18-600 Foundations of Computer Systems

Lecture 10: "From Pipelined to Superscalar Processors"

John P. Shen & Zhiyi Yu
October 3, 2016

- Required Reading Assignment:
  - **Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**
- Recommended Reading Assignment:
  - **Chapter 4 of Shen and Lipasti (SnL).**

Electrical & Computer ENGINEERING

Carnegie Mellon University

18-600 Foundations of Computer Systems

Lecture 10: "From Pipelined to Superscalar Processors"

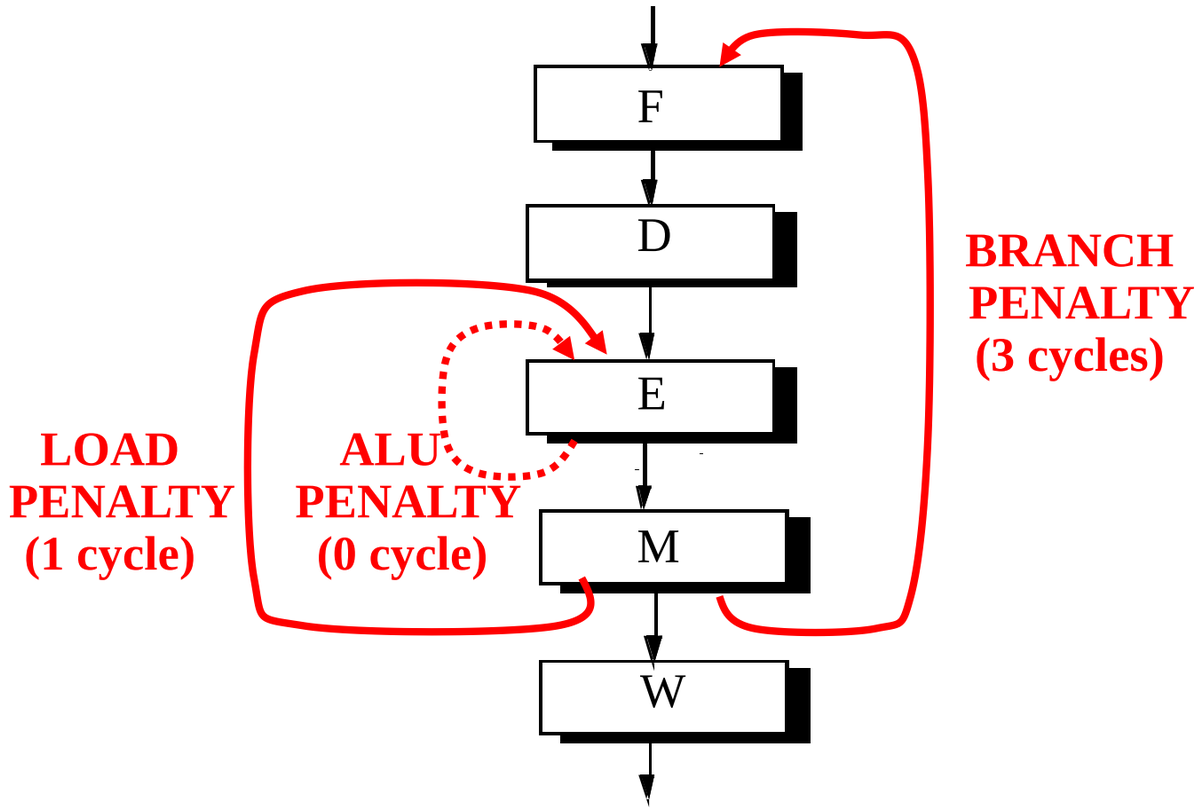
A. Motivations for Superscalar Processors
- Pipelined Processor Limitations
- Superscalar Processor Pipelines
- Instruction Level Parallelism (ILP)

B. Superscalar Pipeline Implementation
- Instruction Fetch and Decode
- Instruction Dispatch and Issue
- Instruction Execute
- Instruction Complete and Retire

C. Instruction Flow Techniques
- Control Flow Prediction
- Dynamic Branch Prediction

Electrical & Computer ENGINEERING

3 Major Penalty Loops of (Scalar) Pipelining



Performance Objective: Reduce CPI as close to 1 as possible.

Best Possible for Real Programs is as Low as CPI = 1.15.

Address Limitations of Scalar Pipelined Processors

- Upper Bound on Scalar Pipeline Throughput

  Limited by IPC = 1.0

  ➔ Parallel Pipelines

- Inefficient Unification Into Single Pipeline

  Long latency for each instruction

  Hazards and associated stalls

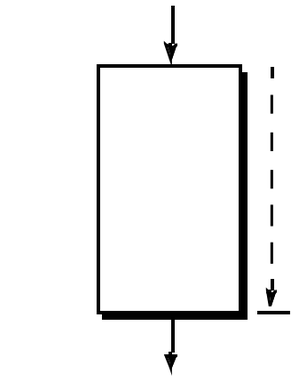
  ➔ Diversified Pipelines

- Performance Lost Due to In-order Pipeline

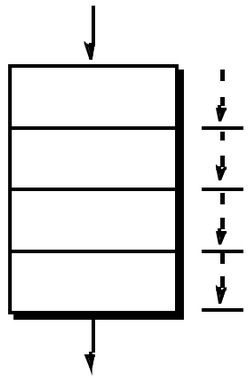
  Unnecessary stalls

  ➔ Dynamic Pipelines

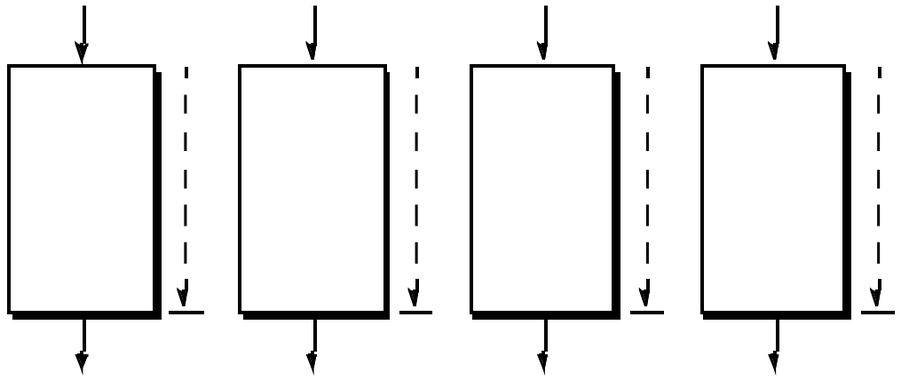
Parallel Pipelines



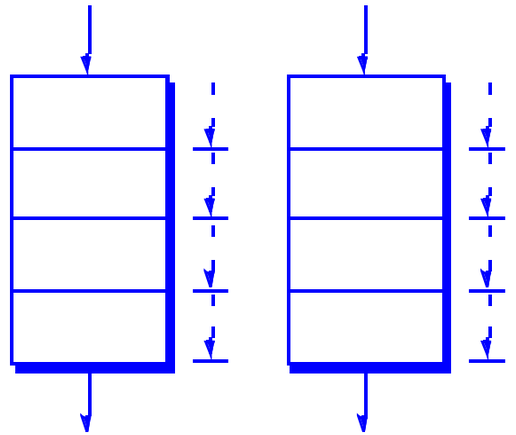
(a) No Parallelism



(b) Temporal Parallelism

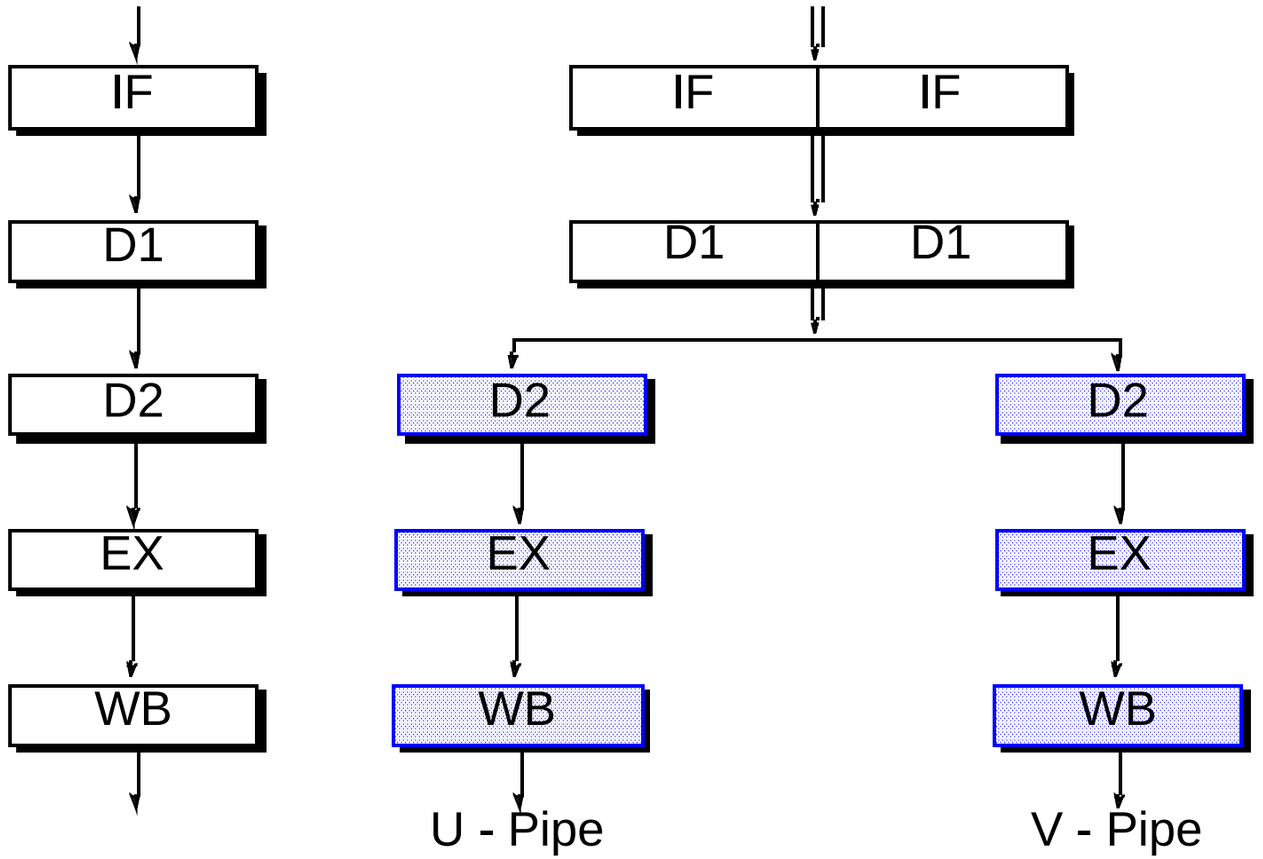


(c) Spatial Parallelism

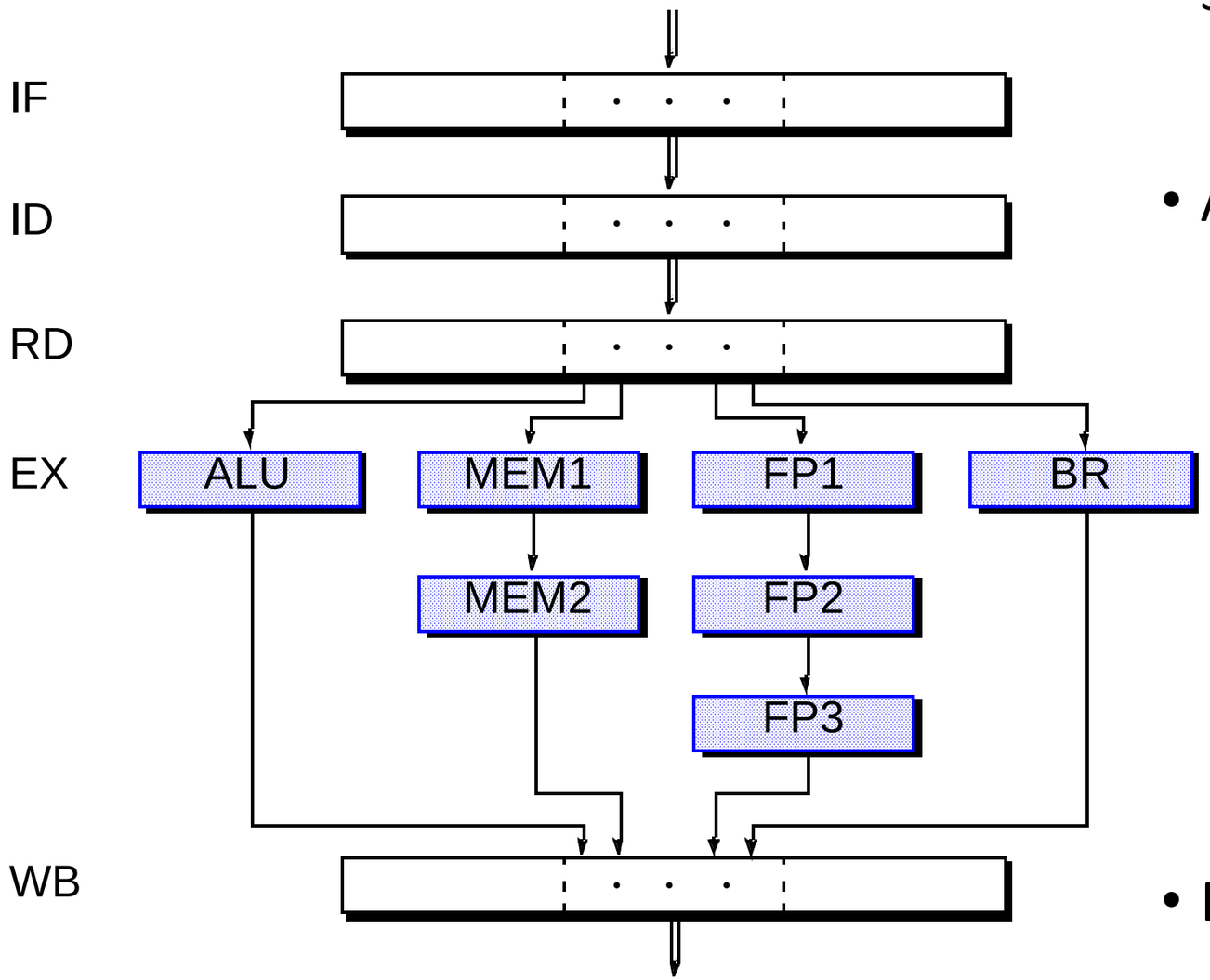


(d) Parallel Pipeline

Intel Pentium Parallel Pipeline

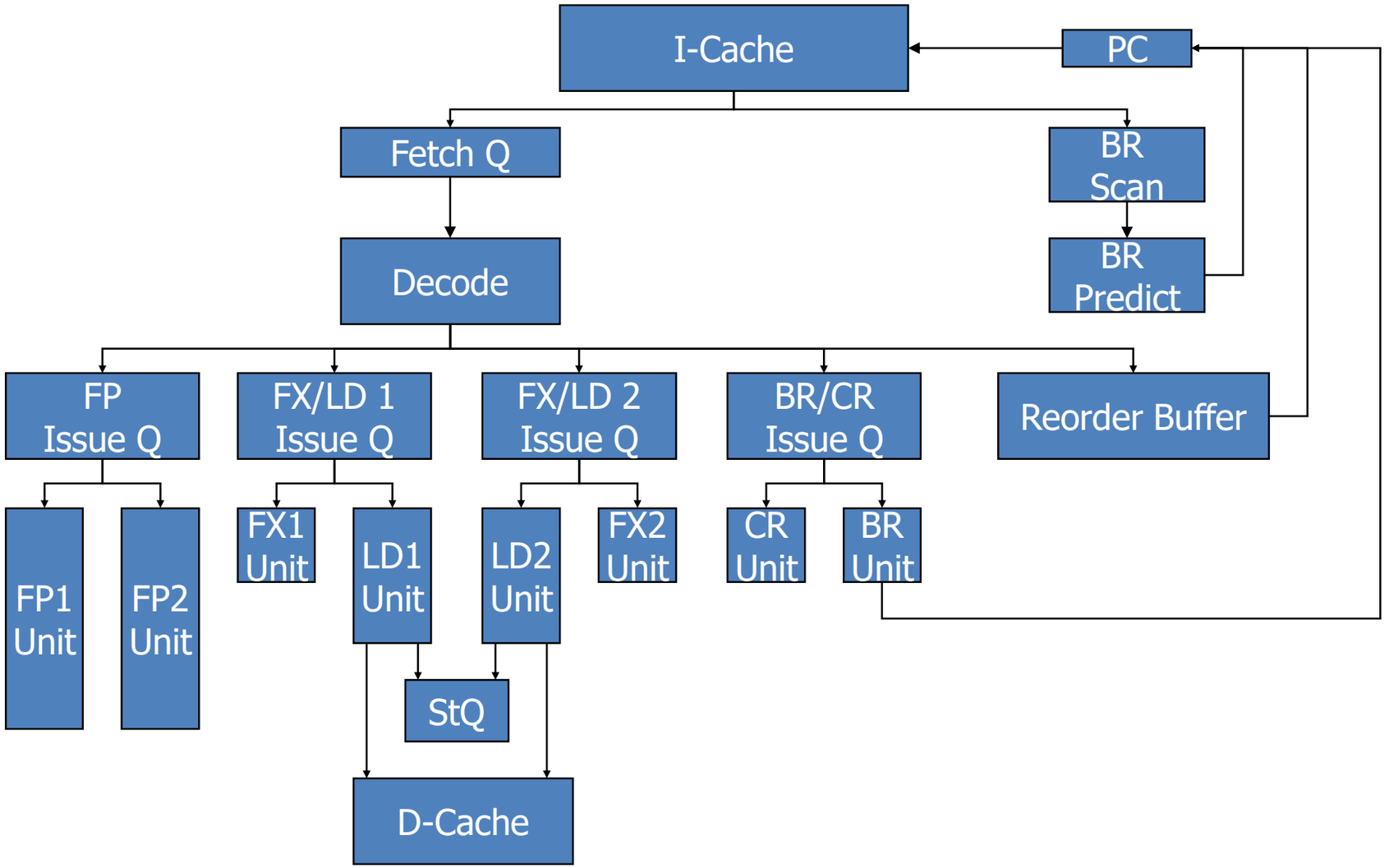


Diversified Pipelines

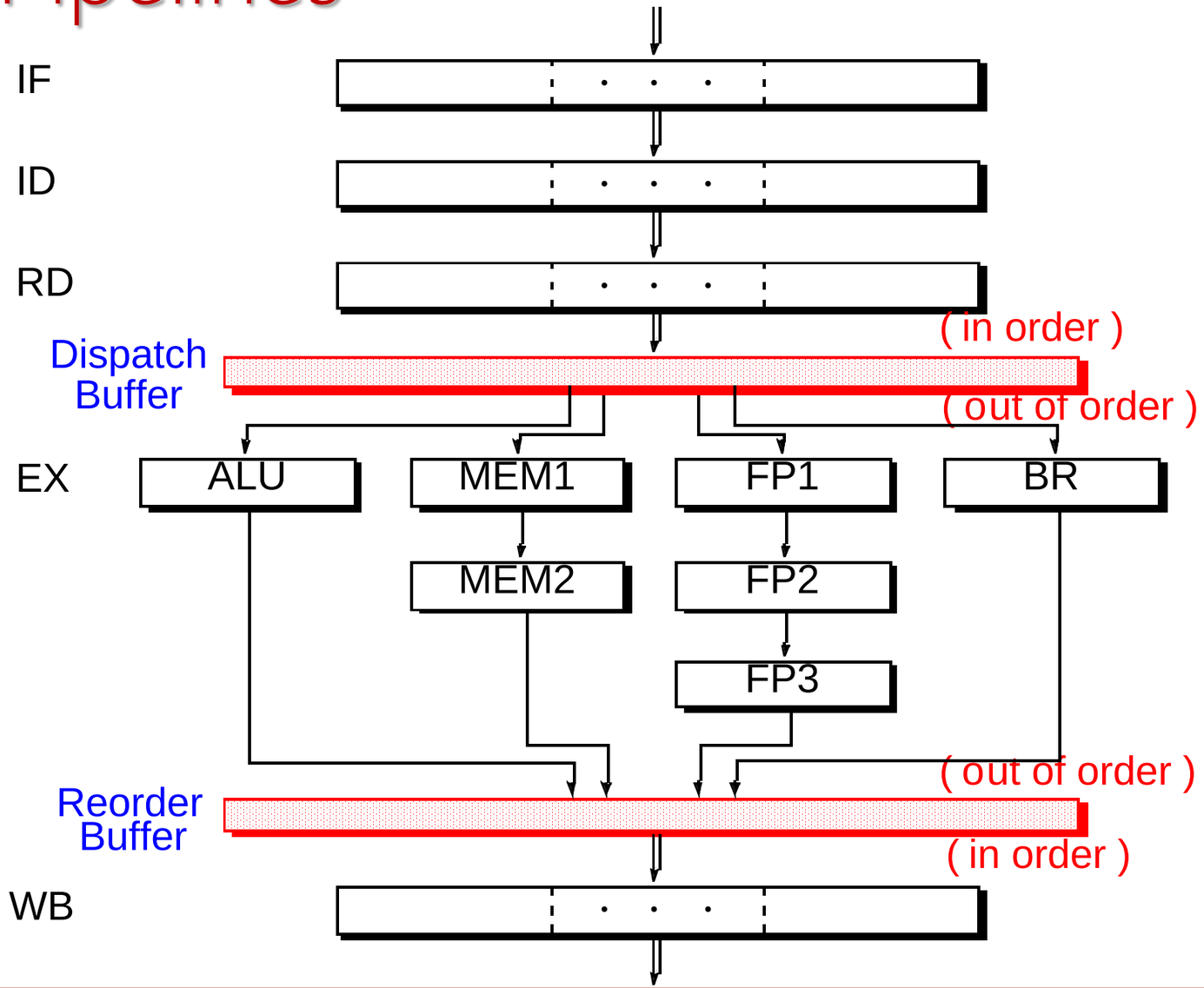


- Separate execution pipelines
  - Integer simple, memory, FP, …
- Advantages:
  - Reduce instruction latency
    - Each instruction goes to WB asap
    - Eliminate need for forwarding paths
  - Eliminate some unnecessary stalls
    - E.g. slow FP instruction does not block independent integer instructions
- Disadvantages ??

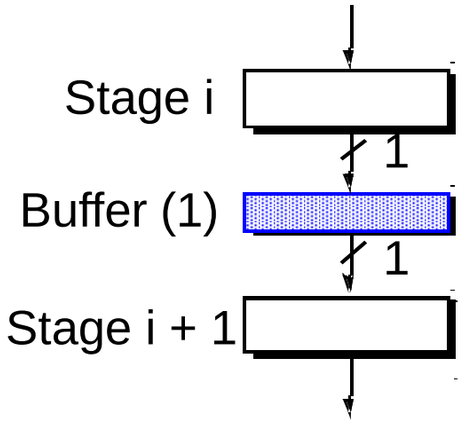
Power4 Diversified Pipelines



Dynamic Pipelines

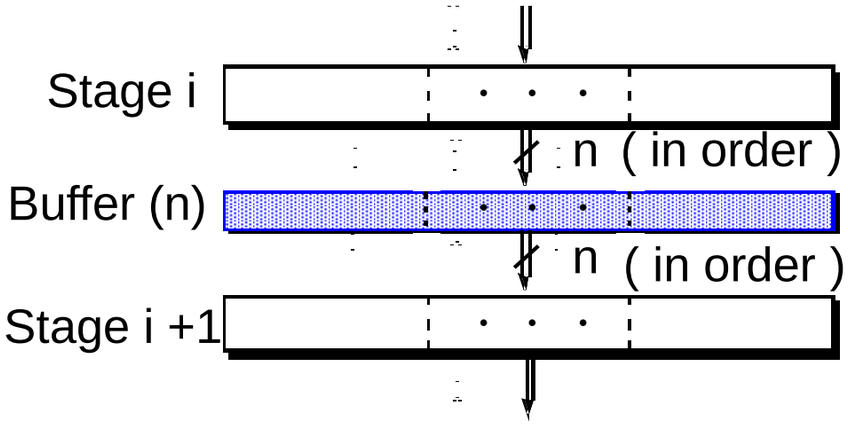


Designs of Inter-stage Buffers



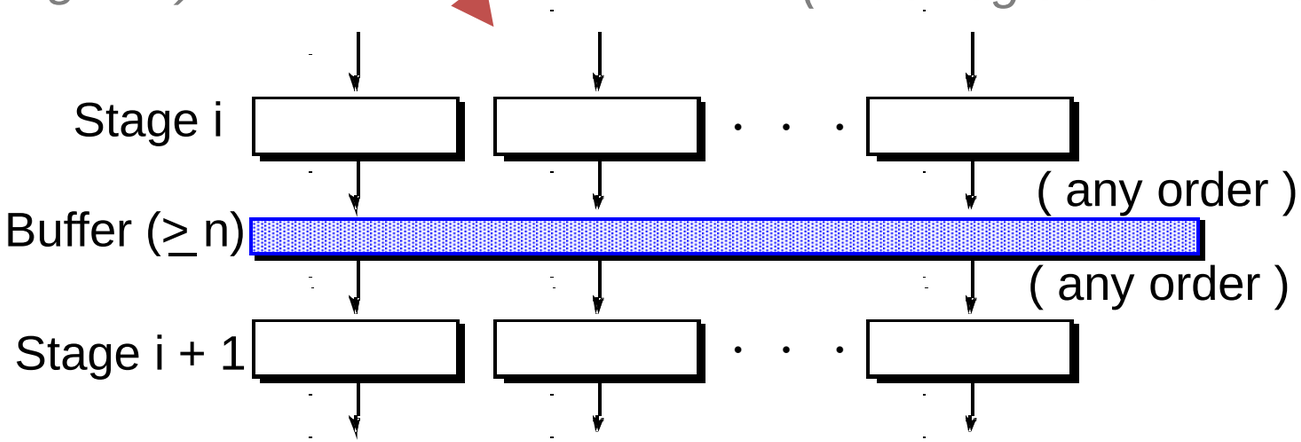
Scalar Pipeline Buffer

(simple register)



In-order Parallel Buffers

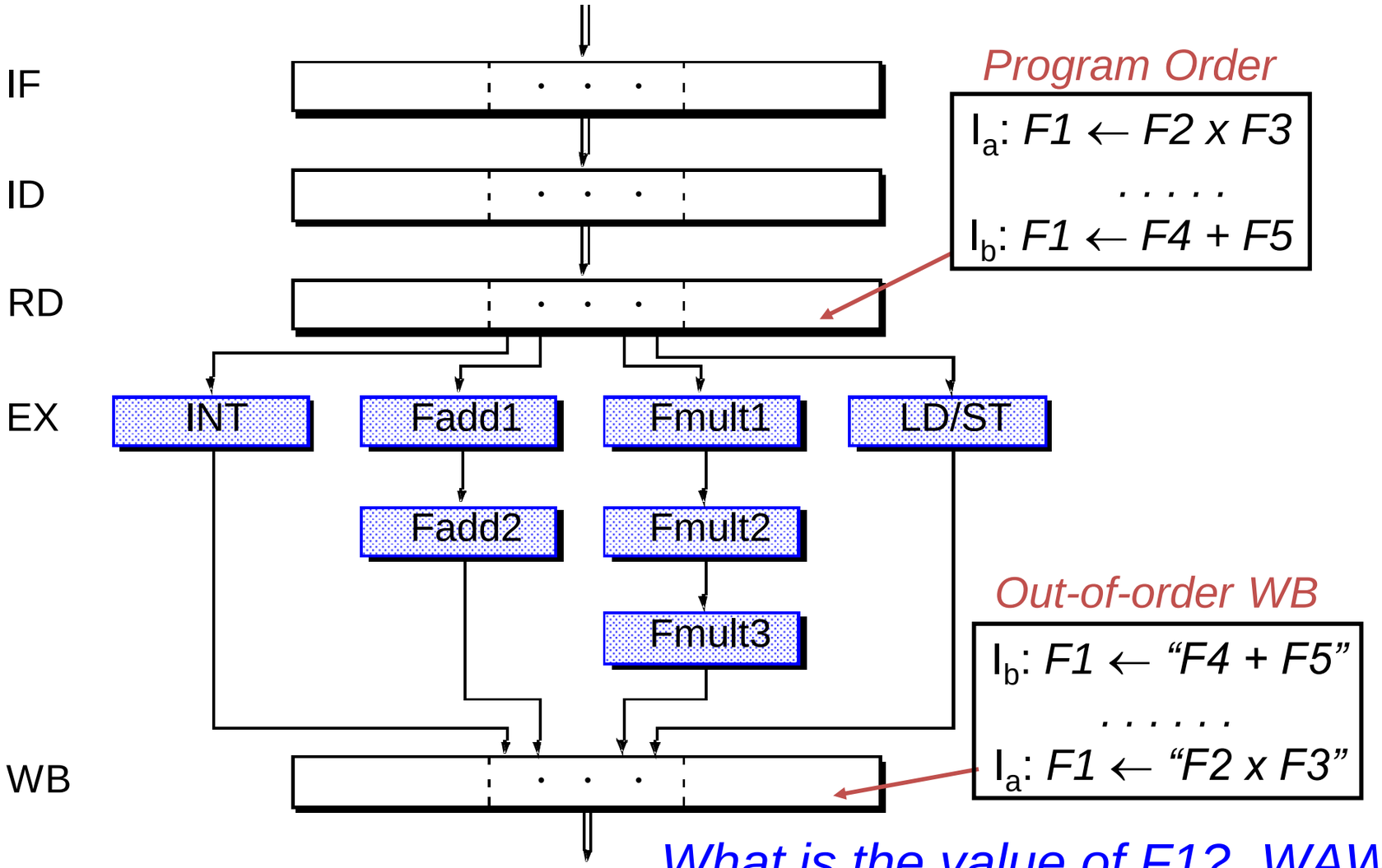
(wide-register or FIFO)



(multiported SRAM and CAM)

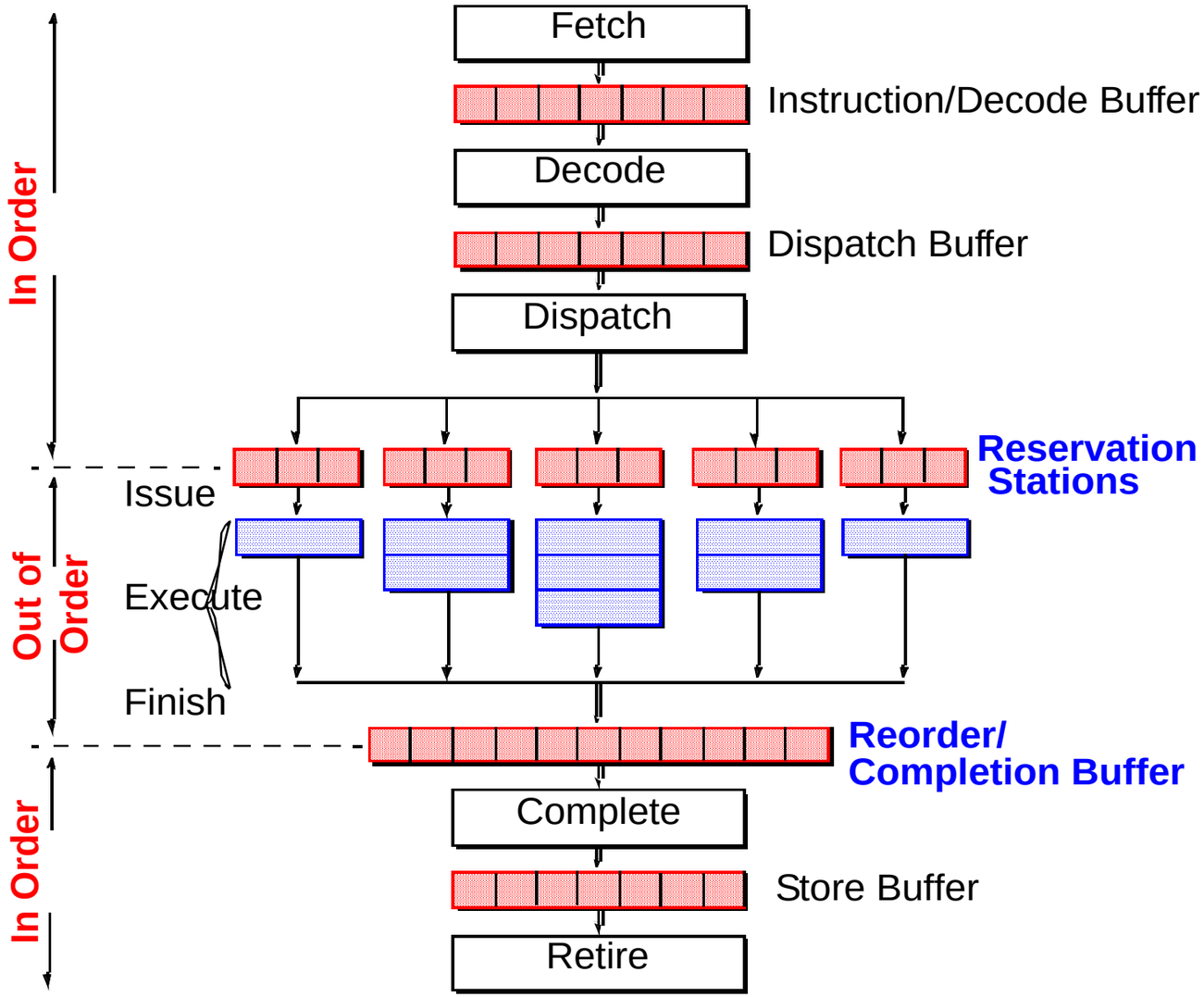
Out-of-order Pipeline Stages

The Challenges of Out-of-Order Execution

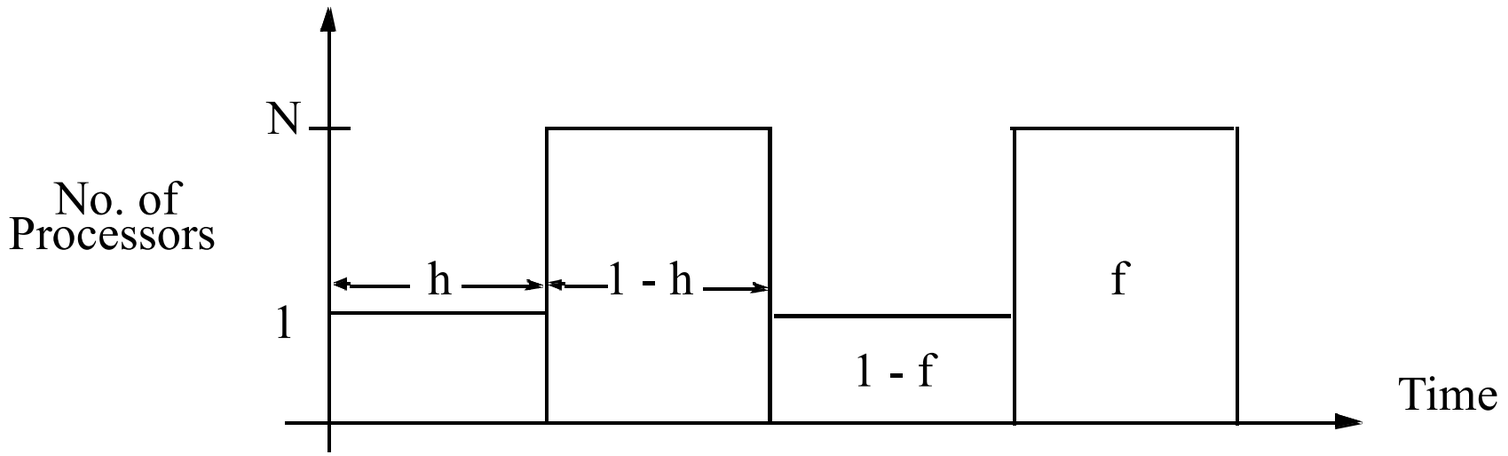


What is the value of F1? WAW!!!

Modern Superscalar Processor Organization



Amdahl's Law and Instruction Level Parallelism



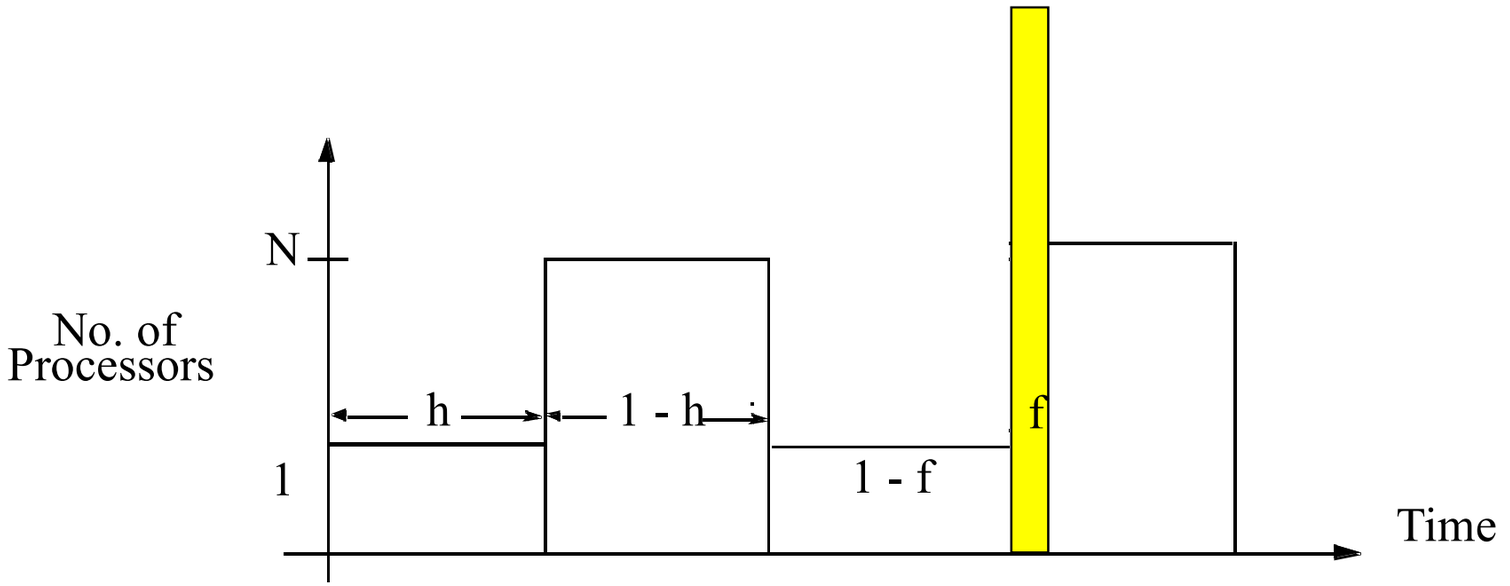
- h = fraction of time in serial code
- f = fraction that is vectorizable or parallelizable
- N = max speedup for f
- Overall speedup → →

$$Speedup = \frac{1}{(1-f) + \frac{f}{N}}$$

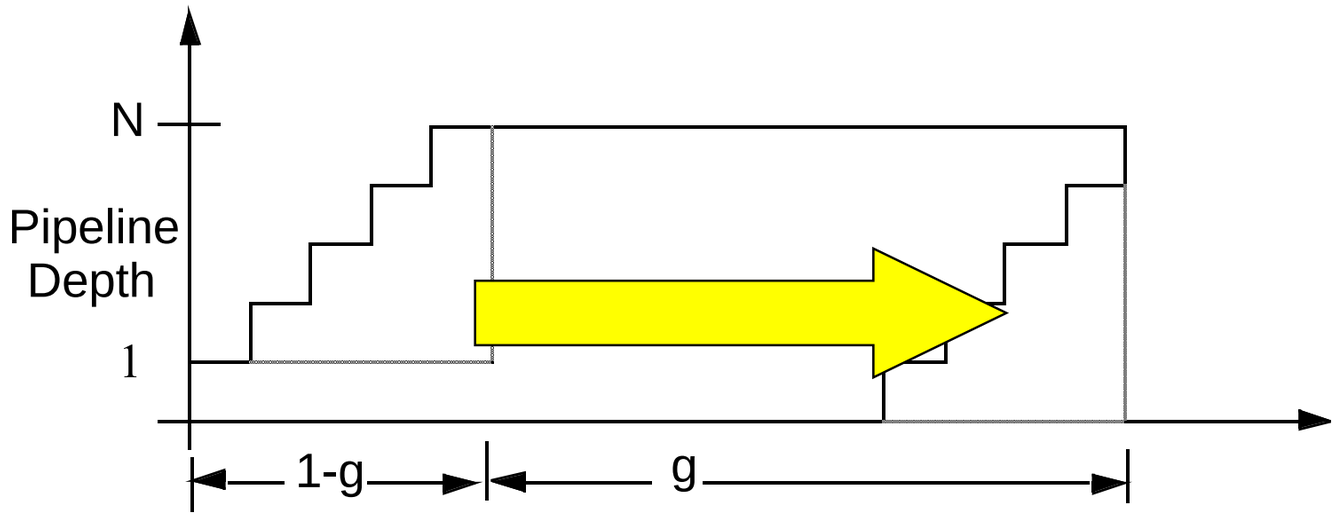
Revisit Amdahl's Law

- Sequential bottleneck
- Even if N is infinite
  - Performance limited by non-vectorizable portion (1-f)

$$\lim_{N\to\infty} \frac{1}{(1-f)+\frac{f}{N}} = \frac{1}{1-f}$$

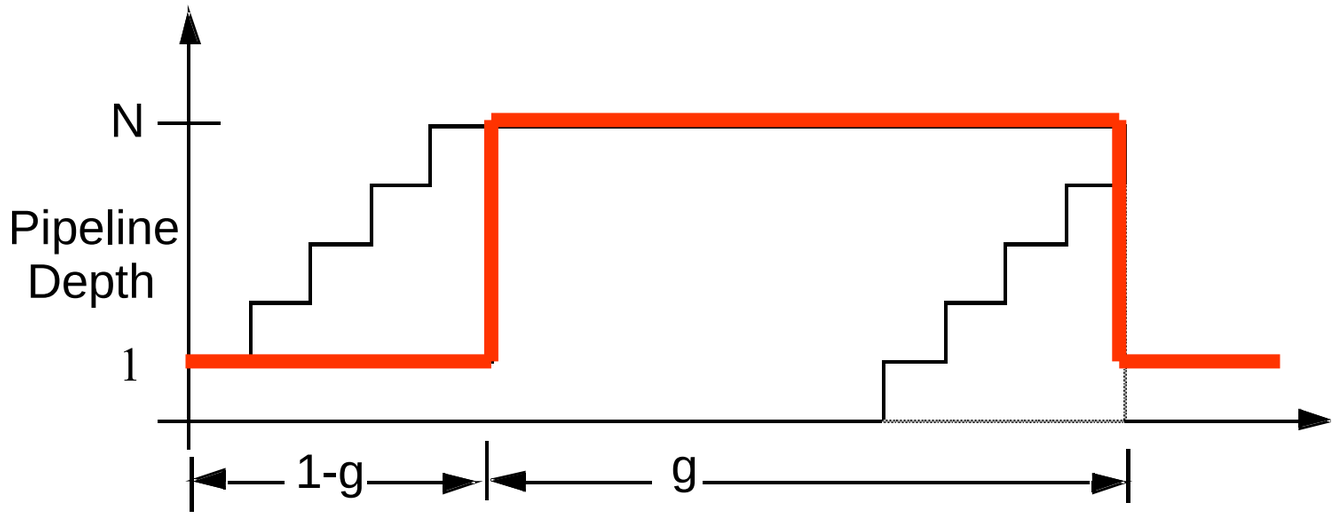


Pipelined Processor Performance Model



- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)

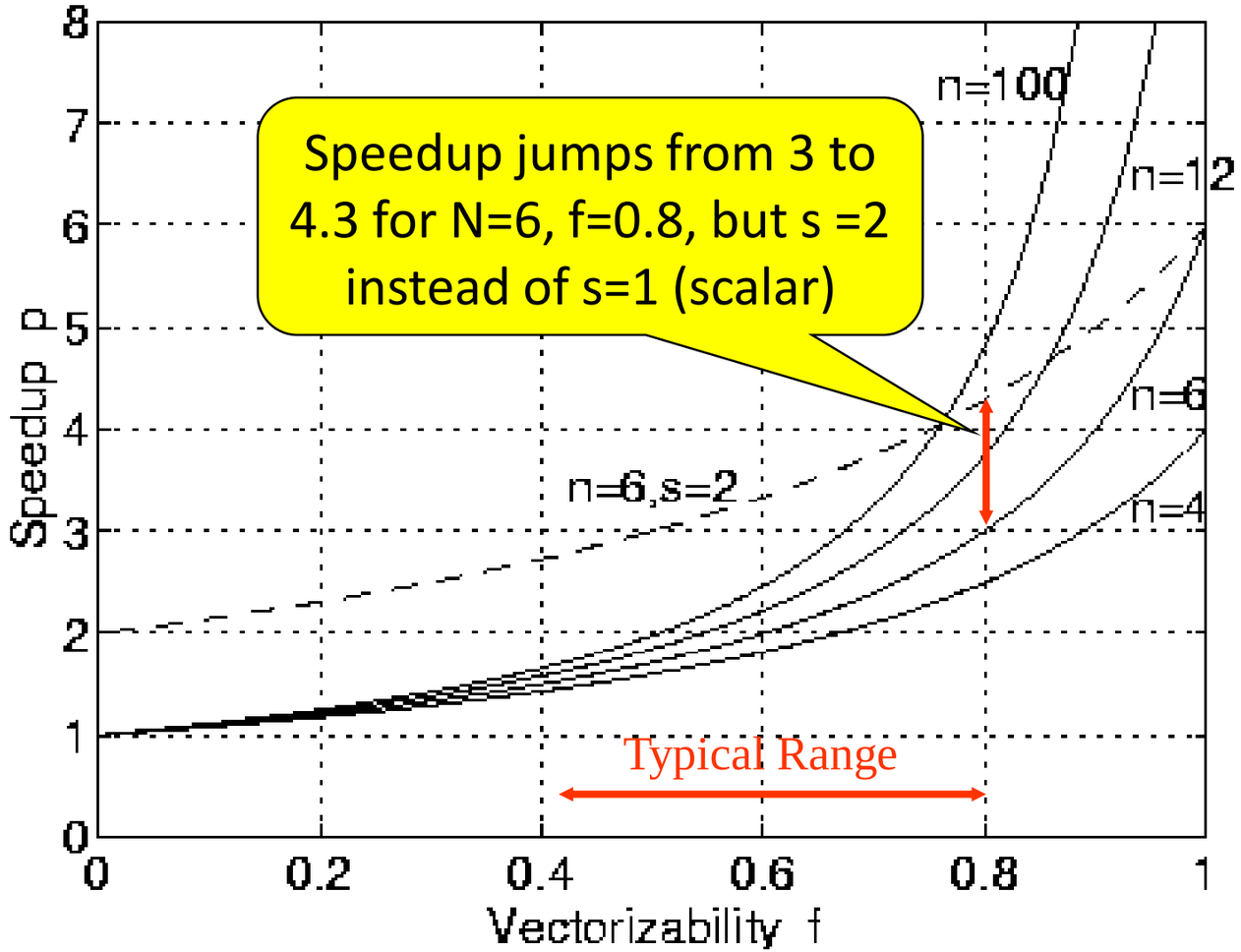
Pipelined Processor Performance Model



- "Tyranny of Amdahl's Law"
  - When g is even slightly below 100%, a big performance hit will result
  - Stalled cycles in the pipeline are the key adversary and must be minimized as much as possible
  - Can we somehow fill the pipeline bubbles (stalled cycles)?

[Tilak Agerwala and John Cocke, 1987]

Motivation for Superscalar Design



Superscalar Proposal

- Moderate the tyranny of Amdahl's Law
  - Ease the sequential bottleneck
  - More generally applicable
  - Robust (less sensitive to f)
  - Revised Amdahl's Law:

$$Speedup = \frac{1}{\frac{(1-f)}{S} + \frac{f}{N}}$$

The Ideas Behind Modern Superscalar Processors

- Superscalar or wide instruction issue
  Ideal IPC = n (CPI = 1/n)
- Diversified pipelines
  Different instructions go through different pipe stages
  Instructions go through needed stages only
- Out-of-order or data-flow execution
  Stall only on RAW hazards and structural hazards
- Speculation
  Overcome (some) RAW hazards through prediction

And it all relies on: **Instruction Level Parallelism (ILP)**
Independent instructions within sequential programs

Limits on Instruction Level Parallelism (ILP)

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

Iron Law of Processor Performance

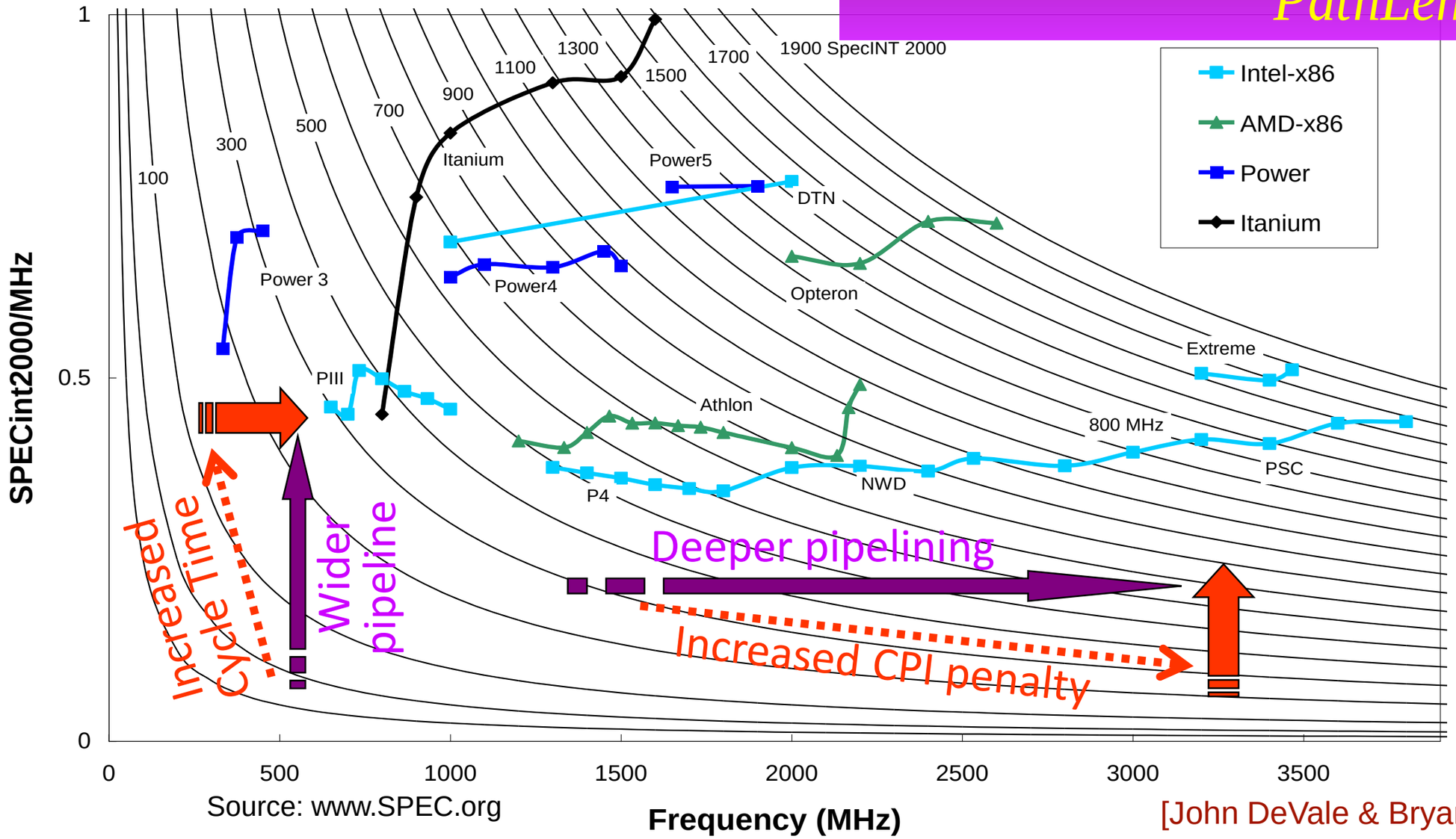
$$1/\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{(path length)}} \times \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{(CPI)}} \times \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{\text{(cycle time)}}$$

- In the 1980's (decade of **pipelining**):
  - CPI: 5.0 → 1.15
- In the 1990's (decade of **superscalar**):
  - CPI: 1.15 → 0.5 (best case)
- In the 2000's (decade of **multicore**):
  - Core CPI unchanged; chip CPI scales with #cores

Frequency vs. Parallelism

$$Performance_{CPU} = \frac{Frequency}{PathLength \times CPI}$$

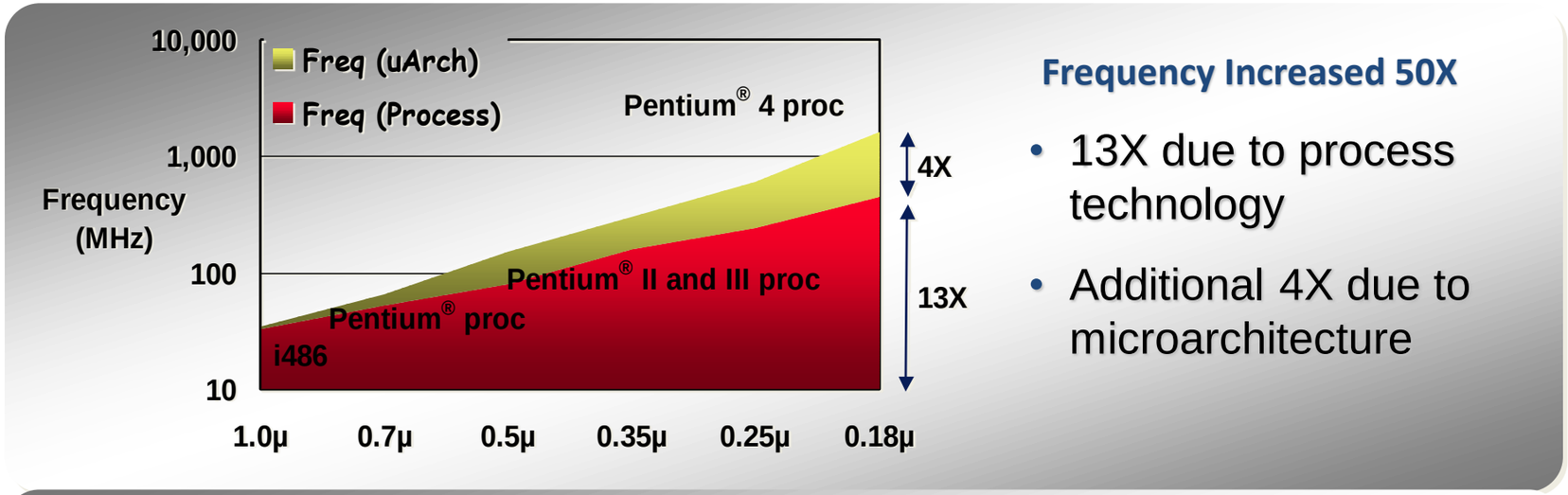


Source: www.SPEC.org

[John DeVale & Bryan Black, 2005]

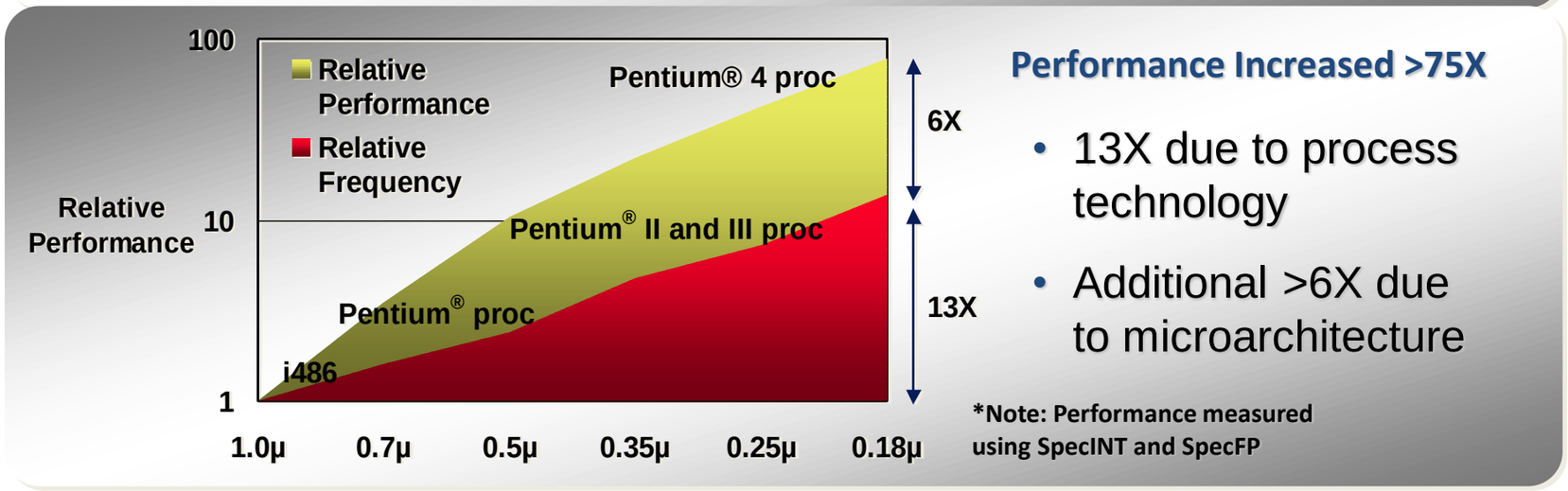
[Source: Intel Corporation]

Frequency and Performance Boost



Frequency Increased 50X

- 13X due to process technology
- Additional 4X due to microarchitecture



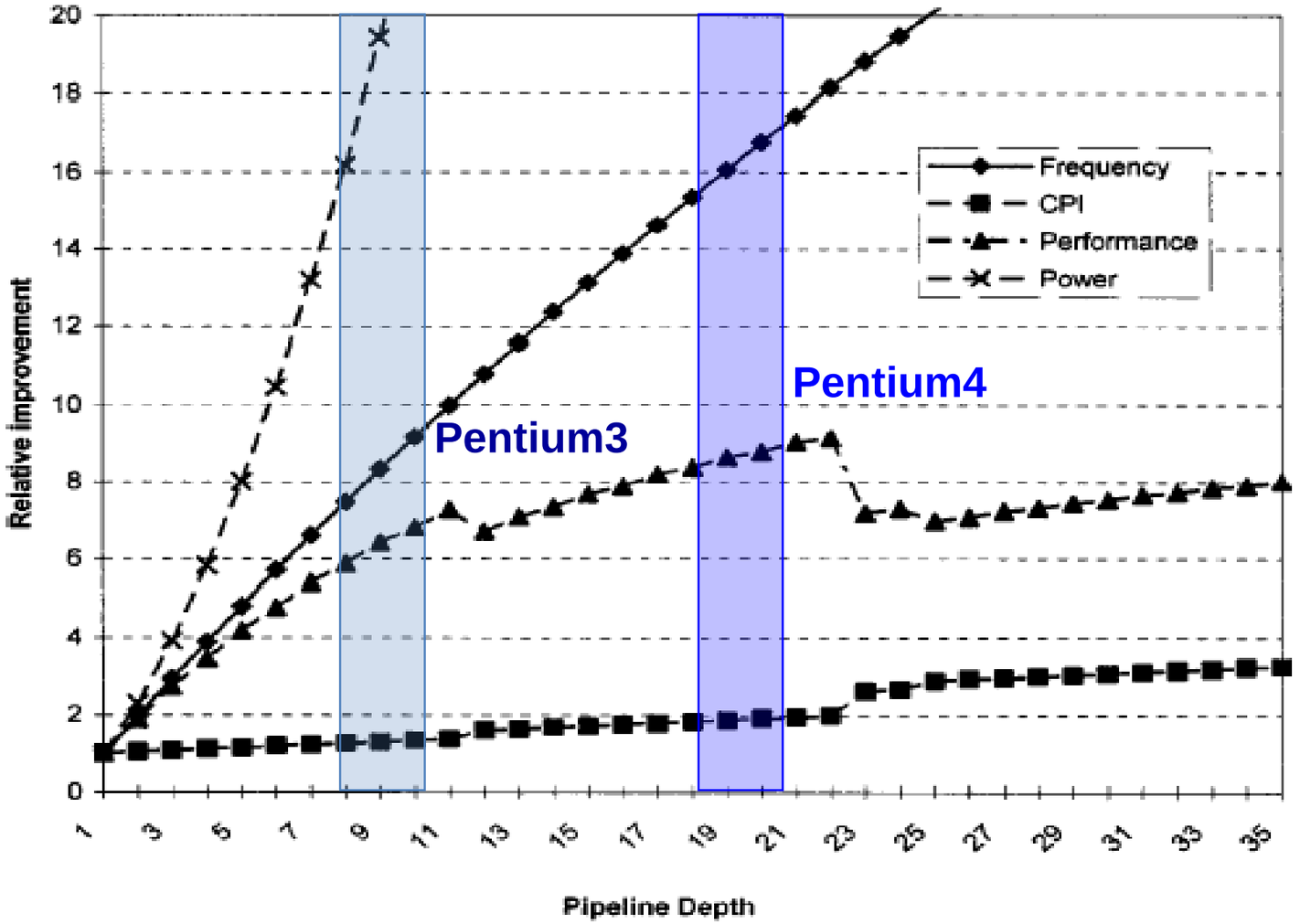
Performance Increased >75X

- 13X due to process technology
- Additional >6X due to microarchitecture

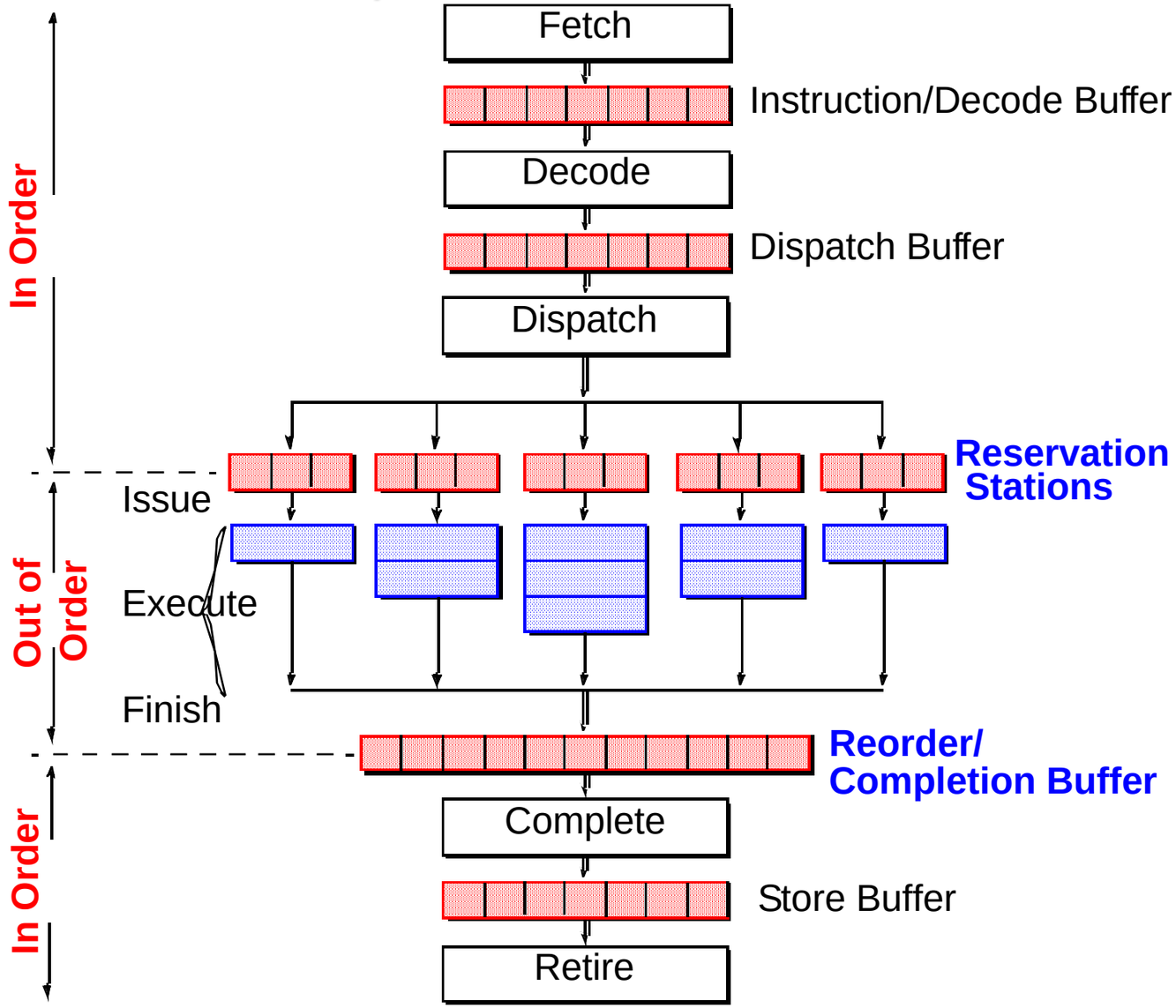
*Note: Performance measured using SpecINT and SpecFP

[Ed Grochowski, Intel, 1997]

Putting it All Together: Limits to Deeper Pipelines



Modern Superscalar Processor Organization



- Buffers provide decoupling
- In OOO designs they also facilitate (re-)ordering
- More details on specific buffers to follow

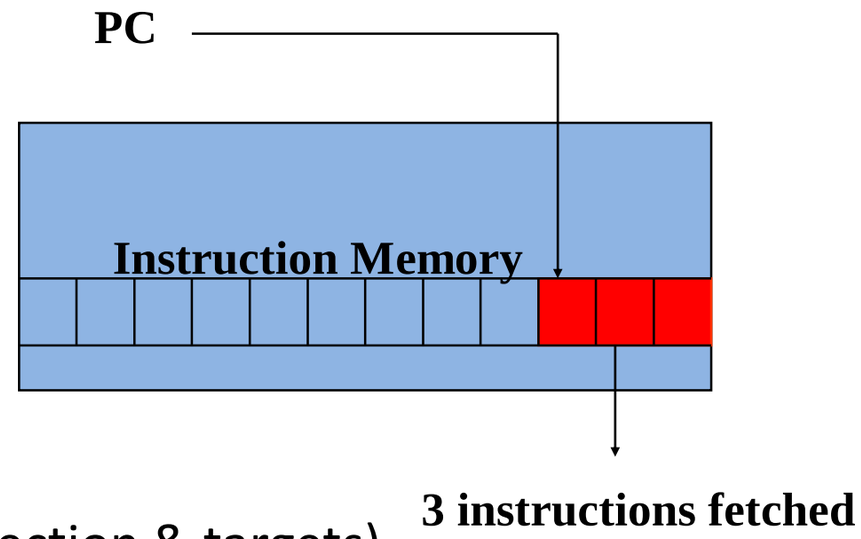
Superscalar Processor Implementation Issues

- Instruction fetching Issues
  - How do we maintain high bandwidth and accurate instruction delivery
- Instruction decoding Issues
- Instruction dispatching Issues
  - Register renaming
- Instruction execution Issues
  - Centralized vs distributed reservation stations
- Instruction completion and Retiring Issues
  - ROB, store queues, …

Instruction Fetch and Decode

- Goal: given a PC, fetch up to N instructions to execute
  - Supply the pipeline with maximum number of useful instructions per cycle
  - The fetch stage **sets the maximum possible performance** (IPCmax)

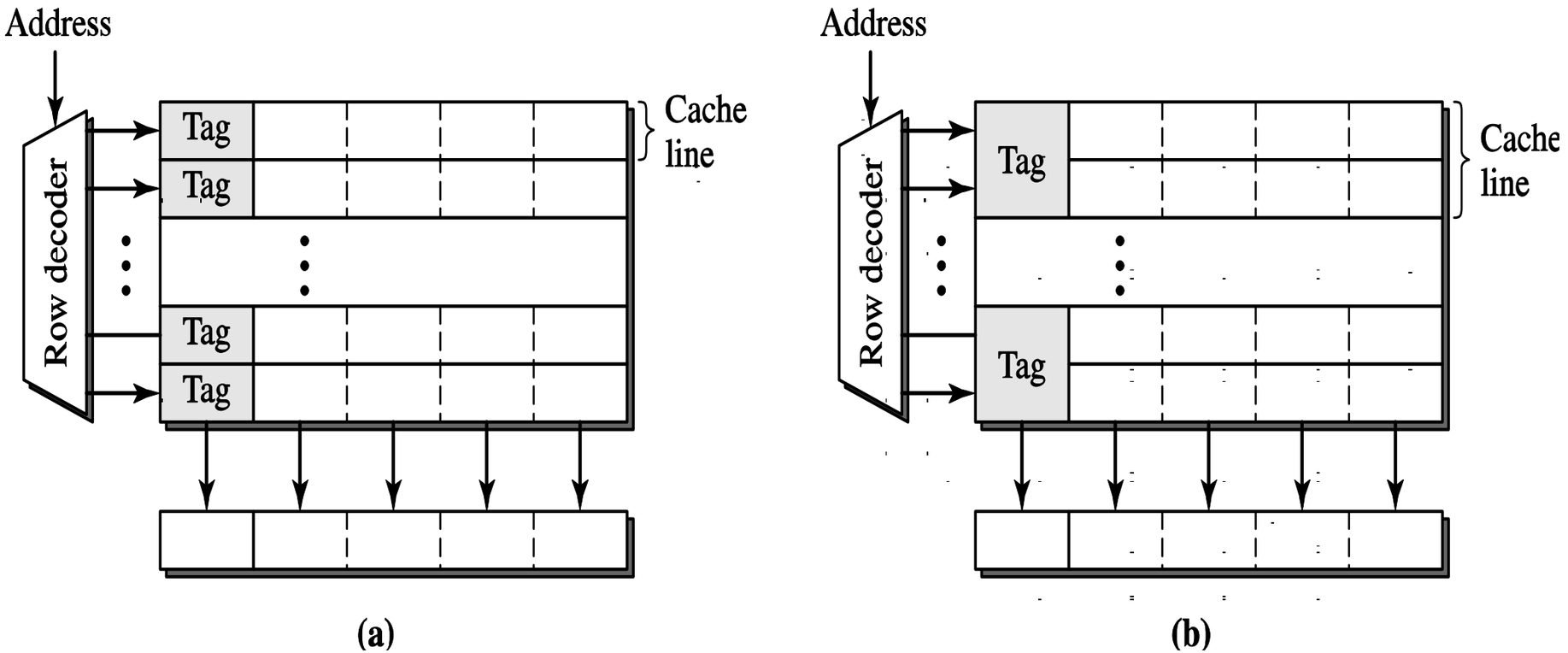
- Impediments
  - Instruction cache misses
  - Instruction alignment
  - Complex instruction sets
    - CISC (x86, 390, etc)
  - Branches and jumps
    - Determining the instruction address (branch direction & targets)
    - Will start in this lecture and finish in next one…



Wide Instruction Fetches

- For a N-way superscalar, need ≥ N-way fetching
  - Otherwise, N-way ILP can never be achieved
  - Sometimes, wider than n-way fetch helps – why?
- Implementation: wide port to I-cache
  - Read many/all words from I-cache
  - Select those from current PC to first taken branch
- Reducing cache misses (remember?)
  - Separate I-cache
  - Larger block size, larger cache size (any problems?)
  - Higher associativity
  - 2nd-level cache, prefetching

Instruction Cache Organization

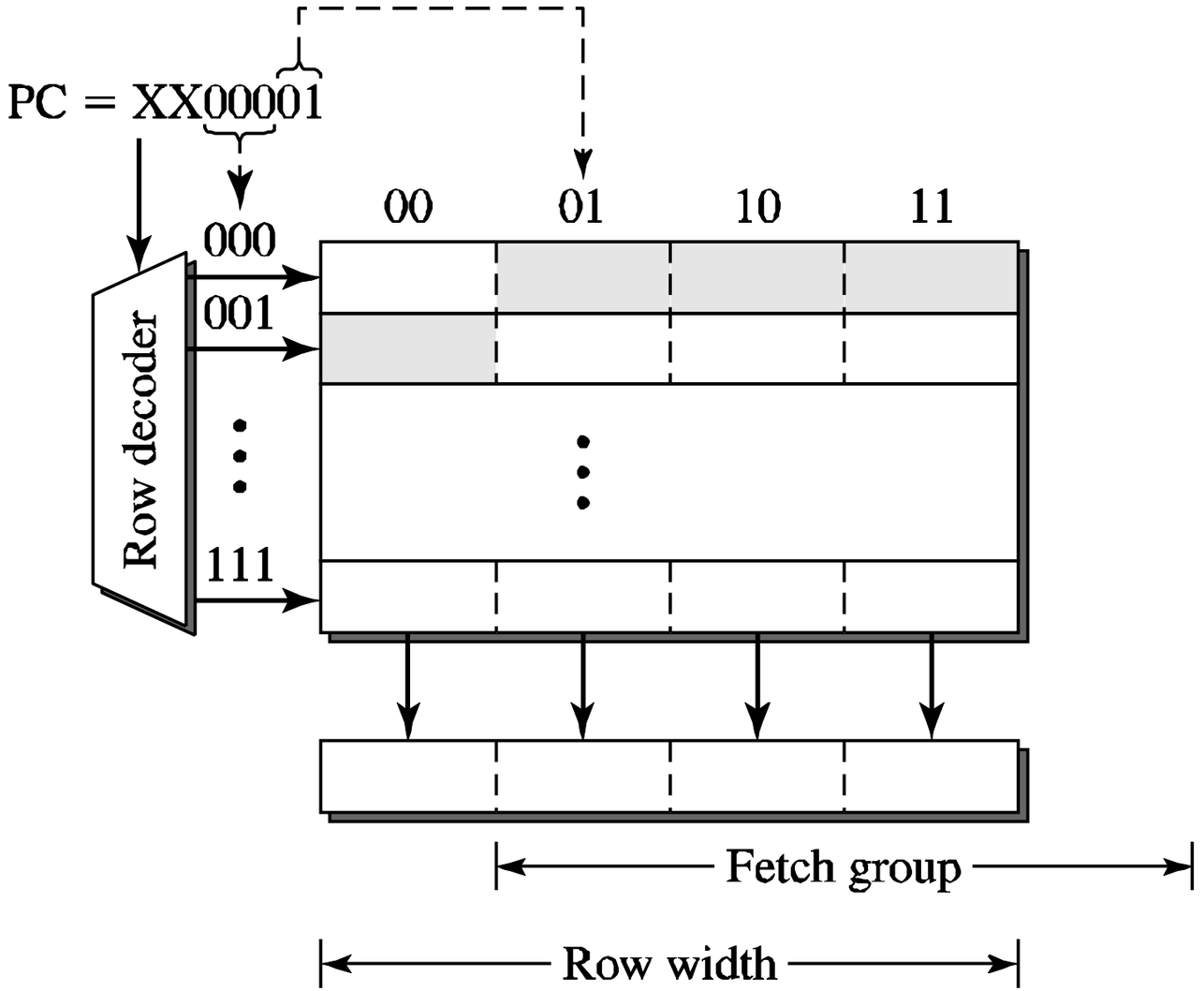


1 cache line = 1 physical row

1 cache line = 2 physical rows

- These are logical views: In practice, tags & data may be stored separately

The Fetch Alignment Problem



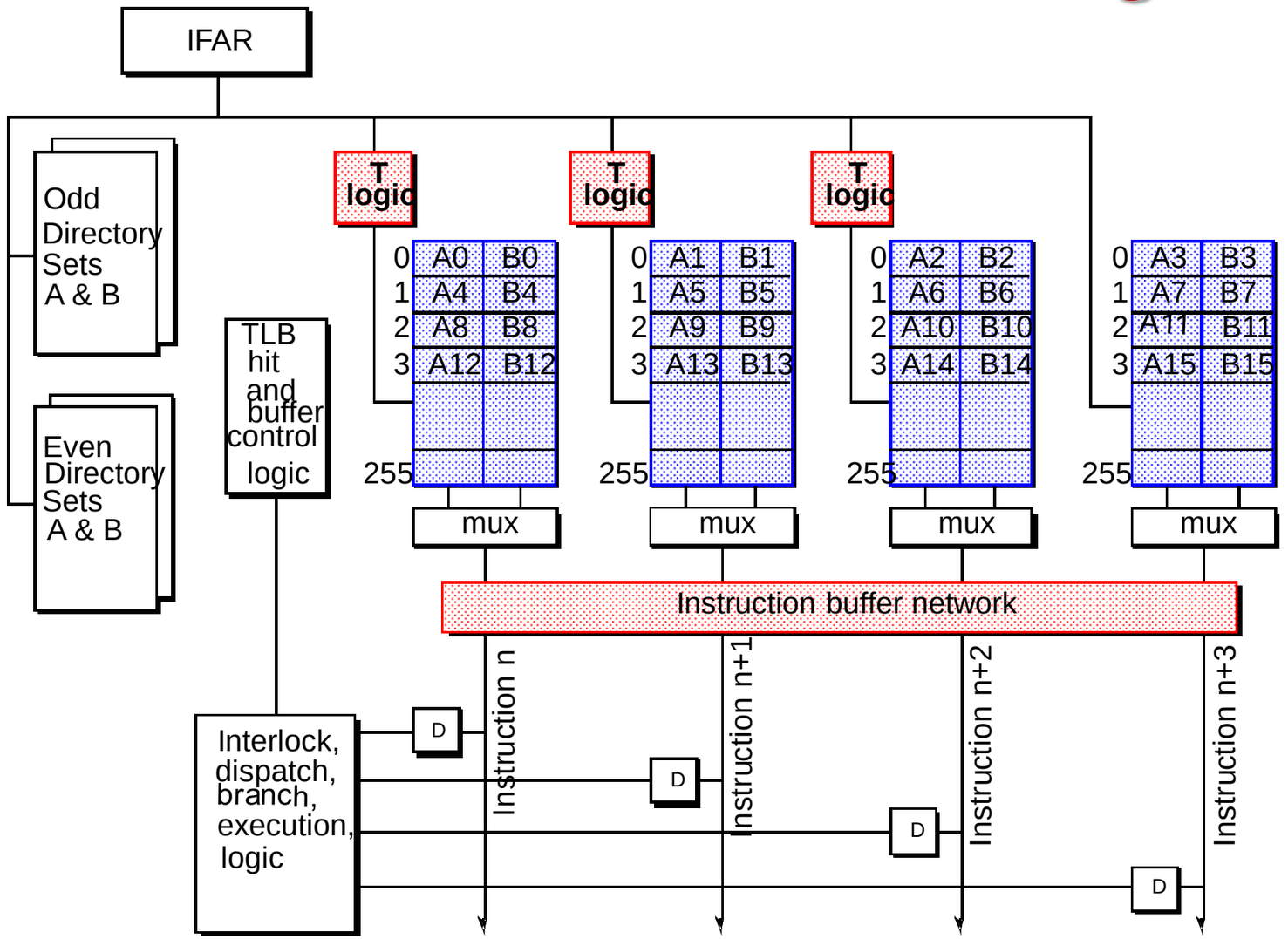
Solving the Alignment Problem

- Software solution: align taken branch targets to I-cache row starts
  - Effect on code size?
  - Effect on the I-cache miss rate?
  - What happens when we go to the next chip?

- Hardware solution
  - Detect (mis)alignment case
  - Allow access of multiple rows (current and sequential next)
    - True multi-ported cache, over-clocked cache, multi-banked cache, …
    - Or keep around the cache line from previous access
      - Assuming a large basic block
  - Collapse the two fetched rows into one instruction group

IBM RS/6000 I-Cache and Fetch (auto alignment)

- 2-way set associative (A and B sets) I-Cache; (8) 256-instruction SRAM modules
- 16 instruction per cache line (64 bytes)



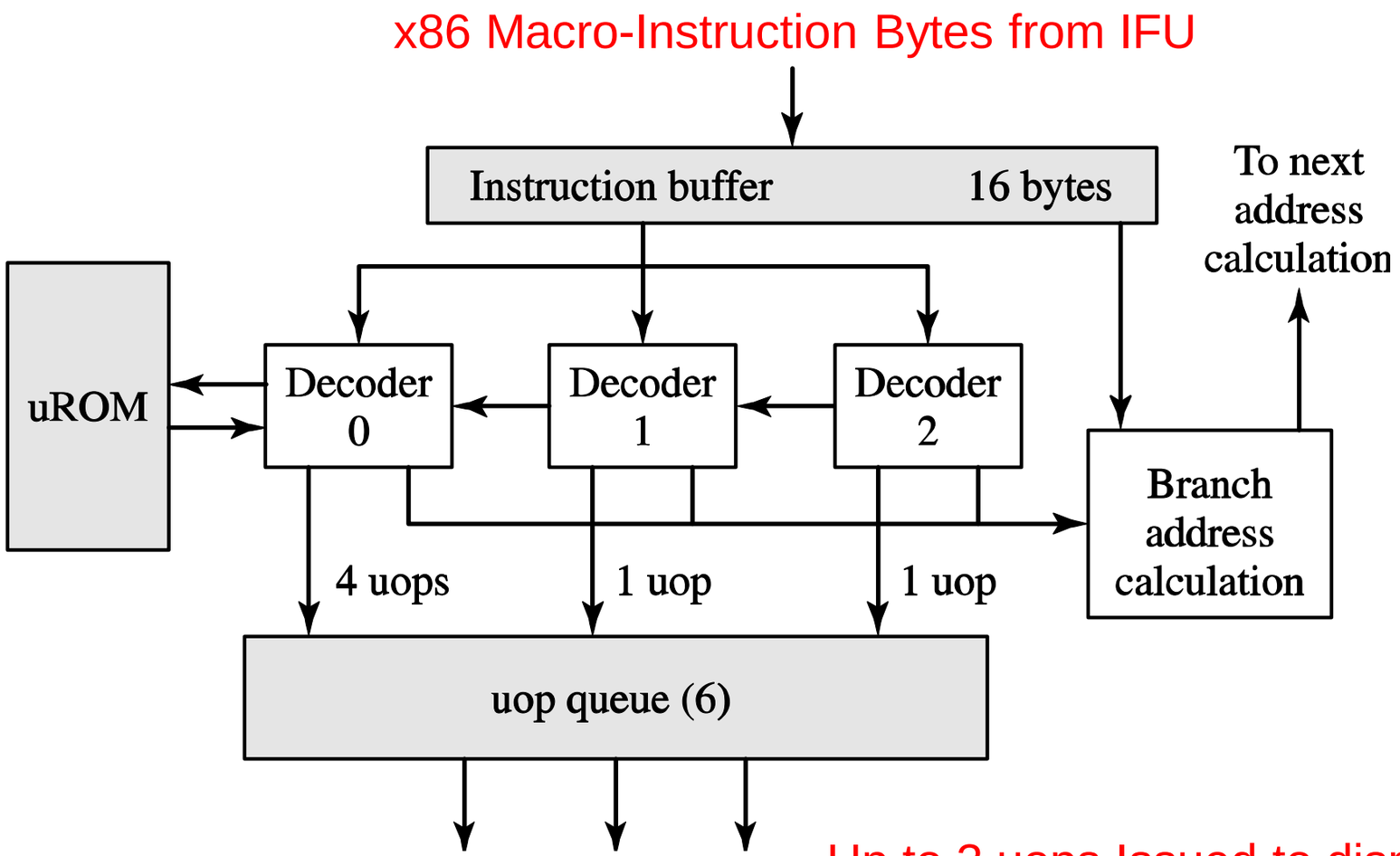
Cache Line Underutilization

- What if we have <N instructions between two taken branches?
  - Or predicted taken branches?
- Solution: read multiple cache lines
  - Current and predicted next cache line
  - Merge instructions from two cache lines using a collapsing buffer
  - Question: how do we get the predicted next cache line address?
    - Need two predictions (PCs) per cycle
- Easier cases to handle
  - Intra-cache line forward & backward branches

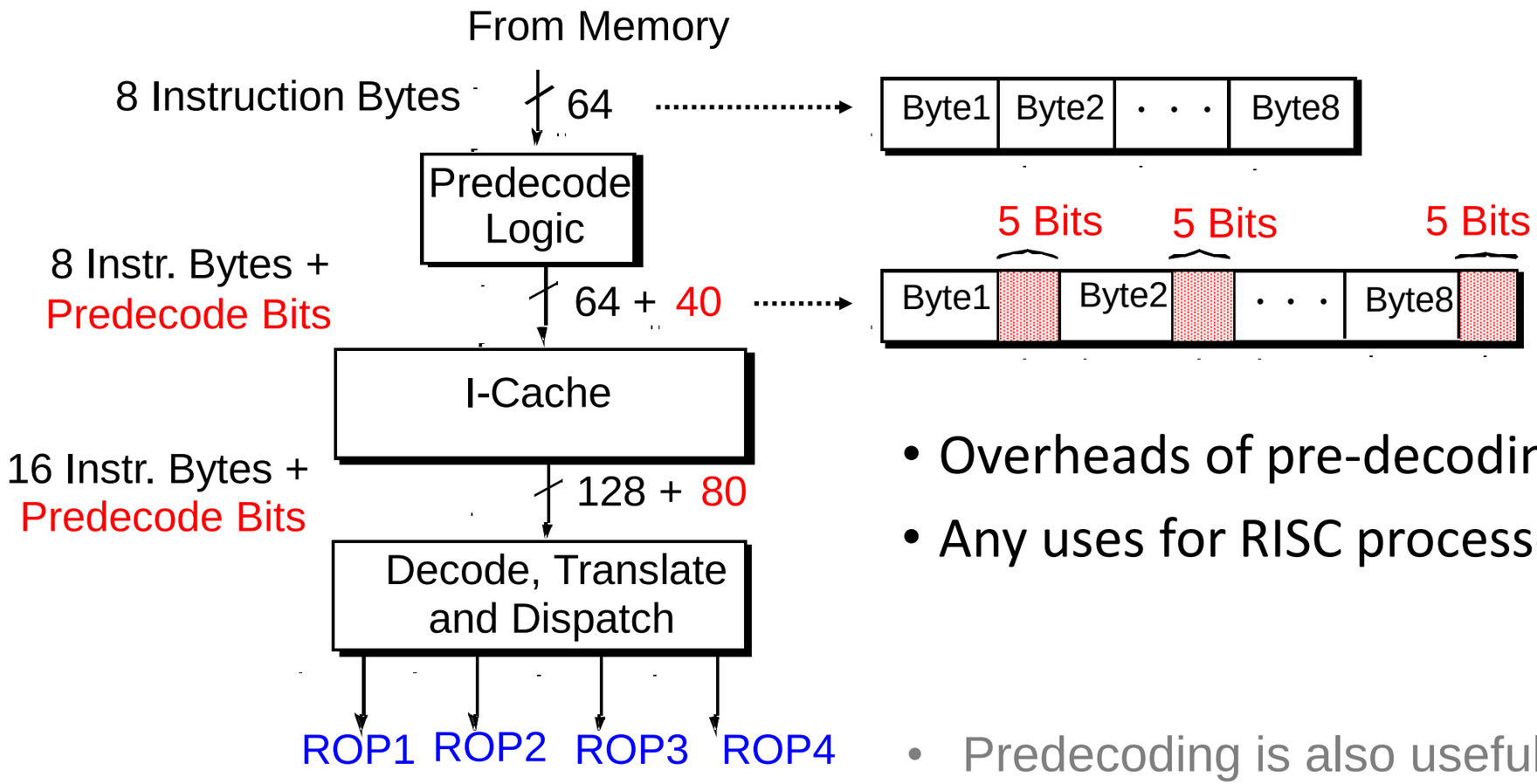
Instruction Decoding Issues

- Primary decoding tasks
  - Identify individual instructions for CISC ISAs
  - Determine instruction types (especially branches)
    - Drop instructions after (predicted) taken branches
    - Potentially restart the fetch pipeline from target address
  - Determine dependences between instructions
- Two important factors
  - Instruction set architecture (RISC vs. CISC)
    - Determines decoding difficulty
  - Pipeline width
    - Sets the number of comparators for dependence detection

Intel Pentium Pro (P6) Fetch/Decode



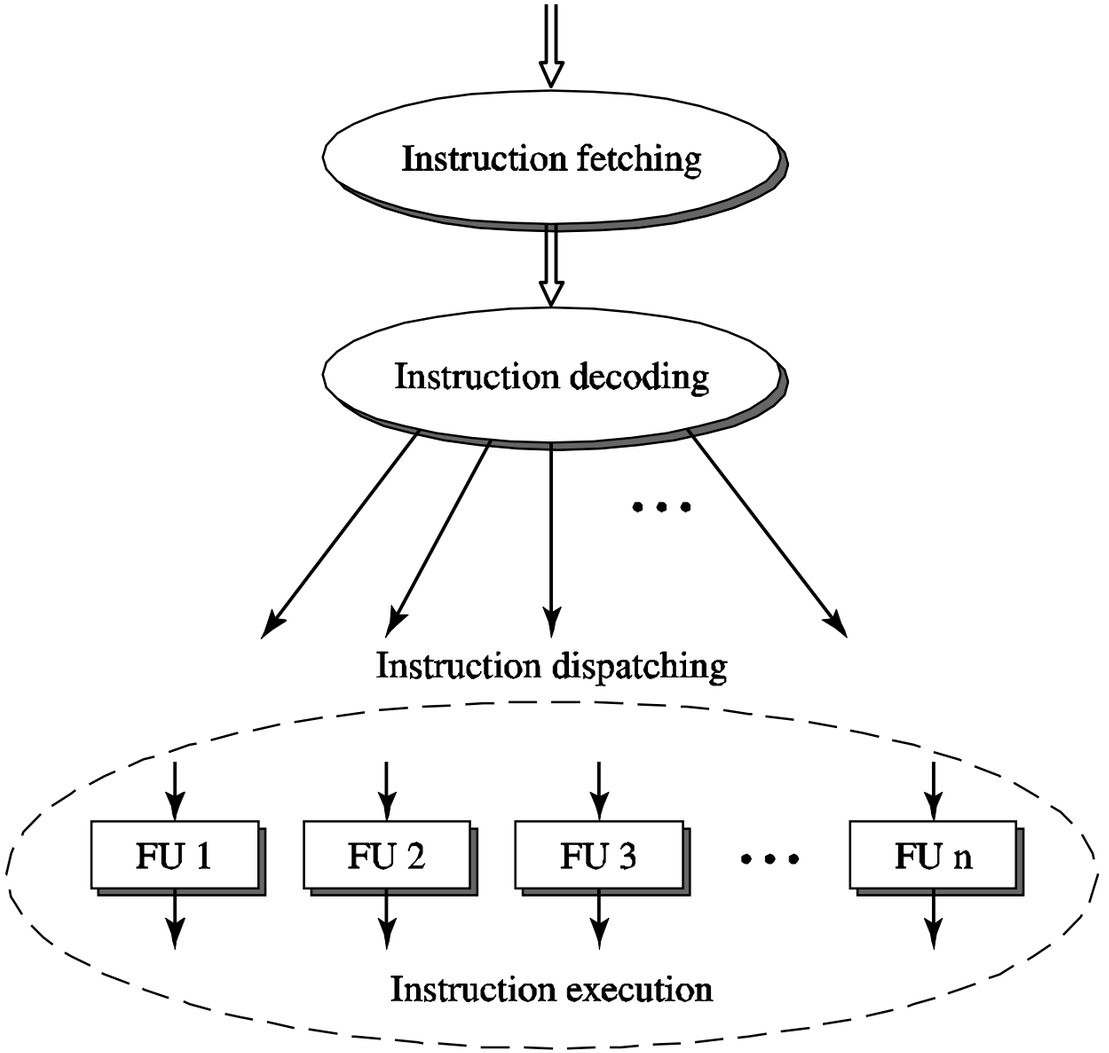
I-Cache Pre-decoding in the AMD K5



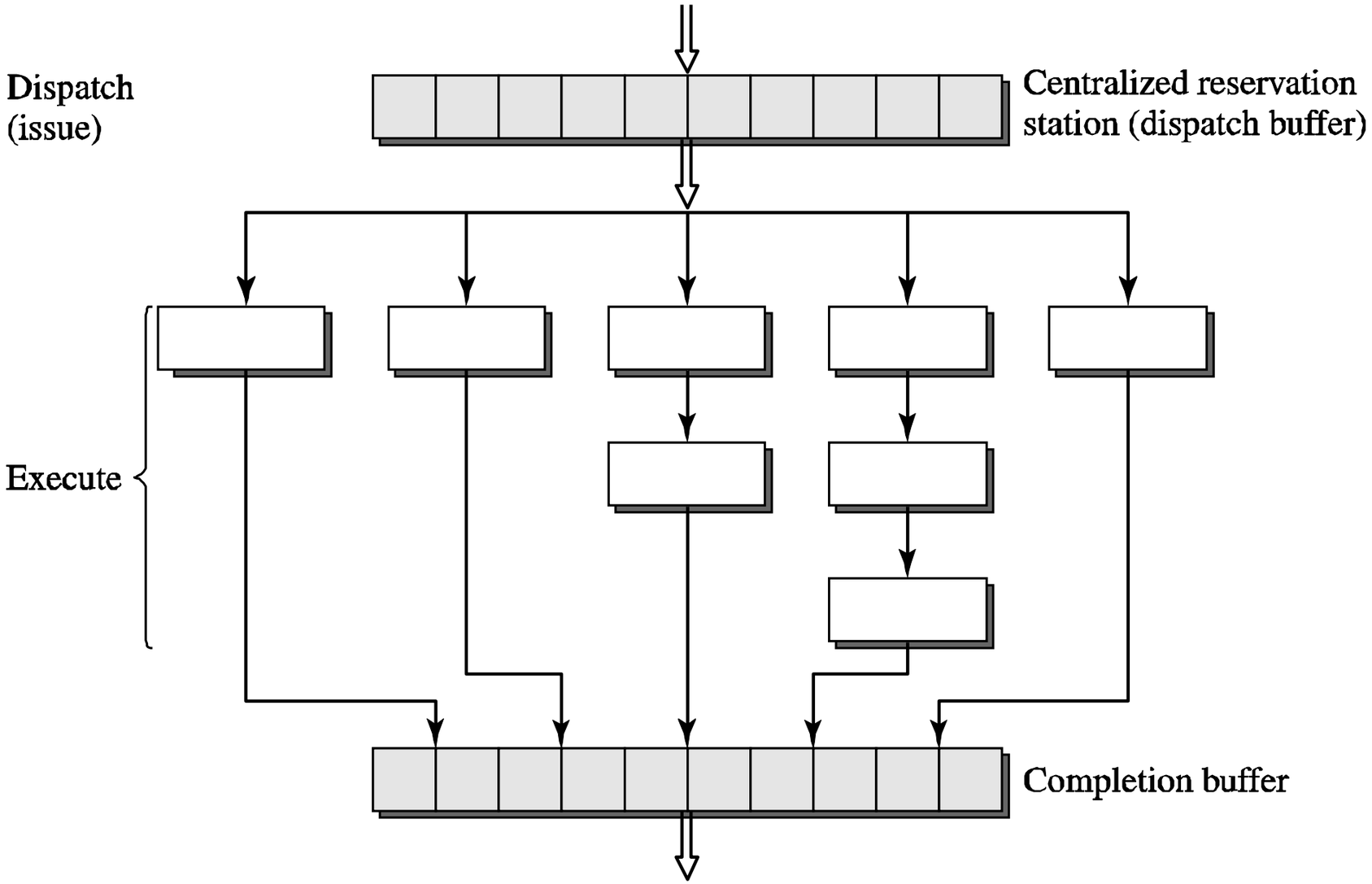
- Overheads of pre-decoding?
- Any uses for RISC processors?

- Predecoding is also useful for RISC
- Cost: cache size, refill time

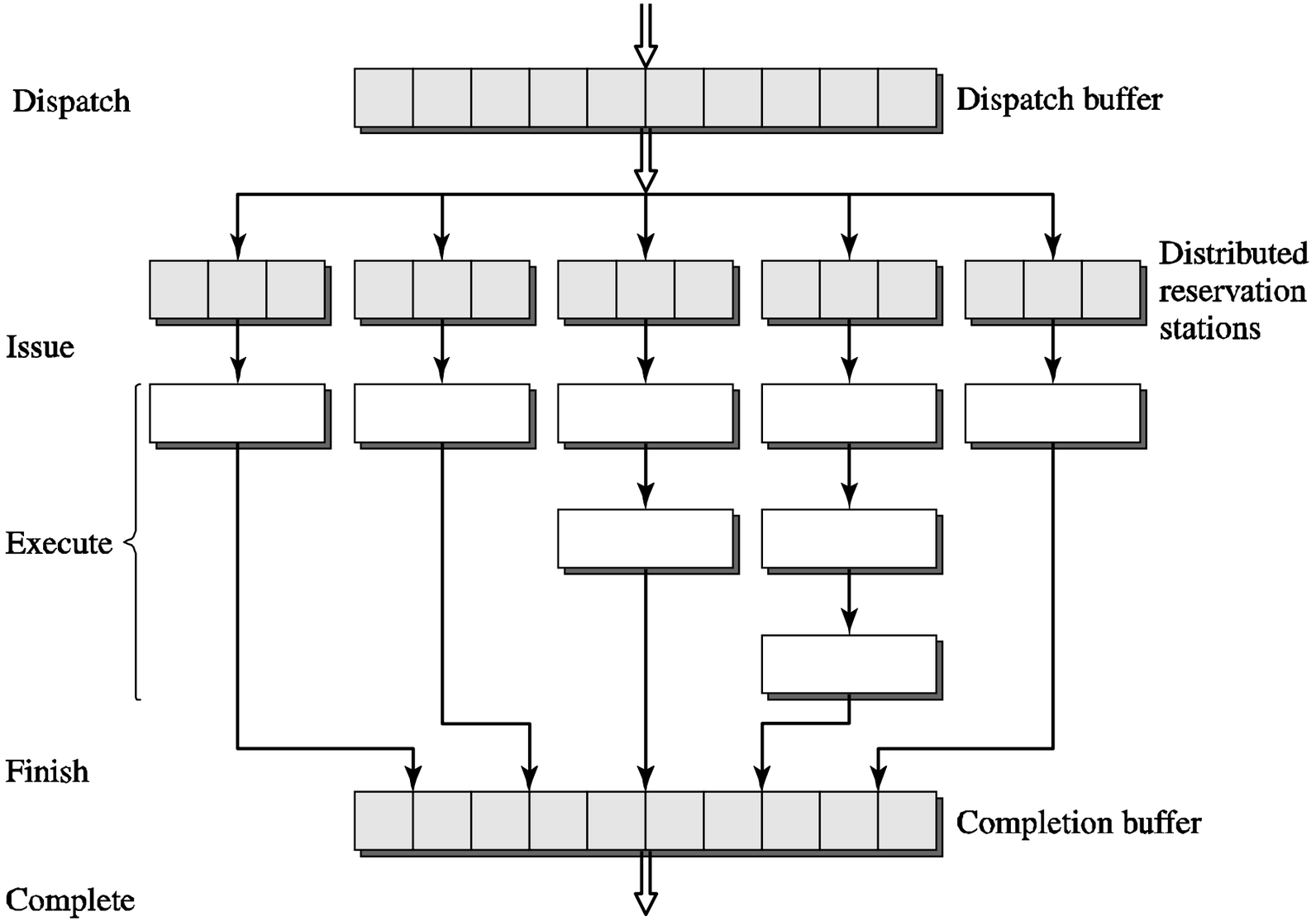
Instruction Dispatch and Issue .



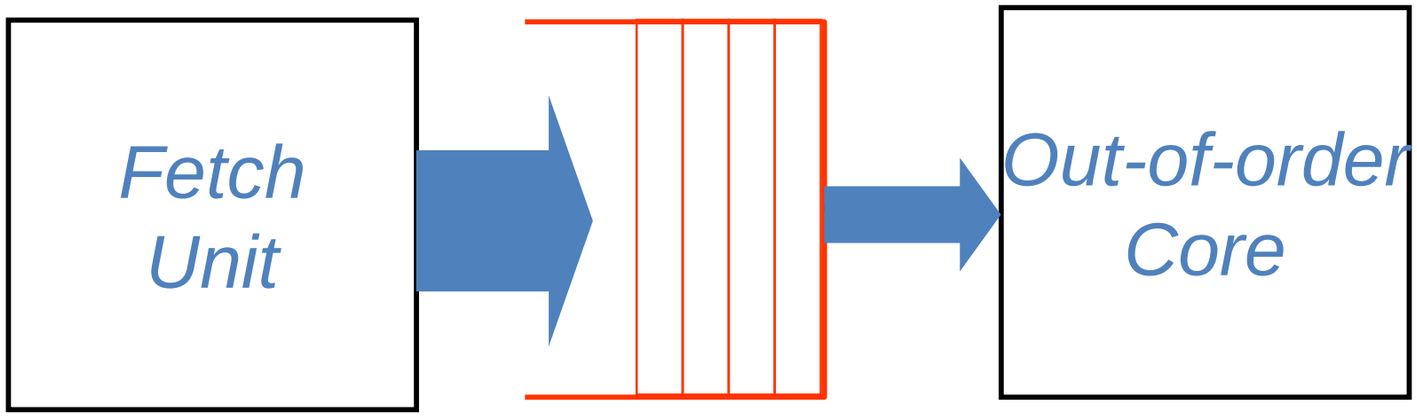
Centralized Reservation Station



Distributed Reservation Stations



Instruction Fetch Buffer



- Smooth out the rate mismatch between fetch and execution
  - Neither the fetch bandwidth nor the execution bandwidth is consistent
- Fetch bandwidth should be higher than execution bandwidth
  - We prefer to have a stockpile of instructions in the buffer to hide cache miss latencies.
  - *This requires both raw cache bandwidth + control flow speculation*

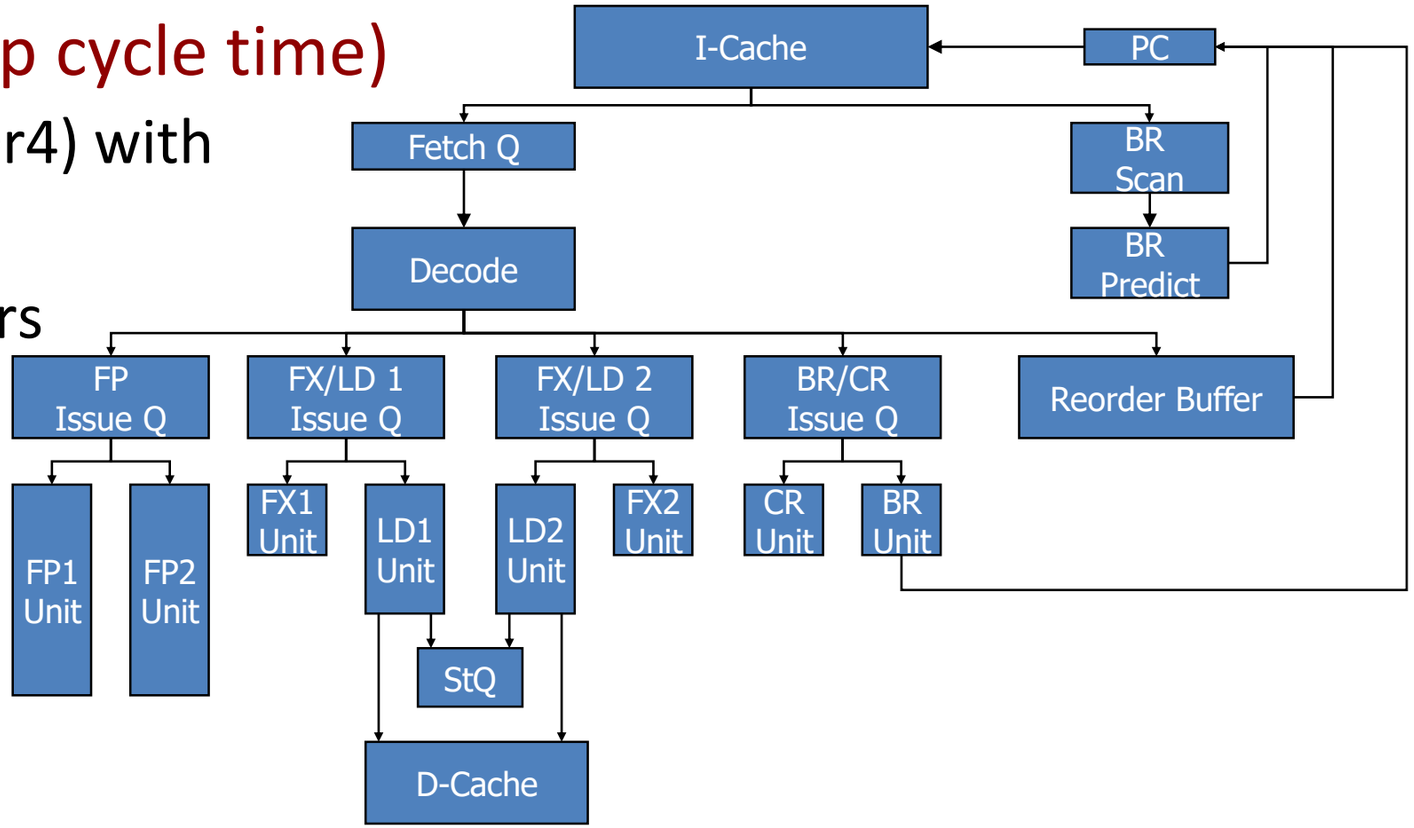
Instruction Execute

- Current trends
  - More parallelism ← forwarding/bypass very challenging
  - Deeper pipelines
  - More diversity

- Functional unit types
  - Integer
  - Floating point
  - Load/store ← most difficult to make parallel
  - Branch
  - Specialized units (media)

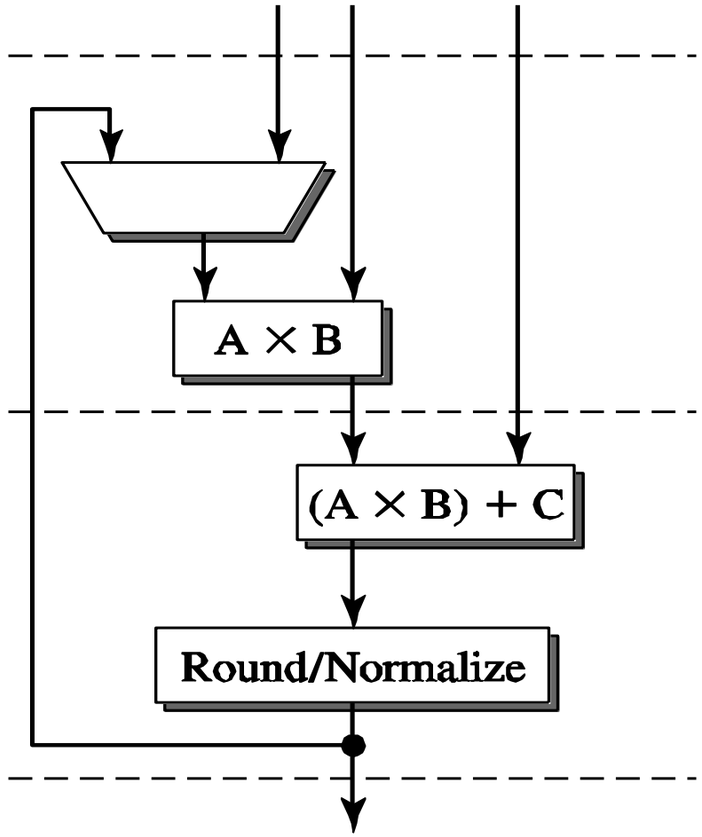
Forwarding/Bypass Networks

- $O(n^2)$ interconnect from/to FU inputs and outputs
- Associative tag-match to find operands
- Solutions (hurt IPC, help cycle time)
  - Use RF only (IBM Power4) with no bypass network
  - Decompose into clusters (Alpha 21264)

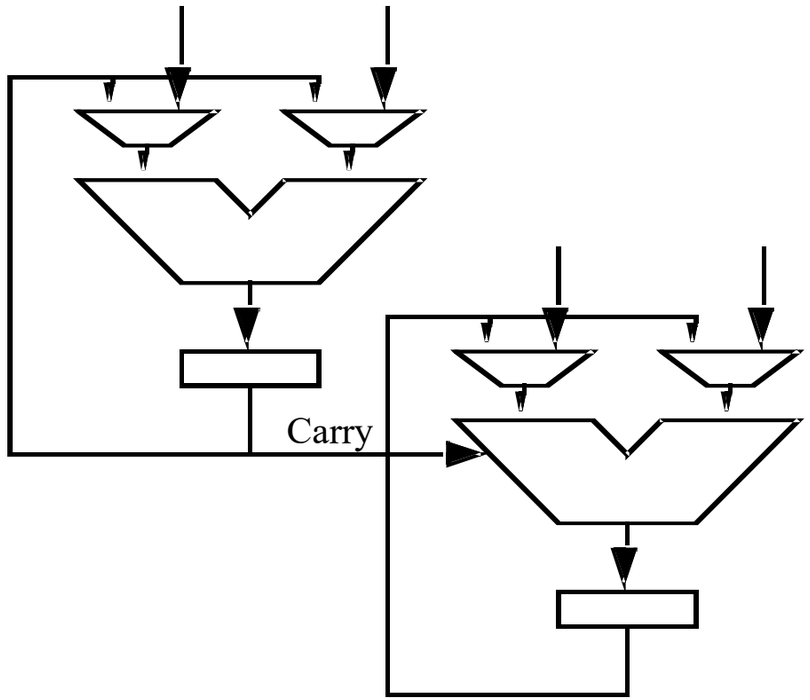


Specialized Functional Units

- FP multiply-accumulate
  R = (A x B) + C
- Doubles FLOP/instruction
- Lose RISC instruction format symmetry:
  – 3 source operands
- Widely used



Specialized Functional Units in Pentium 4



- Intel Pentium 4 staggered adders
  - *Fireball*
- Run at 2x clock frequency
- Two 16-bit bitslices
- Dependent ops execute on half-cycle boundaries
- Full result not available until full cycle later

Instruction Complete and Retire

- Out-of-order execution
  - ALU instructions
  - Load/store instructions
- In-order completion/retirement
  - Precise exceptions
  - Memory coherence and consistency
- Solutions
  - Reorder buffer
  - Store buffer
  - Load queue snooping (later)

Exceptional Limitations

- Precise exceptions: when exception occurs on instruction i
  - Instruction i has not modified the processor state (regs, memory)
  - All older instructions have fully completed
  - No younger instruction has modified the processor state (regs, memory)
- How do we maintain precise exception in a simple pipelined processor?
- What makes precise exceptions difficult on a
  - In-order diversified pipeline?
  - Out-of-order pipeline?

Precise Exceptions and OOO Processors

- Solution: force instructions to update processor state in-order
  - Register and memory updates done in order
  - Usually called instruction retirement or graduation or commitment
- Implementation: Re-Order Buffer (ROB)
  - A FIFO for instruction tracking
  - 1 entry per instruction
    - PC, register/memory address, new value, ready, exception
- ROB algorithm
  - Allocate ROB entries at dispatch time in-order (FIFO)
  - Instructions update their ROB entry when they complete
  - Examine head of ROB for in-order retirement
    - If done retire, otherwise wait (this forces order)
    - If exception, flush contents of ROB, restart from instruction PC after handler

Re-Order Buffer Issues

- Problem
  - Already computed results must wait in the reorder buffer when they may be needed by other instructions which could otherwise execute.
  - Data dependent instructions must wait until the result has been committed to register
- Solution
  - Forwarding from the re-order buffer
    - Allows data in ROB to be used in place of data in registers
  - Forwarding implementation 1: search ROB for values when registers read
    - Only latest entry in ROB can be used
    - Many comparators, but logic is conceptually simple
  - Forwarding implementation 2: use score-board to track results in ROB
    - Register scoreboard notes if latest value in ROB and # of ROB entry
    - Don't need to track FU any more; an instruction is fully identified by ROB entry

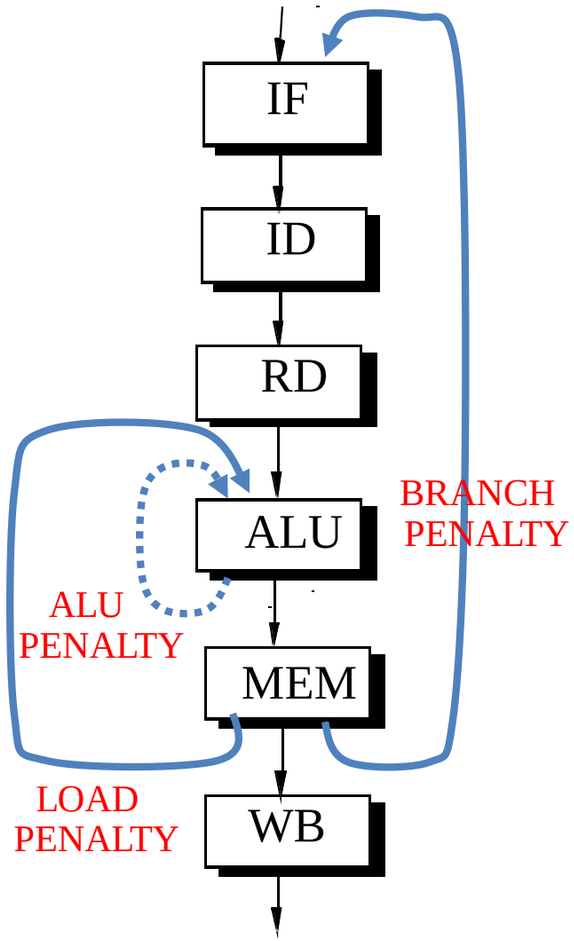
ROB Alternatives: History File

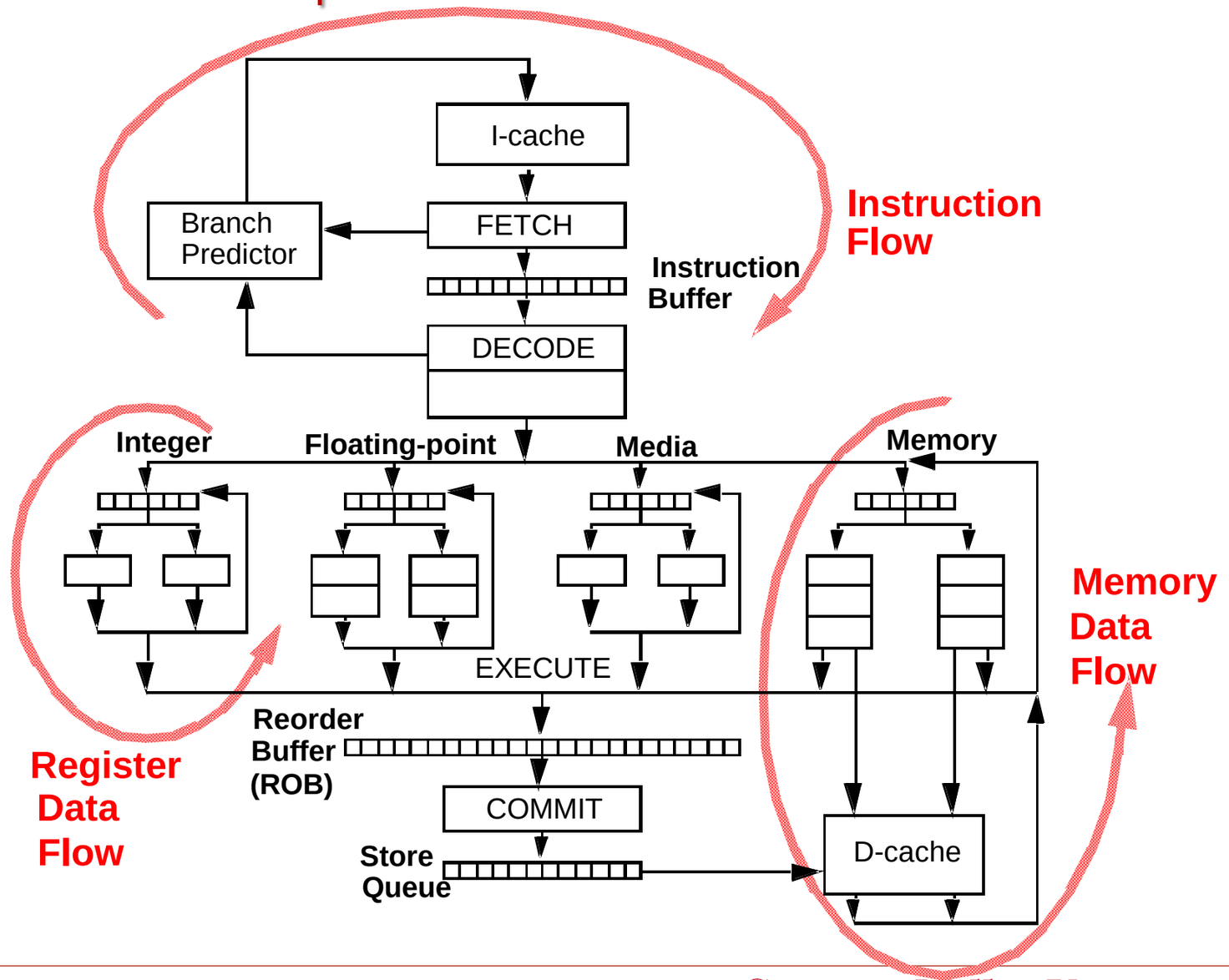
- A FIFO operated similarly to the ROB but logs old register values
  - Entry format just like ROB
- Algorithm
  - Entries allocated in-order at dispatch
  - Entries updated out-of-order at completion time
    - Destination register updated immediately
    - Old value of register noted in re-order buffer
  - Examine head of history file in-order
    - If no exception just de-allocate
    - If exception, reverse history file and undo all register updates before flushing
- Advantage: no need for separate forwarding from ROB
- Disadvantage: slower recovery from exceptions

ROB Alternatives: Future File

- Use two separate register files:
  - Architectural file ⇒ Represents sequential execution.
  - Future file ⇒ Updated immediately upon instruction execution and used as the working file.
- Algorithm
  - When instruction reaches the head of ROB, it is committed to the architectural file
  - On an exception changes are brought over from the architectural file to the future file based on which instructions are still represented in the ROB
- Advantage: no need for separate forwarding from ROB
- Disadvantage: slower recovery from exceptions

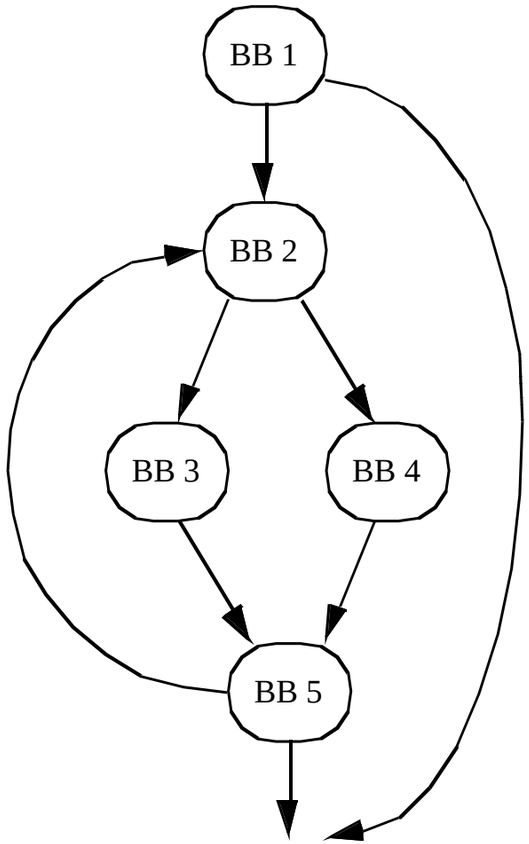
Three Impediments to Superscalar Performance





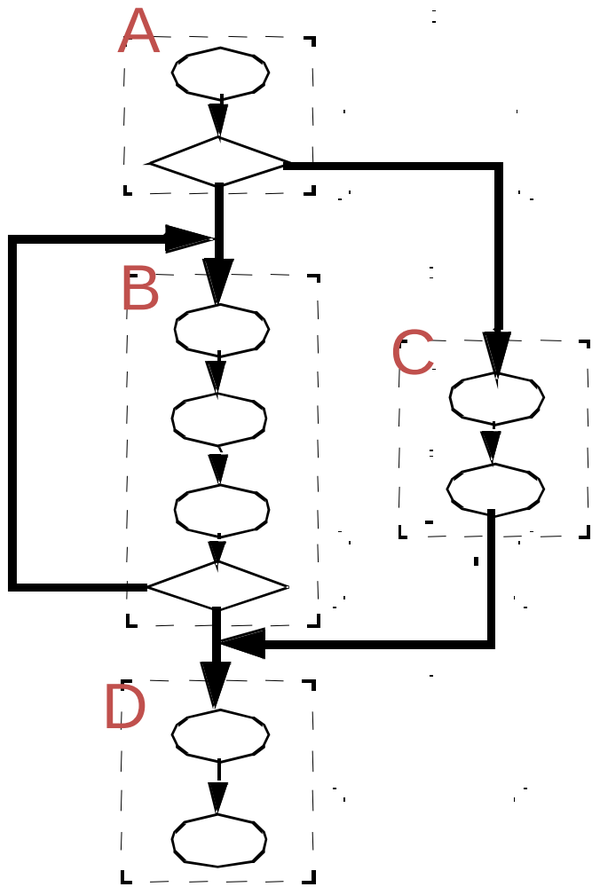
Control Flow Graph (CFG)

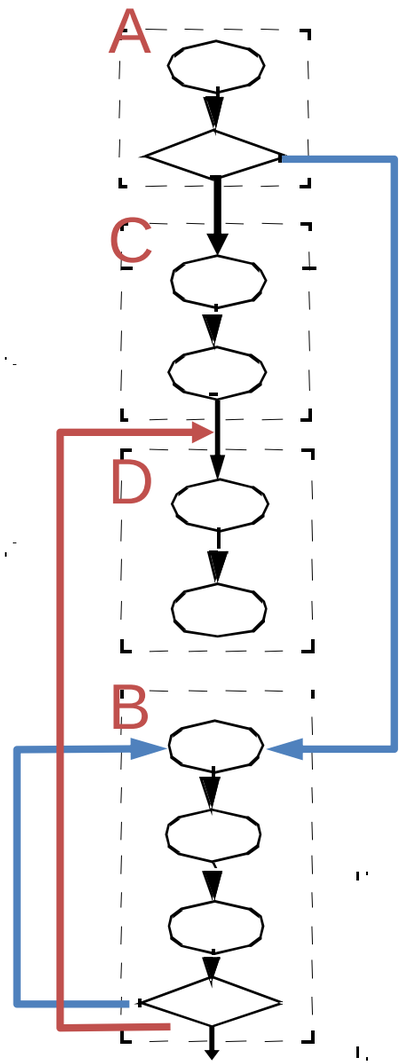
- Your program is actually a control flow graph
  - Shows possible paths of control flow through basic blocks
- Control Dependence
  - Node *X* is control dependent on Node *Y* if the computation in *Y* determines whether *X* executes

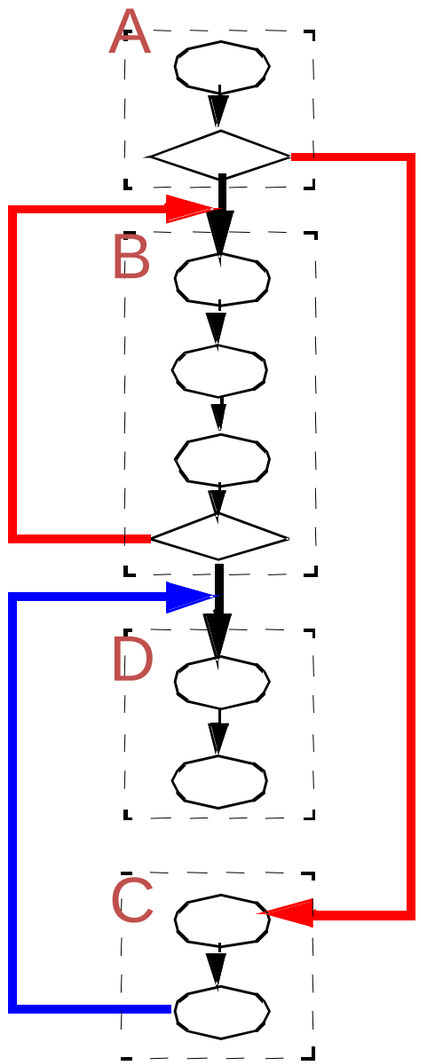



```
main:
        addi  r2, r0, A          |
        addi  r3, r0, B          |
        addi  r4, r0, C          |   BB 1
        addi  r5, r0, N          |
        add   r10,r0, r0         |
        bge   r10,r5, end        |
loop:                            |
        lw    r20, 0(r2)         |
        lw    r21, 0(r3)         |   BB 2
        bge   r20,r21,T1         |
        sw    r21, 0(r4)         |   BB 3
        b     T2                 |
T1:                              |
        sw    r20, 0(r4)         |   BB 4
T2:                              |
        addi  r10,r10,1          |
        addi  r2, r2, 4          |
        addi  r3, r3, 4          |   BB 5
        addi  r4, r4, 4          |
        blt   r10,r5, loop       |
end:
```

Mapping CFG to Linear Instruction Sequence







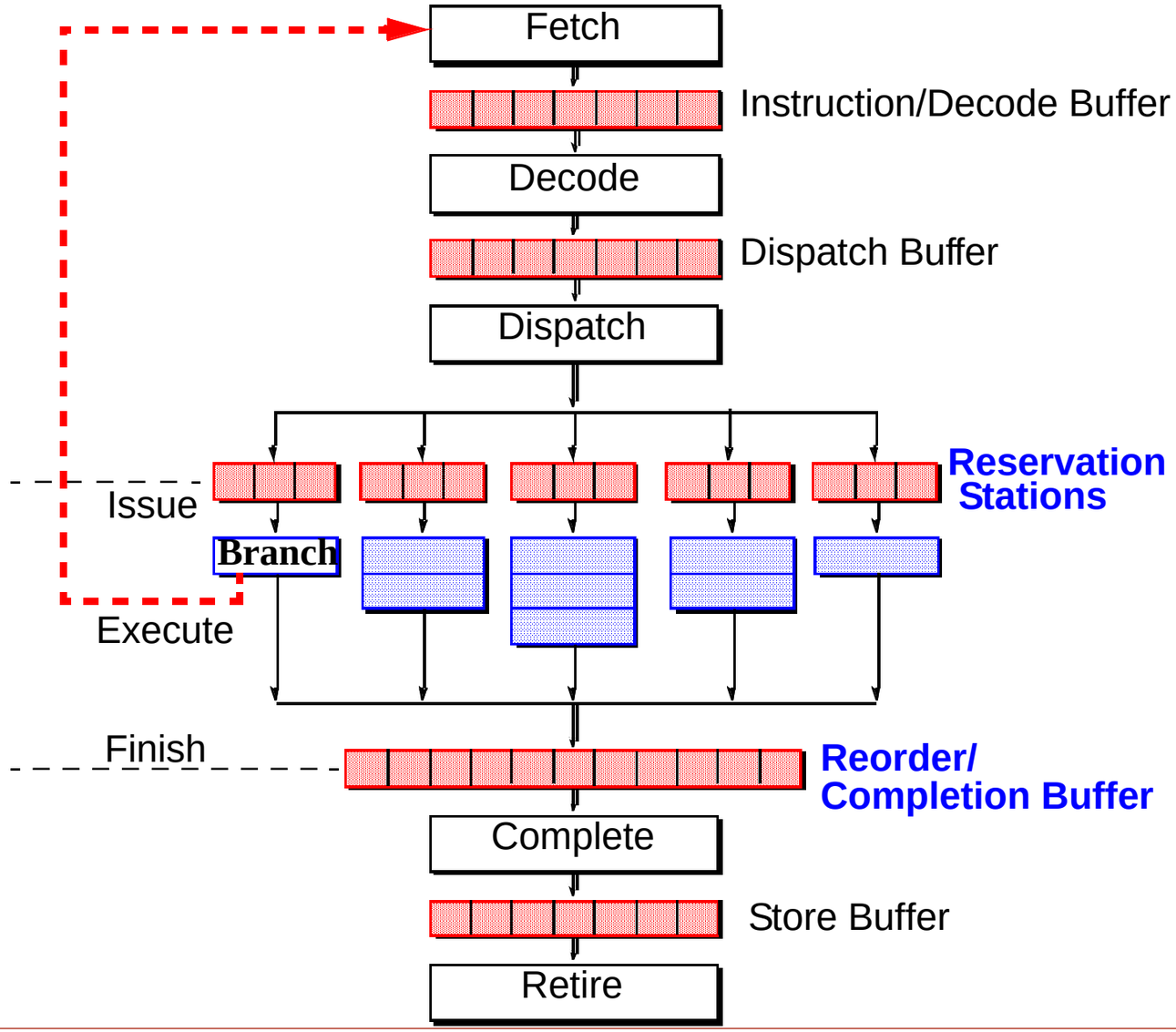
Branch Types and Implementation

- Types of Branches
  - Conditional or Unconditional?
  - Subroutine Call (aka Link), needs to save PC?
  - How is the branch target computed?
    - Static Target e.g. immediate, PC-relative
    - Dynamic targets e.g. register indirect
- Conditional Branch Architectures
  - Condition Code "*N-Z-C-V*" *e.g. PowerPC*
  - General Purpose Register *e.g. Alpha, MIPS*
  - Special Purposes register *e.g. Power's Loop Count*

What's So Bad About Branches?

- Robs instruction fetch bandwidth and ILP
  - Use up execution resources
  - Fragmentation of I-cache lines
  - Disruption of sequential control flow
    - Need to determine branch direction (conditional branches)
    - Need to determine branch target
- Example:
  - We have a N-way superscalar processor (N is large)
  - A branch every 5 instructions that takes 3 cycles to resolve
  - What is the effective fetch bandwidth?

Disruption of Sequential Control Flow



Riseman and Foster's Study

- 7 benchmark programs on CDC-3600
- Assume infinite machine:
  - Infinite memory and instruction stack, register file, fxn units

    Consider only true dependency at data-flow limit
- If bounded to single basic block, i.e. no bypassing of branches $\Rightarrow$ maximum speedup is **1.72**
- Suppose one can bypass conditional branches and jumps (i.e. assume the actual branch path is always known such that branches do not impede instruction execution)

Br. Bypassed:	*0*	*1*	*2*	*8*	*32*	*128*
Max Speedup:	***1.72***	***2.72***	***3.62***	***7.21***	***24.4***	***51.2***

Introduction to Branch Prediction

- Why do we need branch prediction?
- What do we need to predict about branches?
- Why are branches predictable?
- What mechanisms do we need for branch prediction?

Static Branch Prediction

- Option #1: based on type or use of instruction
  - E.g., assume backwards branches are taken (predicting a loop)
  - Can be used as a backup even if dynamic schemes are used
- Option #2: compiler or profile branch prediction
  - Collect information from instrumented run(s)
  - Recompile program with branch annotations (hints) for prediction
    - See heuristics list in next slide
  - Can achieve 75% to 80% prediction accuracy
- Why would dynamic branch prediction do better?

Heuristics for Static Prediction (Ball & Larus, PPoPP1993)

Heuristic	Description
Loop Branch	If the branch target is back to the head of a loop, predict taken.
Pointer	If a branch compares a pointer with NULL, or if two pointers are compared, predict in the direction that corresponds to the pointer being not NULL, or the two pointers not being equal.
Opcode	If a branch is testing that an integer is less than zero, less than or equal to zero, or equal to a constant, predict in the direction that corresponds to the test evaluating to false.
Guard	If the operand of the branch instruction is a register that gets used before being redefined in the successor block, predict that the branch goes to the successor block.
Loop Exit	If a branch occurs inside a loop, and neither of the targets is the loop head, then predict that the branch does not go to the successor that is the loop exit.
Loop Header	Predict that the successor block of a branch that is a loop header or a loop pre-header is taken.
Call	If a successor block contains a subroutine call, predict that the branch goes to that successor block.
Store	If a successor block contains a store instruction, predict that the branch does not go to that successor block.
Return	If a successor block contains a return from subroutine instruction, predict that the branch does not go to that successor block.

Dynamic Branch Prediction Tasks

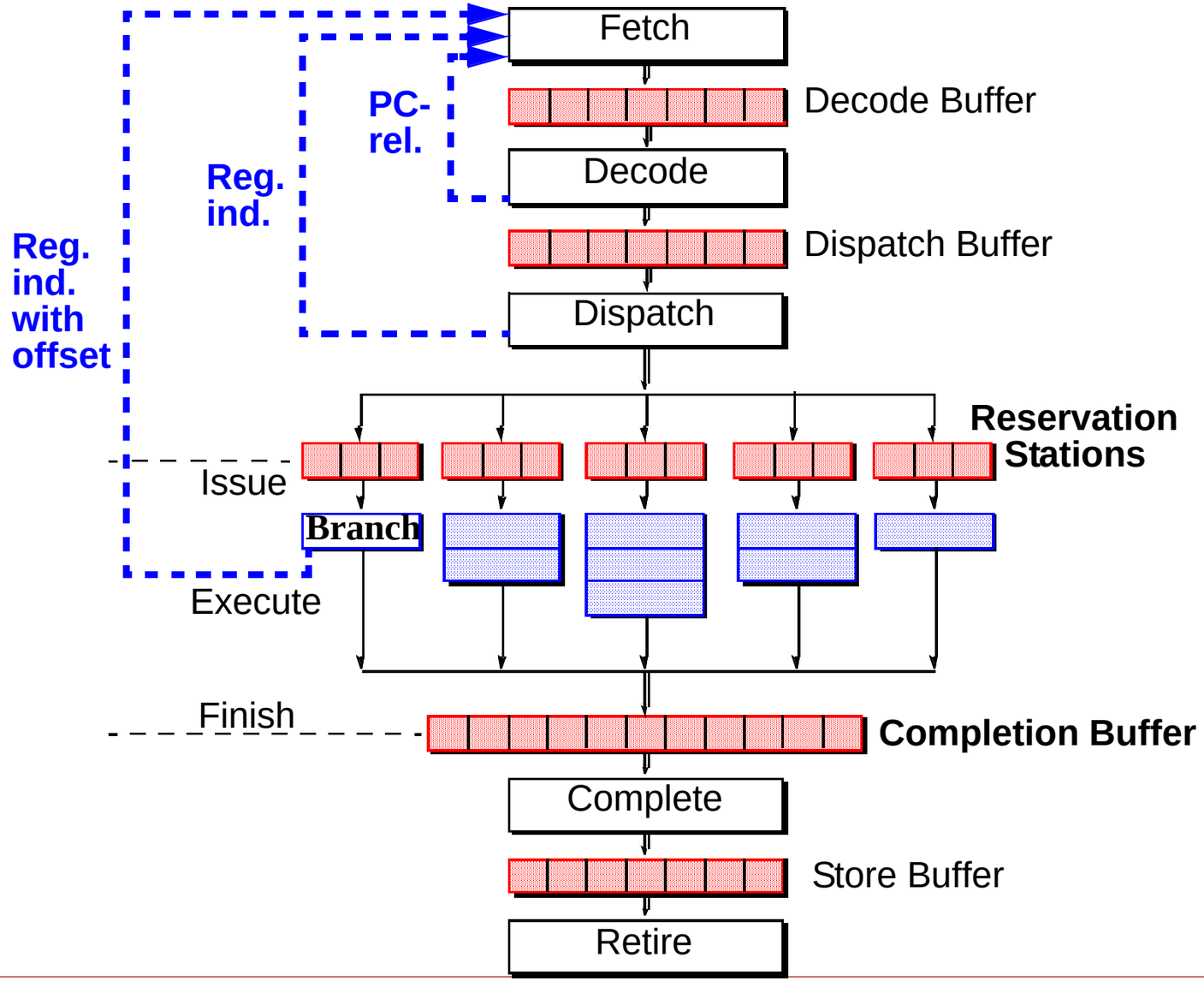
- Target Address Generation
  - Access register
    - PC, GP register, Link register
  - Perform calculation
    - +/- offset, auto incrementing/decrementing

  ⇒ Target Speculation

- Condition Resolution
  - Access register
    - Condition code register, data register, count register
  - Perform calculation
    - Comparison of data register(s)

  ⇒ Condition Speculation

Target Address Generation

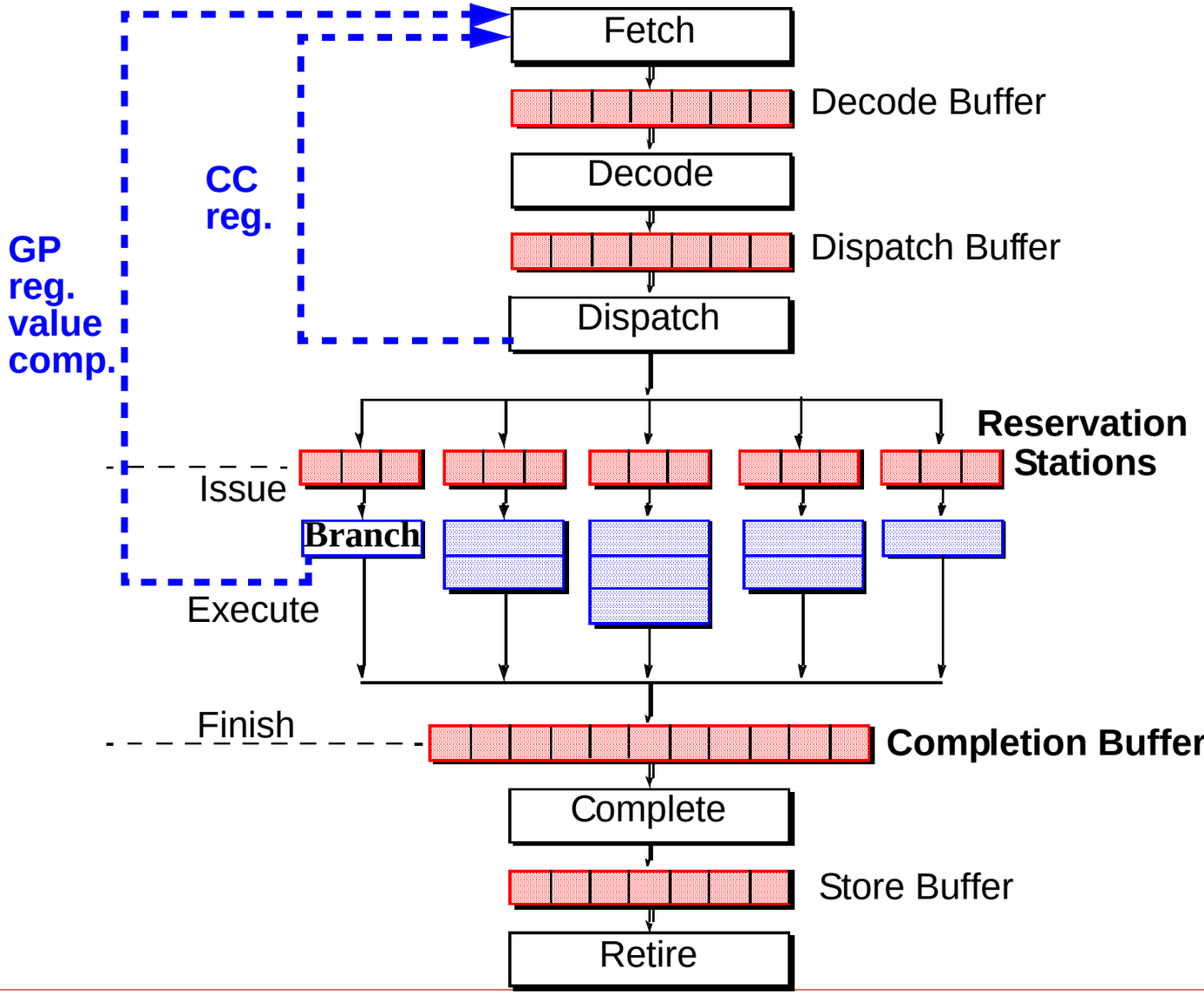


Determining Branch Target

Problem: Cannot fetch subsequent instructions until branch target is determined

- Minimize delay
  - Generate branch target early in the pipeline
- Make use of delay
  - Bias for not taken
  - Predict branch targes
  - For both PC-relative vs register Indirect targets

Condition Resolution

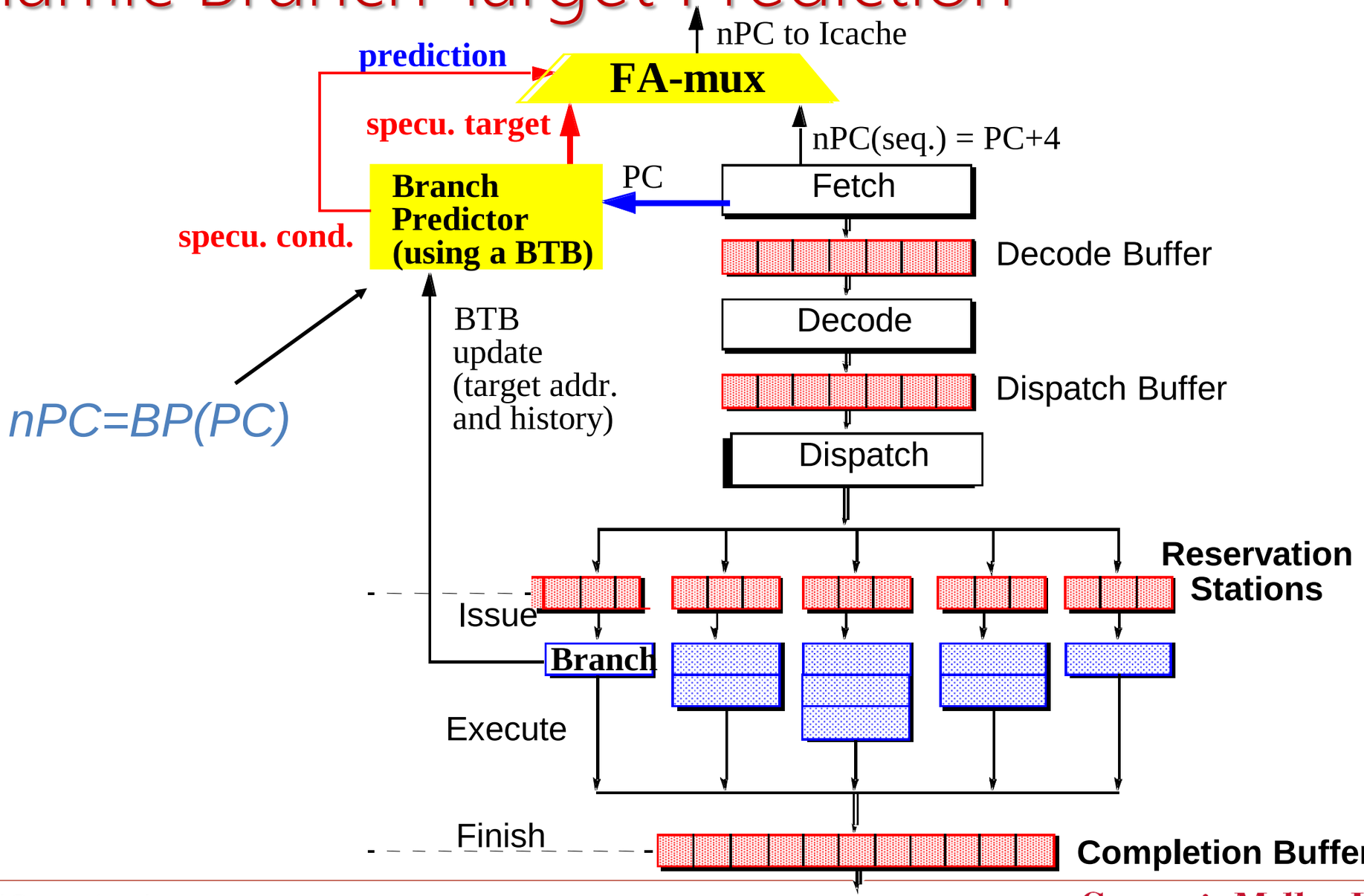


Determining Branch Direction

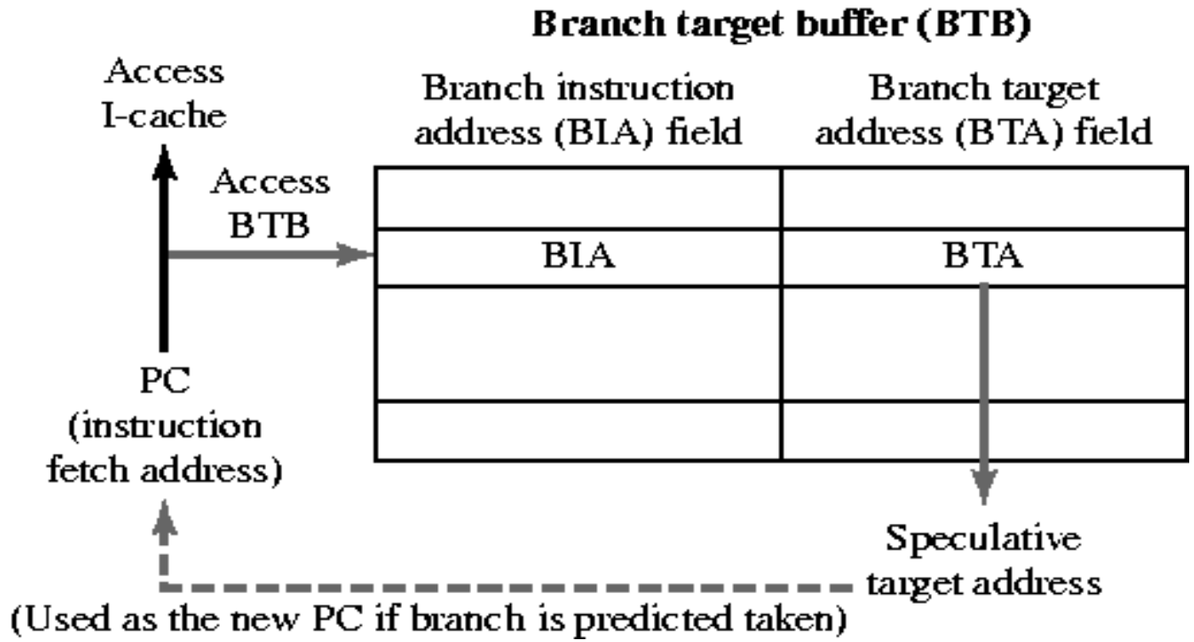
Problem: Cannot fetch subsequent instructions until branch direction is determined

- Minimize penalty
  - Move the instruction that computes the branch condition away from branch (ISA & compiler)
    - 3 branch components can be separated
    - Specify end of BB, specify condition, specify target
- Make use of penalty
  - Bias for not-taken
  - Fill delay slots with useful/safe instructions (ISA &compiler)
  - Follow both paths of execution (hardware)
  - *Predict branch direction (hardware)*

Dynamic Branch Target Prediction



Target Prediction: Branch Target Buffer (BTB)



- A small "cache-like" memory in the instruction fetch stage
- Remembers previously executed branches, their addresses (PC), information to aid target prediction, and most recent target addresses
- I-fetch stage compares current PC against those in BTB to "guess" nPC
  - If matched then prediction is made else nPC=PC+4
  - If predict taken then nPC=target address in BTB else nPC=PC+4
- When branch is actually resolved, BTB is updated

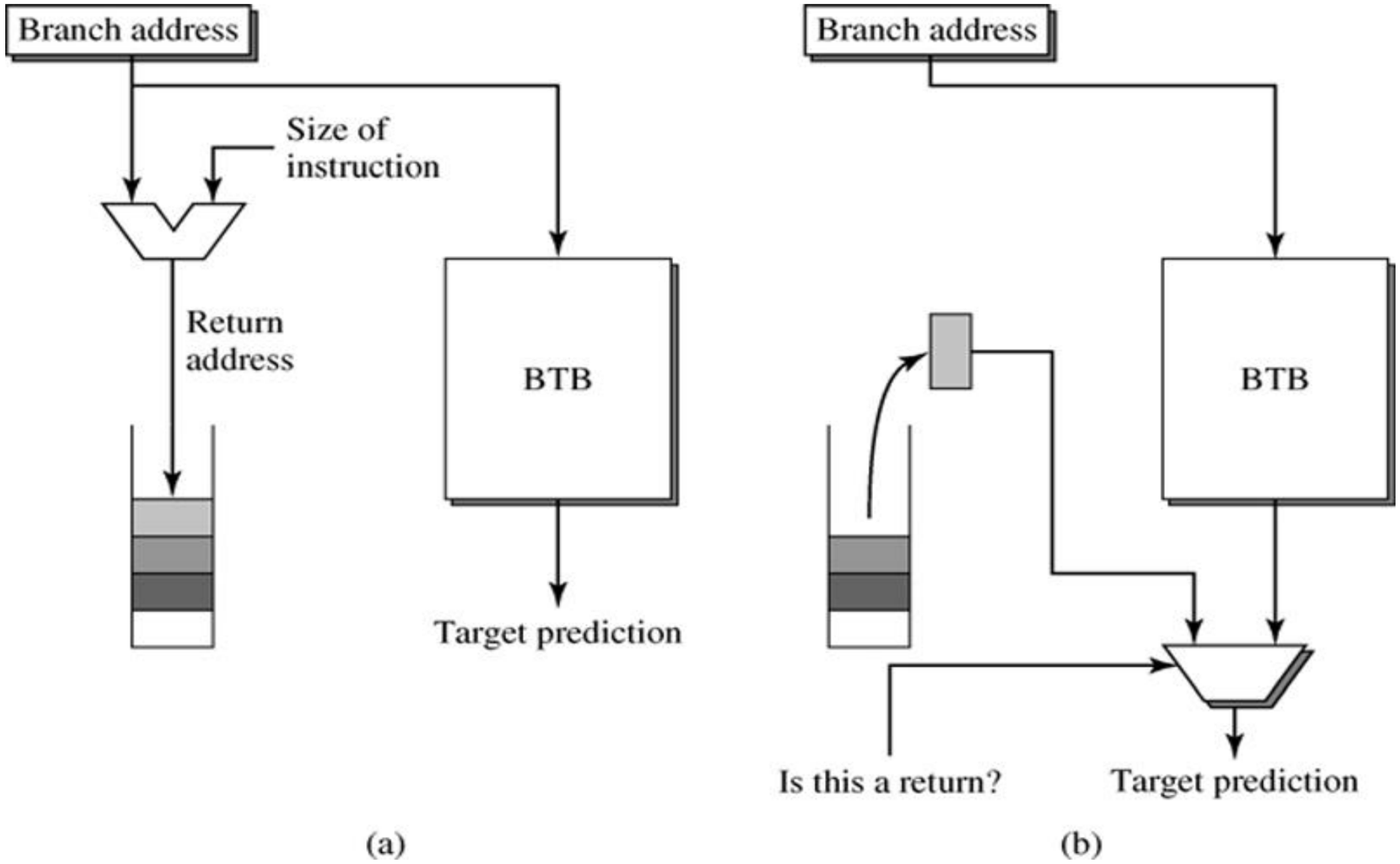
More on BTB (aka BTAC)

- Typically a large associative structure
  - Pentium3: 512 entries, 4-way; Opteron: 2K entries, 4-way
- Entry format
  - Valid bit, address tag (PC), target address, fall-through BB address (length of BB), branch type info, branch direction prediction
- BTB provides both target and direction prediction
- Multi-cycle BTB access?
  - The case in many modern processors (2 cycle BTB)
  - Start BTB access along with I-cache in cycle 0
  - In cycle 1, fetch from BTB+N (predict not-taken)
  - In cycle 2, use BTB output to verify
    - 1 cycle fetch bubble if branch was taken

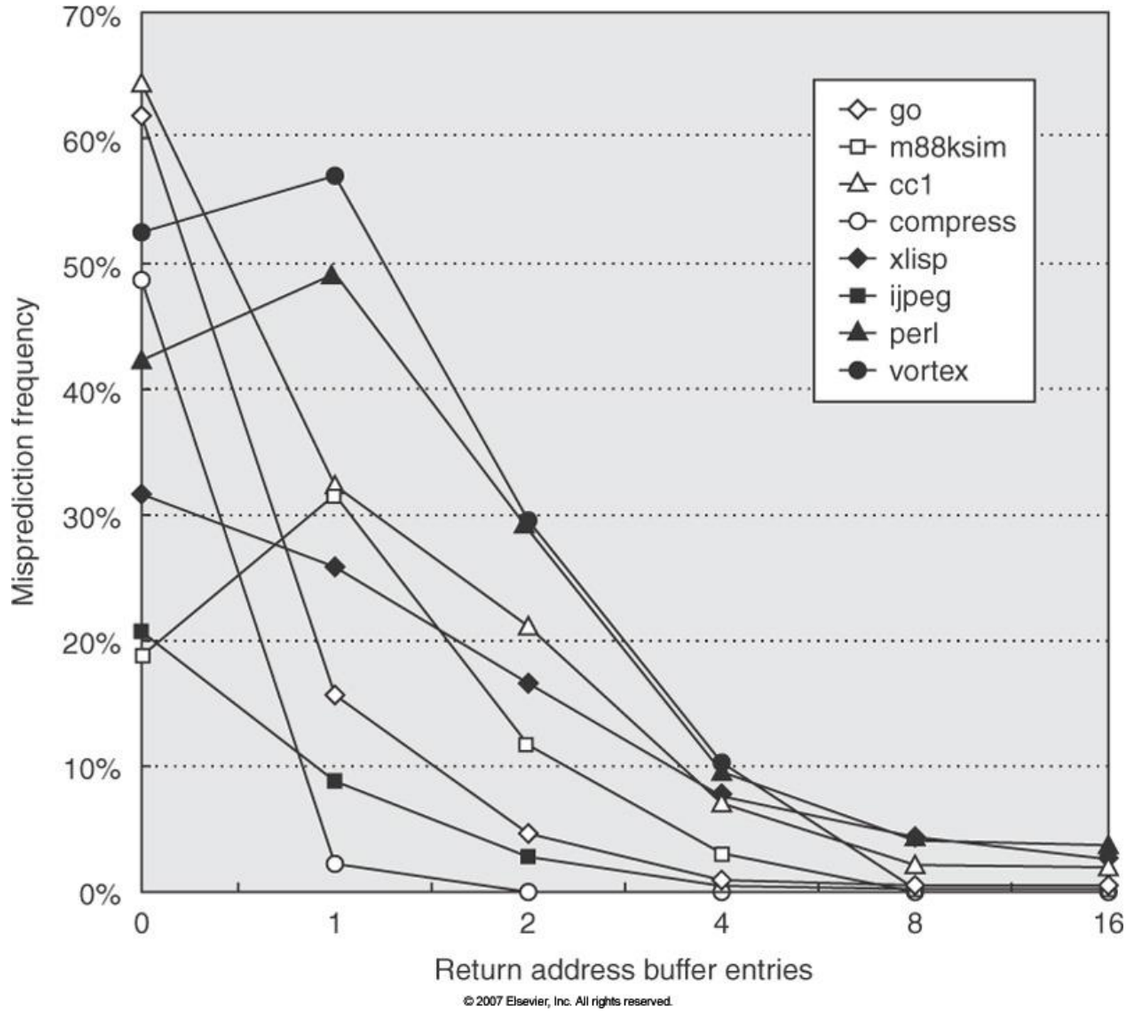
Branch Target Prediction for Function Returns

- In most languages, function calls are fully nested
  - If you call A() ⇒ B() ⇒ C() ⇒ D()
  - Your return targets are PCc ⇒ PCb ⇒ PCa ⇒ PCmain
- Return address stack (RAS)
  - A FILO structure for capturing function return addresses
  - Operation
    - On a function call retirement, push call PC into the stack
    - On a function return, use the top value in the stack & pop
  - A 16-entry RAS can predict returns almost perfectly
    - Most programs do not have such a deep call tree
  - Sources of RAS inaccuracies
    - Deep call statements (circular buffer overflow – will lose older calls)
    - Setjmp and longjmp C functions (irregular call semantics)

RAS Operation



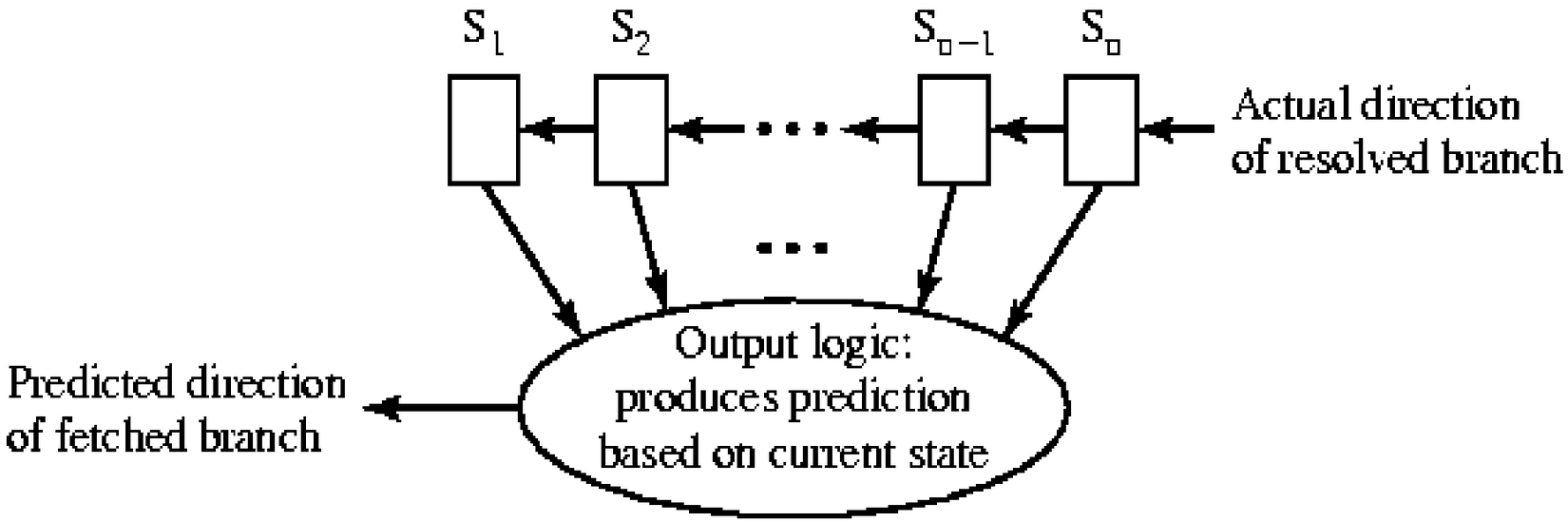
RAS Effectiveness & Size (SPEC CPU'95)



Branch Condition Prediction

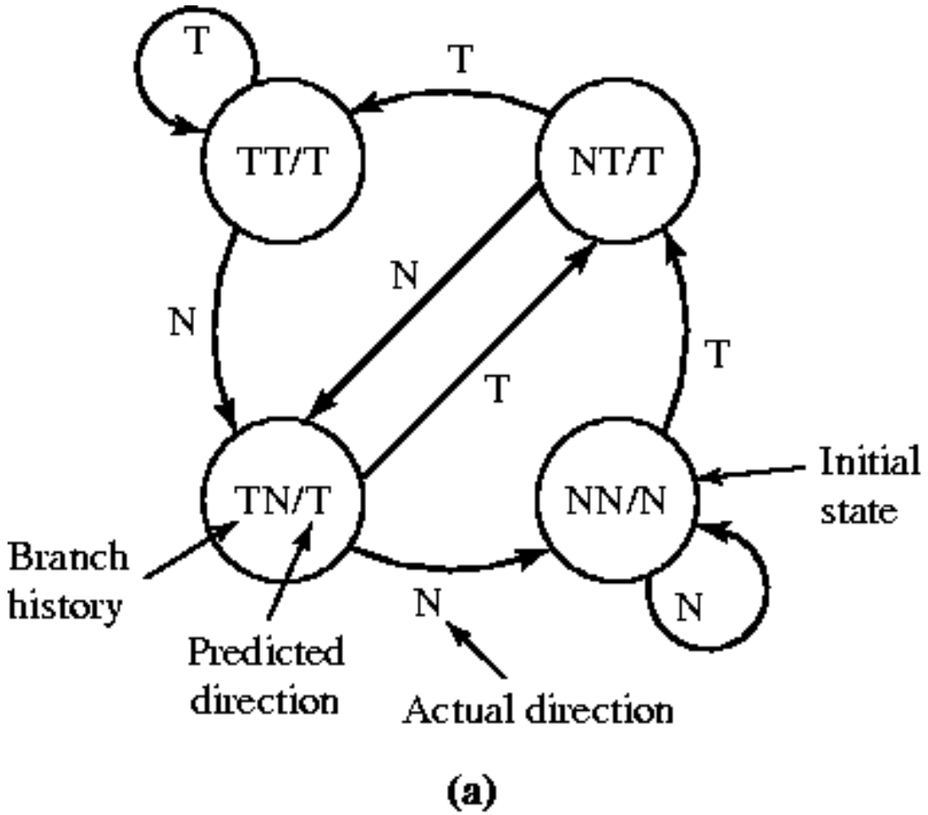
- Biased For Not Taken
  - Does not affect the instruction set architecture
  - Not effective in loops
- Software Prediction
  - Encode an extra bit in the branch instruction
    - Predict not taken: set bit to 0
    - Predict taken: set bit to 1
  - Bit set by compiler or user; can use profiling
  - Static prediction, same behavior every time
- Prediction Based on Branch Offsets
  - Positive offset: predict not taken
  - Negative offset: predict taken
- Prediction Based on History

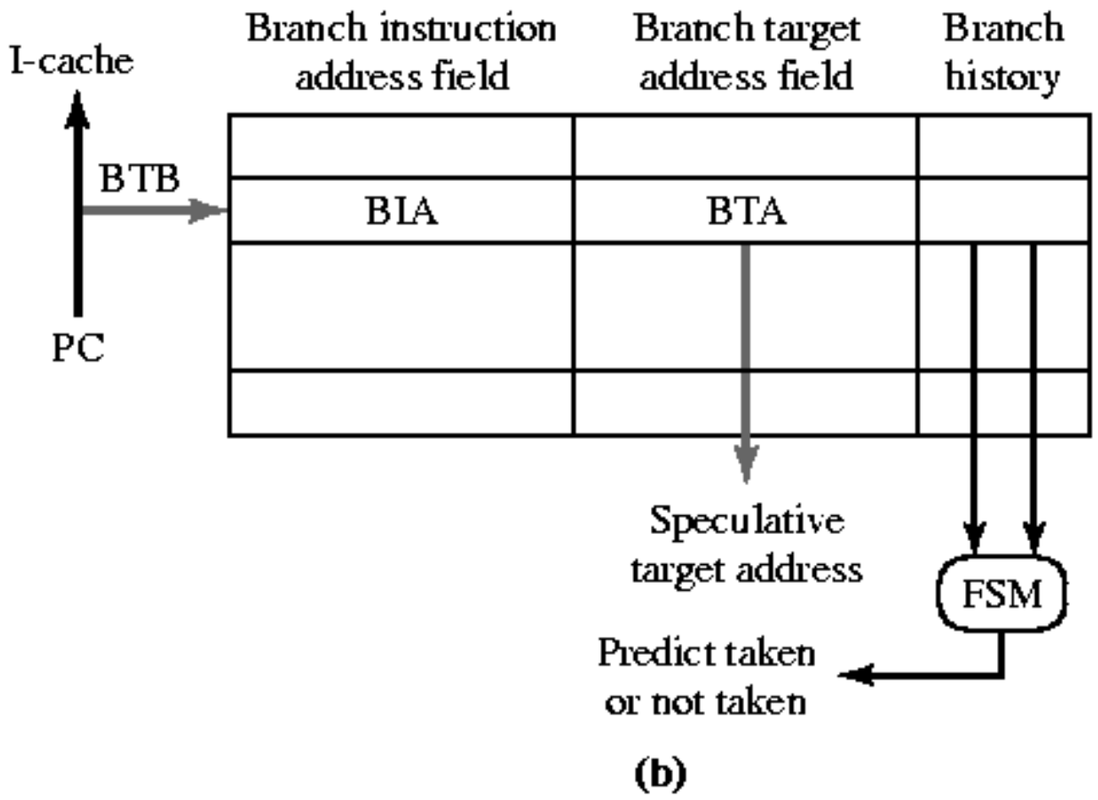
History-Based Branch Direction Prediction



- Track history of previous directions of branches (T or NT)
- History can be local (per static branch) or global (all branches)
- Based on observed history bits (T or NT), a FSM makes a prediction of Taken or Not Taken
- Assumes that future branching behavior is predictable based on historical branching behavior

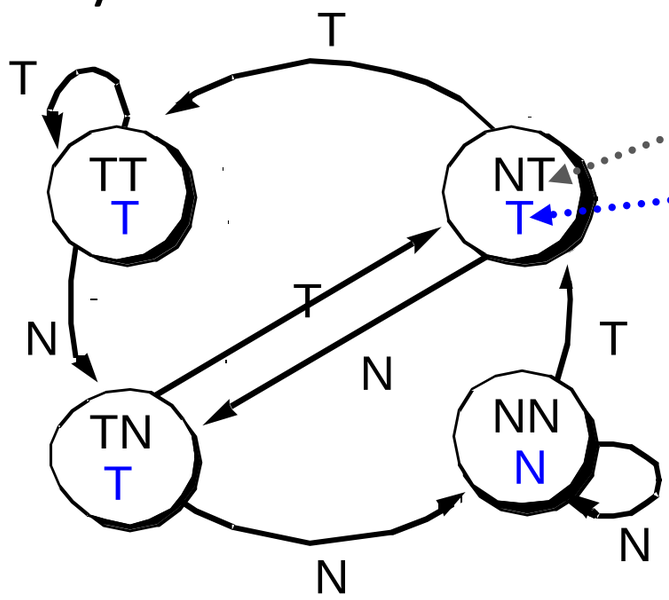
History-Based Branch Prediction





Example Prediction Algorithm

- Prediction accuracy approaches maximum with as few as 2 preceding branch occurrences used as history

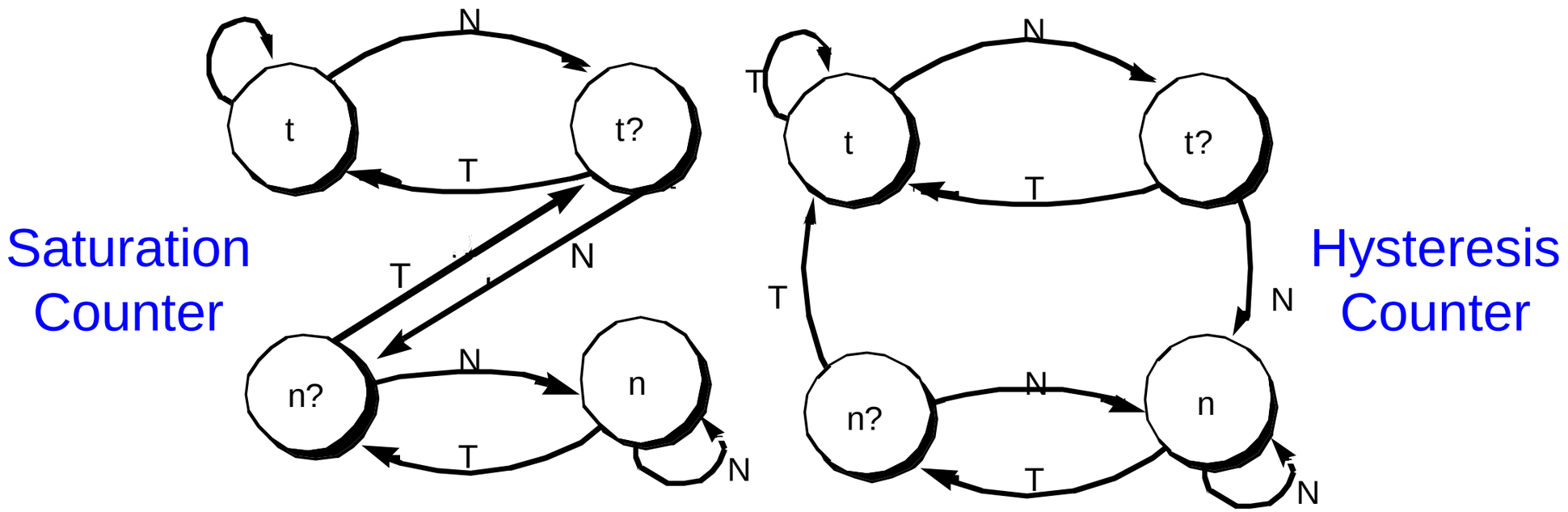


Results (%)

IBM1	IBM2	IBM3	IBM4	DEC	CDC
93.3	96.5	90.8	83.4	97.5	90.6

- History avoids mispredictions due to one time events
  - Canonical example: loop exit
- 2-bit FSM as good as n-bit FSM
- Saturating counter as good as any FSM

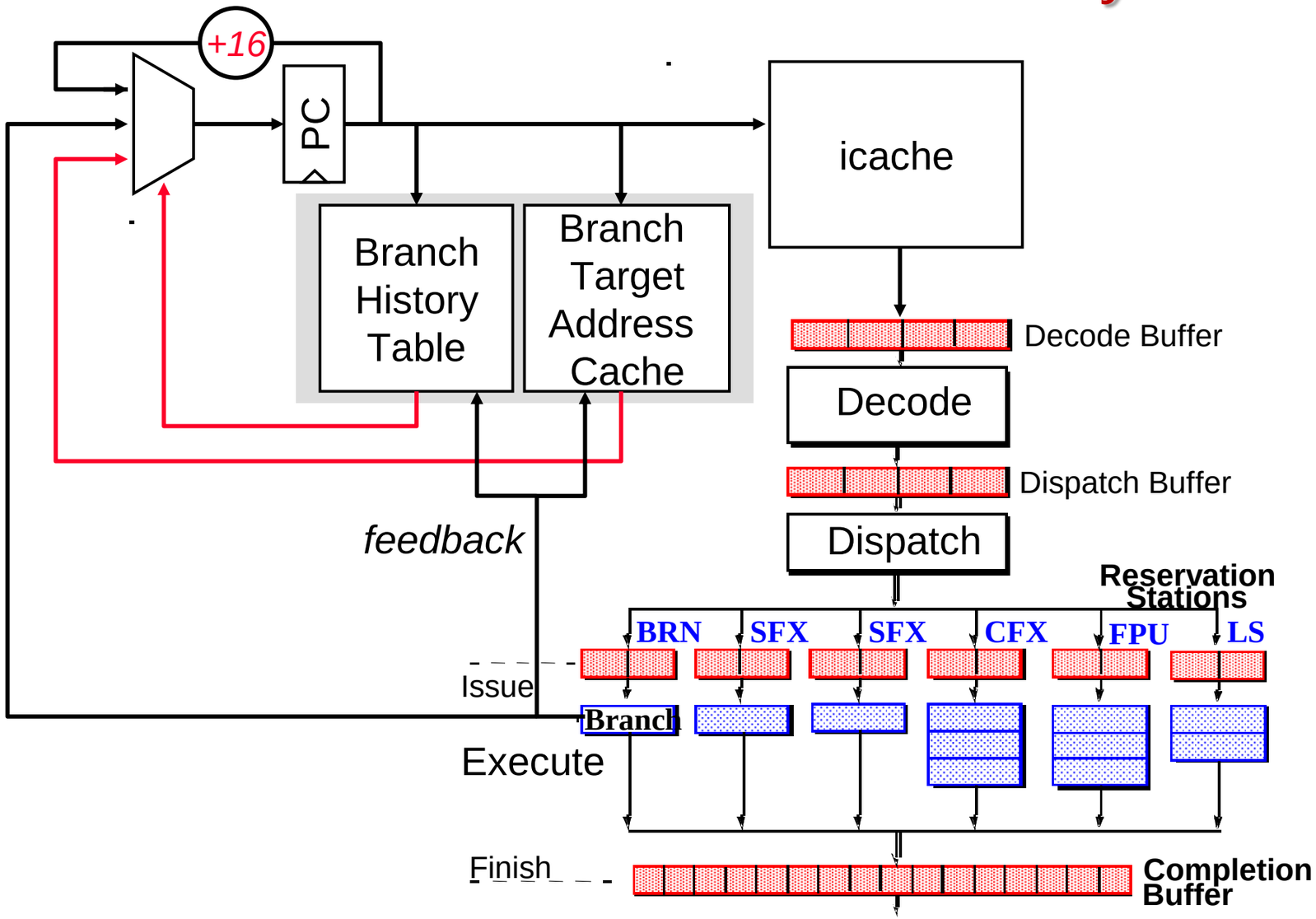
Other Prediction Algorithms



- Combining prediction accuracy with BTB hit rate (86.5% for 128 sets of 4 entries each), branch prediction can provide the net prediction accuracy of approximately 80%. This implies a 5-20% performance enhancement.

Dynamic Branch Prediction Based on History

- Use HW tables to track history of direction/targets
  - nextPC = function(PC, history)
- Need to verify prediction
  - Branch still gets to execute



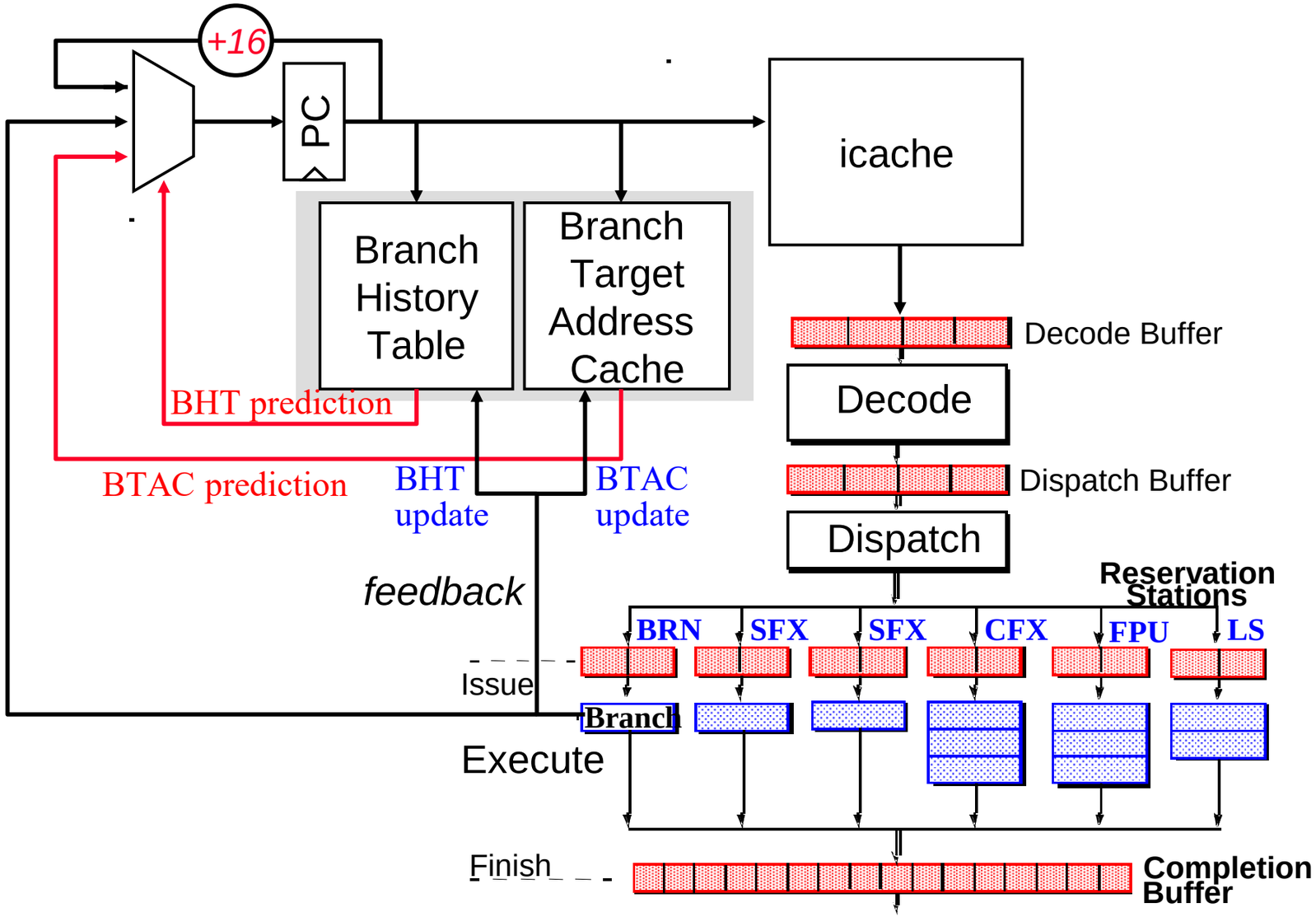
PowerPC 604 Branch Predictor: BHT & BTAC

BTAC:

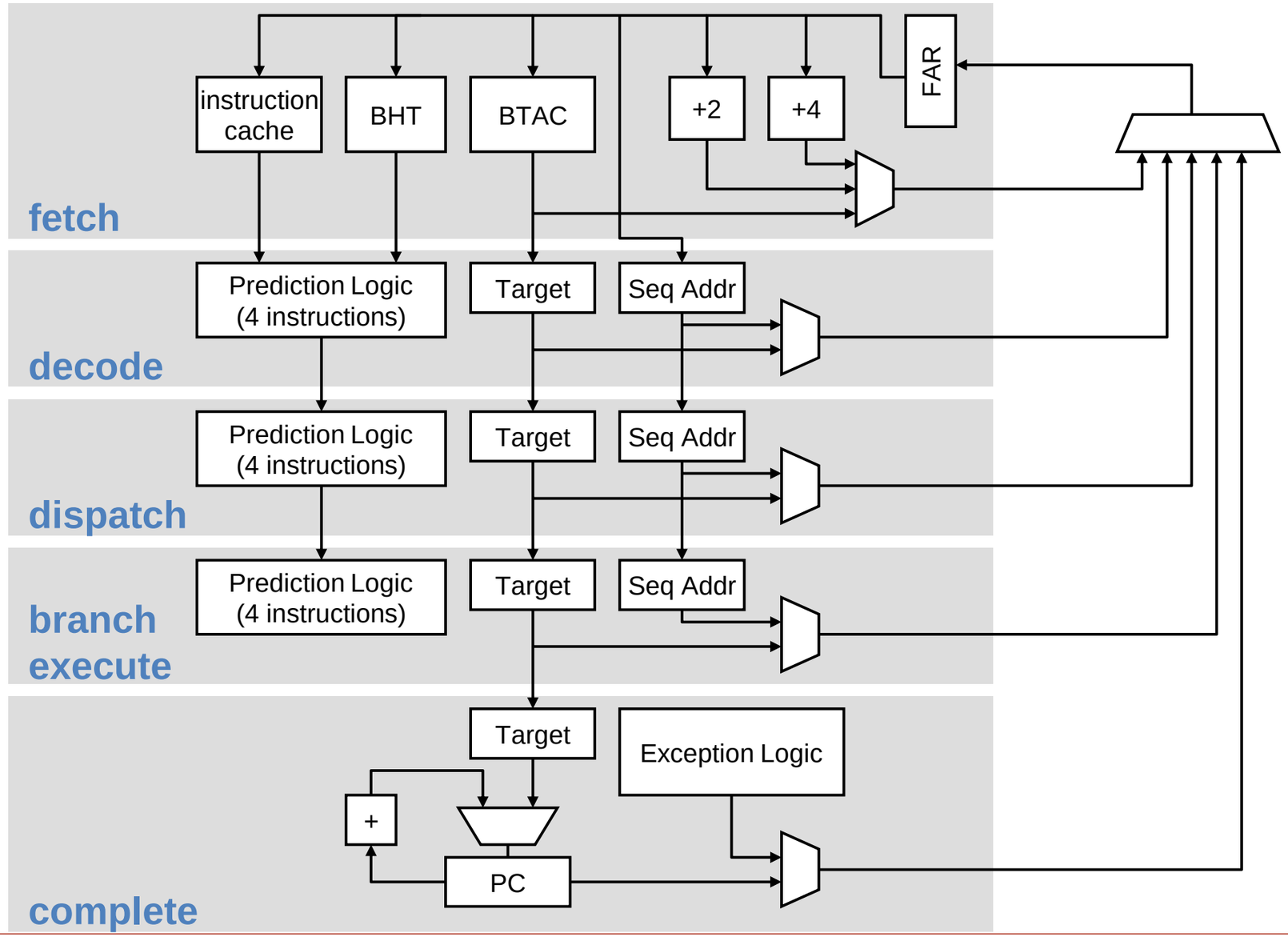
- 64 entries
- Fully associative
- Hit → predict taken

BHT:

- 512 entries
- Direct mapped
- 2-bit saturating counter
- History based prediction
- Overrides BTAC prediction



PowerPC 604 Fetch Address Generation



18-600 Foundations of Computer Systems

Lecture 11: "Modern Superscalar Out-of-Order Processors"

John P. Shen & Zhiyi Yu

October 5, 2016

Next Time ...

- Required Reading Assignment:
  - **Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**
- Recommended Reading Assignment:
  - **Chapter 5 of Shen and Lipasti (SnL).**

Electrical & Computer ENGINEERING