

Organização simplificada

Para todos os efeitos dessas notas introdutórias, podemos entender a organização interna de um computador como um processador central que tem acesso a uma memória e a canais de entrada e de saída de dados.

Um *bit* é a menor unidade de informação que podemos usar, podendo assumir um dos valores 0 ou 1. Usamos o termo *byte* para designar uma seqüência de 8 bits.

A memória nada mais é que um arranjo linear de um certo número de bytes, organizados seqüencialmente. Temos o byte que ocupa a posição 1, outro a posição 2, e assim por diante, até esgotarmos a capacidade física da memória. Normalmente, a memória é composta por 2^N bytes, para algum valor de N . A tabela abaixo mostra algumas possibilidades.

N	2^N	Bytes
10	1024	1K
20	1048576	1M
27	134217728	128M
30	1073741824	1G
32	4294967296	4G

A coluna 2^N indica o número de posições de memória disponíveis para alguns valores de N . A coluna Bytes indica abreviações bastante usadas. Assim, 1K indica 1024 posições, ou cerca de mil posições; 1M indica cerca de um milhão de posições; e 1G indica cerca de um bilhão de posições. Hoje em dia, é comum encontrar-se computadores com 512M posições de memória, ou mais.

O outro componente importante da arquitetura da máquina é o processador central. É o processador que interpreta e executa as instruções elementares que compõem um programa. Basicamente, o processador pode obter dados da memória, i.e. pode consultar uma certa seqüência de bytes lá armazenados; pode realizar operações simples com os dados obtidos, tais como soma, subtração, multiplicação e divisão; e pode remeter resultados para a memória, armazenando-os lá.

Para nosso consumo, não é importante detalhar como essas operações são realizadas em uma máquina física. De fato, será suficiente entender a memória do computador como um conjunto de “locais” onde se pode armazenar dados. Esses locais de armazenamento serão conhecidos e referenciados através de nomes simbólicos, tais como os nomes N , P e Z no algoritmo para o cálculo da potência 2^N , visto anteriormente. Exatamente em quais bytes da memória está armazenado o valor de N , por exemplo, não é importante. Também não é importante, em geral, saber quantos bytes são usados para armazenar os valores das variáveis. Isso é um grande alívio para o programador, pois o liberta de ter que lidar com posições fixas de memória, bem como o poupa da necessidade de conhecer em detalhes a arquitetura da máquina para qual está programando. Mais ainda, em computadores

modernos, é comum que os dados associados a uma mesma variável ocupem posições de memória distintas em duas execuções separadas do mesmo programa. Portanto, nos contentaremos em imaginar que os dados associados a uma variável X estão armazenados em algum ponto da memória, exatamente qual não sendo importante. Sempre que escrevermos o nome X no texto do programa, estaremos nos referindo aos dados lá armazenados. Esse mecanismo ficará mais claro quando lidarmos com detalhes de linguagens de programação específicas. Por outro lado, em alguns programas mais sofisticados pode, eventualmente, surgir a necessidade de um controle mais detalhado sobre as posições de memória ocupadas por variáveis. Nesses casos, a linguagem de programação oferece facilidades especiais para ajudar o programador nessa tarefa.

Finalmente, para completar a arquitetura simplificada da máquina, devemos mencionar os canais de entrada e de saída de dados. Normalmente, estes estão associados aos dispositivos periféricos do computador, tais como teclado, monitor, *mouse*, CD-ROM, entre muitos outros. Para nosso entendimento, podemos assumir que os dados passados através dos canais de entrada são transferidos diretamente para os locais de memória indicados na operação. De forma semelhante, podemos assumir que os dados de saída passam diretamente dos locais de memória indicados para os dispositivos de saída envolvidos na operação.

Linguagens e compilação

Como já mencionado, o processador central é capaz de realizar apenas operações muito primitivas sobre seqüências de bits – ou bytes. Operações tais como trocar um bit de 0 para 1, ou vice-versa; somar (na base 2) o conteúdo de dois bytes; deslocar para a direita ou para a esquerda os bits de um byte; e assim por diante.

Descrever um algoritmo sofisticado fazendo uso apenas dessas operações tão simples seria tremendamente custoso. Para ajudar nessa tarefa, surgiram então as linguagens de programação de alto nível. Estas são dialetos usados para se descrever receitas, ou algoritmos, fazendo uso de operações mais poderosas, e mais próximas do raciocínio abstrato que empregamos no dia-a-dia. E mais ainda, essas linguagens nos permitem fazer uso de dados estruturados de formas muito mais complexas do que as simples e limitadas seqüências de bytes de que poderíamos lançar mão quando programamos usando as instruções primitivas da máquina e sua memória sequencial.

Existem inúmeras linguagens de programação, cada uma voltada mais para um segmento específico de atividade. Entre elas:

- FORTRAN: uma das primeiras, usada em cálculos científicos;
- ALGOL, C, PASCAL: linguagens de uso geral, e que oferecem facilidades mais modernas do que o FORTRAN. No decorrer desse texto estudaremos a linguagem C em detalhes.
- C++, C#, JAVA: linguagens mais atuais ainda, que oferecem primitivas para programação orientada a objetos. Essas primitivas podem facilitar bastante a tarefa do programador, permitindo formas sofisticadas de se organizar o texto do programa. Também daremos detalhes da linguagem C#, mais adiante nesse texto.

- LISP, PROLOG: linguagens mais indicadas para se codificar tarefas específicas em inteligência artificial.
- . . . muitas outras

No entanto, o processador central não consegue interpretar diretamente as operações básicas dessas linguagens. Em consequência, algoritmos descritos nessas linguagens não podem ser diretamente executados no computador. É preciso traduzir esses textos para textos equivalentes que contenham apenas instruções básicas previstas na arquitetura do computador. Ocorre que, fixadas uma linguagem de programação e um computador, existe um algoritmo que opera essa tradução. Isto é, o algoritmo tem como entrada o texto na linguagem de alto nível e produz como saída o texto na linguagem mais primitiva. Esse algoritmo é conhecido como um *compilador*, e o processo de tradução é conhecido como *compilação*. Em geral, o texto em alto nível está armazenado em um arquivo em disco, e o resultado da compilação é também armazenado em outro arquivo também em disco. O primeiro recebe o nome de *programa fonte* – pois contém a descrição original, em alto nível; o segundo recebe o nome de *programa objeto* – será o objeto a ser executado pelo computador.

O processo todo pode ser visualizado da seguinte forma:

1. Temos um problema a resolver.
2. Alguém, provavelmente sem o auxílio de um computador, cria uma idéia que leva a descrição de um algoritmo adequado para resolver o problema.
3. O texto do algoritmo deve ser descrito numa linguagem de programação de alto nível. Essa tarefa não é automatizada, mas provavelmente será realizada com auxílio de um computador, usando editores de texto, por exemplo. O resultado será um arquivo armazenado em disco, contendo o programa fonte.
4. O programa fonte deve ser compilado, gerando um arquivo com o texto do programa objeto. Aqui, o uso do computador e de um compilador apropriado é essencial.
5. O programa objeto, junto com os dados de entrada apropriados, é executado para se obter o resultado esperado. Obviamente, aqui faz-se uso de um computador para executar o algoritmo criado no passo 2.

No passo 4, é possível que o processo de compilação indique a presença de erros. Significa que a programação realizada no passo 3 não foi adequada. Devemos recuar para o passo 3, corrigir os erros indicados e re-executar o passo 4. Esse processo iterativo só termina quando o passo 4 acusar uma compilação sem erros.

Também é possível que o passo 5 não produza os resultados esperados. Nesse caso, é bem provável que o algoritmo criado no passo 2 não seja adequado. Devemos voltar ao passo 2 e re-examinar o algoritmo, efetuando correções, se necessário, ou criando um outro algoritmo. Em seguida, voltamos aos passos 3, 4 e 5. Somente apenas quando o passo 5 indicar respostas corretas para alguns casos típicos de teste é que podemos terminar o processo, e passar a usar o programa objeto gerado para resolver outras instâncias do problema proposto inicialmente no passo 1.

Quando um algoritmo é desenvolvido , a maior parte da atividade criativa está centrada no passo 2. O passo 3 é relativamente rotineiro, uma vez tendo em mãos o algoritmo. Os demais passos são automatizados.

Finalmente, vale reforçar que o uso de linguagens de programação tem a virtude de libertar o programador dos detalhes específicos e intrincados da arquitetura da máquina. Esses detalhes seriam um peso considerável na atividade de programação se tivessem que ser levados em conta diretamente.

Também o processo de compilação é um grande alívio. Não fosse por ele, para cada arquitetura de máquina disponível teríamos que re-programar o mesmo algoritmo, levando em conta as especificidades da máquina alvo. Com o processo de compilação partimos de um mesmo texto de programa fonte. Para cada máquina tem-se um compilador específico que traduz o programa fonte para um programa objeto que usa as operações primitivas daquela particular máquina. E hoje já dispomos de compiladores para virtualmente todas as linguagens de programação e para as arquiteturas mais comuns.