

# Virtual Memory

Hung-Wei Tseng

# Summary of Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
  - Matrix transpose
  - Column-store vs. row-store
- Blocking/tiling — capacity miss, conflict miss
  - Matrix multiplications
- Loop fission — conflict miss — when \$ has limited way associativity
- Loop fusion — capacity miss — when \$ has enough way associativity
- Loop interchange — conflict/capacity miss

# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?

- ① Non-blocking/pipelined/multibanked cache
- ② Critical word first and early restart
- ③ Prefetching
- ④ Write buffer

A. 0

B. 1

C. 2

D. 3

E. 4



# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
    - ① Non-blocking/pipelined/multibanked cache **Miss penalty/Bandwidth**
    - ② Critical word first and early restart **Miss penalty**
    - ③ Prefetching **Miss rate (compulsory)**
    - ④ Write buffer **Miss penalty**
- A. 0
- B. 1**
- C. 2
- D. 3
- E. 4

# Summary of Optimizations

- Hardware
  - Victim/Miss cache — conflict misses
  - Prefetch & Stream Buffer — compulsory miss
  - Write buffer — miss penalty
  - Bank/pipeline — miss penalty
  - Critical word first and early restart — miss penalty
- Software/Programmer
  - Data layout — capacity miss, conflict miss, compulsory miss
    - Matrix transpose
    - Column-store vs. row-store
  - Blocking/tiling — capacity miss, conflict miss
    - Matrix multiplications
  - Loop fission — conflict miss — when \$ has limited way associativity
  - Loop fusion — capacity miss — when \$ has enough way associativity
  - Loop interchange — conflict/capacity miss

**Most hardware optimizations cannot help improve cache misses**

# Outline

- Virtual memory
- Architectural support for virtual memory

**Let's take a look of another aspect  
of memory systems**

# Let's dig into this code

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

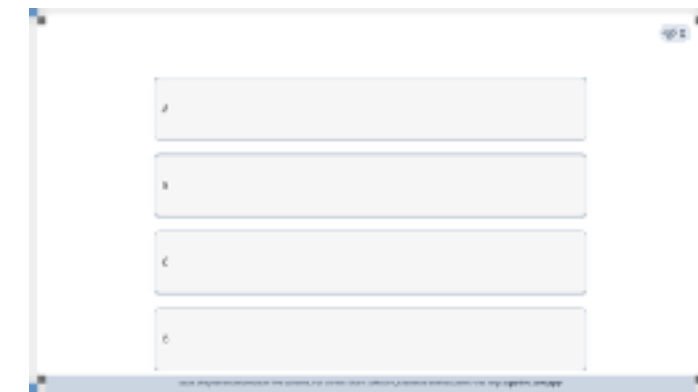




# What will happen?

- If we execute the code on the right-hand side code on a machine with only 4 GB of physical memory installed and the dim is "32768" (requires  $32768 \times 32768 \times 8$  bytes ~ 8 GB memory at least), What will happen?
  - A. The program will crash in one of the malloc function call
  - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
  - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
  - D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```



# What will happen?

- If we execute the code on the right-hand side code on a machine with only 32 GB of physical memory installed and the dim is "70000" (requires  $70000 \times 70000 \times 8$  bytes ~ 37 GB memory at least), What will happen?
  - A. The program will crash in one of the malloc function call
  - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
  - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
  - D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

```
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
```

```
//#define dim 32768
```

```
//#define dim 49152
```

```
int main(int argc, char *argv[])
{
```

```
    int i,j;
```

```
    double **a;
```

```
    double sum=0, average;
```

```
    int dim=32768;
```

```
    if(argc < 2)
```

```
    {
        fprintf(stderr, "Usage: %s dimension\n", argv[0]);
        exit(1);
    }
```

```
    dim = atoi(argv[1]);
```

```
    a = (double **)malloc(sizeof(double *)*dim);
```

File memory\_allocation.c not changed so no update needed

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual\_memory] -bunny- ./memory\_allocation

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual\_memory] -bunny-



# Let's dig into this code

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    // Create processes
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    // Generate rand seed
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    return 0;
}
```

# Consider the following code ...

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?

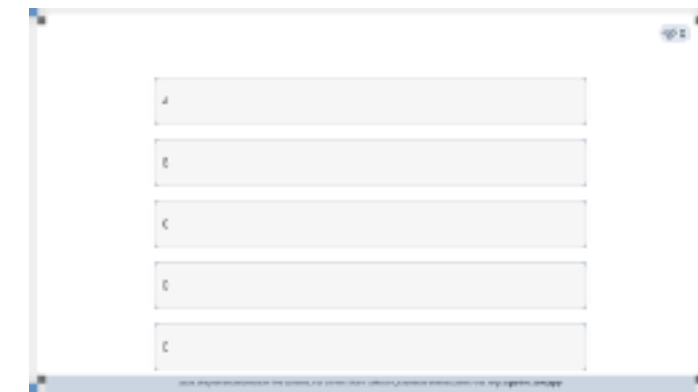
- ① The printed "address of a" is the same for every running instances
- ② The printed "address of a" is different for each instance
- ③ All running instances will print the same value of a
- ④ Some instances will print the same value of a
- ⑤ Each instance will print a different value of a

- A. (1) & (3)
- B. (1) & (4)
- C. (1) & (5)
- D. (2) & (3)
- E. (2) & (4)

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i < number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address  
of a is %p\n", getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address  
of a is %p\n", getpid(), a, &a);
    return 0;
}
```







# Demo revisited

```
sleep(10);  
fprintf(stderr, "\nProcess %d: Value of a is %1f and address of a is %p\n", (int) getpid(), a, &a);  
return 0;  
}
```

File virtualization.c not changed so no update needed

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual\_memory] -bunny- make

gcc -O3 virtualization.c -o virtualization

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual\_memory] -bunny- ./virtualization 4

Process 19719: Value of a is 1671139616.000000 and address of a is 0x104967050

Process 19720: Value of a is 1671156423.000000 and address of a is 0x104967050

Process 19718: Value of a is 1671122809.000000 and address of a is 0x104967050

Process 19721: Value of a is 1671173230.000000 and address of a is 0x104967050

Different values

Process 19719: Value of a is 1671139616.000000 and address of a is 0x104967050

Process 19721: Value of a is 1671173230.000000 and address of a is 0x104967050

Process 19720: Value of a is 1671156423.000000 and address of a is 0x104967050

Process 19718: Value of a is 1671122809.000000 and address of a is 0x104967050

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual\_memory] -bunny-

Different values are  
preserved

The same memory  
address!



# Consider the following code ...

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?
  - ① The printed "address of a" is the same for every running instances
  - ② The printed "address of a" is different for each instance
  - ③ All running instances will print the same value of a
  - ④ Some instances will print the same value of a
  - ⑤ Each instance will print a different value of a

A. (1) & (3)  
B. (1) & (4)  
**C. (1) & (5)**  
D. (2) & (3)  
E. (2) & (4)

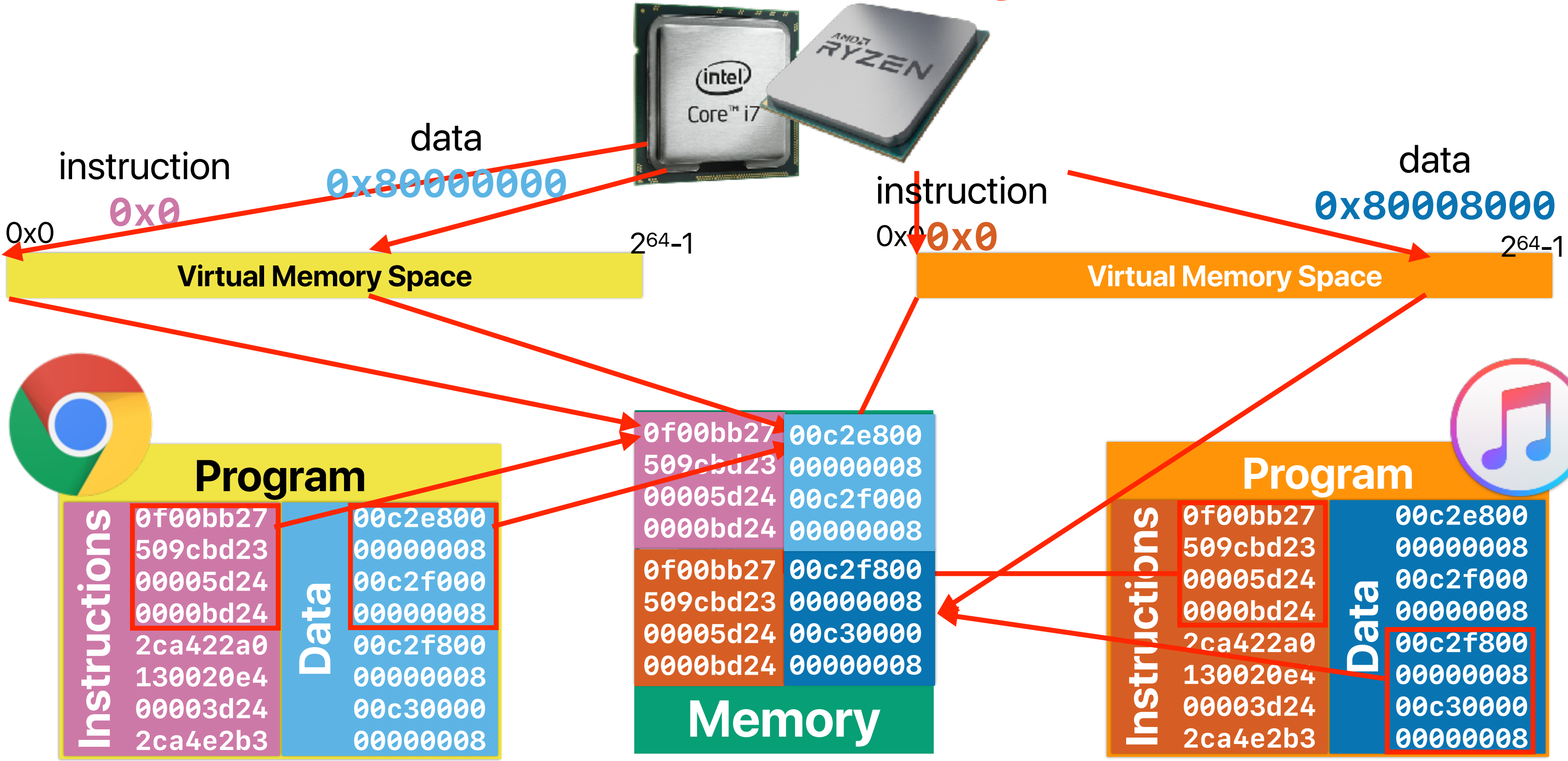
```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address  
of a is %p\n", getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address  
of a is %p\n", getpid(), a, &a);
    return 0;
}
```


# Virtual Memory

# Virtual memory



# Virtual memory

- An **abstraction** of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into "**pages**"



The diagram shows a green rounded rectangle representing a 'Processor Core'. Inside the core, the text 'Processor Core' is written in large white font. Below this, there is an orange rounded rectangle representing a 'Register'. The word 'Registers' is written in white text inside the orange rectangle. To the right of the orange rectangle, a purple line extends horizontally, ending in a large purple number '1'.

# Processor Core

Registers

1

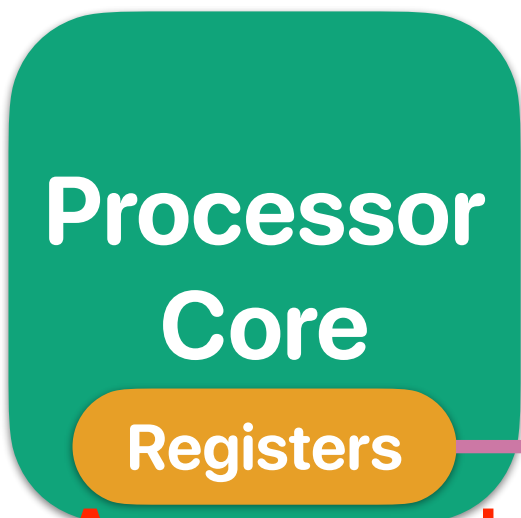
# Registers

load 0x0009

# Page table

# Main memory (DRAM)

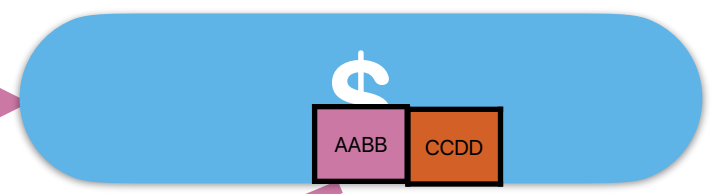




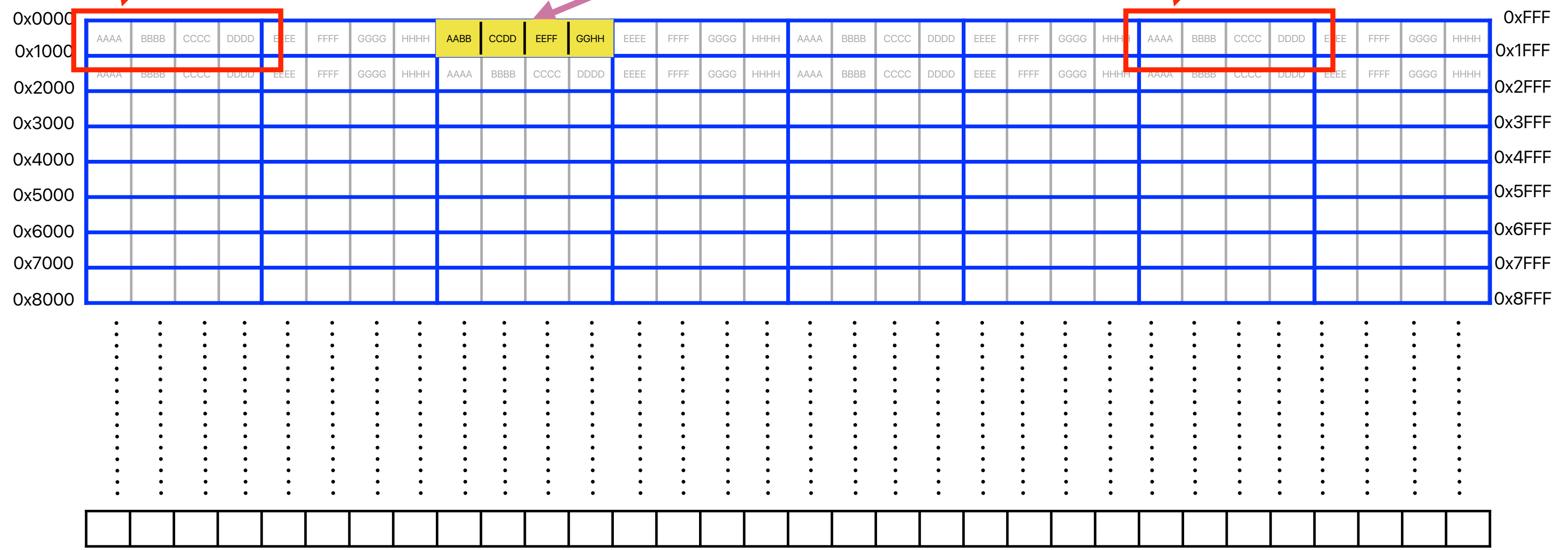
Recap: To capture "spatial" locality, \$ fetch a "block"

lw 0x0024

Assume each block is 16 bytes



"Logically" partition memory space into "blocks"



# Why Virtual memory?

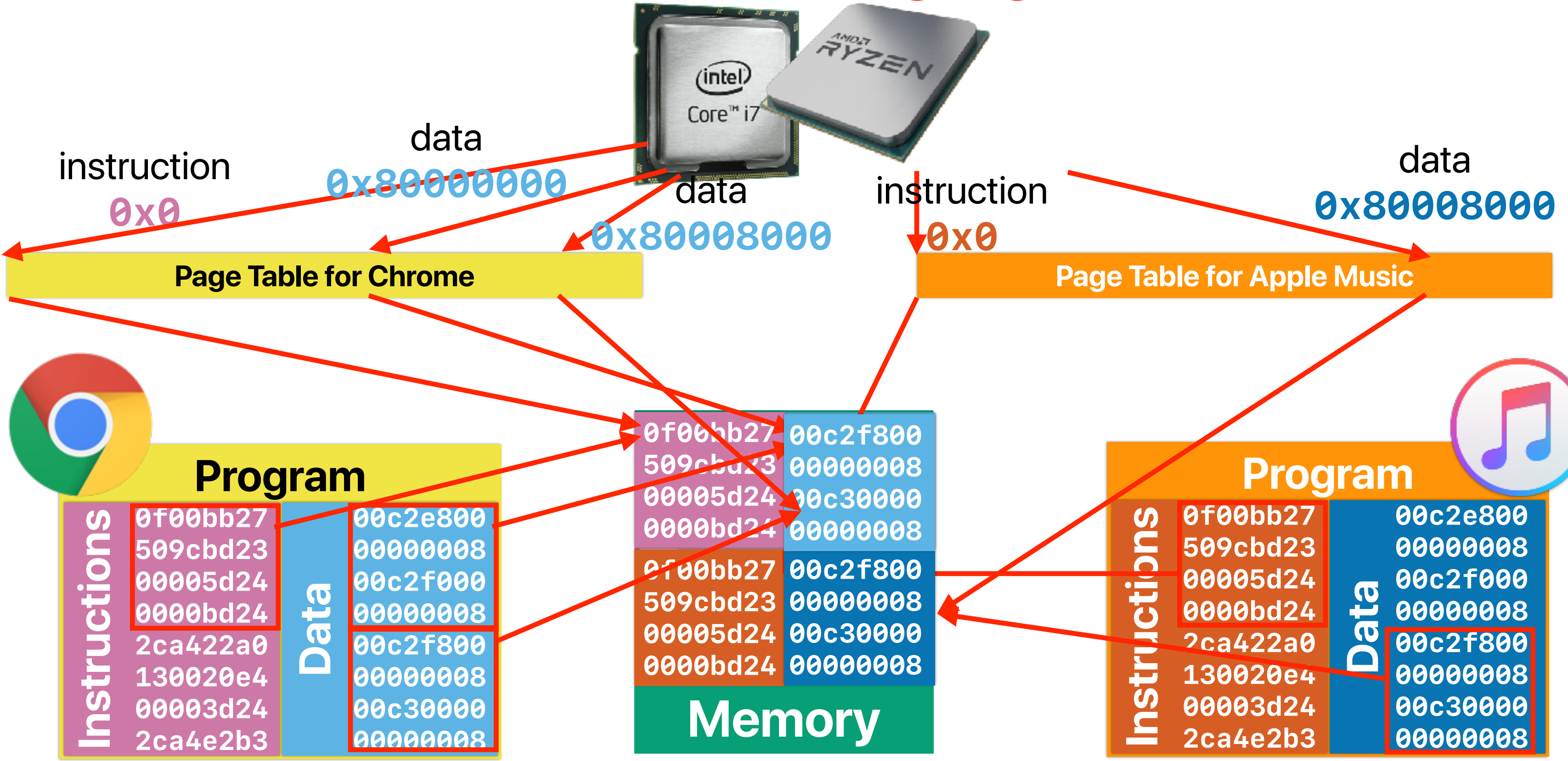
- Allowing multiple applications to share physical main memory
  - Memory protection/isolation among programs/processes is automatically achieved
- Allowing applications to work even the installed physical memory or available physical memory is smaller than the working set of the application
  - Programmer does not need to worry about the physical memory capacity of different machines — make compiled program compatible
  - Multiple programs can work concurrently even though their total memory demand is larger than the installed physical memory

# Demand paging

- Treating physical main memory as a “cache” of virtual memory
- The block size is the “page size”
- The page table is the “tag array”
- It’s a “fully-associate” cache — a virtual page can go anywhere in the physical main memory

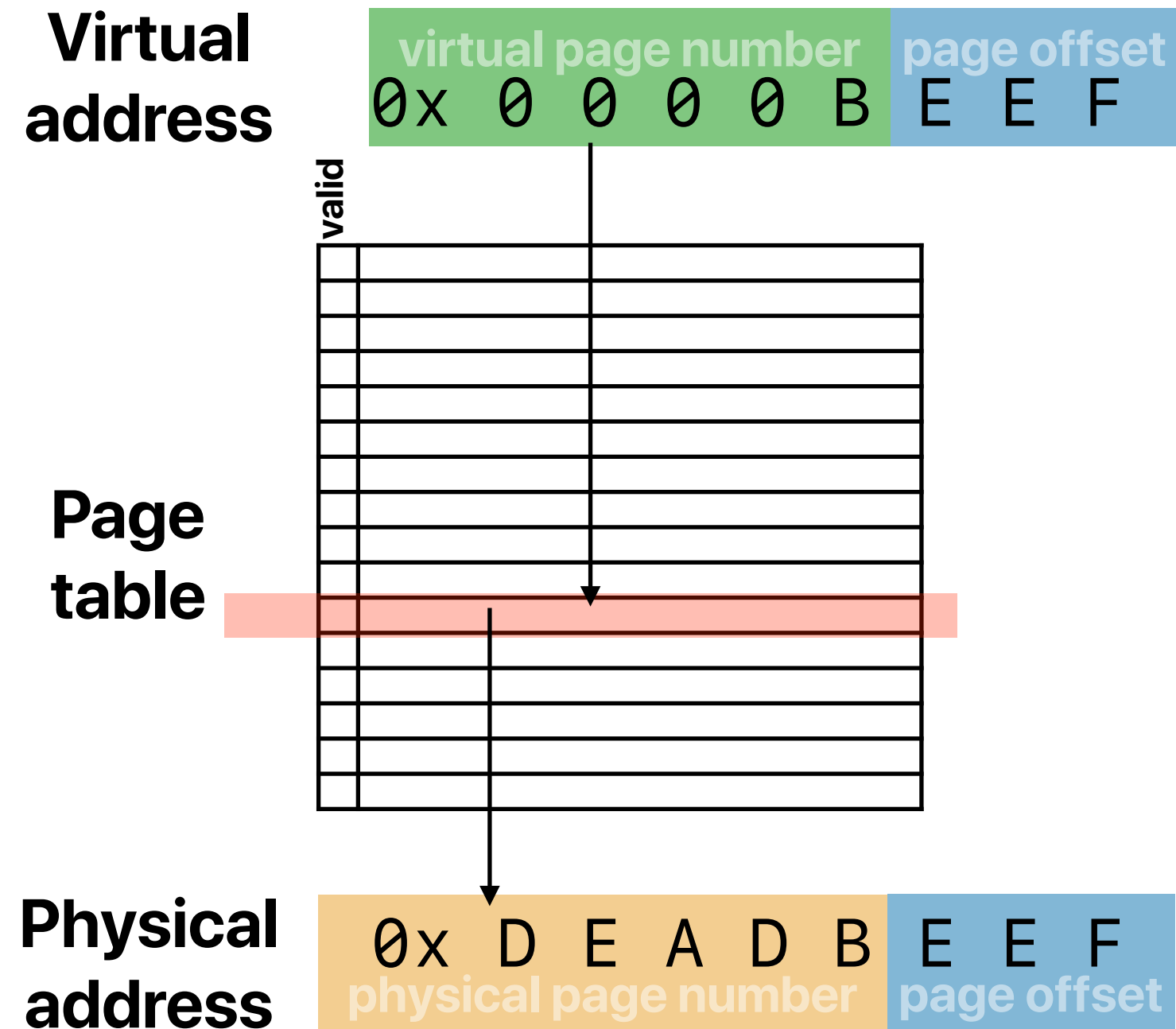


# Demand paging



# Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses
- Virtual address space is organized into "pages"
- The system references the **page table** to translate addresses
  - Each process has its own page table
  - The page table content is maintained by OS





# Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  - A. MB —  $2^{20}$  Bytes
  - B. GB —  $2^{30}$  Bytes
  - C. TB —  $2^{40}$  Bytes
  - D. PB —  $2^{50}$  Bytes
  - E. EB —  $2^{60}$  Bytes

A screenshot of a poll interface. It shows five empty rectangular input boxes stacked vertically, each preceded by a small letter (A, B, C, D, E) in a light blue font. The interface is part of a larger application window with a light blue header and footer.

# Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?

A. MB —  $2^{20}$  Bytes

B. GB —  $2^{30}$  Bytes

C. TB —  $2^{40}$  Bytes

**D. PB —  $2^{50}$  Bytes**

E. EB —  $2^{60}$  Bytes

$$\frac{2^{64} \text{ Bytes}}{4 \text{ KB}} \times 8 \text{ Bytes} = 2^{55} \text{ Bytes} = 32 \text{ PB}$$

**If you still don't know why — you need to take CS202**

# Conventional page table

0x0

0xFFFFFFFFFFFFFFFF

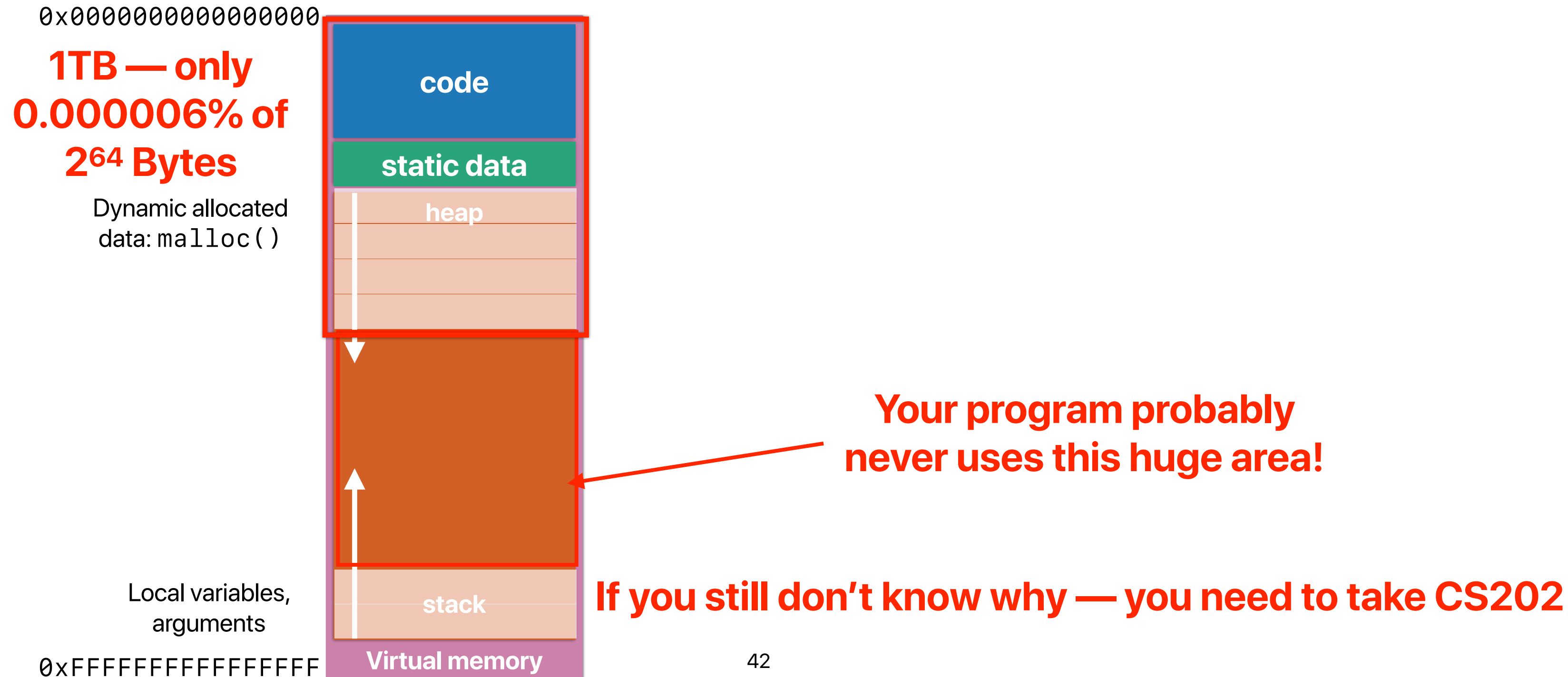
Virtual Address Space

- must be consecutive in the physical memory
- need a big segment! — difficult to find a spot
- simply too big to fit in memory if address space is large!

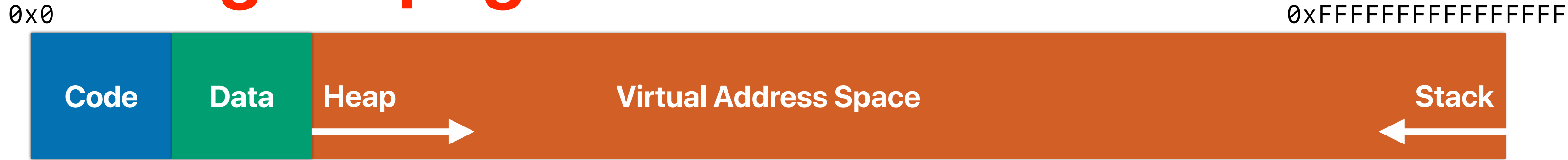
$\frac{2^{64} B}{2^{12} B}$  page table entries/leaf nodes



# Do we really need a large table?



# "Paged" page table & Create on demand

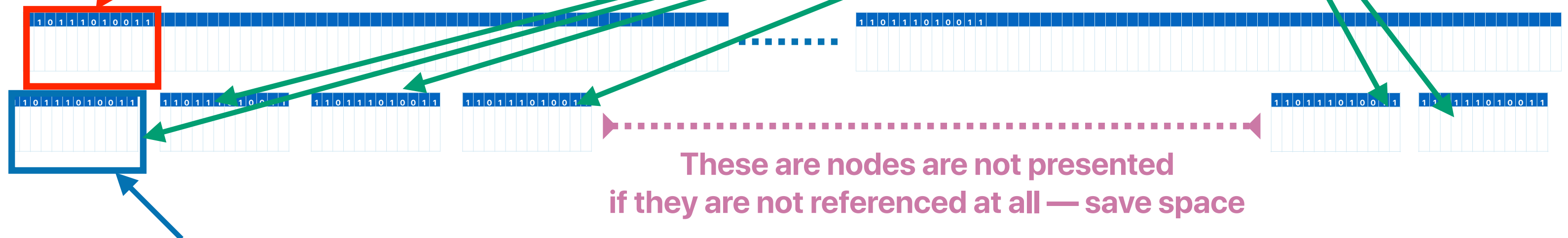


Break up entries into pages!  
Each of these occupies exactly a page

$$\frac{2^{12} B}{2^3 B} = 2^9 \text{ PTEs per node}$$

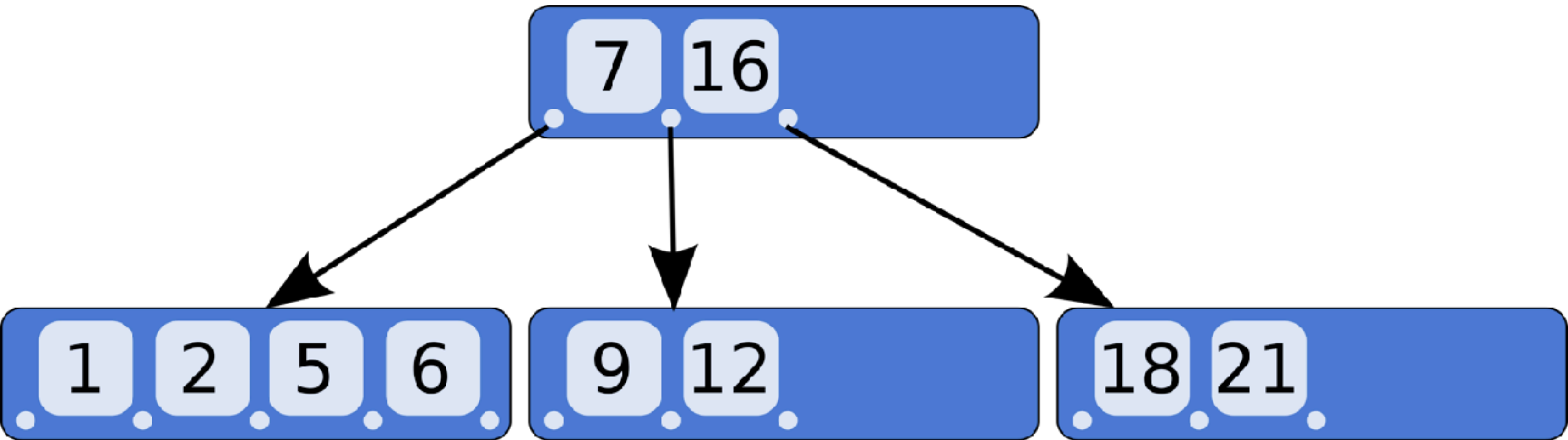
Otherwise, you always need to find more than one consecutive pages — difficult!

Question:  
These nodes are spread out,  
how to locate them in the memory?



Allocate page table entry nodes "on demand"

# B-tree



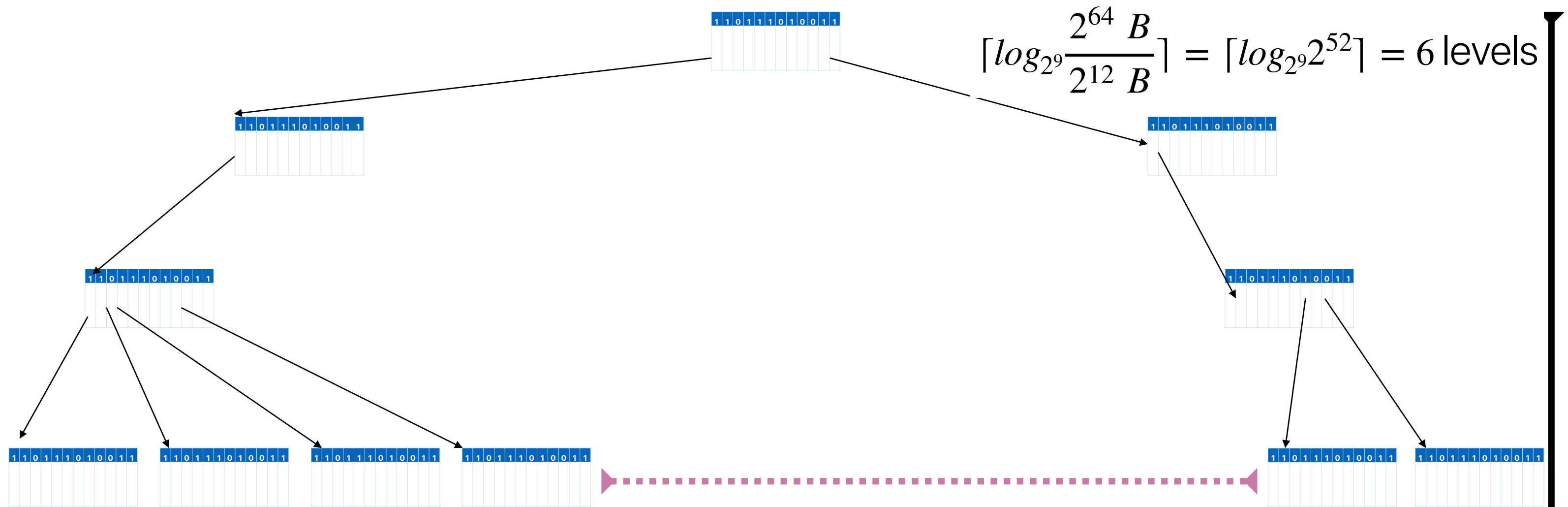
<https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg>



# Hierarchical Page Table

0x0

0xFFFFFFFFFFFFFFFF

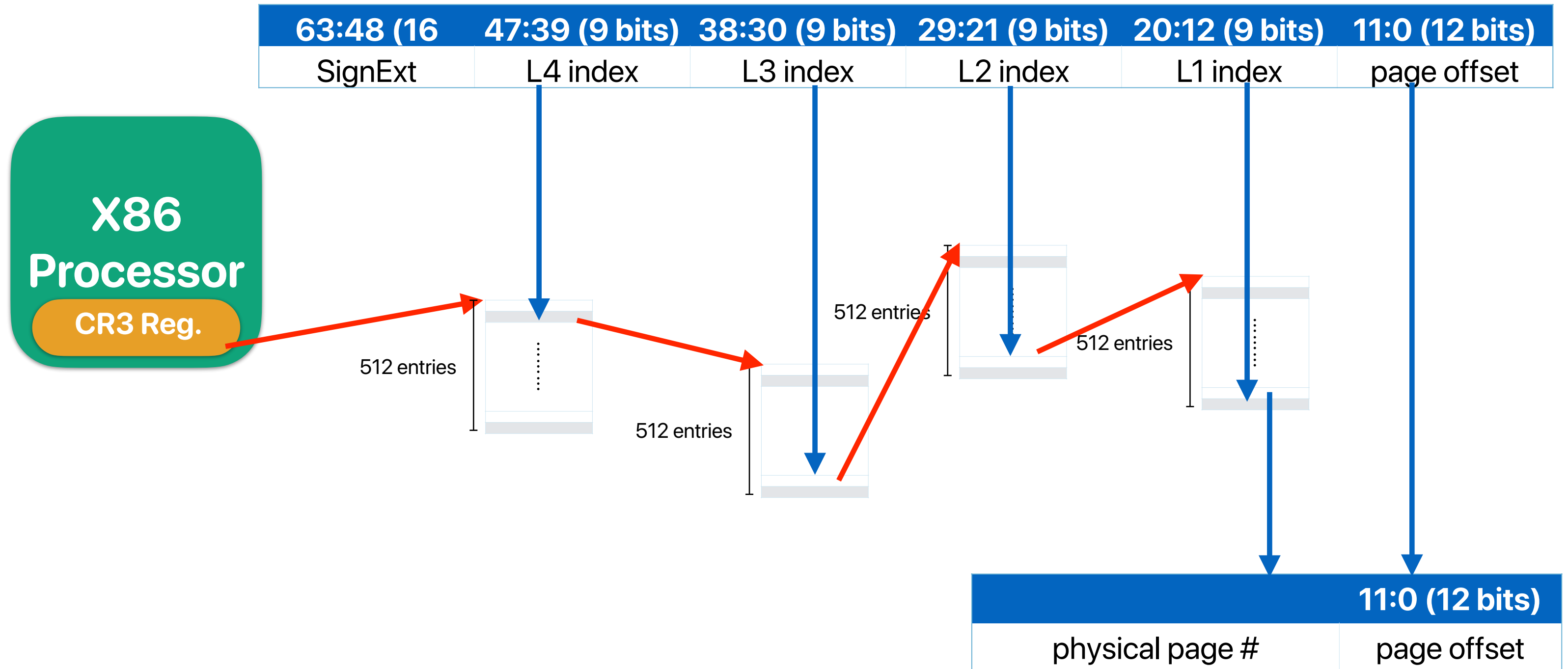


$$\lceil \log_2 \frac{2^{64} B}{2^{12} B} \rceil = \lceil \log_2 2^{52} \rceil = 6 \text{ levels}$$

These are nodes are not presented  
as they are not referenced at all.

$\frac{2^{64} B}{2^{12} B}$  page table entries/leaf nodes (worst case)

# Address translation in x86-64



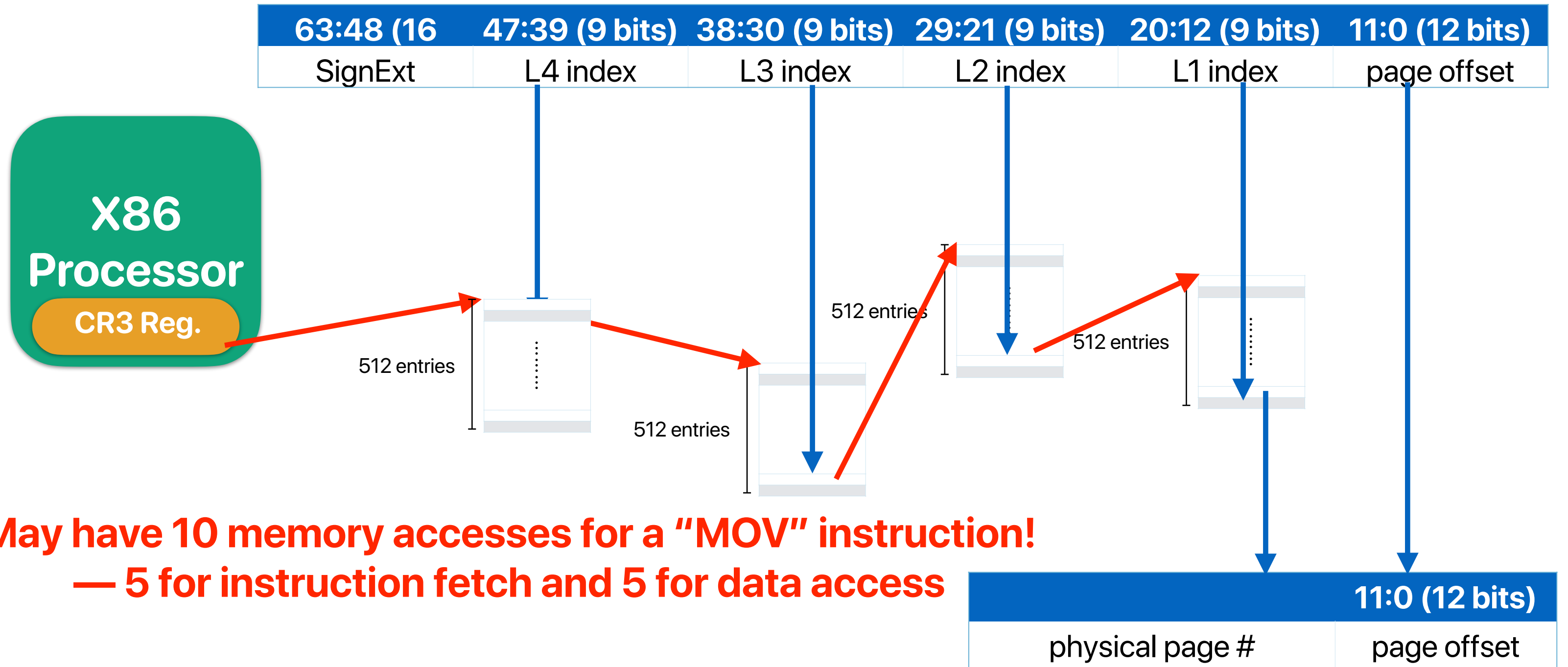


# When we have virtual memory...

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **mov** instruction that access data memory once incur?
- A. 2
  - B. 4
  - C. 6
  - D. 8
  - E. 10

A
B
C
D
E

# Address translation in x86-64

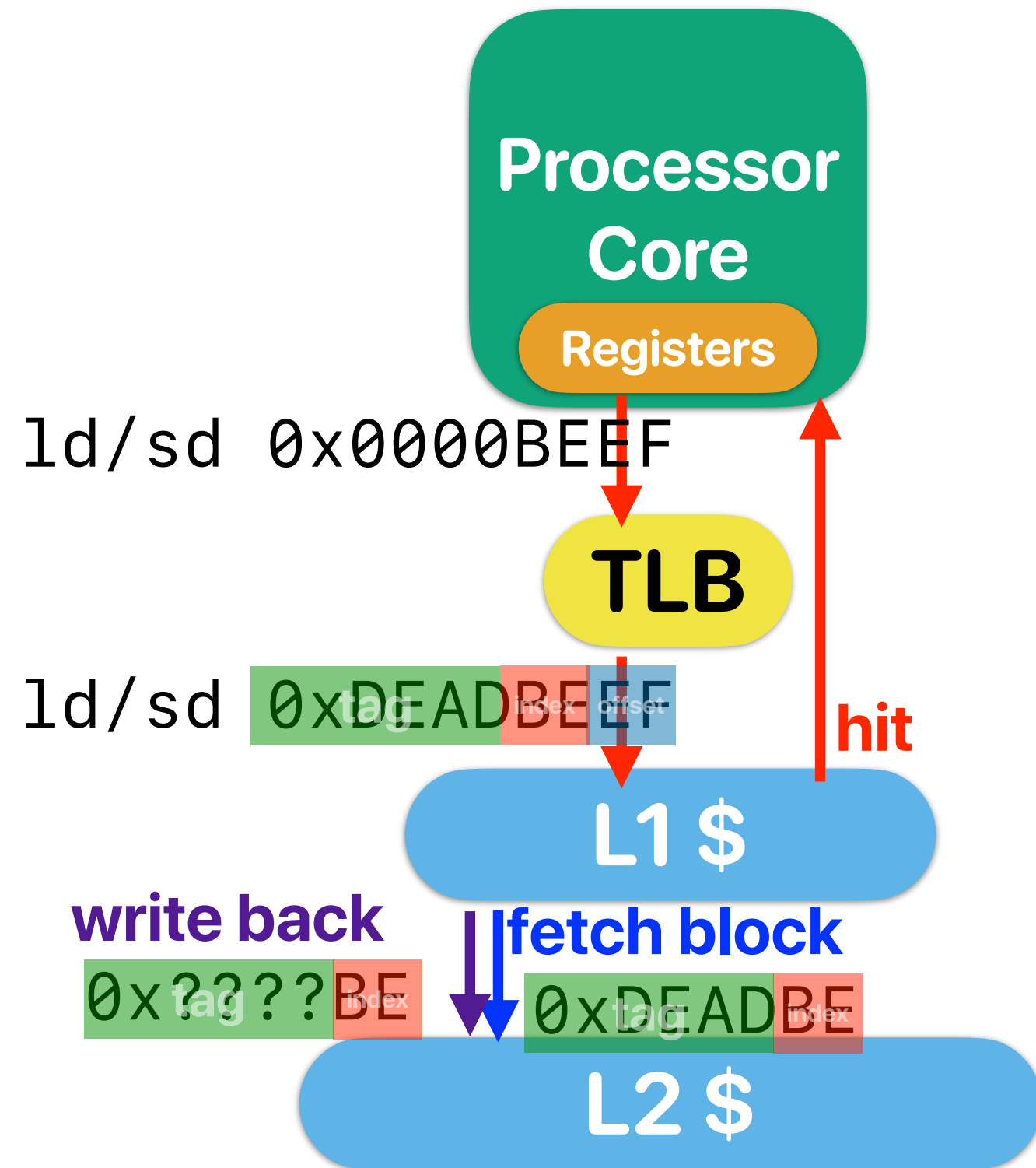


# When we have virtual memory...

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **mov** instruction that access data memory once incur?
  - A. 2
  - B. 4
  - C. 6
  - D. 8
  - E. 10

# **Avoiding the address translation overhead**

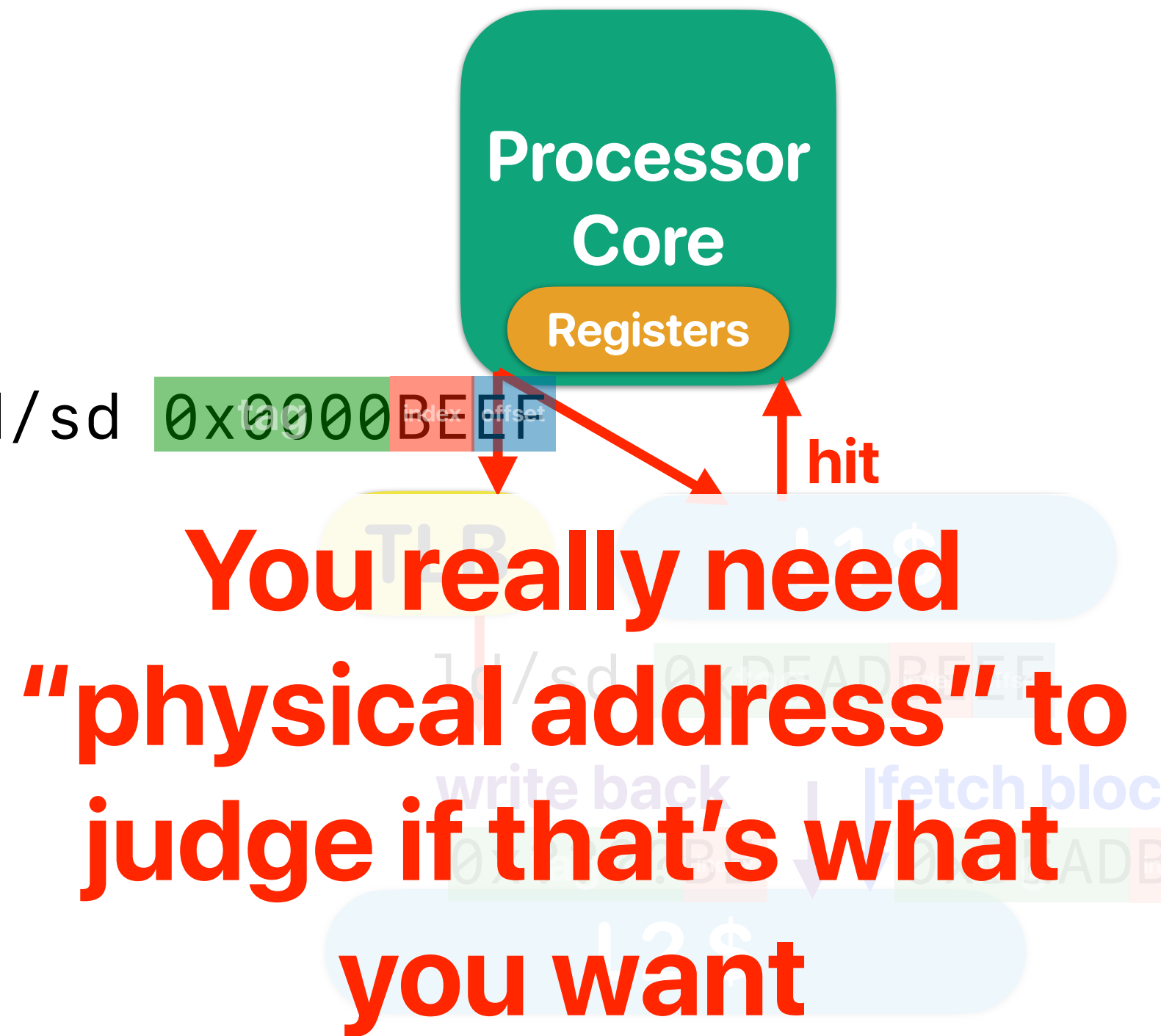
# TLB: Translation Look-aside Buffer



- TLB — a small SRAM stores frequently used page table entries
- Good — A lot faster than having everything going to the DRAM
- Bad — Still on the critical path

# TLB + Virtual cache

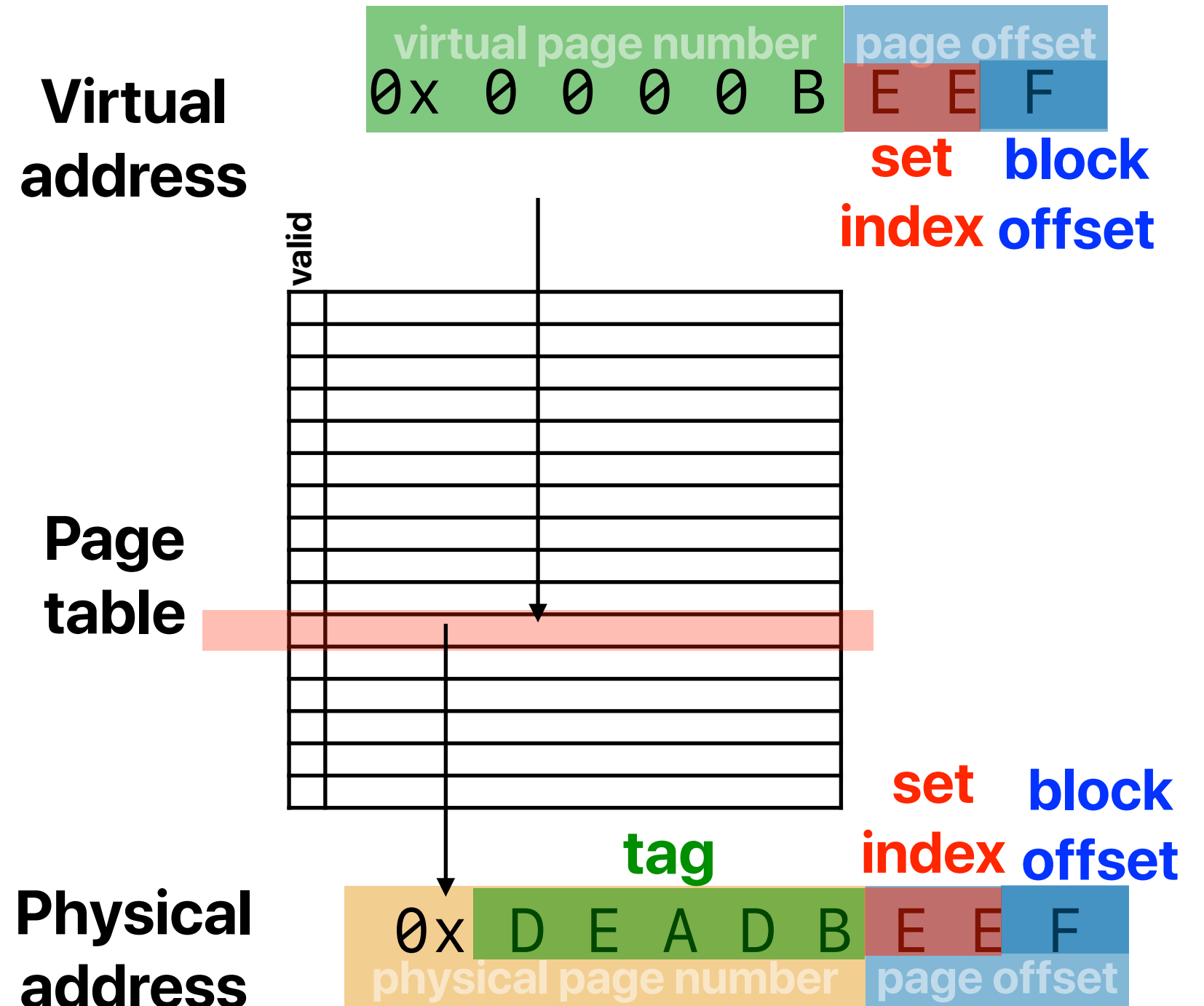
- L1 \$ accepts virtual address — you don't need to translate
- Good — you can access both TLB and L1-\$ at the same time and physical address is only needed if L1-\$ misses
- Bad — it doesn't work in practice
  - Many applications have the same virtual address but should be pointing different **physical addresses**
  - An application can have "aliasing virtual addresses" pointing to the same **physical address**





# Virtually indexed, physically tagged cache

- Can we find physical address directly in the virtual address — Not everything — but the page offset isn't changing!
- Can we indexing the cache using the "partial physical address"?
  - Yes — Just make set index + block set to be exactly the page offset



# Virtually indexed, physically tagged cache

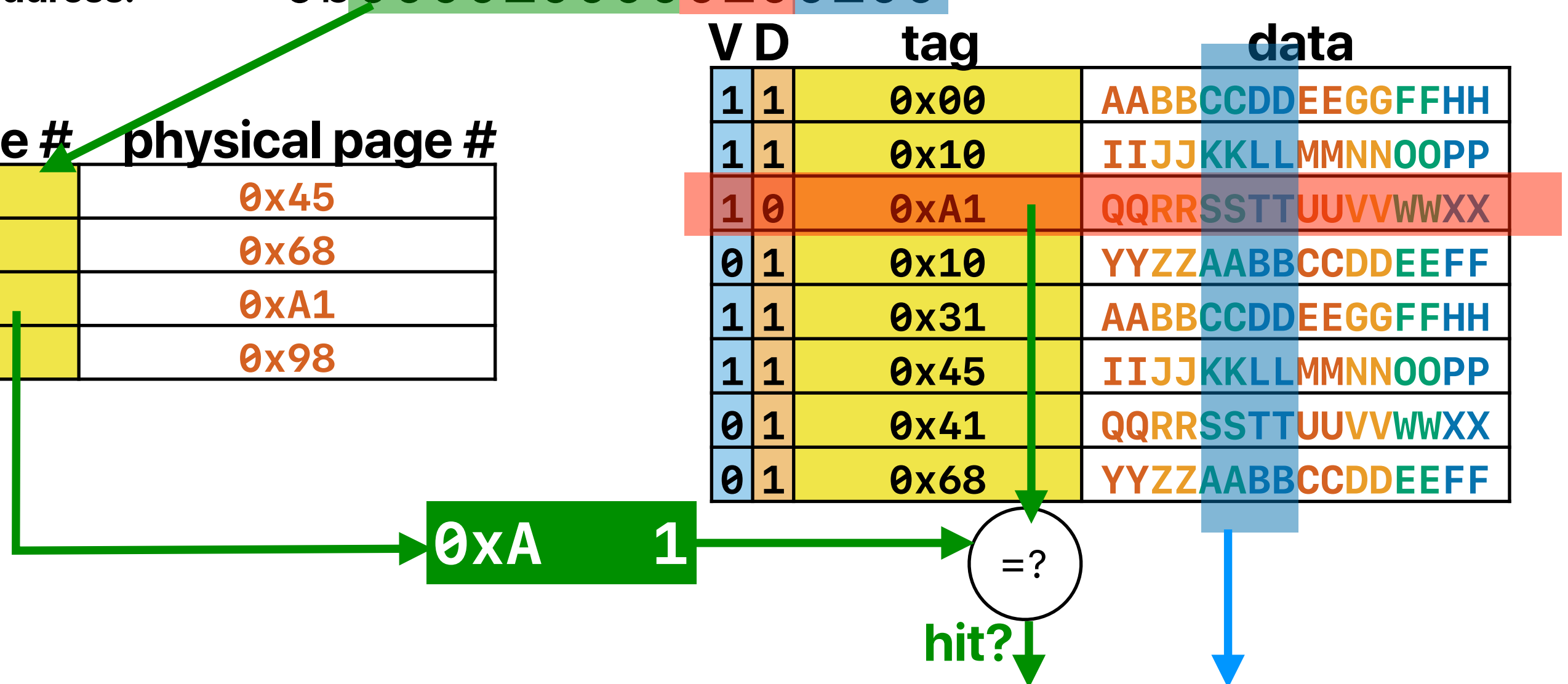
**memory address:**

0x0	8	2	4
		<b>set</b>	<b>block</b>

memory address: 0b0000100000100100

V	virtual page #	physical page #
1	0x29	0x45
1	0xDE	0x68
1	0x10	0xA1
0	0x8A	0x98

V D		tag	data
1	1	0x00	AABBCCDDEEGGFFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFFHH
1	1	0x45	IIJJKKLLMMNNOOPP
0	1	0x41	QQRRSSTTUUVVWWXX
0	1	0x68	YYZZAABBCCDDEEFF



# Virtually indexed, physically tagged cache

- If page size is 4KB —

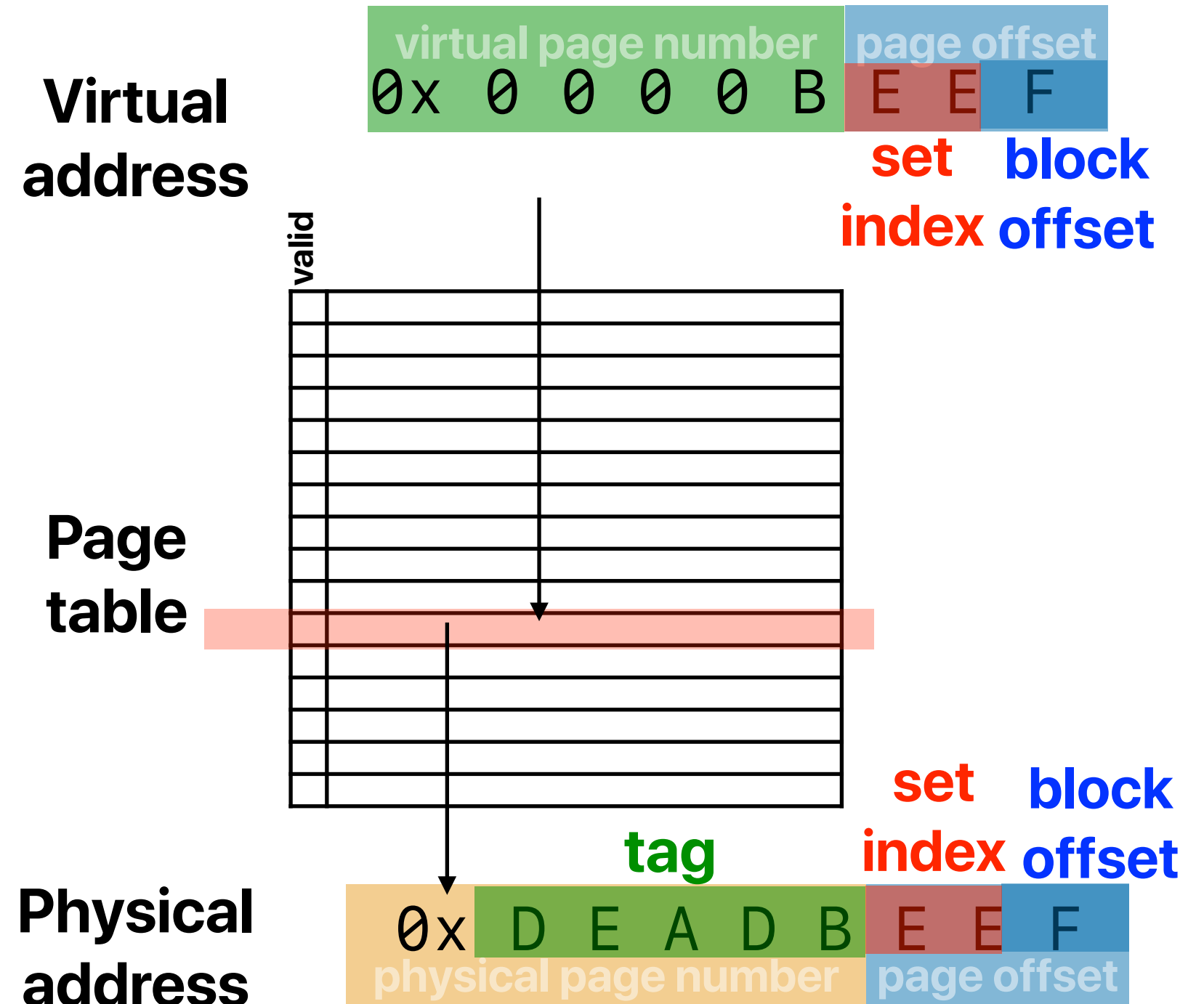
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

$$C = A \times 2^{12}$$

*if*  $A = 1$

$$C = 4KB$$





## Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.
  - A. 32B blocks, 2-way
  - B. 32B blocks, 4-way
  - C. 64B blocks, 4-way
  - D. 64B blocks, 8-way



# Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.

A. 32B blocks, 2-way

B. 32B blocks, 4-way

C. 64B blocks, 4-way

D. 64B blocks, 8-way

Exactly how Core i7 9<sup>th</sup>  
generation configures its own  
cache

$$\lg(B) + \lg(S) = \lg(4096) = 12$$

$$C = ABS$$

$$32KB = A \times 2^{12}$$

$$A = 8$$

# Sample Midterm

# Disclaimer

- This is only giving you an idea about the length and the format about midterm.
- You may find all answers of the sample midterm from lecture slides — don't be lazy

# Format of midterm

- 16x Multiple Choices
- 3x Free answer questions



# Demo — programmer & performance

A

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

B

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

How many of the following make(s) the performance different between version A & version B?

- ① IC
- ② CPI
- ③ CT
- A. 0
- B. 1
- C. 2
- D. 3

# Programmer's impact

- By adding the "sort" in the following code snippet, what the programmer changes in the performance equation to achieve **better** performance?

```
std::sort(data, data + arraySize);
```

```
for (unsigned c = 0; c < arraySize*1000; ++c) {  
    if (data[c%arraySize] >= INT_MAX/2)  
        sum ++;  
}
```

- A. CPI
- B. IC
- C. CT
- D. IC & CPI
- E. CPI & CT

# How compilers affect performance

- Performance equation consists of the following three factors
  - ① IC
  - ② CPI
  - ③ CT

How many can the **compiler** affect?

- A. 0
- B. 1
- C. 2
- D. 3

# Speedup further!

- With the latest flash memory technologies, the system spends 16% of time on accessing the flash, and the software overhead is now 84%. If your company ask you and your team to invent a new memory technology that replaces flash to achieve 2x speedup on loading maps, how much faster the new technology needs to be?
  - A.  $\sim 5x$
  - B.  $\sim 10x$
  - C.  $\sim 20x$
  - D.  $\sim 100x$
  - E. None of the above

# Amdahl's Law on Multicore Architectures

- Regarding Amdahl's Law on multicore architectures, how many of the following statements is/are correct?
  - ① If we have unlimited parallelism, the performance of executing each parallel partition does not matter as long as the performance slowdown in each piece is bounded
  - ② With unlimited amount of parallel hardware units, single-core performance does not matter anymore
  - ③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts
  - ④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

# How reflective is FLOPs?

- Given the FLOPs number measured, how many of the followings are true?
    - ① The FLOPs number remains the same on each architecture if we change the data size
    - ② The FLOPs number remains the same on each architecture if we change the data type to double
    - ③ The FLOPs number remains the same on each architecture if we change the algorithm implementation
    - ④ The FLOPs number reflects the performance ratio of different architectures when executing floating point applications
- A. 0  
B. 1  
C. 2  
D. 3  
E. 4

# How can "memory hierarchy" help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture 90% of the desired data/instructions. what's the average CPI (pick the closest one)?

- A. 6
- B. 8
- C. 10
- D. 12
- E. 67

# Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint32_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint32_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```

- A. Access of `matrix` has temporal locality, `vector` has spatial locality
- B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
- C. Access of `matrix` has spatial locality, `vector` has temporal locality
- D. Both `matrix` and `vector` have spatial locality and temporal locality
- E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality



# NVIDIA Tegra X1

- L1 data (D-L1) cache configuration of NVIDIA Tegra X1 (used by Nintendo Switch and Jetson Nano)
  - Size 32KB, 4-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

- A. Tag is 49 bits
- B. Index is 8 bits
- C. Offset is 7 bits
- D. The cache has 1024 sets
- E. None of the above

# intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 48KB, 12-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?
    - A. Tag is 52 bits
    - B. Index is 6 bits
    - C. Offset is 6 bits
    - D. The cache has 128 sets
    - E. All of the above are correct

# 3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity

How many of the following are correct?

- ① Increasing associativity can reduce conflict misses
- ② Increasing associativity can reduce hit time
- ③ Increasing block size can increase the miss penalty
- ④ Increasing block size can reduce compulsory misses

A. 0

B. 1

C. 2

D. 3

E. 4

# Which of the following schemes can help Tegra?

- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help NVIDIA’s Tegra for the code in the previous slide?
    - ① Missing cache
    - ② Victim cache
    - ③ Prefetch
    - ④ Stream buffer
- A. 0  
B. 1  
C. 2  
D. 3  
E. 4

# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
    - ① Non-blocking/pipelined/multibanked cache
    - ② Critical word first and early restart
    - ③ Prefetching
    - ④ Write buffer
- A. 0  
B. 1  
C. 2  
D. 3  
E. 4

# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

- A. ~10%
- B. ~20%
- C. ~40%
- D. ~80%
- E. 100%

# What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

```
Block
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

Block + Transpose

```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}

for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        // Compute on b_t
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
        }
    }
}
```

# When we have virtual memory...

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **mov** instruction that access data memory once incur?
  - A. 2
  - B. 4
  - C. 6
  - D. 8
  - E. 10



# Performance Equation

- Consider the following c code snippet and x86 instructions implement the code snippet

C	x86
<pre>for(i = 0; i &lt; count; i++) {     s += a[i]; }</pre>	<pre>.L3: movslq    (%rdi), %rdx addq      \$4, %rdi addq      %rdx, %tax cmpq      %rcx, %rdi jne       .L3</pre>

If (1) count is set to 1,000,000,000, (2) a memory instruction takes 4 cycles, (3) a branch/jump instruction takes 3 cycles, (4) other instructions takes 1 cycle on average, and (5) the processor runs at 4 GHz, how much time is it take to finish executing the code snippet?

# Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to  
SELECT AVG(assignment\_1) FROM table  
Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays
<pre>struct grades {     int id;     double *homework;     double average; }; table = (struct grades *) \ malloc(num_of_students*sizeof(struct</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; }; table =(struct grades *)malloc(sizeof(struct grades));</pre>

- A. Array of objects
- B. Object of arrays

# Speedup of Y over X

- Consider the same program on the following two machines, X and Y. By how much Y is faster than X?

	Clock Rate	Instructions	Percentage of Type-A	CPI of Type-A	Percentage of Type-B	CPI of Type-B	Percentage of Type-C	CPI of Type-C
Machine X	4 GHz	5000000000	20%	4	20%	3	60%	1
Machine Y	6 GHz	5000000000	20%	6	20%	3	60%	1

# Practicing Amdahl's Law

- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time and a better processor to accelerate the software overhead by 2x. By how much can we speed up the map loading process?

# NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

# Q & A





# Announcement

- Reading quiz #4 due **next Thursday** before the lecture
- Assignment #3 due tonight
- Programming assignments are perfectly linked to lectures
- Midterm
  - Next Tuesday 3:30p-4:50p
  - No outside materials allowed
  - Please be clear on your answers — we're not psychics

# Computer Science & Engineering

# 203

# つづく

