Advanced Computer Architecture

Instructor

Dr. Dinh-Duc Anh-Vu

http://www.cse.hcmut.edu.vn/~anhvu

Chapter 5

SUPERSCALAR PROCESSORS

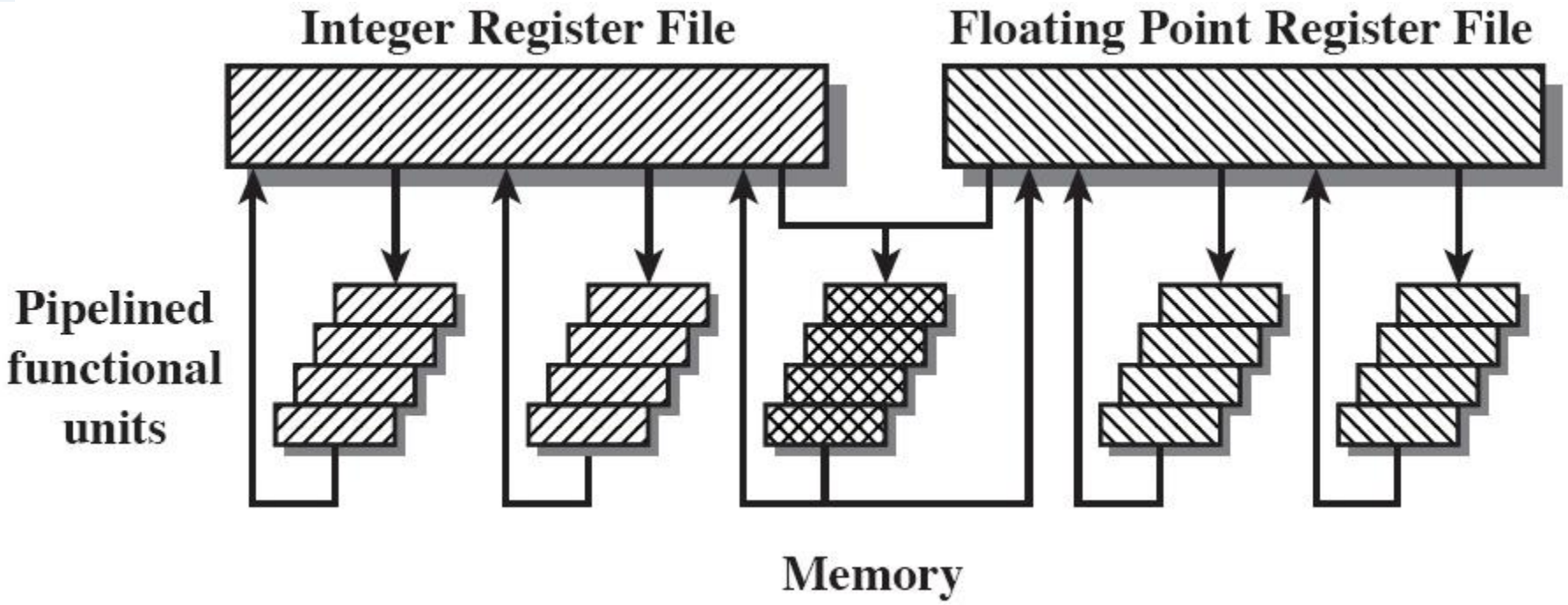
Lecture 5 – Superscalar Processors

- Definition and motivation
- SuperScalar vs. Superpipeline
- Dependency issues
- Parallel instruction execution

Superscalar Architecture

- Superscalar is a computer designed to improve the performance of the execution of **scalar** instructions.
- In a **superscalar architecture** (SSA), several instructions can be initiated simultaneously and executed independently.
- Pipelining allows also several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment.
- SSA includes all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage.
- SSA introduces therefore a new level of parallelism, called **instruction-level parallelism**.

General Superscalar Organization



- In this example, two integer, two floating-point, and one memory (load or store) operations can be executed at the same time.

Superscalar Architecture

- Arrived on the scene hard on the heels of RISC architecture
- Equally applicable to RISC & CISC
- In practice usually RISC
- First superscalar machines became commercially available within just 1 or 2 year of the coining of the term "*superscalar*"
- Become the standard method for implementing high-performance microprocessors

Motivation

- Most operations are on scalar quantities (about 80%).
- Speedup these operations will lead to large overall performance improvement.

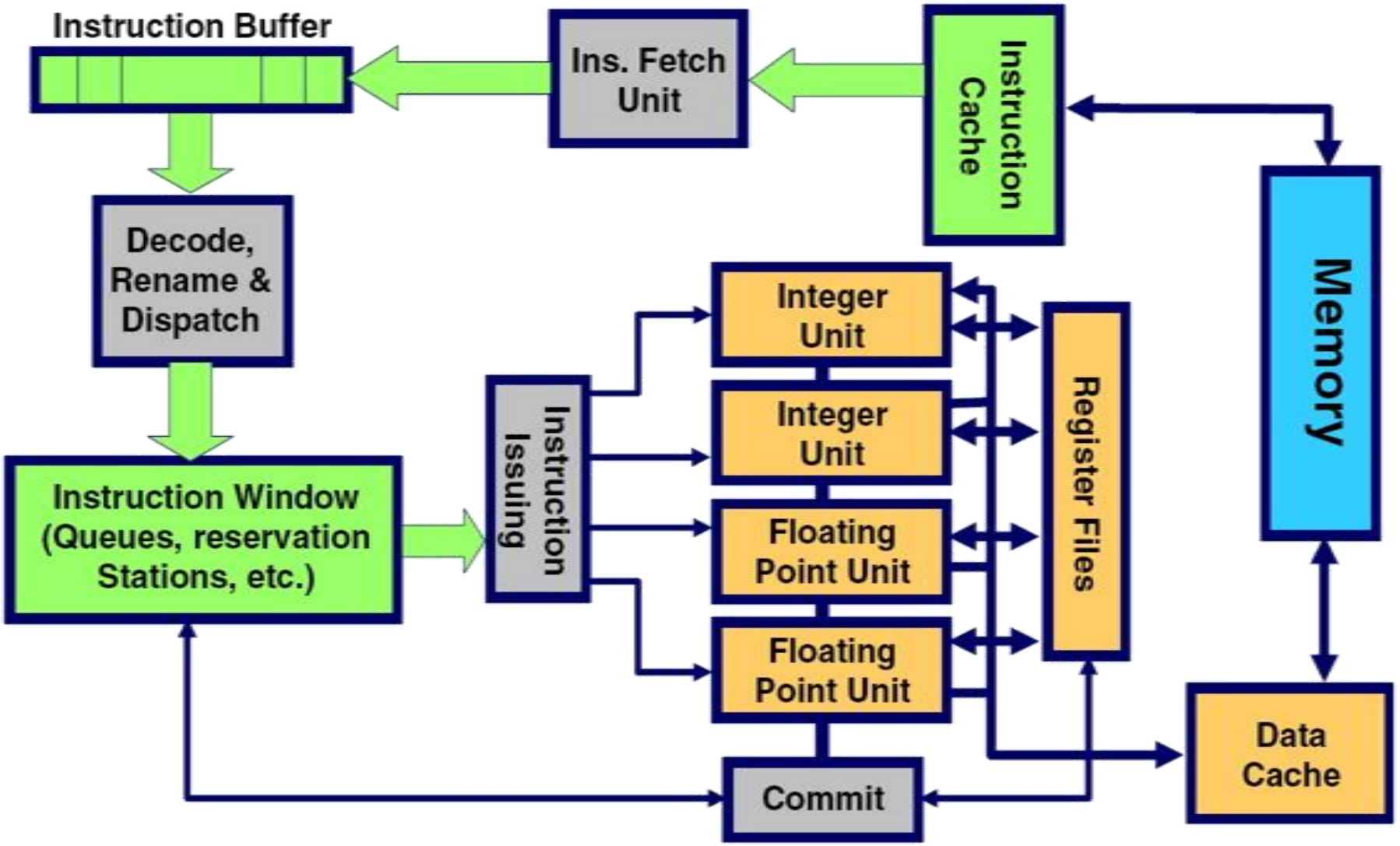
How to implement the idea?

- A SSA processor fetches multiple instructions at a time, and attempts to find nearby instructions that are independent of each other and therefore can be executed in parallel.
- Based on the dependency analysis, the processor may issue and execute instructions in an order that differs from that of the original machine code.
- The processor may eliminate some unnecessary dependencies by the use of additional registers and renaming of register references.

Superscalar Concepts

- Superscalar architectures allow several instructions to be issued and completed per clock cycle.
- A superscalar architecture consists of a number of pipelines that are working in parallel.
- Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.
- In the following example two floating point and two integer operations can be issued and executed simultaneously.
  - Each unit is also pipelined and can execute several operations in different pipeline stages.

A SSA Example



Lecture 5 – Superscalar Processors

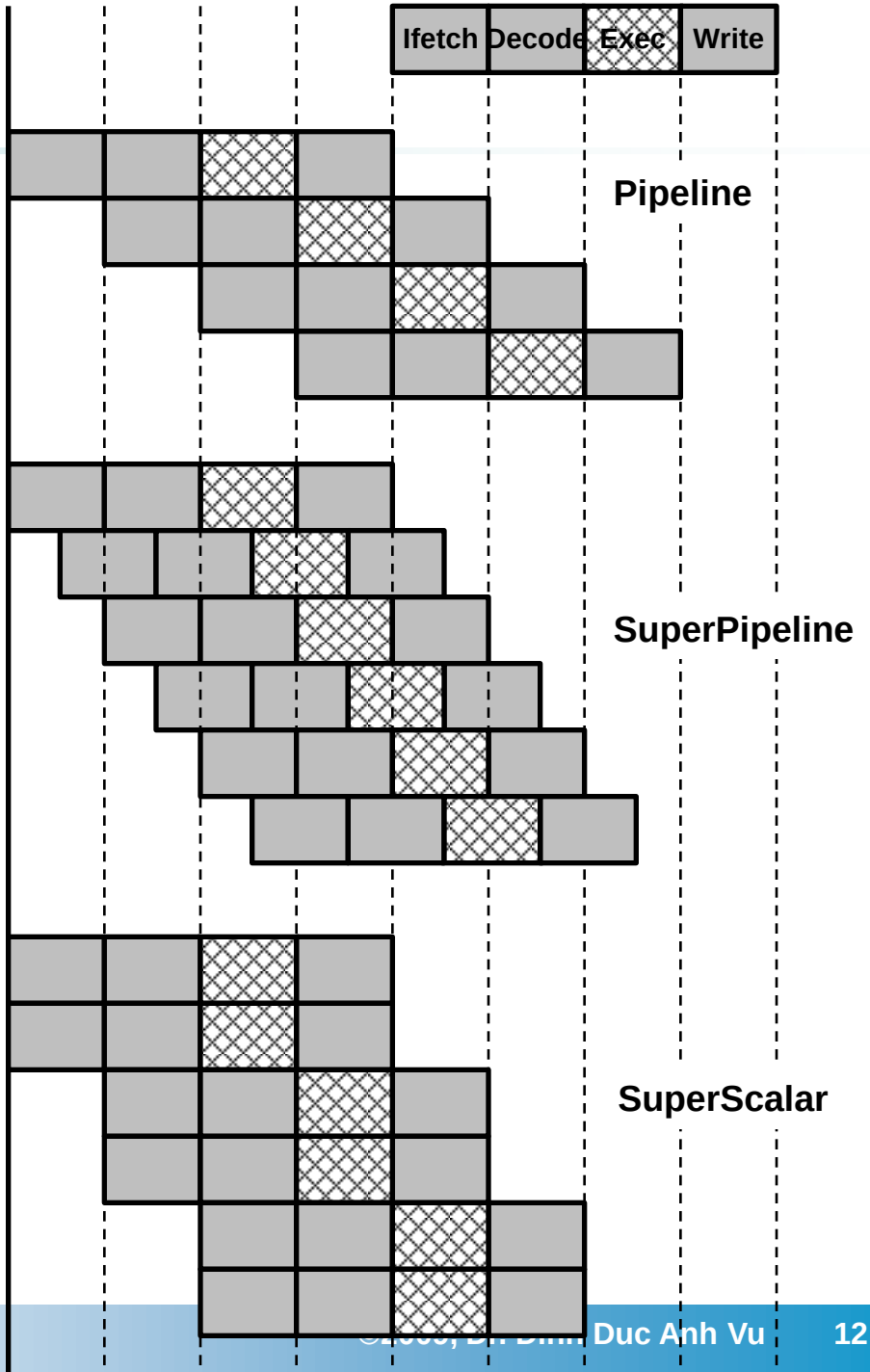
- Definition and motivation
- SuperScalar vs. SuperPipeline
- Dependency issues
- Parallel instruction execution

SuperPipelining

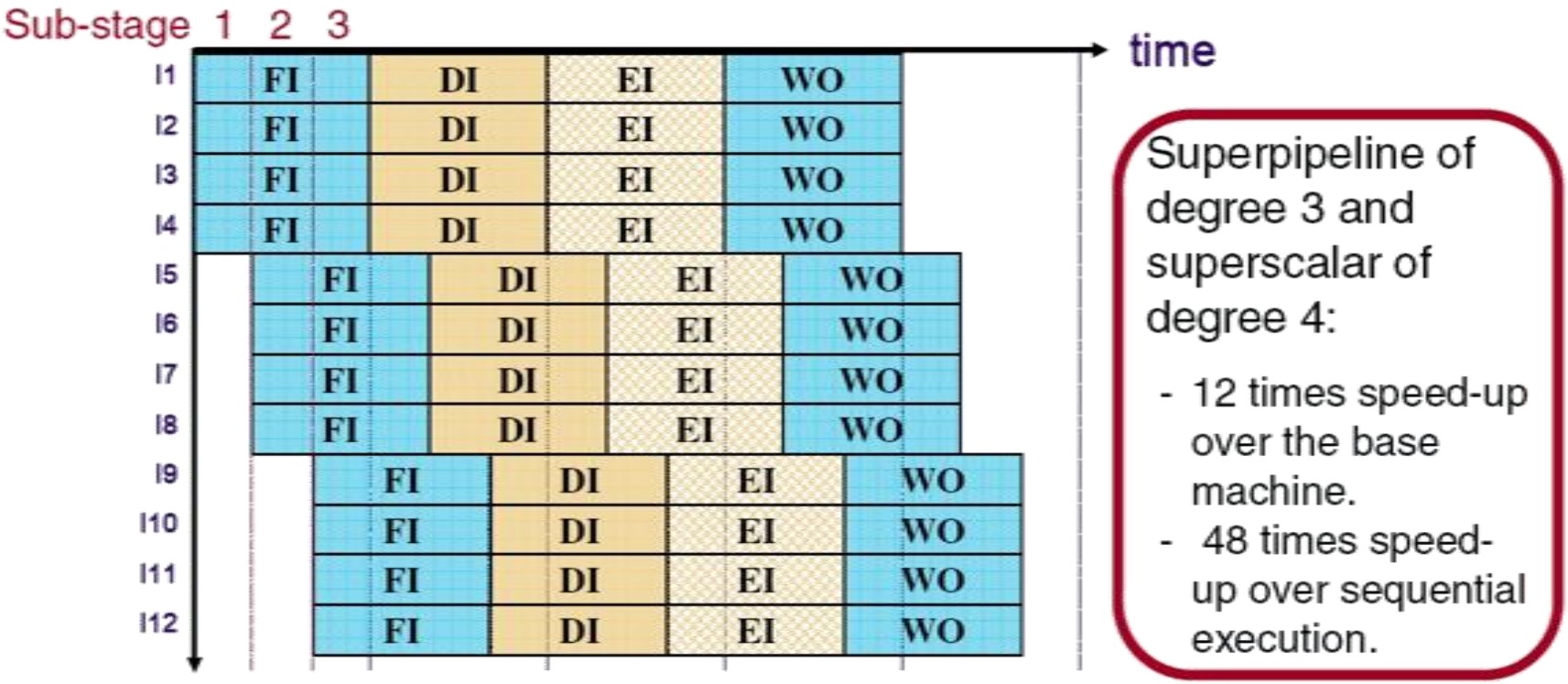
- SuperPipelining is based on dividing the stages of a pipeline into several sub-stages, and thus increasing the number of instructions which are handled by the pipeline at the same time.
- For example, by dividing each stage into two, a pipeline can perform at twice the speed in the ideal situation.
  - Many pipeline stages may perform tasks that require less than half a clock cycle.
  - No duplication of hardware is needed for these stages.
- For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages/sub-stages; increasing the number of stages/sub-stages over this limit reduces the overall performance.

SuperScalar vs. SuperPipeline

- Base machine: 4-stage pipeline
  - Instruction fetch
  - Operation decode
  - Operation execution
  - Result write back
- Superpipeline of degree 2
  - Each sub-stage usually takes half a clock cycle to finish.
- Superscalar of degree 2
  - Two instructions are executed concurrently in each pipeline stage.
  - Duplication of hardware is required.



Superpipelined Superscalar Design



- This is a new trend of architecture design:
  - Pentium Pro(P6): 3-degree superscalar, “14”-stage superpipeline
  - PowerPC 620: 4-degree superscalar.

Lecture 5 – Superscalar Processors

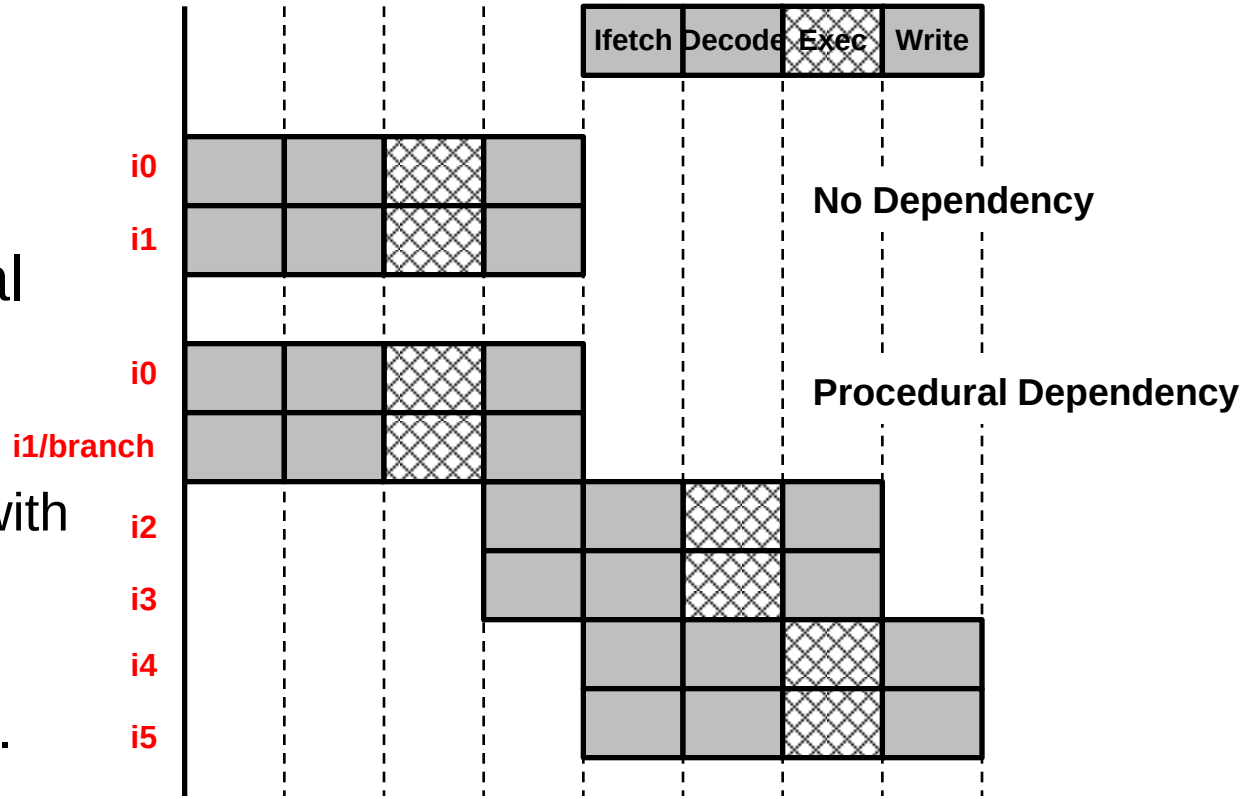
- Definition and motivation
- SuperScalar vs. Superpipeline
- Dependency issues
- Parallel instruction execution

Parallel Execution Limitations

- **Instruction-level parallelism** (ILP) refers to the degree to which (on average) the instructions of a program can be executed in parallel
- To mazimize ILP, use a combination of
  - Compiler-based optimisation
  - Hardware techniques
- The situations which prevent instructions to be executed in parallel by a superscalar architecture are very similar to those which prevent an efficient execution on any pipelined architecture (pipeline hazards)
  - True data dependency
  - Control (Procedural) dependency
  - Resource conflicts
  - Output dependency
  - Antidependency
- The consequences of these situations on SSA are more severe than those on simple pipelines, because the potential of parallelism in SSA is greater and, thus, a greater opportunity might be lost.

Procedural Dependency

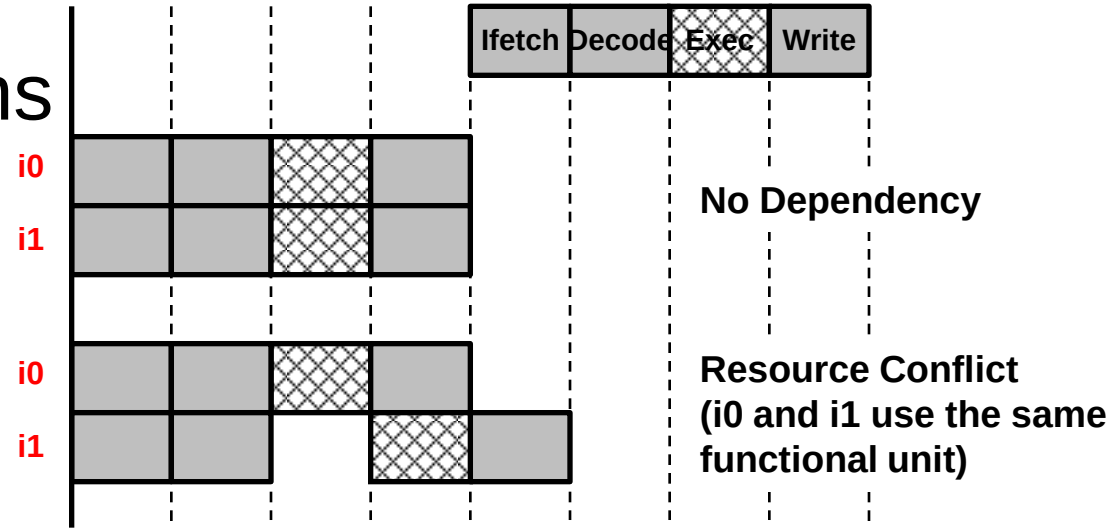
- The presence of branches creates major problems in assuring an optimal parallelism.
  - can not execute instructions after a branch in parallel with instructions before a branch.
  - similar to control hazards in pipeline.



- Also, if instructions are of variable length, they cannot be fetched and issued in parallel, since an instruction has to be decoded in order to find out how many fetches are needed
  - therefore, superscalar techniques are more efficiently applicable to RISCs, with fixed instruction length and format.

Resource Conflicts

- Several instructions compete for the same hardware resource at the same time.
  - e.g., 2 arithmetic instructions need the same floating-point unit for execution.
  - similar to structural hazards in pipeline.



- Can be solved partly by introducing several hardware units for the same functions (duplication of resources).
  - e.g., have two floating-point units.
  - the hardware units can also be pipelined to support several operations at the same time.

Data Conflicts

- Caused by data dependencies between instructions in the program.
  - similar to date hazards in pipeline.
- To address the problem and to increase the degree of parallel execution, SSA provides a great liberty in the order in which instructions can be issued and executed.
- Therefore, data dependencies have to be considered and dealt with much more carefully.

Out of Order Execution

- SSA exploits the potential of instruction-level parallelism present in the program.
- This is often achieved by **dynamic instruction scheduling**:
  - Instructions are issued for execution dynamically, in parallel and out of order.
  - Out of order issuing: instructions are issued independent of their original sequential order, based on dependencies and availability of resources.

- **The results must be identical with those produced by strictly sequential execution!**

Window of Execution

- Due to data dependencies, only some part of the instructions are potential subjects for parallel execution.
- In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.
- **Window of execution**:
  - The set of instructions that is considered for execution at a certain moment.
  - Any instruction in the window can be issued for parallel execution, subject to data dependencies and resource constraints.
- The number of instructions in the window should be as large as possible. However:
  - Capacity to fetch instructions at a high rate is limited.
  - The problem of branches.

Window of Execution Example

```
for (i=0; i<last; i++) {
    if (a[i] > a[i+1]) {
        temp := a[i];
        a[i] := a[i+1];
        a[i+1] := temp;
        change++;
    }
}
```

r6: i (initially 0)
r7: address for a[i]
r3: address for a[i], a[i+1]
r4: last
r5: change
r8: a[i]
r9: a[i+1]

```
L2 move  r3, r7
   load  r8, (r3)    ; r8 := a[i]
   add   r3, r3, #4  ; r3 := r3 + 4
   load  r9, (r3)    ; r9 := a[i+1]
   ble   r8, r9, L3  ; bra if r8 ≤ r9
```

```
   move  r3, r7
   store r9, (r3)    ; a[i] := r9
   add   r3, r3, #4
   store r8, (r3)    ; a[i+1] := r8
   add   r5, r5, #1  ; change++
```

```
L3 add   r6, r6, #1  ; i++
   add   r7, r7, #4
   blt   r6, r4, L2  ; bra if r6 < r4
```

Window of Execution (cont'd)

- The window of execution can be extended over basic block borders by **branch prediction**.
  - Speculative execution.
- With speculative execution, instructions of the predicted path are entered into the window of execution.
  - Instructions from the predicted path are executed tentatively.
  - If the prediction turns out to be correct the state change produced by these instructions will become permanent and visible (the instructions commit); if not, all effects are removed

Data Dependencies

- All instructions in the window of execution may begin execution, subject to data dependence (and resource) constraints.
- Three types of data dependencies can be identified:
  - True data dependency
  - Output dependency
  - Anti-dependency

} Artificial dependencies

True Data Dependency

- Also called **flow dependency** or **write-read dependency**
- True data dependencies exist when the output of one instruction is required as an input to a subsequent instruction:

```
MUL R4, R3, R1   ; (R4 := R3 * R1)
. . .
ADD R2, R4, R5   ; (R2 := R4 + R5)
```

  - Can fetch and decode second instruction in parallel with first.
  - Can NOT execute second instruction until first is finished.
- They are intrinsic features of the user's program, and cannot be eliminated by compiler or hardware techniques.
- They have to be detected and treated.
  - The addition above cannot be executed before the result of the multiplication is available.
  - The simplest solution is to stall the adder until the multiplier has finished.
  - In order to avoid the adder to be idle, the compiler or hardware can find other instructions which can be executed by the adder until the result of the multiplication is available.

True Data Dependency (Cont'd)

```
ADD  r1, r2     ; (r1 := r1+r2)
MOVE r3, r1     ; (r3 := r1)
```

- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished
- No delay is introduced when a simple pipeline is used

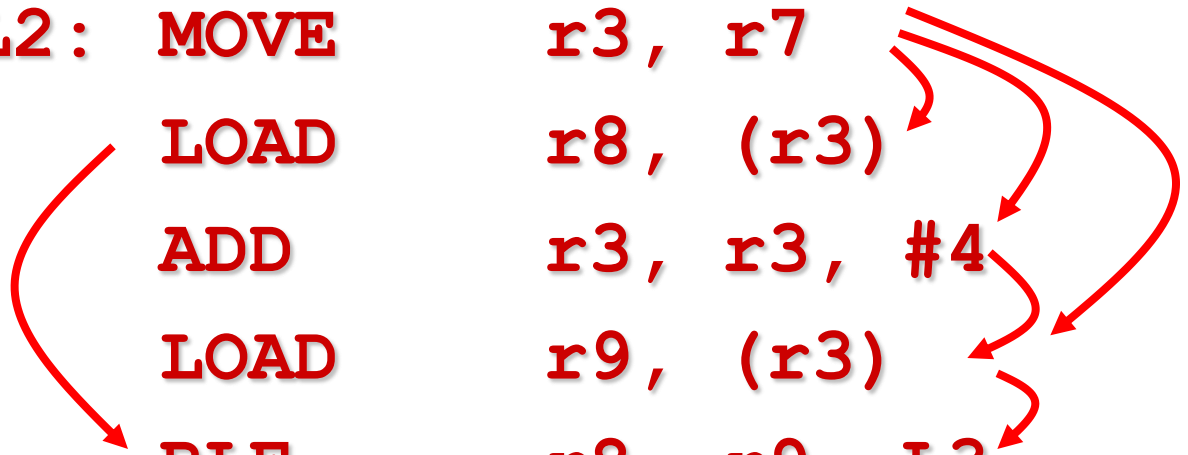
True Data Dependency (Cont'd)

```
LOAD r1, eff ; (r1 := [eff])
MOVE r3, r1  ; (r3 := r1)
```

- Typical RISC processor take 2 (or more) cycles to perform a load from memory (because of delay of an off-chip memory or cache access)
- The compiler can reorder instructions so that one or more subsequent instructions (that donot depend on the memory load) can begin flowing through the pipeline
- In a superscalar pipeline, this scheme is less effective
  - The independent instructions executed during the load are likely to be executed on the first cycle of the load
  - It leaves the processor with nothing to do until the load completes

Problem: True Data Dependency

```
L2:  MOVE    r3, r7
     LOAD    r8, (r3)
     ADD     r3, r3, #4
     LOAD    r9, (r3)
     BLE     r8, r9, L3
```



- There are often a lot of true data dependencies in a small region of a program.
- A compiler cannot help to eliminate them!

Output Dependency

- An **output dependency (write-write dependency)** exists if two instructions are writing into the same location.
- If the second instruction writes before the first one, an error occurs:

```
MUL     R4, R3, R1; (R4 := R3 * R1)
…
ADD     R4, R2, R5; (R4 := R2 + R5)
```

```
L2:  MOVE r3, r7
     LOAD r8, (r3)
     ADD  r3, r3, #4
     LOAD r9, (r3)
     BLE  r8, r9, L3
```

Anti-dependency

- An **anti-dependency (read-write dependency)** exists if an instruction uses a location as an operand while a following one is writing into that location.
- If the first one is still using the location when the second one writes into it, an error occurs:

```
MUL     R4, R3, R1; (R4 := R3 * R1)
…
ADD     R3, R2, R5; (R3 := R2 + R5)
```

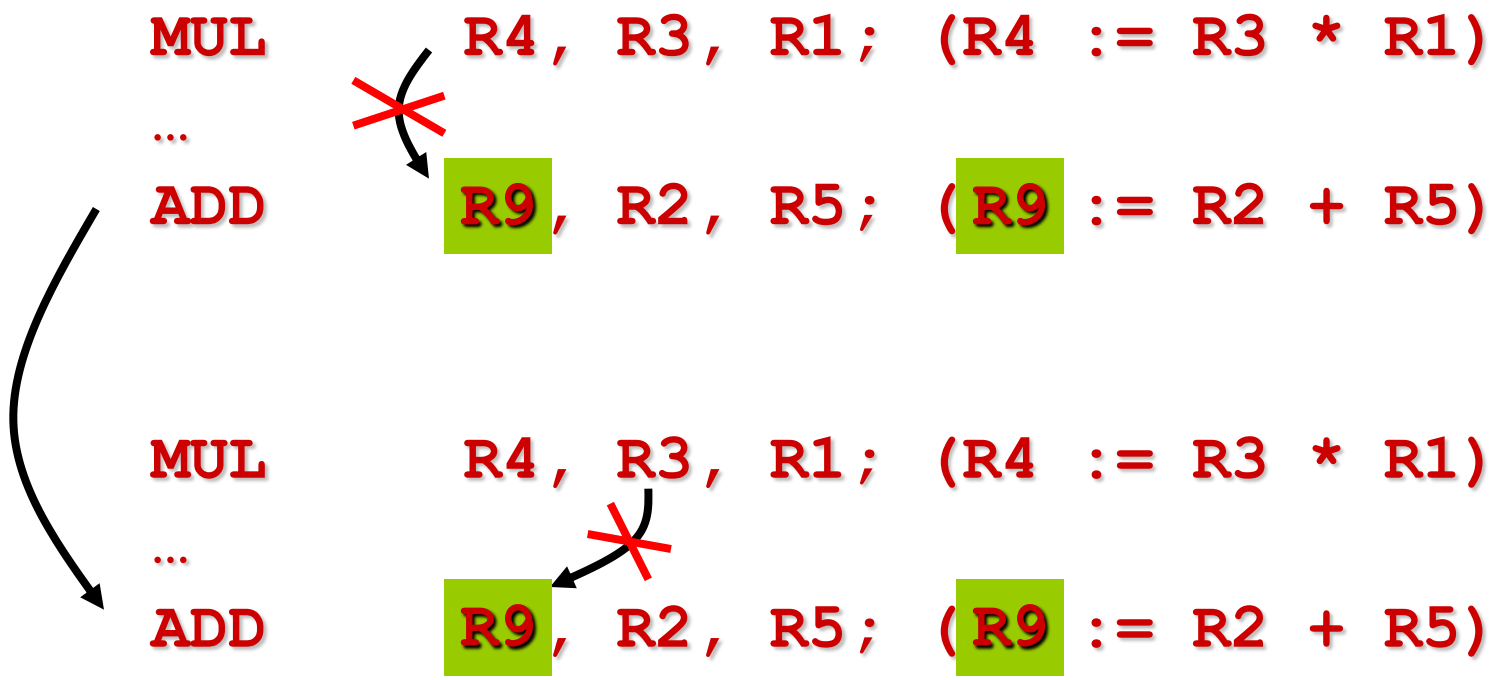
```
L2:  MOVE r3, r7
     LOAD r8, (r3)
     ADD  r3, r3, #4
     LOAD r9, (r3)
     BLE  r8, r9, L3
```

Output and Anti-Dependencies

- Output dependencies and anti-dependencies are not intrinsic features of the executed program.
  - They are not real data dependencies but storage conflicts.
  - They are due to the competition of several instructions for the same register.
- They are only the consequence of the manner in which the programmer or the compiler are using registers (or memory locations).
- In the previous examples the conflicts are produced only because:
  - The output dependency: R4 is used by both instructions to store the result (due to, for example, optimization of register usage);
  - The anti-dependency: R3 is used by the second instruction to store the result.

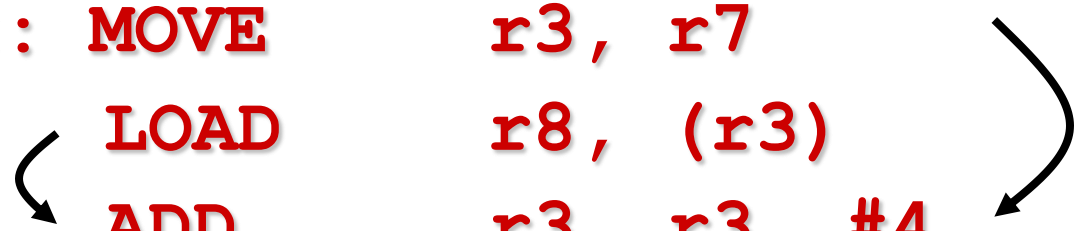
Output and Anti-Dependencies (Cont'd)

- Output dependencies and anti-dependencies can usually be eliminated by using additional registers.
- This technique is called register renaming.



Problem: Output and Anti-Dependencies

```
L2: MOVE    r3, r7
    LOAD    r8, (r3)
    ADD     r3, r3, #4
    LOAD    r9, (r3)
    BLE     r8, r9, L3
```



```
L2: MOVE    r3, r7
    LOAD    r8, (r3)
    ADD     r1, r3, #4
    LOAD    r9, (r1)
    BLE     r8, r9, L3
```

Lecture 5 – Superscalar Processors

- Definition and motivation
- Superpipeline
- Dependency issues
- Parallel instruction execution

Instruction vs. Machine Parallelism

- **Instruction-level parallelism (ILP)** – the average number of instructions in a program that a processor might be able to execute at the same time.
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions.
- **Machine parallelism** of a processor – the ability of the processor to take advantage of the ILP of the program.
  - Determined by the number of instructions that can be fetched and executed at the same time, i.e., the capacity of the hardware, and by the speed and sophistication of the mechanisms that the processor uses to find independent instructions.

- To achieve high performance, we need both ILP and machine parallelism.
  - The ideal situation is that we have the same ILP and machine parallelism.

Instruction Issue and Completion

Parallel instruction execution can be characterized by the order of the following three activities:

- **Instruction issue** – an instruction is initiated and starts execution.
- **Instruction completion** – an instruction has competed its specified operations.
- **Instruction commit** – the results of the instruction operations are written back to the register files or cache.
  - The machine state is changed.

Instruction Execution Policies

- Instructions will be executed in an order different from the strictly sequential one, with the restriction that **the results must be correct.**

- **Execution policies** usually used:
  - In-order issue (IOI) with in-order completion (IOC).
  - In-order issue with out-of-order completion (OOC).
  - Out-of-order issue (OOI) with out-of-order completion.

IOI with IOC

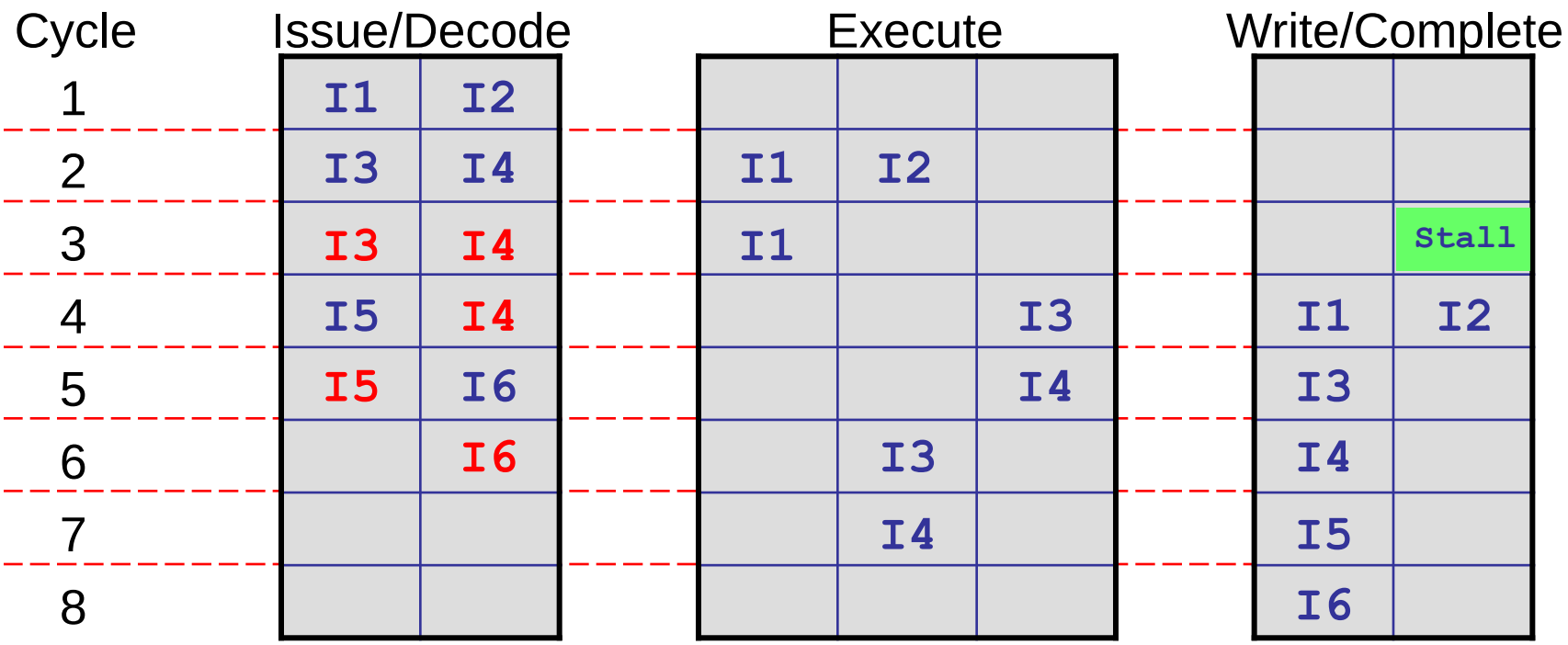
- Instructions are issued in exact program order, and completed in the same order.
  - An instruction cannot be issued before the previous one has been issued;
  - An instruction cannot be completed before the previous one has been completed.
- To guarantee in-order completion, instruction issuing stalls when there is a conflict and when the unit requires more than one cycle to execute;

Example:

- Assume a processor that can issue and decode two instructions per cycle, that has three functional units (two single-cycle integer units, and a two-cycle floating-point unit), and that can complete (and write back) two results per cycle.
- And an instruction sequence with the characteristics given in the next slide.

IOI with IOC Example

```
I1 - Needs two execute cycles (floating-point)
I2 -
I3 -
I4 - Needs the same function unit as I3
I5 - Needs data value produced by I4
I6 - Needs the same function unit as I5
```



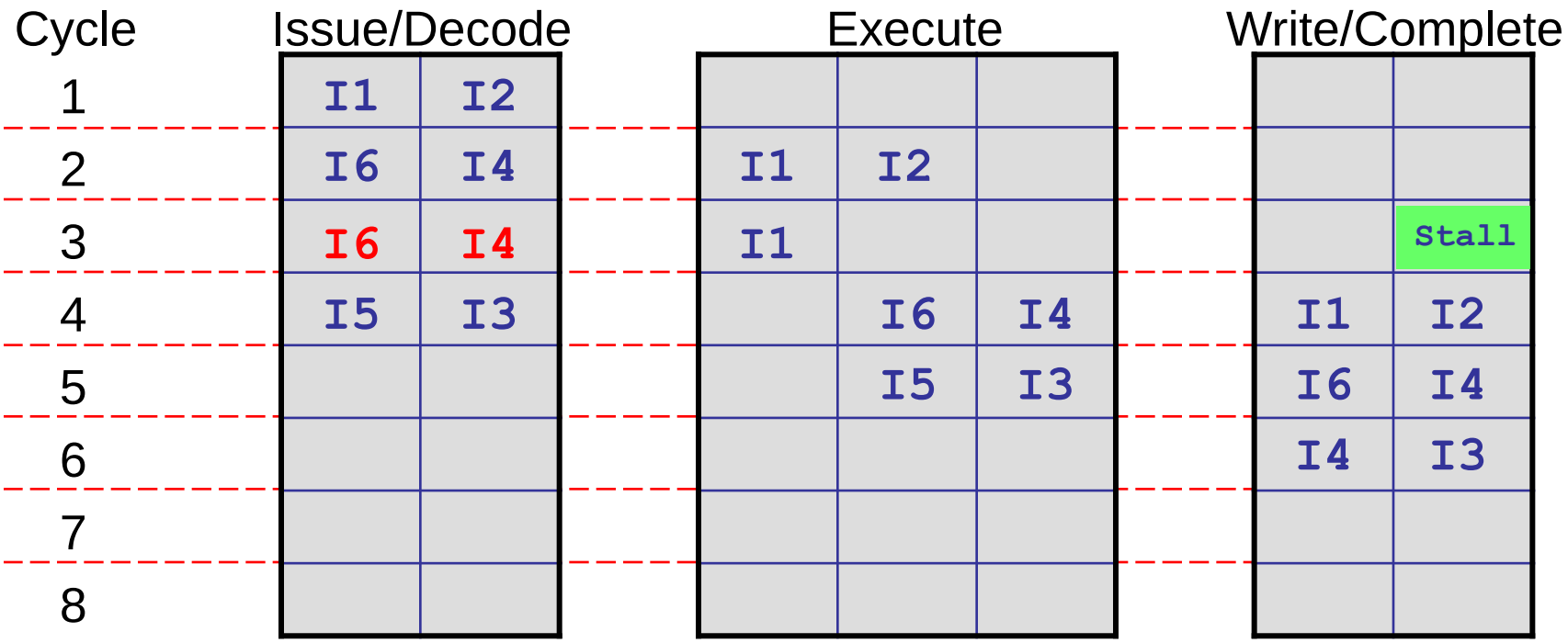
IOI with IOC Discussion

- The processor detects and handles (by stalling) true data dependencies and resource conflicts.
- As instructions are issued and completed in their strict order, the resulting parallelism is very much dependent on the way the program has been written/compiled.
  - Ex. if I3 and I6 switch position, the pairs I4/I6 and I3/I5 can be executed in parallel (see the following slide).

- **With superscalar, we are interested in techniques that are not compiler-based, but allow the hardware alone to detect instructions which can be executed in parallel and to issue them accordingly.**

IOI with IOC Example (Cont'd)

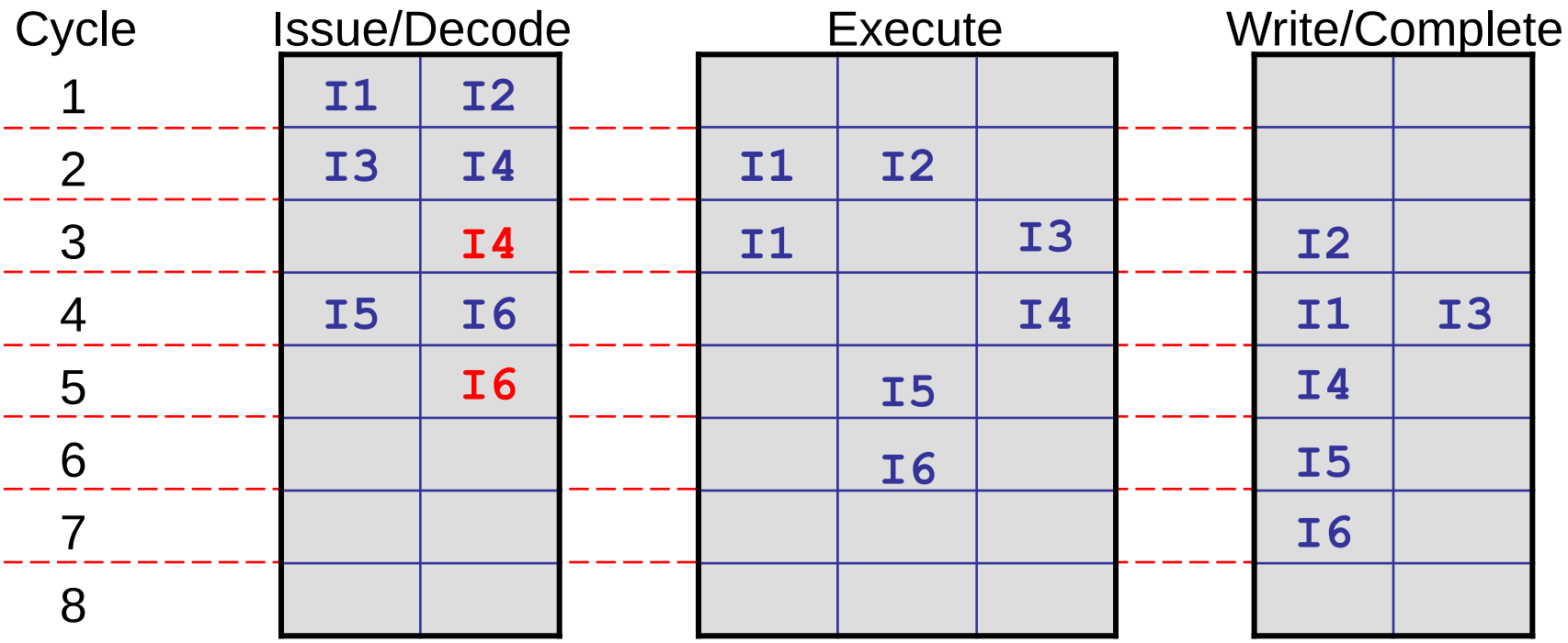
```
I1 - Needs two execute cycles (floating-point)
I2 -
I6 - Needs the same function unit as I5
I4 - Needs the same function unit as I3
I5 - Needs data value produced by I4
I3 -
```



IOI with OOC

- With out-of-order completion, a later instruction may complete before a previous one.
  - Address mainly the issue of long-latency operations such as division

```
I1 – Needs two cycles
I2 –
I3 –
I4 – Conflict with I3
I5 – Depending on I4
I6 – Conflict with I5
```

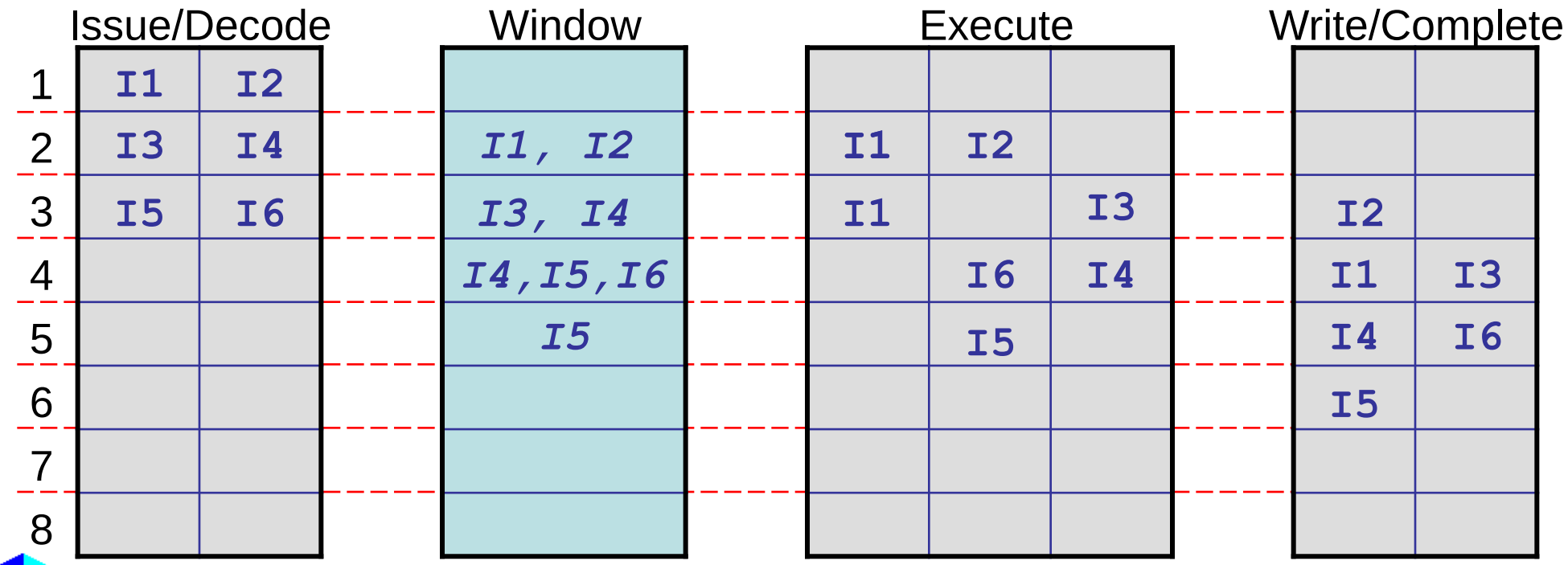


OOI with OOC

- With in-order issue, no new instruction can be issued when the processor has detected a conflict, and is stalled until after the conflict has been resolved.
  - The processor is not allowed to look ahead for further instructions, which could be executed in parallel with the current ones.
- Out-of-order issue takes a set of decoded instructions, issues any instruction, in any order, as long as the program execution is correct.
  - Decouple decode pipeline from execution pipeline, by introducing an instruction window.
  - When a functional unit becomes available an instruction can be executed.
  - Since instructions have been decoded, processor can look ahead.

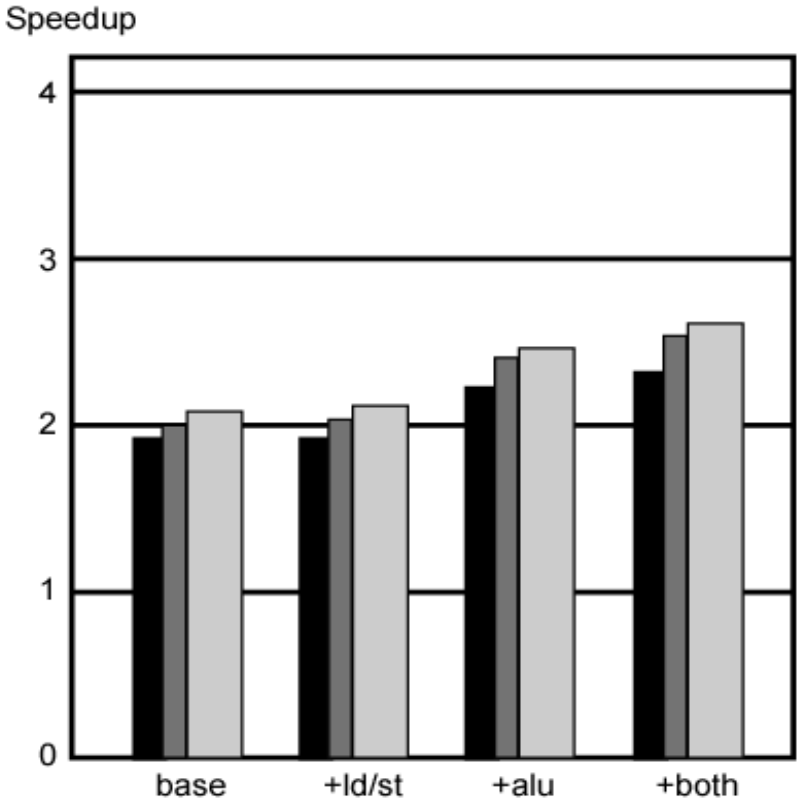
OOI with OOC Example

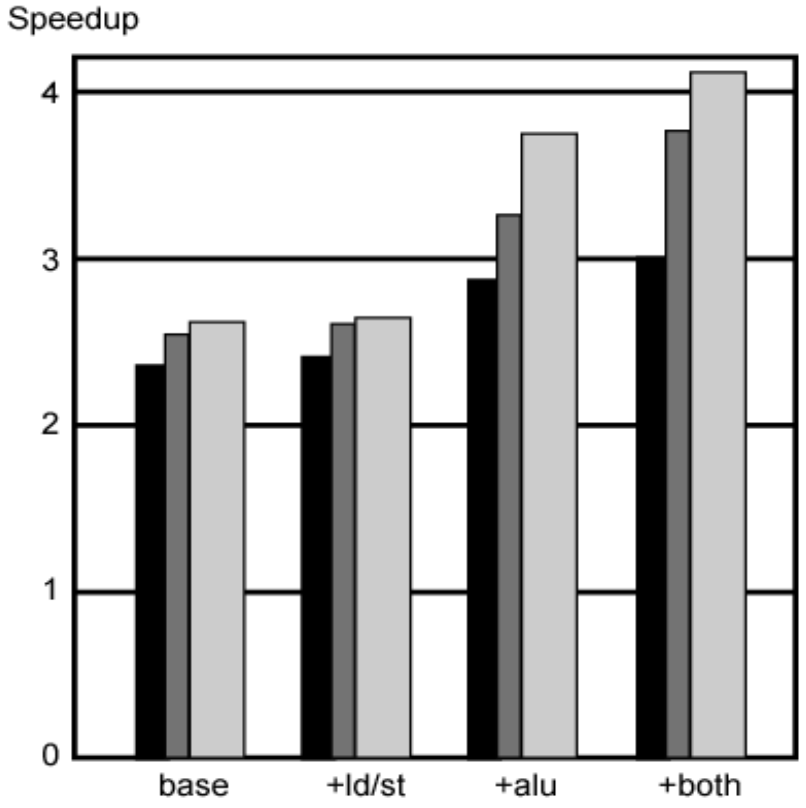
```
I1 – Needs two cycles
I2 –
I3 –
I4 – Conflict with I3
I5 – Depending on I4
I6 – Conflict with I5
```



Speedups of Machine Organizations Without Procedural Dependencies

- Not worth duplication functions without register renaming
- Need instruction window large enough (more than 8)

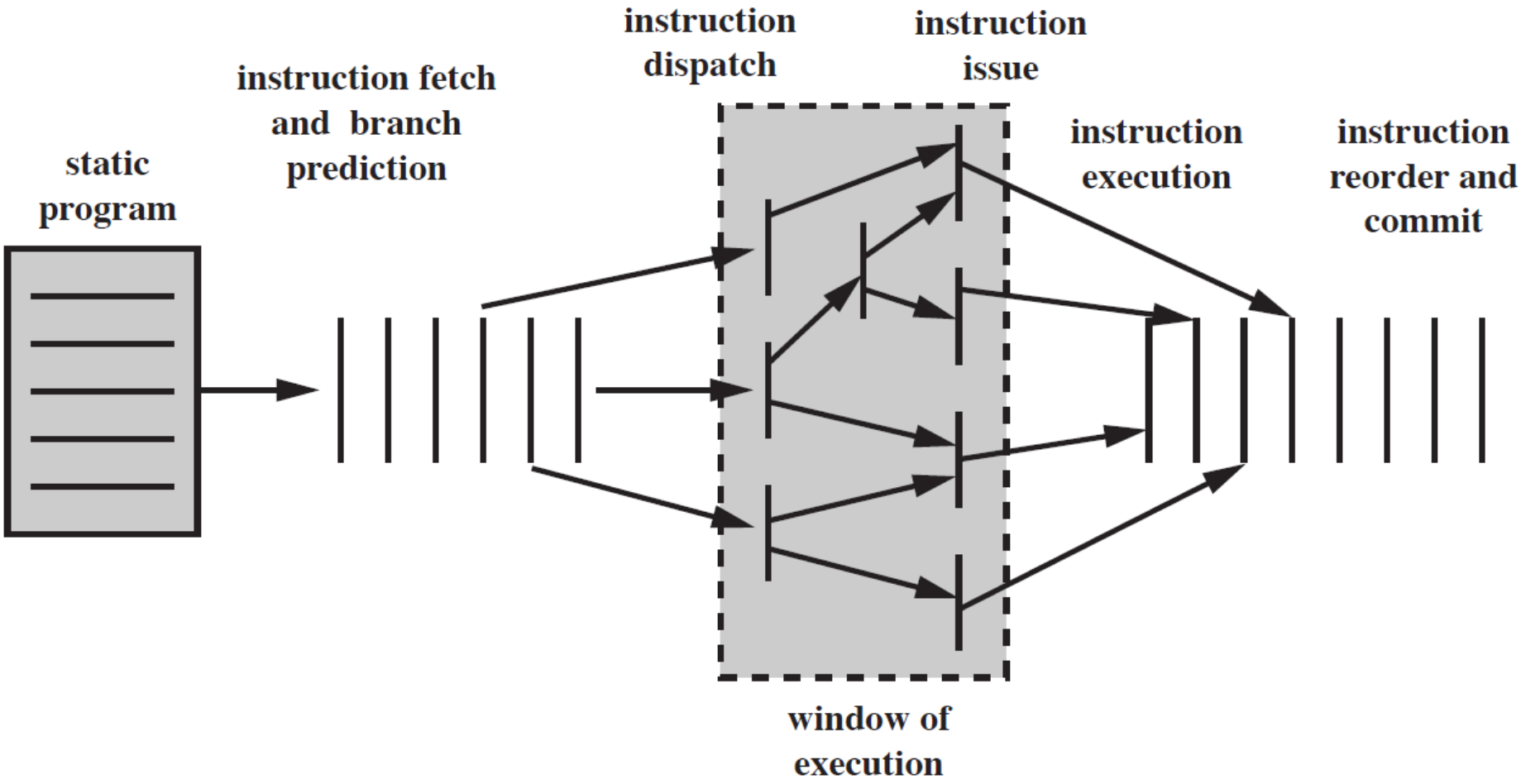




Branch Prediction

- 80486 fetches both next sequential instruction after branch and branch target instruction
- Gives two cycle delay if branch taken
- RISC – Delayed Branch
  - Calculate result of branch before unusable instructions pre-fetched
  - Always execute single instruction immediately following branch
  - Keeps pipeline full while fetching new instruction stream
  - Not as good for superscalar
    - Multiple instructions need to execute in delay slot
    - Instruction dependence problems
  - Revert to branch prediction

Superscalar Execution



Superscalar Implementation

- Simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values
- Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order

Final Remarks

- The following techniques are main features for superscalar processors:
  - Several pipelined units which are working in parallel;
  - Out-of-order issue and out-of-order completion;
  - Register renaming.
- All of the above techniques are aimed to enhance performance.
- Experiments have shown:
  - Only adding additional functional units is not very efficient;
  - Out-of-order issue is extremely important; it allows to look ahead for independent instructions;
  - Register renaming can improve performance with more than 30%; in this case performance is limited only by true dependencies.
  - It is important to provide a fetching/decoding capacity so that the window of execution is sufficiently large.