Multi-Core Architectures

15-213/18-243: Introduction to Computer Systems

27th (and last) Lecture, 29 April 2010

Instructors:

Bill Nace and Gregory Kesden

Today

- Thread safety
- Multi-core
- Parallelism on multi-core

Crucial concept: Thread Safety

- **Functions called from a thread (without external synchronization) must be *thread-safe***
  - Meaning: it must always produce correct results when called repeatedly from multiple concurrent threads

- **Some examples of thread-unsafe activities:**
  - Failing to protect shared variables
  - Relying on persistent state across invocations
  - Returning a pointer to a static variable
  - Calling a thread-unsafe functions

Thread-Unsafe Functions (Class 1)

- **Failing to protect shared variables**
  - Fix: Use P and V semaphore operations
  - Example: **`goodcnt.c`**
  - Issue: Synchronization operations will slow down code
    - e.g., **`badcnt`** requires 0.5s, **`goodcnt`** requires 7.9s

Thread-Unsafe Functions (Class 2)

- **Relying on persistent state across multiple function invocations**
  - Example: Random number generator (RNG) that relies on static state

```
/* rand: return pseudo-random integer on 0..32767 */
static unsigned int next = 1;
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Making Thread-Safe RNG

- **Pass state as part of argument**
  - and, thereby, eliminate static state

```
/* rand - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
  *nextp = *nextp*1103515245 + 12345;
  return (unsigned int)(*nextp/65536) % 32768;
}
```

- **Consequence: programmer using `rand_r` must maintain seed**

Thread-Unsafe Functions (Class 3)

- **Returning a ptr to a `static` variable**

```
struct hostent
*gethostbyname(char name)
{
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

- **Fixes:**
  - 1. Rewrite code so caller passes pointer to **`struct`**
    - Issue: Requires changes in caller and callee

```
hostp = Malloc(...);
gethostbyname_r(name, hostp);
```

  - 2. ***Lock-and-copy***
    - Issue: Requires only simple changes in caller (and none in callee)
    - However, caller must free memory

```
struct hostent
*gethostbyname_ts(char *name)
{
  struct hostent *q = Malloc(...);
  struct hostent *p;
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;   /* copy */
  V(&mutex);
  return q;
}
```

Thread-Unsafe Functions (Class 4)

- **Calling thread-unsafe functions**
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions ☺

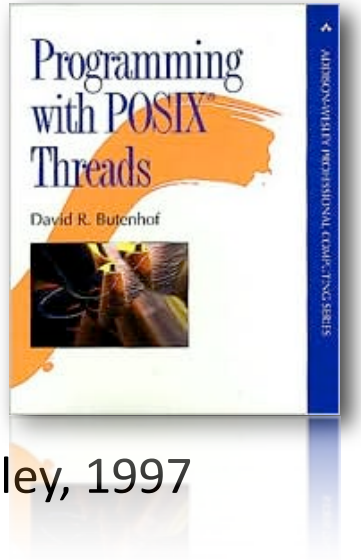
Thread-Safe Library Functions

- **All functions in the Standard C Library (at the back of your K&R text) are thread-safe**
  - Examples: `malloc`, `free`, `printf`, `scanf`
- **Most Unix system calls are thread-safe, with a few exceptions:**

Thread-unsafe function	Class	Reentrant version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	`(none)`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

Threads Summary

- **Threads provide another mechanism for writing concurrent programs**
- **Threads are growing in popularity**
  - Somewhat cheaper than processes
  - Easy to share data between threads
- **However, the ease of sharing has a cost:**
  - Easy to introduce subtle synchronization errors
    - Which are very, very, very, very, very difficult to discover
  - Tread carefully with threads!



- **For more info:**
  - D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997

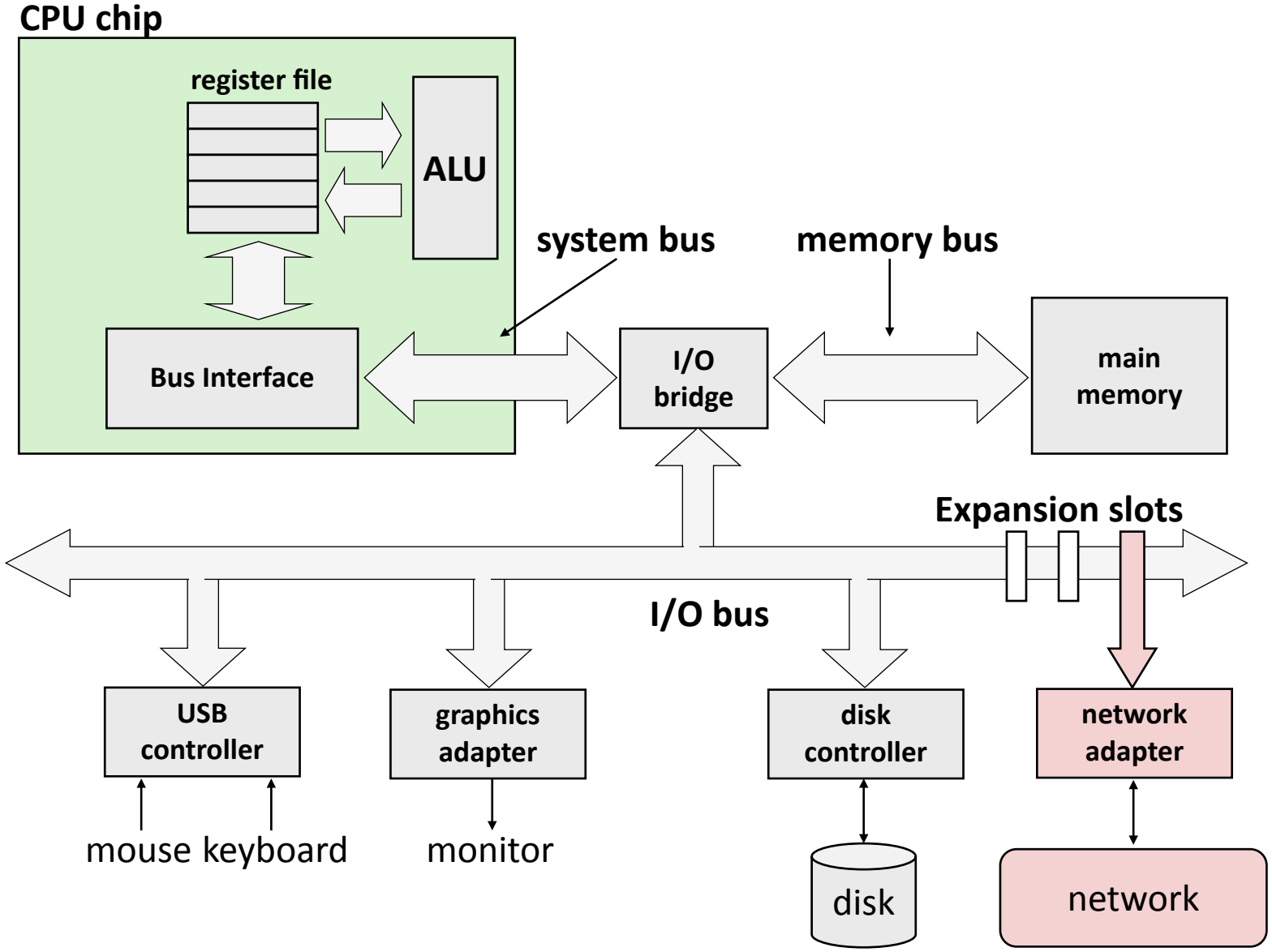
Today

- Thread safety
- **Multi-core**
- Parallelism on multi-core

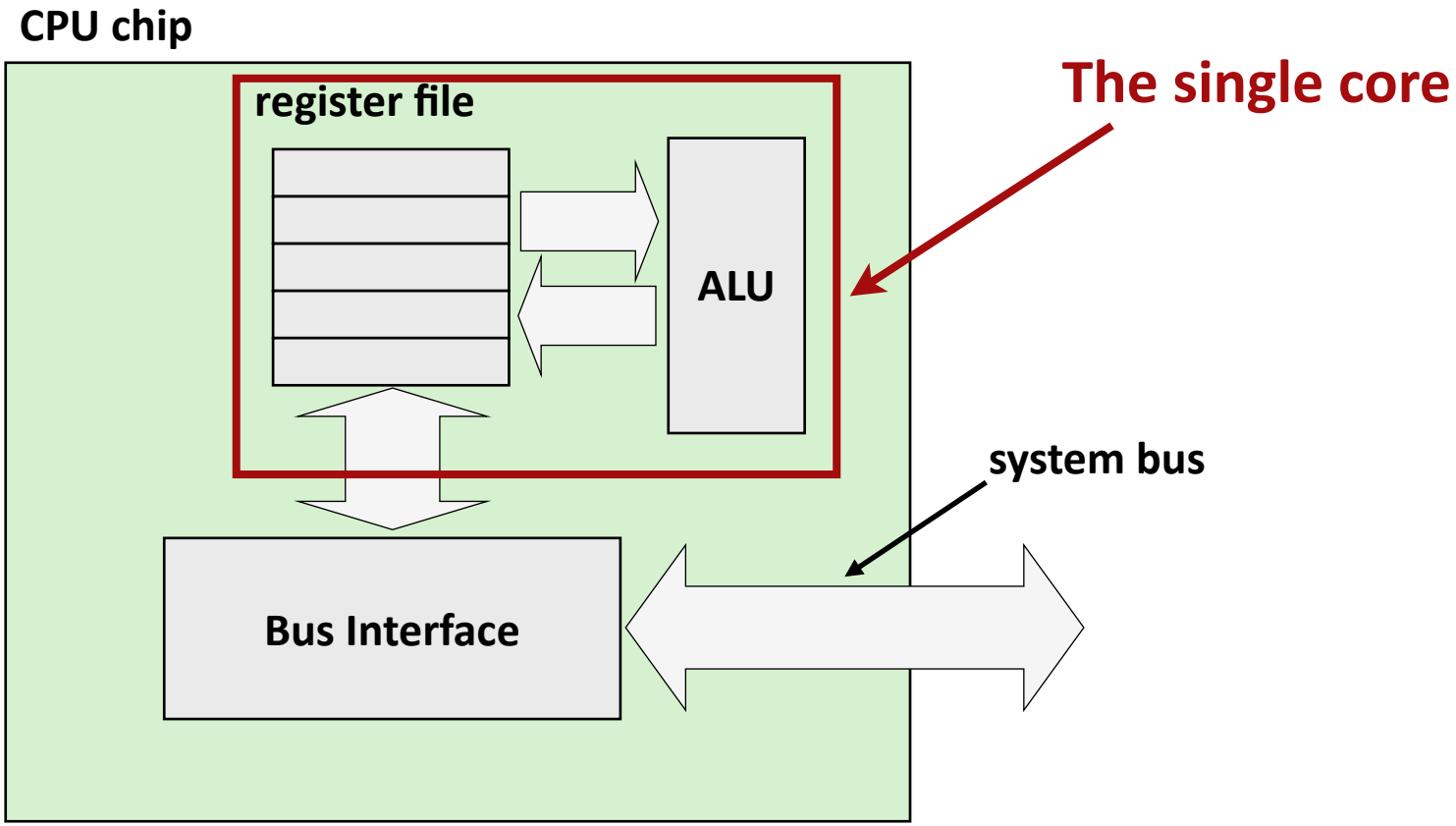
Why Multi-Core?

- **Traditionally, single core performance is improved by increasing the clock frequency...**
- **...and making deeply pipelined circuits...**
- **Which leads to...**
  - Heat problems
  - Speed of light problems
  - Difficult design and verification
  - Large design teams
  - Big fans, heat sinks
  - Expensive air-conditioning on server farms
- **Increasing clock frequency no longer the way to go forward**

Single Core Computer

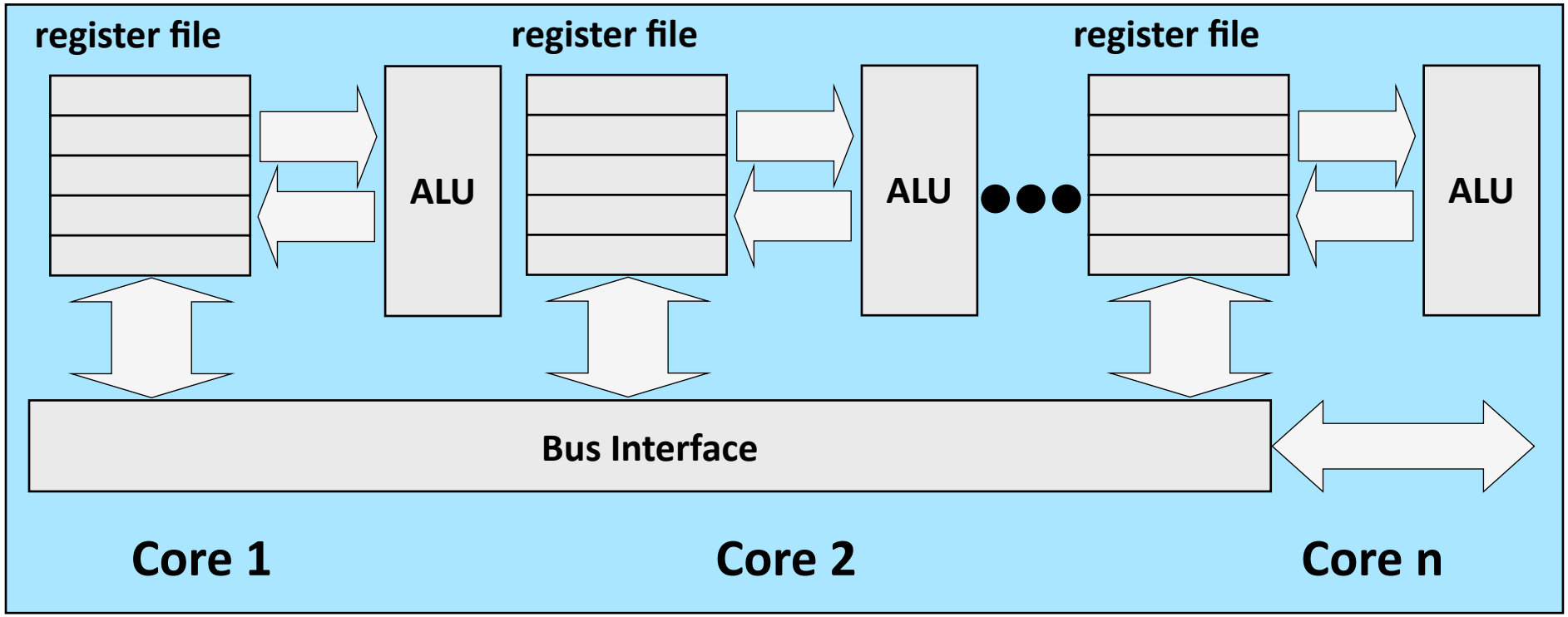


Single Core CPU Chip



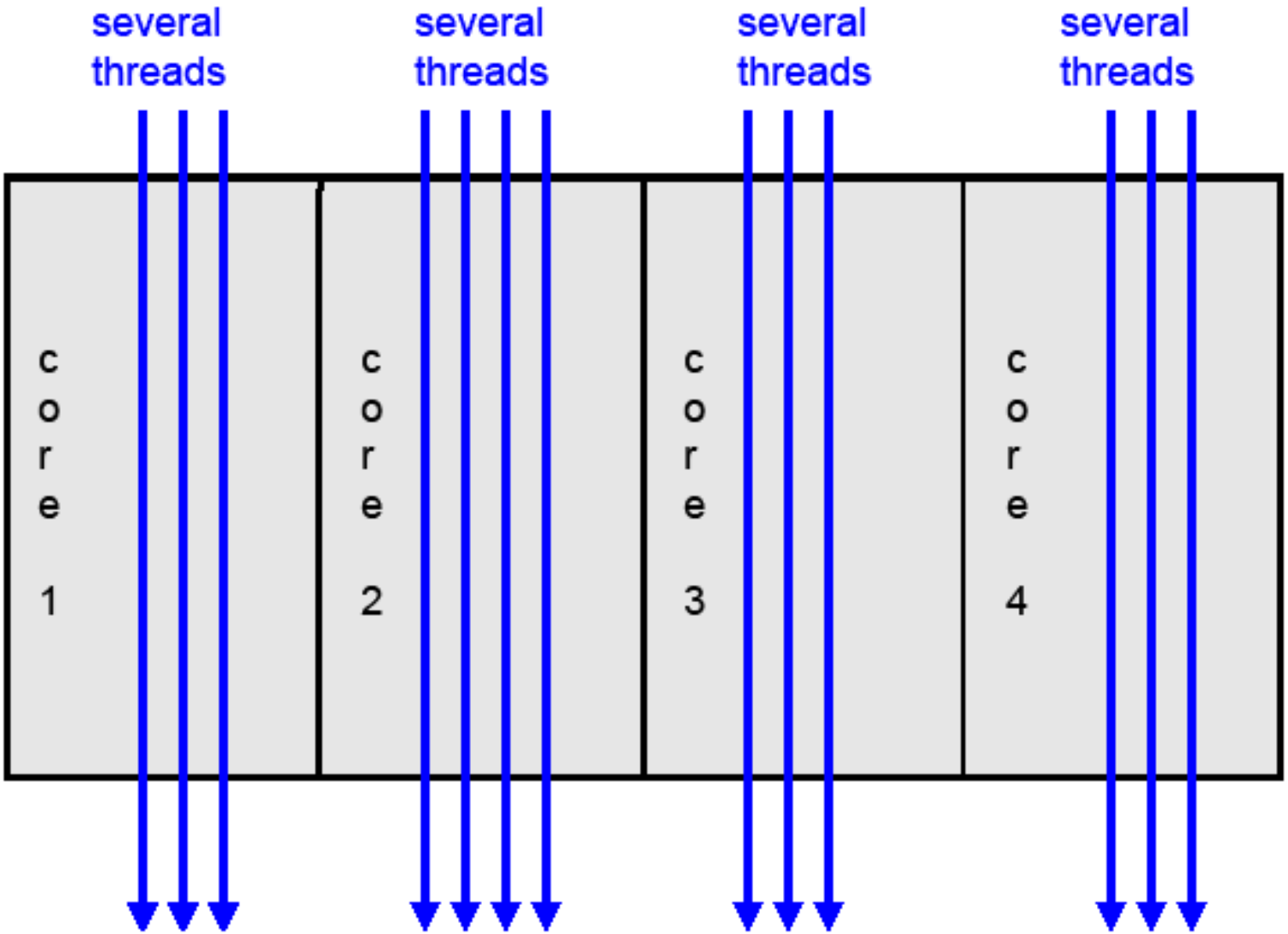
Multi-Core Architecture

- Somewhat recent trend in computer architecture
- Replicate many cores on a single die



Multi-core Chip

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:**
  - Mac OS X, Linux, Windows, …

Today

- Thread safety
- Multi-core
- **Parallelism on multi-core**

Instruction-Level Parallelism

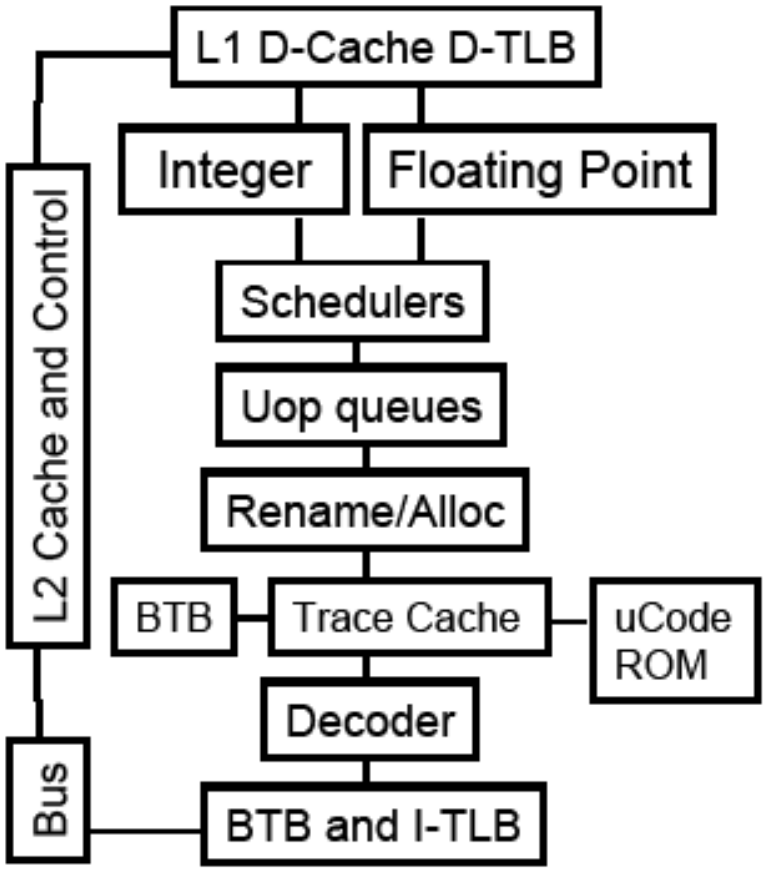
- **Parallelism at the machine-instruction level**
- **Achieved in the processor with**
  - Pipeline
  - Re-ordered instructions
  - Split into micro-instructions
  - Aggressive branch prediction
  - Speculative execution
- **ILP enabled rapid increases in processor performance**
  - Has since plateaued

Thread-level Parallelism

- **Parallelism on a coarser scale**
- **Server can serve each client in a separate thread**
  - Web server, database server
- **Computer game can do AI, graphics, physics, UI in four different threads**
- **Single-core superscalar processors cannot fully exploit TLP**
  - Thread instructions are interleaved on a coarse level with other threads
- **Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP**

Simultaneous Multithreading (SMT)

- **Complimentary technique to multi-core**
- **Addresses the stalled pipeline problem**
  - Pipeline is stalled waiting for the result of a long operation (float?)
  - ... or waiting for data to arrive from memory (long latency)
- **Other execution units are idle**

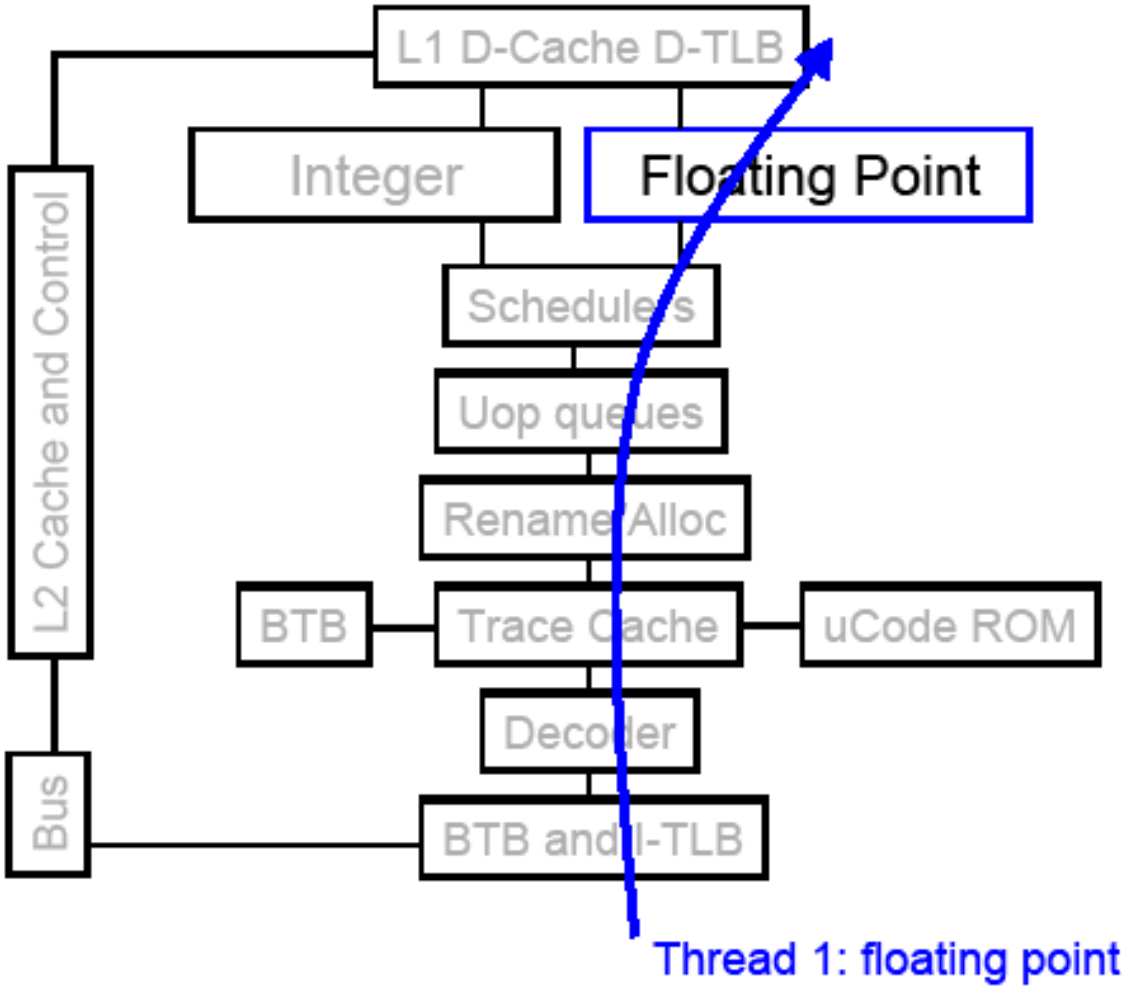


Source: Intel

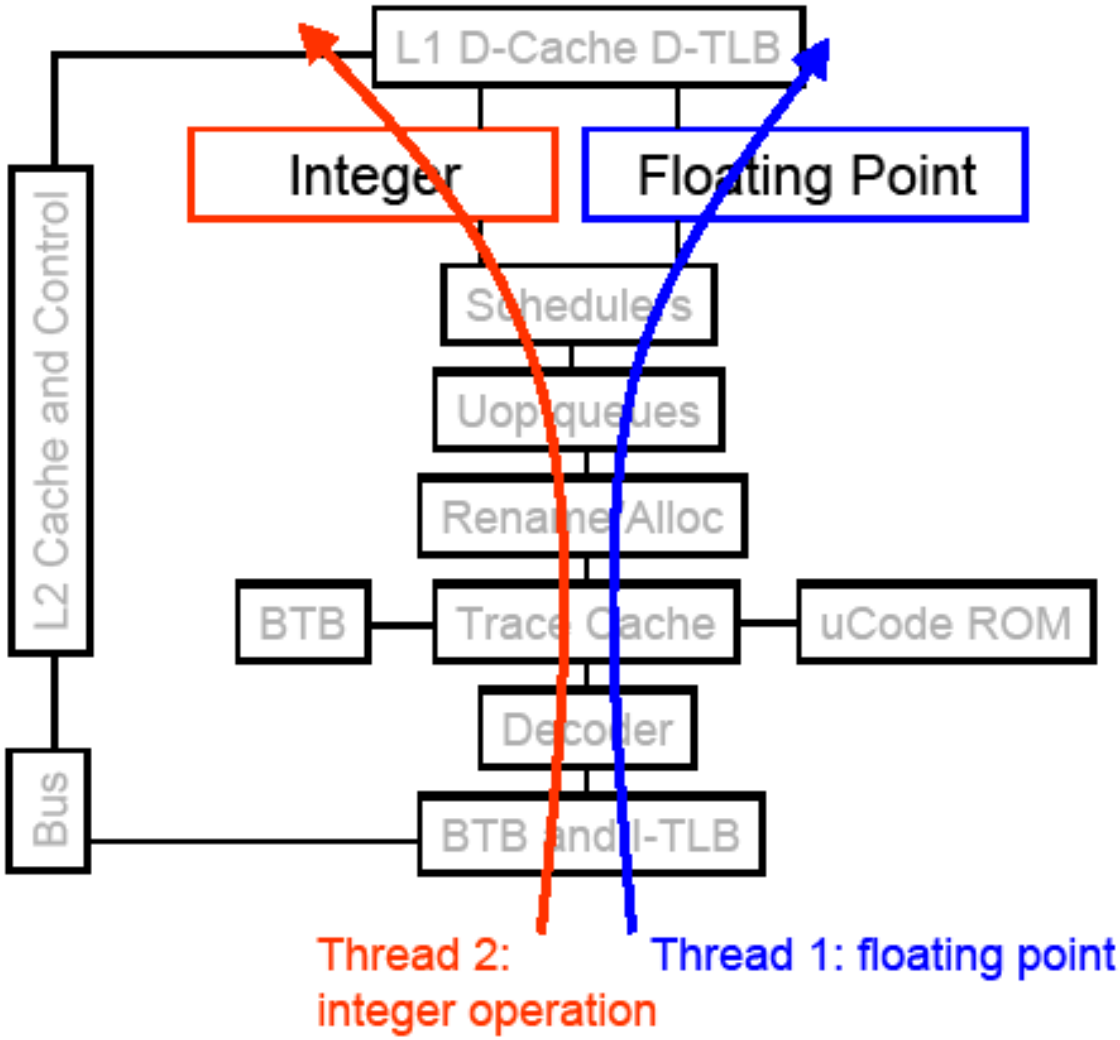
SMT

- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads”
- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

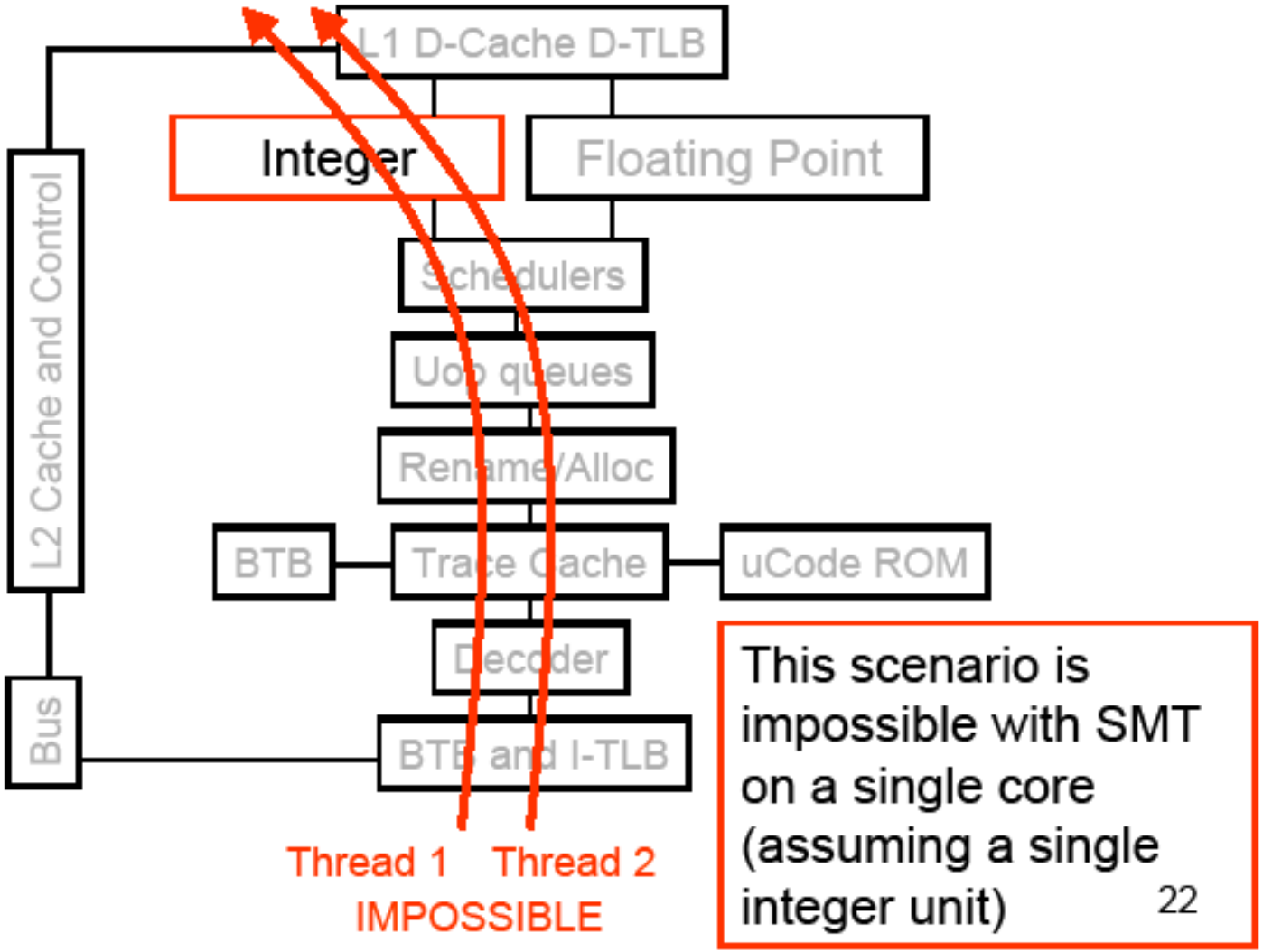
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



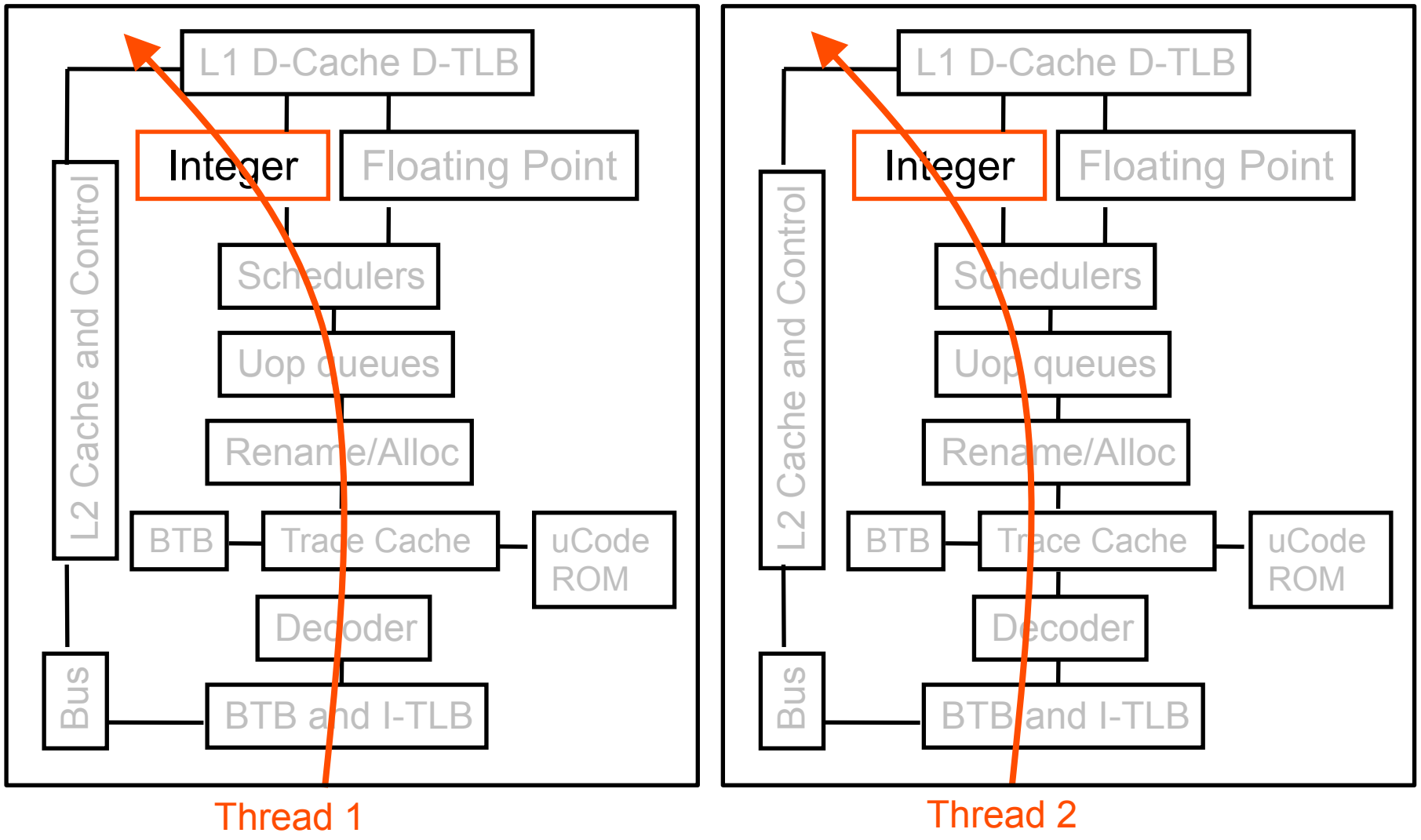
But: Can't simultaneously use the same functional unit



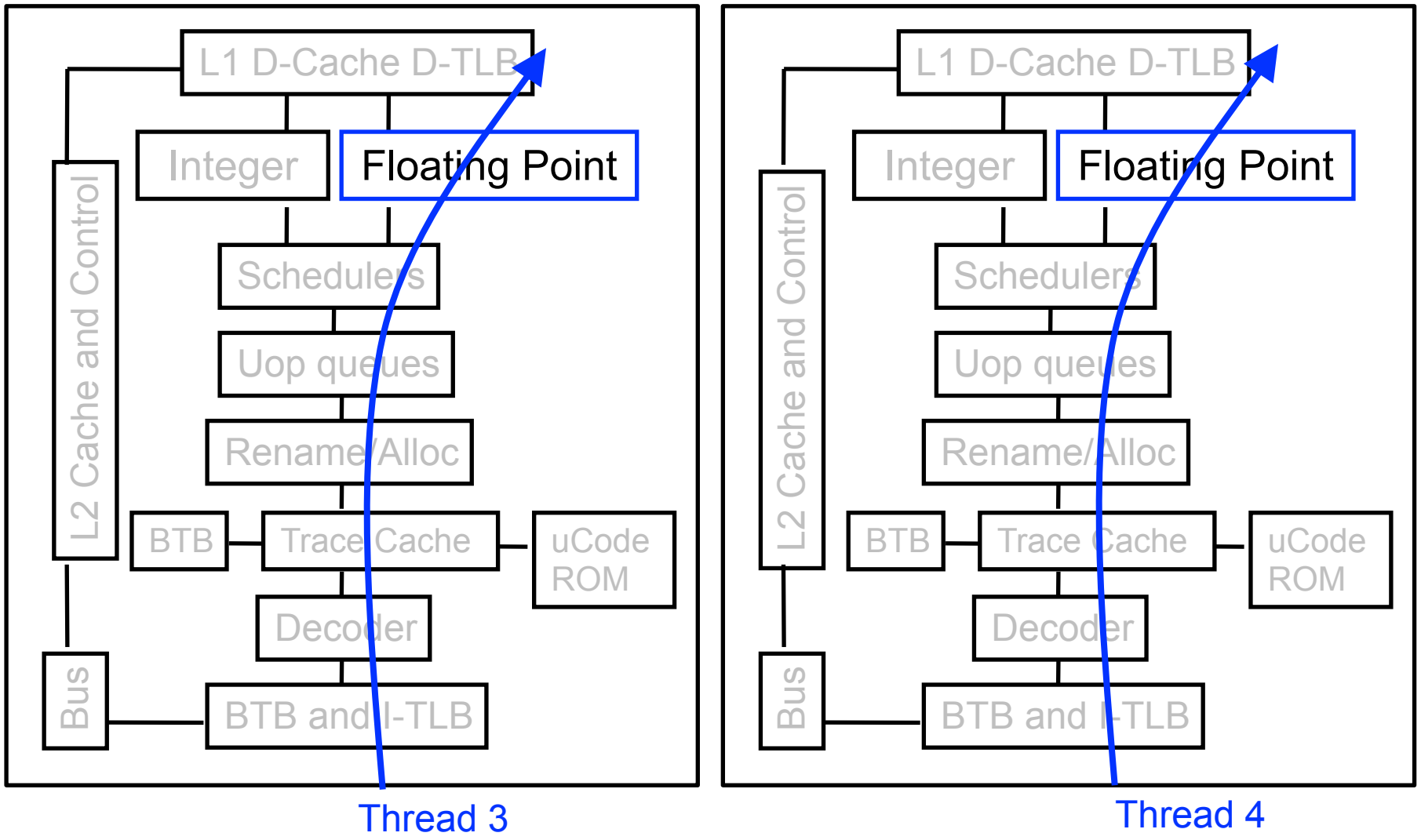
SMT is not a "true" parallel processor

- **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 run on separate cores



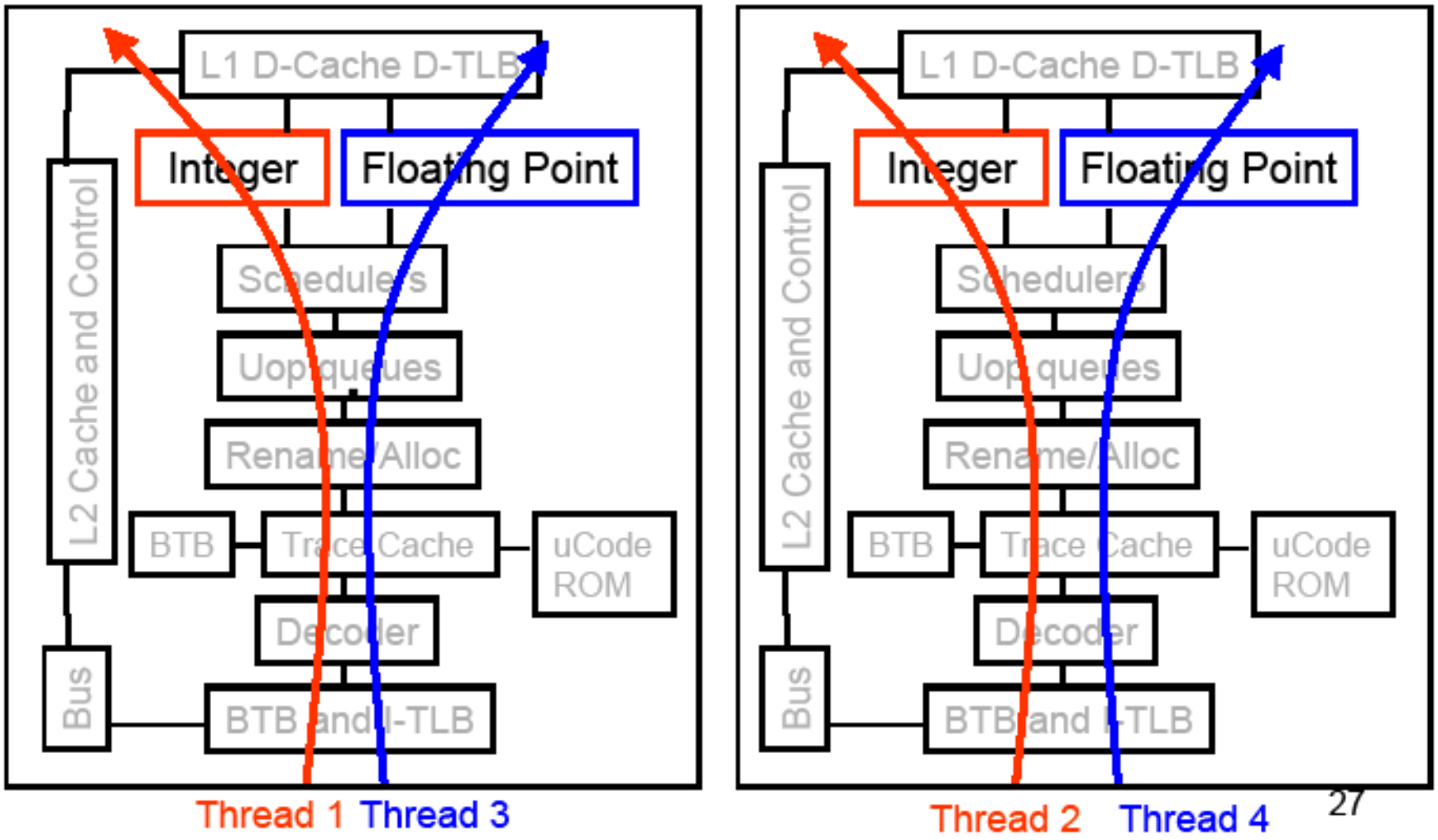
Multi-core: Threads run on separate cores



Combining Multi-core and SMT

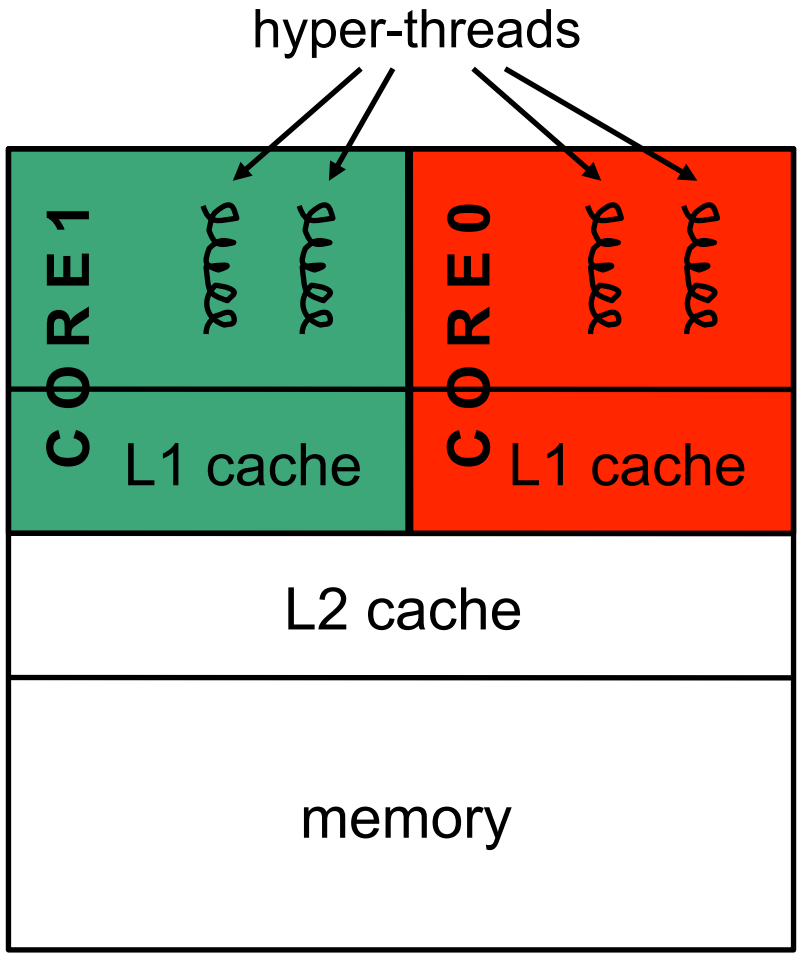
- **Cores can be SMT-enabled (or not)**
- **The different combinations:**
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT: our fish machines
- **The number of SMT threads is determined by hardware design**
  - 2, 4 or sometimes 8 simultaneous threads
- **Intel calls them "Hyper-threads"**

SMT Dual-core: all four threads can run concurrently

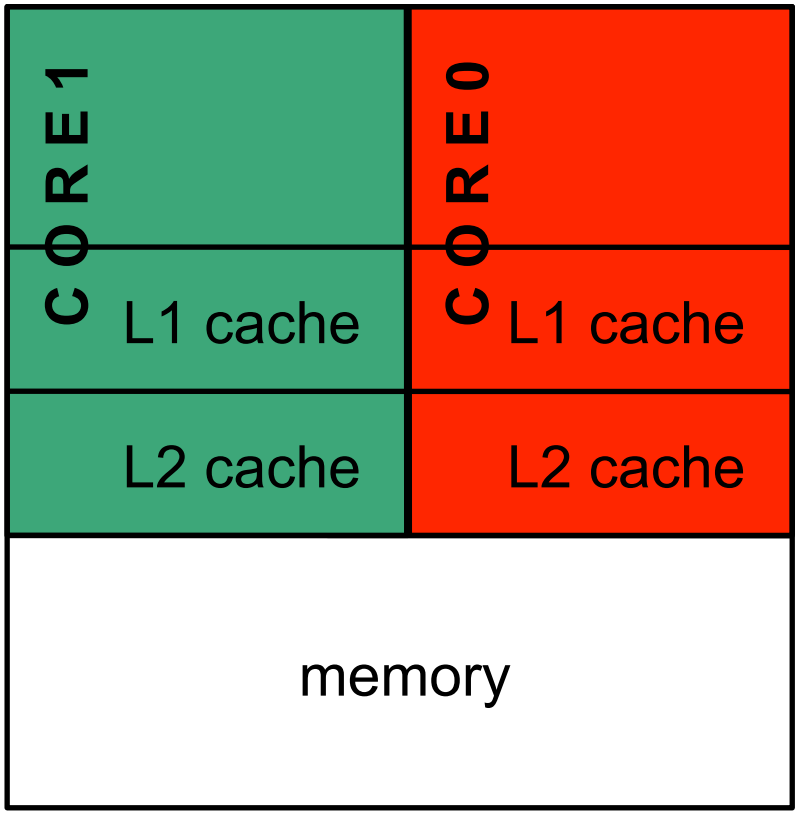


SMT/Multi-Core and the Memory Hierarchy

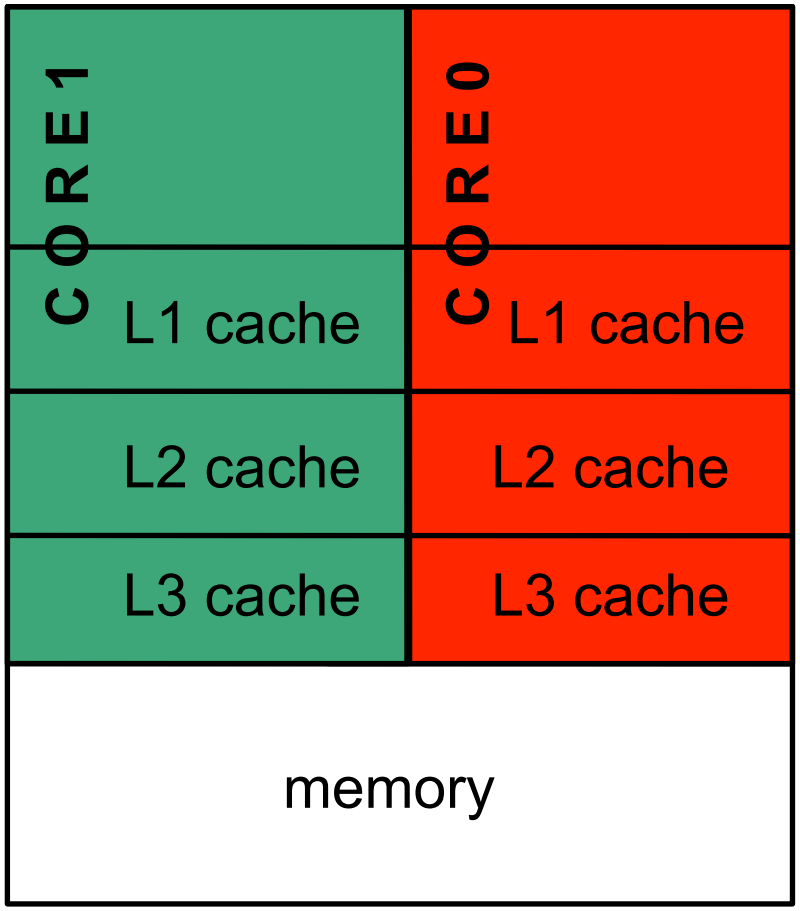
- **SMT is a sharing of pipeline resources**
  - Thus all caches are shared
- **Multi-core chips:**
  - L1 caches are private (i.e. each core has its own L1)
  - L2 cache private in some architectures, shared in others
  - Main memory is always shared
- **Example: Fish machines**
  - Dual-core Intel Xeon processors
  - Each core is hyper-threaded
  - Private L1, shared L2 caches



Designs with Private L2 Caches



Examples: AMD Opteron, AMD Athlon, Intel Pentium D



Example: Intel Itanium 2

Quad Core 2 Duo shares L2 in pairs of cores

Private vs Shared Cache

- **Advantages of Private Cache**
  - Closer to the core, so faster access
  - No contention for core access -- no waiting while another core accesses
- **Advantages of Shared Cache**
  - Threads on different cores can share same cache data
  - More cache space is available if a single (or a few) high-performance threads run
- **Cache Coherence Problem**
  - The same memory value can be stored in multiple private caches
  - Need to keep the data consistent across the caches
  - Many solutions exist
    - Invalidation protocol with bus snooping, ...

Summary

- **Multi-Core Architectures**
- **Simultaneous Multithreading**

- **Next Time:**
  - There is no next time ☹