CS/ECE 5381/7381 Computer Architecture Spring 2023

Dr. Manikas

Computer Science

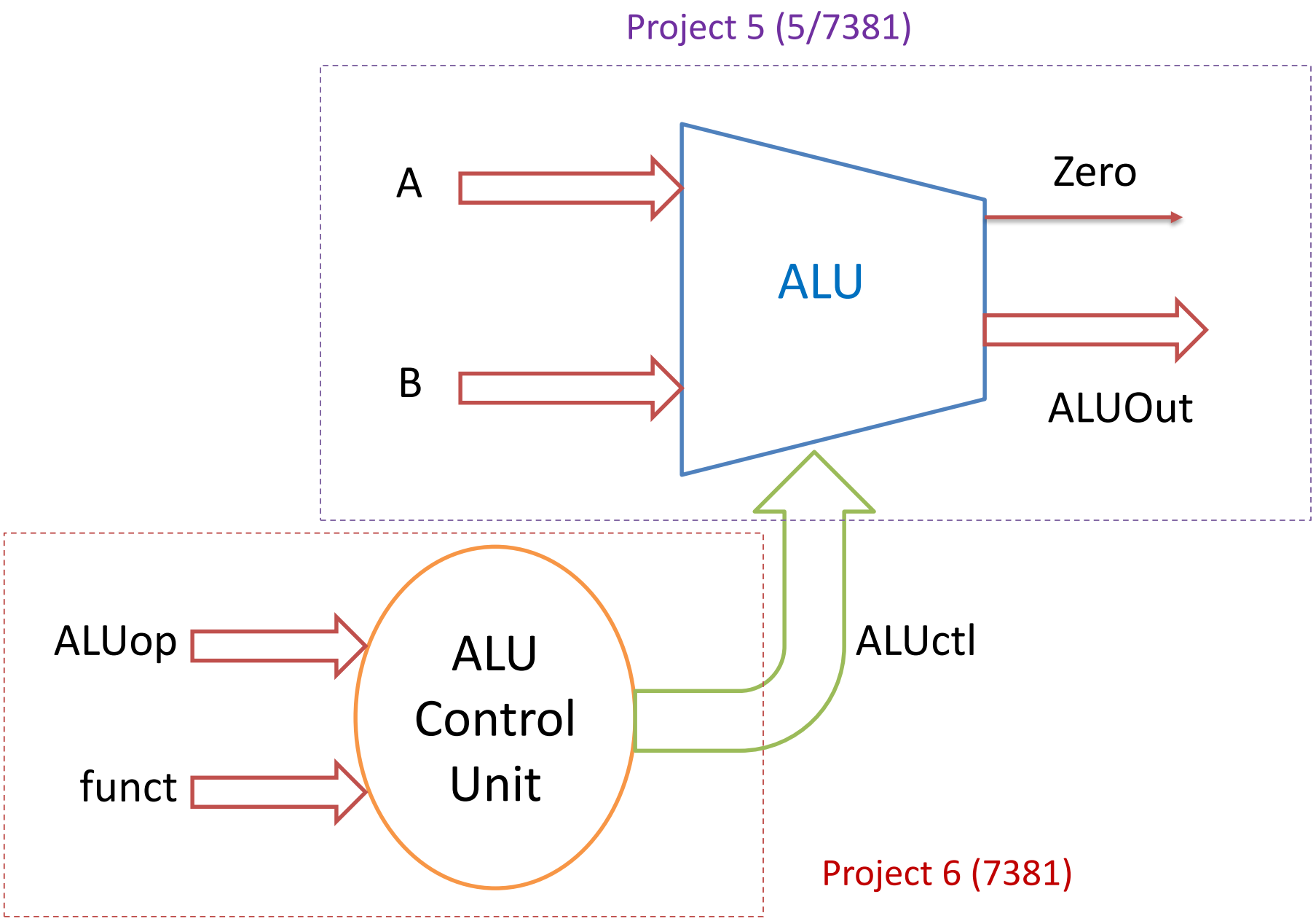
Lecture 17: Apr. 4, 2023

Project 5

- Due TODAY, Apr. 4 (11:59 pm)
- Xcelium tool
- Assignment:
  - Modify Verilog code for basic MIPS ALU
    - Add arithmetic and logic functions

Project 6 (7381 only)

- Due Thur., Apr. 6 (11:59 pm)
- For CS/ECE 7381 students ONLY
- Additional Verilog programming assignment using Xcelium tool
  - Develop control unit for ALU


Project 5 (5/7381)
A
B
ALU
Zero
ALUOut
ALUop
funct
ALU
Control
Unit
ALUctl
Project 6 (7381)

Memory Design Hierarchy

(Chapter 2, Hennessy and Patterson)

Note: some course slides adopted from publisher-provided material

Outline

- 2.1 Introduction
- 2.2 Memory Technology and Optimizations
- 2.3 Ten Advanced Optimizations of Cache Performance
- 2.4 Virtual Memory and Machines

10 Advanced Cache Optimizations

- Reducing hit time

1. Small and simple caches
2. Way prediction

- Increasing cache bandwidth

3. Pipelined caches
4. Multibanked caches
5. Nonblocking caches

- Reducing Miss Penalty

6. Critical word first
7. Merging write buffers

10 Advanced Cache Optimizations

- Reducing Miss Rate

8. Compiler optimizations

8. Reducing Misses by Compiler Optimizations

- *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
- *Loop Interchange*: change nesting of loops to access data in order stored in memory
- *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
- *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

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

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

Reducing conflicts between val and key; improve spatial locality

Loop Interchange Example

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

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

Before Loop Interchange

Direct-mapped cache

	0	1	2	3	4			4999
0								
1								
2								
3								
4								
5								
99								

Repeated cache misses – going through same row of main memory

After Loop Interchange

Direct-mapped cache

	0	1	2	3	4			4999
0								
1								
2								
3								
4								
5								
99								

More cache hits likely as going through different rows per loop

Loop Fusion Example - Before

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

for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     d[i][j] = a[i][j] + c[i][j];
```

Loop Fusion Example - After

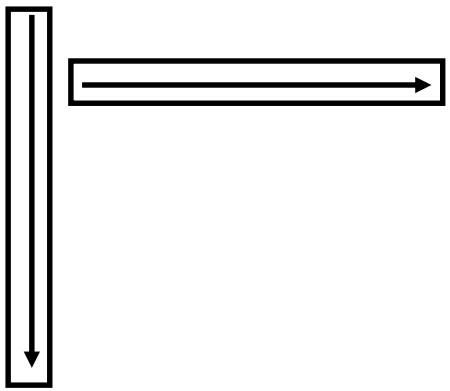
```
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {  a[i][j] = 1/b[i][j] * c[i][j];
     d[i][j] = a[i][j] + c[i][j];}
```

improve the spatial locality.

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

Blocking Example - before

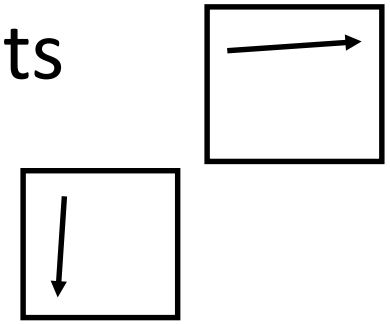
```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {r = 0;
     for (k = 0; k < N; k = k+1){
       r = r + y[i][k]*z[k][j];};
     x[i][j] = r;
    };
```



- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]

Blocking Example – before (cont)

- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

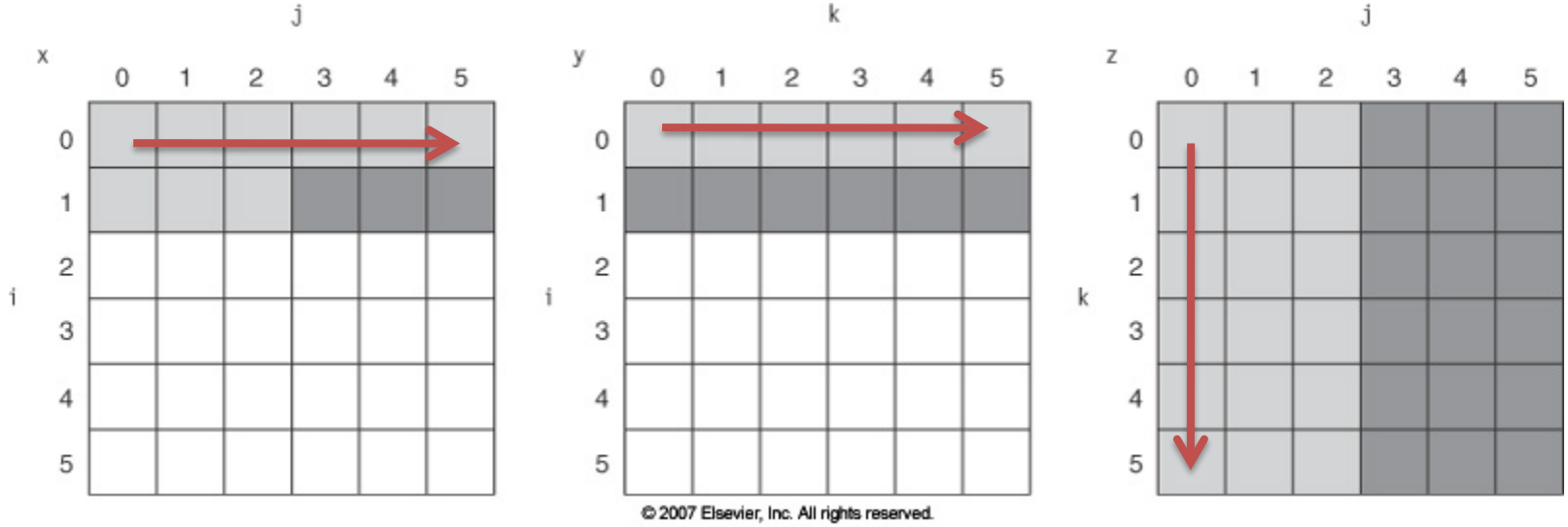


Blocking Example

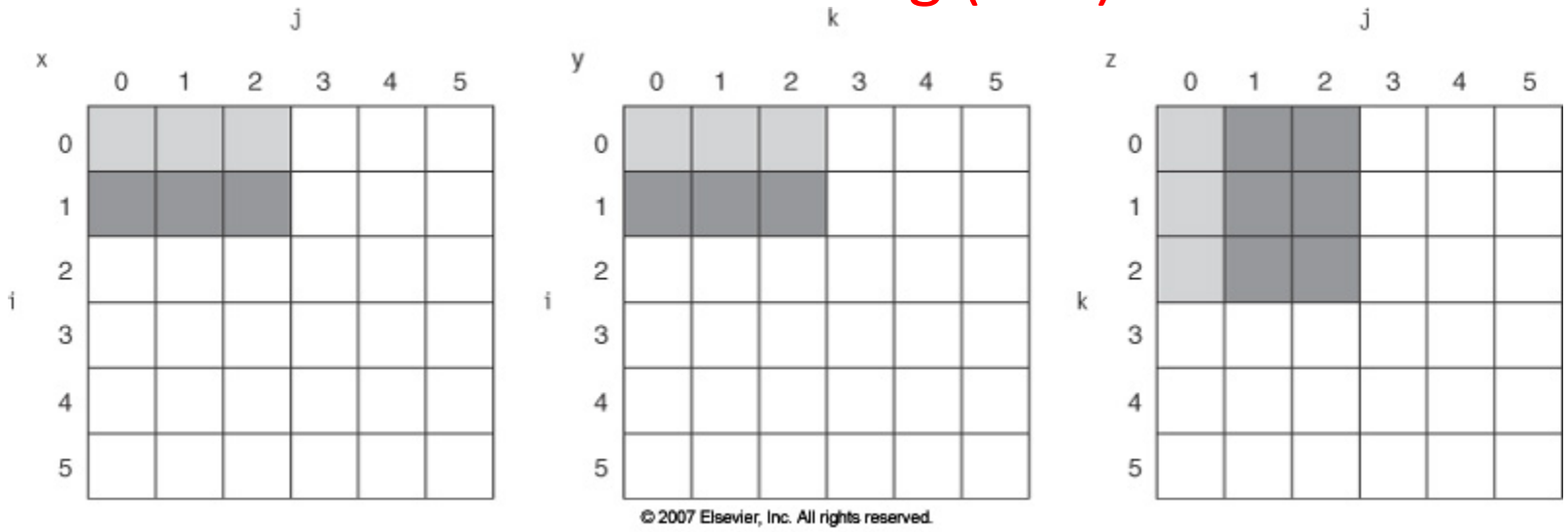
```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1)
     {r = 0;
      for (k = kk; k < min(kk+B-1,N); k = k+1) {
        r = r + y[i][k]*z[k][j];};
      x[i][j] = x[i][j] + r;
     };
```

- B called *Blocking Factor*
- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$

Before blocking



After blocking (B=3)



10 Advanced Cache Optimizations

- Reducing miss penalty or miss rate via parallelism

9. Hardware prefetching
10. Software (compiler) prefetching

verb | prēˈfeCH | *[with object]*
transfer (data) from main memory to temporary storage in readiness for later use: *this model prefetches instructions before they need to be executed*.
noun | ˈprēˌfeCH |
a process or instance of prefetching data.

9. Reducing Misses by Hardware Prefetching of Instructions & Data

- Prefetching relies on having extra memory bandwidth that can be used without penalty
- Instruction Prefetching
  - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
  - Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer

Array Storage in Memory

- We may view items as arrays (2D)
- But these are stored in main memory as 1D items
- How to translate? Two possibilities:
  - Row-Major Mapping
    - Used in C/C++, Python
  - Column-Major Mapping
    - Used in Fortran, MATLAB

Row-Major Mapping

- Example 3 x 4 array:

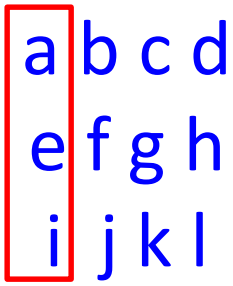
a b c d

e f g h

i j k l

- Convert into 1D array y by collecting elements by *rows*
- Within a row, elements are collected from left to right
- Rows are collected from top to bottom
- We get {a, b, c, d, e, f, g, h, i, j, k, l}

Column-Major Mapping



- Convert into 1D array y by collecting elements by *columns*
- Within a column, elements are collected from top to bottom
- Columns are collected from left to right
- We get {a, e, i, b, f, j, c, g, k, d, h, l}

Example 2.3-2

Assume that we have the following matrix stored in main memory in **column major order**:

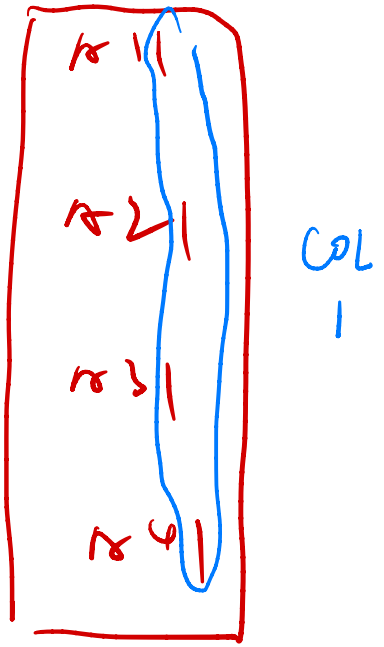
$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix}$$

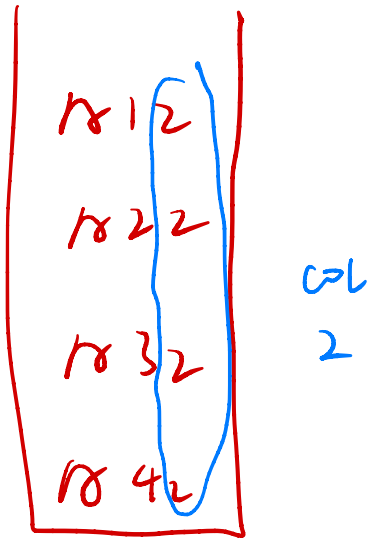
How are the matrix elements arranged in main memory?

2D → 1D

COLUMN MAJOR ORDER

MAIN MEMORY





FORTRAN

MATLAB

Example 2.3-3

Next, assume that our 4x4 matrix contains double-precision numbers.

Assuming the IEEE floating-point standard (as used by MIPS), how many bytes per number?

DOUBLE-PRECISION

IEEE FLOATING-POINT STANDARD

$\Rightarrow$ 64 BITS & 8 BITS / BYTE

Remember this is ok

OF BYTES PER NUMBER

$$= \frac{64 \text{ BITS}}{8 \text{ BITS/BYTE}} = \boxed{8 \text{ BYTES}}$$

e.g. ENTRY #11 TAKES 8 BYTES

Example 2.3-4

Next, assume that we have a fully-associative cache that views our main memory as 64-byte blocks.

How are the matrix elements stored in main memory blocks?

EACH MATRIX ELEMENT

TAKES 8 BYTES

CACHE BLOCK SIZE = 64 BYTES

$$\frac{64 \text{ BYTES / BLOCK}}{8 \text{ BYTE / ELEMENTS}} = 8 \frac{\text{ELEMENTS}}{\text{BLOCK}}$$

HOW STORED IN MAIN MEMORY?

ORIGINAL MATRIX 4×4 = 16 ELEMENTS

16 ELEMENTS / 8 ELEMENTS/BLOCK = 2 BLOCKS

STORED IN COLUMN MAJOR ORDER

BLOCK	→							
0	a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}
1	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}

Example 2.3-5

Next, assume that we now have a 256 x 256 matrix. How are the matrix elements arranged in the memory blocks?

(Assume double-precision numbers and 64-byte blocks as before)

256 x 256 ARRAY (MATRIX)

= 65536 ELEMENTS

$(2^8 \cdot 2^8 = 2^{16} = 2^6 2^{10} = 64 \times 1024)$

NUMBER OF BLOCKS IN MAIN MEMORY?

RECALL: EACH BLOCK HAS 8 ELEMENTS

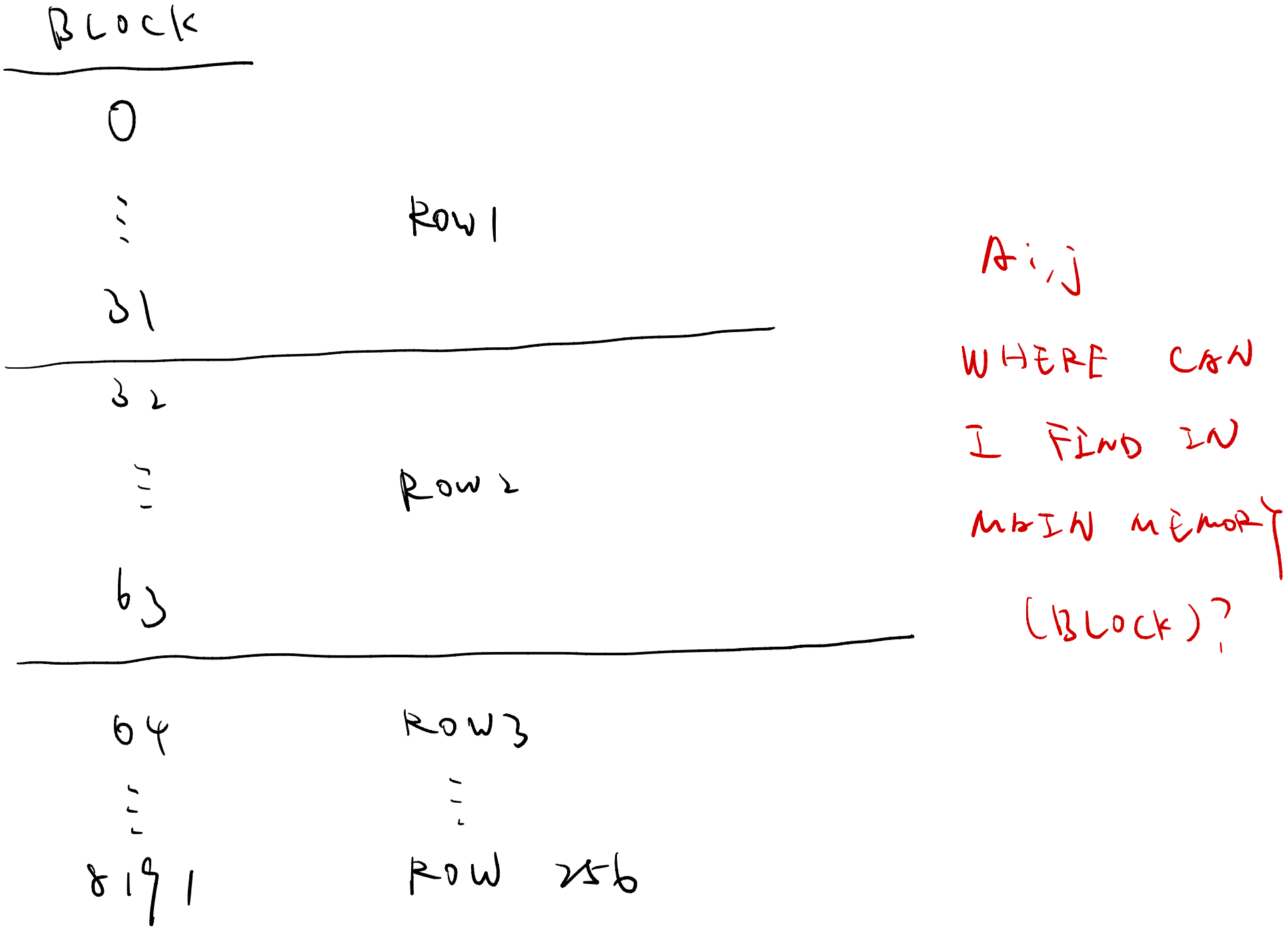
$$\frac{65536}{8} = \frac{2^{16}}{2^3} = 2^{13} = 8192 \text{ BLOCKS}$$

How many blocks per matrix row?

256 elements per row

8 elements per block

$$\frac{256}{8} = \frac{2^8}{2^3} = 2^5 = \underline{32} \text{ blocks per matrix row}$$


BLOCK
0
31
ROW 1
32
63
ROW 2
64
8191
ROW 3
ROW 256
A i, j
WHERE CAN I FIND IN MAIN MEMORY (BLOCK)?

Example 2.3-6

Given a matrix element A(row, column) for our 256x256 matrix, how do we identify the block in main memory where it is stored?

GIVEN MATRIX ELEMENT

A (r, c) r = ROW c = COLOMN

HOW TO FIND BLOCK NUMBER

WHERE THIS ELEMENT IS STORED ?

BLOCK NUMBER = ROW FACTOR + COLUMN FACTOR

ROW FACTOR? FOR ROW r

- WE HAVE 32 BLOCKS PER ROW

- BLOCK NUMBERING STARTS AT ZERO

BLOCK 0 STARTS ROW 1

BLOCK 32 STARTS ROW 2

BLOCK 64 STARTS ROW 3

ROW FACTOR = 32 (r − 1)

FIRST BLOCK OF ROW

COLOMN FACTOR?

ROW FACTOR GIVES US FIRST BLOCK OF

ELEMENT ROW.

ROW 1 — 8 ELEMENTS/BLOCK

BLOCK			
0	(1,1)	.....	(1,8)
1	(1,9)		(1,16)
⋮	⋮		⋮
32	(1,249)	.....	(1,256)

FOR C = 1 TO 8 (C = COLUMN)

BLOCK 0

BLOCK	COLUMN RANGE
0	1 - 8
1	9 - 16
2	17 - 24

Multiples of 8, so DIVIDE BY 8?

COLUMN		DIVIDE BY 8	ROUND DOWN (FLOOR)	$\lfloor \frac{C-1}{8} \rfloor$
1	C-1=0	0.125	0	0
2		0.25	0	0
3		0.375	0	0
4		0.5	0	0
5		0.625	0	0
6		0.75	0	0
7		0.875	0	0
8		1	1	0

COLUMN FACTOR $= \left\lfloor \frac{(c-1)}{8} \right\rfloor$

SO BLOCK NUMBER FOR MATRIX ELEMENT A (r, c)

$= \text{ROW FACTOR} + \text{COLUMN FACTOR}$

$= 32(r-1) + \left\lfloor \frac{(c-1)}{8} \right\rfloor$

TEXT : A (1, 16) r = 1 c = 16

$$\text{BLOCK NUMBER} = 32(1-1) + \left\lfloor \frac{16-1}{8} \right\rfloor$$

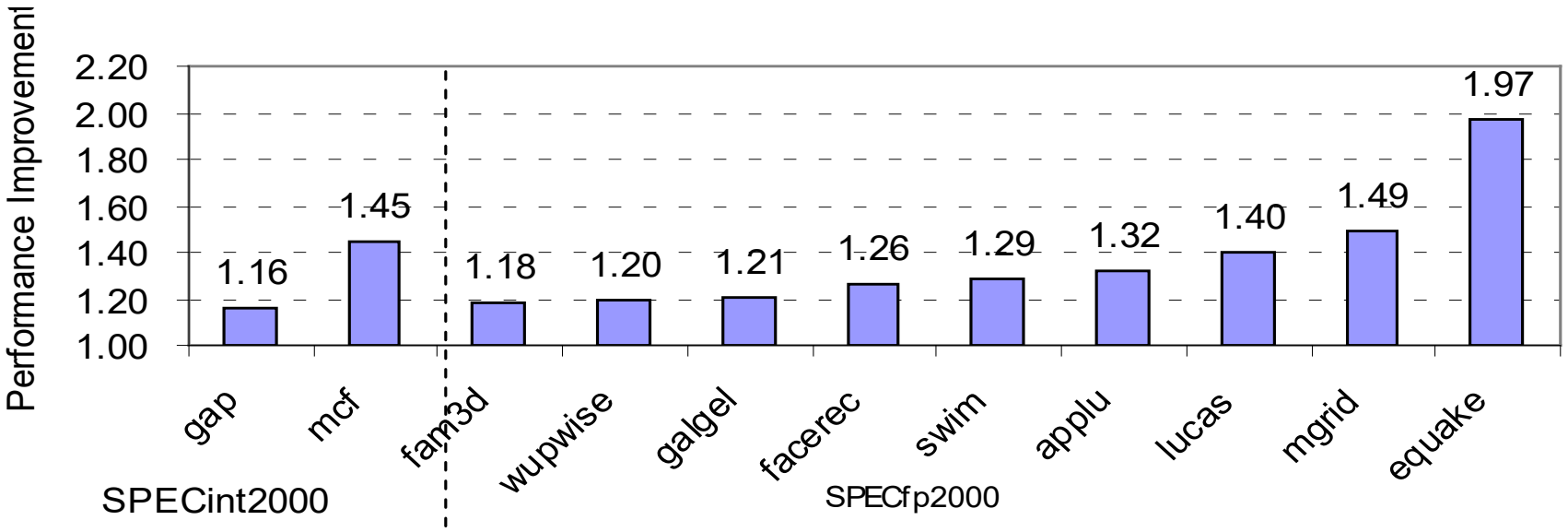
$$= 0 + 1$$

$$= \boxed{11}$$

9. Reducing Misses by Hardware Prefetching of Instructions & Data (cont)

- Data Prefetching
  - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
  - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes

9. Reducing Misses by Hardware Prefetching of Instructions & Data (cont)



10. Reducing Misses by Software Prefetching Data

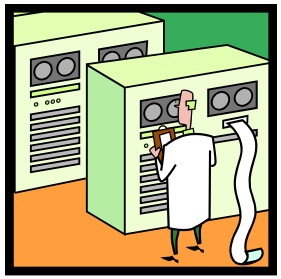
- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

Outline

- 2.1 Introduction
- 2.2 Memory Technology and Optimizations
- 2.3 Ten Advanced Optimizations of Cache Performance
- 2.4 Virtual Memory and Machines

Introduction to Virtual Machines

- VMs developed in late 1960s
  - Remained important in mainframe computing over the years
  - Largely ignored in single user computers of 1980s and 1990s
  - But used in *networked* computers



Introduction to Virtual Machines (cont)

- Recently regained popularity due to
  - increasing importance of isolation and security in modern systems,
  - failures in security and reliability of standard operating systems,
  - sharing of a single computer among many unrelated users,
  - and the dramatic increases in raw speed of processors, which makes the overhead of VMs more acceptable

What is a Virtual Machine (VM)?

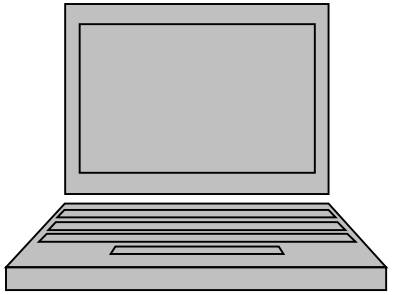
- Broadest definition includes all emulation methods that provide a standard software interface, such as the Java VM
- "(Operating) System Virtual Machines" provide a complete system level environment at binary ISA
  - Here assume ISAs always match the native hardware ISA

What is a Virtual Machine (VM)?

- Present illusion that VM users have entire computer to themselves, including a copy of OS
- Single computer runs multiple VMs, and can support a multiple, different OSes
  - On conventional platform, single OS "owns" all HW resources
  - With a VM, multiple OSes all share HW resources
- Underlying HW platform is called the host, and its resources are shared among the guest VMs

VM Examples?

- We have used virtual machines recently in this course
  - X-windows emulators
    - Xming (Windows)
    - Mac OS X
  - Your PC acts like a Unix machine
    - It is the “guest” VM
    - Lyle Linux Server is the “host” VM



Data-Level Parallelism

(Chapter 4, Hennessy and Patterson)

Note: some course slides adopted from publisher-provided material

Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.4 Graphics Processing Units (GPU's)

Introduction

- SIMD = Single Instruction stream, Multiple Data streams
  - Single set of instructions accesses multiple sets of data
- MIMD = Multiple Instruction streams, Multiple Data streams

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.4 Graphics Processing Units (GPU's)