# 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

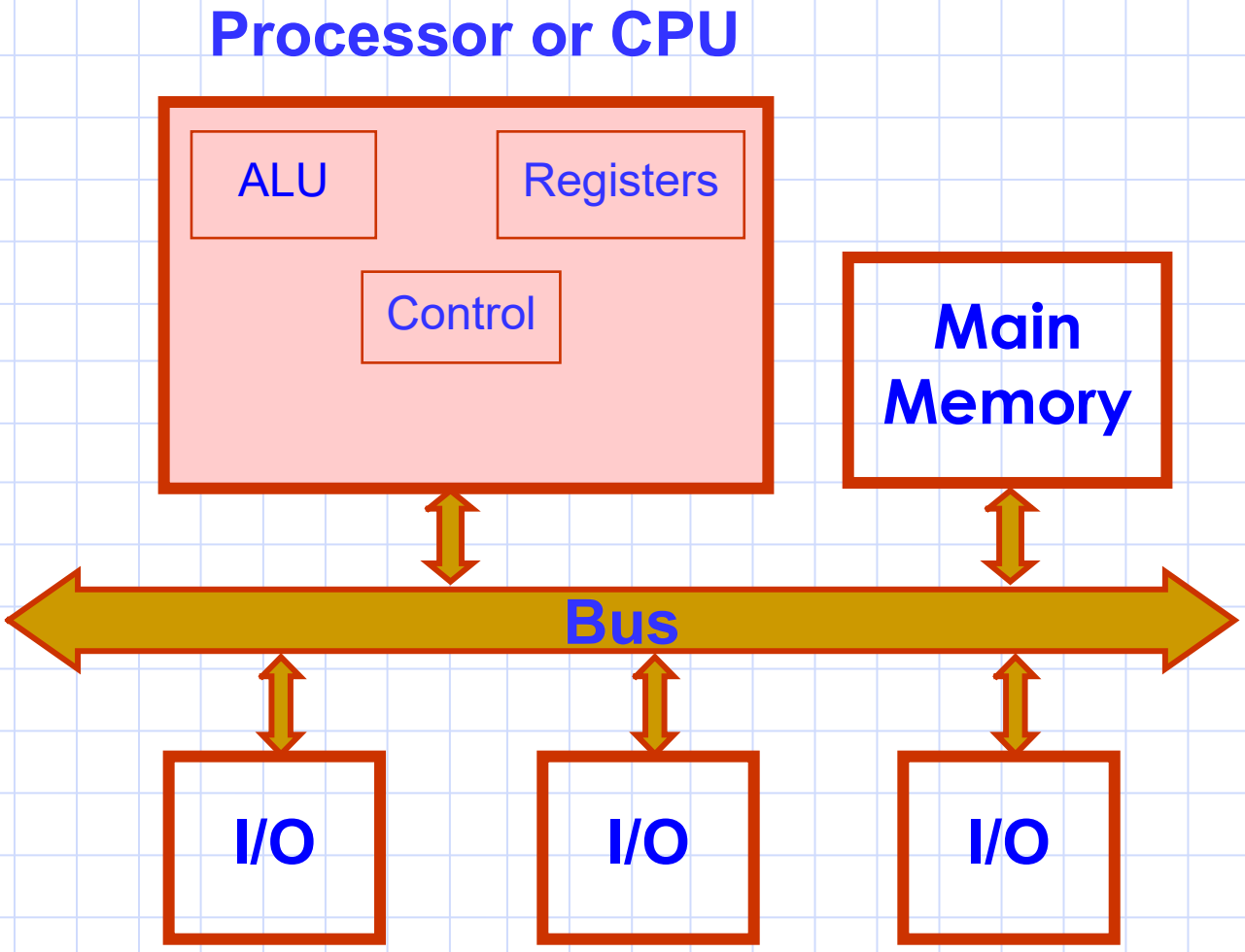
Pacheco. An Introduction to Parallel Programming

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

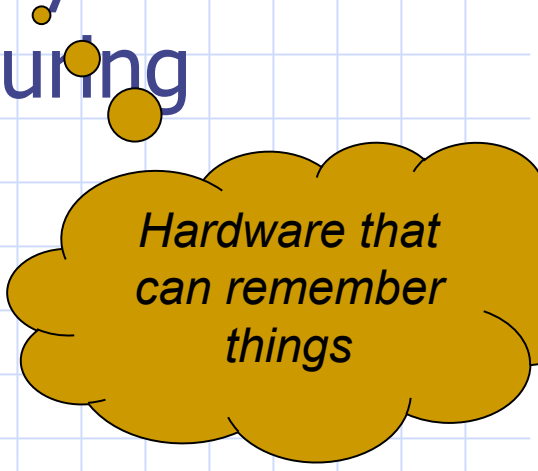


# Inside the Processor…

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

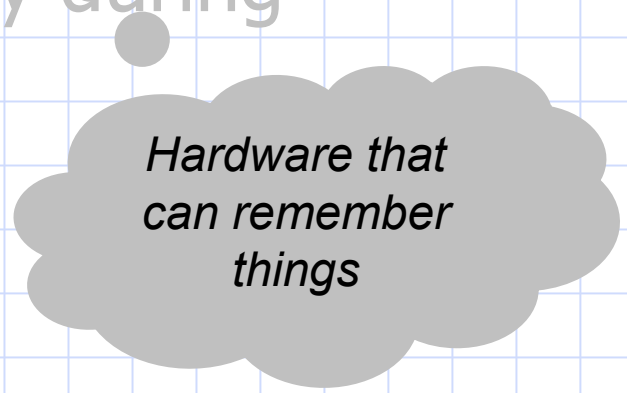
# Inside the Processor…

- 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



# 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



# 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: ~ 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?

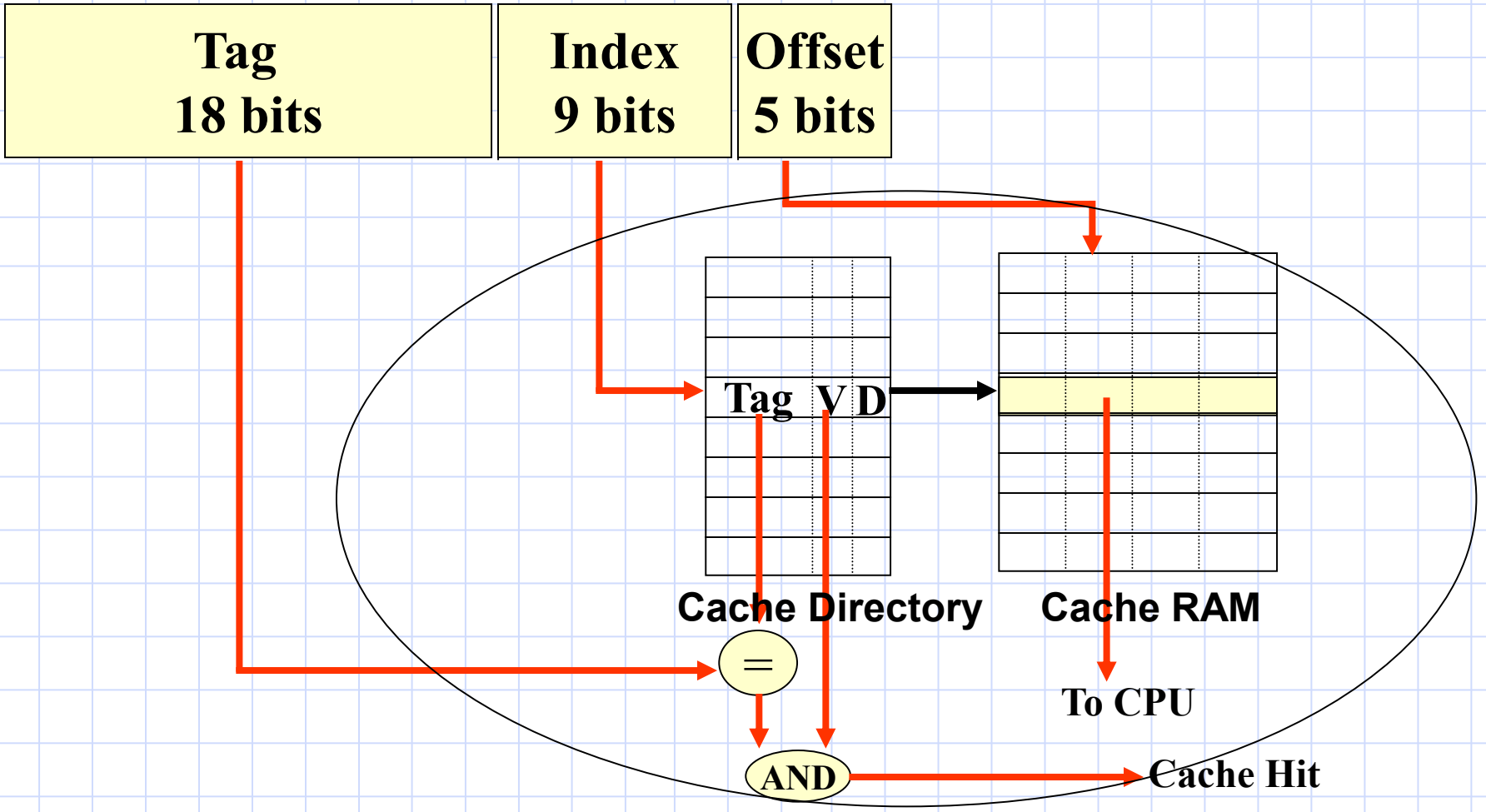
# Memory Address and Cache Composition

- 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

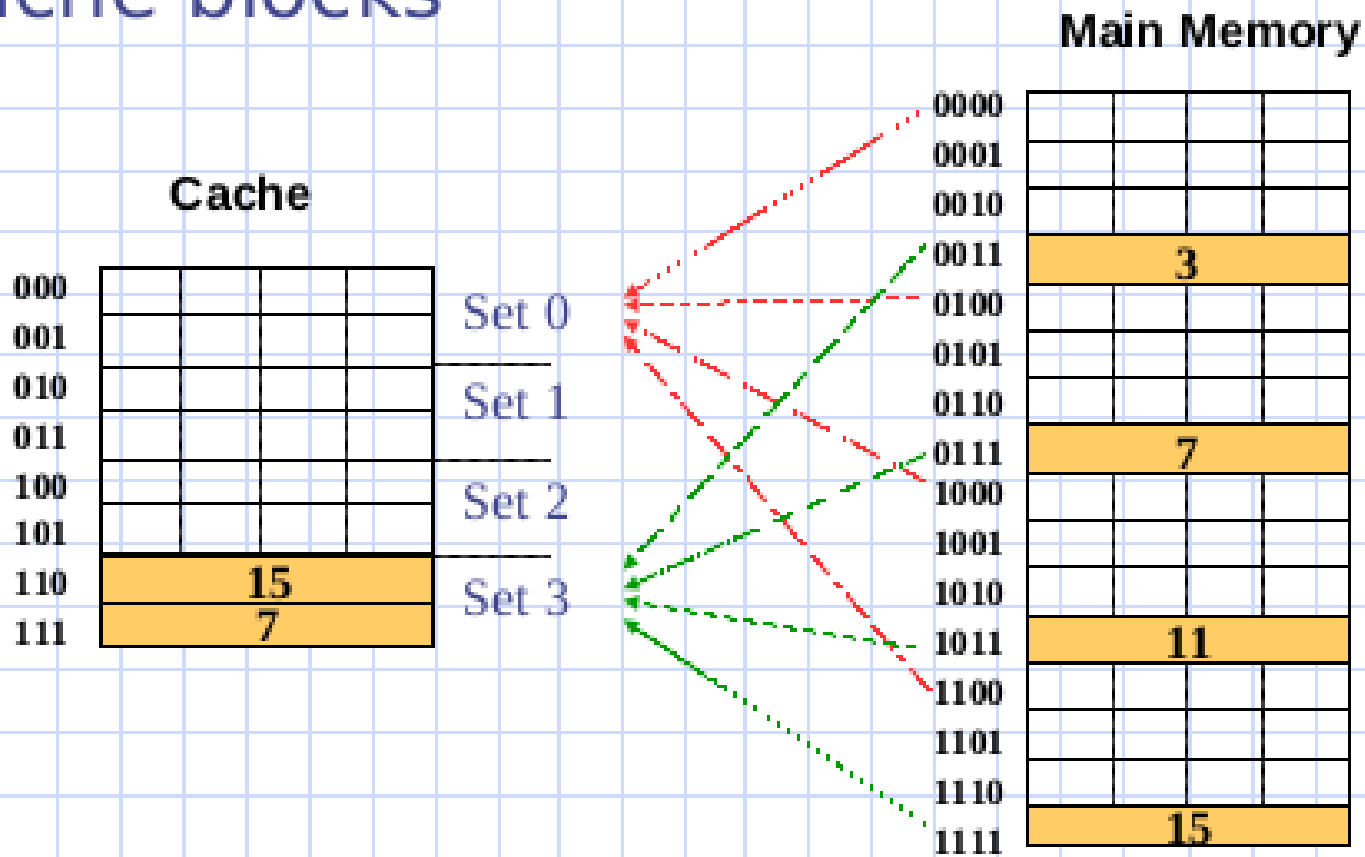
- Number of cache lines in a set = 1
- Refer figures 6.27-6.29 in book
- Disadvantages of direct-mapped caches?
- Conflict misses are more.
- Why?
- Problem is with a single cache line: A simple dot product example.

# 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

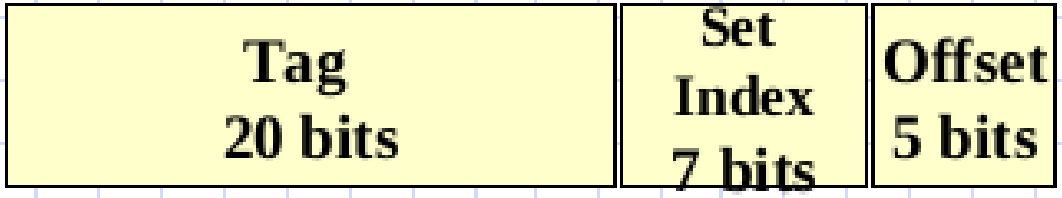
# Set Associative Cache

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

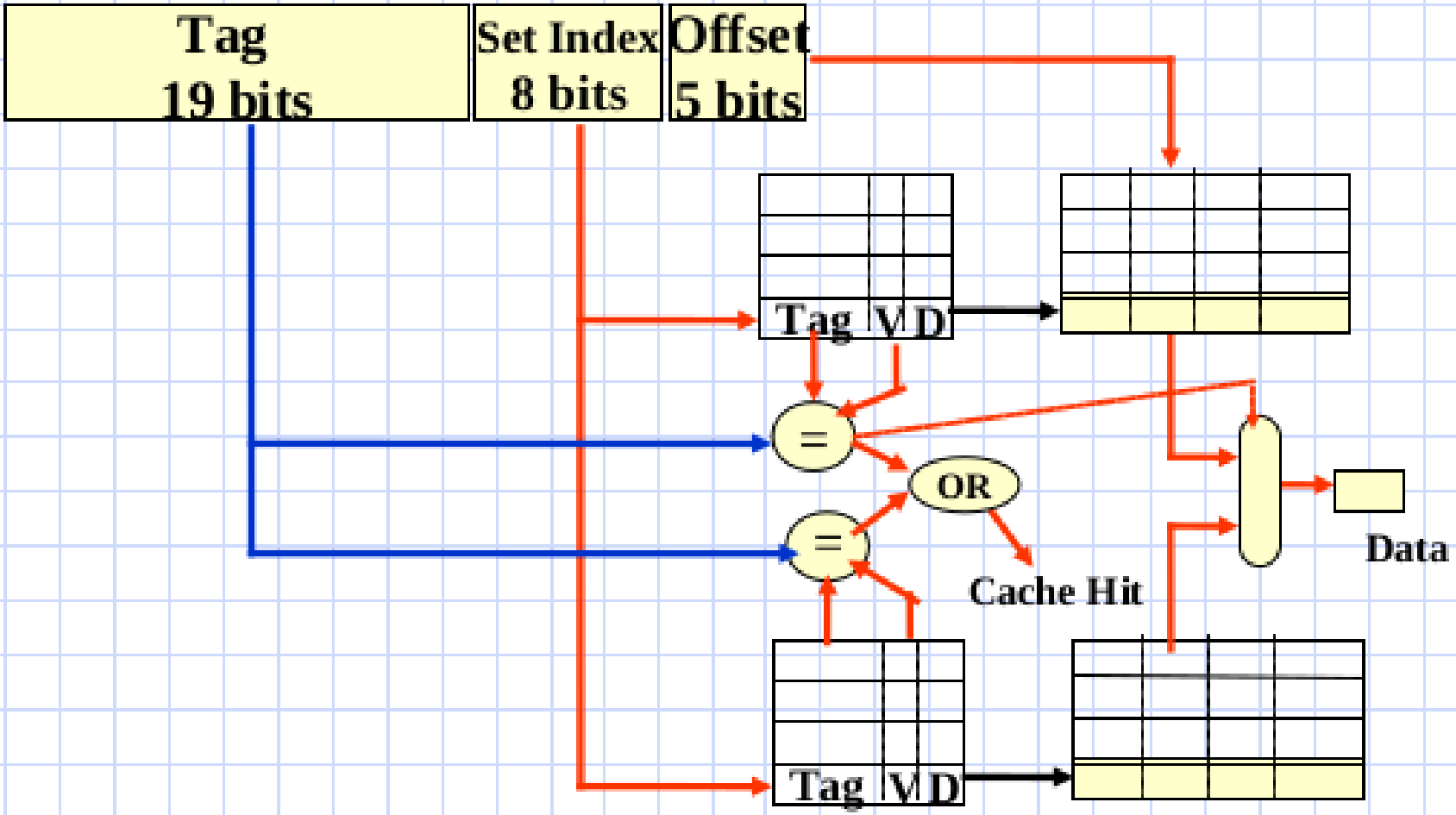


# e.g., 4-way Set Associative Cache

- Assume 16KB cache, 32B block size
- 16KB/32B = 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

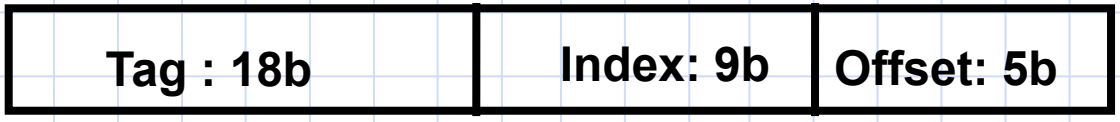
- Refer figures 6.32-6.34 in the book

# 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

.....1010000000000000

Index: 256

.....1110000000000000

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:

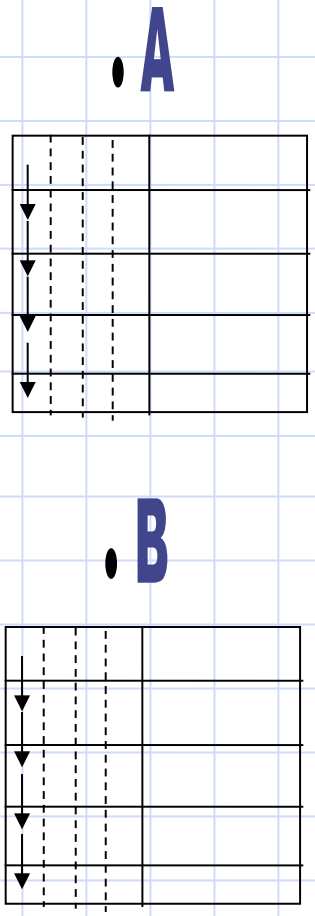
  load A[0,0] load B[0,0] store B[0,0]

  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?



# 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];
```

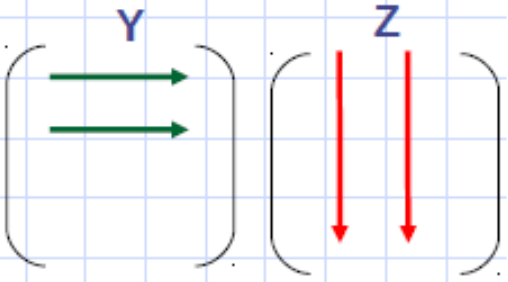
# Example 5: Matrix Multiplication

```
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
  }
```

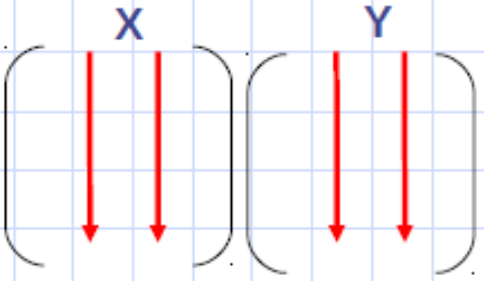



Y[0,0], Z[0,0], Y[0,1], Z[1,0], Y[0,2], Z[2,0] … X[0,0],
Y[1,0], Z[0,1], Y[1,1], Z[1,1], Y[1,2], Z[2,1] … X[0,1],
Y[2,0], Z[0,2], Y[2,1], Z[1,2], Y[2,2], Z[2,2] … X[0,2],
…

# 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

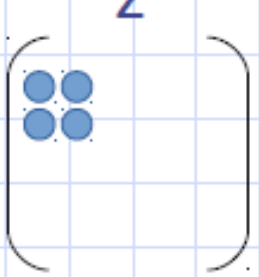
# Matmul: k loop unrolled

```
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

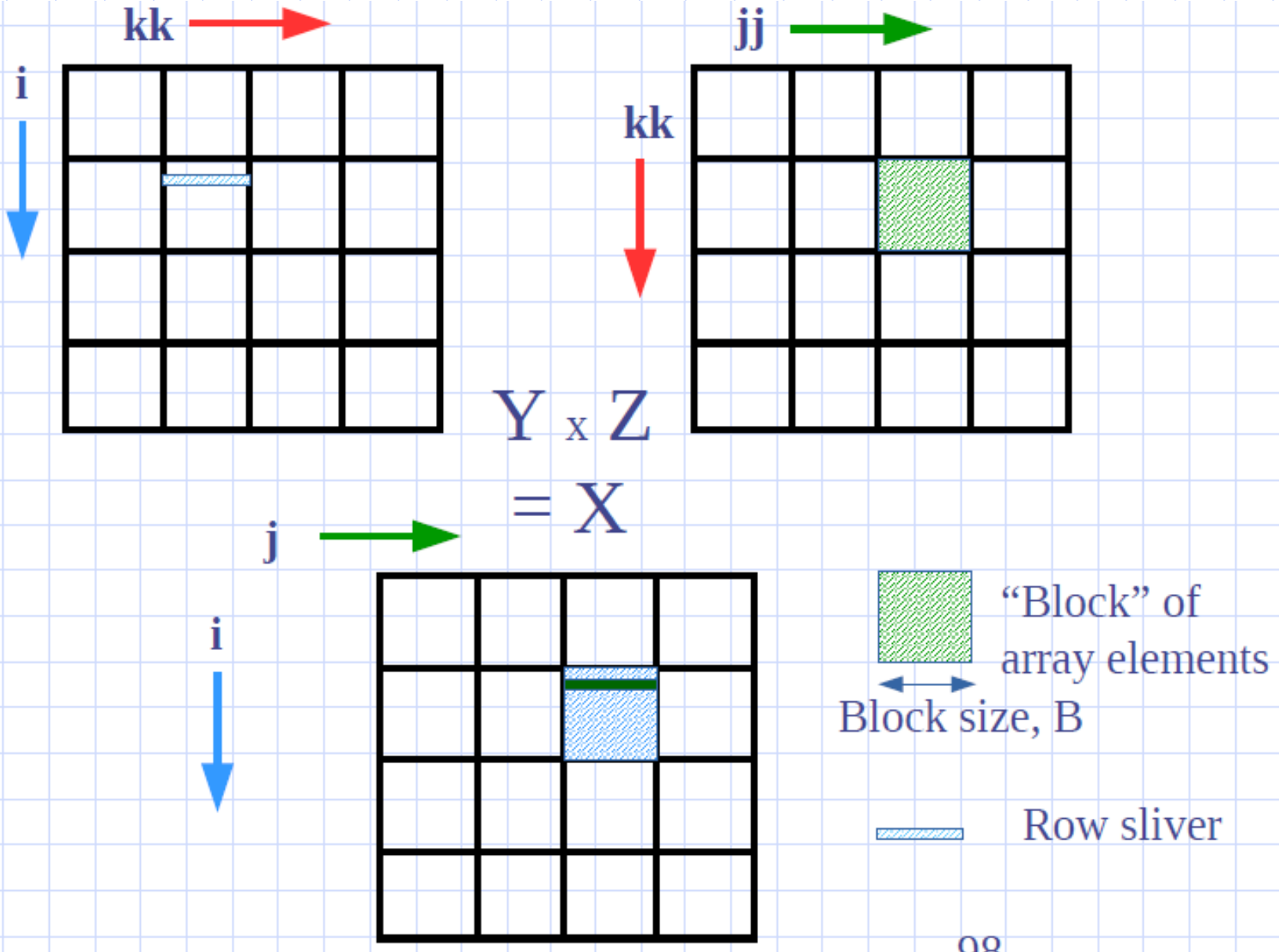
```
double X[N][N], Y[N][N], Z[N][N];
for (i=0; i<N; i++)
    for (j=0; j<N; j+=2)
        for (k=0; k<N; k+=2){      /* j  and k loops unrolled once
            X[i][j]    += Y[i][k] * Z[k][j]+Y[i][k+1] * Z[k+1][j];
            X[i][j+1]+= Y[i][k] * Z[k][j+1]+Y[i][k+1] * Z[k+1][j+1];
        }
```




**Exploits spatial locality  for arrays Y, Z**

**Exploits temporal locality for array Y**

Provides a programming idea for enhancing locality


kk
i
jj
kk
Y x Z
= X
j
i
"Block" of
array elements
Block size, B
Row sliver

# 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

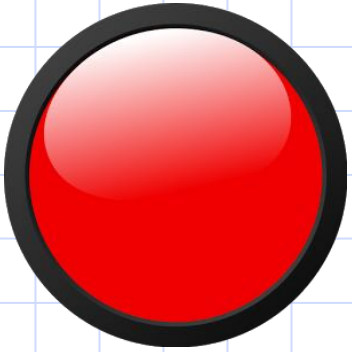
- Dependences between statements within the loop

# 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;
 }
```

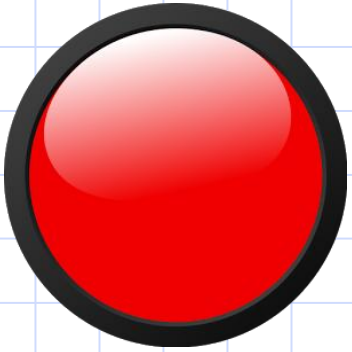


```
for (i=0; i < N; i++) {
      temp[i] = A[i+1];
      A[i] = B[i] + C[i];
      D[i] = (A[i] + temp[i])/2;
}
```

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

# Example 2

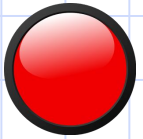
```
for (i=0; i < N; i++) {
      X = A[i] + 1;
      B[i] = X + C[i];
 }
for (i=0; i < N; i++) {
      temp[i] = A[i] + 1;
      B[i] = temp[i] + C[i];
}
```



- Scalar expansion

# Example 3

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

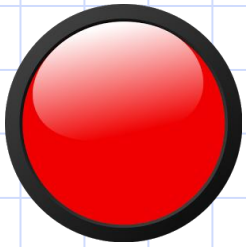


```
for (i=0; i < N; i++) A[i] = B[i];
for (i=0; i<N; i++) C[i] = C[i-1] +1;
```

- 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];
```



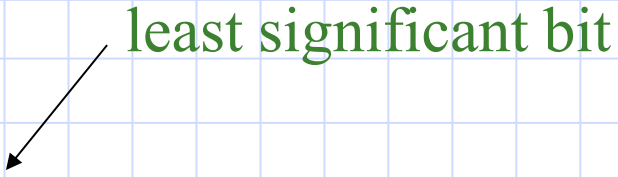
- Loop interchange

# Data Representation

# Integer Data

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

The $n$ bit quantity




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

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

  FFFFFFF2

  Usually written as 0xFFFFFFF2

# 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
- (sign) x mantissa x $2^{+/-\text{exponent}}$
- 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: $1.1010000111 \times 2^5$

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

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

0 10000100 10100001110000000000000

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

Mantessa 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 | $a\ a\ \ldots.\ a\ a$ | ($a$ = 0 or 1) |
| $- 1.f \times 2^{(e-b)}$ | 1 | 00000001 to 11111110 | $a\ a\ \ldots.\ a\ a$ | ($a$ = 0 or 1) |
| + ∞ | 0 | 11111111 | 000 … 00 | (all 23 bits 0) |
| − ∞ | 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 |
| 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 | (at least one 1) |

# 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 |
| $\infty + \infty$ | $\infty$ | $\pm\infty \times 0$ | 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

# IEEE Standard for Floating Point Numbers

*V Rajaraman*

**Floating point numbers are an important data type in computation which is used extensively. Yet, many users do not know the standard which is used in almost all computer hardware to store and process these. In this article, we explain the standards evolved by The Institute of Electrical and Electronic Engineers in 1985 and augmented in 2008 to represent floating point numbers and process them. This standard is now used by all computer manufacturers while designing floating point arithmetic units so that programs are portable among computers.**

**V Rajaraman is at the Indian Institute of Science, Bengaluru. Several generations of scientists and engineers in India have learnt computer science using his lucidly written textbooks on programming and computer fundamentals. His current research interests are parallel computing and history of computing.**

## Introduction

There are two types of arithmetic which are performed in computers: integer arithmetic and real arithmetic. Integer arithmetic is simple. A decimal number is converted to its binary equivalent and arithmetic is performed using binary arithmetic operations. The largest positive integer that may be stored in an 8-bit byte is +127, if 1 bit is used for sign. If 16 bits are used, the largest positive integer is +32767 and with 32 bits, it is +2147483647, quite large! Integers are used mostly for counting. Most scientific computations are however performed using real numbers, that is, numbers with a fractional part. In order to represent real numbers in computers, we have to ask two questions. The first is to decide how many bits are needed to encode real numbers and the second is to decide how to represent real numbers using these bits. Normally, in numerical computation in science and engineering, one would need at least 7 to 8 significant digits precision. Thus, the number of bits needed to encode 8 decimal digits is approximately 26, as $\log_2 10 = 3.32$ bits are needed on the average to encode a digit. In computers, numbers are stored as a sequence of 8-bit bytes. Thus 32 bits (4 bytes) which is bigger than 26 bits is a logical size to use for real numbers. Given 32 bits to encode real

**Keywords**
Floating point numbers, rounding, decimal floating point numbers, IEEE 754-2008 Standard.

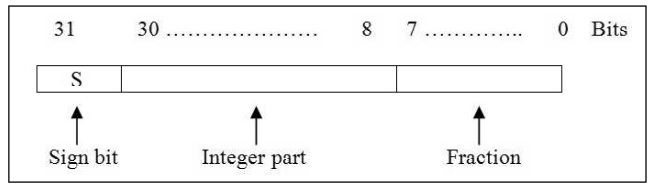



**Figure 1.** Fixed point representation of real numbers in binary using 32 bits.

numbers, the next question is how to break it up into an integer part and a fractional part. One method is to divide the 32 bits into 2 parts, one part to represent an integer part of the number and the other the fractional part as shown in *Figure* 1.

In this figure, we have arbitrarily fixed the (virtual) binary point between bits 7 and 8. With this representation, called *fixed point representation,* the largest and the smallest positive binary numbers that may be represented using 32 bits are

$$\text{Largest: } +\underbrace{111\ \ldots..}_{23\text{ bits}}\ \underbrace{1.11111111}_{8\text{ bits}} = 1677215.998046875$$

$$\text{Smallest: } +\underbrace{000\ \ldots}_{23\text{ bits}}\ \underbrace{0.00000001}_{8\text{ bits}} = 0.00390625$$

### Binary Floating Point Numbers

This range of real numbers, when fixed point representation is used, is not sufficient in many practical problems. Therefore, another representation called *normalized floating point* representation is used for real numbers in computers. In this representation, 32 bits are divided into two parts: a part called the *mantissa* with its sign and the other called the *exponent* with its sign. The mantissa represents fractions with a non-zero leading bit and the exponent the power of 2 by which the mantissa is multiplied. This method increases the range of numbers that may be represented using 32 bits. In this method, a binary floating point number is represented by

As the range of real numbers representable with fixed point is not sufficient, normalized floating point is used to represent real numbers.

$$(\text{sign}) \times \text{mantissa} \times 2^{\pm \text{exponent}}$$

where the sign is one bit, the mantissa is a binary fraction with a non-zero leading bit, and the exponent is a binary integer. If 32

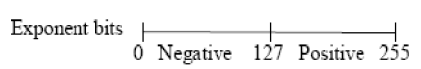
bits are available to store floating point numbers, we have to decide the following:

1. How many bits are to be used to represent the mantissa (1 bit is reserved for the sign of the number).

2. How many bits are to be used for the exponent.

3. How to represent the sign of the exponent.

The number of bits to be used for the mantissa is determined by the number of significant decimal digits required in computation, which is at least seven. Based on experience in numerical computation, it is found that at least seven significant decimal digits are needed if results are expected without too much error. The number of bits required to represent seven decimal digits is approximately 23. The remaining 8 bits are allocated for the exponent. The exponent may be represented in sign magnitude form. In this case, 1 bit is used for sign and 7 bits for the magnitude. The exponent will range from –127 to +127. The only disadvantage to this method is that there are two representations for 0 exponent: + 0 and – 0. To avoid this, one may use an *excess representation* or *biased format* to represent the exponent. The exponent has no sign in this format. The range 0 to 255 of an 8-bit number is divided into two parts 0 to 127 and 128 to 255 as shown below:




All bit strings from 0 to 126 are considered negative, exponent 127 represents 0, and values greater than 127 are positive. Thus the range of exponents is –127 to +128. Given an exponent string *exp*, the value of the exponent will be taken as (*exp* –127). The main advantage of this format is a unique representation for exponent zero. With the representation of binary floating point numbers explained above, the largest floating point number which can be represented is

The number of bits to be used for the mantissa is determined by the number of significant decimal digits required in computation, which is at least seven.

The main advantage of biased exponent format is unique representation for exponent 0.

$$0.1111\ldots1111 \times 2^{11111111}$$
$$|\leftarrow \text{ 23 bits } \rightarrow|$$
$$= (1 - 2^{-23}) \times 2^{255-127}$$
$$\cong 3.4 \times 10^{38}.$$

The smallest floating point number is

$$0.10000\ldots00 \times 2^{-127}$$
$$|\leftarrow \text{ 23 bits} \rightarrow|$$
$$\cong 0.293 \times 10^{-38} .$$

*Example.* Represent 52.21875 in 32-bit binary floating point format.

$52.21875 = 110100.00111 = .11010000111 \times 2^6$.
Normalized 23 bit mantissa = 0.11010000111000000000000.
As excess representation is being used for exponent, it is equal to $127 + 6 = 133$.
Thus the representation is
$52.21875 = 0.11010000111 \times 2^{133} = 0.110100000111 \times 2^{10000101}$.
The 32-bit string used to store 52.21875 in a computer will thus be 01000010111010000111000000000000.
Here, the most significant bit is the sign, the next 8 bits the exponent, and the last 23 bits the mantissa.

Institute of Electrical and Electronics Engineers standardized how floating point binary numbers would be represented in computers, how these would be rounded, how 0 and $\infty$ would be represented and how to treat exception condition such as attempt to divide by 0.

**IEEE Floating Point Standard 754-1985**

Floating point binary numbers were beginning to be used in the mid 50s. There was no uniformity in the formats used to represent floating point numbers and programs were not portable from one manufacturer's computer to another. By the mid 1980s, with the advent of personal computers, the number of bits used to store floating point numbers was standardized as 32 bits. A Standards Committee was formed by the Institute of Electrical and Electronics Engineers to standardize how floating point binary numbers would be represented in computers. In addition, the standard specified uniformity in rounding numbers, treating exception conditions such as attempt to divide by 0, and representation of 0 and infinity ($\infty$). This standard, called IEEE Standard 754 for

floating point numbers, was adopted in 1985 by all computer manufacturers. It allowed porting of programs from one computer to another without the answers being different. This standard defined floating point formats for 32-bit and 64-bit numbers. With improvement in computer technology it became feasible to use a larger number of bits for floating point numbers. After the standard was used for a number of years, many improvements were suggested. The standard was updated in 2008. The current standard is IEEE 754-2008 version. This version retained all the features of the 1985 standard and introduced new standards for 16 and128-bit numbers. It also introduced standards for representing decimal floating point numbers. We will describe these standards later in this article. In this section, we will describe IEEE 754-1985 Standard.

The current standard is IEEE 754-2008 version. This version retained all the features of the 1985 standard and introduced new standards for 16-bit and128-bit numbers. It also introduced standards for representing decimal floating point numbers.

IEEE floating point representation for binary real numbers consists of three parts. For a 32-bit (called single precision) number, they are:

1. Sign, for which 1 bit is allocated.
2. Mantissa (called *significand* in the standard) is allocated 23 bits.
3. Exponent is allocated 8 bits. As both positive and negative numbers are required for the exponent, instead of using a separate sign bit for the exponent, the standard uses a biased representation. The value of the bias is 127. Thus an exponent 0 means that –127 is stored in the exponent field. A stored value 198 means that the exponent value is (198 – 127) = 71. The exponents –127 (all 0s) and + 128 (all 1s) are reserved for representing special numbers which we discuss later.

To increase the precision of the significand, the IEEE 754 Standard uses a normalized significand which implies that its most significant bit is always 1. As this is implied, it is assumed to be on the left of the (virtual) decimal point of the significand. Thus in the IEEE Standard, the significand is 24 bits long – 23 bits of the significand which is stored in the memory and an implied 1 as the most significant 24th bit. The extra bit increases the number

| $b_0$ | $b_1\ b_2\ b_3$ ..... $b_8$ | $b_9\ b_{10}\ b_{11}$.....$b_{30}\ b_{31}$ |
|---|---|---|
| Sign | Exponent | Significand |

**Figure 2.** IEEE 754 representation of 32-bit floating point number.

of significant digits in the significand. Thus, a floating point number in the IEEE Standard is

$$(-1)^{s} \times (1.f)_2 \times 2^{\text{exponent} - 127}$$

where $s$ is the sign bit; $s = 0$ is used for positive numbers and $s = 1$ for representing negative numbers. $f$ represents the bits in the significand. Observe that the implied 1 as the most significant bit of the significand is explicitly shown for clarity.

For a 32-bit word machine, the allocation of bits for a floating point number are given in *Figure* 2. Observe that the exponent is placed before the significand (see *Box* 1).

*Example.* Represent 52.21875 in IEEE 754 – 32-bit floating point format.

$52.21875 = 110100.00111$

$= 1.1010000111 \times 2^5$.

Normalized significand = .1010000111.

Exponent: $(e - 127) = 5$ or, $e = 132$.

The bit representation in IEEE format is

| 0 | 10000100 | 10100001110000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

**Box 1.**

**Why does IEEE 754 standard use biased exponent representation and place it before the significand?**

Computers normally have separate arithmetic units to compute with integer operands and floating point operands. They are called FPU (Floating Point Unit) and IU (Integer Unit). An IU works faster and is cheaper to build. In most recent computers, there are several IUs and a smaller number of FPUs. It is preferable to use IUs whenever possible. By placing the sign bit and biased exponent bits as shown in *Figure* 2, operations such as $x <r.o.> y$, where *r.o.* is a relational operator, namely, $>, <, \geq, \leq$, and $x, y$ are reals, can be carried out by integer arithmetic units as the exponents are integers. Sorting real numbers may also be carried out by IUs as biased exponent are the most significant bits. This ensures lexicographical ordering of real numbers; the significand need not be considered as a fraction.

### Special Values in IEEE 754-1985 Standard (32 Bits)

***Representation of Zero*:** As the significand is assumed to have a hidden 1 as the most significant bit, all 0s in the significand part of the number will be taken as 1.00…0. Thus zero is represented in the IEEE Standard by all 0s for the exponent and all 0s for the significand. All 0s for the exponent is not allowed to be used for any other number. If the sign bit is 0 and all the other bits 0, the number is +0. If the sign bit is 1 and all the other bits 0, it is –0. Even though +0 and –0 have distinct representations they are assumed equal. Thus the representations of +0 and –0 are:

As the significand is assumed to have a hidden 1 as the most significant bit, all 0s in the significand part of the number will be taken as 1.00…0. Thus zero is represented in the IEEE Standard by all 0s for the exponent and all 0s for the significand.

+0

| 0 | 00000000 | 00000000000000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

–0

| 1 | 00000000 | 00000000000000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

***Representation of Infinity*:** All 1s in the exponent field is assumed to represent infinity (∞). A sign bit 0 represents +∞ and a sign bit 1 represents –∞. Thus the representations of +∞ and –∞ are:

+∞

| 0 | 11111111 | 00000000000000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

–∞

| 1 | 11111111 | 00000000000000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

All 1s in the exponent field is assumed to represent infinity (∞).

***Representation of Non Numbers*:** When an operation is performed by a computer on a pair of operands, the result may not be mathematically defined. For example, if zero is divided by zero,

When an arithmetic operation is performed on two numbers which results in an indeterminate answer, it is called NaN (Not a Number) in IEEE Standard.

the result is indeterminate. Such a result is called Not a Number (NaN) in the IEEE Standard. In fact the IEEE Standard defines two types of NaN. When the result of an operation is not defined (i.e., indeterminate) it is called a Quiet NaN (QNaN). Examples are: 0/0, $(\infty - \infty)$, $\sqrt{-1}$. Quiet NaNs are normally carried over in the computation. The other type of NaN is called a Signalling Nan (SNaN). This is used to give an error message. When an operation leads to a floating point underflow, i.e., the result of a computation is smaller than the smallest number that can be stored as a floating point number, or the result is an overflow, i.e., it is larger than the largest number that can be stored, SNaN is used. When no valid value is stored in a variable name (i.e., it is undefined) and an attempt is made to use it in an arithmetic operation, SNaN would result. QNaN is represented by 0 or 1 as the sign bit, all 1s as exponent, and a 0 as the left-most bit of the significand and at least one 1 in the rest of the significand. SNaN is represented by 0 or 1 as the sign bit, all 1s as exponent, and a 1 as the left-most bit of the significand and any string of bits for the remaining 22 bits. We give below the representations of QNaN and SNaN.

QNaN

| 0 or 1 | 11111111 | 00010000000000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

The most significant bit of the significand is 0. There is at least one 1 in the rest of the significand.

SNaN

| 0 or 1 | 11111111 | 10000000000001000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

The most significant bit of the significand is 1. Any combination of bits is allowed for the other bits of the significand.

***Largest and Smallest Positive Floating Point Numbers*:**

Largest Positive Number

| 0 | 11111110 | 11111111111111111111111 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

Significand: $1111 \ldots 1 = 1 + (1 - 2^{-23}) = 2 - 2^{-23}$.
Exponent: $(254 - 127) = 127$.
Largest Number $= (2 - 2^{-23}) \times 2^{127} \cong 3.403 \times 10^{38}$.
If the result of a computation exceeds the largest number that can be stored in the computer, then it is called an *overflow*.

Smallest Positive Number

| 0 | 00000001 | 00000000000000000000000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

Significand = 1.0.
Exponent = 1 – 127 = –126.
The smallest normalized number is $= 2^{-126} \cong 1.1755 \times 10^{-38}$.

***Subnormal Numbers*:** When all the exponent bits are 0 and the leading hidden bit of the siginificand is 0, then the floating point number is called a *subnormal number*. Thus, one logical representation of a subnormal number is

$$(-1)^s \times 0.f \times 2^{-127} \text{ (all 0s for the exponent)},$$

where $f$ has at least one 1 (otherwise the number will be taken as 0). However, the standard uses –126, i.e., bias +1 for the exponent rather than –127 which is the bias for some not so obvious reason, possibly because by using –126 instead of –127, the gap between the largest subnormal number and the smallest normalized number is smaller.

The largest subnormal number is $0.999999988 \times 2^{-126}$. It is close to the smallest normalized number $2^{-126}$.

When all the exponent bits are 0 and the leading hidden bit of the siginificand is 0, then the floating point number is called a *subnormal number.*

By using subnormal numbers, underflow which may occur in some calculations are gradual.

The smallest positive subnormal number is

| 0 | 00000000 | 00000000000000000000001 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

the value of which is $2^{-23} \times 2^{-126} = 2^{-149}$.

A result that is smaller than the smallest number that can be stored in a computer is called an *underflow*.

You may wonder why subnormal numbers are allowed in the IEEE Standard. By using subnormal numbers, underflow which may occur in some calculations are gradual. Also, the smallest number that could be represented in a machine is closer to zero. (See *Box* 2.)

**Box 2. Machine Epsilon and Dwarf**

In any of the formats for representing floating point numbers, *machine epsilon* is defined as the difference between 1 and the next larger number that can be stored in that format. For example, in 32-bit IEEE format with a 23-bit significand, the machine epsilon is $2^{-23} = 1.19 \times 10^{-7}$. This essentially tells us that the precision of decimal numbers stored in this format is 7 digits. The term precision and accuracy are not the same. Accuracy implies correctness whereas precision does not. For 64-bit representation of IEEE floating point numbers the significand length is 52 bits. Thus, machine epsilon is $2^{-52} = 2.22 \times 10^{-16}$. Therefore, decimal calculations with 64 bits give 16 digit precision. This is a conservative definition used by industry. In MATLAB, machine epsilon is as defined above. However, academics define machine epsilon as the upper bound of the relative error when numbers are rounded. Thus, for 32-bit floating point numbers, the machine epsilon is $2^{-23}/2 \cong 5.96 \times 10^{-8}$.

The machine epsilon is useful in iterative computation. When two successive iterates differ by less than |epsilon| one may assume that the iteration has converged. The IEEE Standard does not define machine epsilon.

*Tiny or dwarf* is the minimum subnormal number that can be represented using the specified floating point format. Thus, for IEEE 32-bit format, it is

| 0 | 00000000 | 00000000000000000000001 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>8 bits | Significand<br>23 bits |

whose value is: $2^{-126-23} = 2^{-149} \cong 1.4 \times 10^{-45}$. Any number less than this will signal underflow. The advantage of using subnormal numbers is that underflow is gradual as was stated earlier.

When a mathematical operation such as add, subtract, multiply, or divide is performed with two floating point numbers, the significand of the result may exceed 23 bits after the adjustment of the exponent. In such a case, there are two alternatives. One is to truncate the result, i.e., ignore all bits beyond the 32nd bit. The other is to round the result to the nearest significand. For example, if a significand is 0.110 .. 01 and the overflow bit is 1, a rounded value would be 0.111 … 10. In other words, if the first bit which is truncated is 1, add 1 to the least significant bit, else ignore the bits. This is called *rounding upwards*. If the extra bits are ignored, it is called *rounding downwards*. The IEEE Standard suggests to hardware designers to get the best possible result while performing arithmetic that are *reproducible* across different manufacturer's computers using the standard. The Standard suggests that in the equation

$$c = a \ <op> \ b,$$

where $a$ and $b$ are operands and $<op>$ an arithmetic operation, the result $c$ should be as if it was computed exactly and then rounded. This is called *correct rounding*.

In *Tables* 1 and 2, we summarise the discussions in this section.

**Table 1.** IEEE 754-85 floating point standard. We use $f$ to represent the significand, $e$ to represent the exponent, $b$ the bias of the exponent, and ± for the sign.

| 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 | 00000001to 11111110 | $a\ a$ …. $a\ a$ ($a$ = 0 or 1) |
| $- 1.f \times 2^{(e-b)}$ | 1 | 00000001to 11111110 | $a\ a$ …. $a\ a$ ($a$ = 0 or 1) |
| + ∞ | 0 | 11111111 | 000 … 00 (all 23 bits 0) |
| − ∞ | 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 |
| 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 (at least one 1) |

**Table 2.** Operations on special numbers. All NaNs are Quiet NaNs.

| 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 |
| $\infty + \infty$ | $\infty$ | $\pm \infty \times 0$ | NaN |

## IEEE 754 Floating Point 64-Bit Standard – 1985

The IEEE 754 floating point standard for 64-bit (called double precision) numbers is very similar to the 32-bit standard. The main difference is the allocation bits for the exponent and the significand. With 64 bits available to store floating point numbers, there is more freedom to increase the significant digits in the significand and increase the range by allocating more bits to the exponent. The standard allocates 1 bit for sign, 11 bits for the exponent, and 52 bits for the significand. The exponent uses a biased representation with a bias of 1023. The representation is shown below:

With 64 bits available to store floating point numbers, there is more freedom to increase the significant digits in the significand and increase the range by allocating more bits to the exponent. The standard allocates 1 bit for sign, 11 bits for the exponent, and 52 bits for the significand.

| Sign | Exponent | Significand |
|---|---|---|
| 1 bit | 11 bits | 52 bits |

A number in this standard is thus

$$(-1)^s \times (1.f) \times 2^{(\text{exponent} - 1023)},$$

where $s = \pm 1$, and $f$ is the value of the significand.

The largest positive number which can be represented in this standard is

| 0 | 11111111110 | 1111.....1111 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>11 bits | Significand<br>52 bits |

which is $= (2 - 2^{-52}) \times 2^{(2046 - 1023)} = (2 - 2^{-52}) \times 2^{1023} \cong 10^{3083}$. The smallest positive number that can be represented using 64

bits using this standard is

| 0 | 00000000001 | 0000..... 00000 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>11 bits | Significand<br>52 bits |

whose value is $2^{1-1023} = 2^{-1022}$.

The definitions of $\pm 0$, $\pm \infty$, QNaN, SNaN remain the same except that the number of bits in the exponent and significand are now 11 and 52 respectively. Subnormal numbers have a similar definition as in 32-bit standard.

IEEE 2008 Standard has introduced standards for 16-bit and 128-bit floating point numbers. It has also introduced standards for floating point decimal numbers.

**IEEE 754 Floating Point Standard – 2008**

This standard was introduced to enhance the scope of IEEE 754 floating point standard of 1985. The enhancements were introduced to meet the demands of three sets of professionals:

1. Those working in the graphics area.
2. Those who use high performance computers for numeric intensive computations.
3. Professionals carrying out financial transactions who require exact decimal computation.

This standard has not changed the format of 32- and 64-bit numbers. It has introduced floating point numbers which are 16 and 128 bits long. These are respectively called half and quadruple precision. Besides these, it has also introduced standards for floating point decimal numbers.

**The 16-Bit Standard**

The 16-bit format for real numbers was introduced for pixel storage in graphics and not for computation. (Some graphics processing units use the 16-bit format for computation.) The 16-bit format is shown in *Figure* 3.

The 16-bit format for real numbers was introduced for pixel storage in graphics and not for computation.

| $b_0$ | $b_1\ b_2\ b_3\ b_4\ b_5$ | $b_6\ b_7\ b_8\ b_9\ b_{10}$ .... $b_{14}\ b_{15}$ |
|---|---|---|
| Sign | Exponent | Significand |
| 1 bit | 5 bits | 10 bits |

**Figure 3.** Representation of 16-bit floating point numbers.

With 16-bit representation, 5 bits are used for the exponent and 10 bits for the significand. Biased exponent is used with a bias of 15. Thus the exponent range is: –14 to +15. Remember that all 0s and all 1s for the exponent are reserved to represent 0 and ∞, respectively.

The maximum positive integer that can be stored is

$$+ 1.111..1 \times 2^{15} = 1 + (1 - 2^{-11}) \times 2^{15} = (2 - 2^{-11}) \times 2^{15} \cong 65504.$$

The minimum positive number is

$$+ 1.000 \ldots 0 \times 2^{-14} = 2^{-14} = 0.61 \times 10^{-4}.$$

The minimum subnormal 16-bit floating point number is

$$2^{-24} \cong 5.96 \times 10^{-8}.$$

The definitions of ± 0, ± ∞, NaN, and SNaN follow the same ideas as in 32-bit format.

With improvements in computer technology, using 128 bits to represent floating point numbers has become feasible. This is sometimes used in numeric-intensive computation, where large rounding errors may occur.

## The 128-Bit Standard

With improvements in computer technology, using 128 bits to represent floating point numbers has become feasible. This is sometimes used in numeric-intensive computation, where large rounding errors may occur. In this standard, besides a sign bit, 14 bits are used for the exponent, and 113 bits are used for the significand. The representation is shown in *Figure* 4.

A number in this standard is

$$(-1)^s \times (1.f) \times 2^{(exponent - 16384)}.$$

| Sign | Exponent | Significand |
|---|---|---|
| 1 bit | 14 bits | 113 bits |

**Figure 4.** Representation of 128-bit floating point number.

The largest positive number which can be represented in this format is:

| 0 | 11111111111110 | 111……111 |
|---|---|---|
| Sign<br>1 bit | Exponent<br>14 bits | Significand<br>113 bits |

which equals $(1 - 2^{-113}) \times 2^{16383} = \cong 10^{4932}$.

The smallest normalized positive number which can be represented is

$$2^{1-16383} = 2^{-16382} \cong 10^{-4931}.$$

A subnormal number is represented by

$$(-1)^s \times 0.f \times 2^{-16382}.$$

The smallest positive subnormal number is

$$2^{-16382-113} = 2^{-16485}.$$

The definitions of $\pm 0$, $\pm \infty$, and QNaN, and SNaN are the same as in the 1985 standard except for the increase of bits in the exponent and significand. There are other minor changes which we will not discuss.

**The Decimal Standard**

The main motivation for introducing a decimal standard is the fact that a terminating decimal fraction need not give a terminating binary fraction. For example, $(0.1)_{10}$ = (0.00011 (0011) recurring)$_2$. If one computes 100000000 × 0.1 using binary arithmetic and rounding the non-terminating binary fraction up to the next larger number, the answer is: 10000001.490116 instead of 10000000. This is unacceptable in many situations, particularly in financial transactions. There was a demand from professionals carrying out financial transactions to introduce a standard for representing decimal floating point numbers. Decimal floating point numbers are not new. They were available in COBOL. There was, however, no standard and this resulted in non-portable programs. IEEE 754-2008 has thus introduced a standard for

The main motivation for introducing a decimal standard is the fact that a terminating decimal fraction need not give a terminating binary fraction.

There was a demand from professionals carrying out financial transactions to introduce a standard for representing decimal floating point numbers.

decimal floating point numbers to facilitate portability of programs.

### *Decimal Representation*

The idea of encoding decimal numbers using bits is simple. For example, to represent 0.1 encoded (not converted) to binary, it is written as $0.1 \times 10^0$. The exponent instead of being interpreted as a power of 2 is now interpreted as a power of 10. Both the significand and the exponent are now integers and their binary representations have a finite number of bits. One method of representing $0.1 \times 10^0$ using this idea in a 32-bit word is shown in *Figure* 5.

We have assumed an 8-bit binary exponent in biased form. In this representation, there is no hidden 1 in the significand. The significand is an integer. (Fractions are represented by using an appropriate power of 10 in the exponent field.) For example, .000753658 will be written as $0.753658 \times 10^{-3}$ and 753658, an integer, will be converted to binary and stored as the significand. The power of 10, namely –3, will be an integer in the exponent part.

The IEEE Standard does not use this simple representation. This is due to the fact that the largest decimal significand that may be represented using this method is 0.8388607.

It is preferable to obtain 7 significant digits, i.e., significand up to 0.9999999. The IEEE Standard achieves this by borrowing some bits from the exponent field and reducing the range of the exponent. In other words, instead of maximum exponent of 128, it is reduced to 96 and the bits saved are used to increase the significant digits in the significand. The coding of bits is as shown in *Figure* 6.

**Figure 5.** Representation of decimal floating point numbers.

| 0 | 01111110 | 00000000000000000000001 |
|---|---|---|
| Sign | Exponcnt | Significand |
| 1 bit | 8 bits | 23 bits |

**Figure 6.** Representation of 32-bit floating point numbers in IEEE 2008 Standard.

| Sign | Combination Bits | Exponent Continuation | Coefficient |
|---|---|---|---|
| 1 bit | 5 bits | 6 bits | 20 bits |

Observe the difference in terminology for exponent and significand fields. In this coding, one part of the five combination bits is interpreted as exponent and the other part as an extension of significand bits using an ingenious method. This will be explained later in this section.

IEEE 2008 Standard defines two formats for representing decimal floating point numbers. One of them *converts* the binary significand field to a decimal integer between 0 and $10^{p-1}$ where $p$ is the number of significant bits in the significand. The other, called densely packed decimal form, *encodes* the bits in the significand field directly to decimal. Both methods represent decimal floating point numbers as shown below.

Decimal floating point number = $(-1)^s \times f \times 10^{exp-bias}$, where $s$ is the sign bit, $f$ is a decimal fraction (significand), *exp* and *bias* are decimal numbers. The fraction need not be normalized but is usually normalized with a non-zero most significant digit. There is no hidden bit as in the binary standard. The standard defines decimal floating point formats for 32-bit, 64-bit and 128-bit numbers.

IEEE 2008 Standard defines two formats for representing decimal floating point numbers. One of them *converts* the binary significand field to a decimal integer between 0 and $10^{p-1}$ where $p$ is the number of significant bits in the significand. The other, called densely packed decimal form, *encodes* the bits in the significand field directly to decimal.

**Densely Packed Decimal Significand Representation**

Normally, when decimal numbers are coded to binary, 4 bits are required to represent each digit, as 3 bits give only 8 combinations that are not sufficient to represent 10 digits. Four bits give 16 combinations out of which we need only 10 combinations to represent decimal numbers from 0 to 9. Thus, we waste 6 combinations out of 16. For a 32-bit number, if 1 bit is used for sign, 8 bits for the exponent and 23 bits for the significand, the maximum positive value of the significand will be +0.799999. The exponent range will be 00 to 99. With a bias of 50, the maximum positive decimal number will be $0.799999 \times 10^{49}$.

The number of significand digits in the above coding is not sufficient. A clever coding scheme for decimal encoding of binary numbers was discovered by Cowlinshaw. This method encodes 10 bits as 3 decimal digits (remember that $2^{10} = 1024$ and

one can encode 000 to 999 using the available 1024 combinations of 10 bits). IEEE 754 Standard uses this coding scheme. We will now describe the standard using this coding method for 32-bit numbers.

The standard defines the following:

1. A *sign bit* 0 for + and 1 for – .
2. A *combination field* of 5 bits. It is called a combination field as one part of it is used for the exponent and another part for the significand.
3. An *exponent continuation field* that is used for the exponent.
4. A *coefficient field* that encodes strings of 10 bits as 3 digits. Thus, 20 bits are encoded as 6 digits. This is part of the significand field.

For 32-bit IEEE word, the distribution of bits was given in *Figure* 6 which is reproduced below for ready reference.

| Sign | Combination field | Exponent continuation field | Coefficient field |
|---|---|---|---|
| 1 bit | 5 bits | 6 bits | 20 bits |

The 5 bits of the combination field are used to increase the coefficient field by 1 digit. It uses the first ten 4-bit combinations of 16 possible 4-bit combinations to represent this. As there are 32 combinations of 5 bits, out of which only 10 are needed to extend the coefficient digits, the remaining 22-bit combinations are available to increase the exponent range and also encode $\infty$ and NaN. This is done by an ingenious method explained in *Table* 3.

**Table 3.** Use of combination bits in IEEE Decimal Standard.

| **Combination bits** $b_1\ b_2\ b_3\ b_4\ b_5$ | **Type** | **Most significant bits of exponent** | **Most significant digit of coefficient** |
|---|---|---|---|
| x y a b c | Finite | x y | 0 a b c |
| 1 1 a b c | Finite | a b | 1 0 0 c |
| 1 1 1 1 0 | $\infty$ | – | – |
| 1 1 1 1 1 | NaN | – | – |

| | | |
|---|---|---|
| Sign bit | 1 | 1 |
| Combination field (bits) | 5 | 5 |
| Exponent continuation field (bits) | 8 | 12 |
| Coefficient field (bits) | 50 | 110 |
| Number of bits in number | 64 | 128 |
| Significant digits | 16 | 34 |
| Exponent range (decimal) | 768 | 12288 |
| Exponent bias (decimal) | 384 | 6144 |

**Table 4.** Decimal representation for 64-bit and 128-bit numbers.

Thus, the most significant 2 bits of the exponent are constrained to take on values 00, 01, 10. Using the 6 bits of exponent combination bits, we get an exponent range of $3 \times 2^6 = 192$. The exponent bias is 96. (Remember that the exponent is an integer that can be exactly represented in binary.) With the addition of 1 significant digit to the 6 digits of the coefficient bits, the decimal representation of 32-bit numbers has 7 significant digits. The largest number that can be stored is $0.9999999 \times 10^{96}$. A similar method is used to represent 64- and 128-bit floating point numbers. This is shown in *Table* 4.

In the IEEE 2008 Decimal Standard, the combination bits along with exponent bits are used to represent NaN, signaling NaN and $\pm \infty$ as shown in *Table* 5.

| **Quantity** | **Sign bit** | **Combination bits** |
|---|---|---|
| | $b_0$ | $b_1\ b_2\ b_3\ b_4\ b_5\ b_6$ |
| SNaN | 0 or 1 | 1 1 1 1 1 1 |
| QNaN | 0 or 1 | 1 1 1 1 1 0 |
| $+\infty$ | 0 | 1 1 1 1 0 x |
| $-\infty$ | 1 | 1 1 1 1 0 x |
| + 0 | 0 | 0 0 0 0 0 x |
| – 0 | 1 | 0 0 0 0 0 x |
| | | (x is 0 or 1) |

**Table 5.** Representation of NaNs, $\pm \infty$, and $\pm$ 0.

### Binary Integer Representation of Significand

In this method, as we stated earlier, if there are $p$ bits in the extended coefficient field constituting the significand, they are converted as a decimal integer. This representation also uses the bits of the combination field to increase the number of significant digits. The idea is similar and we will not repeat it. The representation of NaNs and $\pm\infty$ is the same as in the densely packed format. The major advantage of this representation is the simplicity of performing arithmetic using hardware arithmetic units designed to perform arithmetic using binary numbers. The disadvantage is the time taken for conversion from integer to decimal. The densely packed decimal format allows simple encoding of binary to decimal but needs a specially designed decimal arithmetic unit.

### Acknowledgements

I thank Prof. P C P Bhatt, Prof. C R Muthukrishnan, and Prof. S K Nandy for reading the first draft of this article and suggesting improvements.

### Suggested Reading

[1] David Goldberg, What every computer scientist should know about floating point arithmetic, *ACM Computing Surveys,* Vol.28, No.1, pp5–48, March 1991.

[2] V Rajaraman, *Computer Oriented Numerical Methods*, 3rd Edition, PHI Learning, New Delhi, pp.15–32, 2013.

[3] Marc Cowlinshaw, Densely packed decimal encoding, *IEEE Proceedings – Computer and Digital Techniques,* Vol.149, No.3, pp.102-104, May 2002.

[4] Steve Hollasch, IEEE Standard 754 for floating point numbers, steve.hollasch.net/cgindex/coding/ieeefloat.html

[5] Decimal Floating Point, Wikipedia, en.wikipedia.org/wiki/Decimal_floating_point

*Address for Correspondence*
V Rajaraman
Supercomputer Education and Research Centre
Indian Institute of Science
Bengaluru 560 012, India.
Email: rajaram@serc.iisc.in