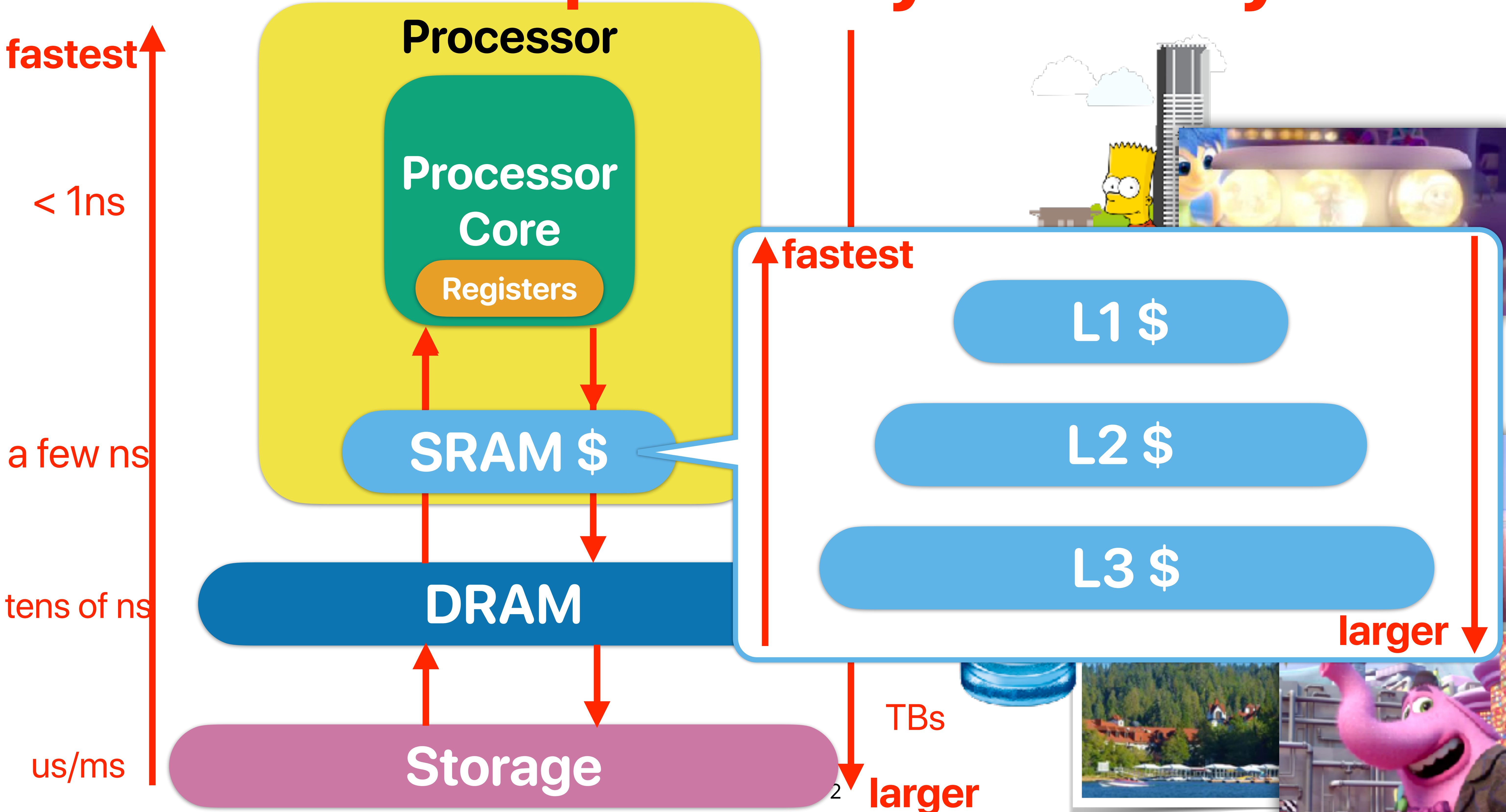
Lab 3: More aspects on memory

Hung-Wei Tseng

Recap: Memory Hierarchy



Recap: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Prefetching — compulsory misses

The concept of working set

Working set size

- The portion of memory that the program is currently using
- The total size of data blocks with “temporal locality”
- The total size of data blocks we have to visit before the next reuse of the current block

What’s the working set size of the code?

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(j = 0; j < runs; j++)
    for(i = 0; i < size; i+=stride) {
        sum += a[i];
        //load a, b, and then store to c
    }
}
```

What’s the data cache miss rate for this code when **stride = 8, runs = 4** and **size = 1024**?

Let’s pick the block with a[0], how many blocks do we have to visit before coming back?

We have to visit a total of $\frac{1024}{8} = 128\ blocks$

We need $128 \times 64 = 8\ KB$ **capacity to keep the “working set”**

What's the working set size of the code?

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(j = 0; j < runs; j++)
    for(i = 0; i < size; i+=stride) {
        sum += a[i];
        //load a, b, and then store to c
    }
}
```

What's the data cache miss rate for this code when **stride = 8, runs = 4** and **size = 8192**?

Let's pick the block with a[0], how many blocks do we have to visit before coming back?

We have to visit a total of $\frac{8192}{8} = 1024\ blocks$

We need $1024 \times 64 = 64\ KB$ **capacity to keep the "working set"**

How to reduce the working set?

How a “struct” is layout?

Thinking in cache lines

- How big is my_struct?
  - 16 bytes
  - 4 uint32_t's
  - 1 cache line?
  - 2 cache lines?

```
struct my_struct {
    uint32_t foo[4]
}
```

Q4: The result of `sizeof(struct student)`

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

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

A. 20

B. 28

C. 32

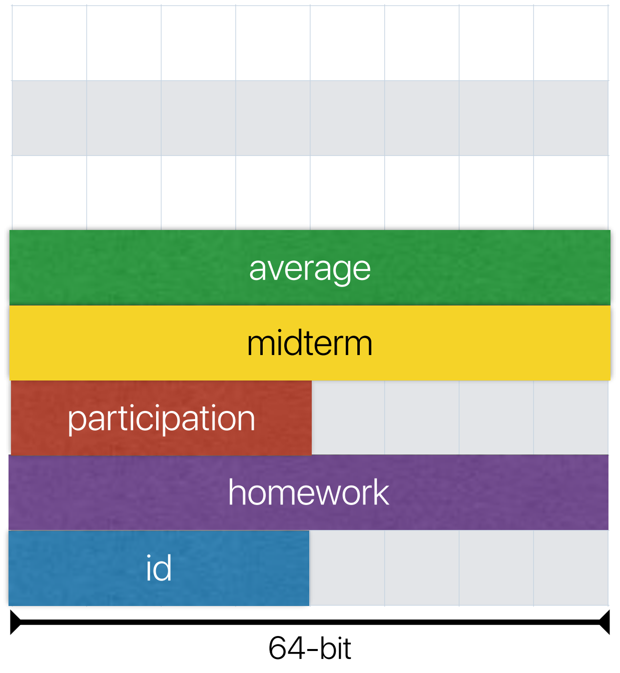
D. 36

E. 40

The result of `sizeof(struct student)`

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```



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

A. 20
B. 28
C. 32
D. 36
[✓ E. 40]

Memory addressing/alignment

- Almost every popular ISA architecture uses “byte-addressing” to access memory locations
- Instructions generally work faster when the given memory address is aligned
    - Aligned — if an instruction accesses an object of size n at address X, the access is **aligned** if **X mod n = 0**.
    - Potentially incurs two cache misses for an access
    - Some architecture/processor does not support aligned access at all
    - Therefore, compilers only allocate objects on “aligned” address

Memory addressing/alignment

- Unaligned accesses are sometimes inefficient: one load can cause 2 cache misses
- Some versions of ARM’s ISA have very complicated semantics for unaligned accesses
- Other ISAs
    - Unaligned access can cause an interrupt.

Compiler cannot optimize the layout

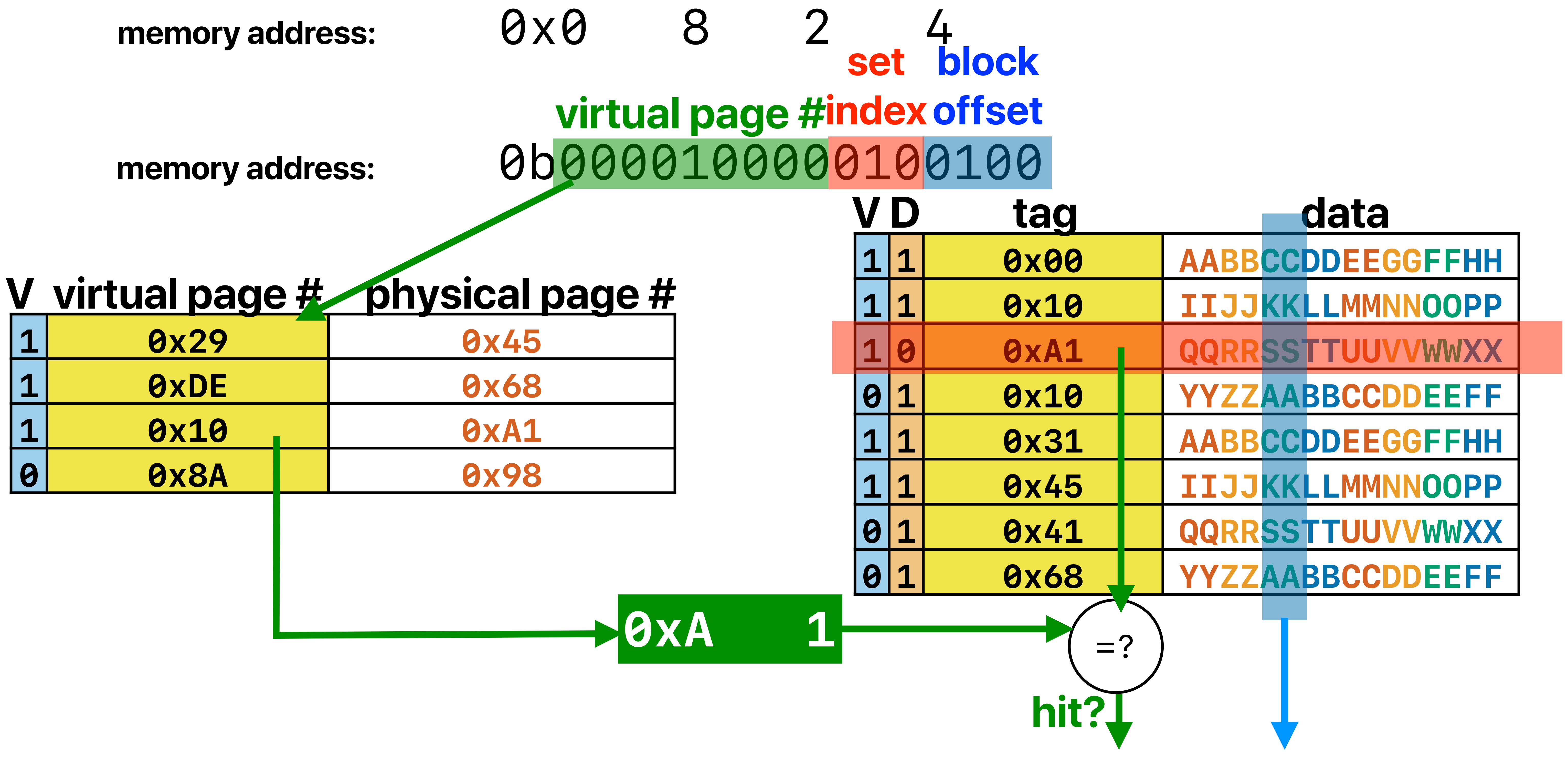
- The struct declaration must be shared among binaries use that struct — not everyone has the same optimization flags
- Accessing the struct element is base + offset in the struct
  - Rearranging without everyone's agreement would cause errors

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Prefetching — compulsory misses
- Data alignments — capacity misses

Remember, we also have virtual memory

Memory hierarchy in practice: TLB + Cache

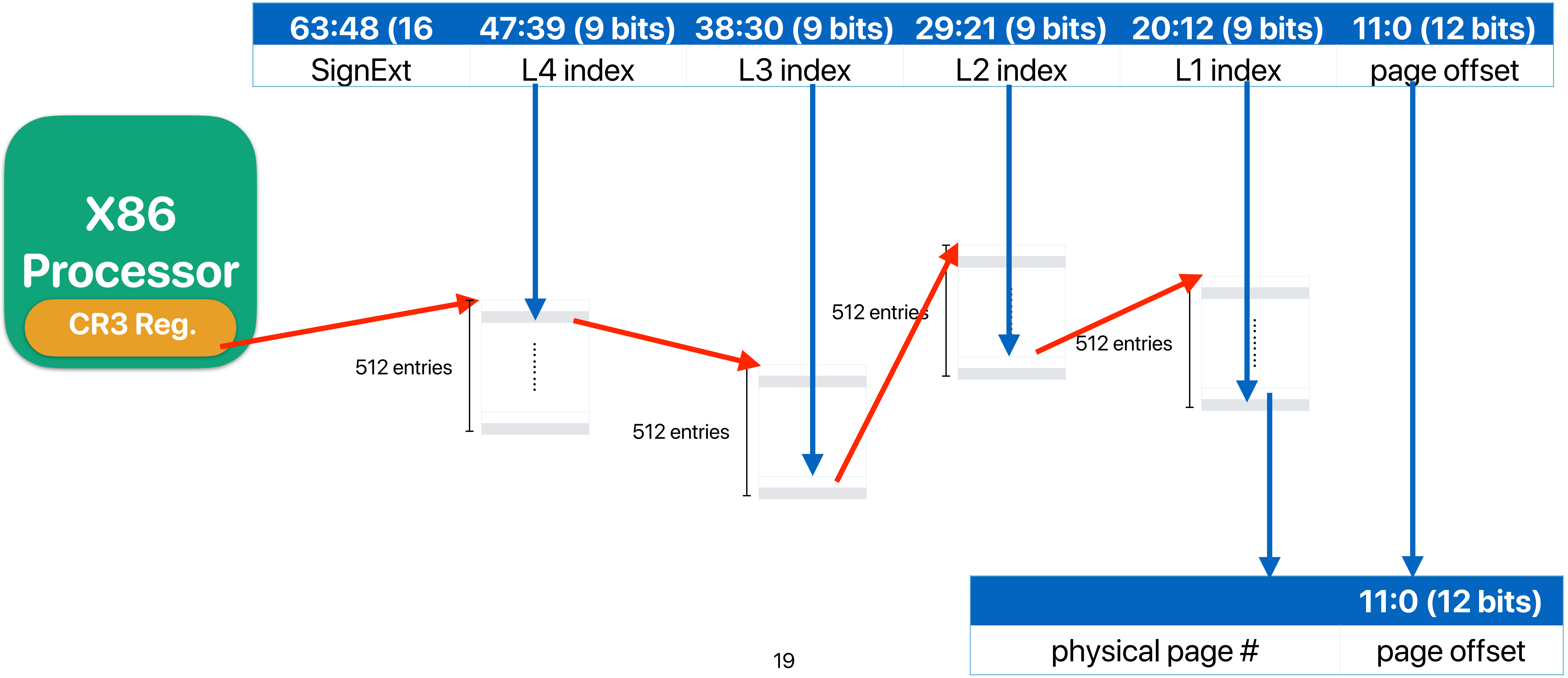

memory address: 0x0 8 2 4
set
block
virtual page #
index
offset
memory address: 0b0000100000 010 0100
0xA 1
=?
hit?

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

What happen on a TLB miss?

- Page table look up — how many memory accesses?



Cache miss, TLB miss, Page fault

Case	$	TLB	Virtual Memory
L1 $ Hit	Hit	Hit	No page fault
L1 $ Miss	Miss	Hit or Miss	May or may not have a page fault
TLB Miss	Miss	Miss	May or may not have a page fault
Page Fault	Miss	Miss	Page fault

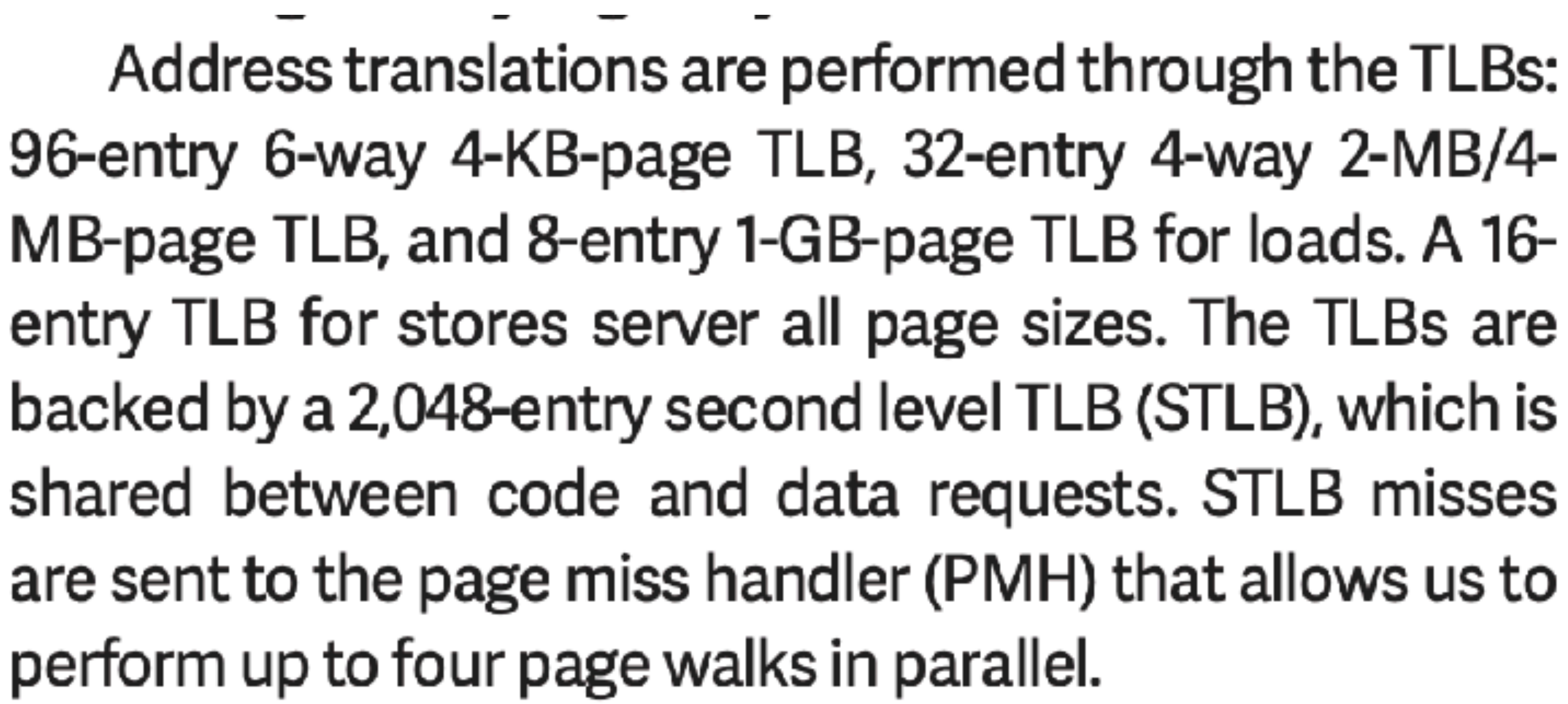
TLB.cpp

```
 TLB.cpp:2-60 (59 lines)
//START
#include<cstdint>
#include<cstdlib>
#include<vector>
#include<algorithm>
#include"function_map.hpp"
#include <sys/mman.h>

template<size_t BYTES>
struct MM {
    struct MM* next;  // I know that pointers are 8 bytes on this machine.
    uint64_t junk[BYTES/8 - 1]; // This forces the struct MM to take a up a whole cache line,
abolishing spatial locality.
};


template<class MM>
MM *  __attribute__ ((noinline))  miss(MM * start, uint64_t count)  {
    for(uint64_t i = 0; i < count; i++) { // Here's the loop that does this misses. It's very simp
        start = start->next;
    }
    return start;
}
```

TLB.cpp

```
//
template<size_t BYTES>
uint64_t* TLB(uint64_t * data, uint64_t size, uint64_t arg1) {
    struct MM<BYTES> * array = NULL;
    int r =  posix_memalign(reinterpret_cast<void**>(&array), 4096, size);
    if (r == -1) {
        std::cerr << "posix_memalign() failed.  Exiting: " << strerror(errno) << "\n";
        exit(1);
    }

    r = madvise(reinterpret_cast<void*>(array), size, MADV_NOHUGEPAGE);
    if (r == -1) {
        std::cerr << "madvise() failed.  Exiting: " << strerror(errno) << "\n";
        exit(1);
    }

    std::cout << "array alignment is " << (reinterpret_cast<uintptr_t>(array) % 4096) << "\n";
    std::cout << "array size is " << size/BYTES << " element; " << size << "B\n";

    // This is clever part  'index' is going to determine where the pointers go.  We fill it consecutive integers.
    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < size/BYTES; i++) {
        index.push_back(i);
    }
    // Randomize the list of indexes.
    std::random_shuffle(index.begin(), index.end());

    // Convert the indexes into pointers.
    for(uint64_t i = 0; i < size/BYTES; i++) {
        array[index[i]].next = &array[index[(i + 1) % (size/BYTES)]];
    }

    MM<BYTES> * start = &array[0];
    start = miss(start, arg1); // 128 million accesses.
    return reinterpret_cast<uint64_t*>(start); // This is a garbage value, but if we don't return it, the compiler will optimize out the call to
miss.
}
//END
```

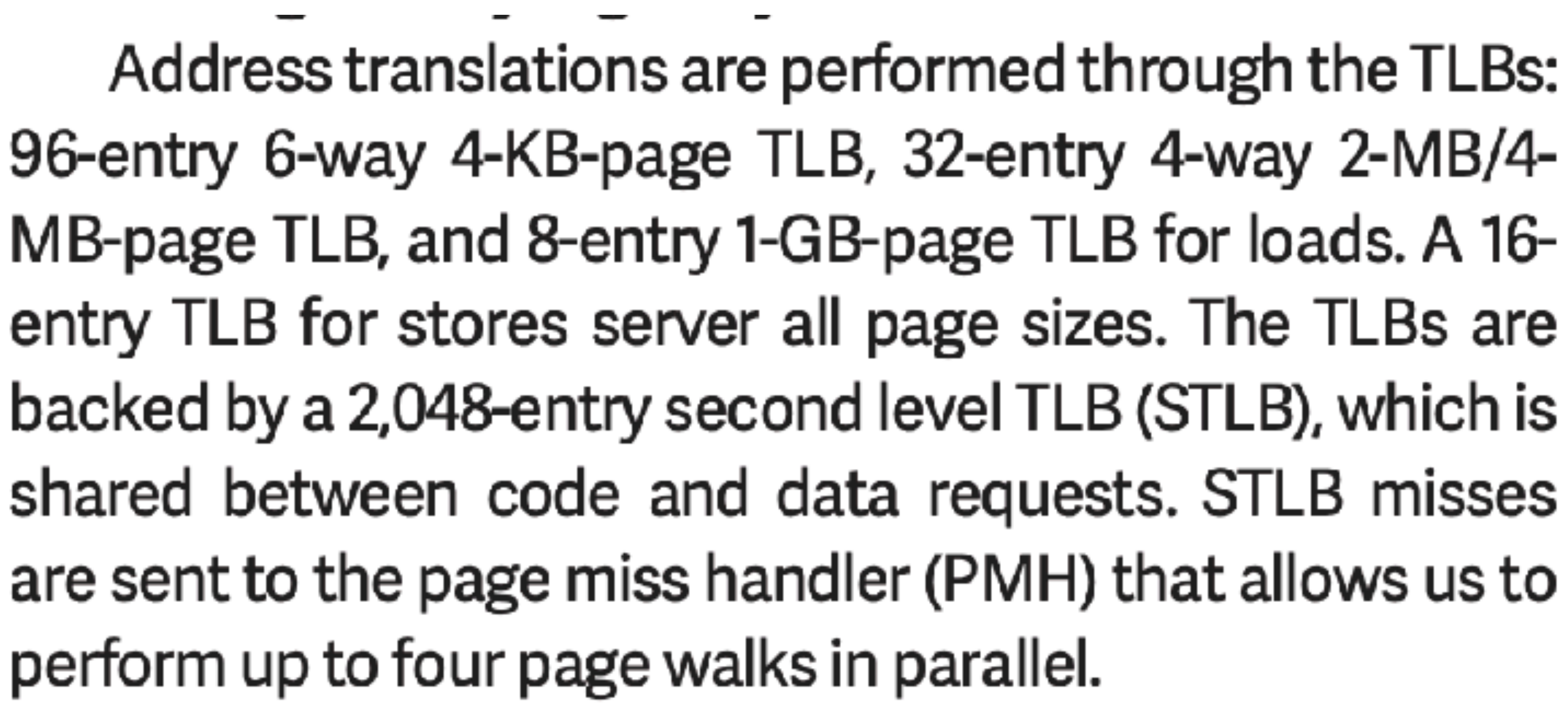
Q1: When are we seeing TLB misses?


Address translations are performed through the TLBs: 96-entry 6-way 4-KB-page TLB, 32-entry 4-way 2-MB/4-MB-page TLB, and 8-entry 1-GB-page TLB for loads. A 16-entry TLB for stores server all page sizes. The TLBs are backed by a 2,048-entry second level TLB (STLB), which is shared between code and data requests. STLB misses are sent to the page miss handler (PMH) that allows us to perform up to four page walks in parallel.

If the "working set size" is smaller than $96 \times 4KB = 384KB$, no L1 TLB misses after the first run!

If the "working set size" is smaller than $2048 \times 4KB = 8MB$, no L2 TLB misses after the first run!

Q2: How to measure L2 TLB misses?


Address translations are performed through the TLBs: 96-entry 6-way 4-KB-page TLB, 32-entry 4-way 2-MB/4-MB-page TLB, and 8-entry 1-GB-page TLB for loads. A 16-entry TLB for stores server all page sizes. The TLBs are backed by a 2,048-entry second level TLB (STLB), which is shared between code and data requests. STLB misses are sent to the page miss handler (PMH) that allows us to perform up to four page walks in parallel.

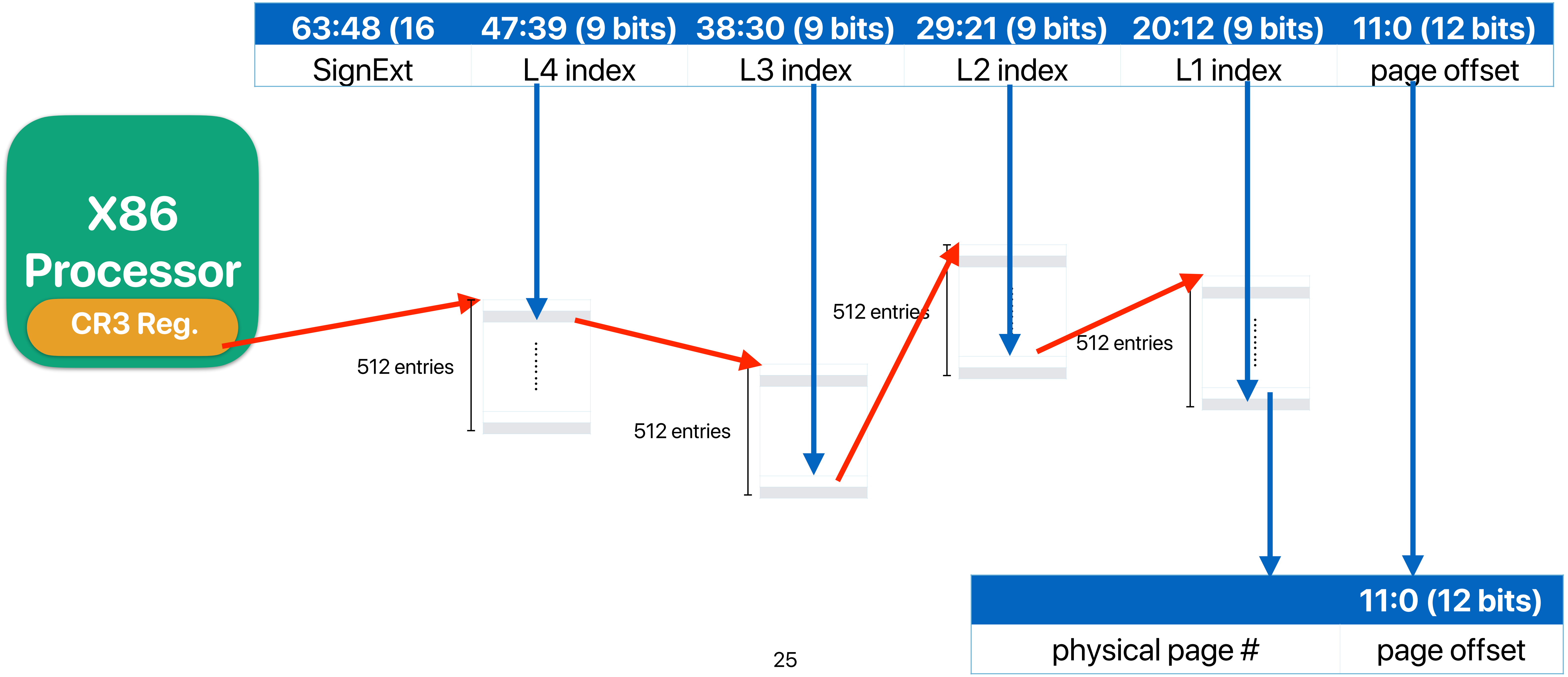
The working set size has to be larger than L2 TLB coverage

It should not go over the L3 capacity — 12MB in our processor

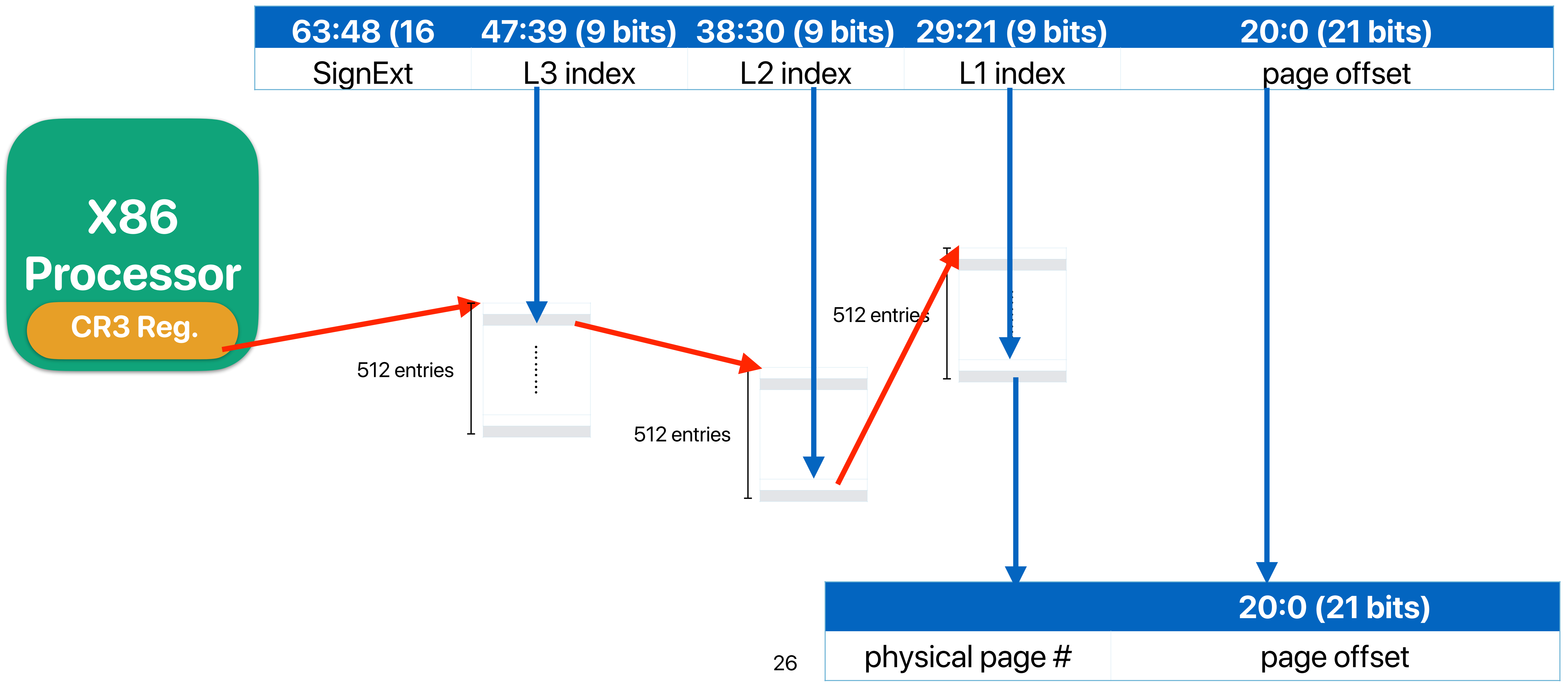
Otherwise, you would measure a lot of L3 miss latency

What happen on a TLB miss?

- Page table look up — how many memory accesses?



Huge page


63:48 (16
47:39 (9 bits)
38:30 (9 bits)
29:21 (9 bits)
20:0 (21 bits)
SignExt
L3 index
L2 index
L1 index
page offset
X86 Processor
CR3 Reg.
512 entries
512 entries
512 entries
20:0 (21 bits)
physical page #
page offset

Pros and cons of huge pages?

- Pros
  - Few page faults! — now only once every 2^{21} Bytes = 2 MB
  - Lower page table lookup overhead — now only 3 levels
- Cons
  - Internal fragmentation (e.g., the data structure does not use the complete 2 MB)
  - May not be able to find consecutive 2MB physical pages if mixed together with 4KB small pages

Q3: How to measure huge pages' overhead?

- Allocate "nodes" each "huge page" (e.g., 2MB)
- Allocate more than the target TLB entries
- Randomly traverse them
- Assume the major latency are coming from memory operations — cycles divided by memory operations.

```
df = render_csv("TLB2.csv", columns=["function","size","arg1",
"IC", "CPI", "CT", "ET","L1_dcache_miss_rate",
"L1_dcache_misses", "L1_dcache_accesses","Cycles"])
df["load_latency_cyc"] = df["Cycles"]/df["arg1"]
df["load_latency_ns"] = df["load_latency_cyc"]*df['CT']*1e9
```

Computer
Science &
Engineering

142L

つづく