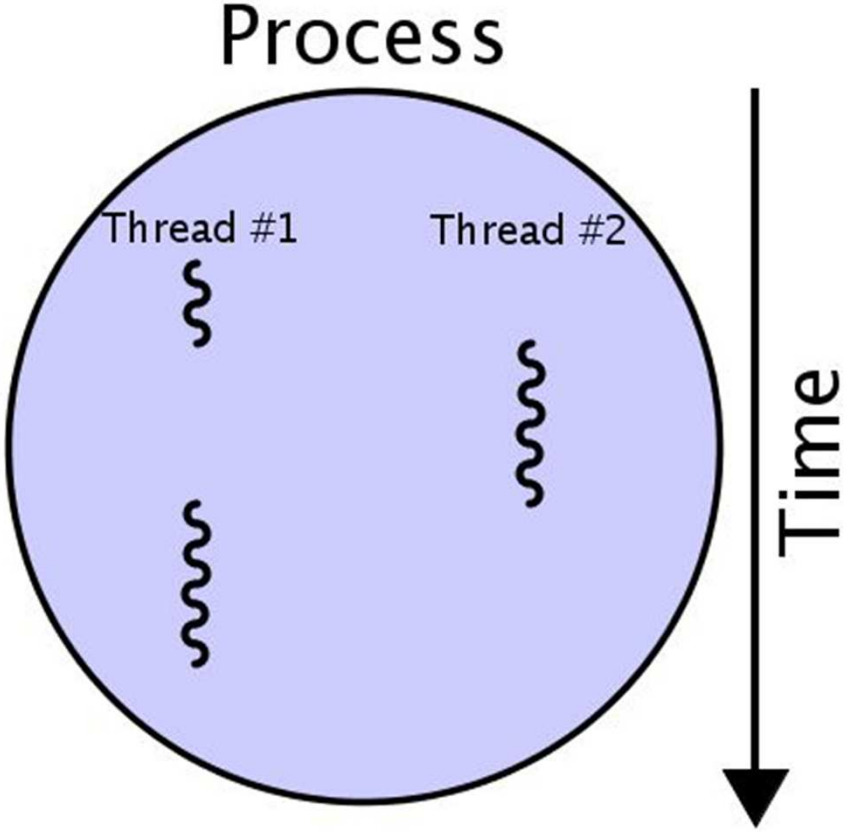
Synchronization

Terminology

- Threads
- Processes (or tasks)
  - Independent
  - Considerable state info
  - Separate address space
- Resources
  - Memory, file handles, sockets, device handles
  - Owned by processes



Threads and processes

- Thread of a same process share the same resources
- Processes can share resources only through explicit methods
- The only resources exclusively owned by a thread are the thread stack, register status and thread-local storage (if any)

Scheduling

- A scheduler is responsible to allocate tasks and threads to the available computing resources
- Various kind of scheduling algorithms
  - Best-effort → minimize makespan
  - Real-Time → meet deadlines
  - … other metrics to minimize
- Preemption and context changes
- Access to shared resources

Multithreading

- Classic multithreading on single processor systems required Time Division Multiplexing (TDM)
  - Time driven
  - Event driven
- Multiprocessors → different threads and processes can run on different CPUs
- Multithreading is easier (“native”) on multicore platforms
- But scheduling requires more attention

Multithreading issues

- Race conditions
- Starvation, priority inversion, deadlock, livelock
  - Mamihlapinatapai
- Synchronization (mutex, lock)
- Atomic execution (semaphores)
- Communication:
  - shared-memory (requires locking)
  - message-passing (slower but easier)

Different kinds of Parallelisms

- Instruction Level Parallelism (ILP)
- Data Level Parallelism (DLP)
- Thread Level Parallelism (TLP)

Instruction Level Parallelism (ILP)

- Execute multiple instruction per clock cycle
- Each functional unit on a core is an execution resource within a single CPU:
  - Arithmetic Logic Unit (ALU)
  - Floating Point Unit (FPU)
  - bit shifter, multiplier, etc.
- Need to solve data dependencies

Data dependency

- Consider the sequential code:
  1. e = a + b
  2. f = c + d
  3. g = e * f
- Operation 3. depends on the results of operations 1. and 2.
- Cannot execute 3. before 1. and 2. are completed

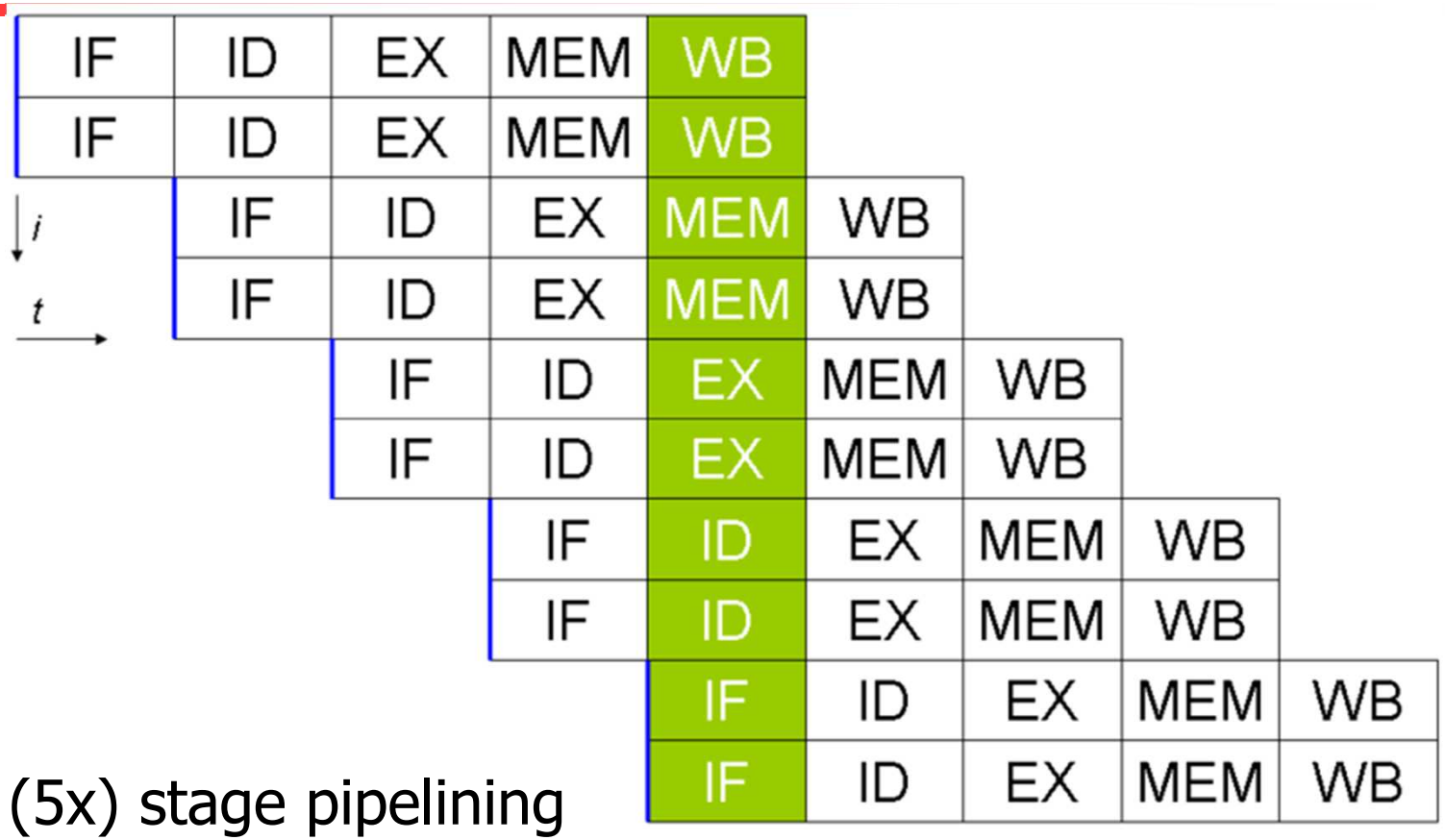
How to “parallelize” software?

- Parallelism can be extracted from ordinary programs
  - At run-time (by complex specific HW)
  - At compile-time (simplifying CPU design and improving run-time performances)
- Degree of ILP is application dependent

ILP: superscalar architectures

- Data dependency check in HW
- Complex mechanisms
  - power, die-space and time consuming
- Problematic when
  - code difficult to predict
  - Intructions have many interdependencies

Superscalar pipelining



(5x) stage pipelining

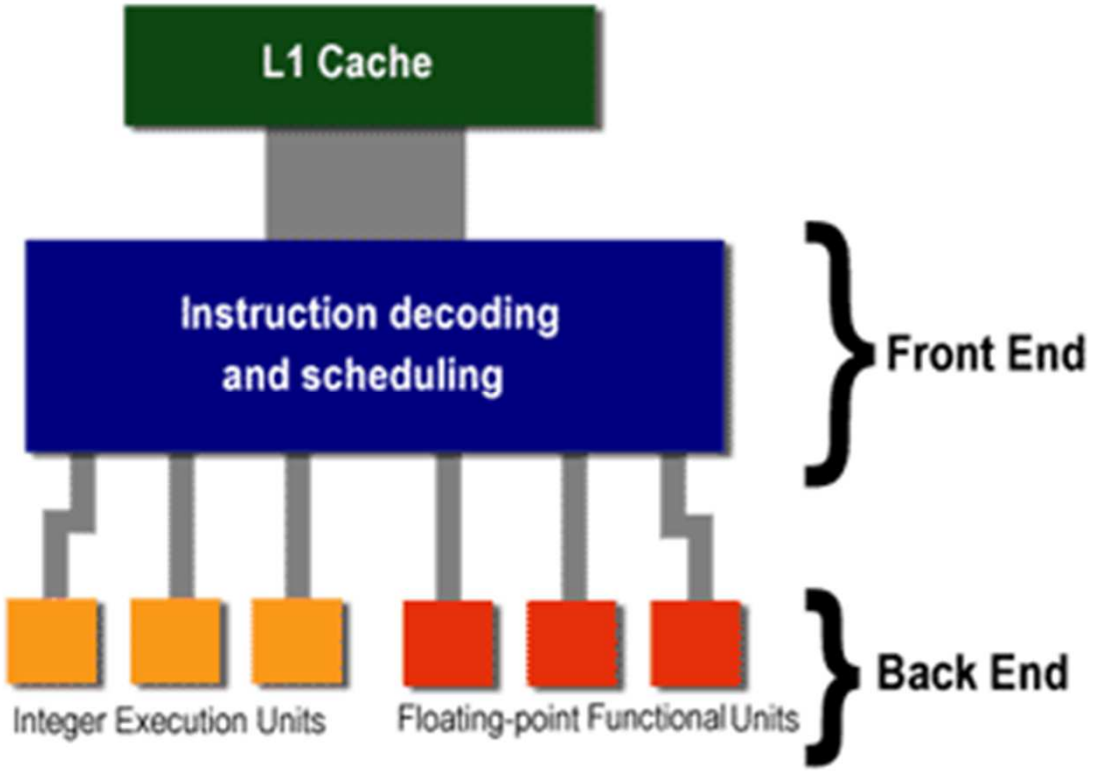
(2x) Superscalar execution

ILP optimizations

- Instruction pipelining
- Superscalar execution
- Out-of-order execution
  - deferred memory accesses
  - combined load and store
- Speculative execution
  - branch prediction,
  - speculative load
  - …

Processor front and back end

- Intel describes its processors having
  - “in-order front end”
  - “out-of-order execution engine”



ILP: compile-time techniques

- Compiler decides which operations can run in parallel
- Removes the complexity of instruction scheduling from HW to SW
- New instruction sets that explicitly encode multiple independent operations per instruction
  - Very Long Instruction Word (VLIW): one instruction encodes multiple operations (one for each execution unit)
  - Explicitly Parallel Instruction Computing (EPIC): adds features to VLIW (cache prefetching instructions, ...)

Data Level Parallelism (DLP)

- Higher parallelism than superscalar architecture
- SIMD instructions (Single Instruction, Multiple Data)
  - Intel's MMX, SSE, SSE2, SSE3, SSE3, SSSE3, SSE4, AVX
  - AMD's 3DNow!, SSE5
  - ARM's NEON, IBM's AltiVec and SPE, etc.
  - Graphic cards (GPU)
  - Cell Processor's SPU
- Useful when the same operation has to be applied to a large set of data (i.e., multimedia, graphic operations on pixels, etc.)
- Multiple data are read and/or modified at the same same

Thread Level Parallelism (TLP)

- Higher level parallelism than ILP
- Different kinds of TLP
  - Superthreading
  - Hyperthreading or Symultaneous MultiThreading (SMT)
    - Needs superscalar processor
  - Chip-level MultiProcessing (CMP)
    - Needs multicore architecture
  - Combinations of the above solutions

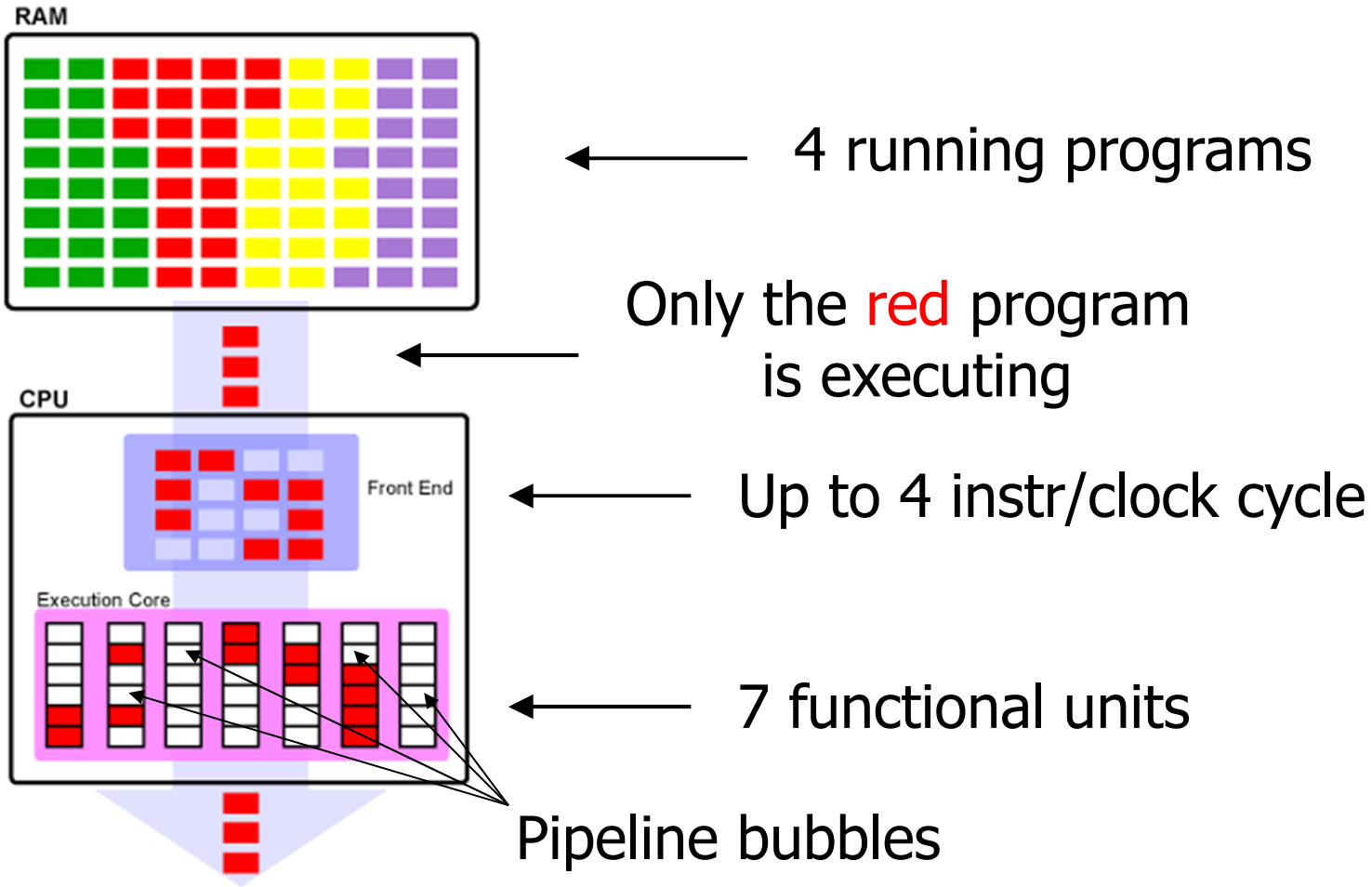
Superthreading

- Temporal Multithreading (fine- or coarse-grained) → when processor idle, execute instruction of another thread
- Makes better use of the computing resources when a thread is blocked
- Requires adequate hardware support to minimize context change overhead

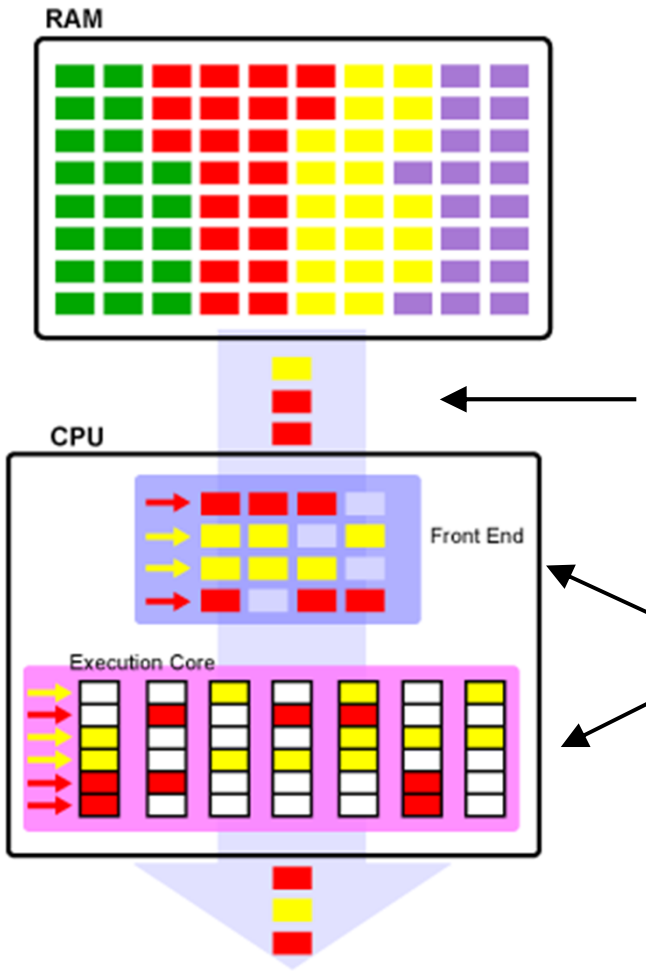
Hyperthreading

- Simultaneous MultiThreading (SMT)
- Introduced in late 90s: Intel's Pentium 4
- Execute instructions from multiple threads simultaneously → needs superscalar support
- Energy inefficient
  - Increases cache thrashing by 42%, whereas dual core results in a 37% decrease

Single-threaded CPU



Super-threading (time-slice multithreading)

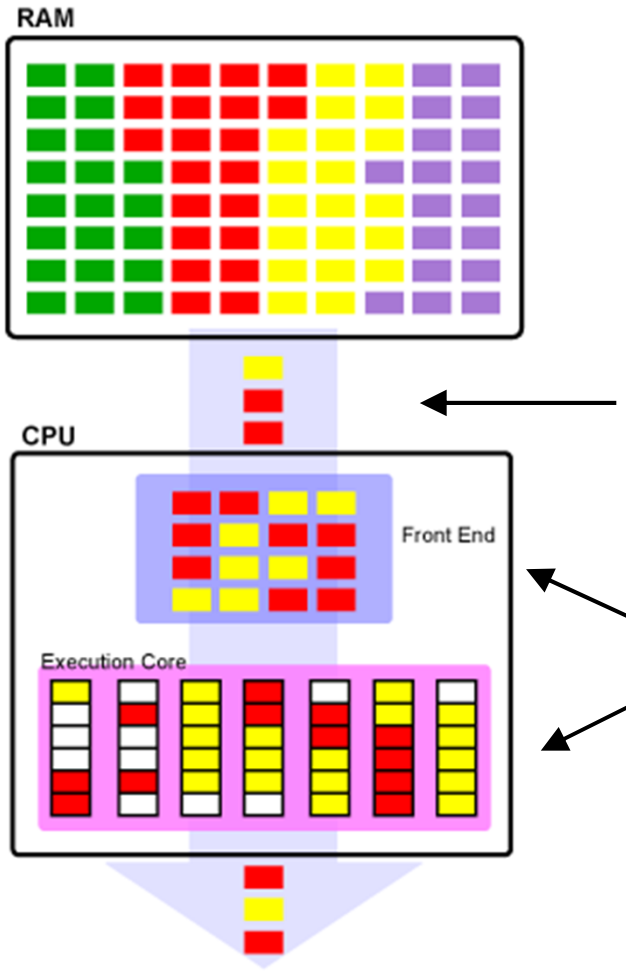


Multithreaded processor: able to handle more than one thread at a time

Interleaved execution of different threads (helps masking memory latency)

All instructions in a pipeline stage must come from the **same** thread

Hyper-threading (Simultaneous MultiThreading)



Interleaved execution of different threads (helps masking memory latency)

Instructions in a pipeline stage may come from **different** threads

Hyper-threading (SMT)

- From OS perspective: many “logical” processors
- Average ILP for a thread = 2.5 instr/cycle
  - Pentium 4 issues at most 3 instr/cycle to the execution core
- Hyperthreaded processor can exploit parallelism beyond a single thread ILP

But..

- SMT can be worse than non-SMT approaches
  - A thread monopolizing an execution unit for many consecutive pipeline stages can stall the available functional units (that could have been used by other threads)
- Cache thrashing problem
  - Different logical processor can execute two threads accessing completely different memory areas

Not a problem for Multicores

- A smart SMT-aware OS running on a multicore would schedule two different tasks on different processors → resource contention is minimized

Summary

- Out-of-order execution problems on multicore platforms
- Data dependencies
- Race conditions
- Memory barriers
- Locking mechanisms
- Spinlocks, semaphores, mutexes, RCU

Introduction: execution ordering

- A processor can execute instructions in any order (or in parallel), provided it maintains **program causality** with respect to itself
- The compiler may reorder instructions in many ways, provided causality maintainance
- Some CPUs are more constrained than others
  - E.g., i386, x86_64, UltraSPARC are more constrained than PowerPC, Alpha, etc.
  - Linux assumes the DEC Alpha execution ordering → the most relaxed one

Out-of-order execution

- Loads are more likely to need to be completed immediately
- Stores are often deferred
- Loads may be done speculatively, leading to discarded results
- Order of memory accesses may be rearranged to better use buses and caches
- Multiple loads and stores can be parallelized

Optimizations

- CPU or Compiler optimizations: overlapping instructions may be replaced
  - E.g., two consecutive loads to the same value/register

```
1) A = V;
2) A = W;
```

→

```
A = W;
```

  - A load operation may be performed entirely inside the CPU

```
1) *P = A;
2) B = *P;
```

→

```
1) *P = A;
2) B = A;
```

Cache coherency is not enough

- While the caches are expected to be coherent, there's no guarantee that that coherency will be ordered
- Whilst changes made on one CPU will eventually become visible on all CPUs, there's no guarantee that they will become apparent in the same order on those other CPUs

Causality maintanance

- If an instruction in the stream depends on an earlier instruction, then that earlier instruction must be "*sufficiently complete*" before the later instruction may proceed
- Need to analyse the *dependencies* between operations

Dependencies

- A “dependency” is a situation in which an instruction refers to the data of a preceding instruction
- Data dependencies
  - Read after Write
  - Write after Read
  - Write after Write
- Control dependencies

Read after Write

- True (or flow) dependence:
  - an instruction depends on the result of a previous instruction
- Example:

```
1) A = 3;
2) B = A + 1;
```

A is modified by the first instruction and used in the second one

It is not possible to use instruction level parallelism

Write after Read

- Antidependence:
  - an instruction requires a value that is updated by a later instruction
- Example:

```
1) B = A + 1;
2) A = 3;
```

A is modified by the second instruction and used by the first one

It is not possible to use instruction level parallelism

Write after Write

- Output dependence:
  - two instructions modify the same resource
- Example:

```
1)  A = 4;
2)  A = 5;
```

A is modified by both instructions

It is not possible to use instruction level parallelism

Other dependencies

- Read after Read a.k.a Input dependency
  - two instructions read the same resource

```
1)  B = A + 1;
2)  C = A + 2;
```

A is read by both instructions

ILP is possible!

- Control dependency
  - instruction execution depends on a previous instruction → e.g., conditional statements

```
if (x == true)
    A = 5;
```

"A = 5" executed only if x is true in the previous instruction

Considerations on dependencies

- No need to consider dependencies when a strict sequential execution model is used
- Otherwise, dependence analysis is needed for
  - CPU exploiting instruction level parallelism
  - out-of-order execution
  - parallelizing compiler
- Need to enforce execution-order constraints to limit the ILP of a system
- Usually automatically solved by compiler/HW
- Concerns the behavior of a single thread

More than dependencies...

- When more threads are concurrently executed → additional “dependency” problems
- Instructions from different tasks/threads may need to access the same global resources
- Problems similar to dependency problems, but more difficult to detect/solve
- Explicit programmer intervention is needed
- Need to synchronize the accesses to shared resources

Synchronization mechanisms

- Process synchronization
  - Barrier
  - Lock/semaphore
  - Critical sections
  - Thread join
  - Mutex
  - Non-blocking synchronization
- Data synchronization
  - Keep multiple copies of a set of data coherent with one another
  - E.g., cache coherency, cluster file systems, RAID, etc.

Process synchronization problems

- Race conditions
- Deadlock
- Livelock
- Starvation
- Lack of fairness

Race conditions

- More tasks read/modify the same global variable
- “Race” for which task modifies the global value first
- Output depends on the particular sequence of task operations
- Non-deterministic behavior

Race conditions: example 1

```
global int A = 0

task T1 ()
        A = A + 1
        print A
end task

task T2 ()
        A = A + 1
        print A
end task
```

- Two concurrent tasks T1 and T2 are activated
- Both tasks modify the same global variable A
- Expected behavior:
  - T1 reads 0
  - T1 increments A to 1
  - T1 prints 1
  - T2 reads 1
  - T2 increments A to 2
  - T2 prints 2
- **Expected final output = 2**

Race conditions: example 1

```
global int A = 0

task T1 ()
        A = A + 1
        print A
end task

task T2 ()
        A = A + 1
        print A
end task
```

- Problem if operation "A=A+1" is not performed atomically
- Possible behavior:
  - T1 reads 0
  - T2 reads 0
  - T1 increments A to 1
  - T2 increments A to 1
  - T1 prints 1
  - T2 prints 1
- **Final output = 1 !!!**

Race conditions: example 2

Two global variables
{ A = 1; B = 2 }

CPU1		CPU2
A = 3; B = 4;	with out-of-order execution	x = A; y = B;

- n! = 24 different combinations for memory accesses
- 4 different outputs:

  (x,y) = (1,2); (1,4); (3,2); (3,4)

Race conditions: example 3

Five global variables
{ A = 1, B = 2, C = 3, P = &A, Q = &C }

CPU1

```
B = 4;
P = &B;
```

with out-of-order execution

CPU2

```
Q = P;
D = *Q;
```

NB: *Data dependency* → Q is modified in the first instruction and used in the second one

Race conditions: example 3

Five global variables
{ A = 1, B = 2, C = 3, P = &A, Q = &C }

CPU1		CPU2
B = 4; P = &B;	with out-of-order execution	Q = P; D = *Q;

- Partial order enforcement by CPU2 → only 12 different combinations for memory accesses and three different outputs:

  (Q=P,D) = (&A,1); (&B,2); (&B,4)
- D will never receive the value in C

Race conditions on devices

- Some devices present their control interfaces as collections of memory locations
- The order in which control registers are accessed may be crucial
- E.g.: Ethernet card with internal register set accessed through address (A) and data (D) port registers

To read internal register #5:

```
A = 5;
x = *D;
```

Race conditions on devices

Read register #5:

```
A = 5;
x = *D;
```

- Out-of-order execution may cause second instruction be executed before the first one → malfunction!
- Since there is no explicit data dependency, these problems are difficult to detect

Things that can be assumed

- Dependent memory access on a given CPU are always executed in order

```
Q = P;
D = *Q;
```

- Overlapping load and stores within a CPU maintain functional dependencies

```
A = *P;
*P = B;
```

or

```
*P = A;
B = *P;
```

...and that cannot be assumed

- Order of execution of independent load and stores

```
A = *P;
B = *Q;
*R = C;
```

- Order of execution of overlapping load and stores (preserving functional dependencies)

```
1) A = *P;
2) B = *(P + 4);
```

may be: 1 → 2
2 → 1
1 & 2

or

```
1) *P = A;
2) B = *P;
```

may be: 1 → 2
1 & 2 (*P=B=A)

Memory barriers (membar)

- Class of instructions to enforce an order on memory operations
- Low-level machine code operating on shared memory
- Ensures that all operations preceding a membar are executed before all operations following the barrier

Membar: example

CPU 1:

```
load    eax, 0

while (eax == 0)
        load    eax, [a]

print   [b]
```

CPU 2:

```
store   33, [b]
store   1, [a]
```

- Two CPUs, each one running a task using global variables *a* and *b*
- Expected behavior:
  - CPU1 spins until *a* becomes ≠ 0
  - Then prints the *b* value stored by CPU2
- Expected output: 33

but…

CPU 1:

```
load    eax, 0

while (eax == 0)
        load    eax, [a]

print   [b]
```

CPU 2:

```
store   33, [b]
store   1, [a]
```

- When CPU2's operations may be executed out-of-order:
  - *a* may be updated before *b*
  - CPU1 prints a meaningless value
- Not acceptable!

Solution: memory barrier

CPU 1:

```
load    eax, 0

while (eax == 0)
        load    eax, [a]

print   [b]
```

CPU 2:

```
store   33, [b]
membar
store   1, [a]
```

- Use a memory barrier before CPU2's second instruction
- "store 33, [a]" will be always executed before "store 1, [b]"
- Final output: 33