Hardware Concurrency within a Core

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Concurrency within a Core

- So far, we've assumed that **one thread executes sequentially** on a core
  - And it's fine to think of it in this way while developing multi-threaded code
- But, under the cover, there is a lot of concurrency in the hardware
- This is done by having many hardware components in the core that can work concurrently while executing the instruction stream of one or even two threads
- In this set of lecture notes, we just review the major techniques (some extremely briefly) for your general culture/understanding

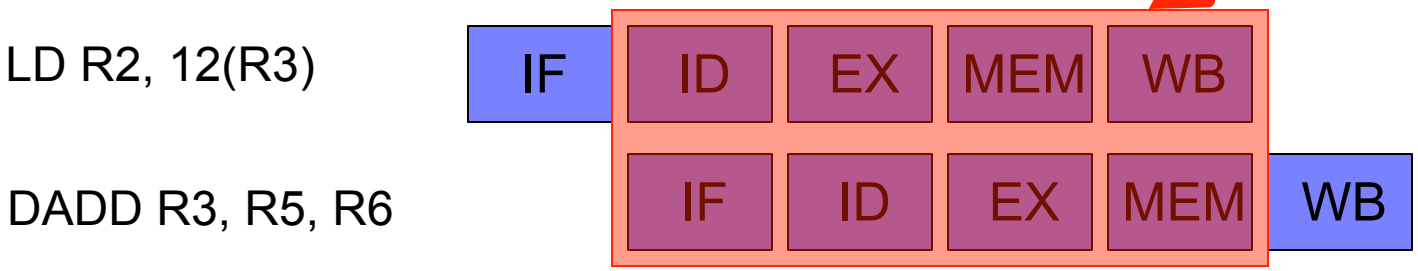
Major Techniques

- Concurrency between instructions of a single thread: Instruction-Level Parallelism (ILP)
  - Pipelining
  - Out-of-order Execution
  - Superscalar
  - Vector instructions
- Concurrency between instructions of multiple threads: Hyperthreading

ILP: Pipelining

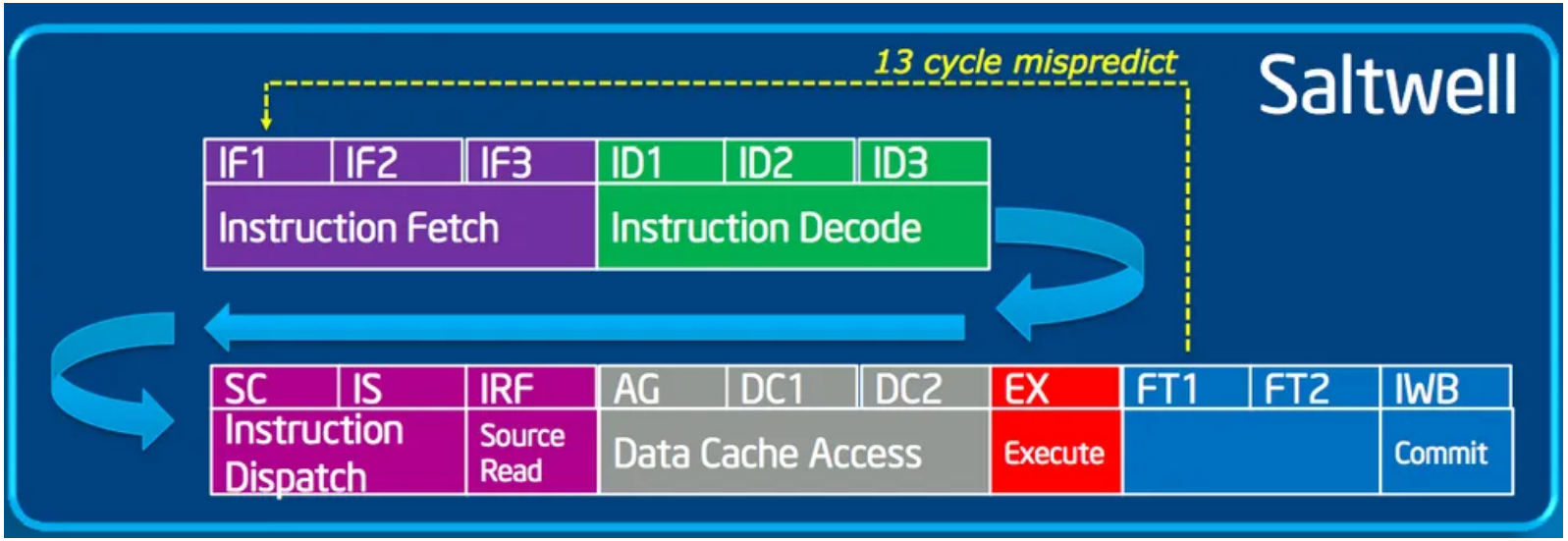
- Having all instructions doable in the same number of stages of the same durations is the RISC idea
- Example: MIPS architecture (See THE architecture book by Patterson and Hennessy)
  - 5 stages
    - Instruction Fetch (IF)
    - Instruction Decode (ID)
    - Instruction Execute (EX)
    - Memory accesses (MEM)
    - Register Write Back (WB)
  - Each stage takes one clock cycle

Concurrent execution of two instructions



ILP: Pipelining

- Modern processors have deep pipelines
- Example: Intel's Saltwell architecture



- The deeper the pipeline, the more opportunity for performance, but one has to avoid dreaded "pipeline stalls"
  - e.g., due to data dependencies between instructions

ILP: Out-of-order Execution

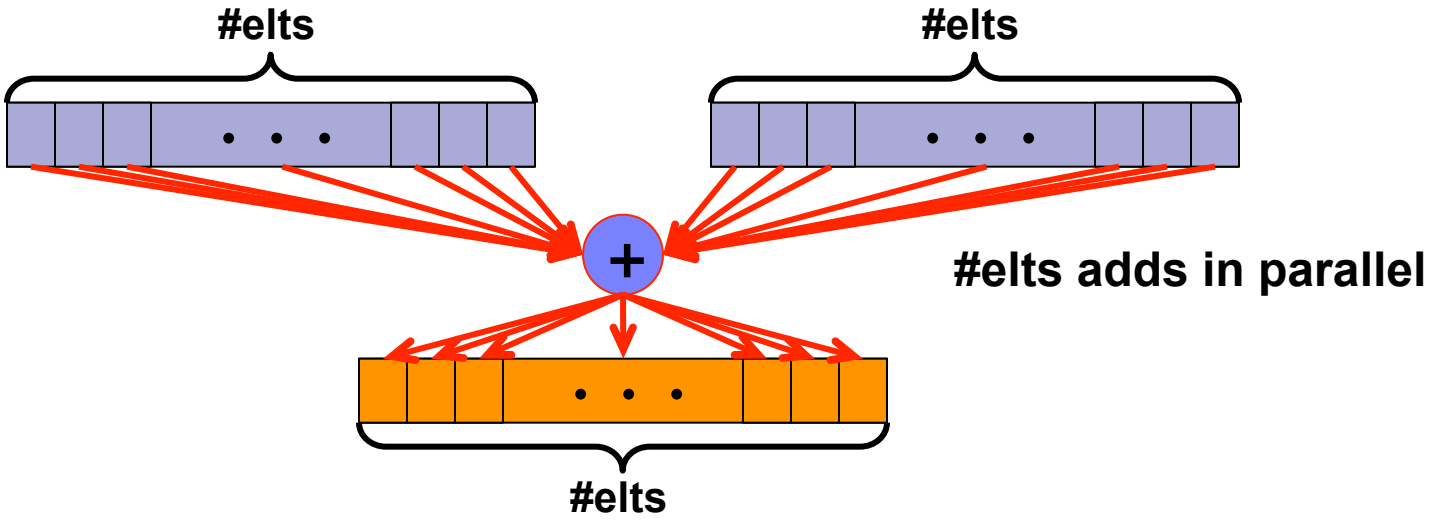
- Modern cores swap instructions at runtime to expose more parallelism while preserving correctness
- The hardware fetches multiple instructions at once, and decides (with an algorithm implemented in hardware!!) how to reorder them to achieve better performance
  - Typically, better use of the pipeline by dealing with data dependencies
- This adds a lot of complexity to the hardware
  - And makes it even more difficult for us human to figure out what our high-level, compiled code will actually be going

ILP: Superscalar

- The term superscalar is used to denote cores that can execute more than one instruction of a single thread at the same time (at least partially)
- Extra hardware is added to the processor so that multiple instructions can be issued and executed during the same cycle
  - e.g., multiple ALUs
  - e.g., multiple pipelines

ILP: Vector Units

- A functional unit that does element-wise operations on multiple values with a single “vector instruction” that operates on “vector registers” (data parallelism in hardware)



- All your compilers are now “vectorizing compilers” that will transform some of your loops into “vectorized loops”

Major Techniques

- Concurrency between instructions of a single thread: Instruction-Level Parallelism (ILP)
  - Pipelining
  - Out-of-order Execution
  - Superscalar
  - Vector instructions
- Concurrency between instructions of multiple threads: Hyperthreading

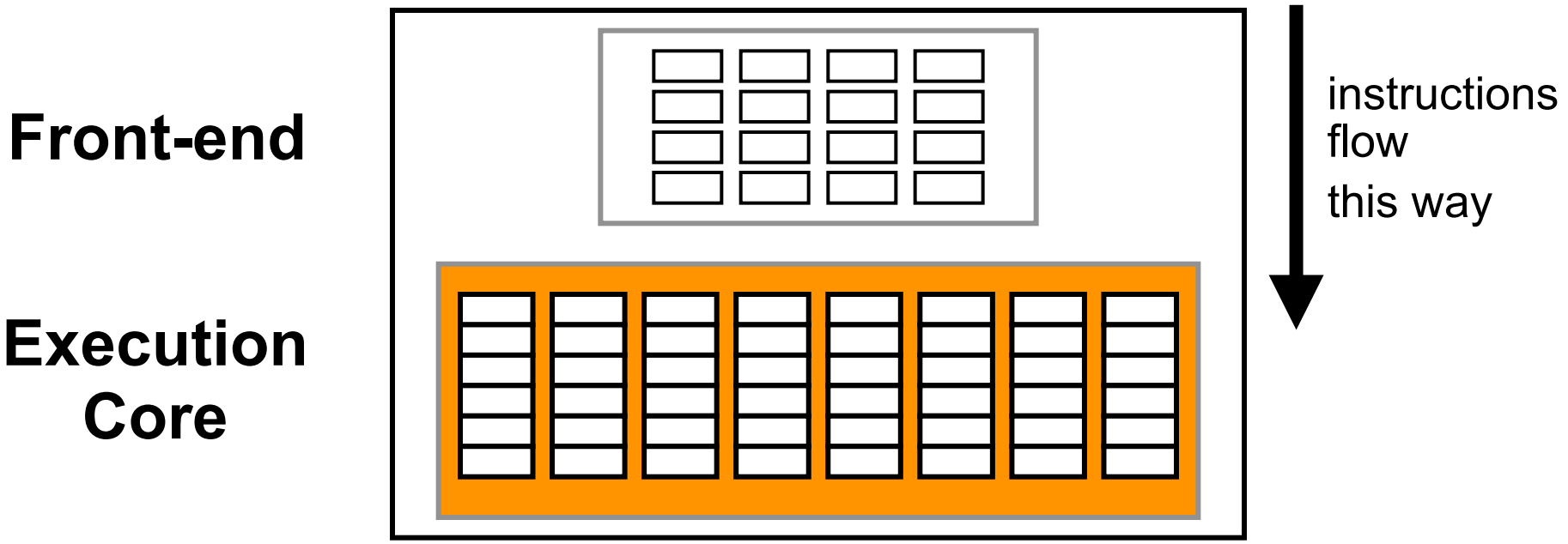
Multi-threading

- Multi-threading has been around for years, so what are we talking about here?
- Here we're talking about Hardware Support for threads, often called Simultaneous Multi-Threading (SMT)
- One can augment a core with additional hardware to support multiple threads at once, so that we no longer need to context-switch threads in and out!
  - e.g., that means multiple register files
- The idea is that one can duplicate some of the hardware in the core (but not all, otherwise it's not worth it) so that threads can happily co-exist

Front-end vs. Execution code

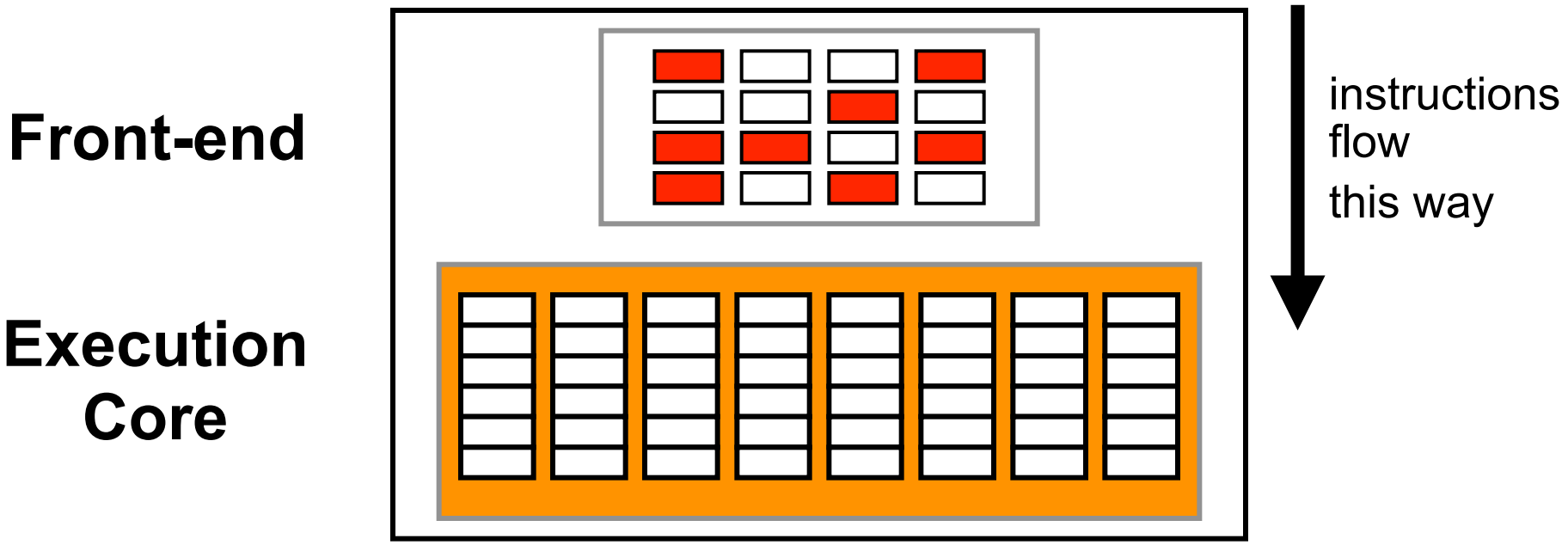
- Conceptually, CPUs split the fetch-decode-execute cycle into two big phases
  - **Front-end**: fetching/decoding/reordering of instruction
  - **Execution core**: executing bits and pieces of instructions in parallel using multiple hardware components
    - e.g., adders, etc.
- Both the front-end and the execution cores are pipelined, superscalar, out-of-order, etc.
- Let's look at the typical graphical depiction of a processor running instructions

Simplified Example Core



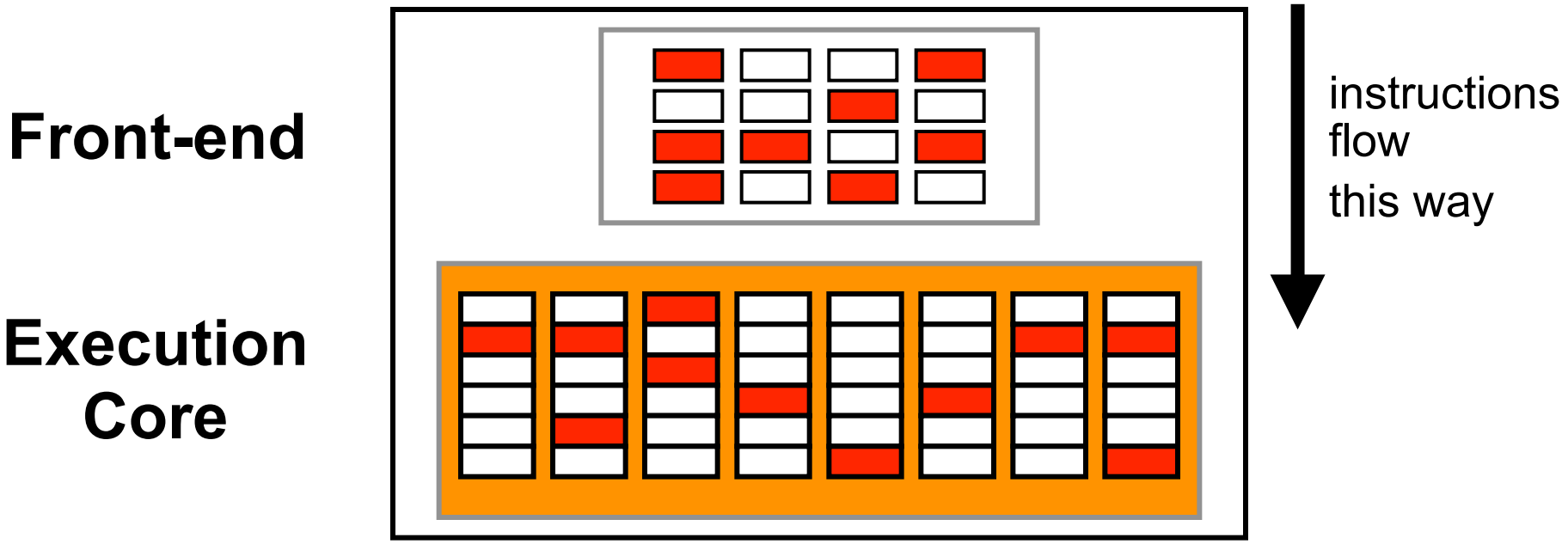
- The front-end can issue four instructions to the execution core simultaneously
  - 4-stage pipeline
- The execution core has 8 functional units
  - each a 6-stage pipeline

Simplified Example Core



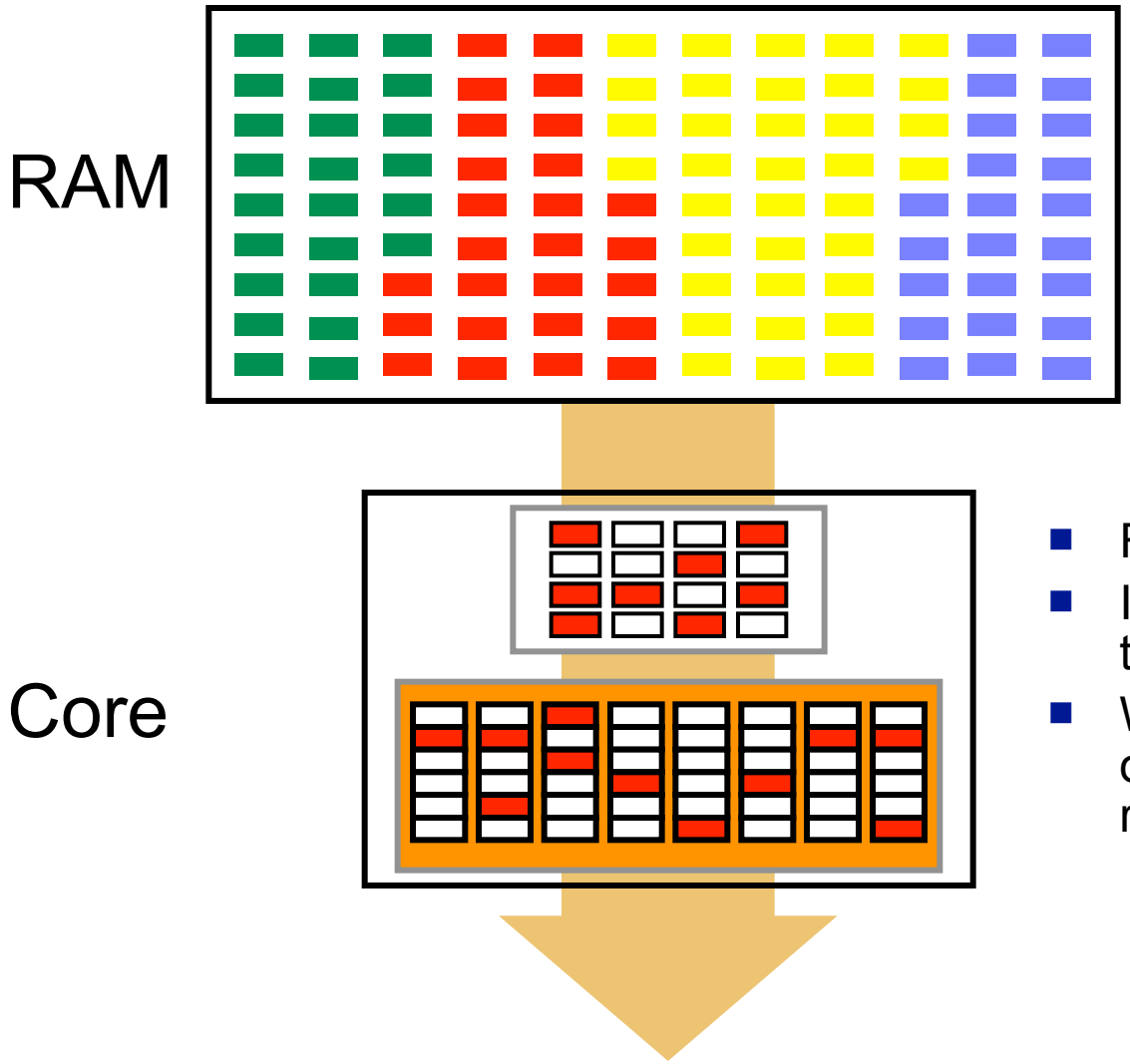
- The front-end is about to issue 2 instructions
- The cycle after it will issue 3
- The cycle after it will issue only 1
- The cycle after it will issue 2
- There is complex hardware that decides what can be issued

Simplified Example Core



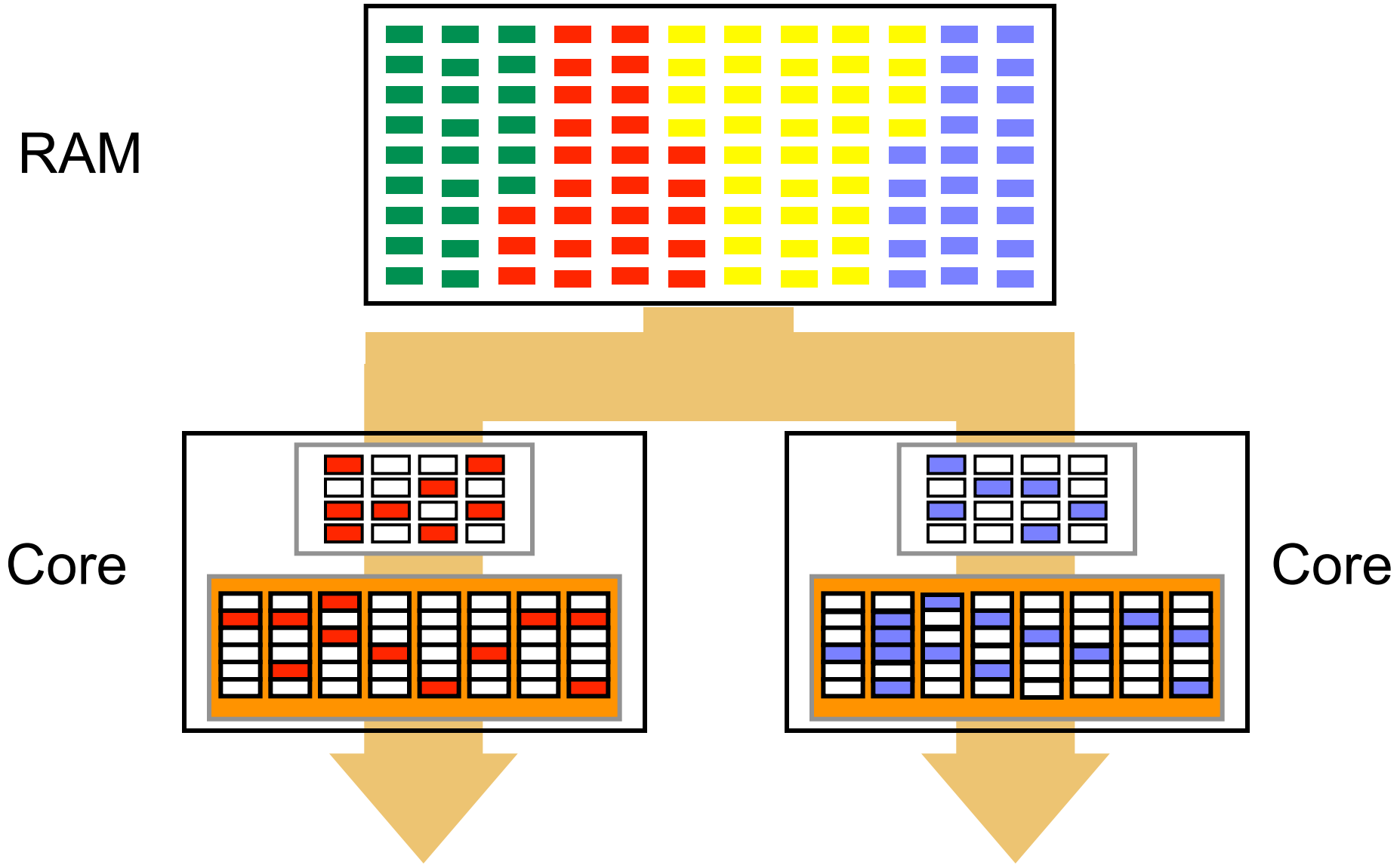
- At the current cycle, two functional units are used
- Next cycle one will be used
- And so on
- The white slots are “pipeline bubbles”: lost opportunity for doing useful work
  - Due to low instruction-level parallelism in the program

Multiple Threads in Memory

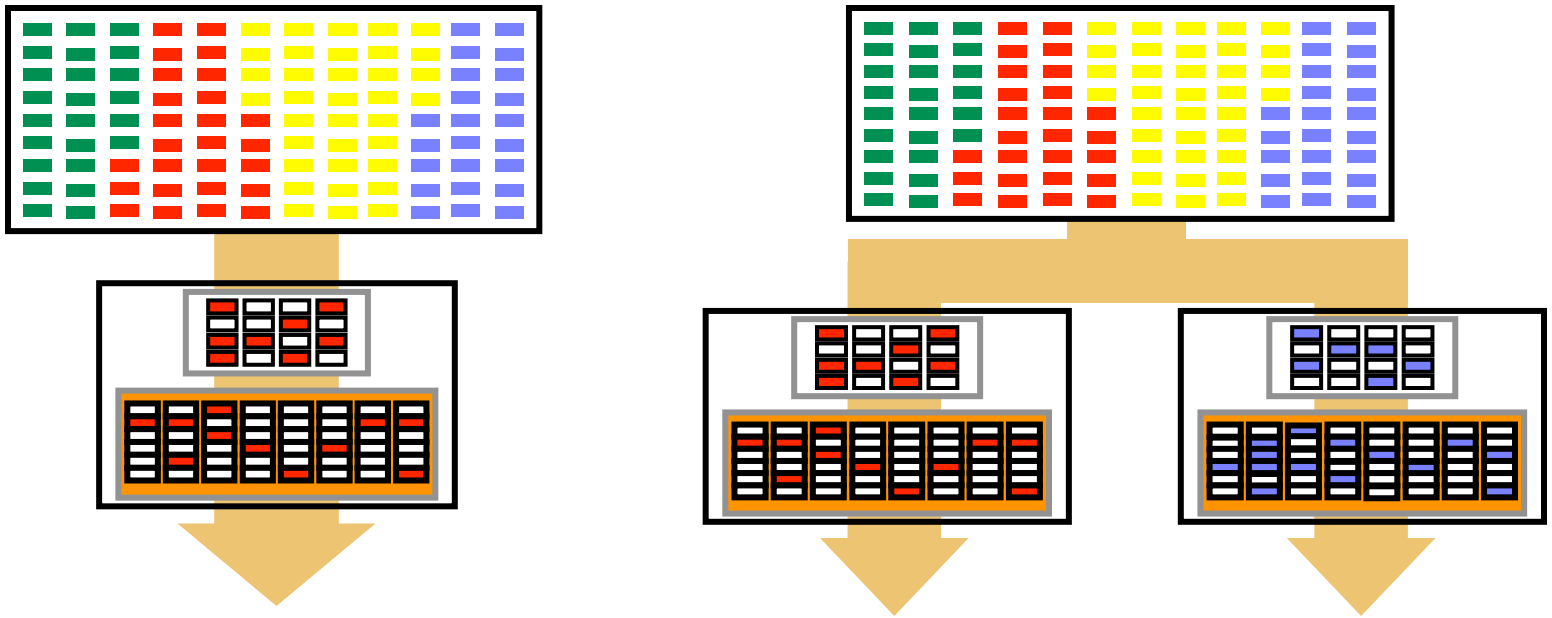


- Four threads in memory
- In a “traditional” architecture, only the “red” thread is executing
- When the O/S context switches it out, then another thread gets to run

Multi-core system



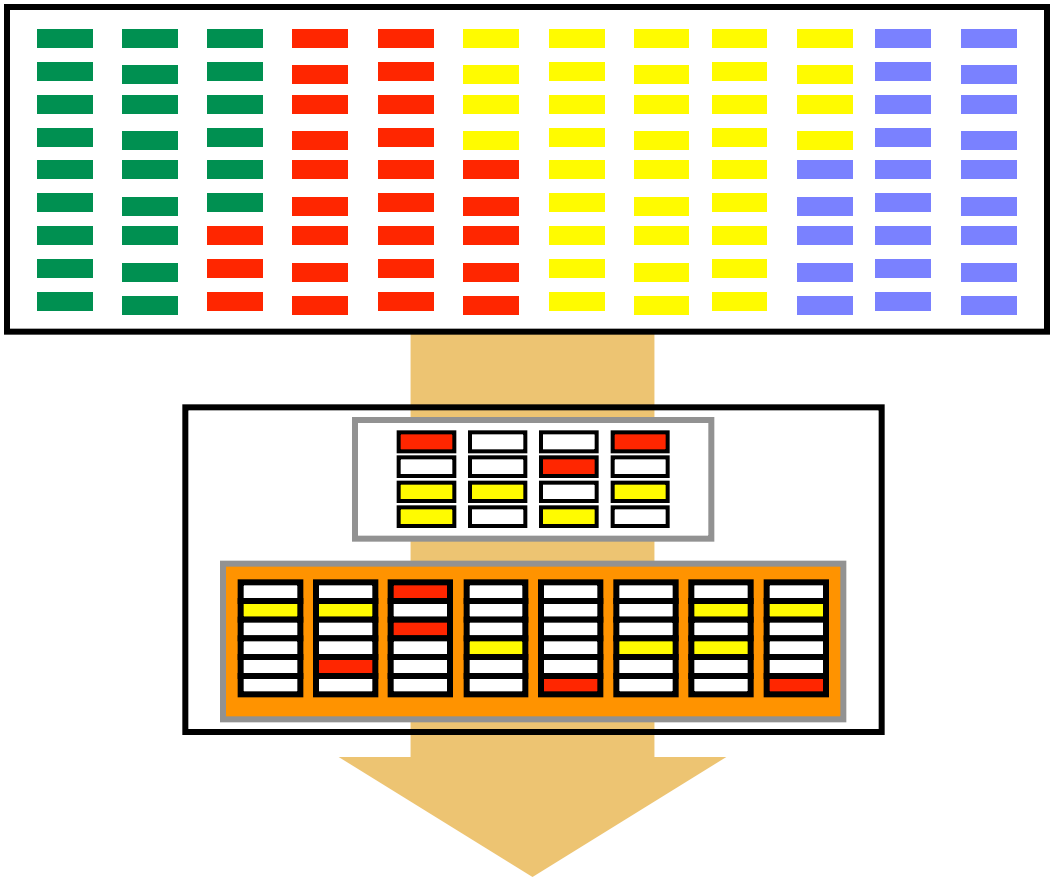
Waste of Hardware



- Both in the single-core and the dual-core systems there are many white slots
- The fraction of white slots in the system is the fraction of the hardware that is wasted
  - Adding a core does not reduce wastage
- Challenge: use more of the white slots!

Super-threading

- The idea behind “super-threading” is to allow instructions from multiple threads to be “in” the same core simultaneously

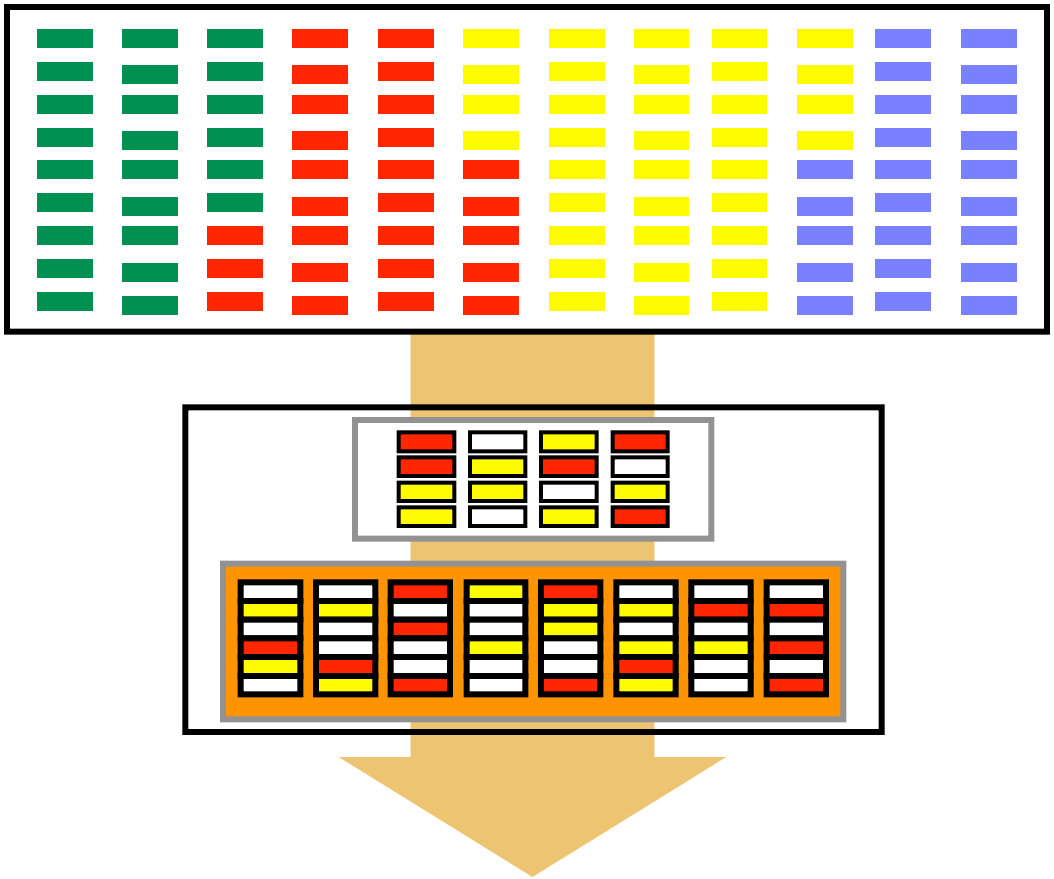


Super-threading

- Super-threading is also called “time-sliced multithreading”
- The core is then called a *multithreaded core*
- Requires more hardware cleverness
  - logic switches at each cycle
- Leads to less waste
  - e.g., a thread can run during a cycle while another thread is waiting for the memory
  - Super-threading “just” provides a finer grain of interleaving
- But there is a restriction
  - Each stage of the front end or the execution core only runs instructions from ONE thread!
- Therefore, super-threading does not help with poor instruction parallelism within one thread
  - It does not reduce bubbles within a row

Hyper-threading

- The idea behind “hyper-threading” is to allow instructions from multiple threads to execute simultaneously on the same core

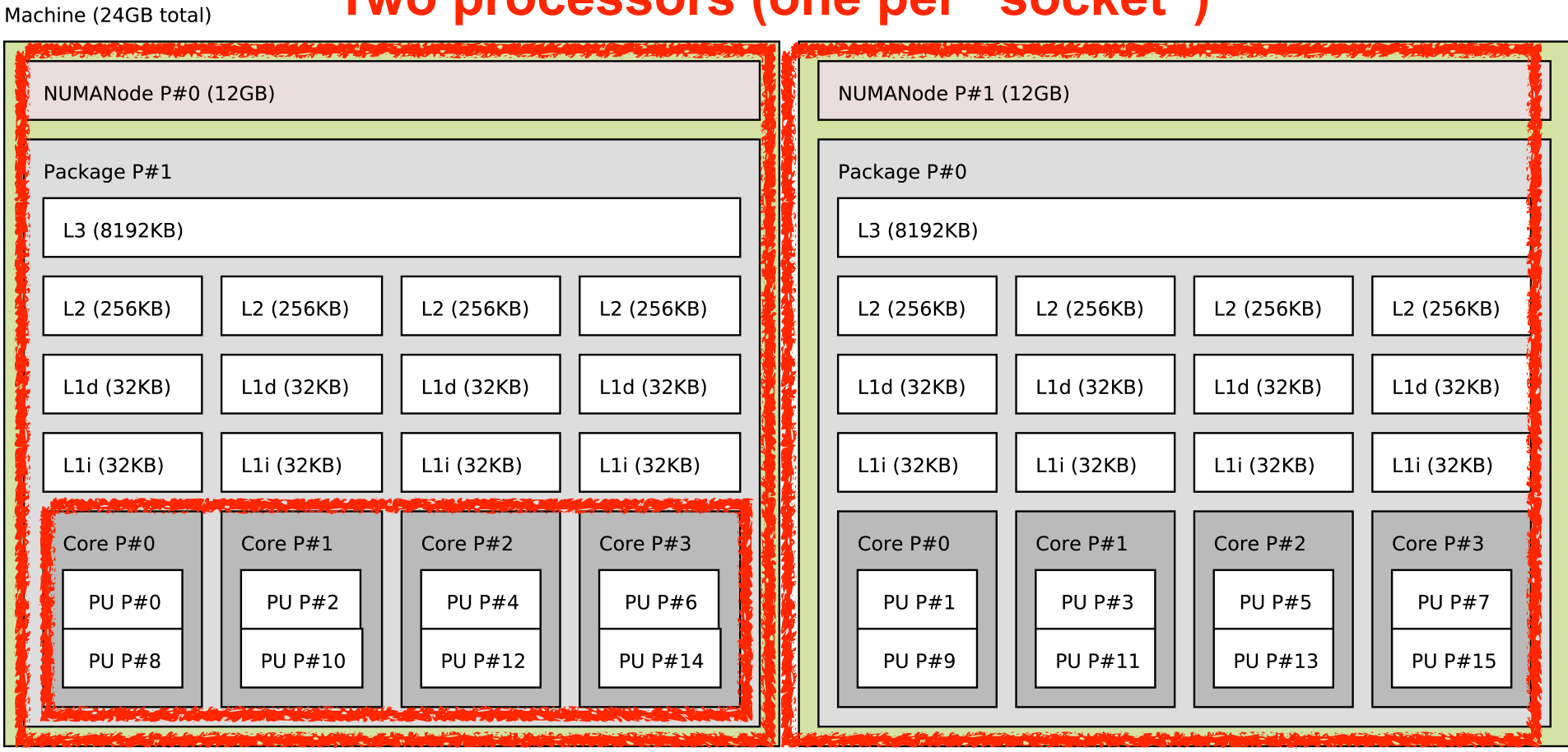


Hyper-threading

- Requires even more hardware cleverness
  - logic switches within each cycle
- In the previous example we only showed two threads executing simultaneously
  - Note that there were still white slots
- In fact, Intel's most talked about hyper-threaded processor is only for two threads
  - Intel's hyper-threading only adds 5% to the die area, therefore the performance benefit is worth it
  - Some people argue that "two" is not "hyper" ☺
  - Some "supercomputer" projects have built "massively multithreaded processors" that have hardware support for many more threads than 2
- Hyper-threading provides the finest level of interleaving
- From the OS perspective, there are two "logical" cores
  - Less performance than two physical cores
  - Less wastage than with two physical cores

My Linux Server (lstopo)

Two processors (one per “socket”)



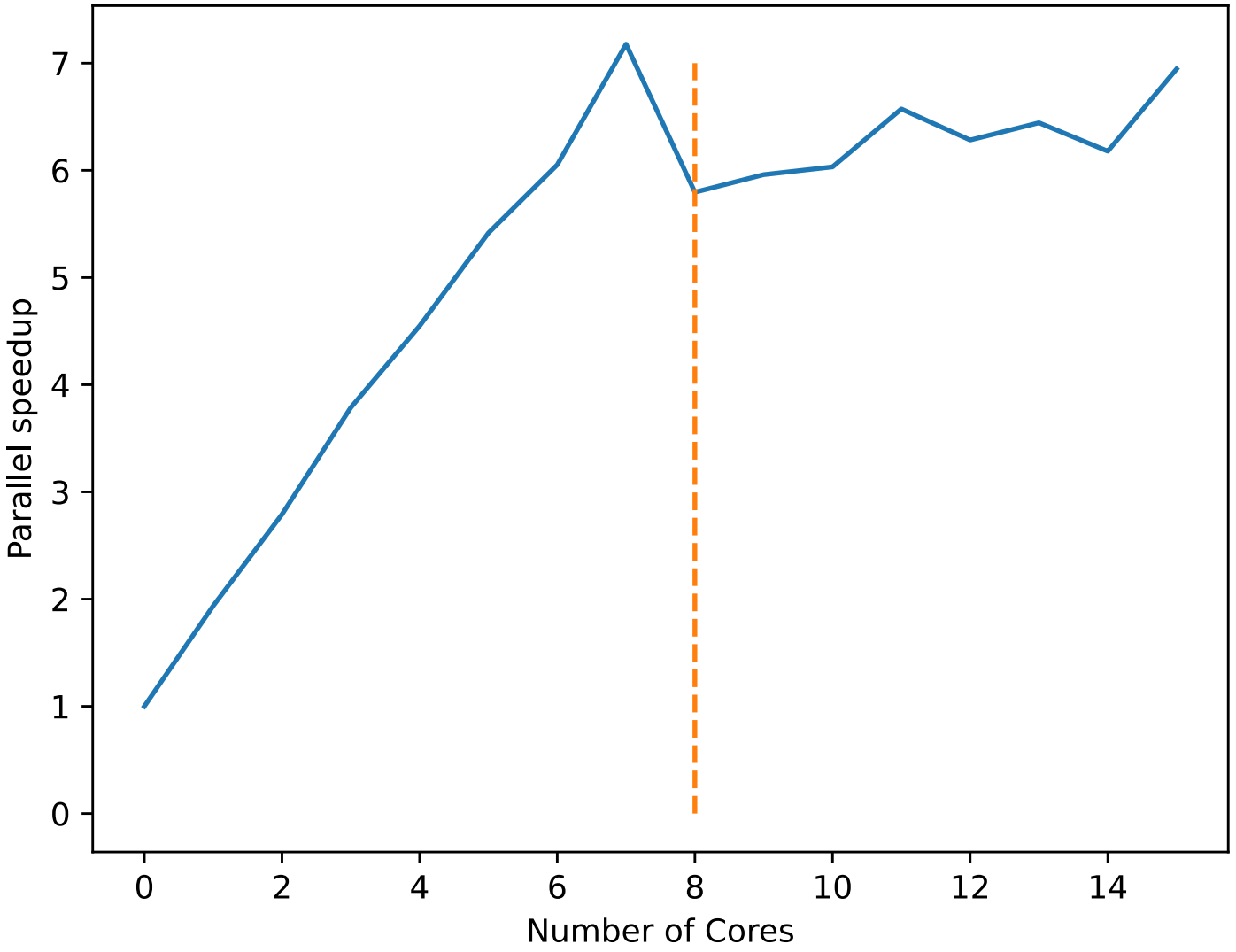
Total: 8 physical cores, each of them hyperthreaded
Looks like 16 physical cores to the OS

How Good is HyperThreading?

- Ideally, a hyperthreaded core would run two threads as fast as one thread
- In practice, a single hyperthreaded core is not as fast as two cores
- And it depends on what the threads are doing
- Let's look at results on my laptop for a simple OpenMP parallelization of

```
#pragma omp parallel for private(i,j,k)
for (i=0; i < N; i++)
  for (k=0; k < N; k++)
    for (j=0; j < N; j++)
       C[i][j] += A[i][k] * B[k][j]
```

How Good is Hyperthreading?



HyperThreading

- Hyperthreading is pure hardware and is transparent to the OS (who just sees twice as many cores)
- Its effectiveness really depends on what threads do, and it's hard to predict it
- The more different stuff threads do, the better it is
  - Our matrix-multiply case was thus not great
- It is often disabled for HPC performance measurements
- But it's sufficiently worthwhile to be a standard features in most CPUs

Conclusion

- As a developer, luckily, all these things are transparent to you and you typically benefit from them to some extent
- If you're into HPC, then you need to know what's going on so that you write your code in a way that will maximize core concurrency
  - e.g., write loops that you know the compiler will vectorize even though from your perspective they are sequential!
- See a computer architecture course for full details/explanations about the glory of our modern cores