

Class Pointers

Code for joining Microsoft Teams
for the class:

Form to be filled

Syllabus

Architecture: computer organization, single-core optimizations including exploiting cache hierarchy and vectorization, parallel architectures including multi-core, shared memory, distributed memory and GPU architectures

Algorithms and Data Structures: algorithmic analysis, overview of trees and graphs, algorithmic strategies, concurrent data structures

Parallelization Principles: motivation, challenges, metrics, parallelization steps, data distribution, PRAM model

Parallel Programming Models and Languages: OpenMP, MPI, CUDA;

Distributed Computing: Commodity cluster and cloud computing;
Distributed Programming: MapReduce/Hadoop model.

Syllabus

Architecture: computer organization, single-core optimizations including exploiting cache hierarchy and vectorization, parallel architectures including multi-core, shared memory, distributed memory and GPU architectures

Algorithms and Data Structures: algorithmic analysis, overview of trees and graphs, algorithmic strategies, concurrent data structures

Parallelization Principles: motivation, challenges, metrics, parallelization steps, data distribution, PRAM model

Parallel Programming Models and Languages: OpenMP, MPI, CUDA;

Distributed Computing: Commodity cluster and cloud computing;
Distributed Programming: MapReduce/Hadoop model.

Reference

Bryant, O'Hallaron. Computer Systems – A Programmer's Perspective

Culler, Singh. Parallel Computing Architecture. A Hardware/Software Approach

Quinn. Parallel Computing. Theory and Practice

Sahni. Data Structures, Algorithms, and Applications in C++

Grama, Gupta, Karypis, Kumar. Introduction to Parallel Computing

Pacheco. An Introduction to Parallel Programming

Hwang, Dongarra, Fox. Distributed and Cloud Computing: From Parallel Processing to the Internet of Things

Lin, Dyer. Data-Intensive Text Processing with MapReduce

Reference

Bryant, O'Hallaron. Computer Systems – A Programmer's Perspective, Pearson Education Limited 2016, 3rd Global Edition

Culler, Singh. Parallel Computing Architecture. A Hardware/Software Approach

Quinn. Parallel Computing. Theory and Practice

Sahni. Data Structures, Algorithms, and Applications in C++

Grama, Gupta, Karypis, Kumar. Introduction to Parallel Computing

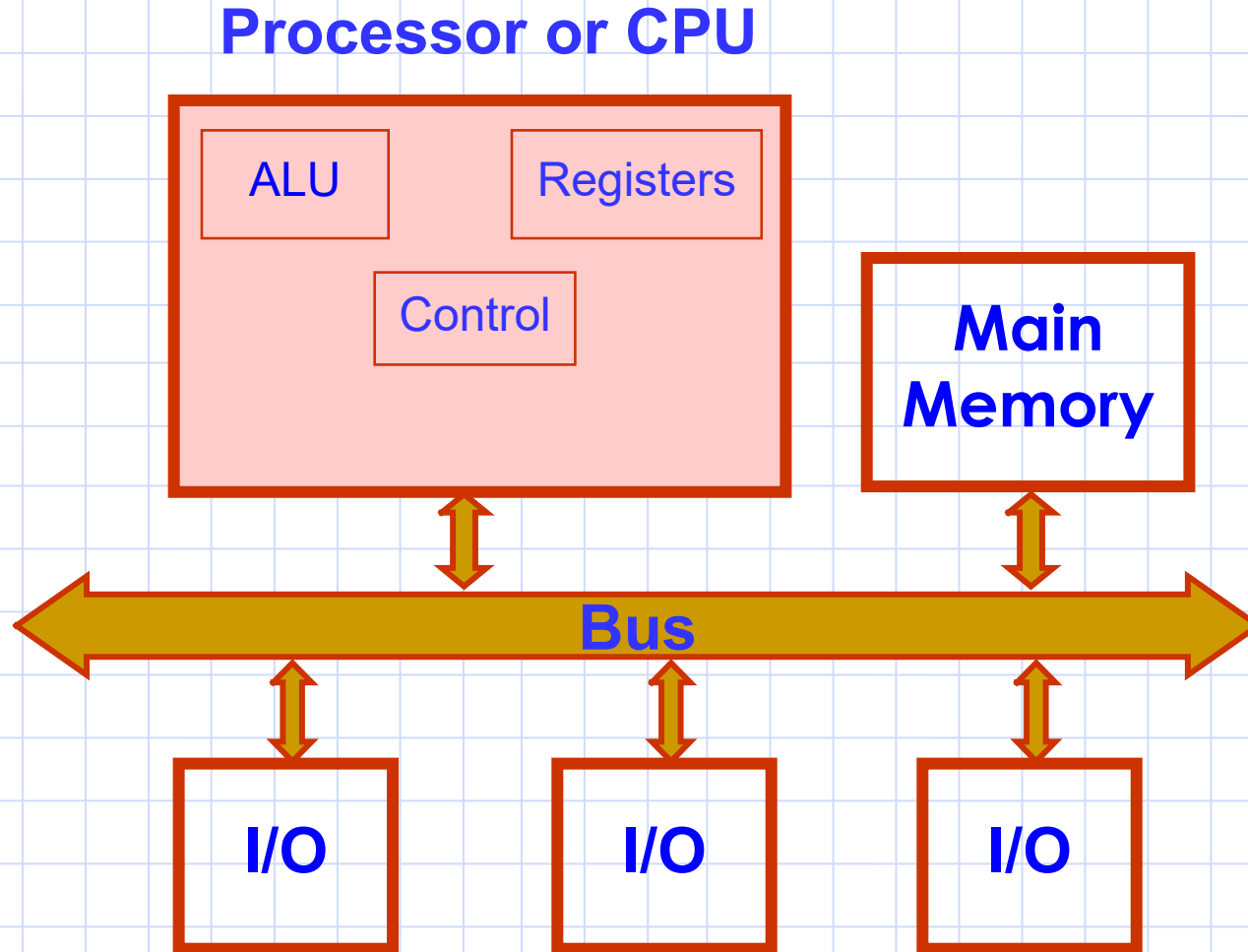
Pacheco. An Introduction to Parallel Programming

Hwang, Dongarra, Fox. Distributed and Cloud Computing: From Parallel Processing to the Internet of Things

Lin, Dyer. Data-Intensive Text Processing with MapReduce

Computer Organization: Memory Hierarchy and Cache Memories

Basic Computer Organization

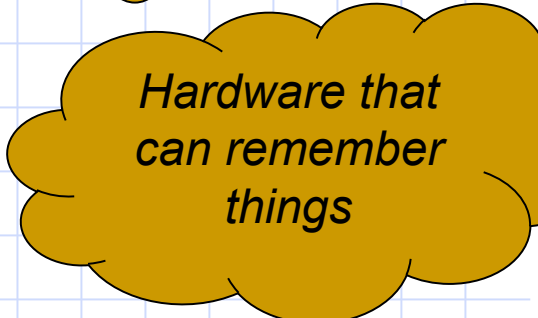


Inside the Processor...

- **Control hardware:** Hardware to manage instruction execution
- **ALU:** Arithmetic and Logical Unit (hardware to do arithmetic, logical operations)

Inside the Processor...

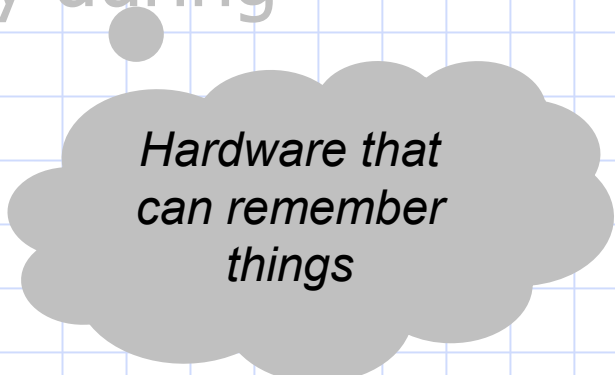
- **Control hardware:** Hardware to manage instruction execution
- **ALU:** Arithmetic and Logical Unit (hardware to do arithmetic, logical operations)
- **Registers:** small units of memory to hold data/instructions temporarily during execution



*Hardware that
can remember
things*

Inside the Processor...

- Control hardware: Hardware to manage instruction execution
- ALU: Arithmetic and Logical Unit (hardware to do arithmetic, logical operations)
- Registers: small units of memory to hold data/instructions temporarily during execution
- Two kinds of registers
 1. Special purpose registers
 2. General purpose registers



*Hardware that
can remember
things*

General Purpose Registers

- Available for use by programmer, possibly for keeping frequently used data
- Why? Since there is a large speed disparity between processor and main memory
 - 2 GHz Processor: 0.5 nanosecond time scale
 - Main memory: \sim 50-100 nsec time scale
- Machine instruction operands can come from registers or from main memory
- But CPUs do not provide a large number of general purpose registers

Problem: Slow Speed of Main Memory

- Main Memory is much slower (around 100x) than the CPU and only a few CPU registers
 - CPU will be waiting for data most of the time
- Solution: Cache Memory
 - Fast memory that is part of CPU
 - Design principle: Locality of Reference
 - Temporal locality: least recently accessed memory locations are least likely to be referenced in the near future
 - Spatial locality: neighbours of recently accessed memory locations are most likely to be referenced in the near future

Memory Address and Cache Composition

- When a CPU refers to a data or instruction, it gives a memory address where the data/instruction is present
- The memory address is used to check if the contents are in the cache.
- But how?

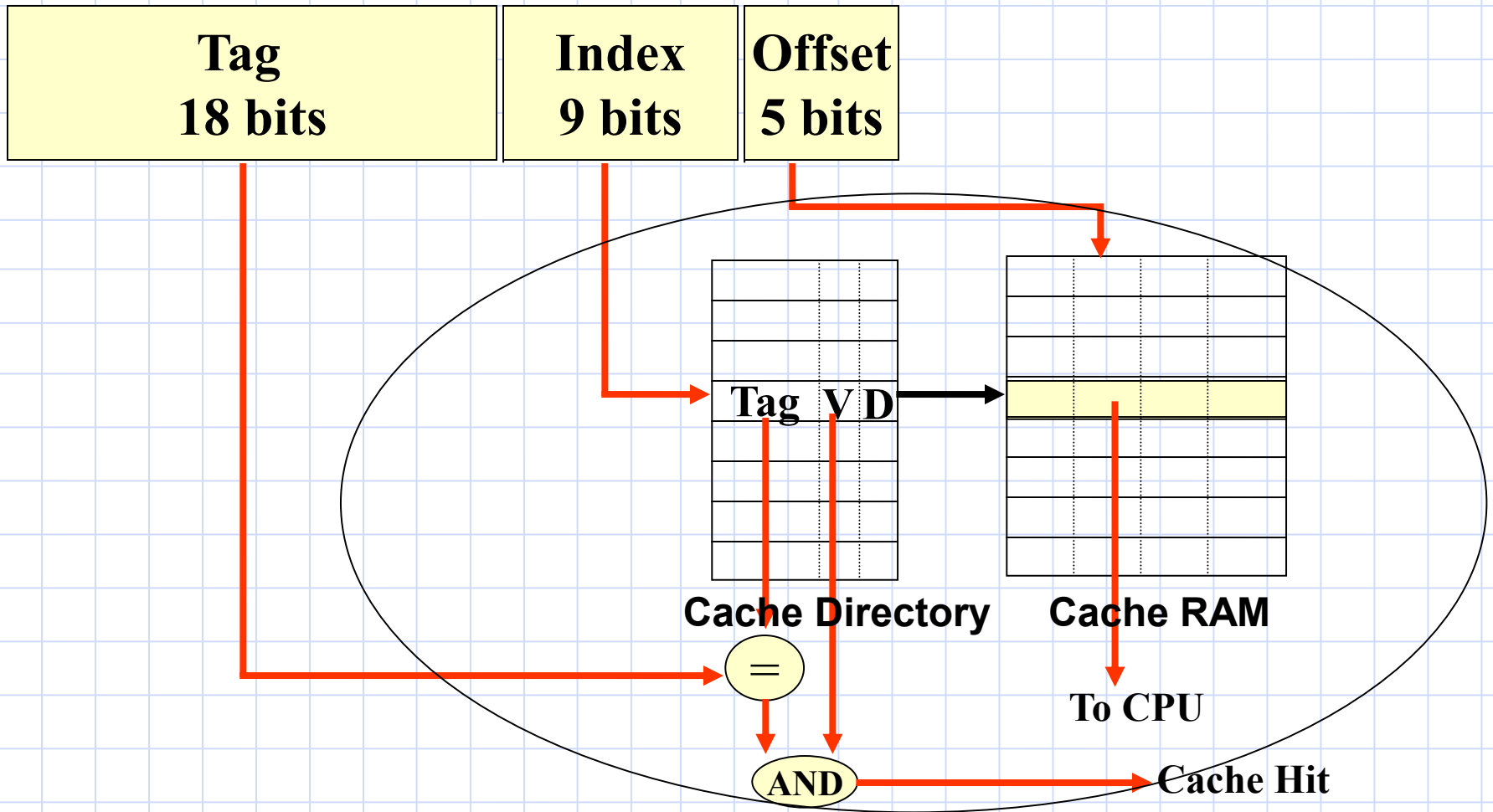
Memory Address and Cache Composition

- Memory is divided into multiple **blocks** of addresses or words
- A cache is decomposed into multiple **sets**
- Each set consists of multiple **cache lines**
- Each cache line consists of
 - A **valid** bit
 - A **tag** (set of bits)
 - A data **block of B bytes**

Cache Lookup

- Memory address divided into tag, index, offset
- Index – to identify the set number
- Then, all the cache lines in the particular set are searched
- The tag in the cache line checked with the tag part of the memory address
- If valid bit is set and the tag matches, then *cache hit*, else *cache miss*
- Offset used to fetch a particular word from the data block in the cache line
- Depending on the number of sets and lines, caches can be of different kinds

Cache Lookup and Access



Direct-Mapped Caches

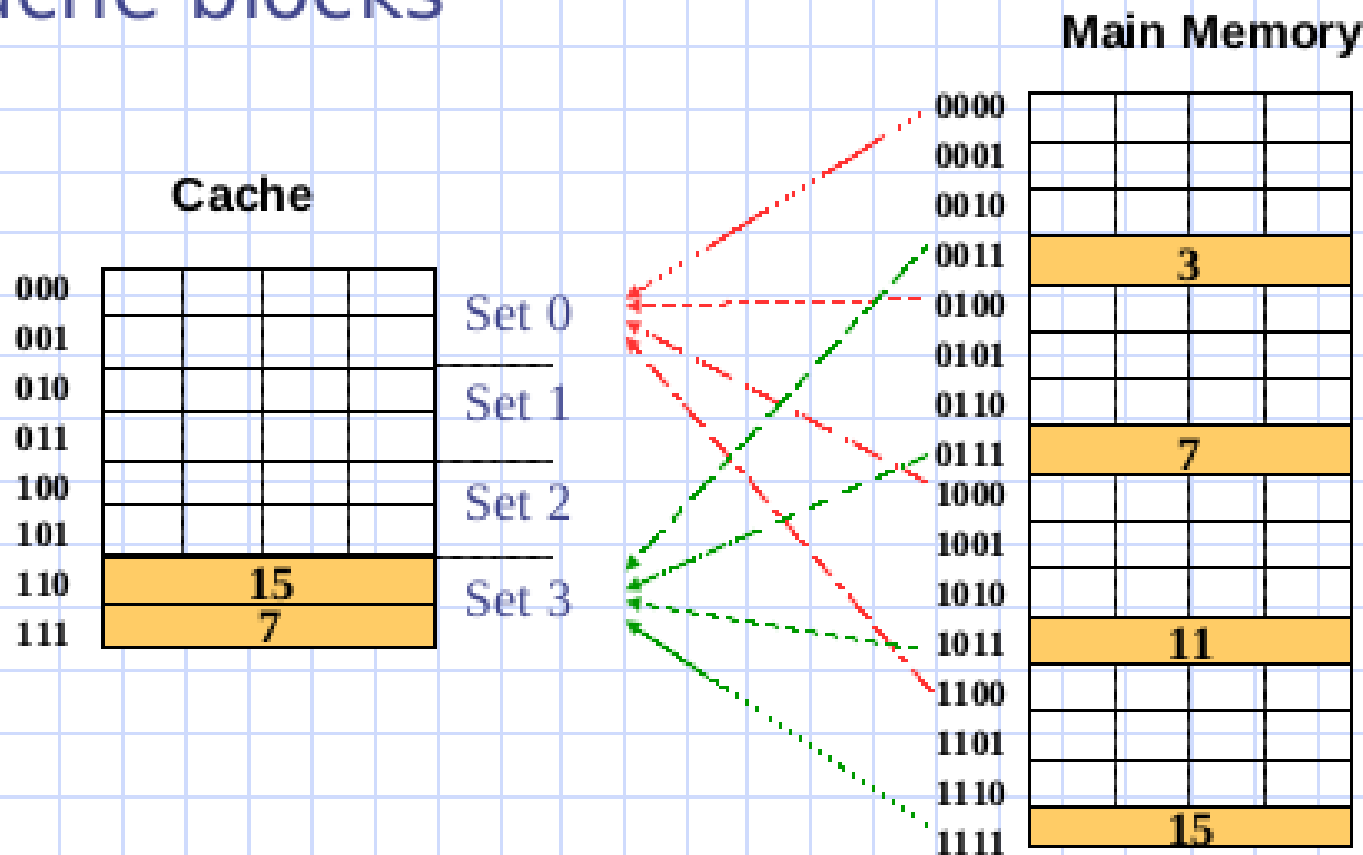
- Number of cache lines in a set = 1
- Refer figures 6.27-6.29 in book
- Disadvantages of direct-mapped caches?
- Conflict misses are more.
- Why?
- Problem is with a single cache line: A simple dot product example.

1. Alternative to Direct Mapping

- Set associative mapping
 - e.g., 2 way set associative: Number of cache lines = 2
 - Idea: A given memory block can be present in either of 2 lines of the cache

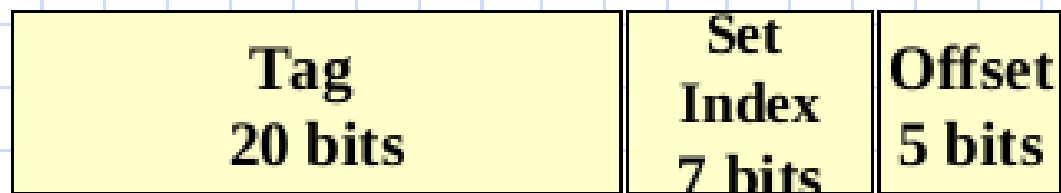
Set Associative Cache

- A memory block can be loaded into any cache block within a unique set of cache blocks

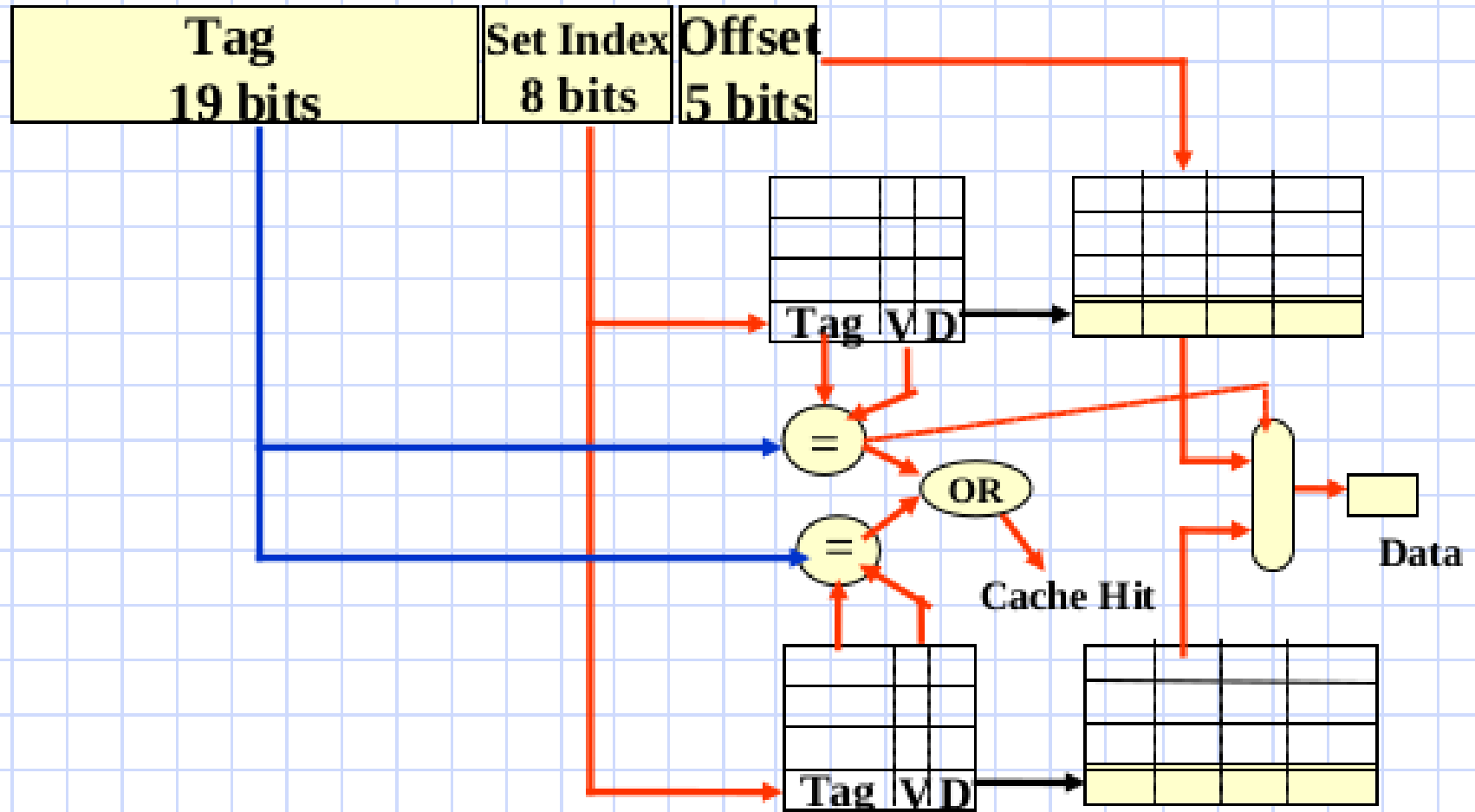


e.g., 4-way Set Associative Cache

- Assume 16KB cache, 32B block size
- $16\text{KB}/32\text{B} = 512$ blocks
- $512/4 = 128$ sets of blocks
- $\log_2 128 = 7$ set index bits
- $\log_2 32 = 5$ offset bits



e.g., 2-way Set Associative Cache



Set Associative Caches

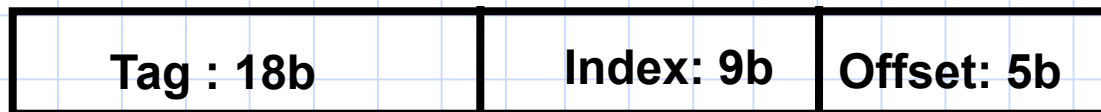
- Number of cache lines, m , in a set greater than 1
- Called *m-way associative* cache
- Refer figures 6.32-6.34 in the book

Fully Associative Caches

- Only one set having all cache lines
- Refer figures 6.35-6.37 in the book

Cache and Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Will look at some examples of programs
- Will consider only data cache, assuming separate instruction and data caches
- Data cache configuration:
 - Direct mapped 16 KB with 32B block size



Example 1: Vector Sum Reduction

```
double A[2048], sum=0.0;
```

```
for (i=0; i<2048, i++) sum = sum +A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)
 - Will assume that both loop index i and variable sum are implemented in registers
- Will consider only accesses to array elements

Example 1: Reference Sequence

- load A[0] load A[1] load A[2] ... load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000

Example 1: Reference Sequence

- load A[0] load A[1] load A[2] ... load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 1010 0000 0000 0000
 - Cache index bits: 100000000 (value = 256)
- Size of an array element (double) = 8B
- So, 4 consecutive array elements fit into each cache block (block size is 32B)
 - A[0] – A[3] have index of 256
 - A[4] – A[7] have index of 257 and so on

Example 1: Cache Misses and Hits

A[0]	0xA000	256	Miss	Cold Start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold Start

```
for (i=0; i<2048; i+=4) tmp=A[i];  
for (i=0; i<2048, i++)  
    sum = sum +A[i];
```

Cold start: we assume that the cache is initially empty

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

What if we precede the loop by a loop that accesses all relevant memory blocks?

Hit ratio of our loop would then be 100%. 25% due to temporal locality and 75% due to spatial locality

Example 1 with double A[4096]

Why should it make a difference?

- Consider the case where the loop is preceded by another loop that accesses all array elements in order
- The entire array no longer fits into the cache – cache size: 16KB, array size: 32KB
- After execution of the previous loop, the second half of the array will be in cache
- Analysis: our loop will see misses as we had calculated

Example 1: Vector Sum Reduction

```
double A[2048], sum=0.0;
```

```
for (i=0; i<2048, i++) sum = sum +A[i];
```

- To estimate data cache hit rate
 - we ignored accesses to sum, i
 - assumed address of A[0] is 0xA000
 - assumed only load/store instructions reference memory operands (others take their operands from registers)

Example 2: Vector Dot Product

```
double A[2048], B[2048], sum=0.0;
```

```
for (i=0; i<2048, i++) sum = sum +A[i] * B[i];
```

- Reference sequence:
 - load A[0] load B[0] load A[1] load B[1] ...
- Assume base addresses of A and B are 0xA000 and 0xE000
- Again, size of array elements is 8B so that 4 consecutive array elements fit into each cache block

Example 2: Vector Dot Product

Base addresses 0xA000 and 0xE000

.....10100000000000000000

Index: 256

.....11100000000000000000

Index: 256

Example 2: Cache Hits and Misses

A[0]	0xA000	256	Miss	Cold Start
B[0]	0xE000	256	Miss	Conflict
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict

Conflict: A miss due to conflict in cache block requirements caused by memory accesses of the same program

Hit ratio for our program:
0%

Source of the problem: the elements of arrays A and B are accessed in order and have the same cache index

Hit ratio would be better if the base address of array A was different from that of array B

Is this a contrived example?

```
double A[2048], B[2048], sum=0.0;
```

```
for (i=0; i<2048, i++) sum = sum +A[i] * B[i];
```

- How are variable addresses assigned?
- Start with some address, say 0xA000
- Assign addresses to variables in order of their declarations
- Array A: starting at 0xA000
- Array B: starting at $0xA000 + 2048 * 8$
= 0xE000

1010	0000	0000	0000
100	0000	0000	0000
1110	0000	0000	0000

Example 2: Cache Hits and Misses

A[0]	0xA000	256	Miss	Cold Start
B[0]	0xE000	256	Miss	Conflict
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict

Conflict: A miss due to conflict in cache block requirements caused by memory accesses of the same program

Hit ratio for our program:
0%

Source of the problem: the elements of arrays A and B are accessed in order and have the same cache index

Hit ratio would be better if the base address of array A was different from that of array B

Example 2 with Packing

- Assume that addresses are assigned as variables are encountered in declarations
- Our objective: to shift base address of B enough to make cache index of B[0] different from that of A[0]
double A[2052], B[2048];
- Base address of B is now 0xE020
 - 0xE020 is 1110 0000 0010 0000
 - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block
- Hit ratio of our loop will rise to 75%

Example 2 with Array Merging

Alternatively, declare the arrays as

```
struct {double A, B;} array[2048];
```

```
for (i=0; i<2048, i++)
```

```
    sum += array[i].A*array[i].B;
```

Hit ratio: 75%

Example 3: DAXPY

- Double precision $Y = aX + Y$, where X and Y are vectors and a is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```
- Reference sequence
 - ▣ load $X[0]$ load $Y[0]$ store $Y[0]$ load $X[1]$ load $Y[1]$ store $Y[1]$...
- Hits and misses: Assuming that base addresses of X and Y don't conflict in cache, hit ratio of 83.3%

Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];  
for (j=0;j<1024;j++)  
  for (i=0;i<1024;i++)  
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

```
load A[0,0] load B[0,0] store B[0,0]  
load A[1,0] load B[1,0] store B[1,0] ...
```

- Question: In what order are the elements of a multidimensional array stored in memory?

Storage of Multi-dimensional Arrays

■ Row major order

- Example: for a 2-dimensional array, the elements of the first row of the array are followed by those of the 2nd row of the array, then the 3rd row, and so on
- This is what is used in C

■ Column major order

- A 2-dimensional array is stored column by column in memory
- Used in FORTRAN

Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];  
for (j=0;j<1024;j++)  
  for (i=0;i<1024;i++)  
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

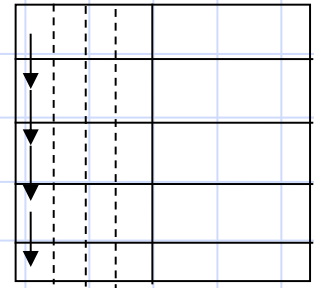
```
load A[0,0] load B[0,0] store B[0,0]  
load A[1,0] load B[1,0] store B[1,0] ...
```

- Question: In what order are the elements of a multidimensional array stored in memory?

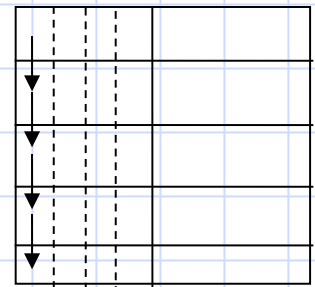
Example 4: Hits and Misses

- Reference order and storage order for our arrays are not the same
- Our loop will show no spatial locality
 - Assume that packing has been done to eliminate conflict misses due to base addresses
 - Miss(cold), Miss(cold), Hit for each array element
 - Hit ratio: 33.3%
 - Question: Will $A[0,1]$ be in the cache when required later in the loop?

• A



• B



Example 4 with Loop Interchange

```
double A[1024][1024], B[1024][1024];  
for (i=0;i<1024;i++)  
  for (j=0;j<1024;j++)  
    B[i][j] = A[i][j] + B[i][j];
```

■ Reference Sequence:

load A[0,0] load B[0,0] store B[0,0]

load A[0,1] load B[0,1] store B[0,1]

Hit ratio: 83.3%

Is Loop Interchange Always Safe?

```
for (j=1; j<2048; j++)
```

```
  for (i=1; i<2048; i++)
```

```
    A[i][j] = A[i+1][j-1] + A[i][j-1];
```

$A[1,1] = A[2,0] + A[1,0]$

$A[2,1] = A[3,0] + A[2,0]$

...

$A[1,2] = A[2,1] + A[1,1]$

Is Loop Interchange Always Safe?

for (i=1; i<2048; i++) / interchanged

for (j=1; j<2048; j++)

$A[i][j] = A[i+1][j-1] + A[i][j-1];$ **NO!**

$A[1,1] = A[2,0] + A[1,0]$

$A[2,1] = A[3,0] + A[2,0]$

...

$A[1,2] = A[2,1] + A[1,1]$

$A[1,1] = A[2,0] + A[1,0]$

$A[1,2] = A[2,1] + A[1,1]$

...

$A[2,1] = A[3,0] + A[2,0]$

Example 5: Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N];
```

```
for (i=0; i<N; i++)
```

```
    for (j=0; j<N; j++)
```

```
        for (k=0; k<N; k++)
```

```
            X[i][j] += Y[i][k] * Z[k][j];
```

Example 5: Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N], tmp;
```

```
for (i=0; i<N; i++)
```

```
    for (j=0; j<N; j++){
```

```
        tmp = 0;
```

```
        for (k=0; k<N; k++)
```

```
            tmp += Y[i][k] * Z[k][j];
```

```
        X[i][j] = tmp; / Dot product inner loop
```

```
    }
```

Y[0,0], Z[0,0], Y[0,1], Z[1,0], Y[0,2], Z[2,0] ... X[0,0],
Y[1,0], Z[0,1], Y[1,1], Z[1,1], Y[1,2], Z[2,1] ... X[0,1],
Y[2,0], Z[0,2], Y[2,1], Z[1,2], Y[2,2], Z[2,2] ... X[0,2],
...

Example 5: Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N], tmp;
```

```
for (i=0; i<N; i++)
```

```
    for (j=0; j<N; j++) {
```

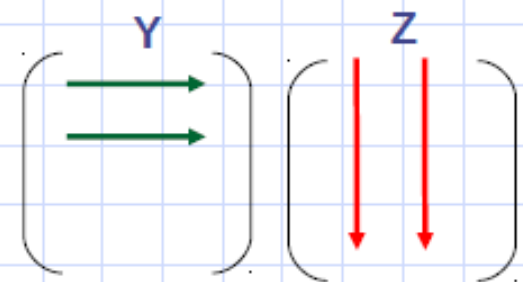
```
        tmp = 0;
```

```
        for (k=0; k<N; k++)
```

```
            tmp += Y[i][k] * Z[k][j];
```

```
        X[i][j] = tmp; / Dot product inner loop
```

```
    } Y[0,0], Z[0,0], Y[0,1], Z[1,0], Y[0,2], Z[2,0] ... X[0,0],  
      Y[1,0], Z[0,1], Y[1,1], Z[1,1], Y[1,2], Z[2,1] ... X[0,1],  
      Y[2,0], Z[0,2], Y[2,1], Z[1,2], Y[2,2], Z[2,2] ... X[0,2],  
      ...
```



Matmul: Loop Interchange

- We can interchange the 3 loops
- Example: Interchange i and k loops – make the loops “kji” instead of “ijk”

```
double X[N][N], Y[N][N], Z[N][N];
```

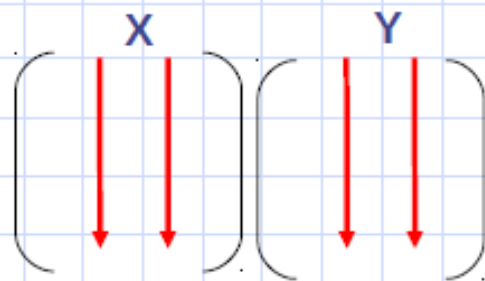
```
for (k=0; k<N; k++)
```

```
    for (j=0; j<N; j++)
```

```
        for (i=0; i<N; i++)
```

```
            X[i][j] += Y[i][k] * Z[k][j];
```

- For the innermost loop: $Z[k][j]$ can be loaded into register once for each (k,j) , reducing the number of memory references



Analysis of loop interchange

- Will the result change?
- Will the total operations remain the same?
- Will the number of times X and Y are read remain the same?
- What about performance?
- Assumptions:
 - Elements are double elements – 8 bytes
 - Cache has 32-byte block size ($B=32$)

3 loops (i,j,k) - Can come up with 6 different versions

ijk variant

```
for(i=0; i<N; i++)  
  for(j=0; j<N; j++)  
    sum=0.0;  
    for(k=0;k<N; k++)  
      sum += A[i][k]*B[k][j]  
    C[i][j] = sum;
```

- Loads per iteration – 2
- Stores per iteration – 0
- A misses per iteration – 0.25 (stride?)
- B misses per iteration – 1.00 (stride)
- C misses per iteration – 0.00
- Total misses per iteration – 1.25

jik variant

```
for(j=0; j<N; j++)  
  for(i=0; i<N; i++)  
    sum=0.0;  
    for(k=0; k<N; k++)  
      sum += A[i][k]*B[k][j]  
    C[j][j] = sum;
```

- Loads per iteration – 2
- Stores per iteration – 0
- A misses per iteration – 0.25
- B misses per iteration – 1.00
- C misses per iteration – 0.00
- Total misses per iteration – 1.25
- Same as ijk variant

jki variant

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

- Loads per iteration – 2
- Stores per iteration – 1
- A misses per iteration – 1.00 (stride?)
- B misses per iteration – 0.00
- C misses per iteration – 1.00 (stride?)
- Total misses per iteration – 2.00

kji variant

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

- Loads per iteration – 2
- Stores per iteration – 1
- A misses per iteration – 1.00
- B misses per iteration – 0.00
- C misses per iteration – 1.00
- Total misses per iteration – 2.00

- Same as jki variant

kij variant

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

- Loads per iteration – 2
- Stores per iteration – 1
- A misses per iteration – 0.00
- B misses per iteration – 0.25 (stride?)
- C misses per iteration – 0.25 (stride?)
- Total misses per iteration – 0.50

ikj variant

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

- Loads per iteration – 2
- Stores per iteration – 1
- A misses per iteration – 0.00
- B misses per iteration – 0.25 (stride?)
- C misses per iteration – 0.25 (stride?)
- Total misses per iteration – 0.50

- Same as kij variant

Summary

Finding the best performance involves trade-off between

Cache performance

Number of memory accesses

How to improve the cache hits and performance further?

Loop unrolling and blocking

“Loop Unrolling”

```
double X[10];  
for (i=0; i<10; i++)  
    X[i] = X[i] - 1;
```

Unrolled once:

```
for (i=0; i<10; i+=2){  
    X[i] = X[i] - 1;  
    X[i+1] = X[i+1] - 1;  
}
```

Fully unrolled:

```
X[0] = X[0] - 1;  
X[1] = X[1] - 1;  
X[2] = X[2] - 1;  
...  
X[9] = X[9] - 1;
```

Unrolling Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N];  
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        for (k=0; k<N; k++)  
            X[i][j] += Y[i][k] * Z[k][j];
```

Let us unroll the k loop once

Matmul: k loop unrolled

```
double X[N][N], Y[N][N], Z[N][N];
```

```
for (i=0; i<N; i++)
```

```
    for (j=0; j<N; j++)
```

```
        for (k=0; k<N; k+=2) /* k loop unrolled once
```

```
            X[i][j] += Y[i][k] * Z[k][j] + Y[i][k+1] * Z[k+1][j];
```

Now, let us also unroll the j loop once

Matmul: k and j loops unrolled

```
double X[N][N], Y[N][N], Z[N][N];
```

```
for (i=0; i<N; i++)
```

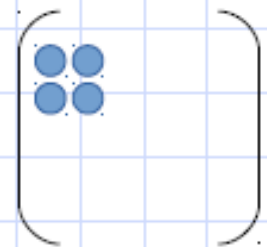
```
    for (j=0; j<N; j+=2)
```

```
        for (k=0; k<N; k+=2){    /* j and k loops unrolled once
```

```
            X[i][j] += Y[i][k] * Z[k][j] + Y[i][k+1] * Z[k+1][j];
```

```
            X[i][j+1] += Y[i][k] * Z[k][j+1] + Y[i][k+1] * Z[k+1][j+1];
```

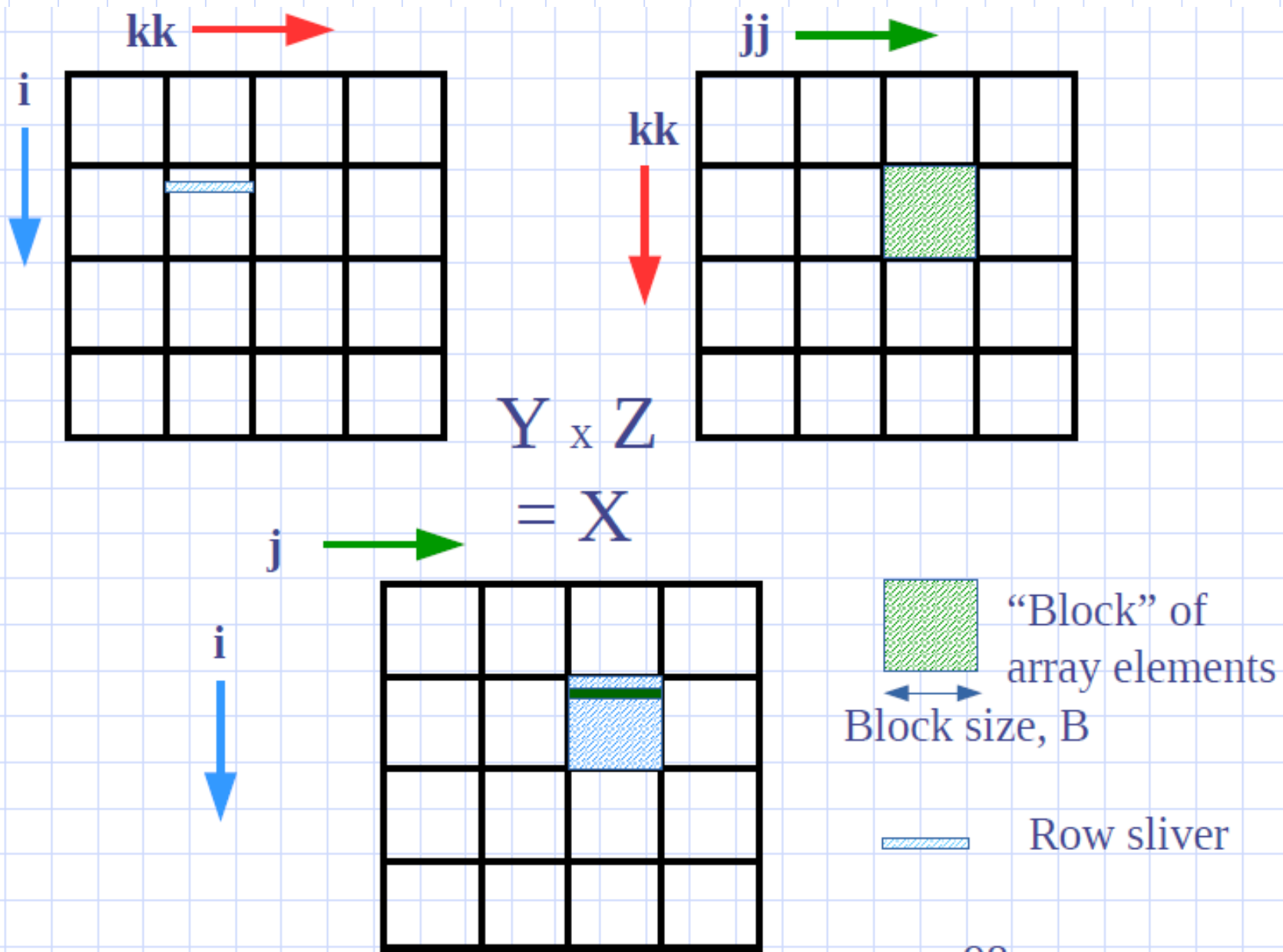
```
        }
```



Exploits spatial locality for arrays Y, Z

Exploits temporal locality for array Y

Provides a programming idea for enhancing locality



Matmul: “Blocking” or “Tiling”

```
double X[N][N], Y[N][N], Z[N][N];
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 = 0.0;
                for (k=kk; k<min(kk+B, N), k++){
                    sum += Y[i][k] * Z[k][j];
                }
                X[i][j] += sum;
            } /* for j */
        } /* for i */
```

Vector Operations

Example: Vector Sum

```
double A[2048], B[2048], C[2048];  
for (i=0; i<2048, i++) C[i] = A[i] + B[i];
```

- What if a CPU has 4 adders?
- It can be designed to support an instruction to do 4 iterations of the Vector Sum loop at a time

```
VADD v1_A[0:3], v2_B[0:3], v3_C[0:3]
```

- Called a vector instruction

Multimedia Extensions

- Hardware support for operations on “short vectors” is provided in existing microprocessors
- Example: 256 bit registers, each split into 4x64b (or 8x32b)
 - Maximum vector length
- Example: Intel “x86” processors
 - SSE (Streaming SIMD Extension)
 - AVX (Advanced Vector Extension)

Vectorization of Loops

We will use a generic notation

Instead of

VADD C[0:3], A[0:3], B[0:3]

$C[0:3] = A[0:3] + B[0:3]$

An example of vectorization

- Given maximum vector length, VL

```
for (i=0; i < N; i++)
```

```
    A[i] = A[i] + B[i];
```

```
for (i=0; i < N; i+=VL)
```

```
    A[i:i+VL-1] = A[i:i+VL-1] + B[i:i+VL-1];
```

What if N is not divisible by VL?

```
for (i=0; i < (N - N%VL); i+=VL)
```

```
    A[i:i+VL-1] = A[i:i+VL-1] + B[i:i+VL-1];
```

```
for (; i<N; i++) A[i] = A[i] + B[i];
```

- This technique is called Stripmining

Possible complications

- Dependences between statements within the loop

Example 1

```
for (i=0; i < N; i++) {
```

```
    A[i] = B[i] + C[i];
```

```
    D[i] = (A[i] + A[i+1])/2;
```

```
}
```

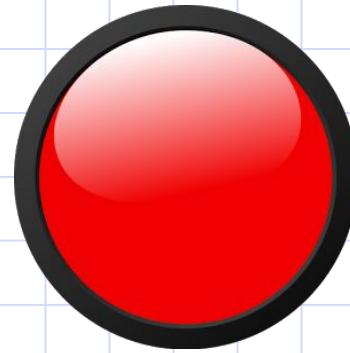
```
for (i=0; i < (N - N%VL); i+=VL){
```

```
    A[i:i+VL-1] = B[i:i+VL-1] + C[i:i+VL-1];
```

```
    D[i: ... will get wrong value of A[i+1], etc
```

Example 1

```
for (i=0; i < N; i++) {  
    A[i] = B[i] + C[i];  
    D[i] = (A[i] + A[i+1])/2;  
}
```



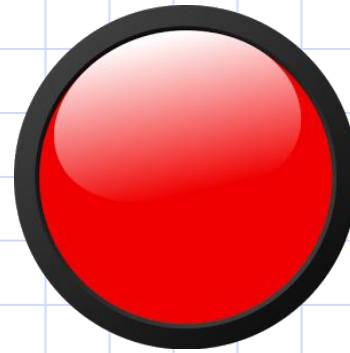
```
for (i=0; i < N; i++) {  
    temp[i] = A[i+1];  
    A[i] = B[i] + C[i];  
    D[i] = (A[i] + temp[i])/2;  
}
```



- This loop transformation, through copying of data, is called Node Splitting

Example 2

```
for (i=0; i < N; i++) {  
    X = A[i] + 1;  
    B[i] = X + C[i];  
}
```



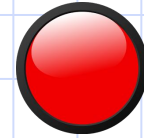
```
for (i=0; i < N; i++) {  
    temp[i] = A[i] + 1;  
    B[i] = temp[i] + C[i];  
}
```



- Scalar expansion

Example 3

```
for (i=0; i < N; i++) {  
    A[i] = B[i];  
    C[i] = C[i-1] + 1;  
}
```



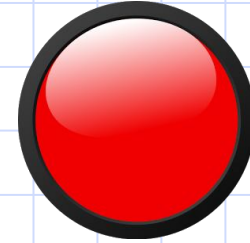
```
for (i=0; i < N; i++) A[i] = B[i];  
for (i=0; i < N; i++) C[i] = C[i-1] + 1;
```



- Loop fission

Example 4

```
for (j=1; i < N; j++)  
    for (i=2; i < N; i++)  
        A[i,j] = A[i-1, j] + B[i];
```



```
for (i=2; i < N; i++)  
    for (j=1; j<N; j++)  
        A[i,j] = A[i-1,j] + B[i];
```



- Loop interchange

Data Representation

Integer Data

- Signed vs Unsigned integer
- Representing a signed integer
 - 2s complement representation

The n bit quantity

$$x_{n-1}x_{n-2}\dots x_2x_1x_0$$

least significant bit



represents the signed integer value

$$-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

Example: 2s complement

- The signed integer -14_{10} (decimal) is represented as
 - 10010 in 5 bits (i.e., $-16 + 2$)
 - 110010 in 6 bits (i.e., $-32 + 16 + 2$)
 - 111...1110010 in 32 bits

Aside: Hexadecimal (base 16)

- Digits 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 0001 0010 ... 1101 1110 1111

- Binary sequences can be written more compactly in hexadecimal

1001 9

1010 A

1011 B

1100 C

1101 D

1110 E

1111 F

Example: 2s complement

- The signed integer -14_{10} (decimal) is represented as

10010 in 5 bits ($-16 + 2$)

110010 in 6 bits ($-32 + 16 + 2$)

111...1110010 in 32 bits

1111 1111 1111 ... 0010

FFFFFFFF2

Usually written as 0xFFFFFFFF2

Real Data

- How to represent real data?
- Fixed point representation
- (sign bit) (Integer part – 23 bits)(Fraction – 8 bits)
- Disadvantage?
- The range of numbers not adequate for many practical problems
- Hence, floating point representation

Normalized floating point numbers

- Consists of two parts
 - Mantissa with sign
 - Exponent with sign
- Floating point represented as
 - $(\text{sign}) \times \text{mantissa} \times 2^{\pm \text{exponent}}$
 - Mantissa (23 bits) – a binary fraction with non-zero leading bit
 - Exponent (8 bits) – 1 bit used for sign of the exponent, other 7 bits used for the exponent magnitude
 - Range of exponent?: -127 to +127
- Disadvantage: Two representation for 0 exponent: -0 and +0

Excess representation or bias format

- Exponent has no sign bit
- 8 bits of exponent divided as
 - (0-127) and (128-255)
 - (0-126): negative
 - 127: represents 0
 - (128-255): positive
- Called as bias 127 for exponent
- Given an exponent, exp , value of exponent will be $(exp - 127)$
- Largest and smallest floating point numbers that can be represented?

Example

Representing 52.21875 in 32-bit floating point format.

Step 1 – Represent using binary: 110100.00111

Step 2 – Normalized representation: 1.1010000111×2^5

Exponent of 5 represented as $(127+5=132) = 10000100$

Floating point represented as (sign)(exponent-8 bits)(mantissa – 23 bits)

0 10000100 101000011100000000000000

Note that the leading 1 in the normalized representation is ignored in the mantissa.

IEEE 754 Floating Point Standard

The scheme that is just described is IEEE 754 floating point standard

Mantissa is called *significand* as per the standard

The standard uses a normalized significand – most significant bit is always 1

Thus significand is 24 bits long – 1 is implied + 23 explicit

Floating point number represented by:
$$(-1)^s \times (1.f)^2 \times 2^{(\text{exponent}-127)}$$

Special Cases (B&O 2.4.2)

- Representation of 0: All 31 (exponent and mantissa/significand/fraction) bits are 0's
 - +0: 0 for the sign bit
 - -0: 1 for the sign bit
 - All 0's for the exponents is not allowed to be used for any other number
- Infinity: All 1's in the exponent and all 0's in the mantissa
 - +infinity: 0 for the sign bit
 - -infinity: 1 for the sign bit

Largest and Smallest Positive Numbers?

Rounding (B&O 2.4.4)

- When mathematical operations are performed with two floating point numbers, the significand of the result may exceed 23 bits after the adjustment of the exponent.
- Rounding:
 - Rounding upwards:
 - e.g., significand: 0.110...01,
 - overflow: 1,
 - significand after rounding: 0.110....10, i.e., add 1 to LSB.
 - Rounding downwards: extra bits ignored

Summary (from notes by Prof. Rajaraman)

Value	Sign	Exponent (8 bits)	Significand (23 bits)
+0	0	00000000	00 ... 00 (all 23 bits 0)
- 0	1	00000000	00 ... 00 (all 23 bits 0)
$+ 1.f \times 2^{(e-b)}$ e exponent, b bias	0	00000001 to 11111110	$a a \dots a a$ ($a = 0$ or 1)
$- 1.f \times 2^{(e-b)}$	1	00000001 to 11111110	$a a \dots a a$ ($a = 0$ or 1)
$+\infty$	0	11111111	000 ... 00 (all 23 bits 0)
$-\infty$	1	11111111	000 ... 00 (all 23 bits 0)
SNaN	0 or 1	11111111	000 ... 01 to 011... 11 leading bit 0 (at least one 1 in the rest)
QNaN	0 or 1	11111111	1000 ... 10 leading bit 1 (at least one 1)
Positive subnormal $0.f \times 2^{x+1-b}$ (x is the number of leading 0s in significand)	0	00000000	000 ... 01 to 111... 11

Summary (from notes by Prof. Rajaraman)

Operation	Result	Operation	Result
$n / \pm \infty$	0	$\pm 0 / \pm 0$	NaN
$\pm \infty / \pm \infty$	$\pm \infty$	$\infty - \infty$	NaN
$\pm n / 0$	$\pm \infty$	$\pm \infty / \pm \infty$	NaN
$\infty + \infty$	∞	$\pm \infty \times 0$	NaN

IEEE-754 Standard for 64-bit floating point numbers

- s: 1 bit
- e: 11 bits
- f: 52 bits

Reading

- Read sections in Bryant and O'Hallaron on the topics we have discussed in class
 - Bryant, O'Hallaron. Computer Systems – A Programmer's Perspective, Pearson Education Limited 2016, 3rd Global Edition
- Try to solve some of the problems