CSE 8383 **Hesham El-Rewini**
Spring 2006

In Class Group Work

(DVD students are required to read all the sections individually)

1. Form 3 groups (2 or 3 students per group)
2. Name a group leader, who will be expected to lead the discussion
3. Spend 5 minutes individually reading Section 1 **(5 minutes)**
4. Spend 5 minutes discussing what you have read to make sure that you all understand the main idea.**(5 minutes)**
5. Repeat steps 3 & 4 for Sections 2, 3, and 4 **(total 30 minutes)**
6. With the help of the entire group, the group leader will prepare a short presentation summarizing the most important point that you got out of this. **(5 minutes)**
7. Each group will be given 5 minutes to make the presentation

---

Introduction to Multithreading, Superthreading and Hyperthreading

(From an article by Jon Stokes, October 3, 2002)

1. Conventional multithreading

Quite a bit of what a CPU does is illusion. For instance, modern out-of-order processor architectures don't actually execute code sequentially in the order in which it was written. I've covered the topic of out-of-order execution (OOE) in previous articles, so I won't rehash all that here. I'll just note that an OOE architecture takes code that was written and compiled to be executed in a specific order, reschedules the sequence of instructions (if possible) so that they make maximum use of the processor resources, executes them, and then arranges them back in their original order so that the results can be written out to memory. To the programmer and the user, it looks as if an ordered, sequential stream of instructions went into the CPU and identically ordered, sequential stream of computational results emerged. Only the CPU knows in what order the program's instructions were actually executed, and in that respect the processor is like a black box to both the programmer and the user.

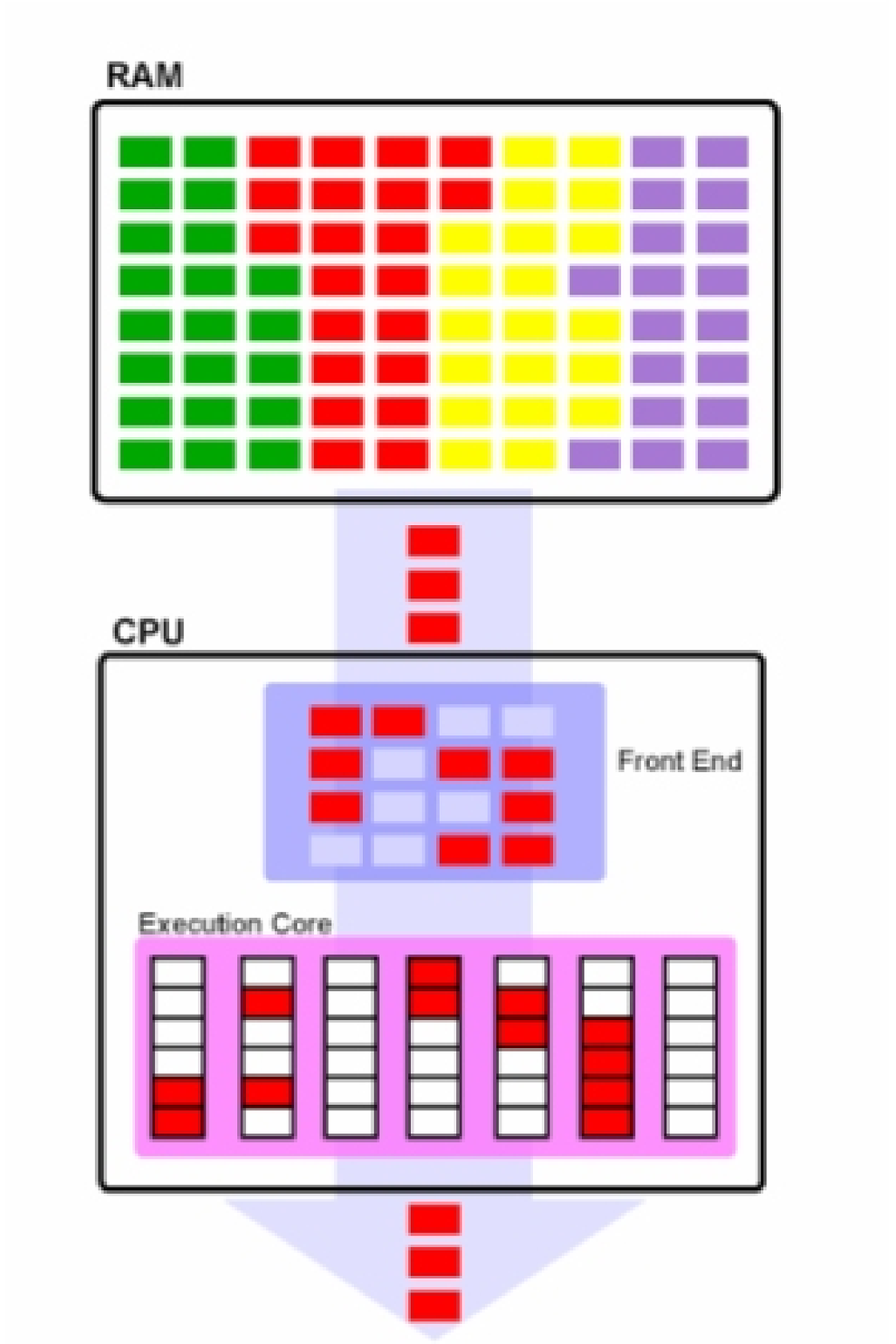
The same kind of sleight-of-hand happens when you run multiple programs at once, except this time the operating system is also involved in the scam. To the end user, it appears as if the processor is "running" more than one program at the same time, and indeed, there actually are multiple programs loaded into memory. But the CPU can *execute* only one of these programs at a time. The OS maintains the illusion of concurrency by rapidly switching between running programs at a fixed interval, called a **time slice**. The time slice has to be small enough that the user doesn't notice any degradation in the usability and performance of the running programs, and it has to be large enough that each program has a sufficient amount of CPU time in which to get useful work done. Most modern operating systems include a way to change the size of an individual program's time slice. So a program with a larger time slice gets more actual execution time on the CPU relative to its lower priority peers, and hence it runs faster.

Front-end & Execution Core

The thing that I should clarify before proceeding is that the way that I divide up the processor in this and other articles differs from the way that Intel's literature divides it. Intel will describe its processors as having an "in-order front end" and an "out-of-order execution engine." This is because for Intel, the front-end consists mainly of the instruction fetcher and decoder, while all of the register rename logic, out-of-order scheduling logic, and so on is considered to be part of the "back end" or "execution core." The way that I and many others draw the line between front-end and back-end places all of the out-of-order and register rename logic in the front end, with the "back end"/"execution core" containing only the execution units themselves and the retire logic. So in this article, the front end is the place where instructions are fetched, decoded, and re-ordered, and the execution core is where they're actually executed and retired.

Each program has a mind of its own

The OS and system hardware not only cooperate to fool the user about the true mechanics of multi-tasking, but they cooperate to fool each running program as well. While the user thinks that all of the currently running programs are being executed simultaneously, each of those programs thinks that it has a monopoly on the CPU and memory. As far as a running program is concerned, it's the only program loaded in RAM and the only program executing on the CPU. The program believes that it has complete use of the machine's entire memory address space and that the CPU is executing it continuously and without interruption. Of course, none of this is true. The program actually shares RAM with all of the other currently running programs, and it has to wait its turn for a slice of CPU time in order to execute, just like all of the other programs on the system.



Single-threaded CPU

In the above diagram, the different colored boxes in RAM represent instructions for four different running programs. As you can see, only the instructions for the red program are actually being executed right now, while the rest patiently wait their turn in memory until the CPU can briefly turn its attention to them.