**Data hazards**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=3" \o "Edit section: Data hazards)]Figure 1

Data hazards occur when instructions that exhibit [data dependence](https://en.wikipedia.org/wiki/Data_dependence) modify data in different stages of a pipeline. Ignoring potential data hazards can result in [race conditions](https://en.wikipedia.org/wiki/Race_condition) (also termed race hazards). There are three situations in which a data hazard can occur:

1. read after write (RAW), a *true dependency*
2. write after read (WAR), an *anti-dependency*
3. write after write (WAW), an *output dependency*

Consider two instructions i1 and i2, with i1 occurring before i2 in program order.

**Read after write (RAW)**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=4" \o "Edit section: Read after write (RAW))]

(i2 tries to read a source before i1 writes to it) A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a prior instruction, the prior instruction has been processed only partly through the pipeline.

**Example**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=5" \o "Edit section: Example)]

For example:

Table

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

i1. **R2** <- R1 + R3  
i2. R4 <- **R2** + R3

The first instruction is calculating a value to be saved in register R2, and the second is going ijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkto use this value to compute a result for register R4. However, in a [pipeline](https://en.wikipedia.org/wiki/Pipeline_(computing)), when operands are fetched for the 2nd operation, the results from the first will not yet have been saved, and hence a data dependency occurs.

A data dependency occurs with instruction i2, as it is dependent on the completion of instruction i1.

**Write after read (WAR)**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=6" \o "Edit section: Write after read (WAR))]

(i2 tries to write a destination before it is read by i1) A write after read (WAR) data hazard represents a problem with concurrent execution.

**Example**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=7" \o "Edit section: Example)]

For example:ijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmkijoiuoiojkokljlkjkjjnkjhnkjhkjhijkjhkljlkjlkjlkmlk,mmk

i1. R4 <- R1 + **R5**  
i2. **R5** <- R1 + R2

In any situation with a chance that i2 may finish before i1 (i.e., with concurrent execution), it must be ensured that the result of register R5 is not stored before i1 has had a chance to fetch the operands.

**Write after write (WAW)**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=8" \o "Edit section: Write after write (WAW))]

(i2 tries to write an operand before it is written by i1) A write after write (WAW) data hazard may occur in a [concurrent execution](https://en.wikipedia.org/wiki/Concurrent_computing) environment.

**Example**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=9" \o "Edit section: Example)]

For example:

i1. **R2** <- R4 + R7  
i2. **R2** <- R1 + R3

The write back (WB) of i2 must be delayed until i1 finishes executing.

**Structural hazards**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=10" \o "Edit section: Structural hazards)]

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory.[[3]](https://en.wikipedia.org/wiki/Hazard_(computer_architecture)#cite_note-FOOTNOTEPattersonHennessy2009336-3) They can often be resolved by separating the component into [orthogonal](https://en.wikipedia.org/wiki/Orthogonal) units (such as separate caches) or [bubbling the pipeline](https://en.wikipedia.org/wiki/Bubbling_the_pipeline).

**Control hazards (branch hazards)**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=11" \o "Edit section: Control hazards (branch hazards))]

*Further information:*[*Branch (computer science)*](https://en.wikipedia.org/wiki/Branch_(computer_science))

Branching hazards (also termed control hazards) occur with [branches](https://en.wikipedia.org/wiki/Branch_(computer_science)). On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the *fetch* stage).

Eliminating hazards[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=12" \o "Edit section: Eliminating hazards)]

**Generic**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=13" \o "Edit section: Generic)]

**Pipeline bubbling**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=14" \o "Edit section: Pipeline bubbling)]

*Main article:*[*Bubble (computing)*](https://en.wikipedia.org/wiki/Bubble_(computing))

*Bubbling the pipeline*, also termed a *pipeline break* or *pipeline stall*, is a method to preclude data, structural, and branch hazards. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts no operations ([NOP](https://en.wikipedia.org/wiki/NOP)s) into the pipeline. Thus, before the next instruction (which would cause the hazard) executes, the prior one will have had sufficient time to finish and prevent the hazard. If the number of NOPs equals the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. All forms of stalling introduce a delay before the processor can resume execution.

*Flushing the pipeline* occurs when a branch instruction jumps to a new memory location, invalidating all prior stages in the pipeline. These prior stages are cleared, allowing the pipeline to continue at the new instruction indicated by the branch.[[4]](https://en.wikipedia.org/wiki/Hazard_(computer_architecture)#cite_note-4)[[5]](https://en.wikipedia.org/wiki/Hazard_(computer_architecture)#cite_note-5)

**Data hazards**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=15" \o "Edit section: Data hazards)]

There are several main solutions and algorithms used to resolve data hazards:

* insert a *pipeline bubble* whenever a read after write (RAW) dependency is encountered, guaranteed to increase latency, or
* use [out-of-order execution](https://en.wikipedia.org/wiki/Out-of-order_execution) to potentially prevent the need for pipeline bubbles
* use [*operand forwarding*](https://en.wikipedia.org/wiki/Operand_forwarding) to use data from later stages in the pipeline

In the case of [out-of-order execution](https://en.wikipedia.org/wiki/Out-of-order_execution), the algorithm used can be:

* [scoreboarding](https://en.wikipedia.org/wiki/Scoreboarding), in which case a *pipeline bubble* is needed only when there is no functional unit available
* the [Tomasulo algorithm](https://en.wikipedia.org/wiki/Tomasulo_algorithm" \o "Tomasulo algorithm), which uses [register renaming](https://en.wikipedia.org/wiki/Register_renaming), allowing continual issuing of instructions

The task of removing data dependencies can be delegated to the compiler, which can fill in an appropriate number of NOP instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

**Operand forwarding**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=16" \o "Edit section: Operand forwarding)]

*Main article:*[*Operand forwarding*](https://en.wikipedia.org/wiki/Operand_forwarding)

**Examples**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=17" \o "Edit section: Examples)]

*In the following examples, computed values are in****bold****, while Register numbers are not.*

For example, to write the value 3 to register 1, (which already contains a 6), and then add 7 to register 1 and store the result in register 2, i.e.:

*Instruction 0: Register 1 =****6***

*Instruction 1: Register 1 =****3***

*Instruction 2: Register 2 = Register 1 +****7****=****10***

Following execution, register 2 should contain the value **10**. However, if Instruction 1 (write **3** to register 1) does not fully exit the pipeline before Instruction 2 starts executing, it means that Register 1 does not contain the value **3** when Instruction 2 performs its addition. In such an event, Instruction 2 adds **7** to the old value of register 1 (**6**), and so register 2 contains **13** instead, i.e.:

*Instruction 0: Register 1 =****6***

*Instruction 2: Register 2 = Register 1 +****7****=****13***

*Instruction 1: Register 1 =****3***

This error occurs because Instruction 2 reads Register 1 before Instruction 1 has committed/stored the result of its write operation to Register 1. So when Instruction 2 is reading the contents of Register 1, register 1 still contains **6**, *not* **3**.

Forwarding (described below) helps correct such errors by depending on the fact that the output of Instruction 1 (which is **3**) can be used by subsequent instructions *before* the value **3** is committed to/stored in Register 1.

Forwarding applied to the example means that *there is no wait to commit/store the output of Instruction 1 in Register 1 (in this example, the output is****3****) before making that output available to the subsequent instruction (in this case, Instruction 2).* The effect is that Instruction 2 uses the correct (the more recent) value of Register 1: the commit/store was made immediately and not pipelined.

With forwarding enabled, the Instruction Decode/Execution (ID/EX) stage of the pipeline now has two inputs: the value read from the register specified (in this example, the value **6** from Register 1), and the new value of Register 1 (in this example, this value is **3**) which is sent from the next stage Instruction Execute/Memory Access (EX/MEM). Added control logic is used to determine which input to use.

**Control hazards (branch hazards)**[[edit](https://en.wikipedia.org/w/index.php?title=Hazard_(computer_architecture)&action=edit&section=18" \o "Edit section: Control hazards (branch hazards))]

To avoid control hazards microarchitectures can:

* insert a *pipeline bubble* (discussed above), guaranteed to increase [latency](https://en.wikipedia.org/wiki/Latency_(engineering)), or
* use [branch prediction](https://en.wikipedia.org/wiki/Branch_prediction) and essentially make educated guesses about which instructions to insert, in which case a *pipeline bubble* will only be needed in the case of an incorrect prediction

In the event that a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly-loaded instructions from having any effect on the processor state excluding energy wasted processing them before they were discovered to be loaded incorrectly.

**Operand forwarding** (or **data forwarding**) is an optimization in pipelined [CPUs](https://en.wikipedia.org/wiki/CPU) to limit performance deficits which occur due to [pipeline stalls](https://en.wikipedia.org/wiki/Pipeline_stall).[[1]](https://en.wikipedia.org/wiki/Operand_forwarding#cite_note-1)[[2]](https://en.wikipedia.org/wiki/Operand_forwarding#cite_note-2) A [data hazard](https://en.wikipedia.org/wiki/Data_hazard) can lead to a [pipeline stall](https://en.wikipedia.org/wiki/Pipeline_stall) when the current operation has to wait for the results of an earlier operation which has not yet finished.

ADD A B C #A=B+C

SUB D C A #D=C-A

If these two [assembly](https://en.wikipedia.org/wiki/Assembly_language) pseudocode instructions run in a pipeline, after fetching and decoding the second instruction, the pipeline stalls, waiting until the result of the addition is written and read.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Without operand forwarding** | | | | | | | |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| Fetch ADD | Decode ADD | Read Operands ADD | Execute ADD | Write result |  |  |  |
|  | Fetch SUB | Decode SUB | *stall* | *stall* | Read Operands SUB | Execute SUB | Write result |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **With operand forwarding** | | | | | |
|  | | | | | |
| **1** | **2** | **3** | **4** | **5** | **6** |
| Fetch ADD | Decode ADD | Read Operands ADD | Execute ADD | Write result |  |
|  | Fetch SUB | Decode SUB | Read Operands SUB: use result from previous operation | Execute SUB | Write result |

, **register renaming**

In [computer architecture](https://en.wikipedia.org/wiki/Computer_architecture), **register renaming** is a technique that eliminates the false [data dependencies](https://en.wikipedia.org/wiki/Data_dependency) arising from the reuse of [architectural registers](https://en.wikipedia.org/wiki/Processor_register) by successive [instructions](https://en.wikipedia.org/wiki/Instruction_(computer_science)" \o "Instruction (computer science))that do not have any real data dependencies between them. The elimination of these false data dependencies reveals more [instruction-level parallelism](https://en.wikipedia.org/wiki/Instruction-level_parallelism) in an instruction stream, which can be exploited by various and complementary techniques such as [superscalar](https://en.wikipedia.org/wiki/Superscalar) and [out-of-order](https://en.wikipedia.org/wiki/Out-of-order_execution) execution for better [performance](https://en.wikipedia.org/wiki/Computer_performance).

n a [register machine](https://en.wikipedia.org/wiki/Register_machine), programs are composed of instructions which operate on values. The instructions must name these values in order to distinguish them from one another. A typical instruction might say, add X and Y and put the result in Z. In this instruction, X, Y, and Z are the names of storage locations.

In order to have a compact instruction encoding, most processor instruction sets have a small set of special locations which can be directly named. For example, the x86 instruction set architecture has 8 integer registers, x86-64 has 16, many RISCs have 32, and IA-64 has 128. In smaller processors, the names of these locations correspond directly to elements of a [register file](https://en.wikipedia.org/wiki/Register_file).

Different instructions may take different amounts of time; for example, a processor may be able to execute hundreds of instructions while a single load from the main memory is in progress. Shorter instructions executed while the load is outstanding will finish first, thus the instructions are finishing out of the original program order. [Out-of-order execution](https://en.wikipedia.org/wiki/Out-of-order_execution) has been used in most recent high-performance CPUs to achieve some of their speed gains.

Consider this piece of code running on an out-of-order CPU:

|  |  |
| --- | --- |
| **#** | **Instruction** |
| **1** | R1 = M[1024] |
| **2** | R1 = R1 + 2 |
| **3** | M[1032] = R1 |
| **4** | R1 = M[2048] |
| **5** | R1 = R1 + 4 |
| **6** | M[2056] = R1 |

Instructions 4, 5, and 6 are independent of instructions 1, 2, and 3, but the processor cannot finish 4 until 3 is done, otherwise instruction 3 would write the wrong value. This restriction can be eliminated by changing the names of some of the registers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Instruction** |  | **#** | **Instruction** |
| **1** | R1 = M[1024] | **4** | R2 = M[2048] |
| **2** | R1 = R1 + 2 | **5** | R2 = R2 + 4 |
| **3** | M[1032] = R1 | **6** | M[2056] = R2 |

Now instructions 4, 5, and 6 can be executed in parallel with instructions 1, 2, and 3, so that the program can be executed faster.

When possible, the compiler would detect the distinct instructions and try to assign them to a different register. However, there is a finite number of register names that can be used in the assembly code. Many high performance CPUs have more physical registers than may be named directly in the instruction set, so they rename registers in hardware to achieve additional parallelism.

Figure

Table 1