arm

Multithreading

Module 9

© 2021 Arm Limited

Module Syllabus

- Multitasking
- Fine-grained multithreading
  - Thread selection
  - Case study
- Coarse-grained multithreading
  - Case study
- Simultaneous multithreading (SMT)
  - SMT policies
  - Case study

© 2021 Arm Limited

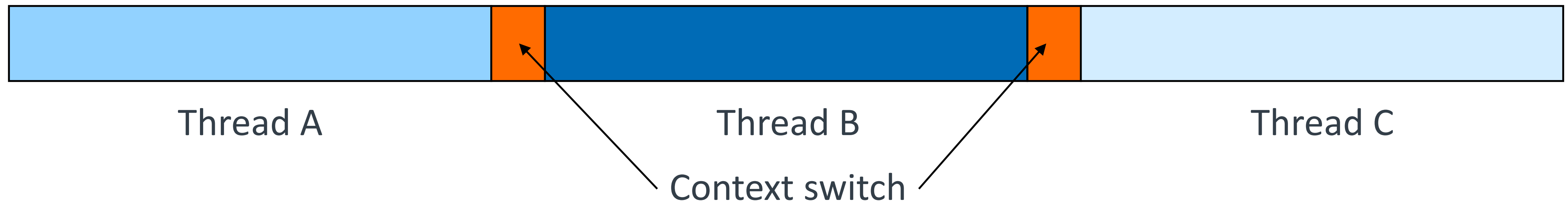
Motivation

- Multicore processors are the most generic way of extracting coarse-grained parallelism.
  - Allow exploitation of thread-level parallelism (within a program) and process-level parallelism (across applications).
- They are also the most simple extension of a uniprocessor.
  - Simply replicate the core several times on the chip.
  - Add the logic for implementing cache coherence, etc.
- However, they come at significant costs.
  - Over 2x area and power cost for a simple replication of the core and private caches
- Other architectural choices allow us to extract this parallelism with fewer overheads.
  - Such as sharing the core between threads at some granularity
- This module considers the trade-offs involved in achieving this.

© 2021 Arm Limited

Multitasking

- Each of our cores can currently be shared by multiple threads.
  - Mediated and controlled by the operating system
- This enables multitasking by having each thread (task) run for a fixed period of time.
  - Short periods give the appearance that all threads actually run concurrently.
- The process of changing between running threads is called a context switch.
  - The OS saves (switches out) the state (context) of the running thread.
  - And replaces it with the state of another thread, which can then run on the core.



© 2021 Arm Limited

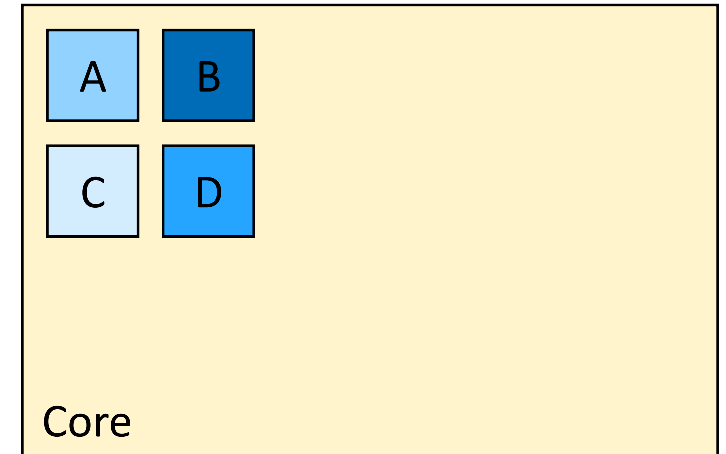
Multitasking

- During a context switch, the OS has to save the thread’s state, which for the core is
  - Contents of the (architectural) register file
  - Program counter (PC)
  - Memory-management information, such as the page table base register
- Saving all this state can be very slow.
  - Usually, 100 s of cycles
- Can we improve this situation to achieve
  - Fewer cycles required for context switching
  - Better throughput from the core
- Without the full-blown costs of going to multicore?

© 2021 Arm Limited

Multithreading

- Reducing the context-switch overhead is easy if we're prepared to pay a small price for it.
- By adding extra storage to the core, we can hold the context for more than one thread.
  - The price here is area devoted to the context storage.
- This means that a context switch between two threads with stored contexts is much faster.
  - Only a few cycles instead of several hundreds
  - Because the storage is all very close to the core
- We now have a core capable of multithreading.
- This core works at arbitrary granularity.
  - i.e., with many cycles between each switch
  - Or even selecting a new thread every cycle



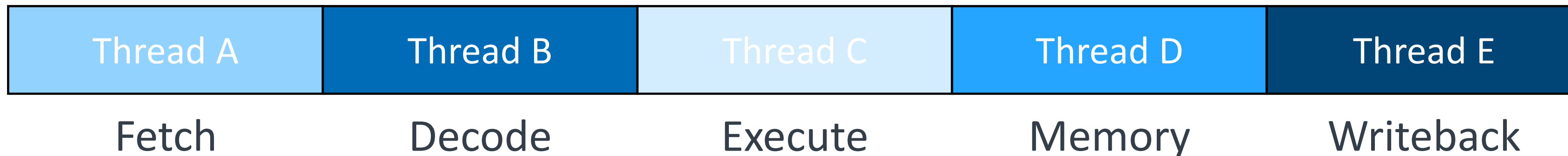
© 2021 Arm Limited

Fine-grained Multithreading

© 2021 Arm Limited
arm

Fine-grained Multithreading

- Now that we have the ability to switch to a new thread, when should we do it?
- With fine-grained multithreading, we switch to a new thread every cycle.
- This means that in each stage of the pipeline is an instruction from a different thread.
- We can use a simple round-robin policy for choosing the next thread to fetch from.
  - If the thread is stalled, send a NOP instead.



© 2021 Arm Limited

Trade-offs of Fine-grained Multithreading

Provided no two instructions from the same thread are in the pipeline together

Benefits

- No thread-switch overhead
- No need for pipeline interlocking
  - No two instructions will have a data dependency through registers (they might through memory).
- No need for branch prediction logic
  - All branches executed before the next instruction are fetched from the same thread.
- Hides short pipeline bubbles
- Improved core throughput

Downsides

- Need to support a large number of threads
  - So as to hide most short-latency pipeline bubbles and avoid nops
  - This comes at an area cost.
- Single-threaded performance suffers.
  - Since threads only get an instruction fetched every N cycles
  - But this can bring strong performance guarantees.

© 2021 Arm Limited

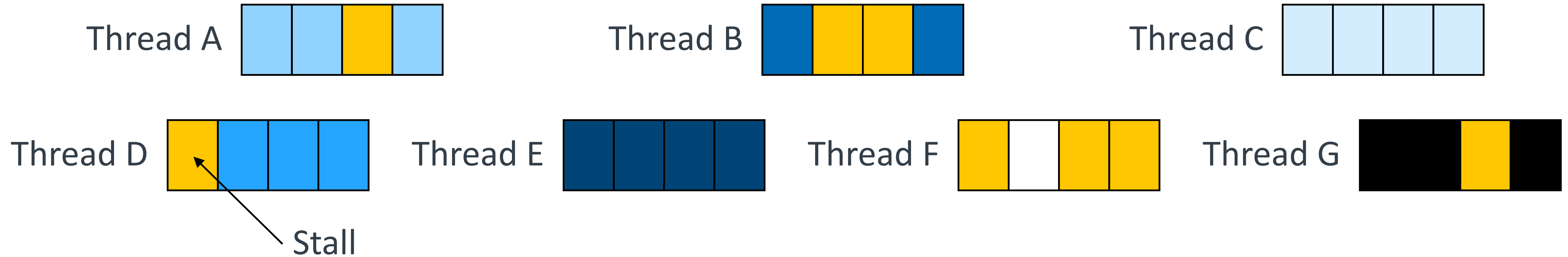
arm

Dynamic Thread Selection

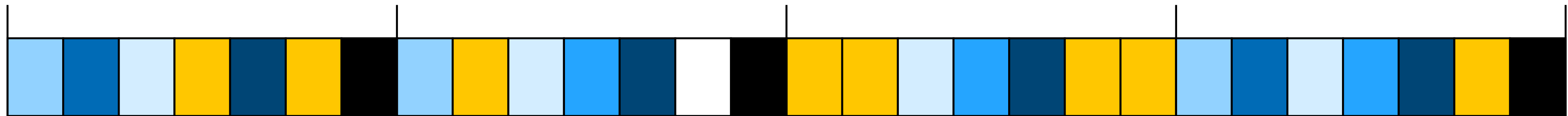
- The most simple approach to fine-grained multithreading *always* fetches from a thread.
  - Even when that thread is stalled – a nop is fetched instead.
- However, the presence of nops reduces the performance of the core.
  - This is wasteful of the time-slot when other threads have actual work to do.
- We can change the basic round-robin thread-switching policy to improve the situation.
  - Order threads as in round robin.
  - Skip a thread if it is stalled and would have sent a nop into the pipeline.
- This kind of dynamic thread selection improves throughput even more.
  - But needs additional logic to implement it
  - And either needs the dependence-check logic adding back in
  - Or extra logic to detect when there are fewer ready threads than pipeline stages

© 2021 Arm Limited

Dynamic Thread Selection



Static round-robin selection



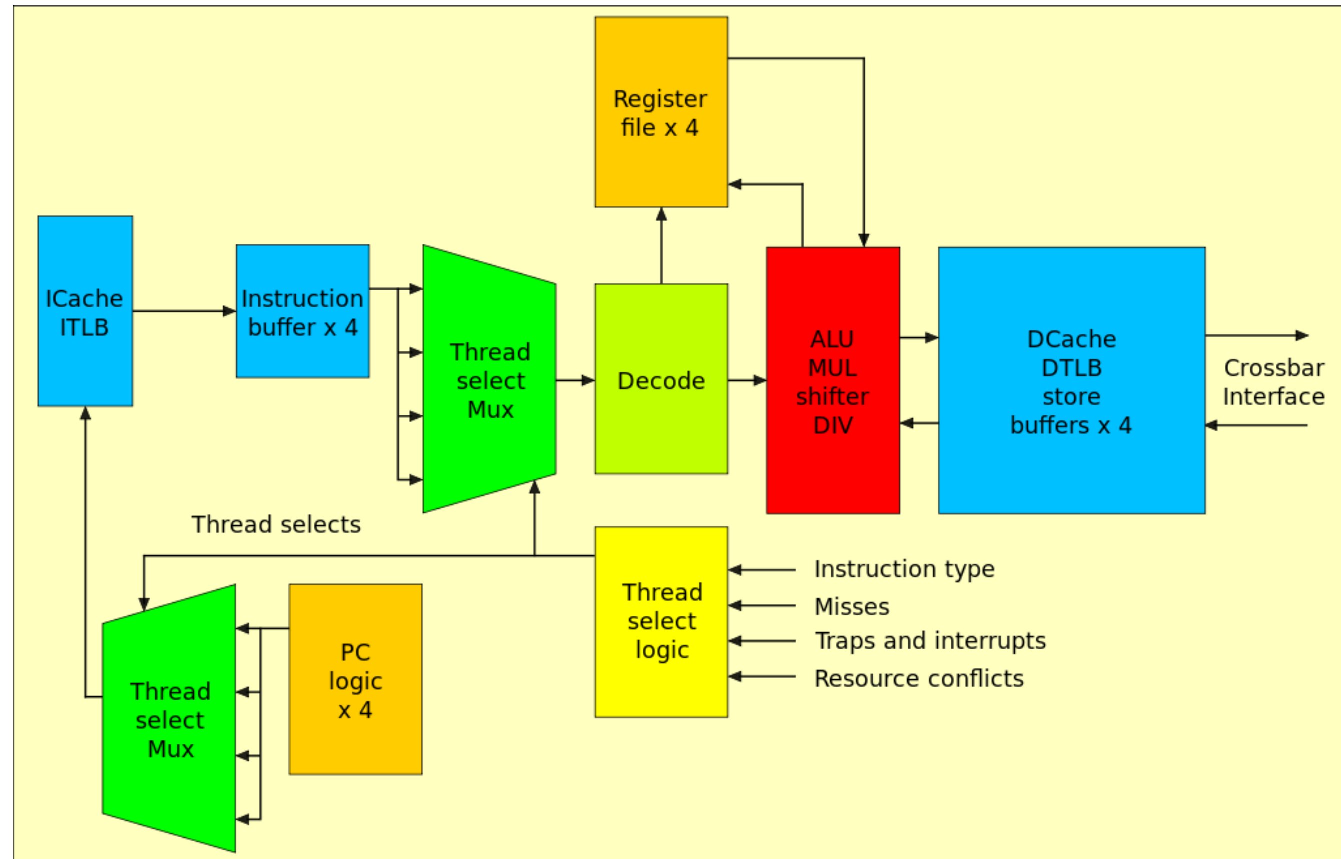
Dynamic round-robin thread selection

Time

© 2021 Arm Limited
arm

Case Study: Sun's Niagara (UltraSPARC T1)

- Each core of eight supports four concurrent threads.
- Threads removed from selection on long-latency events
- This allows provision of only 16KiB instruction and 8KiB data caches.



© 2021 Arm Limited
arm

Coarse-grained Multithreading

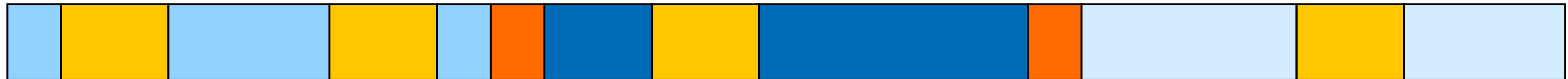
 © 2021 Arm Limited

arm

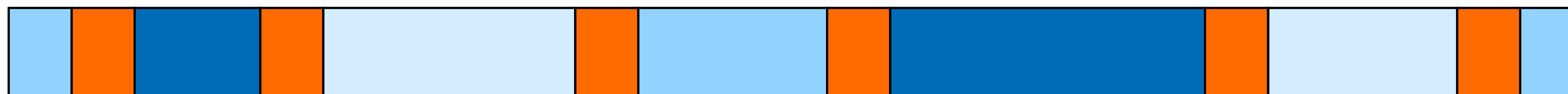
Coarse-grained Multithreading

- An alternative design point is course-grained multithreading.
  - Switching less often (and flushing when doing so) means the core requires less per-thread state.
- An obvious point to switch is when the thread is stalled for some reason, e.g., a cache miss.

Switching at fixed frequency



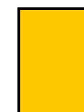
Switching on long-latency event

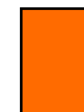


Thread A

Thread B

Thread C

 Stall

 Thread switch

© 2021 Arm Limited

Coarse-grained Multithreading

- The switch to a new thread increases the throughput of the core.
  - Although switching takes time, we only do it for stalls that are longer than that overhead.
- However, care has to be taken in how the switch is implemented.
- In particular, instructions after the long-latency event must be squashed.
  - This adds to the switching time.
- To avoid the additional design complexity of handling instructions from different threads in the pipeline at any given time, the pipeline has to be flushed before switching.
  - This incurs in an overhead.
  - Finer granularities may justify handling instructions from different threads within the pipeline at the same time.

© 2021 Arm Limited

Coarse-grained Multithreading

Reasons for switching threads

- L1 or L2 cache miss
  - Accessing main memory takes hundreds of cycles, and if the thread-switch penalty is small enough, even an L1 miss can be partially hidden.
- Complex ALU operation
  - A floating-point divide may take tens of cycles and may not be pipelined.
- Timeout
  - This ensures fairness for compute-bound threads.
- Higher priority thread becomes ready.

Methods to reduce thread-switch penalty

- Implement a short pipeline.
  - This reduces the time until the pipeline becomes full again.
- Add a thread-switch buffer
  - Essentially prefetch a few instructions from each thread into a buffer so that they can be fetched (for real) immediately after a switch.
- Provide pipeline registers for each thread.
  - So now no instructions need to be squashed on a long-latency event, but added complexity.

© 2021 Arm Limited

Case Study: Intel Montecito (Itanium 2)

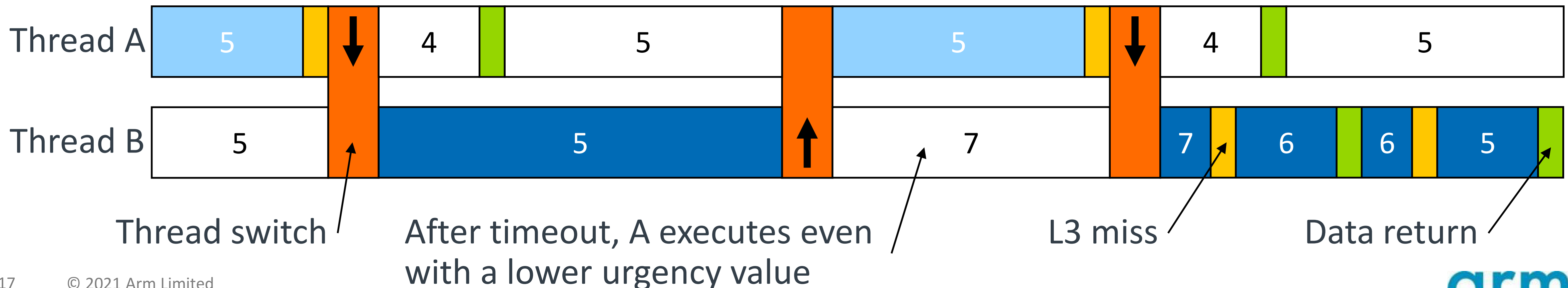
- Support for two threads per core
  - Duplicates all architectural and some microarchitectural state
- Each thread is given an “urgency” value.
  - Larger urgency means higher priority.
  - Urgency is dynamically updated based on system events.

Thread switching possibly occurs on

- L3 cache miss or data return
- Timeout, software hint, or other system event

Diagram shows two threads with events.

- Colored when switched in, white when switched out
- Urgency value inside boxes



© 2021 Arm Limited
arm

Beyond Coarse-grained Multithreading

- Coarse-grained multithreading allows us to hide some of the stall latency.
  - The core performs useful work again once thread switching has occurred.
  - And the techniques discussed previously help reduce the thread-switch penalty.
- However, we don’t switch often, so smaller pipeline stalls still exist.
  - For example, ALU operations taking only a few cycles
- Either pay the price of per-thread pipeline registers or suffer the switching penalty.
  - Per-thread pipeline registers may be overkill if switching occurs infrequently.
- Or we can move in the other direction and allow instructions from multiple threads per pipeline stage simultaneously.

© 2021 Arm Limited

Simultaneous Multithreading (SMT)

© 2021 Arm Limited

Simultaneous Multithreading

- Coarse- and fine-grained multithreading enable higher throughput from a core.
  - By switching to a new thread either every cycle or on certain events
- These are called temporal multithreading.
  - Each pipeline stage holds instructions from only a single thread – the core is shared across time.
  - Instructions from different threads can occur in the pipeline at different stages.
- However, in superscalar cores, this does not increase pipeline utilization.
  - During periods of low IPC in a particular thread, some functional units are idle.
- Simultaneous multithreading is a way to tackle this.
  - Instructions from multiple threads can occupy a pipeline stage at the same time.
  - This increases the number of ready instructions available to execute.

© 2021 Arm Limited

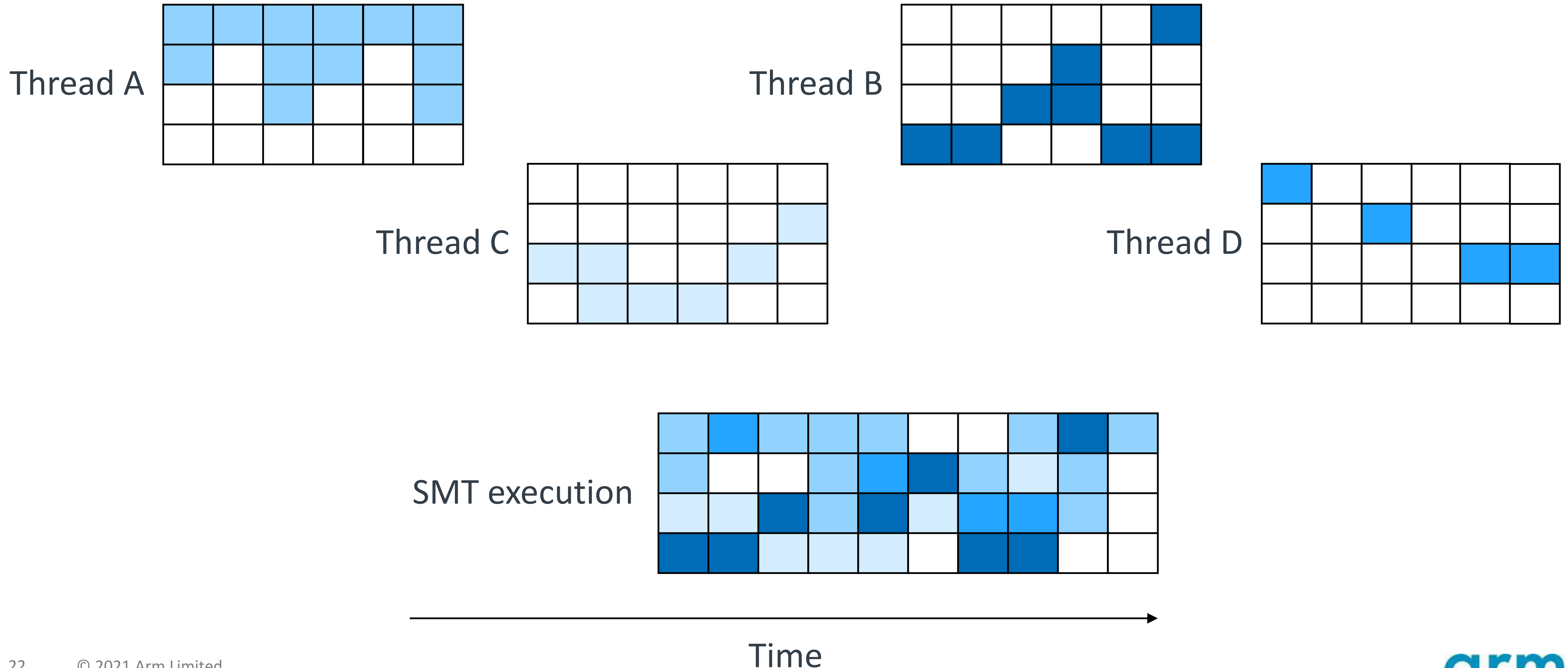
Simultaneous Multithreading

- SMT allows instructions from different threads to be in-flight concurrently.
  - All the way through the pipeline
- Instructions from multiple threads are:
  - Fetched, decoded, renamed, dispatched, issued, executed, and committed each cycle
- Microarchitectural resources throughout the pipeline are shared between threads.
  - This means that there should be fewer unused resources because more independent instructions exist.
- As with other forms of multithreading, the designer has to duplicate architectural state.
  - PC, register file, page table base register, etc.
- And implement fetch logic to fetch instructions from one or more threads in each cycle
  - As well as the additional logic to distinguish between threads within the pipeline

© 2021 Arm Limited

SMT Example

Instruction issue in a 4-way superscalar core



© 2021 Arm Limited

SMT Policies

Fetch policies

- Flexibility in how instructions enter the pipeline
- Fetch from one thread only in each cycle
  - Round-robin selection
  - Prioritize one thread over others, for example, the one with the fewest in-flight instructions
- Fetch from multiple threads
  - Needs additional logic to fetch from multiple cache lines in the same cycle

Issue policies

- Flexibility in how instructions are issued to the functional units when ready
- Issue from all threads in each cycle
  - Skip threads that have no ready instructions
- Prioritize instructions according to
  - Their age (oldest first)
  - Whether they are speculative (e.g., following an unresolved predicted branch)

© 2021 Arm Limited

SMT Downsides

- There are two main drawbacks with SMT, and fine-grained multithreading.
- First, single-threaded performance may suffer.
  - If two or more threads execute, they will contend for resources.
  - Given the extra logic required within the core, the clock rate may not be as high as it could be otherwise.
- Second, the complexity of the core increases.
  - Extra hardware required throughout the pipeline, as well as within the TLB
  - This requires more design, test, and verification time.
  - The load/store queue, in particular, requires careful attention so as to respect the memory consistency model.

© 2021 Arm Limited

SMT vs Multicore

SMT

- Higher utilization of core structures through sharing
- Single-threaded performance can suffer.
- For a modest area and power increase, this provides support for multiple threads.
- Memory-bound applications are a good fit because idle cycles can be filled.
- CPU-bound applications are a poor fit because they are likely to slow down.

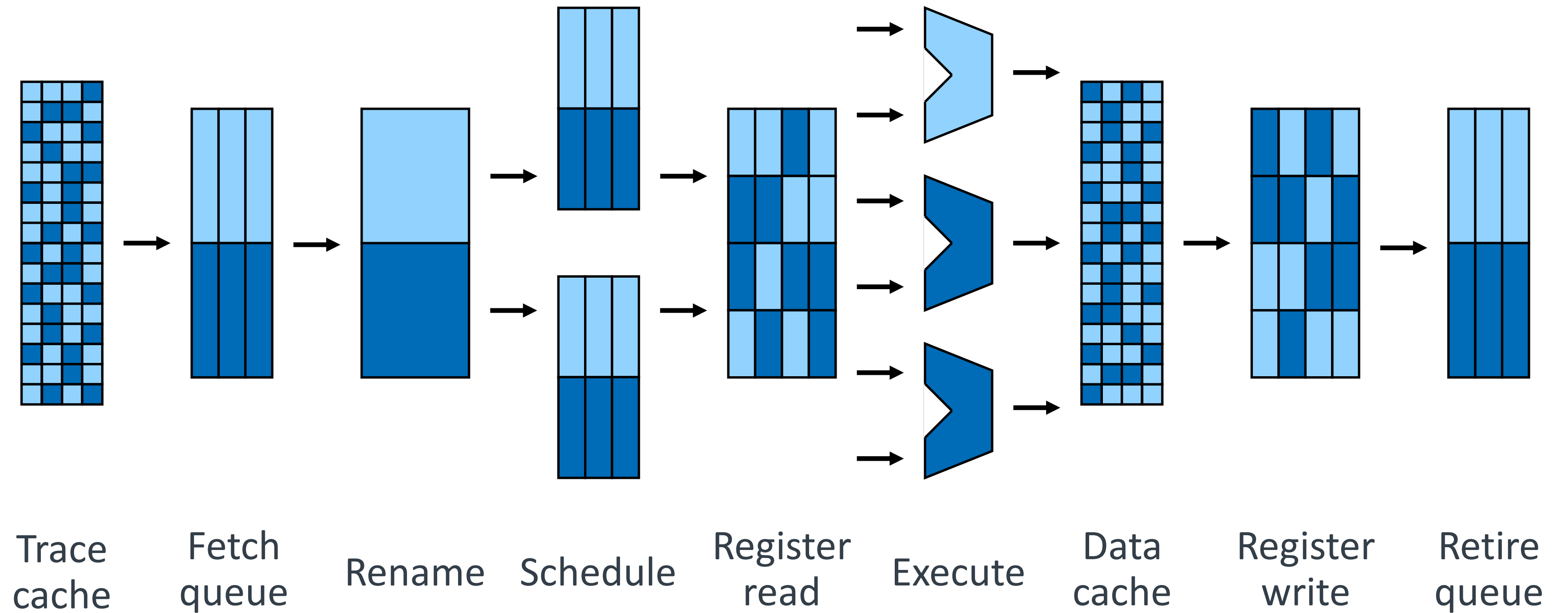
Multicore

- Cores are fully duplicated, so threads have dedicated resources to use.
- Single-threaded performance maintained
- Power and area at least doubled through core duplication
- Memory-bound applications don’t make best use of the core’s resources.
- CPU-bound applications maintain their high IPC.

© 2021 Arm Limited
arm

SMT Case Study

- Intel's Netburst microarchitecture supports 2 threads.
- Performance boosted by 25%
- Die area increased only 5%
- Three types of sharing for microarchitectural structures

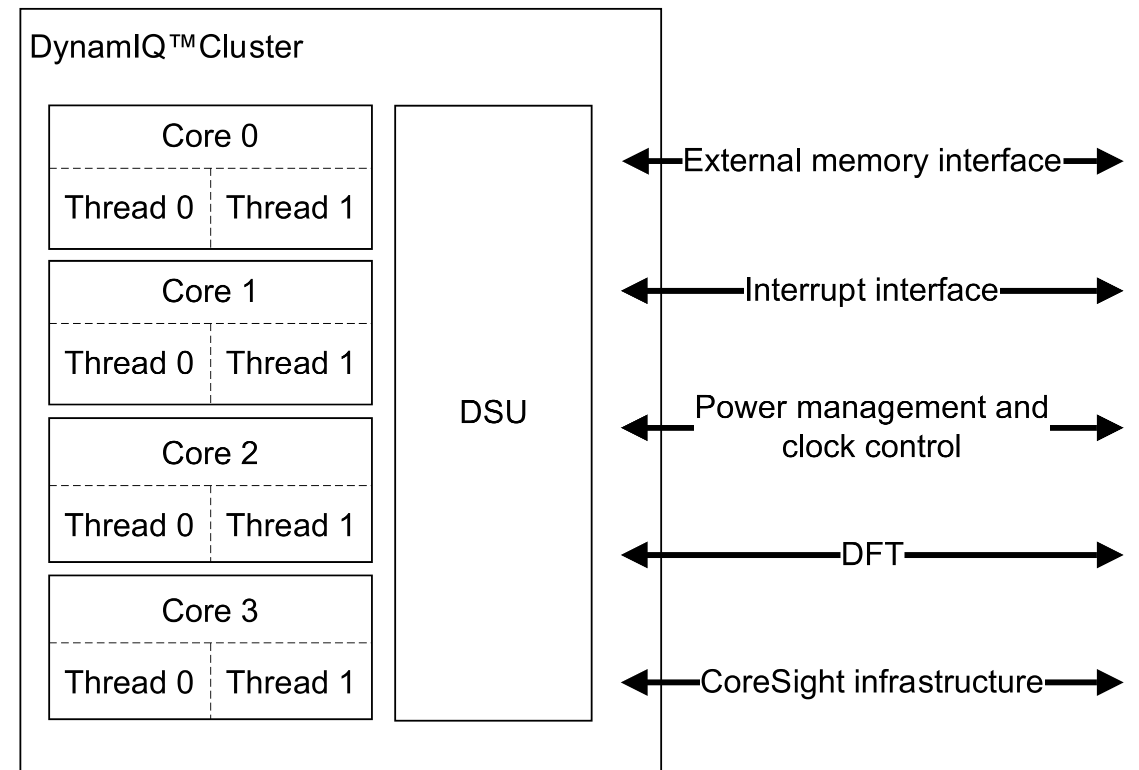


© 2021 Arm Limited

Case Study: Cortex-A65

Scheduled for 2020

- Arm’s first SMT Cortex-A core
- Cortex-A65 implements a 64-bit ISA only
- Built on DynamIQ technology
  - DSU contains all the interfaces to connect to the system on chip (SoC).
- Dual-threaded, out-of-order execution
  - Each thread is a PE.
- Separate L1 instruction and data cache
  - L2 cache is optional.
  - L3 cache in DSU



© 2021 Arm Limited
arm

Summary

- We can increase the throughput of a core by implementing multithreading.
- Coarse-grained multithreading is most similar to multitasking.
  - Thread switching occurs on long-latency events.
  - Because thread contexts are kept near the core, thread switching is much faster than an OS context switch.
- Fine-grained multithreading increases throughput further.
  - Can eliminate many short pipeline bubbles
  - And can also provide strong performance guarantees in certain microarchitectures
- SMT provides full sharing within each stage of the pipeline.
  - This enables higher utilization of core structures.
- All forms of multithreading exploit thread-level parallelism to improve performance.

© 2021 Arm Limited