Lecture 14:
CPU – Multithreading

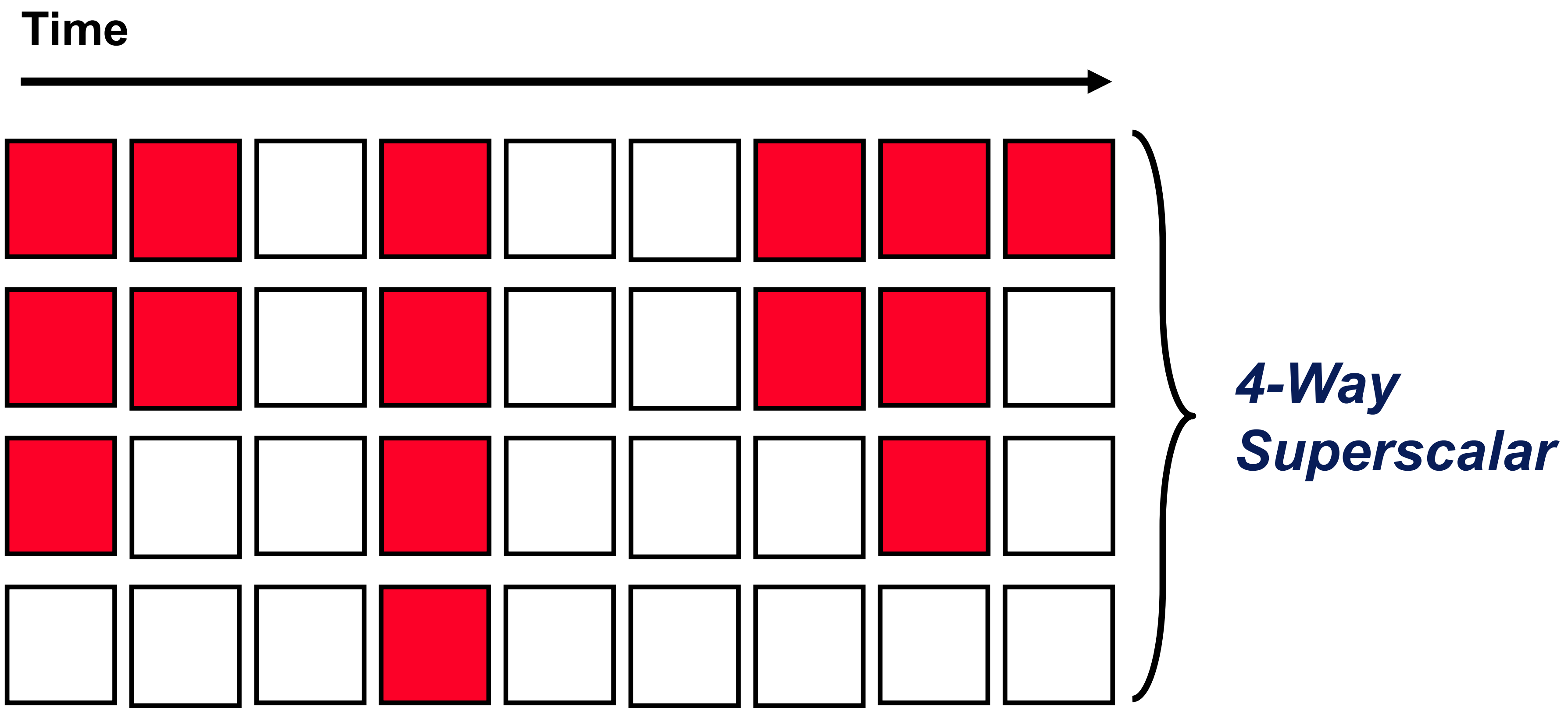
Hunjun Lee
hunjunlee@hanyang.ac.kr

So far …

- We learned about ways to minimize the idle cycles
  - **Data forwarding**
    - Use the data that are yet to be committed
  - **Branch predictions**
    - Predict the next instruction addresses for control instructions
  - **Out-of-order execution**
    - Execute instructions that do not suffer from true dependencies
  - **Load-store queue**
    - Prioritize load to resolve dependencies first (+ support load forwarding …)

- This greatly reduces the idle cycles, but is this sufficient?
  - Deeper pipelines, long latency memory operations, limited ROB capacity and RS, Branch mispredictions, superscalar, …

Underutilization issues …



It is extremely hard to fully utilize the entire pipeline due to large number of arithmetic units, superscalar, branch mispredictions, ……………..

Performance and utilization

- Utilization ➔ (actual IPC / peak IPC)
  - Even though the CPU has a large number of units, it cannot use them …
- Even moderate superscalars (e.g., 4-way) are not fully utilized
  - IPC: 1.5 ~ 2 → < 50% utilization
- Multithreading (MT)
  - One thread cannot fully utilize CPU (maybe we can with more threads)
  - Improve utilization by executing multiple threads on a single CPU

Basics of Multithreading

- Thread basics
  - Instruction stream with state (registers and memory)
  - Register state is also called a thread context
- The threads can be both from the same process or from different processes
- You can either use software or hardware to support for multithreading!
  - SW: The CPU stores the context to the memory and load it back again
  - HW: The old thread's context is kept in a separate register file
    - What's mostly done in modern CPUs

Latency vs. Throughput

- MT trades latency for throughput
  - – Sharing processor degrades latency of individual threads
  - + But improves aggregate latency of both threads
  - + Improves utilization

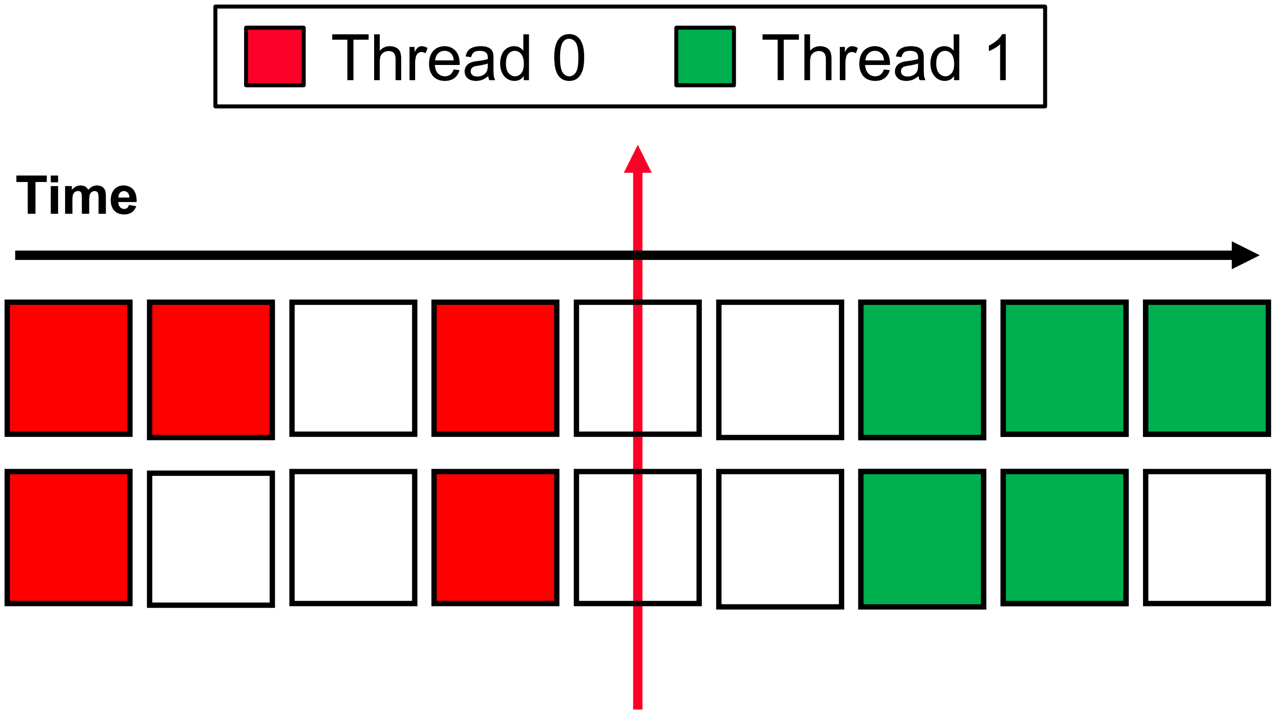
- Example …
  - - Thread A: individual latency=10s, latency with thread B=15s
  - - Thread B: individual latency=20s, latency with thread A=25s
    - Sequential latency (first A then B or vice versa): 30s
    - Parallel latency (A and B simultaneously): 25s
  - – MT slows each thread by 5s
  - + But improves total latency by 5s

How to run multiple threads?

- Time sharing
  - Execute a single thread at a time
  - Switch threads on a long-latency operation (e.g., cache miss)
  - Also known as “switch-on-miss” multithreading

- Space sharing
  - Simple way (Fine-grain multithreading): execute different threads across pipeline depth (i.e., pipeline interleaving)
    - Fetch a different thread each cycle
  - Advanced method (Simultaneous multithreading): both across pipeline depth + width
    - Fetch and issue each cycle from multiple threads
    - Need a policy to decide from which to thread fetch

Coarse-grain Multithreading



Switch to Thread1 on an L2 miss

Multithreading minimizes single-thread performance but only hide very long latencies

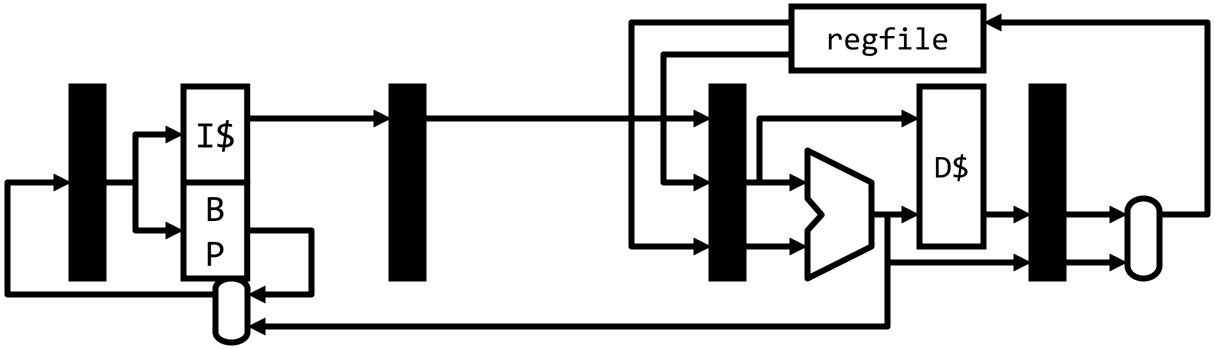
Coarse-Grain Multithreading (CGMT)

- Coarse-Grain Multi-Threading (CGMT)
  - **Thread scheduling policy**
    - Designate a “preferred” thread (e.g., thread A)
    - Switch to thread B on thread A’s L2 miss
    - Switch back to A when A’s L2 miss returns

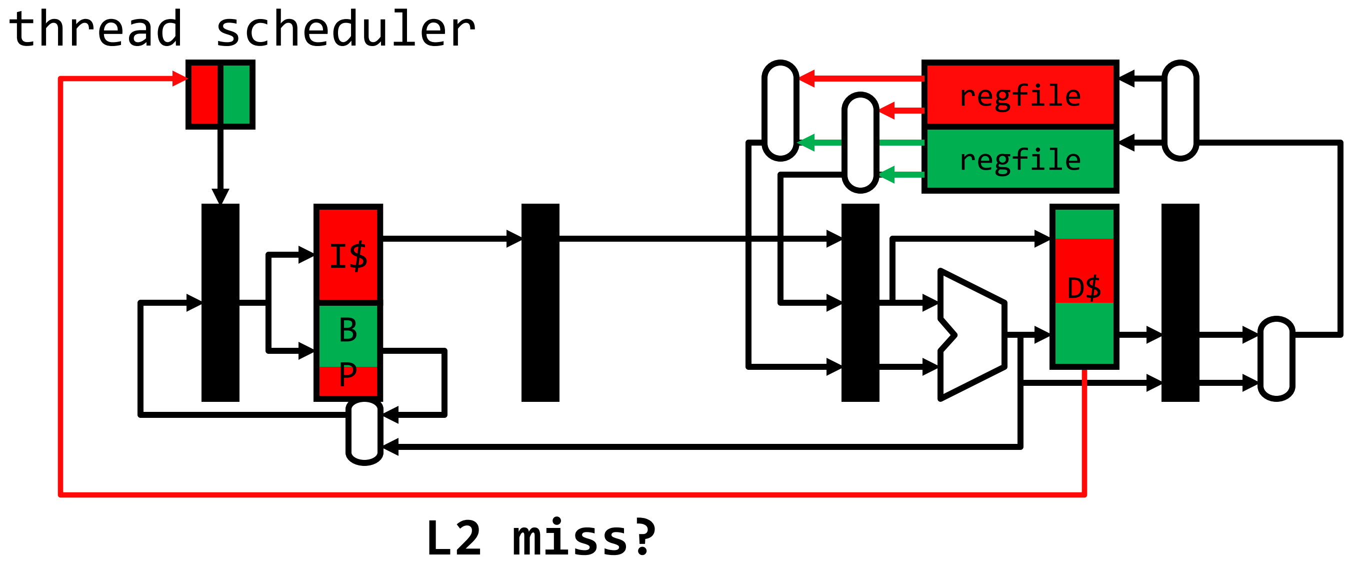
+ Sacrifices very little single thread performance (of one thread)
- Tolerates only long latencies (e.g., L2 misses)
- No pipeline partitioning
  - They flush on switch
    - **Can’t tolerate latencies shorter than twice pipeline depth**
    - Why? We need two context switching → which incurs “at least” twich pipeline depth penalty
    - Need short in-order pipeline for good performance

CGMT

- Normal Superscalar

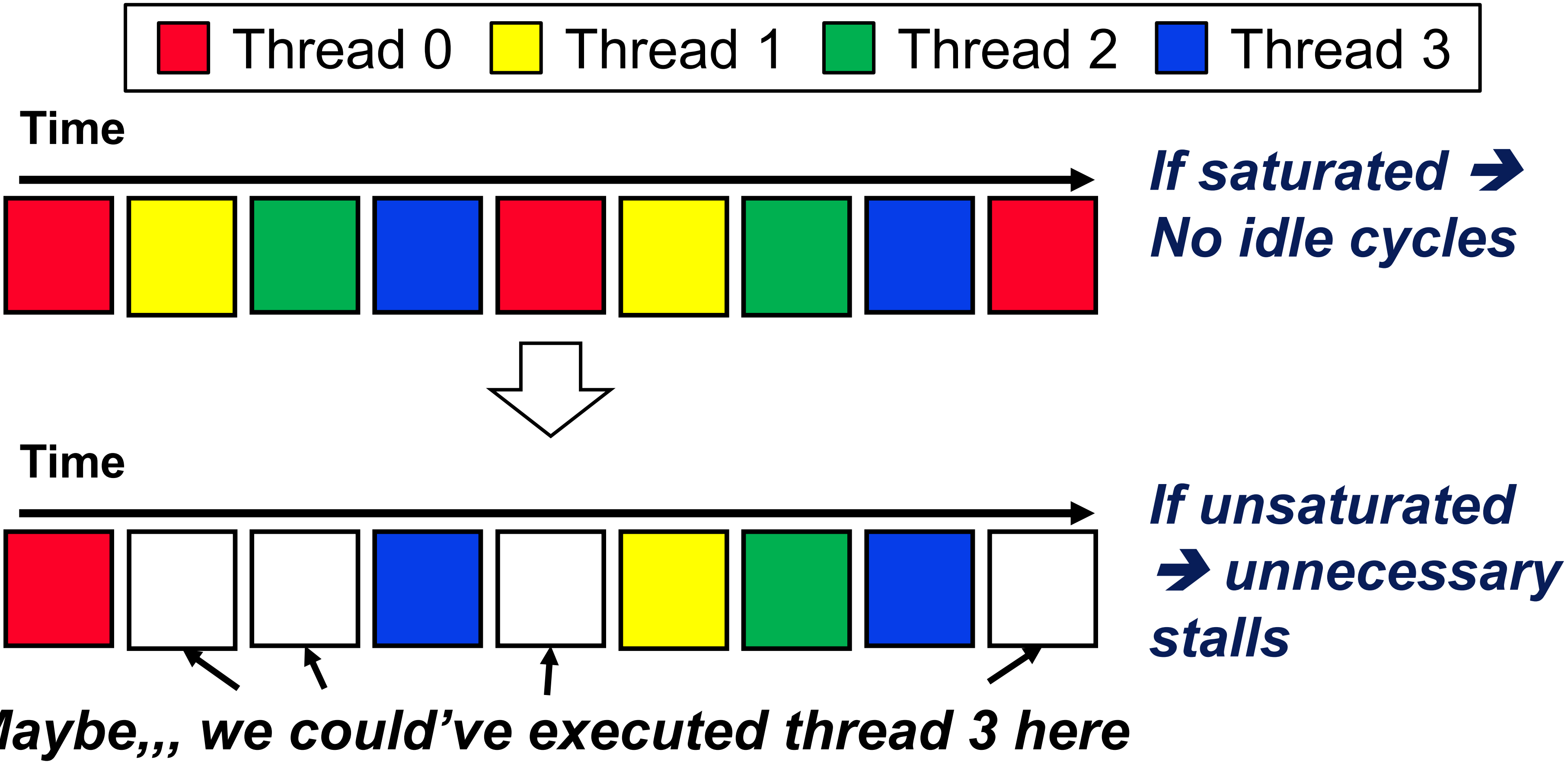


- CGMT



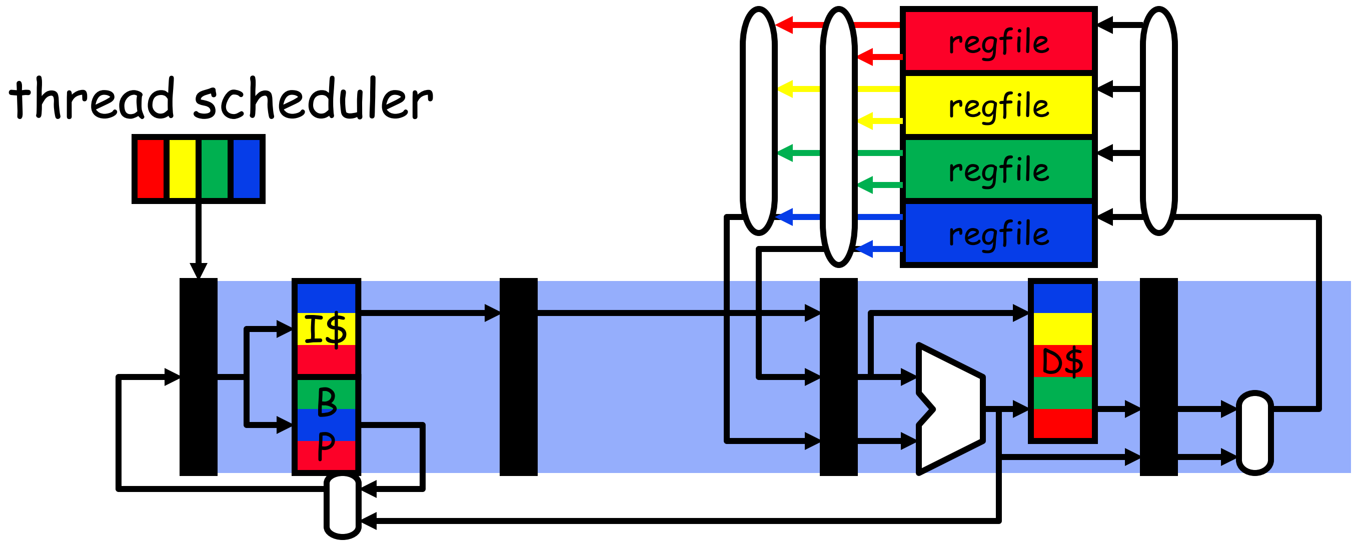
Fine-grain Multithreading (FGMT)

- Why not change the threads very frequently!
  - Extremely simple multithreading strategies
  - Fetch an instruction from different threads every cycle!


🟥 Thread 0 🟨 Thread 1 🟩 Thread 2 🟦 Thread 3
Time
If saturated → No idle cycles
Time
If unsaturated → unnecessary stalls
Maybe,,, we could've executed thread 3 here

Fine-Grain Multithreading

- FGMT
  - (Many) more threads
  - Multiple threads in pipeline at once



Illustrating FGMT: Resource View

🟥 Thread 0 🟨 Thread 1 🟩 Thread 2

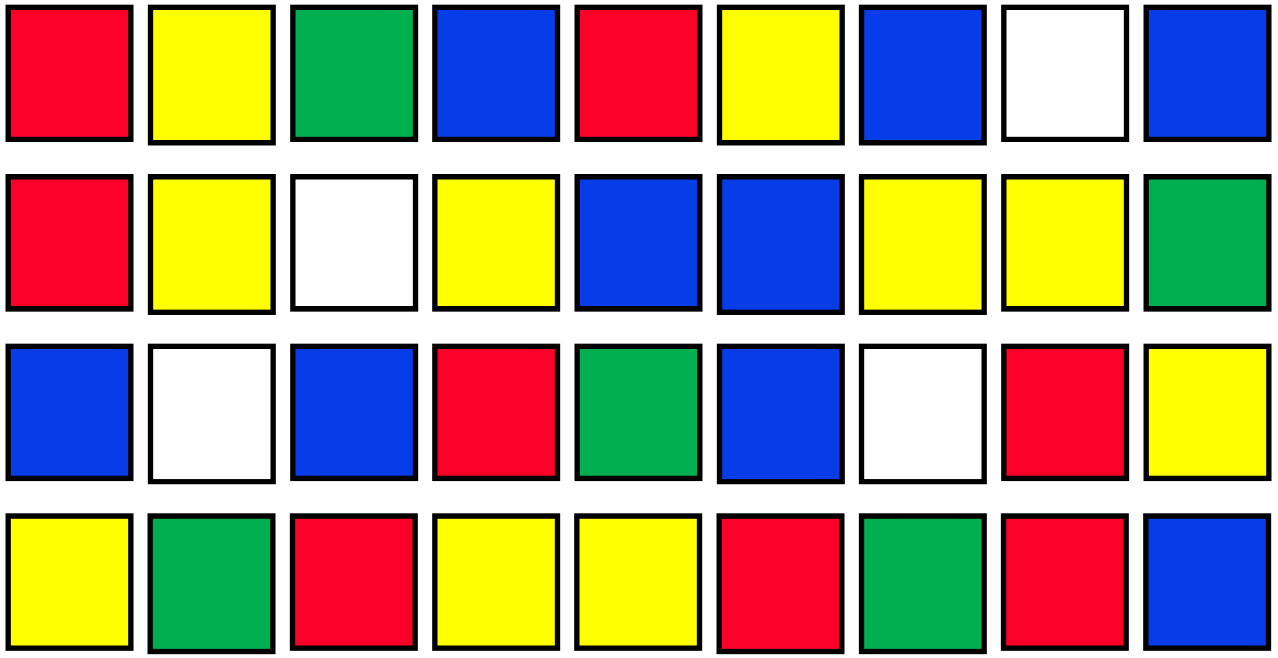
🟦 Thread 3 🟪 Thread 4

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	🟥 I_0	🟨 I_0	🟩 I_0	🟦 I_0	🟪 I_0	🟥 I_1	🟨 I_1	🟩 I_1	🟦 I_1	🟪 I_1	🟥 I_2
ID		🟥 I_0	🟨 I_0	🟩 I_0	🟦 I_0	🟪 I_0	🟥 I_1	🟨 I_1	🟩 I_1	🟦 I_1	🟪 I_1
EX			🟥 I_0	🟨 I_0	🟩 I_0	🟦 I_0	🟪 I_0	🟥 I_1	🟨 I_1	🟩 I_1	🟦 I_1
MEM				🟥 I_0	🟨 I_0	🟩 I_0	🟦 I_0	🟪 I_0	🟥 I_1	🟨 I_1	🟩 I_1
WB					🟥 I_0	🟨 I_0	🟩 I_0	🟦 I_0	🟪 I_0	🟥 I_1	🟨 I_1

Fine-grain Multithreading (FGMT)

- Although FGMT potentially suffers from unnecessary stalls, it greatly simplifies the CPU
- Example: Let’s say the CPU is pipelined into 10 stages and supports 10-thread FGMT
  - Only a single instruction for each thread exists in the entire pipeline
    - No forwarding support
    - No branch prediction
    - No OoO support
    - No flush on switch
    - etc
- It is not widely adopted today

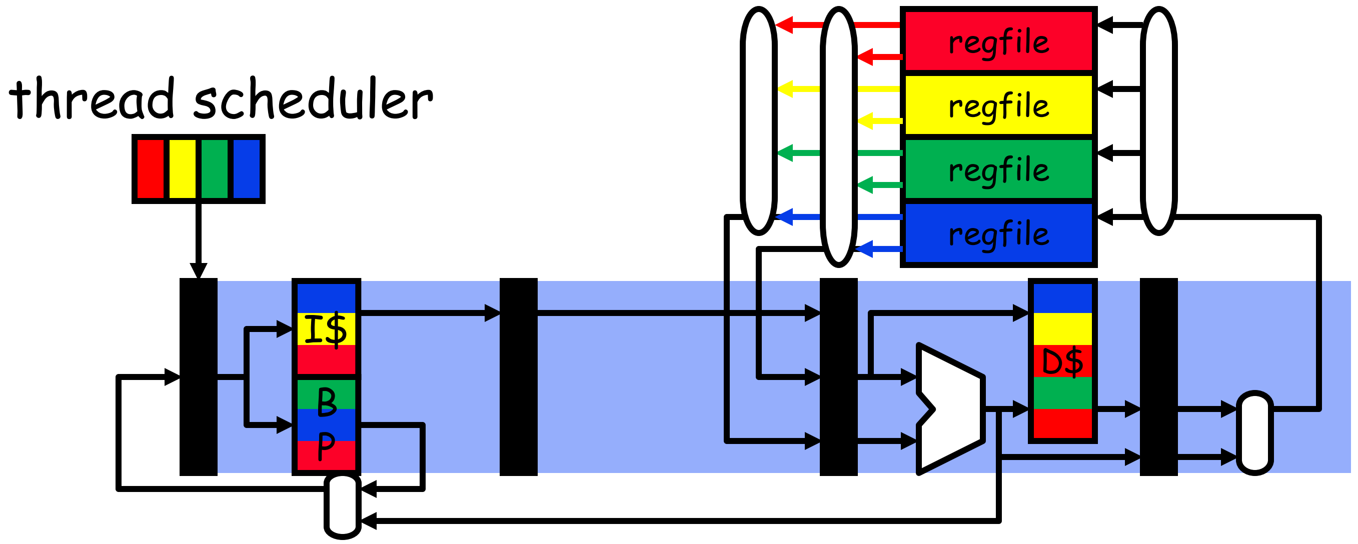
Simultaneous Multithreading (SMT)
(a.k.a. Hyperthreading)



SMT maximizes the utilization (dispatch independent instructions which may or may not be from the same thread)

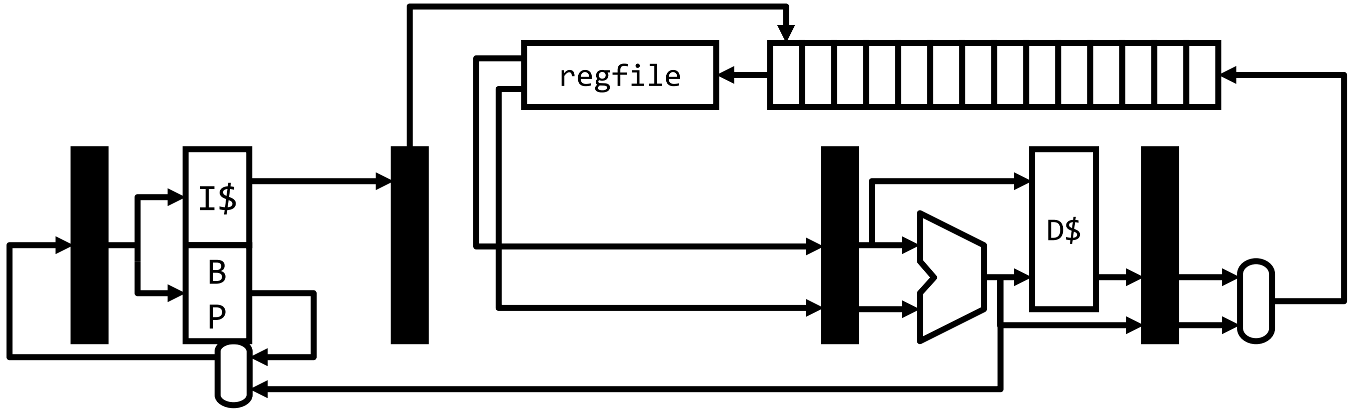
Fine-Grain Multithreading

- FGMT
  - (Many) more threads
  - Multiple threads in pipeline at once

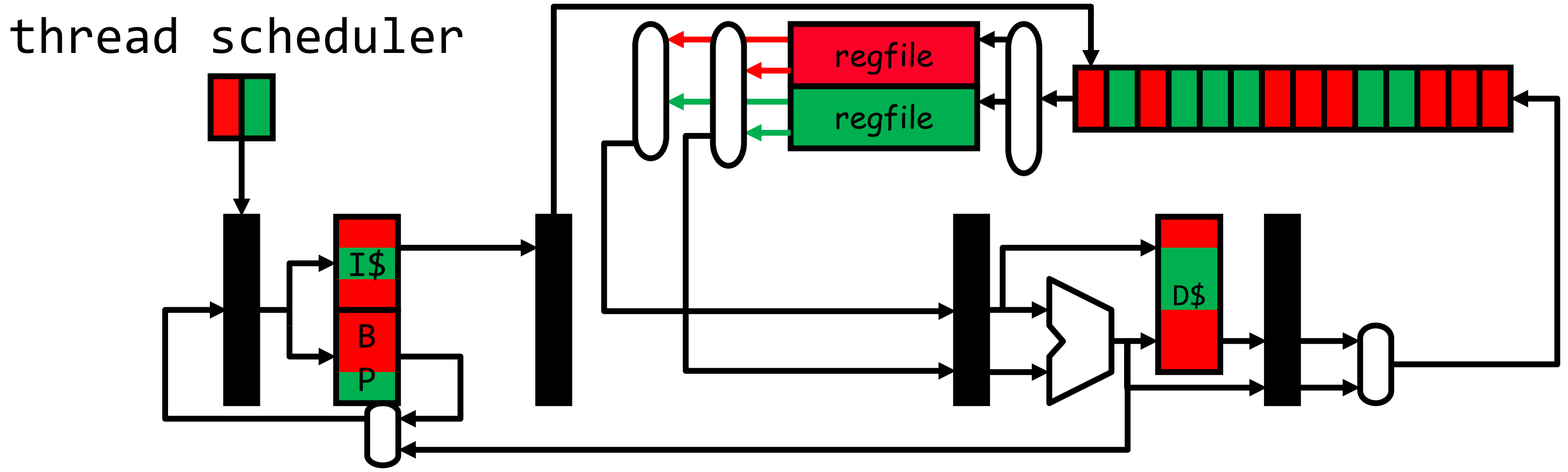


Simultaneous Multithreading (SMT)

- Ordinary Superscalar



- SMT
  - Replicate map table, share physical register file



Simultaneous Multithreading (SMT)
(a.k.a. Hyperthreading)

- Can we multithread an out-of-order machine?
  - Don't want to give up performance benefits
  - Don't want to give up natural tolerance of long memory latency
- **Simultaneous multithreading (SMT)**
  - + Tolerates all latencies (e.g., L2 misses, mispredicted branches)
  - ± Sacrifices some single thread performance due to resource sharing
  - - Thread scheduling policy
    - Round-robin (just like FGMT) or something else?
  - - Pipeline partitioning
    - Dynamic, but hard to be optimal
  - - Example:
    - Pentium4 (hyper-threading): 5-way issue, 2 threads …
    - Intel Xeon processors

Potential issues in SMT

- Fetch policies
  - Similar to FGMT, switch PCs every cycle
  - Or flexible independent fetches (e.g., 2.8 fetch policy)
    - Can fetch from two threads (up to eight instructions), but fetch different number of instructions depending on the stalls
- Cache interference
  - General concern for all MT variants
  - Can the working sets of multiple threads fit in the caches?
  - Memory-sharing thread applications can help here
    - + Same instructions → share I$,
    - + Shared data → less D$ contention
    - MT is good for "server" workloads (e.g., database, web server)
- Large map table and physical register file
  - #phys-regs = (**#threads** * #arch-regs) + #in-flight insns

Potential issues in SMT

- What about branch predictor?
  - Without SMT: a single thread fills up the branch predictor (PHT) + branch target buffer
  - Utilizes the history to index the PHT and makes a predictor → This is good
  - How about SMT?
    - Share global history buffer?
    - Share pattern history table?
    - Separate branch target buffer?

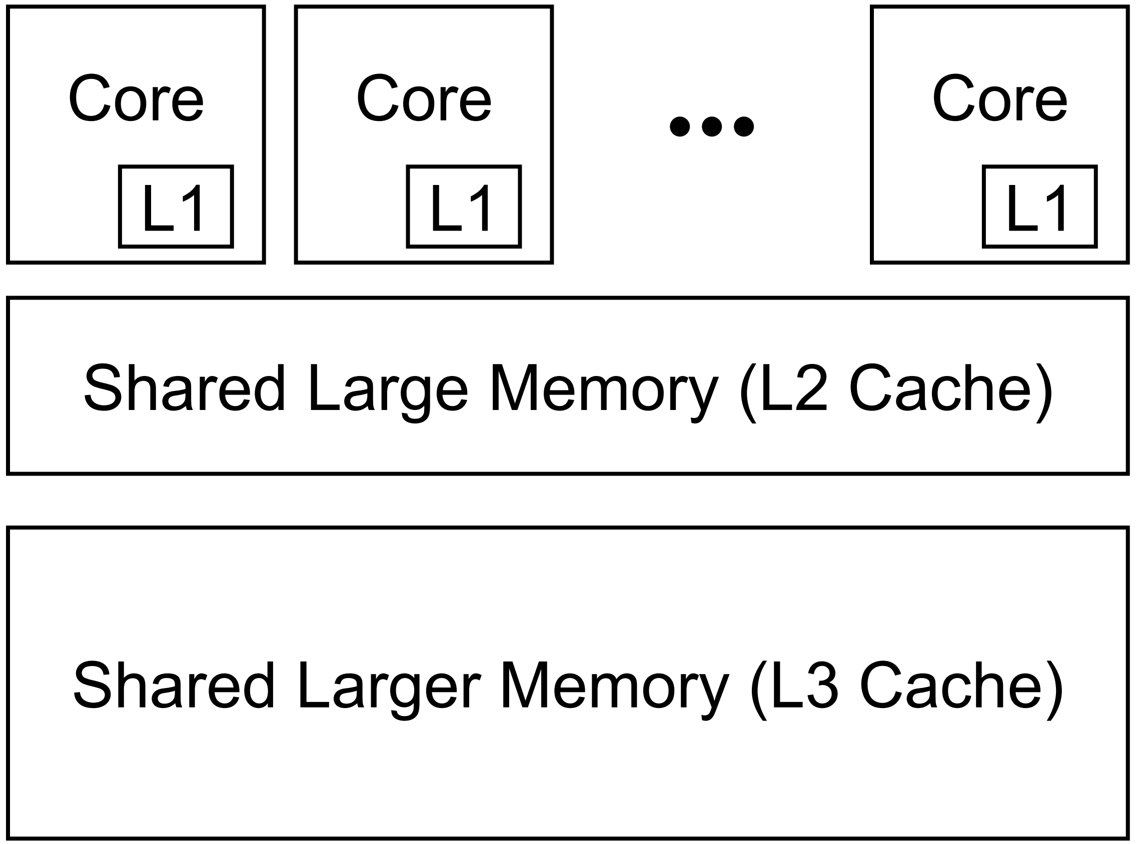
Potential issues in SMT

- How are ROB/LSQ/RS (or cache) partitioned in SMT?
  - There are two ways: Static and Dynamic
- **Static partitioning**
  - Divide the hardware into equal-sized partitions
  - Ensures fairness among different threads (some threads may occupy the slots …)
  - But, suffers from low utilization
- **Dynamic partitioning**
  - Dynamically resize the partition among different threads
  - Higher utilization
  - But, suffers from fairness issues
  - Which thread to fetch? ➔ e.g., fetch a thread having the fewest in-flight instructions

There is no one-for-all solution (e.g., AMD → dynamic cache partition, Intel → static cache partition)

Other method

- SMT is designed to execute multiple threads within a single core …
- Alternatively, we can use multiple cores instead …


Core
L1
Core
L1
•••
Core
L1
Shared Large Memory (L2 Cache)
Shared Larger Memory (L3 Cache)

SMT vs. CMP

- If you wanted to run multiple threads would you build
  - Chip multiprocessor (CMP): multiple separate pipelines?
  - A multithreaded processor (SMT): a single larger pipeline?
- Both will get you throughput on multiple threads
  - CMP will be simpler, possibly faster clock
  - SMT will get you better performance (IPC) on a single thread
- Actually modern CPUs use both of them …
  - Mostly, hyperthreading (2 threads per core) * multiple cores
- But, more recently, Intel removed Hyper-threading …
  - Why? The modern workloads are bounded by the memory bandwidth (not latency) → Think about AI services and databases …

Sidenote: Defeaturing things

- Computer architects are developing tons of new features for higher performance
- We are afraid of “de”featuring what they have already developed!
- But, sometimes, removing features can be as important as adding a new feature (especially as the architects are becoming more and more complex)

Question?

Announcements:	
Reading:	*finish reading P&H Ch.6.4*
Handouts:	*none*