Organização de Computadores

Prof. Robson de Souza

Nível de Microarquitetura - Otimização

Do ponto de vista de otimização, velocidade é apenas uma metade do quadro, custo é a outra. O custo pode ser medido de vários modos, mas uma definição precisa é problemática. Algumas medidas são muito simples, tal como uma contagem do número de componentes, o que era válido em particular na época em que os processadores eram compostos de componentes discretos, que eram comprados e montados. Hoje, o processador inteiro está contido em um único chip, mas chips mais complexos são muito mais caros do que os menores, mais simples.

Redução do comprimento do caminho de execução

Para melhor compreensão, vamos supor que exista uma microarquitetura, que chamaremos de Mic-1, que consistirá em uma unidade de controle que executa microprogramas de uma memória com 512 palavras. A CPU da Mic-1 também usa uma quantidade mínima de hardware: 10 registradores, uma ULA simples, um deslocador, um decodificador e um armazenamento de controle.

É possível fazer com que a IJVM possa ser executada de modo direto em microcódigo com pouco hardware, uma vez que cada instrução tem seu equivalente em hexadecimal, que pode ser convertido facilmente para binário. Com isso em mente, vamos estudar modos de reduzir o número de microinstruções por instrução ISA (isto é, reduzir o comprimento do caminho de execução).

*Arquitetura de três barramentos

Uma solução fácil é ter dois barramentos completos de entrada para ULA, um barramento A e um B, o que dá três barramentos no total. Todos os registradores (ou ao menos a maioria deles) devem ter acesso a ambos os barramentos de entrada. A vantagem de ter dois barramentos de entrada é que então é possível adicionar qualquer registrador a qualquer outro registrador em um só ciclo.

*Unidade de Busca da Instrução

Para conseguir uma melhoria drástica, é preciso algo mais radical. Vamos voltar atrás um pouco e examinar as partes comuns de toda instrução: a busca e a decodificação dos campos da instrução. Observe que, para cada instrução, podem ocorrer as seguintes operações:

1 - O PC é passado pela ULA e incrementado.
2 - O PC é usado para buscar o próximo byte na sequência de instruções.
3 - Operandos são lidos da memória.
4 - Operandos são escritos para a memória.
5 - A ULA efetua um cálculo e os resultados são armazenados de volta.

Se uma instrução tiver campos adicionais (para operandos), cada campo deve ser buscado explicitamente, um byte por vez, e montado antes de poder ser usado. Buscar e montar um campo ocupa a ULA por no mínimo um ciclo por byte para incrementar o PC.

Para sobrepor o laço principal, é necessário liberar a ULA de algumas dessas tarefas. isso poderia ser feito com a introdução de uma segunda ULA, embora não seja necessária uma completa para grande parte da atividade. Na Mic-1, grande parte da carga pode ser retirada da ULA criando uma unidade independente para buscar e processar as instruções. Essa unidade, denomina IFU (Instruction Fetch Unit – Unidade de Busca de Instrução), pode incrementar o PC independentemente e buscar bytes antes de eles serem necessários.

A IFU pode reduzir muito o comprimento do caminho da instrução média. **Primeiro**, ela elimina todo o laço principal, visto que o final de cada instrução apenas desvia diretamente para a próxima. **Segundo**, evita

ocupar a ULA com a tarefa de incrementar o PC.

*Projeto com pipeline

Além dessas técnicas, é possível ainda reduzir o tempo de ciclo. Ele é determinado, em considerável proporção, pela tecnologia subjacente. Quanto menores os transistores e mais curtas as distâncias físicas entre eles, mais rapidamente o clock pode ser executado.

Nossa outra opção é introduzir mais paralelismo na máquina. No momento, a nossa microarquitetura é altamente sequencial. Ela coloca registradores em seus barramentos, espera que a ULA e o deslocador os processem para depois escrever os resultados de volta nos registradores. Exceto pela IFU, há pouco paralelismo presente. Adicionar paralelismo é uma oportunidade real.

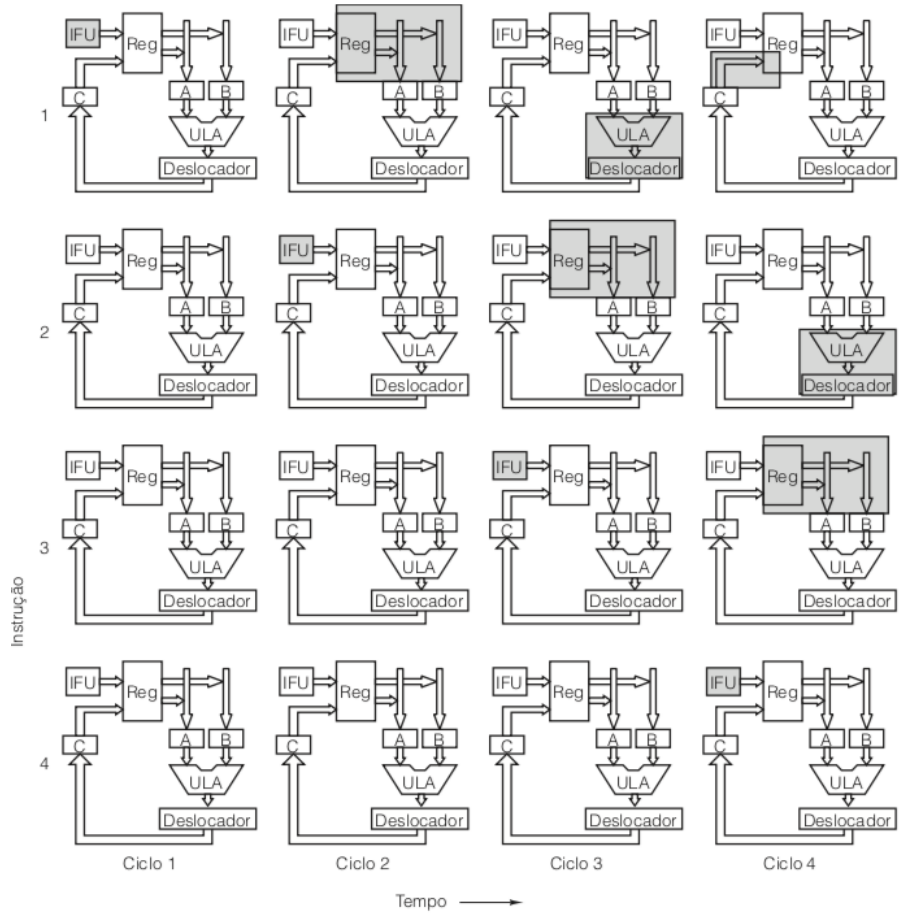
Como já foi visto, o ciclo de clock é limitado pelo tempo necessário para que os sinais se propaguem pelo caminho de dados. Há três componentes importantes no ciclo do caminho de dados propriamente dito:

1 - O tempo para levar os registradores selecionados até os barramentos A e B.
2 - O tempo para que a ULA e o deslocador realizem seu trabalho.
3 - O tempo para os resultados voltarem aos registradores e serem armazenados.

Esses tempos de “espera” podem ser utilizados para fazer alguma tarefa paralela durante o próprio ciclo, para isso, é possível desmembrar o caminho de dados em três partes. Para melhor compreensão, imagine que é como se tivéssemos três “caminhos de dados” menores para executar um caminho de dados completo, para isso são utilizados registradores extras.

Com isso, pode-se imaginar que todo o processo ficou mais lento, afinal agora, leva três ciclos de clock para usar o caminho de dados: um para carregar os barramentos A e B, um para executar a ULA e deslocador e carregar o barramento C, e um para armazenar o valor do barramento C de volta nos registradores. Porém, o que ocorre é o contrário, pois como o ciclo está desmembrado, podemos fazer a execução de cada um de forma paralela. Desse modo, enquanto uma das partes faz a sua tarefa com relação a alguma instrução, as outras partes já podem se adiantar executando tarefas referentes às próximas instruções.

A figura abaixo mostra o funcionamento desse pipeline, indicando quatro instruções e seus ciclos.



Fonte: (Tanenbaum e Austin, 2013)

É possível ainda, aumentar a quantidade de estágios do pipeline e obter mais desempenho.

Melhorias de desempenho

Todos os fabricantes de computadores querem que seus sistemas funcionem com a maior rapidez possível. As diversas opções para melhorias podem ser classificadas, de modo geral, em duas grandes categorias. **Melhorias de implementação** e **melhorias de arquitetura**.

Melhorias de implementação são modos de construir uma nova CPU ou memória para fazer o sistema funcionar mais rápido sem mudar a arquitetura. Modificar a implementação sem alterar a arquitetura significa que programas antigos serão executados na nova máquina, um importante argumento de venda.

Alguns tipos de melhorias só podem ser feitos com a alteração da arquitetura. Às vezes, essas alterações são incrementais, como adicionar novas instruções ou registradores, de modo que programas antigos continuarão a ser executados nos novos modelos. Nesse caso, para conseguir um desempenho completo, o software tem de ser alterado, ou ao menos recompilado com um novo compilador que aproveita as novas características.
Contudo, passadas algumas décadas, os projetistas percebem que a antiga arquitetura durou mais do que sua utilidade e que o único modo de progredir é começar tudo de novo.

*Memória cache

Um dos aspectos mais desafiadores do projeto de um computador em toda a história tem sido oferecer um sistema de memória capaz de fornecer operandos ao processador à velocidade em que ele pode processá-los. Um modo de atacar esse problema é providenciar caches.

Uma cache guarda as palavras de memória usadas mais recentemente em uma pequena memória rápida, o que acelera o acesso a elas. Se uma porcentagem grande o suficiente das palavras de memória estiver na cache, a latência efetiva da memória pode ter enorme redução.

Uma das técnicas mais efetivas para melhorar a largura de banda e também a latência é a utilização de várias caches. Uma técnica básica que funciona com grande eficácia é introduzir uma cache separada para instruções e dados. É possível obter muitos benefícios com caches separadas para instruções e dados, algo que muitas vezes denominamos **cache dividida**.

É possível adicionar outras caches auxiliares que vão sendo classificadas em níveis (cache nível 2, nível 3, etc.). Essas caches podem residir entre as caches de instrução e dados. As caches são em geral inclusivas, sendo que o conteúdo total da de nível 1 (L1) está na de nível 2(L2) e todo o conteúdo da cache de nível 2 está na de nível 3 (L3).

O próprio chip da CPU contém uma pequena cache de instrução e uma pequena cache de dados. Então, há a cache de nível 2, que geralmente não está no chip da CPU, mas pode ser incluída no pacote da CPU próxima ao chip da CPU e conectada a ela por um caminho de alta velocidade.

Caches dependem de dois tipos de **endereço de localidade** para cumprir seu objetivo. **Localidade espacial** é a observação de que localizações de memória com endereços numericamente similares a uma localização de memória cujo acesso foi recente provavelmente serão acessadas no futuro próximo. Caches exploram essa propriedade trazendo mais dados do que os requisitados, na expectativa de poder antecipar requisições futuras.

Localidade temporal ocorre quando localizações de memória recentemente acessadas são acessadas outra vez. isso pode ocorrer, por exemplo, com localizações de memórias próximas ao topo da pilha, ou com instruções dentro de um laço. A localidade temporal é explorada em projetos de cache, principalmente pela escolha do que descartar quando ocorre uma **ausência na cache** (dados necessários não foram encontrados na cache). Muitos algoritmos de substituição de cache exploram a localidade temporal descartando as entradas que não tiveram acesso recente.

Todas as caches usam o seguinte modelo: A memória principal é dividida em blocos de tamanho fixo,

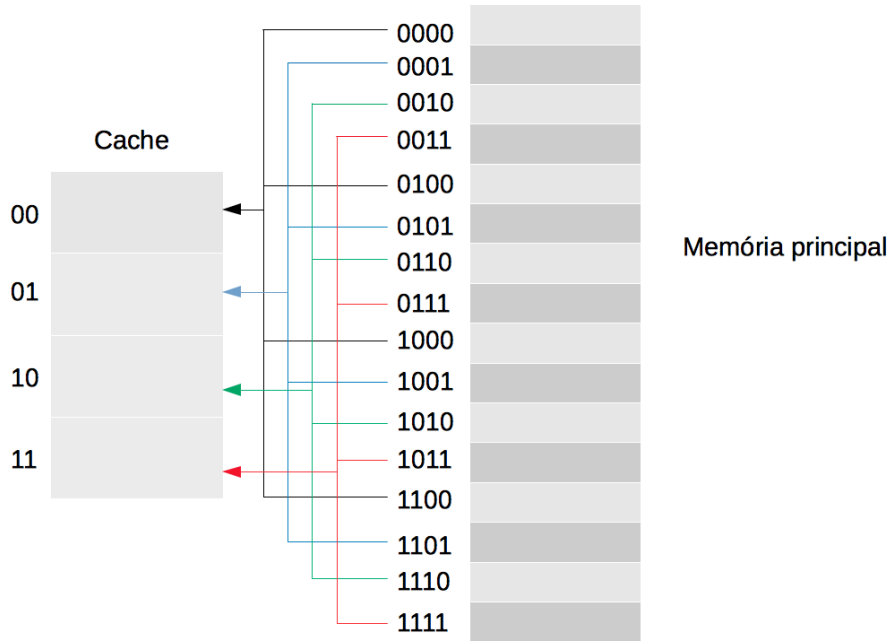
designados **linhas de cache**. Uma linha típica consiste em 4 a 64 bytes consecutivos. As linhas são numeradas em sequência, começando em 0; portanto, se tivermos uma linha de 32 bytes de tamanho, a linha 0 vai do byte 0 ao byte 31, a linha 1 do byte 32 ao 63, e assim por diante. Em qualquer instante, algumas linhas estão na cache.

Quando a memória é referenciada, o circuito de controle da cache verifica se a palavra referenciada está nela naquele instante. Caso positivo, o valor que ali está pode ser usado, evitando uma viagem até a memória principal. Se a palavra não estiver lá, alguma linha de entrada é removida da cache e a linha necessária é buscada na memória ou na cache de nível mais baixo para substituí-la.

Existem muitas variações desse esquema, mas em todas elas a ideia é manter as linhas mais utilizadas na cache o quanto possível, para maximizar o número de referências à memória satisfeitas pela cache.

Caches de mapeamento direto

A cache mais simples é conhecida como cache de mapeamento direto. Um exemplo de cache de mapeamento direto de um só nível é mostrado na Figura abaixo. Esse exemplo contém 4 entradas. Cada entrada (linha) pode conter exatamente uma linha de cache da memória principal. Se a linha tiver 32 bytes de tamanho, para esse exemplo, a cache pode conter 4 entradas de 32 bytes, ou 128 Bytes no total.



Cada entrada de cache consiste em três partes:

1 - O bit **Valid** indica se há ou não quaisquer dados válidos nessa entrada. Quando o sistema é iniciado, todas as entradas são marcadas como inválidas.
2 - O campo **Tag** consiste em um único valor de 4 bits que identifica a linha de memória correspondente da qual vieram os dados.
3 - O campo **Data** contém uma cópia dos dados na memória. Ele contém uma linha de cache de 32 bytes.

Em uma cache de mapeamento direto, uma determinada palavra de memória pode ser armazenada em **exatamente um lugar dentro da cache**. Dado um endereço de memória, há somente um lugar onde procurar por ele. Se não estiver nesse lugar, então ele não está na cache.

Para armazenar e recuperar dados da cache, o endereço é desmembrado em quatro componentes:

1 - O campo **TAG** corresponde aos bits Tag armazenados em uma entrada de cache.

2 - O campo **LINE** indica qual entrada de cache contém os dados correspondentes, se eles estiverem presentes.
3 - O campo **WORD** informa qual palavra dentro de uma linha é referenciada.
4 - O campo **BYTE** em geral não é usado, mas se for requisitado apenas um byte, ele informa qual byte dentro da palavra é necessário.

Quando a CPU produz um endereço de memória, o hardware extrai os bits LINE do endereço e os utiliza para indexá-lo na cache para achar uma das entradas. Se essa entrada for válida, o campo TAG do endereço de memória e o campo Tag na entrada da cache são comparados. Sendo compatíveis, a entrada de cache contém a palavra que está sendo requisitada, uma situação denominada **presença na cache.**

Se ocorrer uma presença na cache, uma palavra que está sendo lida pode ser pega, eliminando a necessidade de ir até a memória. Somente a palavra necessária é extraída da entrada da cache. O resto da entrada não é usado.

Se a entrada for inválida ou os tags não forem compatíveis, a entrada necessária não está presente, uma situação denominada **ausência da cache**. Nesse caso, a linha de cache necessária é buscada na memória e armazenada na linha da cache correspondente, substituindo o que lá estava. Contudo, se a linha de cache existente sofreu modificação desde que foi carregada, ela deve ser escrita de volta na memória principal antes de ser sobrescrita.

A despeito da complexidade da decisão, o acesso à palavra necessária pode ser extraordinariamente rápido. Assim que o endereço for conhecido, a exata localização da palavra é conhecida, se ela estiver presente na cache. isso significa que é possível ler a palavra da cache e entregá-la ao processador ao mesmo tempo em que está sendo determinado se essa é a palavra correta.

O problema é que esse esquema de mapeamento põe linhas de memória consecutivas em linhas de cache consecutivas. Contudo, quando duas linhas de memória com as mesmas linhas de cache precisarem ser armazenadas na cache ao mesmo tempo, ocorre um conflito.

Nesse caso, a segunda instrução forçará a linha de cache a ser recarregada, sobrescrevendo o que lá estava. Se isso acontecer com certa frequência, pode resultar em mau desempenho. Na verdade, o pior comportamento possível de uma cache é ainda pior do que se não houvesse nenhuma, já que cada operação de memória envolve ler uma linha de cache inteira em vez de apenas uma palavra.

Caches de mapeamento direto são as mais comuns e funcionam com bastante eficácia, porque com elas é possível fazer colisões como a descrita ocorrerem apenas raramente, ou nunca ocorrerem. Por exemplo, um compilador muito esperto pode levar em conta as colisões de cache quando colocar instruções e dados na memória. Note que o caso particular descrito não ocorreria em um sistema com caches de instruções e dados separados, porque as requisições conflitantes seriam atendidas por caches diferentes.

Uma possível solução para o problema das colisões é permitir duas ou mais linhas em cada entrada de cache. Uma cache com n entradas possíveis para cada endereço é denominada uma **cache associativa de conjunto** de n vias.

Uma cache associativa de conjunto é inerentemente mais complicada do que uma de mapeamento direto porque, embora a linha de cache correta a examinar possa ser calculada do endereço de memória que está sendo referenciado, um conjunto de n linhas de cache deve ser verificado para ver se a palavra necessária está presente.

Referências bibliográficas:

TANENBAUM, Andrew S. Organização Estruturada de Computadores, 2007, 5ª Edição.

TANENBAUM, Andrew S. AUSTIN, Todd; Organização Estruturada de Computadores, 2013, 6ª Edição.