Hazard (computer architecture)

In the domain of central processing unit (CPU) design, **hazards** are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle,[1] and can potentially lead to incorrect computation results. Three common types of hazards are data hazards, structural hazards, and control flow hazards (branching hazards).[2]

There are several methods used to deal with hazards, including pipeline stalls/pipeline bubbling, operand forwarding, and in the case of out-of-order execution, the scoreboarding method and the Tomasulo algorithm.

1 Background

Further information: Instruction pipeline

Instructions in a pipelined processor are performed in several stages, so that at any given time several instructions are being processed in the various stages of the pipeline, such as fetch and execute. There are many different instruction pipeline microarchitectures, and instructions may be executed out-of-order. A hazard occurs when two or more of these simultaneous (possibly out of order) instructions conflict.

2 Types

2.1 Data hazards

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (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.

2.1.1 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 For example:

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 to use this value to compute a result for register R4. However, in a pipeline, 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.

2.1.2 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 For example:

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.

2.1.3 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 environment.

Example For example:

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

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

2.2 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] They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.

2.3 Control hazards (branch hazards)

Further information: Branch (computer science)

Branching hazards (also termed control hazards) occur with branches. 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).

3 Eliminating hazards

3.1 Generic

3.1.1 Pipeline bubbling

Main article: 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 (NOPs) 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][5]

3.2 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 to potentially prevent the need for pipeline bubbles
- use *operand forwarding* to use data from later stages in the pipeline

In the case of out-of-order execution, the algorithm used can be:

- scoreboarding, in which case a *pipeline bubble* is needed only when there is no functional unit available
- the Tomasulo algorithm, which uses 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.

3.2.1 Operand forwarding

Main article: Operand forwarding

3.2.2 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.

3.3 Control hazards (branch hazards)

To avoid control hazards microarchitectures can:

- insert a *pipeline bubble* (discussed above), guaranteed to increase latency, or
- use 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.

3.4 Other techniques

Memory latency is another factor that designers must attend to, because the delay could reduce performance. Different types of memory have different accessing time to the memory. Thus, by choosing a suitable type of memory, designers can improve the performance of the pipelined data path.[6]

4 See also

- Feed forward (control)
- Register renaming
- Data dependency
- Hazard (logic)
- Hazard pointer
- Classic RISC pipeline § Hazards
- Speculative execution
- Branch delay slot
- Branch predication
- Branch predictor

5 References

[1] Patterson & Hennessy 2009, p. 335.

[2] Patterson & Hennessy 2009, pp. 335-343.

[3] Patterson & Hennessy 2009, p. 336.

[4] "Branch Prediction Schemes". *cs.iastate.edu*. 2001-04-06. Retrieved 2014-07-19.

[5] "Data and Control Hazards". *classes.soe.ucsc.edu*. 2004-02-23. Retrieved 2014-07-19.

[6] "Design Example of Useful Memory Latency for Developing a Hazard Preventive Pipeline High-Performance Embedded-Microprocessor". *hindawi.com*. 2012-12-27. Retrieved 2014-07-29.

- Patterson, David; Hennessy, John (2009). *Computer Organization and Design* (4th ed.). Morgan Kaufmann. ISBN 978-0-12-374493-7.
- Patterson, David; Hennessy, John (2011). *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann. ISBN 978-0-12-383872-8.
- John P. Shen and Mikko H. Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors, 2004, ISBN 0070570647

6 External links

- "Automatic Pipelining from Transactional Datapath Specifications" (PDF). Retrieved 23 July 2014.
- Pipeline hazards, January 18, 2005, by Dean Tulsen

7 Text and image sources, contributors, and licenses

7.1 Text

- **Hazard (computer architecture)** *Source:* https://en.wikipedia.org/wiki/Hazard_(computer_architecture)?oldid=745693932 *Contributors:* Derek Ross, William Avery, CesarB, Dcoetzee, Morwen, Raul654, DavidCary, Kareeser, Brookie, MarcoTolo, JIP, Ligulem, Nihiltres, Ewlyahoocom, Intgr, Fresheneesz, Anrie Nord, Rsrikanth05, Długosz, Allens, RG2, SmackBot, Teimu.tm, Chris the speller, TimBentley, Nbarth, Colonies Chris, Meitme, Rante~enwiki, JonHarder, Henning Makholm, A5b, Plaicy, Dgessner, 16@r, Drae, DJGB, Jesse Viviano, Christian75, Thijs!bot, WinBot, Widefox, MartinBot, KylieTastic, STBotD, Egg D, Abatishchev, Jerryobject, Int21h, Seungbaeim, Arjayay, SchreiberBike, Sleepinj, Dsimic, Addbot, Yobot, Materialscientist, Spike-from-NH, Nexus26, Surya iriventi, Wasted Oompa-Loompa, Chronulator, Suffusion of Yellow, Tbhotch, ZéroBot, Alpha Quadrant (alt), Gz33, ClueBot NG, West.andrew.g.norb, Adysts, Ramsterzao, Comments2010, Edward1819, The Holm, B2u9vm1ea8g and Anonymous: 89

7.2 Images

- **File:Question_book-new.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* Cc-by-sa-3.0 *Contributors:*
  Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:*
  Tkgd2007

7.3 Content license

- Creative Commons Attribution-Share Alike 3.0