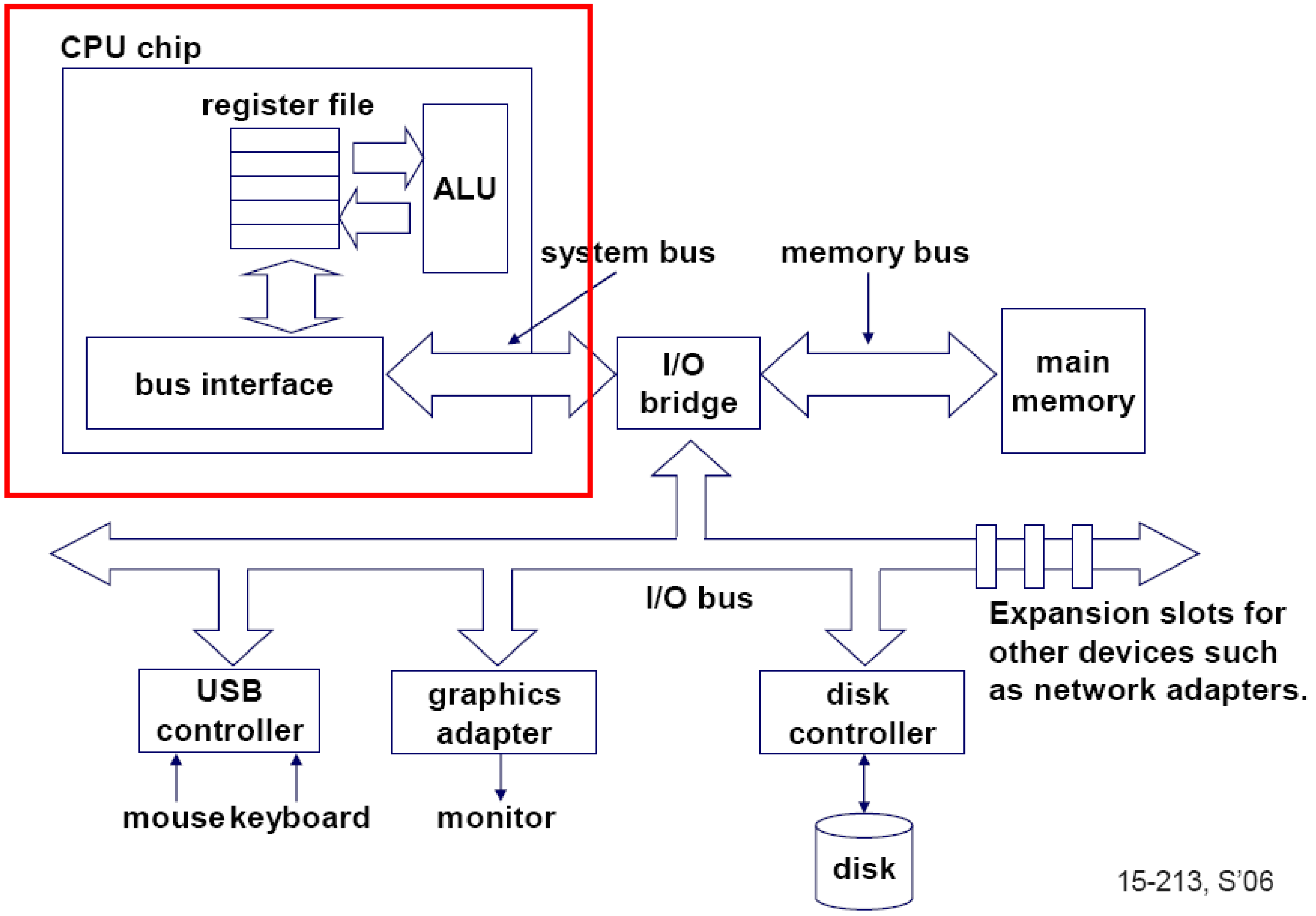
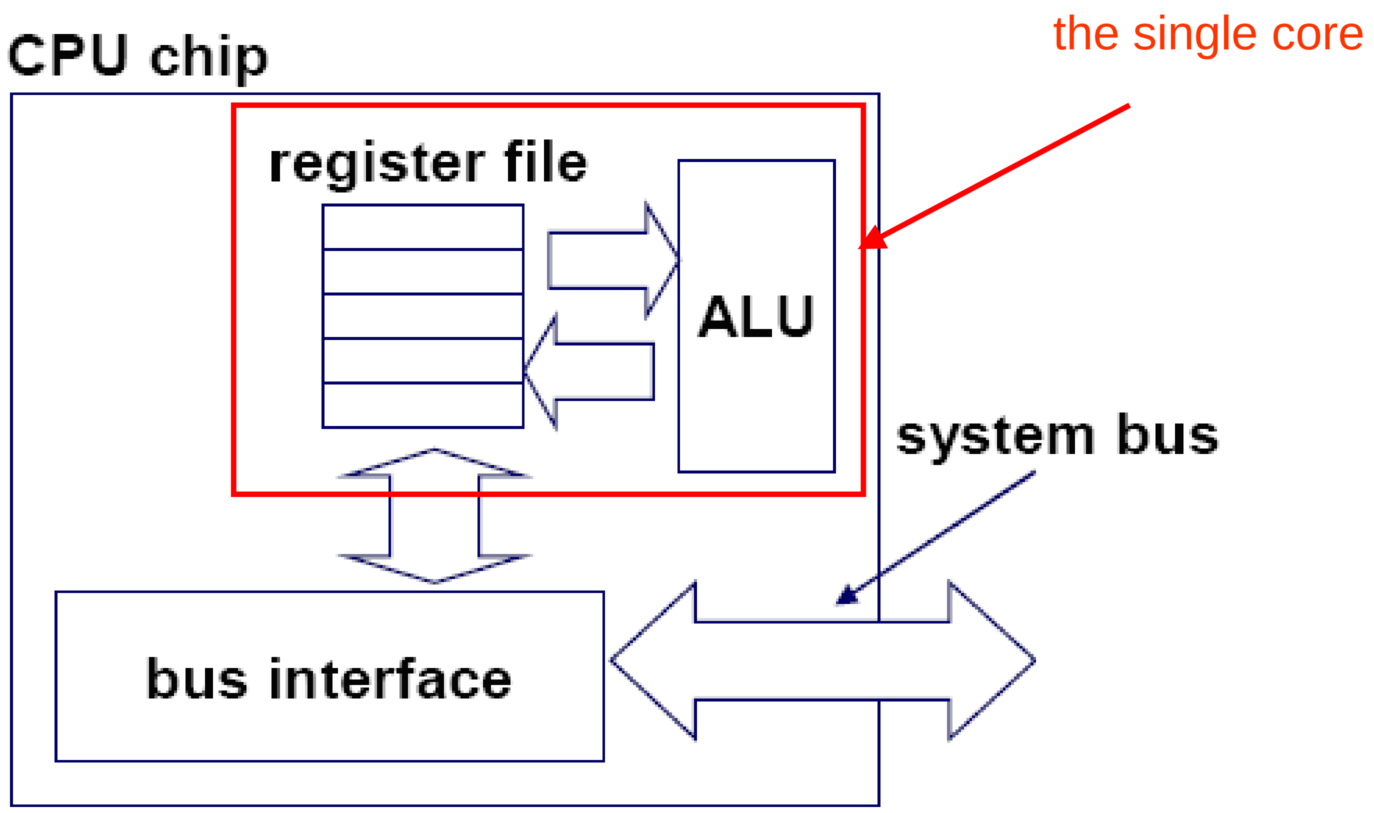
Single-core computer

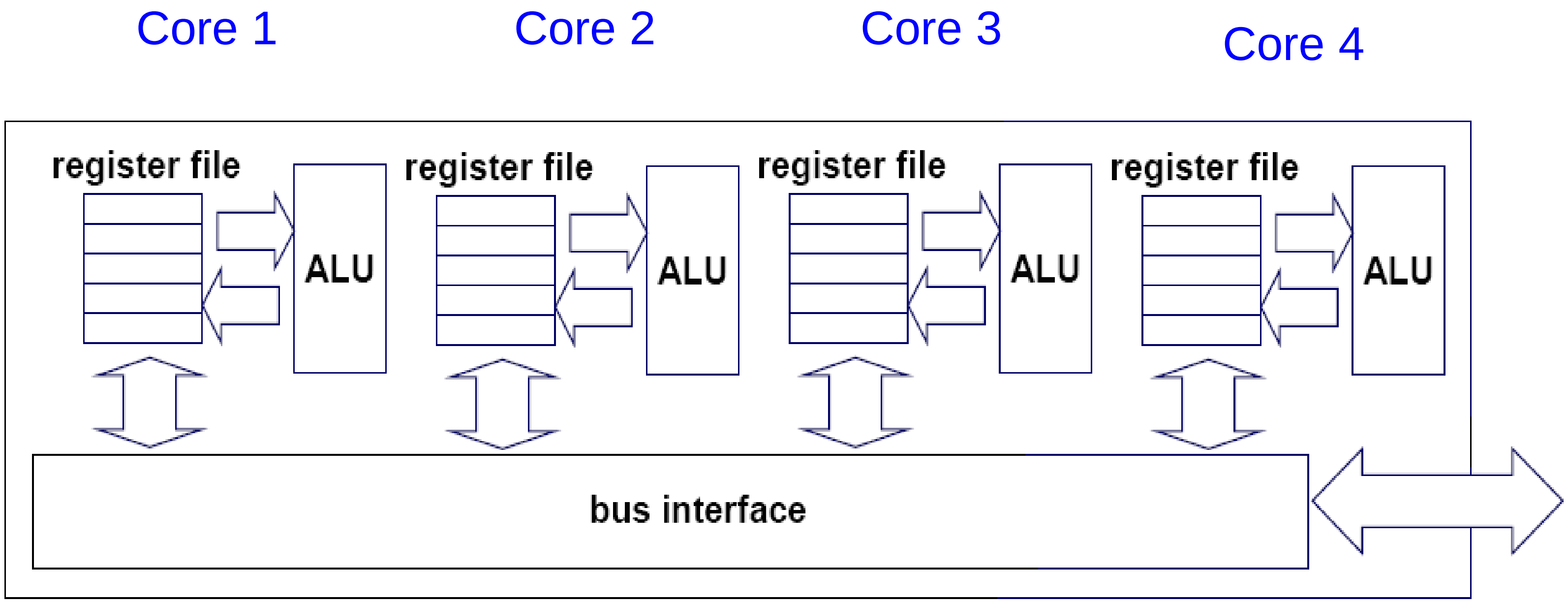


Single-core CPU chip



Multi-core architectures

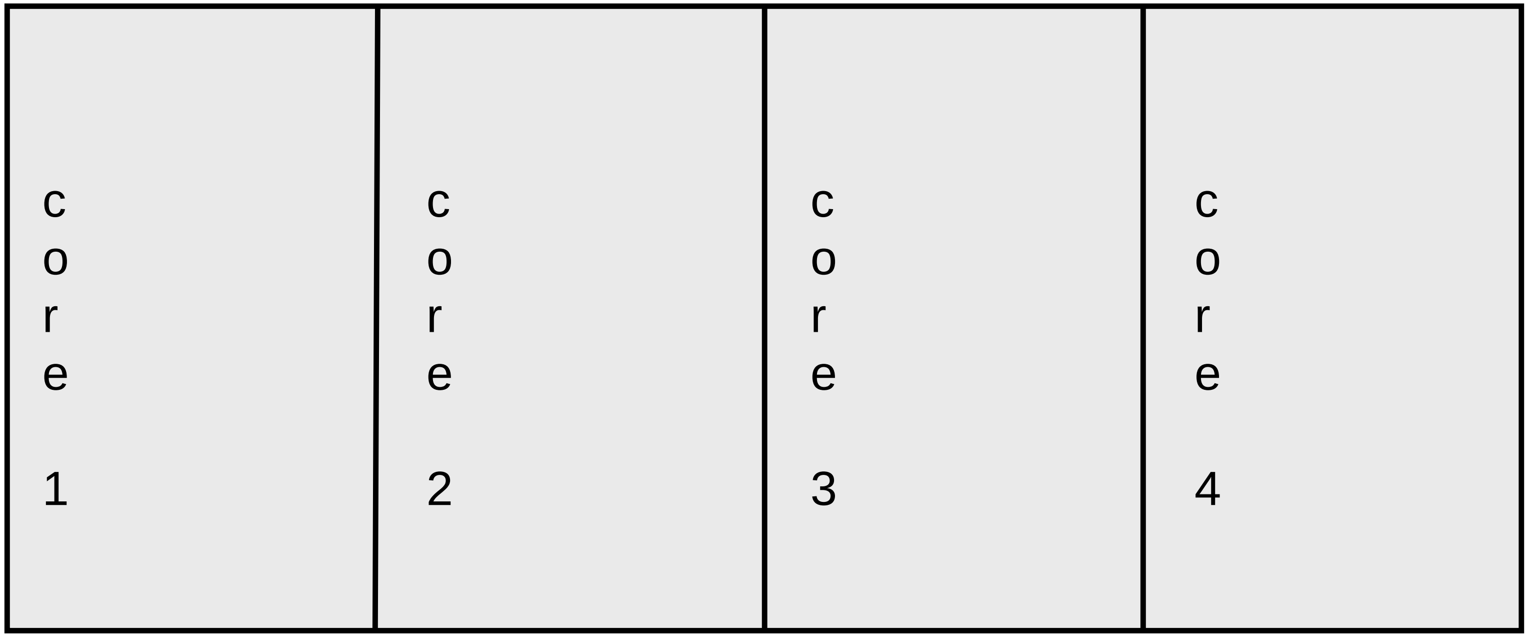
- Replicate multiple processor cores on a single die.



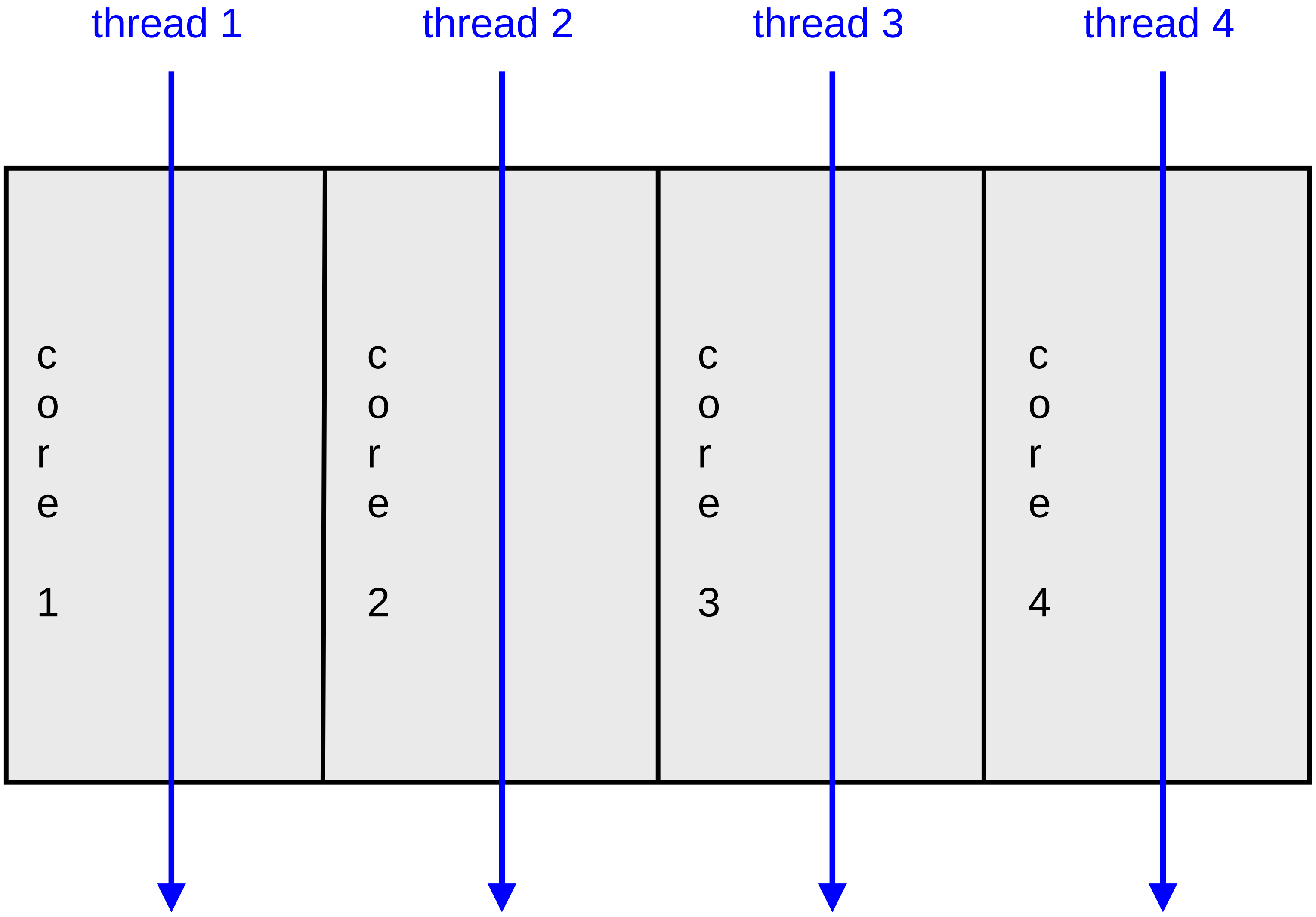
Multi-core CPU chip

Multi-core CPU chip

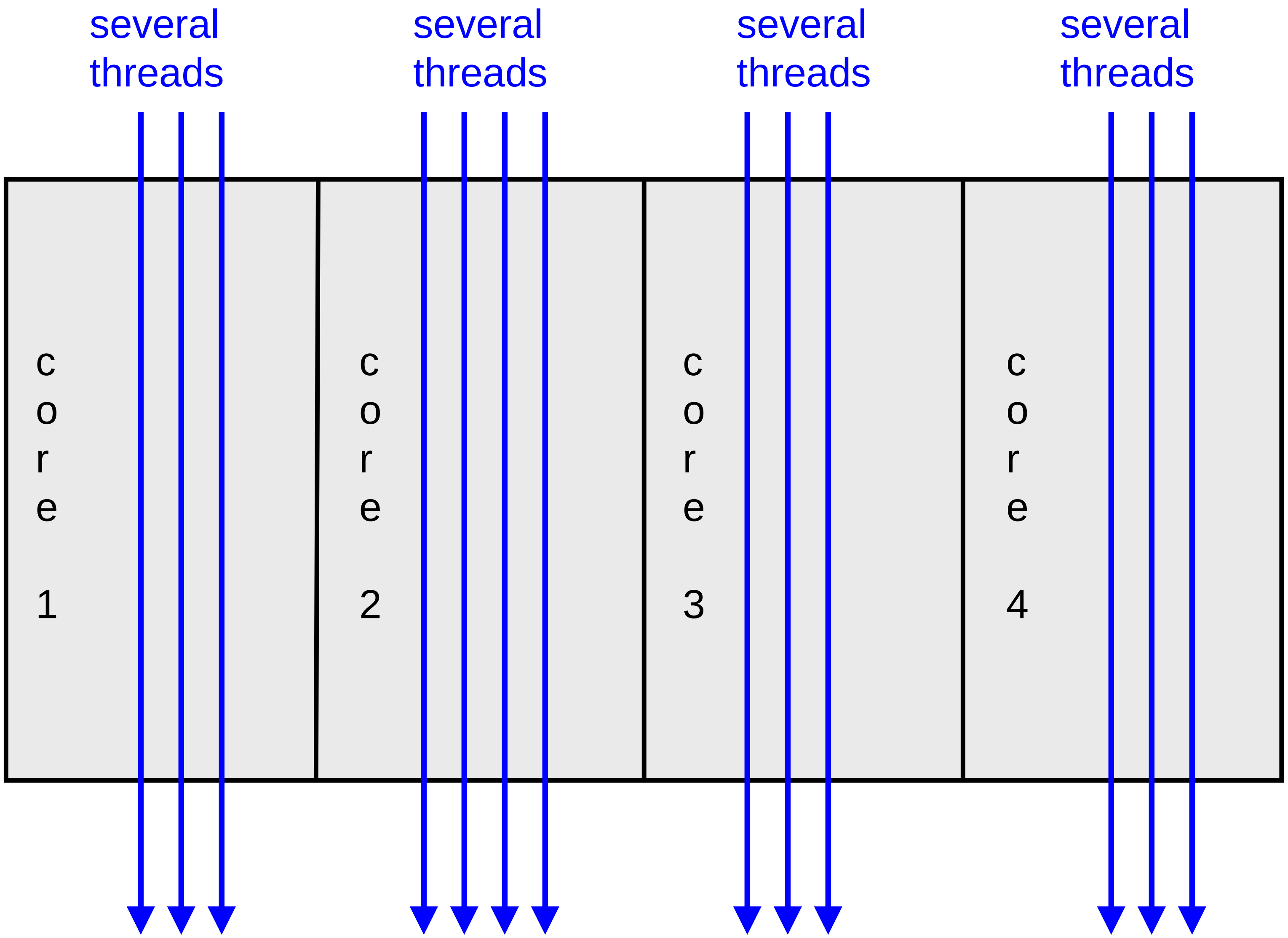
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



Cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures: explicitly exploiting TLP

Multiprocessor memory types

- Shared memory:
  In this model, there is one (large) common shared memory for all processors
- Distributed memory:
  In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

Multi-core processor is a special kind of a multiprocessor: All processors are on the same chip

- Multi-core processors are MIMD:
  Different cores execute different threads (Multiple Instructions), operating on different parts of memory (Multiple Data).

- Multi-core is a shared memory multiprocessor:
  All cores share the same memory

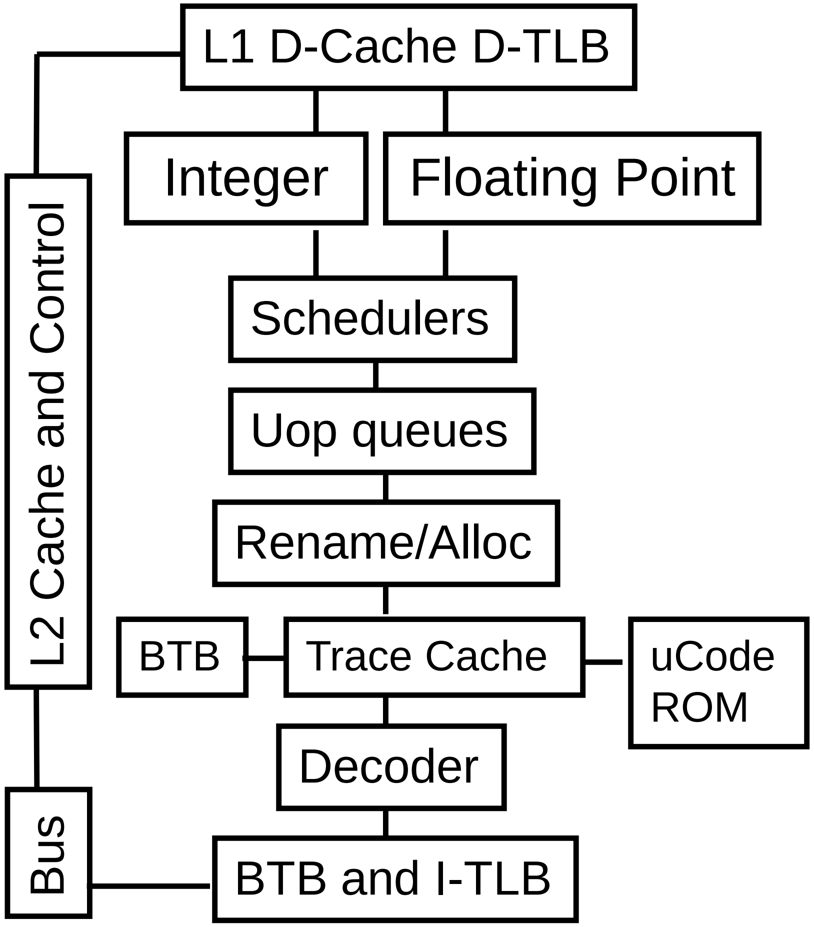
A technique complementary to multi-core: Simultaneous multithreading

Processor pipeline can get stalled: *Why?*

– Waiting for the result of a long floating point (or integer) operation

– Waiting for data to arrive from memory

– Other execution units wait;



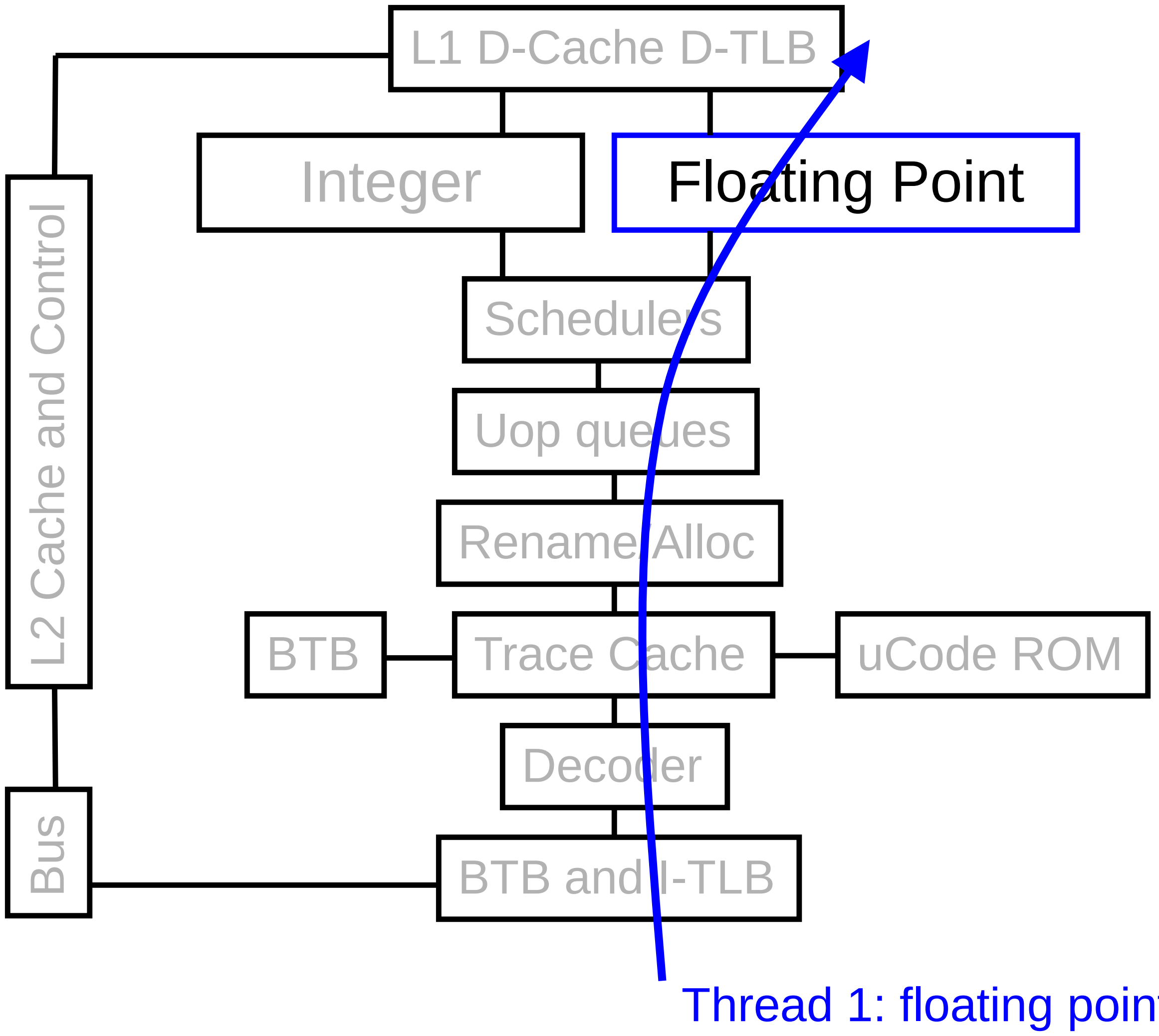
Source: Intel

Simultaneous multithreading (SMT)

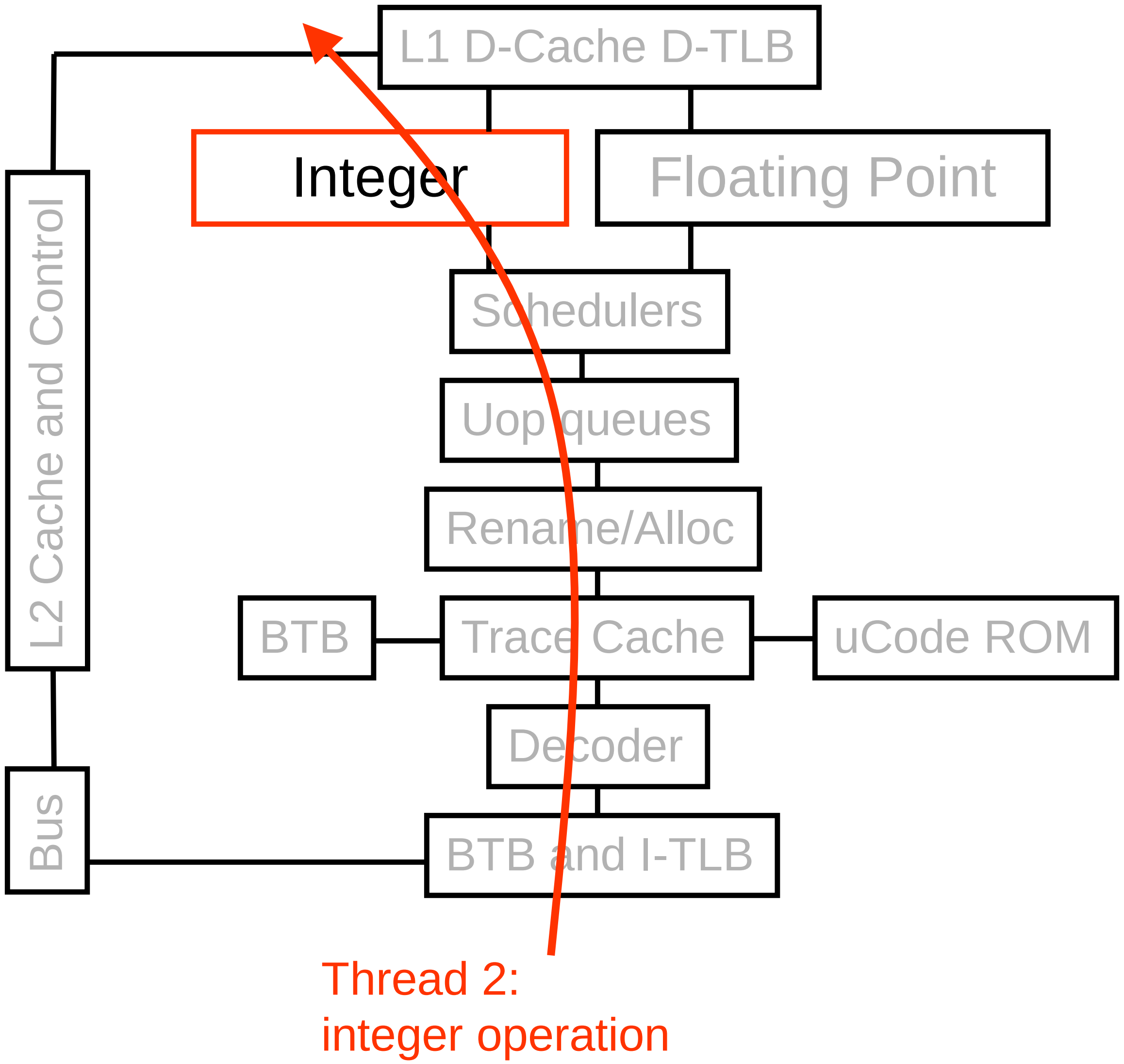
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple "threads" on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

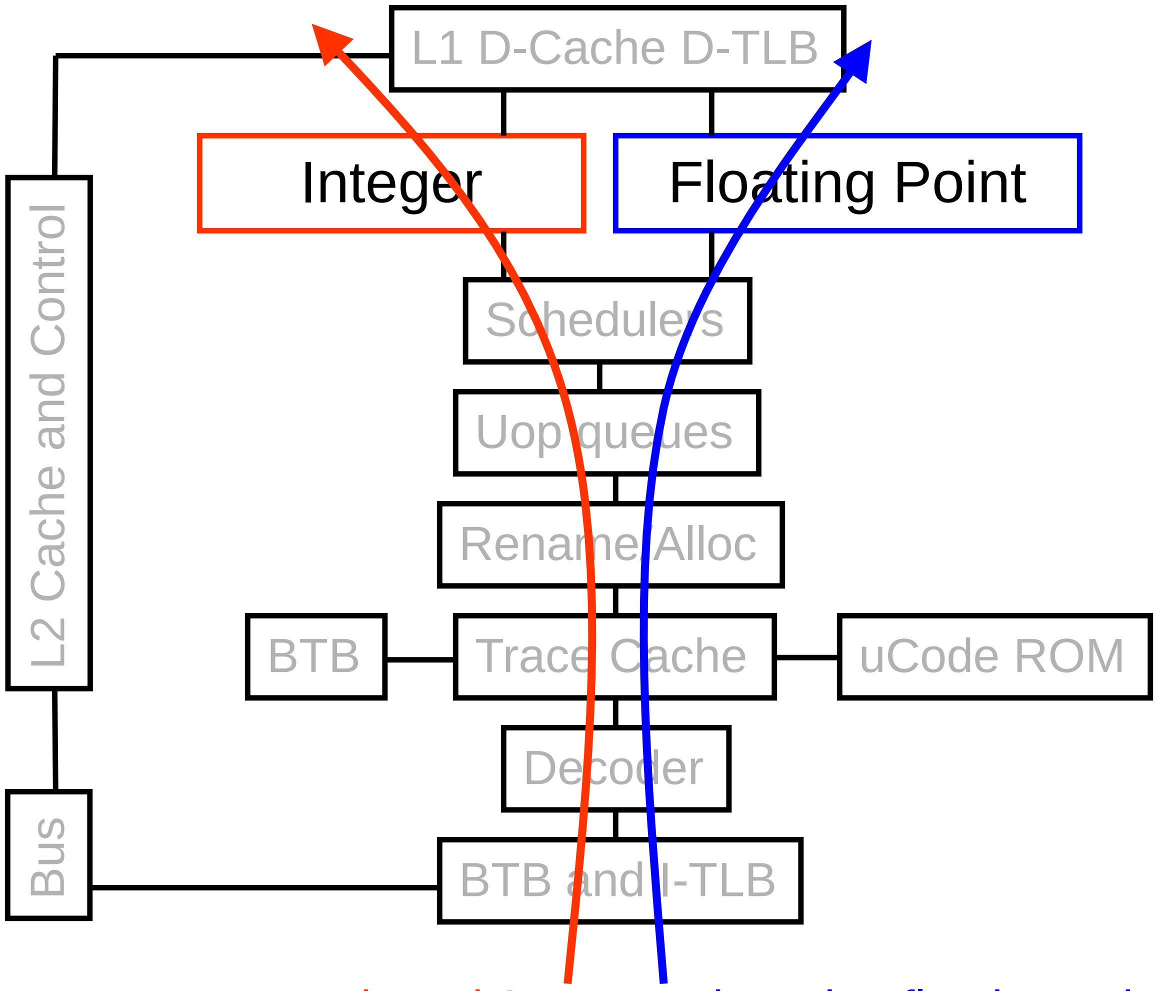
Without SMT, only a single thread can run at any given time



Without SMT, only a single thread can run at any given time



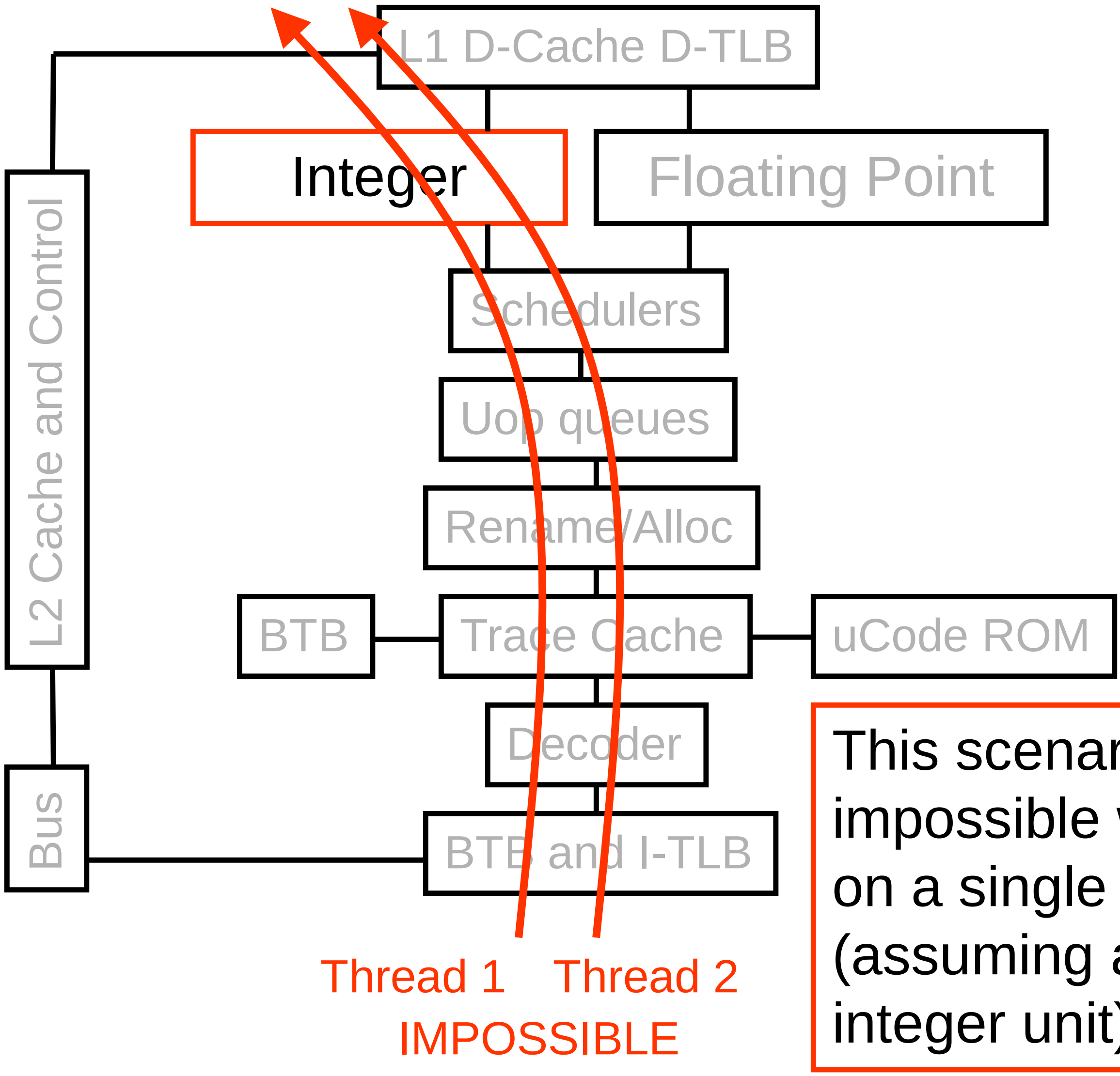
SMT processor: both threads can run concurrently



Thread 2: integer operation

Thread 1: floating point

But can't simultaneously use the same functional unit

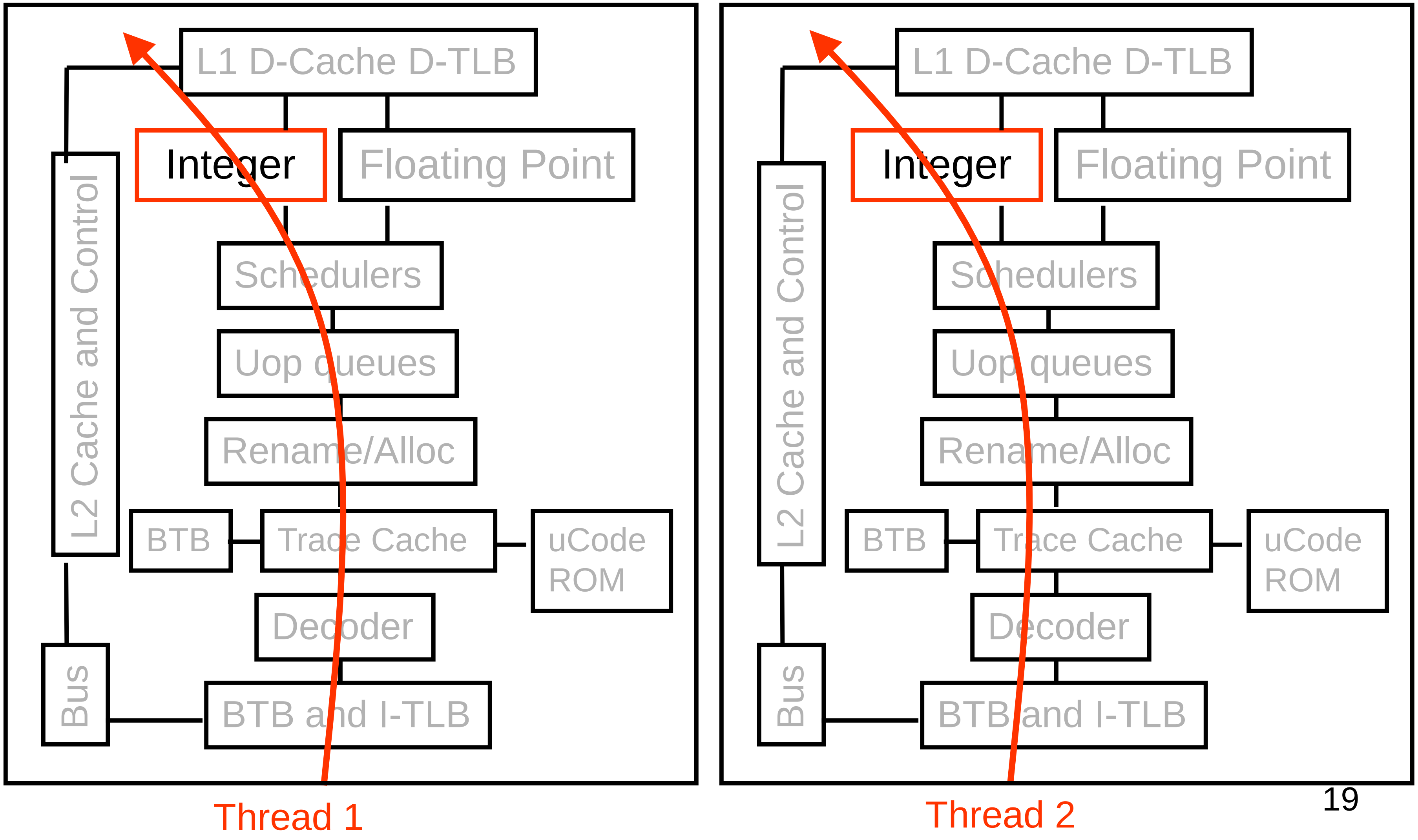


This scenario is impossible with SMT on a single core (assuming a single integer unit)

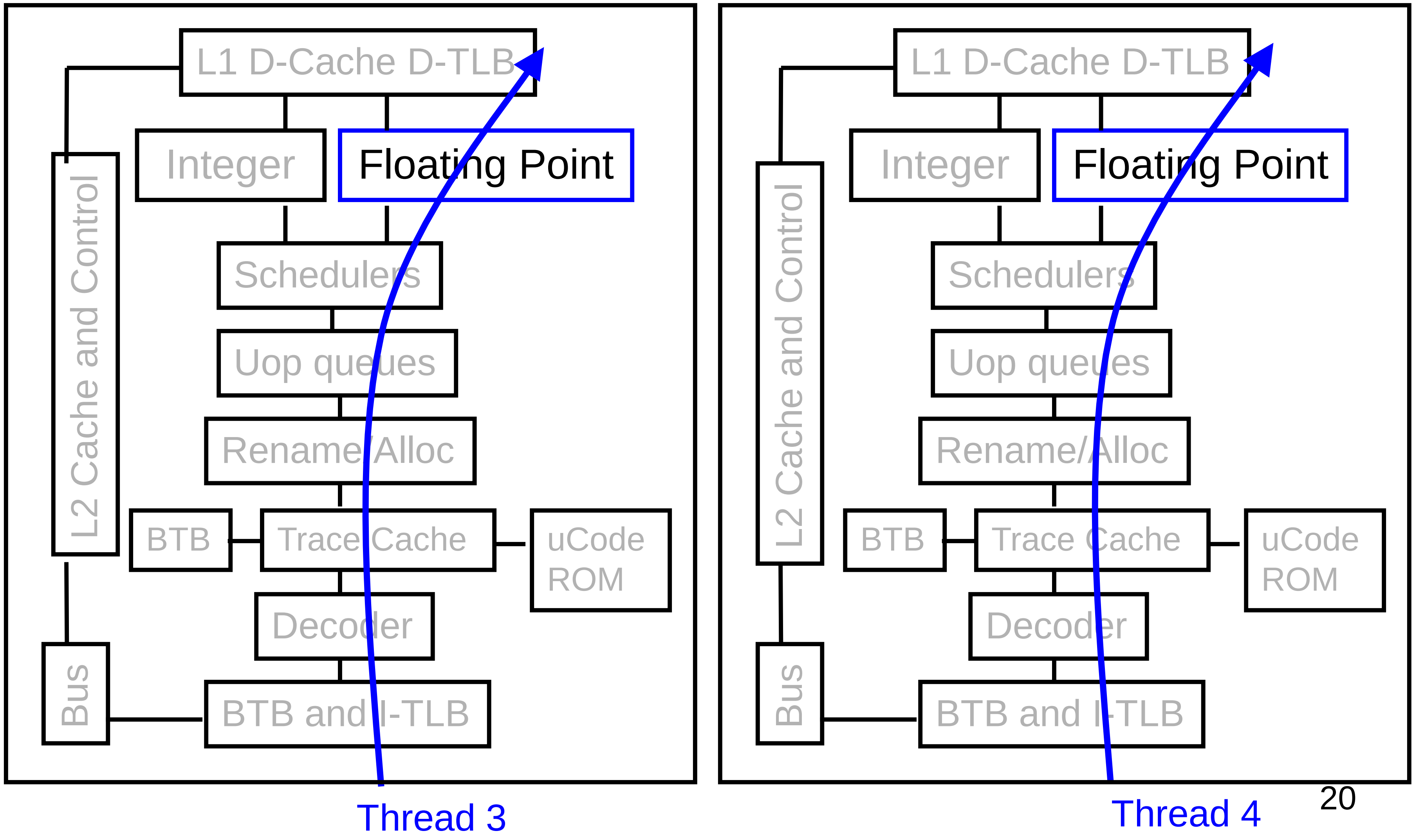
SMT not a “true” parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Compare to multi-core:
  each core has its own copy of resources

Multi-core: threads can run on separate cores



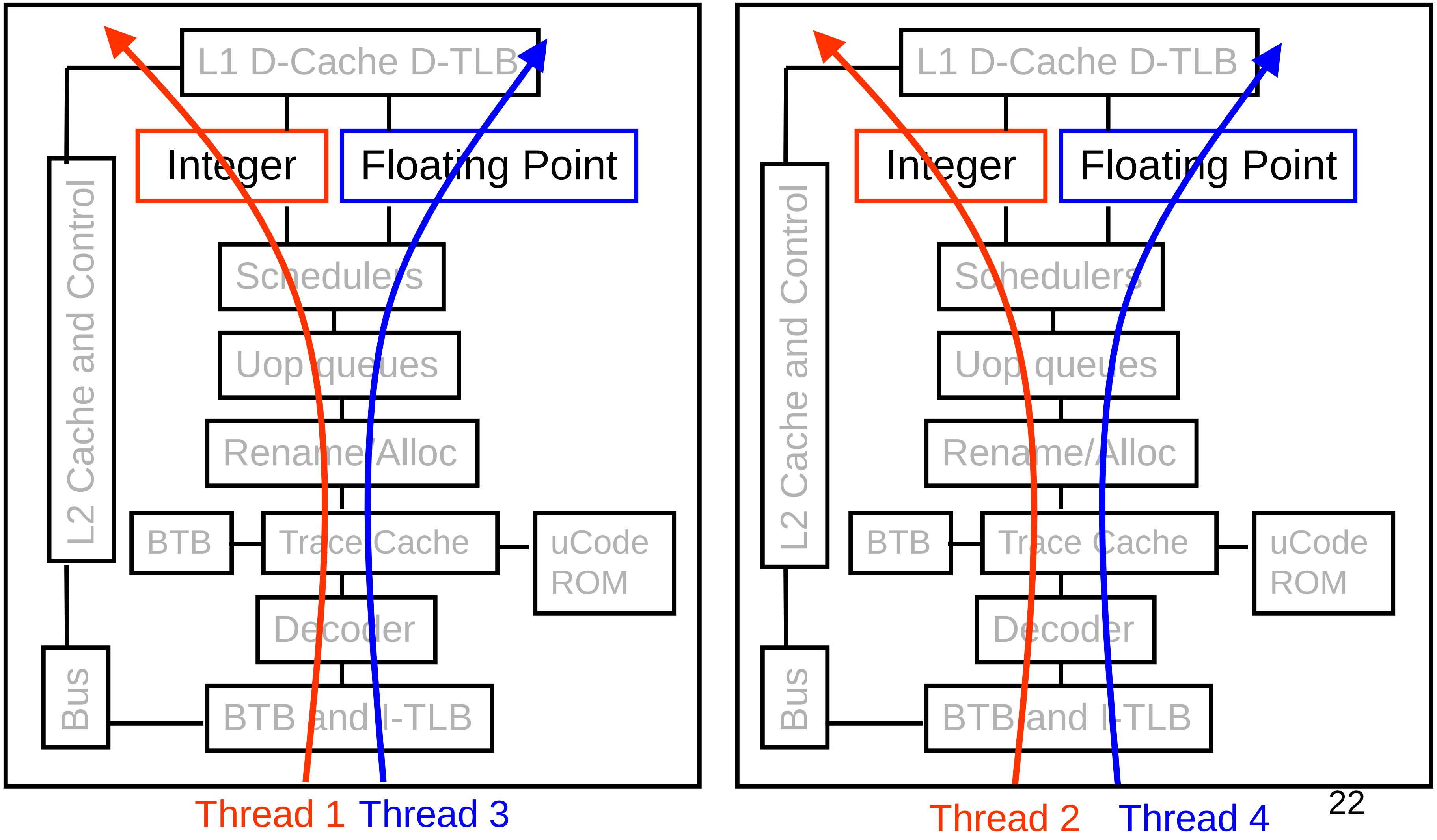
Multi-core: threads can run on separate cores



Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT
- The number of SMT threads:
  2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

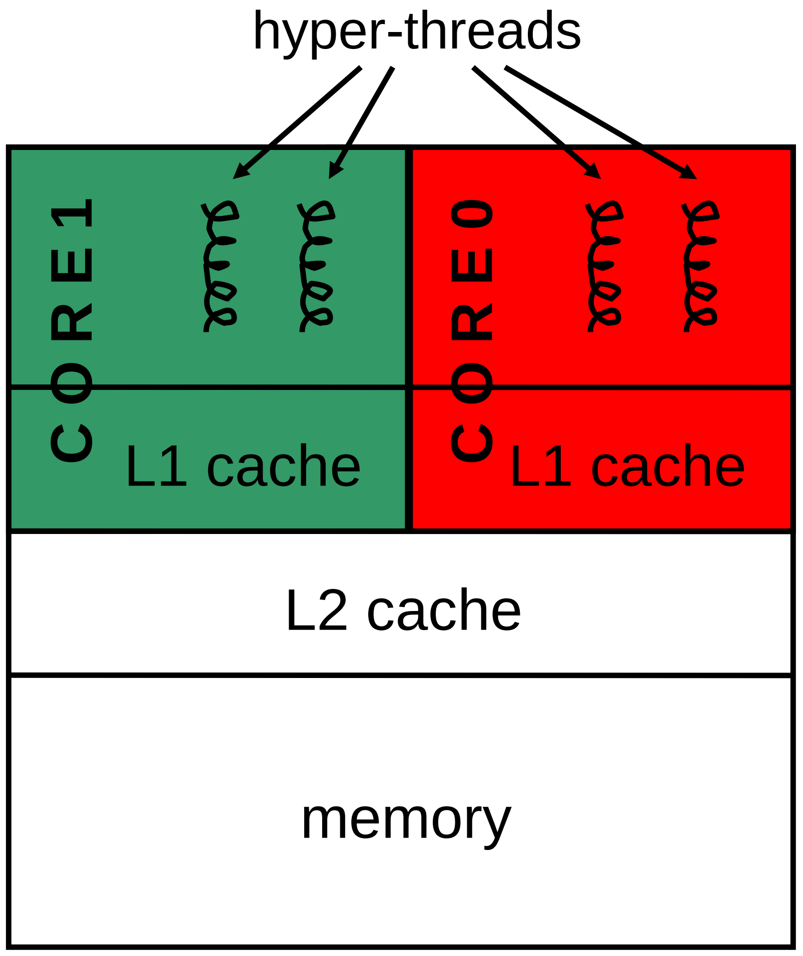
- Multi-core:
  - Since there are several cores,
    each is smaller and not as powerful
    (but also easier to design and manufacture)
  - However, powerful with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

The memory hierarchy

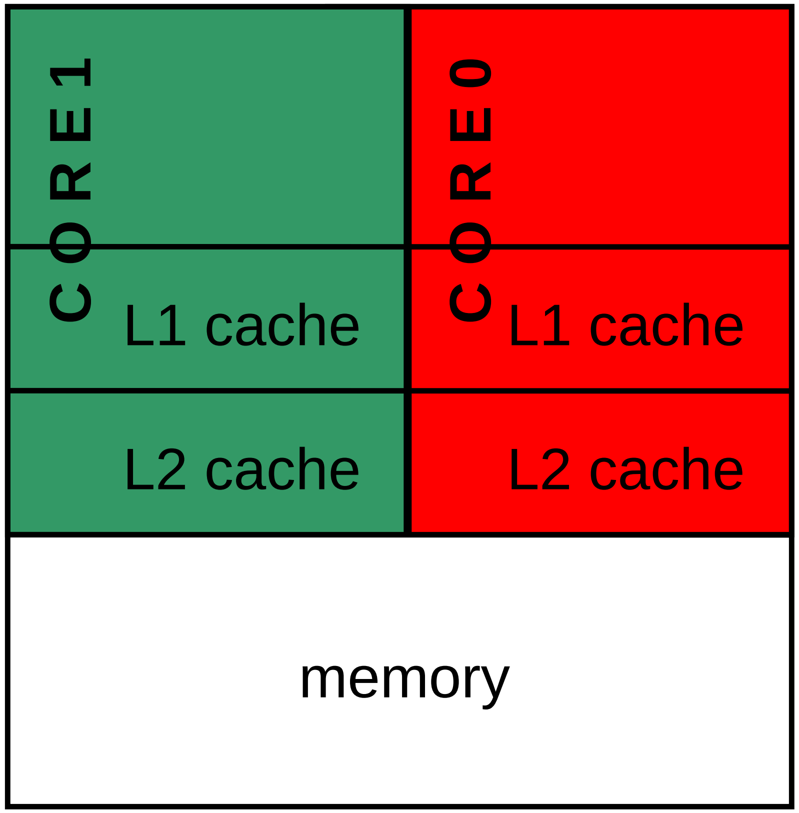
- If simultaneous multithreading only:
  - all caches are shared
- Multi-core chips:
  - L1 caches private
  - L2 caches private in some architectures and shared in others
- Memory is always shared

Typical Dual Core Features

- Dual-core
  Intel Xeon processors
- Each core is
  hyper-threaded
- Private L1 caches
- Shared L2 caches

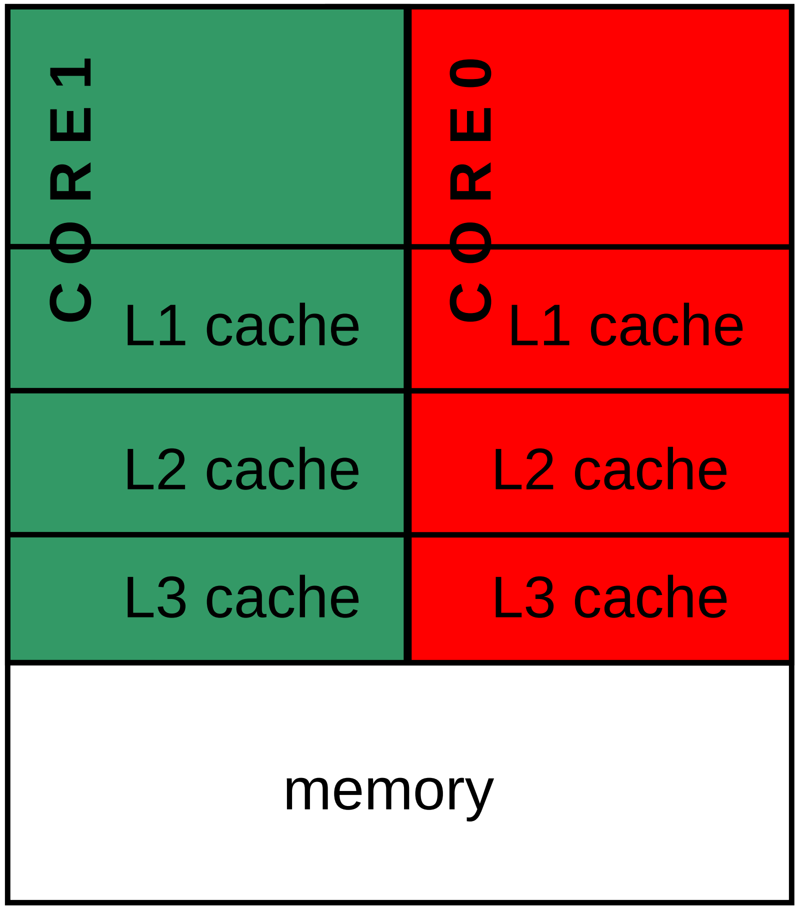


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron,
AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

Private vs shared caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

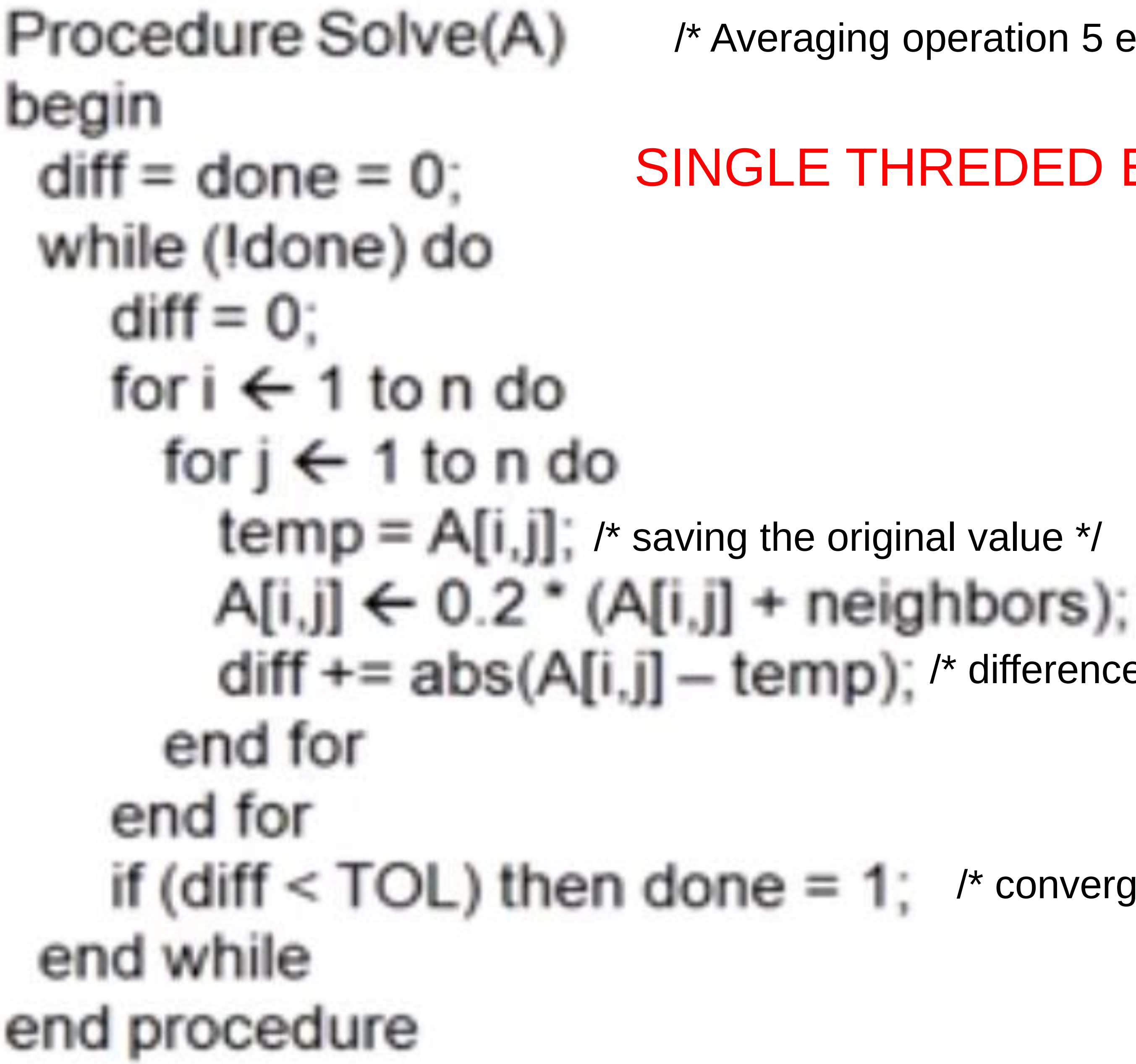
Example: Multi-threaded way of solving a computational problem

Typical application is heat wave propagation computation:

-- Averaging and testing for convergence

(a) Single Threaded Solution &
(b) Multi-threaded approach using Shared Memory Multiprocessor System
(c) Message Passing approach (suitable for HPC)

SINGLE THREDED EXECUTION



```
Procedure Solve(A)          /* Averaging operation 5 elements */
begin
  diff = done = 0;
  while (!done) do
      diff = 0;
      for i ← 1 to n do
         for j ← 1 to n do
            temp = A[i,j]; /* saving the original value */
            A[i,j] ← 0.2 * (A[i,j] + neighbors);
            diff += abs(A[i,j] – temp); /* difference computation */
         end for
      end for
      if (diff < TOL) then done = 1;   /* convergence testing */
  end while
end procedure
```

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

MULTI-THREDED EXECUTION - SM

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
       mydiff = diff = 0;
       BARRIER(bar1,nprocs);
       for i ← mymin to mymax
          for j ← 1 to n do
             ...
          endfor
       endfor
       LOCK(diff_lock);
       diff += mydiff;
       UNLOCK(diff_lock);
       BARRIER (bar1, nprocs);
       if (diff < TOL) then done = 1;
       BARRIER (bar1, nprocs);
    endwhile
```

```
main()
   read(n); read(nprocs);
   CREATE (nprocs-1, Solve);
   Solve();
   WAIT_FOR_END (nprocs-1);

procedure Solve()
   int i, j, pid, nn = n/nprocs, done=0;
   float temp, tempdiff, mydiff = 0;
   myA ← malloc(...)
   initialize(myA);
   while (!done) do
       mydiff = 0;
       if (pid != 0)
         SEND(&myA[1,0], n, pid-1, ROW);
       if (pid != nprocs-1)
         SEND(&myA[nn,0], n, pid+1, ROW);
       if (pid != 0)
         RECEIVE(&myA[0,0], n, pid-1, ROW);
       if (pid != nprocs-1)
         RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
       for i ← 1 to nn do
          for j ← 1 to n do
             ...
          endfor
       endfor
       if (pid != 0)
         SEND(mydiff, 1, 0, DIFF);
         RECEIVE(done, 1, 0, DONE);
       else
         for i ← 1 to nprocs-1 do
            RECEIVE(tempdiff, 1, *, DIFF);
            mydiff += tempdiff;
         endfor
         if  (mydiff < TOL)  done = 1;
         for i ← 1 to nprocs-1  do
            SEND(done, 1, I, DONE);
         endfor
       endif
   endwhile
```

MESSAGE PASSING APPROACH

Sizes of Cache for I & D? Empirical study is possible….*Stay tuned!*

Let us now look into three types of Multithreaded Processors.

Acknowledgements: Thanks to Dr. Jernej Barbic, 2007