CS 252
Microarchitecture:
Superscalar Processor Design

John P. Shen

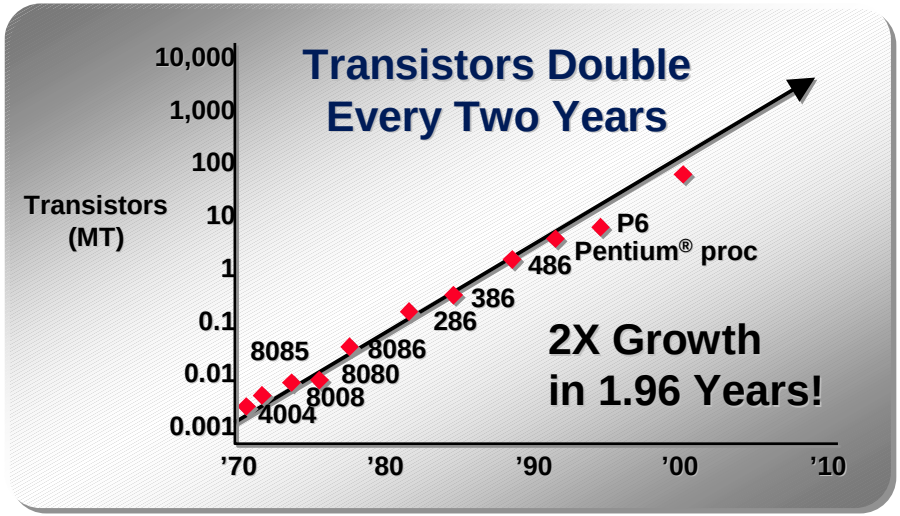
Microprocessor Research

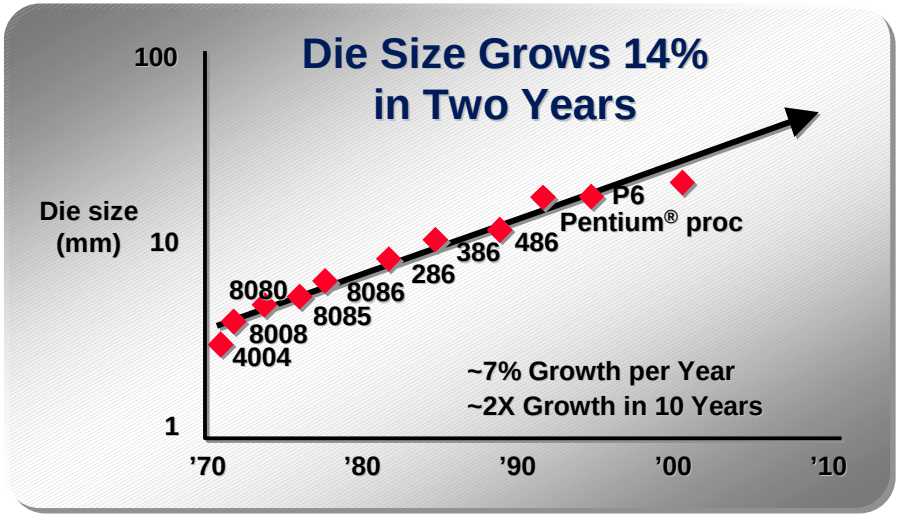
Intel Labs

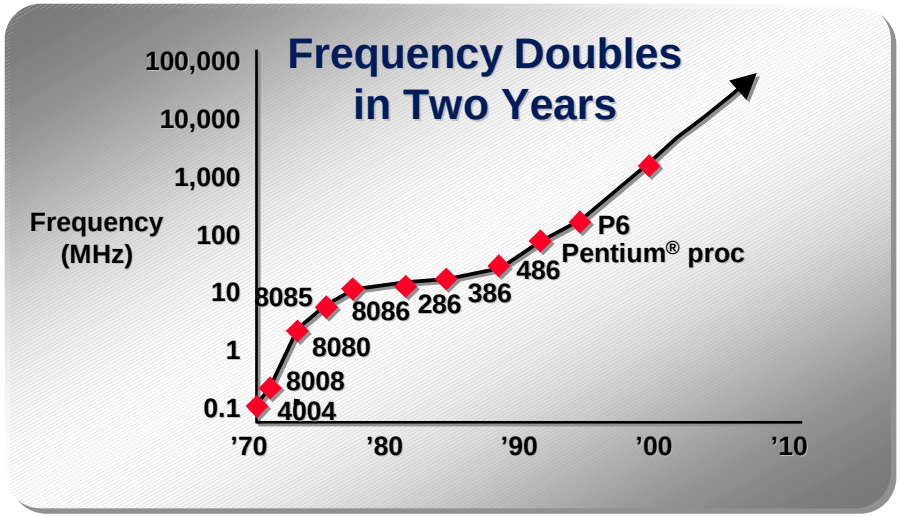
March 19, 2002

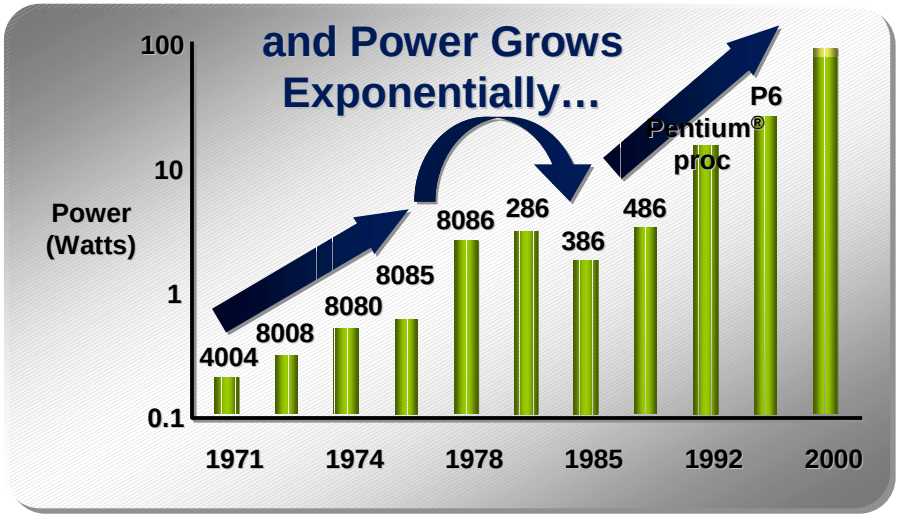
(john.shen@intel.com)

Moore's Law Continues…



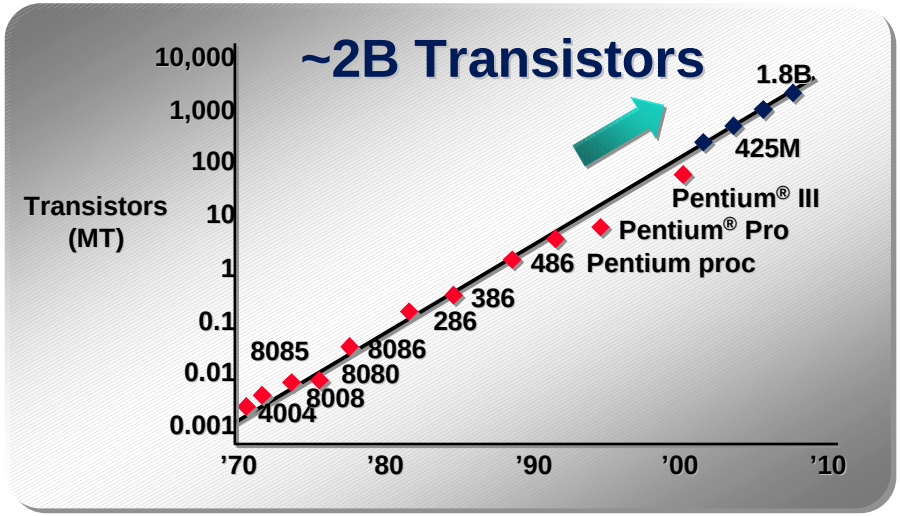


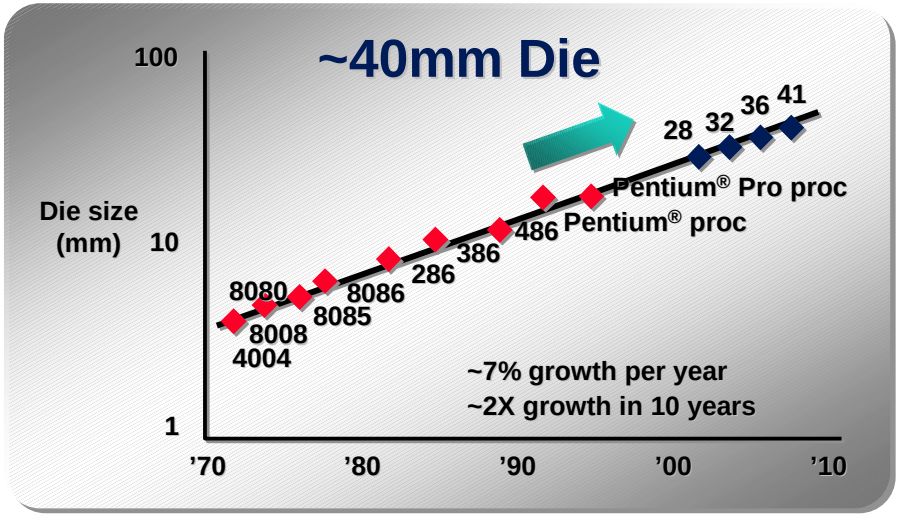


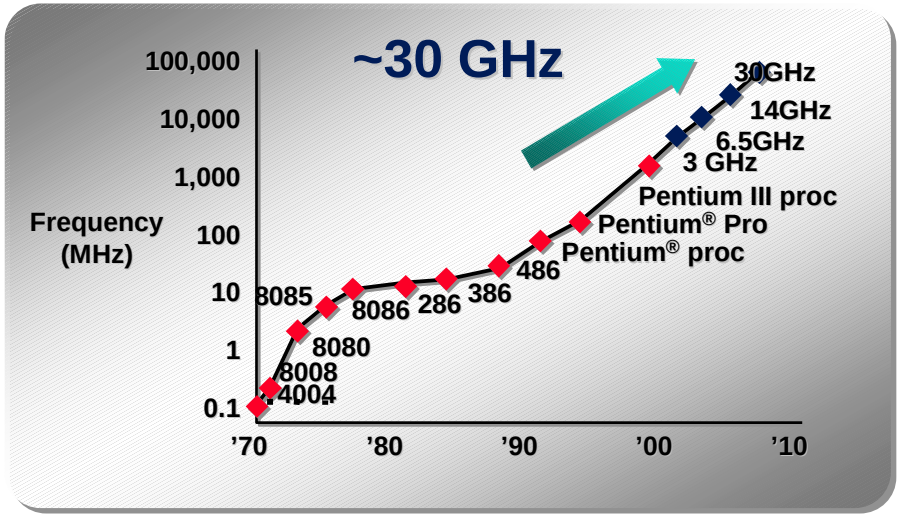


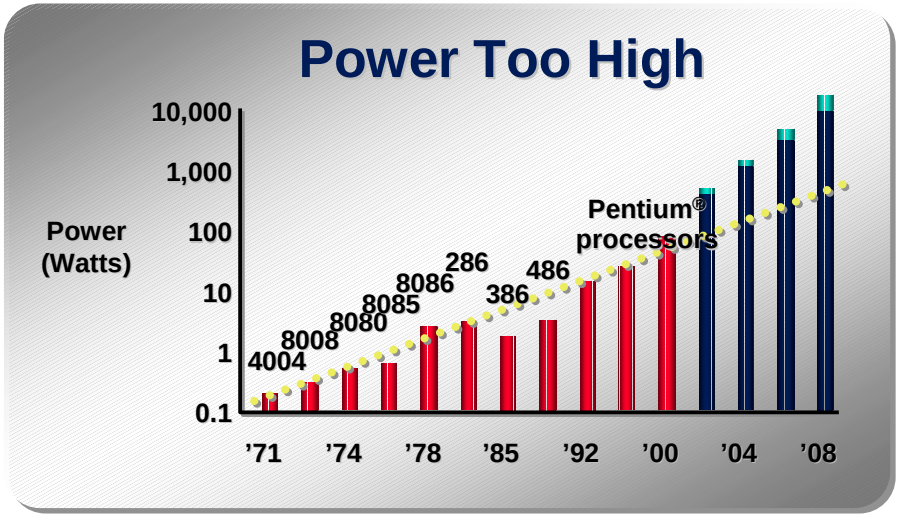
Source: Intel Corporation

… For At Least Another Decade









Source: Intel Corporation

1. Microprocessor Performance

Evolution of Microprocessors

	1970-1980	1980-1990	1990-2000	2000-2010
Transistor Count	2K-100K	100K-1M	1M-100M	100M-1B
Clock Frequency	0.1-3MHz	3-30MHz	30M-1GHz	1-15GHz
Instruction/Cycle	< 0.1	0.1-0.9	0.9- 1.9	1.9-2.9 (?)

Performance Growth in Perspective

- Doubling every 18 months (1982-2000):
  - total of 3,200X
  - Cars travel at 176,000 MPH; get 64,000 miles/gal.
  - Air travel: L.A. to N.Y. in 5.5 seconds (MACH 3200)
  - Wheat yield: 320,000 bushels per acre
- Doubling every 24 months (1971-2001):
  - total of 36,000X
  - Cars travel at 2,400,000 MPH; get 600,000 miles/gal.
  - Air travel: L.A. to N.Y. in 0.5 seconds (MACH 36,000)
  - Wheat yield: 3,600,000 bushels per acre

Unmatched by any other industry!!

[John Crawford, Intel, 1993]

"Iron Law" of Processor Performance

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

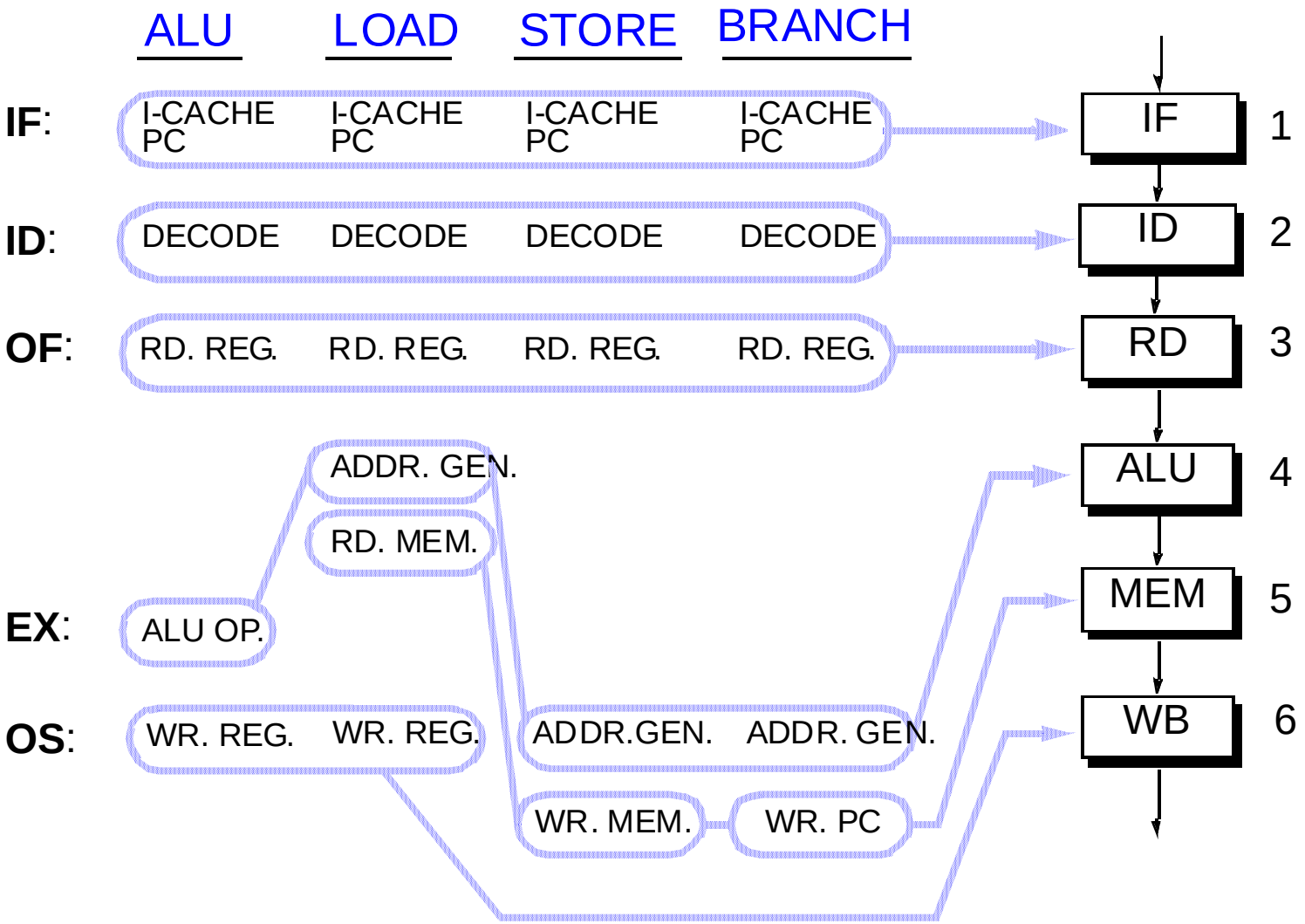
$$= \boxed{\frac{\text{Instructions}}{\text{Program}}} \times \boxed{\frac{\text{Cycles}}{\text{Instruction}}} \times \boxed{\frac{\text{Time}}{\text{Cycle}}}$$

(instr. count) *(CPI)* *(cycle time)*

2. Review of Scalar Pipelined Processors

Scalar Pipelined Processors

The 6-stage TYPICAL pipeline:



Inter-instruction Dependences

- *Data dependence*

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

- *Anti-dependence*

$r_3 \leftarrow r_1$ op r_2
$r_1 \leftarrow r_4$ op r_5

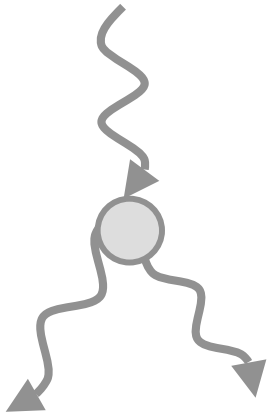
Write-after-Read (WAR)

- *Output dependence*

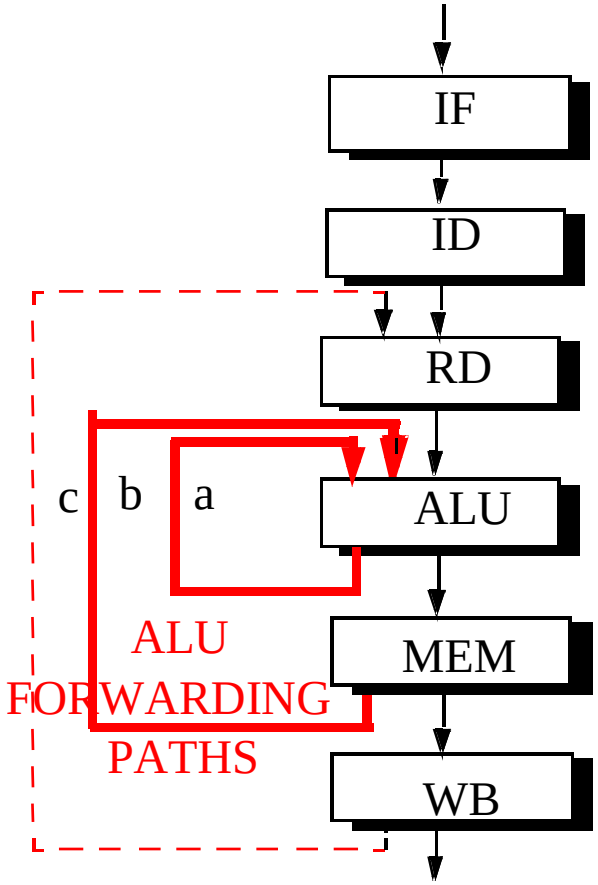
$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

- Control dependence



ALU Interlock and Penalty



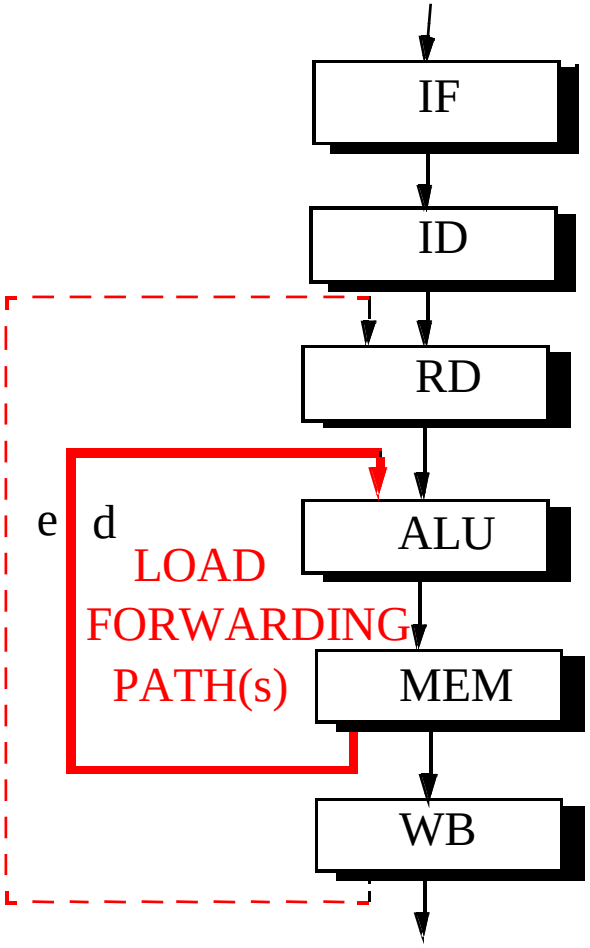
dist=1	dist=2	dist=3
i+1: _ ← R_X	i+2: _ ←R_X	i+3: _ ←R_X
i: R_X ← _	i+1: R_y ← _	i+2: R_z ← _
	i: R_X ← _	i+1: R_Y ← _
		i: R_X ← _

(i → i+1) Forwarding via Path a

(i → i+2) Forwarding via Path b

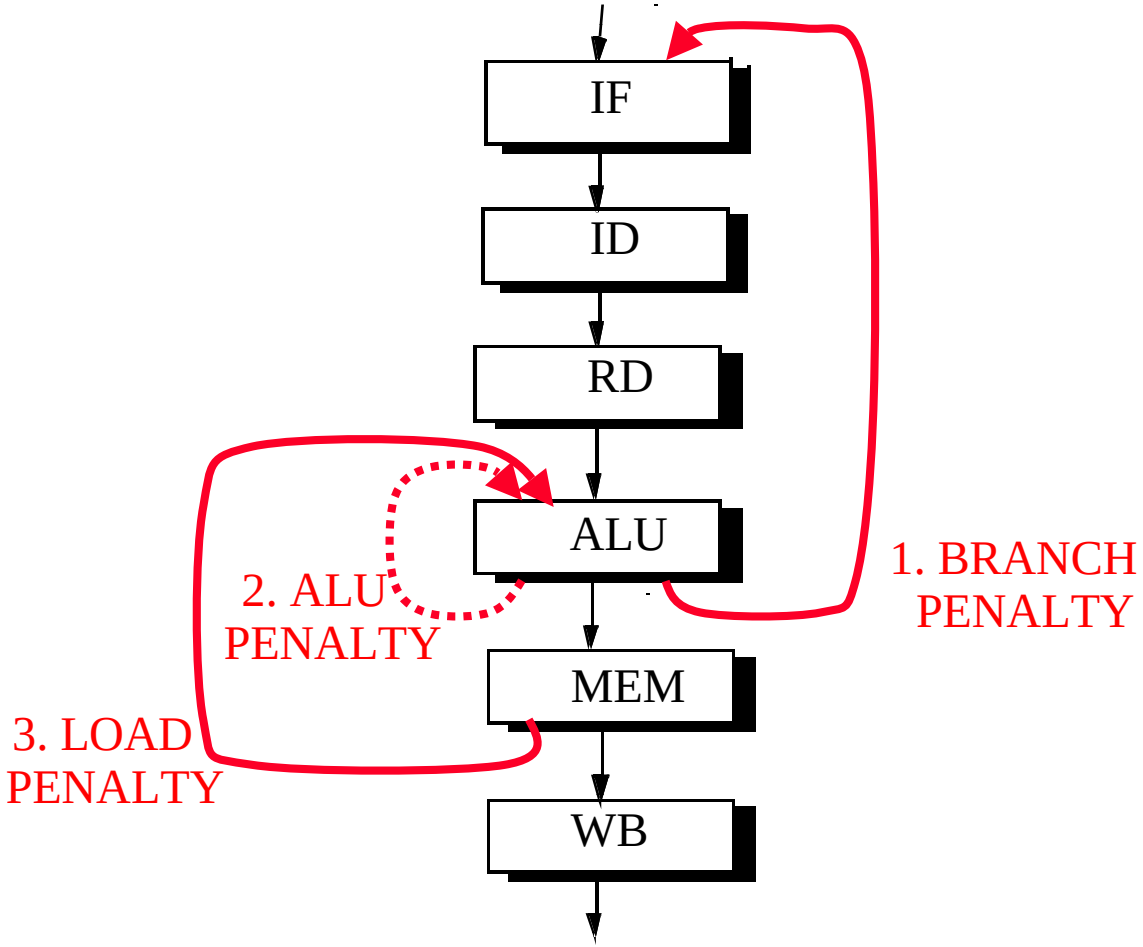
(i → i+3) i writes R1 before i+3 reads R1

Load Interlock and Penalty



dist=1	dist=2	dist=3
i+1: _ ← R_X	i+2: _ ←R_X	i+3: _ ←R_X
i: R_X ← mem[ ]	i+1: R_y ← _	i+2: R_z ← _
	i: R_X ← mem[ ]	i+1: R_Y ← _
		i: R_X ← mem[ ]
(i → i+1) Stall i+1	**(i → i+1) Forwarding via Path d**	**(i → i+2) i writes R1 before i+2 reads R1**

3 Major Penalty Loops of Pipelining



Performance Objective: Reduce CPI to 1.

3. Introduction to Modern Superscalar Processors

Limitations of Scalar Pipelines

- Upper Bound on Scalar Pipeline Throughtput

  Limited by IPC = 1

- Inefficient Unification Into Single Pipeline

  Long latency for each instruction

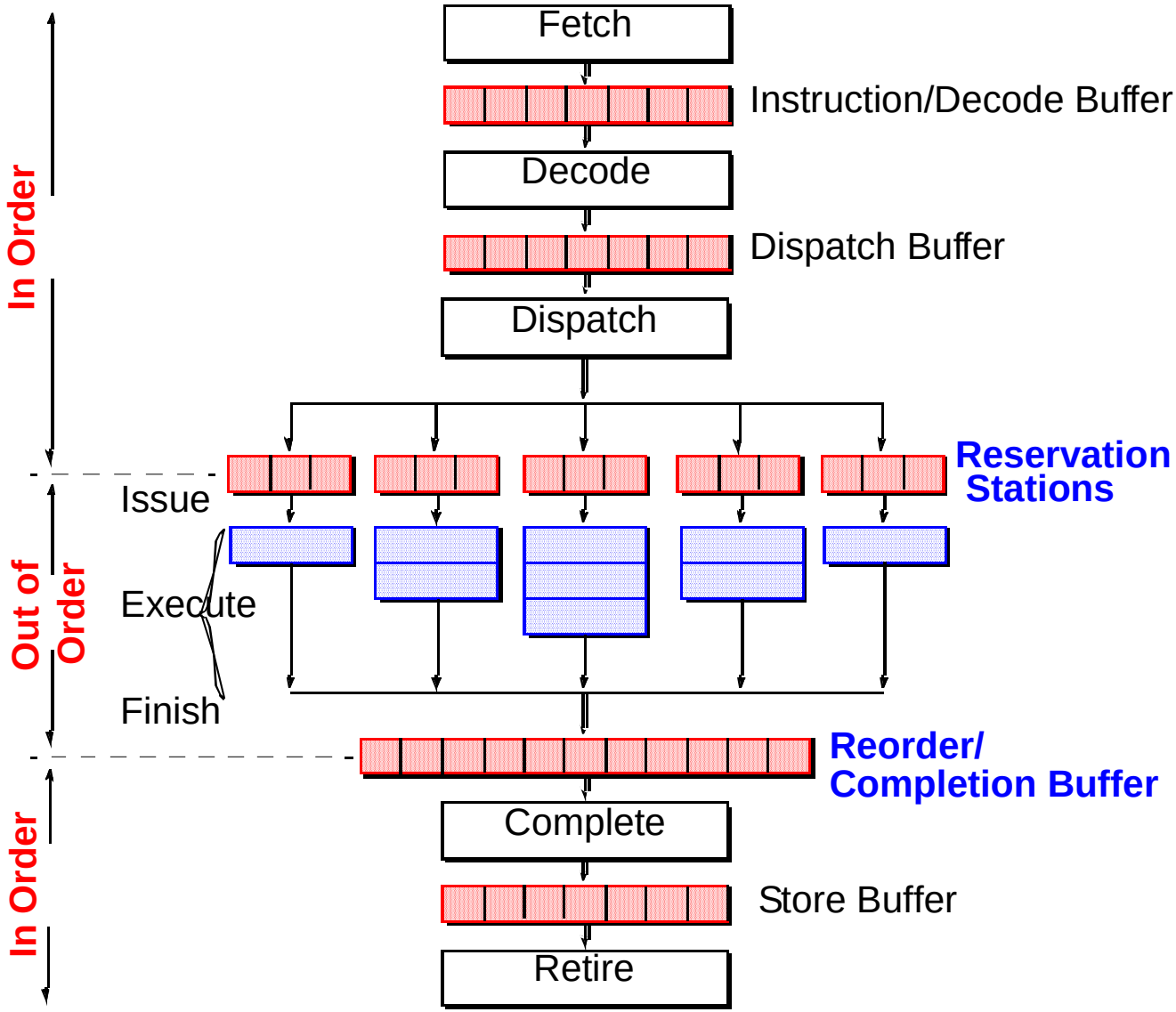
- Performance Lost Due to Rigid Pipeline

  Unnecessary stalls

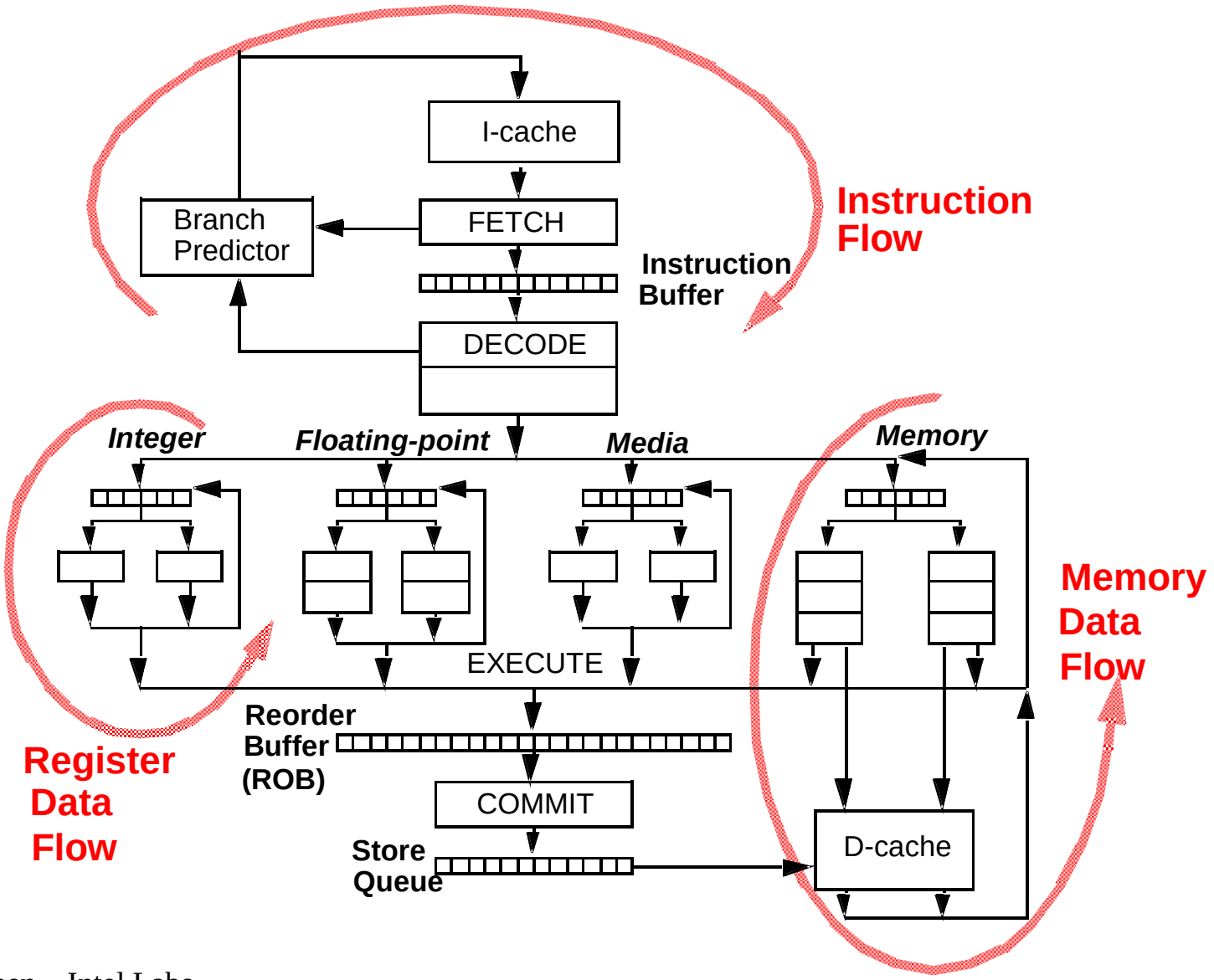
Scalar to Superscalar Pipelines

- Parallel Pipeline
  - Wide pipelines
  - Advance multiple instructions per cycle
- Diversified Pipeline
  - Multiple functional unit types
  - Mix of different functional units
- Dynamic Pipeline
  - Out of order execution
  - Distributed functional units

A Modern Superscalar Processor

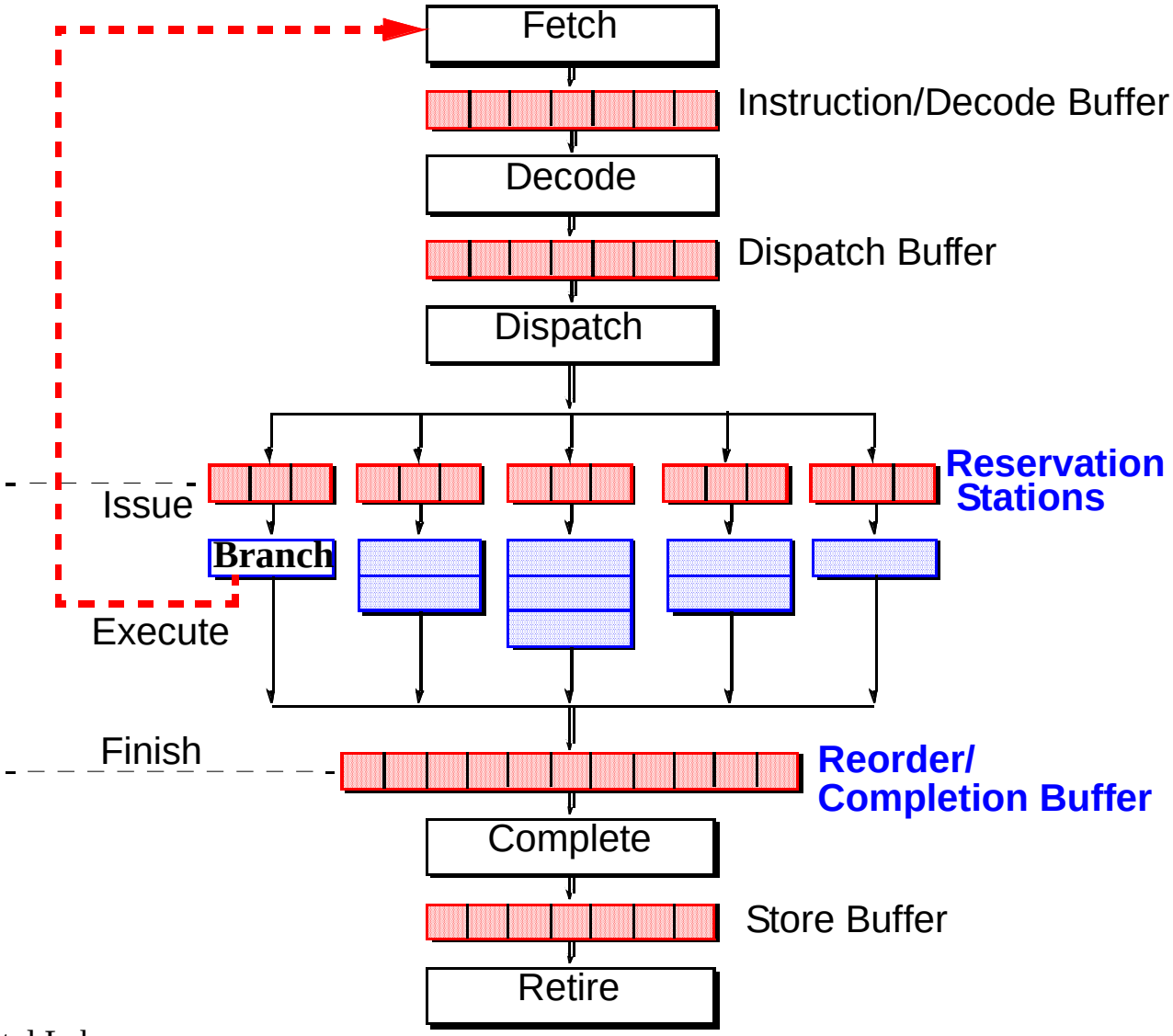


3 Flow Paths of Superscalars



4. Instruction Flow Techniques (Branch Penalty)

What's So Bad About Branches?



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*

Branch Prediction

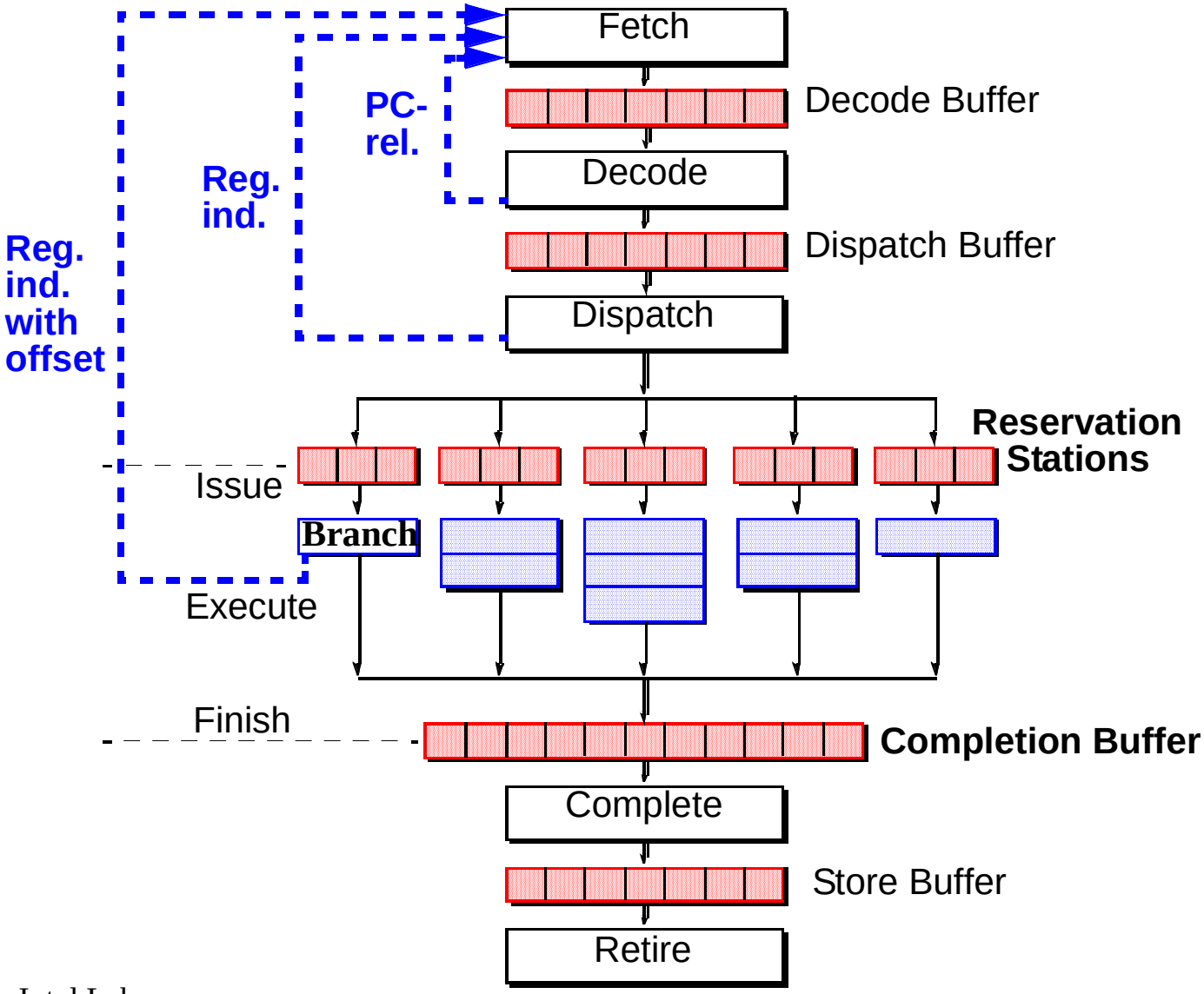
- Target Address Generation
  - Access register
    - PC, GP register, Link register
  - Perform calculation
    - +/- offset, auto incrementing/decrementing

  ⇒ Target Speculation

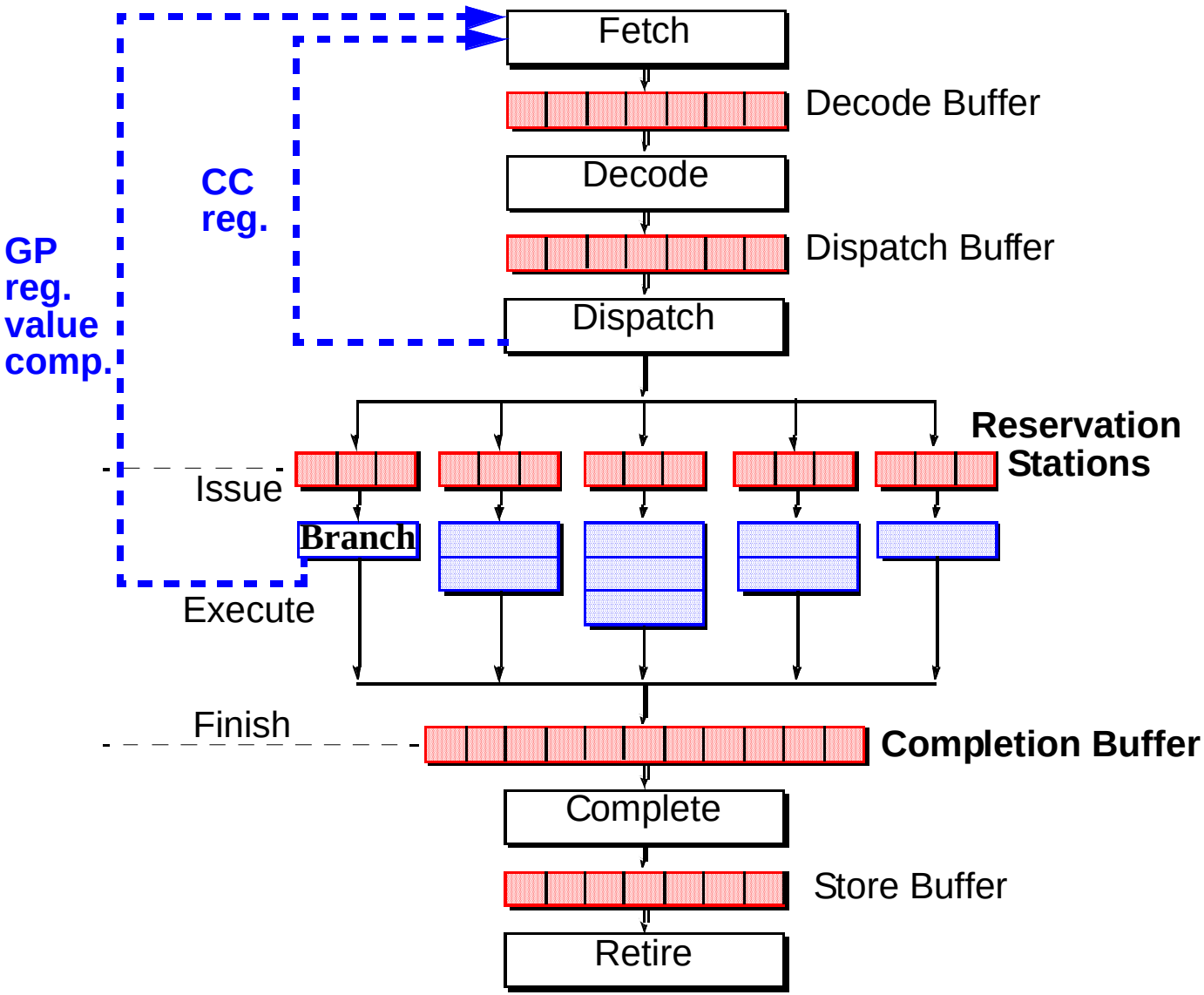
- Condition Resolution
  - Access register
    - Condition code register, data register, count register
  - Perform calculation
    - Comparison of data register(s)

  ⇒ Condition Speculation

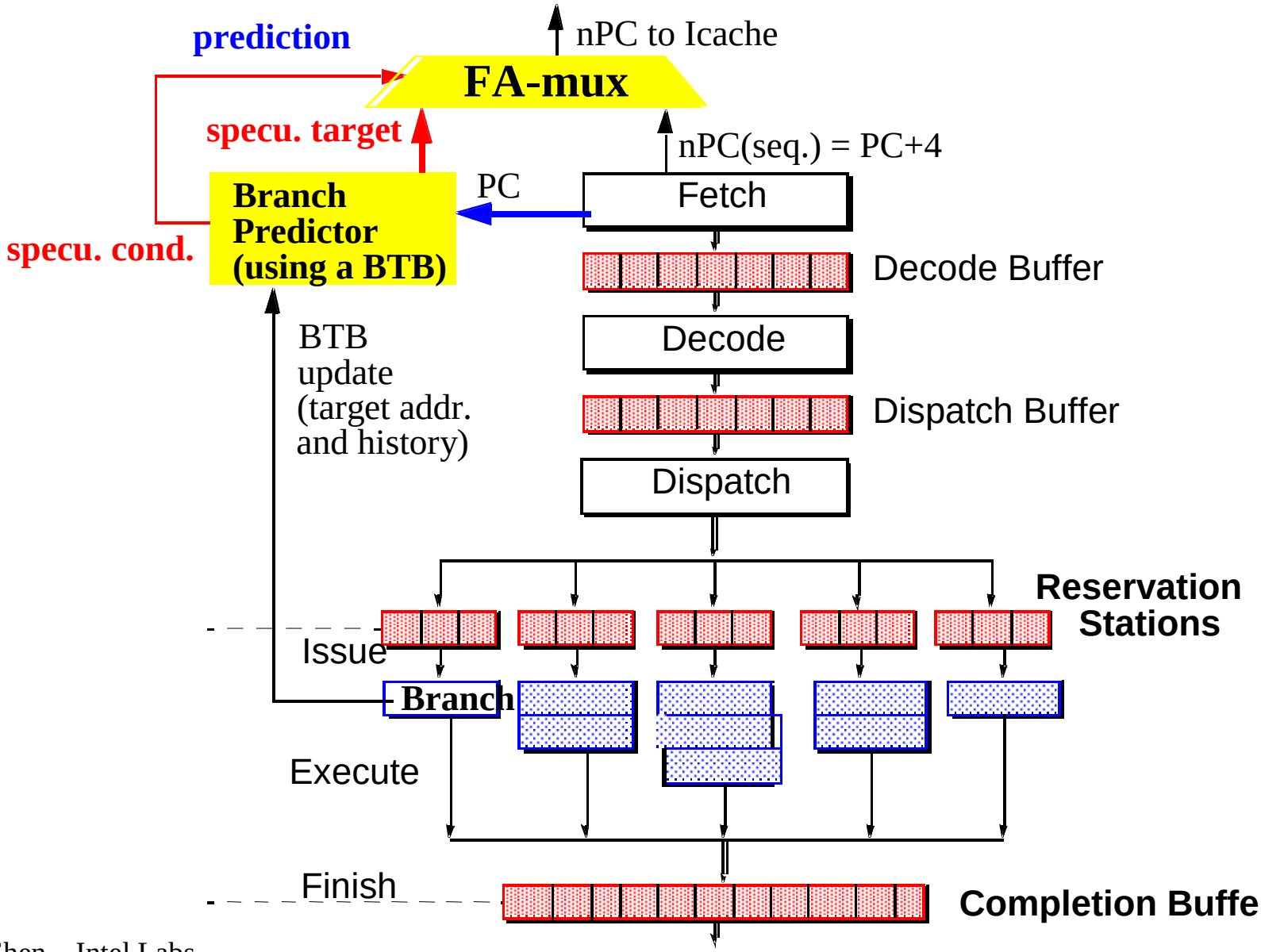
Target Address Generation



Condition Resolution

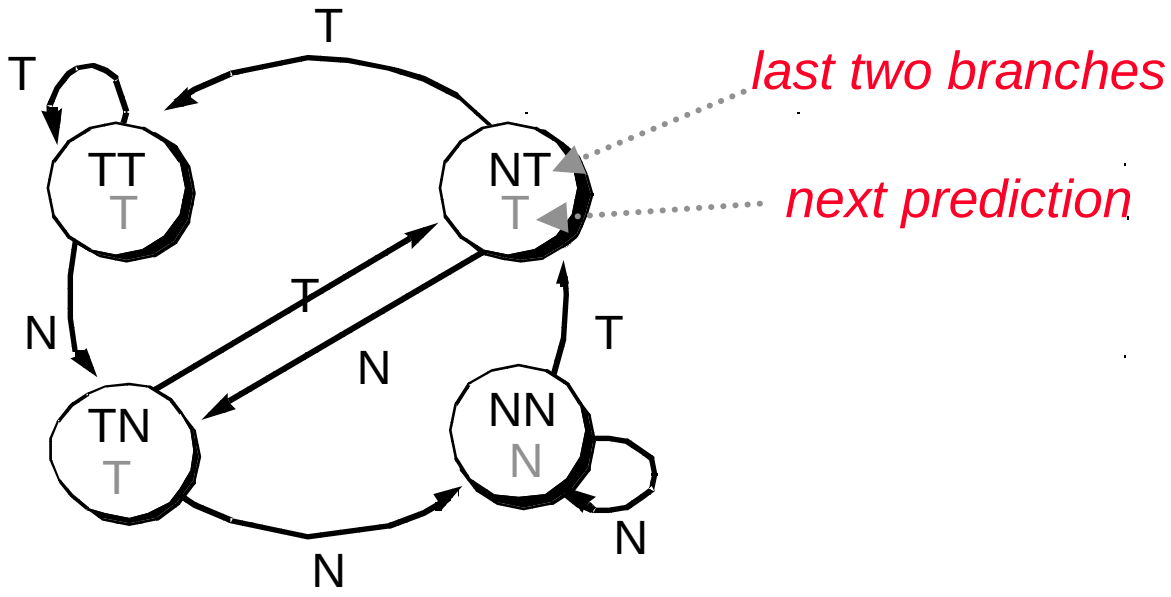


Branch Instruction Speculation



Example Prediction Algorithm

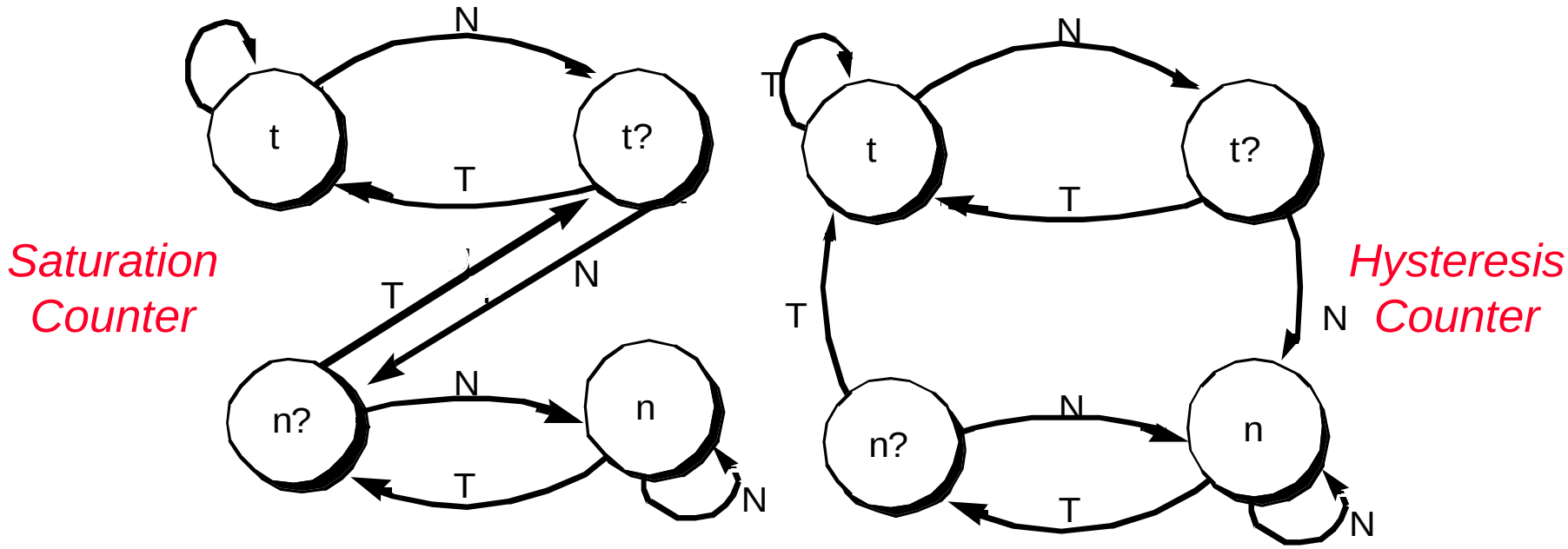
- Prediction accuracy approaches maximum with as few as 2 preceding branch occurrences used as history



Results (%) [IBM RS/6000 Study, Nair, 1992]

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

Other Prediction Algorithms

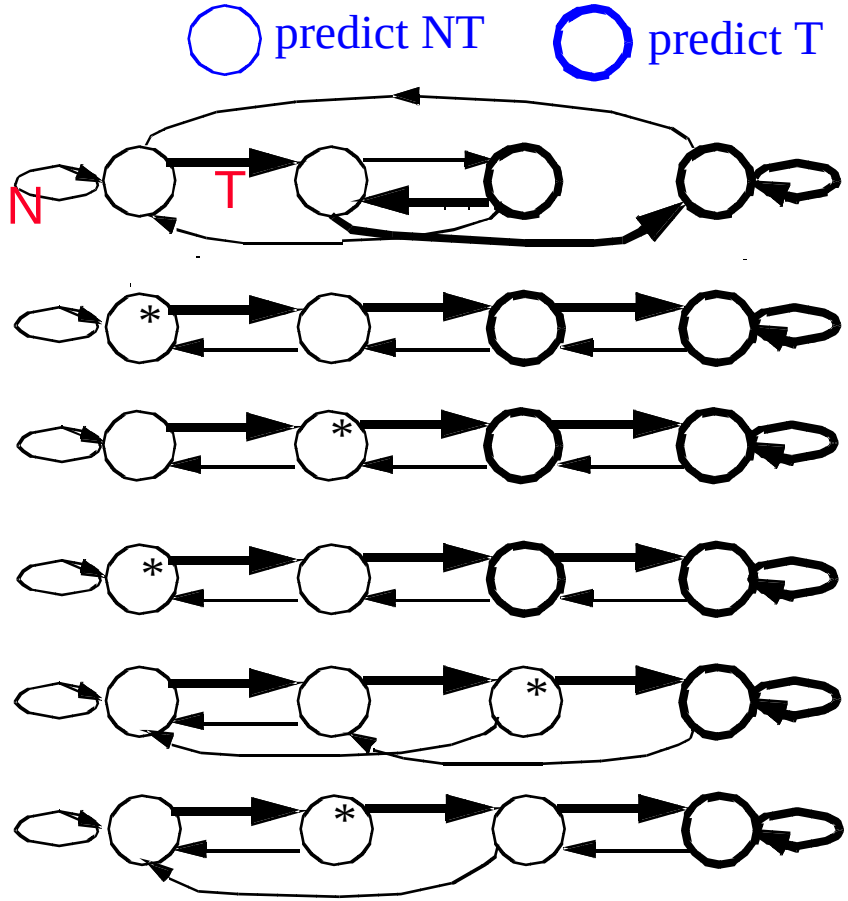


- Combining prediction accuracy with BTB hit rate (86.5% for 128 sets of 4 entries each), branch prediction can provide the net prediction accuracy of approximately 80%. This implies a 5-20% performance enhancement.

Optimal Predictor Exhaustive Search

- There are 2^{20} possible state machines of 2-bit predictors
- Pruning uninteresting and redundant machines leaves 5248
- It is possible to exhaustively search and find the *optimal* predictor for a benchmark

Benchmark	Best Pred. %
spice2g6	97.2
doduc	94.3
gcc	89.1
espresso	89.1
li	87.1
eqntott	87.9



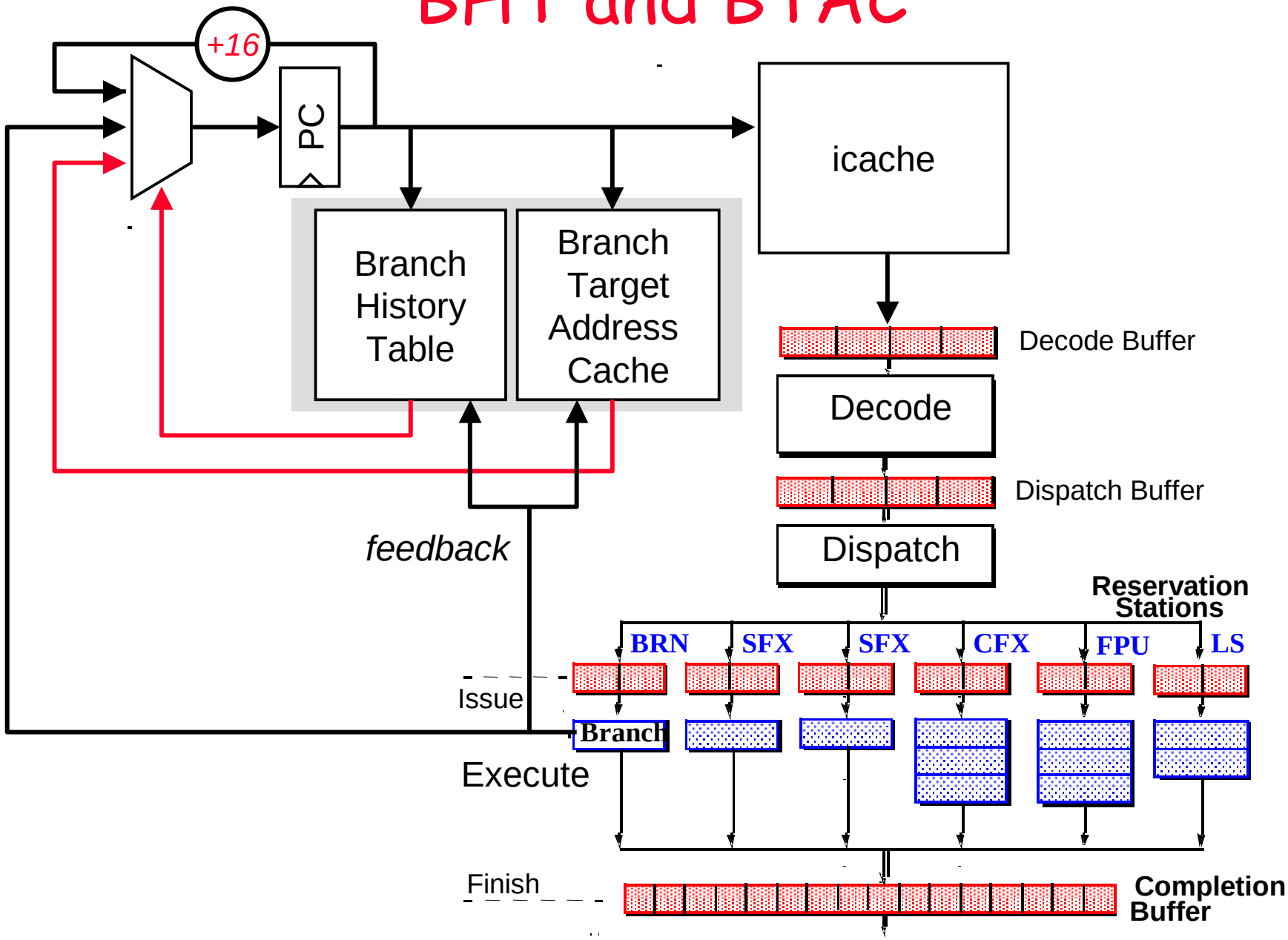
Saturation Counter is near optimal in all cases!

Number of Counter Bits Needed

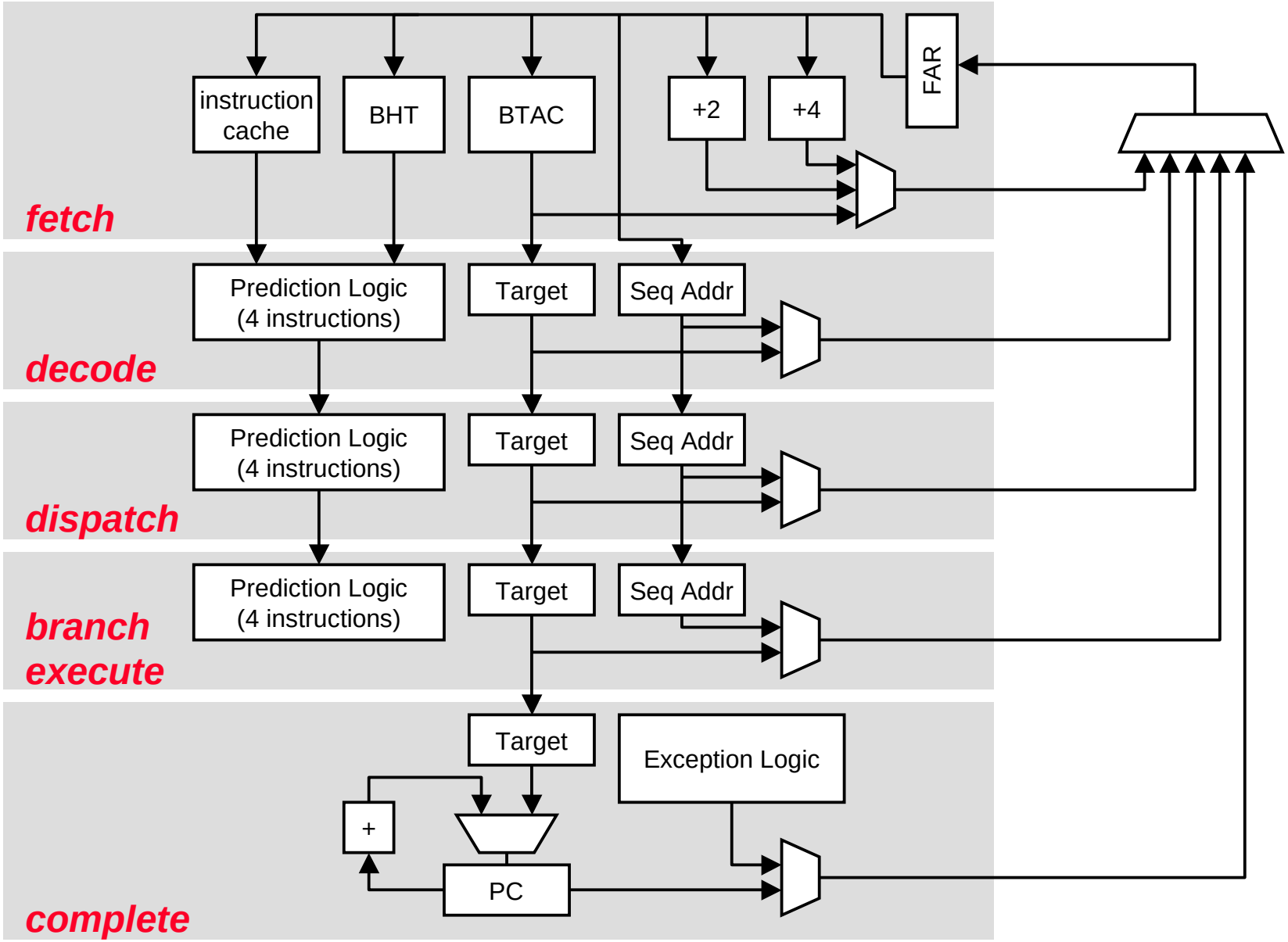
Benchmark	Prediction Accuracy (Overall CPI Overhead)			
	3-bit	2-bit	1-bit	0-bit
spice2g6	97.0 (0.009)	97.0 (0.009)	96.2 (0.013)	76.6 (0.031)
doduc	94.2 (0.003)	94.3 (0.003)	90.2 (0.004)	69.2 (0.022)
gcc	89.7 (0.025)	89.1 (0.026)	86.0 (0.033)	50.0 (0.128)
espresso	89.5 (0.045)	89.1 (0.047)	87.2 (0.054)	58.5 (0.176)
li	88.3 (0.042)	86.8 (0.048)	82.5 (0.063)	62.4 (0.142)
eqntott	89.3 (0.028)	87.2 (0.033)	82.9 (0.046)	78.4 (0.049)

- Branch history table size: Direct-mapped array of 2k entries
- Programs, like gcc, can have over 7000 conditional branches
- In collisions, multiple branches share the same predictor
- Variation of branch penalty with branch history table size level out at 1024

BHT and BTAC

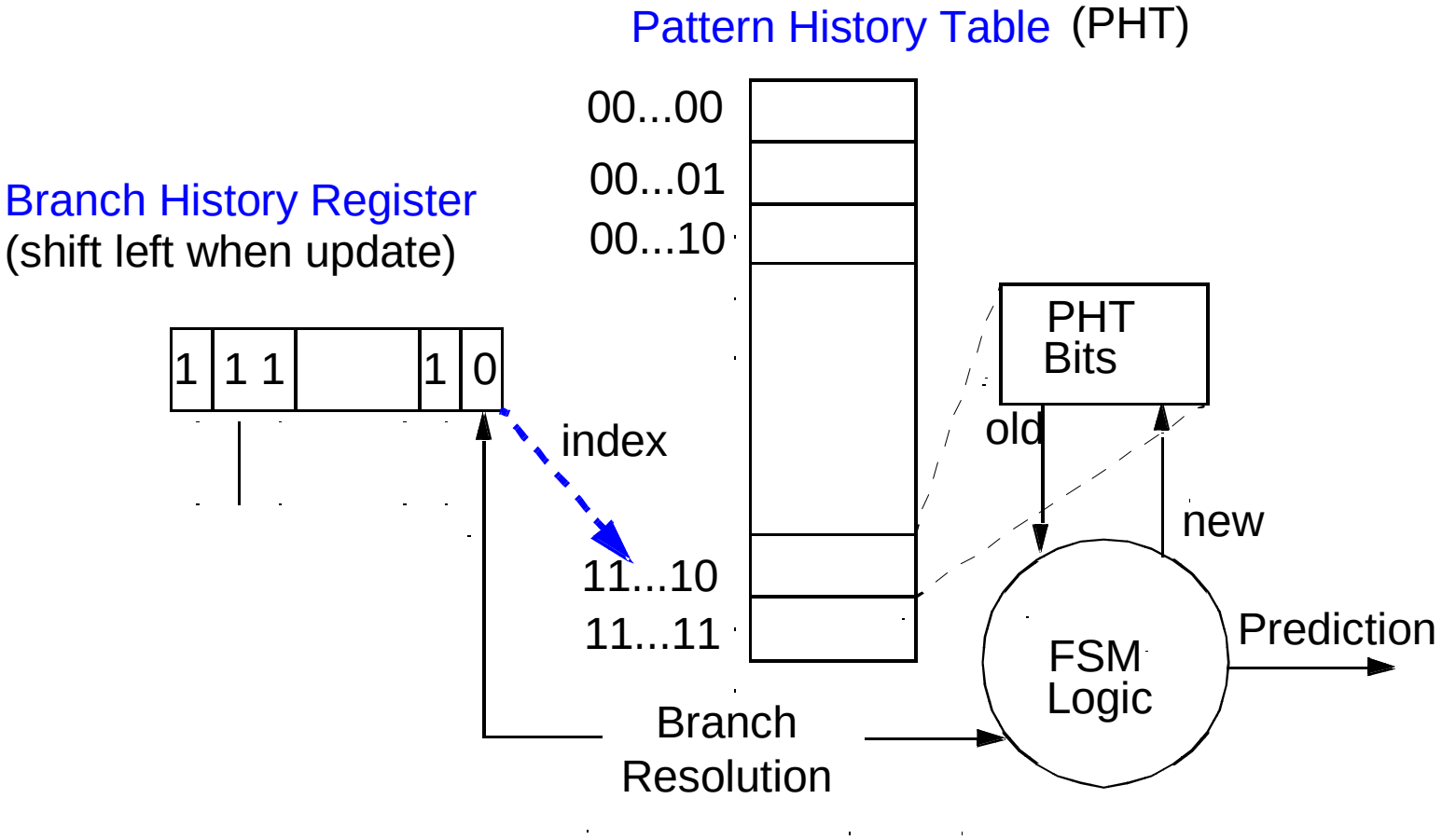

+16
PC
icache
Branch History Table
Branch Target Address Cache
Decode Buffer
Decode
Dispatch Buffer
Dispatch
feedback
Reservation Stations
BRN
SFX
SFX
CFX
FPU
LS
Issue
Branch
Execute
Finish
Completion Buffer

PPC 604 Fetch Address Generation



Global Branch Prediction

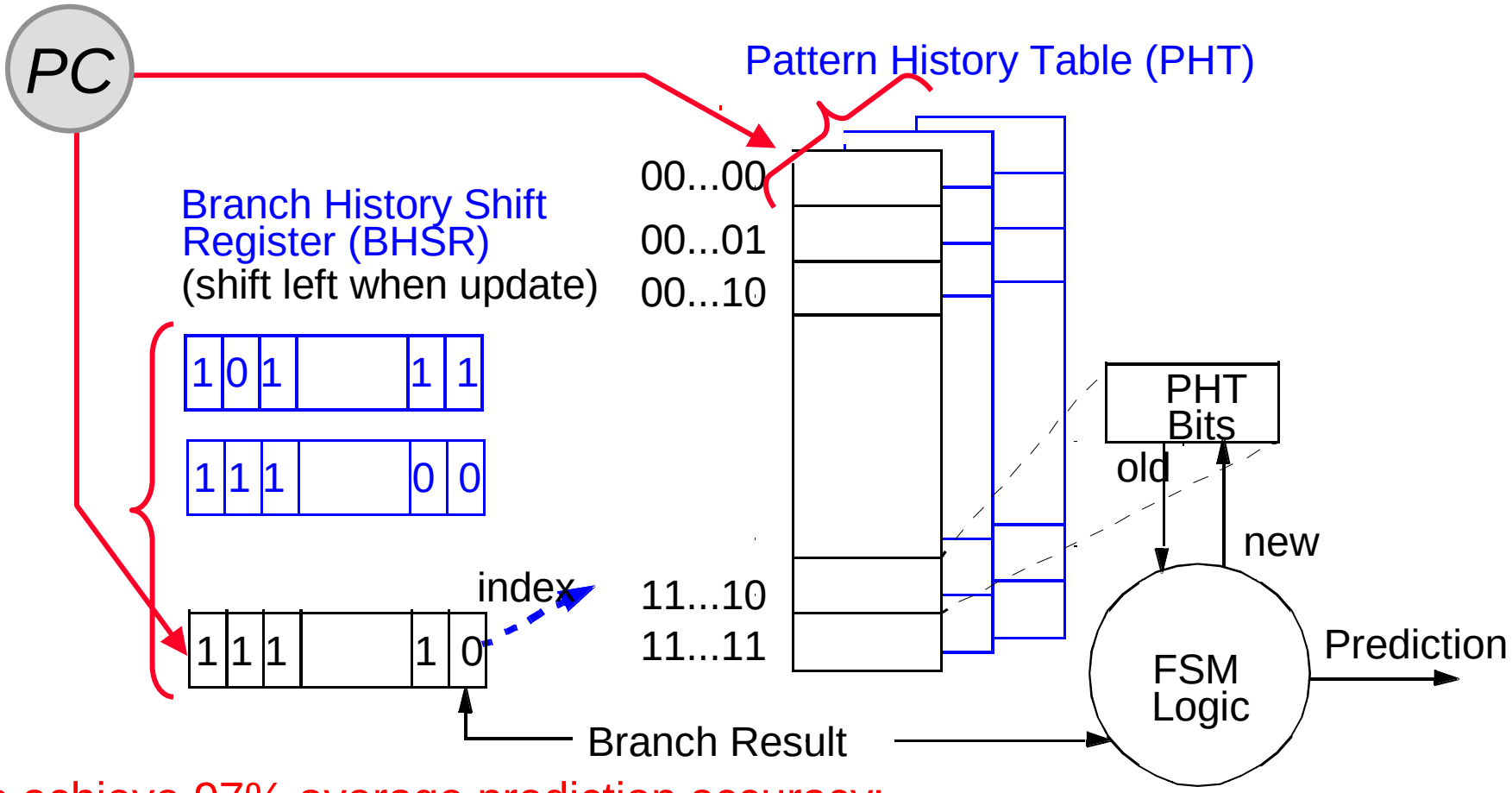
- So far, the prediction of each static branch instruction is based solely on its own past behavior and not the behaviors of other neighboring static branch instructions



2-Level Adaptive Prediction [Yeh & Patt]

- Two-level adaptive branch prediction
  - 1st level: History of last *k* (dynamic) branches encountered
  - 2nd level: branch behavior of the last *s* occurrences of the specific pattern of these *k* branches
  - Use a Branch History Register (BHR) in conjunction with a Pattern History Table (PHT)
- Example: (*k*=8, *s*=6)
  - Last *k* branches with the behavior (11100101)
  - *s*-bit History at the entry (11100101) is [101010]
  - Using history, branch prediction algorithm predicts direction of the branch
- Effectiveness:
  - Average 97% accuracy for SPEC
  - Used in the Intel P6 and AMD K6

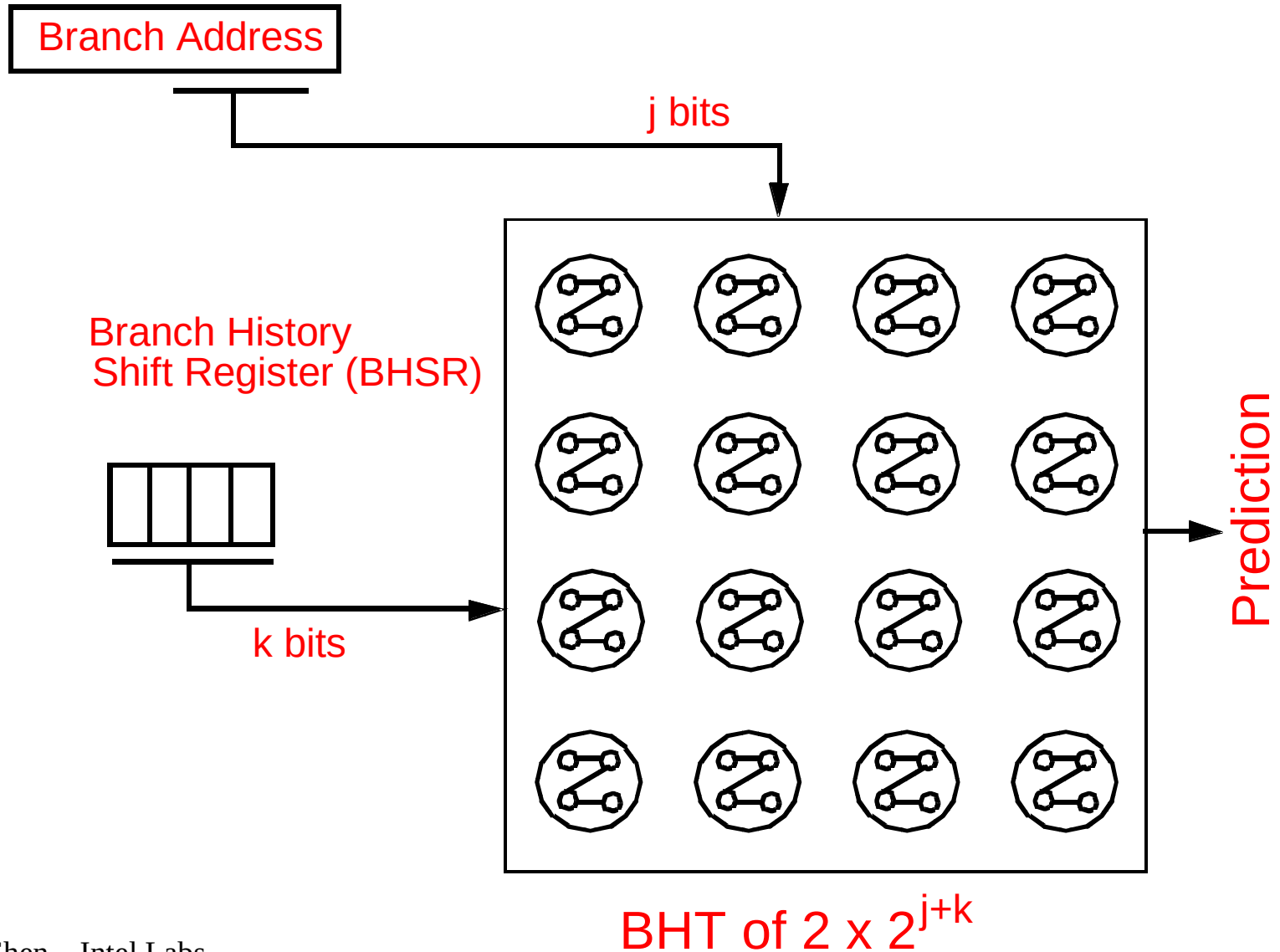
Nomenclature: {G,P}A{g,p,s}



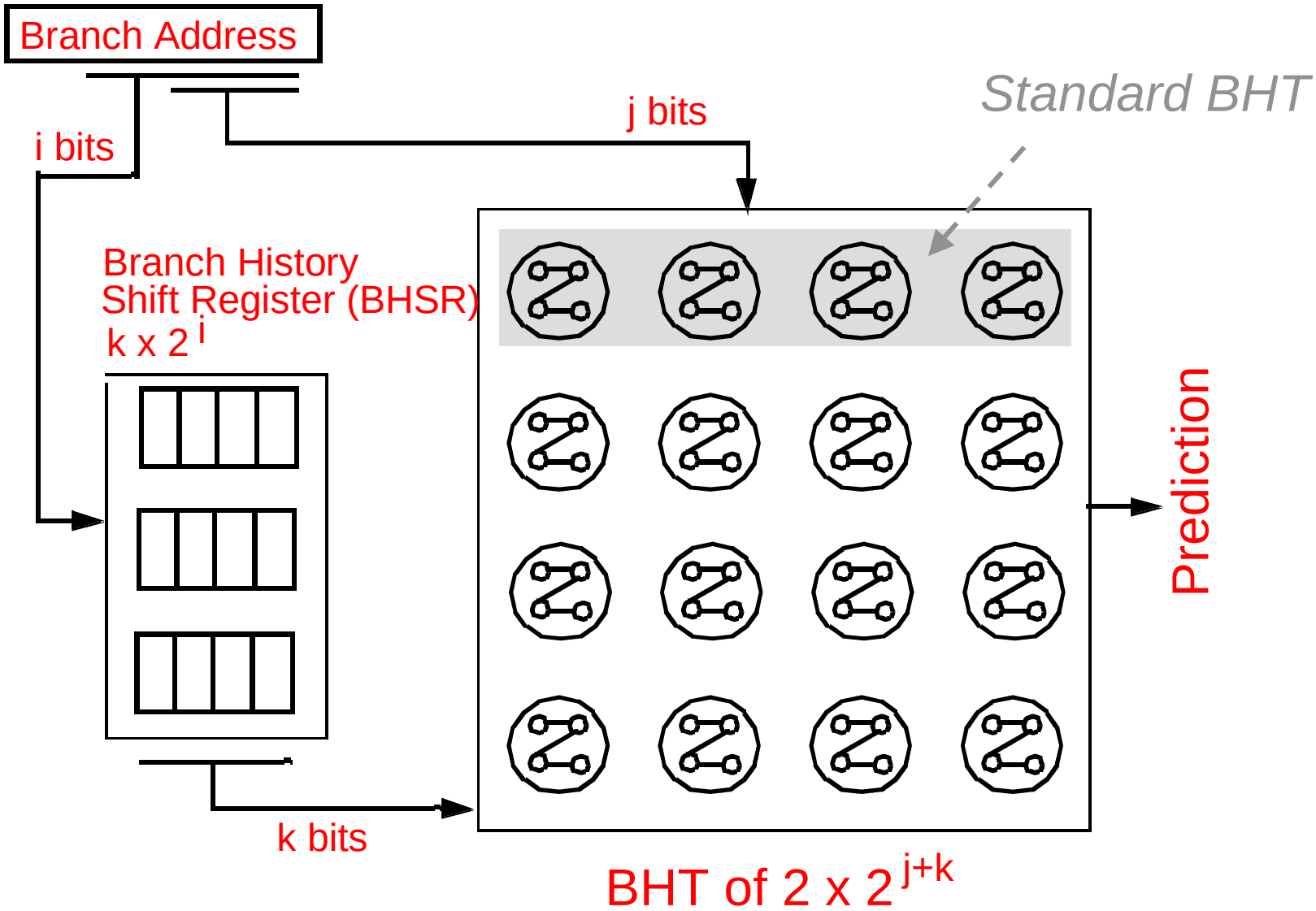
To achieve 97% average prediction accuracy:

G (1) BHR: 18 bits;	g (1) PHT: 2^{18} x 2 bits	*total = 524 kbits*
P (512x4) BHR: 12 bits;	g (1) PHT: 2^{12} x 2 bits	*total = 33 kbits*
P (512x4) BHR: 6 bits;	s (512) PHT: 2^{6} x 2 bits	*total = 78 kbits*

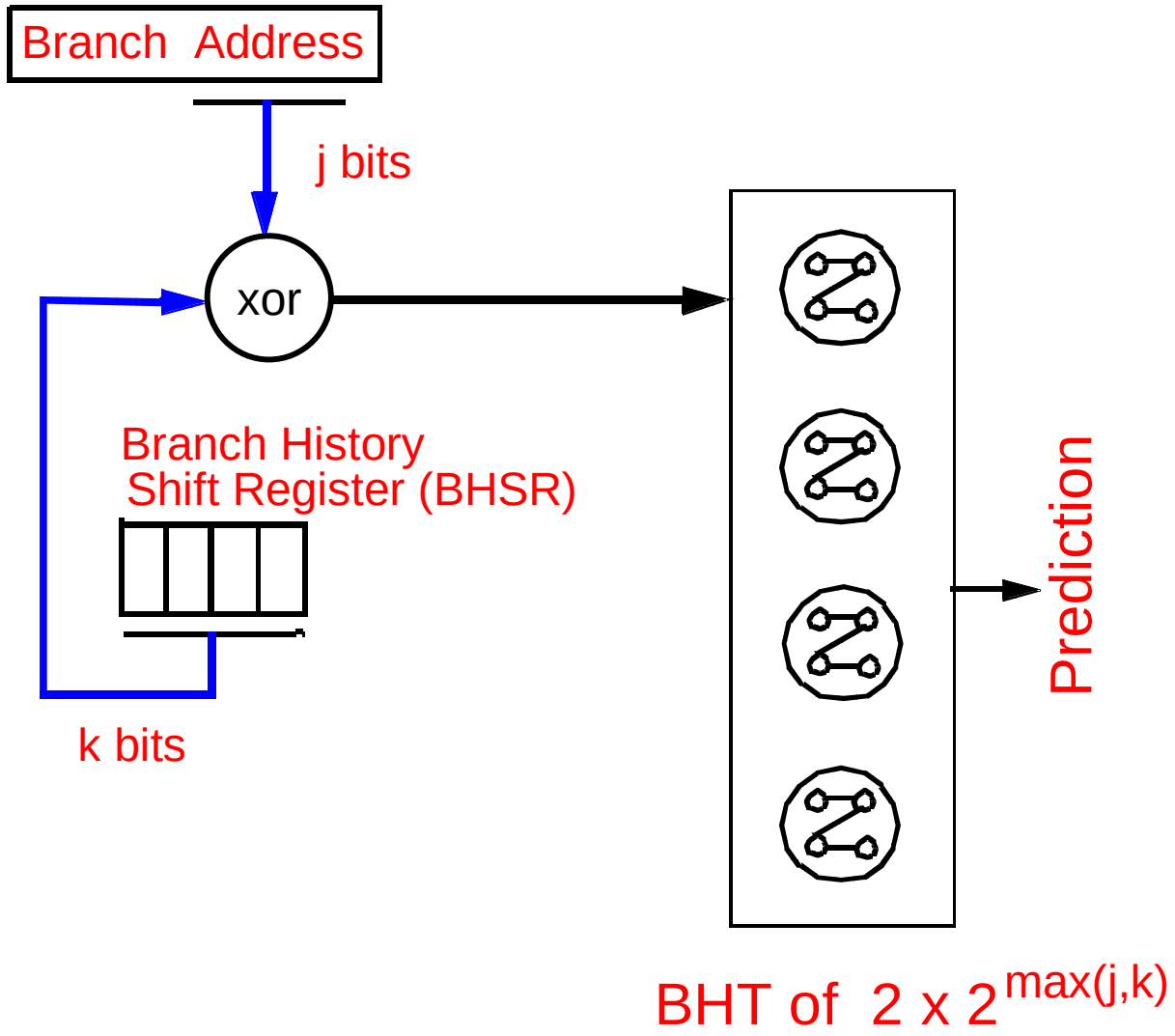
Global BHSR Scheme (GAs)



Per-Branch BHSR Scheme (PAs)



Gshare Branch Prediction [McFarling]



5. Register Data Flow Techniques (ALU Penalty)

Inter-instruction Dependences

- *Data dependence*

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

- *Anti-dependence*

$r_3 \leftarrow r_1$ op r_2
$r_1 \leftarrow r_4$ op r_5

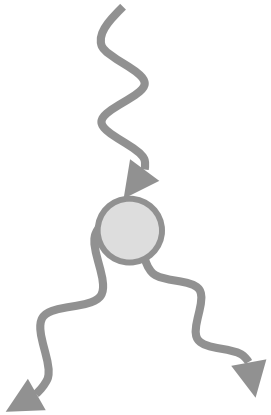
Write-after-Read (WAR)

- *Output dependence*

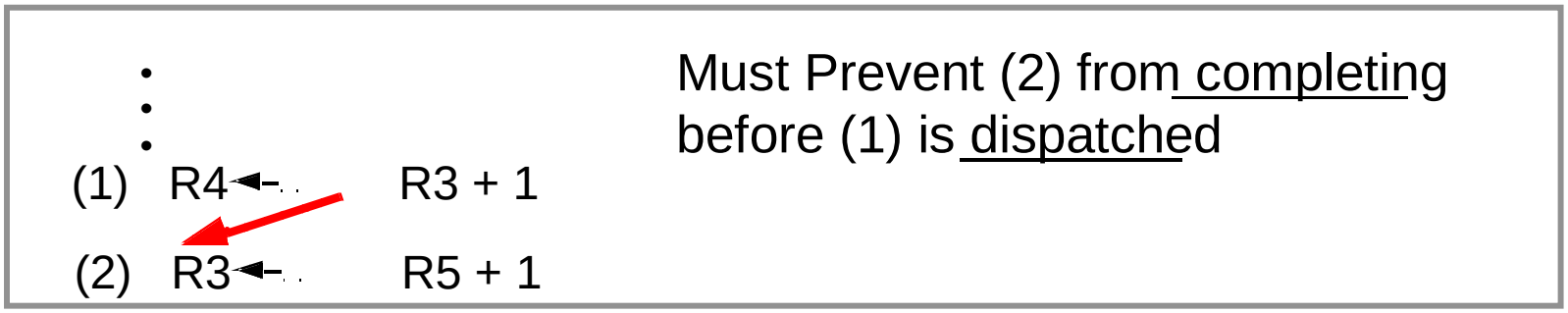
$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

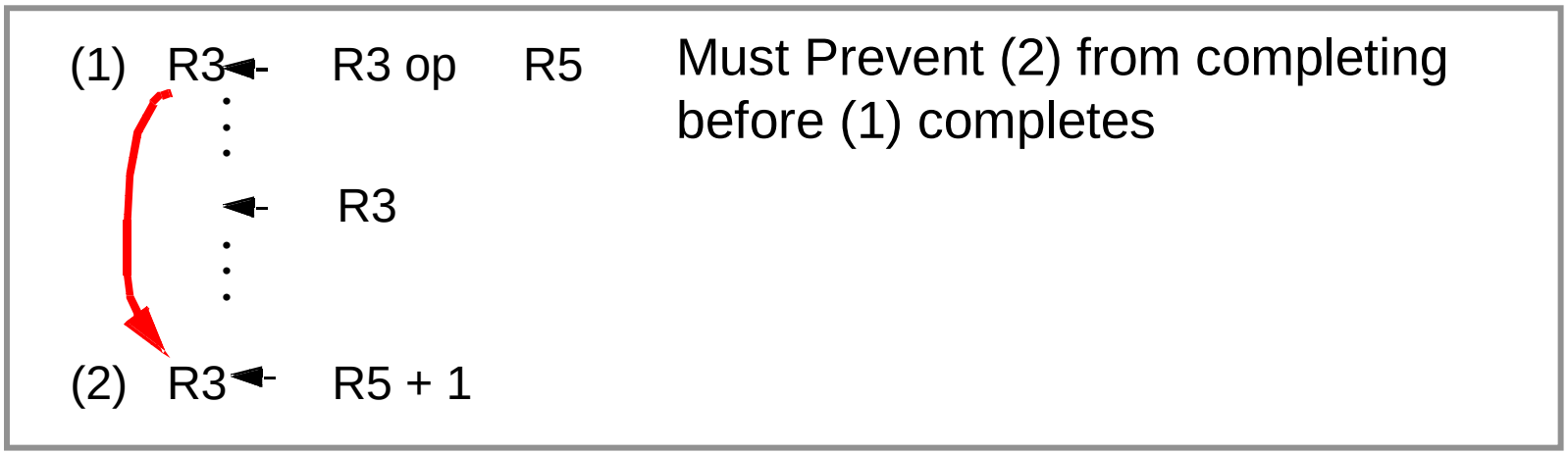
Write-after-Write (WAW)

- Control dependence



Resolving False Dependences





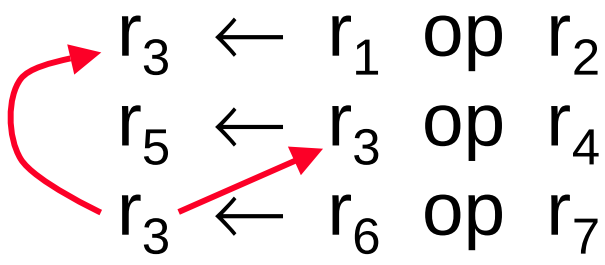
Stalling: delay Dispatching (or write back) of the 2nd instruction

Copy Operands: Copy not-yet-used operand to prevent being overwritten (WAR)

Register Renaming: use a different register (WAW & WAR)

Register Renaming

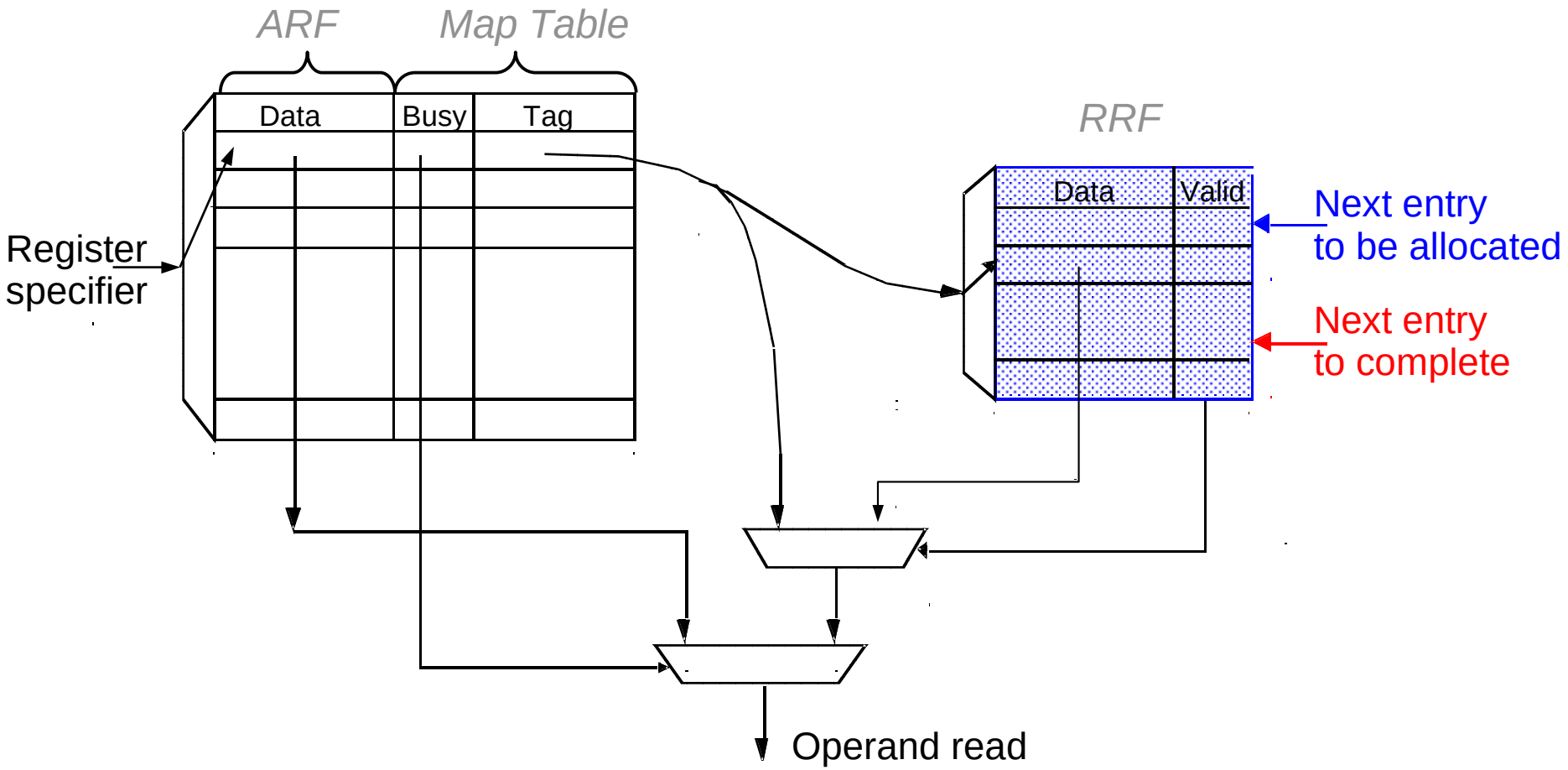
- Anti and output dependencies are false dependencies



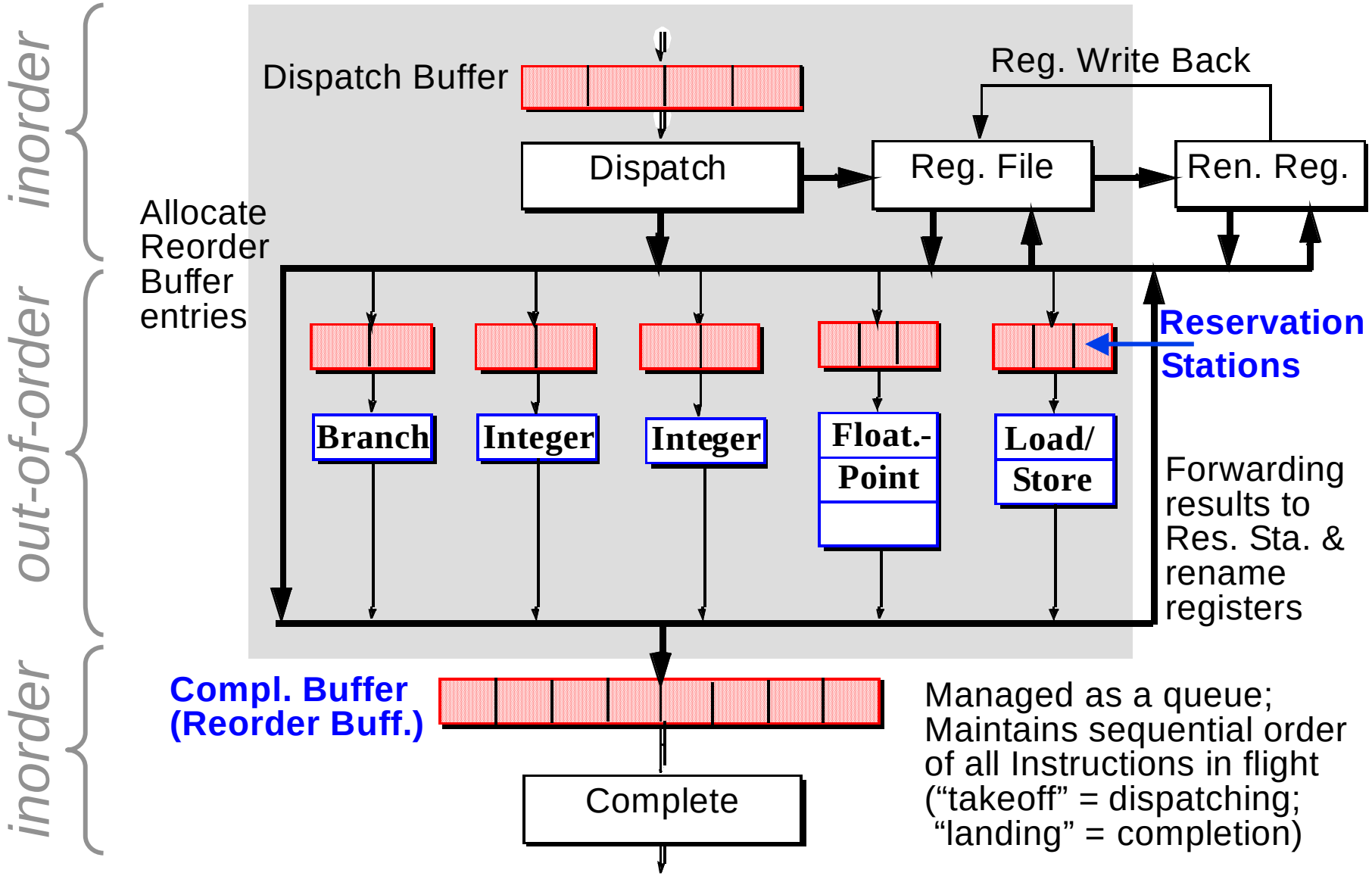
- The dependence is on name/location rather than data
- Given infinite number of registers, anti and output dependencies can always be eliminated

Original	Renamed
r1 ← r2 / r3	r1 ← r2 / r3
r4 ← r1 * r5	r4 ← r1 * r5
r1 ← r3 + r6	r8 ← r3 + r6
r3 ← r1 - r4	r9 ← r8 - r4

Register Renaming Mechanisms



Elements of Modern Micro-dataflow



6. Memory Data Flow Techniques (Load Penalty)

Total Ordering of Loads & Stores

- Keep all loads and stores totally in order with respect to each other

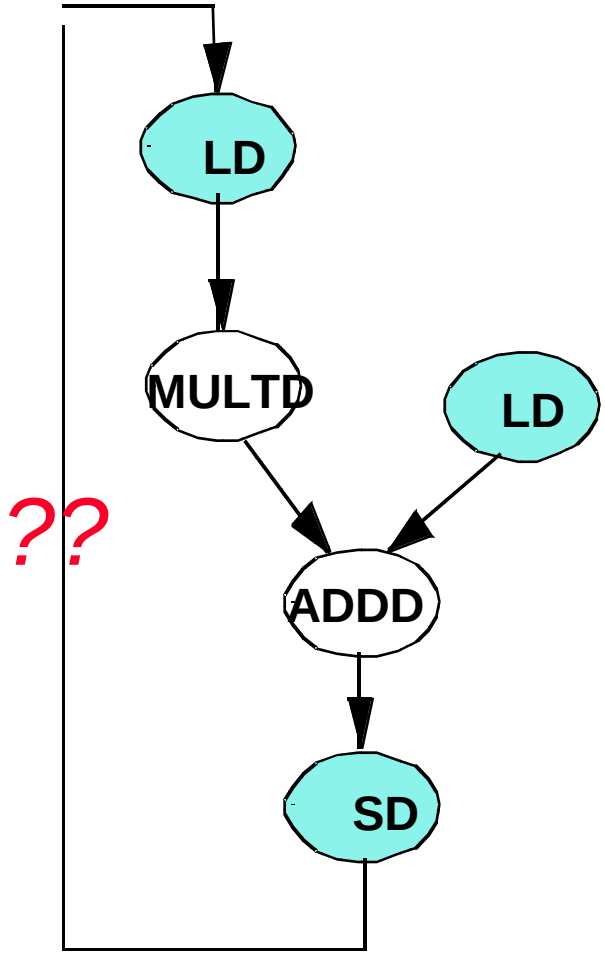
- However, loads and stores can execute out of order with respect to other types of instructions (while obeying register data-dependences)

Except, stores must still be held for all previous instructions

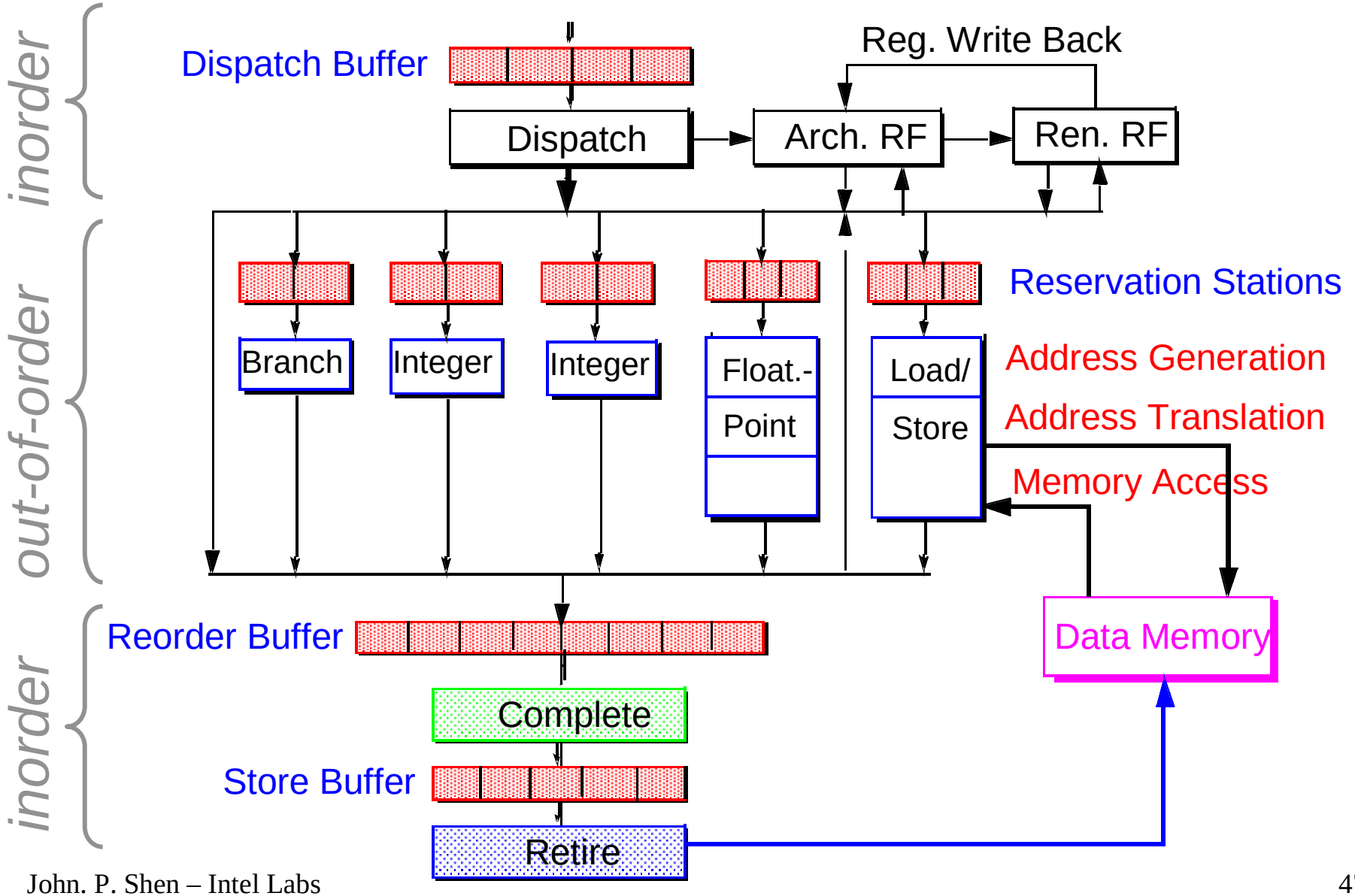
The "DAXPY" Example

Y[i] = A * X[i] + Y[i]

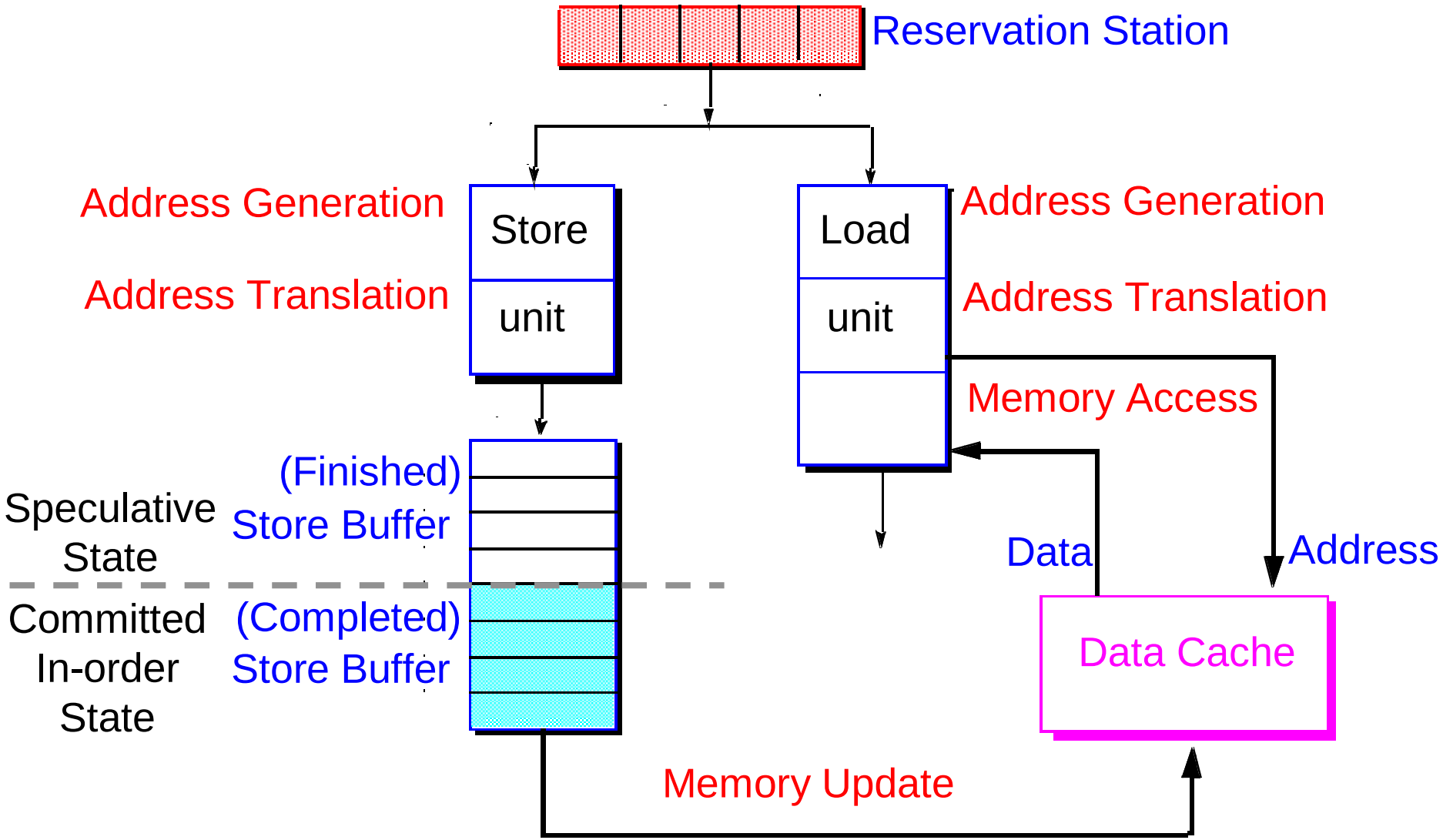
```
        LD      F0, a
        ADDI    R4, Rx, #512 ; last address
Loop:
        LD      F2, 0(Rx)     ; load X[ i ]
        MULTD   F2, F0, F2    ; A*X[ i ]
        LD      F4, 0(Ry)     ; load Y[ i ]
        ADDD    F4, F2, F4    ; A*X[ i ] + Y[ i ]
        SD      F4, 0(Ry)     ; store into Y[ i ]
        ADDI    Rx, Rx, #8    ; inc. index to X
        ADDI    Ry, Ry, #8    ; inc. index to Y
        SUB     R20, R4, Rx   ; compute bound
        BNZ     R20, loop     ; check if done
```



Processing of Load/Store Instructions



Load/Store Units & Store Buffer

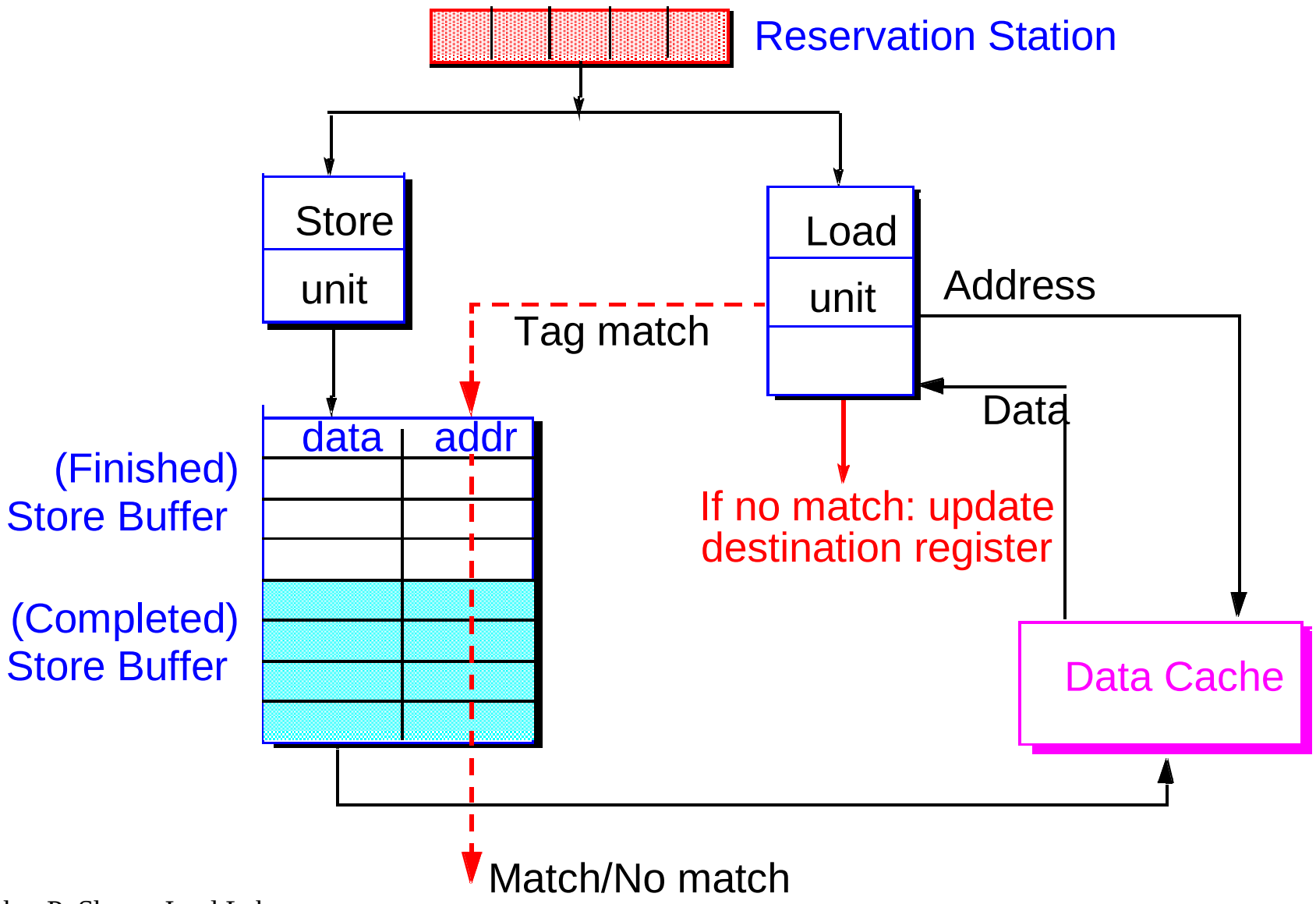


Load Bypassing

- Loads can be allowed to bypass older stores if no aliasing is found
  - Older stores' addresses must be computed before loads can be issued to allow checking for RAW
- Alternatively, a load can assume no aliasing and bypass older stores *speculatively*
  - validation of no aliasing with previous stores must be done and provide mechanism for reversing the effect
- Stores are kept in ROB (or Finished Store Buffer) until all older instructions complete
- At completion time, a store is moved to the Completed Store Buffer to wait for turn to access cache

Store is consider completed. Latency beyond this point has little effect on the processor throughput

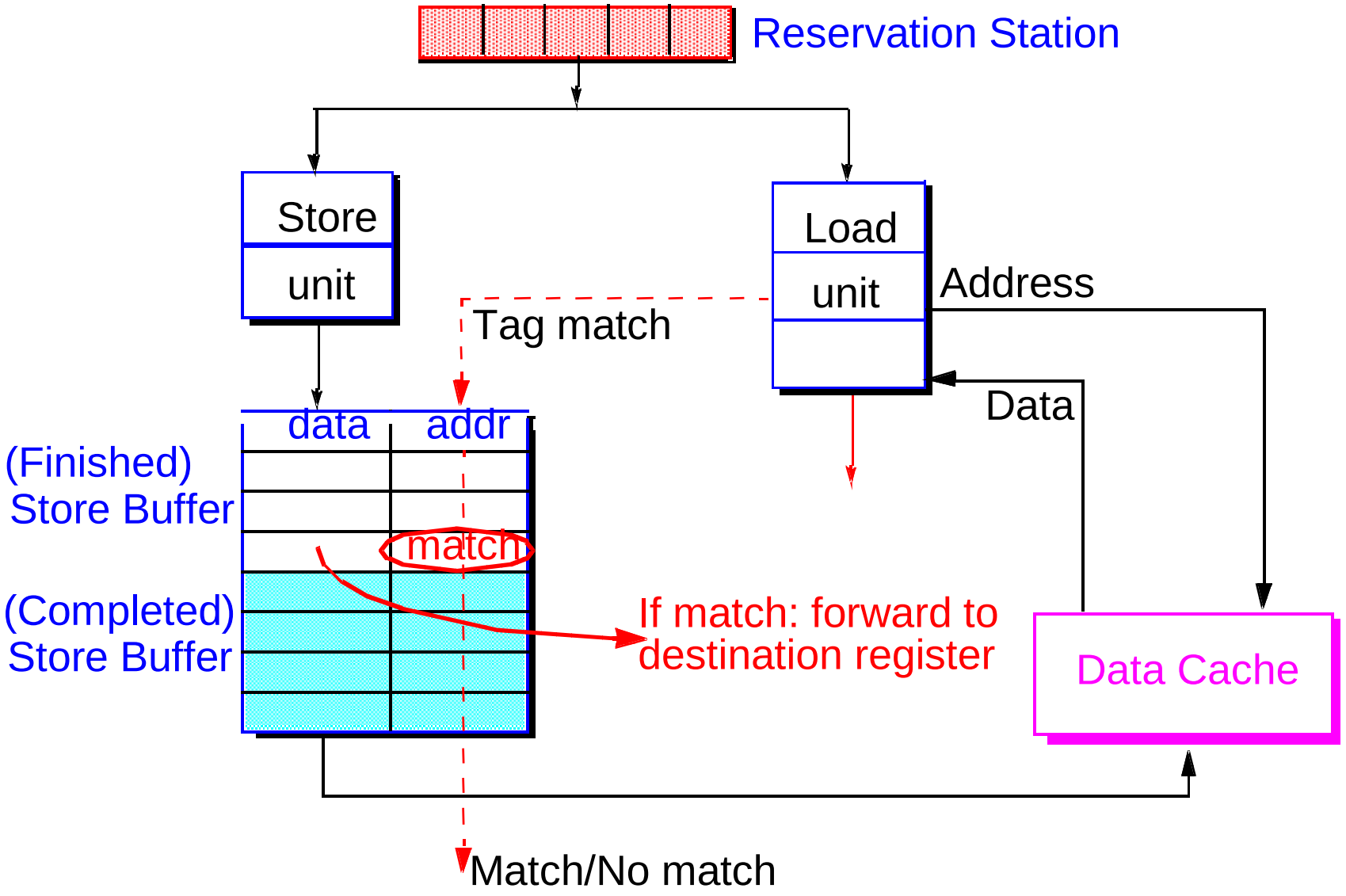
Illustration of Load Bypassing



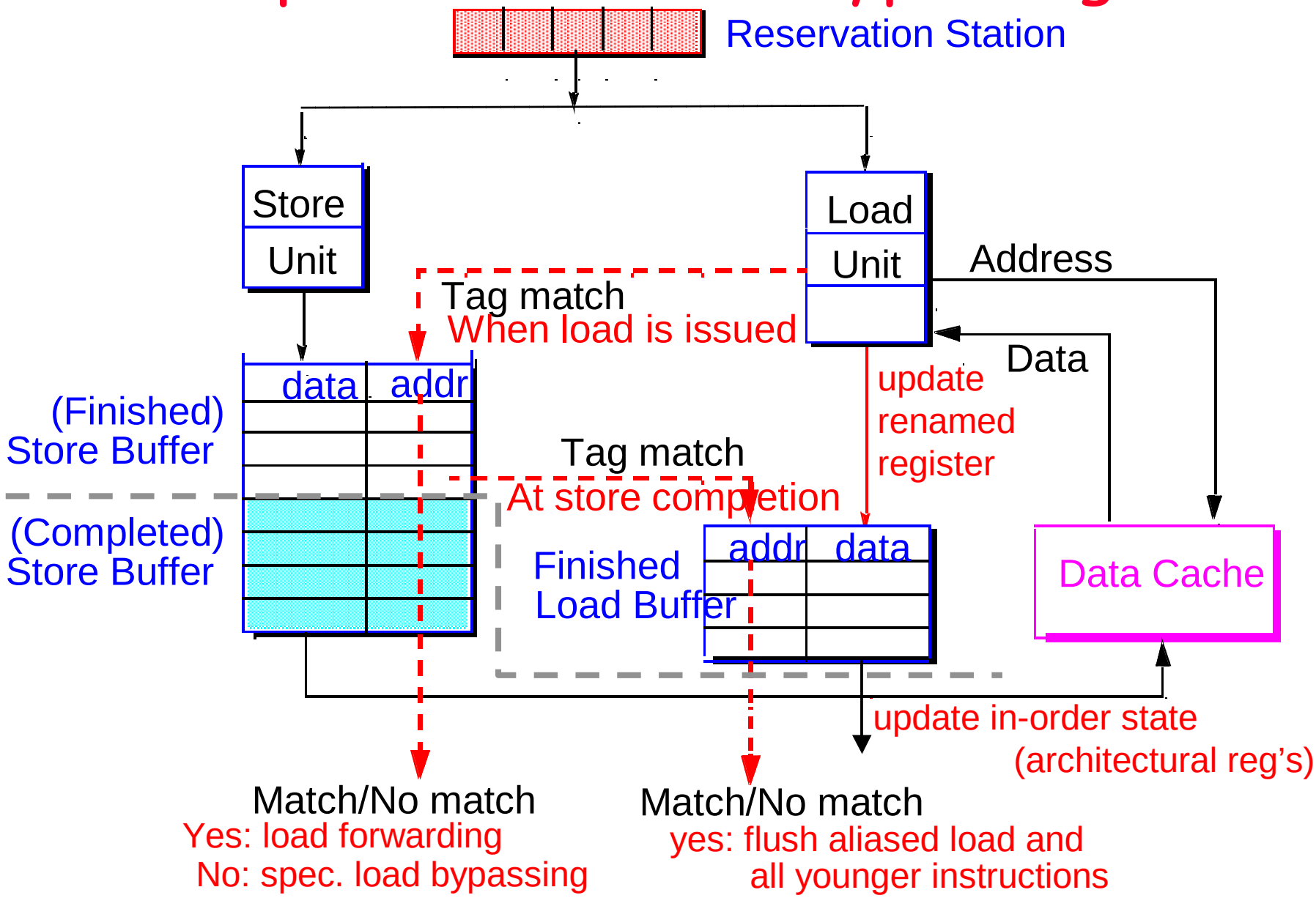
Load Forwarding

- If a pending load is RAW dependent on an earlier store still in the store buffer, it need not wait till the store is issued to the data cache
- The load can be directly satisfied from the store buffer if both load and store addresses are valid and the data is available in the store buffer
- This avoids the latency of accessing the data cache

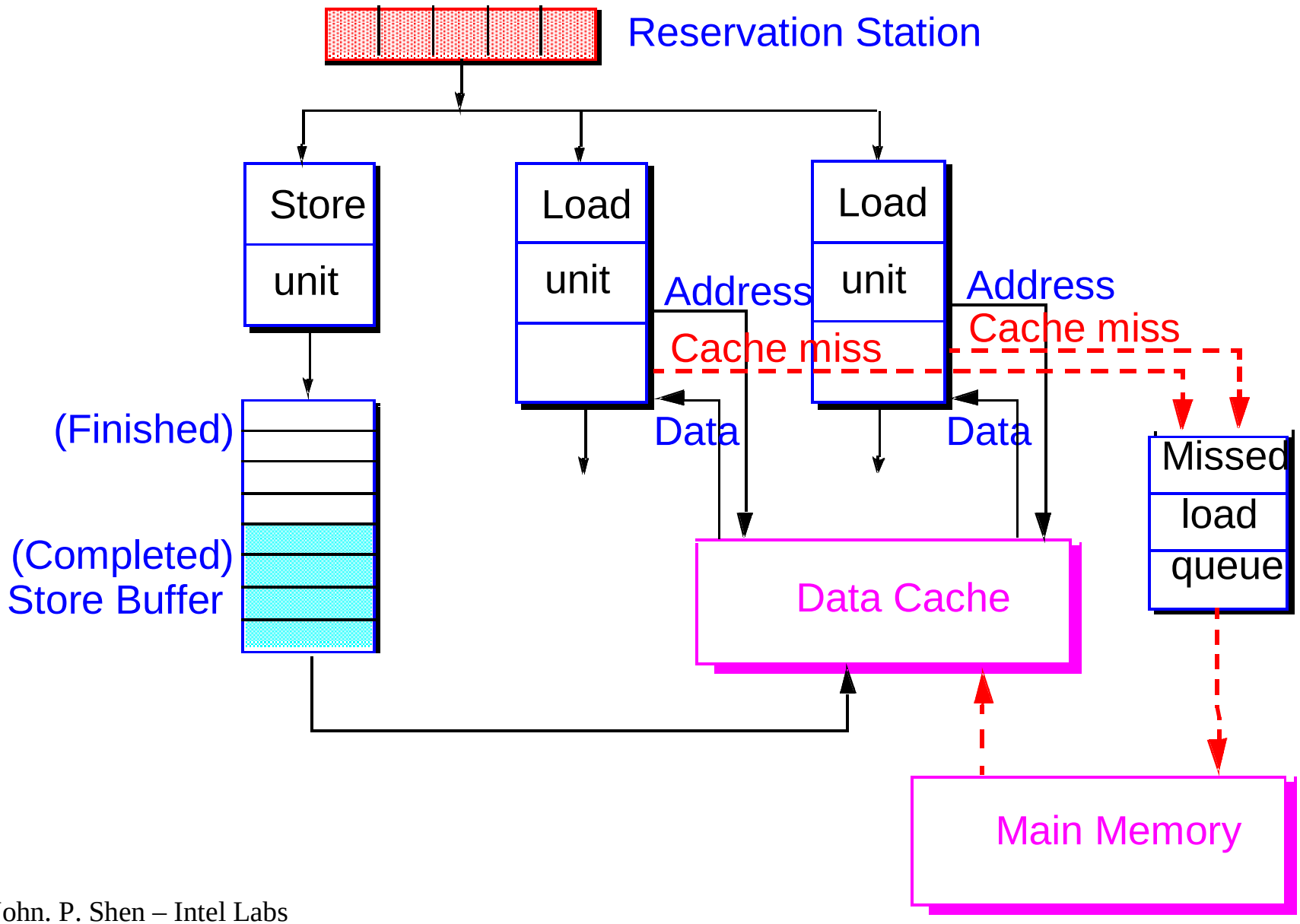
Illustration of Load Forwarding



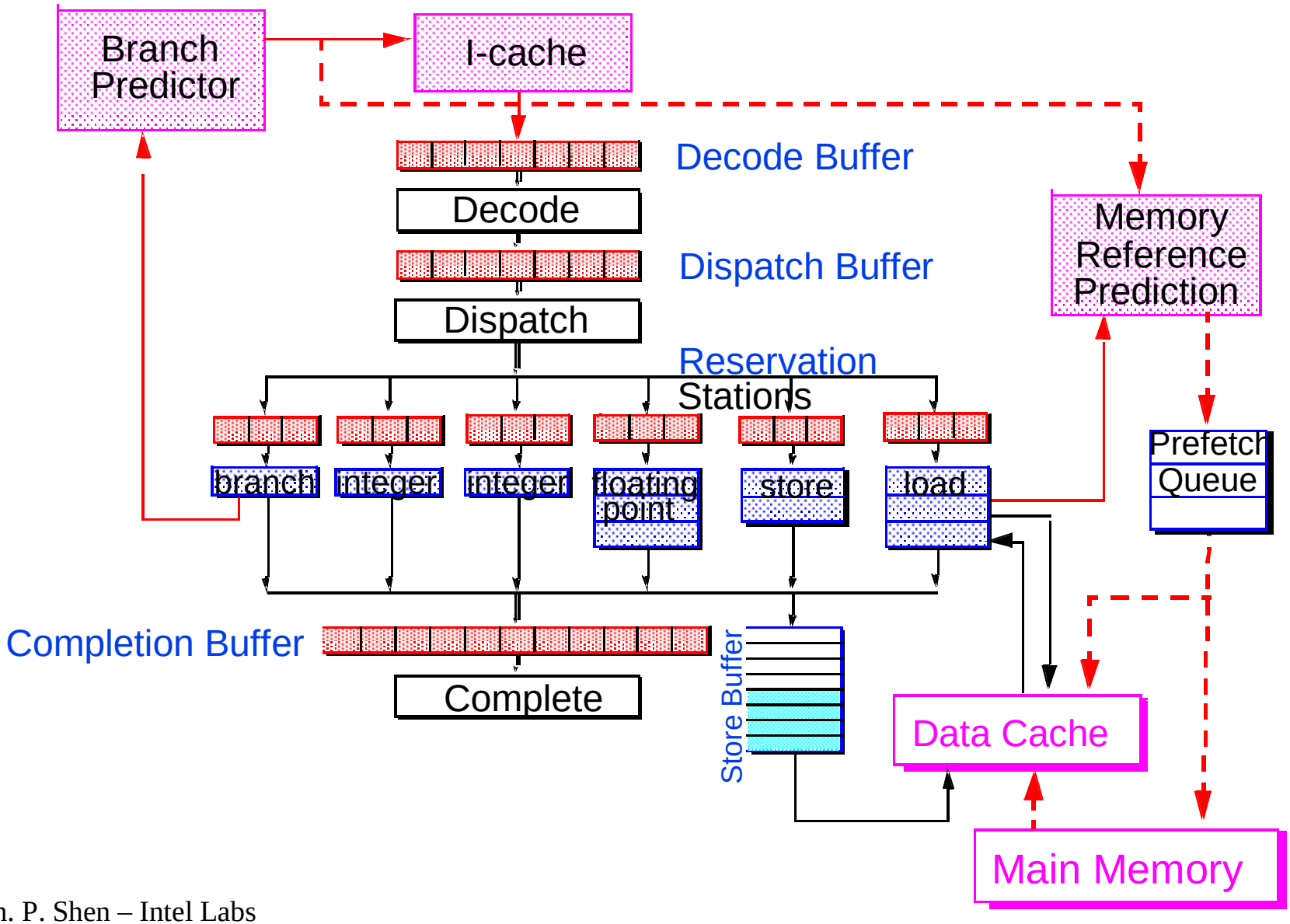
Speculative Load Bypassing



Dual-Ported Non-Blocking Cache



Prefetching Data Cache



7. Current Challenges in Superscalar Design

"Iron Law" of Processor Performance

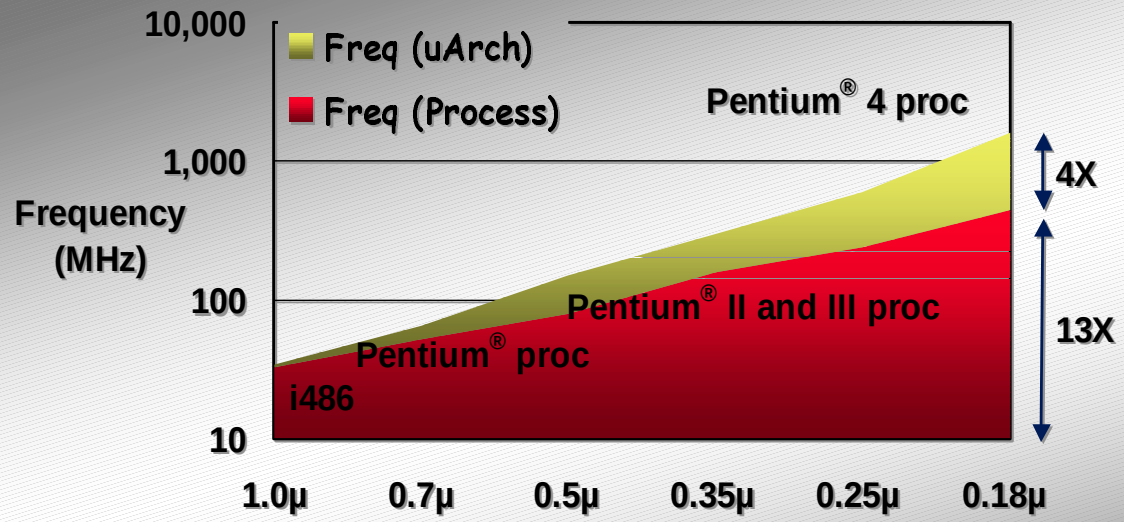
$$1/\text{Processor Performance} = \frac{\text{Wall-Clock Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(instr. count) *(CPI)* *(cycle time)*

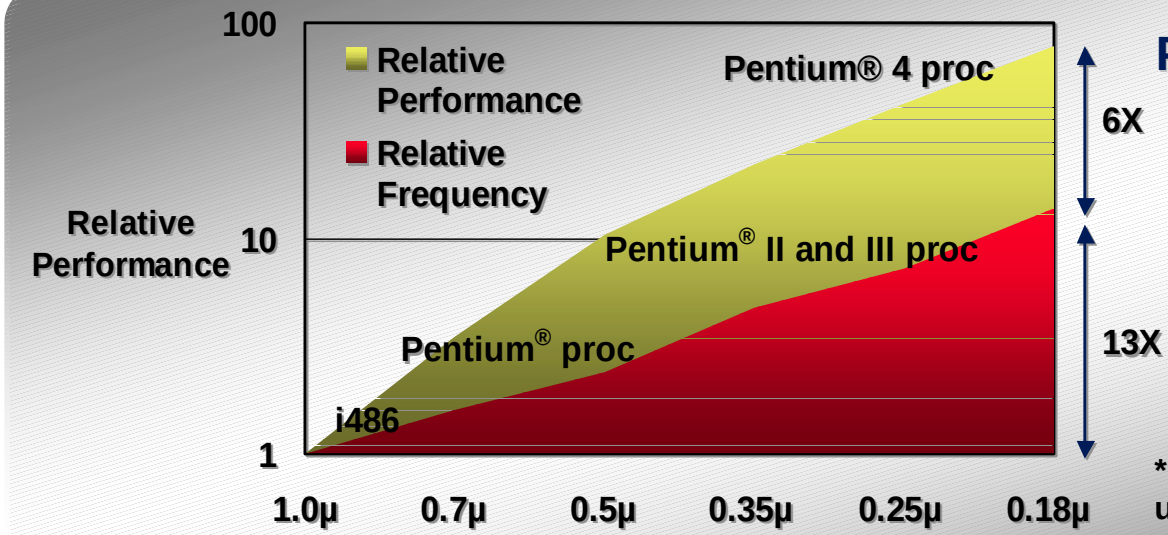
$$\text{Processor Performance} = \frac{\text{IPC x GHz}}{\text{Instr. Count}}$$

Frequency & Performance Boost



Frequency Increased 50X

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



Performance Increased >75X

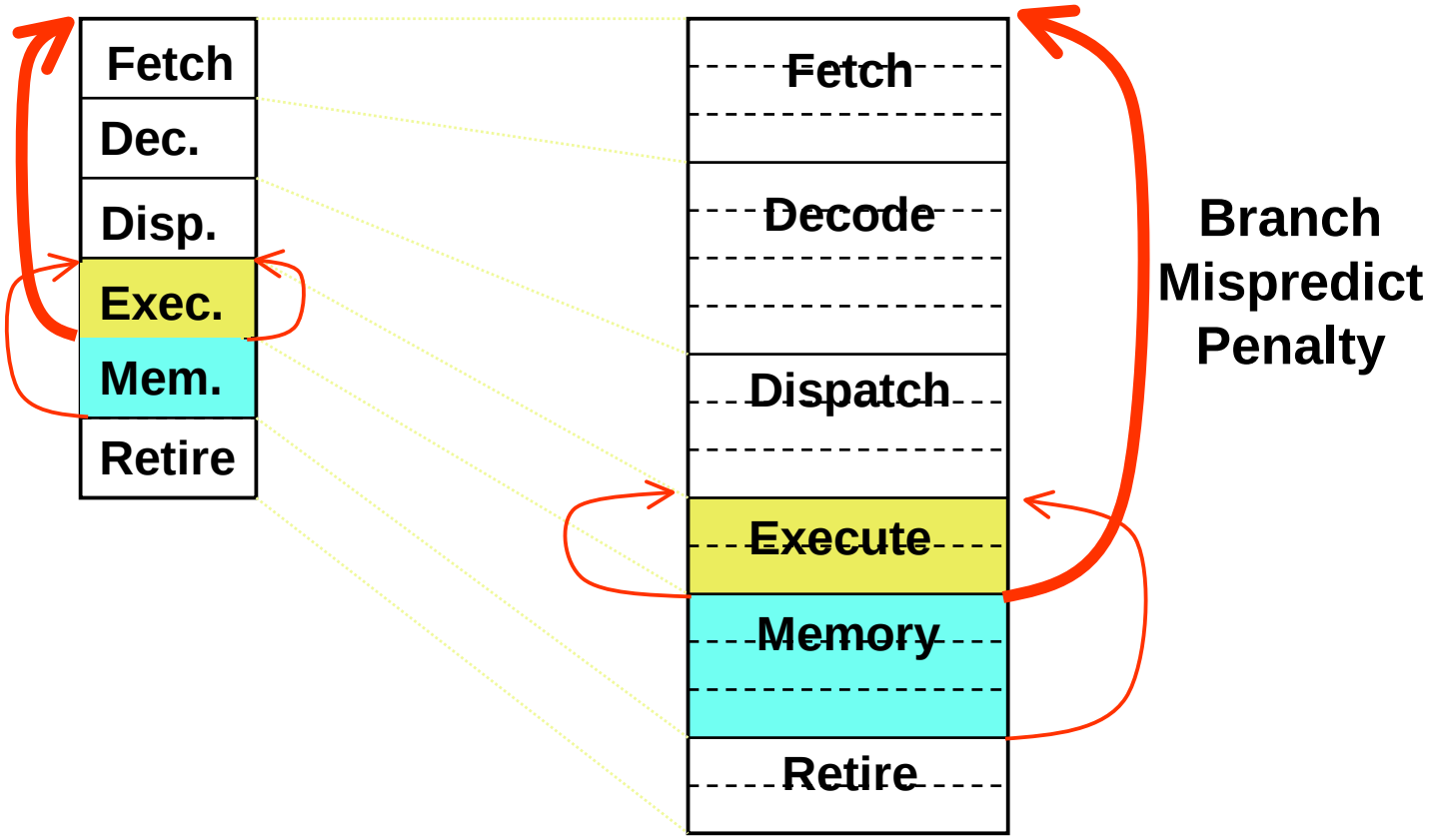
- 13X due to process technology
- Additional >6X due to microarchitecture

***Note: Performance measured using SpecINT and SpecFP**

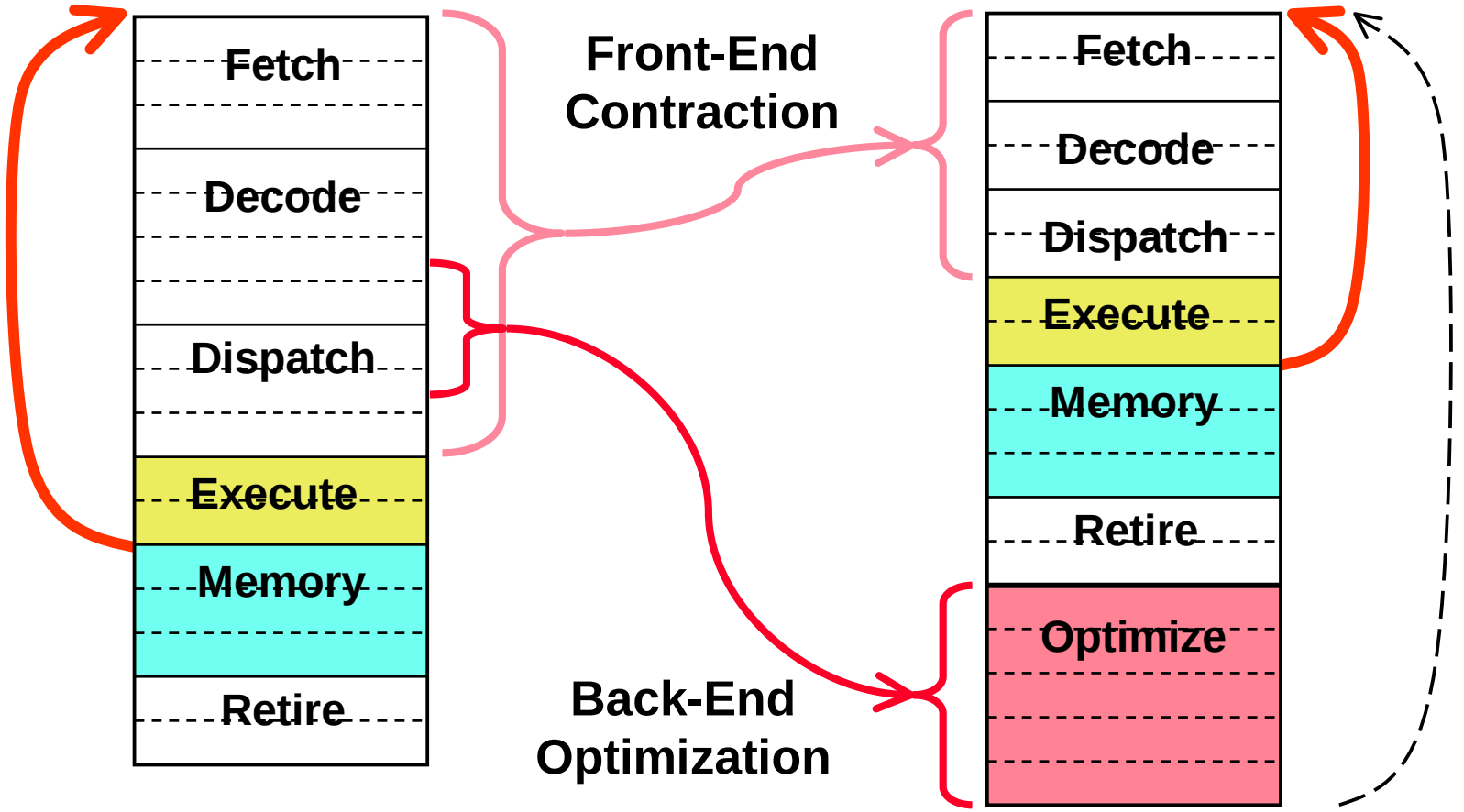
Frequency vs. Parallelism

- Increase Frequency (GHz)
  - Deeper Pipelines
  - Increased Overall Latency
  - Lower IPC
- Increase Instruction Parallelism (IPC)
  - Wider Pipelines
  - Increased Complexity
  - Lower GHz

Deeper and Wider Pipelines



Front-End Pipe-Depth Penalty

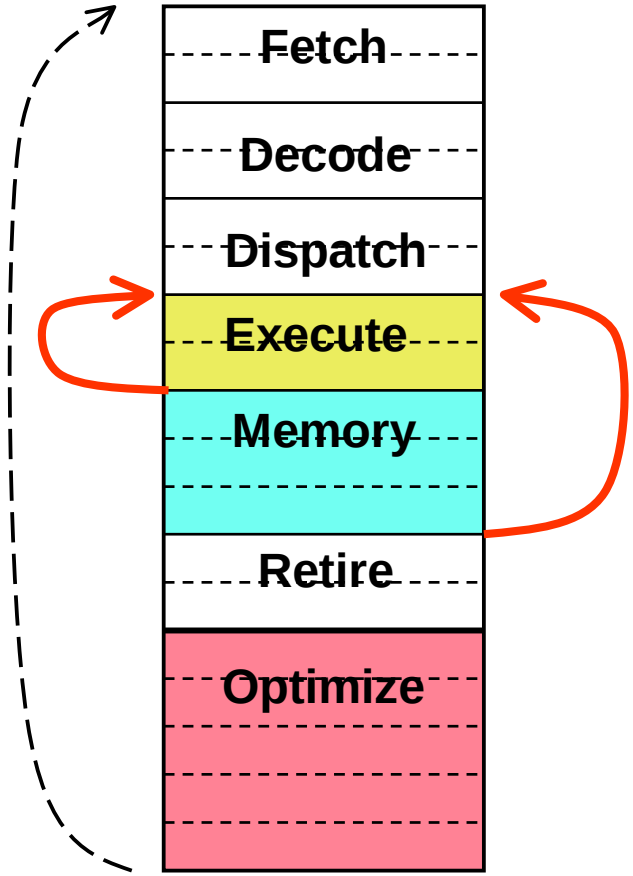


Alleviate Pipe-Depth Penalty

- Front-End Contraction
  - Code Re-mapping and Caching
  - Trace Construction, Caching, Optimization
  - Leverage Back-End Optimizations
- Back-End Optimization
  - Multiple-Branch, Trace, Stream, Prediction
  - Code Reordering, Alignment, Optimization
  - Pre-decode, Pre-rename, Pre-scheduling
  - Memory Pre-fetch Prediction and Control

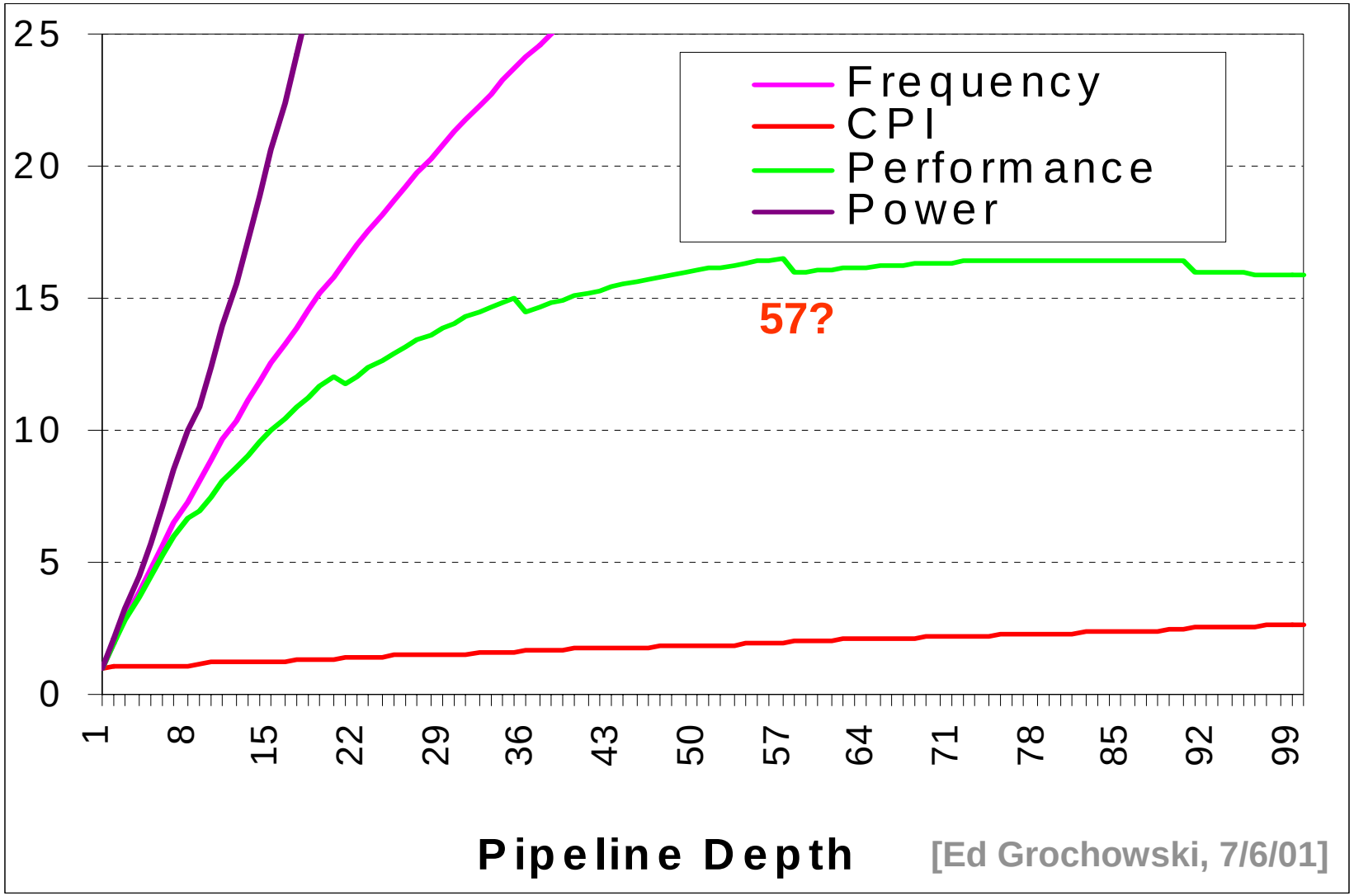
Execution Core Improvement

- Super-pipelined ALU design
- Very high-speed arithmetic units



- Speculative OoO execution
- Criticality-based data caching
- Aggressive data pre-fetching

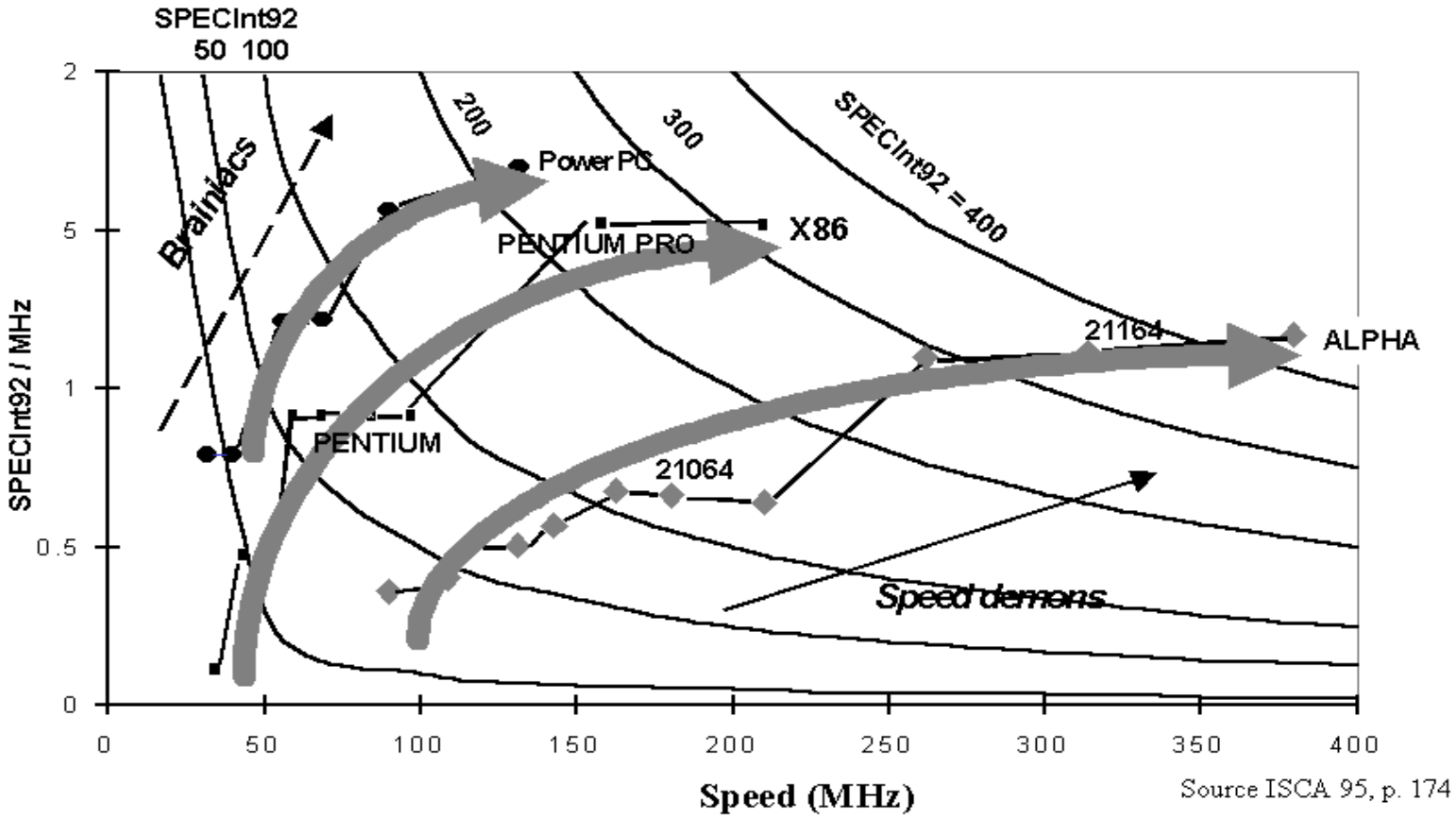
How Deep Can You Go?



How Much ILP Is There?

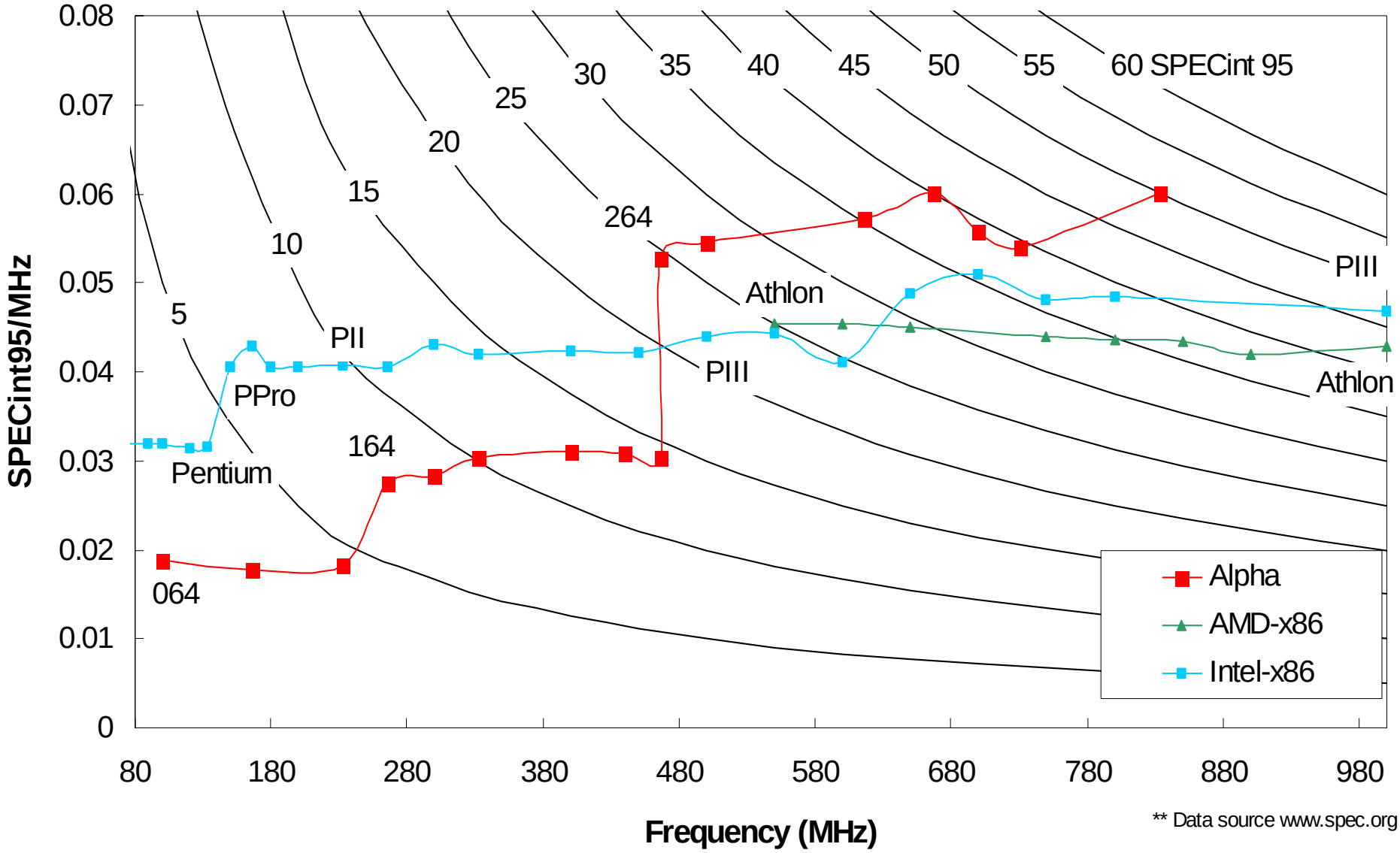
Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7
Kuck et al. [1972]	8
Riseman and Foster [1972]	51
Nicolau and Fisher [1984]	90

Landscape of Microprocessor Families
(SPECint92)



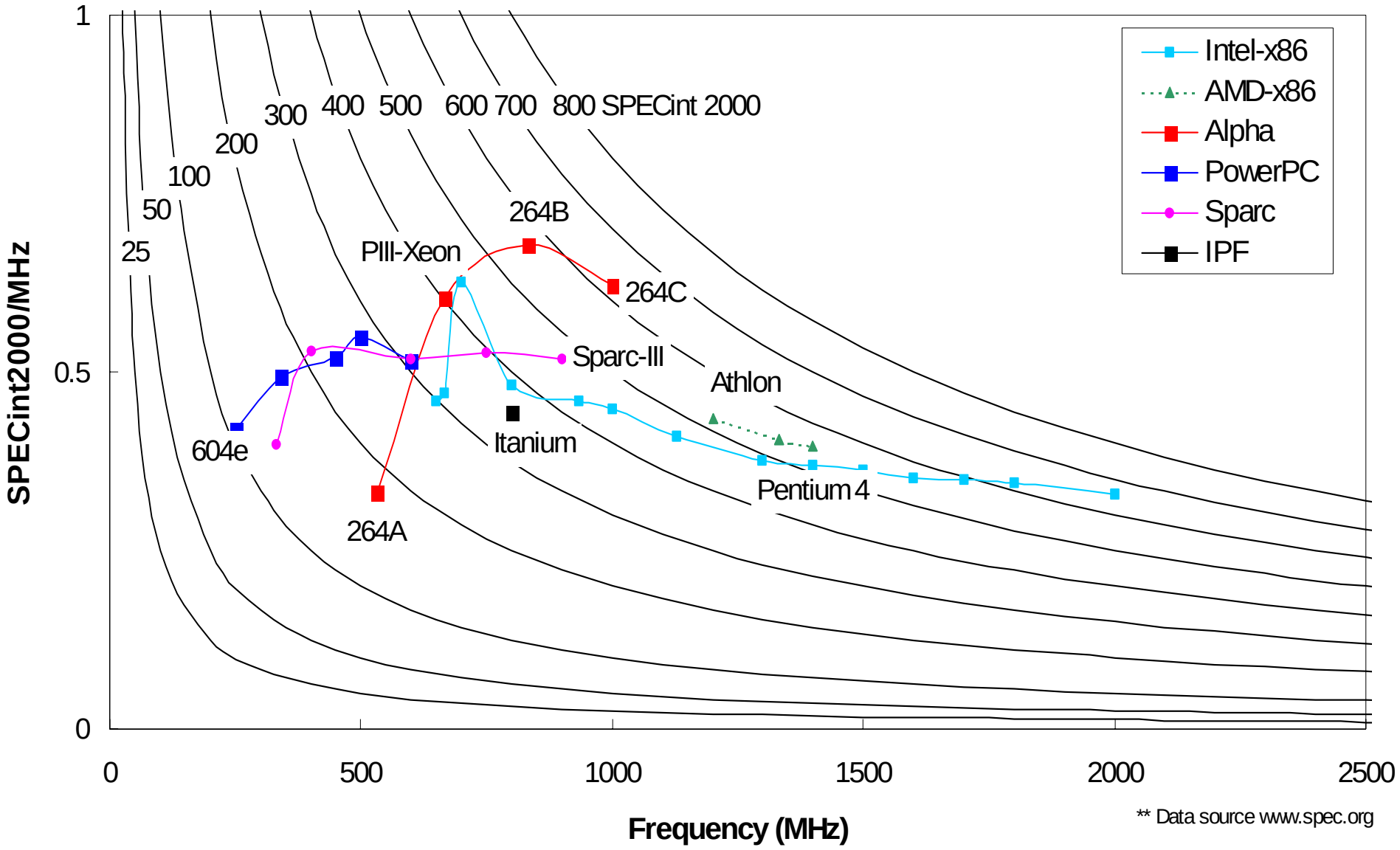
Source ISCA 95, p. 174

Landscape of Microprocessor Families
(SPECint95)



** Data source www.spec.org

Landscape of Microprocessor Families
(SPECint2000)



** Data source www.spec.org

Landscape of Microprocessor Families
(SPECint2010)

The Pentium® 4 Processor