SuperScalar

Lokwon Kim

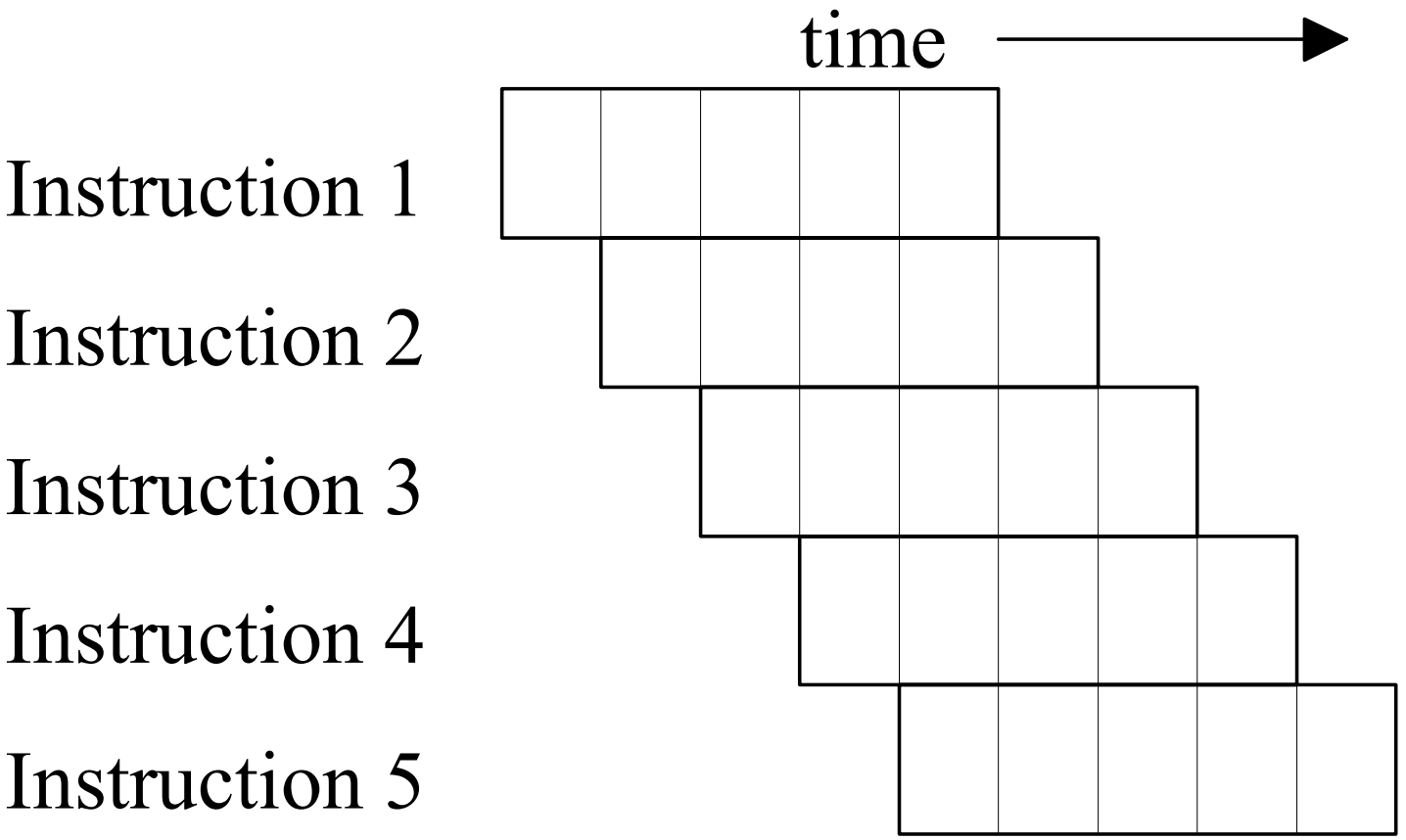
Extracting Yet *More* Performance

- Two options:
  - Increase the depth of the pipeline to increase the clock rate – **superpipelining** (more details to come)
  - Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – **superscalar** (multiple-issue)

- **Launching multiple instructions per stage allows the instruction execution rate, CPI, to be less than 1**

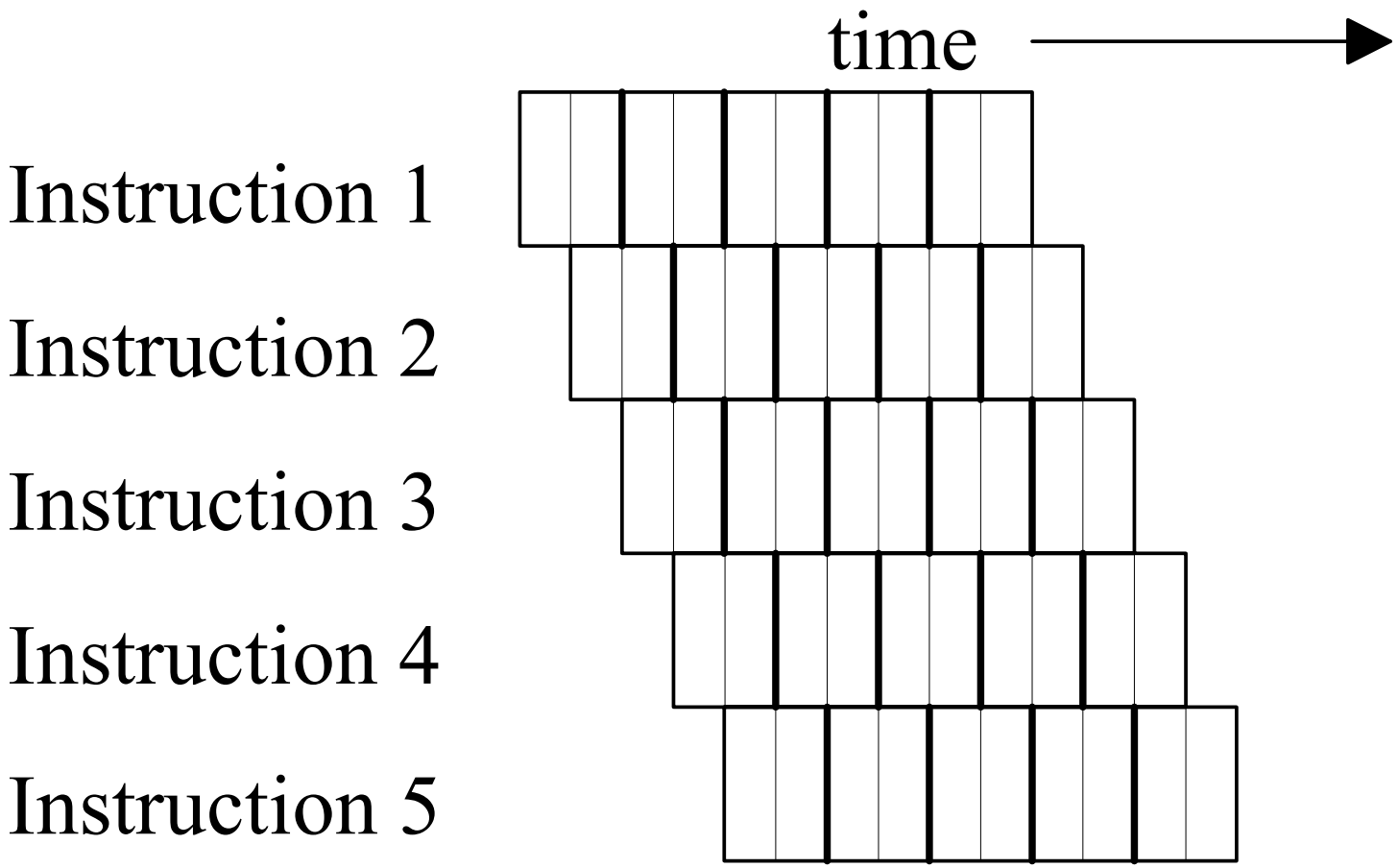
Instruction Pipelining

- pipelining – goal was to complete one instruction per clock cycle



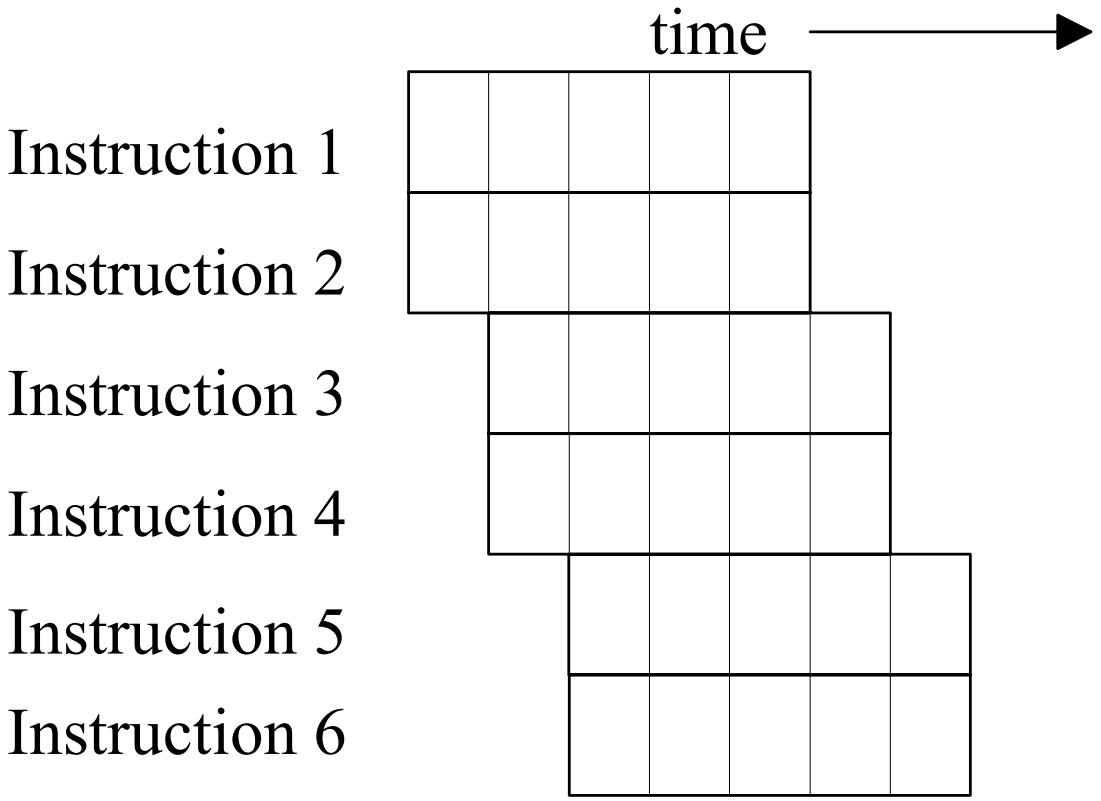
Superpipelining

- **superpipelining** -Increase the depth of the pipeline to increase the clock rate



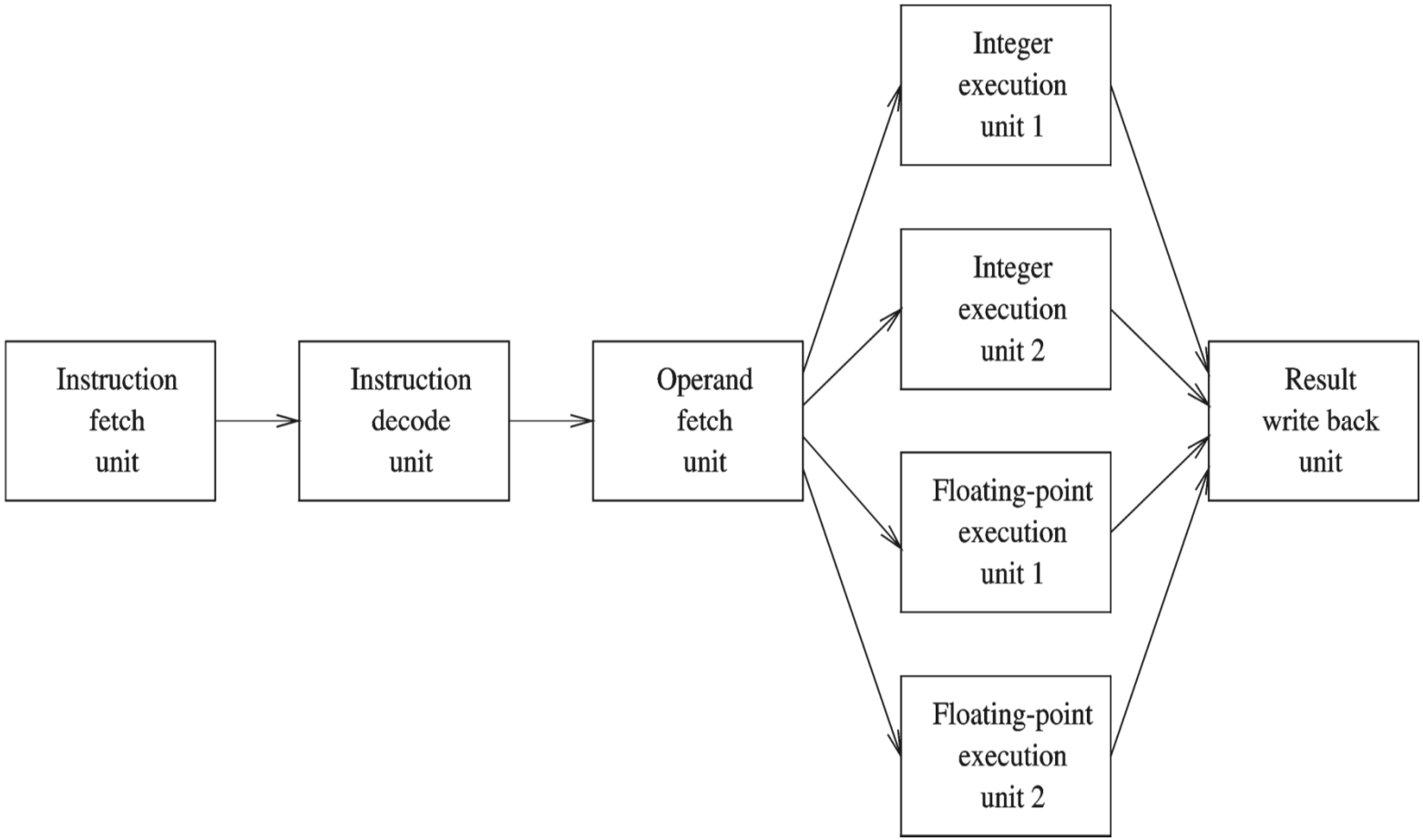
Superscalar – multiple-issue

- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions)



Superscalar – multiple-issue

- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions)



Superpipelined Processors

- Increase the depth of the pipeline leading to shorter clock cycles (and more instructions "in flight" at one time)
  - Higher degree of superpipelining, (1) more forwarding/hazard hardware needed, (2) even more inevitable pipeline stalls **(e.g. pipelined addition with data dependency), (3)** gain of superpipelining is not linear **(uneven balance of divided delay by superpipelining)**

Superpipelined vs Superscalar(SS)

- **Superpipelined processors have longer instruction latency than the SS processors which can degrade performance in the presence of true dependencies**
- **Superscalar processors are more susceptible to resource conflicts – but we can fix this with extra hardware !**

Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW – very long instruction word architecgture)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)

- **Dynamic multiple-issue processors** (aka **superscalar**)
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500

Multiple-Issue Datapath (or SS) Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Data dependencies – aka data hazards
    - Even more severe in SuperScalar (**Why?**).
  - Procedural dependencies – aka control hazards
    - Even more severe in SuperScalar (**Why?**).
    - Use dynamic branch prediction
  - Resource conflicts – aka structural hazards
    - Even more severe in SuperScalar (**Why?**).
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Out of Order (OoO) Execution

- **Instruction-issue** – initiate execution
  - Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- **Instruction-completion** – complete execution
  - Processor lookahead capability – complete issued instructions beyond the current instruction
- **Instruction-commit** – write back results to the RegFile or D$ (i.e., change the machine state)

In-order issue with in-order completion

In-order issue with out-of-order completion

Out-of-order issue with out-of-order completion

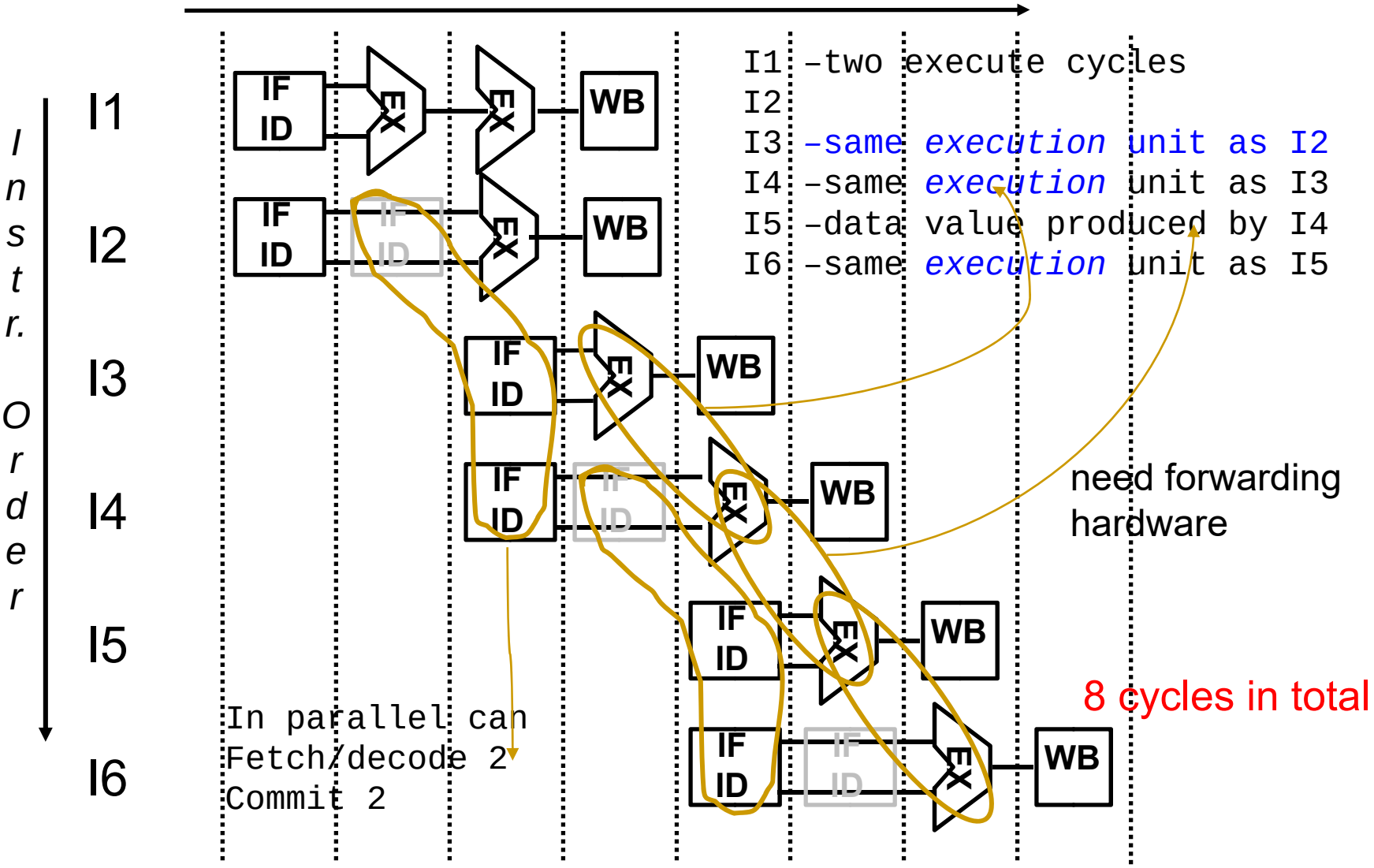
Out-of-order issue with out-of-order completion and in-order commit

In-Order Issue with In-Order Completion

- **Simplest policy is to issue instructions in exact program order and to complete them in the same order they were fetched** (i.e., **in program order**)
- **Example**:
  - Assume a pipelined processor (**3 stage pipeline**) that can fetch and decode **two** instructions per cycle, that has **three** functional units (a single cycle adder, a single cycle shifter, and a two cycle multiplier), and that can complete (and write back) **two** results per cycle
  - And an instruction sequence with the following characteristics

```
I1 – needs two execute cycles (a multiply)
I2
I3 – needs the same execution unit as I2
I4 – needs the same execution unit as I3
I5 – needs data value produced by I4
I6 – needs the same execution unit as I5
```

In-Order Issue, In-Order Completion Example

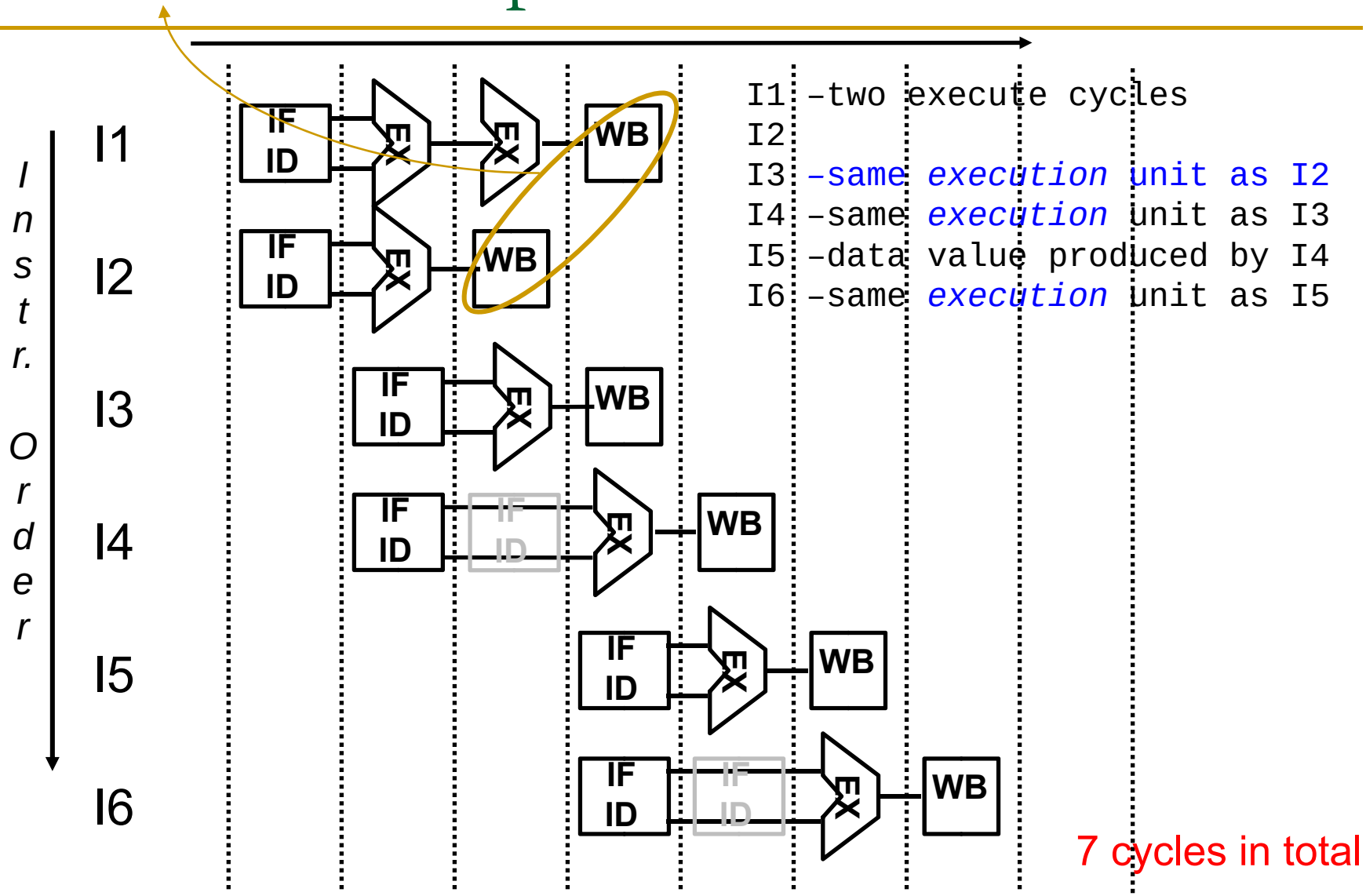


In-Order Issue with Out-of-Order Completion

- **With out-of-order completion, a later instruction may complete before a previous instruction**
  - Out-of-order completion is used in single-issue pipelined processors to improve the performance of long-latency operations such as divide

- When using out-of-order completion, instruction is **stalled** when there is a resource conflict (e.g., for a functional unit) or when the instructions ready to issue need a result that has not yet been computed

IOI-OOC Example



Handling Output Dependencies

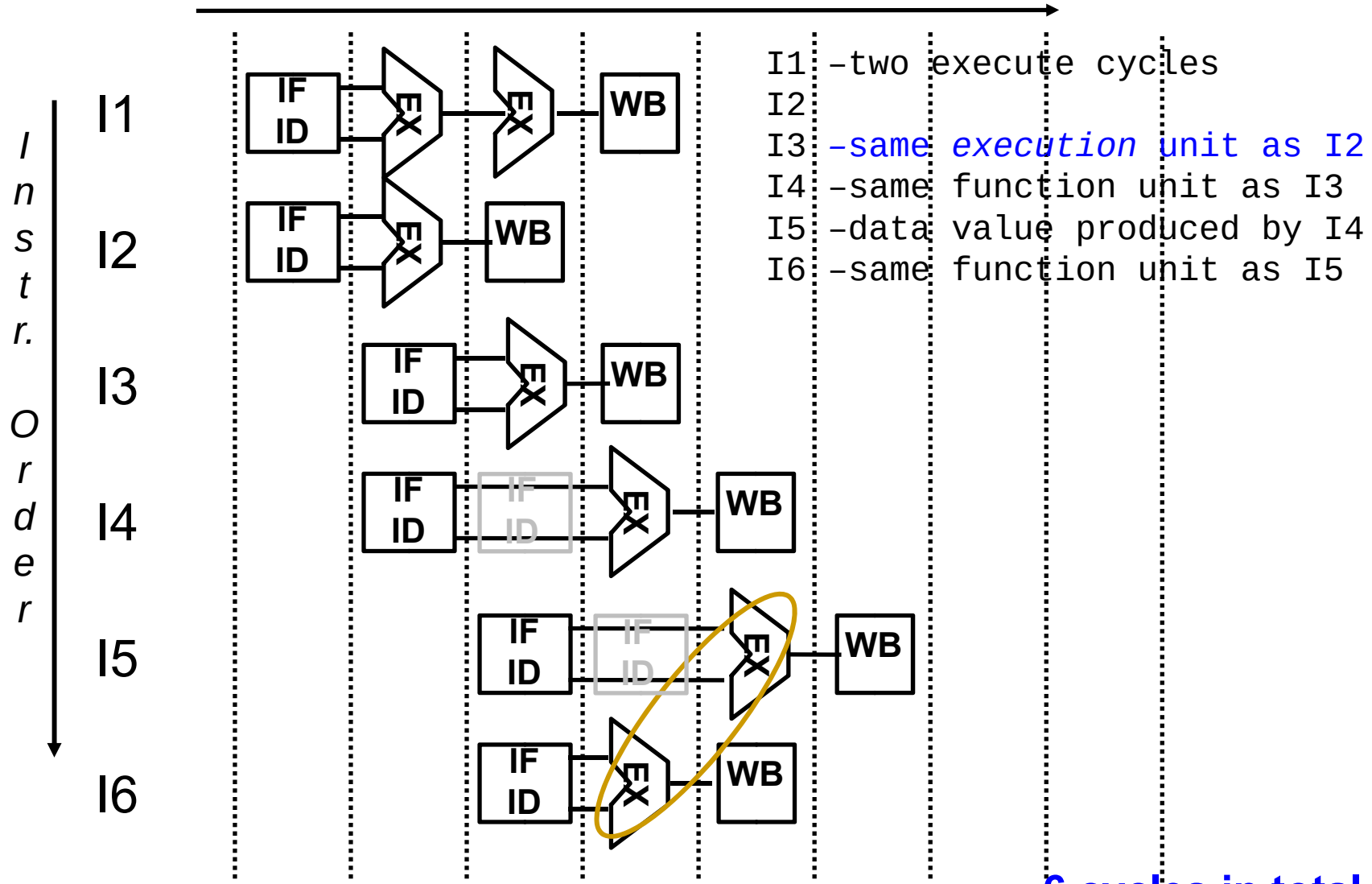
- There is one more situation that stalls instruction issuing with IOI-OOC, assume I1 – writes to R3
  I2 – writes to R3
  I5 – reads R3
  - **If the I1 write occurs after the I2 write, then I5 reads an incorrect value for R3**
  - I2 has an output dependency on I1 – write before write
    - The issuing of I2 would have to be stalled if its result might later be overwritten by an previous instruction (i.e., I1) that takes longer to complete – the stall happens before instruction issue

- **While IOI-OOC yields higher performance, it requires more dependency checking hardware**
  - Dependency checking needed to resolve both **read before write** *and* **write before write**

Out-of-Order Issue with Out-of-Order Completion

- With in-order issue the processor stops decoding instructions whenever a decoded instruction has a resource conflict or a data dependency on an issued, but uncompleted instruction
  - The processor is not able to *look beyond* the conflicted instruction even though more downstream instructions might have no conflicts and thus be issueable
- Fetch and decode instructions *beyond* the conflicted one, store them in an instruction buffer (as long as there's room), and flag those instructions in the buffer that don't have resource conflicts or data dependencies
- Flagged instructions are then issued from the buffer without regard to their program order
- **Issue means inserting instructions into Execution Unit.**

OOI-OOC Example



Antidependencies

- With OOI *also* have to deal with data antidependencies – when a later instruction (that completes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that issues later)

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

True data dependency
Output dependency
Antidependency

- The constraint is similar to that of true data dependencies, except *reversed*
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

Dependencies Review

- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  } storage conflicts

  manifests itself through the use of registers (or other storage locations)
- True dependencies represent the flow of data and information through a program
- **Anti- and output dependencies arise because the limited number of registers** mean that programmers reuse registers for different computations
- When instructions are issued out-of-order, the correspondence between registers and values breaks down and the values *conflict* for registers

Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors
- **Register renaming** – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * R5          R3b := R3 * R5
R4 := R3 + 1     ⇨     R4  := R3b + 1
R3 := R5 + 1           R3c := R5 + 1
```

Remove anti and output dependencies!

- The hardware that does renaming assigns a "replacement" register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

SuperScalar Summary

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Data dependencies – aka data hazards
    - Using OOI-OOC makes even more severe in SuperScalar (**Why?**).
  - Procedural dependencies – aka control hazards
    - Even more severe in SuperScalar (**Why?**).
    - Use dynamic branch prediction
  - Resource conflicts – aka structural hazards
    - Even more severe in SuperScalar (**Why?**).
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource