Capítulo 2

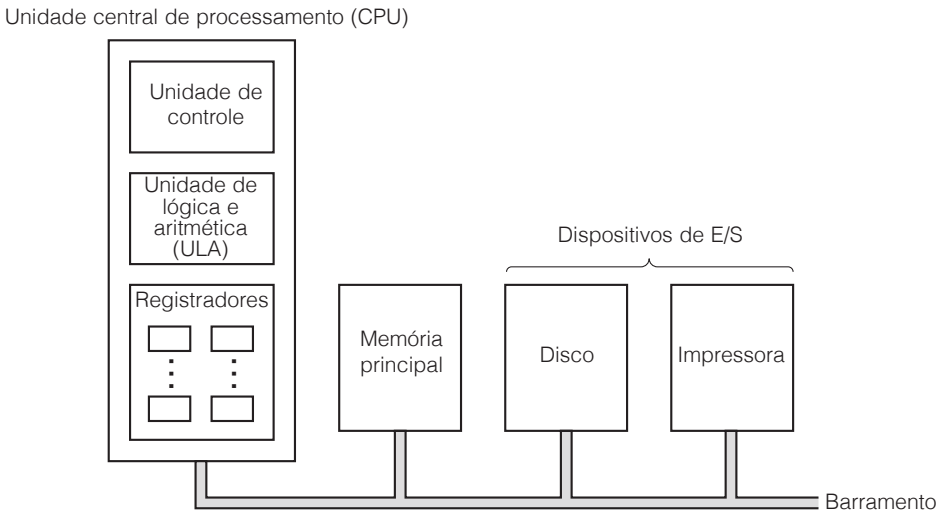
Organização de sistemas de computadores

Um computador digital consiste em um sistema interconectado de processadores, memória e dispositivos de entrada/saída. Este capítulo é uma introdução a esses três componentes e a sua interconexão, como base para o exame mais detalhado de níveis específicos nos cinco capítulos subsequentes. Processadores, memórias e dispositivos de entrada/saída são conceitos fundamentais e estarão presentes em todos os níveis, portanto, iniciaremos nosso estudo da arquitetura de computadores examinando todos os três, um por vez.

2.1 Processadores

A organização de um computador simples com barramento é mostrada na Figura 2.1. A **CPU** (**Central Processing Unit – unidade central de processamento**) é o "cérebro" do computador. Sua função é executar programas armazenados na memória principal buscando suas instruções, examinando-as e então executando-as uma após a outra. Os componentes são conectados por um **barramento**, conjunto de fios paralelos que transmitem endereços, dados e sinais de controle. Barramentos podem ser externos à CPU, conectando-a à memória e aos dispositivos de E/S, mas também podem ser internos, como veremos em breve. Os computadores modernos possuem vários barramentos.

Figura 2.1 A organização de um computador simples com uma CPU e dois dispositivos de E/S.



A CPU é composta por várias partes distintas. A unidade de controle é responsável por buscar instruções na memória principal e determinar seu tipo. A unidade de aritmética e lógica efetua operações como adição e AND (E) booleano para executar as instruções.

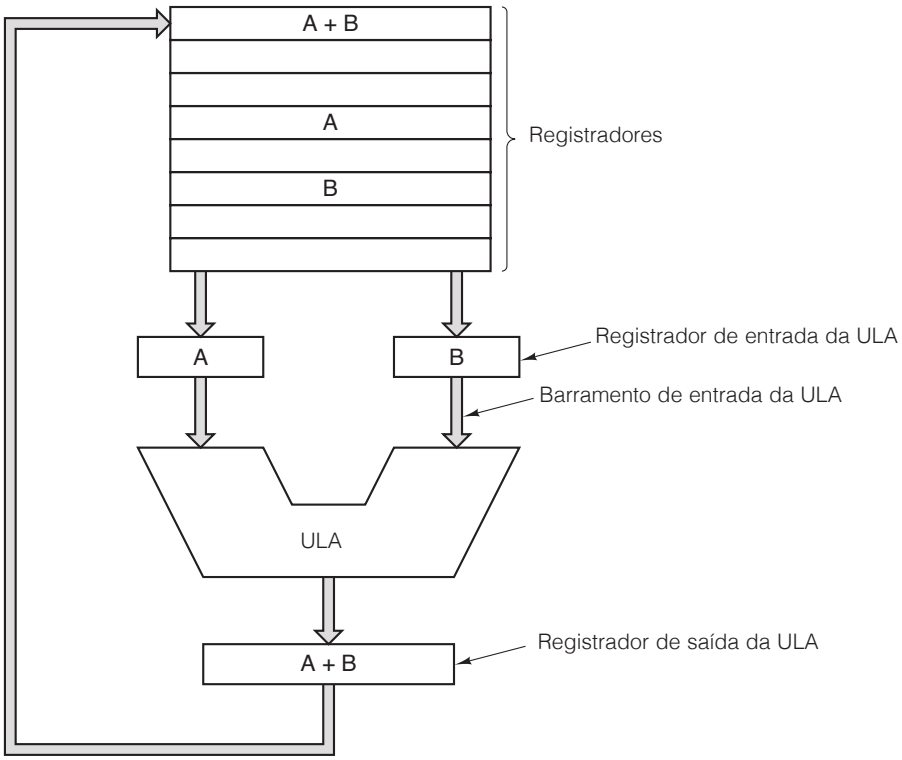
A CPU também contém uma pequena memória de alta velocidade usada para armazenar resultados temporários e para algum controle de informações. Essa memória é composta de uma quantidade de registradores, cada um deles com determinado tamanho e função. Em geral, todos os registradores têm o mesmo tamanho. Cada um pode conter um número, até algum máximo definido pelo tamanho do registrador. Registradores podem ser lidos e escritos em alta velocidade porque são internos à CPU.

O registrador mais importante é o **Contador de Programa** (**PC – Program Counter**), que indica a próxima instrução a ser buscada para execução. (O nome "contador de programa" é um tanto enganoso, porque nada tem a ver com *contar* qualquer coisa; porém, o termo é de uso universal.) Também importante é o **Registrador de Instrução** (**IR – Instruction Register**), que mantém a instrução que está sendo executada no momento em questão. A maioria dos computadores também possui diversos outros registradores, alguns de uso geral, outros de uso específico. Outros registradores são usados pelo sistema operacional para controlar o computador.

2.1.1 Organização da CPU

A organização interna de parte de uma típica CPU de von Neumann é mostrada na Figura 2.2 com mais detalhes. Essa parte é denominada **caminho de dados** e é composta por registradores (em geral 1 a 32), da **ULA** (**unidade lógica e aritmética**) e por diversos barramentos que conectam as partes. Os registradores alimentam dois registradores de entrada da ULA, representados por *A* e *B* na figura. Eles contêm a entrada da ULA enquanto ela está executando alguma operação de computação. O caminho de dados é muito importante em todas as máquinas e nós o discutiremos minuciosamente em todo este livro.

Figura 2.2 O caminho de dados de uma típica máquina de von Neumann.



A ULA efetua adição, subtração e outras operações simples sobre suas entradas, produzindo assim um resultado no registrador de saída, o qual pode ser armazenado em um registrador. Mais tarde, ele pode ser escrito (isto é, armazenado) na memória, se desejado. Nem todos os projetos têm os registradores *A*, *B* e de saída. No exemplo, ilustramos uma adição, mas as ULAs também realizam outras operações.

Grande parte das instruções pode ser dividida em uma de duas categorias: registrador-memória ou registrador-registrador. Instruções registrador-memória permitem que palavras de memória sejam buscadas em registradores, onde podem ser usadas como entradas de ULA em instruções subsequentes, por exemplo. ("Palavras" são as unidades de dados movimentadas entre memória e registradores. Uma palavra pode ser um número inteiro. Discutiremos organização de memória mais adiante neste capítulo.) Outras instruções registrador-memória permitem que registradores voltem à memória para armazenagem.

O outro tipo de instrução é registrador-registrador. Uma instrução registrador-registrador típica busca dois operandos nos registradores, traz os dois até os registradores de entrada da ULA, efetua alguma operação com eles (por exemplo, adição ou AND booleano) e armazena o resultado em um dos registradores. O processo de passar dois operandos pela ULA e armazenar o resultado é denominado **ciclo do caminho de dados** e é o coração da maioria das CPUs. Até certo ponto considerável, ele define o que a máquina pode fazer. Quanto mais rápido for o ciclo do caminho de dados, mais rápido será o funcionamento da máquina.

2.1.2 Execução de instrução

A CPU executa cada instrução em uma série de pequenas etapas. Em termos simples, as etapas são as seguintes:

1. Trazer a próxima instrução da memória até o registrador de instrução.
2. Alterar o contador de programa para que aponte para a próxima instrução.

3. Determinar o tipo de instrução trazida.
4. Se a instrução usar uma palavra na memória, determinar onde essa palavra está.
5. Trazer a palavra para dentro de um registrador da CPU, se necessário.
6. Executar a instrução.
7. Voltar à etapa 1 para iniciar a execução da instrução seguinte.

Tal sequência de etapas costuma ser denominada ciclo **buscar-decodificar-executar**. É fundamental para a operação de todos os computadores.

Essa descrição do modo de funcionamento de uma CPU é muito parecida com um programa escrito em inglês. A Figura 2.3 mostra esse programa informal reescrito como um método Java (isto é, um procedimento) denominado *interpret*. A máquina que está sendo interpretada tem dois registradores visíveis para programas usuários: o contador de programa (PC), para controlar o endereço da próxima instrução a ser buscada, e o acumulador (AC), para acumular resultados aritméticos. Também tem registradores internos para conter a instrução corrente durante sua execução (instr), o tipo da instrução corrente (instr_type), o endereço do operando da instrução (data_loc) e o operando corrente em si (data). Admitimos que as instruções contêm um único endereço de memória. A localização de memória endereçada contém o operando, por exemplo, o item de dado a ser somado ao acumulador.

Figura 2.3 Interpretador para um computador simples (escrito em Java).

```
public class Interp {

    static int PC;                          // contador de programa contém endereço da próxima instr
    static int AC;                          // o acumulador, um registrador para efetuar aritmética
    static int instr;                       // um registrador para conter a instrução corrente
    static int instr_type;                  // o tipo da instrução (opcode)
    static int data_loc;                    // o endereço dos dados, ou –1 se nenhum
    static int data;                        // mantém o operando corrente
    static boolean run_bit = true;          // um bit que pode ser desligado para parar a máquina

    public static void interpret(int memory[ ], int starting_ address) {
        // Esse procedimento interpreta programas para uma máquina simples com instruções que têm
        // um operando na memória. A máquina tem um registrador AC (acumulador), usado para
        // aritmética. A instrução ADD soma um inteiro na memória do AC, por exemplo.
        // O interpretador continua funcionando até o bit de funcionamento ser desligado pela instrução HALT.
        // O estado de um processo que roda nessa máquina consiste em memória, o
        // contador de programa, bit de funcionamento e AC. Os parâmetros de entrada consistem
        / na imagem da memória e no endereço inicial.

    PC = starting_address;
    while (run_bit) {
        instr = memory[PC];                 // busca a próxima instrução e armazena em instr
        PC = PC + 1;                        // incrementa contador de programa
        instr_type = get_instr_type(instr); // determina tipo da instrução
        data_loc = find_data(instr, instr_type); // localiza dados (–1 se nenhum)
        if (data_loc >= 0)                  // se data_loc é –1, não há nenhum operando
           data = memory[data_loc];         // busca os dados
        execute(instr_type, data);          // executa instrução
        }
    }
    private static int get_instr_type(int addr) { ... }
    private static int find_data(int instr, int type) { ... }
    private static void execute(int type, int data) { ... }
}
```

Essa equivalência entre processadores de hardware e interpretadores tem importantes implicações para a organização de computadores e para o projeto de sistemas de computadores. Após a especificação da linguagem de máquina, *L*, para um novo computador, a equipe de projeto pode decidir se quer construir um processador de hardware para executar programas em *L* diretamente ou se quer escrever um interpretador para interpretar programas em *L*. Se a equipe preferir escrever um interpretador, também deve providenciar alguma máquina de hardware para executá-lo. São possíveis ainda certas construções híbridas, com um pouco de execução em hardware, bem como alguma interpretação de software.

Um interpretador subdivide as instruções da máquina em questão em pequenas etapas. Por conseguinte, a máquina na qual o interpretador roda deve ser muito mais simples e menos cara do que seria um processador de hardware para a máquina citada. Essa economia é bastante significativa se a máquina em questão tiver um grande número de instruções e estas forem razoavelmente complicadas, com muitas opções. Basicamente, a economia vem do fato de que o hardware está sendo substituído por software (o interpretador) e custa mais reproduzir hardware do que software.

Os primeiros computadores tinham conjuntos de instruções pequenos, simples. Mas a procura por equipamentos mais poderosos levou, entre outras coisas, a instruções individuais mais poderosas. Logo se descobriu que instruções mais complexas muitas vezes levavam à execução mais rápida do programa mesmo que as instruções individuais demorassem mais para ser executadas. Uma instrução de ponto flutuante é um exemplo de instrução mais complexa. O suporte direto para acessar elementos matriciais é outro. Às vezes, isso era simples como observar que as mesmas duas instruções muitas vezes ocorriam em sequência, de modo que uma única instrução poderia fazer o trabalho de ambas.

As instruções mais complexas eram melhores porque a execução de operações individuais às vezes podia ser sobreposta ou então executada em paralelo usando hardware diferente. No caso de computadores caros, de alto desempenho, o custo desse hardware extra poderia ser justificado de imediato. Assim, computadores caros, de alto desempenho, passaram a ter mais instruções do que os de custo mais baixo. Contudo, requisitos de compatibilidade de instruções e o custo crescente do desenvolvimento de software criaram a necessidade de executar instruções complexas mesmo em computadores de baixo custo, nos quais o custo era mais importante do que a velocidade.

No final da década de 1950, a IBM (na época a empresa que dominava o setor de computadores) percebeu que prestar suporte a uma única família de máquinas, todas executando as mesmas instruções, tinha muitas vantagens, tanto para a IBM quanto para seus clientes. Então, a empresa introduziu o termo **arquitetura** para descrever esse nível de compatibilidade. Uma nova família de computadores teria uma só arquitetura, mas muitas implementações diferentes que poderiam executar o mesmo programa e seriam diferentes apenas em preço e velocidade. Mas como construir um computador de baixo custo que poderia executar todas as complicadas instruções de máquinas caras, de alto desempenho?

A resposta foi a interpretação. Essa técnica, que já tinha sido sugerida por Maurice Wilkes (1951), permitia o projeto de computadores simples e de menor custo, mas que, mesmo assim, podiam executar um grande número de instruções. O resultado foi a arquitetura IBM System/360, uma família de computadores compatíveis que abrangia quase duas ordens de grandeza, tanto em preço quanto em capacidade. Uma implementação de hardware direto (isto é, não interpretado) era usada somente nos modelos mais caros.

Computadores simples com instruções interpretadas também tinham outros benefícios, entre os quais os mais importantes eram:

1. A capacidade de corrigir em campo instruções executadas incorretamente ou até compensar deficiências de projeto no hardware básico.
2. A oportunidade de acrescentar novas instruções a um custo mínimo, mesmo após a entrega da máquina.
3. Projeto estruturado que permitia desenvolvimento, teste e documentação eficientes de instruções complexas.

À medida que o mercado explodia em grande estilo na década de 1970 e as capacidades de computação cresciam depressa, a demanda por máquinas de baixo custo favorecia projetos de computadores que usassem interpretadores. A capacidade de ajustar hardware e interpretador para um determinado conjunto de instruções surgiu como um projeto muito eficiente em custo para processadores. À medida que a tecnologia subjacente dos semicondutores avançava, as vantagens do custo compensavam as oportunidades de desempenho mais alto e as arquiteturas baseadas em interpretador se tornaram o modo convencional de projetar computadores. Quase todos os novos computadores projetados na década de 1970, de microcomputadores a *mainframes*, tinham a interpretação como base.

No final da década de 1970, a utilização de processadores simples que executavam interpretadores tinha se propagado em grande escala, exceto entre os modelos mais caros e de desempenho mais alto, como o Cray-1 e a série Cyber da Control Data. A utilização de um interpretador eliminava as limitações de custo inerentes às instruções complexas, de modo que os projetistas começaram a explorar instruções muito mais complexas, em particular os modos de especificar os operandos a utilizar.

A tendência alcançou seu ponto mais alto com o VAX da Digital Equipment Corporation, que tinha várias centenas de instruções e mais de 200 modos diferentes de especificar os operandos a serem usados em cada instrução. Infelizmente, desde o início a arquitetura do VAX foi concebida para ser executada com um interpretador, sem dar muita atenção à realização de um modelo de alto desempenho. Esse modo de pensar resultou na inclusão de um número muito grande de instruções de valor marginal e que eram difíceis de executar diretamente. Essa omissão mostrou ser fatal para o VAX e, por fim, também para a DEC (a Compaq comprou a DEC em 1998 e a Hewlett-Packard comprou a Compaq em 2001).

Embora os primeiros microprocessadores de 8 bits fossem máquinas muito simples com conjuntos de instruções muito simples, no final da década de 1970 até os microprocessadores tinham passado para projetos baseados em interpretador. Durante esse período, um dos maiores desafios enfrentados pelos projetistas de microprocessadores era lidar com a crescente complexidade, possibilitada por meio de circuitos integrados. Uma importante vantagem do método baseado em interpretador era a capacidade de projetar um processador simples e confinar quase toda a complexidade na memória que continha o interpretador. Assim, um projeto complexo de hardware se transformou em um projeto complexo de software.

O sucesso do Motorola 68000, que tinha um grande conjunto de instruções interpretadas, e o concomitante fracasso do Zilog Z8000 (que tinha um conjunto de instruções tão grande quanto, mas sem um interpretador) demonstraram as vantagens de um interpretador para levar um novo microprocessador rapidamente ao mercado. Esse sucesso foi ainda mais surpreendente dada a vantagem de que o Zilog desfrutava (o antecessor do Z8000, o Z80, era muito mais popular do que o antecessor do 68000, o 6800). Claro que outros fatores também contribuíram para isso, e um dos mais importantes foi a longa história da Motorola como fabricante de chips e a longa história da Exxon (proprietária da Zilog) como empresa de petróleo, e não como fabricante de chips.

Outro fator a favor da interpretação naquela época foi a existência de memórias rápidas somente de leitura, denominadas **memórias de controle**, para conter os interpretadores. Suponha que uma instrução interpretada típica precisasse de 10 instruções do interpretador, denominadas **microinstruções**, a 100 ns cada, e duas referências à memória principal a 500 ns cada. Então, o tempo total de execução era 2.000 ns, apenas um fator de dois pior do que o melhor que a execução direta podia conseguir. Se a memória de controle não estivesse disponível, a instrução levaria 6.000 ns. Uma penalidade de fator seis é muito mais difícil de aceitar do que uma penalidade de fator dois.

2.1.3 RISC *versus* CISC

Durante o final da década de 1970, houve experiências com instruções muito complexas que eram possibilitadas pelo interpretador. Os projetistas tentavam fechar a "lacuna semântica" entre o que as máquinas podiam fazer e o que as linguagens de programação de alto nível demandavam. Quase ninguém pensava em projetar máquinas mais simples, exatamente como agora não há muita pesquisa na área de projeto de planilhas, redes, servidores Web etc. menos poderosos (o que talvez seja lamentável).

Um grupo que se opôs à tendência e tentou incorporar algumas das ideias de Seymour Cray em um minicomputador de alto desempenho foi liderado por John Cocke na IBM. Esse trabalho resultou em um minicomputador denominado **801**. Embora a IBM nunca tenha lançado essa máquina no mercado e os resultados tenham sido publicados só muitos anos depois (Radin, 1982), a notícia vazou e outros começaram a investigar arquiteturas semelhantes.

Em 1980, um grupo em Berkeley, liderado por David Patterson e Carlo Séquin, começou a projetar chips para CPUs VLSI que não usavam interpretação (Patterson, 1985; Patterson e Séquin, 1982). Eles cunharam o termo **RISC** para esse conceito e deram ao seu chip de CPU o nome RISC I CPU, seguido logo depois pelo RISC II. Um pouco mais tarde, em 1981, do outro lado da baía de São Francisco, em Stanford, John Hennessy projetou e fabricou um chip um pouco diferente, que ele chamou de **MIPS** (Hennessy, 1984). Esses chips evoluíram para produtos de importância comercial, o SPARC e o MIPS, respectivamente.

Esses novos processadores tinham diferenças significativas em relação aos que havia no comércio naquela época. Uma vez que essas novas CPUs não eram compatíveis com os produtos existentes, seus projetistas tinham liberdade para escolher novos conjuntos de instruções que maximizassem o desempenho total do sistema. Embora a ênfase inicial estivesse dirigida a instruções simples, que podiam ser executadas rapidamente, logo se percebeu que projetar instruções que podiam ser **emitidas** (iniciadas) rapidamente era a chave do bom desempenho. Na verdade, o tempo que uma instrução demorava importava menos do que quantas delas podiam ser iniciadas por segundo.

Na época em que o projeto desses processadores simples estava no início, a característica que chamou a atenção de todos era o número relativamente pequeno de instruções disponíveis, em geral cerca de 50. Esse número era muito menor do que as 200 a 300 de computadores como o VAX da DEC e os grandes *mainframes* da IBM. De fato, o acrônimo RISC quer dizer **Reduced Instruction Set Computer** (computador com conjunto de instruções reduzido), em comparação com CISC, que significa **Complex Instruction Set Computer** (computador com conjunto de instruções complexo), uma referência nada sutil ao VAX que, na época, dominava os departamentos de ciência da computação das universidades. Hoje em dia, poucas pessoas acham que o tamanho do conjunto de instruções seja um assunto importante, mas o nome pegou.

Encurtando a história, seguiu-se uma grande guerra santa, com os defensores do RISC atacando a ordem estabelecida (VAX, Intel, grandes *mainframes* da IBM). Eles afirmavam que o melhor modo de projetar um computador era ter um pequeno número de instruções simples que executassem em um só ciclo do caminho de dados da Figura 2.2, ou seja, buscar dois registradores, combiná-los de algum modo (por exemplo, adicionando-os ou fazendo AND) e armazenar o resultado de volta em um registrador. O argumento desses pesquisadores era de que, mesmo que uma máquina RISC precisasse de quatro ou cinco instruções para fazer o que uma CISC fazia com uma só, se as instruções RISC fossem dez vezes mais rápidas (porque não eram interpretadas), o RISC vencia. Também vale a pena destacar que, naquele tempo, a velocidade de memórias principais tinha alcançado a velocidade de memórias de controle somente de leitura, de modo que a penalidade imposta pela interpretação tinha aumentado demais, o que favorecia muito as máquinas RISC.

Era de imaginar que, dadas as vantagens de desempenho da tecnologia RISC, as máquinas RISC (como a Sun UltraSPARC) passariam como rolo compressor sobre as máquinas CISC (tal como a Pentium da Intel) existentes no mercado. Nada disso aconteceu. Por quê?

Antes de tudo, há a questão da compatibilidade e dos bilhões de dólares que as empresas tinham investido em software para a linha Intel. Em segundo lugar, o que era surpreendente, a Intel conseguiu empregar as mesmas ideias mesmo em uma arquitetura CISC. A partir do 486, as CPUs da Intel contêm um núcleo RISC que executa as instruções mais simples (que normalmente são as mais comuns) em um único ciclo do caminho de dados, enquanto interpreta as mais complicadas no modo CISC de sempre. O resultado disso é que as instruções comuns são rápidas e as menos comuns são lentas. Mesmo que essa abordagem híbrida não seja tão rápida quanto um projeto RISC puro, ela resulta em desempenho global competitivo e ainda permite que softwares antigos sejam executados sem modificação.

2.1.4 Princípios de projeto para computadores modernos

Agora que já se passaram mais de duas décadas desde que as primeiras máquinas RISC foram lançadas, certos princípios de projeto passaram a ser aceitos como um bom modo de projetar computadores, dado o estado atual da tecnologia de hardware. Se ocorrer uma importante mudança na tecnologia (por exemplo, se, de repente, um novo processo de fabricação fizer o ciclo de memória ficar dez vezes mais rápido do que o tempo de ciclo da CPU), todas as apostas perdem. Assim, os projetistas de máquinas devem estar sempre de olho nas mudanças tecnológicas que possam afetar o equilíbrio entre os componentes.

Dito isso, há um conjunto de princípios de projeto, às vezes denominados **princípios de projeto RISC**, que os arquitetos de CPUs de uso geral se esforçam por seguir. Limitações externas, como a exigência de compatibilidade com alguma arquitetura existente, muitas vezes exigem uma solução de conciliação de tempos em tempos, mas esses princípios são metas que a maioria dos projetistas se esforça para cumprir. A seguir, discutiremos os principais.

• Todas as instruções são executadas diretamente por hardware

Todas as instruções comuns são executadas diretamente pelo hardware – não são interpretadas por microinstruções. Eliminar um nível de interpretação dá alta velocidade à maioria das instruções. No caso de computadores que executam conjuntos de instruções CISC, as instruções mais complexas podem ser subdivididas em partes separadas que então podem ser executadas como uma sequência de microinstruções. Essa etapa extra torna a máquina mais lenta, porém, para instruções que ocorrem com menos frequência, isso pode ser aceitável.

• Maximize a taxa de execução das instruções

Computadores modernos recorrem a muitos truques para maximizar seu desempenho, entre os quais o principal é tentar iniciar o máximo possível de instruções por segundo. Afinal, se você puder emitir 500 milhões de instruções por segundo, terá construído um processador de 500 MIPS, não importa quanto tempo elas realmente levem para ser concluídas. (**MIPS** quer dizer Milhões de Instruções Por Segundo. O processador MIPS recebeu esse nome como um trocadilho desse acrônimo. Oficialmente, ele significa **Microprocessor without Interlocked Pipeline Stages** – microprocessador sem estágios paralelos de interbloqueio.) Esse princípio sugere que o paralelismo pode desempenhar um importante papel na melhoria do desempenho, uma vez que emitir grandes quantidades de instruções lentas em curto intervalo de tempo só é possível se várias instruções puderem ser executadas ao mesmo tempo.

Embora as instruções sempre sejam encontradas na ordem do programa, nem sempre elas são executadas nessa mesma ordem (porque algum recurso necessário pode estar ocupado) e não precisam terminar na ordem do programa. É claro que, se a instrução 1 estabelece um registrador e a instrução 2 usa esse registrador, deve-se tomar muito cuidado para garantir que a instrução 2 não leia o registrador até que ele contenha o valor correto. Fazer isso funcionar direito requer muito controle, mas possibilita ganhos de desempenho por executar várias instruções ao mesmo tempo.

• Instruções devem ser fáceis de decodificar

Um limite crítico para a taxa de emissão de instruções é a decodificação de instruções individuais para determinar quais recursos elas necessitam. Qualquer coisa que possa ajudar nesse processo é útil. Isso inclui fazer instruções regulares, de comprimento fixo, com um pequeno número de campos. Quanto menor o número de formatos diferentes para as instruções, melhor.

• Somente LOAD e STORE devem referenciar a memória

Um dos modos mais simples de subdividir operações em etapas separadas é requerer que os operandos para a maioria das instruções venham de registradores da CPU e a eles retornem. A operação de movimentação de operandos da memória para registradores pode ser executada em instruções separadas. Uma vez que o acesso à

memória pode levar um longo tempo, e que o atraso é imprevisível, o melhor é sobrepor essas instruções a outras se elas nada fizerem exceto movimentar operandos entre registradores e memória. Essa observação significa que somente instruções LOAD e STORE devem referenciar a memória. Todas as outras devem operar apenas em registradores.

● Providencie muitos registradores

Visto que o acesso à memória é relativamente lento, é preciso providenciar muitos registradores (no mínimo, 32) de modo que, assim que uma palavra for buscada, ela possa ser mantida em um registrador até não ser mais necessária. Esgotar os registradores e ter de descarregá-los de volta à memória só para ter de recarregá-los mais tarde é indesejável e deve ser evitado o máximo possível. A melhor maneira de conseguir isso é ter um número suficiente de registradores.

2.1.5 Paralelismo no nível de instrução

Arquitetos de computadores estão sempre se esforçando para melhorar o desempenho das máquinas que projetam. Fazer os chips funcionarem com maior rapidez aumentando suas velocidades de *clock* é um modo, mas, para cada novo projeto, há um limite para o que é possível fazer por força bruta naquele momento da História. Por conseguinte, grande parte dos arquitetos de computadores busca o paralelismo (fazer duas ou mais coisas ao mesmo tempo) como um meio de conseguir desempenho ainda melhor para dada velocidade de *clock*.

O paralelismo tem duas formas gerais, a saber, no nível de instrução e no nível de processador. Na primeira, o paralelismo é explorado dentro de instruções individuais para obter da máquina mais instruções por segundo. Na última, várias CPUs trabalham juntas no mesmo problema. Cada abordagem tem seus próprios méritos. Nesta seção, vamos estudar o paralelismo no nível de instrução; na seção seguinte, estudaremos o paralelismo no nível de processador.

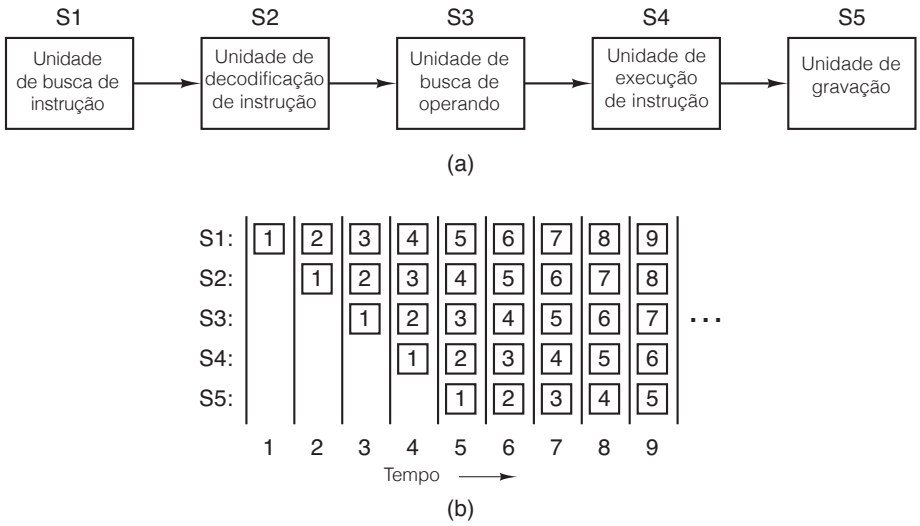
● *Pipelining* (paralelismo)

Há anos sabe-se que o processo de buscar instruções na memória é um grande gargalo na velocidade de execução da instrução. Para amenizar esse problema, os computadores, desde o IBM Stretch (1959), tinham a capacidade de buscar instruções na memória antecipadamente, de maneira que estivessem presentes quando necessárias. Essas instruções eram armazenadas em um conjunto de registradores denominado ***buffer* de busca antecipada** (ou *prefetch buffer*). Desse modo, quando necessária, uma instrução podia ser apanhada no *buffer* de busca antecipada, em vez de esperar pela conclusão de uma leitura da memória.

Na verdade, a busca antecipada divide a execução da instrução em duas partes: a busca e a execução propriamente dita. O conceito de ***pipeline*** (paralelismo, canalização) amplia muito mais essa estratégia. Em vez de dividir a execução da instrução em apenas duas partes, muitas vezes ela é dividida em muitas partes (uma dúzia ou mais), cada uma manipulada por uma parte dedicada do hardware, e todas elas podem executar em paralelo.

A Figura 2.4(a) ilustra um *pipeline* com cinco unidades, também denominadas **estágios**. O estágio 1 busca a instrução na memória e a coloca em um *buffer* até que ela seja necessária. O estágio 2 decodifica a instrução, determina seu tipo e de quais operandos ela necessita. O estágio 3 localiza e busca os operandos, seja nos registradores, seja na memória. O estágio 4 é que realiza o trabalho de executar a instrução, normalmente fazendo os operandos passarem pelo caminho de dados da Figura 2.2. Por fim, o estágio 5 escreve o resultado de volta no registrador adequado.

Figura 2.4 (a) *Pipeline* de cinco estágios. (b) Estado de cada estágio como uma função do tempo. São ilustrados nove ciclos de *clock*.



Na Figura 2.4(b), vemos como o *pipeline* funciona em função do tempo. Durante o ciclo de *clock* 1, o estágio S1 está trabalhando na instrução 1, buscando-a na memória. Durante o ciclo 2, o estágio S2 decodifica a instrução 1, enquanto o estágio S1 busca a instrução 2. Durante o ciclo 3, o estágio S3 busca os operandos para a instrução 1, o estágio S2 decodifica a instrução 2 e o estágio S1 busca a terceira instrução. Durante o ciclo 4, o estágio S4 executa a instrução 1, S3 busca os operandos para a instrução 2, S2 decodifica a instrução 3 e S1 busca a instrução 4. Por fim, durante o ciclo 5, S5 escreve (grava) o resultado da instrução 1 de volta ao registrador, enquanto os outros estágios trabalham nas instruções seguintes.

Vamos considerar uma analogia para esclarecer melhor o conceito de *pipelining*. Imagine uma fábrica de bolos na qual a operação de produção dos bolos e a operação da embalagem para expedição são separadas. Suponha que o departamento de expedição tenha uma longa esteira transportadora ao longo da qual trabalham cinco funcionários (unidades de processamento). A cada 10 segundos (o ciclo de *clock*), o funcionário 1 coloca uma embalagem de bolo vazia na esteira. A caixa é transportada até o funcionário 2, que coloca um bolo dentro dela. Um pouco mais tarde, a caixa chega à estação do funcionário 3, onde é fechada e selada. Em seguida, prossegue até o funcionário 4, que coloca uma etiqueta na embalagem. Por fim, o funcionário 5 retira a caixa da esteira e a coloca em um grande contêiner que mais tarde será despachado para um supermercado. Em termos gerais, esse é o modo como um *pipeline* de computador também funciona: cada instrução (bolo) passa por diversos estágios de processamento antes de aparecer já concluída na extremidade final.

Voltando ao nosso *pipeline* da Figura 2.4, suponha que o tempo de ciclo dessa máquina seja 2 ns. Sendo assim, uma instrução leva 10 ns para percorrer todo o caminho do *pipeline* de cinco estágios. À primeira vista, como uma instrução demora 10 ns, parece que a máquina poderia funcionar em 100 MIPS, mas, na verdade, ela funciona muito melhor do que isso. A cada ciclo de *clock* (2 ns), uma nova instrução é concluída, portanto, a velocidade real de processamento é 500 MIPS, e não 100 MIPS.

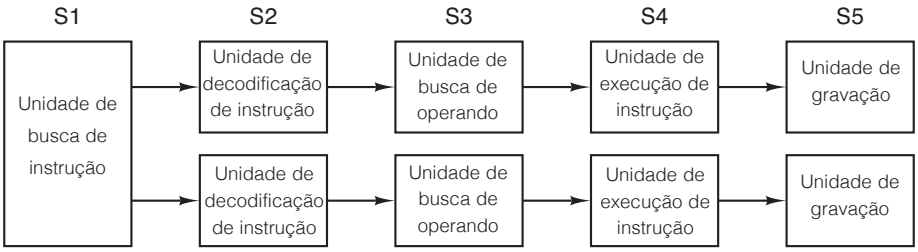
O *pipelining* permite um compromisso entre **latência** (o tempo que demora para executar uma instrução) e **largura de banda de processador** (quantos MIPS a CPU tem). Com um tempo de ciclo de T ns e n estágios no *pipeline*, a latência é nT ns porque cada instrução passa por n estágios, cada um dos quais demora T ns.

Visto que uma instrução é concluída a cada ciclo de *clock* e que há $10^9/T$ ciclos de *clock* por segundo, o número de instruções executadas por segundo é $10^9/T$. Por exemplo, se T = 2 ns, 500 milhões de instruções são executadas a cada segundo. Para obter o número de MIPS, temos de dividir a taxa de execução de instrução por 1 milhão para obter $(10^9/T)/10^6 = 1.000/T$ MIPS. Em teoria, poderíamos medir taxas de execução de instrução em BIPS em vez de MIPS, mas ninguém faz isso, portanto, nós também não o faremos.

Arquiteturas superescalares

Se um *pipeline* é bom, então certamente dois *pipelines* são ainda melhores. Um projeto possível para uma CPU com dois *pipelines*, com base na Figura 2.4, é mostrado na Figura 2.5. Nesse caso, uma única unidade de busca de instruções busca pares de instruções ao mesmo tempo e coloca cada uma delas em seu próprio *pipeline*, completo com sua própria ULA para operação paralela. Para poder executar em paralelo, as duas instruções não devem ter conflito de utilização de recursos (por exemplo, registradores) e nenhuma deve depender do resultado da outra. Assim como em um *pipeline* único, ou o compilador deve garantir que essa situação aconteça (isto é, o hardware não verifica e dá resultados incorretos se as instruções não forem compatíveis), ou os conflitos deverão ser detectados e eliminados durante a execução usando hardware extra.

Figura 2.5 *Pipelines* duplos de cinco estágios com uma unidade de busca de instrução em comum.

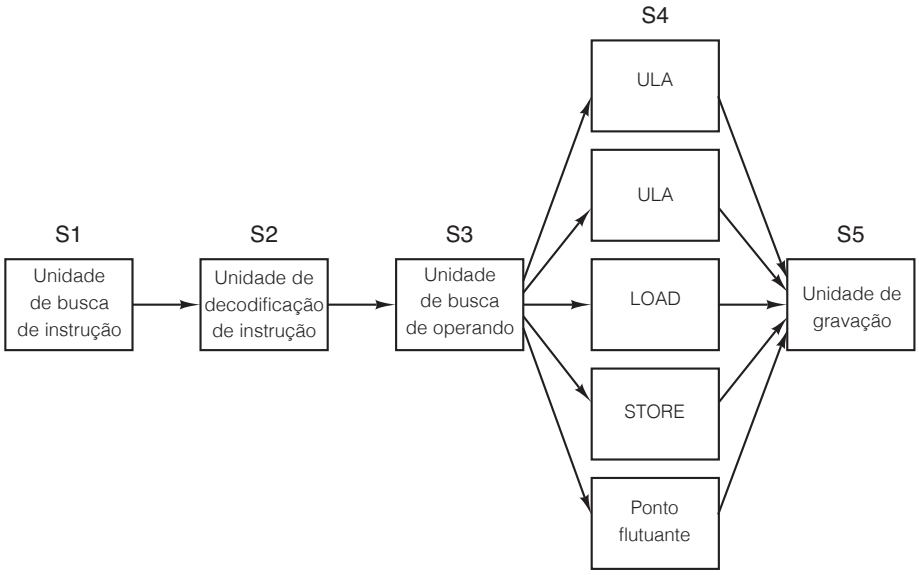


Embora *pipelines*, simples ou duplos, sejam usados em sua maioria em máquinas RISC (o 386 e seus antecessores não tinham nenhum), a partir do 486 a Intel começou a acrescentar *pipelines* de dados em suas CPUs. O 486 tinha um *pipeline* e o Pentium original tinha *pipelines* de cinco estágios mais ou menos como os da Figura 2.5, embora a exata divisão do trabalho entre os estágios 2 e 3 (denominados decode-1 e decode-2) era ligeiramente diferente do que em nosso exemplo. O *pipeline* principal, denominado ***pipeline* u**, podia executar uma instrução Pentium qualquer. O segundo, denominado ***pipeline* v**, podia executar apenas instruções com números inteiros (e também uma instrução simples de ponto flutuante – FXCH).

Regras fixas determinavam se um par de instruções era compatível e, portanto, se elas podiam ser executadas em paralelo. Se as instruções em um par não fossem simples o suficiente ou se fossem incompatíveis, somente a primeira era executada (no *pipeline* u). A segunda era retida para fazer par com a instrução seguinte. Instruções eram sempre executadas em ordem. Assim, os compiladores específicos para Pentium que produziam pares compatíveis podiam produzir programas de execução mais rápidos do que compiladores mais antigos. Medições mostraram que um código de execução Pentium otimizado para ele era exatamente duas vezes mais rápido para programas de inteiros do que um 486 que executava à mesma velocidade de *clock* (Pountain, 1993). Esse ganho podia ser atribuído inteiramente ao segundo *pipeline*.

Passar para quatro *pipelines* era concebível, mas exigiria duplicar muito hardware (cientistas da computação, ao contrário de especialistas em folclore, não acreditam no número três). Em vez disso, uma abordagem diferente é utilizada em CPUs de topo de linha. A ideia básica é ter apenas um único *pipeline*, mas lhe dar várias unidades funcionais, conforme mostra a Figura 2.6. Por exemplo, a arquitetura Intel Core tem uma estrutura semelhante à dessa figura, que será discutida no Capítulo 4. O termo **arquitetura superescalar** foi cunhado para essa técnica em 1987 (Agerwala e Cocke, 1987). Entretanto, suas raízes remontam a mais de 40 anos, ao computador CDC 6600. O 6600 buscava uma instrução a cada 100 ns e a passava para uma das 10 unidades funcionais para execução paralela enquanto a CPU saía em busca da próxima instrução.

Figura 2.6 Processador superescalar com cinco unidades funcionais.



A definição de "superescalar" evoluiu um pouco ao longo do tempo. Agora, ela é usada para descrever processadores que emitem múltiplas instruções – frequentemente, quatro ou seis – em um único ciclo de *clock*. Claro que uma CPU superescalar deve ter várias unidades funcionais para passar todas essas instruções. Uma vez que, em geral, os processadores superescalares têm um só *pipeline*, tendem a ser parecidos com os da Figura 2.6.

Usando essa definição, o 6600 não era tecnicamente um computador superescalar, pois emitia apenas uma instrução por ciclo. Todavia, o efeito era quase o mesmo: instruções eram terminadas em uma taxa muito mais alta do que podiam ser executadas. A diferença conceitual entre uma CPU com um *clock* de 100 ns que executa uma instrução a cada ciclo para um grupo de unidades funcionais e uma CPU com um *clock* de 400 ns que executa quatro instruções por ciclo para o mesmo grupo de unidades funcionais é muito pequena. Em ambos os casos, a ideia fundamental é que a taxa final é muito mais alta do que a taxa de execução, sendo a carga de trabalho distribuída entre um conjunto de unidades funcionais.

Implícito à ideia de um processador superescalar é que o estágio S3 pode emitir instruções com rapidez muito maior do que o estágio S4 é capaz de executá-las. Se o estágio S3 executasse uma instrução a cada 10 ns e todas as unidades funcionais pudessem realizar seu trabalho em 10 ns, nunca mais do que uma unidade estaria ocupada ao mesmo tempo, o que negaria todo o raciocínio. Na verdade, grande parte das unidades funcionais no estágio 4 leva um tempo bem maior do que um ciclo de *clock* para executar, decerto as que acessam memória ou efetuam aritmética de ponto flutuante. Como pode ser visto na figura, é possível ter várias ULAs no estágio S4.

2.1.6 Paralelismo no nível do processador

A demanda por computadores cada vez mais rápidos parece ser insaciável. Astrônomos querem simular o que aconteceu no primeiro microssegundo após o *Big Bang*, economistas querem modelar a economia mundial e adolescentes querem se divertir com jogos multimídia em 3D com seus amigos virtuais pela Internet. Embora as CPUs estejam cada vez mais rápidas, haverá um momento em que elas terão problemas com a velocidade da luz, que provavelmente permanecerá a 20 cm/nanossegundo em fio de cobre ou fibra ótica, não importando o grau de inteligência dos engenheiros da Intel. Chips mais velozes também produzem mais calor, cuja dissipação é um problema. De fato, a dificuldade para se livrar do calor produzido é o principal motivo pelo qual as velocidades de *clock* da CPU se estagnaram na última década.

Paralelismo no nível de instrução ajuda um pouco, mas *pipelining* e operação superescalar raramente rendem mais do que um fator de cinco ou dez. Para obter ganhos de 50, 100 ou mais, a única maneira é projetar computadores com várias CPUs; portanto, agora vamos ver como alguns deles são organizados.

Computadores paralelos

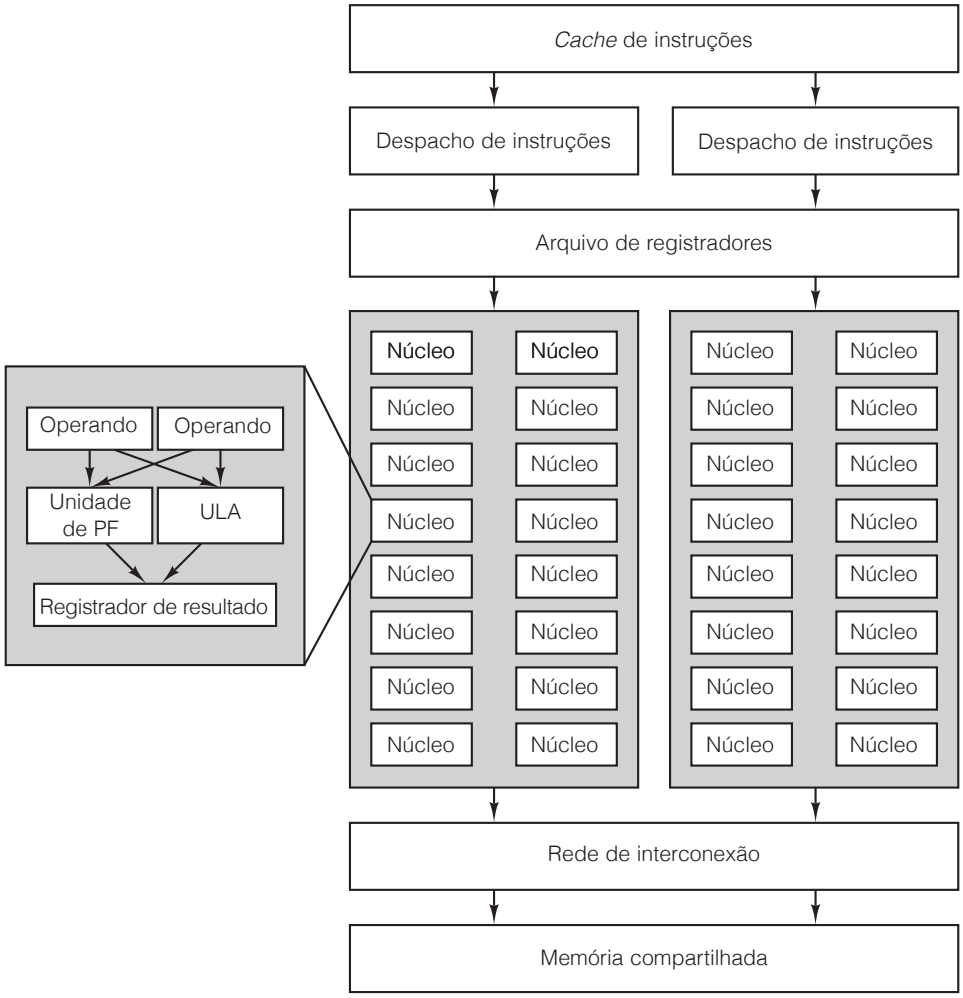
Um número substancial de problemas em domínios de cálculo como ciências físicas, engenharia e gráficos de computador envolve laços e matrizes, ou então tem estrutura de alta regularidade. Muitas vezes, os mesmos cálculos são efetuados em muitos conjuntos diferentes de dados ao mesmo tempo. A regularidade e a estrutura desses programas os tornam alvos especialmente fáceis para aceleração por meio de execução paralela. Há dois métodos que têm sido usados para executar esses programas altamente regulares de modo rápido e eficaz: processadores SIMD e processadores vetoriais. Embora esses dois esquemas guardem notáveis semelhanças na maioria de seus aspectos, por ironia o primeiro deles é considerado um computador paralelo, enquanto o segundo é considerado uma extensão de um processador único.

Computadores paralelos de dados encontraram muitas aplicações bem-sucedidas como consequência de sua notável eficiência. Eles são capazes de produzir poder de computação significativo com menos transistores do que os métodos alternativos. Gordon Moore (da lei de Moore) observou que o silício custa cerca de 1 bilhão de dólares por acre (4.047 m²). Assim, quanto mais poder de computação puder ser espremido desse acre de silício, mais dinheiro uma empresa de computador poderá obter vendendo silício. Os processadores paralelos de dados são um dos meios mais eficientes de espremer o desempenho do silício. Como todos os processadores estão rodando a mesma instrução, o sistema só precisa de um "cérebro" controlando o computador. Em consequência, o processador só precisa de um estágio de busca, um estágio de decodificação e um conjunto de lógica de controle. Essa é uma enorme economia no silício, que dá aos computadores paralelos uma grande vantagem sobre outros processadores, desde que o software que eles estejam rodando seja altamente regular, com bastante paralelismo.

Um **processador SIMD (Single Instruction-stream Multiple Data-stream**, ou **fluxo único de instruções, fluxo múltiplo de dados**) consiste em um grande número de processadores idênticos que efetuam a mesma sequência de instruções sobre diferentes conjuntos de dados. O primeiro processador SIMD do mundo foi o ILLIAC IV da Universidade de Illinois (Bouknight et al., 1972). O projeto original do ILLIAC IV consistia em quatro quadrantes, cada um deles com uma grade quadrada de 8 × 8 elementos de processador/memória. Uma única unidade de controle por quadrante transmitia uma única instrução a todos os processadores, que era executada no mesmo passo por todos eles, cada um usando seus próprios dados de sua própria memória. Por causa de um excesso de custo, somente um quadrante de 50 megaflops (milhões de operações de ponto flutuante por segundo) foi construído; se a construção da máquina inteira de 1 gigaflop tivesse sido concluída, ela teria duplicado a capacidade de computação do mundo inteiro.

As modernas unidades de processamento de gráficos (GPUs) contam bastante com o processamento SIMD para fornecer poder computacional maciço com poucos transistores. O processamento de gráficos foi apropriado para processadores SIMD porque a maioria dos algoritmos é altamente regular, com operações repetidas sobre *pixels*, vértices, texturas e arestas. A Figura 2.7 mostra o processador SIMD no núcleo da GPU Fermi da Nvidia. A GPU Fermi contém até 16 multiprocessadores de fluxo (com memória compartilhada – SM) SIMD, com cada multiprocessador contendo 32 processadores SIMD. A cada ciclo, o escalonador seleciona dois *threads* para executar no processador SIMD. A próxima instrução de cada *thread* é então executada em até 16 processadores SIMD, embora possivelmente menos se não houver paralelismo de dados suficiente. Se cada *thread* for capaz de realizar 16 operações por ciclo, um núcleo GPU Fermi totalmente carregado com 32 multiprocessadores realizará incríveis 512 operações por ciclo. Esse é um feito impressionante, considerando que uma CPU *quad-core* de uso geral com tamanho semelhante lutaria para conseguir 1/32 desse processamento.

Figura 2.7 O núcleo SIMD da unidade de processamento de gráficos Fermi.



Para um programador, um **processador vetorial** se parece muito com um processador SIMD. Assim como um processador SIMD, ele é muito eficiente para executar uma sequência de operações em pares de elementos de dados. Porém, diferente de um processador SIMD, todas as operações de adição são efetuadas em uma única unidade funcional, de alto grau de paralelismo. A Cray Research, empresa fundada por Seymour Cray, produziu muitos processadores vetoriais, começando com o Cray-1 em 1974 e continuando até os modelos atuais.

Processadores SIMD, bem como processadores vetoriais, trabalham com matrizes de dados. Ambos executam instruções únicas que, por exemplo, somam os elementos aos pares para dois vetores. Porém, enquanto o processador SIMD faz isso com tantos somadores quantos forem os elementos do vetor, o processador vetorial tem o conceito de um **registrador vetorial**, que consiste em um conjunto de registradores convencionais que podem ser carregados com base na memória em uma única instrução que, na verdade, os carrega serialmente com base na memória. Então, uma instrução de adição vetorial efetua as adições a partir dos elementos de dois desses vetores, alimentando-os em um somador com paralelismo (*pipelined*) com base em dois registradores vetoriais. O resultado do somador é outro vetor, que pode ser armazenado em um registrador vetorial ou usado diretamente como um operando para outra operação vetorial. As instruções SSE (Streaming SIMD Extension) disponíveis na arquitetura Intel Core utilizam esse modelo de execução para agilizar o cálculo altamente regular, como multimídia e software científico. Nesse aspecto particular, o ILLIAC IV é um dos ancestrais da arquitetura Intel Core.

Tanenbaum • Austin

Organização estruturada de computadores

6ª EDIÇÃO

ALWAYS LEARNING

PEARSON