Computer Architecture

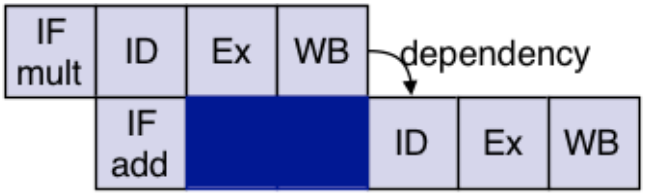
R. Poss
Computer Systems Architecture group (UvA)
e-mail: r.c.poss@uva.nl

Data hazards

Data hazard

- Occurs when the output of one operation is the input of a subsequent operation
- The hazard occurs because of the latency in the pipeline
  - the **result** (output from one instruction) is not written back to the register file until the last stage of the pipe
  - the **operand** (input of a subsequent instruction) is required at register read – some cycles prior to writeback
  - the longer the RR to WB delay, the more cycles there must be between the writeback of the producer instruction and the read from the consumer
- Example:

mult a b c
add d a f

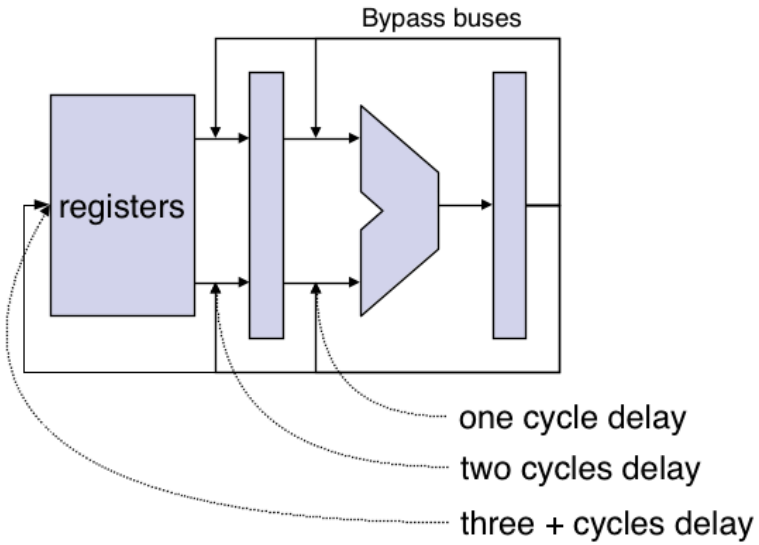


Two bubbles

n.b. register read is in the ID stage

How to overcome

- **Do nothing** i.e. expose to the programmer, e.g. MIPS 1
- **Stall the read stage**
- **Bypass buses**:



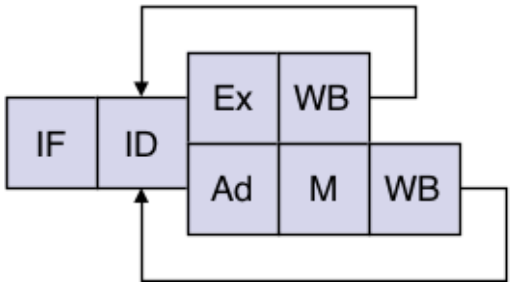
The operand is taken from the pipeline register and input directly to the ALU on the subsequent cycle

- Also: **reorder the instructions**

Structural hazards and scalar ILP

Scalar pipelines

- In the simple pipeline, register-to-register operations have a wasted cycle
  - a memory access is not required, but this stage still requires a cycle to complete the operations
- Decoupling memory access and operation execution avoids this
  e.g. use an ALU plus a memory unit - this is **scalar ILP**



Performs R-to-R ops

Performs loads and stores

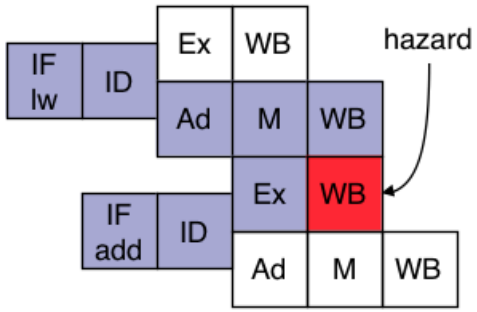
- ... note: either we need two write ports to the register file or arbitration on a single port

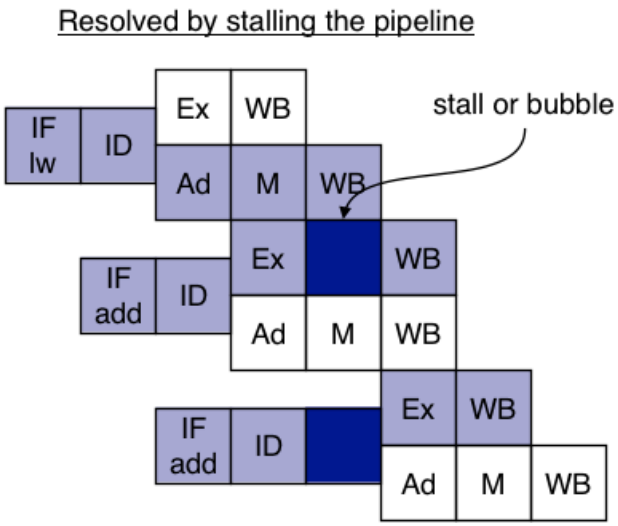
Structural hazard - registers

- A structural hazard occurs when **a resource in the pipeline is required by more than one instruction**
- a resource may be an execution unit or a register port
  - Example: **only one write port**

    lw a addr

    add b c d

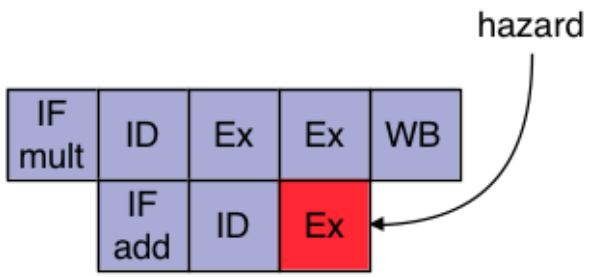




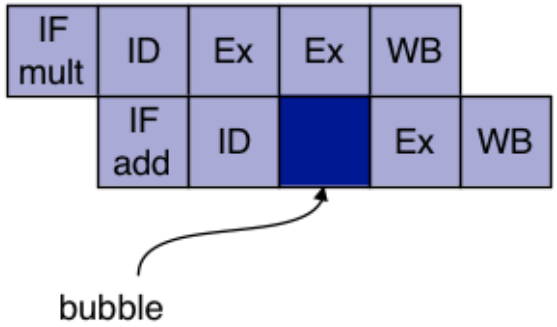
Structural hazard - execution units

- Some operations require more than one pipeline cycle
  - mult is more complex than add (often requires 2 cycles)
  - floating point still more complex still (~ 5 cycles)
- Example: 2-cycle multiply

mult c d e
add f g h

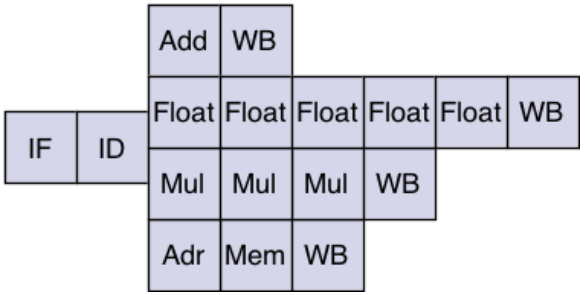


Resolved again by stalling the pipeline



How to overcome

- They result from **contention**
  ⇒ they can be removed by **adding more resources**
  - register write hazard: add more write ports
  - execution unit: add more execution units
- Example: CDC 6600 (1963)
  10 units, 4 write ports, only FP div not pipelined



- Note:
  - **more resources = more cost** (area, power)

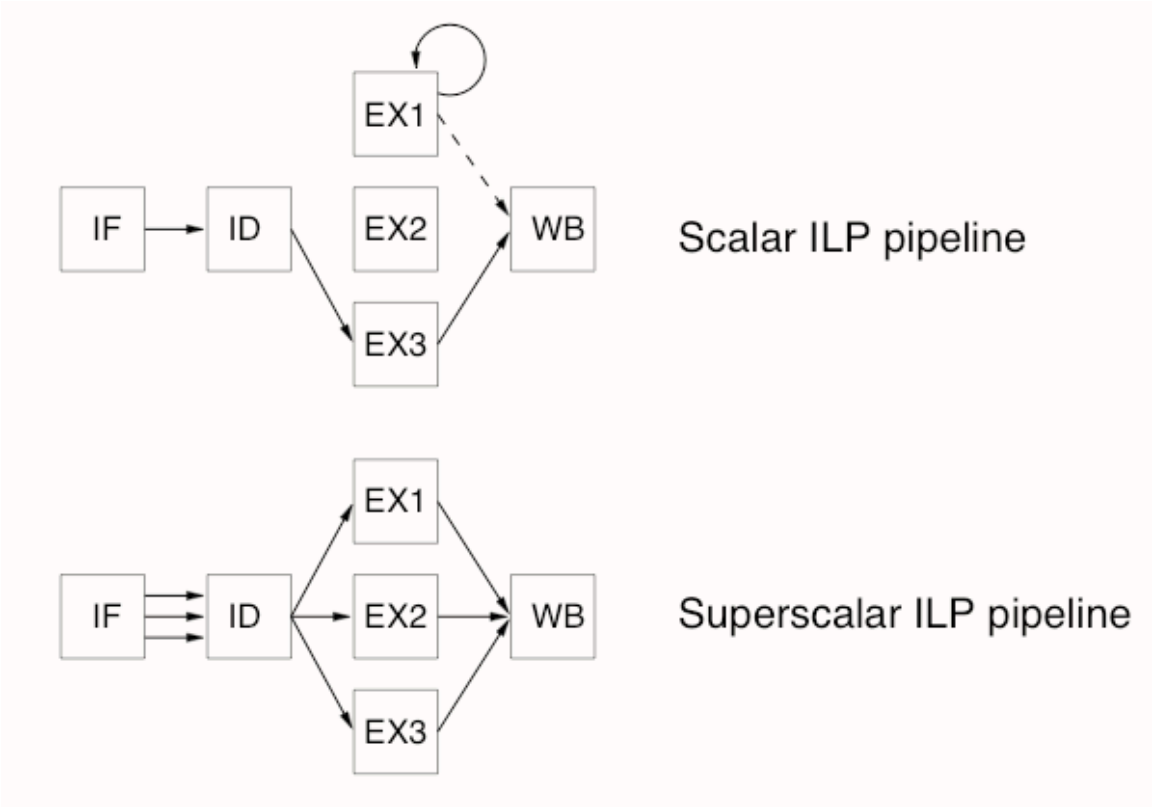
Superscalar processors

Introduction / overview

Pipelining - summary

- Depth of pipeline - **Superpipelining**
  - further dividing pipeline stages **increases frequency**
  - but introduces **more scope for hazards**
  - and higher frequency means **more power dissipated**
- Number of functional units - **Scalar pipelining -** avoids waiting for long operations to complete
  - instructions fetched and **decoded in sequence**
  - multiple operations **executed in parallel**
- Concurrent issue of instructions - **Superscalar ILP**
  - multiple instructions **fetched and decoded concurrently**
  - new **ordering issues** and **new data hazards**

Scalar vs. superscalar



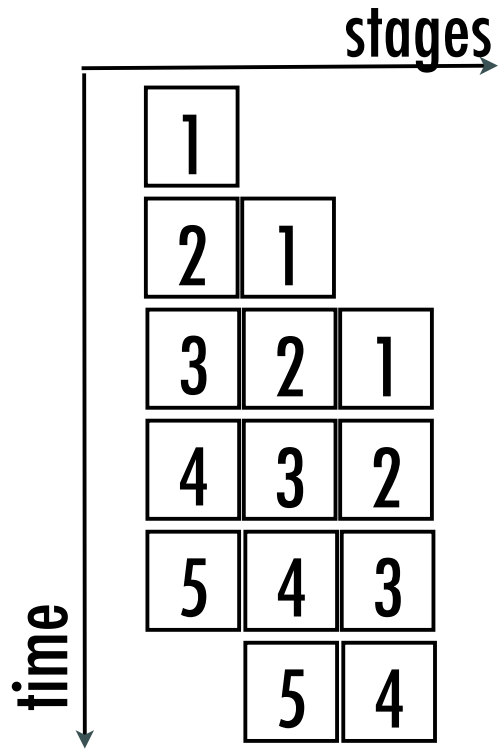
in-order issue

concurrent issue,
possibly out of order

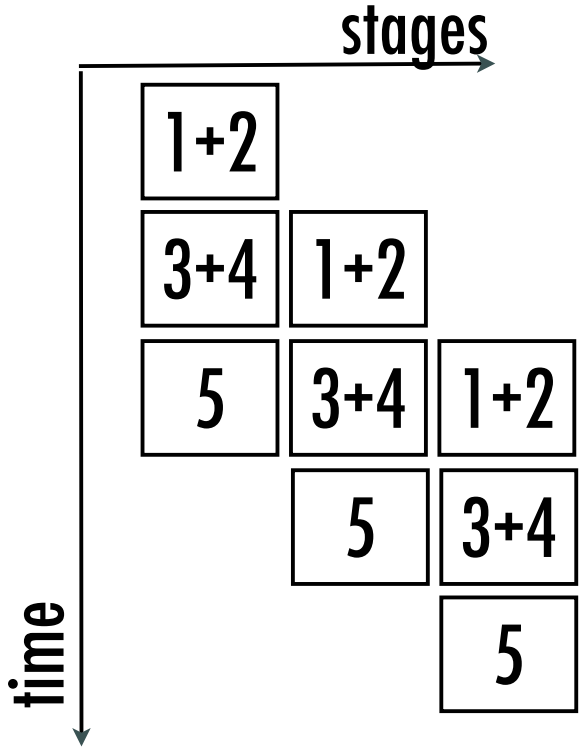
Most "complex" general-purpose processors are superscalar

Basic principle

Example based on simple 3-stage pipeline



Scalar pipeline, max IPC = 1

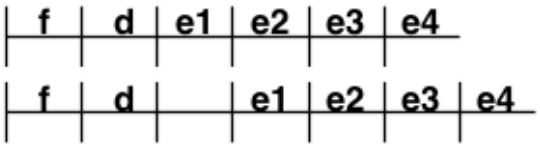


Superscalar, max IPC ≥ 1

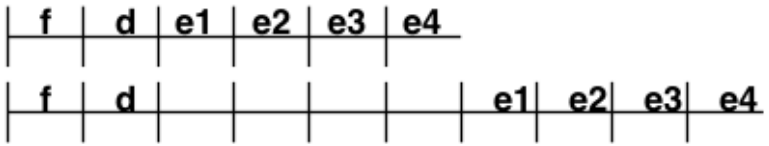
Instruction-level parallelism

- **ILP** is the number of instructions issued per cycle (**issue parallelism / issue width**)
- **IPC** the number of instructions executed per cycle is limited by:
  - the ILP
  - the number of true dependencies
  - the number of branches in relation to other instructions
  - the latency of operations in conjunction with dependencies
- Current microprocessors: 4-8 max ILP, 12 functional units, however IPC of typically 2-3

Long execution time with resource dependency (pipelined fn. unit)



Long execution time with a true data dependency



Aspects of superscalar execution

- parallel fetch decoding and issue
  - 100s of instructions in-flight simultaneously
- out-of-order execution and sequential consistency
  - Exceptions and false dependencies
- finding parallelism and scheduling its execution
- application specific engines, e.g. SIMD & prefetching

Instruction policies & related hazards

Instruction issue vs completion, new data hazards

Instruction issue basics

- Just widening of the processor's pipeline does not necessarily improve its performance
- The processor's **policy in fetching, decoding and executing instructions** also has a significant effect on its performance
- The instruction issue policy is determined by its **look-ahead capability** in the instruction stream
  - For example with no look-ahead, if a resource conflict halts instruction fetching the processor is not able to find any further instructions until the conflict is resolved
  - If the processor is able to continue fetching instructions it may find an independent instruction that can be executed on a free resource out of programmed order
- Policies characterized by **issue order** and **completion order**

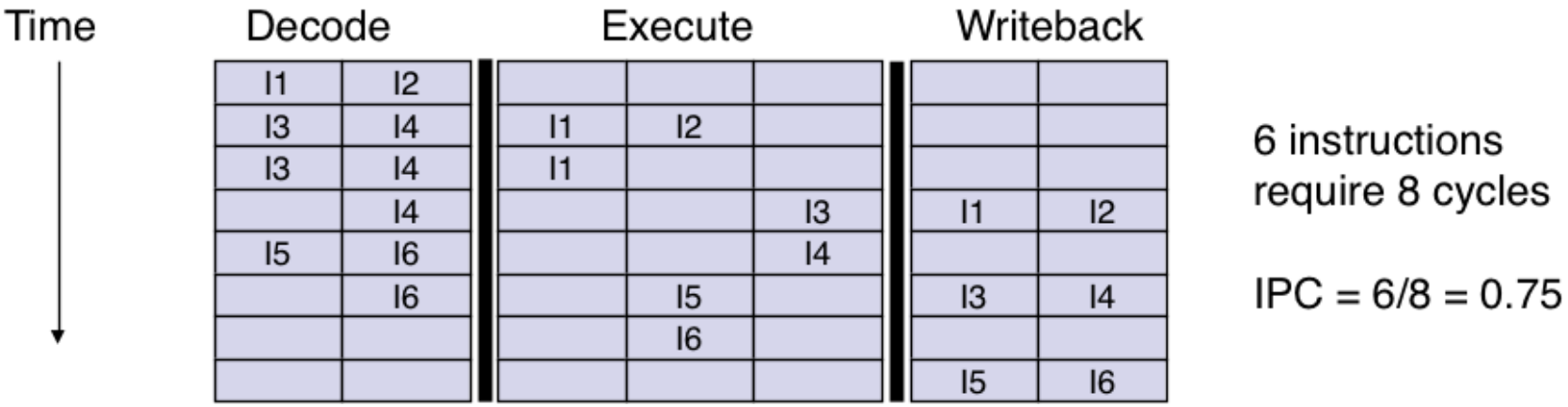
In-order issue, in-order completion

- Simplest, unusual with superscalar designs
- Instructions issued in exact program order with results written in the same order
- This is shown here for comparison purposes only, as very few pipelines use in-order completion

In-order issue, in-order completion

Assume a 3 stage execution in a pipeline that can issue two instructions, execute three instructions and write back two results every cycle... assume:

- I1 requires 2 cycles to execute
- I3 and I4 are in conflict for a functional unit
- I5 depends on the value produced by I4
- I5 and I6 are in conflict for a functional unit



6 instructions require 8 cycles

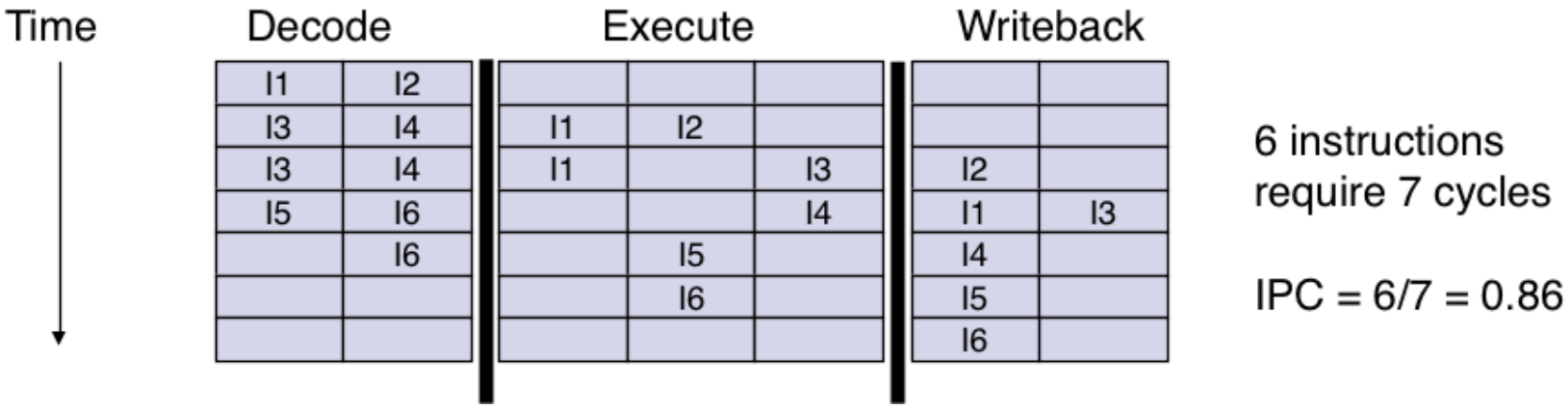
IPC = 6/8 = 0.75

In-order issue, out-of-order completion

- Out-of-order completion, improves performance of instructions with long latency operations, such as loads and floating point
- The modifications made to execution are:
  - any number of instructions allowed in the execution stage up to the total number of pipeline slots (stages × functional units)
  - instruction issue is not stalled when an instruction takes more than one cycle to complete

In-order issue, out-of-order completion

- Again assume a processor issues two instructions, executes three instructions and writes back two results every cycle
  - I1 requires 2 cycles to execute
  - I3 and I4 are in conflict for a functional unit
  - I5 depends on the value produced by I4
  - I5 and I6 are in conflict for a functional unit



In-order issue, out-of-order completion

- In a processor with out-of-order completion, instruction issue is stalled when:
  - There is a **conflict** for a functional unit
  - An instruction depends on a result that is not yet computed - a **data dependency**
    - can use register specifiers to detect dependencies between instructions and logic to ensure synchronisation between producer and consumer instructions – e.g. scoreboard logic, cf CDC 6600
  - Also: a new type of dependency caused by out-of-order completion: the **output dependency**

Output dependencies

- Consider the code to the right:
  - the 1st instruction must be completed before the 3rd, otherwise the 4th instruction may receive the wrong result!
  - this is a new type of dependency caused by allowing out-of-order completion
  - the result of the 3rd instruction has an **output dependency** on the 1st instruction
  - the 3rd instruction must be stalled if its result may be overwritten by a previous instruction which takes longer to complete

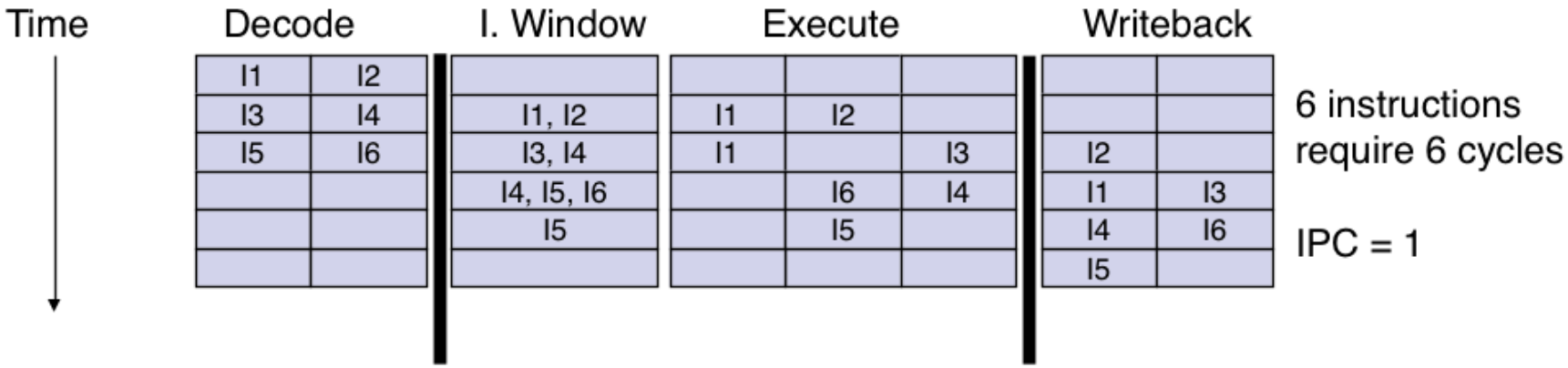
```
R3 := R3 op R5
R4 := R3  + 1
R3 := R5  + 1
R7 := R3 op R4
```

Out-of-order issue, out-of-order completion

- In-order issue stalls when the decoded instruction has:
  - a resource conflict, a true data dependency or an output dependency on an uncompleted instruction
  - this is true even if instructions after the stalled one can execute
  - to avoid stalling, decode must be decoupled from execution
- Conceptually out-of-order issue decouples the decode/issue stage from instruction execution
  - it requires an **instruction window** between the decode and execute stages to buffer decoded or part pre-decoded instructions
  - this buffer serves as a pool of instructions giving the processor a look-ahead facility
  - instructions are issued from the buffer in any order, provided there are no resource conflicts or dependencies with executing instructions

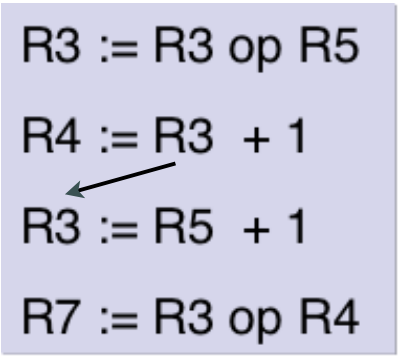
Out-of-order issue, out-of-order completion

- Again assume a processor issues two instructions, executes three instructions and writes back two results every cycle but now has a issue window of at least three instructions
  - I1 requires 2 cycles to execute
  - I3 and I4 are in conflict for a functional unit
  - I5 depends on the value produced by I4
  - I5 and I6 are in conflict for a functional unit



Anti-dependencies

- Out-of-order issue introduces yet another dependency - called an anti-dependency
  - the 3rd instruction can not be completed until the second instruction has read its operands
  - otherwise the 3rd instruction may overwrite the operand of the 2nd instruction
  - we say that the result of the 3rd instruction has an **anti-dependency** on the 1st operand of the 2nd instruction
  - this is like a true dependency but reversed



Summary of data hazards

- We have now have seen three kinds of dependencies
  - **True (data) dependencies** ... read after write (RAW)
  - **Output dependencies** ... write after write (WAW) - out of order completion
  - **Anti dependencies** ... write after read (WAR) - out of order issue
- Only true dependencies reflect the flow of data in a program and should require the pipeline to stall
  - when instructions are issued and completed out of order,
    the one-to-one relationship between registers and values at any given time is lost
  - new dependencies arise because registers hold different values from independent computations at different times – they are **resource dependencies**
- **Resource dependencies are really just storage conflicts** and can be eliminated by introducing new registers to re-establish the one-to-one relationship between registers and values at a given time