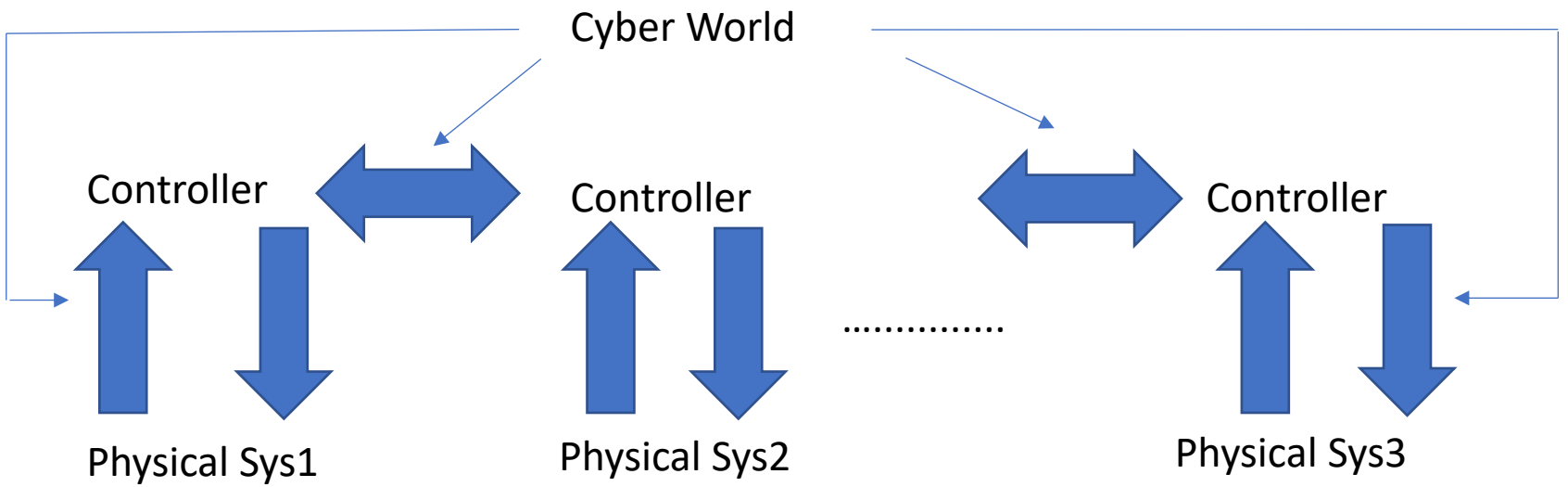
Computational Foundations of Cyber Physical Systems (CS61063)

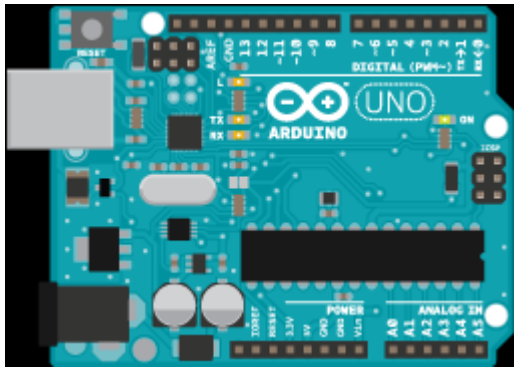


- Soumyajit Dey
- CSE, IIT Kharagpur

CPS Compute Platforms

- CPS involves significant on-board computation
  - Signal processing – filtering the plant state data
  - State estimation
  - On-board intelligence (can run several optimizations for real time problem solving)
- Low power computation
  - Typically performed with help from on board battery
- Need to use low power processors instead of workstation class processors
  - RISC CPUs- ARM, PowerPC
  - Microcontrollers – Atmel (8 bit RISC ATmega328)

Arduino Uno – a popular choice



- 8 bit Atmega / 32 bit ARM
- Supports lot of peripherals
  - USB
  - Serial port
  - Ethernet
  - Motor control and other interfaces

RISC computing : heart of low power compute

- IBM 1970 → standardised by Berkeley, Stanford
  - Basic arch models
- 3 basic philosophies
  - Single cycle instructions
  - Memory access only for load and store
  - All execution units are “pure hardware”, no micro-codes

RISC features

- HW implementation of lesser no. of instr frees up die space which can be used for
- Large register file
- More on-chip cache
- Additional FUs for superscalar execution
- FP units

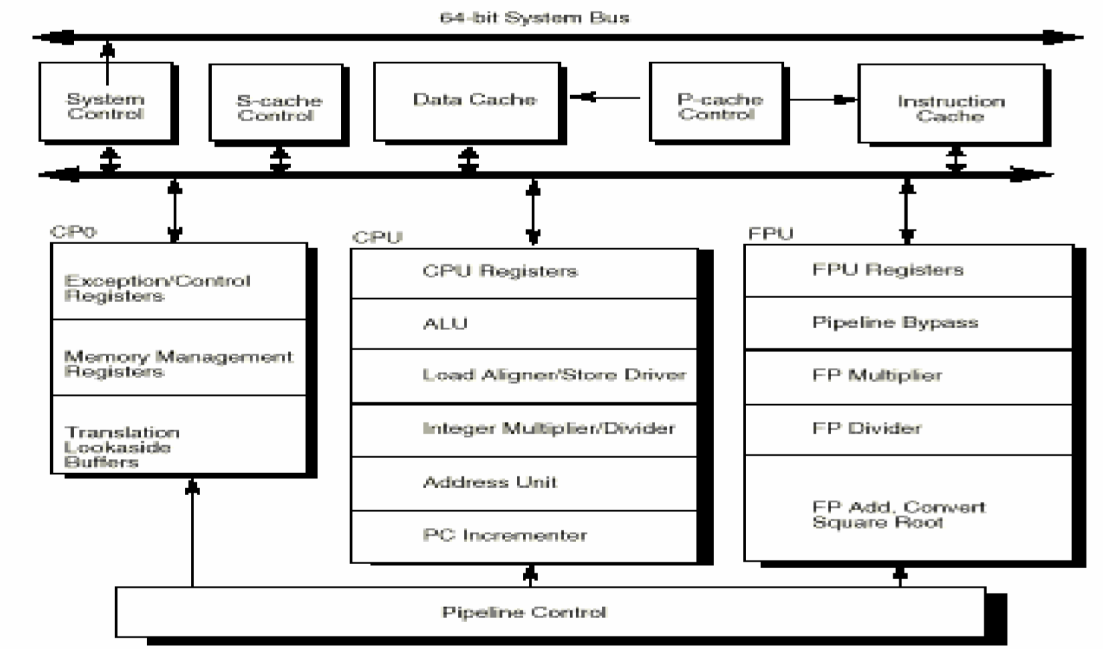
Classic RISC pipeline stages

- **Fetch** the instruction
- **Decode** the instruction semantics
- **Execute** ALU operations
- **Memory** load n store
- **Register** file update

RISC instructions

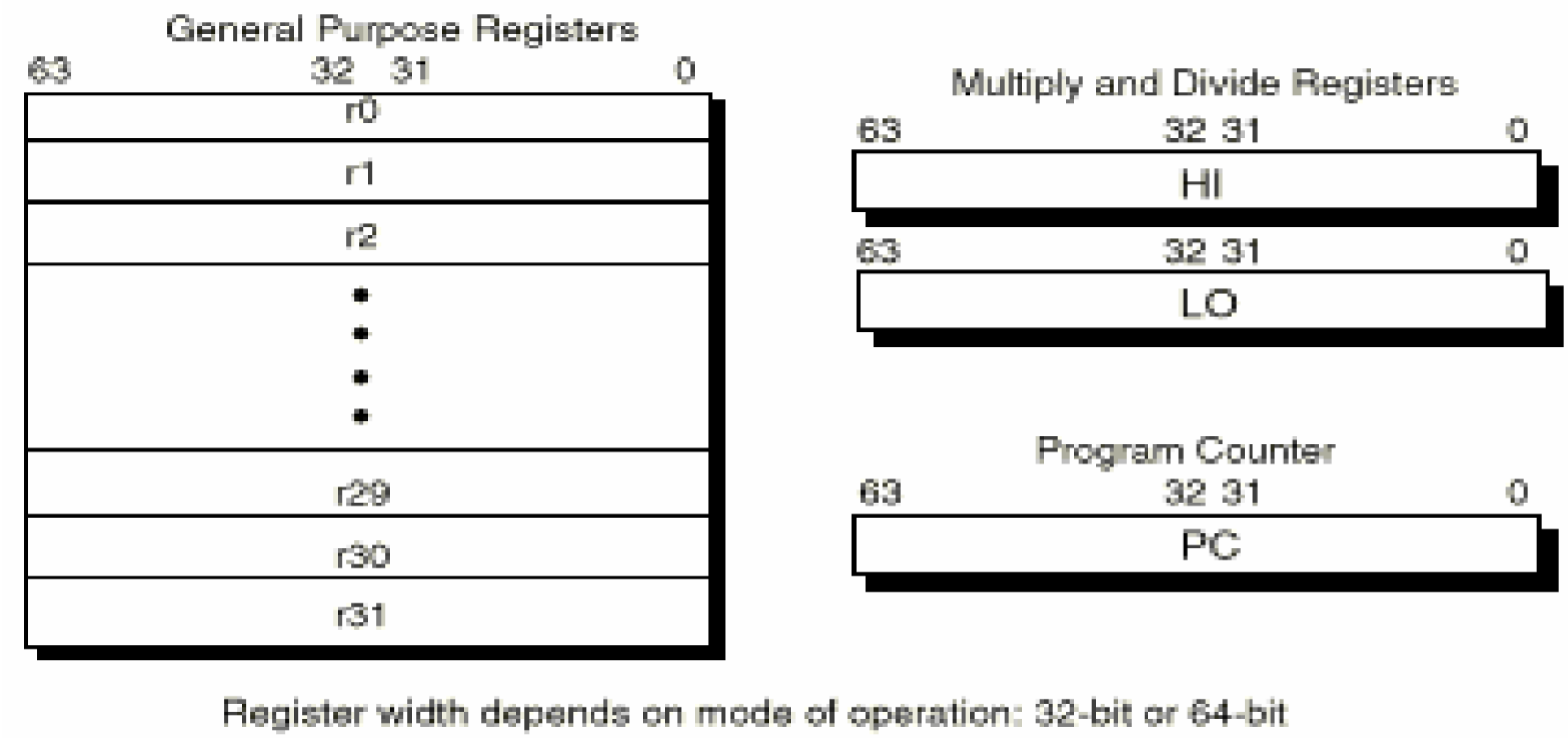
- Databus width (32/64 bits)
- Make all instrs 32/64 bits
- 32 bits = 4 bytes = 1 byte instr op code, 3 bytes for encoding 3 registers (operation inputs and output)
- add r0 r1 r2 ➔ r0 = r1 + r2
- All instrs ➔ 32 bit / 64 bit
- All instrs → execute in single machine cycle

R4000 Internal Block Diagram



Images and some other material are from ni2.com

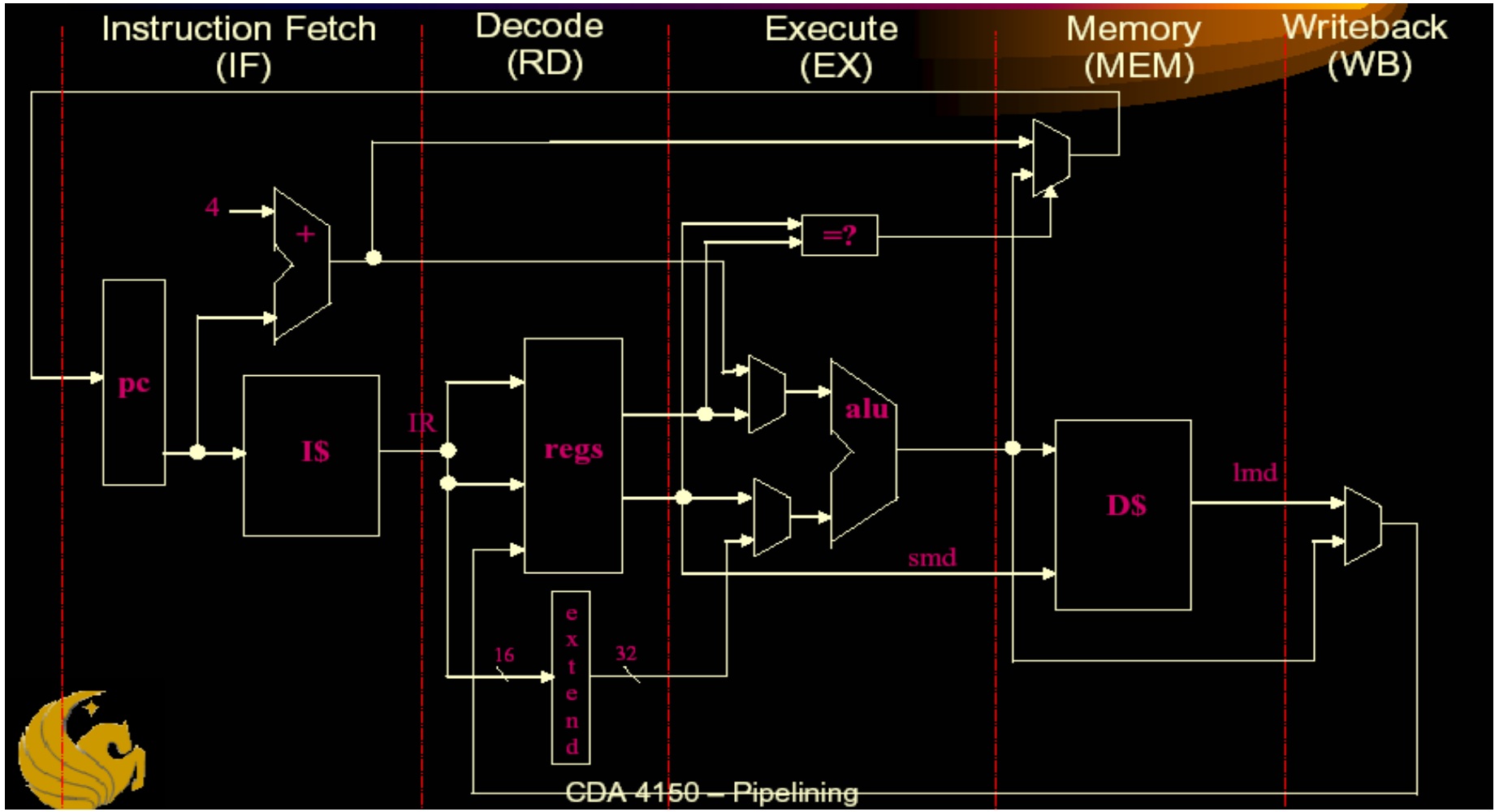
R4000 RISC register file



Instruction types

- Load/Store
- Arithmetic
- Logical
- Jump
- Special co-processor instructions

Pipeline Arch



source: http://cse.stanford.edu/class/sophomore-college/projects-00/risc/pipelining/

MMU

- Translates virtual addresses to Physical ones
- Some addresses are cached in “Translation lookaside buffer” (TLB)
- TLB is **fully associative** memory

Summary :

- RISC vs CISC
  - RISC has more no of GPRs
  - All RISC instructions have same encoding length
  - Simple hardware design with all single cycle instructions
- Cons : Code quality
- Applications : Data transmission, real time control, -→ widely accepted in CPS/IoT applications

Memory Hierarchy

DRAM vs. SRAM Memory Cell Complexity

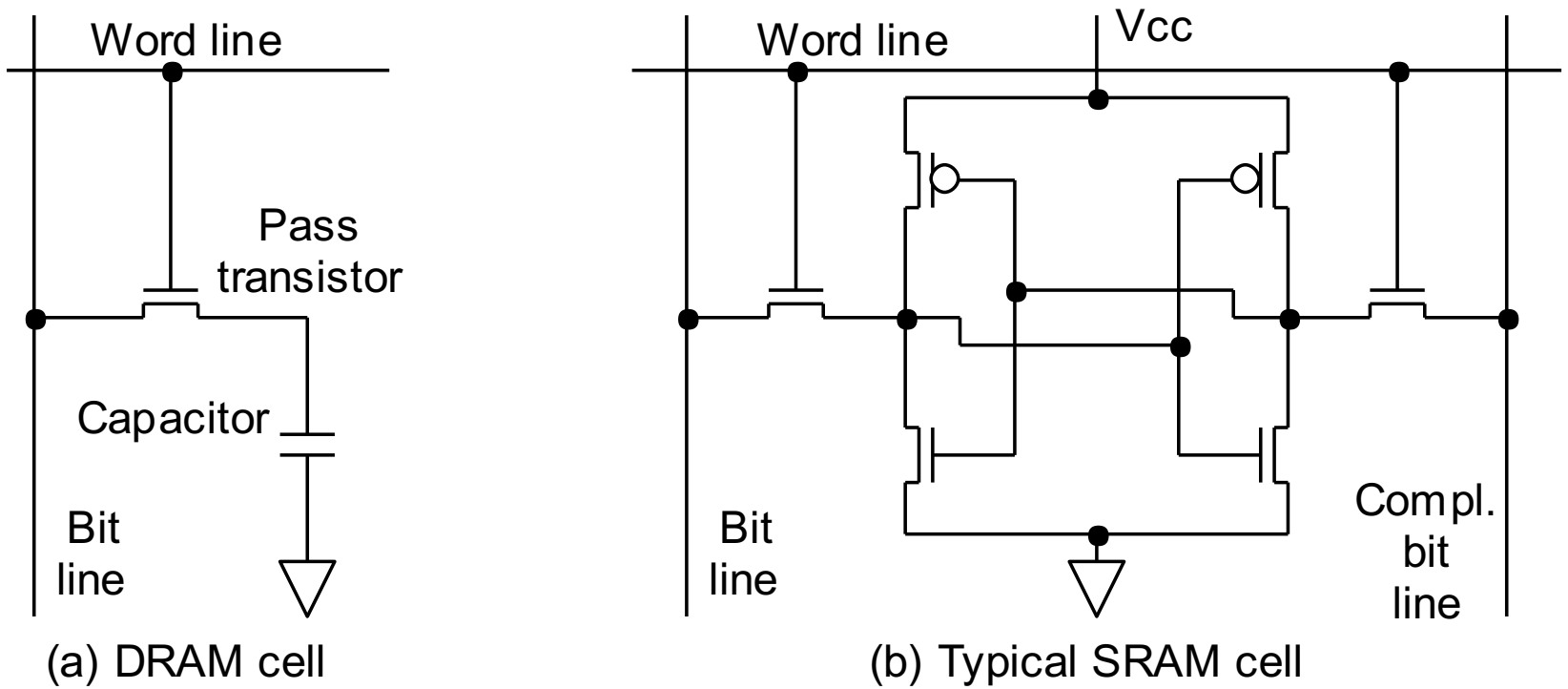


Fig. 17.4 Single-transistor DRAM cell, which is considerably simpler than SRAM cell, leads to dense, high-capacity DRAM memory chips.

Source of all this and following slides: Computer Architecture, Memory System Design by B. Parhami UCSB

DRAM Refresh Cycles and Refresh Rate

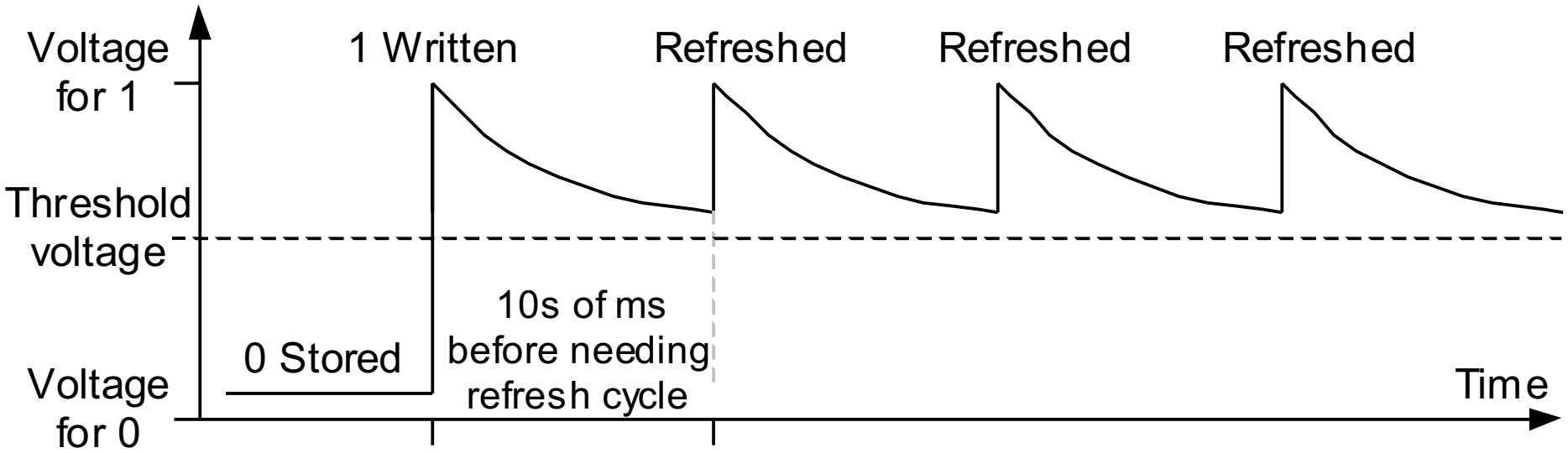
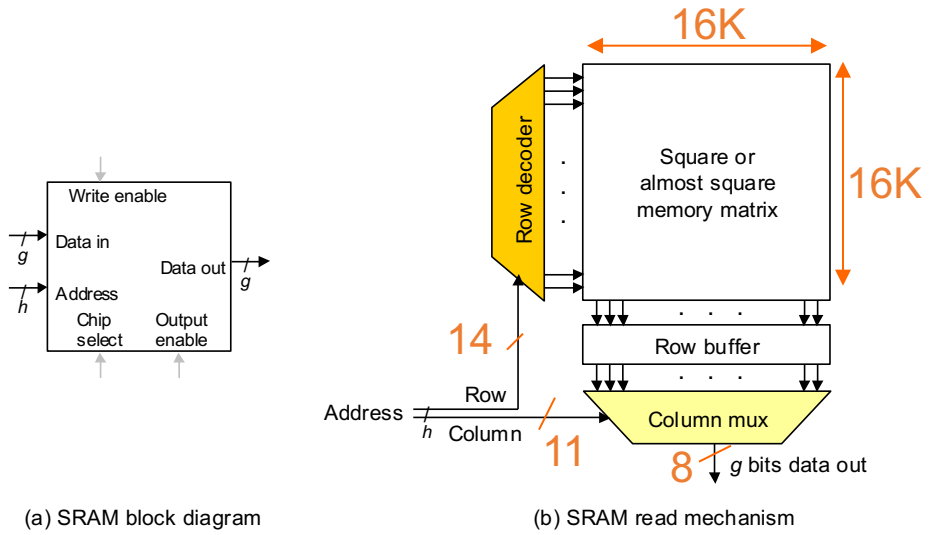


Fig. 17.5 Variations in the voltage across a DRAM cell capacitor after writing a 1 and subsequent refresh operations.

Loss of Bandwidth to Refresh Cycles

A 256 Mb DRAM chip is organized as a 32M × 8 memory externally and as a 16K × 16K array internally. Rows must be refreshed at least once every 50 ms to forestall data loss; refreshing a row takes 100 ns. What fraction of the total memory bandwidth is lost to refresh cycles?

Figure 2.10

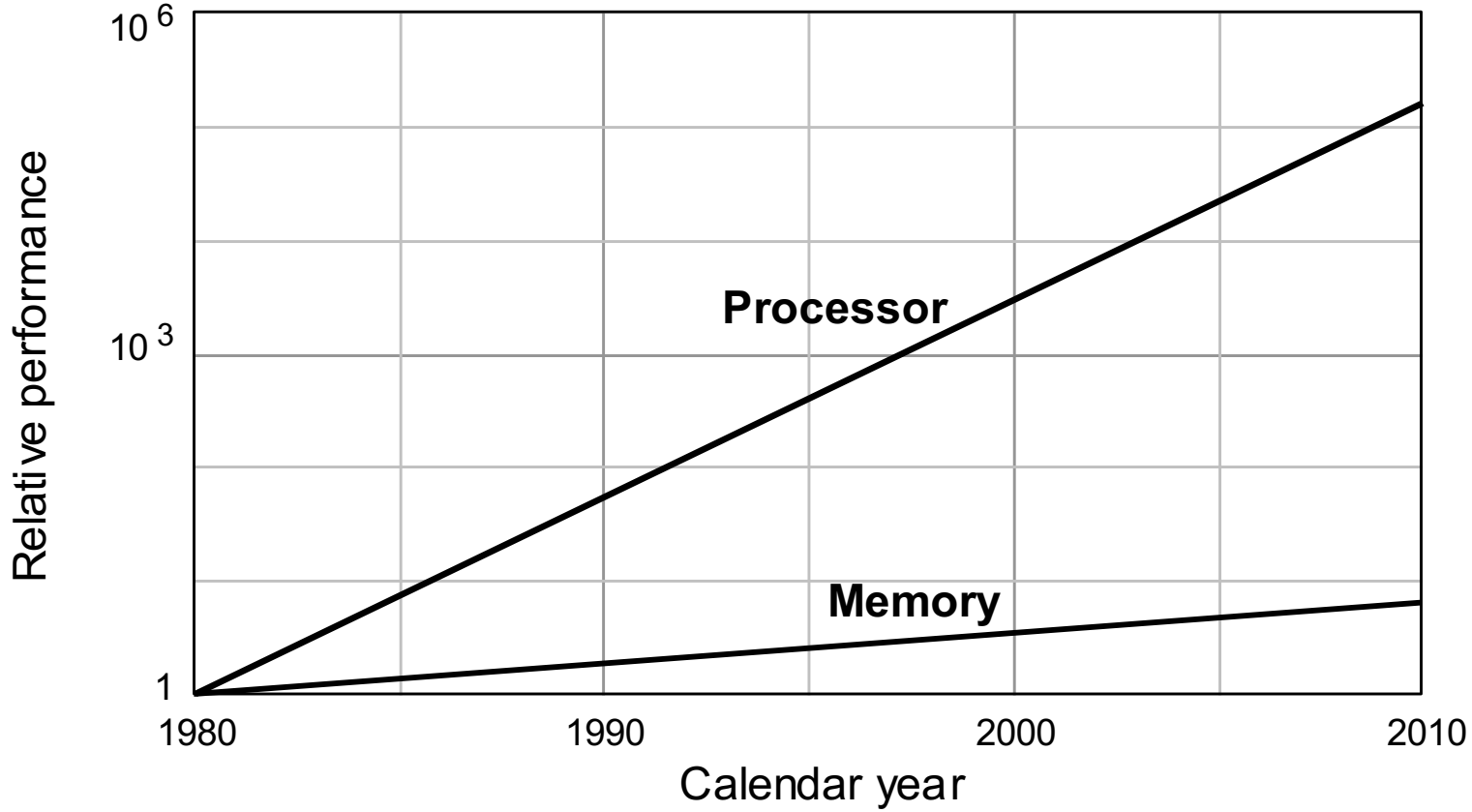


(a) SRAM block diagram (b) SRAM read mechanism

Solution

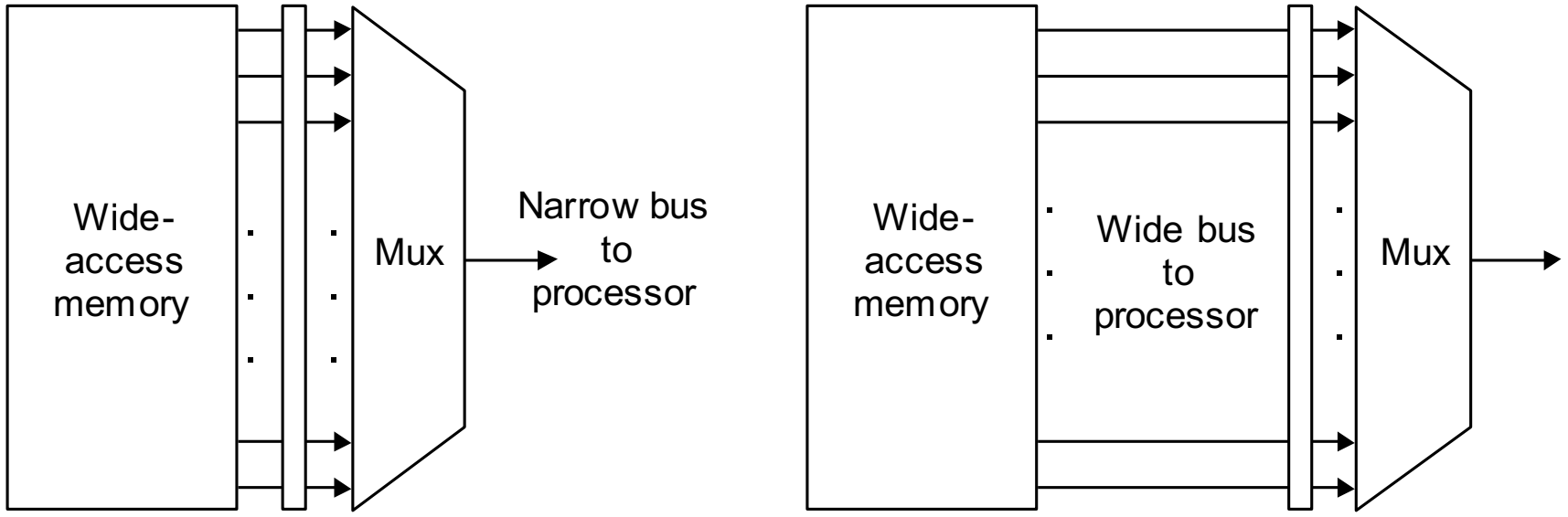
Refreshing all 16K rows takes 16 × 1024 × 100 ns = 1.64 ms. Loss of 1.64 ms every 50 ms amounts to 1.64/50 = 3.3% of the total bandwidth.

Hitting the memory wall



Bridging the CPU-Memory Speed Gap

Idea: Retrieve more data from memory with each access



(a) Buffer and multiplexer at the memory side

(a) Buffer and multiplexer at the processor side

Fig. 17.9 Two ways of using a wide-access memory to bridge the speed gap between the processor and memory.

Memory Interleaving

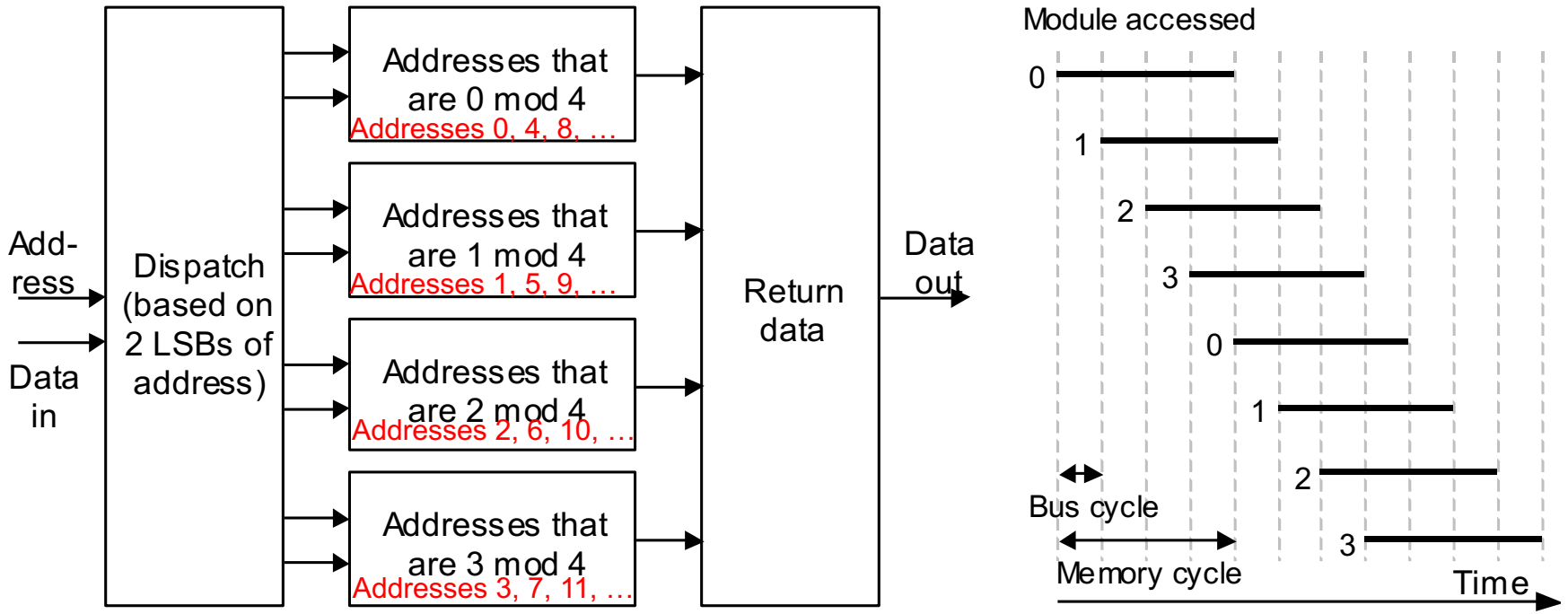


Fig. 17.11 Interleaved memory is more flexible than wide-access memory in that it can handle multiple independent accesses at once.

Soumyajit Dey, IIT Kharagpur

The Need for a Memory Hierarchy

The widening speed gap between CPU and main memory

Processor operations take of the order of 1 ns

Memory access requires 10s or even 100s of ns

Memory bandwidth limits the instruction execution rate

Each instruction executed involves at least one memory access

Hence, a few to 100s of MIPS is the best that can be achieved

A fast buffer memory can help bridge the CPU-memory gap

The fastest memories are expensive and thus not very large

A second (third?) intermediate cache level is thus often used

Typical Levels in a Hierarchical Memory

Capacity	Access latency	Cost per GB
100s B	ns	$Millions
10s KB	a few ns	$100s Ks
MBs	10s ns	$10s Ks
100s MB	100s ns	$1000s
10s GB	10s ms	$10s
TBs	min+	$1s

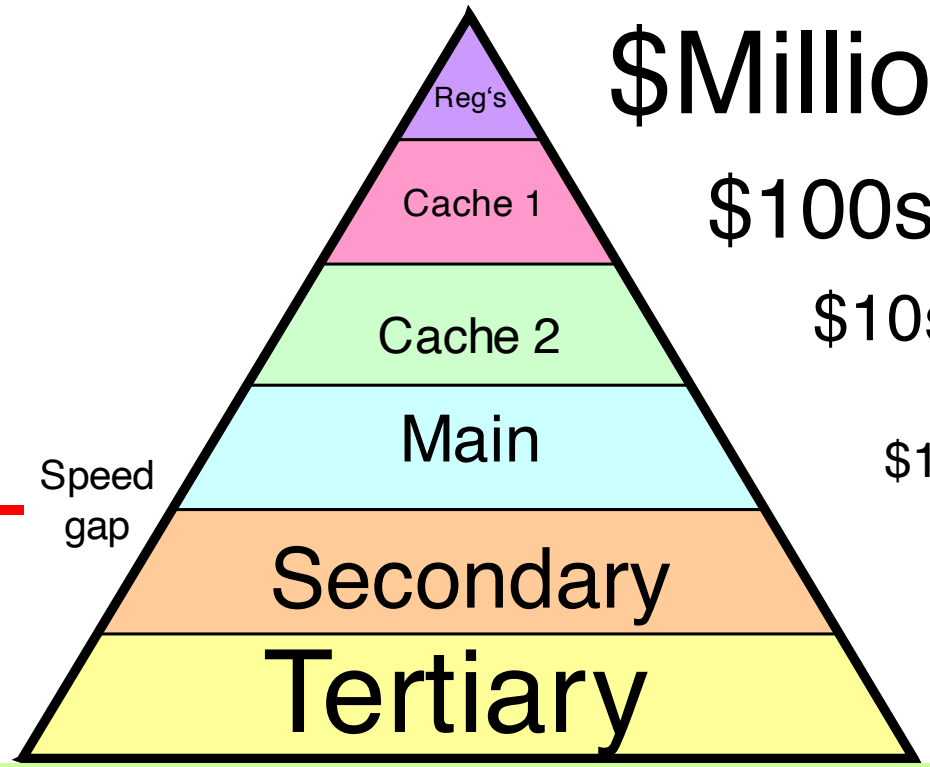
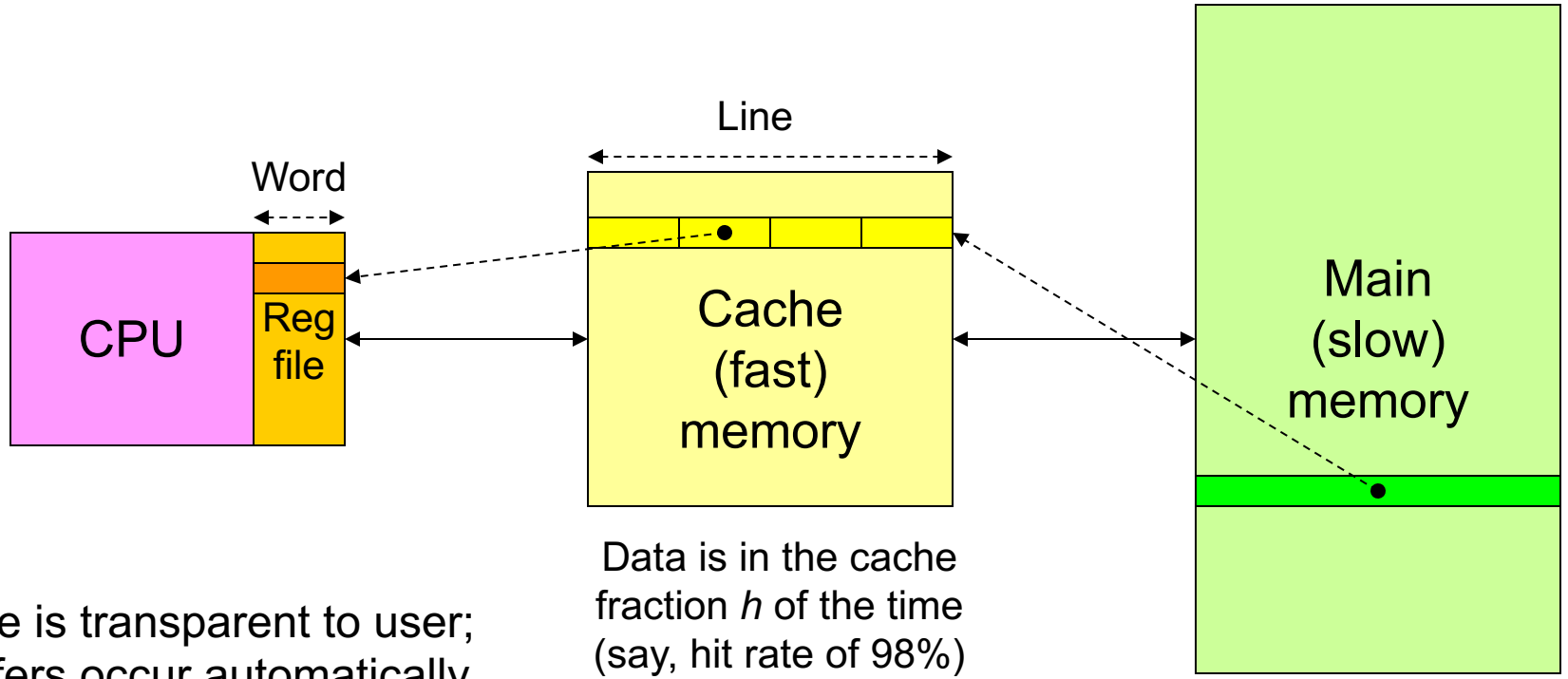


Fig. 17.14 Names and key characteristics of levels in a memory hierarchy.

Cache, Hit/Miss Rate, and Effective Access Time



Cache is transparent to user; transfers occur automatically

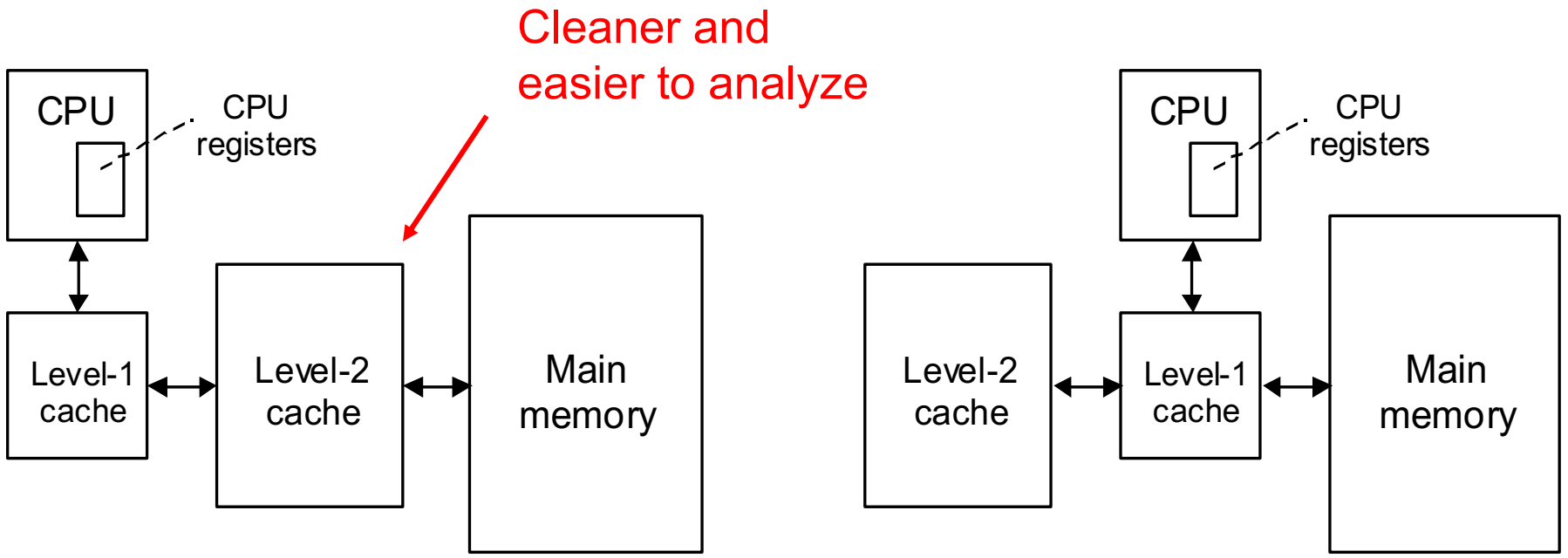
Data is in the cache fraction *h* of the time (say, hit rate of 98%)

Go to main 1 – *h* of the time (say, cache miss rate of 2%)

One level of cache with hit rate *h*

$$C_{eff} = hC_{fast} + (1 - h)(C_{slow} + C_{fast}) = C_{fast} + (1 - h)C_{slow}$$

Multiple Cache Levels



(a) Level 2 between level 1 and main

(b) Level 2 connected to “backside” bus

Fig. 18.1 Cache memories act as intermediaries between the superfast processor and the much slower main memory.

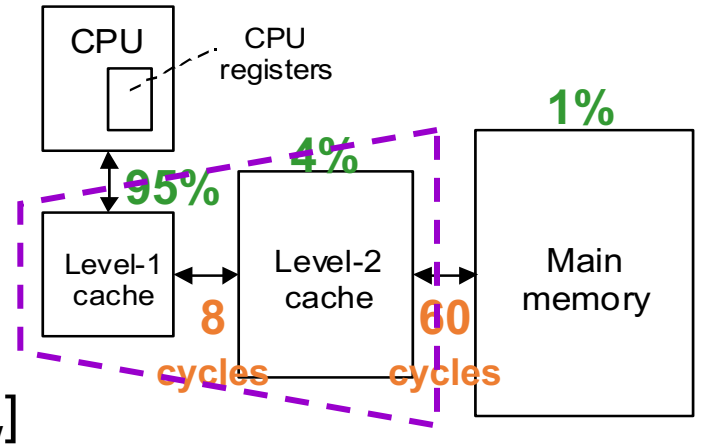
Performance of a Two-Level Cache System

A system with L1 and L2 caches has a CPI of 1.2 with no cache miss. There are 1.1 memory accesses on average per instruction.

What is the effective CPI with cache misses factored in?

What are the effective hit rate and miss penalty overall if L1 and L2 caches are modeled as a single cache?

Level	Local hit rate	Miss penalty
L1	95 %	8 cycles
L2	80 %	60 cycles



Solution

$C_{eff} = C_{fast} + (1 - h_1)[C_{medium} + (1 - h_2)C_{slow}]$

Because C_{fast} is included in the CPI of 1.2, we must account for the rest

CPI = 1.2 + 1.1(1 – 0.95)[8 + (1 – 0.8)60] = 1.2 + 1.1 × 0.05 × 20 = 2.3

Overall: hit rate 99% (95% + 80% of 5%), miss penalty 60 cycles

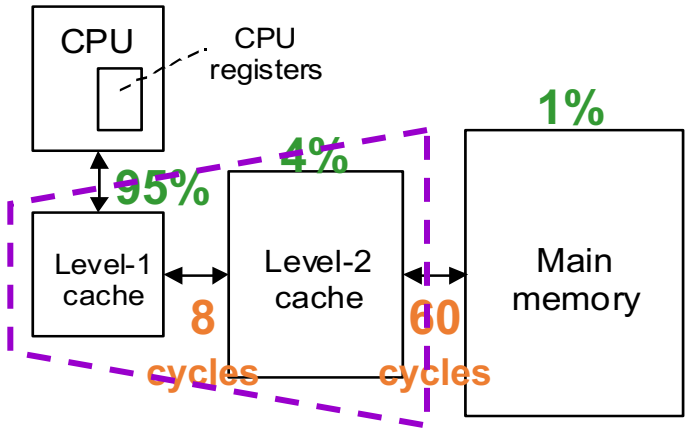
Performance of a Two-Level Cache System

A system with L1 and L2 caches has a CPI of 1.2 with no cache miss. There are 1.1 memory accesses on average per instruction.

What is the effective CPI with cache misses factored in?

What are the effective hit rate and miss penalty overall if L1 and L2 caches are modeled as a single cache?

Level	Local hit rate	Miss penalty
L1	95 %	8 cycles
L2	80 %	60 cycles



Solution

Consider 100 instructions, so there are 110 mem access.

Cycles w/o misses = 120.

L1 miss = 110 * 0.05 = 5.5 , cycles = 5.5 * 8 = 44

L2 miss = 5.5 * 0.2 = 1.1 , cycles = 1.1 * 60 = 66

Total cycles = 120+44+66 = 120+110 = 230

Hence, CPI = 2.3

Overall: hit rate 99% (95% + 80% of 5%), miss penalty 60 cycles

Cache Memory Design Parameters

Cache size (in bytes or words). A larger cache can hold more of the program's useful data but is more costly and likely to be slower.

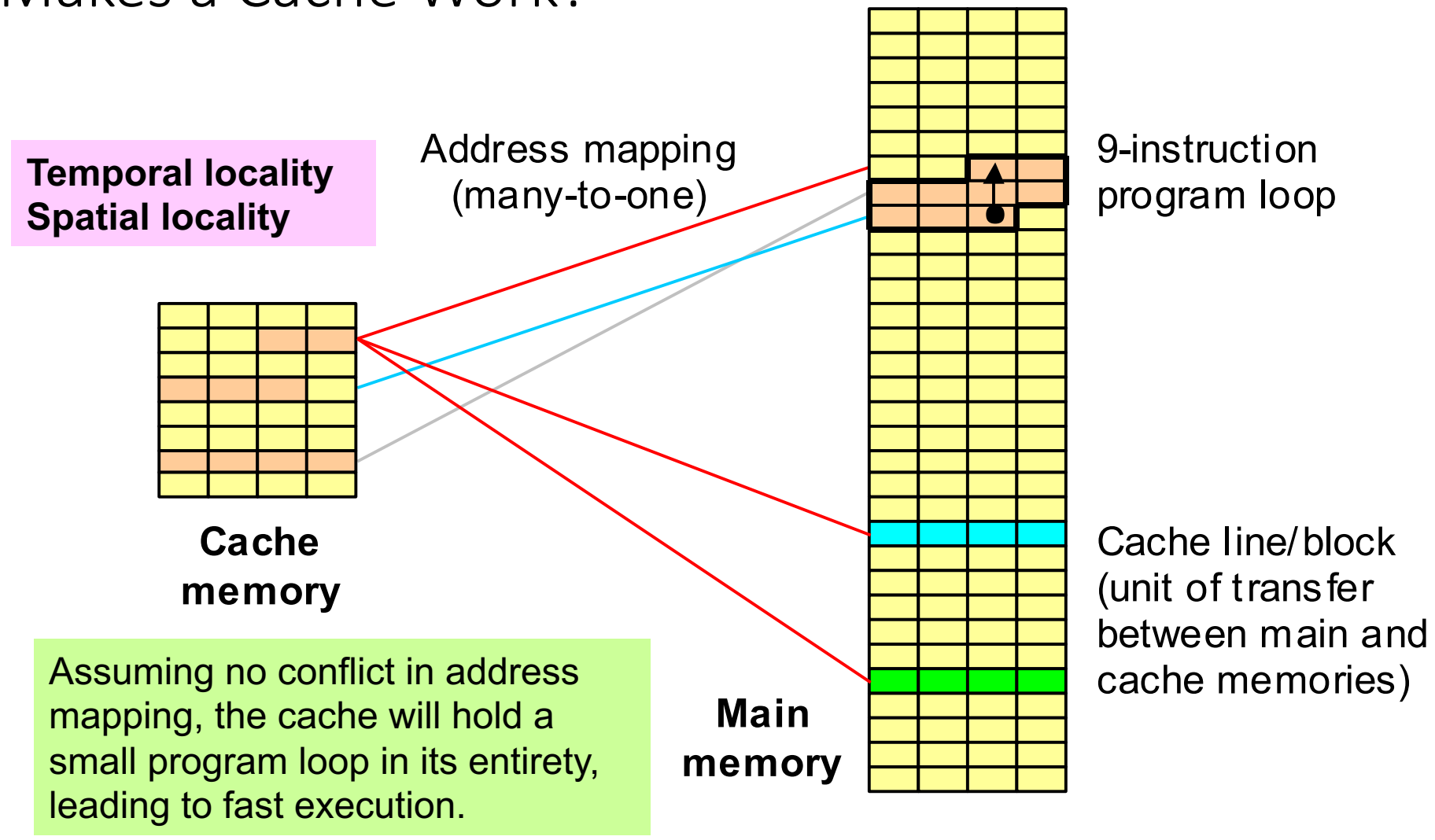
Block or *cache-line size* (unit of data transfer between cache and main). With a larger cache line, more data is brought in cache with each miss. This can improve the hit rate but also may bring low-utility data in.

Placement policy. Determining where an incoming cache line is stored. More flexible policies imply higher hardware cost and may or may not have performance benefits (due to more complex data location).

Replacement policy. Determining which of several existing cache blocks (into which a new cache line can be mapped) should be overwritten. Typical policies: choosing a random or the least recently used block.

Write policy. Determining if updates to cache words are immediately forwarded to main (*write-through*) or modified blocks are copied back to main if and when they must be replaced (*write-back* or *copy-back*).

What Makes a Cache Work?



Assuming no conflict in address mapping, the cache will hold a small program loop in its entirety, leading to fast execution.

Desktop, Drawer, and File Cabinet Analogy

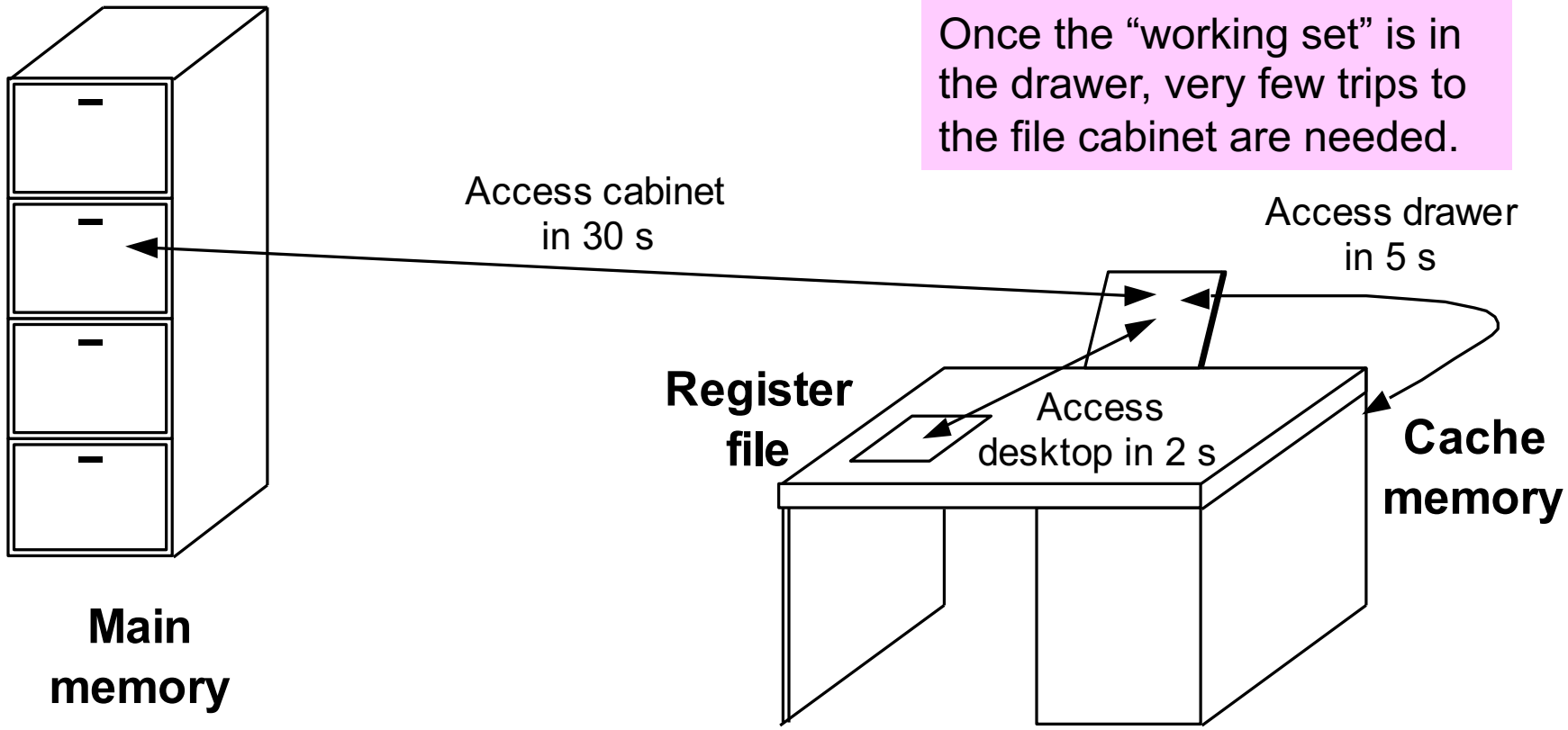
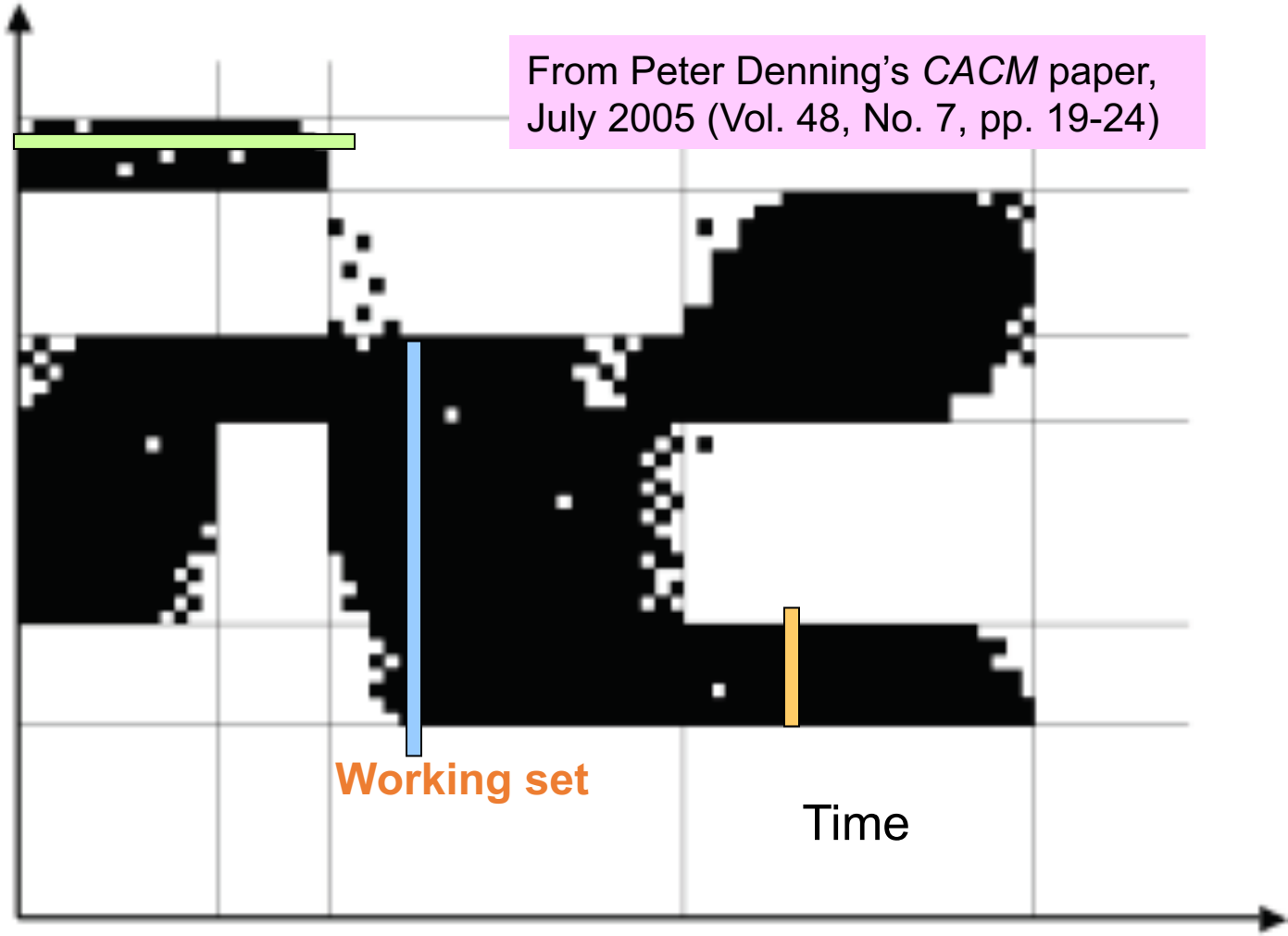


Fig. 18.3 Items on a desktop (register) or in a drawer (cache) are more readily accessible than those in a file cabinet (main memory).

Temporal and Spatial Localities

Temporal:
Accesses to the same address are typically clustered in time

Spatial:
When a location is accessed, nearby locations tend to be accessed also



Caching Benefits Related to Amdahl's Law

In the drawer & file cabinet analogy, assume a hit rate *h* in the drawer. Formulate the situation in terms of Amdahl's law.

Solution

Without the drawer, a document is accessed in 30 s. So, fetching 1000 documents, say, would take 30 000 s. The drawer causes a fraction *h* of the cases to be done 6 times as fast, with access time unchanged for the remaining 1 – *h*. Speedup is thus $1/(1 - h + h/6) = 6 / (6 - 5h)$. Improving the drawer access time can increase the speedup factor but as long as the miss rate remains at 1 – *h*, the speedup can never exceed $1 / (1 - h)$. Given $h = 0.9$, for instance, the speedup is 4, with the upper bound being 10 for an extremely short drawer access time.

Note: Some would place everything on their desktop, thinking that this yields even greater speedup. This strategy is not recommended!

Compulsory, Capacity, and Conflict Misses

Compulsory misses: With *on-demand fetching*, first access to any item is a miss. Some "compulsory" misses can be avoided by prefetching.

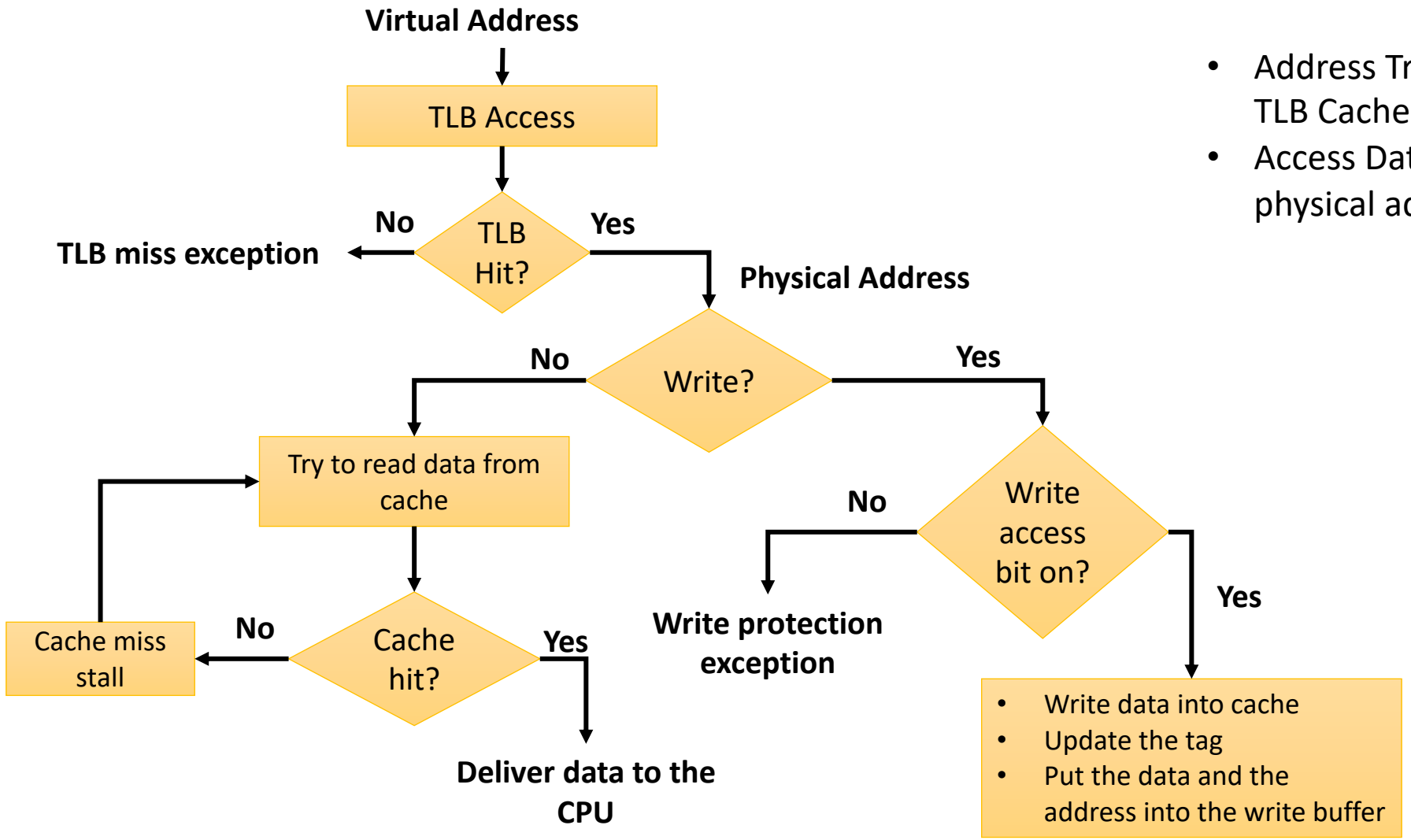
Capacity misses: We have to oust some items to make room for others. This leads to misses that are not incurred with an infinitely large cache.

Conflict misses: Occasionally, there is free room, or space occupied by useless data, but the mapping/placement scheme forces us to displace useful items to bring in other items. This may lead to misses in future.

Given a fixed-size cache, dictated, e.g., by cost factors or availability of space on the processor chip, compulsory and capacity misses are pretty much fixed. Conflict misses, on the other hand, are influenced by the data mapping scheme which is under our control.

We study two popular mapping schemes: direct and set-associative.

Operations during mem access



- Address Translation through TLB Cache
- Access Data Cache with physical address

Soumyajit Dey, IIT Kharagpur

Branch Prediction

- Prediction has become essential to getting good performance
- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems

Branch Prediction

- *Keep a buffer (cache) indexed by the lower portion of the address of the branch instruction.*
  - *Along with some bit(s) to indicate whether or not the branch was recently taken or not.*
- *If the prediction is incorrect , the prediction bit is inverted and stored back.*
- *The branch direction could be incorrect because:*
  - *Of misprediction OR*
  - Instruction mismatch
- *In either case, the worst that happens is that you have to pay the full latency for the branch.*

Branch Prediction : timing analysis

- Branch Prediction creates unpredictability in program execution time
- For a worst case estimate, we can
  - Consider branch penalties in all cases, but that is very pessimistic
  - Can use static program analysis to predict execution time estimates while inferring some decisions about branch behaviors

Set-associative cache Cache performance

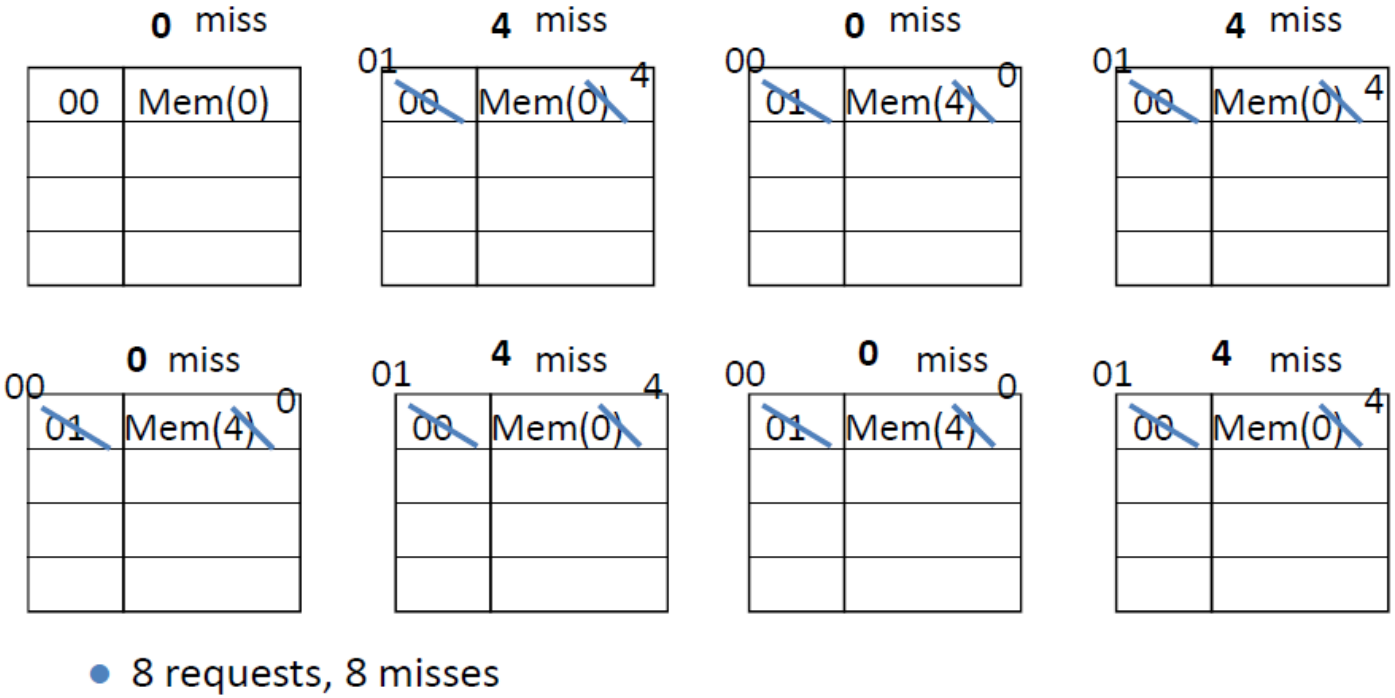
Flexible block replacement reduces cache miss rate

1. In ***Direct mapped*** cache a memory block maps to exactly one cache block
2. ***Fully associative cache*** allows a memory block to be mapped to any cache block
3. A compromise is to divide the cache into sets each of which consists of n "ways" (***n-way set associative***). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)

   (block address) modulo (# sets in the cache)

Limitation of Direct Mapping

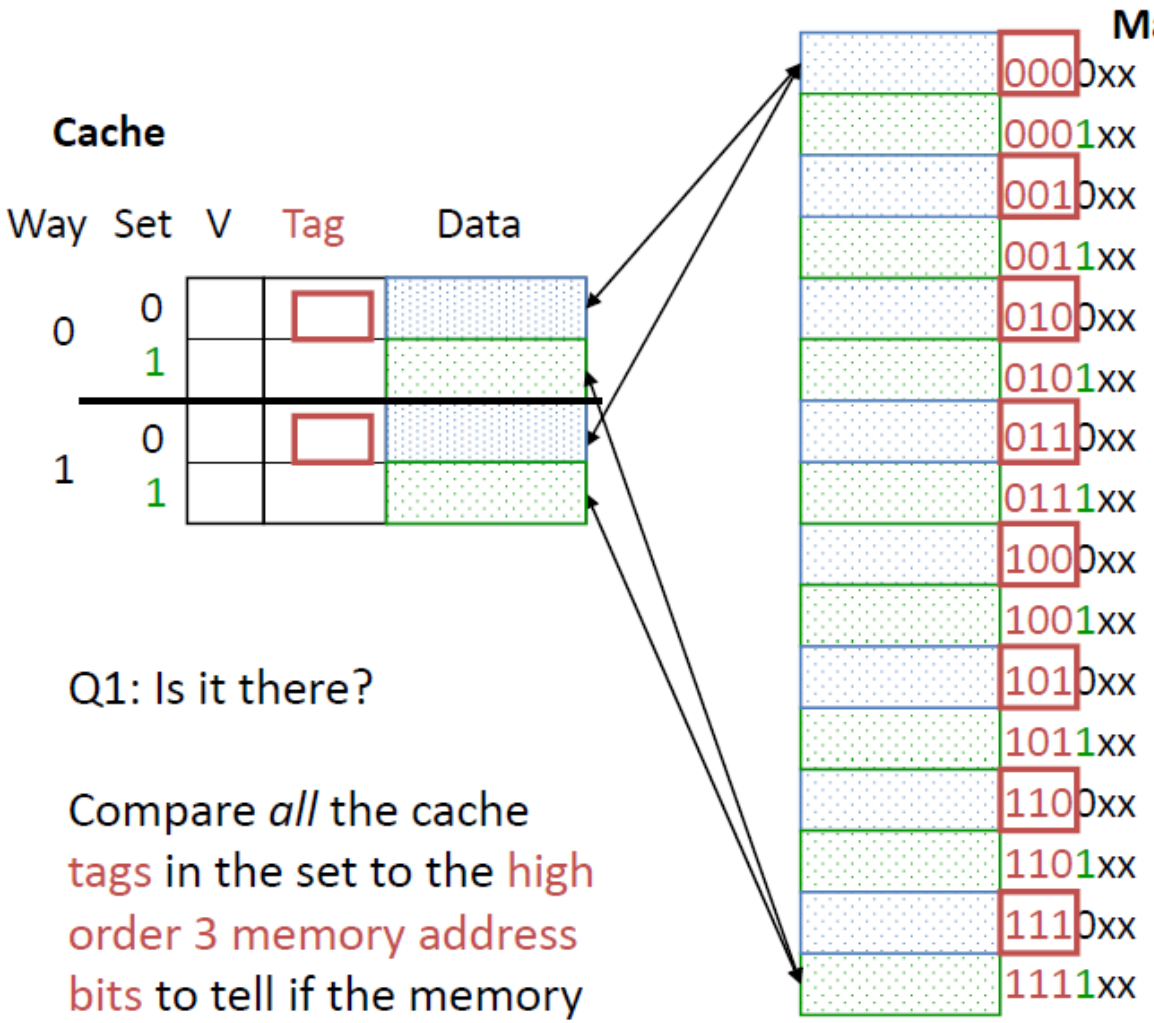
- Consider main memory reference string: 04040404
- Start with an empty cache with all blocks marked as not valid



- 8 requests, 8 misses

- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

Set Associative Cache Example



One word blocks
Two low order bits define the byte in the word (32b words)

Q1: Is it there?

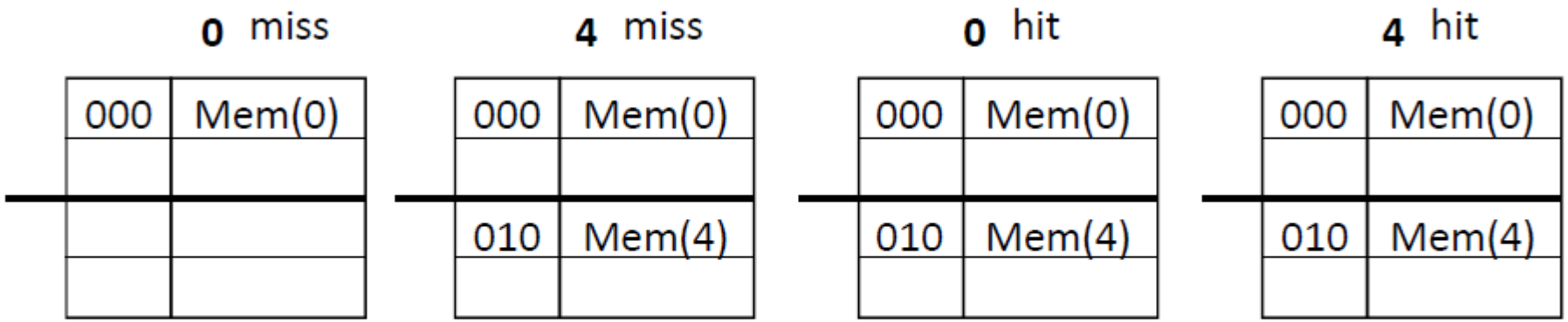
Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Set Associative Cache Example

- Consider main memory reference string: 04040404
- Start with an empty cache with all blocks marked as not valid



  - 8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

Program optimizations to reduce cache miss

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts(using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

Source : CS252/Culler

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key; improve spatial locality

Source : CS252/Culler

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Source : CS252/Culler

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
       a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
       d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {    a[i][j] = 1/b[i][j] * c[i][j];
       d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to `a` & `c` vs. one miss per access; improve spatial locality

Source : CS252/Culler

Blocking Example

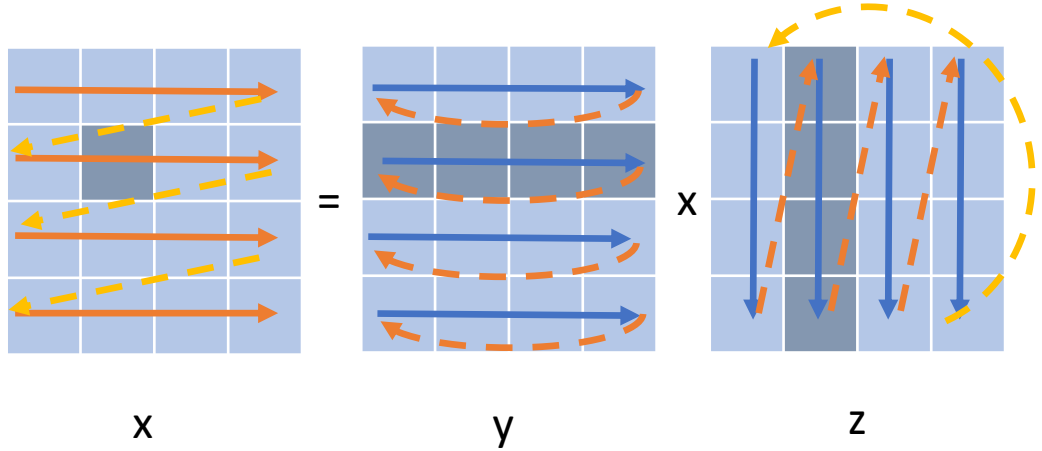
- Since we intend to calculate the worst case cache miss count for NxN matrix multiplication, we assume:
  1. Cache Size << Row Size i.e. N.
  2. Each cache block < Element size in Matrix (8 or 16Bytes for Double).
  3. Only Compulsory and Capacity misses are considered.
- For y matrix:

  The **innermost loop with variable k, needs N memory accesses** to traverse through each row of y to generate single element in x. There are **N^2 elements to fill in x**. So total memory access/cache miss = **N^3**

  Because we consider each cache block can't hold even single element and the cache doesn't hold the whole row, so by the time we need to access the first element (y[0][k] for x[i][j+1]) the last element is in the cache.

→ k → j → i

```
/* without blocking*/
for (i = 0; i < N; i = i+1){
      for (j = 0; j < N; j = j+1){
          r = 0;
          for (k = 0; k < N; k = k+1){
                r = r + y[i][k]*z[k][j];
          }x[i][j] = r;
      };
```



Blocking Example

- For z matrix:

  The **innermost loop with variable k, needs N memory accesses** to traverse through each column of z to generate single element in x. There are **N^2 elements to fill in x**. So total memory access/cache miss = **N^3**.

  Because we input elements in x in row-major format, so by the time we need to access the first element (z[k][0] for x[i+1][j]) it is not in the cache.

- For x matrix:

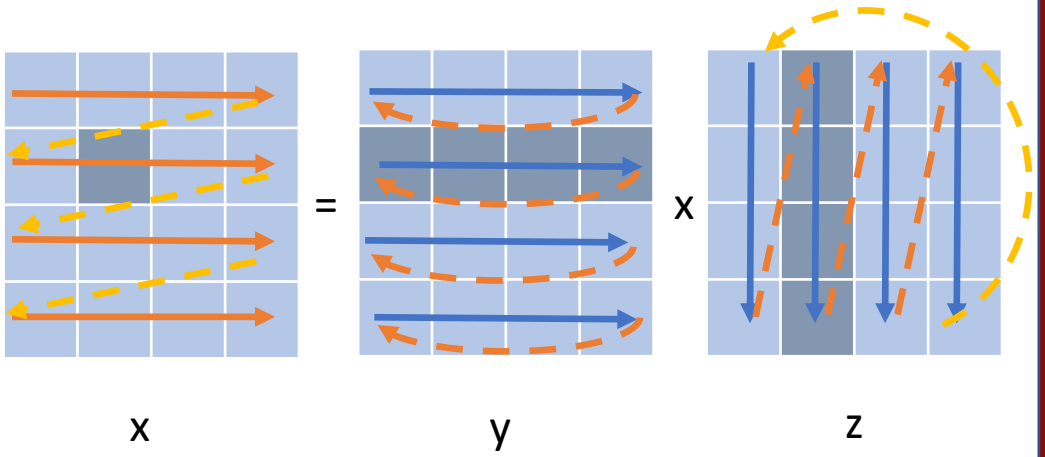
  There are **N^2 elements to fill in x**. So total memory access/cache miss = **N^2**

  Because we consider each cache block can't hold even single element and the cache doesn't hold the whole row, so by the time we need to write back the first element its address is not in the cache.

- Total Cache Misses : **$2N^3+N^2$**

→ k → j → i

```
/* without blocking*/
for (i = 0; i < N; i = i+1){
      for (j = 0; j < N; j = j+1){
          r = 0;
          for (k = 0; k < N; k = k+1){
                  r = r + y[i][k]*z[k][j];
          }x[i][j] = r;
      };
```



Blocking Example

- Loop interchange is not going to help because we access y in row-major fashion but z in column-major fashion, hence we apply **blocking**.

- We divide every NxN matrix into multiple BxB submatrices while multiplying.

- Blocking factor B must be chosen s.t. B^2number of elements are accommodated in the cache.

Blocking Example

- For y matrix:
  - The innermost loop with variable k accesses B elements among the N elements in a row in y matrix and finishes all the calculations concerning these B elements, see the loop with variable j.

  So there are only B compulsory misses.
  - This happens for all the outer loops with var i (size N) and vars kk, jj (N^2/B^2).

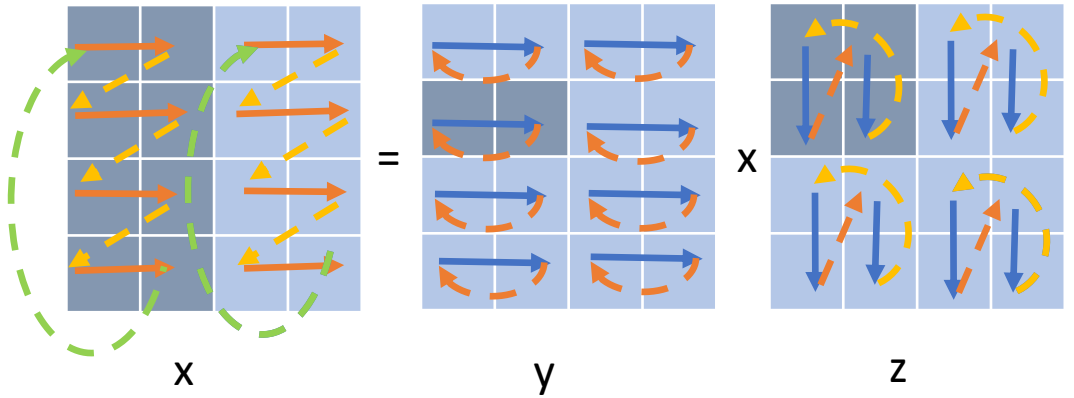
  So total number of cache misses= $N^2/B^2 x N x B = \mathbf{N^3/B}$
- For z matrix:
  - The inner loops with vars j and k access a BxB submatrix of z matrix everytime and since cache can hold B2 elements.

  There are only B^2 compulsory misses for N iterations in loop with variable i.
  - This happens for all the outer loops with variables kk, jj (N^2/B^2).

  So total number of cache misses= $N^2/B^2 x B^2 = \mathbf{N^2}$

k j i kk

```
/* After Blocking*/
for(jj = 0; jj < N; jj = jj+B){
for(kk = 0; kk < N; kk = kk+B){
for(i = 0; i < N; i = i+1){
      for(j = jj; j < min(jj+B-1,N); j = j+1){
          r = 0;
          for(k = kk; k < min(kk+B-1,N);k = k+1){
                r = r + y[i][k]*z[k][j];
          }x[i][j] = x[i][j] + r;
}}}}
```



Source : CS252/Culler

Blocking Example

- For x matrix:
  - The innermost loop (with var k) doesn't access x. So we start with the loops with variables j and I which concerns a NxB submatrix. Since the first element is again written back after these NxB size submatrix is written back,

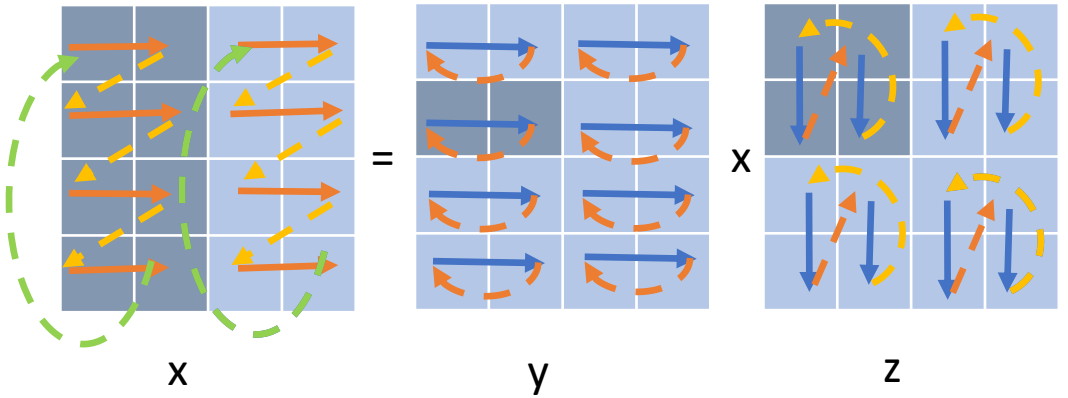
  there will be NB capacity misses (NxB > Cache Size).
  - This is done for all the outer loops with var i (size N) and vars kk, jj (N^2/B^2).

  So total number of cache misses= N^2/B^2xNxB= ***N^3/B***

- So total number of Cache misses after applying blocking = $\mathbf{N^3/B+N^3/B+N^2=2N^3/B+N^2}$

→ k → j → i → kk

```
/* After Blocking*/
for(jj = 0; jj < N; jj = jj+B){
for(kk = 0; kk < N; kk = kk+B){
for(i = 0; i < N; i = i+1){
        for(j = jj; j < min(jj+B-1,N); j = j+1){
            r = 0;
            for(k = kk; k < min(kk+B-1,N);k = k+1){
                    r = r + y[i][k]*z[k][j];
            }x[i][j] = x[i][j] + r;
}}}}
```



Source : CS252/Culler

Blocking Example

- Considering $\mathbf{N^3}$ being the dominating term in both the cases (*without blocking :* $\boldsymbol{2N^3+N^2}$ *, with blocking:* $\boldsymbol{2N^3/B+N^2}$) the cache miss is reduced by almost a factor of B!

- Considering a 32kB data cache (eg. intel i7 L1D cache), it will support 32k/8= 4k doubles i.e. a 64x64 matrix i.e. **B=64, which is a significant reduction factor in terms of cache misses!**

Source : CS252/Culler