Computer Architecture

Chapter 5:
PIPELINE MICROARCHITECTURE

Parallelism

- Instruction-level parallelism
  - pipeline
  - superscale
  - latency issues
- Processor-level parallelism
  - array/vector of processors (1 control unit, limited application)
  - multiprocessor (multiple CPUs, common memory)
  - multicomputer (multiple CPUs, each with own memory)

Instruction-Level Parallelism

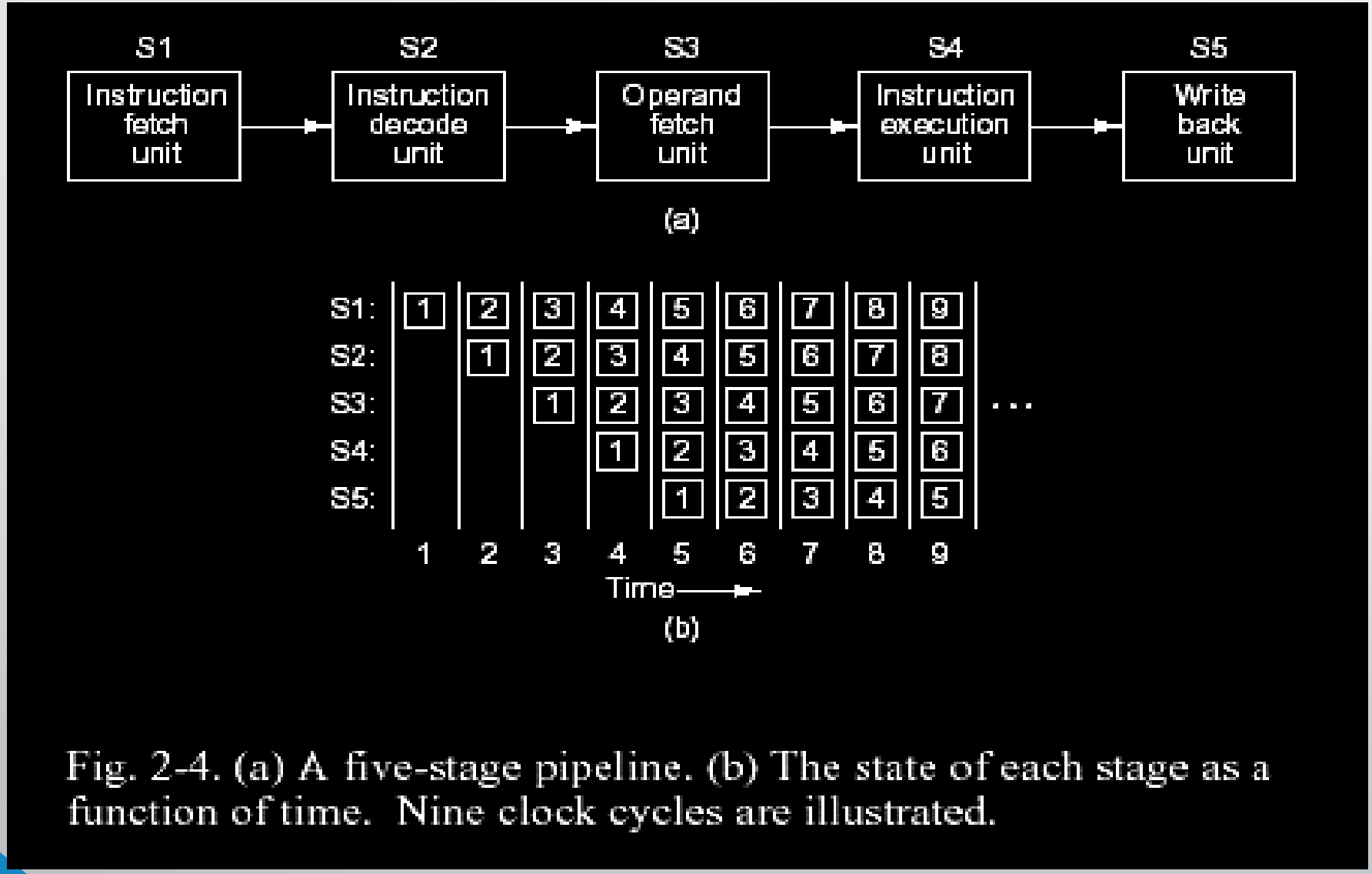


Fig. 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

- Pipelining allows a trade-off between latency (how long it takes to execute an instruction), and processor bandwitdh (how many MIPS the CPU has).
- If:
- Cycle time= T nsec
- N stages in the pipeline
- So Latency= NxT nsec and Bandwidth= 1000/T MIPS

Instruction Pipelining

- Similar to assembly line in manufacturing plants:

  Products at various stages can be worked on simultaneously

  $\Rightarrow$ Performance improved

- First attempt: 2 stages
  - Fetch
  - Execution

Prefetch

- Fetch accessing main memory
- Execution usually does not access main memory
- Can fetch next instruction during execution of current instruction
- Called instruction prefetch or fetch overlap
- *Ideally* instruction cycle time would be halved

  (if $\text{duration}_F = \text{duration}_E$ …)

Improved Performance (1)

- But not doubled in reality, why?
  - Fetch usually shorter than execution
    - Prefetch more than one instruction?
  - Any jump or branch (branching) means that prefetched instructions are not the required instructions
    - e.g., ADD A, B

      BEQ NEXT

      ADD B, C

      NEXT SUB C, D

Improved Performance (2)

- Add more stages to improve performance
- Reduce time loss due to branching by guessing
  - Prefetch instruction after branching instruction
  - If not branched

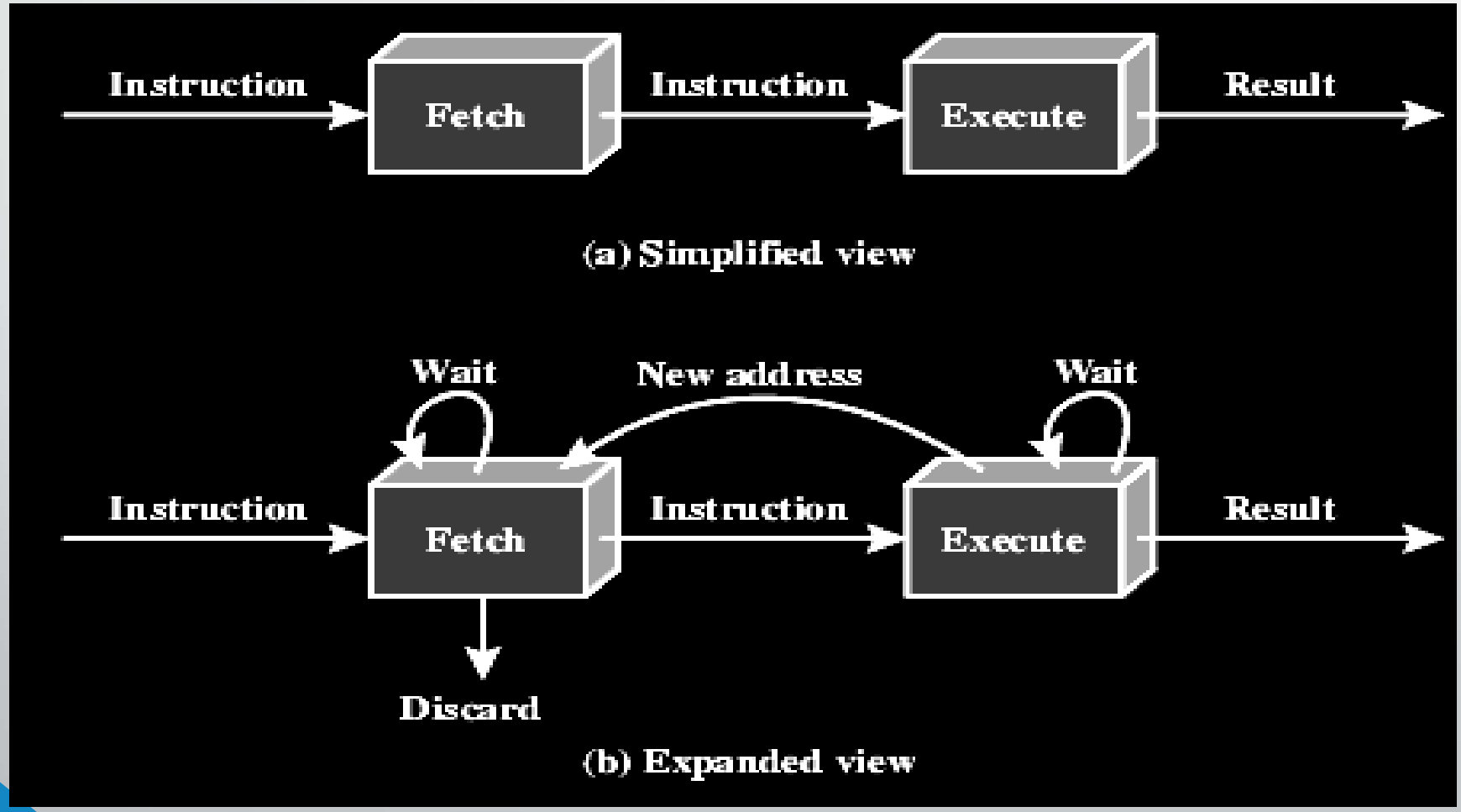
        use the prefetched instruction

    else

        discard the prefetched instruction

        fetch new instruction

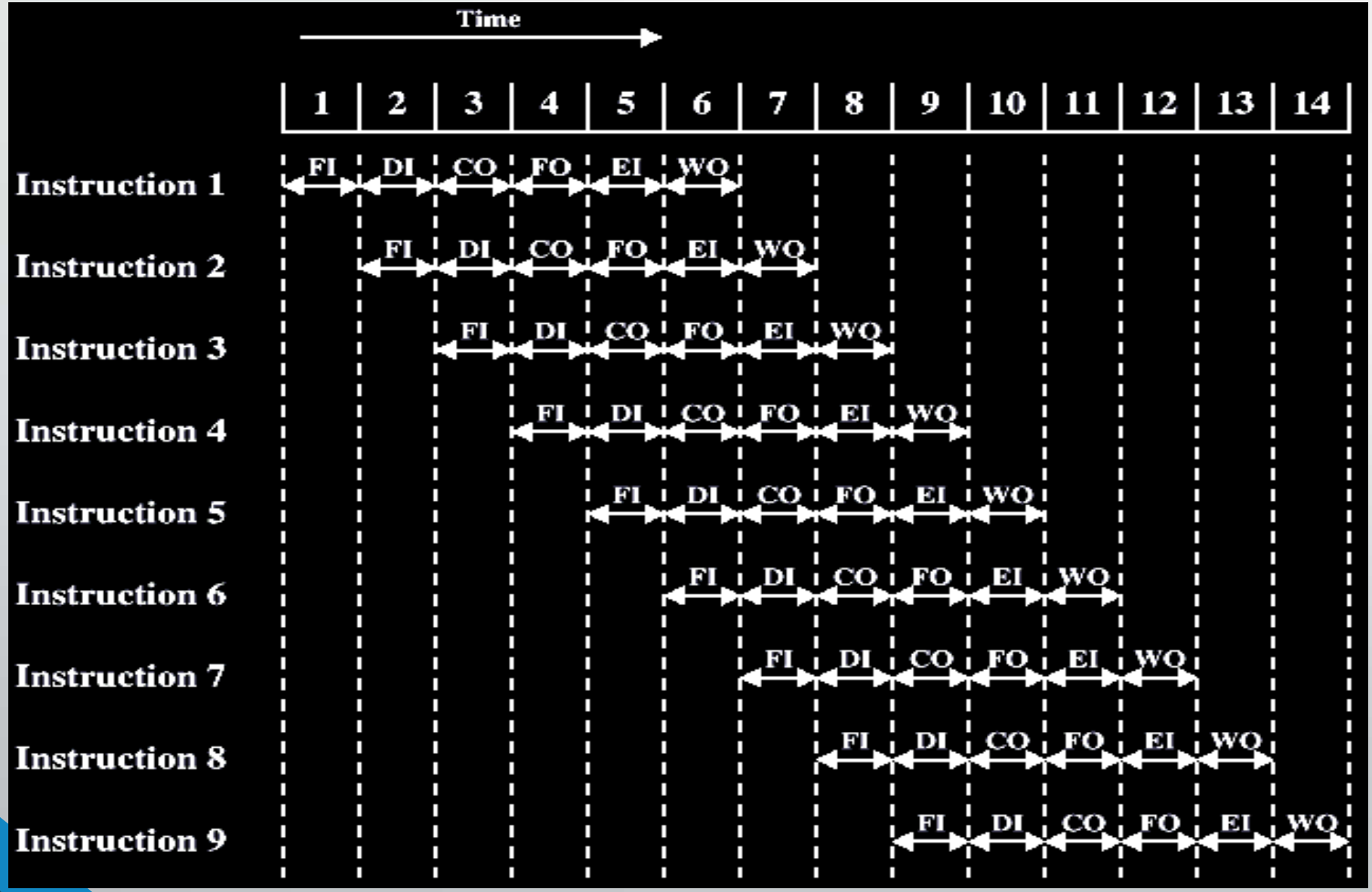
Two Stage Instruction Pipeline



Pipelining

- More stages ⇒ more speedup
  - FI: Fetch instruction
  - DI: Decode instruction
  - CO: Calculate operands (i.e. EAs)
  - FO: Fetch operands
  - EI: Execute instructions
  - WO: Write result
- Various stages are of nearly equal duration
- Overlap these operations

Timing of Pipeline



Speedup of Pipelining (1)

- 9 instructions 6 stages

  w/o pipelining: __ time units

  w/ pipelining: __ time units

  speedup = _____
- Q: 100 instructions 6 stages, speedup = ____
- Q: ∞ instructions *k* stages, speedup = ____

Speedup of Pipelining (2)

- Parameters
  - τ = pipeline cycle time = time to advance a set of instructions one stage
  - k = number of stages
  - n = number of instructions
- Assume no branch, time to execute n instructions

  $T_k = [k + (n - 1)]\tau$
- Time to execute n instructions without pipelining

  $T_1 = nk\tau$

Speedup of Pipelining (3)

- Speedup of *k*-stage pipelining compared to without pipelining

$$S_k = \frac{Performance_{pipelining}}{Performance_{no\_pipelining}}$$

$$= \frac{T_1}{T_k} = \frac{nk\tau}{[k + (n-1)]\tau} = \frac{nk}{k + (n-1)}$$

- Q: ∞ instructions *k* stages, speedup = ____

Pipelining - Discussion

- Not all stages are needed in one instruction
  - e.g., LOAD: WO not needed
- Timing is set up assuming all stages are needed by each instruction

  $\Rightarrow$ Simplify pipeline hardware
- Assume all stages can be performed in parallel
  - e.g., FI, FO, and WO $\Rightarrow$ memory conflicts

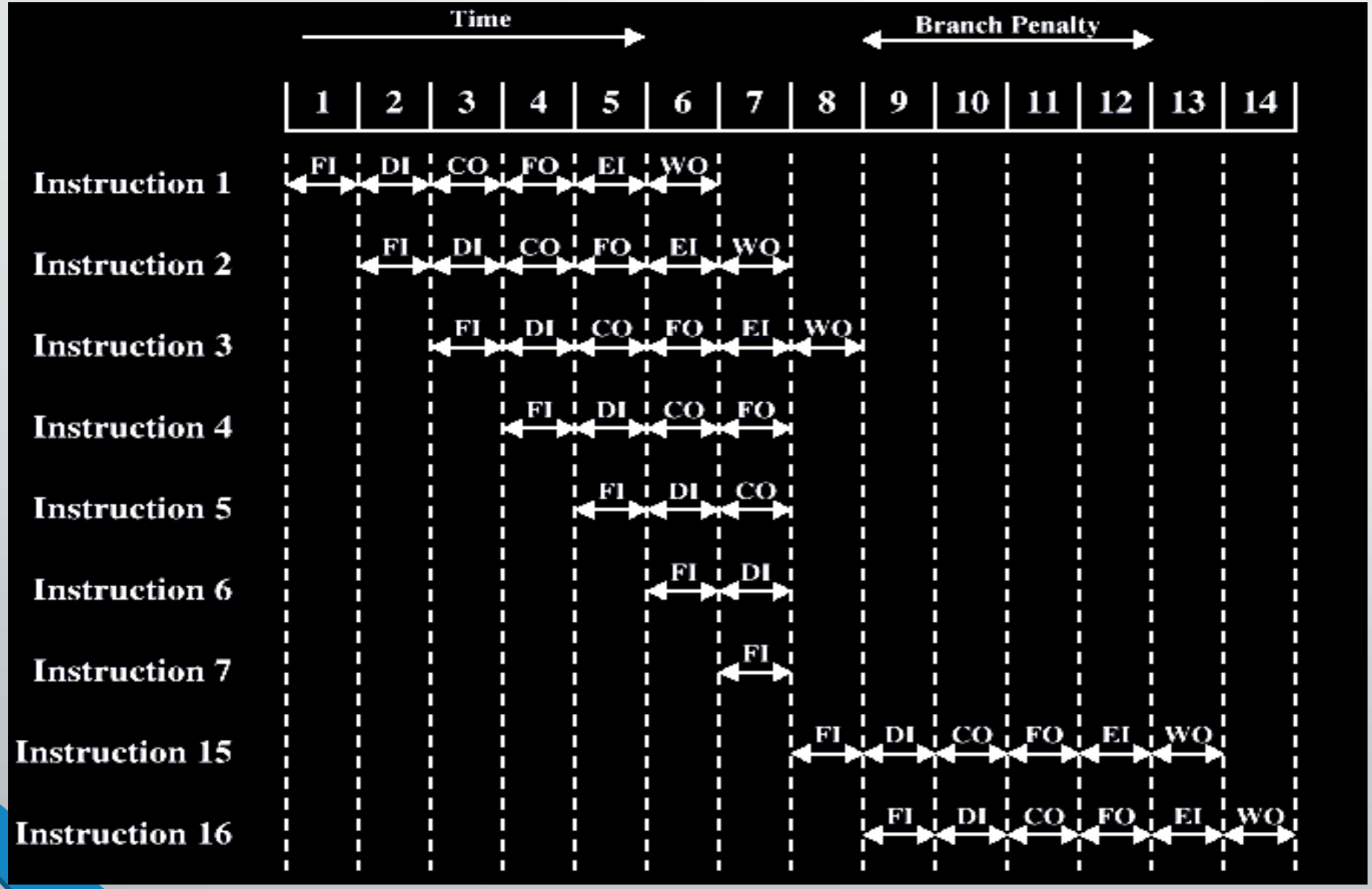
Limitation by Branching

- Conditional branch instructions can invalidate several instruction prefetches
- In our example (see next slide)
  - Instruction 3 is a conditional branch to instruction 15
  - Next instruction's address won't be known till instruction 3 is executed (at time unit 7)

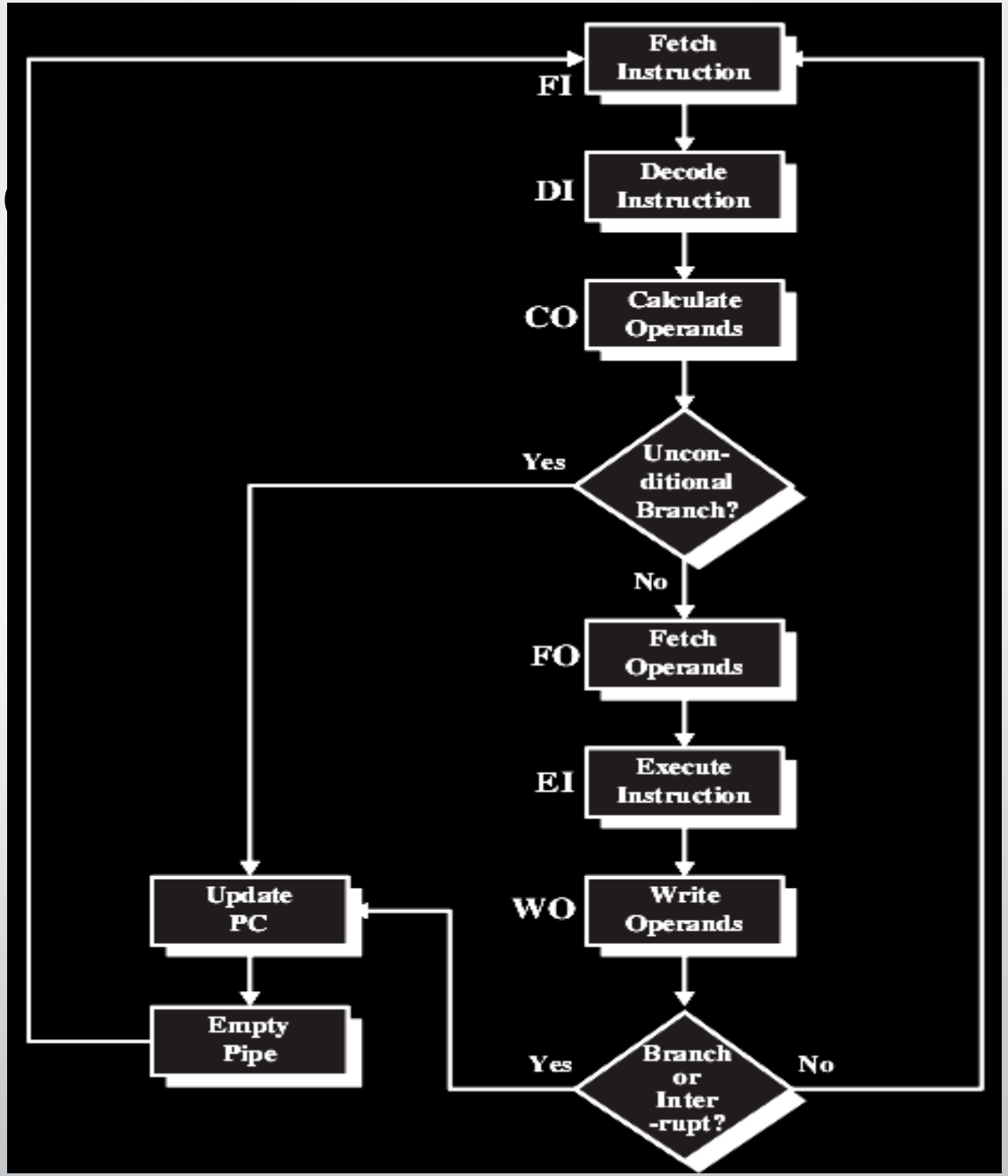
  $\Rightarrow$ pipeline must be cleared
  - No instruction is finished from time units 9 to 12

  $\Rightarrow$ performance penalty

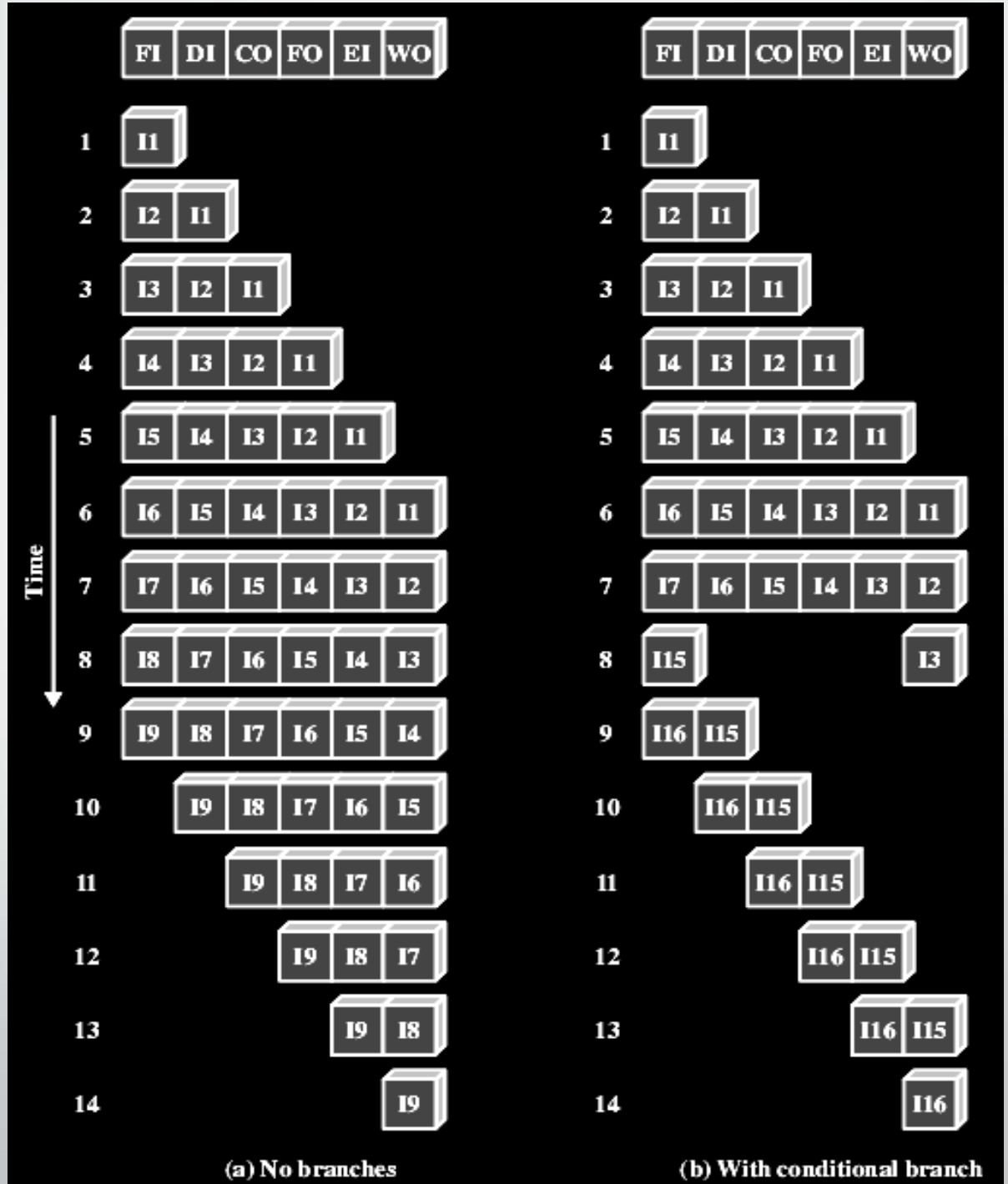
Branch in a Pipeline



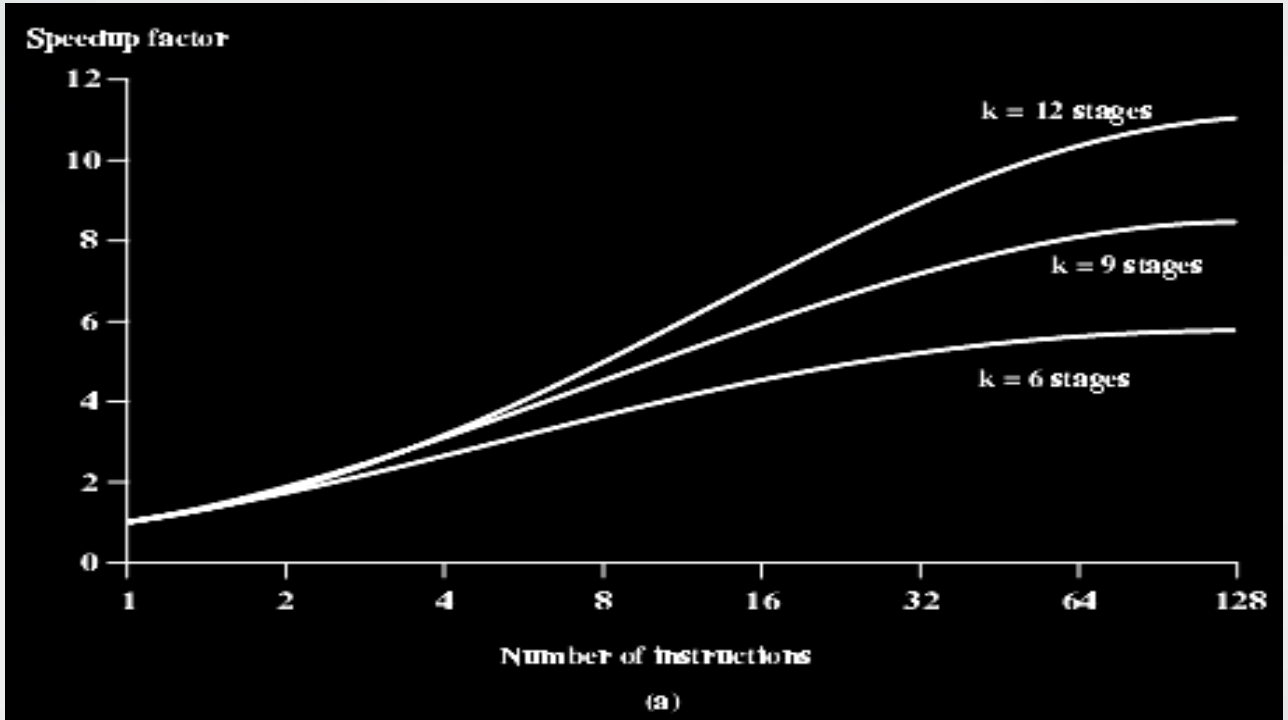
Six Stage Instruction Pipelin

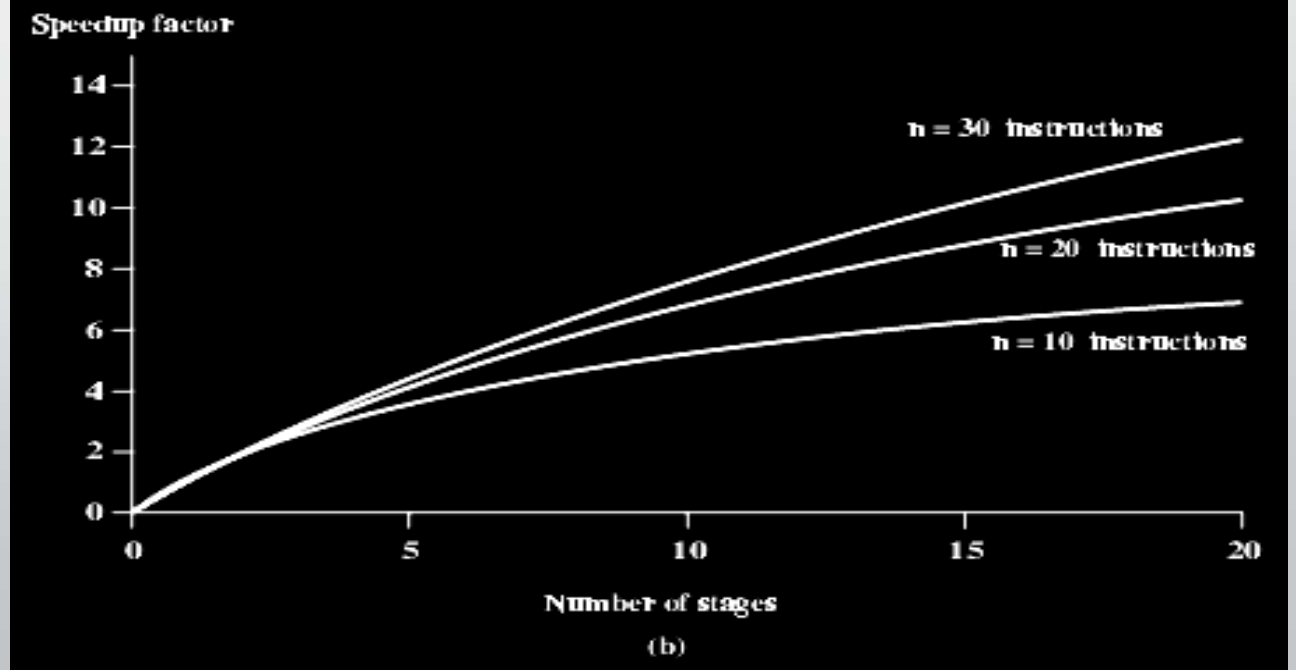


Alternative Pipeline Depiction



Speedup Factors with Instruction Pipelining





Limitation by Data Dependencies

- Data needed by current instruction may depend on a previous instruction that is still in pipeline
- E.g., $A \leftarrow B + C$

  $D \leftarrow A + E$

Data Hazards in pipeline

MIPS instructions have the format

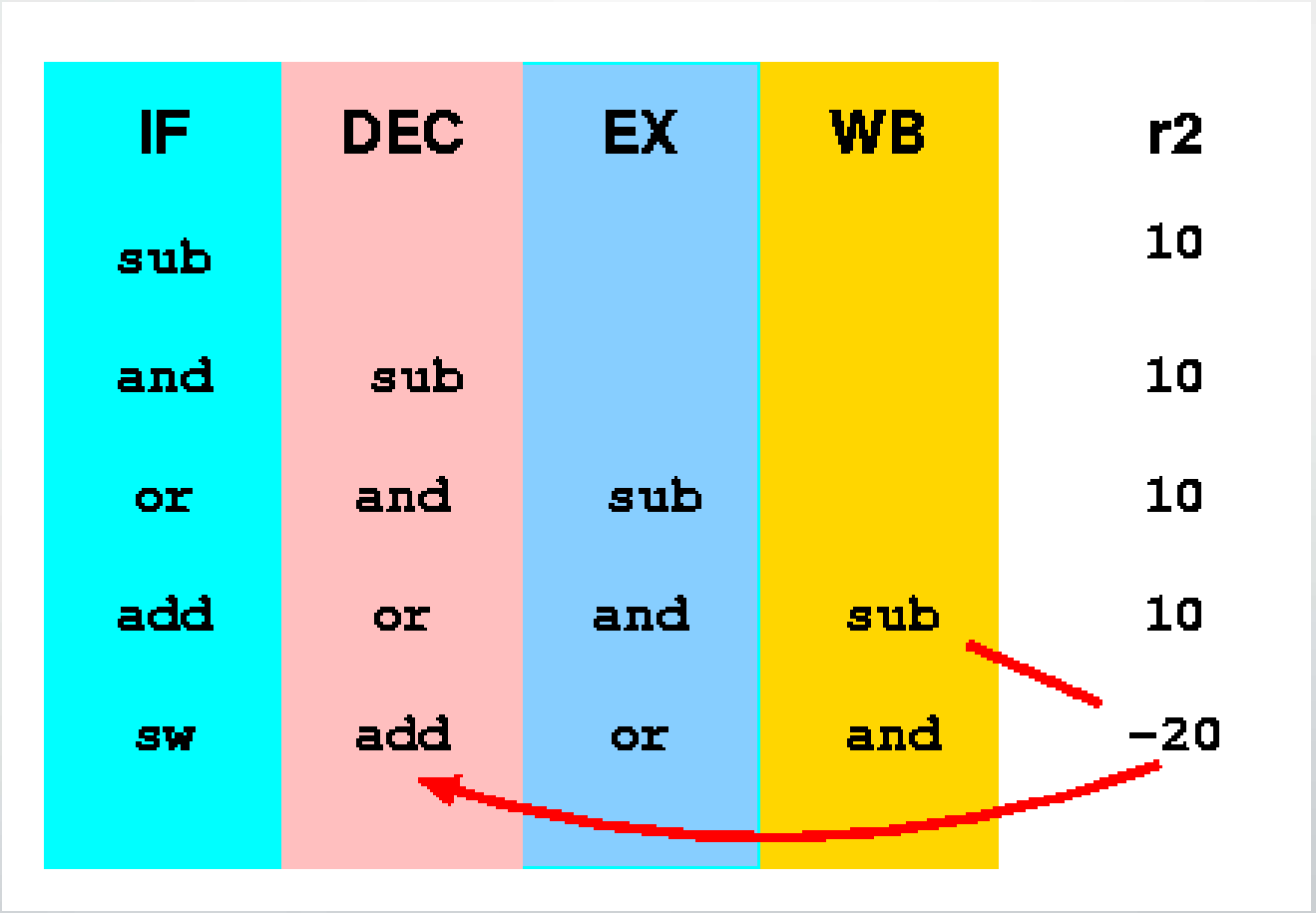
op dest, src_a, src_b

- **Code:**
  - ```
    sub  $2, $1, $3
    and $12, $2, $5
    or $13, $6, $2
    add $14, $2, $2
    sw $15,100($2)
    ```

The last four instructions all depend on a result produced by the first!
    ```

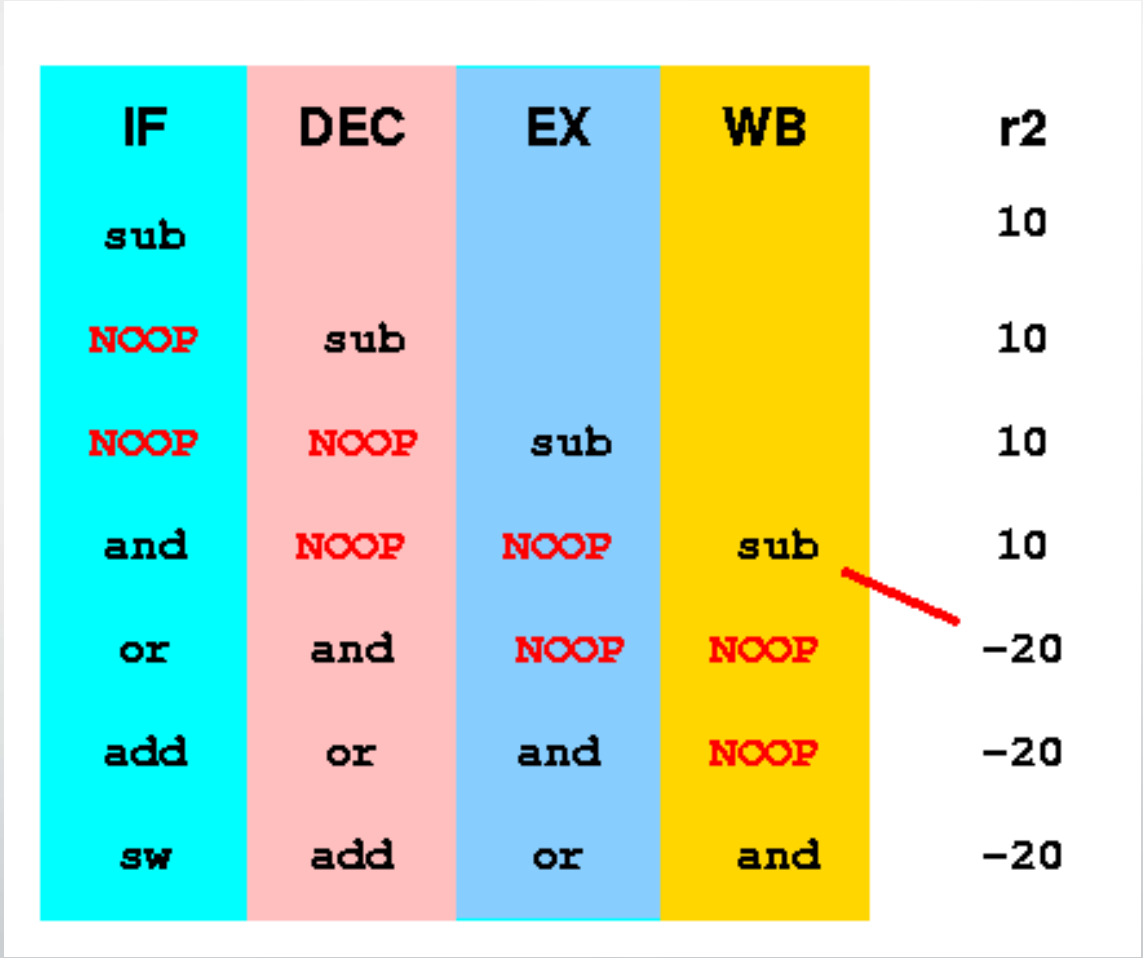
Pipelines - Data hazards

- Examine the pipeline
- r2 only updated for add!



Pipelines - Data Hazards

- Compiler solution
  - Insert NOOPs
  - Inefficient!



Performance of Pipeline

- Ideally, more stages, more speedup
- However,
  - more overhead in moving data between buffers
  - more overhead in preparation
  - more complex circuit for pipeline hardware

Dealing with Branches

- Multiple Streams
- Prefetch Branch Target
- Loop buffer
- Branch prediction
- Delayed branching

Multiple Streams

- Have two pipelines
- Prefetch each branch into a separate pipeline
- Use appropriate pipeline
- Problems
  - Leads to bus & register contention delays
  - Multiple branches (i.e., additional branch entering pipelines before original branch decision made) lead to further pipelines being needed
- Can improve performance, anyway
  - e.g., IBM 370/168

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch
- Keep target until branch is executed
- If branch is taken, target is already prefetched
- Used by IBM 360/91

Loop Buffer (1)

- Small, very fast memory
- Maintained by fetch (IF) stage of pipeline
- Contains the *n* most recently fetched instructions in sequence
- If a branch is to be taken
  - Hardware checks whether the target is in buffer

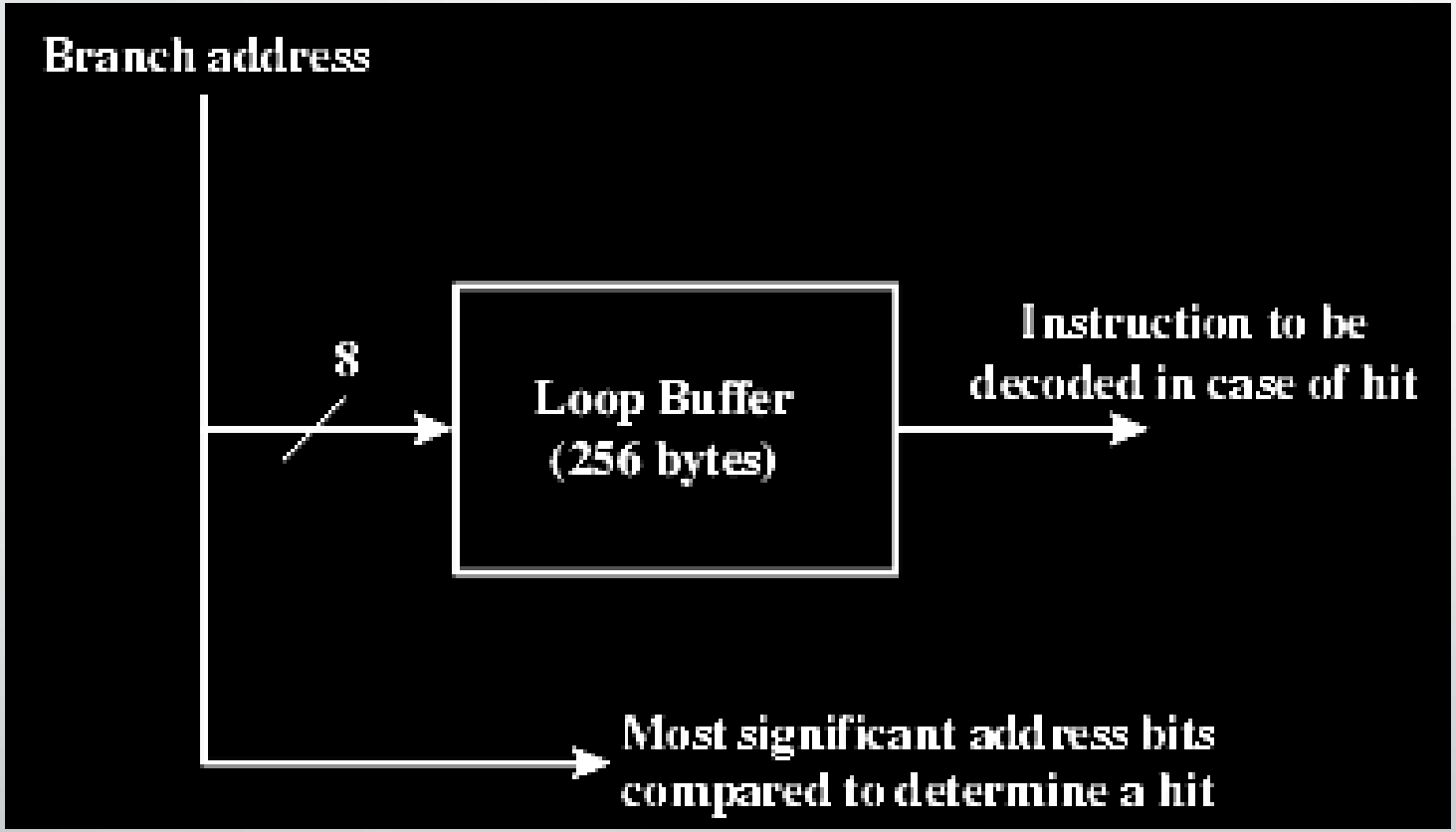
    If YES then next instruction is fetched from the buffer

    else fetch from memory

Loop Buffer (2)

- Reduce memory access time
- Very good for small loops or jumps
- If buffer is big enough to contain entire loop, instructions in the loop need to be fetched from memory only once at the first iteration
- c.f. cache
- Used by CRAY-1

Loop Buffer Diagram



Branch Prediction (1)

- Predict whether a branch will be taken
- If the prediction is right

  ⇒ No branch penalty
- If the prediction is wrong

  ⇒ Empty pipeline

  Fetch correct instruction

  ⇒ Branch penalty

Branch Prediction (2)

- Predict techniques
- Static
  - Predict never taken
  - Predict always taken
  - Predict by opcode
- Dynamic
  - Taken/not taken switch
  - Branch history table

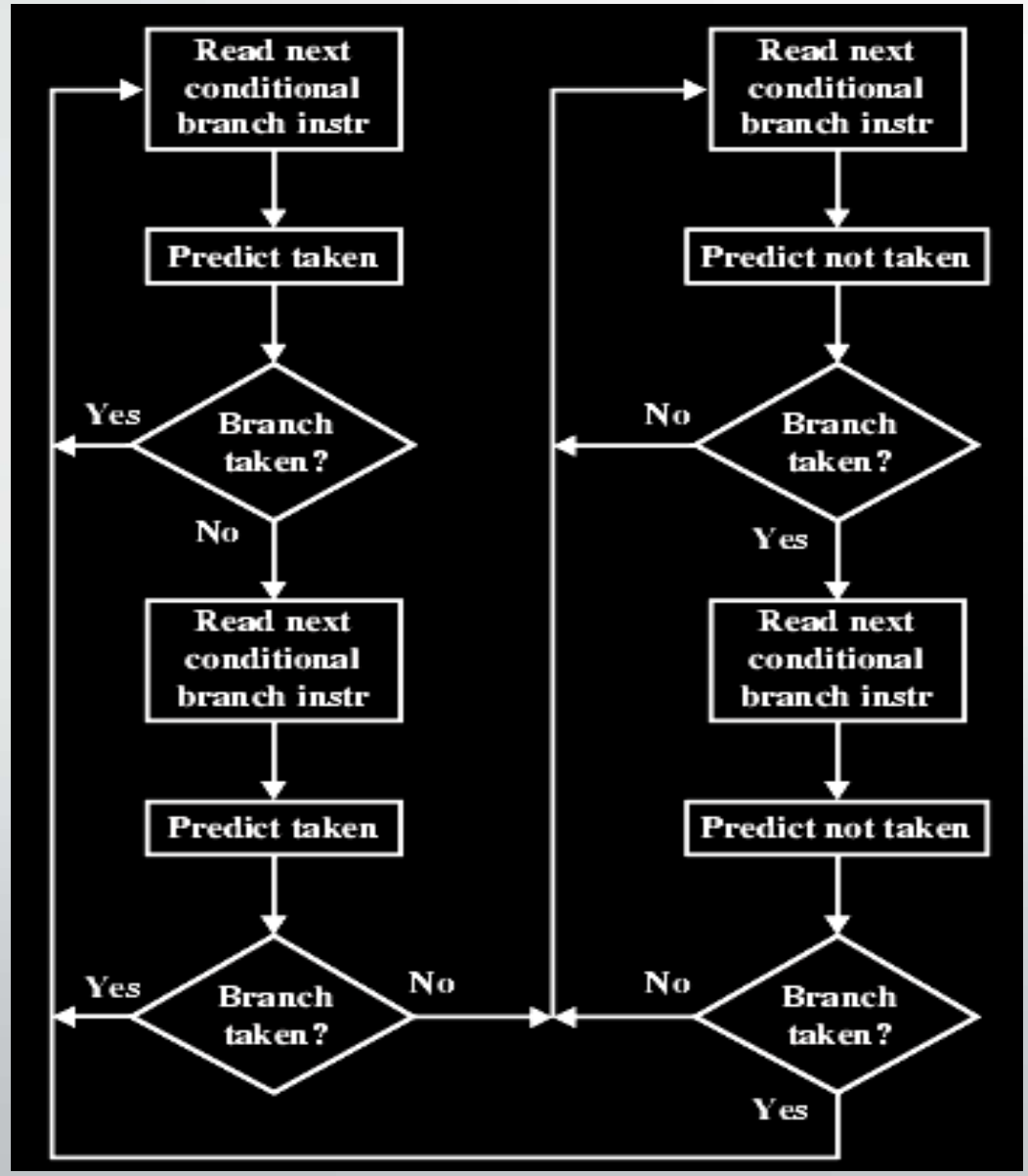
Branch Prediction (3)

- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
  - VAX will not prefetch after branch if a page fault would result (O/S v CPU design)
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction

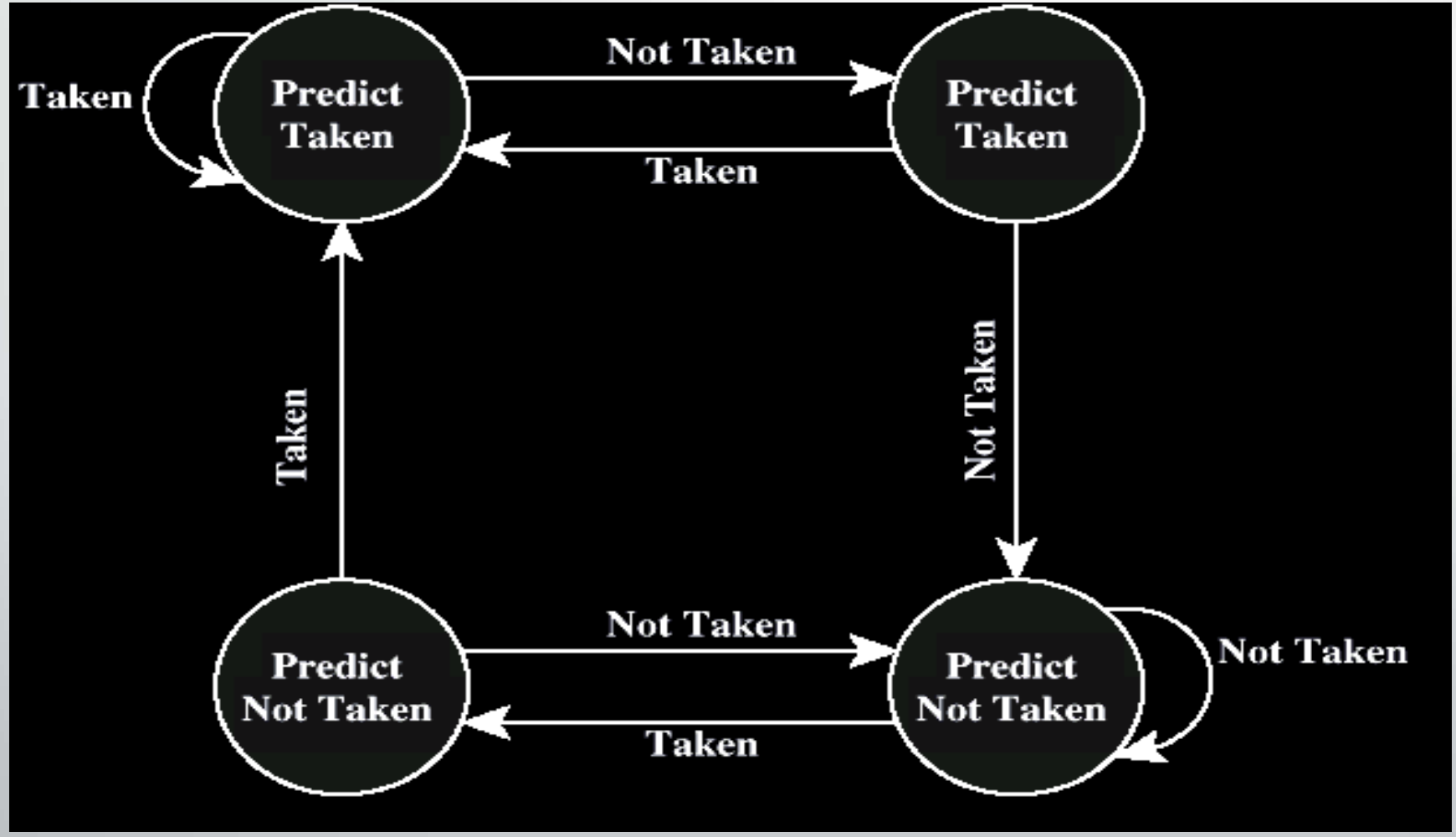
Branch Prediction (4)

- Predict by Opcode
  - Some instructions are more likely to result in a jump than others
  - Can get up to 75% success
- Taken/Not taken switch
  - Based on previous history
  - Good for loops
- Branch history table
  - Like a cache to look up

Branch Prediction Flowchart



Branch Prediction State Diagram



Dealing With Branches