

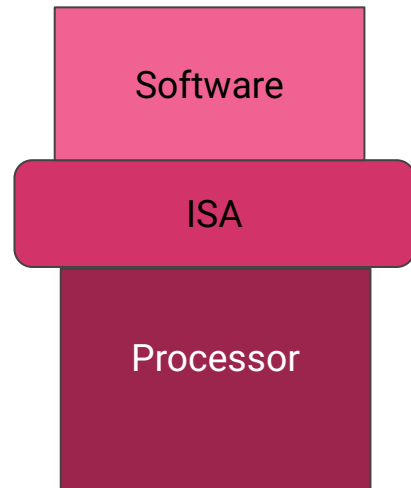
# In Order Superscalar from Basics

By-

- Anirudh Kr. Jangid
- Praveen Prajapat

# ISA (Instruction Set Architecture)

- **Defines the set of instructions** a processor can execute (e.g., arithmetic, logical, memory access, control flow).
- **Specifies hardware-visible elements** like registers, data types, addressing modes, and instruction formats.
- Acts as a **contract** between software and hardware, enabling software compatibility across different processors.
- Defines the actual requirements from the processor and partially affects the design.
- Examples are x86, RISC-V, SPARC, etc.



# ISA & Assembly

- **Low-level human-readable code** that directly corresponds to machine instructions defined by an ISA.
- **Uses mnemonics** (e.g., ADD, SUB, MOV) and symbolic names for registers and memory addresses.
- **Requires an assembler** to convert it into executable machine code for the processor.
- Machine code is a binary file which is executed by processor.

```
.section .text
.global floatadd
floatadd:
    # float floatadd(float a, float b)
    # fa0 floatadd(fa0, a0, fa1, b0)

    flw fa2, 0(a0)
    fld fa3, 0(a1)

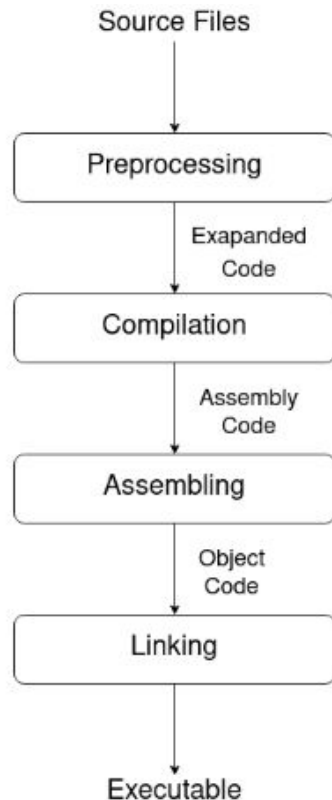
    fcvts.d    fa1, fa1
    fcvts.d    fa3, fa3

    fadd.s    fa1, fa1, fa3
    fadd.s    fa0, fa0, fa2
    fadd.s    fa0, fa0, fa1

    ret
```

# Breaking down a program

- **All software runs on hardware**, which only understands machine instructions defined by the ISA.
- **Compilers translate high-level code** into low-level ISA assembly instructions during the build process.
- **Complex logic is decomposed** into simple operations like arithmetic, memory access, and control flow supported by the ISA.
- The process for C code is explained in the diagram.



# Breaking down a program

The ISA instructions can be categorized as:

- **Arithmetic and Logical Instructions:** Perform operations like ADD, SUB, AND, OR, XOR, etc.
- **Data Transfer Instructions:** Move data between memory and registers, e.g., LOAD, STORE, MOV.
- **Control Flow Instructions:** Change the execution order, e.g., JUMP, BRANCH, CALL, RETURN.

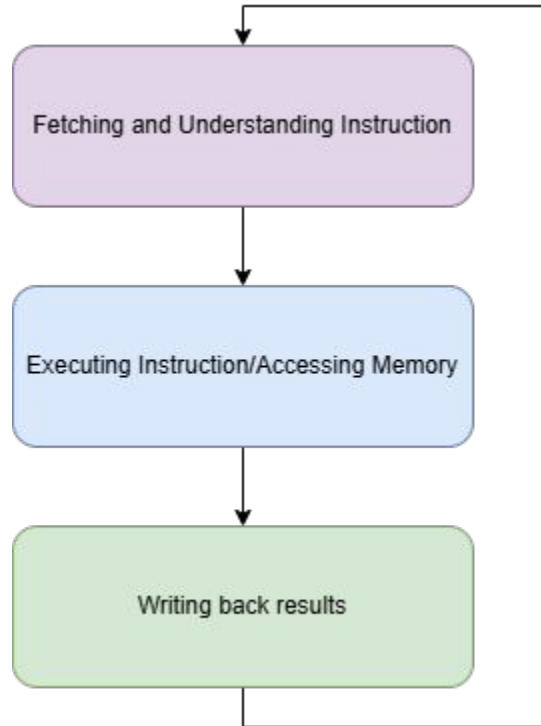


# Types of Instructions

- **Memory Instructions:** Used for memory access and data movement
  - **LOAD:** Fetch data from memory to register (e.g., LD R1, 0(R2))
  - **STORE:** Save data from register to memory (e.g., ST R1, 4(R2))
- **Arithmetic and Logic Instructions:** Perform computation on register operands
  - **Arithmetic:** ADD, SUB, MUL, DIV (e.g., ADD R1, R2, R3)
  - **Logic:** AND, OR, XOR, NOT (e.g., AND R4, R5, R6)
- **Control Flow Instructions:** Alter the sequence of program execution
  - **Conditional Branch:** BEQ, BNE (e.g., BEQ R1, R2, Label)
  - **Unconditional Jump:** JMP, JAL, RET



# Stages of program execution



# Stages of program execution

- Instruction Fetch (IF):
  - Fetch the instruction from instruction memory using the **Program Counter** (and update the PC).
- Instruction Decode (ID):
  - Decode the instruction in order to generate control signals (which shall be used ahead by the components).
- Register File Read (RF):
  - Read operands from register file.
  - This stage may be merged with the ID stage.
- Execute (EX):
  - Perform ALU operations (add, sub, etc.).
  - Compute branch targets or memory addresses.
- Memory Access (MEM):
  - Access memory for **load/store** operations.
  - Only used for instructions involving memory.
- Register WriteBack (WB):
  - Write the result back to the register file.





# Parts of Processor

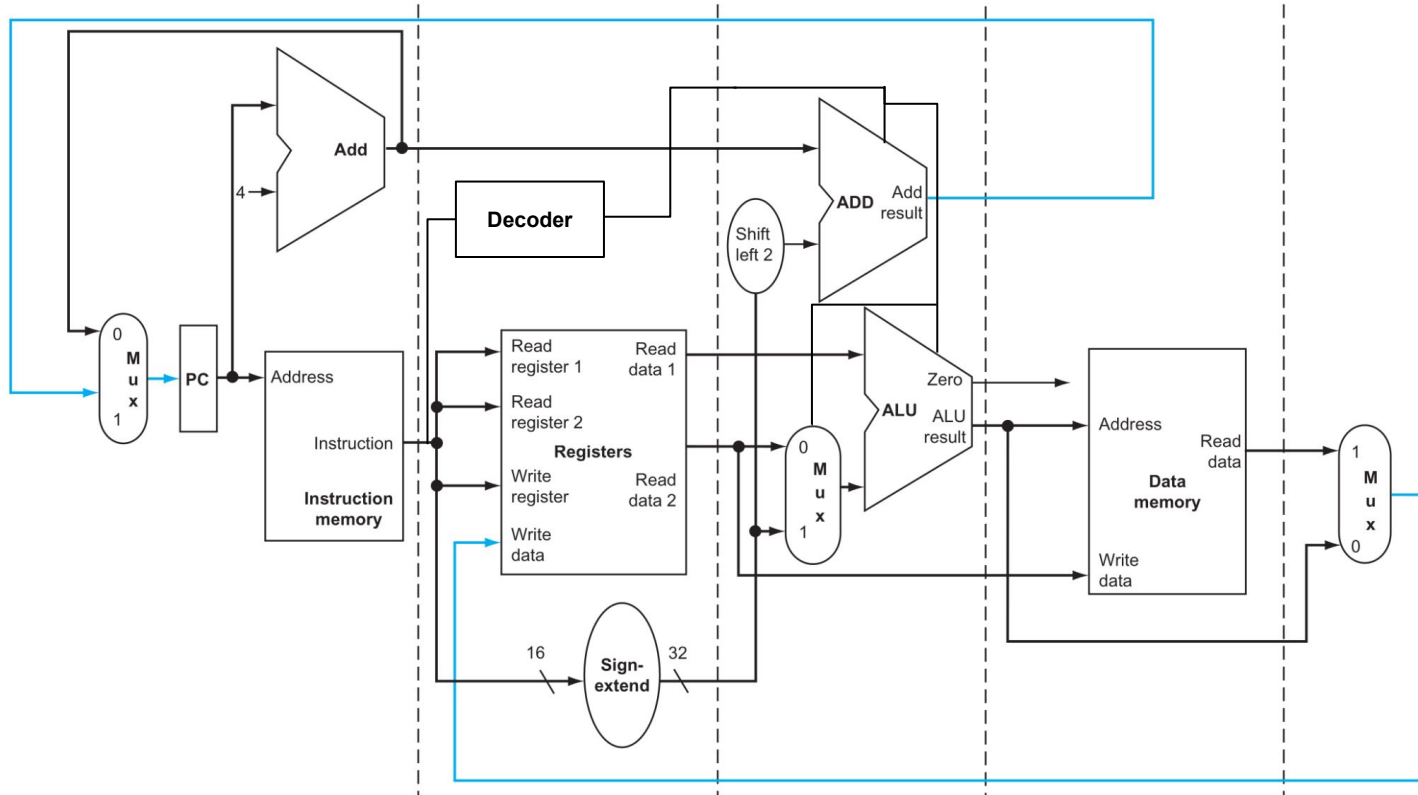
## Datapath-

- **Carries out actual data operations**, including arithmetic, logic, memory access, and register transfers.
- **Includes components like ALU, registers, multiplexers, and memory interfaces** that process and route data.
- **Works in coordination with the control unit**, which directs how data flows through the datapath each cycle.

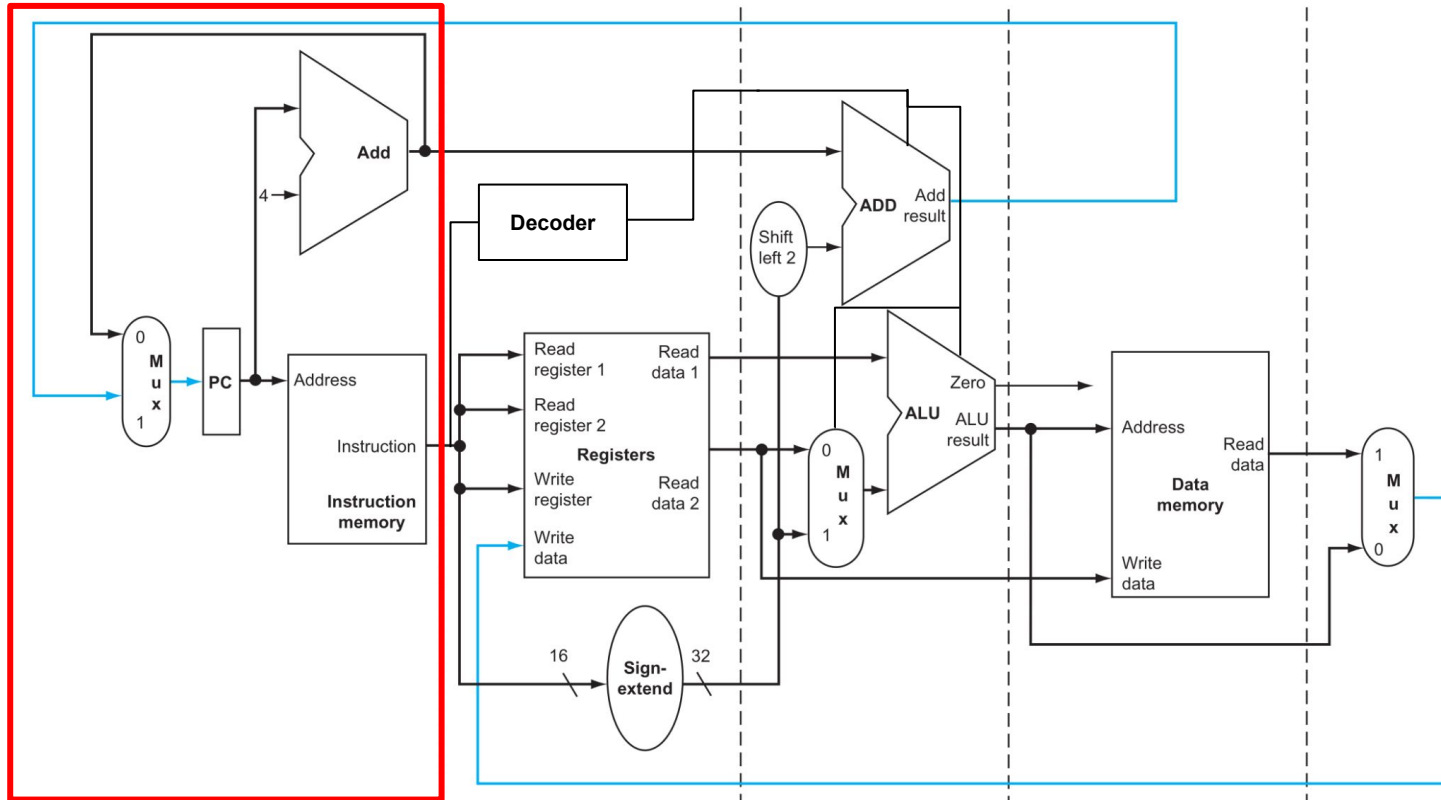
## Controller-

- **Generates control signals** to direct the operation of the datapath components (e.g., ALU control, mux selection).
- **Interprets instruction opcodes** and decides the sequence of actions needed to execute them.
- **Ensures correct timing and coordination** between stages like fetch, decode, execute, memory access, and write-back.

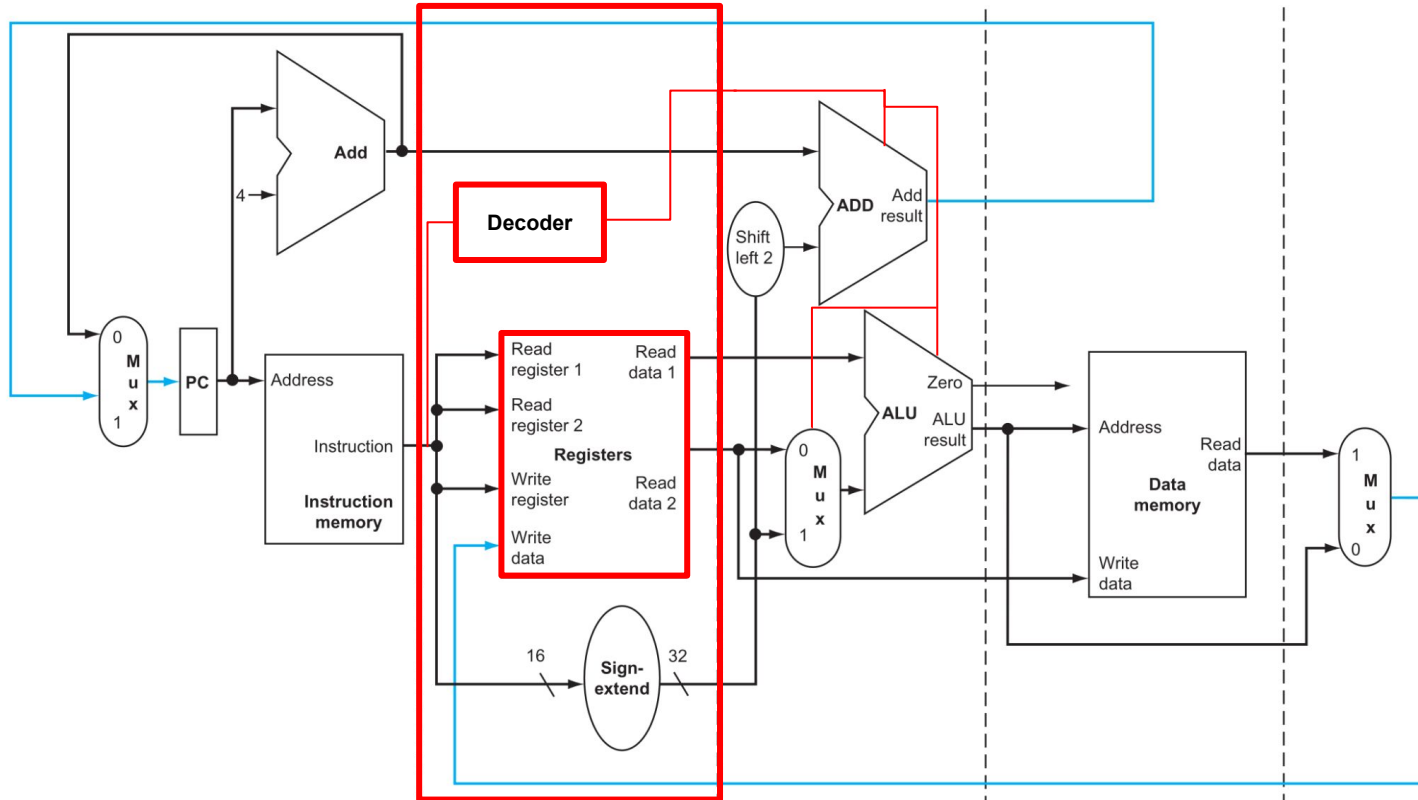
# Datapath



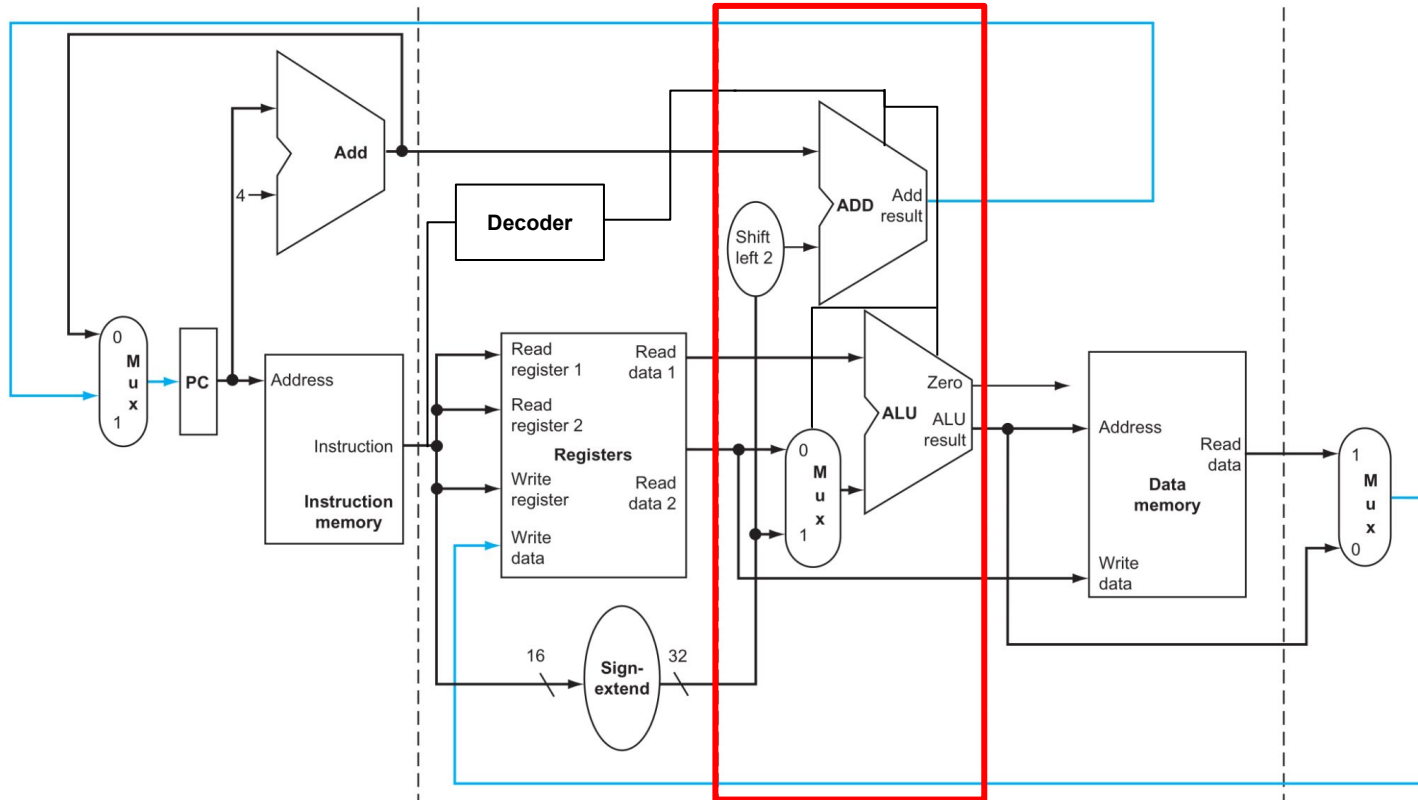
# Instruction Fetch



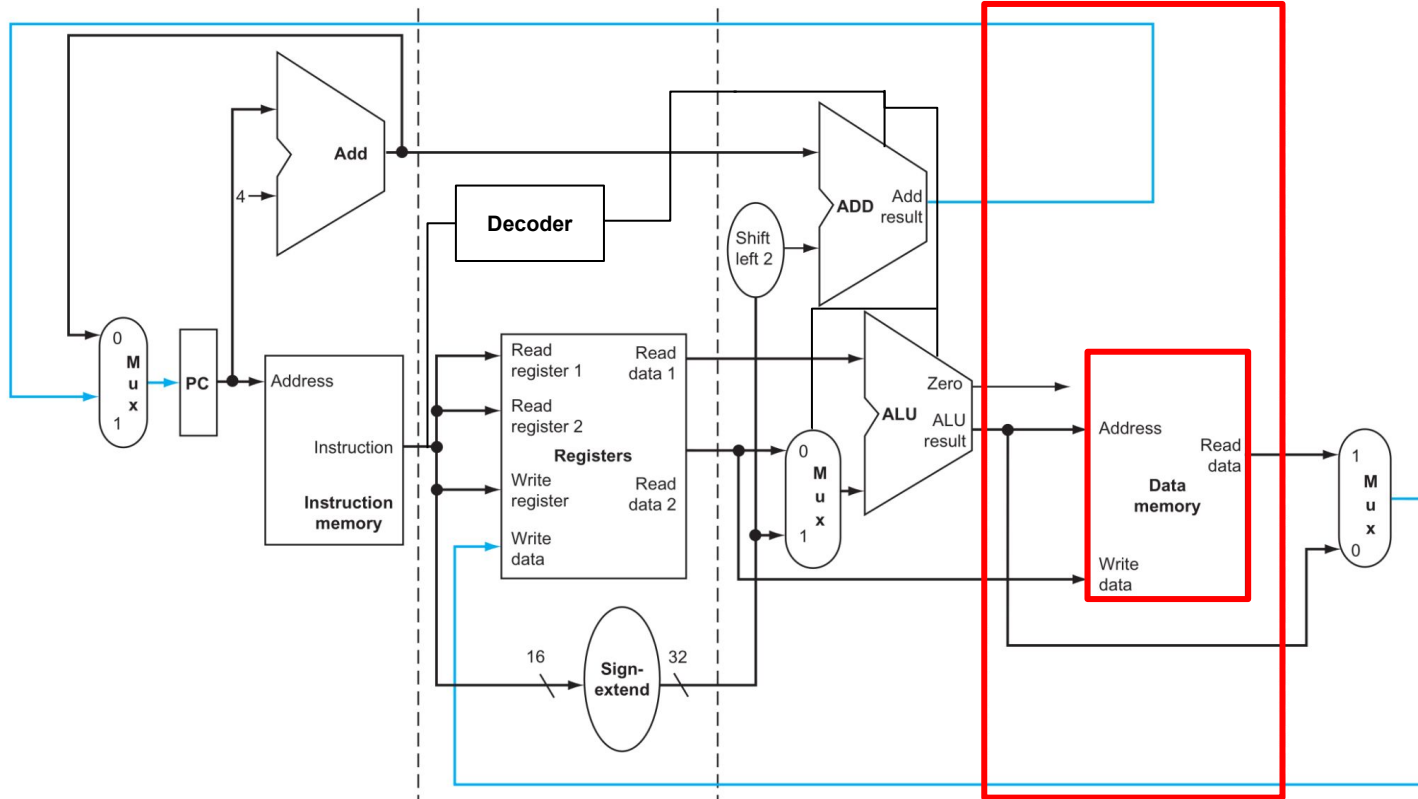
# Instruction Decode and Register File Read (ID & RR)



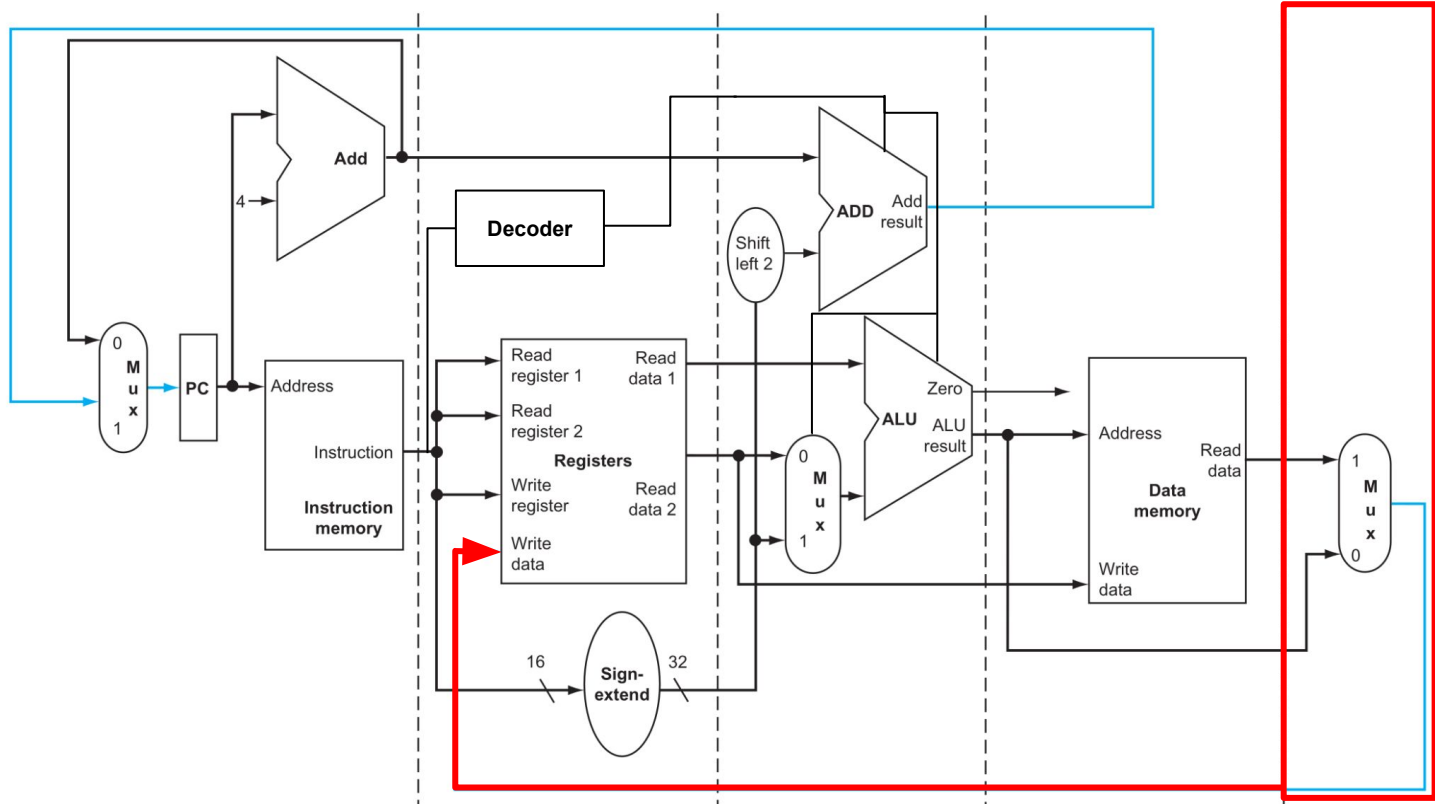
# Execute (EX)



# Memory Access (MEM)



# Register WriteBack (WB)



# Multicycle Processor

1	2	3	4	5	6	7	8	9	10
IF1	ID1	EX1	MEM1	WB1					
					IF2	ID2	EX2	MEM2	WB2





# Pipelined Processor

1	2	3	4	5	6	7	8	9	10
IF1	ID1	EX1	MEM1	WB1					
	IF2	ID2	EX2	MEM2	WB2				
		IF3	ID3	EX3	MEM3	WB3			
			IF4	ID4	EX4	MEM4	WB4		
				IF5	ID5	EX5	MEM5	WB5	

# HAZARDS!!!

## 1. Structural Hazards

- Occurs due to limitation of hardware resources in the processor.
- Causes program to stall if the required hardware resource is busy.

## 2. Data Hazards

- When consecutive instructions are dependent on each other for data.
- Latest data is not updated in register file yet for other processes to read.

## 3. Control Hazards

- Instruction like branching may hamper the performance.
- What will happen if I3 was a taken branch and I4, I5 were on not taken path.



# Mitigations

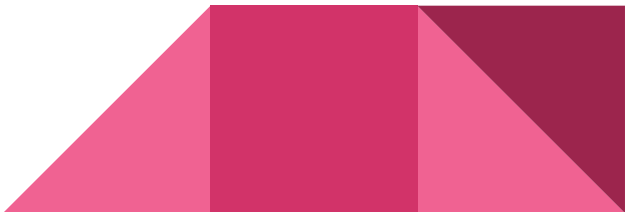
## 1. Structural hazards

- Proper arrangement and division of hardware can mitigate this issue.
- Making the resource time shared or space shared is the way.

## 2. Data hazards

- Bypassing or making values available immediately by skipping the pipe!
- Stalls may be required to mitigate this hazard.

## 3. Control Hazards

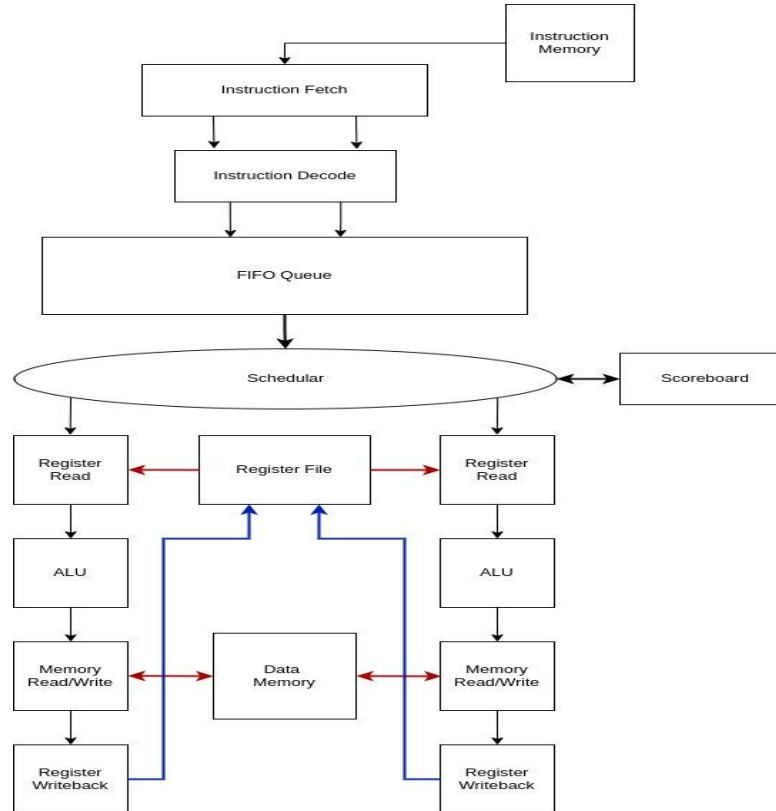
- Flushing the pipe! Or Making the instructions invalid
  - Usage of Branch Predictors
  - Using branch delay slots
  - And others...
- 

# Superscalar Processors

- Can execute more than one instruction per clock cycle.
- Uses multiple execution units (e.g., ALUs, FPUs) in parallel.
- Fetches and decodes multiple instructions at once.
- Improves performance through instruction-level parallelism.
- Needs hardware logic to detect dependencies and avoid hazards.



# In Order Superscalar Flow



# In Order Superscalar

- **Executes multiple instructions per cycle**, like any superscalar processor.
- **Follows program order strictly**—instructions are issued, executed, and completed in the order they appear.
- **Can issue multiple instructions only if there are no dependencies** and execution units are available.
- **Simpler control logic** compared to out-of-order processors.
- **Performance limited by instruction dependencies** and branch delays, since it can't reorder to avoid stalls.



# In Order Superscalar(2)

- **Instruction-Level Parallelism (ILP):** Executes multiple independent instructions in parallel.
- **Static Scheduling:** Compiler arranges instructions to reduce stalls and improve pipeline efficiency.
- **Data Independence:** Takes advantage of instructions that don't rely on each other.
- **Predictable Control Flow:** Performs best with linear code and minimal branching.
- **Multiple Functional Units:** Uses separate execution units to handle different instruction types simultaneously.




# Out of Order Superscalar and Memory Management

By-

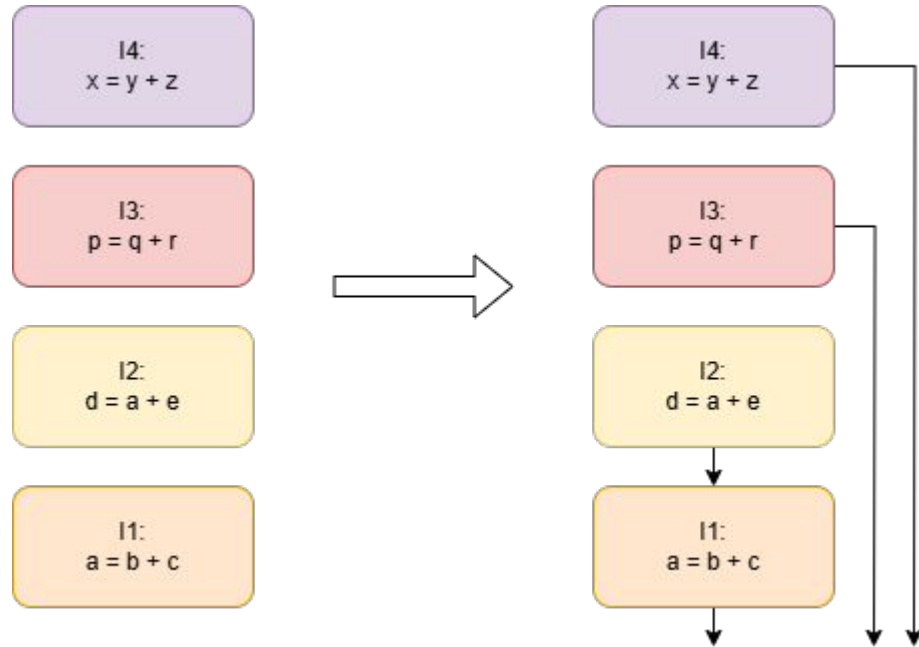
- Anirudh Kr. Jangid
- Praveen Prajapat



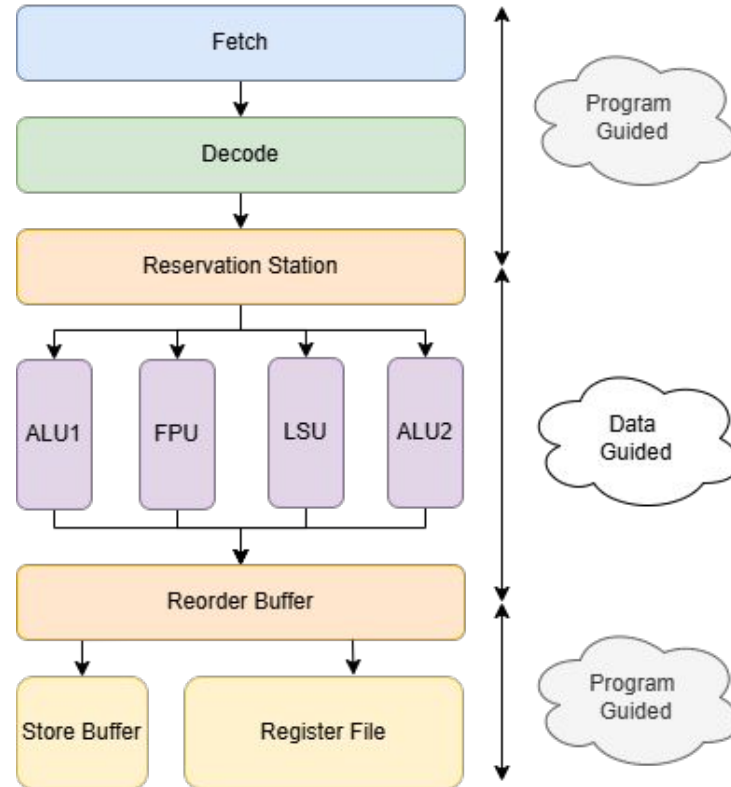
# Out of Order Superscalar

- **Fetches and decodes instructions in program order**, but **executes them as soon as operands are ready**.
  - **Improves performance** by avoiding stalls due to instruction dependencies or long-latency operations.
  - **Uses hardware structures** like reservation stations, reorder buffer (ROB), and register renaming.
  - **Executes instructions out-of-order**, but **commits results in-order** to maintain program correctness.
  - **Handles data and control hazards dynamically**, enabling more efficient use of execution units.
- 

## Out of Order Superscalar(2)



# Out of Order Superscalar(3)



# Out of Order Superscalar(4)

## Properties exploited-

- **Instruction-Level Parallelism (ILP):** Executes independent instructions in parallel, even if they are far apart in program order.
- **Dynamic Scheduling:** Decides execution order at runtime based on operand availability and resource readiness.
- **Latency Hiding:** Continues executing other ready instructions while waiting for slow operations (e.g., memory loads).
- **Register Renaming:** Eliminates false dependencies (name dependencies) to increase parallelism and avoid write hazards. Mitigates WAW and WAR hazards.

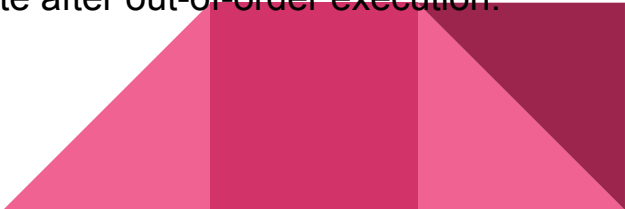


# Reservation Station

- **Hold instructions waiting for execution**, along with their operands or tags (if operands are not yet ready).
- **Allow instructions to issue out-of-order** as soon as all their operands are available.
- **Track operand readiness dynamically**, reducing stalls due to dependencies.
- **Enable multiple functional units** to stay busy by feeding them ready instructions independently.



# Reorder Buffer

- **Stores instructions in program order** after they are issued, to support in-order commitment. Implemented as a **FIFO queue**.
  - **Holds results temporarily** until instructions are ready to commit without violating program correctness.
  - **Ensures precise exceptions** by allowing the processor to roll back to a known good state.
  - **Tracks instruction status** (issued, executed, committed) to manage flow and recovery.
  - **Works with register renaming** to maintain correct architectural state after out-of-order execution.
- 

## Reorder Buffer(2)

I1
I2
I3
I4
<b>I5</b>
I6
I7
I8

<b>I5</b>
I6
I7
I8
I9
I10
I11
I12

FLUSH IT!!!

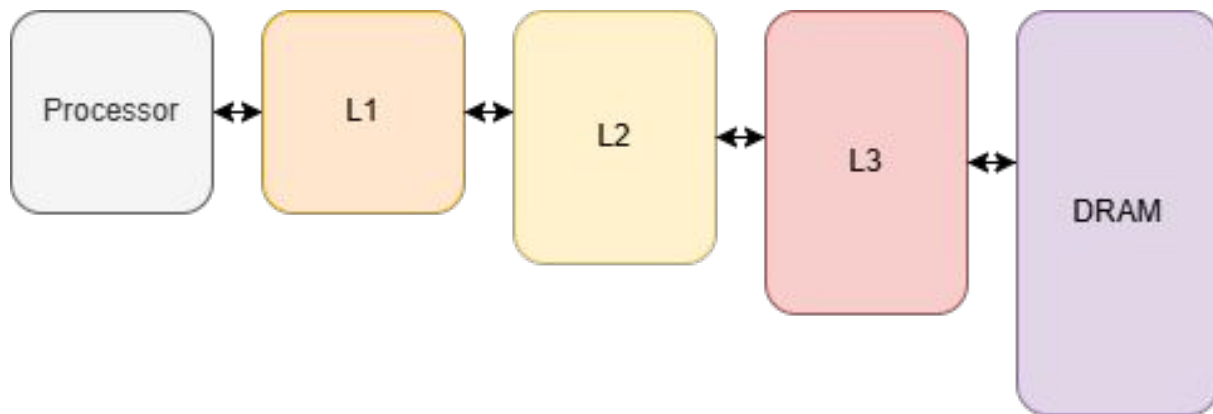
Doubts?





# Memory Hierarchy

- **Bridges the speed gap** between fast CPUs and slow main memory.
- **Provides fast access** to frequently used data through caches and registers.
- **Balances cost, size, and speed** by combining small, fast memory with large, slower memory.
- **Improves overall system performance** by reducing average memory access time.
- **Scales efficiently** as programs and data grow, without needing all memory to be fast or expensive.



# Memory Hierarchy(2)

## **L1 Cache (~1 – 5 ns or ~ 2 to 10 cycles):**

- Small (32–64 KB), very fast on-chip cache.
- Stores frequently accessed instructions and data.


## **L2 Cache (~5 – 20 ns or ~ 10 to 40 cycles):**

- Larger (256 KB – 1 MB), slightly slower than L1.
- Acts as a backup to L1 cache.

## **L3 Cache (~20 – 50 ns or ~40 to 100 cycles):**

- Shared across cores (size up to 10–30 MB).
- Slower but helps reduce access to main memory.

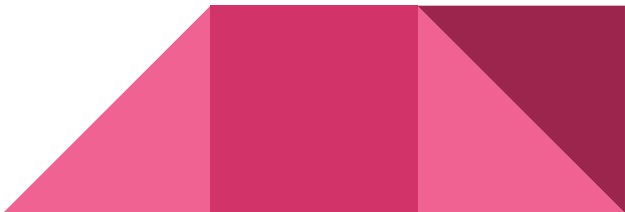
## **Main Memory (DRAM) (~200 – 600 ns):**

- Much larger (GBs), but significantly slower than caches.
  - Accessed when data is not found in any cache.
- 

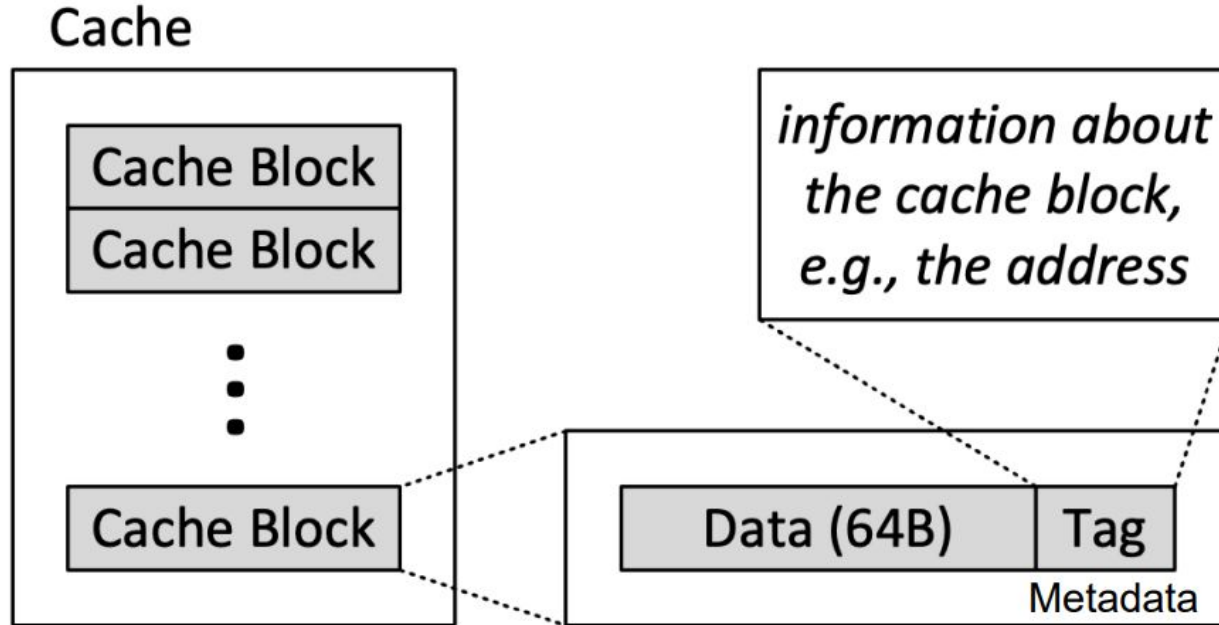
# Caching

- **Caching** stores frequently accessed data in small, fast memory (cache) close to the CPU.
- **Improves performance** by reducing average memory access time.
- **Used at multiple levels** (L1, L2, L3) in the memory hierarchy.

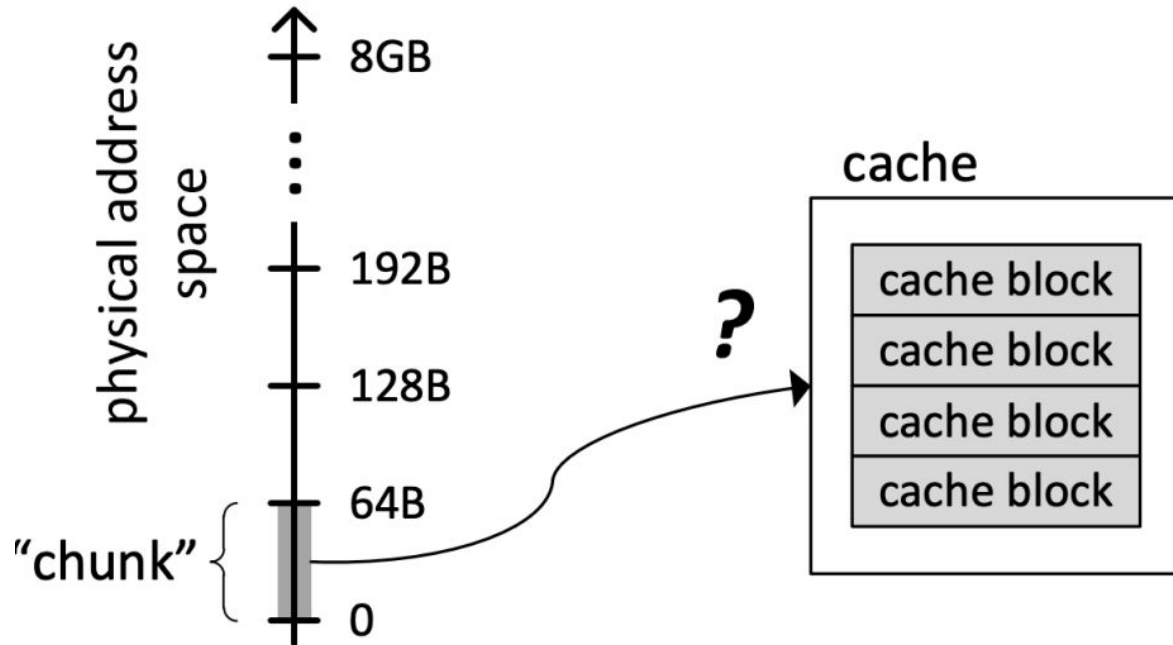
## Key Terms

- **Cache Hit:** The requested data is found in the cache.
  - **Cache Miss:** The requested data is not in the cache and must be fetched from lower memory.
  - **Hit Rate:** Percentage of memory accesses that result in a cache hit.
  - **Miss Penalty:** Time taken to fetch data from lower memory after a miss.
  - **Block (or Line):** The smallest unit of data transferred between memory and cache.
  - **Valid Bit:** Indicates whether the given line holds valid data or not.
  - **Dirty Bit:** Indicates whether the data in the cache line is modified.
- 

# Cache Organization



## Cache Organization(2)

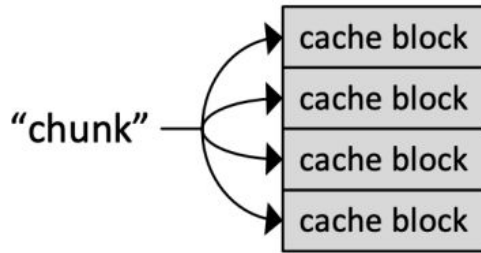


Credits: Safari, ETHZ,

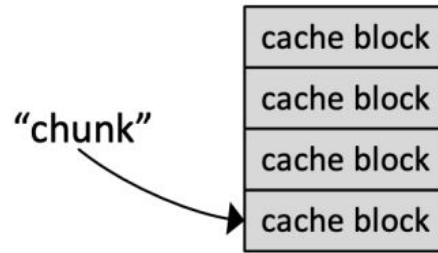
<https://safari.ethz.ch/digitaltechnik/spring2023/lib/exe/fetch.php?media=onur-ddca-2023-lecture22-memory-hierarchy-and-caches-afterlecture.pdf>

# Cache Organization(3)

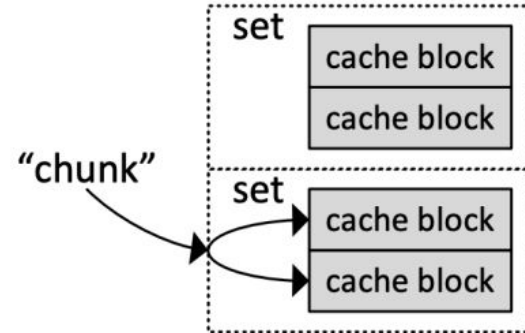
**fully-associative**



**direct-mapped**



**set-associative**



# Accessing a cache

- Say the size of cache is 32kB.
- Size of a cache line is 64B.
- Number of cache lines is 512.
- Number of bits required to access all the lines is 9 bits
- Say the system supports 32 bit addressing.




# Directly Mapped Cache

In a **direct-mapped cache**, each memory block maps to **exactly one cache line** based on a simple index derived from the memory address. It's fast and easy to implement.

## Advantages:

- **Simple design** with low hardware complexity and fast access time.
- **Cost-effective** and power-efficient—ideal for embedded and small systems.

## Disadvantages:

- **High conflict misses**, especially when multiple frequently-used addresses map to the same line.
  - **Poor performance in certain access patterns**, such as arrays that stride through similarly indexed locations.
- 



# Accessing a Directly mapped cache

- Say the size of cache is 32kB.
- Size of a cache line is 64B.
- Number of cache lines is 512.
- Number of bits required to access all the lines is 9 bits
- The remaining  $32 - 9(\text{index}) - 6(\text{block offset}) = 17$  bits will serve as **tag**.
- Say the system supports 32 bit addressing.
- All the addresses whose last 9 bits are same will be mapped to same location in cache.



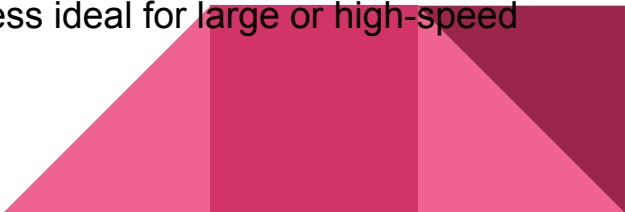
# Fully Associative Cache

In a **fully associative cache**, any memory block can be placed in any cache line, allowing maximum flexibility and reducing conflict misses.

## Advantages:

- **Very low conflict misses** since blocks can go anywhere in the cache.
- **Best cache utilization**, especially for irregular or frequently overlapping access patterns.

## Disadvantages:

- **Complex hardware** needed to compare tags of all cache lines in parallel.
  - **Slower and more power-hungry** than simpler designs, making it less ideal for large or high-speed caches.
- 

# Accessing a fully associative cache

- Say the size of cache is 32kB.
- Size of a cache line is 64B.
- Number of cache lines is 512.
- In this case the entire address will act as tag.
- All the addresses will be looked up in parallel, hence hardware demanding.
- No arrangement order, have to look for the address in the entire cache.
- This is power consuming but VERY flexible.



# Set Associative Cache

In a **set-associative cache**, each memory block maps to a **specific set**, but **can be placed in any line within that set** (e.g., 2-way, 4-way, 8-way). This provides a **balance between speed and flexibility** by reducing conflict misses while keeping lookup hardware manageable.

## Advantages:

- **Reduces conflict misses** compared to direct-mapped caches by allowing multiple blocks per set.
- **More efficient than fully associative** in terms of speed and power, especially for large caches.

## Disadvantages:

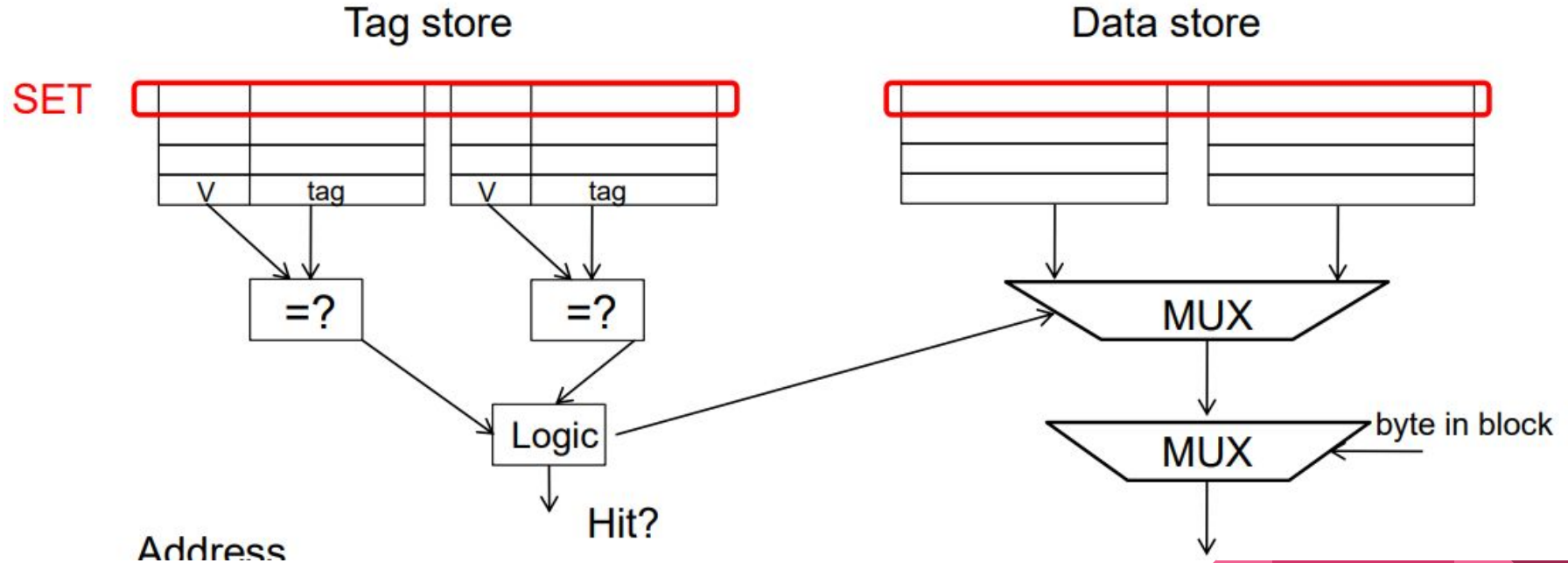
- **More complex hardware** than direct-mapped—requires tag comparisons within all ways of a set.
- **Slightly higher latency and power use** than direct-mapped due to multiple tag comparisons per access.

# Accessing a 4-way set associative cache

- Say the size of cache is 32kB.
- Size of a cache line is 64B.
- Number of cache lines is 512.
- 4 way set associative cache means each set will contain 4 lines.
- Number of sets will be  $512/4 = 128$ .
- Number of bits required to access 128 sets will be 7.
- After accessing a set, tag comparison will run in parallel for all 4 lines.
- Number of tag bits will be  $32 - 7(\text{set index}) - 6(\text{block offset}) = 19$ .



# Accessing a 4-way set associative cache



Credits: Safari, ETHZ,

<https://safari.ethz.ch/digitaltechnik/spring2023/lib/exe/fetch.php?media=onur-ddca-2023-lecture22-memory-hierarchy-and-caches-afterlecture.pdf>

# Cache replacement policies

- **Least Recently Used:** Replaces the **cache line that hasn't been accessed for the longest time**. Tracks access history to approximate temporal locality. Common but **requires more logic and storage** to implement precisely in hardware.
- **First-In First-Out:** Evicts the **oldest block in the set**, regardless of how often or recently it's been used. Simple to implement (just a queue), but **doesn't consider access frequency** or recency.
- **Least Frequently Used:** Replaces the **line with the lowest access count** over time. Prioritizes lines with high reuse, but **requires counters** and may not adapt well to changing access patterns.
- **Random Replacement:** Chooses a **random line** in the set to evict. Extremely simple and low-overhead, often used in highly associative caches. Performance is surprisingly good in many practical cases.



# Cache Write Strategies

Write Through

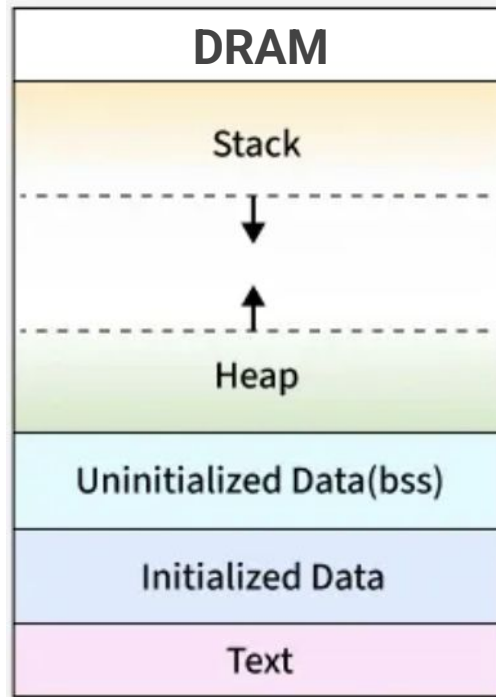




# Virtual Memory

```
int* ptr = malloc(sizeof(int));
```

- The pointer `ptr` holds a virtual address, not a direct physical address in DRAM.
- This address is used in your program, and the MMU translates it to a physical address when accessing memory.
- Even though the actual memory is allocated in DRAM, the program works entirely with virtual addresses.



# Benefits of Virtual Memory

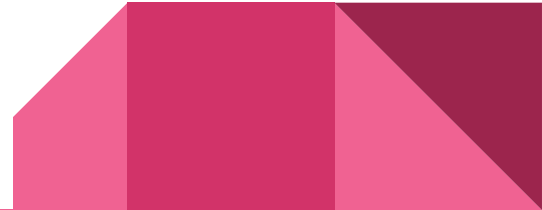
Balance the  
shortage of  
physical  
space

Inexpensive  
virtual  
memory

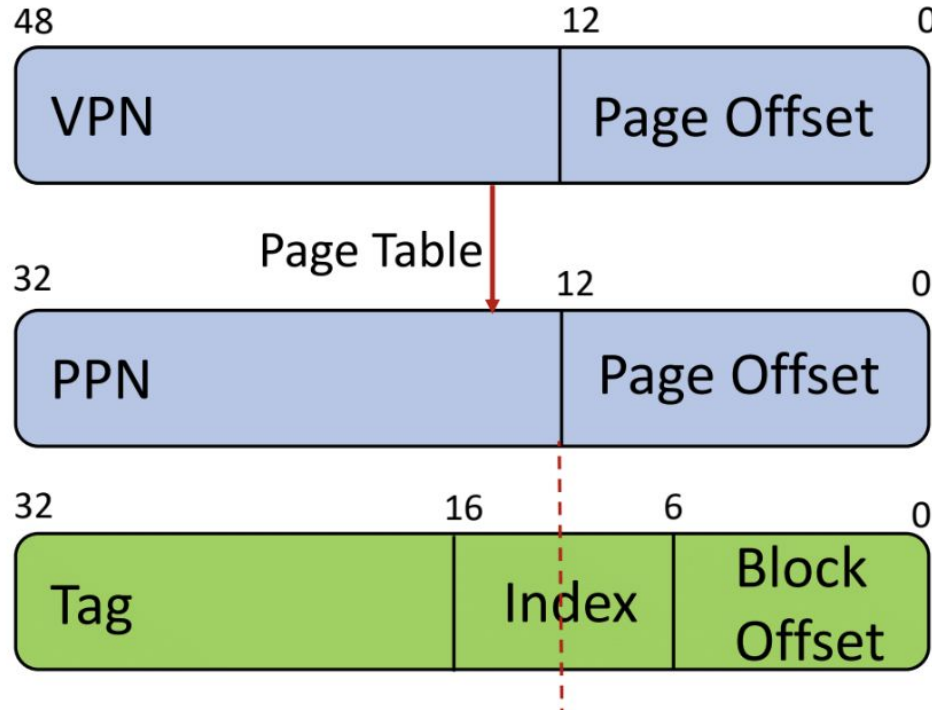
Supports  
multitasking  
and  
collaboration

Avoid  
memory  
fragmentation

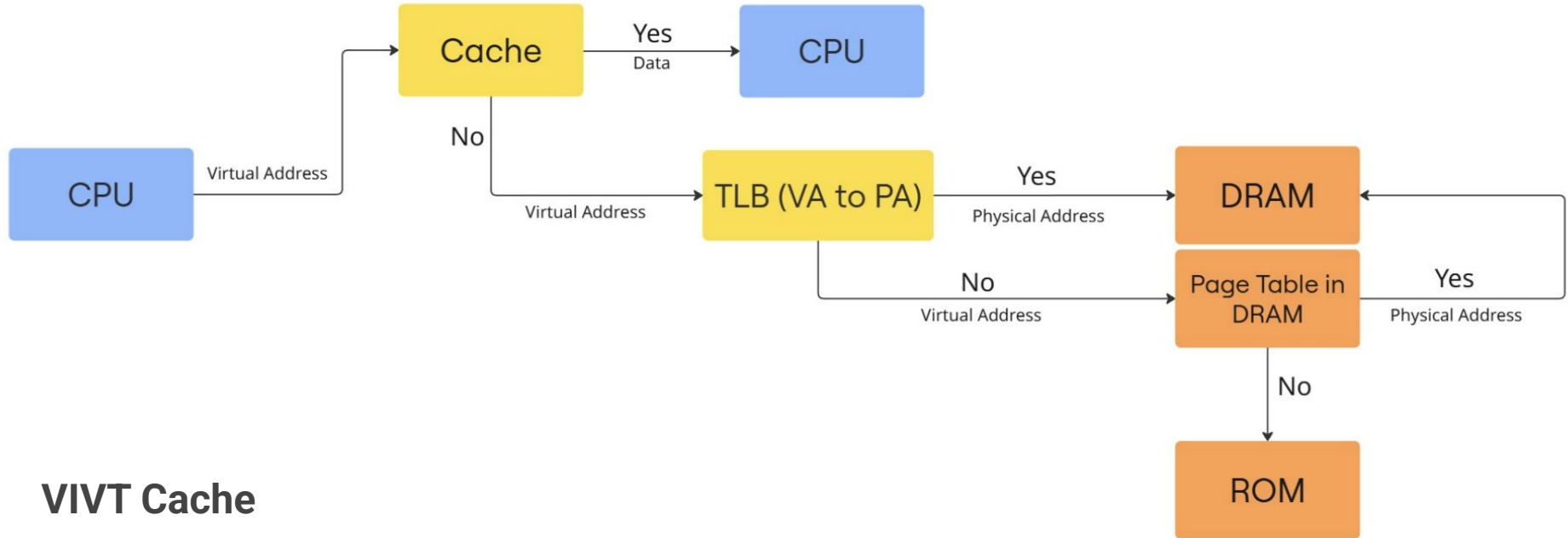
Enhance data  
security



# Virtual to Physical Address translation - by MMU

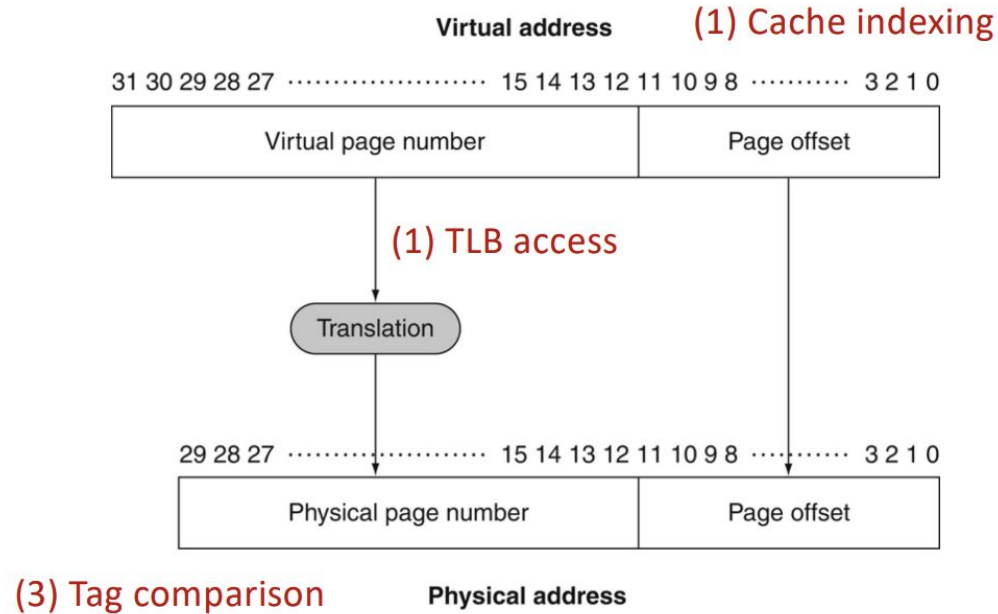


# Dataflow while executing a program



**VIVT Cache**

# Types of Caches - VIVT, PIPT, PIVT, VIPT

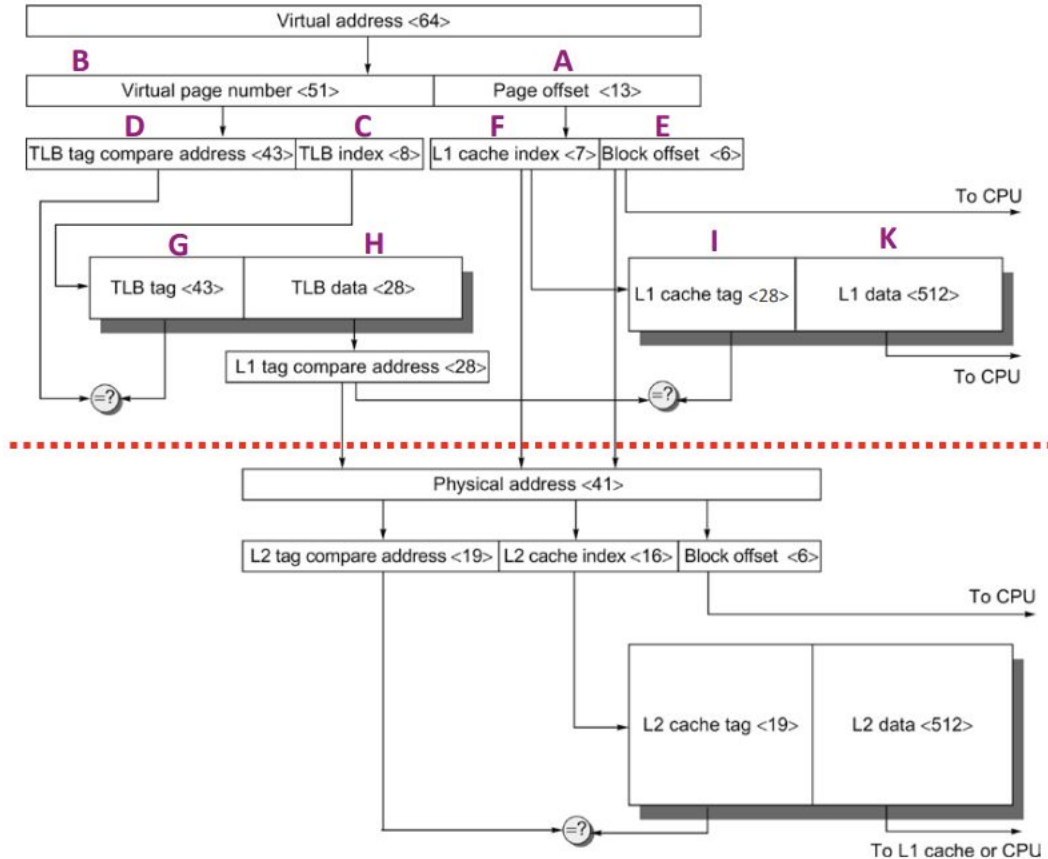


## Popular Addressing Strategy

Level 1 is VIPT

Level 2-3 are PIPT

# VIPT cache - Example



## Specs

- Address spaces: 64-bit virtual, 41-bit physical
- 8 KB page size
- 8 KB direct-mapped L1 cache
- 64 Byte blocks (lines)
- 256 entry direct-mapped TLB

## Explaining A-K (# bits)

8 KB page

- 13 bit page offset (**A**)
- 51 bits virtual page # (**B**)

256-entry direct-mapped TLB

- 8 bits to index the TLB (C)
- TLB tag is 51 *minus* 8 = 43 (D)

## Tagging the physical pages

- $41 - 13 = 28$  (H,I)

## L1 Cache

- 64-byte line size (6 bits) **E**
- 128 sets (blocks)  $\rightarrow$  7 bits **F**



# Special Instructions - For CPU info

1. `sudo dmidecode -t cache`
2. `getconf -a | grep CACHE`
3. `cat /proc / cpuinfo`
4. `lscpu`
5. Wiki chip: <https://en.wikichip.org/wiki/WikiChip>
6. Perf tool: <https://perfwiki.github.io/main/>
7. `sudoperf stat-e cache-misses`



# Special Instructions - For measuring memory latencies

1. **RDTSC** – Reads CPU cycle counter.
2. **CLFLUSH** – Flushes a cache line.
3. **LFENCE** – Orders memory loads.
4. **MFENCE** – Orders all memory ops.
5. **SFENCE** – Orders memory stores.





# How to use RDTSC and CLFLUSH?

```
unsigned long long rdtsc() {  
    unsigned long long a, d;  
    asm volatile ("mfence");  
    asm volatile ("rdtsc" : "=a" (a), "=d" (d));  
    a = (d<<32) | a;  
    asm volatile ("mfence");  
    return a;  
}
```

```
void flush(void* p) {  
    asm volatile ("clflush 0(%0)\n"  
        :  
        : "c" (p)  
        : "rax");  
}
```

# Assignment Q2 Approach

1. **Allocate memory**
2. **For cache hit**
  - Access the data to bring it into cache.
  - Measure access time.
3. **For cache miss**
  - Flush the cache line using `clflush`.
  - Access the data and measure time.
4. **Repeat multiple times** (eg, 10000 iterations) and average the results

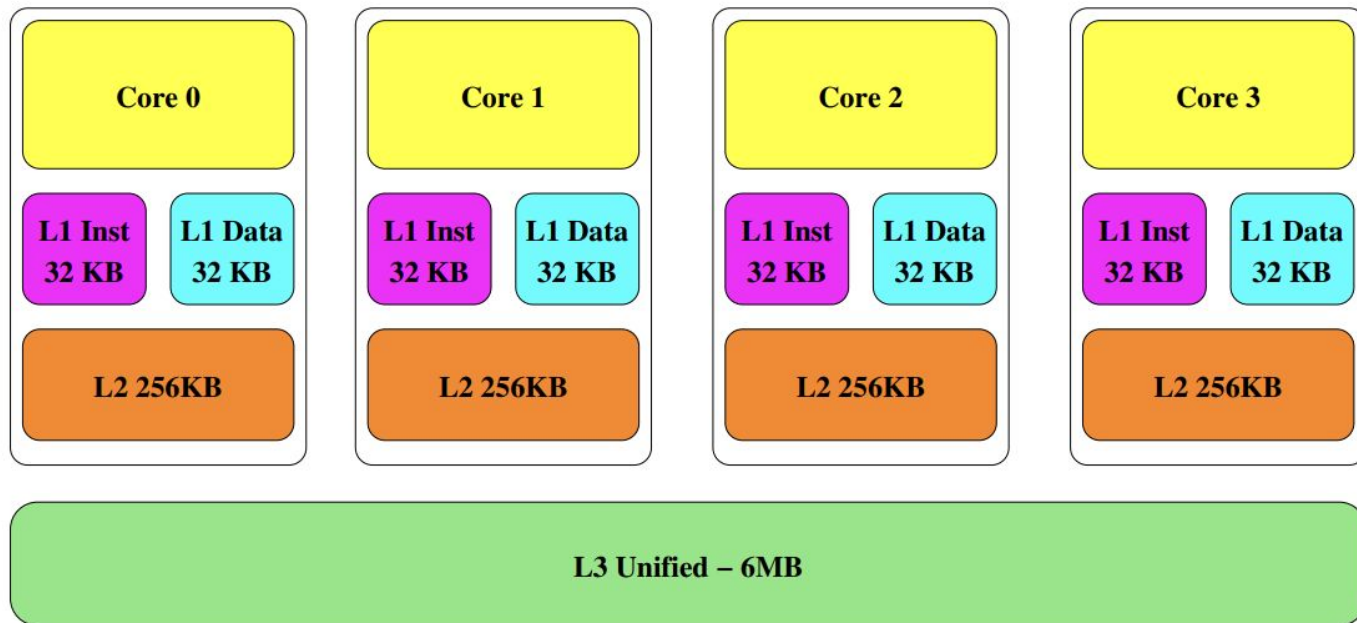


# Flush & Reload attack

By


- Praveen Prajapat
- Anubhav Bhatla

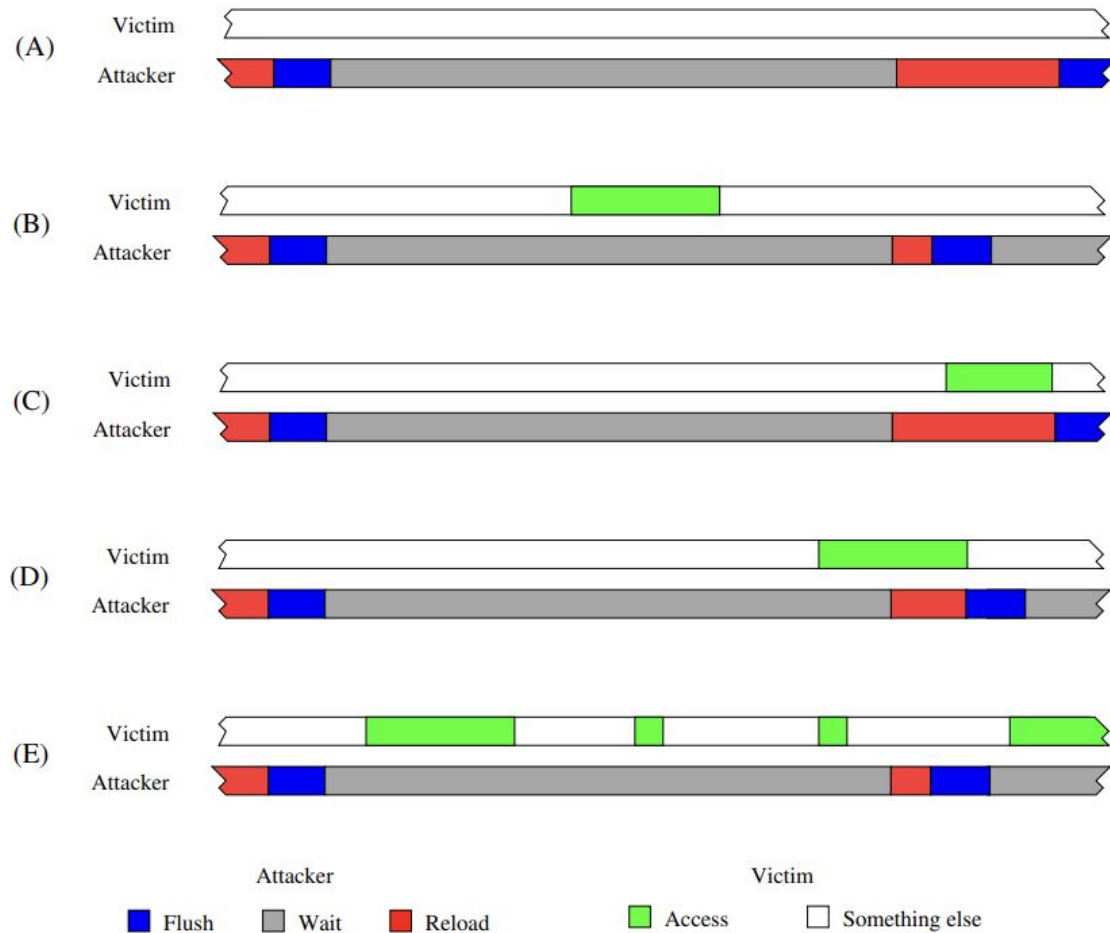
# General Cache and core structure



# Square and Multiply - Exponentiation of $x = b^e \bmod m$

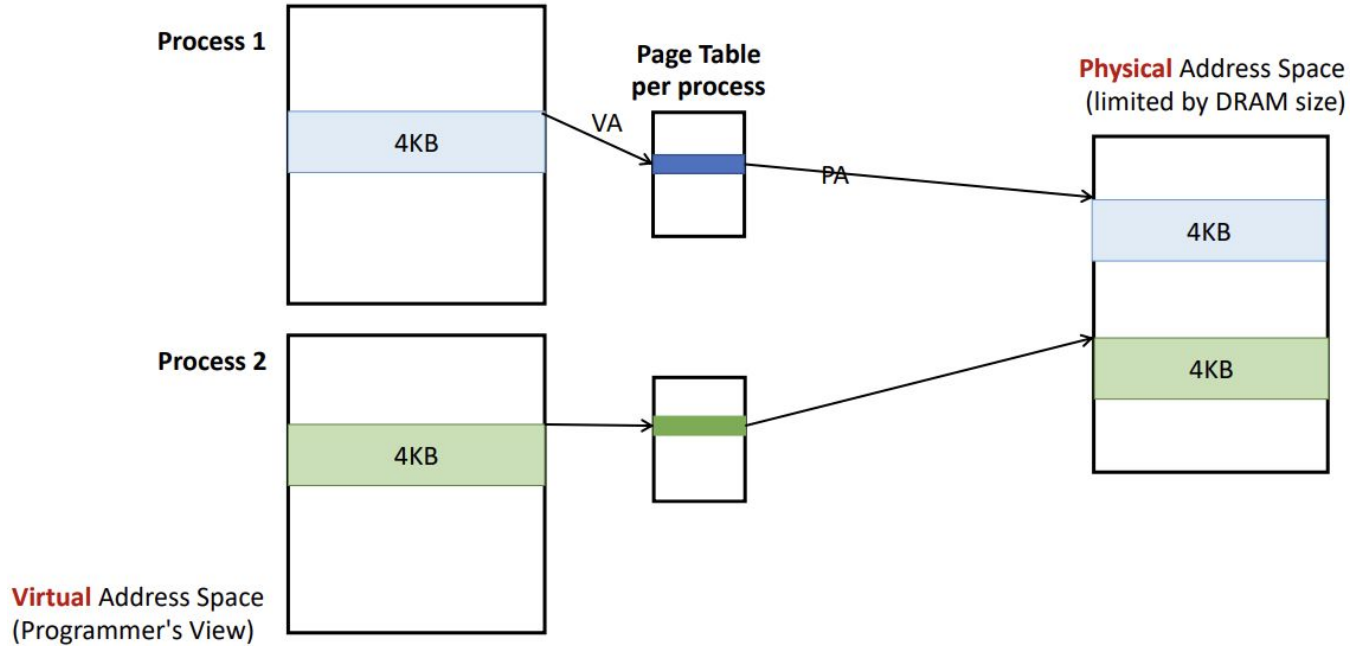
```
1 function exponent( $b, e, m$ )
2 begin
3    $x \leftarrow 1$ 
4   for  $i \leftarrow |e| - 1$  downto 0 do
5      $x \leftarrow x^2$ 
6      $x \leftarrow x \bmod m$ 
7     if ( $e_i = 1$ ) then
8        $x \leftarrow xb$ 
9        $x \leftarrow x \bmod m$ 
10    endif
11  done
12  return  $x$ 
13 end
```



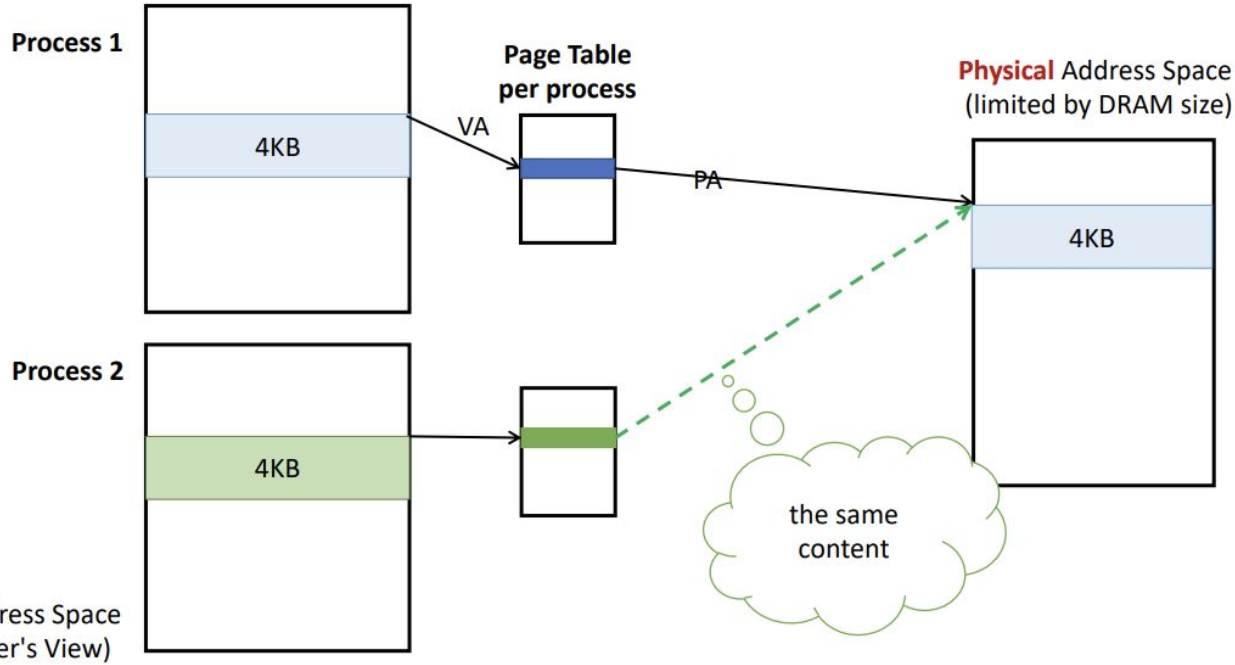


Possible  
access  
patterns

# Shared Memory - Page mapping

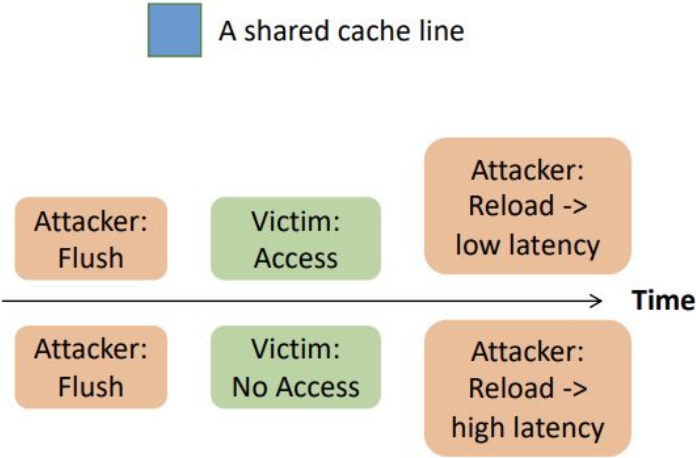
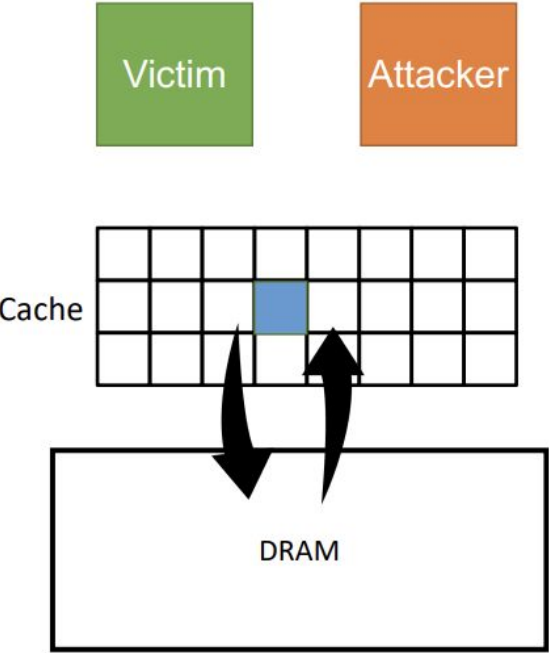


# Shared Memory - Page sharing





# Attack semantics



# Attacker pseudocode

```
shared_address = get_shared_memory_address()
```

```
while True:
```

```
    flush(shared_address)           // Step 1: Flush from cache using clflush
    wait_short_interval()           // Step 2: Wait for victim to access shared address
    start = timestamp()              // Step 3: Start timing
    read_memory(shared_address)      // Step 4: Access the address again
    end = timestamp()                // Step 5: End timing
```

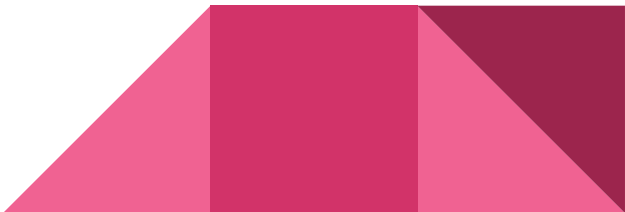
```
access_time = end - start
```

```
if access_time < CACHE_HIT_THRESHOLD:
```

```
    print("Victim likely accessed the address")
```

```
else:
```

```
    print("Victim did NOT access the address")
```



# Cache occupancy based attack

By

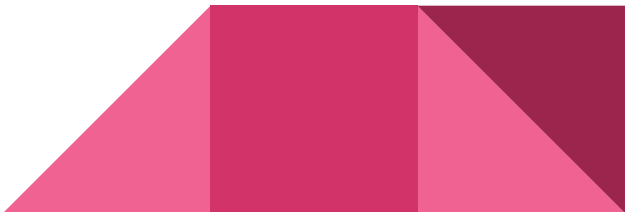
- Praveen Prajapat
- Anubhav Bhatla

# Attack semantics

**The sender encodes bits by -**

- Accessing large memory regions that fully occupy the LLC to send a 1 .
- Remaining idle to send a 0.

**The receiver decodes bits by measuring how many memory operations it can perform within a time window -**

- If the LLC is congested (due to sender activity), memory access is slower → interpret as 1 .
  - If the LLC is free, access is faster → interpret as 0 .
- 

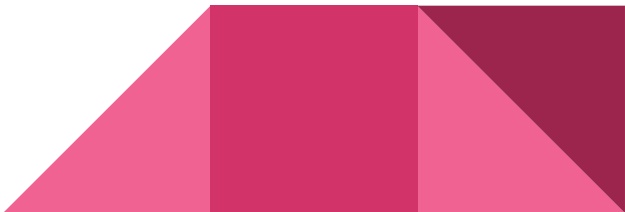
# Pseudocode

```
int Trace[T*1000];
loop {
    counter = 0;
    t_begin = time();

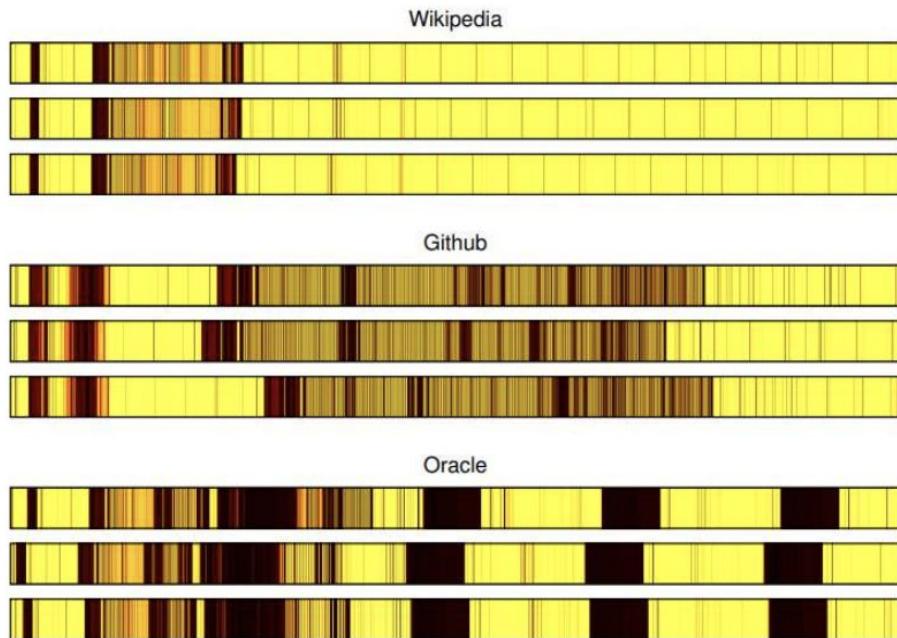
    do {
        // count iterations
        counter++;
        // memory accesses
        for (i=0; i<size; i++) {
            tmp = buffer[i * 64]
        }
    } while(time()-t_begin < P);

    Trace[t_begin] = counter;
}
```

- Initializes a timer and sets `counter = 0` to begin measuring CPU occupancy.
- Continuously accesses memory (`buffer[i * 64]`) to generate load.
- Counts how many such memory sweeps fit in a fixed time window `P`.
- Saves the iteration count to `Trace[]` as an indicator of system load or interference



# Occupancy based website fingerprinting - Memory gram



X axis -> time

More red -> more cache usage

## What we can infer ?

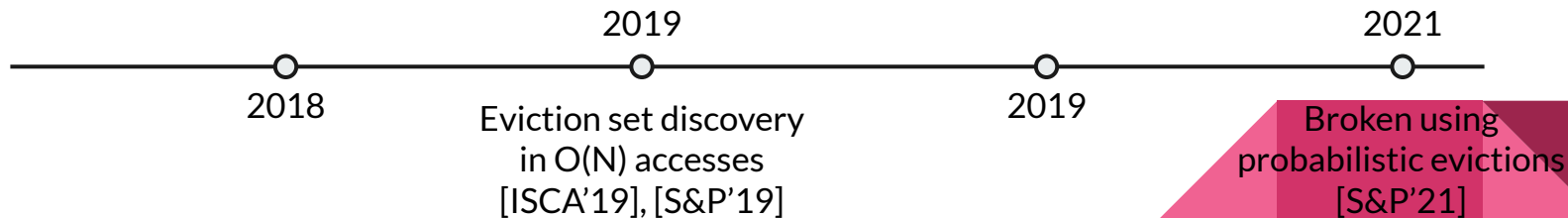
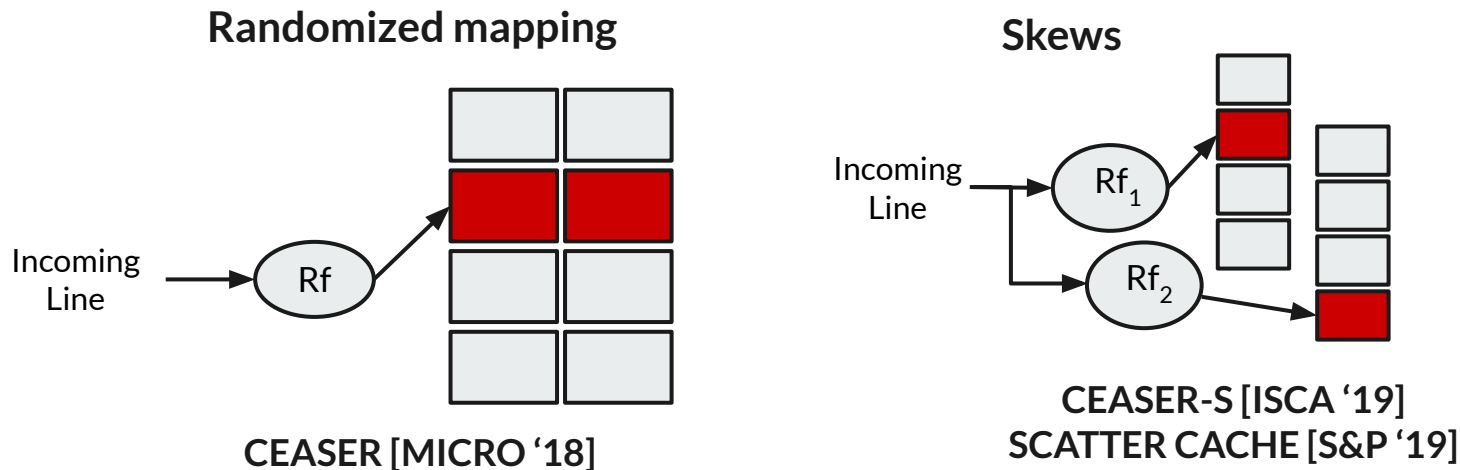
Privacy can be breached by knowing what websites someone is using

# Mitigations - Randomization & Partitioning

By

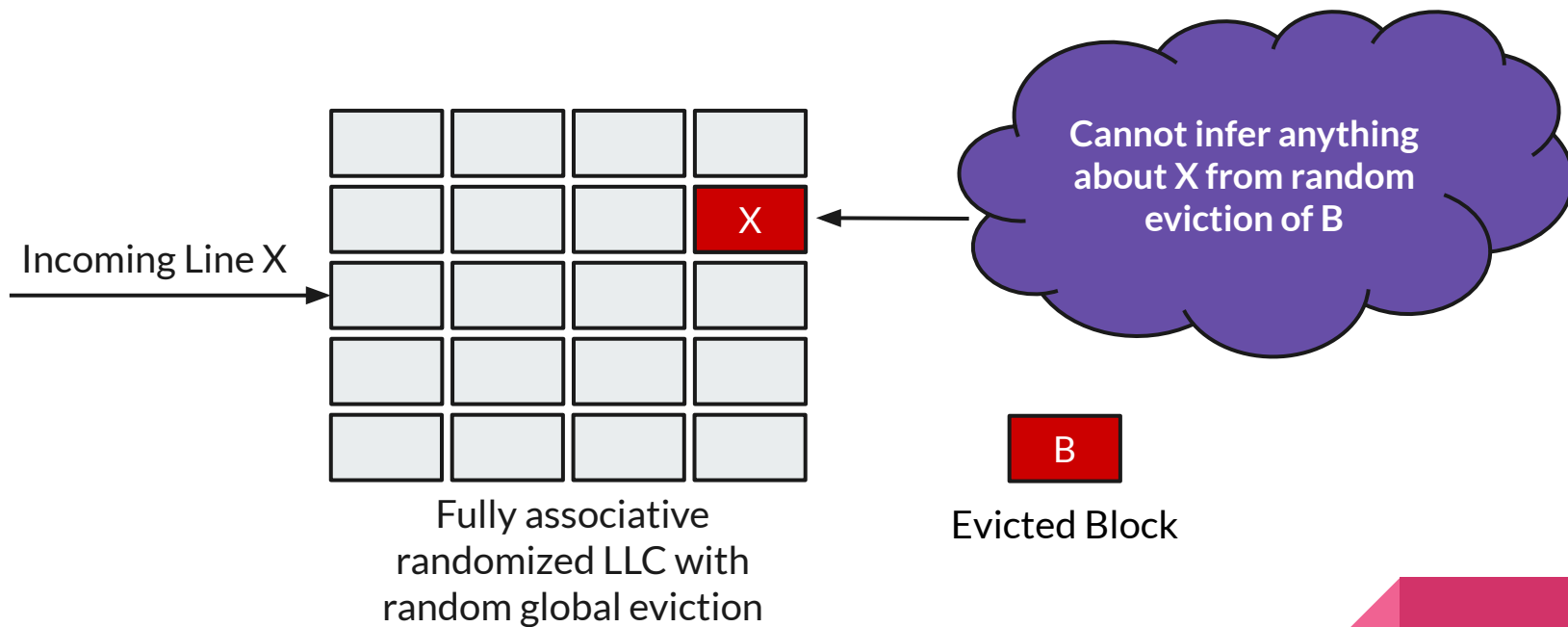
- Anubhav Bhatla

# Cache randomization



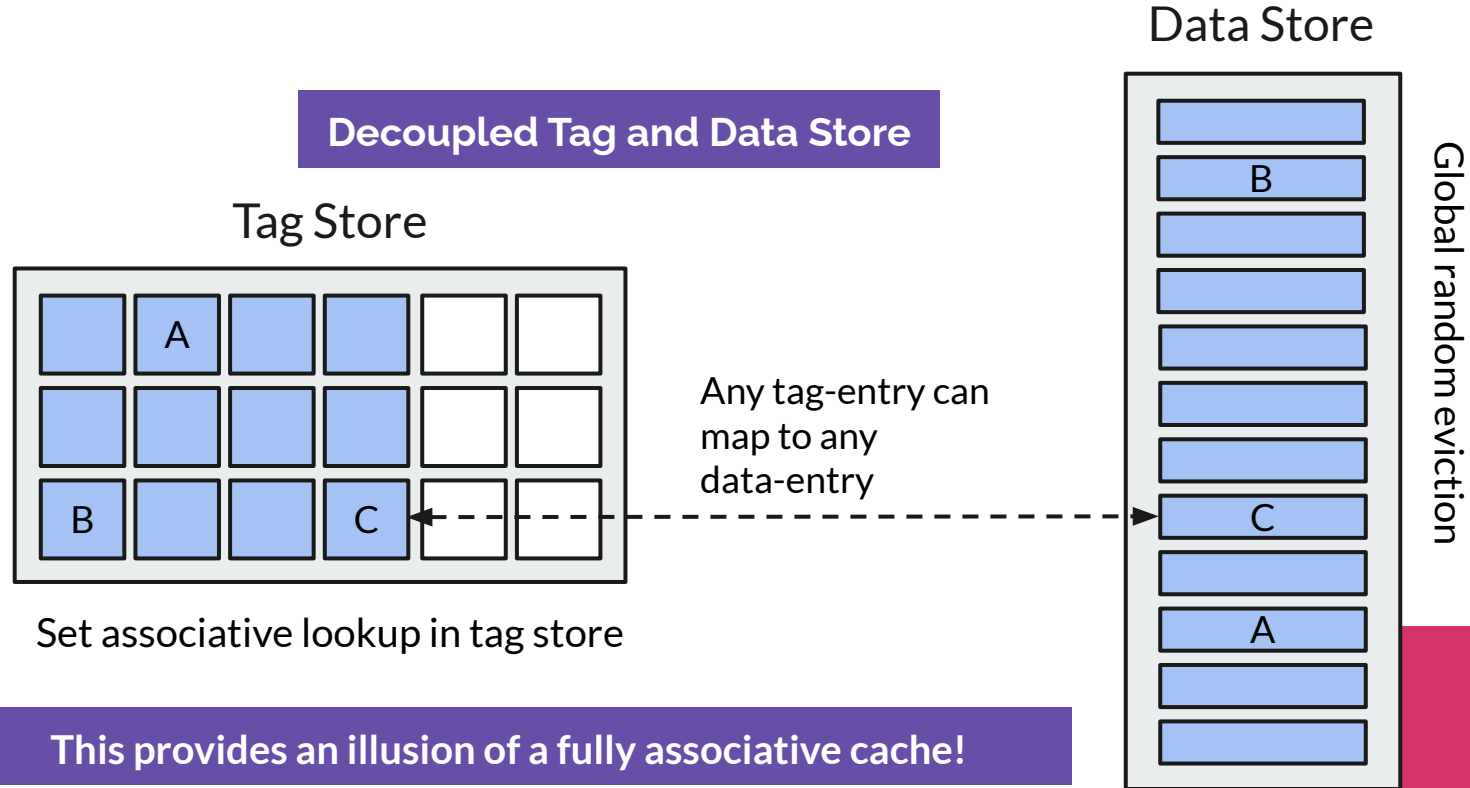


# Ideal Mitigation: Fully Associative Randomized Cache

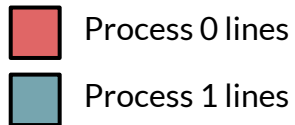


No set-associative evictions; makes it harder for conflict-based attacks

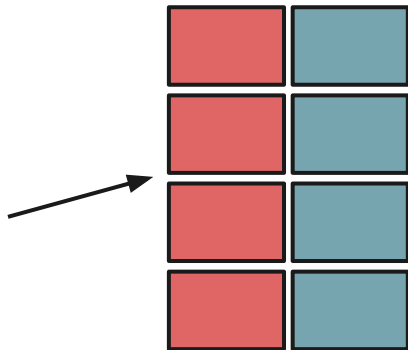
# MIRAGE [USENIX Security '20]



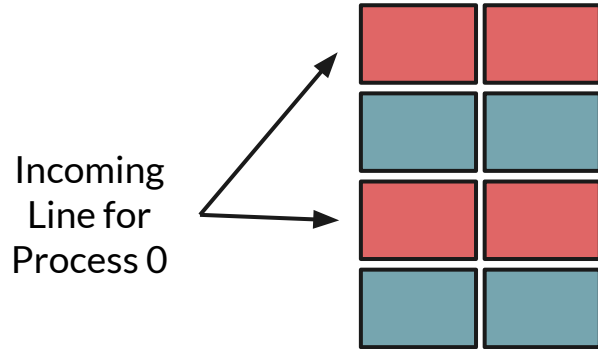
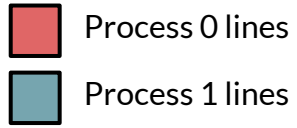
# Way partitioning



Incoming  
Line for  
Process 0



# Set partitioning



# The Maya Cache

By

- Anubhav Bhatla

# Motivation

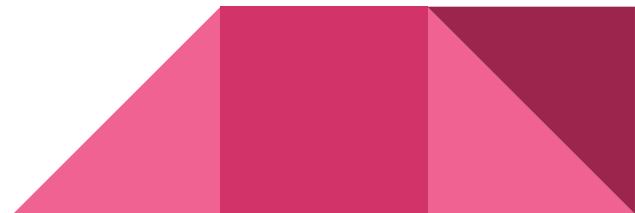
MIRAGE Tradeoffs:

(+) Provides complete security against conflict-based attacks

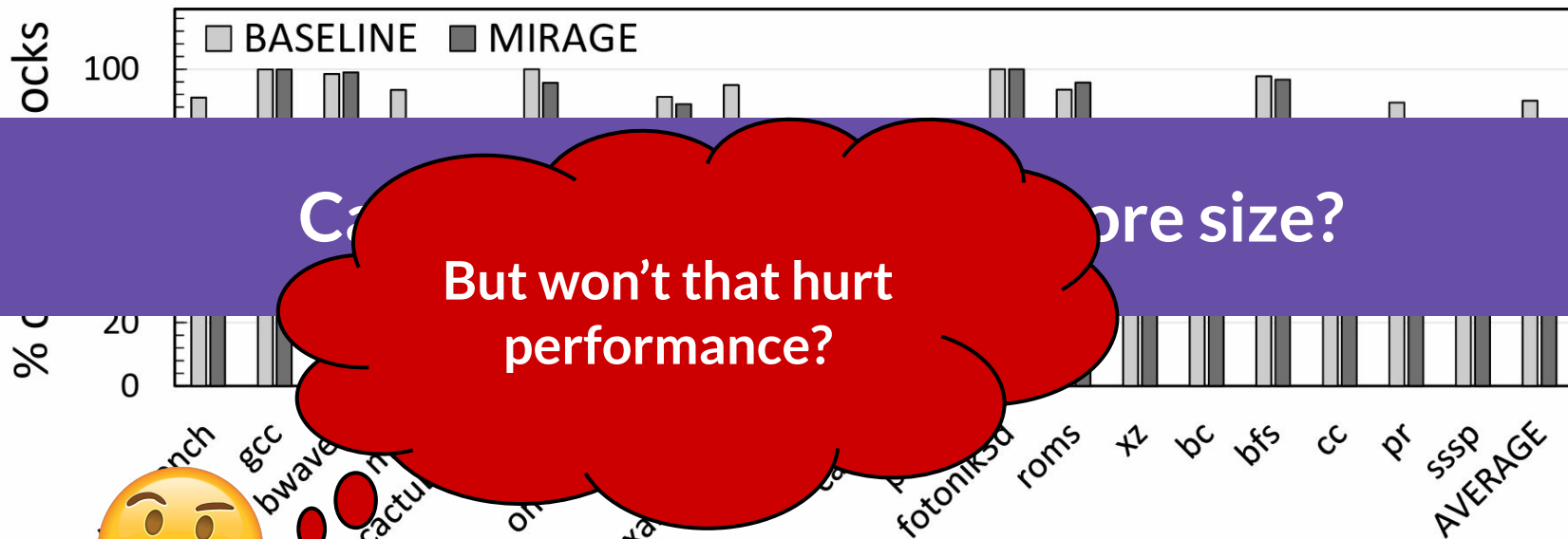
(+) Performance comparable to a non-secure baseline

(-) High storage and area overheads (>20%)

(-) High power overheads (19%)

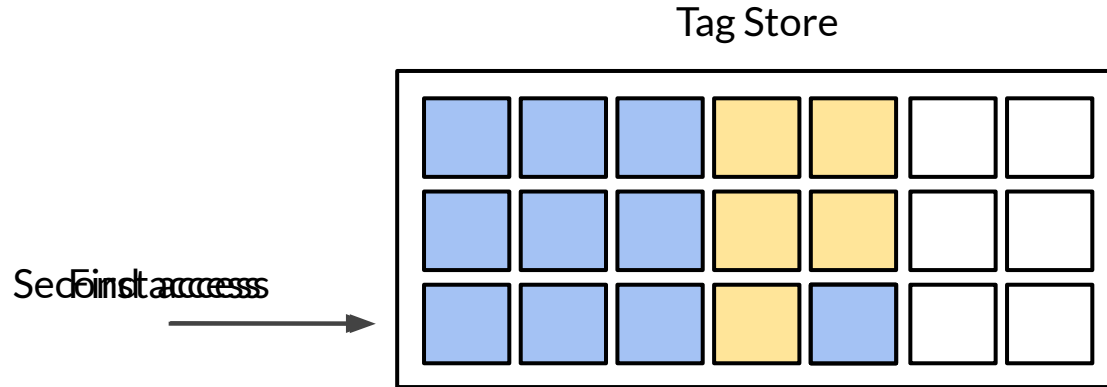


# Motivation



>80% dead blocks in the LLC

# Tracking reuse [MICRO '13]



Tag Hit, Data Miss  $\Rightarrow$  Data is brought in

Data is stored only when reuse is detected

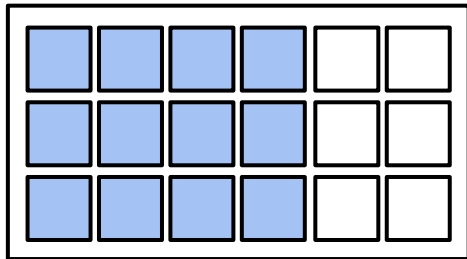


# Smaller data store and reuse

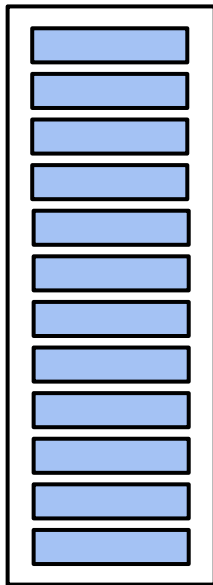
## MIRAGE

>20% storage overhead

Tag Store



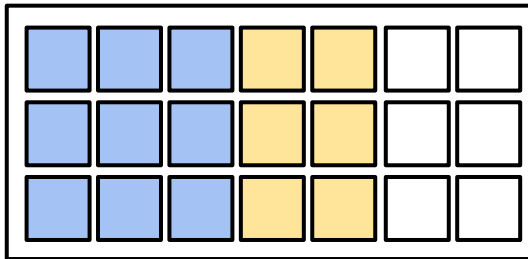
Data Store



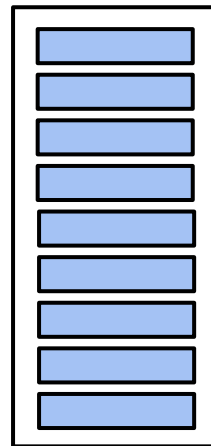
## MAYA

2% storage savings

Tag Store



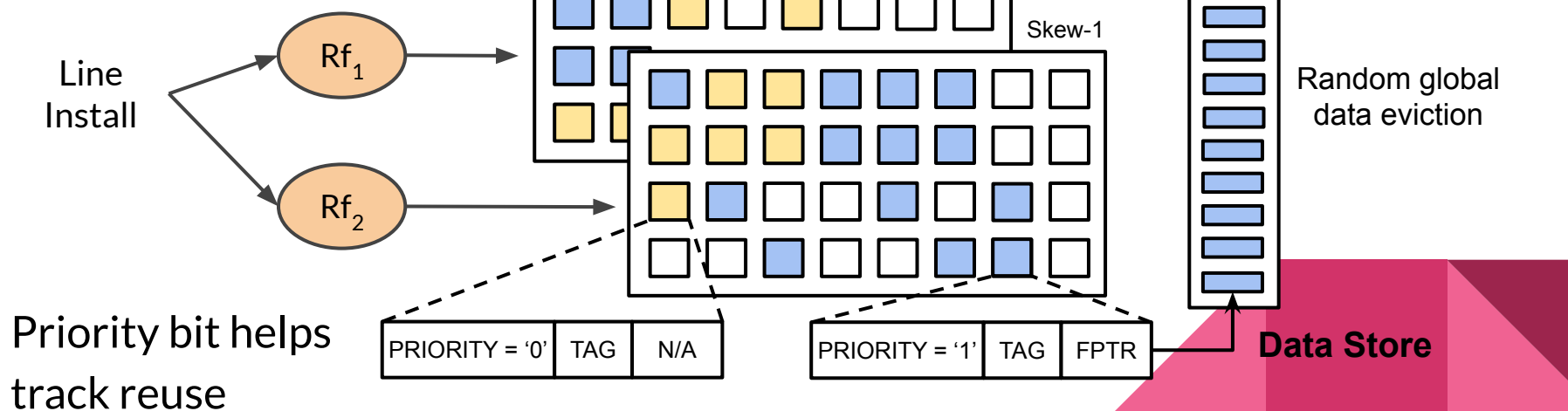
Data Store



Data entry requires 8X the number of bits compared to a tag entry

# Maya cache design

More tag store entries but fewer data store entries



# Maya tradeoffs

Performance	~2% improvement
Storage	2% savings
Area	28% savings
Read Energy	15% savings
Write Energy	11% savings
Leakage Power	5% savings
Security?	Yes :)

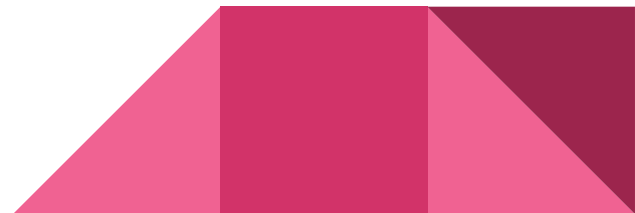


# Give it a read!



Maya

Pre-print + Artifact



**Thank you !**

