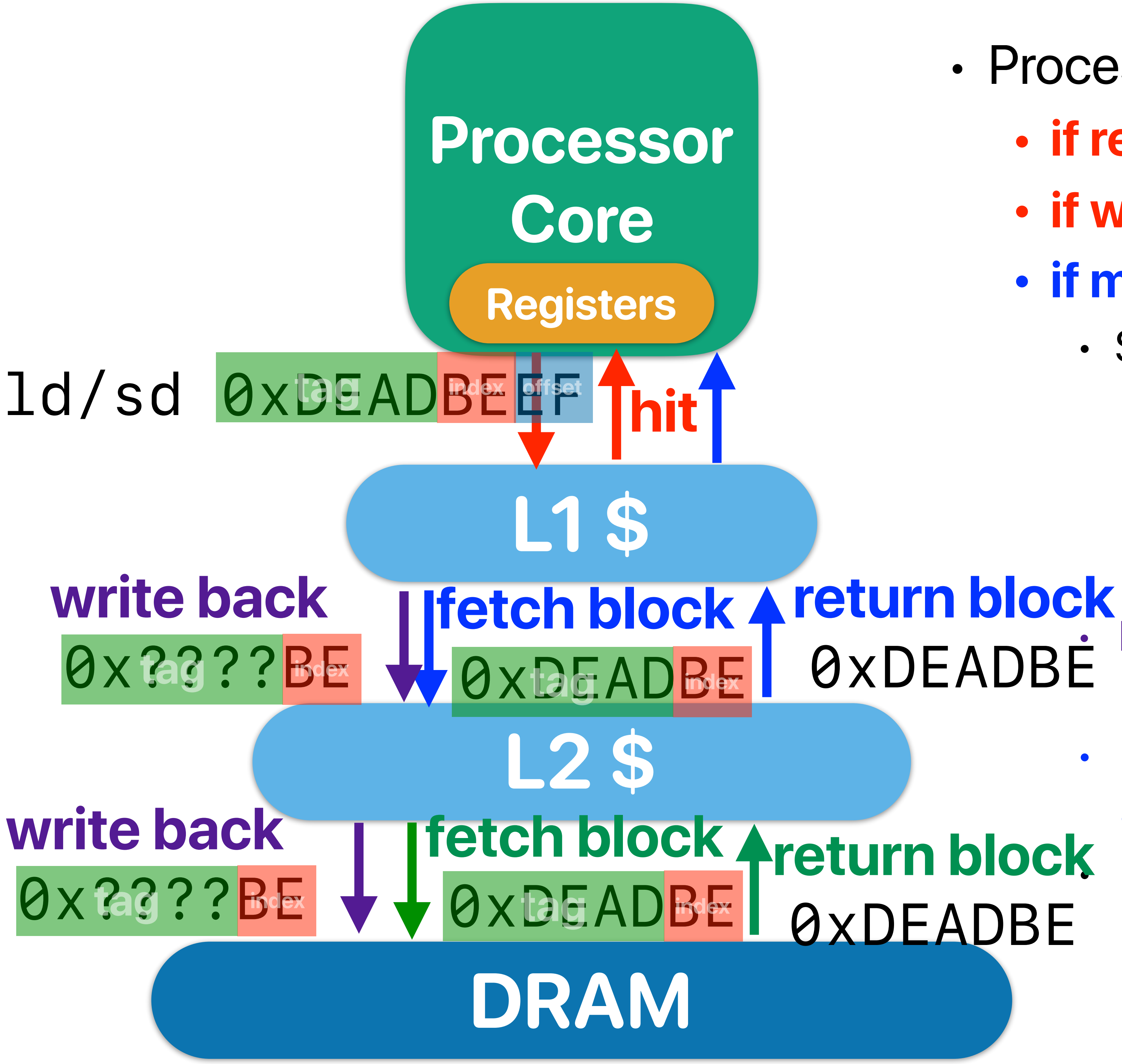
Virtual memory & memory hierarchy

Hung-Wei Tseng

Recap: What happens when we access data



- Processor sends load request to L1-$
  - **if read hit — return data**
  - **if write hit — set dirty and update in the block**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process

Recap: causes of $ misses

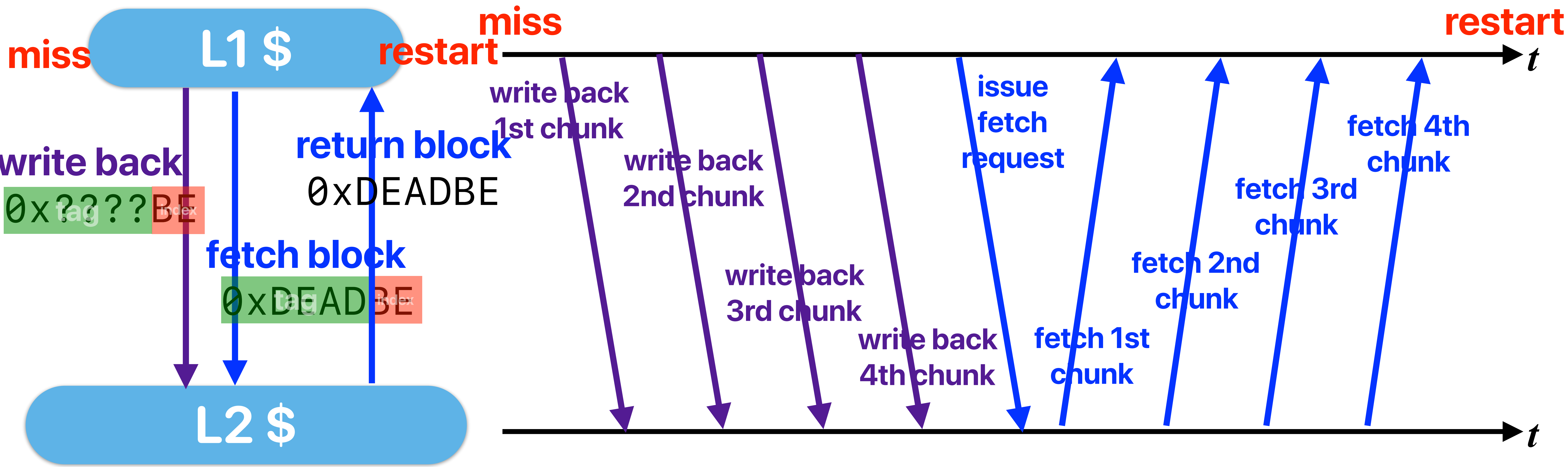
- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data block replaced by block(s) mapping to the same set
  - Similar collision in hash — if the conflict miss doesn't go away even though you made the cache fully-associative — it's a capacity miss

Recap: optimizations

- Software
  - Data layout — capacity miss, conflict miss, compulsory miss
  - Blocking — capacity miss, conflict miss
  - Loop fission — conflict miss — when $ has limited way associativity
  - Loop fusion — capacity miss — when $ has enough way associativity
  - Loop interchange — conflict/capacity miss
- Hardware
  - Prefetch — compulsory miss

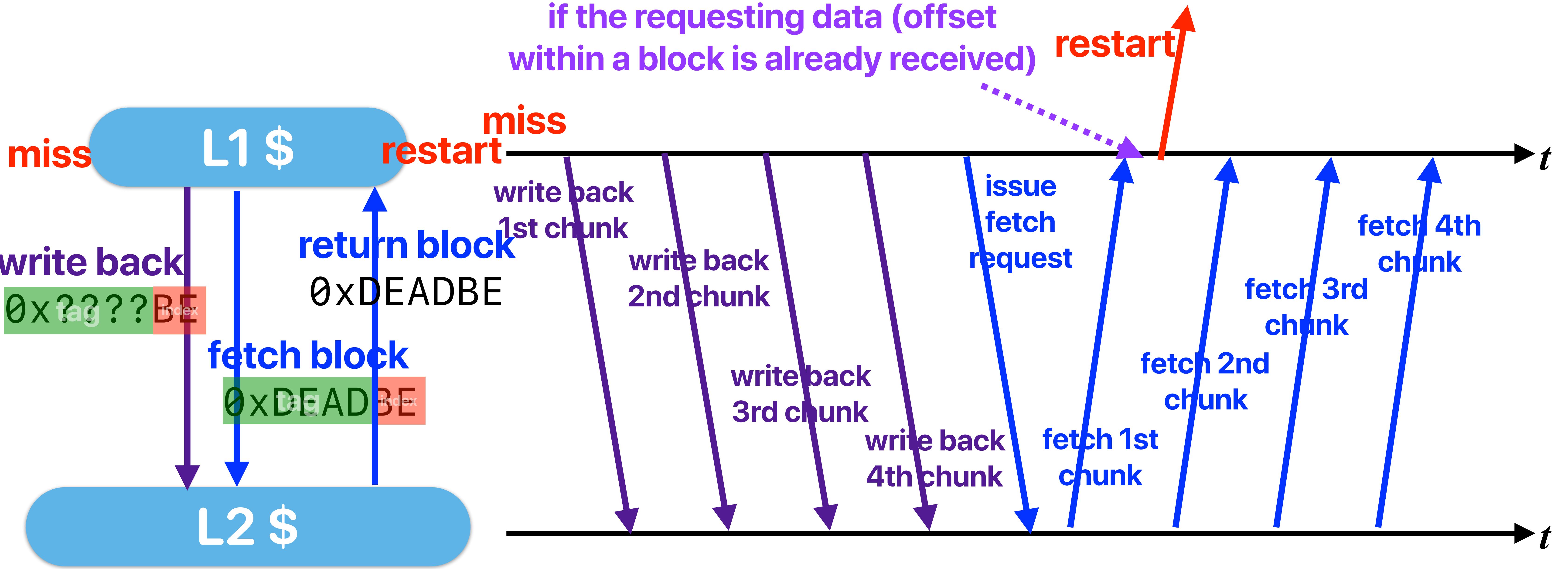
Cache Optimizations

When we handle a miss



assume the bus between L1/L2 only allows a quarter of the cache block go through it

Early Restart and Critical Word First

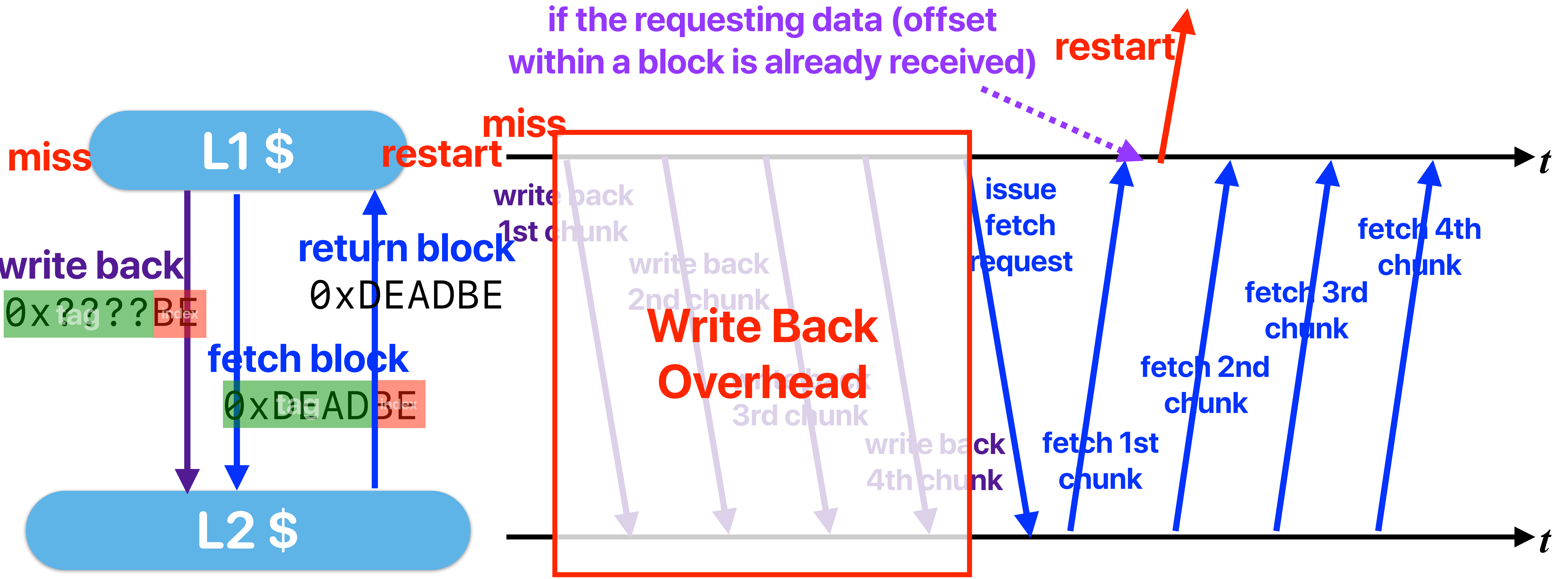


assume the bus between L1/L2 only allows a quarter of the cache block go through it

Early Restart and Critical Word First

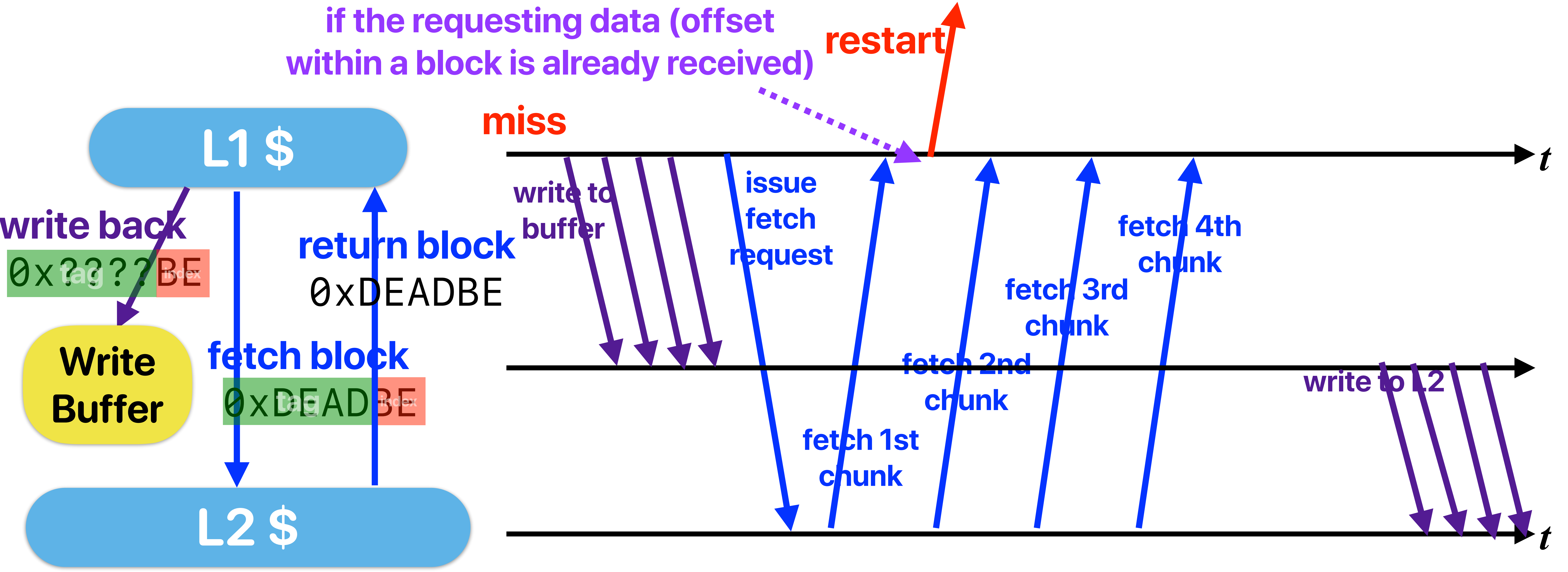
- Don't wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Most useful with large blocks
- Spatial locality is a problem; often we want the next sequential word soon, so not always a benefit (early restart).

Can we avoid the overhead of writes?



assume the bus between L1/L2 only allows a quarter of the cache block go through it

Write buffer!



assume the bus between L1/L2 only allows a quarter of the cache block go through it

Can we avoid the "double penalty"?

- Every write to lower memory will first write to a small SRAM buffer.
  - store does not incur data hazards, but the pipeline has to stall if the write misses
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can response as soon as the data is written to write buffer.
- Write merge
  - Since application has locality, it's highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.

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

- Software
  - Data layout — capacity miss, conflict miss, compulsory miss
  - Blocking — capacity miss, conflict miss
  - Loop fission — conflict miss — when $ has limited way associativity
  - Loop fusion — capacity miss — when $ has enough way associativity
  - Loop interchange — conflict/capacity miss
- Hardware
  - Prefetch — compulsory miss
  - Write buffer — miss penalty
  - Bank/pipeline — miss penalty
  - Critical word first and early restart — miss panelty

Recap: Virtual memory

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 see
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, 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)

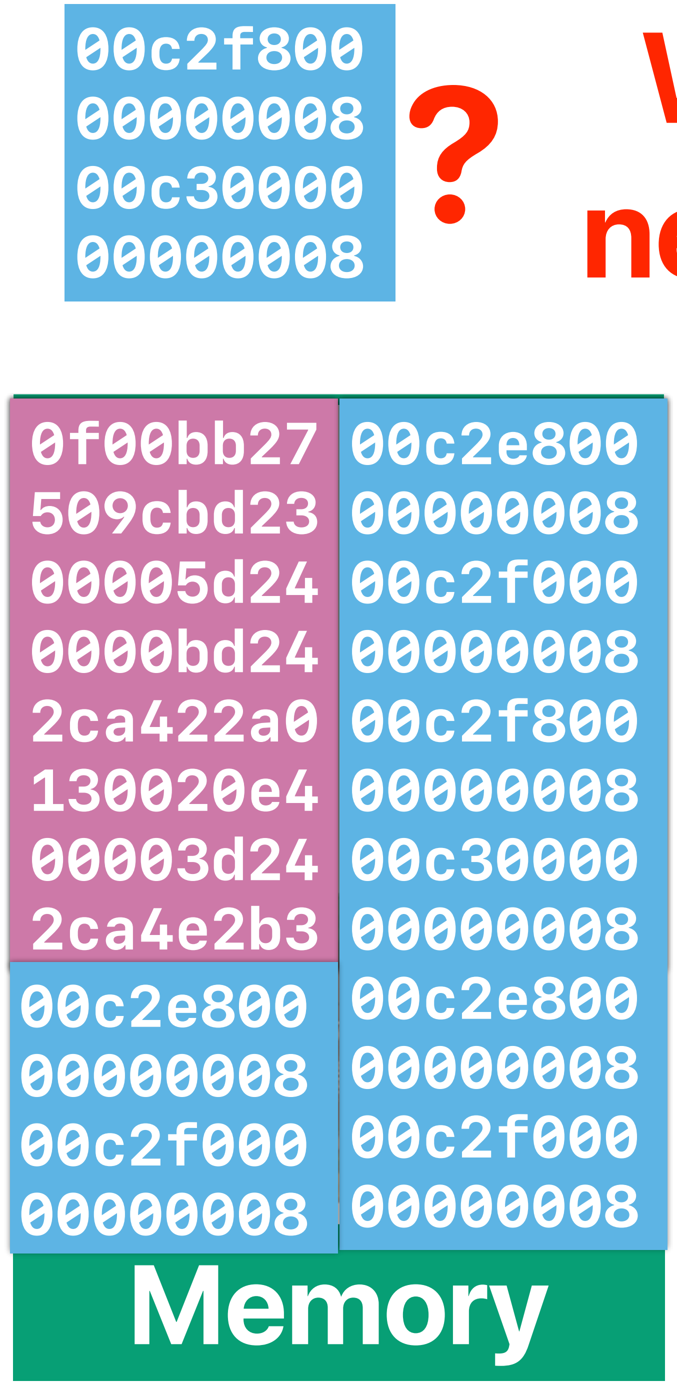
```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

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 is using CPU: %d. Value of a is 
%lf and address of a is %p\n",getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is 
%lf and address of a is %p\n",getpid(), cpu, a, &a);
    return 0;
}
```

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

If we expose memory directly to the processor (I)



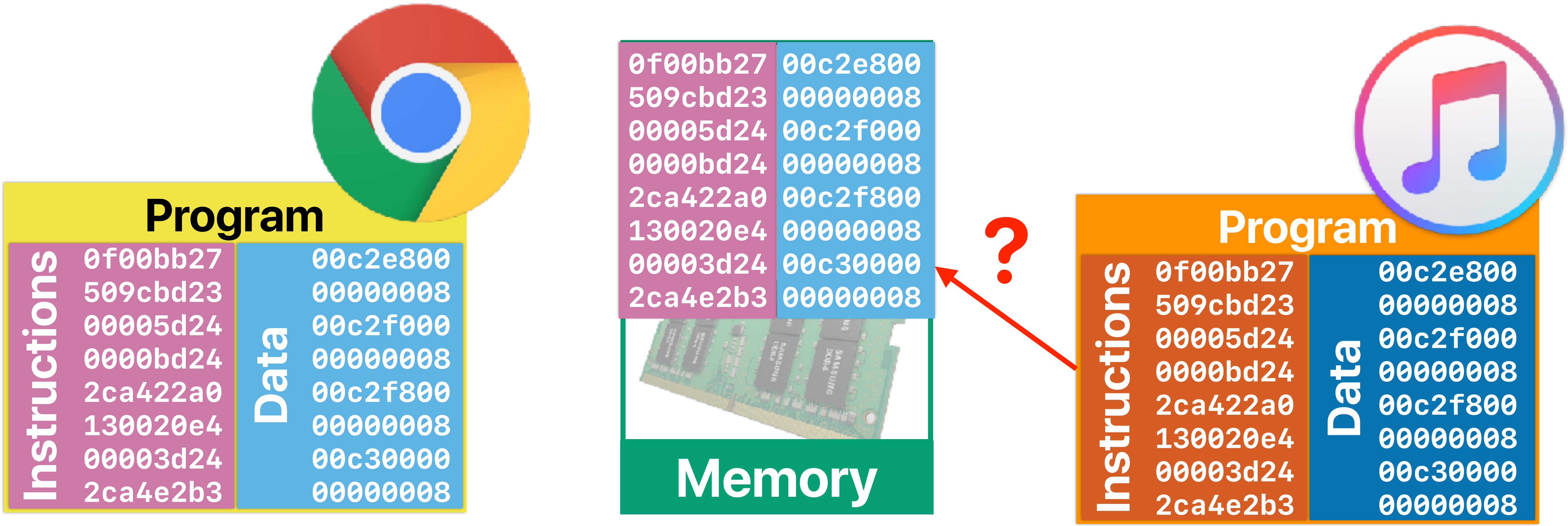
What if my program needs more memory?

If we expose memory directly to the processor (II)

What if my program runs on a machine with a different memory size?

If we expose memory directly to the processor (III)

What if both programs need to use memory?

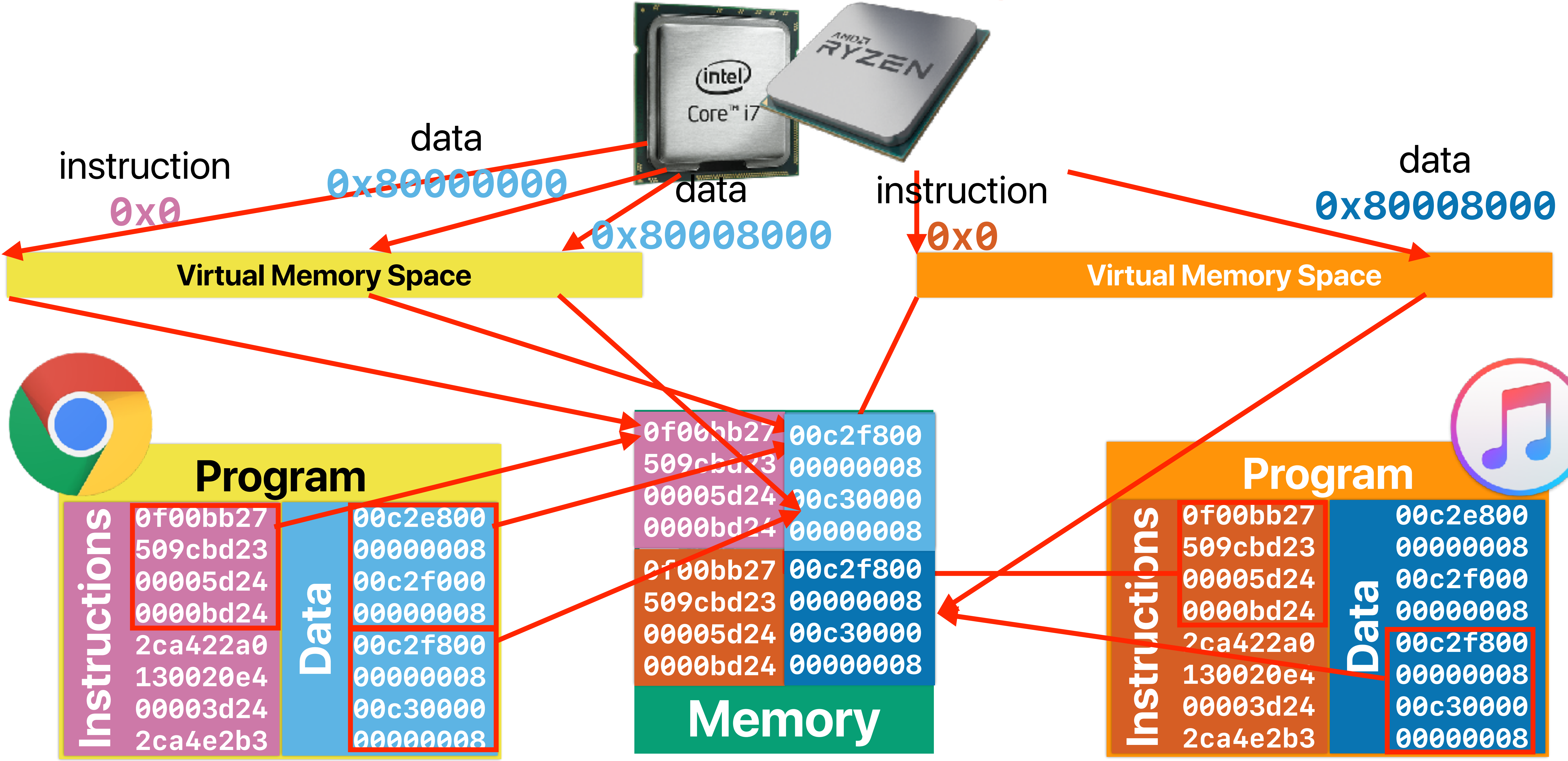


If we can only use physical memory …

- If there is no abstraction between the processor and memory, the processor/cache needs to directly using main memory's byte address to read/write data. How many of the following would be happening?
  ① The program's memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
  ② There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
  ③ Two programs cannot run simultaneously if they use the same memory addresses
  ④ One program can maliciously access data from other concurrently executing programs

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

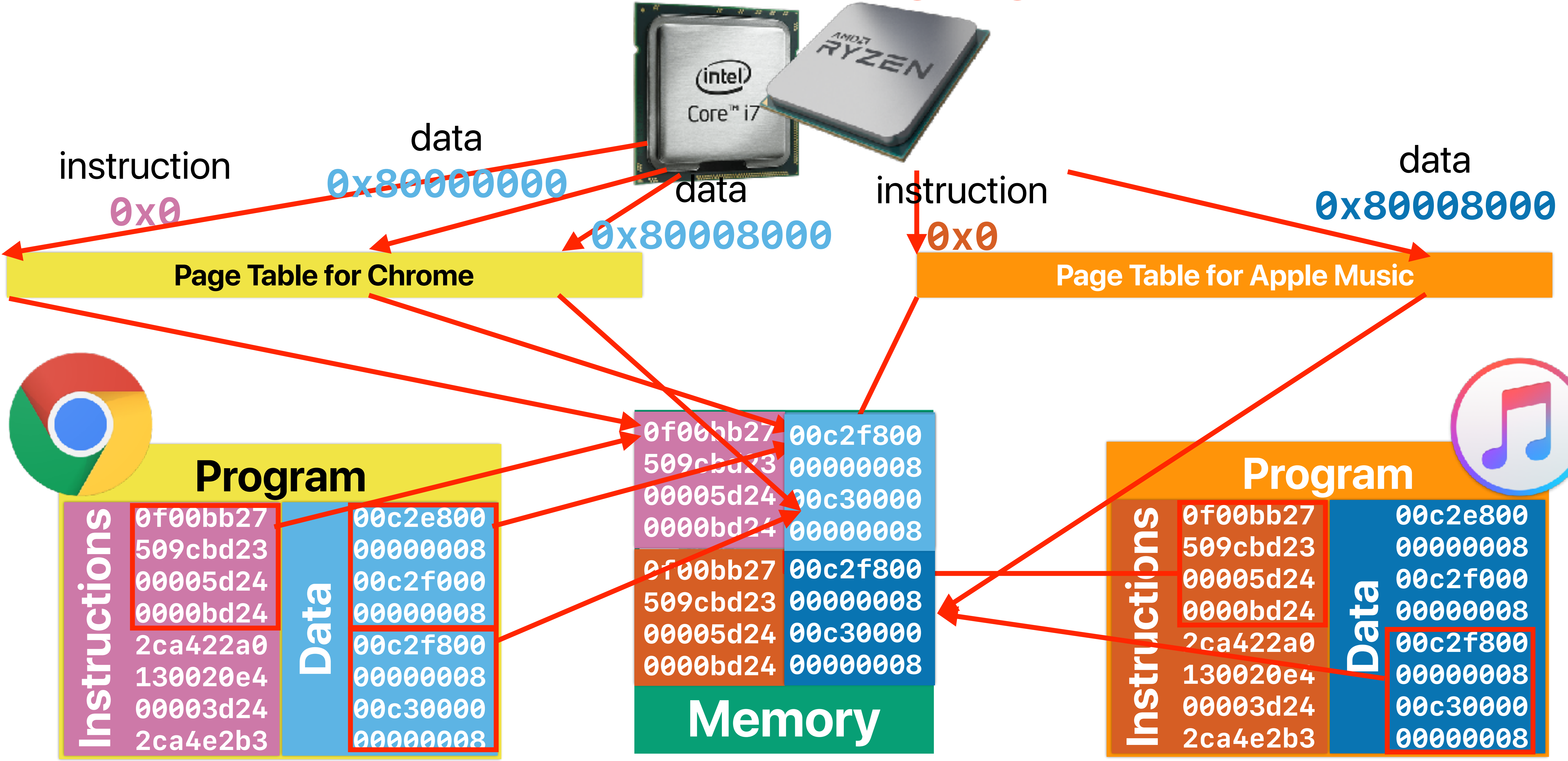
Virtual memory



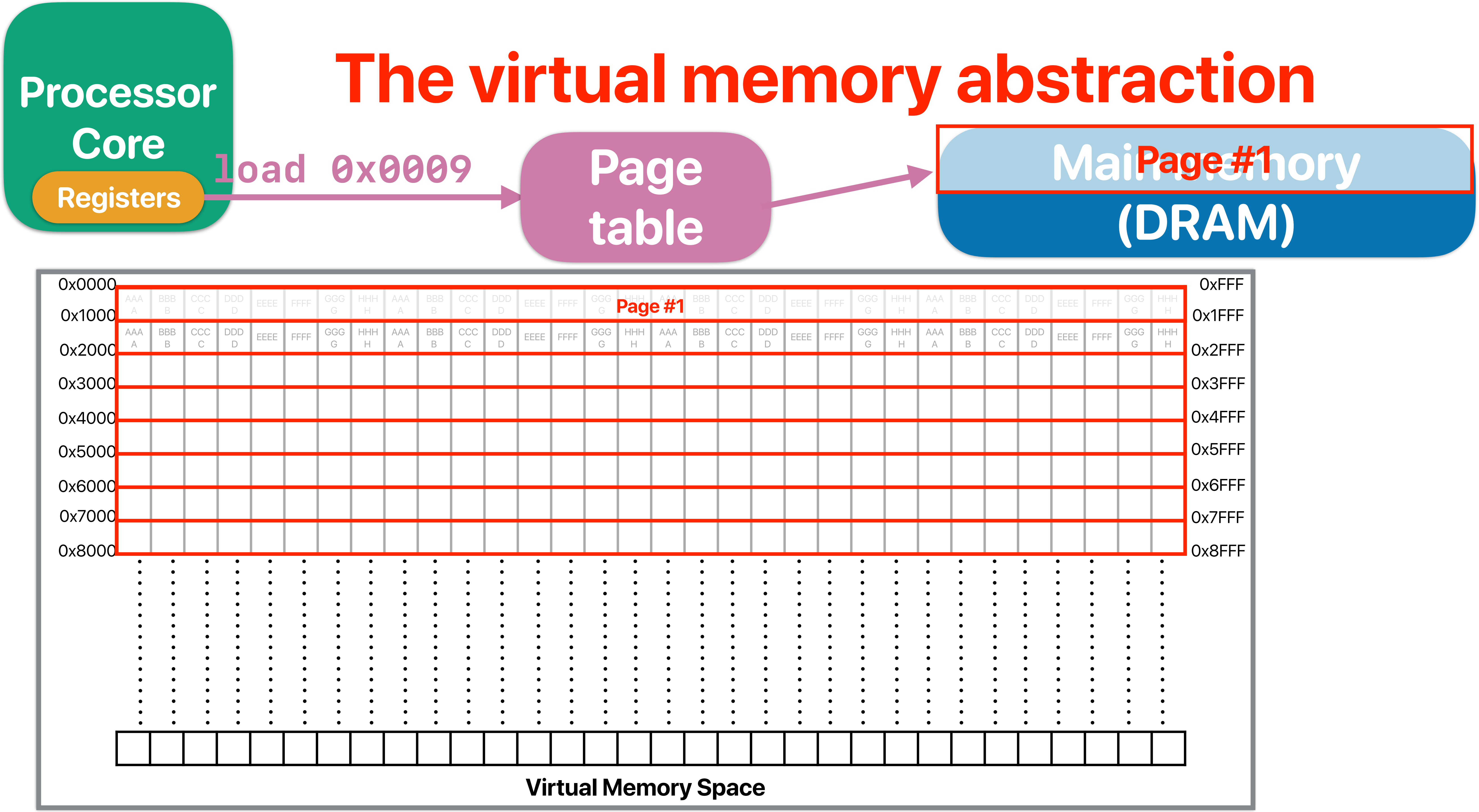
Virtual memory

- 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**”

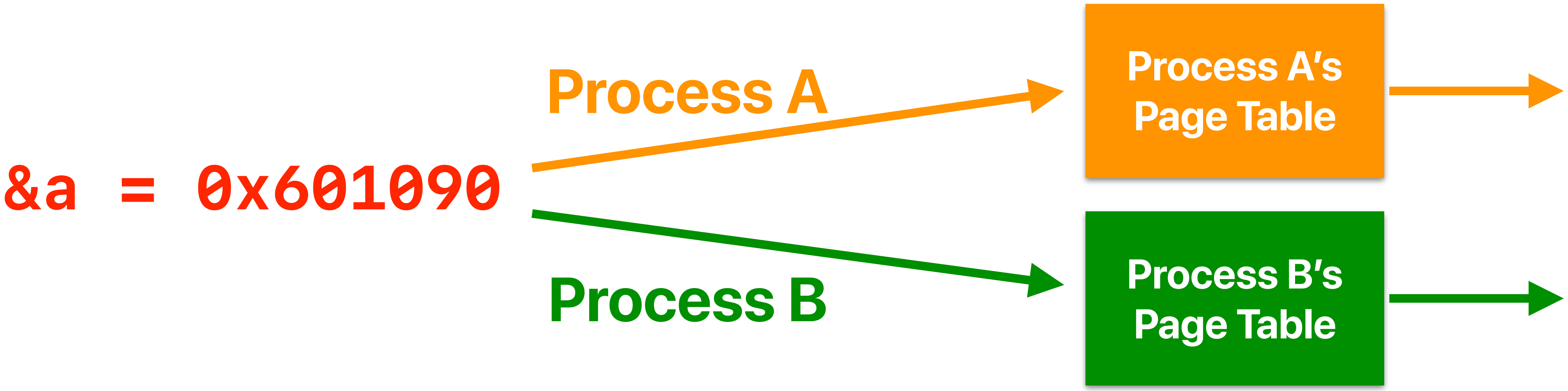
Demand paging



The virtual memory abstraction



Demo revisited



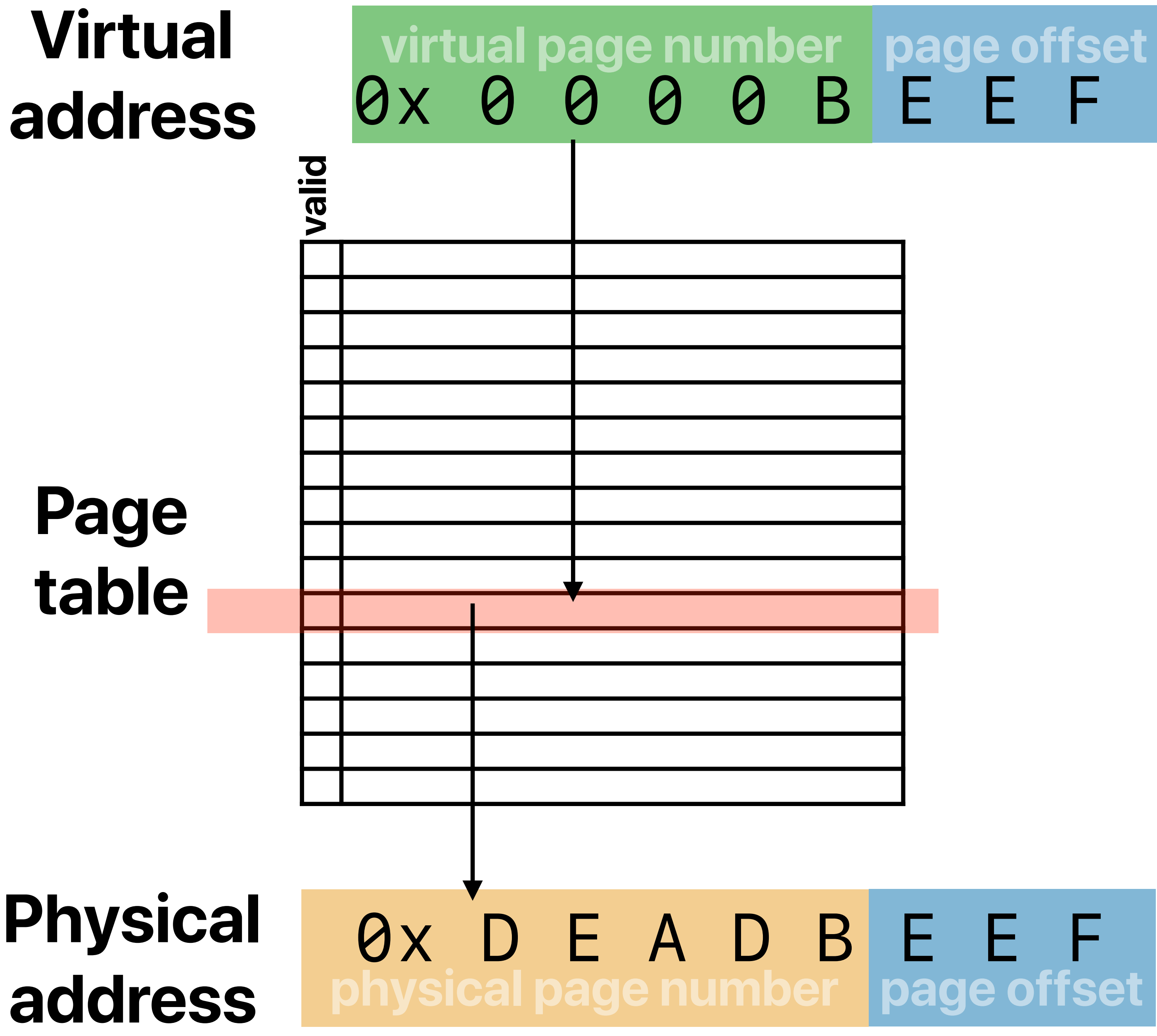

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

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 is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, a, &a);
    return 0;
}
```

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



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

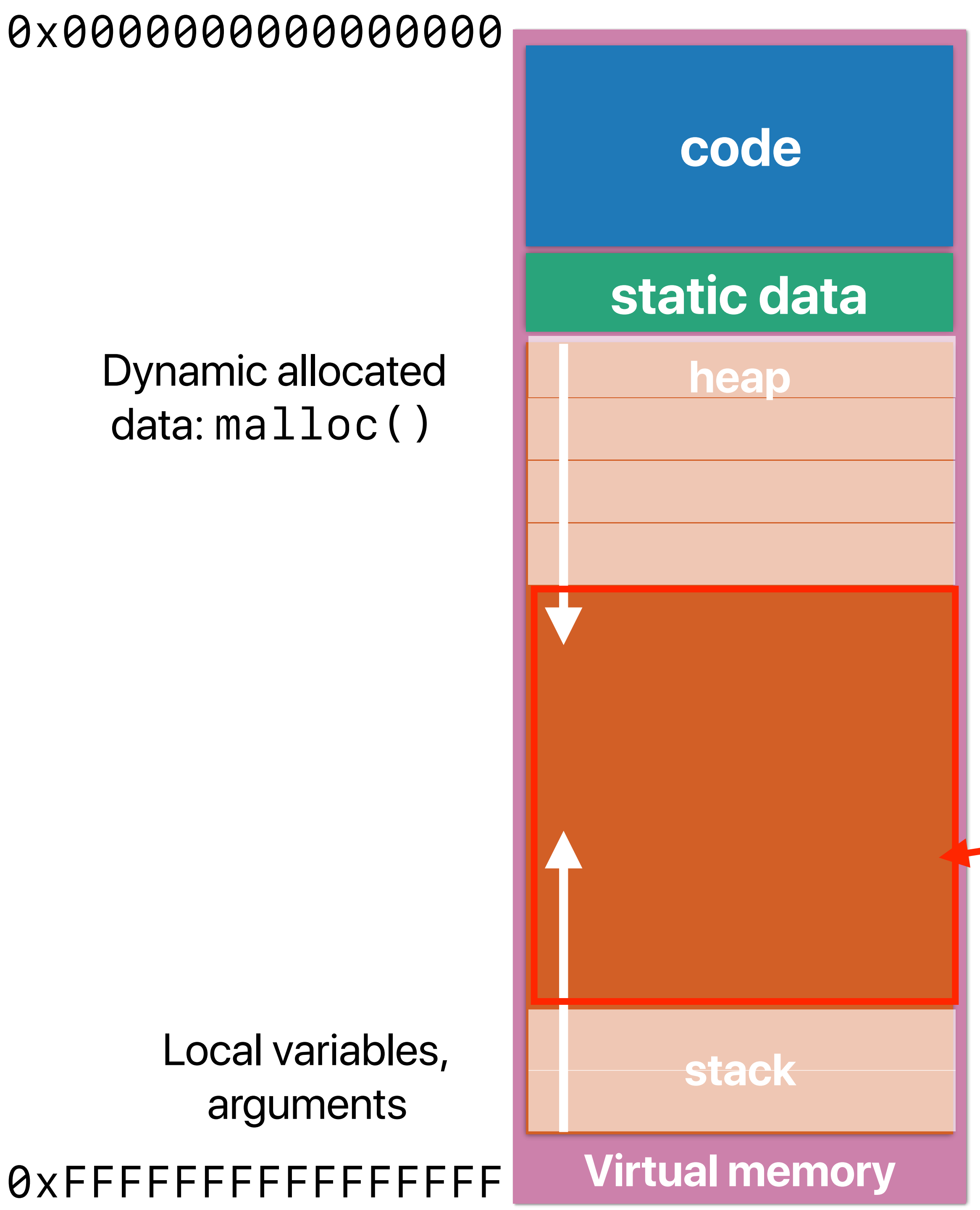
Size of page table

- 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

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

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

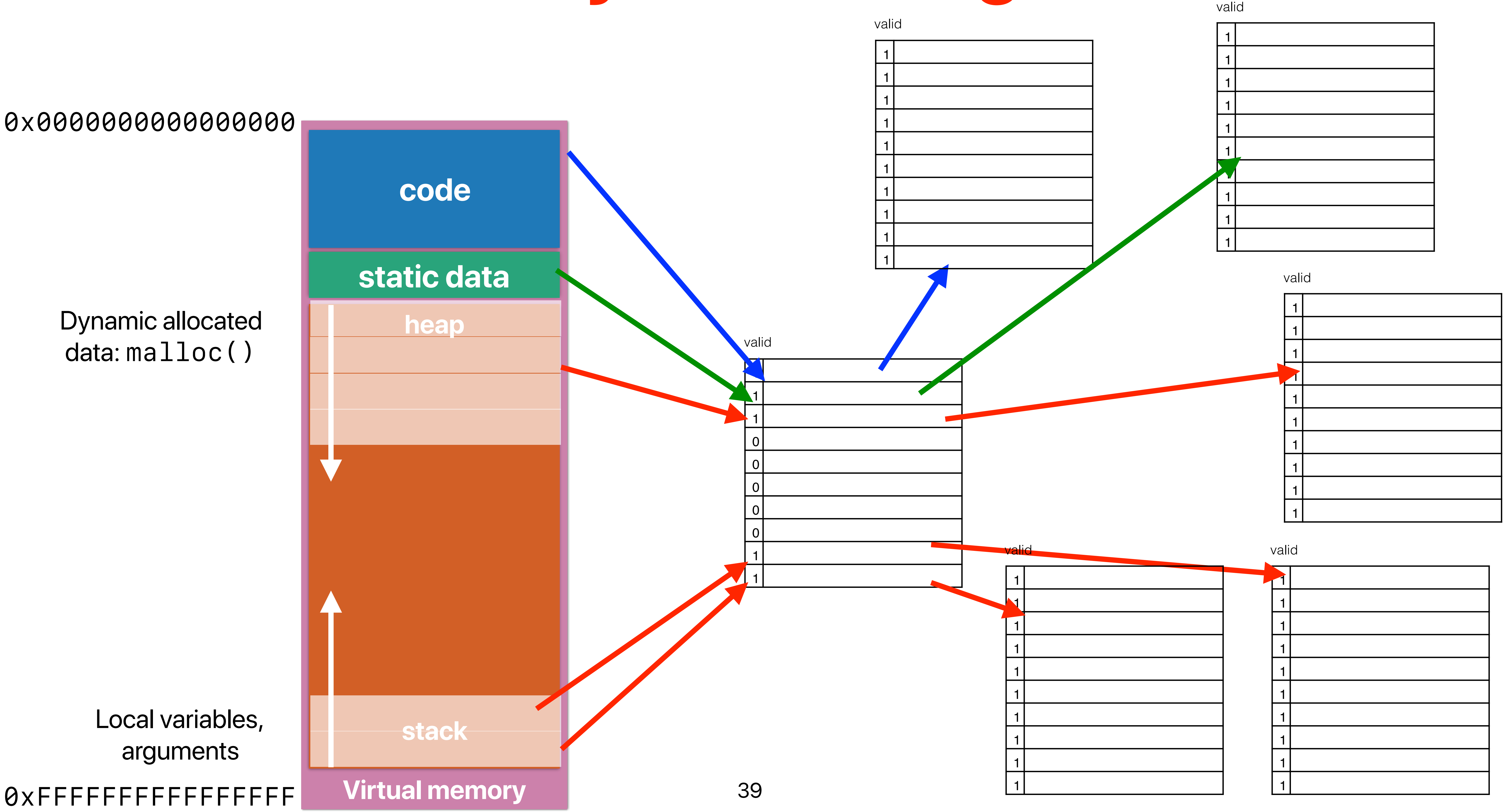
Do we really need a large table?



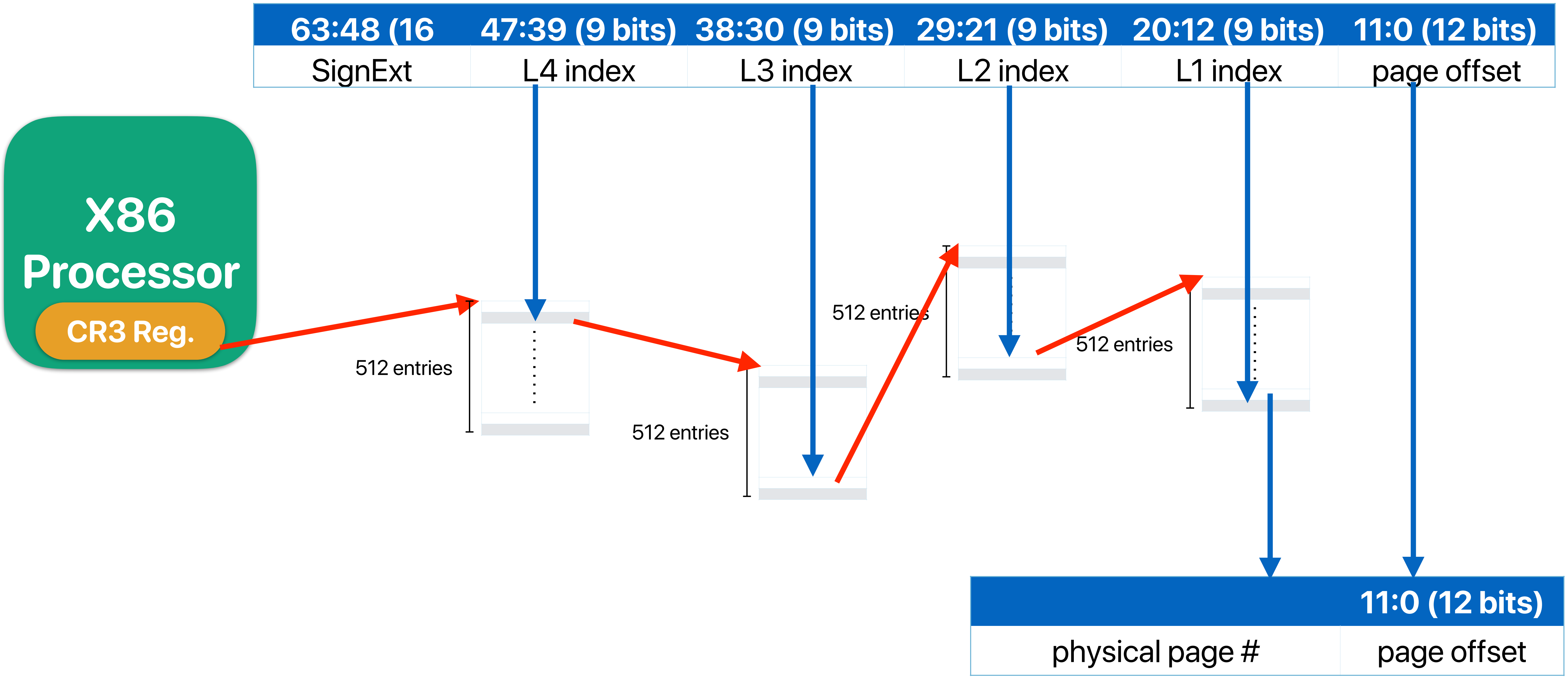
Your program probably never uses this huge area!

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

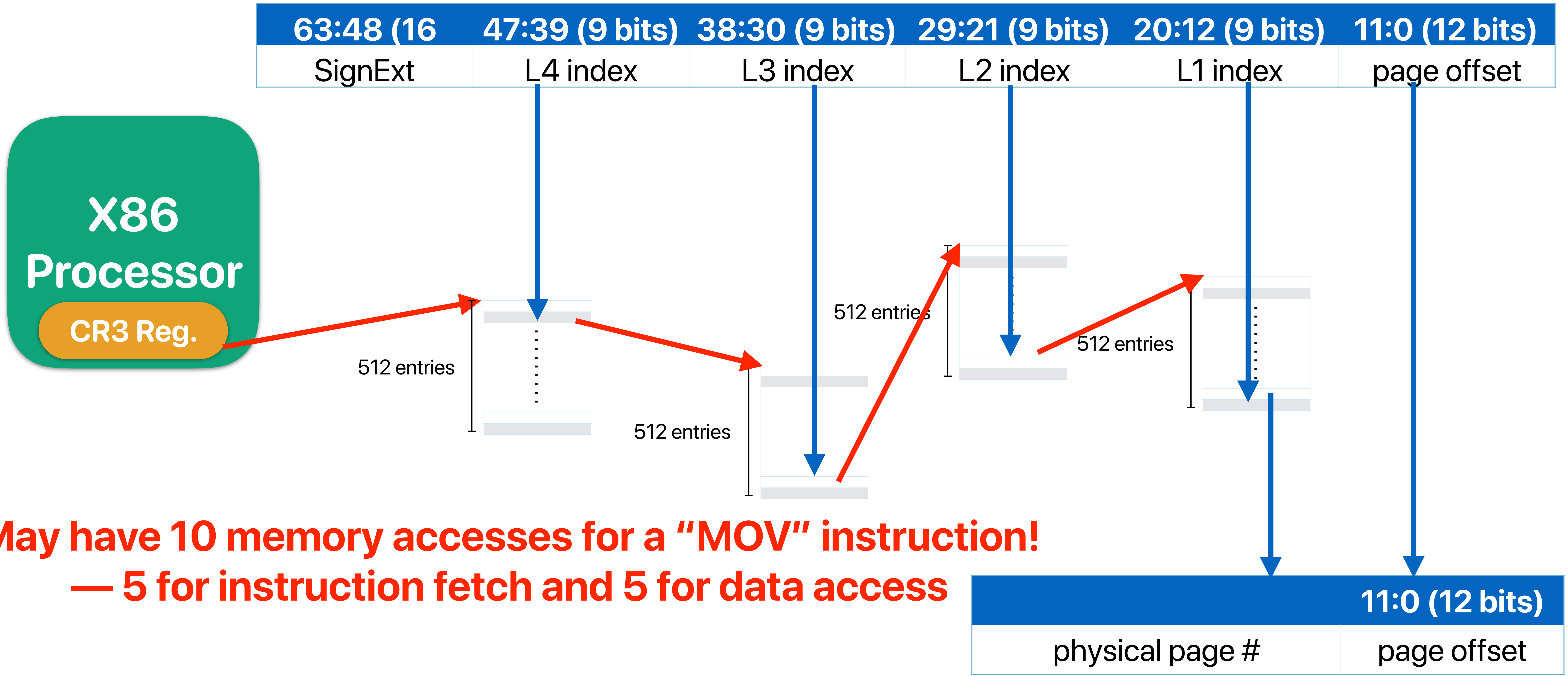
Do we really need a large table?



Address translation in x86-64



Address translation in x86-64



May have 10 memory accesses for a "MOV" instruction!

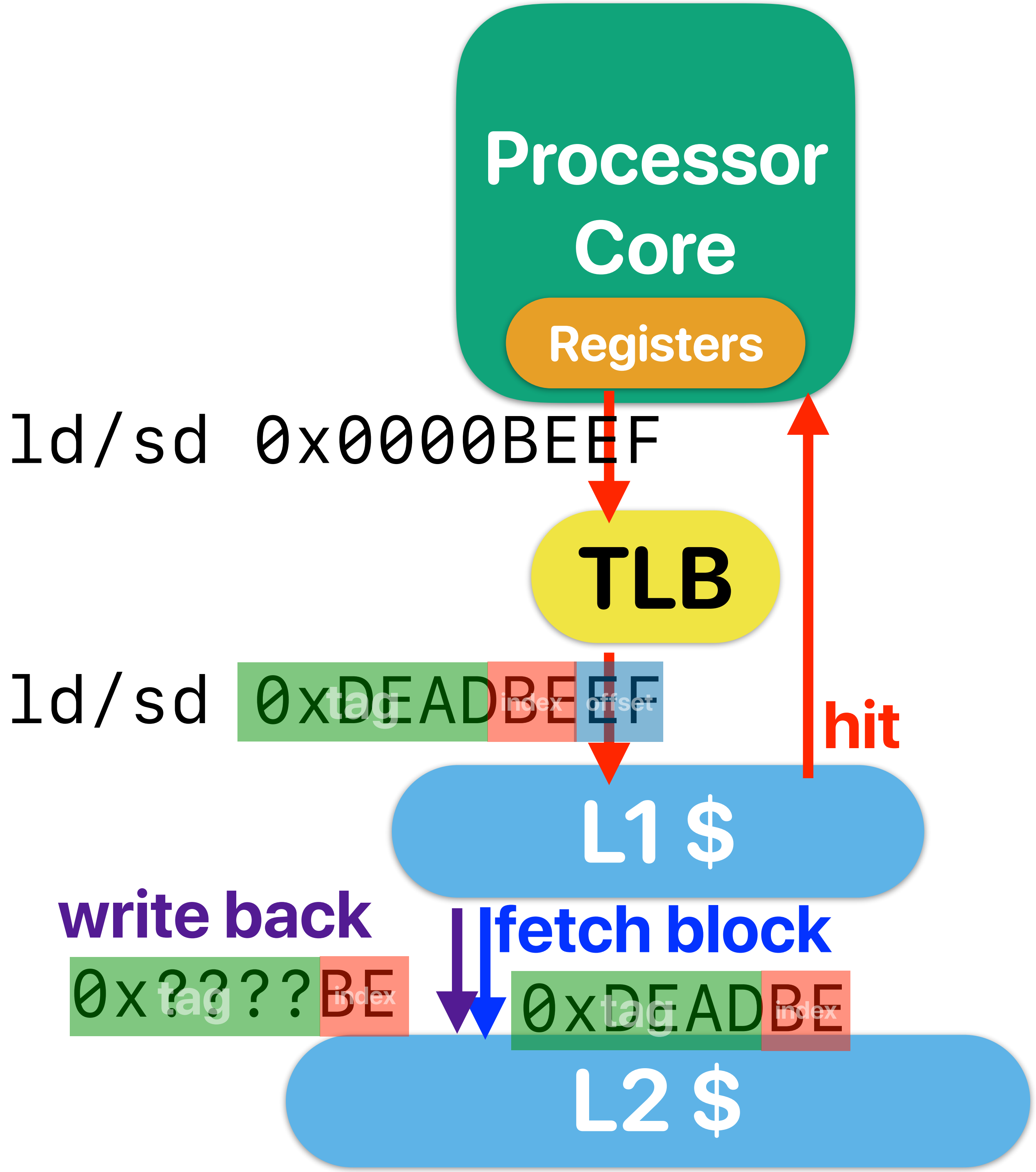
— 5 for instruction fetch and 5 for data access

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

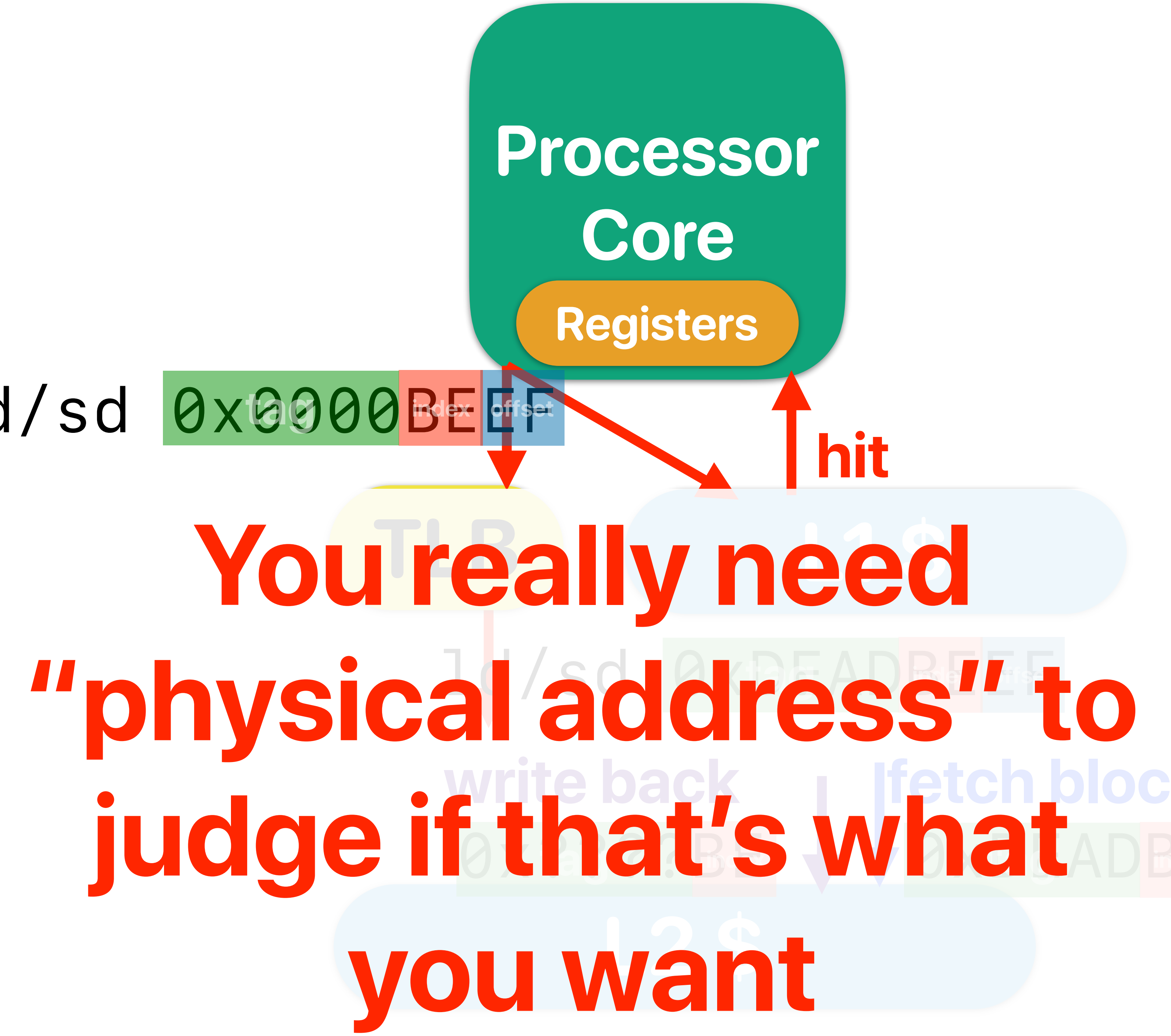
TLB: Translation Look-aside Buffer



- 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**



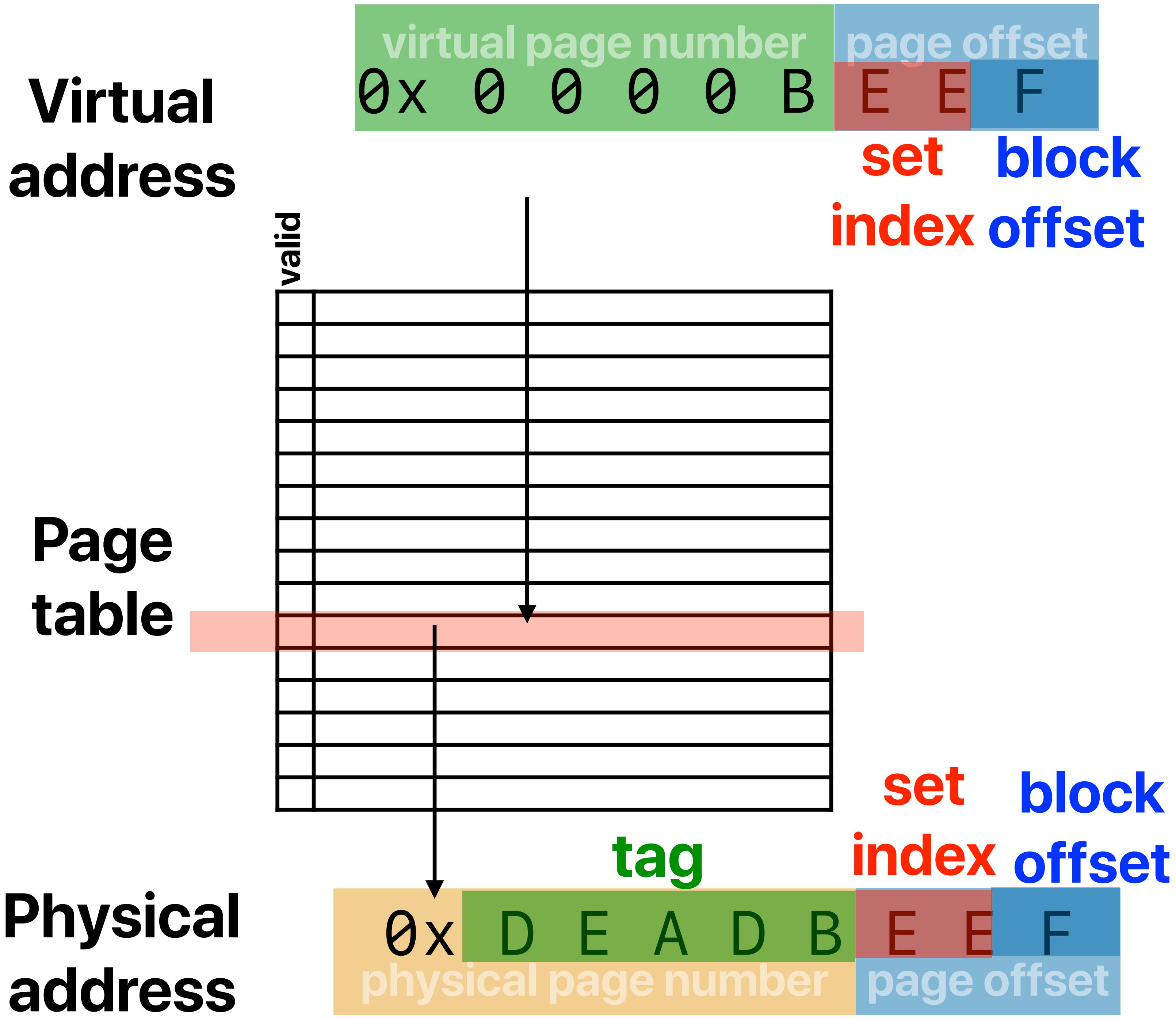
Virtually indexed, physically tagged cache

- 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

memory address: 0b000010000010 0100

virtual page # = 00001000 | set index = 0010 | block offset = 0100

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

0xA 1

=?

hit?

Virtually indexed, physically tagged cache

- If page size is 4KB —

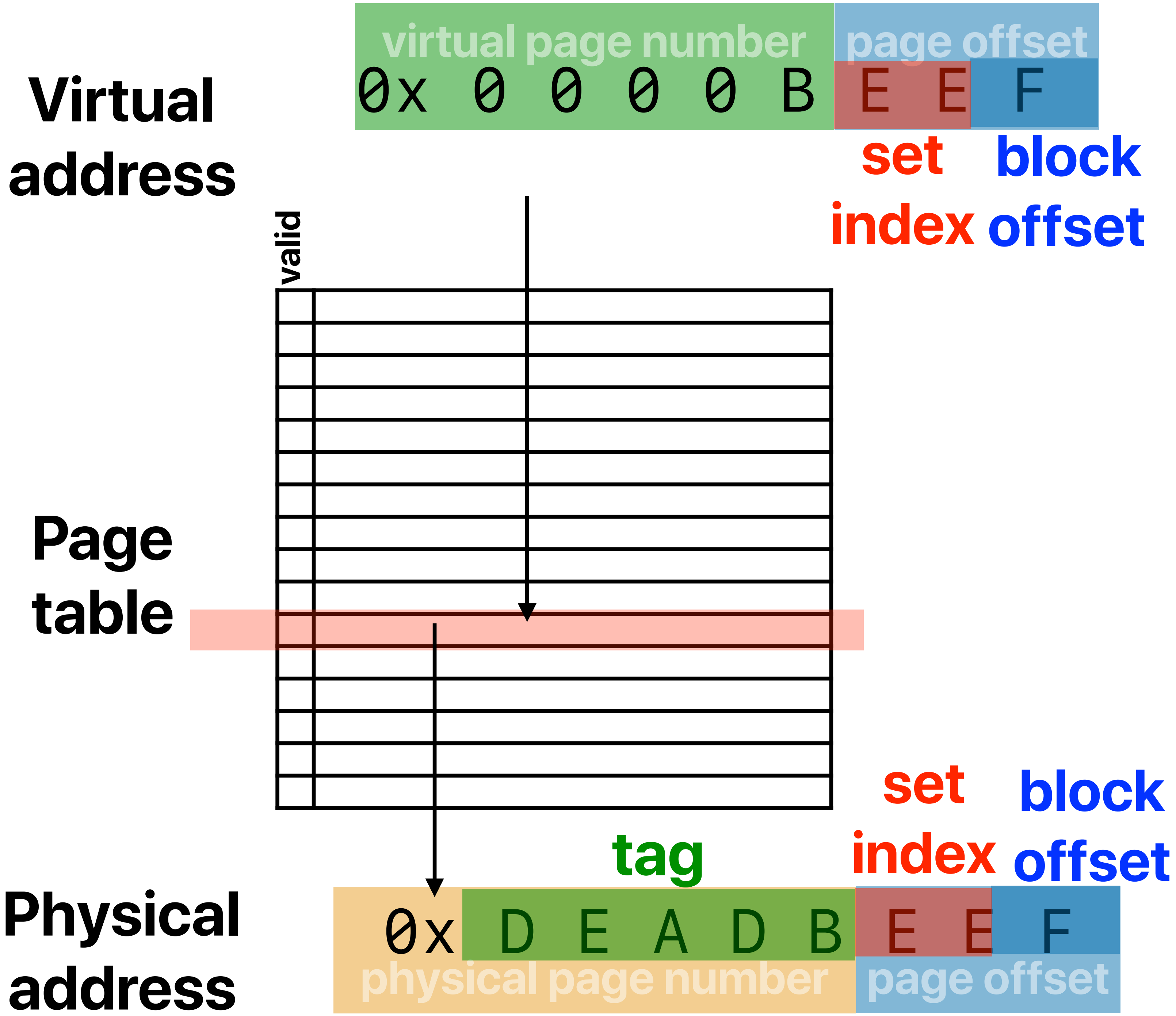
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

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

$$if\ A = 1$$

$$C = 4KB$$



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

Exactly how Core i7 configures its own cache

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

$$C = ABS$$

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

$$A = 8$$

Announcement

- Midterm next Monday
- Assignment #2 due tonight
- Office hour for Hung-Wei — back to MW 1p-2p