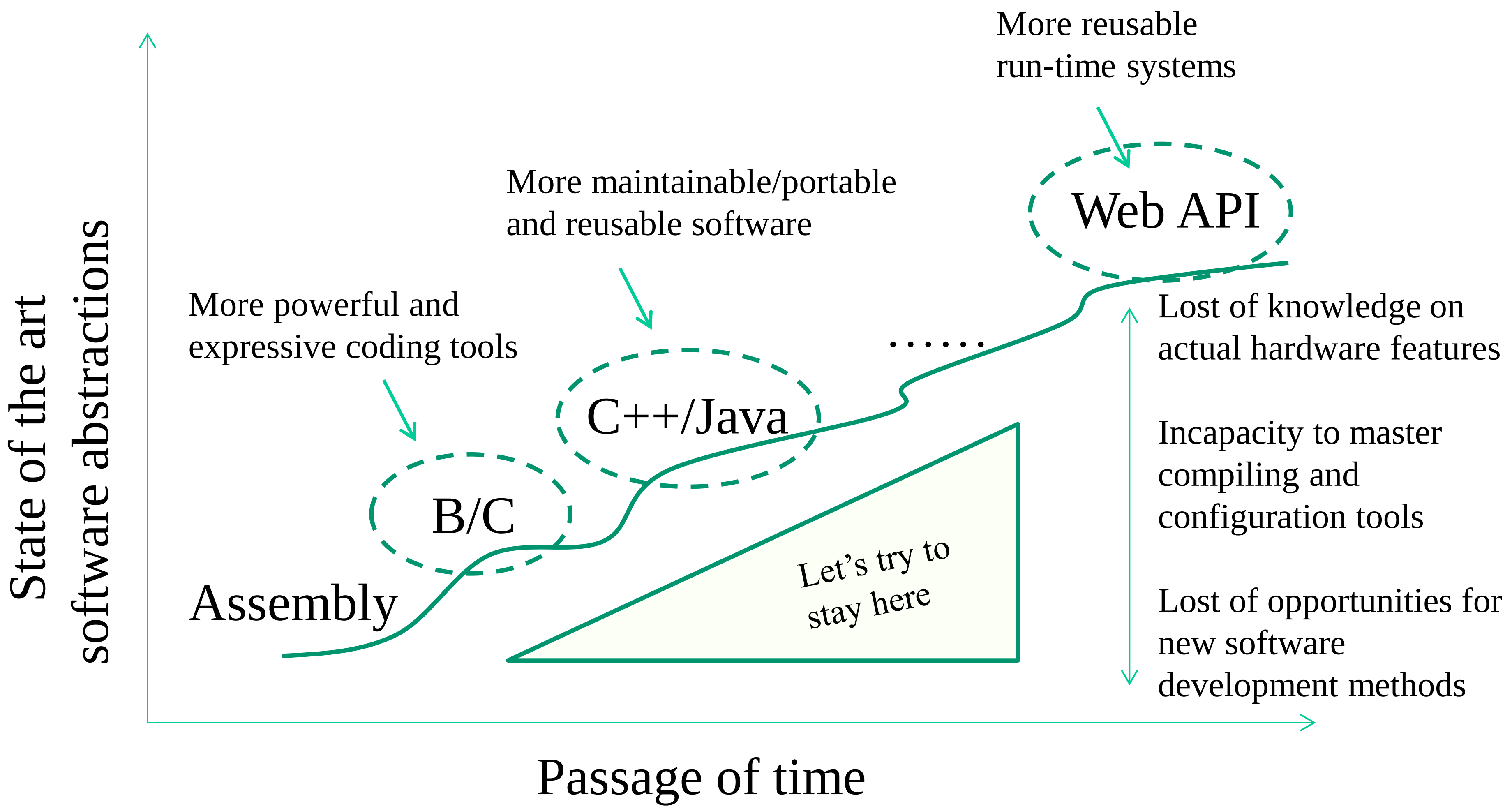
MS degree in Computer Engineering
University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Hardware review

- Pipelining and superscalar processors
- Speculative hardware
- Multi-processors and multi-cores
- Physical memory organization
- Memory coherency and consistency
- Hardware synchronization support
- Linearizability and thread coordination schemes
- Vectorization

The need for holistic programming



More reusable run-time systems
More maintainable/portable and reusable software
More powerful and expressive coding tools
Web API
C++/Java
B/C
Assembly
. . . . . .
Let's try to stay here
Lost of knowledge on actual hardware features
Incapacity to master compiling and configuration tools
Lost of opportunities for new software development methods
State of the art software abstractions
Passage of time

The missing piece of information

- The actual state of a program is not the “puzzle” of the states of its individual software components
- Each component sees and updates a state portion that is not trivially reflected into the view by other components
- A component even does not know whether hardware state beyond the ISA-exposed one can be affected by its execution
- In real systems things may occur in different ways because of
  - ✓ Compiler decisions
  - ✓ Hardware run-time decisions
  - ✓ Availability (vs absence) of hardware features
- More abstractly, there is a combination of software and hardware non-determinism
- Ideally programmers should know of all this to produce **<u>correct, secure and efficient software</u>**

The common fallback … and our path

- Simply exploit what someone already did (libraries, run-time environments, algorithmic and coding approaches ….)
- But you should know that this still does not guarantee you are writing (or running) a program the most efficient (or even correct) way
- …. in the end knowing the hardware and the under-the-hood layers we actually work with provides us with possibilities for better achievements in software development
- Nowadays hardware is multi-core, which is characterize by some major aspect that need to be reflected on software programming
- We could ideally study those aspects, and proper software design approaches at different levels – we will take the OS kernel one as our core reference

A very trivial example: Lamport’s Bakery

```
var choosing: array[1,n] of boolean;
     number: array[1,n] of int;

repeat {
    choosing[i] := TRUE;
    number [i] := <max in array number[] + 1>;
    choosing[i] := FALSE;
    for j = 1 to n do {
        while choosing[j] do no-op;
        while number[j] ≠0 and (number [j],j)< (number [i],i) do no-op;
    }
    <critical region>;
    number[i] := 0;

}until FALSE
```

Typically no machine (unless single-core) guarantees **globally consistent** view of this update sequence

Entering a few details

- The machine model we have been used to think of is the von Newman’s one
  - ✓ Single CPU abstraction
  - ✓ Single memory abstraction
  - ✓ Single control flow abstraction: fetch-execute-store
  - ✓ Time separated state transitions in the hardware: no more than one in-flight instruction at anytime
  - ✓ Defined memory image at the startup of any instruction
- The modern way of thinking architectures is instead not based on the flow of things as coded in a program, rather on the concept of **scheduling things** (e.g. usage of hardware components) to do something equivalent to that program flow
- Hopefully the schedule allows doing stuff in parallel
- …. what about programs naturally made up by multiple flows? – this is exactly an OS kernel!!

Types of scheduling

- In the hardware
  - ✓ Instruction executions within a single program flow
  - ✓ Instruction executions in parallel (speculative) program flows
  - ✓ <u>Propagation of values</u> within the overall memory (more generally hardware) system
- At software level
  - ✓ Definition of time frames for threads' execution on the hardware
  - ✓ Definition of time frames for activities' execution on the hardware
  - ✓ Software based synchronization supports (thread/task synchronization)

Parallelism

- Baseline hardware form – ILP (Instruction Level Parallelism):
  - ✓ The CPU is able to process 2 or more instructions of a same flow during the same cycle (<u>even vectorized or dynamically scheduled – or both</u>)
  - ✓ It can therefore deliver instruction commits at each individual cycle (even tough a single instruction can take several cycles to complete)
- Software reflected form (Thread Level Parallelism):
  - ✓ A program can be though as of the combination of multiple concurrent flows
  - ✓ Concurrency can boil down to actual wall-clock-time parallelism in multi-processing (or ILP) hardware systems

Baseline notion of computing speed

- It is typically related to the Gigahertz (GHz) rating of a processor
- However, we clearly know that this way of thinking is only partially correct
- There are instructions that can take long sequences of CPU-cycles just because of unpredictable factors
    - ✓ Hardware interactions
    - ✓ Asymmetries and data access patterns
- In the end we can generally think of categories of programs (or programs' blocks) that are more ore less importantly affected by the clock speed
    - ✓ CPU-bound programs
    - ✓ Memory-bound programs - that' why we need to know about how to deal with memory in modern systems!!

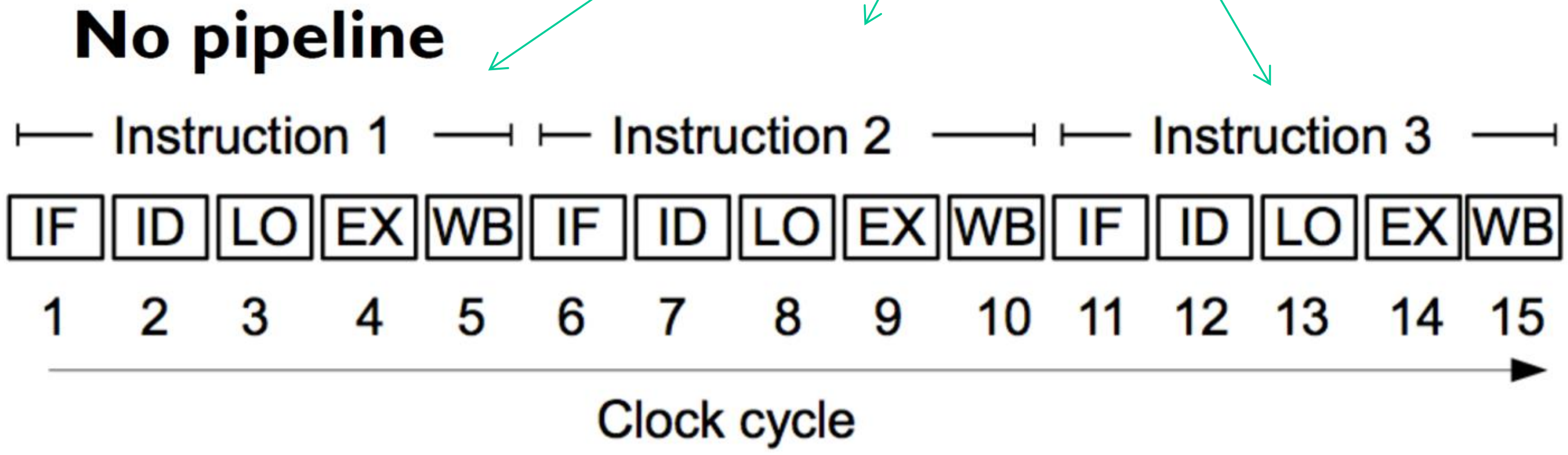
Overlapped processing: the pipeline

- The very baseline hardware form of overlapped processing is pipelining
- It is a **Scheduling+Parallelism** hardware-based technique
- Here we no longer have a clear temporal separation of the execution windows of different instructions (this is parallelism!!)
- What is sequenced within a program (I'm here referring to an actual executable) is not necessarily executed in that same sequence in the hardware (this is scheduling!!)
- However, causality needs to be preserved
- This is actually a data flow model (a source should be read based on the actual latest update along the instruction sequence)

Instruction stages

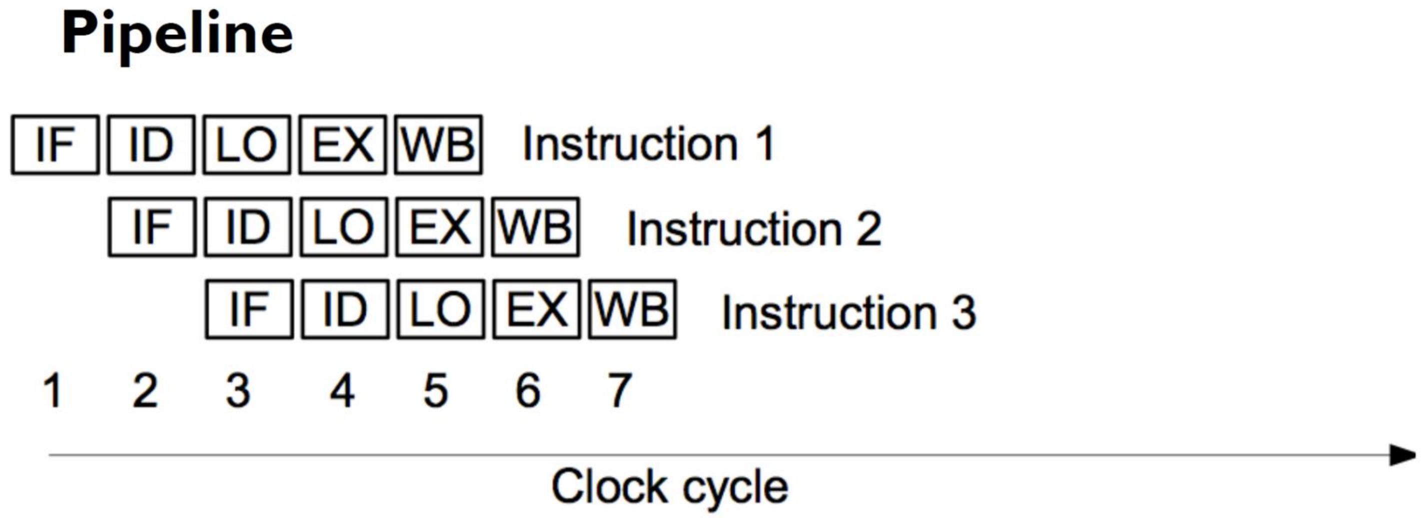
- IF – Instruction Fetch
- ID – Instruction Decode
- LO – Load Operands
- EX – Execute
- WB – Write Back

The different phases hopefully need to rely on different hardware components



Overlapping stages: the pipeline

- Each instruction uses 1/5 of the resources per cycle
- We can overlap the different phases
- We can therefore get speedup in the execution of a program, as compared to the non-pipeline version



Speedup analysis

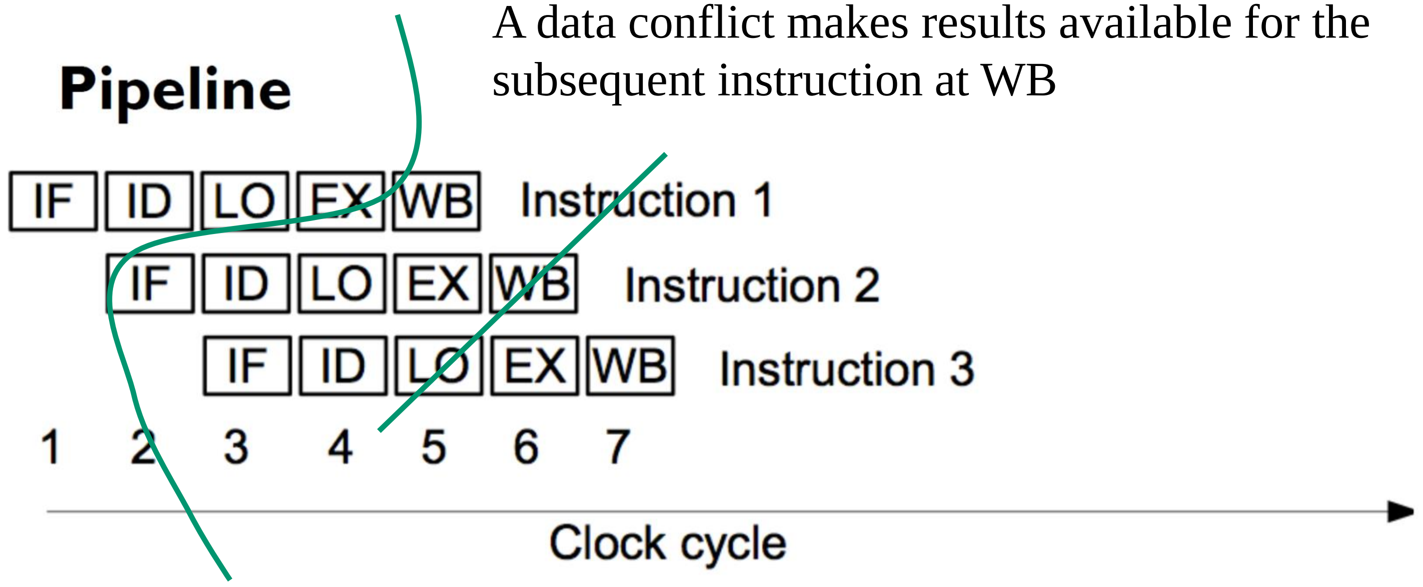
- Suppose we want to provide N outcomes (one per instruction) and we have L processing stages and clock cycle T
- With no pipelining we get (N x L x T) delay
- With pipelining we get the order of ([N+L] x T) delay
- The speedup is (NxL)/(N+L) so almost L (for large N)
- For N = 100 and L = 5 we get 4.76 speedup
- For L = 1 no speedup at all arises (obviously!!)
- So ideally the greater L the better the achievable performance
- But we do not live in an ideal world, in fact pipelined processors typically entail no more than the order of tens of stages (Pentium had 5 – i3/i5/i7 have 14 – ARM-11 has 8), although a few implement parts of an original instruction step

From the ideal to the real world: pipeline breaks

- Data dependencies
- Control dependencies

A conditional branch leads to identify the subsequent instruction at its EX stage

A data conflict makes results available for the subsequent instruction at WB


Pipeline
IF ID LO EX WB Instruction 1
IF ID LO EX WB Instruction 2
IF ID LO EX WB Instruction 3
1 2 3 4 5 6 7
Clock cycle

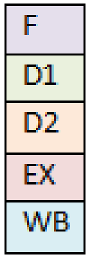
Handling the breaks

- Software stalls – compiler driven
- Software re-sequencing (or scheduling) – compiler driven
- Hardware propagation (up to what is possible)
- Hardware reschedule (**out-of-order pipeline - OOO**)
- Hardware supported hazards (for branches)

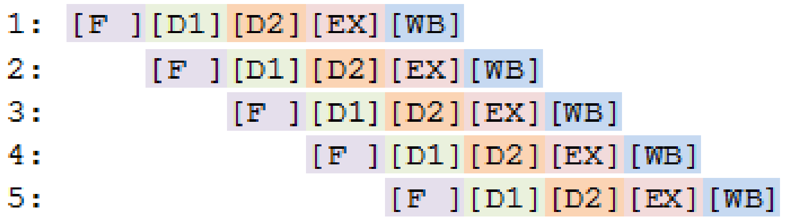
Ordering of the execution steps of the instructions is not based on how they touch ISA exposed hardware components (such as registers)

The Intel x86 pipeline

- In a broad analysis, Intel x86 processors did not change that much over time in terms of software exposed capabilities
- The 14 registers (AX, BX, .. etc) of the 8086 are still there on e.g. core-i7 processors (RAX, RBX .. etc)
- However, the 8086 was not pipelined, it processed instructions via [FETCH, DECODE, EXECUTE, RETIRE] steps in pure sequence (not in a pipeline)
- In 1999 the i486 moved to a 5 stage pipeline, with a classical organization plus 2 DECODE steps (primary and secondary – Decode/Translate)



The i486 Pipeline

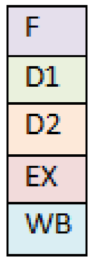


Five instructions going through a pipeline at the same time.

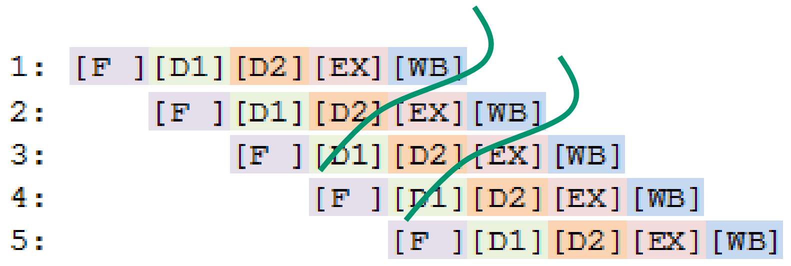
This was for calculations like displacements in a complex addressing model

Pipelining vs software development

- Programmers cannot drive the internal behavior of a pipeline processor (that's microcode!!!!)
- However, the way software is written can hamper the actual pipeline efficiency
- An example – XOR based swap of 2 values:
  - XOR a,b – XOR b,a XOR a,b
- Each instruction has a source coinciding with a destination of the previous instruction



The i486 Pipeline



```
1: [F ][D1][D2][EX][WB]
2:     [F ][D1][D2][EX][WB]
3:         [F ][D1][D2][EX][WB]
4:             [F ][D1][D2][EX][WB]
5:                 [F ][D1][D2][EX][WB]
```

Five instructions going through a pipeline at the same time.

Some examples

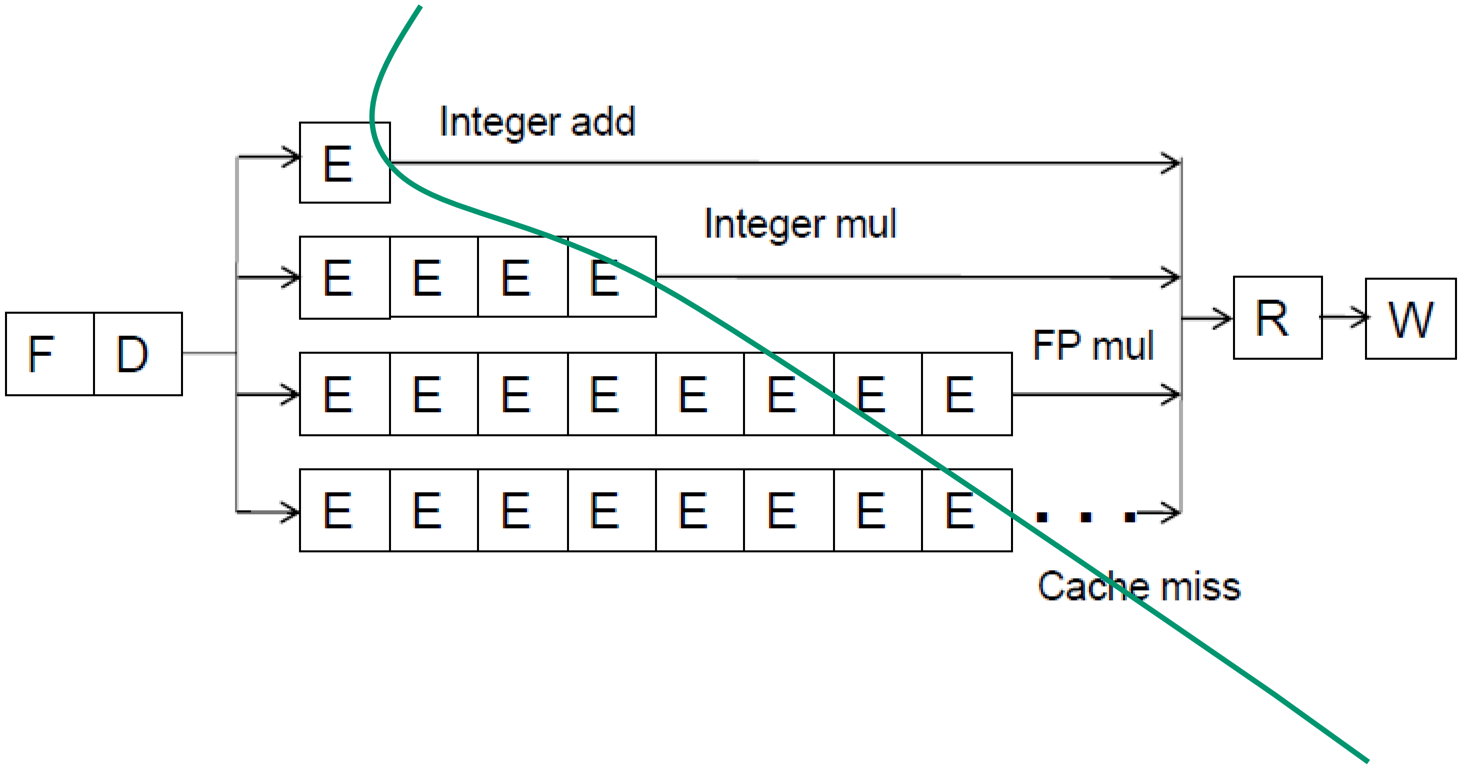
- Pointer based accesses plus pointer manipulation should be carefully written
- Writing in a cycle the following two can make a non negligible difference
  - `a = *++p`
  - `a = *p++`
- Also, there are machine instructions which lead to flush the pipeline, because of the actual organization of the internal CPU circuitry
- In x86 processors, one of them is `CPUID` which gets the numerical id of the processor we are working on
- <u>On the other hand using this instruction you are sure that no previous instruction in the actual executable module is still in flight along the pipeline</u>

The Intel x86 superscalar pipeline

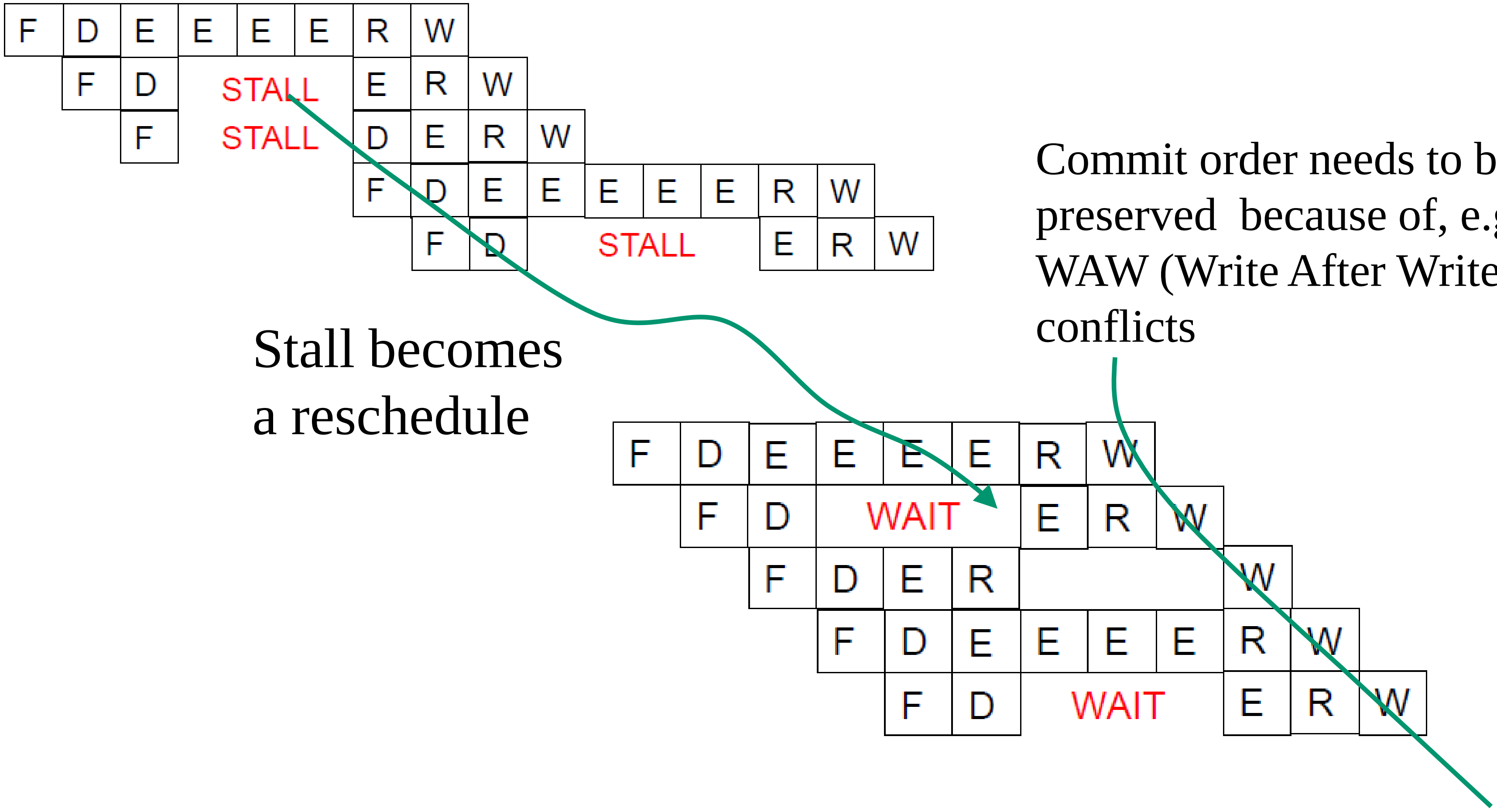
- Multiple pipelines operating simultaneously
- Intel Pentium Pro processors (1995) had 2 parallel pipelines
- EX stages could be actuated in real parallelism thanks to hardware redundancy and differentiation (multiple ALUs, differentiated int/float hardware processing support etc)
- Given that slow instructions (requiring more processor cycles) were one major issue, this processor adopted the OOO model (originally inspired by Robert Tomasulo’s Algorithm – IBM 360/91 1966)
- Baseline idea:
  - ✓ Commit (retire) instructions in program order
  - ✓ Process independent instructions (on data and resources) as soon as possible

The instruction time span problem

Delay reflected in to a pipeline execution of independent instructions



The instruction time span problem



Stall becomes a reschedule

Commit order needs to be preserved because of, e.g. WAW (Write After Write) conflicts

OOO pipeline - speculation

- **Emission:** the action of injecting instructions into the pipeline
- **Retire:** The action of committing instructions, and making their side effects "visible" in terms of ISA exposed architectural resources
- What's there in the middle between the two?
- An execution phase in which the different instructions can surpass each other
- Core issue (beyond data/control dependencies): **exception preserving!!!**
- OOO processors may generate **imprecise exceptions** such that the processor/architectural state may be different from the one that should be observable when executing the instructions along the original order

Imprecise exceptions

- The pipeline may have already executed an instruction *A* that, along program flow, is located after an instruction *B* that causes an exception
- Instruction *A* may have changed the micro-architectural state, although finally not committing its actions onto ISA exposed resources (registers and memory locations updates) – <u>the recent Meltdown security attack exactly exploits this feature</u>
- The pipeline may have not yet completed the execution of instructions preceding the offending one, so their ISA exposed side effects are not yet visible upon the exception
- …. we will be back with more details later on

Robert Tomasulo's algorithm

- Let's start from the tackled hazards – the scenario is of two instructions *A* and *B* such that $A \rightarrow B$ in program order:
  - RAW (Read After Write) – *B* reads a datum before *A* writes it, which is clearly stale – this is a clear data dependency
  - WAW (Write After Write) – *B* writes a datum before *A* writes the same datum – the datum exposes a stale value
  - WAR (Write After Read) – *B* writes a datum before *A* reads the same datum – the read datum is not consistent with data flow (it is in the future of *A*'s execution)

Algorithmic ideas

- RAW – we keep track of “when” data requested in input by instructions are ready
- Register renaming for coping with both WAR an WAW hazards
- In the renaming scheme, a source operand for an instruction can be either an actual register label, or another label (a renamed register)
- In the latter case it means that the instruction needs to read the value from the renamed register, rather than from the original register
- A renamed register materializes the concept of speculative (not yet committed) register value, made anyhow available as input to the instructions

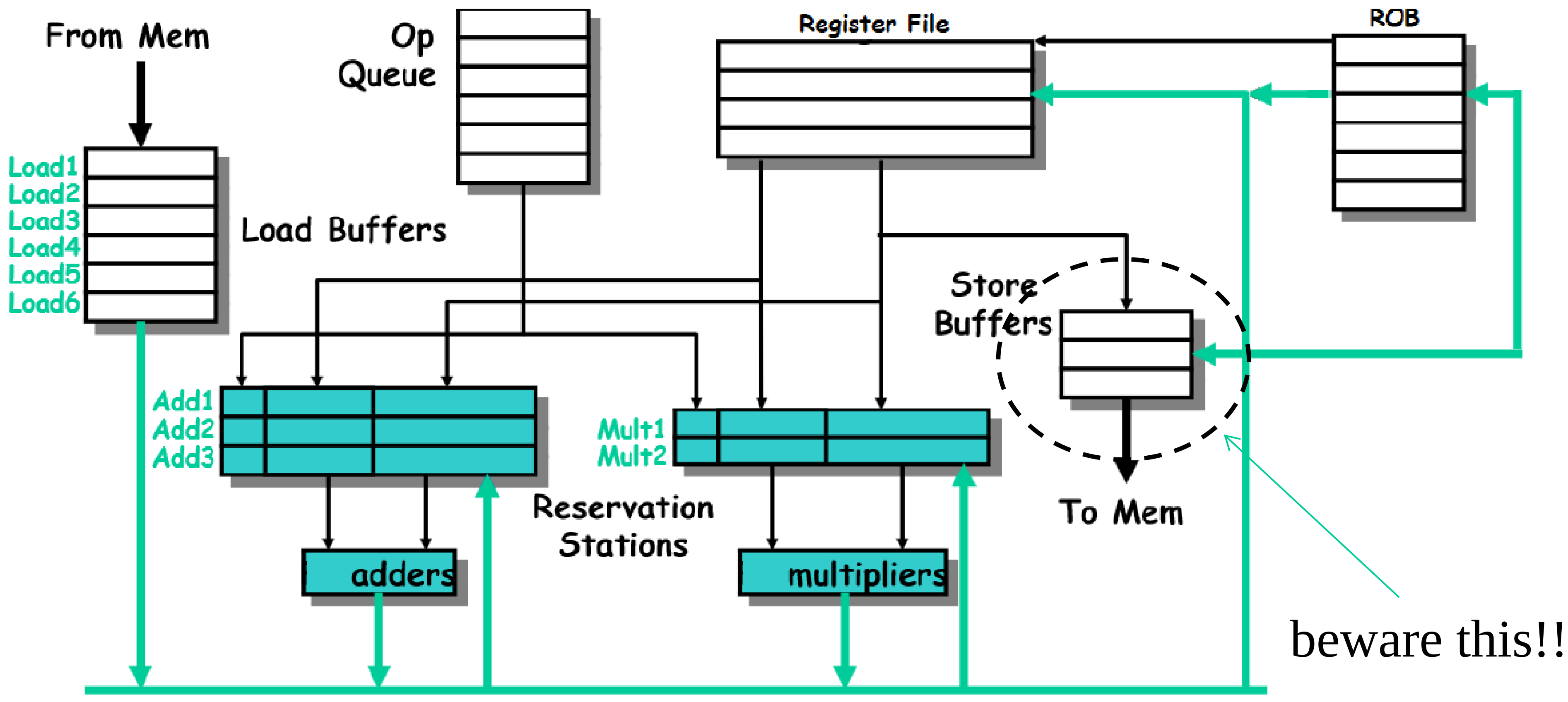
Reservation stations

- They are buffers (typically associated with different kinds of computational resources – integer vs floating point operators)
- They contain:
  - OP – the operations to be executed
  - Qj, Qk – the reservation stations that will produce the input for OP
  - Alternatively, Vj, Vk, the actual values (e.g. register values) to be used in input by OP
- By their side, registers are marked with the reservation station name Q such that it will produce the new value to be installed, if any

CDB and ROB

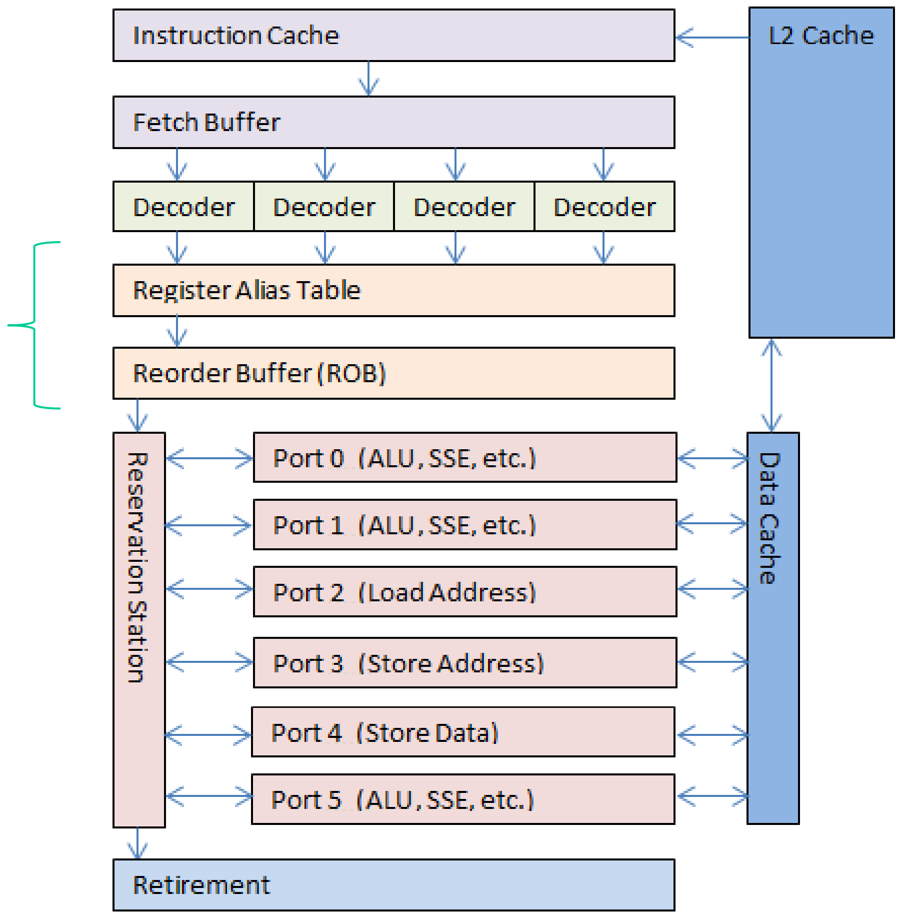
- A Common Data Bus (CDB) allows data to flow across reservation stations (so that an operation is fired when all its input data are available)
- A Reorder Buffer (ROB) acquires all the newly produced instruction values (also those transiting on CDB), and keeps them uncommitted up to the point where the instruction is retired
- ROB is also used for input to instructions that need to read from uncommitted values

An architectural scheme



x86 OOO main architectural organization

Who depends on who?

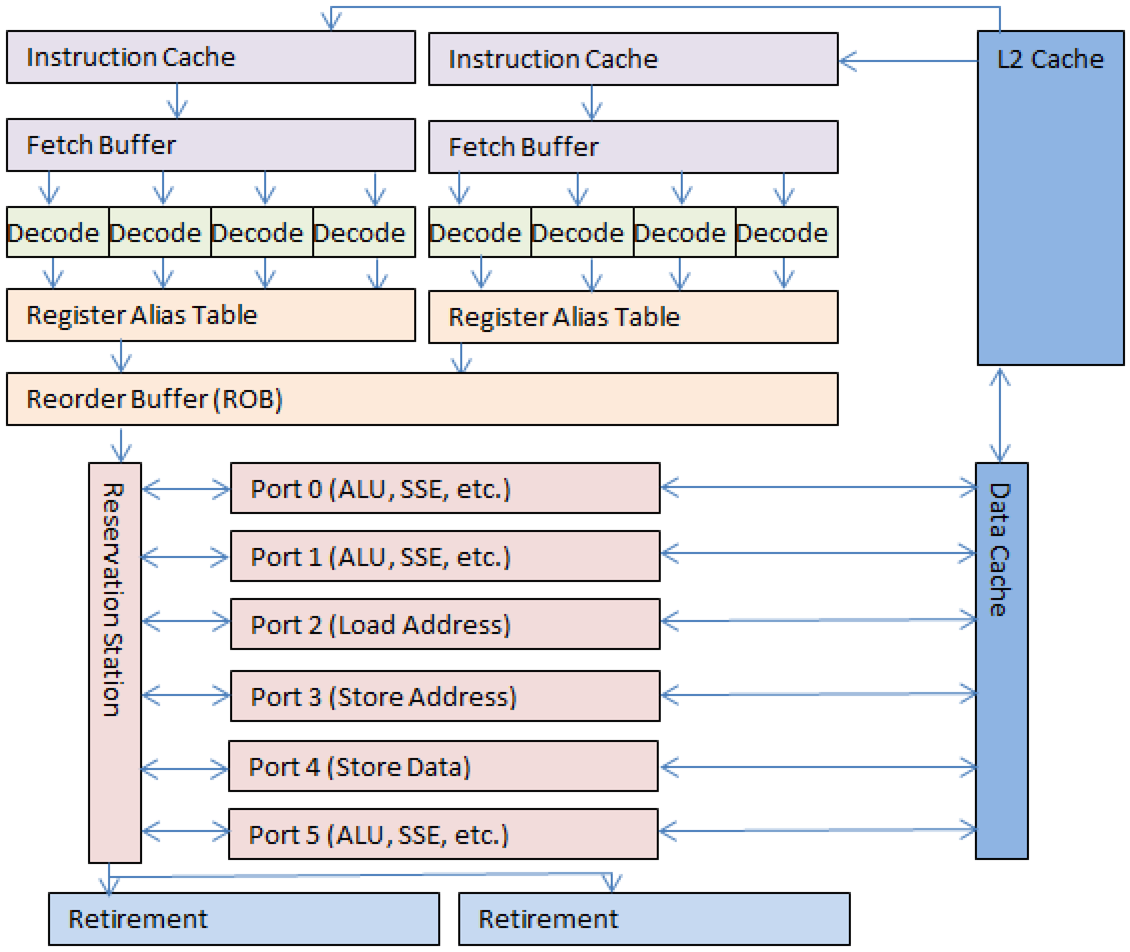


Out Of Order Core as used from 1995 to present. The color coding follows the same five stages used in previous processors. Some stages and buffers are not shown since they vary from processor to processor.

Impact of OOO in x86

- OOO allowed so fast processing of instructions that room was still there on core hardware components to actually carryout work
- … also because of delays within the memory hierarchy
- … why not using the same core hardware engine for multiple program flows?
- This is called hyper-threading, and is actually exposed to the programmer at any level (user, OS etc.)
- ISA exposed registers (for programming) are replicated, as if we had 2 distinct processors
- Overall, OOO is not exposed (instructions are run as in a black box) although the way of writing software can impact the effectiveness of OOO and more generally of pipelining

Baseline architecture of OOO Hyper-threading



Hyper-Threading OOO Core

Coming to interrupts

- Interrupts typically flush all the instructions in the pipeline, as soon as one commits and the interrupt is accepted
- As an example, in a simple 5-stage pipeline IF, ID, EX, MEM residing instructions are flushed because of the acceptance of the interrupt on the WB phase of the currently finalized instruction
- This avoids the need for handling priorities across interrupts and exceptions possibly caused by instructions that we might let survive into the pipeline (no standing exception)
- Interrupts may have a significant penalty in terms of wasted work on modern OOO based pipelines

Back to exceptions: types vs pipeline stages

- Instruction Fetch, & Memory stages
  - Page fault on instruction/data fetch
  - Misaligned memory access
  - Memory-protection violation
- Instruction Decode stage
  - Undefined/illegal opcode
- Execution stage
  - Arithmetic exception
- Write-Back stage
  - *No exceptions!*

Back to exceptions: handling

- When an instruction in a pipeline gives rise to an exception, the latter is not immediately handled
- As we shall see later, such instruction in fact might even require to disappear from program flow (as an example because of miss-prediction in branches)
- It is simply marked as **offending** (with one bit traveling with the instruction across the pipeline)
- When the retire stage is reached, the exception takes place and the pipeline is flushed, resuming fetch operations from the right place in memory
- **NOTE**: micro architectural effects of in flight instructions that are later squashed (may) still stand there – see the Meltdown attack …

Meltdown primer

Flush cache
Read a kernel level byte B
Use B for displacing a reading memory

A sequence with imprecise exception under OOO

Offending instruction (memory protection violation)

“Phantom” instruction with real micro-architectural side effects

Pipeline vs branches

- The hardware support for improving performance under (speculative) pipelines in face of branches is called ***Dynamic Predictor***
- Its actual implementation consists of a Branch-Prediction Buffer (BPB) – or Branch History Table (BHT)
- The baseline implementation is based on a cache indexed by lower significant bits of branch instructions and one status bit
- The status bit tells whether the jump related to the branch instruction has been recently executed
- The (speculative) execution flow follows the direction related to the prediction by the status bit, thus following the recent behavior
- Recent past is expected to be representative of near future

Multiple bits predictors

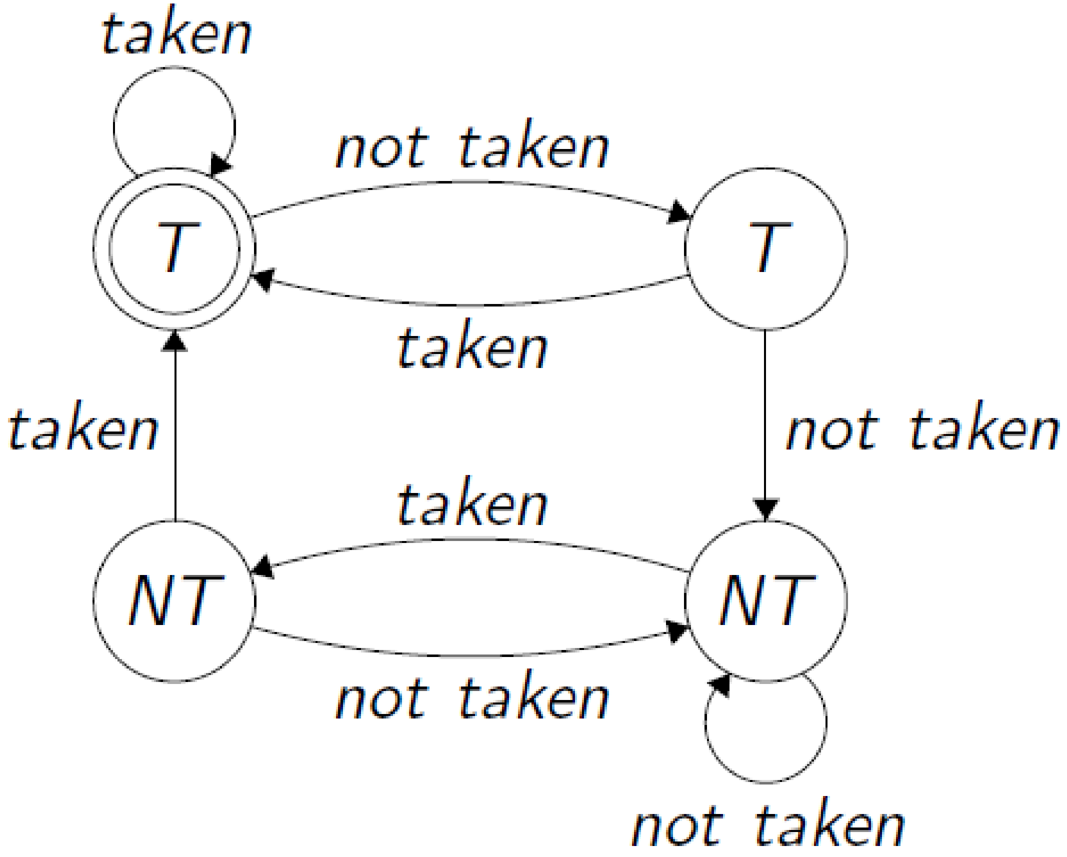
- One bit predictors “fail” in the scenario where the branch is often taken (or not taken) and infrequently not taken (or taken)
- In these scenarios, they leads to 2 subsequent errors in he prediction (thus 2 squashes of the pipeline)
- If this really important? Nested loops tell yes
- The conclusion of the inner loop leads to change the prediction, which is anyhow re-changed at the next iteration of the outer loop
- Two-bit predictors require 2 subsequent prediction errors for inverting the prediction
- So each of the four states tells whether we are running with
  - ✓ YES prediction (with one or zero mistakes since the last passage on the branch)
  - ✓ NO prediction (with one or zero mistakes since the last passage on the branch)

An example

```
  mov $0, %ecx
. outerLoop:
  cmp $10, %ecx
  je .done
  mov $0, %ebx

.innerLoop:
  ; actual code
  inc %ebx
  cmp $10, %ebx
  jnz .innerLoop

  inc %ecx
  jmp .outerLoop
 .done:
```

This branch prediction is inverted at each ending inner-loop cycle

The actual two-bit predictor state machine



Do we nee to go beyond two-bit predictors?

- Conditional branches are around 20% of the instructions in the code
- Pipelines are deeper
  - ✓ A greater misprediction penalty
- Superscalar architectures execute more instructions at once
  - ✓ The probability of finding a branch in the pipeline is higher
- The answer is clearly yes
- One more sophisticate approach offered by Pentium (and later) processors is Correlated Two-Level Prediction
- Another one offered by Alpha is Hybrid Local/Global predictor (also known as Tournament Predictor)

A motivating example

```
if (aa == VAL)
     aa = 0 ;
if (bb == VAL )
     bb = 0;
if (aa != bb){
     //do the work
}
```

Not branching on these implies not branching on the subsequent

Idea of correlated prediction: lets' try to predict what will happen at the third branch by looking at the history of what happened in previous branches

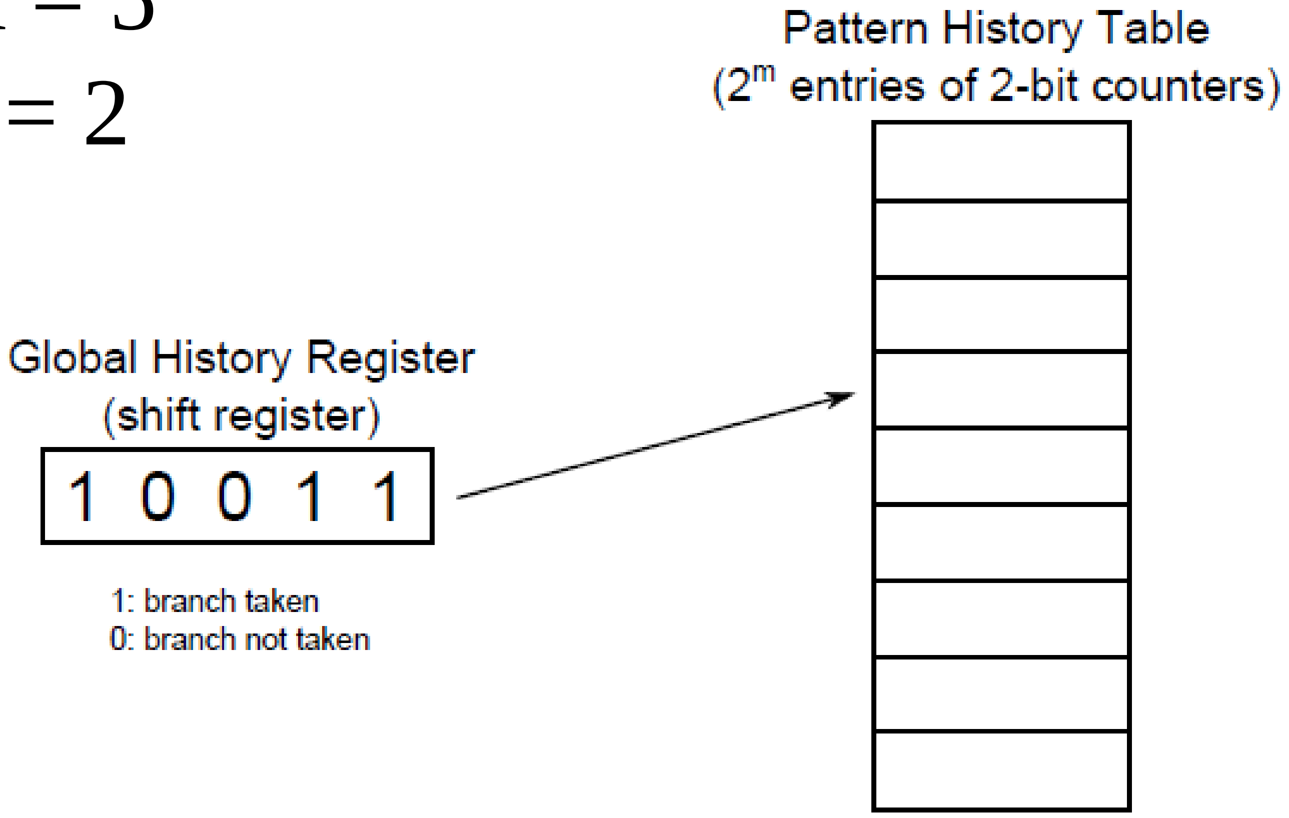
The (m,n) two-level correlated predictor

- The history of the last **m** branches is use to [preicti what will happen to the current branch
- The current branch is predicted with an **n**-bit predictor
- There are 2^m n-bit predictors
- The actual predictor for the current prediction is selected on the basis of the results of the last **m** branches, as code in to the 2^m bitmask
- A two-level correlated predictor of the form (0,2) boils own to a classical 2-bit predictor

(m,n) predictor architectural schematization

$m = 5$

$n = 2$

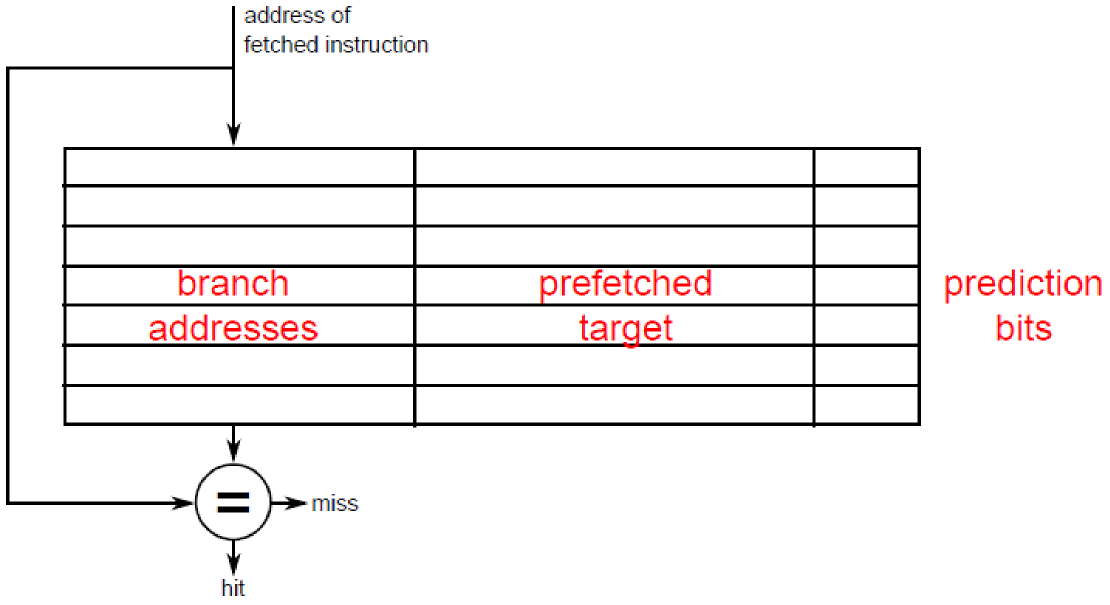


Tournament predictor

- The prediction of a branch is carried out by either using a local (per branch) predictor or a correlate (per history) predictor
- In the essence we have a combination of the two different prediction schemes
- Which of the two needs to be exploited at each individual prediction is encoded into a 4-states (2-bit based) history of success/failures
- This way, we can detect whether treating a branch as an individual in the prediction leads to improved effectiveness compared to treating it as an element in a sequence of individuals

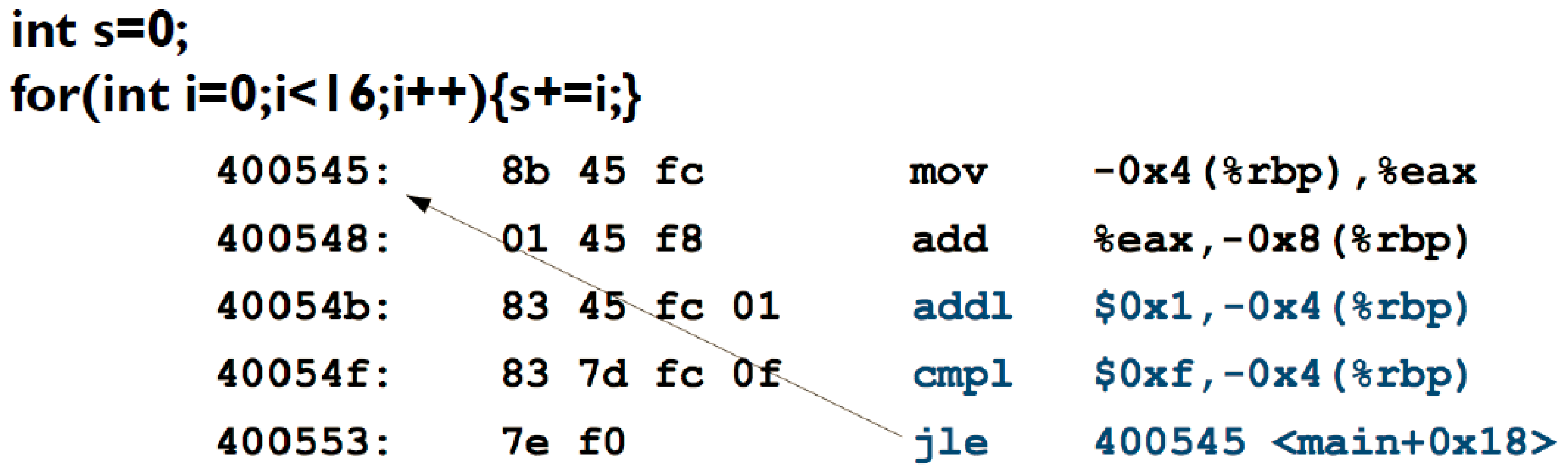
The very last concept on branch prediction: indirect branches

- These are branches for which the target is not know at instruction fetch time
- Essentially these are kind of families of branches (multi-target branches)
- An x86 example: `jmp eax`



Loop unrolling

- This is a software technique that allows reducing the frequency of branches when running loops, and the relative cost of branch control instructions
- Essentially it is based on having the code-writer or the compiler to enlarge the cycle body by inserting multiple statements that would otherwise be execute in different loop iterations



```
int s=0;
for(int i=0;i<16;i++){s+=i;}
```

```
400545:    8b 45 fc        mov    -0x4(%rbp),%eax
400548:    01 45 f8        add    %eax,-0x8(%rbp)
40054b:    83 45 fc 01     addl   $0x1,-0x4(%rbp)
40054f:    83 7d fc 0f     cmpl   $0xf,-0x4(%rbp)
400553:    7e f0           jle    400545 <main+0x18>
```

gcc unroll directives

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
```

  Region to unroll

```
#pragma GCC pop_options
```

- One may also specify the unroll factor via

  ```
  #pragma unroll(N)
  ```

- In more recent gcc versions (e.g. 4 or laters) it works with the `-O` directive

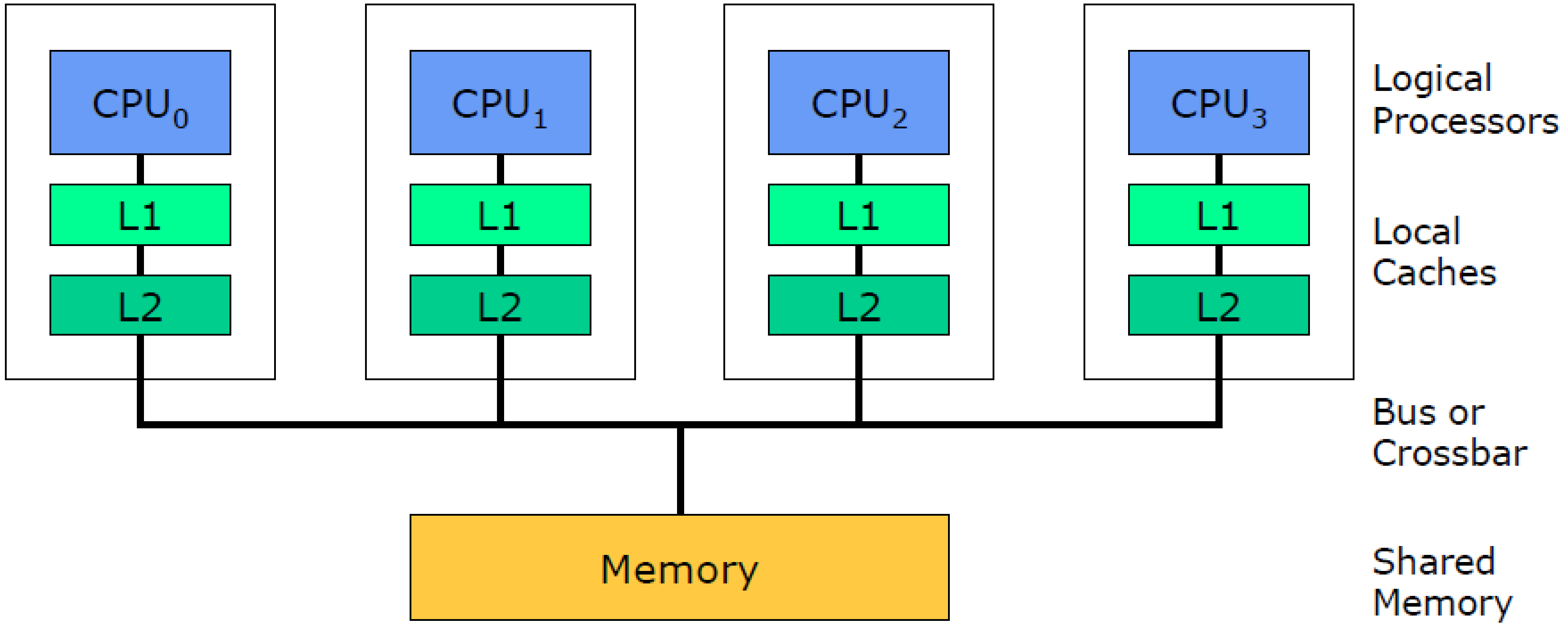
Beware unroll side effects

- In may put increased pressure on register usage leading to more frequent memory interactions
- When relying on huge unroll values code size can grow enormously, consequently locality and cache efficiency may degrade significantly
- Depending on the operations to be unrolled, it might be better to reduce the number of actual iterative steps via “vectorization”, a technique that we will look at later on

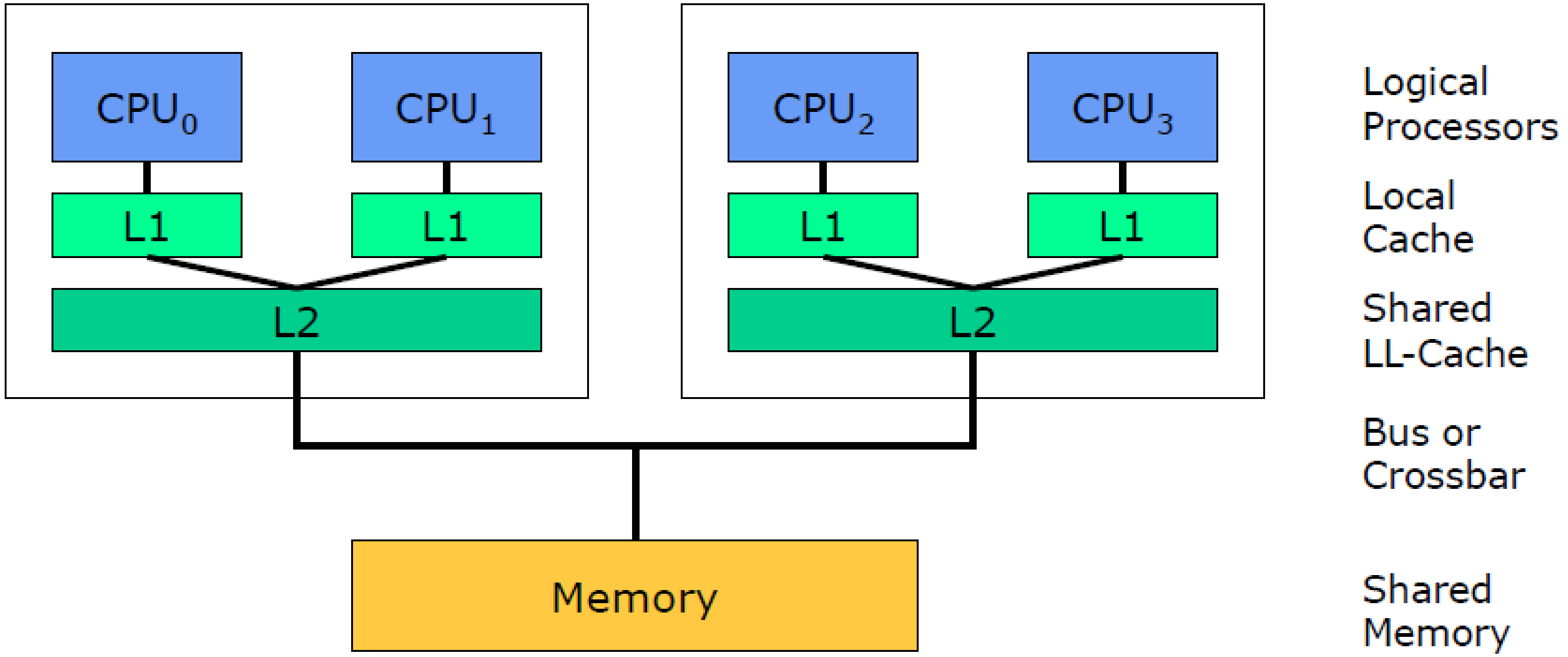
Clock frequency and power wall

- How can we make a processors run faster?
- Better exploitation of hardware components and growth of transistors’ packaging – e.g. the More’s low
- Increase of the clock frequency
- But nowadays we have been face with the power wall, which actually prevents the building of processors with higher frequency
- In fact the power consumption grows exponentially with voltage according to the VxVxF rule (and 130 W is considered the upper bound for dissipation)
- The way we have for continuously increasing the computing power of individual machines is to rely on parallel processing units

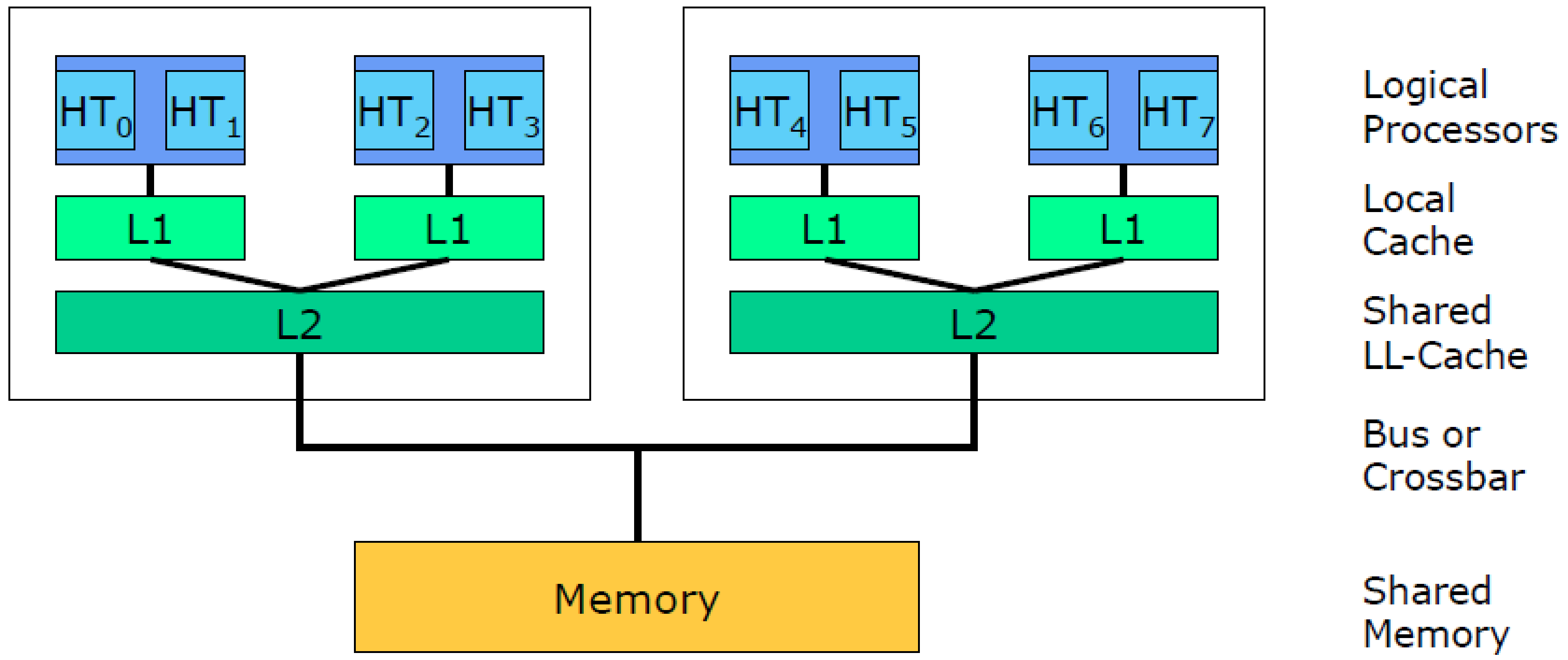
Symmetric multiprocessors



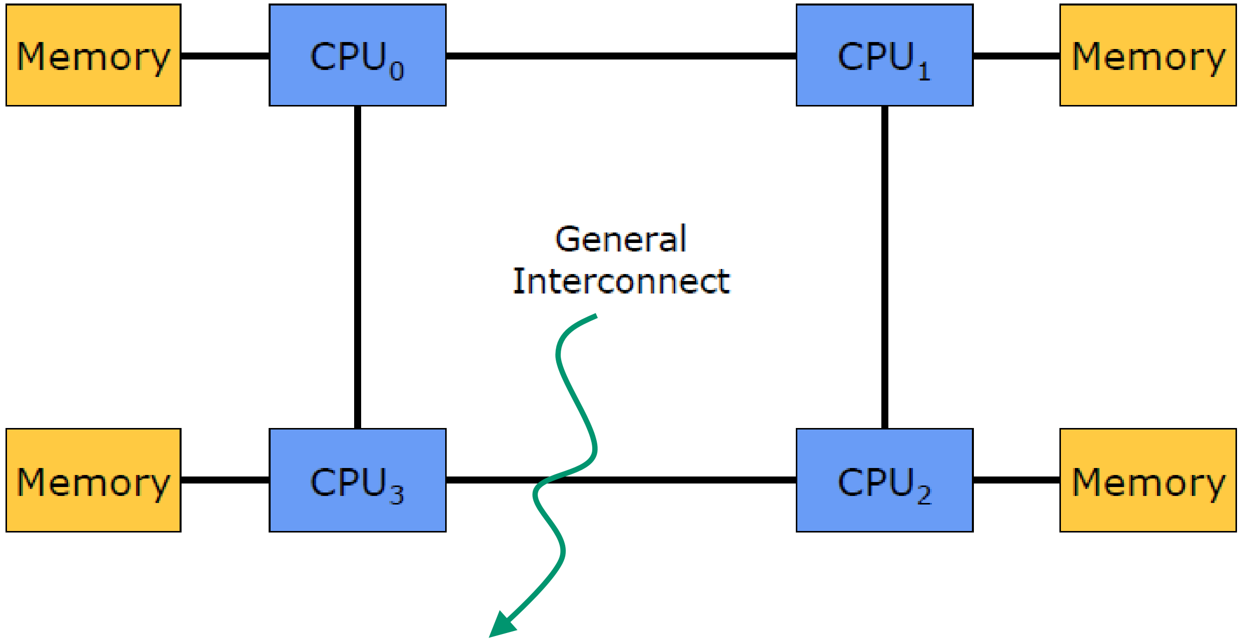
Chip Multi Processor (CMP) - Multicore



Symmetric Multi-threading (SMT) - Hyperthreading



Making memory circuitry scalable – NUMA (Non Uniform memory Access)



This may have different shapes depending on chipsets

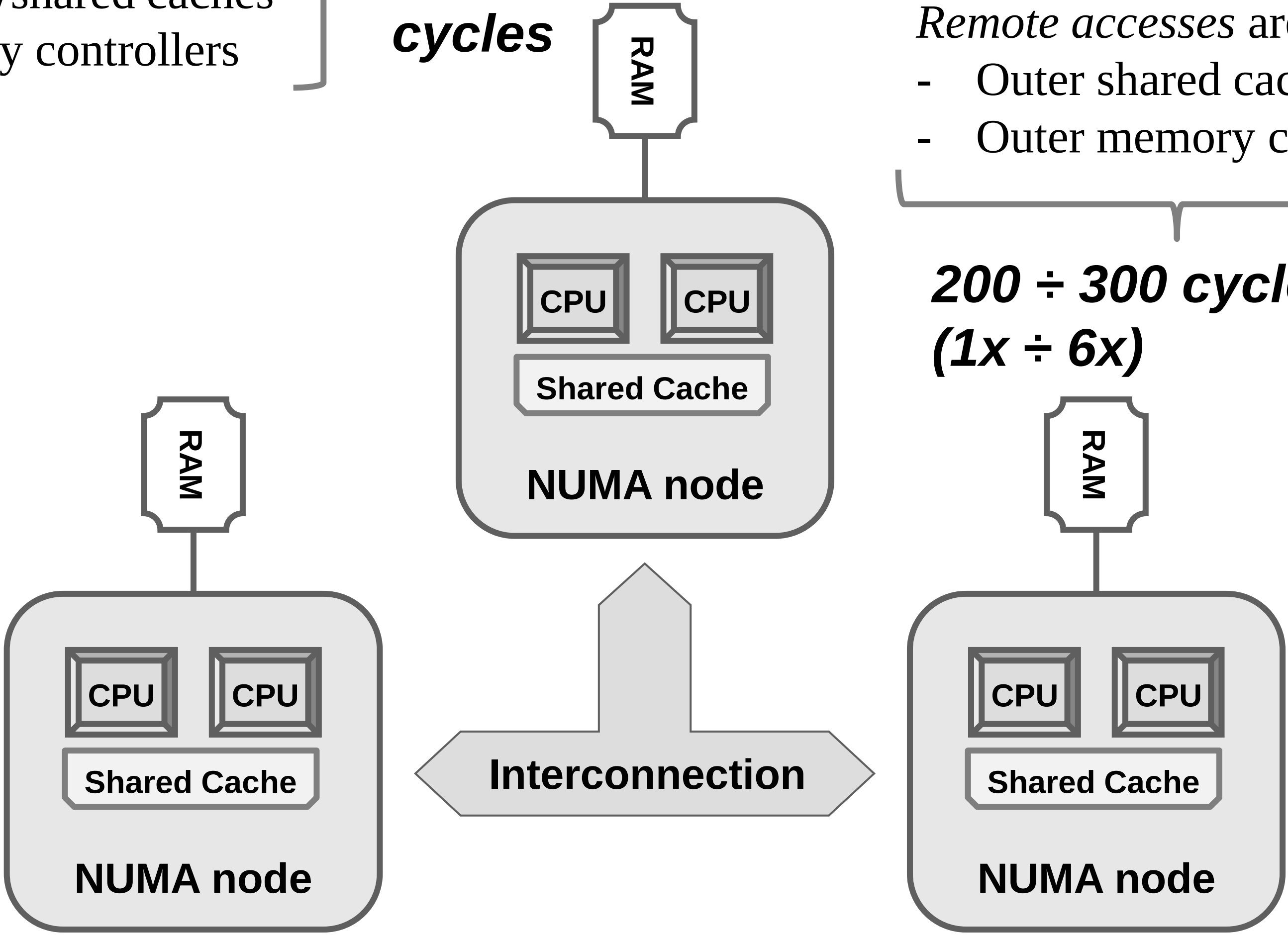
NUMA latency asymmetries

Local accesses are served by
- Inner private/shared caches
- Inner memory controllers

50 ÷ 200 cycles

Remote accesses are served by
- Outer shared caches
- Outer memory controllers

200 ÷ 300 cycles (1x ÷ 6x)



Cache coherency

- CPU-cores see memory contents through their caching hierarchy
- This is essentially a **replication system**
- The problem of defining what value (within the replication scheme) should be returned upon reading from memory is also referred to as “cache coherency”
- This is definitely different from the problem of defining when written values by programs can be actually read from memory
- The latter is in fact know to as the “consistency” problem, which we will discuss later on
- Overall, cache coherency is not memory consistency, but it is anyhow a big challenge to cope with, with clear implications on performance

Defining coherency

- A read from location X, previously written by a processor, returns the last written value if no other processor carried out writes on X in the meanwhile – **Causal consistency along program order**
- A read from location X by a processor, which follows a write on X by some other processor, returns the written value if the two operations are sufficiently separated along time (an no other processor writes X in the meanwhile) – **Avoidance of staleness**
- All writes on X from all processors are serialized, so that the writes are seen from all processors in a same order – **We cannot (ephemerally or permanently) invert memory updates**
- …. however we will come back to defining when a processor actually writes to memory!!
- Please take care that coherency deals with individual memory location operations!!!

Cache coherency (CC) protocols: basics

- A CC protocol is the result of choosing
  - ✓ a set of transactions supported by the distributed cache system
  - ✓ a set of states for cache blocks
  - ✓ a set of events handled by controllers
  - ✓ a set of transitions between states
- Their design is affected by several factors, such as
  - ✓ interconnection topology (e.g., single bus, hierarchical, ring-based)
  - ✓ communication primitives (i.e., unicast, multicast, broadcast)
  - ✓ memory hierarchy features (e.g., depth, inclusiveness)
  - ✓ cache policies (e.g., write-back vs write-through)
- Different CC implementations have different performance
  - ✓ Latency: time to complete a single transaction
  - ✓ Throughput: number of completed transactions per unit of time
  - ✓ Space overhead: number of bits required to maintain a block state

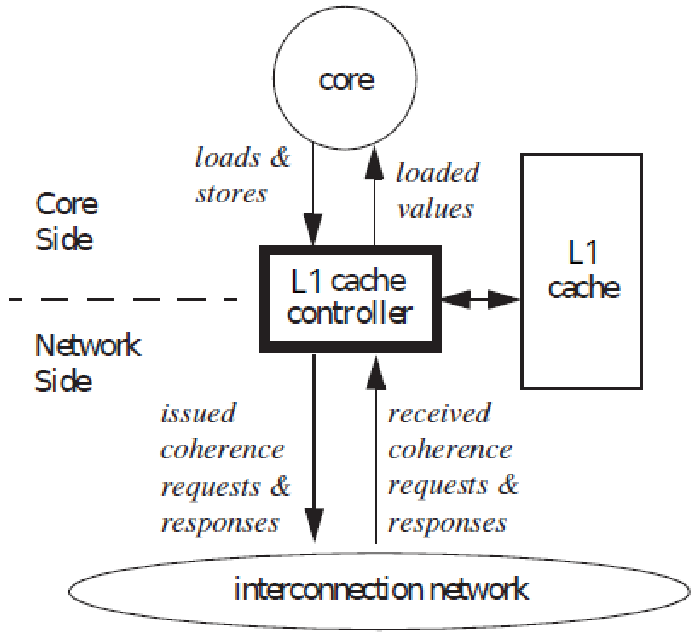
Families of CC protocols

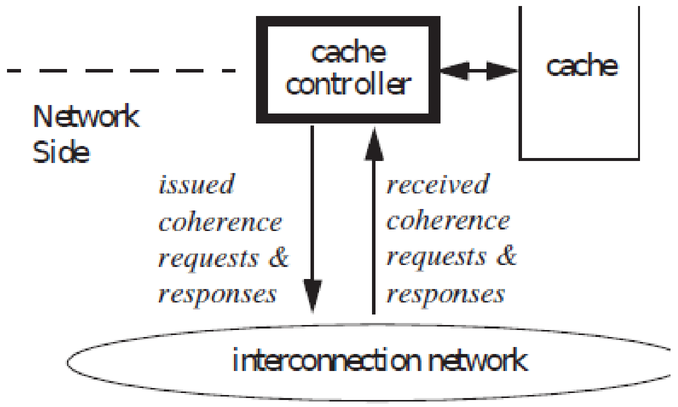
- When to update copies in other caches?
- Invalidate protocols:
  - ✓ When a core writes to a block, all other copies are invalidated
  - ✓ Only the writer has an up-to-date copy of the block
  - ✓ Trades latency for bandwidth
- Update protocols:
  - ✓ When a core writes to a block, it updates all other copies
  - ✓ All cores have an up-to-date copy of the block
  - ✓ Trades bandwidth for latency

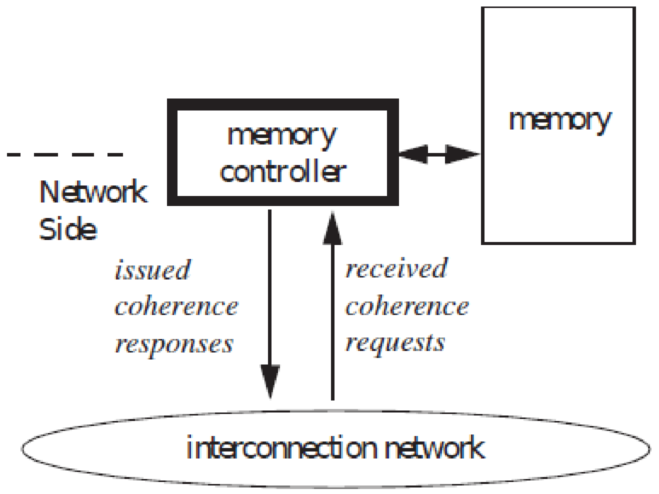
“Snooping cache” coherency protocols

- At the architectural level, these are based on some broadcast medium (also called network) across all cache/memory components
- Each cache/memory component is connected to the broadcast medium by relying on a controller, which snoops (observes) the in-flight data
- The broadcast medium is used to issue “transactions” on the state cache blocks’ state
- Agreement on state changes comes out by <u>serializing the transactions traveling along the broadcast medium</u>
- A state transition cannot occur unless the broadcast medium is acquired by the source controller
- Sate transitions are distributed (across the components), but are carried out atomically thanks to serialization over the broadcast medium

An architectural scheme




cache controller
cache
Network Side
issued coherence requests & responses
received coherence requests & responses
interconnection network



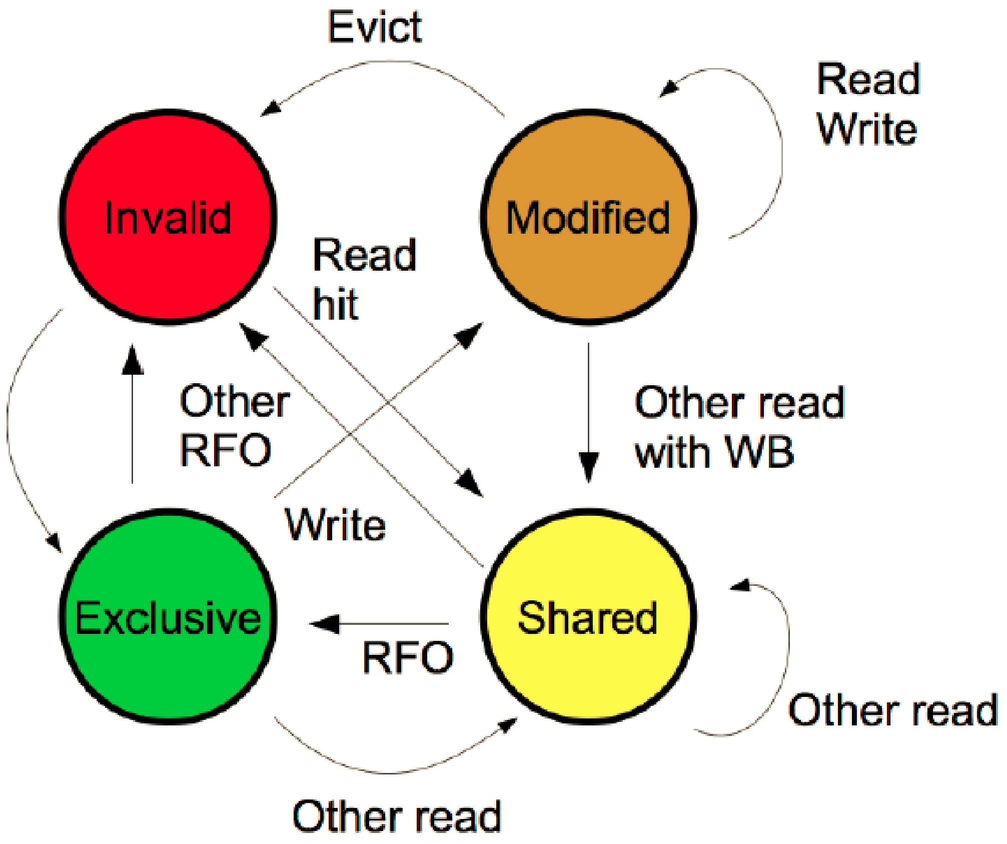
Write/read transactions with invalidation

- A write transaction invalidates all the other copies of the cache block
- Read transactions
  - ✓ Get the latest updated copy from memory in write-through caches
  - ✓ Get the latest updated copy from memory or from another caching component in write-back caches (e.g. Intel processors)
- We typically keep track of whether
  - ✓ A block is in the modified state (just written, hence invalidating all the other copies)
  - ✓ A block is in shared state (someone got the copy from the writer or from another reader)
  - ✓ A block is in the invalid state
- This is the MSI (Modified-Shared-Invalid) protocol

Reducing invalidation traffic upon writes: MESI

- Similar to MSI, but we include an “exclusive” state indicating that a unique valid copy is owned, independently of whether the block has been written or not

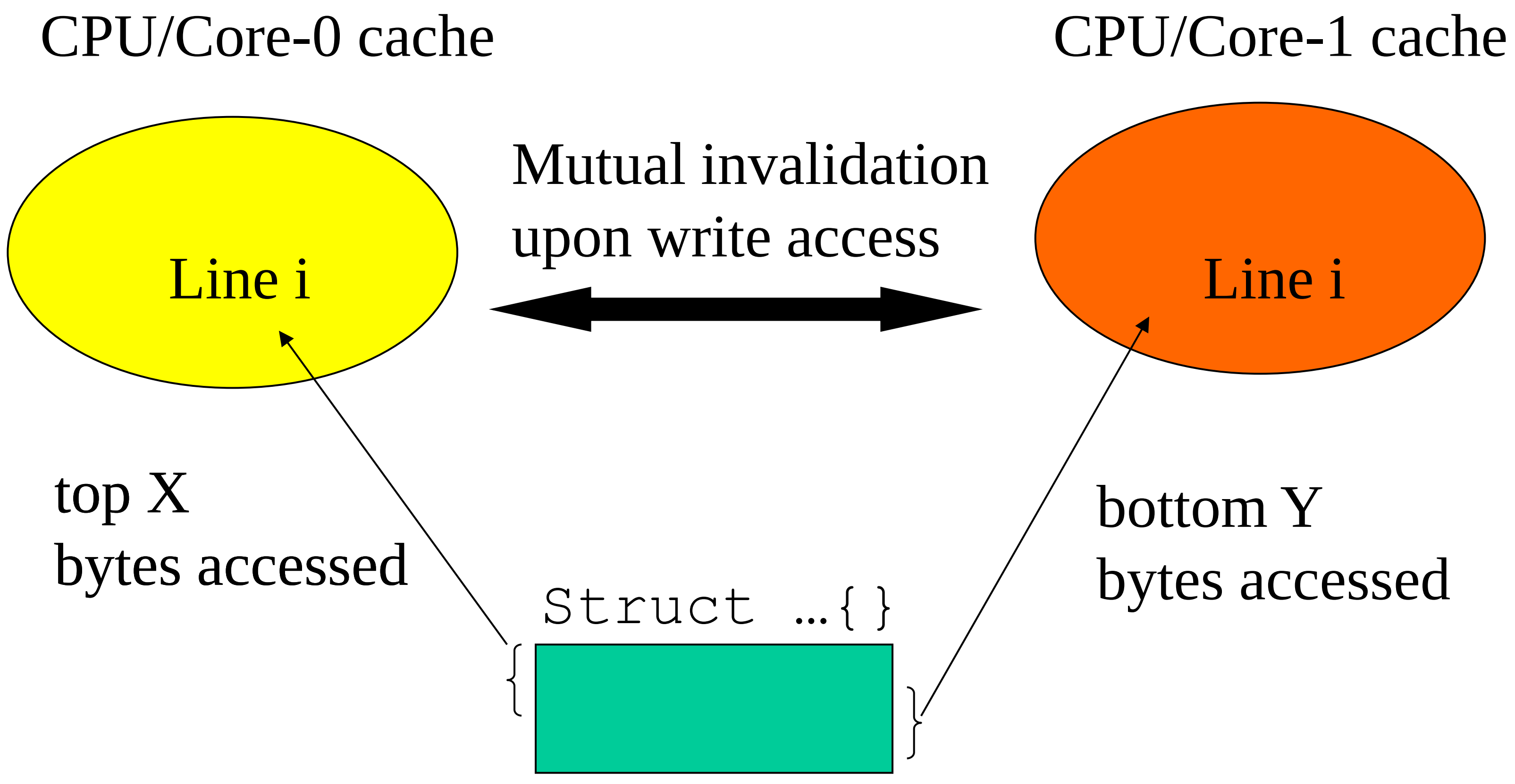
RFO = Request
For
Ownership



Software exposed cache performance aspects

- “Common fields” access issues
  - ✓ Most used fields inside a data structure should be placed at the head of the structure in order to maximize cache hits
  - ✓ This should happen provided that the memory allocator gives <u>cache-line aligned addresses for dynamically allocated memory chunks</u>
- “Loosely related fields” should be placed sufficiently distant inside the data structure so to avoid <u>performance penalties due to false cache sharing</u>

The false cache sharing problem



X+Y < 2 x CACHE_LINE

Example code leading to false cache sharing

Fits into a same cache line
(typically 64/256 bytes)

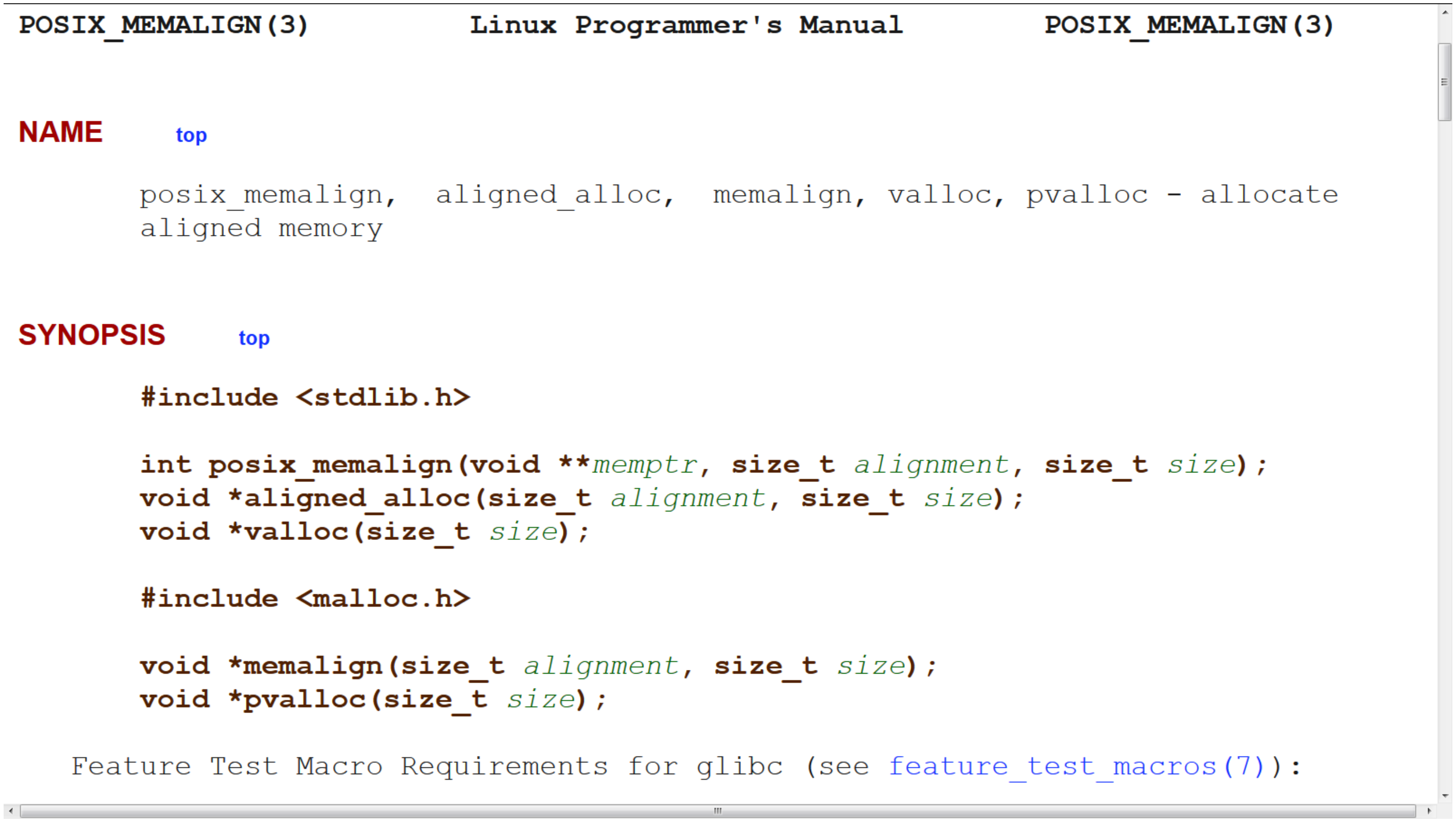
```c
struct foo {
    int x;
    int y;
};

struct foo f;
```

```c
void inc_x(void)
{
    int i;
    for(i = 0; i <
            1000000; i++)
        f.x++;
}
```

```c
int sum_y(void) {
    int s = 0;
    int i;
    for (i = 0; i <
            1000000; i++)
        s += f.y;
    return s;
}
```

These reads from the cache
line find cache-invalid data, even though
the actual memory location we are reading from
does not change over time

Posix memory-aligned allocation



```
POSIX_MEMALIGN(3)          Linux Programmer's Manual         POSIX_MEMALIGN(3)
```

NAME top

```
       posix_memalign,  aligned_alloc,  memalign, valloc, pvalloc - allocate
       aligned memory
```

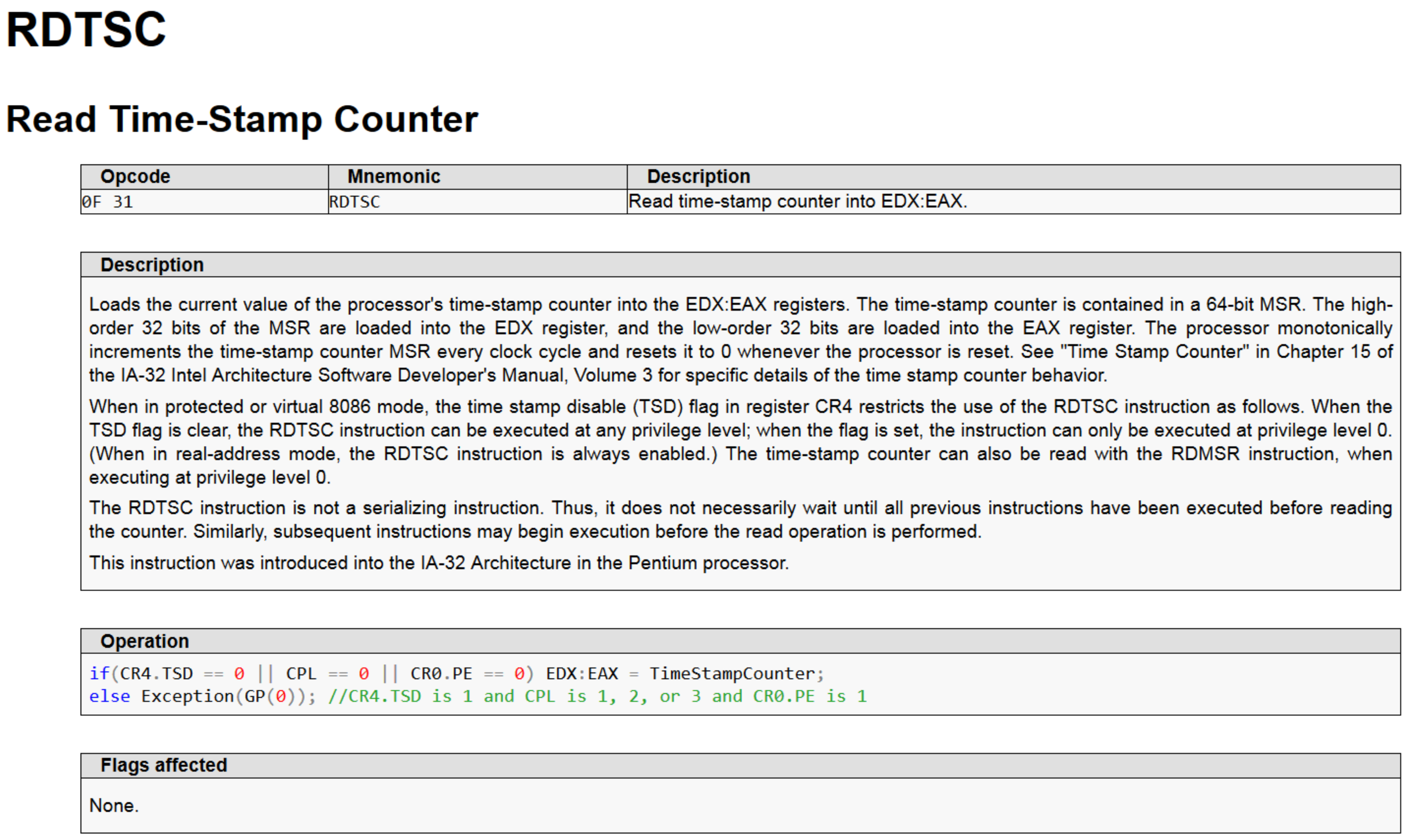
SYNOPSIS top

```
       #include <stdlib.h>

       int posix_memalign(void **memptr, size_t alignment, size_t size);
       void *aligned_alloc(size_t alignment, size_t size);
       void *valloc(size_t size);

       #include <malloc.h>

       void *memalign(size_t alignment, size_t size);
       void *pvalloc(size_t size);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
```

Inspecting cache line accesses

- A technique presented at [USENIX Security Symposium – 2013] is based on observing access latencies on shared data
- Algorithmic steps:
  - ✓ The cache content related to some shared data is flushed
  - ✓ Successively it is re-accessed in read mode
  - ✓ Depending on the timing of the latter accesses we gather whether the datum has been also accessed by some other thread
- Implementation on x86 is based on 2 building blocks:
  - ✓ A high resolution timer
  - ✓ A non-privileged cache line flush instruction
- <u>These algorithmic steps have been finally exploited for he Meltdown attack</u>
- … let's see the details ….

x86 high resolution timers



RDTSC

Read Time-Stamp Counter

Opcode	Mnemonic	Description
`0F 31`	`RDTSC`	Read time-stamp counter into EDX:EAX.

Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 15 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3 for specific details of the time stamp counter behavior.

When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDTSC instruction is always enabled.) The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the IA-32 Architecture in the Pentium processor.

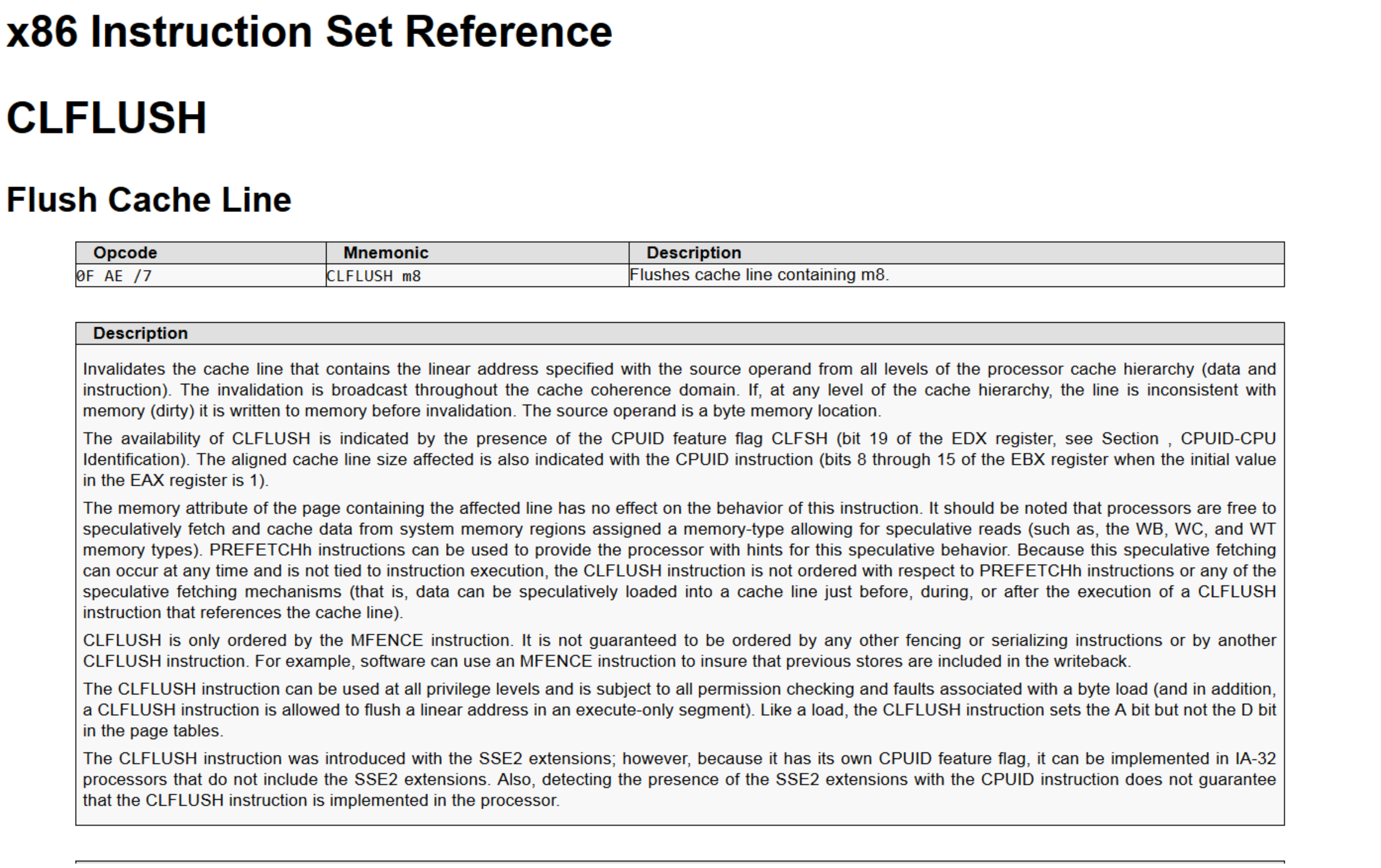
Operation

```
if(CR4.TSD == 0 || CPL == 0 || CR0.PE == 0) EDX:EAX = TimeStampCounter;
else Exception(GP(0)); //CR4.TSD is 1 and CPL is 1, 2, or 3 and CR0.PE is 1
```

Flags affected

None.

x86 (non privileged) cache line flush



x86 Instruction Set Reference

CLFLUSH

Flush Cache Line

Opcode	Mnemonic	Description
`0F AE /7`	`CLFLUSH m8`	Flushes cache line containing m8.

Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see Section , CPUID-CPU Identification). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to insure that previous stores are included in the writeback.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

ASM inline

- Exploited to define ASM instruction to be posted into a C function
- The programmer does not leave freedom to the compiler on that instruction sequence
- Easy way of linking ASM and C notations
- Structure of an ASM inline block of code

```
__asm__ [volatile][goto] (AssemblerTemplate
                          [ : OutputOperands ]
                          [ : InputOperands ]
                          [ : Clobbers ]
                          [ : GotoLabels ]);
```

Meaning of ASM inline fields

- `AssemblerTemplate` - the actual ASM code
- `volatile` – forces the compiler not to take any optimization (e.g. instruction placement effect)
- `goto` – assembly can lead to jump to any label in `GoToLabels`
- `OutputOperands` – data move post conditions
- `InputOperands` – data move preconditions
- `Clobbers` – registers involved in update by the ASM code, additionally to the callee-save ones already specified as operands

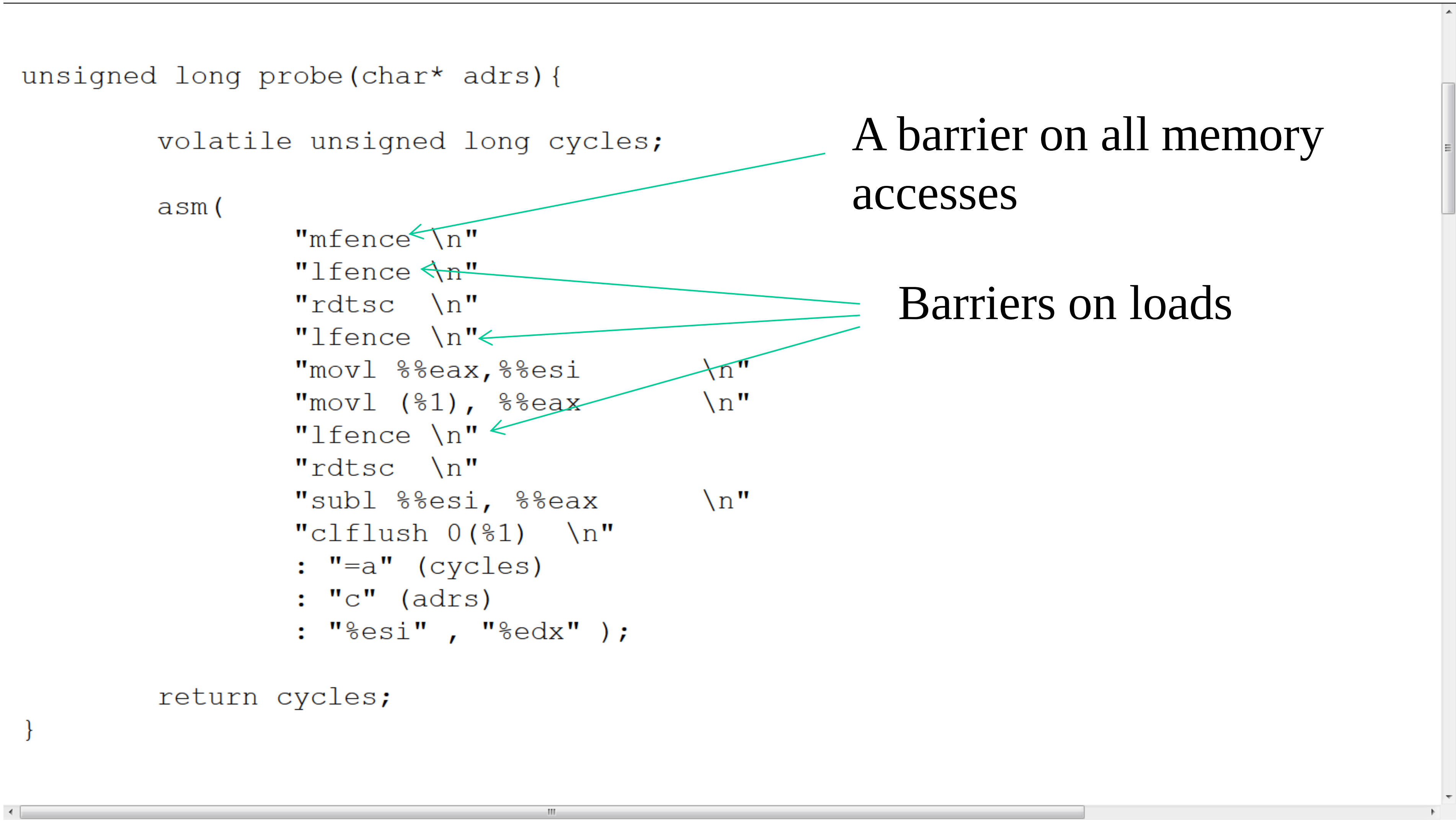
C compilation directives for Operands

- The **=** symbol means that the corresponding perand is used as an **<u>output</u>**
- Hence after the execution of the ASM code block, the operand value becomes the source for a given target location (e.g. for a variable)
- In case the operand needs to keep a value to be used as an **<u>input</u>** (hence the operand is the destination for the value of some source location) then the = symbol does not need to be used

Main gcc supported operand specifications

- **r** – generic register operands
- **m** – generic memory operand (e.g. into the stack)
- **0-9** – reused operand index
- **i/I** – immediate 64/32 bit operand
- **q -** byte-addressable register (e.g. eax, ebx, ecx, edx)
- **A -** eax or edx
- **a, b, c, d, S, D -** eax, ebx, ecx, edx, esi, edi respectively (or al, rax etc variants depending on the size of the actual-instruction operands)

Flush+Reload: measuring cache access latency at user space

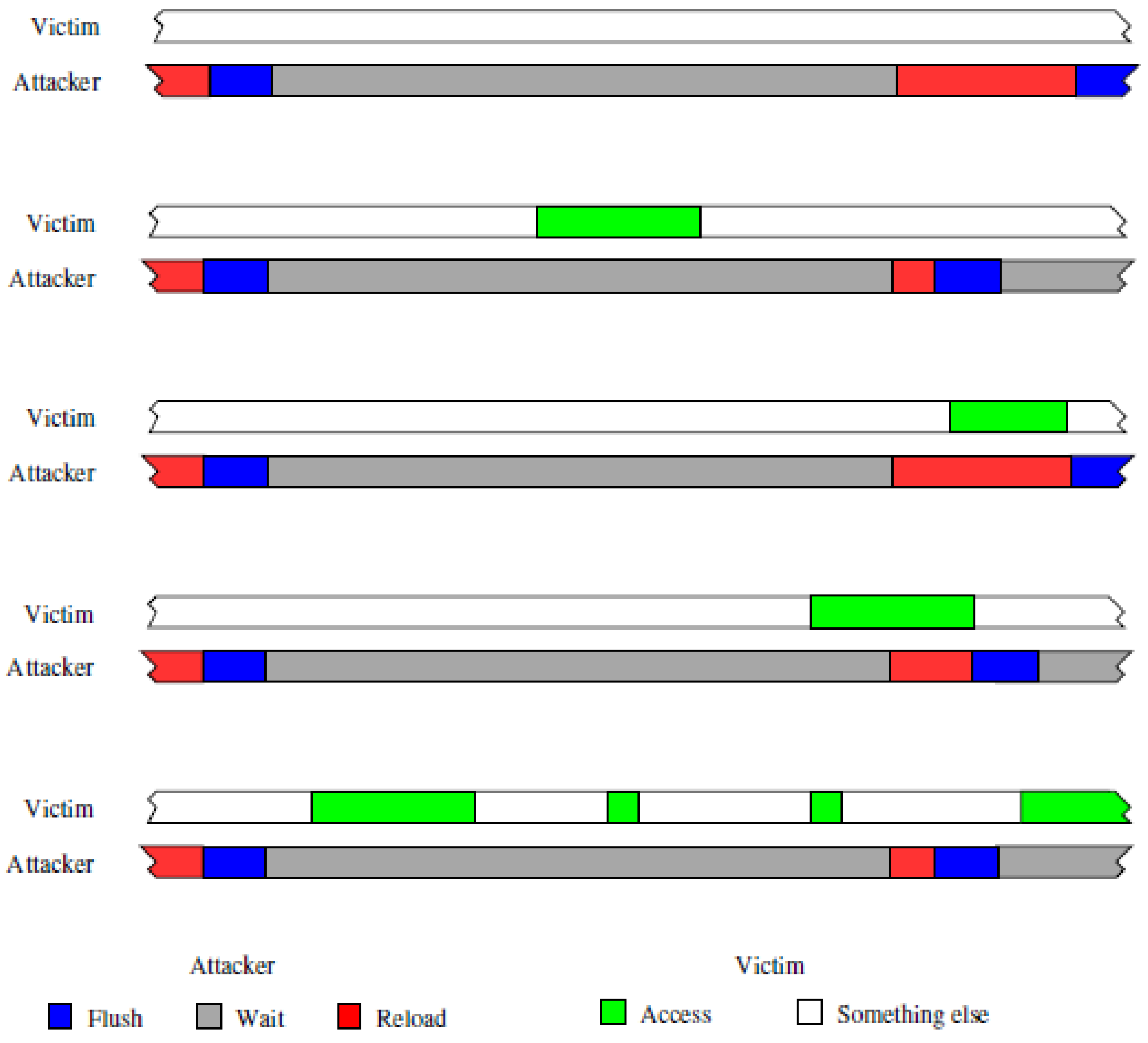


```
unsigned long probe(char* adrs){

        volatile unsigned long cycles;

        asm(
                "mfence \n"
                "lfence \n"
                "rdtsc  \n"
                "lfence \n"
                "movl %%eax,%%esi         \n"
                "movl (%1), %%eax         \n"
                "lfence \n"
                "rdtsc  \n"
                "subl %%esi, %%eax        \n"
                "clflush 0(%1)   \n"
                : "=a" (cycles)
                : "c" (adrs)
                : "%esi" , "%edx" );

        return cycles;
}
```

A barrier on all memory accesses

Barriers on loads

Typical Flush+Reload timelines



The actual meaning of reading/writing from/to memory

- What is the memory behavior under concurrent data accesses?
  - ✓ Reading a memory location should return last value written
  - ✓ **The last value written** not clearly (or univocally) defined under concurrent access

- The memory consistency model
  - ✓ Defines in which order processing units perceive concurrent accesses
  - ✓ Based on ordering rules, not necessarily timing of accesses

- Memory consistency is not memory coherency!!!

Terminology for memory models

- Program Order (of a processor‘s operations)
  - ✓ per-processor order of memory accesses determined by program (software)

- Visibility Order (of all operations)
  - ✓ order of memory accesses observed by one or more processors
  - ✓ every read from a location returns the value of the most recent write

Sequential consistency

``A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." (Lamport 1979)

Program order based memory accesses cannot be subverted in the overall sequence, so they cannot be observed to occur in a different order by a "remote" observer

An example

CPU1
[A] = 1;(a1)
[B] = 1;(b1)

CPU2
u = [B];(a2)
v = [A];(b2)

[A],[B] ... Memory
u,v ... Registers

a1,b1,a2,b2
Sequentially consistent
Visibility order does not violate program order

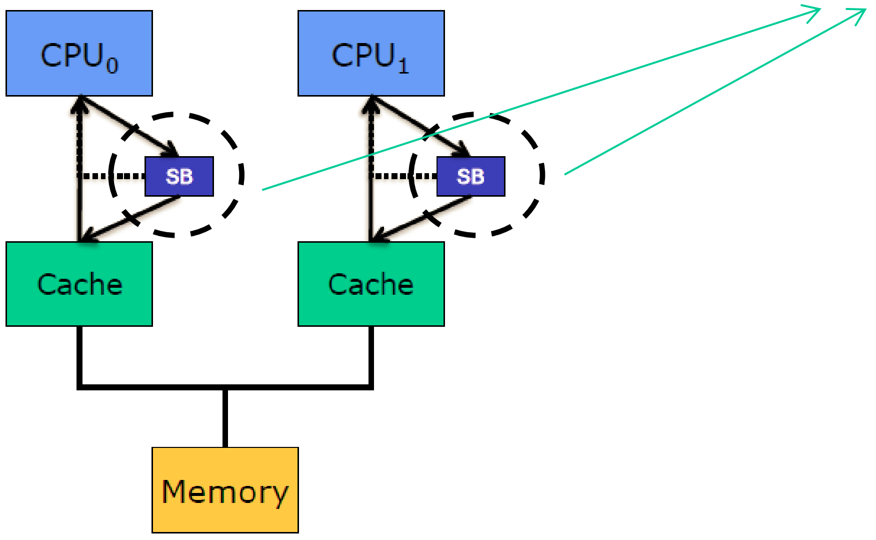
b1,a2,b2,a1
Not sequentially consistent
Visibility order violates program order

Total Store Order (TSO)

- Sequential consistency is “inconvenient” in terms of memory performance
- Example: cache misses need to be served ``sequentially” even if they are write-operations with no currently depending instruction
- <u>TSO is based on the idea that storing data into memory is not equivalent to writing to memory (as it occurs along program order)</u>
- Something is positioned in the middle between a write operation (by software) and the actual memory update (in the hardware)
- A write materializes as a store when it is ``more convenient” along time
- Several off-the-shelf machines rely on TSO (e.g. SPARC V8, x86)

TSO architectural concepts

- Store buffers allow writes to memory and/or caches to be saved to optimize interconnect accesses (e.g. when the interconnection medium is locked)
- CPU can continue execution before the write to cache/memory is complete (i.e. before data is stored)
- Some writes can be combined, e.g. video memory
- Store forwarding allows reads from local CPU to see pending writes in the store buffer
- Store buffer invisible to remote CPUs

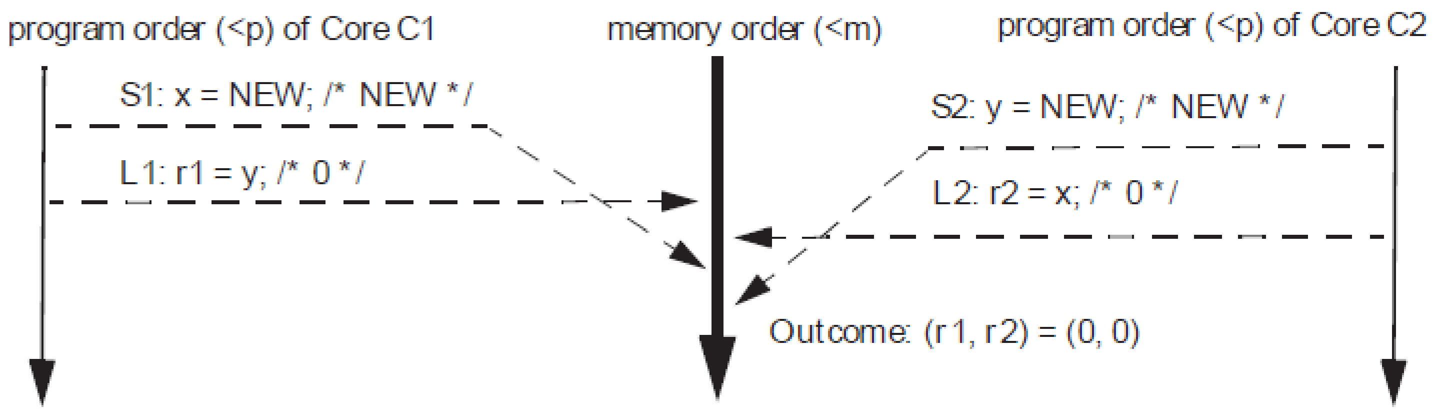


Store buffers
not directly visible
in the ISA

Forwarding of pending
writes in the store buffer
to successive read operations
of the same location

Writes become visible to
writing processor first

A TSO timeline



On x86 load operations may be reordered with older store operations to different locations

This breaks, e.g., Dekker’s mutual exclusion algorithm

x86 memory synchronization

- x86 ISA provides means for managing synchronization (hence visibility) of memory operations
- SFENCE (Store Fence) instruction:
  - ✓ Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes the SFENCE instruction in program order becomes globally visible before any store instruction that follows the SFENCE instruction.
- LFENCE (Load Fence) instruction:
  - ✓ Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE

x86 memory synchronization

- MFENCE (Memory Fence) instruction:
  - ✓ Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction
- Fences are guaranteed to be ordered with respect to any other serializing instructions (e.g. CPUID, LGDT, LIDT etc.)
- Instructions that can be prefixed by LOCK become serializing instructions
- These are ADD, ADC, AND, BTC, BTR, BTS, **CMPXCHG**, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XAND
- CMPXCHG is used by spinlocks implementations such as

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Read-Modify-Write (RMW) instructions

- More generally, CMPXCHG (historically known as Compare-and-Swap – CAS) stands in the more general class of Read-Modify-Write instructions like also Fetch-and-Add, Fetch-and-Or etc…
- These instructions perform a pattern where a value is both read and updated (if criteria are met)
- This can also be done atomically, with the guarantee of not being interfered by memory accesses by remote program flows (and related memory accesses)
- In the essence, the interconnection medium (e.g. the memory bus) is locked in favor of the processing unit that is executing the Read-Modify-Write instruction

gcc built-in

```
void _mm_sfence(void)
void _mm_lfence(void)
void _mm_mfence(void)
bool __sync_bool_compare_and_swap (type *ptr,
type oldval, type newval)
```

....

- The definition given in the Intel documentation allows only for the use of the types *int*, *long*, *long long* as well as their unsigned counterparts

- gcc will allow any integral scalar or pointer type that is 1, 2, 4 or 8 bytes in length

Implementing an active-wait barrier

```
long control_counter = THREADS;
long era_counter = THREADS;

void barrier(void){
     int ret;

     while(era_counter != THREADS && control_counter == THREADS);

     ret = __sync_bool_compare_and_swap(&control_counter,THREADS,0);
     if(ret) era_counter = 0;

     __sync_fetch_and_add(&control_counter,1);
     while(control_counter != THREADS);
     __sync_fetch_and_add(&era_counter,1);

}
```

Locks vs (more) scalable coordination

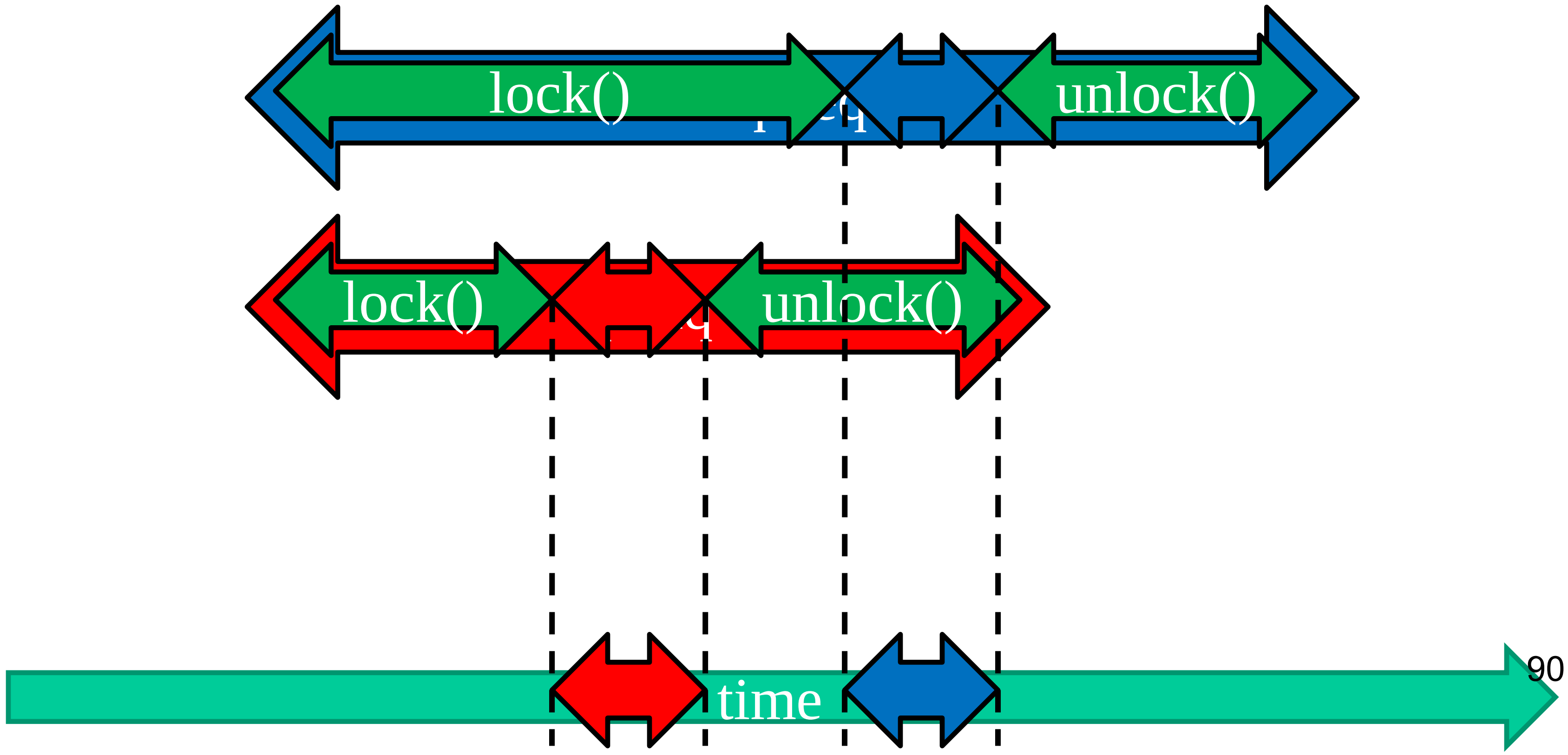
- The common way of coordinating the access to shared data is based on locks
- Up to know we understood what is the actual implementation of spin-locks
- In the end most of us <u>never cared about hardware level memory consistency</u> since spin-locks (and their Read-Modify-Write based implementation never leave) pending memory updates upon exiting a lock protected critical section
- Can we exploit memory consistency and the RMW support for achieving more scalable coordination schemes??
- The answer is yes
  - ✓ Non-blocking coordination (lock/wait-free coordination)
  - ✓ Read Copy Update (originally born within the Linux kernel)

A recall on linearizability

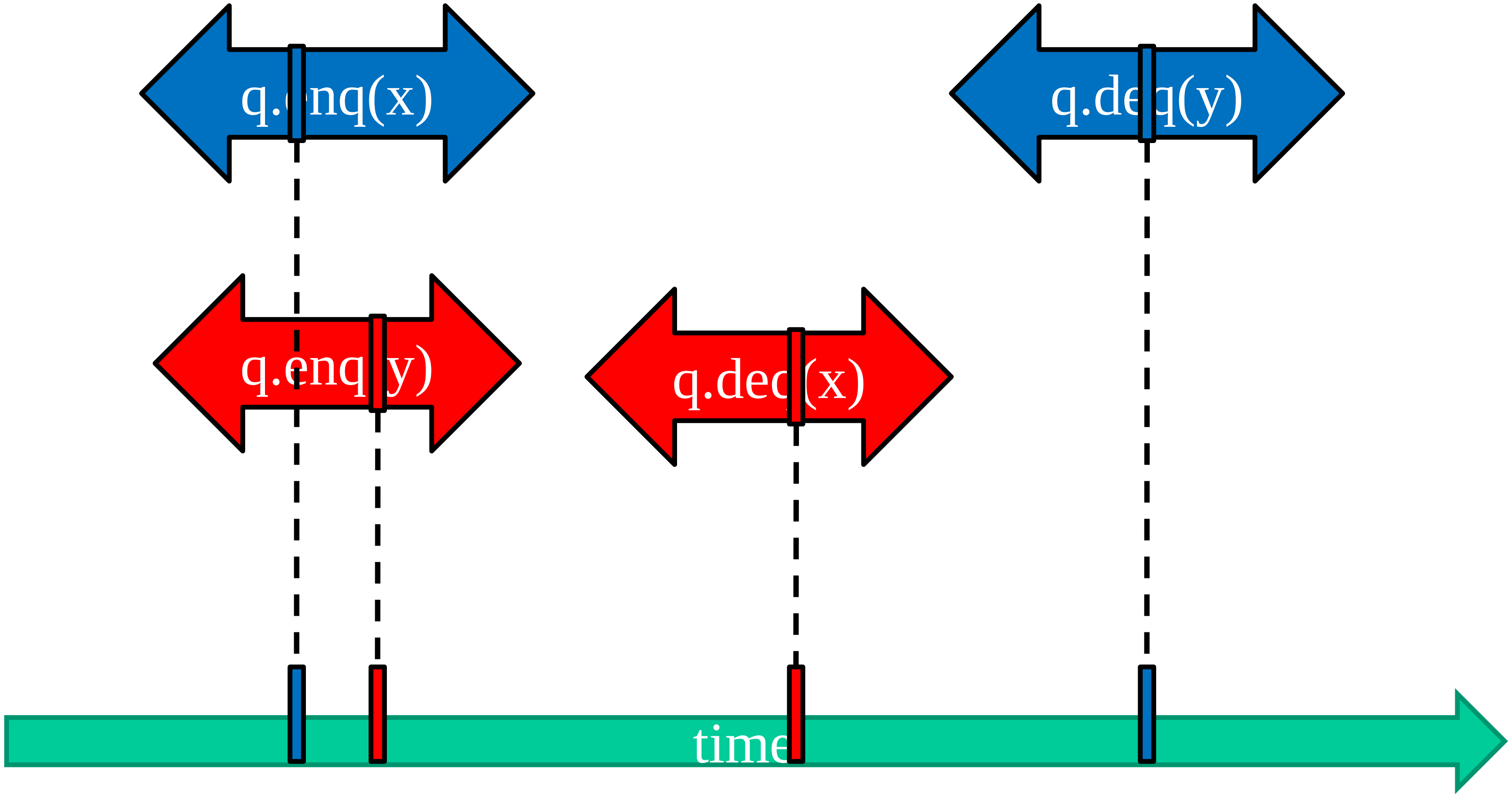
- A share data structure is “linearizable” (operation always look to be sequentializable) if
  - ✓ all its access methods/functions, although lasting a wall-clock-time period, can be seen as taking effect (materialize) at a specific point in time
  - ✓ all the time-overlapping operations can be ordered based on their “selected” materialization instant
- RMW instructions appear as atomic across the overall hardware architecture, so they can be exploited to define linearization points of operations
- Thus they can be use to order the operations
- If two ordered operations are incompatible, then one of them can be accepted, and the other one is refused (an maybe retried)
- This is the core of lock-free synchronization

RMW vs locks vs linearizability

- RMW-based locks can be used to create explicit wall clock time separation across operations
- We get therefore a sequential object with trivial linearization

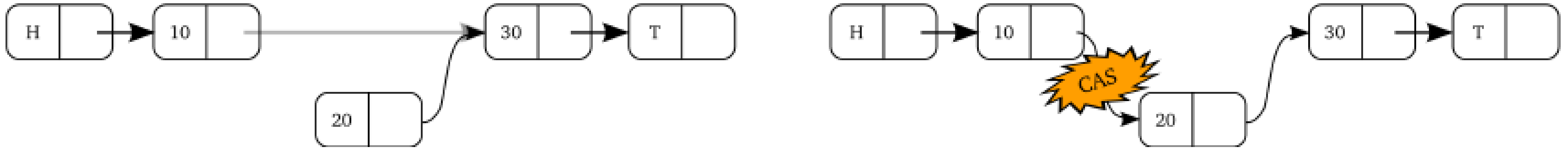


Making RMW part of the operations

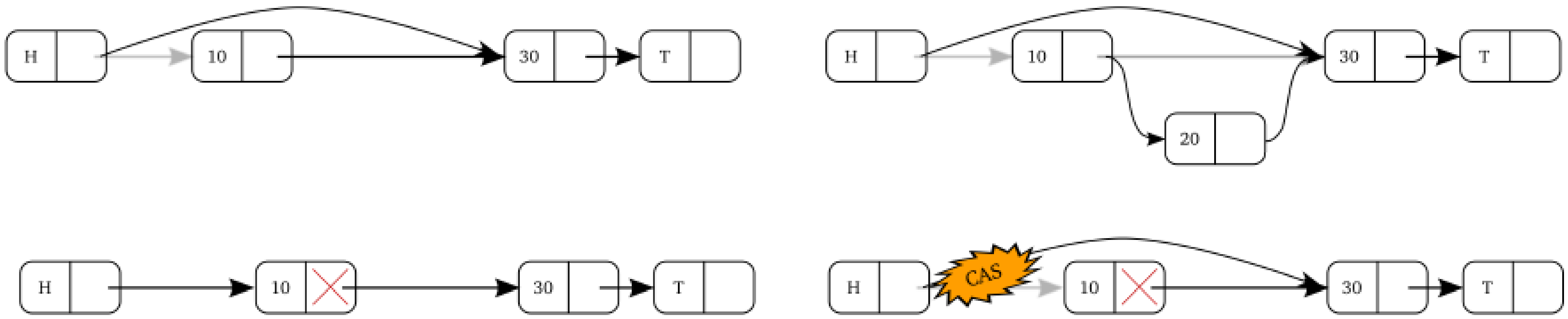


On the non-blocking linked list example

- Insert via CAS on pointers (based on failure retries)



- Remove via CAS on node-state prior to node linkage



The big problem: buffer re-usage

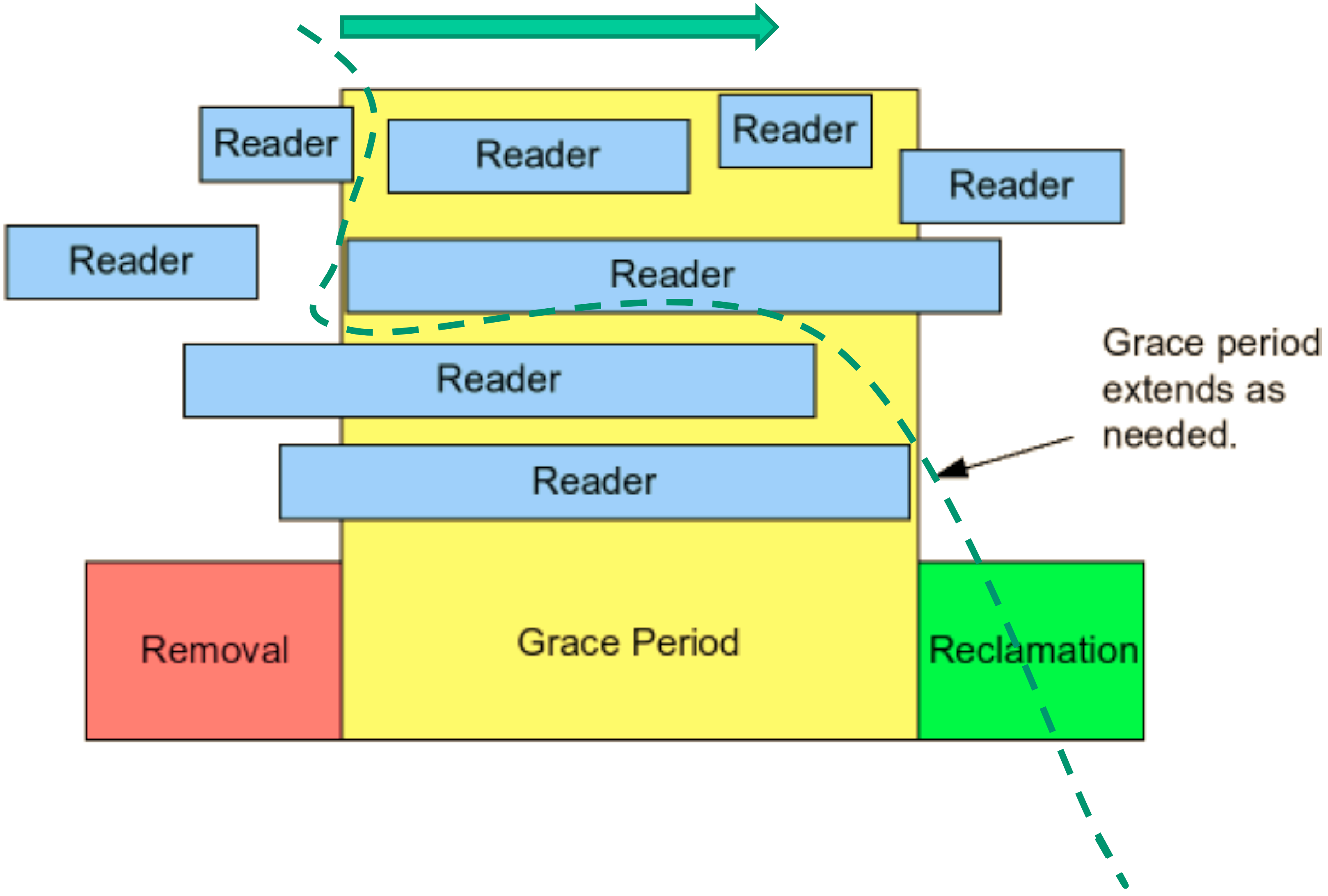
- Via CAS based approaches allow us to understand what is the state of some data structure (still in or already out of a linkage)
- But we cannot understand if traversals on that data structure are still pending
- If we reuse the data structure (e.g. modifying its fields), we might give rise to data structure breaks
- This my even lead to security problems:
  - ✓ We traverse a thread un-allowed piece of information

Read Copy Update (RCU)

- Baseline idea
  - ✓ A writer at any time
  - ✓ Concurrency between readers and writers
- Actuation
  - ✓ Out-links of logically removed data structures are not destroyed prior being sure that no reader is still traversing the modified copy of the data structure
  - ✓ Buffer re-reuse (hence release) takes place at the end of a so called "**grace period**", allowing the standing readers not linearized after the update to still proceed
- Very useful for read intensive shared data structures

General RCU timeline

Readers linearized after the writer



RCU reads and writes

- The reader
  - ✓ Signals it is there
  - ✓ It reads
  - ✓ Then it signals it is no longer there
- The writer
  - ✓ Takes a write lock
  - ✓ Updates the data structure
  - ✓ Waits for standing readers to finish
  - ✓ **NOTE: readers operating on the modified data structure instance are don’t care readers**
  - ✓ Release the buffers for re-usage

Kernel level RCU

- With non-preemptable (e.g. non-RT) kernel level configurations the reader only needs to turn off preemption upon a read and re-enable it upon finishing
- The writer understands that no standing reader is still there thanks to its own migration to all the remote CPUs, in Linux as easily as

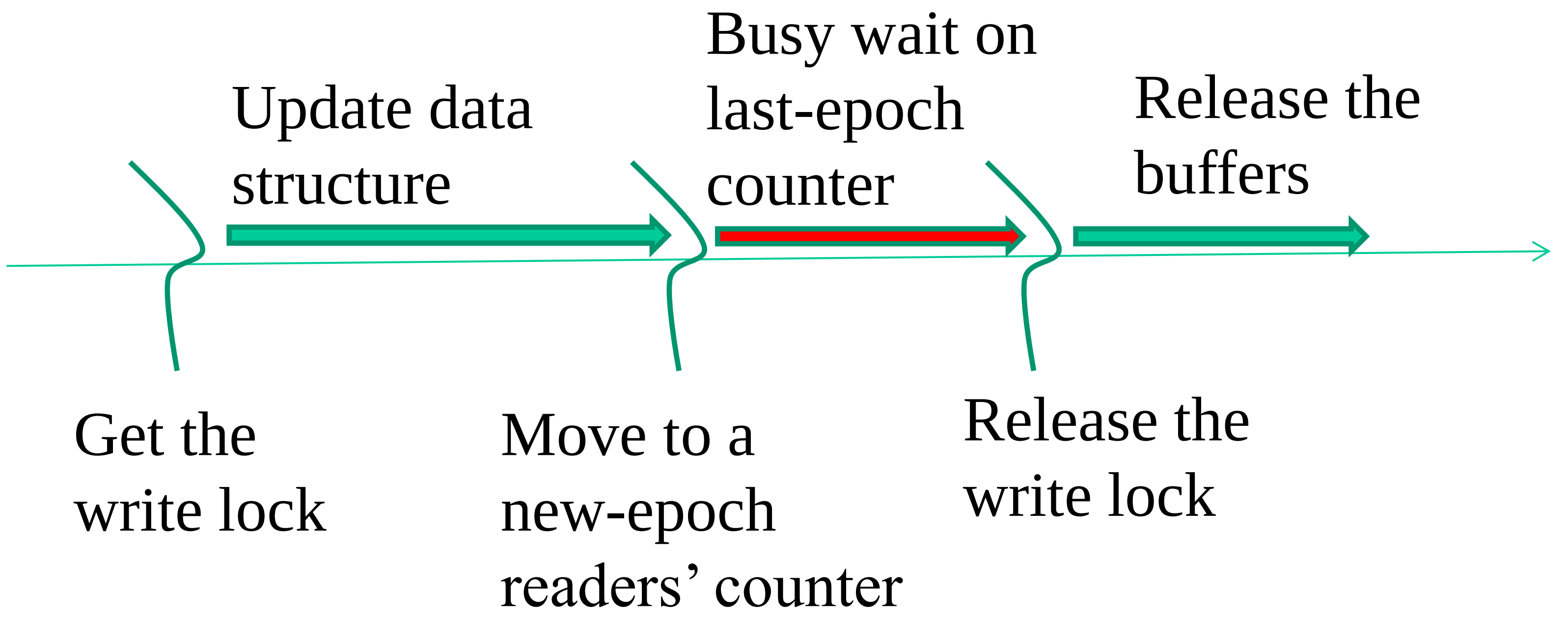
```
 for_each_online_cpu(cpu)  run_on(cpu);
```

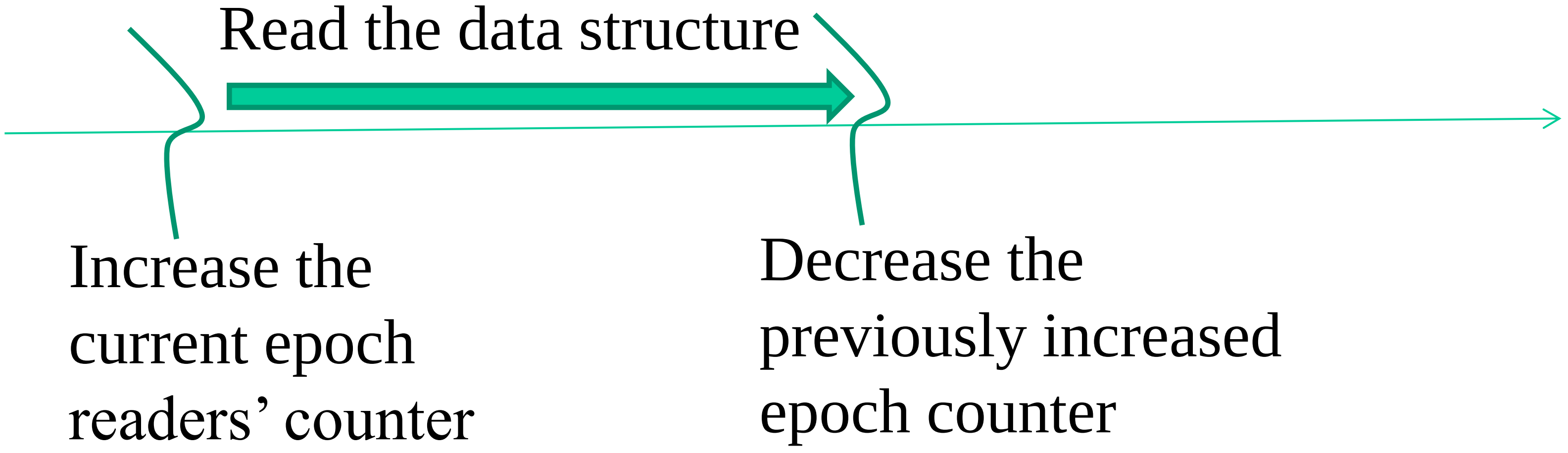
- The migrations create a context switch leading the writer to understand that no standing reader, not linearized after the writer, is still there.

Preemptable (e.g. user level) RCU

- Discovering standing readers in the grace periods is a bit more tricky
- An atomic presence-counter indicates an ongoing read
- The writer updates the data structure and redirects readers to a new presence counter (a new epoch)
- It the waits up to the release of presence counts on the last epoch counter
- Data-structure updates and epoch move are not atomic
- However, the only risk incurred is the one of waiting for some reader that already sow the new shape of the data structure, but got registered as present in the last epoch

Preemptable CRU reader/writer timeline



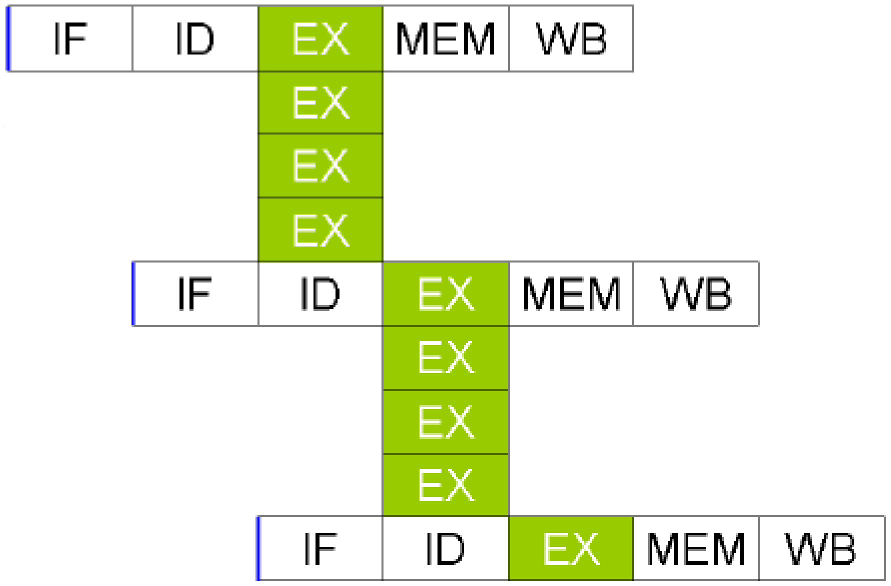

Read the data structure
Increase the current epoch readers' counter
Decrease the previously increased epoch counter

Additional parallelization aspects

- This is the so called “vectorization”
- It was born way before speculative computing and multi-processor/multi-core
- Essentially it is a form of **SIMD** (Single Instruction Multiple Data) processing
- As opposed to classical **MIMD** (Multiple Instruction Multiple Data) processing of multi-processors/multi-cores
- **SIMD** is based on vectorial registers and/or vectorial computation hardware (e.g. GPUs)
- Less common is **MISD** (although somebody says that a speculative processor is MISD)
- … **SISD** is a trivial single-core non speculative machine

The vector processor scheme

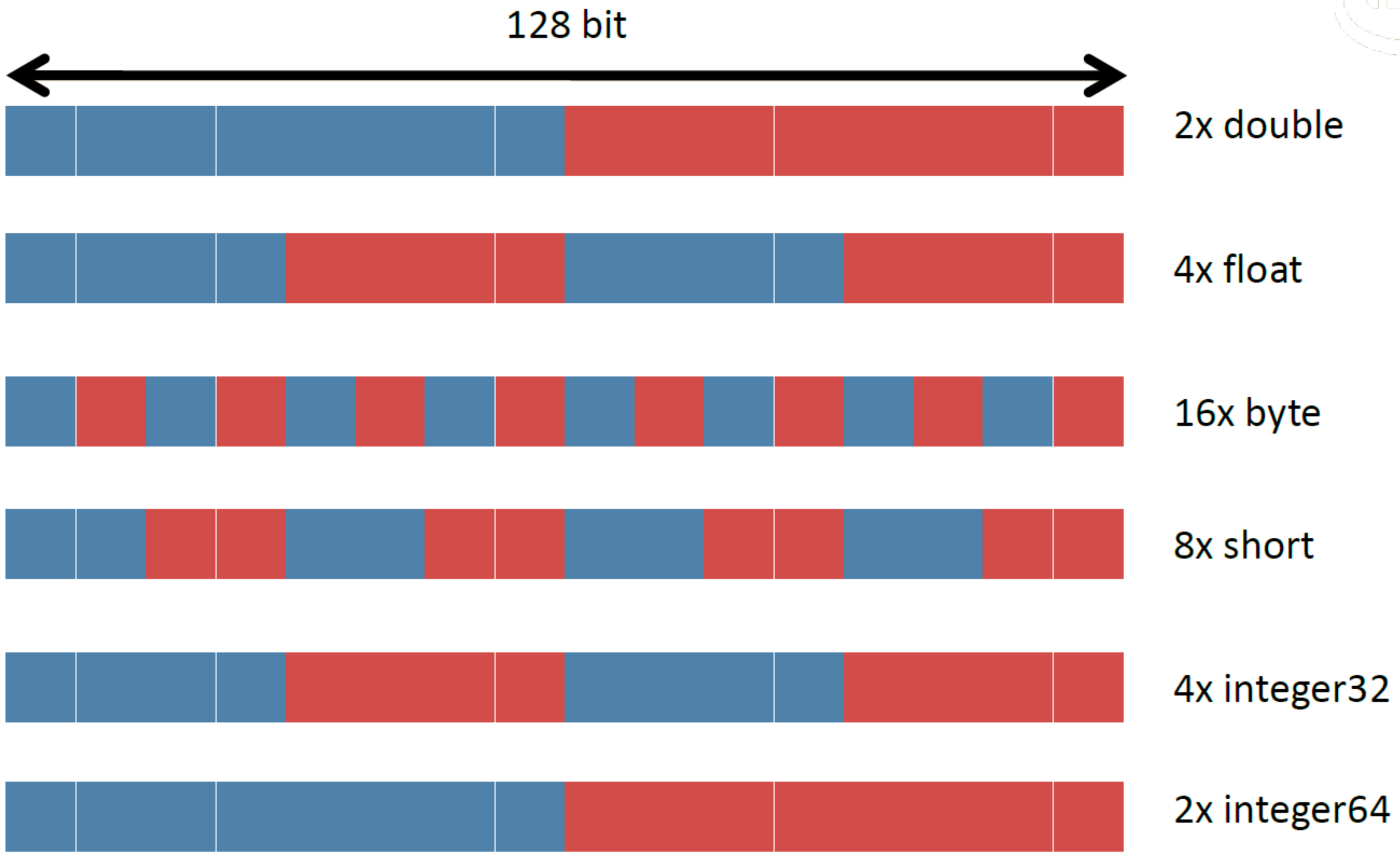
- Vector registers
- Vectorized and pipelined functional units
- Vector instructions
- Hardware data scatter/gather



x86 vectorization

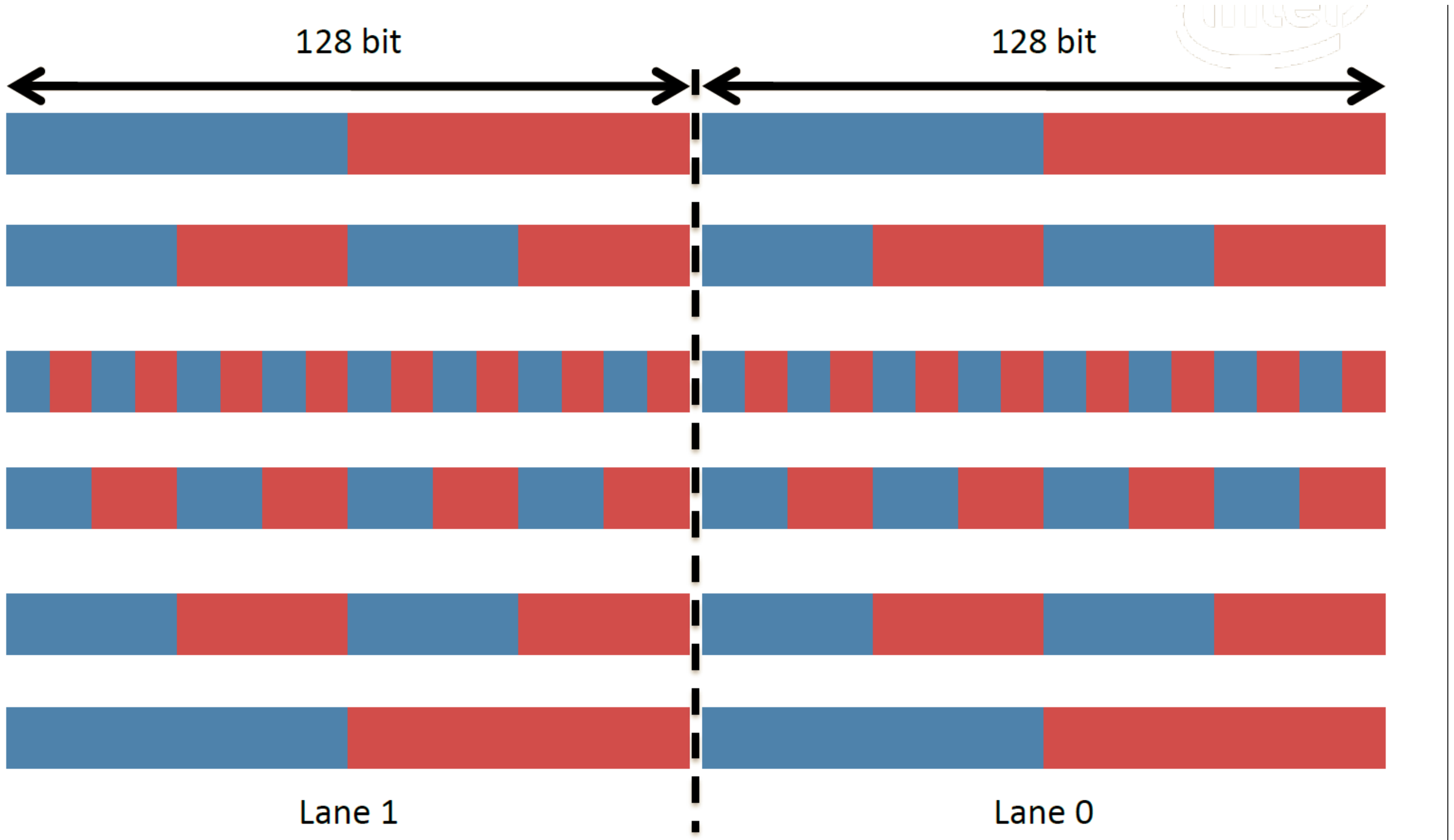
- Called SSE (Streaming SIM Extension)
- Introduced in Pentium III processors (from 1999)
- Previously called MMX (MultiMedia eXtension or Matrix Math eXtension) on Pentium processors
- In the SSE programming mode there are 8 128-bit XMM registers (16 in x86-64 SSE2) keeping vectors of
  - ✓ 2 64-bit double precision floats
  - ✓ 4 32-bit single precision floats
  - ✓ 1 128-bit integer (unsigned)
  - ✓ 2 64-bit integers (signed or unsigned)
  - ✓ 4 32-bit integers (signed or unsigned)
  - ✓ 8 16-bit integers (signed or unsigned)
  - ✓ 16 8-bit integers (signed or unsigned)

SSE data types



Sandy Bridge AVX (Advanced Vector Extensions)

- Registers are this time YMM[0-15]



Memory alignment

- Memory/XMM*/YMM* data move instructions in x86 operate with either 16-byte aligned memory or not
- Those with aligned memory are faster
- gcc offers the support for aligning static (arrays of) values via the `__attribute__ ((aligned (16)))`
- it enables compile level automatic vectorization with –O flags (originally -O2), whenever possible
- Clearly, one may also resort to dynamic memory allocation with explicit alignment
- 4KB page boundaries are intrinsically 16-bit aligned, which helps with `mmap()`
- Usage of instructions requiring alignment on non-aligned data will cause a general protection error

C intrinsics for SSE programing

- Vectorized addition - 8/16/32-bit integers

MMX(64-bit)	`d = _mm_add_pi8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi8(a, b)`	`__m128i a, b;`	`__m128i d;`

MMX(64-bit)	`d = _mm_add_pi16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi16(a, b)`	`__m128i a, b;`	`__m128i d;`

MMX(64-bit)	`d = _mm_add_pi32(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi32(a, b)`	`__m128i a, b;`	`__m128i d;`

MMX(64-bit)	`d = _mm_add_si64(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi64(a, b)`	`__m128i a, b;`	`__m128i d;`

- Vectorized addition - 32-bit floats

SSE(64-bit)	`d = _mm_add_ss(a, b)`	`__m128 a, b;`	`__m128 d;`
SSE(128-bit)	`d = _mm_add_ps(a, b)`	`__m128 a, b;`	`__m128 d;`

- Vectorized addition - 64-bit doubles

SSE2(64-bit)	`d = _mm_add_sd(a, b)`	`__m128d a, b;`	`__m128d d;`
SSE2(128-bit)	`d = _mm_add_pd(a, b)`	`__m128d a, b;`	`__m128d d;`

Additional C intrinsics

- Additional features are available for, e.g.:
  - ✓ Saturated addition
  - ✓ Subtraction
  - ✓ Saturate subtraction
  - ✓ Addition/subtraction with carry
  - ✓ Odd/even addition/subtraction
  - ✓ In vector sum reduction
- Similar functionalities are offered for the AVX case