Memory Close to the CPU: Caches

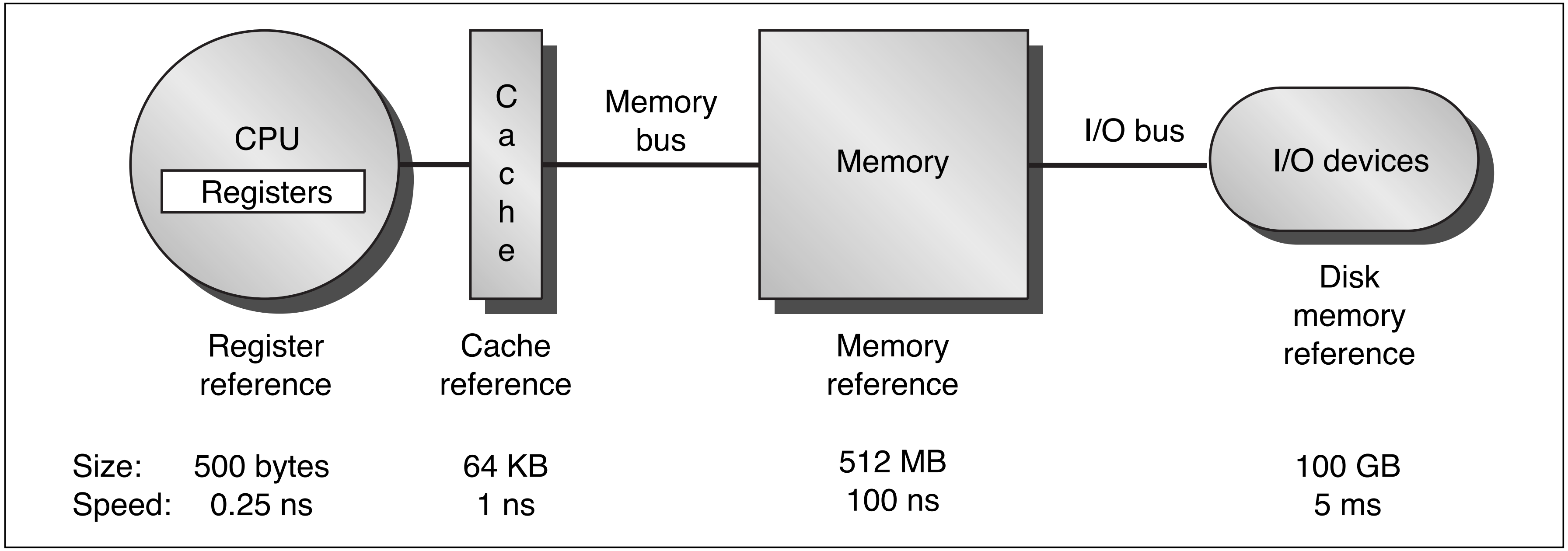
Chapter 6

Cache: a safe place for hiding or storing things.
Webster's Dictionary (1976)

The Memory Hierarchy

- Ideally, a computer system would provide unlimited amounts of fast memory.
- An economical solution is the memory hierarchy:
  - Takes advantage of principle of locality and cost-performance of memory technologies:
    1. Programs do *not* access data uniformly, and
    2. smaller hardware is faster.

The Memory Hierarchy



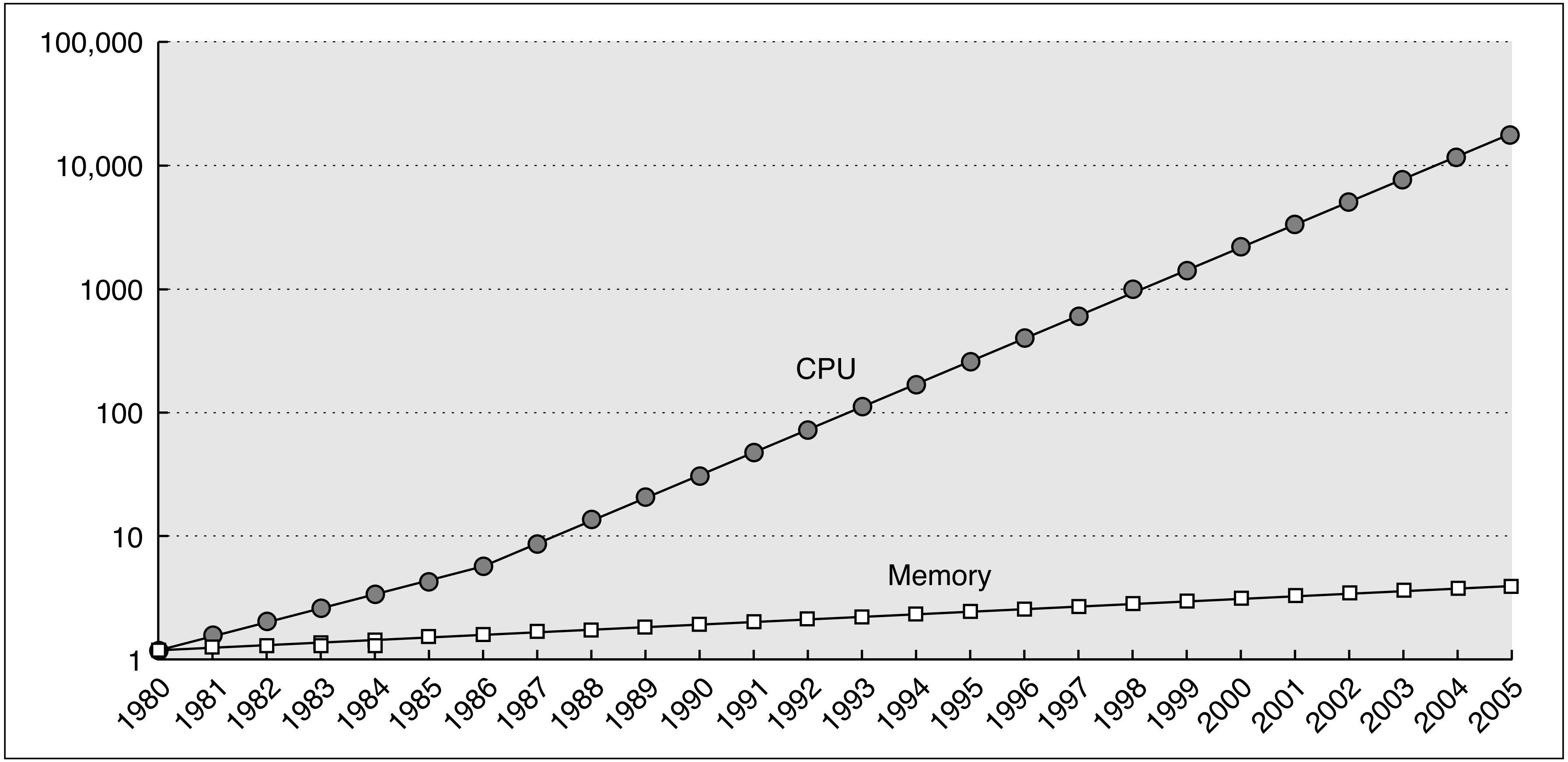
- Usually, levels of the hierarchy subset one another: data in one level are also found in the level below.

CPU and Memory Performance Diverges

- Since 1986, CPU performance improved by a factor of 1.55/year (55%/year).
- DRAM (Dynamic RAM) access speed improves by about 7%/year.

▶ Modern CPUs spend larger and larger fractions time to wait for memory reads and writes to complete (memory latency).

The CPU–Memory Speed Gap



Cache ABC

- Cache: First level of the memory hierarchy encountered once a memory read/write request leaves the CPU.
- Cache hit: CPU finds the requested item in the cache. Otherwise: cache miss.
  - On a cache miss, copy a fixed-size data block from lower-level memory and place it in the cache.
  - Temporal locality: *The item* contained in the copied block will be *needed again soon*.
    Spatial locality: *Other items in the same block* will be needed soon.

Cache ABC

- A cache miss is handled by hardware and causes the CPU to stall until the data are available.
  - Latency: Time required to transport the *first word* of the block into the CPU.
  - Bandwidth^{-1}: Time required to transport the *rest of the block*.
  - Crossing more memory hierarchy levels leads to higher latency and smaller bandwidth.

Impact of Memory Stalls

CPU execution time =
(CPU clock cycles +
Memory stall cycles) × Clock cycle time

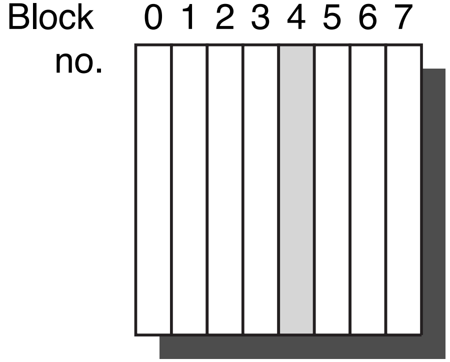
Memory stall cycles =
Instruction count ×
Memory accesses / Instruction ×
Miss rate ×
Miss penalty

Placing a Block in the Cache: Direct Mapping

- Direct mapping: Each block has only one place it can appear in the cache. Usual mapping:

  block no = (block address) *mod* (# of blocks in cache)

Placing block 12 in a direct-mapped cache of size 8 blocks:

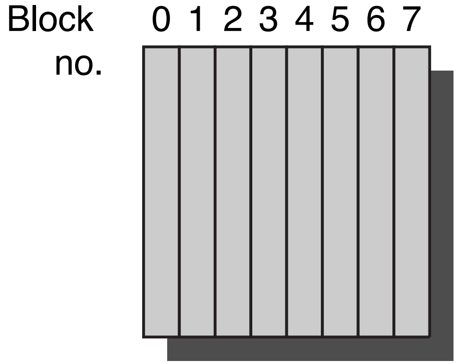


Placing a Block in the Cache: Fully Associative

- Fully associative cache: Each block can appear anywhere in the cache:

block no ∈ { 0, ... (# of blocks in cache) - 1 }

Placing a block
in a fully-associative
cache of size 8 blocks:

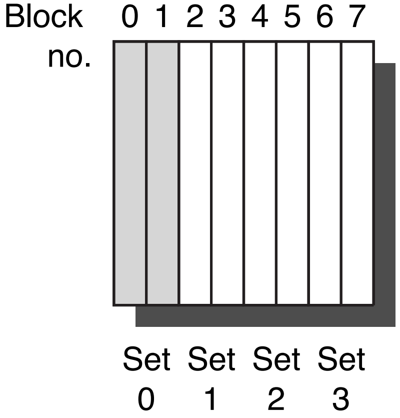


Placing a Block in the Cache: *n*-Way Set Associative

- *n*-way set associative cache: Each block can appear in a restricted set (= group of *n* blocks in the cache). Placement inside the set is arbitrary. Usually:

  set no = (block address) mod (# of sets in cache)

Placing block 12
in a 2-way-associative
cache of size 8 blocks:



Finding a Block in the Cache

- Divide block memory address into tag, index, and offset:

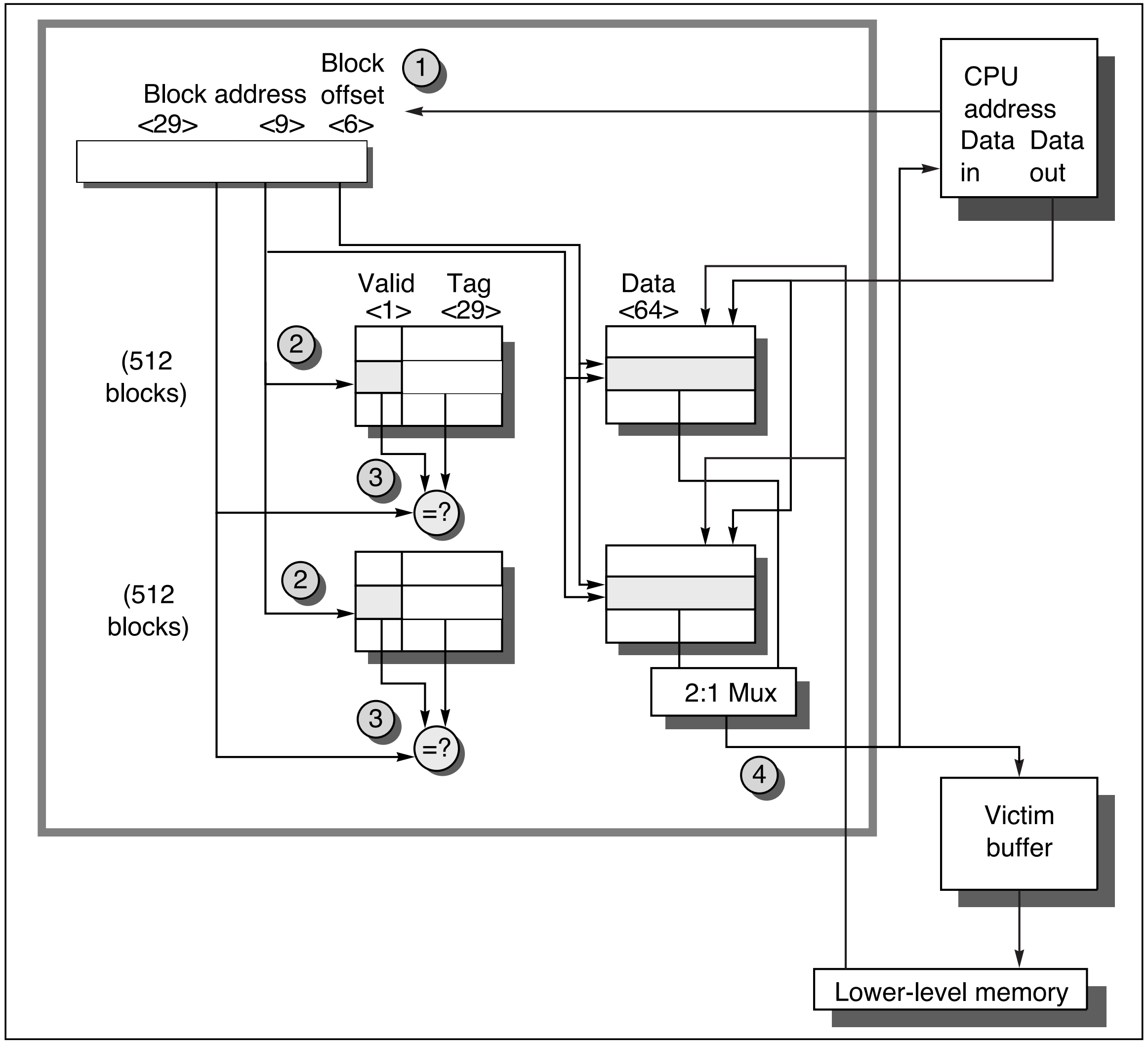
Physical block address		
Tag	Index	Offset

1. Index: selects the set (*e.g.*, 512 sets ≡ 9 bits). Fully associative caches have no index field.

2. Tag: compared against all tags of the blocks in the selected set (in parallel) to determine hit/miss.

3. Offset: address of the requested item in the cache block (e.g., cache block size 64 bytes ≡ 6 bits)

The Alpha 21264 Data Cache



Which Block to Replace on a Cache Miss?

- Whenever a miss occurs, the cache controller selects a block to be replaced with the desired data.
  - There is *no choice* in a direct-mapped cache: the direct mapping immediately determines the block to be replaced.
  - With fully or set-associative placement, *more than one* block may be replaced.
    A replacement strategy selects the victim block.

Replacement Strategies

- Random:
  Spread allocation uniformly between candidate blocks (easy to build in hardware).
- Least-recently used (LRU):
  Rely on temporal locality: replace the block that has been unused for the longest time
  (record of block accessed required).
- First in, first out (FIFO):
  Computationally simple approximation of LRU:
  replace *oldest* block.

What Happens on a Write?

- Caches tend to be read-optimized: all instruction access are reads and typical instruction mixes show ≈ 10% writes, but ≈ 35% reads.
  - On a read access, read the cache block contents and the block tag *in parallel*.
    Compare tag: if hit, immediately deliver data to CPU; if miss, ignore the read data.
  - No such optimism is allowed for write accesses.
    Due to tag checking, writes take longer than reads.

Strategy 1: Write Back

- Write back:
  Data is written to the cache block only. Modified block is written to lower level only when replaced.

1. Use clean/dirty status bit to reduce write memory traffic.
2. From the CPU viewpoint, writes occur at the speed of cache memory.
3. May save memory bandwidth, thus attractive in multi-processor systems and embedded devices (saves power).

Strategy 2: Write Through

- Write through:
  Data is always written to both the cache block and the block in lower-level memory
  (option no-write allocate: on a write miss, write to lower-level memory only).

1. Easy to implement (no clean/dirty bits).
2. Cache is always clean: read misses never result in writes to lower-level memory.
3. Cache and lower levels always agree on memory contents: aids consistency and coherency.

Example: (No-)Write Allocate

No-Write Allocate

```
WriteMem[100];
WriteMem[100];
ReadMem[200];
WriteMem[200];
WriteMem[100];
```

Write Allocate

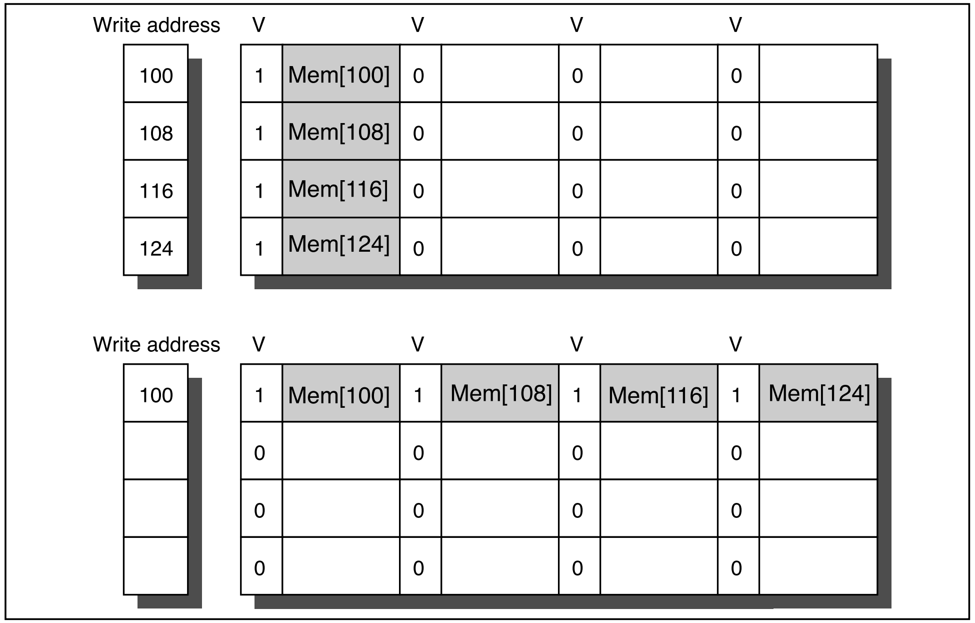
```
WriteMem[100];
WriteMem[100];
ReadMem[200];
WriteMem[200];
WriteMem[100];
```

Cache Blocks

0	1	2	···
100	200		

Write Buffers

- Minimize effect of CPU write stalls with a write buffer:
  - CPU writes (*address*, *data*) pair into write buffer — CPU immediately resumes execution thereafter.
  - Flush write buffer to lower-level memory *asynchronously*. If possible, try to merge writes:



Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Write Buffer Consistency

- Assume direct-mapped (addresses 512, 1024 map to the same cache block) write-through cache with write buffer.
  Will the value in R2 always be equal to the R3 value?

```
SW R3, 512(R0)    ; M[512] ← R3
LW R1, 1024(R0)   ; R1 ← M[1024]
LW R2, 512(R0)    ; R2 ← M[512]
```

⇒ On read miss, delay until write buffer empty (or check write buffer for potential conflicting addresses).

Impact of Caches

CPU execution time =
Instruction count ×
(CPI + Miss rate × Memory accesses/Instruction × Miss penalty) ×
Clock cycle time

CPU execution time =
Instruction count ×
(1.0 + 2% × 1.5/Instruction × 100 cycles) ×
Clock cycle time

Reduce Miss Penalty: Second-Level Cache

- With a two-level cache,
  - the first-level (L1) cache can be small enough to match the CPU clock cycle time, while
  - the L2 cache can be large enough to capture many accesses (typically, L1 cache ⊂ L2 cache).

Average memory access time =
Hit time_{L1} +
Miss rate_{L1} ×
$\left.(\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})\right\} = \text{Miss penalty}_{L1}$

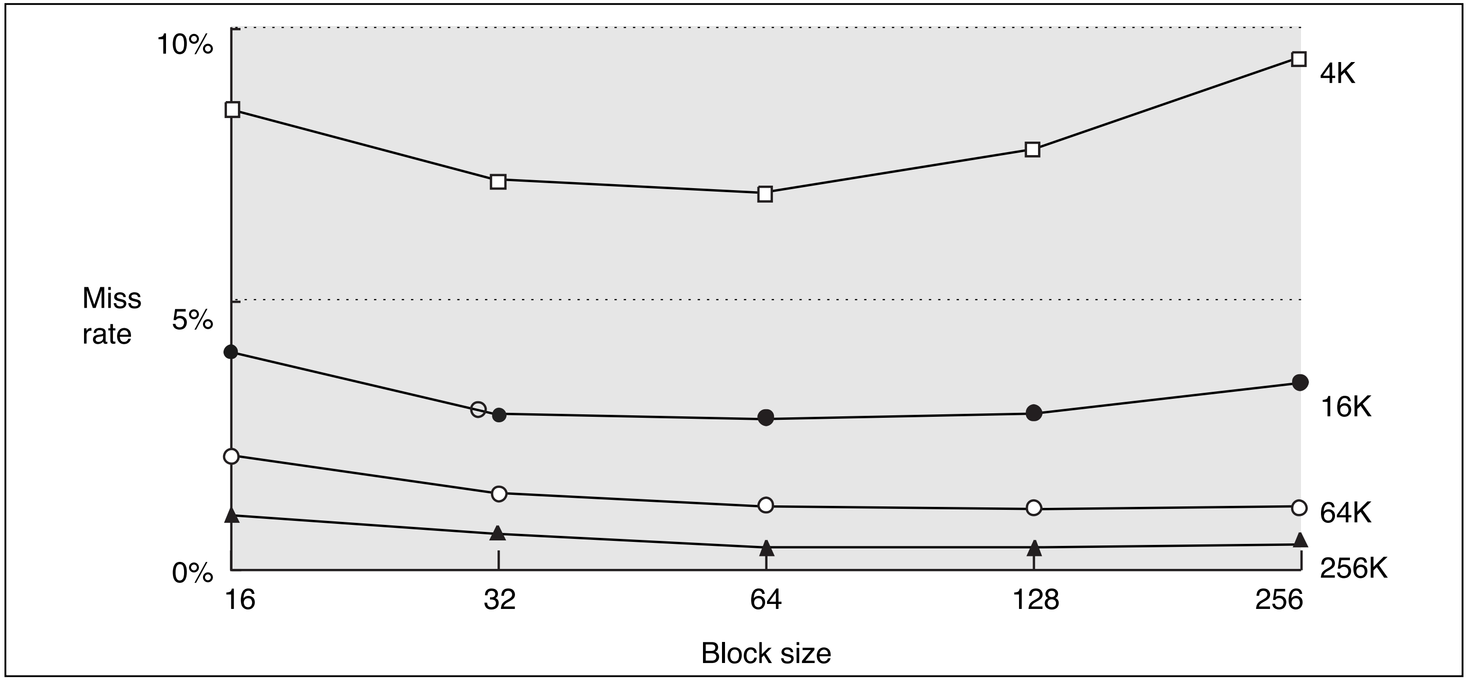
Reducing Miss Rate

- Types of cache misses (the "*three C*"):

  1. Compulsory: The very first access to a block cannot be a hit (cold-start miss).
  2. Capacity: The cache cannot hold all blocks needed during program execution.
  3. Conflict: In direct-mapped or set associative caches, too many blocks may map to a set (blocks will be discarded and then later be retrieved again).

Reducing Miss Rate

- Reduce number of compulsory misses: Enlarge cache block size to take advantage of spatial locality.
  - But: (Too) large block sizes may *increase* the number of conflict misses in small caches:



Reducing Miss Rate

- If the active memory of a program (part) is larger than first-level memory, capacity misses occur.
    - Cache thrashing: Significant data movement between memory hierarchy levels.
    - Computer runs at speed of lower-level memory (or even slower due to miss overhead).
- Most obvious measure to reduce capacity misses: Enlarge cache size (also reduces conflict misses).

Reducing Instruction Cache Misses

- Tight, compactly-coded loops lead to near-zero instruction cache misses after the first iteration.
- Reorder procedures in the program code.
- Try to align basic blocks of code on cache block boundaries.

? Is the Volcano iterator model vulnerable to instruction cache misses?

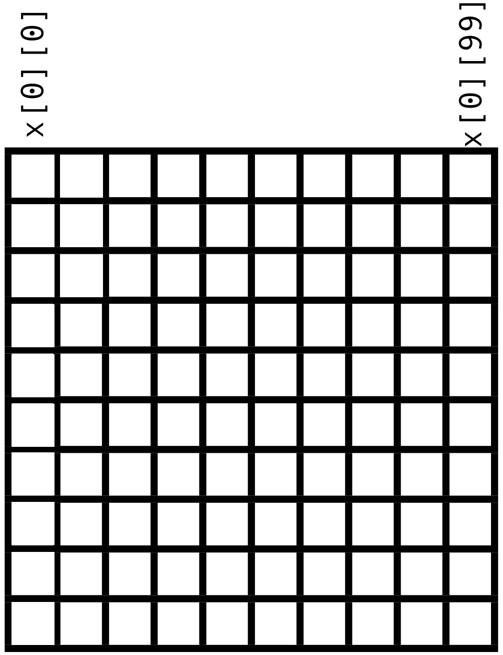
Improving Spatial Locality

- Instruction reordering can improve spatial locality. Do not walk memory in large strides.

Scalar matrix multiplication:

```
for (j = 0; j < 100; j++)
  for (i = 0; i < 5000; i++)
    x[i][j] = 2 * x[i][j];
```

$$x[i][j] \equiv x_{base} + (i*x_{cols} + j) * x_{elem}$$



Improving Spatial Locality

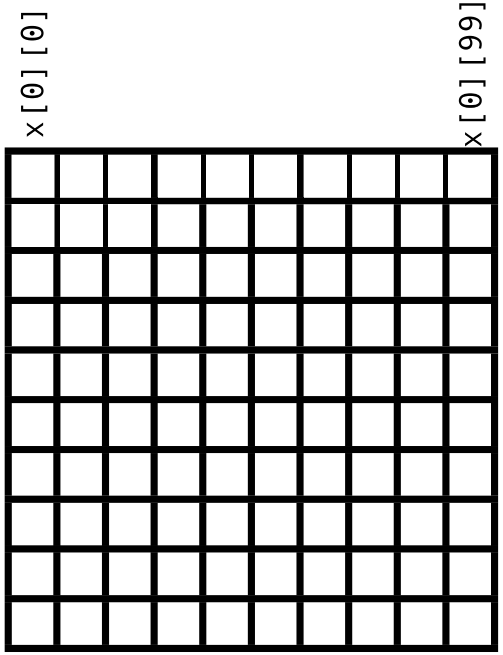
- Align array visits with its row-order memory layout. Try to maximize the use of a cache block before it has to be discarded.

Interchanged array visit order:

```
for (i = 0; i < 5000; j++)
  for (j = 0; j < 100; i++)
    x[i][j] = 2 * x[i][j];
```

$$x[i][j] \equiv$$

$$x_{base} + (i * x_{cols} + j) * x_{elem}$$

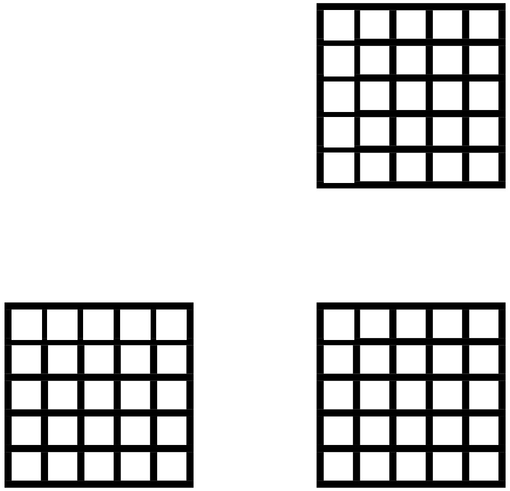


Improving Temporal Locality

- Loop interchange does *not* help if an algorithm requires row-by-row as well as column-by-column array access:

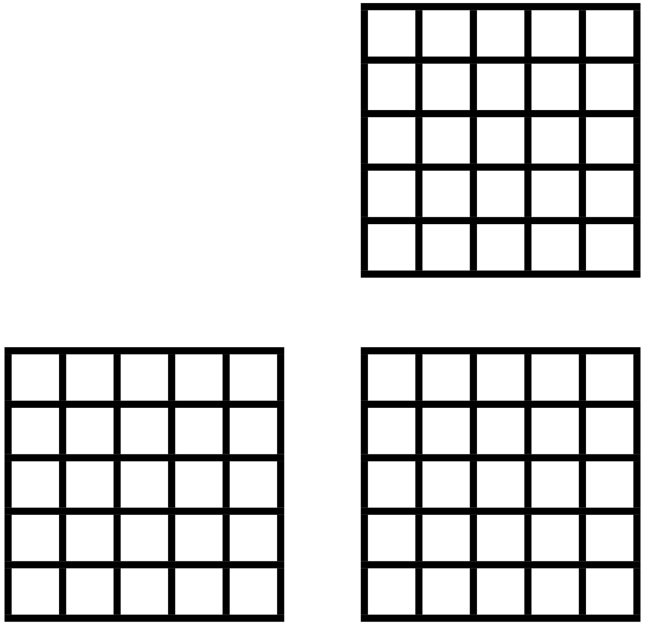
  Matrix/matrix multiplication:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    sum = 0;
    for (k = 0; k < N; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
  }
```



Improving Temporal Locality

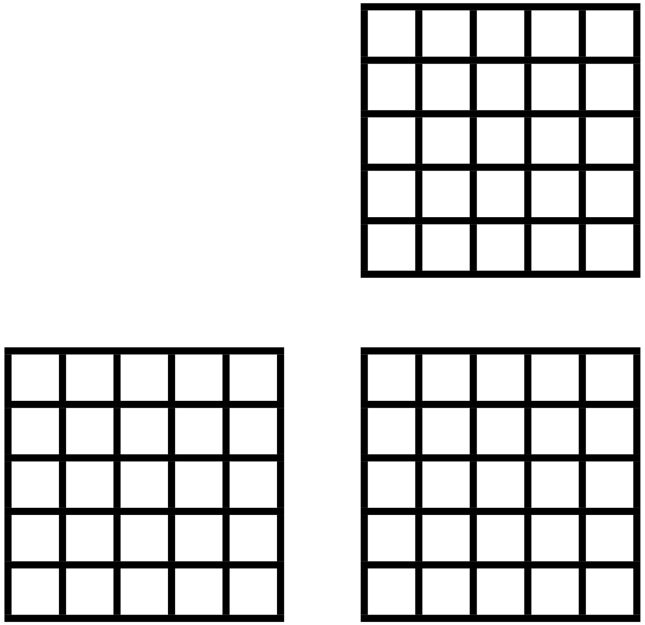
- Age of matrix access (light: old, dark: young):



- If cache can hold $N \times N$ plus one row, then A, B may stay in cache.
- Worst case: $2 \times N^3 + N^2$ capacity misses for N^3 operations.

Blocking

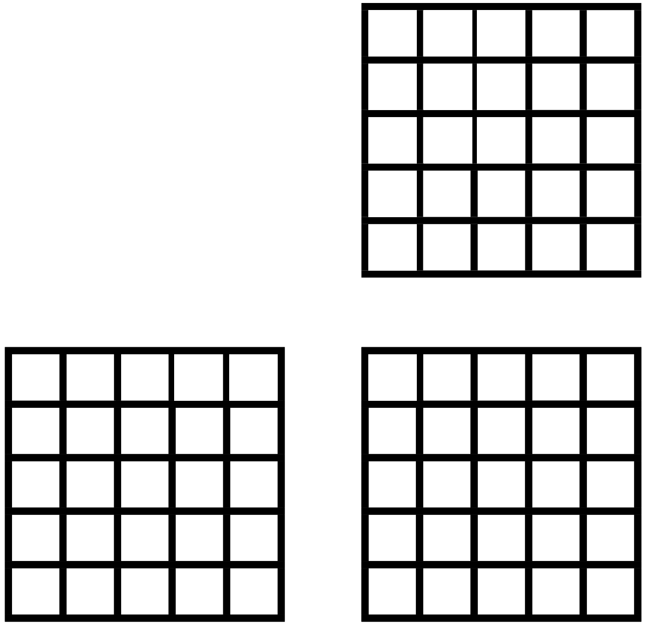
- Perform matrix multiplication for submatrices (*i.e.*, in blocks).
  Block size $B \times B$, with blocking factor $B < N$. Choose blocking factor to match cache size:



- # of updates on an element of C before its final value is computed: $\lceil N/B \rceil$.
- A benefits from spatial, B benefits from temporal locality.

Blocking — Access Pattern

- The “blocked” matrix/matrix multiplication follows a local access pattern. Here: `N` = 5, `B` = 3:



Blocking

- Assume `C[i][j] = 0` before routine starts.
- This operates on a block (submatrix) of size `B` × `B` in matrix `B`.

```
for (jj = 0; jj < N; jj += B)
  for (kk = 0; kk < N; kk += B)
    for (i = 0; i < N; i ++)
      for (j = jj; j < min(jj+B, N); j++) {
        sum = 0;
        for (k = kk; k < min(kk+B, N); k++)
          sum += A[i][k] * B[k][j];
        C[i][j] += sum;
  }
```

Nonblocking Caches

- For pipelined computers with out-of-order completion, CPUs need not stall on a cache miss.
  - Example: continue to execute instructions while waiting for an outstanding data read/write request.
  - A nonblocking cache is able to continue to supply hits during a miss ("*hit under miss*").
    Effectively reduces overall miss penalty.
  - Caches may even support "miss under miss" and deal with multiple outstanding memory requests.

Prefetching

- Nonblocking caches allow out-of-order CPUs to overlap memory access to with instruction execution.
- Another approach to benefit from nonblocking caches is prefetching.
  Typically driven by hardware outside the cache itself.
  - Instruction prefetching.
    CPU fetches two blocks on a miss:

    1. Requested block → Instruction cache
    2. Consecutive block → Instruction stream buffer

Data Prefetching

- The prefetching idea can also be applied to data prefetching.
  - Modern CPUs support multiple separate data stream buffers, each prefetching at different addresses.
  - Some CPUs (e.g., UltraSparcIII) calculate address differences to detect the stride.
- Tradeoff:
  Prefetching utilizes otherwise unused memory bandwidth but may interfere with demand misses.

Compiler-Controlled Prefetching

- Some CPU architectures support explicit prefetch instructions:

```
prefetch addr
```

- Instructions are designed to be *semantically invisible:*
  - Do not change contents of registers or memory.
  - Do not cause virtual memory faults (non-binding prefetch — might turn into no-op, if required).

Compiler-Controlled Prefetching

- Loops lend themselves to suitable places to inserts `prefetch` instructions: it is necessary to be able to anticipate request addresses in future iterations.
  - If miss penalty small:
    Compiler unrolls loop once or twice, then inserts `prefetch`es into emitted instructions.
  - If miss penalty is significant:
    Unroll multiple times or use software pipelining, then judiciously insert `prefetch`es.

Inserting Prefetch Instructions

- Assess the expected data cache misses for the following code fragment.
  Assume 16-byte cache blocks, ignore conflict or capacity misses. Elements of A and B are 8 bytes long.

```
double A[3][100];
double B[101][10];

for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j ++)
    A[i][j] = B[j][0] * B[j+1][0];
```

Inserting Prefetch Instructions

- Assume a miss penalty that is equivalent to 7 iterations of the inner loop. Prefetch instructions:

```
for (j = 0; j < 100; j++) {
  prefetch(B[j+7][0]);
  prefetch(A[0][j+7]);
  A[0][j] = B[j][0] * B[j+1][0];
}

for (i = 1; i < 3; i++)
  for (j = 0; j < 100; j ++) {
    prefetch(A[i][j+7]);
    A[i][j] = B[j][0] * B[j+1][0];
  }
```

- Note: A *write hint* for `A[i][j+7]` would suffice here.

DRAM Memory Technology

- Since mid-1970s, computers are equipped with Dynamic RAM (DRAM) technology.
  - DRAM memory chips are organized in rows and columns — which are addressed in two separate phases: row and column access strobe (*RAS*, *CAS*).
  - In DRAMs, each bit is implemented by a single transistor. Reading a bit can disturb the transistor state.

⇒ Refresh state periodically by addressing each row of the DRAM and "reading" all bits simultaneously.

SRAM Memory Technology

- Static RAM (SRAM) uses 6 transistors per bit to prevent information from being disturbed when read.
- SRAM does not require dynamic refresh — DRAM refresh time may account for 5% of total time.
- Address lines are not multiplexed as in DRAMs to reduce access time.
- SRAM cycle time 8 to 16 times faster than DRAM. (But: SRAM 8 to 16 times as expensive.)

Main Memory Performance

- Main memory is the next level down in the hierarchy.
  - Main memory latency — time elapsed before first byte is read/written — is of concern to the caches.
  - Main memory bandwidth — bytes read/written per time unit — is of interest for large transfers (e.g., I/O, large second level cache blocks).
- Typical timing (read a memory word ≡ 8 bytes):

4 cy	56 cy	4 cy
send address	access word	send data

Main Memory Performance

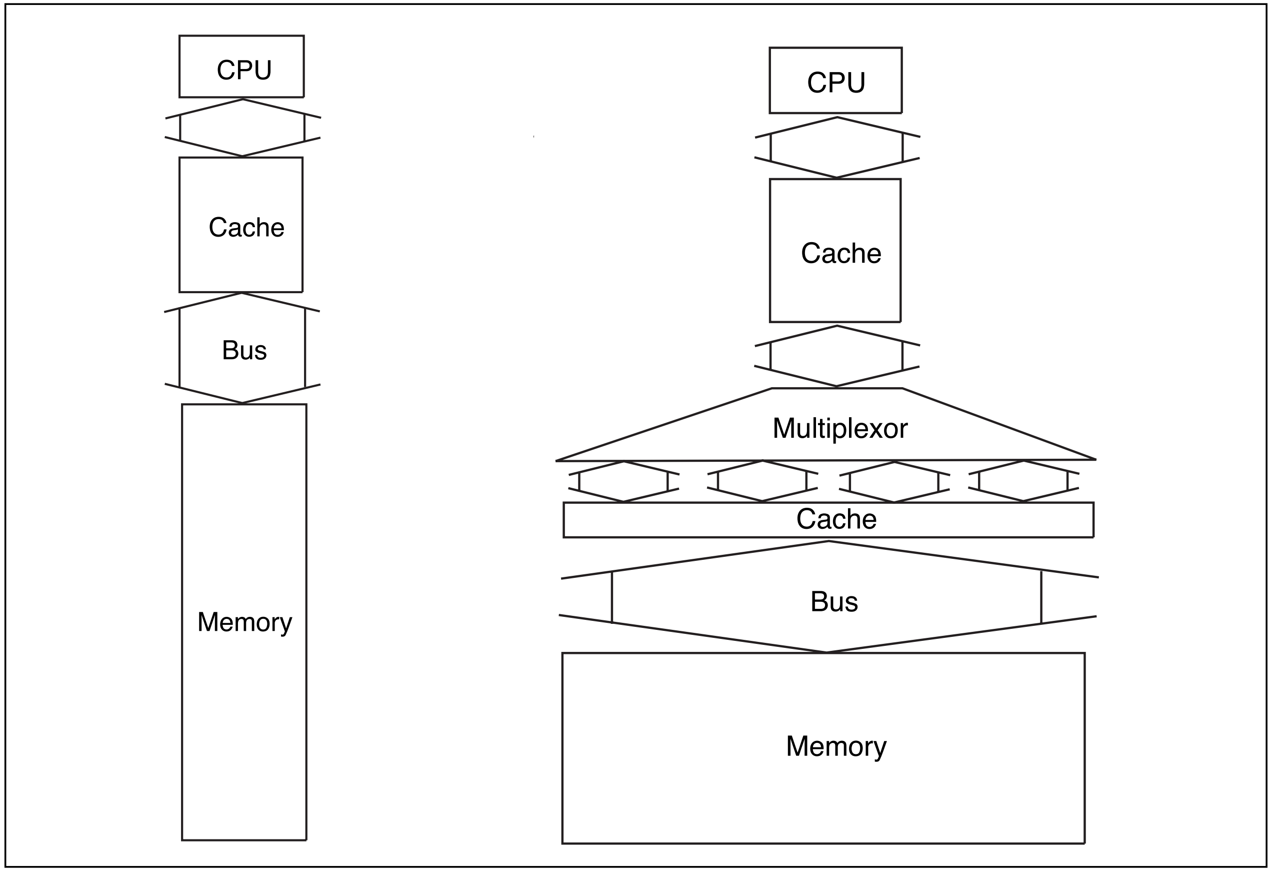
- Penalty for a miss in a cache of 32-byte blocks (≡ retrieve 4 memory words):

  Resulting memory bandwidth: 1/8 byte/cy.

- Organizing memory word-by-word makes sense since the CPU request data that way.
  We repeatedly pay memory access costs, though.

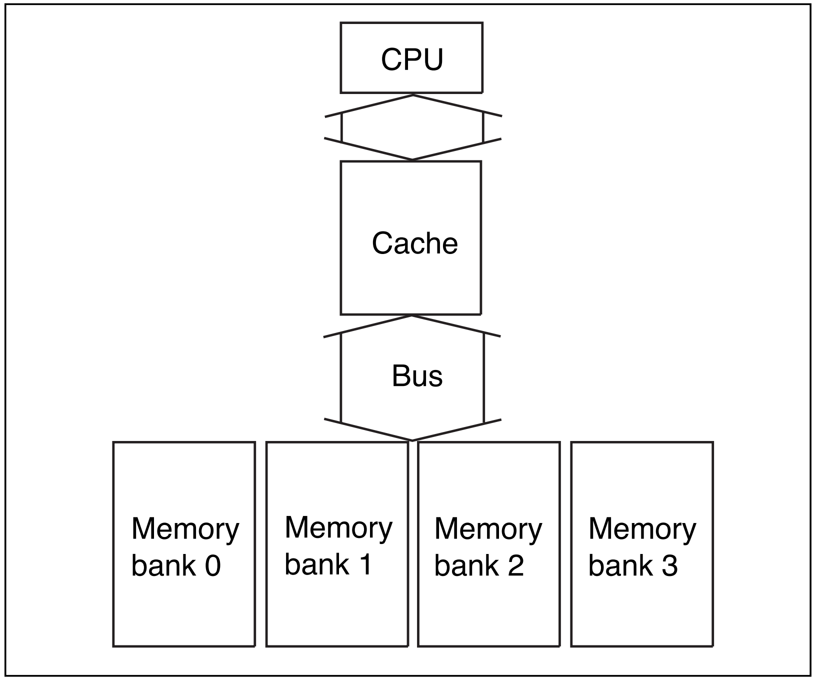
Wider Main Memory

- Reduce number of memory accesses by doubling or quadrupling width of cache and memory.



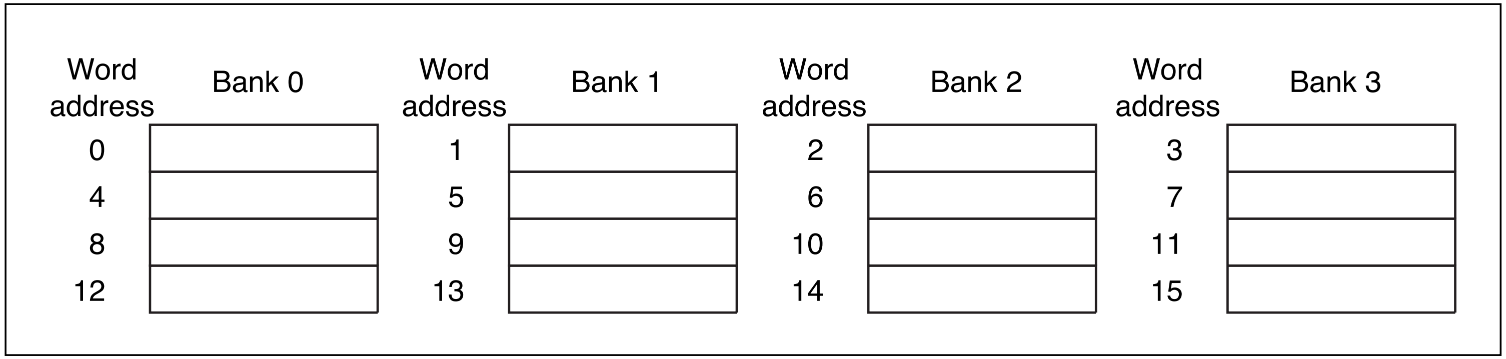
Interleaved Memory

- Take advantage of multiple memory chips in system — banks inside single DRAM or multiple DRAMs.
- Banks are one word wide such that bus and cache width need not change:



Interleaved Memory

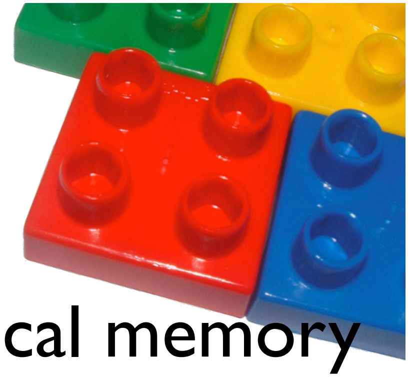
- Mapping of addresses to banks affects behavior of interleaved memory.
  - Interleave memory at word level (four banks): bank *i* stores words with addresses (*addr* mod 4) = *i*:



Word address	Bank 0	Word address	Bank 1	Word address	Bank 2	Word address	Bank 3
0		1		2		3	
4		5		6		7	
8		9		10		11	
12		13		14		15	

- Miss penalty: 4 cy + 56 cy + (4 × 4 cy) = 76 cy.

Virtual Memory



- Virtual memory divides the available physical memory into blocks (*pages*) and allocates these to different processes.

- Each process tends to use only a fraction of its overall address space. Virtual memory enables the coexistence of such processes.

  - Virtualized memory further defines protection schemes to separate process address spaces.

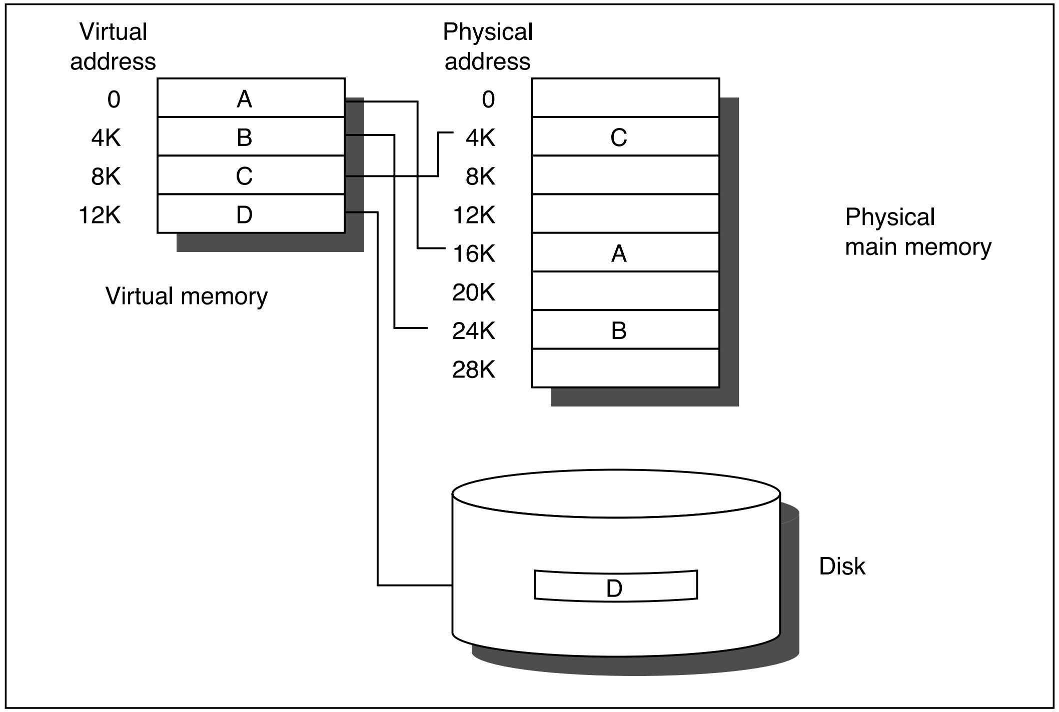
  - But: Read-only pages may be shared by processes.

Virtual Memory

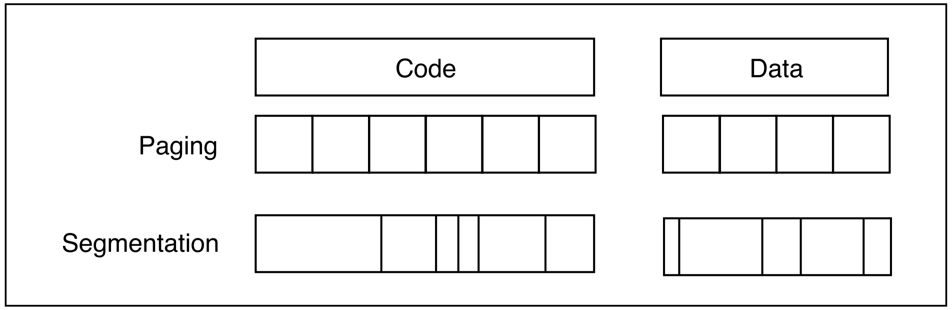
- With virtual memory, a process' overall code and data space may *exceed* available physical memory.
    - Prior to virtual memory schemes, it was the responsibility of programmers to (un)load *overlays*.
- Virtual memory automatically manages two levels of the memory hierarchy:
  main memory and secondary storage (disk).
    - A page mapping keeps track of where a page on secondary storage belongs in the virtual memory space of a process.

Page Mapping

- Processes use a contiguous virtual address space. Hardware generates page faults for unmapped paged (D). OS loads affected pages and updates the mapping.



Paging vs. Segmentation



	Page	Segment
Words per address	One (*page no*|*offset*)	Two (segment, offset)
Programmer visible?	Invisible	May be visible
Replacing a block	Trivial (all block are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (page portions unused)	External fragmentation (unused portions of RAM)
Efficient disk traffic	Yes (adjust page size to balance disk access and transfer time)	Not always (small segments may transfer few bytes only)

Caches vs. Virtual Memory

Parameter	L1 Cache	Virtual Memory
Block (page) size	16–128 bytes	4K–64K bytes
Hit time	1-3 cy	50–150 cy
Miss penalty	8-150 cy	1,000,000–10,000,000 cy
(access time)	(6-130 cy)	(800,000–8,000,000 cy)
(transfer time)	(2-20 cy)	(200,000–2,000,000 cy)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45 bit phys.addr. to 14–20 bit cache addr.	32-64 bit virtual addr. to 25–45 bit physical addr.

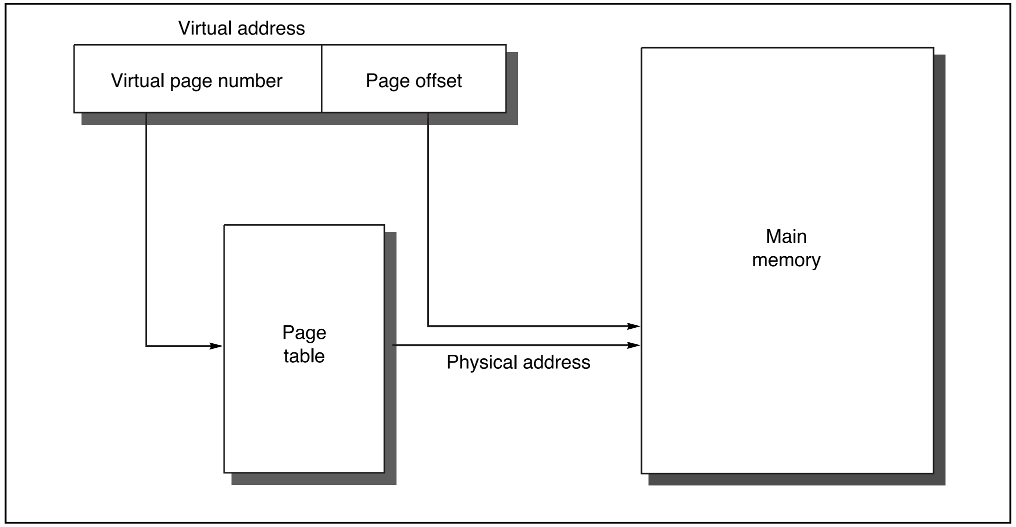
Virtual Memory: Block Placement

- Page placement in physical memory?
  Due to the exorbitant miss penalty — access a rotating magnetic storage device — try hard to reduce number of conflict misses.

⇒ Allow page placement anywhere in main memory (cf. fully associative caches).

Virtual Memory: Page Addressing

- Addressing a page in physical memory?
  Divide virtual address bits into *page no|offset*. Use *page no* for lookup in mapping data structure (→ physical page addess), then concatenate *offset*.



Virtual Memory: Page Replacement

- Almost all OSs apply the *Least Recently Used* (LRU) heuristics to select victim pages in physical memory for replacement to reduce page faults (conflict misses).
  - Implementation:
    Maintain a use or reference bit for each mapped page. Set bit if page is accessed. OS clears all bits periodically.

Virtual Memory: Write Policy

- Because a page write on secondary storage is worth millions of clock cycles, virtual memory systems use write back plus dirty bits.

- Some virtual memory systems support copy-on-write (COW): When a process allocates a page, share this page with other processes, if possible.
  Only if the page is written to, perform a page copy (→ `fork()`, `calloc()`).

Fast Address Translation?

- Paging means that every memory access effectively takes *twice* as long:

  1. One memory access to perform the virtual → physical address translation, and

  2. a second access to get the data.

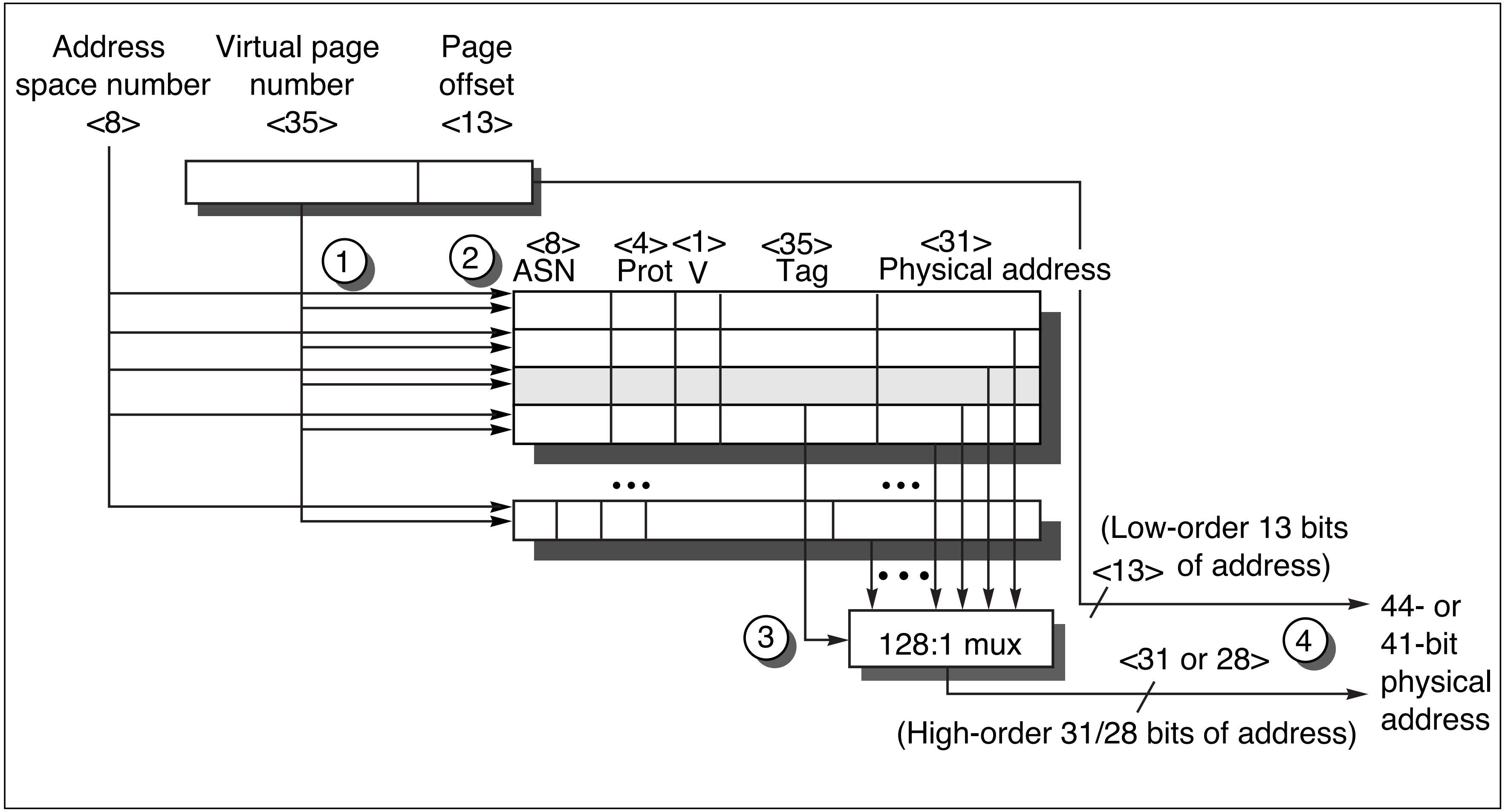
- Since mapping is performed page-wise, effort is saved if we remember the translation for the last page.

- Principle if locality: address translations will exhibit locality, too. Maintain an address translation cache.

Translation Lookaside Buffer

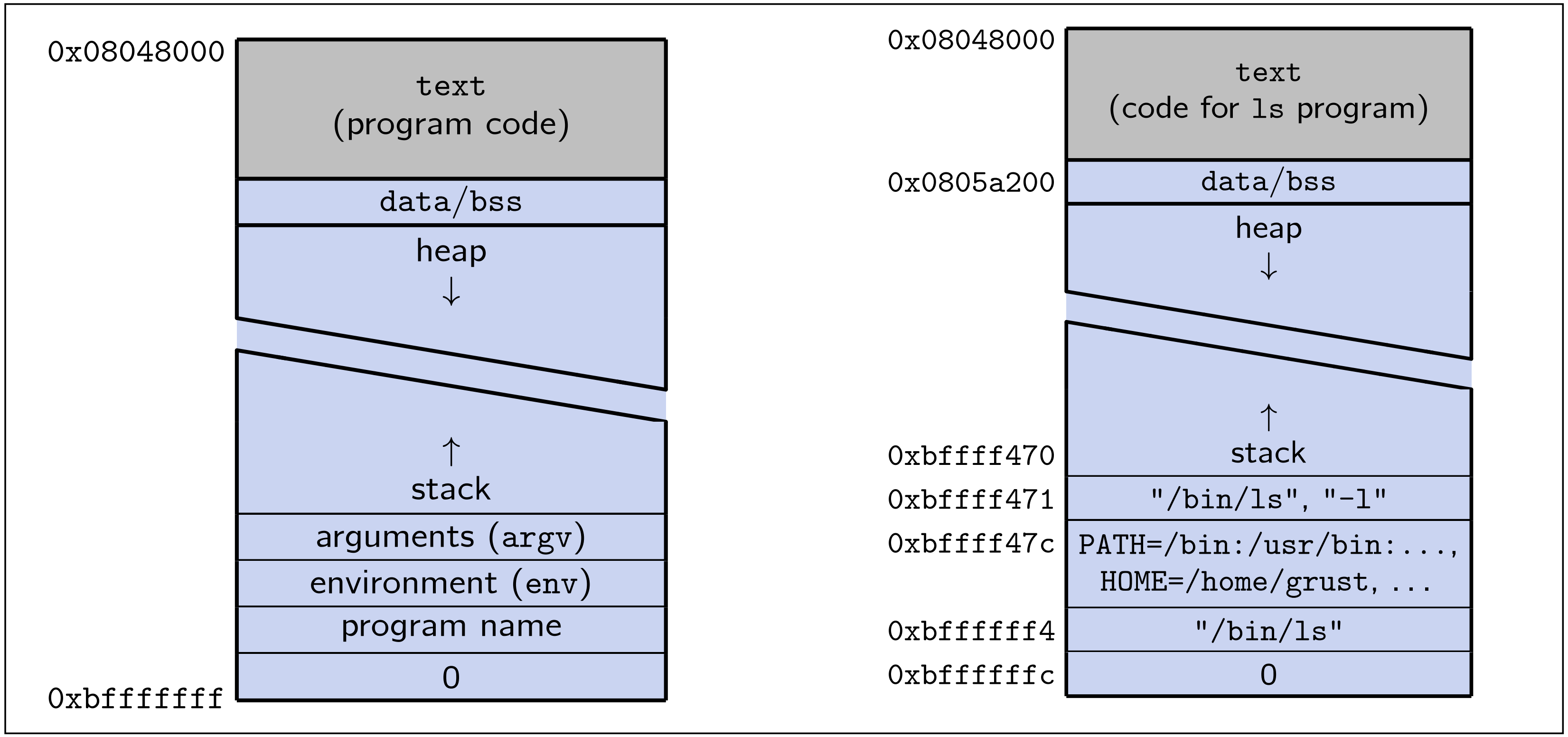
- Translation lookaside buffer (TLB) entries:
  - Tag: High-order bits of the virtual address.
  - Data: Physical page number +
    valid | dirty | protection bits.
- Note: If the OS needs to remap a page or modify protection information for a process, its TLB entries need to be *invalidated*
  ⇒ TLB entry will be reloaded later.

The Alpha 21264 TLB



Process Context Switches

- In modern multi-tasking OSs, different processes all use the *very same* address space layout.
  - Cf. the layout of an ELF binary:



Address Translation and Process Context Switches

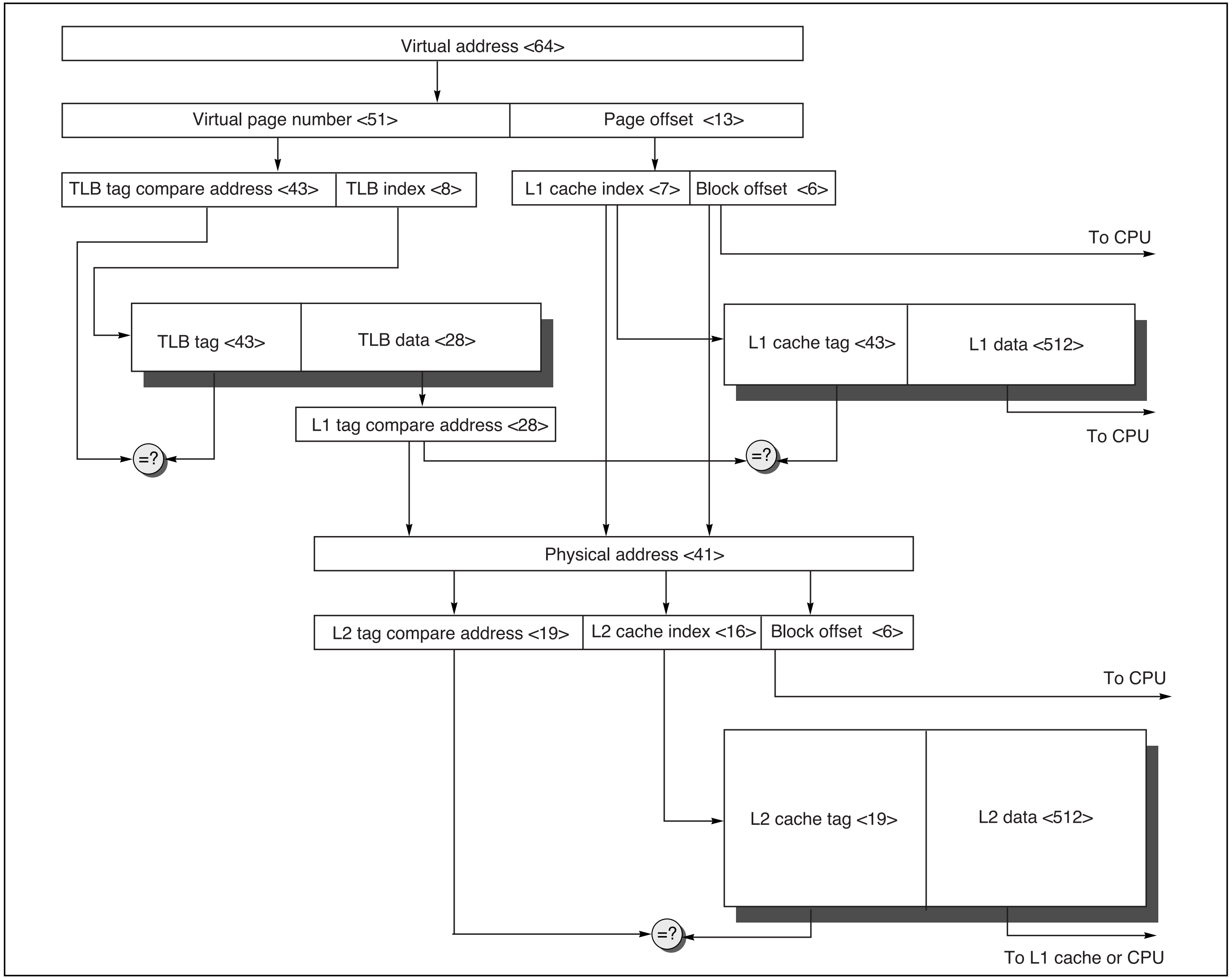
- One a process context switch, the TLB needs to be flushed (the physical address space of separate process is disjoint).

  ⇒ Directly after a context switch, a process suffers from TLB misses.

- To avoid: Prefix virtual address and TLB tags with process identifier (Alpha 21264: "ASN").

  - After context is switched back to a process, its TLB entries may still be present.

Virtual Address → L2 Cache


Virtual address <64>
Virtual page number <51>
Page offset <13>
TLB tag compare address <43>
TLB index <8>
L1 cache index <7>
Block offset <6>
To CPU
TLB tag <43>
TLB data <28>
L1 cache tag <43>
L1 data <512>
L1 tag compare address <28>
To CPU
=?
=?
Physical address <41>
L2 tag compare address <19>
L2 cache index <16>
Block offset <6>
To CPU
L2 cache tag <19>
L2 data <512>
=?
To L1 cache or CPU