APOSTILA

DE

OC 2

SUMÁRIO

1 INTRODUÇÃO 4
1.1 O CONCEITO DE ARQUITETURA 4
1.2 NÍVEIS DE ARQUITETURA 5
2 CONCEITOS BÁSICOS DE ARQUITETURA DE PROCESSADOR 7
2.1 A SEÇÃO DE PROCESSAMENTO 7
2.2 A EXECUÇÃO DE INSTRUÇÕES 9
2.3 A SEÇÃO DE CONTROLE 12
2.4 O SINAL DE CLOCK 14
2.5 IMPLEMENTAÇÃO DA UNIDADE DE CONTROLE 15
3 O CONJUNTO DE INSTRUÇÕES DO PROCESSADOR 18
3.1 CONJUNTO DE INSTRUÇÕES NO CONTEXTO DE SOFTWARE 18
3.2 TIPOS DE INSTRUÇÕES E DE OPERANDOS 19
3.3 NÚMERO E LOCALIZAÇÃO DOS OPERANDOS 21
3.4 MODOS DE ENDEREÇAMENTO 23
3.5 FORMATOS DE INSTRUÇÃO 26
4 A ARQUITETURA DOS PROCESSADORES INTEL 80X86 28
4.1 QUADRO EVOLUTIVO DA FAMÍLIA INTEL 80X86 28
4.2 O INTEL 8086/8088 29
4.2.1 A Unidade de Execução 30
4.2.2 A Unidade de Interface de Barramento (BIU) 31
4.2.3 O Conjunto de Instruções do 8086 34
4.3 O INTEL 80186/80188 39
4.4 O INTEL 80286 39
4.5 A ARQUITETURA DO INTEL 80386 41
4.6 A ARQUITETURA DO INTEL 80486 44
4.7 O INTEL PENTIUM 45
5 O SUB-SISTEMA DE MEMÓRIA 47
5.1 A INTERAÇÃO ENTRE PROCESSADOR E MEMÓRIA PRINCIPAL 47
5.1.1 Ciclo de Barramento 49
5.1.2 Estados de Espera 50
5.2 TIPOS DE DISPOSITIVOS DE MEMÓRIA 52
5.3 MEMÓRIAS CACHE 53
5.3.1 A Memória Cache no Intel 80486 56
5.3.2 Memórias Cache Primária e Secundária 59
5.4 MEMÓRIA VIRTUAL 60
5.4.1 O Conceito de Memória Virtual 60
5.4.2 O Mecanismo de Memória Virtual 60
5.4.3 A Memória Secundária 63
5.4.4 Memória Virtual no Intel 80486 65
6 O SUB-SISTEMA DE ENTRADA/SAÍDA 71
6.1 A INTERAÇÃO ENTRE PROCESSADOR E INTERFACES DE E/S 71

6.2 ORGANIZAÇÃO DE UMA INTERFACE DE E/S 72
6.3 TÉCNICAS DE TRANSFERÊNCIA DE DADOS 74
6.3.1 E/S com Polling 74
6.3.2 E/S com Interrupção 76
6.3.3 E/S com Acesso Direto à Memória 78
6.4 PADRÕES DE BARRAMENTOS 80
6.4.1 Barramentos Locais 81
6.4.2 Barramentos de Periféricos 83

7 TÓPICOS ESPECIAIS 84

7.1 A TÉCNICA DE PIPELINING 84
7.2 ARQUITETURAS SUPER-ESCALARES 87
7.3 A ARQUITETURA ALPHA AXP 88
7.4 ARQUITETURA DO PENTIUM 91
7.5 A ARQUITETURA POWERPC 601 94
7.6 ARQUITETURAS RISC 97
7.6.1 Lacuna Semântica 98
7.6.2 Críticas às Arquiteturas Complexas 100
7.6.3 A Filosofia RISC 102
7.6.4 Características das Arquiteturas RISC 103
7.6.5 Histórico das Arquiteturas RISC 104
7.7 SISTEMAS PARALELOS 105
7.7.1 Sistemas SISD 105
7.7.2 Sistemas SIMD 106
7.7.3 Sistemas MISD 106
7.7.4 Sistemas MIMD 106

1 INTRODUÇÃO

1.1 O Conceito de Arquitetura

Este é um curso introdutório sobre arquitetura de computadores. É importante, em primeiro lugar, definir precisamente o significado que aqui será dado ao termo “arquitetura de computador”. A arquitetura de um computador é um modelo da organização e funcionamento de um sistema de processamento. A descrição da arquitetura destaca as funções dos componentes básicos de um computador, a interconexão destes componentes e o modo como os componentes interagem.

Este curso focaliza aqueles tipos de sistemas que já fazem parte do nosso cotidiano profissional, basicamente microcomputadores e estações de trabalho. A Figura 1.1 mostra a organização típica destes dois tipos de sistemas.

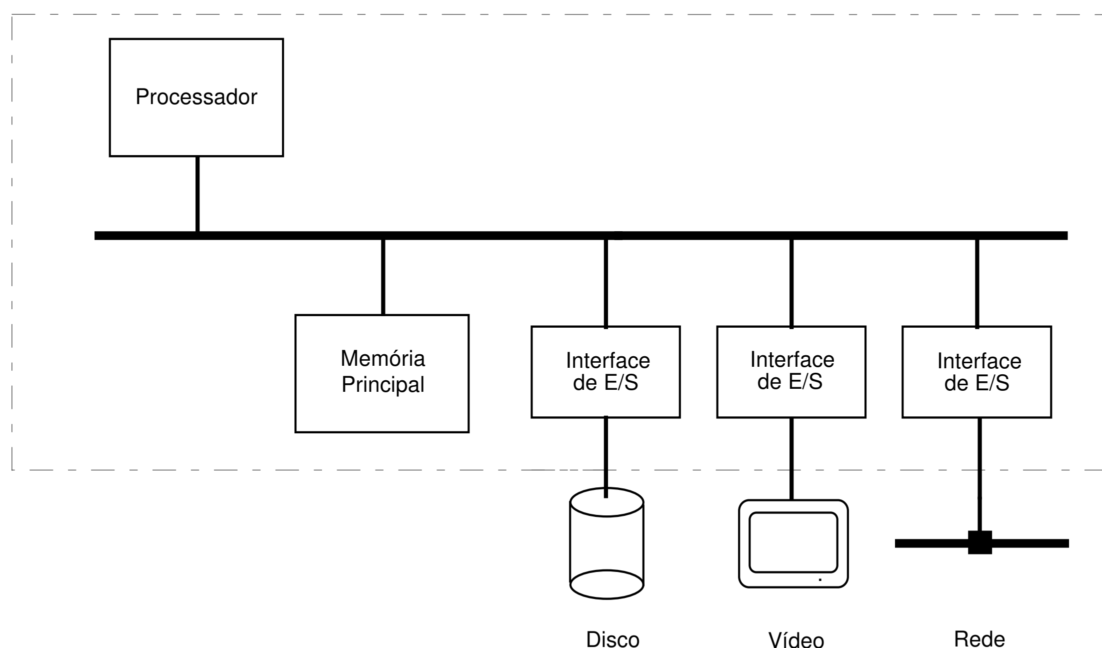


Figura 1.1. Organização típica de um computador.

Esta figura mostra os três componentes básicos de um computador: o processador (também conhecido como unidade central de processamento), a memória principal e as interfaces de entrada e saída. O processador é o componente ativo, controlador de todo o sistema. É o processador que realiza todas as operações sobre os dados, de acordo com o indicado pelas instruções no código do programa. A memória principal armazena as instruções que são executadas pelo processador, e os dados que serão manipulados. As interfaces de entrada e de saída são as portas de comunicação para o mundo externo, às quais estão conectados os dispositivos periféricos tais como vídeo, teclado, discos e impressora.

Estes componentes estão interconectados por meio de um barramento, através do qual o processador realiza o acesso a instruções e dados armazenados na memória principal. É também através deste mesmo barramento que o processador recebe ou envia dados de ou para as interfaces de entrada/saída. Instruções e dados estão armazenados em locações de memória, na memória principal. Para realizar o acesso a uma informação, o processador envia para a memória, através do barramento, o endereço da locação de memória que contém a informação aonde será feito o acesso. A informação é trocada entre o processador e a memória também através do barramento. O acesso à interfaces de entrada/saída é semelhante, sendo cada interface identificada por um endereço único.

A Figura 1.1 e a discussão a ela associada descrevem sucintamente a arquitetura de um computador. Conforme mencionado acima, esta descrição indica quais são os componentes básicos da arquitetura (processador, memória principal e interfaces) e suas funções, como estão interconectados (através de um barramento) e como interagem (troca de endereços, instruções e dados através do barramento). Nos capítulos que se seguem, esta arquitetura será descrita em maiores detalhes.

1.2 Níveis de Arquitetura

Na realidade, o conceito de arquitetura pode ser aplicado a diferentes sistemas de hardware e software, levando a diversos níveis de arquitetura conforme mostra a Figura 1.2.

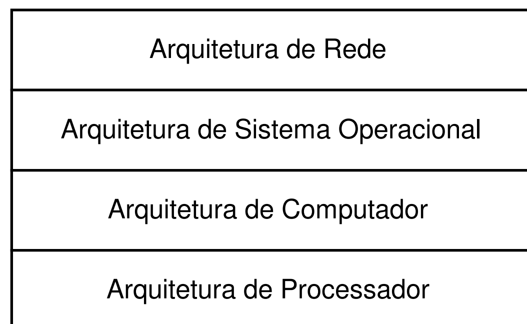


Figura 1.2. Níveis de arquitetura.

O nível de arquitetura de processador descreve a organização e o funcionamento de um dos componentes de um sistema de processamento. Neste nível são descritos os elementos básicos de um processador, o modo como instruções são executadas pelo processador e o seu conjunto de instruções. O próximo nível é o de arquitetura de computador que, como já visto, descreve o sistema de processamento como um todo.

O nível acima é o de arquitetura de sistema operacional. Em termos simples, o sistema operacional serve de interface entre o hardware do computador e o usuário, tornando os detalhes de operação do computador transparentes ao usuário. Neste nível, são descritas a organização e as funções de um sistema operacional e especificados os serviços por ele oferecidos. Finalmente, o nível de

rede de computadores aborda um sistema formado por computadores interligados por um meio de comunicação. No nível de arquitetura de redes é descrita a conexão física entre os computadores, bem como os protocolos de comunicação usados na troca de informações entre os computadores.

É muito importante perceber que estes níveis de arquitetura não estão isolados. O perfeito entendimento de um certo nível exige uma compreensão de vários aspectos de um ou mais níveis inferiores. Por exemplo, para entender o gerenciamento de memória virtual — um assunto que é tratado dentro do nível de arquitetura de sistema operacional — é necessário conhecer o suporte para memória virtual oferecido pelo processador, o que é abordado a nível de arquitetura de processador. Atualmente, está cada vez mais claro que o pleno domínio de algumas áreas da computação exige do indivíduo uma visão de conjunto destes quatro níveis de arquitetura.

2 CONCEITOS BÁSICOS DE ARQUITETURA DE PROCESSADOR

Este capítulo descreve a arquitetura básica de um processador. Podemos considerar que um processador é organizado em duas unidades, a seção de processamento e a seção de controle. Este capítulo descreve os principais componentes em cada uma destas unidades. Dentro deste contexto, o capítulo também descreve como um processador executa as instruções de um programa. No capítulo seguinte, aborda-se um tema central na arquitetura de um processador, qual seja, o seu conjunto de instruções.

2.1 A Seção de Processamento

A seção de processamento é formada basicamente pela unidade lógica e aritmética (ALU) e por diversos registradores. Estes componentes normalmente estão organizados conforme mostra a Figura 2.1.

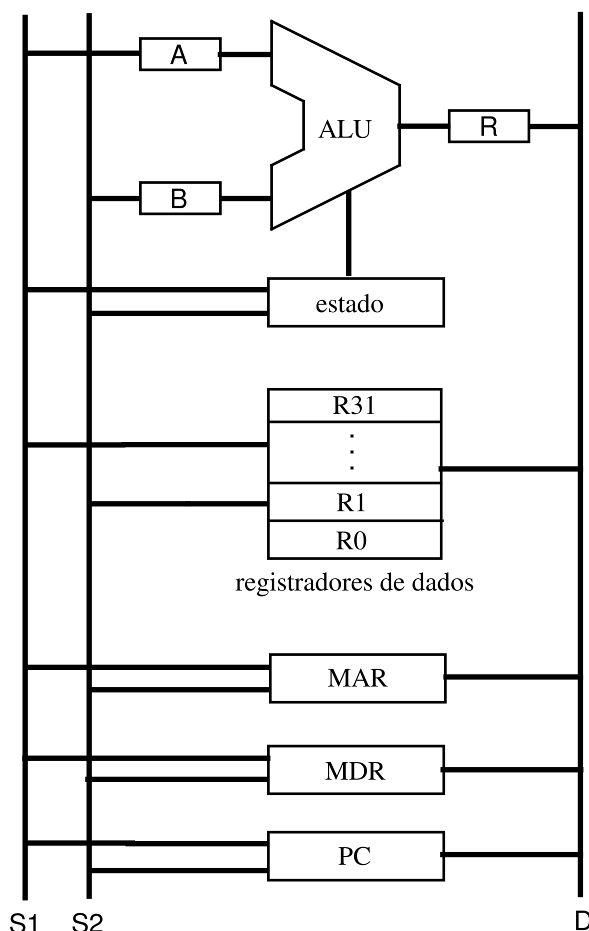


Figura 2.1. Componentes da seção de processamento.

A ALU realiza as operações aritméticas, tais como adição e subtração, e operações lógicas, tais como and, or, not. Podemos dizer então que a ALU é o componente da arquitetura que, de fato, processa os dados. Os registradores são utilizados para armazenar informações internamente no processador. Um

registrador pode ser utilizado tanto para acesso de leitura quanto para acesso de escrita: uma informação é armazenada no registrador em uma operação de escrita, enquanto a informação contida no registrador é recuperada em uma operação de leitura.

A Figura 2.1 mostra aqueles registradores normalmente encontrados na seção de processamento. Os diversos registradores possuem um uso bem definido dentro da arquitetura, e de uma maneira geral podem ser classificados em três tipos: registradores de uso geral, registradores de uso específico e registradores auxiliares. Registradores de uso geral normalmente são usados para armazenar dados que serão processados pela ALU, bem como resultados produzidos pela ALU. Na seção de processamento mostrada na Figura 2.1, existem 32 registradores de uso geral, denominados R0,...,R31. Coletivamente, estes registradores são chamados de conjunto de registradores de dados (data register file).

O registrador de estado (status register) associado à ALU é um registrador de uso específico, e contém informações sobre o resultado produzido pela ALU. Este registrador possui bits sinalizadores que são ativados ou desativados[1] de acordo com o tipo de resultado produzido pela ALU. Por exemplo, o registrador de estado pode ter um bit denominado Z, o qual é ativado quando o resultado for nulo e desativado quando o resultado for não-nulo. Também é comum encontrar no registrador de estado um bit chamado N que é ativado se o resultado for negativo, sendo desativado se o resultado for positivo.

Um outro exemplo de registrador de uso específico é o contador de programa (program counter). O contador de programa contém o endereço da locação de memória onde se encontra a próxima instrução a ser executada pelo processador.

Os registradores auxiliares normalmente são usados para armazenamento temporário. Este é o caso dos registradores A e B, que armazenam os operandos de entrada da ALU, enquanto estes estão sendo processados. Antes de cada operação da ALU, os operandos são transferidos dos registradores de dados ou da memória principal para estes registradores temporários. O resultado produzido pela ALU é temporariamente armazenado no registrador R até ser transferido para o seu destino, que pode ser um registrador de dados ou a memória principal. A Figura 2.1 mostra dois outros registradores temporários, denominados MAR (memory address register) e MDR (memory data register). O MAR armazena o endereço da locação de memória onde será feito o acesso, ao passo que o MDR armazena temporariamente a informação transferida de ou para a locação de memória endereçada por MAR. Em geral, registradores auxiliares não são visíveis, no sentido que um programador de baixo nível não dispõe de instruções para acessar diretamente tais registradores.

A interligação entre a ALU e os registradores é feita através de três vias, chamadas barramentos internos. Os barramentos internos S1 e S2 permitem a

[1] Considera-se aqui que um *bit* é ativado quando ele recebe o valor lógico 1, sendo desativado ao receber o valor lógico 0.

transferência de dados dos registradores para a ALU. O barramento interno D permite a transferência do resultado produzido pela ALU, temporariamente armazenado no registrador R, para outro registrador.

Uma arquitetura de processador é uma arquitetura de n bits quando todas as operações da ALU podem ser realizadas sobre operandos de até n bits. Normalmente, em uma arquitetura de n bits os registradores de dados e os barramentos internos também são de n bits, de forma a permitir que os dados sejam armazenados e transferidos de forma eficiente.

2.2 A Execução de Instruções

Tendo examinado os componentes e a organização da seção de processamento, podemos agora analisar como as instruções são executadas. A execução de uma instrução envolve a realização de uma seqüência de passos, que podemos chamar de passos de execução. Em geral, a execução de uma instrução envolve quatro passos, como mostra a Figura 2.2.

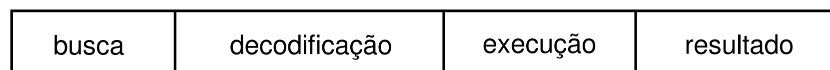


Figura 2.2. Passos na execução de uma instrução.

No primeiro passo, denominado busca, o processador realiza o acesso ao código binário da instrução, armazenado na memória principal. A etapa seguinte é a decodificação da instrução, na qual as informações contidas no código da instrução são interpretadas. Em algumas arquiteturas, neste passo também são acessados os dados usados pela instrução. Após a decodificação, a execução da instrução entra no terceiro passo, denominado execução, no qual a operação indicada pela instrução (por exemplo, uma operação na ALU) é efetuada. Finalmente no quarto passo, chamado resultado, é armazenado em um registrador ou na memória o resultado produzido pela instrução.

Cada passo de execução envolve a realização de várias operações básicas. As operações básicas acontecem dentro da seção de processamento, sob a coordenação da seção de controle. Existem quatro principais tipos de operações básicas:

- transferência de dados entre os registradores e a ALU;
- transferência de dados entre os registradores;
- transferência de dados entre os registradores e a memória;
- operações aritméticas e lógicas realizadas pela ALU.

Esta sub-divisão dos passos de execução em operações básicas pode ser visualizada na Figura 2.3.

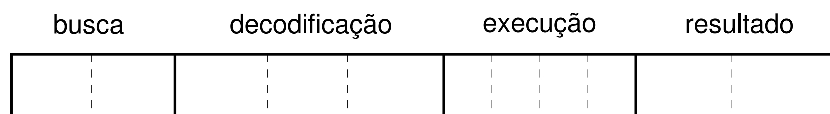


Figura 2.3. Divisão dos passos de execução em operações básicas.

A sub-divisão dos passos sugerida pela Figura 2.3 é apenas um exemplo ilustrativo. Os passos de execução não possuem necessariamente o mesmo número de operações básicas em todas as instruções. O que diferencia cada instrução é justamente o número e o tipo de operações básicas executadas em cada passo.

Para tornar mais claro o mecanismo de execução de instruções, considere uma arquitetura com uma seção de processamento idêntica à da Figura 2.1. Considere também que a arquitetura oferece quatro tipos de instruções: instruções aritméticas e lógicas, instruções de desvio incondicional e condicional, e instruções de acesso à memória. Exemplos destes tipos de instruções aparecem no quadro da Figura 2.4.

Tipo	Exemplo	Descrição
aritmética/lógica	`ADD Rs1,Rs2,Rd`	O conteúdo dos registradores Rs1 e Rs2devem ser somados e o resultado armazenado em Rd.
desvio incondicional	`JMP dst`	A próxima instrução a ser executada deve ser a que se encontra na locação de memória com endereço dst.
desvio condicional	`JZ dst`	A próxima instrução a ser executada deve ser a que se encontra no endereço dst, caso o bit Z no registrador de estado esteja ativado.
acesso à memória	`LOAD end,R1`	O dado armazenado na locação de memória com endereço end deve ser transferido para o registrador R1.

Figura 2.4. Exemplos de instruções.

A Figura 2.5 apresenta um exemplo hipotético relacionando, para cada tipo de instrução, as operações básicas que acontecem dentro de cada passo de execução. Nesta figura, Ry $\leftarrow$ Rx representa a transferência do conteúdo do registrador Rx para o registrador Ry. M[R] denota o conteúdo da locação de memória cujo endereço está no registrador R. Finalmente, IR (instruction register) representa um registrador especial que recebe o código da instrução a ser executada, enquanto PC é o contador de programa.

	Aritméticas e Lógicas	Desvios Incondicionais	Desvios Condicionais	Acessos à Memória
Busca	MAR ← PC MDR ← M[MAR] IR ← MDR PC++	MAR ← PC MDR ← M[MAR] IR ← MDR PC++	MAR ← PC MDR ← M[MAR] IR ← MDR PC++	MAR ← PC MDR ← M[MAR] IR ← MDR PC++
Decodificação	decod A ← Rs1 B ← Rs2	decod	decod	decod
Execução	R ← A op B	PC ← destino	cond se (cond) PC ← destino	MAR ← end MDR ← Rs (E) M[MAR] ← MDR (E) MDR ← M[MAR] (L)
Resultado	Rd ← R			Rd ← MDR (L)

Figura 2.5. Operações básicas na execução de instruções.

O passo de busca é idêntico para todos os tipos de instruções e envolve quatro operações básicas: (1) o conteúdo do contador de programa é transferido para o registrador de endereço de memória MAR; (2) é realizado oacesso à memória, usando o endereço em MAR; (3) o código de instrução recebido da memória, temporariamente armazenado em MDR, é transferido para o registrador de instrução IR (este registrador faz parte da seção de controle, como será visto mais à frente); (4) contador de programa é incrementado, passando a indicar a próxima instrução onde será feito o acesso e executada.

Concluído o passo de busca, inicia-se o passo de decodificação da instrução armazenada em IR. A interpretação do código da instrução é indicado por "decod" na Figura 2.5. Como mencionado, em algumas arquiteturas este passo também inclui o acesso aos operandos da instrução. Na Figura 2.5, isto é indicado pela transferência do conteúdo dos registradores de dados Rs1 e Rs2 para os registradores temporários A e B, respectivamente, no caso de instruções aritméticas e lógicas.

O próximo passo é o de operação. As operações básicas neste passo dependem inteiramente do tipo de instrução que está sendo executada. No caso das instruções aritméticas e lógicas, este passo corresponde à execução pela ALU da operação indicada na instrução, utilizando como operandos o conteúdo dos registradores A e B, com armazenamento do resultado no registrador R.

Em instruções de desvio incondicional, apenas uma operação básica é realizada: o endereço destino é carregado no contador de programa. Como o contador de programa indica a próxima instrução a ser executada, isto resulta em um desvio para a instrução armazenada no endereço destino. Em desvios condicionais, o contador de programa é modificado somente se a condição de desvio for verdadeira. A avaliação da condição de desvio é indicada por "cond", na Figura 2.5. O endereço destino é carregado no contador de programa apenas se a condição de desvio testada for verdadeira.

Em instruções de acesso à memória, o passo de execução inicia-se com a transferência do endereço da locação onde será feito o acesso para o registrador MAR. As demais operações básicas dependem se o acesso é de escrita (indicadas por E) ou de leitura (indicadas por L). No caso de uma escrita, são executadas duas operações básicas: a transferência do dado a ser escrito para o registrador MDR e a escrita na memória propriamente dita. No caso de uma leitura, aé realizado o acesso à locação de memória e o dado obtido é armazenado no registrador MDR.

O último passo é o de armazenamento do resultado. Em instruções aritméticas e lógicas, o resultado no registrador R é transferido para o registrador destino indicado na instrução. Em instruções de leitura à memória, o dado que se encontra no registrador MDR é transferido para o registrador destino.

É importante salientar que o quadro na Figura 2.5 é extremamente simplificado. Por exemplo, na prática um acesso à memória não é feito por uma única operação básica como indicado, mas normalmente requer várias operações básicas. No entanto, este quadro reflete corretamente como a execução de uma instrução é logicamente organizada.

2.3 A Seção de Controle

Como mencionado anteriormente, as operações básicas que ocorrem dentro da seção de processamento são todas comandadas pela seção de controle. Ao efetuar a busca da instrução, a unidade de controle interpreta a instrução de modo a identificar quais as operações básicas que devem ser realizadas, e ativa sinais de controle que fazem uma operação básica de fato acontecer.

A Figura 2.6 apresenta um diagrama em blocos da seção de controle e, de forma bastante simplificada e ilustrativa, a sua interligação com a seção de processamento. Como mostra a figura, a seção de controle é formada basicamente pela unidade de controle e pelo registrador de instrução, ou IR (Instruction Register). A interpretação do código da instrução e a ativação dos sinais de controle são realizados pela unidade de controle. Os sinais de controle que são

ativados, bem como a seqüência com que são ativados, depende de cada instrução em particular.

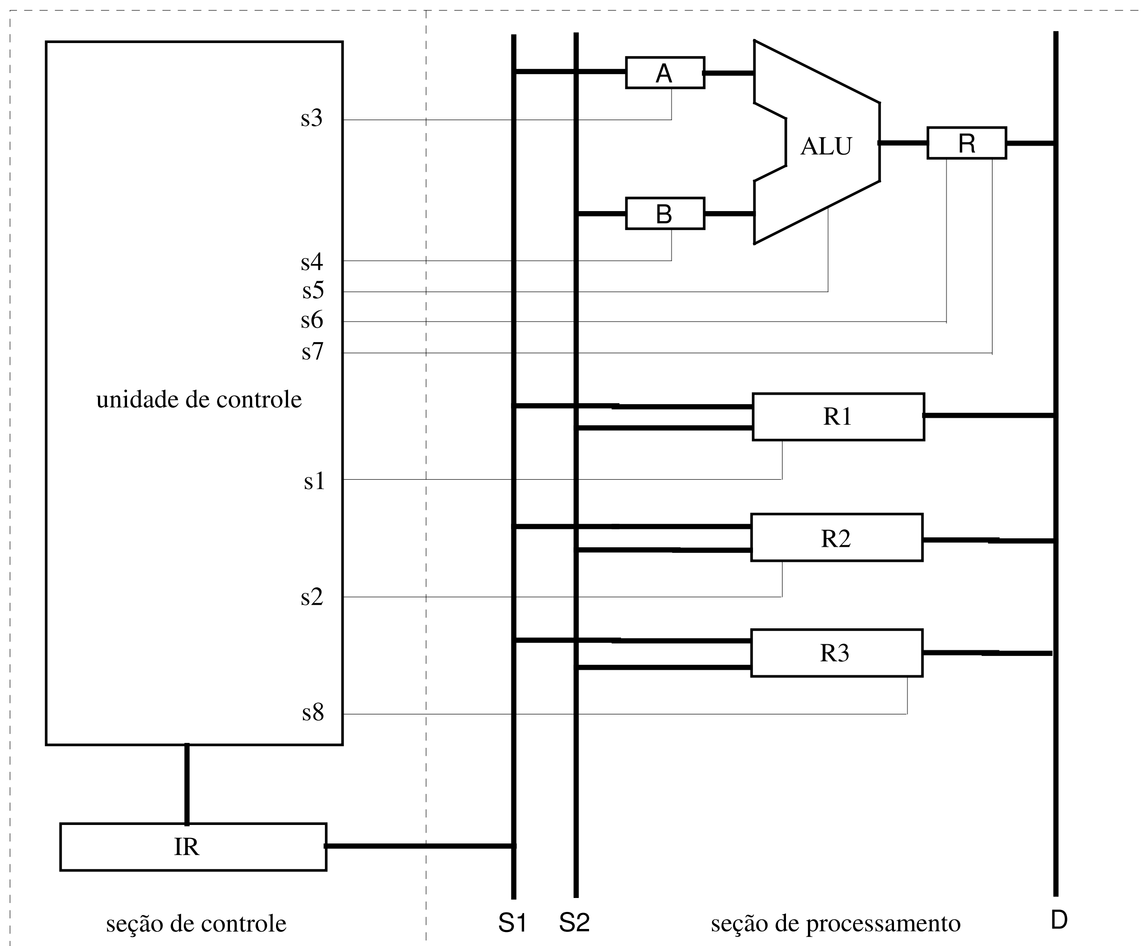


Figura 2.6. A seção de controle e a parte de processamento.

A título de exemplo, vamos verificar como a execução da instrução `ADD R1,R2,R3` é direcionada pela unidade de controle. Para tanto, a seção de processamento na figura acima foi representada com apenas os três registradores de dados envolvidos na execução desta instrução. A execução desta instrução requer as seguintes operações básicas:

(1) transferência do conteúdo do registrador de dados R1 para o registrador temporário A;

(2) transferência do conteúdo do registrador de dados R2 para o registrador temporário B;

(3) adição dos dados armazenados nos registradores A e B e armazenamento do resultado no registrador R;

(4) transferência do conteúdo do registrador R para o registrador R3.

A seqüência de ativação do sinais de controle, com as operações básicas correspondentes, é apresentada na Figura 2.8.

operação básica	sinal de controle	descrição da operação básica
(1) (2)	s1, s2 s3,s4	coloca o conteúdo de R1, R2 para os barramentos S1,S2, respectivamente. armazena a informação presente nos barramentos S1, S2 em A,B, respectivamente.
(3)	s5 s6	seleciona a operação de soma na ALU. armazena o resultado produzido pela ALU em R.
(4)	s7 s8	coloca o conteúdo de R para o barramento D. armazena a informação presente no barramento D em R3.

Figura 2.8. Ativação dos sinais de controle e operações básicas correspondentes.

Este é um exemplo típico de como a unidade de controle coordena a execução das operações básicas na seção de processamento. Para cada registrador existem sinais que controlam a leitura e a escrita do registrador. Outros sinais indicam à ALU a operação aritmética ou lógica que deve realizada. Para qualquer outro componente na seção de processamento existem os sinais de controle necessários. Para executar uma operação básica, a unidade de controle simplesmente ativa os sinais apropriados na seqüência correta.

2.4 O Sinal de Clock

Pode-se observar pelo exemplo acima que a ordem na qual os sinais de controle são ativados é crítica. Alguns sinais devem obrigatoriamente preceder outros (s3, por exemplo, não pode ser ativado antes de s1), enquanto que outros sinais podem ser ativados simultaneamente (s1 e s2, por exemplo). Mais ainda, para garantir um tempo suficiente para a transmissão da informação através dos

barramentos internos, em alguns casos deve ser observado um intervalo de tempo mínimo entre a ativação de dois sinais.

Para atender as relações de tempo requeridas na ativação dos sinais de controle, a unidade de controle opera em sincronismo com um sinal de clock. Como mostra a Figura 2.9, uma nova operação básica é executada no momento em que inicia-se um novo ciclo de clock. Em muitos casos, várias operações básicas podem ser comandadas simultaneamente, dentro de um mesmo ciclo de clock.

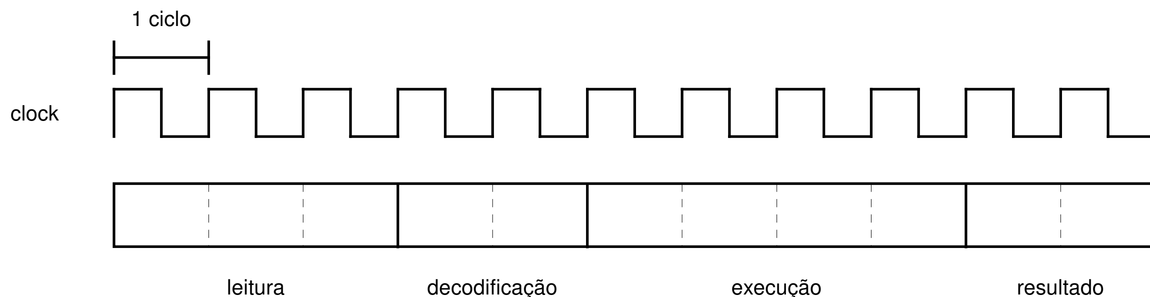


Figura 2.9. A sincronização da execução de uma instrução com o sinal de clock.

A execução de uma instrução consome um certo número de ciclos de clock. O número de ciclos de clock por instrução não é o mesmo para todas as instruções, já que cada instrução pode envolver um número diferente de operações básicas em cada passo de execução.

O tamanho do ciclo de clock é um dos fatores que determinam diretamente o desempenho de um processador. Quanto menor o tamanho do ciclo de clock, menor será o tempo de execução das instruções, e assim maior será o número de instruções executadas por unidade de tempo. Ao longo das décadas de 70 e 80, procurava-se diminuir o tamanho do ciclo de clock com o desenvolvimento de novas tecnologias que permitissem velocidades de operação cada vez maiores. No entanto, as tecnologias de integração foram se aproximando dos limites impostos pela própria física, tornando esta evolução mais lenta e elevando os custos.

Por este motivo, a redução do ciclo de clock passou a ser considerada sob o ponto de vista arquitetural. Atualmente, procura-se diminuir o ciclo de clock não somente através de novas tecnologias, mas também através de simplificações na arquitetura, de modo que a arquitetura possa ser implementada através de circuitos mais simples e inerentemente mais rápidos. Esta relação entre a complexidade funcional da arquitetura e o tamanho do ciclo de clock será discutida em maiores detalhes na discussão sobre arquiteturas RISC, no Capítulo 7.

2.5 Implementação da Unidade de Controle

Existem basicamente duas maneiras de implementar uma unidade de controle. A primeira delas é usando lógica aleatória (o termo original é hardwired control). A outra forma é usando microprogramação. A Figura 2.10 mostra a estrutura típica de uma unidade de controle implementada com lógica aleatória.

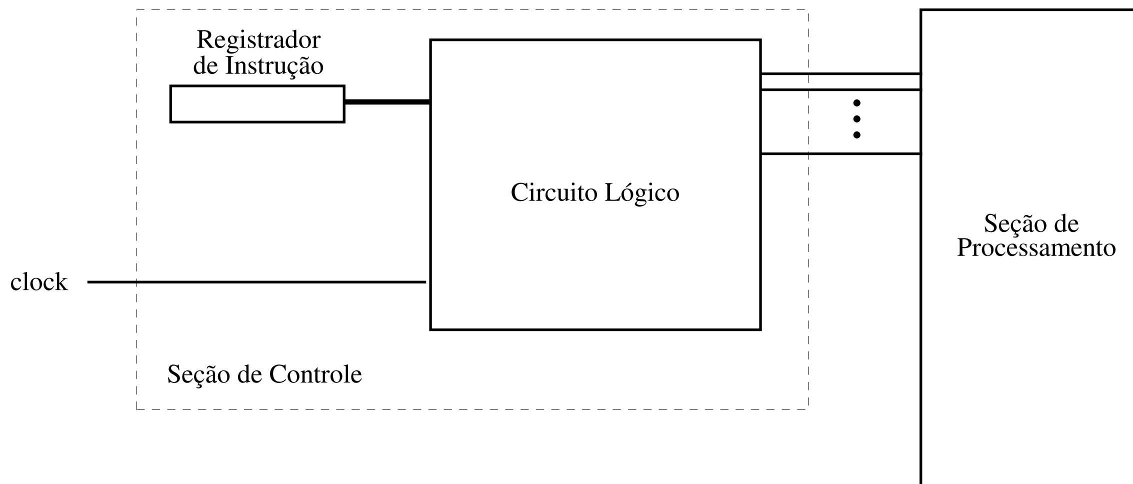


Figura 2.10. Organização de uma unidade de controle implementada com lógica aleatória.

Na implementação por lógica aleatória, a unidade de controle é formada por um único circuito lógico, cuja entrada é o código de instrução armazenado em IR e cujas saídas são os próprios sinais de controle que comandam as operações básicas na seção de processamento. De acordo com o código da instrução, a cada ciclo de clock este circuito ativa os sinais de controle que comandam as operações básicas que devem ser realizadas naquele ciclo.

A desvantagem desta forma de implementação é que a complexidade do circuito de controle, em termos do número de dispositivos lógicos, aumenta rapidamente com o número de instruções oferecidas pela arquitetura e com o número de operações básicas que devem ser realizadas na execução de uma instrução. Em arquiteturas com instruções funcionalmente complexas, o projeto de uma unidade de controle com lógica aleatória torna-se muito difícil e propenso a erros.

A técnica de microprogramação corrige esta desvantagem da implementação com lógica aleatória. Nela, cada instrução oferecida pela arquitetura, que passa a ser chamada de macroinstrução, é na realidade executada por uma seqüência de instruções primitivas, extremamente simples, chamadas microinstruções. Os próprios bits das microinstruções são usados para ativar e desativar os sinais de controle que comandam as operações básicas. A seqüência de microinstruções que executa uma macroinstrução formam uma microrotina. Usando de uma interpretação mais simples, podemos considerar que a execução de uma macroinstrução consiste na chamada de uma microrotina, feita pela unidade de controle. As microinstruções da microrotina executam as operações básicas associadas à macroinstrução.

A Figura 2.11 mostra a estrutura de uma unidade de controle implementada com a técnica de microprogramação.

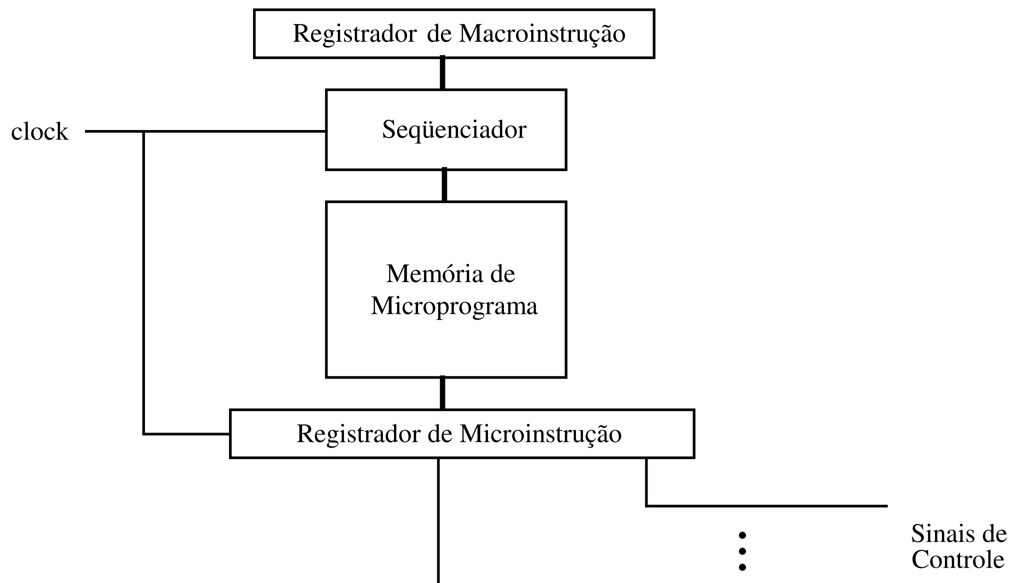


Figura 2.11. Organização de uma unidade de controle microprogramada.

As microrotinas encontram-se armazenadas na memória de microprograma. Quando o código da macroinstrução é armazenado no registrador de (macro)instrução, o seqüenciador interpreta este código e determina o endereço de entrada da microrotina que executa aquela macroinstrução. O seqüenciador fornece, a cada ciclo de clock, o endereço da próxima microinstrução a ser executada. Após o acesso à microinstrução, ela é armazenada no registrador de microinstrução. Alguns bits da microinstrução são usados diretamente para comandar os sinais de controle para a seção de processamento. Outros bits são utilizados pelo seqüenciador para determinar a próxima microinstrução a ser executada.

A execução de uma microinstrução envolve o acesso da microinstrução na memória de microprograma, o armazenamento no registrador de microinstrução e a realização das operações básicas comandadas pelos bits da microinstrução. Uma nova microinstrução é executada a cada ciclo de clock. Quando a execução de uma microrotina é concluída, uma nova macroinstrução é acessada na memória principal e armazenada no registrador de instrução, e iniciada a execução de uma nova microrotina.

A vantagem da microprogramação está no fato que a implementação das instruções reduz-se basicamente à escrita das microrotinas que serão gravadas na memória de microprograma. Esta vantagem se torna especialmente significativa quando a arquitetura oferece um grande número de instruções e estas instruções são complexas, ou seja, a sua execução envolve um grande número de operações básicas. Neste caso, o projeto de um complicado circuito lógico, como aconteceria na implementação com lógica aleatória, é substituído pela escrita das microrotinas, uma tarefa comparativamente bem mais simples.

3 O CONJUNTO DE INSTRUÇÕES DO PROCESSADOR

O conjunto de instruções é um dos pontos centrais na arquitetura de um processador. Vários aspectos na definição e implementação da arquitetura são influenciados pelas características do conjunto de instruções. Por exemplo, as operações realizadas pela unidade lógica e aritmética, o número e função dos registradores e a estrutura de interconexão dos componentes da seção de processamento são influenciadas pelo conjunto de instruções. Além disso, como fica claro a partir do capítulo anterior, as operações básicas que acontecem dentro da seção de processamento dependem das instruções que devem ser executadas. O conjunto de instruções afeta não somente o projeto da seção de processamento: a estrutura e a complexidade da unidade de controle é determinada diretamente pelas características do conjunto de instruções.

Este capítulo discute as principais aspectos de um conjunto de instruções, como tipos de operações, operandos, e modos de endereçamento. Um exemplo real de conjunto de instruções será apresentado no próximo capítulo, na discussão da família Intel 80x86.

3.1 Conjunto de Instruções no Contexto de Software

A Figura 3.1 situa o conjunto de instruções do processador dentro dos diversos níveis de software existentes em um sistema de computação.

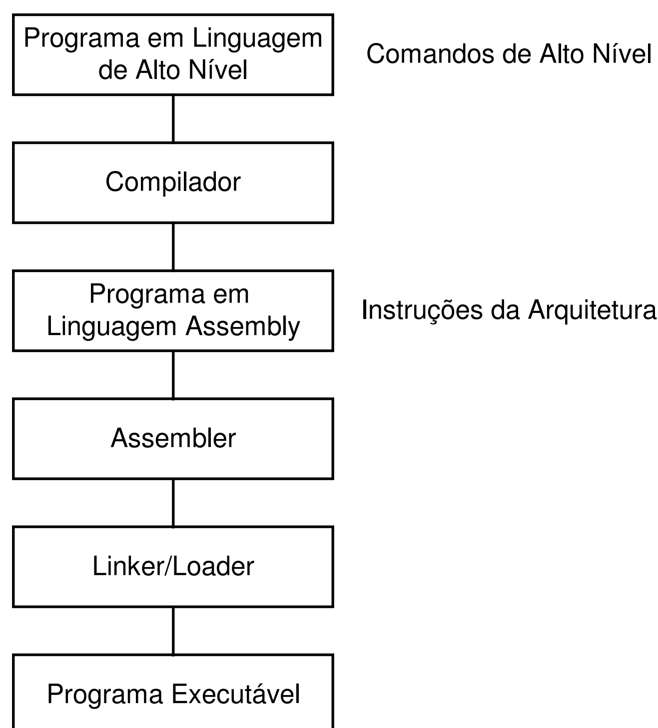


Figura 3.1. O conjunto de instruções dentro do contexto de software.

Em geral, os programas são desenvolvidos em uma linguagem de alto nível como FORTRAN, Pascal ou C. O compilador traduz o programa de alto nível em uma seqüência de instruções de processador. O resultado desta tradução é o programa em linguagem de montagem (assembly language). A linguagem de montagem é uma forma de representar textualmente as instruções oferecidas pela arquitetura. Cada arquitetura possui uma particular linguagem de montagem. No programa em linguagem de montagem, as instruções são representadas através de mnemônicos, que associam o nome da instrução à sua função, como por exemplo, `ADD` ou `SUB`.

O programa em linguagem de montagem é convertido para um programa em código objeto pelo montador (assembler). O montador traduz diretamente uma instrução da forma textual para a forma de código binário. É sob a forma binária que a instrução é carregada na memória e interpretada pelo processador.

Programas complexos são normalmente estruturados em módulos. Cada módulo é separadamente compilado e submetido ao montador, gerando diversos módulos em código objeto. Estes módulos são reunidos pelo ligador (linker), resultando finalmente no programa executável que é carregado na memória.

O conjunto de instruções de uma arquitetura se distingue através de diversas características. As principais características de um conjunto de instruções são: tipos de instruções e operandos, número e localização dos operandos em instruções aritméticas e lógicas, modos de endereçamento para acesso a dados na memória, e o formato dos códigos de instrução. Estes aspectos são analisados a seguir.

3.2 Tipos de Instruções e de Operandos

As instruções oferecidas por uma arquitetura podem ser classificadas em categorias, de acordo com o tipo de operação que realizam. Em geral, uma arquitetura fornece pelo menos três categorias de instruções básicas:

- instruções aritméticas e lógicas: são as instruções que realizam operações aritméticas sobre números inteiros, tais como adição e subtração, e operações lógicas bit-a-bit, tais como AND e OR;
- instruções de movimentação de dados: instruções que transferem dados entre os registradores ou entre os registradores e a memória principal;
- instruções de transferência de controle: instruções de desvio e de chamada de rotina, que transferem a execução para uma determinada instrução dentro do código do programa.

Várias arquiteturas oferecem outras categorias de instruções, voltadas para operações especializadas. Dentre estas, podemos citar:

- instruções de ponto flutuante: instruções que realizam operações aritméticas sobre números com ponto flutuante;

- instruções decimais: instruções que realizam operações aritméticas sobre números decimais codificados em binário (BCD);

- instruções de manipulação de bits: instruções para testar o valor de um bit, ou para atribuir um valor a um bit;

- instruções de manipulação de strings: instruções que realizam operações sobre cadeias de caracteres (strings), tais como movimentação, comparação ou ainda procura de um caracter dentro de um string.

Existem muitas diferenças entre as arquiteturas quanto às categorias de instruções oferecidas. Arquiteturas de uso geral oferecem a maioria das categorias acima relacionadas. Arquiteturas destinadas para uma aplicação específica podem oferecer outros tipos de instruções, especializadas para aquela aplicação. Um exemplo seria uma arquitetura voltada para processamento gráfico, que ofereceria instruções para realizar operações sobre pixels.

Os tipos de operandos que podem ser diretamente manipulados por uma arquitetura depende, é claro, dos tipos de instruções oferecidas. A Figura 3.2 mostra como os principais tipos de dados são normalmente representados em uma arquitetura de uso geral.

A Figura 3.2(a) mostra a representação de inteiros, neste exemplo particular, inteiros com 32 bits. Números inteiros podem ser representados com ou sem sinal. Em um número inteiro com sinal, o bit mais significativo é reservado para indicar o estado do sinal (positivo, negativo). Números inteiros sem sinal assumem apenas valores positivos. Algumas arquiteturas oferecem instruções específicas para aritmética com ou sem sinal. Estas instruções diferem no modo como são alterados os bits do registrador de estado associado à ALU. Algumas linguagens de programação tornam visível para o programador esta distinção entre inteiros com ou sem sinal. Na linguagem C por exemplo, uma variável declarada do tipo `int` é representada por um inteiro com sinal. Ao contrário, variáveis do tipo `unsigned int` são representadas por inteiros sem sinal, sendo normalmente usadas para indexar elementos de vetores.

A Figura 3.2(b) mostra a representação de números com ponto flutuante, com precisão simples e dupla. A diferença entre precisões está no número de bits usados para representar a mantissa e o expoente. Atualmente, a maioria da arquiteturas que operam números com ponto flutuante obedecem a um padrão, denominado IEEE 754, que define a representação e um conjunto de operações aritméticas e lógicas para números com ponto flutuante.

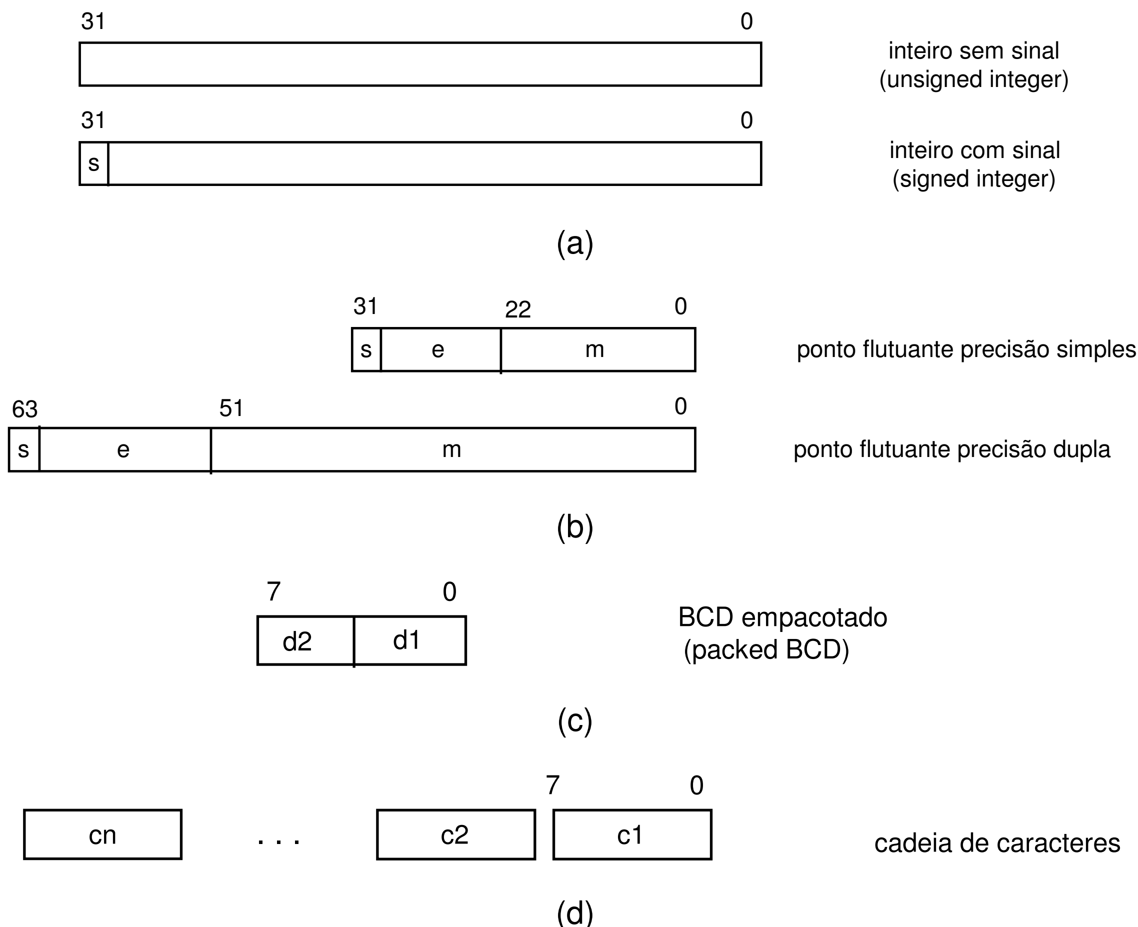


Figura 3.2. Representação dos tipos de operandos mais comuns.

A Figura 3.2(c) mostra a representação de números BCD empacotados (packed BCD). Nesta representação, dois dígitos decimais codificados em binário são representados dentro de um byte, cada dígito sendo codificado em quatro bits do byte. Finalmente, a Figura 3.2(d) mostra a representação de cadeias de caracteres, onde cada byte dentro de uma seqüência de bytes codifica um caracter segundo um certo padrão (p. ex. o padrão ASCII).

3.3 Número e Localização dos Operandos

Uma outra característica de um conjunto de instruções é o número de operandos explicitamente indicados em uma instrução aritmética ou lógica. Em algumas arquiteturas, estas instruções referenciam explicitamente três operandos, dois operandos-fonte e um operando-destino, como por exemplo em:

```
ADD     R1, R2, R3
```

onde R1 e R2 são os operandos-fonte e R3 é o operando-destino. Em outras arquiteturas, instruções aritméticas/lógicas especificam apenas dois operandos; neste caso, um dos operandos-fonte é também o operando-destino. Por exemplo, na instrução `ADD     R1, R2`

R2 contém um dos operandos-fonte e também é usado como operando-destino.

Quanto à localização dos operandos especificados por uma instrução aritmética/lógica, podemos encontrar arquiteturas onde podem ser realizados acessos aos operandos diretamente a partir da memória principal. Por exemplo, nestas arquiteturas podemos ter instruções tais como:

```
ADD    M1,R1,R2

ADD    M1,M2,R1

ADD    M1,M2,M3
```

onde `M1` e `M2` e `M3` são endereços de locações de memória. Em um outro extremo, existem arquiteturas onde todos os operandos encontram-se apenas em registradores. As instruções aritméticas/lógicas são todas do tipo:

```
ADD    R1,R2,R3

ADD    R1,R2
```

A partir do número de operandos explicitamente referenciados e da localização destes operandos, podemos classificar as arquiteturas nos seguintes tipos:

- arquiteturas memória-memória: as instruções aritméticas/lógicas usam três operandos e todos os operandos podem estar na memória;
- arquiteturas registrador-memória: as instruções aritméticas/lógicas usam dois operandos, sendo que apenas um deles pode residir na memória.
- arquiteturas registrador-registrador: as instruções aritméticas/lógicas usam três operandos, todos em registradores. Neste caso, apenas duas instruções acessam diretamente a memória: uma instrução `LOAD`, que carrega em um registrador um dado armazenado na memória e uma instrução `STORE`, que armazena na memória o conteúdo de um registrador.

Arquiteturas memória-memória e registrador-memória apresentam como vantagem um menor número de instruções no código do programa, já que não é necessário carregar previamente em registradores os operandos-fonte de uma instrução aritmética/lógica, como acontece em uma arquitetura registrador-registrador. Por outro lado, a existência de instruções aritméticas/lógicas mais poderosas torna mais complexa a implementação da arquitetura. As arquiteturas Intel 80x86 e Motorola MC680x0 são do tipo registrador-memória. Dentre as arquiteturas memória-memória podemos citar o DEC VAX 11.

3.4 Modos de Endereçamento

Os operandos de uma instrução podem encontrar-se em registradores, na memória principal ou ainda embutidos na própria instrução. O modo de endereçamento refere-se à maneira como uma instrução especifica a localização dos seus operandos. Existem três modos de endereçamento básicos:

- modo registrador: neste modo, a instrução indica o número de um registrador de dados onde se encontra um operando (fonte ou destino).
- modo imediato: neste modo, a instrução referencia um operando que se encontra dentro do próprio código da instrução;
- modo implícito: neste modo, a localização do operando não está explicitamente indicada na instrução. Por exemplo, nas chamadas arquiteturas acumulador, um dos operandos-fonte e o operando-destino nas instruções aritméticas/lógicas encontra-se sempre em um registrador especial, o acumulador. Assim, não é necessário que este registrador seja explicitamente referenciado pela instrução.

A Figura 3.3 mostra exemplos de instruções que usam os modos de endereçamento implícito, registrador e imediato.

Modo	Exemplo	Significado
Implícito	`ADD R1`	Ac ← Ac + R1
Registrador	`ADD R1,R2`	R2 ← R1 + R2
Imediato	`ADD R1,#4`	R1 ← R1 + 4

Figura 3.3. Exemplos de uso dos modos de endereçamento implícito, registrador e imediato.

Os modos de endereçamento acima referenciam apenas operandos que se encontram em registradores ou na instrução. Existem ainda os modos de endereçamento usados para referenciar dados armazenados na memória principal. Entre as diferentes arquiteturas, existe uma enorme variedade de modos de endereçamento referentes à memória principal, e que formam, na realidade, uma classe de modos de endereçamento à parte.

Um modo de endereçamento referente à memória indica como deve ser obtido o endereço da locação de memória onde se encontra o dado que será acessado. Este endereço é chamado endereço efetivo. Apesar da variedade acima mencionada, é possível identificar alguns modos de endereçamento referentes à memória que são oferecidos pela maioria das arquiteturas. Estes modos de endereçamento mais comuns estão relacionados na Figura 3.4.

Modo	Exemplo	Significado	Uso

Direto	`ADD (100),R1`	R1 ← M[100] + R1	acesso a variáveis estáticas
Indireto	`ADD (R1),R2`	R2 ← M[R1] + R2	acesso via ponteiros
Relativo à base	`ADD 100(R1),R2`	R2 ← M[100+R1] + R2	acesso a elementos em estruturas
Indexado	`ADD (R1+R2),R3`	R3 ← M[R1+R2] + R3	acesso a elementos em um vetor

Figura 3.4. Modos de endereçamento à memória mais comuns.

No modo direto, o endereço efetivo é um valor imediato contido no código da instrução. Por exemplo, na instrução `ADD (100),R1`, um dos operandos encontra-se na locação de memória com endereço 100. O modo de endereçamento direto é usado principalmente no acesso às variáveis estáticas de um programa, cujo endereço em memória pode ser determinado durante a compilação do programa.

No modo indireto, o endereço efetivo encontra-se em um registrador. Por exemplo, na instrução `ADD (R1),R2`, um dos operandos encontra-se na locação de memória cujo endereço está no registrador R1. Ou seja, o operando na memória é indicado indiretamente, através de um registrador que contém o endereço efetivo. Este modo de endereçamento é usado no acesso a variáveis dinâmicas, cujo endereço na memória é conhecido apenas durante a execução do programa. O acesso a uma variável dinâmica é realizado através de um ponteiro, que nada mais é do que o endereço da variável. Para realizar o acesso à variável dinâmica, o ponteiro é carregado em um registrador, e a instrução que acessa a variável usa este registrador com o modo de endereçamento indireto.

No modo relativo à base, o endereço efetivo é a soma do conteúdo de um registrador, chamado endereço-base, com um valor imediato contido na instrução, chamado deslocamento. Por exemplo, na instrução `ADD 100(R1),R2`, R1 contém o endereço-base e 100 é o deslocamento. O endereço efetivo do operando em memória é a soma do conteúdo de R1 com o valor 100. O modo relativo à base é usado no acesso a componentes de variáveis dinâmicas estruturadas (por exemplo, record em Pascal ou struct em C). A Figura 3.5 mostra como isto é feito.

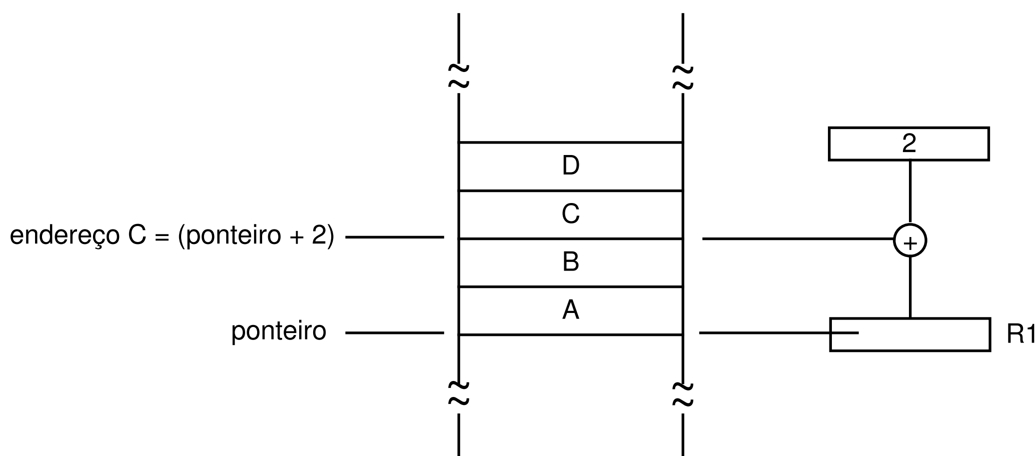


Figura 3.5. Acesso a estruturas dinâmicas com o modo de endereçamento relativo à base.

A figura mostra a localização na memória de uma estrutura com quatro campos A, B, C, D. O endereço inicial da estrutura é indicado por um ponteiro, que torna-se conhecido apenas durante a execução do programa. No entanto, a posição de cada campo em relação ao início da estrutura é fixo, sendo conhecido durante a compilação. O endereço de um campo é obtido somando-se a posição do campo (o deslocamento) ao ponteiro que indica o início da estrutura (o endereço-base). Por exemplo, na figura acima, para somar um valor ao campo C, o compilador pode usar a instrução `ADD 2(R1),R2`, precedida de uma instrução para carregar em R1 o endereço-base da estrutura.

No modo indexado, o endereço efetivo é dado pela soma de um índice com um endereço-base, ambos armazenados em registradores. Por exemplo, na instrução `ADD (R1+R2),R3`, R1 contém o endereço-base, e R2 o índice. O modo indexado é normalmente usado no acesso aos elementos de um vetor, como mostra a Figura 3.6.

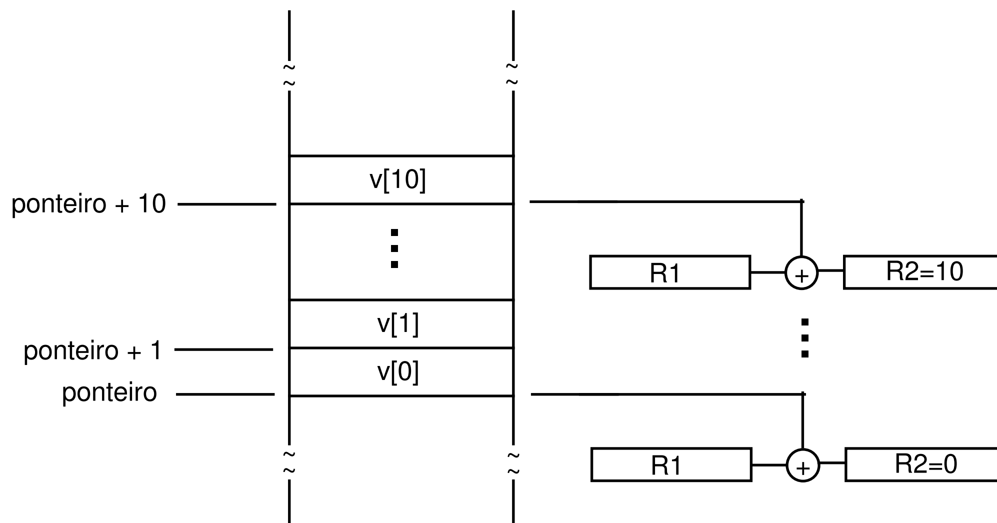


Figura 3.6. Acesso aos elementos de um vetor com o modo de endereçamento indexado.

A figura representa a localização na memória de um vetor V. Um ponteiro indica o endereço-base do vetor, onde se encontra o primeiro elemento. O endereço de cada elemento é obtido somando o índice do elemento ao endereço-base. Para realizar o acesso seqüencialmente os elementos do vetor, o índice é inicialmente carregado no registrador com o valor 0. O índice é então incrementado dentro de um loop após o acesso a cada elemento. Por exemplo, para somar um valor em registrador aos elementos do vetor, o compilador pode usar as seguintes instruções em um loop:

```
ADD   R1,(R2+R3)

ADD   1,R3
```

onde R1 contém o valor a ser somado, R2 contém o ponteiro para o vetor e R3 é o registrador com o índice, com valor inicial 0.

3.5 Formatos de Instrução

Como mencionado no início deste capítulo, as instruções de um programa compilado são armazenadas na memória sob a forma de um código em binário, ou código de instrução. Um código de instrução é logicamente formado por campos de bits, que contém as informações necessárias à execução da instrução. Estes campos de bits indicam, por exemplo, qual a operação a ser realizada e quais os operandos a serem usados. A Figura 3.7 mostra um exemplo de código de instrução com seus campos de bits.

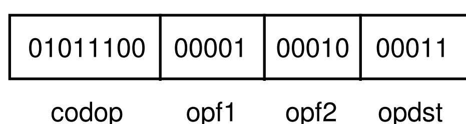


Figura 3.7. Código de instrução e seus campos de bits.

Neste código de instrução, o campo codop contém o código da operação a ser realizada, enquanto os campos opf1, opf2 e opdst indicam os operandos fonte e destino, respectivamente. Suponha que o código 01011100 no campo codop indique uma operação de adição, e que os valores 00001, 00010 e 00011 nos campos opf1, opf2 e opdst indiquem os registradores R1, R2 e R3, respectivamente. Assim, este código de instrução é a representação binária da instrução `ADD R1,R2,R3`.

O formato de instrução refere-se a características do código de instrução tais como tamanho do código, tipos de campos de bits e localização dos campos de bits dentro do código. Uma arquitetura se caracteriza por apresentar instruções com formato irregular ou com formato regular. No primeiro caso, as instruções podem apresentar códigos com tamanhos diferentes, e um certo campo de bits pode ocupar posições diferentes nos códigos onde aparece. Em uma arquitetura com instruções regulares, todos os códigos de instrução possuem o mesmo tamanho, e um certo campo de bits sempre ocupa as mesma posição nos códigos onde aparece.

Arquiteturas com formatos de instrução irregular possibilitam programas com menor tamanho de código executável. Isto acontece porque aqueles campos de bits não necessários à uma instrução são eliminados, economizando espaço de armazenamento na memória.

Por outro lado, arquiteturas com formatos de instrução regular apresentam uma grande vantagem quanto à simplicidade no acesso às instruções. Se todas as instruções possuem um tamanho de n bits, basta que o processador realize um único acesso de n bits à memória principal para obter uma instrução completa. Considere agora um processador com códigos de instrução com tamanho variável. Neste caso, o processador não sabe, a priori, quanto bits deve buscar para obter uma instrução completa. Após realizar um acesso, torna-se necessário que o

processador interprete parcialmente o código da instrução para determinar se deve realizar um outro acesso à memória para completar a busca da instrução. A decodificação parcial e o acesso adicional podem comprometer o desempenho ao aumentar o tempo de execução da instrução.

A segunda vantagem de instruções regulares é a simplicidade na decodificação das instruções. Em instruções regulares, um certo campo de bits sempre ocupa a mesma posição. Isto permite, por exemplo, que os operandos da instrução sejam acessados ao mesmo tempo que o código de operação é interpretado, já que o processador sabe antecipadamente onde encontrar as informações sobre os operandos. Em instruções irregulares, os campos que indicam os operandos podem aparecer em qualquer posição dentro do código da instrução. Assim, é necessário antes interpretar o código de operação, para determinar as posições dos campos de operando dentro daquele particular código de instrução. Agora, a decodificação e o acesso aos operandos são realizados seqüencialmente, o que contribui para aumentar o tempo de execução das instruções.

4 A ARQUITETURA DOS PROCESSADORES INTEL 80X86

Os Capítulos 2 e 3 discutiram os conceitos básicos relacionados à arquitetura de um processador. Este capítulo ilustra estes conceitos através de exemplos de processadores reais. Os processadores selecionados para os exemplos fazem parte da família Intel 80x86. Esta escolha deve-se ao fato destes serem os processadores usados nos microcomputadores pessoais do tipo IBM PC.

O objetivo deste capítulo é dar uma visão geral da arquiteturas destes processadores com base nos conceitos vistos até agora. Conceitos avançados, tais como memórias cache, memória virtual e pipelining, incorporados nos processadores da linha 80x86, serão discutidos em capítulos posteriores. O mais recente lançamento da Intel, o processador Pentium, será analisado no capítulo que discute arquiteturas de processadores super-escalares.

4.1 Quadro Evolutivo da Família Intel 80x86

O primeiro microprocessador Intel foi o Intel4004, produzido em 1969. Este processador era um dispositivo de 4 bits que continha uma ALU, uma unidade de controle e alguns registradores. O Intel 4004 foi rapidamente substituído pelo Intel 8008. O 8008 era um processador de 8 bits com características similares às do 4004. Em 1979, a Intel produziu o seu primeiro processador de 16 bits, o 8086. Um processador similar a este, o 8088, foi usado no primeiro IBM PC. A diferença entre eles era o barramento de dados com 8 bits no 8088 e 16 bits no 8086. O sucessor deste processador foi o Intel 80286, também com 16 bits. Em 1986, surgiu o primeiro processador de 32 bits, o Intel 80386. O Intel 80386 foi substituído, em 1989, pelo Intel 80486, um processador com arquitetura de 32 bits. Em 1993, surgiu o processador Pentium, com características distintas dos processadores anteriores e uma arquitetura de 64 bits. Nesta linha, os próximos processadores de 64 bits foram o Pentium Pro, o PentiumII e o PentiumIII.

Desde o lançamento do processador 8086, em 1978, a família Intel 80x86 passou por cinco gerações, lançando no mercado um novo processador aproximadamente a cada quatro anos. A Tabela 4.1 mostra a evolução da linha 80x86.

Arquitetura	Introdução	Número Transistores
8086/8088	Junho 1978	29.000
80286	Fevereiro 1982	134.000
80386	Outubro 1985	275.000
80486	Agosto 1989	1.200.000
Pentium	Março 1993	3.100.000

Pentium Pro (150 – 200 MHZ)	Novembro 1995	5.500.000
Pentium II (350 – 400 MHZ)	Abril 1998	7.500.000

Tabela 4.1. Evolução da família Intel 80x86.

Cada processador da família pode possuir mais de uma versão, que diferem entre si apenas na tecnologia de integração. Em geral, a última versão de um processador apresenta um ganho no desempenho maior que o dobro da sua versão original. Por sua vez, o desempenho de cada nova geração se aproxima do dobro do desempenho da última versão da geração anterior.

A produção em larga escala garantiu uma queda significativa no custo. Observe também que o número de transistores usados na implementação dos processadores cresceu exponencialmente desde o lançamento do 8086, que contava com 29.000 transistores. Durante este período, o rápido desenvolvimento na tecnologia de integração possibilitou que o número de transistores ultrapassasse 7 milhões na época do lançamento do Pentium II.

4.2 O Intel 8086/8088

O 8086 foi lançado procurando atender às necessidades de aplicações que exigiam processamento de 16 bits, maior capacidade de memória, modos de endereçamento mais sofisticados que facilitassem a execução de programas escritos em linguagens de alto nível e suporte para sistemas operacionais multiprogramados. A arquitetura do 8086 é mostrada na Figura 4.1.

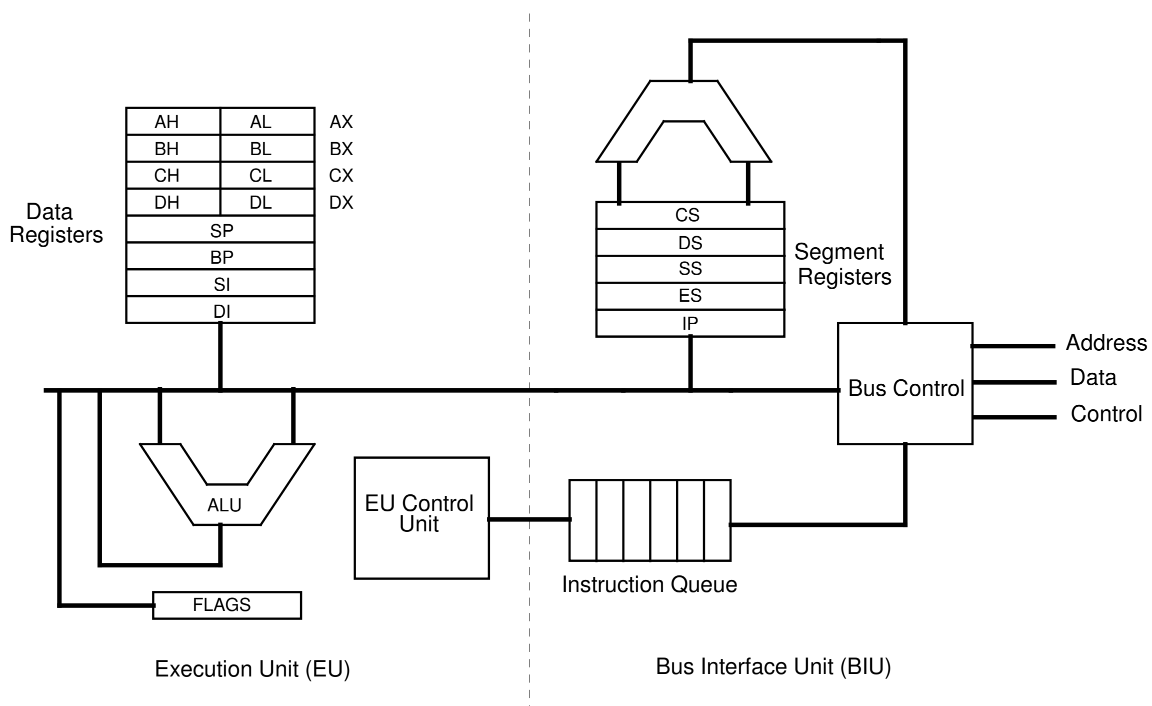


Figura 4.1. Arquitetura do Intel 8086.

O Intel 8088 possui a mesma organização do 8086. As únicas diferenças entre estes dois processadores estão no tamanho da fila de instruções e na largura do barramento de dados. No 8086, a fila de instruções é de 8 bytes e o barramento de dados possui uma largura de 16 bits, enquanto no 8088 a fila de instruções e o barramento de dados são de 4 bytes e 8 bits, respectivamente. O barramento de 8 bits no 8088 foi adotado para manter uma compatibilidade com as interfaces de entrada/saída de 8 bits, existentes na época. Dada a semelhança das duas arquiteturas, a partir de agora se fará referência apenas ao 8086.

A arquitetura do 8086 é composta de duas unidades principais. A unidade de execução, ou EU (Execution Unit) engloba a unidade lógica e aritmética e a unidade de controle. A unidade de interface de barramento, BIU (Bus Interface Unit) é responsável por controlar os acessos externos à memória principal e aos dispositivos de entrada/saída. Se durante sua execução uma instrução requer um acesso a um dado armazenado na memória, a EU solicita o acesso à BIU. As principais características destas duas unidades são descritas a seguir.

4.2.1 A Unidade de Execução

A unidade aritmética e lógica opera sobre dados de 8 ou 16 bits. Associado à ALU existe um conjunto de 4 registradores de dados de uso geral, denominados AX, BX, CX e DX. Cada um destes registradores de 16 bits é formado por uma metade inferior (AL, BL, CL, DL) e por uma metade superior (AH, BH, CH, DH), onde podem ser realizados acessos individualmente, constituindo cada metade um registrador de 8 bits. Esta organização dos registradores no 8086 foi adotada para manter a compatibilidade com programas desenvolvidos para o Intel 8080/8085, que possuía apenas registradores de 8 bits.

O registrador AX serve como acumulador em instruções aritméticas e lógicas. O registrador BX é usado por algumas instruções para armazenar ponteiros para dados armazenados na memória. CX é usado em algumas instruções especiais, como a de controle de loop. O registrador DX é utilizado como uma extensão do registrador AX em instruções de divisão e multiplicação, sendo nele armazenada a metade superior de um produto de 32 bits ou de um dividendo de 32 bits.

Além dos registradores de uso geral, existem os registradores indexadores e os registradores apontadores. Os registradores indexadores SI (Source Index) e DI (Destination Index) são utilizados em operações envolvendo cadeias de caracteres (strings). Os registradores SP (Stack Pointer) e BP (Base Pointer) são normalmente usados para acessar a pilha em chamadas e retornos de rotinas[2]. O registrador SP aponta para o topo da pilha, enquanto o registrador BP aponta para a área de dados locais (na pilha) da rotina em execução.

[2]Pilha é uma estrutura de dados em que o último dado armazenado é o primeiro a ser recuperado.

O registrador de estado (flags) associado à ALU inclui bits para registrar informações sobre o resultado de uma operação aritmética e lógica e ainda alguns bits de controle.

4.2.2 A Unidade de Interface de Barramento (BIU)

Todos os acessos à memória e às interfaces de e/s são controlados pela BIU. O 8086 possui um barramento de endereços de 20 bits, comportando assim uma memória principal com tamanho máximo de 1 Mbyte.

Sob o ponto de vista do 8086, a memória principal apresenta uma organização especial, chamada segmentação. Nesta organização, a memória é logicamente dividida em blocos, chamados segmentos, conforme mostra a Figura 4.2(a). Cada segmento possui um endereço-base, que é o endereço onde inicia-se o segmento. Uma locação de memória é referenciada pelo programa através de seu deslocamento (offset), que indica a posição da locação relativa ao início do segmento. Com o endereço-base do segmento e o deslocamento da locação dentro do segmento, o hardware calcula o endereço da locação em relação ao início da memória.

A Figura 4.2(b) mostra como o cálculo do endereço da locação é feito no caso do 8086. A BIU possui um conjunto de registradores de segmento, onde cada registrador armazena os 16 bits mais significativos do endereço-base do segmento. A cada acesso, este valor é concatenado à esquerda com quatro bits 0, formando um endereço de 20 bits. A este endereço é somado o deslocamento, resultando no endereço da locação. O deslocamento é um valor de 16 bits, o que significa que os segmentos podem ter um tamanho máximo de 64 Kbytes. Para efetuar o cálculo do endereço, a BIU possui um somador dedicado.

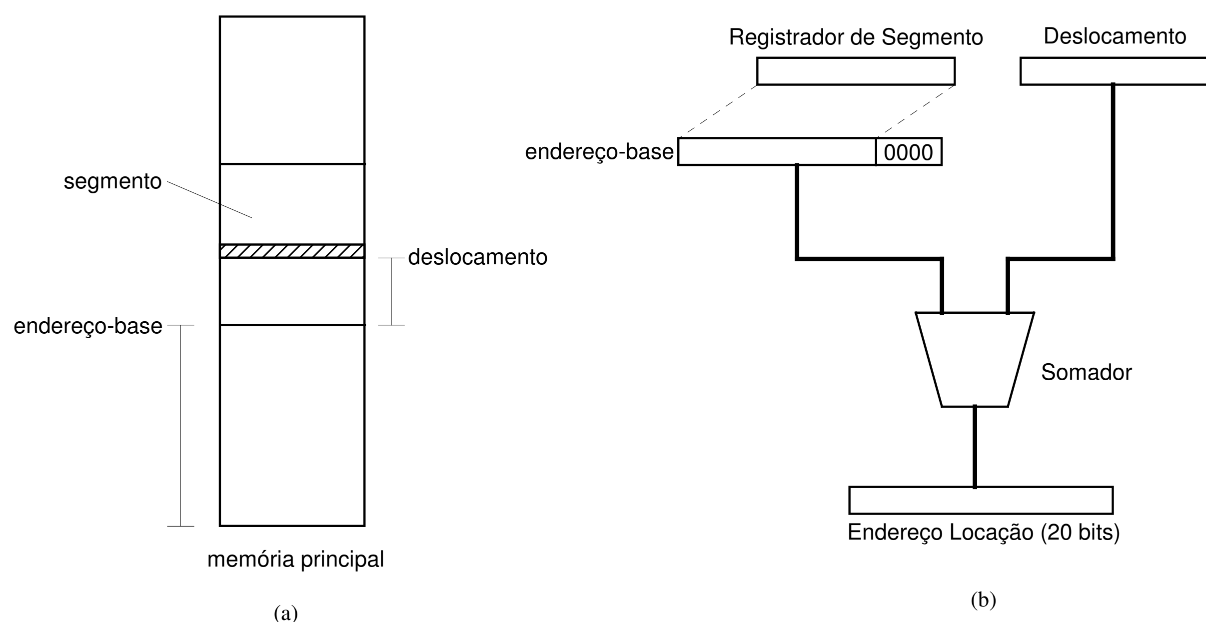


Figura 4.2. O mecanismo de segmentação.

No caso do 8086, um programa residente na memória possui quatro segmentos. O segmento de código que contém as instruções do programa. O

segmento de dados que corresponde à área de dados do programa. O segmento de pilha que é a área que se destina à pilha do programa. E o segmento extra, que consiste de uma área de dados adicional. Para cada um destes segmentos existe um registrador de segmento na BIU que, como mencionado, armazena os 16 bits mais significativos do endereço-base do segmento correspondente. Estes registradores são denominados CS (Code Segment register), DS (Data Segment register), SS (Stack Segment register) e ES (Extra Segment register).

Os quatro segmentos de um programa são referenciados de acordo com o tipo de acesso à memória. Buscas de instrução são feitas dentro do segmento de código indicado pelo valor corrente no registrador de segmento CS. Nos acessos à instruções, o deslocamento é dado pelo conteúdo de um registrador especial da BIU, denominado IP (Instruction Pointer). O IP indica a próxima instrução a ser acessada na memória. Note que existe uma diferença entre IP e contador de programa, definido no Capítulo 2, já que este último indica a próxima instrução a ser executada. Esta diferença deve-se ao mecanismo de busca antecipada de instruções existente no 8086, que será descrito logo à frente.

Acessos à dados em memória acontecem dentro do segmento de dados indicado pelo registrador DS. O deslocamento é gerado pelo programa de acordo com o modo de endereçamento usado. Acessos a dados também podem acontecer dentro do segmento extra, apontado pelo registrador ES, quando explicitamente indicado pelo programa. Finalmente, acessos à pilha ocorrem dentro do segmento de pilha indicado pelo registrador SS.

Uma outra característica do 8086 é o mecanismo de busca antecipada de instruções (instruction prefetch). Neste mecanismo, o passo de busca de instrução é desvinculado dos demais passos de execução da instrução. Isto permite que novas instruções sejam buscadas antecipadamente, enquanto uma instrução anterior ainda está sendo executada. A Figura 4.3 compara a busca antecipada de instruções com o modo de busca convencional.

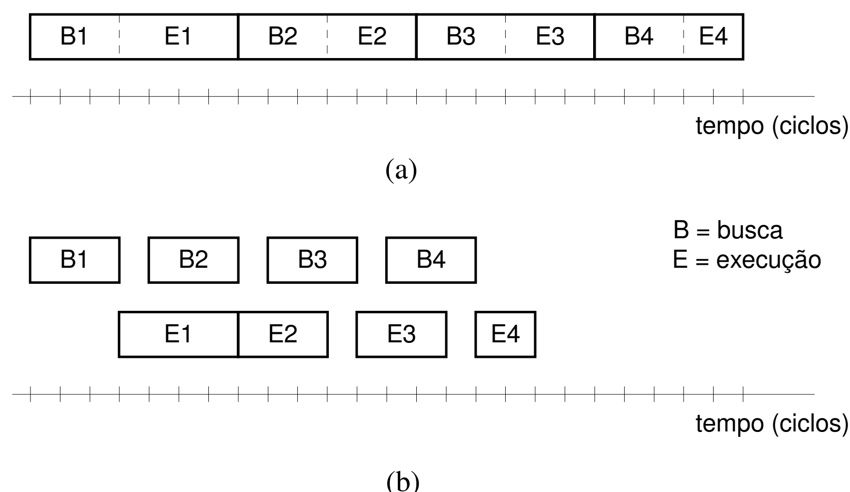


Figura 4.3. Busca convencional e busca antecipada de instruções.

A Figura 4.3(a) mostra um modo de acesso seqüencial às instruções. Neste caso, a busca de uma nova instrução inicia-se somente após a instrução anterior ter

sido completada. A Figura 4.3(b) mostra como funciona a busca antecipada. Neste modo, a busca e a execução de uma instrução passam a ser atividades distintas. Desta forma, o acesso de uma nova instrução pode ser iniciado antes mesmo que a execução da instrução anterior tenha sido completada. Por exemplo, na figura acima, a busca da instrução 2 é iniciada enquanto a instrução 1 ainda está sendo executada. O mesmo acontece para as instruções seguintes.

Qual o efeito da busca antecipada de instruções? Considere o tempo de conclusão de uma instrução como sendo o intervalo de tempo entre o término de uma instrução e o término da instrução seguinte. No modo convencional, como a busca da instrução está acoplada à execução propriamente dita, o tempo de busca soma-se ao tempo de execução, aumentando o tempo de conclusão. Por exemplo, na Figura 4.3(a), vê-se que dentro do tempo de conclusão de cada instrução sempre existe uma parcela fixa, de 3 ciclos, consumida na busca da instrução.

Na busca antecipada, a fase de busca e a fase de execução são separadas, e o tempo de busca da instrução seguinte é "mascarado" pelo tempo de execução da instrução corrente. Assim, a contribuição do tempo de busca dentro do tempo de conclusão é menor. Por exemplo, na Figura 4.3(b), o tempo de busca da instrução 2 é totalmente mascarado pelo tempo de execução da instrução 1. Neste caso, a contribuição do tempo de busca é efetivamente anulado. Em um outro exemplo, dois ciclos do tempo de busca da instrução 3 são mascarados pelo tempo de execução da instrução 2. Agora, apenas um ciclo de busca aparece no tempo de conclusão da instrução 3.

Na busca antecipada de instruções, a BIU utiliza uma fila de instruções. Uma instrução acessada antecipadamente permanece nesta fila até ser retirada pela EU. A BIU procura manter a fila de instruções sempre cheia, realizando uma busca de instrução sempre que não existe um pedido de acesso solicitado pela EU.

Note que em uma arquitetura sem busca antecipada, o contador de programa indica a próxima instrução a ser executada, que é exatamente a próxima instrução a ser acessada. Em uma arquitetura com busca antecipada, a próxima instrução a ser executada não é necessariamente a próxima instrução a ser acessada. No 8086, a próxima instrução a ser executada já foi trazida anteriormente, e encontra-se na frente da fila de instruções. O registrador IP, localizado na BIU, aponta para a próxima instrução a ser buscada.

A busca antecipada de instruções reduz o tempo médio de conclusão de cada instrução. Como conseqüência, o tempo necessário para executar um certo número de instruções é menor, o que significa um maior desempenho. Na realidade, a busca antecipada de instruções é uma forma limitada de paralelismo empregada no 8086 para aumentar o desempenho. O uso de paralelismo a nível de execução de instruções, em formas bem mais sofisticadas do que a existente no 8086, está se tornando comum em arquiteturas de processadores. Execução paralela de instruções, como na técnica de pipelining e nas arquiteturas super-escalares, serão discutidas no Capítulo 7.

4.2.3 O Conjunto de Instruções do 8086

As instruções do 8086 podem ser classificadas nos seguintes grupos:

- instruções aritméticas
- instruções lógicas
- instruções de manipulação de strings
- instruções de transferência de dados
- instruções de transferência de controle

A arquitetura do 8086 é do tipo memória-registrador. Como visto no Capítulo 3, neste tipo de arquitetura as instruções aritméticas e lógicas especificam explicitamente dois operandos, sendo que um deles serve ao mesmo tempo como operando-fonte e operando-destino. Apenas um dos operandos pode se encontrar na memória. Estas instruções estão relacionadas na Tabela 4.2.

ADD	add
ADC	add with carry
INC	increment by 1
AAA	ASCII adjust for addition
DAA	decimal adjust for addition
SUB	subtract
SBB	subtract with borrow
DEC	decrement by 1
NEG	negate
CMP	compare
AAS	ASCII adjust for subtraction
DAS	decimal adjust for subtraction
MUL	multiply byte
IMUL	integer multiply
AAM	ASCII adjust for multiply
DIV	divide
IDIV	integer divide

AAD	ASCII adjust for division
CBW	convert byte to word
CWD	convert word to doubleword

Tabela 4.2. Instruções aritméticas no 8086.

As instruções aritméticas operam sobre números de 8 ou 16 bits, com ou sem sinal. Existem instruções aritméticas que efetuam adição, subtração, multiplicação e divisão sobre inteiros representados em binário e sobre inteiros representados em BCD. Além destas principais, existem várias outras instruções. Por exemplo, existem instruções para comparar dois inteiros, para incrementar ou decrementar o conteúdo de registradores, para converter entre um número inteiro e um caracter em representação ASCII e ainda para converter entre inteiros com tamanhos diferentes.

As instruções lógicas estão relacionadas na Tabela 4.3. São oferecidas instruções que realizam as operações booleanas básicas (complemento, and, or, xor). Estão incluídas instruções que realizam deslocamentos aritméticos e lógicos à esquerda e à direita, como também instruções para rotação do conteúdo de registradores.

NOT	boolean not
AND	boolean and
OR	boolean or
XOR	boolean exclusive-or
TEST	boolean test
SHL	shift logical left
SAL	shift arithmetic left
SHR	shift logical right
SAR	shift arithmetic right
ROL	rotate left
ROR	rotate right
RCL	rotate through carry left
RCR	rotate through carry right

Tabela 4.3. Instruções lógicas no 8086.

As instruções de manipulação de strings, relacionadas na Tabela 4.4, possibilitam vários tipos de operações sobre cadeias de caracteres. Estas operações são: movimentação de strings entre regiões diferentes da memória, comparação de dois strings, procura de um caracter dentro de um string e transferência de strings entre registradores e a memória. Na realidade, estas operações são executadas por instruções primitivas (`MOVS`, `CMPS`, `SCAS`, `LODS`, `STOS`) que, em geral, são precedidas de um prefixo (`REP`, `REPE/REPZ`, `REPNE/REPNZ`) que fazem a execução da instrução primitiva ser repetida até que uma condição indicada pelo prefixo seja satisfeita.

`MOVS`	move string
`CMPS`	compare string
`SCAS`	scan string
`LODS`	load string
`STOS`	store string
`REP`	repeat
`REPE/REPZ`	repeat while equal/zero
`REPNE/REPNZ`	repeat while not equal/not zero

Tabela 4.4. Instruções de manipulação de strings.

As instruções de transferência de dados estão relacionadas na Tabela 4.5. Basicamente, existem dois tipos de instruções de transferência de dados, as de uso geral e as especiais. As instruções de uso geral (`MOV`, `PUSH`, `POP`, `XCHG`, `XLAT`) transferem dados entre registradores ou entre registradores e a memória. Nestas estão incluídas as instruções de acesso à pilha. Nas instruções de transferência especiais, estão instruções para carregar registradores de segmento, salvar/carregar o registrador de estado e para realizar o acesso às interfaces de entrada/saída.

`MOV`	move byte or word
`PUSH`	push word onto stack
`POP`	pop word off stack
`XCHG`	exchange byte or word
`XLAT`	translate byte
`LEA`	load effective address
`LDS`	load pointer using DS

LES	load pointer using ES
LAHF	load AH register from flags
SAHF	store AH register in flags
PUSHF	push flags onto stack
POPF	pop flags off stack
IN	input byte or word
OUT	output byte or word

Tabela 4.5. Instruções de transferência de dados.

As instruções de transferência de controle estão relacionadas na Tabela 4.6. Nas instruções de transferência condicional um desvio pode ou não ser executado, dependendo se a condição testada pela instrução é ou não verdadeira. Estas instruções testam o valor de um bit do registrador de estado associado à ALU e realizam o desvio de acordo com o estado do bit. O bit(ou bits) a ser testado é indicado pela instrução. Nas instruções de transferência incondicional o desvio é sempre efetuado. Nas instruções de transferência incondicional estão as de chamada/retorno de rotinas. Existem também as instruções de controle de loop. Estas instruções aparecem normalmente no final do loop, e fazem com que uma nova interação do loop seja executada enquanto uma certa condição for satisfeita. Existe ainda uma instrução especial (INT), que provoca uma interrupção (interrupções serão discutidas no Capítulo 6). Esta instrução é normalmente usada para implementar chamadas ao sistema operacional.

JA	jump if above	CALL	call routine
JAE	jump if above or equal	RET	return from routine
JB	jump if below	JMP	jump unconditionally
JBE	jump if below or equal	LOOP	loop
JC	jump if carry	LOOPE/LOOPZ	loop if equal/zero
JE	jump if equal	LOOPNE/LOOPNZ	loop if not equal/not zero
JG	jump if greater	JCXZ	jump if CX=0
JGE	jump if greater or equal	INT	interrupt
JL	jump if less		
JLE	jump if less or equal		

JNC	jump if not carry
JNE	jump if not equal
JNO	jump if not overflow
JNP	jump if not parity
JNS	jump if not sign
JO	jump if overflow
JP	jump if parity
JS	jump if sign

Tabela 4.6. Instruções de transferência de controle.

O conjunto de instruções do 8086 oferece os seguintes modos de endereçamento: registrador, imediato, direto, indireto via registrador, relativo à base, indexado e relativo à base indexado. Estes modos de endereçamento já foram discutidos genericamente no capítulo anterior. No caso particular do 8086, os registradores BX, BP, SI e DI são usados como base ou índice nos modos de endereçamento relativo à base e indexado. A Tabela 4.7 mostra as possibilidades de cálculo do endereço efetivo nestes modos de endereçamento.

Modo	Endereço Efetivo
Baseado	BX + deslocamento BP + deslocamento
Indexado	SI + deslocamento DI + deslocamento
Baseado Indexado	BX + SI + deslocamento BX + DI + deslocamento BP + SI + deslocamento BP + DI + deslocamento

Tabela 4.7. Cálculo do endereço efetivo nos modos relativo à base/indexado.

4.3 O Intel 80186/80188

A arquitetura básica do 80186 (80188) é praticamente igual a do 8086 (8088). A diferença é que estes processadores incorporaram no mesmo dispositivo cerca de 15 a 20 dos componentes mais comuns dos sistemas baseados no

8086/8088, tais como controlador de interrupção e controlador de DMA (ver Capítulo 6). O código objeto do 80186/80188 é totalmente compatível com o do 8086/8088. Devido às inúmeras similaridades com o 8086/888, estes processadores não serão aqui descritos.

4.4 O Intel 80286

As principais novidades introduzidas na arquitetura do 80286 foram a execução de instruções em pipeline e o suporte para memória virtual. A Figura 4.4 mostra a arquitetura do 80286.

A arquitetura do 80286 é composta por unidades independentes que realizam funções específicas. A unidade de barramento (bus unit) controla os acessos à memória e interfaces de e/s, e realiza a busca antecipada de instruções. Instruções acessadas são colocadas em uma fila na unidade de barramento. A unidade de instrução (instruction unit) decodifica as instruções, colocando as instruções decodificadas em uma fila. A instrução decodificada fornece de maneira mais detalhada todas as informações necessárias à execução da instrução. Por exemplo, a instrução decodificada indica explicitamente o endereço de entrada da microrotina que executa a instrução.

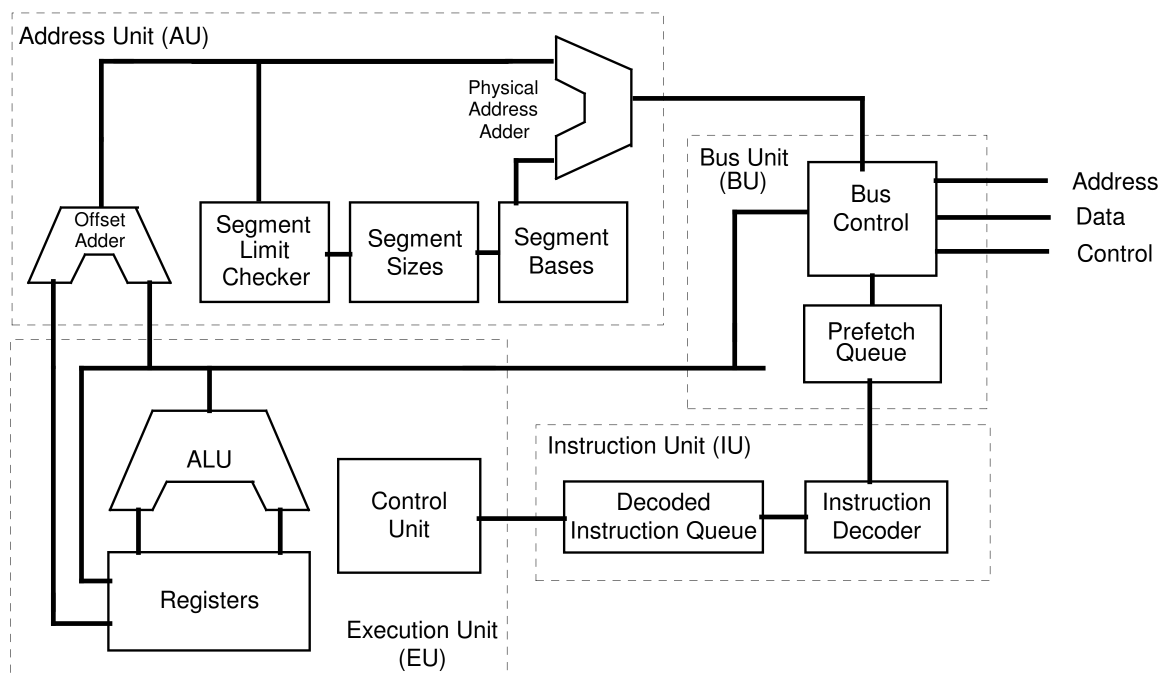


Figura 4.4. Arquitetura do Intel 80286.

A unidade de controle, a ALU e o conjunto de registradores formam a unidade de execução (execution unit). Instruções decodificadas são retiradas da fila onde se encontram pela unidade de controle. A ALU do 80286 é de 16 bits, e o conjunto de registradores é idêntico ao do 8086. A unidade de endereço (address unit) é responsável por controlar a memória virtual. A memória virtual é uma técnica que torna possível a execução eficiente de programas com tamanho de código maior que o tamanho da memória principal.

O 80286 pode operar em dois modos, chamados modo real (real mode) e modo protegido (protected mode). No modo real, o 80286 opera com um modelo de memória semelhante ao 8086, ou seja, uma memória segmentada com tamanho máximo de 1 Mbyte. No modo protegido, o 80286 opera com memória virtual. Neste modo, a memória do 80286 é logicamente dividida em 16.384 segmentos, cada um com 64 Kbytes de tamanho. Logo, pode-se executar programas com até 1 Gbyte de tamanho, embora o tamanho máximo da memória principal seja 16 Mbytes (barramento de endereço externo de 24 bits). Além disso, este modo de operação fornece um suporte mais eficiente para sistemas operacionais multiprogramados, já que provê proteção entre segmentos pertencentes a diferentes programas (daí a denominação de protegido a este modo de operação). A técnica de memória virtual será discutida em detalhes no próximo capítulo.

Um ponto importante é que estes dois modos de operação são estáticos. Quando o 80286 é iniciado, ele começa a operar no modo real. A execução de uma instrução especial comuta a sua operação para o modo protegido. Uma vez neste modo, o 80286 não mais pode voltar para o modo real sem uma reinicialização. Também é importante salientar que programas para o 8086 não podem ser executados no modo protegido. Isto acontece porque, no modo protegido, os registradores de segmento assumem funções diferentes daquelas no 8086. Como será visto, esta limitação é corrigida no 80386.

As diversas unidades no 80286 operam simultaneamente, realizando tarefas que fazem parte da execução de diferentes instruções. Por exemplo, enquanto a unidade de barramento está realizando o acesso ao operando de uma instrução, a unidade de instrução está decodificando uma outra instrução e a unidade de execução está executando uma terceira instrução. Esta execução paralela de instruções em diferentes unidades é conhecida como pipeline. Com pipelining é possível obter aumentos significativos no desempenho, e por isso esta técnica é atualmente usada em praticamente todas as arquiteturas de processador. A técnica de pipelining é discutida no Capítulo 7.

O conjunto de instruções do 80286 é basicamente o mesmo do 8086. Foram acrescentadas duas instruções, `PUSHA` e `POPA`, que realizam o salvamento/recuperação dos registradores na pilha em memória. A instrução `BOUND` verifica se um índice de vetor encontra-se dentro dos limites. A instrução `ENTER` cria uma área para armazenamento de dados na pilha, sendo usada na implementação de chamada de rotinas. A instrução `LEAVE` é usada em retornos de rotinas, e libera uma área de dados na pilha. Existem também outras instruções relacionadas com o controle da memória virtual. Os modos de endereçamento são os mesmos do 8086. A Tabela 4.8 resume as principais diferenças entre as arquiteturas 8086 e 80286.

8086	80286
modo real apenas	modos real e protegido
1 Mbyte memória principal máxima	16 Mbytes memória principal máxima

fraco suporte para multiprogramação	melhor suporte para multiprogramação, com mecanismo de proteção entre programas
execução de instruções seqüencial	execução de instruções em pipeline

Tabela 4.8. Principais diferenças entre o 8086 e o 80286.

4.5 A Arquitetura do Intel 80386

Existem duas versões da arquitetura 80386 denominadas SX e DX. A única diferença entre estas versão é que, no 80386 SX, o barramento de dados externo é de 16 bits, enquanto no 80386 DX o barramento de dados é de 32 bits. A Figura 4.5 mostra a organização do Intel 80386.

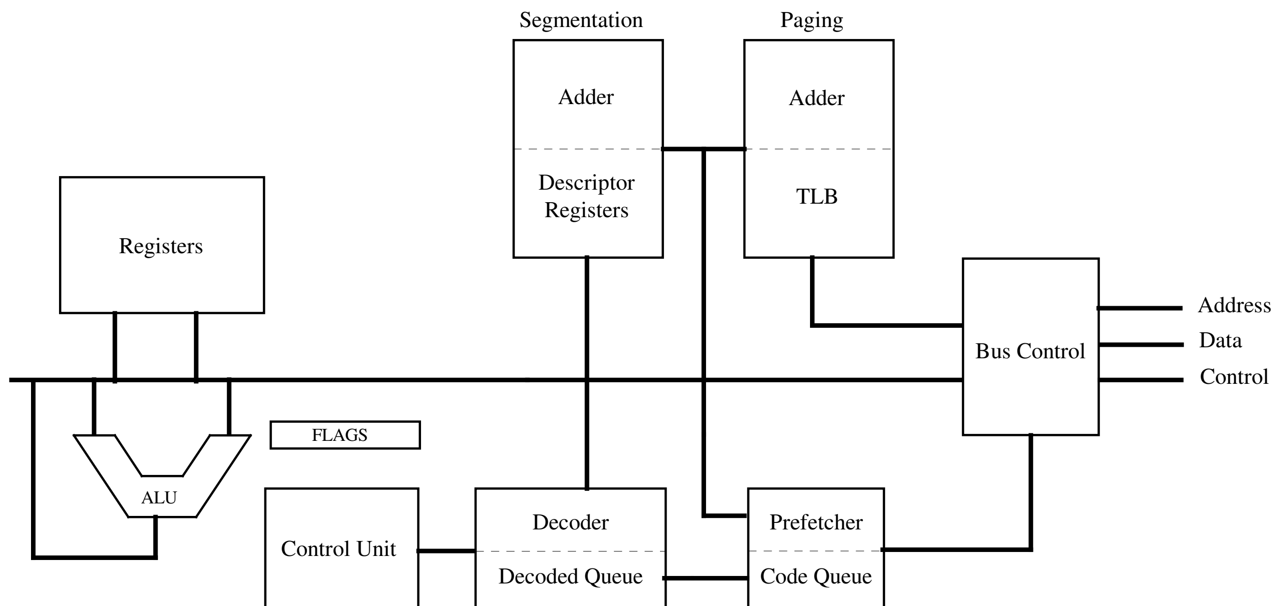


Figura 4.5. Arquitetura do Intel 80386.

A arquitetura do 80386 é formada por oito unidades lógicas. Assim como nas arquiteturas 8086 e 80286, a unidade de barramento (bus interface unit) é responsável por controlar os acessos externos à memória e interfaces de e/s. Esta unidade recebe pedidos de acesso à memória de várias outras unidades. A busca antecipada de instruções é feita pela unidade de pré-busca (prefetch unit). Esta unidade possui uma fila de instruções (code queue) de 16 bytes, e solicita acessos à memória para a unidade de barramento sempre que esta fila se encontra parcialmente vazia ou quando uma instrução de transferência de controle é executada.

A unidade de decodificação (decode unit) é responsável por decodificar as instruções. Esta unidade retira uma instrução da fila de instruções e gera uma

instrução decodificada que é armazenada na fila de instruções decodificadas (decoded queue). A unidade de controle retira instruções decodificadas desta fila para serem executadas. Esta unidade controla todas as demais unidades que participam na execução da instrução.

A unidade de dados (data unit) é formada por uma ALU e registradores de 32 bits. O conjunto de registradores do 80386 é mostrado na Figura 4.6.

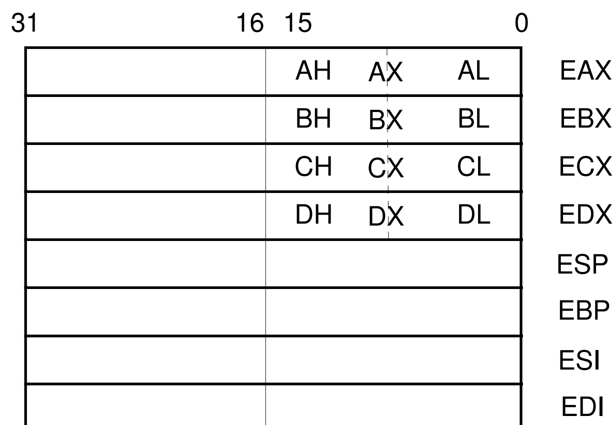


Figura 4.6. Conjunto de registradores do 80386.

O 80386 possui 4 registradores de dados de 32 bits, EAX, EBX, ECX e EDX. Podem ser feitos acessos à metade inferior de cada um destes registradores, como registradores de 16 bits AX, BX, CX e DX. Por sua vez, cada um destes registradores de 16 bits podem ser considerados como contendo dois registradores de 8 bits. Com esta organização, o 80386 mantém a compatibilidade com as arquiteturas anteriores. Além dos registradores de dados, existem os registradores apontadores ESP e EBP e os registradores indexadores ESI e EDI. Nestes registradores são acessados apenas como registradores de 32 bits.

As três unidades restantes, protection unit, segmentation unit e paging unit gerenciam a memória virtual. O 80386 implementa um modelo de memória virtual organizada em segmentos e páginas. A unidade de proteção verifica se um endereço se encontra dentro dos limites de um segmento, enquanto as unidades de segmentação e paginação realizam a conversão entre endereço virtual e endereço real. O mecanismo de memória virtual do 80386 será examinado em detalhes no próximo capítulo.

Além dos modos real e protegido, existentes no 80286, o 80386 pode também operar em um terceiro modo, chamado modo 8086 virtual (virtual 8086 mode). Neste modo, programas desenvolvidos para o 8086 podem ser executados em um ambiente multiprogramado protegido, ou seja, em um ambiente onde múltiplos programas podem ser executados simultaneamente, sem que um programa interfira na execução de um outro programa. Além disso, ao contrário do que acontece entre os modos real e protegido no 80286, no 80386 é possível comutar livremente do modo protegido para o modo virtual, e vice-versa. Isto é o

que acontece, por exemplo, quando uma janela MS-DOS é aberta a partir da interface gráfica MS Windows. Normalmente, o MS Windows opera no modo protegido. Quando é solicitada uma janela MS-DOS, o 80386 é comutado para o modo virtual, possibilitando que programas MS-DOS, escritos para o 8086, possam ser executados.

A arquitetura 80386 oferece um longo conjunto de instruções, incluindo todas as instruções das arquiteturas 8086 e 80286. Dentre as novas instruções introduzidas na arquitetura 80386, destacam-se novas instruções lógicas para deslocamento e teste de bits, para procura de caracteres em strings, e para movimentação e conversão de dados.

O 80386 oferece 11 modos de endereçamento, incluindo todos os modos de endereçamento do 8086 e 80286. O 80386 adiciona um novo componente no cálculo do endereço efetivo, chamado escala (scale). A escala é um fator (1, 2, 4 ou 8) que multiplica o índice nos modos de endereçamento indexados. Isto facilita o acesso a vetores com elementos de 16, 32 ou 64 bits. Por exemplo, para percorrer um vetor com elementos de 32 bits, são necessários incrementos de 4 bytes no endereço efetivo. Usando uma escala 4, cada incremento de 1 do índice resulta, após a multiplicação do índice pela escala, no incremento de 4 bytes no endereço efetivo.

A Tabela 4.9 relaciona as principais diferenças entre o 80286 e o 80386.

80286	80386
modos real e protegido	modos real, protegido e virtual
16 Mbytes de memória principal máxima	4 Gbytes de memória principal máxima
aritmética de 8 e 16 bits	aritmética de 8, 16 e 32 bits
registradores de 16 bits	registradores de 32 bits

Tabela 4.9. Principais diferenças entre o 80286 e o 80386.

4.6 A Arquitetura do Intel 80486

A arquitetura do 80486 é basicamente uma extensão da arquitetura do 80386. No 80486, uma memória cache de 8 Kbytes e uma unidade de ponto flutuante são acrescentados à arquitetura do 80386. Em sistemas baseados no 80386, a memória cache e a unidade de ponto flutuante são externos ao processador. A Figura 4.7 mostra a arquitetura do 80486.

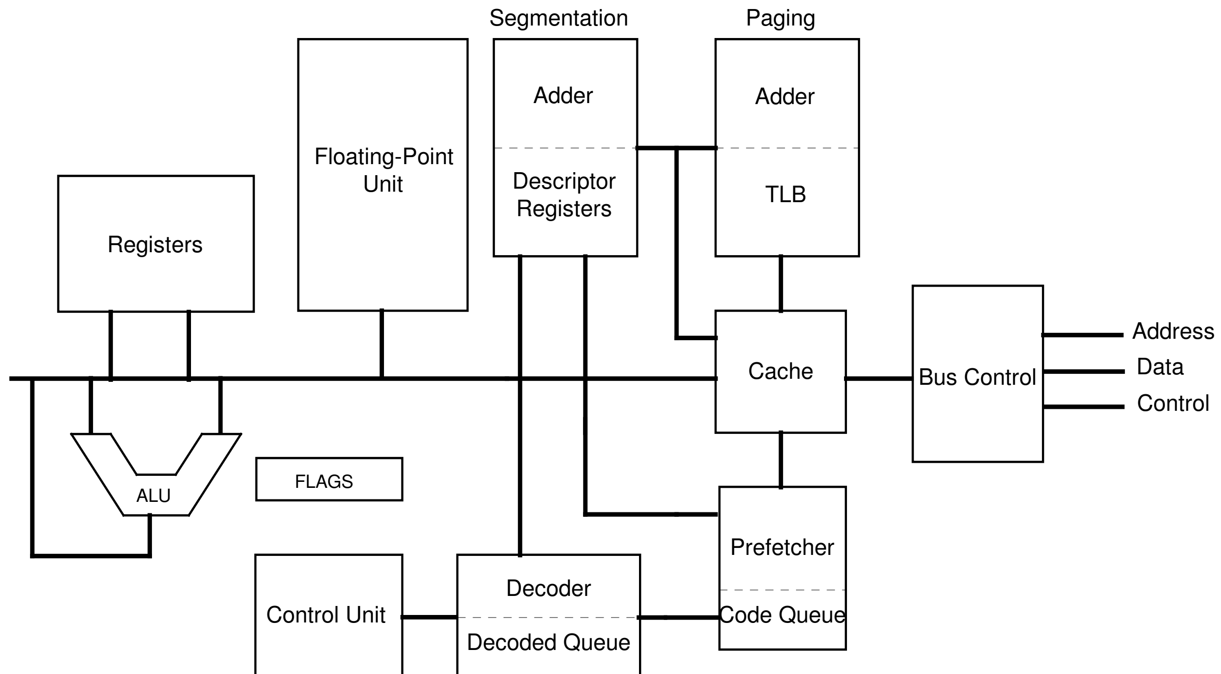


Figura 4.7. Arquitetura do Intel 80486.

A arquitetura 80486 possui três versões de implementação, denominadas 486-SX, 486-DX e 486-DX2. A versão 486-SX não possui a unidade de ponto flutuante integrada com o processador, mas apenas a memória cache. A versão 486-DX possui unidade de ponto flutuante e memória cache integradas com o processador. Na versão 486-DX, a freqüência de clock do processador é a mesma usada nos ciclos de barramento para acesso à memória e interfaces de e/s. Na versão 486-DX2, a freqüência de clock interna ao processador é o dobro da freqüência externa usada nos ciclos de barramento. Isto permite obter uma compatibilidade entre novas versões mais rápidas do processador com um hardware de sistema mais antigo e lento.

Devido à total similaridade entre as arquiteturas 80386 e 80486, a descrição da arquitetura 80486 seria uma repetição da seção anterior. No próximo capítulo serão descritos a memória cache e o mecanismo de memória virtual usados no 80486.

4.7 O Intel Pentium

O Pentium é um dos membros mais recente da família 80x86. Até o Pentium, a principal inovação arquitetural na família 80x86 era a técnica de pipeline, introduzida com o Intel 80286. O Pentium introduz uma outra facilidade igualmente importante para o desempenho, a execução super-escalar de instruções.

Na arquitetura Pentium podemos destacar algumas versões de implementação como o Pentium Pro, o Pentium II e o Pentium III. Na versão Pentium Pro, o processador usa uma arquitetura de execução dinâmica. Neste processador, diferentes unidades de execução podem se agrupar no mesmo pipeline; por exemplo, a ALU e a unidade de ponto flutuante podem compartilhar

um pipeline. A versão Pentium II, por sua vez, incorpora a tecnologia Intel MMX, que é uma extensão do conjunto de instruções da arquitetura Intel. Esta tecnologia utiliza uma técnica SIMD para aumentar a velocidade processando dados em paralelo. A versão mais recente dos processadores Pentium é o Pentium III. Este processador caracteriza-se por possuir 70 instruções a mais que a versão anterior e mais de 9 milhões de transistores.

Devido à importância atual do conceito de paralelismo no nível de instruções, a arquitetura do Intel Pentium é descrita em outro capítulo.

5 O SUB-SISTEMA DE MEMÓRIA

Os capítulos anteriores discutiram conceitos a nível de arquitetura de processador. Com este conhecimento, podemos agora subir para o nível superior, o de arquitetura de computadores.

No Capítulo 1, a Figura 1.1 mostra a organização básica de um computador, incluindo o processador, a memória principal e as interfaces de entrada/saída, interconectados por um barramento. Na realidade, a memória principal é apenas um dos componentes dentro do sub-sistema de memória de um computador.

Este capítulo introduz os componentes normalmente encontrados no sub-sistema de memória de um microcomputador ou de uma estação de trabalho. Inicialmente, é analisado em detalhe o mecanismo de acesso do processador à memória principal. Em seguida, são apresentados os principais tipos e características de dispositivos de memória. Finalmente, são apresentados os conceitos de memória cache e memória virtual. Como exemplos reais, é descrita a memória cache e o suporte para memória virtual no processador Intel 80486.

5.1 A Interação entre Processador e Memória Principal

O componente básico da memória principal é chamado célula de bit. A célula de bit é um circuito eletrônico que armazena um bit de informação. Em uma referência à memória, o processador não realiza o acesso ao conteúdo de apenas uma célula de bit, mas sim a informação armazenada em um grupo de células de bits. O menor conjunto de células de bits que é acessado pelo processador é chamado locação de memória. Na maioria dos computadores, uma locação de memória é formada por 8 células de bits. Como 8 bits formam o chamado byte, uma locação de memória também é conhecida informalmente como “byte de memória”.

Em um acesso, o processador deve fornecer à memória principal uma identificação da locação onde será feito o acesso. Esta identificação é simplesmente um número, chamado endereço de memória. Ao receber um endereço de memória, a memória principal seleciona a locação correspondente e fornece ao processador a informação ali contida, no caso de um acesso de leitura. No caso de um acesso de escrita, a memória principal armazena na locação indicada pelo endereço a informação fornecida pelo processador.

A Figura 1.1 do Capítulo 1 mostra o processador conectado à memória principal através de um barramento. Na realidade, o processador e a memória principal estão interligados através de três barramentos distintos, conforme mostra a Figura 5.1.

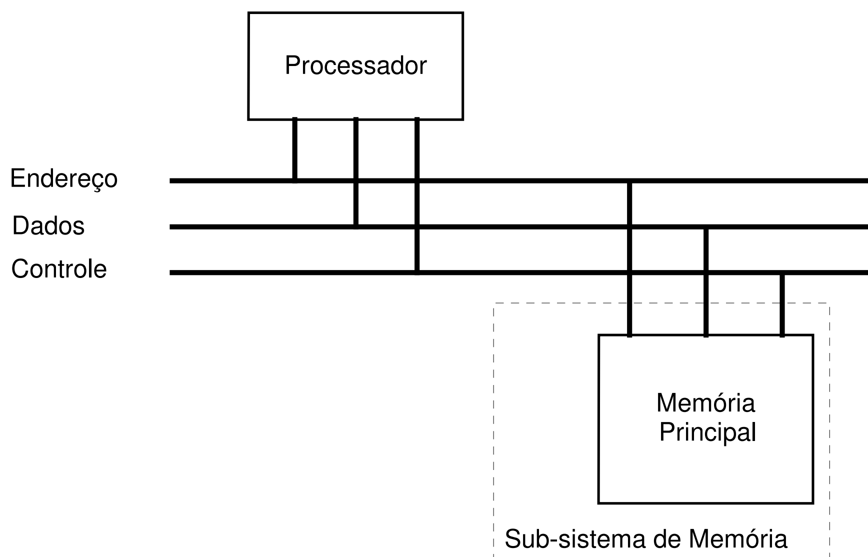


Figura 5.1. Interligação entre processador e memória principal.

O barramento de endereço é a via através da qual o processador transmite para a memória principal o endereço da locação onde será feito o acesso. O barramento de dados é a via através da qual o conteúdo da locação é transferida entre o processador e a memória principal. Finalmente, o barramento de controle é formado por diversos sinais através dos quais o processador controla o acesso à memória, indicando, por exemplo, se o acesso é de leitura ou de escrita

A largura, em número de bits, dos barramentos de endereço e de dados varia entre computadores, sendo determinada basicamente pelo processador usado no sistema. A Tabela 5.1 relaciona a largura dos barramentos de endereço e de dados nos diversos modelos da linha de microcomputadores do tipo IBM PC.

Sistema	Processador	Barramento de Endereço	Barramento de Dados
IBM PC XT	Intel 8088	20 bits	8 bits
IBM PC AT	Intel 80286	24 bits	16 bits
IBM PC 386	Intel 80386 DX	32 bits	32 bits
IBM PC 486	Intel 80486	32 bits	32 bits

Tabela 5.1. Largura dos barramentos em microcomputadores do tipo IBM PC.

Um sistema com um barramento de endereço com largura de n bits pode endereçar até 2^n locações de memória distintas. No entanto, isto não significa necessariamente que o sistema possui esta capacidade de memória principal instalada. Por exemplo, um sistema baseado no processador 80386 ou 80486 poderia, a princípio, ter uma memória principal de até 2^{32} = 4 Gbytes . No entanto, em geral microcomputadores possuem no máximo 16 Mbytes de memória principal. Alguns sistemas, como por exemplo estações de trabalho e servidores, permitem

até 2 Gbytes de memória principal. A capacidade máxima de memória principal que pode ser instalada em um sistema é determinada pelo compromisso entre custo e desempenho característico do sistema. Ainda neste capítulo será discutido o mecanismo de memória virtual, que permite a execução de programas que usam uma capacidade de memória maior do que a memória principal fisicamente existente no sistema.

5.1.1 Ciclo de Barramento

Como discutido acima, um acesso à memória envolve a seleção de uma locação de memória, a ativação de sinais de controle e a troca de informações entre o processador e a memória principal. A seqüência de eventos que acontecem durante um acesso do processador à memória principal é chamado de ciclo de barramento (bus cycle), e será agora descrito em maiores detalhes.

Para descrever um ciclo de barramento, é conveniente usar uma representação gráfica chamada diagrama de tempo. O diagrama de tempo indica as alterações no estado dos barramentos durante um acesso à memória. A Figura 5.3 mostra o diagrama de tempo do ciclo de barramento em um acesso de leitura na memória principal.

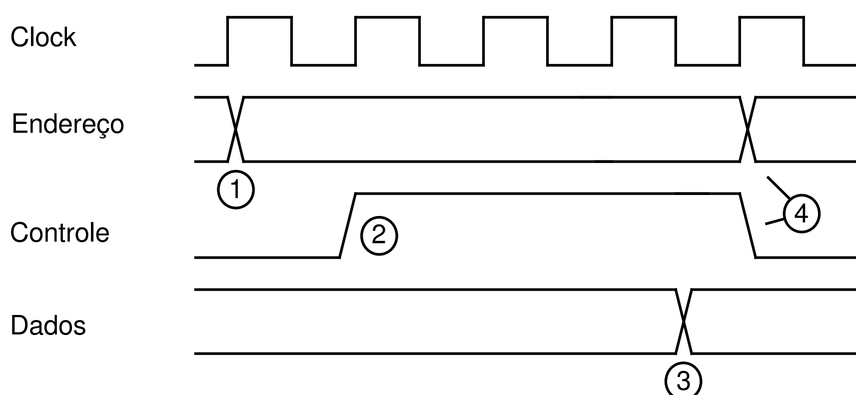


Figura 5.3. Ciclo de barramento em um acesso de leitura.

No diagrama acima, a representação indica que em um certo momento o barramento mudou de estado, com alguns bits mudando de valor 0 para 1 ou vice-versa. A representação indica que o sinal mudou do estado lógico 0 para o estado lógico 1, enquanto indica uma transição do sinal do estado 1 para 0. Como indica o diagrama de tempo, as mudanças nos estados dos barramentos estão sincronizados com o sinal de clock.

O processador inicia o ciclo de barramento colocando o endereço da locação de memória a ser acessada no barramento de endereço (1). No início do próximo ciclo de clock o processador ativa um sinal de controle indicando uma operação de leitura (2). O endereço e o sinal de controle chegam à memória, que após algum tempo coloca o conteúdo da locação de memória endereçada no barramento de dados (3). No início do quarto ciclo de clock, o processador captura a informação presente no barramento de dados e finaliza o ciclo de barramento, desativando o sinal de controle e retirando o endereço (4).

A Figura 5.4 mostra o diagrama de tempo do ciclo de barramento em um acesso de escrita na memória. Novamente, o ciclo de barramento inicia-se com o processador colocando o endereço da locação de memória no barramento de endereço (1). Além disso, o processador também coloca a informação a ser armazenada no barramento de dados (2). No próximo ciclo de clock, o processador ativa um sinal de controle indicando uma operação de escrita na memória (3). Ao receber o sinal de escrita, a memória armazena o dado na locação endereçada. O processador finaliza o ciclo de barramento retirando o endereço e o dado dos respectivos barramentos, e desativando o sinal de controle (4).

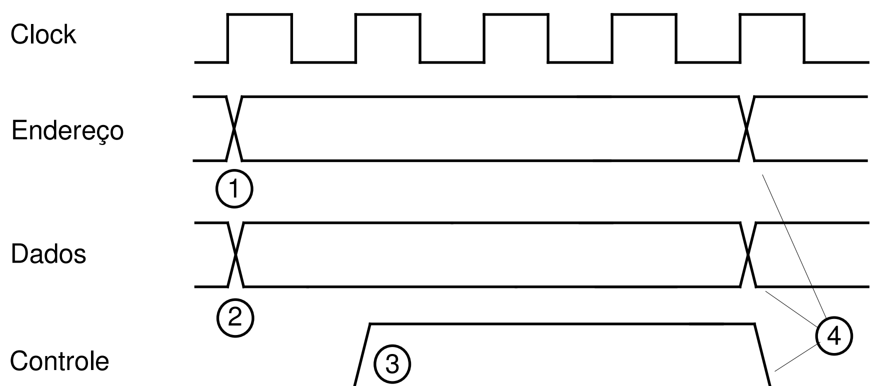


Figura 5.4. Ciclo de barramento em um acesso de escrita.

Nos diagramas mostrados, o número de ciclos de clock que separam os eventos ao longo de um ciclo de barramento são hipotéticos. O número de ciclos de clock consumidos no ciclo de barramento varia entre os processadores. No entanto, a seqüência de eventos mostrada nos diagramas de fato ocorrem para a maioria dos processadores.

5.1.2 Estados de Espera

A “velocidade” da memória principal é medida pelo tempo de acesso dos dispositivos de memória (circuitos integrados) usados em sua implementação. O tempo de acesso é o intervalo de tempo entre o instante em que a memória recebe o endereço até o instante em que a memória fornece a informação endereçada. Na Figura 5.3 é representado o caso onde o tempo de acesso da memória é equivalente a cerca de 3,5 ciclos de clock. Se o ciclo de clock é, digamos, 20 ns (1 ns = 10^{-9}s), então o tempo de acesso da memória é cerca de 70 ns. Atualmente, os dispositivos de memória instalados na memória principal de sistemas tais como microcomputadores e estações de trabalho apresentam um tempo de acesso entre 60 ns e 80 ns.

Um ponto importante a observar quanto ao ciclo de barramento é que o seu tamanho, em termos do número de ciclos de clock, é a princípio determinado pelo processador. Caso não haja uma solicitação externa quanto ao tamanho apropriado do ciclo de barramento, o processador sempre executará um ciclo de barramento padrão com um certo número de ciclos. Nos exemplos hipotéticos das Figuras 5.3 e 5.4, o ciclo de barramento padrão possui um tamanho de 4 ciclos de clock.

A questão que se coloca é: o que acontece se o processador possui um ciclo de barramento padrão cujo tamanho é menor que o tempo de acesso da memória principal? O diagrama de tempo na Figura 5.5(a) representa esta situação.

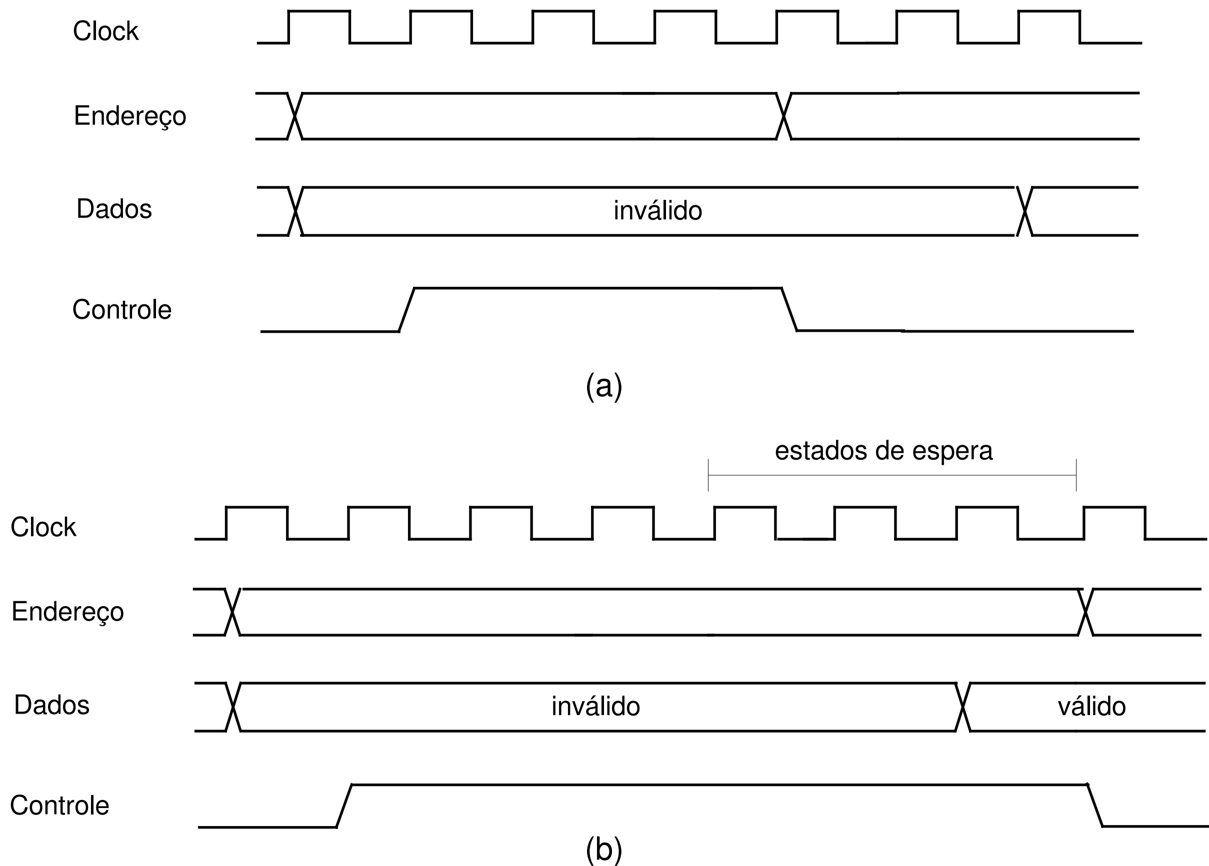


Figura 5.5. Os estados de espera no ciclo de barramento.

Na Figura 5.5(a) está representado o caso onde o ciclo de barramento padrão continua com 4 ciclos de clock, mas agora o processador se encontra em um sistema cuja memória principal possui um tempo de acesso de 6 ciclos de clock. Ao final do ciclo de barramento, o processador captura uma informação no barramento de dados que naquele momento é inválida, já que a memória forneceria a informação endereçada apenas dois ciclos de clock depois. Nesta situação, o sistema opera incorretamente.

Em geral, os processadores oferecem um mecanismo que permite compatibilizar o tamanho do ciclo de barramento ao tempo de acesso da memória. O processador possui um sinal de entrada que, ao ser ativado, faz com que o tamanho do ciclo de barramento seja estendido pela introdução de ciclos de clock extras. Estes ciclos de clock adicionais são denominados estados de espera.

A Figura 5.5(b) mostra um ciclo de barramento com estados de espera. Quando a memória detecta o início de um ciclo de barramento, a memória principal envia um sinal solicitando ao processador que o ciclo de barramento seja estendido. O processador não finaliza o ciclo de barramento enquanto está solicitação for mantida. Ao colocar a informação endereçada no barramento de dados, a memória

principal retira a solicitação de espera. O processador captura a informação, agora válida, presente no barramento de dados, e finaliza o ciclo de barramento. No exemplo mostrado, foram introduzidos três ciclos de espera.

Microcomputadores do tipo IBM PC oferecem este recurso dos estados de espera. O número de estados de espera é um dos parâmetros de configuração do sistema, e pode ser escolhido através da interface de configuração do sistema. Note que o uso deste recurso afeta o desempenho do sistema, já que o tempo consumido nos acessos à memória torna-se maior. No entanto, as memórias cache, que serão discutidas mais à frente, atenuam em grande parte o efeito do aumento do ciclo de barramento sobre o desempenho.

5.2 Tipos de Dispositivos de Memória

No sub-sistema de memória de um computador é possível encontrar três diferentes tipos de dispositivos de memória. O primeiro tipo é a memória programável somente de leitura, mais conhecida como PROM (Programmable Read Only Memory). Este tipo de dispositivo permite apenas acessos de leitura. As informações ali armazenadas são atribuídas pelo fabricante do sistema, e não podem ser modificadas após o seu armazenamento.

O segundo tipo é a memória programável e apagável somente de leitura, ou simplesmente EPROM (Erasable Programmable Read Only Memory). Assim como no caso da PROM, o processador realiza apenas acessos de leitura a uma EPROM. No entanto, ao contrário da PROM, o conteúdo de uma memória EPROM pode ser apagado, e a memória pode ser novamente usada para armazenar um novo conjunto de informações.

Memórias PROM ou EPROM são usadas para armazenar o firmware do computador. Em geral, o firmware é formado por sub-rotinas usadas pelo sistema operacional, e que interagem diretamente com o hardware do computador. O BIOS (Basic Input/Output System) em sistemas microcomputadores do tipo IBM PC é um exemplo de firmware. As rotinas que formam o BIOS são armazenadas em memórias do tipo EPROM.

O terceiro tipo de dispositivo de memória encontrado em um computador é a memória de leitura/escrita, ou RAM (Random Access Memory). Como a própria denominação indica, o processador pode efetuar acessos de leitura e escrita a um dispositivo RAM. Memórias deste tipo são usadas para armazenar as instruções e dados de um programa em execução. Existem dois tipos de memória RAM: as memórias RAM estáticas, ou SRAM (Static RAM) e as memórias RAM dinâmicas, ou DRAM (Dynamic RAM). Estes dois tipos de memórias RAM diferem quanto à capacidade de armazenamento e ao tempo de acesso.

Para memórias fabricadas com a mesma tecnologia, a capacidade de um dispositivo DRAM é até 16 vezes maior que a de um dispositivo SRAM. Em geral, a capacidade máxima de um dispositivo SRAM é de 256 Kbits, enquanto dispositivos de memória DRAM atingem uma capacidade de armazenamento de 4 Mbits. Esta diferença na capacidade de armazenamento deve-se ao modo como uma célula de

bit é implementada. Nas memórias DRAM, a célula de bit é implementada por um circuito eletrônico que ocupa uma área de integração menor que a ocupada pelo circuito usado nas memórias SRAM. Como a área por célula bit é menor, para a mesma área total de integração o número total de células de bit em uma DRAM é maior.

No entanto, as memórias SRAM apresentam uma vantagem sobre as memórias DRAM no que se refere ao tempo de acesso. Para a mesma tecnologia, o tempo de acesso de uma SRAM é até 4 vezes menor que o tempo de acesso de uma DRAM. Atualmente, existem memórias SRAM com tempo de acesso de 15 ns, enquanto as memórias dinâmicas mais rápidas apresentam um tempo de acesso de 60 ns. Melhorias no tempo de acesso dos dispositivos DRAM não acontecem na mesma taxa que o aumento observado na sua capacidade de armazenamento. Enquanto a tendência de aumento na capacidade de armazenamento dos dispositivos DRAM é de 60% ao ano, quadruplicando a cada três anos, a tendência observada indica uma diminuição do tempo de acesso de apenas 30% a cada dez anos.

Os dispositivos DRAM são utilizados em sistemas de baixo e médio custo tais como microcomputadores e estações de trabalho. Com este tipo de dispositivo é possível dotar um sistema com uma capacidade de memória principal elevada, sem aumentar significativamente o seu custo. No entanto, o uso de dispositivos DRAM favorece a capacidade de armazenamento da memória principal em detrimento do seu tempo de acesso, comprometendo o desempenho do sistema. Isto fica claro ao comparar-se a freqüência de clock de um processador atual com o tempo de acesso de uma DRAM. Um processador com freqüência de clock de 50 MHz executa uma operação aritmética ou lógica sobre operandos em registradores em 20 ns. Se um dos operandos encontra-se na memória, o tempo de execução desta instrução é pelo menos três vezes maior, tempo em vista o tempo de acesso da DRAM de 60 ns.

Novas classes de aplicações tais como processamento gráfico e multimídia estão fazendo com que o desempenho seja um fator cada vez mais importante na faixa de microcomputadores e estações de trabalho. A solução para obter o compromisso desejado entre capacidade de armazenamento, desempenho e custo encontra-se no uso apropriado de tecnologias de memória com diferentes relações entre estes três fatores. Esta é a idéia por detrás das memórias cache, cujo conceito e operação serão examinados a seguir.

5.3 Memórias Cache

A memória cache é uma pequena memória inserida entre o processador e a memória principal, conforme mostra a Figura 5.6.

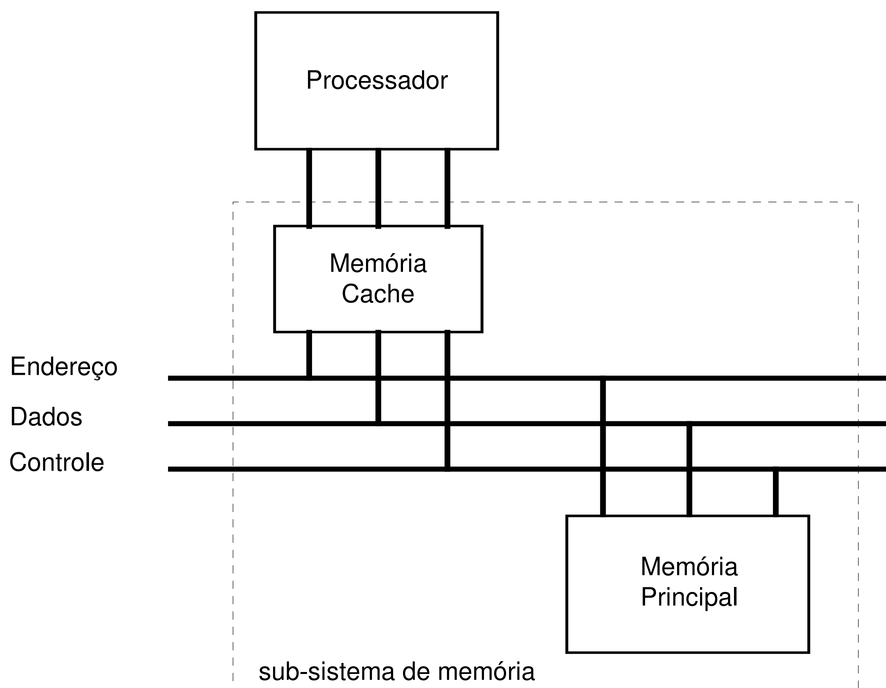


Figura 5.6. Localização da memória cache.

A memória cache é implementada com o mesmo tipo de circuito eletrônico usado nas células de bit de uma memória RAM estática. Por este motivo, a capacidade de armazenamento de uma memória cache situa-se entre 8 Kbytes e 64 Kbytes . No entanto, a memória cache apresenta um baixo tempo de acesso, possibilitando que o processador acesse dados ali armazenados em apenas um único ciclo de clock.

Nesta nova organização, o processador direciona os acessos inicialmente para a memória cache. Se o dado referenciado encontra-se na cache, o acesso é completado em apenas um ciclo de clock. Diz-se que ocorre um cache hit quando o dado referenciado pelo processador encontra-se na cache. Se o dado não se encontra na cache, diz-se que ocorreu um cache miss. Quando acontece um cache miss um bloco contendo o dado referenciado, e os dados armazenados em sua vizinhança, é copiado da memória principal para a memória cache. Após a transferência deste bloco de dados para a memória cache, o processador completa o acesso. Quando ocorre um cache miss, o acesso consome vários ciclos de clock para ser completado.

Na prática, observa-se que entre 90% e 98% dos acessos resultam em um cache hit. Isto acontece devido a uma propriedade exibida por vários tipos de programas, chamada propriedade da localidade de referência. Segundo esta propriedade, os acessos à memória que acontecem ao longo da execução de um programa não são uniformemente distribuídos através da memória, mas tendem a se concentrar em pequenas regiões da memória durante um certo intervalo de tempo. A Figura 5.7 representa a propriedade da localidade através de um gráfico.

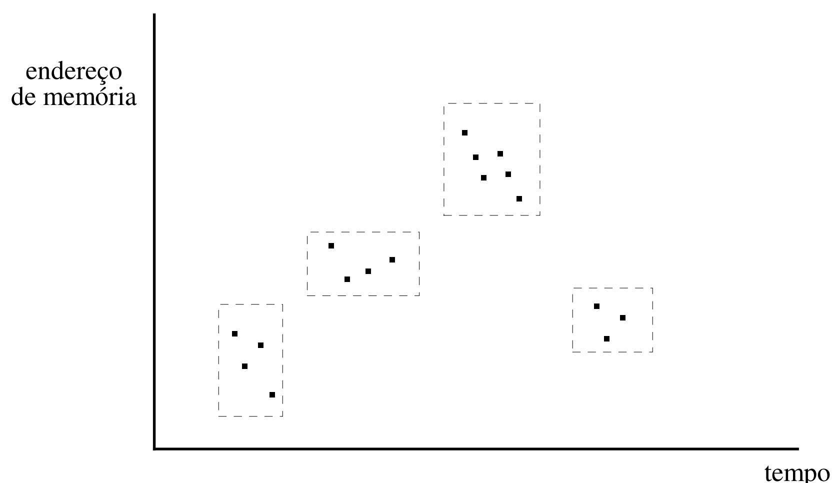


Figura 5.7. A localidade de referência.

Cada ponto no gráfico acima representa a ocorrência de um acesso à memória. No eixo horizontal está representado o instante em que ocorre o acesso, enquanto o eixo vertical corresponde ao endereço da locação de memória onde é feito o acesso. A localidade de referência manifesta-se no fato que, durante alguns intervalos de tempo, são feitos acessos a locações de memória em endereços próximos. Estas regiões onde os acessos se concentram durante um certo intervalo de tempo estão delimitadas na Figura 5.7.

O princípio da localidade sugere que quando é feito um acesso a uma locação de memória , existe uma grande probabilidade que acessos a locações em endereços vizinhos também sejam feitos no futuro próximo. Aplique agora esta propriedade ao funcionamento da memória cache. Como mencionado anteriormente, quando ocorre um cache miss, o dado referenciado é copiado da memória principal para a memória cache, juntamente com os dados armazenados nas locações vizinhas. Segundo a propriedade da localidade, existe uma grande chance de que os acessos aos dados vizinhos também sejam feitos em breve. Se estes outros acessos de fato acontecem, ocorrerá um cache hit em todos os acessos, que serão completados em um único ciclo de clock.

Na realidade, a memória cache armazena os dados que se encontram em regiões da memória principal onde se verifica uma concentração de acessos. Observe que a memória cache tem a capacidade de se adaptar a mudanças nas regiões de concentração de acesso, que ocorrem ao longo da execução de um programa. Uma nova região com concentração de acessos torna-se ativa quando é feito um acesso a um dado que se encontra distante dos dados onde foram feitos acessos recentemente. Como provavelmente este dado não se encontra na cache, ocorrerá um cache miss. Devido a este cache miss, serão copiados para a memória cache o dado referenciado e os dados na vizinhança, que pertencem justamente à nova região onde se concentrarão os próximos acessos, os quais serão novamente satisfeitos pela cache.

Com a inclusão de uma memória cache, apenas uma pequena parcela dos acessos realizados sofrem com o alto tempo de acesso da memória principal. Assim, as memórias cache são extremamente importantes no sentido de obter-se

um melhor desempenho. Atualmente, a maioria das arquiteturas de computador incluem memórias cache em seu sub-sistema de memória.

5.3.1 A Memória Cache no Intel 80486

Alguns processadores, como por exemplo o Intel 80486, já incluem uma memória cache em sua arquitetura. A memória cache no 80486 será aqui usada para mostrar em maiores detalhes a organização e operação de uma memória cache típica. A Figura 5.8 mostra a organização da memória cache no Intel 80486. No 80486, a memória cache tem uma capacidade de 8 Kbytes, e armazena instruções e dados. Outros processadores possuem memórias cache separadas, uma para instruções e outra para dados.

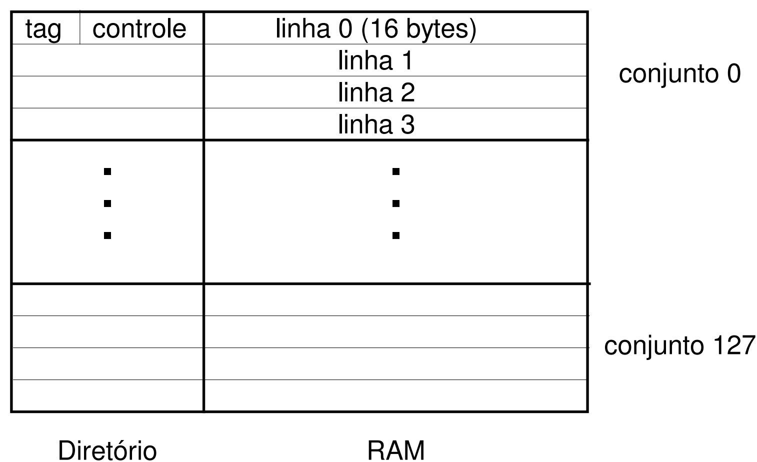
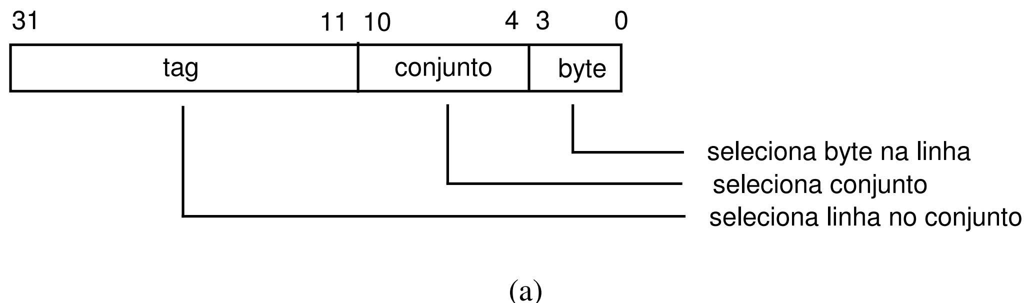


Figura 5.8. Organização da memória cache no Intel 80486.

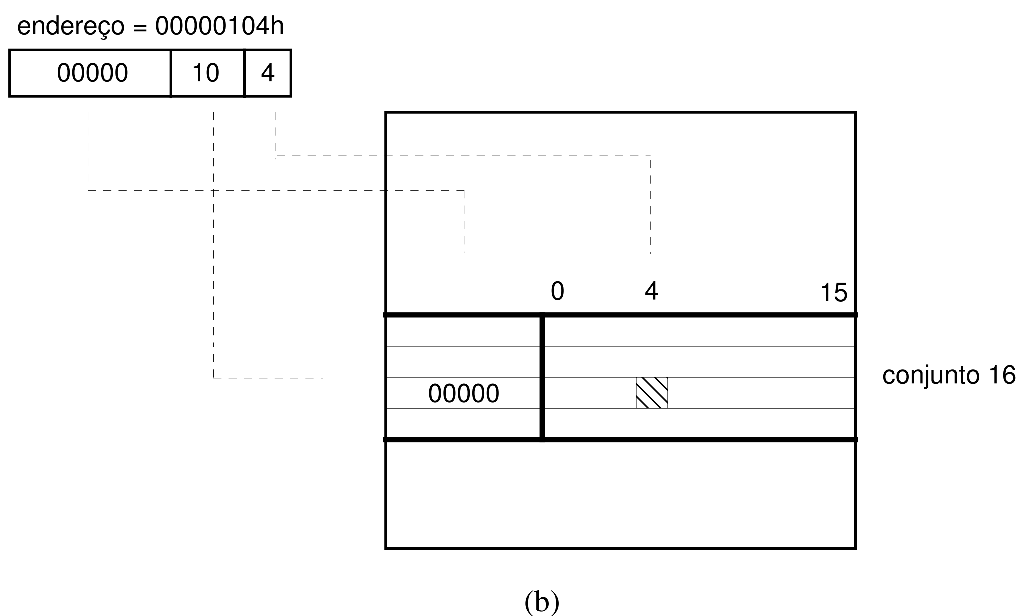
A memória cache é formada por uma memória RAM e pelo seu diretório. A memória RAM é organizada em linhas, sendo que em cada linha é armazenado um bloco de dados proveniente da memória principal. Cada linha na memória cache do 80486 armazena um bloco de 16 bytes . Para cada linha, existe uma entrada correspondente no diretório. Cada entrada de diretório contém um tag e bits de controle, cujas funções serão descritas logo à frente. Um grupo de 4 linhas com suas respectivas entradas no diretório formam um conjunto. A memória cache no 80486 possui 128 conjuntos. Esta organização em linhas formando um conjunto é característica das chamadas memórias cache associativas por conjunto (set associative cache memory). Uma memória cache associativa por conjunto com n linhas em cada conjunto é chamada cache n-associativa por conjunto (n-way set associative cache).

Existem outros tipos de memórias cache com organização diferente da mostrada na Figura 5.8, mas que não serão discutidas neste texto. No entanto, memória cache associativas por conjunto são as encontradas na maioria dos processadores. É importante ressaltar que o tamanho de linha, o número de linhas por conjunto e o número de conjuntos acima mencionados é específico da memória cache do Intel 80486. Memórias cache associativas por conjunto em outros processadores podem apresentar configurações diferentes.

Vamos agora examinar como é feito um acesso à memória cache no Intel 80486, e, em geral, a uma memória cache associativa por conjunto. Ao receber o endereço de uma locação de memória, a memória cache interpreta este endereço como mostrado na Figura 5.9(a).



(a)



(b)

Figura 5.9. Modo de acesso à memória cache no Intel 80486.

O endereço é logicamente dividido em três campos: o campo byte, formado pelos quatro bits menos significativos; o campo conjunto, formado pelos 7 bits seguintes; e o campo tag, composto pelos 21 bits mais significativos do endereço. A memória cache usa os bits no campo byte para selecionar um byte específico dentro de uma linha. O campo conjunto é usado para selecionar um dos conjuntos, enquanto o campo tag é usado para verificar se o dado referenciado se encontra em alguma das linhas do conjunto selecionado. Note que o tamanho de cada campo está relacionado com a configuração da memória cache, e assim pode variar entre as memórias cache de diferentes processadores. O número de bits por campo mostrado na Figura 5.9(a) é particular para o caso do 80486.

A Figura 5.9(b) mostra como o acesso de fato acontece. Como exemplo, suponha um acesso ao byte armazenado na locação de memória de endereço 10000104h (o sufixo h indica base hexadecimal). A memória cache interpreta este endereço da seguinte forma: byte = 4h, conjunto = 10h e tag = 10000h. A memória cache seleciona então o conjunto 16 (10h = 16), e em seguida compara o tag no endereço recebido (10000h) com os tags armazenados nas entradas de diretório do

conjunto selecionado. Se o tag no endereço coincide com algum dos tags no diretório (como é o caso na figura), isto significa que o bloco com o byte referenciado encontra-se na linha associada à entrada do diretório que contém o tag coincidente. Esta linha é então selecionada e o byte 4 dentro desta linha é finalmente acessado.

A não coincidência do tag do endereço com os tags armazenados no conjunto indica que o byte referenciado não se encontra na cache (ou seja, ocorreu um cache miss). Neste caso, a lógica de controle da memória cache se encarrega de copiar o bloco apropriado a partir da memória principal. Este bloco será armazenado em uma das linhas do conjunto indicado pelo endereço (no exemplo, o conjunto 16). Uma vez que o bloco tenha sido carregado na memória cache, o acesso prossegue como descrito acima.

Quando ocorre um cache miss, pode acontecer que todas as linhas do conjunto selecionado já estejam ocupadas – na realidade, este é o caso que normalmente acontece. Nesta situação, a memória cache tem que substituir um dos blocos residentes no conjunto para dar lugar ao novo bloco. Se o bloco selecionado para substituição não foi modificado por um acesso de escrita, este bloco é simplesmente descartado. Se o bloco selecionado foi modificado por um acesso de escrita anterior, este bloco é copiado de volta para a memória principal e então descartado. Um bit de controle na entrada de diretório indica se o bloco armazenado na linha correspondente foi ou não modificado desde o seu carregamento na memória cache.

A questão central na substituição acima descrita é a escolha do bloco a ser descartado.

A maneira mais simples seria escolher aleatoriamente qualquer um dos blocos do conjunto selecionado. A escolha aleatória é atrativa devido à sua simplicidade de implementação. No entanto, esta política aumenta a chance de ser escolhido um bloco que será referenciado em breve. Se isto de fato acontece, ocorrerá um cache miss porque o bloco não mais se encontrará na memória cache. Em um caso extremo, pode acontecer uma triste coincidência onde sempre é escolhido um bloco que será referenciado logo em seguida, levando a um círculo vicioso: um novo bloco expulsa um outro bloco que será referenciado em breve, o que vai causar um futuro cache miss; devido a este cache miss, é carregado um bloco que expulsa outro bloco que seria referenciado logo em seguida, causando um outro cache miss, e assim por diante. Em outras palavras, com a escolha aleatória existe o risco de um aumento excessivo no número de cache misses.

Em geral, é adotada uma política de escolha mais segura, denominada LRU (Least Recently Used). A política LRU diz que o bloco a ser substituído é aquele que não é referenciado há mais tempo. Este critério de escolha se baseia no seguinte raciocínio. Pelo princípio da localidade, quando um bloco é referenciado, existe uma grande chance de que ele seja novamente referenciado em breve. Se um bloco não é referenciado já há algum tempo, isto pode indicar que as referências concentradas naquele bloco já aconteceram e que o bloco não mais será referenciado, pelo menos no futuro próximo. Assim, este bloco pode ser substituído sem o risco de gerar cache misses imediatos. Bits de controle no diretório indicam

se um bloco foi ou não referenciado, sendo consultados quando um bloco deve ser substituído. Na prática, observa-se que a política LRU de fato não introduz um aumento significativo no número de cache misses. A memória cache no 80486 adota a política LRU.

5.3.2 Memórias Cache Primária e Secundária

Normalmente, a freqüência de cache hit aumenta à medida que o tamanho da memória cache aumenta. No entanto, observa-se que a partir de um certo ponto o número de cache hits não aumenta significativamente com o aumento da capacidade da memória cache. A partir deste ponto não compensa aumentar o tamanho da memória cache, pois isto acarretaria um aumento no custo do processador que não traria um benefício proporcional.

Para minimizar o efeito dos cache misses residuais, que podem chegar a até 10%, alguns sistemas incorporam duas memórias cache, uma denominada primária e a outra secundária, em uma estrutura conhecida como memória cache em dois níveis (two-level cache). A memória cache primária é integrada com o processador em um mesmo dispositivo, possui um tamanho pequeno, no máximo 64 Kbytes atualmente, e apresenta um tempo de acesso que permite acessos em um único ciclo de clock. Por sua vez, a cache secundária é externa ao processador, localizando-se na placa-mãe do sistema, possui um tamanho bem maior, atingindo até 512 Kbytes , e apresenta um tempo de acesso maior que a memória cache primária. Um acesso à cache secundária normalmente consome dois ou três ciclos de clock.

Em um sistema com esta organização, a memória cache primária captura a maioria dos acessos realizados pelo processador. Os acessos que resultam em miss são em sua maioria capturados pela memória cache secundária. Apesar de ser um pouco mais lenta que a memória cache primária, a memória cache secundária ainda é significativamente mais rápida que a memória principal. Assim, os misses da cache primária apresentam uma penalidade menor quando capturados pela cache secundária do que se fossem direcionados diretamente para a memória principal. Além disso, como a tecnologia usada na implementação da cache secundária é mais barata, o seu tamanho pode aumentar sem as mesmas limitações de custo da cache primária. Atualmente, memórias cache secundárias são usadas em diversas classes de sistemas, inclusive microcomputadores pessoais.

5.4 Memória Virtual

Como mencionado anteriormente, a memória principal disponível em um microcomputador ou em uma estação de trabalho é em geral bem menor do que o tamanho máximo de memória permitido pelo processador. O esquema de memória virtual foi originalmente criado para permitir a execução de programas cujas

exigências quanto ao tamanho da memória sejam maiores do que a capacidade de memória instalada no sistema.

5.4.1 O Conceito de Memória Virtual

Em um sistema sem memória virtual, o endereço gerado pelo programa em execução é o próprio endereço usado para acessar a memória principal. O mesmo não acontece em um sistema com memória virtual: o endereço gerado pelo programa, ou endereço virtual, é diferente do endereço real usado para acessar a memória principal. Os possíveis endereços virtuais que podem ser gerados pelo programa formam o espaço de endereçamento virtual, enquanto a faixa de endereços na memória principal constitue o espaço de endereçamento real.

Sob o ponto de vista de um programa, a memória disponível é aquela representada pelo espaço de endereçamento virtual. O espaço de endereçamento virtual visto e utilizado pelo programa pode ser bem maior do que o espaço de endereçamento real, efetivamente retirando do programa as limitações impostas pela capacidade da memória física de fato existente no sistema. É importante perceber que o espaço de endereçamento virtual, como o próprio nome indica, é uma abstração. Embora sob o ponto de vista do programa as instruções e dados estejam armazenados dentro do espaço de endereçamento virtual, na realidade eles continuam armazenados na memória principal, representada pelo espaço de endereçamento real.

Esta distinção entre endereços e espaços de endereçamento exige um mecanismo que faça a correspondência entre o endereço virtual gerado pelo programa e o endereço real que será usado para acessar a memória principal. Além disso, ela abre uma possibilidade que não ocorre em um sistema sem memória virtual: como o espaço de endereçamento virtual é maior que o espaço de endereçamento real, as instruções e dados que para o programa se encontram no espaço virtual podem, na realidade, não se encontrar presentes na memória principal no momento em que são referenciados. Assim, além do mapeamento acima mencionado, é necessário um mecanismo para o carregamento automático na memória principal das instruções e dados que são referenciados pelo programa dentro da sua memória virtual e que não se encontram presentes na memória física. Estes dois aspectos são discutidos em seguida.

5.4.2 O Mecanismo de Memória Virtual

O mapeamento entre endereços virtuais e reais é feito por um componente do sub-sistema de memória chamado tradutor dinâmico de endereços, ou DAT (Dynamic Address Translator). A Figura 5.9 mostra como o DAT se insere dentro do sub-sistema de memória.

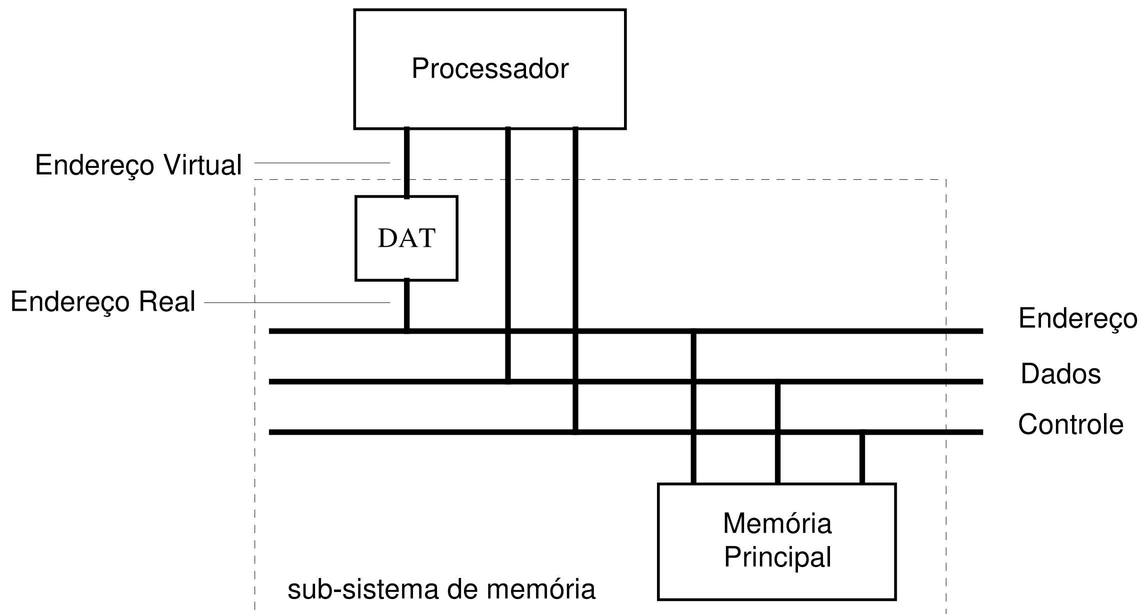


Figura 5.9. O tradutor dinâmico de endereços em um sistema com memória virtual.

O processador fornece ao DAT o endereço virtual gerado pelo programa em execução. O endereço virtual é mapeado em um endereço real pelo DAT e enviado para a memória principal através do barramento de endereço. Note que o mapeamento é feito em cada acesso à memória, no início do acesso.

Para realizar o mapeamento, o DAT utiliza uma tabela de mapeamento, localizada na memória principal. A tabela de mapeamento permanece na memória principal durante a execução do programa. Ao receber um endereço virtual, o DAT usa este endereço para indexar a tabela de mapeamento. A entrada indexada contém o endereço real correspondente ao endereço virtual. Na realidade, o mapeamento não é feito a nível de cada locação de memória, pois isto exigiria uma tabela de mapeamento com um número de entradas igual ao tamanho do espaço de endereçamento virtual. Para manter um tamanho de tabela aceitável, o mapeamento é feito a nível de blocos, como mostra a Figura 5.10(a).

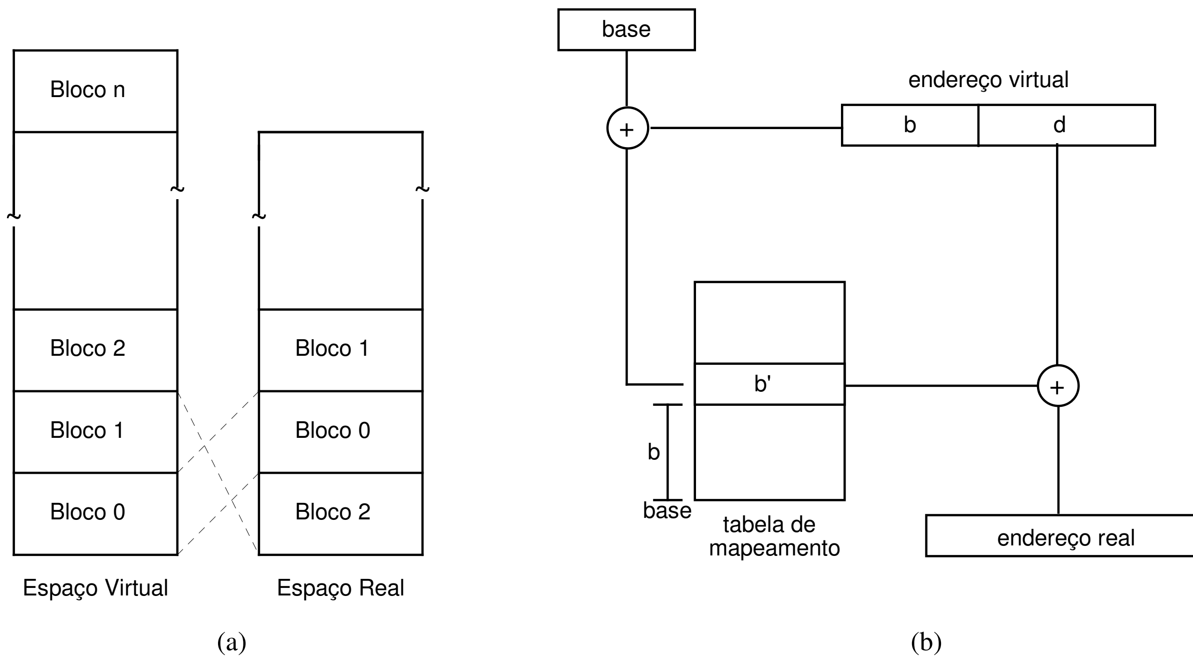


Figura 5.10. Mecanismo básico de mapeamento.

O espaço de endereçamento virtual é logicamente dividido em blocos, que são mapeados para o espaço de endereçamento real pelo DAT. Cada entrada na tabela de mapeamento contém o endereço-base de um bloco, ou seja, o endereço real a partir do qual o bloco está armazenado na memória principal. Em uma memória virtual segmentada, os blocos são chamados segmentos, e podem ter tamanho variável. Em uma memória virtual paginada, os blocos são chamados páginas e possuem tamanho fixo. Alguns processadores, como o Intel 80486, suportam estes dois modelos de memória virtual.

A Figura 5.10(b) mostra como é feito o mapeamento dos blocos. O DAT considera que o endereço virtual recebido do processador é constituído por dois campos. O campo b, formado pelos bits mais significativos do endereço virtual, indica o número do bloco onde se encontra a locação de memória referenciada. O campo d, formado pelos bits menos significativos, indica o deslocamento da locação de memória (isto é, a sua posição) em relação ao início do bloco.

O DAT usa o valor do campo b para realizar o acesso a uma entrada da tabela de mapeamento. Para tanto, o DAT soma o valor do campo b ao endereço-base da tabela de mapeamento. O endereço-base da tabela é mantido internamente no próprio DAT, em um registrador chamado TBR (Table Base Register). A soma resultante fornece o endereço de uma entrada da tabela e, com este endereço, o DAT acessa a tabela de mapeamento e obtém o endereço-base do bloco (indicado por b', na figura acima). Para obter finalmente o endereço real da locação de memória referenciada, o DAT soma o valor do campo d, que indica a posição da locação dentro do bloco, ao endereço-base do bloco.

Note que no mecanismo de mapeamento mostrado na Figura 5.10, para cada referência à memória realizada pelo programa é necessário um acesso adicional para consultar a tabela de mapeamento. Ou seja, neste esquema o número de acessos à memória principal durante a execução de um programa seria

duplicado, comprometendo seriamente o desempenho. Para solucionar este problema, o DAT possui internamente uma pequena memória, denominada TLB (Translation Lookaside Buffer). O TLB age como uma memória cache, armazenando os pares (b, b') que foram usados nos acessos mais recentes.

Uma operação de mapeamento com a TLB ocorre então da seguinte forma. O DAT usa o número do bloco no campo b para endereçar a TLB. Se acontece um TLB hit, o DAT obtém o endereço-base do bloco (b') a partir da TLB. Neste caso, o mapeamento não acrescenta nenhum retardo significativo ao acesso. Se acontece um TLB miss, o DAT consulta a tabela de mapeamento na memória para realizar o mapeamento. Além disso, o DAT armazena no TLB o par (b, b') usado neste mapeamento. Na prática, a maioria dos mapeamentos são satisfeitos pelo TLB. Isto acontece devido ao princípio da localidade, discutido anteriormente. Quando uma locação dentro de um bloco é referenciado, existe uma grande chance de que as locações vizinhas também sejam referenciados em breve. Se estas referências de fato ocorrerem, a informação de mapeamento usada nestas referências já estará na TLB, e cada mapeamento poderá ser completado rapidamente.

5.4.3 A Memória Secundária

Como mencionado anteriormente, devido à diferença de tamanho entre os espaços de endereçamento virtual e real, pode acontecer que um bloco referenciado pelo programa não esteja presente na memória principal no momento em que a referência acontece. Os blocos de instruções e dados de um programa em execução ficam armazenados na chamada memória secundária, que na realidade é uma unidade de disco do sistema.

A ausência de um bloco referenciado pelo programa é detectada pelo DAT no momento do mapeamento, e o bloco é então carregado da memória secundária para a memória principal. A Figura 5.11 mostra em detalhes como isto acontece.

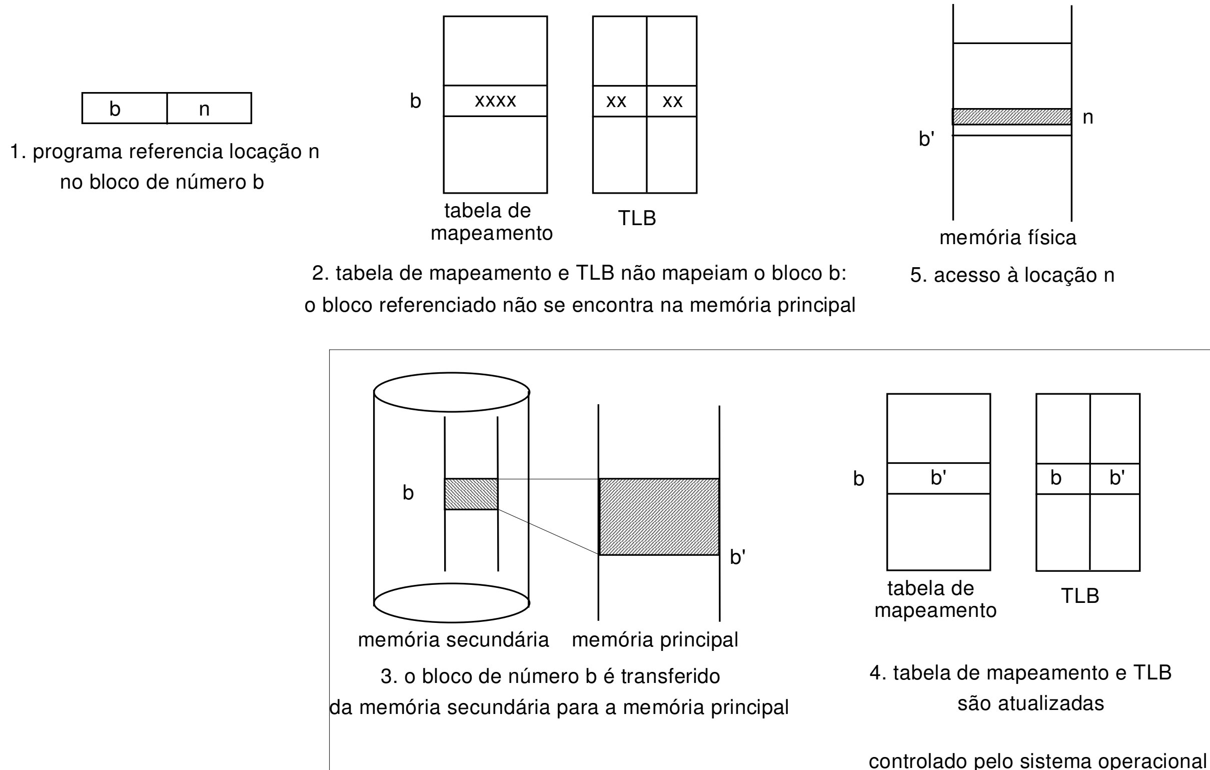


Figura 5.11. Carregamento de blocos na memória principal.

Neste exemplo, o programa referencia a locação n no bloco b. Ao receber o endereço virtual, o DAT tenta fazer o mapeamento através da TLB. Como a informação de mapeamento não se encontra na TLB, o DAT procura fazer o mapeamento através da tabela de mapeamento. Um bit de controle na entrada da tabela indica ao DAT que não existe um mapeamento válido para o bloco, significando que o bloco não se encontra na memória principal. A ausência de um bloco da memória é também denominada falha de página (no caso de memória virtual paginada) ou falha de segmento (para memória virtual segmentada). Ao detectar uma falha, o DAT gera uma interrupção (interrupções serão vistas no próximo capítulo), transferindo o controle do processador para o sistema operacional.

O sistema operacional obtém do DAT o número do bloco que foi referenciado. O sistema operacional localiza este bloco na memória secundária, aloca um espaço na memória principal e transfere o bloco da memória secundária para a memória principal. Em seguida, o sistema operacional atualiza a tabela de mapeamento e o TLB e devolve o controle para o programa que estava em execução. O controle é devolvido exatamente para o ponto onde se encontra o acesso que provocou o carregamento do bloco. O acesso é então re-executado pelo programa, sendo agora completado com sucesso porque o bloco referenciado já se encontra na memória principal.

Com a memória virtual, não é necessário que todas as instruções e dados de um programa permaneçam na memória principal durante a execução do programa. Blocos do programa são transferidos da memória secundária para a memória principal à medida que são referenciados. É importante observar que esta

transferência é totalmente transparente para o programa e, mais ainda, para o programador que desenvolveu o programa.

O mecanismo de memória virtual é em parte controlado por hardware e em parte por software. A consulta ao TLB e à tabela de mapeamento são realizadas pelo hardware, representado pelo DAT. O software, representado pelo sistema operacional, realiza transferências de blocos entre a memória secundária e a memória principal e as atualizações necessárias das estruturas de mapeamento. Além disso, o sistema operacional controla a ocupação da memória principal, localizando áreas livres onde um novo bloco pode ser armazenado, ou ainda determinando blocos que podem ser retirados da memória principal para dar lugar a novos blocos. A participação do sistema operacional dentro do mecanismo de memória virtual será brilhantemente discutida no curso subseqüente de sistemas operacionais.

5.4.4 Memória Virtual no Intel 80486

Alguns processadores oferecem suporte para memória virtual como parte integrante de sua arquitetura. Estes processadores possuem uma unidade, em geral denominada Memory Management Unit (MMU), que inclui todo o hardware de mapeamento discutido acima. O Intel 80486 é um exemplo de processador com suporte para memória virtual. O 80486 pode operar em dois modos. No modo real, o mecanismo de memória virtual é desativado, e o modelo de memória é idêntico ao do 8086: os programas visualizam uma memória segmentada, com um tamanho máximo de 1 Mbyte. No modo protegido, o mecanismo de memória virtual é ativado. Com a memória virtual, o tamanho do espaço de endereçamento virtual é 64 Tbytes (1 Tbyte = 2^{40} bytes), enquanto o tamanho do espaço de endereçamento real é de 4 Gbytes .

O 80486 suporta os modelos de memória virtual segmentada e paginada. O espaço de endereçamento virtual é organizado em segmentos com tamanho entre 1 byte e 4 Gbytes . Por sua vez, cada segmento pode ser sub-dividido em páginas com 4 Kbytes de tamanho. A paginação é opcional, e pode ser desativada através do software. A Figura 5.12 mostra o fluxo de endereços através destes dois níveis de memória virtual.

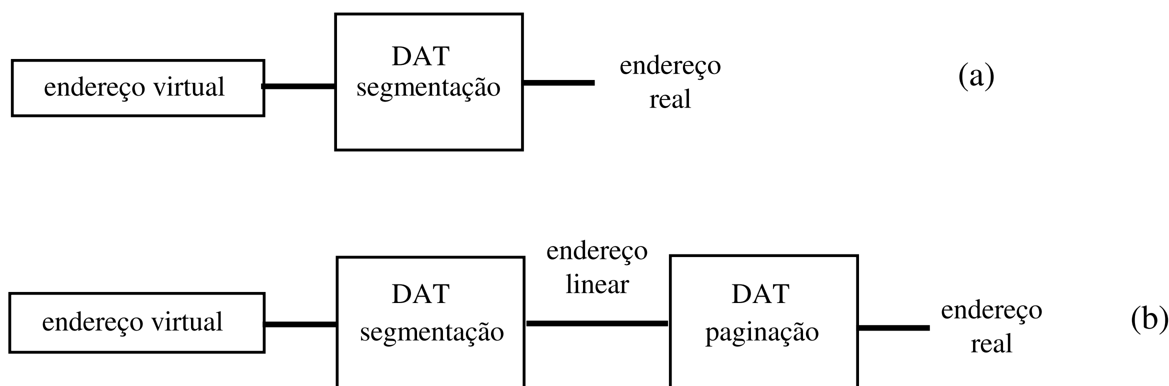


Figura 5.12. Níveis de segmentação e paginação na memória virtual do 80486.

A Figura 5.12(a) corresponde ao caso de memória virtual apenas segmentada. O DAT de segmentação recebe o endereço virtual e gera diretamente o endereço real. Como mostra a Figura 5.12(b), quando a paginação está ativada, o DAT de segmentação gera um endereço, chamado endereço linear, que é convertido pelo DAT de paginação no endereço real. Os níveis de segmentação e de paginação são descritos a seguir.

Vamos examinar inicialmente o nível de segmentação. Como mostra a Figura 5.13, o DAT de segmentação considera que os endereços virtuais são formados por um seletor de segmento (segment selector) de 14 bits e por um deslocamento (offset) de 32 bits.

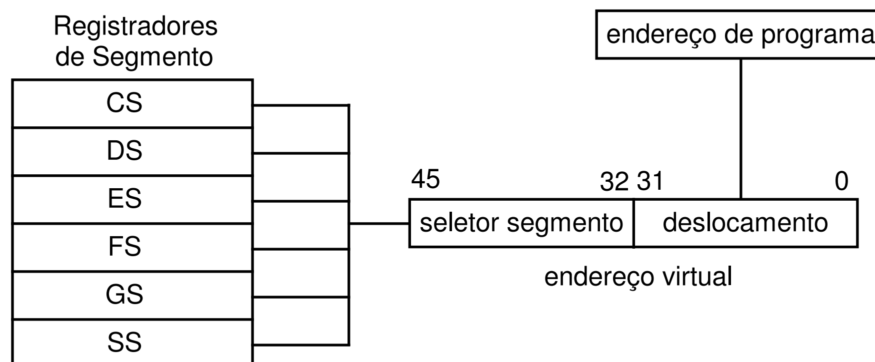


Figura 5.13. Composição do endereço virtual no 80486.

O seletor de segmento é dado pelos 14 bits mais significativos de um dos registradores de segmento. O seletor de segmento corresponde ao número de bloco na descrição apresentada acima. O deslocamento é o próprio endereço gerado pelo programa, e indica a posição do byte a ser acessado em relação ao início do segmento.

Os 13 bits mais significativos do seletor de segmento são usados para indexar uma tabela chamada tabela de descritores (descriptor table). A tabela de descritores desempenha o papel da tabela de mapeamento descrita anteriormente. Cada entrada da tabela contém um descritor de segmento que, dentre outras informações, indica o endereço-base do segmento na memória principal. A Figura 5.14 mostra como é feito o mapeamento dos segmentos.

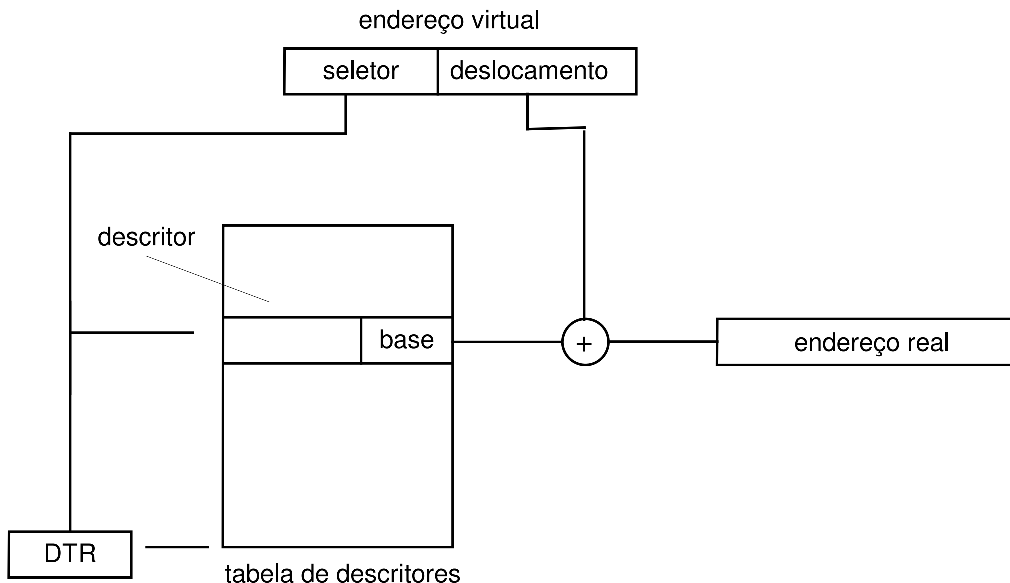


Figura 5.14. Mapeamento no nível de segmentação do 80486.

O endereço-base da tabela de descritores é armazenada no registrador DTR (Descriptor Table Register). O seletor de segmento é concatenado ao endereço em DTR, resultando no endereço de um descritor de segmento. O deslocamento é então somado ao endereço-base do segmento indicado pelo descritor de segmento, resultando no endereço real do byte referenciado.

Na realidade, o nível de segmentação opera com dois tipos de tabelas de descritores. Uma das tabelas é denominada, Global Descriptor Table, ou GDT. A GDT pode ser acessada por qualquer programa, permitido o compartilhamento de segmentos entre programas. A segunda tabela é chamada Local Descriptor Table, ou LDT. A LDT contém descritores de segmento privativos de um programa, podendo ser acessada apenas por aquele programa. Em um sistema multiprogramado, onde vários programas podem ser executados simultaneamente, cada programa possui sua própria LDT. O sistema operacional se encarrega de fazer a associação entre uma LDT e um programa. O endereço-base da GDT é armazenada no registrador GDTR (Global Descriptor Table Register), e o endereço-base da LDT do programa em execução é armazenada no registrador LDTR (Local Descriptor Table Register). O bit menos significativo do seletor de segmento indica se o mapeamento deve ser feito usando a GDT ou a LDT.

Na descrição do mecanismo de memória virtual apresentada acima, o TLB foi introduzido para reduzir o tempo de mapeamento. No 80486, os registradores de descritores (descriptor registers) desempenham uma função equivalente ao TLB. A organização dos registradores de descritores é mostrada na Figura 5.15.

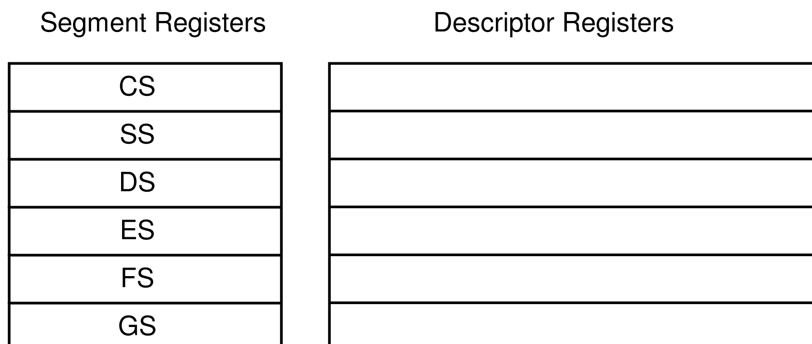


Figura 5.15. Registradores de descritores no DAT de segmentação do 80486.

Como mostra a figura, existe um registrador de descritor para cada um dos seis registradores de segmento. Quando um registrador de segmento é usado para indexar a tabela de descritores, o descritor selecionado é copiado para o registrador de descritor correspondente. Esta cópia do descritor é usada em futuros mapeamentos para o mesmo segmento, evitando assim o acesso à tabela de descritores. O registrador de descritor é automaticamente atualizado quando um novo seletor de segmento é armazenado no registrador de segmento correspondente.

Podemos agora examinar o nível de paginação no 80486. Neste nível, o mapeamento é feito em dois níveis. No primeiro nível existe uma tabela de diretório (directory table), que indica os endereços-base das tabelas no segundo nível. Cada tabela no segundo nível, chamada tabela de página (page table), contém os endereços-base das páginas na memória principal. Este esquema é mostrado na Figura 5.16.

O endereço linear recebido do DAT de segmentação é logicamente dividido em três campos. O campo diretório, com 10 bits, é usado para acessar uma entrada na tabela de diretório; o campo tabela, com 10 bits, é usado para acessar uma entrada em uma das tabelas de página; o campo deslocamento, com 12 bits, indica a posição do byte dentro da página.

Um registrador armazena a raiz da estrutura de mapeamento. A raiz é o endereço-base da tabela de diretório. No primeiro nível de mapeamento, o campo diretório é somado à raiz, resultando no endereço de uma entrada da tabela de diretório. A partir desta entrada da tabela de diretório é obtido o endereço-base de uma tabela de página, que será usado no segundo nível de mapeamento.

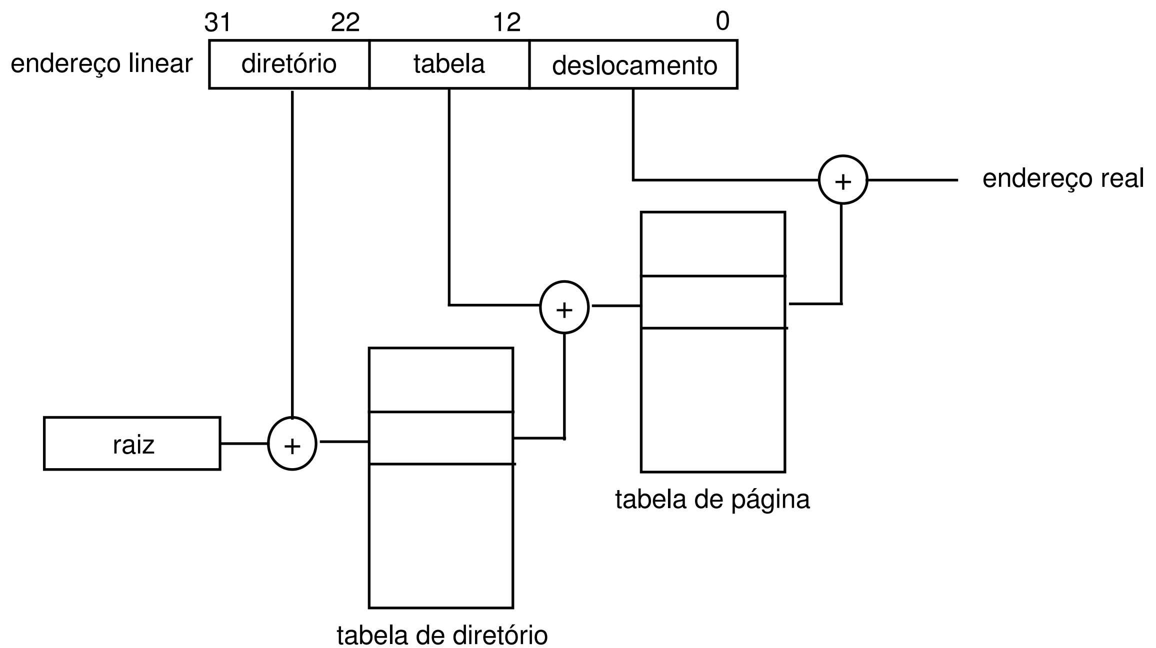


Figura 5.16. Mapeamento no nível de paginação do 80486.

No segundo nível de mapeamento, o campo tabela é somado ao endereço-base obtido no nível de mapeamento anterior. O resultado é o endereço de uma entrada aa tabela de página. A entrada selecionada indica o endereço-base da página na memória principal. O endereço real é finalmente obtido somando este endereço-base ao campo deslocamento. Para evitar o acesso a uma tabela de página a cada mapeamento, o DAT de paginação possui um TLB com 32 entradas. O DAT de paginação usa os 20 bits mais significativos do endereço linear (os campos diretório e tabela) para verificar se a informação de mapeamento se encontra na TLB. Caso ocorra um TLB hit, o TLB fornece o endereço-base da página, que é somado ao campo deslocamento para formar o endereço real. Se ocorre um TLB miss, o mapeamento ocorre como descrito acima. Além disso, o endereço-base da página usado no segundo nível do mapeamento é armazenado no TLB, para que possa ser usado em mapeamentos futuros.

6 O SUB-SISTEMA DE ENTRADA/SAÍDA

Depois do processador e do sub-sistema de memória, este capítulo examina o terceiro componente na arquitetura de um computador, o sub-sistema de entrada e saída (e/s). Neste sub-sistema estão incluídas as interfaces de e/s, através das quais os dispositivos periféricos são conectados ao sistema.

Este capítulo inicia descrevendo como processador e interfaces de e/s se comunicam, a organização típica de uma interface de e/s, e como o processador exerce controle sobre um dispositivo periférico através de uma interface de e/s. Em seguida são apresentadas as principais técnicas de transferência de dados em operações de e/s. Por último, são discutidos alguns aspectos relacionados com barramentos de e/s.

6.1 A Interação entre Processador e Interfaces de E/S

Em sistemas tais como microcomputadores e estações de trabalho, as interfaces de e/s são ligadas ao processador através de barramentos de endereço, dados e controle, de maneira semelhante à conexão entre memória principal e processador. A organização típica de um computador incluindo o sub-sistema de e/s é mostrada na Figura 6.1.

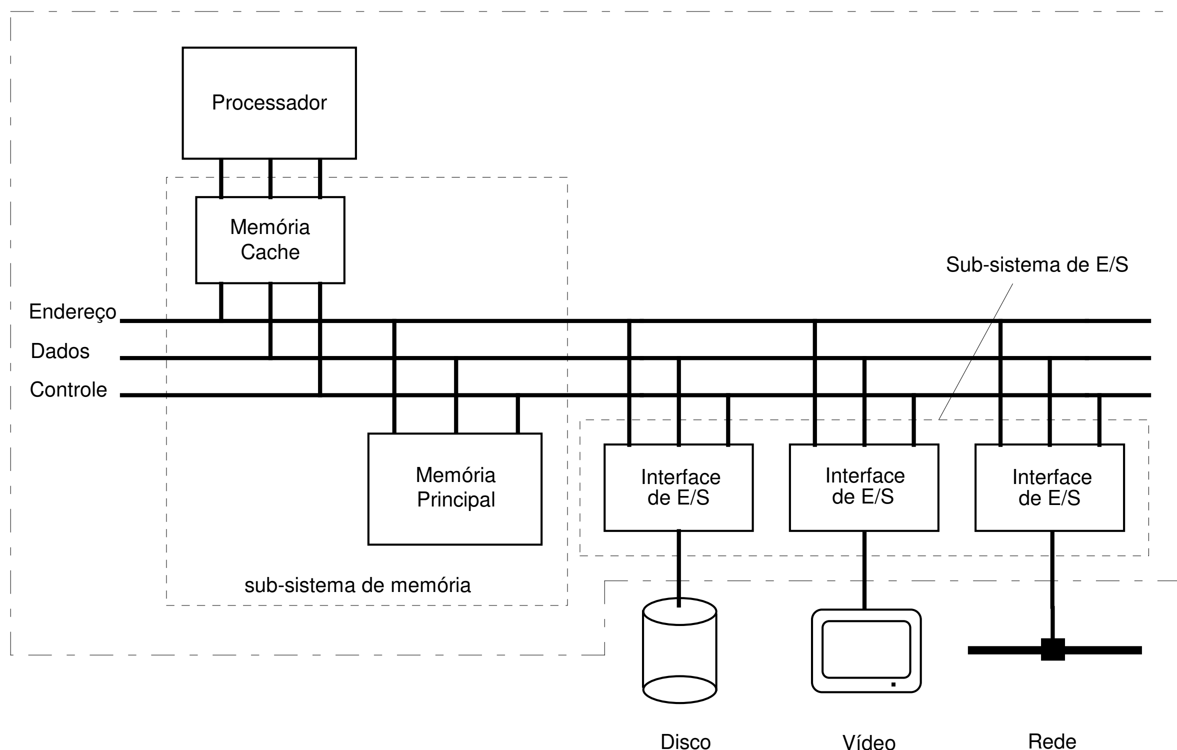


Figura 6.1. Arquitetura de um computador, incluindo o sub-sistema de e/s.

Assim como acontece em relação à memória principal, o processador realiza acessos de leitura ou de escrita a uma interface de e/s. Em um acesso de leitura, o processador obtém um dado recebido do dispositivo periférico conectado à

interface, ou então uma informação de estado sobre uma operação de e/s em andamento ou recém-completada. Em um acesso de escrita, o processador fornece à interface um dado que deve ser enviado ao dispositivo periférico, ou então o código de um comando que inicia uma operação de e/s ou uma operação de controle sobre o dispositivo periférico.

Nos acessos às interfaces, o processador executa ciclos de barramento semelhantes aos descritos no capítulo anterior. Cada interface de e/s é identificada por um endereço único. Em um acesso de leitura, o processador coloca o endereço da interface no barramento de endereço e ativa um sinal de leitura. Após um certo intervalo de tempo, a interface coloca a informação desejada no barramento de dados. O processador finaliza o ciclo de barramento lendo a informação presente no barramento de dados e retirando o endereço e o sinal de controle.

Em um acesso de escrita, o processador coloca o endereço da interface e o dado nos respectivos barramentos, e ativa um sinal de escrita. A interface selecionada armazena a informação presente no barramento de dados. No final do ciclo de barramento, o processador retira o endereço e o dado e desativa o sinal de controle. Assim como nos ciclos de barramento com a memória, todos estes eventos são comandados pelo processador e ocorrem em sincronismo com o sinal de clock.

6.2 Organização de uma Interface de E/S

A principal função de uma interface de e/s é tornar transparente para o processador os detalhes de operação e controle dos dispositivos periféricos. Podemos considerar que uma interface de e/s é organizada em duas partes, como mostra a Figura 6.2.

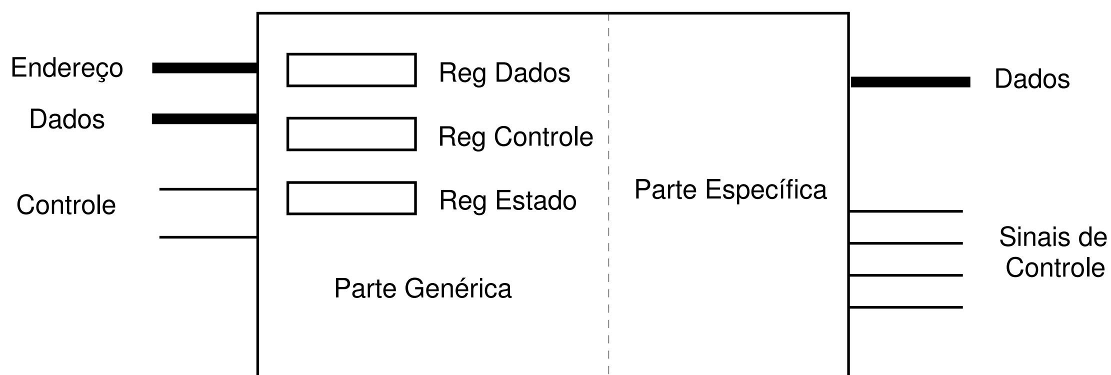


Figura 6.2. Organização típica de uma interface de e/s.

A parte genérica, como o próprio nome indica, é semelhante entre os diferentes tipos de interfaces de e/s. É esta porção da interface que é vista pelo processador. Em geral, na parte genérica existem alguns registradores, cujo número e função depende em parte do tipo de periférico acoplado à interface. No entanto, como mostra a figura acima, na maioria das interfaces a parte genérica inclui pelo menos um registrador de dados, um registrador de controle e um

registrador de estado. O acesso a cada um destes registradores é feito pelo processador através de um endereço de e/s diferente.

O registrador de dados é usado para as transferências de dados entre o processador e o dispositivo periférico. Em uma operação de saída, o processador escreve um dado neste registrador e a interface se encarrega de enviá-lo para o periférico. No sentido contrário, em uma operação de entrada, a interface recebe um dado do periférico e o armazena no registrador de dados. O processador executa então um acesso de leitura à interface o obtém o dado depositado no registrador.

O processador usa o registrador de controle para enviar comandos à interface. Este comando é enviado sob a forma de um código. Cada interface possui um repertório de comandos próprio. Quando o processador escreve um comando no registrador de controle, a interface interpreta o código do comando e executa a operação solicitada, que pode ser uma operação interna à interface ou sobre o periférico a ela conectado. Finalmente, o registrador de estado é usado para veicular informações gerais sobre uma operação de e/s. Tipicamente, este registrador possui bits para indicar o término de uma operação e para indicar condições de erro que eventualmente possam acontecer durante a operação.

A parte específica interage diretamente com o periférico, e por isso ela difere bastante entre os diferentes tipos de interfaces. No entanto, apesar das diferenças, a parte específica na maioria das interfaces possui dois conjuntos de sinais. Um deles é a própria via através da qual são transferidos os dados entre a interface e o periférico. O outro conjunto é formado pelos sinais usados no controle do periférico.

Como exemplo de interface de e/s, a Figura 6.3 mostra a organização simplificada de uma interface para unidades de disco rígido, a interface Intel 82064. Em sua parte genérica, esta interface possui sete registradores. O data register registrador é usado na transferência de dados. O command register equivale ao registrador de controle descrito anteriormente. Algumas operações exigem informações adicionais, que são escrita pelo processador nos registradores de parâmetro. O cylinder register é o registrador de parâmetro onde o processador escreve o número do cilindro (trilha) onde será feito o acesso. Os registradores sector number register e sector count register servem para indicar, respectivamente, o número do setor inicial e a quantidade de setores que devem ser acessados a partir do setor inicial. No sector/drive/head register o processador escreve o tamanho do setor em bytes, o número da unidade de disco e o número da cabeça. Finalmente, o error register indica alguma condição de erro ocorrida.

Na parte específica, a interface possui dois circuitos que realizam a leitura e a escrita de dados no disco. Um terceiro circuito controla a parte mecânica da unidade de disco. Por exemplo, este circuito gera os sinais STEP, que faz a cabeça avançar, e STEP DIR, que indica a direção de avanço da cabeça. Este circuito também recebe sinais, como o sinal READY, que indicam o estado da unidade de disco.

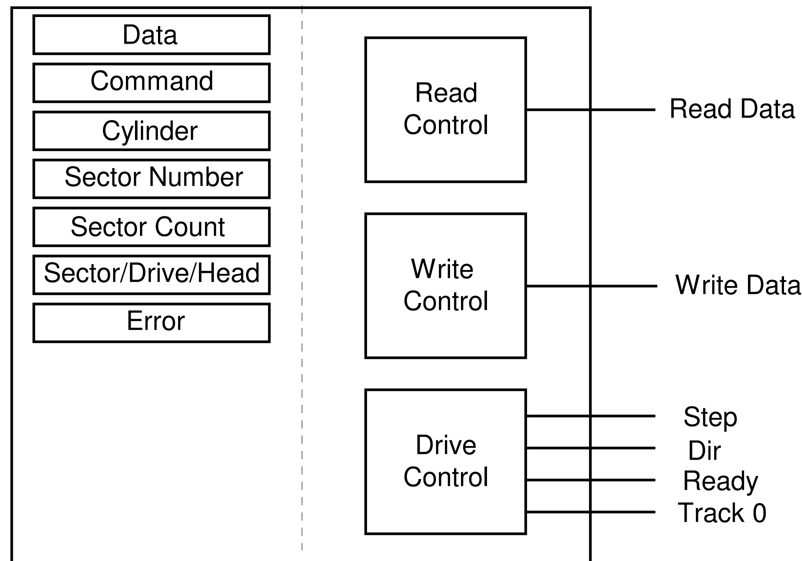


Figura 6.3. Organização simplificada de uma interface para unidades de disco rígido.

Este exemplo mostra a organização típica de uma interface de e/s. De um lado, a interface possui registradores através dos quais o processador envia e recebe dados, indica o tipo e os parâmetros da operação de e/s e obtém informações sobre o sucesso da operação. Do outro lado, a interface possui os circuitos e sinais necessários para controlar um particular periférico. Organização semelhante pode ser encontrada em interfaces para vídeo, impressoras, redes locais, entre outros.

6.3 Técnicas de Transferência de Dados

Em geral, uma operação de e/s envolve a transferência de dados entre a memória e a interface de e/s. Existem basicamente três técnicas de como realizar esta transferência, que são discutidas a seguir.

6.3.1 E/S com Polling

Na e/s com polling, o processador controla toda a transferência de dados entre a memória e a interface de e/s. Para entender como funciona esta técnica, considere o exemplo de uma operação de escrita em um setor de disco. Suponha que a interface controladora de disco é semelhante àquela mostrada na Figura 6.3. Normalmente, o registrador de estado possui um bit, chamado done bit, que é desativado quando um dado é escrito no registrador de dados, sendo ativado quando este dado é escrito no setor do disco. O diagrama na Figura 6.4 mostra como acontece a escrita de um setor de disco usando-se e/s com polling.

Figura 6.4. Exemplo de e/s com polling.

Após escrever um dado no registrador de dados, o processador lê o registrador de estado e testa o done bit, para verificar se o mesmo já foi escrito no setor do disco. Este teste do bit de estado é chamado polling. O processador continua realizando o polling até encontrar o done bit ativado, o que indica que o dado já foi escrito no setor do disco. Quando isto acontece, e se ainda existe algum dado a ser enviado, o processador escreve o novo dado no registrador de dados e reinicia o polling. Este ciclo é repetido até que todos os dados tenham sido escritos no setor do disco.

A principal vantagem da e/s com polling é a sua simplicidade. No entanto, esta técnica possui a desvantagem de que o processador fica dedicado à operação de e/s. Isto pode ser extremamente ineficiente, sob o ponto de vista da utilização do processador. Considere uma operação de envio de um bloco de caracteres para uma impressora. O tempo de impressão de um caracter é infinitamente maior que o tempo de execução de uma instrução. Manter o processador em polling durante o tempo de impressão de cada caracter é um desperdício, já que durante este intervalo de tempo o processador poderia executar alguns milhões de instruções de um outro programa. Devido ao fato que o processador fica dedicado à operação de e/s até o seu término, o uso da técnica de e/s com polling é restrita apenas a sistemas onde apenas um programa pode se encontrar em execução a cada instante.

6.3.2 E/S com Interrupção

Na e/s com polling, o processador fica dedicado à operação de e/s porque ele é o responsável por determinar quando um novo dado pode ser transferido entre a memória e a interface de e/s. O mesmo não acontece na e/s com interrupção. Nesta técnica, a interface é responsável por notificar o processador quando um novo dado pode ser transferido. Enquanto a e/s com polling é uma técnica puramente de software, a e/s com interrupção requer um suporte de hardware. A interface deve gerar um sinal de interrupção, através do qual ela notifica o processador quando uma operação de e/s foi concluída.

Considere novamente o exemplo da operação de escrita de um setor de disco. O diagrama na Figura 6.5 mostra como esta operação é realizada através de e/s com interrupção. A operação é dividida em duas fases. Na fase de disparo da operação, o processador envia para a interface o comando, o número da trilha e do setor. Ao final da fase de disparo, o processador passa a executar uma outra atividade qualquer, por exemplo, parte de um outro programa.

A interface inicia a fase de transferência de dados fazendo um pedido de interrupção ao processador, através do sinal de interrupção. Ao receber o pedido de interrupção, o processador suspende a execução do programa corrente e passa a executar uma rotina especial, chamada rotina de serviço de interrupção (também chamada device driver ou device handler). Nesta rotina, o processador verifica inicialmente se o último dado já foi enviado. Se este é o caso, o processador conclui a escrita do setor do disco lendo o registrador de estado da interface. Caso contrário, o processador envia um novo dado e retorna para o programa que se encontrava em execução.

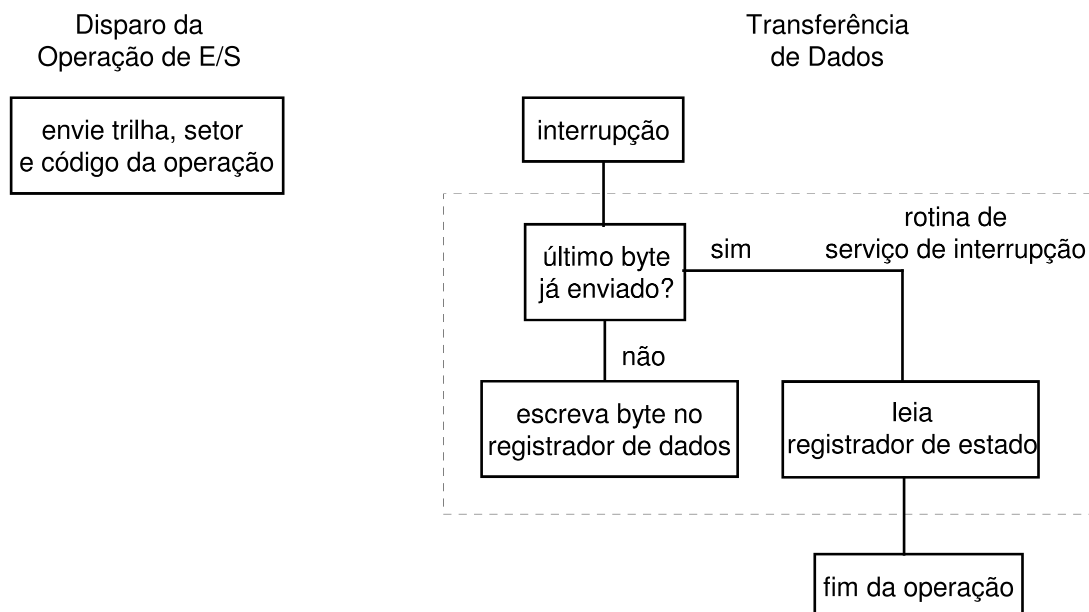


Figura 6.5. Exemplo de e/s com interrupção.

Durante a fase de transferência de dados, a interface faz um pedido de interrupção a cada dado escrito no setor do disco. O processador responde ao

pedido de interrupção executando a rotina de serviço e enviando um novo dado. Isto se repete até que todos os dados tenham sido escritos no setor do disco. Normalmente, a interface de disco conhece o tamanho do setor e mantém uma contagem dos dados já recebidos, de forma que ela pode determinar quando deve encerrar a seqüência de pedidos de interrupção.

Em um sistema é comum existirem várias interfaces diferentes que fazem pedidos de interrupção ao processador. Cada interface deve ser atendida por uma rotina de serviço de interrupção específica para aquela interface. Assim, ao receber um pedido de interrupção, o processador deve determinar qual a rotina de serviço a ser executada. Além disso, quando duas ou mais interfaces fazem pedidos de interrupção simultâneos, é necessário decidir qual o pedido de interrupção que será atendido. Estas duas funções são suportadas por um componente do sub-sistema de e/s, chamado controlador de interrupção (interrupt controller) A Figura 6.6 ilustra o funcionamento do controlador de interrupção em um sistema baseado nos processadores da família Intel 80x86.

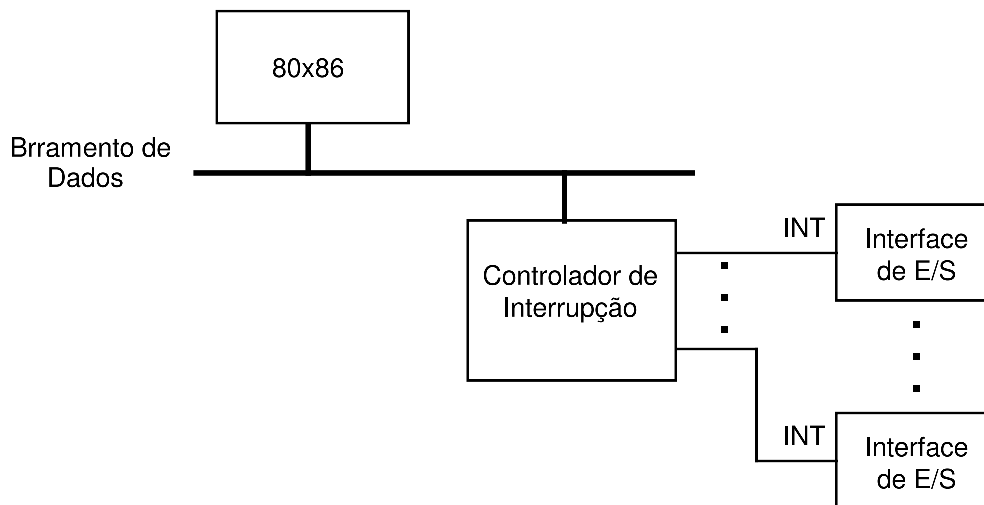


Figura 6.6. O controlador de interrupção.

Como mostra a figura, o sinal de interrupção de cada interface é ligado ao controlador de interrupção. O controlador de interrupção atribui um número e uma prioridade a cada um destes sinais. Quando um pedido de interrupção acontece, o controlador de interrupção envia para o processador, através do barramento de dados, o número do pedido. No caso de dois ou mais pedidos simultâneos, o controlador decide qual é o pedido com maior prioridade e envia para o processador o número correspondente.

O processador usa o número recebido do controlador para indexar uma tabela armazenada na memória, chamada tabela de vetores de interrupção (interrupt vector table). Cada entrada desta tabela contém o ponteiro, ou vetor, para uma rotina de serviço. Ao receber um número de interrupção n, o processador lê o vetor contido na posição n da tabela e passa a executar a rotina de serviço de interrupção apontada por este vetor.

Na e/s com interrupção, o processador não fica dedicado à operação de e/s. O processador é alocado somente quando realmente deve ser transferido um dado entre a memória e a interface, resultando em uma utilização mais eficiente do processador. No entanto, esta técnica apresenta uma desvantagem quanto à velocidade de transferência dos dados. Note que a transferência de um dado envolve a arbitração pelo controlador de interrupção, a comunicação entre o controlador e o processador, o acesso à memória para a leitura do vetor de interrupção e finalmente o desvio para a rotina de serviço. Todas estas etapas acrescentam um retardo antes que o dado seja realmente transferido. Este retardo é chamado de tempo de latência de interrupção (interrupt latency time).

Em alguns tipos de periféricos, a taxa de transferência de dados entre o periférico e a interface é muito alta, ou em outras palavras, o intervalo de tempo entre a transferência de dois dados consecutivos entre o periférico e a interface é muito pequeno. Devido ao tempo de latência, o intervalo de tempo entre acessos do processador à interface pode tornar-se maior que o intervalo de tempo com que os dados chegam à interface. Se isto acontece, um novo dado chega à interface antes que o processador leia o dado anterior, e assim o dado anterior é perdido.

Na realidade, o que contribui para aumentar o tempo de latência é o fato de que o processador ainda é o responsável por controlar a transferência de dados. Para atender periféricos com alta taxa de transferência, usa-se a técnica de e/s com acesso direto à memória, onde o processador não toma parte na fase de transferência de dados. Esta técnica é analisada a seguir.

6.3.3 E/S com Acesso Direto à Memória

Na e/s com DMA (Direct Memory Access), um componente do sub-sistema de e/s chamado controlador de DMA é responsável por transferir os dados entre a memória e a interface de e/s. A Figura 6.7 mostra como o controlador de DMA é ligado ao resto do sistema. Os sinais mostrados nesta figura são aqueles encontrados em sistemas baseados em processadores da família Intel 80x86.

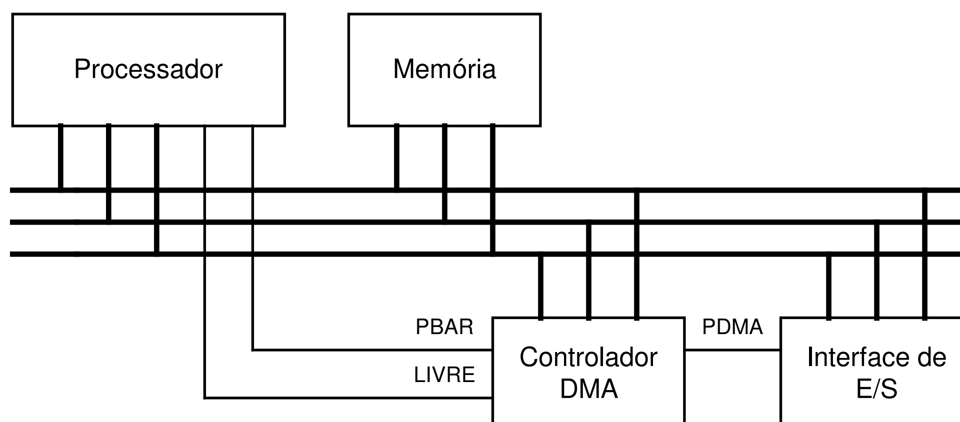


Figura 6.7. Sistema com controlador de DMA.

Considere novamente o exemplo da operação de escrita de um setor de disco. Na fase de disparo da operação, o processador informa ao controlador de

DMA o número de dados a serem transferidos, o endereço do primeiro dado e o sentido da transferência (no caso do exemplo, o sentido de transferência é da memória para a interface de e/s). Em seguida, o processador envia para a interface controladora de disco o número de trilha, o número de setor e o comando da operação.

O processador participa apenas da fase de disparo. Na fase de transferência de dados, o controlador de DMA assume o controle dos barramentos para realizar a transferência entre a memória e a interface. Para tanto, o controlador de DMA coloca o processador em um estado, chamado hold state, no qual o processador fica impedido de iniciar ciclos de barramento. Mais detalhadamente, a fase de transferência de dados envolve os seguintes passos:

- após receber o comando do processador, a interface de disco faz um pedido de DMA ao controlador de DMA através do sinal PDMA. Por sua vez, o controlador faz um pedido de barramento ao processador, através do sinal PBAR. Ao liberar os barramentos, o processador responde ativando o sinal LIVRE, indicando ao controlador de DMA que este já pode usar os barramentos.

- controlador de DMA coloca no barramento de dados o endereço do primeiro dado e ativa o sinal de leitura de memória. A memória responde colocando o dado endereçado no barramento de dados. O controlador de DMA ativa o sinal de escrita em interface de e/s, fazendo com que a interface de disco capture o dado presente no barramento de dados.

- ao escrever o dado no setor do disco, a interface faz um novo pedido de DMA. O controlador de DMA inicia uma nova transferência, colocando o endereço do próximo dado no barramento de endereço e ativando os sinais de controle apropriados. Este passo se repete até que todos os dados tenham sido transferidos. Ao concluir a última transferência, o controlador de DMA retira o pedido de barramento, permitindo que o processador volte à operação normal.

Note que na e/s com DMA a transferência de cada dado envolve apenas uma leitura de memória e uma escrita de interface de e/s, realizadas pelo próprio controlador de DMA. A e/s com DMA efetivamente elimina o tempo de latência associado a cada dado transferido, que existe na e/s com interrupção. Isto permite que a e/s com DMA atinja taxas de transferência bem maiores que as técnicas de e/s que envolvem o controle do processador.

Em geral, é possível ter várias interfaces de e/s operando com a técnica de acesso direto à memória. Para tanto, o controlador de DMA possui várias entradas para pedido de DMA. O controlador de DMA associa a cada uma destas entradas um conjunto independente de registradores para armazenar o número de dados a serem transferidos, o endereço inicial e o sentido da transferência. Um grupo de sinais de controle com seus respectivos registradores formam o chamado canal de DMA. O controlador de DMA se encarrega de arbitrar entre interfaces que fazem pedidos de DMA simultâneos, usando um esquema de prioridades atribuídas aos canais de DMA.

6.4 Padrões de Barramentos

A Figura 6.1 mostra a arquitetura de sistema típica de microcomputadores e algumas estações de trabalho, onde processador, memória principal e interfaces de e/s estão interligados através de um conjunto de três barramentos. Estes barramentos são chamados coletivamente de barramento de sistema. Algumas características do barramento de sistema, tais como largura do barramento de endereço e do barramento de dados, são determinadas pelo processador. Outras características estão relacionadas com o sub-sistema de e/s, como por exemplo, o número de sinais de interrupção e o número de canais de DMA disponíveis no barramento de controle.

Na categoria de microcomputadores, foram estabelecidos alguns padrões de barramento de sistema, com a finalidade de garantir a compatibilidade com interfaces de e/s de diferentes fabricantes. A Tabela 6.1 relaciona alguns padrões de barramentos de sistema em microcomputadores e suas principais características.

Padrão	Endereço	Dados	Interrupções	Canais DMA	Desempenho (E/S, DMA)
IBM PC XT	20 bits	8 bits	8	4	62.5 Kbytes/s
ISA	24 bits	16 bits	15	7	100 Kbytes/s
EISA	32 bits	32 bits	ilimitado	7	33 Mbytes/s
PCI	32 bits	32/64 bits	ilimitado		132/264 Mbytes/s

Tabela 6.1. Características de alguns padrões de barramentos de sistema.

O barramento de sistema no IBM PC XT era, na realidade, uma simples extensão dos barramentos do processador Intel 8088. O número de sinais de interrupção e de canais de DMA era bastante limitado. O IBM PC XT foi substituído pelo IBM PC AT, cujo barramento de sistema tornou-se o padrão ISA (Industry Standard Architecture). Com o ISA, o número de sinais de interrupção e de canais de DMA foi quase duplicado, enquanto que o desempenho das operações de e/s com DMA aumentou em 60%. Apesar da diferença na largura dos barramentos de endereço e de dados, o padrão ISA permite a conexão de interfaces originalmente projetadas para o barramento IBM PC XT.

Com o lançamento dos processadores Intel 80386 e 80486, aumentaram as exigências quanto ao desempenho de e/s. Para fazer face à estas exigências, foi criado o padrão ISA estendido, ou EISA (Extended Industry Standard Architecture). No padrão EISA, transferências de dados em operações de e/s são controladas por

um componente especial, denominado bus master. Na verdade, um bus master está contido em uma interface de e/s sofisticada, que normalmente possui uma lógica dedicada ao controle do barramento e por uma memória local. Em um barramento EISA podem existir ate 15 bus masters.

O bus master executa ciclos de barramento para transferir dados entre ele mesmo e a memória principal, ou ainda entre ele e uma interface de e/s. Como as transferências são sempre controladas pelo bus master, a memória ou a interface de e/s são denominados slaves. O bus master pode realizar dois tipos de transferências. No primeiro tipo, chamado standard transfer, o bus master e o slave executam um protocolo de sincronização a cada dado transferido. No modo burst transfer, este protocolo de sincronização é executado apenas uma vez, no início da transferência de um bloco de dados. Assim, transferências no modo burst transfer são mais rápidas que no modo standard transfer. Tipicamente, no modo standard transfer, são necessários 8 ciclos de clock para transferir um dado, enquanto no modo burst transfer a sincronização inicial consome 4 ciclos de clock e apenas 1 ciclo adicional para cada dado transferido. O modo burst transfer foi introduzido no padrão EISA para atender as necessidades de periféricos que transferem blocos de dados com uma alta taxa de transferência.

O padrão EISA também permite transferências usando as tradicionais técnicas de interrupção e de acesso direto à memória, mas com vantagens. Nos barramentos PC XT e ISA, existe um numero máximo de sinais para pedidos de interrupção disponíveis no barramento de controle, o que limita o número de interfaces que podem usar a técnica de interrupção. No barramento EISA, não existe limitação quanto ao número de interfaces que podem solicitar interrupções. Além disso, transferências via DMA são mais rápidas.

Um ponto importante no padrão EISA é a compatibilidade. Como o próprio nome sugere, o EISA foi criado como uma extensão de barramentos anteriores. Isto significa que interfaces para os barramentos PC XT e ISA podem ser usados em sistemas com barramento EISA.

Atualmente, os computadores são fabricados segundo o padrão PCI. Este padrão é uma interface de 64 bits num pacote de 32 bits. O barramento PCI roda em 33 MHz e pode transferir 32 bits de dados em cada ciclo de clock. Uma característica importante dos padrões de barramento mais recentes, como o EISA e o PCI, é o compartilhamento de interrupções. Esta técnica permite que dois dispositivos utilizem a mesma interrupção.

6.4.1 Barramentos Locais

Normalmente, um barramento de sistema possui conectores (os chamados slots) destinados à conexão das interfaces de e/s ao sistema. Em geral, o barramento de sistema inclui um número razoável de conectores, de forma a permitir eventuais expansões com o acréscimo de novas interfaces.

Devido ao grande número de conectores, o barramento torna-se fisicamente longo. Infelizmente, quanto maior o comprimento do barramento, maior será o efeito de alguns fatores elétricos indesejáveis que limitam a taxa de transferência de

dados. Esta limitação afeta não somente transferências de dados em operações de e/s, mas também os acessos à memória. Isto acontece porque um mesmo barramento é usado tanto para acessar a memória quanto para conectar as interfaces de e/s ao sistema. Para eliminar os efeitos negativos de um único barramento sobre os acessos à memória, alguns sistemas utilizam dois barramentos distintos, como mostra a Figura 6.9. Nesta organização, existe um barramento local que interliga o processador à memória principal e um barramento de e/s à parte, usado apenas para a conexão das interfaces de e/s. O barramento de e/s é isolado do barramento local através de um adaptador.

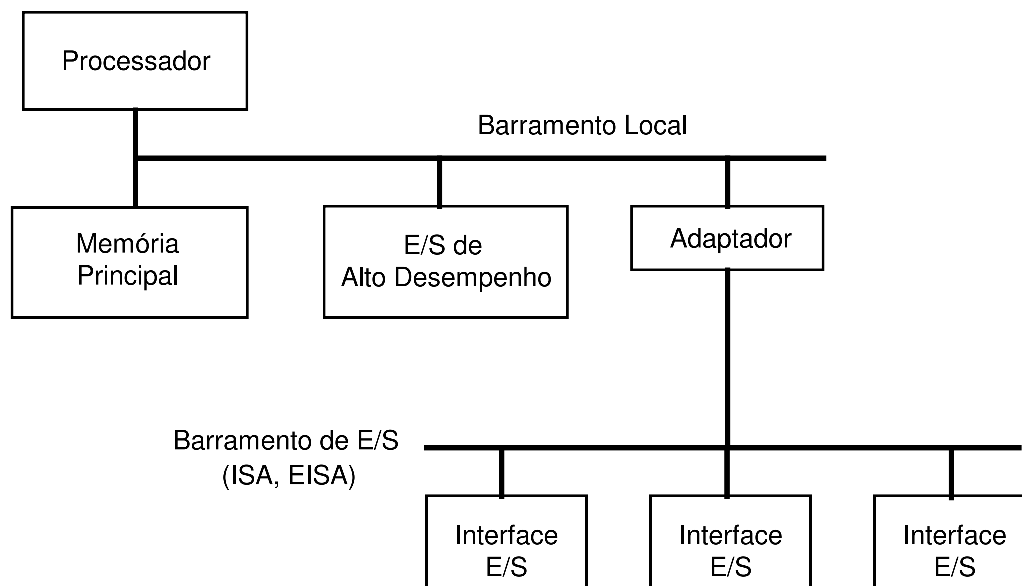


Figura 6.9. Estrutura de um sistema com dois barramentos.

Com barramentos diferentes, a comunicação entre processador e memória principal fica isolada dos efeitos negativos decorrentes do comprimento excessivo do barramento de e/s. O comprimento do barramento local é extremamente reduzido: a este barramento podem ser conectados apenas algumas poucas (no máximo duas ou três) interfaces de e/s de alto desempenho, como por exemplo, interfaces de vídeo. Atualmente, existem alguns padrões de barramentos locais estabelecidos. Os mais comuns são o VESA Local-Bus, desenvolvido pela Video Equipment Standards Association, e o Peripheral-Connect Interface, ou PCI, desenvolvido pela Intel Corp. Com o aumento do desempenho dos microprocessadores, existe uma tendência que arquiteturas com barramentos distintos venham a ser adotadas em uma faixa cada vez maior de sistemas.

6.4.2 Barramentos de Periféricos

Em alguns sistemas é possível encontrar um terceiro tipo de barramento, denominado barramento de periféricos. Este nível de barramento tem como principal objetivo prover um padrão de compatibilidade entre diferentes fabricantes de dispositivos periféricos. Em geral, o barramento de periférico é conectado ao barramento de e/s através de uma interface adaptadora especial.

Atualmente, o principal padrão de barramento de periféricos é o SCSI (Small Computer Systems Interface). A primeira versão do padrão SCSI foi lançada em 1986. A segunda versão, SCSI 2, foi lançada em 1990. O barramento SCSI possui uma estrutura em árvore, como mostra a Figura 6.10.

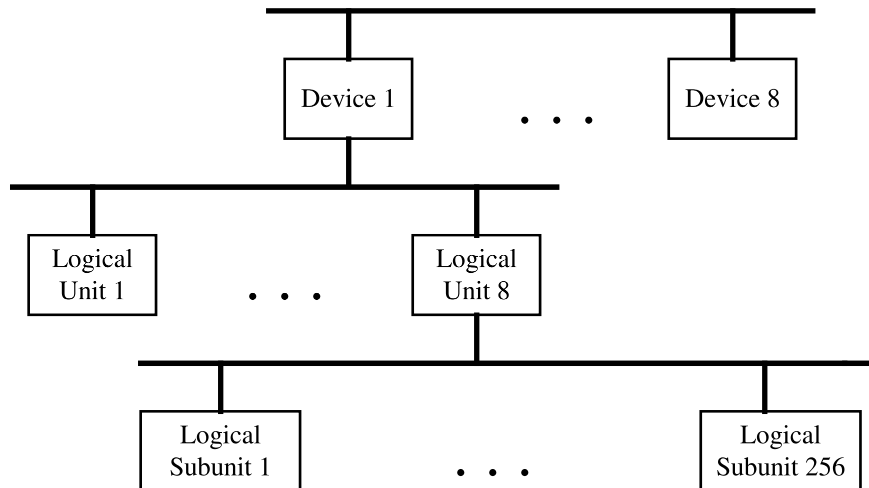


Figura 6.10. Estrutura do barramento de e/s no padrão SCSI.

No primeiro nível podem existir até 8 SCSI devices. Existem dois tipos de SCSI devices: o primeiro tipo é chamado host adapter, e é usado para conectar o barramento SCSI ao barramento de e/s do sistema; o segundo tipo é chamado peripheral controller. Cada controller pode ser uma interface de e/s ou um adaptador para o próximo nível do barramento.

A cada controller podem estar ligados até 8 logical units no nível abaixo do barramento. Cada logical unit pode ser um periférico de e/s ou um adaptador para o próximo nível do barramento. Finalmente, a cada logical unit podem estar conectados até 256 logical subunits no terceito nível do barramento. Cada logical subunit corresponde a um periférico de e/s.

O padrão SCSI 2 apresenta dois modos especiais de operação. O primeiro modo é chamado fast SCSI. Enquanto a velocidade no modo básico de operação é de 4 milhões de transações por segundo, o modo fast SCSI permite uma velocidade de 10 milhões de transações por segundo. No entanto este modo exige um hardware especial, que aumenta o custo do barramento. O outro modo de operação é chamado wide SCSI. Neste modo são usadas 32 linhas de dados, ao invés das 16 linhas usadas no modo básico. Os modos fast SCSI e wide SCSI podem ser usadas em conjunto, possibilitando uma taxa de transferência de 40 Mbytes/s.

7 TÓPICOS ESPECIAIS

Este capítulo discute alguns tópicos que atualmente representam o estado-da-arte na área de arquitetura de computadores. Inicialmente, são apresentadas a técnica de pipelining e as arquiteturas super-escalares, dois conceitos que se baseiam na execução paralela de instruções para aumentar o desempenho de um processador. Em seguida é introduzida a filosofia RISC, cujos princípios são adotados pela maioria dos processadores lançados recentemente. Finalmente, este capítulo discute sistemas multiprocessadores e supercomputadores voltados para processamento de alto desempenho.

7.1 A Técnica de Pipelining

No Capítulo 2 foi apresentado o mecanismo básico de execução de instruções, no qual as instruções são executadas seqüencialmente. Neste modo, a execução de uma nova instrução inicia-se somente quando a execução da instrução anterior é completada. Isto significa que apenas uma instrução encontra-se em execução a cada instante de tempo.

Ao contrário desta forma de execução seqüencial, a técnica de pipelining permite que várias instruções sejam executadas simultaneamente. Na técnica de pipelining, os passos de execução de uma instrução são realizados por unidades independentes, denominadas estágios do pipeline. A Figura 7.1(a) mostra a representação de um pipeline com quatro estágios.

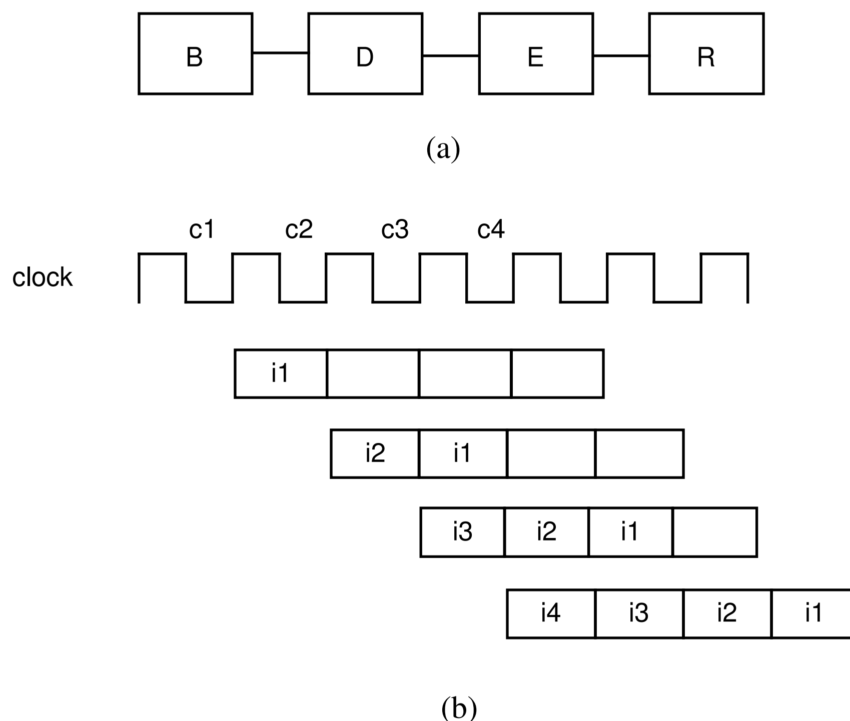


Figura 7.1. Execução de instruções em pipeline.

Neste pipeline, o estágio B realiza a busca da instrução, o estágio D decodifica a instrução, o estágio E executa a operação indicada pela instrução, e finalmente o estágio R armazena o resultado produzido pela instrução. A execução de uma instrução inicia-se pelo estágio B, sendo completada no estágio R. Ao ser finalizado um passo em um certo estágio, a instrução avança para o estágio seguinte. Em condições normais, uma instrução avança para o estágio seguinte a cada novo ciclo de clock.

O modo de execução em pipeline é mostrado na Figura 7.1(b). Esta figura mostra a posição das instruções dentro do pipeline ao final de cada ciclo de clock. No ciclo c1, a instrução i1 é buscada pelo estágio B. No ciclo c2, a instrução i1 é decodificada pelo estágio D, enquanto o estágio B busca uma nova instrução, i2. No ciclo c3, o estágio E executa a operação indicada pela instrução i1, ao mesmo tempo que o estágio D decodifica a instrução i2 e o estágio B busca a instrução i3. No ciclo c4 o resultado da instrução i1 é armazenado pelo estágio R. Ainda em c4, as instruções i2 e i3 avançam para o próximo estágio, e o estágio B acessa a instrução i4.

Note que novas instruções entram no pipeline antes que a execução das instruções anteriores seja completada. Quando o pipeline encontra-se cheio, várias instruções estão sendo executadas em paralelo, uma em cada estágio do pipeline. No pipeline da Figura 7.1(b), a cada instante de tempo até quatro instruções podem se encontrar em execução.

Na realidade, o aspecto mais importante na técnica de pipeline é que ela permite que uma instrução seja completada a cada ciclo de clock. Isto pode ser visto a partir da Figura 7.2.

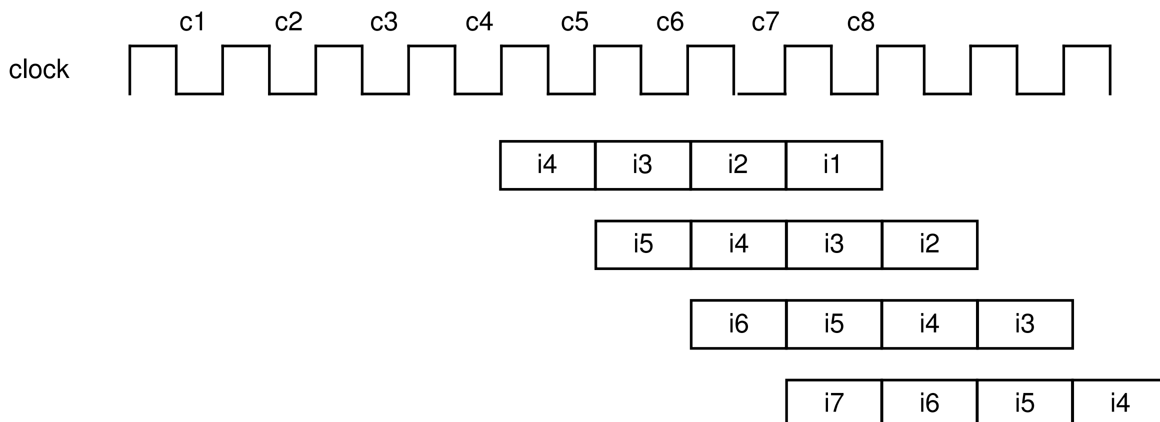


Figura 7.2. Completando instruções a cada ciclo de clock.

Esta figura mostra a continuação da execução iniciada na Figura 7.1(b). Ao final do ciclo c4 a execução da instrução i1 é completada, ao final do ciclo c5 a execução de i2 é completada, e assim por diante. Entre os ciclos c4 e c7 são completadas quatro instruções, i1, i2, i3 e i4. Isto significa que, em média, foi necessário um ciclo para executar cada instrução.

Como visto no Capítulo 2, em uma arquitetura seqüencial a execução de uma instrução consome vários ciclos de clock. Nestas arquiteturas, o número médio

de ciclos por instrução está bem acima da média de 1 ciclo/instrução obtida com o emprego da técnica de pipelining. A redução do número médio de ciclos por instrução contribui diretamente para aumentar o desempenho. Isto pode ser visto intuitivamente: se um processador requer um menor número de ciclos para executar cada instrução, um certo número de instruções será executado em um número de ciclos menor, o que equivale a dizer, em um tempo menor. Assim, este processador apresentará um desempenho maior.

A influência do número de ciclos por instrução sobre o desempenho pode ser mostrada de uma maneira mais formal através da expressão que mede o desempenho pelo tempo de execução de um programa. Quanto menor o tempo que um processador consome para executar um programa, maior será o seu desempenho. O tempo de execução de um programa depende do número de instruções executadas e do tempo médio de execução de cada instrução, ou seja:

tempo de execução = número de instruções x tempo médio por instrução

Por sua vez, o tempo médio de execução de cada instrução é determinado pelo número médio de ciclos por instrução (cpi) e pelo tempo do ciclo de clock:

tempo médio por instrução = cpi x tempo de ciclo de clock

Assim, o tempo de execução é dado por:

tempo de execução = número de instruções x cpi x tempo de ciclo do clock

Através desta expressão, vê-se que o tempo de execução é diretamente proporcional ao número médio de ciclos por instrução. Ao reduzir este fator para 1 ciclo/instrução, a técnica de pipelining contribui para aumentar o desempenho de forma significativa. Por isso, atualmente todas as arquiteturas de processador voltadas para aplicações de alto desempenho utilizam a técnica de pipelining.

Apesar de conceitualmente simples, na prática a técnica de pipelining apresenta alguns problemas. Voltando à Figura 7.2, note que a taxa de 1 ciclo/instrução é obtida somente quando é mantido um fluxo contínuo de instruções através do pipeline. Se isto acontece, uma nova instrução deixa o pipeline a cada ciclo, ou seja, o número de instruções executadas será igual ao número de ciclos, resultando na média de 1 ciclo por instrução executada. No entanto, existem alguns impedimentos para que seja mantido um fluxo contínuo de instruções através do pipeline. Um das dificuldades em manter a continuidade do fluxo de instruções acontece, por exemplo, quando existe uma dependência de dados entre duas instruções. Uma dependência entre duas instruções i e j existe quando um dos operandos da instrução j é o resultado da instrução i anterior. Lembre-se que, em um pipeline, instruções são executadas em paralelo. Se estas duas instruções são executadas simultaneamente no pipeline, pode acontecer que a instrução i ainda não tenha produzido o seu resultado no momento em que a instrução j lê os seus operandos. Se não existir nenhum controle, a instrução j lê um dos operandos com um valor antigo, ainda não atualizado pela instrução i, levando a uma execução incorreta. Note que este problema não existe quando instruções são executadas sequencialmente.

A solução mais simples para garantir que a dependência seja respeitada consiste em paralizar a execução da instrução j (e das instruções subseqüentes) quando esta chega ao estágio onde seus operandos são acessados, até que a instrução i seja completada. No entanto, esta paralização parcial do pipeline resulta em uma descontinuidade no fluxo de instruções, elevando o número médio de ciclos por instrução.

Devido as situações que provocam a quebra do fluxo de instruções, na prática o número médio de ciclos por instrução aproxima-se, mas não é exatamente igual, a 1 ciclo/instrução. No entanto, ainda assim esta média fica bem abaixo daquela observada em arquiteturas sem pipeline. Existem várias técnicas que procuram minimizar a ocorrência de situações onde o fluxo de instruções é interrompido. No entanto, a análise destas técnicas, apesar de extremamente interessante, não cabe no escopo deste texto.

7.2 Arquiteturas Super-Escalares

A expressão apresentada na seção anterior relaciona o tempo de execução com o número médio de ciclos necessários para executar uma instrução (cpi). Podemos também expressar o tempo de execução em função do número de instruções executadas por ciclo, ou ipc. Se em média são necessários C ciclos para executar I instruções, o número médio de ciclos por instrução será cpi = C/I, enquanto o número médio de instruções completadas por ciclo será ipc = I/C. Note então que cpi = 1/ipc, e assim a expressão para o tempo de execução pode ser reescrita como:

$$\text{tempo de execução} = \frac{\text{número de instruções} \ \text{x} \ \text{tempo de ciclo de } \textit{clock}}{\text{ipc}}$$

Esta expressão indica que o tempo de execução diminui, e por conseguinte o desempenho aumenta, na medida que o número de instruções completadas por ciclo aumenta.

Na técnica de pipelining, apenas uma instrução é completada por ciclo (veja a Figura 7.2), resultando em um fator ipc máximo de 1 ciclo/instrução. Seria possível aumentar ainda mais o desempenho caso o fator ipc fosse elevado para acima desta média. Para obter um aumento adicional no desempenho pela elevação do fator ipc, seria necessário permitir que mais de uma instrução fosse completada a cada ciclo. Esta é a idéia central nas arquiteturas super-escalares.

Uma arquitetura super-escalar é dotada de múltiplas unidades funcionais independentes, que executam instruções em paralelo. A cada ciclo, várias instruções podem ser enviadas, ou despachadas, para execução nestas unidades funcionais. Desta forma, é possível completar a execução de várias instruções a cada ciclo de clock, e assim aumentar o fator ipc para além de 1 instrução/ciclo. A Figura 7.3 mostra a organização básica de uma arquitetura super-escalar.

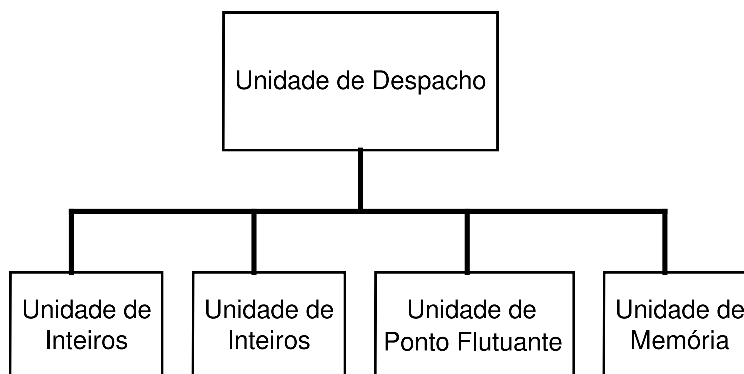


Figura 7.3. Arquitetura super-escalar básica.

Nesta arquitetura super-escalar hipotética, até quatro instruções podem ser completadas por ciclo. Existem duas unidades de inteiros que executam instruções aritméticas e lógicas sobre números inteiros, uma unidade de ponto flutuante que executa instruções aritméticas sobre números com ponto flutuante, e ainda uma unidade de memória que executa instruções de acesso à memória.

A cada ciclo, a unidade de despacho acessa e decodifica um certo número de instruções, e verifica quais destas instruções podem ser despachadas para as unidades funcionais. Tipicamente, uma instrução é despachada quando a unidade funcional apropriada encontra-se disponível e quando não existe uma dependência de dados entre esta instrução e uma outra instrução ainda em execução.

Os processadores mais recentes destinados a aplicações de alto desempenho apresentam arquiteturas super-escalares. A título de exemplo, a seguir descrevemos resumidamente três arquiteturas super-escalares: Alpha AXP, Pentium e PowerPC 601.

7.3 A Arquitetura Alpha AXP

A arquitetura Alpha AXP foi introduzida pela Digital Equipment Corp. em 1993. A primeira implementação desta arquitetura é o DEC Alpha 21064, um processador que opera com freqüência de clock de 200 MHz. A organização da arquitetura Alpha é mostrada na Figura 7.4.

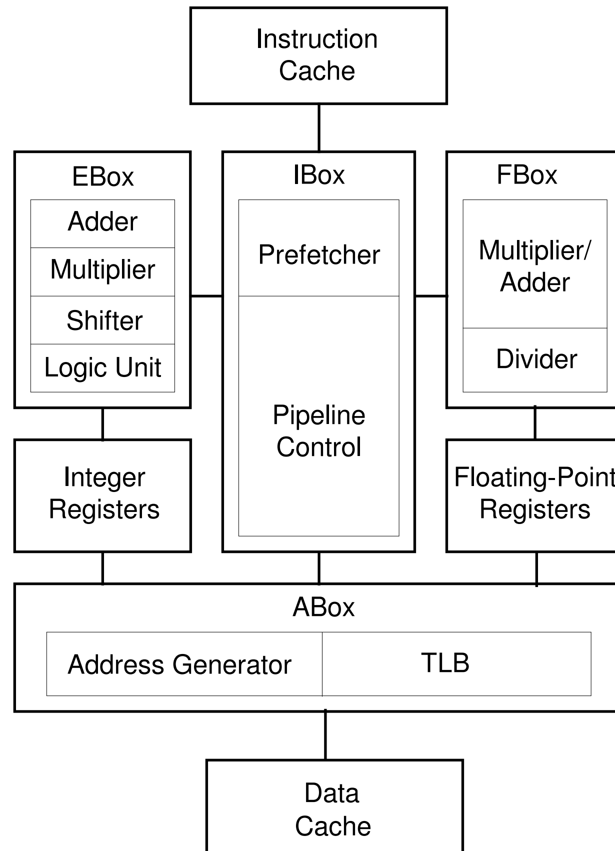


Figura 7.4. Organização da arquitetura DEC Alpha AXP.

A arquitetura Alpha possui quatro unidades funcionais. A unidade IBox realiza a busca e o despacho de instruções, e executa as instruções de transferência de controle. A unidade EBox executa as instruções aritméticas e lógicas sobre inteiros, enquanto a unidade FBox executa as instruções aritméticas sobre números com ponto-flutuante. A Ebox opera sobre inteiros de 64 bits, enquanto a FBox manipula números com ponto flutuante com precisão simples (32 bits) e precisão dupla (64 bits). Instruções de acesso à memória são executadas pela unidade ABox. As unidades EBox e FBox possuem possuem conjuntos de registradores separados, cada um com 32 registradores de 64 bits. A arquitetura ainda inclui memórias cache separadas para instruções e dados. Ambas as caches são do tipo associativa por conjunto. No Alpha 21064, cada memória cache possui 8 Kbytes, com linhas de 32 bytes e quatro linhas por conjunto.

No DEC Alpha 21064, as unidades funcionais são implementadas com a técnica de pipelining. Os pipelines no Alpha 21064 são mostrados na Figura 7.5.

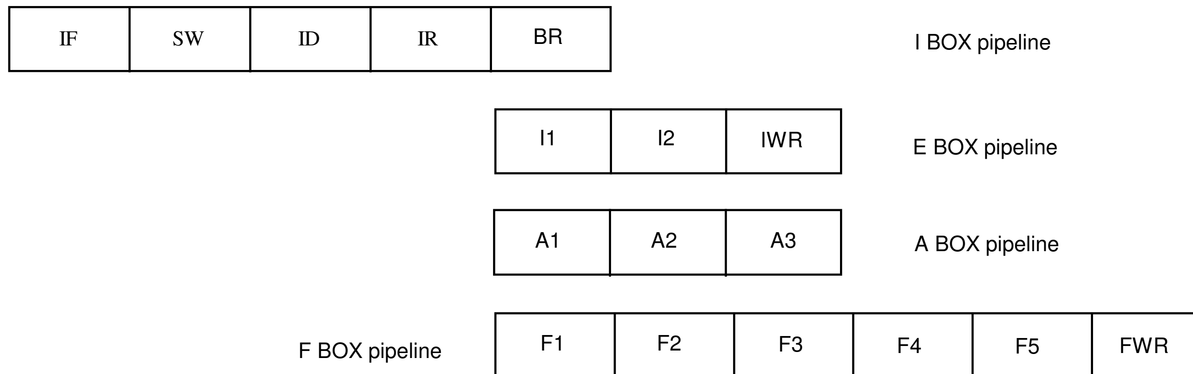


Figura 7.5. Pipelines no DEC Alpha 21064.

Os quatro primeiros estágios estão na IBox, e efetuam o acesso e despacho das instruções. A cada ciclo, o estágio IF acessa duas instruções na memória cache de instruções. O estágio SW faz uma previsão do resultado de instruções de desvio. Normalmente, o estágio SW solicita ao estágio IF a busca das duas próximas instruções seqüenciais. Ao encontrar uma instrução de desvio, este estágio procura antecipar o resultado do desvio (desvio efetuado/desvio não-efetuado) e solicita a busca das instruções no destino previsto.

O estágio SW também é responsável por determinar se duas instruções podem ser despachadas simultaneamente. No Alpha 21064, algumas combinações de instruções não podem ser despachadas em um mesmo ciclo, devido a conflitos no acesso aos conjuntos de registradores e no uso dos barramentos internos. A Tabela 7.1 mostra as combinações de instruções que podem ser despachadas simultaneamente.

Instrução 1	Instrução 2
instruções inteiras	instruções de ponto flutuante
instruções de acesso à memória	instruções inteiras/ponto flutuante
instruções de transferência de controle	instruções de acesso à memória, inteiras ou de ponto flutuante

Tabela 7.1. Possíveis combinações de instruções para despacho simultâneo no Alpha 21064.

Quando não é possível despachar duas instruções no mesmo ciclo, o estágio SW serializa o despacho, enviando as instruções do par para o estágio seguinte em ciclos diferentes. O estágio ID completa a decodificação de instruções iniciada no estágio SW, verificando possíveis dependências entre as instruções a serem despachadas. Caso haja alguma interdependência, o despacho destas instruções é serializado. Finalmente, no estágio IR os operandos das instruções são acessados nos registradores, e as instruções são enviadas para execução.

Instruções de desvio são executadas no estágio BR, ainda na IBox. As instruções aritméticas e lógicas sobre inteiros são executadas nos estágios I1 e I2

na EBox. O resultado é armazenado em registradores no estágio IWR. Instruções de ponto flutuante são executadas na FBox nos estágios F1 a F5. O resultado é escrito em um registrador de ponto flutuante no estágio FWR.

As instruções de acesso à memória são executadas nos estágios A1, A2 e A3 na Abox. O endereço efetivo é calculado no estágio A1. A conversão de endereço virtual para endereço real e o acesso à memória cache de dados acontecem no estágio A2. Em instruções de leitura da memória, o dado acessado é armazenado no registrador-destino no estágio A3.

7.4 Arquitetura do Pentium

O processador Pentium, lançado pela Intel Corp. em 1993, também apresenta uma arquitetura super-escalar. A versão inicial do Pentium opera com uma freqüência de clock de 66 MHz. A organização da arquitetura do Pentium aparece na Figura 7.6.

O Intel Pentium inclui três unidades funcionais. As unidades denominadas U-Pipe e V-Pipe executam instruções aritméticas e lógicas sobre números inteiros de 64 bits. A FPU (Floating-Point Unit) executa instruções aritméticas sobre números com ponto-flutuante com precisão simples (32 bits), precisão dupla (64 bits) e precisão estendida (128 bits). Ao contrário do 80486, que também inclui uma unidade de ponto flutuante em sua arquitetura, no Pentium a FPU usa a técnica de pipelining. No 80486 instruções de ponto flutuante são executadas seqüencialmente, com a execução uma nova instrução iniciando-se apenas após o término da anterior. Com pipelining, a FPU do Pentium permite que uma nova instrução de ponto flutuante seja iniciada antes do término da anterior.

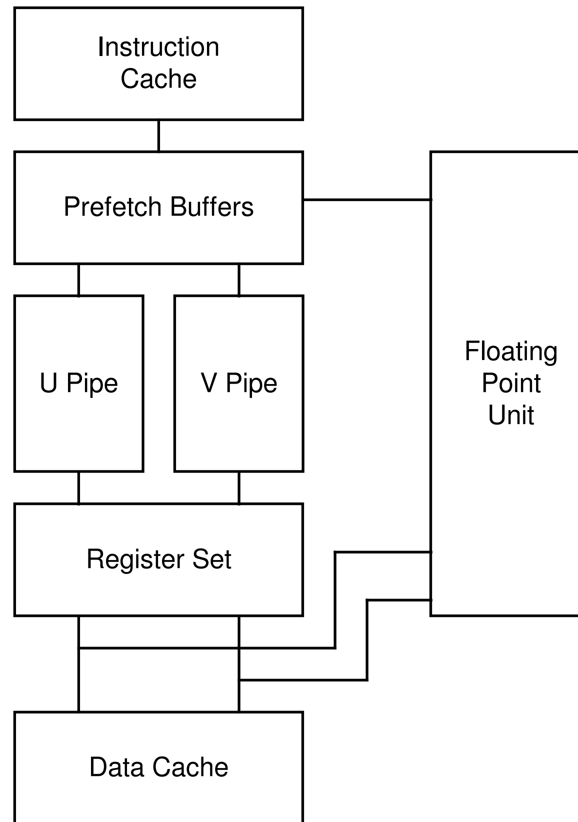


Figura 7.6. Organização da arquitetura do Intel Pentium.

Associado às duas unidades de inteiros existe um conjunto de registradores semelhante ao existente nos processadores da família 80x86 (veja Capítulo 4). A FPU contém um conjunto de 8 registradores. Ao contrário do 80486, que possui apenas uma memória cache para instruções e dados, o Pentium inclui memórias cache separadas. Ambas as memórias cache no Pentium possuem 8 Kbytes, e são do tipo associativa por conjunto, com 32 bytes por linha e duas linhas por conjunto.

No Pentium, as instruções sobre inteiros são executadas em um pipeline com 5 estágios, enquanto instruções de ponto flutuante são executadas em um pipeline com 8 estágios. Os pipelines no Pentium são mostrados na Figura 7.7.

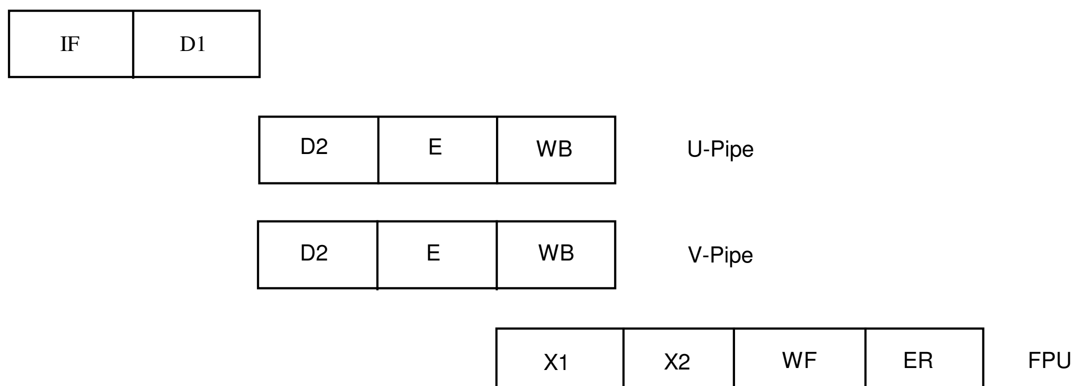


Figura 7.7. Estrutura dos pipelines no Intel Pentium.

Os dois primeiros estágios são comuns às unidades U-Pipe, V-Pipe e FPU. O estágio IF acessa simultaneamente duas instruções na memória cache, as quais são decodificadas no estágio D1. O estágio D1 decide se as duas instruções podem ser despachadas no mesmo ciclo para as unidades U-Pipe e V-Pipe. As instruções serão despachadas simultaneamente se ambas forem “instruções simples”, e se não existir uma dependência de dados entre elas. Uma “instrução simples” é aquela cuja execução não é controlada via microprograma, e que pode ser executada em um único ciclo de clock. Se uma das instruções não é simples, a primeira instrução do par é despachada para a U-Pipe. No ciclo seguinte, a segunda instrução do par é associada com uma nova instrução decodificada, e o estágio D1 novamente verifica se ambas podem ser despachadas simultaneamente. Se este ainda não é o caso, a segunda instrução do par anterior é despachada e o processo se repete.

Os estágios D2, E e WB são replicados na U-Pipe e V-Pipe. No estágio D2, a decodificação da instrução é completada. No estágio E é realizada a operação na ALU no caso de instruções aritméticas e lógicas, ou a memória cache de dados é acessada em instruções de acesso à memória. No estágio WB o resultado da instrução é escrito no registrador-destino.

Além dos estágios IF e D1, o pipeline da FPU também usa os dois primeiros estágios da U-Pipe e V-Pipe. Isto significa que uma instrução sobre inteiros não pode ser despachada juntamente com uma instrução de ponto flutuante no mesmo ciclo. Uma instrução de ponto flutuante é despachada para o estágio D2 da U-Pipe. A FPU usa os estágios E da U-Pipe e V-Pipe para acessar operandos na memória cache de dados. Os estágios seguintes localizam-se de fato na FPU. Nos estágios X1 e X2 a instrução é executada, e o resultado armazenado no registrador-destino no estágio WF. O estágio ER acontece o tratamento de erros que podem acontecer execução da instrução de ponto flutuante.

Com a adoção de uma arquitetura super-escalar, o Pentium pode atingir um desempenho significativamente maior que o processador da geração anterior, o Intel 486. Uma comparação do desempenho do Pentium com o 80486 é mostrada na Figura 7.8.

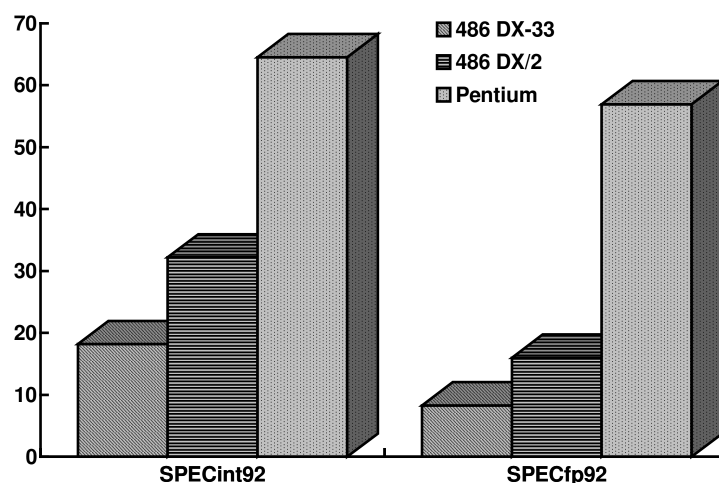


Figura 7.8. Comparação do desempenho do 80486 e do Pentium.

7.5 A Arquitetura PowerPC 601

A arquitetura PowerPC foi desenvolvida em conjunto pela Apple Computer, IBM Corp. e Motorola. Na realidade, a arquitetura PowerPC é baseada na arquitetura IBM POWER (Power Optimization with Enhanced RISC), implementada pelos processadores da linha de estações de trabalho IBM RS/6000. Por este motivo, os processadores PowerPC e RS/6000 são semelhantes em diversos aspectos.

O PPC 601 é o primeiro processador da família PowerPC. Outros três processadores, o PPC 603, PPC 604 e PPC 620, estão com lançamento previsto até 1995. Estes processadores diferem quanto à freqüência de clock e o número de instruções que podem ser completadas por ciclo. A Tabela 7.2 mostra uma comparação entre os processadores da família PowerPC.

	Objetivo de Mercado	Clock	Instruções/ciclo
PPC 601	relação custo/desempenho	50, 66, 80 MHz	3
PPC 603	baixo consumo (laptops)	66, 80 MHz	3
PPC 604	relação custo/desempenho	-	4
PPC 620	alto desempenho	-	6

Tabela 7.2. Processadores da família PowerPC.

A descrição que se segue é particular para o PPC 601. A Figura 7.9 mostra a organização da arquitetura do PowerPC 601.

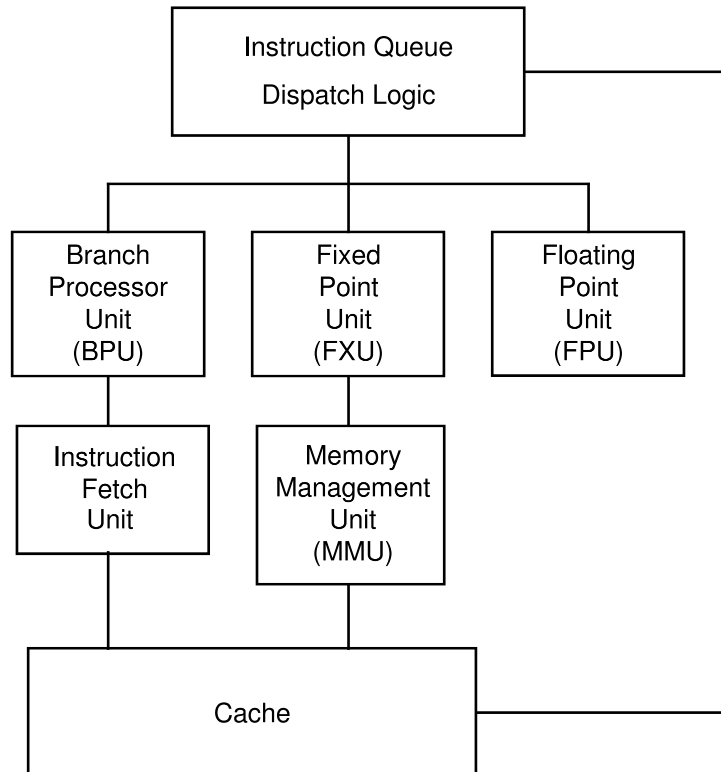


Figura 7.9. Organização da arquitetura do PowerPC 601.

O PPC 601 possui três unidades funcionais. A unidade FXU executa as instruções aritméticas e lógicas sobre inteiros de 64 bits. Operações sobre números com ponto flutuante são executadas na FPU. Números com ponto flutuante podem ter precisão simples (32 bits) ou precisão dupla (64 bits). A FXU e FPU possuem conjuntos de registradores separados, cada um com 32 registradores de 64 bits. A unidade BPU executa apenas as instruções de transferência de controle. A unidade de busca de instruções (Instruction Fetch Unit) não executa nenhum tipo de instrução, sendo apenas responsável por acessar a memória cache e preencher a fila de instruções na unidade de despacho. A unidade MMU é responsável por controlar a memória virtual. O PPC 601 inclui uma memória cache unificada para instruções e dados. A memória cache armazena 32 Kbytes e é do tipo associativa por conjunto, com 64 bytes por linha e oito linhas por conjunto. A arquitetura do PPC 603, e provavelmente as do PPC 604 e PPC 620, possui memórias cache de instruções e de dados separadas.

A estrutura dos pipelines na arquitetura PPC 601 é mostrada na Figura 7.10.

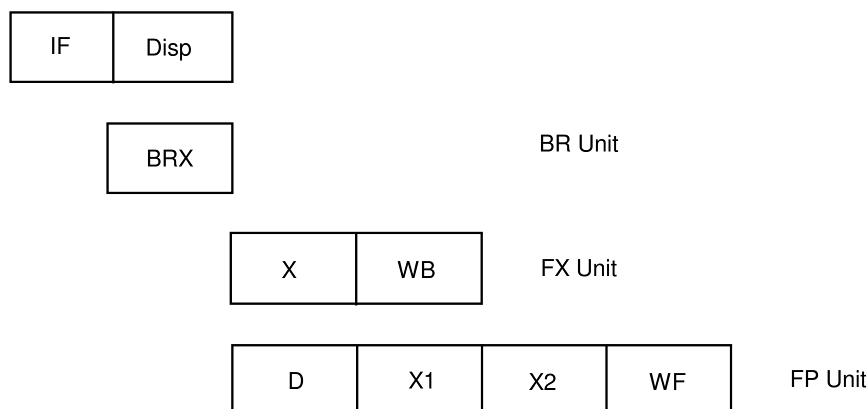


Figura 7.10. Pipelines no PowerPC 601.

O estágio IF realiza a busca de instruções, enquanto o estágio Disp decodifica e despacha instruções. A cada ciclo, o estágio IF acessa até 8 instruções na memória cache, que são armazenadas em uma fila até serem despachadas. A unidade de despacho pode enviar até três instruções para execução em um mesmo ciclo.

Como a figura indica, uma instrução de transferência de controle é executada no estágio BRX no mesmo ciclo em que despachada. Instruções aritméticas e lógicas são executadas no estágio X da FXU, sendo o resultado armazenado no registrador-destino no estágio WB.

Instruções de ponto flutuante não são decodificadas no estágio Disp, mas no estágio D da FPU. A operação de ponto flutuante é executada nos estágios X1 e X2, e o resultado armazenado no estágio WF.

Embora conceitualmente simples, a execução paralela de instruções em arquiteturas super-escalares envolve vários problemas. Os principais são as dependências de dados entre instruções e as instruções de transferência de controle, que limitam o número de instruções despachadas por ciclo. Arquiteturas super-escalares incorporam mecanismos sofisticados para minimizar os efeitos destes dois fatores e permitir uma exploração mais efetiva do potencial oferecido. Um outro aspecto que se torna muito importante com as arquiteturas super-escalares é a qualidade do código gerado pelo compilador. O compilador desempenha um papel fundamental em otimizar a seqüência de instruções no código do programa, de modo a minimizar a freqüência de instruções que não podem ser despachadas simultaneamente. Avaliações no Intel Pentium mostram que um programa compilado por um compilador otimizador apresenta um desempenho 30% maior que um programa compilado por um compilador para uma arquitetura não super-escalar.

Apesar destes problemas, o modelo super-escalar vem sendo utilizado em um número cada vez maior de arquiteturas. Além daqueles aqui mostrados, podemos ainda citar como exemplos de processadores com arquitetura super-escalar o IBM RS/6000, o Motorola M88110, e o Sun SuperSPARC .

7.6 Arquiteturas RISC

Uma das principais tendências nas arquiteturas de processador que surgiram ao longo das décadas de 70 e 80 foi a crescente sofisticação dos seus conjuntos de instruções. Nestas arquiteturas, operações que antes eram realizadas por uma seqüência de várias instruções passavam a ser executadas por uma única instrução. Esta sofisticação das instruções decorria das limitações do hardware e do software disponíveis na época.

A principal limitação do hardware estava na capacidade de armazenamento e na velocidade da memória principal. Os dispositivos de memória apresentavam baixa densidade e alto tempo de acesso. Devido à baixa capacidade de armazenamento da memória principal, era desejável que o código executável de um programa tivesse tamanho reduzido. A velocidade da memória principal contribuía para reforçar esta necessidade. O tempo de acesso à memória principal era em média 10 vezes maior que o tempo de execução de uma instrução com operandos em registradores. Este desbalanceamento levou ao princípio amplamente aceito na época de que o tempo de execução de um programa seria proporcional ao tamanho do seu código. Programas longos teriam um maior tempo de execução devido ao maior número de instruções acessadas na memória. Assim, desejava-se reduzir o tamanho do código não somente para economizar espaço, mas também para obter-se um melhor desempenho.

Do lado do software, o estado inicial dos compiladores apresentavam limitações que somavam-se às do hardware. Como ainda não haviam sido desenvolvidas técnicas eficientes de otimização de código, o código gerado por um compilador era em geral maior do que o código do mesmo programa quando escrito diretamente em linguagem assembly. Programas em linguagens de alto nível eram assim considerados ineficientes em termos de espaço e tempo de execução. No entanto, a crescente complexidade das aplicações foi tornando proibitiva a programação em linguagem assembly. À medida que as aplicações se tornavam mais sofisticadas, o uso de linguagens assembly resultava em uma dificuldade e custo de desenvolvimento cada vez maiores. O uso de linguagens de alto nível tornava-se imprescindível.

Tais circunstâncias motivaram o desenvolvimento de arquiteturas para suportar o uso de linguagens de alto nível. Programas de alto nível compilados para tais arquiteturas deveriam ser tão eficientes quanto programas escritos em assembly. Estas arquiteturas receberam o nome de arquiteturas para linguagens de alto nível, ou arquiteturas HLLC (High-Level Language Computer Architectures). As arquiteturas conhecidas como CISC (Complex Instruction Set Computers) são um tipo particular de arquiteturas HLLC. Usaremos aqui o termo “arquitetura complexa” referindo-se às arquiteturas HLLC em geral e às arquiteturas CISC em particular.

Para alcançar o seu objetivo, a abordagem usada nas arquiteturas complexas foi elevar o nível de funcionalidade do conjunto de instruções. O raciocínio por detrás desta abordagem é apresentado a seguir.

7.6.1 Lacuna Semântica

Uma das características em uma arquitetura de processador é o nível de funcionalidade do seu conjunto de instruções. Em um conjunto de instruções com alto nível de funcionalidade, cada instrução realiza um grande número de operações. Ao contrário, em um conjunto de instruções com baixo nível de funcionalidade, cada instrução tipicamente executa apenas uma única operação. A Figura 7.11 mostra um exemplo que esclarece este conceito. Nesta figura, o comando de alto nível C = A + B é implementado usando-se instruções com diferentes níveis de funcionalidade.

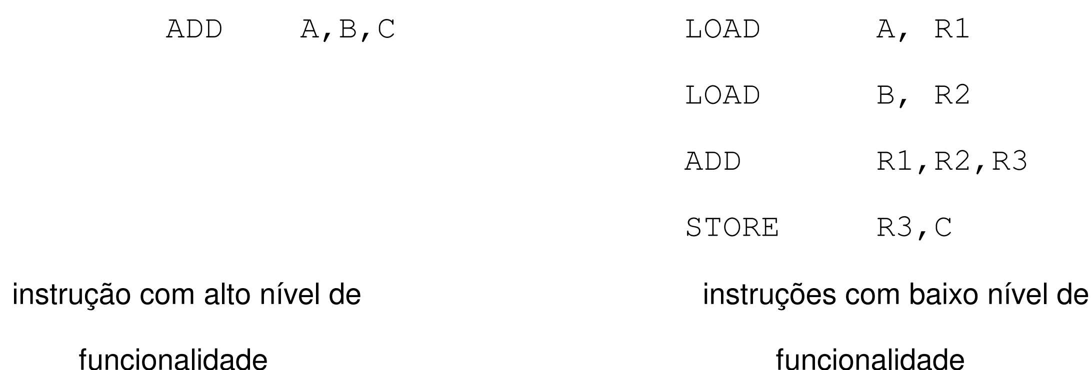


Figura 7.11. Comparação entre instruções com alto e baixo nível de funcionalidade.

O comando A=B+C envolve quatro operações: carregamento das variáveis A e B em registradores, a adição propriamente dita e finalmente o armazenamento do resultado em C. Em um conjunto de instruções com alto nível de funcionalidade, existe uma instrução de adição que executa todas estas operações. Por exemplo, a instrução ADD A,B,C acessa diretamente os operandos na memória, realiza a adição e ainda armazena o resultado na memória. Em um conjunto de instruções com baixo nível de funcionalidade, a instrução de adição realiza apenas uma operação, a adição em si, sobre operandos em registradores. O carregamento dos registradores e o armazenamento do resultado são realizados por instruções adicionais, como mostra a figura.

O conceito de nível de funcionalidade também é aplicável à linguagens de programação. Linguagens como Pascal e C apresentam um alto nível de funcionalidade — daí serem chamadas "linguagens de alto nível". Por exemplo, considere o comando FOR na linguagem C. Cada iteração do FOR envolve (1) o incremento de uma variável de controle, (2) a comparação do novo valor da variável de controle com um certo valor limite e (3) um eventual desvio para o início do loop. Comandos de alto nível como o FOR possuem um alto nível de funcionalidade porque a execução de um único comando envolve várias operações.

Linguagens de programação de alto nível e conjuntos de instruções normalmente apresentam níveis de funcionalidade diferentes. À esta diferença entre níveis de funcionalidade foi dado o nome de lacuna semântica (semantic gap). A abordagem usada pelas arquiteturas complexas foi o de elevar o nível de

funcionalidade do conjunto de instruções, diminuindo a lacuna semântica. Com isto, procurava-se reduzir o número de instruções necessárias na implementação dos comandos de alto nível. Para ilustrar esta idéia, considere o exemplo na Figura 7.12.

A Figura 7.12(a) mostra implementação do comando `FOR` em uma arquitetura com instruções com baixo nível de funcionalidade. Neste caso, o acesso e decremento da variável de controle do loop e o desvio condicional exigem quatro instruções. Suponha agora uma arquitetura que oferece uma instrução `DBNZ` (Decrement and Branch if Not Zero), a qual decrementa um valor em memória e realiza um desvio condicional se o resultado do decremento é não nulo. Nesta arquitetura, a implementação do `FOR` exige apenas uma instrução, ao invés de quatro instruções, como mostra a Figura 7.12(b).

```
for (i = 0; i < 100; i++)
{
(corpo do loop)
}
```

```
  STORE  100,I
LOOP:    .
         . corpo do loop
         .
     LOAD    I,R1
     SUB     1,R1,R1
     STORE   R1,I
     JNZ     LOOP
```

(a)

```
  STORE    100,I
LOOP:      .
           . corpo do loop
           .
     DBNZ      I,LOOP
```

(b)

Figura 7.12. Implementação do FOR usando instruções com diferentes níveis de funcionalidade.

Uma arquitetura com uma pequena lacuna semântica, ou seja, com um conjunto de instruções com um alto nível de funcionalidade, traria dois benefícios. Em primeiro lugar, os programas de alto nível seriam mais eficientes em termos do espaço ocupado na memória, já que a implementação dos comandos de alto nível exigiriam um menor número de instruções. Isto aliviaria os problemas decorrentes

da limitação na capacidade de armazenamento da memória principal. Em segundo lugar, ao implementar comandos de alto nível com um menor número de instruções, o número total de acessos à memória para busca de instruções seria também menor. Assim, os programas de alto nível seriam mais eficientes também em tempo de execução. Isto compensaria o desbalanceamento entre a velocidade do processador e da memória principal.

7.6.2 Críticas às Arquiteturas Complexas

À medida que surgiam novas arquiteturas complexas, vários estudos foram sendo realizados com o objetivo de avaliar o comportamento destas arquiteturas. Os resultados obtidos apontaram várias desvantagens na filosofia de aumentar o nível de funcionalidade das instruções. As principais críticas contra as arquiteturas complexas, formuladas com base nas evidências fornecidas por estas pesquisas, são as seguintes.

Eficiência dos Programas. A primeira crítica levantada contra as arquiteturas complexas é a de que o aumento do nível de funcionalidade das instruções não necessariamente resulta em programas mais eficientes. Como mencionado, uma das vantagens esperadas das arquiteturas complexas seria a redução no tempo de execução de programas, conseqüência da substituição de seqüências de instruções simples por uma única instrução complexa. No entanto, foram observados casos onde isto não é verdade. Um exemplo é a `INDEX`, do VAX-11, usada no acesso a vetores. Esta instrução verifica se o índice do elemento a ser acessado está dentro dos limites do vetor e em seguida calcula o endereço da posição de memória onde está armazenado aquele elemento. Verificou-se que a instrução `INDEX` poderia ser substituída por uma seqüência de instruções simples que realizariam estas mesmas operações em um tempo 45% menor. Exemplos como este mostravam que algumas instruções complexas apresentavam um tempo de execução elevado, às vezes maior que o tempo de execução de uma seqüência de instruções simples que realiza a mesma tarefa.

Utilização de Instruções. Várias pesquisas foram realizadas para identificar quais as instruções que apareciam com maior freqüência no código compilado para arquiteturas complexas. Os resultados obtidos revelaram que apenas uma pequena parcela das instruções oferecidas era efetivamente utilizada. Por exemplo, foi observado que dentre as 183 instruções oferecidas pelo IBM 370, apenas 48 formavam a maioria (99%) das instruções executadas por um conjunto de diversos tipos de programas escritos em diferentes linguagens. Além disso, foi verificado que as instruções mais comuns eram justamente aquelas mais simples.

Verificou-se que o alto nível de funcionalidade na realidade restringe o uso das instruções complexas. Uma instrução complexa se assemelha ou implementa diretamente um comando de alto nível. Existem vários comandos em diferentes linguagens de alto nível que são semelhantes, no sentido que executam uma mesma operação. No entanto, em geral estes comandos diferem em vários detalhes. Por exemplo, o comando `DO` na linguagem FORTRAN e o comando `FOR` na linguagem C são usados para executar a mesma operação, qual seja, um loop.

No entanto, embora realizem a mesma operação, estes comandos diferem quanto a inicialização, atualização e teste da variável que controla a repetição do loop.

Diferenças como estas fazem com que comandos funcionalmente equivalentes sejam implementados de maneiras diferentes. Assim, uma instrução complexa semelhante a um comando em uma certa linguagem de alto nível pode não ser adequada na implementação do comando equivalente em outras linguagens. Na prática, estas instruções são substituídas por seqüências de instruções simples, com as quais é possível satisfazer exatamente os detalhes de implementação dos comandos em diferentes linguagens, diminuindo a utilização das instruções complexas.

Efeito sobre o Desempenho. Uma das principais críticas contra as arquiteturas complexas é que o aumento do nível de funcionalidade possui um efeito negativo sobre o desempenho. Esta afirmação baseia-se na relação entre o nível de funcionalidade das instruções e a complexidade de implementação da arquitetura.

Em geral, a unidade de controle de uma arquitetura complexa é implementada com a técnica de microprogramação. Em geral, arquiteturas complexas oferecem um número elevado de instruções e modos de endereçamento. Além disso, a execução de uma instrução com alto nível de funcionalidade envolve a realização de um grande número de operações básicas. Devido à estas características, a implementação da unidade de controle usando lógica aleatória torna-se extremamente difícil. Na prática, a microprogramação é a única alternativa para contornar as dificuldades de implementação de uma arquitetura complexa.

Como visto no Capítulo 2, a execução de uma microinstrução envolve (1) a geração do endereço da microinstrução, (2) o acesso à memória de microinstrução e (3) o armazenamento no registrador de microinstrução. Cada um destes passos apresenta um certo retardo. A realização de cada um destes passos apresenta um certo retardo, que se somam determinando o tempo de execução da microinstrução. Como uma microinstrução é executada a cada ciclo de clock, o tempo de ciclo do clock também é determinado por estes retardos. Na prática, os retardos na unidade de controle microprogramada dificultam a redução do ciclo de clock, limitando assim o desempenho.

A questão aqui colocada é que nos casos onde os retardos inerentes à uma unidade de controle microprogramada contribuem para limitar ou aumentar o tempo do ciclo de clock, a existência de instruções funcionalmente complexas contribuiria na realidade para reduzir o desempenho.

Existe um ponto em comum por detrás de todas estas críticas contra as arquiteturas complexas: o compromisso entre a funcionalidade da arquitetura (isto é, do seu conjunto de instruções) e o seu desempenho. De um lado, por questões de eficiência e flexibilidade, as instruções com alto nível de funcionalidade são na prática pouco utilizadas na codificação dos programas de alto nível. Do outro lado, a

inclusão destas instruções afeta negativamente o desempenho, na medida que implementação da arquitetura exige uma técnica que pode limitar o ciclo de clock. Assim, o argumento central é o de que as arquiteturas complexas apresentam um mau compromisso entre funcionalidade e desempenho, porque paga-se com uma perda no desempenho por uma funcionalidade que não é efetivamente utilizada. Foi esta constatação que levou à idéia de simplificar o conjunto de instruções para obter-se arquiteturas com melhor desempenho. Esta idéia é discutida a seguir.

7.6.3 A Filosofia RISC

As desvantagens das arquiteturas complexas relacionadas acima constituem um dos fatores que motivaram a filosofia RISC (Reduced Instruction Set Computers). Um outro fator foi o rápido avanço tecnológico ocorrido ao longo da década de 80.

Como visto, as arquiteturas complexas surgiram devido às limitações na capacidade de armazenamento e no tempo de acesso da memória principal. No entanto, a partir da década de 80 verificou-se um aprimoramento significativo na tecnologia de memórias semicondutoras. A capacidade de armazenamento dos dispositivos de memória aumentava rapidamente: a tendência era de um crescimento na capacidade de armazenamento de 60% ao ano, com uma quadruplicação a cada três anos. Além disso, devido ao aperfeiçoamento dos processos de fabricação e à produção em larga escala, havia uma constante diminuição no preço dos dispositivos de memória. Esta queda no preço possibilitava que novos sistemas fossem equipados com uma memória principal cada vez maior. O aumento na capacidade de armazenamento e a queda do preço dos dispositivos de memória praticamente eliminaram as limitações quanto ao tamanho da memória principal. O desbalanceamento entre a velocidade do processador e a velocidade da memória principal também foi atenuada com novos processos de fabricação, mas deixou realmente de ser um fator limitante com a introdução das memórias cache.

O novo quadro que existia no início da década de 80 era então o seguinte. Surgiam evidências de que instruções complexas não necessariamente contribuíam para melhorar o desempenho. Ao mesmo tempo, eram minimizadas as limitações no tamanho e na velocidade da memória principal. Assim, enquanto eram apontadas desvantagens no uso de instruções com alto nível de funcionalidade, desapareciam as circunstâncias que haviam justificado esta abordagem. Este novo quadro motivou e viabilizou a proposta das arquiteturas RISC.

A princípio básico na filosofia RISC é a simplicidade das instruções. Arquiteturas RISC fornecem apenas aquelas instruções simples mais freqüentemente usadas na codificação de programas de alto nível. Uma instrução complexa é incluída na arquitetura somente se forem satisfeitos dois critérios. Primeiro, a utilização desta instrução deverá ser significativa, ou seja, a instrução deverá ser realmente útil na codificação de diferentes tipos de programas escritos em diferentes linguagens. Segundo, o ganho final no desempenho deverá ser maior do que eventuais perdas no tempo de execução das instruções simples que ocorram com o acréscimo da instrução complexa. Nos casos em que algum destes

critérios não é atendido, torna-se preferível a implementação da instrução complexa em software, através de uma seqüência de instruções simples.

Ao adotar um conjunto de instruções simples, a filosofia RISC procura melhorar o desempenho através da diminuição da complexidade de implementação da arquitetura. Se a arquitetura oferece apenas instruções simples, torna-se viável implementar a unidade de controle usando lógica aleatória no lugar da técnica de microprogramação. Isto eliminaria os retardos em uma unidade de controle microprogramada, permitindo reduções no ciclo de clock e o aumento no desempenho.

7.6.4 Características das Arquiteturas RISC

Ao analisar as arquiteturas RISC hoje existentes é possível identificar várias características que são comuns a todas elas. As principais características das arquiteturas RISC são as seguintes:

Implementação com Lógica Aleatória. Esta primeira característica já foi discutida acima. Em arquiteturas RISC, a unidade de controle é implementada com lógica aleatória para permitir uma diminuição no tamanho do ciclo de clock.

Pipeline de Instruções. Todas as arquiteturas RISC adotam a técnica de pipelining na execução de instruções. Com isto, arquiteturas RISC procuram melhorar o desempenho também com a redução do número médio de ciclos por instrução.

Arquitetura Registrador-Registrador. Arquiteturas RISC adotam o modelo de arquitetura registrador-registrador. Como visto no Capítulo 3, neste tipo de arquitetura os operandos de instruções aritméticas e lógicas encontram-se apenas em registradores. A existência de instruções aritméticas/lógicas que podem acessar operandos diretamente na memória, como acontece nas arquiteturas complexas, contribui para aumentar a complexidade da unidade de controle. A adoção do modelo registrador-registrador segue o princípio de simplificar a implementação da arquitetura para tornar possível um menor tempo de ciclo.

Regularidade no Formato das Instruções. As instruções em arquiteturas RISC possuem uma codificação altamente regular. Todos os códigos de instrução possuem o mesmo tamanho, igual ao de uma palavra da memória. Com isto, uma instrução completa pode ser acessada em um único ciclo (caso ela se encontre na cache), o que é importante para manter um fluxo contínuo de instruções através do pipeline e simplificar a unidade de controle. Além disso, a regularidade permite, por exemplo, que os operandos sejam acessados em paralelo com a decodificação do código de operação da instrução. Isto diminui o tempo de decodificação e favorece a diminuição do tempo de ciclo.

7.6.5 Histórico das Arquiteturas RISC

Apesar das arquiteturas terem chegado ao mercado somente no final da década de 80, a idéia RISC surgiu em meados da década de 70. Em 1975, um grupo de pesquisa na IBM Yorktown Heights, liderados por John Cocke,

desenvolveram o IBM 801, que foi na realidade o primeiro sistema baseado em um processador com arquitetura RISC. A arquitetura POWER, usada nos processadores da linha de estações de trabalho IBM RS/6000 herda algumas características do IBM 801.

No início da década de 80, dois projetos em universidades americanas deram um impulso nas arquiteturas RISC. Na Universidade da Califórnia em Berkeley, um grupo dirigido por David Patterson desenvolveu o RISC I e o RISC II; um outro grupo na Universidade de Stanford, tendo à frente John Hennessy, desenvolveu o MIPS. Estes grupos desenvolveram os dois primeiros microprocessadores RISC, e demonstraram na prática o ganho de desempenho que poderia ser obtido com arquiteturas RISC. Posteriormente, o RISC II deu origem ao microprocessador Sun SPARC, usado nas estações de trabalho Sun SPARCStation 2. O MIPS deu origem origem à família de microprocessadores MIPS Rx000, usados nas estações de trabalho da DEC.

Devido às vantagens que apresenta para a implementação e para o desempenho, a filosofia RISC vem sendo adota por praticamente todos os processadores de alto desempenho lançados recentemente. As únicas exceções significativas são o Intel 80486 e Pentium (ainda assim, a própria Intel produz um processador RISC, usado basicamente em aplicações de engenharia e de processamento científico). A Tabela 7.3 relaciona os principais processadores RISC hoje (1994) disponíveis comercialmente.

Fabricante	Processador
DEC	Alpha AXP 21064
Hewlett-Packard	PA7100
IBM	RS/6000, PowerPC
Intel	i860
MIPS	R3000, R4000
Motorola	M88110, PowerPC
Sun	SPARC

Tabela 7.3. Relação de alguns processadores com arquitetura RISC.

7.7 Sistemas Paralelos

Até o momento, examinamos arquiteturas de computadores onde existe apenas um processador acoplado a uma memória principal. Um sistema paralelo é aquele onde existem vários processadores e módulos de memória, que se comunicam através de uma certa estrutura de interconexão. Nesta seção fornecemos uma visão geral dos tipos de sistemas paralelos existentes.

Atualmente, existe uma enorme variedade de sistemas paralelos. Uma das possíveis classificações de sistemas paralelos foi proposta por M. J. Flynn, e é mostrada na Tabela 7.4.

Denominação	Número de Fluxos de Instrução	Número de Fluxos de Dados	Exemplos de Sistemas Reais
SISD - Single Instruction, Single Data Stream	1	1	Sistemas convencionais Uniprocessadores
SIMD - Single Instruction Multiple Data Stream	1	Múltiplos	Supercomputadores vetoriais
MISD - Multiple Instruction Single Data Stream	Múltiplos	1	Nenhum
MIMD - Multiple Instruction Multiple Data Stream	Múltiplos	Múltiplos	Multiprocessadores Multicomputadores

Tabela 7.4. Classificação de sistemas paralelos, segundo Flynn.

A classificação de Flynn se baseia no número de fluxos de instrução e de fluxos de dados existentes no sistema. Um fluxo de instrução é uma seqüência de instruções endereçadas pelo contador de programa de um elemento processador. Se um sistema possui n contadores de programa, este sistema é capaz de processar n fluxos de instrução distintos. Um fluxo de dados corresponde a um conjunto de dados que é manipulado por um elemento processador.

7.7.1 Sistemas SISD

Um sistema SISD possui apenas um fluxo de instrução, e opera sobre um único fluxo de dados. Este tipo corresponde aos sistemas vistos até agora, onde existe um único processador. O contador de programa deste processador estabelece o único fluxo de instruções do sistema.

7.7.2 Sistemas SIMD

Um sistema SIMD também possui um único fluxo de instrução, mas opera sobre diversos fluxos de dados. Em um sistema deste tipo, existe um único contador de programa. Cada instrução endereçada por este contador de programa é executada em paralelo por diversos elementos processadores, cada elemento operando sobre um dado pertencente a um fluxo de dados diferente.

Os supercomputadores vetoriais constituem um exemplo de sistema SIMD. O conjunto de instruções de um supercomputador vetorial inclui instruções vetoriais, que realizam a mesma operação sobre múltiplos elementos de um vetor. Por exemplo, um supercomputador vetorial pode oferecer uma instrução `VADD` que soma, em paralelo, os elementos correspondentes de dois vetores.

Supercomputadores vetoriais são normalmente empregados em áreas científicas que exigem altíssimo desempenho, tais como física nuclear, mecânica de fluidos e de sistemas rígidos, previsão meteorológica, análise sísmica em prospecção de petróleo, dentre outras. Outra característica destes sistemas é o alto preço, que pode atingir centenas de milhões de dólares. Os sistemas Cray-1S, Cray-2S, Cray X-MP e Cray Y-MP, todos da Cray Research, Inc., são exemplos de supercomputadores vetoriais.

7.7.3 Sistemas MISD

Neste tipo de sistema, instruções pertencentes a múltiplos fluxos de dados operam simultaneamente sobre um mesmo dado. Na prática, não existe nenhum tipo de sistema que possa ser considerado de fato como do tipo MISD.

7.7.4 Sistemas MIMD

Em um sistema MIMD, existem múltiplos fluxos de instrução, cada um destes fluxos sendo executado por um processador diferente sobre um conjunto de dados diferente. Atualmente, dentro desta classe existem diversas sub-classes de sistemas com diferentes características. Uma classificação dos sistemas MIMD, proposta por C. G. Bell, é mostrada na Figura 7.13.

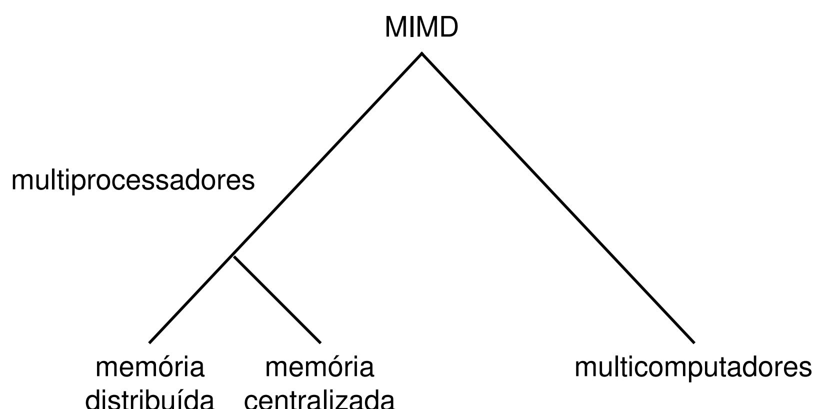


Figura 7.13. Classificação de sistemas MIMD, segundo Bell.

Esta classificação está baseada em dois critérios: o número de espaços de endereçamento vistos pelos processadores e o mecanismo de comunicação entre os processadores. Em um ramo estão os sistemas multiprocessadores, onde os múltiplos processadores do sistema vêem e compartilham apenas um único espaço de endereçamento lógico. Nestes sistemas, os processadores se comunicam através do espaço de endereçamento compartilhado por todos eles.

Os sistemas multiprocessadores diferem no modo como a memória lógica é implementada fisicamente. Em sistemas com memória compartilhada centralizada (centralized shared-memory) o espaço de endereçamento é implementado sob a forma de um único módulo de memória central, acessado por todos os processadores. Em sistemas com memória compartilhada distribuída (distributed shared-memory), o espaço de endereçamento é fisicamente implementado através de múltiplos módulos de memória. Os processadores acessam estes módulos de memória através de uma rede de interconexão.

No outro ramo dos sistemas MIMD estão os sistemas multicomputadores. Nestes sistemas, cada processador possui seu próprio espaço de endereçamento. Estes espaços de endereçamento não são compartilhados, ou seja, cada espaço de endereçamento é privativo de um certo processador. Por não haver um compartilhamento de memória, os processadores usam mensagens para comunicarem entre si.

Multicomputadores diferem quando à sua implementação física. Alguns sistemas são formados por módulos contendo processador e memória, sendo estes módulos interligados por canais de comunicação formando uma malha, cubo ou anel. Em outros multicomputadores, os componentes do sistema são computadores completos, incluindo processador(es), memória e dispositivos periféricos, interconectados por um meio de comunicação. Redes locais de computadores são exemplos deste tipo de multicomputador.

Atualmente, a tendência em sistemas paralelos segue em duas direções: no ramo dos multiprocessadores, para os sistemas com memória compartilhada distribuída e, dentro dos multicomputadores, para as implementações distribuídas como redes locais de computadores. Isto acontece porque estes dois tipos de sistemas são escaláveis. Em um sistema escalável, o desempenho aumenta (linear ou sub-linearmente) à medida que são acrescentados mais processadores. Por exemplo, atualmente existem sistemas com memória compartilhada distribuída maciçamente paralelos, formados por milhares de processadores. Estes sistemas possuem um desempenho da mesma ordem dos supercomputadores vetoriais, mas com um preço significativamente menor. Esta melhor relação desempenho-custo dos sistemas maciçamente paralelos vem atraindo a atenção de um número cada vez maior de pesquisadores e fabricantes.