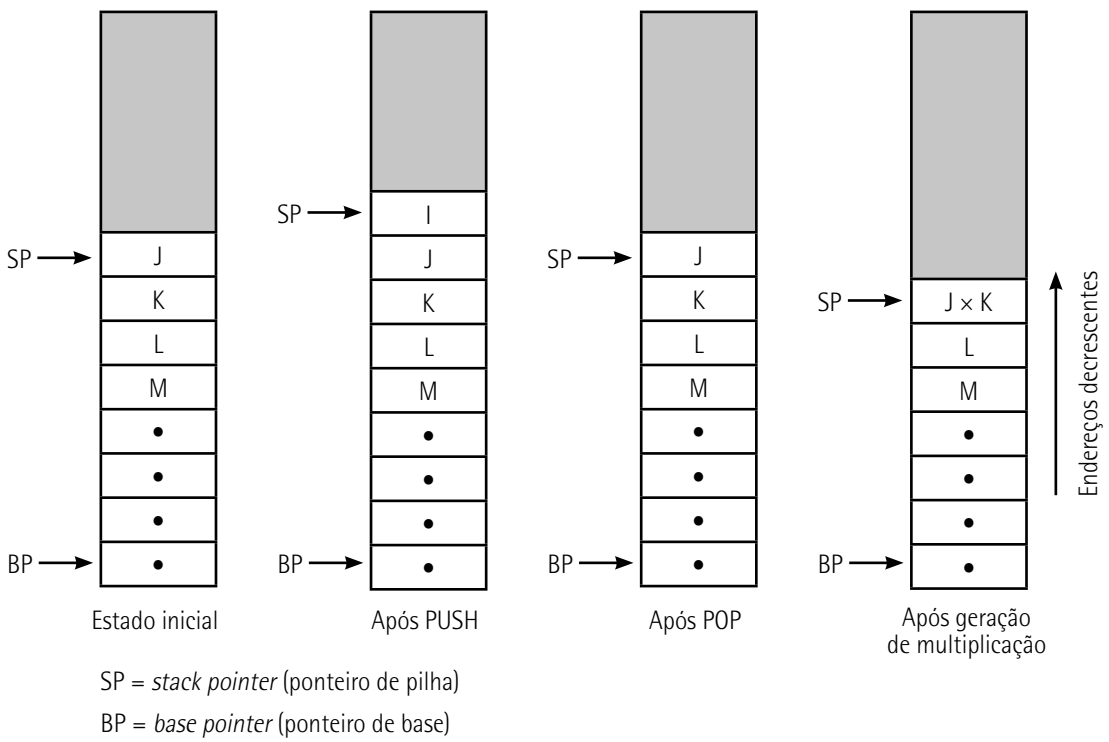


Figura 83 – Operações básicas de pilha

No exemplo da figura anterior, nota-se que a pilha já está constituída de alguns elementos em seu estado inicial. O estado se modifica através da operação PUSH, que tem como função acrescentar um elemento novo ao topo da pilha. A operação POP, na sequência, irá retirar o elemento do topo da pilha. Nesses dois casos, o topo da pilha se modifica de acordo com a operação realizada. Operadores binários também podem ser utilizados e exigirão dois operandos (multiplicar, dividir, somar, subtrair). Nesse caso, ao utilizar dois itens no topo da pilha como operandos, serão removidos dois itens relacionados à operação e o resultado é colocado novamente no topo da pilha. Em outras operações, conhecidas como unárias, exige-se apenas um operando (NOT) para modificar o topo da pilha. Algumas operações em pilhas podem ser observadas no quadro a seguir.

Quadro 2 – Operações orientadas à pilha

PUSH	Acrescenta um novo elemento ao topo da pilha
POS	Retira um elemento do topo da pilha
Operação unária	Realiza operação sobre o elemento do topo da pilha; substitui o elemento do topo pelo resultado
Operação binária	Realiza operação sobre os dois elementos do topo da pilha; exclui os dois elementos da pilha; coloca o resultado da operação no topo da pilha

Adaptado de: Stallings (2010, p. 322).

O quadro a seguir mostra, de forma resumida, as possíveis operações de cálculo de endereçamento discutidas até o momento.

Quadro 3 – Resumo dos modos básicos de endereçamento

Modo	Algoritmo	Principal vantagem	Principal desvantagem
Imediato	Operando = A	Nenhuma referência de memória	Magnitude de operando limitada
Direto	EA = A	Simples	Espaço de endereçamento limitado
Indireto	EA = (A)	Espaço de endereçamento grande	Múltiplas referências de memória
Registrador	EA = R	Nenhuma referência de memória	Espaço de endereçamento limitado
Indireto por registrador	EA = (R)	Espaço de endereçamento grande	Referência extra de memória
Deslocamento	EA = A + (R)	Flexibilidade	Complexidade
Pilha	EA = topo da pilha	Nenhuma referência de memória	Aplicabilidade limitada

Adaptado de: Stallings (2010, p. 331).

4.8 Interrupções

Todo computador oferece algum mecanismo para que operações que envolvam alguma comunicação com a memória, o processador ou dispositivos de E/S possam ser interrompidas em algum momento de sua operação.

Observação

Embora possa parecer ao usuário que uma máquina trabalhe melhor sem ser interrompida, as interrupções na verdade garantem uma operação otimizada de um computador.

No quadro a seguir, é possível observar os tipos mais comuns de classes de interrupções.

Quadro 4 – Classes mais comuns de interrupções

Programa	Gerada por alguma condição que ocorre como resultado da execução de uma instrução, como *overflow* aritmético, divisão por zero, tentativa de executar uma instrução de máquina ilegal ou referência fora do espaço de memória permitido para o usuário
Timer	Gerada por um *timer* dentro do processo; isso permite que o sistema operacional realize certas funções regularmente
E/S	Gerada por um controlador de E/S para sinalizar o término normal de uma operação ou para sinalizar uma série de condições de erro
Falha de *hardware*	Gerada por uma falha, como falta de energia ou erro de paridade de memória

Adaptado de: Stallings (2010, p. 60).

Como mencionado anteriormente, as interrupções são fornecidas como uma forma de melhorar a eficiência do processamento. Por exemplo, suponha que um processador esteja transferindo dados a um dispositivo de impressão, utilizando o esquema de ciclo de instrução mostrado na figura 75. Após cada operação para realização de escrita (gravação), o processador deverá permanecer ocioso até que o dispositivo de impressão o alcance. O tempo dessa pausa deve ser da ordem de centenas ou milhares de ciclos de instrução que não envolverão memória, o que ocasiona um desperdício de uso do processador que ficará aguardando instruções nesse período. A figura a seguir ilustra essa situação, em que o programa de usuário realiza uma série de chamadas WRITE intercaladas com algum processamento. Nessa figura, é possível observar que os segmentos de código 1, 2 e 3 referem-se às sequências de instruções que não envolvem alguma comunicação com dispositivos de E/S. Nessa situação, o programa de E/S consiste em três seções, respectivamente:

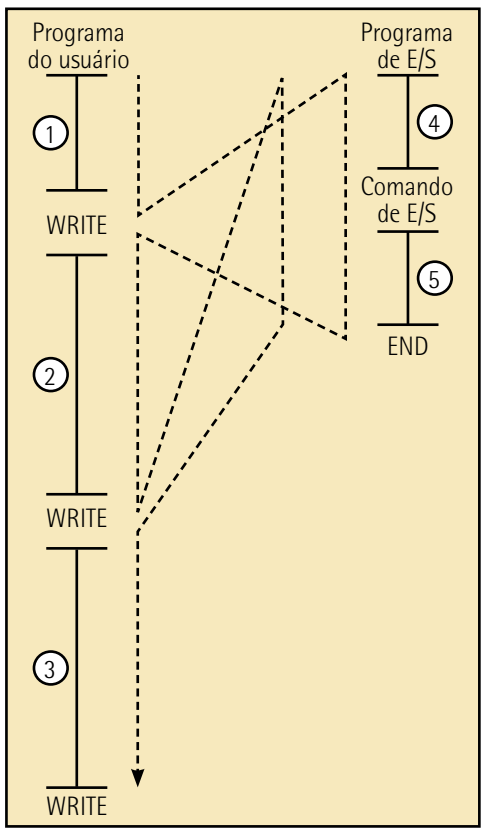
- Uma sequência de instruções, denominadas de 4, como mostra a figura a seguir, é utilizada a fim de preparar para a operação de E/S, o que pode incluir a cópia dos dados para a saída em um *buffer* (memória auxiliar temporária).

- O comando de E/S, sem o uso de interrupções, é emitido, e o programa precisa esperar até que o dispositivo de E/S esteja pronto para realizar a função desejada ou sondar, de forma periódica, se a operação de E/S terminou.

- A instrução seguinte, rotulada de 5, completará a operação, o que pode incluir uma notificação (*flag*) para indicar o sucesso ou falha na operação.

Lembrete

As operações de E/S podem levar um tempo longo para serem finalizadas, e o programa de E/S geralmente fica esperando que a operação termine.

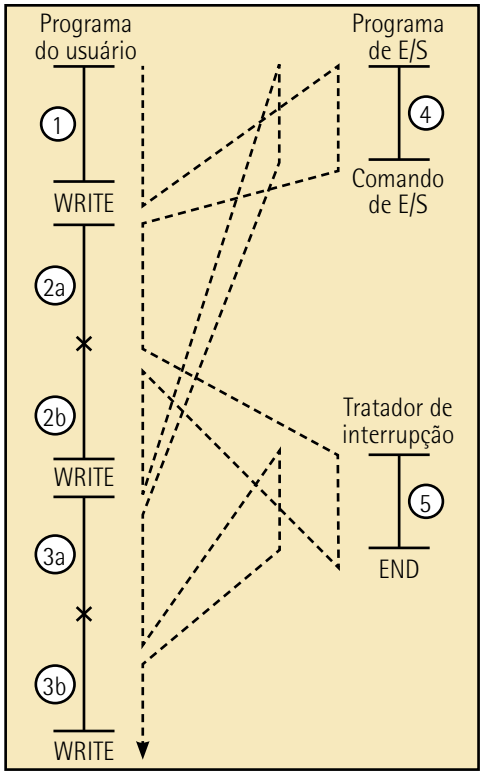


(a) Sem interrupções

Figura 84 – Fluxo de controle de programa sem interrupções

4.8.1 Interrupções e o ciclo de instrução

O uso de interrupções durante o ciclo de operação do computador pode de fato otimizar o desempenho do processador. Considere o fluxo de controle de um programa da figura a seguir. Nessa situação, o programa de usuário alcança um ponto em que é realizada uma chamada sistema (*system call*) na forma de uma chamada WRITE. O programa de E/S será invocado depois que algumas poucas instruções tiverem sido executadas. Então, o controle retorna ao programa de usuário enquanto o dispositivo externo está ocupado aceitando/imprimindo dados oriundos da memória principal. A operação de E/S será realizada simultaneamente com a execução de instruções no programa de usuário. Quando algum dispositivo de E/S estiver pronto para aceitar mais dados do processador, o módulo de E/S para o dispositivo externo enviará um sinal de interrupção ao processador de modo que o processador responderá suspendendo a operação do programa em execução, desviando para um programa do sistema operacional (tratador de interrupção) atender a algum dispositivo de E/S, retomando a execução original depois que o dispositivo for atendido, conforme mostra o diagrama da figura a seguir.



(b) Interrupções; curta espera pela E/S

Figura 85 – Fluxo de controle de programa com interrupções de curta espera

Uma interrupção pode ser definida de forma simplificada também como uma quebra na sequência de execução normal de algum *software* ou *hardware*. Quando o tratamento da interrupção tiver terminado, a execução retornará do ponto onde foi interrompida. O processador e o sistema operacional são os responsáveis pela suspensão do programa de usuário para depois retorná-lo ao mesmo ponto. Assim, o programa de usuário não precisará conter qualquer outro código especial para acomodar as interrupções, como mostra o esquema da figura a seguir.

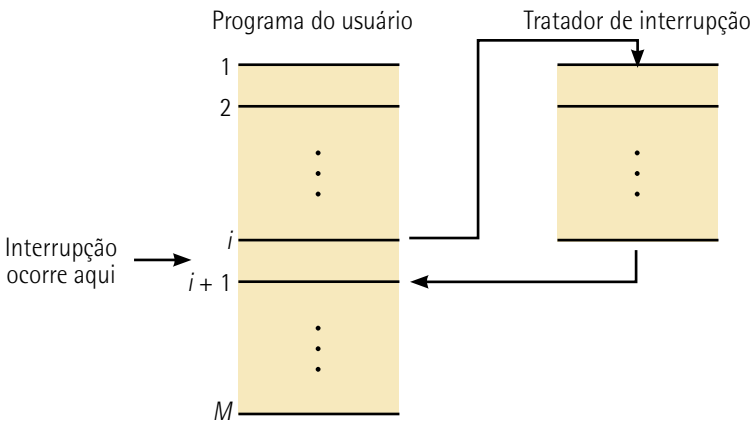


Figura 86 – Transferência de controle de tarefas via interrupções

Para que as interrupções ocorram, um ciclo de interrupção será acrescentado ao ciclo de instrução simples, como pode ser observado na figura a seguir.

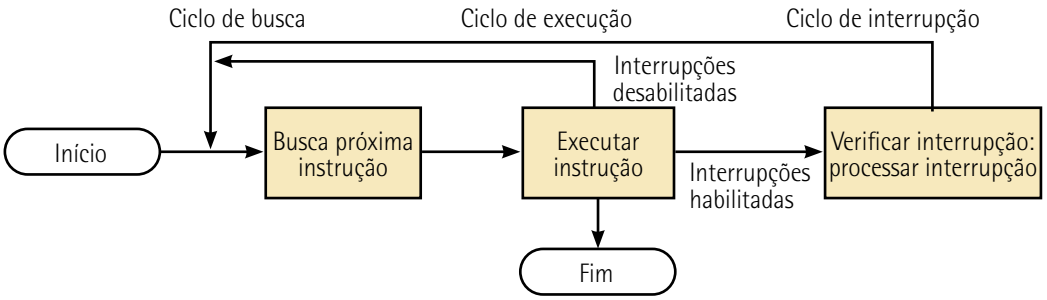


Figura 87 – Ciclo de instruções com interrupções

No ciclo de instrução com interrupções, o processador verificará se houve algum sinal de interrupção. Se nenhuma interrupção ocorrer ou estiver pendente, o processador pode seguir para a próxima etapa do ciclo de busca, que é ler a próxima instrução do programa em execução. Porém, se alguma interrupção estiver pendente, o processador fará as seguintes tarefas (STALLINGS, 2010):

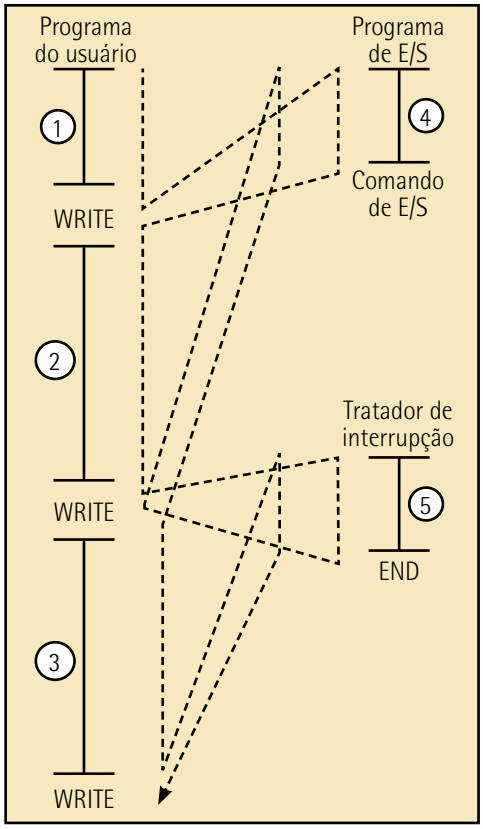
- Suspender o programa que está em execução e salvar o endereço da próxima instrução (indicada pelo *program counter* – PC) a ser executada ou qualquer outro dado relacionado à operação em atividade.

- Armazenar no *program counter* o endereço inicial da rotina que irá tratar a interrupção.

Como observado na figura anterior, o ciclo de busca obtém a primeira instrução da rotina de tratamento de interrupção que irá atender à interrupção requerida. O tratador de interrupção então determina qual a finalidade da interrupção e realiza as ações necessárias para sua realização. No exemplo da figura 85 (fluxo de interrupção de curta espera), o tratador determina qual módulo de E/S foi o responsável pelo sinal de interrupção e desvia a solução para um programa que escreverá os dados no dispositivo de E/S. Quando a rotina de tratamento de interrupção termina, o processador poderá retomar a execução do programa de usuário que estava em operação antes da interrupção, justamente do ponto de parada.

Esse tipo de operação de tratamento de interrupção geralmente envolve o uso excessivo de memória (*overhead*); apesar disso, devido à quantidade de tempo relativamente grande envolvida no processo, o uso indevido de tempo do processador poderia ser desperdiçado simplesmente pela longa espera de um sinal que envolva a comunicação do processador com algum dispositivo de E/S, tornando muito mais eficiente o uso de interrupções. O tempo exigido para a operação de E/S será relativamente mais curto (interrupção de curta espera), ou seja, menos do que o tempo total para completar a execução das instruções entre as operações de escrita no programa de usuário. Situações com respostas mais lentas, como dispositivos de impressão, resultarão em operações de E/S com muito mais tempo de execução para uma sequência de instruções do usuário.

A figura a seguir mostra uma situação em que o programa de usuário alcança a segunda chamada WRITE antes do término da primeira operação de E/S. O resultado dessa operação é que o programa do usuário ficará travado nesse ponto, de modo que, quando a operação de E/S anterior tiver terminado, essa nova chamada WRITE será processada, fazendo com que uma nova operação de E/S seja iniciada.



(b) Interrupções; curta espera pela E/S

Figura 88 – Fluxo de controle de programa com interrupções de longa espera

Já a figura a seguir mostra o diagrama de ciclo de instruções revisado, incluindo o processamento do ciclo de interrupção. Como pode ser observado, o processo se inicia geralmente com o cálculo do endereço da próxima instrução. Após esse primeiro estágio, são realizadas a busca da instrução e a decodificação dos operandos envolvidos no processo.

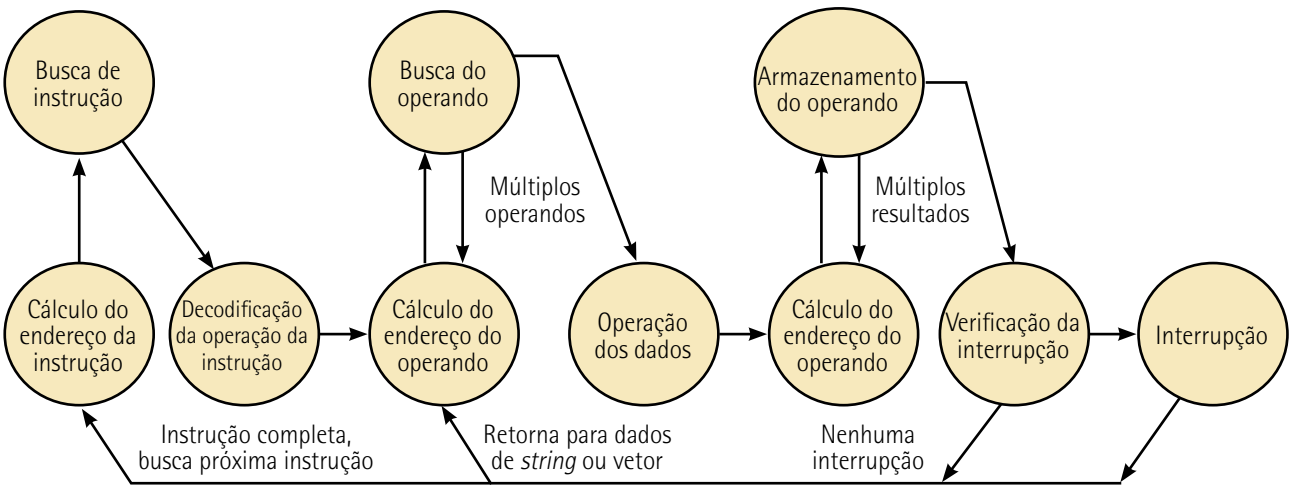


Figura 89 – Diagrama de ciclo de instruções com interrupções

Logo em seguida, é realizado o cálculo do endereço de onde estão esses operandos para que sejam buscados e, por fim, a operação de dados seja realizada. O processo finaliza com um novo cálculo do endereço para que seja possível armazenar o resultado em um novo operando. Note que, na figura, durante esse ciclo inicial não houve nenhuma interrupção, o que poderia ter ocorrido em qualquer momento do ciclo. Se houver uma interrupção, é possível observar que ela será atendida prioritariamente e um novo ciclo será iniciado, com a busca da instrução de interrupção requerida. Após o término do tratamento da interrupção, o ciclo voltará a ser executado do estágio em que havia sido interrompido.

4.8.2 Interrupções múltiplas

Lembrete

As interrupções abordadas até o momento focaram na ocorrência de interrupções simples, ou seja, que ocorram uma a uma.

Suponha agora que ocorram múltiplas interrupções, em que um programa receberá dados de um controlador de linha de comunicações e precisará imprimir o resultado disso, de forma que a impressora gere uma interrupção toda vez que dados unitários chegarem. Nessa situação, duas técnicas podem ser utilizadas para tratar múltiplas interrupções. Na primeira, desativam-se as interrupções enquanto uma interrupção já estiver sendo tratada, o que significa que o processador poderá ignorar qualquer novo sinal de requisição de interrupção. Se alguma interrupção ocorrer durante esse período, ela permanecerá pendente e será averiguada pelo processador após ele ter habilitado novamente as interrupções. Depois que a rotina de tratamento de interrupção tiver terminado, as interrupções serão novamente habilitadas antes que o programa de usuário dê continuidade do ponto de parada, de modo que o processador verificará se houve interrupções adicionais durante o período em que as interrupções adicionais foram desabilitadas. Apesar de simples, essa técnica é muito boa, pois as interrupções são tratadas de forma sequencial, como pode ser observado na figura a seguir. Uma desvantagem dessa técnica é que ela não considera a prioridade relativa ou mesmo necessidades de tempo crítico em que, por exemplo, a entrada chega à linha de comunicações e precisa ser absorvida rapidamente para que seja liberado mais espaço para outras entradas, de forma que se o primeiro lote de entrada não for processado antes da chegada do segundo lote, alguns dados ou todos poderão ser perdidos.

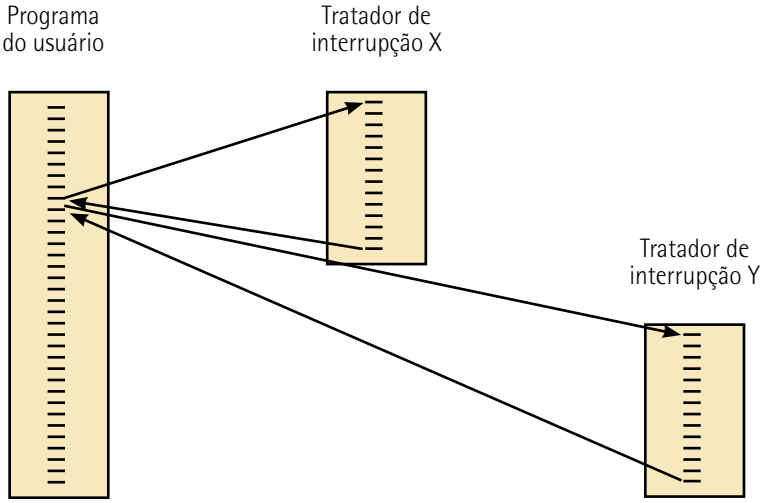


Figura 90 – Transferência de controle de execução
de processos com interrupção sequencial

Em uma segunda técnica, definem-se as prioridades de interrupção a fim de permitir que uma interrupção de maior prioridade possa ter um tratamento de interrupção distinto ao de menor prioridade, inclusive ocasionando a pausa deste último, como se observa na figura a seguir.

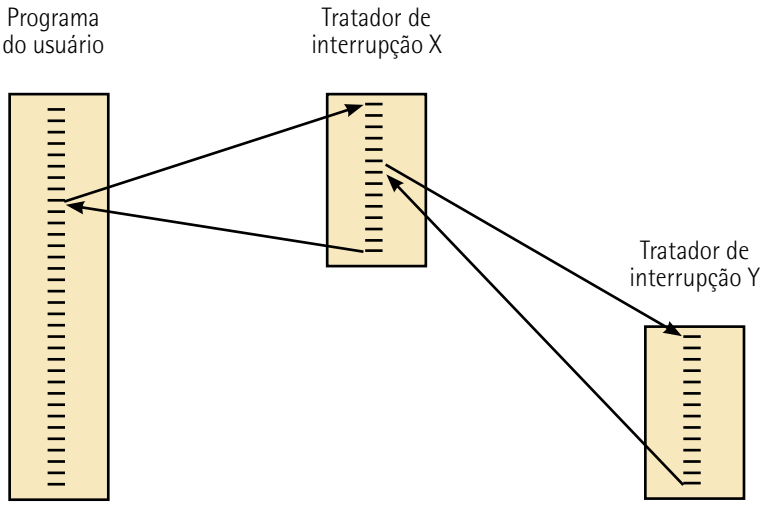


Figura 91 – Transferência de controle de execução
de processos com interrupção aninhada

Por exemplo, considere um sistema formado por três dispositivos de E/S (impressora, disco rígido e linha de comunicações) constituídos com as prioridades 2, 4 e 5, respectivamente. Um usuário inicia uma tarefa em t = 0, e em t = 10 ocorre uma interrupção ocasionada pela impressora. A informação do usuário é posta na pilha do sistema, e a execução continua na rotina de serviço de interrupção, como mostra a figura a seguir.

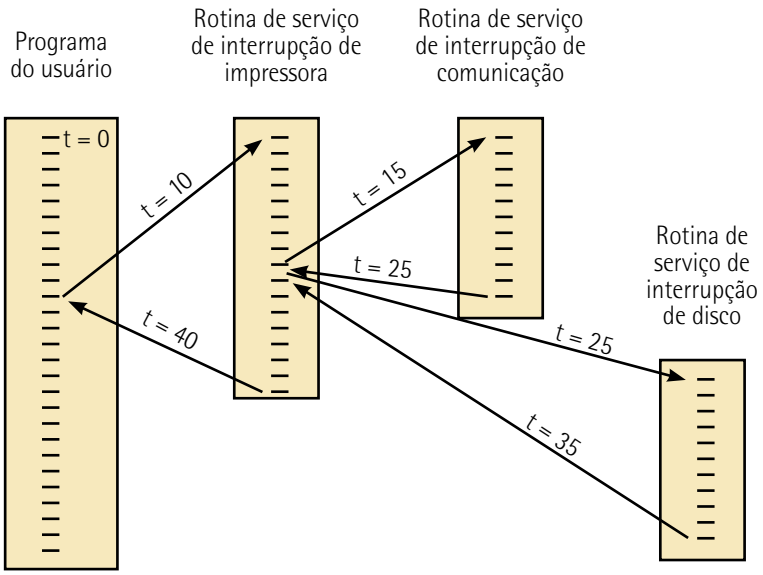


Figura 92 – Sequência de tempo para múltiplas interrupções aninhadas

Enquanto essa rotina ainda está em execução, em t = 15 ocorre uma interrupção de comunicação, de forma que, como a linha de comunicação possui prioridade maior se comparada à impressora, a interrupção será aceita. Sendo aceita a interrupção da impressora, seu estado é colocado na pilha de execução contínua. Só que, para complicar a situação, ocorre uma nova interrupção de disco em t = 20 durante o processo em execução (serviço de comunicação). Como esse tipo de interrupção tem menor prioridade (disco rígido) do que o serviço de comunicação, ela será retida, e o serviço de comunicação será executado até o final. Quando o serviço de comunicação termina em t = 25, o estado anterior do processador (impressora) é restaurado. Entretanto, antes que uma instrução da rotina seja executada, o processador aceita a interrupção de disco rígido que estava em espera e que possuía maior prioridade que a impressora, transferindo o controle para a rotina de serviço de interrupção. Por fim, quando t = 40 (última rotina), o controle retorna para o programa de usuário, aguardando novas instruções.

4.9 *Pipeline* de instruções

O *pipeline* de instrução pode ser entendido como um processo de linha de montagem em uma fábrica, como na figura a seguir. Um produto na linha de montagem passa por vários estágios os quais podem ser trabalhados de forma simultânea. Esse processo ficou conhecido como *pipelining*, pois, como em uma tubulação, novas entradas serão aceitas em um lado da linha de fabricação antes que as entradas aceitas anteriormente apareçam finalizadas na saída do outro lado.

Figura 93 – Linha de montagem de placas-mãe

Aplicando esse conceito para computadores, sabe-se que uma instrução também possui vários estágios, como observado na figura. Nela, há uma quebra do ciclo de instruções em dez tarefas que ocorrem sequencialmente, de forma que há uma necessidade intrínseca para que o conceito de *pipeline* seja aplicado a fim de otimizar o processo. Considere então dividir o processamento da instrução em apenas dois estágios: ler instrução e executar instrução. Nessa situação, haverá momentos durante a execução de uma instrução em que a memória principal não será acessada. Essa fatia do tempo poderia ser utilizada então para obter a próxima instrução de forma paralela e sem interferir na execução da instrução atual, como está exemplificado na figura a seguir.

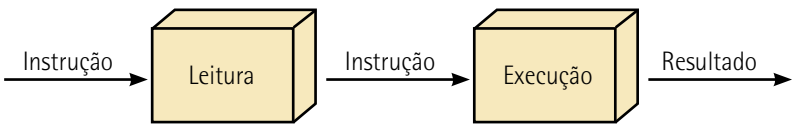


Figura 94 – *Pipeline* de instruções de dois estágios simplificados

Como observado, o *pipeline* possui dois estágios independentes, em que o primeiro deles obtém a instrução e a coloca em espera (*buffer*). Após o segundo estágio estar liberado, o primeiro estágio repassa para ele a instrução do *buffer*. Enquanto o segundo estágio está em execução, o primeiro estágio aproveita qualquer ciclo de memória que não está sendo utilizado para obter a próxima instrução a ser executada e a coloca no *buffer*. Esse processo ficou conhecido como busca antecipada (*prefetch*) ou busca sobreposta (STALLINGS, 2010). Uma possível desvantagem nessa abordagem de tratamento de instruções em paralelo é a grande utilização de registradores que são utilizados para guardar não só instruções, mas também dados utilizados entre os estágios. Apesar disso, o processo de *pipelining* irá acelerar a execução das instruções, e se os estágios de leitura e execução tiverem o mesmo tempo de duração, o ciclo de instrução poderá ser reduzido pela metade do tempo. Porém, ao verificar o esquema da figura a seguir, nota-se que seria pouco provável dobrar a taxa de execução devido aos seguintes fatores:

- Obviamente que o tempo de execução é maior do que o tempo de leitura da instrução, de forma que o estágio de leitura poderá ter que esperar por algum tempo antes de esvaziar seu *buffer*.

- Outro fator é que uma instrução de desvio condicional pode ocorrer durante a execução da instrução, o que faz com que o endereço da próxima instrução, que ainda será obtida, não seja conhecido. Assim, o estágio de leitura precisará esperar até que receba o endereço da próxima instrução contida no estágio de execução, daí o estágio de execução também precisará esperar até que a próxima instrução seja recebida.

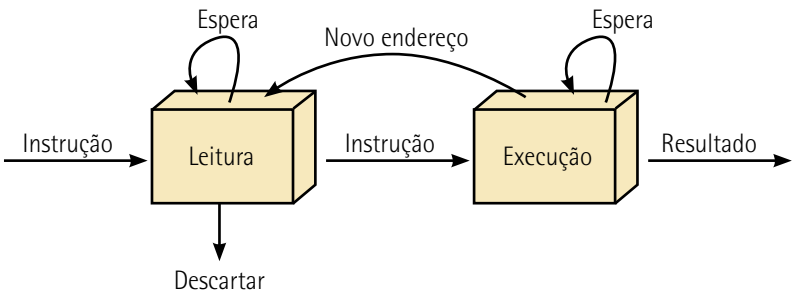


Figura 95 – *Pipeline* de instruções de dois estágios expandidos

Nessa situação de desvio condicional, a instrução passa do estágio de leitura para o estágio de execução. O estágio de leitura obtém então a próxima instrução na memória principal após a instrução de desvio; e se o desvio requerido não for atendido, nenhum tempo será perdido durante o processo. Porém, se o desvio for atendido, a instrução ativa no processo poderá ser descartada e uma nova instrução será lida e executada. É certo que esses fatores podem reduzir a eficiência do processo de *pipeline* de dois estágios, mas algum ganho na aceleração ainda ocorrerá, de forma que, para obter mais velocidade, o *pipeline* deverá conter mais estágios a fim de tratar possíveis atrasos em estágios de forma individual. Para que o *pipeline* obtenha maior desempenho, mais estágios serão necessários, de forma que os estágios possam ser decompostos e com menos operações em cada ciclo. Essa divisão de estágios de *pipeline* varia em tamanho, de acordo principalmente com a capacidade do *hardware* de conter ou não mais registradores em seu processador. Um sistema de *pipeline* padrão é constituído de seis estágios, divididos em:

- **Buscar instrução (*fetch instruction* – FI)**: estágio que tem como função ler a próxima instrução esperada em um *buffer*.

- **Decodificar instrução (*decode instruction* – DI)**: estágio responsável por determinar o código da operação (*opcode*) e a especificação dos operandos.

- **Calcular operandos (*calculate operand* – CO)**: estágio responsável pelo cálculo do endereço de cada operando. Essa operação pode envolver o endereçamento por deslocamento, endereçamento indireto por registrador, endereçamento indireto pela memória e outros tipos de endereçamento.

- **Obter operandos (*fetch operands* – FO)**: estágio que obtém cada operando da memória principal. Nesse caso, os operandos que estão nos registradores não precisarão ser lidos da memória.

- **Executar instrução (*instruction execution* – IE)**: excuta a operação indicada após a obtenção dos operandos e armazena o resultado em algum local específico.

- **Escrever operandos (*write operands* – WO)**: estágio responsável pelo armazenamento do resultado da operação na memória principal.

A partir dessa decomposição baseada em seis estágios, é possível deduzir que eles terão, de forma fictícia, o mesmo tempo de duração. A figura a seguir ilustra essa situação hipotética de processamento paralelo utilizando o *pipeline*; vê-se que é possível reduzir o tempo de execução das instruções em paralelo (tempo 14) em vez de executá-las todas em série (enfileiradas).

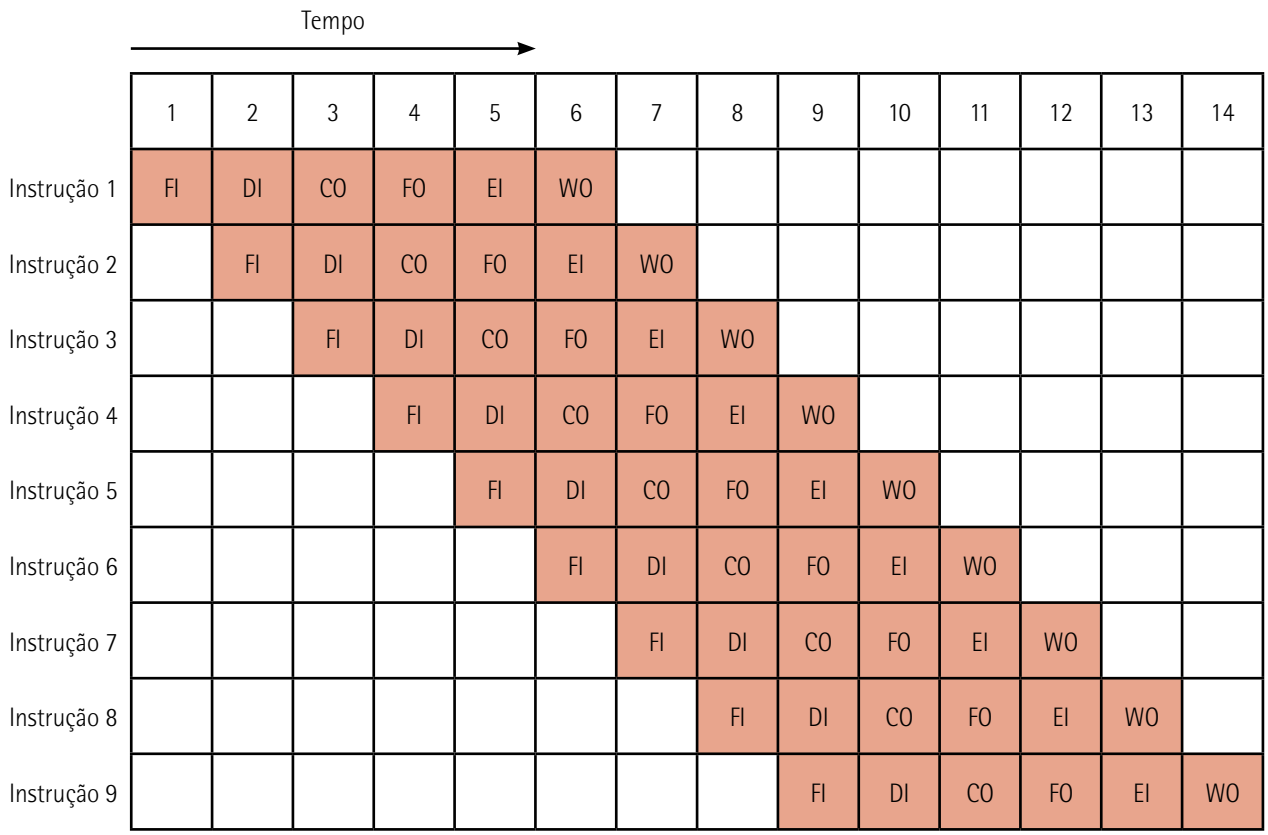


Figura 96 – Diagrama temporal para operação de instruções em paralelo

O diagrama apresentado na figura anterior mostra que cada instrução necessariamente passa pelos seis estágios do *pipeline* e que todos os estágios podem ser executados em paralelo, supondo-se também que não haverá em nenhum momento algum conflito de memória. Mas os estágios FI, FO e WO envolvem acesso à memória, o que implica que nesses casos os acessos podem ocorrer de forma simultânea, o que fará com que sistemas de memória não permitam tais acessos simultâneos. Porém, o valor desejado na operação pode estar contido na memória *cache* ou mesmo os estágios FO ou WO podem estar nulos, causando diversos conflitos, mas que por sua vez não irão desacelerar o processo de *pipeline*.

Observação

Outros fatores poderão limitar o desempenho no *pipeline*. Em uma situação real, os seis estágios não terão a mesma duração, ocasionando espera em vários estágios diferentes.

Uma instrução de desvio condicional pode invalidar várias leituras de instruções, de modo semelhante ao mostrado no esquema da figura a seguir.

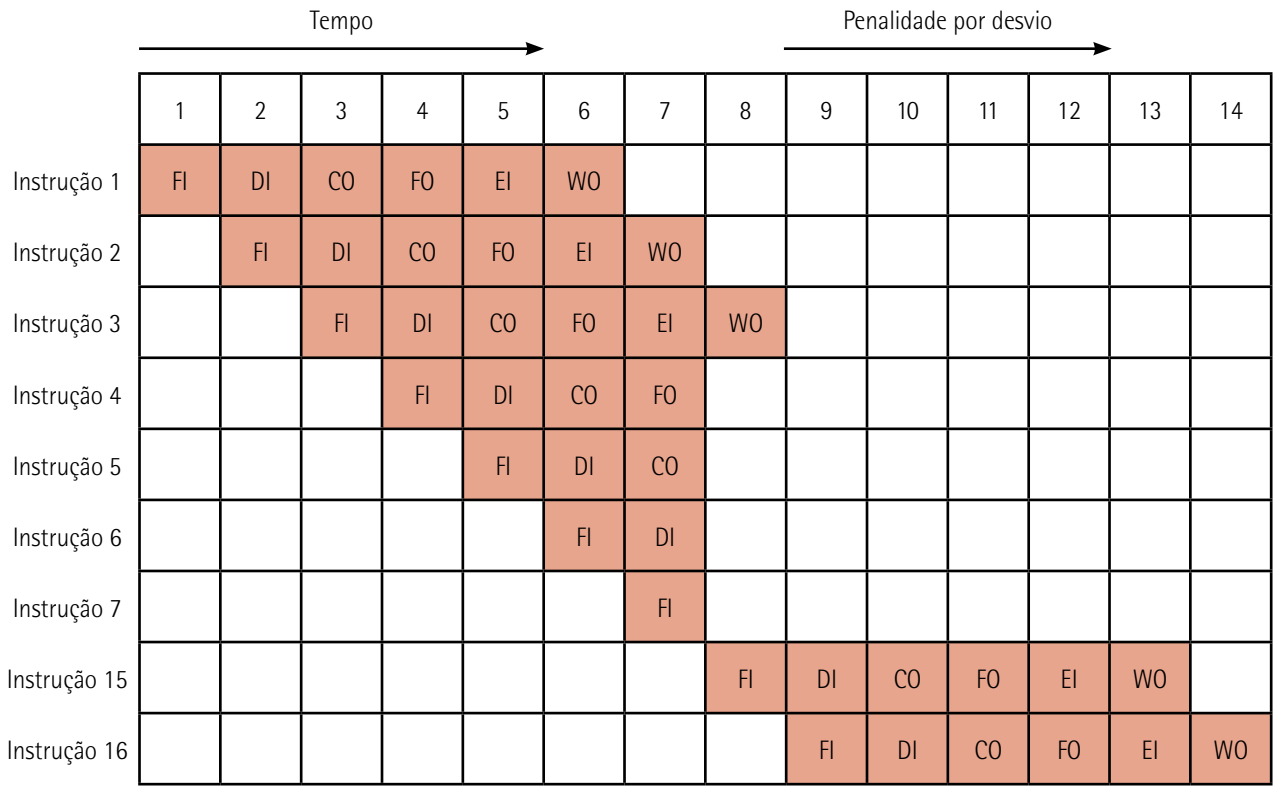


Figura 97 – Diagrama temporal para operação de instruções em paralelo com desvio condicional

Como ilustrado na figura anterior, o tempo 3 (eixo X), representa um desvio condicional para a instrução 15 (eixo X). Antes de a instrução ser executada não há como prever qual instrução virá na sequência. Dessa forma, o *pipeline* irá carregar a próxima instrução (4) na sequência e prosseguirá com o carregamento das próximas instruções. Na figura 96 não ocorreram desvios; sendo assim, houve um benefício total do aumento do desempenho do *pipeline*. Entretanto, na figura 97 o desvio ocorre sem ser interpretado até o fim da unidade de tempo 7 (eixo X), sendo que nesse ponto será necessário que o *pipeline* seja limpo das instruções que não serão úteis. Assim, durante o tempo 8 (eixo X) a instrução 15 (eixo Y) entra no *pipeline*, fazendo com que nenhuma instrução seja completa durante as unidades de tempo de 9 a 12 (eixo X). Isso caracteriza uma penalidade de desempenho do *pipeline*, pois não foi possível, durante todo o processo, antecipar o desvio como sendo uma interrupção a ser tratada pelo tratador de interrupção e depois retornar ao ponto da

parada do processo. Como observado, o simples fato de limpar as instruções do *pipeline* faz com que os dados e instruções já carregados no processo sejam perdidos (STALLINGS, 2010). A figura a seguir mostra, na forma de um fluxograma, a lógica necessária no funcionamento do *pipeline* para computar desvios e interrupções.

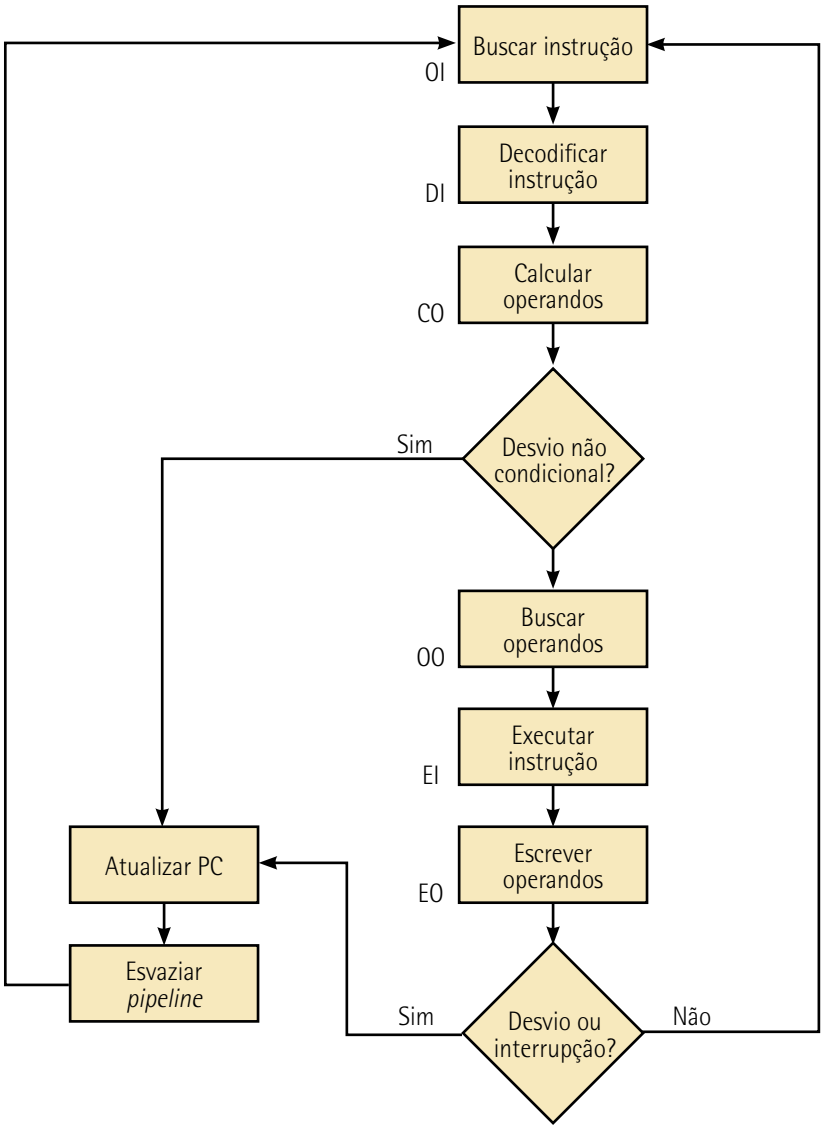


Figura 98 – *Pipeline* de seis instruções com desvio condicional

Algumas situações que não apareciam em *pipeline* com dois estágios surgem com o aumento do número de estágios. O estágio calcular operandos (CO) pode ter que depender do conteúdo de um registrador que é alterado por uma instrução anterior que ainda esteja contida no *pipeline*. Outros conflitos também podem ocorrer com registradores e a memória principal, de modo que o sistema operacional que gerencia todos os dispositivos também precisa lidar com esses conflitos.

A partir dos exemplos de *pipeline* discutidos até aqui, é possível concluir que, quanto maior o número de estágios do *pipeline*, maior será a taxa de execução de instruções, aumentando seu desempenho de processamento em paralelo. Além disso, outras conclusões podem ser obtidas, como:

- Em cada estágio do *pipeline* existe algum esforço extra envolvido na movimentação de dados de um sistema de *buffer* para outro e para efetuar a entrega de dados. Esse esforço desacelera o tempo total de execução de uma única instrução, o que melhora o desempenho em situações em que as instruções sequenciais são dependentes umas das outras ou mesmo pelo uso demasiado de desvios de acesso à memória principal.

- A lógica de controle necessária para lidar com a dependência de memória e registradores e para otimizar o uso do *pipeline* aumenta em função do número de estágios. Outro fator a ser considerado é o aumento de tempo gasto pelos *buffers* de *pipeline*, que aumentarão o tempo total do ciclo de instruções.

Por fim, a conclusão é que o *pipeline* de fato é uma técnica muito poderosa para melhorar o desempenho de um processador; em contrapartida, o projeto requer muitos cuidados para que sejam alcançados bons resultados, devido à atual complexidade dos sistemas.

4.9.1 Desempenho do *pipeline*

Para realizar o cálculo do desempenho do *pipeline*, inicialmente é necessário definir quais parâmetros estão envolvidos no processo. O tempo de ciclo de uma instrução de *pipeline* é definido como o tempo necessário para que uma instrução avance um estágio no processo, de modo que cada coluna nas figuras 96 e 97 simbolizava um tempo de ciclo. Assim, o tempo de ciclo do *pipeline* pode ser definido como:

$\tau = \max[\tau_i] + d$ para $1 < i < k$

Em que:

- τ_i é o tempo de demora de resposta do circuito no estágio *i* do *pipeline*;

- τ_m é o tempo de demora máximo de resposta do estágio;

- k é o número de estágios no *pipeline*;

- d é o tempo de resposta do ciclo utilizado para avançar de um estágio para outro.

Rearranjando a expressão de modo a isolar para descobrir o tempo gasto em cada ciclo de instrução unitário (buscar, decodificar, executar etc.), tem-se:

$\tau_i = \tau - d$

Exemplo de aplicação

Qual é o tempo estimado de resposta de um único ciclo, dado que o computador em operação possua frequência de *clock* τ = 233 MHz e um tempo estimado para transição entre estágios d = 3,3 ns?

Resolução

Colocando os valores em notação científica: 233×10^6 Hz e $3,3 \times 10^{-9}$ segundos, e substituindo na equação modificada, lembrando que:

$$\tau = \frac{1}{f} = \frac{1}{233x10^6} = 4,291 \times 10^{-9} \text{ segundos}$$

Tem-se:

$\tau_i = 4,291 \times 10^{-9} - 3,3 \times 10^{-9} = 1,191 \times 10^{-9}$ segundos ou 1,191 nanosegundos ou ns

Suponha em um outro exemplo que *n* instruções serão processadas sem algum desvio. Considere $T_{k,n}$ o tempo total necessário para que um *pipeline* que possua *k* estágios processe *n* instruções, como:

$$T_{k,n} = \left[k + (n-1)\right]\tau$$

Um total de *k* ciclos será necessário para que se complete a execução da primeira instrução, de modo que o restante de *n* – 1 instruções irá requerer *n* – 1 ciclos? Essa equação pode ser obtida através do entendimento da figura 96, em que se nota que a instrução nove completará o ciclo no tempo quatorze, resultando em: 14 = [(6 + (9 – 1)].

Agora, considere que um processador possua funções equivalentes ao exemplo anterior, porém sem o uso do *pipeline*. Suponha também que o tempo de ciclo da instrução seja $k\tau$. Então, é necessário calcular qual será o fator de aceleração, também conhecido como *speedup*, para as instruções do *pipeline* comparado com a execução sem *pipeline*, o que será definido como:

$$Speedup = \frac{nk\tau}{\left[k + (n-1)\right]\tau}$$

Nota-se na equação anterior que o valor ocorre tanto no numerador quanto no denominador e pode ser cortado, resultando em:

$$Speedup = \frac{nk}{\left[k + (n-1)\right]}$$

Exemplo de aplicação

Qual será o *speedup* para um computador que possua um *pipeline* de k = 8 estágios e é solicitado que ele execute n = 515 instruções? Qual será o tempo total (em milissegundos) para que todas as instruções sejam executadas? Considere também que um outro computador execute as mesmas n = 515 instruções, entretanto utilizando um *pipeline* de k = 3 estágios. Ao final, faça um comparativo do desempenho dos dois computadores, levando em consideração a influência que o número de estágios produz no resultado.

Resolução

Para o primeiro computador, n = 515 e k = 8.

$$\text{Speedup} = \frac{515 \,.8}{\left[8+\left(515-1\right)\right]} = 7{,}89 \text{milissegundos ou ms}$$

Para o segundo computador, n = 515 e k = 3.

$$\text{Speedup} = \frac{515 \,.3}{\left[3+\left(515-1\right)\right]} = 2{,}98 \text{ms}$$

De acordo com os resultados obtidos, percebe-se que houve um ganho de 2,64 na aceleração do processamento das instruções para o computador com apenas k = 3 estágios em comparação com o computador que possuía k = 8 estágios. Entretanto, esse ganho pode não ser real, pois não há como garantir que o desempenho em cada estágio será o mesmo ou melhor do que o outro estágio unitário, podendo haver atrasos, incluindo perda de tempo de processamento.

Como já abordado, um estágio utilizado para buscar uma instrução não terá o mesmo tempo de duração se comparado com o estágio de execução. Além disso, diminuir o número de estágios implica que os estágios reduzidos terão muito mais funcionalidades, aumentando seu tempo operacional.

Outro grau comparativo que também poderia ser explorado é o uso da mesma quantidade de estágios em um computador, somente variando a quantidade de instruções envolvidas. Nessa situação, como o computador utilizaria o mesmo número de estágios, obviamente que operar menos instruções geraria um ganho real. Assim, em algumas situações o ganho pode até diminuir devido a fatores como:

- Aumento de custo computacional na implementação.
- Atrasos possíveis entre estágios do *pipeline*.
- Atrasos ocorridos no processo de esvaziamento do *pipeline*.

A figura a seguir mostra o fator de aceleração (*speedup*) como sendo uma função do número de instruções que são executadas sem desvio. Como observado, no limite $n \rightarrow \infty$ (lê-se "*n* tendendo ao infinito"), tem-se uma aceleração pelo fator *k*.

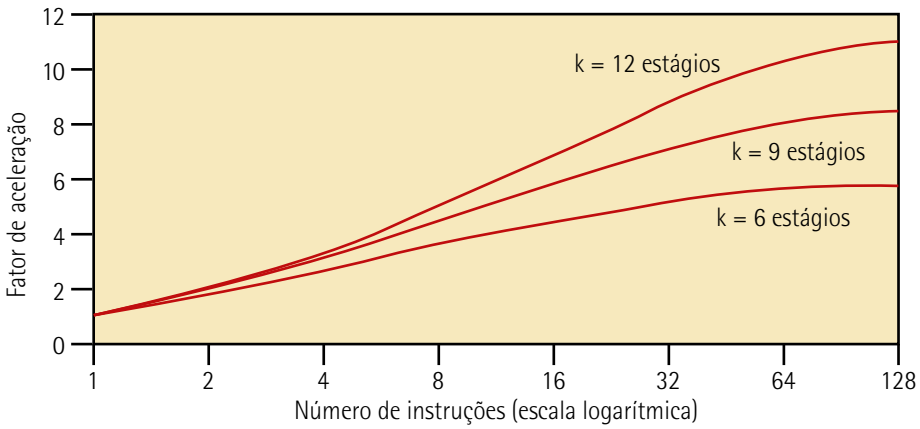


Figura 99 – Fator de aceleração em função do número de instruções em escala logarítmica

Já a figura a seguir mostra o fator de aceleração como uma função do número de estágios no *pipeline* da instrução. Nessa situação, o fator de aceleração se aproxima do número de instruções que podem ser utilizadas no *pipeline* sem desvios. Dessa forma, hipoteticamente, quanto maior o número de estágios de *pipeline*, maior será a aceleração. Mas em termos práticos, os ganhos reais dos estágios adicionais do *pipeline* podem ser confrontados pelo aumento do custo de tempo e computacional, além de possíveis atrasos entre estágios e pelo fato de que os desvios geralmente requerem o esvaziamento do *pipeline*, que deverá ser recarregado com novas instruções.

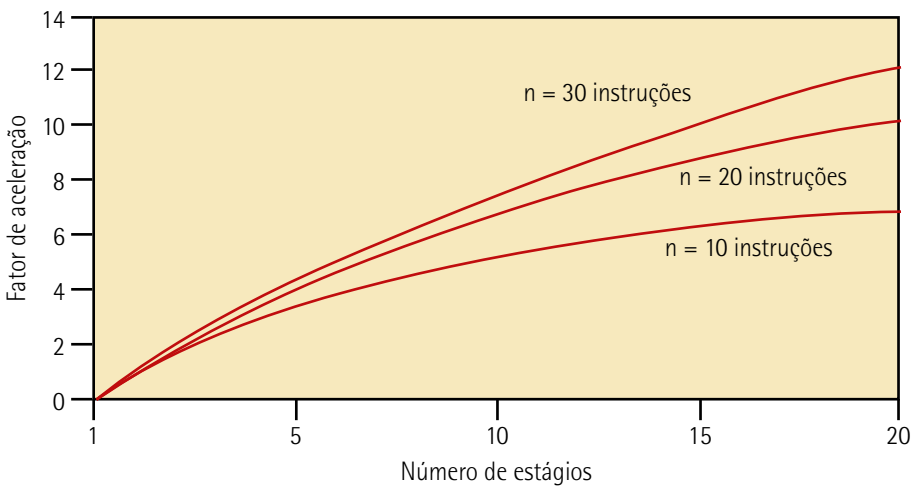


Figura 100 – Fator de aceleração em função do número de instruções

Ao se considerar somente o número de ciclos necessários para executar um dado conjunto de instruções, pode parecer que o *pipeline* não melhora o desempenho de execução de um processador. É certo que ao se projetar um *pipeline*, geralmente aumenta-se o número de ciclos de *clock* necessários para executar um dado programa, pois algumas funções ficam aprisionadas no *pipeline* aguardando

que as instruções que geram suas entradas sejam executadas. Dessa forma, a duração do ciclo em um processador com *pipeline* ficará dependente de quatro fatores:

- Duração do ciclo de *clock*.
- Número de estágios do *pipeline*.
- Homogeneidade na divisão de tarefas entre os estágios do *pipeline*.
- Latência dos *latches* (registradores encarregados de armazenar dados temporários).

Exemplo de aplicação

Um processador fictício que não tenha *pipeline* possui uma duração de ciclo de *clock* de 25 ns (nanossegundos ou x 10^{-9} segundos). Qual será a duração do ciclo de uma versão desse processador com *pipeline* de 5 estágios divididos homogeneamente, se cada *latch* tem uma latência de 1 ns? Suponha em uma outra situação que o processador foi dividido em 50 estágios de *pipeline* com o mesmo 1 ns de latência de *latch*.

Resolução

Como um sistema sem *pipeline* possui tempo de ciclo de 25 ns, se considerar os 5 estágios e a latência de 1 ns, tem-se (CARTER, 2002):

$$\text{Duração do ciclo com pipeline} = \frac{25\,\text{ns}}{5\,\text{estágios}} + 1\text{ns} = 6\text{ns}$$

Para a outra situação, em que a duração do ciclo sem pipeline também é 25 ns porém aumentando-se o número de estágios para 50, tem-se:

$$\text{Duração do ciclo com pipeline} = \frac{25\,\text{ns}}{50\,\text{estágios}} + 1\text{ns} = 1{,}5\ \text{ns}$$

No *pipeline* de 5 estágios, a latência do *latch* é apenas 1/6 da duração global do ciclo, enquanto a latência do *latch* é 2/3 da duração total do ciclo no *pipeline* de 50 estágios. Uma outra maneira de ver o resultado é que o *pipeline* de 50 estágios tem uma duração de ciclo que é 1/4 daquela do *pipeline* de 5 estágios, a um custo de 10 vezes mais *latches*.

Enquanto o *pipeline* pode reduzir a duração do ciclo de um processador, aumentando assim a taxa de rendimento das instruções, ele aumenta a latência do processador em, pelo menos, a soma de todas as latências dos *latches*. A latência de um *pipeline* é a soma do tempo que uma única instrução demora para passar através do *pipeline*, o que é o produto do número de estágios pela duração de ciclo de *clock*.

4.9.2 *Hazards* de *pipeline*

Um *hazard* de *pipeline* é definido como um evento em que a próxima instrução de *pipeline* não poderá ser executada no ciclo de *clock* seguinte, ocasionando um efeito bolha no *pipeline*. Isso ocorre no *pipeline* devido a alguma parte dele precisar parar de executar por causas que não permitem sua execução contínua, por exemplo um desvio. Os *hazards* de *pipeline* são subdivididos em três categorias: *hazards* de recursos, *hazards* de dados e *hazards* de controle (STALLINGS, 2010).

Hazards de recursos

Também conhecido como *hazard* estrutural, as bolhas de recursos ocorrem quando duas ou mais instruções que já estão no *pipeline* necessitam do mesmo recurso, resultando que as instruções precisarão ser executadas em série em vez de paralelo, o que, como já abordado, melhora seu desempenho. Por exemplo, suponha um *pipeline* simples contendo cinco estágios, em que cada estágio ocupa um ciclo de *clock*, como mostra a figura a seguir. Em uma situação idealizada, uma nova instrução entra no *pipeline* a cada ciclo de *clock*.

Ciclo de *clock*

Instrução	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			FI	DI	FO	EI	WO		
I4				FI	DI	FO	EI	WO	

Figura 101 – *Hazard* de recursos de cinco estágios

Entretanto, suponha agora que a memória principal tenha uma única porta de entrada/saída, ou seja, responsável por realizar as leituras e escritas de instruções e que deva ser executada uma a uma. Nessa situação, uma leitura ou escrita do operando na memória principal não poderá ser executada em paralelo, como exemplificado na figura a seguir. Nessa situação, assume-se que o operando de origem para a instrução I1 no eixo y da figura a seguir está na memória principal em vez de estar alocado em algum registrador. Assim, o estágio responsável pela busca de uma instrução deverá ficar ocioso por um ciclo de instrução antes de começar a busca da instrução para a instrução I3.

Ciclo de *clock*

Instrução	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			Ocioso	FI	DI	FO	EI	WO	
I4					FI	DI	FO	EI	WO

Figura 102 – *Hazard* de recursos de cinco estágios com operando de origem na memória

Outra situação de conflito de recurso ocorre quando várias instruções estão prontas para entrar na fase de execução da instrução, entretanto só há uma ULA disponível. Uma solução para essa bolha de recursos é aumentar os recursos disponíveis, por exemplo ter múltiplas portas para acesso à memória principal ou para a ULA.

Hazards de dados

Uma bolha de dados irá ocorrer quando houver um conflito no acesso de uma posição na memória de algum operando. Também se pode definir um *hazard* de dados da seguinte maneira: duas instruções de um programa estão a ponto de serem executadas na sequência, de modo que ambas as instruções acessam um determinado operando localizado ou na memória ou em algum registrador, ao mesmo tempo. Porém, se as instruções são executadas em um *pipeline*, então possivelmente o valor do operando será atualizado de forma que produza um resultado diferente do que o esperado do obtido por uma instrução em série. Isso tudo pode ser traduzido como um programa que gera um resultado incorreto para o usuário, devido ao uso do *pipeline*. Pode-se tomar como exemplo uma sequência de instruções em uma máquina com arquitetura x86. A primeira instrução no exemplo irá somar o conteúdo dos registradores EAX e EBX e armazenar o resultado em EAX. A segunda instrução irá subtrair o conteúdo de EAX de ECX e armazenar o resultado em ECX. A figura a seguir mostra como será o comportamento desse *pipeline*.

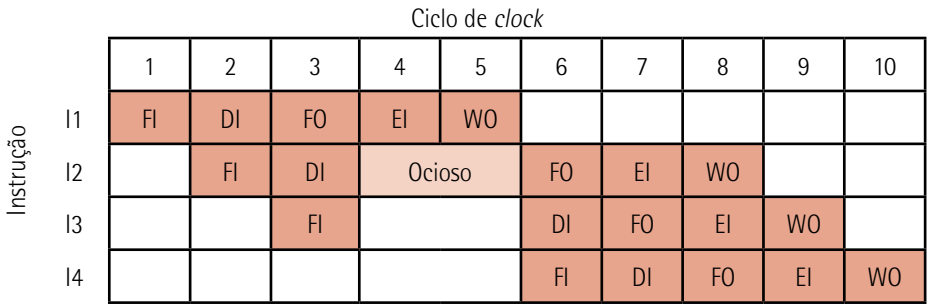


Figura 103 – *Hazard* de dados

Como mostra a figura, a instrução de soma ADD não atualiza EAX até o fim do estágio cinco, que ocorre no ciclo cinco de *clock* (eixo X). Porém, a instrução de subtração (SUB) necessita desse valor no começo de seu estágio dois, que ocorre no ciclo quatro de *clock*. De maneira que para manter a operação de forma correta, o *pipeline* deverá atrasar dois ciclos de *clock*. Então, na falta de *hardware* disponível e de algoritmos específicos que evitem esse tipo de problema, o *hazard* de dados poderá resultar em um uso ineficiente do *pipeline*. O *hazard* de dados pode ainda ser subdividido em três:

- **Leitura após escrita ou dependência verdadeira**: caso em que uma instrução modificará o conteúdo de um registrador ou de uma posição da memória principal de modo que uma instrução subsequente irá ler os dados dessa posição. Nessa situação, a operação de leitura ocorre antes de a operação de escrita ter sido completada.

- **Escrita após leitura ou antidependência**: caso em que uma instrução irá ler o conteúdo de um registrador ou posição da memória e uma instrução subsequente será escrita nessa posição. Nessa situação, o *hazard* ocorre se a operação de escrita for completada antes da operação de leitura.

- **Escrita após escrita ou dependência de saída**: caso em que duas instruções escrevem dados ou instruções na mesma posição (memória ou registrador). O problema nesse caso ocorre quando as operações de escrita são executadas na sequência inversa da qual era esperada.

Hazards de controle

Uma bolha de controle, conhecida também como *hazard* de desvio, ocorre quando o *pipeline* toma uma decisão errada ao prever algum tipo de desvio e, dessa forma, traz instruções para dentro do *pipeline* que precisarão ser descartadas na sequência.

4.9.3 Previsão de desvio em *pipeline*

Um dos maiores problemas para se implementar um projeto ótimo de *pipeline* é garantir um fluxo estável e contínuo de instruções para os estágios iniciais do *pipeline*, o que é geralmente afetado por desvios condicionais. Assim, até que a instrução seja executada, é quase impossível prever se o desvio para uma outra instrução será tomado ou não. Essa indecisão sobre a tomada de decisão para desvios, que podem afetar o desempenho do *pipeline*, pode ser implementada através de soluções com as seguintes abordagens: múltiplos fluxos, busca antecipada do alvo de desvio, *buffer* de laço de repetição, previsão de desvio e desvio atrasado.

Múltiplos fluxos

Mesmo um *pipeline* simplificado possui penalidades (atrasos) na execução de uma instrução, pois durante a busca por duas ou mais instruções, algumas delas podem estar incorretas. Nessa situação, uma possível abordagem envolve um esforço do tipo força bruta, que replica as partes iniciais do *pipeline* e permite que as instruções corretas sejam obtidas, mantendo assim o uso de dois fluxos. No entanto, ainda existem dois problemas com essa abordagem:

- Devido aos múltiplos *pipelines*, ainda existirão atrasos no acesso à memória principal ou aos registradores.

- Algumas instruções de desvio adicionais também podem entrar no *pipeline* antes mesmo da solução da tomada de decisão referente ao desvio original, o que ocasionaria fluxos adicionais e desnecessários para tomadas de decisão.

Apesar das desvantagens apresentadas, a estratégia de múltiplos fluxos ainda pode melhorar o desempenho do *pipeline*, como os obtidos nas máquinas IBM 370/168 e IBM 3033.

Busca antecipada do alvo de desvio

Após um desvio condicional ser reconhecido, o alvo do desvio condicional é lido antecipadamente. Assim, o alvo é salvo até que a instrução de desvio seja executada, de maneira que se o desvio for tomado, o alvo já terá sido obtido.

Buffer de laço de repetição

Um *buffer* de laço de repetição nada mais é do que uma pequena e rápida memória mantida no estágio de busca de instrução e que contém *n* instruções recentemente lidas na sequência. Dessa forma, se um desvio está a ponto de ser tomado, o *hardware* irá verificar se o alvo do desvio está no *buffer*. Assim, se o alvo estiver, a próxima instrução será obtida do *buffer*. Nessa abordagem, o *buffer* de laço de repetição pode trazer alguns benefícios, como:

- Através da busca antecipada, o *buffer* de laço irá conter algumas instruções na sequência, antes do endereço da instrução atual. Dessa forma, as instruções obtidas na sequência ficarão disponíveis, sem a necessidade do uso do tempo para acesso à memória principal.

- Se o desvio para o alvo estiver contido em apenas algumas posições na frente do endereço da instrução de desvio, então o alvo já estará no *buffer*, o que é muito útil em ocorrências de tomadas de decisão IF-THEN-ELSE.

- Se o *buffer* de laço de repetição for muito grande para conter todas as instruções do laço, então essas instruções precisarão estar disponíveis na memória apenas uma vez, logo na primeira iteração. Nas demais iterações, todas as instruções necessárias já estarão no *buffer*.

O *buffer* de laço de repetição pode ser entendido como uma memória *cache* dedicada para instruções. A única diferença é que o *buffer* de laço pode guardar apenas instruções na sequência; além disso, ele também possui um tamanho e custo menor se comparado com a *cache*.

Previsão de desvio

As abordagens para previsão de desvio podem se basear em várias técnicas, como: previsão nunca tomada, previsão sempre tomada, previsão por código de operação (*opcode*), chave tomada/não tomada e tabela de histórico de desvio. As três primeiras abordagens são estáticas e não dependem do histórico de execução até ocorrer a instrução de desvio condicional. Já as duas últimas abordagens (chave tomada/não tomada e tabela de histórico de desvio) são dinâmicas e dependerão do histórico da execução. Nas situações mais simples como a previsão nunca tomada e previsão sempre tomada, assume-se que o desvio nunca será tomado e se continuará obtendo as instruções sequencialmente, ou também se pode assumir que o desvio será tomado, obtendo assim o alvo do desvio para ser tratado.

Observação

Alguns estudos que lidam com o tratamento de desvio e seu comportamento indicam que os desvios condicionais nos programas são tomados em cerca 50% das situações, de modo que, se o custo de uma busca antecipada dos dois caminhos for o mesmo, realizar a busca antecipada no mesmo endereço do alvo do desvio pode oferecer uma melhora no desempenho.

Ao realizar uma busca utilizando uma memória virtual como a paginação, poderá ocasionar uma falha na página de busca, o que traz como consequência alguma penalidade no desempenho (STALLINGS, 2010). A abordagem estática, como já discutido, toma uma decisão baseada no código de operação da instrução de desvio, de modo que o processador assumirá que o desvio realizado será executado somente para alguns códigos de operações e não para outros, reportando uma taxa de sucesso superior a 75%. Já as abordagens dinâmicas tentam melhorar não só o desempenho como a precisão da previsão, principalmente armazenando um histórico de instruções de desvios condicionais contidas em um programa. Por exemplo, 1 bit ou mais podem ser associados com cada instrução de desvio condicional referente ao histórico recente da instrução. Esses *bits*, conhecidos como uma chave tomada/não tomada, direcionam o processador a tomar uma decisão na próxima vez que a instrução desejada for encontrada.

No geral, esses *bits* contidos no histórico não são associados com a instrução na memória principal, mas sim guardados em um armazenamento temporário. Outra possibilidade seria associar esses *bits* com qualquer instrução de desvio condicional que esteja associada em uma memória *cache*. Assim, quando a instrução for substituída na *cache*, o seu histórico será perdido. Entre outras possibilidades, é possível manter uma tabela com instruções de desvio recentemente executadas. Dessa forma, o processador pode acessar a tabela usando os *bits* de ordem mais baixa do endereço da instrução de desvio condicional.

Com apenas 1 bit, o que pode ser guardado se refere à última execução dessa instrução que resultou em um desvio condicional ou não. Logicamente que há desvantagens nessa abordagem, uma delas é que usar 1 bit único pode levar a instrução de desvio a sempre tomar como uma instrução um *loop* repetitivo. Além disso, com apenas 1 bit no histórico, um erro de previsão de desvio ocorrerá duas vezes para cada uso do *loop*: uma vez ao entrar no *loop* e outra vez ao sair do *loop*. Se 2 bits forem utilizados, então sua função será guardar o resultado das duas últimas instâncias da execução da instrução associada ao processo no momento. A figura a seguir mostra um fluxograma representando o comportamento de uma previsão de desvio. Iniciando o fluxograma a partir da leitura da próxima instrução do desvio condicional, à medida que cada instrução de desvio condicional na sequência for tomada, o processo de decisão poderá prever se o próximo desvio será tomado.

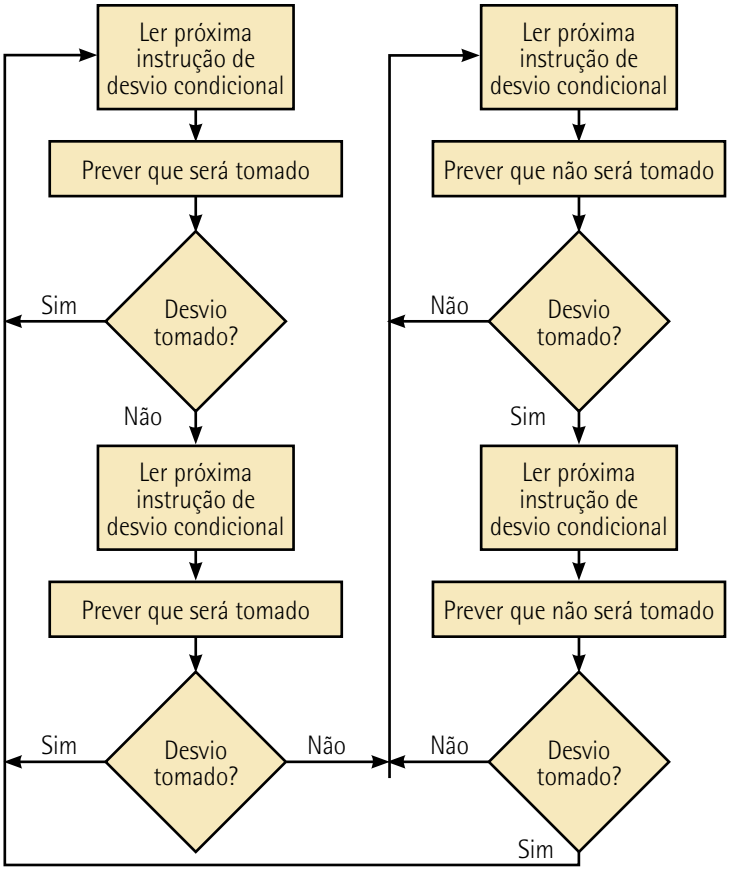


Figura 104 – Fluxograma contendo tomadas de decisão para previsões de desvio

Se durante o processo uma única previsão estiver errada, então o algoritmo continuará prevendo que o próximo desvio também será tomado. Apenas se dois ou mais desvios seguidos não forem atendidos fará com que o algoritmo seja direcionado para o lado direito do fluxograma. Somente nos casos em que o algoritmo contiver duas previsões erradas é que ele irá mudar sua decisão de previsão. Uma forma mais compacta para a previsão de desvios pode ser obtida através da representação de máquina de estados finitos, como mostra a figura a seguir.

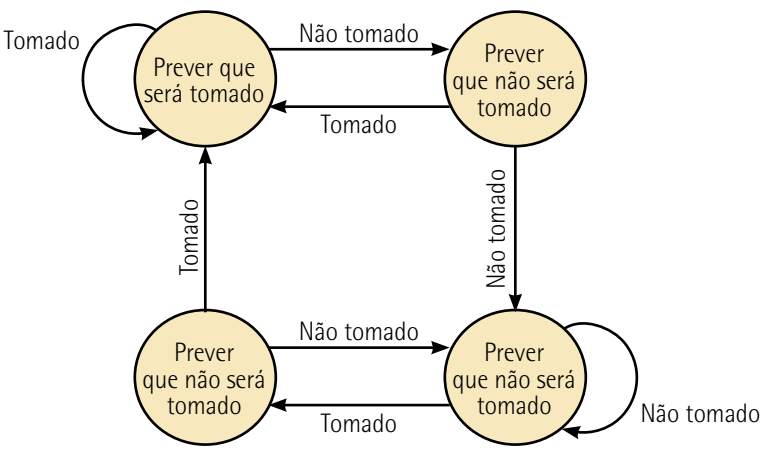


Figura 105 – Previsão de desvio baseado em máquinas de estados finitos

Entretanto, o uso de *bits* de histórico possui uma desvantagem: se houver uma decisão de aceitar e tomar o desvio, a instrução alvo não poderá ser obtida até que o endereço do alvo seja decodificado. Obtém-se uma eficiência maior no processo se a leitura da instrução for iniciada assim que a decisão de desvio tomada for realizada. Para isso, mais informações do processo precisam ser salvas no *buffer* de alvo do desvio ou em uma tabela de histórico de desvio.

Saiba mais

Conheça um pouco mais sobre processamento paralelo:

PROCESSAMENTO paralelo chega aos computadores de mesa. *Inovação Tecnológica*, 27 jun. 2017. Disponível em: https://bit.ly/3qDoC23. Acesso em: 7 fev. 2021.

4.10 Linguagem de montagem

Uma linguagem de montagem pode ser definida como uma codificação em que cada declaração produzirá exatamente uma instrução de máquina, ou seja, existe uma correspondência um-para-um entre as instruções de máquina e as declarações obtidas no programa de montagem. Assim, se cada linha no código em linguagem de montagem contiver uma declaração e cada palavra de máquina contiver uma instrução, então um código em linguagem de montagem de *n* linhas irá produzir um código em linguagem de máquina de *n* instruções (TANENBAUM; AUSTIN, 2013).

As linguagens de montagem estão presentes em todos os computadores modernos. Esse tipo de linguagem possui características diferentes se comparado aos demais níveis da arquitetura de um computador como o nível *instruction set architecture* (ISA – arquitetura do conjunto de instrução) ou mesmo o nível de sistema operacional, pois se implementa através da tradução em vez de uma interpretação. Os tradutores podem ser entendidos como um programa que converte um código em nível de usuário em um outro programa. A linguagem na qual o código original é escrito denomina-se linguagem-fonte e, por sua vez, a linguagem na qual esse código é convertido denomina-se linguagem-alvo. Outra diferença entre as linguagens é que na tradução, o código original na linguagem-fonte não será executado diretamente, pois primeiro ele é convertido para um programa equivalente denominado programa-objeto ou programa binário executável. Essa tarefa de conversão é realizada somente após a conclusão da tradução, gerando, nesses casos, duas etapas distintas:

- Geração de um programa equivalente na linguagem-alvo.
- Execução do programa recém-gerado na linguagem-alvo.

Essas etapas não irão ocorrer simultaneamente, por motivos óbvios, pois a segunda etapa só poderá iniciar após a conclusão da primeira. Enquanto isso, na interpretação, só há uma etapa, que é a execução do programa-fonte original. Assim, não é preciso gerar com alguma antecedência qualquer código equivalente,

mesmo que em algumas situações exista a necessidade de o programa-fonte ser convertido para alguma forma intermediária de codificação, como ocorre na linguagem de programação Java e seus *bytecodes*.

De forma geral, os tradutores podem ser divididos em dois grupos:

- **Assembler (montador)**: nessa situação, a linguagem-fonte se baseia em uma representação simbólica para uma linguagem de máquina numérica.

- **Assembly**: representa a linguagem de montagem legível para o ser humano e é utilizada para programar códigos específicos identificados pelas máquinas.

Observação

O uso da linguagem de montagem se justifica devido à dificuldade em se programar em linguagem de máquina pura, que é baseada em números binários ou hexadecimais, dificultando seu manuseio pelo usuário final.

É muito mais fácil para o usuário trabalhar com nomes ou abreviaturas simbólicas como ADD, SUB, MUL e DIV para operações de soma, subtração, multiplicação e divisão, do que utilizar valores numéricos em binários como os utilizados pela máquina. Os tipos mais comuns de operações em linguagem de máquina são: transferência de dados, aritmética, lógica, conversão, E/S, controle do sistema e transferência de controle. Os quadros a seguir mostram os tipos mais comuns de operações utilizadas em linguagens de máquina.

Quadro 5 – Operações lógicas e aritméticas do conjunto de instruções

Tipo	Nome da operação	Descrição
Transferência de dados	*Move* (transferência)	Transfere palavra ou bloco da origem ao destino
	Store (armazenamento)	Transfere palavra do processador para a memória
	Load (busca)	Transfere palavra da memória para o processador
	Exchange (troca)	Troca o conteúdo da origem e do destino
	Clear (*reset*)	Transfere palavra de 0s para o destino
	Set	Transfere palavra de 1s para o destino
	Push	Transfere palavra da origem para o topo da pilha
	Pop	Transfere palavra do topo da pilha para o destino
Aritmética	Soma	Calcula a soma de dois operandos
	Subtração	Calcula a diferença de dois operandos
	Multiplicação	Calcula o produto de dois operandos
	Divisão	Calcula o quociente de dois operandos
	Absoluto	Substitui o operando pelo seu valor absoluto
	Negativo	Troca o sinal do operando
	Incremento	Soma 1 ao operando
	Decremento	Subtrai 1 do operando

Tipo	Nome da operação	Descrição
Lógica	AND	Realiza o AND lógico
	OR	Realiza o OR lógico
	NOT (complemento)	Realiza o NOT lógico
	Exclusive-OR	Realiza o XOR lógico
	Test	Testa condição especificada; define *flag*(s) com base no resultado
	Compare	Comparação lógica/aritmética de dois ou mais operandos; define *flag*(s) com base no resultado
	Definir variáveis de controle	Classe de instruções para definir controles para fins de proteção, controle etc.
	Shift	Desloca o operando para a esquerda (direita), introduzindo constantes na extremidade
	Rotate	Desloca ciclicamente o operando para a esquerda (direita), de uma à direita

Adaptado de: Stallings (2010, p. 18).

Quadro 6 – Operações de controle e transferência de dados do conjunto de instruções

Tipo	Nome da operação	Descrição
Transferência de controle	*Jump* (desvio)	Transferência incondicional; carrega PC com endereço especificado
	Jump condicional	Testa condição especificada; ou carrega PC com endereço especificado ou carrega PC com endereço especificado
	Jump para sub-rotina	Coloca informação do controle do programa atual em local conhecido
	Return	Substitui conteúdo do PC por outro registrador de local conhecido
	Execute	Busca operando do local especificado e executa como instrução
	Skip	Incrementa o PC para saltar para a próxima instrução
	Skip condicional	Testa condição especificada; ou salta ou não faz nada, com base na condição
	Halt	Termina a execução do programa
	Wait (*hold*)	Termina a execução do programa; testa condição especificada repetidamente
	No operation	Nenhuma operação é realizada, mas a execução do programa continua
Entrada/Saída	*Input* (leitura)	Transfere dados da porta de E/S ou dispositivo especificado para o destino
	Output (escrita)	Transfere dados da origem especificada para porta de E/S ou dispositivo
	Start I/O	Transfere instruções para o processador de E/S para iniciar operação de E/S
	Test I/O	Transfere informações de estado do sistema de E/S para destino especificado
Conversão	*Translate*	Traduz valores em uma seção da memória com base em uma tabela de correspondências
	Convert	Converte o conteúdo de uma palavra de uma forma para outra

Adaptado de: Stallings (2010, p. 18).

Lembrete

Um desenvolvedor em linguagem de montagem pode atribuir nomes simbólicos a endereços de memória e deixar que o Assembler se preocupe em fornecer os valores numéricos corretos.

As linguagens de montagem ainda possuem outras propriedades além do mapeamento um-para-um para as declarações. O desenvolvedor em linguagem de montagem possuirá acesso a todos os recursos e instruções disponíveis na máquina-alvo, fato que não ocorre para desenvolvedores em linguagens de alto nível. As linguagens de montagem também possuem acesso completo ao *hardware*, além de facilitar a configuração em várias aplicações de *hardware* embutido ou embarcado (*embedded systems*). De forma resumida, tudo o que pode ser realizado em linguagem de máquina também pode ser realizado em linguagem de montagem, entretanto, muitas instruções, registradores ou outros recursos não estarão disponíveis para utilização por um programador de linguagem de alto nível.

Logicamente, nem tudo é vantagem para as linguagens de montagem. Linguagens de alto nível podem, a princípio, ser executadas em todos os computadores atuais, porém linguagens de montagem só podem ser executadas em uma família específica de máquinas, como a família Intel x86 ou a família *advanced risc machine* (ARM).

Em termos de codificação, a estrutura de uma declaração em linguagem de montagem é parecida com a estrutura de linguagem de máquina, de forma que praticamente todas as operações aritméticas e lógicas podem ser representadas em linguagem de montagem.

O quadro a seguir mostra, de forma simplificada, alguns fragmentos de programas de linguagem de montagem para uma arquitetura x86 para efetuar o cálculo N = I + J. Nesse exemplo, as declarações são comandos necessários para o Assembler reservar memória para as variáveis I, J e N. Nessa situação, são necessários no máximo dois endereços de memória para referenciar os operandos de origem. Os resultados dessas operações precisam ser armazenados, o que sugere a necessidade de um terceiro endereço para definir o operando de destino. Após o término de uma instrução, a próxima instrução precisará ser buscada e seu endereço também será necessário para realizar a operação.

Quadro 7 – Cálculo de N = I + J na arquitetura Intel x86

Etiqueta	*Opcodes*	Operandos	Comentários
Fórmula	MOV	EAX,I	;resgistrador EAX=I
	ADD	EAX,J	;resgistrador 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

Adaptado de: Tanenbaum e Austin (2013, p. 416).

Através dessa linha de raciocínio, é possível observar que uma instrução pode conter quatro referências de endereços, sendo dois operandos de origem, um operando de destino e o endereço da próxima instrução. Na figura a seguir, é possível observar uma comparação entre instruções típicas de um, dois e três endereços de operandos, que são utilizados para realizar o cálculo: $Y = \frac{(A - b)}{c + (D.E)}$.

Instrução		Comentário
SUB	Y, A, B	Y ← A – B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

Figura 106 – Programa para executar a operação aritmética utilizando três operandos de endereçamento

Nesse primeiro exemplo, é possível observar o início do programa realizando uma operação aritmética SUB Y, A, B, em que os conteúdos de A e B são subtraídos e armazenados temporariamente no operando Y. Na linha seguinte, é realizada a operação de multiplicação de D, E, armazenando o resultado da operação em T. A penúltima operação envolve a adição do resultado contido na multiplicação e armazenada em T, somado ao operando C e armazenando o resultado da operação também em T. Para finalizar a operação, o resultado da subtração (A – B) armazenado em Y é dividido pelo resultado da operação (C + (D · E)) armazenado em T, e colocando tudo como resposta em Y. A mesma equação pode ser resolvida utilizando dois endereços de operandos, como mostra a figura a seguir.

Instrução		Comentário
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y – B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

Figura 107 – Programa para executar a operação aritmética utilizando dois operandos de endereçamento

No segundo exemplo, inicialmente é realizada uma cópia do conteúdo do operando A, movendo-o para o endereço do operando Y. Depois, realiza-se a subtração de Y, B que nada mais é do que a subtração de (A – B). Na terceira linha, move-se o conteúdo do operando D para o endereço T. Em seguida, multiplica-se T, E. Na penúltima linha, aciona-se o resultado obtido na multiplicação anterior armazenado em T por C, como na equação original. Por fim, dividem-se os valores armazenados em Y, resultantes de (A – B) pelos valores contidos em T (C + (D · E)). Para operações que envolvem apenas um endereço de operando, as instruções são ainda mais simples. Para que esse tipo de instruções funcione, um segundo endereço precisa ser implícito, como ocorria em computadores mais antigos. Um endereço implícito geralmente corresponde a um registrador acumulador (AC). O acumulador irá conter um dos operandos que será utilizado para armazenar o resultado, como mostra o exemplo da figura a seguir.

Instrução		Comentário
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC – B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

Figura 108 – Programa para executar a operação aritmética utilizando um operando de endereçamento

Nesse último exemplo, inicialmente o operando D é carregado para o acumulador através da operação LOAD. Após ter carregado D, a operação seguinte é a multiplicação do operando E pelo conteúdo do acumulador (D). A operação prossegue com a soma do conteúdo do acumulador pelo operando C. Com a parte do denominador da equação já resolvida (C + (D · E)), o resultado é transferido para o operando Y através da operação LOAD, pois o acumulador terá seus valores alterados com as próximas operações. A operação do numerador se inicia com o carregamento do operando A para o acumulador, pela operação LOAD A. Depois, o conteúdo do acumulador (A) é subtraído de B, terminando as operações no numerador. Por fim, necessita-se dividir o numerador pelo denominador através da operação DIV Y, que representa a divisão do conteúdo armazenado em Y pelo conteúdo armazenado no acumulador, tudo isso sendo armazenado novamente em Y (STOR Y).

Saiba mais

Para conhecer um pouco mais sobre linguagens de baixo nível, leia:

ANIDO, R. *Linguagens de montagem*. 9. ed. Rio de Janeiro: Elsevier, 2016. Capítulo 3.

Resumo

Nesta unidade, estudamos o quão importantes são as instruções de máquina para o pleno funcionamento de um computador. Vimos que as instruções são, em geral, tabeladas e utilizadas de acordo com sua funcionalidade, utilizando basicamente caracteres em base hexadecimal.

Além disso, também foi possível observar que os computadores podem ter um desempenho melhor se, em seu ciclo de funcionamento, houver interrupções simples ou múltiplas.

Em se tratando da melhora no desempenho do processamento dos computadores, vimos que o processo denominado *pipeline* pode otimizar o tempo total de processamento, executando as instruções em paralelo em vez de sequencialmente.

Exercícios

Questão 1. Observe a figura a seguir, que representa os modelos Harvard e von Neumann de arquitetura de computadores.

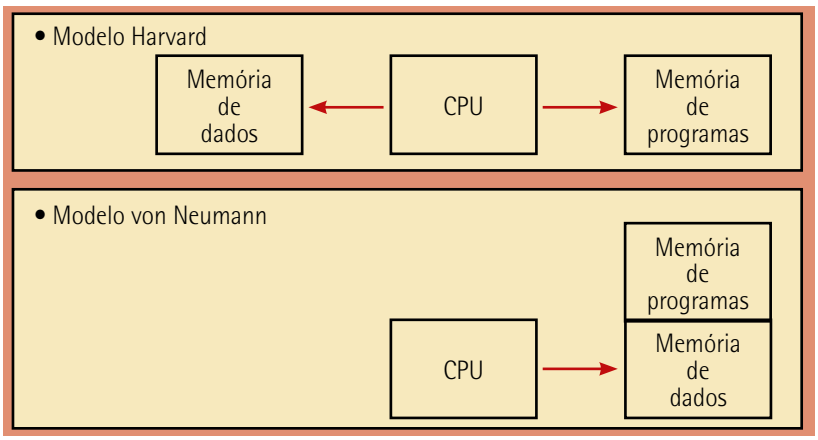


Figura 109 – Modelos Harvard e Von Neumann de arquitetura de computadores

PEREIRA, J. *Arquitetura de computadores*. [s.d.]. Disponível em: https://docente.ifrn.edu.br/jonathanpereira/disciplinas/organizacao-e-manutencao-de-computadores/arquiteturas-harvard-e-von-neumann. Acesso em: 17 nov. 2020.

Com base na figura e nos seus conhecimentos, avalie as afirmativas a seguir sobre as arquiteturas Harvard e Von Neumann.

I – Na arquitetura Von Neumann, o barramento de endereços e o barramento de dados costumam estar separados, assim como a memória de dados e a memória de instruções.

II – Na arquitetura Harvard, a memória de dados é a mesma utilizada para as instruções, assim como os respectivos barramentos.

III – Uma vantagem da arquitetura Harvard é que existe a possibilidade de leitura de instruções e de dados simultaneamente, uma vez que as memórias e os barramentos de dados e de instruções estão separados.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) III, apenas.

D) I e II, apenas.

E) I e III, apenas.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: a memória de dados e a memória de instruções não estão separadas na arquitetura von Neumann.

II – Afirmativa incorreta.

Justificativa: na arquitetura Harvard, a memória de dados está separada da memória de instruções.

III – Afirmativa correta.

Justificativa: a possibilidade de paralelismo vem do fato de haver memórias e barramentos independentes para dados e instruções, o que pode ser explorado pelo microprocessador para aumentar a *performance* do sistema.

Questão 2. Suponha que se queira avaliar a arquitetura computacional de uma máquina A de um ponto de vista quantitativo. Considere que o tempo de execução de um programa com 1.000 instruções foi de 10 μs. Suponha que todas as instruções desse programa levem a mesma quantidade de ciclos para a sua execução, que não exista nenhuma forma de paralelismo e que a CPU do computador trabalhe com um *clock* de 1 GHz.

Com base no cenário apresentado, avalie as afirmativas a seguir.

I – A quantidade de ciclos por instruções (*cycles per instruction* – CPI) no cenário do enunciado é igual a 10.

II – Segundo os dados do enunciado, podemos afirmar que o computador em análise não é capaz de executar mais de 50 milhões de instruções por segundo.

III – Se houver outra máquina B, trabalhando com o mesmo *clock* e com o mesmo conjunto de instruções e executando o mesmo programa do enunciado, mas com um valor de ciclos por instruções (*cycles per instruction* – CPI) igual à metade do valor relativo à máquina A, a quantidade de milhões de instruções por segundo (*million instructions per second* – MIPS) para a máquina B é sempre inferior à quantidade relativa à máquina A.

É correto o que se afirma apenas em:

A) I.

B) II.

C) III.

D) I e II.

E) I e III.

Resposta correta: alternativa A.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: para avaliarmos o que é dito na afirmativa, basta calcularmos o CPI, que é igual ao tempo T de execução multiplicado pela frequência f e dividido pelo número I_c de instruções do programa executado. Assim, ficamos com:

$$CPI = Txf/I_c = 10 \times 10^{-6} \times 10^9/10^3 = 10$$

Vale notar que o CPI é uma grandeza adimensional.

Observe que, normalmente, esse valor pode ser considerado uma média ou uma estimativa do número de ciclos que o microprocessador deve gastar para executar cada instrução. No caso do problema do enunciado, como todas as instruções levam o mesmo tempo, tal valor é o mesmo para todas as instruções. Note ainda que, em uma situação real, esse valor seria aplicado apenas às instruções que foram efetivamente executadas pelo programa.

II – Afirmativa incorreta.

Justificativa: vamos calcular a quantidade instruções por segundo, indicada por V_{MIPS}:

$$V_{MIPS} = f/(CPI \times 10^6) = 10^9/(10 \cdot 10^6) = 100 \text{ MIPS}$$

O valor obtido é maior do que 50 MIPS, indicado como máximo na afirmativa.

III – Afirmativa incorreta.

Justificativa: vamos analisar a fórmula do cálculo de MIPS, dada por:

$MIPS = f/(CPI \times 10^6)$

Podemos perceber que, se diminuirmos o valor de CPI para uma mesma frequência (f), o valor de MIPS deve aumentar, e não diminuir, como dito na afirmativa.