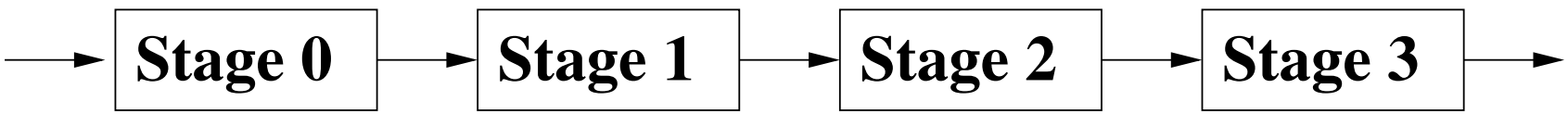
Pipelining: Basic Concepts

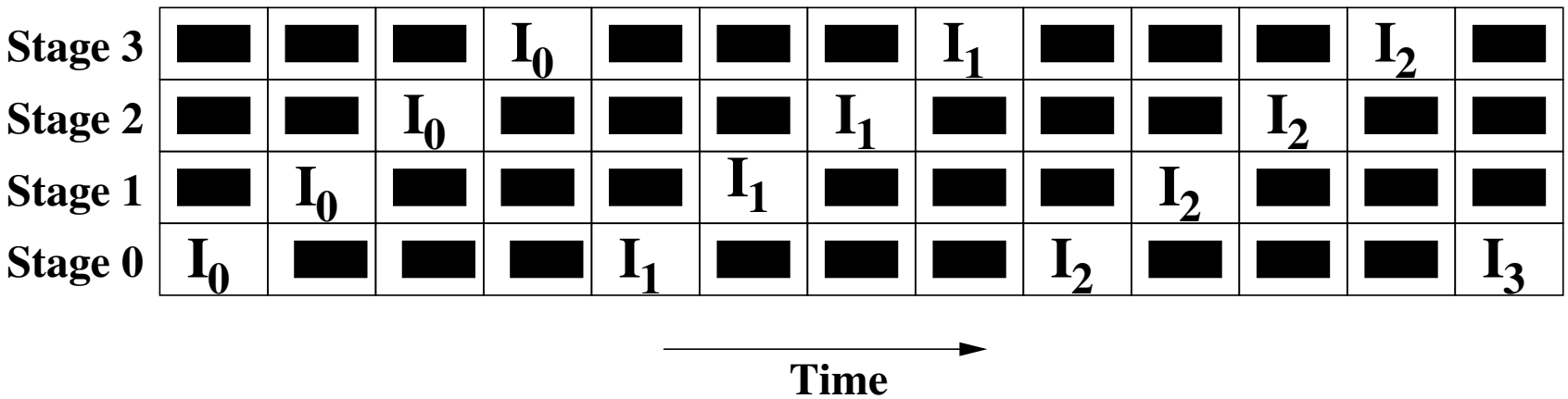
The Basic Concept

Pipelining for instruction execution is similar to construction of factor assembly line for product manufacturing. The basic idea is to decompose the instruction execution process into a collection of smaller functions that can be independently performed by discrete subsystems in the processor implementation. An illustration of this decomposition into 4 parts is:



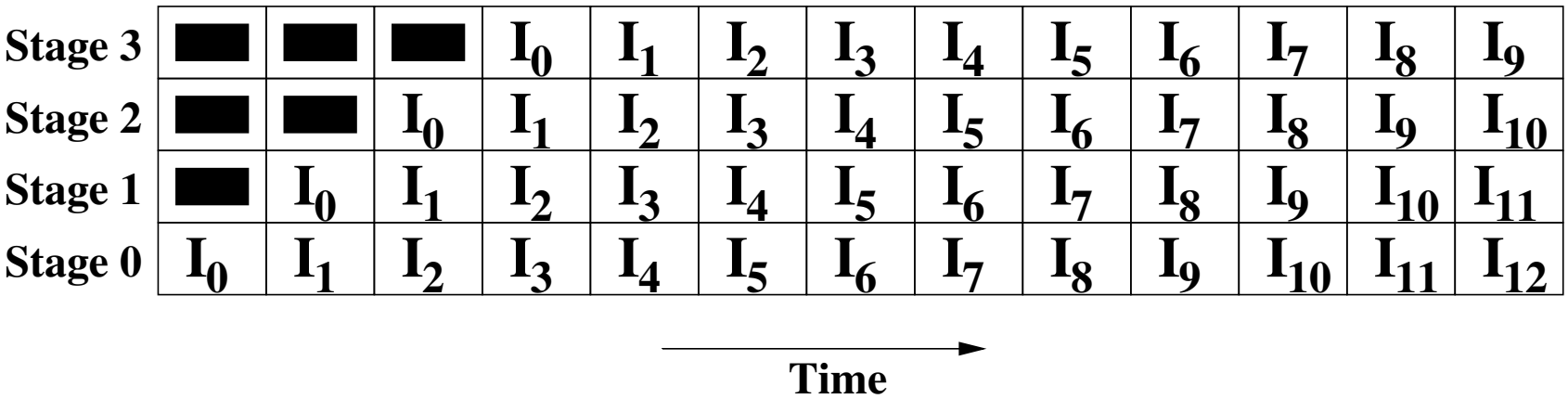
For pipelining, we will organized these discrete subsystems (which are called pipeline stages) implementing the instruction interpretation process into concurrently executing systems each operating on distinct instructions in the instruction stream (much like a factory assembly line).

Typical Non-Pipelined Execution



Time to execute n instructions: $4nt$.

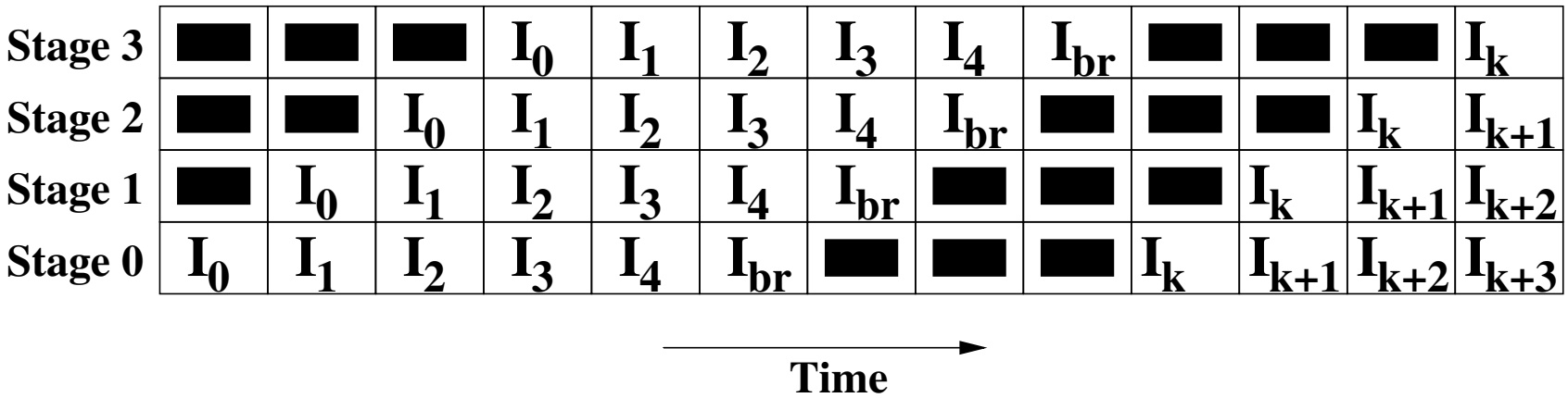
Idealized Pipelined Execution



Time to execute n instructions: $(3 + n)t$.

Steady state: $\frac{\textit{Clock cycles for unpipelined execution}}{\textit{Pipeline depth}}$

Consider the presence of a branch instruction in the pipeline:



$$\textit{Speedup from pipelinining} = \frac{1}{1+\textit{Pipeline stall cycles per instruction}} \times \textit{Pipeline depth}$$

Branch instructions introduce control hazards into the pipeline, negatively impacting performance. There are several types of hazards, namely **control**, **data**, and **structural**.

Pipeline Hazards

- Structural Hazards: arises from hardware resource conflicts. That is, when the hardware cannot service all the combinations of parallel use attempted by the stages in the pipeline.
- Data Hazards: arise when an instruction depends on the (data) results of another instruction that has not yet produced the desired/needed result.
- Control Hazards: arising from the presence of branches or other instructions in the pipeline that alter the sequential instruction flow.

Structural Hazards

UNIVERSITY OF Cincinnati

Issue Description

Insufficient resources to service need.

Commonly arises when you have uneven service rates in the pipe stages.

Sometimes resources are not sufficiently duplicated: *e.g.,* read/writes ports to the register file.

Possible Solutions

- Stall.
- Refactor pipeline or pipeline the pipe stage.
- Duplicate/split the resource (split I/D caches to alleviate memory pressure).
- Build instruction buffers to alleviate memory pressure.

Data Hazards

UNIVERSITY OF Cincinnati

Issue Description

Early pipe stage attempts to read a data/operand value that has not yet been produced by an instruction in a later pipe stage.

This problem gets worse when we look at more complex out of order instruction execution.

Possible Solutions

- Stall.
- Data forwarding (allow earlier pipe stage to fetch incorrect data, but then overwrite the fetched result from the later pipe stage over the prematurely fetched data). Forwarding also called **bypassing** or **short-circuiting**.
- Sometimes forwarding is insufficient and we must use it and stalling to solve all data conflicts.

Control Hazards

Issue Description

The presence of a (conditional) branch alters the sequential flow of instructions and it is not known where to continue until the branch outcome is resolved (general in one of the later stages of the pipeline).

Possible Solutions

- Stall until the branch is revolved.
- Delayed branch: Redefine the runtime behavior of branches to take affect only after the partially fetched/executed instructions flow through the pipeline.
- Branch prediction: Predict (statically or dynamically) the outcome of the branch and fetch there. Next slide.

Branch Prediction

UNIVERSITY OF Cincinnati

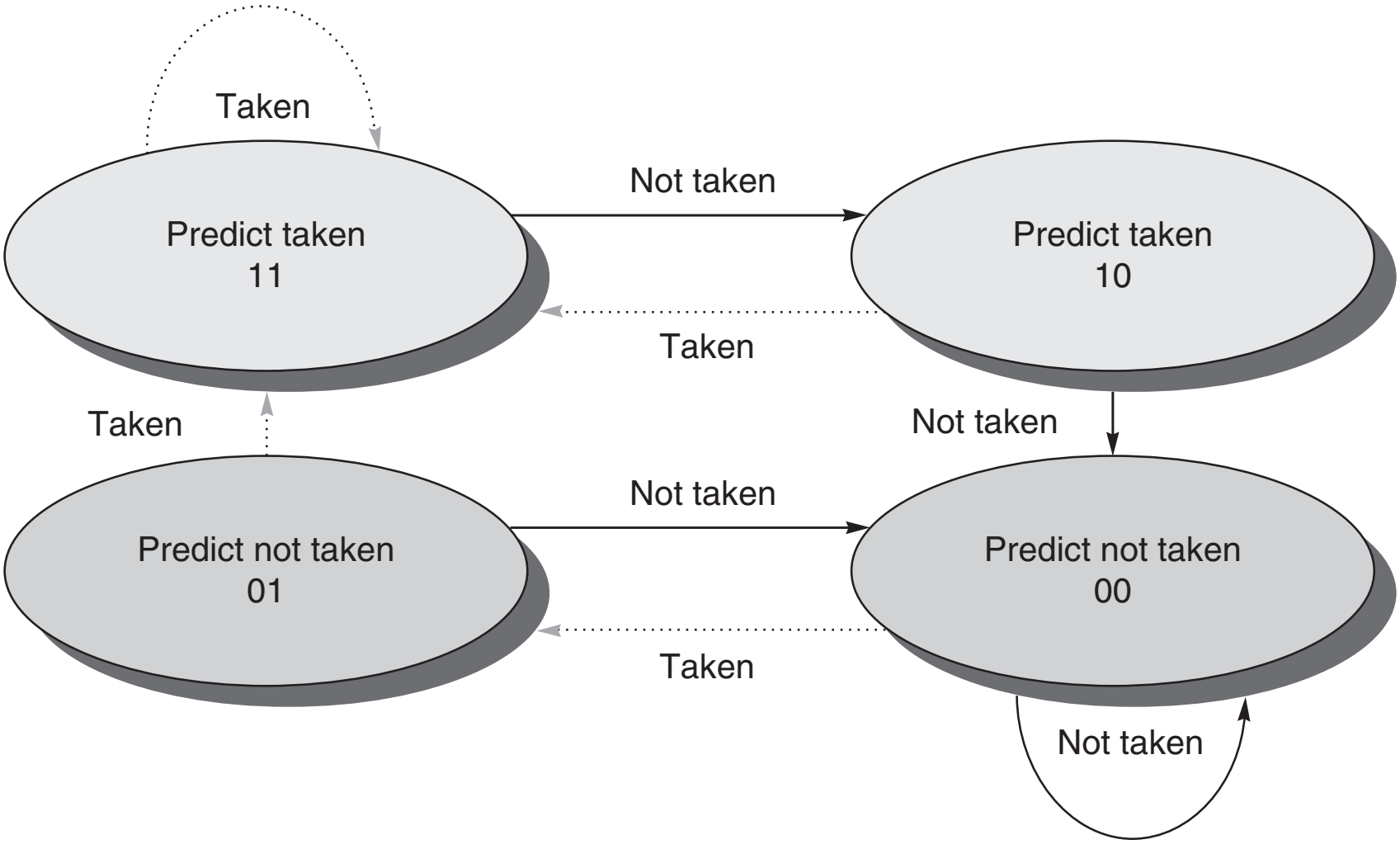
- **Static:** Continue fetching instructions following the branch and design the pipeline so that instructions following the branch can be run safely[1] through the pipe stages and **flush the pipeline** if the branch is taken.
- **Static:** Similar to above, except fetch at branch destination and flush if branch not taken.
- **Dynamic:** Track branch instruction behavior using a branch-prediction buffer (or branch history table) and use it to predict[2] the direction of the branch. Continue fetching at the predicted location.

---

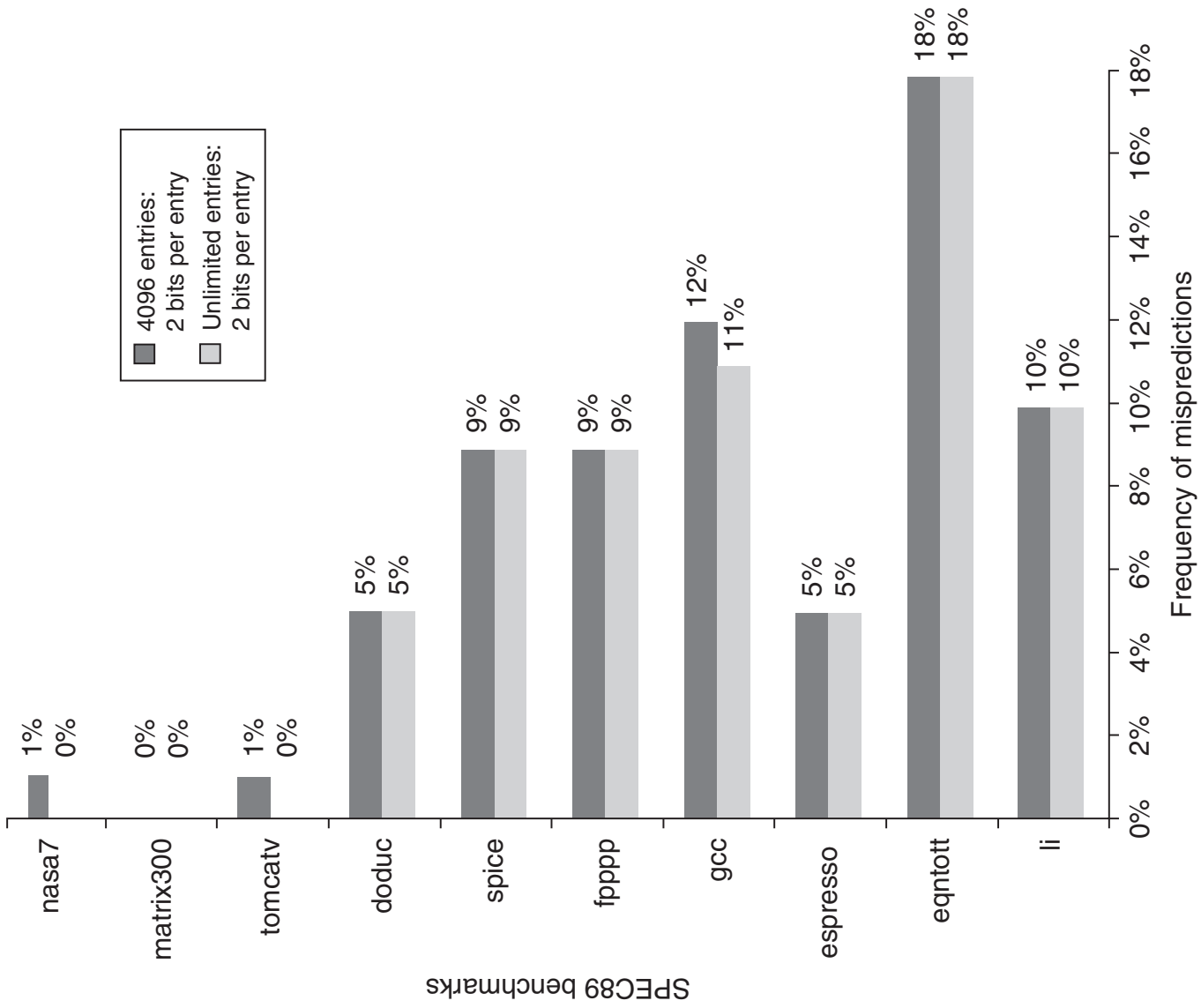
[1] The instructions following the branch do not cause changes to the visible machine state, or the changes made can easily be undone.

[2] There are numerous mechanisms to implement the branch-prediction buffer. Simple ones are 1-bit or 2-bit predictors; more complex techniques are also possible.

An example 2-bit (dynamic) predictor



Effectiveness of dynamic prediction



Characterizations of Exceptions

1. Synchronous vs asynchronous: asynchronous triggered by external devices; usually can be handled after completion of current instruction.
2. User requested vs coerced: examples of user requested: invoke O/S, trace instr execution, breakpoint.
3. Maskable vs nonmaskable: mask controls whether the hardware responds to the exception or not.
4. Within vs between instrs: exceptions that occur within (during) an instruction execution is more difficult to process if restart of the interrupted instr stream is required.
5. Resume vs terminate: terminate is easier as the system does not have to save a restart state.

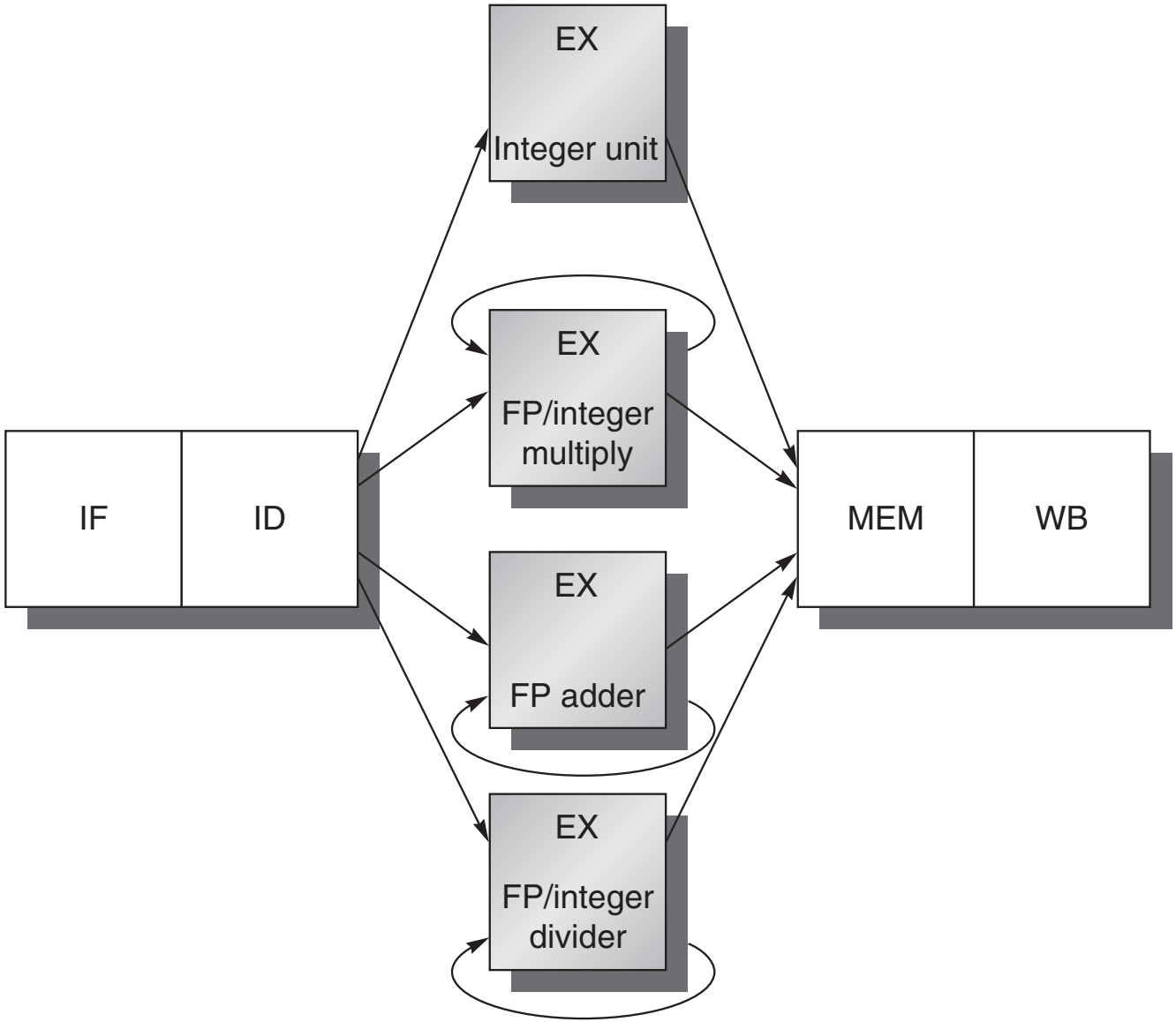
Precise Exceptions

A precise exception occurs when the machine completes all instructions preceding the interrupting instruction and does not (visibly) execute any instructions following the faulting instruction. The machine can thus be restarted at the interrupting instruction.

Some architectural features make precise exceptions impossible to realize: specifically the delayed branch

Some machines operate in 2 modes: a **fast mode** without precise exceptions and a **slow mode** with precise exceptions. Of course any machine with **non-precise exceptions of restartable events** must have some mechanism to capture the partial state and resume/finish executing the from the imprecise state so that proper execution occurs.

Multicycle Operation



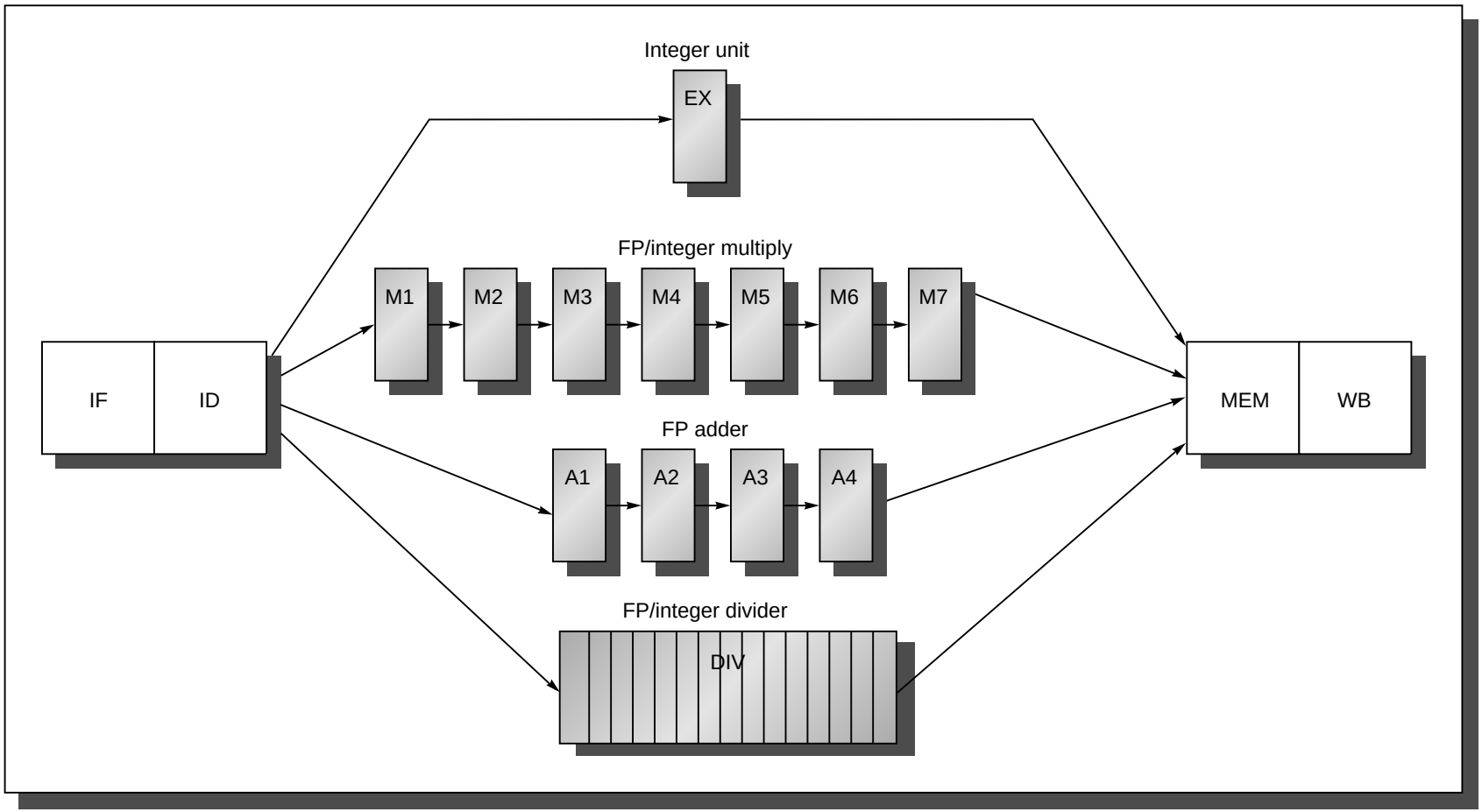


FIGURE 3.44 A pipeline that supports multiple outstanding FP operations.

Multicycle Operation

Latency: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.

Initiation interval (or repeat interval): the number of cycles that must elapse between issuing two operations of a given type.

Warning, lots of forward references here; we will define the hazard taxonomy (RAW, WAW, WAR) and dependencies and types of data dependencies.

Out-of-Order Execution

Allowing instructions to finish from the pipeling before earlier instructions with longer execution steps are finished.

How to manage exceptions in the slow instructions?

history file: track the original values of registers to they can be restored when interrupt/exception occurs.

future file: capture register changes in a future file and commit in order of instruction completion.

Allow imprecise exceptions, save enough pipeline state to restart only the incompleted instructions.

Delay schedule instructions for execution until you can guarantee that forward instructions will not cause a restartable exception.

Dynamically Scheduled Pipelines

Hardware schedules instructions to reduce stalls due to hazards.

Two main techniques: a scheduling Scoreboard and Tomasulo's algorithm.

Scoreboarding: use a centralized scoreboard to record instructions and the data/registers to be read/written by them that are in flight. Details in this appendix.

Tomasulo: distributed mechanism.