Practical Cache Attacks

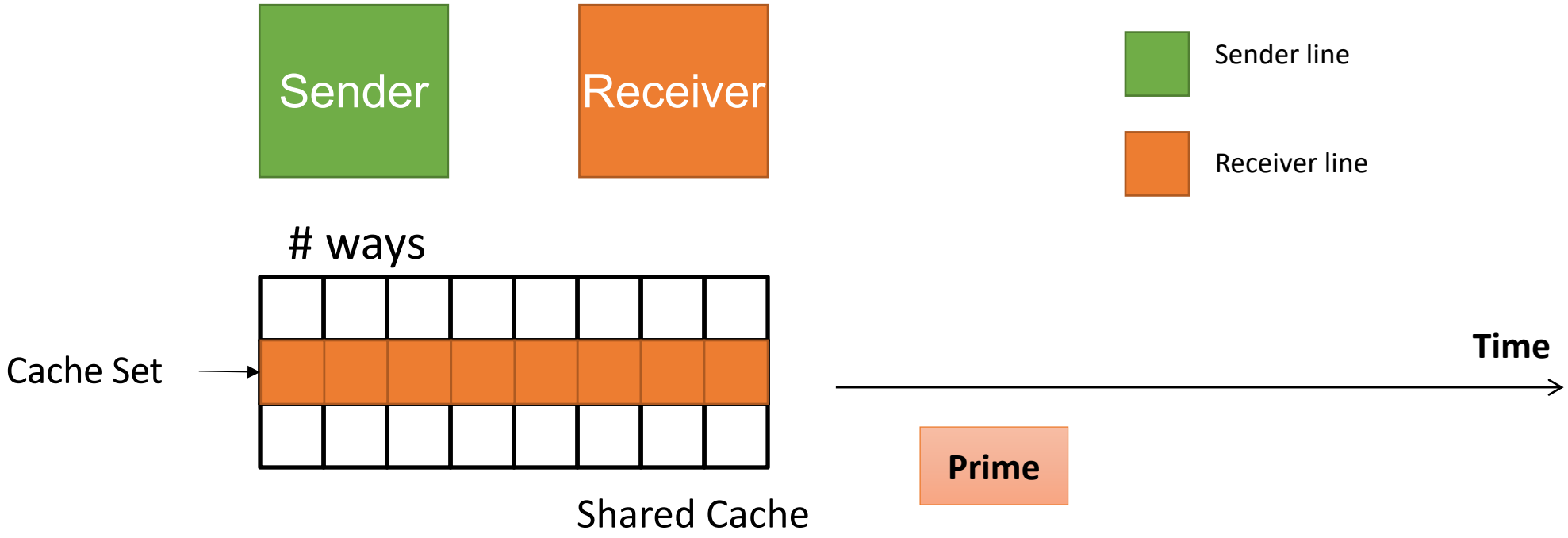
Mengjia Yan

Spring 2022

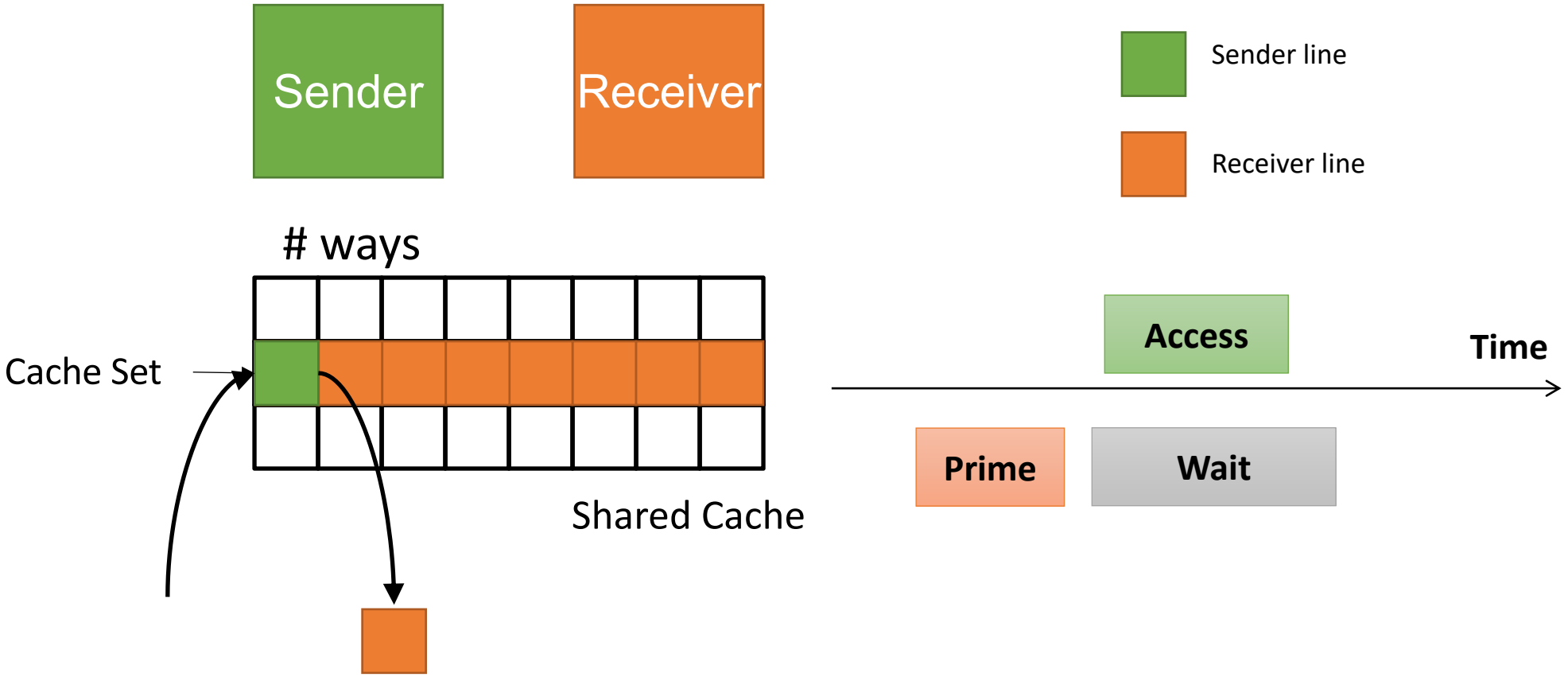
Reminder

- Lab 1 will be out TODAY
- Fill in paper preference on HotCRP by Friday

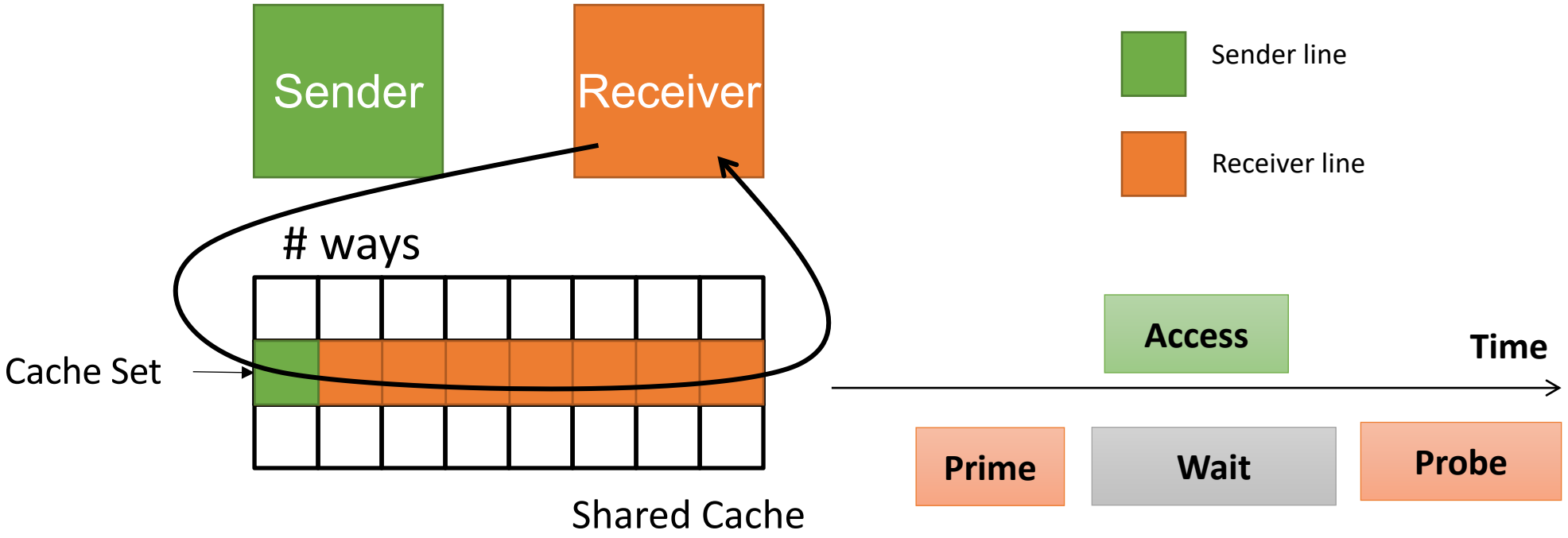
Recap: Prime+Probe



Recap: Prime+Probe

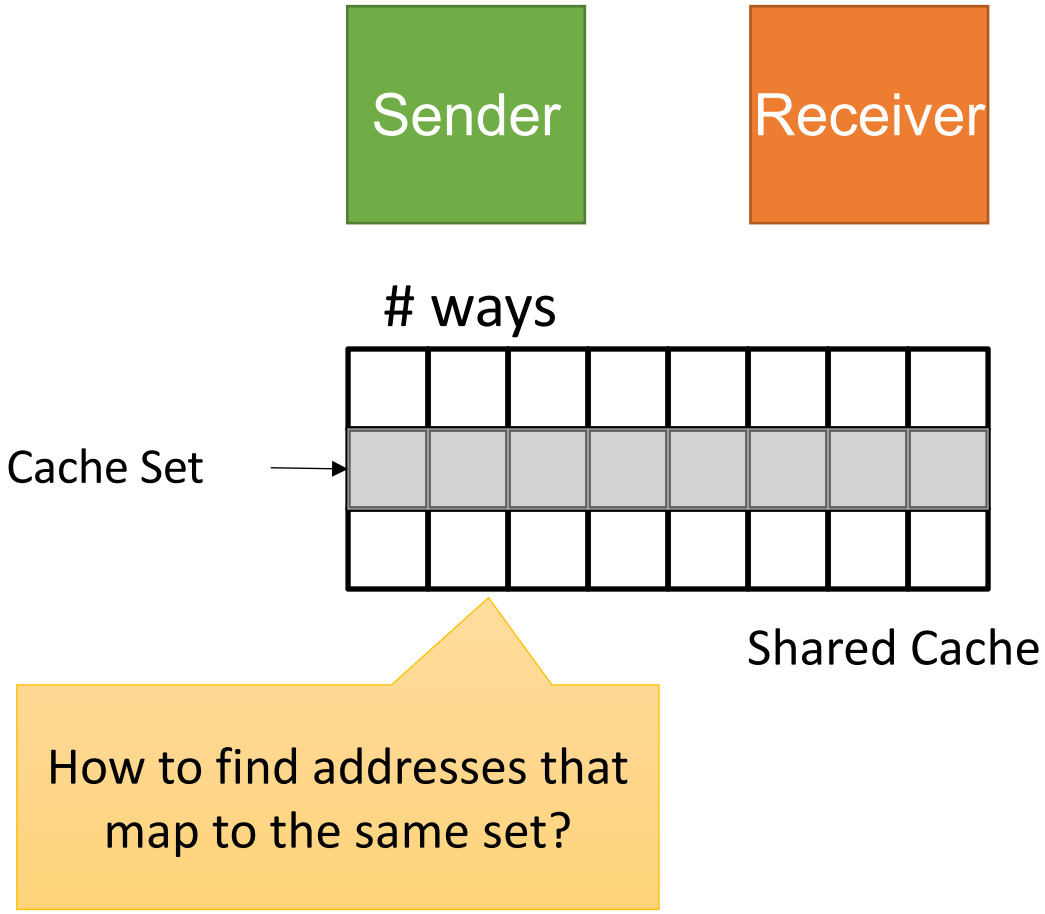


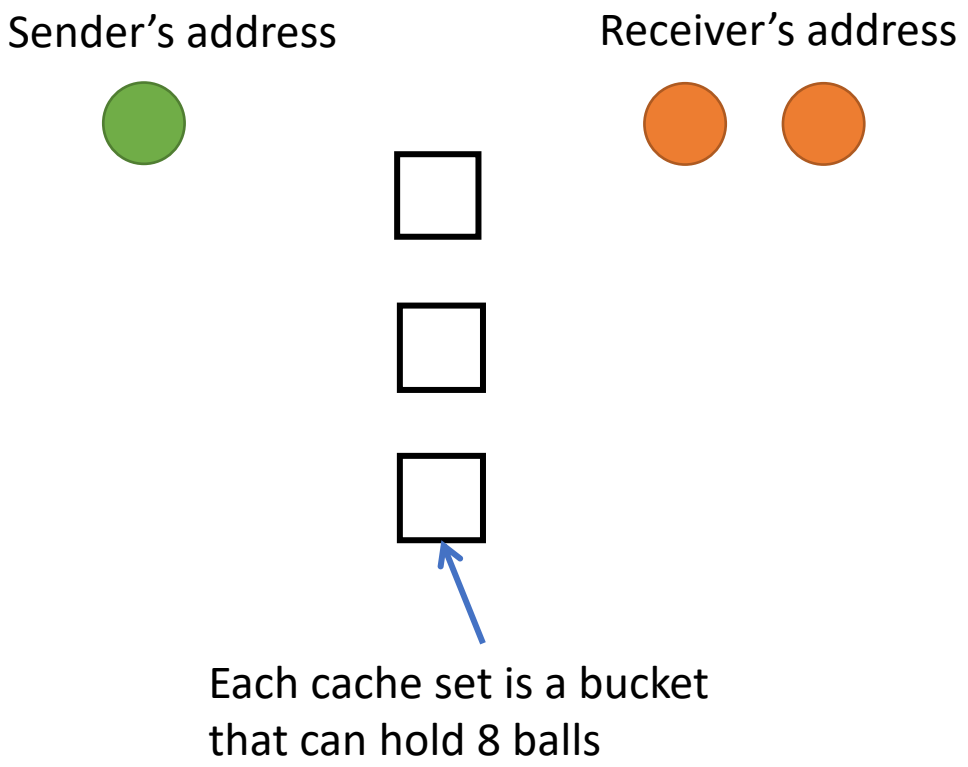
Recap: Prime+Probe



Receive “1” = 8 accesses → 1 miss

Analogy: Bucket/Ball





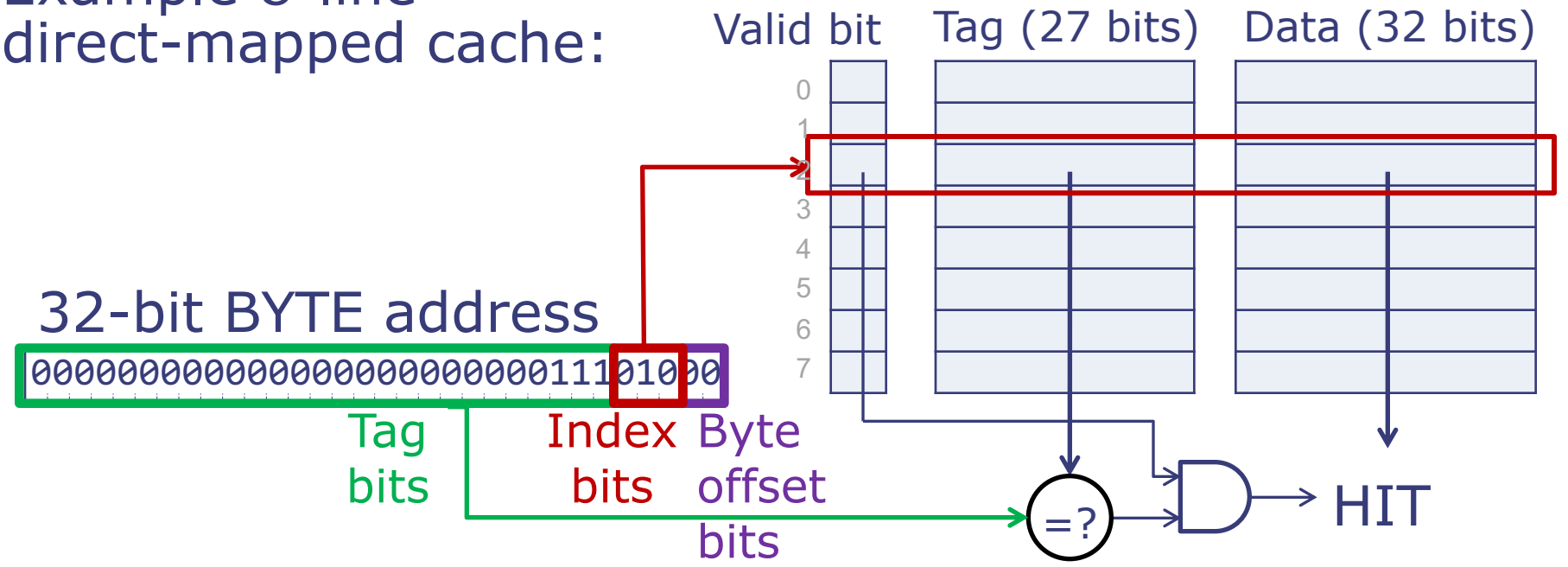
Review Cache and Address Translation

On blackboard

- Cache
  - Data + tag, directly mapped, N-way associative
- Page translation
  - Page table
  - The case when the same virtual address in different processes map to different physical addresses
  - The case when multiple virtual addresses map to the same physical address
- Hierarchical page tables and TLB
- Cache and virtual address

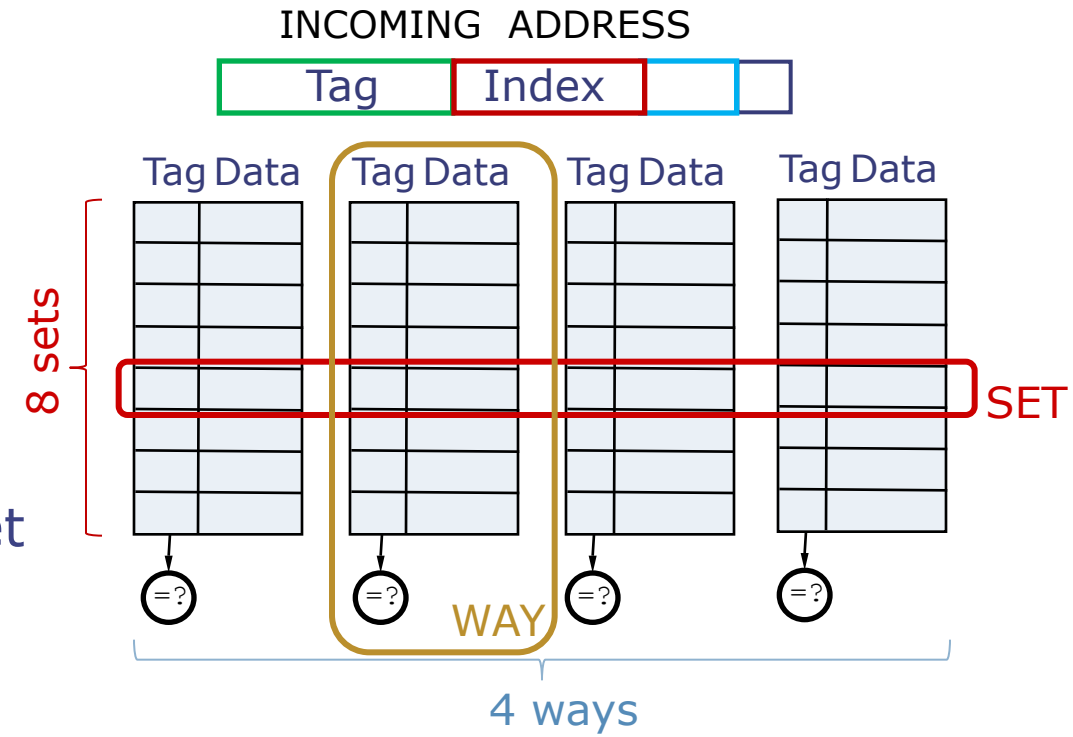
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with 2^W lines):
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



N-way Set-Associative Cache

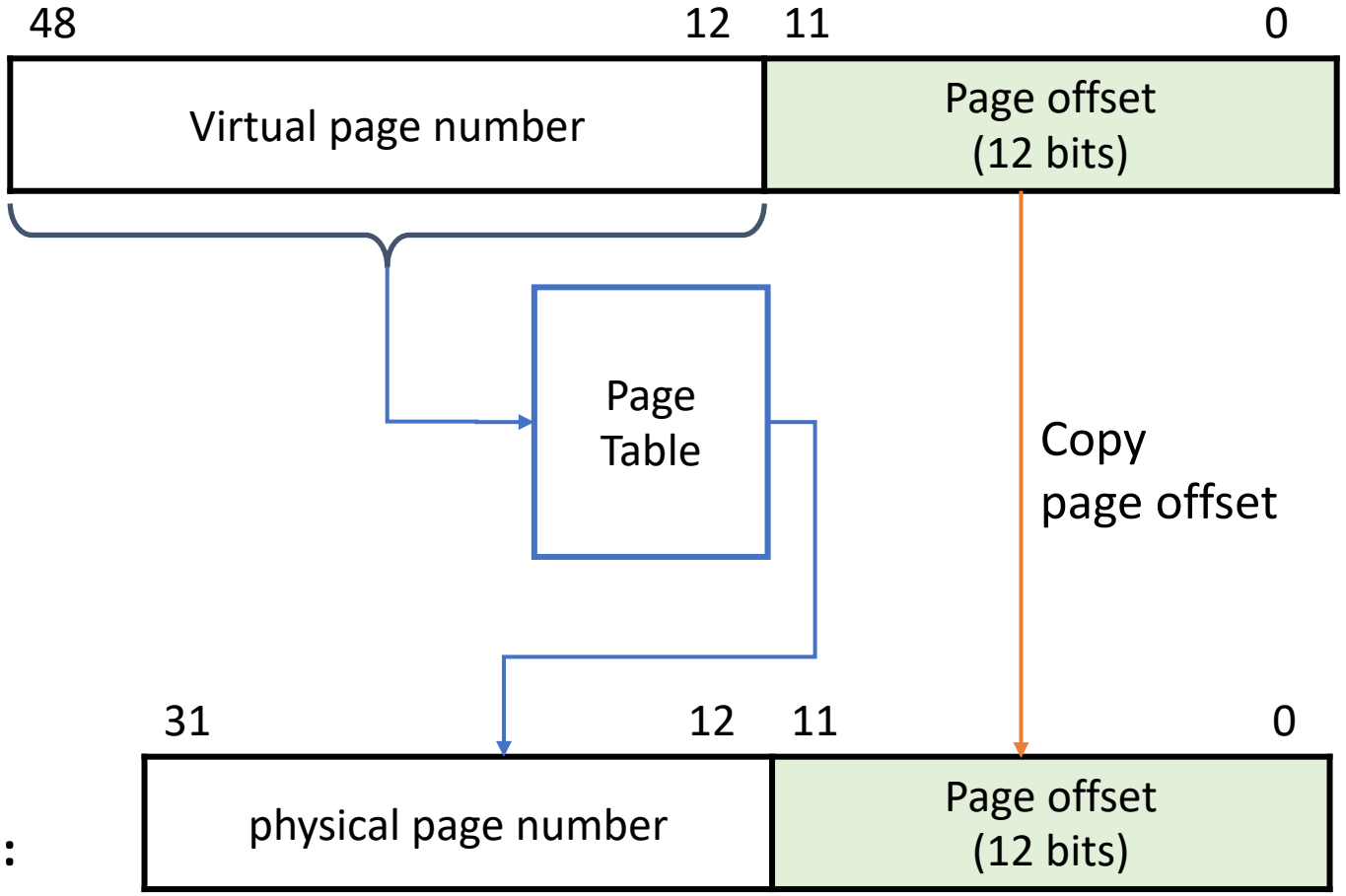
- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = #ways = "set associativity" (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
  - Tags from all ways are checked in parallel



- Fully-associative cache: Extreme case with a single set and as many ways as cache lines

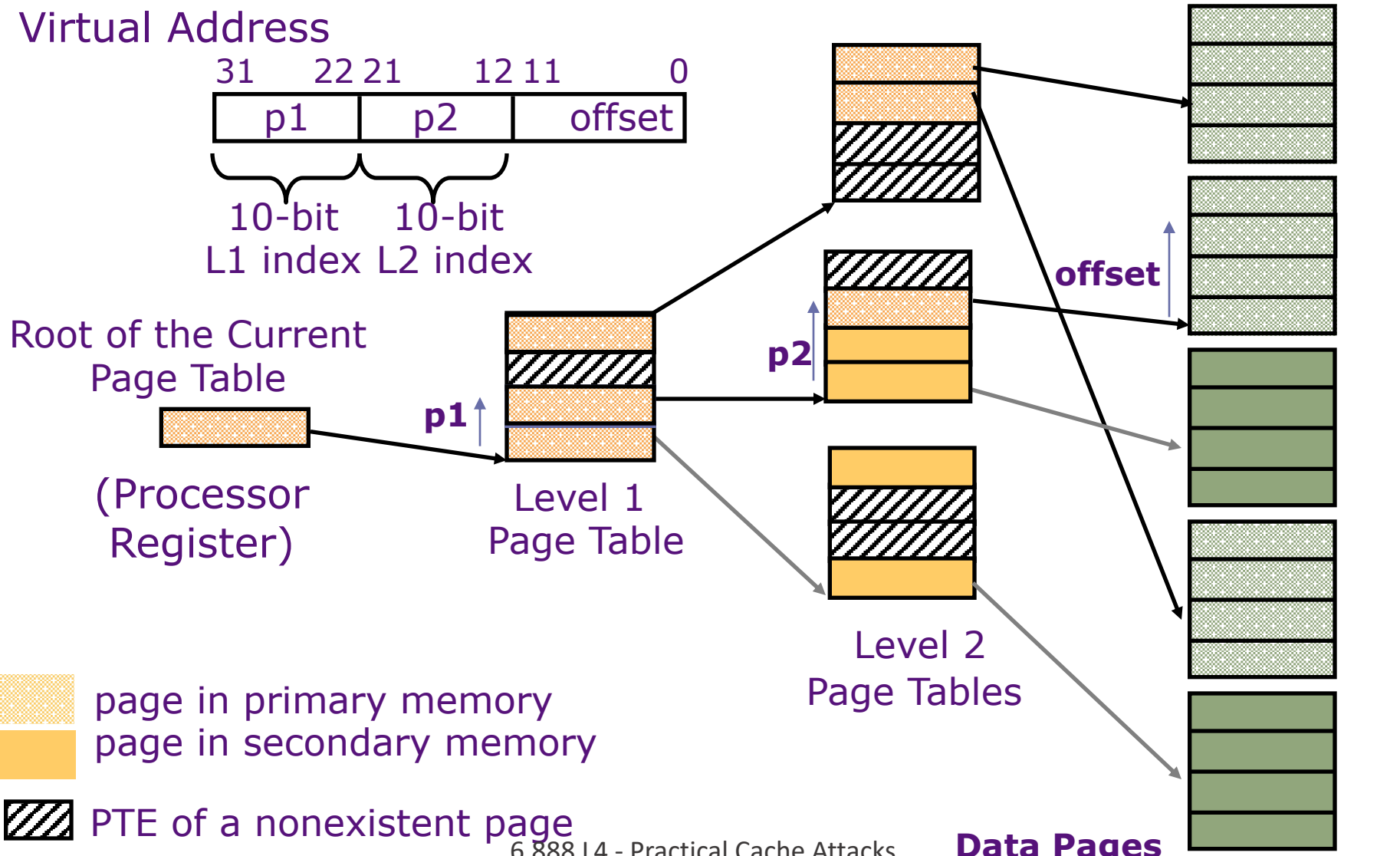
Address Translation (4KB page)

Programmer's view
Virtual Address (48bit):



system's view
Physical Address (32bit):

Hierarchical Page Table

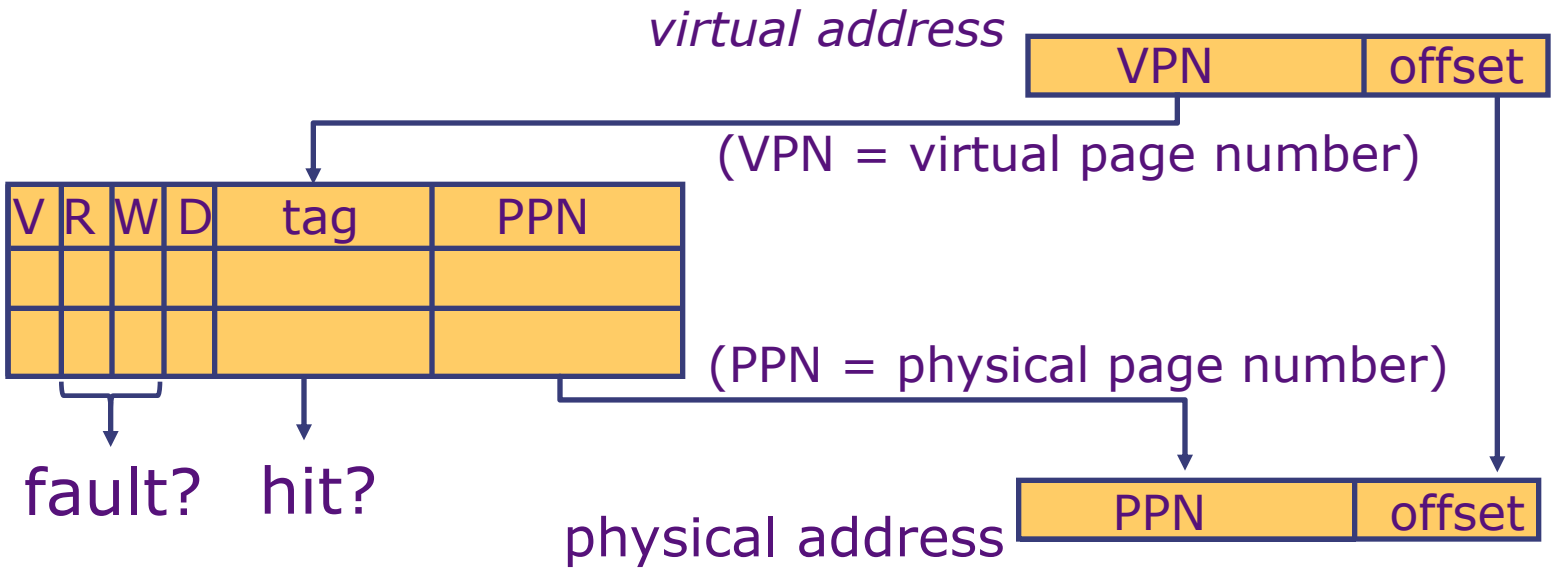


Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive!
Each reference requires accessing page table

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single-cycle translation*
TLB miss ⇒ *Access page table to refill TLB*



Using Caches with Virtual Memory

Virtually-Addressed Cache

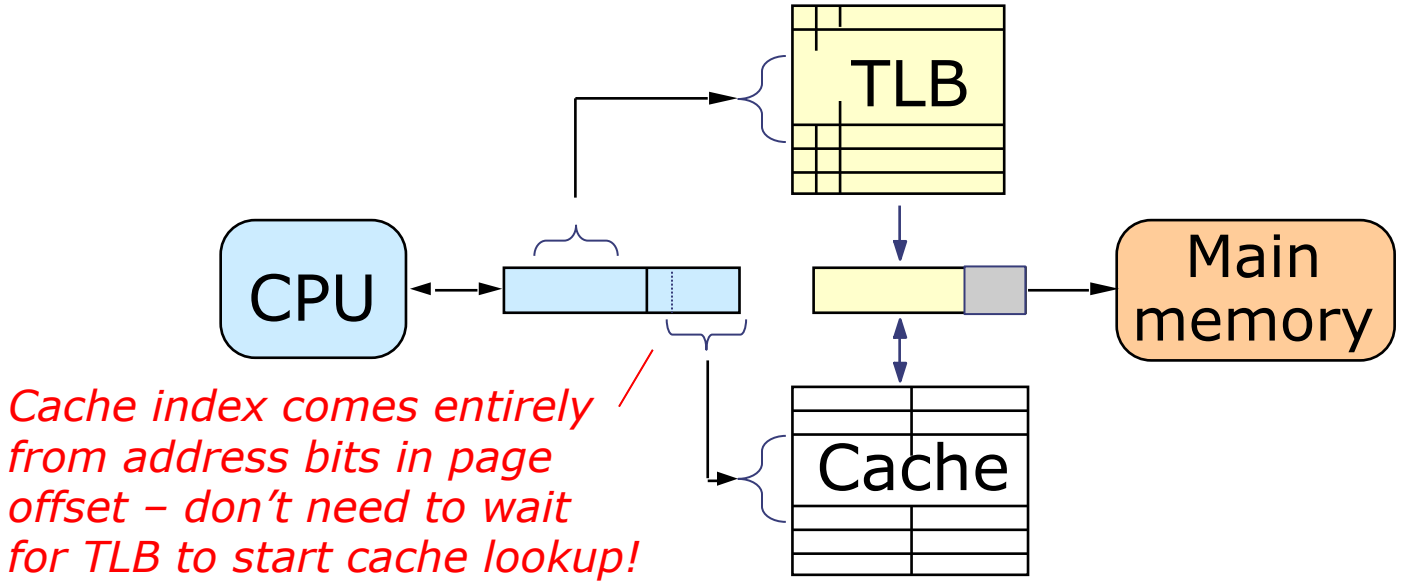
Physically-Addressed Cache

Which one is better?

Can we do even better?

Hint: some part of the virtual address and physical address are the same

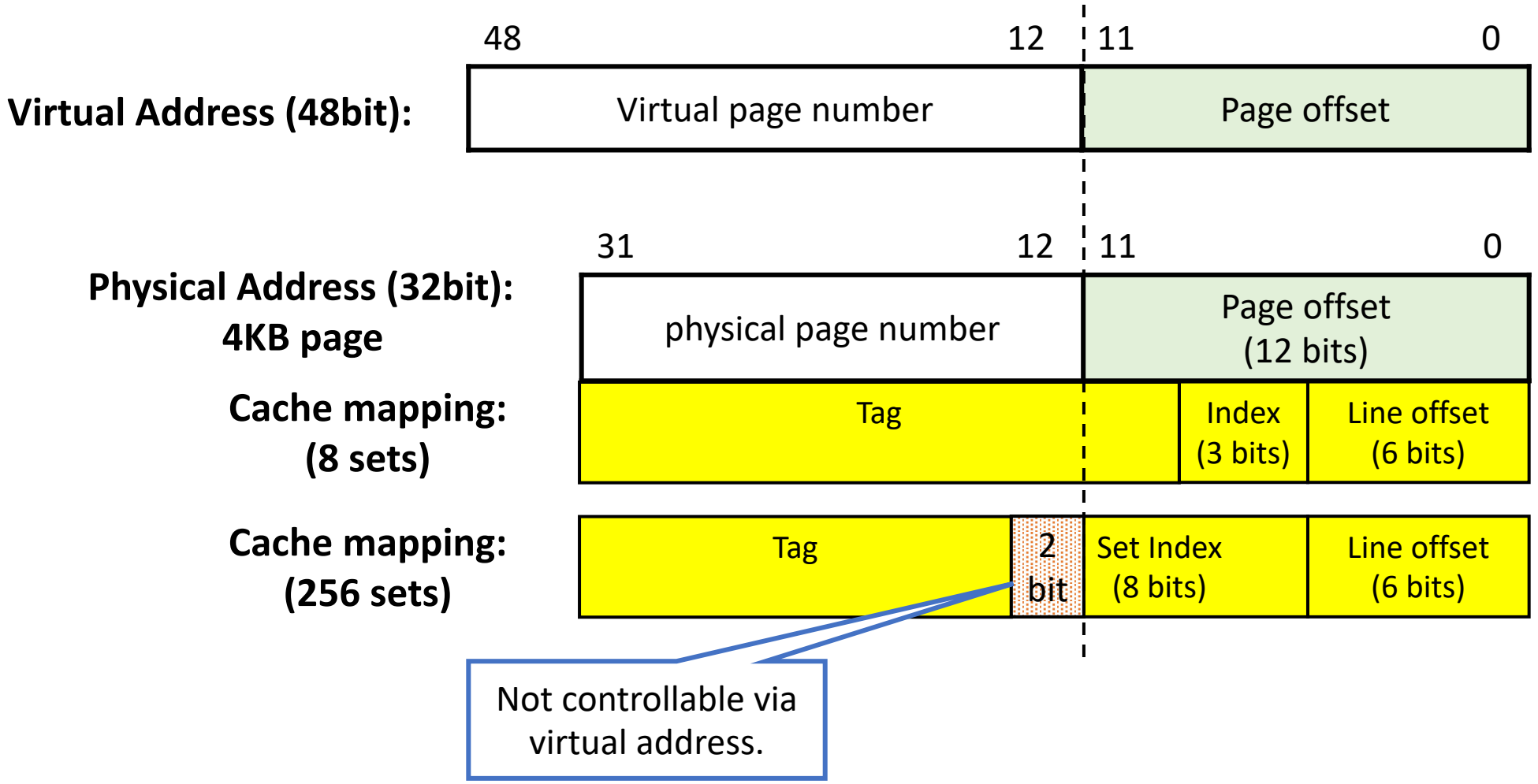
Best of Both Worlds: Virtually-Indexed, Physically-Tagged Cache (VIPT)



OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done *in parallel* with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!

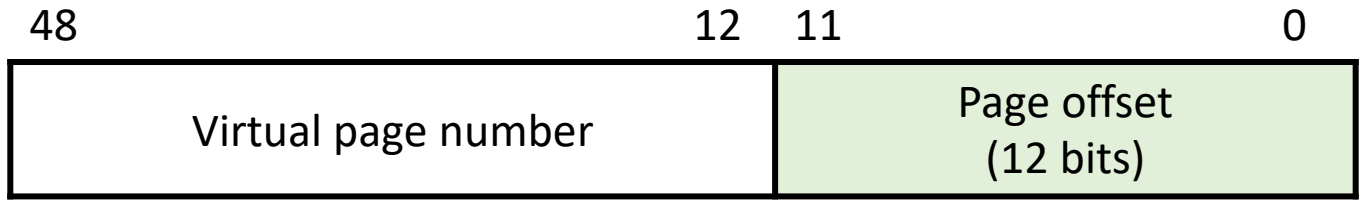
Find Conflicting Addressing From Virtual Address



w/ Huge Pages

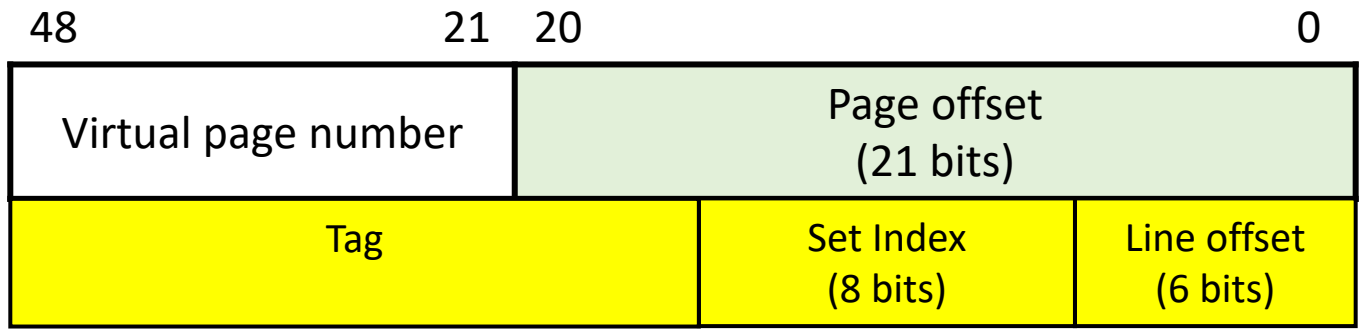
- Huge page size: 2MB or 1GB
  - Number of bits for page offset?

Virtual Address : 4KB page

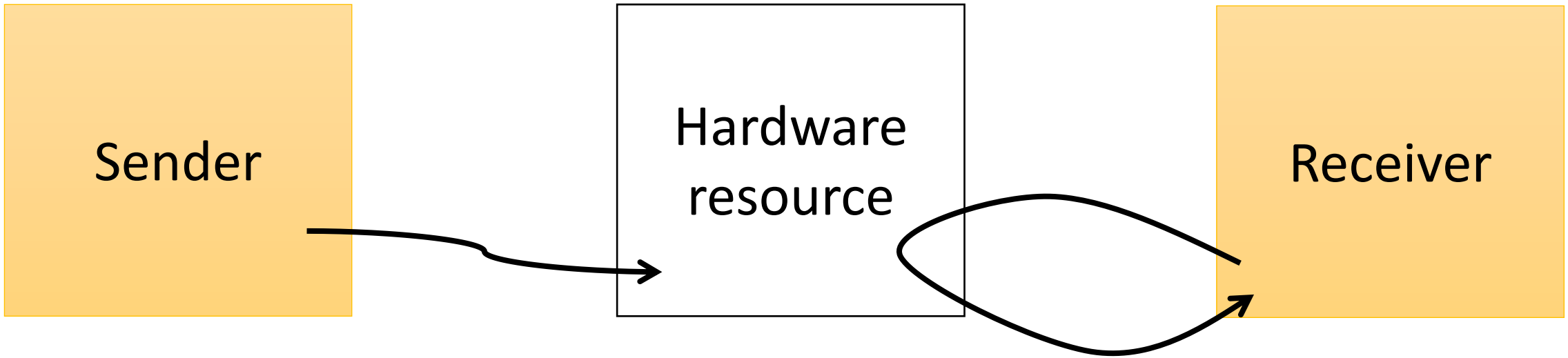


Virtual Address : 2MB page

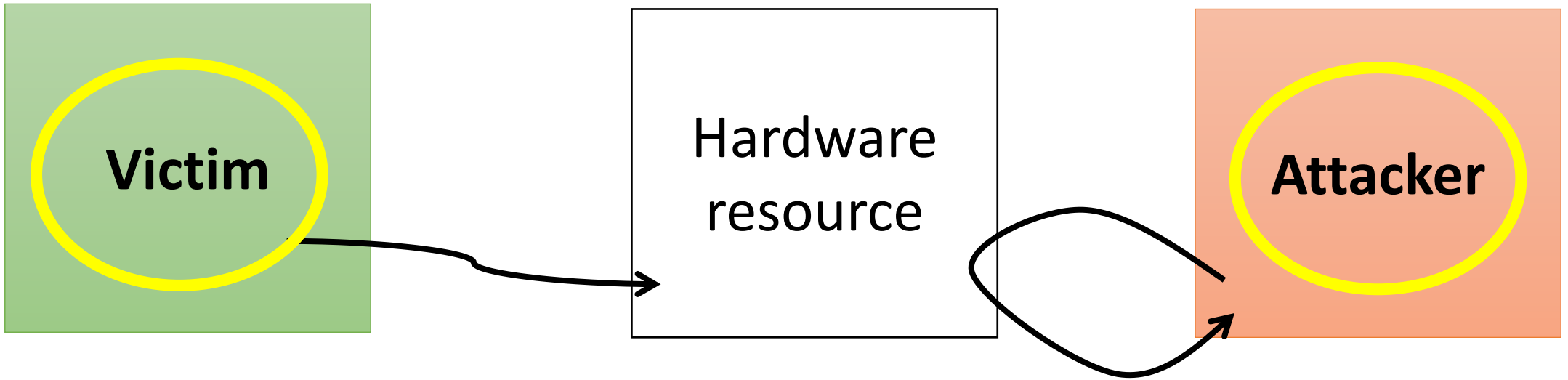
Cache mapping: (256 sets)



Micro-arch Side Channel Generalization



From Covert → Side Channels



Covert channel:

if (**send '1'**)
  Use resource
else
  idle

Side channel:

if (**secret**)
  Use resource
else
  idle

t1 = rdtsc()
Use resource
t2 = rdtsc()

if (t2 – t1 > THRESH)
  read '1'
else
  read '0'

Breaking RSA

- A real-world example: Square-and-Multiply Exponentiation

What you generally see in papers:

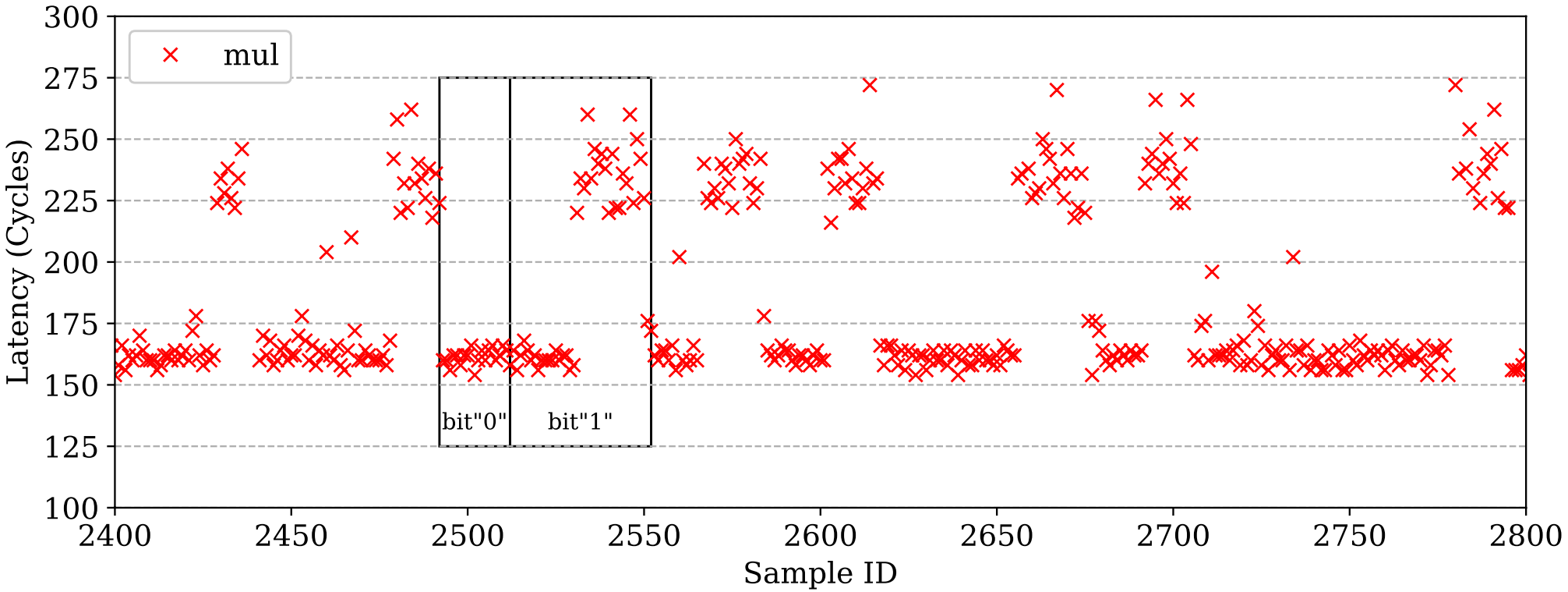
```
for i = n-1 to 0 do
        r = sqr(r)
        r  = r mod n
        if e_i == 1 then
            r = mul(r, b)
            r  = r mod n
        end
end
```

The Multiply Function

```
471 mpi_limb_t
472 mpihelp_mul( mpi_ptr_t prodp, mpi_ptr_t up, mpi_size_t usize,
473                   mpi_ptr_t vp, mpi_size_t vsize)
474 {
475     mpi_ptr_t prod_endp = prodp + usize + vsize - 1;
476     mpi_limb_t cy;
477     struct karatsuba_ctx ctx;
478
479     if( vsize < KARATSUBA_THRESHOLD ) {
480     mpi_size_t i;
481     mpi_limb_t v_limb;
482
483     if( !vsize )
484         return 0;
485
486     /* Multiply by the first limb in V separately, as the result can be
487      * stored (not added) to PROD.  We also avoid a loop for zeroing.  */
488     v_limb = vp[0];
489     if( v_limb <= 1 ) {
490         if( v_limb == 1 )
491         MPN_COPY( prodp, up, usize );
492         else
493         MPN_ZERO( prodp, usize );
494         cy = 0;
495     }
496     else
497         cy = mpihelp_mul_1( prodp, up, usize, v_limb );
498
499     prodp[usize] = cy;
500     prodp++;
```

```
501
502     /* For each iteration in the outer loop, multiply one limb from
503      * U with one limb from V, and add it to PROD.  */
504     for( i = 1; i < vsize; i++ ) {
505         v_limb = vp[i];
506         if( v_limb <= 1 ) {
507         cy = 0;
508         if( v_limb == 1 )
509             cy = mpihelp_add_n(prodp, prodp, up, usize);
510         }
511         else
512         cy = mpihelp_addmul_1(prodp, up, usize, v_limb);
513
514         prodp[usize] = cy;
515         prodp++;
516     }
517
518     return cy;
519     }
520
521     memset( &ctx, 0, sizeof ctx );
522     mpihelp_mul_karatsuba_case( prodp, up, usize, vp, vsize, &ctx );
523     mpihelp_release_karatsuba_ctx( &ctx );
524     return *prod_endp;
525 }
```

Raw Trace

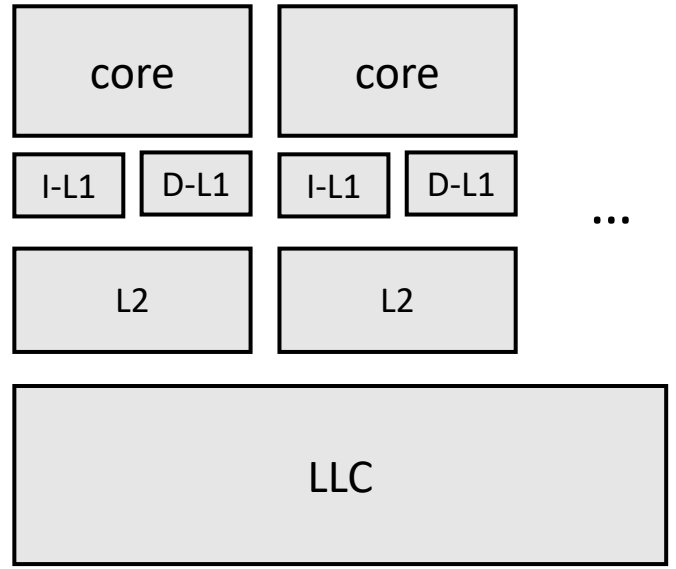


Access latencies measured in the probe operation in Prime+Probe.
A sequence of "01010111011001" can be deduced as part of the exponent.

More Advanced Cache Attacks

You may not need the following tips for Lab 1

Multi-level Caches



- Motivation:
  - A memory cannot be large and fast. Add level of cache to reduce miss penalty

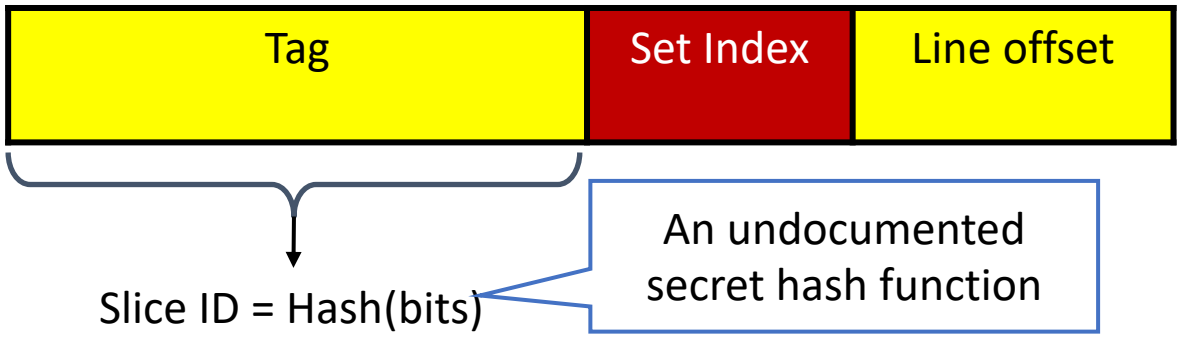
A typical configuration of Intel Ivy Bridge.
Configurations are different with processor types.

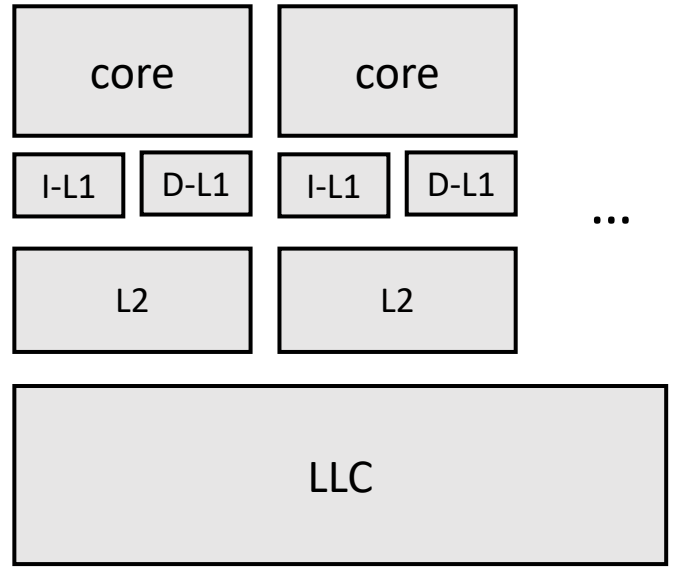
	L1-I/D cache	L2 cache	L3 cache (LLC)	DRAM
Size	32KB	256KB	1MB/core	16GB
Associativity (# ways)	4 or 8	8	16	N/A
Latency (cycles)	1-5	12	~40	~150

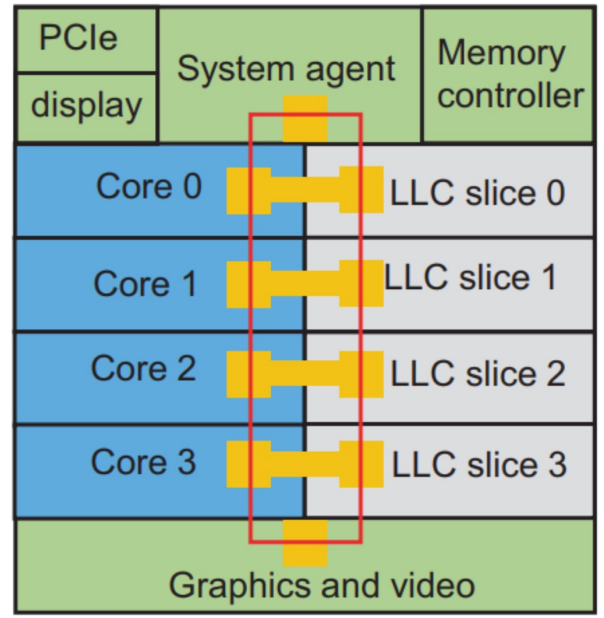
Multi-level Caches

- Motivation:
  - A memory cannot be large and fast. Add level of cache to reduce miss penalty

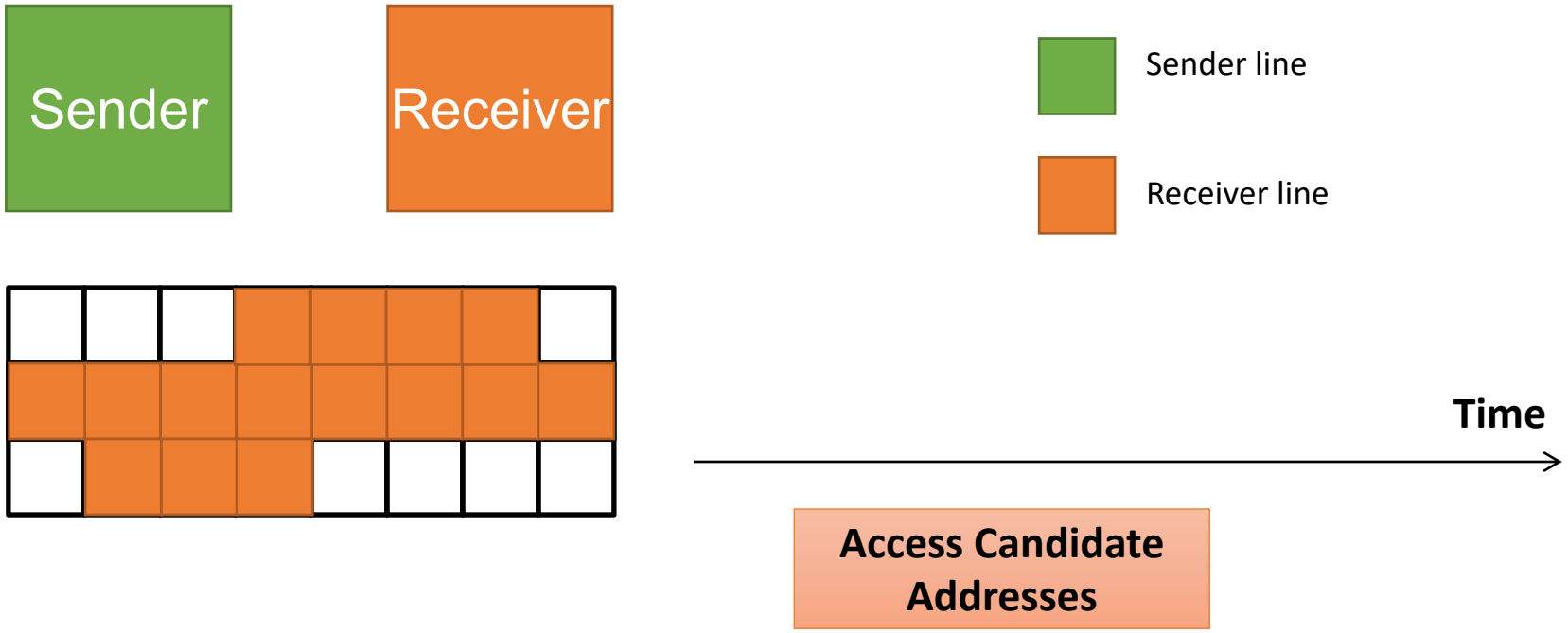
- LLC is generally divided into multiple slices
  - Conflict happens if addresses map to **the same slice and the same set**







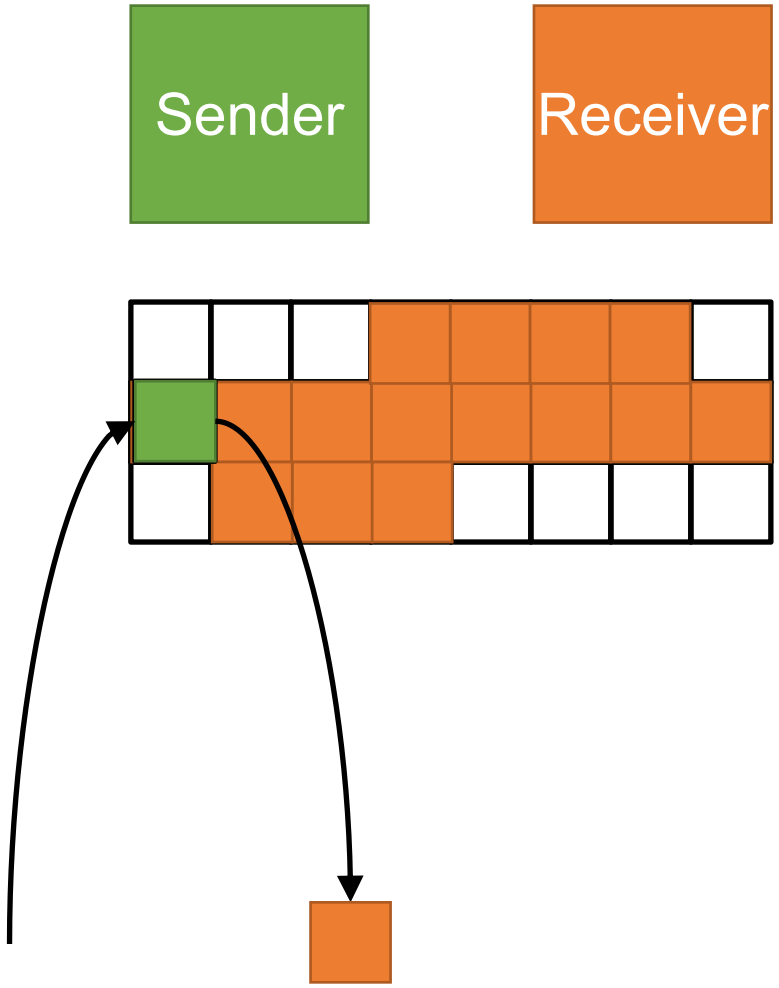
Eviction Set Construction Algorithm

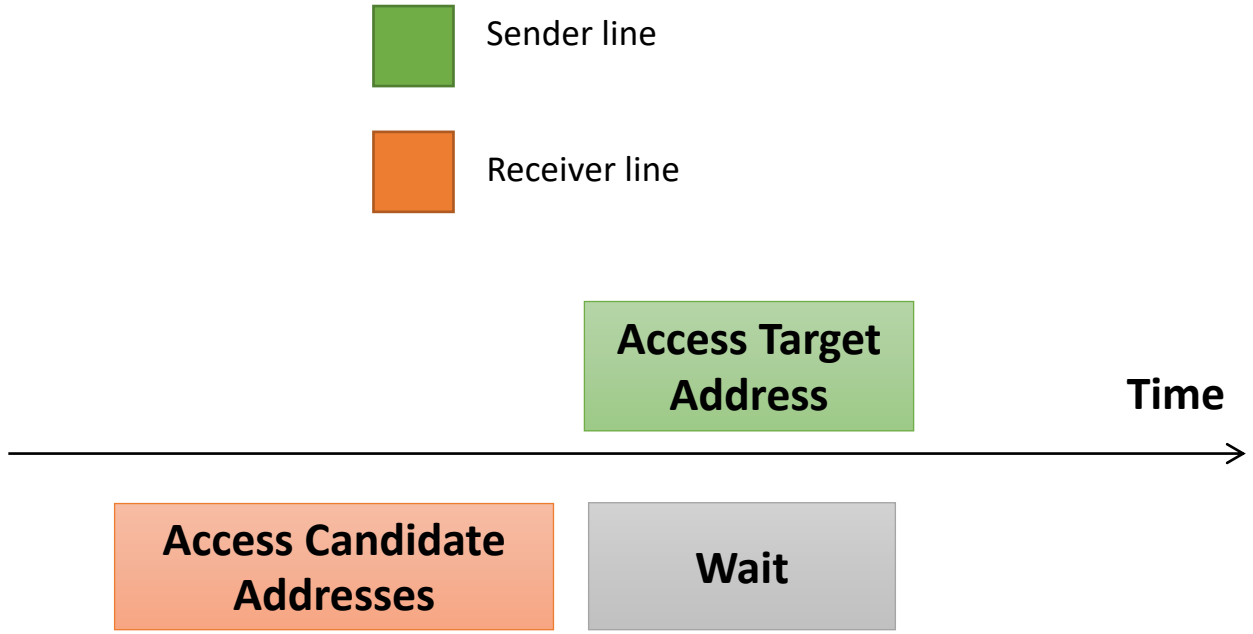


Vila et al. Theory and Practice of Finding Eviction Sets. S&P'19

Shared Cache

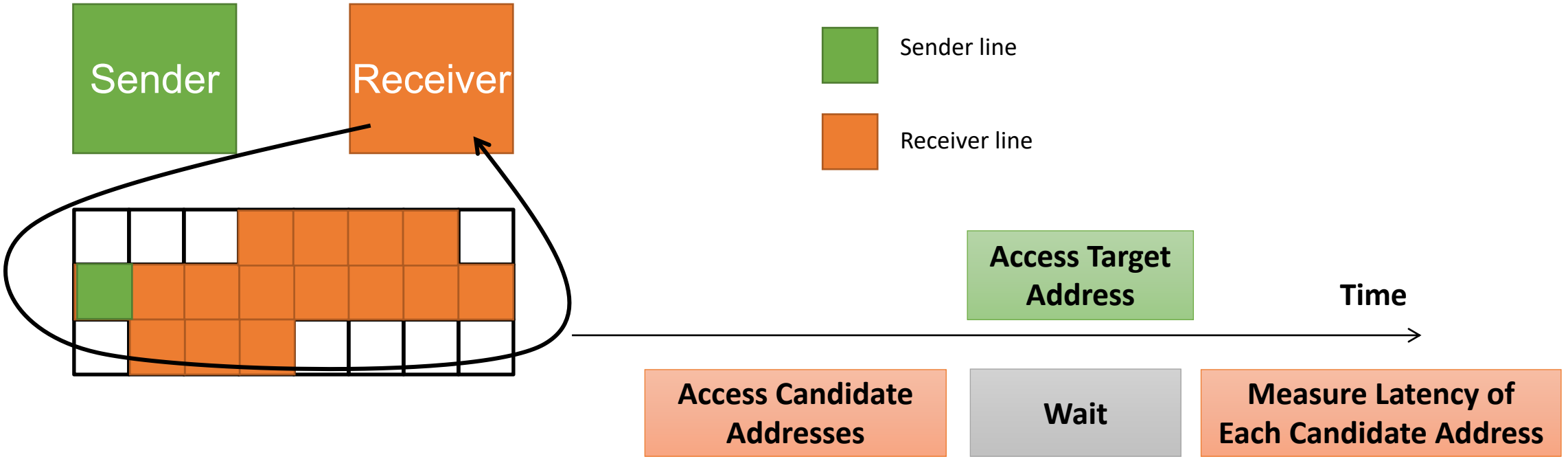
Eviction Set Construction Algorithm





Vila et al. Theory and Practice of Finding Eviction Sets. S&P'19

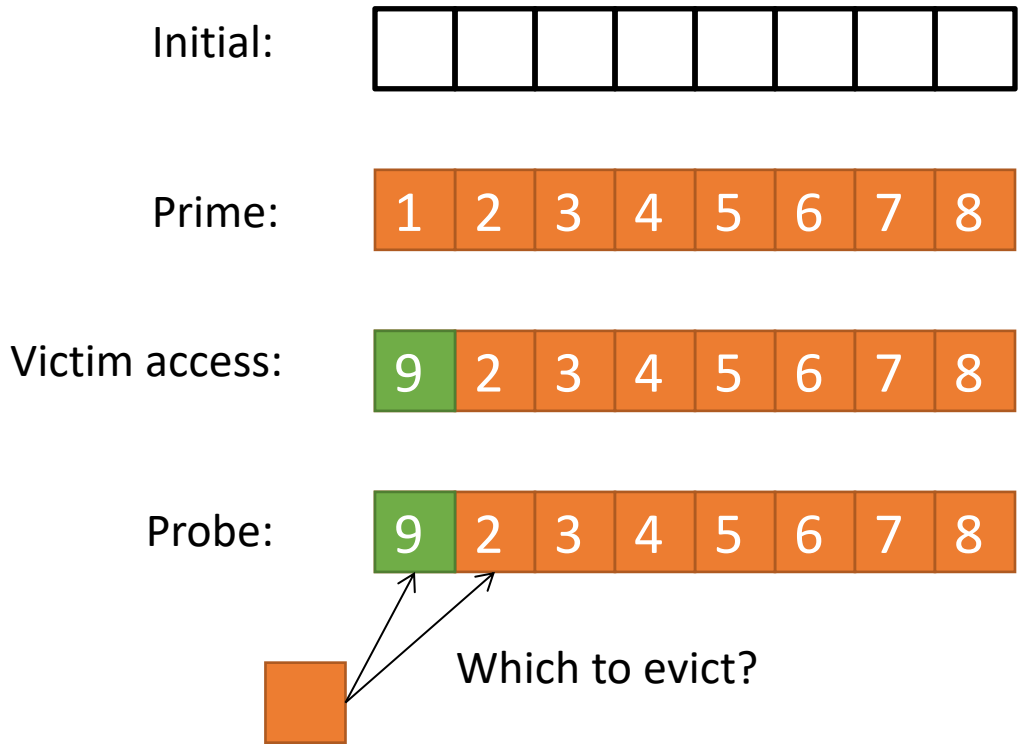
Eviction Set Construction Algorithm



Vila et al. Theory and Practice of Finding Eviction Sets. S&P'19

Problems Due to Replacement Policy

- Self-eviction due to replacement policy
  - An LRU (least recently used) example
- A small trick:
  - Access addresses in reverse order

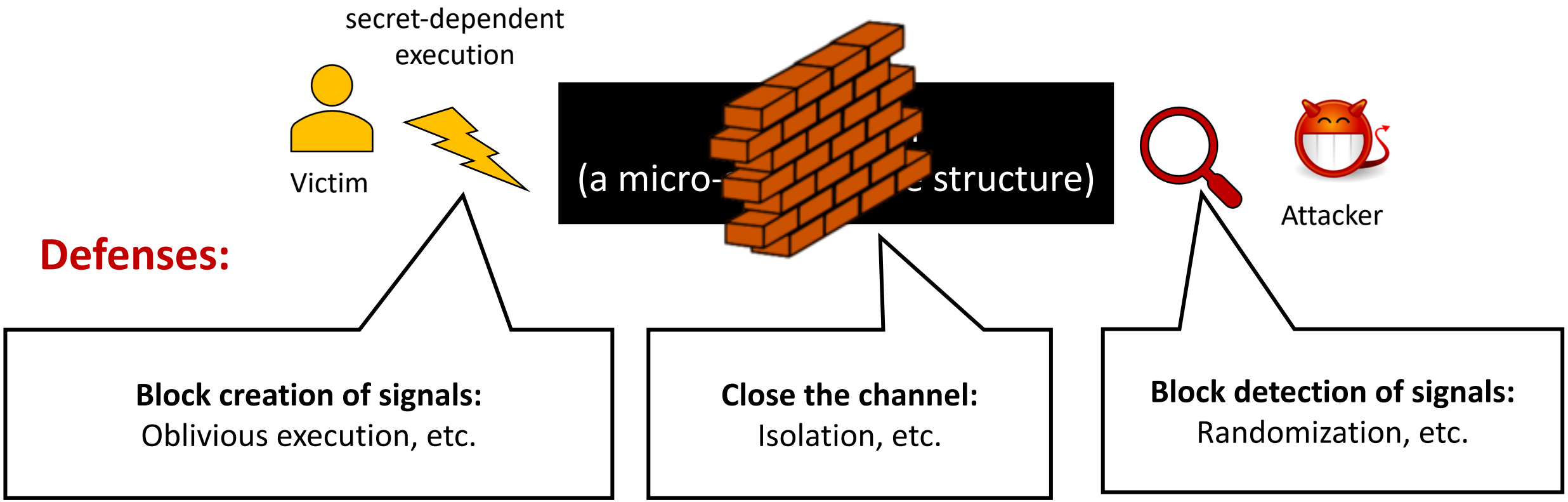


Defenses

Micro-architecture Side Channels
A Communication Model

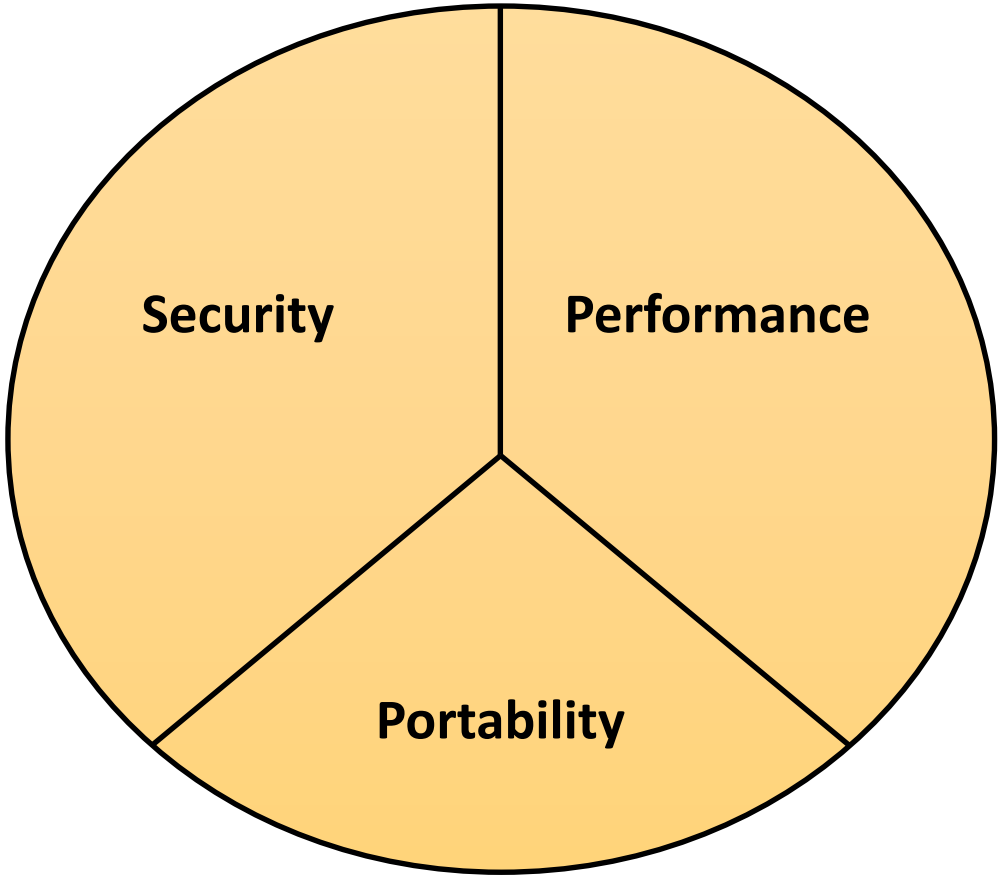
Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Micro-architecture Side Channels



Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Defense Design Considerations



Breaking RSA

- A real-world example: Square-and-Multiply Exponentiation

What you generally see in papers:

```
for i = n-1 to 0 do
        r = sqr(r)
        r  = r mod n
        if e_i == 1 then
            r = mul(r, b)
            r  = r mod n
        end
end
```

Data Oblivious/“Constant time” Programming

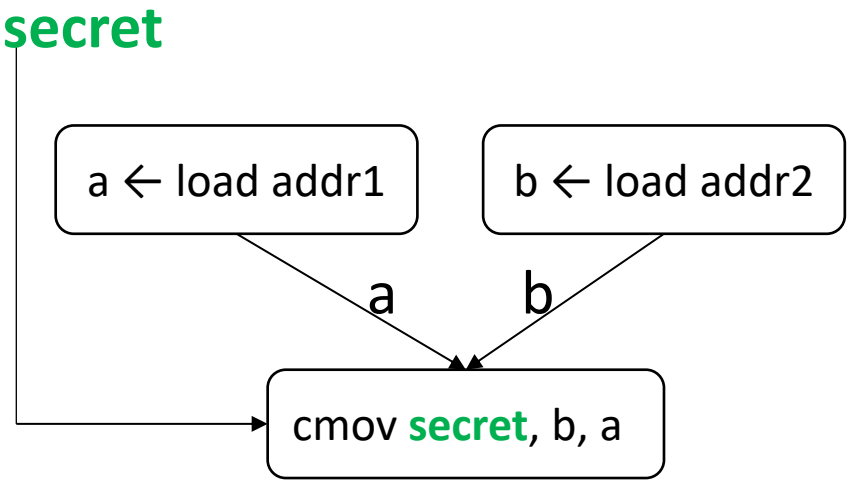
Write program w/o data-dependent behavior

Original:

```
if (secret)
    a = *(addr1);
else
    a = *(addr2);
```

Data Oblivious:

```
a ← load (addr1);
b ← load (addr2);
cmov a = (secret) ? a : b;
```



secret = confidential
addr1 = public
addr2 = public

How about nested branches?

Data Oblivious/“Constant time” Programming

Original:

```
a = buffer[secret]
```

secret = confidential
buffer = public

Data Oblivious:

```
for (int i=0; i<size; i++){
      tmp = buffer[i];
      cmov a = (i==secret) ? tmp : a;
}
```

secret = confidential
buffer = public

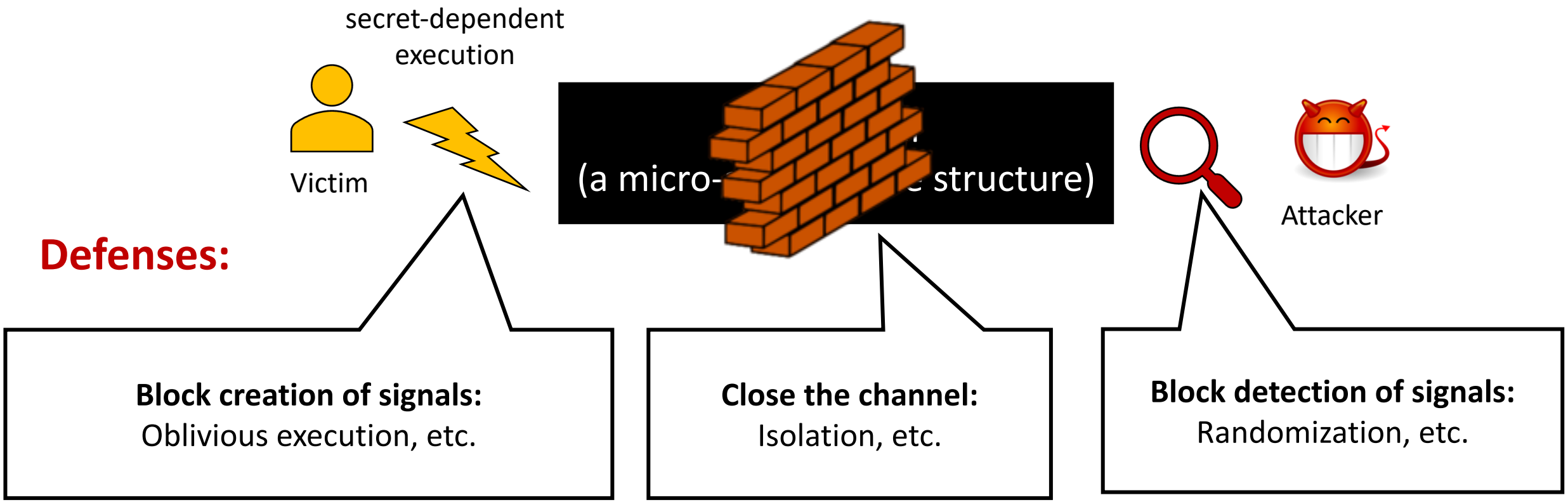
Other **data-dependent instruction optimizations**: e.g., zero-skip, early exit, microcode, silent stores, …

Zero	Normal	Subnormal	Infinity	NaN
7	11	153	7	7

Latency (in cycles) of the SQRTSS instruction for various operands.

Rane et al. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. USENIX’16

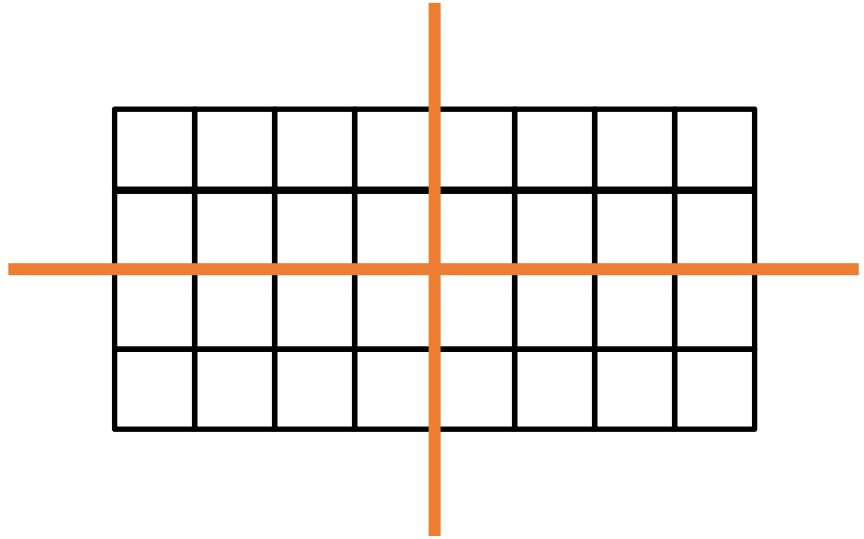
Micro-architecture Side Channels



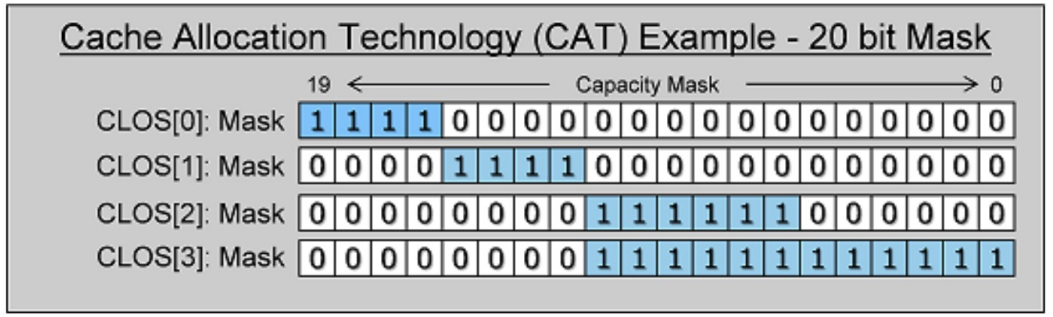
Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

HW Resource Partition

- Cache way-partition v.s. Set partition
- Temporal Partition v.s. Spatial Partition
- Security v.s. Quality of Service (QoS)
  - Intel Cache Allocation Technology (CAT)
- Challenges nowadays:
  - Security domain determination is tricky nowadays
  - Scalability: what is #domains > #partitions
  - How to partition inside cores?
  - Why not execute applications on a single node?



From https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html



Randomization/Fuzzing

- Introduce noise to time measurement/Make time measurement coarse-grained
  - Pros and cons?

+ Simple and no performance overhead
+ Effective towards a group of popular attacks
……
- Not effective to attacks that do not measure time
- Not effective to victims that cause big timing difference
- Affect usability if benign application needs to use a fine-grained timer

Shusterman et al. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. USENIX'21

Next: Paper Discussion

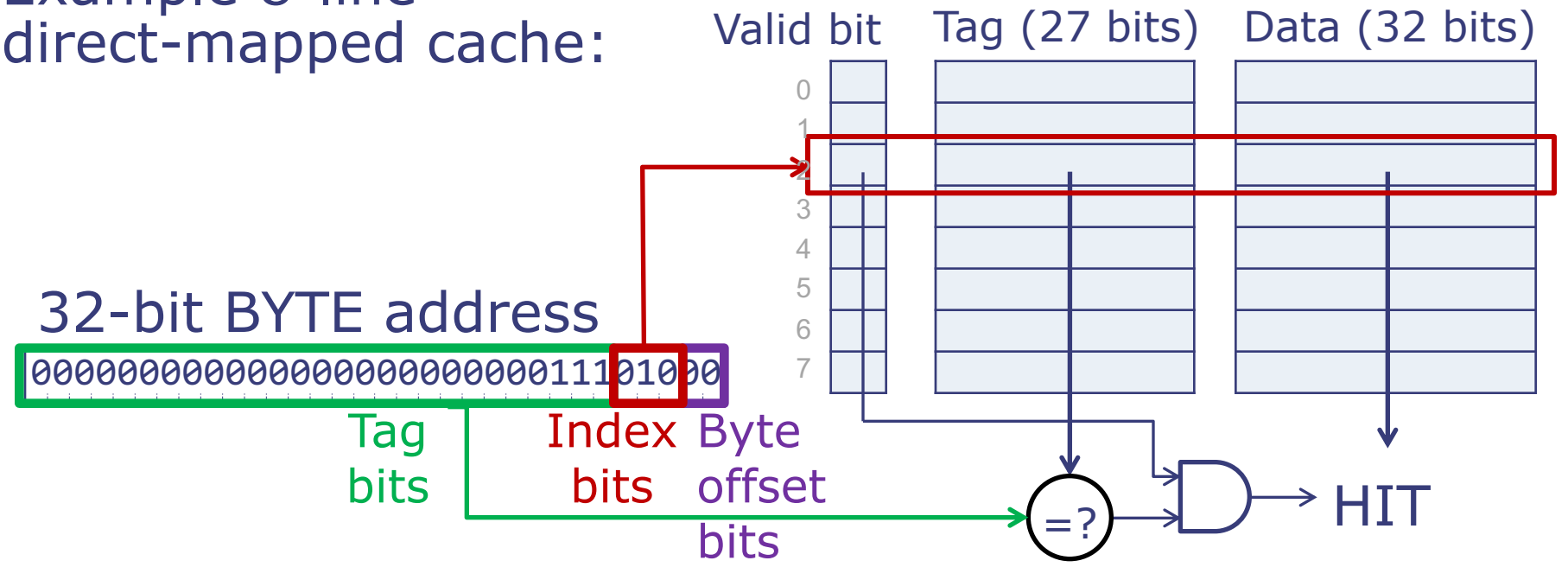
Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data

Review
Address Translation and Cache

Slides from 6.004

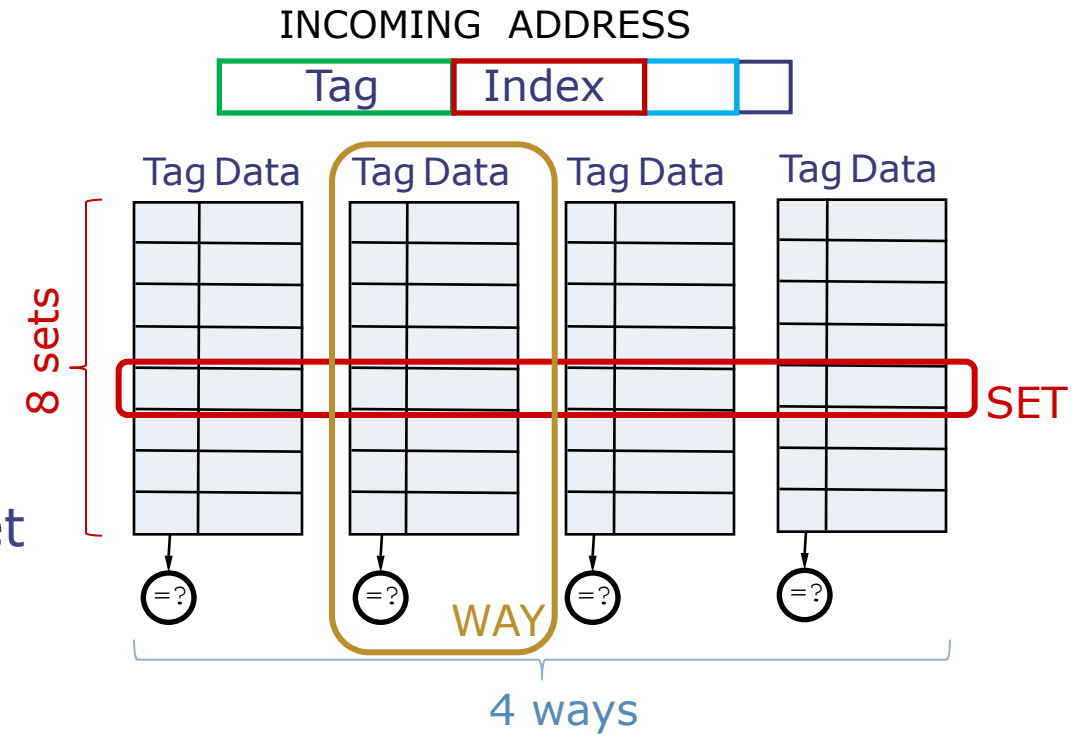
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with 2^W lines):
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



N-way Set-Associative Cache

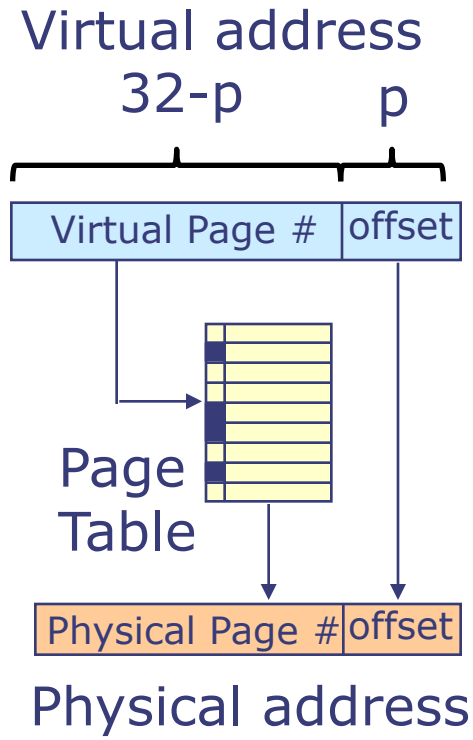
- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - Set size = #ways = "set associativity" (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
  - Tags from all ways are checked in parallel



- Fully-associative cache: Extreme case with a single set and as many ways as cache lines

Paged Memory Systems

- Divide physical memory in fixed-size blocks called pages
  - Typical page size: 4KB
- Interpret each virtual address as a pair <virtual page number, offset>
- Use a page table to translate from virtual to physical page numbers
  - Page table contains the physical page number (i.e., starting physical address) for each virtual page number



Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

⇒ 2^{20} PTEs, i.e, 4 MB page table per user

⇒ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

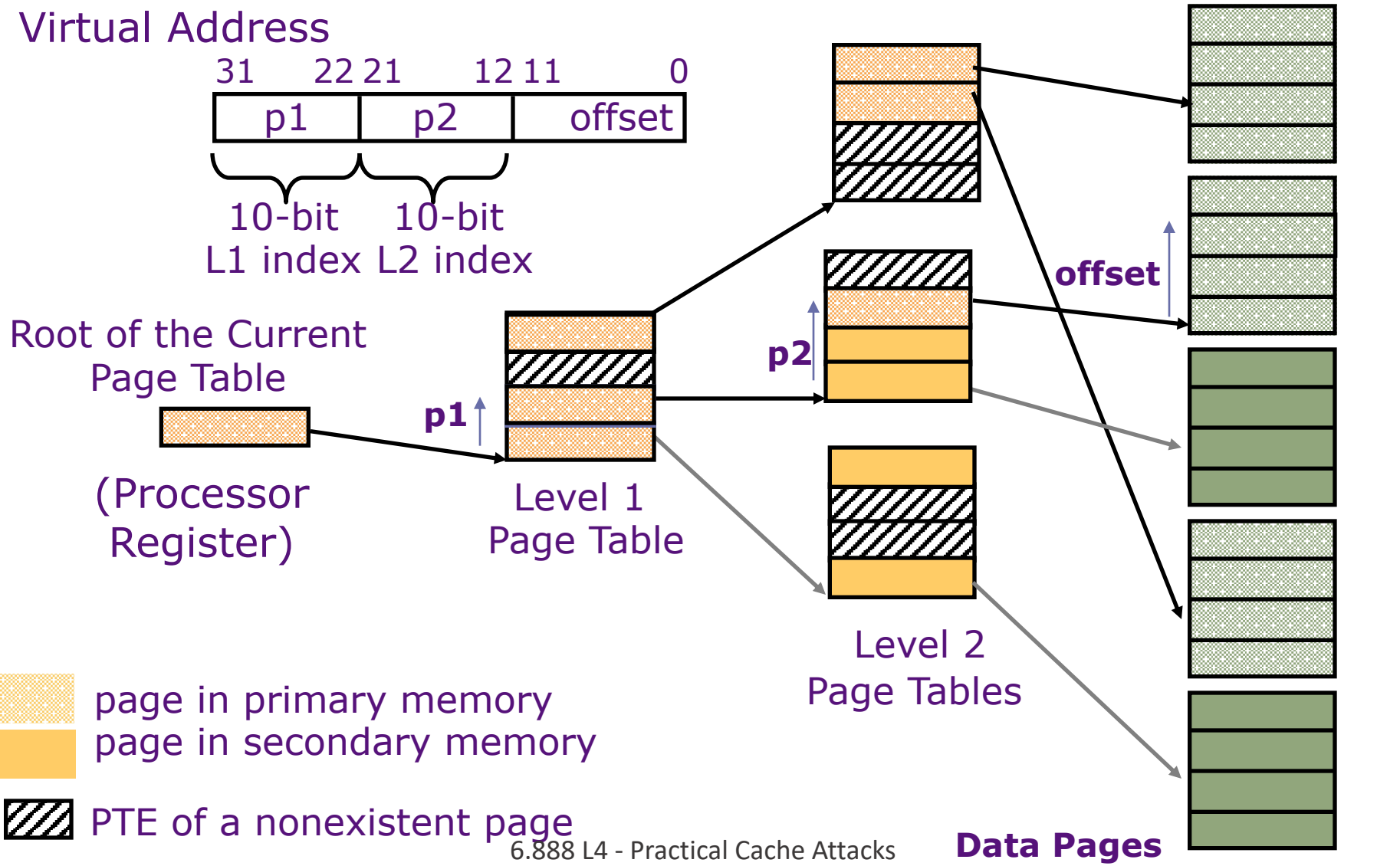
- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

What is the "saving grace"?

Hierarchical Page Table

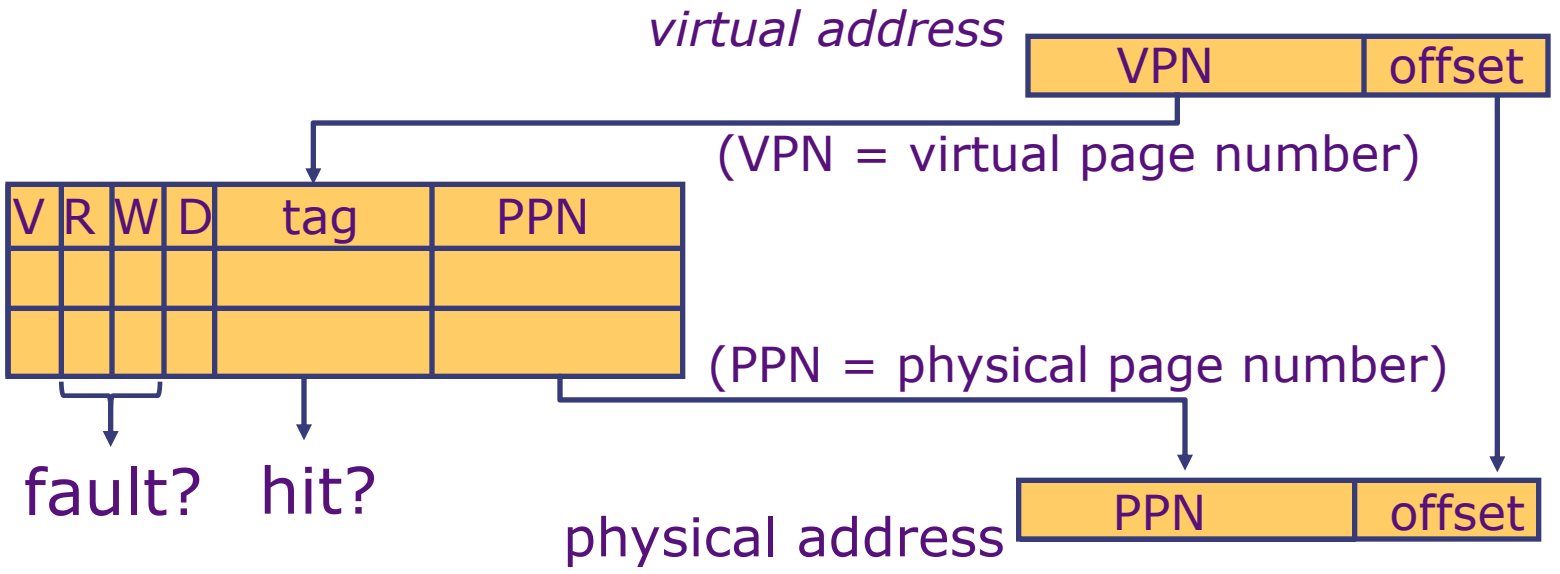


Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive!
Each reference requires accessing page table

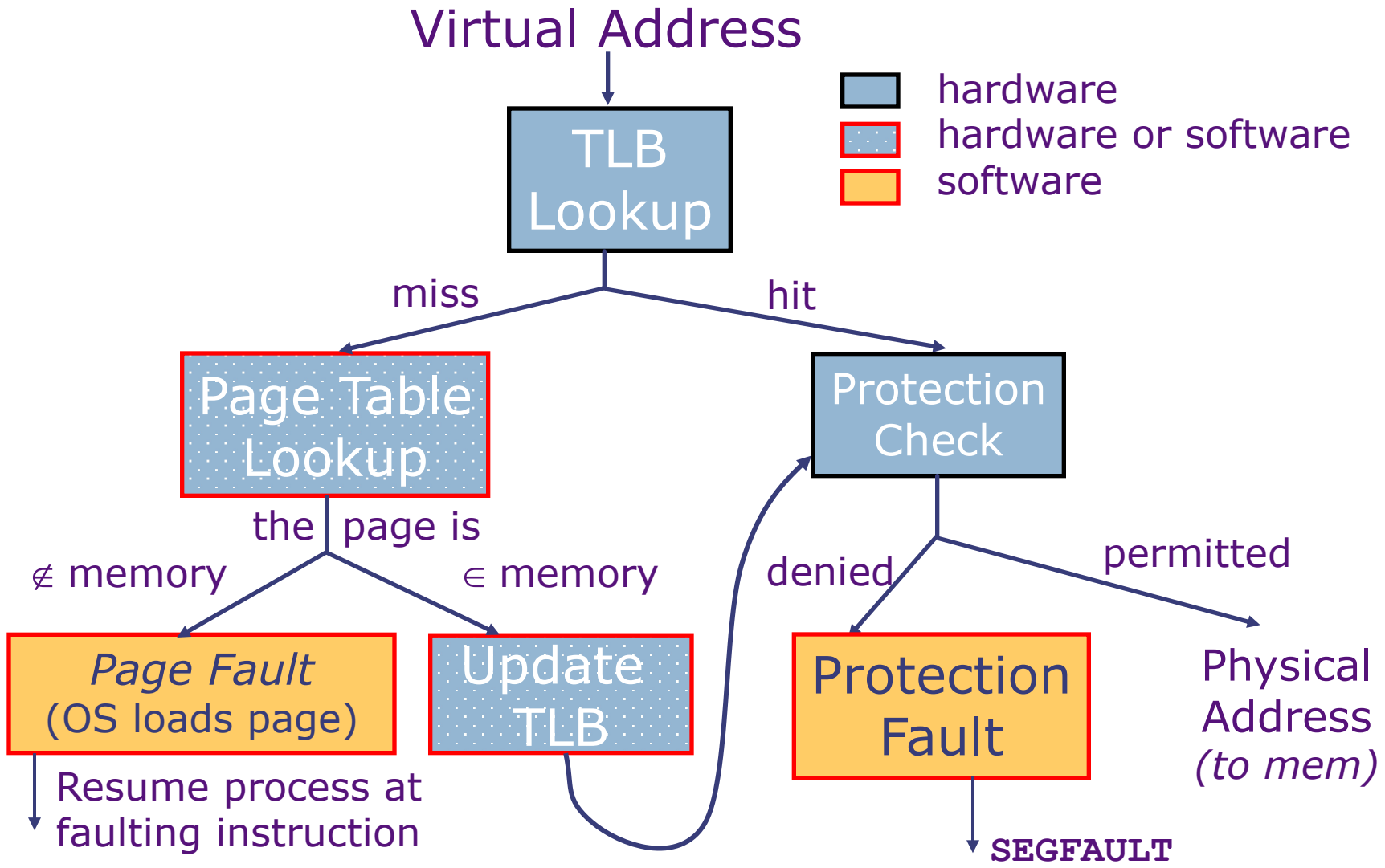
Solution: *Cache translations in TLB*

TLB hit ⇒ *Single-cycle translation*
TLB miss ⇒ *Access page table to refill TLB*



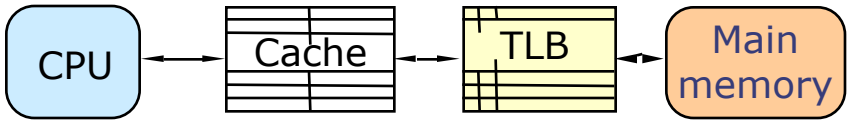
Address Translation

Putting it all together



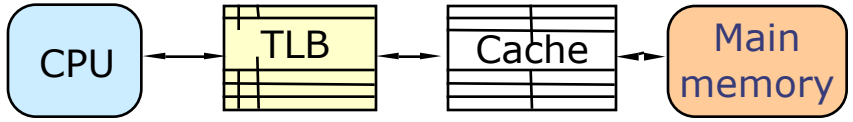
Using Caches with Virtual Memory

Virtually-Addressed Cache



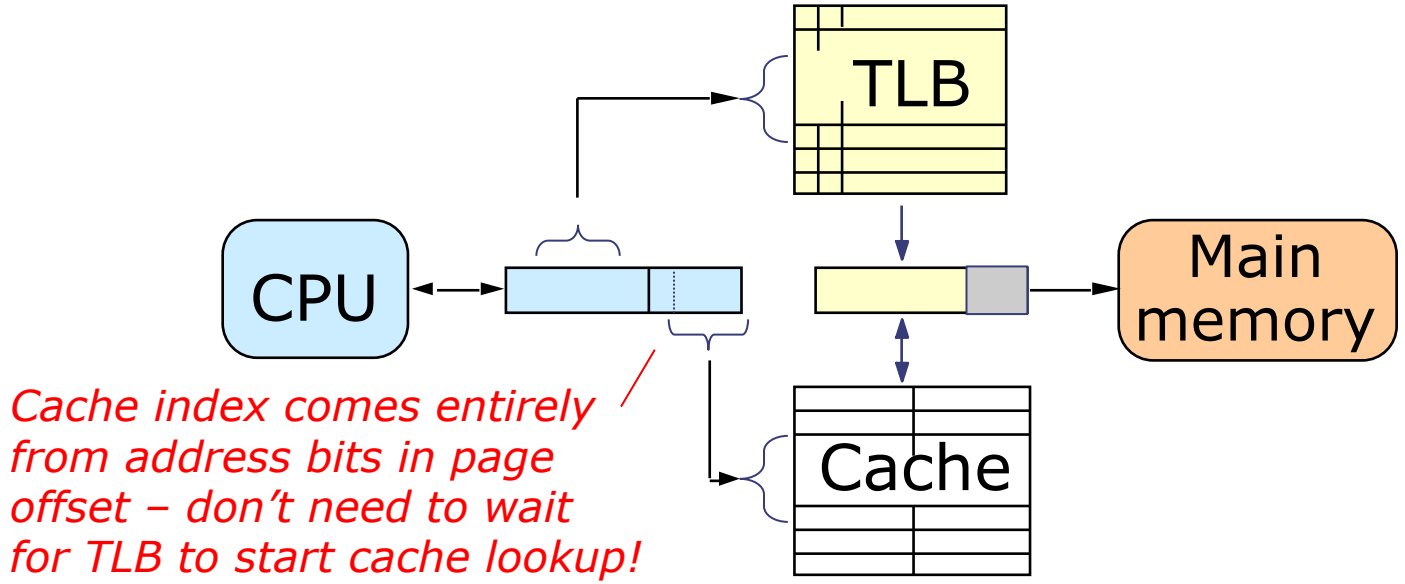
- FAST: No virtual→physical translation on cache hits
- Problem: Must flush cache after context switch

Physically-Addressed Cache



- Avoids stale cache data after context switch
- SLOW: Virtual→physical translation before every cache access

Best of Both Worlds: Virtually-Indexed, Physically-Tagged Cache (VIPT)



OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done *in parallel* with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!