Advanced ILP Techniques

Pipelining Delivers CPI = 1 at Best

- How to get CPI < 1?
  - Issuing and completing multiple instructions/cycle
  - Challenge: checking and resolving dependences among the instructions issued at the same cycle

- Lecture outline:
  - Multiple issue
  - Dynamic scheduling, multiple issue, and speculation
  - Multithreading: exploiting thread-level parallelism to improve uniprocessor throughput

Issuing Multiple Instructions/Cycle

- Static multiple issue
  - Compiler handles data and control hazards and decides what instructions can be issued in the same cycle
  - Often restrict mix of instructions can be initiated in a clock
  - Recompilation required for machines with diff. pipelines

- Dynamic multiple Issue
  - Fetch, decode, and commit multiple instructions
  - Use dynamic pipeline scheduling, HW-based speculation and recovery
  - Always beneficial if compiler can help, but recompiling not required for new machines

Hardwere Strategies for Multiple Issue

- **Superscalar:** varying number of instructions/cycle (1 to 8), scheduled by HW (dynamic)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium 4, i7
- **Multithreading:** exploiting thread-level parallelism to improve uniprocessor throughput
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium 4, i7
- **Vector Processing:** Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors

Multiple Issue

- Issue packet: group of instructions from fetch unit that could potentially issue in 1 clock
  - If instruction causes structural hazard or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction does not issue
  - 0 to N instruction issues per clock cycle, for N-issue

- Performing issue checks in 1 cycle could limit clock cycle time:
  - => issue stage usually split and pipelined
  - 1st stage decides how many instructions from within this packet can issue,
  - 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important

Multiple Issue Challenges

- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  - Register file: need 2x reads and 1x writes/cycle
  - Rename logic: must be able to rename same register multiple times in one cycle! For instance, consider 4-way issue:

```
add r1, r2, r3          add p11, p4, p7
sub r4, r1, r2    ⇒     sub p22, p11, p4
lw  r1, 4(r4)           lw  p23, 4(p22)
add r5, r1, r2          add p12, p23, p4
```

    Imagine doing this transformation in a single cycle!
  - Result buses: Need to complete multiple instructions/cycle
    - So, need multiple buses with associated matching logic at every reservation station.
    - Or, need multiple forwarding paths

Superscalar Dynamic Scheduling

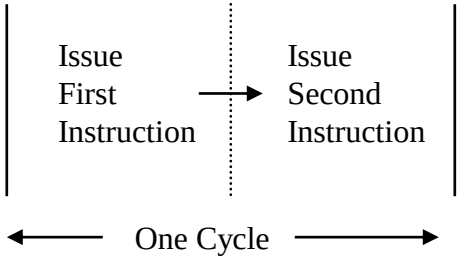
- The Tomasulo dynamic scheduling algorithm is extended to issue more than one instruction per cycle.
- However the restriction that instructions must issue in program order still holds to avoid violating instruction dependencies (construct correct dependency graph dynamically).
  - The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.
- How to issue two instructions and keep in-order instruction issue for Tomasulo?
- Simplest Method: Restrict Type of Instructions Issued Per Cycle
- To simplify the issue logic, issue one integer + one floating-point instruction per cycle (for a 2-way superscalar).
  - 1 Tomasulo control for integer, 1 for floating point.
- FP loads/stores might cause a dependency between integer and FP issue:
  - Replace load reservation stations with a load queue; operands must be read in the order they are fetched (program order).
  - Replace store reservation stations with a store queue; operands must be written in the order they are fetched.
    - Load checks addresses in Store Queue to avoid RAW violation
      - (get load value from store queue if memory address matches)
    - Store checks addresses in Load Queue to avoid WAR, and checks Store Queue to avoid WAW.

Superscalar Dynamic Scheduling

❖ Three techniques can be used to support multiple instruction issue in Tomasulo without putting restrictions on the type of instructions issued per cycle:

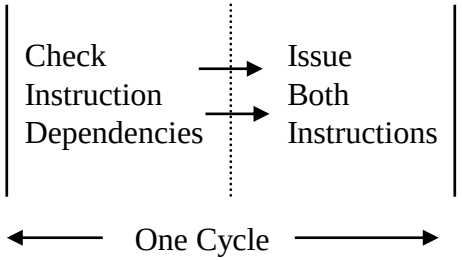
1 Issue at a higher clock rate so that issue remains in order.

✧ For example for a 2-Issue supercalar issue at 2X Clock Rate.



2 Widen the issue logic to handle multiple instruction issue

✧ All possible dependencies between instructions to be issues are detected at once and the result of the multiple issue matches in-order issue



2-Issue superscalar

0, 1 or 2 instructions issued per cycle for either method

Why?

For correct dynamic construction of dependency graph:
The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.

Superscalar Dynamic Scheduling

3 To avoid increasing the CPU clock cycle time in the last two approaches, multiple instruction issue can be spilt into two pipelined issue stages:

- ✧ Issue Stage One: Decide how many instructions can issue simultaneously checking dependencies within the group of instructions to be issued + available RSs, ignoring instructions already issued.
- ✧ Issue Stage Two: Examine dependencies among the selected instructions from the group and the those already issued.

- ❖ This approach is usually used in dynamically-scheduled wide superscalars that can issue four or more instructions per cycle.
- ❖ Splitting the issue into two pipelined staged increases the CPU pipeline depth and increases branch penalties
  - ✧ This increases the importance of accurate dynamic branch prediction methods.
- ❖ Further pipelining of issue stages beyond two stages may be necessary as CPU clock rates are increased.
- ❖ The dynamic scheduling/issue control logic for superscalars is generally very complex *growing at least quadratically* with issue width.
  - ✧ e.g 4 wide superscalar -> 4x4 = 16 times complexity of single issue CPU

Example

- Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:  ld    x2,0(x1)      //x2=array element
       addi  x2,x2,1       //increment x2
       sd    x2,0(x1)      //store result
       addi  x1,x1,8       //increment pointer
       bne   x2,x3,Loop    //branch if not last
```

- Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Assume that up to two instructions of any type can commit per clock.

Performance for a two-issue, dynamically scheduled processor, without speculation

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	`ld x2,0(x1)`	1	2	3	4	First issue
1	`addi x2,x2,1`	1	5		6	Wait for `ld`
1	`sd x2,0(x1)`	2	3	7		Wait for `addi`
1	`addi x1,x1,8`	2	3		4	Execute directly
1	`bne x2,x3,Loop`	3	7			Wait for `addi`
2	`ld x2,0(x1)`	4	8	9	10	Wait for `bne`
2	`addi x2,x2,1`	4	11		12	Wait for `ld`
2	`sd x2,0(x1)`	5	9	13		Wait for `addi`
2	`addi x1,x1,8`	5	8		9	Wait for `bne`
2	`bne x2,x3,Loop`	6	13			Wait for `addi`
3	`ld x2,0(x1)`	7	14	15	16	Wait for `bne`
3	`addi x2,x2,1`	7	17		18	Wait for `ld`
3	`sd x2,0(x1)`	8	15	19		Wait for `addi`
3	`addi x1,x1,8`	8	14		15	Wait for `bne`
3	`bne x2,x3,Loop`	9	19			Wait for `addi`

Performance for a two-issue, dynamically scheduled processor, with speculation

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	`ld x2,0(x1)`	1	2	3	4	5	First issue
1	`addi x2,x2,1`	1	5		6	7	Wait for `ld`
1	`sd x2,0(x1)`	2	3			7	Wait for `addi`
1	`addi x1,x1,8`	2	3		4	8	Commit in order
1	`bne x2,x3,Loop`	3	7			8	Wait for `addi`
2	`ld x2,0(x1)`	4	5	6	7	9	No execute delay
2	`addi x2,x2,1`	4	8		9	10	Wait for `ld`
2	`sd x2,0(x1)`	5	6			10	Wait for `addi`
2	`addi x1,x1,8`	5	6		7	11	Commit in order
2	`bne x2,x3,Loop`	6	10			11	Wait for `addi`
3	`ld x2,0(x1)`	7	8	9	10	12	Earliest possible
3	`addi x2,x2,1`	7	11		12	13	Wait for `ld`
3	`sd x2,0(x1)`	8	9			13	Wait for `addi`
3	`addi x1,x1,8`	8	9		10	14	Executes earlier
3	`bne x2,x3,Loop`	9	13			14	Wait for `addi`

Limits to Multi-Issue Processors

- Inherent limitations of ILP
  - Need about Pipeline Depth x No. Functional Units of independent operations to keep all pipelines busy
  - Difficulties in building HW
    - Easy: more instruction bandwidth, duplicate FUs
    - Hard: increase ports to RF and memory (bandwidth)

- Most techniques for increasing performance also increase power consumption
  - Growing gap between peak issue rates and sustained performance → performance gain is not linearly proportional to power increase

How to Find More Parallelism?

- Hardware?
- Compiler?
- Runtime environment, e.g., virtual machine?

- The key is to find independent instructions
- But, our attentions are focused mainly on the current thread of execution
  → why not from other programs or threads?
  → the instructions are completely independent
- Programmer may need to be involved
  - Parallel programming, program annotations, ...

Multi-Threading

- A multithreaded CPU is not a parallel architecture, strictly speaking; multithreading is obtained through a single CPU, but it allows a programmer to design and develop applications as a set of programs that can virtually execute in parallel: namely, threads.
- If these programs run on a “multithreaded” CPU, they will best exploit its architectural features.
- What about their execution on a CPU that does not support multithreading?
- A multithreaded CPU is not a parallel architecture, strictly speaking; multithreading is obtained through a single CPU, but it allows a programmer to design and develop applications as a set of programs that can virtually execute in parallel: namely, threads.

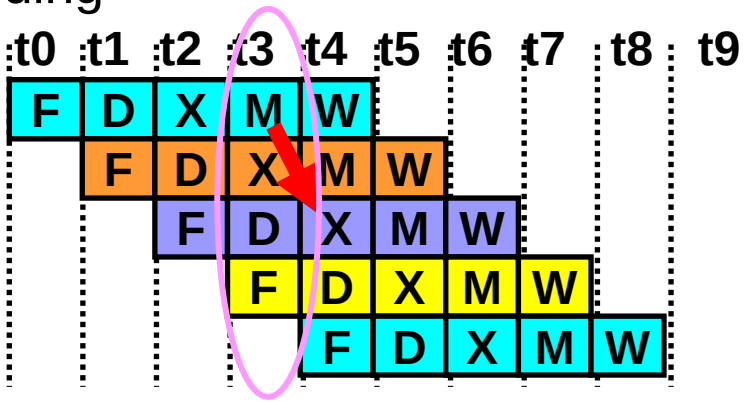
Multi-Threading

- Multithreading is solution to avoid waisting clock cycles as the missing data is fetched: making the CPU manage more peer-threads concurrently; if a thread gets blocked, the CPU can execute instructions of another thread, thus keeping functional units busy.
- So, why cannot be threads form different tasks be issued as well?
- To realize multithreading, the CPU must manage the computation state of each single thread.
- Each thread must have a privateProgram Counter and a set of private registers, separate from other threads.
- Furthermore, thread switch must be much more efficient than process switch, that requires usually hundreds or thousands of clock cycles (process switch is a software procedure, mostly)

Use Multithreading to Help ILP

- One idea: allow instructions from different threads to be mixed and executed together in the pipeline
  - Original: pipeline with internal forwarding

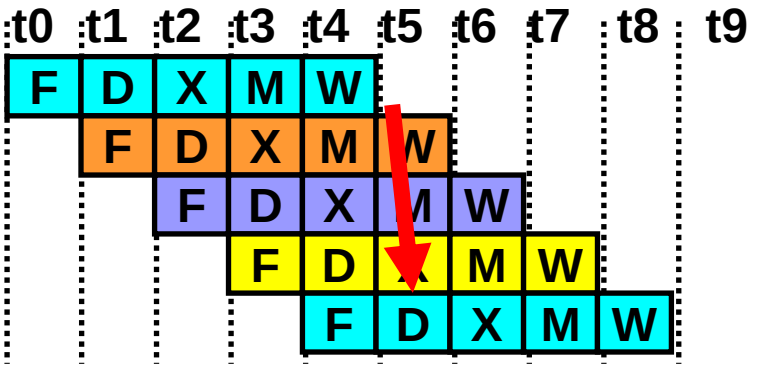
```
T1: LW   r1, 0(r2)
  <bubble>
T1: SUB  r5, r1, r4
T1: AND r4, r1, r3
T1: SW   0(r7),  r5
```



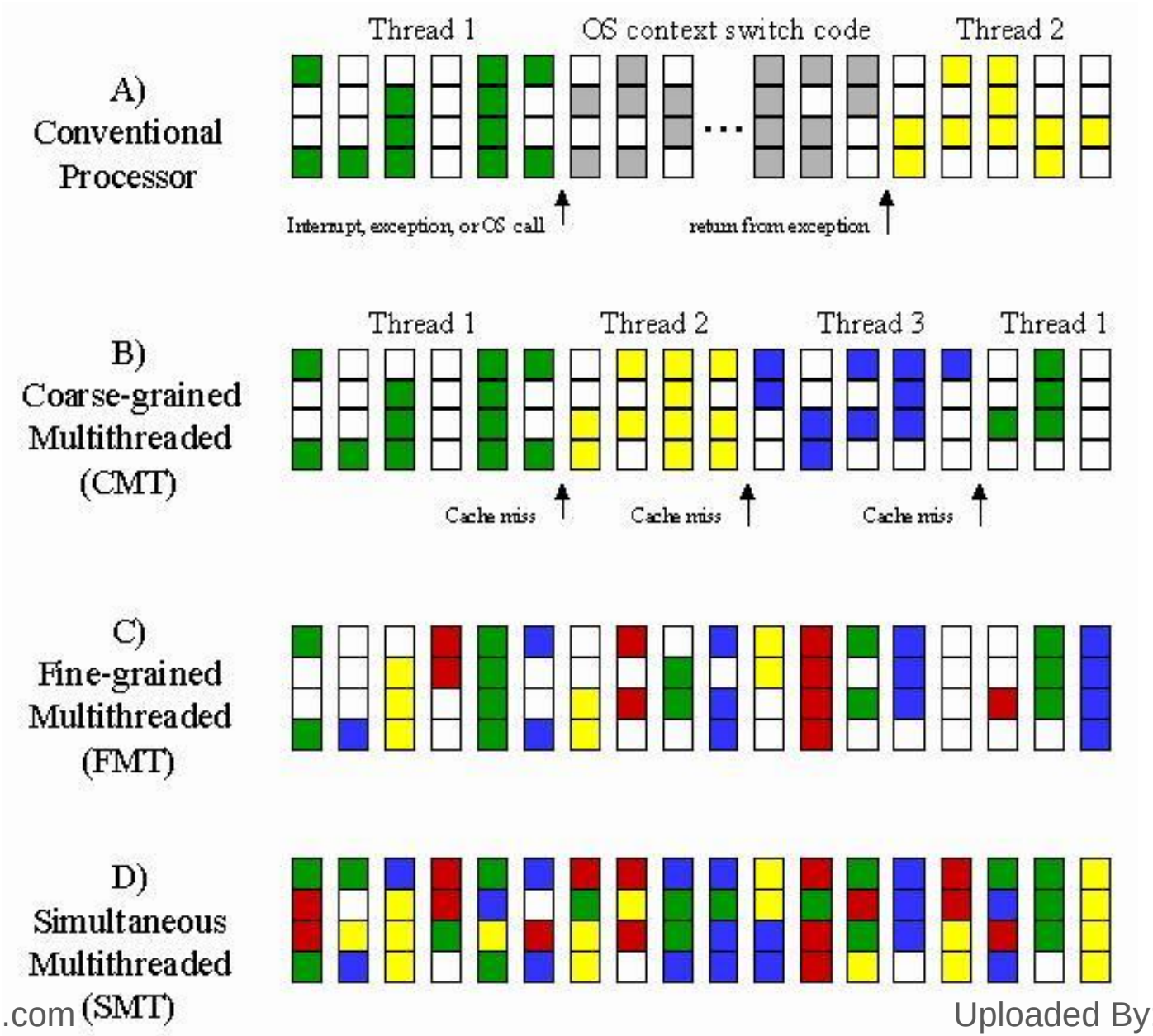
  - Multithreaded pipeline:

No need for internal forwarding

```
T1: LW   r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW   0(r7),  r5
T1: SUB  r5, r1, r4
```



Strategies for Multithreaded Execution

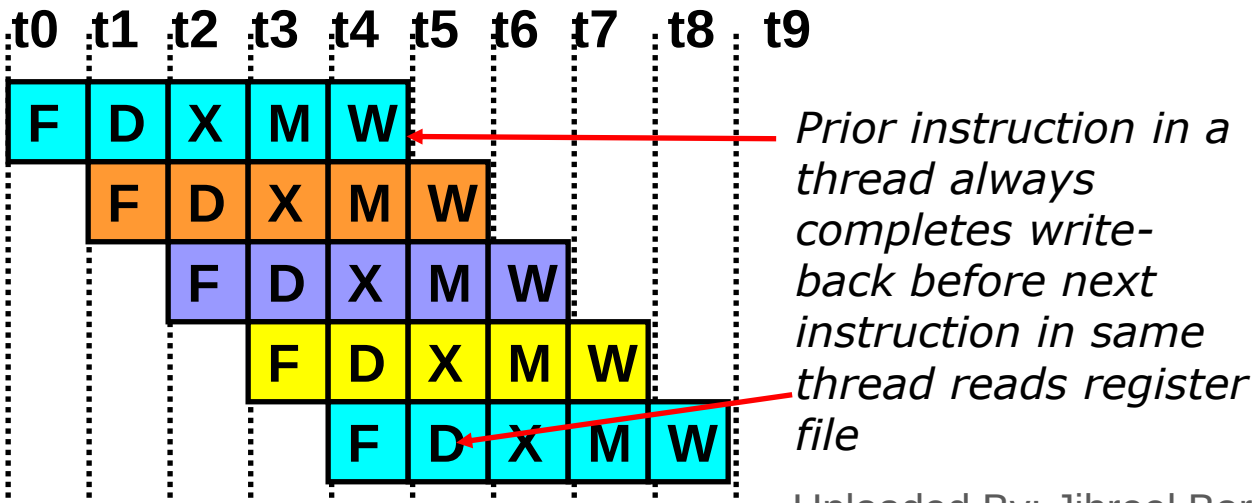


Fine-grained Multi-Threading

- **Fine-grained Multithreading**: switching among threads happens at each instruction, independently from the the fact that the thread instruction has caused a cache miss.
- Instructions "scheduling" among threads obeys a round robin policy, and the CPU must carry out the switch with no overhead, since overhead cannot be tolerated
- If there is a sufficient number of threads, it is likely that at least one is active (not stalled), and the CPU can be kept running.

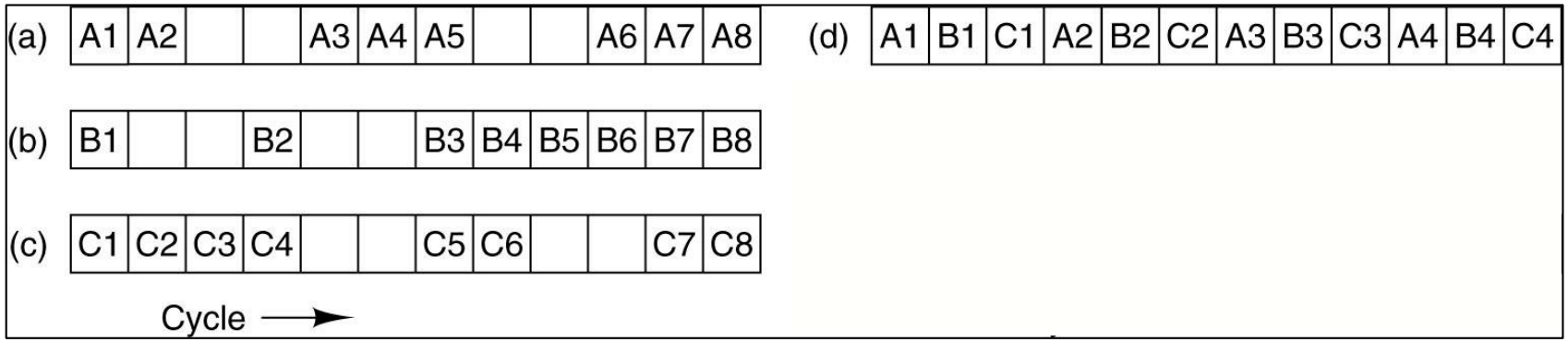
Interleave 4 threads, T1-T4, 5-stage pipe, no internal forwarding

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: SUB r5, r1, r4



Fine-grained Multi-Threading

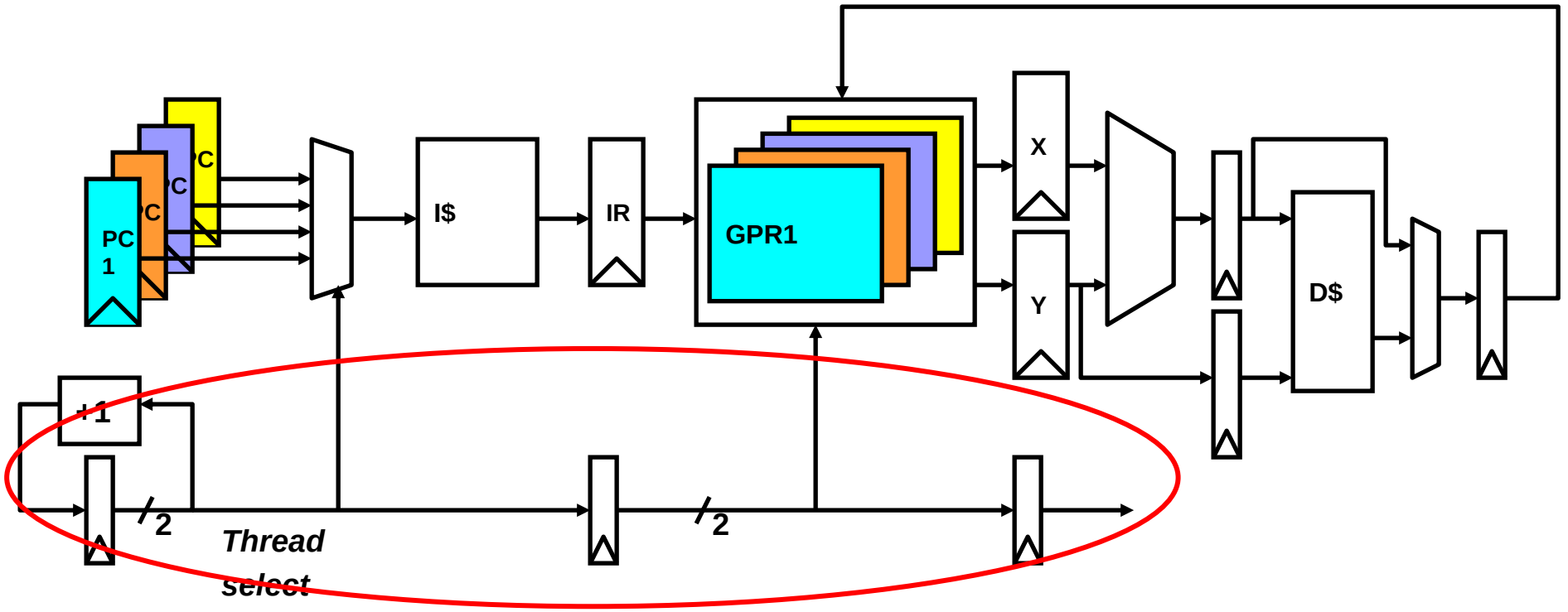
- (a)-(c) three threads and associated stalls (empty slots).
- (d) Fine-grained multithreading. Each slot is a clock cycle, and we assume for simplicity that each instruction can be completed in a clock cycle, unless a stall happens.



- In this example, 3 threads keep the CPU running, but what if A2 stall lasts 3 or more clock cycles?

Fine-Grained Multithreading Pipeline

- Carry thread-select down pipeline to ensure correct state bits read/written at each pipe stage
  - Appears to software (including OS) as multiple, albeit slower, CPUs



Multithreading Costs

- Each thread requires its own user state
  - PC
  - GPRs
- Also, needs its own system state
  - Virtual-memory page-table-base register
  - Exception-handling registers
- Other overheads:
  - Additional cache/TLB conflicts from competing threads
    - or add larger cache/TLB capacity
  - More OS overhead to schedule more threads (where do all these threads come from?)

Fine-grained Multi-Threading

- CPU stalls can be due to a cache miss, but also to a true data dependence, or to a branch: dynamic ILP techniques do not always guarantee that a pipeline stall is avoided.

- With fine-grained multithreading in a pipelined Architecture, if: the pipeline has *k* stages,
  - there are at least *k* threads to be executed,
  - and the CPU can execute a thread switch at each clock cycle

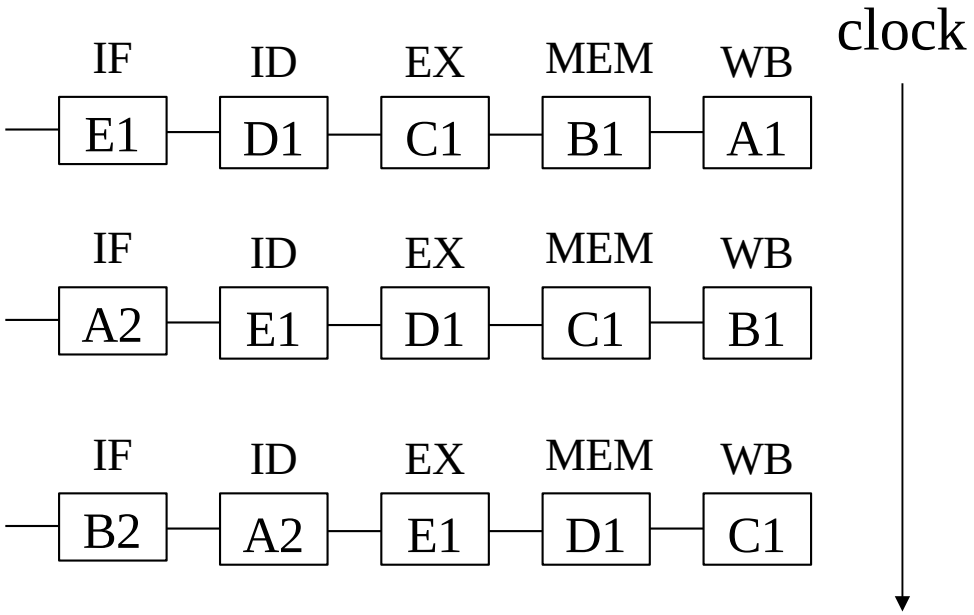
- Then there can never be more than a single instruction per thread in the pipeline at any instant, so there cannot be hazards due to dependencies, and the pipeline never stalls (… another assumption is required …).

Fine-grained Multi-Threading

- Fine-grained multithreading in a CPU with a 5-stage pipeline:
  - There are never two instructions of the same thread concurrently active in the pipeline.
  - If instructions can be executed out of order, then it is possible to keep the CPU fully busy even in case of a cache miss.

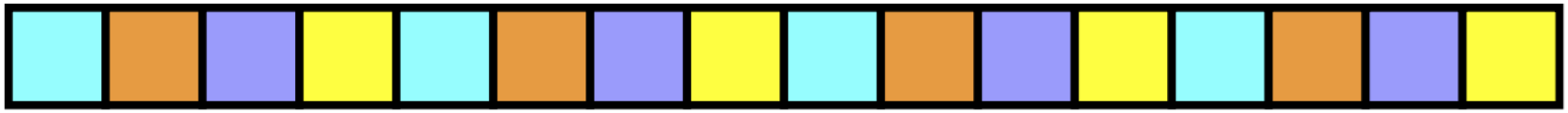
5 threads in execution:

A1	A2	A3	A4	A5	A6	...
B1	B2	B3	B4	B5	B6	...
C1	C2	C3	C4	C5	C6	...
D1	D2	D3	D4	D5	D6	...
E1	E2	E3	E4	E5	E6	...



Thread Scheduling Policies

- Fixed interleave
  - Each of N threads executes one instruction every N cycles
  - If thread not ready to go in its slot, insert pipeline bubble

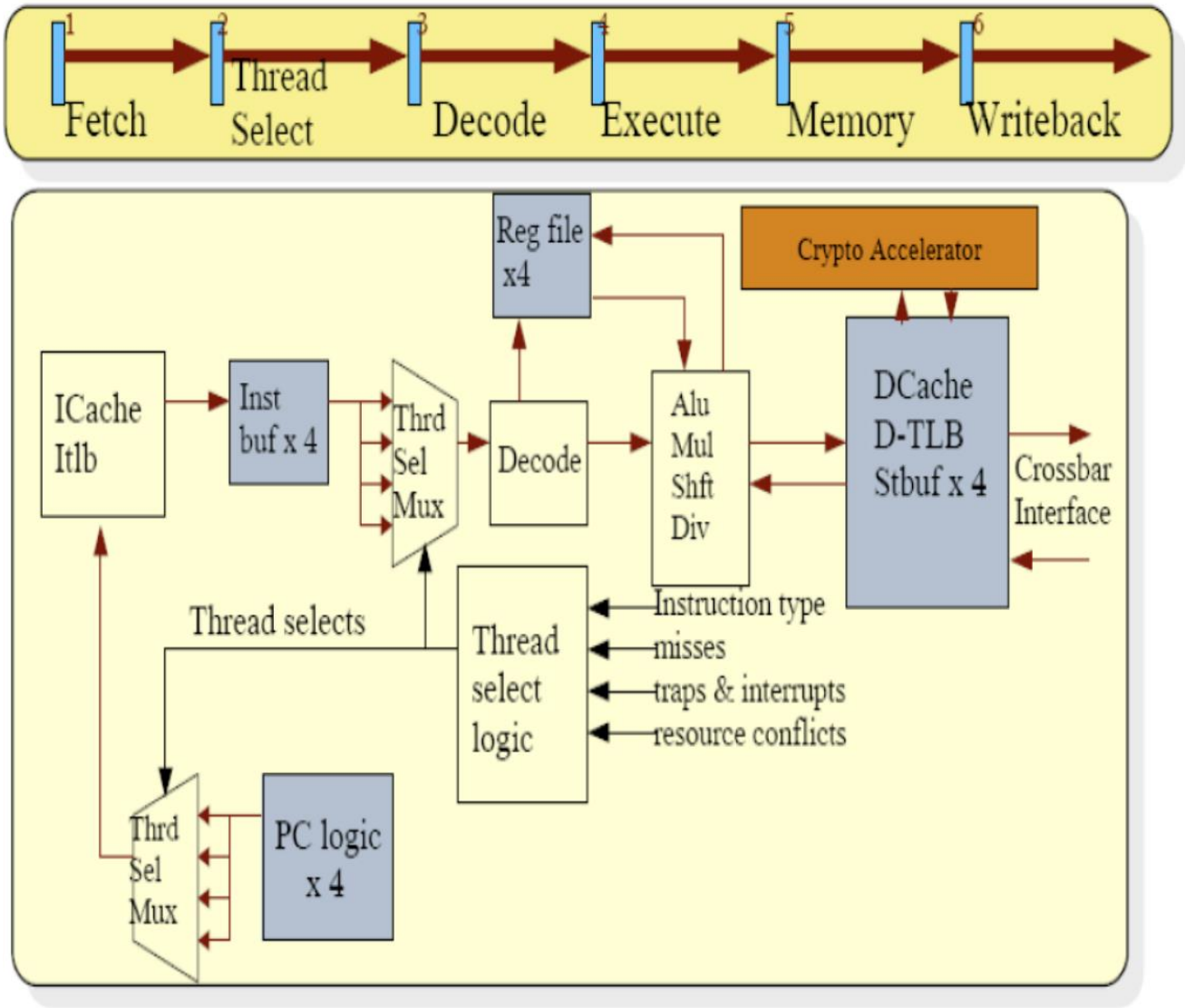


- Software-controlled interleave
  - OS allocates S pipeline slots for N threads
  - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- Hardware-controlled thread scheduling
  - Hardware keeps track of which threads are ready to go
  - Picks next thread to execute based on hardware priority scheme

Sun Niagara Multithreaded Pipeline

- Each SPARC core has hardware support for four threads.
- The four threads share the instruction, the data caches, and the TLBs.
- Each SPARC core has simple, in-order, single issue, six stage pipeline.



Fine-Grained Multithreading

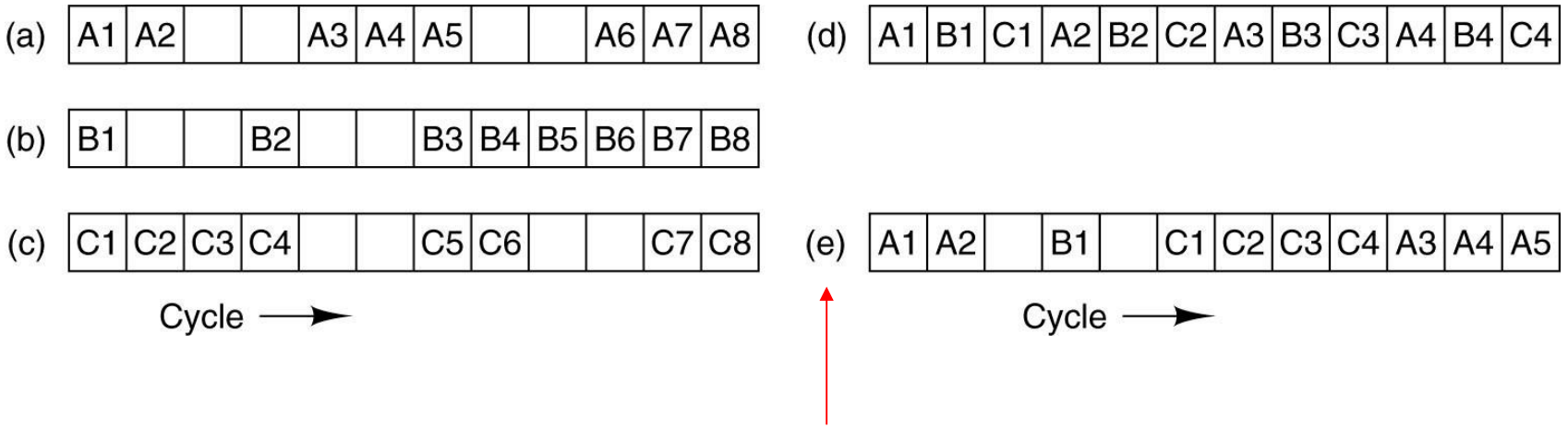
- **Advantage:**
  - Hide both short and long stalls, e.g., latency of memory operations, dependent instructions, branch resolution, etc., since instructions from other threads executed when one thread stalls → *latency hiding*
  - No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
  - No need for branch prediction logic
  - Otherwise-bubble cycles used for executing useful instructions from different threads
  - Improved system throughput, latency tolerance, utilization
- **Disadvantage:**
  - Extra hardware complexity: multiple hardware contexts, thread selection logic
  - Slow down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
  - There might be fewer threads than stages in the pipeline (actually, this is the usual case), so keeping the CPU busy is no easy matter.
  - Requiring an efficient context switch among threads
  - Resource contention between threads in caches and memory
  - Dependency checking logic between threads remains (load/store)

Coarse-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses or when an explicit context switch instruction is encountered
  - Instructions from one thread use the pipeline for a certain period of time
- At this point, a switch is made to another thread. When this thread in turn causes a stall, a third thread is scheduled (or possibly the first one is re-scheduled) and so on.
- This approach potentially wastes more clock cycles than the fine- grained one, because the switch happens only when a stall happens.
- but if there are few active threads (even just two), they can be enough to keep the CPU busy.
- Possible stall events
  - Cache misses
  - Synchronization events (e.g., load an empty location)
  - FP operations

Coarse vs Fine-grained Multi-Threading

- (a)-(c) three threads with associated stalls (empty slots).
- Fine-grained multithreading.
- Coarse-grained multi-threading



Coarse-Grained Multithreading

- Advantages:
  - Relieve the need to have very fast thread-switching
  - Do not slow down thread, since context-switch only when the thread encounters a costly stall
  - Priority may be given to critical thread

- Disadvantages:
  - Instructions must be drained and refilled on a context switch → bad for long pipeline, good only for costly stalls
  - Fairness: a low cache miss thread gets to use pipeline longer and other threads may starve
    - Possible solution: low miss thread may be preempted after a time slice expires, forcing a thread switch

Fine Grained vs Coarse Grained

FINE GRAINED MULTITHREADING	COARSE GRAINED MULTITHREADING
A multithreading mechanism in which switching among threads happens despite the cache miss caused by the thread instruction	A multithreading mechanism in which the switch only happens when the thread in execution causes a stall, thus wasting a clock cycle
More efficient	Less efficient
Requires more threads to keep the CPU busy	Requires fewer threads to keep the CPU busy

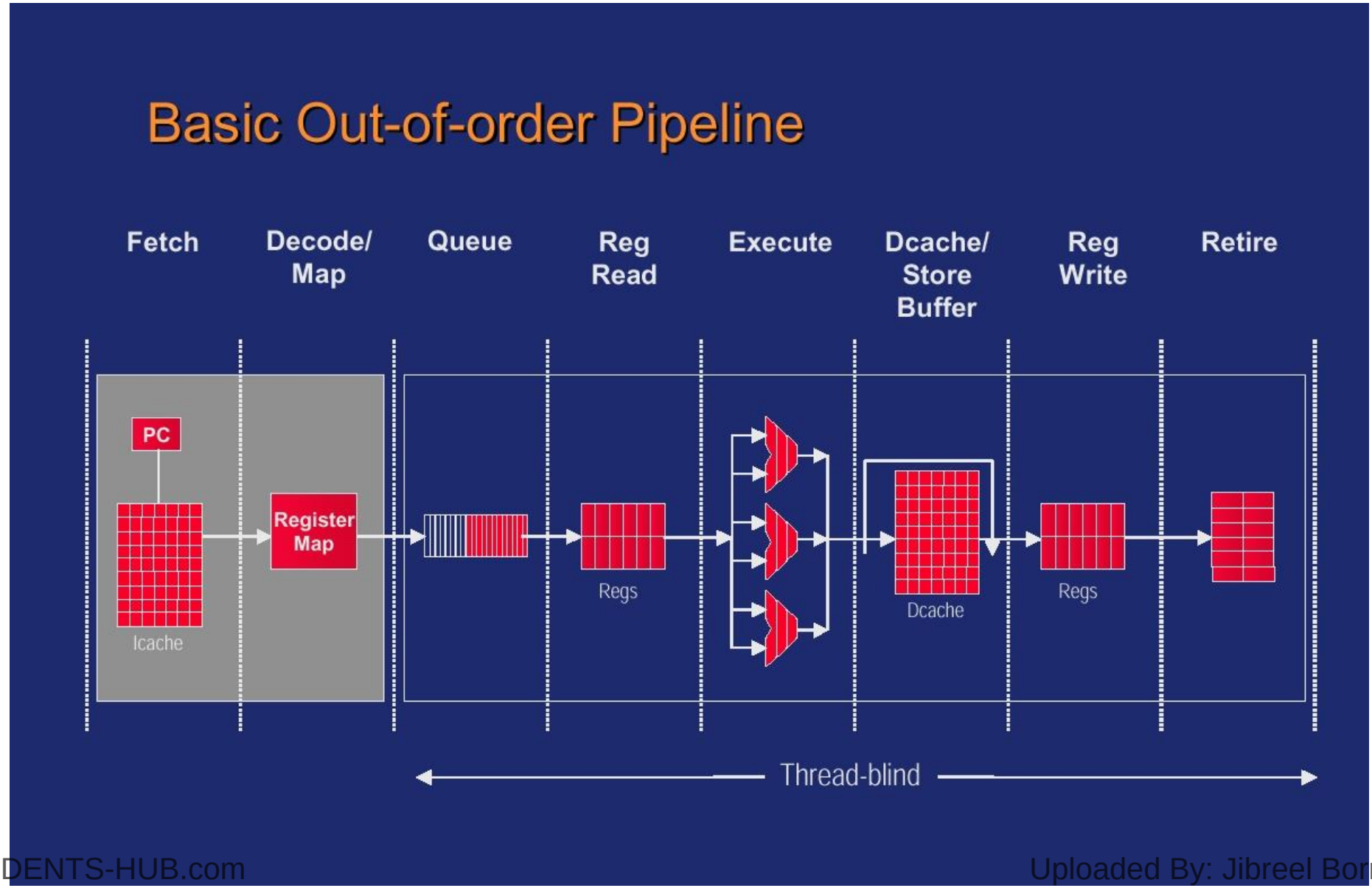
Simultaneous Multithreading (SMT)

- Modern superscalar, multiple issue and dynamic scheduling pipeline architectures allow to exploit both ILP (instruction level) and TLP (thread level) parallelism.
  - **ILP + TLP = Simultaneous Multi-Threading (SMT)**
- SMT is convenient since modern multiple-issue CPUs have a number of functional units that cannot be kept busy with instructions from a single thread.
- By applying register renaming and dynamic scheduling, instructions belonging to different threads can be executed concurrently.
- In SMT, multiple instructions are issued at each clock cycle, possibly belonging to different threads; this increases the utilization of the various CPU resources

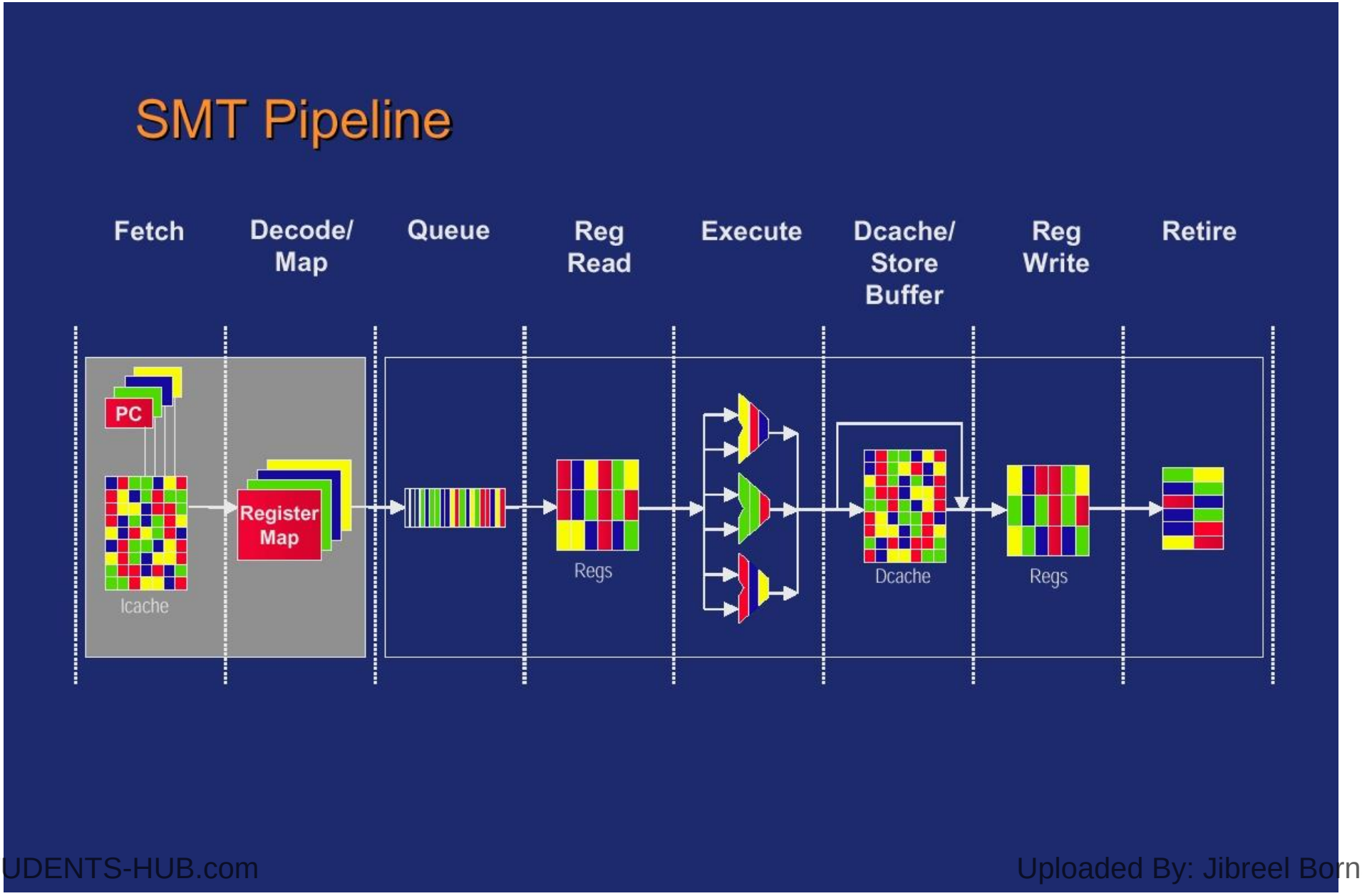
Simultaneous Multithreading (SMT)

- In superscalar CPUs with no multithreading, multiple issue can be useless if there is not enough ILP in each thread, and if a long lasting stall (a L3 cache miss) freezes the whole processor.
- In coarse-grained MT, long-lasting stalls are hidden by thread switching, but a poor ILP level in each thread limits CPU resource exploitation (e.g., not all issue slots available can effectively be used)
- Even in fine-grained MT, a poor ILP level in each thread limits CPU resource exploitation.
- SMT: instructions belonging to different threads are (almost certainly) independent, and by issuing them concurrently, CPU resources utilization raises.

SMT Pipeline Architecture



SMT Pipeline Architecture



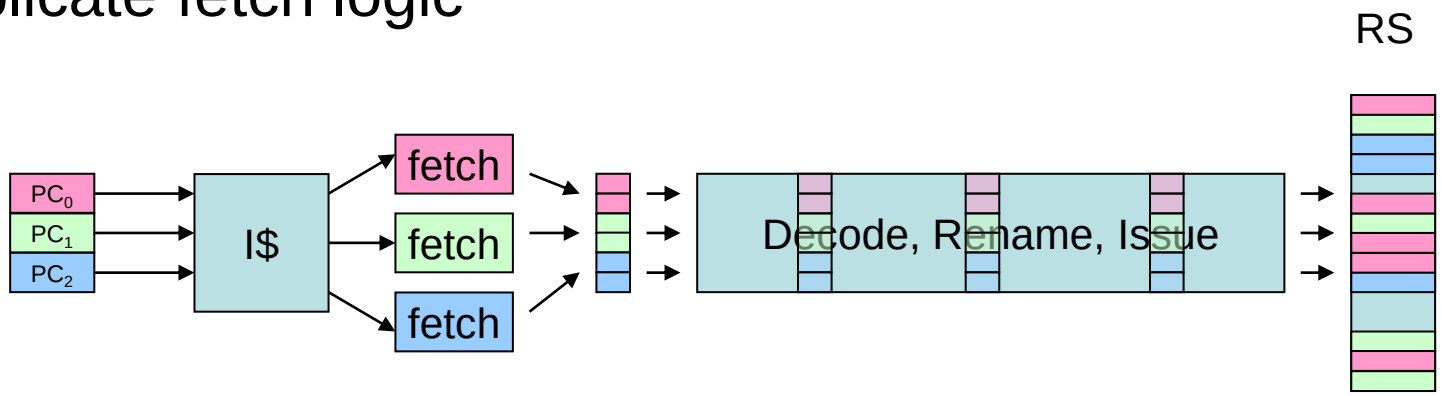
Resources in Typical SMT

- Per thread:
  - State for hardware context (separate PC, arch register file, rename mapping table, reorder buffer, L/S queues, etc.)
  - Instruction commit/retirement, exception, subroutine return stack
  - Per thread id in TLB
  - BTB may be shared or have separate thread id (optional)
  - Ability to fetch instructions for multiple threads (I cache port)

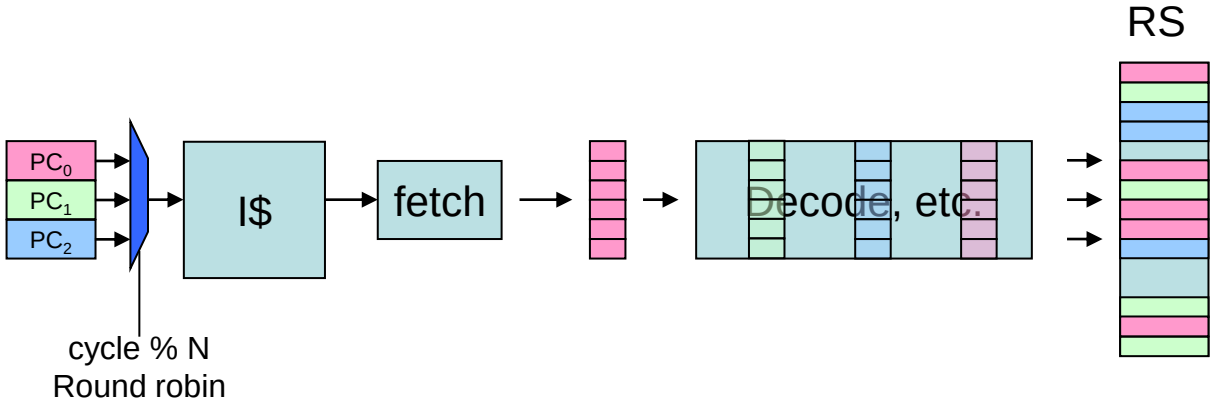
- Shared
  - Physical register, cache hierarchy, TLB (with TID), branch predictor and branch target buffer, functional units

SMT Fetch

- Duplicate fetch logic

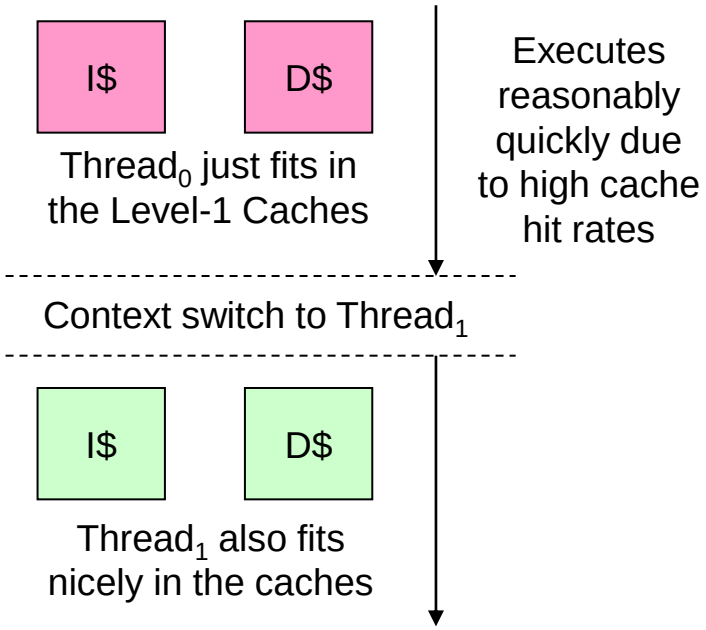


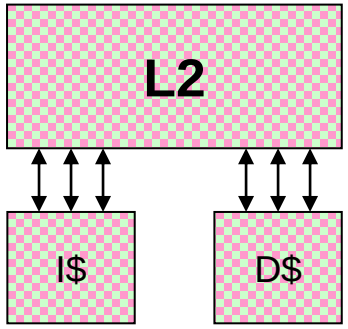
- Cycle-multiplexed fetch logic



Effects of SMT on Cache

- Cache thrashing





Caches were just big enough to hold one thread's data, but not two thread's worth

Now both threads have significantly higher cache miss rates

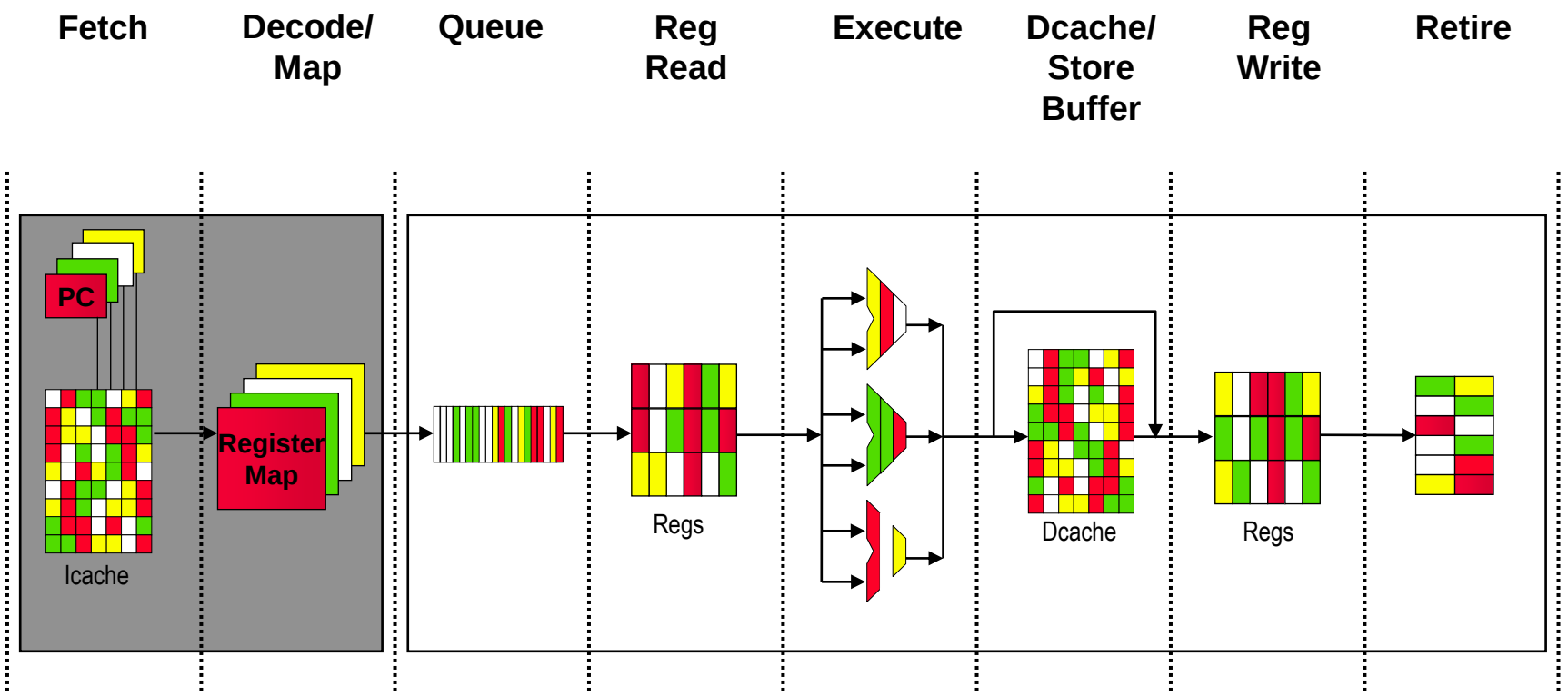
Observations on SMT

- Higher throughput
  - May be useful for server
- May incur longer latency for a single thread
  - Instruction fetch complexity
  - Increase associativity of TLB and L1 cache
  - Larger L2 cache, etc., to handle multiple threads
  - May increase cache misses/conflicts
  - Complexity in branch prediction and committing multiple threads simultaneously
  - More registers (per thread RF, rename mapping table)
  - Stretch hardware design and may affect cycle time

Multithreaded Processors Comparison

MT Approach	Resources shared between threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file and con trol logic/state	Switch on pipeline stall
SMT	Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.	All contexts concurrently active; no switching

Early Design: Alpha 21464 4-way SMT



SMT with 4 threads. Each thread appears to the outside world as executed sequentially

Intel Multi-Threading

- Multithreading was first introduced by Intel in Xeon processor in 2002, later in the 3,06 GHz Pentium 4, with code name hyperthreading. The name is attractive, actually hyperthreading supports only two threads in SMT mode.
- According to Intel, designers had speculated that multithreading was the simplest way to increase performance: an increase by 5% of CPU area would allow to run a second thread, thus effectively using CPU resources otherwise wasted.
- Intel benchmark suggested an increase of CPU performance by 25% -- 30%.
- To the Operating system, a multithreaded processor is indeed a double processor, with two CPUs sharing caches and RAM: if two applications can run independently and share the same address space, they can be executed in parallel in two threads.
- A movie editing code can use different filters to be applied in each frame. The code can be structured as two threads, that process odd/even frames, and that execute in parallel.

Intel Multi-Threading

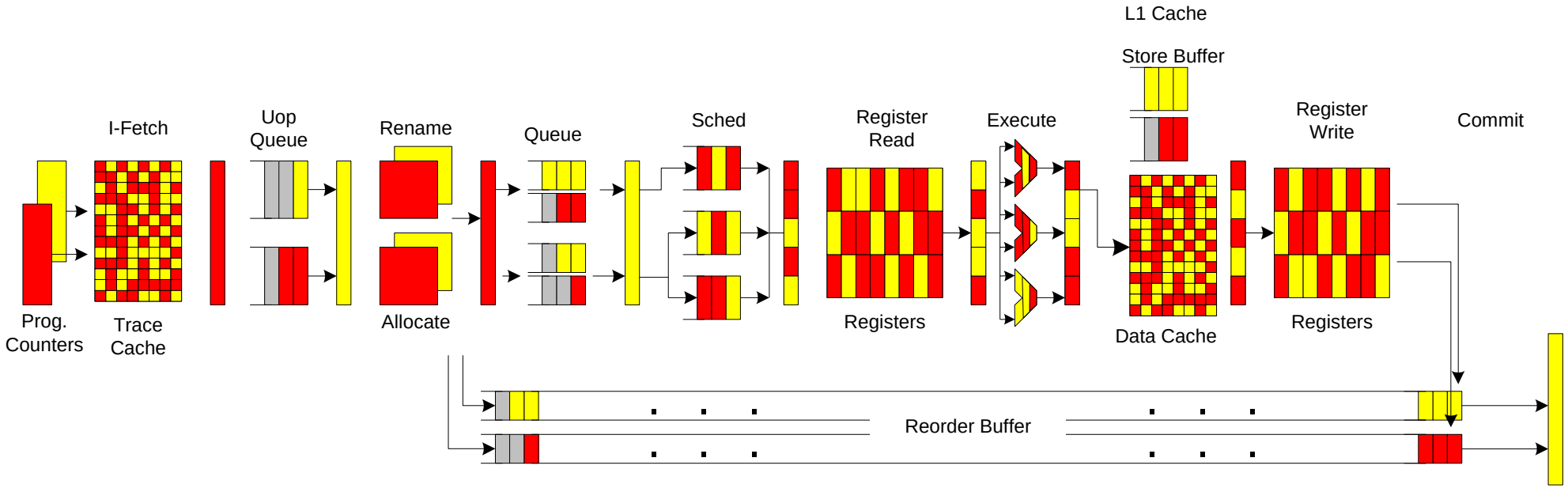
- Since two threads can use the CPU concurrently, it is necessary to design a strategy that allows both threads to effectively use CPU resources.
- Intel uses 4 different strategies to share resources between the two threads.
- **Replication.** Obviously, some resources have to be replicated, in order to manage the two threads: two program counters and registers mapping tables (ISA registers vs rename registers) so that each thread has an independent set of registers. This replications accounts for the 5% increase in processor area.
- **Partitioning.** Some hardware resources are rigidly partitioned between the two threads. Each thread can use exactly half of each resource. This applies to all buffers (for LOAD, STORE instructions) and to the ROB ("retirement queue" in Intel terminology).
  - Partitioning can of course reduce the utilization of the partitioned resources, when a thread does not use its part of the resource, which could be used by another thread.

Intel Multi-Threading

- **Sharing.** The hardware resource is completely shared. The first thread that gets hold of the resources uses it, and the other thread waits.
  - This type of resource management solves the problem due to an unused resource (if the thread does not need it), since it can be allocated to the second one. Obviously, the reverse problem arises: a thread can be slowed down if the required resource is completely allocated to the other one.
  - For this reason, in Intel processor the only resources completely shared are those available in a great quantity: for them, it is unlikely that a "starvation" problems arises, e.g. cache lines.
- **Threshold sharing.** A thread can use dynamically the resource, up to a given percentage; so, a part remains available for the other task (possibly less than half).
  - The scheduler that dispatches uops to the reservation stations uses this policy.

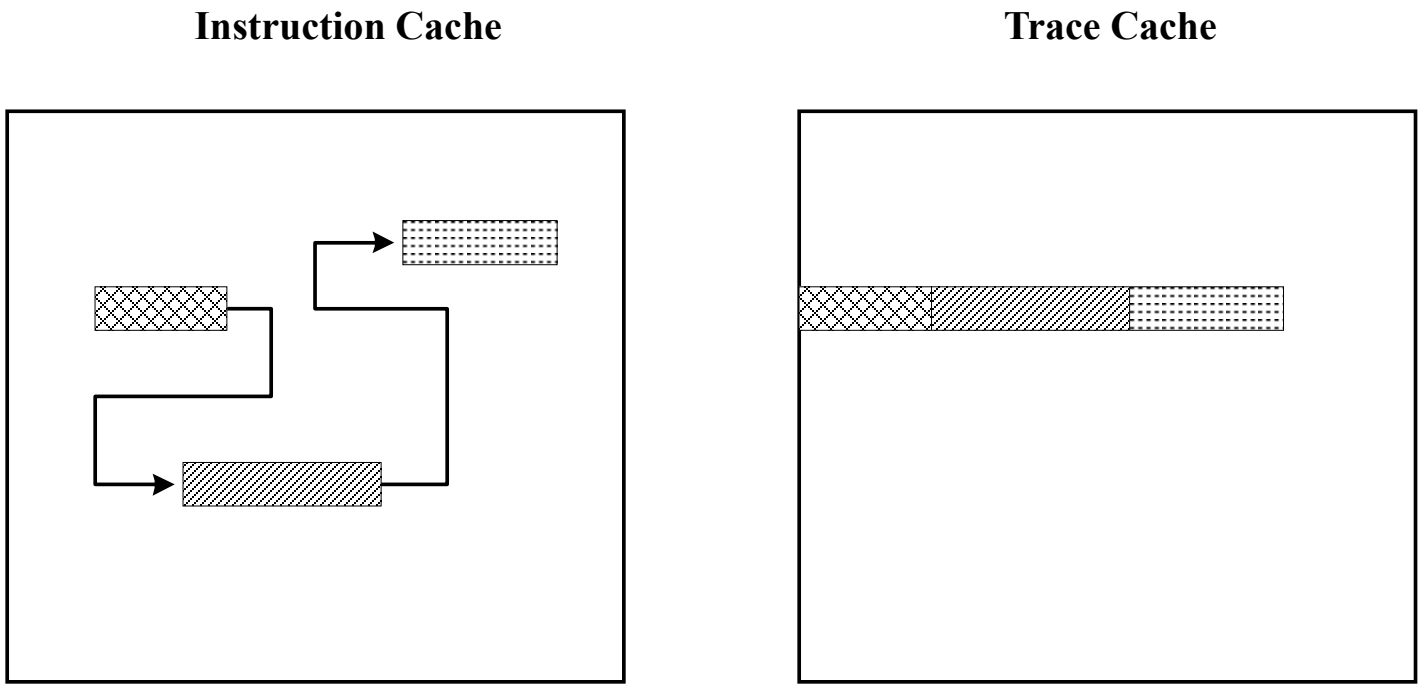
Intel Multi-Threading Pipeline Datapath

- Main pipeline
  - Pipeline prior to trace cache not shown
- Round-Robin instruction fetching
  - Alternates between threads
  - Avoids dual-ported trace cache
  - BUT trace cache is a shared resource



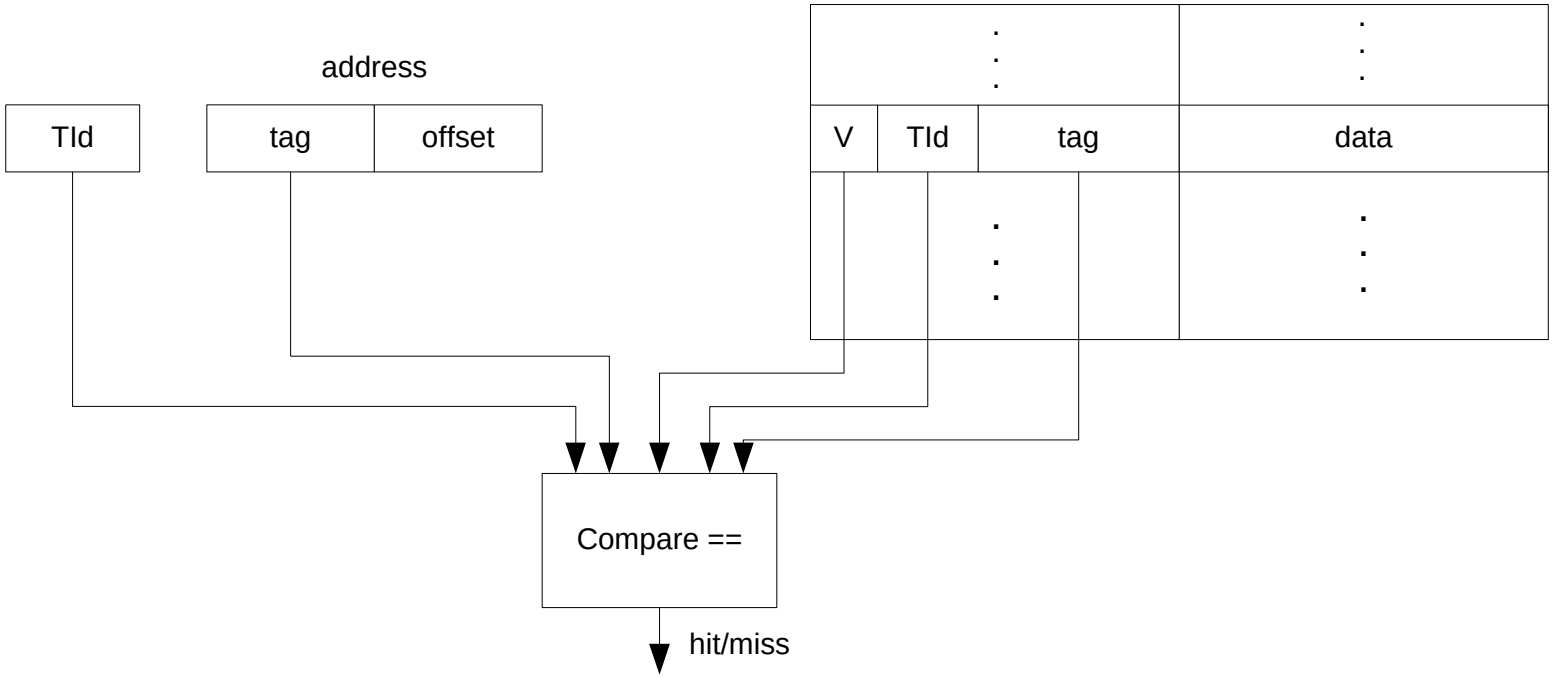
Trace Caches

- Trace cache captures dynamic traces
- Increases fetch bandwidth
- Help shorten pipeline (if predecoded)



Capacity Resource Sharing

- Append thread identifier (TId) to threads in shared capacity (storage) resource
- Example: cache memory



Effectiveness of Simultaneous Multithreading on Superscalar Processors

Speedup from using multithreading on one core on an i7 processor

The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks

An increase by 5% of CPU area yields a performance increase by an average of 30%