ECE/CS 752: Midterm 1 Review

ECE/CS 752 Fall 2017

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

Lecture notes based on notes by John P. Shen
Updated by Mikko Lipasti

Computer Architecture

- Instruction Set Architecture (IBM 360)
  - *… the attributes of a [computing] system as seen by the programmer. I.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation. -- Amdahl, Blaaw, & Brooks, 1964*
- Machine Organization (microarchitecture)
  - ALUS, Buses, Caches, Memories, etc.
- Machine Implementation (realization)
  - Gates, cells, transistors, wires

Iron Law

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

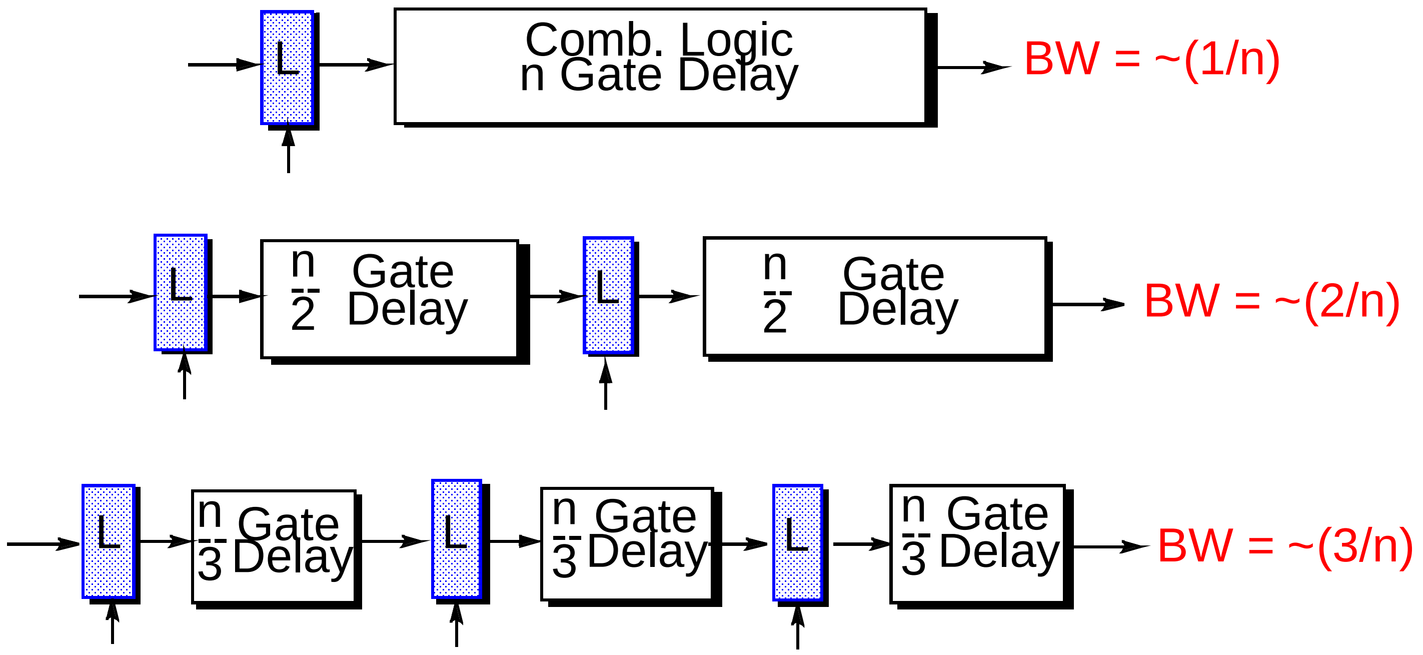
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

Architecture --> Implementation --> Realization

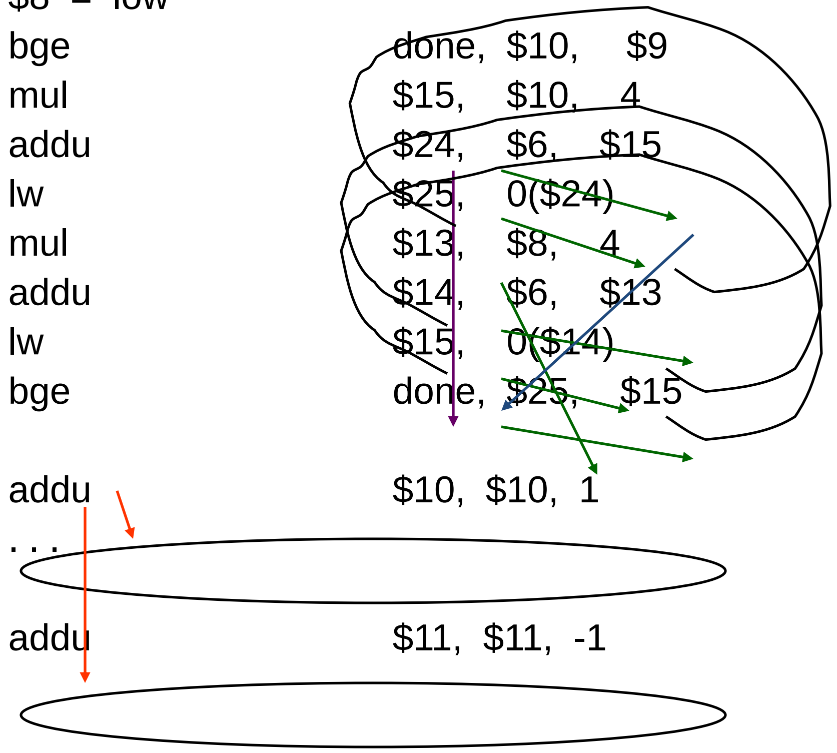
Compiler Designer **Processor Designer** **Chip Designer**

Ideal Pipelining



- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

Example (quicksort/MIPS)



```
#           for (;  (j < high)  &&  (array[j] < array[low])  ;  ++j  );
#           $10  =  j
#           $9  =  high
#           $6  =  array
#           $8  =  low
            bge                 done,  $10,    $9
            mul                 $15,   $10,    4
            addu                $24,   $6,    $15
            lw                  $25,   0($24)
            mul                 $13,   $8,    4
            addu                $14,   $6,    $13
            lw                  $15,   0($14)
            bge                 done,  $25,    $15
cont:
            addu                $10,  $10,  1
            . . .
done:
            addu                $11,  $11,  -1
```

Pipeline Hazards

- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipelining Review

- Pipelining Overview
- Control
  - Data hazards
    - Stalls
    - Forwarding or bypassing
  - Control flow hazards
    - Branch prediction

Technology Challenges

- Technology scaling, Moore vs. Dennard
- Power: dynamic, static
  - CMOS scaling trends
  - Power vs. Energy
  - Dynamic power vs. leakage power
- Usage Models: thermal, efficiency, longevity
- Circuit Techniques
- Architectural Techniques
- Variability
- Packaging

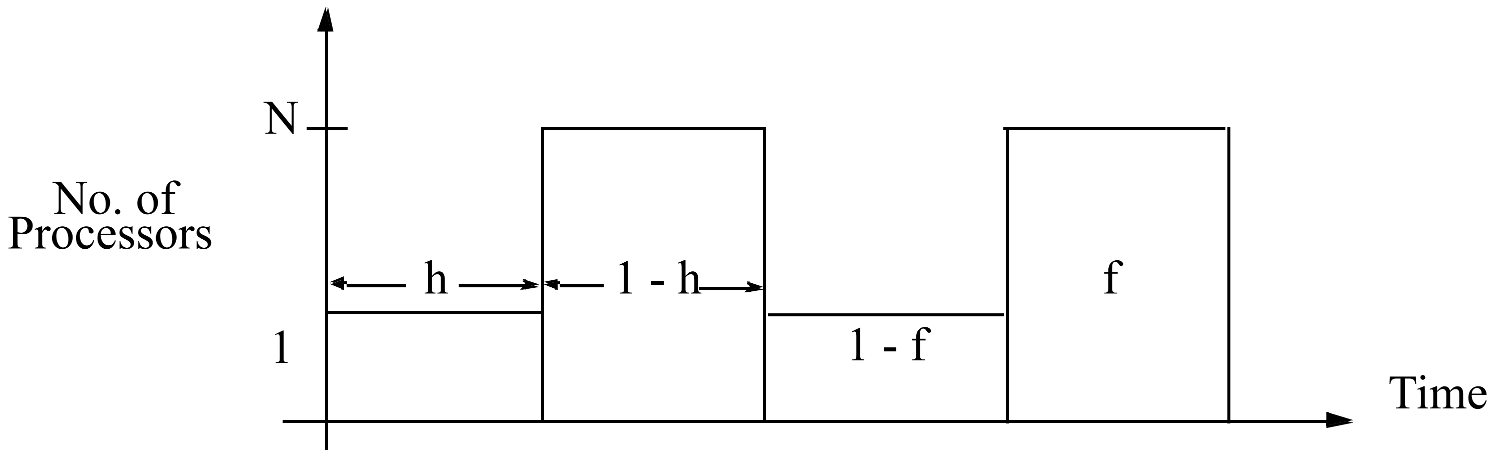
Readings

- Read on your own:
  - Shekhar Borkar, Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation, IEEE Micro 2005, November/December 2005 (Vol. 25, No. 6) pp. 10-16.
  - 2015 ITRS Roadmap -- Executive Summary. Read sections 1, 5, 6, 8, 9, and skim the rest.
- Review by Wed 9/13/2017:
  - Jacobson, H, et al., “Stretching the limits of clock-gating efficiency in server-class processors,” in Proceedings of HPCA-11, 2005.

Pipelining to Superscalar

- Forecast
  - Limits of pipelining
  - The case for superscalar
  - Instruction-level parallel machines
  - Superscalar pipeline organization
  - Superscalar pipeline design

Amdahl's Law



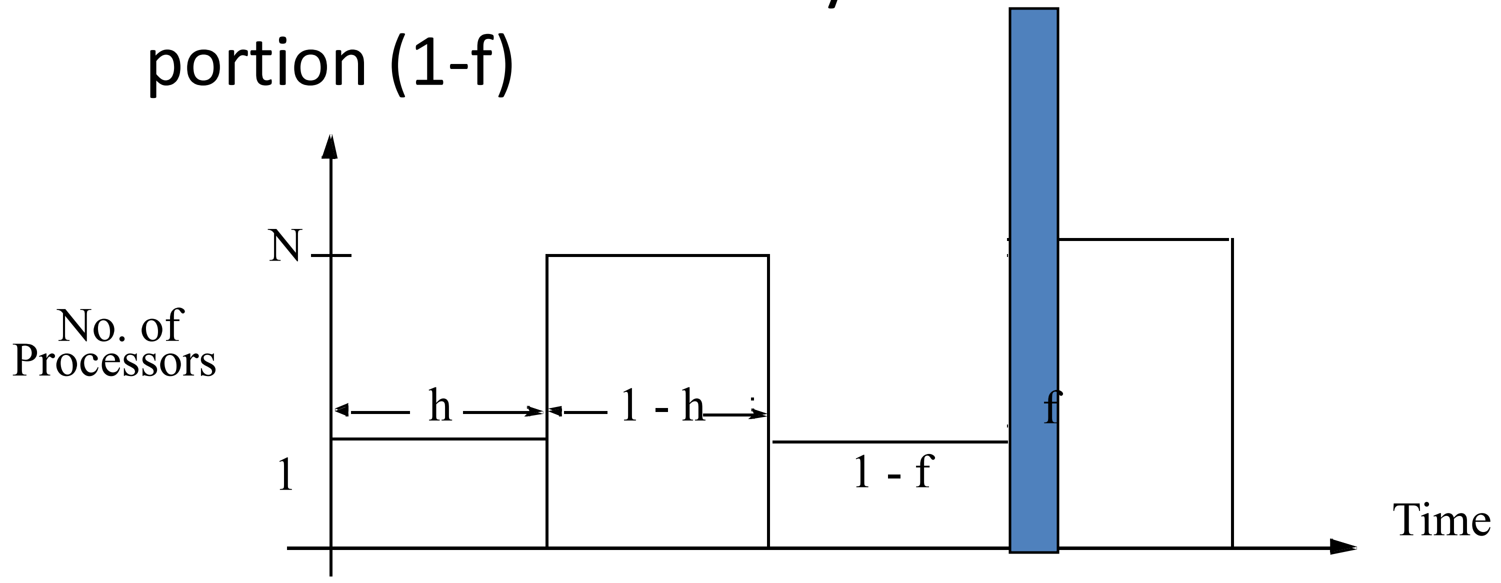
- h = fraction of time in serial code
- f = fraction that is vectorizable
- v = speedup for f
- Overall speedup:

$$Speedup = \frac{1}{1 - f + \frac{f}{v}}$$

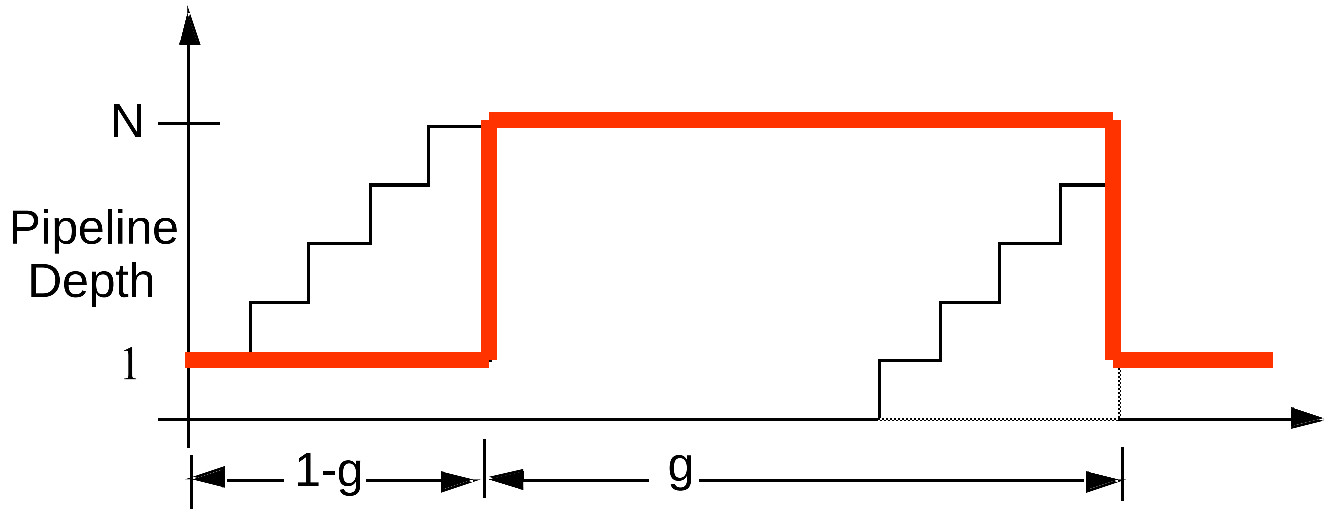
Revisit Amdahl's Law

$$\lim_{v\to\infty} \frac{1}{1-f+\frac{f}{v}} = \frac{1}{1-f}$$

- Sequential bottleneck
- Even if v is infinite
  - Performance limited by nonvectorizable portion (1-f)



Pipelined Performance Model



- Tyranny of Amdahl's Law [Bob Colwell]
  - When g is even slightly below 100%, a big performance hit will result
  - Stalled cycles are the key adversary and must be minimized as much as possible

Superscalar Proposal

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

$$\boxed{Speedup = \frac{1}{\frac{(1-f)}{s} + \frac{f}{v}}}$$

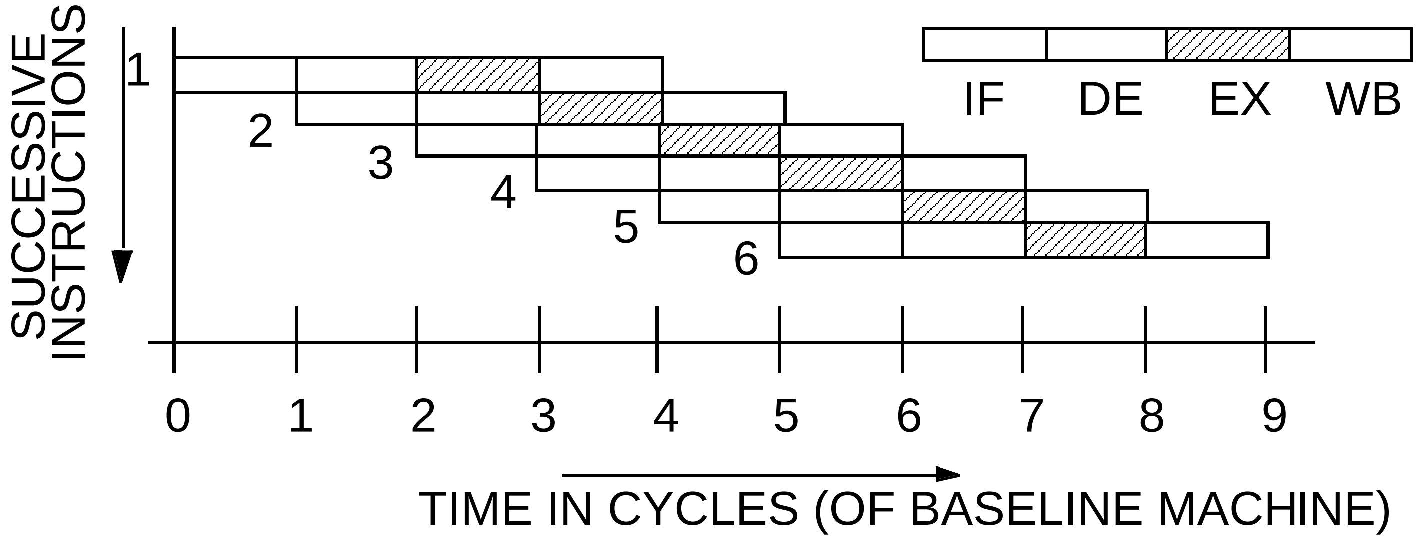
Limits on Instruction Level Parallelism (ILP)

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)

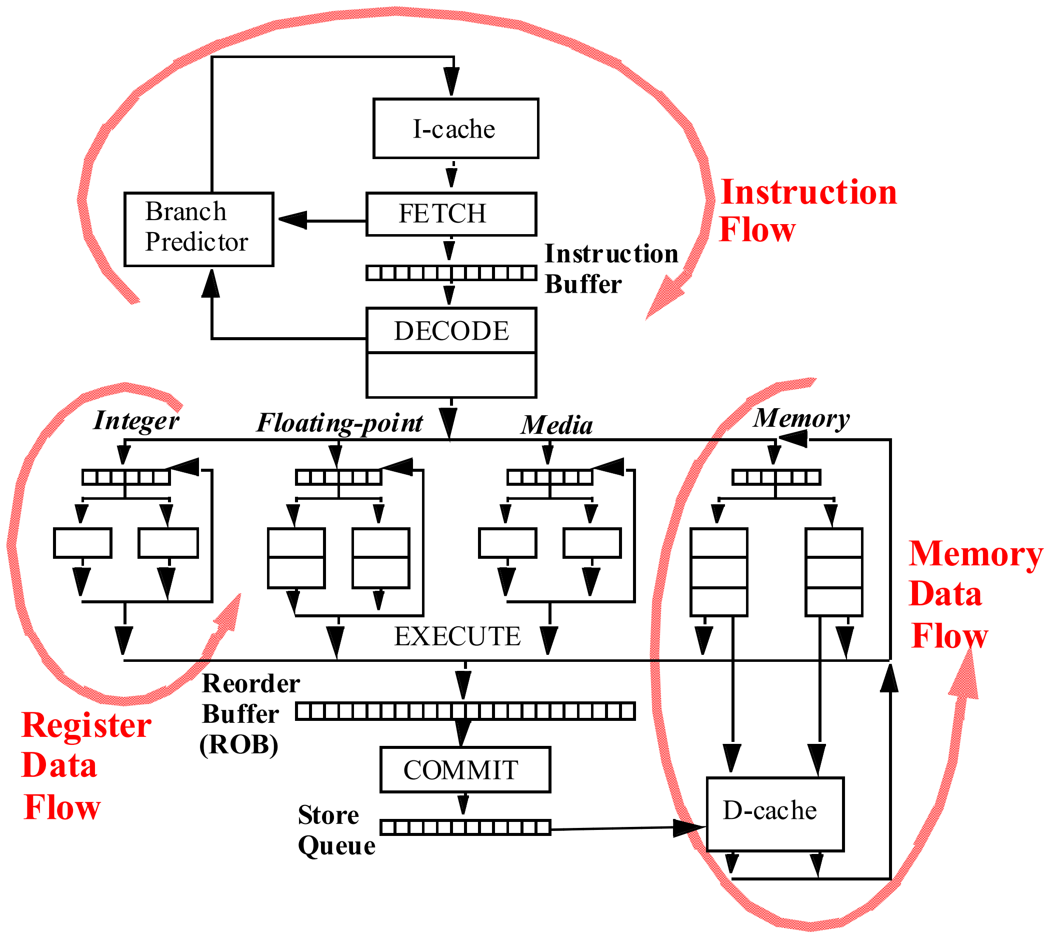
Classifying ILP Machines

[Jouppi, DECWRL 1991]

- Baseline scalar RISC
  - Issue parallelism = IP = 1
  - Operation latency = OP = 1
  - Peak IPC = 1



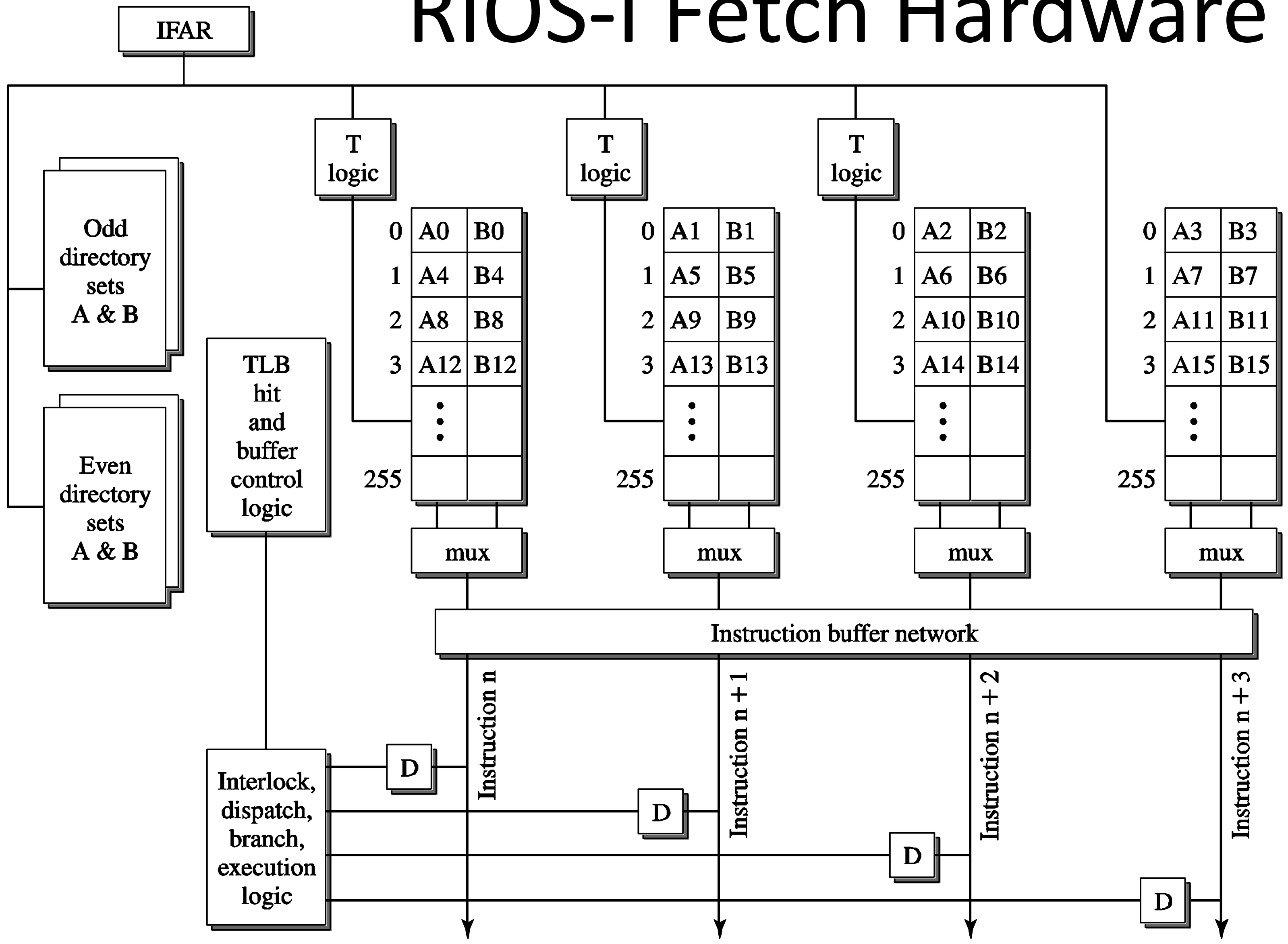
Superscalar Challenges



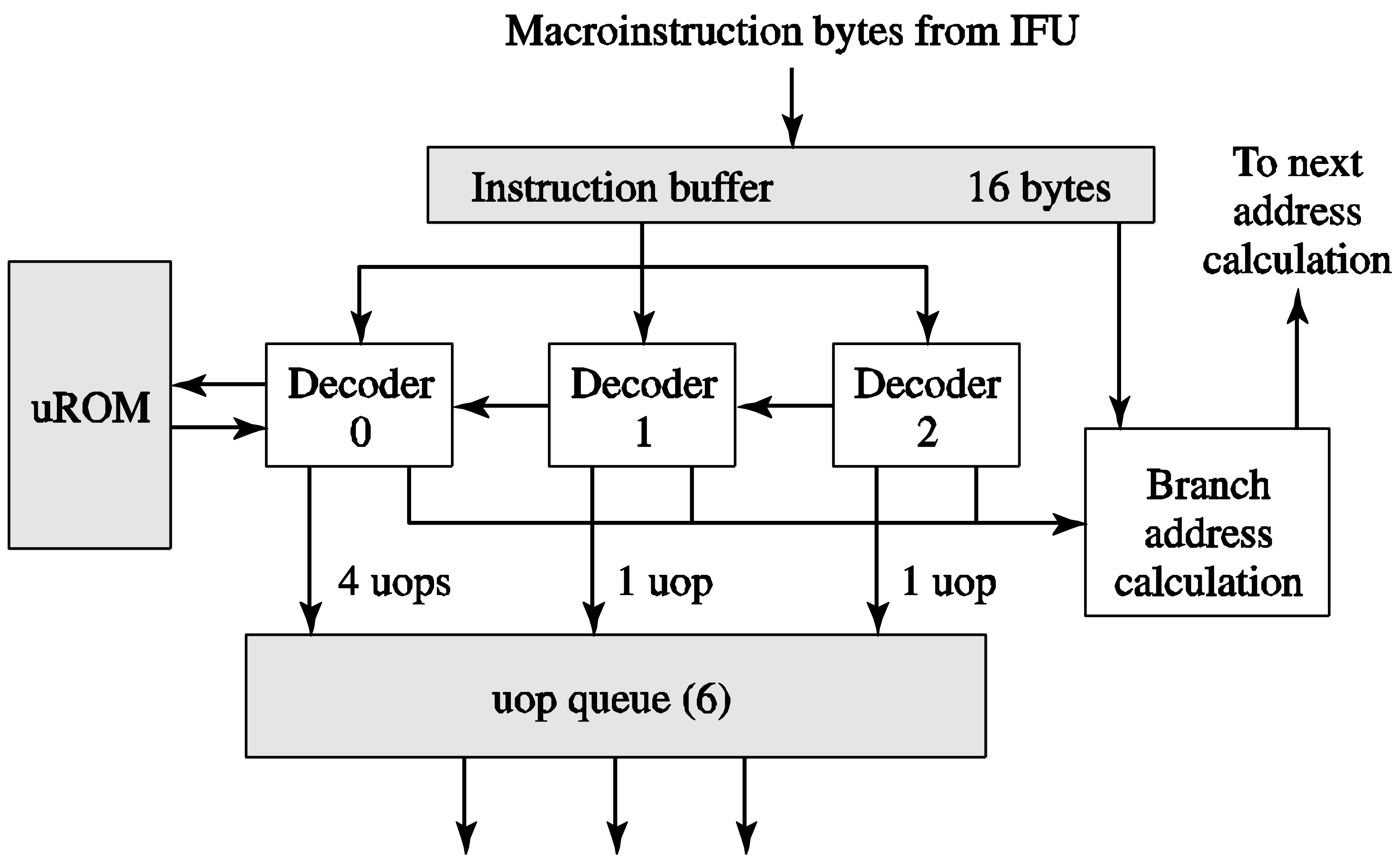
Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
  - Solution: wide (superscalar) pipeline
- Inefficient unified pipeline
  - Long latency for each instruction
  - Solution: diversified, specialized pipelines
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Solution: Out-of-order execution, distributed execution pipelines

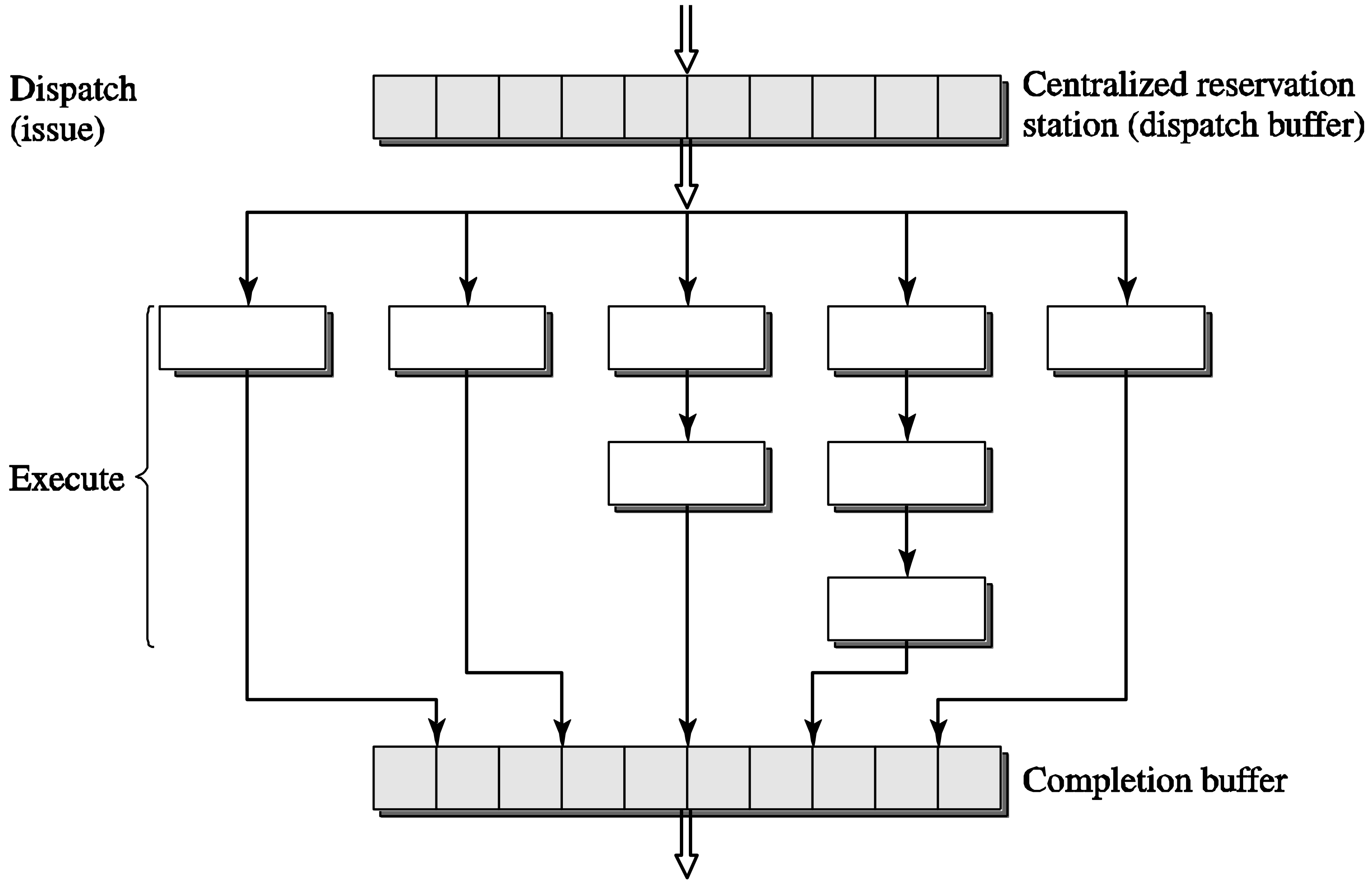
RIOS-I Fetch Hardware



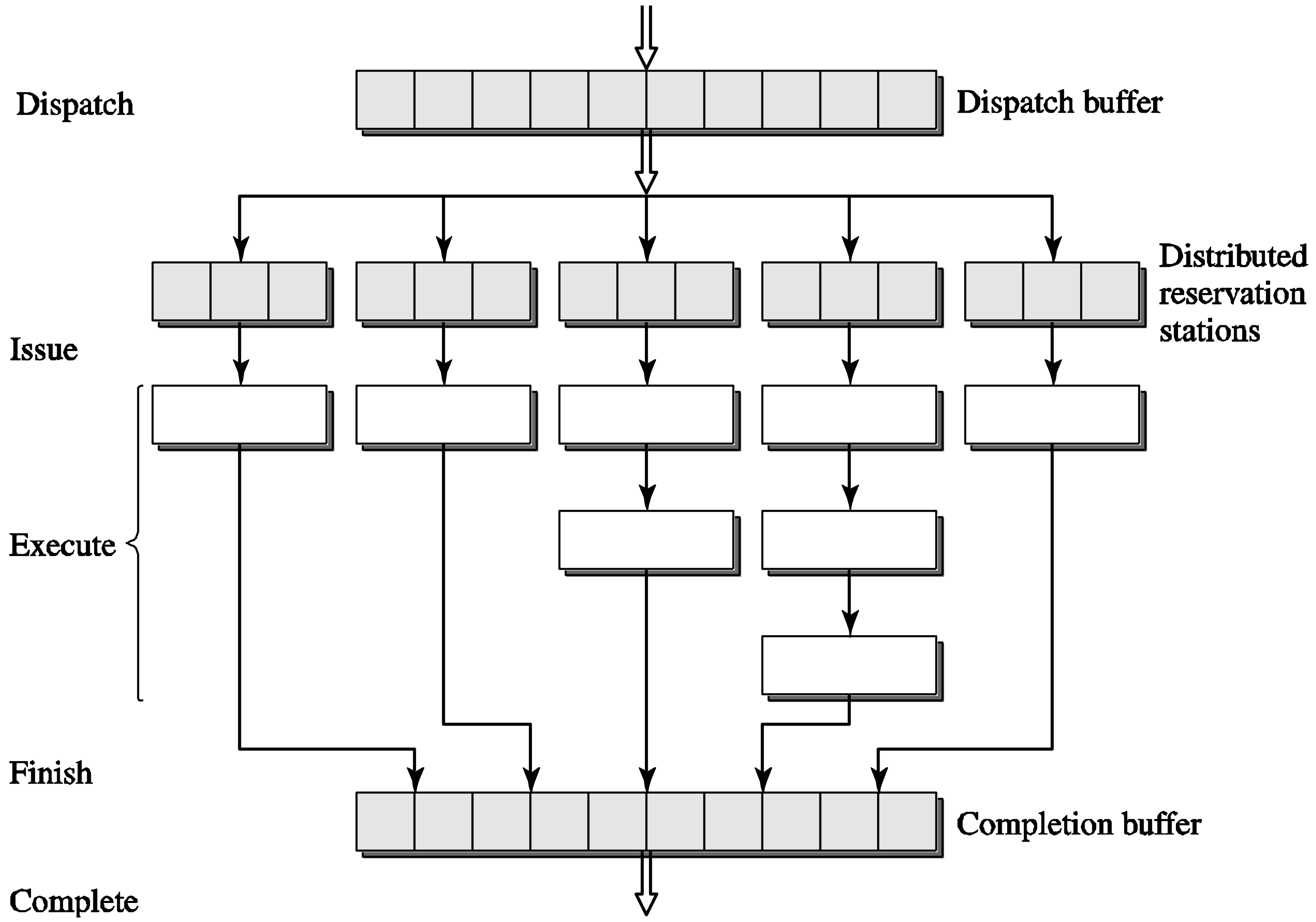
Pentium Pro Fetch/Decode



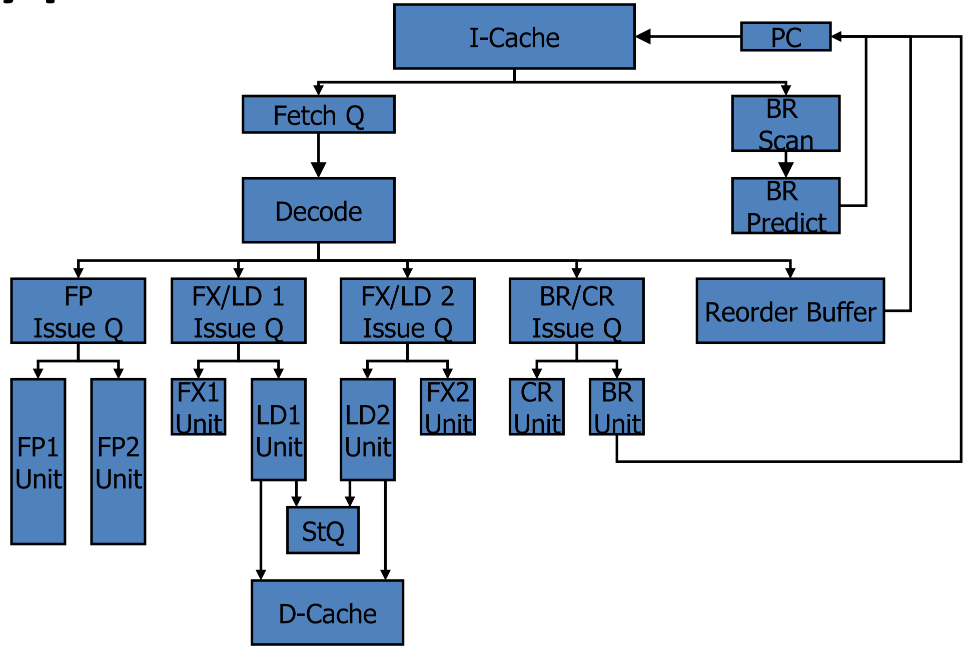
Centralized Reservation Station



Distributed Reservation Station



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)

Issues in Completion/Retirement

- 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)

Superscalar Summary

- Instruction flow
  - Branches, jumps, calls: predict target, direction
  - Fetch alignment
  - Instruction cache misses
- Register data flow
  - Register renaming: RAW/WAR/WAW
- Memory data flow
  - In-order stores: WAR/WAW
  - Store queue: RAW
  - Data cache misses

Instruction Flow Techniques

- Goal of Instruction Flow and Impediments
- Branch Types and Implementations
- What’s So Bad About Branches?
- What are Control Dependences?
- Impact of Control Dependences on Performance
- Improving I-Cache Performance

Goal and Impediments

- Goal of Instruction Flow
  - Supply processor with maximum number of useful instructions every clock cycle
- Impediments
  - Branches and jumps
  - Finite I-Cache
    - Capacity
    - Bandwidth restrictions

Branch Types and Implementation

1. Types of Branches
   A. Conditional or Unconditional
   B. Save PC?
   C. How is target computed?
      - Single target (immediate, PC+immediate)
      - Multiple targets (register)
2. Branch Architectures
   A. Condition code or condition registers
   B. Register

What’s So Bad About Branches?

Problem: Fetch stalls until direction is determined

Solutions:

- Minimize delay
  - Move instructions determining branch condition away from branch (CC architecture)
- Make use of delay
  - Non-speculative:
    - Fill delay slots with useful safe instructions
    - Execute both paths (eager execution)
  - Speculative:
    - Predict branch direction

What’s So Bad About Branches?

Problem: Fetch stalls until branch target is determined

Solutions:

- Minimize delay
  - Generate branch target early
- Make use of delay: Predict branch target
  - Single target
  - Multiple targets

Riseman and Foster's Study

- 7 benchmark programs on CDC-3600
- Assume infinite machines
  - Infinite memory and instruction stack
  - Infinite register file
  - Infinite functional units
  - True dependencies only at dataflow limit
- If bounded to single basic block, speedup is 1.72 (Flynn's bottleneck)
- If one can bypass n branches (hypothetically), then:

Branches Bypassed	0	1	2	8	32	128	∞
Speedup	1.72	2.72	3.62	7.21	14.8	24.4	51.2

Improving I-Cache Performance

- Larger Cache Size
- More associativity
- Larger line size
- Prefetching
  - Next-line
  - Target
  - Markov
- Code layout
- Other types of cache organization
  - Trace cache [Ch. 9]

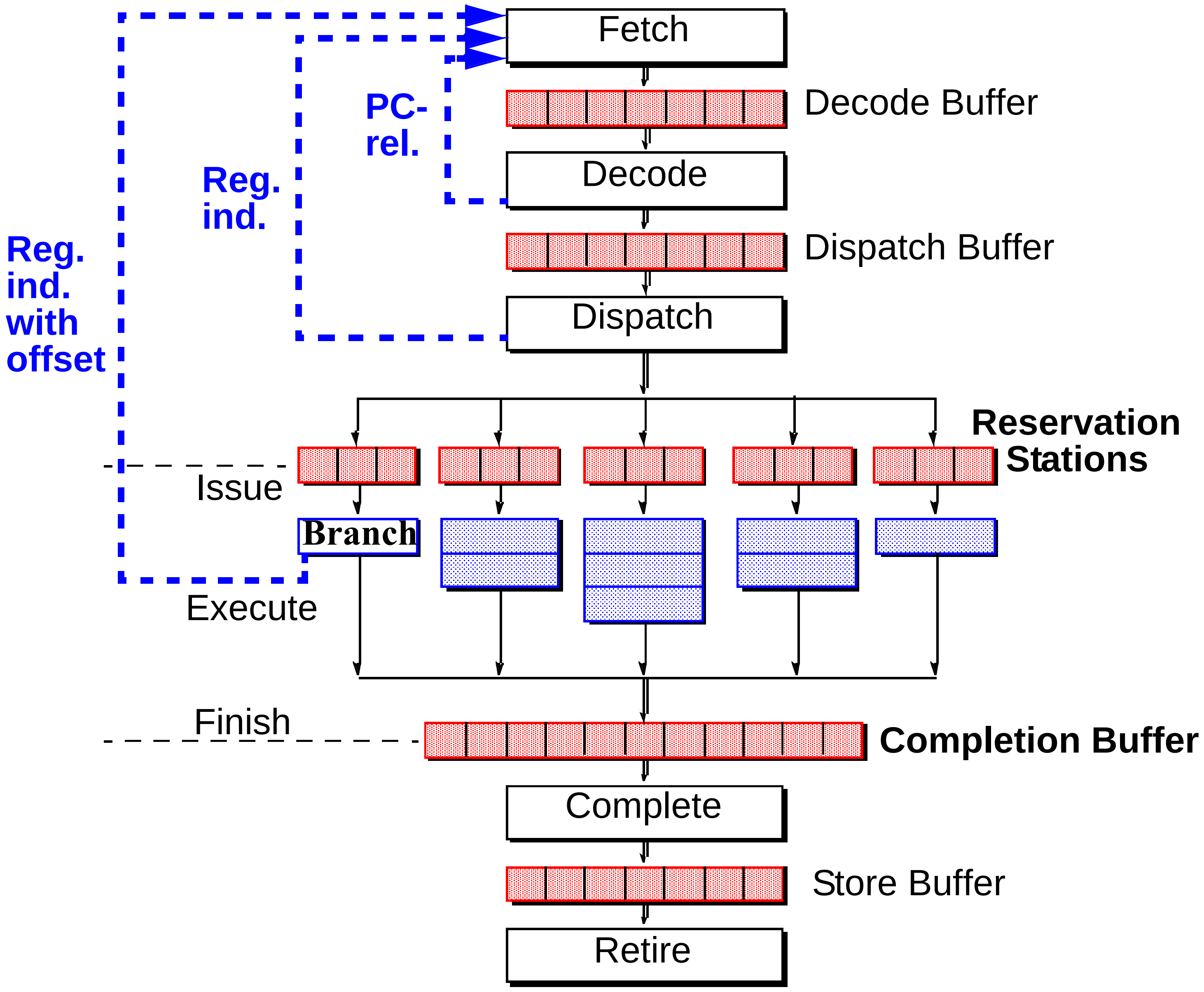
Lecture Overview

- Program control flow
  - Implicit sequential control flow
  - Disruptions of sequential control flow
- Branch Prediction
  - Branch instruction processing
  - Branch instruction speculation
- Key historical studies on branch prediction
  - UCB Study [Lee and Smith, 1984]
  - IBM Study [Nair, 1992]
- Branch prediction implementation (PPC 604)
  - BTAC and BHT design
  - Fetch Address Generation

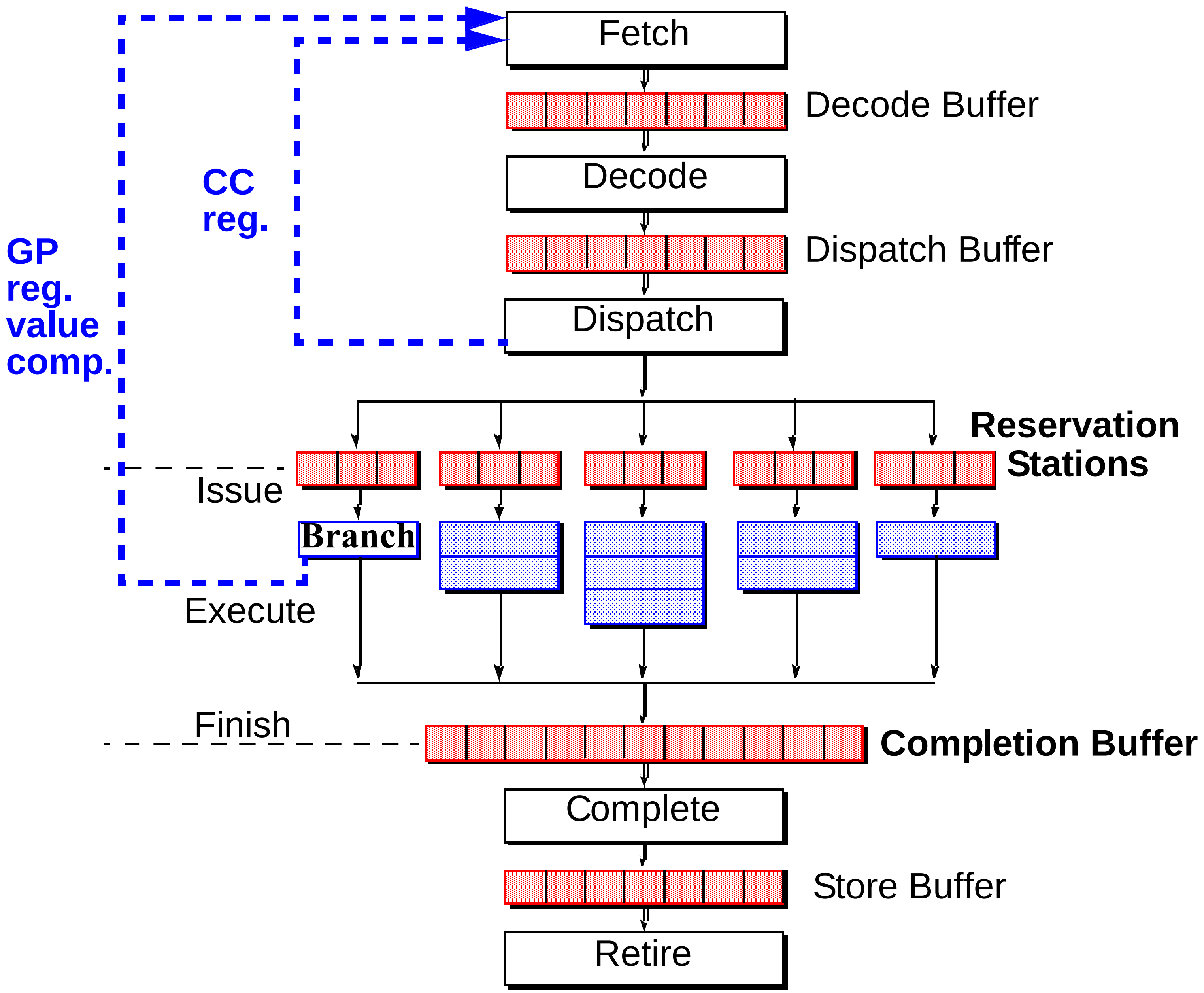
Branch Prediction

- Target address generation $\rightarrow$ <u>Target Speculation</u>
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement, autodecrement
- Condition resolution $\rightarrow$ <u>Condition speculation</u>
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)

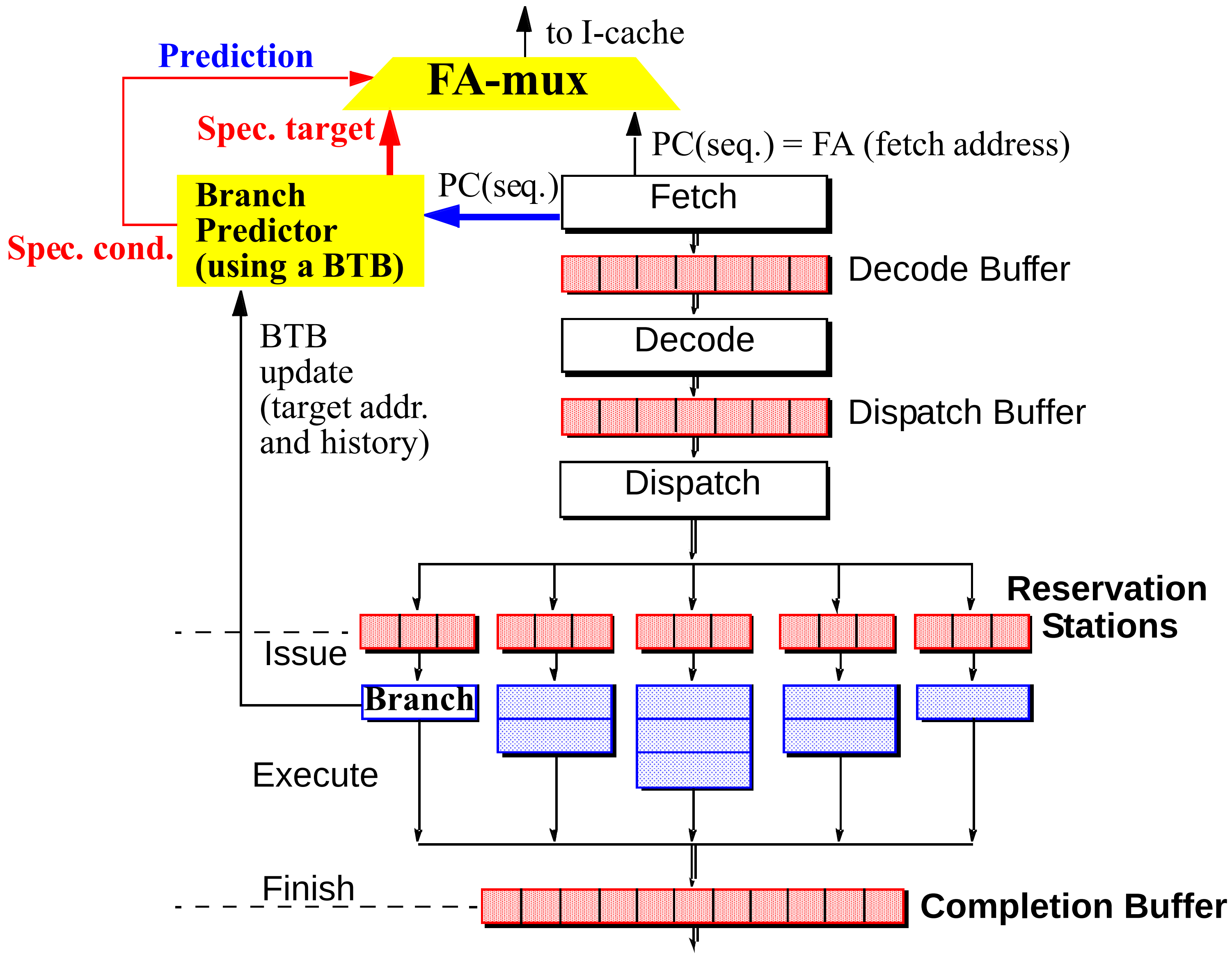
Target Address Generation



Condition Resolution



Branch Instruction Speculation



Branch/Jump Target Prediction

Branch inst. address	Information for predict.	**Branch target address** (most recent)

- Branch Target Buffer: small cache in fetch stage
  - Previously executed branches, address, taken history, target(s)
- Fetch stage compares current FA against BTB
  - If match, use prediction
  - If predict taken, use BTB target
- When branch executes, BTB is updated
- Optimization:
  - Size of BTB: increases hit rate
  - Prediction algorithm: increase accuracy of prediction

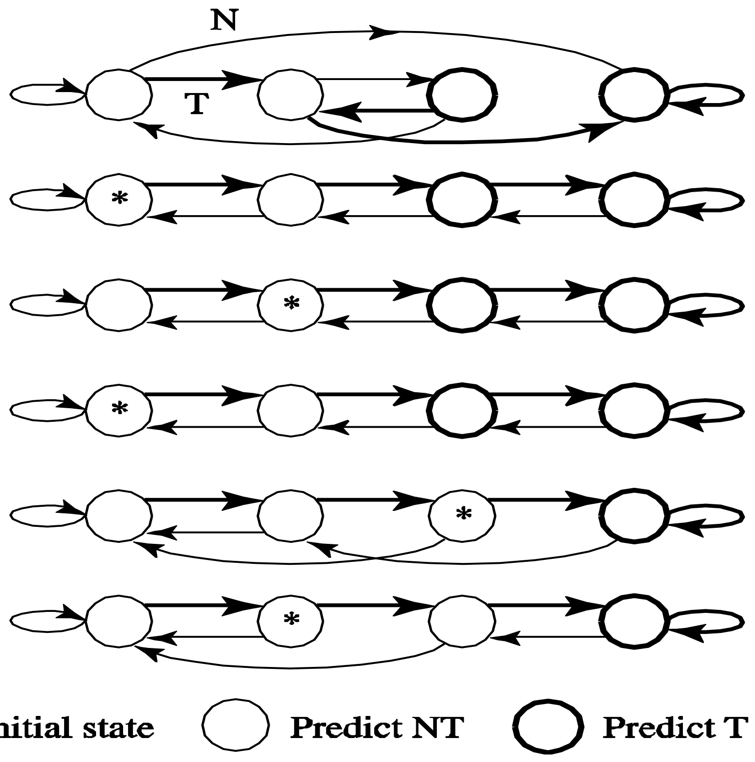
Branch Prediction: Condition Speculation

1. Biased Not Taken
   - Hardware prediction
   - Does not affect ISA
   - Not effective for loops
2. Software Prediction
   - Extra bit in each branch instruction
     - Set to 0 for not taken
     - Set to 1 for taken
   - Bit set by compiler or user; can use profiling
   - Static prediction, same behavior every time
3. Prediction based on branch offset
   - Positive offset: predict not taken
   - Negative offset: predict taken
4. Prediction based on dynamic history

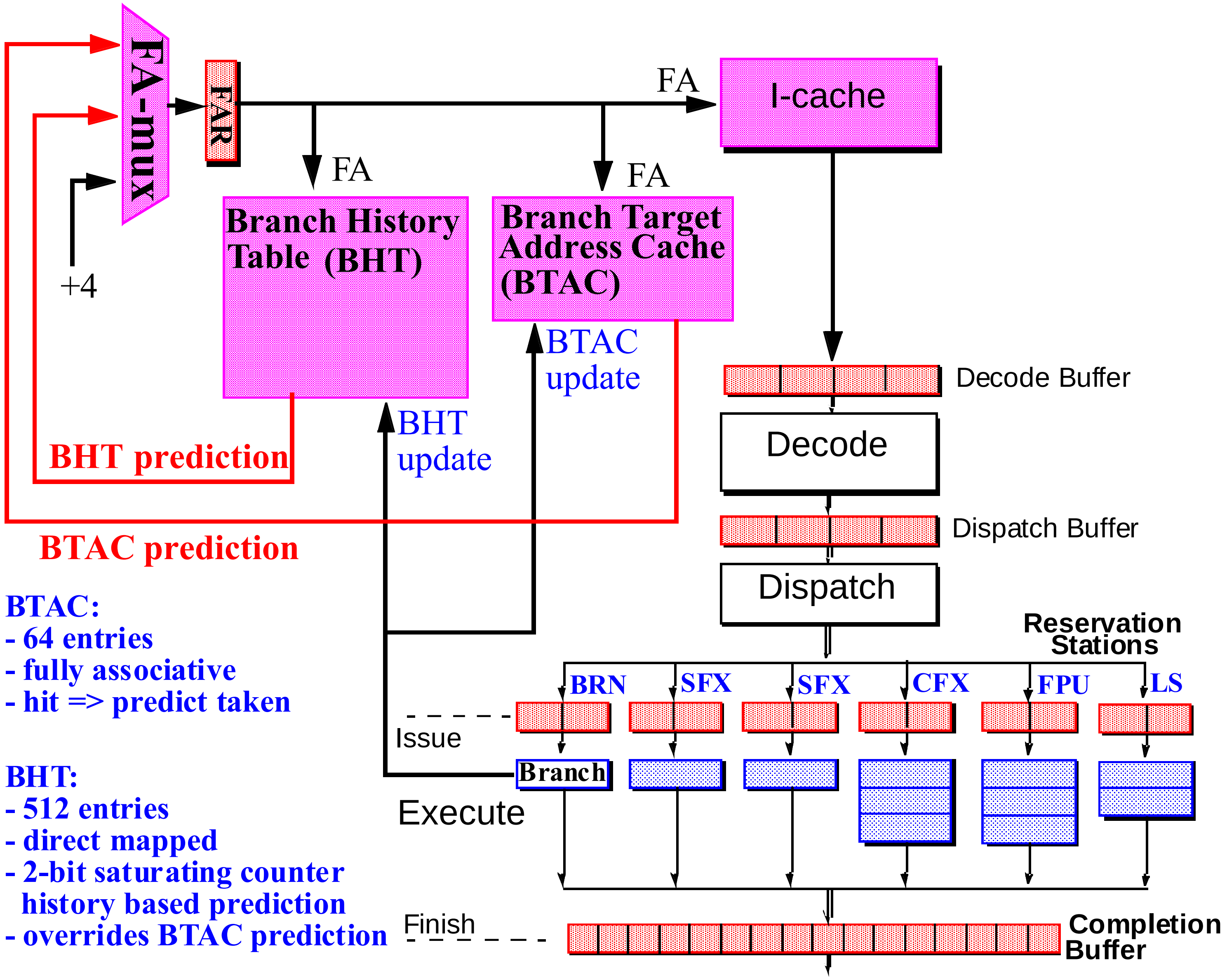
Exhaustive Search for Optimal 2-bit Predictor

- There are 2^{20} possible state machines of 2-bit predictors
- Some machines are uninteresting, pruning them out reduces the number of state machines to 5248
- For each benchmark, determine prediction accuracy for all the predictor state machines
- Find optimal 2-bit predictor for each application

Benchmark	Optimal
spice2g6	97.2
doduc	94.3
gcc	89.1
espresso	89.1
li	87.1
eqntott	87.9



BTAC and BHT Design (PPC 604)



BTAC:
- 64 entries
- fully associative
- hit => predict taken

BHT:
- 512 entries
- direct mapped
- 2-bit saturating counter history based prediction
- overrides BTAC prediction

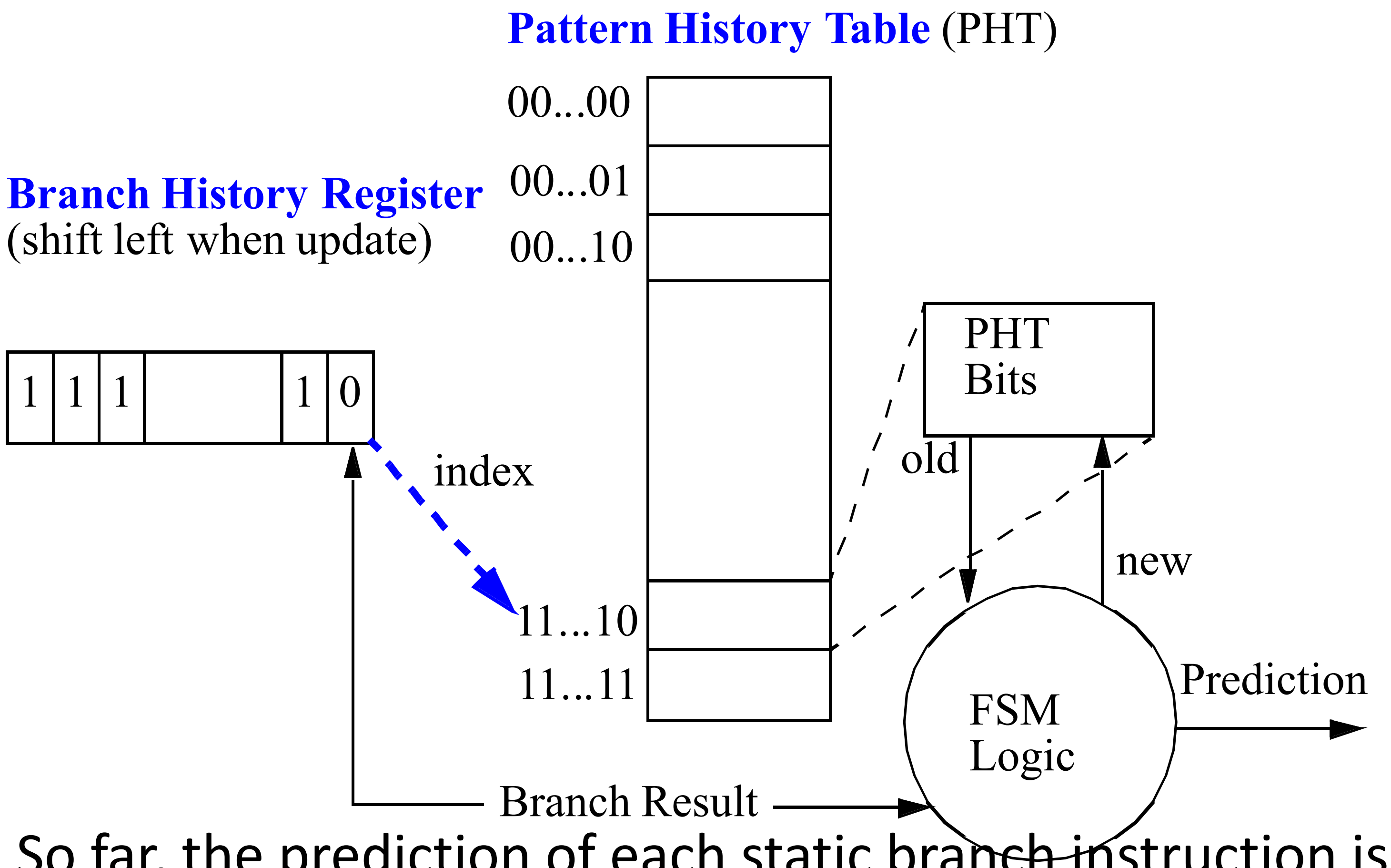
Advanced Branch Prediction

- Control Flow Speculation
  - Branch Speculation
  - Mis-speculation Recovery
- Two-Level Adaptive Branch Prediction
- Global BHSR Scheme (GAs)
- Per-Branch BHSR Scheme (PAs)
- Gshare Branch Predictor
- Combining branch predictor
- Understanding Advanced Predictors
- Perceptron branch predictor

Branch Speculation

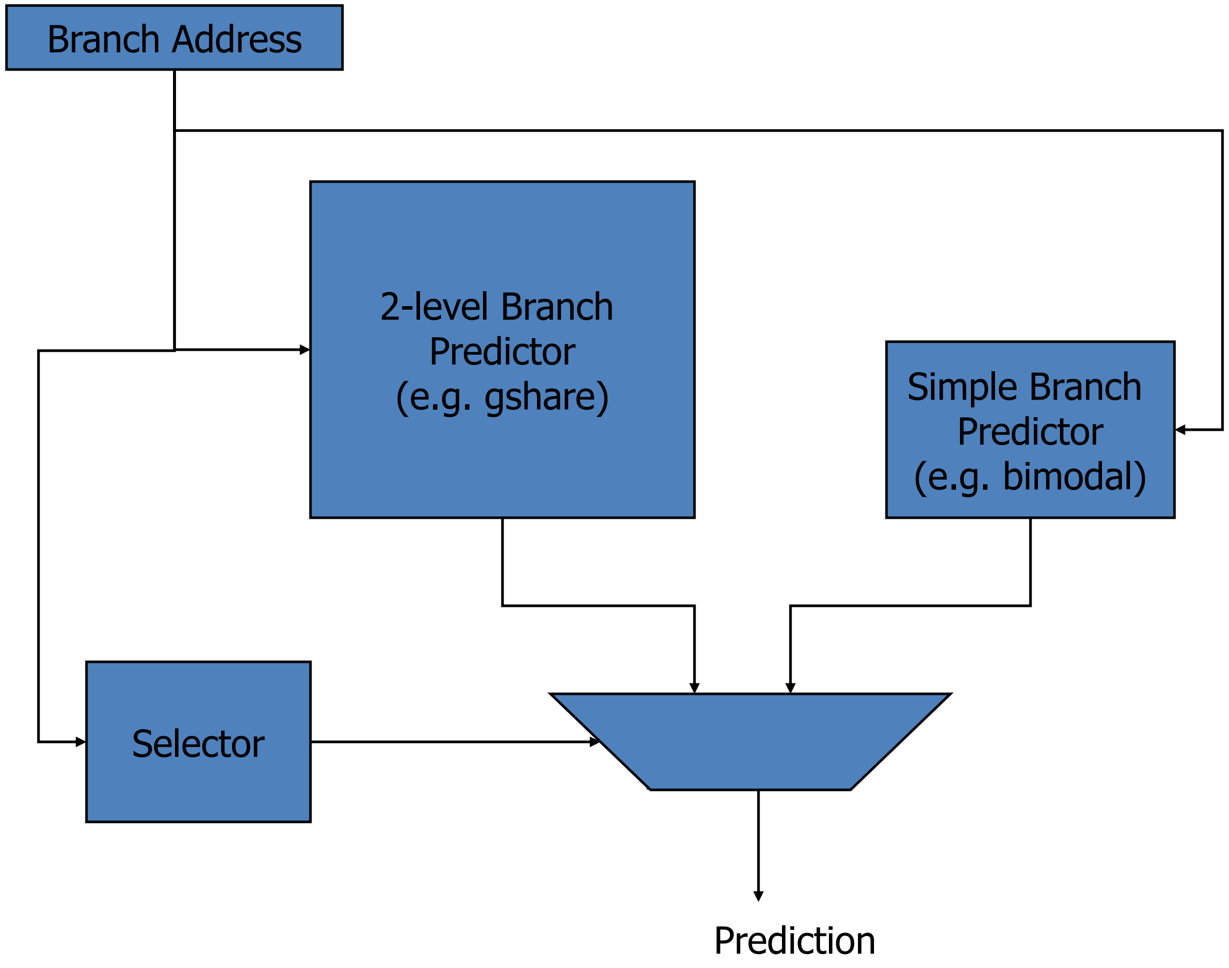
- <u>Leading Speculation</u>
  1. Tag speculative instructions
  2. Advance branch and following instructions
  3. Buffer addresses of speculated branch instructions
- <u>Trailing Confirmation</u>
  1. When branch resolves, remove/deallocate speculation tag
  2. Permit completion of branch and following instructions

Two-Level Adaptive Branch Prediction



- So far, the prediction of each static branch instruction is based solely on its own past behavior and independent of the behaviors of other neighboring static branch instructions (except for inadvertent aliasing).

Combining Branch Predictor


Branch Address
2-level Branch Predictor (e.g. gshare)
Simple Branch Predictor (e.g. bimodal)
Selector
Prediction

Understanding Advanced Predictors

- Four types of history
  - Local (bimodal) history (Smith predictor)
    - Table of counters summarizes local history
    - Simple, but only effective for biased branches
  - Local outcome history
    - Shift register of individual branch outcomes
    - Separate counter for each outcome history
  - Global outcome history
    - Shift register of recent branch outcomes
    - Separate counter for each outcome history
  - Path history
    - Shift register of recent (partial) block addresses
    - Can differentiate similar global outcome histories
- Can combine or “alloy” histories in many ways

Understanding Advanced Predictors

- History length
  - Short history—lower training cost
  - Long history—captures macro-level behavior
  - Variable history length predictors
- Really long history (long loops)
  - Fourier transform into frequency domain
- Limited capacity & interference
  - Constructive vs. destructive
  - Bi-mode, gskewed, agree, YAGS
  - Read sec. 9.3.2 carefully

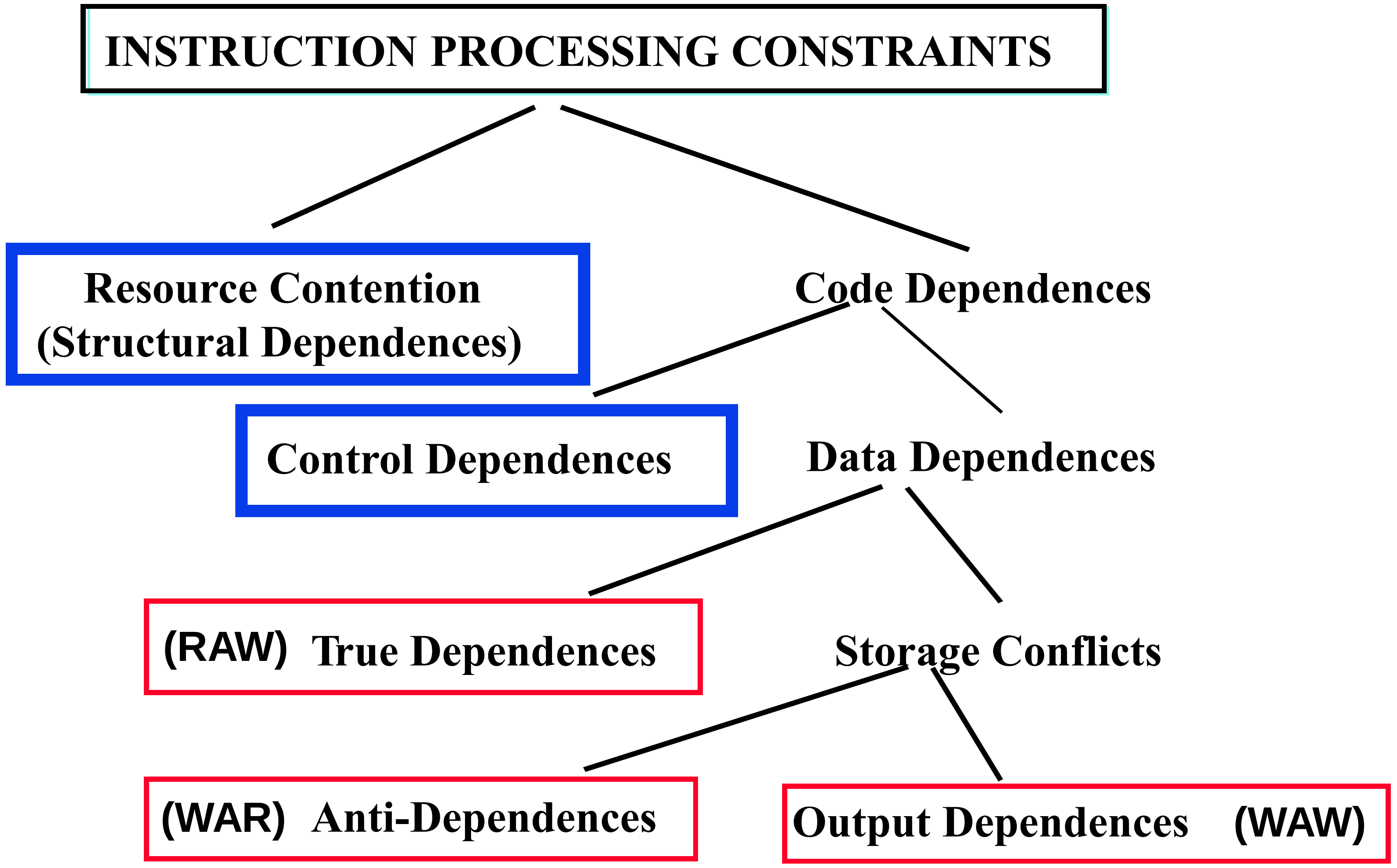
Summary

- Control Flow Speculation
  - Branch Speculation
  - Mis-speculation Recovery
- Two-Level Adaptive Branch Prediction
- Global BHSR Scheme (GAs)
- Per-Branch BHSR Scheme (PAs)
- Gshare Branch Predictor
- Combining branch predictor
- Understanding advanced predictors
  - Study Chapter 9 !!!
- Perceptron branch predictor

Register Data Flow Techniques

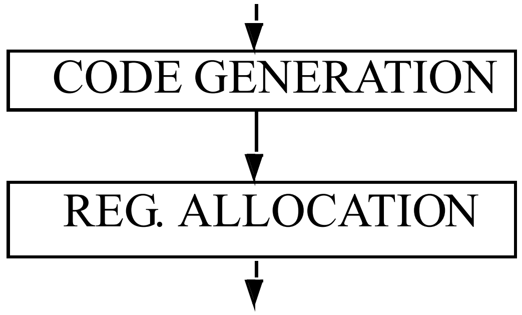
- Register Data Flow
  - Resolving Anti-dependences
  - Resolving Output Dependences
  - Resolving True Data Dependences
- Tomasulo’s Algorithm [Tomasulo, 1967]
  - Modified IBM 360/91 Floating-point Unit
  - Reservation Stations
  - Common Data Bus
  - Register Tags
  - Operation of Dependency Mechanisms

The Big Picture



Contribution to Register Recycling

COMPILER REGISTER ALLOCATION



Single Assignment, Symbolic Reg.

Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

INSTRUCTION LOOPS

```
 9  $34:  mul   $14   $7,   40
10        addu  $15,  $4,   $14
11        mul   $24,  $9,   4
12        addu  $25,  $15,  $24
13        lw    $11,  0($25)
14        mul   $12,  $9,   40
15        addu  $13,  $5,   $12
16        mul   $14,  $8,   4
17        addu  $15,  $13,  $14
18        lw    $24,  0($15)
19        mul   $25,  $11,  $24
20        addu  $10,  $10,  $25
21        addu  $9,   $9,   1
22        ble   $9,   10,   $34
```

```
For (k=1;k<= 10; k++)
t += a [i] [k] * b [k] [j] ;
```

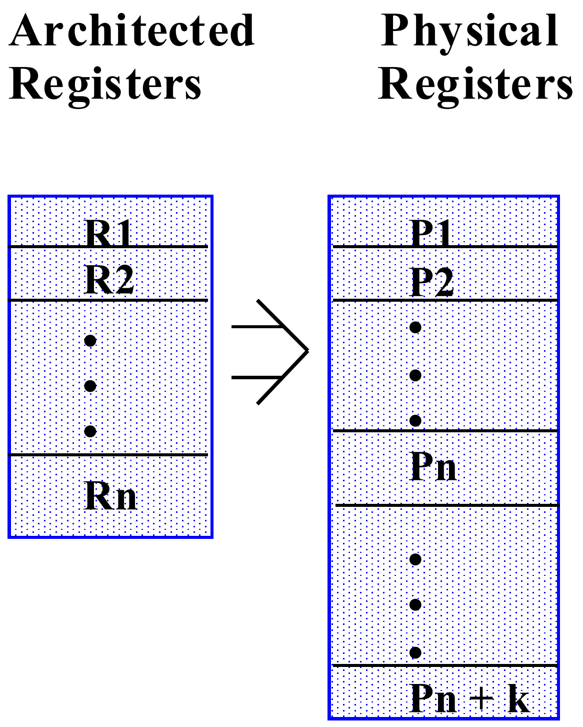
Reuse Same Set of Reg. in Each Iteration

Overlapped Execution of Different Iterations

Register Renaming

Register Renaming Resolves:

Anti-Dependences
Output Dependences



<u>Design of Redundant Registers</u>

Number:

- **One**
- **Multiple**

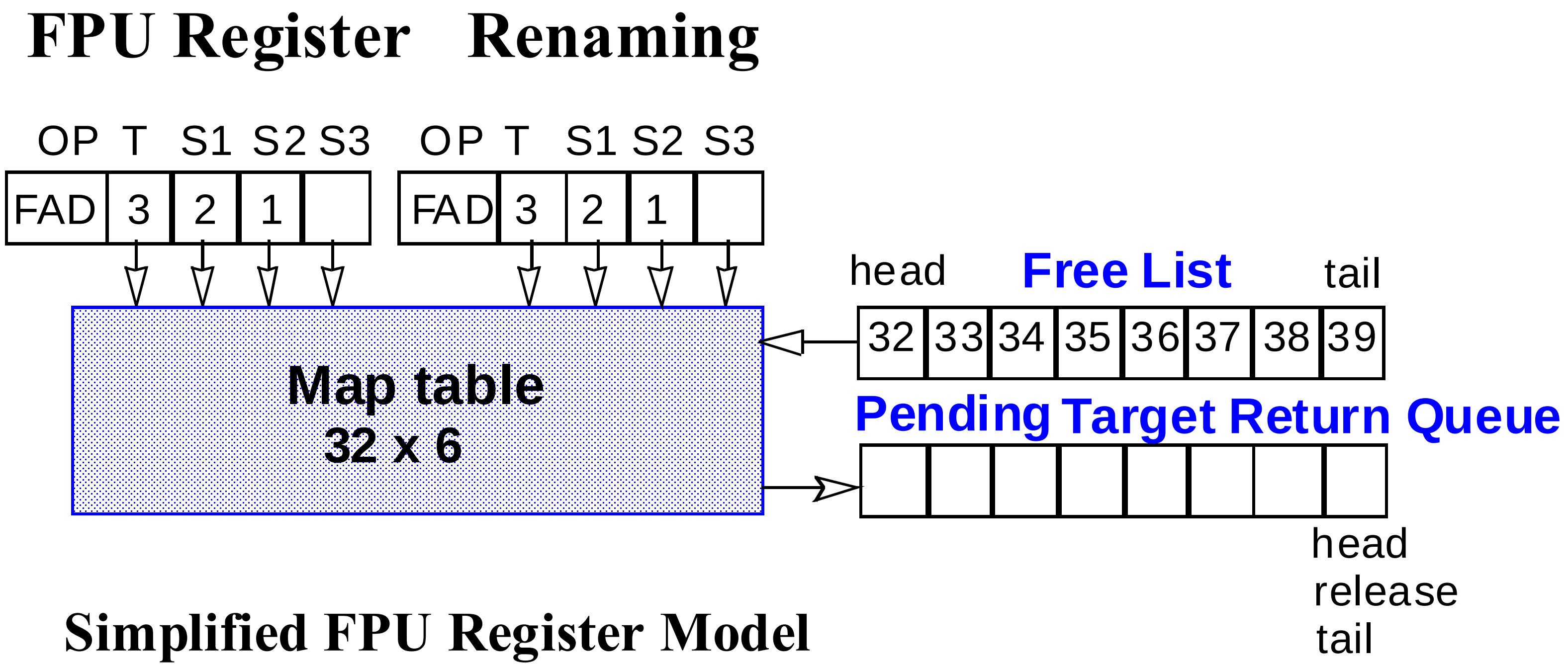
Allocation:

- **Fixed for Each Register**
- **Pooled for all Regsiters**

Location:

- **Attached to Register File (Centralized)**
- **Attached to functional units (Distributed)**

Register Renaming in the RIOS-I FPU

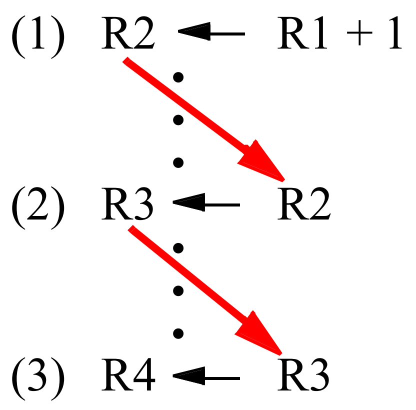


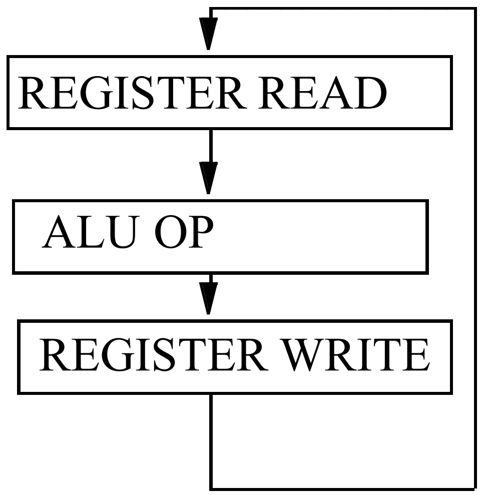
Incoming FPU instructions pass through a renaming table prior to decode

The 32 architectural registers are remapped to 40 physical registers

- Physical register names are used within the FPU
- Complex control logic maintains active register mapping

Resolving True Data Dependences





STALL DISPATCHING

ADVANCE INSTRUCTIONS

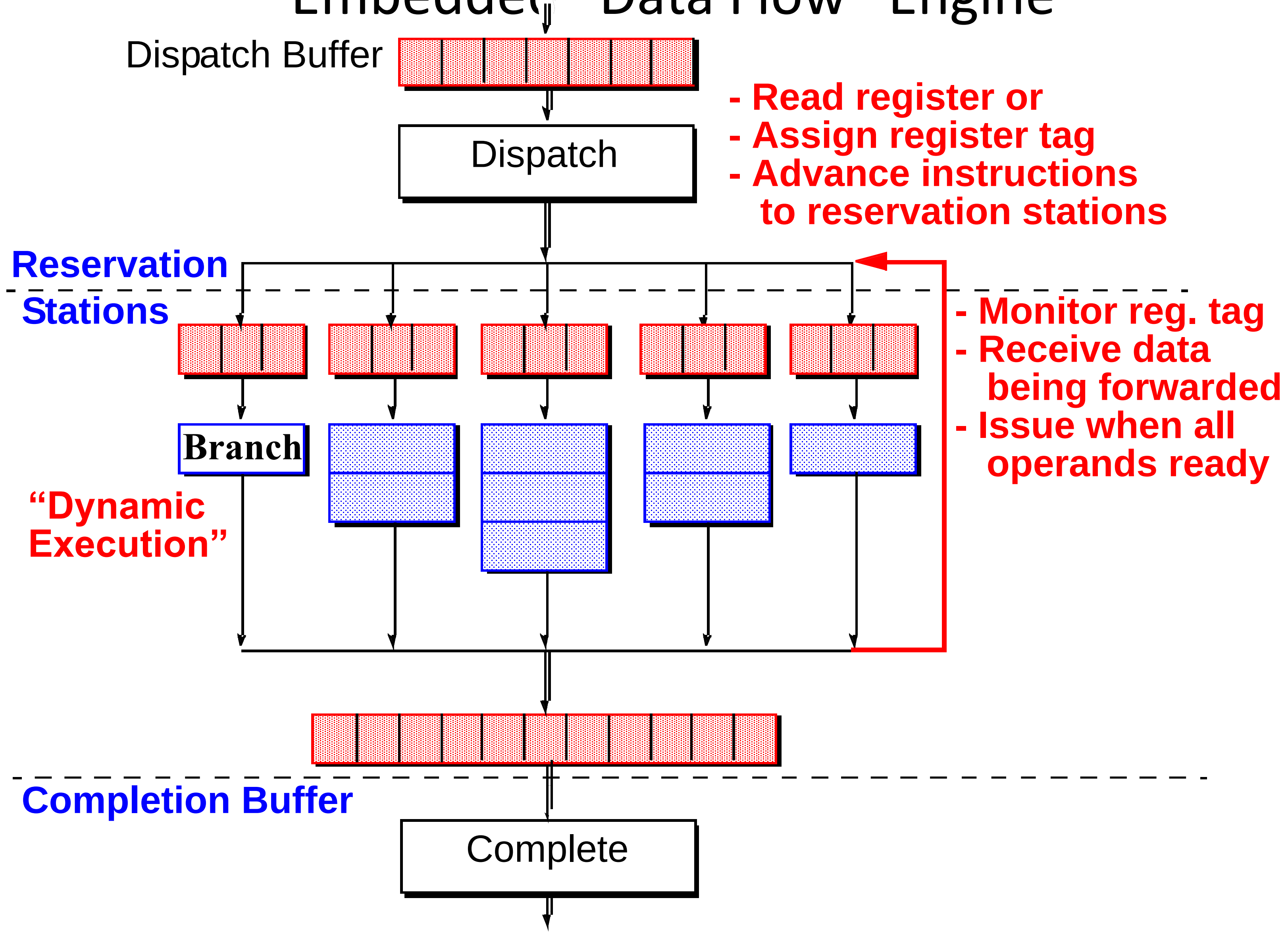
“DYNAMIC EXECUTION”

Reservation Station + Complex Forwarding

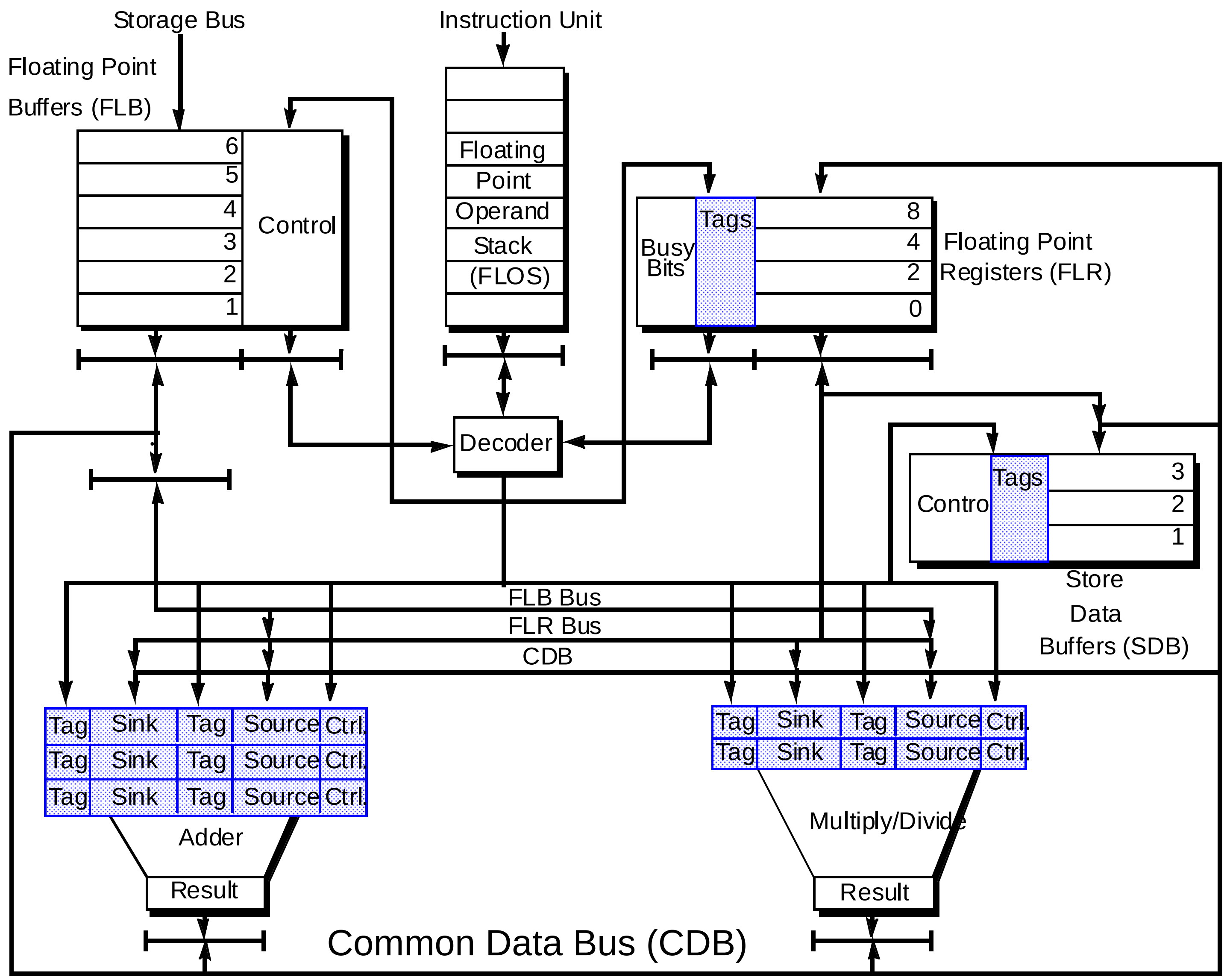
Out-of-order (OoO) Execution

Try to Approach the “Data-Flow Limit”

Embedded “Data Flow” Engine



IBM 360/91 FPU



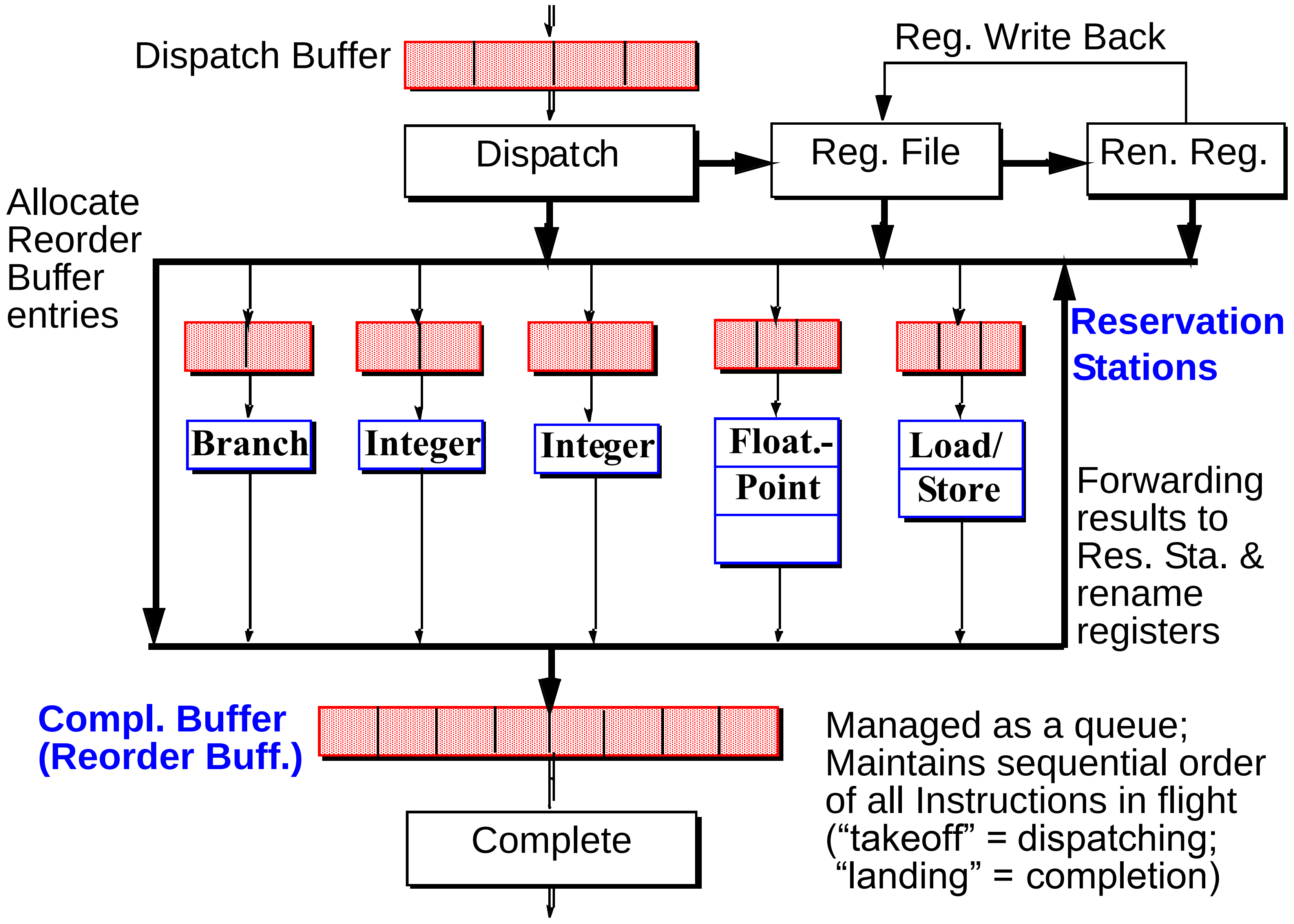
Summary of Tomasulo's Algorithm

- **Supports out of order execution of instructions.**
- **Resolves dependences dynamically using hardware.**
- **Attempts to delay the resolution of dependencies as late as possible.**
- **Structural dependence does not stall issuing; virtual FU's in the form of reservation stations are used.**
- **Output dependence does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag.**
- **True dependence with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station.**
- **Anti-dependence does not stall write back; earlier copying of operand awaiting read to the reservation station.**
- **Can support sequence of multiple output dependences.**
- **Forwarding from FU's to reservation stations bypasses the register file.**

Tomasulo vs. Modern OOO

	IBM 360/91	Modern
Width	Peak IPC = 1	4+
Structural hazards	2 FPU Single CDB	Many FU Many busses
Anti-dependences	Operand copy	Reg. Renaming
Output dependences	Renamed reg. tag	Reg. renaming
True dependences	Tag-based forw.	Tag-based forw.
Exceptions	Imprecise	Precise (ROB)
Implementation	3 x 66” x 15” x 78” 60ns cycle time 11-12 gate delays per pipe stage >$1 million	1 chip 300ps < $100

"Dataflow Engine" for Dynamic Execution

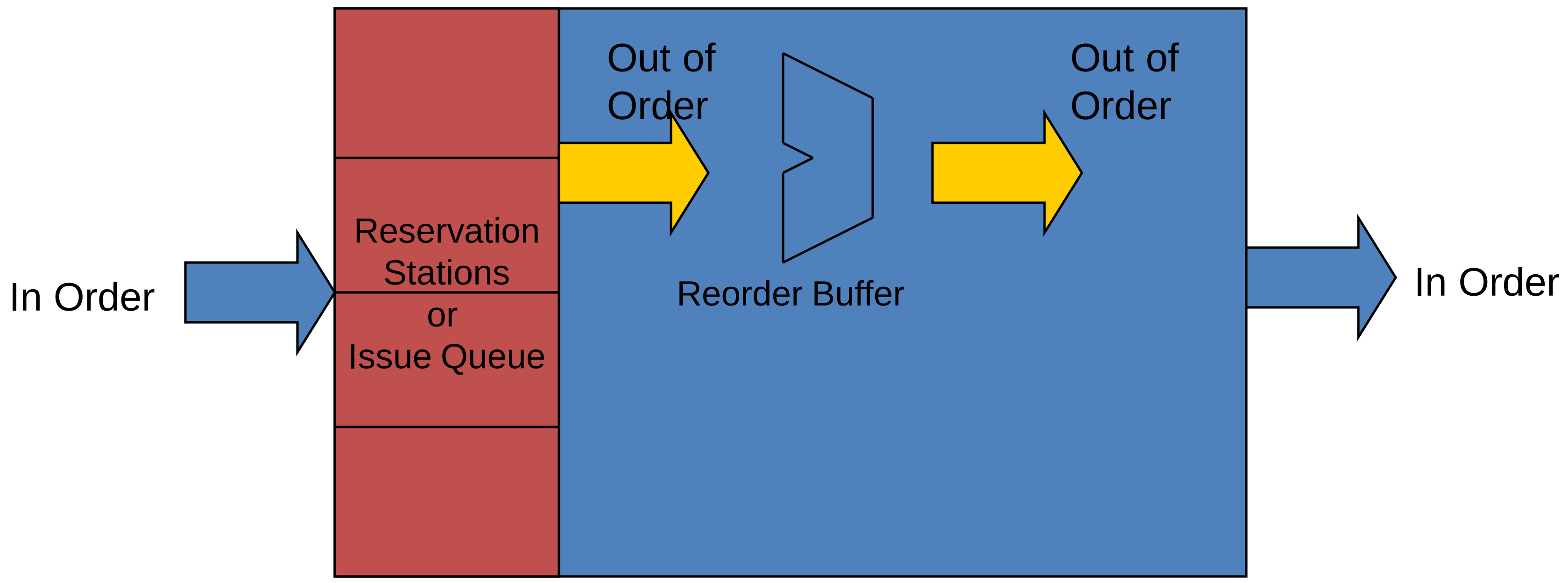


Managed as a queue;
Maintains sequential order of all Instructions in flight
("takeoff" = dispatching;
"landing" = completion)

Instruction Processing Steps

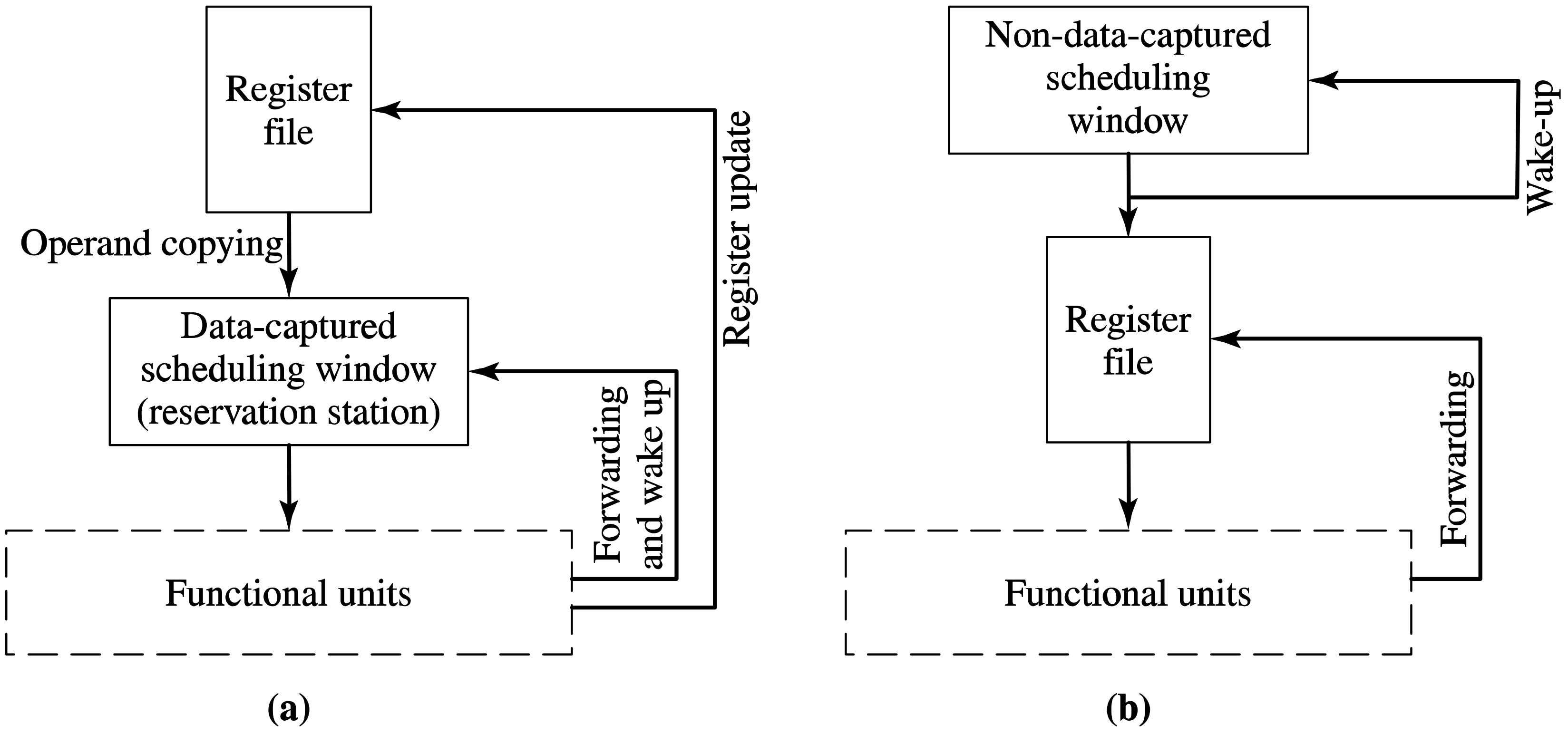
- **DISPATCH:**
  - **Read operands from Register File (RF) and/or Rename Buffers (RRB)**
  - **Rename destination register and allocate RRB entry**
  - **Allocate Reorder Buffer (ROB) entry**
  - **Advance instruction to appropriate Reservation Station (RS)**
- **EXECUTE:**
  - **RS entry monitors bus for register Tag(s) to latch in pending operand(s)**
  - **When all operands ready, issue instruction into Functional Unit (FU) and deallocate RS entry (no further stalling in execution pipe)**
  - **When execution finishes, broadcast result to waiting RS entries, RRB entry, and ROB entry**
- **COMPLETE:**
  - **Update architected register from RRB entry, deallocate RRB entry, and if it is a store instruction, advance it to Store Buffer**
  - **Deallocate ROB entry and instruction is considered architecturally completed**

Reservation Station Implementation



- **Reservation Stations: distributed vs. centralized**
  - **Wakeup: benefit to partition across data types**
  - **Select: much easier with partitioned scheme**
    - **Select 1 of n/4 vs. 4 of n**

Data Capture Reservation Station



- **Reservation Stations**
  - **Data capture vs. no data capture**
  - **Latter leads to “speculative scheduling”**

Register File Alternatives

Register Lifetime	Status	Duration (cycles)	Result stored where?		
			Future File	History File	Phys. RF
Dispatch	Unavail	≥ 1	N/A	N/A	N/A
Finish execution	Speculative	≥ 0	FF	ARF	PRF
Commit	Committed	≥ 0	ARF	ARF	PRF
Next def. Dispatched	Committed	≥ 1	ARF	HF	PRF
Next def. Committed	Discarded	≥ 0	Overwritten	Discarded	Reclaimed

- Rename register organization
  - Future file (future updates buffered, later committed)
    - Rename register file
  - History file (old versions buffered, later discarded)
  - **Merged (single physical register file)**

Rename Table Implementation

- MAP checkpointing
  - Recovery from branches, exceptions
  - Checkpoint granularity
    - Every instruction
    - Every branch, playback to get to exception boundary
- RAM Map
  - Just a lookup table; checkpoints nxm each
- CAM Map
  - Positional bit vectors; checkpoints a single column

Summary

- Register dependences
  - True dependences
  - Antidependences
  - Output dependences
- Register Renaming
- Tomasulo’s Algorithm
- Reservation Station Implementation
- Reorder Buffer Implementation
- Register File Implementation
  - History file
  - Future file
  - Physical register file
- Rename Table Implementation

Memory Data Flow

- **Memory Data Flow**
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- **Basic Memory Hierarchy Review**

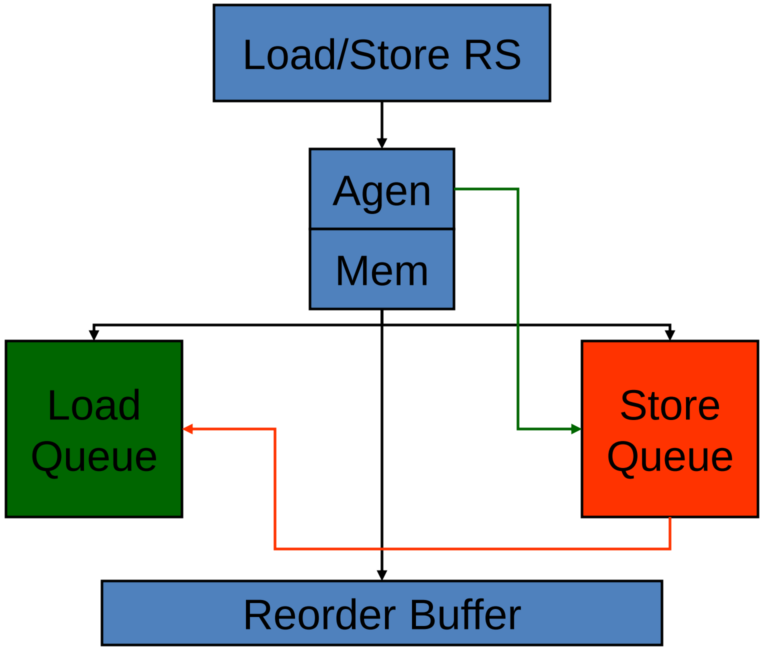
Optimizing Load/Store Disambiguation

- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match
- (1) can limit performance:

```
load r5,MEM[r3]      ← cache miss
store r7, MEM[r5]    ← RAW for agen, stalled
…
load r8, MEM[r9]     ← independent load stalled
```

Speculative Disambiguation

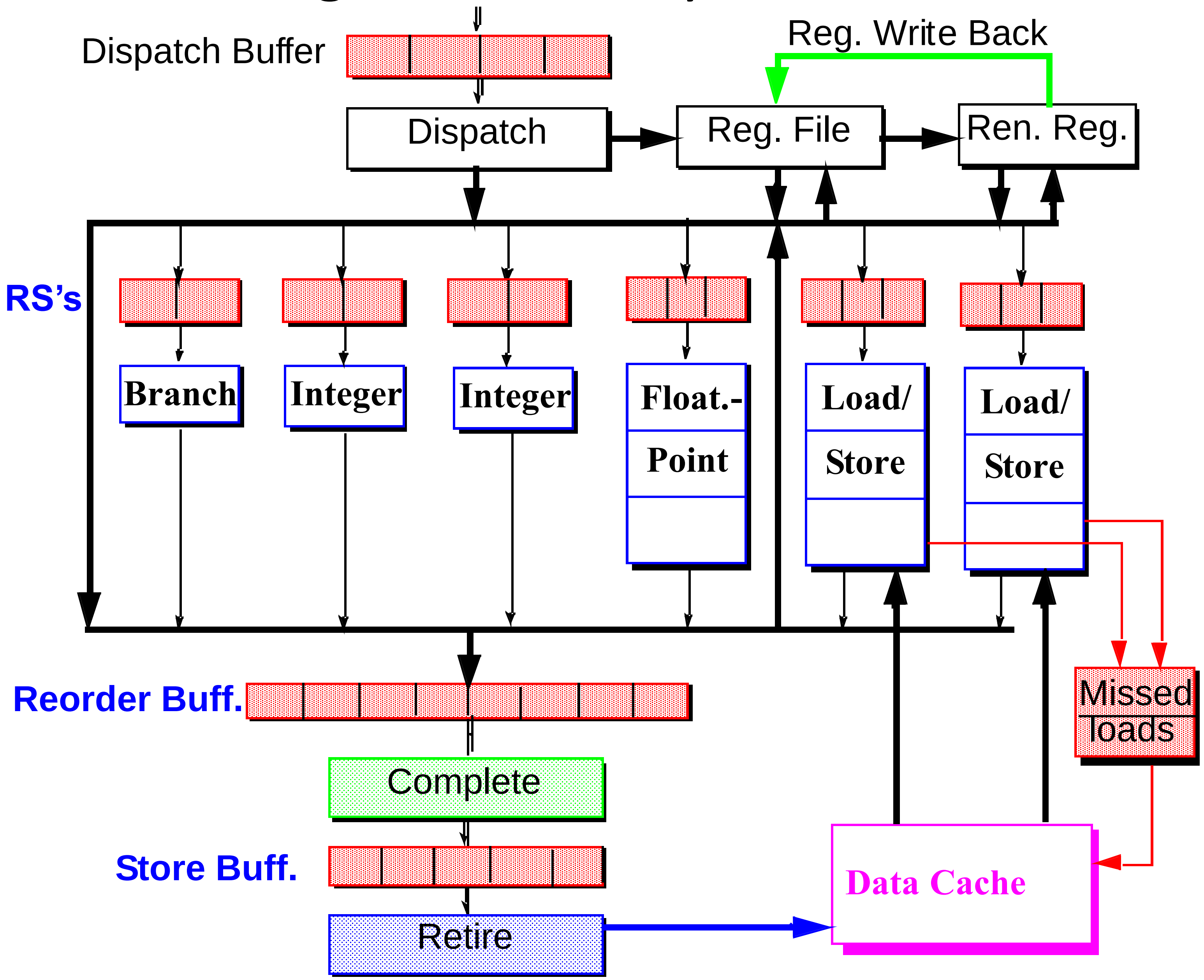
- What if aliases are rare?
  1. Loads don't wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching unbypassed load – incorrect speculation
  5. Replay starting from incorrect load



Use of Prediction

- If aliases are rare: static prediction
  - Predict no alias every time
    - Why even implement forwarding? PowerPC 620 doesn’t
  - Pay misprediction penalty rarely
- If aliases are more frequent: dynamic prediction
  - Use PHT-like history table for loads
    - If alias predicted: delay load
    - If aliased pair predicted: forward from store to load
      - More difficult to predict pair [store sets, Alpha 21264]
  - Pay misprediction penalty rarely
- Memory cloaking [Moshovos, Sohi]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

Easing The Memory Bottleneck



Memory Bottleneck Techniques

Dynamic Hardware (Microarchitecture):

Use Non-blocking D-cache (need missed-load buffers)

Use Multiple Load/Store Units (need multiported D-cache)

Use More Advanced Caches (victim cache, stream buffer)

Use Hardware Prefetching (need load history and stride detection)

Static Software (Code Transformation):

Insert Prefetch or Cache-Touch Instructions (mask miss penalty)

Array Blocking Based on Cache Organization (minimize misses)

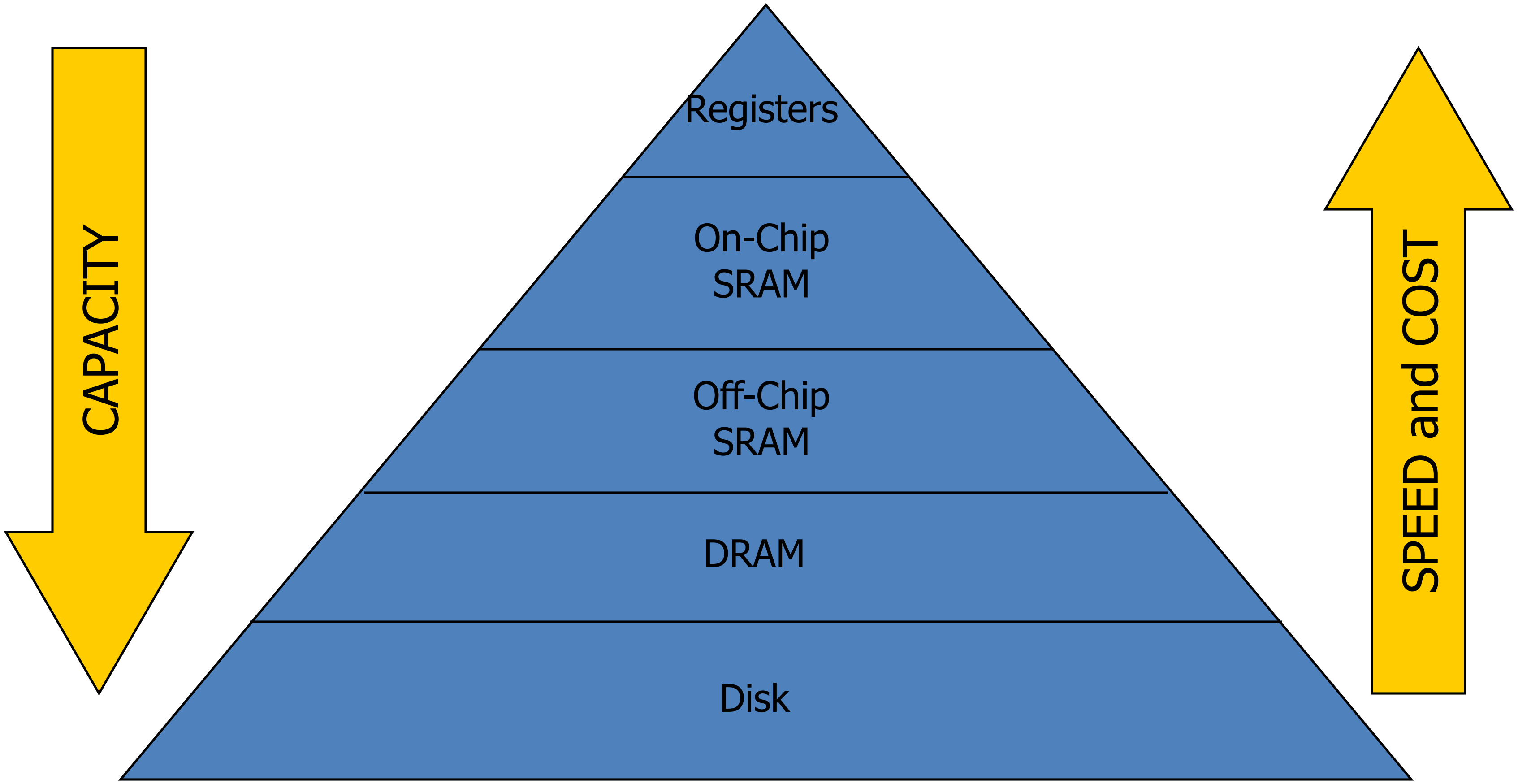
Reduce Unnecessary Load/Store Instructions (redundant loads)

Software Controlled Memory Hierarchy (expose it to above DSI)

Memory Hierarchy

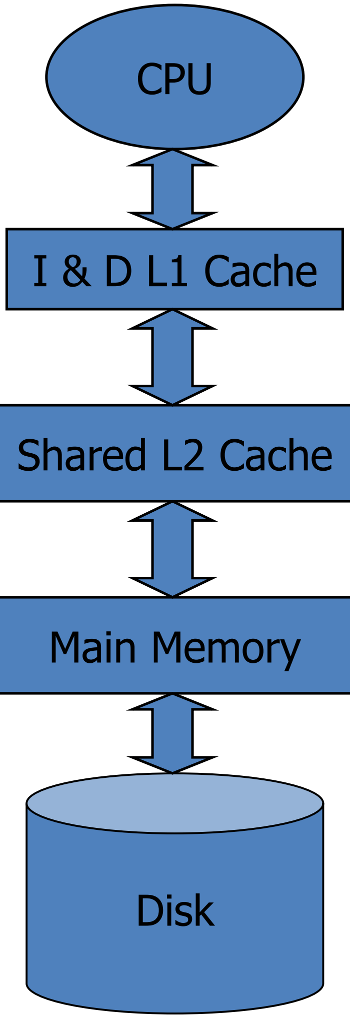
- Memory
  - Just an “ocean of bits”
  - Many technologies are available
- Key issues
  - Technology (how bits are stored)
  - Placement (where bits are stored)
  - Identification (finding the right bits)
  - Replacement (finding space for new bits)
  - Write policy (propagating changes to bits)
- Must answer these regardless of memory type

Memory Hierarchy



Memory Hierarchy

Temporal Locality
- Keep recently referenced items at higher levels
- Future references satisfied quickly

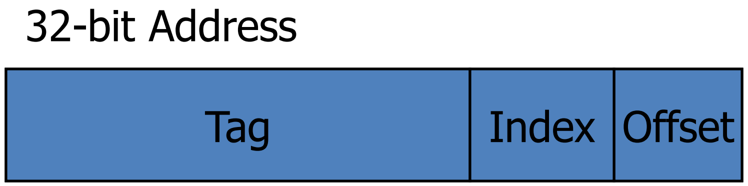


Spatial Locality
- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly

Four Burning Questions

- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?
- Consider these for caches
  - Usually SRAM
- Will consider main memory, disks later

Placement and Identification



Portion	Length	Purpose
Offset	$o=\log_2(\text{block size})$	Select word within block
Index	$i=\log_2(\text{number of sets})$	Select set of blocks
Tag	$t=32 - o - i$	ID block within set

- Consider: <BS=block size, S=sets, B=blocks>
  - <64,64,64>: o=6, i=6, t=20: direct-mapped (S=B)
  - <64,16,64>: o=6, i=4, t=22: 4-way S-A (S = B / 4)
  - <64,1,64>: o=6, i=0, t=26: fully associative (S=1)
- Total size = BS x B = BS x S x (B/S)

Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where a = *associativity*
  - If a <= 2, LRU is cheap and easy (1 bit)
  - If a > 2, it gets harder
  - Pseudo-random works pretty well for caches

Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a *writeback* or *castout*

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

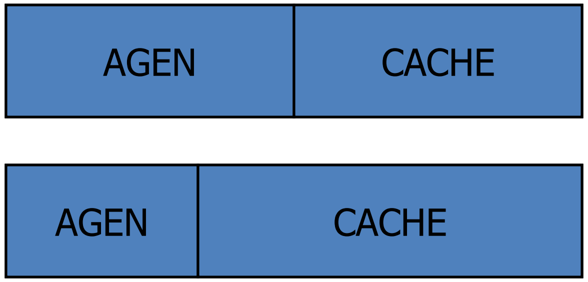
Caches and Performance

- Caches
  - Enable design for common case: cache hit
    - Cycle time, pipeline organization
    - Recovery policy
  - Uncommon case: cache miss
    - Fetch from next level
      - Apply recursively if multiple levels
    - What to do in the meantime?
- What is performance impact?
- Various optimizations are possible

Performance Impact

- Cache hit latency
  - Included in “pipeline” portion of CPI
    - E.g. IBM study: 1.15 CPI with 100% cache hits
  - Typically 1-3 cycles for L1 cache
    - Intel/HP McKinley: 1 cycle
      - Heroic array design
      - No address generation: load r1, (r2)
    - IBM Power4: 3 cycles
      - Address generation
      - Array access
      - Word select and align
      - Register file write (no bypass)

Cache Hit continued



- Cycle stealing common
  - Address generation < cycle
  - Array access > cycle
  - Clean, FSD cycle boundaries violated
- Speculation rampant
  - “Predict” cache hit
  - Don’t wait for tag check
  - Consume fetched word in pipeline
  - Recover/flush when miss is detected
    - Reportedly 7 (!) cycles later in Pentium 4

Replacement Recap

- Replacement policies affect *capacity* and *conflict* misses
- Policies covered:
  - Belady's optimal replacement
  - Least-recently used (LRU)
  - Practical pseudo-LRU (tree LRU)
  - Protected LRU
    - LIP/DIP variant
    - *Set dueling* to dynamically select policy
  - Not-recently-used (NRU) or *clock* algorithm
  - RRIP (re-reference interval prediction)
  - Least frequently used (LFU)
- Championship contests

Prefetching Recap

- Prefetching anticipates future memory references
  - Software prefetching
  - Next-block, stride prefetching
  - Global history buffer prefetching
- Issues/challenges
  - Accuracy
  - Timeliness
  - Overhead (bandwidth)
  - Conflicts (displace useful data)

Memory Data Flow

- Memory Data Flow Challenges
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- Cache Hits and Cache Misses
- Replacement Policies
- Prefetching

Pentium Pro Case Study

- **Microarchitecture**
  - Order-3 Superscalar
  - Out-of-Order execution
  - Speculative execution
  - In-order completion
- **Design Methodology**
- **Performance Analysis**

Review Summary

- Performance
- Program dependences
- Pipelining
- Technology challenges
- Superscalar
  - Instruction flow: fetch alignment, branch prediction, decoding
  - Register data flow: renaming, issuing
  - Memory data flow: load bypassing, forwarding, speculation, speculative disambiguation
- Basic memory hierarchy
- Pentium Pro case study

Readings 1

- Introduction & Technology Challenges & Pipelining
  - Shen & Lipasti, Chapter 1 and Chapter 2.
  - 2015 ITRS Update [PDF] **(other)**. Read Section 1, 5, 6, 8, 9, and skim the rest.
  - Shekhar Borkar, “Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation,” IEEE Micro 2005, November/December 2005 (Vol. 25, No. 6) pp. 10-16.
  - Jacobson, H, et al., “Stretching the limits of clock-gating efficiency in server-class processors,” in Proceedings of HPCA-11, 2005.
  - J. E. Smith. An Analysis of Pipeline Clocking, University of Wisconsin-Madison ECE Unpublished Note, March 1990.

Readings 2

- Superscalar Processors
  - Shen & Lipasti Chapter 4, 5, 9
  - J. E. Smith. A Study of Branch Prediction Strategies, Proceedings of the 8th Annual Symposium on Computer Architecture, pp. 135-148, May 1981 (B4).
  - T-Y. Yeh and Y. Patt. Two-level Adaptive Training Branch Prediction, Proc. 24th Annual International Symposium on Microarchitecture, Nov 1991 (B4).

Readings 3

- Superscalar Processors
  - D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 model 91: Machine Philosophy and Instruction-Handling, IBM Journal of Research and Development, Jan. 1967 (B4).
  - J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors, IEEE Trans. on Computers, May 1988 (B4).
  - Y. N. Patt, W. W. Hwu, and M Shebanow. HPS, a New Microarchitecture: Rationale and introduction, Proceedings of the 18th Workshop on Microprogramming, Pacific Grove, CA, pp. 103-108, Dec. 1985 (B4).

Readings 4

- Superscalar Processors cont’d
  - Gurindar S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers, Proc. 14th Annual Symposium in Computer Architecture, June 1987 (B4)
  - Borch, E., Tune, E., Manne, S., and Emer, J. Loose Loops Sink Chips. In Proceedings of HPCA-8,

Readings 6

- Case Studies
  - Shen/Lipasti Ch. 6-7: read, Ch 8 (skim)
  - G. F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor, IBM Journal of Research and Development, 34(1):37-58, 1990 (B4).
  - Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor, IEEE Micro, April 1996 (B4).
  - K. Czechowski, V. Lee, E. Grochowski, R. Ronen, R. Singhal, R. Vuduc, P. Dubey. Improving the Energy Efficiency of Big Cores. *Proceeding of ISCA-14*, June 2014.