Non-transient Side Channels

Mengjia Yan

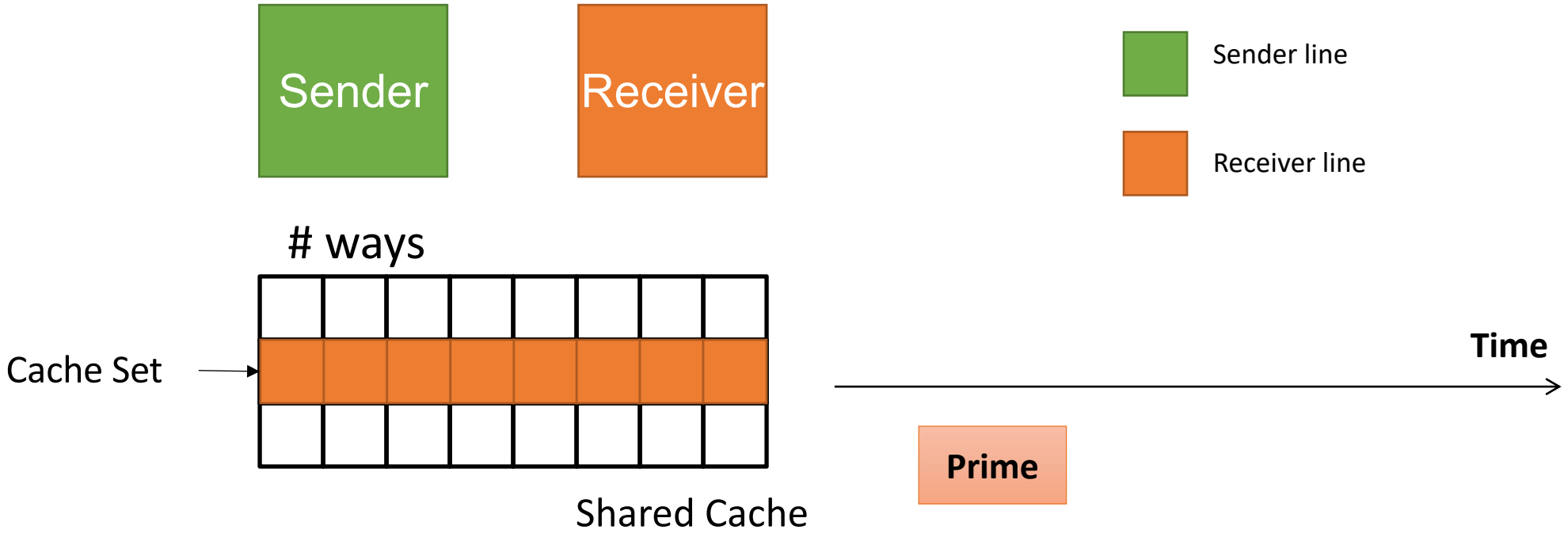
Fall 2020

Lab Assignment

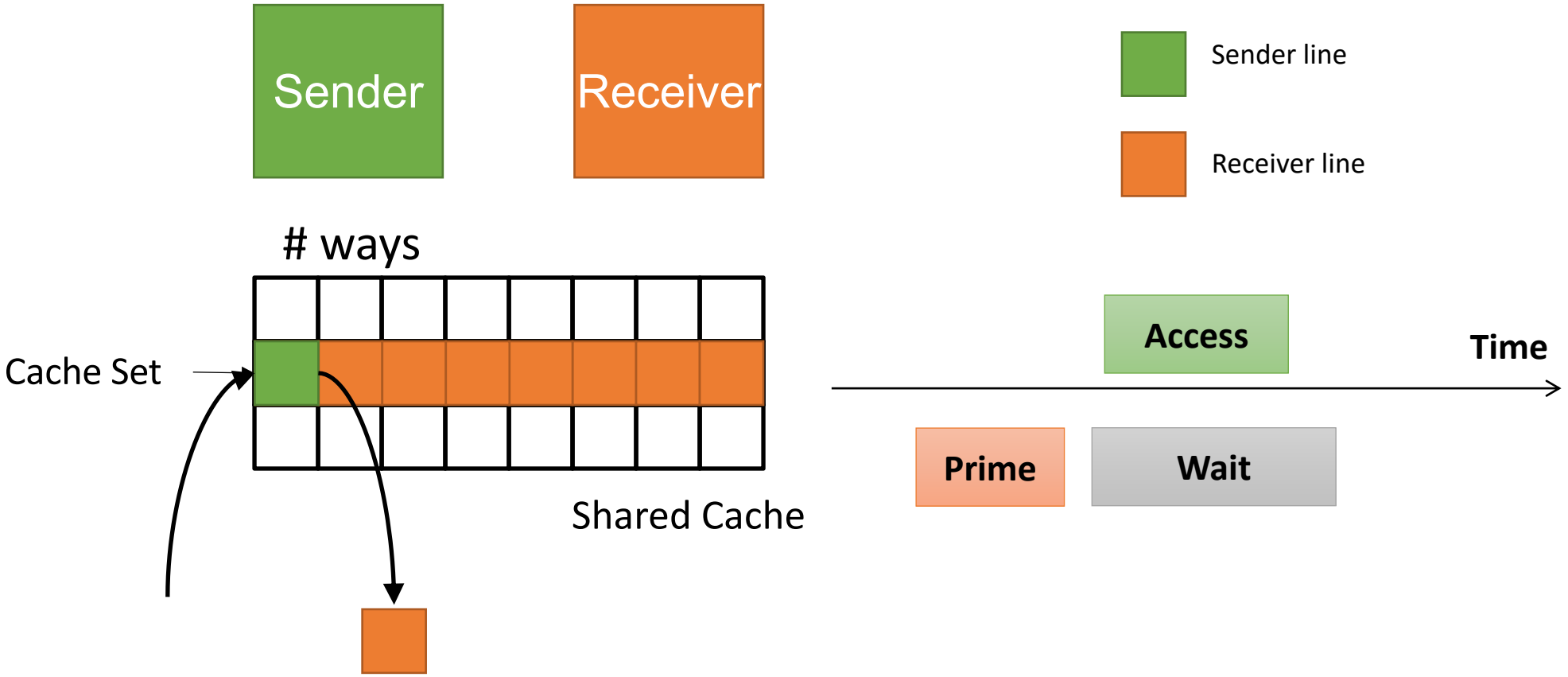
- Handout on course website
- Each (regular) student will receive an email
  - Solo or 2-person group
  - Individual GitHub repo
  - Info about accessing a server machine
- Listeners can send us an email if you want to try the lab

- Advice:
  - Start early. The first step is not to implement the attack, but to reverse engineer the machine.

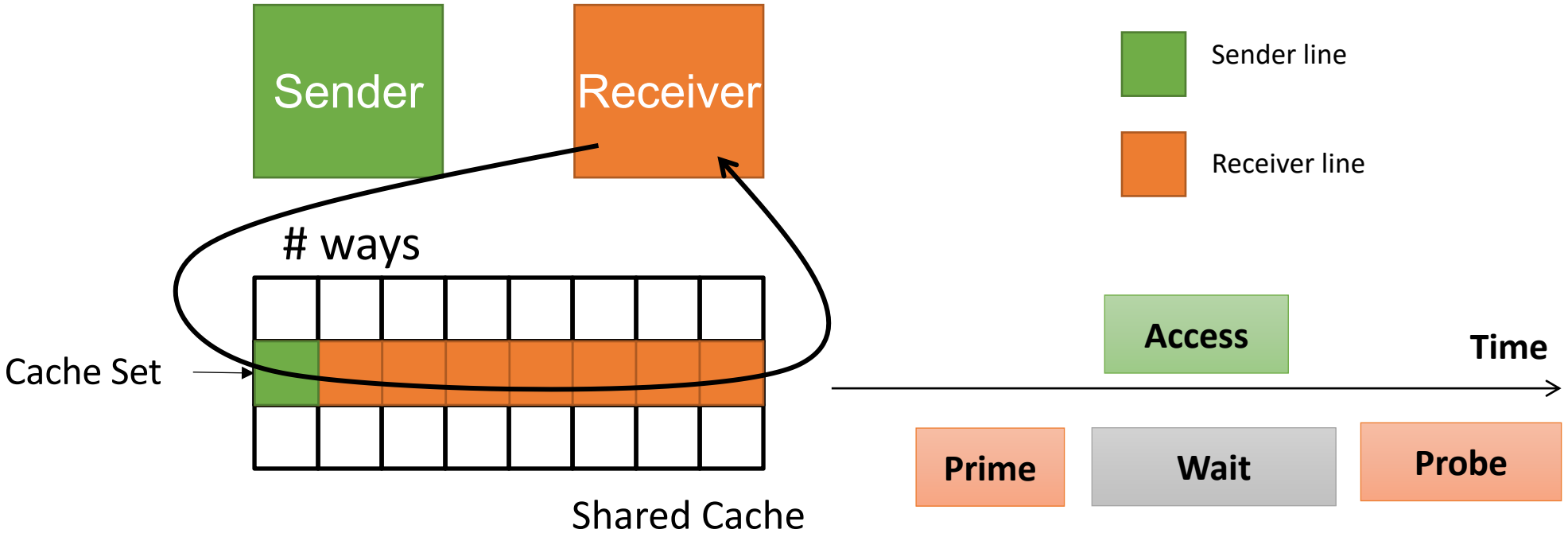
Recap: Prime+Probe



Recap: Prime+Probe

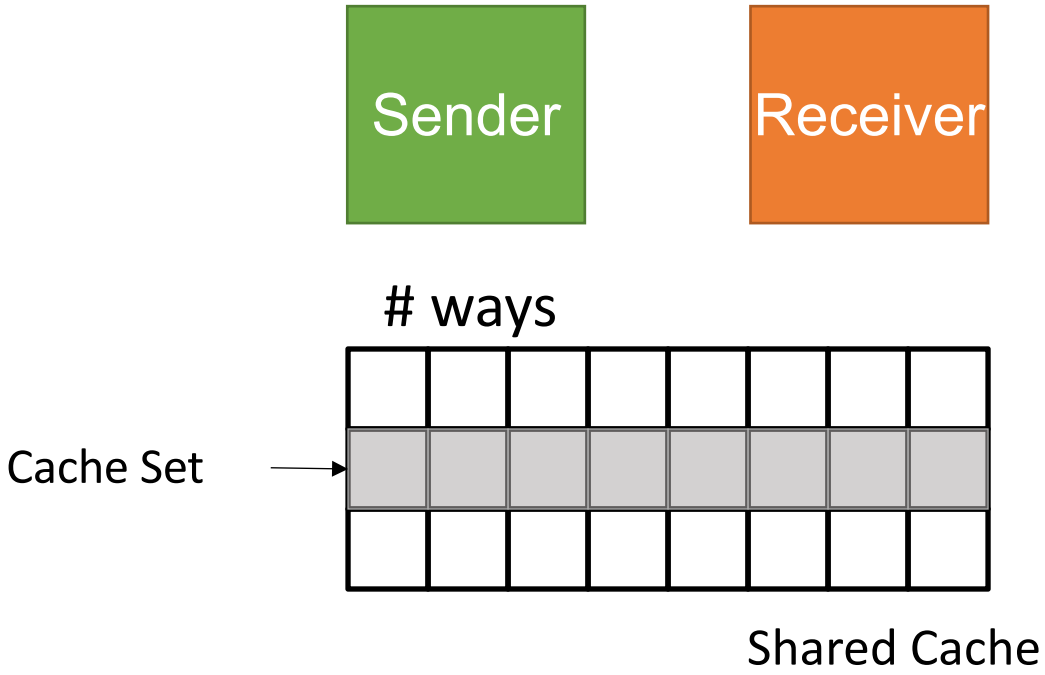


Recap: Prime+Probe

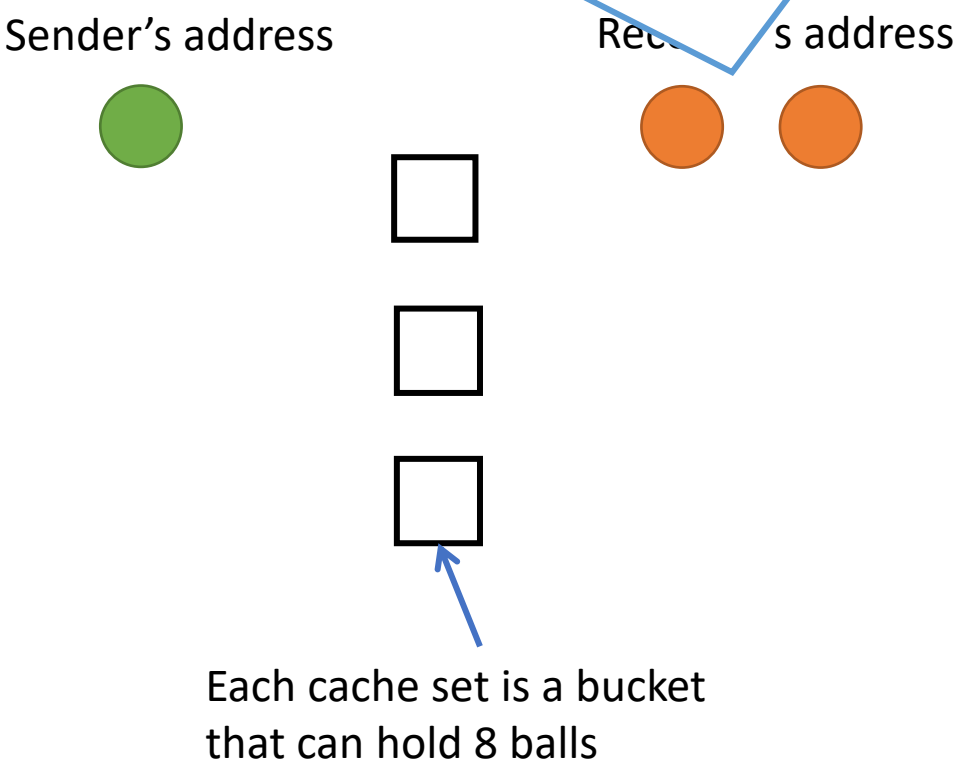


Receive “1” = 8 accesses → 1 miss

Analogy: Bucket/Ball



How many cache lines in total in the system?
How to find the bucket used by the sender?

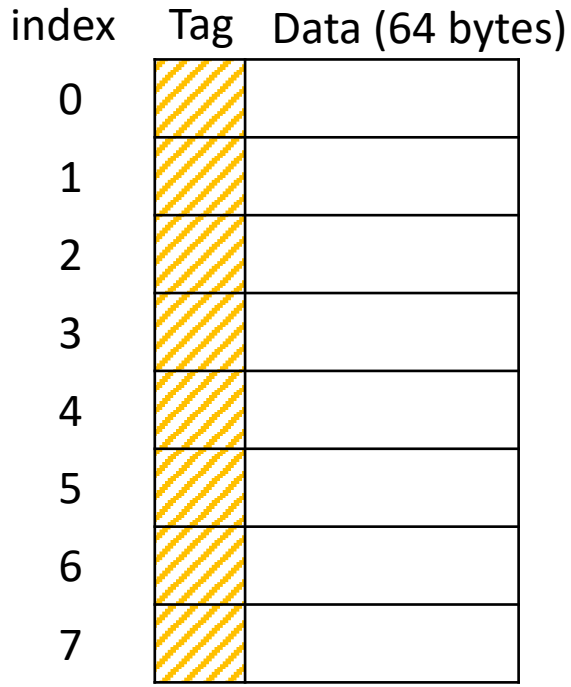


Practical Cache Side Channels

Cache Mapping – Directly Mapped Cache

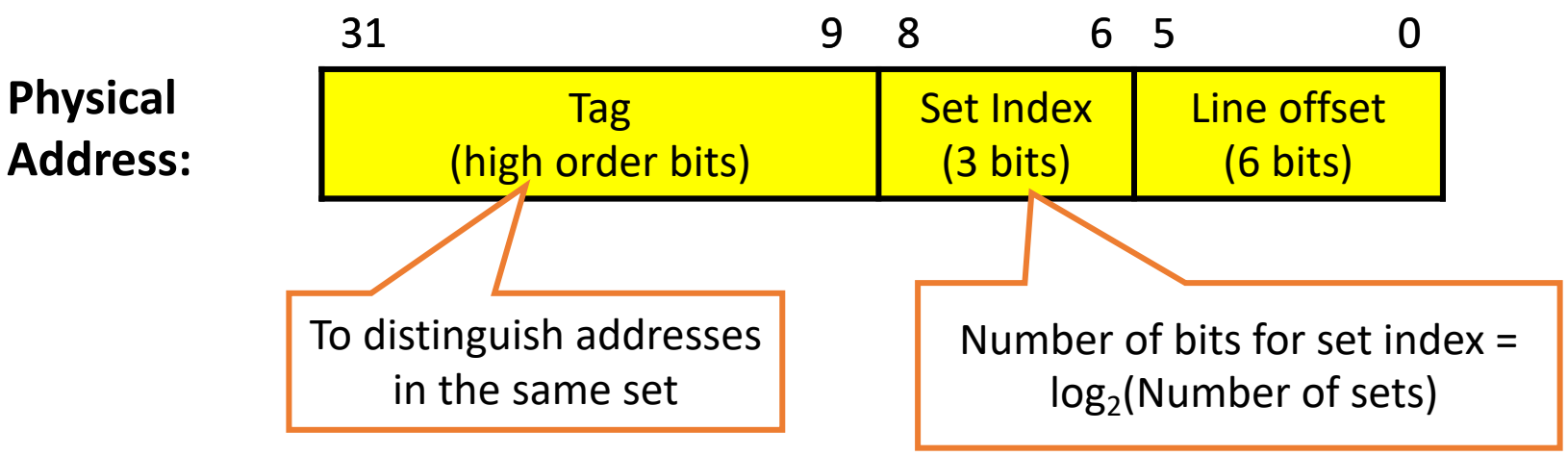
- Can think cache mapping as a hash table with limited size
- Linear cache set mapping using modular arithmetic

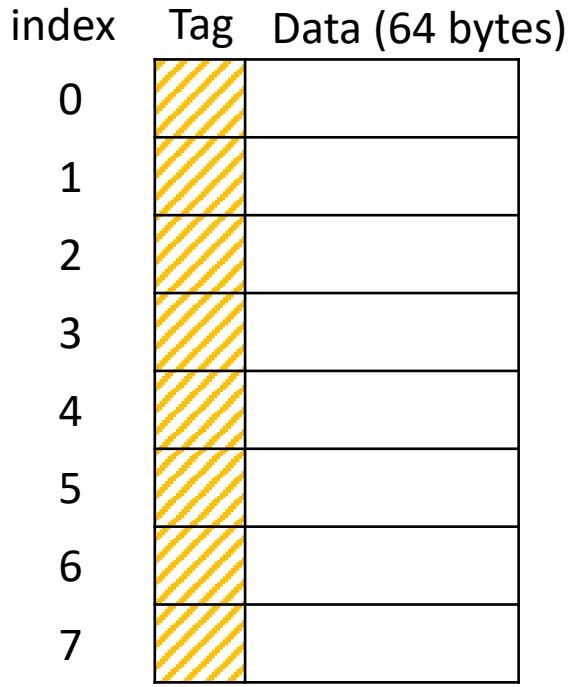
Set Index = (Addr / Block Size) % Number of Sets



Cache Mapping – Directly Mapped Cache

- Can think cache mapping as a hash table with limited size
- Linear cache set mapping using modular arithmetic *Assuming byte-addressable*

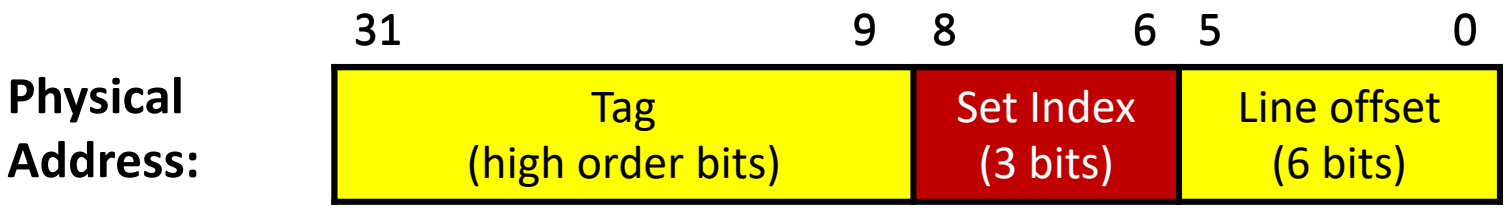




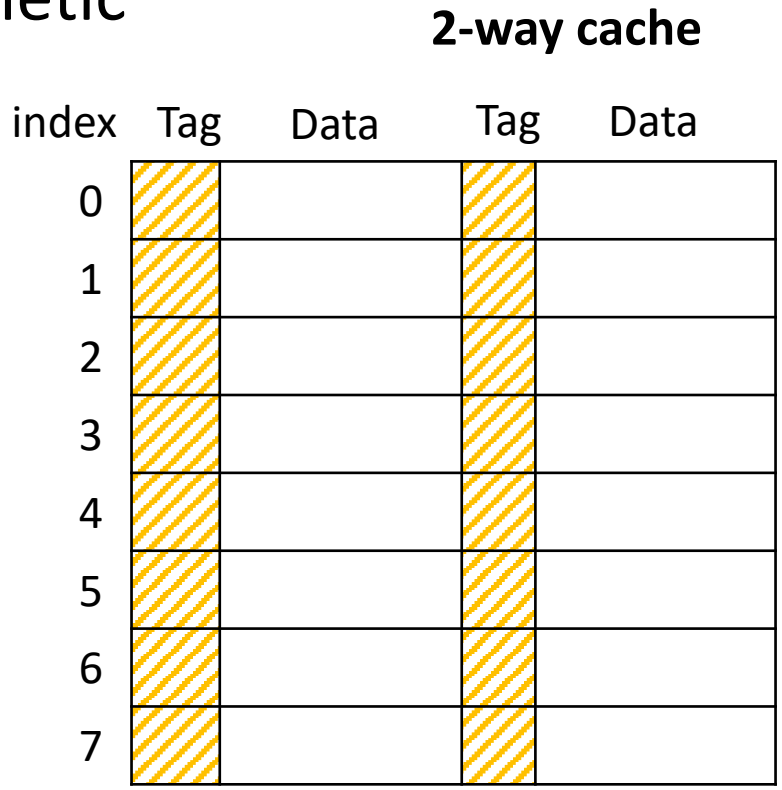
Question: Given an 1MB L2 with 1024 sets, how many bits are used for set index?

Cache Mapping – Set Associative Cache

- Can think cache mapping as a hash table with limited size
- Linear cache set mapping using modular arithmetic



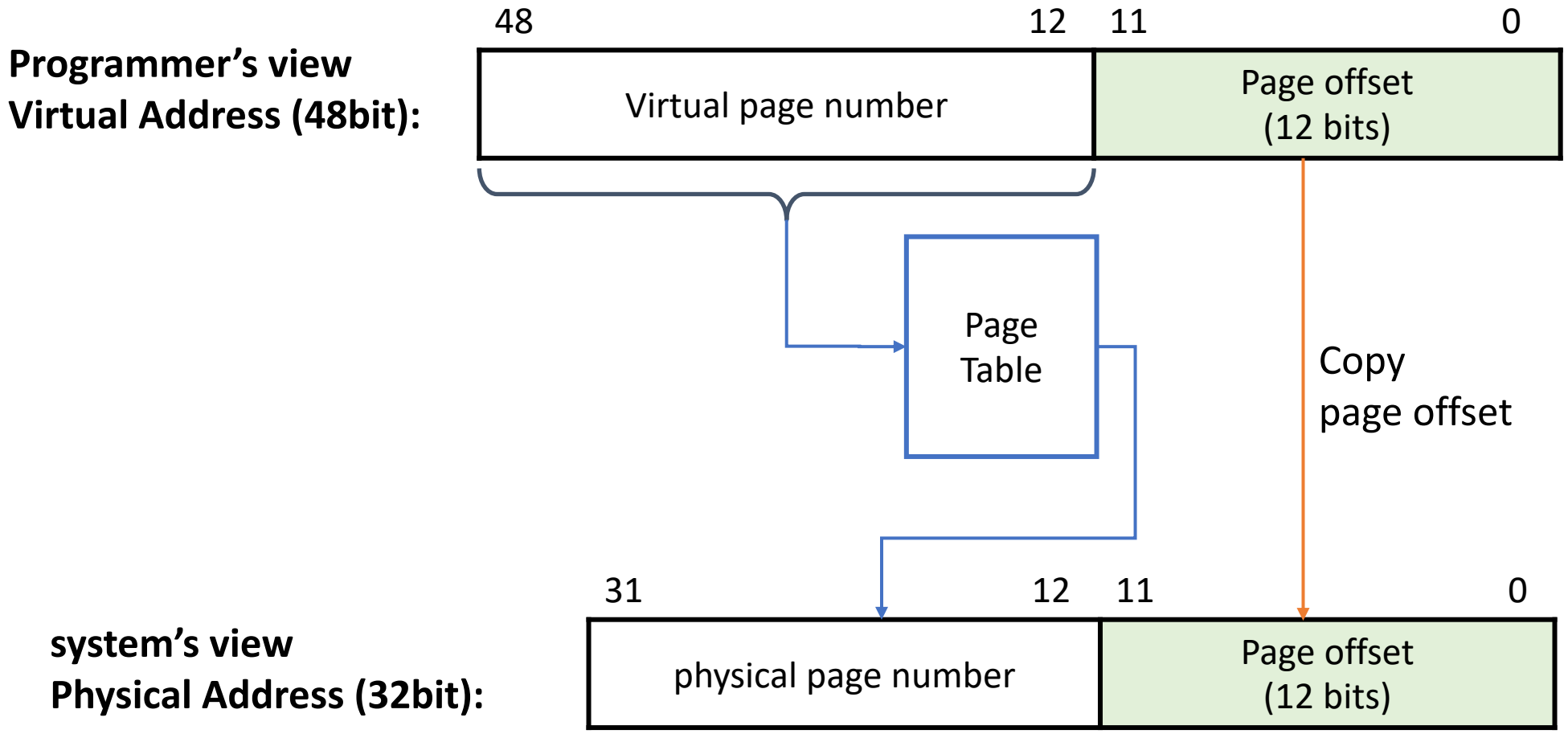
Find eviction set
==
Find addresses with the same set index bits



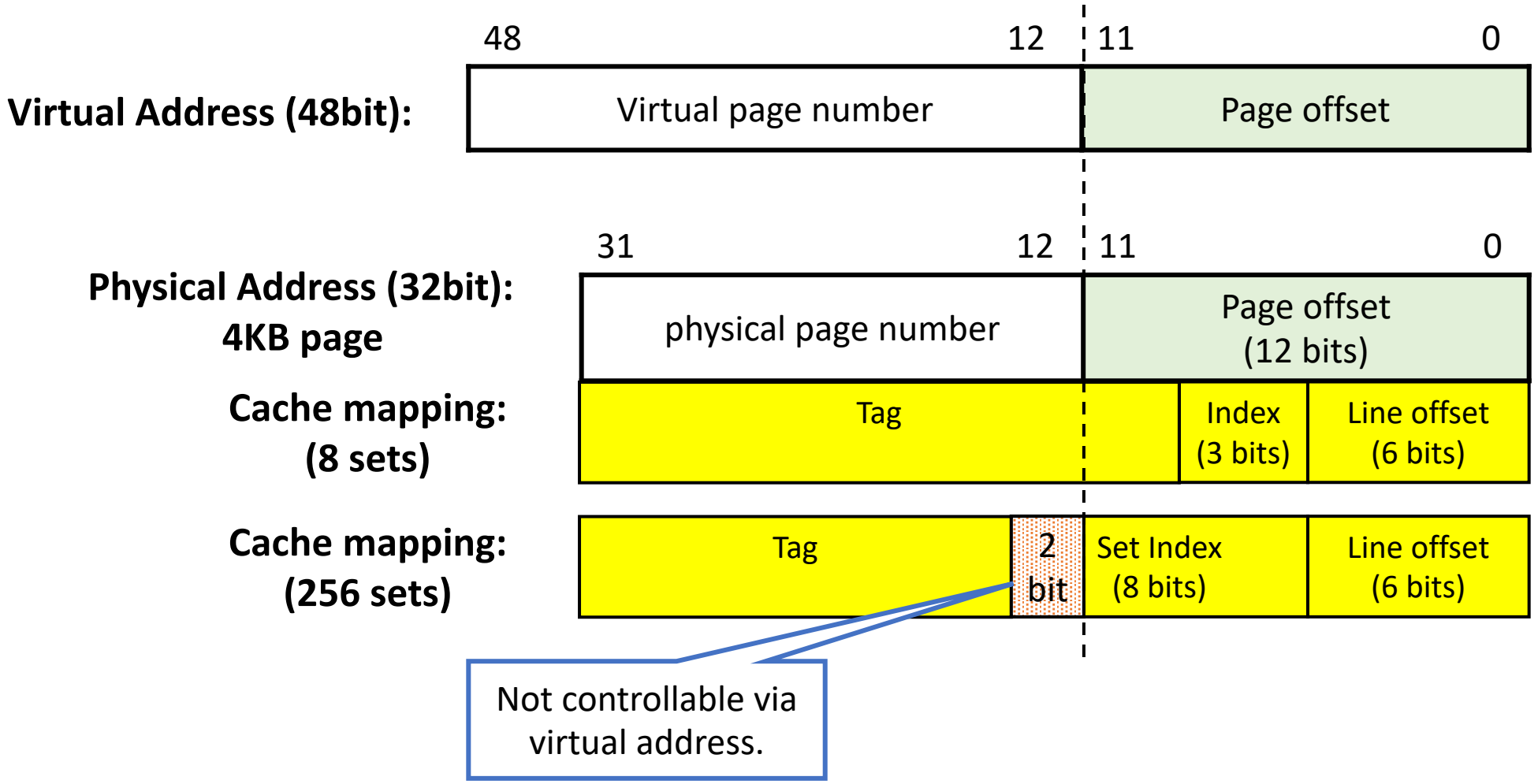
Question: How to decide which way to use?

Answer: Cache replacement policy.

Address Translation (4KB page)



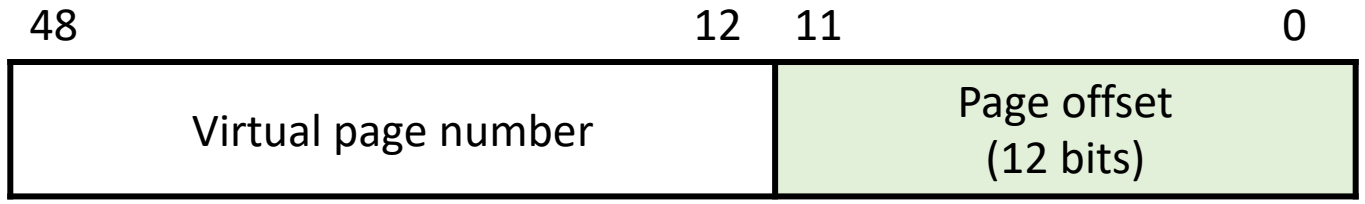
Find Eviction Set Using Virtual Addresses



Huge Pages

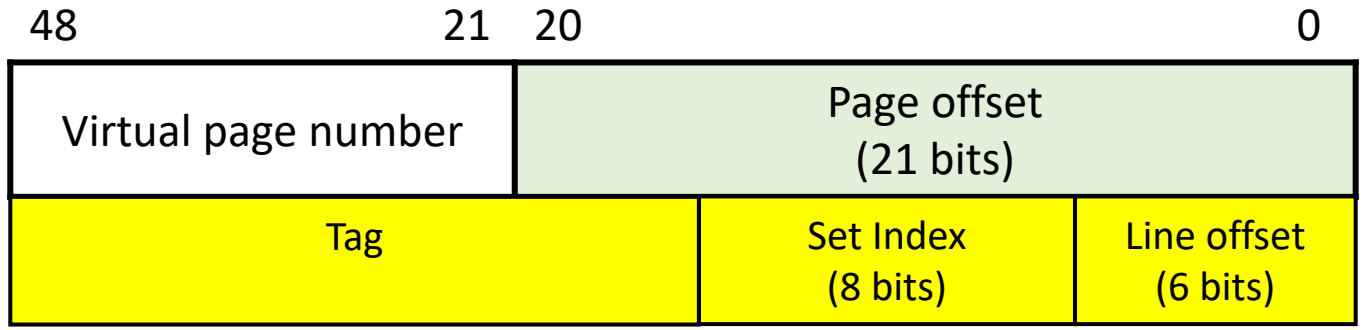
- Huge page size: 2MB or 1GB
  - Number of bits for page offset?

Virtual Address : 4KB page

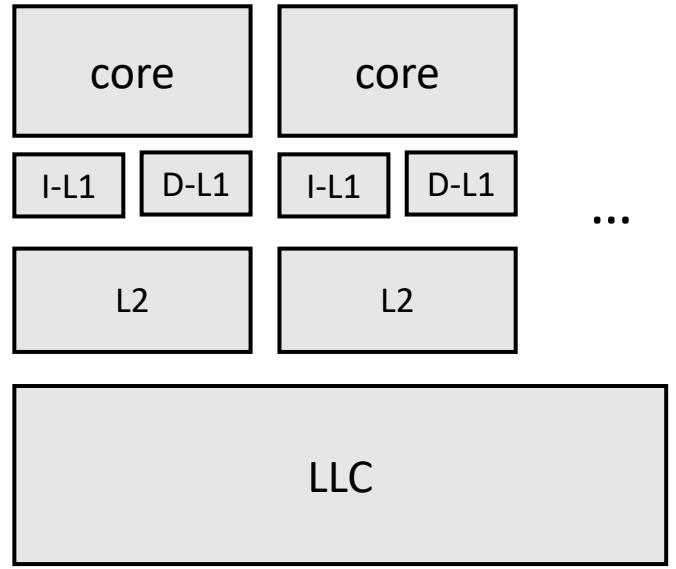


Virtual Address : 2MB page

Cache mapping: (256 sets)



Multi-level Caches



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

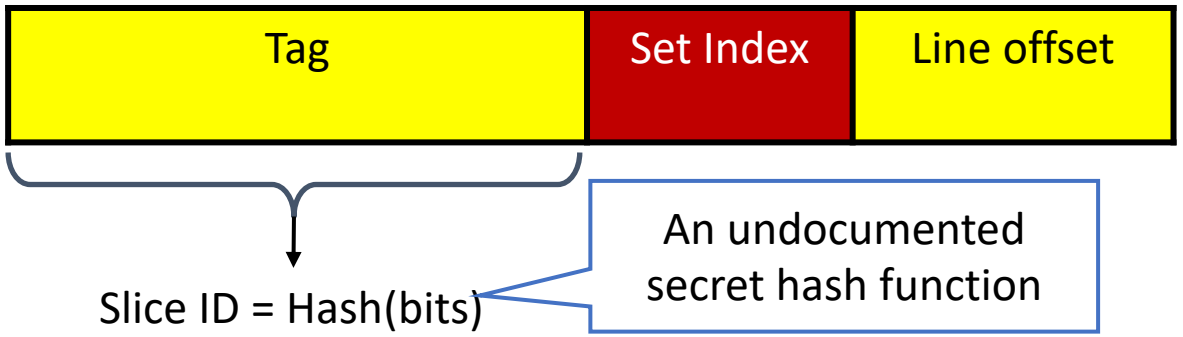
A typical configuration of Intel Ivy Bridge.
Configurations are different with processor types.

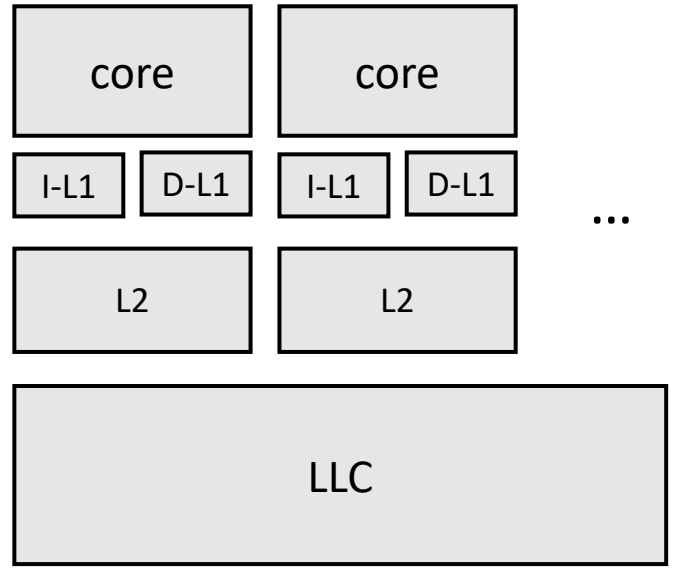
	L1-I/D cache	L2 cache	L3 cache (LLC)	DRAM
Size	32KB	256KB	1MB/core	16GB
Associativity (# ways)	4 or 8	8	16	N/A
Latency (cycles)	1-5	12	~40	~150

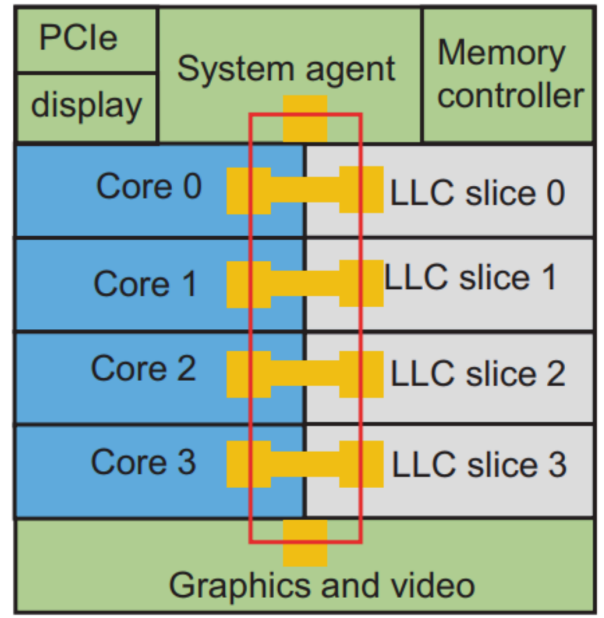
Multi-level Caches

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

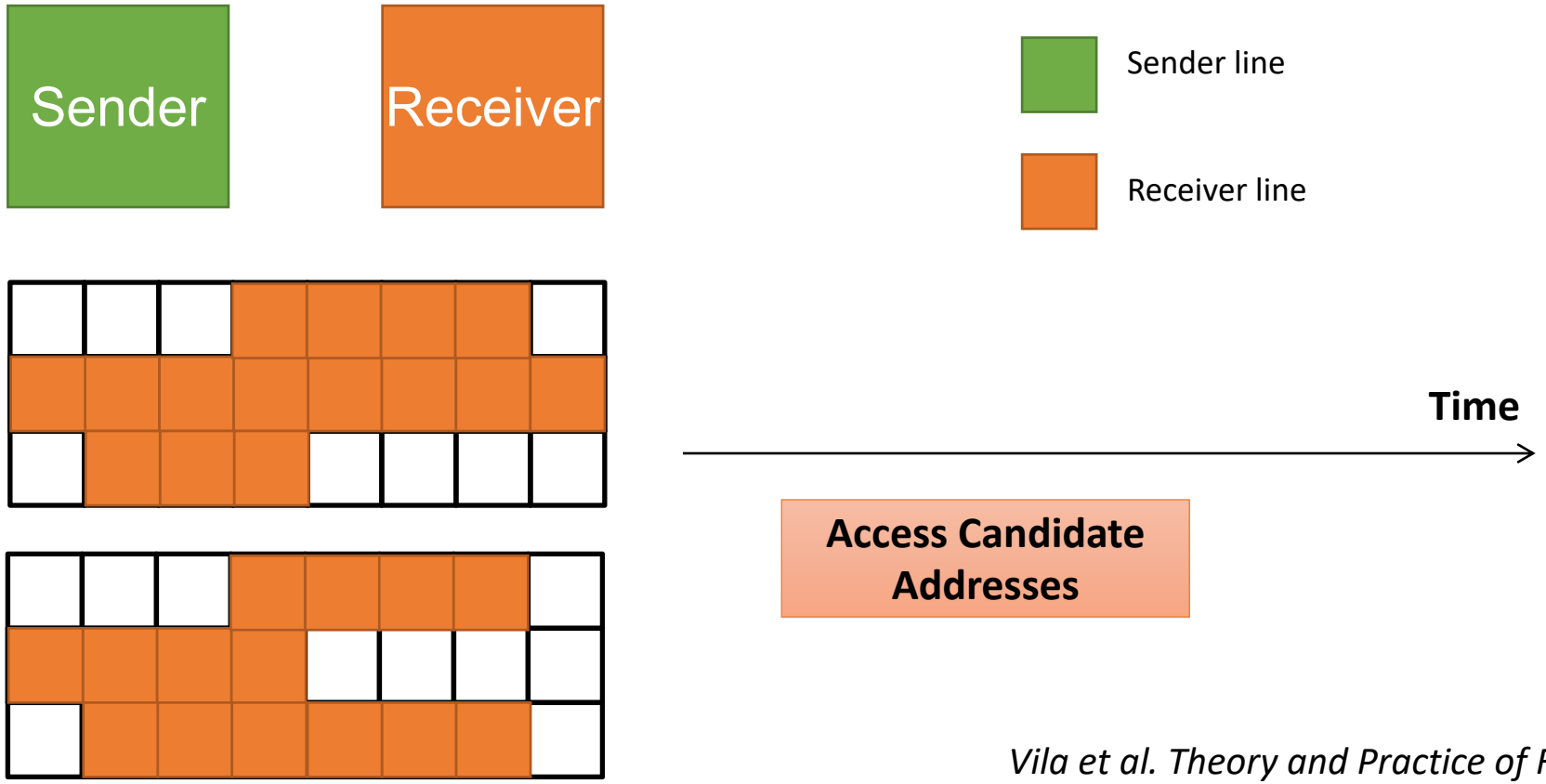
- LLC is generally divided into multiple slices
  - Conflict happens if addresses map to **the same slice and the same set**





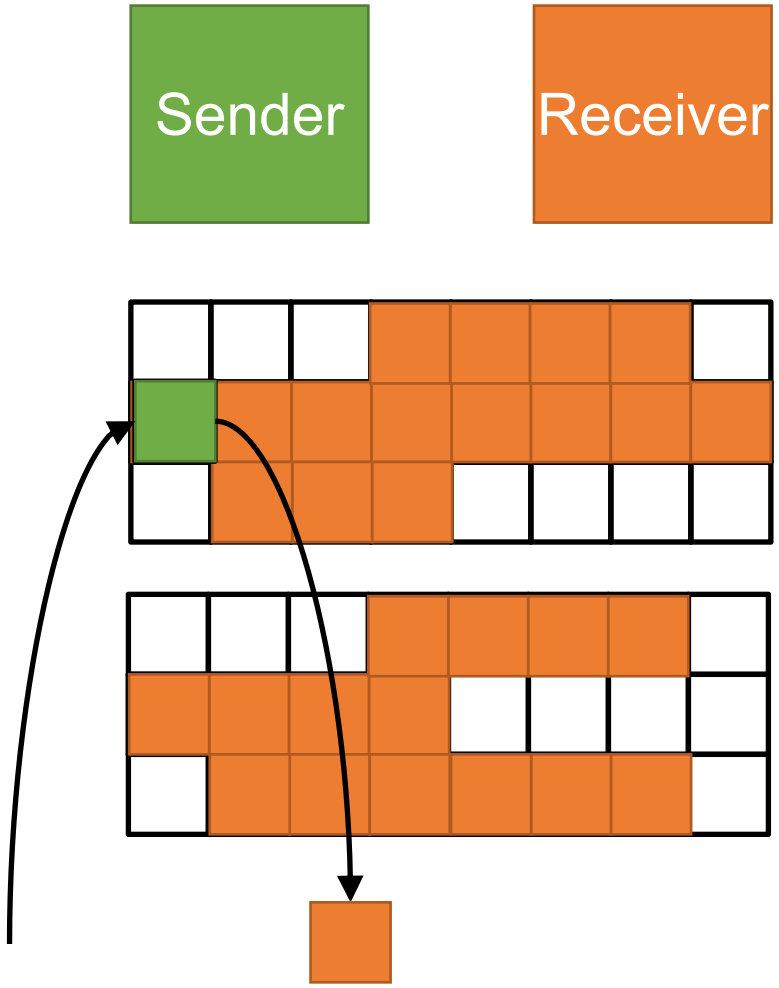


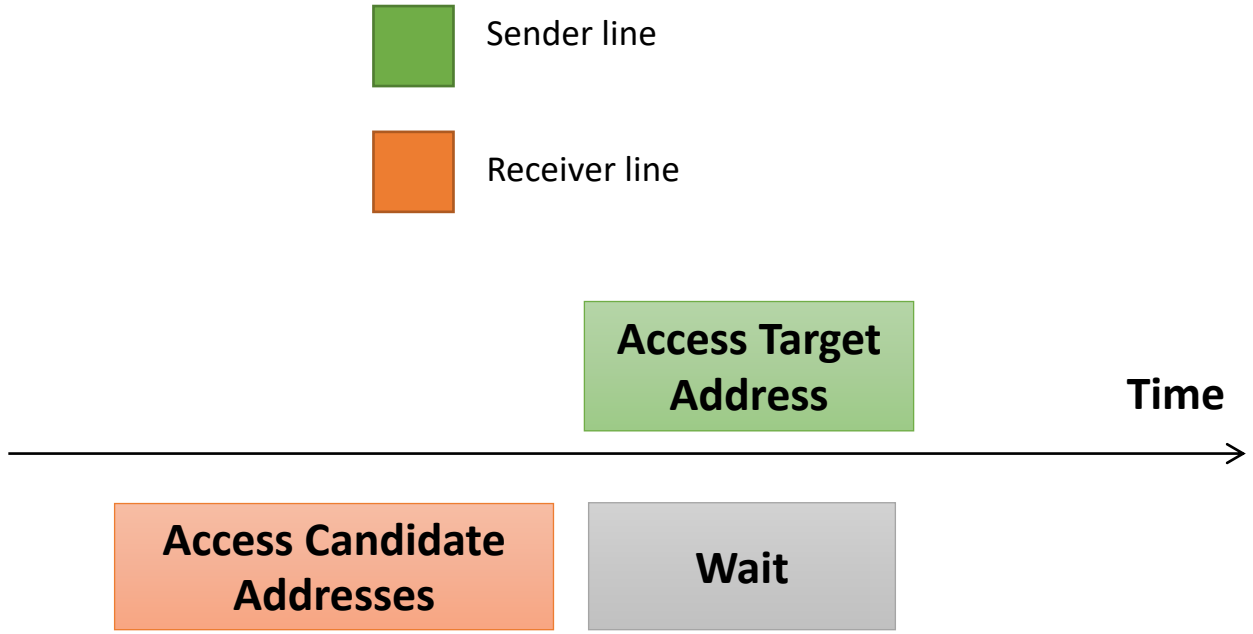
Eviction Set Construction Algorithm



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

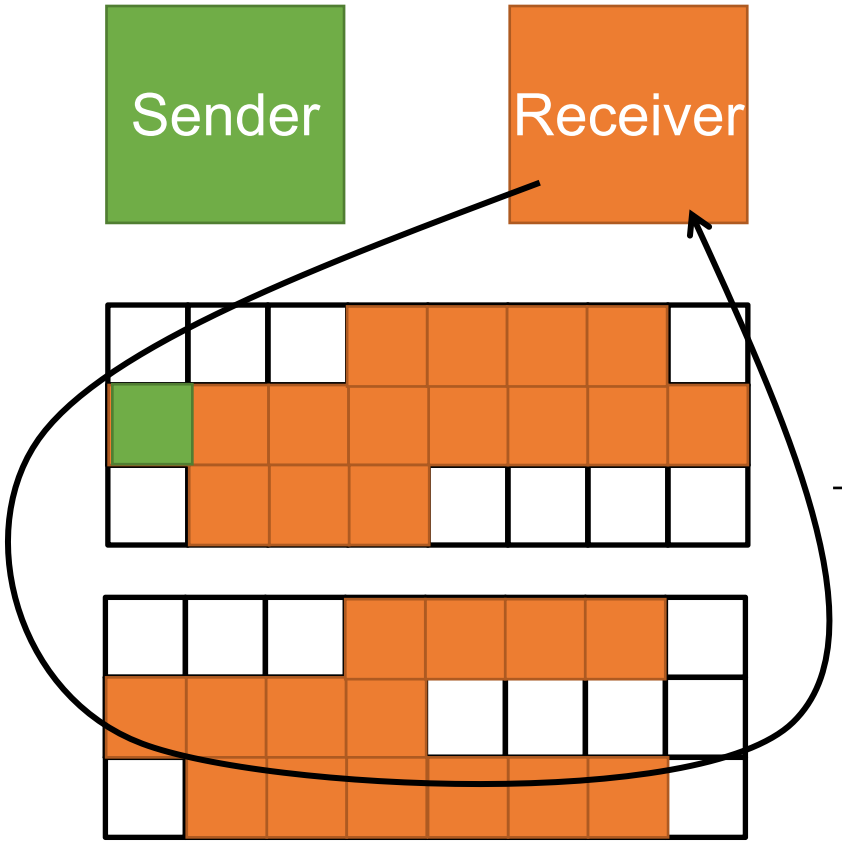
Eviction Set Construction Algorithm

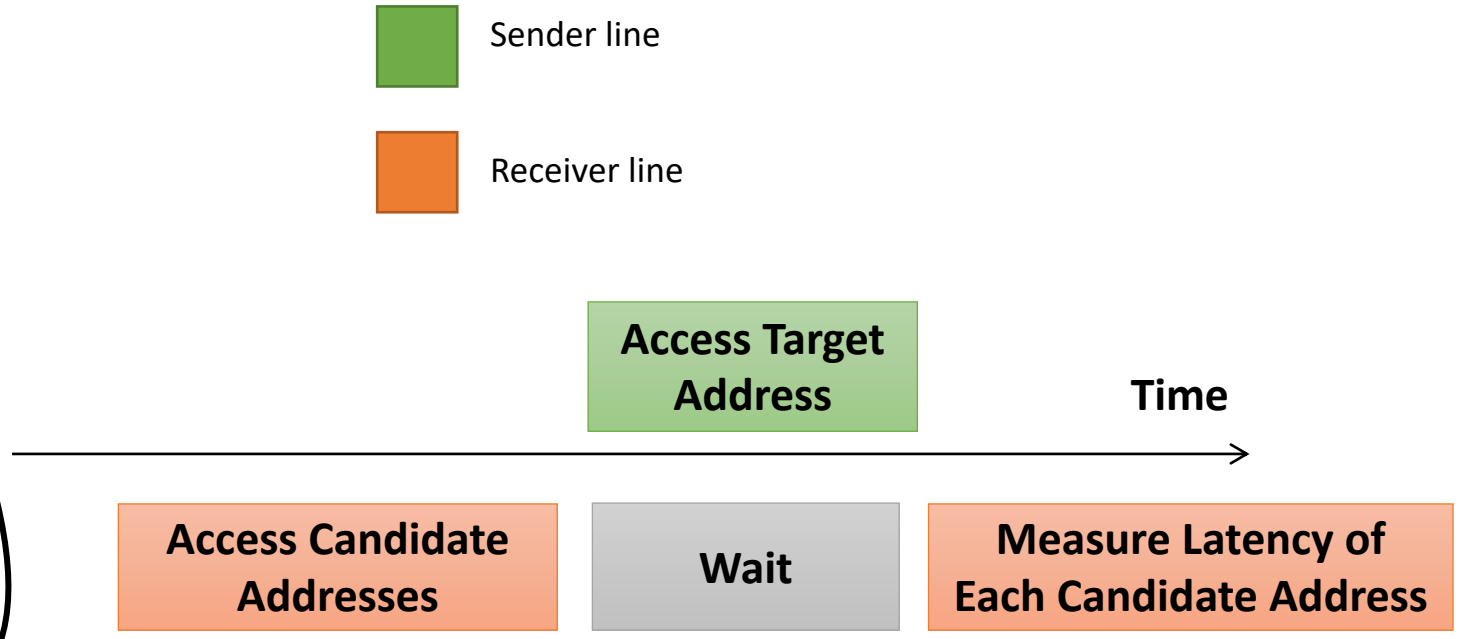




Vila et al. Theory and Practice of Finding Eviction Sets. S&P'19

Eviction Set Construction Algorithm

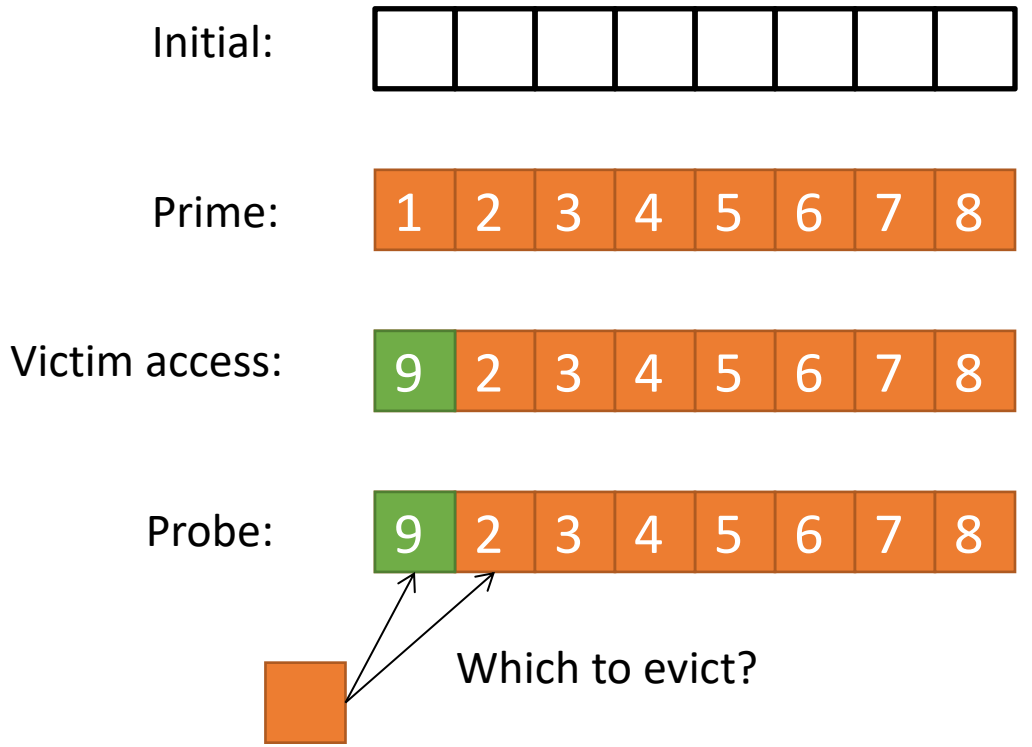




Vila et al. Theory and Practice of Finding Eviction Sets. S&P'19

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

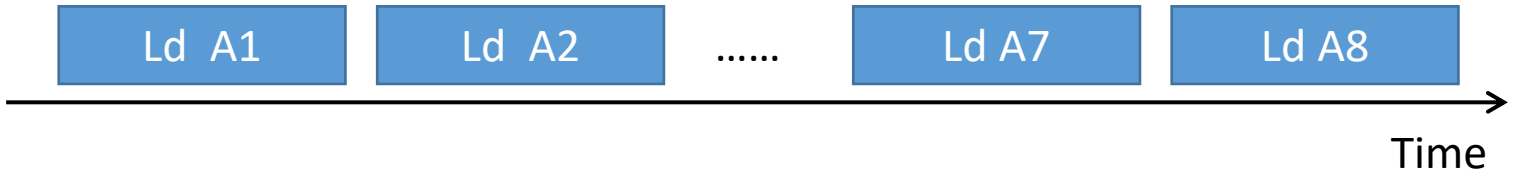


Measure Latency of Multiple Accesses

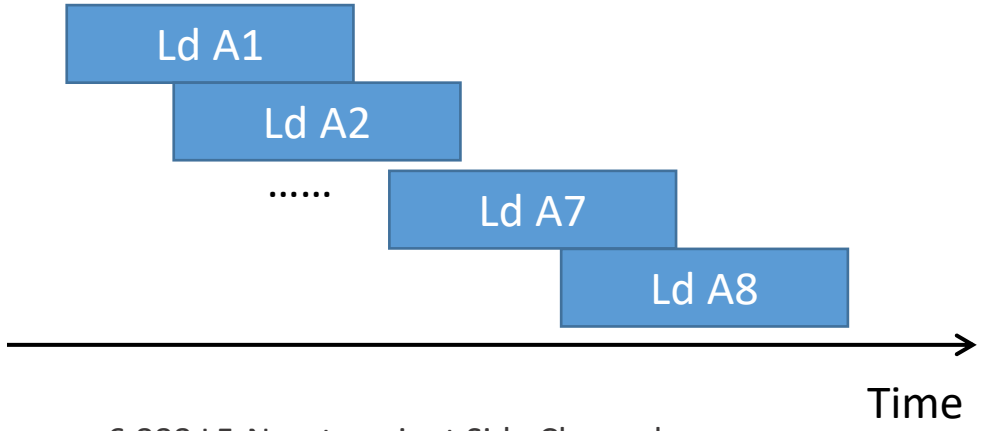
- HW Prefetcher + Out-of-order execution

```
T1 = rdtsc()
Dummy1=Ld(Addr1)
……
Dummy8=Ld(Addr8)
T2 = rdtsc()
Latency = T2-T1
```

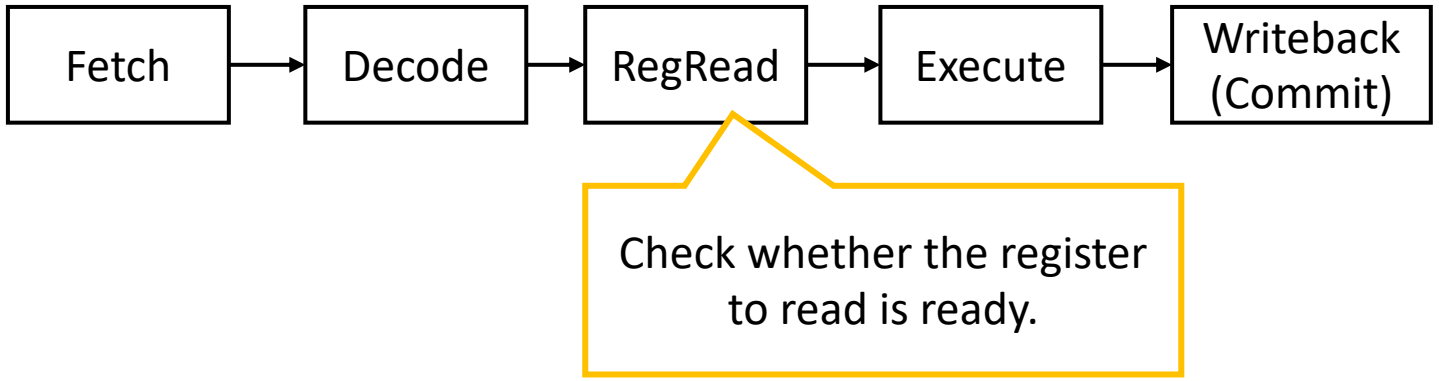
What we expect:

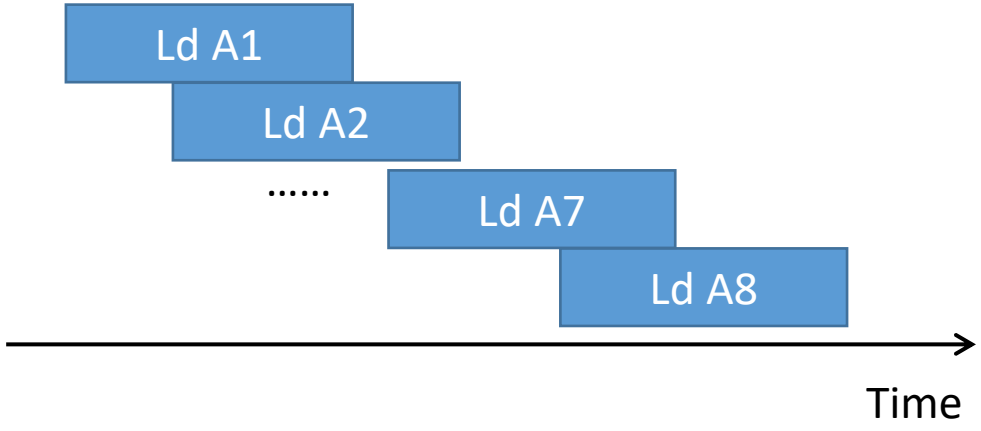


What actually will happen:



Out-of-Order Processor

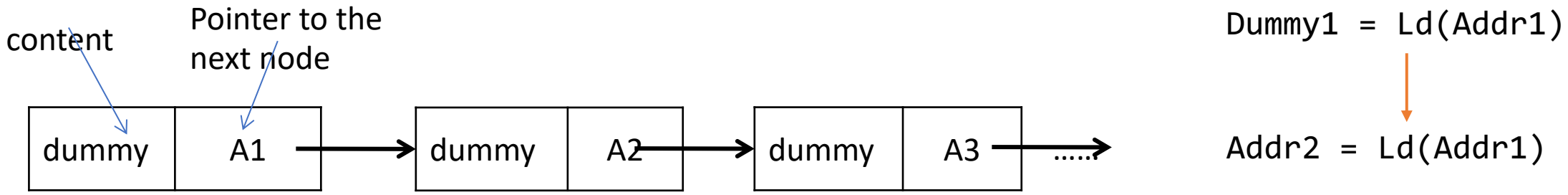




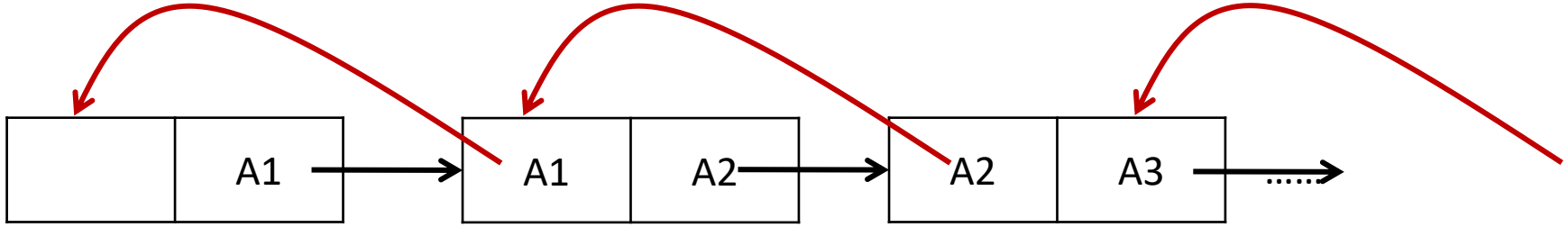
Question: How to serialize data accesses?

Serialize Data Accesses

- A special instruction “mfence” *https://www.felixcloutier.com/x86/mfence*
- Add data dependency by creating a linked list



- Double linked list to access addresses in reverse order



Handle Noise

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

What you generally see in papers:

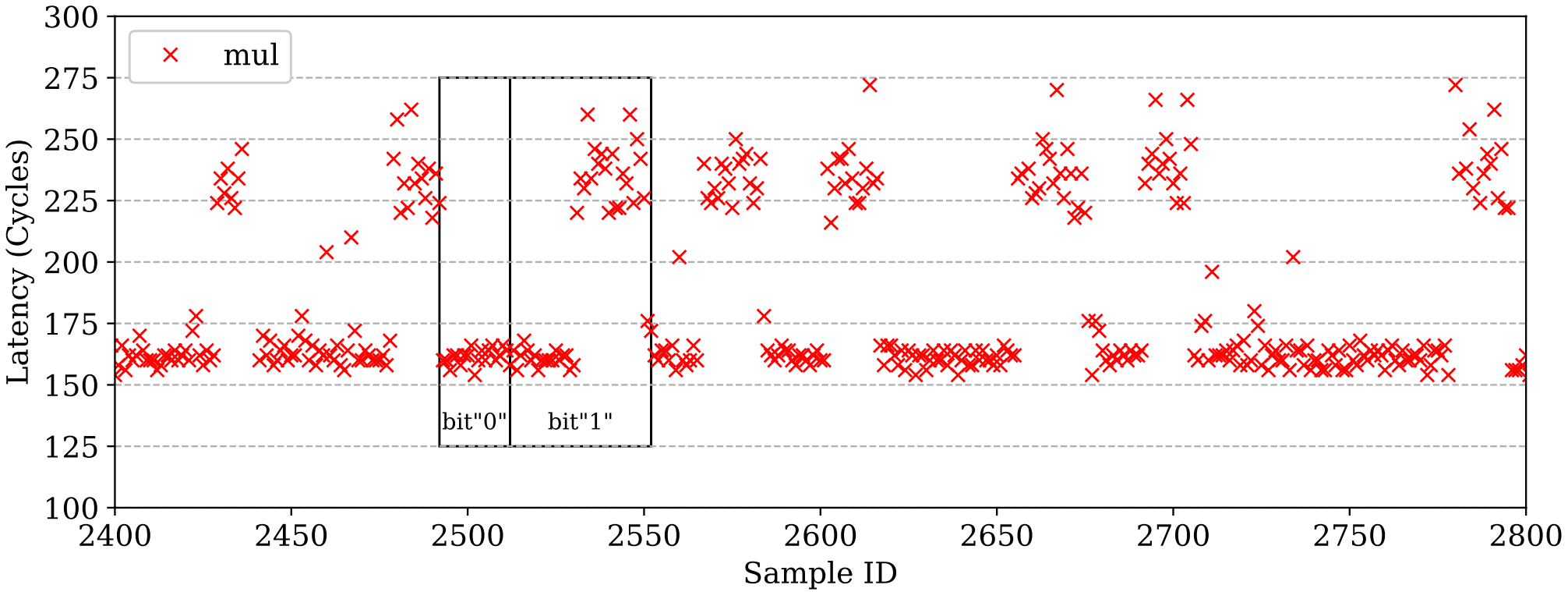
```
for i = n-1 to 0 do
        r = sqr(r) mod n
        if e_i == 1 then
            r = mul(r, b) mod n
        end
end
```

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



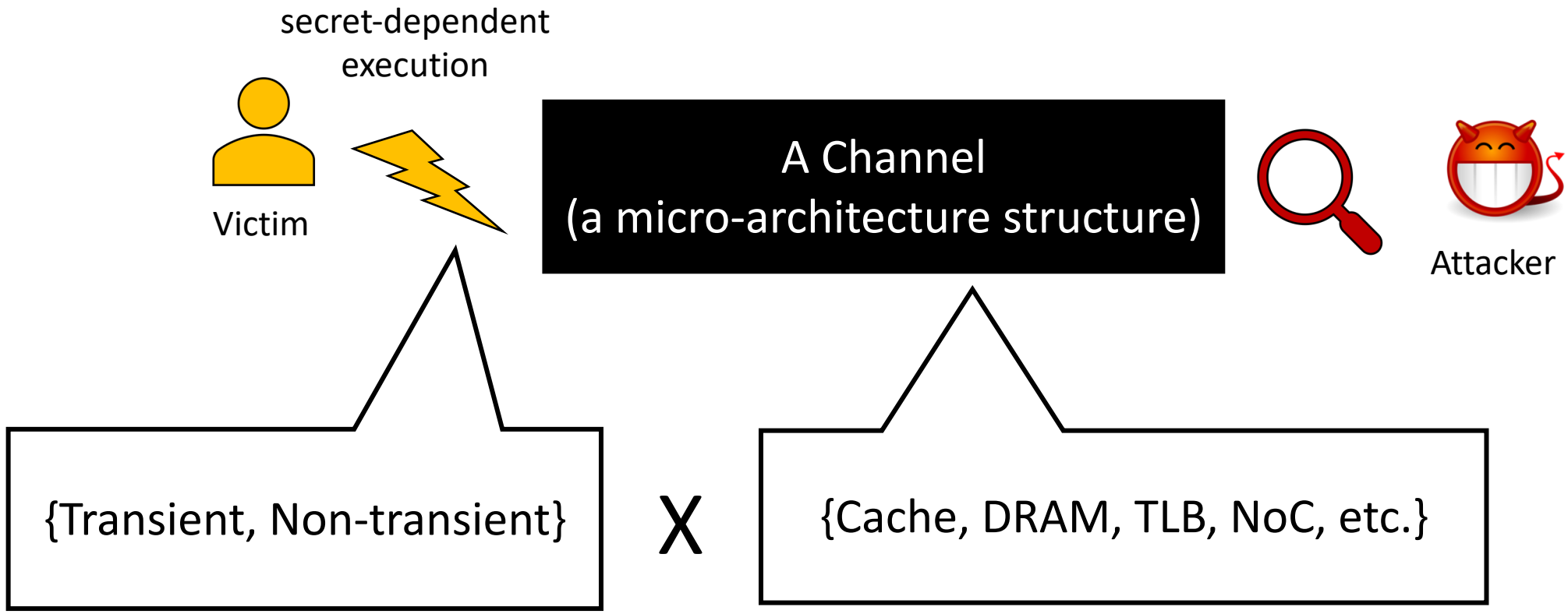
Access latencies measured in the probe operation in Prime+Probe.
A sequence of "01010111011001" can be deduced as part of the exponent.

There may exist other problems

- Tips for lab assignment
  - Build the attack step-by-step
  - Recommend to read “Last-Level Cache Side-Channel Attacks are Practical”
  - Ask questions via Piazza

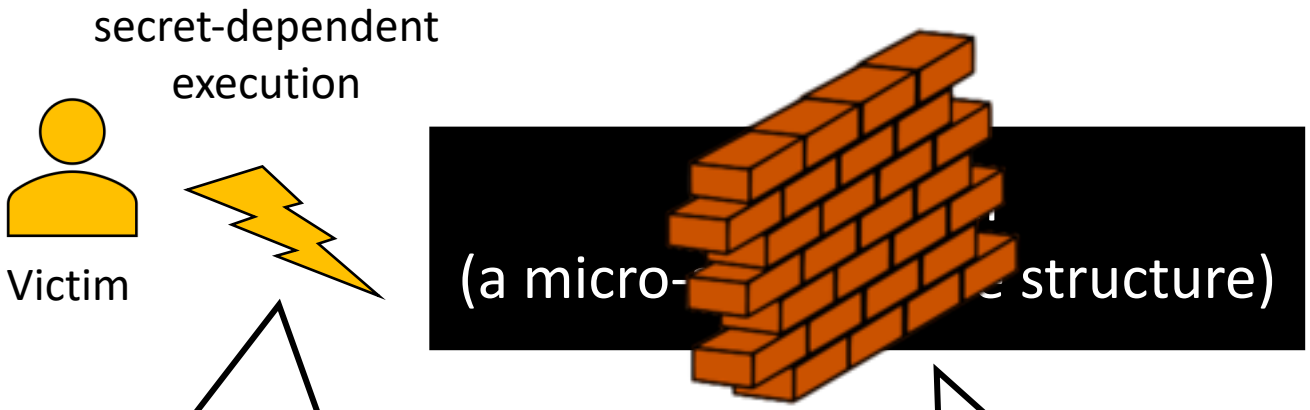
Defenses

Micro-architecture Side Channels



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

Micro-architecture Side Channels



Defenses:

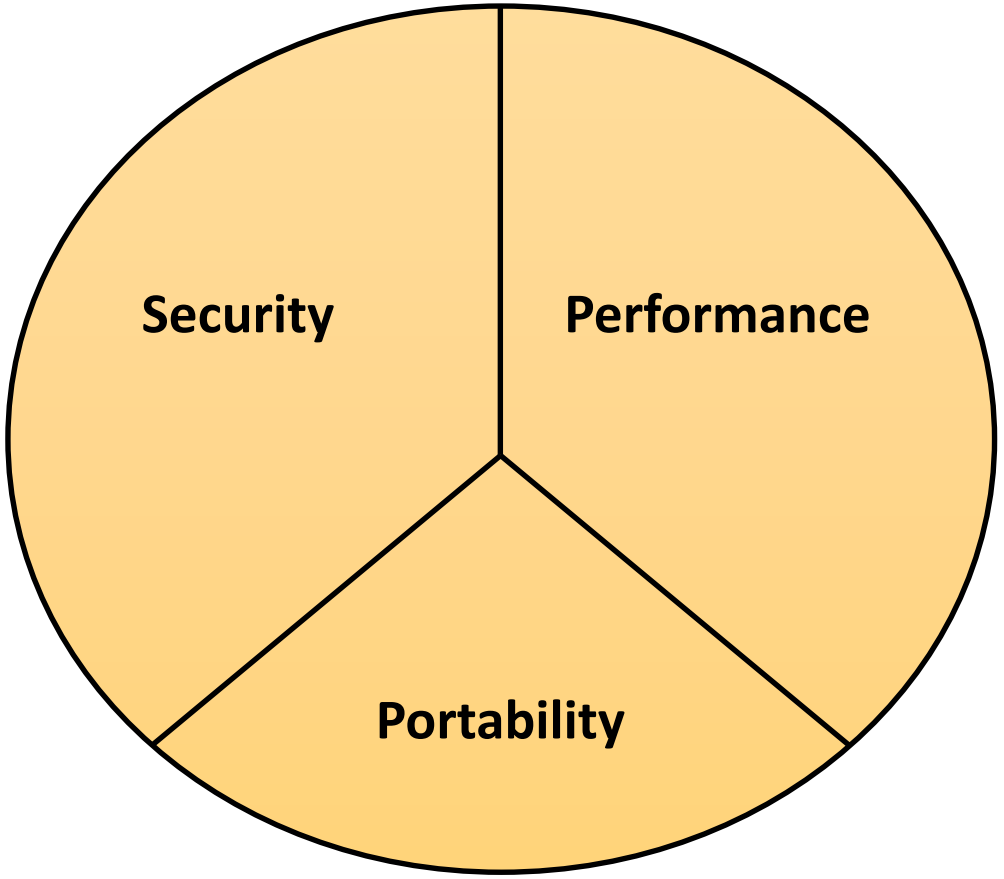
Block creation of signals:
Oblivious execution,
speculative execution defenses, etc.

Close the channel:
Isolation, etc.

Block detection of signals:
Randomization, etc.

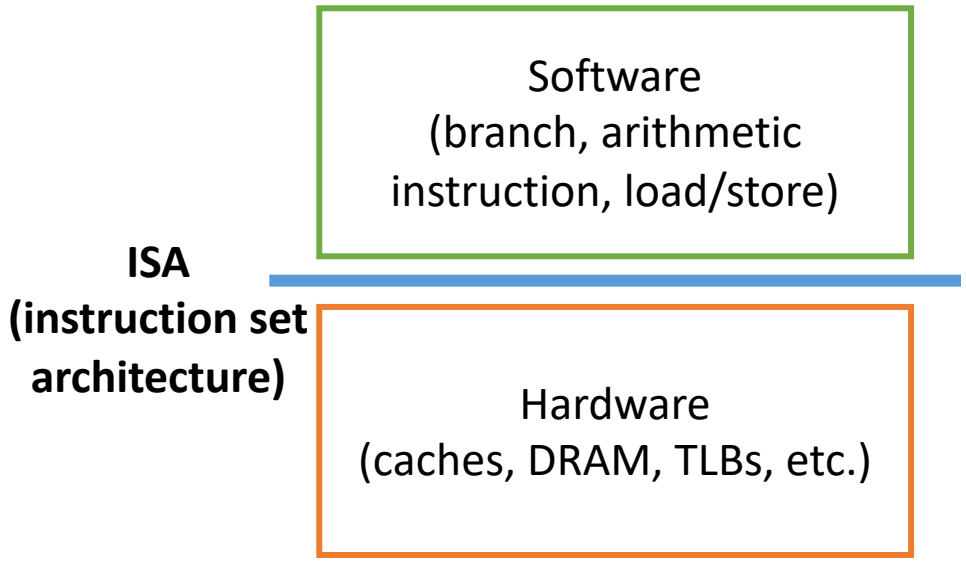
Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Defense Design Considerations



The Problem: The ISA Abstraction

- Interface between HW and SW: ISA
  - Advantage: HW optimizations without affecting usability/portability



DEC — Decrement by 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FE /1	DEC *r/m8*	M	Valid	Valid	Decrement *r/m8* by 1.
REX + FE /1	DEC *r/m8**	M	Valid	N.E.	Decrement *r/m8* by 1.
FF /1	DEC *r/m16*	M	Valid	Valid	Decrement *r/m16* by 1.
FF /1	DEC *r/m32*	M	Valid	Valid	Decrement *r/m32* by 1.
REX.W + FF /1	DEC *r/m64*	M	Valid	N.E.	Decrement *r/m64* by 1.
48+rw	DEC *r16*	O	N.E.	Valid	Decrement *r16* by 1.
48+rd	DEC *r32*	O	N.E.	Valid	Decrement *r32* by 1.

* In64-bitmode,r/m8cannotbeencodedtoaccessthefollowingbyteregistersifaREXprefixisused:AH,BH,CH,DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

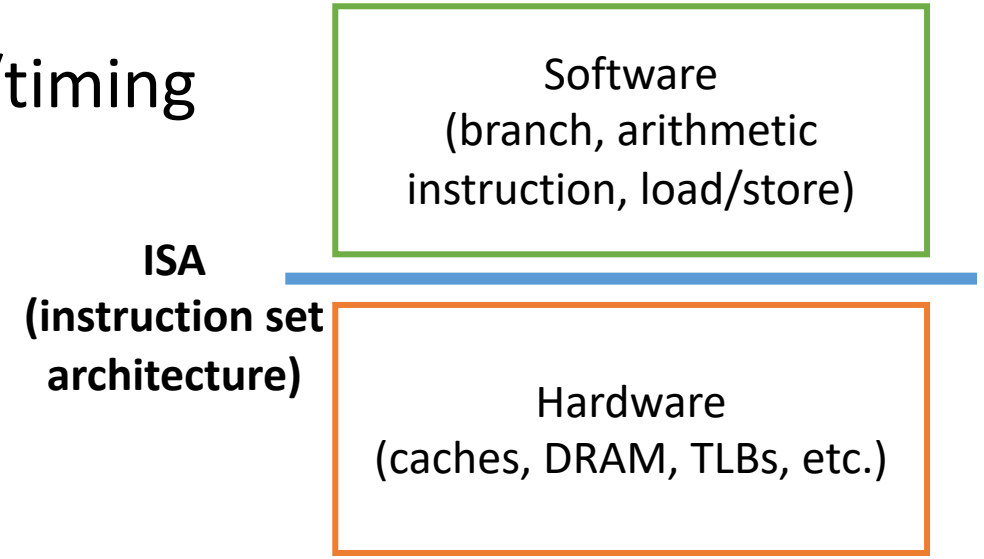
From https://www.felixcloutier.com/x86/index.html

```
DEST ← DEST – 1;
```

The Problem: The ISA Abstraction

- Interface between HW and SW: ISA

- ISA specifies functionality, not performance/timing
  - Compare Intel Ivy Bridge and Cascade Processor



Example:

DEC [addr]

Data Oblivious/“Constant time” Programming

Write program w/o data-dependent behavior

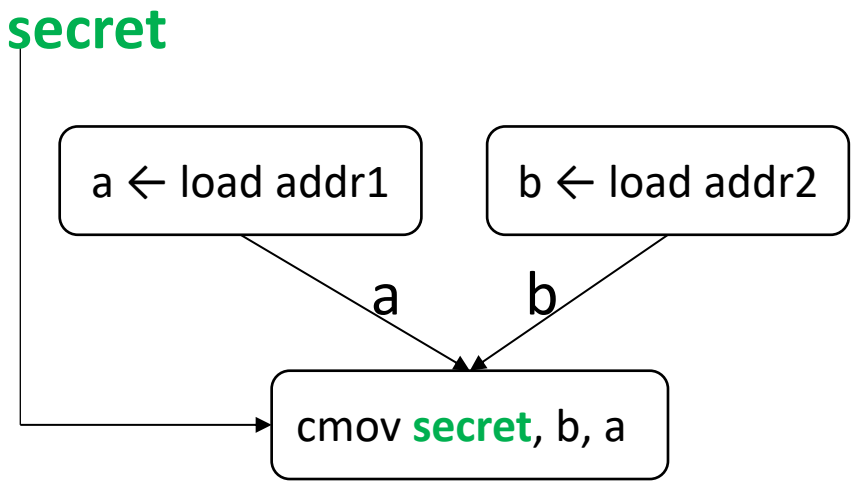
Original:

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

secret = confidential
addr1 = public
addr2 = public

Data Oblivious:

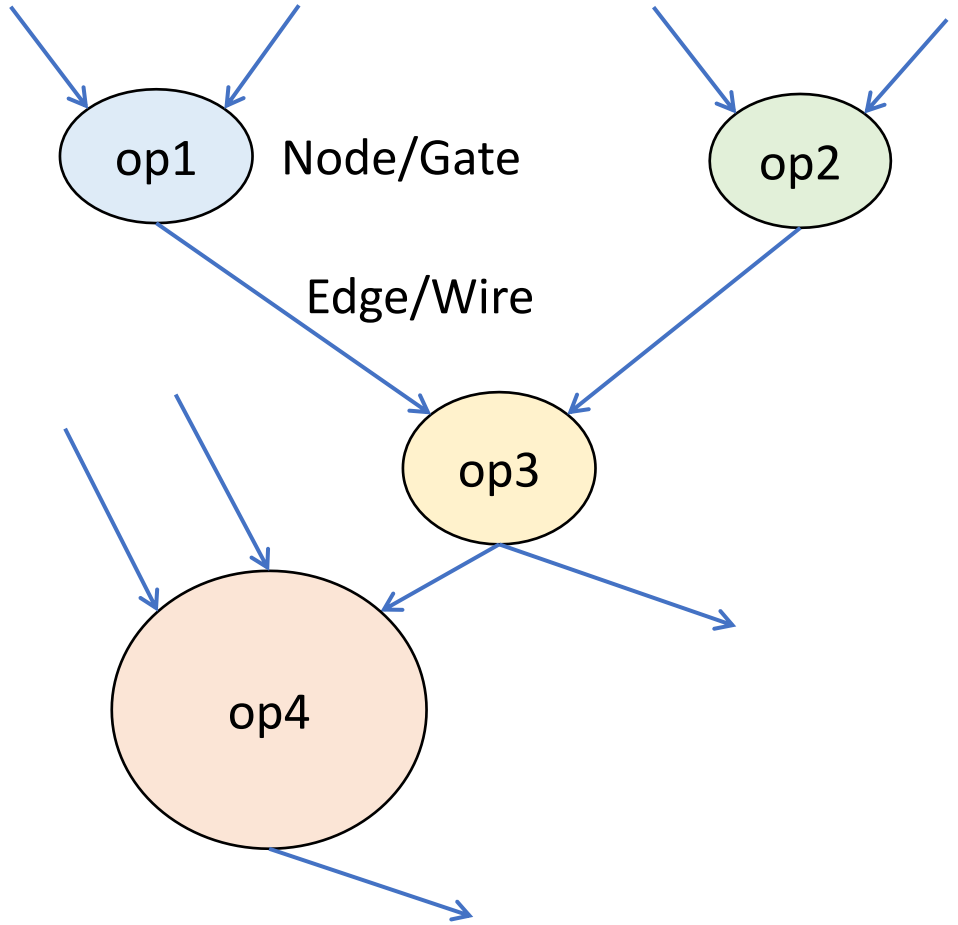
```
a ← load (addr1);
b ← load (addr2);
cmov a = (secret) ? a : b;
```



Programming in Circuit Abstraction

- Program = DAG (“circuit”)
- Operations = nodes (“gates”)
- Data transfers = edges (“wires”)

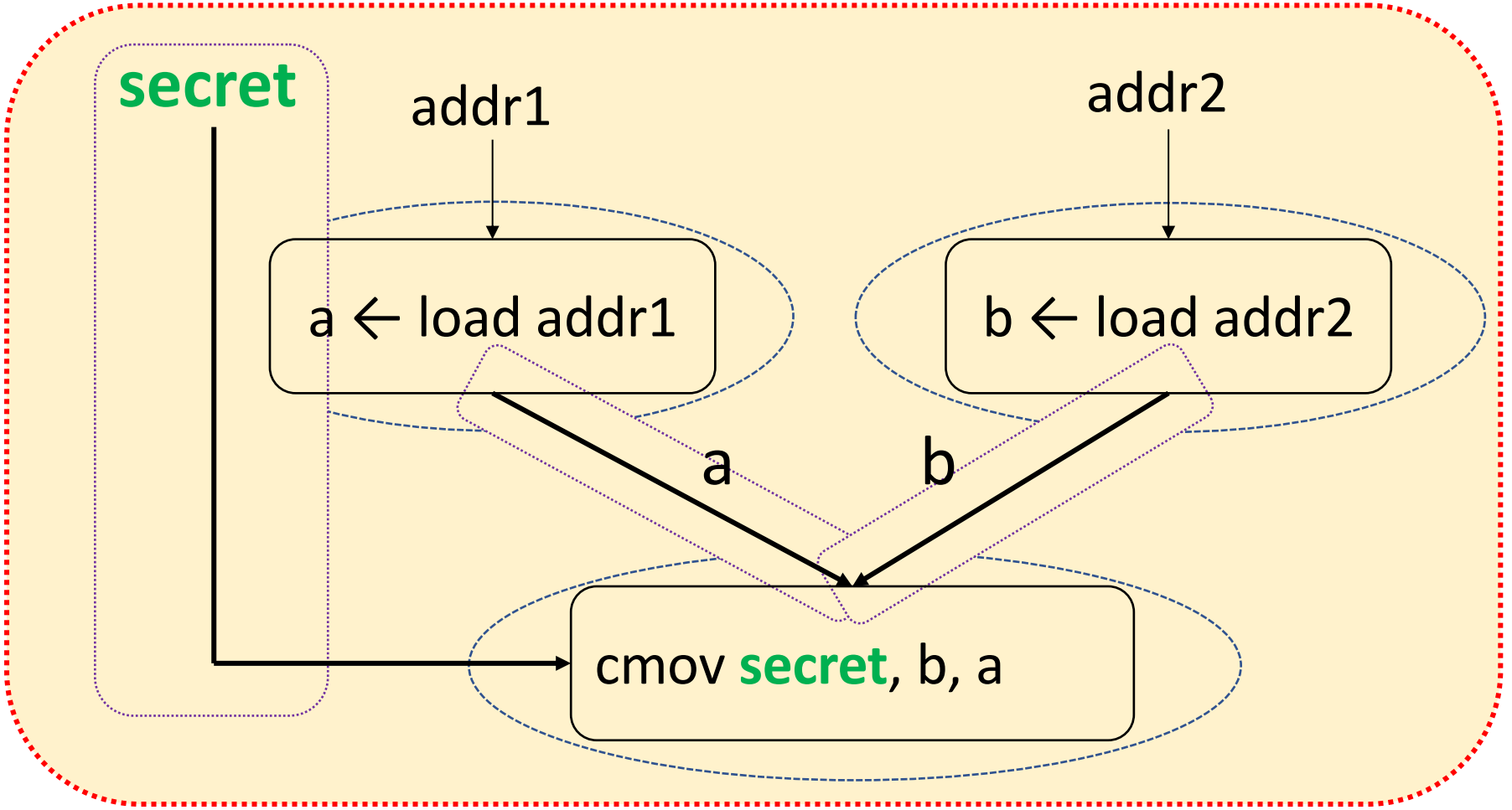
- Topology must be confidential data-**independent**
- Each gate’s execution must hide its inputs
- Each wire must hide the value it carries



What assumptions underpin the model?

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

secret = confidential
addr1 = public
addr2 = public



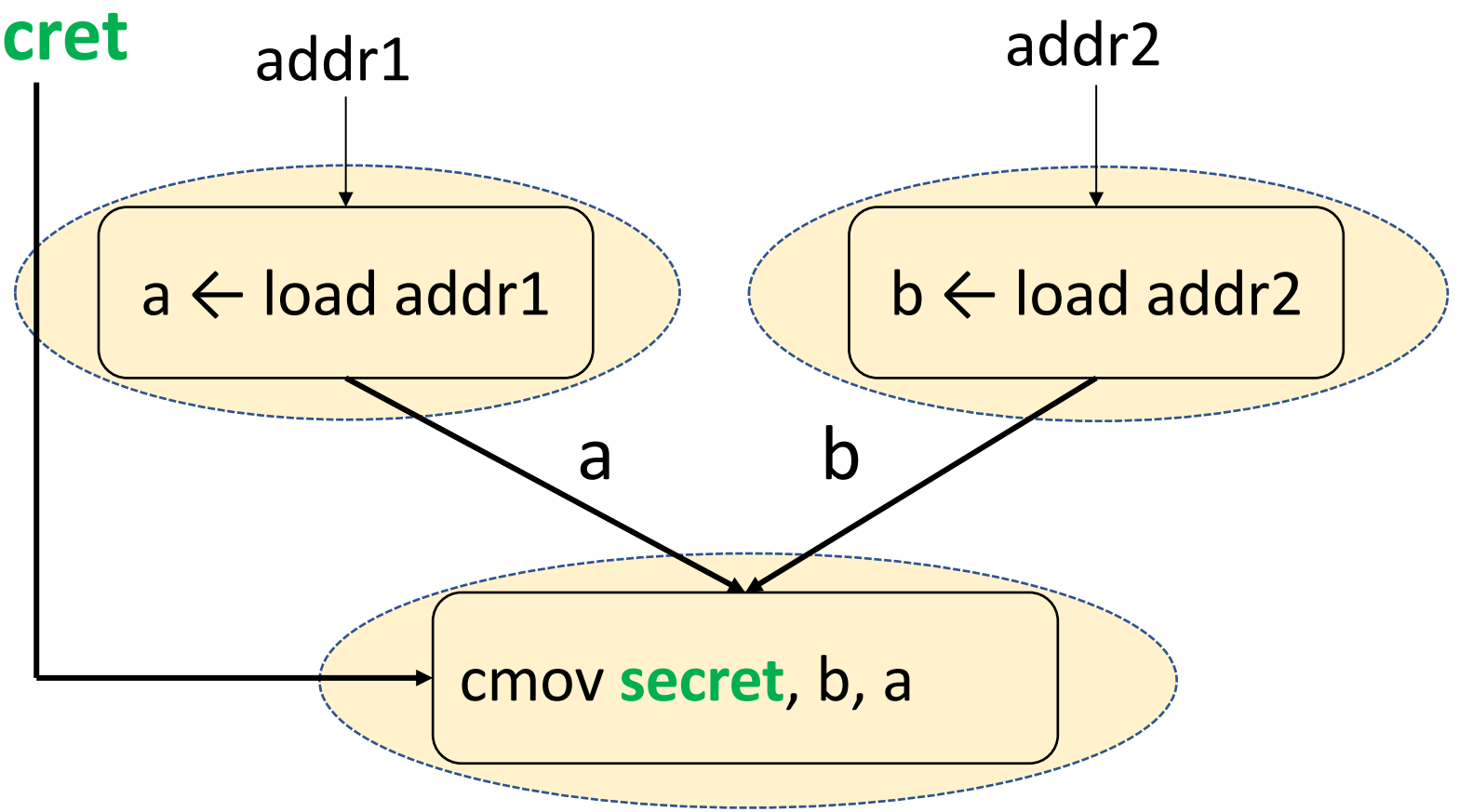
- **Rule 1:** instruction/gate execution = confidential data-independent
- **Rule 2:** data transfer/wire = confidential data-independent
- **Rule 3:** circuit/program topology = fixed

Today's machines can violate these assumptions

Violations due to:

Data-dependent instruction optimizations

(e.g., zero-skip, early exit, microcode, silent stores, …)



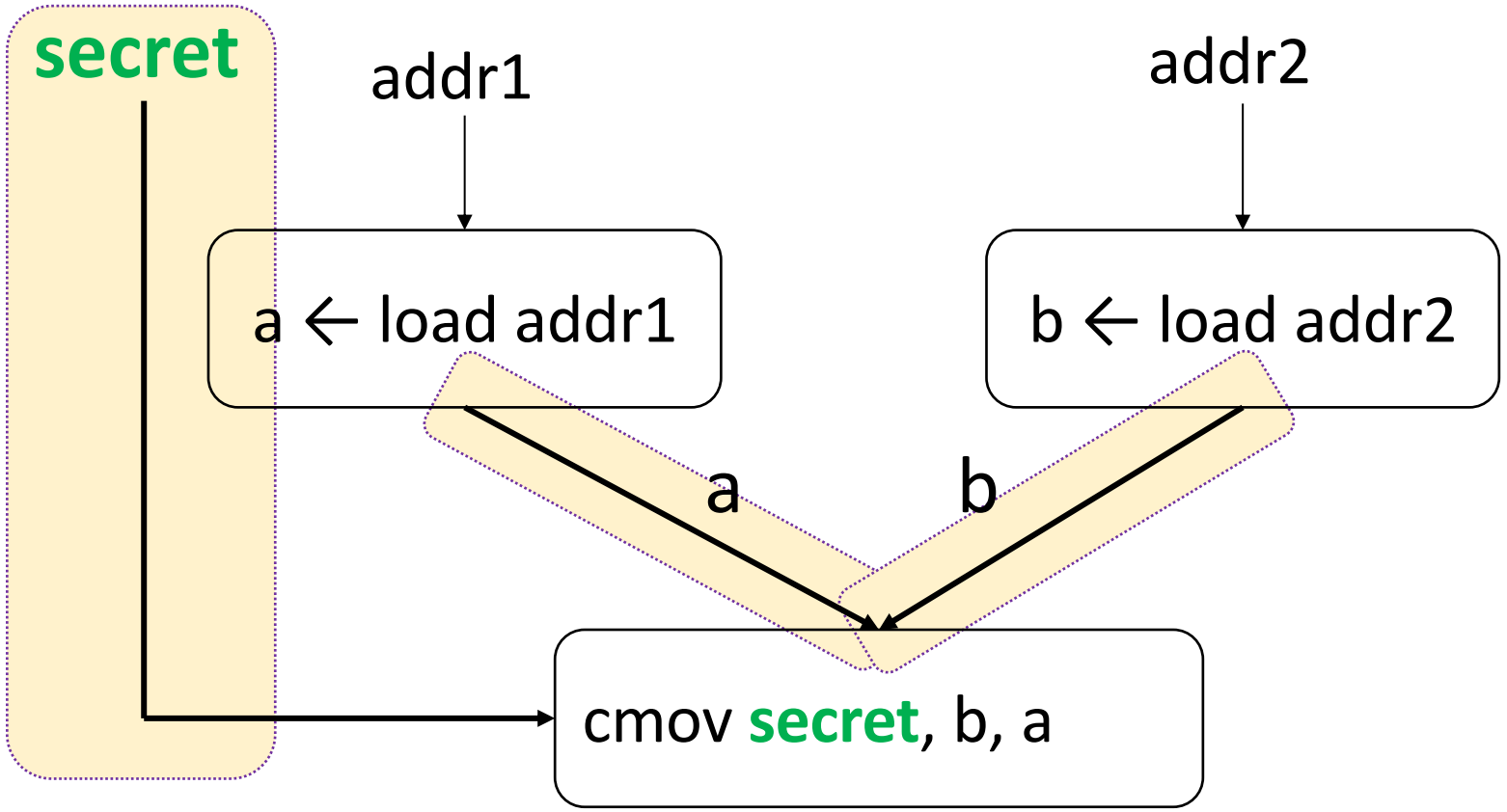
- **Rule 1:** instruction/gate execution = confidential data-independent
- **Rule 2:** data transfer/wire = confidential data-independent
- **Rule 3:** circuit/program topology = fixed

Today's machines can violate these assumptions

Violations due to:

Data at rest optimizations

(e.g., compression in register file/uop fusion, cache, page tables, …)

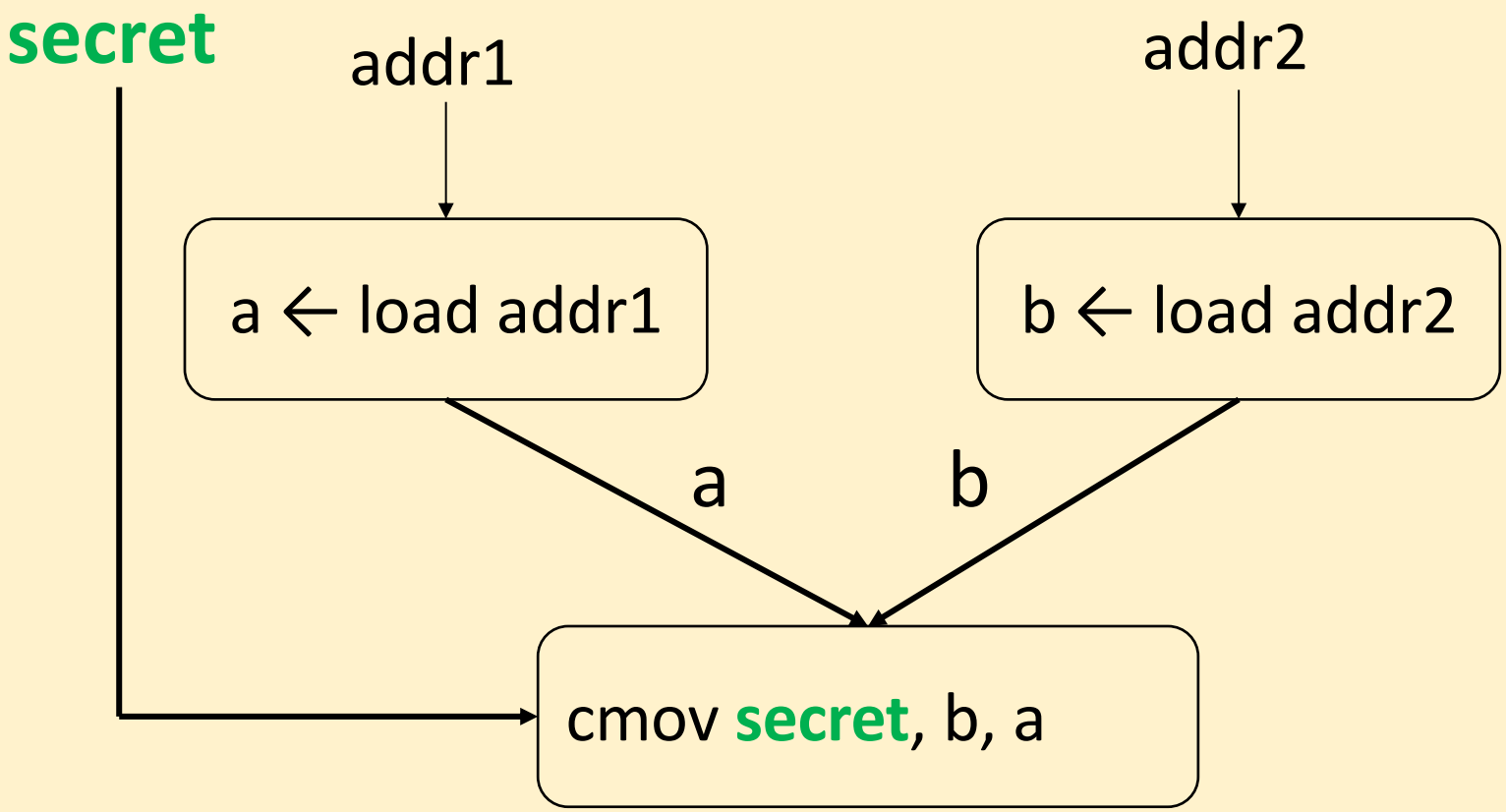


- **Rule 2:** data transfer/wire = confidential data-independent

Today's machines can violate these assumptions

Violations due to:

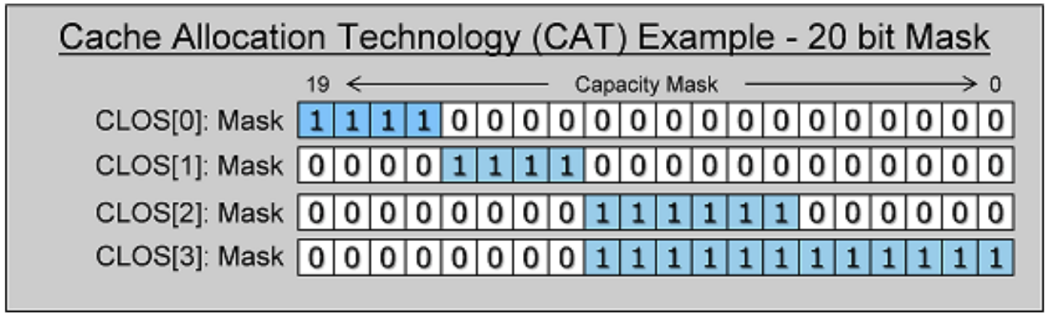
Speculative/OoO execution



- **Rule 1:** instruction/gate execution = confidential data-independent
- **Rule 2:** data transfer/wire = confidential data-independent
- **Rule 3:** circuit/program topology = fixed

HW Resource Partition

- Security v.s. Quality of Service (QoS)
  - Intel Cache Allocation Technology (CAT)
- Temporal Partition v.s. Spatial Partition



- 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?

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

- Randomize cache mapping functions
  - Pros and cons?

+ Generally low performance overhead (still allow cache to be shared)
- Difficult to reason about security
+/- Can reduce attack bandwidth, but unlikely to eliminate attacks

Next Lecture:
Transient Side Channels