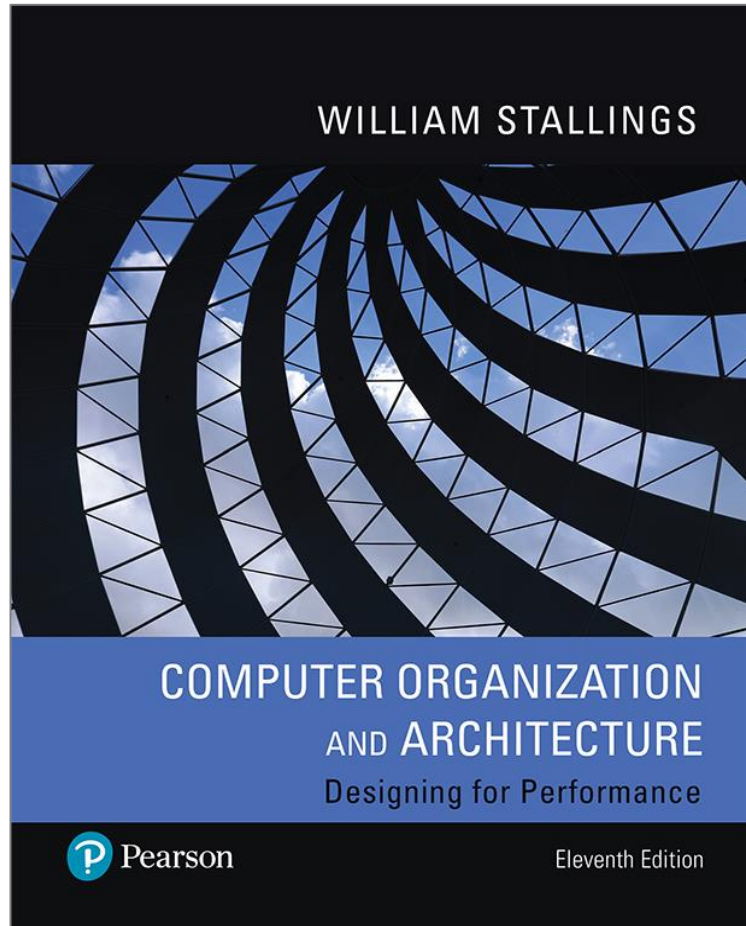


Computer Organization and Architecture

Designing for Performance

11th Edition



Chapter 16

Processor Structure and Function

Pipeline Hazards

Occur when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution

There are three types of hazards:

- Resource
- Data
- Control

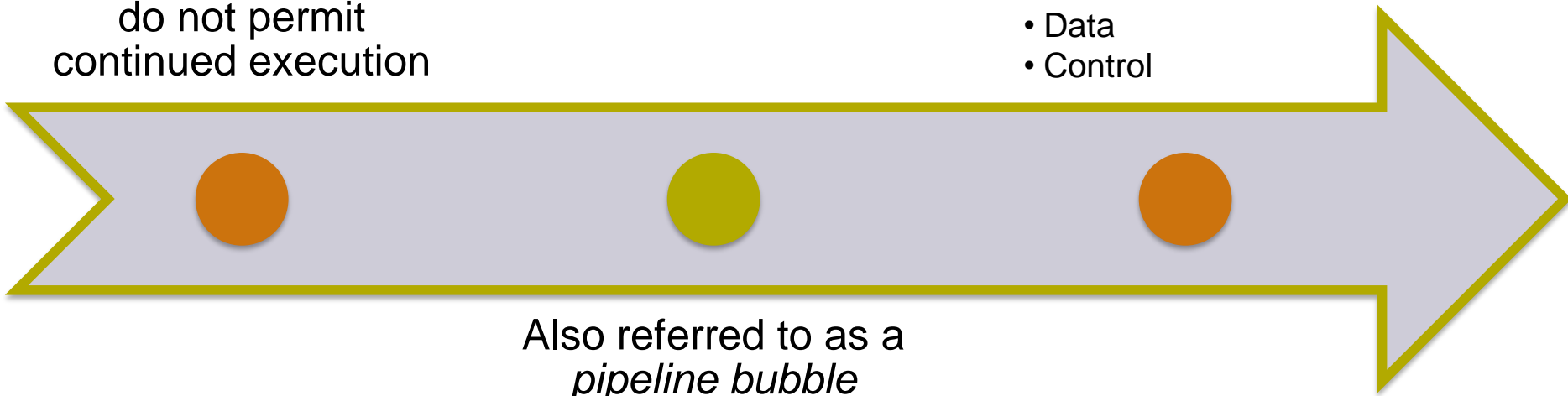


Figure 16.15

Example of Resource Hazard

| | | Clock cycle | | | | | | | | |
|-------------|----|-------------|----|----|----|----|----|----|----|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Instrutcion | I1 | FI | DI | FO | EI | WO | | | | |
| | I2 | | FI | DI | FO | EI | WO | | | |
| | I3 | | | FI | DI | FO | EI | WO | | |
| | I4 | | | | FI | DI | FO | EI | WO | |

(a) Five-stage pipeline, ideal case

| | | Clock cycle | | | | | | | | |
|-------------|----|-------------|----|------|----|----|----|----|----|----|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Instrutcion | I1 | FI | DI | FO | EI | WO | | | | |
| | I2 | | FI | DI | FO | EI | WO | | | |
| | I3 | | | Idle | FI | DI | FO | EI | WO | |
| | I4 | | | | | FI | DI | FO | EI | WO |

(b) I1 source operand in memory

Resource Hazard

- Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Figure 16.15b, assumes that the source operand for instruction I1 is in memory.
- Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3.
- Another situation where multiple instructions need to use the ALU unit.
- Possible solution? have multiple memory ports and ALU units?

Figure 16.16

Example of Data Hazard

| | | Clock cycle | | | | | | | | | |
|--------------|--------------|-------------|----|----|------|----|----|----|----|----|----|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| ADD EAX, EBX | | FI | DI | FO | EI | WO | | | | | |
| | SUB ECX, EAX | | FI | DI | Idle | | FO | EI | WO | | |
| | I3 | | | FI | | | DI | FO | EI | WO | |
| | I4 | | | | | | FI | DI | FO | EI | WO |

Data Hazard

- The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX.
- The second instruction subtracts the contents of EAX from ECX and stores the result in ECX.
- The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5.
- But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clock cycles.

Types of Data Hazard

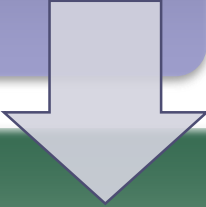
- Read after write (RAW), or true dependency
 - An instruction modifies a register or memory location
 - Succeeding instruction reads data in memory or register location
 - Hazard occurs if the read takes place before write operation is complete
- Write after read (WAR), or antidependency
 - An instruction reads a register or memory location
 - Succeeding instruction writes to the location
 - Hazard occurs if the write operation completes before the read operation takes place
- Write after write (WAW), or output dependency
 - Two instructions both write to the same location
 - Hazard occurs if the write operations take place in the reverse order of the intended sequence

Control Hazard

- Also known as a *branch hazard*
- Occurs when the pipeline makes the wrong decision on a branch prediction
- Brings instructions into the pipeline that must subsequently be discarded
- Dealing with Branches:
 - Multiple streams
 - Prefetch branch target
 - Loop buffer
 - Branch prediction
 - Delayed branch

Multiple Streams

A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice



A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams



Drawbacks:

- With multiple pipelines there are contention delays for access to the registers and to memory
- Additional branch instructions may enter the pipeline before the original branch decision is resolved

Prefetch Branch Target

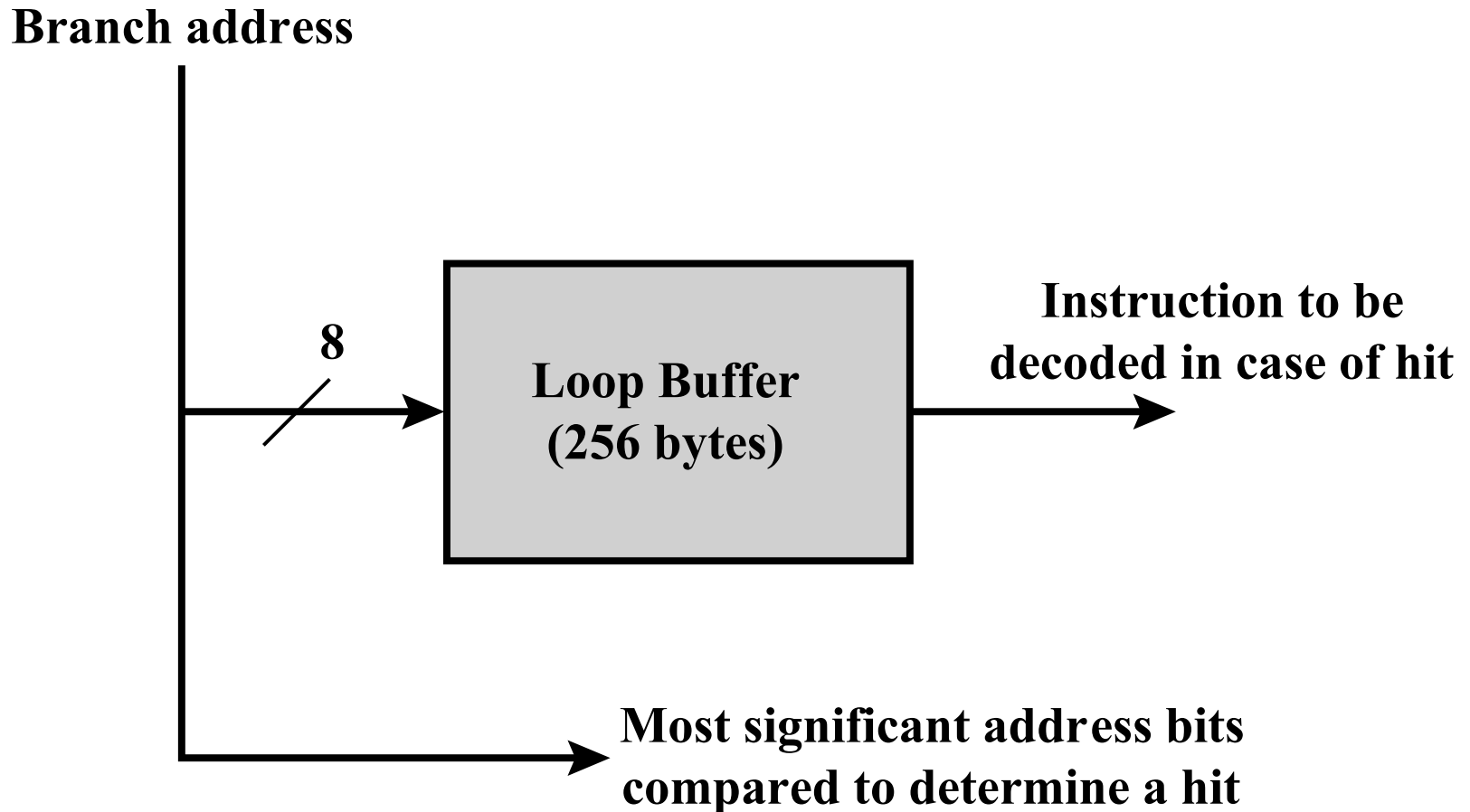
- When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch
- Target is then saved until the branch instruction is executed
- If the branch is taken, the target has already been prefetched
- IBM 360/91 uses this approach

Loop Buffer

- Small, very-high speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence
- Benefits:
 - Instructions fetched in sequence will be available without the usual memory access time
 - If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer
 - This strategy is particularly well suited to dealing with loops
- Similar in principle to a cache dedicated to instructions
 - Differences:
 - The loop buffer only retains instructions in sequence
 - Is much smaller in size and hence lower in cost

Figure 16.17

Loop Buffer



Branch Prediction

- Various techniques can be used to predict whether a branch will be taken:

1. Predict never taken
2. Predict always taken
3. Predict by opcode

- These approaches are static
- They do not depend on the execution history up to the time of the conditional branch instruction

1. Taken/not taken switch
2. Branch history table

- These approaches are dynamic
- They depend on the execution history

Static Strategies

- Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88], then always prefetching from the branch target address should give better performance than always prefetching from the sequential path.
- However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in sequence, and so this performance penalty should be taken into account.

Static Strategies

- The final static approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. [LILJ88] reports success rates of greater than 75% with this strategy.

Dynamic Strategies

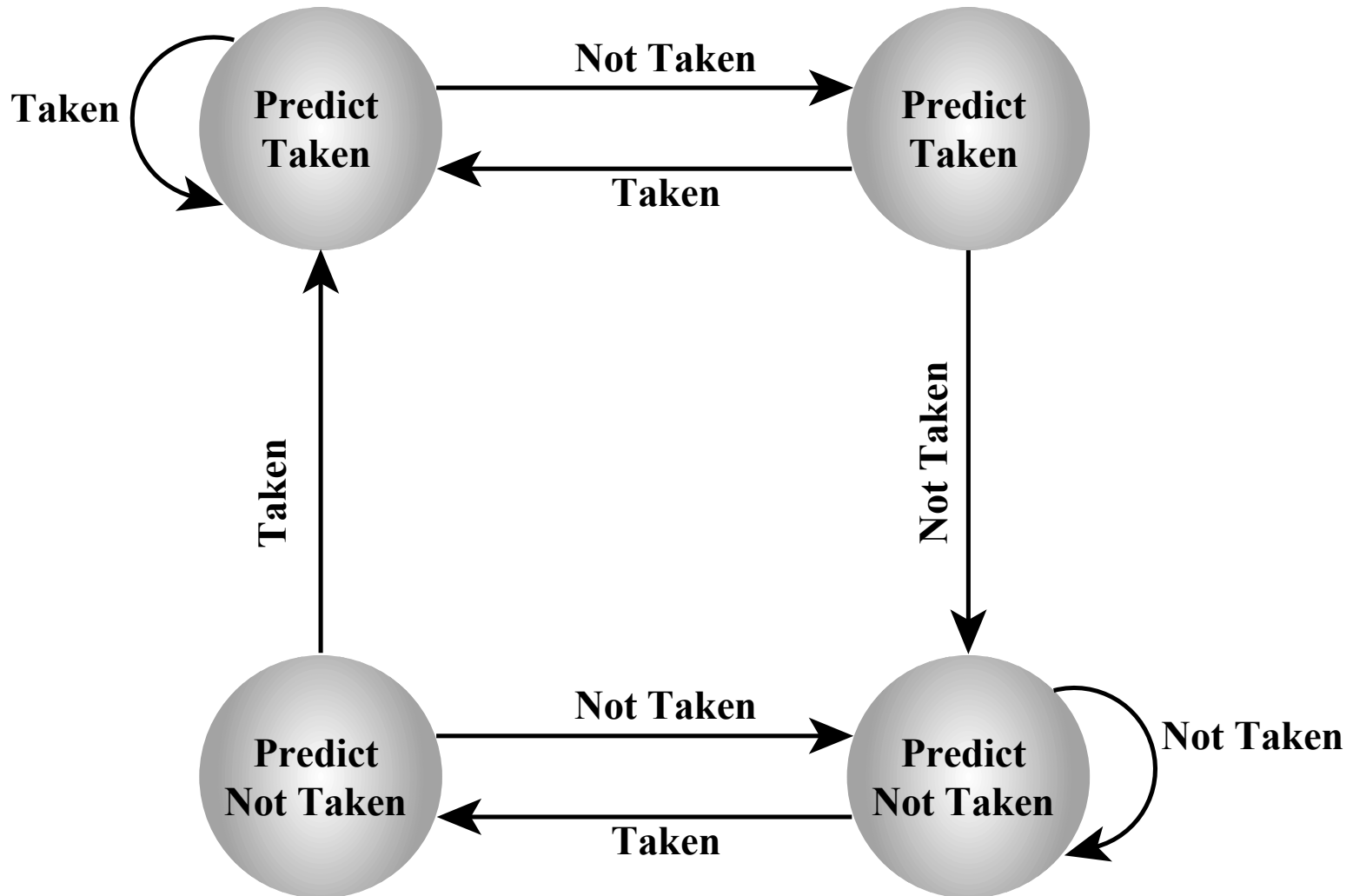
- Dynamic branch strategies attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program.
- For example, one or more bits can be associated with each conditional branch instruction that reflect the recent history of the instruction.
- These bits are referred to as a taken/ not taken switch that directs the processor to make a particular decision the next time the instruction is encountered.

Dynamic Strategies

- One possibility is to associate these bits with any conditional branch instruction that is in a cache.
- Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry.
- The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction's address.

Figure 16.19

Branch Prediction State Diagram



Taken/not taken switch

- As long as each succeeding conditional branch instruction that is encountered is taken, the decision process predicts that the next branch will be taken.
- If a single prediction is wrong, the algorithm continues to predict that the next branch is taken.
- Only if two successive branches are not taken does the algorithm predict that branches are not taken until two branches in a row are taken.
- Thus, the algorithm requires two consecutive wrong predictions to change the prediction decision.

Branch History Table

- The use of history bits has one drawback: If the decision is made to take the branch, the target instruction cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded.
- More information must be saved, in what is known as a **branch target buffer, or a branch history table**.
- The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline.

Branch History Table

- Each entry in the table consists of three elements: the address of a branch instruction, some number of history bits that record the state of use of that instruction, and the address of the target instruction.

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructions in teaching their courses and assessing student learning. dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.