Chapter 5

Memory Hierarchy Part 1

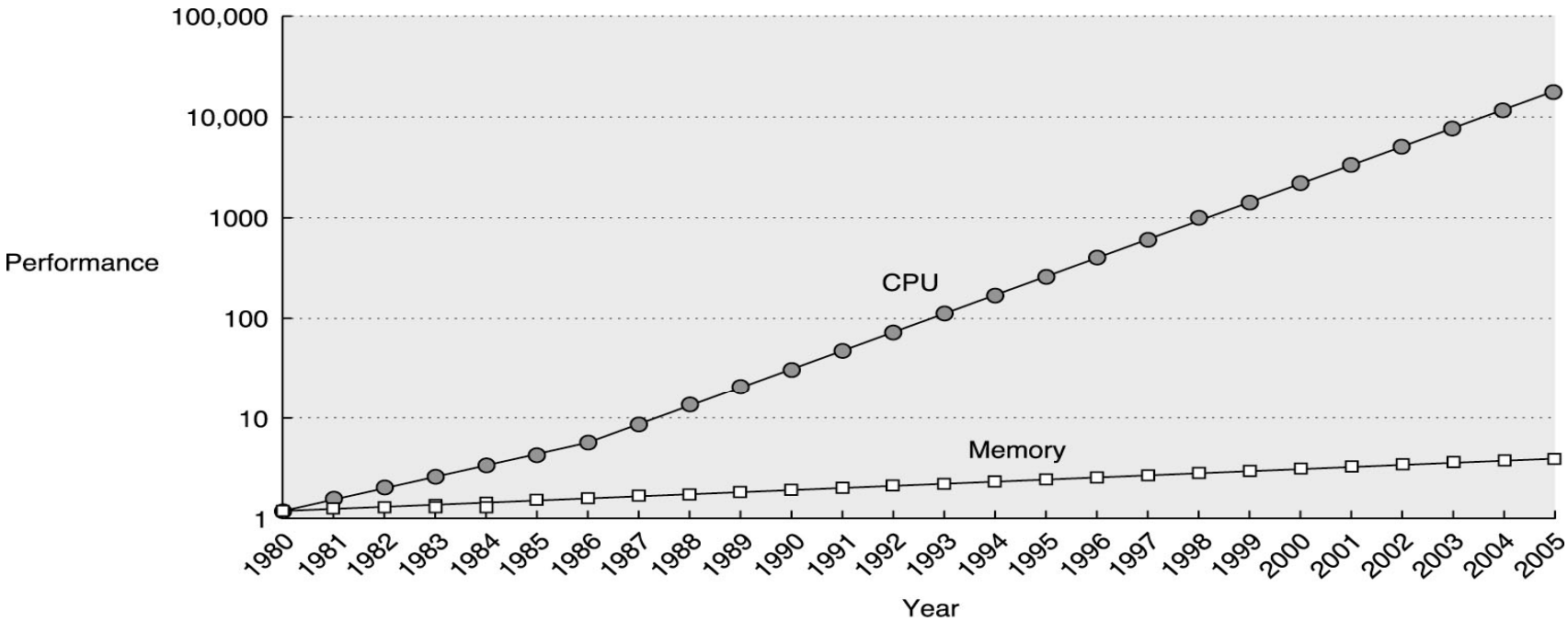
Slides: W. Gross, V. Hayward, T. Arbel

"Ideally one would desire an idefinitely large memory capacity such that any particular…word would be immediately available…We are…forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, and J. von Neumann, 1946

Introduction and Motivation

- Goal: unlimited amounts of fast memory.



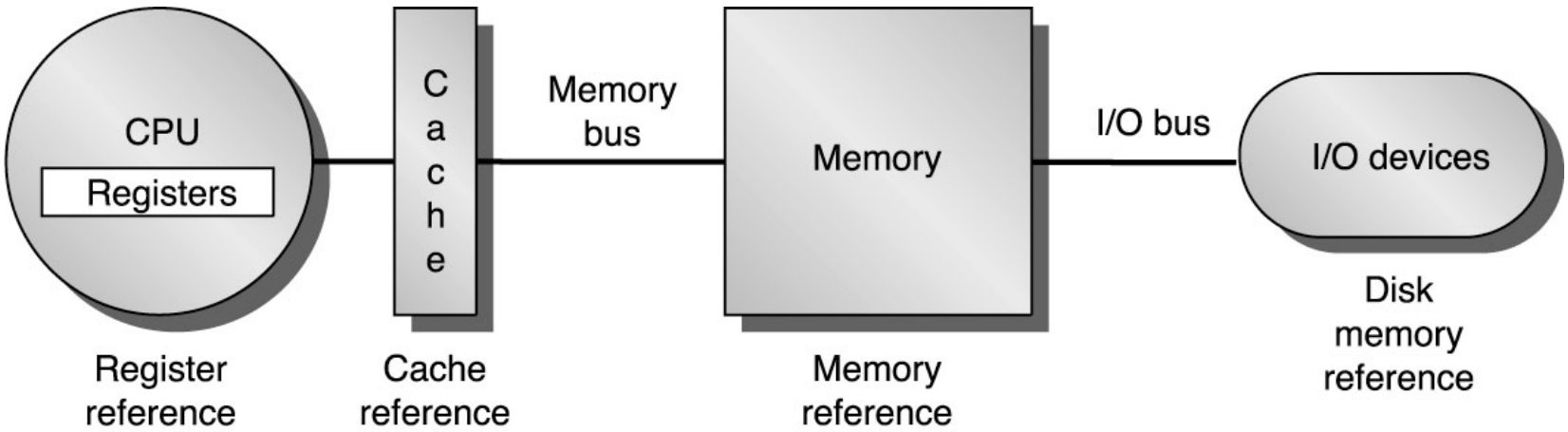
Principle of Locality

- Why does it work?
  - Principle of locality – nonuniform access
  - Smaller memories are faster
- Problem: fast memory is expensive !
- Solution: hierarchy of memory organized into different levels, each progressively larger, but slower

Memory Hierarchy

- Requirements: (apparently contradictory)
  - Provide virtually unlimited storage
  - Allow the CPU to work at register speed
- Analogy with personal belongings:
  - wear what you need for a day
  - suitcase for a week
  - house or apartment as permanent storage

Memory Hierarchy



Memory Hierarchy in 2003

Level	1 registers	2 cache	3 main memory	4 disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Technology	Multi-port custom CMOS memory	On-chip/off-chipd CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (ns)	0.25 – 0.5	0.5 - 25	80-250	5,000,000
Bandwidth (Mb/s)	20,000-100,000	5000 - 10,000	1000 - 5000	20 - 150
Managed by	compiler	hardware	OS	OS/people
Next level	cache	main memory	disk	CD/tape/network

ABCs of Caches

- Cache is a buffer between CPU registers and main memory
  - Divided into blocks to take advantage of temporal and spatial locality in programs
  - Similarly, virtual memory divides a large address space in pages stored in main memory
- Principle: keep in higher level storage a copy of a subset of the next lower level.
  - E.g. registers hold a copy of subset of blocks
  - Blocks are a copy of a subset of the main memory, etc…

ABCs…

- Every time an item is requested
  - Hit: if found in next lower level
  - Miss: if not found
- E.g.
  - load or fetch found in cache => cache hit
- E.g.
  - If cache loads a block and it is not found in main memory => page fault
  - Page is retrieved from the disk
- If a hit …deliver data item
- If a miss…fetch the correct block from the lower level of the hierarchy
  - CPU must stall

CPU Performance with Caches

$$\begin{aligned}\text{MemoryStallCycles} &= \text{NumberOfMisses} \times \text{MissPenalty} \\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instructions}} \times \text{MissPenalty} \\ &= \text{IC} \times \frac{\text{MemoryAccesses}}{\text{Instruction}} \times \text{MissRate} \times \text{MissPenalty}\end{aligned}$$

Miss Rate

- The measurement problem becomes finding the miss rate
  - the average number of misses per access
- Address trace
  - A large sequential collection of addresses accessed by a benchmarks and fed to a cache simulator

Reads / Writes

If we refine the previous formula to distinguish reads and writes, we have:

$$\begin{aligned}\text{MemoryStallCycles} &= \text{IC} \times \frac{\text{MemoryAccesses}}{\text{Instruction}} \times \text{MissRate} \times \text{MissPenalty} \\ &= \text{IC} \times \text{ReadsPerInstruction} \times \text{ReadMissRate} \times \text{ReadMissPenalty} \\ &+ \text{IC} \times \text{WritesPerInstruction} \times \text{WritesMissRate} \times \text{WritesMissPenalty}\end{aligned}$$

Example.

Assume that a machine has a CPI of 1.0 when all memory accesses are hits in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles, and the miss rate is 2%, how much faster would the machine be if all accesses were cache hits?

$$\mathbf{CPUTime} = (\mathbf{CPUClockCycles} + \mathbf{MemoryStallCycles}) \times \mathbf{CCTime}$$

In the perfect case:

$$\mathbf{CPUTime}_{\mathbf{perfect}} = (\mathbf{IC} \times \mathbf{CPI} + 0.0) \times \mathbf{CCTime} = \mathbf{IC} \times \mathbf{CCTime}$$

With the real cache:

$$\mathbf{MemoryStallCycles} = \mathbf{IC} \times \frac{\mathbf{MissRate} \times \mathbf{MemoryAccesses}}{\mathbf{Instruction}} \times \mathbf{MissPenalty}$$

$$= \mathbf{IC} \times 0.02 \times (1.0 + 0.5) \times 25 = \mathbf{IC} \times 0.75$$

$$\mathbf{CPUTime}_{\mathbf{real}} = (\mathbf{IC} + 0.75 \times \mathbf{IC}) \times \mathbf{CCTime} = \mathbf{IC} \times 1.75 \times \mathbf{CCTime}$$

The machine with no cache misses is 1.75 faster.

Four Memory Hierarchy Questions

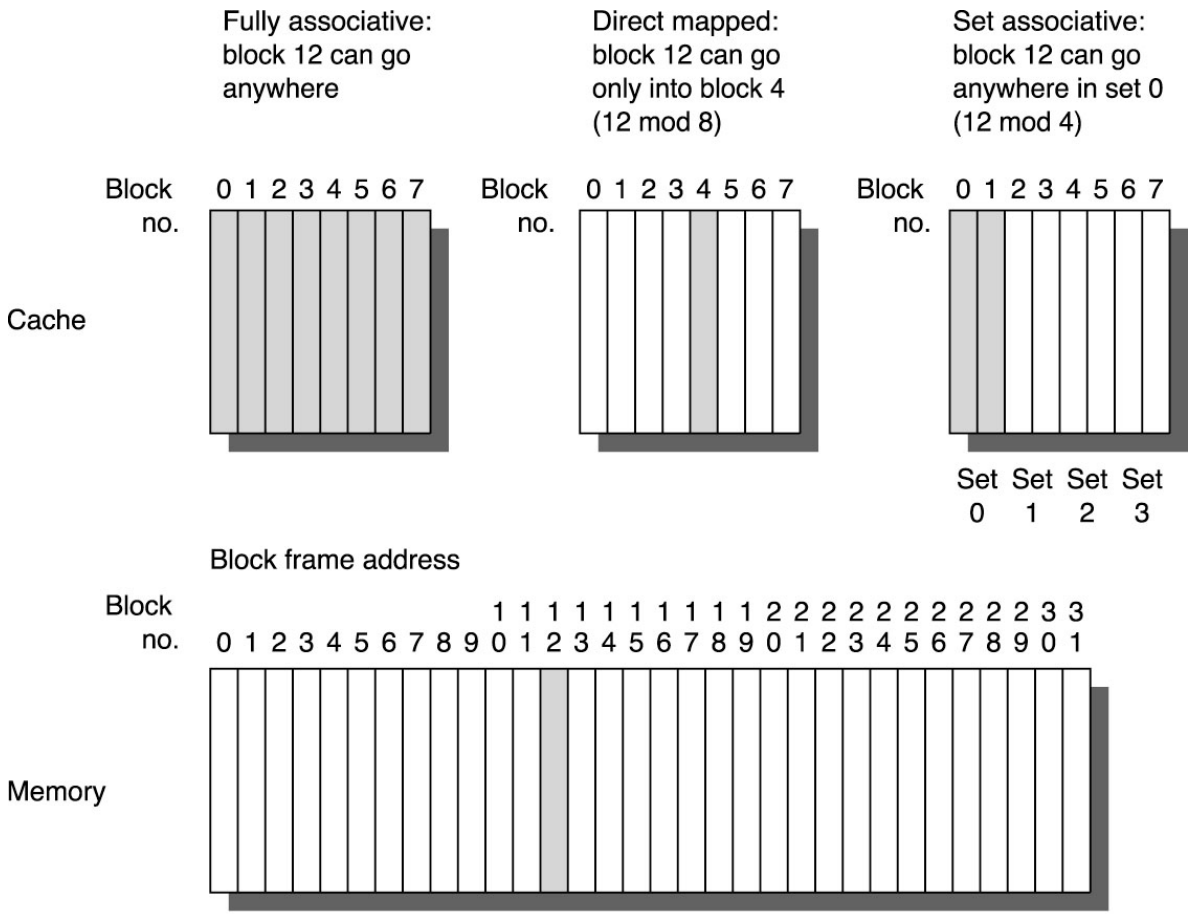
- Caches: Main memory is divided into blocks each consisting of several data elements (e.g. bytes)

1. Where can a block be placed in the upper level?
   - Block placement
2. How is a block found if it is in the upper level?
   - Block identification
3. Which block should be replaced on a miss?
   - Block replacement
4. What happens on a write?
   - Write strategy

Q1: Block Placement

- Fully associative
  - A block can appear anywhere in the cache
- Direct mapped cache
  - Each block can only appear in one place in the cache
    - » *block address* mod *# blocks in cache*
- Set associative
  - A block can be placed in a restriced set of places in the cache
  - First, map a block onto a set
  - The block can be placed anywhere in that set
  - Set chosen by bit selection
    - » *Block address* mod *# sets in cache*
  - n blocks in a set => n-way set associative

Block Placement



Associative Caches

- In general, m blocks in cache, n blocks in a set, s sets in cache

  m = s * n

- n-way set associative => n > 1 and s > 1
- fully associative => n = m and s = 1 (m-way s.a.)
- direct mapped => n = 1 and s = m (1-way s.a.)

- Mapping: set number is called "index"
- *index* = *block #* mod *s*

Real Caches

- Most processor caches today are
  - Direct mapped, or
  - 2-way set associative, or
  - 4-way set associative

Q2: Block Identification

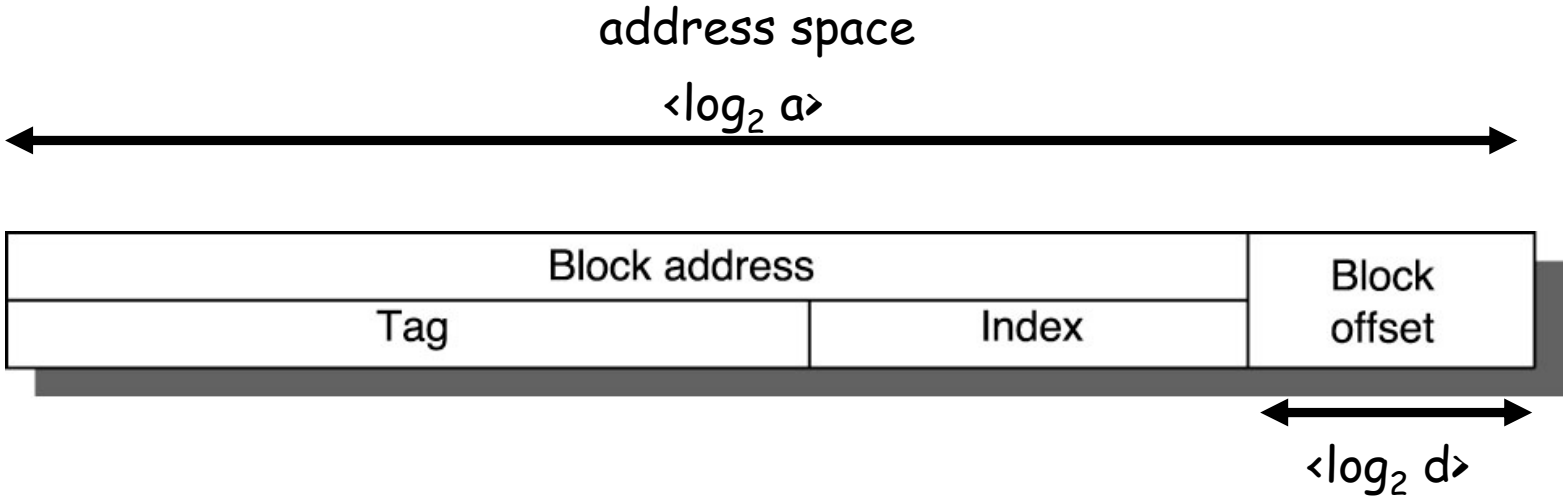
- How is a block found if it is in the cache ?
- Why is this an issue?
  - Many blocks in main memory map to one or few blocks in cache

- A complete address in the address space of *a* addresses can be divided into fields
  - Each field has a significance in the hierarchy

- Each block contains d data items (bytes, words…). So a field in the address called the "block offset" (lower-order bits) indicates which of the d data items in a particular block we want to access

- We then also need a field "block address" to indicate which block number in memory we will access (higher-order bits).

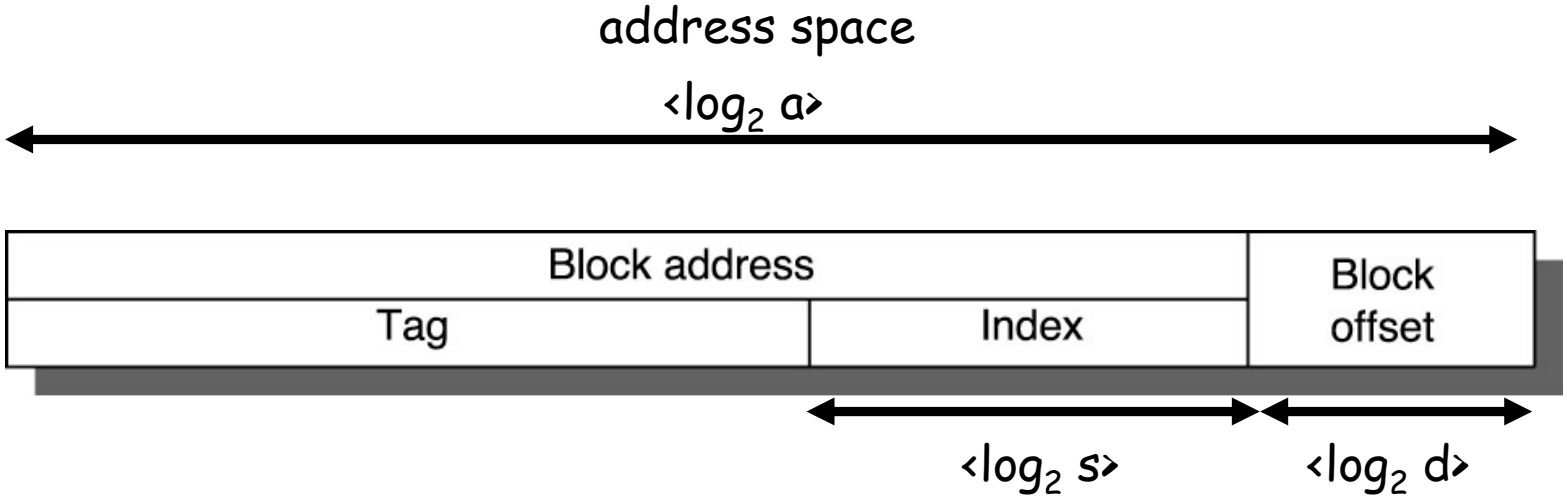
Address Fields



Search

- Now that we know which block to look for, we need to identify which one of the *s* sets the block could be in (*s* is called the "index")
- Now, the block could be anywhere in this set, so we need to do a search within this set to identify the particular block
  - Search for the "tag"

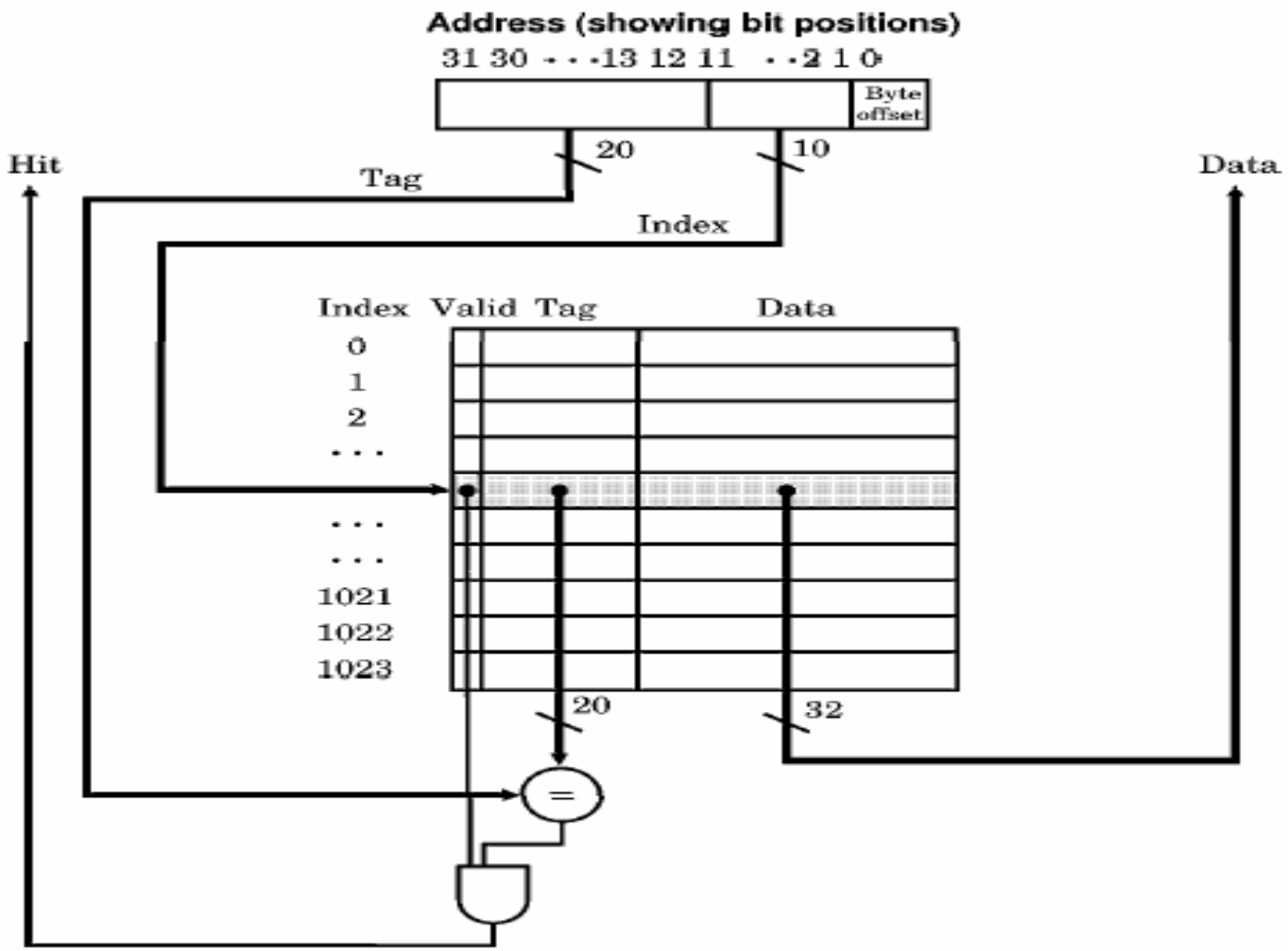
Address Fields



Searching…

- Direct mapped
  - No search
  - Index directly provides block number in cache
- n-way set associative
  - smaller index and a larger tag as n gets larger
- fully associative
  - no index
- Tag check search is done in parallel for speed – if the tag is found => a hit !
- How do we know if the cache block has been loaded with valid data ?
  - Add a "valid bit" to every block in the cache

Direct Mapped Cache



Q3: Block Replaement

- Which block should be replaced on a miss?
- Direct mapped
  - no choice !
- Fully associative or set associative
  - Many blocks to choose to replace

3 Replacement Strategies

1. Random
   - Simple
2. Least-recently used (LRU)
   - If recently used blocks are likely to be used again (locality) then a good candidate for disposal is the LRU block
   - Record accesses to blocks
3. First in, first out (FIFO)
   - Replace oldest block
   - Approximation to LRU

Q4: Write Strategy

- Most cache accesses are reads
  - Instruction accesses are reads
  - MIPS: 10% stores, 37% loads
  - Writes 7% of overall memory traffic
  - Data cache only: writes are 21%

Reads / Writes

- Make common case fast
  - Optimize caches for reads
  - Reads are the easiest to make fast
  - Read block at same time as tag check
  - Only drawback to reading a block immediately is power
  - Ahmdahl's law: don't neglect speed of writes
- Why writes so slow?
  - Writes cannot begin until after tag check
  - Writes can also be of variable width (as can reads, but there is no harm in reading more, except power)

2 Write Strategies

1. **Write through**
   - Write information to the block in the cache and to the block in lower-level memory

2. **Write back**
   - Write information only to block in the cache. Write the modified cache block to the main memory only when it is replaced

Write Back

- "Dirty bit" kept for each block
  - Indicates whether the block is "dirty" (modified while in the cache) or "clean" (not modified)
  - Reduces frequency of write back
- Advantages:
  - writes occur at speed of cache
  - Multiple writes to a block require only one write to main memory
  - Some writes never make it to main memory
    - » Memory bandwidth reduced
    - » Lower power

Write Through

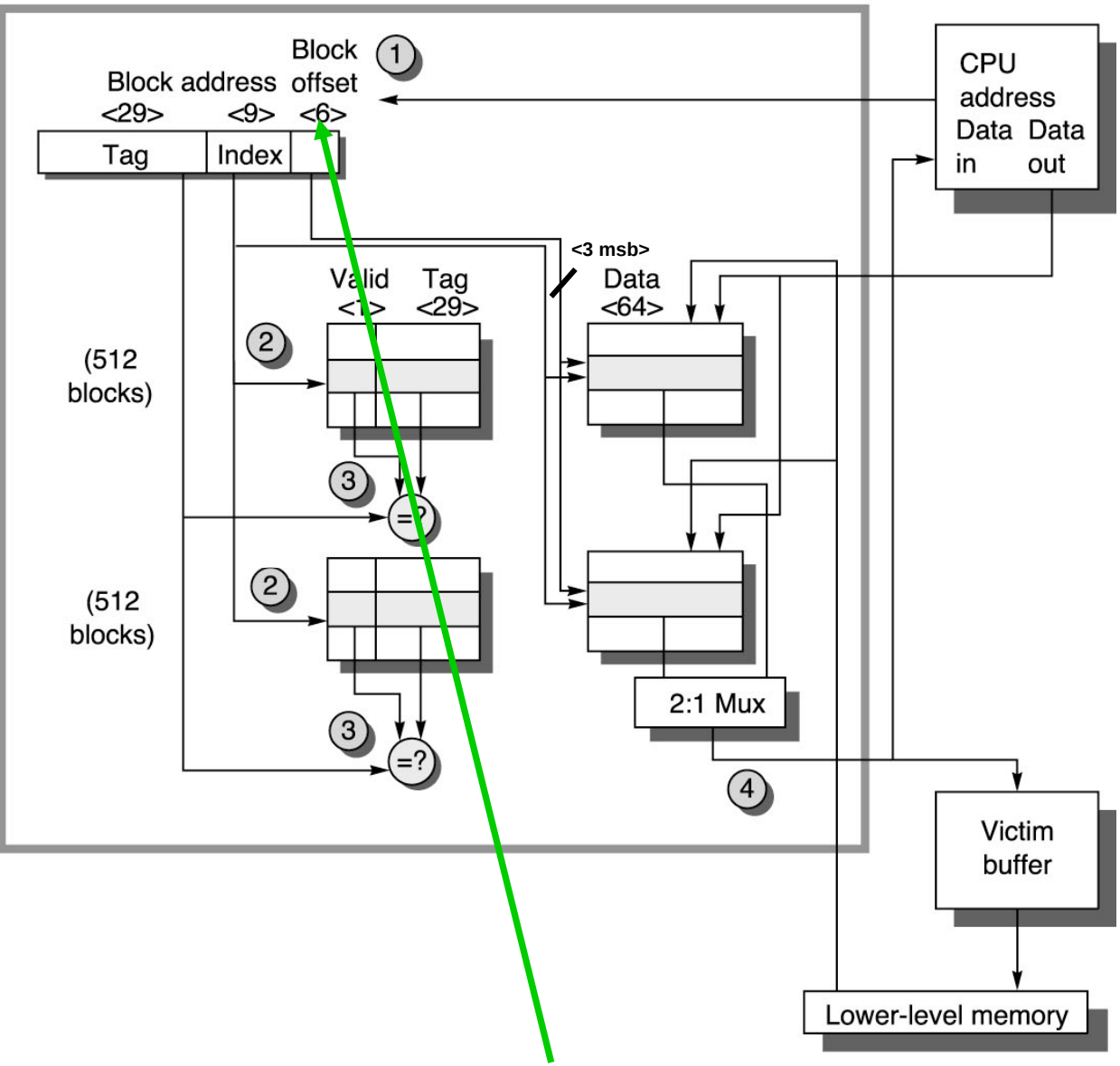
- Advantages:
  - Simple, easier to implement
- Cache is always clean
  - Read misses never result in writes to lower level
- Write stalls
  - CPU must wait for writes to complete
  - Use a write buffer

Write Misses

- Two strategies (used with both WB and WT)

- Write allocate
  - Make write misses act like read misses
  - Allocate the block in the cache then follow same steps as a write hit
- No-write allocate
  - A little weird- write misses do not write the cache
  - Block is modified only in lower-level memory
  - Blocks stay out of the cache until the program tries to read the blocks

- Normally:
  - Write back caches use write allocate (benefit from locality)
  - Write through caches use "no write allocate" (avoid redundant writes)



Example: Alpha 21264 Data Cache

- 64 Kbytes
- 64-byte blocks
- 2-way set associative
- Write back
- Write-allocate on a write miss

2^{index} = cache size / (block size * associativity) = # blocks / associativity

Unified vs. Split Caches

- Instruction and data streams are different
  - Instructions are fixed size, read-only, very local (except for branches)
- Two caches: one optimized for instructions and one for data
  - Drawback: Fixed capacity is split between the two caches
- Split cache:

$$\begin{aligned}\text{MemoryStallCycles} &= \text{NumberOfMisses} \times \text{MissPenalty} \\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{MissPenalty} \\ &= \text{IC} \times \left(\frac{\text{FetchMisses}}{\text{Instruction}} \times \text{FetchMissPenalty} \right. \\ &\quad \left. + \frac{\text{DataMisses}}{\text{Instruction}} \times \text{DataMissPenalty} \right)\end{aligned}$$

Chapter 5

Memory Hierarchy Part 2

Slides: W. Gross, V. Hayward, T. Arbel

Cache Performance

- Miss rate is independent of the speed of the hardware
- Better measure of mem. Hierarchy performance is Average Memory Access Time

AMAT = HitTime + MissRate x MissPenalty

AMAT and CPU Performance

$$\text{CPUTime} = \text{IC} \times \text{CPI} \times \text{CCtime}$$

- Assumes perfect memory hierarchy performance (all accesses were hits)
- Pipeline: assumes all memory accesses complete within one clock cycle
  - Hit time < CCtime
- How to account for real memory hierarchy?

CPU Performance with Imperfect Caches

$$\text{CPUTime} = \text{IC} \times \left(\text{CPI}_{\text{base}} + \frac{\text{MemoryStalls}}{\text{Instruction}} \right) \times \text{CCTime}$$

$$= \text{IC} \times \left(\text{CPI}_{\text{base}} + \frac{\text{MemoryAccesses}}{\text{Instruction}} \times \text{MissRate} \times \text{MissPenalty} \right) \times \text{CCTime}$$

Example

- UltraSparc III (in-order execution)
- Cache miss penalty = 100 cc
- Instructions: 1.0 cc (ignoring mem. stalls)
- Miss rate = 2%
- 1.5 mem ref/instructions on average
- Cache misses → 30 misses /1000 instr.

- Q: what is the impact on performance when the cache is considered?

Example

$$\text{CPUTime} = \text{IC} \times \left(\text{CPI}_{\text{base}} + \frac{\text{MemoryStalls}}{\text{Instruction}} \right) \times \text{CCTime}$$

$$= \text{IC} \times \left(\text{CPI}_{\text{base}} + \frac{\text{MemoryAccesses}}{\text{Instruction}} \times \text{MissRate} \times \text{MissPenalty} \right) \times \text{CCTime}$$

$\text{CPUTime}_{\text{cache}}$ = IC x (1.0+(30/1000) x 100) x *CC*

= IC x 4.00 x *CC*

- or use miss rate

$\text{CPUTime}_{\text{cache}}$ = IC x (1.0+(1.5 x 2% x 100)) x *CC*

= IC x 4.00 x *CC*

Cache Impact on Performance

- With low CPI (< 1) the relative impact of a cache miss is higher
- With faster clocks, a fixed memory delay yields more stall clock cycles

AMAT is not CPUtime !

- Example: 2 cache designs fitted to the same CPU CPI = 2 with perfect cache
  - Cycle time = 1.0 ns
  - 1.5 mem refs / instruction

- Both caches are 64 KB and have block sizes of 64 bytes, miss penalty = 75 ns

1. Direct mapped: 1ns hit time, mp = 75 cc,mr = 1.4%
2. 2-way set assoc.: 1.25 ns hit time (slows down system clock), mp = 60 cc, mr = 1%

Example (in ns, not cc)

$$\text{AMAT}_{\text{1-way}} = 1.00 + 0.014 \times 75 = 2.05 \text{ ns}$$

$$\text{AMAT}_{\text{2-way}} = 1.25 + 0.010 \times 75 = 2.00 \text{ ns}$$

$$\text{CPUTime}_{\text{1-way}} / \text{IC} = \text{CPI}_{\text{1-way}} \times \text{CCTime}_{\text{1-way}} = 2 \times 1.00 + (1.5 \times 0.014 \times 75) = 3.58$$

$$\text{CPUTime}_{\text{2-way}} / \text{IC} = \text{CPI}_{\text{2-way}} \times \text{CCTime}_{\text{2-way}} = 2 \times 1.25 + (1.5 \times 0.010 \times 75) = 3.63$$

- AMAT is lower for 2-way
- CPU time is lower for 1-way !
- Why?
- 2-way stretches clock cycle for ALL instructions even though there are fewer misses!
  - build the 1-way in this case

Out-of-Order Execution

- Miss penalty does not mean the same thing
  - The machine does not totally stall on a cache miss
  - Other instructions allowed to proceed
- What is MP for O-O-E?
  - Full latency of the memory miss?
  - The nonoverlapped latency when the CPU must stall ?
- Define:

$$\frac{\text{MemoryStallCycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{TotalMissLatency} - \text{OverlappedMissLatency})$$

Performance Summary

- Textbook Page 412 Figure 5.9 summarizes all the performance equations (12 of them!)
- AMAT = HitTime + MissRate x MissPenalty
- Cache optimizations will be studied next focussing on reducing hit time, miss rate and miss penalty (analogy to CPU performance equation)

Cache Optimizations

- Penalty reduction: multilevel caches, critical word first, read priority over writes, merging writes and victim caches
- Miss rate reduction: Block size, cache size, associativity, pseudoassociativity, compiler optimizations
- Parallelism: non-blocking caches, hardware prefeteching, compiler prefetching
- Hit time reduction: cache size and organization, address translation avoidance, pipelined caches, trace caches (brief coverage)

Reducing Miss Penalty – Multilevel Caches

- Reduce penalty by adding a cache to the cache !
- Penalty at a given level is determined by the AMAT of the next lower level
- E.g. penalty to replace a register is the load delay

$$\text{AMAT}_i = \text{HitTime}_i + \text{MissRate}_i \times \text{MissPenalty}_i$$

$$\text{MissPenalty}_i = \text{AMAT}_{i+1}$$

SO

$$\begin{aligned}\text{AMAT}_{L1} &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times (\text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2}) \\ &= \underbrace{\text{HitTime}_{L1} + \text{MissRate}_{L1} \times \text{HitTime}_{L2}}_{\text{New hit time}} + \underbrace{\text{MissRate}_{L1} \times \text{MissRate}_{L2}}_{\text{New miss rate}} \times \text{MissPenalty}_{L2}\end{aligned}$$

Multilevel Caches

- For small miss rates:
- Small increase in hit time
- Huge reduction in miss penalty because misses that go all the way to memory (global misses) are rare (mr1 x mr2)
- L1 should be fast for a small hit time
- L1 usually on-chip to reduce interconnect delays
- L2 should be bigger (typically off-chip) and should have a more sophisticated organization to reduce miss rate

Multilevel Exclusion

- What if a design cannot afford a L2 cache that is much larger than L1
  - Wastes most of L2 with redundant copies of what is in L1?
- In this case use multilevel exclusion
  - L1 data is never found in L2 cache
  - L1 miss results in a swap instead of replacement
- AMD Athlon (2x64Kb L1 and 256 Kb L2)

Critical Word First and Early Restart

- Observation: CPU normally needs just one word of the block at a time
  - Impatience! Don't wait for whole block to be loaded before sending the requested word and restarting the CPU
- Critical word first – request the missed word from memory and send it CPU as soon as it arrives – CPU continues while rest of block fills in
- Early restart – fetch the words in normal order, but as soon as the requested word arrives, send it to the CPU and let it continue execution

Priority of Read Misses Over Writes

- Write buffers for write-through caches case RAW hazards
- E.g. direct-mapped, assume 512 and 1024 mapped to same block

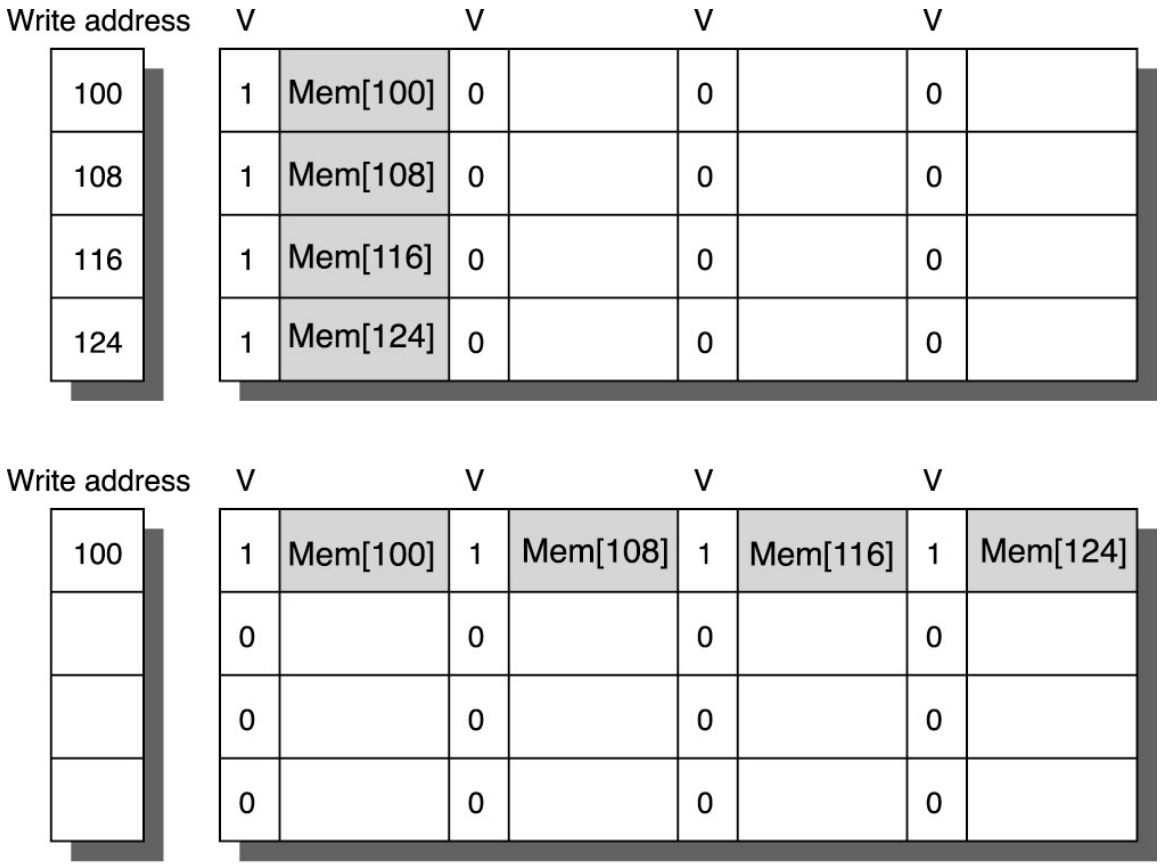
```
SW R3, 512(R0)  ; R3 in write buffer
LW R1, 1024(R0) ; miss, replace block
LW R2, 512(R0)  ; miss
```

- If write buffer has not completed store then wrong value written to cache block and R2
- Soln: on read miss, check write buffer for conflicts and if memory system is available, let read miss continue

Merging Write Buffer

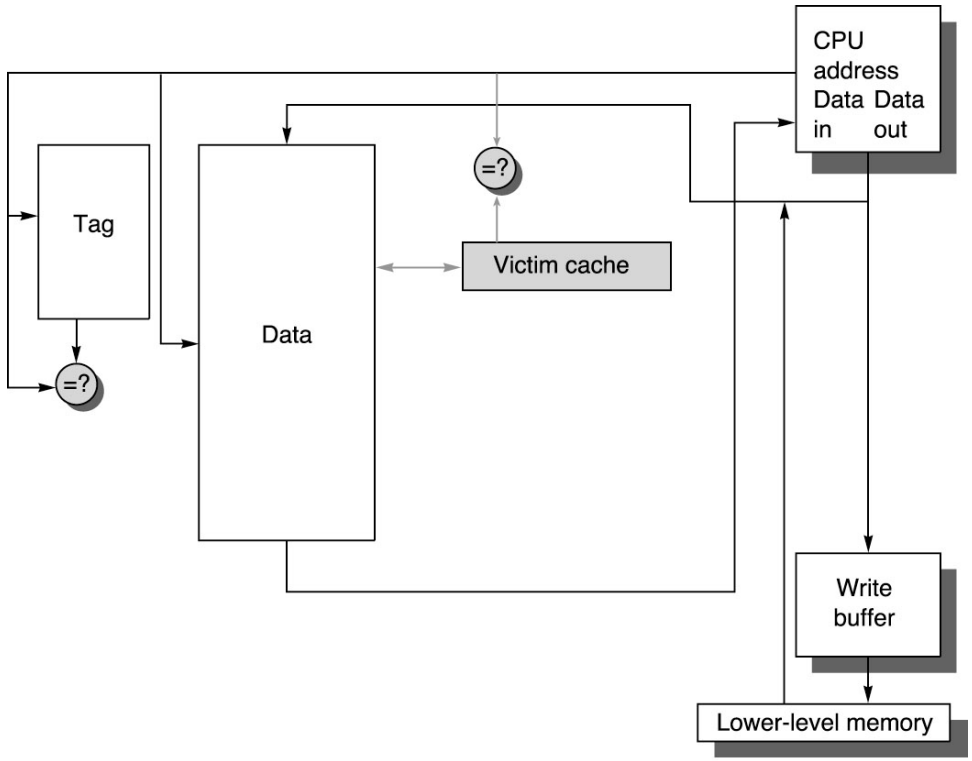
- When writing to the write buffer, check if address matches an existing write buffer entry
  - Combine data with that write – write merging
  - Multiword writes are faster than one word at a time

Merging Write Buffer



Victim Caches

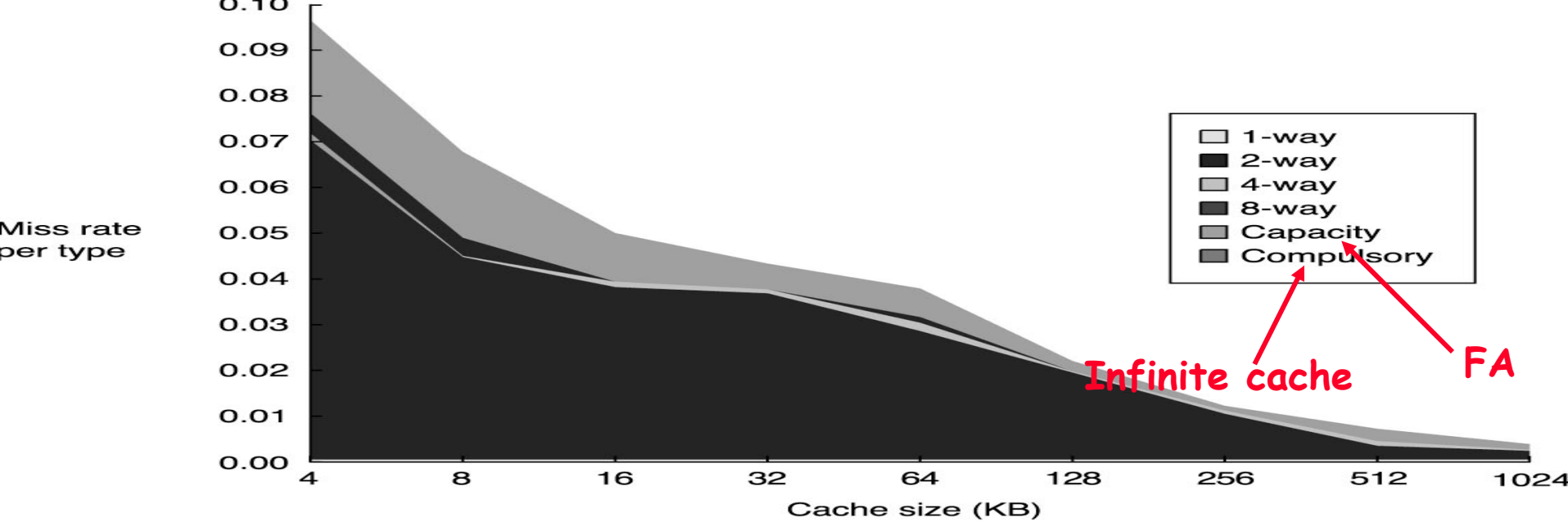
- Remember what was discarded in case it is needed again
- Small fully associative cache between a cache and its refill path
- AMD Athlon – 8 entry victim cache

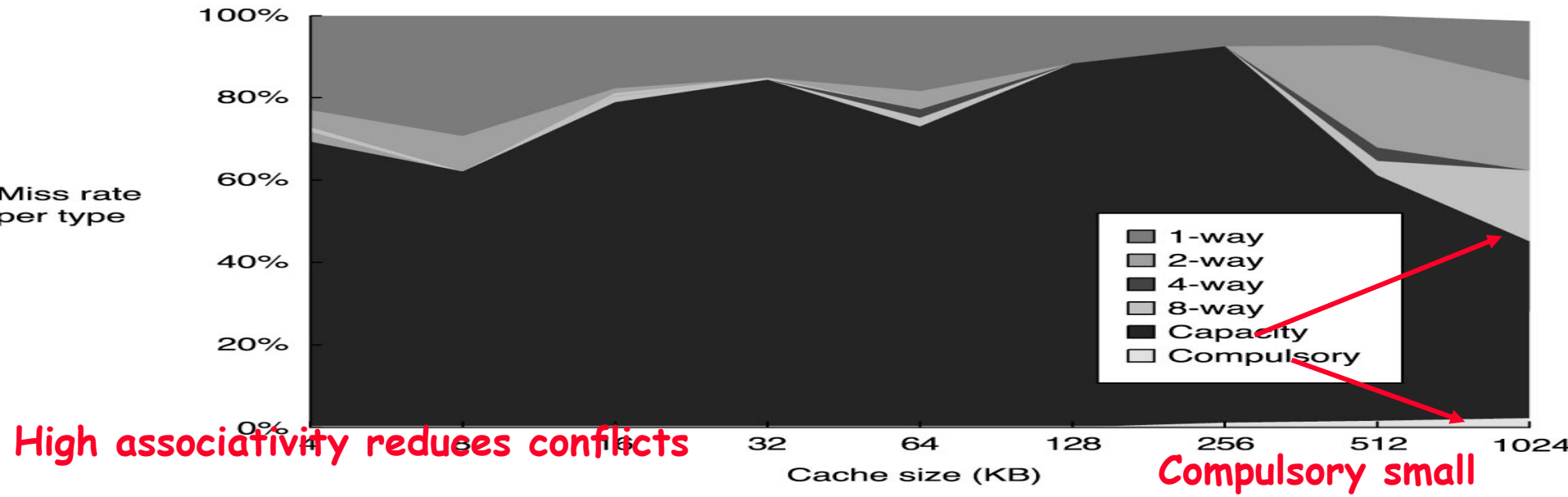


Miss Rate Reduction

- 3 kinds of misses

1. Compulsory – first access to a block cannot be in cache
2. Capacity – cache cannot contain all blocks needed during execution of a program
3. Conflict – for set-associative or direct mapped – if too many blocks map to a set
   - Hits in FA cache become misses in an n-way SA cache if there are more than n requests to a popular set





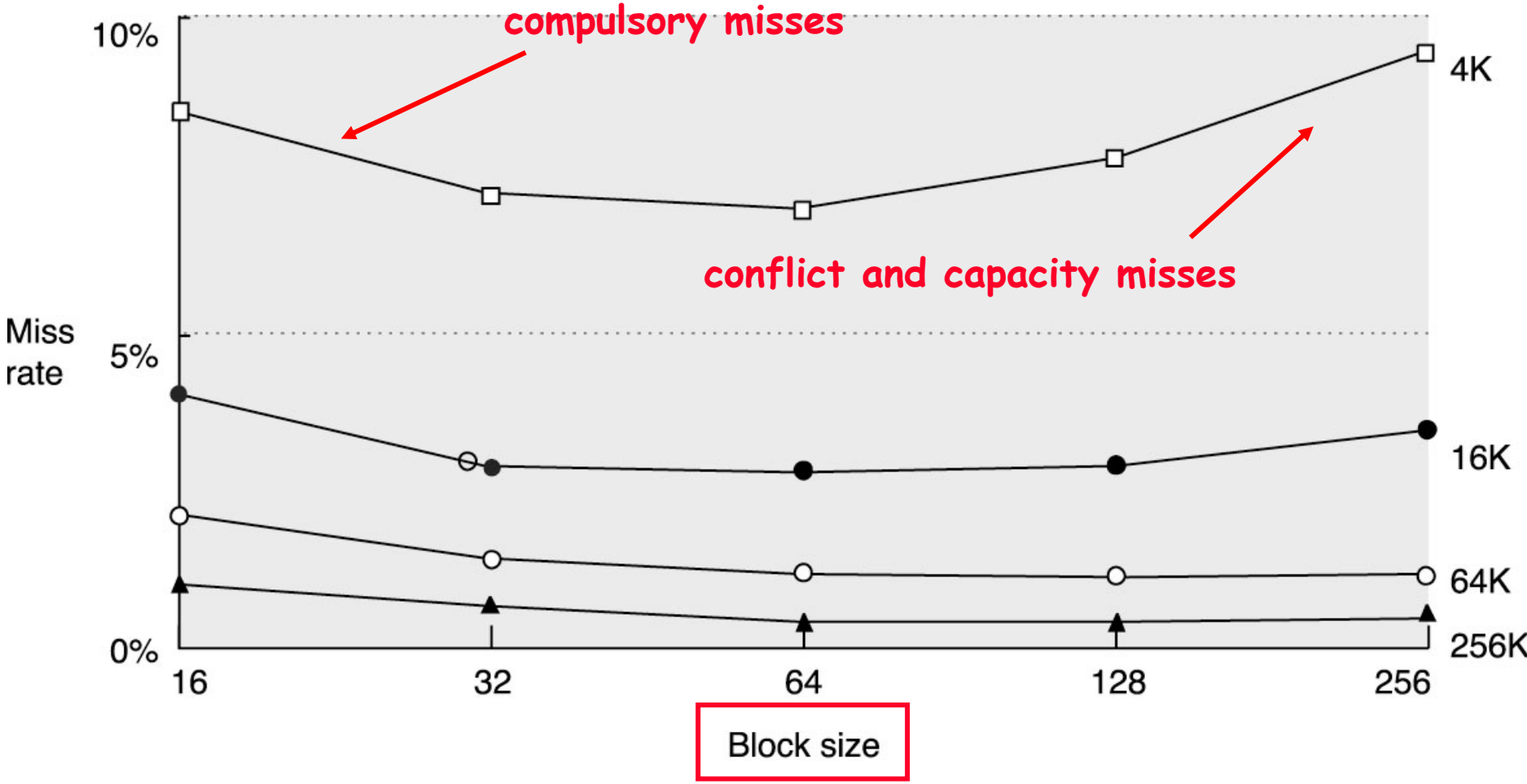
High associativity reduces conflicts

Compulsory small

Capacity misses require large caches

Tradeoffs

Larger blocks exploit spatial locality (but might increase MP)



Other Techniques to Reduce Miss Rate

- Larger caches
  - Cost tradeoff
  - Higher hit times
- Higher associativity
  - Rule of thumb: Direct mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2
  - Higher hit times (search)
  - Way prediction: use extra bits to predict the next block in the set that will be accessed (similar to branch prediction)
- Pseudoassociativity – cheap form of associativity for direct mapped caches.
  - On a miss a second entry is checked, say by inverting index bits ('pseudo-set')
  - One fast hit and one slower one

Compiler Optimizations

- Matrix and vector code can be written to have an impact on cache performance – improve spatial locality
- Instead of going through arrays in whatever order the programmer chose, operate on all the data in a cache block
- Loop interchange: x[i][0] is contiguous with x[i][1]

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

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

Loop fusion:

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        a[i][j] = b[i][j] * c[i][j];

for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        d[i][j] = a[i][j] + c[i][j];
```

Becomes:

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j) {
        a[i][j] = b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

The second version of the code increases temporal locality since each element of `a` and `c` is reused in the next statement.

Blocking:

One attempts to stride though arrays by amounts that do not exceed the cache size.

```
for (i = 0; i < N; ++1)
    for (j = 0, r = 0.0; j < N; ++j) {
        for (k = 0; k < N, ++k)
            r += y[i][k] * z[k][j];
        x[i][j] = r;
    }

for (jj = 0; jj < N; jj += B)
    for (kk = 0; kk < N; kk += B)
        for (i = 0; i < N; ++i)
            for (j = jj, r = 0.0; j < min(jj + B, N); ++j) {
                for (k = kk; k < min(kk + B, N, ++k)
                    r += y[i][k] * z[k][j];
                x[i][j] = r;
            }
```

This way, tight-loop processing occurs within sub matrices of size `B * B` which are designed to fit in the cache.

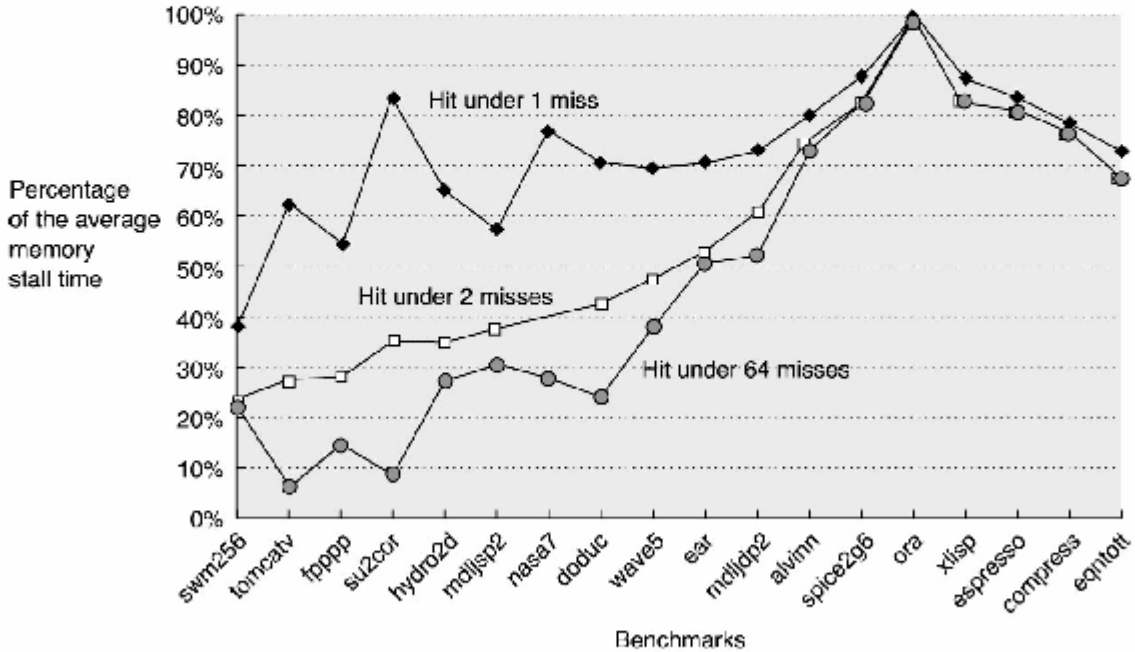
Chapter 5

Memory Hierarchy Part 3

Slides: W. Gross, V. Hayward, T. Arbel

Parallelism.

For out-of-order execution processors, there is no need to stall the processor on a miss since instructions not pending on a load can proceed. A cache organization which allows hits during the resolution of a miss is called "hit under miss" *non-blocking* cache. The level of sophistication varies with the number of allowed pending misses before blocking the cache.



As measurements show, this is extremely important for FP matrix code, but useless for integer code.

With *prefetching*, the idea is to fetch one or several blocks ahead, instead of one on a miss. It is not the same thing as designing a larger block size because the two fetches are done in parallel.

The common organization is to add yet another buffer or buffers on the instruction or the data stream. These *prefetch buffers* have the size of a block. Upon the next miss, the prefetch buffer(s) is/are checked before going to the lower memory. Just four buffers can reduce the miss rate by almost half.

If the hardware does not support prefetching, the compiler can use this technique by inserting load instructions. There are even special prefetch load instructions which do not create exceptions such as page faults called *nonbinding* prefetch.

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

```
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; ++i) {
    prefetch(a[i][j+7]);
   a[i][j] = b[j][0]*b[j+1][0];}
```

What the prefetch instructions do is load the cache with the blocks containing data needed 7 iterations later achieving a tremendous amount of overlap.

Hit Time Reduction.

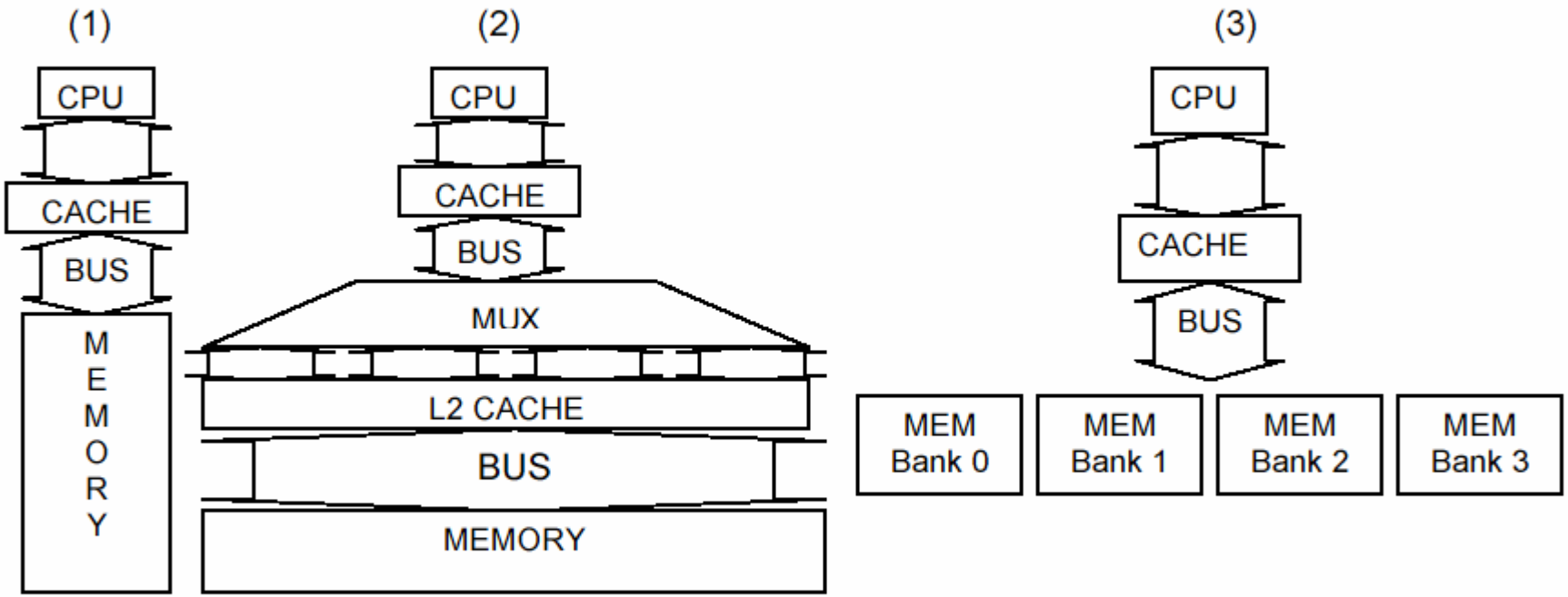
Hit time is also important because it is on the critical path of (hopefully) the majority of memory accesses.

The first necessity is plainly at the circuit design level. So the first level cache is nowadays on-chip, and electronically optimized. The same can be said of the tags for the second level cache.

For the purpose of this course, we skip the details of hit time reduction (pp. 443-448).

Main Memory Organization

The main memory can also benefit from architectural improvements. The basic idea is to increase the traffic between the main memory and the cache(s). The tradeoff is described by this simple diagram.



1. Base design: all data paths one word wide.
2. Expensive improvement, not necessarily efficient with much extra hardware.
3. *Interleaved* design has several advantages: same bus width, expandability.

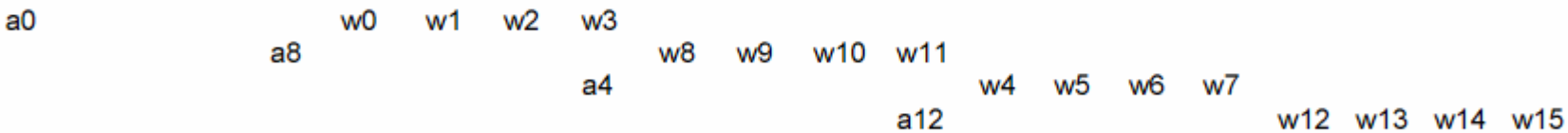
Interleaved Memory Organization

The memory is organized in banks such that Bank 0 contains all the words whose addresses are 0 modulo the number of banks, etc.

The advantage is to make it possible to overlap or *interleave* memory accesses because the banks are read or written sequentially. Here is the address layout for four banks. A block is read by hitting the banks sequentially.

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

For example, there would be the pattern of activity for consecutive replacements of blocks at addresses 0, 8, 4, and 12. a0 means address sent, wx means word at address x supplied after a given latency:



Here, the number of banks is equal to the number of cycles needed to access a word. In this condition, the bus is kept 100% occupied. In general, it is desired that

number of banks ≥number of access cycles

(access cycles in CPU clock cycles) to keep the bus fully occupied.

Some numbers

Assume that 4 CC are needed to send an address, 56 CC are needed for word access, and 4 CC spent in sending one word to the cache.

- A 4-word block replacement for the 1-word wide bus organization costs:
  $4 \times (4 + 56 + 4) = 256$ CC.
- Increasing the bus width by n, ideally divides this number by n.
- With the interleaved design, at no hardware cost, the block replacement costs:
  $4 + 56 + (4 \times 4) = 76$ CC,
  the equivalent of tripling the bus width at almost no hardware cost.

Consider a machine of ideal CPI of 2 and a benchmark with 1.2 memory accesses per instruction, and a cache such that

Block size (word):	1	2	4
Miss rate (%):	3	2	1.2

All the design cases are here:

	1 word block	2 word blocks	4 word blocks
1 word bus	2+(1.2 x 0.03 x 64) = **4.30**	2+(1.2 x 0.02 x 128) = **5.07**	2+(1.2 x 0.012 x 4 x 64) = **5.69**
interleave	same	2+(1.2 x 0.02 x (4+56+8)) = **3.63**	2+(1.2 x 0.012 x (4+56+16))= **3.09**
2 word bus	N/A	2 + (1.2 x 0.02 x 64) = **3.54**	2+(1.2 x 0.012 x 2 x 64) = **3.84**

Interleaved memory organization is essentially a method to take advantages of caches to pipeline memory accesses; a re-occurring theme in CA. Needless to say, this design is wide-spread.

Memory Technology

Memory technology:

DRAM, SRAM, ROM, Flash, SDRAM, RAMBUS, RDRAM.

We skip this subsection.

Virtual Memory

- Original motivation: to increase the memory capacity of the computer beyond the size of the main memory
- The problem:
  - If a program became too large to fit into memory…
- The original solution (before VM):
  - The programmer was responsible for dividing the program up into mutually exclusive parts that would fit into main memory
  - The programmer was also responsible for making sure the correct part (overlay) was loaded in to the main memory at the proper time

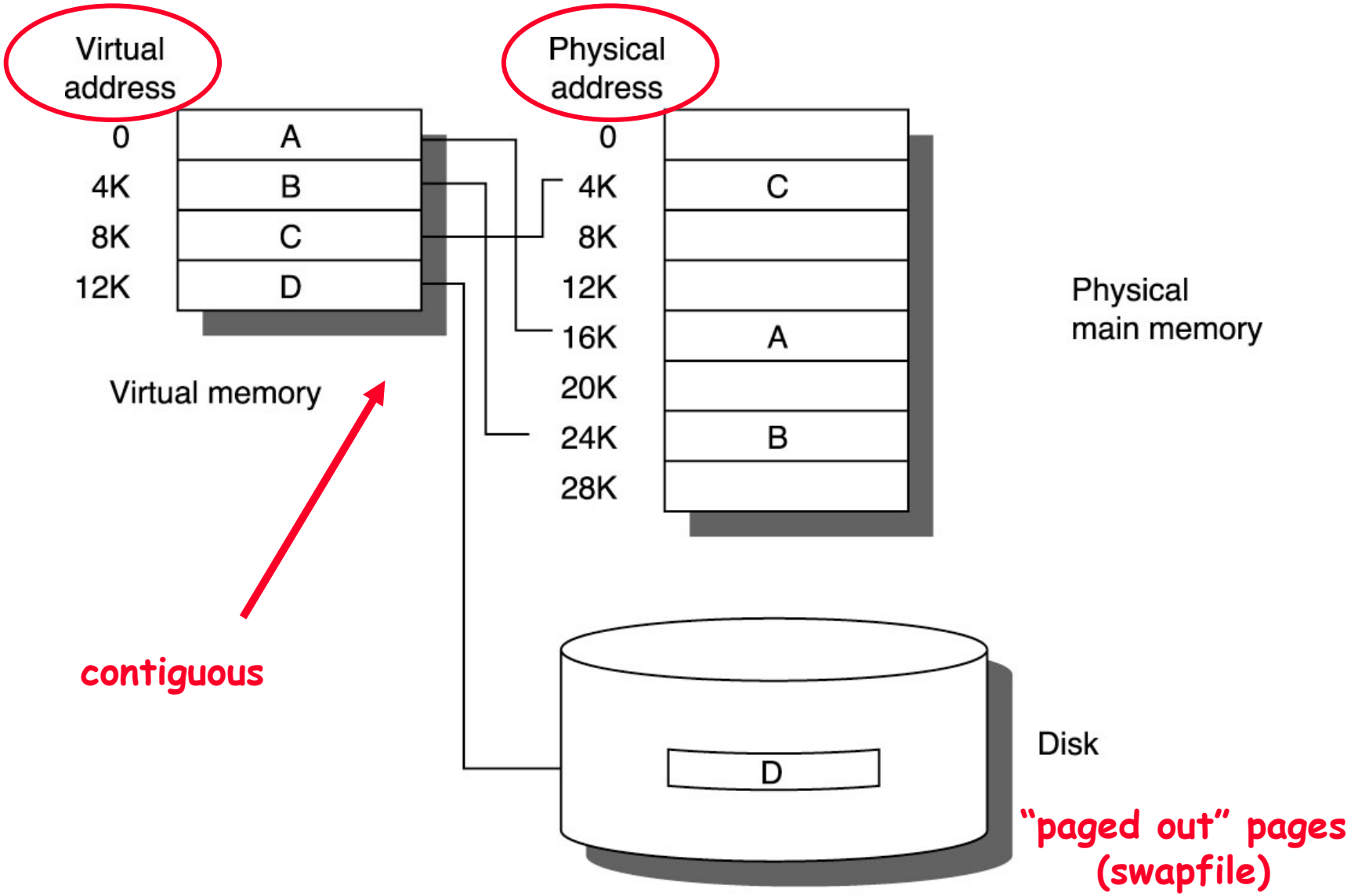
Virtual Memory

- With "virtual memory", the disk is used as the lowest level in the memory hierarchy
- The address space is the range of memory addresses
  - usually much larger than the capacity of the main memory
  - E.g. 32-bit addresses => 2^{32} ~ 4 x 10^9 addresses (usually 4 Gigabytes capacity)

More Motivation for VM

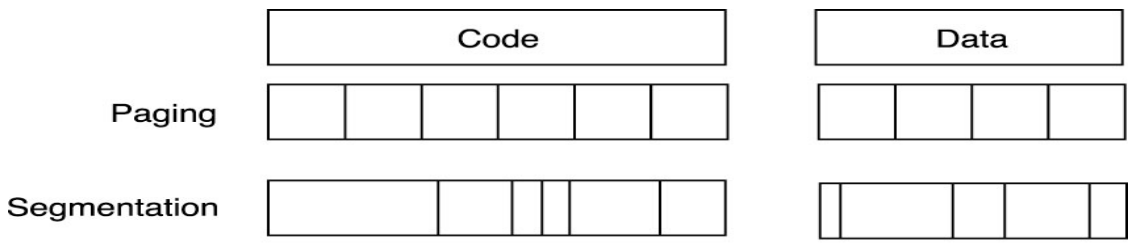
- But there are other motivations for VM, which are just as important…

- Multitasking
  - Many processes (programs) are sharing the memory space
  - Each one thinks it has a contiguous chunk of memory – hide details from each process (# of processes,size of processes..)
  - Memory protection => don't let a process access another's memory
- Relocation
  - Allows a program to run anywhere in memory
  - maps the addresses generated by the compiler to the real address of the memory in the main memory or disk



Pages and Segments

- Chunks of memory are called pages or segments (instead of blocks in caches)
  - Pages are a fixed size (con: internal fragmentation)
  - Some machines use variable size pages called segments (hard to replace since you need to find contiguous, variable sized, free space)
- A reference to a page that is on the disk => page fault

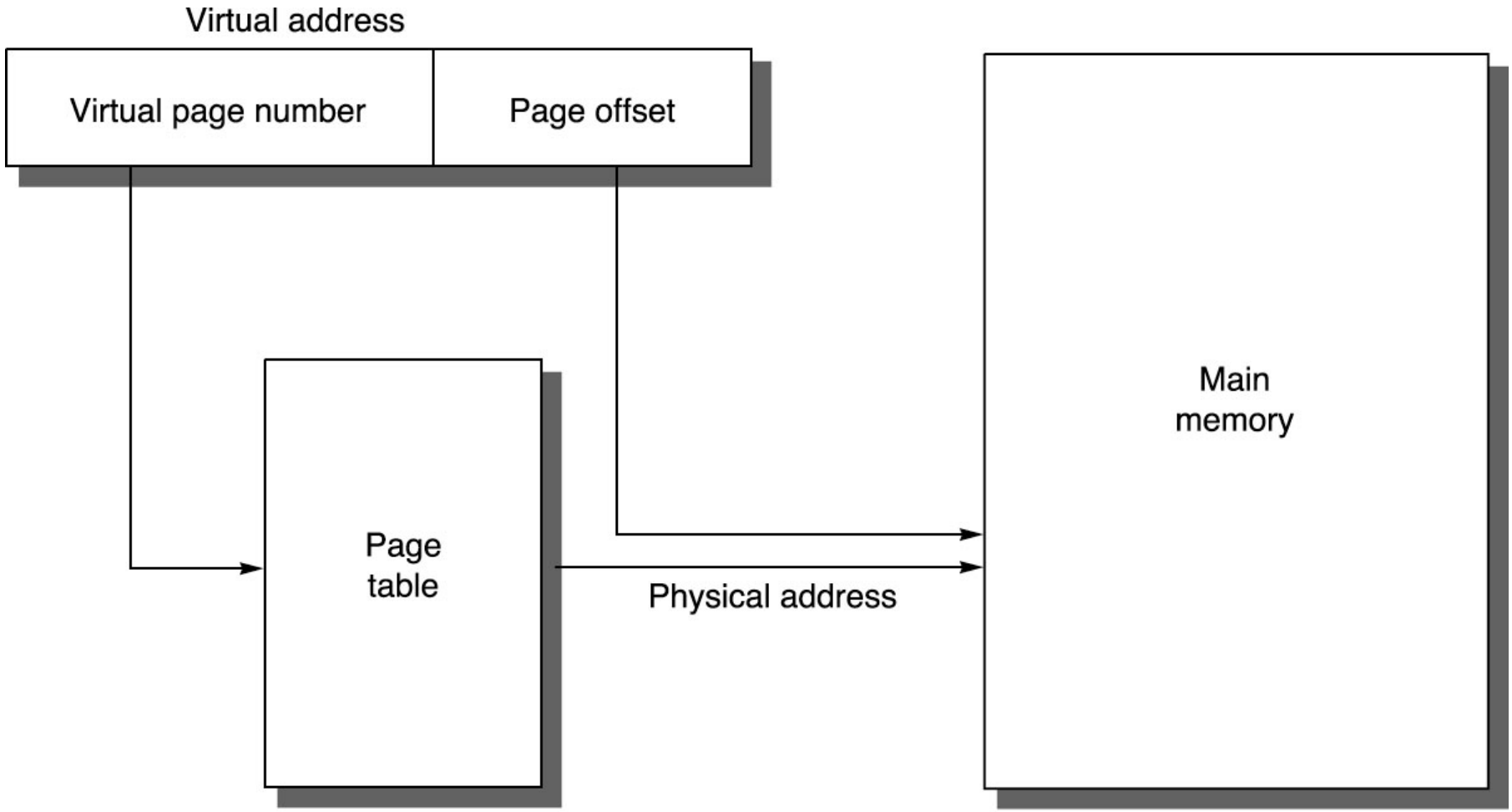


4 Questions

- Block placement
  - Miss penalty is huge ! (1,000,000 to 10,000,000 cycles)
  - Go for lower miss rate at expense of more complex algorithm (O/S)
  - VM uses a fully associative strategy (pages can be placed anywhere in main memory)
- Block Replacement
  - Minimize page faults !
  - LRU replacement (set "use bit" when a page is accessed. O/S keeps track of them)

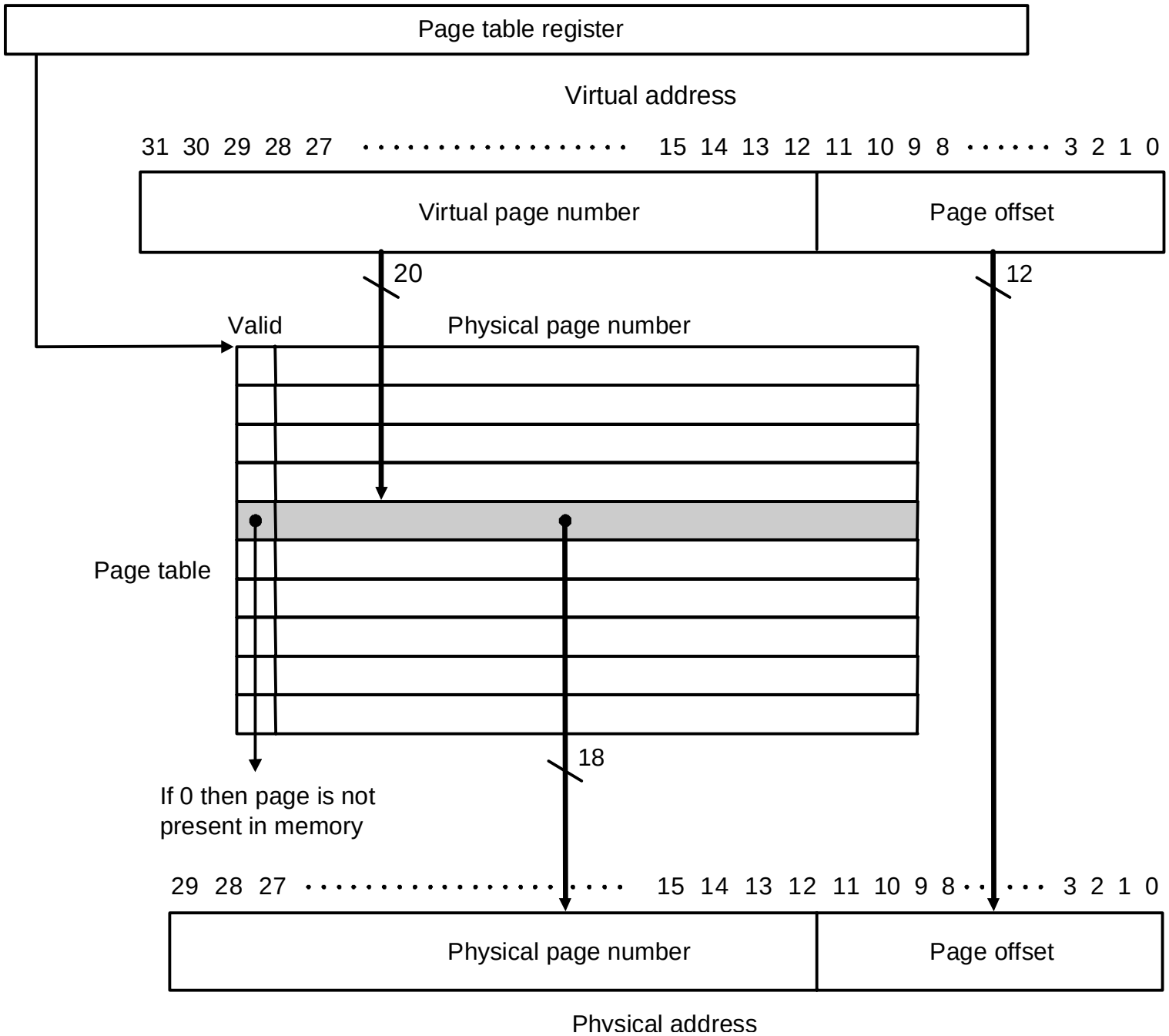
4 Questions

- **Write strategy**
  - Write back (avoid writing to disk whenever possible)
- **Block ID**
  - Need to translate (map) the virtual address on-the-fly to a physical address
  - Store the mapping in a page table (maintained by O/S)



Page Tables

- Index with the virtual page #
- The page table stores the corresponding physical address
- Each process gets a page table
- Page tables are stored in main memory (and can therefore be in the cache)
- Page tables can be large


Page table register
Virtual address
31 30 29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Physical page number
Page table
18
If 0 then page is not present in memory
29 28 27 ..... 15 14 13 12 11 10 9 8 ... 3 2 1 0
Physical page number
Page offset
Phvsical address

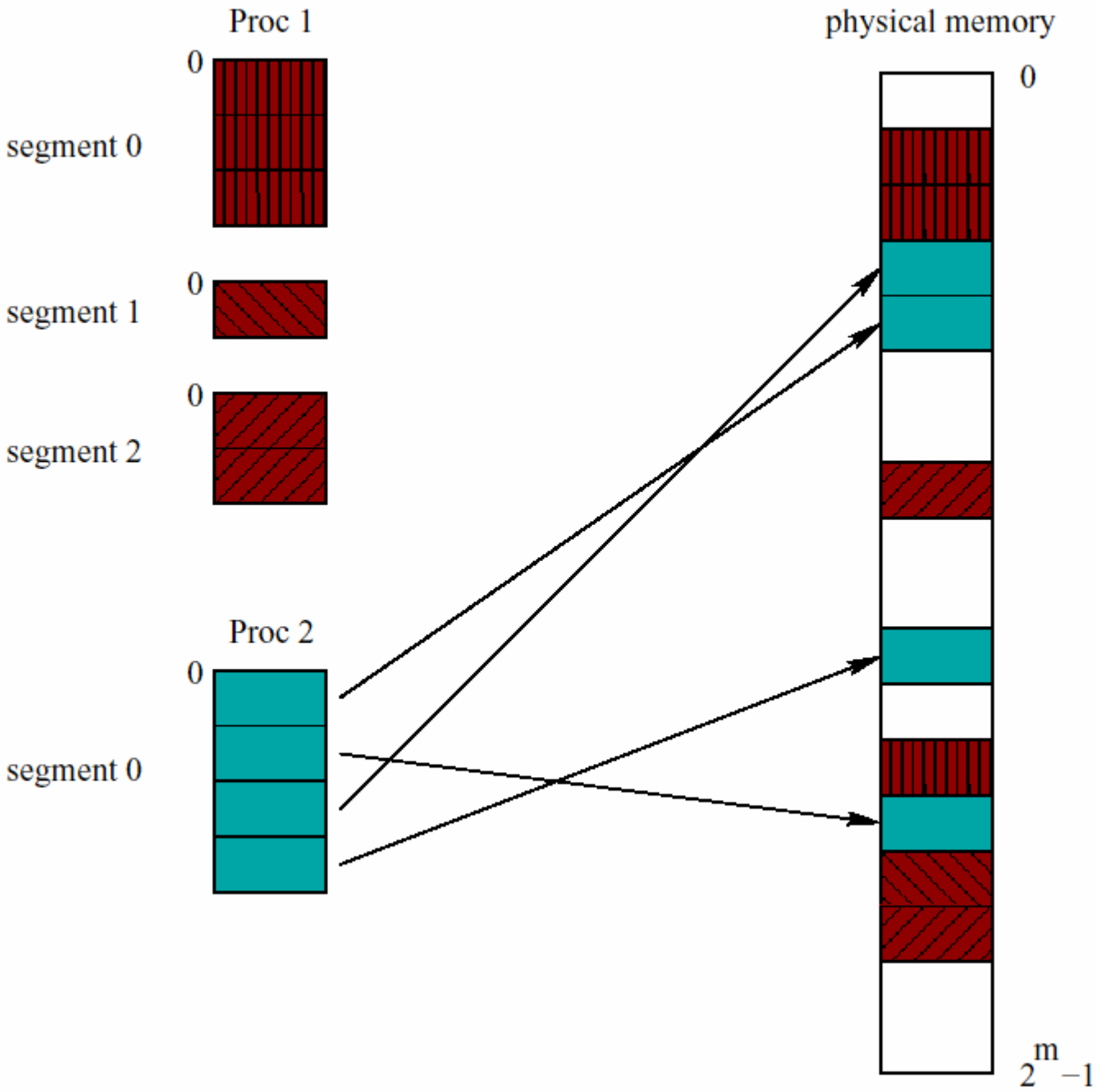
Page Table Example

- 32-bit virtual address
- 4 KB pages
- 4 bytes per page table entry

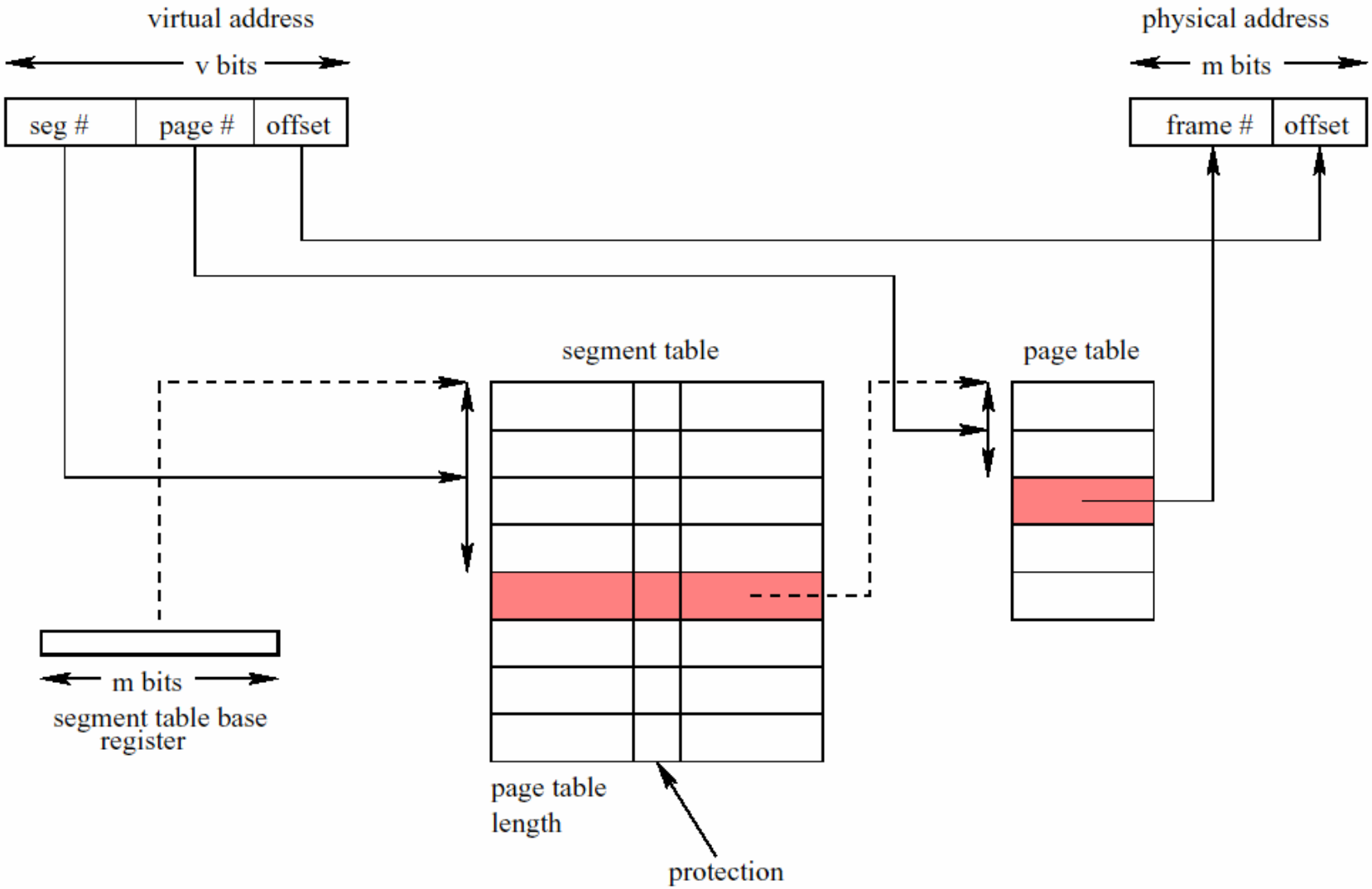
- The page table would be $2^{32}/2^{12} \times 2^2 = 2^{22}$ bytes (4 MB)

Combining Segmentation with Paging

- Processes are often divided into several spaces for code, data, stack
- Segments can be used to dynamically adjust each process' memory usage
- Pure segmentation is not often used, but segments can be divided into multiples of pages



Courtesy: UW CS350



Courtesy: UW CS350

Fast Address Translation

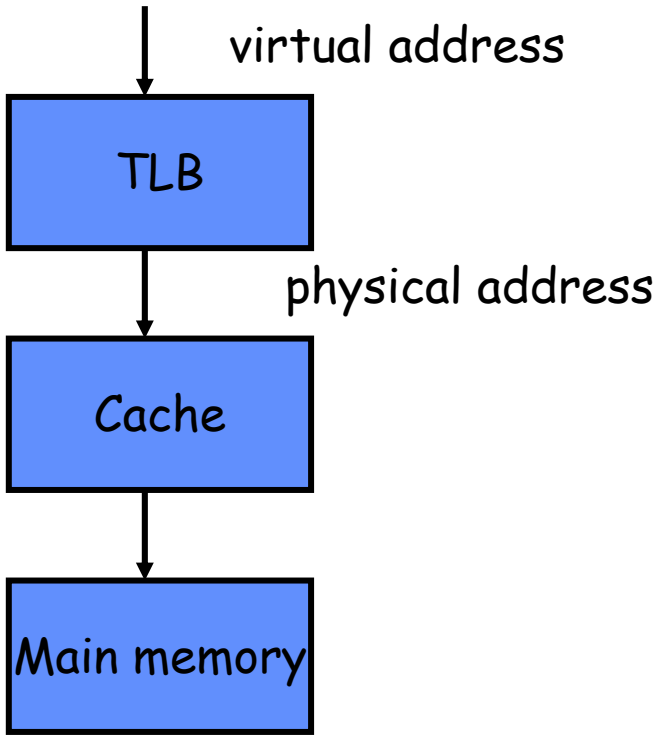
- Paging means that every request results in two memory accesses
  - One to read the page table to get the physical address
  - One to get the data
- This is very costly. Especially wasteful since locality tells us that consecutive accesses are likely to be on the same page
- Why redo the address translation every time?
- Solution => cache the most recent translations !
  - notice how in a small number of architectural techniques keep coming up in this course…e.g. caching, using the past to predict the future, pipelining, parallelism…

Translation Lookaside Buffers (TLB)

- The name of the special cache used to remember the most recent address translation is the translation lookaside buffer
- Just like a cache – tag is portion of virtual address and data is the physical page number (along with a field used for protection, valid bit, use bit, and dirty bit for the memory page)

TLB Placement

- The TLB is usually inserted between the CPU which supplies virtual addresses and the memory system which requires physical addresses

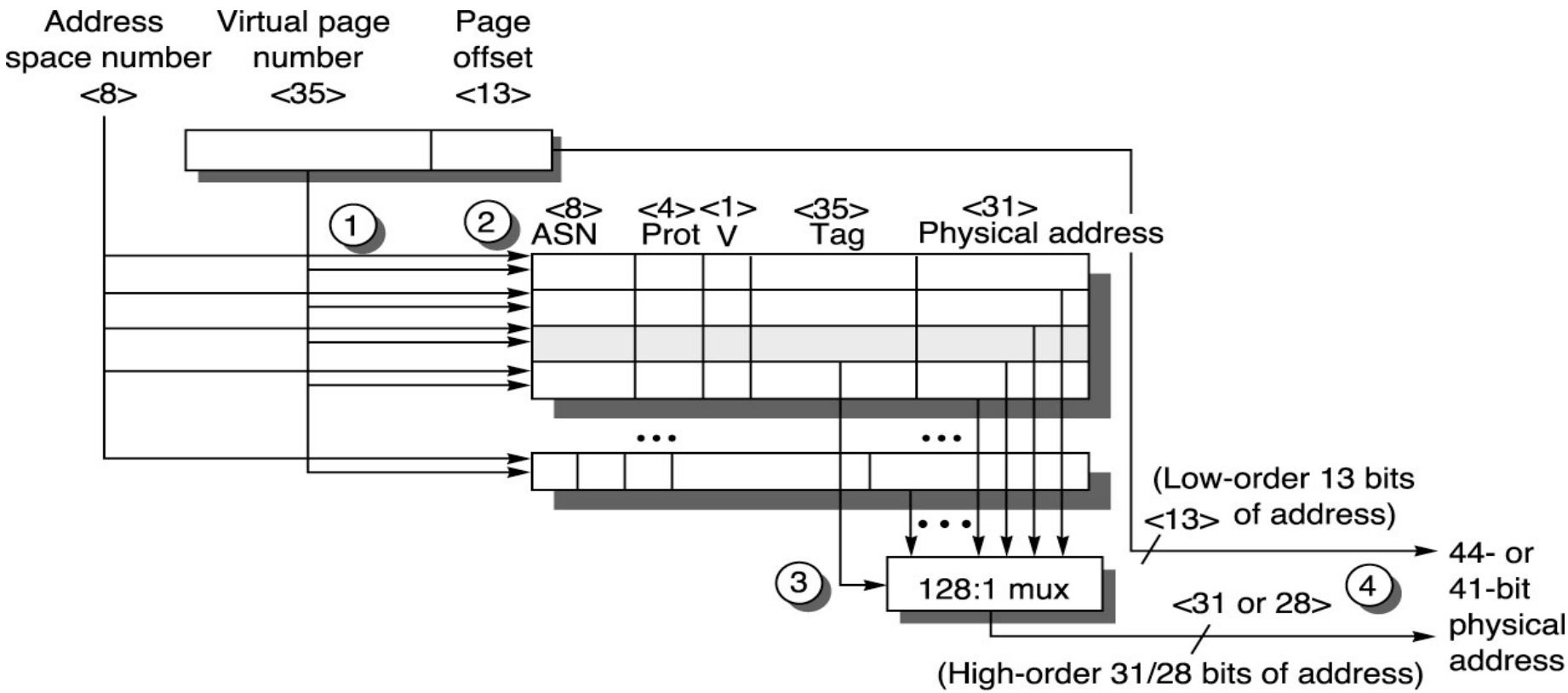


TLB Placement

- This raises the question of whether caches should operate on physical or on virtual addresses
  - Caches can be virtually or physically tagged and indexed.

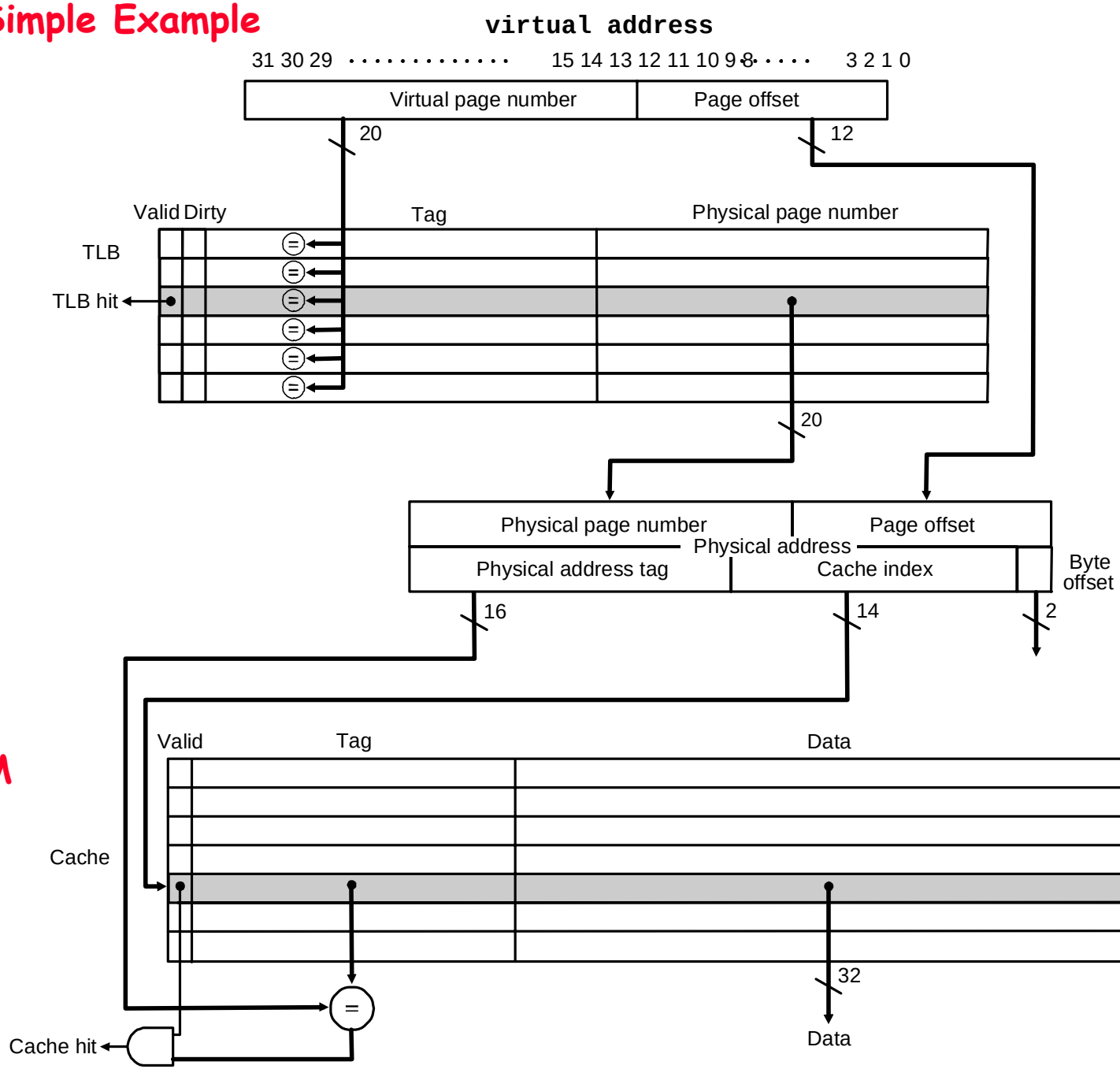
process #

Alpha 21264 TLB



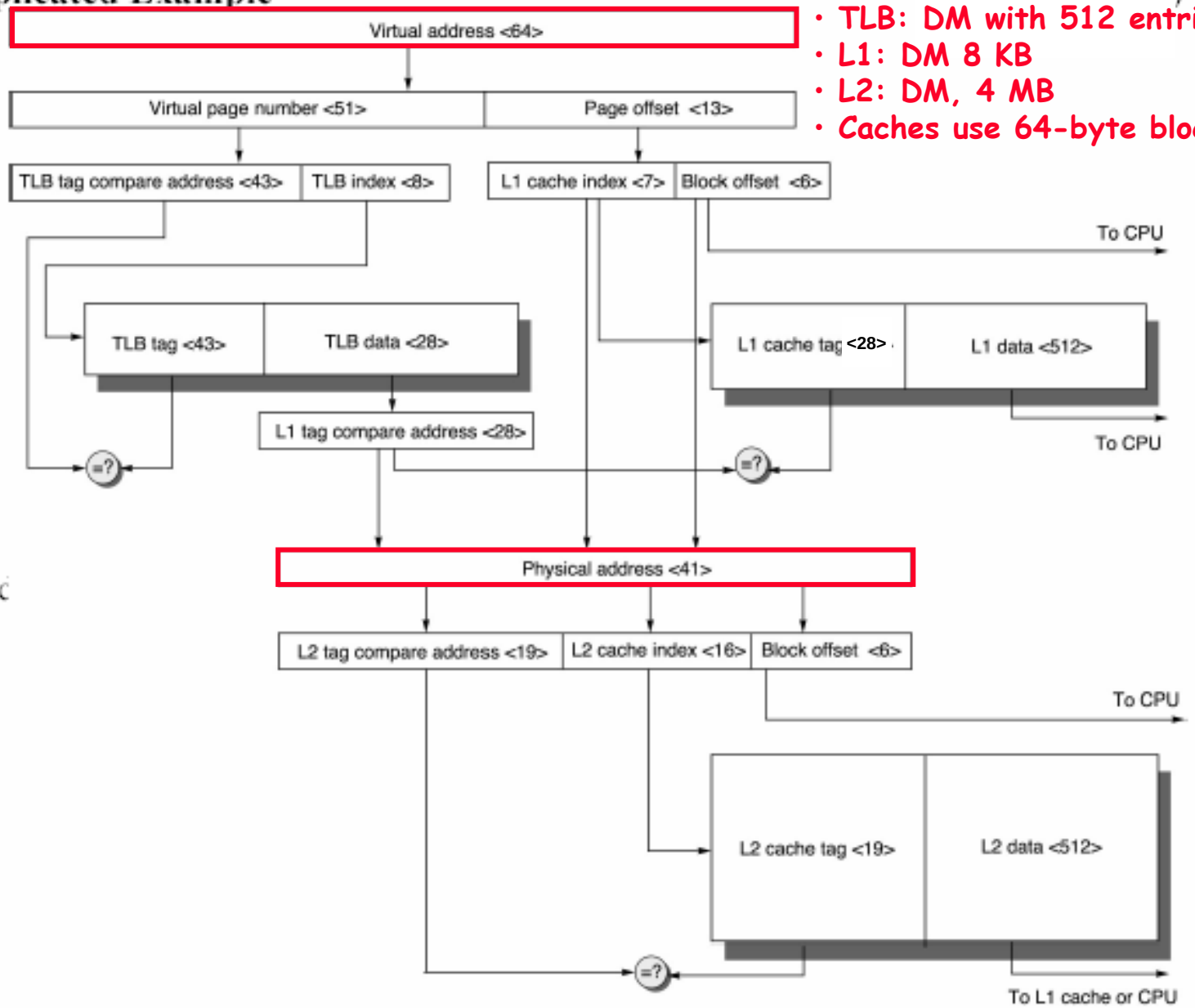
- The Alpha TLB is FA (in general, can use any strategy)

A Simple Example



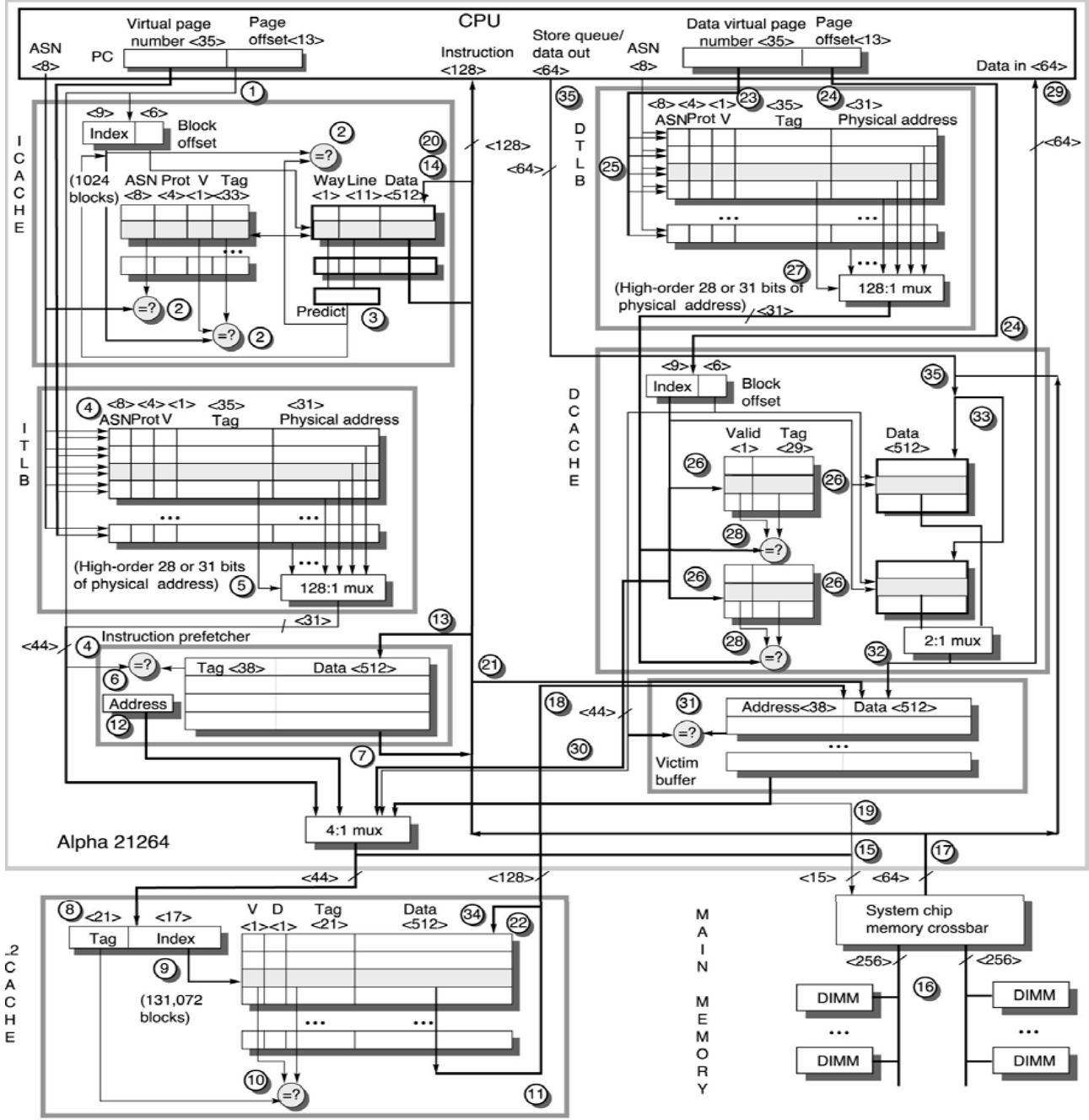
A More Complicated Example

- Page size 8 KB
- TLB: DM with 512 entries
- L1: DM 8 KB
- L2: DM, 4 MB
- Caches use 64-byte blocks



L1 is virtually indexed but physically tagged
L2 is physically indexed and tagged.
On a L1 miss, the physical address is used to fetch a block from L2.

Alpha 21264 Memory Hiearchy (page 483)



Protection

Multitasking (or *multiprogramming*), i.e. the possibility for a computer to give the illusion that many processes run simultaneously, is one of important the features provides by VM. Just like the possibility of allocating virtual space to processes had to be supported by hardware features to let many processes share the physical memory, multitasking also requires simple but crucial CPU support.

The basic idea is *time sharing*, whereby a collection of processes share the time line by slicing it into short periods, short enough (a few ms i.e. millions CC) to be unnoticeable to users. The OS, using a scheduling algorithm, allocates these time slices to each process in turn. The exact manner is a topic in operating systems. Here, we are interested in its hardware aspect.

Each time the CPU is about to run a new process, the process which was running is interrupted by the kernel (an exception which synchronous, coerced, resumable, and non-maskable), its state is saved and control is given to another process. This is called a *process switch*. Since this happens in virtual space, pages form the old process can be paged out to make room for the new. One key point is to make the process switch efficient. The TLB does that well: the PTEs of the old process are invalidated as well as the cache by turning off the V bits and the whole machine starts from fresh.

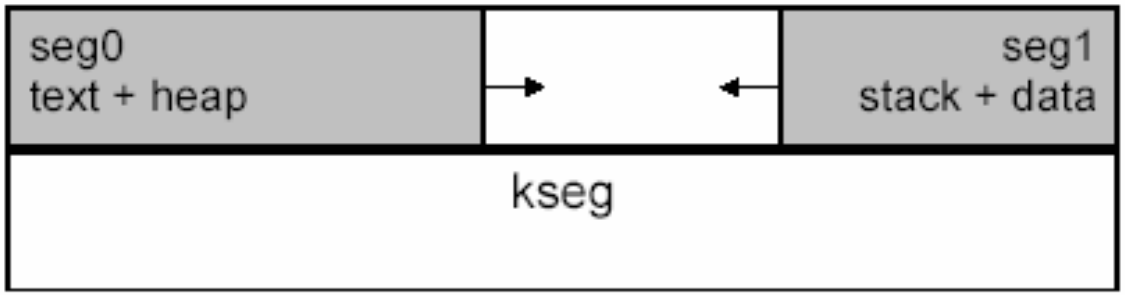
What's missing in this picture is how to prevent a process from illegally accessing the space allocated to others. There exists several hardware mechanisms to make this efficient (recall that **every** memory access has to be checked for validity). We describe the simplest and the most elegant scheme used by the family of Unix OS's.

The first element is those protection bits in each PTE (i.e. for each page). They operate like the permission bits for Unix files: read, write, execute: data that cannot change, data that can change, and code (there is usually a hierarchy of permissions).

The other piece of hardware is a pair of registers, *base* & *bound*. The address generation hardware makes sure that all addresses are such that $base \leq a \leq bound$, or that $(base + a) \leq bound$ (if not the, offending process is terminated). To see how this works in general, notice that any process consists of four parts:

- Text. The code itself, the instructions (read only, fixed length).
- Data. The portion of memory allocation which has an initial value when a process starts. In C this is all the static variables (read/write, fixed length).
- Stack. A portion of memory which grows and shrinks (read/write, variable length).
- Heap. A portion of memory that a process can claim (malloc) and release for later use (free) (read/write variable length).

The OS divides the virtual address space into segments called seg0, seg1, and kseg:



The hardware translation mechanism interprets the two most significant bits in a special way. The first distinguishes addresses in seg0 which grow upward, from those in seg1 which grow downward (makes it easy to determine when they cross).

The second most significant bit distinguishes addresses produced by ordinary processes from addresses produces by the kernel addresses.

The whole computer operates in one of two modes. In *kernel mode* all addresses have the kernel bit on and are allowed to read and write everywhere. In *user mode*, the hardware checks the base and bound registers and the protection bits and the interrupt masking mechanism is disabled. When the computer is kernel mode, only the clock and I/O events can interrupt it and the protection mechanism is disabled. Using clock interrupts, the kernel's scheduler periodically gives control by time slices to each user process in the machine. During each time slice, it runs in user mode.

This way protection is ensured, whatever a user process does cannot affect the machine. Since the user processes cannot anything but compute, one more mechanism is required for services like I/O, allocating more space, launching other processes, or passing data from one process to another. All this is done using *syscalls*, that is traps to invoke system services.

An example: what happens when one type ^C because a program must to be terminated because it is in an infinite loop?

The keyboard sends an interrupt to the CPU when pressed. CPU switches to kernel mode to run the keyboard driver and stops whatever it was doing, probably running that infinite loop during a large number of time slices. The driver get the character ^C which is recognized as special and let the kernel know of it. The kernel simply removes this process from the list of processes to be scheduled on the basis that this ^C came from the keyboard that launched it. All this happens by switching back and forth from user to kernel mode.

Another example: a process exceed its bounds trying to write in the space of another process. A memory exception is generated which wakes up the kernel. From the process tables, the offending process is found and removed from the schedule. The user is given a "core" file, that is an exact copy of the state of the machine when the exception occurred so debugging can be done by "post-mortem analysis".