Micro-Architectural Attacks

Sumanta Chaudhuri

(04 Jan, 2021)

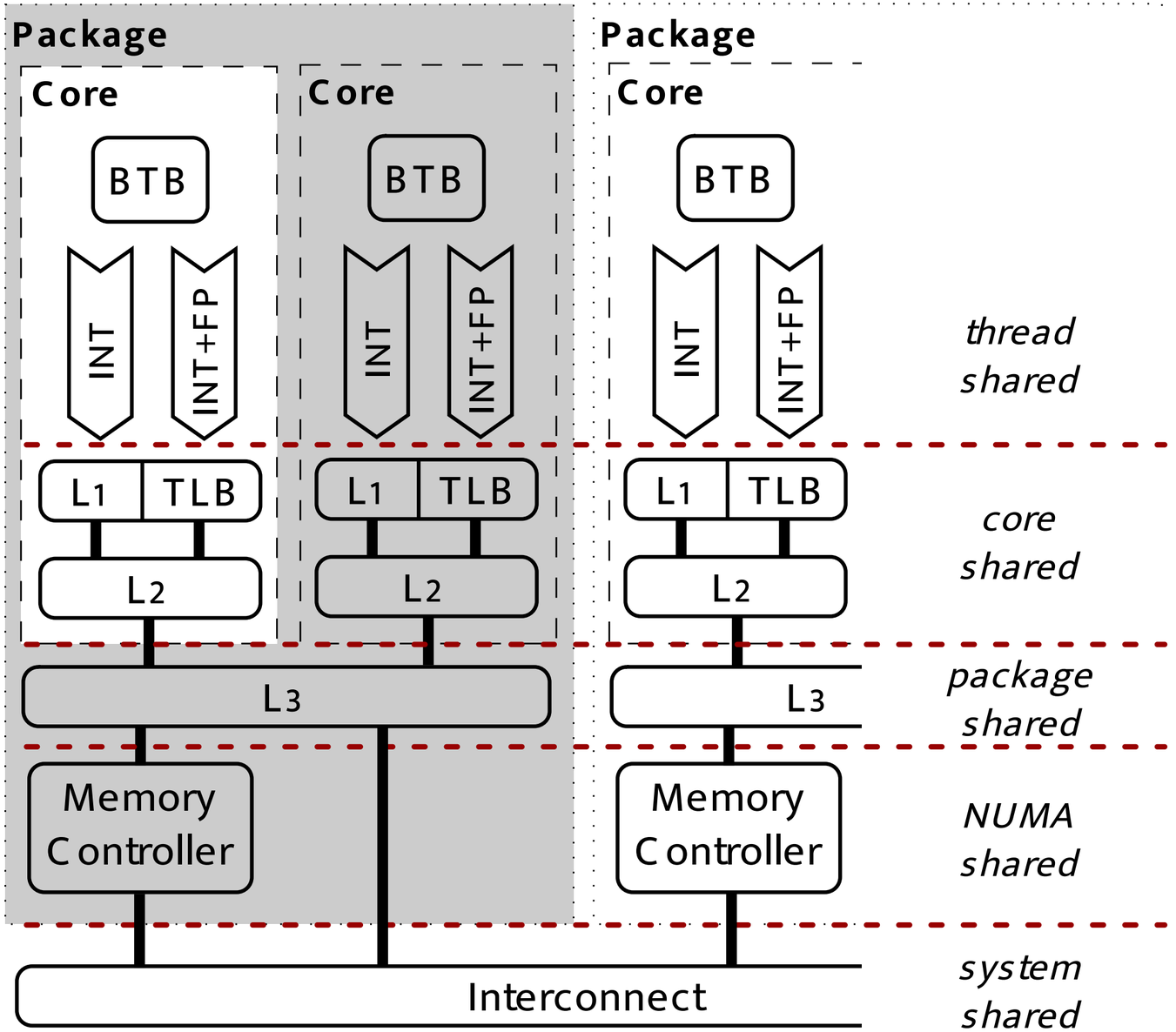
Micro-Architectural Attacks

- Information Leakage
  - Side Channels: Unintentional leakage of sensitive data
  - Covert Channels: Deliberate leakage of sensitive data (by a Trojan)
- Denial of Service
- Reverse Engineering.

Information Leakage

- Types of Side Channel in the micro-architectural Context
  - Storage Side-Channels: e.g unprotected memory locations.
  - Timing Side channels: e.g Information contained in cache hit/miss time difference.

SCA Classifications



Recap: Computer Architecture

Recap: Computer Architecture

- Processor

Recap: Computer Architecture

- Processor
- MMU

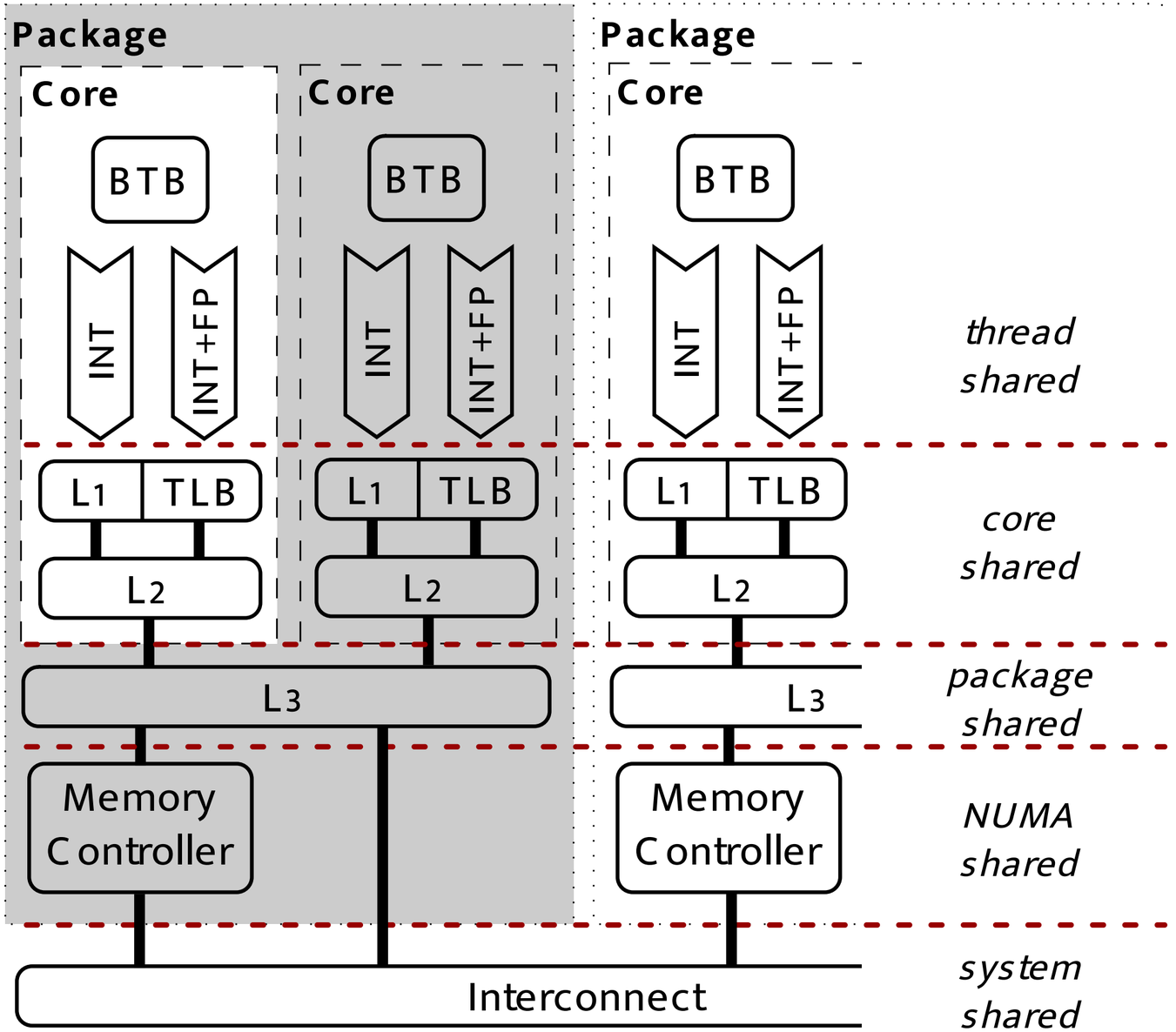
Recap: Computer Architecture

- Processor
- MMU
- Cache

Recap: Computer Architecture

- Processor
- MMU
- Cache
- Main Memory (DDR)

SCA Classifications



Recap: Processors

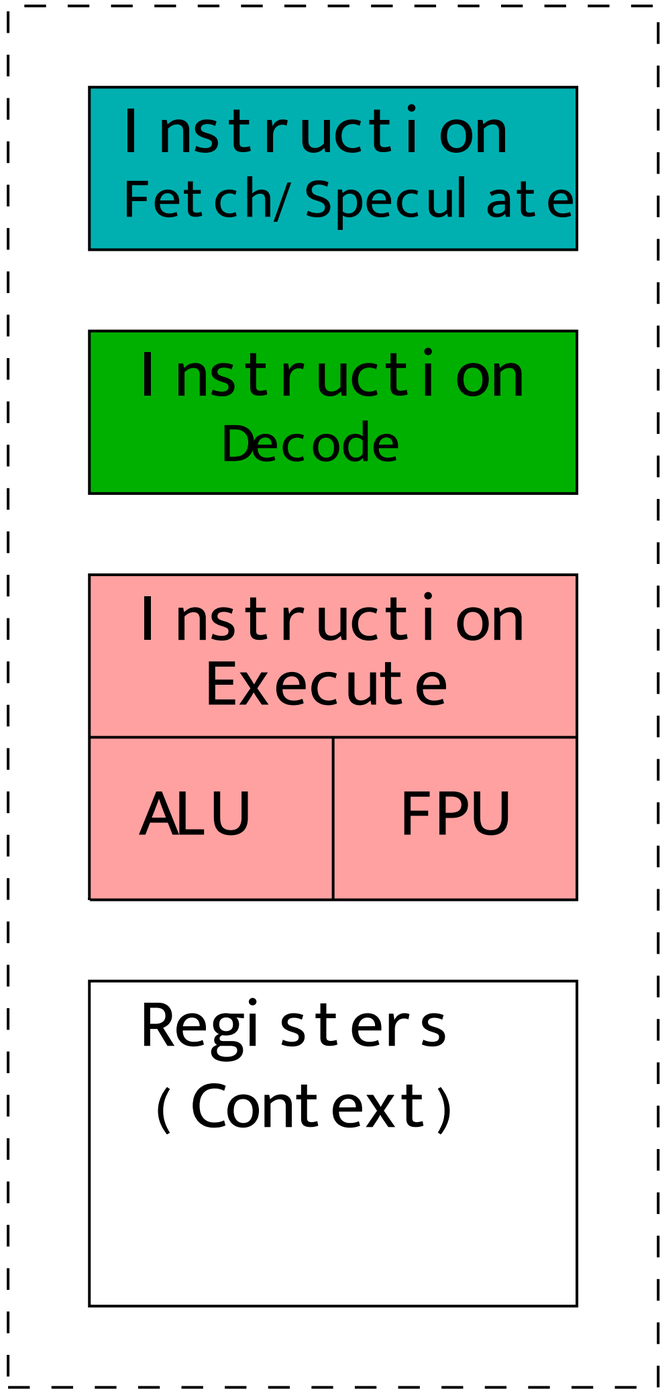
	Single Data	Multiple Data
Single Instr.	SISD	SIMD
Multiple Instr.	MISD	MIMD

	Single Data	Multiple Data
Single Instr.	CPU e.g 8086	SIMD
Multiple Instr.	MISD	MIMD

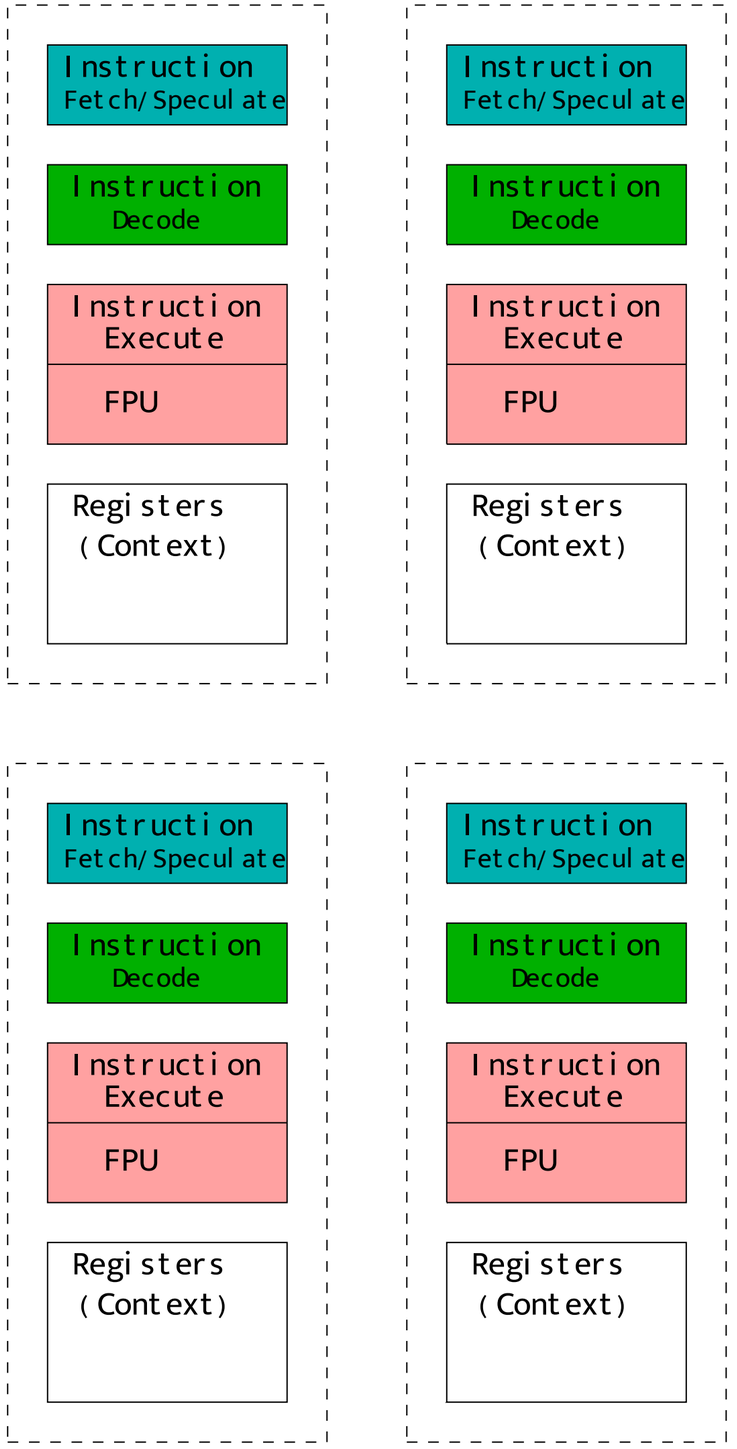
	Single Data	Multiple Data
Single Instr.	CPU e.g 8086	VPU, GPU e.g CRAY NVIDIA
Multiple Instr.	MISD	MIMD

	Single Data	Multiple Data
Single Instr.	CPU e.g 8086	VPU, GPU e.g CRAY NVIDIA
Multiple Instr.	MISD	Multicore e.g Intel i7

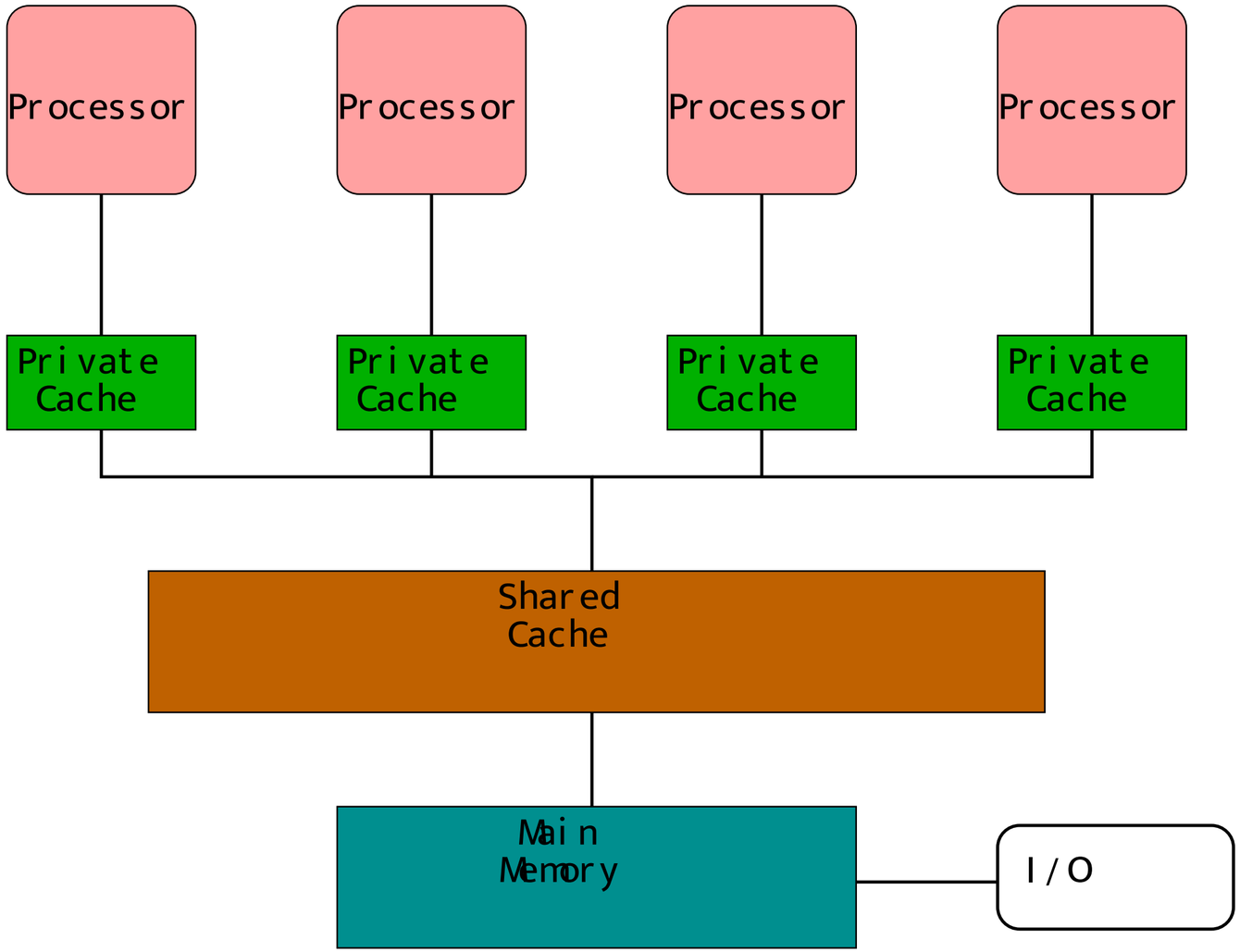
Uniprocessor



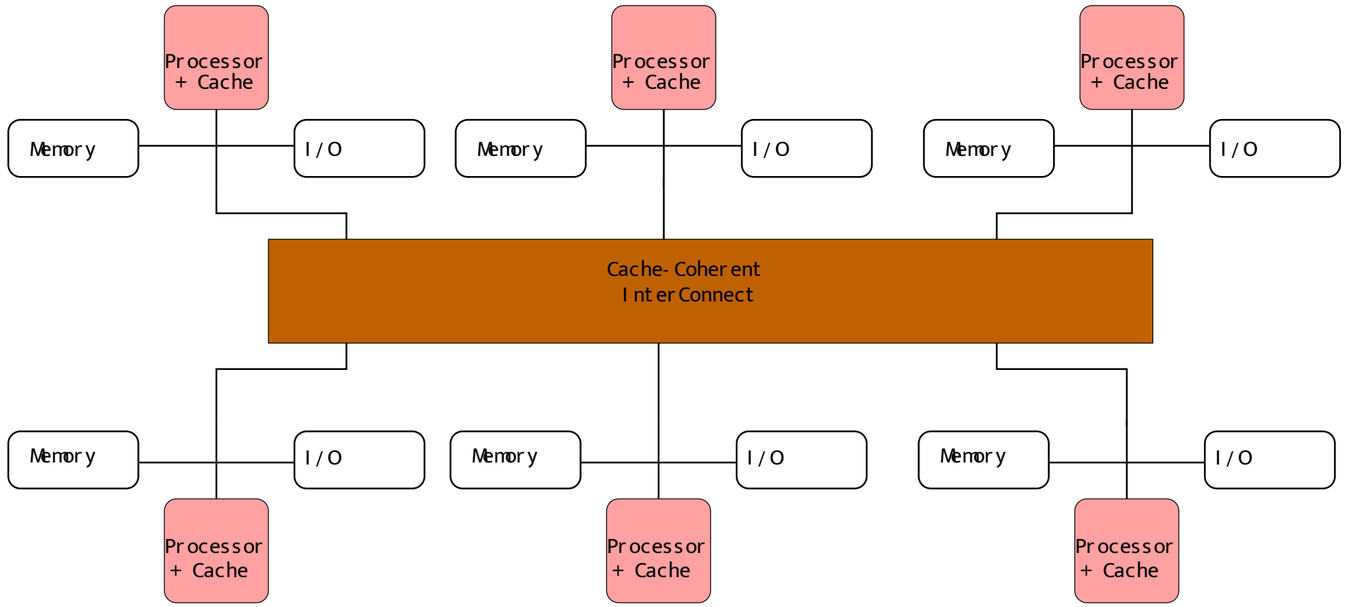
Multiprocessor



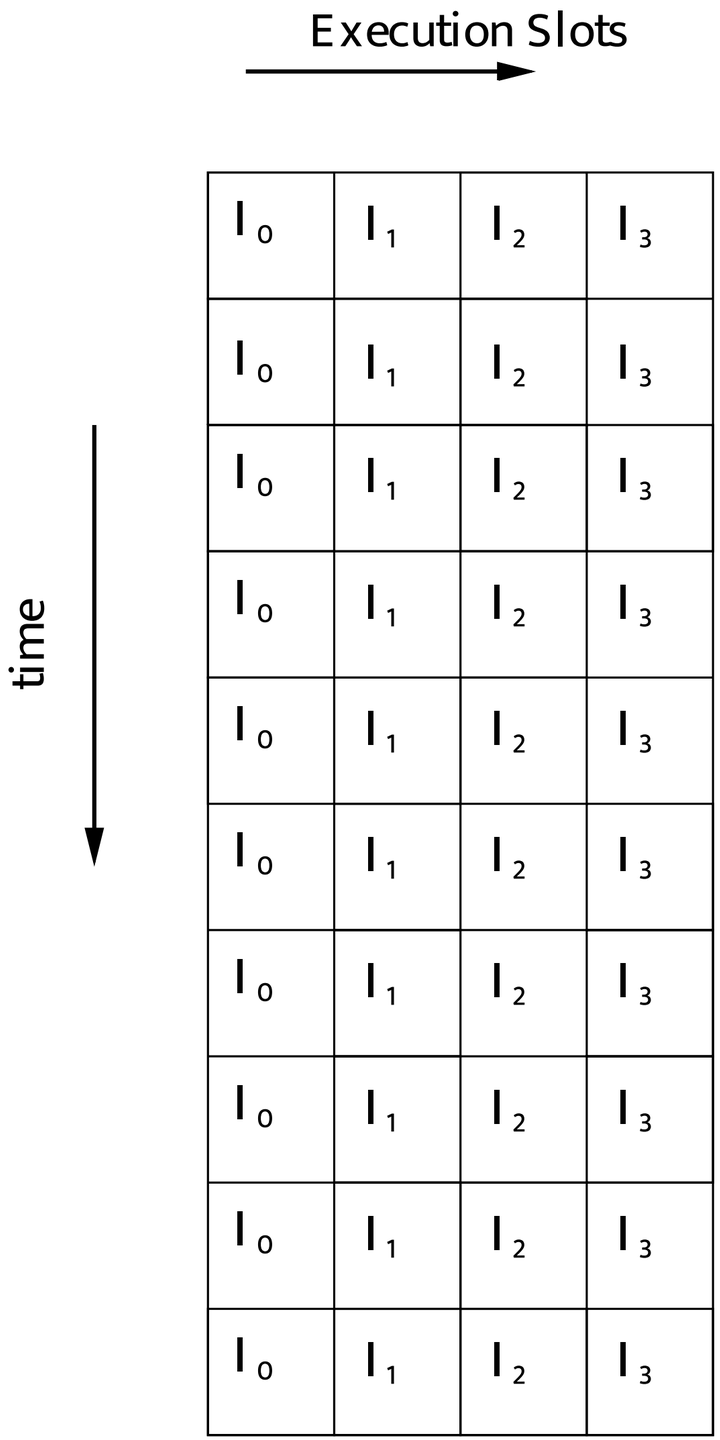
Multiprocessor: UMA



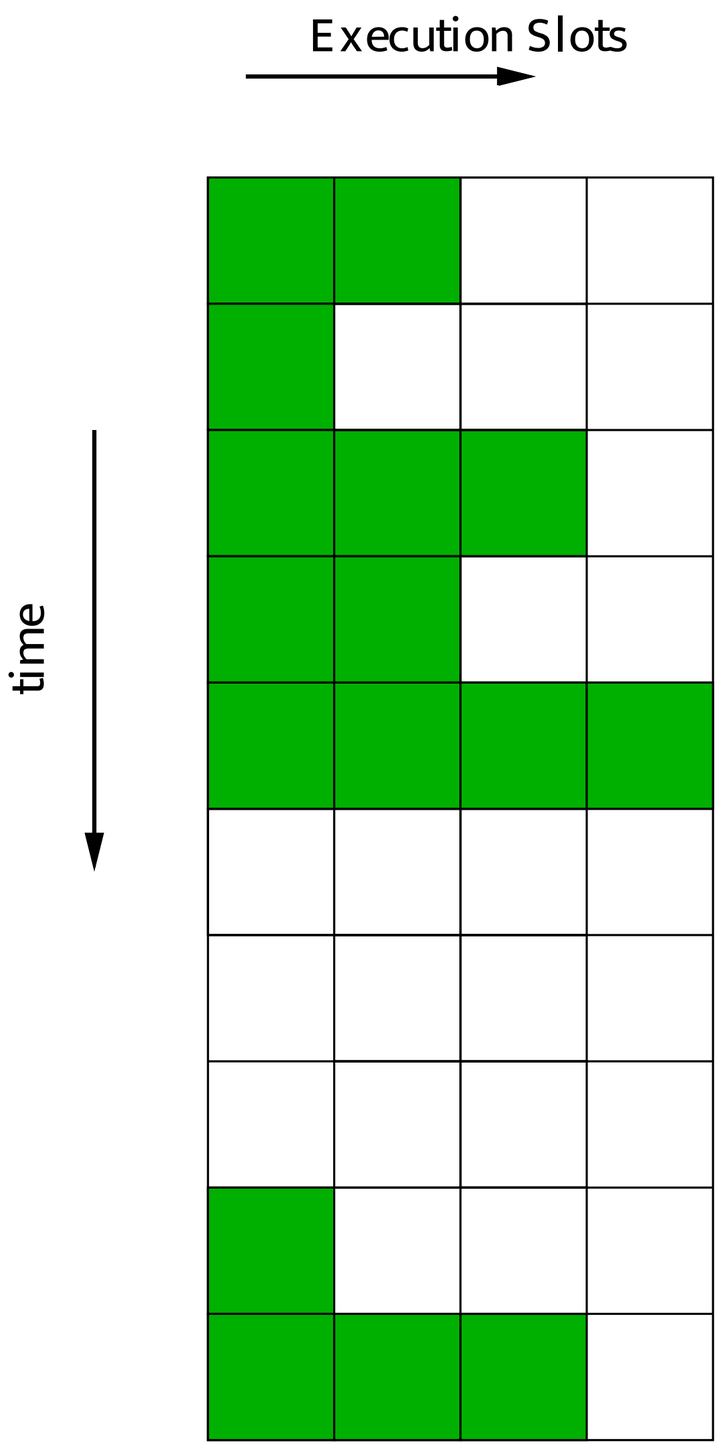
Multiprocessor: NUMA



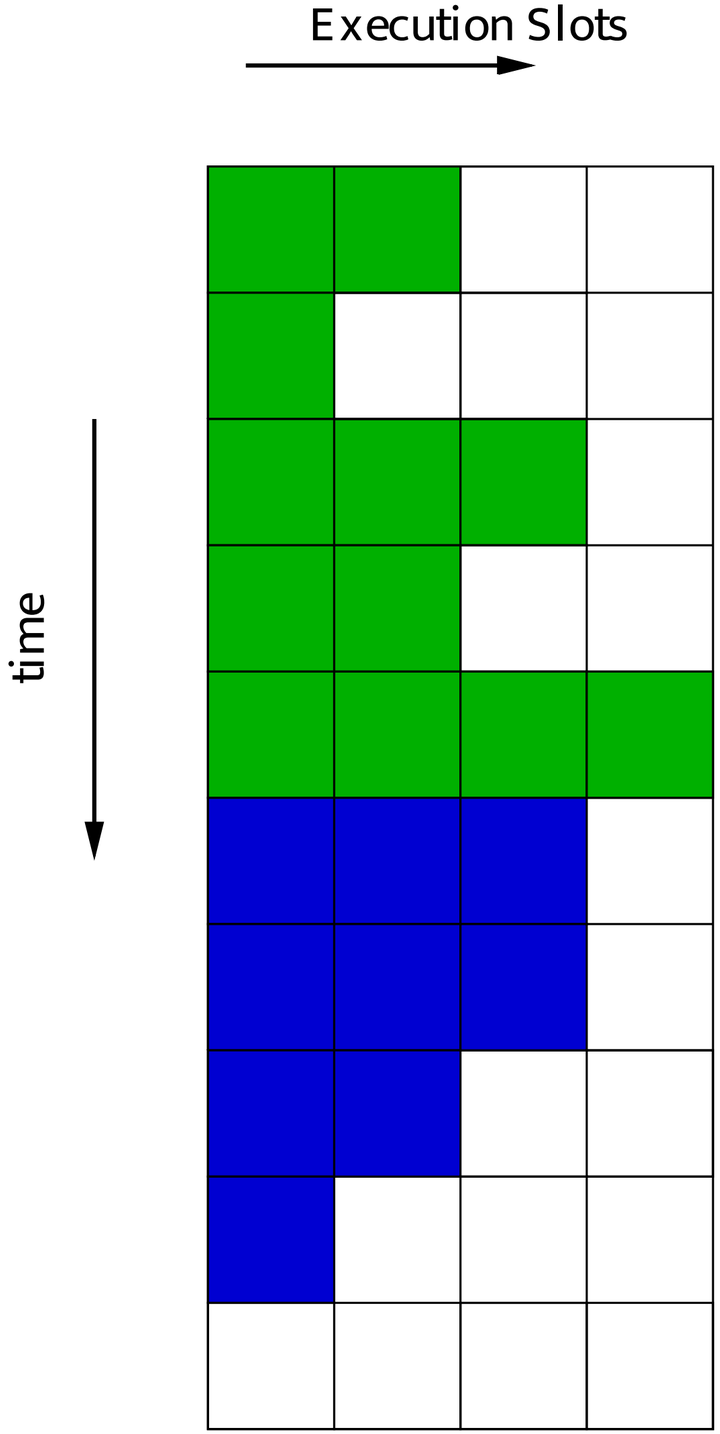
Processor: VLIW



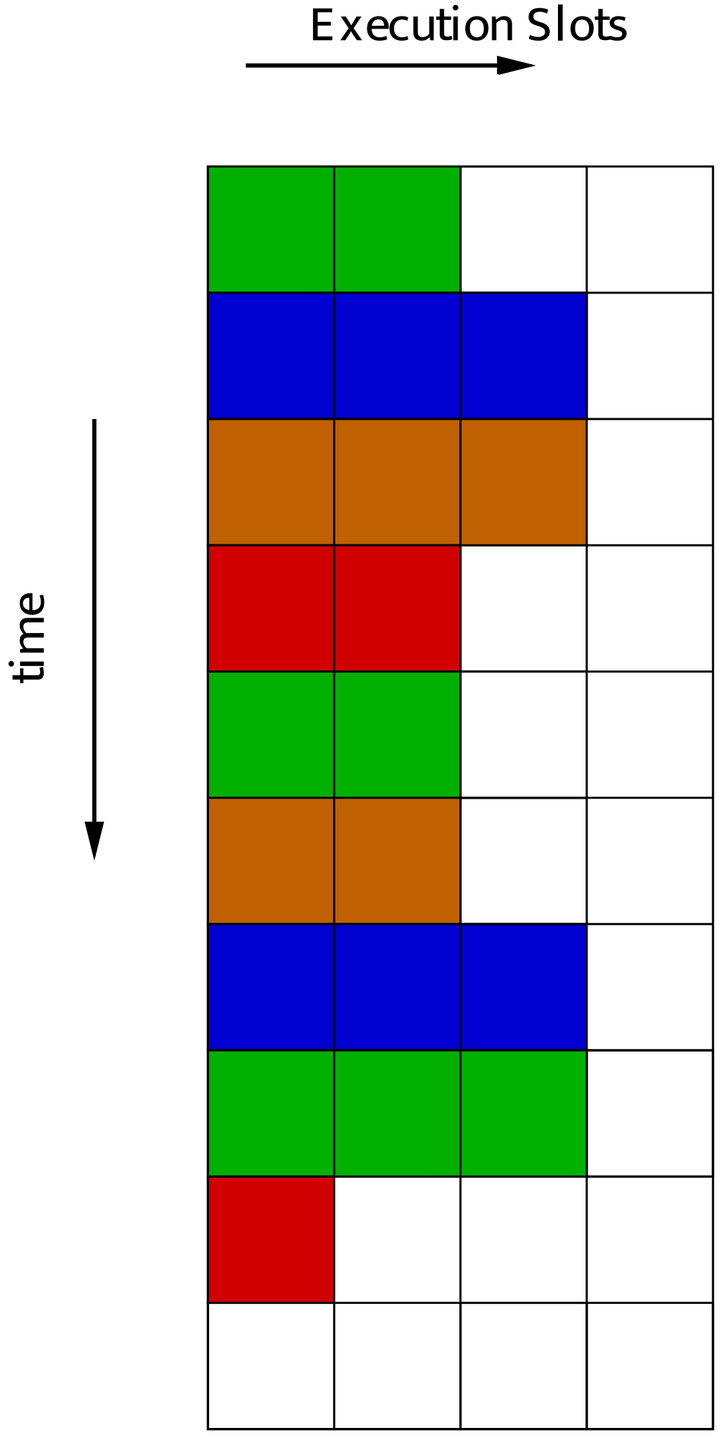
Multi-Threading: Superscalar



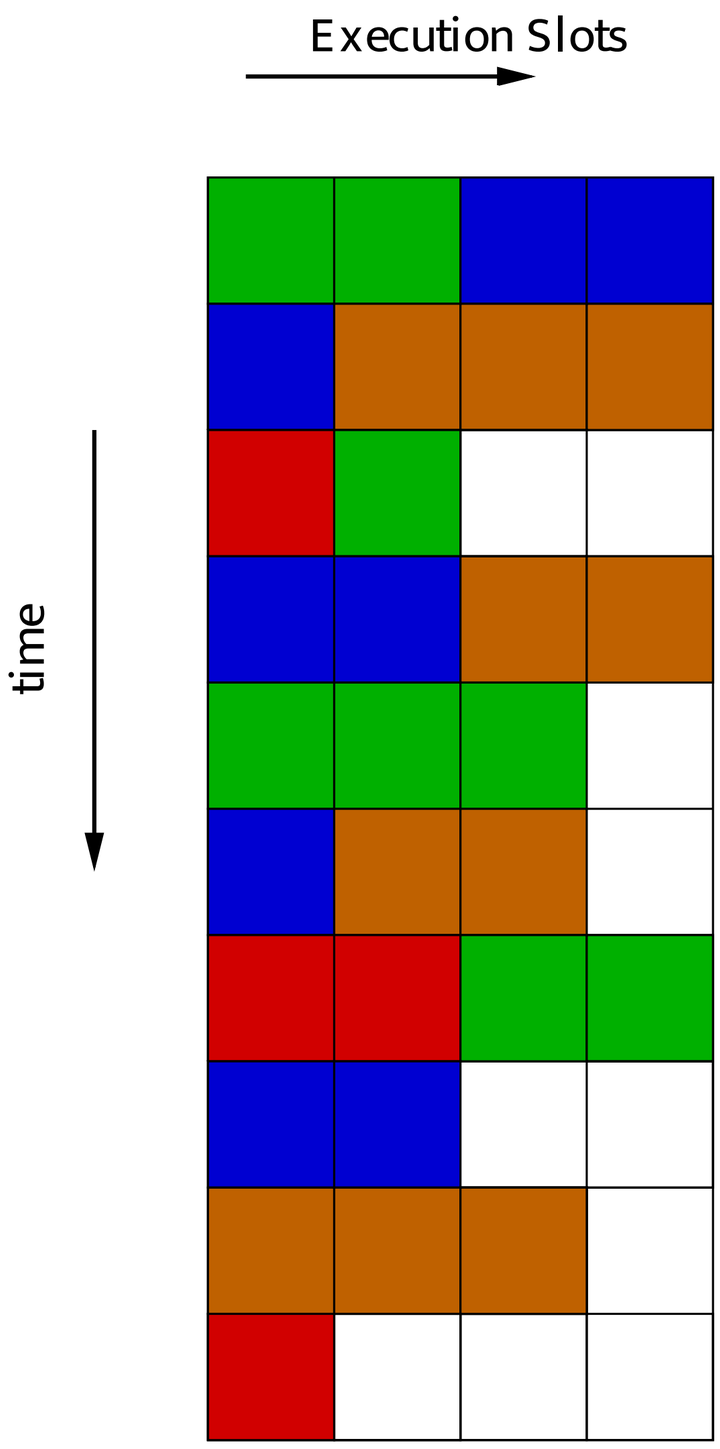
Multi-Threading: Coarse-Grained



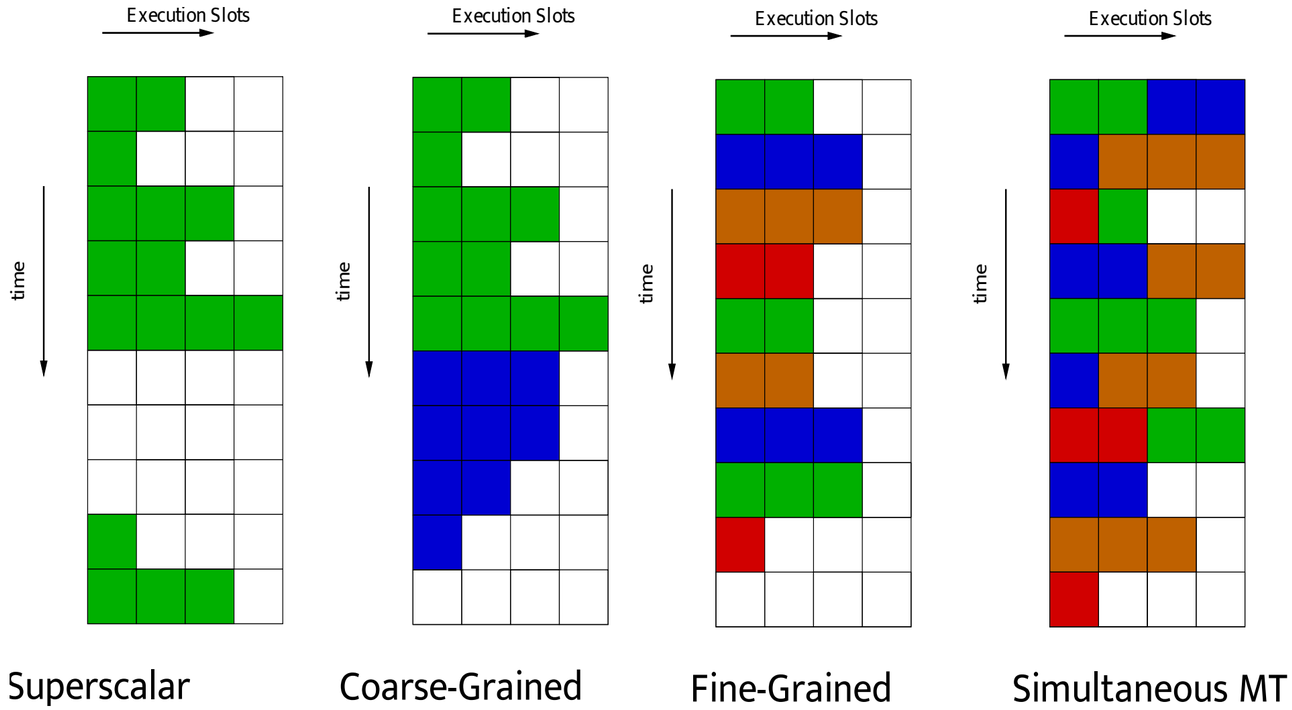
Multi-Threading: Fine-Grained



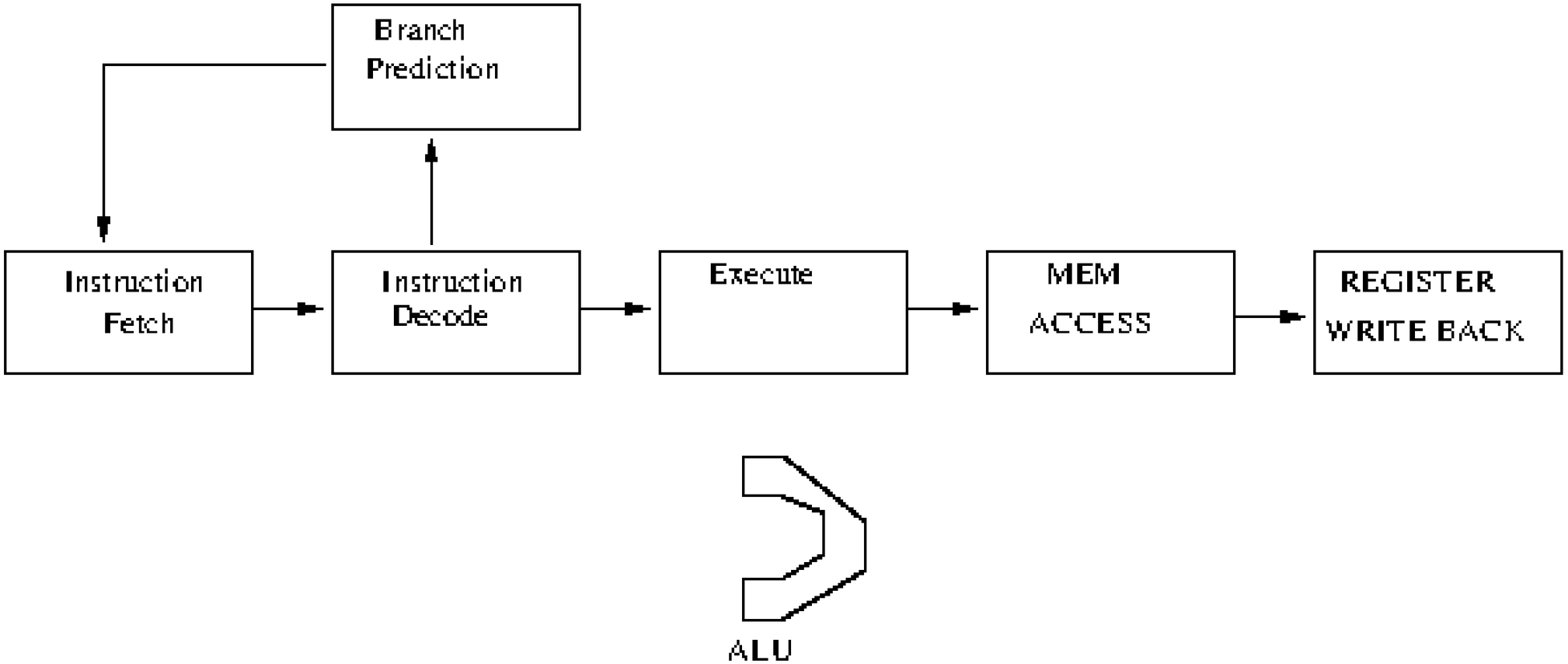
Multi-Threading: Simultaneous



Reacp: Multi-Threading



In Order Processor Pipeline



In Order Processor Pipeline

- Statically Scheduled (During Compile Time)
- Pipeline Hazards
  - Structural Hazards: e.g two back-to-back instructions using the floaitng point unit.
  - Data Hazards: e.g an instruction depends on the result of the previous instruction.
  - Control Hazards: e.g branhces , future value of PC is not known.

In Order Processor Pipeline

- Data hazards stall the pipeline.
- Need Branch predictors for control hazards:
  - Simple 1 bit/ 2 bit predictors. ([1], p.C-24)
  - Correlating Branch Predictor. ([1] p. 182)
  - Tournament Predictors. ([1] p. 184)
  - Tagged Hybrid Predcitors. ([1] p.188)

-Dynamic Scheduling and Out-of-Order execution [1] p.193

```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

-Dynamic Scheduling and Out-of-Order execution [1] p.193

Dependence

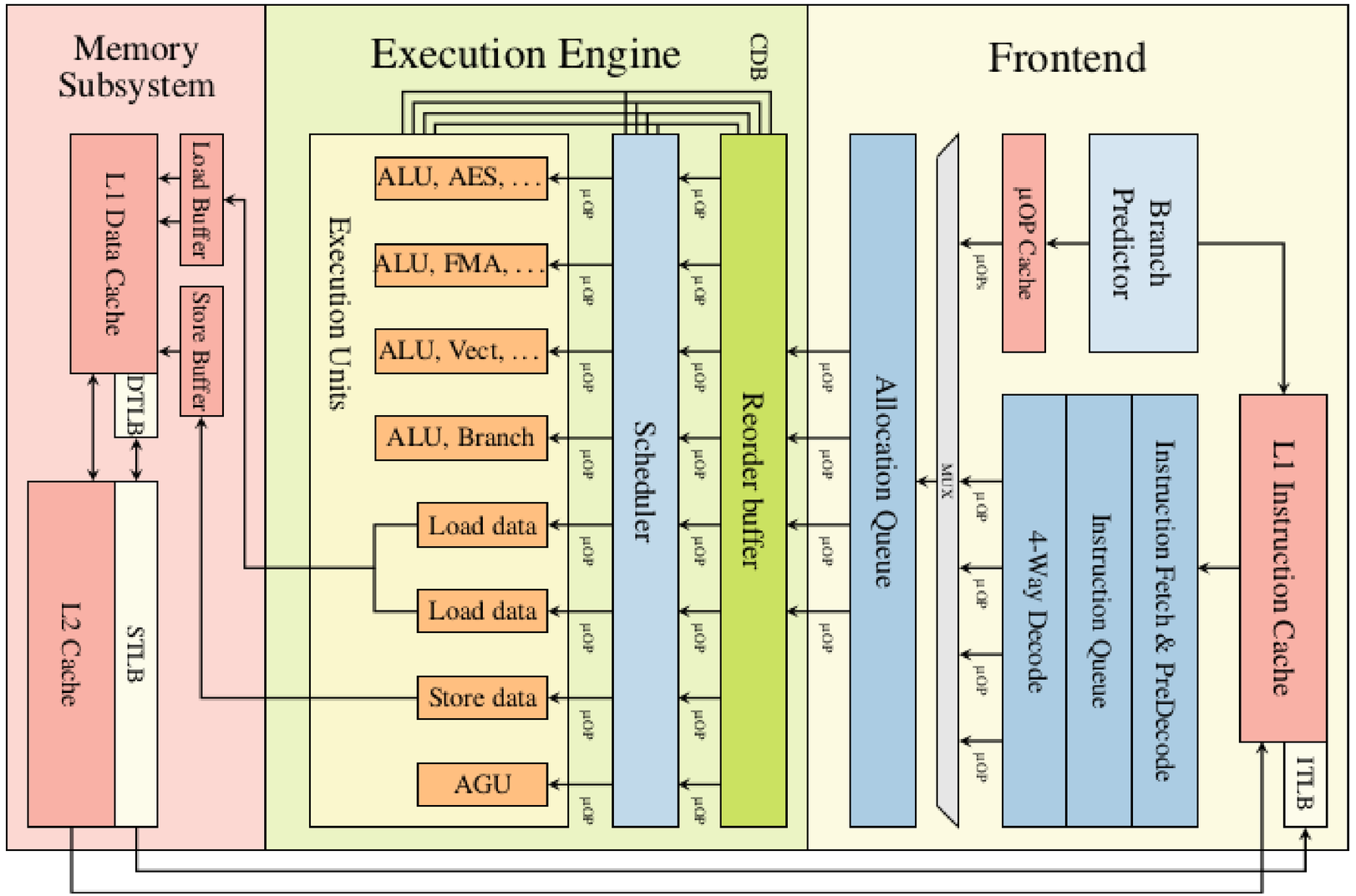
```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

-Dynamic Scheduling and Out-of-Order execution [1] p.193

Stalled

```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

Out-of-Order Pipeline



Out-of-Order Pipeline

- Multiple Insturction are fetched in parallel.
- Execute Instructions that are ready (I.e data available)
- Instructions are commited in-order using the reorder buffer

Out-of-Order Pipeline

- Hides Latency (Like Cache, & Multiple threads)
- Much More complex
- Security Hazards (As we will see later)
- Can not be done in compiler as compiler does not have runtime data.
  - e.g dynamic Scheduling

Example: Branch Target Buffer Side Channel (Ref [2])

- BTB stores the target addresses of previous branches.
- Acts like a cache.

Montgomery Multiplier BTB Attack

```
function exponent(b, e, m)
begin
    x ← 1
    for i ← |e| -1 downto 0 do
        x ← x^2
        x ← x mod m
        if (ei = 1) then
            x ← xb
            x ← x mod m
         endif
     done
return x
end
```

Montgomery Multiplier BTB Attack

```
function exponent(b, e, m)
begin
    x ← 1
    for i ← |e| -1 downto 0 do
        x ← x^2
        x ← x mod m
        if (ei = 1) then
            x ← xb
            x ← x mod m
        endif
    done
return x
end
```

branch not taken

Montgomery Multiplier BTB Attack

```
function exponent(b, e, m)
begin
    x ← 1
    for i ← |e| -1 downto 0 do
        x ← x^2
        x ← x mod m
        if (ei = 1) then
            x ← xb
            x ← x mod m
         endif
    done
return x
end
```

branch taken

Example: Branch Target Buffer Side Channel (Ref [2])

Example: Branch Target Buffer Side Channel (Ref [2])

- assume that an adversary can run a spy process simultaneously with the cipher

Example: Branch Target Buffer Side Channel (Ref [2])

- assume that an adversary can run a spy process simultaneously with the cipher
- spy process continuously executes unconditional branches

Example: Branch Target Buffer Side Channel (Ref [2])

- assume that an adversary can run a spy process simultaneously with the cipher
- spy process continuously executes unconditional branches
- these branches map to the same BTB set with the conditional branch under attack.

Example: Branch Target Buffer Side Channel (Ref [2])

Example: Branch Target Buffer Side Channel (Ref [2])

- The adversary starts the spy process before the cipher

Example: Branch Target Buffer Side Channel (Ref [2])

- The adversary starts the spy process before the cipher
- Cipher cannot find the target address of the target branch in BTB -> misprediction

Example: Branch Target Buffer Side Channel (Ref [2])

- The adversary starts the spy process before the cipher
- Cipher cannot find the target address of the target branch in BTB -> misprediction
- misprediction -> the target address of the branch needs to be stored in BTB.

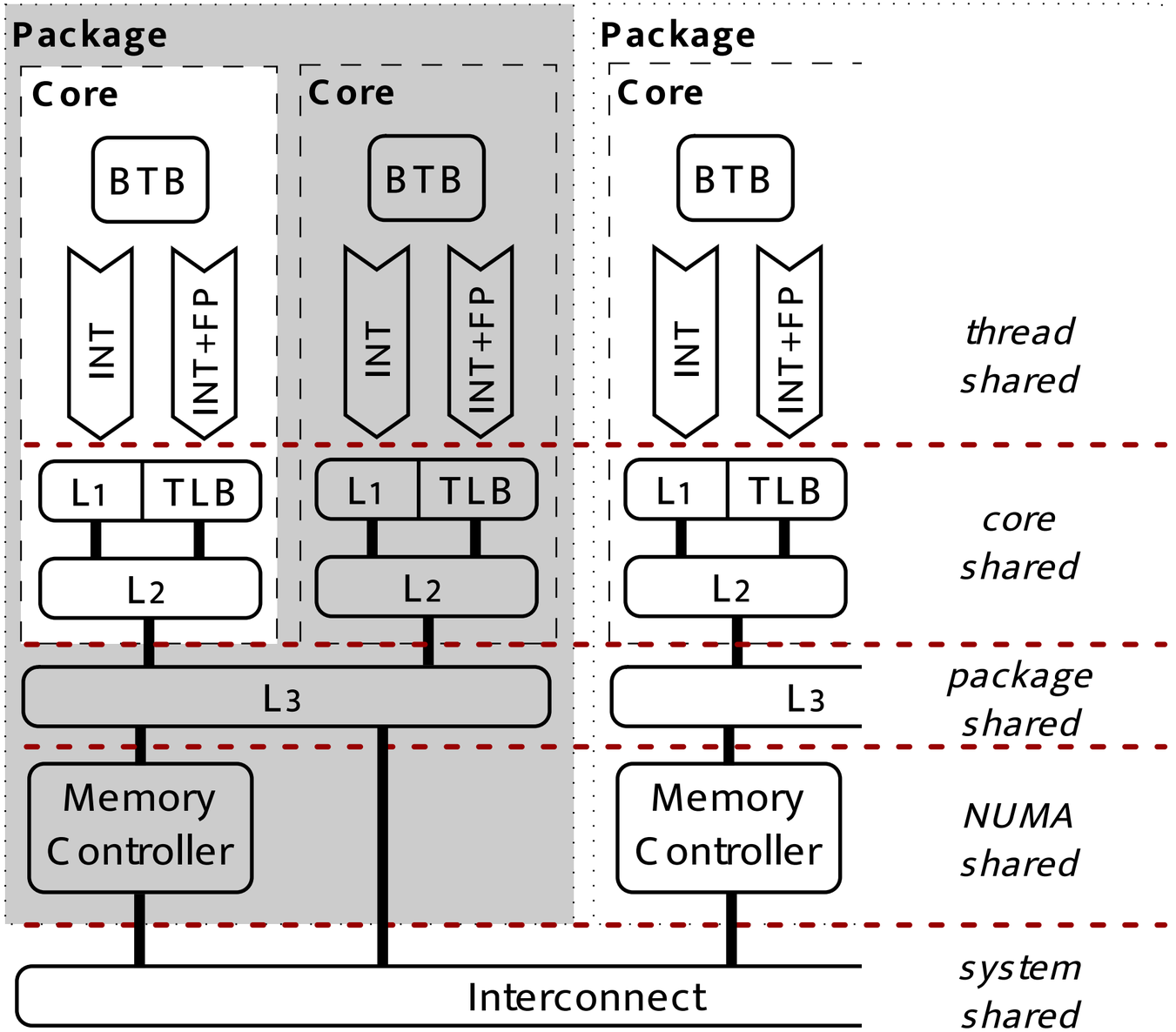
Example: Branch Target Buffer Side Channel (Ref [2])

- The adversary starts the spy process before the cipher
- Cipher cannot find the target address of the target branch in BTB -> misprediction
- misprediction -> the target address of the branch needs to be stored in BTB.
- spy branch is evicted. (they occupy the whole BTB set)

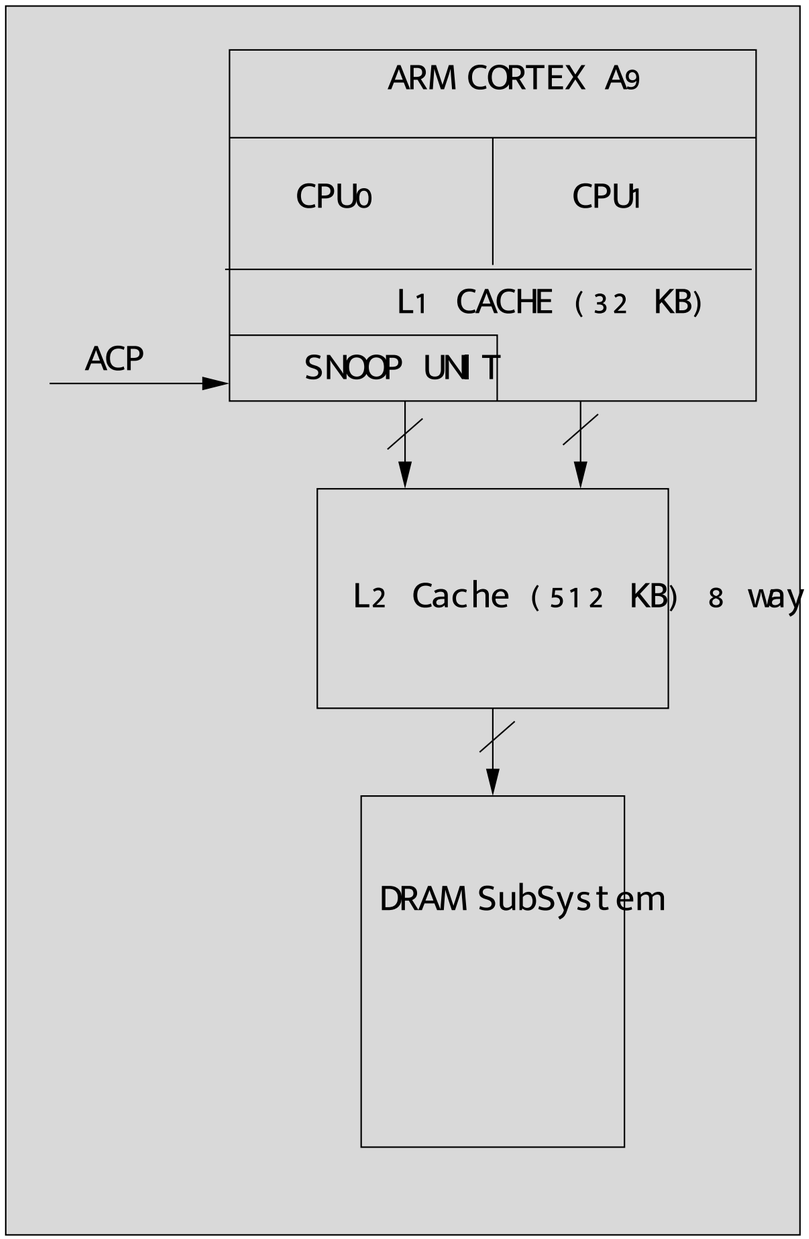
Example: Branch Target Buffer Side Channel (Ref [2])

- The adversary starts the spy process before the cipher
- Cipher cannot find the target address of the target branch in BTB -> misprediction
- misprediction -> the target address of the branch needs to be stored in BTB.
- spy branch is evicted. (they occupy the whole BTB set)
- spy finds from its own execution time if the branch was taken.

SCA Classifications



RECAP: Cache


ARM CORTEX A9
CPU0
CPU1
L1 CACHE (32 KB)
ACP
SNOOP UNIT
L2 Cache (512 KB) 8 way
DRAM SubSystem

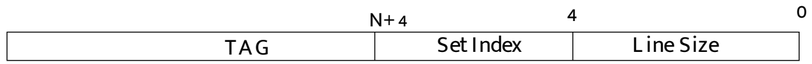
RECAP: Cache: Cache Terminology

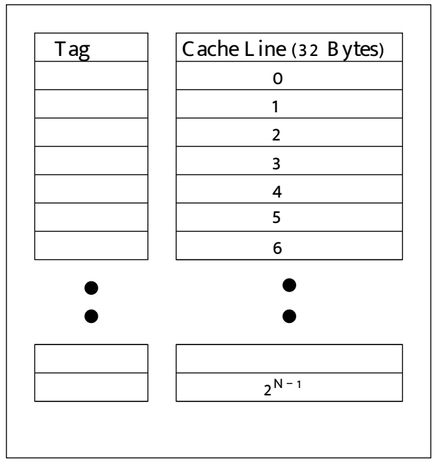
- Memory contains up-to-date data, and cache has a copy (cache line): CLEAN
- Cache has up-to-date data, and it must be written back to memory: DIRTY
- Memory contains up-to-date data, and cache does not : INVALID
- Memory does not have up-to-date data, cache does not : INVALID

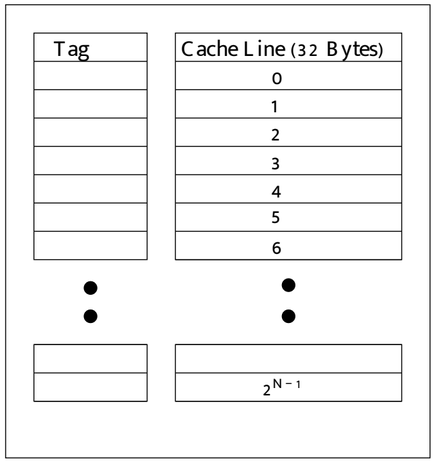
RECAP Cache Terminology

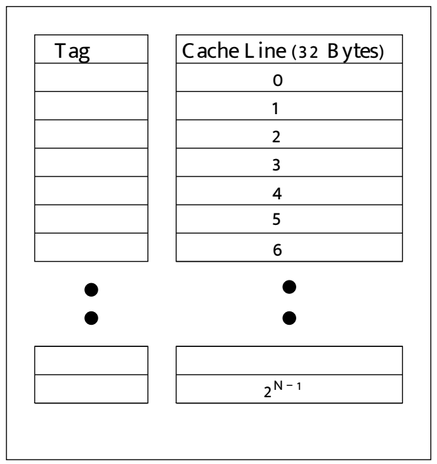
- HIT: Data found in Cache.
- MISS: Data is not in the cache.
- EVICT: A clean cache line is replaced due to a new allocation.

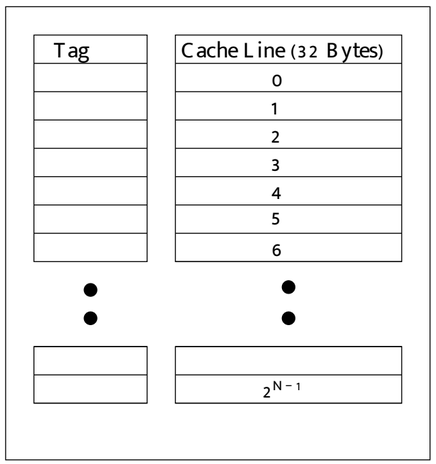
RECAP Cache Organization (4 Way)











RECAP Cache Policies

- Allocation
  - Write Allocate : On a Write miss replace the cache line.
  - Read Allocate : On a read miss replace the cache line.

RECAP Cache Policies

- Update
  - Write Through : A write updates both the cache and the main memory.
  - Write Back: Write updates the cache only (marked as dirty). Main memory is updated, when the line is evicted, cache is flushed.

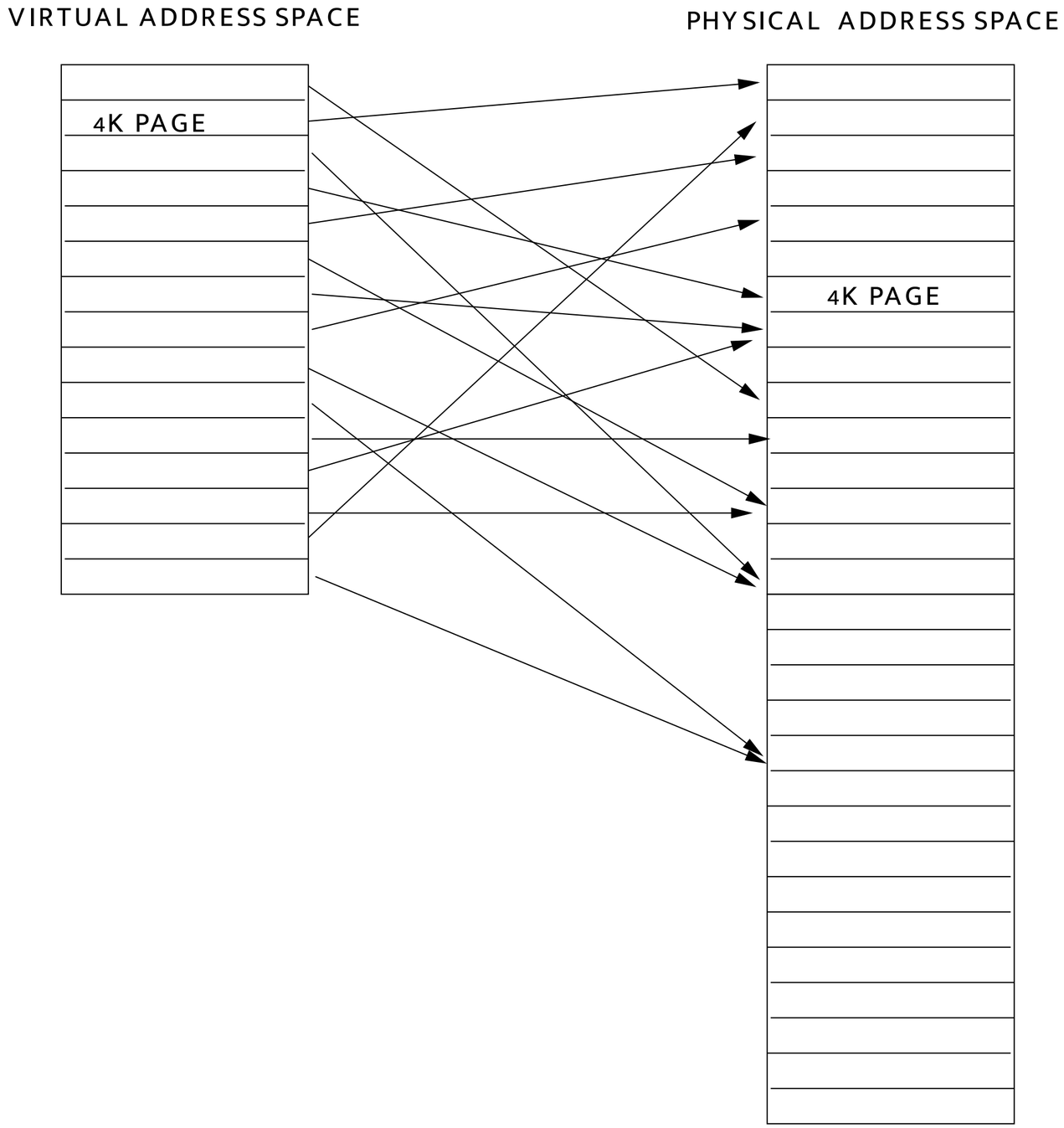
RECAP: Cache Coherence

- Case 1. Memory update by another master. Cached copy is out of date.
- Case 2. For write back cache, when master writes to cache, main memory is out of date.

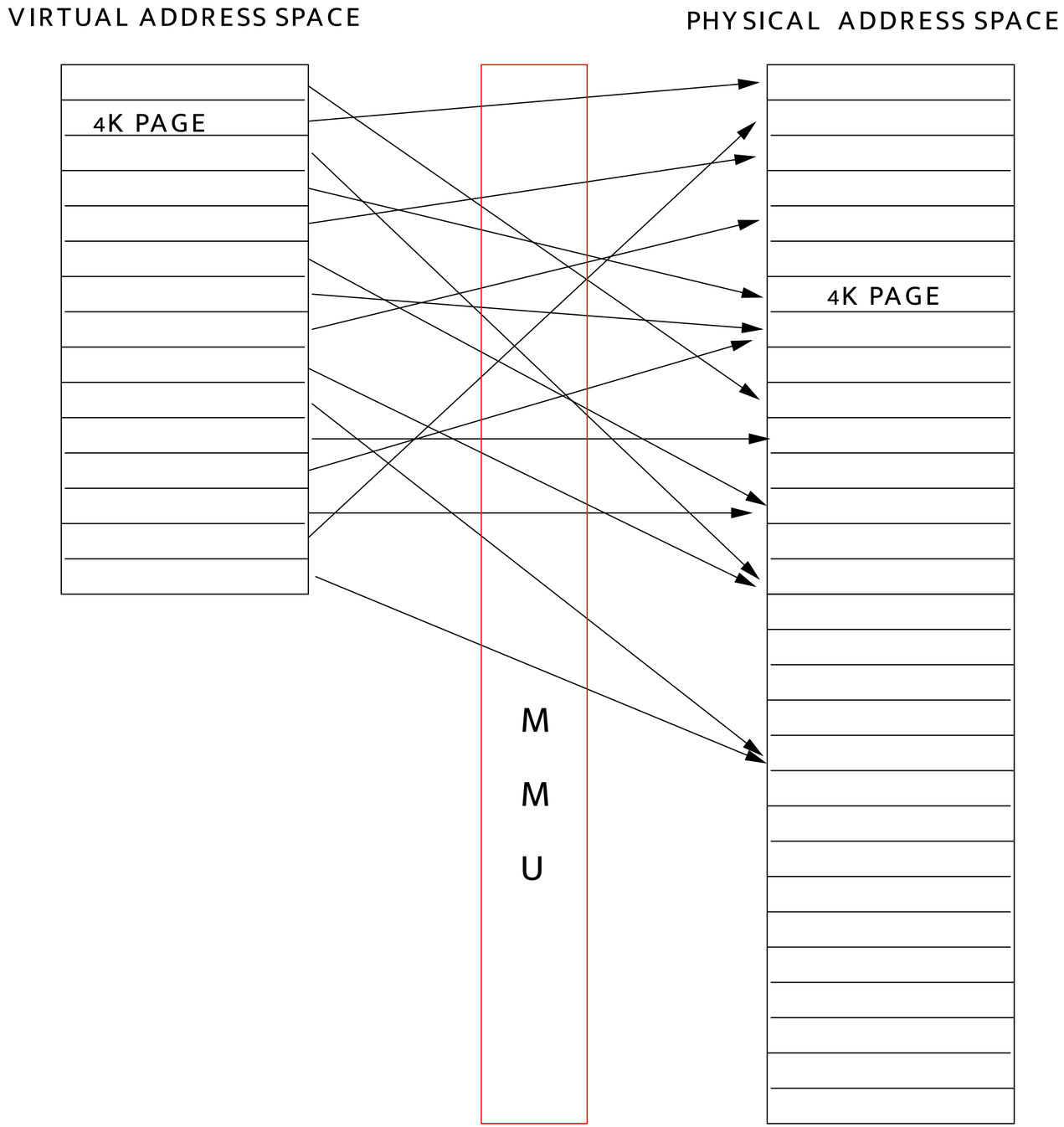
RECAP: Cache Coherence

- Cache Coherency Protocols
  - MEI (Modified, Exclusive, Invalid)
  - MESI (Modified, Exclusive, Shared Invalid)
  - MOESI (Modified, Owned, Exclusive, Shared Invalid)
- Goals
  - Cache to Cache copy of clean data.
  - Cache to Cache move of Dirty data without accessing external memory.

- Recap : Virtual Memory



- Recap : Virtual Memory
- MMU

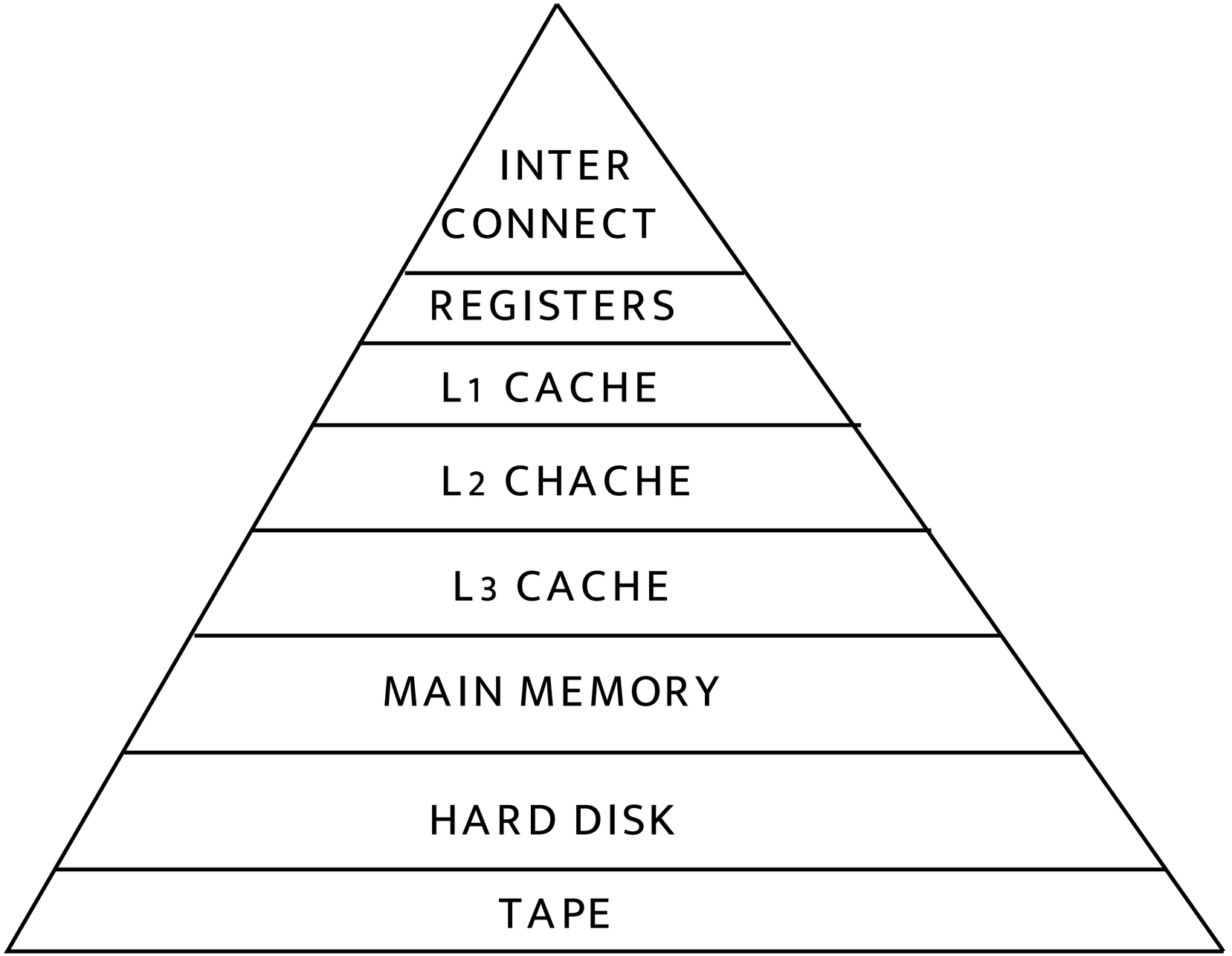


Recap: MMU Operation

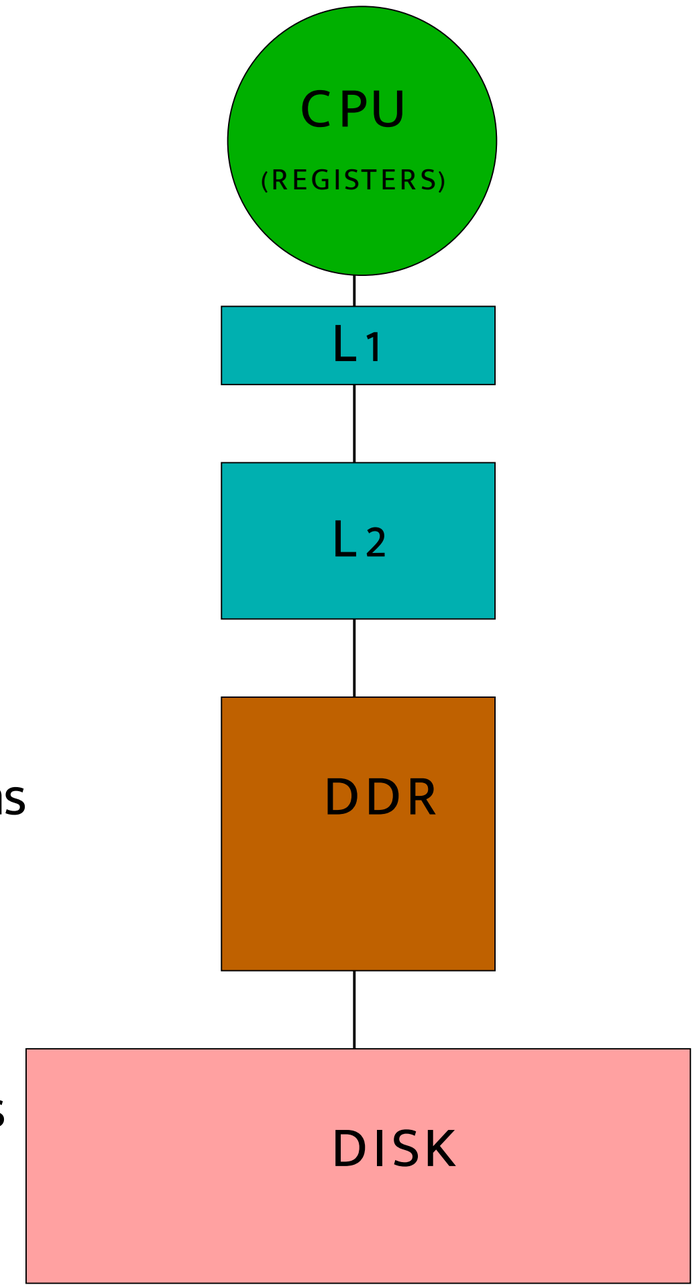
- Translation Lookaside Buffer
  - Keeps a page table for virtual to physical address translation.
  - 4GB memory with page size of 4K => ~4MB
  - Each process has a different page table.
  - page table is kept in main memory.
  - Each access will need two accesses to main memory.
  - TLB acts as a cache for page table entries (PTE).

Recap : Life of a Memory Request

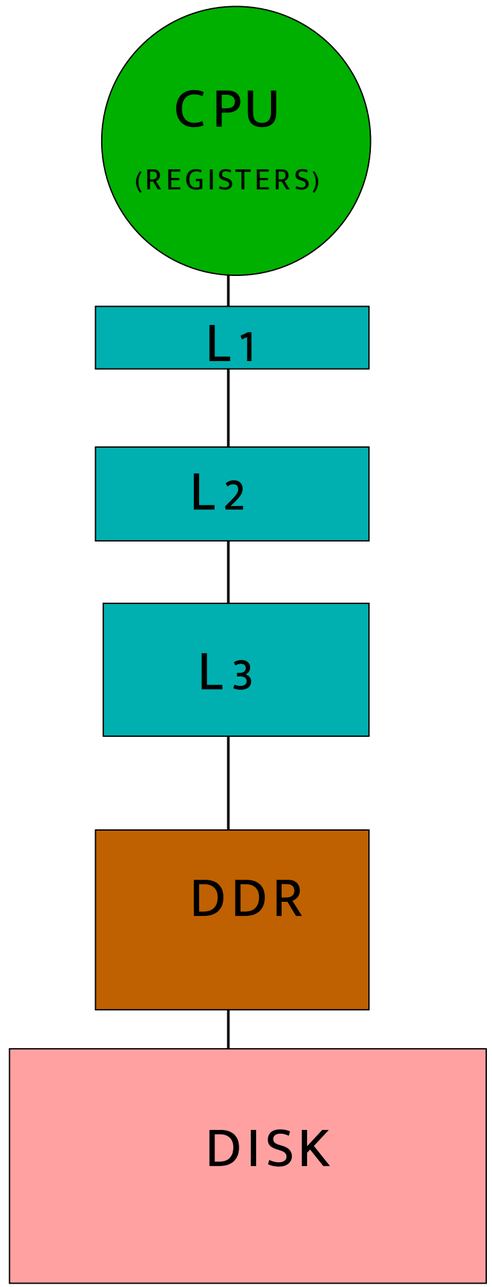
Recap : Memory Hierarchy


INTER
CONNECT
REGISTERS
L1 CACHE
L2 CHACHE
L3 CACHE
MAIN MEMORY
HARD DISK
TAPE

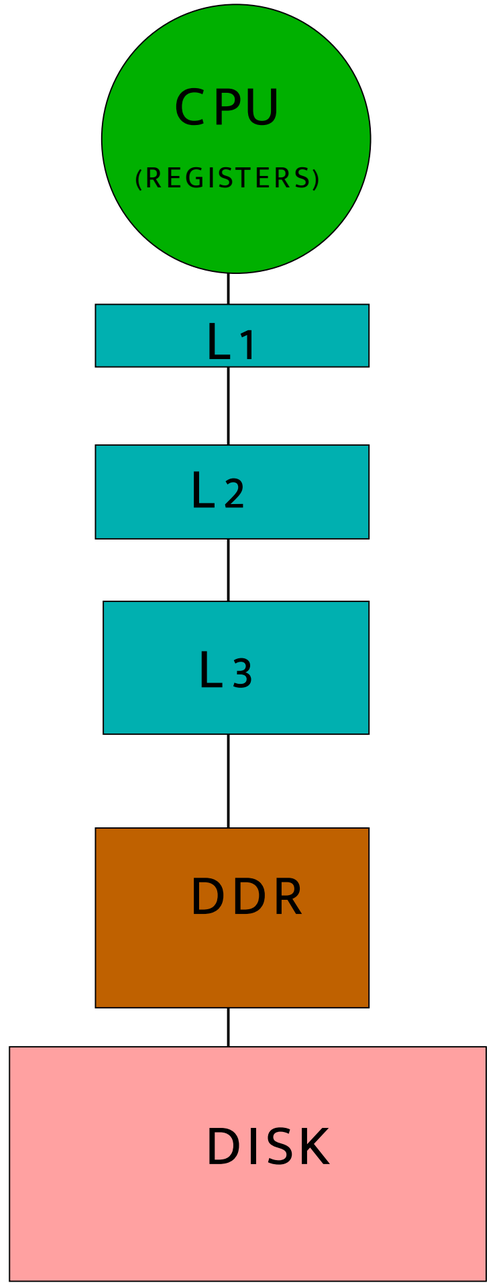
SIZE	SPEED
1KB	~300ps
64KB	~1ns
256KB	~5-10ns
1-2GB	~50-100ns
4-64GB	~25-50us



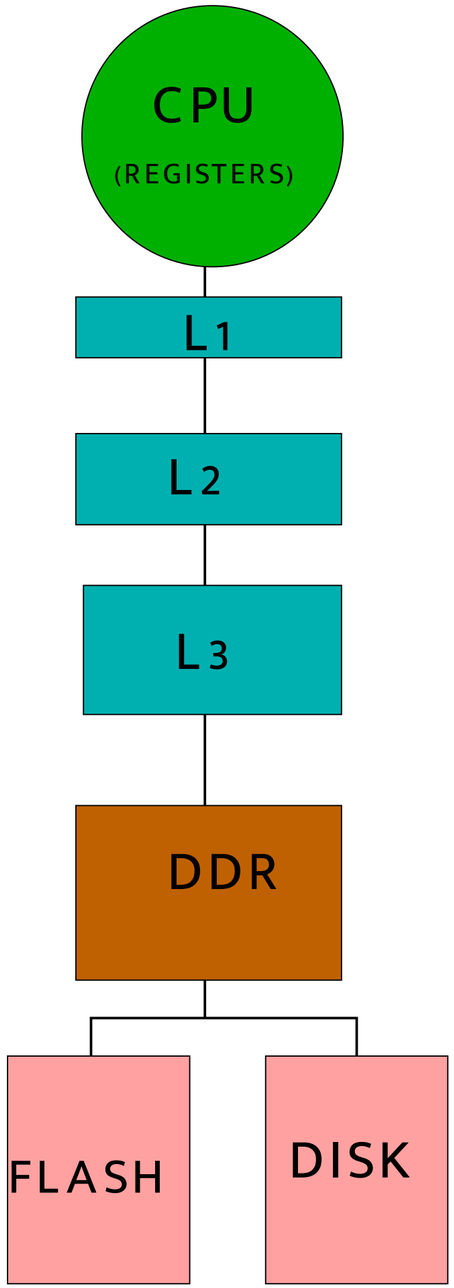
SIZE	SPEED
1KB	~300ps
64KB	~1ns
256KB	~5-10ns
4-8MB	~10-20ns
4-16GB	~50-100ns
256GB -1TB	~50-100us



SIZE	SPEED
2KB	~300ps
64KB	~1 ns
256KB	~3-10ns
8-32MB	~10-20ns
8-64GB	~50-100ns
256GB -2TB	~50-100us

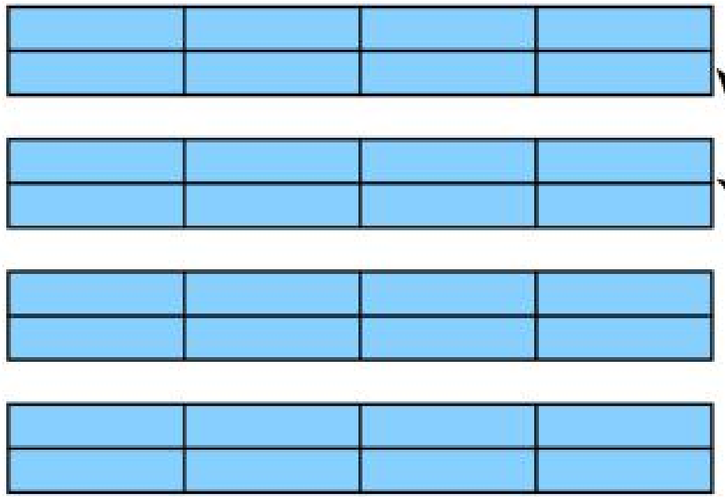


SIZE	SPEED
4KB	~300ps
64KB	~1 ns
256KB	~3-10ns
16-64MB	~10-20ns
32-256GB	~50-100ns
16-64TB	~5-10ms
1-16TB	~100-200us



Example: Cache Side Channel (Ref [3])

Flush+Reload



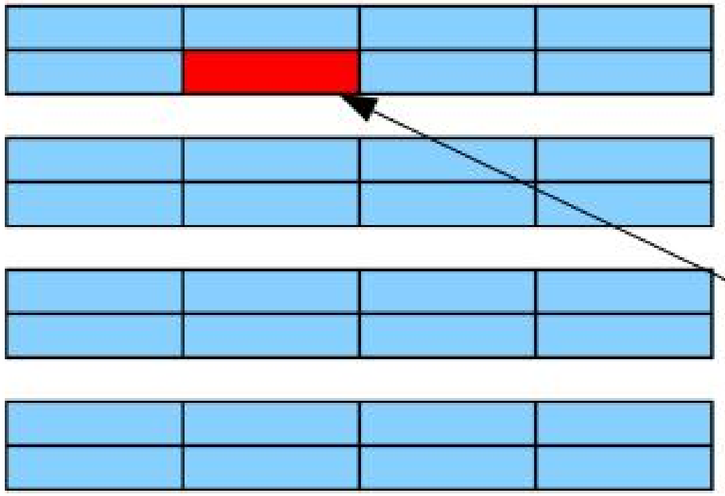
Shared Mem Between Attacker And Victim

Cache Lines

Attacker **flushes** the whole array from the cache

```
#include <intrin.h>
__mm_clflush(array);
```

Flush+Reload

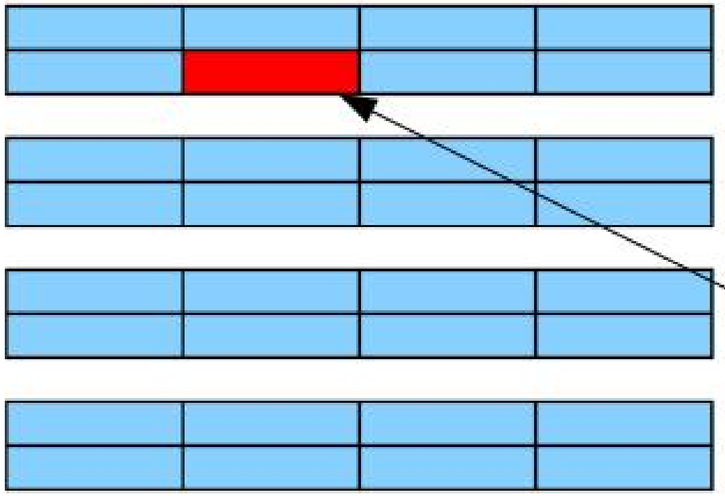


Shared Mem Between Attacker And Victim

Victim acces : bring line into cache

Victim Acceses the shared cache

Flush+Reload



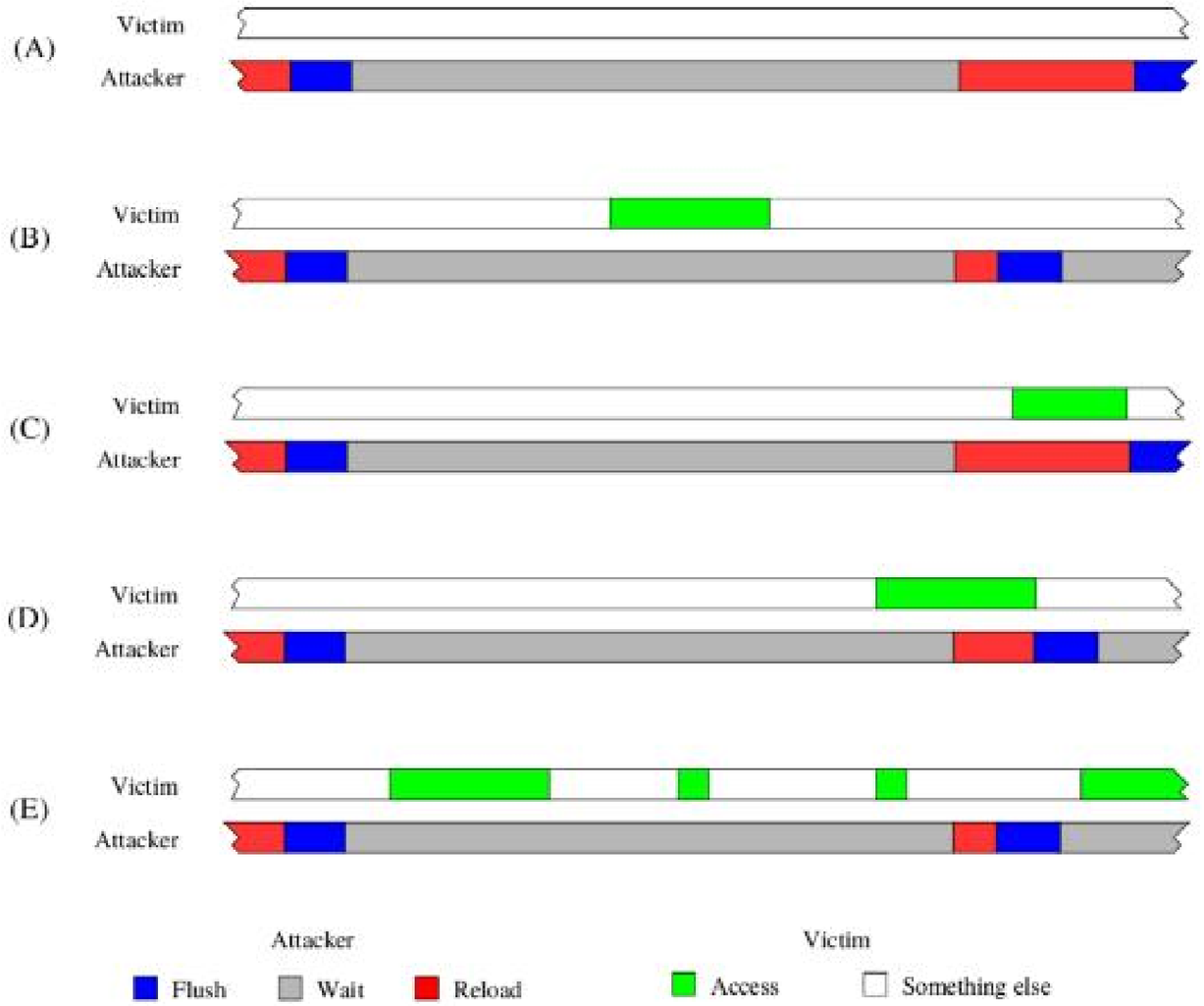
Shared Mem Between Attacker And Victim

Reload acces time

Attackere **Re-acceses** the shared cache, Low access time due to cache hit; Measures access time

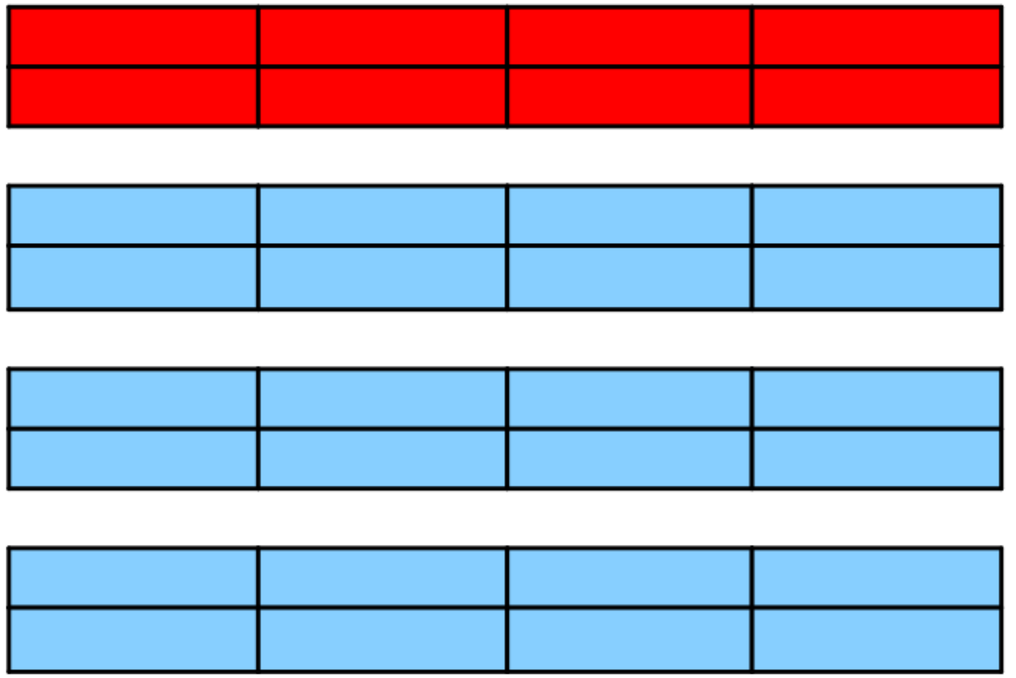
time1 =__rdtscp(array)

Flush+Reload



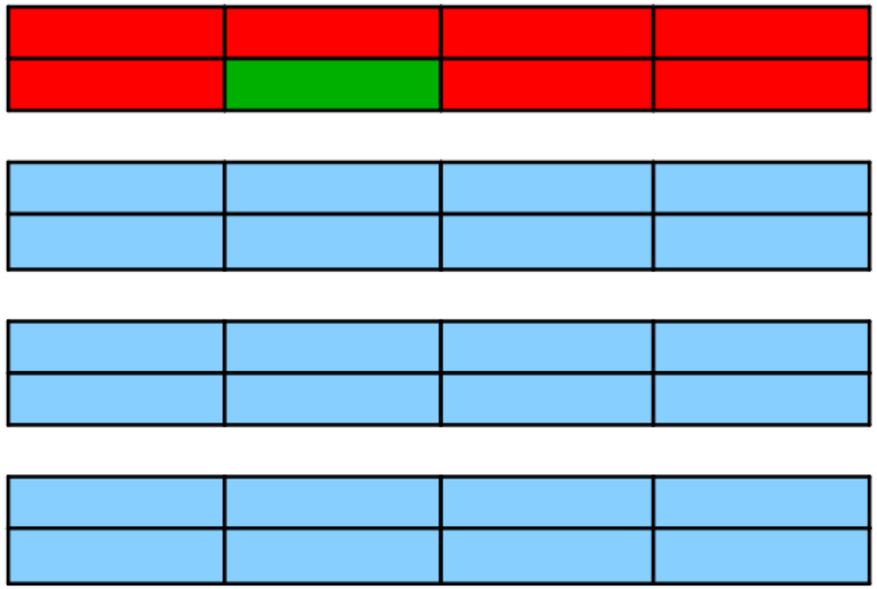
Prime+Probe

- Attacker fills the Cache with an array
-
- Tries to use the same cache lines as the victim
-
- Victim does not acces the cache.
-
- Attacker Probes (reads and measure timing for his own array)
-
-
- Guesses that Victim has not accessed the target memory location

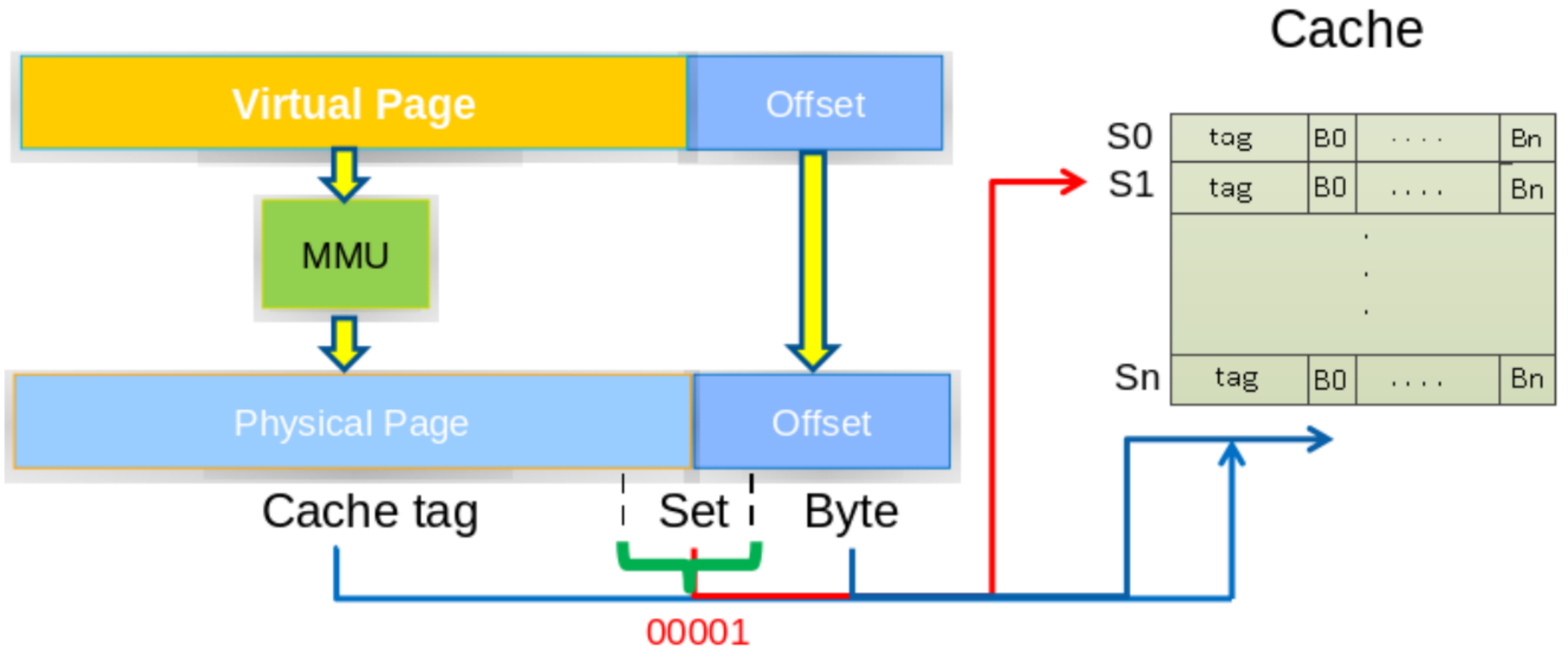


Prime+Probe

- Attacker fills the Cache with an array. (Prime)
- Tries to use the same cache lines as the victim
- Victim access the cache.
- Attacker Probes (reads and measure timing for his own array)
- Guesses that Victim has accessed the target memory location because it has evicted his own array cache line.

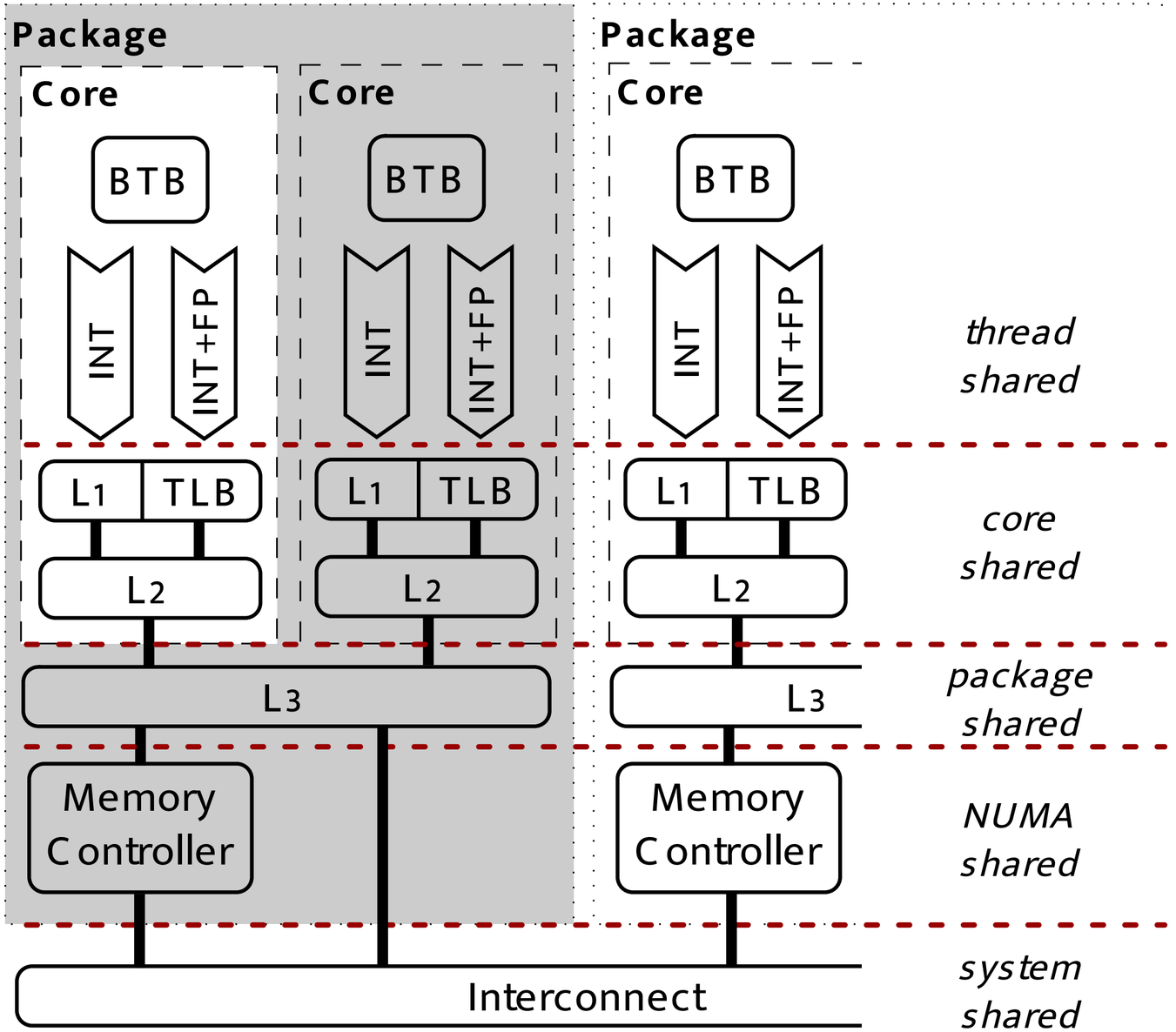


Prime+Probe

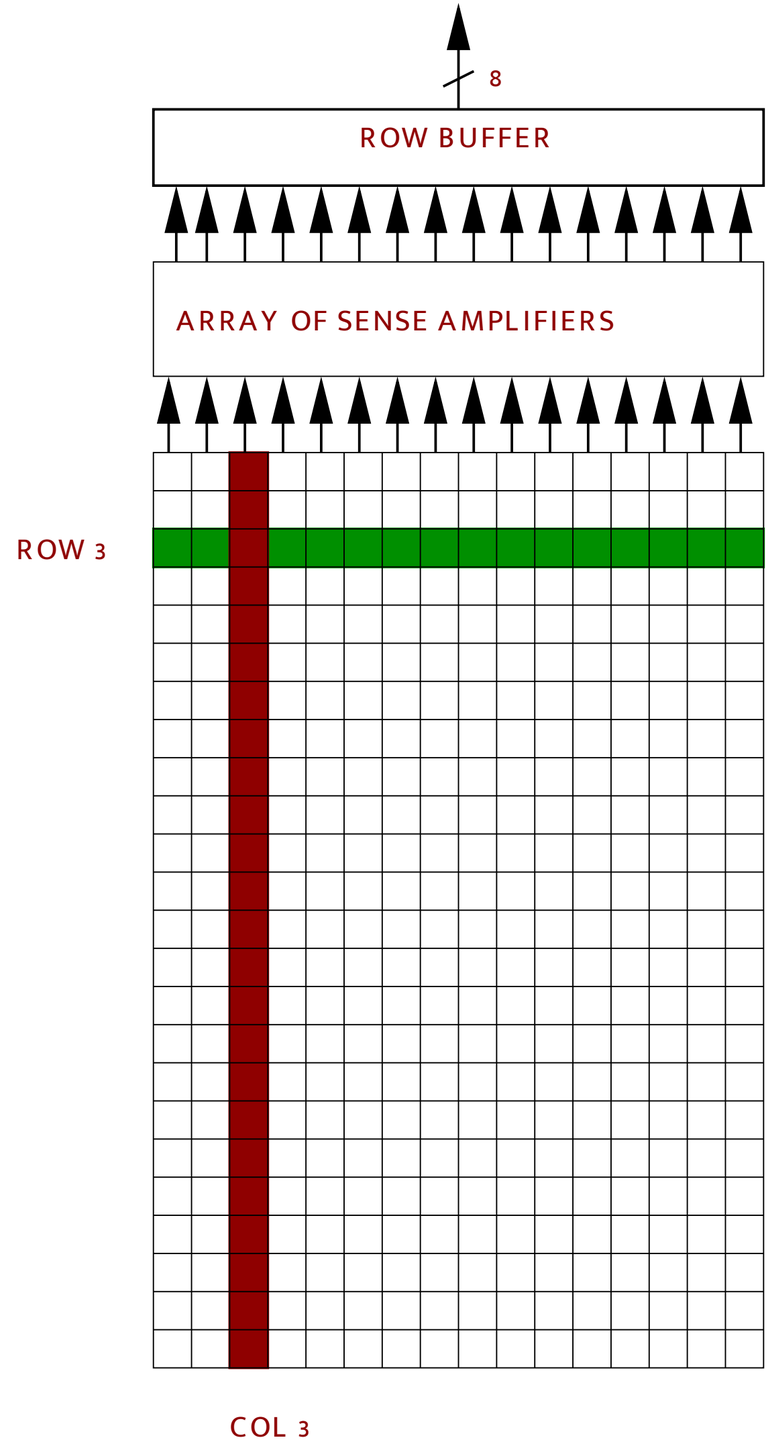


How to calculate the eviction sets ?
What is the offset for a cache line size of 64 bytes ?

SCA Classifications



- DRAM Operation: A single DRAM Bank



DRAM Operation: DRAM Operation

- READ: Activate (open the row)-> Read -> Precharge (close).
- WRITE: Activate (open the row)-> Write -> Precharge (close).
- REFRESH: READ-> WRITE back.

DRAM Traffic TCL,TRCD,TRP

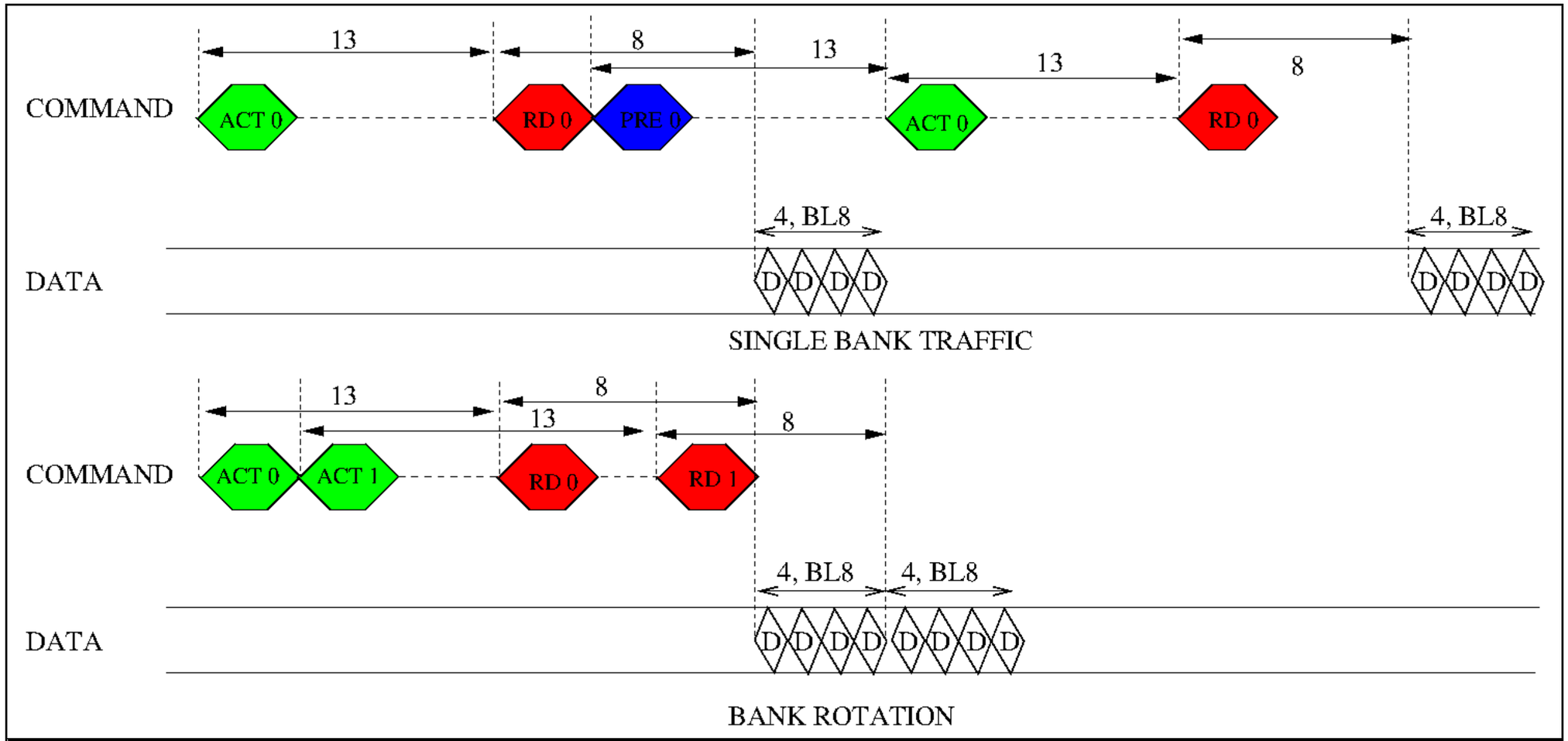
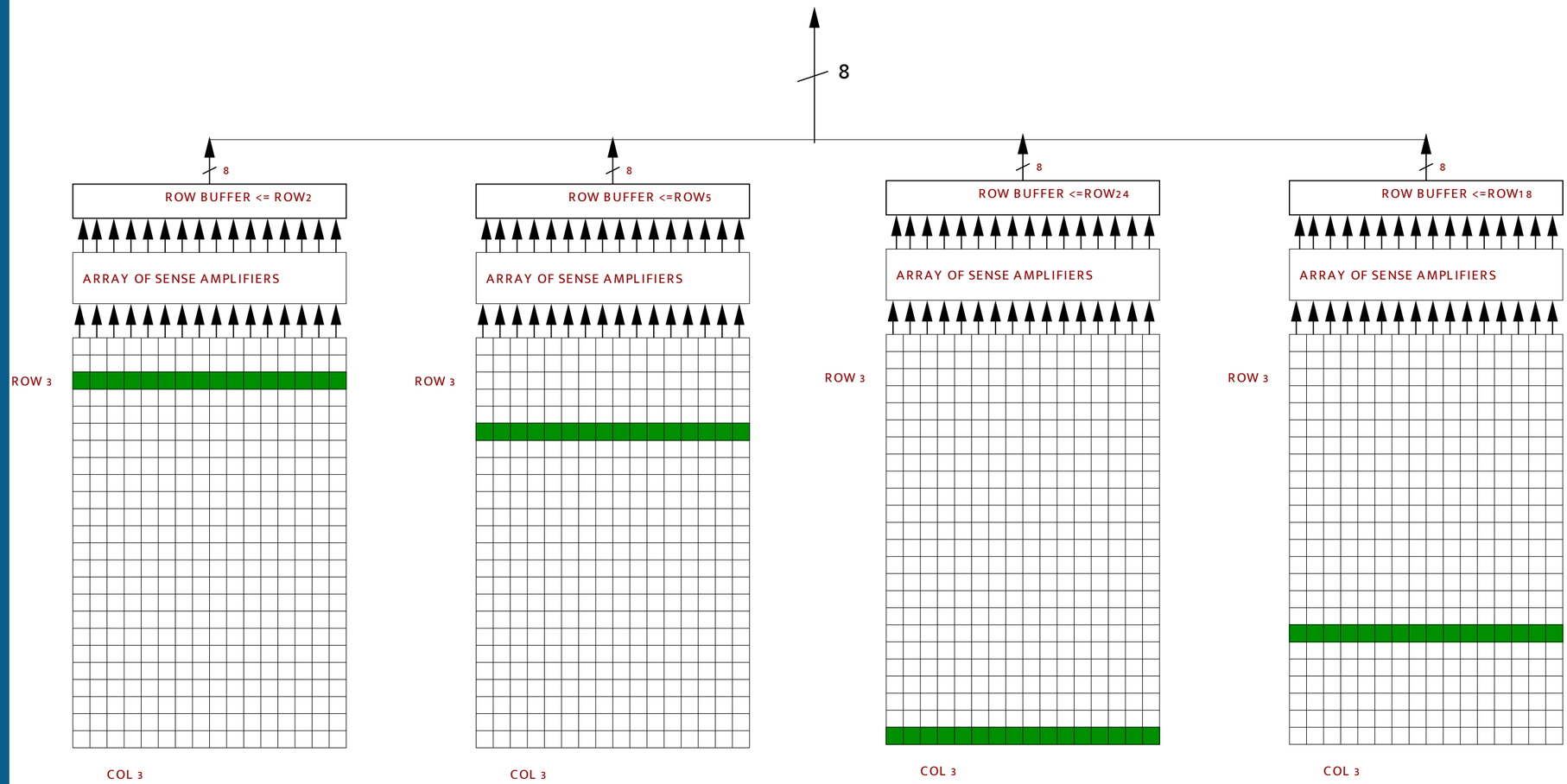


Figure 1 Advantages of Bank Rotation for LPDDR2-1066 with TCL=8, TRCD=13, TRP=13

TCL: CAS Latency, TRCD: ACT to RD or WR command delay, TRP:PRE to ACT command delay

- DRAM Operation: Banks



DRAM Traffic TCL,TRCD,TRP

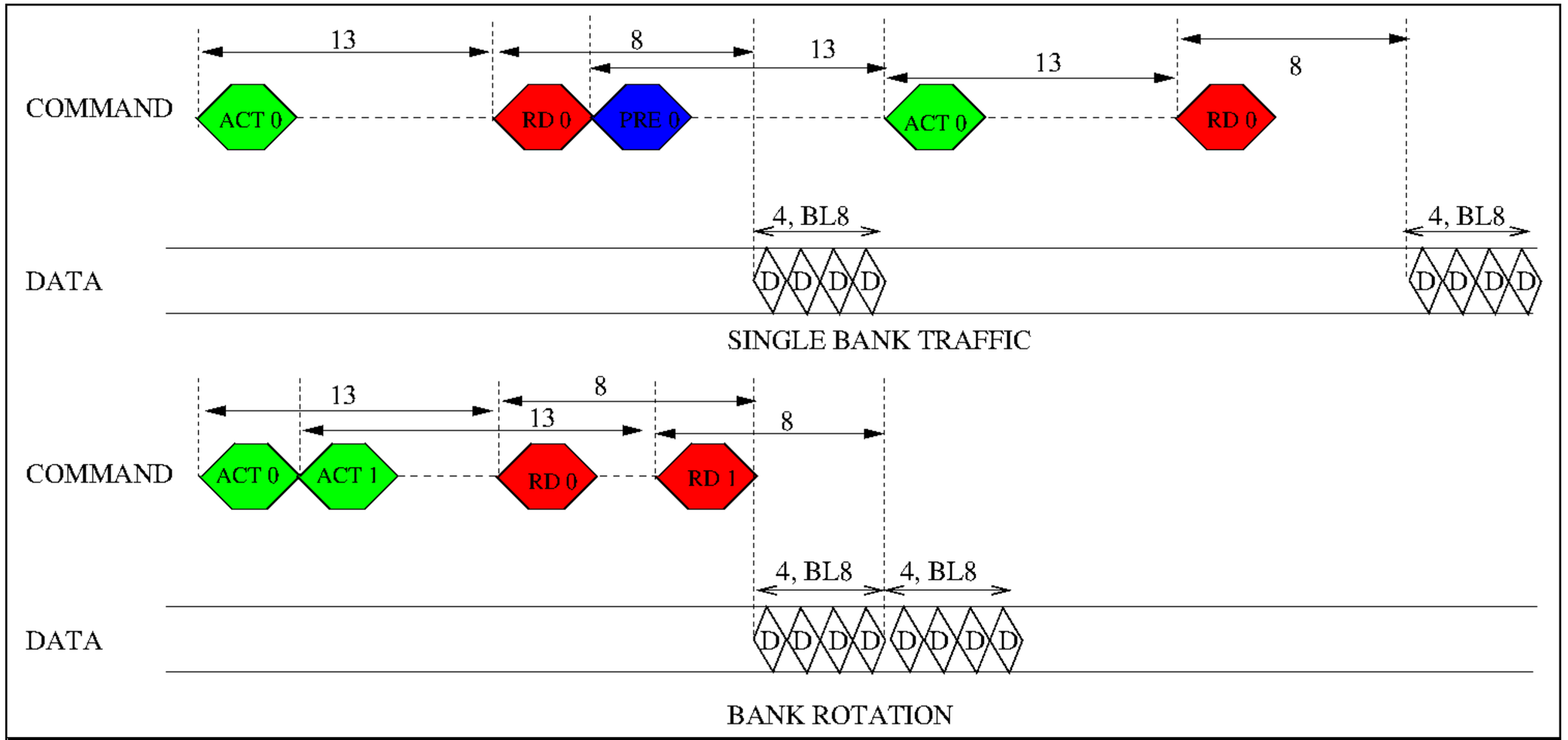


Figure 1 Advantages of Bank Rotation for LPDDR2-1066 with TCL=8, TRCD=13, TRP=13

TCL: CAS Latency, TRCD: ACT to RD or WR command delay, TRP:PRE to ACT command delay

DRAM Traffic

- DRAM is the main performance bottleneck in a SoC.
- DRAM response can come out of order, has high initial latency.

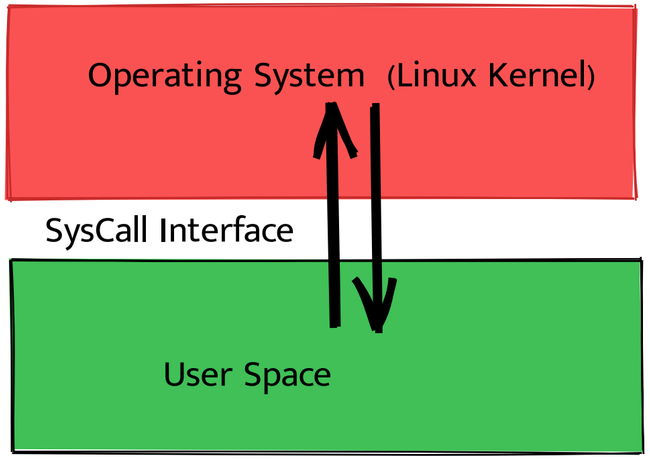
Example: Memory Controller Side Channel (Ref [4])

- Detecting firefox keystrokes from row buffer conflicts.
- find target addresses
- open a row in the same bank.
- detect memory access from rowbuffer hit/miss time.

Standard Protections

(Doesn't protect from Side Channel Attacks)

- Each User Process runs in its own virtual space.
- The security is guranteed through isolation of virtual memory spaces.
- Enforced during address translation.



- Syscalls are the only way to access operating system functions.

Spectre

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt",on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];
```

Speculative execution of branch even when x > array1_size.

```
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}

/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD(80) /* assume cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[256];
```

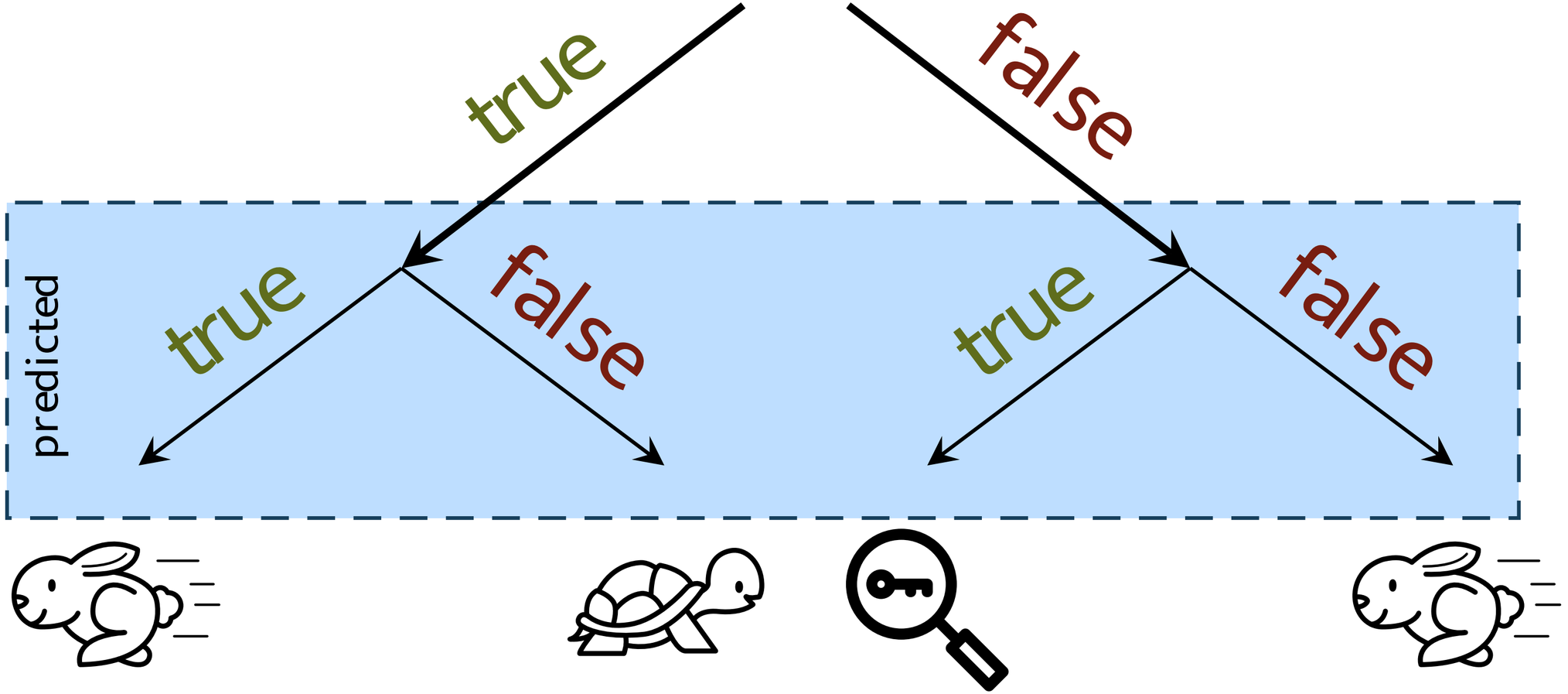
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt",on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
```

```
**********/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];
```

Train the branch predictor for some iterations. Force it to mispredict.

Spectre

if <in bounds>



Spectre

if (x < array1_size) y = array2[array1[x] * 4096];

- To attack
  - â victim_address=array1+x
  - â So â x=victim_address-array1
  - â The array2 index accessed is the value stored in victim_ad

Spectre

if (x < array1_size) y = array2[array1[x] * 4096];

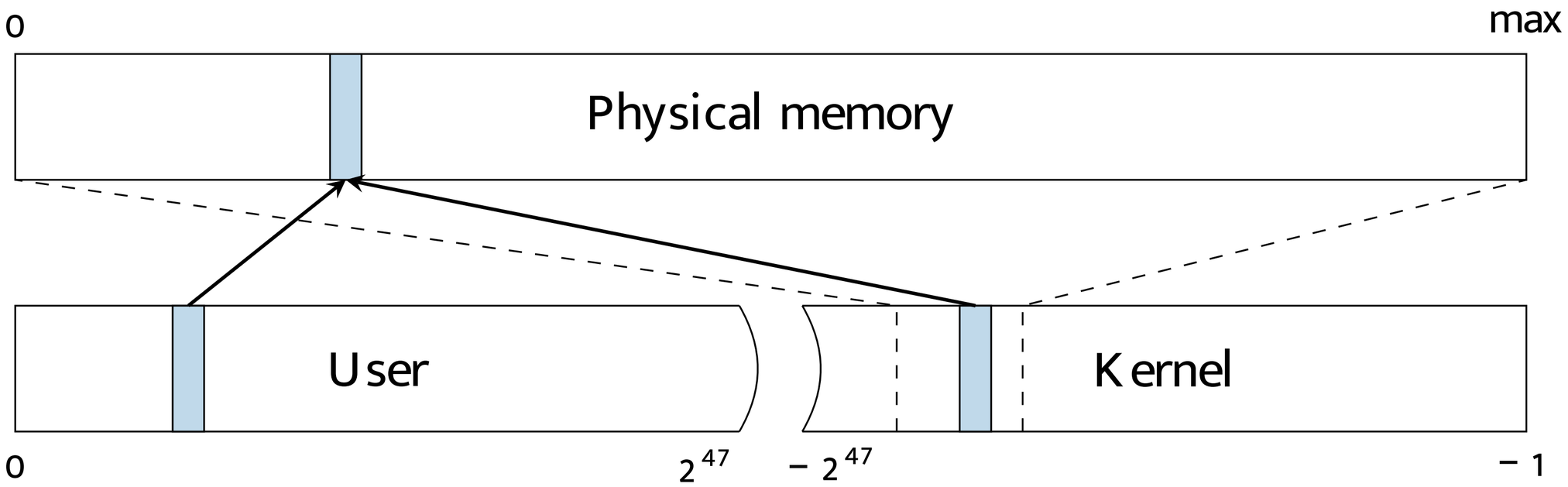
- To attack
  - â Find out the array2 index accessed with Flush+Reloa
  - â Why do we need to multiply by a stride of 64 ?

Spectre Mitigations

- All Out-of-Order Processors are affected by spectre.
- However it is harder to exploit. Need to find code pattern in the victim:

```
    if (x < array1_size)
                y = array2[array1[x] * 4096];
```

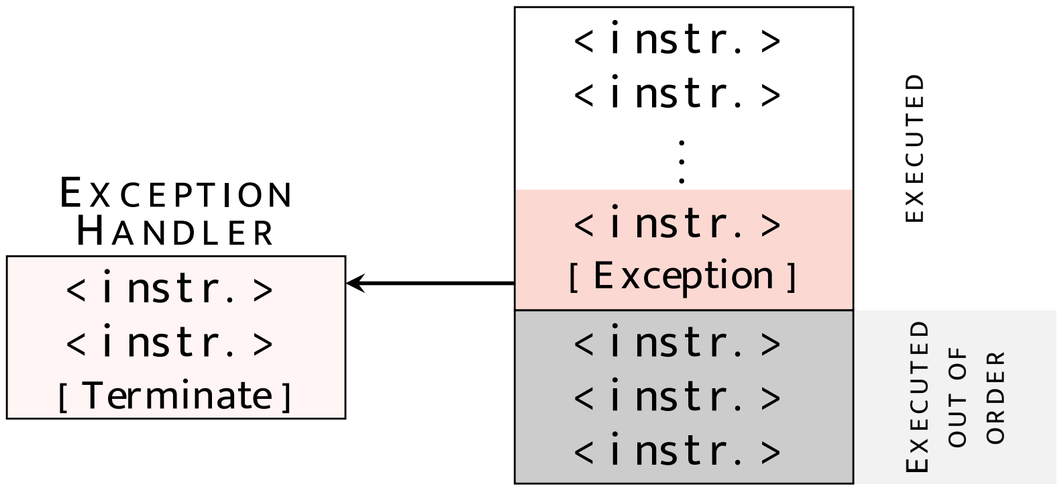
MeltDown



MeltDown

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Listing 1: A toy example to illustrate side-effects of out-of-order execution.



MeltDown

- 1. raise_exception();
- 1. // the line below is never reached
- 1. access(probe_array[data * 4096]); Spill over to the Kernel memory space. Find the value through Flush+Reload.

MeltDown

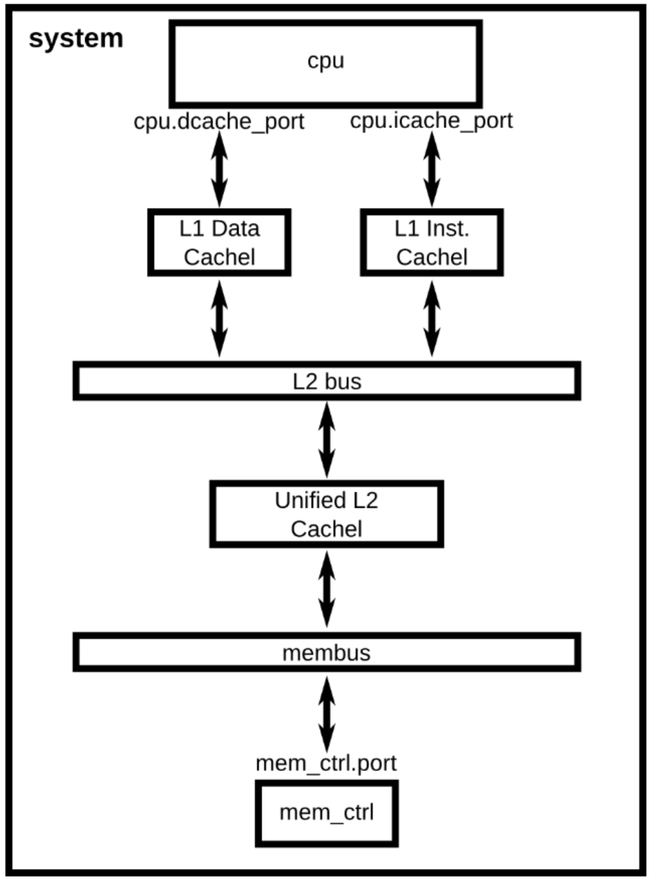
- Step 1 The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.
- Step 2 A transient instruction accesses a cache line based on the secret content of the register.
- Step 3 The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location

MeltDown Mitigations

- KAISER Patch: User space does not have access to kernel memory.
- KASLR (Address space layout randomization): Makes the attack difficult.

TP : GEM5 Config

TP: CONFIG



TP STEP 1

- Clone the repository https://github.com/amusant/micro_archi_attacks
- $source env.sh → sets up environment variables.
- Go to directory hit_miss; look into code hit_miss.c
- Run make to compile the code in hit_miss directory
- Runs $make launch to launch simulation.
- We use the gem5 simulator to simulate a basic system with x86 processor and two levels of cache.
- Understand the code used for
  - Flush
  - Acces
  - Reload
- By changing the acces pattern do you see any difference in the output ?
- What is the role of STRIDE, does the code still work after changing STRIDE ?

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <x86intrin.h> /* for rdtscp and clflush */
#define STRIDE 64
uint8_t array2[256 * STRIDE];

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */



void main(){
  int tries, i, j, k, mix_i, junk = 0;
  register uint64_t time1, time2;
  volatile uint8_t * addr;
  static int results[256];

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 9; tries > 0; tries--) {
```

```
void main(){
  int tries, i, j, k, mix_i, junk = 0;
  register uint64_t time1, time2;
  volatile uint8_t * addr;
  static int results[256];

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 9; tries > 0; tries--) {

        /* Flush  */
        for (i = 0; i < 256; i++)
          _mm_clflush( & array2[i * STRIDE]); /* intrinsic for clflush instruct

        /* access */
        for (i = 0; i < 64; i++)
         temp &= array2[i*STRIDE];

        /* RELOAD */
        for (i = 0; i < 256; i++) {
```

Flushing the array

```
static int results[256];

for (i = 0; i < 256; i++)
  results[i] = 0;
for (tries = 9; tries > 0; tries--) {

    /* Flush  */
    for (i = 0; i < 256; i++)
      _mm_clflush( & array2[i * STRIDE]); /* intrinsic for clflush instruct

    /* access */
    for (i = 0; i < 64; i++)
     temp &= array2[i*STRIDE];

    /* RELOAD */
    for (i = 0; i < 256; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = & array2[mix_i * STRIDE];
      time1 = __rdtscp( & junk); /* READ TIMER */
      junk = * addr; /* MEMORY ACCESS TO TIME */
      time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TI
      results[mix_i]+=time2;
    }
```

Access by Victim

```
      /* Flush */
      for (i = 0; i < 256; i++)
        _mm_clflush( & array2[i * STRIDE]); /* intrinsic for clflush instruct

      /* access */
      for (i = 0; i < 64; i++)
       temp &= array2[i*STRIDE];

      /* RELOAD */
      for (i = 0; i < 256; i++) {
        mix_i = ((i * 167) + 13) & 255;
        addr = & array2[mix_i * STRIDE];
        time1 = __rdtscp( & junk); /* READ TIMER */
        junk = * addr; /* MEMORY ACCESS TO TIME */
        time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TI
        results[mix_i]+=time2;
      }
  }
/* print access time*/
for (i = 0; i < 256; i++)
   printf("%d = %d \n",i,results[i]);
}
```

Reload and measure time

TP STEP 2

- Go to directory flush_reload; look into code flush_reload.c
- The function victim does the following:
- It accesses the array[secret[desknumber][i]*STRIDE]
- Where the secret is a 16 character secret key.
  - secret[desknumber]=”XXXXXXXXXXXXXXXX”
- Your goal is to find the 16 characters of the secret value.
- The secret value changes with desk number.
- Run make to compile the code in flush_reload directory
- Runs $make launch to launch simulation.
- Inspire yourself from the hit_miss code.

TP STEP 3

- Download the Spectre Example link from
  - spectre/link
- Read and Understand the code.
- Compile the code -$gcc spectre.c -Launch the experiment

```
-$gem5.opt ../configs/two_level.py ./a.out
-does it work ?
```

- Change line 99 in ../configs/two_level.py
  - from DerivO3CPU() to TimingSimpleCPU()
  - realaunch simulation
  - Does it work ?

References

- [1] Computer Architecture: A Quantitative Approach. Hennesssy & Patterson
- [2] Predicting Secret Keys via Branch Prediction, Onur Acıicmez, Jean-Pierre Seifert, and C¸ etin Kaya Ko¸c
- [3] FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack
- [4] Meltdown: Reading Kernel Memory from User Space Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher,Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard,Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg

References

- [5] DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard,
- [6] A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware Qian Ge, Yuval Yarom2 , David Cock, and Gernot Heiser
- [7] Spectre Attacks: Exploiting Speculative Execution Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin,Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp,Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom