

1 Arquitectura

Arquitectura MIPS Tipos de datos Integer: Bytes: 8 bits Half word: 16 bits (Se agrego porque se usa en C y en algunos programas populares) Words: 32 bits Doble word: 64 bits Tipos de datos Floating Point: Single precision: 32 bit Doble precision: 64 bit

MIPS64 trabaja con 64 bits para integers y 32 o 64 para Floating Point.

Modos de direccionamiento:

Mas dos modos de direccionamiento utilizando el ZERO Indireccion de registro: Coloca un Zero en el campo de desplazamiento.

Dirreccionamiento absoluto: Utiliza el campos de 16bits de desplazamiento y colocando Zero como registro base.

Formato de instruccion MIPS

Tipos de instruccion I

2 Jerarquia de Memoria

La Jerarquia de Memoria, toma las ventajas del principio de localidad y el costo de performance de las memorias. El principio de localidad dice que los programas no acceden al todo el codigo o datos uniformemente.

Tipico nivel de Jerarquia:

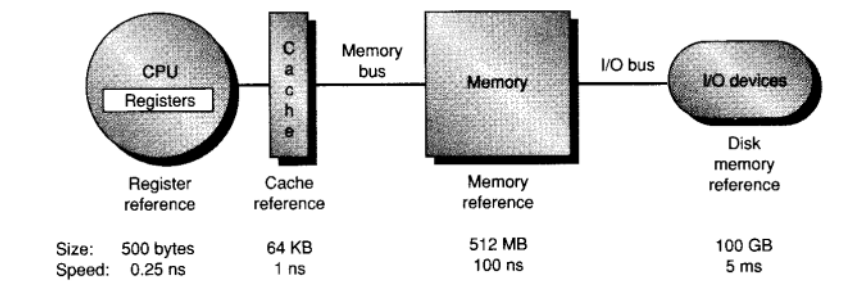


Figure 5.1 The levels in a typical memory hierarchy in embedded, desktop, and server computers. As we move farther away from the CPU, the memory in the level below becomes slower and larger. Note that the time units change by factors of 10—from picoseconds to milliseconds—and that the size units change by factors of 1000—from bytes to terabytes. Figure 5.3 shows more parameters for desktops and small servers.

Como la memorias rapidas son caras, la Jerarquia de Memoria esta organizada en distitos niveles.

El objetivo es proveer un sistema de memoria con costos tan bajos como el nivel de memoria mas barato y velocidad casi tan tapidos como el nivel mas

veloz.

2.1 Review Caches

A continuación algunos terminos utilizados:

Cache: Es el primer nivel de la Jerarquia.

1. Se utiliza cuando sea necesario un buffer para reusar items que ocurren comunmente.

Cache Hit: Cuando la CPU requiere un dato y lo encuentra como item en la cache.

Cache Miss: Cuando la CPU requiere un dato y NO lo encuentra como item en la cache.

1. El tiempo requerido depende de la latencia y el ancho de banda de la memoria.
2. Es manejado por el hardware.

Bloque: Una coleccion de datos con tamaño fijo que contiene la palabra requerida, es recuperada de memoria y situada en la cache.

Localidad temporal: Nos dice que podemos probablemente vamos a necesitar el mismo datos en un futuro cercano. Entonces es util dejarlo en la cache para utilizarlo rapidamente.

Localidad espacial: Nos dice que hay altas posibilidades de utilizar pronto otro dato del mismo bloque.

Paginas: Espacios de memoria separados en bloques de tamaño fijo.

1. Una pagina puede recibir tanto en memoria como en disco.
2. Cuando el CPU hace referencia a un item dentro de una pagina que no esta presente (not present) en la cache, or memoria principal, se produce un PAGE FAULT, y la pagina entera es movida del disco a memoria principal.
3. Como los page fault toman mucho tiempo, estos se manejan por software y la CPU no se detiene(STALLED)
4. La cache y la memoria principal tienen la misma relacion en memoria y en disco.

2.2 Perfomance de Cache

Para evaluar la perfomance de la cache expandiremos el uso de la ecuación de CPU Execution Time.

Memory Stall Cycles: Número de ciclos que necesita esperar la CPU para acceder a la memoria.

$$CPU_{exe} = (CPU_{clockcycles} + MemoryStallcycles) * Clockcyclotime. \quad (1)$$

Memory Stall cycles depende de la cantidad de misses y el costo por miss.

$$MemoryStallcycles = NumberofMisses * MissPenalty \quad (2)$$

$$MemoryStallcycles = IC * \frac{Misses}{Instruction} * MissPenalty \quad (3)$$

$$MemoryStallcycles = IC * \frac{MemoryAccess}{Instruction} * MissRate * MissPenalty \quad (4)$$

Donde IC es Instruction Count.

En la ultima formula los componentes se pueden medir facilmente. La cantidad de instrucciones (**IC**) se puede medir Para medir las referencias a memoria (**MemoryAccess**), cada instrucción requiere un acceso a memoria (Instruction Access) y dependiendo de la instrucción se cuentan los accesos a datos en memoria (Data Access).

Calculamos el **MissPenalty** como un promedio.

El componente **MissRate** es simplemente la fraccion entre los accesos a cache que resultaron en misses dividido el numero de accesos totales

El calculo de **MemoryStallCycle** puede definirse en terminos de número de accesos a memoria por instrucción, *miss penalty* para lectura y escritura, y el *miss rate* para lectura y escritura:

$$MemoryStallclockcycles = IC * Instr.delectura * MissRateLectura * MissPenaltyLectura + IC * Instr.deEscritura * MissRateEscritura * MissPenaltyEscritura$$

Para simplificar la formula, combinamos las lecturas y escrituras y buscamos un promedio para el miss rate y el miss penalty de escritura y lectura.

$$MemoryStallcycles = IC * \frac{MemoryAccess}{Instruction} * MissRate * MissPenalty \quad (5)$$

El miss rate es una de las medidas mas importantes de diseño de cache.

2.3 Categorías de organizacion de Cache

En la imagen de abajo se muestra las restricciones cuando se quiere situar un bloque en cache.

- Si cada bloque tiene solo un lugar donde situarse en la cache, se dice que la cache es *Direct Mapped*.
- Si cada bloque puede situarse en cualquier lugar de la cache, se dice que la cache es *Fully Associative*.
- Si cada bloque puede situarse solo en algunos lugares restringidos de la cache, se dice que la cache es *Set Associative*. Si hay N bloques en un conjunto se dice que la cache es *N-Way Set Associative*.

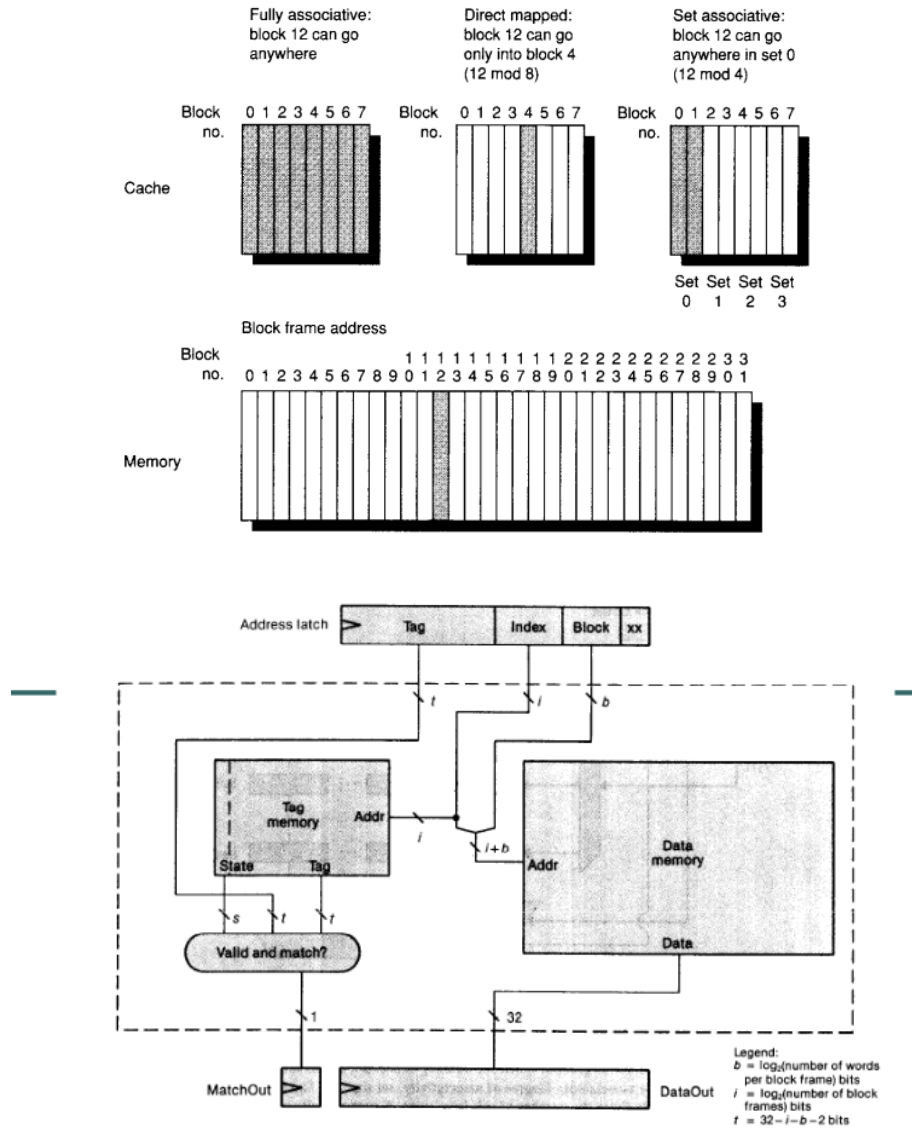


Figure 2. A direct-mapped cache. The access logic (hit, not miss logic) for a direct-mapped cache using bit selection to select the set (block frame) of the reference has three components. The first component, the data memory, holds all cached data and instructions. The second component, tag memory, holds the state bits and address tag associated with a cached block. The last component, the match logic, produces a single bit indicating whether the referenced block is present.

2.4 ¿Como se ubica un bloque en la Cache?

La cache tiene un tag de direcciones (*address tag*) en cada bloque de frame que te da la direccion del bloque (*block address*). El tag de cada bloque de cache

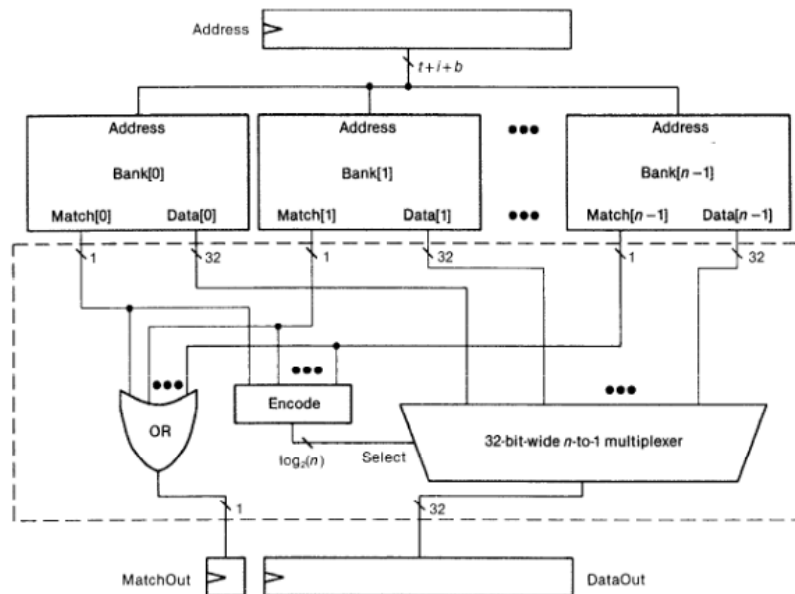
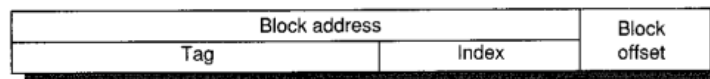


Figure 3. A set-associative cache. Cache access (hit) logic for an n -way set-associative cache of c blocks consists of n banks and the logic to combine bank results. Each bank can be thought of as a direct-mapped cache of c/n blocks and can be implemented using the logic in the dashed box of Figure 2.

debe tener toda la informacion necesario para validar si este coincide con la direccion del bloque provisto por la CPU. Como regla todos los tags deben ser buscados en paralelo.



Para saber si el bloque de cache no tiene información valida se utiliza un bit de validez junto al tag para saber si o no contiene una dirección valida.

Exploremos la relación entra la dirección de la CPU (*CPU Address*) y la cache. La direccion se divide entre *block address* y *block offset*. A su vez el el block address se puede dividir en *tag field* y *index field*. El block offset seleccionar el dato deseado dentro del bloque, el campo indice selecciona el conjunto, y el campo tag field es comparado con la porcion que corresponde de la dirección para buscar un hit.

El indice es usado para seleccionar el conjunto. Pero el caso de unaa cache full asociativa, no tiene indice.

2.5 ¿Que bloque debe ser reemplazado ante un Cache Miss?

Cuando un ocurre un cache miss, el controlador de la cache debe seleccionar un bloque para ser reemplazado con el dato deseado. Para una cache *direct map* el hardware requerido para la decision se simplifica ya que no requiere elección. Con las caches *full associative* y *set associative* hay muchos bloques para elegir frente a un cache miss. Hay tres estrategias para seleccionar que bloque reemplazar:

- *Random* - El bloque candidato es seleccionado aleatoriamente
- *Least-recently used (LRU)* - Si el bloque recientemente usado es el mas probable para ser usado de nuevo, entonces un buen candidato para ser desechado es el bloque menos recientemente usado (*the least-recently used block*).
- *First in, first out(FIFO)* - Se reemplaza por el bloque mas viejo. Es una aproximación a LRU ya que es complicado de implementar.

2.6 ¿Que pasa ante una escritura?

Las lecturas dominan los accesos a cache del procesador. Todos los accesos son lecturas y muchas intrucciones no escriben en memoria. Haciendo que los casos comunes sean rapidos, estamos optimizando las caches para lecturas, especialmente cuando los procesadores tradicionales esperan para completar una lectura pero no necesitan esperar por escrituras.

Los bloques pueden ser leidos de la cache al mismo tiempo que es leído el *tag* y comparado. Entonces la parte solicitada del bloque es pasado al CPU inmediatamente. Si la lectura es un miss, no hay un beneficio en este paralelismo.

Esta optimización no se puede realizar con escrituras. No se puede modificar un bloque hasta que se haya validado el tag para ver si la direccion es un hit. El chequeo del tag no puede hacerse en paralelo, ya que las escrituras normalmente llevan mas tiempo que las lecturas. Otra complicación es que el procesador tambien especifica el tamaño del dato a escribir, usualmente entre 1 y 8 bytes; solo esa porción del bloque puede ser modificado. Al contrario que las lecturas, estas pueden acceder a mas bytes de los que son necesario.

Las politicas de escritura a menudo distinguen dos diseños de cache.

- *Write through* - La información es escrita en ambos bloques, en la cache y en el bloque de la memoria mas baja (*lower-level memory*).
- *Write back* - La información es escrita solo en el bloque de la cache. *El bloque de cache modificado es escrito en memoria solo cuando esta es reemplazada.*

Para reducir la frecuencia de remplazo de bloques con politica write back, se utiliza una marca de limpieza (**dirty bit**). El estado de este bit indica si el bloque esta *dirty* (bloque modificado mientras estuvo en cache) o *clean* (no fue

modificado). Si esta marca esta en clean, no es necesario escribir el bloque en memoria ante un miss, porque la información contenida en el cache es idéntica al bloque encontrado en los niveles mas bajos.

Tanto write back y write through tienen sus *ventajas*. Con write back las escrituras ocurren a la misma velocidad que la memoria cache, y si hay múltiples escrituras dentro del bloque requerido, solo se requiere una escritura en el nivel mas bajo de memoria. Write back usa menos ancho de banda ya que no todas las escrituras van a memoria. Y también se ahorra energía al usar en menor medida el resto de la jerarquía de memoria y menos buses de memoria.

Write through es fácil de implementar que write back. La cache siempre esta clean. Además tiene la ventaja de que el siguiente nivel de memoria mas bajo se tiene la copia del dato mas actualizado, lo cual se simplifica la coherencia de datos.

Dado que a veces el dato no es necesario en una escritura, hay dos opciones durante un *write miss*

- *Write allocate* - El bloque es alocado ante un write miss, seguido de un write hit por ensima. En este caso, *el write miss actua como un read miss*.
- *No-write allocate* - Ante un write miss, este no afecta la cache. El bloque es modificado solo en la memoria de nivel mas bajo.

Normalmente write back caches usa write allocate, suponiendo que el sub-siguiente write al bloque sea capturado por la cache. Write through cache se usa con no-write allocate y la razón es que si hay subsecuentes writes al bloque, los writes van a ir a la memoria de mas bajo nivel, y no hay ganancia al tenerlos cache.

2.7 Un ejemplo: Alpha 21264 Data Cache

La cache contiene 64 Kbytes de datos y 64 bytes por bloque.

2.8 Cache Performance

Para tener una medición en una jerarquía de memoria se usa el *average memory access time*:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} * \text{Miss Penalty} \quad (6)$$

1. Hit time: El tiempo que toma un hit en la cache.

La medida de *average memory access* es aún una medición indirecta de la performance.

Esta formula nos puede ayudar a decidir entre una cache dividida ó unificada.

El comportamiento de la cache puede tener un enorme impacto en la performances.

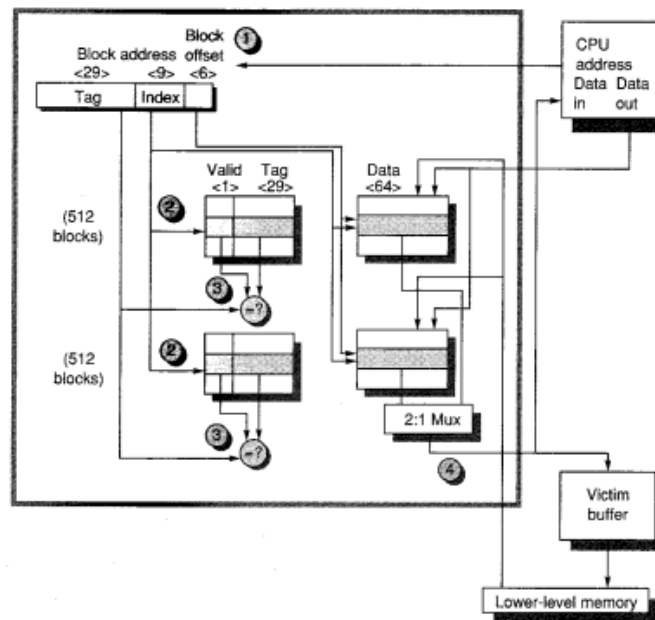


Figure 5.7 The organization of the data cache in the Alpha 21264 microprocessor. The 64 KB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache. The size of address leaving the CPU is 44 bits because it is a physical address and not a virtual address. Figure 5.36 on page 466 explains how the Alpha maps from virtual to physical for a cache access.

Aunque minimizar el *average memory access* es un objetivo razonable, el objetivo final es reducir el tiempo de ejecución de la CPU.

Resumen de formulas:

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{L1}}{\text{Instruction}} \times \text{Hit time}_{L2} + \frac{\text{Misses}_{L2}}{\text{Instruction}} \times \text{Miss penalty}_{L2}$$

Figure 5.9 Summary of performance equations in this chapter. The first equation calculates the cache index size, and the rest help evaluate performance. The final two equations deal with multilevel caches, which are explained early in the next section. They are included here to help make the figure a useful reference.

3 Memoria Virtual

En una computadora para un instante de tiempo pueden ejecutarse multiples procesos, cada uno con su propio espacio de memoria. Muchos procesos usan solo una pequeña parte de su espacio de direcciones. La memoria virtual, divide la memoria fisica en bloques y los aloca para diferentes procesos. Intrinsecamente se puede intepretar como un esquema de proteccion para que un proceso este restringido solo a sus bloques alocados. Tambien reduce el tiempo de arranque de los programas, desde que no se necesita que todo el código y datos esten en la memoria fisica.

La memoria virtual maneja de manera automatica los dos niveles de jerarquia de memoria representados por la memoria principal y los dispositivos de almacenamiento secundarios.

La memoria virtual implementa la traducción del espacio de direcciones de un programa a direcciones fisicas. Este proceso de traducción asegura del espacio de direcciones de un programa de otros. Ademas permite que un simple programa de usuario exceda el tamaño de la memoria principal

Ademas de proveer protección para la memoria compartida y el manejo automatico de la jerarquia de memoria, tambien simplifica la carga de programas para su ejecución. Llamamos *relocación* al mecanismo que permite que el mismo programa se ejecutó en cualquier locación de la memoria fisica.

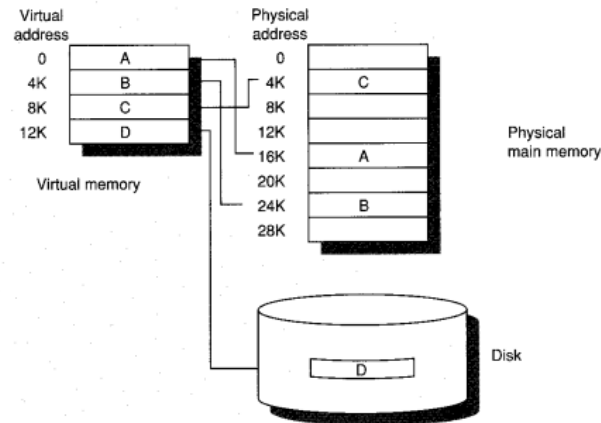


Figure 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

Varias ideas extraídas de la jerarquía de memorias con caches se pueden aplicar análogamente a memorias virtuales, aunque los términos sean diferentes. Las páginas (*pages*) o segmentos (*segment*) serían los bloques, y los *page fault* o *address fault* actuarían como un miss. Con memoria virtual, la CPU produce *direcciones virtuales*, las cuales son traducidas mediante una combinación de hardware y software a *direcciones físicas*. Este proceso se llama *memory mapping* o *address translation*.

Hay más diferencias entre caches y la memoria virtual:

1. El reemplazo ante un cache miss es controlado principalmente por hardware, mientras que la memoria virtual el reemplazo es controlado principalmente mediante el sistema operativo.
2. El tamaño de direcciones del procesador determina el tamaño de la memoria virtual, pero el tamaño de la cache es independiente del tamaño de direcciones del procesador.

Existen dos categorías de memoria virtual. Bloques de tamaño fijo llamados *páginas* ó *pages* y Bloques de tamaños variable llamados *segmentos* o *segment*

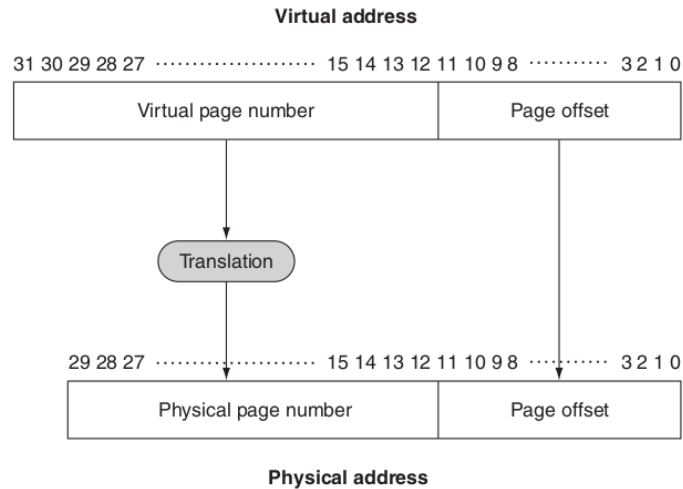


FIGURE 7.20 Mapping from a virtual to a physical address. The page size is $2^{12} = 4$ KB. The number of physical pages allowed in memory is 2^{18} , since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GB, while the virtual address space is 4 GB.

| Parameter | First-level cache | Virtual memory |
|-------------------|--|--|
| Block (page) size | 16–128 bytes | 4096–65,536 bytes |
| Hit time | 1–3 clock cycles | 50–150 clock cycles |
| Miss penalty | 8–150 clock cycles | 1,000,000–10,000,000 clock cycles |
| (access time) | (6–130 clock cycles) | (800,000–8,000,000 clock cycles) |
| (transfer time) | (2–20 clock cycles) | (200,000–2,000,000 clock cycles) |
| Miss rate | 0.1–10% | 0.00001–0.001% |
| Address mapping | 25–45 bit physical address to 14–20 bit cache address | 32–64 bit virtual address to 25–45 bit physical address |

Figure 5.32 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10–1,000,000 times over cache parameters. Normally first-level caches contain at most 1 MB of data, while physical memory contains 32 MB to 1 TB.

3.1 Cuatro preguntas sobre Jerarquía de Memoria

3.1.1 ¿Dónde puede ubicarse los bloques en la memoria principal?

El sistema operativo permite que los bloques puedan ser situados en cualquier lugar en la memoria principal.

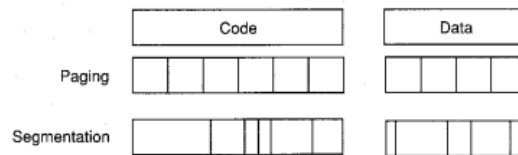


Figure 5.33 Example of how paging and segmentation divide a program.

| | Page | Segment |
|-------------------------|---|---|
| Words per address | One | Two (segment and offset) |
| Programmer visible? | Invisible to application programmer | May be visible to application programmer |
| Replacing a block | Trivial (all blocks are the same size) | Hard (must find contiguous, variable-size, unused portion of main memory) |
| Memory use inefficiency | Internal fragmentation (unused portion of page) | External fragmentation (unused pieces of main memory) |
| Efficient disk traffic | Yes (adjust page size to balance access time and transfer time) | Not always (small segments may transfer just a few bytes) |

Figure 5.34 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: Segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

3.1.2 ¿Como es encontrado un bloque si este esta en memoria principal?

En ambos casos, paginado o segmentacion, se confia en la estructura de datos que esta indexada por la *page number* o *segment number*. Esta estructura de datos contiene la *dirección fisica del bloque*. En segmentacion, el offset es adicionado a la dirección fisica del segmento para obtener la dirección fisica final. En el caso del paginado, el offset es concatenado a la dirección de la pagina fisica (*physical page address* o *physical page number* - *PPN*).

La estructura de datos contiene el *physical page address* que se obtiene de la *page table*. Indexada por la *virtual page number* o *VPN* el tamaño de la tabla es el numero de paginas en el espacio de memoria virtual.

Para reducir el tiempo de traducción, las computadores utilizan un cache dedicada para la traducción de direcciones, llamada *translation lookaside buffer* o *TLB*.

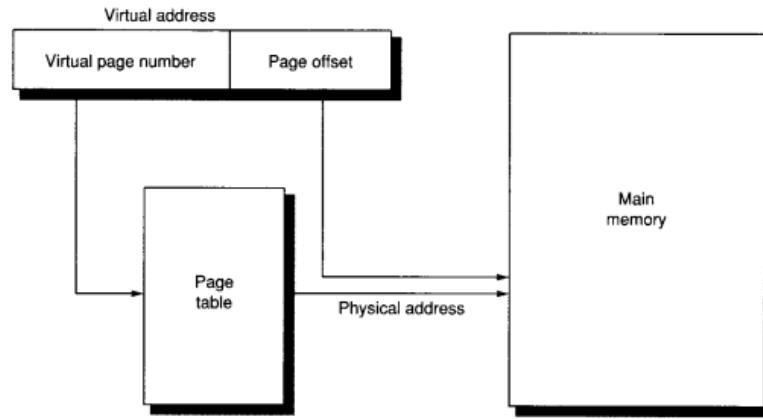


Figure 5.35 The mapping of a virtual address to a physical address via a page table.

3.1.3 ¿Que bloque debe ser reemplazado ante un miss de memoria virtual?

Casi todos los sistemas operativos tratan de reemplazar el bloque menos recientemente usado (LRU). Para estimar el LRU muchos procesadores proveen un *bit de uso* o *bit de referencia*, el cual se setea logicamente cuando una pagina es accedida. El sistema operativo limpia de forma periodica el bit de uso que luego son escritos mas tarde, y entonces con esto se pueden determinar cuales paginas fueron accedidas durante un periodo de tiempo particular. De esta forma, el sistema operativo puede seleccionar una pagina de entre los menos recientemente usados(LRU).

3.1.4 ¿Que pasa ante una escritura?

La estrategia de escritura es simple Write Back. La memoria virtual incluye un *dirty bit*. Este bit permite que los bloques sean escritos a disco solo si estos tuvieron cambios desde que fueron leidos desde disco.

3.2 Técnicas para una rápida traducción de direcciones

Las tablas de paginas usualmente son tan largas que son guardadas en memoria principal y hasta incluso son paginas por ellas mismas. Pagar significa que cada acceso a memoria se hace dos veces, con un acceso a memoria se obtiene la dirección física y con un segundo acceso se obtiene el dato.

Solución, recordar la ultima traducción y de esta forma evitar el proceso de traducción si la dirección que se quiere referencias es la misma pagina que se accedio como la última.

Si se mantiene al direccion traducida en una cache especial, un acceso a memoria rara vez requerira un segundo acceso a memoria para traducir el dato. Esta cache especial es la *TLB* o *translation lookaside buffer*

Una entrada en la TLB es muy parecida a una entrada de cache. El Tag es una porcion de la memoria virtual y la porcion de data que mantiene es la *physical page number*, *protection field*, *valid bit*, y tambien un *use bit* y *dirty bit*.

Notar que *bit de dirty* corresponde a que la pagina esta sucia o dirty, no coorresponde a la traducción en entrada en la TLB.

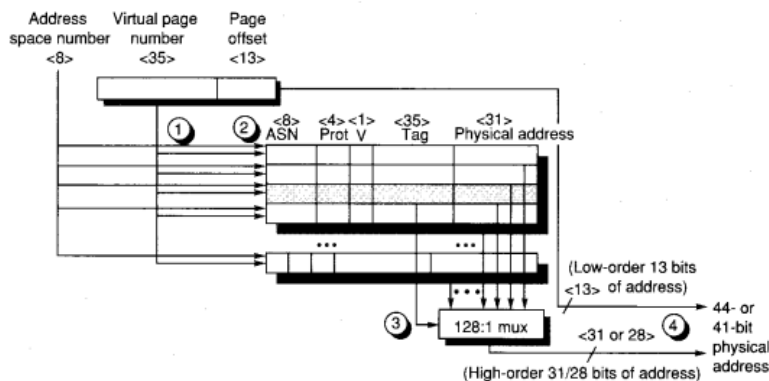


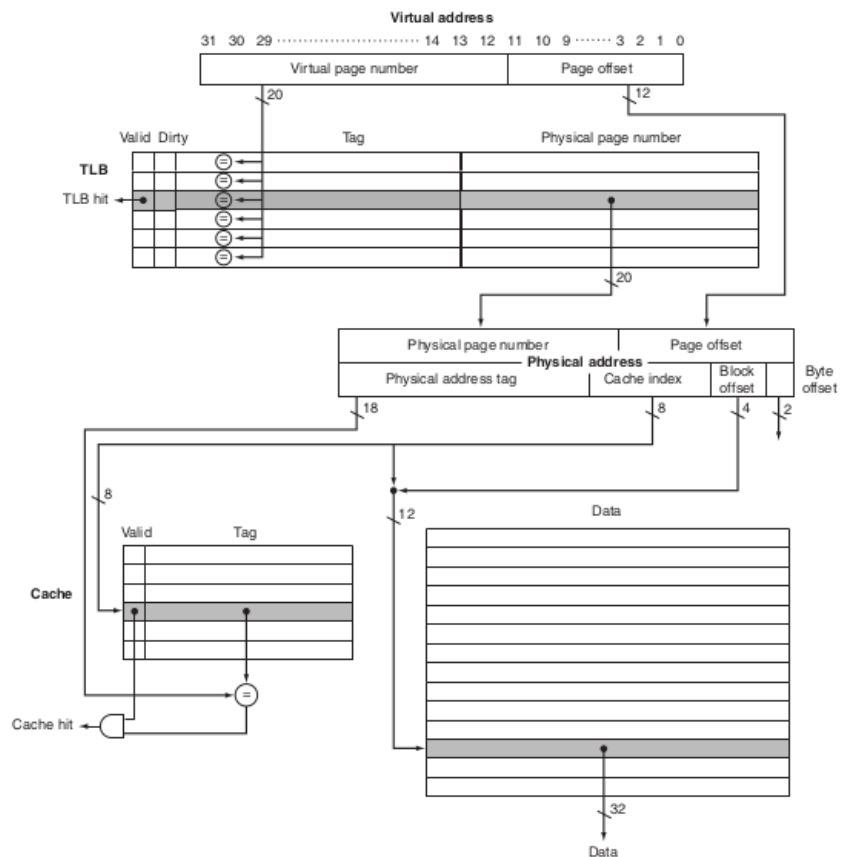
Figure 5.36 Operation of the Alpha 21264 data TLB during address translation. The four steps of a TLB hit are shown as circled numbers. The address space number (ASN) is used like a PID for virtual caches, in that the TLB is not flushed on a context switch, but only when ASNs are recycled. The next fields of an entry are protection permissions (Prot) and the valid bit (V). Note that there is no specific reference, use bit, or dirty bit. Hence, a page replacement algorithm such as LRU must rely on disabling reads and writes occasionally to record reads and writes to pages to measure usage and whether or not pages are dirty. The advantage of these omissions is that the TLB need not be written during normal memory accesses nor during a TLB miss. Alpha 21264 has an option of either 44-bit or 41-bit physical addresses. This TLB has 128 entries.

Para reducir los TLB Misses debido al *context switch* cada entrada tiene 8 bits para el *address space number* o *ASN*. Si el context swiching retorna al proceso con el mismo ASN, este aun puede machear con la TLB. Asi el proceso ASN y la page table entry deberían tambien machear con un tag valido.

No es necesario incluir los 13 bit del page offset. El page offset luego es combinado con el *physical page frame* para formar la dirección física entera.

3.3 Cache direccionada fisicamente

La cache esta fisicamente indexada y fisicamente tagueada. Todas las direcciones de memoria deben ser traducidas a direcciones fisica antes de acceder a la cache. Se asume un tamaño de pagina de 4 KB, pero en realidad es de 16 KB.



3.4 Cache virtualmente indexada y físicamente tagueada

Ejemplo de traducción una dirección virtual de 64 bits a una dirección física de 41 bits con dos niveles de cache. Mientras que la TLB es full asociativa, la cache es de mapeo directo. Para implementar un TLB full asociativa, se requiere que todos los TLB Tag sea comprado con el *virtual page number* dado que puede estar en cualquier lugar de la TLB. Si el bit de validez de la entrada que macheo, el acceso a la TLB es un hit y los bit de la PPN (*physical page number*) junto con los bits del *page offset* forman el índice para acceder a la cache.

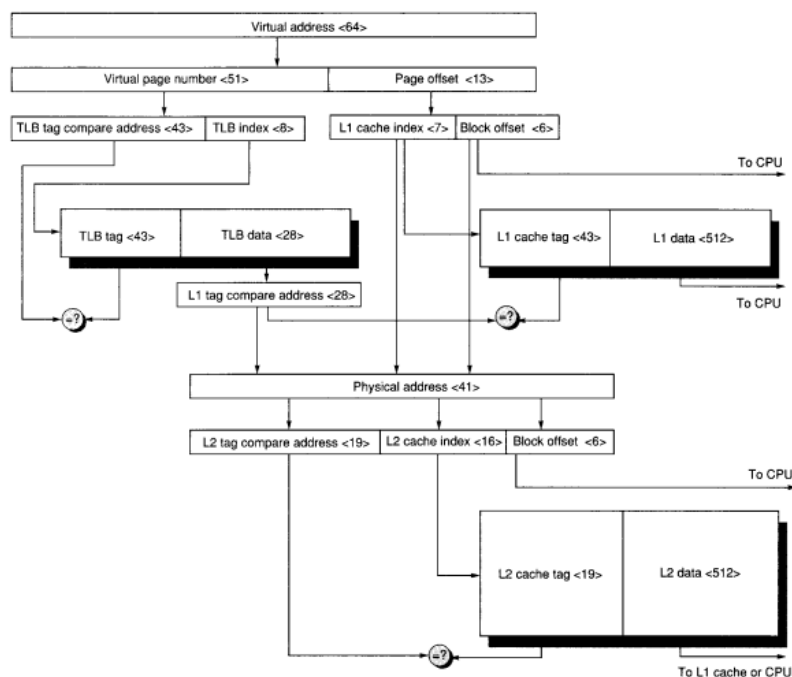


Figure 5.37 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB, and the L2 cache is a direct-mapped 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache, as in Figure 5.43, is replication of pieces of this figure.

La cache L1 esta virtualmente indexada y físicamente tagueada desde que ambos tamaños de cache y tamaño de paginas son de 8 KB. La cache L2 es de 4MB. Ambas caches usan bloques de 64 bytes (512 bits).

La dirección virtual se divide logicamente entre una *Virtual Page Number* y *Page Offset*. El primero es enviada a la TLB para ser traducida en una *dirección física* y la segunda parte es enviada a la cache L1 para ser utilizada como un índice. Si la TLB machea con un hit, la *physical page number* es enviado a la cache L1 para comparar si coincide con el *L1 cache tag*. Si coincide, se produce un hit en la cache L1. El *block offset* es enviado para seleccionar la palabra

(*word*) en la CPU.

Si el resultado de chequear el tag en cache L1 es un miss, la dirección física es usado en la cache L2. La parte el medio de la dirección física se utiliza como índice y la parte superior es comparado con tag de cache L2 seleccionado con el índice. Si coincide con el tag, entonces se produce un L2 cache hit, y el L2 data es enviado a la CPU, que junto con el offset se selecciona la palabra deseada. Ante un cache miss en L2, la dirección física es utilizada para obtener el bloque de memoria.

3.5 Modelo 3C

Es un modelo que permite buscar las fuentes de fallos en una jerarquía de memoria y ver como los cambios en la jerarquía repercuten en la tasa de fallos. Este modelo clasifica los cache misses en tres tipos:

- *Fallos inevitables (Compulsory misses)* - Estos son los fallos de cache causados por el primer acceso a un bloque que nunca ha estado en la cache. También se llaman fallos de arranque en frío (cold start misses). Es el miss rate de una cache LRU de tamaño infinito.
- *Fallos de capacidad (Capacity misses)* - Estos son fallos cuando la cache no puede alojar todos los bloques necesarios para la ejecución de un programa. Los fallos de capacidad ocurren porque los bloques reemplazados son accedidos más tarde. Es el miss rate de una cache LRU full associative menos el miss rate de una cache LRU full associative de tamaño infinito.
- *Fallos de conflicto (Conflict misses)* - Estos ocurren en cache asociativas o de mapeo directo, cuando múltiples bloques compiten por una misma ubicación. Los fallos de conflictos son eliminados por una cache totalmente asociativa del mismo tamaño. También se llaman fallos por colisión. Es el miss rate de la cache bajo estudio *menos* el miss rate de una cache LRU full associative del mismo tamaño.

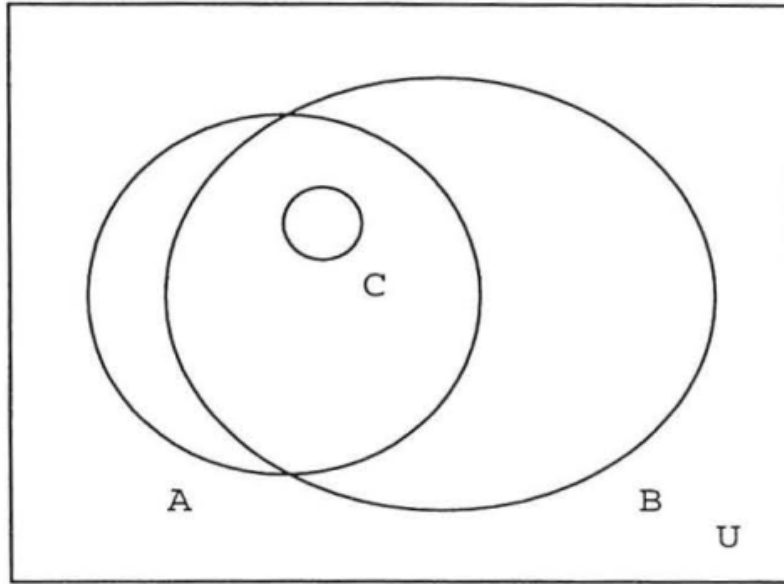


Fig. 1. Representation by sets of the hits and misses in a cache memory.

Donde U es el conjunto de todas las referencias a memoria hechas por el programa, B es el conjunto de referencias a memoria que dieron miss en la cache bajo analisis, A es el conjunto de referencias que dieron miss en una cache full associative del mismo tamaño que la cache bajo analisis y usando una politica de reemplazo LRU y C es el conjunto que corresponde a las referencias a memoria en una cache de tamaño infinito.

$$Compulsory = \#C$$

$$Capacity = \#A - \#C$$

$$Conflict = \#B - \#A$$

Donde $\#C$, $\#A$ y $\#B$ es el numero de elementos en los conjuntos C , A y B respectivamente. En algunos casos es posible obtener valores negativos de miss rate de conflicto.

3.6 Arquitectura multihilo

Multithreading permite que multiples hilos compartan las unidades funcionales de un procesador simple de forma solapada. Para que puedan compartirse, el procesador debe duplicar el estado de cada hilo. Por ejemplo, una copia separada

del register file, un PC separado, y una tabla de paginas separada para cada hilo. La memoria puede ser compartida mediante el mecanismo de la memoria virtual, el cual ya soporta multiprogramación. Además, el hardware debe poder tener la habilidad de poder cambiar a un hilo diferente relativamente rapido. Un thread switch debe ser mas rapido que un context switch de un procesador.

Hay dos aproximaciones de multihilos, *Fine-grained Multithreading(FMT)* cambia entre hilos en cada instrucción, intercalando así la ejecución de multiples hilos. Para intercarlar hilos se utiliza round-robin, saltando los hilos que esten en stall. Para que sea practico, el CPU debe poder cambiar de hilo en cada ciclo de reloj. Una ventaja es que se puede ocultar la perdida throughput de tanto stall largos como cortos, ejecutando instrucciones de otro hilo mientras un hilo este en stall. La principal desventaja es que se demora la ejecución de hilos individuales, debido a que si un hilo que no esta en stall, debe esperar a que se terminen de ejecutar intrucciones de otros hilos.

Coarse-grained Multithreading(CMT) Solo cambia de hilo cuando un stall es muy costoso. Esta limitado su capacidad de obtener bajo throughput, especialmente en stall cortos.

Simultaneous multithreading (SMT)

4 Un viaje de Arquitectura de conjunto de instrucciones (ISA)

4.1 E

s un producto de diferentes grupos que se involucraron en la arquitectura durante 20 años.

1. 1985 - El 80386 extiende la arquitectura de 80286 a 32 bits. Además de agregar una arquitectura de 32 bits con registros de 32bits y un espacio de direcciones de 32 bits, agrega nuevos modos de direccionamiento y operaciones adicionales.

]