Memória Cache

O modelo de Von Neumann estabelece que “para ser executado, o programa deve estar armazenado na memória”. A memória é organizada em grupos de *bits* chamados células (ou palavras), onde as informações são armazenadas. Para acessá-las, foi criado um mecanismo de identificação de cada uma destas células denominado **endereço de memória**, através dele podemos acessar (ler ou gravar) qualquer palavra da Memória Principal.

Para que um programa seja executado, é necessário transferir suas instruções da memória para o interior da CPU. Existe na CPU um registrador apontando para a próxima instrução a ser executada. Este registrador é chamado de PC (*Program Counter*).

Os registradores também são palavras de uma memória especial localizada no interior da CPU (denominado memória local) e estão conectados às unidades funcionais da mesma. Eles atuam como operandos fonte e destino das operações realizadas pela CPU.

Há algum tempo atrás, tanto a CPU como as memórias tinham velocidades de trabalho muito próximas uma das outras. Com o passar do tempo, tornou-se necessário memórias com maior capacidade de armazenamento motivando o aparecimento da Memória Dinâmica. Este tipo de memória permitiu aumentar a capacidade de armazenamento, mas limitou sua velocidade. Já na CPU, é utilizada uma tecnologia diferente que permite uma alta velocidade de processamento (tecnologia de Memória Estática). Com isso, a CPU foi se tornando cada vez mais rápida enquanto que as memórias não foram capazes de acompanhar tal aumento de velocidade. Como é necessário trazer informações da memória para os registradores, esta diferença de velocidade acabou se tornando um grande problema.

Para reduzir o efeito causado por esta diferença de velocidade, estudos sobre o comportamento dos programas foram realizados e destes estudos foram estabelecidos dois princípios: Localidade Temporal e Localidade Espacial.

A Localidade Temporal caracteriza aqueles programas que apresentam uma grande quantidade de *loops* (repetição de trechos de códigos) e estabelece que **uma vez acessada uma determinada posição de memória, existe uma grande probabilidade desta posição ser novamente acessada em um curto intervalo de tempo**. Isto é, devido ao fato da grande maioria dos programas serem formados por *loops* e que muitos deles têm um alto grau de repetição, a CPU ficará executando um pequeno grupo de instruções durante longos períodos do tempo de processamento.

O princípio da Localidade Espacial, por outro lado, estabelece que **uma vez que uma determinada posição de memória é acessada, existe uma grande probabilidade de que as posições vizinhas também sejam acessadas**. Isto é, a execução de um programa é automaticamente seqüencial, a não ser que a instrução executada seja um desvio (transferência de controle), a próxima instrução estará no endereço adjacente. Como o programa pode ser formado por uma quantidade muito

maior de outras instruções do que de transferências de controle, então existe uma grande probabilidade da próxima instrução estar no endereço adjacente.

A reduzida capacidade de armazenamento das memórias estimulou a criação de mais um nível na hierarquia do sistema de memória na década de 70. Este nível é formado por uma memória de alta velocidade, mas com pequena capacidade de armazenamento, que é conhecida por Memória *Cache*. Cópias de trechos da Memória Principal podem estar armazenados nesta Memória *Cache* e toda vez que a CPU fizer um acesso de leitura à memória, a *cache* é examinada antes. Se a cópia da palavra estiver na *cache*, dizemos que ocorreu um acerto (*cache hit*) e neste caso o tempo entre o pedido e o recebimento da informação é mais curto. Caso a informação solicitada não esteja na *cache*, dizemos que ocorreu uma falta (*cache miss*) e neste caso o tempo entre o pedido e o recebimento da informação será bem maior que o anterior, pois a palavra será buscada na Memória Principal que continua sendo de baixa velocidade.

As memórias utilizadas nos computadores sempre tiveram um desempenho inferior ao da CPU. Em 1970 surgiram as primeiras memórias de semicondutores facilitando a sua interface e tornando o seu desempenho próximo ao da CPU. O ideal seria que as velocidades das memórias e das CPUs evoluíssem na mesma taxa, mas não foi o ocorrido. Devido a diferente tecnologia utilizada nas memórias para conseguir uma alta densidade de *bits* (DRAM – Memórias Dinâmicas), tivemos uma perda na sua velocidade. Desde 1980, esta diferença de velocidades vem se agravando: a CPU tem seu desempenho aumentado por volta de 60% ao ano enquanto que as memórias DRAM aumentam por volta de 10% ao ano.

Para se ter uma noção da gravidade do problema, em uma estação Alpha 21264 de 500Mhz, uma falha na Memória *Cache* interrompe o processador por 128 ciclos. Para que a Memória *Cache* atinja uma alta velocidade, é utilizada a mesma tecnologia usada na fabricação da CPU (SRAM - Memórias Estáticas). Devido a uma restrição desta tecnologia, não é possível ter uma alta densidade de *bits*, o que resulta na pequena capacidade de armazenamento e na elevada taxa de faltas.

A Memória *Cache* e suas Vantagens

A Memória *Cache* está localizada entre a CPU e a Memória Principal (MP) como mostra a Figura 1.

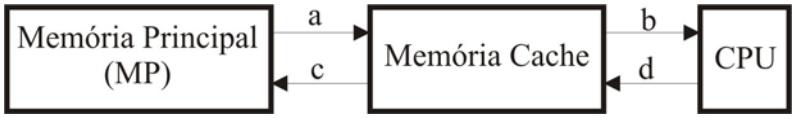

Figura 1: Sistema de Memória

A Memória *Cache* pode ser caracterizada pela:

- **Política de Mapeamento** (Estrutura da *Cache*): esta característica especifica como as palavras da Memória Principal são armazenadas na Memória *Cache*.
  - – Mapeamento Direto
  - – Puramente Associativo
  - – Associativo por Conjunto
- **Política de Substituição**: escolhe a posição da Memória *Cache* cujo conteúdo será substituído pelo de uma outra palavra da Memória Principal.
  - Aleatório
  - FIFO ("*First In First Out*")
  - LFU (*Least Frequently Used* - Menos Freqüentemente Usado)
  - LRU (*Least Recently Used* - Menos Recentemente Usado)
- **Política de Atualização**: define quando os dados alterados pela CPU na Memória *Cache* serão transferidos para a Memória Principal.
  - *Write Through* (Escrita em Ambas)
  - *Write Back* (Escrita Atrasada)
  - *Write Once*

Organização da Memória *Cache*

A *Cache* é formada por linhas (*frame*), cada uma capaz de armazenar uma certa quantidade de células da Memória Principal. Conforme ilustra a Figura 2, cada linha possui alguns *bits* de controle. O *bit Válido*(V) indica se o conteúdo da linha é válido e o *bit Modificado*(M) informa se ela sofreu alguma alteração por parte da CPU. Existe também o campo TAG que contém parte do endereço que indica de que lugar da Memória Principal pertence o dado carregado na linha da *Cache*. Para a Memória *cache* é como se a Memória Principal estivesse dividida em blocos com o mesmo número de células existentes em cada uma de suas linhas.

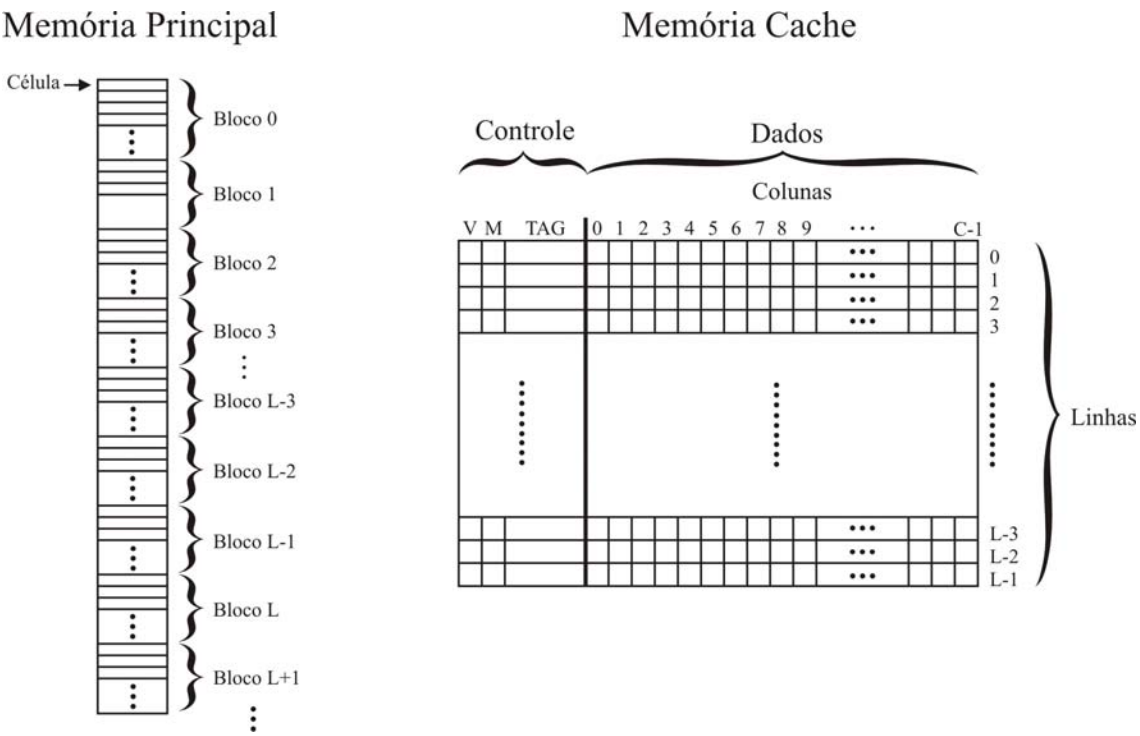

Figura 2: Organização Interna da Memória Cache e da Memória Principal

Memória *Cache* com Mapeamento Direto

Nesta organização, cada bloco da Memória Principal só pode ser carregado numa determinada linha da *Cache*, não sendo possível carregá-lo em uma outra linha. O Bloco "0" é carregado na linha "0", o bloco "1" na linha "1" e assim sucessivamente. Como a Memória Principal é muito maior que a Memória *Cache*, haverá muito mais blocos do que linhas. Neste caso, quando um determinado bloco é mapeado para a última linha da *cache*, o bloco seguinte volta a ser mapeado na primeira linha dando a criação dos ciclos de mapeamento. Veja a Figura 3

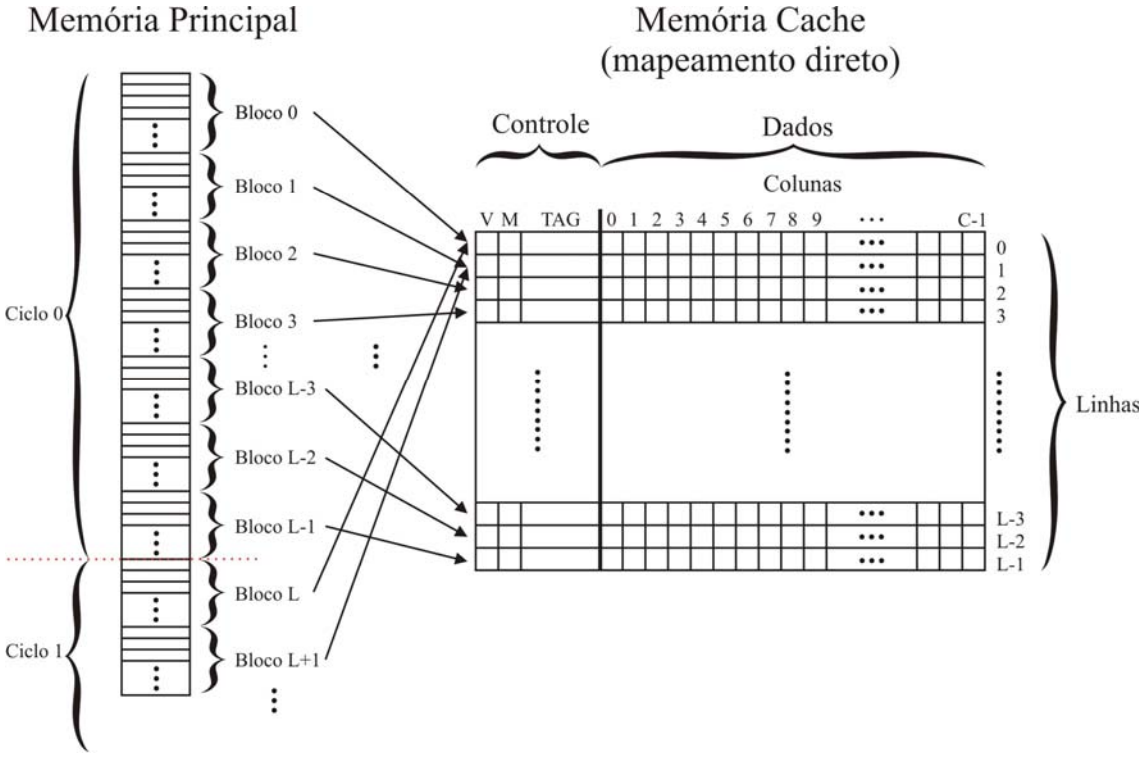

Figura 3: Memória Cache de Mapeamento Direto

Diferentes endereços de blocos de memória podem estar mapeados na mesma linha. Como saber de que região da memória pertence o conteúdo de uma determinada linha da *cache*? É aí que entra o campo TAG. Ele guarda os *bits* mais significativos do endereço que representam o número do ciclo, permitindo que a *cache* identifique com exatidão de que lugar da Memória Principal pertence o conteúdo carregado naquela linha.

Normalmente, o carregamento do bloco na linha se dá sob demanda, isto é, conforme a necessidade. No momento que a CPU solicita o conteúdo de uma determinada célula da Memória Principal, a Memória *Cache* identifica qual é o número do bloco na Memória Principal e examina o TAG da linha correspondente. Se o TAG armazenado na linha coincidir com os *bits* mais significativos do endereço (*TAG_CPU*), teremos um *HIT*, e neste caso a Memória *Cache* fornece o seu conteúdo numa velocidade muito superior caso o acesso tivesse sido feito à Memória Principal. Agora, se o TAG não coincidir com os *bits* mais significativos do endereço (*TAG_CPU*), teremos um *MISS*, e neste caso a Memória *Cache* terá que carregar todo o bloco em questão para a respectiva linha, atualizando o valor do TAG. Repare que neste caso aguardaremos o tempo necessário para acessar a Memória Principal. Outra observação

importante é que ao carregar todo o bloco na linha de *cache*, está sendo feita a suposição, baseando-se no conceito de localidade espacial e temporal, de que todos os dados deste bloco serão utilizados pela CPU. Caso a CPU não os utilize, teremos desperdiçado tempo e espaço de *cache*.

No Mapeamento Direto, se um processo referenciar repetidamente endereços de memória que são mapeados na mesma linha, porém pertencentes a localizações diferentes (ciclos diferentes), haverá uma grande ocorrência de *misses* forçando ao repetido carregamento destes dados, provocando uma grande queda no desempenho do processador. Este problema é conhecido como *Ping-Pong*. Por exemplo: suponha que dentro de um determinado *looping*, existam chamadas a várias funções ou procedimentos do programa. Se houver a coincidência de duas ou mais destas funções ou procedimentos estarem mapeadas na mesma linha da Memória *Cache*, haverá a ocorrência de diversos *misses* devido à alternância entre estas funções ou procedimentos.

Na política de mapeamento direto, o endereço do programa é dividido em três partes: deslocamento dentro da linha (*Coluna*), número da linha e TAG. O número da linha junto com o valor do TAG formam o endereço na Memória Principal de um bloco de células. Veja a Figura 4.

Figura 4: Formato do Endereço da Memória Principal visto por uma Memória Cache com Mapeamento Direto

Memória *Cache* Puramente Associativa

Neste tipo de *cache*, o bloco da Memória Principal pode ser carregado em qualquer uma das linhas existentes (veja a Figura 5). Esta versatilidade resolve o problema de endereços conflitantes mencionado na Memória *Cache* de Mapeamento Direto. Sendo assim, a variedade de blocos da Memória Principal que podem vir a ser carregados em uma determinada linha é bem maior que a da *cache* de Mapeamento Direto, necessitando então de um TAG com mais bits. Enquanto no Mapeamento Direto o TAG armazenava o número do ciclo, neste caso, o TAG irá armazenar o endereço do bloco, já que pode ser carregado em uma determinada linha qualquer bloco da Memória Principal.

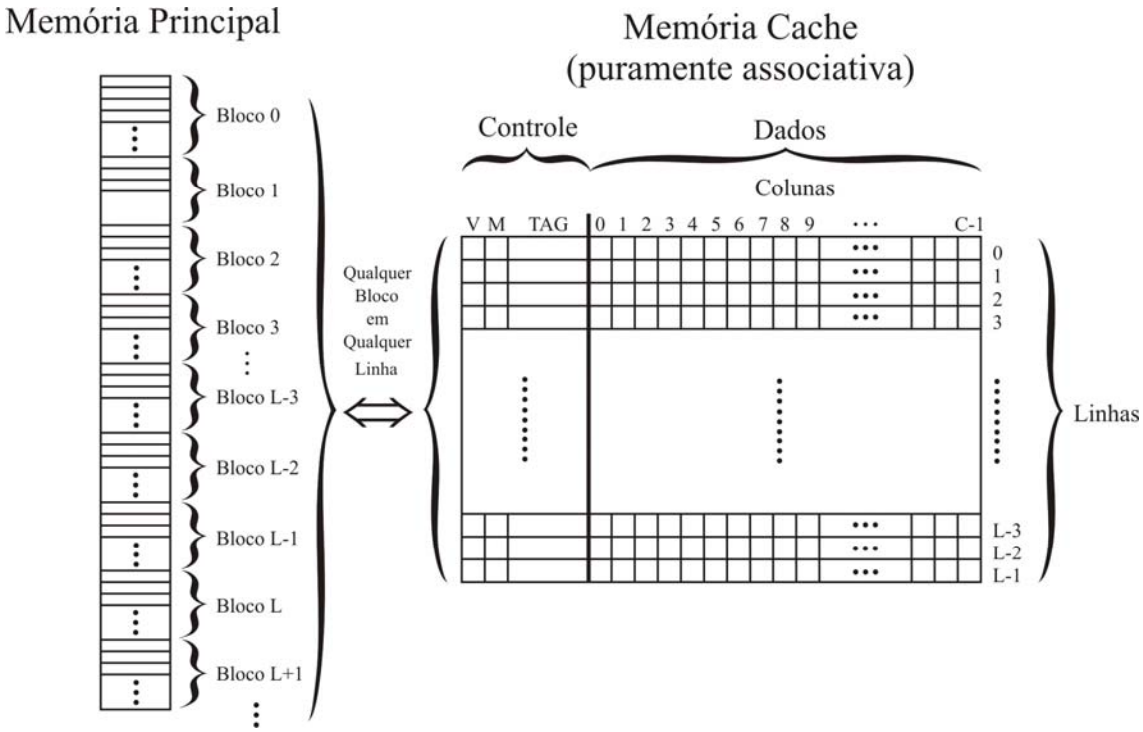

Figura 5: Memória Cache Puramente Associativa

O carregamento do bloco da Memória Principal na linha da *cache* se dá sob demanda. Quando a CPU solicita o conteúdo de uma determinada célula da Memória Principal, a Memória *Cache* verifica se algum TAG já contém o endereço. Como o bloco pode estar carregado em qualquer linha, a Memória *Cache* (que é do tipo associativa) compara o endereço do bloco, com os TAGs de todas as linhas simultaneamente. Caso ocorra alguma coincidência, então teremos um *hit*, e neste caso a *cache* fornece o conteúdo correspondente numa velocidade muito superior ao da Memória Principal. Contudo, se nenhum TAG contiver o endereço do bloco, então teremos um *miss*, forçando a Memória *Cache* a carregar este bloco em uma de suas linhas. Diferente do Mapeamento Direto esta *cache* possui uma vasta opção de linhas para carregar o bloco, sendo então necessário estabelecer um método de escolha da linha. É aí que foram criadas as políticas de substituição.

No mapeamento Puramente Associativo, o endereço da Memória Principal é dividido em dois campos: deslocamento dentro da linha (Coluna) e o TAG. Veja a Figura 6.

Figura 6: Formato do Endereço da Memória Principal Visto por uma Memória Cache Puramente Associativa

Políticas de Substituição

As políticas de substituição têm por objetivo escolher qual das linhas terá o seu conteúdo descartado para que um outro bloco seja carregado. A esta linha damos o nome de Linha Vítima. Como na maioria das vezes, fazer o carregamento de um bloco implica no descarregamento das informações referentes à linha vítima, então este método de escolha irá influenciar no desempenho da Memória *Cache*. A seguir temos quatro métodos de escolha da linha vítima.

- Aleatório: Este método faz uma escolha casual de uma linha para ser a linha vítima. Como este método não leva em consideração os acessos anteriores à Memória *Cache*, existirá uma grande chance de se fazer uma má escolha.
- FIFO: Este método faz a escolha da linha vítima de acordo com a ordem de carregamento (*First-in First-out*). O método de escolha é simples e é levado em consideração a ordem de carregamento, que só é atualizado quando a linha for carregada, isto é, na ocorrência de um *miss*.
- LFU (Menos Freqüentemente Usada): Este método usa como base para a escolha da linha vítima a quantidade de vezes que as linhas foram referenciadas. A linha que foi referenciada menos vezes, é a linha vítima. Este método usa como critério a freqüência de acessos a uma determinada linha.
- LRU (Menos Recentemente Usada): O critério de escolha deste método é a ordem dos acessos feitos às linhas de *cache*. A linha que não é acessada a mais tempo é a candidata a linha vítima.

Memória *Cache* Associativa por Conjunto

Neste tipo de *cache*, as suas linhas são agrupadas formando diversos conjuntos de linhas dentro da *cache*. O bloco da Memória Principal será mapeado para um desses conjuntos de forma semelhante ao Mapeamento Direto. O bloco 0 será carregado no conjunto 0, o bloco 1 no conjunto 1, e assim sucessivamente. Quando o bloco referente ao último conjunto da *cache* for carregado, o bloco seguinte voltará para o conjunto "0" iniciando mais um ciclo. Veja a Figura 7

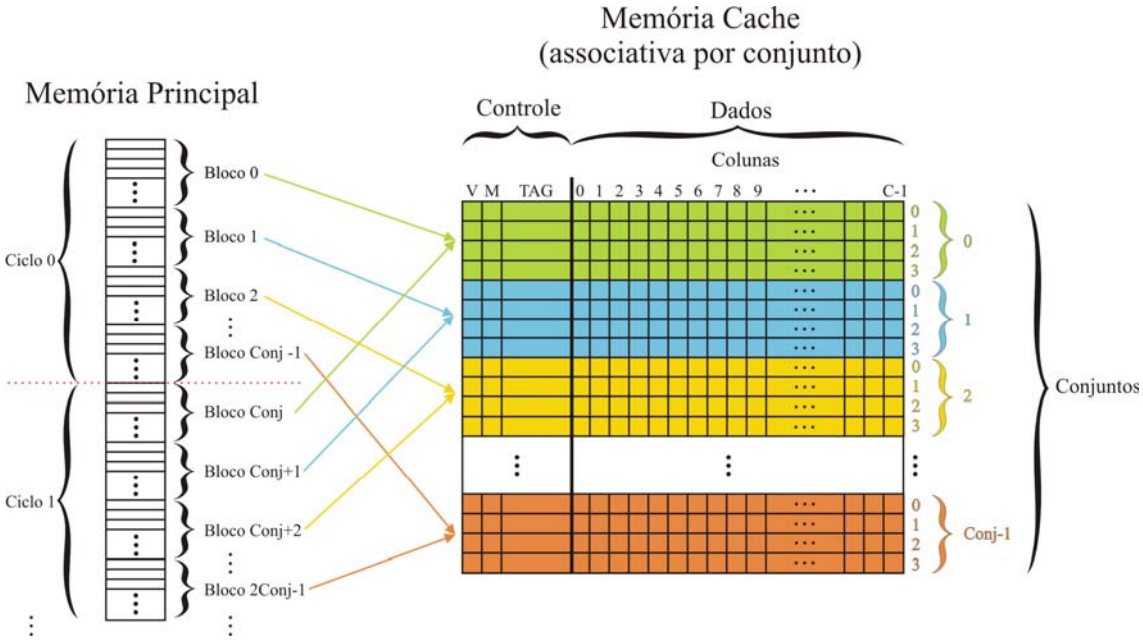

Figura 7: Memória Cache Associativa por Conjunto com grau de associatividade igual a quatro (four-way set associative)

Diferentes endereços de blocos de memória podem estar mapeados no mesmo conjunto. Neste tipo de *cache*, a TAG guarda os bits mais significativos do endereço que representam o número do ciclo, permitindo identificar de que posição da Memória Principal pertence o conteúdo carregado naquela linha dentro daquele conjunto.

Considerando que este mapeamento é feito para um conjunto de linhas, então, poderão ser carregados a mesma quantidade de blocos que linhas existentes no conjunto, diminuindo a possibilidade de *misses* na *cache* provocadas por colisão. Dentro de um determinado conjunto, o bloco pode ser carregado em qualquer uma das linhas. Apesar desta *cache* não ser tão versátil quanto a Puramente Associativa, ela resolve o problema de endereços conflitantes mencionado na Memória *Cache* de Mapeamento Direto. Sendo assim, a variedade de blocos da Memória Principal que podem ser carregados em uma determinada linha é bem inferior que a da *cache* Puramente Associativa, necessitando então de um TAG com uma capacidade menor de *bits*.

O carregamento do bloco na linha se dá sob demanda. No momento que a CPU solicita o conteúdo de uma determinada célula da Memória Principal, a Memória *Cache* identifica qual é o número do bloco, o número do conjunto e o número do TAG (ciclo). Como este bloco pode estar carregado em qualquer linha do respectivo conjunto, a Memória *Cache*, através de um circuito do tipo associativo, compara os *bits* mais

significativos do endereço, que representam o número do ciclo (TAG_CPU), com os TAGs de todas as linhas do grupo ao qual ele está mapeado. Caso haja alguma coincidência, então teremos um *hit*, e neste caso a *cache* fornece o seu conteúdo numa velocidade muito superior caso o pedido tivesse sido feito à Memória Principal. Agora, se não for encontrado nenhum TAG igual aos bits mais significativos do endereço (TAG_CPU), teremos um *miss*, forçando a Memória *Cache* a carregar este bloco em alguma das linhas do conjunto. Como esta Memória *Cache* possui varias linhas para colocar o mesmo bloco, também será necessário estabelecer alguma política de substituição.

Neste tipo de Memória *Cache*, o endereço da Memória Principal será dividido em três partes: deslocamento dentro da linha (Coluna), número do conjunto e o TAG. O número do conjunto junto com o valor do TAG forma o número do bloco. Veja a Figura 8.

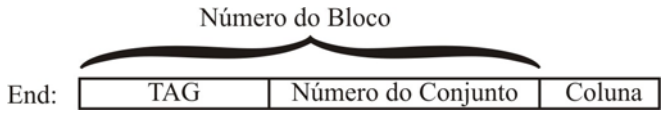

Figura 8: Formato do Endereço da Memória Principal visto por uma Memória Cache Associativa por Conjunto

Políticas de Atualização

Como as escritas feitas pela CPU são inicialmente realizadas na Memória *Cache*, de alguma forma estas informações deverão ser repassadas para a Memória Principal. Estas formas de repassar são chamadas de políticas de atualização. Vou citar três formas de realizar estas atualizações:

- *Write Through*: Este método faz a atualização da Memória Principal no momento em que foi feita a escrita na Memória *Cache*. A vantagem deste método é de manter a Memória Principal sempre atualizada, só que para cada escrita feita pela CPU, será consumido o tempo referente ao da Memória Principal. A vantagem da *cache* será apenas para as operações de leitura que por serem muito mais freqüentes, ainda terá algum ganho com este método, mas, suponha o seguinte caso: um determinado programa possui um *looping* baseado em uma variável que é alterada a cada iteração, por exemplo, a variável "i". Suponha que devido a um grande número de variáveis existentes no *looping*, a variável "i" não pode ser mantida em registrador tendo que ter seu conteúdo transferido para a Memória *Cache*. Neste tipo de política de atualização, a cada alteração feita na variável "i" também será feita uma alteração na Memória Principal levando a uma perda de tempo relativamente grande que poderá se agravar mais ainda se este *looping* for muito repetitivo. O problema mencionado anteriormente não se aplica a Memória *Cache* de instruções (*I-cache*), já que não se faz escritas nas áreas de código. Se por ventura acontecer de haverem escritas, estas serão uma raridade dentre as operações de leitura.
- *Write Back*: Para conseguir aumentar o desempenho nas operações de escrita e com isso resolver o caso mencionado na técnica *Write Through*, esta memória atrasa as atualizações na Memória Principal, mantendo os dados mais recentes apenas na Memória *Cache*. A atualização será realizada somente no momento em que a respectiva linha tiver que ser substituída por outra, a não ser que ela não tenha sido alterada pela CPU. Um problema resultante desta política de

atualização é a possibilidade da Memória Principal não ter as mesmas informações que a Memória *Cache*. Se houver algum outro dispositivo neste equipamento que solicite informações à Memória Principal, ele corre o risco de receber dados desatualizados. Uma forma de resolver este problema é selecionar algumas áreas da Memória Principal para serem do tipo não "cacheável", isto é, seus conteúdos não são enviados para a Memória *Cache*. Desta forma podemos usar esta região de memória para transferências entre dispositivos que compartilham o uso da Memória Principal, por exemplo, o controlador de DMA (Acesso Direto à Memória).

- *Write Once*: Esta política de atualização é uma mistura dos dois métodos anteriores e tem como objetivo trazer vantagens tanto nas operações de leitura como nas operações de escrita em sistemas com mais de uma CPU. Este é um caso em que mais de um dispositivo está acessando a Memória Principal. Para que não aconteça o envio de uma informação errada para uma das CPUs, a Memória *Cache* utiliza do recurso da primeira escrita para avisar as demais *caches* do sistema que aquele bloco da Memória Principal passará a ser de uso exclusivo dela. A partir deste momento, todas as operações de escrita e leitura naquele bloco da Memória Principal ficarão restritos a esta *cache*. Quando alguma outra CPU necessitar das informações deste bloco, será feita uma solicitação de atualização à *cache* portadora do bloco. Esta *cache* irá então atualizar a Memória Principal possibilitando que as demais CPUs recebam o dado mais recente.