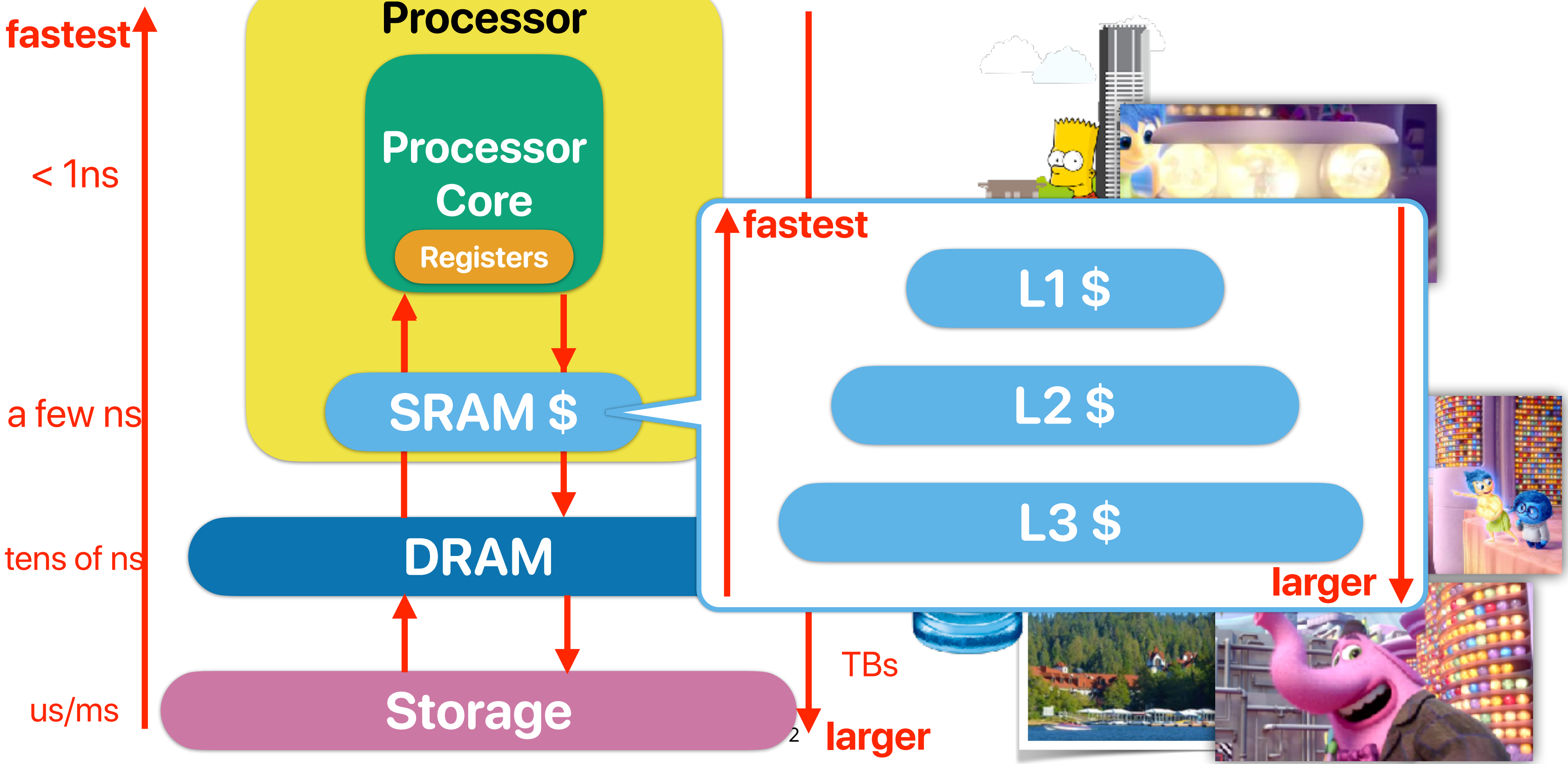


# **Software optimizations for cache/ Virtual memory & memory hierarchy**

Hung-Wei Tseng

# Recap: Memory Hierarchy



# Recap: NVIDIA Tegra X1 100% miss rate!

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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]
```

C = ABS  
32KB = 4 \* 64 \* S  
S = 128  
offset = lg(64) = 6 bits  
index = lg(128) = 7 bits  
tag = the rest bits

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0 <b>b0001000</b> 000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0 <b>b0010000</b> 000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0 <b>b0011000</b> 000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0 <b>b0100000</b> 000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0 <b>b0101000</b> 000000000000000	0x28	0x0	Compulsory Miss	a[0-7]
a[1]	0x10008	0 <b>b0001000</b> 00000000001000	0x8	0x0	<b>Conflict Miss</b>	b[0-7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	<b>Conflict Miss</b>	c[0-7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	<b>Conflict Miss</b>	d[0-7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	<b>Conflict Miss</b>	e[0-7]
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	<b>Conflict Miss</b>	a[0-7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

# Loop optimizations

## Loop interchange

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];
    }
}
```



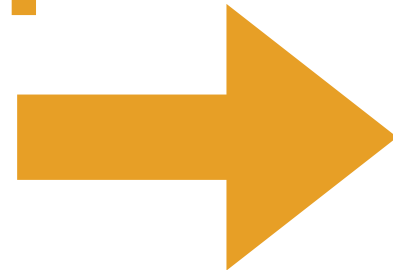
B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

## Loop fission

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```



A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

## Loop fusion

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```



# Outline

- Case studies of software optimizations
- Virtual memory
- Architectural support for virtual memory
- Advanced hardware support for virtual memory

# Case study: Matrix Multiplication

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

**Algorithm class tells you it's  $O(n^3)$**

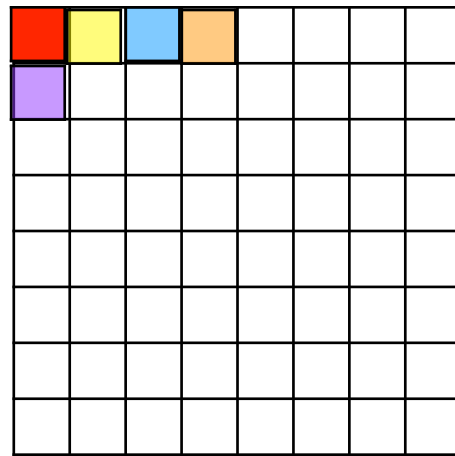
**If  $n=1024$ , it takes about 1 sec**

**How long is it take when  $n=2048$ ?**

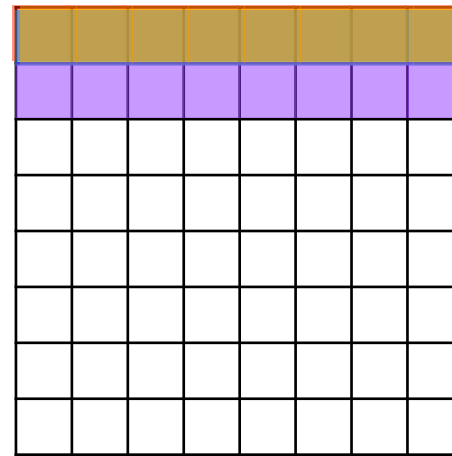
# Recap: Matrix Multiplication

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

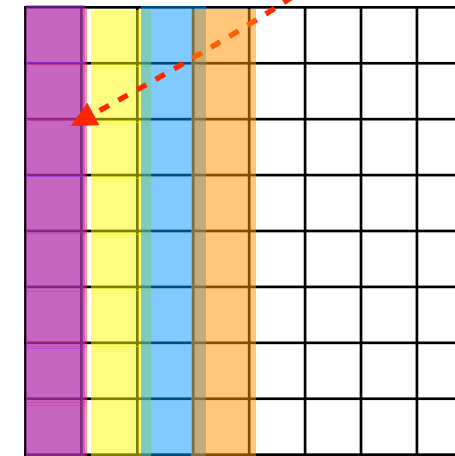
Very likely a miss if  
array is large



c



a



b

- If each dimension of your matrix is 2048
  - Each row takes  $2048 \times 8$  bytes = 16KB
  - The L1 \$ of intel Core i7 or AMD RyZen is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 2 rows/columns of each matrix!
  - You need the same row when j increase!

**It's show time!**



# Simply the addresses of B[]

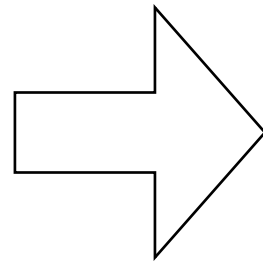
element	address	tag	index
b[0][0]	7F68297E1000	7F68297E1	0
b[1][0]	7F68297E1800	7F68297E1	20
b[2][0]	7F68297E2000	7F68297E2	0
b[3][0]	7F68297E2800	7F68297E2	20
b[4][0]	7F68297E3000	7F68297E3	0
b[5][0]	7F68297E3800	7F68297E3	20
b[6][0]	7F68297E4000	7F68297E4	0
b[7][0]	7F68297E4800	7F68297E4	20
b[8][0]	7F68297E5000	7F68297E5	0
b[9][0]	7F68297E5800	7F68297E5	20
b[10][0]	7F68297E6000	7F68297E6	0
b[11][0]	7F68297E6800	7F68297E6	20
b[12][0]	7F68297E7000	7F68297E7	0
b[13][0]	7F68297E7800	7F68297E7	20
b[14][0]	7F68297E8000	7F68297E8	0
b[15][0]	7F68297E8800	7F68297E8	20
b[16][0]	7F68297E9000	7F68297E9	0
b[17][0]	7F68297E9800	7F68297E9	20
b[18][0]	7F68297EA000	7F68297EA	0

We only used  $17 \times 64 = 1088$  bytes!

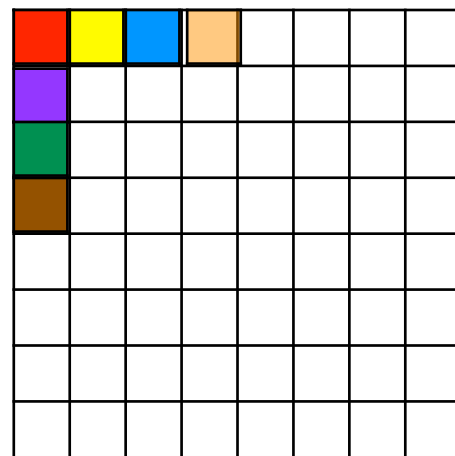
Conflict miss starts! 

# Block algorithm for matrix multiplication

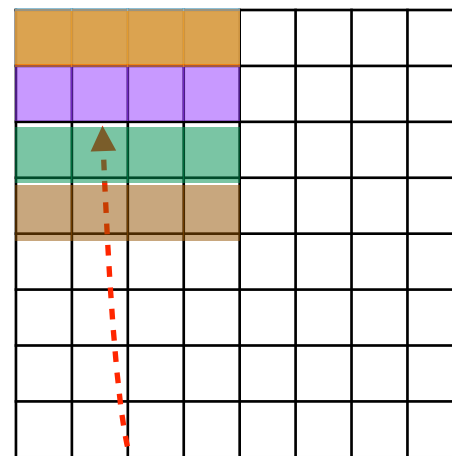
```
for(i = 0; i < ARRAY_SIZE; i++) {  
  for(j = 0; j < ARRAY_SIZE; j++) {  
    for(k = 0; k < ARRAY_SIZE; k++) {  
      c[i][j] += a[i][k]*b[k][j];  
    }  
  }  
}
```



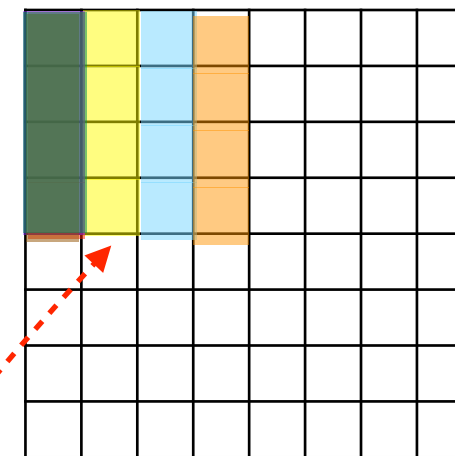
```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
          for(kk = k; kk < k+(ARRAY_SIZE/n);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



c



a



b

**You only need to hold these  
sub-matrices in your cache**

# What kind(s) of misses can block algorithm remove?

- Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an intel Core i7)

## Naive

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

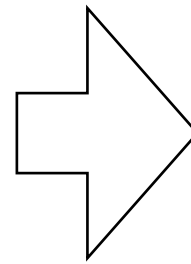
## Block

```
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

# Matrix Transpose

```
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];  
        }  
    }  
}
```



```
// 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];  
        }  
    }  
}
```

# 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];
        }
    }
}
```

**It's show time!**

# Tips of software optimizations

- Carefully layout your data structure can improve capacity misses!
- Make your data structures align with the access pattern can better exploit cache locality — improve conflict misses
- Implementing algorithms in a more cache friendly way!

# Virtual memory



# 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 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;
}
```

**It's show time!**

# 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;
}
```

# 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;
}
```

**It's show time!**

# 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



**Program A**

0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

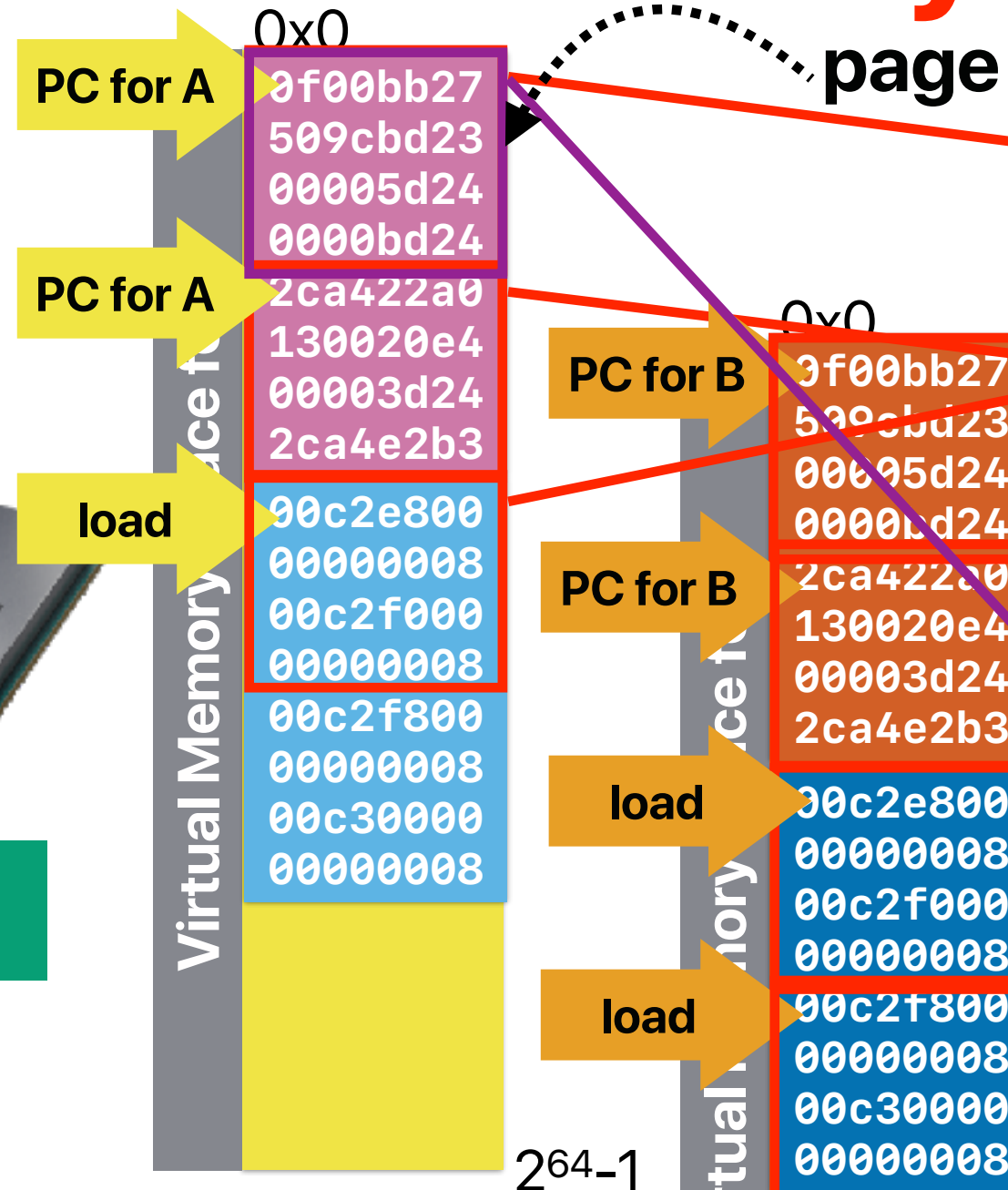


**Program B**

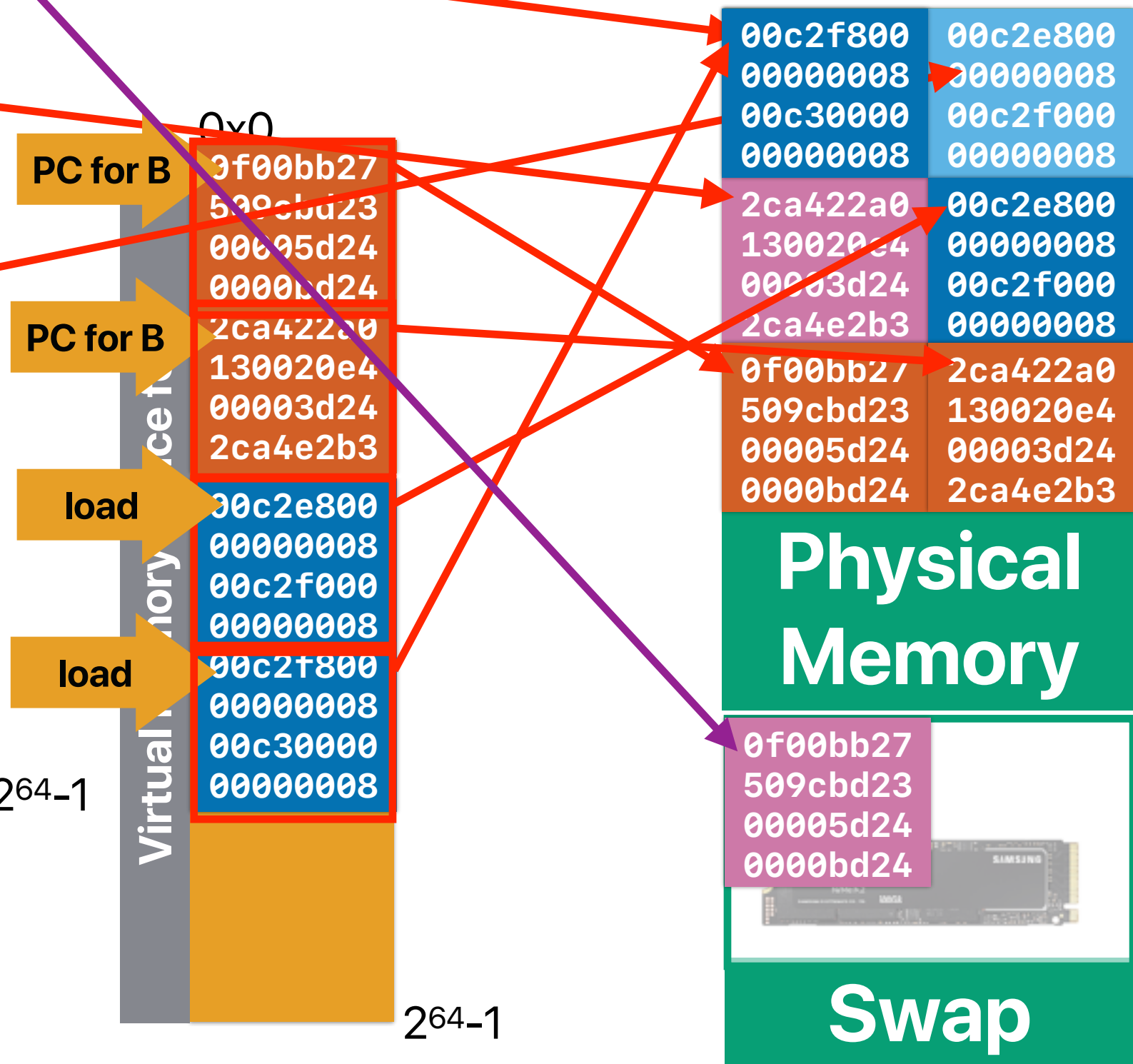
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008



# Virtual memory



This approach is called demand paging + swapping



# Demo revisited

```
#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;
```

```
}
```

**&a = 0x601090**

**Process A**

**Process B**

**Process A's  
Virtual  
Memory Space**


**Process B's  
Virtual  
Memory Space**

# 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**"

# 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



# Processor Core

Registers

1

# Registers

load 0x0009

# Page table

# Main memory (DRAM)

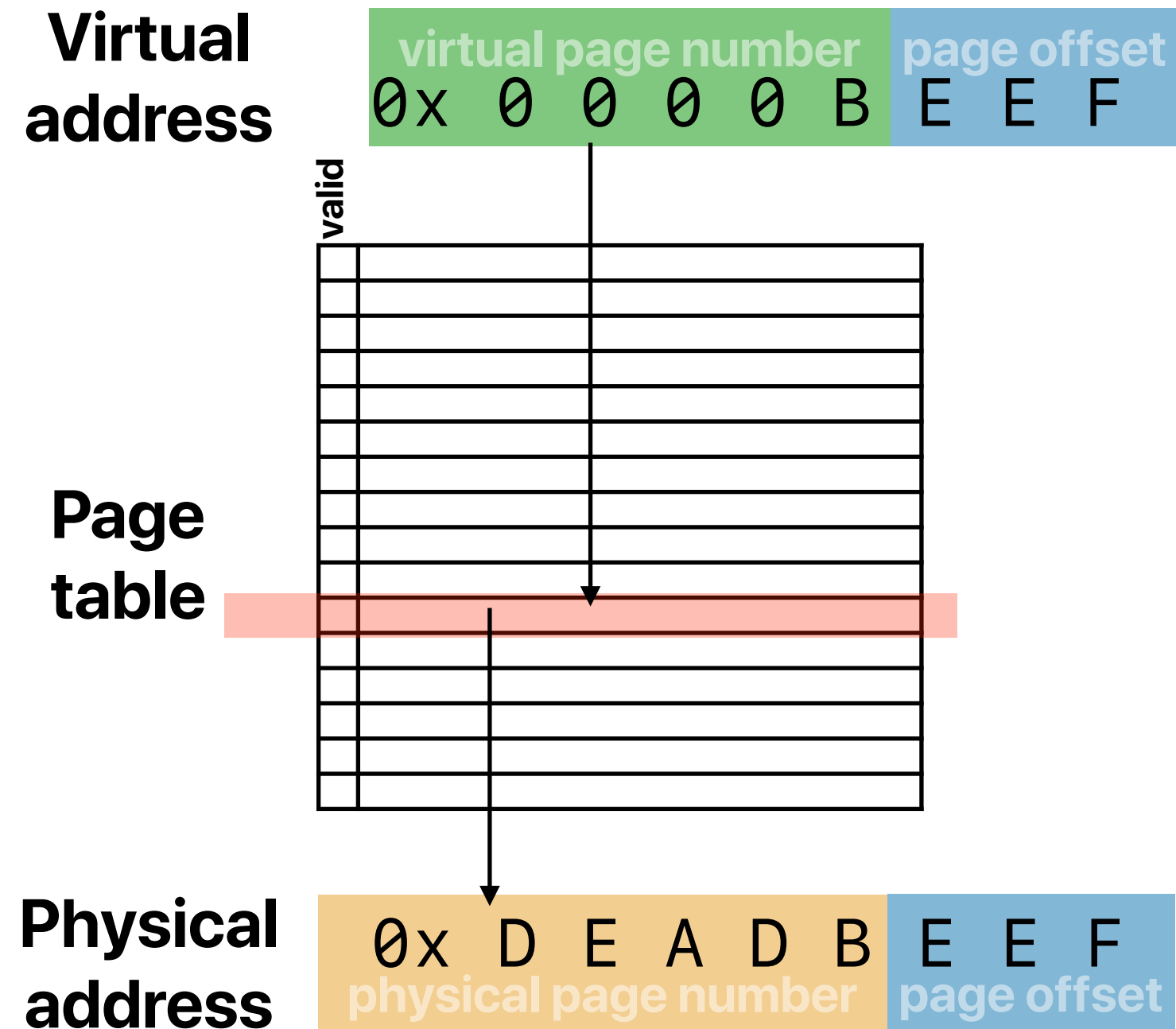


# Demand paging

- 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

# 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

# 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

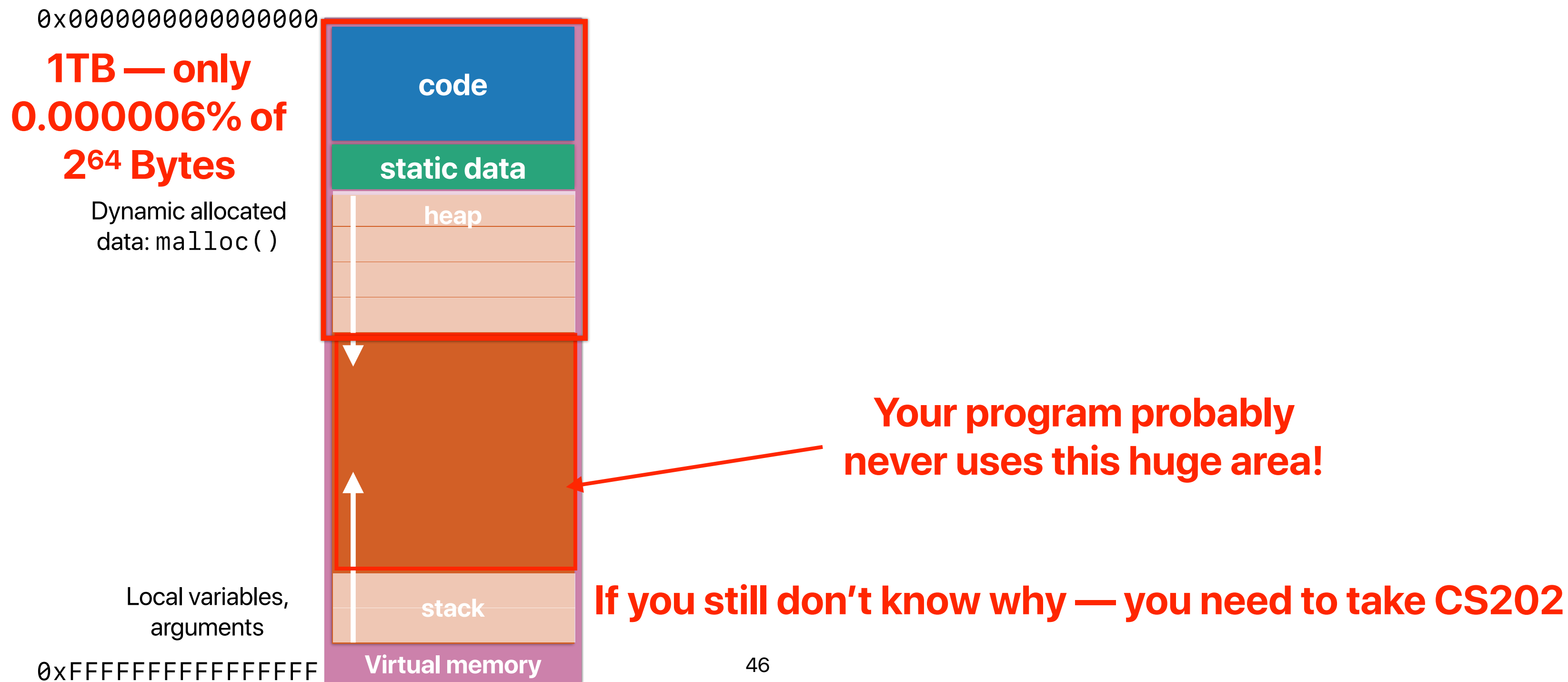
Virtual Address Space

- 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!

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



# Do we really need a large table?

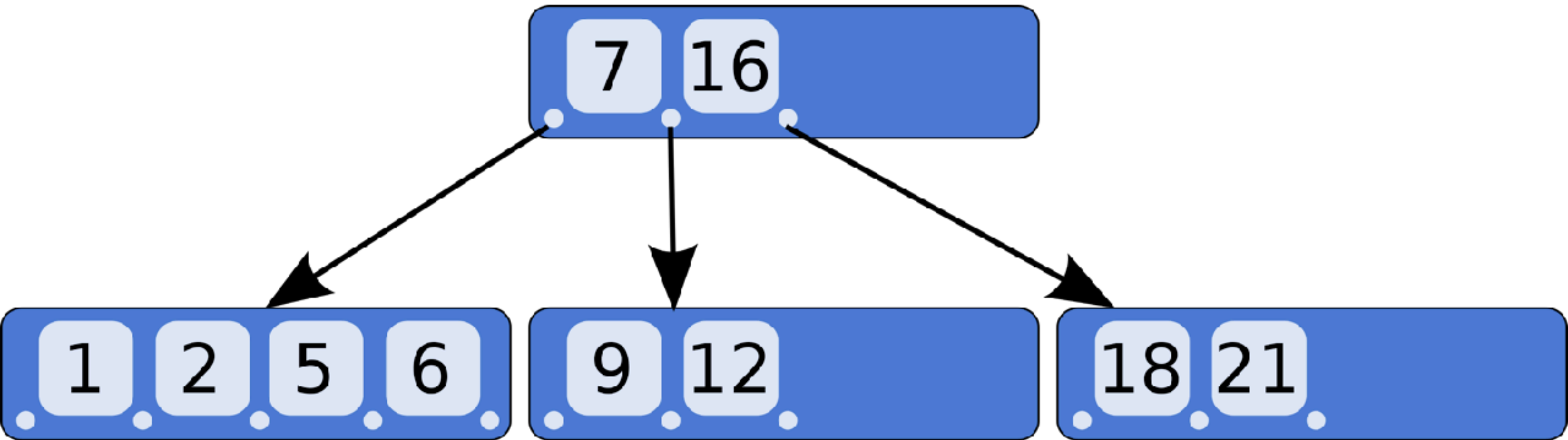


```
0xFFFFFFFFFFFFFFFF
```



45

# B-tree

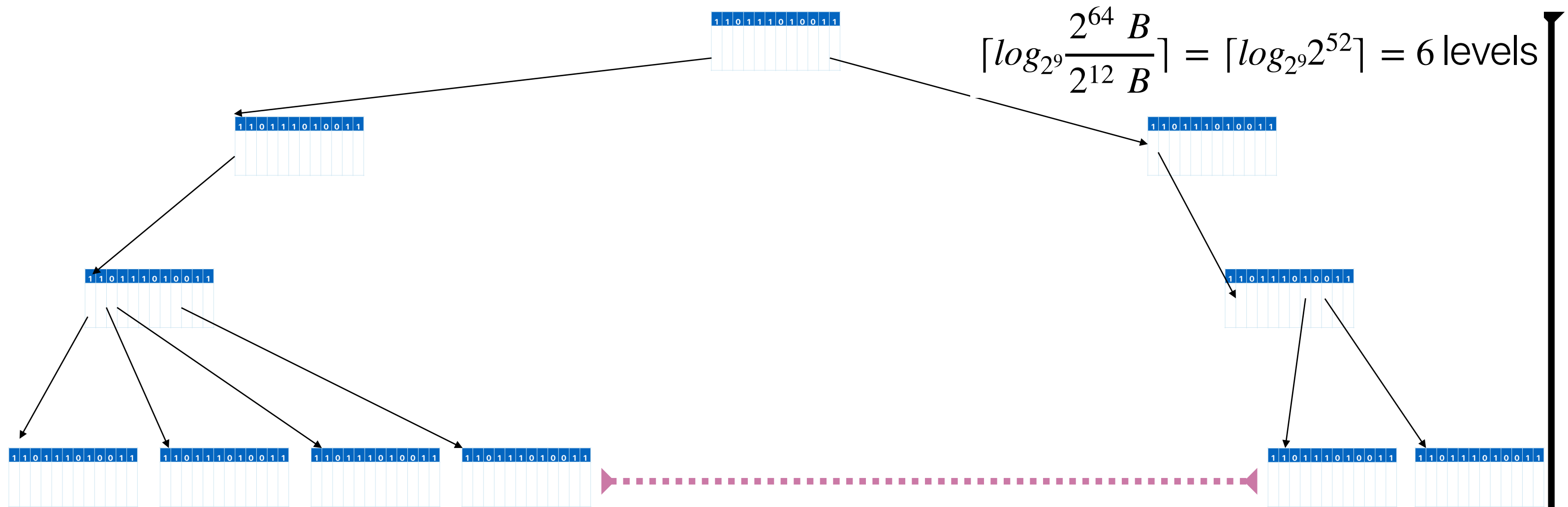
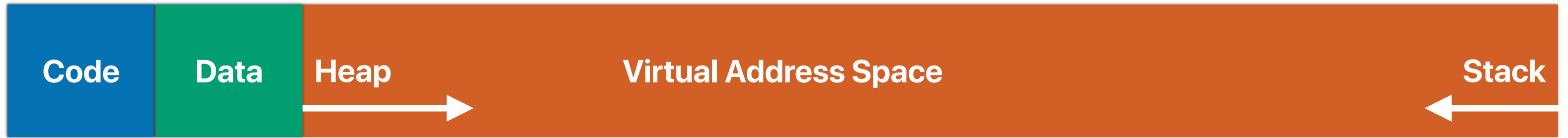


<https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg>

# Hierarchical Page Table

0x0

0xFFFFFFFFFFFFFFFF

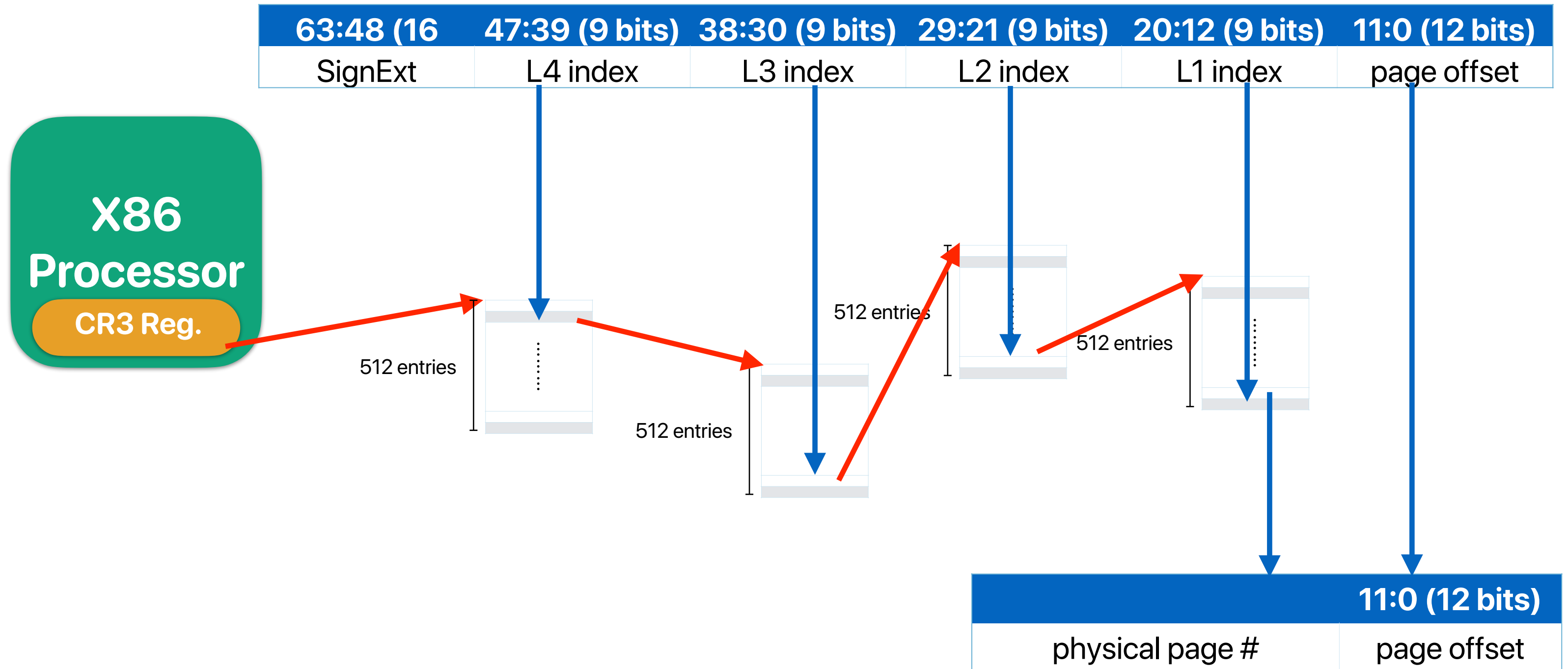


$$\lceil \log_2 \frac{2^{64} B}{2^{12} B} \rceil = \lceil \log_2 2^{52} \rceil = 6 \text{ levels}$$

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

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

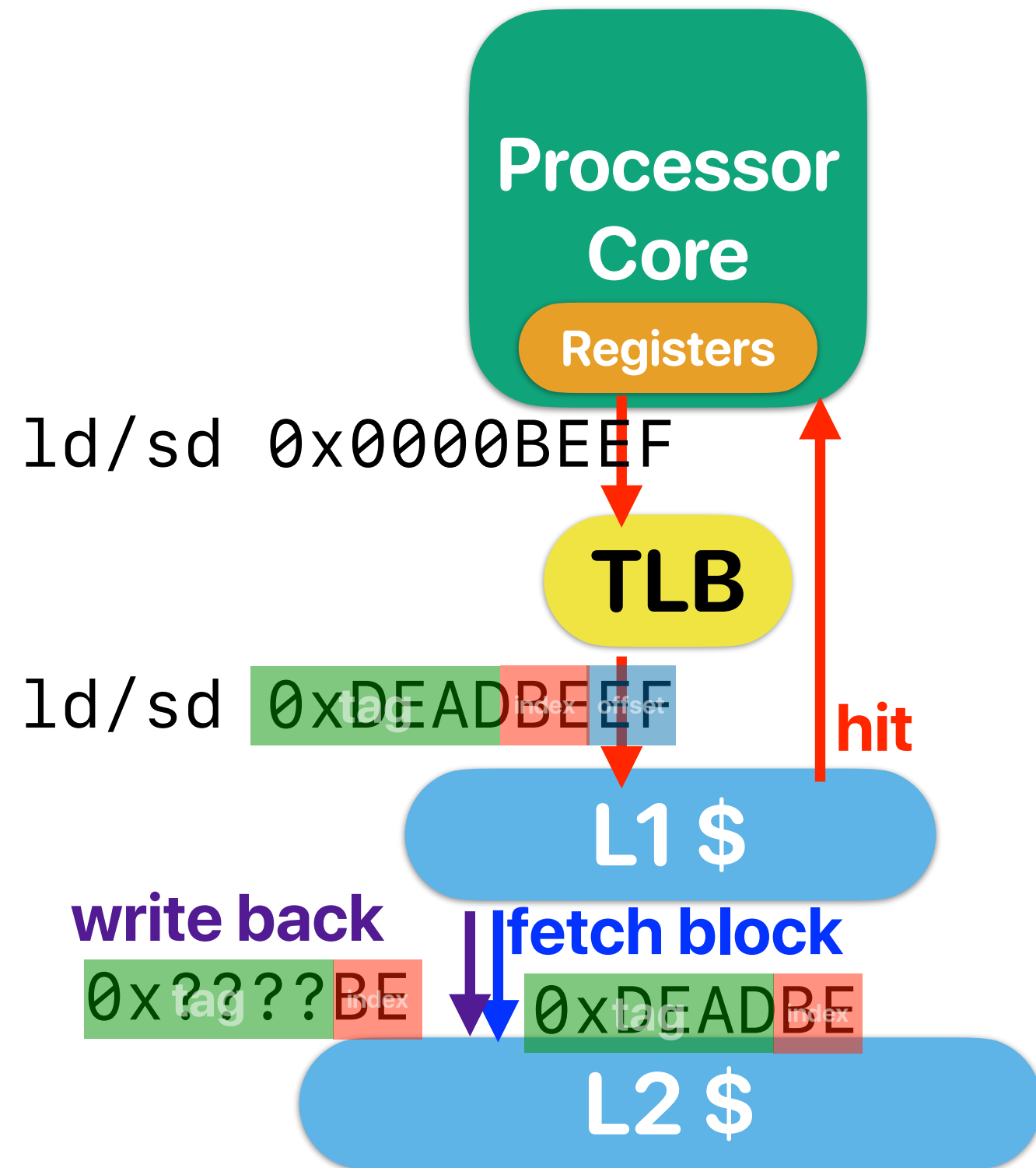
# Address translation in x86-64





# **Avoiding the address translation overhead**

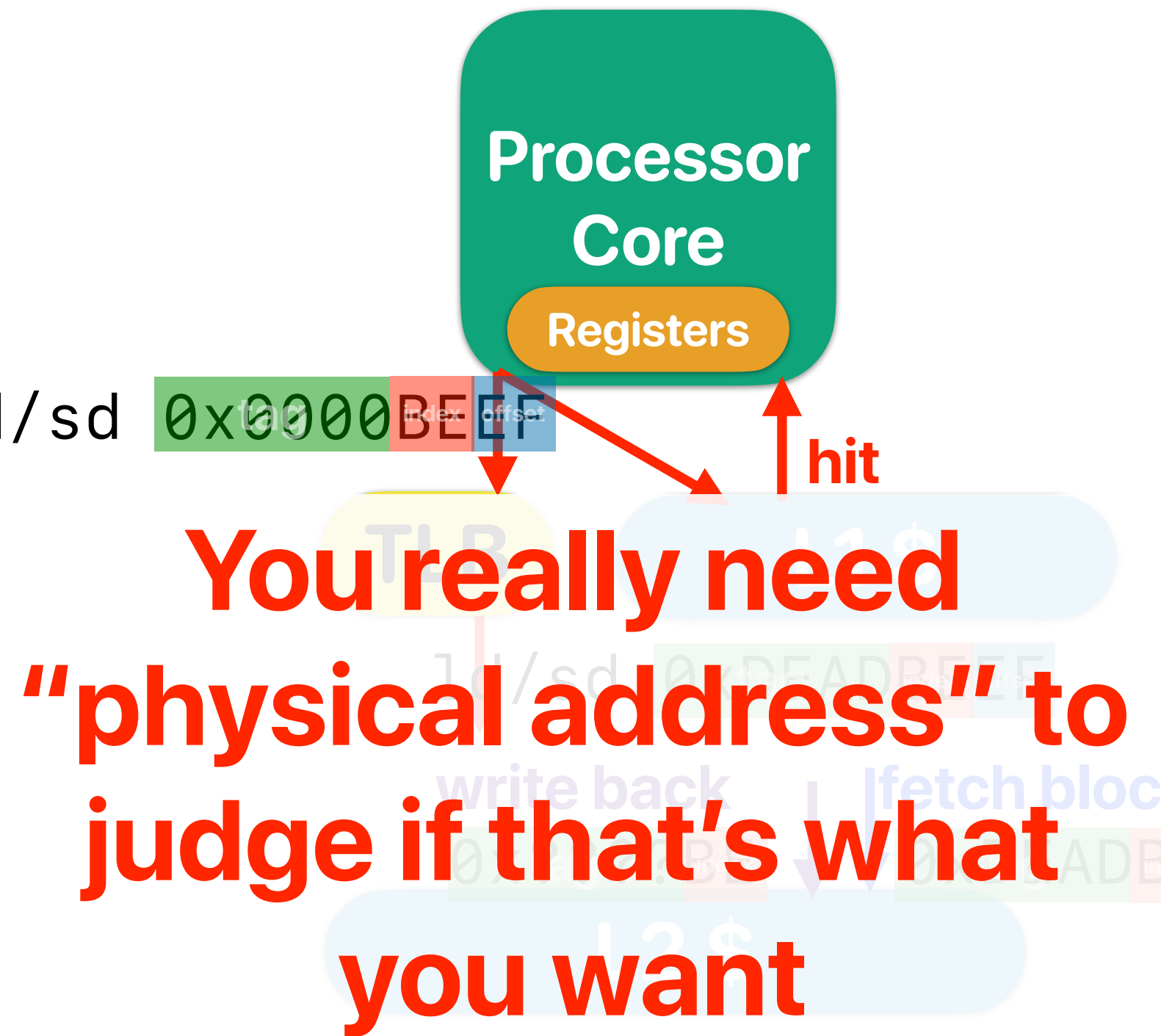
# TLB: Translation Look-aside Buffer



- 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

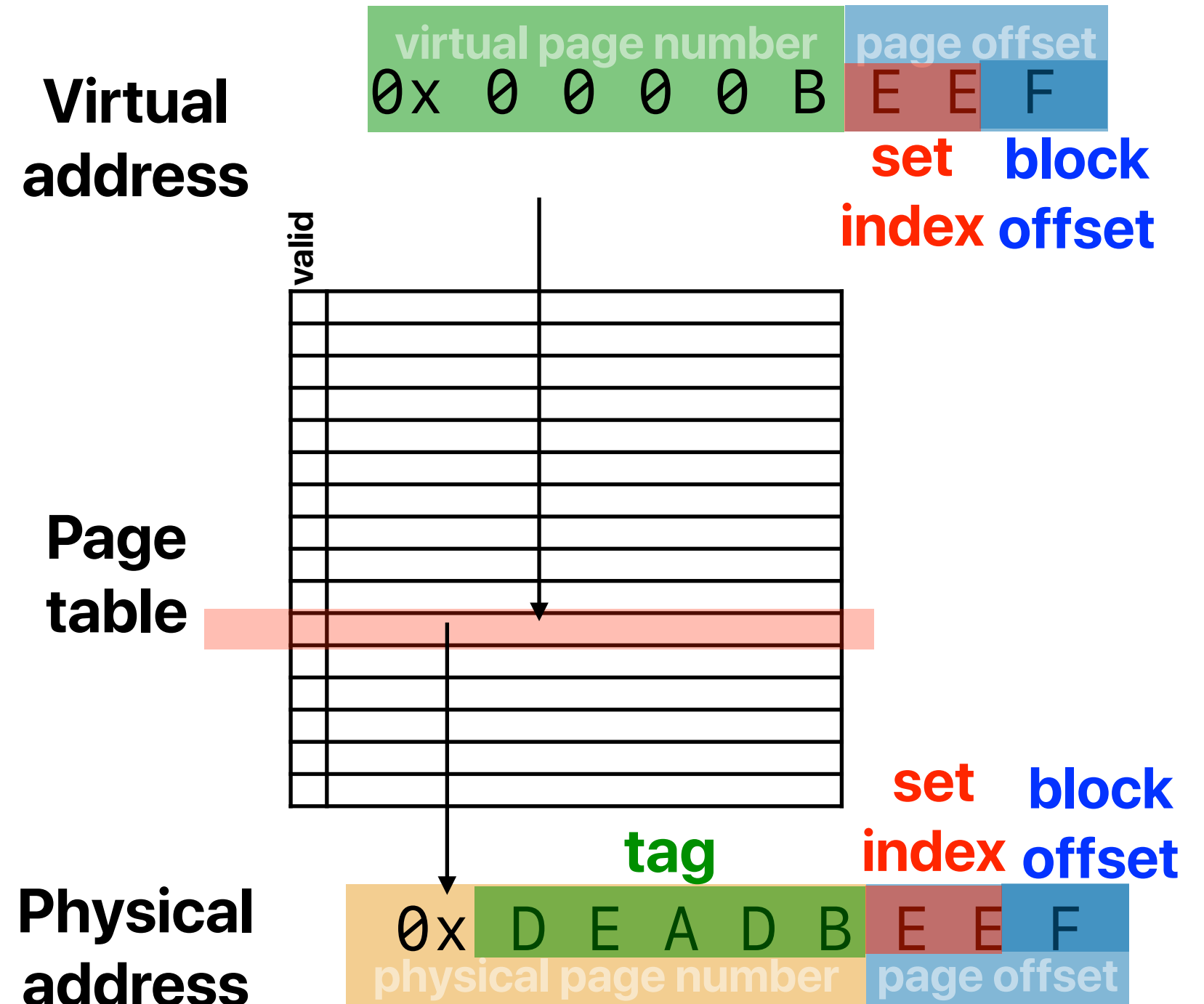
# TLB + Virtual cache

- 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

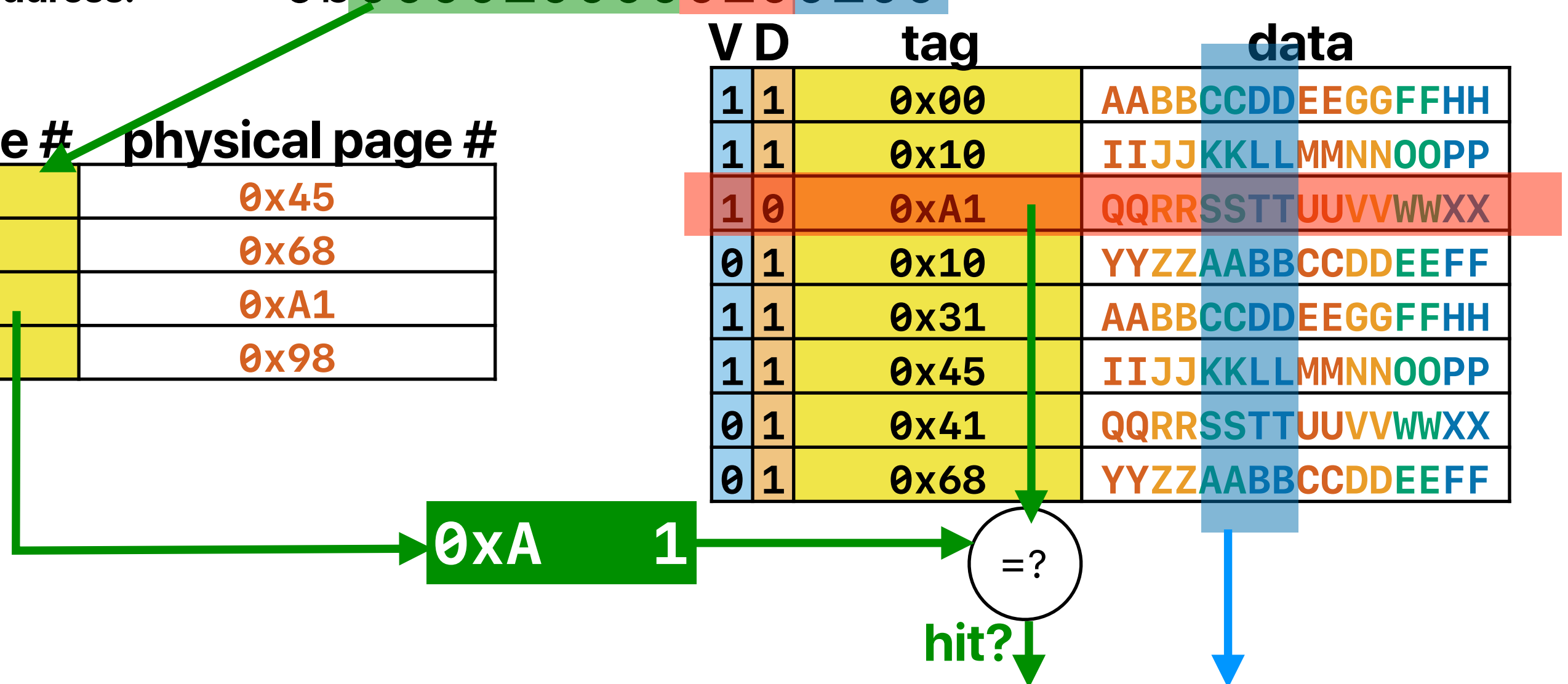
**memory address:**

0x0	8	2	4
		set	block

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 —

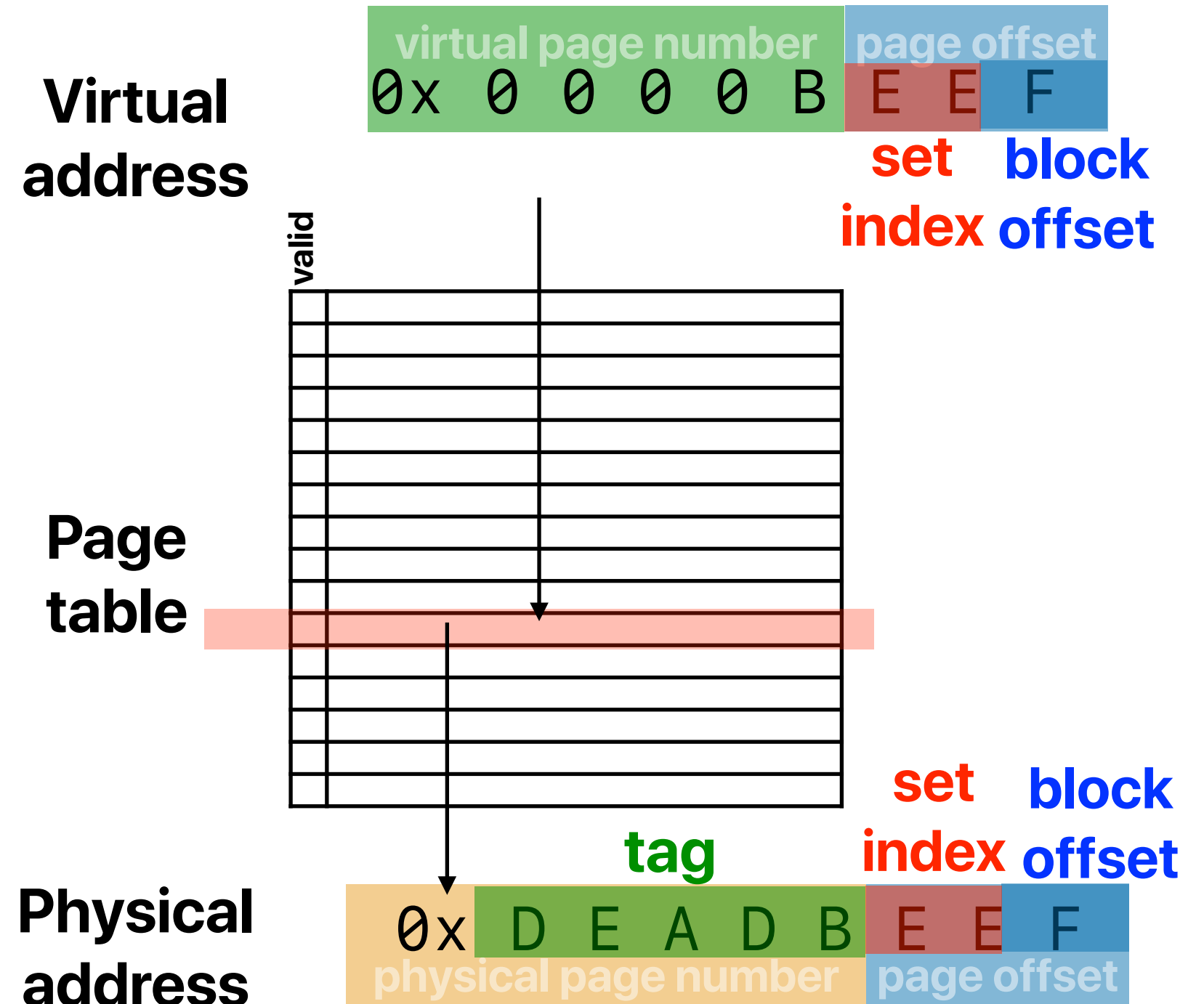
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

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

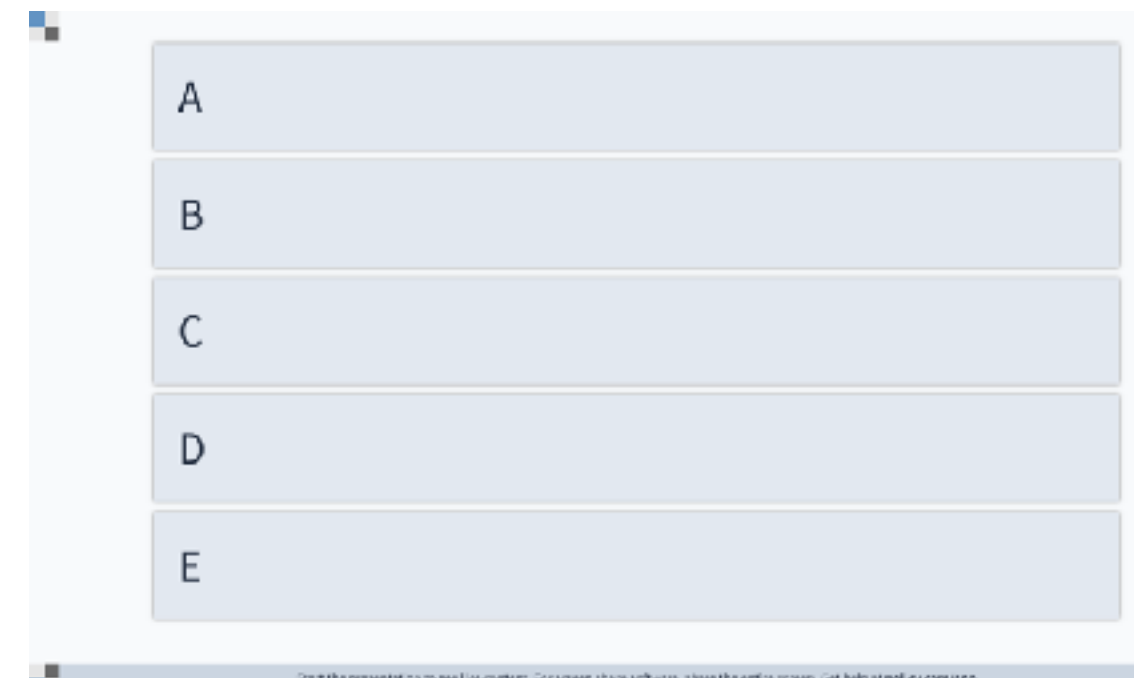
*if*  $A = 1$

$$C = 4KB$$



## 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

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

$$C = ABS$$

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

$$A = 8$$

Exactly how Core i7 configures  
its own cache



# About midterm

# Format of the midterm

- Multiple choices \* 15 — like your poll/reading quizzes multiple choices questions
- Short answer questions \* 4
- Homework style correctness questions \* 3
  - You need to clearly write down the original form of the applied equation/formula
  - You need to replace each term accordingly with numbers
  - You will have some credits for right equations even though the final number isn't correct
  - You will receive 0 credits if we only see the numbers

# Sample Midterm

# 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

# RISC-V v.s. x86

- Using the same language, the same source code, regarding the compiled program on x86 and RISC-V, how many of the following statements is/are "generally" correct?
    - ① The RISC-V version would contain more instructions than its x86 version
    - ② The RISC-V version tends to incur fewer memory accesses than its x86 version
    - ③ The RISC-V version needs a processor with higher clock rate than its x86 version if the CPI of both versions are similar
    - ④ The RISC-V version needs a processor with lower CPI than its x86 version if the x86 processor runs at the same clock rate
- A. 0  
B. 1  
C. 2  
D. 3  
E. 4

# 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 each parallel piece 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 programmer affects performance?

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

How many can a **programmer** affect?

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

# 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 of A better than B?

- ① IC
- ② CPI
- ③ CT

A. 0

B. 1

C. 2

D. 3



# 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

# 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

# intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-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

## 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 system use 4K pages.
  - A. 32B blocks, 2-way
  - B. 32B blocks, 4-way
  - C. 64B blocks, 4-way
  - D. 64B blocks, 8-way

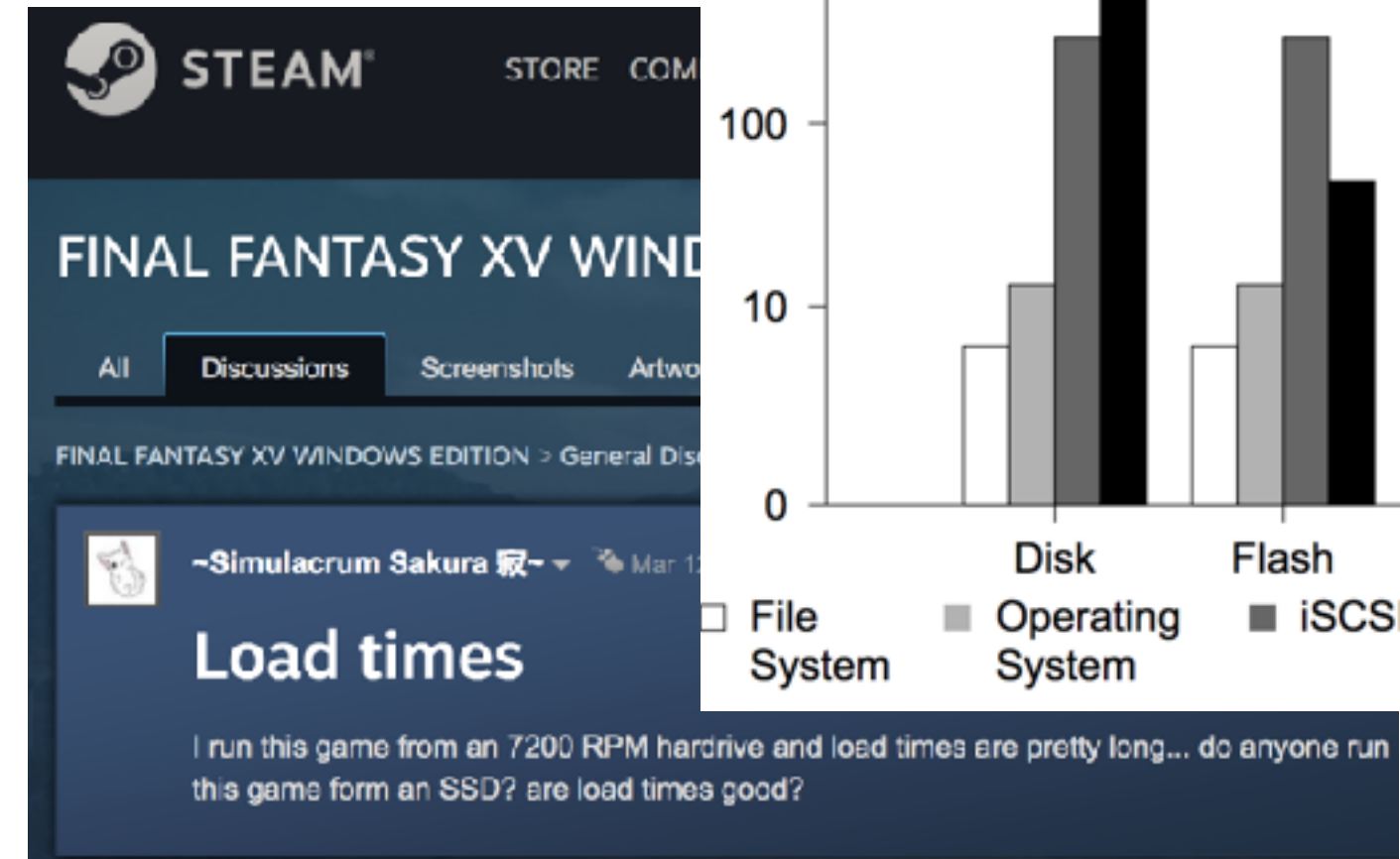
# When we have virtual memory...

- In a modern x86-64 processor supports virtual memory through, how many memory accesses can an instruction incur?
  - A. 2
  - B. 4
  - C. 6
  - D. 8
  - E. More than 10

# Practicing Amdahl's Law (2)

- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time is on 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?

- A. ~7x
- B. ~10x
- C. ~17x
- D. ~29x
- E. ~100x



# What data structure is performing better

	Array of objects	object of arrays
	<pre>struct grades {     int id;     double *homework;     double average; };</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; };</pre>
average of each homework	<pre>for(i=0;i&lt;homework_items; i++) {     gradesheet[total_number_students].homework[i] = 0.0;     for(j=0;j&lt;total_number_students;j++)         gradesheet[total_number_students].homework[i]             +=gradesheet[j].homework[i];     gradesheet[total_number_students].homework[i] /=         (double)total_number_students; }</pre>	<pre>for(i = 0;i &lt; homework_items; i++) {     gradesheet.homework[i][total_number_students] = 0.0;     for(j = 0; j &lt;total_number_students;j++)     {         gradesheet.homework[i][total_number_students] +=             gradesheet.homework[i][j];     }     gradesheet.homework[i][total_number_students] /=         total_number_students; }</pre>

- Considering your workload would like to calculate the average score of **one of the homework** for **all students**, which data structure would deliver better performance?
  - A. Array of objects
  - B. Object of arrays

# 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



# The result of `sizeof(struct struct_8)`

- Consider the following data structure:

```
struct struct_8 {  
    uint8_t a;  
    uint64_t b;  
    uint8_t c;  
} ;
```

What's the output of  
`printf( "%lu\n", sizeof(struct struct_8) )`?

- A. 10
- B. 17
- C. 24
- D. 36
- E. 80

# Pipelined access and multi-banked caches

- Assume each bank in the \$ takes 10 ns to serve a request, and the \$ can take the next request 1 ns after assigning a request to a bank — if we have 4 banks and we want to serve 4 requests, what's the speedup over non-banked, non-pipelined \$? — pick the closest one
  - A. 1x — no speedup
  - B. 2x
  - C. 3x
  - D. 4x
  - E. 5x

# Sample short answer questions (< 30 words)

- What are the limitations of compiler optimizations? Can you list two?
- Please define Amdahl's Law and explain each term in it
- Please define the CPU performance equation and explain each term.
- Can you list two things affecting each term in the performance equation?
- What's the difference between latency and throughput? When should you use latency or throughput to judge performance? When will throughput mislead performance?
- What's "benchmark" suite? Why is it important?
- Why TFLOPS or inferences per second is not a good metrics?

# Amdahl's Law for multiple optimizations

- Assume that a program is mainly composed of 2 functions — `baseline_int()` and `matrix()`. If the program takes 60% in `baseline_int()` and remaining 30% in `matrix()`.
  - If there exists an optimization that speedup `baseline_int` by 10x. What's the total speedup?
  - If there exists an optimization that speedup `matrix` by 30x. What's the total speedup?
  - What if we can apply both optimizations?

# 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	3 GHz	5000000000	20%	8	20%	4	60%	1
Machine Y	5 GHz	5000000000	20%	13	20%	4	60%	1

# How can deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has “perfect” memory, the CPI is just 1. Now, in addition to DDR4, whose latency 26 ns, we also got a 2-level SRAM caches with
  - it's 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5ns and can capture 60% of the desired data/instructions

What's the average CPI?

# Cache simulation

- The processor has a 32KB, 64B blocked, 4-way L1 cache. Consider the following code:

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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 total miss rate? How many of the misses are compulsory misses? How many of the misses are conflict misses?
- How can you improve the cache performance of the above code through changing hardware?
- How can you improve the performance **without** changing hardware?

# Announcement

- Assignment #2 due this Friday
- Midterm next Monday
  - You can take exam for any 80-minute slot between 12p 10/30/2022 and 12p 10/31/2022
  - You may review/focus on the materials/topics covered in lectures
  - You SHOULD review your assignments
  - Cover topics including today