# Multiple Issue Introduction

[Adapted from Mary Jane Irwin for
*Computer Organization and Design*,
Patterson & Hennessy, © 2005, UCB]

# Review: Pipeline Hazards

- Structural hazards
  - Design pipeline to eliminate structural hazards
- Data hazards – read before write
  - Use data forwarding inside the pipeline
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream
- Control hazards – `beq`, `bne,j,jr,jal`
  - Stall – hurts performance
  - Move decision point as early in the pipeline as possible – reduces number of stalls at the cost of additional hardware
  - Delay decision (requires compiler support) – not feasible for deeper pipes requiring more than one delay slot to be filled
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (BHT) is correct and if the branched-to instruction is cached (BTB)

# Extracting Yet *More* Performance

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

- Launching multiple instructions per stage allows the instruction execution rate, CPI, to be less than 1
  - So instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

# Superpipelined Processors

- Increase the depth of the pipeline leading to shorter clock cycles (and more instructions "in flight" at one time)
  - The higher the degree of superpipelining, the more forwarding/ hazard hardware needed, the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time), and the bigger the clock skew issues (i.e., because of faster and faster clocks)

Superpipelined vs Superscalar

- 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 hardware !

# Instruction vs Machine Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of 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
- Data-level parallelism (DLP)

```
DO  I = 1  TO  100
   A[I] = A[I] + 1
CONTINUE
```

- Machine parallelism of a processor – a measure of 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
- To achieve high performance, need *both* ILP and machine parallelism

# Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW)
  - 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 Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

# Instruction Issue and Completion Policies

- 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

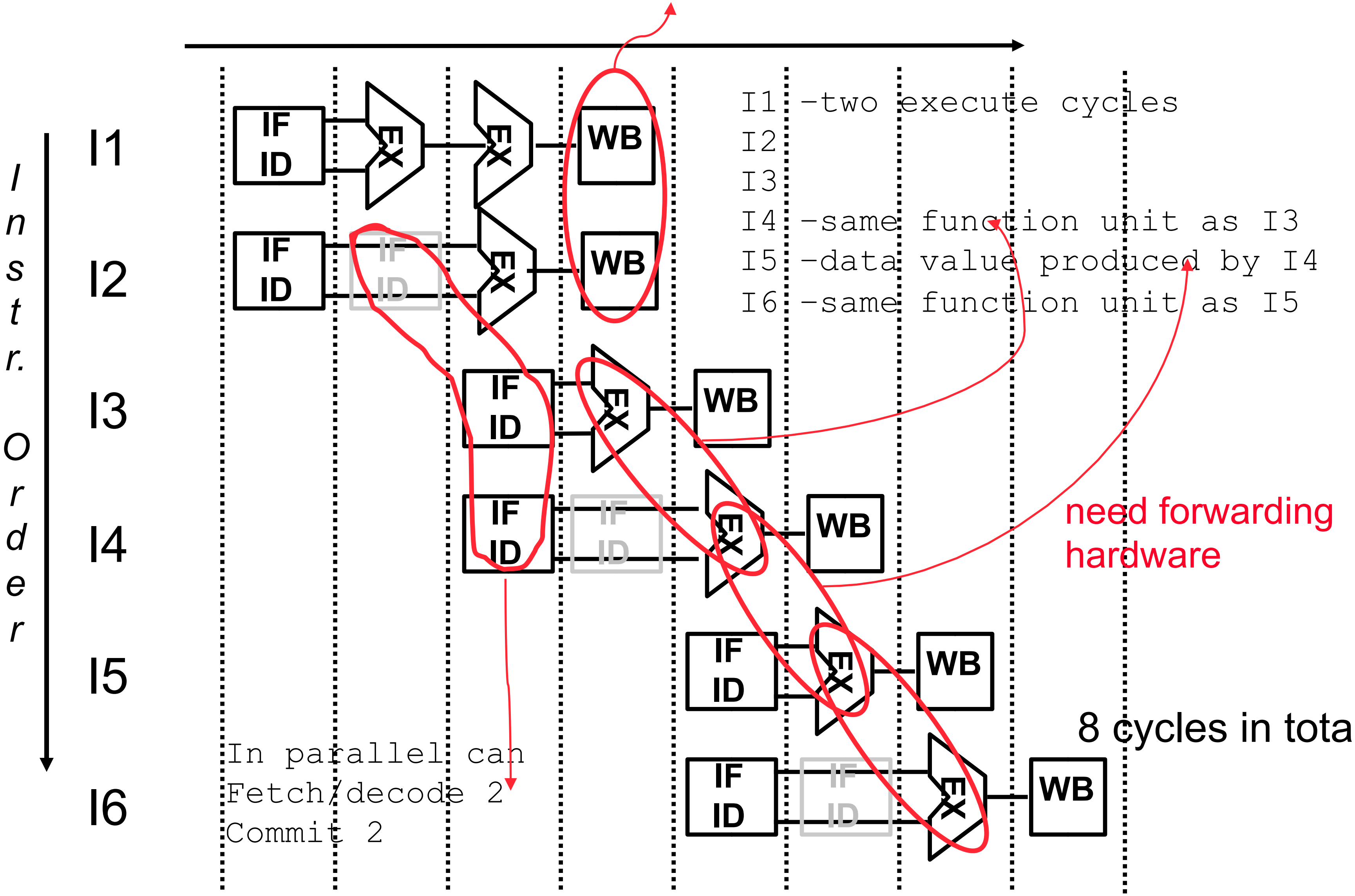
# In-Order Issue with In-Order Completion

- 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 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
I4 – needs the same function unit as I3
I5 – needs data value produced by I4
I6 – needs the same function unit as I5
```

# In-Order Issue, In-Order Completion Example

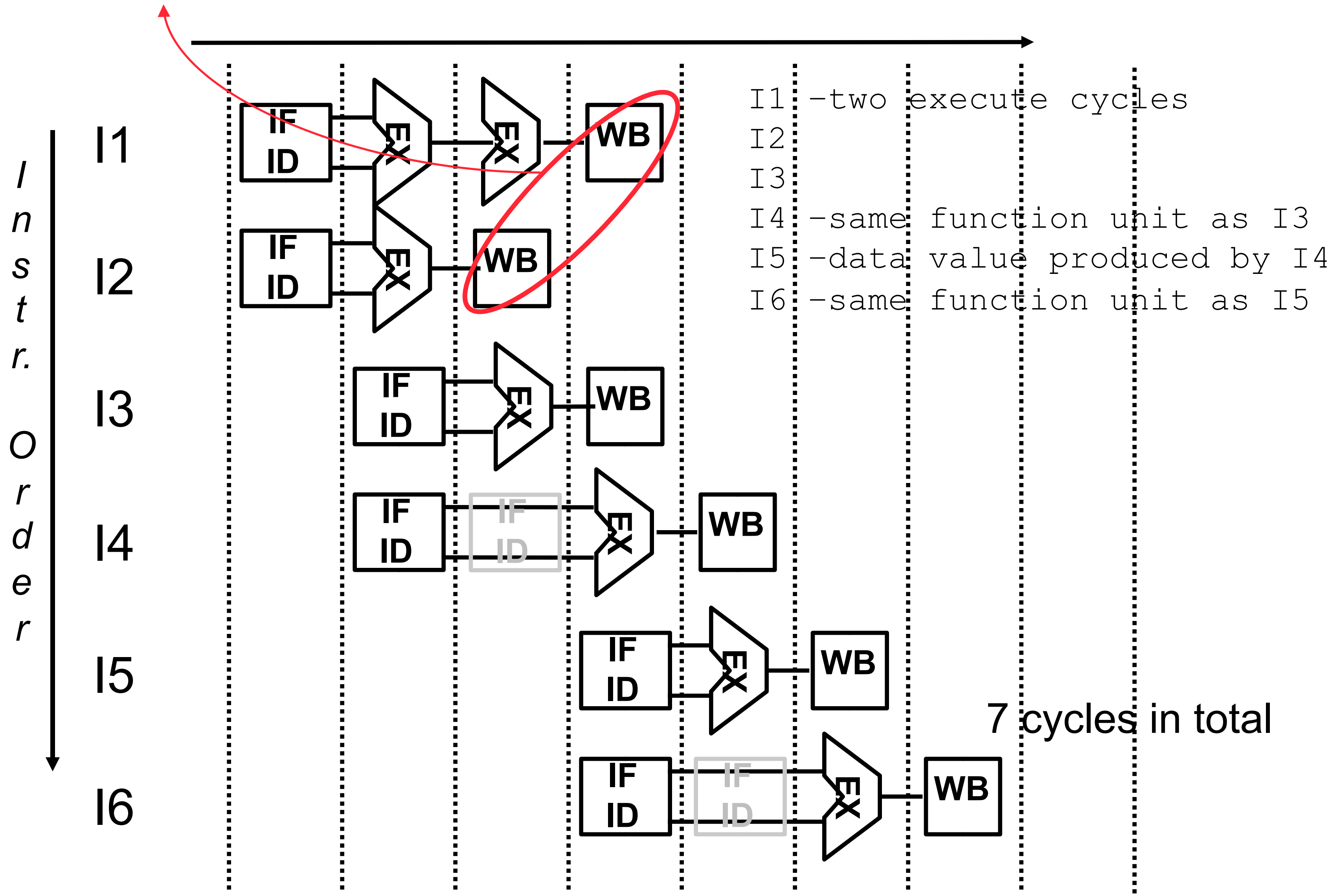


# In-Order Issue with Out-of-Order Completion

- 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 issue 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

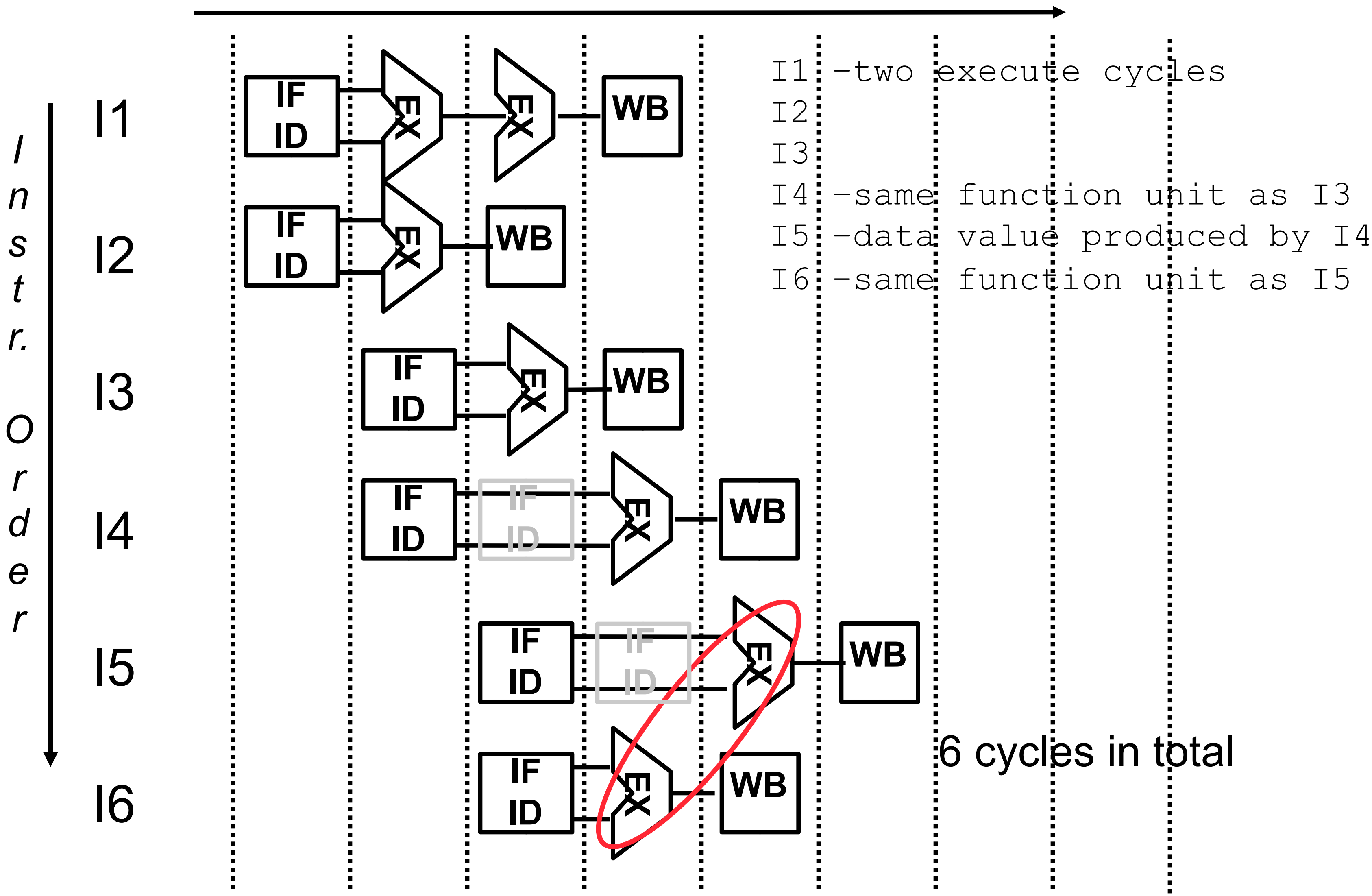
# Out-of-Order Issue with Out-of-Order Completion

- 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

# 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 := R3a * R5a
R4 := R3 + 1      ⇒      R4a := R3b + 1
R3 := R5 + 1             R3c := R5a + 1
```

  - 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